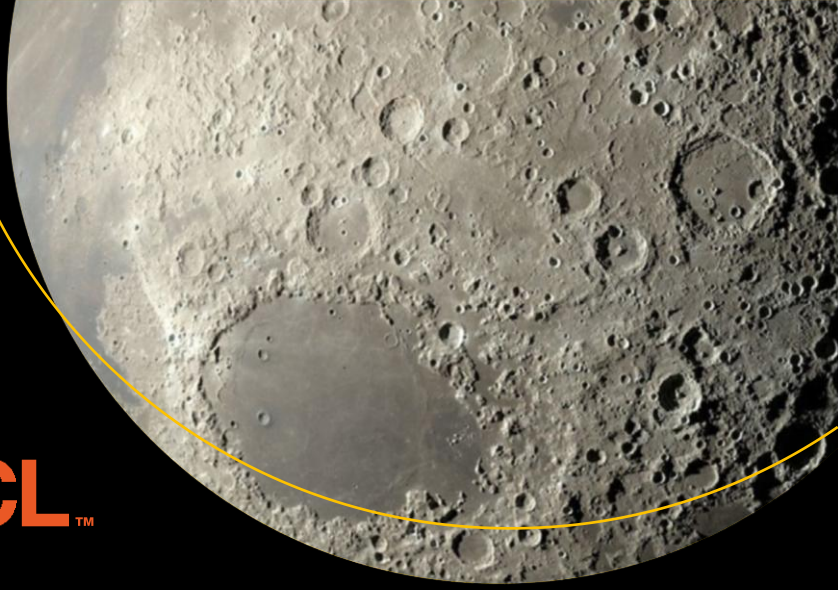

SYCL™

データ並列 C++

C++ と SYCL を使用した高速システムのプログラミング

第 2 版

—

James Reinders
Ben Ashbaugh
James Brodman
Michael Kinsner
John Pennycook
Xinmin Tian

GROMACS およびストックホルム大学の Erik Lindahl 氏による序文

iSUS 編集部訳

Apress open

データ並列 C++

C++ と SYCL を使用した
高速システムのプログラミング

第 2 版

James Reinders
Ben Ashbaugh
James Brodman
Michael Kinsner
John Pennycook
Xinmin Tian

GROMACS およびストックホルム大学の
Erik Lindahl 氏による序文

iSUS 編集部訳

Apress open

データ並列 C++: C++ と SYCL を使用した高速システムのプログラミング、第 2 版

James Reinders
Beaverton、OR、USA

Ben Ashbaugh
Folsom、CA、USA

James Brodman
Marlborough、MA、USA

Michael Kinsner
Halifax, NS, Canada

John Pennycook
San Jose、CA、USA

Xinmin Tian
Fremont、CA、USA

ISBN-13 (pbk): 978-1-4842-9690-5
https://doi.org/10.1007/978-1-4842-9691-2

ISBN-13 (電子書籍): 978-1-4842-9691-2

Copyright © 2023 by Intel Corporation

この書籍は著作権の対象です。資料の全体または一部に関するすべての権利、具体的には翻訳、再印刷、図解の再利用、朗読、放送、マイクロフィルムまたはその他の物理的な方法での複製、および送信、情報の保存と検索、電子的翻案、コンピューター・ソフトウェア、または現在知られているまたは今後開発される類似または異なる方法による権利は、発行者に帰属します。

オープンアクセス この書籍は、Creative Commons Attribution 4.0 International License (https://creativecommons.org/licenses/by/4.0/) の条項に基づいてライセンスされています。このライセンスでは、元著者と情報源を適切に明記し、クリエイティブ・コモンズ・ライセンスへのリンクを提供し、ライセンスされた資料を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

この書籍に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が書籍のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

本書には商標名、ロゴ、画像が使用されている場合があります。本書では、商標名、ロゴ、画像が登場するたびに商標記号を使用せず、商標を侵害する意図なく、編集上および商標所有者の利益のためにのみ、名前、ロゴ、画像を使用します。

この出版物で使用されている商号、商標、サービスマーク、および類似の用語は、明記されていない場合でも、所有権の対象であるか意見の表明として解釈されるものではありません。

Intel、インテル、Intel ロゴ、Intel Optane、および Xeon は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。OpenCL および OpenCL ロゴは、アメリカ合衆国およびその他の国における Apple Inc. の商標です。OpenMP および OpenMP ロゴは、アメリカ合衆国およびその他の国における OpenMP Architecture Review Board の商標です。SYCL、SYCL ロゴ、Khronos、および Khronos Group ロゴは、Khronos Group Inc. の商標です。オープンソースの DPC++ コンパイラーは、公開されている Khronos SYCL 仕様に基づいています。SYCL 実装の現在の適合状況については、https://www.khronos.org/conformance/adopters/conformant-products/sycl を参照してください。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。詳細については、https://www.intel.com/benchmarks (英語) を参照してください。パフォーマンス結果は 2019 年 5 月 10 日現在の測定値であり、すべての公開済みセキュリティー・アップデートが適用されていない可能性があります。詳細は、システム構成を参照してください。絶対的なセキュリティーを提供できるコンピューター・システムはありません。インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、またはサービスの有効化が必要となる場合があります。実際の性能はシステム構成によって異なります。絶対的なセキュリティーを提供できるコンピューター・システムはありません。詳細については、各システムメーカーまたは販売店にお問い合わせいただくか、http://www.intel.co.jp/ を参照してください。

本書に記載されているアドバイスと情報は、発行日時点では真実かつ正確であると考えられていますが、著者、翻訳者、編集者、出版社は、誤りや欠落があった場合の法的責任を一切負いません。発行者は、ここに含まれる資料に関して、明示的または黙示的を問わず、いかなる保証も行いません。

Apress Media LLC マネージング・ディレクター: Welmoed Spahr Acquisitions
編集者: Susan McDermot
開発編集者: James Markham
コーディネーティング・エディター: Jessica Vakili

Springer Science+Business Media New York, 1 NY Plaza, New York, NY 10004 により世界中の書籍業界に配布されます。電話 1-800-SPRINGER、ファックス (201) 348-4505、e-mail: orders-ny@springer-sbm.com または visit www.springeronline. com。Apress Media, LLC はカリフォルニア州の LLC であり、唯一のメンバー (所有者) は Springer Science + Business Media Finance Inc (SSBM Finance Inc) です。SSBM Finance Inc はデラウェア州の法人です。

翻訳に関する情報については、booktranslations@springernature.com まで電子メールでお問い合わせください。再版、ペーパーバック、またはオーディオの権利については、bookpermissions@springernature.com まで電子メールでお問い合わせください。

Apress のタイトルは、学術的、企業的、または販促的な用途で大量に購入できます。ほとんどのタイトルには電子書籍版とライセンスも用意されています。詳細については、当社の印刷物および電子書籍の大量販売ウェブページ (http://www.apress.com/bulk-sales) を参照してください。

本書で著者が参照しているソースコードやその他の補足資料は、Github リポジトリー (https://github.com/Apress/Data-Parallel-CPP) で入手できます。詳しい情報については、https://www.apress.com/gp/services/source-code をご覧ください。

この製品に使用されている紙はリサイクルできます

目次

著者紹介 iv
はじめに vi
第 2 版 vi
本書の構成 viii
第 1 章から第 4 章: 基礎を築く viii
第 5 章から第 12 章: 基礎の上に築く ix
第 13 章 – 第 21 章: SYCL 実践のためのヒント/アドバイス ix
エピローグ: 将来を推測 ix
序文 x
謝辞 xiv

第 1 章 概要 16
仕様書ではなく本書を参照 2
SYCL 2020 と DPC++ 3
なぜ CUDA ではないのか？ 4
標準 C++ と SYCL である理由 5
SYCL をサポートする C++ コンパイラーの入手 5
Hello, World! と SYCL プログラムの分析 6
キューとアクション 7
すべては並列性にかかっている 8
スループット 8
レイテンシー 9
Think Parallel（並列に考える） 9
アムダールとグスタフソン 10
スケーリング 11
異種システム 11
データ並列プログラミング 13
C++ と SYCL の主な特徴 14
単一ソース 14
ホスト 15
デバイス 15
カーネルコード 16
非同期実行 18
ミスによる競合状態 19
デッドロック 22
C++ ラムダ式 23
機能の移植性とパフォーマンスの移植性 26
並行性（Concurrency）と並列性（Parallelism） 28

まとめ 30

第 2 章 コードが実行される場所 31
単一ソース 31
ホストコード 33
デバイスコード 34
デバイスの選択 36
方法 #1: あらゆるタイプのデバイスで実行 37
キュー 37
任意のデバイスが利用可能な場合のキューとデバイスのバインド 41
方法 #2: 開発、デバッグ、デプロイメントに CPU デバイスを使用 42
方法 #3: GPU（または他のアクセラレーター）を使用 45
アクセラレーター・デバイス 46
デバイスセレクター 46
方法 #4: 複数のデバイスを使用 50
方法 #5: カスタム（非常に特定の）デバイス選択 51
デバイスアスペクトに基づく選択 51
カスタムセレクターによる選択 53
デバイス上でワークを作成 54
タスクグラフ 54
デバイスコードはどこにあるか？ 56
アクション 60
ホストタスク 63
まとめ 65

第 3 章 データ管理 67
はじめに 68
データ移動の問題 69
デバイスローカルとデバイスリモート 69
複数のメモリーを管理 70
明示的なデータ移動 70
暗黙的なデータ移動 71
適切な方法の選択 71
USM、バッファーおよびイメージ 72
統合共有メモリー 72
ポインターを介してメモリーにアクセス 73
USM とデータ移動 74
バッファー 77
バッファーを作成 78
バッファーのアクセス 78
アクセスモード 80

データの使用を順序付け 80
インオーダー・キュー 83
アウトオブオーダー・キュー 84
データ管理方法の選択 92
ハンドラークラス: 主要メンバー 93
まとめ 96

第 4 章 並列処理の表現 97
カーネル内の並列処理 98
ループとカーネル 99
多次元カーネル 101
言語機能の概要 102
カーネルとホストコードの分離 102
並列カーネルの異なる形式 103
基本データ並列カーネル 105
基本データ並列カーネルを理解する 105
基本データ並列カーネルを記述 107
基本データ並列カーネルの詳細 109
明示的な ND-Range カーネル 112
明示的な ND-Range 並列カーネルを理解する 113
明示的な ND-Range データ並列カーネルを記述 121
明示的な ND-Range データ並列カーネルの詳細 122
sub_group クラス 126
計算をワーク項目へマッピング 127
1 対 1 マッピング 128
多対 1 マッピング 128
カーネル形式の選択 130
まとめ 132

第 5 章 エラー処理 135
安全が第一 135
エラーのタイプ 136
エラーを発生させてみましょう! 138
同期エラー 139
非同期エラー 139
アプリケーションエラー処理方法 140
エラー処理の無視 141
同期エラー処理 143
非同期エラー処理 144
非同期ハンドラー 145
ハンドラーの起動 148

デバイス上のエラー.....149
まとめ.....150

第 6 章 統合共有メモリー.....153
USM を使用する理由は?.....153
割り当てタイプ.....154
デバイス割り当て.....154
ホスト割り当て.....155
共有割り当て.....155
メモリー割り当て.....156
何を知る必要があるか?.....156
複数のスタイル.....157
メモリーを解放.....164
割り当ての例.....165
データ管理.....165
初期化.....165
データ移動.....166
クエリー.....174
あと一つだけ.....177
まとめ.....178

第 7 章 バッファー.....179
バッファー.....180
バッファーを作成.....181
バッファーで何ができるでしょう.....188
アクセサー.....189
アクセサーを作成.....192
アクセサーで何ができるでしょう.....198
まとめ.....199

第 8 章 カーネルのスケジュールとデータ移動.....201
グラフのスケジュールとは.....202
SYCL におけるグラフの仕組み.....202
コマンドグループのアクション.....203
コマンドグループが依存関係を宣言する方法.....203
例.....204
コマンドグループの各部分はいつ実行されるか.....213
データ移動.....213
明示的なデータ移動.....213
暗黙的なデータ移動.....214
ホストとの同期.....216

まとめ 218

第 9 章　通信と同期 221
ワークグループとワーク項目 221
効率的な通信のための構成要素 223
同期とバリア 223
ワークグループ・ローカル・メモリー 225
ワークグループ・バリアとローカルメモリーの使用 227
ND-range カーネルのワークグループ・バリアとローカルメモリー 231
同期関数 232
サブグループ 235
同期とサブグループ・バリア 236
サブグループ内でのデータ交換 237
完全なサブグループ ND-Range カーネルの例 239
グループ関数とグループ・アルゴリズム 241
ブロードキャスト 241
投票 242
シャッフル 243
まとめ 246
第 10 章　カーネルの定義 249
カーネルを表現する方法が 3 つあるのはなぜか 249
ラムダ式としてカーネルを表現 251
カーネルラムダ式の要素 251
カーネルラムダ式の識別 254
名前付き関数オブジェクトのカーネル 255
カーネル名前付き関数オブジェクトの要素 256
カーネルバンドル内のカーネル 259
他の API との相互運用性 264
まとめ 264

第 11 章　ベクトルと数学配列 267
ベクトル型の曖昧さ 268
SYCL ベクトル型に関するメンタルモデル 269
数学配列（marray） 271
ベクトル（vec） 273
ロードとストア 274
バックエンド・ネイティブ・ベクトル型との相互運用性 276
スウィズル操作 276
ベクトル型の実行方法 280
利便性のある型としてのベクトル 280
SIMD 型としてのベクトル 284

まとめ 286

第 12 章 デバイス情報とカーネルの特殊化 289
GPU が存在するか? 290
カーネルコードをより規範的なものに改良 291
デバイスと機能を列挙する方法 293
アスペクト 296
カスタム・デバイス・セレクター 298
興味を持ちましょう: get_info<> 300
好奇心を持ちましょう: 詳細な列挙コード 301
非常に興味深い: get_info と has() 303
デバイス情報記述子 303
デバイス固有のカーネル情報記述子 303
詳細: 正当性 304
デバイスクエリー 305
カーネルの照会 306
詳細: チューニング/最適化 307
デバイスクエリー 307
カーネルの照会 308
実行時プロパティーとコンパイル時プロパティー 308
カーネルの特殊化 309
まとめ 312

第 13 章 実用的なヒント 313
サンプルコードとコンパイラーの入手 313
オンラインリソース 313
プラットフォーム・モデル 314
複数アーキテクチャー・バイナリー 315
コンパイルモデル 316
コンテキスト: 知っておくべき重要なこと 319
既存の C++ プログラムへの SYCL の追加 321
複数のコンパイラーを使用する場合の考慮事項 322
デバッグ 323
デッドロックや他の同期の問題のデバッグ 325
カーネルコードのデバッグ 326
ランタイム機能のデバッグ 327
キューのプロファイルと結果のタイミング機能 330
トレースおよびプロファイル・ツールのインターフェイス 334
データの初期化とカーネル出力へのアクセス 335
複数の翻訳単位 344
複数の翻訳単位のパフォーマンスへの影響 345

匿名ラムダ式に名前が必要な場合 345
まとめ 346

第 14 章 一般的な並列パターン 349
パターンを理解する 350
マップ 351
ステンシル 352
リダクション 354
スキャン 356
パックとアンパック 358
ビルトイン関数とライブラリーの使用 360
SYCL リダクション・ライブラリー 360
グループ・アルゴリズム 366
ダイレクト・プログラミング 370
マップ 370
ステンシル 371
リダクション 373
スキャン 374
パックとアンパック 377
まとめ 380
関連情報 381

第 15 章 GPU 向けのプログラミング 383
パフォーマンスに関する注意事項 383
GPU の仕組み 384
GPU ビルディング・ブロック 384
よりシンプルなプロセッサー（ただし、数は多い） 386
簡略化された制御ロジック（SIMD 命令） 391
ワークの切り替えとレイテンシーの隠匿 398
カーネルを GPU へオフロード 400
SYCL ランタイム・ライブラリー 400
GPU ソフトウェア・ドライバー 401
GPU ハードウェア 402
オフロードのコストに注意 403
GPU カーネルのベスト・プラクティス 405
グローバルメモリーのアクセス 405
ワークグループ・ローカル・メモリーのアクセス 409
サブグループを使用してローカルメモリーを完全に回避 412
小さなデータ型を使用した計算の最適化 412
数学関数の最適化 413
特殊機能と拡張機能 414

まとめ 414
関連情報 415

第 16 章 CPU 向けのプログラミング 417
パフォーマンスに関する注意事項 418
マルチコア CPU の基本 419
SIMD ハードウェアの基本 422
スレッドレベルの並列処理の活用 428
スレッド・アフィニティーの調査 431
メモリーへのファーストタッチに注意 435
CPU 上の SIMD ベクトル化 436
SIMD 実行の合法性を確保 437
SIMD マスクとコスト 440
SIMD 効率のために構造体配列を避ける 442
データ型が SIMD 効率に与える影響 444
single_task を使用した SIMD 実行 446
まとめ 448

第 17 章 FPGA 向けのプログラミング 451
パフォーマンスに関する注意点 452
FPGA の考え方 452
パイプラインの並列処理 456
カーネルはチップの「領域」を消費 459
FPGA を使用するタイミング 460
たくさんのワーク 460
カスタム操作または操作幅 461
スカラー・データ・フロー 462
低いレイテンシーと豊富な接続性 463
カスタム・メモリー・システム 464
FPGA で実行 465
コンパイル時 467
FPGA エミュレーター 469
FPGA ハードウェアのコンパイルは「事前に」行われる 470
FPGA 向けにカーネルを記述 471
並列処理の可視化 472
ND-Range を使用してパイプラインを常に稼働状態に保つ 475
パイプラインはデータ依存関係を気にしません 478
空間パイプラインによるループの実装 481
ループ開始間隔 483
パイプ 489
カスタム・メモリー・システム 495

最後に 498
FPGA ビルディング・ブロック 498
クロック周波数 500
まとめ 501

第 18 章 ライブラリー 503
ビルトイン関数 504
sycl:: を使用ビルトイン関数のプリフィクス 506
C++ 標準ライブラリー 507
oneAPI DPC++ ライブラリー（oneDPL）..... 510
SYCL 実行ポリシー 511
バッファーで oneDPL を使用 513
USM で oneDPL を使用 517
SYCL 実行ポリシーによるエラー処理 519
まとめ 520

第 19 章 メモリーモデルとアトミック 523
メモリーモデルに含まれるものは？ 525
データ競合と同期 526
バリアとフェンス 529
アトミック操作 531
メモリー順序 532
メモリーモデル 534
memory_order 列挙クラス 536
memory_scope 列挙クラス 538
デバイス機能の照会 540
バリアとフェンス 542
SYCL におけるアトミック操作 543
バッファーでアトミックを使用 548
統合共有メモリーでアトミック操作を使用 550
アトミックを活用 550
ヒストグラムの計算 551
デバイス全体の同期の実装 553
まとめ 556
関連情報 557

第 20 章 バックエンドの相互運用性 559
バックエンドの相互運用性とは 559
バックエンドの相互運用性はいつ役立つか 561
既存のコードベースへの SYCL の追加 562
SYCL で既存のライブラリーを使用する 564

カーネルのバックエンド相互運用性の使用 569
API 定義カーネル・オブジェクトとの相互運用性 569
SYCL 以外のソース言語との相互運用性 571
バックエンドの相互運用性に関するヒントとコツ 574
特定のバックエンド用のデバイスの選択 574
コンテキストに注意してください 576
低レベル API 固有の機能にアクセスする 576
他のバックエンドのサポート 576
まとめ 577

第 21 章 CUDA コードの移行 579
CUDA と SYCL の設計上の違い 579
複数のターゲットと単一のデバイスターゲット 579
C++ への準拠と C++ の拡張 581
CUDA と SYCL の用語の違い 582
類似点と相違点 583
実行モデル 584
メモリーモデル 589
その他の違い 592
SYCL にはまだない CUDA の機能 595
グローバル変数 595
協調グループ 596
行列乗算ハードウェア 597
移植ツールとテクニック 598
dpct と SYCLomatic を使用したコードの移行 598
まとめ 603
関連情報 604

エピローグ SYCL の将来の方向性 605
C++11、C++14、C++17 とのより緊密な連携 606
C++20、C++23 以降の機能の採用 607
SPMD と SIMD プログラミングの混在 608
アドレス空間 610
特殊化メカニズム 611
コンパイル時のプロパティー 611
まとめ 612
関連情報 613

訳者あとがき 614
インデックス 615

著者紹介

James Reinders は、並列コンピューティングの分野で 30 年以上の経験を持つ Intel Corporation のエンジニアであり、並列プログラミングに関する 10 冊以上の技術書籍の著作、共著、編集に携わっています。James はシステムの最適化と教育に情熱を持っています。James は、TOP500 リストで 1 位を獲得した世界最速のいくつかのコンピューター・システムをはじめ、数多くのスーパーコンピューターやソフトウェア開発ツールに重要な貢献を果たすという幸運に恵まれました。

Ben Ashbaugh は、Intel Corporation のソフトウェア・アーキテクトであり、20 年以上にわたりインテル・グラフィックス製品向けのソフトウェア・ドライバーとコンパイラーの開発に携わってきました。過去 10 年間、Ben は SYCL や DPC++ コンパイラーを含む、グラフィックス・プロセッサー向けの汎用計算の並列プログラミング・モデルに注目してきました。Ben は、Khronos SYCL、OpenCL、SPIR ワーキンググループで積極的に活動し、並列プログラミングの業界標準の定義に貢献しています。また、独自のインテル GPU 機能を利用する多数の拡張機能も作成しています。

James Brodman は、Intel Corporation のプリンシパル・エンジニアであり、並列プログラミングのランタイムとコンパイラー開発に取り組んでおり、DPC++ の設計者の 1 人です。James は、イリノイ大学アーバナ・シャンペーン校でコンピューター・サイエンスの博士号を取得しています。

Michael Kinsner は、Intel Corporation のプリンシパル・エンジニアであり、さまざまなアーキテクチャー向けの並列プログラミング言語とコンパイラーを開発しています。Michael は、空間アーキテクチャーとプログラミング・モデルに幅広く貢献しており、Khronos グループ内の Intel 代表として並列プログラミングの SYCL および OpenCL の業界標準に取り組んでいます。Mike はマクマスター大学でコンピューター・エンジニアリングの博士号を取得しており、パフォーマンスを維持しながら複数のアーキテクチャーに対応するプログラミング・モデルに熱心に取り組んでいます。

John Pennycook は、Intel Corporation のソフトウェア・イネーブリングおよび最適化アーキテクトであり、開発者が最新プロセッサーで利用可能な並列処理能力を最大限に活用できるようにすることに注力しています。John は、さまざまな科学分野のアプリケーションの最適化と並列化の経験があり、以前は Intel eXtreme Performance User's Group (IXPUG) の運営委員会で Intel の代表を務めていました。
John は、ウォーリック大学でコンピューター・サイエンスの博士号を取得しています。彼の研究対象は多岐にわたりますが、繰り返し取り上げられているテーマは、異なるハードウェア・アーキテクチャー間でアプリケーションの「パフォーマンスの移植性」を実現する能力です。

Xinmin Tian は、Intel Corporation のフェロー兼コンパイラー・アーキテクトであり、OpenMP Architecture Review Board (ARB) で Intel の代表を務めています。Xinmin は、インテル・アーキテクチャー向けの OpenMP オフロード、ベクトル化、および並列コンパイラー・テクノロジーを推進しています。彼は現在、LLVM ベースの OpenMP オフロード、CPU/GPU 向けの SYCL/DPC++ コンパイラーの最適化、および HPC/AI アプリケーションのパフォーマンス・チューニングに重点を置いています。彼は清華大学でコンピューター・サイエンスの博士号を取得しており、27 件の米国特許を保有し、60 本以上の技術論文を発表し、その論文は 1300 回以上引用されています。また、専門知識を網羅した 2 冊の本の共著者でもあります。

はじめに

本書は、並列プログラミングを初めて行う読者にも適しています。また、SYCL や DPC++ コンパイラーについて知識がなくても大丈夫です。

CUDA でのプログラミングと比較すると、SYCL を用いた C++ は、NVIDIA 以外のプラットフォームへの移植性を提供し、GPU 以外のデバイスへの移植性も実現します。さらに、進化を続ける現代の C++ に緊密に連携し、その強化を図ります。C++ と SYCL は、性能を犠牲にすることなくこれらの利点をもたらします。

SYCL を使用した C++ を使用すると、特定のベンダーに縛られることなく、CPU、GPU、FPGA、および将来の処理デバイスを組み合わせた機能を活用して、アプリケーションを高速化できます。

SYCL は、加速された（異種）システムを活用するため、C++ によるデータ並列処理の高度なサポートを追加する業界主導の Khronos 標準です。SYCL は、C++ および C++ ビルドシステムと高度な相乗効果を発揮する C++ コンパイラーのメカニズムを提供します。DPC++ は、SYCL のサポートを追加した LLVM に基づくオープンソース・コンパイラー・プロジェクトです。本書のすべての例は、DPC++ コンパイラーを含む SYCL 2020 をサポートする任意の C++ コンパイラーで動作するはずです。

C++ に精通していない C プログラマーであれば、あなたも同じような状況に陥るでしょう。本書の著者の何人かは、本書のような C++ を使用した書籍を読むことで C++ を習得したことを喜んで認めている。少しの忍耐があれば、最新の C++ プログラムを書きたいと考えている C プログラマーは、本書を読みこなせるようになるはずです。

第 2 版

拡大を続ける SYCL ユーザーのコミュニティーからのフィードバックにより、SYCL をこれまで以上に効果的に学習できるコンテンツを追加することができました。

第 2 版では、SYCL 2020 を使用して C++ を学習します。本書の最初の版は、SYCL 2020 仕様に先行して出版されていますが、第 2 版はその内容とわずかに異なるだけです（第 2 版の SYCL 2020 の最も明らかな変更点は、ヘッダーファイルの場所、デバイスセレクターの構文、および明示的な*ホストデバイス*の廃止です）。

既知の書籍の正誤表を含む、最新の SYCL の重要なリソースには、書籍の Github（`https://github.com/Apress/data-parallel-CPP`）（英語）、Khronos Group の SYCL 標準 Web サイト（`www.khronos.org/sycl`）（英語）、および主要な SYCL 教育 Web サイト（`https://sycl.tech`）（英語）などがあります。

第 20 章と第 21 章は、この本の初版を読んだ読者からの要望により追加されたものです。

バックエンドの相互運用性について説明するため、第 20 章を追加しました。SYCL 2020 標準の主要な目標の 1 つは、さまざまなアーキテクチャーを備えた多くのベンダーのハードウェアを幅広くサポートすることです。これには、SYCL 1.2.1 の OpenCL バックエンドのサポートを超えて拡張する必要がありました。一般的には“問題なく動作する”しますが、第 20 章ではこのレベルでの理解と操作を重要視する読者向けに、より詳しく説明しています。

経験豊富な CUDA プログラマー向けに、アプローチと語彙の両面で *C++ と SYCL* の概念を *CUDA* の概念に明示的に結び付けるため、第 21 章を追加しました。異種並列処理を表現する核心的な課題は基本的に同じですが、C++ と SYCL の組み合わせは、マルチベンダーかつマルチアーキテクチャーへのアプローチにより多くの利点をもたらします。第 21 章は、CUDA 用語について言及している唯一のセクションです。本書の残りの部分では、オープンなマルチベンダー、マルチアーキテクチャーのアプローチによる C++ および SYCL 用語について説明します。第 21 章では、CUDA コードの移行を自動化するのに役立つオープンソース・ツールである“SYCLomatic”（`github.com/oneapi-src/SYCLomatic`）を参照することを強く推奨します。

これは便利なツールであり、コード移行の最初のステップとして推奨されます。SYCL を使用した C++ を使用する開発者からは、CUDA から移植されたコードと SYCL を使用したオリジナルの C++ コードの両方で、NVIDIA、AMD、およびインテル GPU で優れた結果が得られたことが報告されています。最終的に得られる SYCL を使用した C++ コードは、NVIDIA CUDA では不可能な移植性を実現します。

C++、SYCL 、DPC++ を含むコンパイラーの進化は続きます。SYCL を使用して C++ の異種システム向けのプログラムを作成する方法を学習した後、エピローグで将来の展望について説明します。

私たちの書籍が SYCL コミュニティーをサポートし、成長させ、SYCL と C++ によるデータ並列プログラミングの促進に役立つことを願っています。

本書の構成

本書では、SYCL と C++ を使用して高速化された異種システムの効果的なプログラマーになるために必要な知識について説明します。

第 1 章から第 4 章: 基礎を築く

C++ と SYCL に初めて取り組む読者は、第 1 章から第 4 章を順番に読むことが重要です。

第 1 章では、新たに学ぶべき、あるいは改めて理解を深める価値のある中心概念を取り上げ、最初の基礎を築きます。

第 2 章から第 4 章では、C++ と SYCL のデータ並列プログラミングを理解する基礎を築きます。第 1 章から第 4 章までを読み終えると、C++ を使用したデータ並列プログラミングのための強固な基盤が身につきます。第 1 章から第 4 章は相互に関連しているので、順番に読むことをお勧めします。

第 5 章から第 12 章: 基礎の上に築く

第 5 章から第 12 章までは、ある程度相互に関連しながら重要な詳細を補完しており、必要に応じて各章を自由に読み飛ばすこともできます。これらの章はすべて、SYCL を使用する C++ のすべてのユーザーにとって価値のあるものとなるはずです。

第 13 章 – 第 21 章: SYCL 実践のためのヒント/アドバイス

最後の章では、特定のニーズに応じたアドバイスと詳細を示します。少なくともすべてにざっと目を通し、ニーズに合う重要なコンテンツを見つけてください。

エピローグ: 将来を推測

本書は、SYCL を使用した C++ の今後の方向性と、SYCL 向けデータ並列 C++ コンパイラーについて説明するエピローグで締めくくります。

SYCL と C++ の使い方を学習していただければ幸いです。

序文

SYCL 2020 は並列コンピューティングにおけるマイルストーンです。初めて、あらゆる種類のハードウェアを対象とする、最新の安定した完全な移植可能なオープン・スタンダードが実現されました。本書は、SYCL 2020 を学習するための最高の手引き書です。

コンピューター・ハードウェアの開発は、大規模で複雑な問題を解決したいというニーズによって推進されますが、皆さんや私たちのようなプログラマーが、自分たちのアイデアを実装し、妥当な労力で利用可能なパワーを活用できる言語を持っていなければ、こうしたハードウェアの進歩はほとんど役に立ちません。驚くべきハードウェアの例は数多くありますが、それらを活用する最初のソリューションは、標準化委員会による合意形成に時間を割かずに済むため、しばしば独自のものであることがありました。しかし、コンピューティングの歴史において、最終的には、オープン・ソフトウェア・スタンダードがハードウェアの競争を促進することは言うまでもなく、世界中のコミュニティーとエコシステムのリソースが個々のベンダーよりもはるかに大きいため、開発者があらゆるハードウェアを対象としてコードを共有できるオープン・スタンダードと競争できず、ベンダーロックインに陥る結果となっています。

過去数年間、私のチームは、世界で最も広く使用されている科学 HPC コードの 1 つである GROMACS の開発を通じて、新興の SYCL エコシステムの形成に貢献するという大きな特権を得てきました。 私たちのコードは、ラップトップだけでなく、世界中のあらゆるスーパーコンピューターで実行できる必要があります。パフォーマンスを失う余裕はありませんが、私たちが依存しているライブラリーに他のチームが労力を注ぎ、オープン・コンパイラーが利用可能で、優秀な人材を採用できる、より大きなコミュニティーの一員であることも重要です。本書の初版以来、SYCL はコミュニティーに成長しました。

ベンダーが提供するいくつかのコンパイラーに加えて、現在ではすべてのハードウェアを対象とする主要なコミュニティー主導の実装[1] があり、世界中で何千人もの開発者が経験を共有し、トレーニング・イベントに貢献し、フォーラムに参加しています。オープンソースの優れた点は、それがアプリケーション、コンパイラー、オープン・スタンダードのいずれであっても、その内部を参照して学習し、借用し、拡張できることです。私たちがインテル主導の LLVM 実装[2] のコード、ハイデルベルグ大学のコミュニティー主導の実装、およびその他のいくつかのコードから繰り返し学んでいるのと同じように、私たちのパブリック・リポジトリー[3] を使用して、大規模な運用環境のコードベースで CUDA と SYCL の実装を比較したり、ニーズに合わせてソリューションを参照することができます。そうすることで、私たちのコミュニティーのさらなる拡大に貢献することになるからです。

意外かもしれませんが、データ並列プログラミングというパラダイムは、メッセージ・パッシング通信や明示的なマルチスレッドなど従来のソリューションよりもはるかに簡単ですが、ハードウェアと明示的なデータ配置に重点を置いた古いパラダイムで何十年も過ごしてきた私たちにとっては、特別な課題を与えられます。小規模であれば、少数のプロセス間でデータをどのように移動させるか明示的に決定するのは簡単なことでしたが、問題が数千のユニットにまで拡大すると、バグが混入したり、ハードウェアがアイドル状態でデータを待機しないような複雑な管理には悪夢が伴います。SYCL を使用したデータ並列プログラミングでは、*アルゴリズム固有の並列処理を明示的に表現*することで、この問題を解決します。その表現が完了すれば、コンパイラーとドライバーがデータ局所性と数万もの機能ユニットにわたるスケジュールをほぼ処理してくれます。データ並列プログラミングを成功させるには、コンピューターを 1 つのプログラムを実行する単一のユニットとして考えるのではなく、大きな問題の各部分を解決する独立して動作するユニットの集合として考えることが重要です。

---

[1] ハイデルベルク大学によるコミュニティー主導の実装: tinyurl.com/HeidelbergSYCL

[2] DPC++ コンパイラー・プロジェクト: github.com/intel/llvm

[3] GROMACS: gitlab.com/gromacs/gromacs/

問題を、各部分が他の部分に依存しないアルゴリズムとして表現できる限り、たとえばデバイスキューを介して GPU 上で実行される並列 for ループとして実装するのは理論的には容易です。ただし、実用的な例では、問題がデバイス全体を効率的に使用できるほど大きくないことが多く、あるいは毎秒数万回の反復処理を必要とする場面では、デバイスドライバーのレイテンシーが主要なボトルネックとなり始めます。本書は、パフォーマンス・ポータブルな GPU プログラミング向けの優れた入門書ですが、それにとどまらず、実際のアプリケーションでスループットとレイテンシーの両方がいかに重要であるか、SYCL を使用して CPU、GPU、SIMD ユニット、FPGA の独自の機能を活用する方法も示しています。また、優れたパフォーマンスを得るには、それぞれのハードウェア・タイプの特性を理解し、必要に応じてコードを調整する必要があるという注意事項も取り上げています。これは単にデータ並列プログラミングに関する優れたチュートリアルであるだけでなく、現代のコンピューター・ハードウェアのプログラミング全般に興味がある人なら一読すべき信頼できるテキストにもなります。

SYCL の強みの 1 つは、最新の C++ との緊密な連携です。 これは一見困難に思えるかもしれません。C++ を完全に習得するのは、容易な言語ではありません（私の場合は確かにそうでした）。しかし、Reinders 氏と共著者は、私たちがほんの少し C++ の概念を学ぶだけで、実際のデータ並列プログラミングに取り組んで生産的になれる道へと導いてくれます。しかし、経験を積むにつれて、SYCL 2020 では C++17 の極めて汎用的な機能と組み合わせて、さまざまなデバイスを動的にターゲットとするコードを記述したり、異なるタスクに CPU、GPU、ネットワーク・ユニットを並列に使用する異種並列処理に依存したりできるようになりました。SYCL は、アクセラレーターを有効にするために別途追加したソリューションではなく、C++ でデータの並列性を表現する一般的な方法となる可能性を秘めています。SYCL 2020 標準には、統合共有メモリー、サブグループ、アトミック操作、リダクション、シンプルなアクセサー、コードをクリーンにする多くの概念など、以前はベンダー拡張機能としてのみ利用可能であったいくつかの機能が含まれており、開発と、多様なハードウェアをターゲットとするコードを作成する標準 C++17、または CUDA からの移植の両方が容易になります。

本書は、それらすべてについて分かり易く書かれた素晴らしい入門書であり、C++ が急速に発展する中で SYCL がどのように進化していくか学ぶこともできます。

理論的には素晴らしいように思えますが、SYCL は実際にはどの程度移植可能でしょう？ 私たちのアプリケーションは、データ・アクセス・パターンがランダムで、各ステップで処理するデータ量が限られており、1 秒あたり何千もの反復処理を実行する必要があり、メモリー帯域幅、浮動小数点、整数演算すべてに制約されるため、最適化が非常に難しいコードベースの例です。これは、単純なデータ並列問題とは正反対です。私たちは 20 年以上にわたって、さまざまな GPU アーキテクチャー向けのアセンブリ SIMD 命令とネイティブ実装を記述してきましたが、SYCL との最初の出会いでは、違いに適応することと、ドライバーおよびコンパイラー開発者にパフォーマンスの低下を報告することの両方で苦労しました。

しかし、2023 年春の時点では、私たちの SYCL カーネルは、単一のコードベースだけでなく、単一のプリコンパイル済みバイナリーからでも、すべての GPU アーキテクチャーでネイティブ・パフォーマンスの 80～100% を達成できています。

SYCL はまだ若く、急速に進化するエコシステムを持っています。言語としてまだ実装されていないいくつかの要素は存在しますが、SYCL は現代のあらゆるハードウェアをターゲットに成功した、唯一のパフォーマンス・ポータブルな標準として特異な存在です。並列プログラミングを学習したい初心者、データ並列プログラミングに興味のある経験豊富な開発者、または 100,000 行の独自の API コードをオープン・スタンダードに移植する必要があるメンテナーであっても、この第 2 版は、このコミュニティーの一員になるために必要な唯一の書籍です。

エリック・リンダール
生物物理学教授
ストックホルム大学
生物物理学・生化学部
生命科学研究所

謝辞

この第 2 版では、コミュニティーから多くのご意見をいただきました。SYCL を製品、クラス、チュートリアル、ワークショップ、カンファレンス、ハッカソンで使用している開発者との交流から、多くのインスピレーションが生まれました。特に、NVIDIA ハードウェアを含む SYCL の展開は、この第 2 版での SYCL 教育における包括性と実用的なヒントを強化するのに役立っています。

SYCL のコミュニティーは大きく成長しており、コンパイラーとツールを実装するエンジニアと、多種多様なベンダーのハードウェアを対象に SYCL を採用する大規模なユーザーグループで構成されています。彼らの努力と共有された知見に感謝しています。

また、非常に機能的な仕様の作成に熱心に取り組んでくれた Khronos SYCL ワーキンググループにも感謝します。特に、Ronan Keryell 氏は SYCL 仕様の編集者であり、長年にわたり SYCL を積極的に支持してきました。

私たちは、SYCL コミュニティーからあらゆる形でフィードバックを寄せてくれた多くの方々に感謝しています。また、数年前に初版の出版に協力してくれた方々にも深く感謝しています。その皆さんは初版の謝辞の中でお名前を記載させていただきました。

初版では GitHub[4] 経由でフィードバックを受け取りました。レビューはしましたが、必ずしもすぐに応答できたわけではありませんでした（6 人の共著者全員が「あなたがしましたよね？」と思ったと想像してください）。私たちはフィードバックから大きな恩恵を受けており、この版のサンプルとテキストですべてのフィードバックに対処できたと考えています。最も熱心にコメントを寄せ、私たちを助けてくれたのは Jay Norwood でした。著者一同、Jay に心から感謝しています。

[4] github.com/apress/data-parallel-CPP

そのほかのフィードバック提供者には、Oscar Barenys、Marcel Breyer、Jeff Donner、Laurence Field、Michael Firth、Piotr Fusik、Vincent Mierlak、Jason Mooneyham がいます。ここにあなたの名前があるかどうかに関わらず、フィードバックを提供し、SYCL を使用した C++ の指導の改善に協力してくれたすべての人に感謝します。

この版では、ごく少数の方々が原稿の下書きを精力的に読み、貴重な意見をくれたことに心から感謝しています。査読者には、Aharon Abramson、Thomas Applencourt、Rod Burns、Joe Curley、Jessica Davies、Henry Gabb、Zheming Jin、Rakshith Krishnappa、Praveen Kundurthy、Tim Lewis、Eric Lindahl、Gregory Lueck、Tony Mongkolsmai、Ruyman Reyes Castro、Andrew Richards、Sanjiv Shah、Neil Trevett、および Georg Viehöver が含まれます。

私たちは皆、家族や友人のサポートを受けており、感謝してもしきれません。共著者として、私たちはチームとして互いに挑戦し合い、共に学びながら作業を楽しんできました。本書の出版にあたり、Apress チーム全員と協力できたことに感謝します。

この書籍プロジェクトに多大な貢献をされた方々にも、残念ながらお名前を挙げられなかった方も少なからず居られることを承知しております。私たちを助けてくれたすべての人に感謝します。

この第 2 版を読んで、改善点が見つかった場合はぜひフィードバックを寄せてください。GitHub 経由のフィードバックから会話が始まり、必要に応じてオンラインの正誤表と書籍のサンプルが更新されます。

皆様、ありがとうございます。本書が皆様の取り組みにおいて、かけがえのない一冊となることを願っています。

第 1 章　概要

時代は間違いなく加速型コンピューティングに突入しています。世界が求める計算能力への要求を満たすため、加速型コンピューティングは、従来のソリューションに対して高いパフォーマンスと改善された電力効率を提供することで、複雑なシミュレーションや AI などさらに多くの分野を推進しています。

“コンピューター・アーキテクチャーの新たな黄金時代”[5] を迎えた今日、計算デバイスの多様化によって、私たちは大きなチャンスに直面しています。加速型コンピューティングの潜在能力を最大限に引き出すには、特定のベンダーやアーキテクチャーに依存しないポータブルなソフトウェア開発が必要です。

SYCL（*シクル*と読みます）は、業界主導の Khronos Group 標準であり、C++ による高度なデータ並列処理をサポートし、加速型（異種）システムを支援します。SYCL は、C++ コンパイラーが加速型（異種）システムを活用するメカニズムを提供し、現代の C++ および C++ ビルドシステムと高い相乗効果を発揮します。SYCL は何かの頭字語ではなく、単なる名称です。

---

[5] *A New Golden Age for Computer Architecture* by John L. Hennessy, David A. Patterson; Communications of the ACM, February 2019, Vol. 62 No. 2, Pages 48-60。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_1

加速型と異種混合型

これらの用語はセットで使用されます。異種混合（ヘテロジニアス）とは、異なる方法でプログラムされた計算デバイスの組み合わせを認める技術的な説明です。加速こそが、システムやプログラミングを複雑にする原因ですが、加速が保証されることは決してありません。異種混合システムのプログラミングは、正しく行われた場合にのみアプリケーションを加速させます。本書は、その正しい方法を教えてくれます。

SYCL を使用した C++ のデータ並列処理により、最新の加速型（異種）システム内のすべての計算デバイスにアクセスできるようになります。単一の C++ アプリケーションは、GPU、CPU、FPGA、特定用途向け集積回路（ASIC）など、身近な問題に適したデバイスの組み合わせを利用できます。独自の単一ベンダー・ソリューションでは、これほどの柔軟性を提供できません。

本書では、SYCL を使用した C++ によるデータ並列プログラミングによって、加速型コンピューティングを活用する方法を伝え、アプリケーションのパフォーマンス、計算デバイス間の移植性、そしてプログラマー自身の生産性のバランスを高める実用的なアドバイスをもたらします。この章では、データ並列処理を使用して C++ プログラムを高速化する際に覚えておくべき重要な用語など中心となる概念について説明することで本書を理解する基礎を築きます。

仕様書ではなく本書を参照

“仕様書を読んでください!” と言われて喜ぶ人はいないでしょう。一般に仕様は読みにくく、SYCL 仕様（`www.khronos.org/sycl/`（英語））も例外ではありません。あらゆる優れた言語仕様と同様に、正確性に富み、動機付けや使用法、教育については簡単に扱われています。本書は、SYCL と C++ コンパイラーの使い方を教える “学習ガイド” です。

*一度にすべて*を説明することはできません。したがって、この章では他の章にはないものを用意しています。例には、後の章まで説明されないプログラミング構造が含まれています。第 1 章のコーディング例を完全に理解することにこだわらず、章が進むごとに理解が深まることを信じましょう。

SYCL 2020 と DPC++

本書では、SYCL 2020 を使用して C++ を学習します。本書の最初の版は SYCL 2020 仕様に先行していたため、この版には、ヘッダーファイルの場所（CL ではなく sycl）、デバイスセレクター構文の調整、明示的なホストデバイスの削除などの更新が含まれています。

DPC++ は、LLVM に基づいたオープンソース・コンパイラー・プロジェクトです。SYCL が最終的に LLVM コミュニティーによってデフォルトでサポートされ、DPC++ プロジェクトがそれに貢献することを願っています。DPC++ コンパイラーは、GPU、CPU、FPGA を含む幅広い異種デバイスのサポートを提供します。本書に示すサンプルコードは、DPC++ コンパイラーで動作しますが、SYCL 2020 をサポートする他の C++ コンパイラーでも動作するはずです。

書籍の既知の正誤表を含む、最新の SYCL の重要なリソースには、書籍の Github（`https://github.com/Apress/data-parallel-CPP`）（英語）、Khronos Group の SYCL 標準 Web サイト（`www.khronos.org/sycl`）（英語）、および主要な SYCL 教育 Web サイト（`https://sycl.tech`）（英語）などがあります。

出版時点では、SYCL 2020 仕様に完全に準拠、または準拠していると主張する C++ コンパイラーはありません。[6] 本書に示すコードは、DPC++ コンパイラーで動作し、SYCL 2020 の大部分が実装されている他の C++ コンパイラーでも動作するはずです。本書では、標準 C++ と SYCL 2020 のみを使用します。

[6] 訳者注: インテルは、インテル® oneAPI DPC++/C++ コンパイラーの 2024.1 リリースで SYCL 準拠を達成した最初のベンダーになりました（https://www.isus.jp/products/c-compilers/pu56-06/）。

ただし、第 17 章（FPGA 向けプログラミング）で明確に示す少数の DPC++ 固有の拡張機能、レベルゼロ・バックエンド接続時の第 20 章（バックエンド相互運用性）、および将来展望を論じるエピローグでは例外的に使用します。

なぜ CUDA ではないのか？

CUDA とは異なり、SYCL はすべてのベンダーとすべてのタイプのアーキテクチャー（GPU だけでなく）の C++ でのデータ並列処理をサポートします。CUDA は NVIDIA GPU のサポートのみに重点を置いており、他のベンダーによる GPU での再利用の取り組み（HIP/ROCm など）は、確実な成功と有用性があるにもかかわらず、その可能性は限られています。アクセラレーターのアーキテクチャーが急増する中、この多様性を活用し、CUDA が提供していない移植性を支援するマルチベンダー/マルチアーキテクチャー・アプローチを提供できるのは SYCL だけです。

この動機をより深く理解するには、業界の伝説的人物である John L. Hennessy 氏と David A. Patterson 氏による“コンピューター・アーキテクチャーの新しい黄金時代”（英語）を読むこと（または彼らの素晴らしい講演のビデオ録画を見ること）を強く推奨します。これは必読の記事だと私たちは考えています。

第 21 章では、SYCL を使用して CUDA から C++ にコードを移行する際に役立つトピックを取り上げているほか、CUDA プログラミングの経験者が用語や機能の違いを理解する上で役立ちます。CUDA を上回る最も重要な機能は、SYCL が複数のベンダー、複数のアーキテクチャー（GPU だけでなく）、および同じデバイスに対しても複数のバックエンドをサポートできることです。この柔軟性は、“なぜ CUDA ではないのか？” という疑問に答えます。

SYCL では、CUDA や HIP と比べてオーバーヘッドは多くありません。これは互換性レイヤーではなく、ベンダーやアーキテクチャーに関係なくすべてのデバイスにオープンであり、同時に最新の C++ に同調する汎用的なアプローチです。OpenMP や OpenCL などオープンなマルチベンダーおよびマルチアーキテクチャー技術と同様に、究極的な根拠は、絶対に必要であればハードウェア固有の最適化にアクセスするオプションを含む実装にあります。

標準 C++ と SYCL である理由

繰り返しますが、SYCL を使用するすべてのプログラムは、まず第 1 に C++ プログラムです。SYCL は C++ の言語変更に依存しません。SYCL は、SYCL なしでは実現できない C++ プログラミングを実現します。加速型コンピューティング向けのすべてのプログラミングが C++ を含む言語標準に影響することは間違いありませんが、C++ 標準がすぐに SYCL を必要としなくなるほど進化するとは考えていません。SYCL には、クラスを通じて C++ を拡張する豊富な機能と、マルチベンダーおよびマルチアーキテクチャーのサポートに対する（現在すでに存在する）ニーズを満たすのに必要な新しいコンパイラー機能など豊富なサポートが本書で説明されています。

SYCL をサポートする C++ コンパイラーの入手

本書のすべての例は、DPC++ コンパイラーのさまざまなディストリビューションでコンパイルおよび動作し、SYCL をサポートする他の C++ コンパイラーでもコンパイルできるはずです（www.khronos.org/sycl の “SYCL Compilers in Development”（開発中の SYCL コンパイラー）を参照）。公開時点で DPC++ 固有の拡張機能が使用されているのはごくわずかであるため、注意して記載しています。

著者は、DPC++ コンパイラーと近い関係にあるなど、さまざまな理由から DPC++ コンパイラーを推奨しています。DPC++ は、SYCL をサポートするオープンソースのコンパイラー・プロジェクトです。LLVM を使用することで、DPC++ コンパイラー・プロジェクトは多数のデバイスのバックエンドにアクセスできるようになります。でにインテル GPU、NVIDIA GPU、AMD GPU、多数の CPU、インテル FPGA のサポートが実現しています。複数のベンダーと複数のアーキテクチャーに対するサポートをオープンに拡張および強化できるため、LLVM は SYCL をサポートするオープンソースの取り組みに最適な選択肢です。

追加のツールとライブラリーが拡張された DPC++ コンパイラーのディストリビューションがあり、ライブラリー、デバッガー、およびその他のツールを含む異種システムのサポートを提供する大規模なプロジェクト（oneAPI プロジェクトと呼ばれる）の一部として利用できます。

DPC++ コンパイラーを含む oneAPI ツールは無料で入手できます（www.oneapi.io/implementations（英語））。

```
 1. #include <iostream>
 2. #include <sycl/sycl.hpp>
 3. using namespace sycl;
 4.
 5. conststd::string secret{
 6.     "Ifmmp-!xpsme\"\012J(n!tpssz-!Ebwf/!"
 7.     "J(n!bgsbje!J!dbo(u!ep!uibu/!.!IBM\01"};
 8.
 9. constautosz = secret.size();
10.
11. int main() {
12.   queue q;
13.
14.   char* result = malloc_shared<char>(sz, q);
15.    std::memcpy(result, secret.data(), sz);
16.
17.    q.parallel_for(sz, [=](auto&i) {
18.         result[i] -= 1;
19.    }).wait();
20.
21.    std::cout << result << "\n";
22.    free(result, q);
23.    return 0;
24. }
```

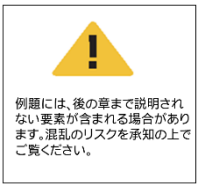

図 1-1. こんにちは、データ並列プログラミング

Hello, World! と SYCL プログラムの分析

図 1-1 に簡単な SYCL プログラムを示します。コンパイルして実行すると次のような結果が出力されます:

Hello, world!（そして、実行して体験できる追加のテキストがあります）

第 4 章の終わりまでに、この例を完全に理解できるようになります。それまでは、すべての SYCL 構成要素を定義するのに必要な `<sycl/sycl.hpp>` ファイルを `include` 文（行 2）で参照します。すべての SYCL 構成要素は、`sycl` 名前空間にあります。

- 3 行目の宣言により、`sycl::` を何度も記述する必要がなくなります。

- 12 行目では、特定のデバイス向のワーク要求キューをインスタンス化します（第 2 章）。

- 14 行目は、デバイスと共有されるデータの割り当てを作成します（第 3 章）。

- 15 行目では、未知の文字列をデバイスメモリーにコピーし、カーネルによって処理されます。

- 17 行目では、デバイスに作業をキューに投入します（第 4 章）。

- 18 行目は、デバイス上で実行される唯一のコード行です。その他のすべてのコードはホスト（CPU）で実行されます。

18 行目は、デバイス上で実行する*カーネル*コードです。このカーネルコードは 1 文字を減分します。`parallel_for()` により、カーネルは未知の文字列の各文字に対して実行され、結果の文字列にデコードされます。必要なワークの順序付けはなく、`parallel_for` がワークをキューに投入すると、メインプログラムとは非同期的に実行されます。この例では便利な機能（統合共有メモリー、第 6 章）を使用しているため、結果を確認する前にカーネルの完了を待機（19 行目）する必要があります。待機しないと、すべての文字が復号化される前に出力される可能性があります。議論すべきことはまだありますが、それは後の章で扱うことにします。

キューとアクション

第 2 章ではキューとアクションについて詳しく説明しますが、まず簡単な説明から始めましょう。キューは、アプリケーションがデバイス上で実行するワークを指示できる唯一の接続手段です。キューに配置できるアクションには、
(a) 実行するコードと、(b) メモリー操作の 2 つがあります。

実行するコードは、`single_task` や `parallel_for`（図 1-1 で使用）で表現されます。メモリー操作は、ホストとデバイス間のコピー操作、またはメモリーを初期化するフィル操作を実行します。

自動的に行われる以上に制御が必要な場合にのみ、メモリー操作を使用します。これらについては、第 2 章以降で詳しく説明します。ここでは、キューはデバイスにコマンドを送信する接続であり、コードの実行やデータの移動のため、キューに投入できる一連のアクションがあることを認識しておく必要があります。要求されたアクションは待機せずにキューに配置されることを理解することも重要です。ホストはアクションをキューに送信した後、プログラムの実行を継続しますが、デバイスは非同期に、キューを介して要求されたアクションを実行します。

キューはデバイスと接続

計算ワークとデータ移動を要求するため、キューにアクションを送信します。アクションは非同期に発生します。

すべては並列性にかかっている

C++ におけるデータ並列プログラミングは、並列処理がすべてであるため、まずこの重要な概念から始めます。並列プログラミングの目的は、何かをより高速に計算することです。これには、*スループットの向上*と*レイテンシーの短縮*という 2 つの側面があります。

スループット

プログラムのスループット向上は、一定の時間内により多くのワークを実行できるようになることで実現します。パイプライン処理などの手法では、ワークの重複が可能になり、単位時間あたりのワーク量が増えるため、単一ワーク項目の完了に必要な時間が長くなることがあります。

私たちは共同作業をするときこれによく遭遇します。ワークを共有すること自体に調整のオーバーヘッドが伴い、多くの場合、単一のワークを実行するのに要する時間が長くなります。しかし、複数の作業者の労力により、スループットが向上します。コンピューターでも同様です。ワークをより多くの処理コアに分散すると、各ワーク単位にオーバーヘッドが追加され、遅延が生じる可能性がありますが、より多くの処理コアが連携してワーク全体を完了することを目標とします。

レイテンシー

たとえば、音声コマンドを分析して応答する場合など、1 つのワークをより速く実行するにはどうすればよいでしょう。スループットだけを気にすると、応答時間が容認できないほど長くなる可能性があります。レイテンシー削減のコンセプトでは、ワーク項目を並行して処理できるように分割する必要があります。スループットに関しては、イメージ処理でイメージ全体を異なる処理ユニットに割り当てることがあります。この場合、目標は 1 秒あたりの処理できるイメージ数を最適化することになります。レイテンシーに関しては、イメージ処理をイメージ内の各ピクセルを異なる処理コアに割り当てることがあります。この場合、目標は単一のイメージの 1 秒あたりのピクセル数を最大化することです。

Think Parallel（並列に考える）

賢い並列プログラマーは、プログラミングで両方の手法を使用します。これが、*Think Parallel（並列に考える）*への私たちの探求の始まりです。

私たちは、アルゴリズムやアプリケーションに並列処理がどこに適用できるか考えるように、思考を調整したいと考えています。また、並列性を表現するさまざまな方法が、最終的に達成するパフォーマンスにどのように影響するかも考えます。一度に*すべてを*把握するのは容易ではありません。並列プログラミングを行う開発者にとって、*Think Parallel（並列に考える）*の探求は生涯にわたる目標となります。 ここでは、いくつかのヒントを学ぶことができます。

アムダールとグスタフソン

アムダールの法則は、スーパーコンピューターの先駆者であるジーン・アムダールが 1967 年に提唱したもので、複数のプロセッサーを使用する場合の理論上の最大スピードアップを予測する公式です。アムダールは、並列処理による最大ゲインは（`1/(1-p)`）に制限される（`p` は並列実行されるプログラムの割合）と嘆きました。つまり、プログラムの 3 分の 2 だけを並列で実行する場合、そのプログラムは最大でも 3 倍しかスピードアップしません。私たちはこの概念を深く理解する必要があります。これは、プログラムの 3 分の 2 をどれだけ速く実行しても、残りの 3 分の 1 が完了するまでに同じ時間がかかるためです。GPU を 100 個追加しても、パフォーマンスは 3 倍しか向上しません。

長年にわたり、一部の人々はこれを並列コンピューティングが実を結ばないことの根拠と見なしていました。1988 年、ジョン・グスタフソンは “アムダールの法則の再評価” と題する論文を発表しました。彼は、並列処理は固定されたワークの負荷を高速化するために使用されるのではなく、むしろワークのスケールアップ可能にするために用いられると指摘しました。人間も同じことを経験します。1 人の配達員は、人員とトラックを増やしても 1 つの荷物をより速く配達することはできません。しかし、1 人の配達員と 1 台のトラックよりも、100 人の配達員とトラックで 100 個の荷物をより早く配達できます。複数のドライバーを配置すると、スループットが確実に向上し、荷物配送の待ち時間も短縮されます。アムダールの法則によれば、1 人のドライバーが、自分のトラックを持つ 99 人のドライバーを追加しても、1 つの荷物をより早く配達できないことが明らかです。グスタフソンは、ドライバーとトラックを追加すれば、100 個の荷物をより早く配達できる可能性があることに気づきました。

これは、私たちが取り組む問題の規模が年々大きくなり続けており、並列処理が最も有用であることを強調しています。毎年同じサイズの問題をより速く実行したいだけであれば、並列処理の研究はそれほど重要ではありません。より大きな問題解決への取り組みこそが、コンピューター（異種/加速システム）の将来に向けて、SYCL と C++ 用いたデータ並列処理を活用することへの関心を高めています。

スケーリング

以前の議論で “スケーリング” という言葉が出てきました。スケーリングとは、追加のコンピューティングが利用可能になったときにプログラムがどれだけ高速化されるか（単に “スピードアップ” と呼ばれる）を測る尺度です。1 台のトラックと 1 人のドライバーの代わりに、100 台のトラックと 100 人のドライバーを配備するだけで、1 個の荷物と同じ時間で 100 個の荷物を配達できれば、完璧なスピードアップが実現します。もちろん、実際はその通りにはなりません。ある時点で、スピードアップを制限するボトルネックが発生します。配送センターには、トラックを 100 台駐車できる場所がないかもしれません。コンピューター・プログラムでは、ボトルネックには、データが処理される場所へのデータの移動が行われることがよくあります。100 台のトラックに配布することは、100 個の処理コアにデータを転送することに似ています。転送は瞬時に行われるものではありません。第 3 章では、異機種混合システムでデータを必要な場所に転送する方法を探ります。データ転送にはコストがかかり、そのコストがアプリケーションのスケーリングに影響することを認識することが重要です。

異種システム

ここで言う異種システムとは、複数種類の計算デバイスを含むシステムのことです。たとえば、中央処理装置（CPU）とグラフィックス処理装置（GPU）の両方を備えたシステムは異種システムです。CPU は単にプロセッサーと呼ばれることがありますが、異機種混合システム内のすべての処理ユニットを計算プロセッサーと呼ぶと混乱が生じる可能性があります。この混乱を避けるため SYCL では処理ユニットを *デバイス*と呼びます。アプリケーションは常に*ホスト*上で実行され、ホストは*デバイス*にワークを送信します。第 2 章では、メイン・アプリケーション（*ホスト*コード）が、異種システム内の特定の*デバイス*にワーク（計算）を誘導する方法について説明します。

SYCL を使用した C++ プログラムは*ホスト*上で実行され、*デバイス*にワークのカーネルを発行します。混乱を招くかもしれませんが、ホストがデバイスとして機能できる場合も多いことを知っておくことは重要です。これは、主に次の 2 つの理由で価値があります:

(1) ホストは、アクセラレーターが存在しない場合にカーネルを実行する CPU です。アプリケーションの移植性に関する SYCL の重要な約束事は、アクセラレーターがなくてもカーネルが常にどのシステムでも実行できることです。
(2) CPU には、ベクトル、行列、テンソル、および/または AI 処理機能が搭載されていることがあります。

ホストコードはデバイス上のコードを呼び出します。ホスト機能は、バックアップ・デバイスとしてだけでなく、カーネルを処理するホストの加速機能を提供するためにも、デバイスとして利用できる状況があります。ホストは、ほとんどの場合 CPU であるため、CPU デバイスとして利用できます。SYCL では CPU デバイスの保証はなく、アプリケーションのデフォルトデバイスとして使用できるデバイスが少なくとも 1 つあることのみが保証されます。

異機種混合は技術的な観点からシステム説明するものですが、ハードウェアとソフトウェアを複雑にする理由は、より高いパフォーマンスを得ようとするためです。そのため、*加速型コンピューティング*という用語は、異種システムまたはそのコンポーネントのマーケティングでよく使用されます。加速が保証されるわけではない、ことを強調しておきます。加速が保証されることは決してありませんが、異種混合システムのプログラミングが正しく行われると、アプリケーションは加速されます。本書は、正しい方法を教えてくれます。

GPU はハイパフォーマンス・コンピューティング (HPC) デバイスへと進化したため、汎用 GPU、または GPGPU と呼ばれることもあります。異種プログラミングの目的上、私たちは単に強力な GPGPU をプログラミングしていると仮定し、それらを GPU と呼ぶことができます。

現在、異種システム内のデバイスには、CPU、GPU、FPGA（フィールド・プログラマブル・ゲート・アレイ）、DSP（デジタル信号プロセッサー）、ASIC（特定用途向け集積回路）、AI チップ（グラフ、ニューロモルフィックなど）が含まれます。

このようなデバイスの設計には、計算プロセッサー（マルチプロセッサー）の重複と、メモリーなどデータソースへの接続の増加（帯域幅の増加）が伴います。これらの中でマルチプロセッシングは、スループットの向上に特に役立ちます。ここでの例えでは、追加のドライバーとトラックを導入することで実現されました。後者のデータ帯域幅が広いことは、レイテンシーを減らすのに有効です。これはトラックに並行して荷物を積み込むことができるよう、さらに多くの積み込みドックを設置することで実現できます。

複数種類のデバイスには、それぞれのアーキテクチャーがあり、特性も異なるためデバイスごとに異なるプログラミングと最適化が要求されます。これが、SYCL を使用した C++ のモチベーションとなり、本書で伝える内容の大部分を占めます。

SYCL は、異種（加速型）システムにおける C++ データ並列プログラミングの課題に対処するために開発されました。

データ並列プログラミング

“データ並列プログラミング” というフレーズは、本書のタイトルに示していますが、説明されないまま残っていました。データ並列プログラミングは、並列に操作するデータの集合として考えることができる並列性に重点を置いています。この焦点の移行は、グスタフソンとアムダールの議論に似ています。100 台のトラックとドライバーに作業を分割するには、配達する荷物が 100 個（実質的には大量のデータ）必要です。重要な概念は、何を分割するかということです。イメージ全体を処理するか、小さなタイルで処理するか、ピクセルごとに処理するか、いろいろ考えられます。

イメージ全体を処理するか、小さなタイルで処理するか、ピクセルごとに処理するか、いろいろ考えられます。さらに、オブジェクトの集合を単一の集合として分析するべきか、それともオブジェクトの小さなグループのセットとして、あるいはオブジェクトごとに分析することも考慮しなければなりません。

適切なワーク分割を選択し、そのワークを計算リソースに効果的にマッピングするのは、SYCL で C++ を使用する並列プログラマーの責任です。この議論は第 4 章から始まり、本書全体を通じて続きます。

C++ と SYCL の主な特徴

すべての SYCL プログラムは、C++ プログラムでもあります。SYCL は C++ の言語変更に依存しません。

SYCL をサポートする C++ コンパイラーは、SYCL 仕様のビルトイン知識に基づいてコードを最適化するとともに、従来の C++ ビルドシステム内で異種コンパイルが "問題なく機能" するサポートを実装します。

次に、SYCL を使用する C++ の主な特徴である、*単一ソース*スタイル、ホスト、デバイス、カーネルコード、非同期タスクグラフについて説明します。

単一ソース

プログラムは単一ソースにできます。つまり、同じ翻訳単位[7] に、デバイスで実行される計算カーネルを定義するコードと、それらのカーネルの実行を調整するホストコードの両方が含まれます。第 2 章では、この機能についてさらに詳しく説明します。必要に応じて、プログラムソースをホストコードとデバイスコードの異なるファイルと翻訳単位に分割することもできますが、重要なことは、その必要がないということです。

[7] 単に "ファイル" と言うこともできますが、ここではそれは正確でありません。翻訳単位はコンパイラーへの入力であり、C プリプロセッサーによってインライン・ヘッダー・ファイルが処理され、マクロが展開された後のソースファイルです。

ホスト

すべてのプログラムはホスト上で実行が開始され、プログラム内の*コード行*のほとんどは通常ホスト用です。これまで、ホストは常に CPU でした。標準ではこれは必須ではないため、ホストとして慎重に記述します。すべての C++ と SYCL プログラムをサポートするには、ホストが C++17 を完全にサポートする必要があるため、これが CPU 以外である可能性は低いと思われます。後で説明しますが、デバイス（アクセラレーター）は C++17 のすべてをサポートする必要はありません。

デバイス

プログラム内で複数のデバイスを使用すると、異種混合プログラミングになります。そのため、数ページ前の異種システムの説明から、この章では*デバイス*という言葉が繰り返し登場しています。異種システム内のデバイスには、GPU、FPGA、DSP、ASIC、CPU、AI チップが含まれる可能性がありますが、固定リストに限定されるわけではないことはすでに説明しました。

計算をオフロードする考え方は、ワークの完了を加速できるデバイスにワークを転送することです。これは、データの移動で失われる時間を補う必要があるため、常に忘れてはならないトピックです。

デバイスを共有

GPU などのデバイスを備えたシステムでは、複数のプログラムが同時に実行される単一のデバイスを使用する状況が想定されます。それは SYCL を使用したプログラムである必要はありません。別のプログラムがデバイスを使用している場合、プログラムの処理に遅延が発生する可能性があります。これは、CPU 向けの C++ プログラムにおける一般的な考え方とまったく同じです。CPU 上であまりにも多くのアクティブなプログラム（メール、ブラウザ、ウイルススキャン、ビデオ編集、写真編集など）を一度に実行すると、システムが過負荷になることがあります。

スーパーコンピューターでは、ノード（CPU + 接続されているすべてのデバイス）が単一のアプリケーションに排他的に許可されている場合、共有は通常問題になりません。スーパーコンピューター以外のシステムでは、複数のアプリケーションが同時に同じデバイスを使用すると、プログラムのパフォーマンスが影響を受ける可能性があることに注意してください。

すべてが引き続き機能するため、異なるプログラミングの必要はありません。

カーネルコード

デバイスコードはカーネルとして指定されます。これは C++ と SYCL++ に固有の概念ではなく、OpenCL や CUDA など他のオフロード言語でも中核となる概念です。これはループ指向のアプローチ（OpenMP `target` オフロード使用される）とは異なりますが、プログラマーがループのネストを明示的に記述する必要がなく、最も内側のループ内のコード本体に類似している場合があります。

カーネルコードには、より広範なデバイスのサポートと大規模な並列処理を可能にするため、一定の制限があります。カーネルコードで *サポートされていない* 機能リストには、動的ポリモーフィズム、動的メモリー割り当て（`new` や `delete` を使用したオブジェクト管理はありません）、静的変数、関数ポインター、ランタイム型情報（RTTI）、および例外処理が含まれます。仮想メンバー関数および可変引数関数は、カーネルコードから呼び出すことはできません。カーネルコード内では再帰は許可されません。

仮想関数

本書ではこれ以上説明しませんが、DPC++ コンパイラー・プロジェクトには、カーネル内で仮想関数のサポートを実装する実験的な拡張機能（オープンソース・プロジェクトで確認できます）があります。アクセラレーターへの効率的なオフロードの性質上、何らかの制限なしに仮想関数は十分にサポートできませんが、多くのユーザーはたとえ制限があっても SYCL がそのようなサポートを提供することに関心を示しています。オープンソースとオープン SYCL 仕様の優れた点は、C++ と SYCL 仕様の将来に影響をする実験に参加できる機会があることです。詳細については、DPC++ プロジェクト（`github.com/intel/llvm`（英語））をご覧ください。

第 3 章では、カーネルが呼び出される前後でメモリー割り当てがどのように行われるか説明し、それによってカーネルが超並列計算に集中できるようにします。第 5 章では、デバイスに関連して発生する例外処理について説明します。

カーネル内では、ファンクター、ラムダ式、演算子オーバーロード、テンプレート、クラス、静的ポリモーフィズムを含む C++ の残りの機能は自由に使用できます。また、ホストとデータを共有したり（第 3 章を参照）、（非グローバル）ホスト変数の読み取り専用値を共有することもできます（ラムダ式のキャプチャー経由）。

カーネル: ベクトル加算（DAXPY）

複雑な計算コードを扱うプログラマーには、カーネルは馴染みのあるものです。`DAXPY`（倍精度 `A × X + Y` の略）の実装を検討してください。これは何十年も定番の手法です。図 1-2 は、最新の Fortran、C/C++、および SYCL で実装された `DAXPY` を示しています。驚くべきことに、計算行（行 3）は実質的に同一です。第 4 章と第 10 章ではカーネルについて詳しく説明します。図 1-2 は、カーネルの理解が難しいという懸念を払拭するのに役立ちます。用語が新しくても、カーネルは馴染みのあるものに感じられるはずです。

```
1. ! Fortran ループ
2. do i = 1, n
3.        z(i) = alpha * x(i) + y(i)
4. end do
```

```
1. // C/C++ ループ
2. for (int i=0;i<n;i++) {
3.       z[i] = alpha * x[i] + y[i];
4. }
```

```
1. // SYCL カーネル
2. q.parallel_for(range{n},[=](id<1> i) {
3.        z[i] = alpha * x[i] + y[i];
4. }).wait();
```

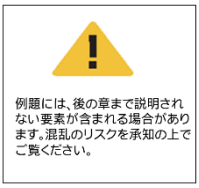

図 1-2. Fortran、C++、SYCL それぞれの DAXPY 計算

非同期実行

SYCL を使用した C++ プログラミングの非同期性を*見逃してはなりません*。非同期プログラミングを理解するのは、次の 2 つの理由から重要です:

（1）適切に使用することでパフォーマンスが向上しますが（スケーリングが向上し）、
（2）誤った使い方は並列プログラミング・エラー（通常は競合状態）につながり、

アプリケーションの信頼性を低下させます。

非同期性は、要求されたアクションの “キュー” を介してワークがデバイスに転送されることから生じます。ホストプログラムは要求されたアクションをキューに送信し、プログラムはその結果を待たずに続行されます。*待機しない* ことは、計算リソース（デバイスとホスト）を常にビジー状態に保つには重要です。待機する必要がある場合、ホストが有用なワークを行える時間を拘束することになります。また、デバイスがワークを終了すると、新しいワークをキューに投入するまで、シリアル・ボトルネックも発生します。前述したように、アムダールの法則は、ワークを並列に行わなかった時間に対してペナルティーを課します。デバイスがビジー状態であっても、デバイスとの間でデータを転送できるようにプログラムを構築し、ワークが利用可能な状態である間は常に、デバイスとホストの計算能力をすべてビジー状態に保つ必要があります。そうしないと、アムダールの法則の呪いを受けることになるでしょう。

第 3 章では、プログラムを非同期タスクグラフとして考える方法について説明し、第 8 章ではこの概念を大幅に拡張します。

ミスによる競合状態

最初のコード例（図 1-1）では、結果の値が使用可能になる前に行 21 で書き込まれないように、行 19 で“`wait`”を明示的に実行しました。この非同期動作を忘れてはなりません。同じコード例では、もう 1 つ微妙な処理が行われています。15 行目では、`std::memcpy` を使用して入力データをコピーしています。`std::memcpy` はホスト上で実行されるため、行 14 が完了するまで行 17 以降は実行されません。第 3 章を読んだ後では、これを `q.memcpy`（SYCL を使用）に変更したくなるかもしれません。図 1-3 の 7 行目でまさにそれを行っています。これはキューへの送信であるため、9 行目より前に完了する保証はありません。これにより、並列プログラミングにおけるバグの一種である*競合状態*が発生します。競合状態は、プログラムの 2 つの部分が調整なしに同じデータにアクセスするときに発生します。7 行目でデータを書き込み、9 行目でそれを読み取ることを想定しているため、7 行目が完了する前に 9 行目が実行される可能性がある競合は望ましくありません。このような競合状態により、プログラムの動作は予測不可能になり、実行ごとや異なるシステムでは結果が異なる可能性があります。これを解決するには、7 行目の末尾に `.wait()` を追加して、続行する前に `q.memcpy` が完了するのを明示的に待機します。しかし、これは最善の解決策ではありません。イベント依存関係を使用することもできます（第 8 章）。キューを順序付きキューとして作成すると、`memcpy` と `parallel_for` の間に暗黙的な依存関係が追加されます。代替案として、第 7 章では、バッファーとアクセサーのプログラミング・スタイルを使用して、SYCL に依存関係と待機を自動的に管理する方法について説明します。

```
1.  // ... 図 1-1 の 1 行を変更しています
2.  char* result = malloc_shared<char>(sz, q);
3.
4.  // 潜在的なデータ競争　後続の操作との
5.  // 正しい順序関係を保証するため
6.  // 依存関係を定義しません。
7.  q.memcpy(result, secret.data(), sz);
8.
9.    q.parallel_for(sz, [=](auto& i) {
10.      result[i] -= 1;
11.    }).wait();
12.
13.  // ...
```

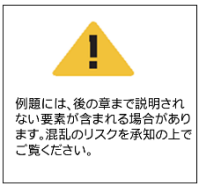

図 1-3. 非同期であることを説明するため競合状態を発生させる

競合状態は必ずしもプログラムの失敗を引き起こすわけではない

賢明な読者は、図 1-3 のコードはすべてのシステムで失敗するわけではないことに気付いているでしょう。`partition_max_sub_devices==0` の GPU を使用しても失敗しませんでした。これは小さな GPU であったため、`memcpy` が完了するまで `parallel_for` を実行する能力がなかったためです。いずれにしても、実行時に普遍的に失敗を引き起こさないとしても競合状態が存在するため、このコードには欠陥があります。私たちはそれを競合と呼んでおり、時には上手くいき、時には失敗します。このようなコーディング上の欠陥は、コンパイル環境とランタイム環境の組み合わせによって実際に障害が発生するまで、潜伏したままになる可能性があります。

`wait()` を追加すると、`memcpy` とカーネル間のホスト同期が強制されますが、これはデバイスを常にビジー状態にしておくという以前のアドバイスに反しています。本書では、プログラムの簡潔さとシステム効率のバランスを取る様々な選択肢とトレードオフについて取り上げています。

アウトオブオーダー・キューとインオーダー・キュー

本書では、パフォーマンス上の利点があるため、アウトオブオーダー・キューを使用しますが、インオーダー・キューの存在することを知っておくべきです。インオーダーは、キューを作成するときに要求できる属性です。CUDA プログラマーは、CUDA ストリームが無条件にインオーダーであることを知っています。SYCL キューはデフォルトではアウトオブオーダーですが、SYCL キューの作成時に `in_order` キュー・プロパティーを渡すことで、オプションでインオーダーにすることができます（第 8 章を参照）。第 21 章では、CUDA の使用経験があるプログラマー向けに、この点とその他の考慮事項について説明します。

カーネルを含むプログラム内のデータ競合状態の検出には、インテル® Inspector[8]（前述の “DPC++ コンパイラーの入手” で説明した oneAPI ツールキットで利用可能）などのツールが役立ちます。こうしたツールが用いる高度な手法は、多くの場合、すべてのデバイスで機能するとは限りません。競合状態を検出するには、すべてのカーネルを CPU 上で実行するのが最適であり、これは開発中のデバッグ手法として実行できます。このデバッグのヒントについては、第 2 章の「方法 #2」で説明します。

デッドロックの概念を教えるため、食事する哲学者の問題はコンピューター・サイエンスにおける同期問題の典型的な例です

円形のテーブルの周りに哲学者たちが座り、それぞれの哲学者の間には箸が 1 本ずつ置かれているところを想像してください。食事をするのに哲学者は 2 本の箸が必要でであるため、彼らは常に左右の箸を 1 本ずつ取り上げります。残念ながら、哲学者全員が最初に左側の箸をつかみ、それから右側の箸を持とうとすると、食事を開始できず全員が同時に空腹になってしまうという問題が発生します。彼らは結局、決して入手できない箸を待ち続けることになります。

[8] 訳者注: インテル® Inspector は、インテル® HPC ツールキット 2024.1 以降には含まれなくなりました。

この場合、アルゴリズムの設計が不十分（左を取り上げてから右を取り上げるまで待つ）であり、デッドロックが発生し、すべての哲学者が餓死する可能性があります。これは悲劇です。より少数の哲学者が餓死する、できれば公平に全員が食事する（誰も餓死しない）アルゴリズムを設計するため、さまざまな方法について議論することは考えるのが楽しいトピックであり、これは何度も取り上げられてきました。

このようなプログラミング・エラーがいかに起こりやすいかを認識し、デバッグ時にそれらを探して回避する方法の感覚を身につけることは、効果的な並列プログラマーになるには不可欠な経験です。

デッドロック

デッドロックは悪影響であり、並行性と並列性（この章の最後のセクションを参照）を理解することが、デッドロックを回避する方法を理解する上で不可欠であることを強調します。

デッドロックは、2 つ以上のアクション（プロセス、スレッド、カーネルなど）がブロックされ、それぞれが他方のリソースの解放またはタスクの完了を待機し、結果としてストール状態に陥った場合に発生します。つまり、デッドロックに陥ったアプリケーションは決して完了しません。待機、同期、またはロックを使用するたびに、デッドロックが発生する可能性があります。同期が不足するとデッドロックが発生する可能性がありますが、多くの場合、競合状態として現れます（前のセクションを参照）。

デッドロックはデバッグが困難です。これについては、この章の最後にある “並行性と並列性” のセクションで再度取り上げます。

第 4 章では、“ラムダ式は害ではない” ことについて説明します。DPC++、SYCL、最新の C++ をうまく活用するには、ラムダ式に慣れておくべきです。

C++ ラムダ式

ラムダ式は、並列プログラミング技術で頻繁に使用される最新の C++ の機能です。カーネル（デバイス上で実行されるコード）は複数の方法で表現できますが、最も一般的な方法はラムダ式です。第 10 章では、ラムダ式を含む、カーネルを定義するさまざまな形式について説明します。ここでは、C++ ラムダ式の復習と、カーネルの定義に関連する注意事項を示します。第 10 章では、SYCL についてさらに詳しく学んだ後、カーネルの側面について詳しく説明します。

図 1-3 のコードにはラムダ式が含まれています。それは非常に明確な `[=]` で始まるので良く分かります。C++ では、ラムダは角括弧（`[`）で始まり、閉じ角括弧（`]`）で挟まれた情報は、ラムダ内で使用されますが、パラメーターとして明示的に渡されない変数を*キャプチャー*する方法を示します。SYCL カーネルでは、キャプチャーは*値によって*行う必要があり、これは括弧内に等号（`=`）を含めることで示されます。

ラムダ式は C++11 で導入されました。これらは、囲まれたスコープから変数をキャプチャーできる匿名関数オブジェクト（名前付き変数に割り当てることも可能）を作成するのに使用されます。C++ ラムダ式の基本構文は、

```
[ capture-list ] ( params ) -> ret { body }
```

として指定されます。ここで、

- *`capture-list`* はカンマで区切られたキャプチャー・リストです。キャプチャー・リストに変数を追加することで、変数を値でキャプチャーします。変数を参照でキャプチャーするには、&v のように、先頭にアンパサンド（`&`）を付けます。スコープ内のすべての自動変数に適用されるショートカットもあります。`[=]` は、本体で使用されるすべての自動変数を値でキャプチャーし、現在のオブジェクトを参照でキャプチャーするために使用され、`[&]` は、本体で使用されるすべての自動変数と現在のオブジェクトを参照でキャプチャーするのに使用されます。`[]` は何もキャプチャーしません。SYCL では、カーネルで使用する変数の参照によるキャプチャーが許可されていないため、常に `[=]` が使用されます。
  C++ 標準に従い、グローバル変数はラムダで*キャプチャーされませ*

ん。非グローバル静的変数はカーネル内で*使用できます*が、`const` である場合に*限られます*。ここで説明するいくつかの制限により、カーネルはさまざまなデバイス・アーキテクチャーと実装にわたって一貫して動作できるようになります。

- *`params`* は、名前付き関数と同様に、関数のパラメーター・リストです。SYCL は、呼び出され他カーネルが処理する要素を識別するパラメーターを提供します。これは、一意の ID（1 次元、2 次元または 3 次元）のいずれかです。これらについては第 4 章で説明します。
- *`ret`* は戻り型です。`->ret` が指定されていない場合、`return` 文から推測されます。`return` 文がない、または値なしの `return` では、戻り値の型が `void` であることを意味します。SYCL カーネルの戻り値の型は*常に* `void` であるため、カーネルの戻り値の型を指定する際にこの構文を気にする必要はありません。
- *`body`* は関数本体です。SYCL カーネルでは、カーネルの内容にいくつかの制限があります（この章の前半の “カーネルコード” の項を参照）。

図 1-4 の例は、変数 `i` を値で取得し、変数 `j` を参照で取得する C++ ラムダ式を示しています。また、パラメーター `k0` と、参照によって受け取る別のパラメーター `l0` もあります。この例を実行すると、図 1-5 に示す出力が得られます。

```
int i = 1, j = 10, k = 100, l = 1000;

auto lambda = [i, &j](int k0, int& l0) -> int {
  j = 2 * j;
  k0 = 2 * k0;
  l0 = 2 * l0;
  return i + j + k0 + l0;
};

print_values(i, j, k, l);
std::cout << "First call returned " << lambda(k, l)
          << "\n"; print_values(i, j, k, l);
std::cout << "Second call returned " << lambda(k, l)
          << "\n"; print_values(i, j, k, l);
```

図 1-4. *C++ コードのラムダ式*

```
i == 1
j == 10
k == 100
l == 1000
First call returned 2221
i == 1
j == 20
k == 100
l == 2000
Second call returned 4241
i == 1
j == 40
k == 100
l == 4000
```

図 1-5. *図 1-4 のラムダ式コードからの出力*

ラムダ式は関数オブジェクトのインスタンスと考えることができますが、クラス定義はコンパイラーによって生成されます。たとえば、前の例のラムダ式は、図 1-6 に示すように、クラスのインスタンスに類似しています。C++ ラムダ式を使用する場合はどこでも、図 1-6 のように、関数オブジェクトのインスタンスで置き換えることができます。

関数オブジェクトを定義するときは、名前を割り当てなければなりません（図 1-6 の Functor）。インラインで表現されたラムダ式（図 1-4 のように）は、名前を必要としないため匿名です。

```
class Functor {
  public:
     Functor(int i, int &j) : my_i{i}, my_jRef{j} { }

     int operator()(int k0, int &l0) {
        my_jRef = 2 * my_jRef;
        k0 = 2 * k0; l0 = 2 * l0;
        return my_i + my_jRef + k0 + l0;
     }

  private:
   int my_i;
   int &my_jRef;
};
```

図 1-6. ラムダ式の代わりに関数オブジェクトを使用する（詳細は第 10 章を参照）

機能の移植性とパフォーマンスの移植性

移植性は SYCL を使用する C++ にとって重要な目標ですが、それを保証することはできません。言語とコンパイラーができることは、必要なときにアプリケーションで移植性を少し容易にするだけです。ドメイン固有言語、ライブラリー、フレームワークなどの高レベル（より抽象的な）プログラミングでは、規範的なプログラミングが少なくなるため、移植性が大幅に向上するのは事実です。

本書では C++ での*データ並列*プログラミングに焦点を当てており、より多くの制御が求められ、それに伴い、コーディングが移植性に与える影響を理解する責任も大きくなります。

移植性は複雑なトピックであり、これには*機能の移植性*と*パフォーマンスの移植性*の概念が含まれます。機能の移植性により、プログラムはさまざまなプラットフォームで同様にコンパイルおよび実行できることが期待されます。パフォーマンスの移植性では、さまざまなプラットフォームでプログラムが適切なパフォーマンスを発揮することを目指します。これはかなり曖昧な定義ですが、逆に言えば分かり易いかもしれません。つまり、あるプラットフォームで超高速に実行されるプログラムを作成しても、別のプラットフォームでは不当に遅くなるという状況は避けたいということです。 実際、私たちはプログラムが動作するあらゆるプラットフォームで最大限の性能を発揮することを望んでいます。異種混在システムにおける多様なデバイスを考慮すると、パフォーマンスの移植性を確保するには、プログラマーである私たちに多大な労力が必要です。

幸いなことに、SYCL はパフォーマンスの移植性を向上させるコーディング方法を定義しています。まず、汎用カーネルはどこでも実行できます。限られたケースでは、これで十分かもしれません。より一般的には、異なる種類のデバイス向けに、重要なカーネルの複数のバージョンが作成されます。 具体的には、カーネルには*汎用 GPU と汎用 CPU* バージョンが存在する場合があります。状況によっては、特定の GPU デバイス向けにカーネルを特化させたい場合があります。そのような場合は、複数のバージョンを記述し、それぞれを異なる GPU モデルに特化することができます。または、1 つのバージョンをパラメーター化して GPU の属性を使用し、GPU に適応するように GPU カーネルの実行方法を変更することもできます。

パフォーマンスの移植性に関する効果的な計画を考案するのはプログラマーの責任ですが、SYCL は計画を実装するための構文を定義しています。
前述のように、すべてのデバイス向けカーネルから始めて、必要に応じて特殊なカーネル バージョンを徐々に導入することで、機能を階層化できます。これは素晴らしいように思われますが、プログラム全体の流れも大きな影響を与える可能性があります。なぜなら、データの移動とアルゴリズムの選択全体が重要であるためです。

これを知ることで、SYCL と C++ （またはその他のダイレクト・プログラミング・ソリューション）がパフォーマンスの移植性を解決すると主張すべきでない理由が分かります。しかし、これは課題に取り組むのに役立つツールキット内のツールです。

並行性（Concurrency）と並列性（Parallelism）

*並行性*と*並列性*という用語は同義ではありませんが、同義であると誤解されることもあります。これらの用語に関する議論は、様々な情報源が同じ定義に合意することは稀であるため、さらに複雑化しています。

Sun Microsystems の*マルチスレッド・プログラミング・ガイド*にある以下の定義を考えてみましょう:[9]

- **並行性**: 少なくとも 2 つのスレッドが進行中であるときの状態
- **並列性**: 2 つのスレッド同時に実行中であるときの状態

この概念の違いを十分に理解するには、ここで何が重要であるかを直感的に理解する必要があります。以下は、その理解を得るのに役立ちます:

- **同時実行は偽装できます**: 一度に複数の処理を実行するハードウェアのサポートがない場合でも、ソフトウェアは多重化によって、一度に複数の処理を実行しているように見せかけることができます。多重化は並列性のない同時実行の良い例です。

[9] 著者たちは、決して消えることのない基礎を網羅したこのプログラミング・ガイドのファンです。オンライン docs.oracle.com/cd/E19253-01/816-5137/816-5137.pdf（英語） でご覧いただけます。

- **ハードウェア・リソースは限られています**: ハードウェアの実行リソース（プロセッサー、コア、実行ユニットなど）は有限であるため、ハードウェアが無限に“広い”ことはありません。ハードウェアが専用リソースを使用して各スレッドを実行できる場合、並行性と並列性の両方が実現されます。

私たちプログラマーが“X、Y、Z を同時に実行する”と言うとき、ハードウェアが並行性や並列性を提供するかどうかを実際には気にしないことがあります。同時に 2 つのタスクしか実行できないマシン上で、プログラム（3 つのタスクを含む）の起動が失敗するのは望ましくありません。できるだけ多くのタスクを並行して処理し、すべてのタスクが完了するまでタスクのバッチを繰り返し実行することを期待します。

しかし、時々、気にかけることもあります。そして、思考の誤りは壊滅的な結果（“デッドロック”など）をもたらす可能性があります。前の例のタスク（X、Y、または Z）の最後に、“*すべて*のタスクが完了するまで待機する”処理を追加したと想像してください。タスクの数がハードウェアの制限を超えなければ、プログラムは正常に実行されます。しかし、タスクをバッチに分割すると、最初のバッチ内のタスクは永久に待機することになります。残念ながら、それはアプリケーションが決して終了しないことを意味します。

これはよくある誤りであり、容易に発生するため、これらの概念を強調しています。熟練したプログラマーであっても、これを避けるには集中する必要があります。そして、私たちは皆、考慮が足りないときに問題をデバッグする必要があることに気づきます。これらの概念は単純ではなく、C++ 仕様には、スレッドが確実に進行する条件を詳細に説明した長いセクションが含まれています。
この導入セクションで私たちにできることは、これらの概念をできる限り理解することの重要性を強調することだけです。

これらの概念を直感的に理解することは、異種システムや加速型システムを効果的にプログラミングする上で重要です。私たちは皆、そのような直感を得るには時間がかかります。それは一度に起こるものではありません。

まとめ

この章では、SYCL を使用した C++ に必要な用語を説明し、SYCL にとって重要な並列プログラミングと C++ の主要な側面について復習しました。第 2 章、第 3 章、第 4 章では、SYCL で C++ を使用するデータ並列プログラミングの 3 つの鍵について詳しく説明します。デバイスには、実行するワークを与える（デバイス上で実行するコードを送信）、データを提供する（デバイスで使用するデータを送信）、そしてコード（カーネル）を記述する必要があります。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 2 章 コードが実行される場所

並列プログラミングは、追い越し車線を走行することではありません。実際には、*すべての*車線で高速走行することが重要です。この章では、コードをどこにでも配置できるようにする方法について説明します。私たちは、必要に応じて、異機種混合システム内のすべての計算リソースを有効にすることを選択します。したがって、それらの計算リソースがどこに隠れているか把握し（見つけて）、それらを活用（コードを実行）する必要があります。

コードが実行される*場所*を制御できます。つまり、どのカーネルにどのデバイスを使用するか制御できます。SYCL を使用した C++ は、ホスト CPU とデバイスの混合でコードを実行する異種プログラミングのフレームワークを提供します。コードが実行される場所を決定するメカニズムを理解し、使用することが重要です。

この章では、コードを実行できる場所、いつ実行されるか、および実行場所を制御するメカニズムについて説明します。第 3 章では、コードを実行する場所にデータが到達するようデータを管理する方法について説明し、第 4 章ではカーネルの記述について説明します。

単一ソース

SYCL を使用した C++ プログラムは単一ソースにすることができます。つまり、同じ翻訳単位（通常はソースファイルとそのヘッダー）に、SYCL デバイスで実行される計算カーネルを定義するコードと、それらのカーネルの実行を調整するホストコードの両方が含まれます。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_2

図 2-1 はこれら 2 つのコードパスを示しており、図 2-2 はホストコード領域とデバイスコード領域がマークされたアプリケーションの例を示しています。

デバイスコードとホストコードの両方を単一のソースファイル（または翻訳単位）に結合すると、異種アプリケーションの理解と保守が容易になります。
この組み合わせにより、言語タイプの安全性も向上し、コードに対するコンパイラーの最適化も促進される可能性があります。

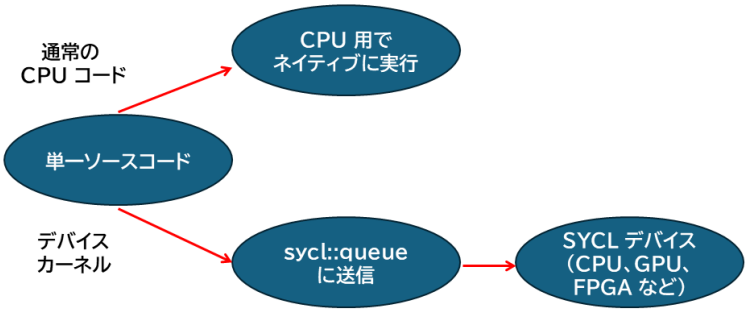

図 2-1. 単一ソースコードには、ホストコード（CPU 上で実行）とデバイスコード（SYCL デバイス上で実行）の両方が含まれます

```
#include <array>
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  constexpr int size = 16;
  std::array<int, size> data;

  // 実装で選択されたデフォルトデバイスにキューを作成
  queue Q;
     // ホストに割り当てられた "data" 配列を使用してバッファーを作成
  buffer B { data };

  q.submit([&](handler& h) {

        accessor A{B, h};
        h.parallel_for(size, [=](auto& idx) {

           A[idx] = idx;

        });
  });

  // ホスト上のバッファーへのアクセスを取得
  // デバイスカーネルがデータを生成するまで待機
  host_accessor A{B};
  for (int i = 0; i < size; i++)
    std::cout << "data[" << i << "] = " << A[i] << "\n";

    return 0;
 }
```

ホスト
コード

デバイス
コード

ホスト
コード

図 2-2. 単純な SYCL プログラム

ホストコード

アプリケーションには C++ ホストコードが含まれており、オペレーティング・システムがアプリケーションを起動した CPU で実行されます。ホストコードは、利用可能なデバイスへのワークの割り当てを定義、および制御するアプリケーションの基盤となります。これはまた、SYCL ランタイムによって管理されるデータと依存関係を定義するインターフェイスでもあります。

ホストコードは、C++ ライブラリーとして実装可能な SYCL 固有の構造とクラスが拡張された標準 C++ のコードです。これにより、ホストコードで何が許可されるか（C++ で許可されるものすべて）を判断しやすくなり、ビルドシステムとの統合が簡素化されます。

アプリケーション内のホストコードは、デバイスへのデータ移動と計算のオフロードを調整するだけでなく、計算集約型のワーク自体も実行でき、他の C++ アプリケーションと同様にライブラリーを利用できます。

デバイスコード

デバイスは、ホストコードを実行している CPU から概念的に独立したアクセラレーター、またはプロセッサーに相当します。この章の後半で説明するように、実装ではホスト・プロセッサーもデバイスとしても公開する必要がありますが、ホスト・プロセッサーとデバイスは論理的に互いに独立していると考える必要があります。ホスト・プロセッサーはネイティブ C++ コードを実行し、デバイスは追加の機能と制限を含むデバイスコードを実行します。

キューは、将来の実行のためデバイスにワークを送信するメカニズムです。デバイスコードには、理解しておくべき 3 つの重要な特性があります:

1. **ホストコードから非同期に実行されます。**ホストプログラムはデバイスコードをデバイスに送信し、ランタイムは実行に必要なすべての依存関係が満たされた場合にのみ、ワークを追跡して開始します（詳細は第 3 章で説明します）。ホストプログラムの実行は、送信されたワークがデバイス上で開始される前に続行され、明示的に 2 つを関連付けない限り、デバイス上の実行はホストプログラムの実行とは非同期であるという特性があります。

   この非同期実行の作用として、ホストプログラムがホストアクセサーやブロッキング・キュー待機操作などのメカニズムを通じて実行を強制するまで、デバイスでワークが開始されることは保証されません。これらのメカニズムについては、後の章で説明します。

2. アクセラレーター・デバイス上でコンパイルしてパフォーマンスを実現するため、**デバイスコードには制限があります。**たとえば、動的メモリー割り当てやランタイム型情報（RTTI）は、多くのアクセラレーターでパフォーマンスの低下につながるため、デバイスコード内ではサポートされていません。デバイスコードの制限の一部については、第 10 章で詳しく説明します。

3. **SYCL によって定義された一部の関数とクエリーは、デバイスコード内でのみ意味を成すため、デバイスコード内でのみ使用できます。**たとえば、デバイスコードの実行インスタンスが、より大きなデータ並列範囲での位置を照会するワーク項目の識別子クエリーなどがありますす（第 4 章で説明）。

一般に、キューに送信されるワークを *アクション*と呼びます。第 3 章では、アクションには実行するデバイスコードだけでなく、メモリー移動コマンドも含まれることを学習しますこの章では、アクションのデバイスコードに焦点を当てているため、デバイスコードについて具体的に説明しています。

デバイスの選択

デバイスコードが実行される場所を制御するメカニズムを知るため、次の 5 つのユースケースを考えてみます:

方法 #1: どのデバイスが使用されるかを気にせず、デバイスコードを実行します。これは最も単純なステップであり、開発の最初のステップとなります。

方法 #2: 明示的に CPU デバイス上でデバイスコードを実行します。ほとんどの開発システムにはアクセス可能な CPU があるため、デバッグ時に使用されます。
CPU デバッガーは非常に豊富な機能を備えています。

方法 #3: デバイスコードを GPU または別のアクセラレーターにディスパッチします。

方法 #4: GPU や FPGA など異機種デバイスのセットにデバイスコードをディスパッチします。

方法 #5: 利用可能な FPGA タイプの集合から特定のタイプの FPGA を選択するなど、一般的なデバイスクラスから特定のデバイスを選択します。

開発者は通常、方法 #2 で可能な限りコードをデバッグし、可能な限りコードをテストした後にのみ、方法 #3 ～ #5 に移行します。

方法 #1: あらゆるタイプのデバイスで実行

デバイスコードがどこで実行されるかを気にしない場合は、ランタイムに選択させるのが簡単です。この自動選択は、どのデバイスを選択するかまだ決定していない場合に、コードの記述と実行を容易に開始できるように設計されています。このデバイス選択は実行されるコードを考慮していないため、最適ではない恣意的な選択と見なすべきです。

デバイス選択に詳しく触れる前に、実装によって選択されていても、最初にプログラムがデバイスと対話するメカニズムである*キュー*について説明します。

キュー

キューは、単一のデバイスで実行するためにアクションが送信される抽象化です。簡略化されたキュークラスの定義を、図 2-3 と図 2-4 に示します。
アクションは通常、データ並列計算の開始を意味しますが、SYCL ランタイムによる自動移動よりも詳細な制御が必要な場合など、データ移動を手動で制御する他のコマンドも利用できます。キューに送信されたワークは、入力データの可用性など、ランタイムが追跡する前提条件が満たされた後に実行されます。これらの前提条件については、第 3 章と第 8 章で説明します。

```
class queue {
 public:
  // デフォルトデバイスに関連付けられたキューを作成
  queue(const property_list = {});

  queue(const async_handler &, const property_list & = {});

  // DeviceSelector を使用してキューを作成します。
  // DeviceSelector は、デバイスを数値でランク付けする呼び出し可能
  // オブジェクトです。cpu_selector_v や gpu_selector_v など、
  // SYCL 定義のデバイスセレクターがいくつかあります。
  template <typename DeviceSelector>
  explicit queue(const DeviceSelector &deviceSelector,
                 const property_list &propList = {});

  // プログラムがすでに参照を保持している
  // 明示的なデバイスに関連付けられたキューを作成します。
  queue(const device &, const property_list & = {});

  // 特定の SYCL コンテキスト内のデバイスに関連付けられた
  // キューを作成します。デバイスの代わりにデバイスセレクターを
  // 使用することもできます。
  queue(const context &, const device &,
        const property_list & = {});
};
```

図 2-3. キュークラスのコンストラクターの簡略化された定義

```
class queue {
 public:
  // コマンドグループをこのキューに送信します。
  // コマンドグループは、ラムダ式または、関数オブジェクトにすることができます。
  // コマンドグループで実行されたアクションのステータスを反映するイベントを返します。
  template <typename T>
  event submit(T);

  // 以前に送信されたすべてのアクションの実行が完了するまで待機します。
  void wait();

  // 以前に送信されたすべてのアクションの実行が完了するまで待機します。
  // 非同期例外を async_handler 関数に渡します。
  void wait_and_throw();
};
```

図 2-4. キュークラスの主要メンバー関数の簡略化された定義

キューは単一のデバイスにバインドされ、そのバインドはキューの構築時に発生します。キューに送信されたワークは、キューがバインドされている単一デバイス上で実行されることを理解することが重要です。キューをデバイスの集合にマップすることはできません。そうすると、どのデバイスがワークを実行するべきか曖昧になります。同様に、キューは自身に送られたワークを複数のデバイスに分散できません。代わりに、図 2-5 に示すように、キューと、そのキューに送信されたワークが実行されるデバイスの間には明確なマッピングが存在します。

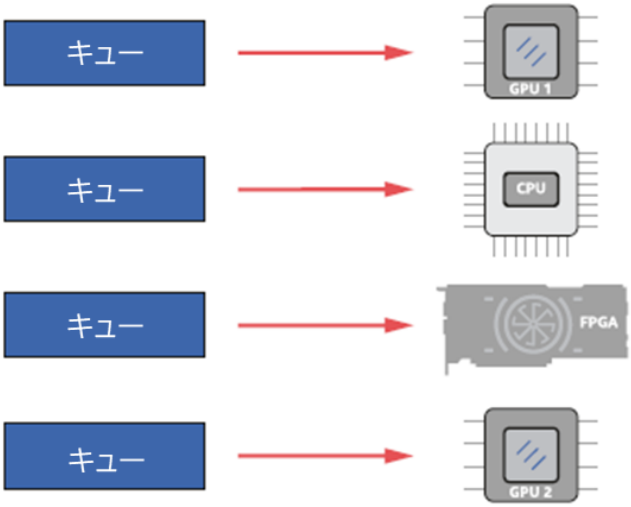

図 2-5. キューは単一デバイスに割り当てられます。キューに送信されたワークはそのデバイス上で実行されます

アプリケーションの設計やプログラミング・スタイルに応じて、プログラム内に複数のキューを作成できます。たとえば、それぞれ異なるデバイスにバインドしたり、ホストプログラム内の異なるスレッドで使用するため、複数のキューを作成できます。複数の異なるキューを GPU などの単一のデバイスにバインドすることができます。それらの異なるキューに送信すると、バインドされたワークがデバイス上で実行されます。この例を図 2-6 に示します。逆に、前述したように、キューは複数のデバイスにバインドできません。それは、アクションの実行要求がどこに対して行われているか、いかなる曖昧さもあってはならないためです。 たとえば、複数のデバイス間でワークを分散するキューが必要な場合、コード内にその抽象化を作成できます。

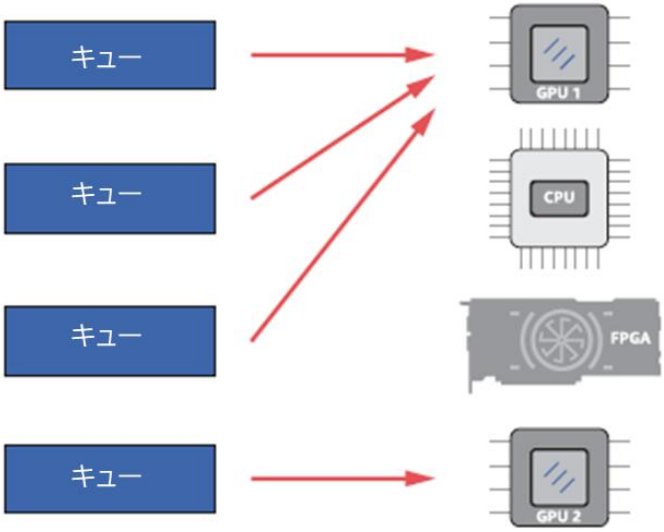

図 2-6. 同じデバイスに複数のキューを割り当てることができます

キューは特定のデバイスにバインドされるため、キューの構築は、キューに送信されたアクションが実行されるデバイスをコード内で選択する最も一般的な方法です。キューを構築する際のデバイス選択は、デバイスセレクターの抽象化によって実現されます。

任意のデバイスが利用可能な場合のキューとデバイスのバインド

図 2-7 は、キューがバインドされるデバイスが指定されていない例です。引数を取らないデフォルト・キュー・コンストラクター（図 2-7 を参照）は、バックグラウンドで利用可能なデバイスを選択します。
SYCL は、少なくとも 1 つのデバイスが常に使用可能であることを保証するため、デフォルトの選択メカニズムによって常に何らかのデバイスを選択します。多くの場合、選択されたデバイスはホストプログラムを実行している CPU である可能性がありますが、これは保証されません。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  // 実装で選択されたデフォルトのデバイス上にキューを作成します。
  // 暗黙的に default_selector_v を使用します。
  queue q;

  std::cout << "Selected device: "
            << q.get_device().get_info<info::device::name>()
            << "\n";

   return 0;
}
```

サンプル出力（システムに応じて実行ごとに 1 行）:

```
Selected device: NVIDIA GeForce RTX 3060
Selected device: AMD Radeon RX 5700 XT
Selected device: Intel(R) Data Center GPU Max 1100
Selected device: Intel(R) FPGA Emulation Device
Selected device: AMD Ryzen 5 3600 6-Core Processor
Selected device: Intel(R) UHD Graphics 770
Selected device: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Selected device: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
```

他にもたくさん考えられます…これらはほんの一例です

図 2-7. キューの簡単な構築による暗黙的なデフォルト・デバイス・セレクター

単純なキュー・コンストラクターを使用するのは、アプリケーション開発を開始し、デバイスコードを起動して実行する簡単な方法です。アプリケーションの要件に応じて、キューにバインドされたデバイスの選択をさらに制御できるようになります。

方法 #2: 開発、デバッグ、デプロイメントに CPU デバイスを使用

CPU デバイスは、ホスト CPU が独立したデバイスとして動作するものと考えることができ、これによりシステム内のアクセラレーターの有無にかかわらず、デバイスコードを実行できるようになります。ホストプログラムを実行するプロセッサーが常に稼働しているため、CPU デバイスはアプリケーションから常に利用できます

（さまざまな理由から、実装によって CPU が SYCL デバイスとして公開されないこともまれにあります）。コード開発に CPU デバイスを使用すると、いくつかの利点があります:

1. アクセラレーターを搭載していない低性能システムでの**デバイスコードの開発**: 一般的な用途としては、パフォーマンス・テストと最適化のため、HPC クラスターに展開する前にローカルシステムでデバイスコードを開発およびテストすることが挙げられます。
2. 非アクセラレーター・ツールを使用した**デバイスコードのデバッグ**: アクセラレーターは、多くの場合、ホスト CPU で利用できるほど高度なデバッグツールを備えていない低レベルの API を通じて公開されます。この点を踏まえ、CPU デバイスは、開発者に馴染み深い標準ツールを用いたデバッグをサポートすることが期待されます。
3. 他のデバイスが利用できない場合の**バックアップ**として、デバイスコードが機能的に実行できることを保証します: CPU デバイスは、パフォーマンスを主目的としない場合や、カーネルコードが最適化されたアーキテクチャーに適合しない可能性がありますが、デバイスコードがどのアプリケーションでも常に実行できることを保証する機能的なバックアップとして扱われます。

SYCL アプリケーションでは複数の CPU デバイスが使用可能であり、デバッグの容易さを目的としたデバイスもあれば、実行パフォーマンスに重点を置いたデバイスもあります。この章の後半で説明するように、デバイスアスペクトを使用して、これらの異なる CPU デバイスを区別できます。

デバイスコードの開発とデバッグに CPU デバイスを使用すること検討する場合、CPU とターゲット・アクセラレーター・アーキテクチャー（GPU など）の違いを考慮する必要があります。

特にコードのパフォーマンス最適化を目的とし、さらにサブグループなどより高度な機能を使用する場合、アーキテクチャー間で機能とパフォーマンスに差異が生じる可能性があります。たとえば、新しいデバイスに移行すると、サブグループのサイズが変わる場合があります。通常、開発とデバッグのほとんどは CPU デバイス上で実行され、その後にターゲット・デバイス・アーキテクチャー上で最終的なチューニングとデバッグが行われることでしょう。

CPU デバイスは、キューをバインドしてデバイスコードを実行できるという点で、機能的にはハードウェア・アクセラレーターと同様です。図 2-8 は、CPU デバイスがシステムで使用可能な他のアクセラレーターと対等な関係にある様子を示しています。GPU や FPGA と同様にデバイスコードを実行でき、1 つ以上のキューを構築してバインドできます。

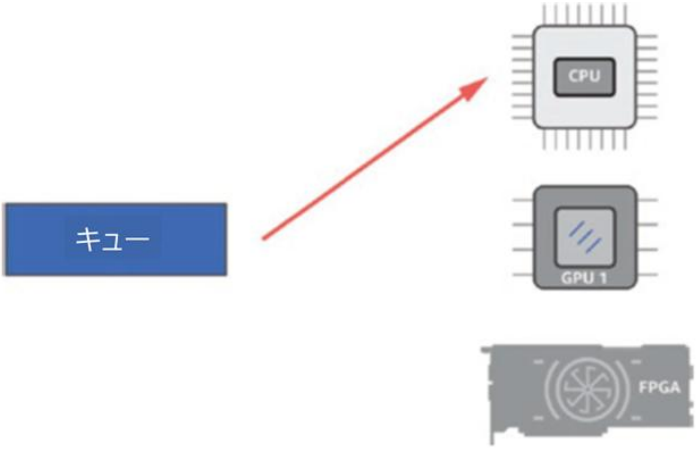

図 2-8. CPU デバイスは他のアクセラレーターと同様にデバイスコードを実行できます

図 2-9 に示すように、アプリケーションは、キュー・コンストラクターに `cpu_selector_v` を明示的に渡すことで、CPU デバイスにバインドされたキューを作成することもできます。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
   // CPU デバイスを明示的に使用するキューを作成
   queue  q{ cpu_selector_v };

   std::cout << "Selected device: "
             << q.get_device().get_info<info::device::name>()
             << "\n";
   std::cout
       << " -> Device vendor: "
       << q.get_device().get_info<info::device::vendor>()
       << "\n";

   return 0;
}
```

出力例:

```
Selected device: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
 -> Device vendor: Intel(R) Corporation
```

図 2-9. cpu_selector_v を使用してホストデバイスを選択します

特に要求していない場合でも (cpu_selector_v を使用するなど)、図 2-7 の出力に示すように、デフォルトセレクターによって CPU デバイスが選択されることがあります。

デバイスセレクターにはいくつかのバリエーションが定義されており、特定種類のデバイスを簡単にターゲットにできます。cpu_selector_v はこれらのセレクターの 1 例であり、次のセクションで他のセレクターについても説明します。

方法 #3: GPU (または他のアクセラレーター) を使用

次の例は GPU を示していますが、どのタイプのアクセラレーターでも同様に適用できます。アクセラレーターの一般的なクラスを容易にターゲットにできるように、デバイスはいくつかの広範なカテゴリーにグループ化されており、SYCL はそれらのビルトイン・セレクター・クラスを提供しています。“システムで利用可能な任意の GPU” など、広範なデバイス・タイプ・カテゴリーから選択する場合、対応するコードはこのセクションで説明するように非常に簡潔です。

アクセラレーター・デバイス

SYCL 仕様の用語では、アクセラレーターの種類はいくつかの広範なグループに分けられます:

1. CPU デバイス。
2. GPU デバイス。
3. アクセラレーターは、CPU デバイスでも GPU でもないデバイスをキャプチャーします。これには、FPGA や DSP デバイスが含まれます。

これらのカテゴリーのデバイスは、ビルトインセレクターを使用してキューに簡単にバインドでき、キュー（およびその他のクラス）のコンストラクターに渡すことができます。

デバイスセレクター

キュークラスなど、特定のデバイスにバインドする必要があるクラスには、`DeviceSelector` を受け入れるコンストラクターを持ちます。`DeviceSelector` は、デバイスへの const 参照を受け取る呼び出し可能オブジェクトであり、デバイスを数値的にランク付けして、ランタイムが最高ランクのデバイスを選択できるようにします。例えば、`DeviceSelector` を受け入れるキューのコンストラクターの 1 つは、`queue( const device_selector &deviceSelector,const property_list &propList = {});` です。

一般的な広範囲のクラスのデバイスに対応する 4 つのビルトインセレクターがあります:

`default_selector_v`	実装者が選択した任意のデバイス
`cpu_selector_v`	デバイスクエリーで自身を CPU として識別するデバイスを選択します
`gpu_selector_v`	デバイスクエリーで自身を GPU として識別するデバイスを選択します。
`accelerator_selector_v`	FPGA を含む、“アクセラレーター” として識別されるデバイスを選択します

DPC++ に含まれる追加セレクター（SYCL では利用できません）は、ヘッダー ”sycl/ext/intel/fpga_extensions.hpp” をインクルードすることで利用できます。

`ext::intel::fpga_selector_v` は、自身を FPGA として識別するデバイスを選択します

キューは、`queue myQueue{ gpu_selector_v{} };` などのビルトインセレクターを使用して構築できます。

図 2-10 は GPU セレクターを使用した例を示し、図 2-11 はキューと使用可能な GPU デバイスの対応するバインドを示しています。

図 2-12 は、さまざまなビルトイン・セレクター・クラスを使用する例を示しており、構築時にデバイスセレクターを受け入れる別のクラス（デバイス）のデバイスセレクターの使い方も示しています。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int  main() {
   // 利用可能な GPU デバイスにバインドされたキューを作成
   queue q{gpu_selector_v};

   std::cout << "Selected device: "
             << q.get_device().get_info<info::device::name>()
             << "\n";
   std::cout
       << " -> Device vendor: "
       << q.get_device().get_info<info::device::vendor>()
       << "\n";

   return 0;
}
```

出力例:

```
Selected device: AMD Radeon RX 5700 XT
 -> Device vendor: AMD Corporation
```

図 2-10. GPU デバイスセレクターの例

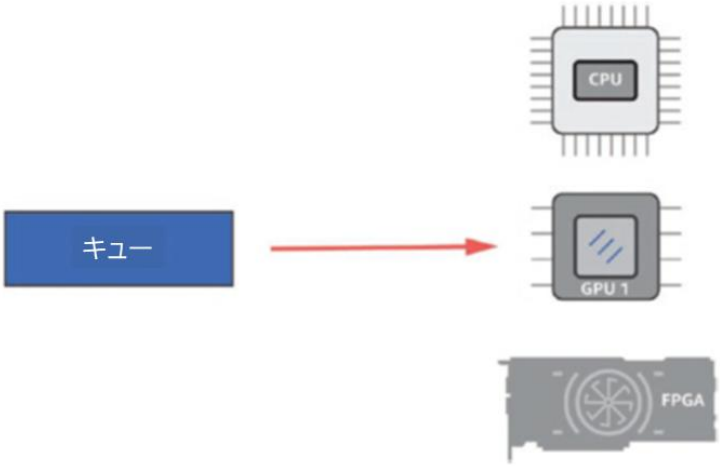

図 2-11. アプリケーションで利用可能な GPU デバイスにバインドされたキュー

```
#include <iostream>
#include <string>
#include <sycl/ext/intel/fpga_extensions.hpp>  //  fpga_selector_v 向け
#include <sycl/sycl.hpp>
using namespace sycl;

void output_dev_info(const device& dev,
                     const std::string& selector_name) {
  std::cout << selector_name << ": Selected device: "
            << dev.get_info<info::device::name>() << "\n";
  std::cout << "            -> Device vendor: "
            << dev.get_info<info::device::vendor>() << "\n";
}

int main() {
 output_dev_info(device{default_selector_v},
                 "default_selector_v");
 output_dev_info(device{cpu_selector_v}, "cpu_selector_v");
 output_dev_info(device{gpu_selector_v}, "gpu_selector_v");
 output_dev_info(device{accelerator_selector_v},
                 "accelerator_selector_v");
 output_dev_info(device{ext::intel::fpga_selector_v},
                 "fpga_selector_v");

   return 0;
}
```

出力例:

```
default_selector_v: Selected device: Intel(R) UHD Graphics [0x9a60]
                  -> Device vendor: Intel(R) Corporation
cpu_selector_v: Selected device: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
                  -> Device vendor: Intel(R) Corporation
gpu_selector_v: Selected device: Intel(R) UHD Graphics [0x9a60]
                  -> Device vendor: Intel(R) Corporation
accelerator_selector_v: Selected device: Intel(R) FPGA Emulation Device
                  -> Device vendor: Intel(R) Corporation
fpga_selector_v: Selected device: pac_a10 : Intel PAC Platform (pac_ee00000)
                  -> Device vendor: Intel Corp
```

図 2-12. さまざまなクラスのデバイスセレクターのデバイス識別出力の例と、デバイスセレクターがキュー以外の構築にも使用できる例 (この場合は、デバイス・クラス・インスタンスの構築)

デバイスの選択に失敗した場合

キューなどのオブジェクトを作成する際に GPU セレクターが使用され、ランタイムで使用できる GPU デバイスがない場合、セレクターは `runtime_error` 例外をスローします。これはすべてのデバイス・セレクター・クラスに当てはまり、必要なクラスのデバイスが利用できない場合、`runtime_error` 例外がスローされます。

複雑なアプリケーションでは、エラーをキャッチし、代わりに（アプリケーション/アルゴリズムにとって）あまり望ましくないデバイスクラスを取得するのが合理的です。例外とエラー処理については、第 5 章で詳しく説明します。

方法 #4: 複数のデバイスを使用

図 2-5 および 2-6 に示すように、アプリケーション内には複数のキューを構築できます。これらのキューは、単一のデバイス（キューへのワークの合計が単一のデバイスに集約されます）、複数のデバイス、またはこれらの組み合わせにバインドできます。図 2-13 は、GPU にバインドされたキューと、FPGA にバインドされたキューをそれぞれ 1 つずつ作成する例を示しています。対応するマッピングを図 2-14 に示します。

```
#include <iostream>
#include <sycl/ext/intel/fpga_extensions.hpp>  //  fpga_selector_v 向け
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
   queue my_gpu_queue(gpu_selector_v);
   queue my_fpga_queue(ext::intel::fpga_selector_v);

   std::cout << "Selected device 1: "
             << my_gpu_queue.get_device()
                    .get_info<info::device::name>()
             << "\n";

   std::cout << "Selected device 2: "
             << my_fpga_queue.get_device()
                    .get_info<info::device::name>()
             << "\n";

   return 0;
}
```

出力例:

```
Selected device 1: Intel(R) UHD Graphics [0x9a60]
Selected device 2: pac_a10 : Intel PAC Platform (pac_ee00000)
```

図 2-13. GPU と FPGA デバイス両方へのキューを作成

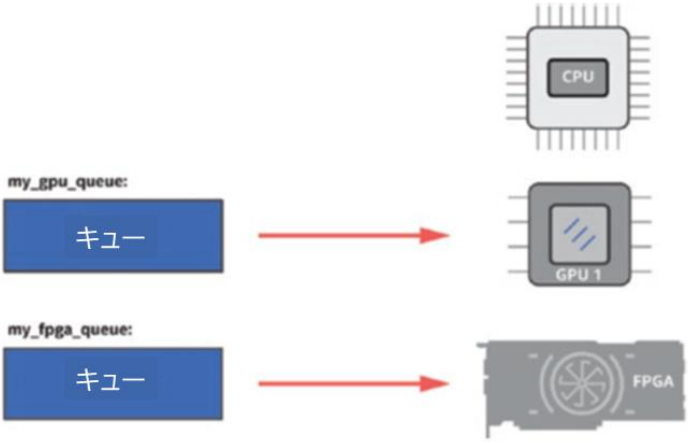

図 2-14. GPU + FPGA デバイスセレクターの例: 1 つのキューは GPU にバインドされ、もう 1 つのキューは FPGA にバインドされます

方法 #5: カスタム (非常に特定の) デバイス選択

ここでは、カスタムセレクターの記述方法を見ていきます。この章の例のほかに、第 12 章にもいくつかの例が示されています。ビルトイン・デバイス・セレクターは、コードを素早く起動して実行することを目的としています。実際のアプリケーションでは通常、システムで利用可能な GPU タイプの中から必要な GPU を選択するなど、専用デバイスの選択が必要になることがあります。デバイス選択のメカニズムは、複雑なロジックへ容易に拡張できるため、希望するデバイスの選択に必要なコードを自由に記述できます。

デバイスアスペクトに基づく選択

SYCL は、*アスペクト*と呼ばれるデバイスのプロパティーを定義します。たとえば、デバイスが示す可能性のあるアスペクト (アスペクトクエリーで `true` を返す) には、`gpu`、`host_debuggable`、`fp64`、`online_compiler` などがあります。標準的なアスペクトとその定義の完全なリストについては、SYCL 仕様の “デバイスアスペクト” セクションを参照してください。

SYCL で定義されたアスペクトを使用してデバイスを選択するには、図 2-15 に示すように aspect_selector を使用できます。カンマで区切られたアスペクトのグループを受け取る aspect_selector の形式では、デバイスが選択されるには、すべてのアスペクトがデバイスによって示される必要があります。aspect_selector の代替形式は、2 つの std::vector を取ります。最初のベクトルには、デバイスに必要なアスペクトが含まれ、2 番目のベクトルには、デバイスに必要のないアスペクト（否定的な側面をリストする）が含まれます。図 2-15 は、aspect_selector でこれら両方の形式を使用する例を示しています。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
   // アスペクトセレクターでは、コンマで区切られたアスペクトのグループを取得するため、
   // デバイスを選択するにはすべてのアスペクトが存在している必要があります。
   //
   queue q1{aspect_selector(aspect::fp16, aspect::gpu)};

   // 2 つのベクトルを取る aspect_selector 形式では、
   // 最初のベクトルにはデバイスが示さなければならない側面が含まれ、
   // 2 番目のベクトルには示されてはならない側面が含まれます。
   //
   queue q2{aspect_selector(
      std::vector{aspect::fp64, aspect::fp16},
      std::vector{aspect::gpu, aspect::accelerator})};

   std::cout
       << "First selected device is: "
       << q1.get_device().get_info<info::device::name>()
       << "\n";

   std::cout
       << "Second selected device is: "
       << q2.get_device().get_info<info::device::name>()
       << "\n";

   return 0;
}
```

出力例:

```
First selected device is: Intel(R) UHD Graphics [0x9a60]
Second selected device is: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
```

図 2-15. アスペクトセレクター

いくつかのアスペクトは、デバイスのパフォーマンス特性を推測するのに使用される場合があります。たとえば、エミュレートされたアスペクトを持つデバイスは、エミュレートされていない同種のデバイスほど性能が良くない可能性がありますが、代わりにデバッグ性の向上に関連する他のアスペクトを示す場合があります。

カスタムセレクターによる選択

既存のアスペクトが特定のデバイスの選択に十分でない場合、カスタム・デバイス・セレクターを定義できます。このようなセレクターは、単に `const Device&` をパラメーターとして受け取り、特定デバイスの整数スコアを返す C++ 呼び出し可能オブジェクト (関数やラムダなど) です。SYCL ランタイムは、検出したすべての利用可能なルートデバイスに対してセレクターを呼び出し、セレクターが最高のスコアを返したデバイスを選択します (選択されるには、スコアが負でない必要があります)。

最高スコアが同点の場合、SYCL ランタイムは同点のデバイスの 1 つを選択します。セレクターが負の数を返すデバイスは、ランタイムによって選択されません。つまり、セレクターから負の数を返すと、デバイスが選択されないことを保証できます。

デバイスのスコアメカニズム

次のように、特定のデバイスに対応する整数スコアを作成する多数のオプションがあります:

1. 特定のデバイスクラスの正の値を返します。
2. デバイス名やデバイスベンダーの文字列との文字列の一致。
3. デバイスまたはプラットフォームのクエリーに基づいて、整数値につながると考えられるあらゆるものを計算します。

たとえば、インテル® Arria FPGA アクセラレーター・ボードを選択する方法を図 2-16 に示します。

```
int my_selector(const device &dev) {
   if (dev.get_info<info::device::name>().find("pac_a10") !=
            std::string::npos &&
       dev.get_info<info::device::vendor>().find("Intel") !=
            std::string::npos) {
       return 1;
   }
   return -1;
}
```

出力例:

```
Selected device is: pac_a10 : Intel PAC Platform (pac_ee00000)
```

図 2-16. インテル® Arria FPG アクセラレーター・ボードの用カスタムセレクター

第 12 章では、デバイスの選択に関する詳細な説明と例を示し、get_info メソッドについてさらに詳しく説明します。

デバイス上でワークを作成

アプリケーションには通常、ホストコードとデバイスコードの組み合わせが含まれます。デバイスコードを実行するために送信できるクラスメンバーがいくつかあります。これらのワーク・ディスパッチ構造はデバイスコードを送信する唯一の方法であるため、デバイスコードとホストコードを簡単に区別できます。

この章の残りでは、デバイスコードとホスト・プロセッサー上でネイティブに実行されるホストコードを理解して識別することを目的として、いくつかのワークのディスパッチ構造を紹介します。

タスクグラフ

SYCL 実行モデルの基本的な概念は、ノードのグラフです。このグラフ内の各ノード（ワーク単位）には、デバイスで実行されるアクションが含まれており、最も一般的なアクションはデータ並列デバイスカーネルの呼び出しです。

図 2-17 は 4 つのノードを持つグラフの例を示しています。各ノードはデバイスカーネルの呼び出しと考えることができます。

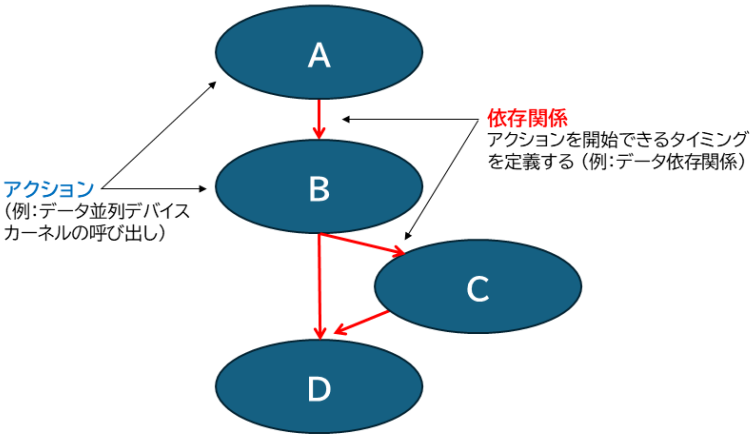

図 2-17. タスクグラフは、1 つまたは複数のデバイスで（ホストプログラムとは非同期に）実行するアクションと、アクションを安全に実行できるタイミングを決定する依存関係を定義します

図 2-17 のノードには、ノードのワーク実行をいつ開始するのが適切であるか定義する依存エッジがあります。依存関係エッジは、通常、データ依存関係から自動的に生成されますが、必要に応じてカスタム依存関係を手動で追加する方法もあります。たとえば、グラフのノード B には、ノード A からの依存関係エッジがあります。このエッジは、ノード A が実行を完了する必要があり、依存関係に応じて、ノード B のアクションが開始される前に、ノード B が実行されるデバイスで生成されたデータを使用できるようにする必要があることを意味します。ランタイムは、ホストプログラムの実行とは完全に非同期に依存関係の解決とノード実行のトリガーを制御します。

アプリケーションを定義するノードのグラフは、本書ではタスクグラフと呼び、第3 章で詳しく説明します。

デバイスコードはどこにあるか？

デバイス上で実行されるコードを定義するメカニズムは複数ありますが、簡単な例でそのようなコードを識別する方法を示します。例のパターンは複雑に見えますが、そのパターンはすべてのデバイスコード定義で同じままであるため、すぐに慣れることができます。

図 2-18 でラムダ式として定義されている、parallel_for の最後の引数で渡されるコードは、デバイス上で実行されるデバイスコードです。この場合の parallel_for は、デバイスコードとホストコードを区別する構造です。parallel_for は、デバイス上で実行されるコードを定義するハンドラークラスのすべてのメンバーであるデバイス・ディスパッチ・メカニズムの小さなセットです。ハンドラークラスの簡略化された定義を図 2-19 に示します。

```
q.submit([&](handler& h) {
  accessor acc{B, h};

  h.parallel_for(size,
        [=](auto& idx) { acc[idx] = idx; });
});
```

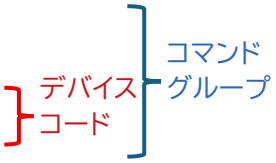

図 2-18. デバイスコードの送信

```
class handler {
 public:
   // このコマンド グループで定義されたアクションが実行される前に
   // 完了する必要があるイベントを指定します。
   void depends_on(std::vector<event> & events);

   //  このアクションの実行後、アクセサーによってアクセスされた
   //  メモリー・オブジェクトがホスト上で更新されることを保証ます。
   template <typename AccessorT>
   void update_host(AccessorT acc);

   // 指定されたポインターに書き込む memset 操作を送信します。
   // この操作を表すイベントを返します。
   event memset(void *Ptr, int Value, size_t Count);

   // src から dest にコピーする memcpy 操作を送信します。
   // この操作を表すイベントを返します。
   event memcpy(void *dest, const void *src, size_t count);

   // アクセサーとホストメモリー間のコピーを行います。
   // アクセスには適切な権限が必要です。
   //ポインターは、生のポインターまたは shared_ptr にできます。
   template <typename SrcAccessorT, typename DestPointerT>
   void copy(SrcAccessorT src, DestPointerT dest);

   template <typename SrcPointerT, typename DestAccessorT>
   void copy(SrcPointerT src, DestAccessorT dest);

   // アクセサー間でコピーします。
   // アクセスには適切な権限が必要です。
   template <typename SrcAccessorT, typename DestAccessorT>
   void copy(SrcAccessorT src, DestAccessorT dest);

   // 実行用にさまざまな形式のカーネルを送信します、
   template <typename KernelName, typename KernelType>
   void single_task(KernelType kernel);

   template <typename KernelName, typename KernelType, int Dims>
   void parallel_for(range<Dims> num_work_items,
                     KernelType kernel);

   template <typename KernelName, typename KernelType, int Dims>
   void parallel_for(nd_range<Dims> execution_range,
                     KernelType kernel);

   template <typename KernelName, typename KernelType, int Dims>
   void  parallel_for_work_group(range<Dims> num_groups,
                                 KernelType kernel);

   template <typename KernelName, typename KernelType, int Dims>
   void parallel_for_work_group(range<Dims> num_groups,
                                range<Dims> group_size,
                                KernelType kernel);
};
```

図 2-19.ハンドラークラスのメンバー関数の簡略化された定義

ハンドラークラスのメンバーを呼び出してデバイスコードを送信するだけでなく、ワークを送信できるキュークラスのメンバーもあります。図 2-20 に示すキュークラスのメンバーは、特定のパターンを簡略化するショートカットであり、以降の章でこれらのショートカットの使用方法を説明します。

```
class queue {
  public:
   // 指定されたポインターに書き込む memset 操作を送信します。
   // この操作を表すイベントを返します。
   event memset(void* ptr, int value, size_t count);

   // src から dest にコピーする memcpy 操作を送信します。
   // この操作を表すイベントを返します。
   event memcpy(void* dest, const void* src, size_t count);

   // 実行用にさまざまな形式のカーネルを送信します。
   // この操作を表すイベントを返します。
   template <typename KernelName, typename KernelType>
   event single_task(KernelType kernel);

   template <typename KernelName, typename KernelType,
             int Dims>
   event parallel_for(range<Dims> num_work_items,
                      KernelType kernel);

   template <typename KernelName, typename KernelType,
             int Dims>
   event parallel_for(nd_range<Dims> execution_range,
                      KernelType kernel);

   // 実行用にさまざまな形式のカーネルを送信します。
   // カーネルを実行する前に、指定されたイベントが完了するまで待機します。
   // カーネル操作を表すイベントを返します。
   template <typename KernelName, typename KernelType>
   event single_task(const std::vector<event>& events,
                     KernelType kernel);

   template <typename KernelName, typename KernelType,
             int Dims>
   event parallel_for(range<Dims> num_work_items,
                      const std::vector<event>& events,
                      KernelType kernel);

   template <typename KernelName, typename KernelType,
             int Dims>
   event parallel_for(nd_range<Dims> execution_range,
                      const std::vector<event>& events,
                      KernelType kernel);
};
```

図 2-20. キュークラスにおけるメンバー関数の簡略化された定義。これはハンドラークラス内の同等な関数に対する省略記法として機能します

アクション

図 2-18 のコードには、デバイス上で実行されるワークを定義する `parallel_for` が含まれています。`parallel_for` はキューに送信されたコマンドグループ（CG）内にあり、キューはワークが実行されるデバイスを定義します。コマンドグループ内には、次の 2 つのコードカテゴリーがあります:

（1）バッファーへのアクセサーを作成（第 3 章で説明）など、（2）で定義されたワークの実行をランタイムが安全に開始できるタイミングを定義する依存関係を設定する**ホストコード**。
（2）実行のためデバイスコードをキューに投入するか、コピーなどの手動のメモリー操作を実行する**アクションへの呼び出しを最大 1 回実行します**。

ハンドラークラスには、タスク・グラフ・ノードの実行時に実行されるアクションを定義するメンバー関数のセットが含まれています。図 2-21 はアクションを要約したものです。

ワークタイプ	アクション（ハンドラー・クラス・メソッド）	まとめ
デバイスコードの実行	`single_task`	デバイス関数の単一インスタンスを実行します。
	`parallel_for`	さまざまなワークサイズの組み合わせでデバイスコードを起動する複数のフォームが用意されています。
明示的なメモリー操作	`copy`	アクセサー、ポインター、`shared_ptr` で指定された場所の間でデータをコピーします。コピーは依存関係の追跡を含む DAG の一部として実行されます。
	`updat_host`	バッファー・オブジェクトのホスト・データ・バックアップの更新をトリガーします。
	`fill`	バッファー内のデータを指定された値に初期化します。

図 2-21. デバイスコードを呼び出したり、明示的なメモリー操作を実行するアクション

図 2-21 のアクションは、コマンドグループ内で 1 度だけ呼び出すことができます（複数のアクションを呼び出すとエラーになります）。また、`submit` 呼び出しごとにキューに送信できるコマンドグループは 1 つだけです。その結果、図 2-21 の 1 つの（あるいは可能性がない）操作がタスク・グラフ・ノードごとに存在し、ノードの依存関係が満たされ、実行しても安全であるとランタイムが判断した場合に実行されます。

コマンドグループには、カーネルの起動や明示的なメモリー操作などのアクションが最大 1 つ含まれている必要があります。

コードが将来非同期的に実行されるという考え方こそが、ホストプログラムの一部として CPU 上で実行されるコードと、依存関係が満たされたときに実行されるデバイスコードとの決定的な違いです。コマンドグループには通常、各カテゴリーのコードが含まれており、依存関係を定義するコードはホストプログラムの一部として実行され（ランタイムが依存関係を認識）、デバイスコードは依存関係が満たされると後に実行されます。

図 2-22 には 3 つのコードクラスがあります:

1. **ホストコード**: データバッファーの作成と管理、キューへのワークの送信など、アプリケーションを起動して、非同期実行タスクグラフに新しいノードを形成します。
2. **コマンドグループ内のホストコード**: このコードは、ホストコードが実行されているプロセッサー上で実行され、`submit` 呼び出しがリターンする前に直ちに実行されます。このコードは、たとえばアクセサーを作成してノードの依存関係を設定します。ここでは任意の CPU コードを実行できますが、ベスト・プラクティスとしては、ノード依存関係を構成するコードに限定することが推奨されます。

3. **アクション**: 図 2-21 にリストされている任意のアクションをコマンドグループに含めることができ、ノード要件が満たされたときに後で非同期的に実行されるワークを定義します（(2) で設定）。

```
#include <array>
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
 constexpr int size = 16;
 std::array<int, size> data;
 buffer B{data};

 queue q{};     // このキューに任意のデバイスを選択


 std::cout << "Selected device is: "
           <<
 q.get_device().get_info<info::device::name>()
      << "\n";

 q.submit([&](handler& h) {
   accessor acc{B, h};
   h.parallel_for(size,
                  [=](auto& idx) { acc[idx] = idx; });
 });

 return 0;
}
```

ホストコード

タスク・グラフ・ノードを設定する即時コード

デバイスコードは、依存関係が満たされたときに将来実行されます

ホストコード

図 2-22. デバイスコードの送信

アプリケーション内のコードがいつ実行されるかを理解するため、図 2-21 にリストされるデバイスコードの実行を開始するアクションに渡されるもの、または図 2-21 にリストされている明示的なメモリー操作は、将来、SYCL タスクグラフ（以降で説明）ノードの依存関係が満たされたときに*非同期*に実行されることに注意してください。そのほかのすべてのコードは、一般的な C++ コードと同様に、ホストプログラムの一部として*直ちに*実行されます。

タスク・グラフ・ノードの依存関係が満たされると、デバイスコードの実行が（非同期的に）開始されますが、その時点では開始される*保証はない*ことに注意することが重要です。デバイスコードの実行を確実に開始する唯一の方法は、ホストアクセサーやキュー待機操作などのメカニズムを使用して、ホストプログラムがデバイスコード実行の結果を待機（ブロック）することです。これについては、後の章で説明します。

このようなホスト・ブロッキング操作がない場合、SYCL および低レベルのランタイムは、デバイスコードの実行開始タイミングを決定し、電力や輻輳の最適化など、“できるだけ早く実行する” 以外の目的に合わせて最適化することがあります。

ホストタスク

一般に、キューに送信されたアクション (`parallel_for` など) によって実行されるコードはデバイスコードであり、多くのアーキテクチャーで効率良く実行するためいくつかの言語制限に従います。ただし、重要な例外が 1 つあります。これは `host_task` というハンドラーメソッドを通じてアクセスされます。この方法により、任意の C++ コードをタスクグラフ内のアクションとして送信し、タスクグラフの依存関係がすべて満たされた時点で、ホスト上で実行されます。

ホストタスクは、次の 2 つの理由により、一部のプログラムでは重要です。

1. `std::cout` や `printf` など、任意の C++ を含めることができます。これは、容易なデバッグ、OpenCL など低レベル API との相互運用性、あるいは既存のコードにおけるアクセラレーターの使用を段階的に有効にする上で重要となる場合があります。

2. ホストタスクは、ホストプログラムと同期して実行されるのではなく、タスクグラフの一部として非同期的に実行されます。ホストプログラムは追加のスレッドを起動したり、他のタスク並列化アプローチを使用したりできますが、ホストタスクは SYCL ランタイムの依存関係追跡メカニズムと統合されます。 これは非常に便利であり、デバイスコードとホストコードを交互に配置する場合にパフォーマンスが向上する可能性があります。

```
#include <array>
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
   queue q;
   int* A = malloc_shared<int>(N, q);

   std::cout << "Selected device: "
             << q.get_device().get_info<info::device::name>()
             << "\n";

   // 共有割り当ての値を初期化
   auto eA = q.submit([&](handler& h) {
       h.parallel_for(N, [=](auto& idx) { A[idx] = idx; });
   });

   // ホストタスクを使用して、タスクグラフの一部として
   // ホスト上の値を出力します。
   // depends_on は、前のデバイスコードが完了したことへの
   // 依存関係を定義するために使用されます。
   q.submit([&](handler& h) {
       h.depends_on(eA);
       h.host_task([=]() {
         for (int i = 0; i < N; i++)
            std::cout << "host_task @ " << i << " = " << A[i]
                      << "\n";
       });
  });

  // ホストプログラム内の共有データにアクセスする前に、
  // キュー内のワークが完了するまで待機します。
  q.wait();

  for (int i = 0; i < N; i++)
      std::cout << "main @ " << i << " = " << A[i] << "\n";

  free(A, q);

  return 0;
}
```

出力例:

```
Selected device: NVIDIA GeForce RTX 3060
host_task @ 0 = 0
host_task @ 1 = 1
host_task @ 2 = 2
host_task @ 3 = 3
main @ 0 = 0
main @ 1 = 1
main @ 2 = 2
main @ 3 = 3
```

図 2-23. 簡単な host_task

図 2-23 は、タスクグラフの依存関係が満たされたときに `std::cout` を使用してテキストを出力する簡単なホストタスクを示しています。ホストタスクは、ホストプログラムの残りの部分とは非同期に実行されることに注意してください。これは、タスクグラフの強力な機能であり、ホストプログラムとの対話なしに、安全なときに SYCL ランタイムがワークをスケジュールし、代わりに他のワークを続行できます。また、ホストタスクのコード本体は、デバイスコードに課せられる制限（第 10 章で説明）に従う必要がないことも留意してください。

図 2-23 の例は、デバイスコードの送信と後続のホストタスク間の依存関係を生成するイベント（第 3 章で説明）に基づいていますが、ホストタスクをアクセサー（第 3 章でも説明）と組み合わせて使用することもできます。これは、`target::host_task`（第 7 章）の特別なアクセサー・テンプレートのパラメーター化を通じて実現されます。

まとめ

この章では、キューの概要、キューが関連付けられるデバイスの選択、カスタム・デバイス・セレクターの作成方法について説明しました。また、依存関係が満たされたときにデバイス上で非同期的に実行されるコードと、C++ アプリケーションのホストコードの一部として実行されるコードについても触れました。第 3 章では、データ移動の制御方法について説明します。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 3 章　データ管理

スーパーコンピューターの設計者は、“獣に餌をやらなければならない（Feed the beast）” とよく嘆きます。“feed the beast” という表現は、大量の並列処理を使用したときに生み出されるコンピューターの “beast（獣）” を指し、それにデータを供給することが解決すべき重要な課題です。

異機種マシン上で SYCL プログラムを実行する場合、データが必要なときに必要な場所にあるようにする必要がありますが、大規模なプログラムでは、それは大変な作業です。既存の C++ プログラムでは、必要なデータ移動のすべてを管理する方法を考えるだけでも、悪夢のような作業になりかねません。

データを管理する 2 つの方法である、統合共有メモリー（USM）とバッファーについて詳しく説明します。USM はポインターベースであり、C++ プログラマーには馴染みのあるものです。バッファーは、より高レベルの抽象化を提供します。選択肢があるのは良いことです。

データの移動を制御する必要があり、この章ではまさにそれを実行する方法を解説します。

第 2 章では、コードが実行される場所を制御する方法について学びました。私たちのコードは入力データを必要とし、出力としてデータを生成します。コードは複数のデバイスで実行される可能性があり、それらのデバイスは必ずしもメモリーを共有するとは限らないため、データの移動を管理する必要があります。たとえ USM のようなデータ共有が行われていても、同期と一貫性を理解して管理する必要があります。

論理的な疑問として “なぜコンパイラーはすべてを自動的にやってくれないのか？” と思うかもしれません。多くの作業は、コンパイラーによって自動的に処理されますが、プログラマーとして自ら主張しなければ、パフォーマンスは、最適にはなりません。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_3

実際には、最高のパフォーマンスを得るには、異種プログラムを作成するときに、コードの配置（第 2 章）とデータの移動（この章）に注意する必要があります。

この章では、データを使用する順序の制御を含む、データ管理の概要について説明します。これは、コードの実行場所を制御する方法を示した前の章を補完するものです。この章では、コードの実行を指定した場所で、データを効率良く使用する方法を学びます。これは、アプリケーションを正しく実行するだけでなく、実行時間と電力消費を最小限に抑えるためにも重要です。

はじめに

データなしでは計算の意味がありません。計算を高速化する最大の目的は、より早く答えを導き出すことです。つまり、データ並列計算の最も重要なことの 1 つは、データにアクセスする方法であり、マシンにアクセラレーター・デバイスを導入すると、状況はさらに複雑になります。従来のシングルソケット CPU ベースのシステムでは、メモリーは 1 か所にしかありませんが、アクセラレーター・デバイスには、多くの場合、ホストから直接アクセスできない個別のメモリーが搭載されています。したがって、個別のデバイスをするサポート並列プログラミング・モデルでは、これら複数のメモリーを管理し、メモリー間でデータを移動するメカニズムを提供する必要があります。

この章では、データ管理のさまざまなメカニズムの概要について説明します。統合共有メモリーとバッファー抽象化のデータ管理を紹介し、カーネルの実行とデータ移動の関係について説明します。

データ移動の問題

歴史的に、並列プログラミングにおける共有メモリーモデルの利点の 1 つは、単一のメモリー共有ビューを提供することです。メモリーの単一ビュー持つことで、作業が簡素化されます。並列タスクからのメモリーアクセスに特別なことを行う必要はありません（データ競合を回避する適切な同期を除く）。一部の種類のアクセラレーター・デバイス（統合 GPU など）はホスト CPU とメモリーを共有しますが、多くのディスクリート・アクセラレーターは、図 3-1 に示すように、CPU とは独立したローカルメモリーを備えています。

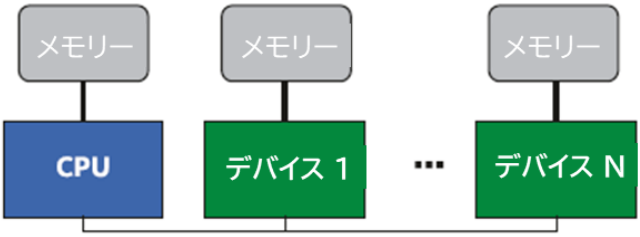

図 3-1. 複数の独立メモリー

デバイスローカルとデバイスリモート

デバイス上で実行されるプログラムは、通常、リモートメモリーではなく、デバイスに直接接続されたメモリーのデータを読み書きする方がパフォーマンス上有利です。デバイスに直接接続されたメモリーへのアクセスを*ローカル*アクセスと呼びます。別のデバイスのメモリーアクセスは*リモート*アクセスです。リモート・メモリー・アクセスは、帯域幅が狭くレイテンシーが長いデータリンクを介して移動する必要があるため、ローカルアクセスよりも低速になる傾向があります。これは、計算処理とそれに使用されるデータを同一場所に配置することが、しばしば有利であることを意味します。 これを実現するには、計算が行われる場所にデータを近づけるため、異なるメモリー間でデータをコピーまたは移動する必要があります。

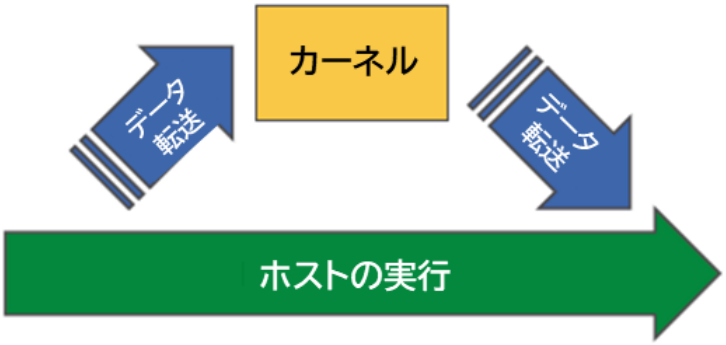

図 3-2. データ移動とカーネル実行

複数のメモリーを管理

複数のメモリー管理は、大きく分けて、プログラムによって*明示的*に行う方法と、SYCL ランタイム・ライブラリーによって*暗黙的*に行う 2 つの方法があります。それぞれに長所と短所があり、状況や好みに応じてどちらかを選択します。

明示的なデータ移動

複数のメモリーを管理する方法の 1 つは、異なるメモリー間でデータを明示的にコピーすることです。図 3-2 は、カーネルが必要とするデータを最初にホストメモリーからアクセラレーターのメモリーにコピーする、独立したアクセラレーターを備えたシステムを示しています。カーネルが結果を計算した後、ホストプログラムがそのデータを使用する前に、結果をホストにコピーする必要があります。

明示的なデータ移動の利点は、異なるメモリー間でデータが転送されるタイミングを完全に制御できることです。これは、特定のハードウェアで最高のパフォーマンスを得るには、計算とデータ転送のオーバーラップが不可欠となる場合があるため重要です。

明示的なデータ移動の欠点は、すべてのデータ移動を指定することが煩雑で、エラーが発生しやすくなることです。不必要なデータを転送したり、カーネルが計算を開始する前にすべてのデータ転送できない場合、誤った結果が生成される可能性があります。

最初からすべてのデータ移動を正しく行うことは、非常に時間のかかる作業になる可能性があります。

暗黙的なデータ移動

プログラムの制御による明示的なデータ移動の代替手段として、並列ランタイムまたはドライバーによって制御される暗黙的なデータ移動があります。この場合、異なるメモリー間で明示的なコピーを行う代わりに、並列ランタイムはデータが使用される前に適切なメモリーに転送する責任を負います。

暗黙的なデータ移動の利点は、デバイスに直接接続された高速メモリーをアプリケーションが活用する労力が少なくなることで、すべての面倒な作業はランタイムによって自動的に行われます。また、ランタイムはデータを転送するタイミングと転送するデータ量を自動的に識別するため、プログラムでエラーが発生する可能性も減ります。

暗黙的なデータ移動の欠点は、ランタイムの暗黙的なメカニズムの動作を制御できない、または制御が制限されることです。ランタイムは機能的な正確性を提供しますが、データ転送と計算処理のオーバーラップを最大化する最適な方法でデータを移動しない可能性があり、プログラムのパフォーマンスに影響する場合があります。

適切な方法の選択

プログラムに最適な方法を選択するには、さまざまな要因を考慮する必要があります。プログラム開発の異なる段階では、それぞれに適した方法が存在する可能性があります。プログラムの各部分に対して、明示的な手法と暗黙的な手法を組み合わせて使うことが最善であると判断することもできます。アプリケーションを新しいデバイスに移植する作業を簡素化するため、暗黙的なデータ移動から始めることを選択します。アプリケーションのパフォーマンス・チューニングを開始すると、パフォーマンスが重要なコード部分では、暗黙的なデータ移動を明示的なデータ移動に置き換えることから始めます。

以降の章では、パフォーマンスの最適化にデータ転送を計算とオーバーラップする方法について説明します。

USM、バッファーおよびイメージ

メモリーを管理する抽象化には、**統合共有メモリー**(USM)、**バッファー**、および**イメージ**の 3 つの方法があります。USM は、C/C++ プログラマーには馴染み深いポインターベースのアプローチです。USM の利点の 1 つは、ポインターを操作する既存の C++ コードとの統合が容易なことです。バッファーは、バッファー・テンプレート・クラスによって表されるように、1 次元、2 次元、または 3 次元の配列を記述します。ホストまたはデバイスからアクセスできるメモリーへの抽象化されたビューを提供します。バッファーはプログラムによって直接アクセスされるのではなく、アクセサー・オブジェクトを通じて使用されます。イメージは、イメージ処理に固有の追加機能を提供する特殊なタイプのバッファーとして機能します。この機能には、特殊なイメージ形式のサポート、サンプラー・オブジェクトを使用したイメージの読み取りなどが含まれます。バッファーとイメージは多くの問題を解決する強力な抽象化ですが、バッファーまたはアクセサーを受け入れるため既存のコードのすべてのインターフェイスを書き直すには時間を要することがあります。バッファーとイメージのインターフェイスはほぼ同じであるため、この章では USM とバッファーのみに注目します。

統合共有メモリー

USM はデータ管理に使用できる方法の 1 つです。USM は、ポインターベースのアプローチであり、C/C++ プログラマーがデータ割り当てに使用する `malloc` や `new` と使い方は同じです。USM は、ポインターを多用する既存の C/C++ コードの移植を容易にします。USM をサポートするデバイスは、統合された仮想アドレス空間をサポートします。統合された仮想アドレス空間を持つということは、ホスト上の USM 割り当てルーチンが返す任意のポインター値が、デバイス上でも有効なポインター値になることを意味します。

“デバイスのバージョン” を取得するためホストポインターを手動で変換する必要はなく、ホストとデバイスの両方で同じポインター値が使用されます。

USM の詳細な説明については、第 6 章を参照してください。

ポインターを介してメモリーにアクセス

システムにホストメモリーとデバイスに接続されたローカルメモリーの両方がある場合、すべてのメモリーが同じように作成されるわけではないため、USM ではデバイス、ホスト、共有の異なる 3 つの割り当てタイプを定義します。すべてのタイプの割り当てはホスト上で実行されます。図 3-3 は、それぞれの割り当てタイプの特性をまとめたものです。

割り当てタイプ	説明	ホストでアクセス可能	デバイスでアクセス可能	割り当て場所
デバイス	デバイスメモリーに割り当て	いいえ	はい	デバイス
ホスト	ホストメモリーに割り当て	はい	はい	ホスト
共有	ホストとデバイス間で共有される割り当て	はい	はい	移動可能

図 3-3. USM 割り当てタイプ

デバイス割り当ては、デバイスに接続されたメモリーで行われます。この割り当てはデバイス上で読み書き可能ですが、ホストからは直接アクセスできません。ホストメモリー内の通常の割り当てと、デバイスの割り当て間でデータを移動するには、明示的なコピー操作が必要です。

ホスト割り当ては、ホストとデバイスの両方からアクセス可能なホストメモリーで行われます。これは、ホストコードとデバイスカーネルの両方で同じポインター値が有効であることを意味します。ただし、このようなポインターにアクセスする場合、データは常にホストメモリーから取得されます。

デバイス上でアクセスされた場合、データはホストからデバイスのローカルメモリーに移行されることはありません。代わりに、データは通常、デバイスをホストに接続する PCI Express（PCI-E）などのバスを介して送信されます。

共有割り当ては、ホストとデバイスの両方からアクセスできます。この点ではホスト割り当てと似ていますが、データがホストメモリーとデバイスのローカルメモリー間で移行できる点が異なります。つまり、移行後のデバイスへのアクセスは、レイテンシーが高い接続を介してホストメモリーにリモートアクセスするのではなく、はるかに高速なデバイスのローカルメモリーから行われます。通常、これはランタイムのメカニズムと、ユーザーからは見えない低レベルのドライバーによって制御されます。

USM とデータ移動

USM は明示的および暗黙的なデータ移動の両方をサポートしており、異なる割り当てタイプは異なる方法でマップされます。デバイス割り当てでは、ホストとデバイス間でデータを明示的に移動する必要がありますが、ホストと共有の割り当てでは暗黙的なデータの移動が行われます。

明示的な USM のデータ移動

USM による明示的なデータ移動は、デバイスの割り当てと、キューおよびハンドラークラスに存在する特別な `memcpy()` によって実現されます。ホストからデバイスへ、またはデバイスからホストへデータを転送するため、`memcpy()` 操作（アクション）をキューにエンキューします。

図 3-4 には、デバイス割り当てを操作する 1 つのカーネルが含まれています。データは、カーネルが `memcpy()` 操作を使用して実行する前と後に、`host_array` と `device_array` 間でコピーされます。キューで `wait()` を呼び出すと、カーネルが実行される前にデバイスへのコピーが完了し、データがホストにコピーされる前にカーネルが完了していることが保証されます。これらの呼び出しをどのように排除できるかは、この章の後半で説明します。

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;


int main() {
   queue q;

   std::array<int, N> host_array;
   int *device_array = malloc_device<int>(N, q);

   for (int i = 0; i < N; i++) host_array[i] = N;

   // この例を簡略化する方法については後で説明します
   q.submit([&](handler &h) {
     // host_array を device_array にコピー
     h.memcpy(&host_array[0], device_array, N * sizeof(int));
   });
   q.wait();

   q.submit([&](handler &h) {
     h.parallel_for(N, [=](id<1> i) { device_array[i]++; });
   });
   q.wait();

   q.submit([&](handler &h) {
     // device_array を host_array にコピー
     h.memcpy(&host_array[0], device_array, N * sizeof(int));
   });
   q.wait();

   free(device_array, q);
   return 0;
}
```

図 3-4. USM 明示的なデータ移動

暗黙的な USM のデータ移動

USM による暗黙的なデータ移動は、ホストと共有割り当てによって実現されます。このタイプの割り当てでは、ホストとデバイス間のデータ移動にコピー操作を明示的に挿入する必要はありません。代わりに、カーネル内のポインターにアクセスすることで、必要なデータ移動はプログラマーの介入なし自動的に行われます（デバイスが割り当てをサポートしている限り）。

これにより、既存のコードの移植が大幅に簡素化されます。malloc または new を適切な USM 割り当て関数（およびメモリーの割り当てを解除する free の呼び出し）に置き換えるだけで、すべてが機能するようになります。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   queue q;
   int *host_array = malloc_host<int>(N, q);
   int *shared_array = malloc_shared<int>(N, q);

   for (int i = 0; i < N; i++) {
       // ホスト上の host_array を初期化
       host_array[i] = i;
   }

   // この例を簡略化する方法については後で説明します
   q.submit([&](handler  &h) {
       h.parallel_for(N, [=](id<1> i) {
         // デバイス上の shared_array と host_array にアクセス
         shared_array[i] = host_array[i] + 1;
       });
   });
   q.wait();

   for (int i = 0; i < N; i++) {
       // ホスト上の shared_array にアクセス
       host_array[i] = shared_array[i];
   }

   free(shared_array, q);
   free(host_array, q);
   return 0;
}
```

図 3-5. USM 暗黙的なデータ移動

図 3-5 では、それぞれホスト割り当てと共有割り当てで 2 つの配列、host_array と shared_array を作成します。ホストコードではホスト割り当てと共有割り当ての両方に直接アクセスできますが、ここでは host_array のみを初期化します。

同様に、カーネル内部で直接アクセスして、データをリモートで読み取ることもできます。ランタイムは、カーネルが `shared_array` にアクセスする前に `shared_array` がデバイス上で使用可能であること、および使用後にホストコードによって読み取られたときに `shared_array` が元に戻されることを保証します。これらはすべてプログラマーの介入なしで行われます。

バッファー

データ管理に提供されるもう 1 つの抽象化は、バッファー・オブジェクトです。バッファーは、特定の C++ 型の 1 つ以上のオブジェクトを表すデータの抽象化です。バッファー・オブジェクトの要素は、スカラーデータ型 (`int`、`float` または `double` など)、ベクトルデータ型 (第 11 章)、またはユーザー定義のクラスや構造体にできます。SYCL 2020 では、許容される型のセットへの追加によって、簡単にコピー可能な概念を拡張した、*デバイスコピー可能*という新しい概念が定義されています。特に、`std::array`、`std::pair`、`std::tuple`、`std::span` など一般的な C++ クラスのテンプレート化された型自体がデバイスコピー可能である場合、それらの型を使用して構築された C++ クラスの特殊化もデバイスコピー可能になります。バッファーで使用する前に、データ型がデバイスでコピー可能であることを確認してください。

バッファー自体は単一のオブジェクトですが、バッファーによってカプセル化される C++ 型は複数のオブジェクトを含む配列になる可能性があります。バッファーは特定のメモリーアドレスではなくデータ・オブジェクトを表すため、通常の C++ 配列のように直接アクセスすることはできません。実際、バッファー・オブジェクトは、パフォーマンス上の理由から、複数の異なるデバイス上、または同じデバイス上の複数の異なるメモリー位置にマップされることがあります。代わりに、バッファーの読み取りと書き込みには、*アクセサー*・オブジェクトを使用します。

バッファーの詳細については、第 7 章を参照してください。

バッファーを作成

バッファーはさまざまな方法で作成できます。最も簡単な方法は、バッファーサイズを指定する範囲を持つ新しいバッファーを単純に構築することです。ただし、この方法でバッファーを作成してもデータは初期化されません。つまり、バッファーから有用なデータを読み取る前に、まず他の手段でバッファーを初期化する必要があります。

ホスト上の既存のデータからバッファーを作成することもできます。これは、既存のホスト割り当てへのポインター、`InputIterators` のセット、または特定のプロパティーを持つコンテナーのいずれかを受け取る複数のコンストラクターの 1 つを呼び出すことによって行われます。バッファー構築中に、データは既存のホスト割り当てからバッファー・オブジェクトのホストメモリーにコピーされます。SYCL の相互運用機能を使用して、バックエンド固有のオブジェクト (OpenCL `cl_mem` オブジェクトなど) からバッファーを作成することもできます。詳細については、第 20 章の相互運用性を参照してください。

バッファーのアクセス

バッファーは、ホストおよびデバイスから直接アクセスできません (ここで説明されていない、高度であまり使用されないメカニズムを使用する場合を除く)。代わりに、バッファーの読み取りと書き込みには、アクセサー・オブジェクトを使用します。アクセサーは、バッファー内のデータをどのように使用するか情報をランタイムに提供し、データの移動を正しく配置できるようにします。

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   std::array<int, N> my_data;
   for (int i = 0; i < N; i++) my_data[i] = 0;

   {
     queue q;
     buffer my_buffer(my_data);

     q.submit([&](handler &h) {
       // デバイス上のバッファーを更新するための
       // アクセサーを作成
       accessor my_accessor(my_buffer, h);

       h.parallel_for(N, [=](id<1> i) { my_accessor[i]++; });
     });

     // ホストアクセサーを作成
     host_accessor host_accessor(my_buffer);

     for (int i = 0; i < N; i++) {
        // ホスト上の myBuffer にアクセス
        std::cout << host_accessor[i] << " ";
     }
     std::cout << "\n";
   }

   // myData はスコープ終了時に
   // myBuffer が破棄されると更新されます
   for (int i = 0; i < N; i++) {
      std::cout << my_data[i] << " ";
   }
   std::cout << "\n";
}
```

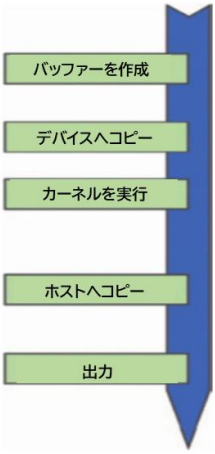

図 3-6. バッファーとアクセサー

アクセスモード	説明
read	読み取り専用アクセス。
write	書き込み専用アクセス。以前の内容は破棄されます。
read_write	読み取りと書き込みアクセス。

図 3-7. バッファーのアクセスモード

アクセスモード

アクセサーを作成するときに、それをどのように使用するかランタイムに通知して、最適化向けに詳細情報を提供できます。これは、*アクセスモード*を指定して行います。アクセスモードは、図 3-7 に示す `access_mode` 列挙型で定義されます。図 3-6 に示すコード例では、アクセサー `my_accessor` はデフォルトのアクセスモード `access_mode::read_write` で作成されています。これによって、ランタイムは、`my_accessor` を介してバッファーの読み取りと書き込みの両方を行うことを認識します。アクセスモードは、ランタイムが暗黙的なデータ移動を最適化する方法を示します。例えば、`access_mode::read` はカーネルが実行を開始する前に、デバイスでデータが利用可能でなければならないことをランタイムに通知します。カーネルがアクセサーを介してデータを読み取るだけの場合、データは変更されないため、カーネルが完了した後にデータをホストにコピーする必要はありません。同様に、`access_mode::write` は、バッファーの内容を更新して計算が完了すると、結果をホストに書き戻す必要があることをランタイムに知らせます。

適切なモードでアクセサーを作成すると、プログラム内でデータをどのように使用するか詳細な情報がランタイムに提供されます。ランタイムは、アクセサーを介してデータを利用する順序を決定しますが、このデータを使用してカーネルのスケジュールやデータの移動を最適化することもあります。アクセスモードと最適化タグについては、第 7 章で詳しく説明します。

データの使用を順序付け

カーネルは、実行のために送信される非同期タスクであると見なすことができます。これらのタスクは、デバイス上で実行するようにスケジュールされているキューに送信する必要があります。

多くの場合、正しい結果が計算されるように、カーネルは特定の順序で実行する必要があります。正しい結果を得るため、タスク A をタスク B の前に実行する必要がある場合、タスク A とタスク B の間に*依存関係*[10] が存在すると言います。

ただし、スケジュールの必要があるタスクの形式はカーネルだけではありません。カーネルがアクセスするデータは、カーネルが実行を開始する前にデバイス上で使用可能である必要があります。これらのデータ依存関係により、あるデバイスから別のデバイスへのデータ転送という形で追加のタスクが生成されることがあります。データ転送タスクは、明示的にコード化されたコピー操作、または一般的にはランタイムによって実行される暗黙的なデータ移動のいずれかです。

プログラム内のすべてのタスクとそれらの間の依存関係を取得すると、この情報をグラフとして可視化できます。このタスクグラフは、具体的には、ノードがタスク、エッジが依存関係である有向非巡回グラフ（DAG）です。依存関係が一方向であるため、グラフは*有向*です。つまり、タスク A はタスク B の前に実行する必要があります。グラフは*非循環*です。これは、ノードからそれ自体に戻るサイクルやパスが含まれていないためです。

図 3-8 では、タスク A はタスク B と C の前に実行する必要があります。同様に、タスク B と C はタスク D の前に実行する必要があります。B と C は相互に依存していないため、タスク A が既に実行されている限り、ランタイムはそれらを任意の順序で（または並列に）実行できます。したがって、このグラフの可能な正当な順序は、A ⇒ B ⇒ C ⇒ D、A ⇒ C ⇒ B ⇒ D、さらに B と C が同時に実行できる場合は A ⇒ {B,C} ⇒ D となります。

[10] 他の英文テキストでは、「dependence」と「dependences」が「dependency」と「dependencies」と表記される場合があることに注意してください。これらは同じ意味ですが、データフロー解析に関するいくつかの重要な論文で使用されているスペルを優先します。https://dl.acm.org/doi/pdf/10.1145/75277.75280 と https://dl.acm.org/doi/pdf/10.1145/113446.113449 を参照してください。

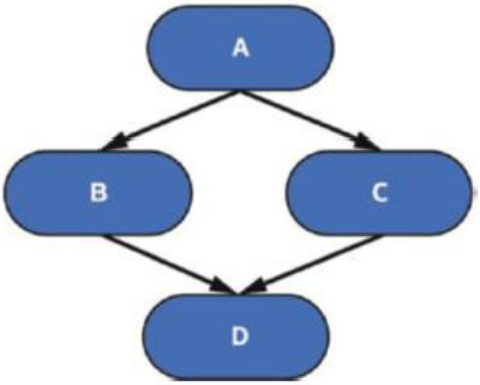

図 3-8. 単純なタスクグラフ

タスクは、すべてのタスクのサブセットと依存関係を持つ場合があります。これらのケースでは、正確性に重要な依存関係のみを指定します。この柔軟性により、実行時にタスクグラフの実行順序を最適化する自由度が与えられます。図 3-9 では、図 3-8 の前のタスクグラフを拡張して、タスク E と F を追加します。E は F の前に実行する必要があります。ただし、タスク E と F は、ノード A、B、C、および D とは依存関係がありません。これにより、ランタイムは実行可能なタスクの順序付けを複数から選択できるようになります。

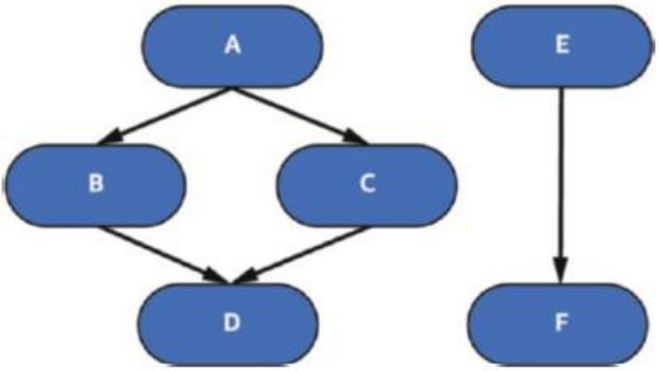

図 3-9. 依存関係が互いに素なタスクグラフ

キュー内でカーネル起動などタスクの実行をモデル化するには、2 つの異なる方法があります。キューは、送信された順序でタスクを実行することも、定義した依存関係に従って任意の順序でタスクを実行することもできます。正しい順序付けに必要な依存関係を定義するメカニズムがいくつかあります。

インオーダー・キュー

タスクを順序付ける最も簡単な方法は、タスクをインオーダー・キュー・オブジェクトに送信することです。インオーダー・キューは、図 3-10 に示すように、タスクを送信された順序で実行します。タスクの順序付けが直感的であるため、インオーダー・キューには単純であるという利点がありますが、独立したタスク間に依存関係が存在しない場合でもタスクをシリアル化するという欠点があります。インオーダー・キューは、シンプルで直感的であり、実行順序が決定論的であり、多くのコードに適しているため、アプリケーションの開発初期に役立ちます。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 4;

int main() {
   queue q{property::queue::in_order()};

   q.submit([&](handler& h) {
     h.parallel_for(N, [=](id<1> i) { /*...*/ }); // タスク A
   });
   q.submit([&](handler& h) {
     h.parallel_for(N, [=](id<1> i) { /*...*/ }); // タスク B
   });
   q.submit([&](handler& h) {
     h.parallel_for(N, [=](id<1> i) { /*...*/ }); // タスク C
   });

   return 0;
}
```

図 3-10. インオーダー・キューの使い方

アウトオブオーダー・キュー

キュー・オブジェクトはアウトオブオーダー・キューであるため（インオーダー・キュー・プロパティーを使用して作成されない限り）、キュー・オブジェクトに送信されたタスクの順序付け方法を提供する必要があります。キューは、タスク間の依存関係を実行時に通知することで、タスクを順序付けます。これらの依存関係は、*コマンドグループ*を使用して明示的または暗黙的に指定できます。これらについては、次のセクションで個々に検討します。

コマンドグループは、タスクとその依存関係を指定するオブジェクトです。コマンドグループは通常、キュー・オブジェクトの `submit()` メソッドに引数として渡される C++ ラムダ式として記述されます。このラムダの唯一のパラメーターは、ハンドラー・オブジェクトへの参照です。ハンドラー・オブジェクトは、アクションを指定し、アクセサーを作成し、依存関係を指定するためにコマンドグループ内で使用されます。

イベントとの明示的な依存関係

タスク間の明示的な依存関係は、タスク A がタスク B の前に実行されなければならない、これまで見てきた例（図 3-8）のようになります。このように依存関係を表現すると、計算によってアクセスされるデータではなく、発生する計算に基づく明示的な順序付けに重点が置かれます。

バッファーを使用するコードは、ほとんどの依存関係をアクセサー経由で表現するため、計算間の依存関係を表現することは、主に USM を使用するコードに適していることに注意してください。図 3-4 および 3-5 では、続行する前に、以前に送信されたすべてのタスクが完了するまでキューを待機するように指示しています。代わりに、*イベント*・オブジェクトを通じてタスクの依存関係を表現できます。コマンドグループをキューに送信すると、`submit()` メソッドはイベント・オブジェクトを返します。このイベントは 2 つの方法で使用できます。

まず、イベントで `wait()` メソッドを明示的に呼び出すことで、ホストを介して同期できます。これにより、ランタイムは、イベントを生成したタスクの実行が完了するまで待機してから、ホストプログラムの実行を続行します。

イベントを明示的に待機することはアプリケーションのデバッグに非常に役立ちますが、wait() はホストスレッド上のすべての実行を停止するため、タスクの非同期実行を過度に制限する可能性があります。同様に、キュー・オブジェクトで wait() を呼び出すこともできます。これにより、キューに投入されたすべてのタスクが完了するまで、ホスト上の実行がブロックされます。キューに投入されたタスクによって返されるすべてのイベントを追跡しない場合、これが便利なツールになります。

ここで、イベントが活用される第 2 の方法について説明します。ハンドラークラスには、depends_on() というメソッドが含まれています。このメソッドは、単一のイベントまたはイベントのベクトルのいずれかを受け入れ、送信されるコマンドグループで、コマンドグループ内のアクションを実行する前に指定されたイベントが完了する必要があることをランタイムに通知します。図 3-11 は、depends_on() を使用したタスクの順序付けの例を示しています。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 4;

int main() {
  queue q;

  auto eA = q.submit([&](handler &h) {
    h.parallel_for(N, [=](id<1> i) {  /*...*/  }); // タスク A
  });
  eA.wait();
  auto eB = q.submit([&](handler &h) {
    h.parallel_for(N, [=](id<1> i) {  /*...*/  }); // タスク B
  });
  auto eC = q.submit([&](handler &h) {
    h.depends_on(eB);
    h.parallel_for(N, [=](id<1> i) {  /*...*/  }); // タスク C
  });
  auto eD = q.submit([&](handler &h) {
    h.depends_on({eB, eC});
    h.parallel_for(N, [=](id<1> i) {  /*...*/  }); // タスク D
 });

 return 0;
}
```

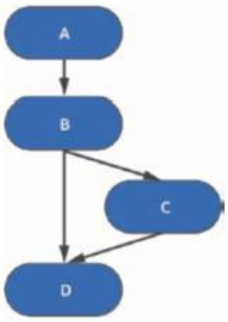

図 3-11. イベントと depends_on を使用

アクセサーとの暗黙的な依存関係

タスク間の暗黙的な依存関係は、データの依存関係から生じます。タスク間のデータ依存関係には、図 3-12 に示す 3 つの形式があります。

依存関係タイプ	説明
書き込み後の読み取り（RAW）	タスク B がタスク A によって計算されたデータを読み取る必要があるときに発生します。
読み取り後の書き込み（WAR）	タスク A によって読み取られたデータをタスク B が上書きするときに発生します。
書き込み後の書き込み（WAW）	タスク B がタスク A によって書き込まれたデータを上書きするときに発生します。

図 3-12. データ依存の 3 つの形式

データの依存性は、アクセサーとプログラム順序という 2 つの方法でランタイムに表現されます。ランタイムがデータの依存関係を適切に計算するには、両方を使用する必要があります。これを、図 3-13 と 3-14 に示します。

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   std::array<int, N> a, b, c;
   for (int i = 0; i < N; i++) {
       a[i] = b[i] = c[i] = 0;
   }

   queue q;

   // この例を簡略化する方法については後で説明します
   buffer a_buf{a};
   buffer b_buf{b};
   buffer c_buf{c};

  q.submit([&](handler &h) {
      accessor a(a_buf, h, read_only);
       accessor b(b_buf, h, write_only);
       h.parallel_for(  // computeB
         N, [=](id<1> i) { b[i] = a[i] + 1; });
  });

  q.submit([&](handler &h) {
     accessor a(a_buf, h, read_only);
     h.parallel_for(    // readA
         N, [=](id<1> i) {
            // 例としてのみ有用
            int  data = a[i];
         });
   });

   q.submit([&](handler &h) {
       // バッファー B の RAW
       accessor b(b_buf, h, read_only);
       accessor c(c_buf, h, write_only);
       h.parallel_for(  // computeC
         N, [=](id<1> i) { c[i] = b[i] + 2; });
   });

   // ホストで C を読み取り
   host_accessor host_acc_c(c_buf, read_only);
   for (int i = 0; i < N; i++) {
       std::cout << host_acc_c[i] << " ";
   }
   std::cout << "\n";
   return 0;
}
```

図 3-13. 書き込み後の読み取り（RAW）

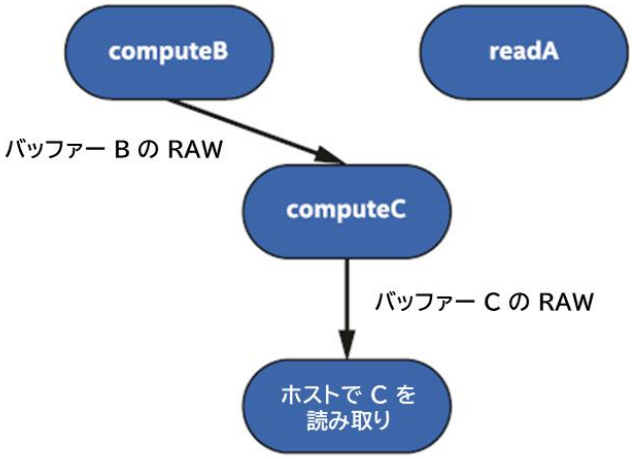

図 3-14. RAW タスクグラフ

図 3-13 と 3-14 では、3 つのカーネル（computeB、readA、computeC）を実行し、最終結果をホストで読み戻しています。カーネル computeB のコマンドグループは、2 つのアクセサー a と b を作成します。これらのアクセサーは、デフォルトのアクセスモード access_mode::read_write を使用しないように最適化するため、アクセスタグ read_only と write_only を使用します。アクセスタグについては、第 7 章で詳しく説明します。カーネル computeB はバッファー a_buf を読み取り、バッファー b_buf に書き込みます。カーネルが実行を開始する前に、バッファー a_buf をホストからデバイスにコピーする必要があります。

カーネル readA は、バッファー a_buf の読み取り専用アクセサーも作成します。カーネル readA は、カーネル computeB の後に送信されるため、読み取り後の読み取り（RAR）シナリオが作成されます。ただし、RAR はランタイムに制限を課さないため、カーネルは任意の順序で自由に実行できます。実際、ランタイムはカーネル readA をカーネル computeB の前に実行する傾向があり、あるいは両方を同時に実行することもあるかもしれません。どちらもバッファー a_buf をデバイスにコピーする必要がありますが、カーネル computeB では、既存の値が computeB によって上書きされず、後のカーネルで使用される可能性がある場合に備えて、バッファー b_buf もコピーする必要があります。

これは、バッファー b_buf のデータ転送が行われている間にランタイムがカーネル readA を実行できることを意味し、また、カーネルがバッファーに書き込むだけの場合でも、バッファー内のすべての値がカーネルによって書き込まれる保証がないため、バッファーの元の内容がデバイスに移動される可能性があることを示しています（このような場合に最適化を可能にするタグについては、第 7 章を参照してください）。

カーネル computeC は、カーネル computeB で計算されたバッファー b_buf を読み取ります。カーネル computeB を送信した後にカーネル computeC を送信したため、カーネル computeC はバッファー b_buf に対して RAW データ依存関係を持ちます。RAW 依存関係は、正しい結果を計算するにはデータが 1 つの計算から別の計算に流れる必要があるため、真の依存関係またはフロー依存関係とも呼ばれます。最後に、カーネルが終了した後にホストが C を読み取る必要があるため、カーネル computeC とホスト間のバッファー c_buf に対する RAW 依存関係も作成します。これにより、ランタイムはバッファー c_buf をホストに戻すためにコピーを強制されます。デバイスのバッファー a_buf への書き込みは発生しなかったため、ホストにはすでに最新のコピーがあり、ランタイムはそのバッファーをホストにコピーする必要はありません。

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   std::array<int,N> a, b;
   for (int i = 0; i < N; i++) {
        a[i] = b[i] = 0;
   }

   queue q;
   buffer a_buf{a};
   buffer b_buf{b};

   q.submit([&](handler &h) {
      accessor a(a_buf, h, read_only);
      accessor b(b_buf, h, write_only);
      h.parallel_for(    // computeB
              N, [=](id<1> i) { b[i] = a[i] + 1; });
   });

   q.submit([&](handler &h) { // バッファー A の WAR
       accessor a(a_buf, h, write_only);
       h.parallel_for(   // rewriteA
              N, [=](id<1> i) { a[i] = 21 + 21; });
   });

   q.submit([&](handler &h) {
      // バッファー B の WAW
      accessor b(b_buf, h, write_only);
      h.parallel_for(    // rewriteB
              N, [=](id<1> i) { b[i] = 30 + 12; });
   });

   host_accessor host_acc_a(a_buf, read_only);
   host_accessor host_acc_b(b_buf, read_only);
   for (int i = 0; i < N; i++) {
      std::cout << host_acc_a[i] << " " << host_acc_b[i]
                << " ";
   }
   std::cout << "\n";
   return 0;
}
```

図 3-15. 読み取り後の書き込み（WAR）と書き込み後の書き込み（WAW）

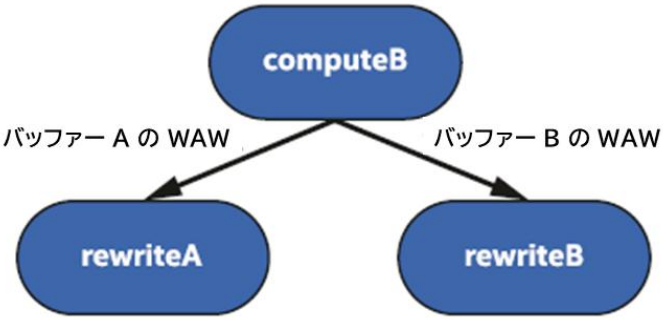

図 3-16. WAR と WAW タスクグラフ

図 3-15 と 3-16 では、computeB、rewriteA、rewriteB の 3 つのカーネルを再度実行しています。カーネル computeB は再びバッファー a_buf を読み取ってバッファー b_buf に書き込み、カーネル rewriteA はバッファー a_buf に書き込み、カーネル rewriteB はバッファー b_buf に書き込みます。カーネルの準備が整う前に転送する必要があるデータは僅かであるため、理論的にはカーネル rewriteA はカーネル computeB よりも早く実行できますが、バッファー a_buf に対する WAR 依存関係があるため、カーネル computeB が完了するまで待機する必要があります。

この例では、カーネル computeB はホストからの A の元の値を必要としますが、カーネル computeB の前にカーネル rewriteA を実行すると誤った値が読み取られます。WAR 依存関係はアンチ依存関係とも呼ばれます。RAW 依存関係はデータが正しい方向に適切に流れることを保証し、WAR 依存関係は既存の値が読み取られる前に上書きされないことを保証します。カーネルの書き換え関数で見られる WAW のバッファー b_buf への依存性も同様です。カーネル computeB と rewriteB の間にバッファー b_buf の読み取りが送信された場合、タスクを適切に順序付ける RAW および WAR 依存関係が発生します。ただし、この例では最終的にデータをホストに書き戻す必要があるため、カーネル rewriteB とホストの間には暗黙的な依存関係が存在します。この書き戻しの原因については、第 7 章で詳しく説明します。WAW 依存関係（出力依存関係とも呼ばれます）により、ホスト上の最終出力が正しいことが保証されます。

データ管理方法の選択

アプリケーションに適したデータ管理方法の選択は、主に個人の好みによって決まります。 実際、最初は 1 つの方法から始めて、プログラムが成熟するにつれて別の方法に切り替えることもあります。ただし、ニーズに合った方法を選択するのに役立つガイドラインがいくつかあります。

最初に決定すべきことは、明示的なデータ移動を使用するか、暗黙的なデータ移動を使用するかということです。これは、プログラムに必要な処理に大きく影響するからです。暗黙的なデータ移動は、すべてのデータ移動が自動的に処理され、開発者は計算の表現に集中できるため、一般的に始めるのが簡単です。

最初からすべてのデータ移動を完全に制御したい場合、USM デバイス割り当てを使用明示的なデータ移動が適しています。ホストとデバイス間に必要なコピーをすべて追加するだけです。

暗黙的なデータ移動を選択する場合でも、バッファーを使用するか、USM ホストまたは共有ポインターを使用するか選択できます。繰り返しになりますが、この選択は個人の好みの問題ですが、どちらかを選択する際に役立つ質問がいくつかあります。ポインターを使用する既存の C/C++ プログラムを移植する場合、コードをほとんど変更する必要がないため、USM が適しているかもしれません。データ表現の優先順位が定まらない場合、カーネル間の依存関係をどのように表現するかという別の質問があります。 カーネル間でのデータ依存関係を優先する場合は、バッファーを使用してください。依存関係を、ある計算を別の計算の前に実行するものとして考え、それをインオーダー・キューまたは明示的なイベントを使用して表現したり、カーネル間で待機する場合は USM を選択します。

USM ポインター（明示的または暗黙的なデータ移動のどちらか）を使用する場合、使用するキューのタイプを選択できます。インオーダー・キューはシンプルで直感的ですが、実行時間を制限し、パフォーマンスを低下させる可能性があります。アウトオブオーダー・キューはより複雑ですが、実行順序を変更したり、実行を重複させたりするための自由度がランタイムに与えられます。

プログラムにカーネル間の複雑な依存関係がある場合、アウトオブオーダー・キュー・クラスが適切な選択となります。プログラムが単純に多数のカーネルを次々に実行する場合、インオーダー・キューの方が適しています。

ハンドラークラス: 主要メンバー

ハンドラークラスを使用するさまざまな方法を示しました。図 3-17 と 3-18 では、この重要なクラスの主要メンバーについてさらに詳しく説明します。これらのメンバーはまだすべて使用していませんが、本書の後半で使用します。ここも、それらを配置するのに適した場所です。

密接に関連するクラスであるキュークラスについても、第 2 章の最後で説明されています。

```
class handler {
   ...
   // このコマンド グループで定義されたアクションが実行される前に
   //   完了する必要があるイベントを指定
    void depends_on({event / std::vector<event> & });

   // Src から Dest に memcpy をエンキュー
   // Count バイトがコピーされます
   void memcpy(void* Dest, const void* Src, size_t Count);

   // Src から Dest に memcpy をエンキュー
   // Count 要素がコピーされます
   template <typename T>
   void copy(const T* Src, T* Dest, size_t Count);

  // 指定されたポインターに対して memset 操作をエンキュー
  // Value の最初のバイトを Count バイトに書き込みます
  void memset(void* Ptr, int Value, size_t Count)

      // 指定されたポインターに対して fill 操作をエンキュー
      // パターンを Ptr に Count 回入力
      template <typename T>
      void fill(void* Ptr, const T& Pattern, size_t Count);


  // 実行のために 1 つのワーク項目のカーネルを送信
  template <typename KernelName, typename KernelType>
  void single_task(KernelType KernelFunc);

  // NumWork 個のワーク項目を含むカーネル
  // を実行するため送信
  template <typename KernelName, typename KernelType,
            int Dims>
  void parallel_for(range<Dims> NumWork - items,
                    KernelType KernelFunc);

  // 指定された nd_range にわたって
  // 実行するためにカーネルを送信
  template <typename KernelName, typename KernelType,
            int Dims>
  void parallel_for(nd_range<Dims> ExecutionRange,
                    KernelType KernelFunc);
   ...
};
```

図 3-17. ハンドラークラスの非アクセサーメンバーの簡略化された定義

```
class handler {
   ...
  // アクセサーとの間実行する前に
  // 完了する必要があるイベントを指定アクセサーへのコピー、
  // またはアクセサーからのコピー。有効な組み合わせ：
  // Src: アクセサー、Dest: shared_ptr
  // Src: アクセサー、Dest: ポインター
  // Src: shared_ptr、Dest: アクセサー
  // Src: ポインター、Dest: アクセサー
  // Src: アクセサー、Dest: アクセサー
  template <typename T_Src, typename T_Dst, int Dims,
            access::mode AccessMode,
            access::target AccessTarget,
            access::placeholder IsPlaceholder =
              access::placeholder::false_t>::placeholder::false_t>
   void copy(accessor<T_Src, Dims, AccessMode,
                      AccessTarget, IsPlaceholder> Src,
                      shared_ptr_class<T_Dst> Dst);
   void copy(shared_ptr_class<T_Src> Src,
             accessor<T_Dst, Dims, AccessMode, AccessTarget,
                       IsPlaceholder>
                       Dst);
   void copy(accessor<T_Src, Dims, AccessMode, AccessTarget,
                      IsPlaceholder> Src,
                      T_Dst *Dst);
   void copy(const T_Src *Src,
             accessor<T_Dst, Dims, AccessMode, AccessTarget,
                       IsPlaceholder> Dst);
   template <typename T_Src, int Dims_Src,
             access::mode AccessMode_Src,
             access::target AccessTarget_Src, typename T_Dst,
             int Dims_Dst,
             access::mode AccessMode_Dst,
             access::target AccessTarget_Dst,
             access::placeholder IsPlaceholder_Src =
                 access::placeholder::false_t,
              access::placeholder IsPlaceholder_Dst =
                 access::placeholder::false_t>
   void copy(accessor<T_Src, Dims_Src, AccessMode_Src,
                      AccessTarget_Src, IsPlaceholder_Src> Src,
             accessor<T_Dst, Dims_Dst, AccessMode_Dst,
                      AccessTarget_Dst, IsPlaceholder_Dst> Dst);

   //  このアクションの実行後、アクセサーによってアクセスされた
   //  メモリー・オブジェクトがホスト上で更新されることを保証
   template <typename T, int Dims, access::mode AccessMode,
             access::target AccessTarget,
             access::placeholder IsPlaceholder =
                 access::placeholder::false_t>
   void update_host(accessor<T, Dims, AccessMode,
                             ccessTarget, IsPlaceholder> Acc);
  ...
};
```

図 3-18. ハンドラークラスのアクセサーメンバーの簡略化された定義

まとめ

この章では、データ管理の問題に対処するメカニズムと、データの使用を整理する方法について紹介しました。アクセラレーター・デバイスを使用する場合、さまざまなメモリーへのアクセスを管理することが重要な課題であり、ニーズに合わせてさまざまなオプションがあります。

データ利用間に存在する可能性のある様々な依存関係の概要を説明し、これらの依存関係に関する情報をキューに提供してタスクを適切に順序付ける方法を示しました。

この章では、統合共有メモリーとバッファーの概要について説明しました。USM のすべてのモードと動作については、第 6 章でさらに詳しく説明します。第 7 章では、バッファーの作成と動作を制御するさまざまな方法を含め、バッファーについてさらに詳しく説明します。第 8 章では、カーネルの実行とデータ移動の順序を制御するキューのスケジュール・ メカニズムについて再度説明します。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 4 章　並列処理の表現

デバイス上にコード（第 2 章）とデータ（第 3 章）を配置する方法はすでにわかっています。あとは、それに対してどう対処するか決める作業に取り組むだけです。そのために、これまで省略したり、軽く触れただけであった点をいくつか補足していきます。この章では、簡単な例から実際の並列コードへの移行を示し、これまでの章で簡単に示したサンプルコードを詳しく拡張します。

新しい並列言語で最初のプログラムを書くことは、特に並列プログラミングに慣れていない場合、困難な作業のように思えるかもしれません。言語仕様はアプリケーション開発者向けに書かれたものではなく、用語に関するある程度の知識を前提としていることが多く、次のような問に対する答えは含まれていません:

- 並列処理を表現する方法が複数あるのはなぜですか？
- 並列処理を表現するには、どのような方法を使用すればよいですか？
- 実行モデルについて、実際にどれくらい理解しておく必要がありますか？

この章では、これらの質問やその他いくつかの質問に答えていきます。データ並列カーネルの概念を紹介し、実際のコード例を使用してさまざまなカーネル形式の長所と短所について説明し、カーネル実行モデルの最も重要な面を示します。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_4

カーネル内の並列処理

並列カーネルは、データの並列処理を表現する強力な手段として近年登場しました。カーネルベースのアプローチの主な設計目標は、さまざまなデバイス間での*移植性*とプログラマーの高い*生産性*です。そのため、カーネルは通常、特定数のハードウェア・リソース（コア、ハードウェア・スレッド、SIMD［single instruction, multiple data］命令など）や、その構成に合わせてハードコードされていません。代わりに、カーネルは並列処理を抽象的な概念で記述し、実装（つまり、コンパイラーとランタイムの組み合わせ）が特定のターゲットデバイスで利用可能なハードウェアの並列性にマッピングできるようにします。このマッピングは実装定義ですが、実装が適切かつハードウェアの並列処理を効果的に活用できるマッピングを選択することを信頼します。

ハードウェアに依存しない方法で大量の並列処理を公開することで、さまざまなプラットフォームの機能に合わせてアプリケーションをスケールアップ（またはスケールダウン）できるようになりますが、…

機能的な移植性を保証することは、高いパフォーマンスを保証することとは異なります!

サポート対象のデバイスにはかなりの多様性があり、異なるアーキテクチャーはそれぞれ異なるユースケース向けに設計および最適化されていることを認識しておく必要があります。特定のデバイスで最高レベルの*パフォーマンス*を実現しようとする場合、使用しているプログラミング言語に関係なく、常に手動による最適化作業が必要になることを想定すべきです。このようなデバイス固有の最適化の例として、特定のキャッシュサイズ向けのブロック化、スケジュールのオーバーヘッドを償却するワーク粒度サイズの選択、特殊な命令またはハードウェア・ユニットの利用、そして最も重要である適切なアルゴリズムの選択などが挙げられます。これらの例のいくつかを、第 15 章、第 16 章、第 17 章で取り上げています。

アプリケーション開発中にパフォーマンス、移植性、生産性に適切なバランスをとることは、開発者が直面する課題ですが、本書ですべてを網羅することはできません。ただし、SYCL を使用した C++ は、単一の高水準プログラミング言語を使用して、汎用性のある移植可能なコードと最適化されたターゲット固有のコードの両方の維持に必要なすべてのツールを提供することを示したいと考えています。残りはみなさんの課題として残しておきます。

ループとカーネル

反復ループは本質的にシリアルな構造です。つまり、ループの各反復は順番に（つまり、順番に）実行されます。最適化コンパイラーは、ループの一部またはすべての反復を並列実行できると判断する場合がありますが、それは保守的でなければいけません。つまり、コンパイラーの最適化能力が高くない、または並列実行が常に安全であることを証明する十分な情報がない場合、正当性のためループの順次セマンティクスを保持する必要があります。

```
for (int i = 0; i < N; ++i) {
    c[i] = a[i] + b[i];
}
```

図 4-1. ベクトル加算をシリアルループとして表現

単純なベクトル加算を表す図 4-1 のループを考えてみます。このような単純な例でも、ループを並列実行できることを証明するのは簡単ではありません。並列実行は、`c` が `a` または `b` と重複しない場合にのみ安全ですが、これは一般に実行時のチェックなしでは証明できません。これに対処するため、言語には、分析を簡素化する追加情報（ポインターが `restrict` と重複しないことをアサートするなど）をコンパイラーに提供したり、すべての分析を完全にオーバーライドする (ループのすべての反復が独立していると宣言したり、ループを並列リソースにスケジュールする方法を正確に定義するなど) 機能が追加されています。

*並列ループ*の正確な意味は、異なる並列プログラミング言語やランタイムによる用語の多重定義により、やや曖昧ですが、一般的な並列ループ構造の多くは、順次ループに適用されるコンパイラーによる変換を表します。このようなプログラミング・モデルを使用すると、順次ループを記述して、その後に異なる反復を安全に並列実行する方法に関する情報を通知できます。これらのモデルは非常に強力であり、他の最先端のコンパイラーの最適化と統合され、並列プログラミングを大幅に簡素化しますが、開発の初期段階で並列性について考えることは必ずしも推奨されるものではありません。

並列カーネルはループではなく、反復処理はありません。むしろ、カーネルは単一の操作を記述し、これを複数回インスタンス化して、異なる入力データに適用できます。カーネルが並列に起動されると、その操作の複数のインスタンスが同時に実行される可能性があります。

```
launch N kernel instances {
   int id =
       get_instance_id();     // [0, N) 内の一意の識別子
   c[id] = a[id] + b[id];
}
```

図 4-2. *ループを並列カーネルとして（擬似コードで）書き直す*

図 4-2 は、擬似コードを使用してカーネルに書き直したループの例を示しています。このカーネルにおける並列処理の可能性は明確かつ明示的です。カーネルは任意の数のインスタンスによって並列に実行可能であり、各インスタンスは独立して個別のデータ処理を適用します。この操作をカーネルとして記述することで、並列実行が安全であること（そして理想的には並列実行されること）を示します。

カーネルベースのプログラミングは、既存の逐次コードに対して後から並列性を組み込む方法ではなく、明示的に並列なアプリケーションを記述する方法論です。

並列ループからカーネルへの考え方を素早く移行できれば、SYCl で C++ を使用した効果的な並列プログラミングが容易になります。

多次元カーネル

多くの言語による並列構造は 1 次元であり、ワークを対応する 1 次元のハードウェア・リソース（ハードウェア・スレッド数など）に直接マッピングします。SYCL の並列カーネルはこれよりも高レベルの概念であり、その次元はコードが通常解決しようとする問題（1 次元、2 次元、または 3 次元空間）を反映しています。

ただし、並列カーネルによって提供される多次元インデックスは、プログラマーにとって便利な機能であり、基盤となる一次元空間の上に実装される可能性があることを忘れてはなりません。このマッピングの動作を理解することは、特定の最適化（メモリー・アクセス・パターンの調整など）の重要な要素となることがあります。

重要な考慮事項の 1 つは、どの次元が*連続*しているか、または*ユニットストライド*であるか（つまり、多次元空間のどの位置が 1 次元空間で互いに隣接しているか）知ることです。SYCL における並列処理に関連するすべての多次元量には、同じ規則が適用されます。次元には `0` から `N-1` までの番号付され、次元 `N-1` は連続した次元に対応します。多次元量がリストとして記述されている場合（コンストラクター内など）、またはクラスが複数の添字演算子をサポートしている場合、この番号付けは左から右に適用されます（左端の次元 `0` から開始）。この規則は、標準 C++ における多次元配列の動作と一致しています。

SYCL 規則を使用して 2 次元空間を線形インデックスにマッピングする例を図 4-3 に示します。もちろん、この規則から逸脱して、インデックスを線形化する独自の方法を採用することはできますが、慎重に行う必要があります。SYCL 規則から逸脱すると、ストライド 1 のアクセスのメリットを享受するデバイスのパフォーマンスに悪影響を与える可能性があります。

図 4-3. 線形インデックスにマッピングされたサイズ (2, 8) の 2 次元範囲

アプリケーションが 3 次元以上のデータを必要とする場合、モジュロー演算を使用して多次元インデックスと線形インデックス間のマッピングを手動で行います。

言語機能の概要

並列カーネルの記述を決定したら、起動するカーネルの種類と、それをプログラム内でどのように表現するか決める必要があります。並列カーネルを表現する方法は多数あり、言語を習得するには、これらに精通する必要があります。

カーネルとホストコードの分離

ホストコードとデバイスコードを分離する方法はいくつかあり、アプリケーション内で組み合わせることができます。C++ ラムダ式または関数オブジェクト、相互運用性インターフェイス (OpenCL C ソース文字列など) を介して定義されたカーネルやバイナリーがあります。

これらのオプションの一部はすでに第 2 章で説明されており、その他のオプションについては第 10 章と第 20 章で詳しく説明します。

並列性を表現する基本的な概念は、これらすべてのオプションで共有されています。一貫性と簡潔性を保つため、この章のすべてのコード例では、C++ ラムダ式を使用してカーネルを表現します。

ラムダ式は有害ではない

SYCL を使い始めるため、C++ 仕様にあるラムダ式についてすべて完全に理解する必要はありません。ラムダ式の本体がカーネルを表し、キャプチャーされた変数（値によって）が引数としてカーネルに渡されることを知っておくだけで十分です。

カーネルを定義する冗長なメカニズムの代わりにラムダ式を使用することによって生じるパフォーマンスへの影響はありません。SYCL をサポートする C++ コンパイラーは、ラムダ式が並列カーネルの本体を表す場合、それに応じて並列実行を最適化できます。

C++ ラムダ式の復習と SYCL での使用に関する注意事項については、第 1 章を参照してください。カーネルを定義するラムダ式のより具体的な詳細については、第 10 章を参照してください。

並列カーネルの異なる形式

SYCL には 3 つの異なるカーネル形式があり、それぞれ異なる実行モデルと構文をサポートしています。任意のカーネル形式を使用して移植可能なカーネルを作成でき、そのカーネルを調整して、さまざまなデバイスタイプで高いパフォーマンスを達成できます。

ただし、特定の並列アルゴリズムをさらに簡単に表現したり、他の方法ではアクセスできない言語機能を利用するため、特定の形式を利用したい場合があります。

最初の形式は*基本的*なデータ並列カーネルで使用され、カーネル記述の最も簡単な入門を提供します。基本カーネルでは、カーネルの表現をできるだけ単純にするため、スケジュールなど低レベルの機能に対する制御を犠牲にします。
個々のカーネル・インスタンスがハードウェア・リソースにどのようにマップされるかは実装によって制御されるため、基本カーネルが複雑になるにつれて、そのパフォーマンスを判断することがますます難しくなります。

2 番目の形式では、基本カーネルを拡張して、低レベルのパフォーマンス・チューニング機能へのアクセスを提供します。この形式は、歴史的な理由から *ND-range*（N 次元範囲）データ並列として知られています。最も重要なことは、特定のカーネル・インスタンスをグループ化できるようになり、データの局所性と、個々のカーネル・インスタンスとそれらの実行に使用されるハードウェア・リソース間のマッピングをある程度制御できるようになることです。

3 番目の形式では、ネストされた並列ループに類似した構文を使用して ND-range カーネルを表現する実験的な代替構文が提供されます。この形式は、階層型データ並列と呼ばれ、ユーザーのソースコードに現れるネストされた構造の階層を指します。この構文に対するコンパイラーのサポートはまだ未熟であり、多くの SYCL 実装では、階層型データ並列カーネルを他の 2 つの形式ほど効率良く実装できていません。SYCL のパフォーマンス向上機能の多くは階層型カーネルと互換性がなかったり、階層型カーネルからアクセスできない場合があり構文も不完全です。SYCL の階層的並列処理は現在更新中であり、SYCL 仕様には、機能が準備されるまで新しいコードでは階層的並列処理の使用を控えるよう注記が含まれています。この注記に沿って、本書のでは、基本的な並列処理と ND-range の並列処理のみを説明します。

この章の最後で、カーネルのさまざまな形式の機能について詳しく説明した後、それらの選択方法を再度検討します。

基本データ並列カーネル

並列カーネルの最も基本的な形式は、*驚異的並列処理*（つまり、すべてのデータに完全に独立して任意の順序で適用できる操作）に適しています。この形式を使用することで、実装ではワークのスケジュールを完全に制御できるようになります。したがって、これは*記述的プ*ログラミング構造の例です。つまり、操作が驚くほど並列化可能であることを*記述*するだけで、スケジュールに関する全ての決定は実装によって行われます。

基本的なデータ並列カーネルは、単一プログラム複数データ（SPMD）スタイルで記述されます。つまり、単一の“プログラム”（カーネル）が複数のデータに適用されます。このプログラミング・モデルでは、データ依存の分岐により、カーネルの各インスタンスがコード内で異なるパスを通過することが依然として許可されていることに注意してください。

SPMD プログラミング・モデルの最大の利点の 1 つは、明示的な指示なしに、同じ“プログラム”を複数のレベルとタイプの並列処理にマッピングできることです。同じプログラムのインスタンスは、パイプライン化したり、まとめてパックして SIMD 命令で実行したり、複数のハードウェア・スレッドに分散したり、またはこれら 3 つを組み合わせることもできます。

基本データ並列カーネルを理解する

基本的な並列カーネルの実行空間は実行*範囲（レンジ）*と呼ばれ、カーネルの各インスタンスは*項目（アイテム）*と呼ばれます。この様子を図 4-4 に示します。

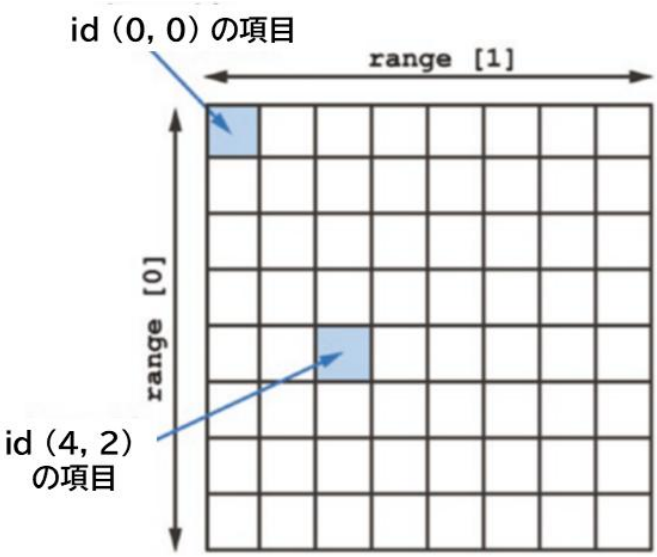

図 4-4. 基本的な並列カーネルの実行空間。2D 範囲の 64 項目を表示。

基本的なデータ並列カーネルの実行モデルは非常に単純です。完全な並列実行が*可能*ですが、それは*保証*または*要求*されるわけではありません。項目は、単一のハードウェア・スレッド上で順番に（つまり、並列処理なしで）実行するなど、任意の順序で実行できます。したがって、すべての項目が並列に実行されることを想定するカーネルでは、（項目の同期を試みるなどで）一部のデバイスでプログラムが簡単にハングする可能性があります。

ただし、正当性を保証するには、常にカーネルが並列に実行されるという前提でカーネルを作成する必要があります。たとえば、競合状態を防ぐために、メモリーへの同時アクセスがアトミックメモリー操作（第 19 章を参照）によって適切に保護されることを保証するのは、私たちの責任です。

基本データ並列カーネルを記述

基本的なデータ並列カーネルは、parallel_for 関数を使用して表現されます。図 4-5 は、この関数を使用してベクトルの加算を表現する方法を示しています。これは、並列アクセラレーター・プログラミングにおける “Hello, world!” であると言えます。

```
h.parallel_for(range{N}, [=](id<1> idx) {
    c[idx] = a[idx] + b[idx];
});
```

図 4-5. parallel_for によるベクトル加算カーネルの表現

この関数は 2 つの引数のみを取ります。1 つ目は各次元で起動する項目の数を指定する範囲（または整数）であり、2 つ目は範囲内の各インデックスに対して実行されるカーネル関数です。カーネル関数への引数として受け入れ可能なクラスはいくつかありますが、どのクラスを使用するかは、必要な機能を示すクラスによって異なります。これについては後で説明します。

図 4-6 は、この関数を非常によく似た方法で表現しています。これは、2 次元データ以外はベクトルの加算と（数学的には）同一です。これはカーネルに反映されています。2 つのコードの唯一の違いは、使用されている範囲と ID クラスの次元です。SYCL アクセサーは多次元 ID でインデックス付けできるため、このように記述することができます。一見奇妙に思えますが、これは非常に強力で、データの次元に基づいてテンプレート化された汎用カーネルを記述できます。

```
h.parallel_for(range{N, M}, [=](id<2> idx) {
    c[idx] = a[idx] + b[idx];
});
```

図 4-6. parallel_for による行列加算カーネルの表現

C/C++ では、多次元データ構造のインデックス作成に複数のインデックスと複数の添字演算子を使用するのが一般的であり、この明示的なインデックス作成はアクセサーでもサポートされています。

複数のインデックスを使用すると、カーネルが異なる次元のデータに対して同時に操作を行う場合や、カーネルのメモリー・アクセス・パターンが項目の ID を直接使用して記述できるよりも複雑な場合に、読みやすさが向上します。

たとえば、図 4-7 の行列乗算カーネルは、2 つの行列の行と列の間のドット積を記述できるように、インデックスの 2 つの成分を抽出する必要があります。著者の意見では、複数の添字演算子（例: [j][k]）を一貫して使用すると、複数のインデックス・モードを混在させて 2 次元の id オブジェクト（例: id(j,k)）を構築するよりも読みやすくなりますが、これは単に個人の好みの問題です。

この章の残りの例では、アクセスされるバッファーの次元に曖昧にならないように、複数の添字演算子を使用します。

```
h.parallel_for(range{N, N}, [=](id<2> idx) {
  int j = idx[0];
  int i = idx[1];
  for (int k = 0; k < N; ++k) {
    c[j][i] +=
        a[j][k] * b[k][i]; // または c[idx] += a[id(j,k)]
                           // * b[id(k,i)];
  }
});
```

図 4-7. parallel_for を用いた正方行列の単純な行列乗算カーネル

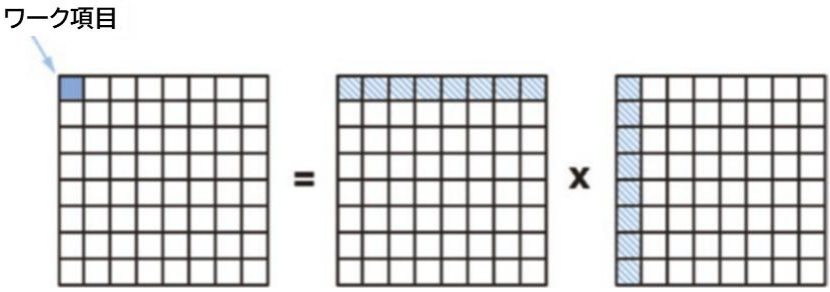

図 4-8. 行列乗算演算を実行範囲内の項目にマッピング

図 4-8 の図は、行列乗算カーネルのワークが個々の項目にどのようにマッピングされるかを示しています。項目数は出力範囲のサイズから導き出され、同じ入力値が複数の項目によって読み取られる可能性があることに注意してください。各項目は、A 行列の（連続した）行と B 行列の（連続していない）列を順番に反復処理して、C 行列に単一の値を計算します。

基本データ並列カーネルの詳細

基本的なデータ並列カーネルの機能は、range、id、item の 3 つの C++ クラスを通じて公開されます。これまでの章で range クラスと id クラスについては説明しましたが、ここでは別の視点を持って再度取り上げます。

range クラス

range は、1 次元、2 次元、または 3 次元の範囲を表します。range の次元はテンプレート引数であるため、コンパイル時に既知である必要がありますが、各次元のサイズは動的であり、実行時にコンストラクターに渡されます。range クラスのインスタンスは、並列構造の実行範囲とバッファーサイズ両方の記述に使用されます。

図 4-9 に、range クラスの簡略化された定義を示します。この定義には、range の範囲を照会するコンストラクターとさまざまなメソッドが示されています。

```
template <int Dimensions = 1>
class range {
 public:
   // 1 次元、2 次元、または 3 次元の範囲を構築
   range(size_t dim0);
   range(size_t dim0, size_t dim1);
   range(size_t dim0, size_t dim1, size_t dim2);

   // 特定の次元範囲のサイズを返す
   size_t get(int dimension) const;
   size_t &operator[](int dimension);
   size_t operator[](int dimension) const;

   // 各次元のサイズの積を返す
   size_t size() const;

   // 範囲に対する算術演算もサポートされます
};
```

図 4-9. range クラスの簡略化された定義

id クラス

id は、1 次元、2 次元、または 3 次元の範囲のインデックスを表します。id の定義は多くの点で range と似ています。次元もコンパイル時に既知である必要があり、並列構造内のカーネルの個々のインスタンスまたはバッファーへのオフセットをインデックスとして使用できます。

図 4-10 の id クラスの定義に示されているように、id は概念的には 1 、2 、または 3 つの整数のコンテナーにすぎません。利用できる操作もシンプルです。各次元のインデックスのコンポーネントを照会したり、簡単な演算を実行して新しいインデックスを計算できます。

任意のインデックスを表す id を作成できますが、特定のカーネル・インスタンスに関連付けられた id を取得するには、その id（またはそれを含む項目）をカーネル関数の引数として受け入れる必要があります。この id（またはそのメンバー関数によって返される値）は、インデックスを照会する任意の関数に転送する必要があります。現在、プログラム内の任意のポイントでインデックスを照会する関数はありませんが、これは SYCL の将来のバージョンで簡素化される可能性があります。

id を受け入れるカーネルの各インスタンスは、計算のために割り当てられた範囲内のインデックスのみを認識し、範囲自体は認識しません。カーネル・インスタンスに自身のインデックスと範囲を認識させたい場合は、代わりに item クラスを使用する必要があります。

```
template <int Dimensions = 1>
class id {
 public:
   // 1 次元、2 次元、または 3 次元の範囲を構築
   id(size_t  dim0);
   id(size_t dim0, size_t dim1);
   id(size_t dim0, size_t dim1, size_t dim2);

   // 特定の次元の ID のコンポーネントを返します
   size_t get(int dimension) const;
   size_t &operator[](int dimension);
   size_t operator[](int dimension) const;

   // ids に対する算術演算もサポートされます
};
```

図 4-10. id クラスの簡略化された定義

item クラス

item はカーネル関数の個々のインスタンスを表し、カーネルの実行範囲とその範囲内のインスタンスのインデックスの両方をカプセル化します（それぞれ範囲と id を使用）。range や id と同様に、その次元はコンパイル時に判明していなければなりません。

item クラスの簡略化された定義を図 4-11 に示します。item と id の主な違いは、item が実行範囲のプロパティー（サイズなど）を照会する追加関数と、線形化されたインデックスを計算する関数を公開していることです。id と同様に、特定のカーネル・インスタンスに関連付けられた項目を取得する唯一の方法は、それをカーネル関数への引数として受け入れることです。

```
template <int Dimensions = 1, bool WithOffset = true>
class item {
 public:
   // カーネルの実行範囲内のこの項目の
   // インデックスを返します
   id<Dimensions> get_id() const;
   size_t get_id(int dimension) const;
   size_t operator[](int dimension) const;

   // この項目によって実行されるカーネルの実行範囲を返します
   // 実行範囲を返します
   range<Dimensions> get_range() const;
   size_t get_range(int dimension) const;

   // この項目のオフセットを返します (WithOffset == true の場合)
   id<Dimensions> get_offset() const;

   // この項目の線形インデックスを返します
   // 例: id(0) * range(1) * range(2) + id(1) * range(2) + id(2)
   size_t get_linear_id() const;
};
```

図 4-11. item クラスの簡略化された定義

明示的な ND-Range カーネル

並列カーネルの 2 番目の形式では、基本的なデータ並列カーネルのフラットな実行範囲が、項目がグループに属する実行範囲に置き換えられます。この形式は、カーネル内の局所性の概念を表現する場合に最適です。

異なるタイプのグループに対して異なる動作が定義され保証されることで、ワークを特定のハードウェア・プラットフォームにマッピングする方法の理解と制御が深まります。

したがって、これらの明示的な ND-range カーネルはより*規範的*な並列構造の例です。ここでは、各タイプのグループへのワークのマッピングを*規定*し、実装はそのマッピングに従う必要があります。ただし、グループ自体は任意の順序で実行される可能性があり、実装では各タイプのグループがハードウェア・リソースにどのようにマップされるかは、ある程度の自由が保持され完全に規定されているわけではありません。この規範的プログラミングと記述的プログラミングの組み合わせにより、移植性を損なうことなく、カーネルを局所性に合わせて設計および調整できるようになります。

基本的なデータ並列カーネルと同様に、ND-range カーネルは SPMD スタイルで記述され、すべてのワーク項目が複数のデータに適用される同じカーネル "プログラム" を実行します。主な違いは、各プログラム・インスタンスが、それを含むグループ内の位置を照会でき、各グループタイプに固有の追加機能にアクセスできることです (第 9 章を参照)。

明示的な ND-Range 並列カーネルを理解する

ND-range カーネルの実行範囲は、ワークグループ、サブグループ、およびワーク項目に分割されます。ND-range は、均一なサイズのワークグループに分割された合計実行範囲を表します (つまり、ワークグループのサイズは、各次元で ND-range のサイズを正確に分割する必要があります)。各ワークグループは、実装によってさらにサブグループに分割できます。ワーク項目と各タイプのグループに定義された実行モデルを理解することは、正確で移植可能なプログラムを作成する上で重要ことです。

図 4-12 は、サイズ (8、8、8) の ND-range をサイズ (4、4、4) の 8 つのワークグループに分割した例を示しています。各ワークグループは、4 つの項目を持つ 1 次元サブグループを 16 個持ちます。次元の番号付けには注意してください。

サブグループは常に 1 次元であるため、ND-range とワークグループの次元 2 はサブグループの次元 0 になります。

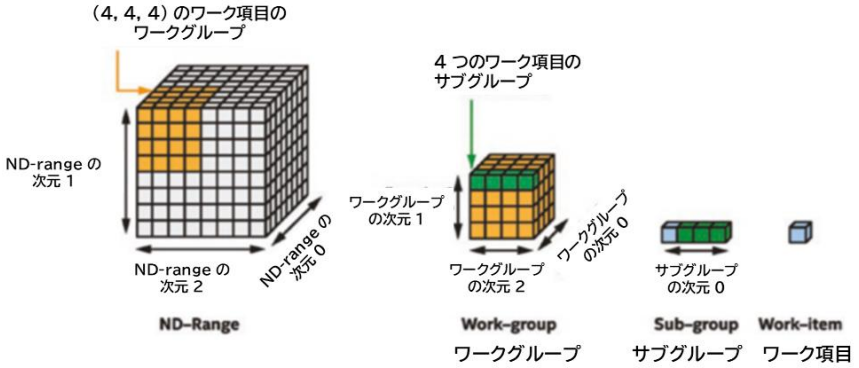

図 4-12. 3 次元 ND-range をワークグループ、サブグループ、およびワーク項目に分割

各タイプのグループからハードウェア・リソースへの正確なマッピングは*実装によって定義*されており、この柔軟性により、プログラムをさまざまなハードウェア上で実行できるようになります。たとえば、ワーク項目は完全に順次実行することも、ハードウェア・スレッドや SIMD 命令で並列実行することも、カーネル用に特別に構成されたハードウェア・パイプラインで実行することもできます。

この章では、汎用ターゲット・プラットフォームの観点から見た ND-range 実行モデルのセマンティック保証のみに焦点を当てており、特定のプラットフォームへのマッピングについては説明しません。GPU、CPU、FPGA のハードウェア・マッピングとパフォーマンスの推奨事項の詳細については、第 15 章、第 16 章、第 17 章を参照してください。

ワーク項目（`work-item`）

ワーク項目は、カーネル関数の個々のインスタンスを表現します。他のグループ化がない場合、ワーク項目は任意の順序で実行され、グローバルメモリーへのアトミックメモリー操作を行わない限り、相互に通信または同期することはできません（第 19 章を参照）。

ワークグループ（`work-group`）

ND-range 内のワーク項目はワークグループに編成されます。ループ内のワーク項目は、グローバルメモリーへのアトミックメモリー操作を除いて相互に通信できません（第 19 章を参照）。ただし、特定の構造が使用される場合、ワークグループ内のワーク項目には一定のスケジュールの保証があり、この局所性によっていくつかの追加機能があります。

1. ワークグループ内のワーク項目は *ワークグループのローカルメモリー* にアクセスできます。このメモリーは一部のデバイスでは専用の高速メモリーにマップされる場合があります（第 9 章を参照）。

2. ワークグループ内のワーク項目は、*ワークグループ・バリア*を使用して同期し、*ワークグループ・ メモリーフェンス*を使用してメモリーの一貫性を保証できます（第 9 章を参照）。

3. ワークグループ内のワーク項目は *グループ関数*にアクセスでき、共通の通信ルーチン（第 9 章を参照）と *グループ・アルゴリズム*に実装され、リダクションやスキャンなどの共通の並列パターンの実装が提供されます（第 14 章を参照）。

ワークグループ内のワーク項目の数は、通常、実行時にカーネルごとに構成されます。これは、最適なグループ化は、利用可能な並列処理の量（つまり、ND-range のサイズ）とターゲットデバイスのプロパティーの両方に依存するためです。`device` クラスのクエリー関数を使用して、特定のデバイスでサポートされているワークグループあたりのワーク項目の最大数を決定できます（第 12 章を参照）。また、各カーネルに要求されたワークグループ・サイズが有効であることを確認するのは、プログラマーの責任です。

ワークグループの実行モデルには、強調する価値のある微妙な点がいくつかあります。

まず、ワークグループ内のワーク項目は単一の計算ユニットにスケジュールされますが、ワークグループ数と計算ユニットの数の間には関連がなくてもかまいません。実際、ND-range のワークグループ数は、特定のデバイスが同時に実行できるワークグループ数よりも何倍か大きくなることがあります。ワークグループ間で同期するカーネルを、デバイス固有の高度なスケジュールに任せて記述したくなるかもしれませんが、それは推奨しません。その場合、カーネルは動作しているように見えても、将来の実装で動作が保証されるわけではなく、異なるデバイスに移行した時に動作しなくなる可能性が極めて高いためです。

2 番目に、ワークグループ内のワーク項目は互いに連携できるようにスケジュールされますが、特定の*前方進行を保証*する必要はありません。つまり、ワークグループ内のワーク項目をバリアと集合演算の間で順番に実行することは有効な実装です。同じワークグループ内のワーク項目との通信と同期は、バリア機能と集合演算を使用して実行される場合にのみ安全であることが保証され、手動でコーディングされた同期ルーチンはデッドロックする可能性があります。

ワークグループでの思考

ワークグループは、多くの点で他のプログラミング・モデル（スレッディング・ビルディング・ブロックなど）のタスクの概念に似ています。タスクは任意の順序で実行できます（スケジューラーによって制御されます）。タスクでマシンをオーバーサブスクライブ状態にすることは可能（望ましい場合もあります）ですが、タスクのグループ全体にバリアを実装することは多くの場合、得策ではありません（コストが非常に高かったり、スケジューラーと互換性がなかったりする可能性があるため）。タスクベースのプログラミング・モデルに精通している場合は、ワークグループをデータ並列タスクとして考えると良いかもしれません。

サブグループ（sub-group）

多くの最新ハードウェア・プラットフォームでは、ワークグループ内のワーク項目のサブセットである*サブグループ*が、保証付きのスケジューリングで実行されます。たとえば、サブグループ内のワーク項目は、コンパイラーのベクトル化の結果として同時実行される可能性があり、また、サブグループ自体は独立したハードウェア・スレッドにマップされているため、強力な前方進行の保証付きで実行される可能性もあります。

単一プラットフォームで作業する場合、これらの実行モデルに関する想定をコードに組み込みたくなりますが、コードが本質的に安全でなくなり、移植性が失われます。異なるコンパイラーを使用する場合や、同じベンダーの異なる世代のハードウェアを使用する場合にも、コードが破綻する可能性があります。

サブグループを言語の中核として定義すると、後でデバイス固有であることが判明する可能性を仮定する代わりに、安全な代替手段が得られます。サブグループ機能を活用することで、低レベル（ハードウェアに近いレベル）でのワーク項目の実行を推測することも可能になり、多くのプラットフォームで高いレベルのパフォーマンスを実現する鍵となります。

ワークグループと同様に、サブグループ内のワーク項目は、グループ関数とグループ・アルゴリズムを使用して同期したり、メモリーの一貫性を保証したり、共通の並列パターンを実行したりできます。ただし、サブグループにはワークグループのローカルメモリーに相当するものはありません（サブグループのローカルメモリーは存在しません）。代わりに、サブグループ内のワーク項目は、“*シャッフル*” 操作と呼ばれるグループ・アルゴリズムのサブセットを使用して、明示的なメモリー操作なしで直接データを交換できます（第 9 章）。

なぜシャッフルか？

OpenCL、CUDA、SPIR-V などの言語のシャッフル操作はすべて、その名前に “`shuffle`” を含んでいます（例：`sub_group_shuffle`、`__shfl`、`OpGroupNonUniformShuffle`）。
SYCL は、`std::shuffle` 関数との混同を避けるため異なる命名規則を採用しています。

サブグループのいくつかのアスペクトは実装によって定義され、私たちの制御外にあります。ただし、サブグループは、デバイス、カーネル、および ND-range の特定の組み合わせに対して固定（1 次元）サイズを持ち、`kernel` クラスのクエリー関数を使用してサイズを照会できます（第 10 章と第 12 章を参照）。デフォルトでは、サブグループあたりのワーク項目の数も実装によって選択されます。コンパイル時に特定のサブグループ・サイズを指定することでこの動作を変更できますが、要求するサブグループ・サイズをデバイスがサポートすることを確認する必要があります。

ワークグループと同様に、サブグループ内のワーク項目は、特定の前方進行の保証を持つ必要はありません。実装では、サブグループ内の各ワーク項目を順番に実行し、サブグループの集合関数に遭遇した場合にのみワーク項目を切り替えることができます。ただし、一部のデバイスでは、ワークグループ内のすべてのサブグループが最終的に実行される（進行する）ことが保証されており、これはいくつかの生産と消費パターンの基礎となります。

これは現在、実装定義の動作であるため、カーネルを移植可能な状態に保ちたい場合は、サブグループに依存してワークを進めてはなりません。SYCL の将来のバージョンでは、サブグループの進行保証を記述するデバイスクエリーが用意される予定です。

特定のデバイス向けのカーネルを作成する場合、ワーク項目とサブグループのマッピングは既知であり、コードではこのマッピングのプロパティーを利用してパフォーマンスを向上できる場合があります。しかし、よくある過ちは、コードが 1 つのデバイスで動作するからといって、すべてのデバイスでも動作と想定することです。 図 4-13 と 4-14 は、{4, 4} の範囲を持つ多次元カーネル内のワーク項目をサブグループにマッピングする 2 つの例を示しています (サブグループの最大サイズは 8)。図 4-13 のマッピングでは 8 つのワーク項目を含むサブグループが 2 つ生成されますが、図 4-14 のマッピングでは 4 つのワーク項目を含むサブグループが 4 つ生成されます。

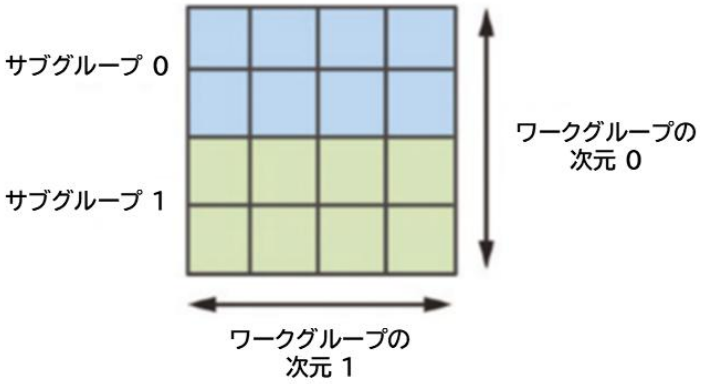

図 4-13. サブグループ・マッピングの 1 つの可能性として、サブグループ・サイズがワークグループの最も高い番号 (連続) の次元範囲よりも大きくなることが許可され、サブグループが “ラップアラウンド” することがあります

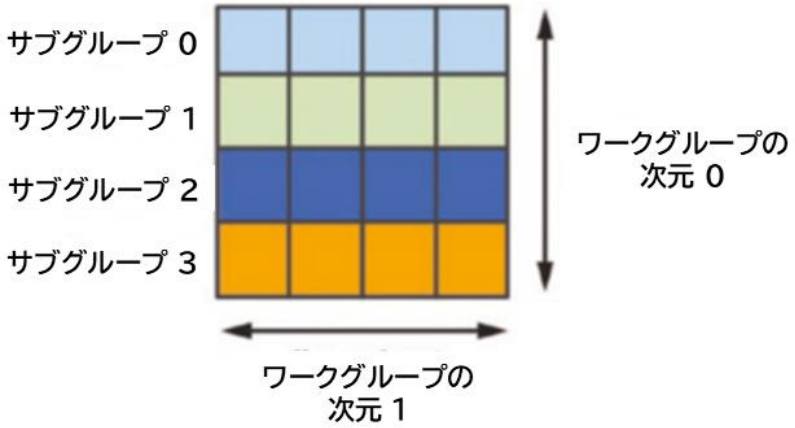

図 4-14. もう一つの可能なサブグループのマッピングでは、サブグループのサイズは、ワークグループの最大番号（連続）の次元の範囲を超えることはできません

SYCL は現在、ワーク項目がサブグループにどのようにマッピングされているか照会する方法や、特定のマッピングを要求するメカニズムを提供していません。サブグループを使用して移植可能なコードを記述する最良の方法は、1 次元のワークグループを使用するか、最も高い番号の次元がカーネルで必要なサブグループ・サイズで割り切れる多次元のワークグループを使用することです。

サブグループでの思考

明示的なベクトル化を考慮するプログラミング・モデルを使用する場合、各サブグループを SIMD レジスターにパックされたワーク項目のセットとして考え、サブグループ内の各ワーク項目が SIMD レーンに対応すると考えると便利です。複数のサブグループが同時に実行され、デバイスがそれらのサブグループが前方進行を保証している場合、このメンタルモデルは、各サブグループが並列に実行される個別のベクトル命令のストリームであるかのように拡張されます。

明示的な ND-Range データ並列カーネルを記述

```
range global{N, N};
range local{B, B};
h.parallel_for(nd_range{global, local},
               [=](nd_item<2> it) {
                 int j = it.get_global_id(0);
                 int i = it.get_global_id(1);

                  for (int k = 0; k < N; ++k) {
                    c[j][i] += a[j][k] * b[k][i];
                  }
                });
```

図 4-15. ND-range parallel_for を使用した単純な行列乗算カーネルの表現

図 4-15 は、ND-range 並列カーネル構文を使用して、前述の行列乗算カーネルを再実装しており、図 4-16 の図は、このカーネルのワークが各ワークグループ内のワーク項目にどのようにマッピングされるかを示しています。このようにワーク項目をグループ化すると、アクセスの局所性が確保され、キャッシュのヒット率向上が期待できます。たとえば、図 4-16 のワークグループはローカル範囲が（4, 4）で、16 個のワーク項目を含みますが、単一のワーク項目の 4 倍のデータしかアクセスしません。つまり、メモリーからロードされた各値は 4 回再利用できます。

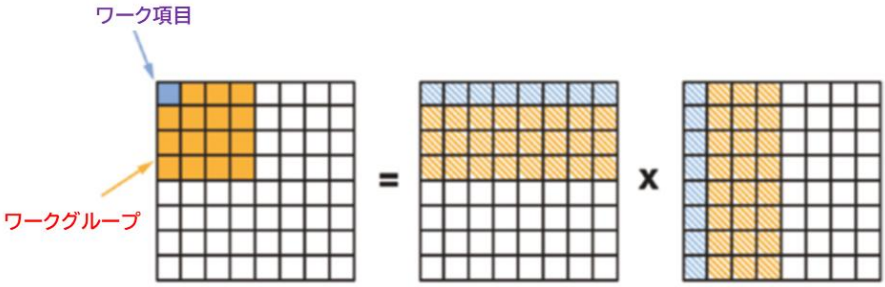

図 4-16. 行列乗算をワークグループとワーク項目にマッピング

これまでの行列乗算の例では、同じワークグループ内のワーク項目からの A 行列と B 行列への繰り返しアクセスを最適化するのはハードウェア・キャッシュに依存してきました。このようなハードウェア・キャッシュは、従来の CPU アーキテクチャーでは一般的であり、GPU アーキテクチャーでも一般的になりつつありますが、いくつかのアーキテクチャーでは、より高いパフォーマンス（たとえば、低レイテンシー）を実現できる明示的に管理される “スクラッチパッド” メモリーが採用されています。ND-range カーネルは、ローカルアクセサーを使用して、ワークグループのローカルメモリーに配置する割り当てを記述できます。その後、実装では、これらの割り当てを特別なメモリー（存在する場合）に自由にマップできます。このワークグループ・ローカル・メモリーの使用法については、第 9 章で説明します。

明示的な ND-Range データ並列カーネルの詳細

ND-range データ並列カーネルは、基本データ並列カーネルとは異なるクラスを使用します。範囲（`range`）は `nd_range` に置き換えられ、項目（`item`）は `nd_item` に置き換えられます。また、ワーク項目が属するさまざまな種類のグループを表す 2 つの新しいクラスもあります。ワークグループに関連付けられた機能は `group` クラスにカプセル化され、サブグループに関連付けられた機能は `sub_group` クラスにカプセル化されます。

`nd_range` クラス

`nd_range` は、範囲クラスの 2 つのインスタンスを使用してグループ化された実行範囲を表します。1 つはグローバル実行範囲を示し、もう 1 つは各ワークグループのローカル実行範囲を示します。`nd_range` クラスの簡略化された定義を図 4-17 に示します。

`nd_range` クラスがサブグループについてまったく触れていないことは少し意外かもしれません。サブグループの範囲は構築中に指定されず、照会することもできません。これには 2 つの理由がありますます。まず、サブグループは、多くのカーネルでは無視できる低レベルの実装です。

2 番目に、正確に 1 つの有効なサブグループ・サイズをサポートするデバイスがいくつかあり、このサイズをあらゆる場所で指定すると冗長になります。
サブグループに関連するすべての機能は、後ほど説明する専用クラスにカプセル化されます。

```
template <int Dimensions = 1>
class nd_range {
 public:
   // グローバル範囲とワークグループ・ローカル範囲から
   // nd_range を構築
  nd_range(range<Dimensions> global,
           range<Dimensions> local);

   // グローバルとワークグループのローカル範囲を返す
   range<Dimensions> get_global_range() const;
   range<Dimensions> get_local_range() const;

   // グローバル範囲内のワークグループ数を返す
   range<Dimensions> get_group_range() const;
};
```

図 4-17. nd_range クラスの簡略化された定義

nd_item クラス

nd_item は項目の ND-range 形式であり、カーネルの実行範囲とその範囲内の項目のインデックスをカプセル化します。nd_item が item と異なるのは、図 4-18 のクラス定義に示されているように、範囲内での位置を照会して表現する方法です。たとえば、get_global_id() 関数を使用して（グローバル）ND-range 内の項目のインデックスを照会したり、get_local_id() 関数を使用して（ローカル）親ワークグループ内の項目のインデックスを照会したりできます。

nd_item クラスは、項目が属するグループとサブグループを記述するクラスへのハンドルを取得する関数も提供します。これらのクラスは、ND-range 内の項目のインデックスを照会する代替インターフェイスも提供します。

```
template <int Dimensions = 1>
class range {
 public:
   // カーネルの実行範囲内のこの項目の
    // インデックスを返します
   id<Dimensions> get_global_id() const;
   size_t get_global_id(int dimension) const;
   size_t get_global_linear_id() const;

   // この項目によって実行されるカーネルの実行範囲を返します
    // 実行範囲を返します
   range<Dimensions> get_global_range() const;
   size_t get_global_range(int dimension) const;

   // 親ワークグループ内のこの項目の
   // インデックスを返します
   id<Dimensions> get_local_id() const;
   size_t get_local_id(int dimension) const;
   size_t get_local_linear_id() const;

   // この項目の親ワークグループの
   // 実行範囲を返します
   range<Dimensions> get_local_range() const;
   size_t get_local_range(int dimension) const;

   // この項目を含むワークグループまたは
   // サブグループへのハンドルを返します
   group<Dimensions> get_group() const;
   sub_group get_sub_group() const;
};
```

図 4-18. nd_item クラスの簡略化された定義

group クラス

グループクラスは、ワークグループに関連するすべての機能をカプセル化しており、簡略化された定義を図 4-19 に示します。

```
template <int Dimensions = 1>
class group {
 public:
   // カーネルの実行範囲内のグループの
   // インデックスを返します
   id<Dimensions> get_id() const;
   size_t get_id(int dimension) const;
   size_t get_linear_id() const;

   // カーネルの実行範囲内のグループの
   // 番号を返します
   range<Dimensions> get_group_range() const;
   size_t get_group_range(int dimension) const;

   // このグループ内のワーク項目の数を返します
   range<Dimensions> get_local_range() const;
   size_t get_local_range(int dimension) const;
};
```

図 4-19. group クラスの簡略化された定義

group クラスが提供する関数の多くは、nd_item クラスに同等の関数があります。たとえば、group.get_group_id() の呼び出しは item.get_group_id() の呼び出しと同等であり、group.get_local_range() の呼び出しは item.get_local_range() の呼び出しと同等です。グループ関数やアルゴリズムを使用していない場合でも、group クラスを使用する必要があるでしょうか？ 中間グループ・オブジェクトを作成するのではなく、nd_item の関数を直接使用する方が簡単ではないでしょうか。これにはトレードオフがあります。グループを使用すると少し多くのコードを記述する必要がありますが、そのコードの可読性は向上します。たとえば、図 4-20 のコードを考えてみましょう。body はグループ内のすべてのワーク項目によって呼び出されることは明らかであり、parallel_for の本体で get_local_range() から返される範囲はグループの範囲であることは明らかです。nd_item のみを使用して同じコードを簡単に記述することもできますが、理解しにくくなる可能性があります。

```
void body(group& g);

h.parallel_for(nd_range{global, local}, [=](nd_item<1> it) {
   group<1> g = it.get_group();
   range<1> r = g.get_local_range();
   ...
   body(g);
});
```

図 4-20. group クラスを使用して読みやすさを向上

group クラスによって有効になるもう 1 つの強力なオプションは、テンプレート引数を介して任意のタイプのグループを受け入れる汎用グループ関数を記述できることです。SYCL は（まだ）公式のグループ “概念”（C++20 の意味で）を定義していませんが、group クラスと sub_group クラスは共通のインターフェイスを公開し、sycl::is_group_v などの特性を使用してテンプレート化された SYCL 関数を制約することができます。現在、この汎用的なコーディング形式の主な利点は、任意の数の次元を持つワークグループをサポートできることと、関数の呼び出し元で関数がワークをワークグループ内のワーク項目に分割するか、サブグループ内のワーク項目に分割するか決定できることです。
ただし、SYCL グループ・インターフェイスの設計は拡張可能であり、SYCL の将来のバージョンでは、さまざまなワーク項目のグループを表すクラスがさらに多く登場することが予想されます。

sub_group クラス

sub_group クラスはサブグループに関連するすべての機能をカプセル化しており、簡略化された定義を図 4-21 に示します。ワークグループとは異なり、sub_group クラスはサブグループの機能にアクセスする唯一の方法です。その機能は nd_item とは重複しません。

```
class sub_group {
  public:
   // サブグループのインデックスを返すします
   id<1> get_group_id() const;

   // この項目の親ワークグループの
   // サブグループの数を返します
   range<1> get_group_range() const;

   // このサブグループ内のワーク項目のインデックスを返します
   id<1> get_local_id() const;

   // このサブグループ内のワーク項目の数を返します
   range<1> get_local_range() const;

   // この項目の親ワークグループ内の
   // サブグループ内のワーク項目の最大数を返します
   range<1> get_max_local_range() const;
};
```

図 4-21. sub_group クラスの簡略化された定義

現在のサブグループ内のワーク項目の数と、ワークグループ内の任意のサブグループ内のワーク項目の最大数を照会する個別の関数があることに注意してください。これらがどのように異なるかは、特定のデバイスに対してサブグループが正確にどのように実装されるかによって異なりますが、その目的は、コンパイラーが対象とするサブグループ・サイズとランタイム・サブグループ・サイズの違いを反映することです。たとえば、非常に小さなワークグループには、コンパイル時のサブグループ・サイズよりも少ないワーク項目が含まれる場合があります。また、異なるサイズのサブグループを使用して、サブグループ・サイズで割り切れないワークグループと次元を処理する場合もあります。

計算をワーク項目へマッピング

これまでのコード例のほとんどは、カーネル関数の各インスタンスが 1 つのデータに対して 1 つの操作に対応することを前提としていました。これはカーネルを記述する簡単な方法ですが、このような 1 対 1 のマッピングは SYCL またはカーネル形式によって決定されません。

個々のワーク項目へのデータ（および計算）の割り当てを常に完全に制御でき、この割り当てのパラメーター化を可能にすると、パフォーマンスの移植性を向上させるのに効果的です。

1 対 1 マッピング

ワークとワーク項目が 1 対 1 でマッピングされるようにカーネルを作成する場合、それらのカーネルは常に、実行するワーク量と正確に一致するサイズの範囲、または `nd_range` を使用して起動する必要があります。これはカーネルを記述する最も明白な方法であり、多くの場合うまく機能します。実装がワーク項目をハードウェアに効率的にマッピングすることを期待できます。

ただし、特定のシステムと実装の組み合わせでパフォーマンスをチューニングする場合、低レベルのスケジューリング動作に注意を払う必要があります。計算リソースへのワークグループのスケジュールは実装定義であり、*動的*である可能性があります（つまり、計算リソースが 1 つのワークグループを完了すると、次に実行するワークグループは共有キューから取得される場合があります）。動的スケジュールがパフォーマンスに与える影響は一定ではなく、その重要性は、カーネル関数の各インスタンスの実行時間や、スケジュールがソフトウェア（CPU など）で実装されているかハードウェア（GPU など）で実装されているかなどの要因によって異なります。

多対 1 マッピング

代替案としては、ワークとワーク項目を多対 1 でマッピングするカーネルを作成する方法があります。この場合、範囲の意味は微妙に変わります。範囲は、実行されるワーク量ではなく、ワーカーの数を表すようになります。ワーカーの数と各ワーカーに割り当てられるワーク量を変更することで、ワークの配分を微調整し、パフォーマンスを最大化できます。

この形式のカーネルを作成するには、次の 2 つの変更が必要です。

1. カーネルは、ワークの合計量を記述するパラメーターを受け入れる必要があります。

2. カーネルには、ワークをワーク項目に割り当てるループが含まれている必要があります。

このようなカーネルの簡単な例を図 4-22 に示します。カーネル内のループが少し変わっていることに注意してください。開始インデックスはグローバル範囲内のワーク項目のインデックスであり、ストライドはワーク項目の合計数です。ワーク項目へのデータの *ラウンドロビン*・スケジュールにより、ループの N 回の反復すべてがワーク項目によって実行されることが保証されるだけでなく、線形ワーク項目が連続したメモリー位置にアクセスすることも保証されます（キャッシュの局所性とベクトル化の動作を改善するため）。ワークをグループ間または個々のグループ内のワーク項目間に同様に分散して、局所性をさらに向上できます。

```
size_t N = ...;      // ワーク量
size_t W = ...;      // ワーカー数
h.parallel_for(range{W}, [=](item<1> it) {
  for (int i = it.get_id()[0]; i < N;
       i += it.get_range()[0]) {
     output[i] = function(input[i]);
   }
});
```

図 4-22. データ範囲と実行範囲が分離されたカーネル

このようなワーク配分パターンは一般的であり、SYCL の将来のバージョンでは ND-range カーネルでのワーク配分の表現を簡素化する構文が導入される予定です。

カーネル形式の選択

異なるカーネル形式の選択は主に個人の好みの問題であり、他の並列プログラミング・モデルや言語に関する過去の経験に影響されます。

特定のカーネル形式を選択するもう 1 つの主な理由は、カーネルが必要とする特定の機能を公開できる唯一の形式であるためです。残念ながら、開発を始める前にどの機能が必要であるか特定するのは難しい場合があります。特に、さまざまなカーネル形式とクラスとの相互作用にまだ慣れていない場合は困難です。

私たちは、この複雑な空間をナビゲートするため、私たち自身の経験に基づいた 2 つのガイドを用意しました。このガイドは、初期段階の提案として捉えるべきものであり、決して私たちの実験に代わるものではありません。さまざまなカーネル形式の中から選択する最善の方法は、それぞれの形式を時間をかけて試し、どの形式がアプリケーションと開発スタイルに最も適しているかを知ることです。

最初のガイドは図 4-23 のフローチャートであり、これに基づいてカーネル形式を選択します:

1. 並列プログラミングの経験があるかどうか

2. 新規にコードをスクラッチから記述するか、または異なる言語で書かれた既存の並列プログラムを移植するか

3. カーネルは超並列であるか、またはカーネル関数の異なるインスタンス間でデータを再利用するか

4. SYCL で新しいカーネルを記述するのは、パフォーマンスを最大化する、コードの移植性を向上させる、あるいは低レベル言語よりも並列性を表現する生産的な手段を提供するためかどうか

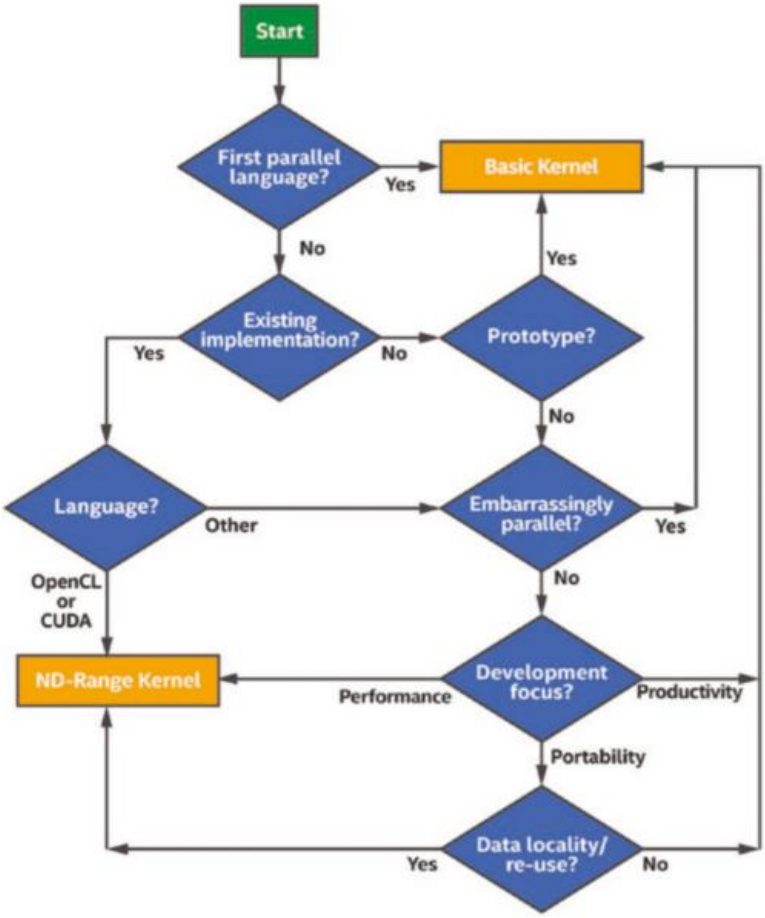

図 4-23. 適切な形式のカーネルを選択する支援

2 番目のガイドは、各カーネル形式に公開される機能のセットです。ワークグループ、サブグループ、グループバリア、グループローカル・メモリー、グループ関数 (例: ブロードキャスト)、およびグループ・アルゴリズム (例: スキャン、リデュース) は、ND-range カーネルでのみ使用できます。そのため、複雑なアルゴリズムを表現したり、パフォーマンスを微調整する必要がある場合、ND-range カーネルを優先します。

各カーネル形式で利用できる機能は、言語の進化によって変化することが予想されますが、基本的な傾向は変わらないと考えられます。つまり、基本的なデータ並列カーネルは局所性を考慮した機能を公開せず、明示的な ND-range カーネルはすべてのパフォーマンスを実現する機能を公開します。

まとめ

この章では、SYCL を使用して C++ で並列処理を表現する基本を紹介し、データ並列カーネルを作成する各アプローチの長所と短所について説明しました。

SYCL はさまざまな形式の並列処理をサポートしており、読者がすぐにコーディングを始められるように十分な情報を提供できたことを願っています。

この章で紹介した概念やクラスの多くは、まだ表面をなぞったに過ぎません。今後、さらに深く掘り下げていきます。ローカルメモリー、バリア、通信ルーチンの使い方については第 9 章で説明します。ラムダ式以外のカーネルの定義方法については第 10 章と第 20 章で説明します。ND-range 実行モデルと特定のハードウェアの詳細なマッピングについては第 15 章、第 16 章、第 17 章で説明します。SYCL を使用して一般的な並列パターンを表現するベスト/プラクティスについては第 14 章で説明します。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 5 章 エラー処理

エラー処理は C++ の重要な機能です。この章では、ワークをデバイス（アクセラレーター）にオフロードするときに発生する固有のエラー処理の課題と、それらが SYCL でどのように管理できるか説明します。

予期しない状況やエラーを検出して対処することは、アプリケーションの開発中に役立ちますが（プロジェクトで作業し、ミスを犯すプログラマーを考えてみてください）、さらに重要なことは、安定して安全な運用アプリケーションとライブラリーでも重要な役割を果たすことです。この章では、SYCL を使用した C++ で使用できるエラー処理メカニズムについて説明します。これにより、エラーの検出と管理を重視するオプションとアプリケーションの設計方法を理解できるようになります。

この章では、SYCL の同期エラーと非同期エラーの概要を説明し、コード内でエラーを処理する場合と、何も処理しない場合のアプリケーションの動作について説明し、非同期エラーを処理する SYCL 固有のメカニズムについて詳しく解説します。

安全が第一

C++ エラー処理の中心となることは、検出された（スローされた）エラーを何も処理しない場合、アプリケーションが終了し何か問題が発生したことが示されることです。この動作により、エラー管理に重点を置かずにアプリケーションを作成し、それでもエラーが何らかの形で開発者やユーザーに通知されることを確信できます。もちろん、これはエラー処理を無視すべきだと言っているわけではありません。製品版のアプリケーションは、エラー管理をアーキテクチャーの中核として設計されるべきですが、多くの場合、アプリケーション開発はそうした配慮なしで開始されます。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_5

C++ は、エラーを処理しないコードであっても、明示的に処理されていない場合でも、多くのエラーを認識できるようにしています。

SYCL はデータ並列 C++ であるため、同様の考え方が適用されます。つまり、コード内でエラー管理なしにエラーが検出されると、プログラムが異常終了し、何か問題が発生したことを通知します。もちろん、運用アプリケーションでは、エラー管理をソフトウェア・アーキテクチャーの中核として考慮する必要があり、レポートだけでなく、エラー状態からの回復も行う必要があります。

エラー管理のコードを追加せずにエラーが発生した場合でも、プログラムの異常終了が発生し、調査の必要があることが示されます。

エラーのタイプ

C++ は、例外メカニズムを通じてエラーを通知して処理するフレームワークを提供します。異種（ヘテロジニアス）プログラミングでは、デバイス上またはデバイス上で処理を開始するときにエラーが発生するため、追加レベルのエラー管理が必要になります。これらのエラーは通常、ホストプログラムの実行とは時間的に分離されているため、通常の C++ 例外処理メカニズムと上手く統合できません。これを解決するため、非同期エラーを一般的な C++ 例外と同じように管理および制御する追加のメカニズムがあります。

図 5-1 は、一般的なアプリケーションの 2 つのコンポーネントを示しています: (1) 順次実行され、将来の実行のためタスクグラフにワークを送信するホストコードと、(2) ホストプログラムとは非同期に実行され、必要な依存関係が満たされたときにデバイス上でカーネルまたはその他のアクションを実行するタスクグラフ。

この例では、タスクグラフの一部として非同期に実行される操作として parallel_for を示していますが、第 3 章、第 4 章、および第 8 章で説明するように、他の操作も可能です。

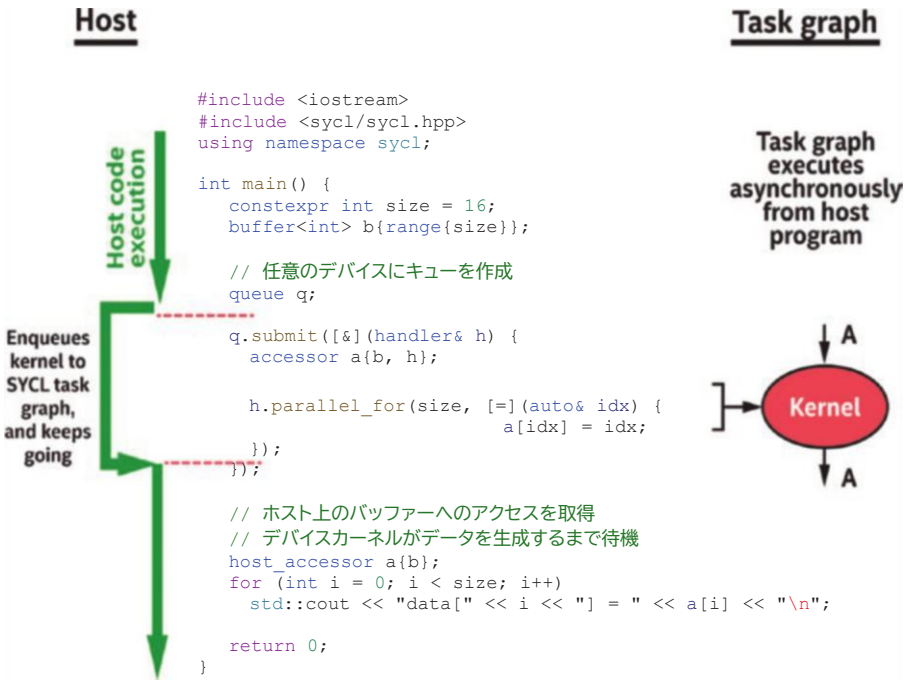

図 5-1. ホストプログラムとタスクグラフ実行の分離

図 5-1 の左側と右側（ホストグラフとタスクグラフ）の区別は、*同期*エラーと*非同期*エラーの違いを理解する鍵となります。

*同期*エラーは、ホストプログラムが API 呼び出しやオブジェクト構築などの操作を行うときにエラー状態が検出された場合に発生します。これらは、図の左側の命令が完了する前に検出され、エラーの原因となった操作によって直ちにエラーがスローされる可能性があります。図の左側の特定の命令を try-catch 構造でラップして、try 内の操作の結果として発生するエラーが try ブロックが終了する前に検出される（つまりキャッチされる）ことが期待できます。

C++ の例外メカニズムは、まさにこのようなタイプのエラーを処理するように設計されています。

非同期エラーは図 5-1 の右側の一部として発生し、タスクグラフ内の操作が実行されたときにのみエラーが検出されます。タスクグラフ実行の一部として非同期エラーが検出される頃には、ホストプログラムは既に実行を続行しているため、これらのエラーをキャッチする `try-catch` 構造でラップするコードはありません。代わりに、SYCL には、ホストプログラムの実行に関連して一見ランダムで制御されていない時間に発生するこれらのエラーを処理する非同期例外処理フレームワークがあります。

エラーを発生させてみましょう!

この章の残りの例で実験を可能にするために、次の例では同期エラーと非同期エラーの両方を作成します。

```
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  buffer<int> b{range{16}};

  // エラー: 親バッファーよりも大きなサブバッファーを定義
  // バッファー・コンストラクターから例外がスローされます。
  buffer<int> b2(b, id{8}, range{16});

  return 0;
}
```

出力例:

```
terminate called after throwing an instance of 'sycl::_V1::invalid_object_error'
  what(): Requested sub-buffer size exceeds the size of the parent buffer -30
(PI_ERROR_INVALID_VALUE)
Aborted
```

図 5-2. 同期エラーの発生

同期エラー

図 5-2 では、バッファーからサブバッファーが作成されていますが、そのサイズは不正です（元のバッファーよりも大きい）。サブバッファーのコンストラクターはこのエラーを検出し、コンストラクターの実行が完了する前に例外をスローします。これは、ホストプログラムの実行の一部として発生するため、同期エラーです。エラーは*コンストラクターが戻る前に*検出できるため、ホストプログラムでエラーが発生した時点、または検出された時点で直ちに処理できます。

このコード例では、C++ 例外をキャッチしても処理は何も行いません。そのため、デフォルトの C++ のキャッチされない例外ハンドラーは、何か問題が発生したことを通知するために `std::terminate` を呼び出します。

非同期エラー

非同期エラーを生成するのは、実装が可能な限り同期的にエラーを検出して報告するように努めているため、少し複雑です。同期エラーはホストプログラムの特定の時点で発生するためデバッグが容易であり、可能な場合は実装で優先されます。デモ用途で非同期エラーを生成する 1 つの方法は、タスクグラフの一部として非同期に実行されるホストタスク内で例外をスローすることです。図 5-3 はこのような例外を示しています。非同期エラーはさまざまな状況で発生し、報告される可能性があります。したがって、図 5-3 に示すホストタスクの例は単なる 1 つの可能性であり、非同期エラーの必須要件ではないことに注意してください。

```
#include <sycl/sycl.hpp>
using namespace sycl;

// 非同期ハンドラー関数の例
auto handle_async_error = [](exception_list elist) {
  for (auto &e : elist) {
      try {
       std::rethrow_exception(e);
      } catch (...){
          std::cout << "Caught SYCL ASYNC exception!!\n";
      }
    }
};

void say_device(const queue &Q) {
  std::cout << "Device : "
            << Q.get_device().get_info<info::device::name>()
            << "\n";
}

class something_went_wrong {};     // 例外タイプの例

int main() {
  queue q{cpu_selector_v, handle_async_error};
  say_device(q);

  q.submit([&](handler &h) {
     h.host_task([]() { throw(something_went_wrong{}); });
   }).wait();

   return 0;
}
```

出力例:

```
Device: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Caught SYCL ASYNC exception!!
```

図 5-3. 非同期エラーの発生

アプリケーションエラー処理方法

C++ の*例外*機能は、プログラム内でエラーが検出された位置と、エラーが処理される位置は明確に分離するように設計されており、この概念は SYCL の同期エラーと非同期エラーの両方にも当てはまります。

`throw` と `catch` メカニズムを通じて、運用アプリケーションで重要になるハンドラーの階層を定義できます。

一貫性と信頼性のある方法でエラーを処理できるアプリケーションを構築するには、事前の方針と、その結果としてエラー管理に構築されたソフトウェア・アーキテクチャーが必要です。C++ は多くの代替方法を実装する柔軟なツールを提供しますが、そのようなアーキテクチャーはこの章では扱いません。このトピックに特化した書籍や参考資料は多数あるため、C++ のエラー管理の詳細については、それらを参考にすることをお勧めします。

ただし、エラーの検出とレポートは必ずしも運用環境で行う必要はありません。実行中にエラーを検出して報告すること（必ずしもエラーからの回復はしない）だけが目的であれば、プログラム内のエラーは最小限のコードで検出して報告できます。次のセクションでは、エラー処理を無視して何もしない場合に何が起こるか（デフォルトの動作はそれほど悪くありません）について説明します。次に、基本的なアプリケーションで簡単に実装できる推奨されるエラー報告について説明します。

エラー処理の無視

C++ と SYCL では、エラーを明示的に処理しなくても問題が発生したことを通知できるように設計されています。未処理の同期または非同期のエラーのデフォルトの結果は、オペレーティング・システムが通知するプログラムの異常終了になります。次の 2 つの例は、それぞれ同期エラーと非同期エラーを処理しない場合に発生する動作を模倣します。

図 5-4 は、ハンドルされていない C++ 例外の結果を示しています。これは、ハンドルされていない SYCL 同期エラーなどが原因である可能性があります。このコードでは、特定のオペレーティング・システムがどのように動作するかテストできます。

```
#include <iostream>

class something_went_wrong {};

int main() {
  std::cout << "Hello\n";

  throw(something_went_wrong{});
}
```

出力例:

```
Hello
terminate called after throwing an instance of 'something_went_wrong'
Aborted
```

図 5-4. C++ の未処理例外

図 5-5 は、呼び出された `std::terminate` からの出力例を示します。これは、アプリケーションで未処理の SYCL 非同期エラーが発生した結果です。このコードでは、特定のオペレーティング・システムがどのように動作するかテストできます。

```
#include <iostream>

int main() {
   std::cout << "Hello\n";

   std::terminate();
}
```

出力例:

```
Hello
terminate called without an active exception
Aborted
```

図 5-5. std::terminate は、SYCL 非同期例外が処理されないときに呼び出されます

プログラム内でエラーを処理する必要があるかもしれませんが、キャッチされない例外は最終的にキャッチされ、プログラムは終了します。これは、例外が失われるよりはましです。

同期エラー処理

SYCL 同期エラーは単なる C++ 例外であるため、ここでは簡単に説明します。SYCL に追加されたエラーメカニズムのほとんどは、次のセクションで説明する非同期エラーに関連していますが、同期エラーは、推論や処理が容易であるため、実装ではできるだけ多くのエラーを同期的に検出して報告するため重要です。

SYCL で定義される同期エラーは、`std::exception` タイプからの派生した `sycl::exception` であり、図 5-6 に示すように `try-catch` 構造によって SYCL エラーをキャッチできます。

```
try {
   // SYCL ワークを実行
} catch (sycl::exception &e) {
   // 例外を出力または処理
   std::cout << "Caught sync SYCL exception: " << e.what()
             << "\n";
   return  1;
}
```

図 5-6. sycl::exception をキャッチするパターン

SYCL は、C++ のエラー処理メカニズムに加えて、ランタイムによってスローされる `sycl::exception` 例外タイプを追加します。それ以外はすべて標準 C++ の例外処理であるため、開発者にはなじみ深いものです。

図 5-7 には、追加の例外クラスが処理される、完全な例が示されています。

```
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
   try {
   buffer<int> b{range{16}};

   // エラー: 親バッファーよりも大きなサブバッファーを定義
   // バッファー・コンストラクターから例外がスローされます。
   buffer<int> b2(b, id{8}, range{16});

   } catch (sycl::exception &e) {
       // 例外を出力または処理
       std::cout << "Caught sync SYCL exception: "
                 << e.what() << "\n";
       return 1;
   } catch (std::exception &e) {
       std::cout << "Caught std exception: " << e.what()
                 << "\n";
       return 2;
  } catch (...){
       std::cout << "Caught unknown exception\n";
       return 3;
   }

  return 0;
}
```

出力例:

```
Caught synchronous SYCL exception: Requested sub-buffer
size exceedsthe size of the parent buffer -30
(PI_ERROR_INVALID_VALUE)
```

図 5-7. コードブロックから例外をキャッチするパターン

非同期エラー処理

非同期エラーは SYCL ランタイム（またはベースとなるバックエンド）によって検出され、エラーはホストプログラムのコマンドの実行とは無関係に発生します。エラーは SYCL ランタイムの内部リストに保存され、プログラマーが制御できる特定のポイントでのみ処理するため解放されます。

非同期エラーの処理をカバーするには、次の 2 つのことを理解してください:

1. 未処理の非同期エラーを処理するため呼び出された場合、ハンドラーが行うべきこと

2. いつ非同期ハンドラーが呼び出されるか

非同期ハンドラー

非同期ハンドラーは、アプリケーションが定義する関数であり、SYCL コンテキストやキューに登録されます。次のセクションで定義される時間に、処理可能な未処理の非同期例外がある場合、SYCL ランタイムによって非同期ハンドラーが呼び出されて例外リストが渡されます。

非同期ハンドラーは、`std::function` としてコンテキストまたはキュー・コンストラクターに渡され、好みに応じて通常の関数、ラムダ式、関数オブジェクトなどの方法で定義できます。ハンドラーは、図 5-8 に示すサンプルのように、`sycl::exception_list` 引数を受け入れる必要があります。

```
// 単純な非同期ハンドラー関数
auto handle_async_error = [](exception_list elist) {
  for (auto& e : elist) {
    try {
      std::rethrow_exception(e);
    } catch (sycl::exception& e) {
      std::cout << "ASYNC EXCEPTION!!\n";
      std::cout << e.what() << "\n";
    }
  }
};
```

図 5-8. ラムダとして定義された非同期ハンドラーの実装例

上の図 5-8 では、`std::rethrow_exception` の後に特定の例外タイプの `catch` が続くことで、例外タイプ（この場合は `sycl::exception` のみ）のフィルター処理が行われます。C++ では代替のフィルター処理を使用することも、タイプに関係なくすべての例外を処理するように選択することもできます。

ハンドラーは、構築時にキューまたはコンテキスト（低レベルの詳細については第 6 章で説明）に関連付けられます。例えば、図 5-8 で定義したハンドラーを作成中のコンテキストに登録するには、次のように記述します。

```
queue my_queue{ gpu_selector_v, handle_async_error };
```

同様に、図 5-8 で定義したハンドラーを作成中のコンテキストに登録するには、次のように記述します。

```
context my_context{ handle_async_error };
```

ほとんどのアプリケーションでは、コンテキストを明示的に作成したり、管理する必要はありません（バックグラウンドで自動的に作成されます）。そのため、非同期ハンドラーを使用する場合、そのハンドラーを特定のデバイスのキューに関連付ける必要があります（明示的なコンテキストではありません）。

非同期ハンドラーを定義する場合、何らかの理由でコンテキストを明示的に管理しない限り、キューで定義する必要があります。

キューやその親コンテキストに対して非同期ハンドラーが定義されていないと、そのキュー（またはコンテキスト）で処理が必要な非同期エラーが発生した場合、デフォルトの非同期ハンドラーが呼び出されます。デフォルトのハンドラーは、図 5-9 のリストと同じように動作します。

```
// 単純な非同期ハンドラー関数
auto handle_async_error = [](exception_list elist) {
  for (auto& e : elist) {
    try {
      std::rethrow_exception(e);
    } catch (sycl::exception& e) {
      // 非同期例外に関連する情報を出力
    } catch (...) {
      // sycl::exception に関連する情報を出力
    }
  }

  // 異常終了して、未処理の例外があることを
  // ユーザーに通知します
  std::terminate();
};
```

出力例:

```
Device: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
terminate called without an active exception
Aborted
```

図 5-9. デフォルトの非同期ハンドラーの動作例

デフォルトのハンドラーは、例外リスト内のエラー情報をユーザーに通知し、`std::terminate` を通じてアプリケーションを異常終了させます。この場合、オペレーティング・システムにも異常終了したことをレポートする必要があります。

非同期ハンドラーにどのようなエラーを通知するかはプログラマーに依存します。アプリケーションが処理を正常に続行できるよう、エラーのログにはアプリケーションの終了、そしてエラー状態の回復までさまざまな情報があります。一般には、`sycl::exception::what()` を呼び出して利用可能なエラーの詳細を通知し、アプリケーションを終了させます。

非同期ハンドラーが内部で何を処理するかを決定するのはプログラマー次第ですが、しばしば見られる間違いとして、エラーメッセージ（プログラムのほかのメッセージで見逃される可能性があります）を出力してから、ハンドラー関数を終了することです。既知のプログラム状態を回復し、実行を続行しても安全であると確信できるエラー管理の原則が確立されていない限り、非同期ハンドラー関数内でアプリケーションを終了することを検討する必要があります。

これにより、エラーが検出されたにもかかわらずアプリケーションの不正実行を続行するプログラムから誤った結果が表示される可能性が減少します。ほとんどのプログラムでは、非同期例外が検出されたら異常終了することが適切です。

包括的なエラー回復および管理メカニズムが用意されていない場合、エラー情報を出力した後、非同期ハンドラー内でアプリケーションを終了することを検討してください。

ハンドラーの起動

非同期ハンドラーは、特定の時間にランタイムによって呼び出されます。エラーの発生はすぐには報告されません。その場合、エラーの管理と安全なアプリケーション・プログラミング（特にマルチスレッド）がより困難となり、コストも高くなります（たとえば、ホストとデバイス間の追加の同期）。非同期ハンドラーは、以下に示す特定のタイミングで呼び出されます:

1. ホストプログラムが特定のキューに対して `queue::throw_asynchronous()` を呼び出すとき
2. ホストプログラムが特定のキューに対して `queue::wait_and_throw()` を呼び出すとき
3. ホストプログラムが特定のイベントに対して `event::wait_and_throw()` を呼び出すとき
4. キューが破棄されたとき
5. コンテキストが破棄されたとき

方法 1 ～ 3 は、ホストプログラムが非同期例外を処理するタイミングを制御するメカニズムを提供し、スレッドの安全性やアプリケーション固有のその他の詳細を管理できるようにします。これらは、非同期例外がホストプログラムの制御フローに入り、同期エラーとほぼ同様に処理できる制御ポイントを提供します。

ユーザーが方法 1 ～ 3 のいずれかを明示的に呼び出さない場合、キューとコンテキストが破棄されるプログラムの終了処理中に非同期エラーが報告されることがあります。多くの場合、これは、何か問題が発生し、プログラムの結果を信頼できないことをユーザーに通知するには十分です。

ただし、プログラムの終了中にエラー検出することは、すべてのケースで機能するとは限りません。たとえば、プログラムが何らかのアルゴリズムの収束基準が達成されたときにのみ終了し、その基準がデバイスカーネルの正常な実行によってのみ達成可能な場合、非同期例外はアルゴリズムが決して収束せず、ティアダウン（エラーが検出される場所）を開始することを通知する可能性があります。
このような場合、また、より完全なエラー処理が導入されている運用アプリケーションでは、プログラム内の通常の制御ポイント（たとえば、アルゴリズムの収束が発生したかどうかを確認する前）で `throw_asynchronous()` または `wait_and_throw()` を呼び出すのが合理的です。

デバイス上のエラー

この章で説明したエラー検出および処理のメカニズムはホストベースです。これらは、ホストプログラムが、ホストプログラム内またはデバイス上のカーネルの実行中に発生した問題を検出し、対処できるメカニズムです。ここでは、作成したデバイスコード内から、何か問題が発生したことを通知する方法は取り上げていません。これは意図的なものです。

SYCL は、デバイスコード内での C++ 例外処理メカニズム（`throw` など）を明示的に禁止します。これは、一部の種類のデバイスでは高いパフォーマンス・コストが発生するためです。デバイスコード内で何らかの問題が発生したことを検出した場合、既存の例外ベースではない手法でエラーを通知する必要があります。たとえば、エラーをログに記録したり、エラーが発生したことを定義した数値計算から無効な結果を返すバッファーに書き込むことができます。このような場合の適切な方法は、アプリケーションによって異なります。

まとめ

この章では、同期エラーと非同期エラーについて説明し、発生する可能性のあるエラーを何も管理しない場合に予想されるデフォルトの動作について説明し、アプリケーションの制御されたポイントで非同期エラーを処理するメカニズムについて説明しました。エラー管理はソフトウェア・エンジニアリングの主要なトピックであり、多くのアプリケーションで記述されるコードのかなりの部分を占めています。SYCL のエラー処理は、既にある C++ の知識と統合してどのようなエラー管理にも統合できる柔軟なメカニズムを提供します。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。
（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 6 章 統合共有メモリー

ここから 2 つの章では、データの管理方法についてさらに詳しく説明します。互いに補完し合う 2 つの異なるアプローチがあります: 統合共有メモリー (USM) とバッファー。USM は、バッファーとは異なるレベルでメモリーを抽象化します。USM はポインターを使用し、バッファーはより高レベルのインターフェイスです。本章では、USM に注目します。次の章でバッファーについて説明します。

バッファーを使用する必要性が明確である場合を除き、USM から始めるのがよいでしょう。USM は、通常の C++ ポインターを介してメモリーの読み取りと書き込みを行うポインターベースのモデルです。

USM を使用する理由は?

USM は C++ ポインターに基づいているため、既存のポインターベースの C++ コードからの移行には最適です。ポインターをパラメーターとして受け取る既存の関数は、変更することなく引き続き動作します。ほとんどの場合、必要となる変更は、既存の `malloc` または `new` の呼び出しを、この章の後半で説明する USM 固有の割り当てルーチンに置き換えるだけです。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_6

割り当てタイプ

USM は C++ ポインターに基づいていますが、すべてのポインターが同じように作成されるわけではありません。USM では、それぞれ固有のセマンティクスを持つ 3 つの異なるタイプの割り当てを定義します。デバイスは、すべてのタイプ（または任意のタイプ）の USM 割り当てをサポートしない場合があります。デバイスがサポートする割り当てを照会する方法については、後で説明します。3 つのメモリー割り当てとその特徴を図 6-1 にまとめます。

タイプ	説明	ホストでアクセス可能か?	デバイスでアクセス可能か?	割り当て場所
`device`	デバイスメモリーに割り当て	x	はい	デバイス
`host`	ホストメモリーに割り当て	はい	はい	ホスト
`shared`	ホストとデバイスで共有される割り当て	はい	はい	ホストとデバイス間で転送可能

図 6-1. USM 割り当てタイプ

デバイス割り当て

最初の割り当てタイプは、GDDR/DDR や HBM など、デバイスに接続されたメモリーへのポインターアクセスを行うために必要なものです。デバイス割り当ては、特定のデバイスで実行されるカーネルによって読み書きできますが、ホスト上で実行されるコードからは直接アクセスできません。ホスト上のデバイス割り当てにアクセスすると、誤ったデータが取得されたり、エラーによりプログラムがクラッシュする可能性があります。明示的な USM の `memcpy` を使用して、ホストとデバイス間でデータをコピーする必要があります。このメカニズムは、2 つのメモリー位置間でコピーするデータの量を指定します。これについては、この章の後半で説明します。

ホスト割り当て

2 番目のタイプ割り当ては、ホストとデバイスの間でデータを手動でコピーする必要がないため、デバイス割り当てよりも簡単に利用できます。ホスト割り当ては、ホストとデバイスの両方からアクセス可能なホストメモリーを割り当てます。この割り当ては、デバイスからアクセスできますが、デバイスに接続されたメモリーに転送することはできません。代わりに、カーネルは多くの場合、PCI Express などの低速バスを介して、このメモリーから*リモート*で読み取りまたは書き込みを行います (CPU デバイスと統合 GPU デバイスでは違いはありません)。これには利便性とパフォーマンスのトレードオフがあります。ホスト割り当てによって生じるアクセスコストは高くなりますが、ホスト割り当てを利用する利点はほかにもあります。例えば、めったにアクセスされないデータ、デバイスに接続されたメモリー内に収まらない大きなデータセット、または次に説明する共有割り当てなどの代替手段をデバイスがサポートしていない可能性があります。

共有割り当て

3 番目のタイプの割り当てでは、デバイス割り当てとホスト割り当ての両方の属性が組み合わされ、ホスト割り当てのプログラマーの利便性とデバイス割り当てによって得られる優れたパフォーマンスが得られます。ホスト割り当てと同様に、共有割り当ては、ホストとデバイスの両方からアクセスできます。ホスト割り当てとの違いは、共有割り当てでは、ホストメモリーとデバイスに接続されたメモリー間で、プログラマーからは透過に自動でデータが転送されることです。割り当てがデバイスに転送されると、デバイスで実行されているカーネルは、ホストからリモートでアクセスするよりも高いパフォーマンスでアクセスできます。 ただし、共有割り当てによって、欠点なしにすべての利点が得られるわけではありません。

自動転送はさまざまな方法で実装できます。ランタイムが共有割り当てをどのように実装した場合でも、通常はレイテンシーが増加する代償を払うことになります。デバイス割り当てを使用すると、コピーする必要があるメモリーの量を正確に把握し、早期にコピーを開始するようスケジュールできます。

自動転送のメカニズムは状況を予測することができません。場合によってはカーネルがデータにアクセスするまでデータの移動を開始しません。その場合、カーネルはデータの移動が完了するまで待機（ブロック）してから、実行を続行する必要があります。他のケースでは、ランタイムはカーネルがアクセスするデータの量を正確に把握できず、必要以上に多くのデータを移動してカーネルの待機時間を増加させる可能性があります。

また、共有割り当ては転送できますが、必ずしもすべての SYCL 実装が転送するわけではないことにも注意してください。ほとんどの実装では、転送を伴う共有割り当てを実装すると予想されますが、一部のデバイスではホスト割り当てと同じように実装することがあります。そのような実装では、割り当てはホストとデバイスの両方で引き続き使用されますが、転送の実装によるパフォーマンスの向上は得られない可能性があります。

メモリー割り当て

USM を使用すると、ニーズと好みに応じてさまざまな方法でメモリーを割り当てできます。ただし、すべての方法を詳しく説明する前に、USM 割り当てが通常の C++ 割り当てとどのように異なるか説明する必要があるでしょう。

何を知る必要があるか？

通常の C++ プログラムでは、`new`、`malloc`、アロケーターなど、複数の方法でメモリーを割り当てます。どの構文を選択しても、メモリーの割り当ては最終的にはホストのオペレーティング・システムのシステム・アロケーターによって実行されます。C++ でメモリーを割り当てるときに考慮すべきことは、“必要なメモリーの量はどれくらいか” と “割り当て可能なメモリーの量はどれくらいか” だけです。USM では割り当てを実行する前に追加の情報が必要です。

まず、USM 割り当てでは、デバイス、ホスト、共有のいずれのタイプの割り当てが必要であるか指定する必要があります。期待する動作を得るには、適切なタイプの割り当てを要求しなければなりません。次に、すべての USM 割り当てでは、割り当てが行われるコンテキスト・オブジェクトを指定する必要があります。本書にあるほとんどの例では、代わりにキュー・オブジェクト（コンテキストを提供するオブジェクト）を渡します。コンテキスト・オブジェクトについては、本書ではこれまで説明してこなかったので、ここで少し説明しておきます。コンテキストは、カーネルを実行できるデバイスまたはデバイスセットを表します。コンテキストは、ランタイムが実行中の処理状態を保存しておくのに便利な場所と考えることができます。ほとんどの SYCL プログラムでは、プログラマーがコンテキストを渡す以外でコンテキストと直接やり取りすることはありません。
第 13 章では、コンテキストに関するヒントをいくつか紹介します。

USM 割り当ては、異なるコンテキスト間で使用できることは保証されません。すべての USM 割り当て、キュー、およびカーネルが同じコンテキスト・オブジェクトを共有することが重要です。通常、このコンテキストは、デバイスにワークを送信するキューから取得できます。

最後に、デバイスの割り当て（および一部の共有割り当て）では、割り当てるメモリーを提供するデバイスを指定する必要もあります。これは、デバイスのメモリーをオーバーサブスクライブしないために重要です（デバイスがこれをサポートできる場合を除きます。これについては、この章の後半でデータ移行について説明するときに詳しく説明します）。USM 割り当てルーチンは、これらの追加パラメーターを追加することで、C++ の類似するルーチンと区別できます。

複数のスタイル

時には、単一のオプションですべてを満足するのは不可能であることが判明します。これは、紅茶よりもコーヒーを好む人や、vi よりも emacs を好む人がいるのと同じです。割り当てインターフェイスがどのようなものであるべきかをプログラマーに尋ねると、異なる答えが返ってきます。USM はこのような多様な選択を受け入れ、さまざまな種類の割り当てインターフェイスを提供します。

これらの異なるスタイルには、C スタイル、C++ スタイル、および C++ アロケーター・スタイルがあります。ここでは、それぞれを説明し、類似点と相違点を示します。

C 風の割り当て

最初のスタイルの割り当て関数（図 6-2 にリストされ、後の図 6-6 と 6-7 の例で使用されるます）は、C のメモリー割り当てをモデルにしています。`malloc` 関数は、割り当てに一定数のバイトを受け取り、`void *` ポインターを返します。このスタイルの関数は型に依存しませんが、割り当てるバイトの合計数を指定する必要があります。つまり、タイプ `x` の `N` 個のオブジェクトを割り当てるには、合計 `N * sizeof(X)` バイトを要求する必要があります。返されるポインターは `void*` 型であるため、適切な `x` 型へのポインターにキャストする必要があります。このスタイルは非常にシンプルですが、サイズの計算と型キャストが必要なため冗長になる可能性があります。

この割り当てスタイルは、さらに名前付き関数と単一関数の 2 つのカテゴリーに分類できます。2 つの違いは、必要な USM 割り当ての種類を指定する方法です。名前付き関数（`malloc_device`、`malloc_host`、`malloc_shared`）では、USM 割り当てのタイプが関数名にエンコードされます。単一関数 `malloc` では、USM 割り当てのタイプを追加パラメーターとして指定します。どちらが優れているというわけではなく、どちらを使用するかは好みによって決まります。

アライメントについて曖昧にして先に進むことはできません。`malloc` の各バージョンには、対応する `aligned_alloc` もあります。`malloc` 関数は、デバイスのデフォルト動作に合わせてメモリーを返します。割り当てが成功すると、有効なアライメントを持つ正当なポインターが返されますが、アライメントを手動で指定する方がよい場合もあります。このような場合、割り当て時にアラインメントを指定する、`aligned_alloc` バリアントのいずれかを使用する必要があります。適切なアラインメントは 2 の累乗です。

多くのデバイスでは、割り当てはハードウェアの機能に合わせて最大限にアライメントされることに注意してください。そのため、割り当てを 4 バイト、8 バイト、16 バイト、または 32 バイトの境界に要求しても、実際には要求したものより大きなアライメントになる場合があります。

```
// 名前付き関数
void *malloc_device(size_t size, const device &dev,
                    const context &ctxt);
void *malloc_device(size_t size, const queue &q);
void *aligned_alloc_device(size_t alignment, size_t size,
                           const device &dev,
                           const context &ctxt);
void *aligned_alloc_device(size_t alignment, size_t size,
                           const queue &q);

void *malloc_host(size_t size, const context &ctxt);
void *malloc_host(size_t size, const queue &q);
void *aligned_alloc_host(size_t alignment, size_t size,
                         const context &ctxt);
void *aligned_alloc_host(size_t alignment, size_t size,
                         const queue &q);

void *malloc_shared(size_t size, const device &dev,
                    const context &ctxt);
void *malloc_shared(size_t size, const queue &q);
void *aligned_alloc_shared(size_t alignment, size_t size,
                           const device &dev,
                           const context &ctxt);
void *aligned_alloc_shared(size_t alignment, size_t size,
                           const queue &q);

// 単一関数
void *malloc(size_t size, const device &dev,
             const context &ctxt, usm::alloc kind);
void *malloc(size_t size, const queue &q, usm::alloc kind);
void *aligned_alloc(size_t alignment, size_t size,
                    const device &dev, const context &ctxt,
                    usm::alloc kind);
void *aligned_alloc(size_t alignment, size_t size,
                    const queue &q, usm::alloc kind);
```

図 6-2. C スタイルの USM 割り当て関数

C++ 風の割り当て

次の USM 割り当て関数（図 6-3）は、最初と似ていますが、C++ の外観と操作性に近いものです。割り当てルーチンには、名前付きバージョンと単一関数バージョンがあり、さらにデフォルト バージョンとユーザー指定のアライメント・バージョンもあります。これらの違いは、関数が `T` 型の `Count` オブジェクトを割り当てて、`T *` 型のポインターを返す C++ テンプレート関数になっていることです。最新の C++ を利用すると、割り当ての合計サイズをバイト単位で手動計算したり、返されたポインターを適切な型にキャストする必要がなくなるため、作業が簡素化されます。これにより、コード内の表現がコンパクトになり、エラーが発生しにくくなる傾向があります。ただし、C++ の “`new`” とは異なり、`malloc` スタイルのインターフェイスは割り当てられるオブジェクトのコンストラクターを呼び出しません。つまり、その型に適合する十分なバイト数を割り当てるだけです。

この種類の割り当ては、USM を前提に作成された新しいコードの開始点として適しています。前述の C スタイルは、すでに C または C++ `malloc` を多用している既存の C++ コードの良い出発点であり、これに USM を適用します。

```
// 名前付き関数
template <typename T>
T *malloc_device(size_t Count, const device &Dev,
                 const context &Ctxt);
template <typename T>
T *malloc_device(size_t Count, const queue &Q);
template <typename T>
T *aligned_alloc_device(size_t Alignment, size_t Count,
                        const device &Dev,
                        const context &Ctxt);
template <typename  T>
T *aligned_alloc_evice(size_t Alignment, size_t Count,
                       const queue &Q);

template <typename T>
T *malloc_host(size_t Count, const context &Ctxt);
template <typename T>
T *malloc_host(size_t Count, const queue &Q);
template <typename T>
T *aligned_alloc_host(size_t Alignment, size_t Count,
                      const context &Ctxt);
template <typename T>
T *aligned_alloc_host(size_t Alignment, size_t Count,
                      const queue &Q);

template <typename T>
T *malloc_shared(size_t Count, const device &Dev,
                 const context &Ctxt);
template <typename T>
T *malloc_shared(size_t Count, const queue &Q);
template <typename T>
T *aligned_alloc_shared(size_t Alignment, size_t Count,
                        const device &Dev,
                        const context &Ctxt);
template <typename T>
T *aligned_alloc_shared(size_t Alignment, size_t Count,
                        const ueue &Q);

// 単一関数
template <typename T>
T *malloc(size_t Count, const evice &Dev,
          const context &Ctxt, usm::alloc Kind);
template <typename T>
T *malloc(size_t Count, const queue &Q, usm::alloc Kind);
template <typename T>
T *aligned_alloc(size_t Alignment, size_t Count,
                 const device &Dev, const context &Ctxt,
                 usm::alloc Kind);
template <typename T>
T *aligned_alloc(size_t Alignment, size_t Count,
                 const queue &Q, usm::alloc Kind);
```

図 6-3. C++ スタイルの USM 割り当て関数

C++ アロケーター

USM 割り当ての最終の種類（図 6-4）は、前述よりもさらに最新の C++ を採用しています。これは C++ アロケーター・インターフェイスに基づいており、`std::vector` などのコンテナー内で直接的または間接的にメモリー割り当てを行うオブジェクトを定義します。このアロケーターは、コードでコンテナー・オブジェクトを多用し、メモリーの割り当てと割り当て解除をユーザーから隠蔽して、コードを簡素化し、バグが発生する可能性を減らすのに最も役立ちます。

```
template <typename T, usm::alloc AllocKind,
          size_t Alignment = 0>
class usm_allocator {
 public:
   using value_type  = T;
   using propagate_on_container_copy_assignment =
        std::true_type;
   using propagate_on_container_move_assignment =
        std::true_type;
   using propagate_on_container_swap = std::true_type;

 public:
   template <typename U>
    struct rebind {
       typedef usm_allocator<U, AllocKind, Alignment> other;
   };

   usm_allocator() = delete;
    usm_allocator(const context& syclContext,
                          const device& syclDevice,
                          const roperty_list& propList = {});
    usm_allocator(const queue& syclQueue,
                          const property_list& propList = {});
    usm_allocator(const usm_allocator& other);
    usm_allocator(usm_allocator&&) noexcept;
    usm_allocator& operator=(const usm_allocator&);
    usm_allocator& operator=(usm_allocator&&);

    template <class U>
    usm_allocator(usm_allocator<U, AllocKind,
                                Alignment> const&) oexcept;

   /// メモリーを割り当て
   T*  allocate(size_t count);

   /// メモリーを解放
   void deallocate(T* Ptr, size_t count);

   /// 同等性比較
   ///
   /// アロケータは、同じ種類の USM、アライメント、コンテキスト、
   /// およびデバイスである場合にのみ同等であると見なされます。
   template <class U, usm::alloc AllocKindU,
            size_t AlignmentU>
   friend bool operator==(
           const usm_allocator<T, AllocKind, Alignment>&,
           const usm_allocator<U, AllocKindU, AlignmentU>&);

   /// 不等比較
   ///
   /// アロケーターは、USM、アライメント、コンテキスト、またはデバイスが
   /// 同じでない場合にのみ不等比較を行います
   template <class U, usm::alloc AllocKindU,
             size_t AlignmentU>
   friend bool operator!=(
         const sm_allocator<T, AllocKind, Alignment>&,
         const usm_allocator<U, AllocKindU, AlignmentU>&);
};
```

図 6-4. C++ アロケーター・スタイルの USM 割り当て関数

メモリーを解放

プログラムが割り当てたメモリーは、最終的には割り当て解除される必要があります。USM は、malloc または alignment_malloc 関数によって割り当てられたメモリーを解放する free メソッドを定義します。この free メソッドは、メモリーが割り当てられたコンテキストも追加パラメーターとして受け取ります。キューをコンテキストの代わりに使用することもできます。メモリーが C++ アロケーター・オブジェクトを使用して割り当てられている場合、そのオブジェクトを使用してメモリーの割り当てを解除する必要があります。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   queue q;

   // N 個の float を割り当て

   // C スタイル
   float *f1 = static_cast<float *>(malloc_shared(
       N * sizeof(float), q.get_device(), q.get_context()));

   // C++ スタイル
   float *f2 = malloc_shared<float>(N, q);

   // C++ アロケーター・スタイル
   usm_allocator<float, usm::alloc::shared> alloc(q);
   float *f3 = alloc.allocate(N);

   // 割り当てを解放
   free(f1, q.get_context());
   free(f2, q);
   alloc.deallocate(f3, N);

   return 0;
}
```

図 6-5. 3 つのスタイルの割り当て

割り当ての例

図 6-5 では、前述の 3 つのスタイルを使用して同じ割り当てを行う方法を示しています。この例では、`N` 個の単精度浮動小数点数を共有割り当てとして割り当てます。最初の割り当て `f1` では、C スタイルの `void *` を返す `malloc` ルーチンが使用されます。この割り当てでは、キューから取得したデバイスとコンテキストを明示的に渡します。また、結果を `float *` にキャストし直す必要があります。2 番目の割り当て `f2` は同じことを行いますが、C++ スタイルのテンプレート化された `malloc` を使用します。要素の型である `float` を割り当てルーチンに渡すため、割り当てる `float` の数を指定するだけでよく、結果をキャストする必要はありません。また、デバイスとコンテキストの代わりにキューを受け取る形式も使用し、非常にシンプルでコンパクトなステートメントを記述できます。3 番目の割り当て `f3` では、USM C++ アロケーター・クラスが使用されます。適切なタイプのアロケーター・オブジェクトをインスタンス化し、そのオブジェクトを使用して割り当てを行います。 最後に、各割り当てを適切に解除する方法を示します。

データ管理

USM を使用してメモリーを割り当てる方法がわかったので、次にデータ管理の方法を説明します。これをデータの初期化とデータの移動という 2 つに分けて考えることができます。

初期化

データの初期化は、計算を開始する前にメモリーに値を格納することです。一般的な初期化パターンの例として、割り当てを使用する前にゼロで埋めることが挙げられます。USM 割り当てを初期化するには、いくつかの方法があります。まず、これを実行するカーネルを記述できます。データセットが特に大きい場合、または初期化に複雑な計算が必要な場合は、初期化を並列で実行できるため（また、初期化されたデータをデバイスにすぐに転送できるため）、これは適切な方法です。

2 番目に、割り当てたすべての要素に対して各要素をゼロに設定するループをホストコードに実装できます。ただし、このアプローチには潜在的な問題があります。ホストと共有割り当てはホスト上でアクセス可能であり、ループは問題なく機能します。

ただし、デバイスの割り当てはホスト上でアクセスできないため、ホストコード内のループではそれらに書き込むことができません。次に 3 番目の選択肢を説明します。

`memset` 関数は、この初期化パターンを効率良く実装するように設計されています。USM は、ハンドラークラスとキュークラスの両方のメンバー関数である `memset` のバージョンを提供します。これには 3 つの引数が必要です。設定するメモリーのベースアドレスを表すポインター、設定するバイトパターンを表すバイト値、およびそのパターンを設定するバイト数です。ホスト上のループとは異なり、`memset` は並列に実行され、デバイス割り当てでも機能します。

`memset` は便利な操作ですが、割り当てに転送するバイトパターンのみを指定できるという点で、かなり制限があります。USM には、メモリーを任意のパターンで埋めることができる `fill` メソッド（ハンドラークラスとキュークラスのメンバーとして）も用意されています。`fill` メソッドは、割り当てに書き込むパターンの型に基づいてテンプレート化された関数です。これを `int` でテンプレート化すると、割り当てを 32 ビットの整数値 “42” で埋めることができます。`memset` と同様に、`fill` は 3 つの引数を取ります。割り当てのベースアドレスへのポインター、埋める値、およびその値を割り当てに書き込む回数です。

データ移動

データの移動は、おそらく USM を理解する上で最も重要なことです。適切なデータが、適切な場所に、適切なタイミングで存在しない場合、プログラムは誤った結果を生成します。USM は、データ管理に使用できる明示的と暗黙的の 2 つの方法を定義します。

使用する方法の選択は、ハードウェアがサポートする、または使用する USM 割り当ての種類に関係します。

明示的

USM が提供する最初の方法は、明示的なデータ移動です（図 6-6）。プログラマーは、ホストとデバイス間でデータを明示的にコピーする必要があります。これは、ハンドラークラスとキュークラスの両方にある `memcpy` メソッドを呼び出して実行できます。`memcpy` メソッドは、デスティネーション・メモリーのポインター、ソースメモリーのポインター、およびホストとデバイス間でコピーするバイト数の 3 つの引数を受け取ります。コピーをどの方向に行うか指定する必要はありません。これは、ソースポインターとデスティネーション・ポインターによって暗黙的に決定されます。

明示的なデータ移動の最も一般的な用途は、ホスト上でアクセスできない USM 内のデバイス割り当てとの間でコピーを行うことです。明示的なデータのコピーを指示する文を追加する必要があるため、プログラマーの労力が必要になります。さらに、これはバグの原因となる可能性もあります。コピーが誤って省略されたり、間違った量のデータがコピーされたり、ソースまたはデスティネーション・ポインターが正しくなかったりする可能性があります。

ただし、明示的なデータ移動は欠点ばかりではありません。これにより、データの移動を完全に制御できる大きな利点が得られます。一部のアプリケーションでは、最高のパフォーマンスを達成するために、コピーされるデータの量とタイミングの両方を制御することが非常に重要です。理想的には、可能な限り計算とデータの移動をオーバーラップして、ハードウェアが高い利用率を維持するようにします。

その他のタイプの USM 割り当て（ホストと共有）は、ホストとデバイスの両方でアクセス可能であり、デバイスに明示的にコピーする必要はありません。これにより、USM でのデータ移動の別の方法が考えられます。

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

   std::array<int, N> host_array;
   int* device_array = malloc_device<int>(N, q);
   for (int i = 0; i < N; i++) host_array[i] = N;

   q.submit([&](handler& h) {
     // host_array を device_array にコピー
     h.memcpy(&host_array[0], device_array, N * sizeof(int));
   });
   q.wait();     // 今は必要（後でもっと良い方法を学ぶ）

   q.submit([&](handler& h) {
     h.parallel_for(N, [=](id<1> i) { device_array[i]++; });
   });
   q.wait();     // 今は必要（後でもっと良い方法を学ぶ）

   q.submit([&](handler& h) {
     // device_array を host_array にコピー
     h.memcpy(&host_array[0], device_array, N * sizeof(int));
   });
   q.wait();     // 今は必要（後でもっと良い方法を学ぶ）

   free(device_array, q);
   return 0;
}
```

図 6-6. USM 明示的なデータ移動の例

暗黙的

USM が提供する 2 番目の方法は、暗黙的なデータ移動です（使用例を図 6-7 に示します）。この方法では、データの移動は*暗黙的*に、つまりユーザーからの入力を必要とせずに行われます。暗黙的なデータ移動では、USM ポインターを介してデータに直接アクセスできるため、`memcpy` の呼び出しを挿入する必要はありません。代わりに、データが使用されるときに、正しい場所で利用できるようにするのがシステムの役割になります。

ホスト割り当てには、実際にデータの移動が発生するか議論の余地があります。定義上、それらは常にホストメモリーへのポインターのままであるため、特定のホストポインターが示すメモリーはデバイスに保存できません。ただし、デバイス上でホスト割り当てにアクセスすると、データの移動が発生します。メモリーがデバイスに転送される代わりに、読み書きされる値は適切なインターフェイスを介してカーネルとの間で転送されます。これは、データがデバイス上に常駐する必要がない、ストリーミング・カーネルに役立ちます。

暗黙的なデータ移動は主に USM 共有割り当てに関係します。このタイプの割り当ては、ホストとデバイスの両方でアクセス可能であり、さらにホストとデバイス間で転送できます。重要なことは、この転送は、別の場所にあるデータにアクセスするだけで自動的に、つまり暗黙的に行われるという点です。次に、共有割り当てのデータ転送に関して考慮すべきいくつかの事項について説明します。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;

  int* host_array = malloc_host<int>(N, q);
  int* shared_array = malloc_shared<int>(N, q);
  for (int i = 0; i < N; i++) host_array[i] = i;

  q.submit([&](handler& h) {
    h.parallel_for(N, [=](id<1> i) {
      // デバイス上の shared_array と host_array にアクセス
      shared_array[i] = host_array[i] + 1;
      });
  });
  q.wait();

  free(shared_array, q);
  free(host_array, q);
  return 0;
}
```

図 6-7. USM 暗黙的なデータ移動の例

転送

明示的なデータ移動により、発生するデータ移動の量を制御します。暗黙的なデータ移動では、システムがこれを処理しますが、効率的でない可能性があります。SYCL ランタイムは予言者ではないため、アプリケーションがアクセスするデータを事前に予測することはできません。さらに、ポインター解析はコンパイラーにとって非常に難しい課題であり、カーネル内で使用される可能性のあるすべての割り当てを正確に識別できない可能性があります。したがって、暗黙的なデータ移動のメカニズム実装では、USM をサポートするデバイスの機能を基に異なる決定が行われる可能性があり、これは共有割り当ての使い方とパフォーマンスの両方に影響します。

デバイスのパフォーマンスが非常に高ければ、オンデマンドでメモリーを転送できる可能性があります。この場合、ホストまたはデバイスが目的の場所にない割り当てにアクセスすると、データの移動が発生します。

オンデマンド・データは、USM 共有ポインターがどこからでもアクセスでき、そのまま機能するという望ましいセマンティクスを提供するため、プログラミングを大幅に簡素化します。デバイスがオンデマンド移行をサポートできない場合（第 12 章でデバイスの機能を照会する方法を説明）、共有ポインターの使用方法に追加の*制限*を付けることで、同じセマンティクスを保証できる可能性があります。

USM 共有割り当てが制限された形式は、共有ポインターにアクセスできるタイミングと場所、および共有割り当てのサイズがどの程度まで許容されるかを制御します。デバイスが要求に応じてメモリーを移行できない場合、ランタイムは保守的になり、カーネルがデバイスに接続されたメモリー内の任意の割り当てにアクセスする可能性があると想定する必要があります。 これによって、いくつかの結果がもたらされます。

まず、ホストとデバイスが同時に共有割り当てにアクセスしてはならないことを意味します。代わりに、アプリケーションは段階的に交互にアクセスする必要があります。ホストは割り当てにアクセスし、カーネルはそのデータを使用して計算し、最後にホストは結果を読み取ることができます。この制限がなければ、ホストはカーネルが現在アクセスしている割り当てに自由にアクセスできるようになります。このような同時アクセスは通常、デバイス・メモリー・ページの粒度で発生します。ホストが 1 つのページにアクセスしている間に、デバイスは別のページにアクセスすることができます。同じデータへのアトミックなアクセスについては、第 19 章で説明します。プログラマーは、デバイスがこの制限を受けているかどうかを照会できます。デバイスの照会メカニズムについては、後述します。

この制限された形式の共有割り当ての結果、割り当てがデバイスに接続されているメモリーの合計量によって制限されます。デバイスがオンデマンドでメモリーを移行できない場合、異なるデータを取り込むスペースを確保するため、データをホストに移行できません。デバイスがオンデマンド移行をサポートしている場合、接続されたメモリーをオーバーサブスクライブして、カーネルがデバイスのメモリーに格納できる量よりも多くのデータを計算するようにできます。ただし、この柔軟性により、追加のデータ移動によるパフォーマンスの低下が生じる可能性があります。

細粒度の制御

デバイスが共有割り当てのオンデマンド転送をサポートしている場合、メモリーが存在しない場所にアクセスされた後に、データの移動が行われます。ただし、データの移動が完了するのを待機している間に、カーネルが停止する可能性があります。次に実行されるステートメントによって、さらに多くのデータ移動が発生し、カーネルの実行に遅延が発生する可能性があります。

SYCL は、自動転送メカニズムのパフォーマンスを改善する方法を提供します。これは、`prefetch` と `mem_advise` という 2 つの関数を定義することで行われます。図 6-8 は、それぞれの簡単な使い方を示しています。これらの関数を使用すると、カーネルがデータにアクセスする方法についてランタイムにヒントを与えることができ、ランタイムはカーネルがデータにアクセスする*前に*データの移動を開始することを選択できます。この例では、送信メソッド（コマンドグループ）に渡されるラムダ内ではなく、キュー・オブジェクトで `parallel_for` を直接呼び出すキューのショートカット・メソッドを使用していることに注意してください。

```
#include <sycl/sycl.hpp>
using namespace sycl;

// 適切な値はハードウェアによって異なります
constexpr int BLOCK_SIZE = 42;
constexpr int NUM_BLOCKS = 2500;
constexpr int N = NUM_BLOCKS * BLOCK_SIZE;

int main() {
  queue q;
  int *data = malloc_shared<int>(N, q);
  int *read_only_data = malloc_shared<int>(BLOCK_SIZE, q);

  for (int i = 0; i < N; i++) {
     data[i] = -i;
  }

  // 初期化後に更新されません
  for (int i = 0; i < BLOCK_SIZE; i++) {
     read_only_data[i] = i;
  }

  // このデータを "読み取り専用" としてマークすると、ランタイムはデータを
  // ホストから移行するのではなく、デバイスにコピーできるようになります。
  // 実際の値はバックエンドによって示されます。
  int HW_SPECIFIC_ADVICE_RO = 0;
  q.mem_advise(read_only_data, BLOCK_SIZE,
               HW_SPECIFIC_ADVICE_RO);
  event e = q.prefetch(data, BLOCK_SIZE * sizeof(int));

  for (int b = 0; b < NUM_BLOCKS; b++) {
    q.parallel_for(range{BLOCK_SIZE}, e, [=](id<1> i) {
       data[b * BLOCK_SIZE + i] += read_only_data[i];
    });
    if ((b + 1) < NUM_BLOCKS) {
       // 次のブロックをプリフェッチ
       e = q.prefetch(data + (b + 1) * BLOCK_SIZE,
                      BLOCK_SIZE * sizeof(int));
     }
  }
  q.wait();

  free(data, q);
  free(read_only_data, q);
  return 0;
}
```

図 6-8. プリフェッチと mem_advise によるきめ細かな制御

これを行う最も簡単な方法は、プリフェッチを呼び出すことです。この関数は、ハンドラーまたはキュークラスのメンバー関数として呼び出され、ベースポインターとバイト数を受け取ります。これにより、特定のデータがデバイス上で使用されることをランタイムに通知し、そのデータの転送を迅速に開始できるようになります。理想的には、これらのプリフェッチ・ヒントを早い段階で発行し、カーネルがデータにアクセスするときにはデータがすでにデバイス上に常駐するようにして、前述のレイテンシーを排除します。

SYCL が提供するもう 1 つの関数は `mem_advise` です。この関数を使用すると、カーネル内でメモリーがどのように使用されるかデバイス固有のヒントを提供できます。指定できる*アドバイス*の例としては、「データはカーネル内で読み取られるだけで、書き込まれることはない」、などがあります。その場合、システムはデバイス上のデータのコピー（複製）が可能であると認識できるため、カーネルの完了後にホストのデータを更新する必要がなくなります。ただし、`mem_advise` に渡される*アドバイス*は、特定のデバイスに固有であるため、この関数を使用する前にハードウェアのドキュメントを確認してください。

クエリー

すべてのデバイスが USM のすべての機能をサポートしているわけではありません。つまり、プログラムをさまざまなデバイス間で移植可能にしたい場合、すべての USM 機能が利用可能であると想定してはなりません。USM は照会できる項目をいくつか定義しています。これらの照会は、ポインタークエリーとデバイス機能クエリーの 2 つのカテゴリーに分けられます。図 6-9 は、それぞれの簡単な使い方を示しています。

USM のポインタークエリーは 2 つの疑問に答えます。最初の疑問は、“ポインターはどのタイプの USM 割り当てを指しているか” です。`get_pointer_type` 関数は、ポインターと SYCL コンテキストを受け取り、`usm::alloc` 型の結果を返します。この結果には、*ホスト*、*デバイス*、*共有*、または*不明*の 4 つの値があります。2 番目の疑問は、“USM ポインターはどのデバイスに対して割り当てられたのか” です。関数 `get_pointer_device` にポインターとコンテキストを渡し、デバイス・オブジェクトを取得できます。

これは、ホスト割り当てではあまり意味がないため、主にデバイスまたは共有USM 割り当てで使用されます。SYCL 仕様では、ホスト割り当てはコンテキスト内の最初のデバイスが返されると規定されています。これは、例外のスローを回避する以外に特別な理由はありません。しかし、USM 割り当てタイプでテンプレート化される可能性のあるコードでは少し奇妙に思えます。

USM で提供される 2 番目のタイプのクエリーは、デバイスの機能に関するものです。USM には、デバイス・オブジェクトで `has` を呼び出すことで照会できる独自のデバイスアスペクトのリストがあります。これらのクエリーを使用して、デバイスでサポートされている USM 割り当ての種類をテストできます。さらに、共有割り当てがホストとデバイスから同時にアクセスできるか照会することもできます。クエリーの完全なリストを図 6-10 に示します。第 12 章では、クエリーのメカニズムについて詳しく説明します。

```
#include <sycl/sycl.hpp>
using namespace sycl;
namespace dinfo = info::device;
constexpr int N = 42;

template <typename T>
void foo(T data, id<1> i) {
  data[i] = N;
}

int main() {
   queue q;
   auto dev = q.get_device();
   auto ctxt = q.get_context();
   bool usm_shared = dev.has(aspect::usm_shared_allocations);
   bool usm_device = dev.has(aspect::usm_device_allocations);
   bool use_USM = usm_shared || usm_device;

   if (use_USM) {
      int *data;
      if (usm_shared) {
         data = malloc_shared<int>(N, q);
      } else  /* use device allocations */ {
         data = malloc_device<int>(N, q);
      }
      std::cout << "Using USM with "
                << ((get_pointer_type(data, ctxt) ==
                      usm::alloc::shared)
                          ? "shared"
                          : "device")
                << " allocations on "
                << get_pointer_device(data, ctxt)
                      .get_info<dinfo::name>()
                << "\n";
      q.parallel_for(N, [=](id<1> i) { foo(data, i); });
      q.wait();
      free(data, q);
  } else  /* バッファーを使用 s */ {
     buffer<int, 1> data{range{N}};
     q.submit([&](handler  &h) {
       accessor a(data, h);
       h.parallel_for(N, [=](id<1> i) { foo(a, i); });
     });
     q.wait();
   }
   return 0;
}
```

図 6-9. USM ポインターとデバイスに関するクエリー

アスペクト	説明
`aspect::usm_device_allocations`	このデバイスはデバイス割り当てをサポートしています
`aspect::usm_host_allocations`	このデバイスはホスト割り当てをサポートしています
`aspect::usm_atomic_host_allocations`	このデバイスは、デバイスによって行われるホスト割り当てをサポートしています
`aspect::shared_allocations`	このデバイスは共有割り当てをサポートしています
`aspect::atomic_shared_allocations`	このデバイスは共有割り当てをサポートしており、ホストとデバイスは共有割り当てに同時にアクセスしてアトミックに更新できます
`aspect::usm_system_allocations`	このデバイスは割り当ての使用をサポートしています

図 6-10. USM デバイスアスペクト

あと一つだけ

まだ取り上げていない USM の形式がもう 1 つあります。この章で説明した USM の形式ではすべて、特別な割り当て関数を使用する必要があります。これは大きな負担ではありませんが、`malloc` または `new` 演算子の形式でシステム・アロケーターを使用する従来の C++ コードを変更する必要があります。CPU など、一部のデバイスではこの要件は必要ないかもしれませんが、ほとんどのアクセラレーター・ デバイスでは依然として必要です。ここでは、移植性を高めるため USM 割り当て関数を使用する方法について説明しました。ただし、システム・アロケーターをサポートするアクセラレーターの設計が今後さらに増えると考えられます。このようなデバイスにより、プログラマーは適切なタイプの USM メモリーを割り当てたり、適切なタイミングで正しいデータをコピーする手間が省けるため、プログラムが大幅に簡素化されます。
ある意味では、システム・アロケーターのサポートは USM の最終的な進化と見なすことができます。つまり、特別な割り当て関数の使用を必要とせずに、共有 USM 割り当ての利点がもたらされることになります。

まとめ

この章では、データ管理にポインターベースを使用する統合共有メモリーについて説明しました。USM が定義する 3 つの割り当てについて説明しました。USM を使用してメモリーを割り当てたり、割り当て解除するさまざまな方法と、デバイス割り当てではデータ移動を私たち（プログラマー）が明示的に制御する方法、ホストまたは共有の割り当てではシステムによって暗黙的に制御する方法について説明しました。最後に、デバイスがサポートするさまざまな USM 機能を照会する方法と、プログラム内の USM ポインターに関する情報を照会する方法について説明しました。

本書では同期についてはまだ詳しく説明していないため、USM については、後の章でスケジュール、通信、および同期について触れるときに詳しく説明します。具体的には、USM に関するこれらの追加事項については、第 8 章、第 9 章、および第 19 章で説明します。

次の章では、データ管理の 2 番目の方法であるバッファーについて説明します。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 7 章 バッファー

この章では、バッファーの抽象化について説明します。前の章では、ポインターベースのデータ管理である統合共有メモリー（USM）について説明しました。USM では、メモリーがどこに配置され、どこからアクセスできるか考えなければなりません。バッファーの抽象化は、これらをプログラマーから隠匿する高レベルのモデルです。バッファーはデータを示すだけであり、データがメモリーにどのように格納および移動されるかはランタイムの役割です。

この章では、データ管理の代替アプローチを紹介します。バッファーと USM の選択は、多くの場合、好みと既存のコードのスタイルによって決まり、アプリケーションでは、アプリケーション内のさまざまなデータを表現するため 2 つのスタイルを自由に組み合わせることができます。

USM は、メモリーのさまざまな抽象化を公開するだけです。USM はポインターベースであり、バッファーはより高レベルの抽象化です。バッファーの抽象化レベルにより、バッファーに含まれるデータはアプリケーション内の任意のデバイスで使用できるようになります。この場合、そのデータを使用可能にするために必要なことは、すべてランタイムによって管理されます。USM のポインターベースのモデルは、リンクリスト、ツリーなどのポインターベースのデータ構造を使用するアプリケーションに適しています。すでにポインターを使用している既存のコードにバッファーを組み込むのは、難しい場合もあります。ただし、バッファーはシステム内のすべてのデバイスで動作することが保証されていますが、一部のデバイスでは USM の特定のモード（またはどのモードも）をサポートしない場合があります。それでは、バッファーについて詳しく見ていきましょう。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_7

バッファーがどのように作成され、使用されるかを詳しく見ていきます。*バッファー*の説明では、*アクセサー*を無視することはできません。バッファーはプログラム内部のデータの表現と保存方法を抽象化しますが、バッファーを介してデータに直接アクセスすることはありません。代わりに、アクセスするデータをどのように利用するかランタイムに通知するアクセサー・オブジェクトを使用します。アクセサーは、タスクグラフ内の強力なデータ依存メカニズムに密接に結合されています。バッファーで実行可能なことを説明した後、プログラムでアクセサーを作成して使用する方法を説明します。

バッファー

バッファーはデータの高レベルの抽象化です。バッファーは必ずしも単一の場所、または仮想メモリーアドレスに関連付けられるわけではありません。実際に、ランタイムはバッファーが表現するメモリー内のさまざまな場所（異なるデバイス間でも）を自由に使用できますが、ランタイムは常にデータの一貫したビューを提供する必要があります。バッファーは、ホスト上および任意のデバイス上でアクセスできます。

```
template <typename T, int Dimensions, AllocatorT allocator>
class buffer;
```

図 7-1. バッファークラスの定義

`buffer` クラスは、図 7-1 に示す 3 つのテンプレート引数を持つテンプレート・クラスです。最初のテンプレートは、バッファーに格納されるオブジェクトのタイプです。このタイプは*デバイスでコピー可能*である必要があり、これは C++ で定義されているように*簡単にコピー可能*という概念を拡張します。簡単にコピー可能なタイプは、特別なコピー・コンストラクターや移動コンストラクターを使用せずにバイト単位でコピーしても安全です。デバイスのコピー可能なタイプは、この概念を `std::pair` や `std::tuple` など特定の C++ タイプに再帰的に拡張します。2 番目のテンプレート引数は、バッファーの次元を表す整数です。最後のテンプレート引数はオプションであり、通常はデフォルト値が使用されます。

この引数は、バッファーに必要なホスト上のメモリー割り当てに使用される C++ スタイルのアロケーター・クラスを指定します。まず、バッファー・オブジェクトを作成するさまざまな方法を見ていきましょう。

バッファーを作成

次の図では、バッファー・オブジェクトを作成するいくつかの方法を示します。例を順に見て、それぞれのインスタンスを確認します。

```
// デフォルトのアロケーターを使用して 2x5 int のバッファーを作成
buffer<int, 2, buffer_allocator<int>> b1{range<2>{2, 5}};

// range にデフォルト・アロケーターと CTAD を使用して、2x5 の int
// バッファーを作成
buffer<int, 2> b2{range{2, 5}};

// デフォルトで構築された std::allocator を使用して、
// 20 個の float バッファーを作成
buffer<float, 1, std::allocator<float>> b3{range{20}};

// 渡されたアロケーターを使用して、の 20 個の float バッファーを作成
std::allocator<float> myFloatAlloc;
buffer<float, 1, std::allocator<float>> b4{range(20),
                                           myFloatAlloc};
```

図 7-2. バッファーを作成 (1)

図 7-2 で作成する最初のバッファー b1 は、10 個の整数の 2 次元バッファーです。すべてのテンプレート引数を明示的に渡し、さらに buffer_allocator<T> のデフォルト値をアロケーター・タイプとして渡します。buffer_allocator もテンプレート型であるため、buffer_allocator<int> を指定して、バッファーと同様に明示的に特殊化する必要があります。ただし、最新の C++ を使用すると、これをさらにコンパクトに表現できます。バッファー b2 も、デフォルトのアロケーターを使用する 10 個の整数の 2 次元バッファーです。ここでは、C++17 のクラス・テンプレート引数推論 (CTAD) を利用して、テンプレート引数を自動推論します。

CTAD は「すべてか、まったくないか」のツールです。つまり、クラスのすべてのテンプレート引数を推論するか、まったく推論しないかのどちらかです。この場合、2 つの引数を取る範囲で b2 を初期化するという事実から、それが 2 次元の範囲であると推測します。アロケーター・テンプレート引数にはデフォルト値があるため、バッファーを作成するときに明示的に指定する必要はありません。

バッファー b3 を使用して、20 個の float のバッファーを作成し、デフォルトで構築された std::allocator を使用してホスト上で必要なメモリーを割り当てます。バッファーでカスタム・アロケーター・タイプを使用する場合、デフォルトで構築されたアロケーター・オブジェクトの代わりに、実際のアロケーター・オブジェクトをバッファーに渡して使用することがよくあります。バッファー b4 は、コンストラクターの呼び出しの範囲の後のアロケーター・オブジェクトを取得して、実行する方法を示しています。

この例の最初の 4 つのバッファーでは、バッファーに必要なメモリーを割り当て、作成時にそのデータを初期化しません。既に初期化されている可能性のある既存の C++ 割り当てを効果的にラップするため、バッファーを使用するのは一般的なパターンです。これを実現するには、初期値のソースをバッファー・コンストラクターに渡します。それにより、いくつかのことが可能になります。次の例でそれを確認します。

```
// 4 つの double のバッファーを作成し、
// ホストポインターから初期化します。
double myDoubles[4] = {1.1, 2.2, 3.3, 4.4};
buffer b5{myDoubles, range{4}};

// 5 つの double のバッファーを作成し、それをホストポインターから
// const double に初期化します。
const double  myConstDbls[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
buffer b6{myConstDbls, range{5}};

// int への共有ポインターからバッファーを作成します。
auto sharedPtr = std::make_shared<int>(42);
buffer b7{sharedPtr, range{1}};
```

図 7-3. バッファーを作成（2）

図 7-3 では、バッファー b5 は 4 つの double 型の 1 次元バッファーを作成します。バッファーのコンストラクターには、バッファーのサイズを指定するレンジと C 配列 myDoubles へのホストポインターを渡します。ここで、CTAD を最大限に活用して、バッファーのすべてのテンプレート引数を推測することができます。引き渡すホストポインターは double を指し、これによりバッファーのデータ型が示されます。次元数は 1 次元の範囲から自動的に推測されます。1 次元の範囲自体は 1 つの数でのみ作成されるため推測できます。最後に、デフォルトのアロケーターが使用されるため、それを指定する必要はありません。

ホストポインターを渡すと、いくつかの注意すべき影響が生じます。ホストメモリーへのポインターを渡すことで、バッファーの存続期間中にホストメモリーにアクセスしないことをランタイムに知らせることになります。これは SYCL 実装によって強制されるものではなく（強制することもできません）、この規約に違反しないことは私たちの責任です。バッファーが有効な間にこのメモリーにアクセスすべきでない理由の 1 つは、最適化の理由から、バッファーがバッファーの内容を表すためホスト上の別のメモリーを使用する可能性があるためです。そうすると、値はホストポインターからこの新しいメモリーにコピーされます。後続のカーネルがバッファーを変更すると、元のホストポインターは指定された同期ポイントまで更新された値を反映しません。データがホストポインターに書き戻されるタイミングについては、この章の後半で詳しく説明します。

バッファー b6 はバッファー b5 と非常に似ていますが、大きな違いが 1 つあります。ここでは、const double へのポインターを使用してバッファーを初期化しています。これは、ホストポインターを介してしか値を読み取ることができず、書き込みはできません。ただし、この例のバッファーの型は、デダクション・ガイドが const 性を考慮していないため、const double ではなく double のままです。つまり、バッファーはカーネルによって書き込まれるかもしれませんが、バッファーが破棄された後にホストを更新するには、別のメカニズムを使用しなければならないことを意味します（この章の後半で説明します）。

バッファーは、C++ 共有ポインター・オブジェクトを使用して初期化することもできます。アプリケーションがすでに共有ポインターを使用している場合、この初期化方法では参照が適切にカウントされ、メモリーが解放されないことが保証されるため便利です。バッファー b7 は単一の整数バッファーを作成し、共有ポインターを使用して初期化します。

```
// 入力イテレータから int のバッファーを作成します。
std::vector<int> myVec;
buffer b8{myVec.begin(), myVec.end()};
buffer b9{myVec};

// 2x5 int のバッファーと、5 int の重複しない
// サブバッファーを 2 つを作成します。
buffer<int, 2> b10{range{2, 5}};
buffer b11{b10, id{0, 0}, range{1, 5}};
buffer b12{b10, id{1, 0}, range{1, 5}};
```

図 7-4. バッファーを作成（3）

コンテナーは、std::array、std::vector、std::list、std::map など、最新の C++ アプリケーションでよく使用されます。コンテナーを使用して 1 次元バッファーを初期化するには、2 つの方法があります。最初の方法は、図 7-4 のバッファー b8 に示すように入力反復子を使用します。ホストポインターの代わりに、データの先頭を表す反復子と末尾を表す反復子の 2 つをバッファー・コンストラクターに渡します。バッファーのサイズは、開始反復子を終了反復子と等しくなるまでインクリメントすることによって返される要素の数として計算されます。これは、C++ InputIterator インターフェイスを実装するすべてのデータ型に役立ちます。バッファーの初期値を提供するコンテナー・オブジェクトも連続していれば、さらに簡単にバッファーを作成できます。バッファー b9 は、コンストラクターにベクトルを渡すだけでベクトルからバッファーを作成できます。バッファーのサイズは初期化に使用されたコンテナーのサイズであり、バッファーのデータ型はコンテナーのデータ型です。このアプローチでバッファーを作成するのは一般的であり、std::vector や std::array などのコンテナーから推奨されます。

バッファー作成の最後の例は、バッファークラスの別の機能を示しています。別のバッファーからのバッファーのビューであるサブバッファーを作成することができます。サブバッファーには、親バッファーへの参照、ベース・インデックス、サブバッファーのレンジの 3 つの要素が必要です。サブバッファーはサブバッファーから作成することはできません。同じバッファーから複数のサブバッファーを作成することができ、それらは自由にオーバーラップできます。バッファー `b10` はバッファー `b2` と同じように作成され、行ごとに 5 つの整数を持つ 2 次元の整数バッファーです。次に、バッファー `b10` から 2 つのサブバッファー `b11` と `b12` を作成します。サブバッファー `b11` はインデックス `(0,0)` から始まり、最初の行のすべての要素を含みます。同様に、サブバッファー `b12` はインデックス `(1,0)` から始まり、2 行目にあるすべての要素を含みます。これにより、2 つの独立したサブバッファーが生成されます。サブバッファーは重複しないため、異なるカーネルが異なるサブバッファー上で同時に動作できますが、実行グラフと依存関係のスケジュール設定については次の章で詳しく説明します。

```
queue q;
int my_ints[42];

// 42 個の int バッファーを作成します
buffer<int> b{range(42)};

// 42 個の int バッファーを作成し、ホストポインターで初期化し、
// use_host_pointer プロパティーを追加します
buffer b1{my_ints,
          range(42),(42),
          {property::buffer::use_host_ptr{}}};

// 42 個の int バッファーを作成し、ホストポインターで初期化し、
// use_mutex プロパティーを追加します
std::mutex myMutex;
buffer b2{my_ints,
          range(42),(42),
          {property::buffer::use_mutex{myMutex}}};
// このバッファーで使用されるミューテックスへのポインターを取得します
auto mutexPtr =
    b2.get_property<property::buffer::use_mutex>()
        .get_mutex_ptr();
// スコープから出るまでミューテックスをロックします
std::lock_guard<std::mutex> guard{*mutexPtr};

// ホストポインターから初期化された 42 個の
// int のコンテキスト・バインド・バッファーを作成します
buffer b3{
    my_ints,
    range(42),
    {property::buffer::context_bound{q.get_context()}}};
```

図 7-5. バッファー・プロパティー

バッファー・プロパティー

バッファーは、その動作を変更する特別なプロパティーを使用して作成することもできます。図 7-5 では、3 つの異なるオプションのバッファー・プロパティーの例を示し、それらの使用方法について説明します。これらのプロパティーは大部分のコードでは稀であることに注意してください。

use_host_ptr

バッファー作成時にオプションで指定できる最初のプロパティーは use_host_ptr です。このプロパティーが存在すると、バッファーがホストメモリーに割り当てられないことを要求します。アロケーターが渡されると無視されます。代わりに、バッファーはコンストラクターに渡されるホストポインターが指すメモリーを使用します。ただし、デバイスがバッファーのデータを保持するため同じメモリーを使用する必要はないことに注意してください。デバイスは、接続されたメモリー内のバッファーの内容をキャッシュできます。このプロパティーは、ホストポインターがコンストラクターに渡される場合にのみ有効であることに注意してください。このオプションは、プログラムがすべてのホストメモリー割り当てを完全に制御する場合に役立ちます。たとえば、プログラマーがアプリケーションのメモリー・フットプリントを最小限に抑えることができます。

図 7-5 の例では、前の例と同じようにバッファー b を作成します。次に、バッファー b1 を作成し、myInts へのポインターで初期化します。また、プロパティー use_host_ptr も渡します。これは、バッファー b1 が myInts によって指されるメモリーのみを使用し、ホスト上に追加の一時ストレージを割り当てないことを意味します。

use_mutex

次のプロパティー use_mutex は、バッファーとホストコード間でのメモリーの細粒度の共有に関係します。このプロパティーを使用してバッファー b2 が作成されます。このプロパティーは、例の示すように、後でバッファーから照会できる mutex オブジェクトへの参照を取得します。このプロパティーでは、ホストポインターをコンストラクターに渡す必要があり、ランタイムは、受け取ったホストポインターを通じてホストコード内の更新された値に安全にアクセスできるか判断できます。ランタイムによってホストポインターがバッファーの最新の値を参照することが保証されるまで、ミューテックスをロックすることはできません。これは use_host_ptr プロパティーと組み合わせることもできますが、必須ではありません。use_mutex は、バッファーがまだ有効である間に、ホスト・アクセサー・メカニズム（後述）を使用せずに、ホストコードがバッファー内のデータにアクセスするメカニズムです。

一般に、ミューテックスを使用する特別な理由がない限り、ホスト・アクセサー・メカニズムが優先されます。これは、ミューテックスが正常にロックされ、データがホストコードで使用できるまでにどれくらいの時間を要するか保証されていないためです。

context_bound

最後のプロパティーは、この例のバッファー b3 の作成に示されています。ここでは、context_bound プロパティーを使用して 42 個の整数のバッファーが作成されます。このプロパティーはコンテキスト・オブジェクトへの参照を受け取ります。通常、バッファーはどのデバイスやコンテキストでも自由に使用できます。ただし、このプロパティーを使用すると、バッファーは指定されたコンテキストにロックされます。別のコンテキストでバッファーを使用すると、ランタイムエラーが発生します。

これは、たとえばカーネルが誤ったキューに送信される可能性があるケースを識別することで、プログラムをデバッグするのに役立ちます。 実際に、このプロパティーが多くのプログラムで使用されることは想定されていません。任意のコンテキストで任意のデバイス上のバッファーにアクセスする機能は、バッファー抽象化の最も強力なプロパティーの 1 つです（このプロパティーはこれを無効にします）。

バッファーで何ができるでしょう

バッファー・オブジェクトを使用しても、さまざまなことを行えます。バッファーの特性を照会したり、バッファーが破棄された後にデータがホストメモリーに書き戻されるか、または書き戻される場所を特定したり、バッファーを異なる特性を持つバッファーとして再解釈することができます。ただし、バッファーが表すデータに直接アクセスすることはできません。代わりに、データにアクセスするアクセサー・オブジェクトを作成します。これについては、この章の後半で詳しく説明します。

バッファーに関して照会可能な例としては、バッファーの範囲、バッファーが表すデータ要素の合計数、バッファー要素の格納に必要なバイト数などがあります。また、バッファーによって使用されるアロケーター・オブジェクトと、バッファーがサブバッファーであるかも照会できます。

バッファーが破棄されたときにホストメモリーを更新するのは、バッファーの使用において考慮すべき重要なことです。バッファーの作成方法に応じて、バッファー破棄後に計算結果でホストメモリーが更新される場合と更新されない場合があります。非定数データへのホストポインターからバッファーが作成および初期化された場合、バッファーが破棄されると、同じポインターが最新のデータで更新されます。ただし、バッファーがどのように作成されたかに関係なく、ホストメモリーを更新する方法もあります。`set_final_data` メソッドは、生ポインター、C++ の `OutputIterator`、または `std::weak_ptr` のいずれかを受け入れるバッファーのテンプレート・メソッドです。バッファーが破棄されると、バッファーに含まれるデータは指定されたホストの場所に書き込まれます。
バッファーが非定数データへのホストポインターから作成および初期化された場合、そのポインターを使用して `set_final_data` が呼び出された場合と同じになることに注意してください。技術的には、生ポインターは `OutputIterator` の特殊ケースです。`set_final_data` に渡されるパラメーターが `std::weak_ptr` である場合、ポインターの有効期限が切れていたり削除されていると、データはホストに書き戻されません。書き戻しが発生するかどうかも、`set_write_back` メソッドによって制御できます。

アクセサー

バッファーで表現されたデータは、バッファー・オブジェクトを介して直接アクセスすることはできません。代わりに、バッファーを安全アクセスできるアクセサー・オブジェクトを作成する必要があります。アクセサーはデータにアクセスする位置と方法をランタイムに伝えます。これにより、ランタイムは適切なデータを適切な位置とタイミングで配置することを保証します。これは非常に強力な概念であり、特に、部分的なデータの依存関係に基づいてカーネルの実行をスケジュールするタスクグラフと組み合わせると、その効果はさらに大きくなります。

アクセサー・オブジェクトは、テンプレート化されたアクセサークラスからインスタンス化されます。このクラスには 5 つのテンプレート・パラメーターがあります。最初のパラメーターは、アクセスされるデータ型です。これは、対応するバッファーに格納されているデータ型と同じである必要があります。

同様に、2 番目のパラメーターはデータとバッファーの次元を記述します。デフォルトの値は 1 になります。

モード	説明
`read`	読み取り専用アクセス
`write`	以前の内容を保持する書き込み専用アクセス
`read_write`	読み取りと書き込みアクセス

図 7-6. アクセスモード

次の 3 つのテンプレート・パラメーターはアクセサー固有です。最初は*アクセスモード*です。アクセスモードは、プログラム内でアクセサーをどのように使用するか示します。可能なモードを図 7-6 に示します。第 8 章では、これらのモードを使用してカーネルの実行順序を決定し、データを移動する方法について学びます。アクセス・モード・パラメーターには、何も指定されていない場合、または自動的に推測される場合、デフォルト値が設定されます。特に指定しない場合、アクセサーは非 `const` データ型の場合はデフォルトで `read_write` アクセスモードになり、`const` データ型の場合は `read` アクセスモードになります。これらのデフォルトは常に正当ですが、より正確な情報を提供することで、ランタイムの最適化が向上する可能性があります。アプリケーションの開発を始めるときは、アクセスモードを指定しない方が安全かつ容易です。その後、アプリケーションのパフォーマンスが重要な領域のプロファイル情報を基にアクセスモードを調整します。

ターゲット	説明
`device`	デバイスのグローバルメモリーを介してバッファーにアクセス
`host_task`	ホストタスクからバッファーにアクセス

図 7-7. ターゲットをアクセス

次のテンプレート・パラメーターは、*アクセスターゲット*です。バッファーはデータの抽象化であり、データがどこにどのように保存されるか記述するものではありません。アクセスターゲットには、データにアクセスする場所を記述します。考えられる 2 つのアクセスターゲットを 7-7 に示します。

SYCL で C++ を使用する場合、ターゲットは `device` と `host_task` の 2 つだけです。デフォルトのテンプレート値は `device` です。これは、デバイス上のバッファーのデータにアクセスすることを意味します。アクセサーはカーネルやデータ転送などデバイス上の操作で最もよく使用されるため、これは妥当です。もう 1 つのアクセスターゲットは `host_task` で、ホストタスクがバッファーのデータにアクセスする場合に使用されます。

デバイスには、使用できるメモリーの種類が異なる場合があります。特に、多くのデバイスには、ワークグループ内の複数のワーク項目間で共有される、ある種の高速ローカルメモリーがあります。SYCL の以前のバージョンでは、ローカルメモリーに対する特別なアクセスターゲットがありましたが、SYCL 2020 では別の方法で処理されます。第 9 章では、ワークグループ・ローカル・メモリーの使用方法を説明します。SYCL の以前のバージョンには、ホスト（SYCL 2020 の新機能であるホストタスクの外部）に対応する特別なアクセスターゲットもありました。これは、ホストコード内のバッファーのデータへのアクセスする新しい `host_accessor` クラスに置き換えられました。ただし、アクセスは、`host_accessor` の存続中は有効なままになります。`host_accessor` が有効な間はバッファーがホストにロックされるため、`host_accessor` オブジェクトのスコープを制限するように注意する必要があります。

最後のテンプレート・パラメーターは、アクセサーが *プレースホルダー・アクセサー*であるかどうかを制御します。これは、プログラマーが直接設定するパラメーターではなく、通常は、アクセサーの作成に使用されるコンストラクター呼び出しによって推測されます。プレースホルダー・アクセサーは、コマンドグループの外部で宣言され、カーネル内のデバイス上のデータアクセスに使用されます。アクセサー作成の例を見れば、プレースホルダー・アクセサーとそうでないアクセサーの違いがわかります。

アクセサーは get_access メソッドを使用してバッファー・オブジェクトから抽出できますが、直接作成（構築）する方が簡単です。これは非常に理解しやすく、コンパクトなスタイルであるため、以降の例で使用します。

アクセサーを作成

図 7-8 は、アクセサーを使用するのに必要なものがすべて含まれたサンプルプログラムです。この例では、A、B、C の 3 つのバッファーがあります。キューに送信する最初の並列タスクは、各バッファーへのアクセサーを作成し、これらのアクセサーを使用してバッファーを値で初期化するカーネルを定義します。各アクセサーは、アクセスするバッファーへの参照と、キューに送信するコマンドグループによって定義されたハンドラー・オブジェクトを使用して構築されます。これにより、コマンドグループの一部として送信するカーネルにアクセサーが効果的にバインドされます。 通常のアクセサーは、デフォルトでデバイスメモリーに格納されているグローバルバッファーをターゲットにするため、デバイスアクセサーです。これは最も一般的な使用例です。

```
#include <cassert>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;
  // 42 個の int のバッファーを 3 つ作成します
  buffer<int> a_buf{range{N}};
  buffer<int> b_buf{range{N}};
  buffer<int> c_buf{range{N}};
  accessor pc{c_buf};

  q.submit([&](handler &h) {
    accessor a{a_buf, h};
    accessor b{b_buf, h};
    accessor c{c_buf, h};
    h.parallel_for(N, [=](id<1> i) {
      a[i] = 1;
      b[i] = 40;
      c[i] = 0;
    });
  });
  q.submit([&](handler &h) {
    accessor a{a_buf, h};
    accessor b{b_buf, h};
    accessor c{c_buf, h};
    h.parallel_for(N,
                   [=](id<1> i) { c[i] += a[i] + b[i]; });
  });
  q.submit([&](handler &h) {
    h.require(pc);
    h.parallel_for(N, [=](id<1> i) { pc[i]++; });
  });

  host_accessor result{c_buf};
  for (int i = 0; i < N; i++) {
    assert(result[i] == N);
  }
  return 0;
}
```

図 7-8. 簡単なアクセサーの作成

2 番目に送信するタスクでは、バッファーへの 3 つのアクセサーも定義します。次に、2 番目のカーネルでこれらのアクセサーを使用して、バッファー A と B の要素をバッファー C に追加します。この 2 番目のタスクは最初のタスクと同じデータを操作するため、ランタイムは最初のタスクが完了した後にこのタスクを実行します。これについては次の章でさらに詳しく説明します。

3 番目のタスクでは、プレースホルダー・アクセサーを使用する方法を示します。アクセサー pC はバッファーを作成した後、図 7-8 の例の先頭で宣言されます。渡すハンドラー・オブジェクトがないため、コンストラクターにはハンドラー・オブジェクトが渡されないことに注意してください。これにより、再利用可能なアクセサー・オブジェクトを事前に作成できます。ただし、このアクセサーをカーネル内で使用するには、送信時にコマンドグループにバインドする必要があります。これには、ハンドラー・オブジェクトの require メソッドを使用します。プレースホルダー・アクセサーをコマンドグループにバインドしたら、他のアクセサーと同様にカーネル内で使用できるようになります。

最後に、計算結果をホスト上で読み戻すために、host_accessor オブジェクトを作成します。これはカーネル内で使用したタイプとは異なることに注意してください。この例のホストアクセサーの結果も、渡すハンドラー・オブジェクトがないため、ハンドラー・オブジェクトを受け取らないことに注意してください。
ホストアクセサーの特殊な型を使用すると、ホストアクセサーとプレースホルダーを区別できます。ホストアクセサーの重要な側面は、データがホスト上で使用可能になったときにのみコンストラクターが完了することです。ホストアクセサーの構築には長い時間がかかる可能性があります。

コンストラクターは、コピーするデータを生成するカーネルの実行完了と、コピー自体の完了を待機する必要があります。ホストアクセサーの構築が完了すると、ホスト上で直接アクセスするデータを安全に使用できるようになり、ホスト上で最新バージョンのデータが利用であることが保証されます。

この例は完全に正しいですが、アクセサーを作成するときにアクセサーをどのように使用するかは何も示していません。代わりに、バッファー内の非 const int データにはデフォルトのアクセスモード（read_write）を使用します。これは潜在的に過度に保守的であり、操作間の不必要な依存関係や余分なデータ移動が生じる可能性があります。

作成したアクセサーをどのように使用するかを示す情報が多ければ、ランタイムはさらによい仕事ができるかもしれません。ただし、これを実行する例を見る前に、デダクション・タグについて説明する必要があります。

デダクション・タグは、アクセサーのアクセスモードとターゲットの必要な組み合わせを表現するコンパクトな方法です。デダクション・タグが使用される場合、アクセサーのコンストラクターにパラメーターとして渡されます。
使用可能なタグを図 7-9 に示します。アクセサーがタグ・パラメーターを使用して構築されると、C++ CTAD は必要なアクセス モードとターゲットを適切に推測できるため、それらのテンプレート・パラメーターの既定値を簡単にオーバーライドできます。 必要なテンプレート・パラメーターを手動で指定することもできますが、タグを使用すると、完全にテンプレート化されたアクセサーを記述しなくても、同じ結果を得るさらにシンプルでコンパクトな方法が提供されます。

タグ値	`access_mode::`	`target::`
`read_only`	`read`	`device`
`read_write`	`read_write`	`device`
`write_only`	`write`	`device`
`read_only_host_task`	`read`	`host_task`
`read_write_host_task`	`read_write`	`host_task`
`write_only_host_task`	`write`	`host_task`

図 7-9. デダクション・タグ

前の例を書き直して、デダクション・タグを追加してみましょう。改良された例を図 7-10 に示します。

```
#include <cassert>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
  queue q;

  // 42 個の int のバッファーを 3 つ作成します
  buffer<int> buf_a{range{N}};
  buffer<int> buf_b{range{N}};
  buffer<int> buf_c{range{N}};

  accessor pc{buf_c};

  q.submit([&](handler &h) {
    accessor a{buf_a, h, write_only, no_init};
    accessor b{buf_b, h, write_only, no_init};
    accessor c{buf_c, h, write_only, no_init};
    h.parallel_for(N, [=](id<1> i) {
      a[i] = 1;
      b[i] = 40;
      c[i] = 0;
    });
  });
  q.submit([&](handler &h) {
    accessor a{buf_a, h, read_only};
    accessor b{buf_b, h, read_only};
    accessor c{buf_c, h, read_write};
    h.parallel_for(N,
                   [=](id<1> i) { c[i] += a[i] + b[i]; });
  });
  q.submit([&](handler &h) {
    h.require(pc);
    h.parallel_for(N, [=](id<1> i) { pc[i]++; });
  });

  host_accessor result{buf_c, read_only};

  for (int i = 0; i < N; i++) {
    assert(result[i] == N);
  }
  return 0;
}
```

図 7-10. 特定の用途を持つアクセサーの作成

まず、図 7-8 のようにバッファーを宣言します。後で使用するプレースホルダー・アクセサーも作成します。では、キューに送信する最初のタスクを考えてみましょう。以前は、バッファーへの参照とコマンドグループのハンドラー・オブジェクトを渡してアクセサーを作成しました。ここでは、コンストラクター呼び出しに 2 つのパラメーターを追加します。最初の新しいパラメーターはデダクション・タグです。このカーネルはバッファーの初期値を書き込むため、`write_only` タグを使用します。これにより、ランタイムは、このカーネルが新しいデータを生成しており、このバッファーから読み取らないことを認識します。

2 番目の新しいパラメーターはオプションのアクセサー・プロパティーであり、この章の前半で説明したバッファーのオプション・プロパティーに似ています。渡すプロパティー `no_init` により、ランタイムはバッファーの以前の内容を破棄可能であることを知ります。これは、ランタイムが不必要なデータ移動を排除できるため有用です。この例では、最初のタスクがバッファーに初期値を書き込んでいるため、カーネルが実行される前に、ランタイムは初期化されていないホストメモリーをデバイスにコピーする必要がありません。`no_init` プロパティーはこの例では有効ですが、`read-modify-write` 行うケースやバッファーの一部分だけが更新されるカーネルには適用すべきではありません。

キューに送信する 2 番目のタスクは前と同じですが、今度はアクセサーにデダクション・タグを追加します。ここで、アクセサー `aA` と `aB` に `read_only` タグを追加して、これらのアクセサーを通じてのみバッファー `A` と `B` の値を読み取ることをランタイムに知らせます。3 番目のアクセサー `aC` は、`A` と `B` の要素の合計を `C` に累積するため、`read_write` デダクション・タグを取得します。例では一貫性を保つためタグを明示的に使用していますが、デフォルトのアクセスモードは `read_write` であるため、これは省略できます。

プレースホルダー・アクセサーを使用する 3 番目のタスクでは、デフォルトの使用法が維持されます。これは、図 7-8 で示した例から変更されていません。最後のアクセサーであるホストアクセサーの結果は、作成時のデダクション・タグを受け取ります。ホスト上の結果値のみを読み取るため、コンストラクターに `read_only` タグを渡します。

ホストアクセサーが破棄されるようにプログラムを書き直すと、バッファー `C` を操作する別のカーネルを起動するときに、バッファー `C` をデバイスに書き戻す必要がなくなります。これは、`read_only` タグによってランタイムに、バッファー `C` がホストによって変更されないことが通知されるためです。

アクセサーで何ができるでしょう

アクセサー・オブジェクトを使用すると、さまざまなことを行えます。しかし、私たちが実行できる最も重要なことは、アクセサーの名前、つまりアクセスデータに示されています。これは通常、アクセサーの `[]` 演算子の 1 つを通じて行われます。図 7-8 と 7-10 の例では、`[]` 演算子を使用しています。この演算子は、多次元データを適切にインデックスできる `id` オブジェクトまたは単一の `size_t` のどちらかを受け取ります。2 番目のケースは、アクセサーに複数の次元がある場合に使用できます。その場合、スカラー値に到達するまで `[]` で再度インデックス付けされたオブジェクトが返されます。これは、2 次元の場合、`a[i][j]` の形式になります。アクセサー次元の順序は C++ の規則に従い、右端の次元がユニットストライド次元（“最も速く” 反復）となることに注意してください。

アクセサーは、基になるデータへのポインターを返すこともできます。このポインターには、通常の C++ 規則に従って直接アクセスできます。このポインターのアドレス空間に関しては複雑さが伴う可能性があることに注意してください。

アクセサー・オブジェクトから多くのことを照会できます。例えば、アクセサーを通じてアクセス可能な要素の数、アクセサーがカバーするバッファー領域のバイト単位のサイズ、アクセス可能なデータの範囲などがあります。

アクセサーは C++ コンテナーに似たインターフェイスを提供し、コンテナーを渡す多くの状況で使用できます。アクセサーによってサポートされるコンテナー・インターフェイスには、`get_pointer` と同等のデータメソッドと、各種前方反復子と後方反復子が含まれます。

まとめ

この章では、バッファーとアクセサーについて説明しました。バッファーは、メモリー管理基礎の詳細をプログラマーから隠匿するデータの抽象化です。これは、より単純で高レベルの抽象化を提供するために行われます。
バッファーを構築するさまざまな方法と、バッファー動作の変更に指定できるさまざまなオプションのプロパティーを示すいくつかの例を説明しました。ホストメモリーからのデータを使用してバッファーを初期化する方法と、バッファーを使用し終えたらデータをホストメモリーに書き戻す方法を学びました。

バッファーに直接アクセスすることはできないため、アクセサー・オブジェクトを使用してバッファー内のデータにアクセスする方法を学びました。デバイスアクセサーとホストアクセサーの違いを学びました。さまざまなアクセスモードとターゲットについて説明し、それらによって、アクセサーがプログラムによって、どこでどのように使用されるかランタイムに通知される仕組みについて説明しました。デフォルトのアクセスモードとターゲットを使用してアクセサーを使用する最も簡単な方法を示し、プレースホルダー・アクセサーとそうでないアクセサーを区別する方法を学びました。 次に、アクセサー宣言に推論タグを追加して、アクセサーの使用に関する詳細情報をランタイムに提供することで、サンプルプログラムをさらに最適化する方法を確認しました。最後に、プログラム内でアクセサーを使用するさまざまな方法を説明しました。

次の章では、ランタイムがアクセサーを通じて提供された情報を使用して、さまざまなカーネルの実行をスケジュールする方法を詳しく学びます。また、この情報によって、バッファー内のデータをホストとデバイス間でいつどのようにコピーするかをランタイムに通知する方法についても説明します。そして、バッファーやUSM 割り当てを含むデータの移動を明示的に制御する方法を学びます。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 8 章　カーネルのスケジュールとデータ移動

並列プログラムの設計者としての役割について説明する必要があります。並列プログラムの適切な編成は素晴らしいものです。すべてのデータが適切なタイミングで受信および送信するように調整され、データを待機することなくコードはフルスピードで実行されます。コードはハードウェアを最大限にビジー状態に保つように注意深く構築されます。それは夢のようです...

高速レーン（1 車線だけではありません！）での走行では、設計者として真剣に仕事に取り組む必要があります。それを実現するため、私たちは自身の仕事をタスクグラフの観点から考えることができます。

この章では、複雑なカーネルシーケンスを正確かつ効率的に実行するメカニズムであるタスクグラフについて説明します。アプリケーションでシーケンス処理が必要なものは、カーネルの実行とデータ移動の 2 つです。タスクグラフは、適切なシーケンスの実現に使用されるメカニズムです。

まず、第 3 章の依存関係を使用してタスクを順序付ける方法を簡単に確認します。次に、SYCL ランタイムがグラフを構築する方法を説明します。SYCL グラフの基本的な構成要素であるコマンドグループについて説明します。次に、一般的なパターンのグラフを作成するさまざまな方法を説明します。また、明示的および暗黙的なデータの移動がグラフでどのように表現されるか説明します。最後に、グラフをホストと同期する方法について説明します。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_8

グラフのスケジュールとは

第 3 章では、データ管理とデータを使用する順序について説明しました。また、SYCL のグラフの背後にある重要な抽象化である依存関係についても触れました。カーネル間の依存関係は、基本的にカーネルがアクセスするデータに基づいています。カーネルは、出力を計算する前に、正しいデータを読み取ったことを確認する必要があります。

正しい実行を保証するため。3 種類の重要なデータ依存性についても説明しました。最初の書き込み後の読み取り（RAW）は、1 つのタスクが別のタスクによって生成されたデータを読み取る場合に発生します。このタイプの依存関係は、2 つのカーネル間のデータの流れを表現します。2 番目のタイプの依存関係は、1 つのタスクがデータを読み取った後に別のタスクがそのデータを更新する場合に発生します。このタイプの依存関係を読み取り後の書き込み（WAR）依存関係と呼びます。最後のタイプのデータ依存性は、2 つのタスクが同じデータに書き込みを行うと発生します。これを書き込み後の書き込み（WAW）と呼びます。

データ依存関係は、グラフを構築するために使用する構成要素です。この依存関係のセットは、カーネルの単純な線形チェーンと、複雑な依存関係を持つ数百のカーネルを含む大規模で複雑なグラフの両方を表現に必要なものです。計算にどのようなタイプのグラフが必要であっても、SYCL グラフは、表現された依存関係に基づいてプログラムが正しく実行されることを保証します。ただし、グラフがプログラム内のすべての依存関係を正しく表現することを確認するのはプログラマーの責任です。

SYCL におけるグラフの仕組み

コマンドグループには、アクション、その依存関係、およびその他のホストコードという 3 つの異なる要素を含めることができます。これら 3 つのうち、常に必要なのはアクションです。アクションがないと、コマンドグループは何も実行しません。ほとんどのコマンドグループでは依存関係も表現されますが、そうでない場合もあります。そのような例には、プログラムで送信される最初のアクションがあります。

実行を開始するのに何かに依存することはありません。したがって、依存関係を指定する必要はありません。コマンドグループ内に出現する可能性があるもう 1 つは、ホスト上で実行される任意の C++ コードです。これは完全に合法であり、アクションやその依存関係を指定するのに役立ちます。このコードは、コマンドグループの作成中に実行されます（依存関係が満たされたことに基づいてアクションが実行された時）。

コマンドグループは通常、送信メソッドに渡される C++ ラムダ式として表現されます。コマンドグループは、カーネルとイベントベースの依存関係のセットを受け取るキュー・オブジェクトのショートカット・メソッドを通じて表現することもできます。

コマンドグループのアクション

コマンドグループによって実行できるアクションには、カーネルの実行と明示的なメモリー操作の 2 種類があります。コマンドグループは 1 つのアクションのみを実行できます。前の章で見たように、カーネルは `parallel_for` または `single_task` メソッドの呼び出しを通じて定義され、デバイス上で実行する計算を表現します。明示的なデータ移動の操作は、2 番目のタイプのアクションです。USM の例には、`memcpy`、`memset`、`fill` 操作が含まれます。バッファーの例には、`copy`、`fill`、`update_host` などがあります。

コマンドグループが依存関係を宣言する方法

コマンドグループのもう 1 つの主なコンポーネントは、グループによって定義されたアクションを実行する前に満たす必要がある依存関係のセットです。SYCL では、これらの依存関係をいくつかの方法で指定できます。

プログラムがインオーダー SYCL キューを使用する場合、キューのインオーダー・セマンティクスによって、連続してキューに投入されるコマンドグループ間の暗黙的な依存関係が指定されます。 送信されたタスクが完了するまで、次のタスクは実行できません。

イベントベースの依存関係は、コマンドグループを実行する前に完了する必要がある内容を指定するもう 1 つの方法です。これらのイベントベースの依存関係は、2 つのスタイルで指定できます。最初の方法は、コマンドグループがキューの送信メソッドに渡されるラムダ式として指定される場合に使用されます。この場合、プログラマーはコマンド・グループ・ハンドラー・オブジェクトの `depends_on` メソッドを呼び出し、イベントまたはイベントベクトルのいずれかをパラメーターとして渡します。もう 1 つの方法は、キュー・オブジェクトで定義されたショートカット・メソッドからコマンドグループが作成されるときに使用されます。
プログラマーがキュー上で `parallel_for` または `single_task` を直接呼び出す場合、イベントまたはイベントのベクトルが追加パラメーターとして渡される場合があります。

依存関係を指定する最後の方法は、アクセサー・オブジェクトを作成することです。アクセサーは、バッファー・オブジェクト内のデータの読み取り、または書き込みでどのように使用されるかを指定し、ランタイムがこの情報から異なるカーネル間に存在するデータの依存関係を判断できるようにします。
この章の冒頭で確認したように、データ依存性の例として、カーネルが別のカーネルが生成したデータを読み取ること、2 つのカーネルが同じデータに書き込むこと、またはカーネルが別のカーネルが読み取ったデータを変更するなどがあります。

例

ここで、これまで示したいくつかの例を使って説明します。2 つの異なる依存パターンをいくつかの方法で表現する方法を紹介します。ここで説明する 2 つのパターンは、1 つのタスクが別のタスクの後に実行される線形依存チェーンと、2 つの独立したタスクが連続するタスクの前に実行される “Y” パターンです。

これらの依存パターンのグラフを、図 8-1 と 8-2 に示します。図 8-1 は線形依存チェーンです。最初のノードはデータの初期化を表し、2 番目のノードはデータを 1 つの結果に蓄積するリダクション操作を表します。図 8-2 は、2 つの異なるデータを個別に初期化する “Y” パターンです。

データが初期化された後、加算カーネルが 2 つのベクトルを合計します。最後に、グラフの最終ノードが結果を 1 つの値に蓄積します。

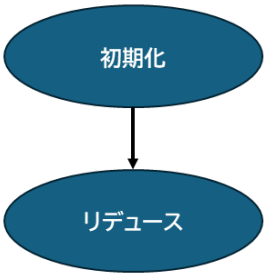

図 8-1. 線形依存チェーングラフ

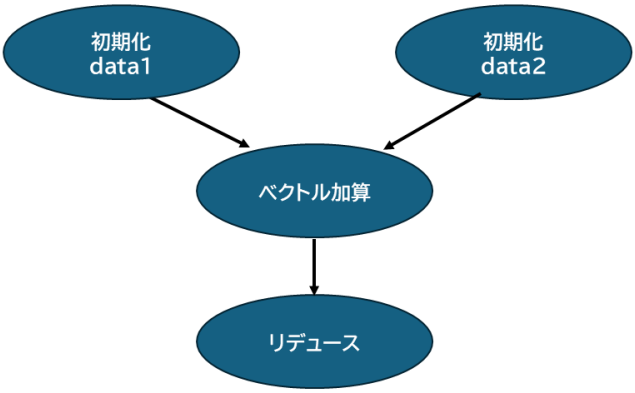

図 8-2. “Y” パターン依存グラフ

各パターンの 3 つの異なる実装を示します。最初の実装では、インオーダー・キューを使用します。2 番目は、イベントベースの依存関係を使用します。
最後の実装では、バッファーとアクセサーを使用して、コマンドグループ間のデータの依存関係を表現します。

図 8-3 は、インオーダー・キューを使用して線形依存チェーンを表現する方法を示しています。この例は非常に単純です。なぜなら、インオーダー・キューのセマンティクスによって、コマンドグループ間の実行順序がすでに保証されているからです。最初に送信するカーネルは、配列の要素を 1 に初期化します。

次のカーネルはそれらの要素を取得し、合計して最初の要素に生成します。キューはインオーダーであるため、最初のカーネルが完了するまで 2 番目のカーネルが実行を開始しないようにするのに、他に何もする必要はありません。最後に、キューがすべてのタスクの実行を完了するまで待機し、結果を検証します。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   queue q{property::queue::in_order()};

   int *data = malloc_shared<int>(N, q);

   q.parallel_for(N, [=](id<1> i) { data[i] = 1; });

   q.single_task([=]() {
     for (int i = 1; i < N; i++) data[0] += data[i];
   });
   q.wait();

   assert(data[0] == N);
   return 0;
}
```

図 8-3. インオーダー・キューによる線形依存チェーン

図 8-4 は、インオーダー・キューとイベントベースの依存関係を使用した同じ例を示しています。ここでは、parallel_for への最初の呼び出しから返されたイベントをキャプチャーします。2 番目のカーネルは、それをパラメーターとして depends_on に渡すことで、そのイベントへの依存関係とそれが表すカーネル実行を指定できます。図 8-6 では、カーネルを定義するショートカットの 1 つを使用して、2 番目のカーネルの式を短縮する方法を示します。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   queue q;

   int *data = malloc_shared<int>(N, q);

   auto e = q.parallel_for(N, [=](id<1> i) { data[i] = 1; });

   q.submit([&](handler  &h) {
     h.depends_on(e);
     h.single_task([=]() {
       for (int i = 1; i < N; i++) data[0] += data[i];
     });
   });
   q.wait();

   assert(data[0] == N);
   return 0;
}
```

図 8-4. イベントによる線形依存チェーン

図 8-5 は、USM ポインターの代わりにバッファーとアクセサーを使用して、線形依存チェーンの例を書き直したものです。ここでもインオーダー・キューを使用しますが、コマンドグループの実行順序付けには、イベントベースの依存関係ではなく、アクセサーを通じて指定されたデータ依存関係を使用します。
2 番目のカーネルは最初のカーネルで生成されたデータを読み取ります。同じバッファー・オブジェクトに基づいてアクセサーを宣言しているため、ランタイムはこれを認識できます。前の例とは異なり、キューがすべてのタスクの実行を完了するまで待機しません。代わりに、2 番目のカーネルの出力と、ホスト上で正しい答えを計算したアサーション間のデータ依存関係を定義するホストアクセサーを構築します。ホストアクセサーはホスト上の最新のデータを表示しますが、バッファー作成時に指定されると、元のホストメモリーが更新されていることは保証されないことに注意してください。

最初にバッファーを破棄するか、第 7 章で説明したミューテックス・メカニズムのような高度なメカニズムを使用しない限り、元のホストメモリーには安全にアクセスできません。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   queue q;

   buffer<int> data{range{N}};

   q.submit([&](handler &h) {
     accessor a{data, h};
     h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
   });

   q.submit([&](handler &h) {
     accessor a{data, h};
     h.single_task([=]() {
       for (int i = 1; i < N; i++) a[0] += a[i];
     });
   });

   host_accessor h_a{data};
   assert(h_a[0] == N);
   return 0;
}
```

図 8-5. バッファーとアクセサーによる線形依存チェーン

図 8-6 は、インオーダー・キューを使用して “Y” パターンを表現する方法を示しています。この例では、2 つの配列 `data1` と `data2` を宣言します。次に、配列をそれぞれ初期化する 2 つのカーネルを定義します。これらのカーネルは相互に依存していませんが、キューがインオーダーであるため、カーネルは順番に実行する必要があります。この例では、2 つのカーネルの順序を入れ替えてもまったく問題ないことに注目してください。2 番目のカーネルが実行された後、3 番目のカーネルは 2 番目の配列の要素を 1 番目の配列の要素に加算します。最後のカーネルは最初の配列の要素を合計して、線形依存チェーンの例で行ったのと同じ結果を計算します。

この合計カーネルは前のカーネルに依存しますが、この線形チェーンもインオーダー・キューによってキャプチャーされます。最後に、すべてのカーネルが完了するまで待機し、正常に計算されたことを確認します。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   queue q{property::queue::in_order()};

   int *data1 = malloc_shared<int>(N, q);
   int *data2 = malloc_shared<int>(N, q);

   q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

   q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

   q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });

   q.single_task([=]() {
     for (int i = 1; i < N; i++) data1[0] += data1[i];

     data1[0] /= 3;
   });
   q.wait();

   assert(data1[0] == N);
   return 0;
}
```

図 8-6. インオーダー・キューと “Y” パターン

図 8-7 は、インオーダー・キューではなく、アウトオブオーダー・キューを使用した “Y” パターンの例を示しています。キューの順序により依存関係が暗黙的ではなくなるため、イベントを使用してコマンドグループ間の依存関係を明示的に指定します。図 8-6 に示すように、最初に依存関係のない 2 つの独立したカーネルを定義します。これらのカーネルを 2 つのイベント `e1` と `e2` で表します。3 番目のカーネルを定義するときは、最初の 2 つのカーネルに依存することを指定する必要があります。

これは、実行する前にイベント e1 と e2 が完了することに依存する、と言う方法で示します。ただし、この例では、ハンドラーの depends_on メソッドの代わりに、ショートカット形式を使用してこれらの依存関係を指定します。
ここで、イベントを追加パラメーターとして parallel_for に渡します。一度に複数のイベントを渡す場合、イベントの std::vector を受け入れる形式を使用しますが、幸いなことに、最新の C++ では、式 {e1, e2} を適切なベクトルに自動的に変換するため、この処理が簡素化されます。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;


int main() {
  queue q;

  int *data1 = malloc_shared<int>(N, q);
  int *data2 = malloc_shared<int>(N, q);

  auto e1 =
      q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

  auto e2 =
      q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

  auto e3 = q.parallel_for(
      range{N}, {e1, e2},
      [=](id<1> i) { data1[i] += data2[i]; });

  q.single_task(e3, [=]() {
     for (int i = 1; i < N; i++) data1[0] += data1[i];

     data1[0] /= 3;
  });
  q.wait();

  assert(data1[0] == N);
  return 0;
}
```

図 8-7. イベントと “Y” パターン

最後の例（図 8-8 を参照）では、USM ポインターとイベントをバッファーとアクセサーに置き換えています。この例では、2 つの配列 `data1` と `data2` をバッファー・オブジェクトとして表します。アクセサーをコマンド・グループ・ハンドラーに関連付ける必要があるため、カーネルではカーネルを定義するショートカット・メソッドを使用していません。ここでも、3 番目のカーネルは、最初の 2 つのカーネルへの依存関係を取得する必要があります。ここでは、バッファーのアクセサーを宣言することによってこれを実現します。既にこれらのバッファーのアクセサーを宣言しているため、ランタイムはこれらのカーネルの実行を適切に順序付けることができます。さらに、アクセサー `b` を宣言する際に、ランタイムに追加情報を提供します。アクセスタグ “`read_only`” を追加することで、このデータを読み取るだけで、新しい値を生成しないことをランタイムに知らせます。線形依存チェーンのバッファーとアクセサーの例で見たように、最終的なカーネルは、3 番目のカーネルで生成された値を更新することによって自身を順序付けます。
計算の最終値を取得するには、最終カーネルの実行が完了するまで待機し、その後データをホストに戻して読み取り、正しい結果を計算したことをアサートするホストアクセサーを宣言します。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
   queue q;

   buffer<int> data1{range{N}};
   buffer<int> data2{range{N}};

   q.submit([&](handler  &h) {
     accessor a{data1, h};
     h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
   });

   q.submit([&](handler  &h) {
     accessor b{data2, h};
     h.parallel_for(N, [=](id<1> i) { b[i] = 2; });
   });

   q.submit([&](handler  &h) {
     accessor a{data1, h};
     accessor b{data2, h, read_only};
     h.parallel_for(N, [=](id<1> i) { a[i] += b[i]; });
   });

   q.submit([&](handler  &h) {
     accessor a{data1, h};
     h.single_task([=]() {
       for (int i = 1; i < N; i++) a[0] += a[i];

       a[0] /= 3;
     });
   });

   host_accessor h_a{data1};
   assert(h_a[0] == N);
   return 0;
}
```

図 8-8. アクセサーと “Y” パターン

コマンドグループの各部分はいつ実行されるか

タスクグラフは非同期であるため、コマンドグループが正確にいつ実行されるか疑問に思うのは当然です。ここまでで、カーネルは依存関係が満たされるとすぐに実行される可能性があることは明らかですが、コマンドグループのホスト部分では何が起こるでしょう？

　コマンドグループがキューに送信されると、ホスト上で直ちに実行されます（送信呼び出しが返される前）。コマンドグループのホスト部分は 1 回だけ実行されます。コマンドグループで定義されたカーネル、または明示的なデータ操作は、デバイス上で実行するためキューに登録されます。

データ移動

データの移動は、アプリケーションのパフォーマンスを理解するために不可欠な、SYCL のグラフのもう 1 つの重要な側面です。ただし、バッファーとアクセサー、または USM 共有割り当てを使用してプログラム内でデータの移動が暗黙的に行われる場合、見落とされてしまうことがあります。
次に、データの移動が SYCL でのグラフ実行にどのような影響するか、さまざまな方法を調べます。

明示的なデータ移動

明示的なデータ移動の利点は、それがグラフに*明示的*に現れるため、グラフの実行中に何が起こっているかプログラマーに明らかになることです。明示的なデータ操作を USM 用とバッファー用に分けます。

第 6 章で説明したように、USM を使用した明示的なデータ移動は、デバイス割り当てとホスト間でデータをコピーする必要があるときに発生します。これは、キュークラスとハンドラークラスの両方にある memcpy メソッドを使用して実行できます。アクションまたはコマンドグループを送信すると、他のコマンドグループを使用してコピーを順序付けるために使用できるイベントが返されます。

バッファーを使用した明示的なデータの移動は、コマンド・グループ・ハンドラー・オブジェクトの copy メソッドまたは update_host メソッドを呼び出すことによって行われます。copy メソッドを使用すると、ホストメモリーとデバイス上のアクセサー・オブジェクト間でデータを手動で交換できます。
これはさまざまな理由で可能です。簡単な例として、長時間実行される計算シーケンスのチェックポイントの作成が挙げられます。copy メソッドでは、デバイスから任意のホストメモリーに一方向にデータを書き込むことができます。これを、バッファーを使用して行う場合、ほとんどのケースでは (つまり、バッファーが use_host_ptr で作成されていない場合)、まずデータをホストにコピーし、次にバッファーのメモリーから目的のホストメモリーにコピーする必要があります。

update_host メソッドは、コピーの特殊形式です。ホストポインターの周囲にバッファーが作成された場合、このメソッドはアクセサーによって表されるデータを元のホストメモリーにコピーします。 これは、プログラムが特別な use_mutex プロパティーを使用して作成されたバッファーと、ホスデータを手動で同期する場合に役立ちます。ただし、このユースケースはほとんどのプログラムで発生する可能性は低いと思われます。

暗黙的なデータ移動

暗黙的なデータ移動は、SYCL のコマンドグループとタスクグラフに隠れた影響を及ぼす可能性があります。暗黙的なデータ移動では、SYCL ランタイムまたはハードウェアとソフトウェアの組み合わせによって、ホストとデバイス間でデータがコピーされます。どちらの場合も、ユーザーからの明示的な介入なしにコピーが行われます。もう一度、USM ケースとバッファーケースを個別に見てみましょう。

USM では、ホストと共有の割り当てによって暗黙的なデータ移動が行われます。第 6 章で説明したように、ホスト割り当ては実際にデータを移動するのではなく、リモートでデータにアクセスし、共有割り当てはホストとデバイス間で転送する場合があります。この転送は自動的に行われるため、USM の暗黙的なデータ移動とコマンドグループを考慮する必要はありません。ただし、共有割り当てに関しては留意しておくべき微妙な点がいくつかあります。

プリフェッチ操作は `memcpy` と同様の方法で動作し、カーネルが共有割り当てを使用する前にランタイムが共有割り当ての転送を開始できるようにします。ただし、正しい結果を確実にするデータをコピーする必要がある `memcpy` とは異なり、プリフェッチはパフォーマンスを向上させるランタイムへの*ヒント*として扱われることが多く、プリフェッチではメモリー内のポインター値が無効になることはありません（新しいアドレス範囲にコピーするように）。カーネルの実行開始前にプリフェッチが完了していなくてもプログラムは正常に実行されますが、プリフェッチ操作は機能要件ではないため、多くのコードではグラフ内のコマンドグループがプリフェッチ操作に依存しないように選択することがあります。

バッファーにも多少のニュアンスがあります。バッファーを使用する場合、コマンドグループは、データの使用方法を指定するバッファーのアクセサーを構築する必要があります。これらのデータ依存関係は、異なるコマンドグループ間の順序を表現したタスクグラフの構築を可能にします。ただし、バッファーを持つコマンドグループは、データの移動に関する要件を指定する別の目的を果たす場合もあります。

アクセサーは、カーネルがバッファーを読み取るか、バッファーに書き込むかを指定します。このことから必然的に、データはデバイス上でも利用可能である必要があり、そうでない場合は、カーネルが実行を開始する前にランタイムがデータをデバイスに移動する必要があります。したがって、SYCL ランタイムは、データ移動操作をスケジュールできるように、現在のバッファーが存在する場所を追跡する必要があります。アクセサーを作成することで、グラフ内に追加の隠しノードが作成されます。データの移動が必要な場合は、ランタイムが最初にそれを実行する必要があります。それにより、初めて、送信されたカーネルが実行できるようになります。

図 8-8 を再度確認してみましょう。この例では、最初の 2 つのカーネルでは、バッファー data1 と data2 をデバイスにコピーする必要があります。ランタイムは、データを移動するため追加のグラフノードを暗黙的に作成します。3 番目のカーネルのコマンドグループが送信されると、これらのバッファーはまだデバイス上にある可能性が高いため、ランタイムは追加のデータ移動を行う必要がありません。4 番目のカーネルのデータも、追加のデータ移動を必要としない可能性がありますが、ホストアクセサーを作成するには、アクセサーが使用可能になる前に、ランタイムがバッファー data1 をホストに戻す移動をスケジュールする必要があります。

ホストとの同期

最後に説明するトピックは、グラフの実行をホストと同期する方法です。この章ではこの点について触れていますが、ここではプログラムがこれを実行する方法をすべて検討します。

ホスト同期の最初の方法は、これまでの例で使用してきた方法、つまりキューで待機する方法です。キュー・オブジェクトには、キューに送信されたすべてのコマンドグループが完了するまでホストの実行をブロックする wait と wait_and_throw の 2 つのメソッドがあります。これは、一般的なケースを処理する非常に単純な方法です。
ただし、この方法は粗粒度であることは指摘しておきます。きめ細かい同期が必要な場合（たとえば、パフォーマンスを向上させるため）、ここで説明する他のアプローチがアプリケーションのニーズにより適している可能性があります。

ホスト同期の次の方法は、イベントで同期することです。これにより、アプリケーションは特定のアクションまたはコマンドグループのみで同期できるため、キューの同期よりも柔軟性が高まります。これは、イベントで wait メソッドを呼び出すか、イベントのベクトルを受け入れるイベントクラスで静的メソッド wait を呼び出すことによって行われます。

図 8-5 と 8-8 では、次のメソッドであるホストアクセサーが使用されています。ホストアクセサーには 2 つの機能があります: 1 つは名前が示すように、ホスト上でデータにアクセスできるようにします。2 つめは、現在アクセスしているグラフとホスト間に新しい依存関係を定義して、ホストとデバイスを同期します。これにより、ホストに書き戻されるデータが、グラフを実行した計算による正しい値であることが保証されます。ただし、バッファーが既存のホストメモリーから構築された場合、この元のメモリーに更新された値が含まれるとは限らないことに注意してください。

ホストアクセサーがブロックされていることにも注意してください。データが利用可能になるまで、ホストアクセサーの作成以降は、ホストの実行を続行できません。同様に、ホストアクセサーが存在し、そのデータを使用可能な状態に維持している間は、デバイス上でバッファーを使用することはできません。一般的なパターンは、ホストアクセサーが不要になったらデータを解放するため、追加の C++ スコープ内にホストアクセサーを作成することです。これはホスト同期のもう一つの方法の例です。

SYCL の特定のオブジェクトは、破棄されると特殊な動作を行い、それらのデストラクターが呼び出されます。ここでは、ホストアクセサーが、データが破棄されるまでホスト上にデータを残す仕組みについて学習しました。また、バッファーやイメージは破棄されたり、スコープから外れると特殊な動作をします。バッファーが破棄されると、そのバッファーを使用するすべてのコマンドグループの実行完了を待機します。カーネルやメモリー操作でバッファーが使用されなくなると、ランタイムはデータをホストに書き戻すことがあります。このようなコピーが行われるのは、バッファーがホストポインターで初期化されている場合と、`set_final_data` メソッドにホストポインターが渡されている場合です。その後ランタイムは、そのバッファーのデータを書き戻して、オブジェクトが破棄される前にホストポインターを更新します。

ホストと同期する最後のオプションには、第 7 章で説明した珍しい機能があります。バッファー・オブジェクトのコンストラクターはオプションでプロパティー・リストを受け取ることを思い出してください。バッファーを作成するときに渡すことができる有効なプロパティーの 1 つは `use_mutex` です。この方法でバッファーを作成すると、バッファーが所有するメモリーをホスト・アプリケーションと共有できるという要件が追加されます。 このメモリーへのアクセスは、バッファーの初期化に使用されるミューテックスによって制御されます。バッファーと共有されているメモリーに安全にアクセスできる場合、ホストはミューテックスのロックを取得

できます。ロックを取得できない場合、ユーザーはメモリー移動操作をキューに登録して、データをホストと同期させる場合があります。この使用法は非常に特殊であり、ほとんどの DPC++ アプリケーションでは見られません。

まとめ

この章では、グラフについて、また SYCL でグラフがどのように構築、スケジュール、実行されるかについて学びました。コマンドグループとは何か、またどのような機能を持つか詳しく説明しました。コマンドグループ内に存在する可能性のある 3 つの要素（依存関係、アクション、およびその他のホストコード）について説明しました。イベントを使用するタスク間の依存関係と、アクセサーによって記述されたデータ依存関係を指定する方法を確認しました。コマンドグループ内の単一のアクションはカーネルまたは明示的なメモリー操作のいずれかであることを学び、次に、一般的な実行グラフパターンを構築する方法を示すいくつかの例を見てきました。
次に、データの移動が SYCL グラフの重要な部分である理由を確認し、それがグラフに明示的または暗黙的にどのように現れるかを学びました。最後に、グラフの実行をホストと同期するすべての方法を検討しました。

プログラムフローを理解することで、デバッグすべき実行時エラーがある場合に取得できるデバッグ情報を理解できるようになります。第 13 章の “ランタイム機能のデバッグ” セクションには、本書のここまでに得た知識をもう少し理解し易くした表があります。ただし、本書では、これらの高度なコンパイラー・ダンプについて詳しく説明しません。

この章を読んで、線形チェーンから、数百のノードと複雑なデータおよびタスクの依存関係を持つ巨大なグラフまで、複雑なグラフを構築できるグラフの専門家になったような気分になっていただければ幸いです。次の章では、特定のデバイス上でのアプリケーションのパフォーマンス向上に役立つ低レベルの詳細について詳しく説明していきます。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 9 章 通信と同期

第 4 章では、基本的なデータ並列カーネル、または明示的な ND-range カーネルを使用して並列性を表現する方法を説明し、基本的なデータ並列カーネルが、同じ操作をすべてのデータに独立して適用する仕組みについて説明しました。また、明示的な ND-range カーネルが実行範囲をワーク項目のワークグループに分割する方法についても触れました。

この章では、*Think Parallel（並列に考える）*への継続的な探求の中で、問題をどのように小さなサイズのチャンクに分割するかという問題に再び取り組みます。また、明示的な ND-range カーネルについて詳しく説明し、ワーク項目のグループ化により一部のタイプのアルゴリズムのパフォーマンスを向上させる方法について説明します。ワーク項目のグループが、並列ワークの実行にどのように追加の保証を提供するか説明し、ワーク項目のグループ化をサポートする言語機能を紹介します。これらのアイデアや概念の多くは、第 15 章、第 16 章、および第 17 章で特定のデバイス向けにプログラムを最適化するときに重要となり、第 14 章で一般的な並列パターンを記述するときにも重要になります。

ワークグループとワーク項目

第 4 章で、明示的な ND-range カーネルがワーク項目をワークグループに編成し、同じワークグループ内のすべてのワーク項目には、追加のスケジュール保証があることを思い出してください。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_9

この特性は、ワークグループ内のワーク項目が協調して問題を解決できることを意味するため重要です。

図 9-1 は、ワークグループに分割された ND-range を示しています。各ワークグループは異なる色で表されます。各ワークグループ内のワーク項目は、同じ色を共有する他のワーク項目と安全に通信できます。

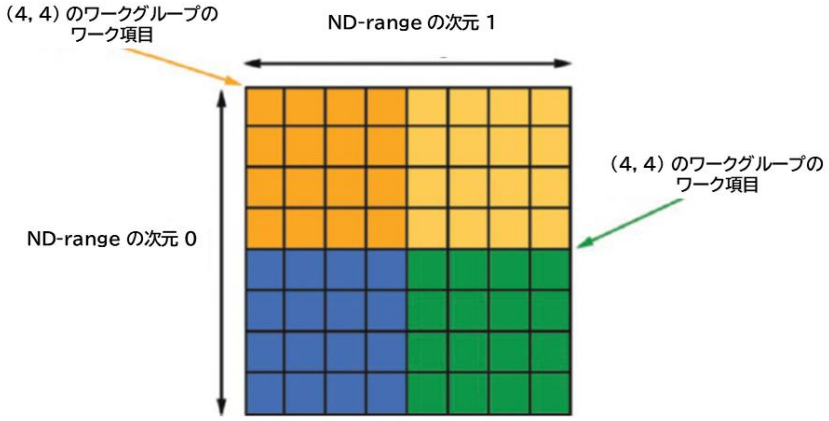

図 9-1. サイズ (8, 8) の 2 次元 ND-range をサイズ (4, 4) の 4 つのワークグループに分割

異なるワークグループ内のワーク項目が同時に実行される保証はないため、ある色のワーク項目は異なる色のワーク項目と通信することはできません。あるワーク項目が現在実行されていない別のワーク項目と通信しようとすると、カーネルがデッドロックする可能性があります。カーネルの実行を完了させるには、1 つのワーク項目が別のワーク項目と通信するときに、それらが同じワークグループ内にあることを確認する必要があります。

効率的な通信のための構成要素

このセクションでは、グループ内のワーク項目間の効率的な通信をサポートする構成要素について説明します。いくつかはカスタム・アルゴリズムの構築を可能にする基本的な構成要素ですが、その他はより高レベルで、多くのカーネルで使用される共通の操作を記述します。

同期とバリア

通信の最も基本的な構成要素は*バリア*関数です。バリア関数には主に 2 つの目的があります:

まず、バリア関数はグループ内のワーク項目の実行を同期します。実行を同期することにより、1 つのワーク項目は、その操作の結果を使用する前に、同じグループ内の別のワーク項目が操作を完了していることを確認できます。 あるいは、別のワーク項目が操作の結果を使用する前に、1 つのワーク項目にその操作を完了する時間が与えられます。

2 番目に、バリア関数は各ワーク項目がメモリー状態の参照を同期します。このタイプの同期操作は、*メモリーの一貫性*の強制またはメモリーの *フェンス*として知られています (詳細は第 19 章を参照)。 メモリーの一貫性は、バリアの前に実行されたメモリー操作の結果が、バリアの後のその他のワーク項目に参照可能となることを保証するため、実行の同期と同じくらい重要です。メモリーの一貫性がないと、1 つのワーク項目の操作は森の中で木が倒れるようなもので、その音が他のワーク項目に届く場合と届かない場合があります。

図 9-2 は、バリア関数で同期するグループ内の 4 つのワーク項目を示しています。各ワーク項目の実行時間は異なる場合がありますが、すべてのワーク項目がバリアを実行するまで、どのワーク項目もバリアを超えて実行することはできません。バリア関数を実行すると、すべてのワーク項目はメモリーを一貫して参照できます。

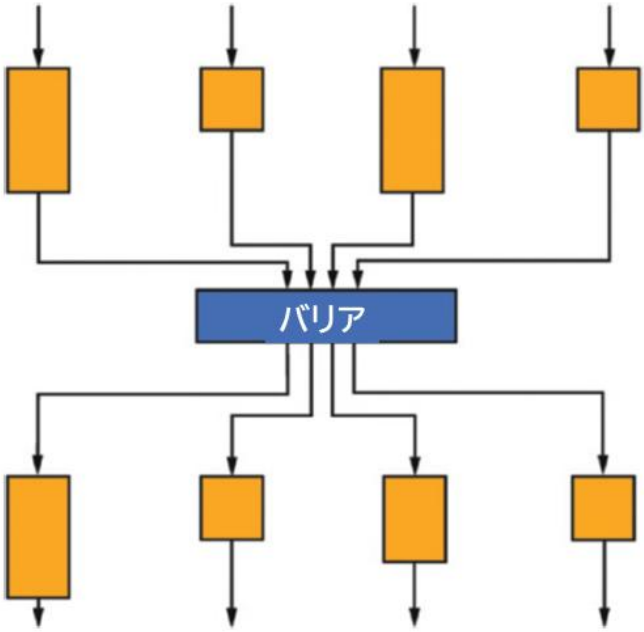

図 9-2. バリア関数で同期するグループ内の 4 つのワーク項目

メモリーがデフォルトで一貫していないのはなぜか

多くのプログラマーにとって、メモリーの一貫性という考え方、そして異なるワーク項目がメモリーに対して異なるビューを持つという考え方は、非常に奇妙に感じられるかもしれません。すべてのメモリーがデフォルトですべてのワーク項目に対して一貫性を持ったらさらに簡単ではないでしょうか？ 簡単に言えばそうですが、実装には非常にコストがかかります。ワーク項目にメモリーの不整合なビューを許可し、プログラム実行中の定義された位置でのみメモリーの整合性を要求することで、アクセラレーター・ハードウェアはより安価になり、パフォーマンスが向上するか、またはその両方になる可能性があります。

バリア関数は実行を同期するため、グループ内のすべてのワーク項目がバリアを実行するか、グループ内のどのワーク項目もバリアを実行しないかが重要です。グループ内の一部のワーク項目がバリア関数の近辺で分岐する場合、グループ内の他のワーク項目はバリアで永久に待機する可能性があります（少なくともユーザーが諦めてプログラムを終了するまで）。

集合関数

ある関数がグループ内のすべてのワーク項目で実行される場合、その操作はグループ内の個々のワーク項目ではなくグループによって実行されるため、集合関数と呼ばれることがあります。バリア関数は SYCL で使用できる唯一の集合関数ではありません。他の集合関数についてはこの章の後半で説明します。

ワークグループ・ローカル・メモリー

ワークグループのバリア関数は、ワークグループ内のワーク項目間の通信を調整するのに十分ですが、通信自体はメモリーを介して行われます。通信は USM またはバッファーを介して行われる場合もありますが、これは不便で非効率です。通信には専用の割り当てが必要であり、その割り当てをワークグループ間で分割する必要があります。

カーネル開発を簡素化し、ワークグループ内のワーク項目間の通信を高速化するため、SYCL はワークグループ内のワーク項目間の通信専用の特別な*ローカルメモリー*空間を定義します。

図 9-3 には、2 つのワークグループが示されています。両方のワークグループは、*グローバルメモリー*空間内の USM とバッファーにアクセスできます。各ワークグループは自身の*ローカルメモリー*空間内の変数にアクセスできますが、別のワークグループのローカルメモリー内の変数にはアクセスできません。

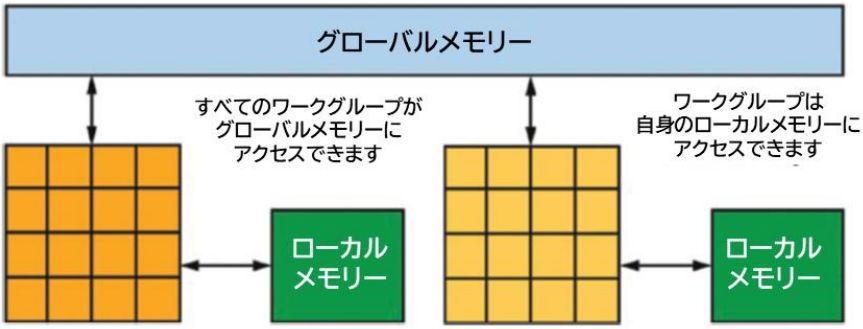

図 9-3. 各ワークグループはすべてのグローバルメモリーと自身のローカルメモリーにアクセス可能

ワークグループが開始すると、そのローカルメモリーの内容は初期化されておらず、ワークグループの実行が終了した後は、ローカルメモリーは保持されません。これらの特性により、ローカルメモリーは、ワークグループの実行中に一時的なストレージとしてのみ使用します。

CPU デバイスなど一部のデバイスでは、ローカルメモリーはソフトウェアによる抽象化であり、グローバルメモリーと同じメモリー・サブシステムを使用して実装されます。これらのデバイスでは、ローカルメモリーは主に通信の利便性を高めるためのメカニズムで使用されます。一部のコンパイラーは、コンパイラーの最適化にメモリー空間情報を使用することがありますが、それ以外は、通信にローカルメモリーを使用すると、これらのデバイス上のグローバルメモリーを介した通信よりもパフォーマンスは低下します。

多くの GPU などのデバイスでは、ローカルメモリー専用のリソースがあります。これらのデバイスでは、ローカルメモリーを介した通信の方が、グローバルメモリー通信よりもパフォーマンスが向上します。

ローカルメモリーを使用すると、ワークグループ内のワーク項目間の通信がより便利かつ高速になります。

デバイスクエリー `info::device::local_mem_type` を使用すると、アクセラレーターにローカルメモリー専用のリソースがあるかどうか、またはローカルメモリーがグローバルメモリーのソフトウェア抽象化として実装されているか判断できます。デバイスのプロパティー照会の詳細については第 12 章を参照してください。CPU、GPU、FPGA のローカルメモリーの一般的な実装方法の詳細は第 15 章、第 16 章、および第 17 章を参照してください。

ワークグループ・バリアとローカルメモリーの使用

ワーク項目間の効率的な通信の基本的な構成要素を特定したので、カーネル内でワークグループ・バリアとローカルメモリーを表現する方法を説明できます。ワーク項目間の通信にはワーク項目のグループ化の概念が必要であるため、これらの概念は Nd-range カーネルに対してのみ表現でき、基本的なデータ並列カーネルの実行モデルには含まれていないことに注意してください。

この章では、第 4 章で紹介した単純な行列乗算カーネルの例を基に、行列乗算を実行するワークグループ内のワーク項目間の通信を導入します。多くのデバイス（必ずしもすべてではありません）では、ローカルメモリーを介して通信すると、行列乗算カーネルのパフォーマンスが向上します。

行列乗算に関する注意事項

本書では、行列乗算カーネルを使用して、カーネルの変更がパフォーマンスにどのように影響するか示します。この章で説明する手法を使用すると、多くのデバイスで行列乗算のパフォーマンスが向上する可能性がありますが、行列乗算は非常に重要かつ一般的な操作であるため、多くのベンダーが高度に調整されたバージョンの行列乗算を実装しています。ベンダーは、特定のデバイスの機能の実装と検証に多くの時間と労力を費やしており、場合によっては、標準的な並列カーネルでは困難または不可能な機能や手法を使用することもあります。

ベンダー提供のライブラリーを使用

ベンダーが関数のライブラリー実装を提供する場合、その関数を並列カーネルとして再実装するよりも、ライブラリーを使用する方がほとんどの場合有利です。行列乗算については、SYCL プログラマーによる C++ に適したソリューションとして、インテルのツールキットの一部として oneMKL を参照できます。

図 9-4 は、第 4 章の行列乗算カーネルに似た、単純な行列乗算カーネルを示しています。 このカーネル、およびこの章のすべての行列乗算カーネルでは、`T` は、32 ビットの `float` や 64 ビットの `double` など、行列に格納されるデータ型を示すテンプレート型です。

```
h.parallel_for(range{M, N}, [=](id<2> id) {
   int m = id[0];
   int n = id[1];

   // テンプレート型 T は行列に格納される
   // データの型
   T sum = 0;
   for (int k = 0; k < K; k++) {
     sum += matrixA[m][k] * matrixB[k][n];
   }

   matrixC[m][n] = sum;
});
```

図 9-4. 第 4 章の純粋な行列乗算カーネル

第 4 章では、行列乗算アルゴリズムの再利用性が高く、ワーク項目をグループ化するとアクセスの局所性が高まることで、キャッシュのヒット率も向上する可能性があることを確認しました。この章では、パフォーマンスを向上させるため*暗黙的*にキャッシュの動作に頼るのではなく、修正された行列乗算カーネルは、アクセスの局所性を高めるためローカルメモリーを*明示的なキャッシュ*として使用します。

多くのアルゴリズムでは、ローカルメモリーを明示的なキャッシュとして考えると便利です。

図 9-5 は、第 4 章を修正した図で、単一行で構成されるワークグループを示しており、ローカルメモリーを使用するアルゴリズムを分かり易くしています。結果行列の行の要素については、結果の各要素が、青とオレンジで表示される入力行列のいずれかからの一意のデータ列を使用して計算されることに注意してください。この入力行列にはデータ共有がないため、ローカルメモリーは最適ではありません。ただし、行内のすべての結果要素が、薄緑色で示されている他の入力行列内の同じデータにアクセスしていることに注目してください。このデータは再利用されるため、ワークグループのローカルメモリーを活用するのに最適です。

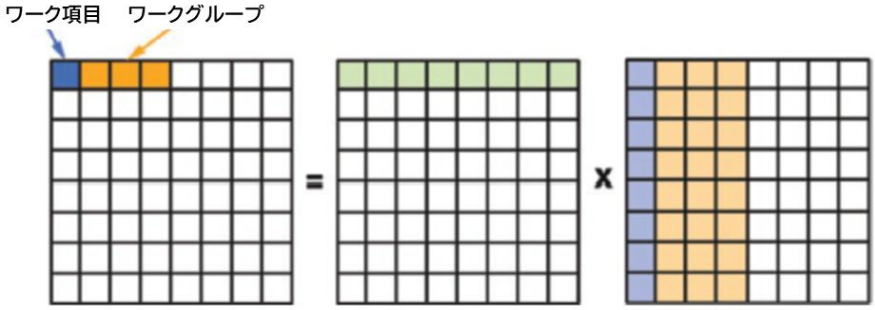

図 9-5. 行列乗算をワークグループとワーク項目にマッピング

潜在的に非常に大きな行列を乗算する、またワークグループのローカルメモリーが限られたリソースであるため、修正されたカーネルは各行列のサブセクション（行列*タイル*と呼ぶ）を処理します。各タイルについて、修正されたカーネルはタイルのデータをローカルメモリーにロードし、グループ内のワーク項目を同期してから、グローバルメモリーではなくローカルメモリーからデータをロードします。最初のタイルでアクセスされるデータを図 9-6 に示します。

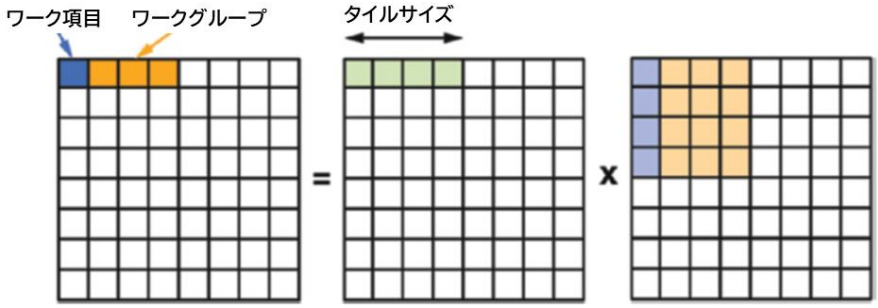

図 9-6. 最初のタイルの処理: 緑の入力データ（X の左側）は再利用され、ローカルメモリーから読み取られ、青とオレンジの入力データ（X の右側）はグローバルメモリーから読み取られます

私たちのカーネルでは、タイルのサイズがワークグループのサイズと同等になるようにしました。これは必須ではありませんが、ローカルメモリーへの転送やローカルメモリーからの転送が簡素化されるため、ワークグループ・サイズの倍数となるタイルのサイズを選択するのが、一般的かつ便利です。

ND-range カーネルのワークグループ・バリアとローカルメモリー

ここでは、ND-range カーネルでワークグループ・バリアとローカルメモリーがどのように表現されるか説明します。ND-range カーネルの表現は明示的です。カーネルは、ローカルアドレス空間内の割り当てを表すローカルアクセサーを宣言して操作し、ワークグループ内のワーク項目を同期するためバリア関数を呼び出します。

ローカルアクセサー

ND-range カーネルで使用するローカルメモリーを宣言するには、ローカルアクセサーを使用します。他のアクセサー・オブジェクトと同様に、ローカルアクセサーはコマンド・グループ・ハンドラー内で構築されますが、第 3 章および第 7 章で説明したアクセサー・オブジェクトとは異なり、ローカルアクセサーはバッファー・オブジェクトからは作成されません。代わりに、型と型の要素の数を記述する範囲を指定して、ローカルアクセサーが作成されます。他のアクセサーと同様に、ローカルアクセサーは 1 次元、2 次元、または 3 次元になります。図 9-7 は、ローカルアクセサーを宣言し、カーネルで使用する方法を示しています。

```
// 典型的なグローバルアクセサー
accessor dataAcc{dataBuf, h};

// 16 個の int からなる 1D ローカルアクセサー:
auto localIntAcc = local_accessor<int, 1>(16, h);

// 4 x 4 の float で構成される
// 2D ローカルアクセサー:
auto localFloatAcc =
    local_accessor<float, 2>({4, 4}, h);

h.parallel_for(
    nd_range<1>{{size}, {16}}, [=](nd_item<1> item) {
        auto index = item.get_global_id();
        auto local_index = item.get_local_id();

        // カーネル内では、他のアクセサーと同様に、
        // ローカルアクセサーの読み取りや書き込みが可能です。
        localIntAcc[local_index] = dataAcc[index] + 1;
        dataAcc[index] = localIntAcc[local_index];
    });
```

図 9-7. ローカルアクセサーの宣言と使用

各ワークグループが開始するとローカルメモリーは初期化されず、各ワークグループの完了後には保持されないことに注意してください。つまり、ローカルアクセサーは常に read_write である必要があります。そうでないと、カーネルはローカルメモリーの内容を割り当てたり、結果を参照することができなくなります。ただし、ローカルアクセサーはオプションでアトミックにすることもできます。その場合、アクセサーを介したローカルメモリーへのアクセスはアトミックに実行されます。アトミックアクセスについては、19 章で詳しく説明しています。

同期関数

ND-range カーネル・ワークグループ内のワーク項目を同期するには、ワークグループを示すグループを指定して group_barrier 関数を呼び出します。ワークグループを示すグループは nd_item からのみ照会でき、項目からは照会できないため、ワークグループ・バリアは ND-range カーネルでのみ使用でき、基本データ並列カーネルでは使用できません。

`group_barrier` 関数は、バリアによって実行されるメモリーの一貫性操作の*スコープ*を記述する追加のオプション引数を 1 つ受け入れます。`group_barrier` 関数に追加の引数が渡されない場合、バリア関数は渡されたグループに基づいてデフォルトのスコープを決定します。デフォルトのスコープは通常は正しいため、明示的なスコープが必要になることはほとんどありませんが、一部のアルゴリズムでは必要に応じてメモリースコープを拡張できます。

明示的なスコープはバリアによって実行されるメモリー操作にのみ影響し、バリアで実行を同期するワーク項目のセットは、バリアに渡されるグループ・オブジェクトによって決定されることに注意してください。異なるメモリースコープをバリアに渡すことで、より多くのワーク項目またはより少ないワーク項目を同期することはできませんが、異なるグループ・オブジェクトをバリアに渡すことで、異なるワーク項目のセットを同期できます。

完全な ND-Range カーネルの例

ローカル・メモリー・アクセサーを宣言し、バリア関数を使用してそのアクセスを同期する方法がわかったので、ワークグループ内のワーク項目間の通信を調整してグローバルメモリーへのトラフィックを削減する、ND-range カーネルの行列乗算を実装できます。完全な例を図 9-8 に示します。

```
// グローバルメモリー内の行列を表す
// 従来のアクセサー:
accessor matrixA{bufA, h};
accessor matrixB{bufB, h};
accessor matrixC{bufC, h};

// 1 つの行列スタイルのローカルアクセサー:
constexpr int tile_size = 16;

// テンプレート型 T は行列に格納されるデータの型
auto tileA = local_accessor<T, 1>(tile_size, h);

h.parallel_for(
   nd_range<2>{{M, N}, {1, tile_size}},
   [=](nd_item<2> item) {
     // グローバル・インデックス空間のインデックス:
     int m = item.get_global_id()[0];
     int n = item.get_global_id()[1];

     // ローカル・インデックス空間のインデックス:
     int  i = item.get_local_id()[1];

     T sum = 0;
    for (int kk = 0; kk < K; kk += tile_size) {
      // 同期して、すべてのワーク項目が
      // ローカルメモリー内の行列タイルの
      // 一貫したビューを持つようにします。
      tileA[i] = matrixA[m][kk + i];
      group_barrier(item.get_group());

      // ローカル・メモリー・タイルとグローバルメモリー内の
      // 行列 B を使用して計算を実行します。
     for (int k = 0; k < tile_size; k++) {
        sum += tileA[k] * matrixB[kk + k][n];
     }

      // 計算後、ローカル・メモリー・タイルからの
      // すべての読み取りが完了していることを
      // 確認するため再度同期します。
      group_barrier(item.get_group());
    }

    // 最終結果をグローバルメモリーに書き込みます。
    matrixC[m][n] = sum;
   });
```

図 9-8. ND-range の parallel_for とワークグループ・ローカル・メモリーを備えたタイル化された行列乗算カーネルを表現

このカーネルのメインループは、2 つの異なるフェーズとして考えることができます。最初のフェーズでは、ワークグループ内のワーク項目が連携して、共有データを A 行列からワークグループのローカルメモリーにロードします。2 番目のフェーズでは、ワーク項目が共有データを使用してそれぞれ計算を実行します。すべてのワーク項目が第 1 フェーズを完了してから第 2 フェーズに進むようにするため、ワークグループ内のすべてのワーク項目を同期し、メモリーフェンスを提供する `group_barrier` 呼び出しで 2 つのフェーズが分離されます。このパターンは一般的なものであり、カーネル内でワークグループ・ローカル・メモリーを使用する場合、ほとんどはワークグループ・バリアを使用する必要があります。

現在のタイルの計算フェーズと次の行列タイルの読み込みフェーズの間で実行を同期するには、`group_barrier` を呼び出す必要があることにも注意してください。この同期を行わないと、別のワーク項目が計算を完了する前に、現在の行列タイルの一部がワークグループ内の 1 つのワーク項目によって上書きされる可能性があります。一般に、あるワーク項目が別のワーク項目によって読み書きされたローカルメモリー内のデータを読み取ったり書き込んだりするときは常に同期が必要です。図 9-8 では、ループの最後に同期が行われていますが、代わりに各ループ反復の開始時に同期しても正しく動作します。

サブグループ

この章のここまでは、ワーク項目は、ワークグループのローカルメモリーを介してデータを交換し、ワークグループでは `group_barrier` 関数で同期することによって、ワークグループ内の他のワーク項目と通信してきました。

第 4 章では、ワーク項目の別のグループ化について説明しました。サブグループとは、同じハードウェア・リソース上で同時に実行されるか、追加のスケジュール保証により実行される、ワークグループ内のワーク項目の実装定義のサブセットです。

実装によってワーク項目をサブグループにグループ化する方法が決定されるため、サブグループ内のワーク項目は任意のワークグループ内のワーク項目よりも効率良く通信、または同期できる可能性があります。

ここでは、サブグループ内のワーク項目間の通信の構成要素について説明します。サブグループにはワーク項目のグループ化の概念も必要であるため、サブグループには Nd-range カーネルも必要であり、基本的なデータ並列カーネルの実行モデルには含まれません。

同期とサブグループ・バリア

ワークグループ内のワーク項目がワークグループ・バリアを使用して同期されるのと同様に、サブグループ内のワーク項目もサブグループ・バリアを使用して同期されます。サブグループ・バリアを実行するには、同じ `group_barrier` 関数を呼び出しますが、図 9-9 に示すように、ワークグループではなくサブグループを表すグループ・オブジェクトを渡します。ワークグループ・オブジェクトと同様に、サブグループを表すグループ・オブジェクトは、ND-range カーネルの `nd_item` クラスから照会できますが、基本データ並列項目からは照会できません。

```
h.parallel_for(
     nd_range{{size}, {16}}, [=](nd_item<1> item) {
       auto sg = item.get_sub_group();
       group_barrier(sg);
       // ...
       auto index = item.get_global_id();
       data_acc[index] = data_acc[index] + 1;
     });
```

図 9-9. クエリー と sub_group クラスの使用

また、ワークグループ・バリアと同様に、サブグループ・バリアは、サブグループ・バリアに関連付けられたメモリー操作の範囲を拡張するオプション引数を受け入れることがありますが、これは一般的ではなく、ほとんどの場合、デフォルトのメモリースコープをそのまま使用できます。

サブグループ内でのデータ交換

ワークグループとは異なり、サブグループにはデータを交換する専用メモリー領域がありません。代わりに、サブグループ内のワーク項目は、ワークグループのローカルメモリー、グローバルメモリー、またはより一般的にはサブグループの*集合関数*を使用してデータを交換できます。

前述のように、*集合関数*は、個々のワーク項目ではなく、ワーク項目のグループによって実行される操作を記述する関数です。バリア同期関数はワーク項目のグループによって実行される操作であるため、集合関数の一例です。

その他の集合関数は共通の通信パターンを表現します。この章の後半で、多くの集合関数のセマンティクスを詳しく説明しますが、ここでは、サブグループを使用した行列乗算の実装に使用する `group_broadcast` 集合関数に焦点を当てます。

`group_broadcast` 集合関数は、グループ内の 1 つのワーク項目から値を取得し、それをグループ内の他のすべてのワーク項目に伝達します。図 9-10 に例を示します。ブロードキャスト関数のセマンティクスでは、通信するグループ内の値を識別する `local_id` がグループ内のすべてのワーク項目で同じである必要があり、ブロードキャスト関数の結果もグループ内のすべてのワーク項目で同じになることが保証されることに注意してください。

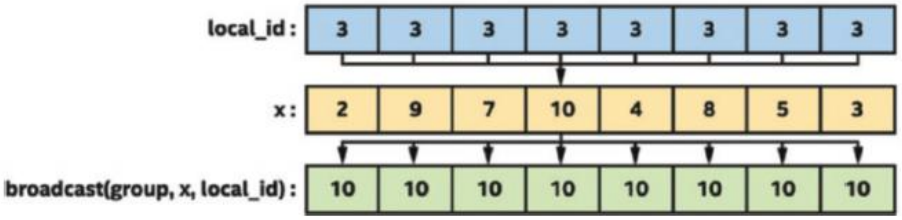

図 9-10. ブロードキャスト関数による処理

図 9-11 に示すローカルメモリー行列乗算カーネルの最も内側のループを見ると、グループ内の各ワーク項目が行列タイルから同じ値を読み取るため、行列タイルへのアクセスはブロードキャスト操作であることがわかります。

```
h.parallel_for(
    nd_range<2>{{M, N}, {1, tile_size}},
    [=](nd_item<2> item) {
      // グローバル・インデックス空間のインデックス:
      int m = item.get_global_id()[0];
      int n = item.get_global_id()[1];

      // ローカル・インデックス空間のインデックス:
      int i = item.get_local_id()[1];

      // テンプレート型 T は行列に格納される
      // データの型
      T sum = 0;
      for (int kk = 0; kk < K; kk += tile_size) {
         // 行列 A から行列タイルをロードし、
         // 同期して、すべてのワーク項目が
         // ローカルメモリー内の行列タイルの
         // 一貫したビューを持つようにします。
         tileA[i] = matrixA[m][kk + i];
         group_barrier(item.get_group());

         // ローカル・メモリー・タイルとグローバルメモリー内の
         // 行列 B を使用して計算を実行します。
         for (int k = 0; k < tile_size; k++) {
            // k の値はグループ内のすべてのワーク項目で
            // 同じであるため、tileA からのこれらの読み取りは
            // ブロードキャスト操作になります。
            sum += tileA[k] * matrixB[kk + k][n];
         }

         // 計算後、ローカル・メモリー・タイルからの
         // すべての読み取りが完了していることを
         // 確認するため再度同期します。
         group_barrier(item.get_group());
      }

      // 最終結果をグローバルメモリーに書き込みます。
      matrixC[m][n] = sum;
    });
```

図 9-11. 行列乗算カーネルにはブロードキャスト演算が含まれます

サブグループ・オブジェクトを持つ `group_broadcast` 関数を使用して、ワークグループのローカルメモリーやバリアが不要な行列乗算カーネルを実装します。多くのデバイスでは、ワークグループのローカルメモリーとバリアを使用したサブグループのブロードキャストは、ワークグループのブロードキャストよりも高速です。

完全なサブグループ ND-Range カーネルの例

図 9-12 は、サブグループを使用して行列乗算を実装する完全な例です。このカーネルは、ワークグループのローカルメモリーや明示的な同期を必要とせず、代わりにサブグループのブロードキャスト集合関数を使用して、サブグループ内のワーク項目間で行列タイルの内容を通信することに注意してください。

```
// 注：この例では、サブグループのサイズがタイルのサイズ以上
// であることを前提としています。
constexpr int tile_size = 4;
h.parallel_for(
    nd_range<2>{{M, N}, {1, tile_size}},
    [=](nd_item<2> item) {
       auto sg = item.get_sub_group();

       // グローバル・インデックス空間のインデックス：
       int m = item.get_global_id()[0];
       int n = item.get_global_id()[1];

       // ローカル・インデックス空間のインデックス：
       int i = item.get_local_id()[1];

       // テンプレート型 T は行列に格納される
       // データの型
       T sum = 0;
       for (int kk = 0; kk < K; kk += tile_size) {
          // 行列 A から行列タイルをロードし、
          T tileA = matrixA[m][kk + i];

          // 行列タイルからブロードキャストし、
          // グローバルメモリー内の行列 B から
          // ロードすることで計算を実行します。
          // ループ変数 k は、サブグループ内のどのワーク項目から
          // データをブロードキャストするかを表します。
          for (int k = 0; k < tile_size; k++) {
             sum += group_broadcast(sg, tileA, k) *
                    matrixB[kk + k][n];
          }
       }

      // 最終結果をグローバルメモリーに書き込みます。
      matrixC[m][n] = sum;
    });
```

図 9-12. ND-range の parallel_for とサブグループ集合関数で表現されたタイル行列乗算カーネル

グループ関数とグループ・アルゴリズム

この章のサブグループのセクションでは、集合関数と、集合関数が共通の通信パターンをどのように表現するか説明しました。特に、グループ内の 1 つのワーク項目からグループ内の他のワーク項目に値を伝達するブロードキャスト集合関数について説明しました。ここでは、そのほかの集合関数について説明します。

ここで説明する集合関数は、アトミック、ワークグループ・ローカル・メモリー、バリアなどの機能を使用してプログラムに直接実装できますが、多くのデバイスには集合関数を高速化する専用ハードウェアが搭載されています。デバイスに専用のハードウェア搭載されていない場合でも、ベンダーが提供する集合関数の実装では、デバイスに合わせて調整されている可能性が高いため、ビルトインの集合関数を呼び出すと、通常、自身で作成する汎用実装よりもパフォーマンスが向上します。

共通の通信パターンに集合関数を使用してコードを簡素化し、パフォーマンスを向上させます。

ブロードキャスト

`group_broadcast` 関数を使用すると、グループ内の 1 つのワーク項目が、グループ内の他のすべてのワーク項目と変数値を共有できるようになります。ブロードキャスト関数の動作を図 9-10 に示します。`group_broadcast` 関数は、ワークグループとサブグループの両方でサポートされています。

投票

any_of_group、all_of_group、none_of_group 関数（以降、投票関数と呼びます）を使用すると、ワーク項目はグループ全体でブール条件の結果を比較できます。グループ内の少なくとも 1 つのワーク項目に対して条件が true の場合、any_of_group は true を返し、グループ内のすべてのワーク項目に対して条件が true の場合、all_of_group は true を返し、グループ内のすべてのワーク項目に対して条件が false の場合、none_of_group は true を返します。入力例に対するこれら 2 つの関数の比較を図 9-13 に示します。

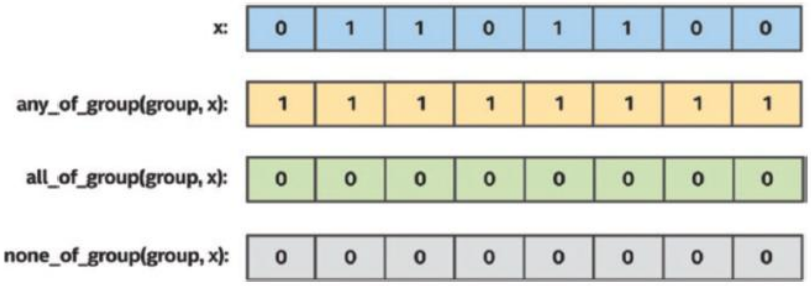

図 9-13. any_of_group、all_of_group、none_of_group 関数の比較

SYCL 2020 は、これらの関数のバリアントもサポートしており、グループ内のワーク項目が協調して、標準の C++ all_of、any_of、none_of アルゴリズムのようなデータの範囲を評価します。これらの関数は、グループ内の各ワーク項目が直接比較するデータを保持するバリアントと区別するため、joint_any_of、joint_all_of、および joint_none_of と名付けられています。

投票関数は、たとえば、グループ内のすべてのワーク項目のソリューションがいつ収束したか判断する一部の反復アルゴリズムに役立ちます。投票機能は、ワークグループとサブグループに対してサポートされています。

シャッフル

サブグループで最も便利な機能の 1 つは、明示的なメモリー操作なしで個々のワーク項目間で直接通信できることです。サブグループ行列乗算カーネルなど多くの場合、これらのシャッフル操作により、カーネルからワークグループのローカルメモリーの使用量を削減し、グローバルメモリーへの不要な繰り返しアクセスを回避できます。利用可能なシャッフル関数にはいくつかの種類があります。

最も一般的なシャッフル関数は `select_from_group` と呼ばれ、図 9-14 に示すように、サブグループ内の任意のワーク項目のペア間での任意の通信を可能にします。ただし、この汎用性にはパフォーマンス・コストが生じる可能性があるため、可能な限り、より特化したシャッフル関数を使用することを推奨します。

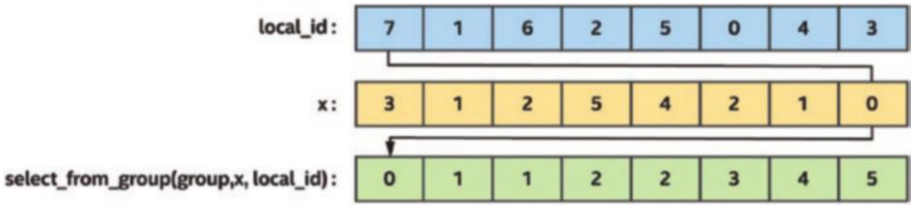

図 9-14. 汎用的な select_from_group を使用して、事前計算されたインデックスに基づいて値を並べ替え

図 9-14 では、事前に計算された順列インデックスを使用してサブグループの値をソートするため汎用シャッフルが使用されています。サブグループ内の 1 つのワーク項目に対して矢印が表示され、シャッフルの結果は `local_id` が 7 であるワーク項目の `x` の値になります。

サブグループの `group_broadcast` 関数は、サブグループ内のすべてのワーク項目のシャッフル・インデックスが同じである汎用 `select_from_group` 関数の特殊バージョンと考えることができます。

サブグループ内のすべてのワーク項目のシャッフル・インデックスが同じであることが判明している場合、select_from_group の代わりに group_broadcast を使用すると、コンパイラーに追加情報が提供され、一部の実装ではパフォーマンスが向上する可能性があります。

図 9-15 に示すように、shift_group_right 関数と shift_group_left 関数は、サブグループの内容を指定された方向に一定数だけシフトします。サブグループ内の最後の 5 つのワーク項目に返される値は未定義であり、図 9-15 では空白として表示されることに注意してください。シフトは、ループ伝達依存関係を持つループを並列化する場合や、排他的スキャンや包括的スキャンなどの一般的なアルゴリズムを実装する場合に役立ちます。

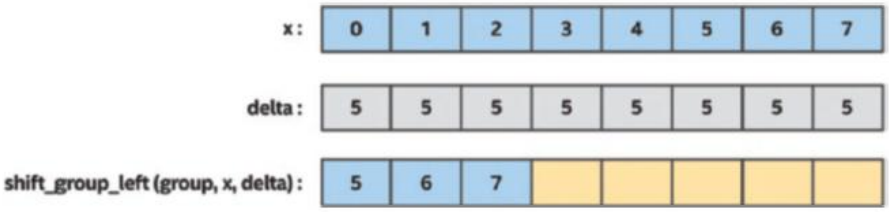

図 9-15. サブグループの x 値を 5 項目シフトするため shift_group_left を使用

permute_group_by_xor 関数は、ワーク項目のサブグループのローカル ID と固定定数に適用された XOR 演算の結果で指定される 2 つのワーク項目の値を交換します。図 9-16 と図 9-17 に示すように、隣接する値のペアの交換やサブグループの値の反転など、いくつかの一般的な通信パターンを XOR を使用して表現できます。

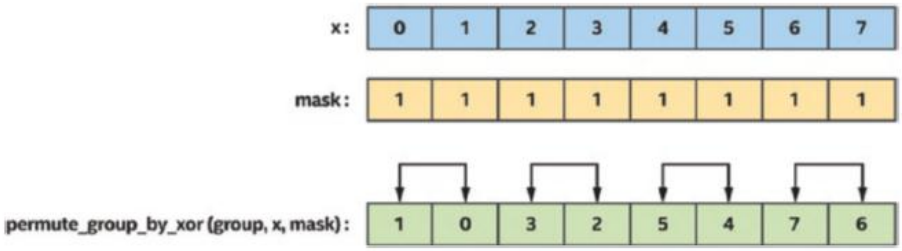

図 9-16. permute_group_by_xor を使用して 隣接する x のペアを交換

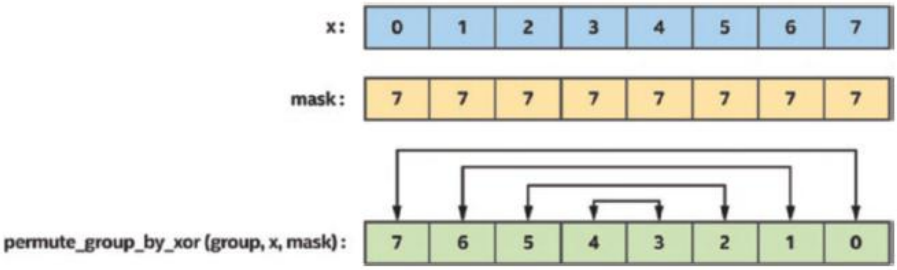

図 9-17. permute_group_by_xor を使用して x の値を反転

ブロードキャスト、投票、集合を用いたサブグループの最適化

サブグループに適用されるブロードキャスト、投票、およびその他の集合関数の動作は、ワークグループに適用される場合と同じですが、特定のコンパイラーでは積極的な最適化が可能になる場合があるため、さらに注意が必要です。たとえば、コンパイラーは、サブグループ内のすべてのワーク項目にブロードキャストされる変数レジスターの使用量を削減したり、`any_of_group` 関数と `all_of_group` 関数に基づいて制御フローの相違を推論できる場合があります。

シャッフル関数は非常に特殊であるため、サブグループでのみ使用でき、ワークグループでは使用できません。

まとめ

この章では、グループ内のワーク項目がどのように通信し、協調し、特定のタイプのカーネルのパフォーマンスを向上させるか説明しました。

まず、ND-range カーネルがワーク項目をワークグループにグループ化する方法について説明しました。ワーク項目をワークグループにグループ化することで並列実行モデルがどのように変化し、ワークグループ内のワーク項目が通信と同期により実行するようスケジュールされる保証について説明しました。

次に、ワークグループ内のワーク項目がバリアを使用して同期する方法と、カーネルでバリアがどのように表現されるか説明しました。また、カーネルを簡素化し、パフォーマンスを向上させるために、ワークグループ内のワーク項目間の通信をワークグループ・ローカル・メモリーを介して実行する方法と、ワークグループ・ローカル・メモリーをローカルアクセサーを使用して表現する方法も説明しました。

ND-range カーネルのワークグループをさらにワーク項目のサブグループに分割し、ワーク項目のサブグループで追加の通信パターンやスケジュール保証をサポートする方法について説明しました。

ワークグループとサブグループの両方において、集合関数を通じて共通の通信パターンをどのように表現し、加速できるか議論しました。

この章の概念は、第 14 章で説明されている一般的な並列パターンを理解し、第 15 章、第 16 章、および第 17 章で特定のデバイス向けに最適化する方法を理解する重要な基礎となります。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 10 章　カーネルの定義

これまで本書のコード例では、C++ ラムダ式を使用してカーネルを表現してきました。ラムダ式は、カーネルが使用される場所でカーネルを表す簡潔で便利な方法ですが、SYCL でカーネルを表す唯一の方法ではありません。
この章では、カーネルを定義する方法を詳しく検討し、C++ コーディングのニーズに最も適したカーネル形式を選択できるようにします。

この章では、カーネルを表現する次の 3 つの方法について説明して比較します:

- ラムダ式
- 名前付き関数オブジェクト（ファンクター）
- 他の言語や API で作成されたカーネルとの相互運用性を使用このトピックについてはこの章では簡単触れし、第 20 章で詳しく説明します。

この章の最後では、カーネルバンドル内のカーネルを明示的に操作して、カーネルのプロパティーを照会し、カーネルをいつどのようにコンパイルするか制御する方法について説明します。

カーネルを表現する方法が 3 つあるのはなぜか

詳細を説明する前に、カーネルを定義する方法が 3 つある理由と、それぞれの長所と短所について簡単に説明します。それらを図 10-1 まとめています。

カーネルは計算単位を表現するために使用され、カーネルの多くのインスタンスは通常、アクセラレーター上で並列に実行されることに留意してください。
SYCL は、さまざまなコーディング・スタイルのコードベースに、自然かつシームレスに統合するカーネルを表現する複数の方法をサポートし、多種多様なアクセラレーター・タイプで効率的に実行できます。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_10

カーネル表現	記述
ラムダ式	長所: ・ ラムダ式は、カーネルが使用される場所を正確に表す簡潔な方法です。 ・ ラムダ式は、最新の C++ コードでカーネルのような操作を表す一般的な方法です。 ・ ラムダのキャプチャー規則は自動的にデータをカーネルに渡します。 短所: ・ ラムダ式として表されるカーネルはテンプレート化できず、追加の作業なしではライブラリーとしてアセンブルされません (通常の関数のように)。 ・ ラムダ構文は、一部の C++ コードベースでは馴染みがないかもしれません。
名前付き関数オブジェクト (ファンクター)	長所: ・ ファンクターは、他の C++ クラスと同様に、テンプレート化、再利用、ライブラリーの一部としてリリースできます。 ・ ファンクターはカーネルに渡されるデータをより詳細に制御します。 短所: ・ 関数として表現されるカーネルでは、ラムダ式として表現されるカーネルよりも多くのコードを記述する必要があります。 ・ カーネル引数は関数に明示的に渡す必要があり、自動的にキャプチャーされません。
他の言語 API との相互運用性	長所: ・ 以前作成されたカーネルまたはライブラリーの再利用を可能にします。 ・ 大規模なアプリケーション・コードベースに SYCL のサポートを段階的に追加できます。 ・ 他の API のカーネル言語では、SYCL では追加されていない機能や表現が難しい機能がサポートされる場合があります。 短所: ・ 相互運用性はオプションの機能であり、すべての SYCL 実装または実装内のすべての SYCL デバイスでサポートされるとは限りません。 ・ 他の API で記述されたカーネルは SYCL デバイス・コンパイラーでコンパイルされないため、コンパイル時の構文チェック、カーネル引数の型チェック、および最適化の可能性が制限される可能性があります。 ・ 他の API で記述されたカーネルは、最新の C++ 機能をサポートしていない可能性があります。

図 10-1. カーネルを表現する 3 つの方法

ラムダ式としてカーネルを表現

C++ ラムダ式は、*匿名関数オブジェクト*、*無名関数オブジェクト*、*クロージャー*、または単に*ラムダ*とも呼ばれ、カーネルが使用される場所でカーネルを表現する便利な方法です。ここでは、カーネルを C++ ラムダ式として表現する方法について説明します。これは、出力付きのいくつかの基本的なサンプルコードを含む、第 1 章の C++ ラムダ式の説明を拡張したものです。

C++ ラムダ式は非常に強力で表現力豊かな構文を持っていますが、SYCL でカーネルを表現する場合、完全な C++ ラムダ式構文のサブセットのみが必要（およびサポート）です。

```
h.parallel_for(
    size,
    // カーネルラムダ式の始まり
    [=](id<1> i) { data_acc[i] = data_acc[i] + 1; }
    // カーネルラムダ式の終わり
);
```

図 10-2. ラムダ式を使用して定義された単純なカーネル

カーネルラムダ式の要素

図 10-2 は、典型的なラムダ式として記述された単純なカーネルを示しています。本書のこれまでのコード例では、この構文が使用されています。

図 10-3 は、カーネルで使用できるラムダ式の要素を示していますが、これらの要素の多くは一般的ではありません。ほとんどの場合、ラムダのデフォルトで十分であるため、一般的なカーネルラムダ式は、図 10-3 の複雑なラムダ式よりも、図 10-2 のラムダ式のようになります。

```
q.submit([&](handler&  h) {
  accessor acc{B, h};
  h.parallel_for(
      nd_range{{size}, {8}},
      [=](id<1> i) noexcept [[sycl::reqd_work_group_size(8)]] ->void {
          data_acc[i] = data_acc[i] + 1;
        });
});
```

図 10-3. カーネルラムダ式のその他の要素（オプション要素を含む）

1. ラムダ式の最初の部分では、ラムダ・*キャプチャー*を記述します。周囲のスコープから変数を*キャプチャー*すると、その変数をパラメーターとしてラムダ式に明示的に渡すことなく、ラムダ式内で使用できます。

   C++ ラムダ式は、変数をコピーするか参照を作成することで、変数のキャプチャーをサポートしていますが、カーネルラムダ式では変数はコピーによってのみキャプチャーできます。
   一般的な方法は、デフォルトのキャプチャー・モード `[=]` を使用することです。このモードでは、すべての変数が暗黙的に値によってキャプチャーされますが、キャプチャーされた各変数にコンマで区切ったキャプチャー・リストを明示的に名前付けできます。カーネル内で使用される変数が値によってキャプチャーされていない場合、コンパイル時にエラーが発生します。
   C++ 標準に従い、グローバル変数はラムダ式でキャプチャーされないことに注意してください。

2. ラムダ式の 2 番目の部分では、名前付き関数に渡されるパラメーターと同様に、ラムダ式に渡されるパラメーターを記述します。

カーネルラムダ式では、パラメーターはカーネルの呼び出し方法に依存し、並列実行スペース内のワーク項目のインデックスを識別します。さまざまな並列実行スペースと、各実行スペースでワーク項目のインデックスを識別する方法の詳細については、第 4 章を参照してください。

3. ラムダ式の最後の部分は関数本体を定義します。カーネルラムダ式の場合、関数本体は並列実行空間内の各インデックスで実行される操作を記述します。

ラムダ式には他にも部分がありますが、それらはオプションであるか、あまり使用されない、または SYCL 2020 ではサポートされていないものです。

4. SYCL 2020 では*指定子*（`mutable` など）は定義されていないため、サンプルコードには何も示されません。

5. *例外の指定*はサポートされていますが、カーネルでは例外がサポートされていないため、指定する場合は `noexcept` にする必要があります。

6. ラムダ*属性*がサポートされており、カーネルのコンパイル方法を制御するために使用できます。たとえば、`reqd_work_group_size` 属性は、カーネルに特定のワークグループ・サイズを要求するのに使用でき、`device_has` 属性はカーネルに特定のデバイスアスペクトを要求するのに使用できます。第 12 章では、属性とアスペクトを使用したカーネルの特殊化に関する詳細を説明します。

7. *戻り値の型*は指定できますが、`void` 以外の戻り値型はカーネルではサポートされていないため、指定する場合は `void` にする必要があります。

ラムダ・キャプチャー: 暗黙的または明示的か?

一部の C++ スタイルガイドでは、特にラムダ式がスコープ境界を越える場合、ダングリング・ポインターの問題が発生する可能性があるため、ラムダ式の暗黙的（またはデフォルト）キャプチャーを推奨していません。カーネルラムダはホストコードとは別にデバイス上で非同期的に実行されるため、ラムダを使用してカーネルを表す場合も同じ問題が発生する可能性があります。

暗黙的なキャプチャーは便利で簡潔であるため、SYCL カーネルでは一般的な方法であり、本書でも慣例的に使用していますが、暗黙的なキャプチャーの簡潔さを優先するか、明示的なキャプチャーの明瞭さを優先するかは私たちが決定できます。

カーネルラムダ式の識別

カーネルがラムダ式として記述されている場合、提供する必要のある要素がもう 1 つあります。ラムダ式は*匿名*であるため、ラムダ式として記述されたカーネルを一意に識別するには、SYCL では明示的なカーネル名テンプレート・パラメーターが必要になることがあります。

```
// この例では、 "class Add" はカーネルラムダ式に名前を付けます。
h.parallel_for<class  Add>(size, [=](id<1> i) {
 data_acc[i] = data_acc[i] + 1;
});
```

図 10-4. カーネルラムダ式の識別

カーネルラムダ式に名前を付けることは、カーネルが別のデバイス・コード・コンパイラーによってコンパイルされたときに、ホスト・コード・コンパイラーがどのカーネルを呼び出すかを識別する方法です。カーネルラムダに名前を付けると、図 10-9 のように、コンパイルされたカーネルの実行時の調査や、名前によるカーネルの構築も可能になります。

カーネル名テンプレート・パラメーターが不要な場合、より簡潔なコードをサポートするため、ほとんどの SYCL 2020 コンパイラーではカーネル名テンプレート・パラメーターはオプションです。カーネル名テンプレート・パラメーターが必要ない場合は、図 10-5 に示すように、コードをコンパクトにできます。

```
h.parallel_for(size, [=](id<1> i) {
   data_acc[i] = data_acc[i] + 1;
});
```

図 10-5. 無名カーネルラムダ式の使用

ラムダ式のカーネル名テンプレート・パラメーターはあまり必要ないため、通常は無名ラムダから開始し、カーネル名テンプレート・パラメーターが必要な特定の場合にのみカーネル名を追加します。

カーネル名テンプレート・パラメーターが必要ない場合、冗長性を減らすために無名カーネルラムダを使用することを推奨します。

名前付き関数オブジェクトのカーネル

名前付き関数オブジェクト（*ファンクター*とも呼ばれる）は、明確に定義されたインターフェイスを維持しながら任意のデータ集合を操作する C++ で確立されたパターンです。カーネルを表すのに使用される場合、名前付き関数オブジェクトのメンバー変数はカーネルが操作できる状態を定義し、オーバーロードされた関数呼び出し `operator()` が並列実行スペース内の各ワーク項目に対して呼び出されます。

名前付き関数オブジェクトでカーネルを表現するには、ラムダ式よりも多くのコードが必要ですが、冗長性が追加されることで、より高度な制御と追加機能が提供されます。たとえば、カーネルが使用するバッファーやデータ値は、ラムダ式によって自動的にキャプチャーされず、カーネルに明示的に渡す必要があるため、名前付き関数オブジェクトとして表現されるカーネルの分析や最適化の方が簡単な場合があります。

名前付き関数オブジェクトとして表現されたカーネルは、デバッグや再利用が容易になり、別のヘッダーファイルやライブラリーの一部としてリリースされる場合もあります。

最後に、名前付き関数オブジェクトは他の C++ クラスと同じであるため、名前付き関数オブジェクトとして表現されるカーネルはテンプレート化できます。C++20 ではテンプレート化されたラムダ式が追加されましたが、C++17 をベースにした SYCL 2020 のカーネルではテンプレート化されたラムダ式はサポートされていません。

カーネル名前付き関数オブジェクトの要素

図 10-6 のコードは、名前付き関数オブジェクトとして表されるカーネルの一般的な使用方法を示しています。この例では、カーネルへのパラメーターはクラス・コンストラクターに渡され、カーネル自体はオーバーロードされた関数呼び出し `operator()` 内にあります。

```
class Add {
 public:
   Add(accessor<int> acc) : data_acc(acc) {}
    void operator()(id<1> i) const {
       data_acc[i] = data_acc[i] + 1;
   }

 private:
   accessor<int> data_acc;
};

int main() {
 constexpr size_t size = 16;
 std::array<int, size> data;

  for (int i = 0; i < size; i++) {
    data[i] = i;
  }

  {
    buffer data_buf{data};

    queue q;
    std::cout
        << "Running on device: "
        << q.get_device().get_info<info::device::name>()
        << "\n";

    q.submit([&](handler& h) {
       accessor data_acc{data_buf, h};
       h.parallel_for(size, Add(data_acc));
    });
  }
  // ...
```

図 10-6. 名前付き関数オブジェクトのカーネル

カーネルが名前付き関数オブジェクトとして表現される場合、名前付き関数オブジェクト型は*デバイスでコピー可能*になるために SYCL 2020 の規則に従う必要があります。非公式には、これは、名前付き関数オブジェクトをバイトごとに安全にコピーでき、名前付き関数オブジェクトのメンバー変数をデバイス上で実行されるカーネルコードに渡してアクセスできることを意味します。簡単にコピー可能な C++ 型は、暗黙的にデバイスコピー可能です。

オーバーロードされた関数呼び出し operator() への引数は、ラムダ式として表現されるカーネルと同様に、カーネルの起動方法によって異なります。

図 10-7 のコードは、名前付き関数オブジェクトとして定義されたカーネルで、reqd_work_group_size 属性などオプションのカーネル属性を使用する方法を示しています。カーネルが名前付き関数オブジェクトとして定義されている場合、オプションのカーネル属性には 2 つの有効な位置があります。これは、オプションのカーネル属性の位置が 1 つだけ有効なラムダ式として記述されたカーネルとは異なります。

```
class AddWithAttribute {
  public:
   AddWithAttribute(accessor<int> acc) : data_acc(acc) {}
    [[sycl::reqd_work_group_size(8)]] void  operator()(
         id<1> i) const {
      data_acc[i] = data_acc[i] + 1;
   }

 private:
   accessor<int> data_acc;
};

class MulWithAttribute {
  public:
   MulWithAttribute(accessor<int> acc) : data_acc(acc) {}
   void operator()
       [[sycl::reqd_work_group_size(8)]] (id<1> i) const {
      data_acc[i] = data_acc[i] * 2;
   }

 private:
   accessor<int> data_acc;
};
```

図 10-7. 名前付き関数オブジェクトでオプション属性を使用

すべての関数オブジェクトには名前が付けられているため、関数オブジェクトがテンプレート化されている場合でも、ホスト・コード・コンパイラーは関数オブジェクトのタイプを使用して、デバイス・コード・コンパイラーで生成されたカーネルコードを識別できます。カーネル関数オブジェクトに名前を付けるには、追加のカーネル名テンプレート・パラメーターは必要ありません。

カーネルバンドル内のカーネル

SYCL カーネルに関連して知っておくべき最後のトピックは、SYCL *カーネル・オブジェクト*と SYCL *カーネルバンドル*に関するものです。カーネル・オブジェクトとカーネルバンドルの知識は、一般的なアプリケーション開発には必要ありませんが、アプリケーションのパフォーマンスを調整するときに役立つことがあります。また、SYCL 実装によってカーネルがどのように構成され、管理されるかを理解するのにも役立ちます。

SYCL カーネルバンドルは、アプリケーションで使用される SYCL カーネルまたは SYCL 関数のコンテナーです。アプリケーション内のカーネルバンドルの数は、SYCL コンパイラーによって異なります。アプリケーションによっては、複数のカーネルがあってもカーネルバンドルが 1 つしかない場合があります。また、カーネルが少数であってもカーネルバンドルが複数存在する場合もあります。

SYCL カーネルバンドルとそれに含まれるカーネルまたは関数は、次の 3 つの状態のいずれかです:

- **入力状態:** この状態のカーネルバンドルは通常、何らかの中間表現になっており、デバイス上で実行する前にジャストインタイム (JIT) コンパイルする必要があります。

- **オブジェクト状態:** この状態のカーネルバンドルは通常、ホストのアプリケーション・コンパイラーによって生成されたオブジェクト・ファイルと同様にコンパイルされますが、リンクはされません。

- **実行状態:** この状態のカーネルバンドルはデバイスコードに完全にコンパイルされており、デバイス上で実行する準備ができています。ホスト・アプリケーションのコンパイル時に事前 (AOT) コンパイルされるカーネルバンドルはこの状態になります。

仕様によれば必須ではありませんが、多くの SYCL コンパイラーは、最大数の SYCL デバイスへの移植性を確保するため、最初にカーネルを中間表現にコンパイルします。これは、通常、アプリケーション・カーネル・バンドルが、最初は入力状態にあることを意味します。次に、多くの SYCL ランタイム・ライブラリーは、必要に応じてカーネルバンドルを入力状態から実行可能状態に “遅延” コンパイルします。

これは通常、実行されないカーネルをコンパイルしないため、アプリケーションの起動を高速化する適切なポリシーであると言えます。ただし、このポリシーの欠点は、カーネルを最初に実行するときに、カーネルをコンパイルする時間とカーネルを送信して実行する通常必要な時間の両方が含まれるため、時間がかかることです。複雑なカーネルでは、カーネルのコンパイルに要する時間が非常に長くなる可能性があるため、アプリケーションのロード時などアプリケーション実行中の別のポイント、または別のバックグラウンド・スレッドにコンパイルを移行することが望ましい場合があります。

カーネルをいつどのようにコンパイルするか、より細かく制御するためカーネルをキューに送信する前に、カーネルバンドルをコンパイルするよう明示的に要求できます。事前コンパイルされたカーネルバンドルは、カーネルが実行のためにキューに送信されると使用できます。図 10-8 は、カーネルがキューに送信される前にアプリケーションで使用されるすべてのカーネルをコンパイルする方法と、コンパイル済みのカーネルバンドルを使用する方法を示しています。

```
auto kb = get_kernel_bundle<bundle_state::executable>(
        q.get_context());

std::cout
    << "All kernel compilation should be done now.\n";

q.submit([&](handler&  h) {
   // カーネルバンドルから事前コンパイルされたカーネルを使用します。
   h.use_kernel_bundle(kb);

   accessor data_acc{data_buf, h};
    h.parallel_for(range{size}, [=](id<1> i) {
      data_acc[i] = data_acc[i] + 1;
   });
});
```

図 10-8. カーネルバンドルを使用してカーネルを明示的にコンパイル

この例では、SYCL キューに関連付けられた SYCL コンテキスト内のすべてのデバイスに対して、実行可能状態のカーネルバンドルを要求します。これにより、アプリケーション内のすべてのカーネルがまだ実行可能状態になっていない場合、ジャストインタイム・コンパイルされます。この例では、カーネルは非常に短いため、コンパイルに時間はかかりませんが、カーネルの数が多い場合や、カーネルが複雑な場合は、この手順にかなりの時間がかかる可能性があります。もちろん、すべてのカーネルが事前にコンパイルされていたり、すべてのカーネルがすでにジャストインタイム・コンパイルされている場合、すべてのカーネルがすでに実行可能状態になっているため、この操作には実質的にコストが発生しません。

カーネルをいつ、どのようにコンパイルするかさらに細かく制御したい場合、特定のデバイス、またはプログラム内の特定のカーネルのカーネルバンドルを要求できます。これにより、プログラム内のカーネルの一部を選択的に即時コンパイルし、他のカーネルは後で、または必要に応じてコンパイルできるようになります。図 10-9 は、プログラム内のすべてのカーネルと SYCL コンテキスト内のすべてのデバイスではなく、クラス Add カーネル名によって識別されるカーネルのみ、および SYCL キューに関連付けられた SYCL デバイスのみをコンパイルする方法を示しています。

```
auto kid = get_kernel_id<class Add>();
auto kb = get_kernel_bundle<bundle_state::executable>(
    q.get_context(), {q.get_device()}, {kid});

std::cout << "Kernel compilation should be done now.\n";

q.submit([&](handler& h) {
  // カーネルバンドルから事前コンパイルされたカーネルを使用します。
  h.use_kernel_bundle(kb);

  accessor data_acc{data_buf, h};
  h.parallel_for<class Add>(range{size}, [=](id<1> i) {
     data_acc[i] = data_acc[i] + 1;
  });
});
```

図 10-9. カーネルバンドルを使用してカーネルを明示的かつ選択的にコンパイル

これは、カーネルラムダ式に名前を付ける必要がある、もう 1 つのまれなケースです。命名しないと、コンパイルするカーネルを識別する方法がありません。

カーネルバンドルを使用して、アプリケーションでカーネルを予測どおりにコンパイルします。

カーネルバンドル内のカーネルは、コンパイルされたカーネルの情報を照会するためにも使用できます。たとえば、特定デバイスのカーネルの最大ワークグループ・サイズを判別できます。場合によっては、カーネルと特定のデバイスに有効な値を選択するため、このようなタイプのカーネルの照会が必要になることがあります。状況によっては、カーネルの照会からヒントを得ることで、アプリケーションがカーネルと特定のデバイスに最適な値を動的に選択できるようになります。

カーネルを識別し、コンパイルされたカーネルバンドルからカーネル・オブジェクトを取得し、そのカーネル・オブジェクトを使用してデバイス固有の照会を実行する基本的なメカニズムを図 10-10 に示します。利用可能なカーネルクエリーの完全なリストについては、第 12 章で説明します。

```
auto kid = get_kernel_id<class Add>();
auto kb = get_kernel_bundle<bundle_state::executable>(
    q.get_context(), {q.get_device()}, {kid});
auto  kernel = kb.get_kernel(kid);

std::cout
    << "The maximum work-group size for the kernel and " "this device is: "
    << kernel.get_info<info::kernel_device_specific::
                            work_group_size>(
           q.get_device())
    << "\n";

std::cout
    << "The preferred work-group size multiple for the 「
       "kernel and this device is: "
    << kernel.get_info<
           info::kernel_device_specific::
               preferred_work_group_size_multiple>(
           q.get_device())
    << "\n";
```

出力例:

```
Running on device: NVIDIA GeForce RTX 3060
The maximum work-group size for the kernel and this device is: 1024
The preferred work-group size multiple for the kernel and this device is: 32
```

出力例:

```
Running on device: Intel(R) Data Center GPU Max 1100
The maximum work-group size for the kernel and this device is: 1024
The preferred work-group size multiple for the kernel and this device is: 16
```

出力例:

```
Running on device: Intel(R) UHD Graphics 770
The maximum work-group size for the kernel and this device is: 512
The preferred work-group size multiple for the kernel and this device is: 64
```

図 10-10. カーネルバンドル内のカーネルの照会

これは、カーネルラムダ式に名前を付ける必要がある、もう 1 つのまれなケースです。命名しないと、照会するカーネルを識別する方法がありません。

他の API との相互運用性

SYCL 実装が別の API 上に構築されると、その実装は、基礎となる API のメカニズムを使用して定義されたカーネルと相互運用できるようになります。これにより、アプリケーションは、基礎となる API をすでに使用している既存のコードベースに SYCL を簡単かつ段階的に統合できるようになります。このトピックの詳細については、第 20 章でカバーしています。この章では、他のソース言語または API を介して作成されたカーネルまたはカーネルバンドルとの相互運用性が、カーネルを表現する 3 つ目の方法を提供することを認識するだけで十分です。

まとめ

この章では、カーネルを定義するさまざまな方法を説明しました。カーネルを C++ ラムダ式、または名前付き関数オブジェクトとして表現することで、SYCL を既存の C++ コードベースにシームレスに統合する方法について説明しました。新しいコードベースでは、アプリケーションやライブラリーのニーズに応じてカーネルを定義する最適な方法を選択できるよう、さまざまなカーネル表現の長所と短所についても説明しました。

SYCL アプリケーションでカーネルがどのようにコンパイルされ、またカーネルバンドル内のカーネルを直接操作してコンパイルプロセスを制御する方法も説明しました。ほとんどのアプリケーションではこのレベルの制御は必要ありませんが、アプリケーションを調整するときに知っておくと便利な手法です。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 11 章　ベクトルと数学配列

ベクトルはデータの集合です。コンピューターの並列処理はコンピューター・ハードウェアの集合から生まれ、データは関連するグループ（RGB ピクセルのカラーチャネルなど）で処理されることが多いため、ベクトルが役立つ場合があります。この概念は非常に重要であるため、さまざまな SYCL ベクトルタイプとその活用方法について 1 つの章として説明します。スカラー演算の*ベクトル化*はデバイスの種類と実装によって異なるため、この章では詳しく説明しません。スカラー演算のベクトル化については、第 16 章で説明します。

この章では、以下をカバーします:

- ベクトル型とは何か？
- SYCL 数学配列（`marray`）型とベクトル（`vec`）型の違いは何か？
- 配列とベクトルはいつ、どのように使用するか？

実際のコード例を使用して配列とベクトルについて説明し、これらの型を活用する上で最も重要なことを強調します。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_11

ベクトル型の曖昧さ

並列プログラミングの専門家と話をすると、ベクトルは驚くほど議論の多いトピックです。著者の経験では、これは人によってベクトルの定義や考え方が異なるためです。

この章でベクトル型と呼ぶものは、大きく分けて 2 つの考え方があります。

1. ピクセルの RGB または YUV カラーチャネルなど、グループとして参照および操作する可能性のあるデータをグループ化する**便利なタイプ**です。ピクセルクラスまたは構造体を定義して、それに `+` などの数学演算子を定義することもできますが、このベクトル型によってすぐにこの処理を実行できます。便利な型は GPU をプログラムする多くのシェーダー言語で見つかるため、これは多くの GPU 開発者の間では一般的です。

2. コードがハードウェア内の **SIMD（単一命令、複数データ）命令セット** にどのようにマップされるかを記述するメカニズムとして。たとえば、一部の言語および実装では、`float8` の操作がハードウェア内の 8 レーン SIMD 命令にマップされる可能性があります。SIMD ベクトル型は、CPU 固有の組み込み関数の高レベルの代替として多くの言語で使用されており、この考え方は多くの CPU 開発者の間では一般的です。

この 2 つのベクトル型の解釈は非常に異なりますが、SYCL や他の言語が CPU と GPU の両方に適用されるようになると、意図せず組み合わさり混同されるようになりました。`vec` クラス（SYCL 1.2.1 に存在し、SYCL 2020 でもまだ存在）は、どちらの解釈とも互換性がありますが、`marray` クラス（SYCL 2020 で導入）は SIMD ベクトル・ハードウェア命令とは無関係な型として明示的に記述されています。

変化が目前に迫っています: SIMD 型

SYCL 2020 には、2 番目の解釈（SIMD マッピング）に明示的に結び付けられたベクトル型はまだ含まれていません。ただし、特定のアーキテクチャー用にコードを調整し、コンパイラーによるベクトル化から制御を取得したい熟練プログラマー向けに設計された、ハードウェア内の SIMD 命令に直接マップする明示的なベクトルコードを記述できる拡張機能がすでに用意されています。また、2 番目の解釈をカバーするため、提案されている C++ **`std::simd`** テンプレートと一致する別のベクトル型が SYCL に最終的に導入されることも予想されます。この新しいクラスでコードが明示的なベクトルスタイルで記述されている場合、非常に明確になり混乱が軽減されます。 SYCL の既存の拡張機能と、将来の **`std::simd`** のような型はどちらもニッチな機能であり、使用する開発者は少数であると予想されます。

`marray` と専用の SIMD クラスを使用すると、プログラマーの意図を記述するコードが明確になります。これにより、エラーが起こりにくくなり、混乱が減り、次のような疑問が生じたときに熟練開発者の間で白熱した議論が減ることもあります: “ベクトルとは何か？”

SYCL ベクトル型に関するメンタルモデル

本書では、ワーク項目をグループ化して、サブグループ・バリアやシャッフルなどの強力な通信、および同期プリミティブを公開する方法について説明します。これらの操作をベクトル・ハードウェアで効率良く実行するには、サブグループ内の異なるワーク項目が結合され、SIMD 命令にマップされているという前提があります。言い換えると、複数のワーク項目がコンパイラーによってグループ化されることでハードウェア内の SIMD 命令にマップできるようになります。第 4 章で説明したように、これはベクトル・ハードウェア上で動作する SPMD（単一プログラム、複数データ）プログラミング・モデルの基本的な前提事項であり、

単一のワーク項目がハードウェアの SIMD 命令となる操作*全体*を定義するのではなく、ハードウェアの SIMD 命令となる可能性のある*レーン*を構成します。SPMD スタイルでプログラミングする場合、ハードウェアで SIMD 命令にマッピングする際に、コンパイラーは常にワーク項目全体をベクトル化すると考えることができます。

ベクトル型を持たない言語や GPU シェーディング言語を使用する開発者にとって、SYCL ベクトル型はワーク項目に対してローカルであると考えることができます。つまり、4 要素のベクトル 2 つを加算する場合、その加算にはハードウェア命令 4 つが必要になる可能性があります（ワーク項目の観点からはスカラー化されます）。ベクトルの各要素は、ハードウェア内の異なる命令/クロックサイクルによって加算されます。これは、ベクトル型を便宜上のものとして解釈することと一致しています。つまり、ソース内では、4 つのスカラー演算を実行する代わりに、単一の操作で 2 つのベクトルを加算できます。

CPU 向けのプログラミング経験がある開発者は、ベクトル型に関係なく、多くのコンパイラーで SIMD ハードウェアを暗黙的に使用するベクトル化がデフォルトで行われることを知っています。コンパイラーは、ワーク項目全体でこの暗黙的なベクトル化を行ったり、整形式のループからベクトル演算を抽出したり、ベクトル命令にマッピングするときにベクトル型を優先したりします。詳細については、第 16 章を参照してください。

この章では、ベクトル型（`marray` と `vec` の両方）を使用してベクトルを習得することに焦点を当てています。これは、SYCL でプログラミングする際に留意すべきことです。

他の実装も可能です

理論上、異なるコンパイラーと SYCL の実装では、コード内のベクトルデータ型を SIMD ベクトル・ハードウェア命令にマップする方法で異なった決定を下すことができます。この章で説明する考え方とプログラミング・パターンは、ほとんどの（理想的にはすべての）SYCL 実装に適用できますが、効率的な SIMD 命令にマップされるコードの記述方法を理解するには、ベンダーのドキュメントと最適化ガイドを調査する必要があります。

数学配列（`marray`）

SYCL 数学配列型（`marray`）（図 11-1 を参照）は、SYCL 2020 で新しく追加されたもので、ベクトル型の動作に関する解釈を明確にする定義が行われています。`marray` は、前述したベクトル型の最初の解釈（ベクトル・ハードウェア命令とは無関係の便利な型）を明示的に表します。名前を「ベクトル」から「配列」に置き換えることで、その型がハードウェア上で論理的にどのように実装されるか、覚えやすくまた理解しやすくなります。

型エイリアス	同義の marray
mcharN	marray<int8_t, N>
mucharN	marray<uint8_t, N>
mshortN	marray<int16_t, N>
mushortN	marray<uint16_t, N>
mintN	marray<int32_t, N>
muintN	marray<uint32_t, N>
mlongN	marray<int64_t, N>
mulongN	marray<uint64_t, N>
mhalfN	marray<half, N>
mfloatN	marray<float, N>
mdoubleN	marray<double N>
mboolN	marray<bool, N>

図 11-1. 数学配列の型エイリアス

marray クラスは、その要素型と要素数のテンプレートです。要素数パラメーター NumElements は正の整数です。NumElements が 1 の場合、marray は同等のスカラー型に暗黙的に変換可能です。要素型パラメーター DataT は、C++ 標準で定義されるため数値型である必要があります。

marray は、std::array のような配列コンテナーですが、配列上の数学演算子（例: +、+=）と SYCL 数学関数（例: sin、cos）もサポートされます。SYCL デバイスでの並列計算に効率的かつ最適化された配列操作を提供するように設計されています。

便宜上、SYCL は数学配列の型エイリアスを提供します。これらの型エイリアスでは、要素数 N は 2、3、4、8、または 16 にする必要があります。

図 11-2 は、4 つの浮動小数点数で構成される配列内のすべての要素に cos 関数を適用する簡単な例を示しています。この例では、各ワーク項目に割り当てられたデータ集合のすべての要素に適用される操作を表現するため marray を使用する利便性を強調しています。

```
queue q;
marray<float, 4> input{1.0004f, 1e-4f, 1.4f, 14.0f};
marray<float, 4> res[M];
for (int i = 0; i < M; i++)
   res[i] = {-(i + 1), -(i + 1), -(i + 1), -(i + 1)};
{
  buffer in_buf(&input, range{1});
  buffer re_buf(res, range{M});

  q.submit([&](handler  &cgh) {
    accessor re_acc{re_buf, cgh, read_write};
    accessor in_acc{in_buf, cgh, read_only};

    cgh.parallel_for(range<1>(M), [=](id<1> idx) {
       int i = idx[0];
       re_acc[i] = cos(in_acc[0]);
    });
  });
}
```

図 11-2. marray を使った簡単な例

このカーネルを大規模なデータ M に対して実行することで、ベクトル型を操作する SIMD 命令セットへのコードのマッピング方法を指定することなく、marray の 4 要素よりもはるかに広いデバイスを含む、さまざまな種類のデバイス上で良好な並列処理を実現できます。

ベクトル（vec）

SYCL ベクトル型（vec）は SYCL 1.2.1 に存在していますが、SYCL 2020 にも引き続き含まれています。前述のように、vec はベクトル型のどちらの解釈とも互換性があります。実際には、vec は通常、便利な型として解釈されるため、コードの可読性を向上させ、曖昧さを減らすには、marray を使用することをお勧めします。ただし、この推奨事項には、ベクトルのロードとストア、バックエンドのネイティブベクトル型との相互運用性、および “スウィズル” の 3 つの例外があり、ここでは、それらについて説明します。

marray と同様に、vec クラスは、要素数と要素タイプに基づいてテンプレート化されます。ただし、marray とは異なり、NumElements パラメーターは 1、2、3、4、8、または 16 のいずれかである必要があり、それ以外の値を指定するとコンパイルに失敗します。これは、vec の設計に影響するベクトル型に関する混乱の良い例です。ベクトルのサイズを 2 の小さな累乗に制限することは、SIMD 命令セットでは理にかなっています。しかし、便利な型を求めているプログラマーには恣意的に見えます。 要素タイプ・パラメーター DataT は、デバイスコードでサポートされる基本スカラータイプのいずれかでなければなりません。

また、marray と同様に、vec は 2、3、4、8、および 16 要素の短縮型エイリアスを用意しています。marray エイリアスにはプリフィクス “m” が付きますが、vec エイリアスには付きません。たとえば、uint4 は vec<uint32_t, 4> のエイリアスであり、float16 は vec<float, 16> のエイリアスです。ベクトル型を扱う場合、どのクラスを扱っているか確実に把握するために、このプリフィクス “m” の有無に注意を払う必要があります。

ロードとストア

vec クラスは、ベクトル要素をロードおよびストアするメンバー関数を提供します。これらの操作は、ベクトルのチャネルと同じタイプのオブジェクトを格納する連続したメモリー位置に対して実行されます。

ロードとストア関数を図 11-3 に示します。load メンバー関数は、multi_ptr が指すアドレスにあるメモリーから、DataT の NumElements * offset 要素だけオフセットされた DataT 型の値を読み取り、それらの値を vec のチャネルに書き込みます。store メンバー関数は、vec のチャネルを読み取り、それらの値を、DataT の NumElements * offset 要素によってオフセットされた multi_ptr のアドレスのメモリーに書き込みます。

パラメーターはアクセサーや生のポインターではなく、multi_ptr であることに注意してください。multi_ptr のデータ型は Data、つまり、vec クラス特殊化のコンポーネントのデータ型です。これには、ロードまたはストアに渡されるポインターが vec インスタンス自体のコンポーネント・タイプと一致する必要があります。

```
template <access::address_space AddressSpace, access::decorated IsDecorated>
  void load(size_t offset, multi_ptr<DataT, AddressSpace, IsDecorated> ptr);

template <access::address_space addressSpace, access::decorated IsDecorated>
   void store(size_t offset, multi_ptr<DataT, AddressSpace, IsDecorated> ptr)
  const;
```

図 11-3. vec ロードとストア関数

ロード関数とストア関数の使用例を図 11-4 に示します。

```
std::array<float, size> fpData;
for (int i = 0; i < size; i++) {
   fpData[i] = 8.0f;
}

buffer fpBuf(fpData);

queue q;
q.submit([&](handler&  h) {
  accessor acc{fpBuf, h};

  h.parallel_for(workers, [=](id<1> idx) {
    float16 inpf16;
    inpf16.load(idx, acc.get_multi_ptr<access::decorated::no>());
    float16 result = inpf16 * 2.0f;
    result.store(idx, acc.get_multi_ptr<access::decorated::no>());
  });
});
```

図 11-4. ロートとストアメンバー関数を使用

SYCL ベクトルロードおよびストア関数は、ベクトル演算を表現する抽象化を提供しますが、実際のパフォーマンス上の利点は、ベースとなるハードウェア・アーキテクチャーとコンパイラーの最適化によって決まります。プロファイル・ツールを使用してパフォーマンスを解析し、さまざまな方法を試して、特定のユースケースでベクトルのロード操作とストア操作を最適に活用する方法を特定することを推奨します。

ベクトルのロードおよびストア操作が SIMD 命令にマップされることは期待できませんが、ベクトルロードおよびストア関数を使用すると、メモリー帯域幅の利用率を向上できます。

ベクトル型を効率良く操作するには、各ワーク項目が連続したメモリーブロックにアクセスすることをコンパイラーに知らせ、特定のデバイスではこの情報を活用して複数の要素を一度にロードまたはストアできるため効率が向上します。

バックエンド・ネイティブ・ベクトル型との相互運用性

SYCL `vec` クラス・テンプレートは、バックエンドのネイティブベクトル型（存在する場合）との相互運用性も提供します。バックエンド・ネイティブ・ベクトル型は、メンバー型 `vector_t` によって定義され、デバイスコードでのみ使用できます。`vec` クラスは `vector_t` のインスタンスから構築でき、暗黙的に `vector_t` のインスタンスに変換できます。

`vector_t` の使用例は限定されているため、ほとんどのユーザーは `vector_t` を使用する必要はありません。これは、カーネル関数内から呼び出されるバックエンド・ネイティブ関数との相互運用性を可能にするためだけに存在します（たとえば、SYCL カーネル内から OpenCL C で記述された関数を呼び出すなど）。

スウィズル操作

グラフィックス・アプリケーションでは、*スウィズル*とはベクトルのデータ要素を並べ替えることを意味します。たとえば、ベクトル `a` に要素 `{1, 2, 3, 4}` が含まれており、4 要素ベクトルのコンポーネントを `{x, y, z, w}` として参照する場合、`b = a.wxyz()` と記述でき、ベクトル `b` の値は `{4, 1, 2, 3}` になります。この構文は、コードをコンパクトにするため、このような操作を効率良く実行できるハードウェアがあるアプリケーションでは一般的です。

`vec` クラスを使用すると、図 11-5 に示すように、2 つの方法でスウィズルを実行できます。

```
template <int... swizzleindexes>
__swizzled_vec__ swizzle() const;
__swizzled_vec__ XYZW_ACCESS() const;
__swizzled_vec__ RGBA_ACCESS() const;
__swizzled_vec__ INDEX_ACCESS() const;

#ifdef SYCL_SIMPLE_SWIZZLES
// numElements <= 4 の場合にのみ利用できます
// XYZW_SWIZZLE は、以下の要素を繰り返す順列です：
// x、y、z、w (numElements の値に従う)
__swizzled_vec__ XYZW_SWIZZLE() const;

// numElements == 4 の場合にのみ利用できます
// RGBA_SWIZZLE は、r, g, b, a の繰り返しによる順列です。
__swizzled_vec__ RGBA_SWIZZLE() const;
#endif
```

図 11-5. vec swizzle メンバー関数

swizzle メンバー関数テンプレートを使用すると、テンプレート・メンバー関数 swizzle を呼び出してスウィズル操作を実行できます。このメンバー関数は、可変数の整数テンプレート引数を受け取ります。各引数は、ベクトル内の対応する要素のスウィズル・インデックスを表します。スウィズル・インデックスは、0 から NumElements-1 までの整数である必要があります。ここで、NumElements は元の SYCL ベクトル内の要素数を表します（たとえば、4 つの要素のベクトルの場合は vec.swizzle<2, 1, 0, 3>()）。swizzle メンバー関数の戻り値の型は、常に __swizzled_vec_ のインスタンスです。これは、スウィズルされたベクトルを表す実装定義の一時クラスです。swizzle を呼び出しても、swizzle 操作はすぐには実行されないことに注意してください。代わりに、返された __swizzled_vec__ インスタンスが式内で使用されると、スウィズル操作が実行されます。

SYCL 仕様で XYZW_SWIZZLE および RGBA_SWIZZLE として記述される単純な swizzle メンバー関数のセットは、スウィズル操作を実行する代替方法として提供されています。これらのメンバー関数は、最大 4 つの要素を持つベクトルに対してのみ使用でき、SYCL ヘッダーファイルの前に SYCL_SIMPLE_SWIZZLES マクロが定義されている場合にのみ使用できます。

単純な swizzle メンバー関数を使用すると、{x、y、z、w} または {r、g、b、a} という名前を使用してベクトル要素を参照し、これらの要素名を直接使用してメンバー関数を呼び出すことでスウィズル操作を実行できます。

たとえば、単純なスウィズルでは、前に使用した XYZW スウィズル構文 a.wxyz() が有効になります。同じ操作は、RGBA スウィズルを使用して a.argb() と記述することで同等に実行できます。シンプルなスウィズルを使用すると、よりコンパクトなコードや、他の言語、特にグラフィックス・シェーディング言語に近いコードを生成できます。ベクトルに XYZW 位置データまたは RGBA カラーデータが含まれている場合、単純なスウィズルによってプログラマーの意図をさらに適切に表現できます。単純な swizzle メンバー関数の戻り値の型も __swizzled_vec__ です。swizzle メンバー関数テンプレートと同様に、返された __swizzled_vec__ インスタンスが式内で使用されるとスウィズル操作が実行されます。

```
constexpr int size = 16;

std::array<float4, size> input;
for (int i = 0; i < size; i++) {
   input[i] = float4(8.0f, 6.0f, 2.0f, i);
}

buffer b(input);

queue q;
q.submit([&](handler&  h) {
  accessor a{b, h};

  // x()、y()、z()、w() などの関数を使用して
  // ベクトルの個々の要素にアクセスできます。
  //
  // "スウィズル" は、必要なスウィズル順序に相当するベクトルメンバー
  // (たとえば zyx() または任意の要素の組み合わせ)
  // を呼び出すことによって使用できます。
  // スウィズルは元のベクトルと同じサイズである必要はありません。
  //
  h.parallel_for(size, [=](id<1> idx) {
     auto e = a[idx];
     float w = e.w();
     float4 sw = e.xyzw();
     sw = e.xyzw() * sw.wzyx();
     sw = sw + w;
     a[idx] = sw.xyzw();
  });
});
```

図 11-6. __swizzled_vec__ クラスの使用例

図 11-6 は、単純なスウィズルと __swizzled_vec__ クラスの使用方法を示しています。__swizzled_vec__ はコード内に直接出現しませんが、b.xyzw() * sw.wzyx() などの式内で使用されます。b.xyzw() と sw.wzyx() の戻り値の型は __swizzled_vec__ のインスタンスであり、乗算は結果が float4 変数 sw に割り当てられるまで評価されません。

ベクトル型の実行方法

この章を通して説明されているように、ベクトル型とそれがハードウェアにマッピングされる方法には 2 つの異なる解釈があります。ここまでは、意図的にこれらのマッピングについて高レベルでのみ説明してきました。このセクションでは、ベクトル型のさまざまな解釈が SIMD レジスターなど低レベルのハードウェア機能にどのようにマッピングされるかを詳しく見て、両方の解釈でベクトル・ハードウェアを効率良く使用できることを示します。

利便性のある型としてのベクトル

ベクトルが利便性タイプ（`marray` や通常は `vec` など）からハードウェアの実装にどのようにマッピングされるかは、主に 3 つのポイントがあります:

1. SPMD プログラミング・モデルの移植性と表現力を活用するには、複数のワーク項目を組み合わせてベクトル・ハードウェア命令を生成することを検討します。具体的には、ベクトル・ハードウェア命令が、単一のワーク項目から単独で生成されると考えるべきではありません。

2. （1）の結果として、ベクトルに対する演算（例えば加算）は、1 つのワーク項目の観点からは毎回、チャネルごと、または要素ごとに実行されるものと考える必要があります。ソースコードでベクトルを使用することは、通常、ベースとなるベクトル・ハードウェア命令を利用することとは無関係です。

3. ベクトルのアドレスを関数に渡すなど、特定の方法でコードを記述する場合、コンパイラーはベクトルと数学配列のメモリーレイアウト要件に従う必要があり、パフォーマンスに驚くほど影響する可能性があります。これを理解すると、コンパイラーが積極的に最適化できるコードを簡単に記述できるようになります。

明確なメンタルモデルがあればコードの記述はさらに容易になるため、最初の 2 つのポイントについてさらに詳しく説明します。

第 4 章と第 9 章で説明したように、ワーク項目は並列処理階層のリーフノードであり、カーネル関数の個々のインスタンスを表します。ワーク項目は任意の順序で実行できますが、ローカルまたはグローバルメモリーへのアトミックメモリー操作、またはグループ集合関数（`select_from_group`、`group_barrier` など）を使用する場合を除き、相互に通信または同期することはできません。

利便性タイプのインスタンスは単一のワーク項目に対してローカルであるため、ワーク項目ごとに `NumElements` のプライベート配列と同等であると考えることができます。例えば、`float4 y4` 宣言のストレージは、`float y4[4]` と同等とみなすことができます。図 11-7 の例について考えてみます。

```
h.parallel_for(8, [=](id<1> i) {
   float x = a[i];
   float4 y4 = b[i];
   a[i] = x + sycl::length(y4);
});
```

図 11-7. ベクトルの実行例

スカラー変数 x の場合、SIMD 命令を持つハードウェア（CPU、GPU など）上の複数のワーク項目を含むカーネル実行の結果では、ベクトルレジスターと SIMD 命令が使用される可能性がありますが、ベクトル化はワーク項目にまたがって行われ、コード内のどのベクトル型とも無関係です。

図 11-8 に示すように、独自のスカラー x を持つワーク業項目は、コンパイラーが生成する暗黙的な SIMD ハードウェア命令で異なるレーンを形成できます。ワーク項目内のスカラーデータは、一部の実装およびハードウェアでは、同時に実行されるワーク項目間で暗黙的にベクトル化（SIMD ハードウェア命令に結合される）されていると考えられますが、私たちが記述するワーク項目コードではこれをエンコードしません。これが SPMD スタイルのプログラミングの中核です。

ワーク項目 ID	w0	w1	w2	w3	w4	w5	w6	w7
SIMD ハードウェア命令レーン	x[w0]	x[w1]	x[w2]	x[w3]	x[w4]	x[w5]	x[w6]	x[w7]

図 11-8. スカラー変数 x から 8 ワイドのハードウェア・ベクトル命令への拡張が可能

ハードウェアに依存しない方法で潜在的な並列処理を公開することで、ベクトル・ハードウェア命令を備えたプラットフォームを含むさまざまなプラットフォームの機能に合わせて、アプリケーションをスケールアップ（またはスケールダウン）できるようになります。アプリケーション開発中に、ワーク項目と他の形式の並列処理で適切なバランスをとることは、私たち全員が取り組むべき課題であり、第 15 章、第 16 章、および第 17 章で詳しく説明します。

図 11-8 に示すように、コンパイラーによるスカラー変数 `x` からベクトル・ハードウェア命令への暗黙的なベクトル拡張により、コンパイラーは複数のワーク項目で行われるスカラー演算からハードウェアで SIMD 演算を生成します。

図 11-7 のコード例に戻ると、ベクトル変数 `y4` の場合、複数のワーク項目（たとえば、8 つのワーク項目）に対するカーネル実行の結果では、ハードウェアでのベクトル演算を使用して 4 要素ベクトルを処理しません。代わりに、各ワーク項目は独立してベクトル（この場合は `float4`）を参照し、そのベクトルの要素に対する操作は複数のクロックサイクル/命令で発生する可能性があります。
この様子を図 11-9 に示します。ワーク項目の観点から見ると、ベクトルはコンパイラーによってスカラー化されたものと考えることができます。

スカラー化された操作	実行サイクル	ワーク項目 ID							
		w0	**w1**	**w2**	**w3**	**w4**	**w5**	**w6**	**w7**
y4.x	**N**	y4[w0].	y4[w1].	y4[w2].	y4[w3].	y4[w4].	y4[w5].	y4[w6].	y4[w7].
y4.y	**N+1**	y4[w0].	y4[w1].	y4[w2].	y4[w3].	y4[w4].	y4[w5].	y4[w6].	y4[w7].
y4.z	**N+2**	y4[w0].	y4[w1].	y4[w2].	y4[w3].	y4[w4].	y4[w5].	y4[w6].	y4[w7].
y4.w	**N+3**	y4[w0].	y4[w1].	y4[w2].	y4[w3].	y4[w4].	y4[w5].	y4[w6].	y4[w7].

図 11-9. ベクトル・ハードウェア命令は SIMD レーンをまたがるストライドメモリー位置にアクセス

図 11-9 は、このセクションの 3 番目の重要なポイント、つまりベクトルの便宜的な解釈が、理解しておくべき重要なメモリーアクセスに影響する可能性があることを示しています。上記のコード例では、各ワーク項目は y4 の元の（連続した）データレイアウトを参照し、推測して調整する直感的なモデルを提供します。

パフォーマンスの観点から見ると、このワーク項目中心のベクトル・データ・レイアウトの欠点は、コンパイラーがワーク項目全体をベクトル化してベクトル・ハードウェア命令を生成すると、ベクトル・ハードウェア命令のレーンが連続したメモリー位置にアクセスできないことです。ベクトルデータのサイズと特定のデバイス機能に応じて、コンパイラーはギャザーまたはスキャッター・メモリー命令を生成する場合があります（図 11-10 を参照）。これは、ベクトルがメモリー内で連続しており、隣接するワーク項目が異なるベクトルで並列に動作している場合に必要です。ベクトル型が特定のデバイスの実行にどのように影響するかに関する詳細は、第 15 章と第 16 章を参照してください。また、ベンダーのドキュメント、コンパイラーの最適化レポートを確認し、ランタイム・プロファイルを使用して実行するシナリオの効率を理解してください。

```
q.submit([&](sycl::handler &h) { // サブグループのサイズは 8 と仮定
  // ...
  h.parallel_for(range<1>(8), [=](id<1> i) {
    // ...
    float4 y4 = b[i];     // i=0、1、2、...
    // ...
    float x = dowork(&y4);                // "dowork" は y4 を期待します。
                                          // 例えば、vec_y[8][4] レイアウト
  });
```

図 11-10. アドレスエスケープを使用したベクトルコードの例

コンパイラーが y4 のアドレスが現在のカーネルのワーク項目の範囲外にないことを証明できる場合、またはすべての呼び出し先関数がインライン展開されている場合、コンパイラーは積極的に最適化してパフォーマンスを向上させる可能性があります。例えば、コンパイラーが y4 のストレージを観測できない場合、合法的にその格納位置を入れ替えることができます。これにより、ギャザー命令やスキャッター命令を必要としない連続したメモリーアクセスが可能となります。コンパイラーの最適化レポートは、ソースコードがベクトル・ハードウェア命令にどのように変換されたかを示す情報を提供し、パフォーマンスを向上させるためコードを調整する方法に関するヒントを提供します。

一般的なガイドラインとして、これらの型を使用するコードは記述や保守がはるかに簡単であるため、論理的に意味がある場合は便宜的なベクトル（marray など）を使用する必要があります。アプリケーションでパフォーマンスのホットスポットが検出された場合にのみ、ソースコードのベクトル演算が最適ではないハードウェアの実装に低下しているか調査すべきです。

SIMD 型としてのベクトル

この章では、marray と vec は SIMD 型ではないことを強調しましたが、完全性を期すために、SIMD 型がベクトル・ハードウェアにどのようにマップされるか簡単に説明します。この説明は、SYCL ソースコード内のベクトルとは結びついていませんが、特定のデバイスタイプ（GPU、CPU、FPGA）について説明する本書の後の章で役立つ背景情報を提供し、SYCL の将来のバージョンで SIMD タイプが導入される可能性に備えるのに役立ちます。

SYCL デバイスでは、1 つのベクトルレジスターまたはレジスターファイルに含まれる複数のデータ値を操作する SIMD 命令ハードウェアがサポートされる場合があります。SIMD ハードウェアを提供するデバイスでは、図 11-11 に示すように、たとえば 8 要素のベクトルに対するベクトル加算演算を考えることができます。

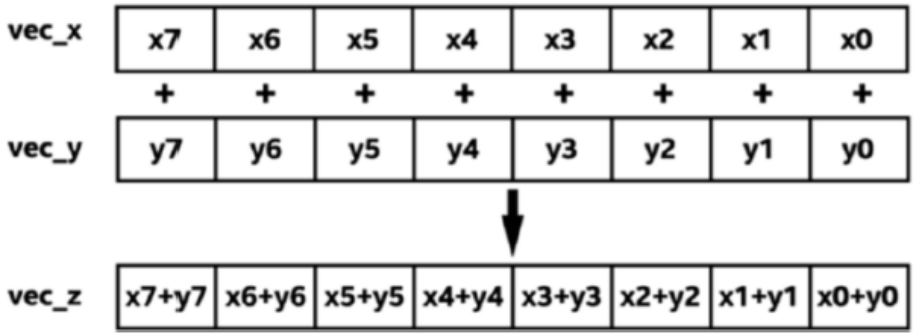

図 11-11. 8 ウェイのデータ並列処理による SIMD 加算

この例のベクトル加算は、ベクトル・ハードウェアを使用して 1 つの命令で実行でき、その SIMD 命令で並列にベクトルレジスター `vec_x` と `vec_y` を加算できます。

SIMD 型からベクトル・ハードウェアへのマッピングは非常に簡単で予測可能であり、どのコンパイラーでも同様の方法で実行される可能性があります。
これらの特性により、SIMD 型は SIMD ハードウェアでの低レベルのパフォーマンス・チューニングで非常に強力な武器となりますが、コードの移植性が低下し、特定のアーキテクチャーの機能に左右されるという欠点もあります。SPMD プログラミング・モデルは、これらの欠点に対処するために進化しました。

開発者が SIMD 型に予測可能なハードウェア・マッピングの特性があることを期待しているからこそ、2 つの異なる言語機能を使用してベクトルの 2 つの解釈を明確に分けることが重要です。開発者が便宜的なタイプを SIMD 型として動作することを想定して使用する場合、コンパイラーの最適化に反して作業することになり、期待または予想よりもパフォーマンスが低下する可能性があります。

まとめ

プログラミング言語では*ベクトル*という用語に複数の解釈があり、パフォーマンスが高くスケーラブルなコードを書くときには、特定の言語またはコンパイラーが構築されている解釈を理解することが重要です。SYCL は、ソースコード内のベクトル型はワーク項目にローカルな便利な型であり、ワーク項目全体にわたるコンパイラーによる暗黙的なベクトル化はハードウェア内の SIMD 命令にマップされることに基づいて構築されています。（非常にまれなケースですが）ベクトル・ハードウェアに直接明示的にマップするコードを記述する場合は、ベンダーのドキュメントを参照し、状況に応じて SYCL の拡張機能を参照してください。ほとんどのアプリケーションは、カーネルはワーク項目全体がベクトル化されることを前提として記述する必要があります。これにより、SPMD の強力な抽象化を活用できます。SPMD は、簡単に理解できるプログラミング・モデルを提供し、デバイスやアーキテクチャー全体でスケーラブルなパフォーマンスを実現します。

この章では、操作対象となる同型のデータのグループ化（複数のカラーチャネルを持つピクセルなど）がある場合にすぐに使用できる便利な機能である `marray` インターフェイスについて説明しました。さらに、特定のパターン（スウィズルを使用）や最適化（ロード/ストアおよびバックエンドの相互運用性を使用）を表現するのに便利なレガシー `vec` クラスについても説明しました。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 12 章　デバイス情報とカーネルの特殊化

この章では、プログラムをより柔軟にして、より移植性の高いものにする高度な概念について説明します。これは、作成したカーネルとコードの選択と、アプリケーションが実行される可能性のあるシステム（およびアクセラレーター）の機能に一致するメカニズムを検討することによって行われます。私たちは常に “デフォルト・アクセラレーターを使用する” だけで、それが何であるかに関係なく、その上で作成したカーネルを実行できるため、これは高度なトピックです。SYCL では、ホスト・アプリケーションを実行するのが CPU であっても、カーネルを実行できるデバイスが常に存在することを保証しているため、アクセラレーターがないシステムでもこれが機能することが分かっています。

“デフォルト・アクセラレーターを使用する” という汎用カーネルの枠を超えて進むと、使用するデバイスを選択するメカニズムと、より特化したカーネルを作成するメカニズムが利用できることが分かります。この章では両方の機能について説明します。これら 2 つの機能を組み合わせることで、実行されるシステムに高度に適応するアプリケーションを構築できます。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_12

幸いなことに、SYCL 仕様の作成者はこのニーズを考慮し、問題を解決するインターフェイスを提供しています。SYCL 仕様は、カーネルを実行できるデバイスをカプセル化するデバイスクラスを定義します。まず、デバイスクラスを照会する機能について説明します。これにより、プログラムはデバイスの特性と機能に適応できるようになります。場合によっては、異なるデバイスごとに異なるアルゴリズムを記述する必要もあります。この章の後半では、カーネルにアスペクトを適用してカーネルを特殊化し、コンパイラーがそれを利用できるようにできることを学習します。この特殊化により、カーネルは特定のクラスのデバイスに合わせてカスタマイズされるようになりますが、他のデバイスには適合しなくなる可能性が高くなります。これらの概念を組み合わせることで、必要に応じてプログラムを適応させることができます。 これにより、幅広い移植性を確保した上で、パフォーマンスを絞り出すためどれだけ投資を行うかを決定できるようになります。

GPU が存在するか?

私たちの多くは、プログラムの実行時に選択を行うため、“GPU が存在するか”を判断するロジックから始めます。これがこの章で扱うことの始まりです。これから説明するように、プログラムを堅牢かつ高性能にするのに役立つさらに多くの情報があります。

プログラムのパラメーター化は、正当性、機能の移植性、パフォーマンスの移植性、将来性の確保に役立ちます。

この章では、最も重要なクエリーと、それらをプログラムで効果的に使用する方法について詳しく説明します。実装では、クエリーできるより詳細なプロパティーが間違いなく提供されています。考えられるすべてのクエリーを学習するには、最新の SYCL 仕様、使用するコンパイラーのドキュメント、および遭遇する可能性のあるランタイム/ドライバーのドキュメントを確認する必要があります。

デバイス固有のプロパティーは、デバイス固有の*カーネル*および*ワークグループ*・プロパティーへのアクセスを含め、`get_info` 関数で照会できます。

カーネルコードをより規範的なものに改良

カーネルごとのコーディングは、次の 3 つのカテゴリーに分類されると考えると便利です:

- **汎用カーネルコード**: 特定のクラスのデバイスに限定されず、どこでも実行できます。
- **デバイスタイプ固有のカーネルコード**: 特定のデバイスタイプの*モデル*に合わせて調整されていない、デバイスタイプ (GPU、CPU、FPGA など) で実行します。これは、多くのデバイスタイプが共通の機能を共有しているため、すべてのデバイス向けに作成された汎用的なコードには適用されない仮定を行っても安全であるため、特に便利です。
- **調整されたデバイス固有のカーネルコード**: デバイスの特定のパラメーターに反応するチューニングを行ったデバイスタイプで実行します。これにより、少量のチューニングから詳細な最適化作業まで、幅広い可能性をカバーします。

異なるデバイスタイプに異なるパターンが必要になることを判断するのが、プログラマーとしての私たちの仕事です。第 14 章、第 15 章、第 16 章、および第 17 章では、この重要な考え方を明らかにします。

最も一般的なのは、汎用カーネルの機能を正しい実装で動作させることに重点を置くことから始めるます。第 2 章では、カーネル実装を開始するときにデバッグが最も容易である方法について具体的に説明しています。カーネルが動作するようになったら、特定のデバイスタイプまたはデバイスモデルの機能に対応させるために進化させることができます。

第 14 章では、デバイスを考慮する前に、まず並列性を考える思考の枠組みを示します。コードで示すのはパターン（アルゴリズムとも呼ばれます）の選択であり、さまざまなデバイスに異なるパターンが必要であるかを判断するのが、プログラマーとしての私たちの仕事です。第 15 章（GPU）、第 16 章（CPU）、および第 17 章（FPGA）では、これらのデバイスタイプを区別する特性と、使用するパターンを選択する理由についてさらに詳しく説明します。こうした特性こそが、デバイスの種類によって最適なアプローチ（パターンの選択）が異なる場合に、異なるバージョンのカーネルを作成することの動機となります。

特定の種類のデバイス（特定の CPU、GPU、FPGA など）向けに作成されたカーネルがある場合、それを特定のベンダーやデバイスのモデルに適合させるのは論理的です。優れたコーディング・スタイルは、機能（デバイスクエリーから見つかった項目サイズのサポートなど）に基づいてコードをパラメーター化することです。

マーケティング情報ではなく、デバイスの実際の機能を説明するクエリー・パラメーターを作成する必要があります。デバイスのモデル番号を照会してそれに反応するのは、プログラミングの習慣としては適切ではありません。このようなコードは将来性がなく、移植性が低くなります。

サポートするデバイスタイプごとに異なるカーネル（カーネルの GPU バージョン、カーネルの FPGA バージョン、および汎用バージョンのカーネル）を作成するのが一般的です。より具体的に、特定のデバイスベンダーやデバイスモデルをサポートする場合、カーネルを複製するのではなくパラメーター化できる場合メリットが得られます。 適切と思われる場合は、どちらでも自由に行うことができます。パラメーターの調整が多すぎるとコードが乱雑になり、読みにくくなったり、実行時に過度の負担がかかる可能性があります。しかし、一般的に、パラメーターはカーネルの 1 つのバージョンにうまく収まります。

パラメーター化は、アルゴリズムの基本構造がほぼ同じでも、特定のデバイスの機能に合わせて調整されている場合に最も有効です。まったく異なるアプローチ、パターン、またはアルゴリズムを使用する場合、異なるカーネルを記述する方がはるかにクリーンです。

デバイスと機能を列挙する方法

第 2 章 では、実行するデバイスを選択する 5 つの方法を示して説明しています。本質的に、方法 #1 は最も規範的でない *どこかで実行する* というものでしたが、デバイスファミリーから特定のデバイスをかなり正確なモデルを基に実行することを考慮し、最も規範的な方法 #5 に進化しました。その間にある方法 #2 から方法 #4 は、柔軟性と規範性を兼ね備えています。図 12-1、図 12-2、および図 12-4 は、デバイスの選択方法を示しています。

図 12-1 は、実装でデフォルトデバイスを選択できるようにした場合でも（第 2 章の方法 #1）、選択されたデバイスに関する情報を照会できることを示しています。

図 12-2 は、特定のデバイス（この場合は GPU）を使用してキューを設定する方法を示していますが、GPU が利用できない場合は明示的にデフォルトデバイスにフォールバックします。これにより、利用可能な GPU がある場合は、常にそれを優先して選択することで、デバイスの選択で一定の制御が可能となります。少なくとも 1 つのデバイスが存在することが保証されているため、カーネルは常に適切に構成されたシステムで実行できます。GPU がない場合、多くのシステムではデフォルトで CPU デバイスが使用されますが、その保証はありません。同様に、CPU デバイスを明示的に要求した場合、そのデバイスが存在する保証もあません（ただし、何らかのデバイスが存在することは保証されます）。

図 12-2 に示すソリューションはお勧めしません。図 12-2 は、不明瞭でエラーが発生しやすい印象を与えるだけでなく、実行時に 複数の GPU が選択可能である場合、どの GPU が選択されるか制御できません。有益で機能的であるにもかかわらず、より良い方法があります。次の例（図 12-4）に示すように、カスタム・デバイス・セレクターを作成することを推奨します。

```
queue q;

std::cout << "By default, we are running on "
          << q.get_device().get_info<info::device::name>()
          << "\n";
```

出力例（実行ごとに 1 行 – システムによって異なります）:

```
By default, we are running on NVIDIA GeForce RTX 3060
By default, we are running on AMD Radeon RX 5700 XT
By default, we are running on Intel(R) UHD Graphics 770
By default, we are running on Intel(R) Xeon(R) Gold 6336Y CPU @ 2.40GHz
By default, we are running on Intel(R) Data Center GPU Max 1100
```

図 12-1. *デフォルトで割り当てられているデバイス*

デバイスに関する問い合わせは、デバイスについて応答するためにインストールされたソフトウェア（特殊なユーザーレベル・ドライバ）に依存します。SYCL はこれに依存しています。これは、オペレーティング・システムがハードウェアへのアクセスにドライバーを必要とするのと同じです。ハードウェアをマシンにインストールするだけでは不十分です。

```
auto GPU_is_available = false;

try {
  device  testForGPU(gpu_selector_v);
  GPU_is_available = true;
} catch (exception const& ex) {
  std::cout << "Caught this SYCL exception: " << ex.what()
            << std::endl;
}

auto q = GPU_is_available ? queue(gpu_selector_v)
                          : queue(default_selector_v);

std::cout
       << "After checking for a GPU, we are running on:\n"
       << q.get_device().get_info<info::device::name>()
       << "\n";
```

4 つの出力例（4 つの異なるシステム、それぞれ GPU 搭載）：

```
After checking for a GPU, we are running on:
 AMD Radeon RX 5700 XT
After checking for a GPU, we are running on:
 Intel(R) Data Center GPU Max 1100
After checking for a GPU, we are running on:
 NVIDIA GeForce RTX 3060
After checking for a GPU, we are running on:
 Intel(R) UHD Graphics 770
```

出力例（GPU のないシステムを使用）：

```
Caught this SYCL exception: No device of
requested type 'info::device_type::gpu' available.
...(PI_ERROR_DEVICE_NOT_FOUND)
After checking for a GPU, we are running on:
 AMD Ryzen 5 3600 6-Core Processor
```

図 12-2. 可能な場合は try-catch を使用して GPU デバイスを選択し、そうでない場合はデフォルトのデバイスを使用します

アスペクト

SYCL 標準には、デバイスの機能を理解し、使用するデバイスを制御し、デバイスに送信するカーネルの制御に使用できるデバイス*アスペクト*のリストがあります。この章の最後では、“カーネルの特殊化” とカーネルのテンプレート化について説明します。ここでは、デバイスのクエリーと選択とそれらの使用方法を列挙します。図 12-3 は、SYCL を使用するすべての C++ プログラムで使用できるように SYCL 標準で定義されているアスペクトを示しています。アスペクトは常にブール値であるため、デバイスに必ずしもアスペクトが備わるとは限りません。最初の 4 つ (`cpu/gpu/accelerator/custom`) は、デバイスタイプが SYCL 2020 によって列挙型として定義されているため、相互に排他的です。`aspect::fp16`、`aspect::fp64`、`aspect::atomic64` などの機能は “オプション機能” であるため、すべてのデバイスでサポートされているとは限りません。堅牢性を求めるアプリケーションでは、これらのテストが特に重要になる場合があります。

標準アスペクト（すべてブール値）	デバイス …
aspect::cpu	CPU でコードを実行
aspect::gpu	GPU でコードを実行
aspect::accelerator	アクセラレーターでコードを実行
aspect::custom	固定関数のみ実行し、プログラム可能なカーネルはサポートされない
aspect::emulated	パフォーマンスのためではなく、エミュレーターでコードを実行します。通常はデバッグやプロファイルなどに使用されます
aspect::host_debuggable	標準的なデバッグを完全にサポートできます
aspect::fp16	sycl::half データ型をサポート
aspect::fp64	double データ型をサポート
aspect::atomic64	64 ビットアトミック操作をサポート
aspect::image	イメージをサポートしますが、本書ではこのトピックについては取り上げていません（代わりに、汎用的で移植性の高いバッファーに重点を置いています）
aspect::online_compiler aspect::online_linker	デバイスコードのオンラインコンパイルおよび/またはリンクをサポートします。 このようなデバイスは build()、compile()、link() 関数をサポートしている可能性がありますが、これらは本書では扱われていない高度なトピックです。
aspect::queue_profiling	キューをサポートする高度なトピックについては、第 13 章で他の実用的なヒントとともに少し説明されています。
aspect::usm_device_allocations aspect::usm_host_allocations aspect::usm_atomic_host_allocations aspect::usm_shared_allocations aspect::usm_atomic_shared_allocations	対応する USM 機能をサポート
aspect::usm_system_allocations	SYCL USM 割り当て呼び出しだけでなく、システム・アロケーターによって割り当てられたデータの共有をサポートします。このような使用法は移植性に影響を与え、パフォーマンスに影響を与える可能性があります

図 12-3. SYCL 標準で定義されているアスペクト（実装によりさらに追加可能）

カスタム・デバイス・セレクター

図 12-4 では、カスタム・デバイス・セレクターが使用されています。カスタム・デバイス・セレクターについては、コードの実行場所を選択する方法 #5 として、第 2 章で説明しました（図 2-16）。カスタム・デバイス・セレクターは、アプリケーションで使用可能な各デバイスを評価します。最高スコアを得たデバイスに基づいて特定のデバイスが選択されます（最高スコアが -1 のデバイスは選択されません）。この例では、セレクターを少し使ってみます:

- GPU 以外を拒否します（-1 を返します）。
- ベンダー名に “ACME” という単語が含まれる GPU を優先します（Martian の場合は 24、それ以外の場合は 824 を返します）。
- Martian 以外の GPU であればどれでも適切です（799 を返します）。
- ACME ではない Martian GPU は拒否されます（-1 を返します）。

次のセクション “興味を持ちましょう: `get_info<>`” では、`get_devices()`、`get_platforms()`、および `get_info<>` が提供する豊富な情報について詳しく説明します。これらのインターフェイスは、図 2-16 と図 12-4 に示す簡単なベンダー名チェックなど、デバイスの選択に利用できるあらゆる種類のロジックを可能にします。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int my_selector(const device& dev) {
    int score = -1;

    // 私たちは非 Martian GPU、特に ACME GPU を好みます
    if (dev.is_gpu()) {
      if (dev.get_info<info::device::vendor>().find("ACME") !=
          std::string::npos)
        score += 25;

      if (dev.get_info<info::device::vendor>().find(
            "Martian") == std::string::npos)
        score += 800;
     }

   // システムに GPU がない場合、すべてのデバイスに負のスコアが与えられ、
   // セレクターはデバイスを選択しません。これにより例外が発生します。
   return  score;
}

int main() {
    try {
       auto  q = queue{my_selector};
        std::cout
               << "After checking for a GPU, we are running on:\n "
               << q.get_device().get_info<info::device::name>()
               << "\n";
   } catch (exception const& ex) {
       std::cout << "Custom device selector did not select a "
                    "device.\n";
       std::cout << "Caught this SYCL exception: " << ex.what()
                 << std::endl;
    }

   return 0;
}
```

4 つの出力例 (4 つの異なるステム、それぞれ GPU 搭載)：

```
After checking for a GPU, we are running on:
 Intel(R) Gen9 HD Graphics NEO.
After checking for a GPU, we are running on:
 NVIDIA GeForce RTX 3060
After checking for a GPU, we are running on:
 Intel(R) Data Center GPU Max 1100
After checking for a GPU, we are running on:
 AMD Radeon RX 5700 XT
```

出力例 (GPU のないシステムを使用)：

```
After checking for a GPU, we are running on:
Custom device selector did not select a device.
Caught this SYCL exception: No device of requested
```

図 12-4. カスタム・デバイス・セレクター - 推奨されるソリューション

興味を持ちましょう: get_info<>

プログラムが実行時に利用可能デバイスを“知る”には、デバイスクラスから利用可能なデバイスを照会し、get_info<> を使用して特定のデバイスについて詳しい情報を取得できます。ここでは、これらのインターフェイスを使用して情報をダンプし、直接確認できるようにする、*興味深い*単純なプログラム（図 12-5 を参照）を示します。

これは、これらのインターフェイスを使用するプログラムを開発またはデバッグするときの健全性チェックに特に役立ちます。このプログラムが期待どおりに動作しない場合、必要なソフトウェア・ドライバーが正しくインストールされていないことが原因であることが多くあります。図 12-6 は、このプログラムの出力例と、存在するデバイスに関する概要情報が表示されています。

独自の“利用可能なすべての SYCl デバイスを一覧表示する”プログラムを作成する前に、システムが sycl-ls などのユーティリティーをサポートしていることを確認することをお勧めします。

```
// 利用可能なプラットフォームをループ
for (auto  const& this_platform :
        platform::get_platforms()) {
    std::cout
         << "Found platform: "
         << this_platform.get_info<info::platform::name>()
         << "\n";

   // このプラットフォームで利用可能なデバイスをループ
   for (auto  const& this_platform :
          this_platform.get_devices()) {
        std::cout
            << " Device: "
            << this_device.get_info<info::device::name>()
            << "\n";
   }
   std::cout << "\n";
}
```

図 12-5. デバイス・クエリー・メカニズムの簡単な使用: curious.cpp

```
% clang++ -fsycl fig_12_5_curious.cpp -o curious

% ./curious
Found platform: NVIDIA CUDA BACKEND
 Device: NVIDIA GeForce RTX 3060

Found platform: AMD HIP BACKEND
 Device: AMD Radeon RX 5700 XT

Found platform: Intel(R) OpenCL
  Device: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz

Found platform: Intel(R) OpenCL  HD Graphics
 Device: Intel(R) UHD Graphics P630 [0x3e96]

Found platform: Intel(R) Level-Zero
 Device: Intel(R) UHD Graphics P630 [0x3e96]

Found platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
 Device: Intel(R) FPGA Emulation Device
```

図 12-6. curious.cpp からの出力例

好奇心を持ちましょう: 詳細な列挙コード

`get_info` で入手できる詳細情報の一部を示すため、`verycurious.cpp`（図 12-7）プログラムを使用します。繰り返しますが、プログラムの開発やデバッグを行う際に、このようなコードを記述します。

情報にアクセスする方法を示したので、次は、アプリケーションでクエリーを実行し、アクションを起こす上で最も重要な情報フィールドについて説明します。

```
template <typename queryT, typename T>
void do_query(const & obj_to_query,
                const std::string& name, int indent = 4) {
   std::cout << std::string(indent, ' ') << name << " is '"
             << obj_to_query.template get_info<queryT>()
             << "'\n";
}

int main() {
   // この利用可能なプラットフォームをループ
   for (auto const& this_platform :
       platform::get_platforms()) {
     std::cout << "Found Platform:\n";
     do_query<info::platform::name>(this_platform,
                                    "info::platform::name");
    // クエリ情報は次のようになります（プログラムではこの図に示されているより
    // も多くの情報があります – 書籍の github を参照）

    // このプラットフォームで利用可能なデバイスをループ
    for (auto& dev : this_platform.get_devices()) {
      std::cout << " Device: "
                << dev.get_info<info::device::name>()
                << "\n";
    // is_cpu() == has(aspect::cpu)
    std::cout << "    is_cpu(): "
              << (dev.is_cpu() ? "Yes" : "No") << "\n";
    // is_cpu() == has(aspect::gpu)
    std::cout << "    is_gpu(): "
              << (dev.is_gpu() ? "Yes" : "No") << "\n";
    std::cout << "    has(fp16): "
               << (dev.has(aspect::fp16) ? "Yes" : "No")
              << "\n";
    // fig_12_7_very_curious.cpp にはさらに多くのクエリが
    // 表示されています。
    // ソースコードについては、本書の github を参照してください。
    }
    std::cout << "\n";
  }
  return 0;
}
```

図 12-7. デバイス・クエリー・メカニズムの詳しい使用法: verycurious.cpp（サブセットを表示）

非常に興味深い: get_info と has()

has() インターフェイスを使用すると、プログラムは図 12-3 にリストされているアスペクトを使用して機能を直接テストできます。簡単な使い方を図 12-7 に示します。さらに詳しい使用法は、GitHub の verycurious.cpp に完全なソースコードがあります。verycurious.cpp プログラムは、システム上のデバイスの詳細を確認するのに役立ちます。

デバイス情報記述子

この章の前半で使用した“興味深い”および“非常に興味深い”プログラム例では、一般的な SYCL デバイスクラスのメンバー関数（is_cpu、is_gpu、is_accelerator、get_info、has）を使用しています。これらのメンバー関数は、SYCL 仕様の“SYCL デバイスクラスのメンバー関数”の表に記載されています。

“興味深い”プログラムの例では、get_info メンバー関数を使用した情報の照会も行いました。すべての SYCL デバイスでサポートが必要なクエリーのセットがあります。このような項目の完全なリストは、SYCL 仕様の“デバイス情報記述子”という表に記載されています。

デバイス固有のカーネル情報記述子

プラットフォームやデバイスと同様に、get_info 関数を使用してカーネルに関する情報を照会できます。これらの情報（サポートされているワークグループのサイズ、推奨されるワークグループのサイズ、ワーク項目ごとに必要なプライベート・メモリー量など）はデバイスに固有である可能性があるため、カーネルクラスの get_info メンバー関数はデバイスを引数として受け入れます。

詳細: 正当性

ここでは、正当性に必須な情報と、チューニングには役立ちますが正当性には必須ではない情報に分けて説明します。

最初の正当性のカテゴリーでは、カーネルを適切に起動するために必須である条件を示します。ここで示すデバイスの制限に従わないと、プログラムが失敗することになります。図 12-8 は、これらのパラメーターをいくつか取得し、その値をホストコードとカーネルコード（ラムダのキャプチャー経由）で使用する方法を示しています。この情報を活用するためコードを変更できます。例えば、バッファーサイズやワークグループ・サイズを決定するのに役立ちます。

```
queue q;
device dev = q.get_device();

std::cout << "We are running on:\n"
          << dev.get_info<info::device::name>() << "\n";

// 次のようなクエリー結果を使用して、カーネル呼び出しが
// どの程度の大きさになるかを計算できます。
auto maxWG =
    dev.get_info<info::device::max_work_group_size>();
auto maxGmem =
    dev.get_info<info::device::global_mem_size>();
auto maxLmem =
    dev.get_info<info::device::local_mem_size>();

std::cout << "Max WG size is " << maxWG
          << "\nGlobal memory size is " << maxGmem
          << "\nLocal memory size is " << maxLmem << "\n";
```

図 12-8. カーネルを形成するために使用できるパラメーターを取得

必須条件（例: `sub_group_sizes`）に違反するカーネルを送信すると、ランタイムエラーが発生します。

デバイスクエリー

device_type: cpu、gpu、accelerator、custom[11]、automatic、all。これらは、is_cpu、is_gpu() などによって最も頻繁にテストされます（図 12-7 を参照）。

max_work_item_sizes: nd_range のワークグループのそれぞれの次元で許容されるワーク項目の最大数。最小値は（1、1、1）です。

max_work_group_size: 単一計算ユニットでカーネルを実行するワークグループで許容されるワーク項目の最大数。最小値は 1 です。

global_mem_size: グローバルメモリーのサイズ（バイト単位）。

local_mem_size: ローカルメモリーのサイズ（バイト単位）。最大サイズは 32K。

max_compute_units: デバイスで利用可能な並列処理の量を示します。実装定義であるため、慎重に解釈してください。

sub_group_sizes: デバイスでサポートされるサブグループ・サイズのセットを返します。

USM 機能など、さらに多くの特性がアスペクトとしてエンコードされることに注意してください（図 12-3 を参照）。

[11] 本書ではカスタムデバイスについては説明しません（“カスタムデバイス” と “カスタム・デバイス・セレクター” を混同しないでください）。*カスタム*タイプを使用して自身を識別するデバイスをプログラミングする場合、そのデバイスのドキュメントを調べて詳細を理解する必要があります。簡単に言えば、カスタムデバイスは珍しく奇妙なので、ここでは取り上げません。ここで説明する機能の一部にカスタムデバイスが課す可能性がある制限については、意図的に無視しています。

プログラムロジックでは MAX_COMPUTE_UNITS の使用を避けることを強く推奨します。

計算ユニットの最大数を問い合わせることは、定義が明確ではないため、コードのチューニングに役立たないこともあり、避けるべきであることがわかりました。`max_compute_units` を使用する代わりに、ほとんどのプログラムでは並列処理を表現し、ランタイムがそれを利用可能な並列処理にマッピングできるようにする必要があります。正確性のため `max_compute_units` に依存することは、実装およびデバイス固有の情報で補足された場合にのみ意味を持ちます。専門家はそうするかもしれませんが、ほとんどの開発者はそうしませんし、そうする必要もありません。この場合は、ランタイムに任せましょう。

カーネルの照会

第 10 章の “カーネルバンドル内のカーネル” で説明したメカニズムは、以下のカーネルクエリーを実行するために必要です:

`work_group_size`: 特定のデバイスでカーネルを実行する際に使用できる最大ワークグループ・サイズを返します

`compile_work_group_size`: 適用可能であれば、カーネルで指定されたワークグループのサイズを返します。そうでない場合は `(0, 0, 0)` を返します

`compile_sub_group_size`: 適用可能であれば、カーネルで指定されたサブグループのサイズを返します。そうでない場合は `0` を返します

`compile_num_sub_groups`: 適用可能であれば、カーネルで指定されたサブグループの数を返します。そうでない場合は `0` を返します

`max_sub_group_size`: 指定されたワークグループ・サイズで起動されたカーネルの最大サブグループ・サイズを返します

`max_num_sub_groups`: カーネルのサブグループの最大数を返します

詳細: チューニング/最適化

カーネルの微調整（ファイン・チューニング）パラメーターとして考慮できる追加パラメーターがいくつかあります。これらはプログラムの正確性を損なうことはないため、無視できます。これにより、カーネルはパフォーマンスのためにハードウェアの詳細を利用できるようになります。

これらのクエリーの結果に注意を払うことは、キャッシュ（存在する場合）のチューニングに役立ちます。

デバイスクエリー

`global_mem_cache_line_size`: グローバルメモリーのキャッシュラインのサイズ（バイト単位）。

`global_mem_cache_size`: グローバルメモリーのキャッシュのサイズ（バイト単位）。

`local_mem_type`: サポートされるローカルメモリーのタイプを返します。これは、SRAM などの専用ローカル・メモリー・ストレージを意味する `info::local_mem_type::local` または `info::local_mem_type::global` のいずれかです。
後者のタイプは、ローカルメモリーがグローバルメモリー上の抽象化として実装されているだけであり、パフォーマンスの向上は見込めません。

カーネルの照会

preferred_work_group_size：特定のデバイス上でカーネルを実行する推奨ワークグループ・サイズ。

preferred_work_group_size_multiple：特定のデバイス上でカーネルを実行して最高のパフォーマンスを得るには、ワークグループ・サイズをこの値（preferred_work_group_size_multiple）の倍数にする必要があります。値は work_group_size より大きくてはなりません。

実行時プロパティーとコンパイル時プロパティー

実装によっては、コンパイル時の定数/マクロ、またはそのほかの機能が提供される場合がありますが、これらは標準ではないため、本書ではその使用は推奨していません（説明もしていません）。この章で説明するクエリーはランタイム API（get_info）を使用して行われるため、結果は実行時までわかりません。次のセクションでは、属性を使用してカーネルのコンパイルを制御する方法について説明します。SYCL 標準では、属性以外では、かなり難解な例外が 1 つありますが、ランタイム情報の使用のみが推奨されています。SYCL には、コンパイル時にアプリケーションがアスペクトを照会するのに使用できる 2 つの特性があります。これらの特性は、いずれのデバイスでもサポートされないデバイス機能のテンプレート化されたカーネルのインスタンス化を回避するのに有用です。これは非常に高度であり、あまり使用されない機能であるため、本書では詳しく説明しません。SYCL 標準には、“デバイスのアスペクト” セクションの最後に、この目的で any_device_has_v<aspect> と all_devices_have_v<aspect> を使用する例があります。この標準では “特殊化定数” も定義されていますが、これは通常、ライブラリーなど高度なターゲット開発で使用されるため、本書では説明しません。実験的なコンパイル時のプロパティー拡張については、エピローグの “コンパイル時プロパティー” で説明します。

カーネルの特殊化

異なる用途ごとに異なるカーネルを用意することでカーネルを特殊化し、ターゲットデバイスのアスペクト（図 12-3 を参照）に基づいて適切なカーネルを選択できます。もちろん、特殊なカーネルを明示的に記述し、C++ テンプレートを使用することもできます。SYCL 属性（図 12-9）とアスペクト（図 12-3）を使用して、カーネルに特定の機能を持たせることをコンパイラーに通知できます。

たとえば、`reqd_work_group_size` 属性（図 12-9）は、カーネルに特定のワークグループ・サイズを要求するのに使用でき、`device_has` 属性はカーネルに特定のデバイスアスペクトを要求するために使用できます。

属性を使用すると、次の 2 つの点で役立ちます:

1. リストのアスペクトを持たないデバイスにカーネルが送信されると、例外がスローされます。

2. カーネル（またはカーネルが呼び出す関数）が、属性にリストされていないアスペクトに関連付けられたオプション機能（`fp16` など）を使用すると、コンパイラーは診断を行います。

最初は、アプリケーションが失敗する可能性が高い場合に進行を止めるのに役立ち、2 つ目はコンパイル時にエラーをキャッチするのに役立ちます。これらの理由から、属性を使用することが有用である場合があります。

図 12-10 は、実行時ロジックを使用して 2 つのコードシーケンスを選択し、属性を使用してカーネルの 1 つを特殊化する例を示しています。

標準属性	説明
`device_has(aspect, ...)`	この属性は、カーネル関数を装飾する機能（すべての属性）に加えて、非カーネル関数の装飾に使用できる唯一の属性です。 要件: カーネルは、図 12-3 の指定されたアスペクトを満たすデバイスでのみ起動されます。

`reqd_work_group_size(dim0)`
`reqd_work_group_size(dim0, dim1)`
`reqd_work_group_size(dim0, dim1, dim2)`
	要件: カーネルはワークグループ・サイズで起動する必要があります。

`work_group_size_hint(dim0)`
`work_group_size_hint(dim0, dim1)`
`work_group_size_hint(dim0, dim1, dim2)`
	ヒント: カーネルは指定されたワークグループ・サイズで起動される可能性が高くなります。

`reqd_sub_group_size(dim)`	要件: カーネルは指定されたサブグループ サイズでコンパイルおよび実行される必要があります。

図 12-9. SYCL 標準で定義された属性（非推奨ではない）

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  queue q;

  constexpr int size = 16;
  std::array<double, size> data;

  // sycl::device_has() を属性として使用しても、
  // 選択するデバイスには影響しません。したがって、ホストコードは、
  // その属性を必要とするカーネルを送信する前に、
  // デバイスのアスペクトをチェックする必要があります。
  if (q.get_device().has(aspect::fp64)) {
     buffer B{data};
     q.submit([&](handler& h) {
        accessor A{B, h};
        // ここでの属性は、カーネルが fp64 サポートを要求することが
        // 許可されていることを示しています。図 12-3 の任意の属性を指定できます。
        // 上記の namespace ステートメント (C++ の場合) は属性に影響を
        // 与えないことに注意してください (/C++ の癖)。
        // そのため、ここでは sycl:: が必要です。
        h.parallel_for(
             size, [=](auto& idx)
                       [[sycl::device_has(aspect::fp64)]] {
                          A[idx] = idx * 2.0;
                       });
      });
      std::cout << "doubles were used\n";
    } else {
      // ここでは、コードの柔軟性を高め、より移植性を高めるため、
      // 代替方法 (デバイス上での二重演算サポートを必要としない)
      // を使用します。
      std::array<float, size> fdata;

         buffer B{fdata};
         q.submit([&](handler& h) {
            accessor A{B, h};
            h.parallel_for(
                size, [=](auto& dx) { A[idx] = idx * 2.0f; });
          }
       }

      for (int i = 0; i < size; i++) data[i] = fdata[i];

      std::cout << "no doubles used\n";
   }
   for (int i = 0; i < size; i++)
       std::out << "data[" << i << "] = " << data[i] << "\n";
   return 0;
}
```

図 12-10. 属性のより明示的にカーネルを特殊化

まとめ

最も移植性の高いプログラムは、システムで使用可能なデバイスを照会し、実行時情報に基づいてその動作を調整します。この章では、実行時に存在するハードウェアに合わせてコードをカスタマイズするために利用できる豊富な情報を紹介しました。また、投資する価値があると判断した場合、カーネルを特定のデバイスタイプにさらに適合させることができるよう、カーネルを特化するさまざまな方法についても議論しました。これらにより、SYCL で C++ を使用する範囲内で、ニーズを満たす必要に応じて移植性とパフォーマンスのバランスをとるツールが提供されます。

アプリケーションをパラメーター化してハードウェアの特性に合わせることで、プログラムの機能的移植性、パフォーマンス移植性、将来性が向上します。また、存在するハードウェアがプログラム設計時の範囲内に収まっているかテストし、想定の範囲外にあるハードウェアが見つかった場合には、警告を発したり中止することもできます。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 13 章　実用的なヒント

この章には、SYCL を使用した C++ プログラミングに役立つことが証明されている、有用な情報、実用的なヒント、アドバイス、テクニックが多数記載されています。これらのトピックはいずれも、すべてを網羅するわけではないため、みなさんの意識を高め、必要に応じてさらに学習を促すことを目的としています。

サンプルコードとコンパイラーの入手

第 1 章では、SYCL コンパイラーの入手方法（例: `oneapi.com/implementations`、`github.com/intel/llvm`）と、本書で使用されているサンプルコードの入手先（`github.com/Apress/data-parallel-CPP`）について説明しています。ここでもう一度触れるのは、実践的な経験を積むため例を試すこと（変更を加えることを含む）がいかに有益であるか示すためです。図 1-1 のコードが実際に何を出力するか、すでに理解してる人たちにみなさんも加わってください。

オンラインリソース

主なオンラインリソースには以下があります:

- `sycl.tech/` の豊富なリソース
- SYCL の公式ホームページは khronos.org/sycl/ にあり、リソースは khronos.org/sycl/resources で提供されています
- tinyurl.com/cuda2sycl にある、CUDA から SYCL を使用した C++ への移行に役立つリソース
- 移行ツール GitHub ホームページ（github.com/oneapi-src/SYCLomatic）

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_13

プラットフォーム・モデル

SYCL をサポートする C++ コンパイラーは、従来の C++ コンパイラーと同じように動作するように設計されています。SYCL をサポートするコンパイラーが、ホスト（CPU など）およびデバイス用のコードを生成する高レベルの内部動作を理解することには価値があります。

SYCL で使用されるプラットフォーム・モデル（図 13-1）は、デバイスで実行される計算ワークを調整および制御するホストを示しています。第 2 章ではデバイスにワークを割り当てる方法を、第 4 章ではデバイスをプログラムする方法について詳しく説明しています。第 12 章では、特定レベルでプラットフォーム・モデルを使用する方法について説明しています。

第 2 章で説明したように、適切に構成された SYCL ランタイムと互換性のあるハードウェアを使用するシステムでは、実行デバイスが常に存在します。これにより、少なくとも 1 つのデバイスが利用可能であると想定してデバイスコードを記述できるようになります。デバイスコードを実行するデバイスの選択は、プログラムによって制御されます。つまり、特定のデバイスでコードを実行する、またどのように実行するかは、完全にプログラマーの選択です（デバイス選択オプションについては、第 12 章で説明しています）。

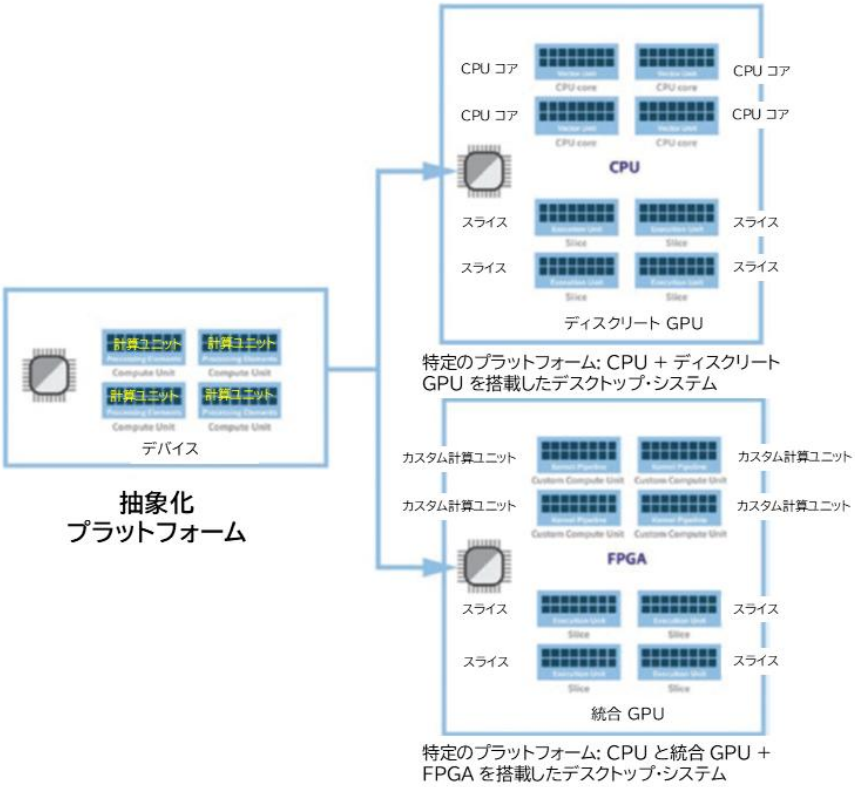

図 13-1. プラットフォーム・モデル: 抽象的にも具体的にも使用可能

複数アーキテクチャー・バイナリー

本書の目標は、異機種マシンをサポートする単一のソースコードを記述することであるため、結果として単一実行可能ファイルの生成を望むのは当然です。

複数アーキテクチャー・バイナリー（別名 *ファットバイナリー*）は、異機種マシンに必要なすべてのコンパイル済みコードと中間コードを含むように拡張された単一のバイナリーファイルです。

複数アーキテクチャー・バイナリーは、使い慣れている `a.out` や `a.exe` と同じように動作しますが、異機種マシンに必要なものがすべて含まれています。これにより、特定のデバイスで実行する適切なコードを選択するプロセスを自動化できます。次に説明するように、ファットバイナリー内のデバイスコードの 1 つの形式は、デバイス命令の最終的な生成を実行時まで延期する中間形式です。

コンパイルモデル

SYCL の単一ソースの性質により、通常の C++ コンパイルのように動作し、感じられます。デバイスに対して追加のパスを呼び出したり、デバイスとホストコードのバンドルを処理する必要はありません。これらはすべてコンパイラーによって自動的に行われます。もちろん、何が起こっているのか理解することは、いくつかの理由から重要なことです。これは、特定のアーキテクチャーをより効率良くターゲットにする場合に役立つ知識であり、コンパイル過程で発生する障害をデバッグする必要があるか理解することも重要です。

知識が必要になったときに備えて、コンパイルモデルを見直します。コンパイルモデルは、ホストと、場合によっては複数のデバイスで同時に実行されるコードをサポートするため、コンパイラー、リンカー、および他のサポートツールを実行するコマンドは、使い慣れた C++ コンパイル（1 つのアーキテクチャーのみを対象とする）よりも複雑になります。まずは異種世界へようこそ！

この異種の複雑性はコンパイラーによって意図的に隠匿されており、“そのまま動作する” ようになっています。

コンパイラーは、従来の C++ コンパイラー（*事前* - AOT コンパイル、オフライン・カーネル・コンパイルとも呼ばれます）と同様に、ターゲット固有の実行可能コードを生成することも、実行時に特定のターゲットに*ジャストインタイム*（JIT）コンパイルできる中間表現を生成することもできます。

カーネルは、“事前”（AOT）または “ジャストイン”（JIT）コンパイルされます。

コンパイラーは、デバイスターゲットが事前に判明している場合（プログラムをコンパイルする時点）にのみ、事前にコンパイルできます。JIT コンパイルでは、コンパイルされるプログラムの移植性は向上しますが、アプリケーションの実行中にコンパイラーとランタイムが追加の処理を実行する必要があります。

GPU を含むほとんどのデバイスでは、JIT コンパイルが最も一般的な方法です。一部のデバイス（FPGA など）ではコンパイルプロセスが非常に低速であるため、AOT コンパイルが一般的です。

AOT コードを使用する必要がある（例: FPGA）か、それを使用するメリットがあることがわかっている場合を除き、JIT を使用します。

デフォルトでは、デバイス用にコードをコンパイルすると、デバイスコードの出力は中間形式で保存されます。実行時に、システム上のデバイスドライバーは、システムで使用可能なデバイスと一致するように、中間形式を実行コードに*ジャストインタイム*でコンパイルします。

AOT コードとは異なり、JIT コードは、実行時にコンパイルして、システム上の任意のデバイスを使用できるようにします。これには、プログラムがファットバイナリーにコンパイルされたときには存在しなかったデバイスが含まれる場合もあります。

特定のデバイスまたはデバイスクラス向けに、事前にコンパイルするようコンパイラーに要求できます。これには実行時間を節約できる利点がありますが、コンパイル時間が増加し、バイナリーが大きくなる欠点もあります。事前コンパイルされたコードは、実行時に利用可能なハードウェアに合わせて調整できないため、ジャストインタイムほど移植性がありません。

AOT と JIT の両方の利点を得るために、バイナリーに両方を含めることもできます。

移植性を最大限に高めるには、一部が AOT コードを含む場合でも、バイナリーに JIT コードも含めます。

事前に特定のデバイス用にコンパイルすることで、ビルド時にプログラムがそのデバイスで動作するか確認するのにも役立ちます。ジャストインタイム・コンパイルでは、プログラムの実行時にコンパイルに失敗する可能性があります（これは、第 5 章のメカニズムで検出できます）。この章の次の “デバッグ” の節では、これに関するデバッグのヒントがいくつか紹介されています。また、第 5 章では、実行時にこれらのエラーをキャッチしてアプリケーションのアボートを回避する方法も詳しく説明しています。

図 13-2 は、ソースコードからファットバイナリー（実行可能ファイル）を生成するコンパイルプロセスを示しています。どの組み合わせを選択しても、ファットバイナリーに結合されます。ファットバイナリーは、アプリケーションの実行時にランタイムによって使用されます（ホスト上で実行されるバイナリー）。状況によっては、特定のデバイスのデバイスコードを別にコンパイルしたいことがあります。このような個別のコンパイルの結果は、最終的にファットバイナリーに結合する必要があります。これは、完全なコンパイル（完全な合成配置配線）の時間が非常に長くなる FPGA 開発に非常に役立ち、実際に、ランタイムシステムに合成ツールのインストールを必要としない FPGA 開発の要件です。
図 13-3 は、このようなニーズに対応するバンドル/アンバンドルのフローを示しています。すべてを一度にコンパイルするオプションは常にありますが、開発中はコンパイルを分割する方が効果的です。

SYCL をサポートするすべての C++ コンパイラーには、同じ目標を持つコンパイルモデルがありますが実装の詳細は異なります。ここで示されている図は、DPC++ コンパイラー・ツールチェーンの実装者から提供されたものです。

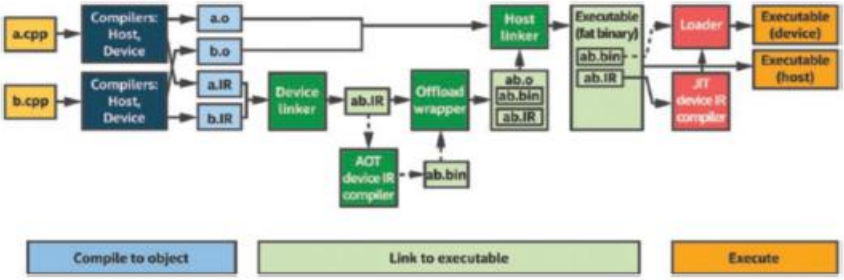

図 13-2. コンパイルプロセス:事前およびジャストインタイムのオプション

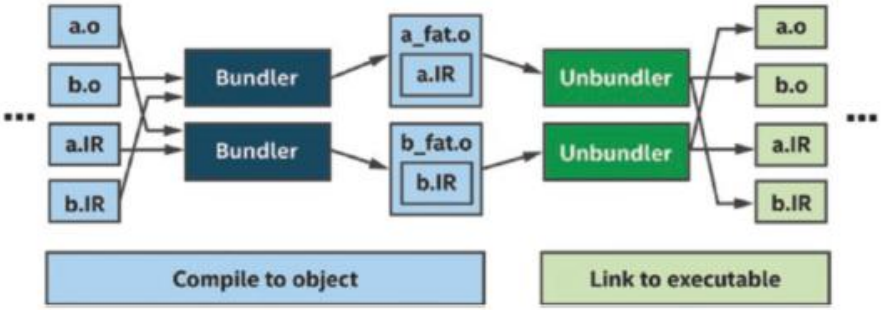

図 13-3. コンパイルプロセス: オフロードのバンドラー/アンバンドラー

コンテキスト: 知っておくべき重要なこと

第 6 章で述べたように、コンテキストはカーネルを実行できるデバイスまたはデバイスのセットを表します。コンテキストは、ランタイムが実行中の処理状態を保存するのに便利な場所であると考えることができます。ほとんどの SYCL プログラムでは、プログラマーがコンテキストを渡す以外、コンテキストと直接やり取りすることはありません。

デバイスはサブデバイスに分割できます。これは問題を分割するのに役立ちます。サブデバイスはデバイスと同じように扱われるため (同じ C++ 型)、デバイスのグループ化に関する説明は、すべてサブデバイスにも当てはまります。

SYCL は、デバイスをプラットフォームにグループ化することを抽象的に考慮します。プラットフォーム内では、デバイスはメモリーの共有などにより相互作用できる場合があります。同じコンテキストに属するデバイスは、何らかのメカニズムを使用して互いのグローバルメモリーにアクセスできる必要があります。SYCL USM メモリー (第 6 章) は、デバイスが同じコンテキストにある場合にのみ、デバイス間で共有できます。 USM メモリー割り当ては、デバイスではなくコンテキストにバインドされているため、1 つのコンテキスト内の USM 割り当ては他のコンテキストからアクセスできません。したがって、USM 割り当ては、単一のコンテキスト (デバイスのサブセットの可能性あり) 内での使用に制限されます。

コンテキストは、ハードウェアがサポートできないものを抽象化しません。たとえば、互いにメモリーを共有できない 2 つの GPU を含むコンテキストを作成することはできません。同じプラットフォームから公開されるすべてのデバイスを、同じコンテキストでグループ化できる必要はありません。

キューを作成する際に、キューを配置するコンテキストを指定できます。デフォルトでは、DPC++ コンパイラー・プロジェクトはプラットフォームごとにデフォルトのコンテキストを実装し、新しいキューをデフォルトのコンテキストに自動的に割り当てます。他の SYCL コンパイラーも同様に行うことができますが、標準ではその必要はありません。

コンテキストの作成にはコストがかかるため、コンテキストが少ないほどアプリケーションの効率が向上します。

特定のプラットフォームのすべてのデバイスを常に同じコンテキストに配置すると、次の 2 つの利点があります: (1) コンテキストの作成にはコストがかかるため、アプリケーションはより効率的です。(2) ハードウェアでサポートされている最大の共有が許可されます (例: USM)。

既存の C++ プログラムへの SYCL の追加

既存の C++ プログラムに適切な並列処理を追加することが、SYCL を使用する最初のステップです。C++ アプリケーションがすでに並列実行を行っている場合、それは利点でもあり、頭痛の種となる場合もあります。これは、アプリケーションのワークを並列実行に分割する方法が、アプリケーションの機能に大きく影響するためです。プログラマーがプログラムの並列性のリファクタリングについて話すとき、それはプログラム内の実行フローとデータを再配置し、並列処理を活用できる状態に整えることを指します。これは複雑なトピックであるため、ここでは簡単に触れるだけにします。並列処理向けにアプリケーションを準備する方法について、*万能の答え*はありませんが、注目に値するヒントがいくつかあります。

C++ アプリケーションに並列処理を追加する場合、検討すべき簡単なアプローチは、並列処理の可能性が最も高いプログラム内の独立したポイントを見つけることです。そこから修正を開始し、必要に応じて他の領域に並列処理を追加することもできます。複雑な要因として、リファクタリング（つまり、プログラムフローの再配置とデータ構造の再設計）が、並列処理の可能性を高める場合があります。

プログラム内で並列処理の可能性が最も高い独立したポイントを見つけたら、そのポイントで SYCL をどのように使用するか検討します。それが本書の残りの部分で説明されていることです。

大まかに言えば、並列処理を導入する主な手順は次のとおりです:

1. 並行性による安全性（従来の CPU プログラミングでは一般にスレッドセーフと呼ばれます）: すべての共有可変データ（変更可能で同時に操作される可能性のあるデータ）の使用を調整して、データ競合を防ぎます。第 19 章を参照してください。

2. 並行性および並列性を導入します。

3. 並列処理のチューニング（最適なスケーリング、スループットまたはレイテンシーの最適化）。

まずステップ #1 を検討します。多くのアプリケーションはすでに並行性のためにリファクタリングされていますが、そうでないアプリケーションも数多くあります。SYCL を唯一の並列処理手段として、カーネル内で使用され、ホストと共有されるデータの安全性に焦点を当てます。プログラムに並列処理を導入する他のテクニック (OpenMP、MPI、TBB など) を使用している場合、それは SYCL プログラミングに加えて追加の懸念事項となります。1 つのプログラム内で複数の手法を使用しても問題ないことを確認することが重要です。SYCL がプログラム内の唯一の並列処理手段である必要はありません。本書では、他の並列処理技術と組み合わせる高度なトピックについては触れていません。

複数のコンパイラーを使用する場合の考慮事項

SYCL をサポートする C++ コンパイラーは、他の C++ コンパイラーが生成したオブジェクト・コード (ライブラリー、オブジェクト・ファイルなど) とのリンクもサポートします。一般に、複数のコンパイラーを使用することで生じる問題は、C++ コンパイラーと同じであり、名前修飾、同じ標準ライブラリーのターゲット設定、呼び出し規約への準拠、などを考慮する必要があります。これらは、Fortran や C などの他の言語のコンパイラーを組み合わせて使用する場合に対処される問題と同じです。

さらに、アプリケーションは、プログラムのビルドに使用されるコンパイラーが提供する SYCL ランタイムを使用する必要があります。SYCL コンパイラーと SYCL ランタイムを混在させるのは安全でありません。異なるランタイムでは、重要な SYCL オブジェクトの実装とデータレイアウトが異なる場合があります。

SYCL と非 SYCL ソース言語との相互運用性は、OpenCL、C、CUDA など他のプログラミング言語で記述されたカーネル関数またはデバイス関数を操作したり、別のコンパイラーでコンパイルされた中間表現のコードを使用する SYCL の機能を指します。非 SYCL ソース言語との相互運用性の詳細については、第 20 章を参照してください。

最後に、SYCL デバイスコードのコンパイルに使用されたコンパイラー・ツールチェーンは、コンパイルのリンクフェーズにも必要です。異なるコンパイラー・ツールチェーンのリンカーでリンクすると、SYCL に対応していないコンパイラーはホストコードとデバイスコードを適切に統合できないため、機能するプログラムを生成できません。

デバッグ

このセクションでは、並列プログラム、特に異種マシンを対象とするプログラムのデバッグに特有の課題を軽減する、デバッグに関する簡単なアドバイスを紹介します。

アプリケーションが CPU デバイス上で実行されている間、デバッグを行う選択肢があることを決して忘れてはなりません。このデバッグのヒントについては、第 2 章の「方法 #2」で説明しています。デバイスのアーキテクチャーには汎用 CPU よりもデバッグの観察点が少ないことが多いため、ツールは CPU 上のコードをより正確に調査できます。 すべてを CPU 上で実行する場合との違いは、同期に関連する多くのエラーが解消される点であり、これにはホストとデバイス間のメモリーの移動も含まれます。最終的にはこうしたエラーをすべてデバッグする必要がありますが、これにより段階的なデバッグが可能となり、一部のバグを他のバグより先に解決できます。

経験上、ターゲットデバイスで可能な限り頻繁に実行することが重要であることがわかります。また、デバッグプロセスの一環として CPU（および他のデバイス）への移植性を活用することも同様に重要です。複数のデバイスで実行して問題点を明らかにすることで、発生したバグがデバイス固有であるか特定するのに役立ちます。

CPU 上で実行されるデバッグのヒントは強力なデバッグツールです。

並列プログラミングのエラー、特にデータ競合とデッドロックは、通常、すべてのコードをホスト上で実行することで検出し、排除しやすくなります。
残念なことですが、ホストとデバイスを組み合わせで実行する場合、このような並列プログラミング・エラーによるプログラムの障害が最も頻繁に発生します。このような問題が発生した場合、CPU のみに戻すことが強力なデバッグツールであることを覚えておくとは、非常に有用です。幸いなことに、SYCL はこのオプションをいつでも利用できるように、また簡単にアクセスできるように設計されています。

プログラムがデッドロックした場合のデバッグのヒント: ホストアクセサーが適切に破棄されていること、およびカーネル内のワーク項目が SYCL 仕様の同期ルールに従っていることを確認してください。

デバッグを始めるときは、次のコンパイラー・オプションの使用をお勧めします:

- `-g`: デバッグ情報を生成します
- `-ferror-limit=1`: SYCL で頻繁に使用されるテンプレート・ライブラリーで C++ の健全性を維持します
- `-Werror -Wall -Wpedantic`: コンパイラーに適切なコーディングを強制させ、実行時にデバッグすべき誤ったコードの生成を回避します

SYCL を使用した C++ ためだけに、警告の修正に時間を費やす必要はないため、`-Wpedantic` を使用しないのは理解できます。

実行時にコードをジャストインタイムでコンパイルできるようにすることで、調査できるコードもあります。これはコンパイラーが使用するレイヤーに大きく依存するため、コンパイラーのドキュメントを参照して推奨事項を確認することをお勧めします。

デッドロックや他の同期の問題のデバッグ

並列プログラミングは、並行して行われるワーク間の調整に依存します。データの使用は、データが利用可能になるタイミングで制御される必要があります。このようなデータの依存関係は、適切な動作を実現するためプログラムのロジックに組み込まれる必要があります。

同期/依存関係のロジックでエラーが発生した場合、特に USM での依存関係の問題のデバッグは困難になる可能性があります。プログラムがハングアップする（完了しない）か、誤った情報が断続的に生成されるでしょう。このような場合、“デバッガーで実行するまでは失敗するが、デバッガーで実行すると完璧に動作する” などの動作が見られることがあります。 このような断続的な障害は、待機処理やロック、キュー送信間の明示的な依存関係など、適切に同期されていない依存関係に起因することが多くあります。

有用なデバッグ手法としては、以下があります:

- インオーダー・キューからアウトオブオーダー・キューへの切り替え
- `queue.wait()` の呼び出し

デバッグ中にこれらを使用すると、依存関係情報が欠落している可能性のある場所を特定するのに役立ちます。このような変更によりプログラム障害が変化または解決した場合、同期/依存関係ロジックに修正すべき問題があることを示しています。修正後は、これらの一時的なデバッグ対策は削除します。

カーネルコードのデバッグ

カーネルコードをデバッグするには、まず CPU デバイスで実行します（第 2 章で推奨されるように）。第 2 章のデバイスセレクターのコードは、デバッグ時にワークをホストデバイスにリダイレクトするランタイムオプション、またはコンパイラー時にオプションを受け入れるように変更できます。

カーネルコードをデバッグする場合、SYCL はカーネル内で使用できる C++ スタイルのストリームを定義します（図 13-4）。DPC++ コンパイラーは、制限付きですが、便利な機能を備えた C スタイルの `printf` の実験的な実装も提供します。

```
q.submit([&](handler &h) {
   stream  out(1024, 256, h);
   h.parallel_for(range{8}, [=](id<1> idx) {
      out << "Testing my sycl stream (this is work-item ID:"
          << idx << ")\n";
   });
});
```

図 13-4. sycl::stream

カーネルコードをデバッグするには、経験上、ブレークポイントを `parallel_for` の前または `parallel_for` 内に配置し、`parallel_for` 上には配置しないことが推奨されます。`parallel_for` に設定されたブレークポイントは、次の操作を実行した後でもブレークポイントを複数回トリガーする可能性があります。この C++ デバッグのアドバイスは、SYCL のような多くのテンプレート拡張にも適用されます。テンプレート呼び出しに設定されたブレークポイントがコンパイラーによって拡張されると、複雑な一連のブレークポイント群に変換されます。

実装によってこれを緩和する方法はあるかもしれませんが、ここでの重要な点は、`parallel_for` 自体にブレークポイントを設定しないことで、すべての実装で混乱を回避できることです。

ランタイム機能のデバッグ

ジャストインタイム・コンパイル中にランタイムエラーが発生した場合、以下のいずれかのケースが考えられます:

1. 利用可能なハードウェアがサポートできない機能を明示的に使用したケース（例: `fp16` や `simd8`）
2. コンパイラー/ランタイムのバグ
3. 誤ったコードを記述し、ランタイムで問題が発生するまで検出されず、理解困難なランタイム・エラー・メッセージが生成されるケース

これら 3 つのケースすべてにおいて、バグを詳しく調べるのは気が滅入るかもしれません。ありがたいことに、ざっと見ただけでも、問題の原因を正確に把握できる場合があります。これにより、問題を回避する知識が得られ、コンパイラー・チームにバグレポートを送るのに役立つかもしれません。いずれにしても、有益なツールがあることを知るのは重要です。

実行エラーを示すプログラムの出力は次の例のようになります:

```
terminate called after throwing an instance of
'sycl::_V1::runtime_error'
   what(): Native API failed.Native API returns: ...

      or

terminate called after throwing an instance of
'sycl::_V1::compile_program_error'
   what(): The program was built for 1 devices
…
```

```
error: Kernel compiled with required subgroup size 8, which
is unsupported on this platform
in kernel: 'typeinfo name for main::'lambda'(sycl::_V1::nd_
item<2>)'
error: backend compiler failed build.
  -11 (PI_ERROR_BUILD_PROGRAM_FAILURE)
```

このような例外を見ると、ホストプログラムがエラーをキャッチするように構築されている可能性があることがわかります。最初のエラーは、ネイティブでサポートされない API にアクセスした場合に発生する、包括的なエラーを示しています（この場合は、プラットフォームでサポートされていないホスト側のメモリー割り当てが使用されています）。2 番目のエラーは、`SIMD8` がサポートされていないデバイスに指定されたことを簡単に認識できます（この場合は、代わりに `SIMD16` がサポートされています）。ランタイム・コンパイラーの障害によってアプリケーションをアボートする必要はありません。障害をキャッチしたり、障害を回避するコードを記述したり、あるいはその両方を行えます。第 5 章でこのトピックについて詳しく説明します。

実行時に障害が発生し、すぐにデバッグするのが困難である場合は、事前（AOT）コンパイルにより再構築を試してみる価値があります。ターゲットデバイス向けに事前コンパイルが可能である場合、これを簡単に試すことができ、分かり易い診断結果が得られる可能性があります。エラーが JIT やランタイムではなくコンパイル時に生成された場合、JIT やランタイムから出力される少量のエラー情報ではなく、コンパイラーからのエラーメッセージには、有用な情報が見つかることがあります。

図 13-5 には、高度なデバッグを支援するコンパイラーまたはランタイムでサポートされる 2 つのフラグと環境変数（Windows および Linux でサポート）がリストされています。これらは、コンパイルモデルを検査および制御するために存在する、DPC++ コンパイラー固有の高度なデバッグオプションです。これらについては本書では説明もなく、使用もされませんが、GitHub プロジェクトのオンラインで詳細が説明されています（intel.github.io/llvm-docs/EnvironmentVariables.html および tinyurl.com/IGCoptions）。

環境変数	値	説明
ONEAPI_DEVICE_SELECTOR	intel.github.io のドキュメントにあるオプションの例については、オンライン・ドキュメントを参照してください。	SYCL を使用するアプリケーションの実行時に使用するデバイスの選択肢を制限するために使用できます。デバイスを特定のタイプ（GPU やアクセラレーター）またはバックエンド（レベルゼロや OpenCL）に制限するのに役立ちます。
SYCL_PI_TRACE	1 （基本）、 2 （高度）、 -1 （すべて）	ランタイム: 値が 1 の場合、プラグインおよびデバイスを検出するランタイム・プラグイン・インターフェイス（PI）のトレースが有効になります。値が 2 の場合、すべての PI 呼び出しのトレースが有効になります。値が -1 の場合、すべてのレベルのトレースが実行されます。
SYCL_PRINT_EXECUTION_GRAPH	always （または、before_addCG、after_addCG、before_addCopyBack、after_addCopyBack、before_addHostAcc、after_addHostAcc を指定して、選択したファイルのみをダンプするように要求します）	ランタイム: 実行グラフをトレースする（DOT 拡張機能付き）を作成します。実行時に何が発生したか、その痕跡を比較的簡単に参照できます。
IGC_ShaderDumpEnable	0 または 1	Linux のみ。ランタイム: インテル・グラフィックス・コンパイラー（JIT）に一部の情報をダンプするよう指示します。
IGC_ShaderDumpEnableAll	0 または 1	Linux のみ。ランタイム: インテル・グラフィックス・コンパイラー（JIT）に最大の情報をダンプするよう指示します。

図 13-5. DPC++コンパイラーの高度なデバッグオプション

これらのオプションについては本書では詳しく説明しませんが、必要に応じて高度なデバッグを行う手段として触れておきます。これらのオプションにより、問題やバグを回避する情報が得られる場合があります。ソースコードが問題を引き起こしている可能性があり、それはソースコードを修正することで解決できる可能性があります。それ以外では、これらのオプションはコンパイラー自体の高度なデバッグに使用されます。 つまり、それらはユーザーよりもコンパイラーの開発者を対象としています。一部の熟練ユーザーにとっては、これらのオプションが有用であると考えられるため、ここではそれらを紹介しますが、詳しくは説明しません。詳細は、DPC++ コンパイラー GitHub プロジェクト（intel.github.io/llvm-docs/EnvironmentVariables.html）を参照してください。

デバッグのヒント: 他の手段が尽きて、実行時の問題をデバッグする必要がある場合、原因の手がかりを与えてくれる可能性のあるダンプツールを探します。

キューのプロファイルと結果のタイミング機能

多くのデバイスは*キューのプロファイル*をサポートしています（`device::has(aspect::queue_profiling)`）— アスペクト全般の詳細については、第 12 章を参照してください。シンプルで強力なインターフェイスにより、キューの送信、デバイス上での実行開始、デバイス上での完了、コマンドの完了などに関する詳細なタイミング情報に簡単にアクセスできます。このプロファイルでは、通常、ホストとデバイス間のデータ転送時間が含まれないため、ホストのタイミングメカニズム（chrono など）を使用するよりも、デバイスのタイミングが正確になります。図 13-6 と図 13-7 の例と、図 13-8 のサンプル出力を参照してください。図 13-8 に示すサンプル出力は、この手法で何ができるか示していますが、最適化されていないため、特定のシステムを選択するメリットを表すものとして使用してはなりません。

`aspect::queue_profiling` アスペクトは、デバイスが `property::queue::enable_profiling` を介したキューのプロファイルをサポートすることを示します。このようなデバイスでは、キューを構築するときに `property::queue::enable_profiling` を指定できます。プロパティーのリストは、キュー・コンストラクター・オプションの最終パラメーターです。
これにより、そのキューに送信されたコマンドグループのプロファイル情報の SYCL ランタイム のキャプチャーがアクティブになります。キャプチャーされた情報は、SYCL `event` クラスの `get_profiling_info` メンバー関数を介して利用できます。キューに関連付けられたデバイスに `aspect::queue_profiling` がない場合、コンストラクターは `errc::feature_not_supported` エラーコードを含む同期例外をスローします。

`event` クラスの `get_profiling_info` メンバー関数を使用して、`info::event_profiling` に列挙されているプロファイル `info` パラメーターの 1 つを指定し、イベントのプロファイル情報を照会できます。それぞれの `info` パラメーターに設定可能な値と制限は、イベントに関連する SYCL バックエンドの仕様で定義されています。`info::event_profiling` のすべての `info` パラメーターは、SYCL 仕様の "SYCL イベントクラスのプロファイル情報記述子" という表に示されており、`info::event_profiling` の概要は、仕様の付録の "イベント情報記述子" に記載されています。

それぞれのプロファイル記述子は、実装で定義されたタイムベースからナノ秒単位の経過時間を示すタイムスタンプを返します。同じバックエンドを共有するすべてのイベントは、同じタイムベースを共有することが保証されています。従って、同じバックエンドからの 2 つのタイムスタンプの差は、それらのイベント間で経過したナノ秒単位の数を生成します。

最後に、イベントのプロファイルを有効にするとオーバーヘッドが増加するため、開発中またはチューニング中に有効にして、運用時には無効にすることに注意してください。

ヒント: 若干のオーバーヘッドがあるため、キューのプロファイルは開発時またはチューニング時にのみ有効にし、運用環境では無効にしてください。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

// この例の配列タイプとデータサイズ
constexpr size_t array_size = (1 << 16);
typedef std::array<int, array_size> IntArray;
// VectorAdd を定義 (図 13-7 参照)

void InitializeArray(IntArray &a) {
   for (size_t i = 0; i < a.size(); i++) a[i] = i;
}

int main() {
   IntArray a, b, sum;
   InitializeArray(a);
   InitializeArray(b);

   queue q(property::queue::enable_profiling{});

   std::cout << "Vector size: " << a.size()
             << "\nRunning on device: "
             << q.get_device().get_info<info::device::name>()
             << "\n";

   VectorAdd(q, a, b, sum);

   return 0;
}
```

図 13-6. キューのプロファイルに使用する設定

```
void VectorAdd(queue &q, const IntArray &a,
               const IntArray &b, IntArray &sum) {
   range<1> num_items{a.size()};
   buffer a_buf(a), b_buf(b);
   buffer sum_buf(sum.data(), num_items);
   auto t1 =
        std::chrono::steady_clock::now();     // 開始タイミング

   event e = q.submit([&](handler &h) {
     auto a_acc = a_buf.get_access<access::mode::read>(h);
     auto b_acc = b_buf.get_access<access::mode::read>(h);
     auto sum_acc =
         sum_buf.get_access<access::mode::write>(h);

     h.parallel_for(num_items, [=](id<1> i) {
        sum_acc[i] = a_acc[i] + b_acc[i];
     });
  });
  q.wait();

  double timeA =
     (e.template get_profiling_info<
          info::event_profiling::command_end>() -
      e.template get_profiling_info<
          info::event_profiling::command_start>());

  auto t2 =
       std::chrono::steady_clock::now();     // 停止タイミング

  double timeB = (std::chrono::duration_cast<
                     std::chrono::microseconds>(t2 - t1)
                     .count());

  std::cout
      << "profiling: Vector add completed on device in "
      << timeA << " nanoseconds\n";
  std::cout << "chrono: Vector add completed on device in "
            << timeB * 1000 << " nanoseconds\n";
  std::cout << "chrono more than profiling by "
            << (timeB * 1000 - timeA) << " nanoseconds\n";
}
```

図 13-7. キューのプロファイルを使用

```
Vector size: 65536
Running on device: Intel(R) UHD Graphics P630 [0x3e96] profiling: Vector
add completed on device in 57602 nanoseconds chrono: Vector add completed
on device in 2.85489e+08 nanoseconds chrono more than profiling by
2.85431e+08 nanoseconds

Vector size: 65536
Running on device: NVIDIA GeForce RTX 3060
profiling: Vector add completed on device in 17410 nanoseconds chrono:
Vector add completed on device in 3.6071e+07 nanoseconds chrono more than
profiling by 3.60536e+07 nanoseconds

Vector size: 65536
Running on device: Intel(R) Data Center GPU Max 1100 profiling: Vector add
completed on device in 9440 nanoseconds
chrono: Vector add completed on device in 5.6976e+07 nanoseconds chrono
more than profiling by 5.69666e+07 nanoseconds
```

図 13-8. キューのプロファイル例からの 3 つのサンプル出力

トレースおよびプロファイル・ツールのインターフェイス

トレースおよびプロファイル・ツールは、アプリケーション実行時の動作を理解するのに役立ち、多くの場合、アルゴリズムの改善点を明らかにします。提供される情報は多くの場合汎用性が高く、幅広い種類のデバイスに適用できるため、使用するプラットフォームで最も有用と思われるトレースおよびプロファイル・ツールを使用することをお勧めします。もちろん、プラットフォームを微調整するには、実際にそのプラットフォーム上で作業を行う必要があります。最大限の移植性を目指すアプリケーションでは、まず調整を行う際に、あらゆる変更を可能な限り移植性の高いものにすることを推奨します。

SYCL プログラムが OpenCL ランタイム上で実行され、OpenCL バックエンドを使用している場合、OpenCL インターセプト・レイヤー（github.com/intel/opencl-intercept-layer）を使用してプログラムを実行できます。これは、アプリケーション（または上位レベルのランタイム）が生成する OpenCL コマンドを検査、ログに記録、および変更できるツールです。多くのコントロールをサポートしていますが、最初に設定するのに適したものは `ErrorLogging`、`BuildLogging`、そして `CallLogging` です（ただし、大量の出力が生成されます）。`DumpProgramSPIRV` を使用すると、有用なダンプが生成できます。

OpenCL インターセプト・レイヤーは独立したユーティリティーであり、特定の OpenCL 実装の一部ではないため、多くの SYCL コンパイラーでも動作します。

SYCL 開発者に人気のある、パフォーマンス・データの収集に優れたツールが他にも多数あります。これらはオープンソース（github.com/intel/pti-gpu）であり、始めるのに役立つサンプルも付属しています。

最も人気のあるツールは次の 2 つです:

- onetrace: DPC++（CPU と GPU の両方）と OpenMP GPU オフロードをサポートする OpenCL およびレベルゼロ・バックエンド用のホストおよびデバイス・トレース・ツール

- oneprof: DPC++ および OpenMP GPU オフロードをサポートする OpenCL およびレベルゼロ・バックエンド用の GPU HW メトリック収集ツール

どちらのツールも、インストルメントされたランタイムからの情報を使用するため、GPU と CPU に適用されます。これらのランタイムを使用するコンパイラーの SYCL、ISPC、および OpenMP サポートはすべてこれらのツールの恩恵を受けます。ツールのウェブサイトを参照して、使用する目的への適用性を調べてください。一般に、すべてのターゲット・プラットフォームがサポートされる保証がない場合でも、サポートされるプラットフォームを検出して、ツールからプログラムに関する有用な情報を得ることができます。プログラムについて学んだことの多くは、どこでも役に立ちます。

データの初期化とカーネル出力へのアクセス

ここでは、SYCL の入門ユーザーを混乱させ、そして遭遇する最も一般的な（私たちの経験上）最初のバグにつながるトピックについて詳しく説明します。

簡単に言えば、ホストメモリー割り当て（配列やベクトルなど）からバッファーを作成すると、バッファーが破棄されるまでホスト割り当てに直接アクセスすることができません。バッファーは、存続期間中、構築時に渡されたすべてのホスト割り当てを所有します。バッファーが存続している間でもホスト割り当てにアクセスできるメカニズムはほとんどありません（例: バッファー・ミューテックス）。しかし、こうした高度な機能は、ここで述べた初期段階のバグの解決には役立ちません。

誰もがこのエラーを犯します。このエラーを知っておくと、長い間悩むよりも早くデバッグできます。

ホストメモリー割り当てからバッファーを構築する場合、バッファーが破棄されるまでホスト割り当てに直接アクセスしてはなりません。バッファーが有効である間は、バッファーが割り当てを所有します。バッファーのスコープとスコープ内のルールを理解しましょう。

バッファーが割り当てを所有している間に、ホストプログラムがホスト割り当てにアクセスすると、一般的なバグが発生します。これが発生すると、バッファーが割り当てを何に使用しているか不明であるため、予測は不可能になります。データが正しくない可能もあります。出力を読み取ろうとしているカーネルがまだ実行を開始していない可能性があります。第 3 章と第 8 章で説明したように、SYCL は非同期タスクグラフのメカニズムによって構築されています。タスクグラフ操作からの出力データを使用する前に、グラフが実行され、ホストでデータを使用できるコードの同期ポイントに到達していることを確認する必要があります。バッファーの破棄とホストアクセサーの作成はどちらも、この同期を引き起こす操作です。

図 13-9 は、よく記述する一般的なコードパターンを示しています。ここでは、バッファーが定義されているブロックスコープを閉じることによって、バッファーが破棄されます。バッファーをスコープ外にして破棄することで、バッファー・コンストラクターに渡された元のホスト割り当てを通じてカーネルの結果を安全に読み取ることができます。

```
constexpr size_t N = 1024;

// 利用可能なデバイスでキューを設定
queue q;

// ホスト上で初期化するためのホストコンテナーを作成
std::vector<int> in_vec(N), out_vec(N);

// 入力と出力ベクトルを初期
for (int i = 0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// ニュアンス: バッファーを簡単にスコープ外にして破棄できるように、
// 新しいスコープを作成します。
{
   // ホスト割り当て (この場合は vector) を使用して
   // バッファーを作成
   buffer in_buf{in_vec}, out_buf{out_vec};

   // カーネルをキューに送信
   q.submit([&](handler&  h) {
     accessor in{in_buf, h};
     accessor out{out_buf, h};

     h.parallel_for(
         range{N}, [=](id<1> idx) { out[idx] = in[idx]; });
     });

     // バッファーが有効なスコープを閉じます。
     // バッファーの破棄を引き起こします。
     // バッファーへのカーネルの書き込みが完了するまで待機し、
     // 書き込まれたバッファーのデータをホストの割り当て
     // (この場合 std::vectors) にコピーします。
     // このスコープの終了によりバッファー・デストラクターが
     // 実行されると、元の in_vec と out_vec に
     // 再度アクセスできるようになります。
}

// すべての出力が期待される値であるか確認
// 警告:  // ホストコードで in_vec と out_vec を再び安全に使用するには、
// バッファー・デストラクターが実行されている必要があります。
// バッファーが有効である間はそれらの割り当てを所有しており、
// それらを使用するのは安全ではありません。  // 少なくとも最新ではない値が含まれます。
// このコードは安全かつ正しいものです。なぜなら、上記のスコープが
// 閉じられたことにより、ベクトルを再度使用するこのポイントの前に
// ベクトルを再度使用するこのポイントの前にバッファーが破棄されているからです
for (int i = 0; i < N; i++)
   std::cout << "out_vec[" << i << "]=" << out_vec[i]
             << "\n";
```

図 13-9. 一般的なパターン: ホスト割り当てからバッファーを作成

図 13-9 のように、バッファーを既存のホストメモリーに関連付ける理由は 2 つあります:

1. バッファー内のデータの初期化を簡素化します。私たち（またはアプリケーションの別の部分）がすでに初期化したホストメモリーからバッファーを構築できます。
2. ‘`}`’でスコープを閉じると、バッファーへの `host_accessor` を作成するより簡潔になります。

ホスト割り当てを使用してカーネルからの出力値をダンプまたは検証する場合、バッファー割り当てをブロックスコープ（またはその他のスコープ）に配置して、バッファーが破棄されるタイミングを制御する必要があります。カーネル出力を取得するため、ホスト割り当てにアクセスする前にバッファーが破棄されていることを確認する必要があります。図 13-9 は、これが正しく実行されたことを示していますが、図 13-10 は、バッファーがまだ有効である間に出力がアクセスされる一般的なバグを示しています。

熟練プログラマーは、バッファー破棄を使用してカーネルから、ホストメモリー割り当てに結果データを返すことを好むことがあります。しかし、ほとんどのプログラマー（特に入門者）には、スコープ付きのホストアクセサーを使用することを推奨します。

```
constexpr size_t N = 1024;

// 利用可能なデバイスでキューを設定
queue q;

// ホスト上で初期化するためのホストコンテナーを作成
std::vector<int> in_vec(N), out_vec(N);

// 入力と出力ベクトルを初期
for (int i = 0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// ホスト割り当て (この場合は vector) を使用して
// バッファーを作成
buffer in_buf{in_vec}, out_buf{out_vec};

// カーネルをキューに送信
q.submit([&](handler& h) {
   accessor in{in_buf, h};
   accessor out{out_buf, h};

   h.parallel_for(range{N},
                  [=](id<1> idx) { out[idx] = in[idx]; });
});

// バグ!! ホスト割り当て out_vec を使用していますが、バッファー out_buf は
// まだ有効であり、その割り当てを所有しています。
// カーネルはまだ実行されていない可能性があり、
// カーネルが実行されたとしてもバッファーには
// 出力をホストにコピーし戻す理由がないため、
// おそらく初期化値 (ゼロ) が出力されます。
for (int i = 0; i < N; i++)
   std::cout << "out_vec[" << i << "]=" << out_vec[i]
             << "\n";
```

図 13-10. よくあるバグ: バッファーの存続期間中にホスト割り当てから直接データを読み取る

このようなバグを回避するには、SYCL を使用した C++ では、バッファースコープではなくホストアクセサーを使用することを推奨します。ホストアクセサーはホストからバッファーへのアクセスを提供し、コンストラクターの実行が終了すると、バッファーへの以前の書き込み (たとえば、host_accessor が作成される前に送信されたカーネルからの書き込み) が実行され、参照可能になることが保証されます。

本書では、両方のスタイル（つまり、ホストアクセサーと、バッファー・コンストラクターに渡されるホスト割り当て）を組み合わせて使用し、両方のスタイルを活用できるようにします。ホストアクセサーを使用すると、実行開始時にエラーが発生する可能性が低くなります。図 13-11 は、バッファーを破棄することなく、ホストアクセサーを使用してカーネルから出力を読み取る方法を示しています。

```
constexpr size_t N = 1024;

// 利用可能なデバイスでキューを設定
queue q;

// ホスト上で初期化するためのホストコンテナーを作成
std::vector<int> in_vec(N), out_vec(N);

// 入力と出力ベクトルを初期
for (int i = 0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// ホスト割り当て (この場合は vector) を使用して
// バッファーを作成
buffer in_buf{in_vec}, out_buf{out_vec};

// カーネルをキューに送信
  q.submit([&](handler&  h) {
     accessor in{in_buf, h};
     accessor out{out_buf, h};

     h.parallel_for(range{N},
                    [=](id<1> idx) { out[idx] = in[idx]; });
});

// すべての出力が期待される値であるか確認
// ホストアクセサーを使用! バッファーはまだスコープ内にあり/有効
host_accessor A{out_buf};

for (int i = 0; i < N; i++)
   std::cout << "A[" << i << "]=" << A[i] << "\n";
```

図 13-11. 推奨事項: ホストアクセサーを使用してカーネルの結果を読み取る

ホストアクセサーは、バッファーが有効であるときはいつでも使用できます。たとえば、バッファーの存続期間の前後で、バッファーの内容の初期化やカーネルからの結果の読み取りに使用できます。図 13-12 にこのパターンの例を示します。

```
constexpr size_t N = 1024;

// 利用可能なデバイスでキューを設定
queue q;

// サイズ N のバッファーを作成
buffer<int> in_buf{N}, out_buf{N};

// ホストアクセサーを使用してデータを初期化
{  // クリティカル: host_accessor の有効期間のスコープの始まり
  host_accessor in_acc{in_buf}, out_acc{out_buf};
  for (int i = 0; i < N; i++) {
       in_acc[i] = i;
       out_acc[i] = 0;
  }
}  // クリティカル: スコープを閉じて、ホスト アクセサーを
   //スコープ外にします。

// カーネルをキューに送信
q.submit([&](handler& h) {
   accessor in{in_buf, h};
   accessor out{out_buf, h};

   h.parallel_for(range{N},
                  [=](id<1> idx) { out[idx] = in[idx]; });
});

// すべての出力が期待される値であるか確認
// ホストアクセサーを使用! バッファーはまだスコープ内にあり/有効
host_accessor A{out_buf};

for (int i = 0; i < N; i++)
   std::cout << "A[" << i << "]=" << A[i] << "\n";
```

図 13-12. 推奨事項: バッファーの初期化と結果の読み取りにはホストアクセサーを使用

最後に触れておくべきことは、ホストアクセサーも存続期間を持つため、アプリケーションにおいて逆のバグを引き起こすことがあることです。バッファーへの `host_accessor` が有効な間、ランタイムはいずれのデバイスにもバッファーの使用を許可しません。ランタイムはホストプログラムを分析して、ホストアクセサーにアクセスするタイミングを判断しないため、ホストプログラムがバッファーへのアクセスを完了したことを知る唯一の方法は、`host_accessor` デストラクターを実行することです。図 13-13 に示すように、ホストプログラムがカーネルの実行を待機している場合（`queue::wait()` または別のホストアクセサーの取得など）、および SYCL ランタイムがバッファーを使用するカーネルの実行前に以前のホストアクセサーが破棄されるのを待機している場合、アプリケーションがハングしているように見えることがあります。

ホストアクセサーを使用する場合、カーネルまたは他のホストアクセサーによるバッファーの使用をロック解除する必要がなくなったときに、ホストアクセサーが破棄されることを確認してください。

```
constexpr size_t N = 1024;

// 利用可能なデバイスでキューを設定
queue q;

// ホスト割り当て (この場合は vector) を使用してバッファーを作成
buffer<int> in_buf{N}, out_buf{N};

// ホストアクセサーを使用してデータを初期化
host_accessor in_acc{in_buf}, out_acc{out_buf};
for (int i = 0; i < N; i++) {
   in_acc[i] = i;
   out_acc[i] = 0;
}

// バグ!! ホストアクセサー in_acc および out_acc はまだ有効です。
// ランタイムはホストアクセサー経由でバッファーへのアクセスが完了したことを認識しないため、
// その後の q.submits はデバイス上で開始されません。
// ホストがバッファの更新を完了するまでデバイスカーネルは起動できません。
// これは、ホストが最初にアクセスを獲得したためです (キューの送信前)。
// このプログラムはハングしたように見えます。その場合はデバッガーを使用してください。

// カーネルをキューに送信
q.submit([&](handler& h) {
   accessor in{in_buf, h};
   accessor out{out_buf, h};

   h.parallel_for(range{N},
                  [=](id<1> idx) { out[idx] = in[idx]; });
});

std::cout << "This program will deadlock here!!! Our "
            "host_accessors used\n"
         << " for data initialization are still in "
            "scope, so the runtime won't\n"
         << " allow our kernel to start executing on "
            "the device (the host could\n"
         << " still be initializing the data that is "
            "used by the kernel). The next line\n"
         << " of code is acquiring a host accessor for "
            "the output, which will wait for\n"
         << " the kernel to run first. Since in_acc "
            "and out_acc have not been\n"
         << " destructed, the kernel is not safe for "
            "the runtime to run, and we deadlock.\n";

// すべての出力が期待される値であるか確認
// ホストアクセサーを使用! バッファーはまだスコープ内にあり/有効
host_accessor A{out_buf};

for (int i = 0; i < N; i++)
   std::cout << "A[" << i << "]=" << A[i] << "\n";
```

図 13-13. 不適切な host_accessor によるデッドロック (バグ - ハングアップ!)

複数の翻訳単位

異なる翻訳単位で定義されるカーネル内の関数を呼び出す場合、それらの関数には SYCL_EXTERNAL ラベルが必要です。
この修飾がない場合、コンパイラーはデバイスコード外で使用する関数のみをコンパイルします（デバイスコード内からその外部関数を呼び出すことは違法になります）。

SYCL_EXTERNAL 関数には、同じ翻訳単位内で関数を定義する場合には適用されない次の制限があります:

- SYCL_EXTERNAL は関数でのみ使用できます。
- SYCL_EXTERNAL 関数では、生のポインターをパラメーターまたは戻り値の型として使用できません。代わりに明示的なポインタークラスを使用します。
- SYCL_EXTERNAL 関数は parallel_for_work_item メソッドを呼び出すことができません。
- SYCL_EXTERNAL 関数は、parallel_for_work_group スコープ内から呼び出すことはできません。

同じ翻訳単位内になく、SYCL_EXTERNAL で宣言されていない関数を呼び出すカーネルをコンパイルすると、次のようなコンパイルエラーが発生する可能性があります。

```
error: SYCL kernel cannot call an undefined function without
SYCL_EXTERNAL attribute
```

関数自体が SYCL_EXTERNAL 属性なしでコンパイルされている場合、次のようなリンクまたは実行時エラーが発生する可能性があります。

```
terminate called after throwing an instance of '...compile_
program_error'...
error: undefined reference to ...
```

SYCL では、コンパイラーが `SYCL_EXTERNAL` をサポートする必要はありません。これは通常*オプション*の機能です。DPC++ は `SYCL_EXTERNAL` をサポートします。

複数の翻訳単位のパフォーマンスへの影響

コンパイルモデル（この章の前半を参照）の意味は、デバイスコードを複数の翻訳単位に分散すると、デバイスコードが同じ場所に配置されている場合よりも、ジャストインタイム・コンパイルの呼び出しが頻繁に発生する可能性があることです。これは実装に大きく依存しており、実装が成熟するにつれて時の経過とともに変わる可能性があります。

このようなパフォーマンスへの影響は、開発作業のほとんどでは無視できるほど小さいものですが、コードのパフォーマンスを最大化するため微調整を行う場合、これらの影響を軽減するできることが 2 つあります:

（1） デバイスコードを同じ翻訳単位にまとめ、
（2） 事前コンパイルしてジャストインタイム・コンパイルの影響を完全に回避します。

これらは両方とも開発者の努力を必要とするため、開発が完了し、アプリケーションのパフォーマンスを最大限に引き出そうとする場合にのみこれを行います。この詳細なチューニングに頼る場合、変更をテストして、使用する SYCL 実装への影響を確認する価値があります。

匿名ラムダ式に名前が必要な場合

SYCL では、ツールが必要とする場合や、デバッグ目的（ユーザー定義名の表示を有効にするなど）に備えて、ラムダ式に名前を割り当てることができます。SYCL 2020 仕様では、ラムダの命名はオプションです。本書のほとんどでは、SYCL を使用する C++ では、カーネルに匿名ラムダを使用するため名前は必要ありません（第 10 章のラムダの命名で説明したコンパイルオプションの受け渡しを除く）。

コードベース内で複数ベンダーの SYCL ツールを混在させるような高度なニーズがある場合、ツールでラムダに名前を付ける必要が生じます。命名は、ラムダが使用される SYCL アクション構造（例: `parallel_for`）に `<class`

uniquename> によって行われます。この命名により、複数ベンダーのツールが単一のコンパイル内で定義されていても対話できるようになり、デバッグツールとレイヤー内で定義したカーネル名を表示できます。

カーネルクエリーを使用する場合は、カーネルに名前を付ける必要もあります。SYCL 標準委員会は、SYCL 2020 標準でこの要求に対する解決策を見つけることができませんでした。たとえば、カーネルの preferred_work_group_size_multiple を照会するには、kernel クラスの get_info() メンバー関数を呼び出す必要がありますが、この関数には kernel クラスのインスタンスが必要であり、最終的に関連する kernel_bundle からカーネルへのハンドルを抽出するためにカーネルの名前（および kernel_id）が必要になります。

まとめ

今日のポピュラーカルチャーでは、ヒントは *ライフハック*と呼ばれることがあります。残念ながら、プログラミング文化ではハックに否定的な意味があるため、著者はこの章を“SYCL ハック”と名付けることを控えました。違いなく、この章では、SYCL で C++ を使用する実用的なヒントのほんの一部を説明したに過ぎません。SYCL を使用して C++ を最大限に活用する方法を共に学ぶことで、さらに多くのヒントを私たち全員で共有できるようになります。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 14 章　一般的な並列パターン

私たちがプログラマーとして最高のパフォーマンスを発揮できるとき、私たちはワークのパターンを認識し、最善の解決策となる実績のある手法を用います。並列プログラミングもまた同様であり、この分野で有用であることが証明されているパターンを研究しないのは大きな過ちです。ビッグデータ・アプリケーションに採用される `MapReduce` フレームワークを考えてみましょう。その成功は、主に、*マップ*と*リデュース*という 2 つのシンプルでありながら効果的な並列パターンに基づいています。

並列プログラミングには、使用するプログラミング言語に関係なく、何度も登場する共通パターンがあります。これらのパターンは汎用性が高く、あらゆるレベルの並列処理（サブグループ、ワークグループ、フルデバイスなど）、およびあらゆるデバイス（CPU、GPU、FPGA など）で使用できます。ただし、パターンの特定の特性（スケーラビリティーなど）は、異なるデバイスへの適合性に影響する可能性があります。アプリケーションを新しいデバイスに適合させるには、適切なパラメーターを選択するか、パターンの実装を微調整するだけで済むこともあります。また、まったく異なるパターンを選択することでパフォーマンスを向上できる場合もあります。

これらの一般的な並列パターンをどのように、いつ、どこで使用するか理解を深めることは、SYCL（および一般的な並列プログラミング）の熟練度を向上させる上で重要なことです。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_14

既に並列プログラミングの経験がある開発者にとって、これらのパターンが SYCL でどのように表現されるか確認することは、言語機能にすぐに慣れるための方法となります。

この章では、次の質問に対する回答を示します:

- 理解すべき一般的なパターンにはどのようなものがあるか？
- パターンは各種デバイスの機能とどのように関連しているか？
- SYCL 関数およびライブラリーとしてすでに提供されているパターンは？
- ダイレクト・プログラミングを使用してパターンをどのように実装するか？

パターンを理解する

ここで説明するパターンは、McCool 氏らによる書籍「*Structured Parallel Programming (構造化並列プログラミング)*」で説明されている並列パターンのサブセットです。ここでは、並列処理の*タイプ*（フォークジョイン、分岐限定など）に関連するパターンについては取り上げませんが、データ並列カーネルの作成に役立ついくつかのアルゴリズム・パターンに注目します。

私たちは、この並列パターンのサブセットを理解することが、優秀な SYCL プログラマーになるために重要であると信じています。図 14-1 の表は、各種パターンの概要を示しています。主な使用例、主要な属性、属性がさまざまなハードウェア・デバイスに与える親和性の影響などを示しています。

パターン	用途	主な属性	デバイス・アフィニティー
マップ	簡単な並列カーネル	データ依存関係がなく、高いスケーラビリティー	すべて
ステンシル	構造化されたデータの依存関係	データの依存関係とデータの再利用	ステンシルのサイズに依存
リダクション	部分的な結果を結合	データ依存関係	すべて
スキャン パック/アンパック	データのフィルター処理と再構築	スケーラビリティーの制限	問題サイズに依存

図 14-1. 並列パターンとデバイスタイプの親和性

マップ

マップパターンは最も単純な並列パターンであり、関数型プログラミング言語の経験がある読者には馴染みのあるものです。図 14-2 に示すように、範囲の各入力要素は、何らかの関数を適用することで個別に出力にマッピングされます。多くのデータ並列操作は、マップパターンのインスタンスとして表現できます（ベクトル加算など）。

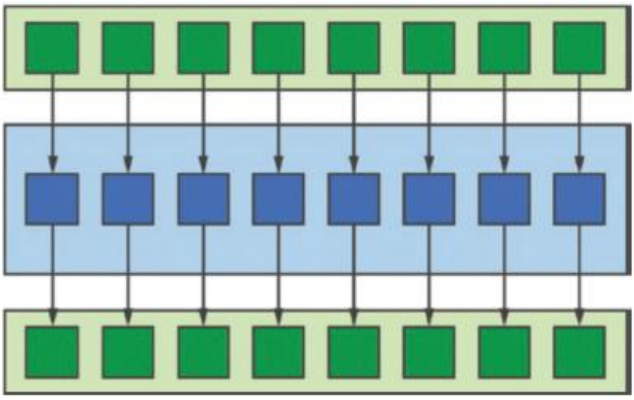

図 14-2. マップパターン

すべての関数アプリケーションは完全に独立しているため、`map` 式の多くは単純で、難しい処理の大部分をコンパイラーやランタイムに任せます。マップパターン向けに記述されたカーネルはあらゆるデバイスに適合し、それらのカーネルのパフォーマンスは利用可能なハードウェア並列処理量に応じてスケールすることが期待されます。

ただし、アプリケーション全体を一連のマップカーネルとして書き直す前に、慎重に検討する必要があります。このような開発アプローチは非常に生産的で、アプリケーションが各種デバイスへ移植可能であることを保証しますが、パフォーマンスを大幅に向上させる可能性のある最適化（データ再利用の改善、カーネルの融合など）を無視（保留）することを推奨します。

ステンシル

ステンシルパターンはマップパターンと密接な関係があります。図 14-3 に示すように、関数は入力と、ステンシルで記述された隣接する一連の入力に適用され、単一の出力が生成されます。ステンシルパターンは、科学/技術アプリケーション（例: 有限差分コード）やコンピューター・ビジョン/マシン・ラーニング・アプリケーション（例: イメージの畳み込み）など、多くの分野で使用されます。

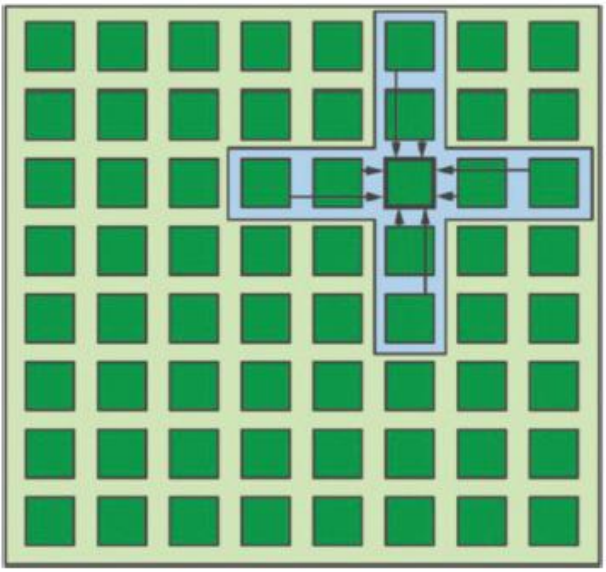

図 14-3. ステンシルパターン

ステンシルパターンをアウトオブプレースで実行すると（つまり、出力を別のストレージに書き込むと）、関数はすべての入力に個別に適用できます。現実世界のステンシルのスケジュール設定は、これよりも複雑です。隣接する出力を計算するには同じデータが必要であり、そのデータをメモリーから複数回ロードするとパフォーマンスが低下します。また、アプリケーションのメモリー・フットプリントを削減するため、ステンシルをインプレースで適用する（つまり、元の入力値を上書きする）場合もあります。

したがって、さまざまなデバイスに対するステンシルカーネルの適合性は、ステンシルの特性と入力する値に大きく依存します。一般に次のことが言えます:

- 小さなステンシルは、GPU のスクラッチパッド・ストレージの恩恵を得られます。
- 大きなステンシルは、CPU の（比較的）大きなキャッシュの恩恵を得られます。
- 小さな入力で動作する小さなステンシルは、FPGA 上のシストリック・アレイとして実装することで、大幅なパフォーマンス向上を実現できます。

ステンシルは記述するのは簡単ですが、効率良く実装するのは複雑なため、多

くのステンシル・アプリケーションではドメイン固有言語（DSL）が使用されます。C++ のテンプレート・メタプログラミング機能を活用して、コンパイル時に高性能なステンシルカーネルを生成する組み込み DSL はすでに存在します。

リダクション

リダクションとは、通常は*結合法*と*可換法*（加算など）の演算子を使用して部分的な結果を*結合*する一般的な並列パターンです。リダクションの最も一般的な例は、合計の計算（ドット積を計算するなど）や最小値/最大値の計算（最大速度を使用して時間ステップのサイズを設定するなど）があります。

図 14-4 は、ツリー・リダクションによって実装されたリダクション・パターンを示しています。これは、N 個の入力要素の範囲に対して $\log_2(N)$ の組み合わせ演算を必要とする実装です。ツリー・リダクションは一般的ですが、他の実装も可能です。一般に、リダクションによって値が特定の順序で結合されると想定すべきではありません。

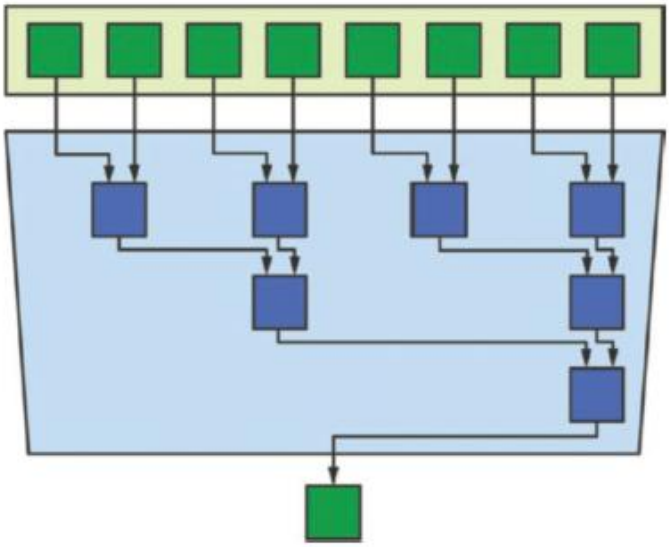

図 14-4. リダクション・パターン

現実の世界では、カーネルが驚異的に並列化されることはめったになく、たとえ並列化される場合でも、結果を要約するため（`MapReduce` フレームワークのように）リダクションと組み合わせられることがあります。そのため、リダクションは理解すべき最も重要な並列パターンの 1 つであり、どのデバイスでも効率良く実行される必要があります。

さまざまなデバイスに向けにリダクションを調整するには、部分的な結果の計算にかかる時間と、それらの結合にかかる時間のバランスを取る必要があります。並列処理が少なすぎると計算時間が長くなり、並列処理が多すぎると結合時間が長くなります。

異なるデバイスを組み合わせて計算と結合ステップを実行することで、システム全体の利用率を向上させたいと思うかもしれませんが、そのようなチューニングでは、デバイス間のデータ移動のコストに細心の注意が必要です。実際には、データが生成されると同じデバイス上で直接リダクションを実行しますが、ほとんどの場合、それが最善のアプローチであることがわかります。

つまり、複数のデバイスを使用してリダクション・パターンのパフォーマンスを向上させるのは、タスクの並列処理ではなく、異なるレベルのデータ並列処理に依存します（各デバイスが入力データの一部に対してリダクションを実行します）。

スキャン

スキャンパターンは、二項結合演算子によって一般化された累積和を計算し、出力の各要素は部分的な結果を表します。要素 i の部分和が範囲 $[0, i]$ 内のすべての要素の合計（つまり、合計に i *が含まれる*）である場合、スキャンは*包括的*であると言われます。また、要素 i の部分和が半開区間 $[0, i)$ 内のすべての要素の合計（つまり、i *を除いた*合計）である場合、スキャンは*排他的*であると言われます。

一見すると、スキャンは本質的にシリアル操作であるように見えます。つまり、各出力の値は前の出力の値に依存します。スキャンは他のパターンよりも並列処理の可能性が少ない（したがってスケーラビリティーが低い）のは事実ですが、図 14-5 は、同一データに対して複数の走査を行うことで並列スキャンを実装できることを示しています。

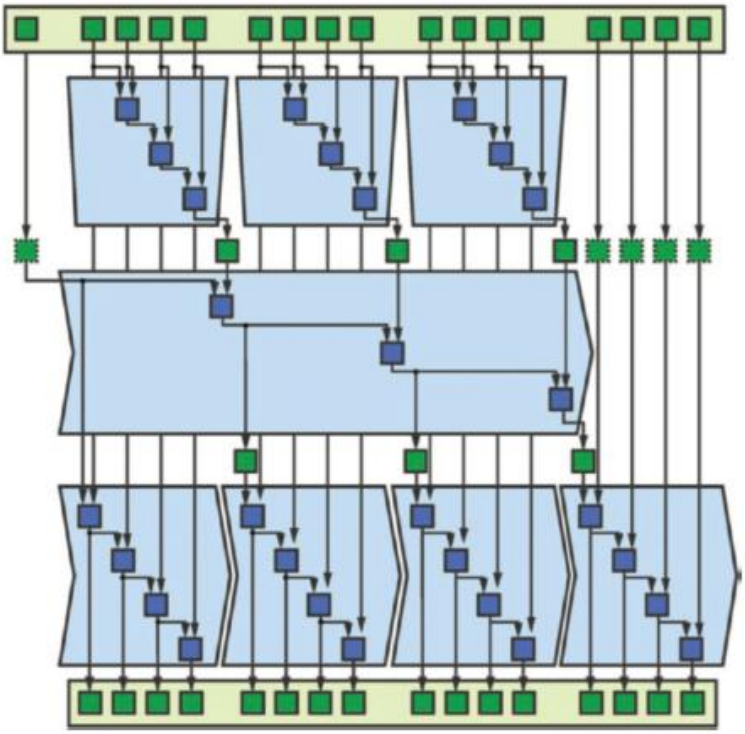

図 14-5. スキャンパターン

スキャン操作内での並列処理の可能性は限られているため、スキャンの実行に最適なデバイスは問題サイズに大きく依存します。つまり、GPU を飽和させるのに十分なデータ並列処理が含まれるのは大きな問題だけであるため、小さな問題は CPU に適しています。スキャンはパイプラインの並列処理に自然に適応するため、FPGA やその他の空間アーキテクチャーでは問題サイズはそれほど問題になりません。リダクションと同様に、通常は、データを生成したデバイスでスキャン操作を実行することを推奨します。最適化中にスキャン操作がアプリケーションに適合するか考慮することは、スキャン操作を個別に最適化するよりも良い結果が得られます。

パックとアンパック

パックパターンとアンパックパターンはスキャンと密接に関連しており、多くの場合、スキャン機能の上に実装されます。ここでそれらを個別に説明するのは、累積和とは関連ない可能性のある一般的な操作（リストへの追加など）を効率良く実装できるためです。

パック

図 14-6 に示すパックパターンは、ブール条件に基づいて入力範囲の要素を破棄し、破棄されない要素を出力範囲の連続した場所にパックします。このブール条件は、事前計算されたマスク値にすることも、各入力要素に何らかの関数を適用して実行時に計算することもできます。

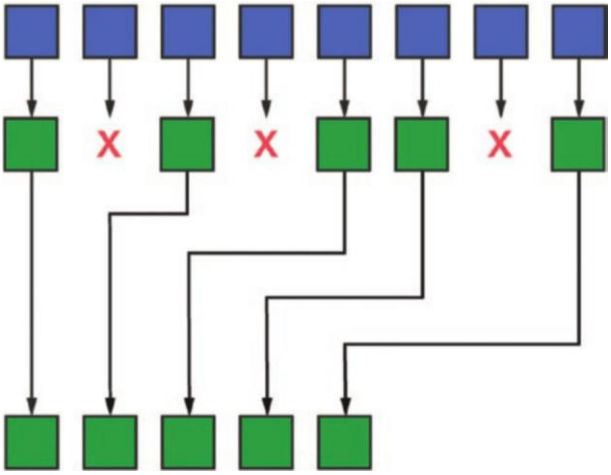

図 14-6. パックパターン

スキャンと同様に、パック操作にも本質的にシリアルな性質があります。パック/コピーする入力要素が与えられた場合、出力範囲内でのその位置を計算するには、出力にパック/コピーされた前の要素の数を知る必要があります。この情報は、パックを駆動するブール条件に対する排他的スキャンに相当します。

アンパック

図 14-7 に示すように（また名前で分かるように）、アンパックパターンはパックパターンの逆になります。入力範囲の連続した要素は、出力範囲の連続していない要素にアンパックされ、他の要素はそのまま残ります。このパターンの最も良くある使用例は、以前にパックされたデータをアンパックすることですが、以前の計算結果データの「隙間」を埋めるためにも使用できます。

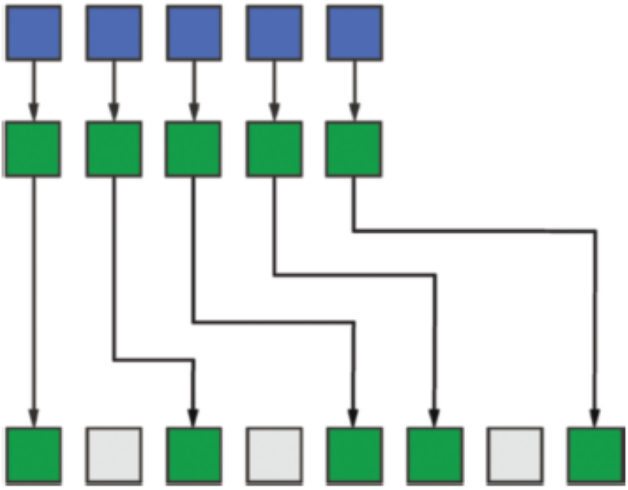

図 14-7. アンパックパターン

ビルトイン関数とライブラリーの使用

これらのパターンの多くは、SYCL のビルトイン関数や SYCL で記述されたベンダー提供のライブラリーを使用して直接表現できます。これらの関数とライブラリーを活用することは、実際の大規模ソフトウェア開発プロジェクトでパフォーマンス、移植性、生産性のバランスをとる最善の方法です。

SYCL リダクション・ライブラリー

SYCL は、移植可能で高性能なリダクション・カーネルの独自ライブラリーを利用者が管理する必要はなく、リダクション・セマンティクスにより変数を記述する便利な抽象化を提供します。この抽象化により、リダクション・カーネルの表現が簡素化され、リダクションの実行が明示的になり、実装ではデバイス、データ型、リダクション操作のさまざまな組み合わせに対して異なるリダクション・アルゴリズムを選択できます。

図 14-8 のカーネルは、リダクション・ライブラリーの使用例を示しています。カーネル本体にはリダクションに関する記述は一切含まれていないことに注意してください。指定すべきは、カーネルが `sum` 変数のインスタンスを `plus` ファンクターを用いて結合するリダクションを含むだけです。これにより、実装で最適化されたリダクション・シーケンスを自動生成するのに十分な情報が提供されます。

```
h.parallel_for(
    range<1>{N}, reduction(sum, plus<>()),
    [=](id<1> i, auto& sum) { sum += data[i]; });
```

図 14-8. リダクション・ライブラリーを使用してデータ並列カーネルとして表現されたリダクション

カーネルが完了するまで、リダクションの結果が元の変数に書き戻されることは保証されません。この制限を除けば、リダクションの結果へのアクセスは、SYCL 内の他の変数へのアクセスと同じように動作します。バッファーに格納されているリダクションの結果にアクセスするには、適切なデバイスまたはホストアクセサーを作成する必要があり、USM 割り当てに格納されているリダクションの結果にアクセスするには、明示的な同期やメモリーの移動が必要になる場合があります。

SYCL リダクション・ライブラリーが他の言語のリダクション抽象化と異なる重要な点は、カーネル実行中にリダクション変数へのアクセスが制限されることです。つまり、リダクション変数の中間値を検証することはできず、指定された結合関数以外でリダクション変数を更新することは禁止されています。これらの制限により、デバッグが困難なエラー（最大値を計算するときにリダクション変数に加算するなど）を防ぎ、さまざまなデバイスでリダクションを効率良く実装できるようになります。

`reduction` クラス

`reduction` クラスは、カーネル内にあるリダクションを記述するために使用するインターフェイスです。リダクション・オブジェクトを構築する唯一の方法は、図 14-9 に示す関数のいずれかを使用することです。リダクション関数には 3 つのファミリー（バッファー、USM ポインター、スパン用）があり、それぞれに 2 つのオーバーロード（ID 変数ありとなし）があることに注意してください。

```
template <typename BufferT, typename BinaryOperation>
unspecified reduction(BufferT variable, handler& h,
                      BinaryOperation combiner,
                      const property_list& properties = {});

template <typename BufferT, typename BinaryOperation>
unspecified reduction(BufferT variable, handler& h,
                      const BufferT::value_type& identity,
                      BinaryOperation combiner,
                      const property_list& properties = {});

template <typename T, typename BinaryOperation>
unspecified reduction(T* variable, BinaryOperation combiner,
                      const property_list& roperties = {});

template <typename T, typename BinaryOperation>
unspecified reduction(T* variable, const T& identity,
                      BinaryOperation combiner,
                      const property_list& properties = {});

template <typename T, typename Extent,
          typename BinaryOperation>
unspecified reduction(span<T, Extent> variables,
                    BinaryOperation combiner,
                    const property_list& properties = {});

template <typename T, typename Extent,
          typename BinaryOperation>
unspecified reduction(span<T, Extent> variables,
                      const T& identity,
                      BinaryOperation combiner,
                      const property_list& properties = {});
```

図 14-9. リダクション関数の関数プロトタイプ

リダクションがバッファーまたは USM ポインターを使用して初期化される場合、リダクションは*スカラー・リダクション*となり、配列内の最初のオブジェクトに対して実行されます。リダクションがスパンを使用して初期化される場合、リダクションは*配列*リダクションになります。配列リダクションの各コンポーネントは独立しています。サイズ N の配列に対して操作される配列リダクションは、同じデータ型と演算子を持つ N 個のスカラー・リダクションと同等であると考えることができます。

関数の最も単純なオーバーロードにより、各ワーク項目からの関与を結合するリダクション変数と演算子を指定できます。2 番目のオーバーロード・セットを使用すると、リダクション演算子に関連付けられたオプションの *ID* 値を提供できます。これは、ユーザー定義のリダクションの最適化であり、後で説明します。

リダクション関数の戻り値の型は指定されておらず、`reduction` クラス自体は完全に実装定義であることに注意してください。これは C++ クラスとしては少々変わっているように見えるかもしれませんが、実装では異なるクラス（または任意の数のテンプレート引数を持つ単一のクラス）を使用して、異なるリダクション・アルゴリズムを表現できます。SYCL の将来のバージョンでは、特定の実行コンテキストで特定のリダクション・アルゴリズムを明示的に要求できるようにするため、この設計を再検討する可能性があります（おそらく `property_list` 引数経由...）。

`reducer` クラス

`reducer` クラスのインスタンスは、リダクション変数をカプセル化し、実装が安全でないと判断する方法では、リダクション変数の更新を禁止する制限されたインターフェイスを公開します。`reducer` クラスの簡略化された定義を図 14-10 に示します。`reduction` クラスと同様に、`reducer` クラスの正確な定義は実装定義です。リデューサーの型はリダクションの実行方法によって異なり、パフォーマンスを最大化するにはコンパイル時に知っておくことが重要です。ただし、リダクション変数を更新する関数と演算子は適切に定義されており、いずれの SYCL 実装でもサポートされることが保証されています。

```
template <typename T, typename BinaryOperation,
          /* 実装定義 */>
class reducer  {
   // 部分的な結果をレデューサーの値と組み合わせる
   void combine(const T& partial);
};

// 他の演算子は標準の二項演算に使用できます
template <typename T>
auto& operator+=(reducer<T, plus::<T>>&, const T&);
```

図 14-10. reducer クラスの簡素化された定義

具体的には、すべてのリデューサーは、部分的な結果（単一のワーク項目から）とリダクション変数の値を結合する `combine()` 関数を提供します。この結合関数がどのように動作するかは実装によって定義されますが、カーネルを作成するときに考慮する必要はありません。また、リデューサーは、リダクション演算子に応じて他の演算子を使用できるようにする必要があります。たとえば、プラス・リダクションの場合は `+=` 演算子が定義されます。これらの追加の演算子は、プログラマーの利便性と可読性の向上のためだけに提供されています。これらの演算子が使用可能な場合、これらの演算子は、`combine()` を直接呼び出すのと同じように動作します。

配列のリダクションを行う場合、リデューサーは追加の添字演算子（つまり、`operator[]`）を提供し、配列の個々の要素にアクセスできるようにします。この演算子は、配列要素の参照を直接返すのではなく、スカラー・リダクションに関連付けられたリデューサーと同じ `combine()` 関数と短縮演算子を公開する別のリデューサー・オブジェクトを返します。図 14-11 は、配列リダクションを使用してヒストグラムを計算するカーネルの例を示しています。ここでは、添字演算子を使用して、ワーク項目によって更新されるヒストグラムのビンのみにアクセスしています。

```
h.parallel_for(
     range{N},
     reduction(span<int, 16>(histogram, 16), plus<>()),
     [=](id<1> i, auto& histogram) {
        histogram[i % B]++;
     });
```

図 14-11. 配列リダクションを使用してヒストグラムを計算するカーネルの例

ユーザー定義のリダクション

いくつかの一般的なリダクション・アルゴリズム（ツリー・リダクションなど）では、各ワーク項目が単一の共有変数を直接更新するのではなく、将来のある時点で結合されるプライベート変数に部分的な結果を蓄積します。このようなプライベート変数は、実装でどのように初期化すればよいかという問題があります。変数を各ワーク項目からの最初の寄与値で変数を初期化すると、初期化されていない変数を検出して処理する追加のロジックが必要になるため、パフォーマンスに影響を及ぼす可能性があります。代わりに、変数をリダクション演算子の `ID` に初期化すると、パフォーマンスの低下を回避できますが、これは `ID` が既知の場合にのみ可能です。

SYCL 実装では、リダクションが単純な算術型で動作し、リダクション演算子がいくつかの標準関数オブジェクト（例: `plus`）の 1 つである場合にのみ、正しい `ID` 値を自動的に決定できます。ユーザー定義のリダクション（つまり、ユーザー定義型の操作や、ユーザー定義関数オブジェクトを使用するもの）の場合、`ID` 値を直接指定することでパフォーマンスを向上できる可能性があります。

ユーザー定義のリダクションのサポートは、簡単にコピーできる型と副作用のない組み合わせ関数に限定されていますが、多くの場合これで十分です。たとえば、図 14-12 のコードは、ユーザー定義のリダクションを使用してベクトル内の最小要素とその位置の両方を計算する方法を示しています。

```
template <typename T, typename I>
using minloc = minimum<std::pair<T, I>>;

int main() {
  constexpr size_t N = 16;

  queue q;
  float* data = malloc_shared<float>(N, q);
  std::pair<float, int>* res =
      malloc_shared<std::pair<float, int>>(1, q);
  std::generate(data, data + N, std::mt19937{});

  std::pair<float, int> identity = {
      std::numeric_limits<float>::max(),
      std::numeric_limits<int>::min()};
  *res = identity;

  auto red =
      sycl::reduction(res, identity, minloc<float, int>());

  q.submit([&](handler& h) {
     h.parallel_for(
         range<1>{N}, red, [=](id<1> i, auto& res) {
           std::pair<float, int> partial = {data[i], i};
           res.combine(partial);
         });
    }).wait();

   std::cout << "minimum value = " << res->first << " at "
             << res->second << "\n";
   ...
```

図 14-12. ユーザー定義のリダクションを使用して最小値の位置を検出

グループ・アルゴリズム

SYCL デバイスコードでの並列パターンのサポートは、グループ・アルゴリズムの別のライブラリーで提供されます。これらの関数は、特定のワーク項目グループ（ワークグループまたはサブグループ）の並列処理を活用して、限定された範囲で共通の並列アルゴリズムを実装し、さらに複雑なアルゴリズムを構築する構成要素として使用できます。

SYCL のグループ・アルゴリズムの構文は、C++ のアルゴリズム・ライブラリーの構文に基づいており、C++ アルゴリズムの制限が適用されます。ただし、重要な違いがあります。STL のアルゴリズムはシーケンシャル（ホスト）コードから呼び出され、ライブラリーが並列処理を行う可能性を示しますが、SYCL のグループ・アルゴリズムは、すでに並列実行されている（デバイス）コード内で呼び出されるように設計されています。この違いを見落とさないようにするため、グループ・アルゴリズムの構文とセマンティクスは C++ のアルゴリズムと若干異なります。

SYCL は 2 つの異なる並列アルゴリズムを区別します。アルゴリズムがグループ内のすべてのワーク項目で協調実行されるが、それ以外は STL のアルゴリズムと同じように動作する場合、そのアルゴリズムの名前には “`joint`” というプリフィクスが付加されます（グループのメンバーがアルゴリズムの実行に “`join`” するため）。このようなアルゴリズムは、メモリーから入力を読み取り、その結果をメモリーに書き込み、特定のグループ内のすべてのワーク項目で参照されるメモリー位置のデータのみ操作できます。代わりに、アルゴリズムがグループ自体を反映する暗黙的な範囲で動作し、入力と出力がワーク項目のプライベート・メモリーに格納される場合、アルゴリズム名は “`group`” という単語を含むように変更されます（アルゴリズムはグループが所有するデータを直接実行するため）。

図 14-13 のコード例では、`std::reduce` の動作と `sycl::joint_reduce` および `sycl::reduce_over_group` の動作を比較しながら、これら 2 種類のアルゴリズムを示しています。

```
// std::reduce
// 各ワーク項目は、入力範囲全体をリダクションします
q.parallel_for(number_of_reductions, [=](size_t i) {
    output1[i] = std::reduce(
        input + i * elements_per_reduction,
        input + (i + 1) * elements_per_reduction);
 }).wait();

// sycl::joint_reduce
// 各ワークグループは、入力範囲全体をリダクションします
// 要素はグループ内のワーク項目に
// 自動的に分散されます
q.parallel_for(nd_range<1>{number_of_reductions *
                                elements_per_reduction,
                            elements_per_reduction},
               [=](nd_item<1> it) {
                 auto g = it.get_group();
                 int sum = joint_reduce(
                     g,
                     input + g.get_group_id() *
                                 elements_per_reduction,
                     input + (g.get_group_id() + 1) *
                                 elements_per_reduction,
                     plus<>());
                 if (g.leader()) {
                   output2[g.get_group_id()] = sum;
                 }
               })
    .wait();

// sycl::reduce_over_group
// 各ワークグループはワーク項目のプライベート・メモリーに保持されている
// データをリダクションします
// 各ワーク項目は 1 つの値をロードして提供する責任があります
q.parallel_for(
     nd_range<1>{
        number_of_reductions * elements_per_reduction,
        elements_per_reduction},
     [=](nd_item<1> it) {
        auto g = it.get_group();
        int x = input[g.get_group_id() *
                          elements_per_reduction +
                       g.get_local_id()];
        int sum = reduce_over_group(g, x, plus<>());
        if (g.leader()) {
           output3[g.get_group_id()] = sum;
        }
     })
    .wait();
```

図 14-13. std::reduce、sycl::joint_reduce、sycl::reduce_over_group の比較

どちらの場合も、各グループ・アルゴリズムの最初の引数は、実行ポリシーの代わりにグループまたはサブグループ・オブジェクトを受け入れ、アルゴリズムの実行に使用するワーク項目のセットを記述することに注意してください。
アルゴリズムは指定されたグループ内のすべてのワーク項目で協調して実行されるため、グループバリアと同様に扱われる必要があります。つまり、グループ内のすべてのワーク項目は、収束した制御フローで同じアルゴリズムに遭遇する必要があります（つまり、グループ内のすべてのワーク項目は、同様にアルゴリズム呼び出しに遭遇するか/しない必要があります）。また、すべてのワーク項目によって提供される引数は、実行される操作についてすべてのワーク項目が合意できるものでなければなりません。たとえば、`sycl::joint_reduce` では、グループ内のすべてのワーク項目が同じデータに対して動作し、同じ演算子で結果を蓄積できるように、すべてのワーク項目の引数は同じである必要があります。

図 14-14 の表は、STL で使用できる並列アルゴリズムがグループ・アルゴリズムとどのように関連しているか、および使用できるグループの種類に制限があるか示しています。場合によっては、グループ・アルゴリズムはサブグループでのみ使用できることに注意してください。このケースは、前の章で紹介した「シャッフル」操作に対応します。

C++ アルゴリズム	SYCL "Joint" アルゴリズム	SYCL "Group" アルゴリズム	グループタイプ
`std::any_of`	`sycl::joint_any_of`	`sycl::any_of_group`	すべて
`std::all_of`	`sycl::joint_all_of`	`sycl::all_of_group`	該当なし
`std::none_of`	`sycl::joint_none_of`	`sycl::none_of_group`	すべて
`std::shift_left`	該当なし	`sycl::shift_group_left`	`sub_group`
`std::shift_right`	該当なし	`sycl::shift_group_right`	`sub_group`
該当なし	該当なし	`sycl::permute_group_by_xor`	`sub_group`
該当なし	該当なし	`sycl::select from group`	`sub_group`
`std::reduce`	`sycl::joint reduce`	`sycl::reduce over group`	すべて
`std::exclusive scan`	`sycl::joint exclusive scan`	`sycl::exclusive scan over group`	すべて
`std::includive scan`	`sycl::jpint includive scan`	`sycl::includive scan over group`	すべて

図 14-14. C++ アルゴリズムと SYCL グループ・アルゴリズム間のマッピング

本書の執筆時点では、グループ・アルゴリズムは、プリミティブ・データ型と、SYCL によって認識されるビルトイン演算子のセット（`plus`、`multiplies`、`bit_and`、`bit_or`、`bit_xor`、`logical_and`、`logical_or`、`minimum`、`maximum`）のみがサポーされています。これは、ほとんどの一般的なユースケースをカバーするのに十分ですが、SYCL の将来のバージョンでは、ユーザー定義の型と演算子に対する総合的なサポートが拡張される予定です。

ダイレクト・プログラミング

可能な限りライブラリーを活用することを推奨しますが、「ネイティブ」SYCL カーネルを使用して各パターンをどのように実装できるかを調べることで、多くのことを学べます。

以降で紹介するカーネルは、高度に調整されたライブラリーと同レベルのパフォーマンスをもたらすことは期待できませんが、SYCL の機能に対する理解を深めるのに役立ちます。また、新しいライブラリー機能のプロトタイプを作成する出発点としても役立つはずです。

ベンダー提供のライブラリーを使用

ベンダーが関数のライブラリー実装を提供する場合、その関数をカーネルとして再実装するよりも、ライブラリーを使用する方がほとんどの場合有利です。

マップ

マップパターンは単純なため、基本的な並列カーネルとして直接実装できます。図 14-15 に示すコードは、マップパターンを使用して範囲内の各入力要素の平方根を計算する実装を示しています。

```
// 各入力値の平方根を計算
q.parallel_for(N, [=](id<1> i) {
   output[i] = sqrt(input[i]);
 }).wait();
```

図 14-15. データ並列カーネルにおけるマップパターンの実装

ステンシル

図 14-16 に示すように、多次元バッファーを備えた多次元基本データ並列カーネルとしてステンシルを直接実装するのは、簡単で容易に理解できます。

```
q.submit([&](handler& h) {
  accessor input{input_buf, h};
  accessor output{output_buf, h};

  // 各セルとその近傍セルの平均を計算
  h.parallel_for(stencil_range, [=](id<2> idx) {
    int i = idx[0] + 1;
    int j = idx[1] + 1;

    float self = input[i][j];
    float north = input[i - 1][j];
    float east = input[i][j + 1];
    float south = input[i + 1][j];
    float west = input[i][j - 1];
    output[i][j] =
        (self + north + east + south + west) / 5.0f;
  });
});
```

図 14-16. データ並列カーネルにおけるステンシルパターンの実装

ただし、このステンシルパターンの表現は非常に単純であり、良好なパフォーマンスは期待できません。この章の前半で述べたように、メモリーから同じデータを繰り返し読み取るのを避けるため、局所性（空間または時間ブロック化経由）を活用することはよく知られています。ワークグループ・ローカル・メモリーを使用した空間ブロック化の簡単な例を図 14-17 に示します。

```
q.submit([&](handler& h) {
   accessor input{input_buf, h};
   accessor output{output_buf, h};

   constexpr size_t B = 4;
   range<2> local_range(B, B);
   range<2> tile_size =
       local_range +
       range<2>(2, 2);     // 境界セルを含む
    auto tile = local_accessor<float, 2>(tile_size, h);

   // 各セルとその近傍セルの平均を計算
   h.parallel_for(
        nd_range<2>(stencil_range, local_range),
        [=](nd_item<2> it) {
          // このタイルをワークグループのローカルメモリーにロード
          id<2> lid = it.get_local_id();
          range<2> lrange = it.get_local_range();
          for (int ti = lid[0]; ti < B + 2;
              ti += lrange[0]) {
            int gi = ti + B * it.get_group(0);
            for (int tj = lid[1]; tj < B + 2;
                tj += lrange[1]) {
              int gj = tj + B * it.get_group(1);
              tile[ti][tj] = input[gi][gj];
            }
          }
          group_barrier(item.get_group());

          // ローカルメモリーの値を使用してステンシルを計算
         int gi = it.get_global_id(0) + 1;
         int gj = it.get_global_id(1) + 1;

          int ti = it.get_local_id(0) + 1;
          int tj = it.get_local_id(1) + 1;

          float self = tile[ti][tj];
          float north = tile[ti - 1][tj];
          float east = tile[ti][tj + 1];
          float south = tile[ti + 1][tj];
          float west = tile[ti][tj - 1];
          output[gi][gj] =
              (self + north + east + south + west) / 5.0f;
        });
});
```

図 14-17. ワークグループ・ローカル・メモリーを使用して、ND-range カーネルにステンシルパターンを実装

特定のステンシルに適した最適化を選択するには、ブロックサイズ、近傍、およびステンシル関数のコンパイル時の内部調査が必要であり、ここで説明したよりもはるかに洗練されたアプローチが求められます。

リダクション

ワーク項目間の同期および通信（アトミック操作、ワークグループとサブグループの関数、サブグループのシャッフルなど）に利用できる言語機能を活用することで、SYCL にリダクション・カーネルを実装できます。図 14-18 と図 14-19 のカーネルは、2 つのリダクションの実装を示しています。1 つは基本的な `parallel_for` とすべてのワーク項目に対するアトミック操作を使用する単純なリダクションで、もう 1 つは ND-range の `parallel_for` とワークグループのリダクション関数を使用して局所性を活用するやや賢いリダクションです。これらのアトミック操作については、第 19 章でさらに詳しく取り上げます。

```
q.parallel_for(N,[=](id<1> i) {
   atomic_ref<int, memory_order::relaxed,
               memory_scope::system,
               access::address_space::global_space>(
        *sum) += data[i];
 }).wait();
```

図 14-18. データ並列カーネルとして表現された自然なリダクションの実装

```
q.parallel_for(nd_range<1>{N, B}, [=](nd_item<1> it) {
    int i = it.get_global_id(0);
    auto grp = it.get_group();
    int group_sum =
        reduce_over_group(grp, data[i], plus<>());
    if (grp.leader()) {
       atomic_ref<int, memory_order::relaxed,
                   memory_scope::system,
                   access::address_space::global_space>(
            *sum) += group_sum;
    }
 }).wait();
```

図 14-19. ND-range カーネルとして表現される自然なリダクションの実装

リダクション・カーネルを記述する方法は他にも多数あり、アトミック操作のハードウェア・サポート、ワークグループのローカル・メモリー・サイズ、グローバル・メモリー・サイズ、デバイス全体で高速バリアが使用できるか、さらには専用のリダクション命令があるか、などの違いにより、デバイスごとに異なる実装が採用される可能性があります。一部のアーキテクチャーでは、$\log_2(N)$ の個別のカーネル呼び出しにより、ツリー・リダクションを実行する方が高速（または必須）である場合があります。

SYCL リダクション・ライブラリーでサポートされない場合、または特定のデバイスの機能に合わせてカーネルを調整する場合にのみ、リダクションを手動で実装することを強く推奨します。その場合も、SYCL のビルトイン・リダクションのパフォーマンスが低下することが確実である場合に限ります。

スキャン

この章の前半で説明したように、並列スキャンを実装するには、データに対して複数回の走査が必要であり、各走査間で同期が行われます。SYCL は ND-range 内のすべてのワーク項目を同期するメカニズムを提供しないため、デバイス全体のスキャンを直接実装するには、グローバルメモリーを介して部分的な結果を通信する複数のカーネルを使用する必要があります。

図 14-20、14-21、および 14-22 に示すコードは、複数のカーネルを使用して実装された包括的スキャンを示しています。最初のカーネルは、入力値をワークグループ間で分散し、ワークグループのローカルメモリー内のワークグループのローカルスキャンを計算します（代わりにワークグループの inclusive_scan 関数を使用するできます）。2 番目のカーネルは、単一のワークグループを使用して、各ブロックの最終値に対してローカルスキャンを計算します。3 番目のカーネルはこれらの中間結果を組み合わせて累積和を確定します。これら 3 つのカーネルは、図 14-5 の 3 つのレイヤーに対応します。

```
// フェーズ 1: 入力ブロック上のローカルスキャンを計算
q.submit([&](handler& h) {
   auto local = local_accessor<int32_t, 1>(L, h);
   h.parallel_for(nd_range<1>(N, L), [=](nd_item<1> it) {
     int i = it.get_global_id(0);
     int li = it.get_local_id(0);

      // 入力をローカルメモリーにコピー
      local[li] = input[i];
      group_barrier(it.get_group());

      // ローカルメモリー内で包括的スキャンを実行
      for (int32_t d = 0; d <= log2((float)L) - 1; ++d) {
         uint32_t stride = (1 << d);
         int32_t update =
              (li >= stride) ? local[li -  stride] : 0;
         group_barrier(it.get_group());
         local[li] += update;
         group_barrier(it.get_group());
      }

      // 各項目の結果を出力バッファに書き込みます。
      // このブロックの最後の結果を一時バッファに書き込みます。
      output[i] = local[li];
      if (li == it.get_local_range()[0] - 1) {
         tmp[it.get_group(0)] = local[li];
      }
   });
 }).wait();
```

図 14-20. ND-range カーネルでグローバル包括的スキャンを実装するフェーズ 1: 各ワークグループ全体を計算

```
// フェーズ 2: 部分的な結果のスキャンを計算
q.submit([&](handler& h) {
   auto local = local_accessor<int32_t, 1>(G, h);
   h.parallel_for(nd_range<1>(G, G), [=](nd_item<1> it) {
      int i = it.get_global_id(0);
      int li = it.get_local_id(0);

      // 入力をローカルメモリーにコピー
      local[li] = tmp[i];
      group_barrier(it.get_group());

      // ローカルメモリー内で包括的スキャンを実行
      for (int32_t d = 0; d <= log2((float)G) - 1; ++d) {
         uint32_t stride = (1 << d);
         int32_t update =
              (li >= stride) ? local[li - stride] : 0;
         group_barrier(it.get_group());
         local[li] += update;
         group_barrier(it.get_group());
      }

      // 各ワーク項目の結果を一時バッファーに上書き
      tmp[i] = local[li];
   });
 }).wait();
```

図 14-21. ND-range カーネルでグローバル包括的スキャンを実装するフェーズ 2：各ワークグループの結果全体をスキャン

```
// フェーズ 3: 部分的な結果を使用してローカルスキャンを更新
q.parallel_for(nd_range<1>(N, L), [=](nd_item<1> it) {
   int g = it.get_group(0);
   if (g > 0) {
     int i = it.get_global_id(0);
     output[i] += tmp[g - 1];
   }
 }).wait();
```

図 14-22. ND-range カーネルでグローバル包括的スキャンを実装するフェーズ 3（最後）

図 14-20 と図 14-21 は非常によく似ています。唯一の違いは、範囲のサイズと、入力値と出力値の処理方法です。このパターンの実際の実装では、異なる引数を持つ単一の関数を使用して 2 つのフェーズを実装できますが、ここでは説明上の理由から別のコードとして示してます。

パックとアンパック

パックとアンパックは、ギャザー操作とスキャッター操作とも呼ばれます。これらの操作は、データがメモリー内でどのように配置されるか、およびそれを計算リソースに提供する方法の違いを処理します。

パック

パックは排他的スキャンに依存するため、ND-range のすべての要素に適用されるパックの実装も、グローバルメモリーを介して、複数のカーネルエンキューにより実行する必要があります。ただし、パックには、ND-range のすべての要素に操作を適用する必要がないケースがあります。それは、特定のワークグループまたはサブグループ内の項目にのみパックを適用する場合です。

図 14-23 のコードは、排他的スキャンの上にグループのパック操作を実装する方法を示しています。

```
uint32_t index =
     exclusive_scan(g, (uint32_t)predicate, plus<>());
if (predicate) dst[index] = value;
```

図 14-23. 排他的スキャンの上にグループパック操作を実装

図 14-24 のコードは、パック操作をカーネル内で使用して、後処理（以降のカーネル）を必要とする要素リストを作成する方法を示しています。

ここに示す例は、分子動力学シミュレーションのカーネルに基づいています。粒子 *i* に割り当てられたサブグループ内のワーク項目は、*i* の固定距離内にある他のすべての粒子を識別するために協調し、この “隣接リスト” 内の粒子のみを使用して各粒子に作用する力が計算されます。

```
range<2> global(N, 8);
range<2> local(1, 8);
q.parallel_for(nd_range<2>(global, local), [=](nd_item<2> it) {
   int i = it.get_global_id(0);
   sub_group sg = it.get_sub_group();
   int sglid = sg.get_local_id()[0];
   int sgrange = sg.get_local_range()[0];

   uint32_t k = 0;
   for (int j = sglid; j < N; j += sgrange) {
      // i と隣接 j 間の距離を計算
      float  r = distance(position[i], position[j]);

      // 後処理が必要な近傍をリストにまとめる
      uint32_t pack = (i != j) and (r <= CUTOFF);
      uint32_t offset =
           exclusive_scan_over_group(sg, pack, plus<>());
      if (pack) {
        neighbors[i * MAX_K + k + offset] = j;
      }

      // これまでにいくつの隣接がパックされたか追跡
      k += reduce_over_group(sg, pack, plus<>());
    }
    num_neighbors[i] =
        reduce_over_group(sg, k, maximum<>());
 }).wait();
```

図 14-24. サブグループ・パック操作により、追加の後処理が必要な要素のリストを作成

パックパターンでは要素の順序が変更されることはありません。出力配列にパックされた要素は入力と同じ順序になります。パックのプロパティーは重要であり、パック機能を使用して他のより抽象的な並列アルゴリズム（`std::copy_if` や `std::stable_partition` など）を実装できます。ただし、順序を維持する必要がないパック機能に実装できる他の並列アルゴリズムもあります（`std::partition` など）。

アンパック

パックと同様に、スキャンを使用してアンパックを実装できます。図 14-25 は、排他的スキャンの上にサブグループのアンパック操作を実装する方法を示しています。

```
uint32_t index =
     exclusive_scan(sg, (uint32_t)predicate, plus<>());
return (predicate) ? new_value[index] : original_value;
```

図 14-25. 排他的スキャンの上にサブグループのアンパック操作を実装

図 14-26 のコードは、サブグループのアンパック操作を使用して、発散制御フロー（この場合はマンデルブロ集合の計算）を持つカーネルの負荷分散を改善する方法を示しています。各ワーク項目には計算する個別のピクセルが割り当てられ、収束するか最大反復回数に達するまで反復されます。次に、アンパック操作で完了したピクセルを新しいピクセルに置き換えます。

```
// 1 つのワーク項目にワークが残っている限り、反復を続行
while (any_of_group(sg, i < Nx)) {
   uint32_t converged = next_iteration(
        params, i, j, count, cr, ci, zr, zi, mandelbrot);
   if (any_of_group(sg, converged)) {
     // アンパックを使用して収束したピクセルを置き換え。
     // 収束していないピクセルは置き換えられません。
     uint32_t index = exclusive_scan_over_group(
          sg, converged, plus<>());
     i = (converged) ? iq + index : i;
     iq += reduce_over_group(sg, converged, plus<>());

     // 新しい i の反復変数をリセット
     if (converged) {
       reset(params, i, j, count, cr, ci, zr, zi);
     }
   }
}
```

図 14-26. サブグループのアンパック操作を使用して、制御フローが異なるカーネルの負荷分散を改善

このようなアプローチによって効率がどの程度向上し、実行時間が短縮されるかは、完了の確認とアンパック操作の実行の両方でオーバーヘッドが発生するため、アプリケーションと入力に大きく依存します。したがって、実際のアプリケーションでこのパターンをうまく使用するには、発散の量と実行される計算に基づいて微調整を行う必要があります（たとえば、アクティブなワーク項目数があるしきい値を下回った場合にのみアンパック操作を実行するなど）。

まとめ

この章では、ビルトイン関数やライブラリーなどの SYCL 機能を使用して、最も一般的な並列パターンのいくつかを実装する方法を説明しました。

SYCL エコシステムはまだ発展途上であり、開発者が言語や運用環境レベルのアプリケーションおよびライブラリーの開発経験を積むにつれて、これらのパターンに関する新しいベスト・プラクティスが明らかになることが期待されます。

関連情報

- Structured Parallel Programming: Patterns for Efficient Computation、Michael McCool、Arch Robison、James Reinders、© 2012, published by Morgan Kaufmann, ISBN 978-0-124-15993-8。

- 『構造化並列プログラミング』効率良い計算を行うためのパターン: マイケル・マックール、アーク・D・ロビソン、ジェームス・レインダース共著、2014 年、カットシステム出版、菅原清文/エクセルソフト訳、ISBN978-4-87783-305-3

- Algorithms library, C++ Reference、https://en.cppreference.com/w/cpp/algorithm。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 15 章　GPU 向けのプログラミング

過去数十年にわたり、グラフィックス・プロセッシング・ユニット（GPU）は、画面上にイメージを描画する特殊なハードウェア・デバイスから、複雑な並列カーネルを実行できる汎用デバイスへと進化してきました。現在、ほぼすべてのコンピューターには従来の CPU に加えて GPU が搭載されており、並列アルゴリズムの一部を CPU から GPU にオフロードすることで多くのプログラムを高速化できます。

この章では、一般的な GPU の動作、GPU ソフトウェアとハードウェアが SYCL アプリケーションを実行する方法、および GPU 向けの並列カーネルを作成そして最適化する際に留意すべきヒントとテクニックについて説明します。

パフォーマンスに関する注意事項

他のプロセッサー・タイプと同様に、GPU はベンダーごとに、あるいは製品世代ごとに異なります。そのため、あるデバイスのベスト・プラクティスが、別のデバイスとは異なる場合があります。この章のアドバイスは、現在および将来にわたって多くの GPU に役立つでしょう …

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_15

特定の GPU で最適なパフォーマンスを実現するには、GPU ベンダーのドキュメントを参照してください。

この章の最後に、多くの GPU ベンダーのドキュメントへのリンクを示します。

GPU の仕組み

ここでは、一般的な GPU の動作と、GPU が他のアクセラレーター・タイプとどのように異なるか説明します。

GPU ビルディング・ブロック

図 15-1 は、3 つの高レベルのビルディング・ブロックで構成される簡略化された GPU の構成を示しています。

1. **実行リソース**: GPU の実行リソースは、計算ワークを実行するプロセッサーです。さまざまな GPU ベンダーが実行リソースに異なる名称を使用していますが、すべての最新の GPU はプログラム可能な複数のプロセッサーで構成されています。プロセッサーは、バーテックスの変換やピクセルのシェーディングなどの特定のタスクに特化された*異種*プロセッサーである場合もあれば、*同種*で置き換え可能な場合もあります。最新の GPU のプロセッサーのほとんどは*同種*であり、互換性があります。

2. **固定機能**: GPU 固定機能は、実行リソースよりもプログラム性が低く、単一のタスクに特化したハードウェア・ユニットです。GPU をグラフィックス処理に使用する場合、ラスタライズやレイ・トレーシングなどグラフィックス・ パイプラインの多くの部分は、電力効率とパフォーマンス向上のために固定機能を使用して実行されます。GPU をデータ並列計算に使用する場合、ワークロードのスケジュール、テクスチャーのサンプリング、依存関係の追跡などのタスクに固定機能が使用されることがあります。

3. **キャッシュとメモリー**: 他のプロセッサー・タイプと同様に、GPU には実行リソースによってアクセスされるデータを格納するキャッシ

ュが備わっています。GPU キャッシュは*暗黙的*である場合があり、プログラマーがアクションを起こす必要ありません。また、*明示的*なスクラッチパッド・メモリーである場合もあり、それにはプログラマーがキャッシュを活用する前に意図的にデータをキャッシュに移動する必要があります。多くの GPU には、実行リソースで使用されるデータへの高速アクセスを提供する大容量のメモリープールもあります。

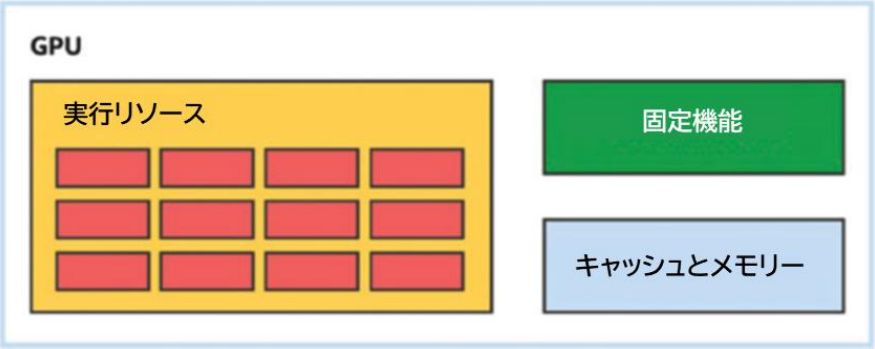

図 15-1. 一般的な GPU ビルディング・ブロック - スケールしません

よりシンプルなプロセッサー（ただし、数は多い）

本来、グラフィックス操作を処理する場合、GPU は大量のデータを処理します。たとえば、一般的なゲームフレームまたはレンダリング・ワークロードには、フレームごとに数百万のピクセルを生成する数千のバーテックスが含まれます。インタラクティブなフレームレートを維持するには、これらの大量のデータをできるだけ速く処理する必要があります。

一般的な GPU 設計のトレードオフは、図 15-2 に示すように、シングルスレッドのパフォーマンスを高速化する実行リソースを形成するプロセッサーから機能を削除し、その分を追加のプロセッサーとして構築することです。たとえば、GPU プロセッサーには、汎用プロセッサーで使用される高度なアウトオブオーダー実行や分岐予測が含まれいない場合があります。このようなトレードオフにより、単一のデータ要素を GPU で処理する速度は他のプロセッサーよりも遅くなる可能性がありますが、プロセッサーの数が多いほど、GPU は多くのデータ要素を迅速かつ効率良く処理できます。

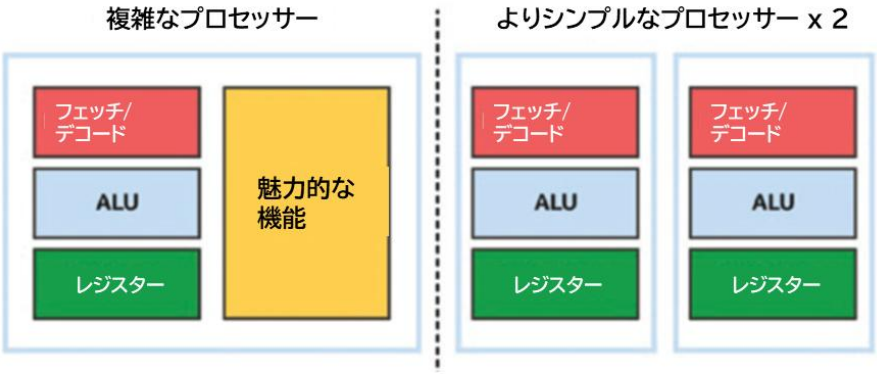

図 15-2. GPU プロセッサーはシンプルですが、多数あります

カーネルの実行にこのトレードオフを利用するには、GPU で処理するのに十分な範囲のデータ要素を与えることが重要です。広範囲のデータをオフロードすることの重要性を示すために、本書を通じて使用してきた行列乗算カーネルを考えてみましょう。

行列乗算に関する注意事項

本書では、行列乗算カーネルを使用して、カーネルの変更やカーネルのディスパッチ方法がどのようにパフォーマンスに影響するかを説明します。この章で説明する手法を使用すると行列乗算のパフォーマンスは大幅に向上しますが、行列乗算は非常に重要かつ一般的な操作であるため、多くのハードウェア・ベンダー（GPU、CPU、FPGA、DSP など）が行列乗算を含む多くの高度に調整されたルーチンを実装しています。ベンダーは、特定のデバイスの機能実装と検証に時間と労力を費やしており、場合によっては標準カーネルでは困難または不可能な機能や手法を提供することもあります。

ベンダー提供のライブラリーを使用

ベンダーが関数のライブラリー実装を提供する場合、その関数をカーネルとして再実装するよりも、ほとんどの場合ライブラリーを使用する方が有利です。oneAPI に含まれる oneMKL プロジェクトでは、インテルは MKL、nVIDIA は cuBLAS、AMD は hipBLAS を呼び出すインターフェイスを提案しています。このようなインターフェイスが利用可能であれば、作業が簡単になりますが、そうでない場合は、ターゲット・ハードウェアに最適なライブラリーを使用していることを確認するため、独自の作業を行う必要があります。

行列乗算カーネルは、単一のタスクとしてキューに送信することで、GPU 上で簡単に実行できます。この行列乗算カーネルの本体は、ホスト CPU 上で実行される関数とまったく同じであり、図 15-3 に示されています。

```
h.single_task([=]() {
  for (int m = 0; m < M; m++) {
    for (int n = 0; n < N; n++) {
      T sum = 0;
      for (int k = 0; k < K; k++) {
        sum += matrixA[m * K + k] * matrixB[k * N + n];
      }
      matrixC[m * N + n] = sum;
    }
  }
});
```

図 15-3. シングルタスクの行列乗算は CPU ホストコードに似ています

このカーネルは CPU 上で問題なく実行できるでしょう。CPU の並列機能を利用することは想定されていないため、パフォーマンスはそれほど高くありませんが、小さな行列サイズであれば十分に良好なパフォーマンスが得られる可能性があります。ただし、図 15-4 に示すように、このカーネルを GPU で実行すると、単一のタスクが 1 つの GPU プロセッサーしか利用しないため、非常に低いパフォーマンスを示すでしょう。

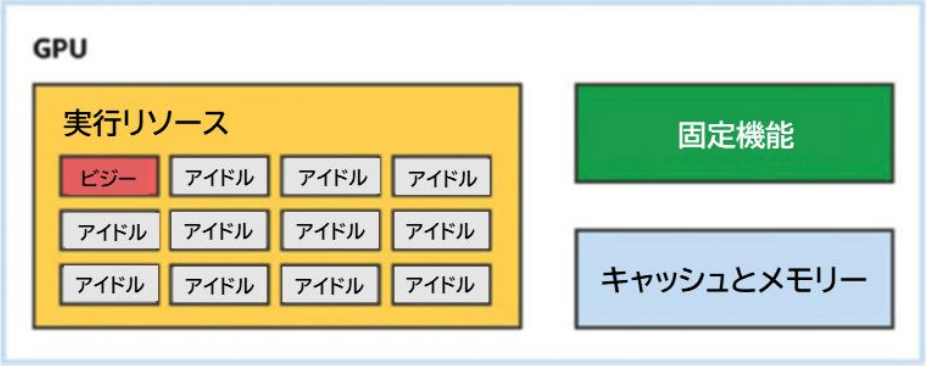

図 15-4. GPU 上のシングルタスク・カーネルは多くの実行リソースをアイドル状態にする

並列処理の表現

CPU と GPU の両方でこのカーネルのパフォーマンスを向上させるには、ループを parallel_for に置き換えて、一連のデータ要素を並列処理するように送信します。行列乗算カーネルでは、最も外側の 2 つのループのいずれかを表すデータ要素の範囲を送信することを選択できます。図 15-5 では、行列の行を並列に処理することを選択しています。

```
h.parallel_for(range{M}, [=](id<1> idx) {
  int m = idx[0];

  for (int n = 0; n < N; n++) {
     T sum = 0;
     for (int k = 0; k < K; k++) {
       sum += matrixA[m * K + k] * matrixB[k * N + n];
     }
     matrixC[m * N + n] = sum;
   }
});
```

図 15-5. ある程度並列な行列乗算

並列処理の方法を選択

どの次元を並列化するか選択することは、GPU と他のデバイスタイプの両方でアプリケーションをチューニングする重要な方法の 1 つです。この章の以降のセクションでは、1 つの次元で並列化すると、別の次元で並列化するよりもパフォーマンスが向上する可能性があるいくつかの理由を説明します。

ある程度並列化されたカーネルはシングルタスク・カーネルと似ていますが、CPU ではより適切に実行され、GPU ではさらに適切に実行されるはずです。図 15-6 に示すように、parallel_for を使用すると、結果行列の行を表すワーク項目を複数のプロセッサー・リソースで並列に処理できるため、すべての実行リソースをビジー状態に維持できます。

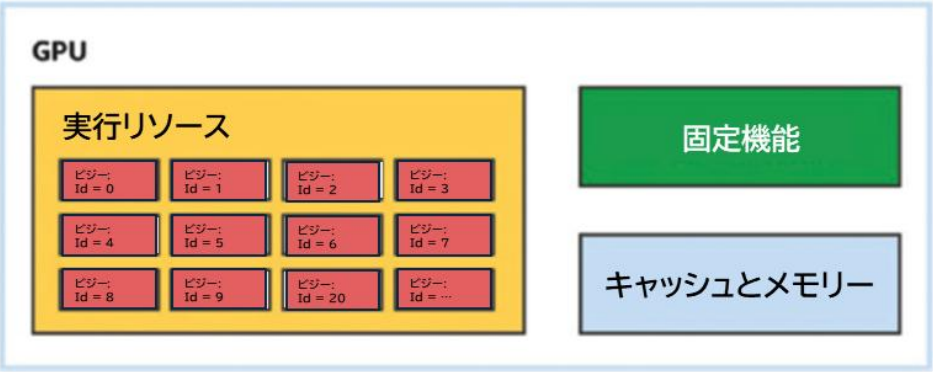

図 15-6. ある程度並列化されたカーネルは多くのプロセッサー・リソースを消費

行を分割して異なるプロセッサー・リソースに割り当てる正確な方法は指定されていないため、デバイス上でカーネルを最適に実行する方法を選択する柔軟性が与えられていることに注意してください。たとえば、プロセッサー上で個々の行を分割して実行する代わりに、実装では、局所性の利点を得るため、同一プロセッサー上で連続した行を実行する場合があります。

並列処理の表現

両方の外側のループを並列に処理することで、行列乗算カーネルをさらに並列化できます。図 15-7 に示すように、`parallel_for` は最大 3 次元の並列ループを表現できます。図 15-7 では、`parallel_for` に渡される範囲と並列実行空間内のインデックスを表す項目の両方が 2 次元であることに注意してください。

```
h.parallel_for(range{M, N}, [=](id<2> idx) {
  int m = idx[0];
  int n = idx[1];

  T sum = 0;
  for (int k = 0; k < K; k++) {
    sum += matrixA[m * K + k] * matrixB[k * N + n];
  }

  matrixC[m * N + n] = sum;
});
```

図 15-7. さらに並列な行列乗算

並列処理を追加すると、GPU 上で実行したときの行列乗算カーネルのパフォーマンスが向上する可能性があります。行列の行数が GPU プロセッサーの数を超える場合でも、これは当てはまります。次のセクションでは、このような状況になる可能性の理由について説明します。

簡略化された制御ロジック（SIMD 命令）

多くの GPU プロセッサーでは、ほとんどのデータ要素がカーネルを介して同じ制御フローパスを通過する傾向があるという事実を活用して、制御ロジックを最適化します。たとえば、行列乗算カーネルでは、ループ境界は不変であるため、各データ要素は最も内側のループを同じ回数実行します。

データ要素がカーネルの同じ制御フローパスを通過する場合、プロセッサーは、複数のデータ要素間で制御ロジックを共有し、それらをグループとして実行することで命令ストリームの管理コストを削減できます。これを実現する 1 つの方法は、*単一命令複数データ*（*SIMD*）命令セットを実装することです。この命令セットでは、複数のデータ要素が単一の命令によって同時に処理されます。

スレッドと命令ストリーム

多くの並列プログラミングのコンテキストや GPU の文献では、「スレッド」という用語は「命令ストリーム」の意味で使用されています。これらのコンテキストでは、「スレッド」は従来のオペレーティング・システムのスレッドとは異なりはるかに軽量です。ただし、常にそうではなく、場合によっては、「スレッド」がまったく異なるものを説明するために使用されることもあります。

「スレッド」という用語は、GPU ベンダー間でもさまざまな定義があり、誤解を招く恐れがあるため、この章では「命令ストリーム」という用語を使用します。

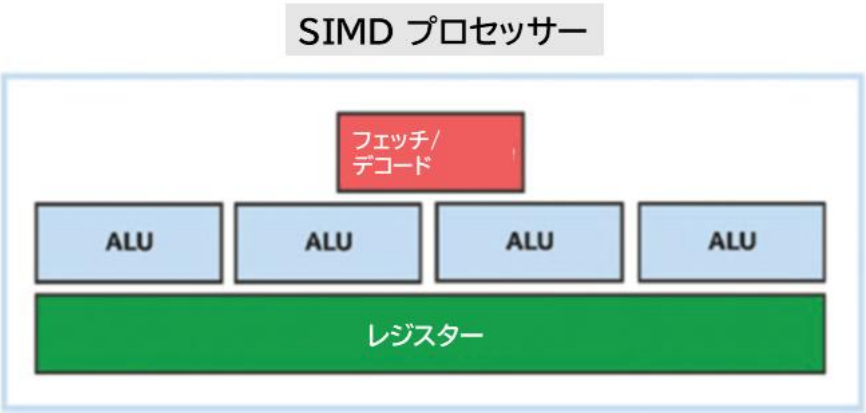

図 15-8. 4 ワイド SIMD プロセッサー: 4 つの ALU がフェッチ/デコードロジックを共有

1 つの命令で同時に処理されるデータ要素の数は、命令の *SIMD 幅（ワイド）*、または命令を実行するプロセッサーの *SIMD 幅（ワイド）* と呼ばれます。図 15-8 では、4 つの ALU が同じ制御ロジックを共有しているため、これは 4 ワイド SIMD プロセッサーとなります。

SIMD 命令セットを実装するのは GPU プロセッサーだけではありません。他のプロセッサー・タイプでも、大量データの処理効率を向上させるため SIMD 命令セットが実装されています。GPU プロセッサーと他のタイプのプロセッサーの主な違いは、GPU プロセッサーは優れたパフォーマンスを達成するため。複数のデータ要素を並列に実行し、他のタイプのプロセッサーよりも広い SIMD 幅をサポートする可能性があることです。たとえば、GPU プロセッサーが 16、32、またはそれ以上のデータ要素の SIMD 幅をサポートすることは珍しくありません。

プログラミング・モデル: SPMD と SIMD

GPU プロセッサーはさまざまな幅の SIMD 命令セットを実装していますが、これは実装レベルの話であり、GPU プロセッサーでデータ並列カーネルを実行するアプリケーションには透過的です。これは、多くの GPU コンパイラーとランタイム API が、単一プログラム複数データ（SPMD）プログラミング・モデルを実装するためです。このモデルでは、GPU コンパイラーとランタイム API が、SIMD 命令を明示的に表現するのではなく、SIMD 命令ストリームで処理する最も効率良いデータ要素のグループを決定します。第 9 章のサブグループのセクションでは、アプリケーションがデータ要素をグループ化するケースについて説明しています。

図 15-9 では、各実行リソースを 4 ワイド SIMD をサポートするように拡張し、4 倍の行列行を並列処理できるようにしています。

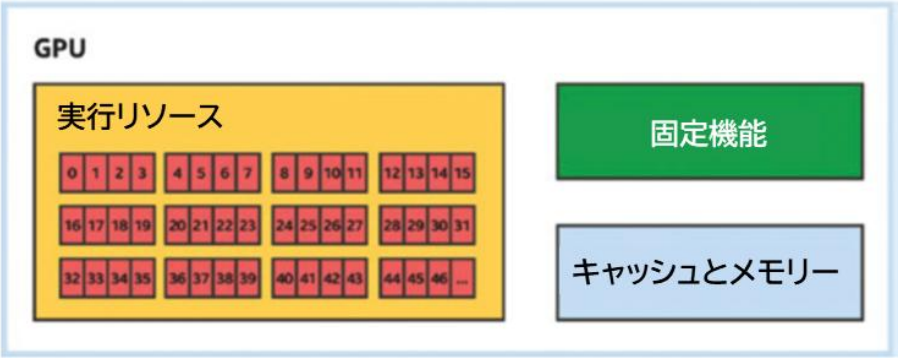

図 15-9. SIMD プロセッサー上である程度並列化されたカーネルを実行

複数のデータ要素を並列に処理する SIMD 命令は、図 15-5 および 15-7 の並列行列乗算カーネルのパフォーマンスをプロセッサー数にかかわらず拡張できる方法の 1 つです。SIMD 命令を使用すると、連続するデータ要素を同じプロセッサーで実行することにより、行列乗算など多くの場合に自然な局所性の利点も得られます。

カーネルはプロセッサー間の並列処理と、プロセッサー内の並列処理の恩恵を受けます。

条件付きとマスク

複数のデータ要素間で命令ストリームを共有することは、すべてのデータ要素がカーネル内の条件付きコードで同じパスを通過する場合は正常に機能します。データ要素が条件付きコードを通過して異なるパスを取る場合、制御フローは*分岐する*と言われます。SIMD 命令ストリームで制御フローが分岐する場合、通常は、一部のチャネルがマスクまたは*条件付き*で、両方の制御フローパスが実行されます。

これにより正しい動作が保証されますが、マスクされたチャネルは有用なワークを実行しないため、正当性にはパフォーマンス・コストが発生します。

条件付きとマスクの仕組みを示すため、図 15-10 のカーネルを考えてみます。このカーネルは、「奇数」インデックスを持つ各データ要素を 2 倍にし、「偶数」インデックスを持つ各データ要素を 1 ずつ増分します。

```
h.parallel_for(array_size, [=](id<1> i) {
  auto condition = i[0] & 1;
  if (condition) {
    dataAcc[i] = dataAcc[i] * 2;    // 奇数
  } else {
    dataAcc [i] = dataAcc[i] + 1;   // 偶数
  }
 });
```

図 15-10. 分岐制御フローを持つカーネル

このカーネルを図 15-8 に示す 4 ワイド SIMD プロセッサーで実行し、最初の 4 つのデータ要素を 1 つの SIMD 命令ストリームで実行し、次の 4 つのデータ要素を別の SIMD 命令ストリームで実行するようにします。図 15-11 は、このカーネルが分岐制御フローで正しく実行するため、チャネルをマスクして実行を予測する方法を示しています。

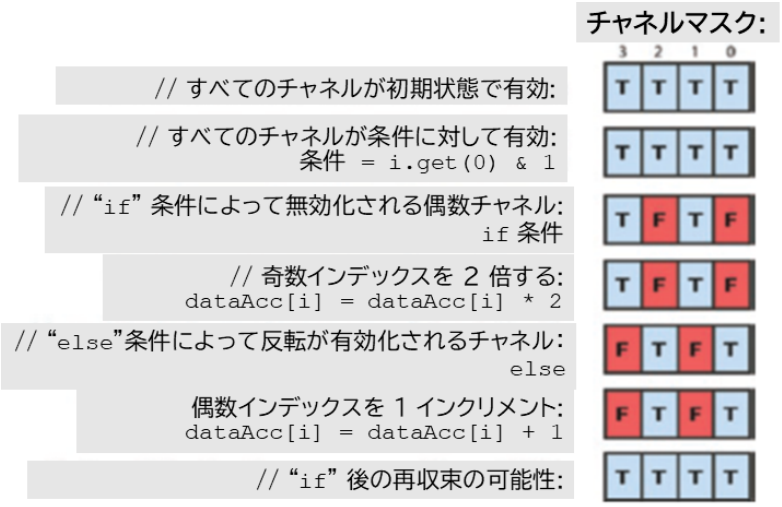

図 15-11. 分岐カーネルで可能なチャネルマスク

SIMD 効率

*SIMD 効率*は、同等のスカラー命令ストリームと比較して、SIMD 命令ストリームのパフォーマンスがどの程度上回っているか測定します。図 15-11 では、制御フローがチャネルを 2 つの等しいグループに分割しているため、分岐した制御フロー内の各命令は半分の効率で実行されます。最悪の場合、大きく異なるカーネルでは、プロセッサーの SIMD 幅に比例して効率が低下する可能性があります。

SIMD 命令セットを実装するすべてのプロセッサーは、SIMD 効率に影響する分岐ペナルティーの影響を受けますが、GPU プロセッサーは通常、他のプロセッサー・タイプよりも広い SIMD 幅をサポートしているため、分岐制御フローを最小限にして収束実行を最大化するようにアルゴリズムを再構築すると、GPU カーネルの最適化に効果があります。これは常に可能であるとは限りませんが、たとえば、より収束した実行を伴う次元に沿って並列化すると、分岐の多い実行を伴う次元に沿って並列化するよりもパフォーマンスが向上する可能性があります。

SIMD 効率と項目のグループ

これまで、この章のカーネルはすべて、実行範囲内の項目をグループ化しない基本的なデータ並列カーネルですが、デバイスに最適なグループ化を選択できる実装になっています。たとえば、SIMD 幅の広いデバイスでは、グループを大きくすることが推奨されますが、SIMD 幅の狭いデバイスでは、グループを小さくしても問題ない場合があります。

カーネルがワーク項目の明示的なグループ化を備えた ND-range カーネルである場合、SIMD 効率を最大化する ND-range のワークグループ・サイズを選択するようにする必要があります。ワークグループのサイズがプロセッサーの SIMD 幅で均等に割り切れない場合、ワークグループの一部がカーネルの実行期間全体でチャネルを無効にして実行されることがあります。`preferred_work_group_size_multiple` のデバイス固有のカーネル照会を使用して、効率良いワークグループ・サイズを選択できます。デバイスのプロパティーを照会する方法の詳細については、第 12 章を参照してください。

多くの GPU は 1 つを除くすべての SIMD チャネルをマスクして単一ワーク項目のワークグループを実装するため、単一ワーク項目で構成されるワークグループ・サイズを選択すると、パフォーマンスが大幅に低下する可能性があります。たとえば、図 15-12 のカーネルは、図 15-5 のよく似たカーネルよりもパフォーマンスが大幅に低下する可能性があります。2 つのカーネルの大きな違いは、基本的なデータ並列カーネルと非効率的な単一ワーク項目の ND-range カーネル（`nd_range<1>{M, 1}`）です。

```
h.parallel_for(
     nd_range<1>{M, 1}, [=](nd_item<1> idx) {
       int m = idx.get_global_id(0);

       for (int n = 0; n < N; n++) {
         T sum = 0;
         for (int k = 0; k < K; k++) {
           sum += matrixA[m * K + k] * matrixB[k * N + n];
         }
         matrixC[m * N + n] = sum;
       }
     });
```

図 15-12. 非効率な単一項目、ある程度並列な行列乗算

ワークの切り替えとレイテンシーの隠匿

多くの GPU では、制御ロジックを簡素化し、実行リソースを最大限に活用し、パフォーマンスを向上させるため、もう 1 つの手法を使用します。それは、プロセッサー上で単一の命令ストリームを実行する代わりに、プロセッサー上に複数の命令ストリームを同時に常駐させることです。

プロセッサー上に複数の命令ストリームを常駐させることは、各プロセッサーで実行するワークの選択肢を与えるため有益です。1 つの命令ストリームがメモリーからの読み取りなど、レイテンシーの長い操作を実行している場合、プロセッサーは操作の完了を待機することなく、実行準備ができている別の命令ストリームに切り替えることができます。十分な命令ストリームがあれば、プロセッサーが元の命令ストリームに戻るまでに、プロセッサーを待機させずに、レイテンシーの長い操作を完了できる可能性があります。

図 15-13 は、プロセッサーが複数の同時命令ストリームを使用してレイテンシーを隠匿し、パフォーマンスを向上させる方法を示しています。最初の命令ストリームでは複数のストリームで実行するのに少し時間がかかりましたが、他の命令ストリームに切り替えることで、プロセッサーは実行準備ができているワークを検出でき、長い操作の完了を待機する必要がなくなりました。

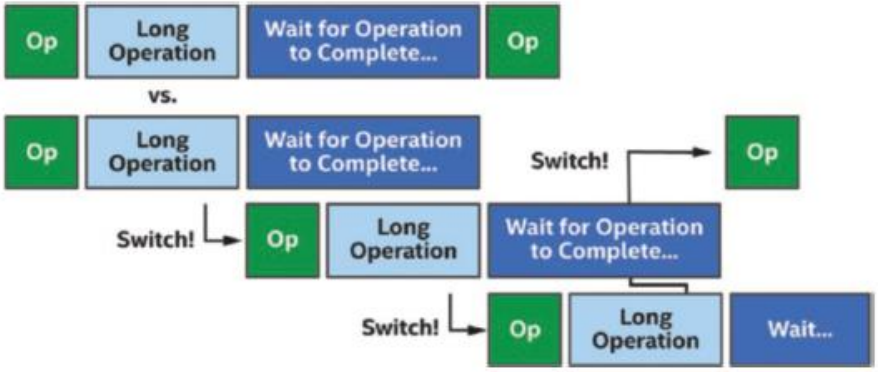

図 15-13. 命令ストリームの切り替えによるレイテンシーの隠蔽

GPU プロファイル・ツールは、GPU プロセッサーが実行している命令ストリーム数と理論上の総命令ストリーム数を比較する際、*占有率*などの用語を用いて表わすことがあります。

占有率が低いことは、必ずしもパフォーマンスが低いことを意味しません。少数の命令ストリームによってプロセッサーがビジー状態になる可能性があるためです。同様に、すべての命令ストリームが非効率でレイテンシーの長い操作を実行する場合、GPU プロセッサーは依然として待機の必要があるため、占有率が高いことは必ずしもパフォーマンスが高いことも意味しません。ただし、他の条件が同じであれば、占有率を高めると GPU プロセッサーのレイテンシーを隠匿する能力が最大化され、パフォーマンスが向上します。占有率の増加は、図 15-7 のさらに並列化されたカーネルでパフォーマンスが向上する可能性がある（もう 1 つの）理由です。

複数の命令ストリームを切り替えてレイテンシーを隠匿する手法は、GPU とデータ並列処理に適しています。図 15-2 からわかるように、GPU プロセッサーは他のプロセッサー・タイプよりもシンプルである場合が多く、複雑なレイテンシーの隠蔽機能がありません。これにより、GPU プロセッサーはレイテンシーの影響を受けやすくなりますが、データ並列プログラミングには大量のデータ処理が含まれるため、GPU プロセッサーには実行する命令ストリームが豊富にあります。

カーネルを GPU へオフロード

このセクションでは、アプリケーション、SYCL ランタイム・ライブラリー、および GPU ソフトウェア・ドライバーが連携して、カーネルを GPU ハードウェアにオフロードする方法について説明します。 図 15-14 の図は、これらの抽象化レイヤーを備えた一般的なソフトウェア・スタックを示しています。多くの場合、これらのレイヤーはアプリケーションに対して透過的ですが、アプリケーションをデバッグまたはプロファイルするときは、これらのレイヤーを理解することが重要です。

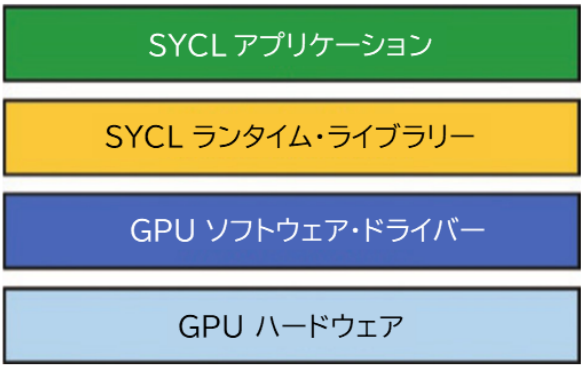

図 15-14. 並列カーネルを GPU にオフロード (簡略化)

SYCL ランタイム・ライブラリー

SYCL ランタイム・ライブラリーは、SYCL アプリケーションとインターフェイスする主要なソフトウェア・ライブラリーです。ランタイム・ライブラリーは、キュー、バッファー、アクセサーなどのクラスとこれらのクラスのメンバー関数の実装に相当します。ランタイム・ライブラリーの一部はヘッダーファイルにある場合があり、それはアプリケーション実行可能ファイルに直接コンパイルされます。ランタイム・ライブラリーの他の部分はライブラリー関数として実装され、アプリケーション・ビルド・プロセスの一部としてアプリケーションの実行可能ファイルにリンクされます。ランタイム・ライブラリーは通常、デバイス固有ではなく、同じランタイム・ライブラリーが CPU、GPU、FPGA、または他のデバイスへのオフロードを調整する場合があります。

GPU ソフトウェア・ドライバー

SYCL ランタイム・ライブラリーが GPU に直接オフロードすることは理論的には可能ですが、実際には、ほとんどの SYCL ランタイム・ライブラリーは GPU ソフトウェア・ドライバーを介してワークを GPU に送信します。

GPU ソフトウェア・ドライバーは通常、OpenCL、レベルゼロ、CUDA などの API の実装です。GPU ソフトウェア・ドライバーの大部分は、SYCL ランタイムが呼び出すユーザーモード・ドライバー・ライブラリーに実装されており、ユーザーモード・ドライバーはオペレーティング・システムやカーネル・モード・ドライバーを呼び出して、メモリーの割り当てやデバイスへのワークの送信などシステムレベルのタスクを実行する場合があります。

ユーザーモード・ドライバーは、他のユーザーモード・ライブラリーを呼び出すこともあります。たとえば、GPU ドライバーは GPU コンパイラーを呼び出して、カーネルを中間表現から GPU ISA（命令セット・アーキテクチャー）にジャストインタイム・コンパイルする場合があります。これらのソフトウェア・モジュールとそれらの相互作用を図 15-15 に示します。

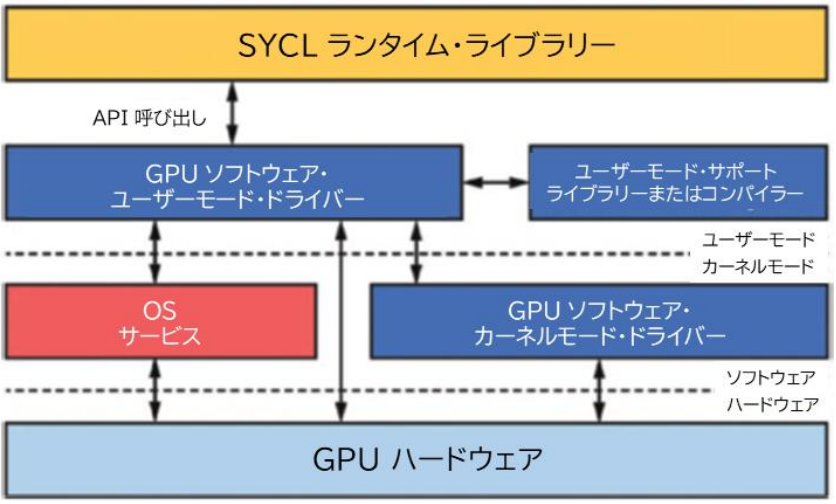

図 15-15. 一般的な GPU ソフトウェア・ドライバー・モジュール

GPU ハードウェア

ランタイム・ライブラリーまたは GPU ソフトウェア・ユーザーモード・ドライバーが、明示的にワークの送信を要求された場合や、GPU ソフトウェアがヒューリスティックでワークを開始する必要があると判断した場合、通常はオペレーティング・システムやカーネル・モード・ドライバーを介して呼び出しを行い、GPU 上でワークの実行を開始します。 場合によっては、GPU ソフトウェア・ユーザーモード・ドライバーが、GPU に直接ワークを送信することもありますが、それはあまり一般的ではなく、すべてのデバイスやオペレーティング・システムでサポートされるわけではありません。

GPU で実行されたワークの結果がホスト・プロセッサーまたは別のアクセラレーターによって利用されると、GPU はワークが完了したことを示すシグナルを発行する必要があります。ワーク完了に必要な手順は、ワーク送信の手順と似ており、逆の順番で実行されます。つまり、GPU はオペレーティング・システムまたはカーネル・モード・ドライバーに実行の完了を通知し、次にユーザーモード・ドライバーに通知され、最後にランタイム・ライブラリーが GPU ソフトウェア API 呼び出しを介してワークが完了したことを確認します。

これらの各ステップではレイテンシーが発生し、多くの場合、ランタイム・ライブラリーと GPU ソフトウェアは、レイテンシー軽減とスループット向上の間でトレードオフを行います。たとえば、ワークを GPU に頻繁に送信するとレイテンシーが短縮される可能性がありますが、頻繁に送信することで送信ごとのオーバーヘッドによりスループットも低下する可能性があります。大量のワークを収集するとレイテンシーは増加しますが、より多くのワークに対して送信のオーバーヘッドが分散され、並列実行の可能性が増えます。ランタイムとドライバーは適切なトレードオフを行うように調整されており、通常はうまく機能しますが、ドライバーのヒューリスティックによってワークが非効率に送信されている疑いがある場合、ドキュメントを参照して、API 固有または実装固有のメカニズムによりデフォルトのドライバー動作をオーバーライドする方法があるか確認する必要があります。 第 20 章で説明した API バックエンドと直接対話する手法は、GPU 送信ポリシーを調整するのに役立ちます。

オフロードのコストに注意

SYCL 実装と GPU ベンダーは、GPU へのワークのオフロードコストを軽減するため、継続して革新と最適化を行っていますが、GPU でワークを開始するときと、ホストまたは別のデバイスで結果を参照するときの両方で、常にオーバーヘッドが発生します。アルゴリズムを実行する場所を選択する場合、デバイス上でアルゴリズムを実行する利点と、アルゴリズムに必要なデータをデバイスに移動するコストの両方を考慮してください。 場合によっては、アルゴリズムをあるプロセッサーから別のプロセッサーに移動するオーバーヘッドを回避するため、ホスト・プロセッサーを使用して並列操作を実行するか、非効率ですがアルゴリズムのシリアル部分を GPU 上で実行するのが最も効率良いことがあります。

アルゴリズム全体のパフォーマンスを考えると、アルゴリズムの一部を別のデバイスに転送して実行するよりも、1 つのデバイスで非効率に実行するのが最も良い場合があります。

デバイスメモリー間との転送

専用メモリーを備えた GPU では、専用 GPU メモリーとホストまたは別のデバイス上のメモリー間の転送コストに特に注意する必要があります。図 15-16 は、システム内の異なるメモリータイプ間の一般的なメモリー帯域幅の違いを示しています。

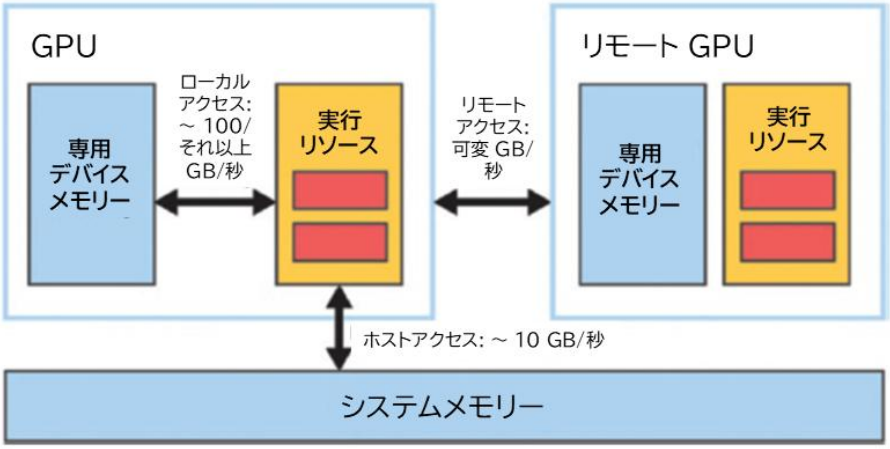

図 15-16. デバイスメモリー、リモートメモリー、ホストメモリーの一般的な違い

第 3 章で説明したように、GPU はホストメモリーや他のデバイスのメモリーではなく、専用のデバイスメモリーで動作する傾向があります。これにより、桁違いに高速化される場合があります。専用デバイスメモリーへのアクセスは、リモートメモリーやシステムメモリーへのアクセスよりも高速ですが、データが専用デバイスメモリーに存在しない場合は、コピーまたは移動する必要があります。

データが頻繁にアクセスされる限り、専用デバイスメモリーに移動することは有益であり、特に GPU 実行リソースが別のタスク処理でビジー状態の間に非同期に転送できる場合は有益です。ただし、データへのアクセス頻度が低い場合や予測できない場合は、アクセスごとのコストが高くても、転送コストを減らし、データをリモートまたはシステムメモリー内で操作することが望ましい場合があります。第 6 章では、メモリーの割り当て場所を制御する方法と、専用のデバイスメモリーにデータをコピーおよびプリフェッチする手法について説明しています。これらの手法は、GPU 向けにプログラム実行を最適化するときに重要です。

GPU カーネルのベスト・プラクティス

前のセクションでは、`parallel_for` に渡されるディスパッチ・パラメーターが、カーネルを GPU プロセッサーのリソースに割り当てる方法と、GPU 上でカーネルを実行する際に関係するソフトウェア・レイヤーとオーバーヘッドにどのように影響するかについて説明しました。ここでは、カーネルが GPU 上で実行されるベスト・プラクティスについて説明します。

大まかに言えば、カーネルは*メモリー依存*（つまり、GPU の実行リソースに対するデータの読み取りと書き込み操作によってパフォーマンスが制限される）であるか、計算依存（つまり、GPU の実行リソースによってパフォーマンスが制限される）であるかのいずれかです。GPU やその他の多くのプロセッサー向けにカーネルを最適化する最初のステップとして適切なことは、カーネルがメモリー依存か計算依存であるか判断することです。これは、メモリー依存のカーネルを改善する手法が、計算依存のカーネルには当てはまらないことがあり、その逆も同様です。GPU ベンダーは、このような判断に役立つプロファイル・ツールを提供しています。

カーネルがメモリー依存であるか、計算依存であるかに応じて、異なる最適化手法が必要になります。

GPU は多数のプロセッサーと広い SIMD 幅を持つ傾向があるため、カーネルはメモリー依存になる傾向があります。どこから始めればよいかわからない場合、カーネルがメモリーにアクセスする方法を調べるのが最初のステップとして適切です。

グローバルメモリーのアクセス

ワーク項目またはワークグループが操作するほぼすべてのデータはグローバルメモリーから取得されるため、グローバルメモリーに効率良くアクセスすることは、アプリケーションのパフォーマンス最適化に重要です。カーネルがグローバルメモリーを非効率に操作すると、ほとんどの場合、パフォーマンスが低下します。

GPU には、メモリー内の任意の場所の読み取りと書き込みを行う専用の*ギャザー*と*スキャッター*のハードウェア・ユニットが含まれることがありますが、グローバルメモリーのパフォーマンスは、データアクセスの*局所性*によって決まります。ワークグループ内の 1 つのワーク項目が、ワークグループ内の別のワーク項目によってアクセスされる要素に隣接するメモリー要素にアクセスする場合、グローバル・メモリー・アクセスのパフォーマンスは良好になる可能性があります。ワークグループ内のワーク項目がストライドまたはランダムにメモリーアクセスする場合、グローバル・メモリー・アクセスのパフォーマンスは低下する可能性があります。一部の GPU ドキュメントでは、近くのメモリーアクセスに対する操作を*結合*されたメモリーアクセスとして説明しています。

図 15-5 のある程度並列化された行列乗算カーネルでは、結果行列の行または列のどちらを並列処理するか選択でき、結果行列の行を並列に操作することを選択しています。これは、不適切な選択であることが分かっています。`ID` が `m` に等しい 1 つのワーク項目が、`ID` が `m-1` または `m+1` に等しい隣接するワーク項目とグループ化されている場合、`matrixB` へのアクセスに使用されるインデックスは各ワーク項目に対して同じですが、`matrixA` へのアクセスに使用されるインデックスは `K` だけ異なります。つまり、アクセスはストライドされていることになります。`matrixA` のアクセスパターンを図 15-17 に示します。

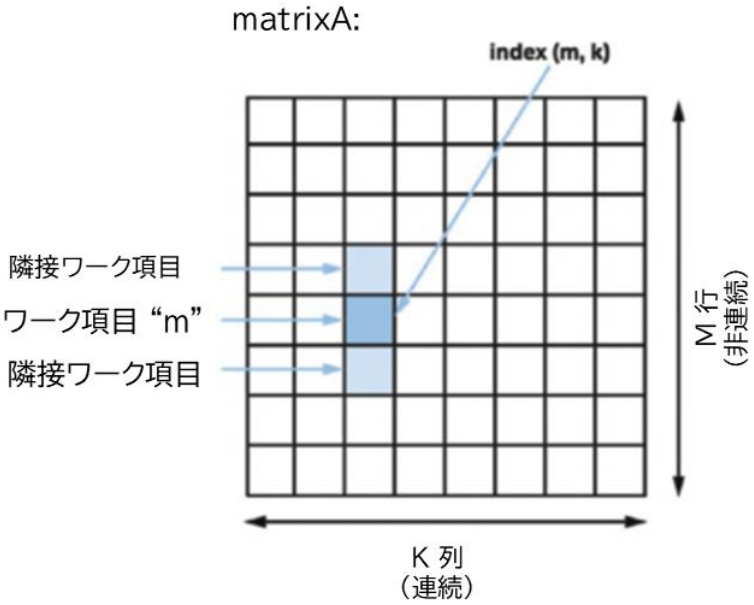

図 15-17. matrixA のアクセスはストライドされており、非効率です

代わりに、結果行列の列を並列に処理する場合、アクセスパターンの局所性が大幅に向上します。図 15-18 のカーネルは構造的に図 15-5 のカーネルと似ていますが、唯一の違いは、図 15-18 の各ワーク項目が結果行列の行ではなく、列を操作することです。

```
h.parallel_for(N, [=](item<1> idx) {
   int n = idx[0];

   for (int m = 0; m < M; m++) {
     T sum = 0;
     for (int k = 0; k < K; k++) {
       sum += matrixA[m * K + k] * matrixB[k * N + n];
     }
     matrixC[m * N + n] = sum;
   }
 });
```

図 15-18. 行列の行ではなく列を並列に計算

2 つのカーネルは構造的に似ていますが、多くの GPU では、データの列を操作するカーネルの方が、メモリーアクセスが効率的であるという理由でデータの行を操作するカーネルよりもパフォーマンスが優れています。`ID` が `n` の 1 つのワーク項目が、`ID` が `n-1` または `n+1` の隣接するワーク項目とグループ化されている場合、`matrixA` へのアクセスに使用されるインデックスは各ワーク項目で同じになり、`matrixB` へのアクセスするインデックスは連続します。`matrixB` のアクセスパターンを図 15-19 に示します。

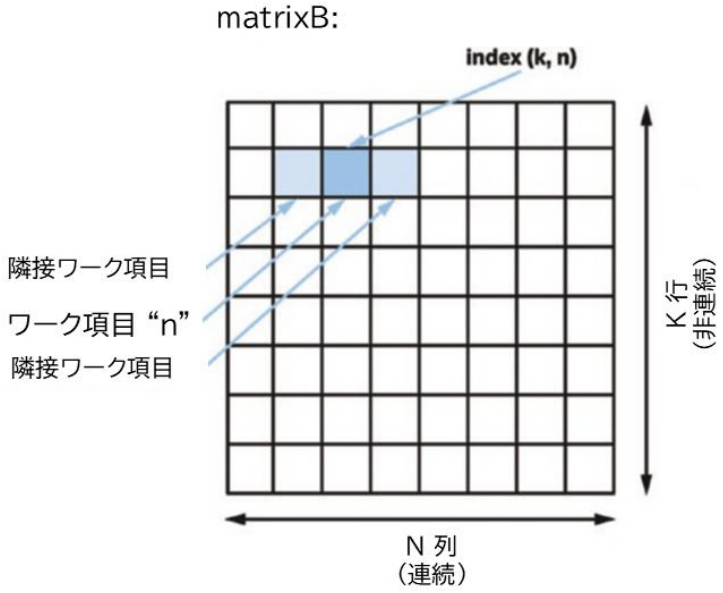

図 15-19. matrixB のアクセスは連続しており効率的です

連続したデータへのアクセスは通常非常に効率的です。経験則として、ワーク項目のグループのグローバルメモリーをアクセスするパフォーマンスは、アクセスされる GPU キャッシュライン数の関数であると言えます。すべてのアクセスが単一のキャッシュラインにある場合、アクセスは最高のパフォーマンスで実行されます。2 つのキャッシュラインにアクセスする場合（たとえば、1 つおきの要素にアクセスしたり、キャッシュが不整合なアドレスから開始する場合）、アクセスのパフォーマンスは約半分になる可能性があります。

グループ内の各ワーク項目が固有のキャッシュラインにアクセスする場合（ストライドされたアクセスやランダムアクセスなど）、そのアクセスは最低のパフォーマンスで動作する可能性が高くなります。

カーネルバリアントのプロファイル

行列乗算では、1 つの次元に沿って並列化すると明らかにメモリーアクセスの効率が向上しますが、他のカーネルでは、それほど明白ではない可能性があります。最高のパフォーマンスを達成することが重要であるカーネルは、並列化する次元が明白でない場合は、各次元に沿って並列化するいくつかのカーネルバリアントを作成してプロファイルを行い、デバイスとデータセットに最も適したものを確認する価値があります。

ワークグループ・ローカル・メモリーのアクセス

前のセクションでは、グローバルメモリーへのアクセスが*局所性*からどのように恩恵を受け、キャッシュ・パフォーマンスを最大化するか説明しました。すでに述べたように、並列化する次元を選択するなどして、メモリーに効率良くアクセスするアルゴリズムを設計できる場合もあります。ただし、この手法はすべてのケースに当てはまるわけではありません。このセクションでは、ワークグループのローカルメモリーを使用して、より多くのメモリー・アクセス・パターンを効率良くサポートする方法について説明します。

第 9 章で説明したように、ワークグループ内のワーク項目は、ワークグループのローカルメモリーを介して通信し、ワークグループのバリアを使用して同期することで協調して問題を解決できます。一般的な GPU にはバリアとワークグループ・ローカル・メモリーの両方を実装する特殊なハードウェアがあるため、この手法は GPU では特に有益です。異なる GPU ベンダーや異なる製品では、ワークグループ・ローカル・メモリーの実装方法が異なる場合がありますが、ワークグループ・ローカル・メモリーには、グローバルメモリーと比較して、次の 2 つの利点があります。

ローカルメモリーは、グローバル・メモリー・アクセスがキャッシュにヒットする場合でも、グローバルメモリーへのアクセスよりも高い帯域幅と低いレイテンシーをサポートできます。また、ローカルメモリーは、多くの場合、*バンク*と呼ばれる異なるメモリー領域に分割されます。グループ内の各ワーク項目が異なるバンクにアクセスする限り、ローカル・メモリー・アクセスは高いパフォーマンスで実行されます。バンクアクセスにより、ローカルメモリーは、グローバルメモリーよりもはるかに多くのアクセスパターンをピークのパフォーマンスでサポートできるようになります。

多くの GPU ベンダーは、連続したローカル・メモリー・アドレスを異なるバンクに割り当てます。これにより、開始アドレスに関係なく、連続したメモリーアクセスが常に最高のパフォーマンスで動作することが保証されます。ただし、メモリーアクセスがストライドされると、グループ内の一部のワーク項目が同じバンクに割り当てられたメモリーアドレスにアクセスする可能性があります。このような状況が発生すると、*バンク競合*によりシリアルアクセスが発生し、パフォーマンスが低下します。

グローバルメモリーのパフォーマンスを最大限に高めるには、アクセスされるキャッシュライン数を最小限に抑えます。

ローカルメモリーのパフォーマンスを最大限に高めるには、バンク競合を最小限に抑えます。

グローバルメモリーとローカルメモリーのアクセスパターンと、予想されるパフォーマンスの概要を図 15-20 に示します。`ptr` がグローバルメモリーを指す場合、ポインターは GPU のキャッシュ・ライン・サイズにアライメントされていると想定されます。グローバルメモリーへのアクセスの最高のパフォーマンスは、キャッシュ・アラインメントされたアドレスから連続してアクセスすることによって実現できます。アラインメントされていないアドレスにアクセスすると、追加のキャッシュラインへのアクセスが必要になる可能性があるため、グローバルメモリーのパフォーマンスが低下する恐れがあります。 アラインメントされていないローカルアドレスにアクセスしてもバンク競合は発生しないため、ローカルメモリーのパフォーマンスは変わりません。

ストライドのケースについては、さらに詳しく説明する必要があります。グローバルメモリー内のすべての要素にアクセスするには、より多くのキャッシュラインにアクセスする必要があり、パフォーマンスが低下する可能性があります。ローカルメモリー内のすべての要素にアクセスすると、バンクの競合が発生し、パフォーマンスが低下する可能性がありますが、これはバンクの数が 2 で割り切れる場合のみです。バンク数が奇数の場合、ピーク・パフォーマンスで動作します。

アクセス間のストライドが非常に大きい場合、各ワーク項目は異なるキャッシュラインにアクセスするため、パフォーマンスが最悪になります。ただし、ローカルメモリーの場合、パフォーマンスはストライドとバンクの数によって異なります。ストライド N がバンクの数に等しい場合、アクセスのたびにバンク競合が発生し、すべてのアクセスがシリアル化されるため、パフォーマンスが最悪になります。ただし、ストライド M とバンク数に共通因数がない場合、アクセスはピーク・パフォーマンスで実行されます。このため、多くの最適化された GPU カーネルでは、バンク競合を軽減または排除するストライドを選択するため、ローカルメモリーにデータ構造をパディングします。

図 15-20. グローバルメモリーとローカルメモリーの異なるアクセスパターンで可能なパフォーマンス

サブグループを使用してローカルメモリーを完全に回避

第 9 章で説明したように、サブグループの集合関数は、グループ内のワーク項目間でデータを交換する代替方法です。多くの GPU では、サブグループは単一の命令ストリームによって処理されるワーク項目の集合を表します。そのような場合、サブグループ内のワーク項目は、ワークグループのローカルメモリーを使用せずに、低コストでデータを交換し同期できます。高性能な GPU カーネルの多くはサブグループを使用しているため、計算コストの高いカーネルは、アルゴリズムを再構成してサブグループの集合関数を使用できるか検討する価値があります。

小さなデータ型を使用した計算の最適化

このセクションでは、メモリーアクセスのボトルネックを排除または軽減した後に、カーネルを最適化する手法について説明します。念頭に置くべき重要な点の 1 つは、GPU は従来、ディスプレイ上にイメージを描画するために設計されているということです。GPU の純粋な計算機能は時間の経過とともに進化、向上してきましたが、一部の領域ではグラフィックスの伝統が依然として残っています。

たとえば、カーネルデータ型のサポートを検討してください。多くの GPU は、グラフィックスやゲームでよく使用される 32 ビット浮動小数点演算に高度に最適化されています。 低い精度に対応できるアルゴリズムの場合、多くの GPU は、精度を犠牲にして処理速度を速める、16 ビット浮動小数点型もサポートしています。逆に、多くの GPU は 64 ビットの倍精度浮動小数点演算もサポートしていますが、高い精度にはコストがかかり、通常、32 ビット演算は 64 ビット演算よりもはるかに優れたパフォーマンスを発揮します。

同じことが整数データ型にも当てはまり、通常、32 ビット整数データ型は 64 ビット整数データ型よりもパフォーマンスが優れており、16 ビット整数はさらに優れたパフォーマンスを発揮する場合があります。より小さな整数を使用するように計算を構造化できれば、カーネルのパフォーマンスは向上する可能性があります。注意を払うべきことの 1 つは、アドレス指定操作です。これは通常、64 ビットの `size_t` データ型で操作されますが、場合によっては、ほとんどの計算を 32 ビットデータ型を使用して実行するように再配置できることもあります。

ほとんどのローカルメモリー割り当ては小さいため、16 ビットのインデックスで十分です。

数学関数の最適化

カーネルがパフォーマンスのために精度をトレードオフする可能性があるもう 1 つの領域は、SYCL ビルトイン関数です。SYCL には、さまざまな入力範囲で明確に定義された精度を備えた豊富な数学関数のセットが含まれています。ほとんどの GPU はこれらの機能をネイティブにサポートしておらず、代替えの長い命令シーケンスを使用して実装します。通常、数学関数の実装は GPU 用に最適化されていますが、アプリケーションが低い精度を許容する場合、低精度でもパフォーマンスが高い実装を検討する必要があります。SYCL ビルトイン関数の詳細については、第 18 章を参照してください。

一般に使用される数学関数の場合、SYCL ライブラリーには、精度要件が低減された、または実装定義の高速関数バリアントまたはネイティブ関数バリアントが含まれています。一部の GPU では、これらの関数は正確なバージョンの関数よりも桁違いに高速になるため、アルゴリズムに十分な精度がある場合は検討する価値があります。たとえば、多くのイメージ後処理アルゴリズムでは、入力が明確に定義されており低い精度でも許容できるため、高速またはネイティブの数学関数を使用するのに適しています。

アルゴリズムが低い精度を許容できる場合、より小さなデータ型や低精度の数学関数を使用してパフォーマンスを向上できます。

特殊機能と拡張機能

GPU 向けにカーネルを最適化する最後の考慮事項は、多くの GPU に共通する特殊な命令です。一例として、ほぼすべての GPU は、1 クロックで 2 つの演算を実行する `mad` または `fma` 乗算加算命令をサポートしています。GPU コンパイラーは一般に、個別の乗算と加算を識別して最適化し、代わりに単一の命令を使用するのが得意ですが、SYCL には明示的に呼び出すことができる `mad` 関数と `fma` 関数も含まれています。もちろん、GPU コンパイラーの乗算と加算の最適化を期待する場合は、浮動小数点の縮約を無効にすることで、最適化を妨げないようにする必要があります。

他の特殊な GPU 命令は、コンパイラーの最適化、SYCL 言語の拡張、または低レベルの GPU バックエンドとの直接対話によってのみ利用できる場合があります。たとえば、一部の GPU は、コンパイラーが識別して最適化する、または直接呼び出される可能性のある特殊なドット積累算命令をサポートしています。GPU 実装でサポートされている拡張機能を照会する方法については第 12 章を参照してください。バックエンドの相互運用性については第 20 章を参照してください。

まとめ

この章では、まず、一般的な GPU がどのように動作するか、また GPU が従来の CPU とどのように違うか説明しました。単一の命令ストリームを高速化するプロセッサー機能を追加のプロセッサーと置き換えるすることで、GPU が大量のデータに対して最適化される方法について説明しました。

GPU がワイド SIMD 命令を使用して複数のデータ要素を並列に処理する方法と、GPU が条件付きとマスクを使用した SIMD 命令により複雑なフロー制御を備えたカーネルを実行する方法について説明しました。条件付きとマスクによって SIMD 効率が低下し、カーネルのパフォーマンスが低下する可能性があること、また、ある次元と別の次元のどちらを並列化するか選択することで SIMD の分岐が軽減される可能性があることについても説明しました。

GPU には多くの処理リソースがあるため、占有率を高く保つには GPU に十分なワークを与えることが重要であると分かりました。また、GPU が命令ストリームを使用してレイテンシーを隠匿する方法についても説明しました。これにより、GPU で実行するワークを大量に与えることがさらに重要になります。

次に、カーネルを GPU にオフロードする際に関係するソフトウェア・レイヤーとハードウェア・レイヤー、およびオフロードのコストについて説明しました。あるデバイスから別のデバイスに実行を転送するよりも、単一のデバイスでアルゴリズムを実行する方が効率的である可能性があることについて説明しました。

最後に、カーネルを GPU 上で実行するベスト・プラクティスについて説明しました。多くのカーネルがメモリー依存であること、グローバルメモリーとローカルメモリーに効率的にアクセスする方法、またはサブグループ操作によりローカルメモリーを完全に排除する方法について説明しました。また、カーネルが計算依存である場合、低い精度と引き換えに高いパフォーマンスを得るか、カスタム GPU 拡張機能を使用して特殊な命令にアクセスすることで、計算を最適化する方法について説明しました。

関連情報

GPU プログラミングについては学ぶべきことは、まだたくさんあります。この章で紹介したのはほんの表面的な内容にすぎません。

GPU の仕様とホワイトペーパーは、特定の GPU と GPU アーキテクチャーを詳しく知る有益なリソースです。多くの GPU ベンダーは、自社の GPU とプログラミング方法に関する非常に詳細な情報を提供しています。

本稿執筆時点では、主要な GPU に関する関連情報は `software.intel.com`、`devblogs.nvidia.com`、`amd.com` でご覧いただけます。

一部の GPU ベンダーは、オープンソース・ドライバーまたはドライバー・コンポーネントを提供しています。利用可能であれば、ドライバーコードを検査またはステップ実行して、どの操作にコストがかかっているか、またはアプリケーションのどこにオーバーヘッドが存在する可能性があるかを把握することも有益です。

この章では、バッファーアクセサーまたは統合共有メモリーを介したグローバルメモリーへの従来のアクセス方法に重点を置きましたが、ほとんどの GPU には、イメージ操作を高速化する固定機能のテクスチャー・サンプラーも搭載されています。イメージとサンプラーの詳細については、SYCL 仕様を参照してください。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 16 章　CPU 向けのプログラミング

カーネル・プログラミングは本来、GPU をプログラムする方法として人気になりました。カーネル・プログラミングが一般化されるにつれ、カーネルスタイルのプログラミングが CPU へのマッピングにどのように影響するか理解することが重要です。

CPU は長年にわたって進化してきました。2005 年頃、クロック速度の向上によるパフォーマンスの向上が減速し、大きな変化が起こりました。フリーランチは終わったとも比喩されました。並列処理がそれに対する解決策として登場し、CPU メーカーはクロック速度を上げる代わりに、マルチコアチップの導入に舵を切りました。これにより、コンピューターは複数のタスクを効率良く同時実行できるようになりました。

マルチコアはハードウェアのパフォーマンスを向上させる方法として普及しましたが、ソフトウェアでそれに対応するには大きな努力が必要でした。マルチコア・プロセッサーでは、ハードウェアの改善をフルに活用するよう開発者はさまざまなアルゴリズムを考案する必要がありましたが、これは必ずしも簡単ではありませんでした。コア数が増えるほど、それらを効率良くビジー状態に保つことが難しくなります。SYCL は、このような課題に対処するプログラミング言語の 1 つであり、CPU（およびその他のアーキテクチャー）上のさまざまな形式の並列処理を活用するのに役立つ多くの構造を備えています。

この章では、CPU アーキテクチャーの詳細、CPU ハードウェアが SYCL アプリケーションをどのように実行するか説明し、CPU プラットフォーム向けに SYCL コードを作成する際のベスト・ プラクティスを紹介します。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_16

パフォーマンスに関する注意事項

SYCL はアプリケーションを並列化したり、並列アプリケーションをゼロから開発するための移植可能なパスを提供します。CPU 上で実行されるアプリケーションのパフォーマンスは、主に次の要因に依存します:

- カーネルコードの起動と実行の基礎となるパフォーマンス
- 並列カーネルで実行されるプログラムの比率とそのスケーラビリティー
- CPU 利用率、効果的なデータ共有、データの局所性、負荷分散
- ワーク項目間の同期と通信の量
- ワーク項目が実行されるスレッドを生成、再開、管理、一時停止、破棄、同期するためのオーバーヘッド。これは、シリアルからパラレル、またはパラレルからシリアルへ遷移する回数によって影響を受けます
- 共有メモリーで発生するメモリー競合（偽共有も含む）
- メモリー、書き込み結合バッファー、メモリー帯域幅など共有リソースのパフォーマンス制限

さらに、他のプロセッサー・タイプと同様に、CPU はベンダーごとに、また製品の世代ごとに異なる場合があります。特定の CPU のベスト・プラクティスが、別の CPU や構成のベスト・プラクティスであるとは限りません。

CPU で最適なパフォーマンスを実現するには、CPU アーキテクチャーの特性を可能な限り多く理解する必要があります。

マルチコア CPU の基本

マルチコア CPU の出現と急速な進歩により、共有メモリー型の並列計算プラットフォームが大幅に普及しました。CPU は、ラップトップ、デスクトップ、サーバーのレベルで並列計算プラットフォームを提供し、あらゆる場所で利用され、その全てでパフォーマンスを発揮します。最も一般的な CPU アーキテクチャーは、キャッシュ・コヒーレントな非均一メモリーアクセス（cc-NUMA）であり、メモリーアクセス時間が完全に均一ではないことが特徴です。多くのデュアルソケット汎用 CPU システムには、この種のメモリーシステムが搭載されています。プロセッサー内のコア数とソケット数が増加し続けており、このアーキテクチャーが主流になってきました。

cc-NUMA CPU システムでは、各ソケットはシステム内の合計メモリーのサブセットに接続します。キャッシュ・コヒーレントな相互接続により、すべてのソケットが結合され、プログラマーには単一のシステムメモリーとして見えます。
このようなメモリーシステムは、システム内のソケット数に応じて総メモリー帯域幅が拡張されるためスケーラブルです。相互接続の利点は、アプリケーションがデータの所在場所に関係なく、システム内のすべてのメモリーに透過的にアクセスできることです。ただし、コストも発生します。メモリーからデータにアクセスするレイテンシーが一定ではありません（つまり、アクセス・レイテンシーが固定されません）。
レイテンシーは、データがシステム内のどこに保存されているかによって異なります。最良のケースでは、データはコードが実行されるソケットに直接接続されたメモリーから取得されます。最悪の場合、データは遠く離れたソケットに接続されたメモリーから取得する必要があり、cc-NUMA CPU システム上のソケット間の相互接続のホップ数によってメモリーアクセスのコストが増加します。

図 16-1 に、cc-NUMA メモリーを備えた汎用 CPU アーキテクチャーを示します。これは、今日の汎用マルチソケット・システムに見られるコアとメモリー・コンポーネントを含む、簡素化されたシステム・アーキテクチャーです。
この章では、この図を基に対応するコード例のマッピングを説明していきます。

最適なパフォーマンスを達成するには、システムの cc-NUMA 構成の特性を確実に理解する必要があります。たとえば、インテル の最近のサーバーではメッシュ相互接続アーキテクチャーが採用されています。この構成では、コア、キャッシュ、メモリー・ コントローラーが行と列に編成されます。システムの最高のパフォーマンス達成に取り組む場合、プロセッサーとメモリーの接続性を理解することが重要になることがあります。

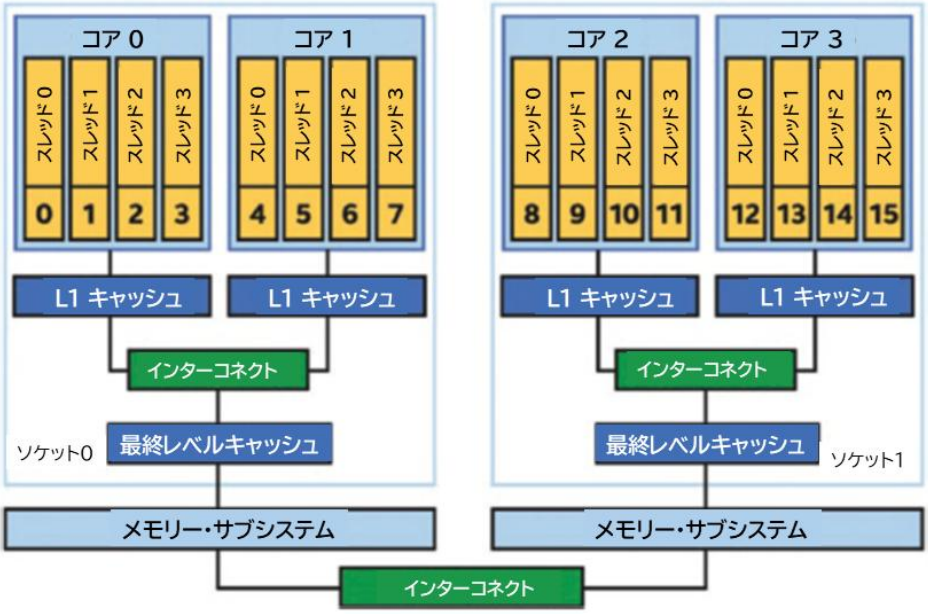

図 16-1. 一般的なマルチ CPU システム

図 16-1 のシステムは 2 つのソケットで構成され、各ソケットには 2 つのコアがあり、コアごとに 4 つのハードウェア・スレッドがあります。各コアには個別のレベル 1（L1）キャッシュがあります。L1 キャッシュは、ソケット上のメモリーシステムに接続された最終レベル共有キャッシュに接続されます。ソケット内のメモリー・アクセス・レイテンシーは均一です。つまり、一貫性があり、正確に予測できます。

2 つのソケットは、キャッシュ・コヒーレント・インターコネクトを介して接続されます。メモリーはシステム全体に分散されていますが、システム内のどこからでもすべてのメモリーに透過的にアクセスできます。コードが実行されているソケットとは異なるソケットのメモリーにアクセスする場合、メモリーの読み取りおよび書き込みレイテンシーは不均一です。つまり、リモートソケットからデータにアクセスするとレイテンシーが大幅に長くなり、一貫性がなくなる可能性があります。 ただし、インターコネクトで重要なことは一貫性です。システム全体のメモリー内の不均一なデータビューを心配する必要はなく、分散メモリーシステムへのアクセスによるパフォーマンスへの影響に集中できます。 より高度な最適化（緩和されたメモリーの順序付けを持つアトミック操作など）により、ハードウェア・メモリーの一貫性をそれほど必要としない操作が可能になりますが、一貫性が必要な場合はハードウェアがそれを提供します。

CPU 内のハードウェア・スレッドは実行の手段であり、命令ストリームを実行するユニットです。図 16-1 のハードウェア・スレッドには 0 から 15 までの番号が付けられています。これは、この章の例の説明を簡略化するための表記です。特に明記されていない限り、この章で説明する CPU システムは、図 16-1 に示す cc-NUMA システムを指します。

SIMD ハードウェアの基本

1996 年、x86 アーキテクチャーに MMX 拡張機能として SIMD 命令セットが導入されました。それ以来、多くの SIMD 命令セット拡張がインテル・アーキテクチャーと業界全体で広く展開されてきました。CPU コアは命令を実行することでその役割を果たします。コアが実行可能な具体的な命令は、実装されている命令セット（x86、x86_64、AltiVec、NEON など）と命令セット拡張（SSE、AVX、AVX-512 など）によって定義されます。命令セット拡張によって追加された操作の多くは SIMD に重点を置いています。

SIMD 命令を使用すると、データの基本単位よりも広いレジスターとハードウェアを使用して、単一のコアで複数の計算を同時に実行できます。たとえば、512 ビットのレジスターを使用すると、1 つのマシン命令で 8 つの 64 ビットの計算を実行できます。

図 16-2 に示すこの例では、理論上は最大 8 倍のスピードアップを実現できます。実際には、8 倍のスピードアップの一部が 1 つのボトルネックを解消すると同時に次のボトルネック（例えばメモリー・スループット）を露出させるため、その効果は多少抑制される可能性があります。一般に、SIMD を使用するパフォーマンス上の利点は、シナリオによって異なります。また、広範な分岐、非ユニットストライドなメモリーアクセスのギャザー/スキャッター、SIMD ロード/ストアのキャッシュライン分割などのケースでは、単純な非 SIMD コードよりもパフォーマンスが低下することもあります。しかしながら、SIMD をいつどのように適用するか（またはコンパイラーに任せる）が分かっていれば、今日のプロセッサーではかなりの性能向上を実現できます。すべてのパフォーマンス最適化と同様に、プログラマーは、運用環境に導入する前にターゲットマシンでパフォーマンスの向上を測定する必要があります。期待されるパフォーマンス向上については、以降で詳しく説明します。

```
h.parallel_for(range(1024),
               [=](id<1> k) { z[k] = x[k] + y[k]; });
```

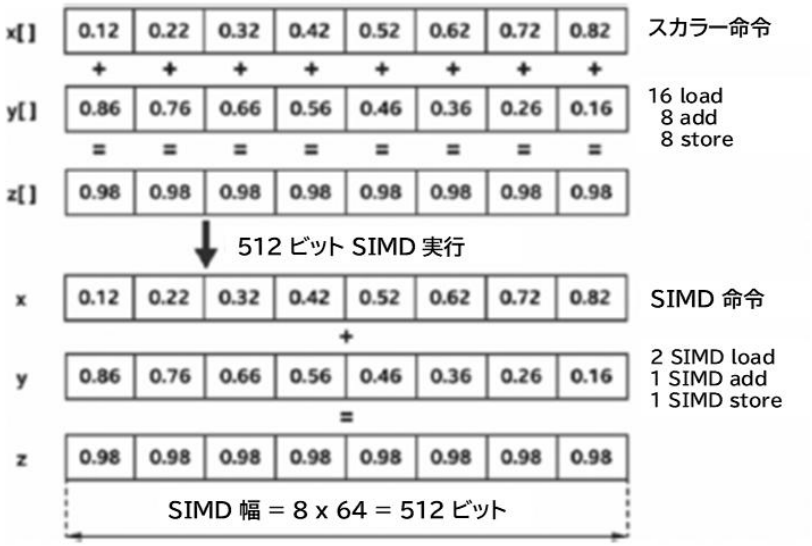

図 16-2. CPU ハードウェア・スレッドで SIMD を実行

SIMD ユニットを備えた cc-NUMA CPU アーキテクチャーは、マルチコア・プロセッサーの基盤を形成し、図 16-3 に示すように、少なくとも 5 つの異なる方法で、命令レベル並列処理から始まり幅広い並列性を活用できます。

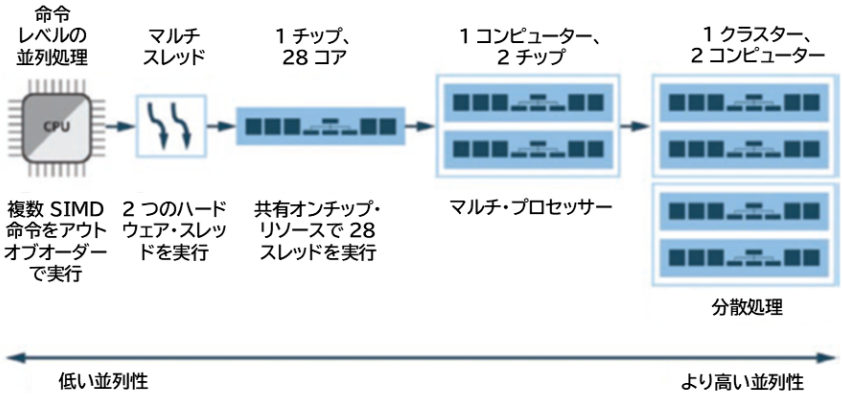

図 16-3. 命令を並列実行する 5 つの方法

図 16-3 では、スカラー命令のアウトオブオーダー実行と、単一スレッド内の SIMD 並列処理の両方によって命令レベルの並列処理を実現します。スレッドレベルの並列性は、同一コア上で複数のスレッドを実行するか、異なる規模の複数のコア上で複数のスレッドを実行することで実現できます。具体的には、スレッドレベルの並列処理は次のように展開できます:

- 最新の CPU アーキテクチャーでは、1 つのコアが 2 つ以上のスレッドの命令を同時に実行できます。

- 各プロセッサー内に 2 つ以上の*コア*を含むマルチコア・アーキテクチャー。オペレーティング・システムは、各実行コアを、すべての実行リソースを備えた個別のプロセッサーとして認識します。

- プロセッサー (チップ) レベルでのマルチプロセッシング。個別のコードスレッドを実行することで実現できます。その結果、プロセッサーは、アプリケーションから 1 つのスレッドを実行し、オペレーティング・システムから別のスレッドを実行したり、単一のアプリケーション内から並列スレッドを実行できるようになります。

- 分散処理は、通常、メッセージ・パッシング・フレームワークを介して通信するコンピューターのクラスター上で、複数のスレッドで構成されるプロセスを実行することによって実現されます。

マルチプロセッサー・コンピューターとマルチコア・テクノロジーが一般的になるにつれて、パフォーマンスを向上させるには並列処理テクノロジーを標準的な方法として使用することが重要です。この章の後半では、マルチコア CPU で最高のパフォーマンスを実現する SYCL 内のコーディング方法とパフォーマンス・チューニング手法について説明します。

他の並列処理ハードウェア（GPU など）と同様に、CPU で処理するのに十分な大きさのデータ要素のセットを提供することが重要です。大規模なデータセットを処理するマルチレベル並列処理を活用する重要性を示すため、図 16-4 に示す単純な C++ STREAM Triad プログラムを検討します。

```
// C++ STREAM Triad ワークロード
// __restrict は引数間のメモリー・エイリアシングがないことを示すために使用されます
template <typename T>
double triad(T* __restrict VA, T* __restrict VB,
             T* __restrict VC, size_t array_size,
             const T scalar) {
  double ts = timer_start();
  for (size_t id = 0;id < array_size; id++) {
    VC[id] = VA[id] + scalar * VB[id];
  }
  double te = timer_end();
  return (te - ts);
}
```

図 16-4. STREAM Triad C++ ループ

STREAM Triad ワークロードに関する注意事項

STREAM Triad ワークロード（www.cs.virginia.edu/stream）は、CPU ベンダーがメモリー帯域幅の能力を実証するために使用する、一般的なベンチマーク・ワークロードです。本章では、STREAM Triad カーネルを使用して、並列カーネルのコード生成と、この章で説明する手法によって大幅なパフォーマンスの向上を実現するようにスケジュールする方法を示します。STREAM Triad は比較的単純なワークロードですが、多くの最適化を（http://www.cs.virginia.edu/stream）で理解しやすい方法で示すには十分です。ブリストル大学のストリーム実装である Babelstream には、SYCL バージョンを含む C++ が含まれています。

STREAM Triad ループは、シリアル実行用の単一の CPU コアを使用して簡単に実行できます。優れた C++ コンパイラーはループをベクトル化し、命令レベルの SIMD 並列処理を活用するハードウェアを備えた CPU 向けの SIMD コードを生成します。たとえば、AVX-512 をサポートするインテル® Xeon® プロセッサーの場合、インテル® C++ コンパイラーは、図 16-5 に示すような SIMD コードを生成します。重要なことは、コンパイラーによるコード変換により、ループ反復ごとにワーク量を増やすことで (SIMD 命令とループアンロールを使用)、ループ反復回数が削減されることです。

```
# %bb.0:                                # %entry
vbroadcastsd    %xmm0, %zmm0            # broadcast “scalar” to SIMD reg zmm0
movq    $-32, %rax
.p2align        4, 0x90
.LBB0_1:                                # %loop.19
                                        # =>This Loop Header: Depth=1
vmovupd 256(%rdx,%rax,8), %zmm1 # load 8 elements from memory to zmm1
vfmadd213pd     256(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
                                        # perform SIMD FMA for 8 data elements
                                        # VC[id:8] = scalar*VB[id:8]+VA[id:8]
vmovupd %zmm1, 256(%rdi,%rax,8) # store 8-element result to mem from zmm1
                                        # This SIMD loop body is unrolled by 4
vmovupd 320(%rdx,%rax,8), %zmm1
vfmadd213pd     320(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
vmovupd %zmm1, 320(%rdi,%rax,8)

vmovupd 384(%rdx,%rax,8), %zmm1
vfmadd213pd     384(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
vmovupd %zmm1, 384(%rdi,%rax,8)

vmovupd 448(%rdx,%rax,8), %zmm1
vfmadd213pd     448(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
vmovupd %zmm1, 448(%rdi,%rax,8)
addq    $32, %rax
cmpq    $134217696, %rax        # imm = 0x7FFFFE0
jb      .LBB0_1
```

図 16-5. STREAM Triad C++ ループの AVX-512 アセンブリー・コード

図 16-5 に示すように、コンパイラーは命令レベルの並列処理を 2 つの方法で活用できます。1 つ目は、SIMD 命令を使用して命令レベルのデータ並列性を活用する方法です。これにより、1 つの命令で 8 つの倍精度データ要素（命令あたり）を同時に処理できます。次に、コンパイラーは、ハードウェアのマルチウェイ命令スケジューリングに基づいて、相互に依存関係のないこれらの命令のアウトオブオーダー実行の効果を得るため、ループアンロールを適用しました。

この関数を CPU 上で実行すると、配列のサイズが小さいとうまく実行される可能性がありますが、CPU のマルチコア機能やスレッド機能を活用していないため、高速には実行されません。ただし、この関数を CPU 上で大きな配列サイズで実行すると、1 つのスレッドが 1 つの CPU コアのみを使用するため、コアのメモリー帯域幅が飽和してボトルネックが発生し、パフォーマンスが大幅に低下する可能性があります。

スレッドレベルの並列処理の活用

STREAM Triad カーネルのパフォーマンスを向上させるには、ループを `parallel_for` カーネルに変換することで、並列処理できる一連のデータ要素の範囲に対して計算できます。

この STREAM Triad SYCL 並列カーネルの本体は、図 16-6 に示すように、CPU 上でシリアル C++ で実行される STREAM Triad ループの本体とまったく同じです。

```
constexpr int num_runs = 10;
constexpr size_t scalar = 3;

double triad(const std::vector<float>& vecA,
             const std::vector<float>& vecB,
             std::vector<float>& vecC) {
assert(vecA.size() == vecB.size() &&
       vecB.size() == vecC.size());
const size_t array_size = vecA.size();
double min_time_ns = std::numeric_limits<double>::max();

queue q{property::queue::enable_profiling{}};
std::cout << "Running on device: "
          << q.get_device().get_info<info::device::name>()
          << "\n";

buffer<float> bufA(vecA);
buffer<float> bufB(vecB);
buffer<float> bufC(vecC);

for (int i = 0; i < num_runs; i++) {
  auto Q_event = q.submit([&](handler& h) {
    accessor A{bufA, h};
    accessor B{bufB, h};
    accessor C{bufC, h};

    h.parallel_for(array_size, [=](id<1> idx) {
      C[idx] = A[idx] + B[idx] * scalar;
    });
  });

  double exec_time_ns =
     Q_event.get_profiling_info<
         info::event_profiling::command_end>() -
     Q_event.get_profiling_info<
         info::event_profiling::command_start>();

  std::cout << "Execution time (iteration " << i
            << ") [sec]: "
            << (double)exec_time_ns * 1.0E-9 << "\n";
   min_time_ns = std::min(min_time_ns, exec_time_ns);
 }

 return min_time_ns;
}
```

図 16-6. SYCL STREAM Triad parallel_for カーネルコード

並列カーネルは、シリアル C++ のループで記述された STREAM Triad 関数によく似ていますが、`parallel_for` によって配列の異なる*要素*を複数のコアで並列処理できるため、はるかに高速に動作します。図 16-7 は、このカーネルを CPU にマッピングする方法を示しています。システムが 1 ソケット、4 コア、コアの 2 ハードウェア・スレッド（合計 8 スレッド）を備え、実装ではそれぞれ 32 個のワーク項目を含むワークグループでデータを処理すると仮定します。
処理すべき倍精度浮動小数点数のデータ要素が 1024 個ある場合、ワークグループは 32 個必要になります。ワークグループのスケジュールは、ラウンドロビン方式で行うことができます。つまり、*スレッド* `ID = ` *ワークグループ* `ID mod 8` です。基本的に、各スレッドは 4 つのワークグループを実行し、各ラウンドで、8 つのワークグループを並行して実行できます。この場合、ワークグループは SYCL コンパイラーとランタイムによって暗黙的に形成されるワーク項目の集合であることに注意してください。

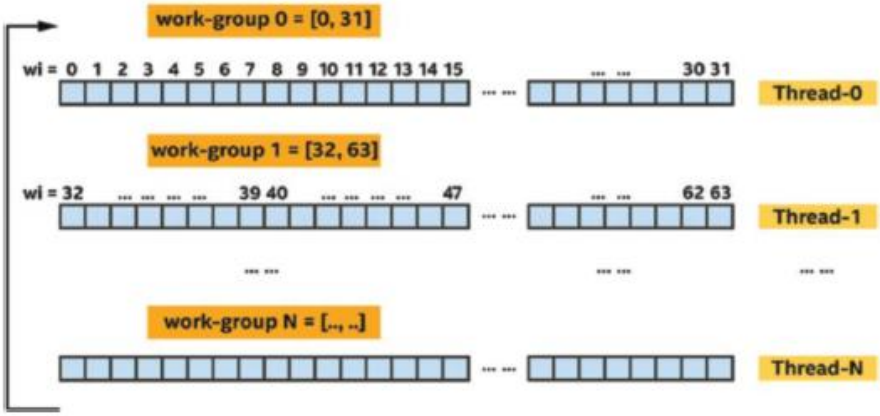

図 16-7. STREAM Triad 並列カーネルのマッピング

SYCL プログラムでは、データ要素がどのように分割され、異なるプロセッサー・コア（またはスレッド）に割り当てられるか、その具体的な方法は指定されないことに注意してください。これにより、SYCL の実装は、特定の CPU 上で並列カーネルを実行する最適な方法を選択できます。

とはいえ、実装によっては、プログラマーがパフォーマンスをチューニングできるように制御機能を提供する場合があります（例えば、コンパイラーのオプションや環境変数など）。

CPU は、比較的コストのかかるスレッドのコンテキスト・スイッチと同期オーバーヘッドを持つ可能性がありますが、プロセッサー・コアに常駐するソフトウェア・スレッド数を増やすことは、各プロセッサー・コアが実行するワークを選択できるため、有益である可能性があります。ソフトウェア・スレッドが、別のスレッドからのデータ生成を待機している場合、プロセッサー・コアをアイドル状態にすることなく、実行準備が整った別のソフトウェア・スレッドに切り替えることができます。

SYCL STREAM Triad の parallel_for カーネルコード

スレッド間でワークを分割、およびスケジュールする効果的な方式を選択することは、CPU やその他のデバイスタイプ上でアプリケーションを最適化するのに重要です。以降のセクションでは、いくつかの手法を説明します。

スレッド・アフィニティーの調査

スレッド・アフィニティーとは、スレッドが実行される CPU コアを指定するものです。スレッドが複数のコア間を移動すると、パフォーマンスが低下する可能性があります。たとえば、スレッドが同じコアにない場合、データは異なるコア間を移動するため、キャッシュの局所性が低下します。

DPC++ コンパイラーのランタイム・ライブラリーは、SYCL では定義されていない環境変数 `DPCPP_CPU_CU_AFFINITY`、`DPCPP_CPU_PLACES`、`DPCPP_CPU_NUM_CUS`、および `DPCPP_CPU_SCHEDULE` によりスレッドをコアにバインドするいくつかの方法をサポートしています。他の実装でも同様の環境変数が定義されている可能性があります。

1 つ目は、環境変数 `DPCPP_CPU_CU_AFFINITY` です。これらの環境変数による制御を用いたチューニングは、シンプルかつ低コストでありながら、多くのアプリケーションに影響する可能性があります。この環境変数の説明を図 16-8 に示します。

DPCPP_CPU_CU_AFFINITY	説明
spread	ソケット 0 から始めて、ラウンドロビン方式で連続するスレッドを それぞれ異なるソケットにバインドします
close	ソケット 0 から始めて、ラウンドロビン方式で連続するスレッドを それぞれ異なるハードウェア・スレッドにバインドします

図 16-8. DPCPP_CPU_CU_AFFINITY 環境変数

環境変数 DPCPP_CPU_CU_AFFINITY が指定されている場合、ソフトウェア・スレッドは次の式によってハードウェア・スレッドにバインドされます。

$$spread:\ boundHT=(tid \bmod numHT)+(tid \bmod numSocket)\times numHT)$$
$$close:\ boundHT=tid \bmod (numSocket\times numHT)$$

ここで、

- tid はソフトウェア・スレッド識別子を表します
- boundHT は、スレッド tid がバインドされているハードウェア・スレッド（論理コア）を表します
- numHT はソケットあたりのハードウェア・スレッド数を表します
- numSocket はシステム内のソケット数を表します

デュアルコア・デュアルソケット・システム上で、8 つのスレッドを持つプログラムを実行すると仮定すると、合計 8 つのスレッドを持つ 4 つのコアをプログラムするということになります。図 16-9 は、異なる DPCPP_CPU_CU_AFFINITY 設定に対して、スレッドがハードウェア・スレッドとコアにどのようにマッピングされるか示しています。

DPCPP_CPU_CU_AFFINITY	ソケット 0		ソケット 1	
	コア 0	コア 1	コア 2	コア 3
spread	<T0, T4>	<T2, T6>	<T1, T5>	<T3, T7>
close	<T0, T1>	<T2, T3>	<T4, T5>	<T6, T7>

図 16-9. ハードウェア・スレッドを使用してスレッドをコアにマッピング

環境変数 DPCPP_CPU_CU_ AFFINITY と組み合わせて、CPU のパフォーマンス・チューニングをサポートする他の環境変数もあります:

- DPCPP_CPU_NUM_CUS = [n]、カーネルの実行に使用するスレッド数を設定します。デフォルト値は、システム内のハードウェア・スレッド数です。

- DPCPP_CPU_PLACES = [sockets | numa_domains | cores | threads] は、OpenMP 5.1 の OMP_PLACES と同様に、アフィニティーが設定される場所を指定します。デフォルト設定は cores です。

- DPCPP_CPU_SCHEDULE = [dynamic | affinity | static] は、ワークグループのスケジュール設定アルゴリズムを指定します。デフォルト設定は dynamic です。

  dynamic: auto_partitioner を有効にします。これは通常、ワーカースレッド間の負荷分散に十分な分割を行います。

  affinity: キャッシュ・アフィニティーを改善し、サブ範囲をワーカースレッドにマッピングするときに比例分割を使用する、affinity_partitioner を有効にします。

  static: static_partitioner を有効にします。これにより、反復処理がワーカースレッド間で可能な限り均一に分散されます。

インテルの OpenCL CPU ランタイムを使用する CPU 上で実行する場合、ワークグループのスケジュールはスレッディング・ビルディング・ブロック (TBB) ライブラリーで処理されます。DPCPP_CPU_SCHEDULE を使用すると、どの TBB パーティショナーを使用するか決定できます。TBB パーティショナーは、ワーク分割を制御するため粒度も使用しており、デフォルトの粒度は 1 で、これはすべてのワークグループを独立して実行できることを意味します。詳細については、tinyurl.com/oneTBBpart をご覧ください。

スレッド・アフィニティーの調整が無くても、パフォーマンスが低下するとは限りません。パフォーマンスは、スレッドとデータの関連性や結合の度合いよりも、並列実行されるスレッドの総数に大きく左右されます。ベンチマークによりアプリケーションをテストすることは、スレッド・アフィニティーがパフォーマンスに影響するかどうか確実に判断する一つの方法です。図 16-1 に示すシステムでは、STREAM Triad コードは、スレッド・アフィニティーの設定がない場合、パフォーマンスは高くはありませんでした。アフィニティーを制御し、環境変数によりソフトウェア・スレッドの静的スケジュールを使用することで、パフォーマンスが向上しました:

```
export DPCPP_CPU_PLACES=numa_domains (Linux)
export DPCPP_CPU_CU_AFFINITY=close (Linux)

set DPCPP_CPU_PLACES=numa_domains (Windows)
set DPCPP_CPU_CU_AFFINITY=close (Windows)
```

アフィニティーに numa_domains を指定することで、TBB タスクアリーナは NUMA ノード、またはソケットにバインドされ、ワークはタスクアリーナ全体に均等に分散されます。一般に、環境変数 DPCPP_CPU_PLACES は、DPCPP_CPU_CU_AFFINITY と共に使用することを推奨します。これらの環境変数を設定することで、2.5GHz で動作する 2 ソケット、ソケットあたり 28 コア、コアあたり 2 ハードウェア・スレッドのインテル® Xeon® サーバーシステムで、約 30% のパフォーマンス向上を実現できます。しかし、この CPU のパフォーマンスをさらに向上させる改善の余地はあります。

メモリーへのファーストタッチに注意

メモリーは、最初にアクセスされた場所（使用された場所）に保存されます。この例の初期化ループは、ホストスレッドでシリアルに実行されるため、すべてのメモリーはホストスレッドが実行されているソケットに関連付けられます。その後、他のソケットからアクセスされると、初期化に使用された最初のソケットのメモリーのデータにアクセスするため、パフォーマンスの観点からは明らかに好ましくありません。図 16-10 に示すように、初期化ループを並列化してソケット間でファーストタッチ効果を制御することで、STREAM Triad カーネルのパフォーマンスを向上できます。

```
template <typename T>
void init(queue& deviceQueue, T* VA, T* VB, T* VC,
          size_t array_size) {
   range<1> numOfItems{array_size};

   buffer<T, 1> bufferA(VA, numOfItems);
   buffer<T, 1> bufferB(VB, numOfItems);
   buffer<T, 1> bufferC(VC, numOfItems);

   auto queue_event = deviceQueue.submit([&](handler& cgh) {
     auto aA = bufA.template get_access<sycl_write>(cgh);
     auto aB = bufB.template get_access<sycl_write>(cgh);
     auto aC = bufC.template get_access<sycl_write>(cgh);

     cgh.parallel_for<class Init<T>>(numOfItems, [=](id<1> wi) {
       aA[wi] = 2.0;
       aB[wi] = 1.0;
       aC[wi] = 0.0;
     });
   });

   queue_event.wait();
}
```

図 16-10. STREAM Triad 並列初期化カーネルによるファーストタッチ効果の制御

初期化コードに並列処理を活用することで、CPU 上でカーネルを実行した際のパフォーマンスが向上します。この例では、インテル® Xeon® プロセッサー・システムにおいて約 2 倍のパフォーマンス向上を実現しました。

本章のこれまでの節では、スレッドレベルの並列性を活用することで、CPU コアとスレッドを効果的に利用できることを示してきました。しかし、最高のパフォーマンスを達成するには、CPU コア・ハードウェアによる SIMD ベクトルレベルの並列性も活用する必要があります。

SYCL の並列カーネルは、コア間およびハードウェア・スレッド間でのスレッドレベルの並列処理の恩恵を得られます。

CPU 上の SIMD ベクトル化

適切に記述された、ワーク項目間で依存関係のない SYCL カーネルは、CPU 上で効率良く並列実行できますが、実装によっては、第 15 章で説明した GPU サポートと同様に、SIMD ハードウェアを活用するため SYCL カーネルにベクトル化を適用できます。基本的に、CPU プロセッサーでは、ほとんどのデータ要素が連続したメモリー領域にあり、データ並列カーネル内では同じ制御フロー経路をたどるという事実を活用することで、SIMD 命令を用いたメモリーのロード、ストア、および演算を最適化できます。例えば、`a[i] = a[i] + b[i]` ステートメントを持つカーネルでは、各データ要素は、複数のデータ要素間でハードウェア・ロジックを共有し、それらをグループ化して実行することにより、同じ命令ストリーム *load*、*load*、*add*、および *store* を実行できます。これは、ハードウェアの SIMD 命令セットに自然にマッピングできます。具体的には、単一の命令で複数のデータ要素を同時に処理できます。

単一の命令で同時に処理されるデータ要素の数は、命令またはそれを実行するプロセッサーのベクトル長 (または SIMD 幅) と呼ばれます。図 16-11 では、命令ストリームは 4 ウェイ SIMD で実行されます。

Serial execution				SIMD execution
work-0	*work-1*	*work-2*	*work 3*	*vector sub-group*
load r0, a[0]	load r0, a[1]	load r0, a[2]	load r0, a[3]	simdload vr0, a[0…3]
load r1, b[0]	load r1, b[1]	load r1, b[2]	load r1, b[3]	simdload vr1, b[0…3]
add r0, r1	add r0, r1	add r0, r1	add r0, r1	simdadd vr0, vr1
store a[0], r0	store a[1], r0	store a[2], r0	store a[3], r0	simdstore a[0…3], vr0

図 16-11. SIMD 実行の命令ストリーム

SIMD 命令セットを実装するのは CPU プロセッサーだけではありません。GPU など他のプロセッサーでも、大量データの処理効率を向上させるため SIMD 命令セットが実装されています。インテル® Xeon® CPU プロセッサーと他のプロセッサー・タイプの重要な違いは、可変長の SIMD レジスター幅ではなく、128 ビット XMM、256 ビット YMM、512 ビット ZMM の 3 つの固定サイズ SIMD レジスター幅を備えている点です。サブグループ型またはベクトル型を使用して SIMD 並列処理で SYCL コードを記述する場合（第 11 章を参照）、ハードウェア内の SIMD 幅と SIMD ベクトルレジスター数に注意する必要があります。

SIMD 実行の合法性を確保

セマンティクス的に言えば、SYCL 実行モデルは、SIMD 実行をあらゆるカーネルに適用することを保証し、各ワークグループ内のワーク項目のセット（つまりサブグループ）を SIMD 命令で並列に実行できるようにします。実装によっては、カーネル内のループで SIMD 命令を実行する場合もありますが、これは元のデータ依存性がすべて維持されている場合、またはプライベート化とリダクション・セマンティクスに基づいてコンパイラーによってデータ依存性が解決される場合に限って可能です。このような実装では、サブグループのサイズが 1 である可能性があります。

単一の SYCL カーネル実行は、ワークグループ内の SIMD 命令を使用することで、単一のワーク項目から一連のワーク項目の処理へ変換できます。ND-range モデルでは、コンパイラーのベクタライザーは、SIMD コードを生成するため最も急速に変化する（ユニットストライドの）次元を選択します。

基本的に、ND-range でベクトル化を有効にするには、同じサブグループ内の任意の 2 つのワーク項目間に依存関係が存在しないか、コンパイラーが同じサブグループ内のワーク項目間の前方依存関係を維持する必要があります。

カーネルのワーク項目の実行が CPU 上のスレッドにマッピングされる場合、細粒度の同期はコストが高く、スレッドのコンテキスト・スイッチのオーバーヘッドも大きくなることが知られています。したがって、CPU 向けの SYCL カーネルを作成する場合、ワークグループ内のワーク項目間の依存関係を排除することが重要なパフォーマンス最適化となります。もう 1 つの効果的なアプローチは、図 16-12 の書き込み前の読み取り依存関係に示すように、依存関係をサブグループ内のワーク項目に限定することです。サブグループが SIMD 実行モデルで実行される場合、カーネル内のサブグループ・バリアはコンパイラーによって NOP 操作として扱われるため、実行時に同期コストは発生しません。

```
const int n = 16, w = 16;

queue q;
range<2> G = {n, w};
range<2> L = {1, w};

int *a = malloc_shared<int>(n * (n + 1), q);

for (int i = 0; i < n; i++)
  for (int j = 0; j < n + 1; j++) a[i * n + j] = i + j;

q.parallel_for(.parallel_for(
     nd_range<2>{G, L},
    [=](nd_item<2> it) [[sycl::reqd_sub_group_size(w)]] {
    // 16 ウェイの冗長計算でサブグループ全体に
    // 均一な "i" を分配
    const int i = it.get_global_id(0);
    sub_group sg = it.get_sub_group();

    for (int j = sg.get_local_id()[0]; j < n; j += w) {
      // ループキャリー・フォワード依存関係を維持するため、
      // a[i*n+j:16] を更新する前に a[i*n+j+1:16] をロード
      auto  va = a[i * n + j + 1];
      group_barrier(sg);
      a[i * n + j] = va + i + 2;
    }
    group_barrier(sg);
   })
.wait();
```

図 16-12. サブグループを使用して、前方依存性のあるループをベクトル化

カーネルはベクトル化されており（ベクトル長 *8* を使用）、その SIMD 実行を図 16-13 に示します。ワークグループはグループサイズが (1、8) で形成され、カーネル内のループ反復はこれらのサブグループのワーク項目に分散され、8 ウェイ SIMD 並列処理で実行されます。

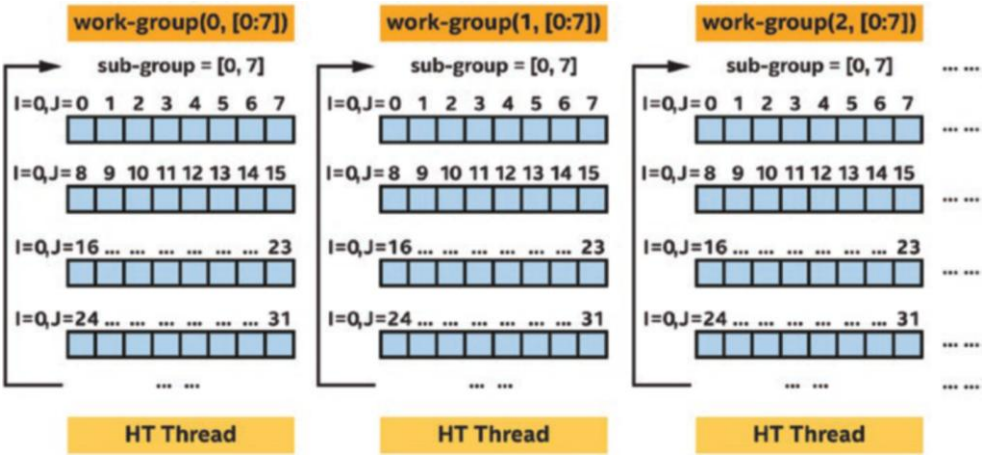

図 16-13. 前方依存性を持つループの SIMD ベクトル化

この例では、カーネル内のループがパフォーマンスに影響している場合、サブグループ全体で SIMD ベクトル化を許容することでパフォーマンスが大幅に向上します。

データ要素を並列処理する SIMD 命令を使用することは、カーネルのパフォーマンスを CPU コア数やハードウェア・スレッド数を超えて拡張する一つの方法です。

SIMD マスクとコスト

実際のアプリケーションでは、`if` 文などの条件文、`a = b > a ? a: b` のような条件式、可変回数の反復を含むループ、`switch` 文などがあります。条件付きの構文は、スカラー制御フローが同じコードパスを実行しない原因となり、GPU（第 15 章）での説明と同様に、パフォーマンスの低下につながる可能性があります。SIMD マスクとは、カーネル内の条件文から生成される、値が `1` または `0` のビットの集合です。 `A={1, 2, 3, 4}`、`B={3, 7, 8, 1}` で比較式 `a < b` の例を考えます。

比較により、ハードウェア・マスク・レジスターに格納される 4 つの値 `{1, 1, 1, 0}` を持つマスクが返され、比較によって保護された（有効化された）コードを後続の SIMD 命令のどのレーンで実行するかが指示します。

カーネルに条件付きコードが含まれている場合、各データ要素（SIMD 命令のレーン）に関連付けられたマスクビットに基づいて実行されるマスク命令を用いてベクトル化されます。各データ要素のマスクビットは、マスクレジスター内の対応するビットです。

マスクを使用すると、マスクなしのコードに比べパフォーマンスが低下します。これは以下の原因が考えられます。

- ロードごとのマスクブレンド操作
- デスティネーションへの依存関係

マスクにはコストが伴うため必要な場合にのみ使用します。カーネルがワーク項目の明示的なグループ化を備えた ND-range カーネルである場合、マスクコストを最小限に抑えて SIMD 効率を最大化するため、ND-range のワークグループ・サイズ選択には注意が必要です。ワークグループのサイズがプロセッサーの SIMD 幅で均等に割り切れない場合、ワークグループの一部がカーネルのマスクを使用して実行される可能性があります。

図 16-14 は、マージマスクを使用する際にどのようにデスティネーション・レジスターと依存関係が生じるか示しています:

- マスクがない場合、プロセッサーは 1 サイクルあたり 2 回の乗算（`vmulps`）を実行します。
- マージマスクでは、乗算命令（`vmulps`）が結果をデスティネーション・レジスターに保存するため、プロセッサーは 4 サイクルごとに 2 回の乗算を実行します（図 16-17 参照）。
- ゼロマスクでは、デスティネーション・レジスターと依存関係を持たないため、1 サイクルあたり 2 回の乗算（`vmulps`）を実行できます。

No Masking	Merge Masking	Zero Masking
vmulps zmm0, zmm6, zmm8 vmulps zmm1, zmm7, zmm8	vmulps zmm0{k1}, zmm6, zmm8 vmulps zmm1{k1}, zmm7, zmm8	vmulps zmm0{k1}{z}, zmm6, zmm8 vmulps zmm1{k1}{z}, zmm7, zmm8
Baseline	Slowdown 4x	Slowdown 1x

図 16-14. カーネルでマスクを行う 3 つのマスクコード

アライメントされたデータのキャッシュへのアクセスは、アライメントされていないデータへのアクセスより高いパフォーマンスをもたらします。多くの場合、アドレスはコンパイル時に不明であるか、判明していてもアライメントされていません。ループでは、メモリーアクセスのピーリングを実装して、アラインメントされたアドレスに達するまでマスクアクセスにより先頭からいくつかの要素を処理し、その後、マルチバージョン技術を使用してマスクされていないアクセスと、マスクされたアクセスで残りを処理ができます。この方式はコードサイズを増加させますが、データ処理全体のパフォーマンスを改善します。並列カーネルでは、プログラマーは同様の手法を手作業で適用したり、割り当てが適切にアラインメントされるようにすることでパフォーマンスを向上できます。

SIMD 効率のために構造体配列を避ける

AOS（配列構造体）は、ギャザーとスキャッターを引き起こすため、SIMD の効率性が低下するだけでなく、メモリーアクセスの帯域幅とレイテンシーが増加します。ハードウェアでギャザー/スキャッター・メカニズムが利用できる場合であっても、このデータ構造の変更は必要です。一般にギャザー/スキャッター操作は、連続したロードと比べて非常に多くの帯域幅とレイテンシーが要求されます。`struct {float x; float y; float z; float w;} a[4]` という AOS データレイアウトの場合、図 16-15 に示すように動作するカーネルを考えます。

```
cgh.parallel_for<class aos<T>>(numOfItems,[=](id<1> wi) {
  x[wi] = a[wi].x;     // x0、x1、x2、x3 を集める
  y[wi] = a[wi].y;     // y0, y1, y2, y3 を集める
  z[wi] = a[wi].z;     // z0, z1, z2, z3 を集める
  w[wi] = a[wi].w;     // w0, w1, w2, w3 を集める
});
```

図 16-15. カーネルの SIMD ギャザー

コンパイラーがワーク項目に沿ってカーネルをベクトル化すると、ユニットストライドではないメモリーアクセスが必要になるため、SIMD ギャザー命令が生成されます。例えば、最も効率的なユニットストライドは 1 ですが、`a[0].x`、`a[1].x`、`a[2].x`、`a[3].x` のストライドは 4 です。

w_3	z_3	y_3	x_3	w_2	z_2	y_2	x_2	w_1	z_1	y_1	x_1	w_0	z_0	y_0	x_0

カーネルでは、メモリーのギャザー/スキャッター操作を排除することで、SIMD 効率を向上できる場合が多くあります。一部のコードは、配列構造体（AOS）で記述されたデータ構造を、構造体配列（SOA）表現へのデータレイアウト変換によってメリットを得られます。つまり、SIMD ベクトル化のメモリーアクセスが連続するように、構造体のフィールドごとに個別の配列を配置します。
例えば、以下に示すような

```
struct {
    float x[4];
    float y[4];
    float z[4];
    float w[4];
} a;
```

という SOA データレイアウトを考えてみましょう:

w_3	w_2	w_1	w_0	z_3	z_2	z_1	z_0	y_3	y_2	y_1	y_0	x_3	x_2	x_1	x_0

カーネルは、図 16-16 に示すように、ベクトル化されたデータであっても、ユニットストライド（連続）なベクトルロードおよびストアによってデータを操作できます。

```
cgh.parallel_for<class aos<T>>(numOfItems,[=](id<1> wi) {
  x[wi] = a.x[wi]; // ユニットストライド・ベクトル・ロード x [0:4]
  y[wi] = a.y[wi]; // ユニットストライド・ベクトル・ロード y[0:4]
  z[wi] = a.z[wi]; // ユニットストライド・ベクトル・ロード z[0:4]
  w[wi] = a.w[wi]; // ユニットストライド・ベクトル・ロード w[0:4]
});
```

図 16-16. カーネル内の SIMD ユニットストライド・ベクトル・ロード

SOA データレイアウトは、配列要素をまたいで構造体の 1 つのフィールドにアクセスするギャザーを防ぐのに役立ち、コンパイラーがワーク項目に関連付けられた連続する配列要素に対してカーネルをベクトル化するのに役立ちます。なお、このような AOS から SOA、または AOSOA へのデータレイアウト変換は、データ構造が使用されているすべてのコードを考慮して、プログラムレベルで行われることが想定されます。この変更をループレベルで行った場合、ループの前後でデータ構造の変換するコストがかかります。しかし、多少のコストはかかっても、コンパイラーに AOS データレイアウトに対するベクトルロードとシャッフルの最適化をさせることもできます。SOA（または AOS）データレイアウトのメンバーがベクトル型である場合、コンパイラーのベクトル化は、ベースとなるハードウェアに基づいて水平展開、または垂直展開のいずれかにより最適なコードを生成する可能性があります。

データ型が SIMD 効率に与える影響

C++ プログラマーは、データが 32 ビット符号付き型に収まることが分かっている場合,常に整数データ型を使用することが多く、次のようなコードになることがあります。

```
int id = get_global_id(0); a[id] = b[id] + c[id];
```

しかし、`get_global_id(0)` の戻り値の型が `size_t`（符号なし整数、多くの場合 64 ビット）であることを考えると、変換によってコンパイラーが積極的に行う最適化が減少する可能性があります。これにより、コンパイラーがカーネル内のコードをベクトル化する際に、SIMD ギャザー/スキャッター命令が生成される可能性があります。例えば、次のようになります:

- `a[get_global_id(0)]` の読み取りは、SIMD ユニットストライド・ベクトルのロードにつながる可能性があります。

- `[(int)get_global_id(0)]` の読み取りは、非ユニットストライド・ギャザー命令につながる可能性があります。

この微妙な状況は、`size_t` から `int`（または `uint`）へのデータ型変換におけるラップアラウンド動作（C++ 標準では「未定義の動作」および/または「明確に定義されたラップアラウンド動作」）によって引き起こされますが、これは主に C ベースの言語の進化による歴史的な遺物です。具体的には、一部の型変換におけるオーバーフローは未定義の動作ですが、コンパイラーはそのような状況は発生しないと想定して、より積極的に最適化を行うことができます。
図 16-17 は、詳細を理解したい方向けにいくつかの例を示しています。

get_global_id(0)	a[(int)get_global_id(0)]	get_global_id(0)	a((uint)get_global_id(0)]
0x7FFFFFFE	a[MAX_INT-1]	0xFFFFFFFE	a[MAX_UINT-1]
0x7FFFFFFF	a[MAX_INT (big positive)]	0xFFFFFFFF	a[MAX_UINT]
0x80000000	a[MIN_INT (big negative)]	0x100000000	a[0]
0x80000001	a[MIN_INT+1]	0x100000001	a[1]

図 16-17. 整数型値のラップアラウンドの例

SIMD ギャザー/スキャッター命令は、SIMD ユニットストライド・ベクトル・ロード/ストア操作よりも遅くなります。最適な SIMD 効率を実現するには、プログラミング言語に関わりなく、アプリケーションでギャザー/スキャッターを回避することが重要です。

ほとんどの `SYCL get_*_id()` ファミリー関数は同じ詳細を持ちますが、戻り値には制限があるため（たとえば、ワークグループ内の最大 `ID`）、多くの場合 `MAX_INT` 内に収まります。したがって、合法であればいつでも、SYCL コンパイラーは、ギャザー/スキャッターを回避するため、隣接するワーク項目のチャンク全体でユニットストライド・メモリー・アドレスを想定できます。

グローバル ID の値および/またはグローバル `ID` からの派生値によるオーバーフローの可能性のため、コンパイラーが安全に線形ユニットストライドのベクトル・メモリー・ロード/ストアを生成できない場合、コンパイラーはギャザー/スキャッターを生成します。

ユーザーに最適なパフォーマンスを提供する目的で、DPC++ コンパイラーはオーバーフローがないものと想定して最適な SIMD コードを生成し、優れたパフォーマンスを実現できます。ただし、DPC++ コンパイラーのコンパイラー・オプション `-fno- sycl-i d-queries-fit-in-int` により、オーバーフローが発生し、`ID` クエリーから派生したベクトル化されたアクセスが安全ではない可能性があることをコンパイラーに通知できます。これはパフォーマンスに大きな影響を与える可能性があり、オーバーフローが発生しないと仮定することが安全でない場合には常に使用すべきです。重要な点は、プログラマーはグローバル `ID` の値が 32 ビット `int` に収まることを確認する必要があるということです。それ以外では、プログラムの正当性を保証するため、コンパイラー・オプション `-fno-sycl-id-queries-fit-in-int` を使用する必要がありますが、これによりパフォーマンスが低下する可能性があります。

`single_task` を使用した SIMD 実行

単一タスク実行モデルでは、ベクトル化できる複数のワーク項目はありません。ベクトル型および関数に関連する最適化は可能かもしれませんが、これはコンパイラーに依存します。コンパイラーとランタイムには、明示的な SIMD 実行を有効にするか、`single_task` カーネル内でスカラー実行を選択するか、選択の自由が与えられており、結果はコンパイラーの実装によって異なります。

C++ コンパイラーは、CPU 向けにコンパイルするときに、`single_task` 内で発生するベクトル型を SIMD 命令にマップする場合があります。`vec` のロード、ストア、およびスウィズル関数は、ベクトル変数に対して直接操作を実行し、データ要素が同じメモリー（均一な）から連続してデータにアクセスすることをコンパイラーに通知し、連続したデータに最適化されたロード/ストアを要求できるようにします。

第 11 章で説明したように、`vec` のこの解釈は有効ですが、この機能は最終的には非推奨となり、より明示的なベクトル型（`std::simd` など）が利用可能になると予想されます。

```
queue q;

bool *resArray = malloc_shared<bool>(1, q);
resArray[0] = true;

q.single_task([=]() {
   sycl::vec<int, 4> old_v = sycl::vec<int, 4>(0, 100, 200, 300);
   sycl::vec<int, 4> new_v = sycl::vec<int, 4>();

   new_v.rgba() = old_v.abgr();
   int vals[] = {300, 200, 100, 0};

  if (new_v.r() != vals[0] || new_v.g() != vals[1] ||
      new_v.b() != vals[2] || new_v.a() != vals[3]) {
     resArray[0] = false;
   }
 }).wait();
```

図 16-18. single_task カーネルでベクトル型とスウィズル操作を使用

図 16-18 に示す例では、シングルタスクの実行時に、4 つのデータ要素を持つベクトルが宣言されています。スウィズル操作は `old_v.abgr()` で実行されます。CPU がスウィズル操作に対して SIMD ハードウェア命令を提供する場合、アプリケーションでスウィズル操作を使用することでパフォーマンス上の利点が得られる可能性があります。

SIMD ベクトル化のガイドライン

CPU プロセッサーは、異なる SIMD 幅を持つ SIMD 命令セットを実装します。多くの場合、これは実装レベルの詳細であり、CPU 上でカーネルを実行するアプリケーションには透過的です。コンパイラーは、特定の SIMD 幅で処理する効率的なデータ要素のグループを決定できるため、SIMD 命令を明示的に使用する必要はありません。サブグループを使用することで、カーネル内でデータ要素のグループ化が SIMD 実行の対象となるケースをより直接的に表現できます。

計算の複雑性を考慮すると、ベクトル化に最も適したコードとデータレイアウトを選択することが、最終的に高いパフォーマンスにつながる可能性があります。データ構造を選択する際には、この章で説明されているように、最も頻繁に実行される計算が最大限の並列性で SIMD に適した方法でメモリーにアクセスできるように、データレイアウト、アラインメント、およびデータ幅を選択するようにしてください。

まとめ

CPU 上でスレッドレベルの並列処理と SIMD ベクトルレベルの並列処理を最大限に活用するには、次のことを念頭に置く必要があります:

- あらゆるタイプの SYCL 並列処理と、ターゲットとする CPU アーキテクチャーに精通している必要があります。
- ハードウェア・リソースに最適なスレッドレベルで、適切な並列処理（多すぎず少なすぎず）を活用します。これを実現するチューニング作業には、アナライザーやプロファイラーなどのパフォーマンス解析ツール使用してください。
- スレッドの親和性とメモリーのファーストタッチがプログラムのパフォーマンスに与える影響に注意してください。
- 最も頻繁に実行される計算が最大限の並列性で SIMD に適した方法でメモリーにアクセスできるように、データレイアウト、アラインメント、およびデータ幅を使用してデータ構造を設計します。
- マスクのコストとコード分岐のコストのバランスに注意してください。
- 潜在的なメモリーのエイリアシングと副作用を最小限に抑える明確なプログラミング・スタイルを使用します。
- ベクトル型とインターフェイスを使用する場合のスケーラビリティーの制限に注意してください。コンパイラーがそれらをハードウェア SIMD 命令に割り当てる場合、固定ベクトルサイズは、複数世代の CPU および異なるベンダーの CPU 間で SIMD レジスターの SIMD 幅と一致しない可能性があります。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。
（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 17 章　FPGA 向けのプログラミング

カーネルベースのプログラミングは、もともと GPU を活用する方法として普及しました。現在では多くの種類のアクセラレーターに採用されているため、私たちのプログラミング・スタイルが FPGA へのコードのマッピングにどのように影響するかを理解することも重要です。

フィールドプログラマブル・ゲート・アレイ（FPGA）は、ほとんどのソフトウェア開発者には馴染みのないものです。その理由の一つは、一般的なデスクトップ・コンピューターには、CPU や GPU に加えて FPGA が搭載されていないためです。しかし、FPGA は多くの用途で利点をもたらすため、知っておく価値はあります。他のアクセラレーターと同様に、「FPGA はいつ使用すべきか？」、「アプリケーションのどの部分を FPGA にオフロードすべきか？」、「FPGA 上で優れたパフォーマンスを発揮するコードはどのように記述すればよいか？」、といった疑問に答える必要があります。

この章では、少なくとも FPGA が私たちのアプリケーションに有益であるか判断できるレベルまで、そしてパフォーマンスを実現にはどのような構造が使用されているか、それらを理解できるレベルまでの知識を提供します。この章は出発点であり、ベンダーのドキュメントを読んで、特定の製品やツールチェーンに関する詳細を補完していくことができます。
まず、プログラムが FPGA などの空間アーキテクチャーにどのようにマッピングされるか概説し、次に FPGA がアクセラレーターとして優れた選択肢と成りうるいくつかの特性について議論し、最後にパフォーマンスの実現に使用されるプログラミング構造を紹介します。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_17

本章の “FPGA の考え方” のセクションは、あらゆる FPGA を考慮する際に適用できます。SYCL は、ベンダーが CPU や GPU 以外のデバイスを指定することを可能にしますが、FPGA をどのようにサポートするかは具体的に規定していません。本章で説明する FPGA に対するベンダー固有のサポートは、現時点では DPC++ 独自のものであり、具体的には FPGA セレクターとパイプに関連するものです。この章で使用する DPC++ 拡張機能は、FPGA セレクターとパイプのみです。ベンダー各社が FPGA をサポートする上で、類似または互換性のある手段を採用することが期待されており、DPC++ はオープンソース・プロジェクトとしてこれを奨励しています。

パフォーマンスに関する注意点

他のプロセッサーやアクセラレーターと同様に、FPGA デバイスはベンダーごとに、あるいは製品世代ごとに異なります。そのため、あるデバイスのベスト・プラクティスが、別のデバイスとは異なる場合があります。この章のアドバイスは、現在および将来にわたって多くの FPGA デバイスに役立つでしょう …

… 特定の FPGA で最適なパフォーマンスを実現するには、ベンダーのドキュメントを参照してください。

FPGA の考え方

FPGA は一般的に*空間*アーキテクチャーに分類されます。それらは、CPU や GPU など、馴染みのある命令セット・アーキテクチャー（ISA）を持つデバイスとは、全く異なるコーディング・スタイルと並列処理の形態から恩恵を受けます。FPGA の理解を深めるには、まずは ISA ベースのアクセラレーターに関するいくつかの概念を簡単に説明し、重要な違いを明確にしていきます。

本書の目的において、ISA ベースのアクセラレーターとは、デバイスが一度に 1 つまたは少数の異なる命令を実行できるものを指します。命令は通常、「アドレス A のメモリーからロードする」、「次の数値を加算する」といった比較的単純なものです。一連の操作が連結されてプログラムが形成され、プロセッサーは概念的には命令を一つずつ順番に実行していきます。

ISA ベースのアクセラレーターでは、チップの単一領域（またはチップ全体）が、クロックサイクルごとにプログラムから異なる命令を実行します。命令は、図 17-1 に示すように、異なるタイミングで、異なる命令を実行できる固定ハードウェア・アーキテクチャー上で実行されます。例えば、加算演算に使用されるメモリー・ロード・ユニットは、減算演算に使用されるメモリー・ロード・ユニットとほぼ同じである可能性があります。
同様に、加算命令と減算命令の実行には、同じ演算ユニットが使用されている可能性があることも考えられます。プログラムの実行が進行するにつれて、チップ上のハードウェアは様々な命令によって*再利用*されます。

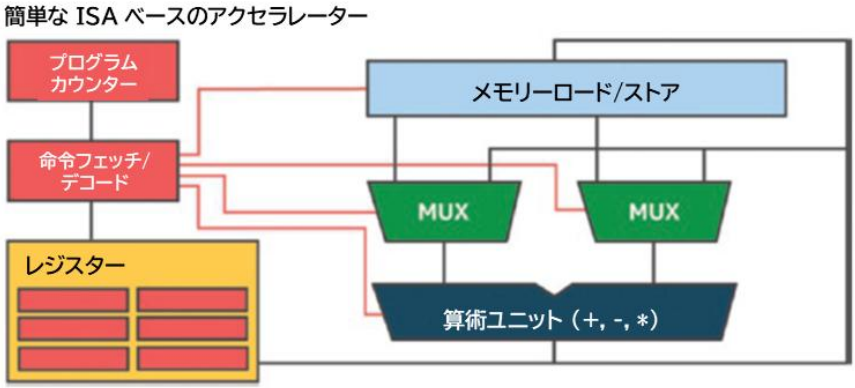

図 17-1. シンプルな ISA ベースの（時間的）処理: ハードウェア（領域）を時間的に再利用

空間的アーキテクチャーは異なります。様々な命令を共通のハードウェア上で実行するマシンではなく、まったく反対の視点から出発します。

プログラムの空間実装は、概念的にはプログラム全体を一つのまとまりとして捉え、それを一度にデバイス上に配置します。デバイスの異なる領域は、プログラム内の異なる命令を実行します。これは、命令間でハードウェアを時間的に共有する（例えば ISA）ものとは、多くの点で正反対の考え方です。空間アーキテクチャーでは、各命令は専用のハードウェアを受け取り、他の命令を実行するハードウェアと同時に（同じクロックサイクルで）実行できます。図 17-2 は、この概念を示しています。これは、プログラム全体（この例では非常に単純なプログラム）を空間的に実装したものです。

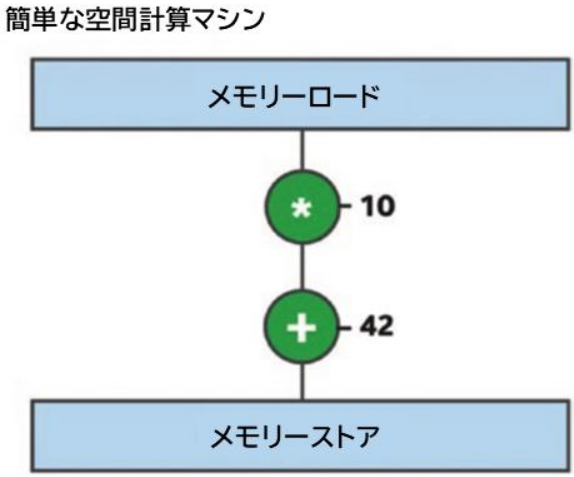

図 17-2. 空間処理: 各演算はデバイスの異なる領域を使用します

プログラムの空間実装に関する説明はかなり単純化されていますが、空間アーキテクチャーでは、プログラムの異なる部分がデバイスの異なる部分で実行されるのに対し、汎用アーキテクチャーでは共有ハードウェア・セットに次々と発行されることを示しています。

FPGA の異なる領域で異なる演算を実行するようにプログラムすることで、ISA ベースのアクセラレーターに関連付けられるハードウェアの一部が不要になります。例えば、図 17-2 では、命令フェッチやデコードユニット、プログラムカウンター、レジスターファイルが不要になったことを示しています。

空間アーキテクチャーは、将来の命令が使用するデータをレジスターファイルに格納する代わりに、命令の出力を次の命令の入力に直接接続します。そのため、空間アーキテクチャーは*データ・フロー・アーキテクチャー*と呼ばれることがよくあります。

ここで紹介する FPGA へのマッピングには、当然ながらいくつかの疑問が生じます。まず、プログラム内の各命令はデバイスの空間領域を一定の割合を占めるため、プログラムが 100% を超える領域を必要とする場合はどうなるでしょうか？一部のソリューションでは、パフォーマンスを犠牲にしてより大きなプログラムを格納するリソース共有メカニズムが提供されていますが、FPGA には*プログラム・フィッティング*の概念があります。これは利点でもあり欠点でもあります:

- **利点:** プログラムが FPGA 上の領域の大部分を占有して、すべてのハードウェアを毎クロックサイクル稼働させるのに十分なワーク量がある場合、デバイス上でプログラムを実行することは、その極めて高い並列性により非常に効率的です。汎用的なアーキテクチャーでは、クロックサイクルごとに未使用のハードウェアが存在する可能性があるのに対し、FPGA では、特定のアプリケーションに合わせて領域を無駄なく最適に割り当てることができます。このカスタマイズにより、アプリケーションは大規模な並列処理によって高速に実行可能となり、優れたエネルギー効率も実現します。

- **欠点:** 大規模なプログラムは、デバイスに収まるように調整や再構成が必要になる場合があります。コンパイラーのリソース共有機能はこの問題の対処に役立ちますが、通常、パフォーマンスが低下するため FPGA を使用する利点を損ないます。ISA ベースのアクセラレーターは、非常に効率良くリソース共有を実現する実装です。FPGA が計算処理で最大の価値を発揮するのは、主に、利用可能な領域の大部分を活用するアプリケーションの設計を構築できる場合です。

極端ケースでは、FPGA 上のリソース共有機能は、ISA ベースのアクセラレーターに似たアーキテクチャーをもたらしますが、これは固定シリコンで最適化されるのではなく、再構成可能なロジックで構築されます。再構成可能なロジックは、固定シリコン設計に比べてオーバーヘッドが生じるため、FPGA は ISA を実装する方法として選択されません。FPGA では、アプリケーションがリソースを活用して効率良いデータ・フロー・アルゴリズムを直接実装できる場合に、最大の利点を得られます。これについては、以降のセクションで詳しく説明します。

パイプラインの並列処理

図 17-2 から生じるもう 1 つの疑問は、プログラムの空間実装がクロック周波数とどのように関連しているか、そしてプログラムの開始から終了までどれくらい早く実行できるかということです。例では、メモリーからデータを読み込み、乗算と加算演算を実行し、その結果を再びメモリーに格納する処理を高速化できることが容易に想像できます。プログラムが大きくなり、FPGA デバイス全体で数万もの演算が行われるようになると、すべての命令を順番に実行するには（演算はしばしば前の演算で生成された結果に依存する）、各演算によって生じる処理遅延を考慮すると、かなりの時間がかかる可能性があることが明らかになります。

図 17-3 に示すように、操作間の中間結果は空間アーキテクチャー内で時間とともに更新（伝播）されます。例えば、ロードが実行され、その結果が乗算器に渡され、その結果が加算器に渡される、といった具合に処理が進行します。一定時間が経過すると、中間データは一連の処理の最後まで伝播し、最終結果が利用可能になるか、メモリーに保存されます。

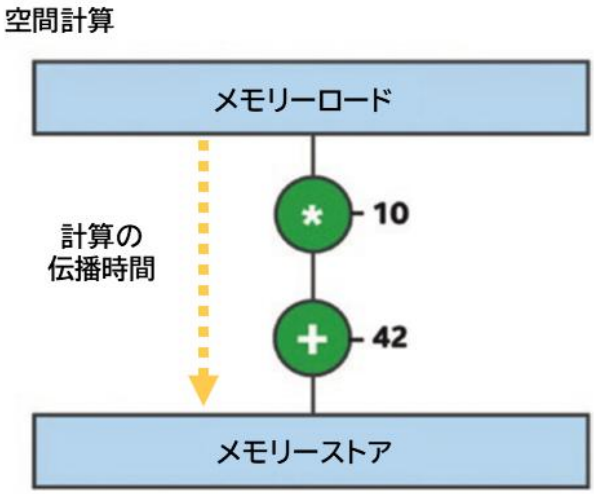

図 17-3. 自然な空間計算実装の伝播時間

図 17-3 に示すような空間実装は非常に非効率的です。なぜなら、ほとんどのハードウェアがごく一部の時間のしか有用なワークを行っていないためです。ほとんどの場合、乗算などの演算は、ロードからの新しいデータを待機しているか、チェーンの後の演算がその結果を利用できるように、自身の出力を保持しているかのどちらかです。 ほとんどの空間コンパイラーと実装は、パイプライン処理によってこの非効率性の問題を解消しています。これは、単一プログラムの実行を多くのクロックサイクルに分散させることを意味します。これは、いくつかの演算の間にレジスター（ハードウェアのデータ記憶プリミティブ）を挿入することによって実現され、各レジスターは 1 クロックサイクルの間バイナリー値を保持します。演算の出力結果を保持することで、チェーン内の次の演算がその値を参照し、それを基に演算を行うことができるため、前の演算は後続の演算への入力に影響することなく、別の計算ができます。

アルゴリズム・パイプライン処理の目的は、各クロックサイクルの大部分において、すべての演算（ハードウェア・ユニット）を常に稼働状態に維持することです。図 17-4 は、前述の例にパイプライン処理を実装したものです。これらのパイプライン処理と負荷分散は、すべてコンパイラーが自動的に行ってくれることを覚えておいてください!

このトピックを取り上げるのは、以降のセクションでパイプラインにワークをどのように組み込むかを理解するためであり、コード内で手動によりパイプライン化するためではありません。

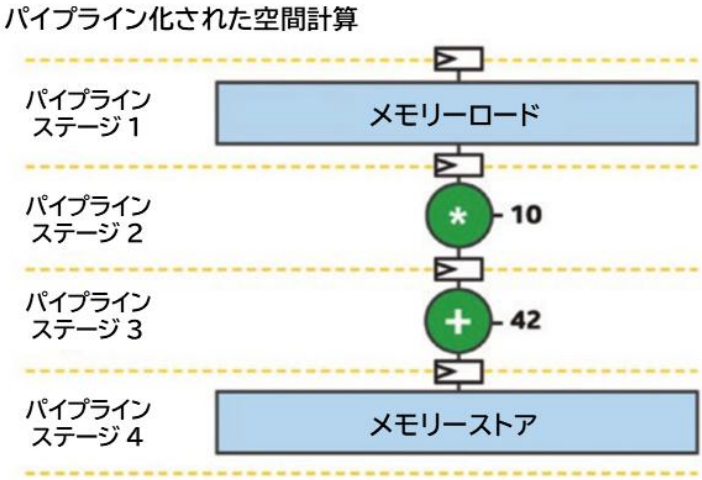

図 17-4. 計算処理のパイプライン化: 各ステージを並列に実行

空間的な実装をパイプライン化することで、工場の組立ラインと同じように効率化できます。パイプラインの各ステージはワーク全体のごく一部しか実行しませんが、迅速に実行し、直後に次のワーク単位に取り掛かります。パイプラインによる*単一*の計算処理には、開始から終了までに多くのクロックサイクルを要しますが、パイプラインは異なるデータに対して同じ計算の複数のインスタンスを同時に実行することができます。

パイプライン内で十分な数の処理が実行され、かつ十分な連続したクロックサイクルが経過すると、パイプラインの各ステージ、ひいてはプログラム内の各演算が、すべてのクロックサイクルで有効な処理を実行できるようになります。つまり、デバイス全体が同時に処理を実行します。これは空間アーキテクチャーの強みの一つであり、デバイス全体が常に並列処理を実行できます。私たちはこれを*パイプライン並列処理*と呼んでいます。

パイプライン並列処理は、FPGA 上でパフォーマンス向上を実現する主要な並列処理形態です。

パイプラインは自動的

FPGA 向け SYCL DPC++ 実装、および他の FPGA 向け高レベル・プログラミング・ソリューションでは、アルゴリズムのパイプライン処理はコンパイラーによって自動的に実行されます。本節で説明するように、空間アーキテクチャーにおける実装を理解しておくことは、パイプライン並列処理を活用するアプリケーションを構築する際に役立ちます。パイプライン・レジスターの挿入とバランス調整は、開発者が手動で行うのではなく、コンパイラーによって行われることを明確にしておく必要があります。

実際のプログラムやアルゴリズムには、制御フロー（例えば、`if/else` 構造）があり、プログラムの一部がクロックサイクルの一定割合で非アクティブ状態になります。FPGA コンパイラーは通常、制御フローの分岐で無駄な空間領域の配置を最小限に抑え、計算効率を最大化するため、可能であれば分岐の両方で同じハードウェア・リソースを共有できるように最適化を行います。これにより、制御フローの分岐は、他のアーキテクチャー、特にベクトル化アーキテクチャーに比べて、はるかにコストが低く、開発上の懸念も少なくなります。

カーネルはチップの「領域」を消費

既存の実装では、SYCL アプリケーション内の各カーネルは、FPGA のリソース（デバイス上の*空間*または*領域*）を消費する空間パイプラインを生成します。図 17-5 に概念を示します。

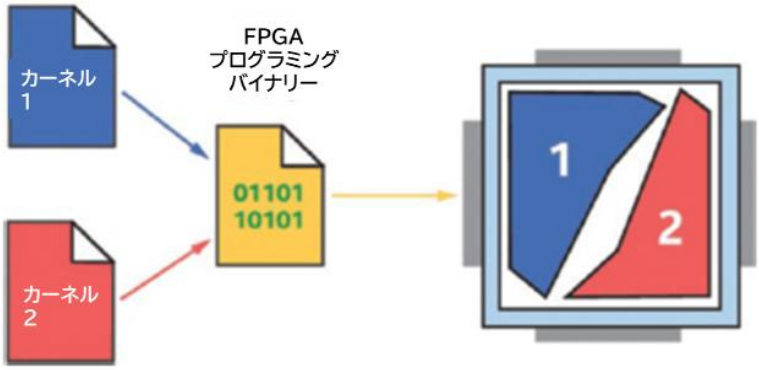

図 17-5. 同一 FPGA バイナリー内の複数のカーネル：カーネルは同時に実行可能

カーネルはデバイス上でそれぞれ専用の領域を使用するため、異なるカーネルを同時に実行できます。あるカーネルがメモリーへのアクセスを待機している場合でも、FPGA 上の他のカーネルは、チップ上の別の場所にある独立したパイプラインであるため、実行を継続できます。これは、正式にはカーネル間における独立した前進処理として説明され、FPGA の空間演算における重要な特性です。

FPGA を使用するタイミング

他のアクセラレーター・アーキテクチャと同様に、FPGA が適切なアクセラレーターであるか、あるいは他の選択肢が適切であるか判断するには、多くの場合、アーキテクチャー、アプリケーションの特性、およびシステム・ボトルネックに関する知識が必要です。このセクションでは、検討すべきアプリケーションの特性について説明します。

たくさんのワーク

他の計算アクセラレーターと同様に、優れたパフォーマンスを実現するには、膨大な量のワークが必要となります。単一のデータ要素から単一の結果を計算する場合、アクセラレーター（どのような種類でも）の活用は有用ではない可能性があります。

これは、FPGA も例外ではありません。FPGA コンパイラーがパイプライン並列処理を活用していることを知っていれば、これはさらに明確です。アルゴリズムのパイプライン実装は、通常、数千以上のステージから構成され、各ステージはクロックサイクルごとに異なる処理を実行する必要があります。大部分のパイプライン・ステージを、ほとんどの時間稼働させる十分なワークがない場合、効率は低下します。パイプラインの各ステージでの時系列の平均利用率を、パイプライン*占有率*と呼びます。これは、GPU など他のアーキテクチャーを最適化する際に使用される占有率の定義とは異なります。

FPGA のパイプライン・ステージを埋める処理を生成する方法は複数あり、それらについては以降のセクションで説明します。

カスタム操作または操作幅

FPGA は元々、効率良い整数演算とビット演算を実行し、他のチップのインターフェイスを相互に連携させる接続ロジックとして機能するように設計されました。FPGA は、単なるロジック回路の接続ではなく、強力な演算能力を持つ装置へと進化してきましたが、ビット演算、カスタムデータ幅や型に対する整数演算、パケットヘッダー内の任意のビットフィールドに対する演算などにおいては、依然として非常に効率的です。

本章の最後に説明する FPGA の詳細なアーキテクチャーにより、斬新で任意のデータ型を効率良く実装することが可能になります。例えば、33 ビットの整数乗算器や 129 ビットの加算器が必要な場合、FPGA はこれらのカスタム演算を効率良く提供できます。この柔軟性により、FPGA は、近年では人工知能など、急速に進化する分野で広く利用されています。人工知能では、データ幅や演算処理が ASIC への組み込みよりも速いペースで変化しているためです。

スカラー・データ・フロー

図 17-4 から明らかなように、FPGA 空間パイプラインで重要なことは、演算間の中間データがチップ上に保持されるだけでなく（外部メモリーには保存されない）、各パイプライン・ステージ間の中間データには専用のストレージレジスターがあることです。FPGA の並列処理は、多くの場合、計算のパイプライン化によって表現されます。つまり、多くの演算が同時に実行され、それぞれがパイプラインの異なるステージで実行されます。その結果、プログラムの算術演算が集中する領域でも、スカラー・データ・フローは（内部的に）一般的な実装となり、複数の計算が共有ベクトル命令のレーンとして実行されるベクトル・アーキテクチャーとは根本的に異なります。

空間パイプラインにおける並列処理のスカラー的な性質は、多くのアプリケーションにおいて重要です。それは、ワーク単位間でデータ依存性が高い場合でも、この性質は依然として適用されるためです。このようなデータ依存性は、パフォーマンスを損なうことなく処理できます。これについては、この章の後半でループ伝搬依存関係について説明する際に詳しく述べます。その結果、空間パイプライン、ひいては FPGA は、ワーク単位（ワーク項目など）間のデータ依存関係を解決できず、細粒度の通信が必要となるアルゴリズムにおいて魅力的なターゲット・アーキテクチャーとなります。他のアクセラレーター向けの最適化手法の多くは、複雑なアプローチによって依存関係を解消したり、サブグループなどの機能を通じて制御された通信を管理することに重点を置いています。代わりに、FPGA は密接な依存関係に基づく通信において優れた性能を発揮するため、そのようなパターンを持つアルゴリズムのクラスを取り扱う際には、FPGA の活用を考慮すべきです。

ループは問題ありません

データ・フロー・アーキテクチャーに関するよくある誤解は、反復回数が固定または動的なループは、単純な前方進行パイプラインではないため、データフローのパフォーマンスが低下するのではないかというものです。
少なくともインテルの FPGA ツールチェーンにおいては、これは当てはまりません。少なくともインテルの場合、ループ反復はパイプライン内で高い占有率を維持する良い方法であり、コンパイラーは複数のループ反復をオーバーラップさせて実行できるようにするという概念に基づいて構築されています。ループにより、簡単にパイプラインを常に稼働させ続けることができます!

低いレイテンシーと豊富な接続性

デバイスに搭載された豊富な入出力トランシーバーを活用する一般的な FPGA の利用法は、SYCL を使用する開発者にとっても有効です。例えば、図 17-6 に示すように、一部の FPGA アクセラレーター・カードには、ネットワーク・インターフェイスが搭載されており、データをデバイスに直接ストリーミングして処理し、その結果をネットワークに直接ストリーミングして戻すことができます。このようなシステムは、処理レイテンシーを最小限に抑える場合や、オペレーティング・システムのネットワーク・スタックを介した処理が遅すぎる場合、あるいは CPU の処理リソースを解放するため処理をオフロードする場合に求められます。

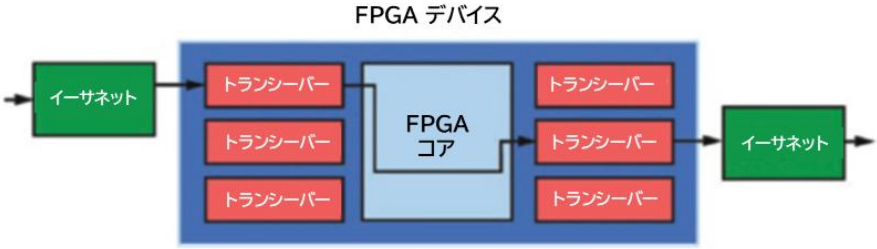

図 17-6. 低レイテンシー I/O ストリーム: FPGA はネットワーク・データと演算を密接に接続

FPGA トランシーバーを介した直接入出力を考慮すると、可能性はほぼ無限大ですが、選択肢はアクセラレーターを構成する回路基板上で実現可能なものによって決まります。

特定のアクセラレーター・カードへの依存度が高く、用途も多岐にわたるため、この後でパイプ言語の構造について説明する以外は、本章ではアプリケーションについては詳しく触れません。代わりに、特定のアクセラレーター・カードに関連付けられたベンダーのドキュメントを参照するか、特定のインターフェイス要件に合うアクセラレーター・カードを探すべきです。

カスタム・メモリー・システム

FPGA 上のメモリーシステム、例えば機能プライベート・メモリーやワークグループ・ローカル・メモリーは、オンチップメモリーの小さなブロックから構成されます。これは重要な点です。なぜなら、各メモリーシステムは、それを使用するアルゴリズムやカーネルの特定の部分に合わせてカスタム構築されるためです。FPGA はオンチップメモリーの帯域幅が非常に広く、カスタム・メモリー・システムの構築と組み合わせることで、通常とは異なるアクセスパターンや構造を持つアプリケーションでも優れたパフォーマンスを発揮できます。図 17-7 は、メモリーシステムが FPGA 上に実装される際にコンパイラーによって行われる最適化の一部を示しています。

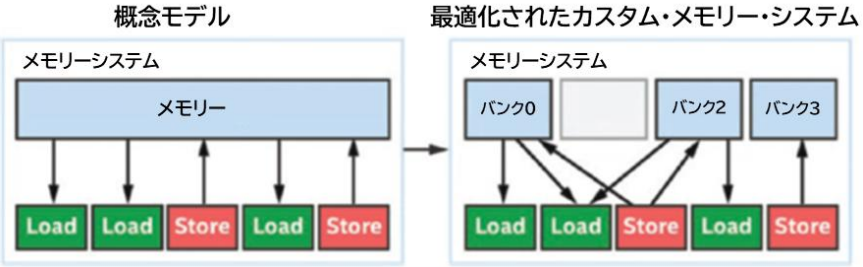

図 17-7. FPGA メモリーシステムは、コンパイラーによって特定のコードに合わせてカスタマイズ

GPU など他のアーキテクチャーは、経験豊富な開発者には理解しやすい固定メモリー構造を持っていますが、多くの場合、最適化が困難な場合もあります。他のアクセラレーターにおける多くの最適化は、バンク競合を回避するためメモリーパターンを変更することに重点を置いています。

バンクごとのアクセスポート数の違いや、通常とは異なるバンク数など、カスタムメモリー構造によってメリットが得られるアルゴリズムがある場合、FPGA はすぐに利点を提供できます。概念的には、固定メモリーシステムを効率良く使用するコードを記述すること（他のほとんどのアクセラレーター）と、コンパイラーによって特定のコードに合わせて効率良く動作するようにメモリーシステムがカスタム設計されること（FPGA）の違いです。

FPGA で実行

FPGA 上でカーネルを実行するには、（他の事前コンパイル・アクセラレーターと同様に）2 つのステップがあります:

- ソースコードを、対象とするハードウェアで実行可能なバイナリーにコンパイルする
- 実行時に、対象とする適切なアクセラレーターを選択する

FPGA ハードウェア上で実行できるカーネルをコンパイルするには、以下のコマンドラインを使用できます:

```
icpx -fsycl -fintelfpga my_source_code.cpp -Xshardware
```

このコマンドは、コンパイラーに対して、`my_source_code.cpp` 内のすべてのカーネルをインテル FPGA アクセラレーター上で実行可能なバイナリーに変換し、生成されたホストバイナリー内にそれらをパッケージ化するように指示します。ホストバイナリーを実行すると（例えば、Linux で `./a.out` を実行すると）、図 17-8 に示すように、ランタイムは、送信されたカーネルを実行する前に、必要に応じて接続されている FPGA を自動的にプログラムします。

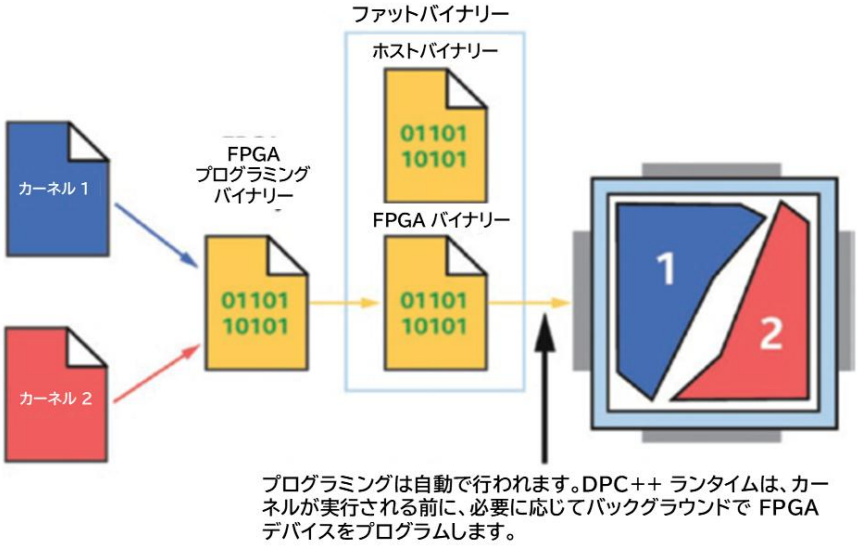

図 17-8. 実行時に FPGA が自動的にプログラムされる

FPGA プログラミング用のバイナリーは、ホスト上で実行するコンパイル済みの DPC++ 実行ファイルに組み込まれています。FPGA はバックグラウンドで自動的に構成されます。

ホストプログラムを実行し、FPGA 上で最初のカーネルを実行するために送信する場合、FPGA へのプログラム書き込みに時間がかかるため、カーネルの実行開始までに若干の遅延が発生する可能性があります。しかし、カーネルは既にデバイスにプログラムされており、実行準備が整っているため、次にカーネルを再送信する際には、同様の遅延は発生しません。

実行時の FPGA デバイスの選択については、第 2 章で説明しました。カーネルを実行する場所をホストプログラムに伝える必要があります。通常、FPGA に加えて CPU や GPU など、複数のアクセラレーター・オプションが利用可能です。プログラム実行中に FPGA を選択する方法の 1 つを簡単にまとめると、図 17-9 のようなコードになります。

```
#include <sycl/ext/intel/fpga_extensions.hpp> // fpga_selector_v 向け
#include <sycl/sycl.hpp>
using namespace sycl;

void say_device(const queue& q) {
   std::cout << "Device : "
             << q.get_device().get_info<info::device::name>()
             << "\n";
}

int main() {
   queue q{ext::intel::fpga_selector_v};
   say_device(q);

   q.submit([&](handler& h) {
     h.parallel_for(1024, [=](auto idx) {
       // ...
     });
   });

   return 0;
}
```

図 17-9. fpga_selector を使用して実行時に FPGA デバイスを選択

コンパイル時

FPGA 向けの設計コンパイルには、ISA ベースのアクセラレーター向けのコンパイルよりも、はるかに長い時間がかかると思われています。それは真実です ...この章の最後では、FPGA の利点と、場合によっては数時間かかることもある計算負荷の高いコンパイル（配置配線最適化）の両方に関係する、FPGA のアーキテクチャー要素の概要を説明します。

ソースコードから FPGA ハードウェアで実行されるまでのコンパイル時間は非常に長いため、コードの開発と繰り返し作業をハードウェアのみで行うことは望ましくありません。FPGA 開発フローは、ハードウェア・コンパイルの回数を最小限に抑える複数のステージを備えており、ハードウェア・コンパイルに時間がかかっても生産性を維持できるように設計されています。図 17-10 は典型的なステージを示しており、時間のほとんどは、迅速なターンアラウンドと反復を可能にする初期ステージに費やされます。

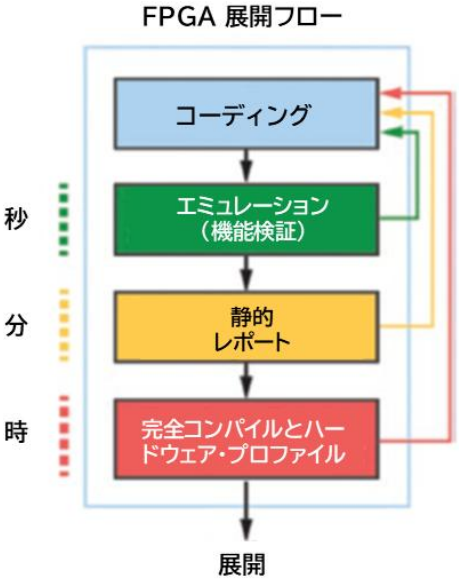

図 17-10. 検証と最適化の大部分は、長い時間を要するハードウェア・コンパイルの前に行われます

エミュレーションとコンパイラーの静的レポートは、DPC++ における FPGA コード開発の基礎となります。このエミュレーターは、関連する拡張機能のサポートや実行モデルのエミュレーションなど、あたかも FPGA であるように動作しますが、ホスト・プロセッサー上で実行されます。したがって、コンパイル時間は CPU デバイスへのコンパイルの場合とほぼ同じですが、実際の FPGA ハードウェア上で実行した時に得られるようなパフォーマンスの向上は期待できません。エミュレーターは、アプリケーションの機能的な正しさを確立し、テストするのに非常に役立ちます。

エミュレーションなどの静的レポートは、ツールチェーンによって迅速に生成されます。レポートは、コンパイラーによって生成された FPGA 構造と、コンパイラーによって特定されたボトルネックについて報告します。これらはどちらも、設計が FPGA ハードウェア上で実行された際に良好なパフォーマンスを発揮するか予測するために使用でき、コードの最適化にも利用されます。レポートに関する情報は、ベンダーのドキュメントを参照してください。ツールチェーンのリリースごとにレポート機能が改善されることがあります（最新の機能についてはドキュメントを参照してください）。ベンダーは、レポートの解釈や最適化に関する詳細なドキュメントを提供しています。これらの情報は別のテーマとなるため、この章だけで詳しく解説することはできません。

FPGA エミュレーター

エミュレーションは主に、アプリケーションの機能デバッグに使用され、アプリケーションが期待どおりに動作し、正しい結果を生成することを確認します。コンパイル時間が長くなる実際の FPGA ハードウェア上で、このレベルの開発を行う理由はありません。エミュレーション・フローは、`icpx` コンパイルコマンドから `-Xshardware` フラグを削除し、同時にホストコード内で `INTEL::fpga_selector_v` の代わりに `INTEL::fpga_emulator_selector_v` を使用することで有効化されます。次のコマンドでコンパイルします:

```
icpx -fsycl -fintelfpga my_source_code.cpp
```

ホスト・プロセッサーで FPGA をエミュレートする `fpga_emulator_selector_v` を使用することで、実際の FPGA ハードウェア向けの時間のかかるコンパイルを行う前に、迅速な開発およびデバッグプロセスを維持できます。`INTEL::fpga_selector_v` の代わりに `INTEL::fpga_emulator_selector_v` を使用する例を図 17-11 に示します。

```
#include <sycl/ext/intel/fpga_extensions.hpp> // fpga_selector_v 向け
#include <sycl/sycl.hpp>
using namespace sycl;

void say_device(const queue& q) {
    std::cout << "Device : "
              << q.get_device().get_info<info::device::name>()
              << "\n";
}

int main() {
   queue q{ext::intel::fpga_emulator_selector_v};
   say_device(q);

   q.submit([&](handler& h) {
       h.parallel_for(1024, [=](auto idx) {
           // ...
       });
   });

   return 0;
}
```

図 17-11. fpga_emulator_selector_v を使用した迅速な開発とデバッグ

FPGA ハードウェアのコンパイルは「事前に」行われる

図 17-10 の*完全コンパイル*と*ハードウェア・プロファイル*ステージは、SYCL の用語でいうところの*事前*コンパイルです。これは、カーネルをデバイスバイナリーにコンパイルするのは、プログラムを最初にコンパイルする時であり、プログラムが実行されるデバイスに送信される時ではないことを意味します。FPGA ではこれは特に重要です:

- コンパイルには、アプリケーションの実行時には発生させたくないほどの時間がかかります。
- DPC++プログラムは、対応するホスト・プロセッサーを持たないシステムでも実行可能できます。FPGA バイナリーへのコンパイル処理は、十分な容量のメモリーを搭載した高速プロセッサーを使用することで効率が向上します。

事前コンパイルにより、プログラムが展開されるシステム上でコンパイルを行うのではなく、コンパイルを実行する場所を容易に選択できます。

FPGA 上で DPC++ を実行すると、背後では実に多くのことが起こります

従来の FPGA 設計（高水準言語を使用しない場合）は非常に複雑になることがあります。カーネルを記述するだけでなく、オフチップメモリーと通信するインターフェイスの構築と構成、コンパイルされた設計が特定の周辺機器と通信するのに十分な速度で動作するようにレジスターを挿入してタイミングを閉じるなど、多くの手順が必要です。
DPC++ はこれらの問題をすべて解決し、従来の FPGA 設計の詳細を知らなくても、動作するアプリケーションを実現できます。このツールは、カーネルをデバイス上で最適化して効率化するコードとして扱い、その後、チップ外の周辺機器との通信、タイミングの調整、ドライバーの設定といったすべての詳細を自動的に処理します。

FPGA で最高のパフォーマンスを実現するには、他のアクセラレーターと同様に、アーキテクチャーに関する詳細な知識が必要ですが、コードから動作する設計への移行手順は、従来の FPGA フローに比べて DPC++ の方がはるかに簡単で効率的です。

FPGA 向けにカーネルを記述

アプリケーションに FPGA を使用することを決定した場合、あるいは単に試してみようと決めた場合でも、良好なパフォーマンスを実現するコードの書き方を理解しておくことが大切です。ここでは、重要な概念を重点的に解説し、混乱を招きやすいトピックをいくつか取り上げることで、より早く学習を始められるようにします。

並列処理の可視化

これまで、FPGA 上で処理を効率良く実行するため、パイプライン並列処理がどのように活用されるか見てきました。図 17-12 に、別の簡単なパイプラインの例を示します。

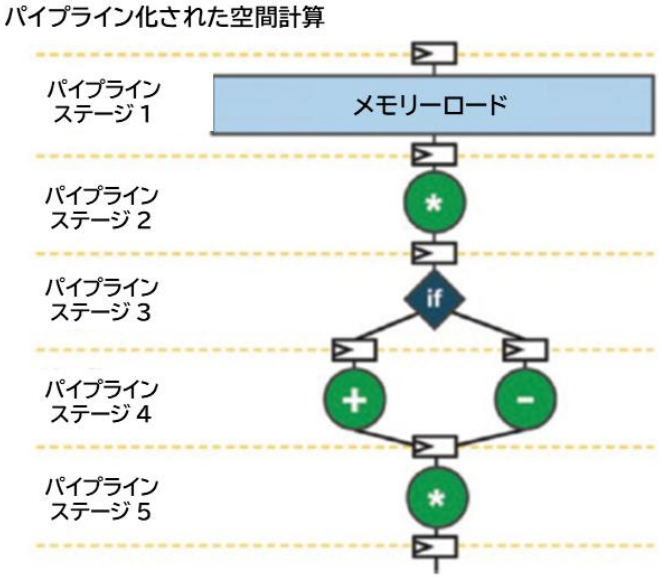

図 17-12. 5 ステージのシンプルなパイプライン: データ要素の処理に 6 クロックサイクル

このパイプラインには 5 つのステージがあります。データはクロックサイクルごとに 1 回、あるステージから次のステージへと移動するため、この単純な例では、データがステージ 1 に入ってからステージ 5 から出るまで 6 クロックサイクル要します。

パイプライン処理の主な目的は、パイプラインの異なるステージで複数のデータ要素を同時に処理できるようにすることです。この点を明確にするため、図 17-13 は、十分なワークがない（この場合はデータ要素が 1 つしかない）パイプラインを示しており、その結果、各パイプライン・ステージはクロックサイクルのほとんどで未使用になります。これは FPGA リソースの非効率的な使用方法です。なぜなら、ハードウェアの大部分がほとんどの時間アイドル状態になっているためです。

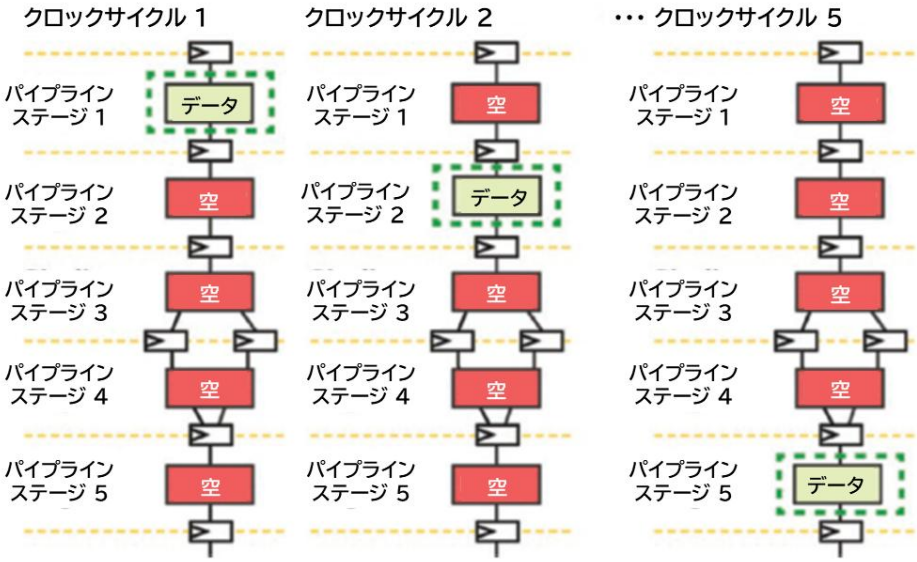

図 17-13. 単一のワーク要素のみを処理する場合、パイプライン・ステージはほとんど使用されません

パイプラインの各ステージをより効率良く活用するには、パイプラインにデータを*供給*する最初のステージの前に、未着手のワークのキューが存在すると想定すると効果的です。図 17-14 に示すように、各クロックサイクルで、パイプラインはキューからワーク要素を 1 つ消費して開始できます。初期の起動サイクルが数回経過すると、パイプラインの各ステージが稼働し、毎クロックサイクルで有用な処理を実行して、FPGA リソースを効率良く利用できるようになります。

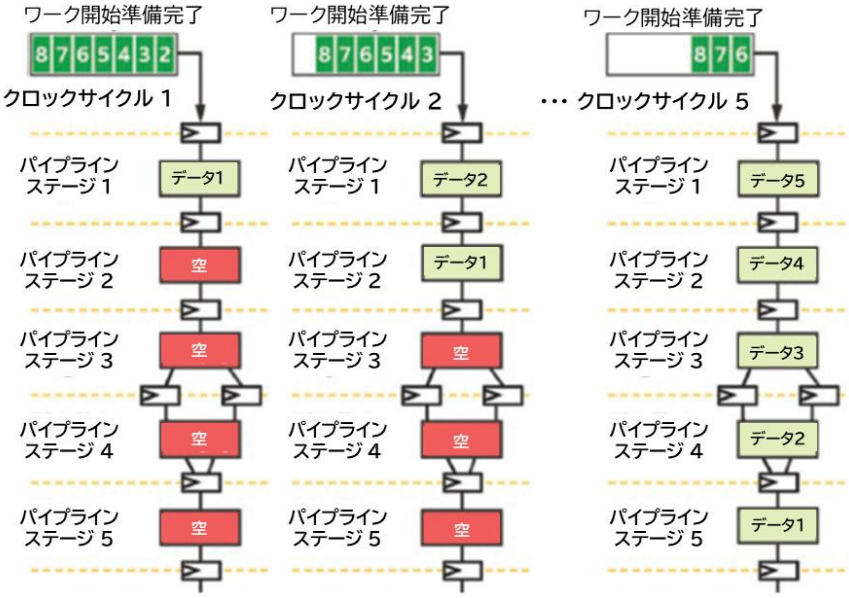

図 17-14. 効率的な利用は、パイプラインの各ステージが常に稼働しているときに実現

次の 2 つのセクションでは、パイプラインに供給するキューを、すぐに開始できるワークで常に満たしておく方法について説明します。以下を見ていきます。

1. ND-range カーネル

2. ループ

これらのどちらを選ぶかによって、FPGA 上で動作するカーネルの基本的なアーキテクチャー設計に影響します。アルゴリズムによっては、どちらかのスタイルに適している場合もあれば、プログラマーの好みや経験によってどちらかを選択する場合もあります。

ND-Range を使用してパイプラインを常に稼働状態に保つ

ND-range の階層型実行モデルについては、第 4 章で説明しました。図 17-15 は、主要な概念を示しています。それは、ワーク項目が階層的にグループ化され、ワーク項目がカーネルによって定義されるワークの基本単位である ND-range 実行モデルです。このモデルは、実行モデル階層のさまざまなレベルでワーク項目が同時に実行される GPU の効率良いプログラミングのために開発されたものです。GPU ハードウェアが効率良く処理できるワークの種類に合わせて、ほとんどのアプリケーションでは ND-range のワーク項目が相互に通信することはありません。

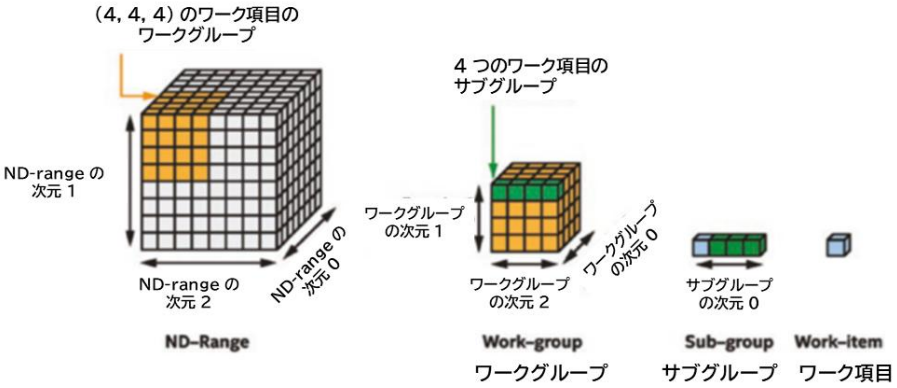

図 17-15. ND-range 実行モデル: ワーク項目の階層的グループ化

FPGA の空間パイプラインは、ND-range を使用することで非常に効率的に処理を実行できます。このプログラミング・スタイルは FPGA で完全にサポートされており、図 17-16 に示すように、クロックサイクルごとに異なるワーク項目がパイプラインの最初のステージに入ると考えられます。

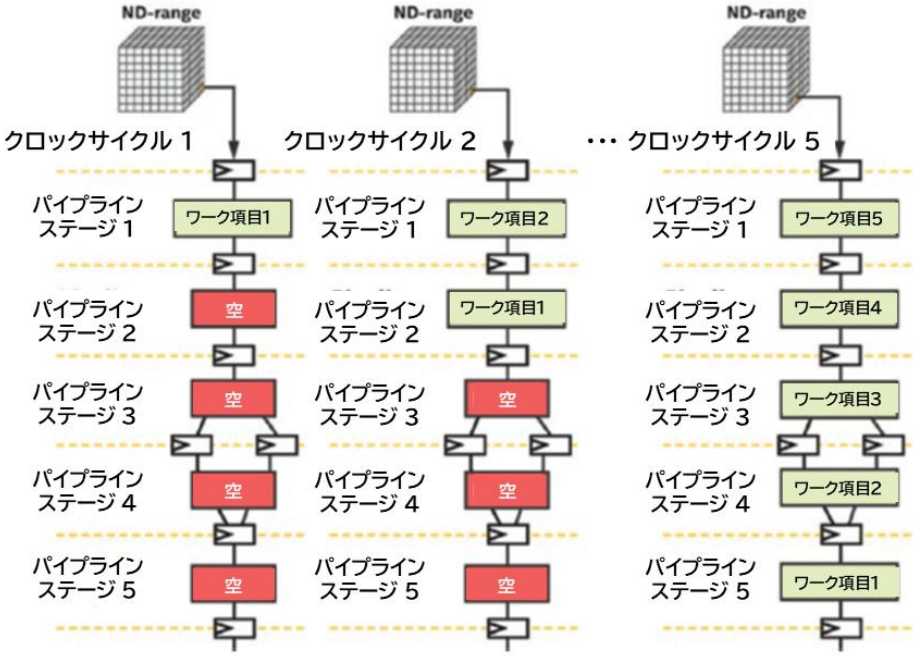

図 17-16. ND-range を空間パイプラインに供給

パイプラインを常に稼働状態に保つために、ワーク項目を使用して FPGA 上に ND-range カーネルを作成するべきタイミングは?
答えは簡単です。アルゴリズムやアプリケーションを、頻繁に（理想的には全く）通信する必要のない独立したワーク項目として構成できる場合は、ND-range を使用すべきです。ワーク項目間で頻繁に通信が必要な場合、または ND-range という概念を自然に意識しない場合は、ループ（次のセクションで説明）を使用することで、アルゴリズムを効率良く表現できます。

アルゴリズムを、ワーク項目間の通信をほとんど（あるいは全く）必要としないように構築できれば、ND-range は空間パイプラインを常に一杯に保つワークを生成する優れた方法となります。

パイプラインに ND-range を供給する効率良いカーネルの例として、乱数生成器が挙げられます。この乱数生成器では、シーケンス内の数値の生成が、以前に生成された数値と独立したアルゴリズムが採用されています。

図 17-17 は、16×16×16 の範囲内の各ワーク項目に対して乱数生成関数を 1 回呼び出す ND-range カーネルを示しています。乱数生成関数が、ワーク項目 ID を入力として受け取る点に注目してください。

```
h.parallel_for({16, 16, 16}, [=](auto I) {
    output[I] = generate_random_number_from_ID(I);
});
```

図 17-17. 複数のワーク項目（16 × 16 × 16）による乱数発生器の呼び出し

この例は、グローバルサイズのみを指定して範囲を使用する parallel_for 呼び出しを示しています。代わりに、グローバル・ワーク・サイズとローカル・ワークグループ・サイズの両方を指定する nd_range を受け取る parallel_for スタイルを使用することもできます。FPGA はオンチップリソースからワークグループ・ローカル・メモリーを効率良く実装できるため、ワークグループ・ローカル・メモリーが必要な場合や、ワークグループ ID を利用することでコードが簡素化される場合など、ワークグループが有効な場合は自由にワークグループを使用できます。

並列乱数発生器

図 17-17 の例では、generate_random_number_from_iD(i) は並列に呼び出されても安全、かつ正しく動作する乱数生成器であると想定しています。例えば、parallel_for の範囲内の異なるワーク項目が関数を実行する場合、各ワーク項目によって異なるシーケンスが生成され、各シーケンスは生成器から期待される分布に従うことが期待されます。並列乱数生成器自体が複雑なテーマであるため、ライブラリーを利用したり、ブロック・スキップアヘッド・アルゴリズムなどの手法を通してこのテーマについて学ぶのが良いでしょう。

パイプラインはデータ依存関係を気にしません

ベクトル・アーキテクチャー（GPU など）をプログラミングする課題の一つは、一部のワーク項目がベクトル命令のレーンとして同時に実行される場合、ワーク項目間の通信を最小限に抑えつつ、効率的なアルゴリズムを構築することです。アルゴリズムやアプリケーションによっては、ベクトル・ハードウェアに適したものもあれば、そうでないものもあります。マッピングがうまくいかない一般的な原因は、隣接する他の計算とのデータ依存関係により、アルゴリズムが広範囲でデータを共有する必要があることです。第 14 章で説明したように、サブグループは、同じサブグループ内のワーク項目間で効率良い通信を提供することで、ベクトル・アーキテクチャーの課題の一部に対処します。

FPGA は、独立した処理に分解できないアルゴリズムにおいて重要な役割を果たします。FPGA の空間パイプラインは、ワーク項目ごとにベクトル化されるのではなく、パイプラインの各ステージで連続したワーク項目を実行します。この並列処理の実装により、空間パイプライン内で、ワーク項目間（異なるワークグループに属するものも含む）の細粒度の通信を容易、かつ効率良く実装できます。

一例として、出力 `N+1` が出力 `N` が何であったかを知っていることに依存する乱数発生器があります。これにより、2 つの出力間にデータ依存性が生じ、各出力が ND-range 内のワーク項目で生成される場合、ワーク項目間にデータ依存性が生じ、一部のアーキテクチャーでは複雑でコストのかかる同期が必要になることがあります。

このようなアルゴリズムをシリアルに記述する場合、通常はループを作成し、反復 N+1 で反復 N の計算結果を使用します。これは図 17-18 に示されています。各反復処理は、前の反復処理で計算された状態に依存します。これは非常によくあるパターンです。

```
int state = 0;
for (int i = 0; i < size; i++) {
   state = generate_random_number(state);
   output[i] = state;
}
```

図 17-18. ループ伝搬データの依存関係（状態）

空間実装では、結果をパイプラインの後方へ、つまり後のサイクルで開始された処理（すなわち、パイプラインの前のステージにある処理）に対して効率良く伝達することができ、空間コンパイラーはこのパターンを基に多くの最適化を実装します。図 17-19 は、ステージ 5 からステージ 4 へのデータの逆方向通信の概念を示しています。空間パイプラインは、ワーク項目全体にわたってベクトル化されません。これにより、パイプライン内で結果を逆方向に渡すことで、データ依存関係を効率良く伝達できます。

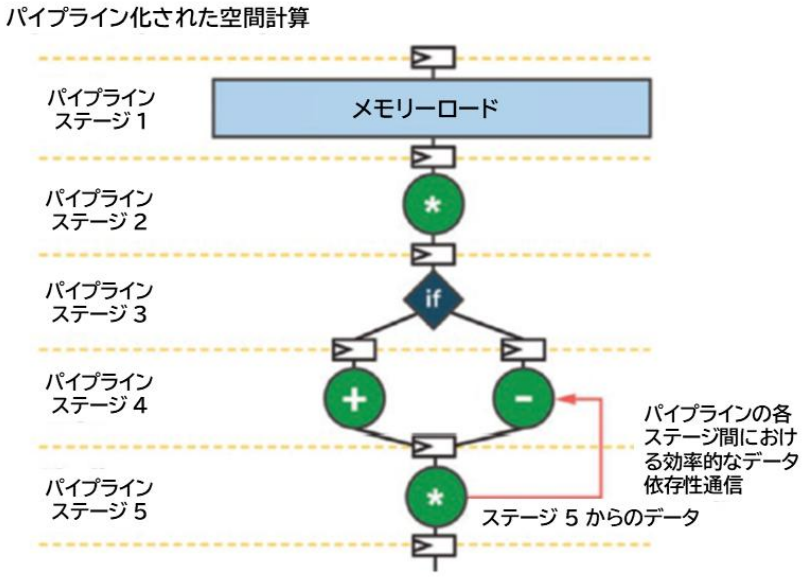

図 17-19. 後方通信により、効率良いデータ依存性の通信が可能

データを（パイプラインの前のステージに）逆方向に渡す機能は、空間アーキテクチャーにとって重要ですが、それを活用するコードの記述方法は明確ではありません。このパターンを簡単に表現するには、2 つの方法があります:

1. ループ

2. ND-range カーネルを使用したカーネル内のパイプ

2 つ目の選択肢は、この章の後半で説明するパイプに基づいています。しかし、ループほど一般的ではないため、ここでは完全を期すために触れるだけで、詳細は省略します。ベンダーのドキュメントにはパイプ方式の詳細な情報が記載されていますが、特に理由がない限り、次に説明するループ方式に従う方が容易です。

空間パイプラインによるループの実装

データ依存性のあるアルゴリズムをプログラミングする場合、ループは自然な選択です。ループは、ループを終了するタイミングを決定するカウンターが、反復をまたいで引き継がれる最も基本的なループの例（図 17-20 の変数 i ）でも反復間の依存関係を持つことがあります。

```
int a = 0;
for (int i = 0; i < size; i++) {
    a = a + i;
}
```

図 17-20. 2 つのループ依存関係（つまり、i と a）を持つループ

図 17-20 の単純なループでは、a= a + i の右辺にある a の値は、前のループ反復で格納された値、またはループの最初の反復の場合は初期値を反映していることが分かります。空間コンパイラーがループを実装する場合、図 17-21 に示すように、ループの反復処理でパイプラインの各ステージを埋めることができます。ワークの開始準備が整ったワークキューには、ワーク項目ではなくループの反復処理が含まれることに注目してください。

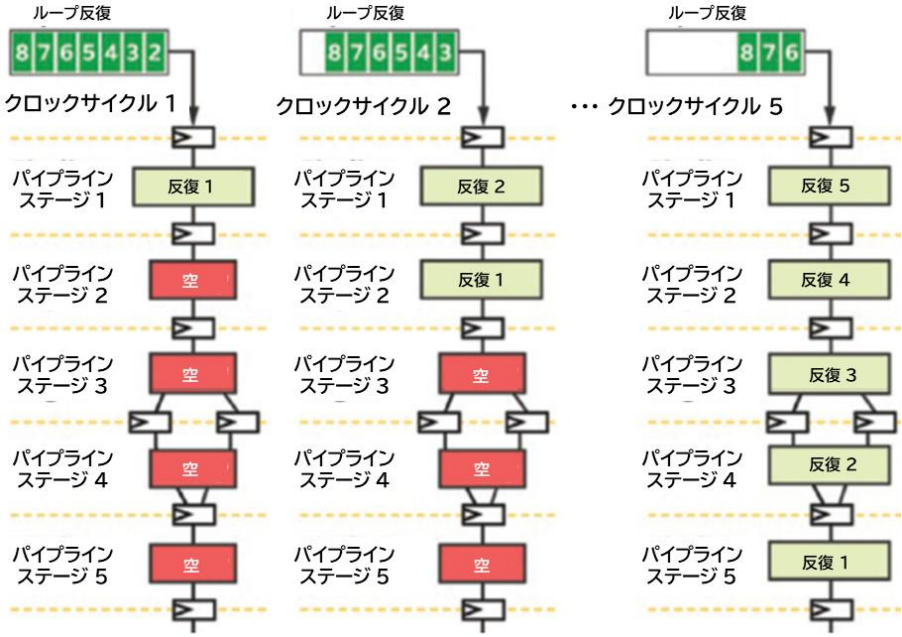

図 17-21. 連続したループ反復によって供給されるパイプライン・ステージ

改良された乱数発生器の例を図 17-22 に示します。この場合、図 17-17 のようにワーク項目の ID に基づいて数値を生成する代わりに、ジェネレーターは以前に計算された値を引数として受け取ります。

```
h.single_task([=]() {
   int state = seed;
   for (int i = 0; i < size; i++) {
     state = generate_incremental_random_number(state);
     output[i] = state;
   }
});
```

図 17-22. 前回生成された値に依存する乱数発生器

この例では、繰り返し実行される処理が単一タスク内のループで表現されているため、parallel_for の代わりに single_task を使用しています。そのため、このコードに複数のワーク項目を含める（parallel_for を使用する）必要はありません。single_task 内のループにより、以前に計算された state 値が乱数生成関数の各呼び出しに渡すことを、より簡単に表現できます（プログラミング上の利便性）。

図 17-22 のような場合、FPGA はループを効率良く実装できます。多くの場合、パイプラインを常に一杯の状態に保つことができ、少なくともレポートを通じて、稼働率を高めるために何を変更すべきかヒントを提供します。この点を考慮すると、ループ反復をワーク項目に置き換えた場合、同じアルゴリズムの説明がはるかに困難になります。なぜなら、あるワーク項目によって生成された値を、インクリメント計算で使用するため別のワーク項目に伝達する必要があるためです。特に、各ワーク項目がそれぞれ独立した乱数列を計算するようにワークをバッチ処理できない場合、コードはさらに複雑になります。

ループ開始間隔

概念的には、C++ のループ反復処理は、図 17-23 に示すように、1 回ずつ実行されると考えるのが妥当です。それがプログラミング・モデルであり、ループについて考える正しい方法です。しかし、実装においては、プログラムの動作の大部分（つまり、定義済みで競合のない動作）が目に見える形で変化しない限り、コンパイラーは多くの最適化を自由に実行できます。コンパイラーの最適化に関係なく、重要なことは、ループが図 17-23 のように実行されるように見えることです。

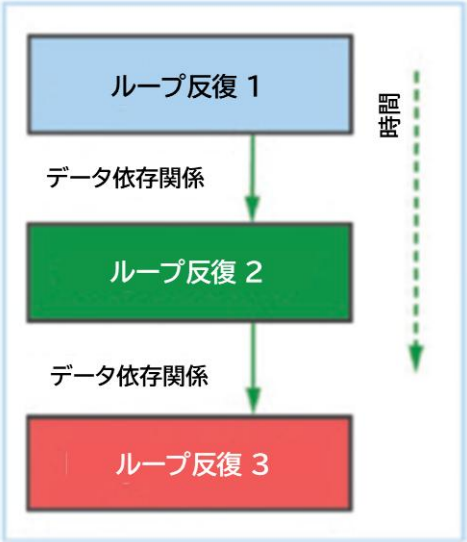

図 17-23. 概念的にループの反復処理は一つずつ順番に実行される

空間コンパイラーの観点からは、図 17-24 はループ反復の実行が時間的にオーバーラップされるループ・パイプライン最適化を示しています。異なる反復処理では、空間パイプラインの異なるステージがそれぞれ実行されます。パイプラインの各ステージ間のデータ依存関係はコンパイラーによって管理されるため、プログラムは反復処理が順次実行されているかのように動作します（ただし、ループの実行はより早く終了します）。

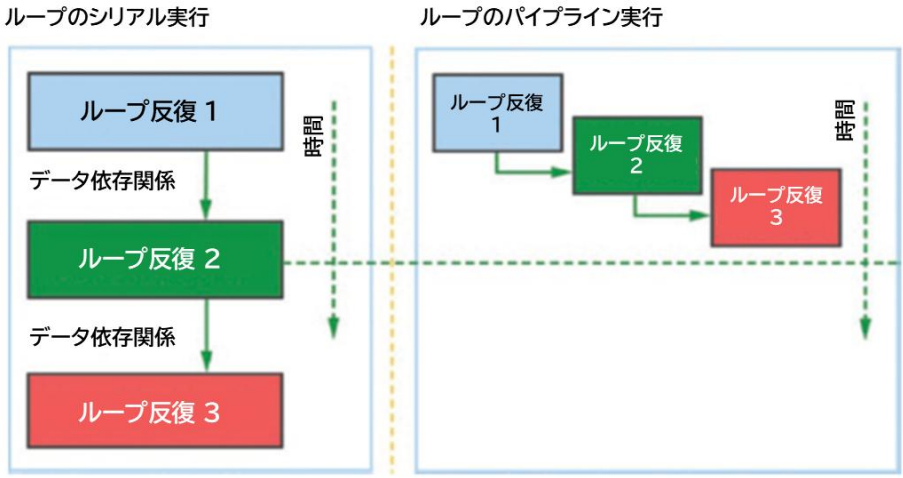

図 17-24. ループ・パイプライン処理では、ループ反復処理をパイプラインの各ステージにまたがってオーバーラップできます

ループ・パイプライン処理は、ループ反復処理内の結果が、ループ反復のすべての処理が完了するよりもかなり前に計算を完了する可能性があること、そして空間パイプラインでは、コンパイラーがそう判断した場合に、結果を前のパイプライン・ステージに渡すことが可能であることを理解すれば、容易に理解できます。図 17-25 は、ステージ 1 の結果がパイプライン内で逆方向にフィードバックされ、前の反復が完了する前に、以降のループ反復でその結果を早期に使用できることを示しています。

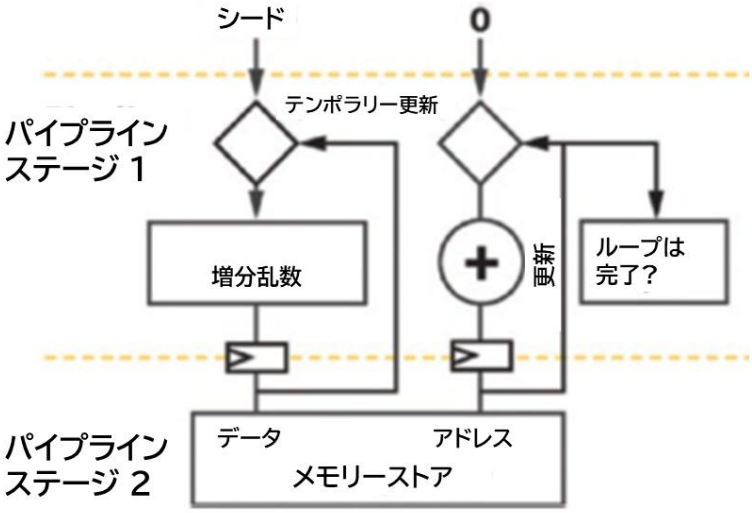

図 17-25. 増分乱数発生器のパイプライン実装

ループ・パイプライン処理では、ループで複数の反復処理の実行がオーバーラップする可能性があります。オーバーラップする部分があることで、ループ内でデータ依存関係があっても、ループの反復処理を利用してパイプラインにワークを埋め込むことができ、効率化につながります。図 17-26 は、図 17-25 に示した単純なパイプライン内で、ループの反復処理がループによって伝搬されるデータ依存関係があっても、どのように実行がオーバーラップするか示しています。

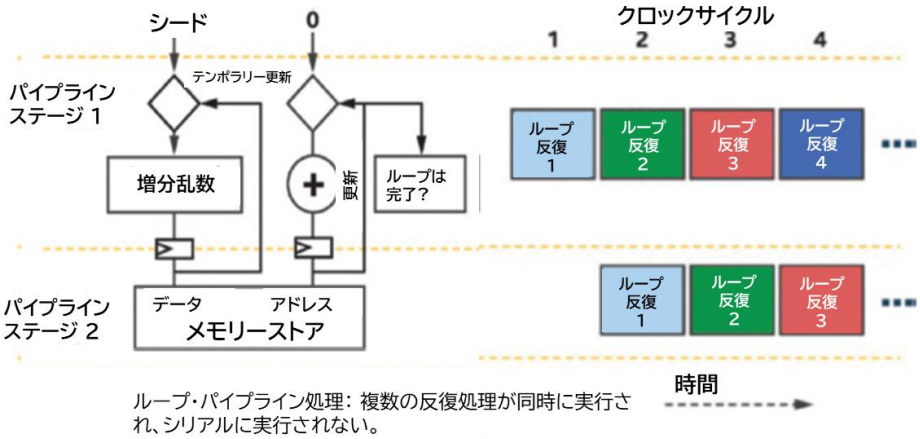

図 17-26. ループ・パイプライン処理は、複数のループ反復処理の一部を同時に処理

実際のアルゴリズムでは、データ依存関係の計算に複数のクロックサイクルを要する場合があるため、クロックサイクルごとに新しいループ反復を開始することはほとんど不可能です。これは、メモリーへのアクセス（特にオフチップメモリーからのアクセス）が、依存関係の計算におけるクリティカルパス上にある場合に発生します。

その結果、新しいループの反復を開始できるのは N クロックサイクルごとに 1 回のみのパイプラインが形成され、これを N サイクルの*開始間隔*（II）と呼びます。図 17-27 に例を示します。ループ開始間隔（II）が 2 の場合、2 サイクルごとに新しいループ反復が開始されるため、パイプライン・ステージの占有率が低下します。

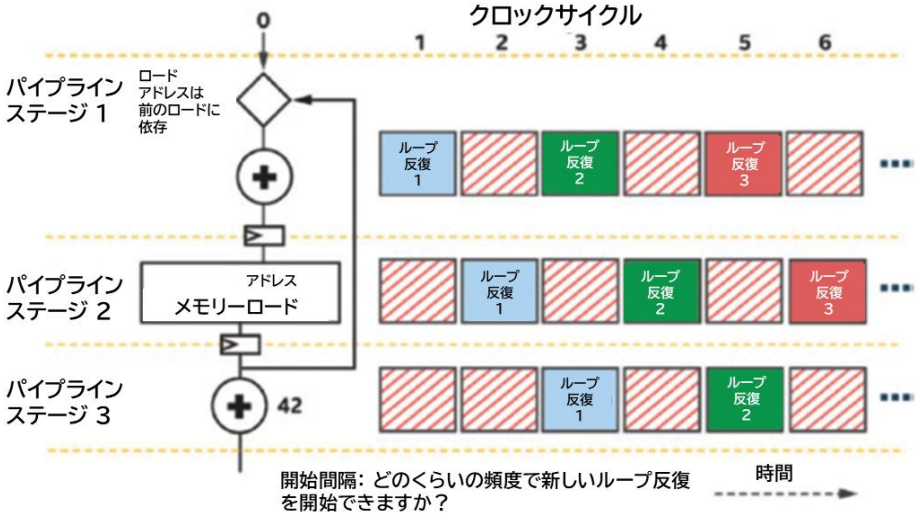

図 17-27. イプラインの各ステージの占有率が最適ではない

II が 1 より大きい場合、各ステージの平均占有率が低下するため、パイプラインの効率が低下する可能性があります。図 17-27 からそれが明らかであり、II=2 の場合、パイプライン・ステージの大部分（50%）は未使用です。この状況を改善する方法は複数あります。

コンパイラーは可能な限り II を削減するため広範な最適化を行い、そのレポートには各ループの開始間隔が示され、開始間隔が 1 より大きい場合はその理由も示します。レポートを基にループ内の計算を再構築すると、多くの場合、II を削減できます。特に、開発者は、コンパイラーには許可されていないループ構造を変更できるためです。コンパイラーのレポートを参照して、特定のケースで II を削減する方法を学びましょう。

II が 1 より大きい場合の非効率性を軽減する別の方法として、ネストされたループを使用する方法があります。ネストされたループでは、外側のループの反復処理と、II > 1 である内側のループの反復処理を交互に行うことで、パイプラインのすべてのステージを埋めることができます。この手法の詳細については、ベンダーのドキュメントおよびコンパイラーのレポートを参照してください。

パイプ

空間アーキテクチャーや他のアーキテクチャーにおける重要な概念の一つに、先入れ先出し（FIFO）バッファーがあります。FIFO が重要な理由は多数ありますが、プログラミングにおいて特に役立つ特性が 2 つあります:

1. **データには、暗黙の制御情報が付随しています。**これらのシグナルは、FIFO の空/フル状態を示しており、問題を独立した部分に分解する際に役立ちます。

2. FIFO には**記憶容量**があります。これにより、メモリーアクセス時のレイテンシーが大きく変動するなど、動的に挙動が変化する場合でも、パフォーマンスを容易に達成できるようになります。

図 17-28 は、FIFO の簡単な動作例を示しています。

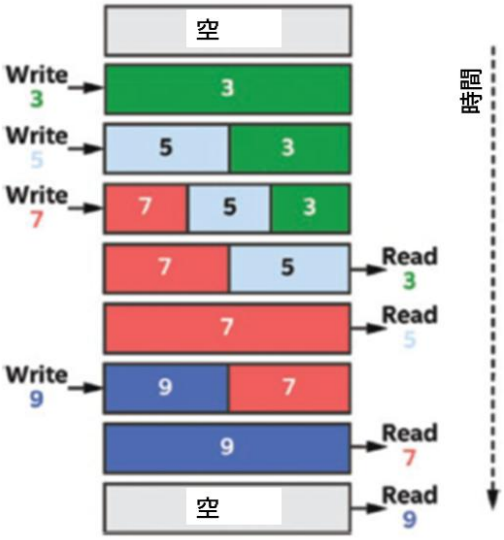

図 17-28. FIFO の時系列の動作例

DPC++ では、*パイプ*と呼ばれる機能を通じて FIFO が利用可能です。FPGA プログラムの作成にパイプを重視すべき主な理由は、パイプによって問題をより小さな部分に分解し、モジュール化された方法で開発と最適化に集中できるためです。また、FPGA の豊富な通信機能を活用することもできます。図 17-29 では、これらの両方をグラフで示しています。

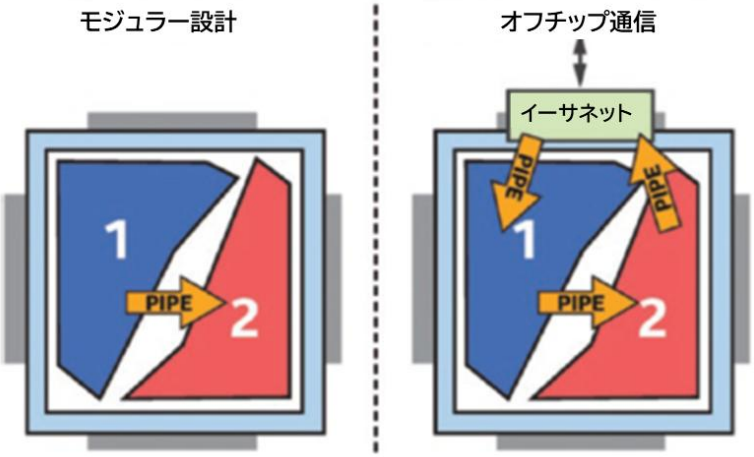

図 17-29. パイプ構造により、モジュール設計とハードウェア周辺機器へのアクセスが容易になります

FPGA カーネルはデバイス上に同時に存在でき（チップの異なる領域に）、効率的な設計では、カーネルのすべての部分が常に、毎クロックサイクルでアクティブであることを覚えておいてください。 つまり、FPGA アプリケーションの最適化には、カーネルやカーネルの一部が互いにどのように相互作用するか考慮する必要があり、パイプはこれを容易にする抽象化を提供します。

パイプは FPGA 上のオンチップメモリーを使用して実装された FIFO であり、データをオフチップメモリーに移動するコストをかけずに、実行中のカーネル間およびカーネル内での通信を可能にします。これにより、安価な通信が可能になり、パイプに関連付けられた制御情報（空/フル信号）によって軽量な同期メカニズムが実現されます。

パイプは必要か？いいえ。パイプを使用することなく効率的なカーネルを記述することは可能です。パイプを使わずに従来型のプログラミング・スタイルで、FPGA のリソースをすべて活用し、最大限のパフォーマンスを実現することもできます。しかし、ほとんどの開発者にとって、モジュール化された空間設計をプログラミングして最適化する方が容易であり、パイプはそれを実現する優れたツールです。

図 17-30 に示すように、一般に利用可能なパイプには 4 つの種類があります。このセクションの残りの部分では、最初のタイプ（カーネル間のパイプ）について説明します。これだけでパイプとは何か、そしてどのように使用されるか示すのに十分です。パイプは、単一のカーネル内だけでなく、ホストや入出力周辺機器とも通信できます。パイプの様々な形状や用途に関する詳細については、ベンダーのドキュメントを確認してください。ここでは、それを詳しく説明するページ数がありません。

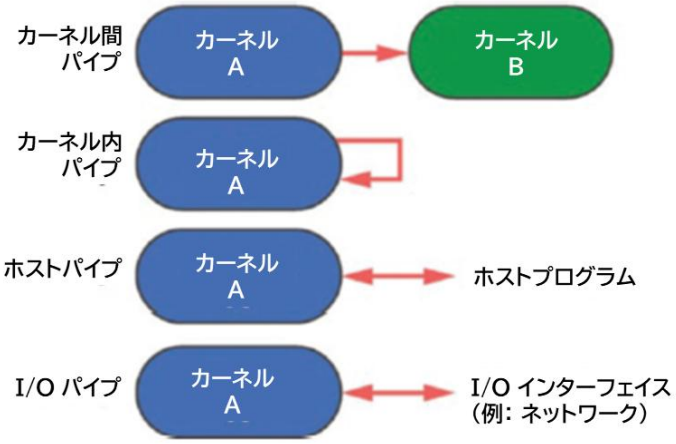

図 17-30. DPC++ におけるパイプ接続のタイプ

簡単な例を図 17-31 に示します。この場合、パイプを介して通信する 2 つのカーネルがあり、それぞれの読み書き操作は整数ユニットに対して行われます。

```
// パイプタイプのエイリアスを作成して、使用方法に一貫性を持たせる
using my_pipe = ext::intel::pipe<class some_pipe, int>;

// ND-range カーネル
q.submit([&](handler& h) {
  auto a = accessor(b_in, h);

  h.parallel_for(
      count, [=](auto idx) { my_pipe::write(a[idx]); });
});

// Single_task カーネル
q.submit([&](handler& h) {
  auto a = accessor(b_out, h);

  h.single_task([=]() {
     for (int i = 0; i < count; i++) {
        a[i] = my_pipe::read();
     }
  });
});
```

図 17-31. 2 つのカーネル間のパイプ処理: (1) ND-range と (2) ループを含む単一タスク

図 17-31 には注目すべき点がいくつかあります。まず、2 つのカーネルがパイプを使って互いに通信します。カーネル間にアクセサーまたはイベントの依存関係がない場合、DPC++ ランタイムは両方を同時に実行し、SYCL メモリーバッファー全体や USM ではなくパイプを介して通信できるようにします。

パイプはタイプベースのアプローチを使用して識別され、各パイプは図 17-32 に示すパイプタイプのパラメーター化を使用して識別されます。パイプタイプのパラメーター設定により、特定のパイプが識別されます。同じパイプタイプに対する読み書きは、同じ FIFO に対して行われます。パイプのタイプの識別情報を定義するテンプレート・パラメーターは 3 つあります。

```
template <typename name, typename dataT,
          size_t min_capacity = 0>
class pipe;
```

図 17-32. パイプタイプのパラメーター化

プログラミング・エラーを減らし、コードの可読性を向上させるため、図 17-31 の最初のコード行に示すように、パイプタイプを定義する際に型エイリアスを使用することを推奨します。

パイプを識別するには型エイリアスを使用してください。これによりコードが簡素化され、意図しないパイプが誤って作成されるのを防ぐことができます。

パイプには `min_capacity` パラメーターがあります。デフォルト値は 0 で、これは*自動選択*ですが、指定すると読み出されるワードが一切ない場合でも、少なくともその数のワードがパイプに書き込まれることが保証されます。パラメーターは次のような場合に役立ちます。

- パイプを介して通信する 2 つのカーネルは同時に実行されません。そのため、2 番目のカーネルが実行を開始してパイプから読み取りを始める前に、最初のカーネルはすべての出力を書き込むのに十分な容量のパイプが必要です。

- カーネルがデータをバースト生成または消費する場合、パイプの容量を増やすことでカーネル間の分離が可能になり、パフォーマンスの影響を互いに切り離すことができます。例えば、データを生成するカーネルは、そのデータを消費するカーネルがビジー状態でも、（パイプの容量が一杯になるまで）書き込みを続行できます。これは、FPGA 上のメモリーリソースを若干消費するだけで、カーネル間の実行順序に柔軟性を持たせることができます。

ブロック化と非ブロック化パイプアクセス

大部分の FIFO インターフェイスと同様に、パイプにも*ブロック化と非ブロック化*の 2 つのインターフェイスがあります。ブロック化アクセスは、操作が成功するまで待機（実行をブロック/一時停止）しますが、非ブロック化アクセスはすぐに戻り、操作が成功したかを示すブール値を設定します。

成功の定義は単純です: パイプからデータを読み取っている場合、読み取れるデータが存在していれば（パイプが空でなければ）、読み取りは成功します。書き込み処理中にパイプが一杯でなければ、書き込みは成功します。図 17-33 は、`pipe` クラスのアクセスメンバー関数の 2 つの形式を示しています。これらのパイプのメンバー関数でパイプへの書き込みや読み取りが可能であることがわかります。パイプへのアクセスには、ブロック化アクセスと非ブロック化アクセスがあることを忘れないでください。

```
// ブロック化
T read();
void write(const T &data);

// 非ブロック化
T read(bool &success_code);
void write(const T &data, bool &success_code);
```

図 17-33. パイプへの書き込みや読み取りを可能にするパイプのメンバー関数

ブロック化アクセスと非ブロック化アクセスには、アプリケーションが実行したいことよってそれぞれ異なる用途があります。カーネルがパイプからデータを読み取るまで処理を続行できない場合、ブロック化読み取りを使用するのが妥当です。カーネルが複数のパイプの 1 つからデータを読み取りたい場合、どのパイプにデータが存在するか不明である場合は、非ブロック化呼び出しを使用してパイプからデータを読み込む方が理にかなっています。その場合、カーネルはパイプからデータを読み取って処理できますが、パイプが空である時は代わりに次のパイプに進み、データが利用可能であるパイプから読み取ろうとします。

パイプに関する詳細情報

この章ではパイプについてごく簡単触れましたが、パイプとは何か、そして基本的な使い方は理解できたはずです。FPGA ベンダーのドキュメントには、多くの情報と、さまざまなアプリケーションでの使用例が掲載されているため、パイプが特定のニーズに適していると思われる場合は、そちらを参照すると良いでしょう。

カスタム・メモリー・システム

ほとんどのアクセラレーター向けにプログラミングを行う場合、最適化作業の大部分はメモリーへのアクセス効率を高めることに費やされる傾向があります。FPGA 設計も同様であり、特に入出力データがチップ外のメモリーを経由する場合は顕著です。

FPGA のメモリーアクセスを最適化する理由は 2 つあります:

- 帯域幅を削減するため、特にその帯域幅の一部が非効率的に使用されている場合
- 空間パイプラインで不要なストールを引き起こしているメモリーのアクセスパターンを変更するため

空間パイプラインにおける*ストール*について簡単に触れておきます。コンパイラーは、特定の種類のメモリーからの読み書きにかかる時間を仮定し、それに応じてパイプラインを最適化およびバランス調整することで、メモリーのレイテンシーを隠匿します。しかし、メモリーへのアクセスが非効率である場合、レイテンシーが長くなり、結果としてパイプラインがストールします。パイプラインの前のステージは、何らかの処理（例えば、メモリーアクセス）を待機するパイプライン・ステージブロックされるため、実行を続行できなくなります。 図 17-34 は、ロードを処理するパイプラインが停止し、前進できない状況を示しています。

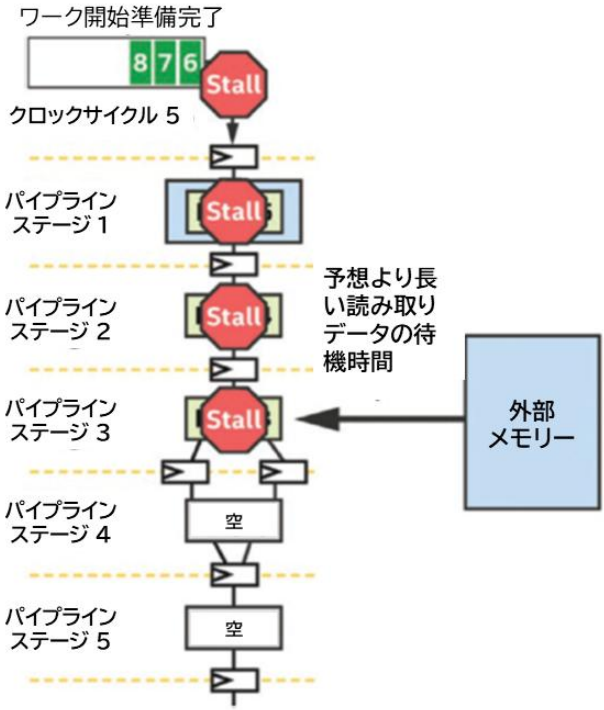

図 17-34. メモリーのストールがパイプラインの前のステージのストールも引き起こす可能性があります

メモリーシステムの最適化は、いくつかの側面から実施できます。これまでのように、コンパイラーのレポートは、コンパイラーが実装した内容と、調整や改善する価値のあることを把握するガイドとなります。ここでは、利用可能な自由度のいくつかを示すために、最適化に関するトピックを挙げておきます。最適化は通常、明示的な制御と、コードを修正してコンパイラーに意図した構造を推論させる両方によって実現できます。コンパイラーの静的レポートとベンダーのドキュメントは、メモリーシステムの最適化における重要な要素であり、検証やチューニングの最終段階では、ハードウェア実行中にプロファイル・ツールと組み合わせて実際のメモリー動作を捕捉することもあります。メモリー最適化に関する考慮事項は以下のとおりです:

- **静的結合:** コンパイラーは、可能であれば、メモリーへのアクセスを少数の幅の広いアクセスに合成します。これにより、パイプライン内のロードユニットやストアユニット数、メモリーシステムのポート数、アービトレーション・ネットワークの規模と複雑性、その他のメモリーシステムの詳細といった点で、メモリーシステムの複雑性が軽減されます。一般に、可能な限り静的結合は最大化されますが、これはコンパイラーのレポートで確認できます。カーネル内のアドレス指定ロジックを簡素化するだけで、コンパイラーの積極的な静的結合が可能になる場合があるため、コンパイラーが期待どおり推論を行っているかレポートで確認してください。

- **メモリーアクセス形式:** コンパイラーはメモリーアクセス用のロードユニットまたはストアユニットを生成します。これらのユニットは、アクセスされるメモリー技術（オンチップ、DDR、HBM など）と、ソースコードから推測されるアクセスパターン（ストリーミング、動的に統合/拡張される、特定のサイズのキャッシュから恩恵を受ける可能性など）に合わせて調整されます。コンパイラー・レポートは、コンパイラーが推論した内容を示しており、必要に応じてコードを修正したり、制御を追加することで、パフォーマンスを向上できます。

- **メモリーシステム構造:** メモリーシステム（オンチップとオフチップの両方）は、バンク構造を持つことができ、コンパイラーによって多数の最適化が実装されます。これらの構造を制御し、空間実装の特定機能の調整に使用できる制御機能やモード変更機能が数多く存在します。

最後に

FPGA の入門開発者と話をする際、デバイスを構成するコンポーネントを大まかに理解すること、そして混乱しやすいクロック周波数について説明することが役立つと感じています。この章の締めくくりとして、これらのトピックを取り上げます。

FPGA ビルディング・ブロック

ツールの処理の流れ（特にコンパイル時）を理解するため、FPGA を構成するビルディング・ブロックについて触れておく必要があります。これらのビルディング・ブロックは DPC++ と SYCL によって抽象化されており、一般的なアプリケーション開発（少なくともコードを機能させるという意味では）では、それらに関する知識は不要です。とはいえ、それらの存在は、空間的アーキテクチャーの最適化やツールの処理の流れに対する直感の形成に役立ち、また、アプリケーションに適したデータ型の選択といった高度な最適化においても、考慮される要素となります。

現代の FPGA デバイスを非常に簡略化して説明すると、以下の 5 つの基本要素から構成されます:

1. **ルックアップ・テーブル**: 数本のバイナリー入力ワイヤーを持ち、バイナリー出力を生成する基本的なブロック。入力に対する出力は、ルックアップ・テーブルにプログラムされたエントリーによって定義されます。これらは極めてプリミティブなブロックですが、計算処理に使用される最新の FPGA には、このようなブロックが多数（数百万個）搭載されています。これらは、私たちの設計の多くが実装される際の基礎となるものです！

2. **数学エンジン**: 単精度浮動小数点数の加算や乗算といった数学演算については、FPGA にはこれらの演算を効率的に行う専用ハードウェアが搭載されています。

現代の FPGA には、こうしたブロックが数千個も搭載されており、1 クロックサイクルごとに、少なくともそれだけの数の浮動小数点基本演算を並列に実行できます。ほとんどの FPGA は、これらの演算エンジンを *デジタル信号プロセッサー* (DSP) と呼んでいます。

3. **オンチップメモリー**: これは FPGA と他のアクセラレーターとの決定的な違いであり、2 種類のメモリーがあります (実際にはもっと種類がありますが、ここでは触れません)。(1) 演算間のパイプライン処理やその他の目的に使用されるレジスター、および (2) デバイス全体に分散された小さなランダム・アクセス・メモリーを提供するブロックメモリー。
   最新の FPGA には、数百万ビットのレジスターと、1 万個以上の 20K ビット RAM メモリーブロックを備えるものもあります。
   これらの要素はクロックサイクルごとにそれぞれアクティブにできるため、効率良く使用すれば、結果として大幅なオンチップメモリー容量と帯域幅を実現できます。

4. **オフチップ・ハードウェアへのインターフェイス**: FPGA が進化した理由の一つは、その非常に柔軟なトランシーバーと入出力の接続性にあります。これにより、チップ外メモリーからネットワーク・インターフェイス、さらにはそれ以外まで、ほぼあらゆるものとの通信が可能です。

5. **他のすべての要素間のファブリックのルーティング**: 一般的な FPGA には、前述のような要素が多数存在し、それらの接続は固定されていません。複雑なプログラマブル・ルーティング・ファブリックにより、FPGA を構成する微細な要素間で信号が伝達されます。

FPGA 上の各タイプのブロック数（数百万個のものもある）と、ルックアップ・テーブルなどのブロックの粒度の細かさを考慮すると、FPGA 構成ビットストリームを生成する際に要するコンパイル時間は理解できます。各細粒度のリソースに機能を割り当てるだけでなく、それらのリソース間のルーティングも構成する必要があります。コンパイル時間の大部分は、最適化前に、設計を FPGA ファブリックに正しくマッピングする手順を見つけることに費やされます。FPGA の高度な構成の可能性こそが、アルゴリズムの空間実装が優れたパフォーマンスを実現できる理由です。

クロック周波数

FPGA は非常に柔軟で構成変更が可能ですが、構成変更の自由度の高さは、CPU やその他の固定演算アーキテクチャーに組み込まれた同等の設計と比較し、FPGA の動作周波数が低下する代償を伴います。しかし、これは問題ではありません。FPGA の空間的なアーキテクチャーは、クロック周波数の不足を十分に補います。それは、FPGA の領域全体に分散された多くの独立した演算が同時に実行されるためです。簡単に言うと、FPGA は構成可能な設計であるため、他のアーキテクチャーよりも動作周波数は低いが、クロックサイクルごとに処理されるイベント数が多いため、周波数の低さを補っています。アクセラレーターのベンチマークや比較を行う際には、演算処理能力（例えば、1 秒あたりの演算回数）を比較するべきであり、周波数自体を比較するべきではありません。

とはいえ、FPGA のリソース利用率が 100% に近づくと、動作周波数が低下し始める可能性があります。これは、デバイス上の信号ルーティング・リソースが過剰に使用されていることが原因です。これを解決するにはいくつかの方法がありますが、通常はコンパイル時間が増加する代償を伴います。しかし、周波数低下に対処する対策を講じる覚悟がない限り、ほとんどのアプリケーションでは FPGA のリソースの 80～90% 以上を使用することは避けるのが最善です。

推奨事項

FPGA 上のリソースの消費は、いずれの場合も 90% を超えないようにしてください。複数のリソースを同時に使用する場合、90% を超えてはなりません。このしきい値を超えると、ルーティング・リソースが枯渇し、動作周波数が低下する可能性があります。これに対処するには、FPGA の低レベルの調査に踏み込む必要があります。

まとめ

本章では、コンパイラーがアルゴリズムを FPGA の空間アーキテクチャーにマッピングする方法について説明しました。また、FPGA が私たちのアプリケーションにとって有用であるか判断するのに役立つ概念や、コード開発を素早く開始し、実行に移すのに役立つ概念についても説明しました。この出発点から、ベンダーのプログラミングおよび最適化マニュアルを参照し、FPGA 向けのコードを記述する準備は整いました! FPGA は、他のアクセラレーターではうまく達成できないパフォーマンスや、アプリケーションを実現する能力を備えているため、念頭に置いておくべき手段の上位に位置づけておくべきです。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 18 章　ライブラリー

私たちは本書全体を、*独自のコードを記述する*という目的の推進に費やしてきました。最近、私たちはようやく、優れたプログラマーが、そのまま使えるコードをすでに開発していることを確認できるようになりました。ライブラリーは作業をこなすのに最適な方法です。これは怠けているわけではなく、他人の仕事をやり直すよりも、もっとやるべきことがあるからです。

この章では、次の 3 つの異なるライブラリーの機能セットについて説明します:

1. SYCL 仕様で定義されたビルトイン関数
2. C ++ 標準ライブラリー
3. oneAPI DPC++ ライブラリー（oneDPL）でサポートされている C++17 並列アルゴリズム

SYCL は、ホストコードとデバイスコードで共有される共通機能を提供する豊富なビルトイン関数セットを定義します。すべての SYCL 実装はこれらの関数をサポートしているため、すべての SYCL デバイスで主要な数学ライブラリーが利用できます。

C++ 標準ライブラリーは、すべての SYCL 実装によってデバイスコードでサポートされるとは限りません。ただし、DPC++ コンパイラー（および他のコンパイラー）は、これを SYCL の拡張機能としてサポートしているため、ここではその拡張機能の制限について簡単に説明します。

最後に、oneAPI DPC++ ライブラリー（oneDPL）は、SYCL で実装された C++17 アルゴリズムに基づく一連のアルゴリズムを提供し、SYCL プログラマーに生産性の高いソリューションを提供します。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_18

これにより、CPU、GPU、FPGA にわたるプログラミングの労力を最小限に抑えることができます。oneDPL は SYCL 2020 の一部ではありませんが、SYCL 上に実装されているため、どの SYCL 2020 対応コンパイラーとも互換性があるはずです。

ビルトイン関数

SYCL は、さまざまなデータ型をサポートする豊富なビルトイン関数を提供します。これらのビルトイン関数は、ホストおよびデバイスの `sycl` 名前空間で使用でき、次のように分類できます:

- **浮動小数点演算関数:** `asin`、`acos`、`log`、`sqrt`、`floor` など。
- **整数関数:** `abs`、`max`、`min` など。
- **共通関数:** `clamp`、`smoothstep` など。
- **幾何関数:** `cross`、`dot`、`distance` など。
- **関係関数:** `isequal`、`isless`、`isfinite` など。

この広範な関数のコレクションに関するドキュメントは、SYCL 2020 仕様とオンライン・ドキュメント (`registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html` の 4.17.5 節から 4.17.9) に記載されています。

一部のコンパイラーでは、これらの関数の精度を制御するオプションが提供されています。たとえば、DPC++ コンパイラーは、`-mfma`、`-ffast-math`、`-ffp-contract=fast` などいくつかのオプションを提供します。同様のオプション (およびそのデフォルト値) の可用性を理解するには、SYCL 実装のドキュメントを確認してください。

SYCL ビルトイン関数のいくつかは、C++ 標準ライブラリーに同等の関数があります (例: `sycl::log` や `std::log`)。SYCL 実装では、デバイスコード内での C++ 標準ライブラリー関数の呼び出しをサポートする必要はありませんが、一部の実装 (DPC++ など) ではサポートされます。

図 18-1 は、デバイスコードで C++ `std::log` 関数と SYCL ビルトイン `sycl::log` 関数の両方を使用する例を示しています。DPC++ コンパイラー実装を使用すると、両方の関数は同じ数値結果を生成します。この例では、ビルトイン関係関数 `sycl::isequal` を使用して、`std::log` と `sycl::log` の結果を比較します。

```
constexpr int size = 9;
std::array<float, size> a;
std::array<float, size> b;

bool pass = true;

for (int i = 0; i < size; ++i) {
   a[i] = i;
   b[i] = i;
}

queue q;

range sz{size};

buffer<float> bufA(a);
buffer<float> bufB(b);
buffer<bool> bufP(&pass, 1);

q.submit([&](handler &h) {
   accessor accA{bufA, h};
   accessor accB{bufB, h};
   accessor accP{bufP, h};

   h.parallel_for(size, [=](id<1> idx) {
     accA[idx] = std::log(accA[idx]);
     accB[idx] = sycl::log(accB[idx]);
     if (!sycl::isequal(accA[idx], accB[idx])) {
       accP[0] = false;
     }
   });
});
```

図 18-1. std::log と sycl::log を使用

SYCL 2020 仕様では、SYCL 数学関数の実装が、特定のハードウェア・ターゲットの対応する C および C++ 標準数学関数とまったく同じ数値結果を生成する必要はないことに注意してください。仕様では、さまざまなハードウェア・プラットフォームの特性と制限を考慮して、実装に一定幅を持たせることが許容されています。したがって、図 18-1 に示すコード例に示すように、SYCL 実装では実際に一致する結果を生成することができます。

`sycl::` を使用ビルトイン関数のプリフィクス

名前の先頭に `sycl::` プリフィクスを付けて SYCL 組み込み関数を呼び出すことを強く推奨します。`sqrt()` だけを呼び出すと、“`using namespace sycl;`” 宣言されていても、すべての実装で SYCL ビルトイン関数が呼び出される保証はありません。

SYCL ビルトイン関数は、常にビルトイン名の前に `sycl::` を付けて呼び出す必要があります。このアドバイスに従わないと、奇妙で移植不可能な結果が生じる可能性があります。

移植可能なコードを記述する際は、`namespace sycl` の使用を避け、代わりに `std::` および `sycl::` 名前空間を明示的に使用することを推奨します。明示的にすることで、特定の SYCL 実装で解決できない競合が発生する可能性を排除します。これにより、将来コードのデバッグが容易になる可能性もあります（たとえば、実装で `std::` および `sycl::` 名前空間の数学関数に異なる精度保証が提供される場合など）。

C++ 標準ライブラリー

前述のように、SYCL 仕様では、C++ 標準ライブラリーの関数がデバイスコードでサポートされる保証はされません。これらの関数をサポートするコンパイラーはいくつかあります。これにより、既存の C++ コードの SYCL デバイスへのオフロードが簡素化され、実装の詳細として SYCL を使用するライブラリーの作成が容易になります（たとえば、関数をライブラリーに渡すユーザーは、SYCL 固有の機能を使用せずにその関数を作成できます）。

距離は異なる場合がある

`std::` 名前空間の関数に対するデバイスコードのサポートは、SYCL 実装によって異なるため、C++ 標準ライブラリーを採用したカーネルが複数の SYCL コンパイラーと実装の間で移植可能であるかどうかは不明です。

DPC++ コンパイラーは、テスト済みの C++ 標準 API のセットと互換性があるため、対応する C++ ヘッダーファイルをインクルードし、`std` 名前空間を使用するだけです。これらの API はすべて、一般的な C++ ホスト・アプリケーションで使用されるのと同じように、デバイスカーネルで使用できます。図 18-2 は、デバイスコードで `std::swap` を使用する例を示しています。

```
int main() {
   std::array<int, 2> arr{8, 9};
   buffer<int> buf{arr};

   {
       host_accessor host_A(buf);
       std::cout << "Before: " << host_A[0] << ", "
                 << host_A[1] << "\n";
   }   // 今後のカーネルが buf を操作できるように
       // host_A のスコープを終了

   queue q;
   q.submit([&](handler  &h) {
       accessor a{buf, h};
       h.single_task([=]() {
           // std::swap を呼び出し
           std::swap(a[0], a[1]);
       });
   });

   host_accessor host_B(buf);
   std::cout << "After: " << host_B[0] << ", " << host_B[1]
             << "\n";
   return 0;
}
```

サンプルの出力:

```
8, 9
9, 8
```

図 18-2. デバイスコードで std::swap を使用

図 18-3 には、執筆時点で CPU、GPU、FPGA デバイスの SYCL カーネルで使用したテスト済みの C++ 標準 API が（“Y” 付きのリスト）示されています。空白は、本書の発行時点ではカバー範囲が不完全であることを示します（3 種類のデバイスすべてがカバーされていない）。

C++ Standard API	libstdc++	libc++	MSVC
std::abs	Y	Y	Y
std::acos	Y	Y	Y
std::acosh	Y	Y	Y
std::add_const	Y	Y	Y
std::add_cv	Y	Y	Y
std::add_volatile	Y	Y	Y
std::alignment_of	Y	Y	Y
std::all_of	Y	Y	Y
std::any_of	Y	Y	Y
std::arg	Y	Y	Y
std::array	Y	Y	Y
std::asin	Y	Y	Y
std::asinh	Y	Y	Y
std::assert	Y	Y	Y
std::atan	Y	Y	Y
std::atan2	Y	Y	Y
std::atanh	Y	Y	Y
std::binary_negate	Y	Y	Y
std::binary_search	Y	Y	Y
std::bit_and	Y	Y	Y
std::bit_not	Y	Y	Y
std::bit_or	Y	Y	Y
std::bit_xor	Y	Y	Y
std::cbrt	Y	Y	Y
std::ceil	Y	Y	Y
std::common_type	Y	Y	Y
std::complex	Y	Y	Y
std::conditional	Y	Y	Y
std::cos	Y	Y	Y
std::copy	Y	Y	Y
std::copy_backward	Y	Y	Y
std::copy_if	Y	Y	Y
std::copy_n	Y	Y	Y
std::copysign	Y	Y	Y
std::copysignf	Y	Y	Y
std::cos	Y	Y	Y
std::cosh	Y	Y	Y
std::count	Y	Y	Y
std::count_if	Y	Y	Y
std::decay	Y	Y	Y
std::declval	Y	Y	Y
std::divides	Y	Y	Y
std::enable_if	Y	Y	Y
std::equal_range	Y	Y	Y
std::equal_to	Y	Y	Y
std::erf	Y	Y	Y
std::erfc	Y	Y	Y
std::exp	Y	Y	Y
std::exp	Y	Y	Y
std::exp2	Y	Y	Y
std::expm1	Y	Y	Y
std::extent	Y	Y	Y
std::fabs	Y	Y	Y
std::fdim	Y	Y	Y
std::fill	Y	Y	Y
std::fill_n	Y	Y	Y
std::find	Y	Y	Y
std::find_if	Y	Y	Y
std::find_if_not	Y	Y	Y
std::floor	Y	Y	Y
std::fmax	Y	Y	Y
std::fmaxf	Y	Y	Y
std::fmin	Y	Y	Y
std::fminf	Y	Y	Y
std::fmod	Y	Y	Y
std::for_each	Y	Y	Y

C++ Standard API	libstdc++	libc++	MSVC
std::for_each_n	Y	Y	Y
std::forward	Y	Y	Y
std::frexp	Y	Y	Y
std::generate	Y	Y	Y
std::generate_n	Y	Y	Y
std::greater	Y	Y	Y
std::greater_equal	Y	Y	Y
std::hypot	Y	Y	Y
std::ilogb	Y	Y	Y
std::imag	Y	Y	Y
std::initializer_list	Y	Y	Y
std::integral_constant	Y	Y	Y
std::is_arithmetic	Y	Y	Y
std::is_assignable	Y	Y	Y
std::is_base_of	Y	Y	Y
std::is_base_of_v(c++17)	Y	Y	Y
std::is_compound	Y	Y	Y
std::is_const	Y	Y	Y
std::is_constructible	Y	Y	Y
std::is_convertible	Y	Y	Y
std::is_copy_assignable	Y	Y	Y
std::is_copy_constructible	Y	Y	Y
std::is_default_constructible	Y	Y	Y
std::is_destructible	Y	Y	Y
std::is_empty	Y	Y	Y
std::is_fundamental	Y	Y	Y
std::is_heap	Y	Y	Y
std::is_heap_until	Y	Y	Y
std::is_literal_type	Y	Y	Y
std::is_member_pointer	Y	Y	Y
std::is_move_assignable	Y	Y	Y
std::is_move_constructible	Y	Y	Y
std::is_object	Y	Y	Y
std::is_permutation	Y	Y	Y
std::is_pod	Y	Y	Y
std::is_reference	Y	Y	Y
std::is_same	Y	Y	Y
std::is_scalar	Y	Y	Y
std::is_signed	Y	Y	Y
std::is_sorted	Y	Y	Y
std::is_sorted_until	Y	Y	Y
std::is_standard_layout	Y	Y	Y
std::is_trivial	Y	Y	Y
std::is_trivially_assignable	Y	Y	Y
std::is_trivially_constructible	Y	Y	Y
std::is_trivially_copyable	Y	Y	Y
std::is_unsigned	Y	Y	Y
std::is_volatile	Y	Y	Y
std::isgreater	Y	Y	Y
std::isgreaterequal	Y	Y	Y
std::isinf	Y	Y	Y
std::isless	Y	Y	Y
std::islessequal	Y	Y	Y
std::isnan	Y	Y	Y
std::isunordered	Y	Y	Y
std::ldexp	Y	Y	Y
std::less	Y	Y	Y
std::less_equal	Y	Y	Y
std::lgamma	Y	Y	Y
std::log	Y	Y	Y
std::log10	Y	Y	Y
std::log1p	Y	Y	Y
std::log2	Y	Y	Y
std::logb	Y	Y	Y
std::logical_and	Y	Y	Y
std::logical_not	Y	Y	Y

C++ Standard API	libstdc++	libc++	MSVC
std::logical_or	Y	Y	Y
std::lower_bound	Y	Y	Y
std::make_heap	Y	Y	Y
std::max	Y	Y	Y
std::min	Y	Y	Y
std::minus	Y	Y	Y
std::modf	Y	Y	Y
std::modulus	Y	Y	Y
std::move	Y	Y	Y
std::move	Y	Y	Y
std::move_backward	Y	Y	Y
std::move_if_noexcept	Y	Y	Y
std::multiplies	Y	Y	Y
std::nan	Y	Y	Y
std::nanf	Y	Y	Y
std::nearbyint	Y	Y	Y
std::nearbyintf	Y	Y	Y
std::negate	Y	Y	Y
std::nextafter	Y	Y	Y
std::none_of	Y	Y	Y
std::norm	Y	Y	Y
std::not_equal_to	Y	Y	Y
std::nullptr_t	Y	Y	Y
std::numeric_limits<float>::infinity	Y	Y	Y
std::numeric_limits<float>::lowest	Y	Y	Y
std::numeric_limits<float>::max	Y	Y	Y
std::numeric_limits::quiet_NaN	Y	Y	Y
std::optional	Y	Y	Y
std::pair	Y	Y	Y
std::partial_sort	Y	Y	Y
std::partial_sort_copy	Y	Y	Y
std::plus	Y	Y	Y
std::polar	Y	Y	Y
std::pop_heap	Y	Y	Y
std::pow	Y	Y	Y
std::proj	Y	Y	Y
std::push_heap	Y	Y	Y
std::rand	Y	Y	Y
std::real	Y	Y	Y
std::reduce	Y	Y	Y
std::ref/cref	Y	Y	Y
std::reference_wrapper	Y	Y	Y
std::remainder	Y	Y	Y
std::remove_all_extents	Y	Y	Y
std::remove_const	Y	Y	Y
std::remove_cv	Y	Y	Y
std::remove_extent	Y	Y	Y
std::remove_volatile	Y	Y	Y
std::remquo	Y	Y	Y
std::round	Y	Y	Y
std::roundf	Y	Y	Y
std::sin	Y	Y	Y
std::sinh	Y	Y	Y
std::sort	Y	Y	Y
std::swap	Y	Y	Y
std::tan	Y	Y	Y
std::tanh	Y	Y	Y
std::tgamma	Y	Y	Y
std::transform	Y	Y	Y
std::trunc	Y	Y	Y
std::truncf	Y	Y	Y
std::tuple	Y	Y	Y
std::unary_negate	Y	Y	Y
std::upper_bound	Y	Y	Y

図 18-3. CPU/GPU/FPGA を網羅したライブラリーのサポート（本書の出版時点）

テスト済みの標準 C++ API は、ホスト CPU で動作する gcc 7.5.0+ の libstdc++（GNU）、clang 11.0+ の libc++（LLVM）、および Microsoft Visual Studio 2019 以降の MSVC 標準 C++ ライブラリーでもサポートされています。

Linux では、GNU libstdc++ が DPC++ コンパイラーのデフォルトの C++ 標準ライブラリーであるため、コンパイルまたはリンクオプションは必要ありません。libc++ を使用する場合、コンパイルオプション `-stdlib=libc++ -nostdinc++` を使用して libc++ を有効にし、システムから C++ std ヘッダーをインクルードしないようにします。

DPC++ コンパイラーは、Linux 上の SYCL カーネルで libc++ を使用して検証されていますが、ランタイムは libstdc++ ではなく libc++ を使用してリビルドする必要があります。詳細については、https://intel.github.io/llvm-docs/GetStartedGuide.html#build-dpc-toolchain-with-libc-library（英語）をご覧ください。これらの追加手順があるため、特別な理由がない限り、libc++ は一般に使用が推奨される C++ 標準ライブラリーではありません。

アーキテクチャー間の移植性を実現するため、図 18-3 で `std::` 関数に "Y" のマークが付いていない場合は、未テストのターゲットデバイスでアプリケーションを実行する際に機能上の誤り（またはビルドの失敗）が発生しないように注意する必要があります。

---

oneAPI DPC++ ライブラリー（oneDPL）

C++17 では、C++ 標準ライブラリーで定義されているアルゴリズムの並列バージョンが導入されました。シリアル版とは異なり、並列アルゴリズムはそれぞれ、最初の引数として*実行ポリシー*を受け入れます。この実行ポリシーは、アルゴリズムの実行方法を示します。

大まかに言えば、実行ポリシーは、スレッド、SIMD 命令、または両方を使用してアルゴリズムを並列化できるかどうかを実装に通知します。実行ポリシーとして、`seq`、`unseq`、`par`、または `par_unseq` のいずれかの値を渡すことができます。意味は図 18-4 に示されています。

実行ポリシー	説明
`seq`	シーケンシャル実行
`unseq`	シーケンスなし SIMD 実行。このポリシーでは、提供されるすべての関数が SIMD で安全に実行できることが求められます。
`par`	マルチスレッドによる並列実行
`par_unseq`	`unseq` と `par` の組み合わせ効果

図 18-4. 実行ポリシー

oneDPL は標準実行ポリシーを拡張して、SYCL デバイスのサポートを提供します。これらの SYCL 対応実行ポリシーは、アルゴリズムの*実行方法*だけでなく、*実行場所*も指定します。SYCL 対応ポリシーは標準の C++ 実行ポリシーを継承し、SYCL デバイスやキューをカプセル化して、オプションのカーネル名を設定できるようにします。SYCL 対応の実行ポリシーは、実行をサポートするすべての標準 C++ アルゴリズムで利用できます。

oneDPL は、DPC++ だけでなくすべての SYCL コンパイラーをサポートするように設計されています。

oneDPL と SYCL 対応実行ポリシーを使用する前に、いくつかのヘッダーファイルを追加する必要があります。どのヘッダーをインクルードするかは、使用するアルゴリズムによって異なります。一般的な例としては次のようなものがあります:

- `#include <oneapi/dpl/algorithm>`
- `#include <oneapi/dpl/numeric>`
- `#include <oneapi/dpl/memory>`

SYCL 実行ポリシー

現在、並列非シーケンスポリシー (`par_unseq`) を持つアルゴリズムのみを SYCL デバイスに安全にオフロードできます。この制限は、SYCL のワーク項目による前方進行の保証に起因しており、これは他の実行ポリシー (par など) の要件と互換性がありません。

SYCL 実行ポリシーを使用するには、次の 3 つの手順があります:

1. コードに `#include <oneapi/dpl/execution>` を追加します。
2. 標準ポリシータイプ、テンプレート引数として一意のカーネル名のクラスタイプ (オプション)、および次のコンストラクター引数のいずれかを指定して、ポリシー・オブジェクトを作成します:

SYCL キュー
SYCL デバイス
SYCL デバイスセレクター
異なるカーネル名を持つ既存のポリシー・オブジェクト

3. 作成したポリシー・オブジェクトをアルゴリズムに渡します。

`oneapi::dpl::execution::dpcpp_default` オブジェクトは、デフォルトのカーネル名とデフォルトキューで作成された定義済の `device_policy` です。これは、カスタム・ポリシー・オブジェクトの作成に使用することも、デフォルトの選択肢ではアルゴリズムを呼び出すときに直接渡すこともできます。

図 18-5 は、`policy` クラスと関数を参照するときに `using namespace oneapi::dpl::execution;` ディレクティブを使用することを前提とした例を示しています。

```
auto policy_b = device_policy<parallel_unsequenced_policy,
                              class PolicyB>{
   sycl::device{sycl::gpu_selector{}}};
std::for_each(policy_b, …);
auto policy_c =
    device_policy<parallel_unsequenced_policy,
                 class  PolicyC>{sycl::default_selector{}};
std::for_each(policy_c, …);
auto policy_d =
   make_device_policy<class PolicyD>(default_policy);
std::for_each(policy_d, …);
auto policy_e =
   make_device_policy<class PolicyE>(sycl::queue{});
std::for_each(policy_e, …);
```

図 18-5. 実行ポリシーを作成

バッファーで oneDPL を使用

C++ 標準ライブラリーのアルゴリズムはすべて反復子に基づいています。これらのアルゴリズムに SYCL バッファーを渡すため、oneDPL は 2 つの特別なヘルパー関数、`oneapi::dpl::begin` と `oneapi::dpl::end` を定義します。

これらの関数は SYCL バッファーを受け取り、次の要件を満たす未指定タイプのオブジェクトを返します:

- `CopyConstructible`、`CopyAssignable`、および比較可能な `==` と `!=` 演算子。
- 有効な式は、`a + n`、`a - n`、`-a - b` であり、`a` および `b` はタイプのオブジェクトで、`n` は整数値です。
- 引数のない `get_buffer` メソッド。このメソッドは、`oneapi::dpl::begin` 関数と `oneapi::dpl::end` 関数に渡された SYCL バッファーを返します。

これらのヘルパー関数を使用するには、コードに `#include <oneapi/dpl/iterator>` を追加します。この機能はデフォルトでは含まれていません。USM を使用する場合、これらの反復子は必要ないためです (これについては後ほど説明します)。

図 18-6 のコードは、`std::fill` 関数を `begin/end` ヘルパーと組み合わせて SYCL バッファーを埋める方法を示しています。アルゴリズムは `std::` 名前空間にあり、実行ポリシーのみが非標準の名前空間にあることに注意してください。これはタイプミスではありません。C++ 標準ライブラリーでは、このようなコーディング・パターンをサポートするため、実装が独自の実行ポリシーを定義することを明示的に許可しています。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp>

int main() {
  sycl::queue q;
  sycl::buffer<int> buf{1000};

  auto buf_begin = oneapi::dpl::begin(buf);
  auto buf_end = oneapi::dpl::end(buf);

  auto policy = oneapi::dpl::execution::make_device_policy<
      class fill>(q);
  std::fill(policy, buf_begin, buf_end, 42);

  return 0;
}
```

図 18-6. std::fill を使用

図 18-7 のコードは、デフォルトポリシーと通常の（ホスト側の）反復子を使用したさらに単純なバージョンを示しています。この場合、一時的に SYCL バッファーが作成され、データがそのバッファーにコピーされます。デバイス上で一時バッファーの処理が完了すると、データはホストにコピーされます。ホストとデバイス間のデータ移動と、バッファーの作成および破棄によるオーバーヘッドを削減するため、既存の SYCL バッファーを直接操作することを推奨します（可能な場合）。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp>

int main() {
   std::vector<int> v(100000);
   std::fill(oneapi::dpl::execution::dpcpp_default,
             v.begin(), v.end(), 42);

   if (v[788] == 42)
     std::cout << "passed" << std::endl;
   else
     std::cout << "failed" << std::endl;

   return 0;
}
```

図 18-7. デフォルトポリシーとホスト側の反復子で std::fill を使用

図 18-8 は、提供された検索シーケンス内の各値に対して入力シーケンスのバイナリー検索を実行する例を示しています。検索シーケンスの i 番目の要素の検索結果（入力シーケンスで検索値が検出されたかどうかを示すブール値）が、結果シーケンスの i 番目の要素に割り当てられます。このアルゴリズムは、結果が割り当てられたシーケンスの最後の要素の次を指す反復子を返します。このアルゴリズムでは、入力シーケンスが提供されるコンパレーターでソートされていることを前提としています。コンパレーターが指定されていない場合、< 演算子を使用して要素を比較する関数オブジェクトが使用されます。

前述の説明の複雑さから、デバッグとチューニングに時間を要する可能性のあるアルゴリズムを独自の実装を記述するのではなく、可能な場合はライブラリー関数を活用する必要があることがわかります。私たちが利用するライブラリーの作成者は、対象とするデバイス・アーキテクチャーの内部構造の専門家であることが多く、私たちが知らない情報にアクセスできる可能性があります。そのため、最適化されたライブラリーが利用可能な場合は常にそれを活用すべきです。

```
#include <oneapi/dpl/algorithm>
#include <iostream>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp>

using namespace sycl;

int main() {
   buffer<uint64_t, 1> kB{range<1>(10)};
   buffer<uint64_t, 1> vB{range<1>(5)};
   buffer<uint64_t, 1> rB{range<1>(5)};
   {
     host_accessor k{kB};
     host_accessor v{vB};

     // データを初期化し、ソート
     k[0] = 0;
     k[1] = 5;
     k[2] = 6;
     k[3] = 6;
     k[4] = 7;
     k[5] = 7;
     k[6] = 8;
     k[7] = 8;
     k[8] = 9;
     k[9] = 9;

     v[0] = 1;
     v[1] = 6;
     v[2] = 3;
     v[3] = 7;
     v[4] = 8;
   }

   //  dpc++ イテレーターを作成
   auto k_beg = oneapi::dpl::begin(kB);
   auto k_end = oneapi::dpl::end(kB);
   auto v_beg = oneapi::dpl::begin(vB);
   auto v_end = oneapi::dpl::end(vB);
   auto r_beg = oneapi::dpl::begin(rB);

   // 既存のポリシーから名前付きポリシーを作成
   auto policy = oneapi::dpl::execution::make_device_policy<
       class bSearch>(oneapi::dpl::execution::dpcpp_default);
```

図 18-8. Binary_search を使用

```
// アルゴリズムを呼び出し
oneapi::dpl::binary_search(policy, k_beg, k_end, v_beg,
                           v_end, r_beg);

// データをチェック
host_accessor  r{rB};
if ((r[0] == false) && (r[1] == true) &&
     (r[2] == false) && (r[3] == true) && (r[4] == true)) {
    std::cout << "Passed. \nRun on "
              << policy.queue()
                     .get_device()
                     get_info<info::device::name>()
              << "\n";
} else
    std::cout << "failed: values do not match.\n";

  return 0;
}
```

図 18-8.（続き）

図 18-8 に示すコード例は、oneDPL を SYCL バッファーと組み合わせて使用する 3 つの一般的な手順を示しています:

1. バッファーから SYCL 反復子を作成します。
2. 既存のポリシーから名前付きポリシーを作成します。
3. 並列アルゴリズムを呼び出します。

USM で oneDPL を使用

このセクションでは、oneDPL を USM と組み合わせて使用する 2 つの方法を説明します:

- USM ポインターを介して
- USM アロケーターを介して

バッファーとは異なり、アルゴリズムに渡される反復子として USM ポインターを直接使用できます。具体的には、割り当ての開始と終了（1 つ先）へのポインターを並列アルゴリズムに渡すことができます。

実行時に未定義の動作を回避するには、実行ポリシーと割り当て自体が同じキュー、またはコンテキストに対して作成されていることを確認することが重要です。（これは oneDPL に固有のものではなく、USM を使用するときは常にコンテキストに細心の注意を払う必要があることに注意してください）

同じ USM 割り当てを複数のアルゴリズムで処理する場合、インオーダー・キューを使用するか、各アルゴリズムの完了を明示的に待機してから次のアルゴリズムで同じ割り当てを使用するか選択できます（これは、USM を使用する場合の一般的な操作順序です）。また、図 18-9 に示すように、ホスト上のデータにアクセスする前に完了を待機するように注意する必要があります。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <sycl/sycl.hpp>

int main() {
   sycl::queue q;
   const int n = 10;
   int* h_head = sycl::malloc_host<int>(n, q);
   int* d_head = sycl::malloc_device<int>(n, q);
   std::fill(oneapi::dpl::execution::make_device_policy(q),
             d_head, d_head + n, 78);
   q.wait();

   q.memcpy(h_head, d_head, n * sizeof(int));
   q.wait();

   if (h_head[8] == 78)
     std::cout << "passed" << std::endl;
   else
     std::cout << "failed" << std::endl;

   sycl::free(h_head, q);
   sycl::free(d_head, q);
   return 0;
}
```

図 18-9. USM ポインターで oneDPL を使用

別の方法として、図 18-10 に示すように、並列スキャンを使用するとさらに効果的です。このアプローチでは、`std::vector` は（通常どおり）独自のメモリーを管理しますが、必要なメモリーは `sycl::malloc_shared` への内部呼び出しを介して割り当てます。次に、`begin()` および `end()` メンバー関数は、USM 割り当てをステップ実行する反復子を返します。このプログラミング・スタイルは、特にコンテナーとアルゴリズムを既に使用している既存の C++ コードを移行する場合に非常に便利です。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <sycl/sycl.hpp>

int main() {
  sycl::queue q;
  const int n = 10;
  sycl::usm_allocator<int, sycl::usm::alloc::shared> alloc( q);
  std::vector<int, decltype(alloc)> vec(n, alloc);

  std::fill(oneapi::dpl::execution::make_device_policy(q),
            vec.begin(), vec.end(), 78);
  q.wait();

  return 0;
}
```

図 18-10. USM アロケーターで oneDPL を使用

SYCL 実行ポリシーによるエラー処理

第 5 章で説明されているように、SYCL エラー処理モデルは 2 種類のエラーをサポートしています。*同期*エラーの場合、ランタイムは例外をスローしますが、*非同期*エラーはプログラム実行中の指定されたタイミングに非同期エラーハンドラーによってのみ処理されます。

SYCL 対応実行ポリシーで実行されるアルゴリズムの場合、すべてのエラー（同期または非同期）の処理は呼び出し側の責任となります。具体例:

- アルゴリズムによって明示的に例外がスローされることはありません。
- SYCL 同期例外を含む、ホスト CPU 上のランタイムによってスローされた例外は、呼び出し元に渡されます。
- SYCL 非同期エラーは oneDPL では処理されないため、通常の SYCL 非同期例外メカニズムを使用して呼び出し元が処理する必

要があります（何らかの処理が必要な場合）。

まとめ

異種アプリケーションでは、共通の機能や並列パターンの書き換えやテストに時間を無駄にしないため、可能な限りライブラリーを使用する必要があります。すべてを自身で記述するのではなく、他の開発者の実績を活用する必要があります。また、アプリケーション開発を簡素化し、（多くの場合）優れたパフォーマンスを実現するため、可能な限りこのアプローチを使用すべきです。

この章では、すべての SYCL 開発者が知っておくべきと思われる 3 つのライブラリー機能セットを簡単に紹介しました:

1. 一般的な数学演算のための SYCL ビルトイン関数
2. その他の一般的な操作のための標準 C++ ライブラリー
3. 完全なカーネルのための C++17 並列アルゴリズム（oneDPL でサポート）

いずれのライブラリーでも、運用環境で使用する前に、どのデバイス、コンパイラー、実装がテストされ、サポートされているかを理解することが重要です。これは SYCL 特有のアドバイスではありませんが、覚えておく価値があります。SYCL のような移植可能なプログラミング・ソリューションの潜在的なターゲットの数は膨大であり、どのライブラリーが目標に合致するか特定するのはプログラマーとしての私たちの責任です。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 19 章　メモリーモデルとアトミック

並列プログラマーを目指すのであれば、*メモリーの一貫性*は決して難解な概念ではありません。これにより、必要な時に必要な場所でデータを利用でき、その値が期待どおりであることを確実にできます。この章では、プログラムを順調に稼働させるために習得すべき重要な事項を明確にします。これは、SYCL に限ったここではありません。

プログラミング言語のメモリー（一貫性）モデルに関する基本的な理解は、メモリーの同時更新（同じカーネル内の複数のワーク項目、複数のデバイス、またはその両方からの更新）を許可するプログラマーには不可欠です。これはメモリー割り当て方法に関係なく当てはまり、バッファーを使用する場合も、USM 割り当てを使用する場合も、この章の内容は重要です。

これまでの章では、ワーク項目が完全に独立したデータ上で動作するか、言語やライブラリー機能によって表現できる構造化された通信パターンを使用してデータを共有する、シンプルなカーネルの開発に焦点を当ててきました。複雑で現実に即したカーネルを作成するにつれて、ワーク項目が構造化されていない方法で通信する必要に遭遇する可能性が高くなります。そのため、メモリーモデルが SYCL 言語の機能やターゲット・ハードウェアのパフォーマンスとの関連性を理解することは、正しく、移植性があり、効率的なプログラムを設計する必要条件となります。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_19

実行スレッド

C++17 では、並列処理や並行処理に関連するライブラリー機能（並列アルゴリズムなど）の動作を説明するため、「実行スレッド」（単に「スレッド」と呼ばれることが多い）という概念が導入されました。
C++ のメモリー一貫性モデルと実行モデルは、この「スレッド」間の相互作用という観点から定義されます。

SYCL と C++ の比較を簡単にするため、本章では「スレッド」という用語を「実行スレッド」という意味で頻繁に使用します。SYCL のワーク項目は、弱い並列処理の進行保証を備えた C++ の実行スレッドに相当するため、これらの用語は互換的に使用しても問題ありません。ただし、SYCL 固有の概念を説明する際には、「ワーク項目」という用語を使用することもあります。

C++ のメモリー一貫性モデルは、ホスト上でのみ実行されるアプリケーションを作成するには十分ですが、異種システムをプログラミングする際に生じるある種の複雑性に対処するため、SYCL によって補完されています。具体的には、以下が可能でなければなりません:

- システム内のどのデバイスがどのタイプのメモリー割り当て（バッファーと USM）にアクセスできるか認識すること
- バリアとアトミック操作を使用することで、カーネル実行中の安全でない同時メモリーアクセス（データ競合）を防ぎます
- バリア、フェンス、アトミック操作、メモリー順序、メモリースコープを使用して、ワーク項目間の安全な通信を可能にします

- バリア、フェンス、アトミック操作、メモリー順序、メモリースコープを使用して、並列アプリケーションの動作を予期せず変更する可能性のある最適化を防ぎつつ、他の最適化は許可します

メモリーモデルは複雑なテーマですが、それには正当な理由があります。プロセッサーの設計者は、プロセッサーとアクセラレーターがコードを可能な限り効率良く実行することを重視しているためです。本章では、この複雑性を意図的に排除し、最も重要な概念と言語の特徴を明確にするために尽力しました。この章では、メモリーモデルを詳細に至るまで理解するだけでなく、多くの人にとって未知である並列プログラミングの重要な側面を楽しみながら学ぶ道へと踏み出していきます。この章に記載されている説明とサンプルコードを読んでも疑問が残る場合は、この章の最後に記載されているウェブサイトを参照するか、C++ および SYCL の仕様書を参照することを強くお勧めします。

メモリーモデルに含まれるものは？

ここでは、プログラミング言語にメモリーモデルが含まれる背景を詳しく説明し、並列プログラマーが理解しておくべきいくつかのコアコンセプトを紹介します:

- データ競合と同期
- バリアとフェンス
- アトミック操作
- メモリー順序

これらの概念を高度なレベルで理解することは、SYCL を用いた C++ における表現と使い方を正しく知るには必要なことです。並列プログラミング、特に C++ を用いたプログラミングに豊富な経験を持つ読者は、この先を読み飛ばしても構いません。

データ競合と同期

私たちがプログラムを記述する操作は、通常、単一のハードウェア命令やマイクロオペレーションに直接対応するものではありません。`data[i] += x` のような単純な加算演算は、いくつかの命令またはマイクロオペレーションのシーケンスに分解できます:

- メモリーから `data[i]` を一時レジスターにロードします。
- `data[i]` に `x` を加算した結果を計算します。
- 結果を `data[i]` に格納します。

これは、逐次アプリケーションの開発には心配ないことです。加算の 3 つのステージは、図 19-1 に示すように、期待どおりの順序で実行されます。

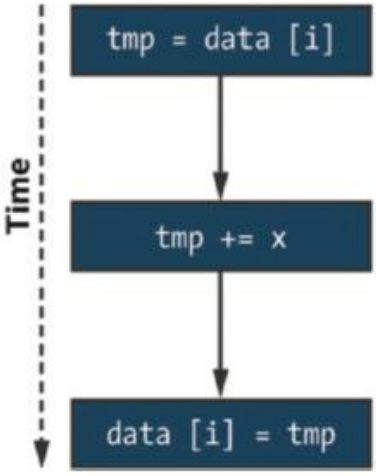

図 19-1. data[i] += x の逐次実行は、3 つの別々の操作に分割されます

並列アプリケーションの開発に移行すると複雑になります。同じデータに対して複数の操作が同時に適用される場合、操作によるデータビューの一貫性をどのように保証できるでしょう？ 図 19-2 に示す状況を考えてみましょう。ここでは、`data[i] += x` の 2 回の実行が 2 つのスレッドで交互に行われています。2 つのスレッドが異なる `i` の値を使用する場合、アプリケーションは正しく実行されます。両者が同じ `i` の値を使用した場合、双方ともメモリーから同じ値を読み込み、一方の結果がもう一方の結果によって上書きされます。これは、操作をスケジュールする方法の 1 つにすぎません。そして、アプリケーションの動作は、どのスレッドがどのデータに最初にアクセスするかに依存します。つまり、アプリケーションには*データ競合*が内在します。

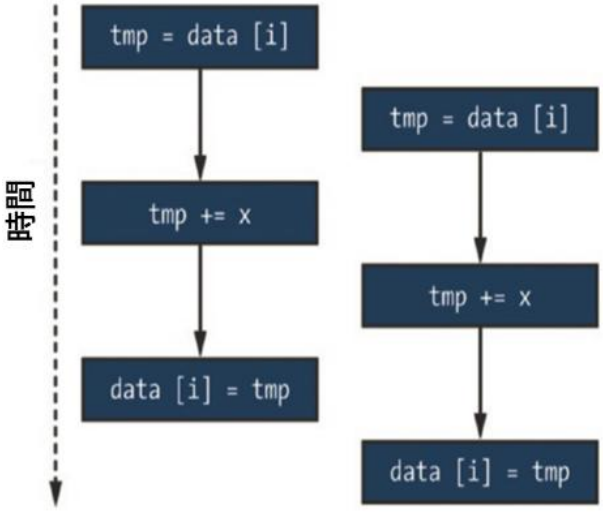

図 19-2. data[i] += x の同時実行で可能なインターリーブ

図 19-3 のコードと図 19-4 の出力は、これがいかに簡単に発生するか示しています。`M` が `N` 以上の場合、各スレッドで使用される `j` の値は一意になります。そうでない場合、`j` の値が競合し、正しく更新されない可能性があります。「失われる可能性がある」とは、データ競合を内在するプログラムでも、実装やハードウェアによる処理のスケジュール方法によっては、正しい答えが時々、あるいは常に生成される可能性があるためです。

コンパイラーもハードウェアも、このプログラムが何を意図しているのか、あるいは実行時に N と M の値がどうなるか把握することはできません。プログラムにデータ競合が発生する可能性があるか、また実行順序の影響を受ける可能性があるか理解するのは、プログラマーの責任です。

```
int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

q.parallel_for(N, [=](id<1> i) {
    int j = i % M;
    data[j] += 1;
 }).wait();

for (int i = 0; i < N; ++i) {
   std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

図 19-3. データ競合を含むカーネル

```
N = 2, M = 2:
data [0] = 1
data [1] = 1

N = 2, M = 1:
data [0] = 1
data [1] = 0
```

図 19-4. 図 19-3 のコードの出力例（N と M の値が小さい場合）

一般に、大規模並列 SYCL アプリケーションを開発する際には、個々のワーク項目が実行される正確な順序を気にする必要はありません。各カーネルには数百（あるいは数千）ものワーク項目が存在しますが、それらを特定の順序に強制すると、スケーラビリティーとパフォーマンスの両方に影響を及ぼします。むしろ、注力すべきは、正しく実行される移植性の高いアプリケーションの開発であり、それには、ワーク項目がデータを共有するタイミング、共有時に必要な保証、および有効な実行順序に関する情報をコンパイラー（およびハードウェア）にフィードバックする必要があります。

大規模並列アプリケーションでは、個々のワーク項目が実行される正確な順序を気にする必要はありません。

バリアとフェンス

同じグループ内のワーク項目間のデータ競合を防ぐ方法の一つは、ワークグループ・バリアと適切なメモリーフェンスを使用して、異なるスレッド間に同期を導入することです。図 19-5 に示すように、ワークグループ・バリアを使用して `data[i]` の更新順序を決定でき、カーネルの更新バージョンを図 19-6 に示します。
注: ワークグループ・バリアは異なるグループのワーク項目を同期しないため、この単純な例は、単一のワークグループでのみ正しく実行されることが保証されます。

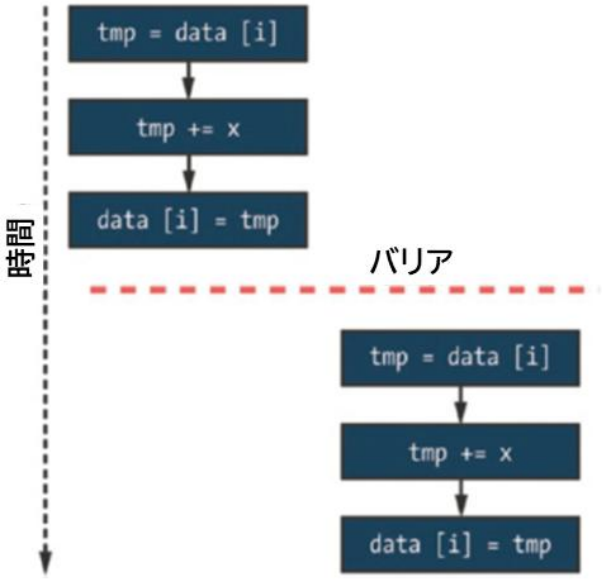

図 19-5. バリアで区切られた 2 つの data[i] += x の実行

```
int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

// ワークグループを 1 つだけ起動
// ワークグループ数 = global / local
range<1> global{N};
range<1> local{N};

q.parallel_for(nd_range<1>{global, local},
                [=](nd_item<1> it) {
                  int i = it.get_global_id(0);
                  int j = i % M;
                  for (int round = 0; round < N; ++round) {
                    // 1 ラウンドにつきワーク項目の更新は 1 件のみ許可
                    if (i == round) {
                       data[j] += 1;
                    }
                    group_barrier(item.get_group());
                  }
                })
    .wait();

for (int i = 0; i < N; ++i) {
  std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

図 19-6. データ競合を回避するバリア

このパターンの実装にバリアを使用することは可能ですが、一般には推奨されません。バリアを使用すると、グループ内の複数のワーク項目が特定の順序で実行されることが強制されるため、負荷が不均衡であるとインアクティブな状態が発生する可能性があるためです。また、必要以上の同期が発生する可能性もあります。異なるワーク項目が偶然異なる `i` の値を使用している場合、それらはバリアで同期を強制されることになります。

バリア同期は、ワークグループまたはサブグループ内のすべてのワーク項目が次のステージに進む前にカーネルのステージ完了を保証するのに役立ちますが、細粒度（場合によってはデータに依存する）同期には強引すぎます。より一般的な同期パターンについては、*アトミック*操作を検討する必要があります。

アトミック操作

アトミック操作は、データ競合を引き起こすことなく、メモリー位置への同時アクセスを可能にします。複数のアトミック操作が同じメモリーにアクセスする場合、それらの操作が重複しないことが保証されます。ただし、アクセスの一部のみがアトミック操作を使用している場合、その保証はありません。また、プログラマーは、アトミックの保証レベルが異なる操作を用いて、同じデータに同時にアクセスしないようにする責任があります。

同じメモリー位置に対してアトミック操作と非アトミック操作を同時に行うと、未定義の動作となります。

単純な加算をアトミック操作で表現すると、結果は図 19-8 のようになります。それぞれの更新は分割不可能なワークのチャンクとなり、アプリケーションは常に正しい結果を生成します。対応するコードを図 19-7 に示します。atomic_ref クラスとそのテンプレート引数の意味については、この章の後半で改めて説明します。

```
int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

q.parallel_for(N, [=](id<1> i) {
   int j = i % M;
   atomic_ref<int, memory_order::relaxed,
               memory_scope::system,
               access::address_space::global_space>
        atomic_data(data[j]);
   atomic_data += 1;
 }).wait();

for (int i = 0; i < N; ++i) {
   std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

図 19-7. アトミック操作を使用してデータ競合を回避

図 19-8. data[i] += x のインターリーブは、アトミック操作と並行して実行されます

ただし、これはあくまでも実行順序の一例に過ぎません。アトミック操作を使用すると、2 つの更新が重複しないことが保証されます（両方のスレッドが同じ `i` の値を使用する場合）。しかし、どちらのスレッドが先に実行されるかは保証されません。さらに重要なことに、これらのアトミック操作が、異なるスレッドの*非アトミック*操作に対してどのような順序で実行されるかについても何の保証もありません。

メモリー順序

逐次処理アプリケーション内であっても、最適化コンパイラーやハードウェアは、アプリケーションの動作を変えないように、処理の順序を自由に並べ替えることができます。つまり、アプリケーションはプログラマーが記述したとおり正確に動作しなければなりません。

残念ながら、この仮定に基づく保証は、並列プログラムの実行を考察する上で十分な根拠とはなり得ません。現在、2 つの要因による順序変更について考慮する必要があります。1 つは、コンパイラーとハードウェアが各シーケンシャル・スレッド内でステートメントの実行順序を変更する可能性があること、もう 1 つは、スレッド自体が任意の順序（場合によってはインターリーブされた順序）で実行される可能性があることです。スレッド間の安全な通信プロトコルを設計および実装するには、この順序変更を制限する必要があります。コンパイラーにメモリー順序に関する情報を提供することで、アプリケーションの意図した動作と互換性のないメモリー順序を変更する最適化を防ぐことができます。

一般に利用可能なメモリー順序付けには次の 3 つがあります:

1. *緩和された*メモリーの順序付け

2. *取得-解放*、または*解放-取得*のメモリー順序

3. *逐次的に一貫性のある*メモリー順序付け

メモリー順序が緩和されている場合、メモリー操作は制限なく順序を変更できます。緩和メモリーモデルの最も一般的な用途は、共有変数のインクリメントです（例えば、単一のカウンター、ヒストグラム計算中の値の配列など）。

取得/解放のメモリー順序付けの下では、アトミック変数を解放するスレッドと、同じアトミック変数を取得する別のスレッドが、これら 2 つのスレッド間の同期ポイントとして機能し、解放スレッドによって発行されたメモリーへの以前の書き込みが、取得スレッドから見えることを保証します。非公式ですが、アトミック操作とは、他のメモリー操作による副作用を他のスレッドに解放したり、他のスレッド上のメモリー操作による副作用を取得する操作であると考えることができます。このようなメモリーモデルは、スレッド間でメモリーを介して値をやり取りする場合に必要となるものであり、これは私たちが考えるよりも一般的かもしれません。 プログラムがロックを*取得*すると、通常は追加の計算を実行したり、メモリーの一部を変更してから、最終的にロックを*解放*します。

ロック変数のみがアトミックに更新されますが、ロックによって保護されたメモリーの更新はデータ競合から保護されることが期待されます。この動作は、メモリーの取得と解放の順序付けが正しく行われることを前提としており、緩和されたメモリー順序付けを使用してロックを実装しても機能しません。

逐次的に一貫性のあるメモリー順序付けの下では、取得/解放の順序付けの保証は有効ですが、それに加えて、すべてのアトミック操作の単一のグローバル順序が存在します。 このメモリー順序付けの動作は、3 つの中で最も直感的であり、逐次アプリケーションを開発する際に私たちが普段頼りにしている、本来の「あたかも～であるかのような」保証に最も近いものです。逐次一貫性を用いることで、すべてのスレッドがすべてのアトミック操作の全体的な順序に合意する必要があるため、スレッドのペア間ではなくグループ間の通信について推論することが容易になります。

プログラミング・モデルとデバイスの組み合わせによって、どのメモリー順序がサポートされているかを理解することは、移植性の高い並列アプリケーションを設計する上で不可欠な要素です。アプリケーションが必要とするメモリー順序を明確に記述することで、要求する動作がサポートされていない場合に（コンパイル時などに）、予測可能な形でアプリケーションが失敗することが保証され、安全でない仮定を防止できます。

メモリーモデル

これまでの章では、メモリーモデルを理解するのに必要な概念を紹介してきました。この章の残りの部分では、メモリーモデルについて詳しく説明します。

- カーネルのメモリー順序要件をどのように表現するか
- 特定のデバイスがサポートするメモリー順序を照会する方法

- メモリーモデルが、互いに分離したアドレス空間と複数のデバイスに関してどのように動作するか
- メモリーモデルがバリア、フェンス、アトミックとどのように相互作用するか
- バッファーと USM におけるアトミック操作の使用方法の違い

このメモリーモデルは、C++ のメモリーモデルに基づいていますが、いくつかの重要な点が異なります。この違いは、SYCL が C++ の将来に役立つべきだという、私たちの長期的なビジョンを反映しています。デフォルトの動作とクラスの命名は C++ 標準ライブラリーと密接に連携しており、C++ の機能を制限するのではなく、拡張することを目的としています。

図 19-9 の表は、C++（C++11、C++14、C++17、C++20）と SYCL で、異なるメモリーモデルの概念が言語機能としてどのように定義されているかまとめたものです。C++14、C++17、および C++20 規格には、C++ の実装に影響を与えるいくつかの明確化が追加されています。これらの説明は、ここで作成するアプリケーション・コードには影響しないはずであるため、ここでは取り上げません。

機能	C++	SYCL
アトミック操作	`std::atomic`	利用できません
アトミック参照	`std::atomic_ref` (C++20 以降)	`sycl::atomic_ref`

メモリー順序	`relaxed`
	`consume`
	`acquire`
	`release`
	`acq_rel`
	`seq_cst`
		`relaxed`
		
		`acquire`
		`release`
		`acq_rel`
		`seq_cst`

メモリースコープ	利用できません
	
	アトミック操作とフェンスの動作は、SYCL システムのスコープに一致しています。
		`work_item`
		`sub_group`
		`work_group`
		`device`
		`system`

フェンス	`std::atomic_thread_fence`	`sycl::atomic_fence`
バリア	`std::barrier` (C++20 以降)	`sycl::group_barrier`
アドレス空間	すべてのメモリーは単一の (ホスト)アドレス空間にあります	ホスト デバイス (グローバル) デバイス (ローカル) デバイス (プライベート) 共有 (USM)

図 19-9. C++ と SYCL のメモリーモデルの比較

`memory_order` 列挙クラス

メモリーモデルは、`memory_order` 列挙クラスの 5 つの値を通じて異なるメモリー順序を定義しています (注: C++ の "`consume`" は SYCL 仕様の一部ではありません)。これらの値は、フェンスやアトミック操作の引数として指定できます。
操作にメモリー順序引数を指定すると、コンパイラーは、その操作を基準とした他のすべてのメモリー操作 (任意のアドレスへの操作) のメモリー順序保証を指示できます。詳細は以下で説明します:

- `memory_order::relaxed`

  読み取り操作と書き込み操作は、操作の前後で自由に順序を変更できます。順序を保証するものではありません。

- `memory_order::acquire`

  プログラム内の操作後の読み取りと書き込みは、必ずその操作の後に実行されなければなりません（つまり、その操作の前には移動できません）。

- `memory_order::release`

  プログラム内の操作前の読み取りと書き込みは、必ずその操作の前に実行されなければなりません（つまり、その操作の後には移動できません）。また、先行する書き込み操作は、対応する取得操作（つまり、同じ変数と `memory_order::acquire` またはバリア関数を使用したアトミック操作）によって同期された他のワーク項目から可視になることが保証されます。

- `memory_order::acq_rel`

  この操作は、取得と解放の両方の役割を果たします。読み取りと書き込み操作は、操作の前後で順序を変更することはできません。また、先行する書き込みは、`memory_order::release` で前述したように可視化する必要があります。

- `memory_order::seq_cst`

  この操作は、読み取り操作、書き込み操作、または読み取り・変更・書き込み操作に応じて、取得、解放、またはその両方として機能します。このメモリー順序によるすべての操作は、逐次一貫性のある順序で実行されます。

各操作でサポートされるメモリー順序にはいくつかの制限があります。図 19-10 の表は、どの組み合わせが有効であるかまとめたものです。

機能	サポートされる memory_order 値				
	relaxed	acquire	release	acq_rel	seq_cst
load	○	○	×	×	○
store	○	×	○	×	○
exchange compare_exchange_* fetch_*	○	○	○	○	○
fence	○	○	○	○	○

図 19-10. memory_order によるアトミック操作のサポート

ロード操作は値をメモリーに書き込まないため、release セマンティクスとは互換性がありません。同様に、ストア操作はメモリーから値を読み取らないため、acquire セマンティクスとは互換性がありません。残りの読み取り・変更・書き込みのアトミック操作とフェンスは、すべてのメモリー順序付けと互換性があります。

C++ のメモリー順序

C++ のメモリーモデルには、memory_order::acquire と同様の動作をする memory_order::consume も含まれています。しかし、C++17 では定義が改訂中であるため、使用は推奨されません。そのため、SYCL 仕様への反映は今後検討されることになっています。

memory_scope 列挙クラス

C++ メモリーモデルは、アプリケーションが単一アドレス空間を持つ、単一デバイスで実行されることを前提としています。これらの仮定はどちらも SYCL++ アプリケーションにはあてはまりません: アプリケーションの異なる領域を各種デバイス（ホストデバイスと 1 つ以上のアクセラレーター・デバイス）で実行できます。

それぞれのデバイスには複数のアドレス空間（プライベート、ローカル、グローバル）があります。また、デバイスのグローバルアドレス空間は互いに分かれている場合があります（USM サポートに依存します）。

この問題を解決するために、SYCL は C++ のメモリー順序の概念を拡張し、アトミック操作の*スコープ*を含めるようにしました。これは、特定のメモリー順序の制約が適用される最小限のワーク項目セットを示します。スコープのセットは、`memory_scope` 列挙クラスによって定義されます:

- `memory_scope::work_item`
  メモリー順序付け制約は、呼び出し元のワーク項目にのみ適用されます。このスコープはイメージ処理操作にのみ有効です。ワーク項目内のその他の操作はすべて、プログラムの順序で実行されることが既に保証されているためです。

- `memory_scope::sub_group`、`memory_scope::work_group`
  メモリー順序の制約は、呼び出し元のワーク項目と同じサブグループまたはワークグループに属するワーク項目にのみ適用されます。

- `memory_scope::device`
  メモリー順序の制約は、呼び出し元のワーク項目と同じデバイス上で実行されるワーク項目にのみ適用されます。

- `memory_scope::system`
  メモリー順序の制約は、システム内のすべてのワーク項目に適用されます。

デバイスの機能による制限を除き、すべてのメモリースコープは、すべてのアトミック操作およびフェンス操作の有効な引数となります。

ただし、スコープ引数は、次の 3 つの状況のいずれかにおいて、自動的に狭いスコープに格下げされる可能性があります:

1. アトミック操作でワークグループのローカルメモリー内の値が更新される場合、`memory_scope::work_group` よりも広いスコープはすべて縮小されます (ローカルメモリーは同じワークグループ内のワーク項目からのみ見えるため)。
2. デバイスが USM をサポートしていない場合、`memory_scope::system` を指定することは常に `memory_scope::device` を指定することと同等です (バッファーは複数のデバイスから同時にアクセスできないため)。
3. アトミック操作で `memory_order::relaxed` を使用する場合、順序付けの保証はなく、メモリースコープ引数は事実上無視されます。

デバイス機能の照会

SYCL の以前のバージョンは、サポートされていたデバイスとの互換性を確保し、移植性を最大限に高めるため、SYCL は OpenCL 1.2 デバイスや、完全な C++ メモリーモデルをサポートできないハードウェア (例えば、特定の種類の組み込みデバイス) をサポートしています。SYCL は、システム内で使用可能なデバイスがサポートするメモリー順序やメモリー範囲を照会するデバイスクエリーを提供します:

- `atomic_memory_order_capabilities`
  特定のデバイス上でアトミック操作によってサポートされているすべてのメモリー順序のリストを返します。
  デバイスは、少なくとも `memory_order::relaxed` をサポートする必要があります。

- `atomic_fence_order_capabilities`
  特定のデバイス上のフェンス操作でサポートされているすべてのメモリー順序のリストを返します。
  すべてのデバイスは、少なくとも `memory_order::relaxed`、`memory_order::acquire`、`memory_order::release`、および `memory_order::acq_rel` をサポートする必要があります。フェンスの最小要件は、アトミック操作の最小要件よりも厳格であることに注意してください。なぜなら、バリアが存在する場合にメモリー順序を推論するには、こうしたフェンスが不可欠であるためです。

- `atomic_memory_scope_capabilities`、`atomic_fence_scope_capabilities`
  特定のデバイス上のアトミック操作およびフェンス操作でサポートされているすべてのメモリー順序のリストを返します。 デバイスは、少なくとも `memory_order::work_group` をサポートする必要があります。

どのメモリー順序とスコープが、いずれの機能とデバイス機能の組み合わせでサポートされているか理解するのは、最初は難しいかもしれません。実際には、以下の 2 つの開発アプローチに従うことで、こうした複雑さを回避できます:

1. **逐次一貫性とシステムフェンスを備えたアプリケーションを開発する。**

   パフォーマンス/チューニングを行う場合にのみ、メモリー順序を緩和することを検討してください。

2. **一貫性やワークグループ間の境界を緩和しながらアプリケーションを開発する。**

   正当性を確保する場合にのみ、より厳密なメモリー順序とより広いメモリー範囲を検討してください。

最初の方法は、すべてのアトミック操作とフェンスのセマンティクスが C++ のデフォルトの動作と一致することを保証します。これは最もシンプルでエラーが発生しにくい選択肢ですが、性能と移植性という点では最も劣ります。

2 番目のアプローチは、SYCL の以前のバージョンや OpenCL などの言語のデフォルトの動作により近いものです。メモリーの順序やスコープの違いをより深く理解するため、複雑にはなりますが、私たちが書く SYCL コードの大部分が、パフォーマンスの低下なしにどのデバイスでも動作することを保証します。

バリアとフェンス

本書でこれまで使用したバリアやフェンスはすべて、デフォルトの動作に依存することで、メモリーの順序やスコープの問題を無視してきました。

デフォルトでは、SYCL のすべてのグループバリアは、呼び出し元のワーク項目がアクセスできるすべてのアドレス空間に対する取得/解放フェンスとして機能し、先行する書き込みを、同じグループ内の少なくとも他のすべてのワーク項目（グループの `fence_scope` メンバー変数で定義）に可視にします。これにより、バリアの後に、ワーク項目のグループ内でのメモリーの一貫性が確保されます。これは、同期とは何かという私たちの直感（および C++ における同期関係の定義）に沿ったものです。`group_barrier` 関数に明示的な `memory_scope` 引数を渡すことで、このデフォルトの動作を上書きできます。

`atomic_fence` 関数を使用すると、さらに細粒度の制御が可能になり、ワーク項目がメモリー順序とスコープの両方を指定してフェンスを実行できるようになります。

SYCL におけるアトミック操作

SYCL は、さまざまなデータ型の多くの種類のアトミック操作をサポートしています。すべてのデバイスは、一般的な操作（ロード、ストア、算術演算子など）のアトミックバージョンをサポートすることが保証されており、ロックフリー・アルゴリズムの実装に必要なアトミックな*比較交換*操作もサポートしています。この言語は、すべての基本整数型、浮動小数点型、およびポインター型に対してこれらの演算を定義しています。すべてのデバイスは 32 ビット型の演算をサポートする必要がありますが、64 ビット型のサポートはオプションです。

`atomic` クラス

C++11 の `std::atomic` クラスは、アトミック変数を作成および操作するためのインターフェイスを提供します。`atomic` クラスのインスタンスは自身のデータを所有し、移動やコピーはできず、アトミック操作でのみ更新できます。この制約により、クラスを誤って使用することで、未定義の動作を引き起こす可能性が大幅に低減されます。残念ながら、これらの制約によってクラスを SYCL カーネルで使用することもできなくなり、ホスト上でアトミック・オブジェクトを作成してデバイスに転送することができなくなります。ホストコード内で `std::atomic` 使用することは任意ですが、デバイスカーネル内で使用しようとするとコンパイルエラーが発生します。

`atomic` クラスは SYCL 2020 では非推奨になりました

SYCL 1.2.1 仕様には、C++11 の `std::atomic` クラスをベースにした `cl::sycl::atomic` クラスが含まれていました。大まかに言えば 2 つのクラスのインターフェイスにはいくつかの違いがあり、最も顕著な違いは、SYCL 1.2.1 バージョンではデータが所有されておらず、デフォルトで緩和されたメモリー順序になっていることです。

`cl::sycl::atomic` クラスは SYCL 2020 では非推奨です。代わりに、次のセクションで説明する `atomic_ref` クラスを使用してください。

atomic_ref クラス

C++20 の std::atomic_ref クラスは、std::atomic よりも柔軟性の高い、アトミック操作の代替インターフェイスを提供します。この 2 つのクラスの最大の違いは、std::atomic_ref のインスタンスは自身のデータを所有するのではなく、既存の非アトミック変数から構築されるという点です。アトミック参照を作成することは、実質的に、その参照の存続期間中、参照先の変数へのアクセスがアトミックに行われる保証となります。これらはまさに SYCL が必要とするセマンティクスであり、ホスト上で非アトミックなデータを作成し、そのデータをデバイスに転送し、転送後にアトミックなデータとして扱うことを可能にします。したがって、SYCL カーネルで使用される atomic_ref クラスは std::atomic_ref に基づいています。

図 19-11 に示すように、SYCL バージョンのクラスには 3 つの追加のテンプレート引数が含まれているため、これに基づいていると言えます。

```
template <typename T, memory_order DefaultOrder,
          memory_scope DefaultScope,
          access::address_space AddressSpace>
class atomic_ref {
  public:
   using value_type = T;
   static constexpr size_t  required_alignment =
       /* 実装定義 */;
   static constexpr bool is_always_lock_free =
       /* 実装定義 */;
   static constexpr memory_order default_read_order =
       memory_order_traits<DefaultOrder>::read_order;
   static constexpr memory_order default_write_order =
       memory_order_traits <DefaultOrder>::write_order;
   static constexpr memory_order
       default_read_modify_write_order = DefaultOrder;
   static constexpr memory_scope default_scope =
       DefaultScope;

   explicit atomic_ref(T& obj);
   atomic_ref(const atomic_ref& ref) noexcept;
};
```

図 19-11. atomic_ref クラスのコンストラクターと静的メンバー

前述のとおり、SYCL デバイスの種類によって機能は異なります。SYCL の atomic クラスのデフォルト動作を選択することは難しい問題です。C++ の動作（memory_order::seq_cst、memory_scope::system）をデフォルトにすると、コードの実行は最も高性能なデバイスに限定されます。一方、C++ の慣例を破り、最小公倍数（memory_order::relaxed、memory_scope::work_group）をデフォルトにすると、既存の C++ コードを移行する際に予期しない動作が発生する可能性があります。SYCL が採用している設計は妥協案であり、オブジェクト型の一部として（DefaultOrder および DefaultScope テンプレート引数を使用して）望ましいデフォルト動作を定義できるようになっています。他の順序やスコープは、必要に応じて特定のアトミック操作の引数として指定できます。DefaultOrder と DefaultScope は、デフォルトの動作をオーバーライドしない、またはオーバーライドできない操作（たとえば、+= のような短縮形演算子を使用する場合）にのみ影響します。最後の（オプションの）テンプレート引数は、参照されるオブジェクトが割り当てられているアドレス空間を示します。最後のテンプレート引数が省略されると、参照される変数は任意のアドレス空間に割り当てられることに注意してください。ここでアドレス空間を指定することは任意ですが、コンパイラーに多くの情報を提供し、不要なパフォーマンス・オーバーヘッドを回避するため、（可能であれば）明示的なアドレス空間を指定することを推奨します。

アトミック参照は、参照するオブジェクトの型に応じて、さまざまな操作をサポートします。図 19-12 には、すべての型でサポートされている基本的な操作が示されており、メモリーとの間でデータをアトミックに移動させる機能を提供します。

```
void store(
    T operand, memory_order order = default_write_order,
    memory_scope scope = default_scope) const noexcept;
T operator=(
    T desired) const noexcept;     // ストアに相当

T load(memory_order order = default_read_order,
        memory_scope scope = default_scope) const noexcept;
operator T() const noexcept;     // ロードに相当

T exchange(
    T operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_weak(
    T &expected, T desired, memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_weak(
    T &expected, T desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(
    T &expected, T desired, memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(
    T &expected, T desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;
```

図 19-12. すべての型に対する atomic_ref を用いた基本操作

整数型および浮動小数点型のオブジェクトへのアトミック参照は、図 19-13 および図 19-14 に示すように、利用可能なアトミック操作のセットを拡張して算術操作を含めます。デバイスは、ハードウェアが浮動小数点アトミック演算をネイティブにサポートするかどうかに関わらず、アトミック浮動小数点型をサポートする必要があります。多くのデバイスはアトミック比較交換を使用してアトミック浮動小数点加算をエミュレートできます。

このエミュレーションは SYCL のパフォーマンスと移植性を実現する上で重要な要素であり、アルゴリズムが必要とすればどこでも自由に浮動小数点アトミック演算を使用できます。結果として得られるコードはどこでも正しく動作し、浮動小数点アトミック・ハードウェアの将来の改善による恩恵を、一切の変更なしに得られます。

```
Integral fetch_add(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_sub(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_and(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_or(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_min(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_max(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral operator++(int) const noexcept;
Integral operator--(int) const noexcept;
Integral operator++() const noexcept;
Integral operator--() const noexcept;
Integral operator+=(Integral) const noexcept;
Integral operator-=(Integral) const noexcept;
Integral operator&=(Integral) const noexcept;
Integral operator|=(Integral) const noexcept;
```

図 19-13. 整数型のみに適用される atomic_ref の加算操作

```
Floating fetch_add(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_sub(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_min(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_max(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating operator+=(Floating) const noexcept;
Floating operator-=(Floating) const noexcept;
```

図 19-14. atomic_ref を使用した加算操作は浮動小数点型のみに適用されます

バッファーでアトミックを使用

前のセクションで説明したように、SYCL ではアトミックデータを割り当てて、ホストとデバイス間で移動させる方法はありません。バッファーと組み合わせてアトミック操作を使用するには、デバイスに転送する非アトミックデータのバッファーを作成し、その後、アトミック参照を介してそのデータにアクセスします。

```
q.submit([&](handler& h) {
   accessor  acc{buf, h};
   h.parallel_for(N, [=](id<1> i) {
     int j = i % M;
     atomic_ref<int, memory_order::relaxed,
                memory_scope::system,
                access::address_space::global_space>
        atomic_acc(acc[j]);
     atomic_acc += 1;
   });
});
```

図 19-15. 明示的に作成された atomic_ref を介してバッファーにアクセス

図 19-15 のコードは、明示的に作成されたアトミック参照オブジェクトを使用して SYCL でアトミック性を表現する例です。バッファーには通常の整数が格納されるため、読み取りと書き込み権限の両方を持つアクセサーが必要です。その後、`fetch_add` メンバー関数の短縮形である `+=` 演算子を使用して、データアクセスごとに `atomic_ref` のインスタンスを作成できます。

このパターンは、同じカーネル内でバッファーへのアトミックアクセスと非アトミックアクセスを混在させる場合に有効です。これにより、アトミック操作が不要な場合に発生するパフォーマンスのオーバーヘッドを回避できます。バッファー内のメモリー位置のうち、複数のワーク項目が同時にアクセスするのは一部であることがわかっている場合、その部分へのアクセスにのみアトミック参照を使用します。あるいは、同じワークグループ内のワーク項目がカーネルの 1 つのステージ（つまり、2 つのワークグループ・バリアの間）でのみローカルメモリーに同時にアクセスする場合は、そのステージでのみアトミック参照を使用するだけで済みます。このようにアトミックアクセスと非アトミックアクセスを混在させる場合、オブジェクトの存続期間に注意しなければなりません。特定のオブジェクトを参照する `atomic_ref` が存在する限り、そのオブジェクトへのすべてのアクセスは `atomic_ref` のインスタンスを介して（アトミックに）行われる必要があります。

統合共有メモリーでアトミック操作を使用

図 19-16（図 19-7 から転載）に示すように、バッファーとまったく同じ方法で、USM に格納されたデータからアトミック参照を構築できます。このコードと図 19-15 に示すコードとの違いは、USM コードではバッファーやアクセサーが不要であるという点だけです。

```
q.parallel_for(N, [=](id<1> i) {
    int j = i % M;
    atomic_ref<int, memory_order::relaxed,
                memory_scope::system,
                access::address_space::global_space>
         atomic_data(data[j]);
    atomic_data += 1;
 }).wait();
```

図 19-16. 明示的に作成された atomic_ref を介して USM 割り当てにアクセス

アトミックを活用

アトミックの潜在的な用途は非常に幅広く多岐にわたるため、本書ですべての用途を挙げることはできません。ここでは、様々な分野で幅広く適用可能な代表的な例を 2 つ紹介します:

1. ヒストグラムの計算
2. デバイス全体の同期の実装

ヒストグラムの計算

図 19-17 のコードは、緩和されたアトミック操作とワークグループ・バリアを組み合わせてヒストグラムを計算する方法を示しています。カーネルはバリアによって 3 つのフェーズに分割され、それぞれに独自のアトミック要件があります。バリアは同期ポイントと取得・解放フェンスの両方の役割を果たすことを覚えておいてください。これにより、あるフェーズで行われた読み取りと書き込みが、後のフェーズのワークグループ内のすべてのワーク項目から見えるようになります。

最初のフェーズでは、ワークグループのローカルメモリーの一部をゼロに設定します。各ワークグループ内のワーク項目は、設計上、ワークグループのローカルメモリー内の独立した場所を更新するため、競合状態は発生せず、アトミック性も必要ありません。

第 2 フェーズでは、部分的なヒストグラムの結果をローカルメモリーに蓄積します。同じワークグループ内のワーク項目は、ワークグループのローカルメモリー内の同じ場所を更新する可能性がありますが、同期はフェーズが終了するまで延期できます。`memory_order::relaxed` と `memory_scope::work_group` を使用してアトミック性の要件を満たすことができます。

第 3 フェーズでは、部分的なヒストグラムの結果をグローバルメモリーに格納されている合計値に加算します。同じワークグループ内のワーク項目は、ワークグループ・ローカル・メモリー内の独立した場所を読み取ることが保証されますが、グローバルメモリー内の同じ場所を更新する場合があります。ワークグループ・ローカル・メモリーのアトミック性は必要なく、以前と同様に `memory_order::relaxed` と `memory_scope::system` を使用してグローバルメモリーのアトミック性の要件を満たすことができます。

```
q.submit([&](handler& h) {
   auto local = local_accessor<uint32_t, 1>{B, h};
   h.parallel_for(
       nd_range<1>{num_groups * num_items, num_items},
       [=](nd_item<1> it) {
         auto grp = it.get_group();

         // フェーズ 1: ワーク項目が連携してローカルメモリーをゼロにする
         for (int32_t b = it.get_local_id(0); b < B;
              b += it.get_local_range(0)) {
            local[b] = 0;
         }
         group_barrier(grp);     // すべてがゼロになるまで待機

         // フェーズ 2: 各ワークグループは、入力データの一部を処理。
         // ワーク項目は連携してローカルメモリー内のヒストグラムを計算
         const auto [group_start, group_end] =
             distribute_range(grp, N);
         for (int i = group_start + it.get_local_id(0);
              i < group_end; i += it.get_local_range(0)) {
            int32_t b = input[i] % B;
            atomic_ref<uint32_t, memory_order::relaxed,
                       memory_scope::work_group,
                        access::address_space::local_space>(local[b])++;
         }
         group_barrier(
             grp);     // ローカル・ヒストグラムの更新がすべて完了するまで待機

         // フェーズ 3: ワーク項目は連携してグローバルメモリーを更新
         for (int32_t b = it.get_local_id(0); b < B;
              b += it.get_local_range(0)) {
            atomic_ref<uint32_t, memory_order::relaxed, memory_scope::system,
                       access::address_space::global_space>(histogram[b]) +=
                 local[b];
         }
       });
 }).wait();
```

図 19-17. 異なるメモリー空間のアトミック参照を使用してヒストグラムを計算

デバイス全体の同期の実装

第 4 章で、ワークグループ間でワーク項目を同期するカーネルの作成に対して警告しました。しかし、この章を読んだ読者の多くは、アトミック操作をベースにデバイス全体にわたる独自の同期ルーチンを実装しようと考えており、私たちの警告は無視されるだろうと予想しています。

デバイス全体の同期機能は移植性に欠けるため、専門のプログラマーに任せるのが最善です。SYCL の今後のバージョンでこの問題に対処する予定です。

本節で説明するコードは危険であり、デバイスのハードウェア機能や SYCL の実装には潜在的な違いがあるため、すべてのデバイスで動作するとは限りません。アトミック操作によるメモリー順序の保証は、前方進行保証とは直交しており、本書の執筆時点では、SYCL におけるワークグループのスケジュールは完全に実装依存です。SYCL の ND-range 実行モデルと、ワーク項目、サブグループ、ワークグループに関連する前方進捗保証を説明するために必要な概念と用語を体系化することは、活発な学術研究分野であり、SYCL の将来のバージョンでは、この研究に基づいて、追加のスケジュール照会と制御機能が提供されることが期待されています。現時点では、これらのトピックは専門家のみが扱うべきものです。

図 19-18 は、デバイス全体に適用されるラッチ（使い捨てのバリア）の簡単な実装例を示しており、図 19-19 はその使用例です。各ワークグループは、グループがラッチへ到着したことを通知する単一のワーク項目を選択し、単純なスピンループで他のグループの到着を待機します。一方、他のワーク項目は、ワークグループ・バリアを使用して選択されたワーク項目を待機します。このスピンループこそが、デバイス全体の同期を危険なものにしている原因です。まだ実行を開始していないワークグループがあったり、現在実行中のワークグループが公平にスケジュールされていない場合、コードがデッドロックに陥る可能性があります。

同期プリミティブの実装にメモリー順序のみに依存すると、十分な前方進行保証がない場合、デッドロックが発生する可能性があります。

コードが正しく動作するには、以下の 3 つの条件を満たす必要があります:

1. 正しいフェンスが生成されることを保証するため、アトミック操作では、少なくとも示されたものと同等かそれ以上の厳格なメモリー順序を使用する必要があります。

2. ND-range の各ワークグループで選択されたリーダーは、他のワークグループのリーダーとは独立して進行状況を把握する必要があります。これは、ループ内で単一のワーク項目が停滞し、まだカウンターがインクリメントされていない他のワーク項目が滞留するのを避けるためです。

3. このデバイスは、ND-range 内のすべてのワークグループを同時に実行でき、強力な前方進行保証を備えている必要があり、これにより、ND-range 内のすべてのワークグループで選択されたリーダーが最終的にラッチに到達することが保証されます。

```
struct device_latch {
   explicit device_latch(size_t num_groups)
       : counter(0), expected(num_groups) {}

   template <int Dimensions>
   void arrive_and_wait(nd_item<Dimensions>& it) {
      auto grp = it.get_group();
      group_barrier(grp);
      // 各ワークグループから、同期化に関わるワーク項目を 1 つ選択。
      // その他のワーク項目はすべて、分岐後のバリアで待機。

       if (grp.leader()) {
          atomic_ref<size_t, memory_order::acq_rel,
                             memory_scope::device,
                             access::address_space::global_space>
              atomic_counter(counter);

          // バリアへの到達を知らせるシグナル。
          // 以前の書き込みは、デバイス上のすべてのワーク項目から
          // 見えるようにする必要があります。
          atomic_counter++;

          // すべてのワークグループが到着するまで待機。
          // デバイス上のすべてのワーク項目を以前のリリースと同期。
          while (atomic_counter.load() != expected) {
          }
       }
       group_barrier(grp);
   }

   size_t counter;
   size_t expected;
};
```

図 19-18. アトミック参照に基づいてシンプルなデバイス全体のラッチを構築

```
// Allocate a one-time-use device_latch in USM
void* ptr = sycl::malloc_shared(sizeof(device_latch), q);
device_latch* latch = new (ptr) device_latch(num_groups);
q.submit([&](handler& h) {
   h.parallel_for(R, [=](nd_item<1> it) {
     // Every work-item writes a 1 to its location
     data[it.get_global_linear_id()] = 1;

     // Every work-item waits for all writes
     latch->arrive_and_wait(it);

     // Every work-item sums the values it can see
     size_t sum = 0;
     for (int i = 0; i < num_groups * items_per_group;
          ++i) {
       sum += data[i];
     }
     sums[it.get_global_linear_id()] = sum;
   });
 }).wait();
free(ptr, q);
```

図 19-19. 図 19-18 のデバイス全体のラッチを使用

このコードは移植性が保証されているわけではありませんが、重要な 2 つの点を示すためここに掲載しました: (1) SYCL は、移植性を犠牲にする場合もありますが、デバイス固有の調整を可能にするのに十分な表現力を持っています。(2) SYCL には、より高レベルの同期ルーチンを実装する構成要素が既に含まれており、将来のバージョンの言語に含まれる可能性があります。

まとめ

この章では、メモリーモデルと `atomic` クラスの概要を説明しました。これらのクラスの使い方を理解することは、正当で、移植性があり、効率良い並列プログラムを開発する上で非常に重要です。

メモリーモデルは非常に複雑なテーマであり、ここでは実際のアプリケーションを作成する基盤を確立することに重点を置いています。さらに詳しい情報が必要な場合は、以下のメモリーモデルに関するウェブサイト、書籍、講演などを参照してください。

関連情報

- A. Williams, C++ Concurrency in Action: Practical Multithreading, Manning, 2012, 978-1933988771
- H. Sutter, “atomic<> Weapons: The C++ Memory Model and Modern Hardware”, herbsutter.com/2013/02/11/ atomic-weapons-the-c-memory-model-and-modern-hardware/
- H-J. Boehm, “Temporarily discourage memory_order_consume,” wg21.link/p0371
- C++ Reference, “std::atomic,” en.cppreference.com/w/ cpp/atomic/atomic
- C++ Reference, “std::atomic_ref,” en.cppreference.com/w/cpp/atomic/atomic_ref

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 20 章 バックエンドの相互運用性

この章では、すでに他のデータ並列技術や API を導入しているアプリケーションに、SYCL を段階的に追加できる SYCL 機能である*バックエンドの相互運用性*について説明します。

また、低レベル API に精通したプログラマーがバックエンドの相互運用性を使用して “舞台裏をのぞき見る”、SYCL プログラムから基礎となるデータ並列 API を直接使用する方法についても説明します。これにより、SYCL の移植性と使いやすさの利点を維持しながら、必要に応じて API 固有の機能に直接アクセスできるようになります。

バックエンドの相互運用性とは

これまで本書では、SYCL デバイス上で実行される SYCL プログラムについて説明してきましたが、実際には多くの SYCL 実装は、システム内の並列ハードウェアにアクセスするため、OpenCL、レベルゼロ、CUDA などの低レベル API 上に構築されています。SYCL 実装が下位レベルの API 上に構築される場合、ターゲット API を *SYCL バックエンド*と呼びます。図 20-1 は、SYCL バックエンド、プラットフォーム、およびデバイスの関係を示しています。ほとんどの SYCL 実装では、複数の SYCL バックエンドで SYCL プログラムを同時に実行して、システム内のすべての並列ハードウェアを利用できます。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_20

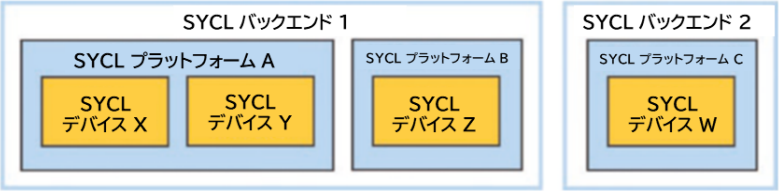

図 20-1. SYCL バックエンド、プラットフォーム、デバイスの関係

図 20-2 に示すように、最初に SYCL プラットフォームを照会し、次に各プラットフォームに関連付けられた SYCL バックエンドを照会することで、システム内の SYCL バックエンドを照会できます。このプログラムからの出力は、システム内の SYCL デバイスの数とタイプによって異なります。同じデバイスが異なる SYCL バックエンドでサポートされている場合、それぞれのバックエンドの SYCL デバイスとして列挙されることがあります。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  for (auto& p : platform::get_platforms()) {
    std::cout << "SYCL Platform: "
              << p.get_info<info::platform::name>()
              << " is associated with SYCL Backend: "
              << p.get_backend() << std::endl;
  }
  return 0;
}
```

出力例:

```
SYCL Platform: Portable Computing Language is associated with SYCL Backend:
opencl
SYCL Platform: Intel(R) OpenCL HD Graphics is associated with SYCL Backend:
opencl
SYCL Platform: Intel(R) OpenCL is associated with SYCL Backend: opencl
SYCL Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM) is associated
with SYCL Backend: opencl
SYCL Platform: Intel(R) Level-Zero is associated with SYCL Backend:
ext_oneapi_level_zero
SYCL Platform: NVIDIA CUDA BACKEND is associated with SYCL Backend:
ext_oneapi_cuda
SYCL Platform: AMD HIP BACKEND is associated with SYCL Backend:
ext_oneapi_hip
```

図 20-2. SYCL プラットフォームの SYCL バックエンドを照会

関連するバックエンドは、SYCL プラットフォームだけでなく、ほとんどの SYCL オブジェクトを照会できます。たとえば、SYCL デバイス、SYCL コンテキスト、または SYCL キューに関連するバックエンドを照会することもできます。

バックエンドの相互運用性により、関連するバックエンドの知識を使用して、バックエンドの SYCL オブジェクトを表す基礎となる*ネイティブ・バックエンド・オブジェクト*と対話し、操作できます。

バックエンドの相互運用性はいつ役立つか

多くの SYCL プログラマーは、バックエンドの相互運用性を使用する必要はありません。実際、バックエンドの相互運用性を使用することは望ましくない場合があります。バックエンドの相互運用性により、複数の SYCL バックエンドに複数のコードパスが必要になりプログラムが*複雑*になるか、単一のバックエンドを持つデバイスに実行が制限されるためプログラムの*移植性が低下*することがあります。

それでも、バックエンドの相互運用性は、特定の問題を解決するツールボックスに備えておくと便利なツールです。この章では、バックエンドの相互運用性が役立つ一般的なユースケースをいくつか紹介します。

バックエンドの相互運用性はインライン・アセンブラーのようなもの

バックエンドの相互運用性に関するメンタルモデルは、SYCL に対するバックエンドの相互運用性は、インライン・アセンブラーが C++ ホストコードに対するものであるというものです。つまり、SYCL を習得したり、SYCL で生産性を高めるにはバックエンドの相互運用性は必須ではなく、バックエンドの相互運用性は複雑さを増大させ、移植性を低下させるため、望ましくないことがあります。それでも、特定の問題を解決するためツールボックスに備えておくと便利なツールです。

既存のコードベースへの SYCL の追加

本書の SYCL プログラムは、特定の SYCL 概念を理解するため、意図的に簡単にそして短く記述されています。対照的に、現実世界のソフトウェアのほとんどは大規模かつ複雑で、数千行または数百万行ものコードで構成され、何年もかけて多くの人々によって開発されています。たとえ望んでも、大規模なアプリケーションを SYCL を使用するため完全に書き直すことは実現的ではありません。

バックエンドの相互運用性による主な利点の 1 つは、その API のネイティブ・バックエンド・オブジェクトから SYCL オブジェクトを作成することにより、すでに低レベル API を使用している既存のコードベースに SYCL を段階的に追加できることです。たとえば、OpenCL コンテキストと OpenCL メモリー・オブジェクトを作成する大規模な OpenCL アプリケーションを考えてください。バックエンドの相互運用性には、`make_context` や `make_buffer` などのテンプレート化された関数があり、これらの OpenCL オブジェクトから SYCL オブジェクトをシームレスに作成できます。
OpenCL オブジェクトから SYCL オブジェクトを作成すると、図 20-3 に示すように、他の SYCL オブジェクトと同様に SYCL キューおよび SYCL カーネルで使用できるようになります。

```
// ネイティブ・バックエンド・オブジェクトから SYCL オブジェクトを作成
context c =
     make_context<backend::opencl>(openclContext);
device d = make_device<backend::opencl>(openclDevice);
buffer data_buf =
     make_buffer<backend::opencl, int>(openclBuffer, c);

// ここで、SYCL オブジェクトを使用してキューを作成し、
// カーネルを送信します。
queue q{c, d};

q.submit([&](handler& h) {
   accessor data_acc{data_buf, h};
   h.parallel_for(size, [=](id<1> i) {
      data_acc[i] = data_acc[i] + 1;
   });
 }).wait();
```

図 20-3. OpenCL オブジェクトから SYCL オブジェクトを生成

SYCL 2020 仕様では OpenCL バックエンドとの相互運用性のみが定義されていますが、SYCL 実装では拡張機能を使用して他のバックエンドとの相互運用性が提供される場合があります。図 20-4 は、sycl_ext_oneapi_backend_level_zero 拡張機能を使用して、レベルゼロ・オブジェクトから SYCL オブジェクトを作成する方法を示しています。

```
// ネイティブ・バックエンド・オブジェクトから SYCL オブジェクトを作成
device d = make_device<backend::ext_oneapi_level_zero>(
    level0Device);
context c =
    make_context<backend::ext_oneapi_level_zero>(
       {level0Context,
       {d},
       ext::oneapi::level_zero::ownership::keep});
buffer data_buf =
     make_buffer<backend::ext_oneapi_level_zero, int>(
         {level0Ptr,
         ext::oneapi::level_zero::ownership::keep},
         c);

// ここで、SYCL オブジェクトを使用してキューを作成し、
// カーネルを送信します。
queue q{c, d};

q.submit([&](handler&  h) {
   accessor  data_acc{data_buf, h};
   h.parallel_for(size, [=](id<1> i) {
      data_acc[i] = data_acc[i] + 1;
   });
 }).wait();
```

図 20-4. レベルゼロ・オブジェクトから SYCL オブジェクトを生成

SYCL オブジェクトを作成するために渡すパラメーターは、レベルゼロ・バックエンドでは若干異なることに注意してください。これは、サポートされるバックエンドの相互運用性に一般的に当てはまります。各バックエンドが SYCL オブジェクトを適切に作成するには、異なる情報を必要とする可能性があります。それ以外の場合、OpenCL とレベルゼロ・バックエンドの相互運用性には同じ make_device、make_context、および make_buffer 関数が使用されます。

また、所有権はバックエンドごとに異なる方法で処理されることにも注意してください。OpenCL バックエンドでは、SYCL 実装は OpenCL によって提供される参照カウントを使用して、ネイティブ・バックエンド・オブジェクトの存続期間を管理します。レベルゼロ・バックエンドの場合、SYCL 実装にネイティブ・バックエンド・オブジェクトの所有権を取得するか、アプリケーションが所有権を保持するか明示的に伝える必要があります。SYCL 実装がネイティブ・バックエンド・オブジェクトの所有権を取得する場合、SYCL オブジェクトが破棄されるとネイティブ・バックエンド・オブジェクトも破棄されます。それ以外の場合、アプリケーションはネイティブ・バックエンド・オブジェクトを直接解放する責任があります。

SYCL で既存のライブラリーを使用する

バックエンドの相互運用性は、SYCL オブジェクトからネイティブ・バックエンド・オブジェクトを抽出するのにも使用できます。これは、既存の低レベル・ライブラリーやその他のヘルパー関数を SYCL アプリケーションで使用する場合に便利です。これには 2 つの方法があります。最初の方法は、`get_native` フリー関数を使用して、SYCL オブジェクトからネイティブ・バックエンド・オブジェクトを取得します。2 番目は、`host_task` と `interop_handle` を使用して、SYCL ランタイムによってスケジュールされたコードから SYCL オブジェクトからネイティブ・バックエンド・オブジェクトを取得します。

フリー関数によるバックエンド・オブジェクトの取得

たとえば、SYCL アプリケーションで使用したい最適化された OpenCL ライブラリーがあるとします。バックエンドの相互運用性 `get_native` 関数を呼び出して、SYCL オブジェクトからネイティブ OpenCL オブジェクトを取得し、OpenCL ライブラリーで使用できるようになります。簡単にするために、図 20-5 のコードはクエリーを実行し、ネイティブ OpenCL オブジェクトを使用してメモリーを割り当てるだけですが、コマンドキューの作成、プログラムのコンパイル、カーネルの実行など、複雑な操作を行うためにも使用できます。

```
cl_device_id openclDevice =
    get_native<backend::opencl>(d);
cl_context openclContext = get_native<backend::opencl>(c);

// OpenCL からデバイス名を照会:
size_t sz = 0;
clGetDeviceInfo(openclDevice, CL_DEVICE_NAME, 0, nullptr, &sz);
std::string openclDeviceName(sz, ' ');
clGetDeviceInfo(openclDevice, CL_DEVICE_NAME, sz,
                &openclDeviceName[0], nullptr);
std::cout << "Device name from OpenCL is: "
          << openclDeviceName << "\n";

// OpenCL からメモリーを割り当て:
cl_mem openclBuffer = clCreateBuffer(
    openclContext, 0, sizeof(int), nullptr, nullptr);

// 完了したら OpenCL オブジェクトをクリーンアップ:
clReleaseDevice(openclDevice);
clReleaseContext(openclContext);
clReleaseMemObject(openclBuffer);
```

図 20-5. get_native フリー関数を使用して SYCL オブジェクトから OpenCL オブジェクトを抽出

図 20-6 に示すように、同じ get_native 関数が、sycl_ext_oneapi_backend_level_zero 拡張機能の一部として、レベルゼロ・バックエンドにも追加されています。

```
ze_device_handle_t level0Device =
    get_native<backend::ext_oneapi_level_zero>(d);
ze_context_handle_t level0Context =
    get_native<backend::ext_oneapi_level_zero>(c);

   // レベルゼロからデバイス名を照会:
   ze_device_properties_t level0DeviceProps = {};
   level0DeviceProps.stype =
       ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES;

   zeDeviceGetProperties(level0Device, &level0DeviceProps);

   std::cout << "Device name from SYCL is: "
             << d.get_info<info::device::name>() << "\n";
   std::cout << "Device name from Level Zero is: "
             << level0DeviceProps.name << "\n";

// レベルゼロからメモリーを割り当て:
void* level0Ptr = nullptr;
ze_host_mem_alloc_desc_t level0HostAllocDesc = {};
level0HostAllocDesc.stype =
    ZE_STRUCTURE_TYPE_HOST_MEM_ALLOC_DESC;
zeMemAllocHost(level0Context, &level0HostAllocDesc,
               sizeof(int), 0, &level0Ptr);

// 完了したらレベルゼロのオブジェクトをクリーンアップ:
zeMemFree(level0Context, level0Ptr);
```

図 20-6. get_native フリー関数を使用して SYCL オブジェクトから レベルゼロ・オブジェクトを抽出

相互運用ハンドルを介してバックエンド・オブジェクトを取得

get_native フリー関数を使用すると、バックエンド API を直接使用するコードの大部分のセクションのバックエンド固有のオブジェクトを効果的に取得できます。ただし、多くの場合、バックエンド API を使用して SYCL タスクグラフ内で特定操作のみを実行する必要があります。このような場合、特別な interop_handle パラメーターを持つ SYCL host_task を使用して、バックエンド固有の操作を実行できます。
interop_handle は、ホストタスクが呼び出されたときの SYCL ランタイムの状態を表し、SYCL キュー、デバイス、コンテキスト、およびホストタスク用にキャプチャーされたバッファーを表すネイティブ・バックエンド・オブジェクトへのアクセスを提供します。

図 20-7 は、interop_handle を使用して、SYCL ランタイムによってスケジュールされた host_task からネイティブ OpenCL オブジェクトを取得する方法を示しています。簡単にするために、このサンプルでもネイティブ OpenCL オブジェクトを使用していくつかのクエリーのみを実行しますが、実際のアプリケーションでは通常、ネイティブ OpenCL オブジェクトを使用してカーネルをキューに追加したり、ライブラリーを呼び出したりします。これらの操作はホストタスクから実行されるため、SYCL キュー内の他の操作と共にスケジュールされます。

```
q.submit([&](handler& h) {
   accessor a{b, h};
   h.host_task([=](interop_handle ih) {
     // 相互運用ハンドルから OpenCL デバイスを取得:
     auto openclDevice =
         ih.get_native_device<backend::opencl>();

     // OpenCL デバイスからデバイス名を照会:
     size_t sz = 0;
       clGetDeviceInfo(openclDevice, CL_DEVICE_NAME, 0,
                       nullptr, &sz);
     std::string openclDeviceName(sz, ' ');
       clGetDeviceInfo(openclDevice, CL_DEVICE_NAME, sz,
                       &openclDeviceName[0], nullptr);
       std::cout << "Device name from OpenCL is: "
                 << openclDeviceName << "\n";

     // 相互運用ハンドルから OpenCL バッファーを取得:
      auto openclMem =
          ih.get_native_mem<backend::opencl>(a)[0];

     // OpenCL バッファーのサイズを照会:
     clGetMemObjectInfo(openclMem, CL_MEM_SIZE, sizeof(sz),
                        &sz, nullptr);
     std::cout << "Buffer size from OpenCL is: " << sz
               << " bytes\n";
   });
});
```

図 20-7. Interop_handle を使用して SYCL オブジェクトから OpenCL オブジェクトを抽出

アクセサーのネイティブ OpenCL オブジェクトを取得するときに、interop_handle の get_native_mem メンバー関数が cl_mem メモリー・オブジェクトのベクトルを返すことに注意してください。

これは SYCL 2020 仕様の要件であり、interop_ ハンドルのメンバー関数の戻り値の型は get_native フリー関数と一致する必要がありますが、interop_ ハンドルではベクトルの最初の要素を使用できます。

get_native フリー関数と同様に、拡張機能を介して他の SYCL バックエンドにも同様の機能を提供できます。図 20-8 は、sycl_ext_oneapi_backend_level_zero 拡張機能を使用して、レベルゼロ・バックエンドで同様の操作を実行する方法を示しています。

```
q.submit([&](handler& h) {
   accessor a{b, h};
   h.host_task([=](interop_handle ih) {
     // 相互運用ハンドルからレベルゼロデバイスを取得:
     auto level0Device = ih.get_native_device<
          backend::ext_oneapi_level_zero>();

     // レベルゼロからデバイス名を照会:
     ze_device_properties_t level0DeviceProps = {};
     level0DeviceProps.stype =
         ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES;
     zeDeviceGetProperties(level0Device,
                           &level0DeviceProps);
     std::cout << "Device name from Level Zero is: "
               << level0DeviceProps.name << "\n";

     // 相互運用ハンドルからレベル 0 のコンテキストと
     // メモリー割り当てを取得:
     auto level0Context = ih.get_native_context<
          backend::ext_oneapi_level_zero>();
     auto ptr =
          ih.get_native_mem<backend::ext_oneapi_level_zero>(
              a);

     // メモリー割り当てのサイズを照会:
     size_t  sz = 0;
     zeMemGetAddressRange(level0Context, ptr, nullptr,
                          &sz);
     std::cout << "Buffer size from Level Zero is: "
               << sz << " bytes\n";
   });
});
```

図 20-8. Interop_handle を使用して SYCL オブジェクトから OpenCL オブジェクトを抽出

カーネルのバックエンド相互運用性の使用

このセクションでは、バックエンド相互運用性を使用してカーネルをコンパイルし、カーネル バンドルを操作する方法について説明します。これは、堅牢性を高め、さまざまな SYCL バックエンドのサポートに必要な柔軟性を追加するため、SYCL 2020 で大幅に再設計された領域です。

SYCL の以前のバージョンでは、カーネルの相互運用性メカニズムが 2 つサポートされていました。最初のメカニズムにより、API 定義のハンドルからカーネルを作成できるようになりました。2 番目は、OpenCL C ソースや SPIR-V 中間表現など、API 定義のソースまたは中間表現からカーネルを作成できるようにしました。これら 2 つのメカニズムは SYCL 2020 でも存在しますが、両方のメカニズムの構文が更新され、現在はバックエンドの相互運用性が使用されています。

API 定義カーネル・オブジェクトとの相互運用性

この形式の相互運用性では、カーネル・オブジェクト自体が低レベル API を使用して作成され、バックエンドの相互運用性を使用して SYCL にインポートされます。図 20-9 のコードは、SYCL コンテキストから OpenCL コンテキストを取得する方法、この OpenCL コンテキストを使用して OpenCL カーネルを作成する方法、そして OpenCL カーネル・オブジェクトから SYCL カーネルを作成して使用する方法を示しています。

```
// SYCL コンテキストからネイティブ OpenCL コンテキストを取得:
auto openclContext = get_native<backend::opencl>(c);
const char* kernelSource =
    R"CLC(
        kernel void add(global int* data) {
            int index = get_global_id(0);
            data[index] = data[index] + 1;
        }
    )CLC";
// このコンテキストを使用して OpenCL カーネルを作成:
cl_program p = clCreateProgramWithSource(
    openclContext, 1, &kernelSource, nullptr, nullptr);
clBuildProgram(p, 0, nullptr, nullptr, nullptr,
               nullptr);
cl_kernel k = clCreateKernel(p, "add", nullptr);

// OpenCL カーネルから SYCL カーネルを作成:
auto sk = make_kernel<backend::opencl>(k, c);

// SYCL キューで OpenCL カーネルを使用:
q.submit([&](handler& h) {
   accessor data_acc{data_buf, h};

   h.set_args(data_acc);
   h.parallel_for(size, sk);
});

// 完了したら OpenCL オブジェクトをクリーンアップ:
clReleaseContext(openclContext);
clReleaseProgram(p);
clReleaseKernel(k);
```

図 20-9. OpenCL カーネル・オブジェクトから作成されたカーネル

SYCL コンパイラーは、低レベル API を直接使用して作成された SYCL カーネルを認識できないため、カーネル引数は `set_arg()` または `set_args()` インターフェイスを使用して明示的に渡す必要があります。さらに、SYCL ランタイムと低レベル API カーネルは、オブジェクトをカーネル引数として渡す規則に従う必要があります。この規則は、バックエンドの相互運用性仕様の一部として記述する必要があります。この例では、アクセサー `data_acc` がグローバル・ポインター・カーネル引数データとして渡されます。

SYCL 2020 標準では、`set_arg()` および `set_args()` インターフェイスの正確なセマンティクスは、各 SYCL バックエンド仕様によって定義されます。これにより柔軟性が得られますが、バックエンドの相互運用性を使用して記述するコードは、対象とするバックエンドに固有となる可能性が高くなります。

SYCL 以外のソース言語との相互運用性

この形式の相互運用性では、カーネルの内容はソースコードとして、または SYCL で定義されていない中間表現として記述されます。この形式の相互運用性により、他のソース言語で記述されたカーネル・ライブラリーを再利用したり、中間表現でコードを生成するドメイン固有言語 (DSL) を使用できます。

SYCL の以前のバージョンには、API 定義のソース言語から SYCL プログラムを直接作成する `build_with_source` などの関数が含まれていましたが、この機能は SYCL 2020 で削除されました。図 20-9 の OpenCL バックエンドで使用される OpenCL C カーネルなど、バックエンドが API 定義のソース言語を直接サポートしている場合、削除されたことは問題ありませんが、バックエンドが特定のソース言語を直接サポートしていない場合どうすればよいでしょうか。

一部の SYCL 実装では、バックエンドで直接使用できないソース言語から、バックエンドでサポートされている形式にコンパイルする明示的な*オンライン・コンパイラー*が提供される場合があります。図 20-10 は、実験的な `sycl_ext_intel_online_compiler` 拡張機能を使用して、レベルゼロ・バックエンドでサポートされていない OpenCL C ソースから、レベルゼロ・バックエンドでサポートされる SPIR-V 中間表現にコンパイルする方法を示しています。この方法を使用すると、オンライン・コンパイラーによってバックエンドでサポートされている形式にコンパイルできる限り、カーネルはどのバックエンドでも使用できます。

注: これは実験的拡張です

`sycl_ext_intel_online_compiler` 拡張機能は実験的な拡張機能であるため、変更または削除される可能性があります。本書にこれを含めたのは、以前の SYCL の `build_with_source` 関数と同様の機能を実現できる方法を提供し、ドメイン固有言語が SYCL バックエンドとインターフェイスしてカーネルを実行する方法を示す便利な方法であるためです。

```
// オンライン・コンパイラーを使用して OpenCL C カーネルソース
// を SPIR-V 中間表現にコンパイル:
const char* kernelSource =
   R"CLC(
      kernel void add(global int* data) {
         int index = get_global_id(0);
         data[index] = data[index] + 1;
      }
   )CLC";
online_compiler<source_language::opencl_c> compiler(d);
std::vector<byte> spirv =
    compiler.compile(kernelSource);

// SYCL キューでレベルゼロカーネルを使用:
auto level0Context =
    get_native<backend::ext_oneapi_level_zero>(c);
auto level0Device =
    get_native<backend::ext_oneapi_level_zero>(d);

// このコンテキストを使用してレベルゼロカーネルを作成:
ze_module_handle_t level0Module = nullptr;
ze_module_desc_t moduleDesc = {};
moduleDesc.stype = ZE_STRUCTURE_TYPE_MODULE_DESC;
moduleDesc.format = ZE_MODULE_FORMAT_IL_SPIRV;
moduleDesc.inputSize = spirv.size();
moduleDesc.pInputModule = spirv.data();
zeModuleCreate(level0Context, level0Device, &moduleDesc,
               &level0Module, nullptr);

ze_kernel_handle_t level0Kernel = nullptr;
ze_kernel_desc_t kernelDesc = {};
kernelDesc.stype = ZE_STRUCTURE_TYPE_KERNEL_DESC;
kernelDesc.pKernelName = "add";
zeKernelCreate(level0Module, &kernelDesc,
               &level0Kernel);

// レベルゼロカーネルから SYCL カーネルを作成:
auto skb =
    make_kernel_bundle<backend::ext_oneapi_level_zero,
                       bundle_state::executable>(
         {level0Module}, c);
auto sk = make_kernel<backend::ext_oneapi_level_zero>(
    {skb, level0Kernel}, c);

// SYCL キューでレベルゼロカーネルを使用:
q.submit([&](handler& h) {
   accessor data_acc{data_buf, h};

   h.set_args(data_acc);
   h.parallel_for(size, sk);
});
```

図 20-10. SPIR-V とオンライン・コンパイラーを使用して作成されたカーネル

この例では、カーネルソース文字列は、SYCL ホスト API 呼び出しと同じファイル内の C++ の文字列リテラルとして表現されていますが、必ずしもそうである必要はなく、一部のアプリケーションではファイルからカーネルソース文字列を読み取ったり、ジャストインタイムで生成することもできます。

以前と同様に、SYCL コンパイラーは API 定義のソース言語で記述された SYCL カーネルを認識できないため、カーネル引数は `set_arg()` または `set_args()` インターフェイスを使用して明示的に渡す必要があります。

バックエンドの相互運用性に関するヒントとコツ

ここでは、バックエンドの相互運用性を効果的に活用する実用的なヒントとコツについて説明します。

特定のバックエンド用のデバイスの選択

バックエンドの相互運用性を適切に使用する最初の要件は、必要な SYCL バックエンドに関連付けられた SYCL デバイスを選択することです。これには色々な方法があります。

1 つは、各デバイスのスコア付け時に関連付けられたバックエンドを照会することにより、必要な SYCL バックエンドを既存のカスタムデバイス選択のロジックに統合することです。アプリケーションですでにカスタムデバイス選択ロジックを使用している場合、これを簡単に追加できるはずです。このメカニズムは標準の SYCL クエリーのみを使用するため、移植性も優れています。

カスタムデバイス選択ロジックをまだ使用していないアプリケーションでは、図 20-11 に示すように、C++ ラムダ式を記述してすべてのデバイスを反復処理し、要求されたバックエンドを持つデバイスを検出できます。
このバージョンの `find_device` は特定のデバイスタイプを要求しないため、実質的には標準の `default_selector_v` の代わりとなります。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  auto find_device = [](backend b,
                        info::device_type t =
                             info::device_type::all) {
    for (auto d : device::get_devices(t)) {
      if (d.get_backend() == b) {
         return d;
      }
    }
    throw sycl::exception(errc::runtime,
                          "Could not find a device with「
                          "the requested backend!");
  };

  try {
    device d{find_device(backend::opencl)};
    std::cout << "Found an OpenCL SYCL device: "
              << d.get_info<info::device::name>() << "\n";
  } catch (const  sycl::exception &e) {
    std::cout << "No OpenCL SYCL devices were found.\n";
  }

  try {
     device d{find_device(backend::ext_oneapi_level_zero)};
     std::cout << "Found a Level Zero SYCL device: "
               << d.get_info<info::device::name>() << "\n";
  } catch (const  sycl::exception &e) {
    std::cout << "レベルゼロ SYCL devices were found.\n";
  }

  return 0;
}
```

出力例:

```
Found an OpenCL SYCL device: pthread-12th Gen Intel(R) Core(TM) i9-12900K
Found a Level Zero SYCL device: Intel(R) UHD Graphics 770 [0x4680]
```

図 20-11. 特定のバックエンドを持つ SYCL デバイスの検索

最後に、高速なプロトタイプ作成のため、一部の SYCL 実装では、環境変数などの外部メカニズムを使用して、列挙する SYCL デバイスに影響を及ぼすことができます。たとえば、DPC++ の SYCL ランタイムは `ONEAPI_DEVICE_SELECTOR` 環境変数を使用して、列挙されたデバイスを特定のデバイスタイプ、または関連するデバイス・バックエンドに限定できます（第 13 章を参照）。

これは外部構成を必要とするため、製品版コードには理想的ではありませんが、プロトタイプ・コードでは、アプリケーションが特定のバックエンドから特定のデバイスを使用していることを確認する便利なメカニズムです。

コンテキストに注意してください

第 6 章と第 13 章で説明したように、カーネルや USM 割り当てなどの多くの SYCL オブジェクトは、通常、別の SYCL コンテキストで作成された SYCL コンテキストからはアクセスできません。これは、バックエンドの相互運用性を使用する場合にも当てはまります。したがって、バックエンド API を使用して作成されたバックエンド固有のコンテキストは、SYCL コンテキストが同じバックエンドに関連付けられている場合でも、異なる SYCL コンテキストで作成されたオブジェクトにアクセスできません (その逆も同様)。

SYCL とバックエンド間でオブジェクトを安全に共有するには、常に `make_context` を使用してネイティブ・バックエンド・コンテキストから SYCL コンテキストを作成するか、`get_native` を使用して SYCL コンテキストからネイティブ・バックエンド・コンテキストを取得する必要があります。

SYCL とバックエンド間でオブジェクトを安全に共有するには、常にネイティブ・バックエンド・コンテキストから SYCL コンテキストを作成するか、SYCL コンテキストからネイティブ ・バックエンド・コンテキストを取得してください。

低レベル API 固有の機能にアクセスする

場合によっては、最先端の機能が SYCL で利用できるようになる前に、低レベル API で利用できることがあります (SYCL 拡張機能であっても)。一部の機能はバックエンド固有またはデバイス固有であるため、SYCL を通じて公開されることはありません。たとえば、一部のネイティブ・バックエンド API は、特定のプロパティーを持つキューや、特定のアクセラレーター・ハードウェア固有のカーネル命令へのアクセスを提供する場合があります。このようなケースは稀ですが、このような種類の機能が存在する場合は、バックエンドの相互運用性を使用してアクセスできる可能性があります。

他のバックエンドのサポート

この章の例では、OpenCL およびレベルゼロ・バックエンドとのバックエンドの相互運用性を示しましたが、SYCL は成長を続けるエコシステムであり、SYCL 実装では追加のバックエンドとデバイスのサポートが定期的に追加されています。たとえば、CUDA および HIP バックエンドをサポートする SYCL 実装では、すでにこれらのバックエンドとの相互運用性がサポートされています。
SYCL 実装のドキュメントをチェックして、どの SYCL バックエンドがサポートされているか、およびバックエンドの相互運用性をサポートしているか確認してください。

まとめ

この章では、各 SYCL オブジェクトが基盤となる SYCL バックエンドとどのように関連しているか、またシステム内の SYCL バックエンドを照会する方法について説明しました。バックエンドの相互運用性によって、SYCL アプリケーションが基盤となるバックエンド API と直接対話するメカニズムがどのように提供されるか説明しました。これにより、バックエンド API を直接使用しているアプリケーションに SYCL を段階的に追加したり、バックエンド API 専用に作成されたライブラリーやユーティリティー関数を再利用する方法を説明しました。また、バックエンドの相互運用性によって、アプリケーションが実行される SYCL デバイスが制限され、アプリケーションの移植性が低下することについても説明しました。

特に、カーネルのバックエンド相互運用性が、SYCL の以前のバージョンと同様の機能を SYCL 2020 でどのように提供するかを調査しました。一部の SYCL バックエンドでは直接理解されない場合でも、オンライン・コンパイラー拡張機能によってカーネルの一部のソース言語を使用する方法を調べました。

最後に、特定の SYCL バックエンドに SYCL デバイスを選択する方法、バックエンドの相互運用性のために SYCL コンテキストを設定する方法、バックエンドの相互運用性が SYCL に追加されていない場合でも機能へのアクセスを提供する方法など、プログラムでバックエンドの相互運用性を効果的に使用する実用的なヒントとコツを確認しました。

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。
（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

第 21 章 CUDA コードの移行

本書の読者の多くは、CUDA で書かれたデータ並列コードを目にしたことがあるでしょう。読者の中には CUDA のエキスパートもいるかもしれません。この章では、CUDA と SYCL の類似点と相違点、そして CUDA コードを SYCL を使用した C++ に効果的かつ効率良く移行する便利なツールとテクニックについて説明します。

CUDA と SYCL の設計上の違い

詳細に入る前に、まず CUDA と SYCL の主な設計上の違いを知ることが重要です。これにより、いくつかの違いが存在する理由を理解し、いずれかがが時間の経過とともに消える可能性があり、また残る可能性があるかを理解する有用な背景情報が得られます。

複数のターゲットと単一のデバイスターゲット

CUDA と SYCL の設計上の最大の違いの 1 つは、サポートするデバイスの範囲です。CUDA は単一のデバイスベンダーの GPU デバイスをサポートするように設計されているため、ほとんどの CUDA デバイスは比較的類似しています。たとえば、現在、すべての CUDA デバイスにはテクスチャー・サンプリング・ハードウェアが含まれており、すべての CUDA デバイスは同じ最大ワークグループ・サイズをサポートします。これにより複雑さは少なくなりますが、CUDA アプリケーションが実行される場所も限定されます。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2_21

対照的に、SYCL は、さまざまなデバイスベンダーのさまざまなデバイスを含む、多様な異種アクセラレーター・セットをサポートするように設計されています。この柔軟性により、SYCL プログラムは最新の異種システムの計算リソースを自由に活用できるようになります。ただし、この柔軟性には多少のコストが発生します。たとえば、SYCL プログラマーは、システム内のデバイスを列挙し、そのプロパティーを調べて、プログラムのさまざまな部分を実行するのに最適なデバイスを選択する場合があります。

もちろん、SYCL プログラムがシステム内のすべての計算リソースを利用する予定がない場合は、標準デバイスセレクターなど、コードの冗長性を排除するさまざまなショートカットがあります。図 21-1 は、SYCL 実装によって選択されたデフォルトデバイスのキューを使用する基本的な SYCL サンプルを示しています。

```
// デフォルトデバイスのインオーダー SYCL キューを宣言
queue q{property::queue::in_order()};
std::cout << "Running on device: "
          << q.get_device().get_info<info::device::name>()
          << "\n";

int* buffer  = malloc_host<int>(count, q);
q.fill(buffer, 0, count);

q.parallel_for(count, [=](auto id) {
   buffer[id] = id;
 }).wait();
```

図 21-1. デフォルトの SYCL デバイス上でカーネルを実行

この SYCL コードは、図 21-2 に示す同等の CUDA コードと非常によく似ています。

```
// CUDA カーネルは別の関数
__global__ void  TestKernel(int* dst) {
   auto id = blockIdx.x  * blockDim.x  + threadIdx.x;
   dst[id] = id;
}

int main() {
   // CUDA はデフォルトでデバイス 0 を使用
   cudaDeviceProp deviceProp;
   cudaGetDeviceProperties(&deviceProp, 0);
   std::cout << "Running on device: " << deviceProp.name << "\n";

   int *buffer = nullptr;
   cudaMallocHost(&buffer, count * sizeof(int));
   cudaMemset(buffer, 0, count * sizeof(int));

   TestKernel<<<count / 256, 256>>>(buffer);
   cudaDeviceSynchronize();
   // ...
```

図 21-2. デフォルトの CUDA デバイス上でカーネルを実行

通常、実際の SYCL コードはさらに複雑です。たとえば、多くの SYCL アプリケーションは、特定のデバイス特性（第 12 章を参照）を検索することによって、実行する特定のデバイスまたはデバイスの組み合わせを列挙して選択します（第 2 章を参照）。ただし、この複雑性を排除する簡潔なオプションが存在し、SYCL は必要な場合に複雑な処理を追加できるように設計されています。

C++ への準拠と C++ の拡張

CUDA と SYCL のもう 1 つの重要な設計上の違いは、他のプログラミング言語、特に C++ との関連性です。SYCL コードは、言語拡張のない標準 C++ コードです。C++ コードの読み方、理解方法、書き方を学ぶことで、SYCL コードを読み、理解できるようになります。同様に、コンパイラーが C++ コードを解析できる場合は、SYCL コードも解析できます。

CUDA は異なる方法を採用しました。CUDA はカーネルを実行する新しいキーワードと特別な構文を追加して C++ を*拡張*しています。言語拡張は簡潔になる場合もありますが、学習して覚える構文が 1 つ増えることになります。また、言語拡張は、CUDA コードを CUDA 対応のコンパイラーでのみコンパイルできることを意味します。

この設計の違いを実際に確認するには、図 21-1 の SYCL の例で、標準 C++ ラムダ式を使用してカーネルコードを記述し、標準 C++ 関数呼び出しでカーネルを実行のため送信する方法に注目してください。図 21-2 の CUDA の例では、代わりに `__global__` キーワードを使用してカーネルコードを識別し、`<<< >>>` 構文でカーネルを実行に送信します。これは、言語拡張で実現されます。

CUDA と SYCL の用語の違い

SYCL と CUDA の主な設計上の違いを説明したので、具体的な類似点と相違点を理解する準備がほぼ整いました。
ただし、最初に確認しておくべき背景情報がもう 1 つあります。CUDA と SYCL では類似の概念に対して異なる用語が使用されることが多いため、図 21-3 のように、2 つの API を理解する必要があります。

コンセプト	SYCL 用語	CUDA 用語
デバイス上で並列に実行される関数。	カーネル	カーネル
N 次元の並列インデックス空間。	レンジ（通常）、または ND-Range（グループ化あり）	グリッド（常にグループ化）
並列インデックス空間内の特定の時点で実行されるカーネル・インスタンス。	ワーク項目	スレッド
通信および同期が可能な並列インデックス空間内のカーネル・インスタンスのアプリケーション定義のグループ。	ワークグループ	ブロック
追加の通信および同期機能を備えた、アプリケーション定義のカーネル・インスタンスのグループ。	サブグループ	ワープ
グループ内のインスタンス間でデータ交換に使用されるメモリー。	ローカルメモリー	共有メモリー
グループ内のインスタンスの同期に使用される関数。	`group_barrier()`	`__syncthreads()`、`__syncwarp()`、`coop_group.sync()`
デバイス上でカーネルやその他の作業を実行するために使用されるオブジェクト。	キュー	ストリーム

図 21-3. CUDA と SYCL の用語

本書では一貫して SYCL 用語が使用されていますが、この章では CUDA 用語と SYCL 用語が同じ意味で使用される場合があります。

類似点と相違点

ここでは、SYCL と CUDA の構文と動作の類似点、および SYCL と CUDA の異なる点について説明します。

実行モデル

基本的に、SYCL と CUDA はどちらも、第 4 章で紹介され、本書全体で説明されている同じデータ並列カーネル実行モデルを使用します。用語は若干異なる場合があります。たとえば、SYCL は ND-range を指し、CUDA はグリッドを指しますが、図 21-3 を参照して、主要な概念を SYCL から CUDA へ、またはその逆に変換できます。

インオーダー・キューとアウトオブオーダー・キュー

実行モデルには多くの類似点がありますが、いくつかの違いもあります。1 つの違いは、CUDA ストリームは無条件に*インオーダー*であることです。つまり、CUDA ストリームに送信されたカーネルまたはメモリー操作は、次に送信するカーネルまたはメモリーコピー操作を開始する前に完了する必要があります。SYCL キューはデフォルトでは*アウトオブオーダー*ですが、SYCL キューの作成時に `in_order` キュー・プロパティーを渡すことで、オプションでインオーダーにできます (第 8 章を参照)。

インオーダー CUDA ストリームは、明示的なスケジュールや依存関係の管理を必要としないためシンプルです。このシンプルさは、CUDA アプリケーションでは通常、ストリーム内の操作を順序付ける `accessors` や `depend_on` などのメカニズムを使用しないことを意味します。ただし、インオーダー・セマンティクスは実行を制約し、単一ストリーム内で 2 つのコマンドを重複して実行することはできません。CUDA アプリケーションは、単一のストリーム内で 2 つのコマンドの実行を重複できないため、CUDA アプリケーションが (潜在的に) コマンドを同時に実行する場合、コマンドを異なる CUDA ストリームに送信して実行します。

複数のインオーダー・キューに送信してカーネルまたはメモリー操作を同時に実行するパターンは SYCL でも機能し、多くの SYCL 実装と SYCL デバイスはこれを処理するように最適化されています。

ただし、アウトオブオーダー SYCL キューは、単一のキューで実行をオーバーラップする代替メカニズムを提供し、多くの SYCL 実装と SYCL デバイスはこのケースも処理できるように最適化されています。

最終的に、複数のインオーダー SYCL キューを使用するか、アウトオブオーダー SYCL キューにするかは好みとプログラミング・スタイルの問題ですが、SYCL プログラムに最も適したオプションを選択できます。
この章の SYCL の例では、同等の CUDA の例に可能な限り近いように、インオーダー SYCL キューを使用しています。

連続次元

初心者と熟練した CUDA プログラマーの両方を混乱させるもう 1 つの違いは、多次元 SYCL 範囲または CUDA グリッドに関するものです: SYCL は標準 C++ の多次元配列と規則を合わせるため、右側の次元は*連続した*次元（*ユニットストライド*次元または*最速移動*次元とも呼ばれる）になります。
代わりに CUDA はグラフィックス規則に準拠するため、左側の次元が連続した次元になります（Fortran に似ています）。この違いにより、多次元 SYCL 範囲は同等の CUDA コードと比較して転置されているように見え、SYCL ID の高位次元は、低位次元ではなく、同等の CUDA 組み込み変数の `x` コンポーネントに対応します。

この違いを、図 21-4 の CUDA の例で考えてみましょう。この例では、各 CUDA スレッドは `threadIdx.x` の値を隣接スレッドと交換します。`x` コンポーネントは CUDA で最も速く移動するコンポーネントであるため、CUDA スレッドの値が隣接するスレッドの値と一致することは想定されていません。

```
__global__ void  ExchangeKernel(int* dst) {
   auto index =  get_global_linear_id();  // ヘルパー関数
   auto fastest = threadIdx.x;
   auto neighbor = __shfl_xor_sync(0xFFFFFFFF, fastest, 1);
   dst[index] = neighbor;
}
...
dim3 threadsPerBlock(16, 2);
ExchangeKernel<<<1, threadsPerBlock>>>(buffer);
cudaDeviceSynchronize();
```

図 21-4. x 成分は CUDA における連続次元

同等の SYCL の例を図 21-5 に示します。SYCL の例では、ND-range が CUDA の例では `(16, 2)` ではなく `{2, 16}` であるため、並列インデックス空間が転置されているように見えることに注意してください。また、ND-range をサイズ `{2, 16}` のワークグループに分割された `{2, 16}` のグローバル範囲として説明していますが、CUDA の例では、ブロックごとに `(16, 2)` CUDA スレッドを持つ 1 つのブロックのグリッドとして説明しています。

さらに、右側の次元が SYCL 内で最も速く移動するコンポーネントであるため、各 SYCL ワーク項目は、`item.get_local_id(1)` (`item.get_local_id(0)` ではありません) の値を隣接するものと交換することにも注意してください。この SYCL の例では、SYCL ワーク項目の値が隣接するワーク項目の値と一致することも想定されていません。

```
q.parallel_for(nd_range<2>{{2, 16}, {2, 16}},
                [=](auto  item) {
                  auto index = item.get_global_linear_id();
                  auto fastest = item.get_local_id(1);
                  auto sg = item.get_sub_group();
                  auto neighbor =
                       permute_group_by_xor(sg, fastest, 1);
                  buffer[index] = neighbor;
                 })
    .wait();
```

図 21-5. 右側の次元は SYCL の連続した次元です

サブグループ・サイズ（ワープサイズ）

これらの例を注意深く見ると、特に近隣とのデータを交換に使用される機能に関して、さらにいくつかの違いが見つかります。

CUDA の例では、関数 `__shfl_xor_sync(0xFFFFFFFF, fastest, 1)` を使用して、隣接ノードとデータを交換します。この関数の最初の引数 `0xFFFFFFFF` は、呼び出しに参加している CUDA スレッドのセットを示すビットフィールド・マスクです。CUDA デバイスでは、ワープサイズは現在すべての CUDA デバイスで 32 であるため、32 ビットマスクで十分です。

SYCL の例では、関数 `permute_group_by_xor(sg, fastest, 1)` を使用して、隣接グループとデータを交換します。この関数では、呼び出しに参加するワーク項目のセットを記述します。この場合、`sg` はサブグループ全体を表します。ワーク項目のセットはビットフィールド・マスクではなく、グループ・オブジェクトによって指定されるため、任意のサイズのセットを表現できます。一部の SYCL デバイスではサブグループのサイズが 32 未満、または 32 を超える場合があるため、この柔軟性が必要になります。

この特定のケースでは、CUDA の例を書き直して、従来の `__shfl_xor_sync` 構文ではなく、新しい CUDA *協調グループ*構文を使用できます。CUDA 協調グループに等価なものを図 21-6 に示します。このバージョンは SYCL カーネルに似ており、CUDA と SYCL 2020 バージョンがさらに近づいていることを示す良い例です。

```
__global__ void ExchangeKernelCoopGroups(int* dst) {
  namespace cg = cooperative_groups;
  auto index = cg::this_grid().thread_rank();
  auto fastest = threadIdx.x;
  auto warp = cg::tiled_partition<32>(cg::this_thread_block());
  auto neighbor = warp.shfl_xor(fastest, 1);
  dst[index] = neighbor;
}
```

図 21-6. CUDA 協調グループのデータ交換

進行保証

図 21-4 と 21-5 の例をよく見ると、もう 1 つの違いが見つかります。ただし、この違いはより微妙です。繰り返しますが、この違いは、隣接関数とデータを交換するために使用される `__shfl_xor_sync` 関数に関連しており、この場合、違いは関数の `_sync` サフィックスによって区別されます。`_sync` サフィックスは、この関数が CUDA スレッドを*同期*していることを示しますが、この関数を呼び出す前に CUDA スレッドがなぜ同期されていないのかという疑問が生じます。

第 15 章と第 16 章では、CPU または GPU 上で実行されるデータ並列カーネルのメンタルモデルを開発しました。このカーネルでは、SIMD 命令を使用して、一連のワーク項目がロックステップで同時に処理されます。これは多くのベンダーの CPU や GPU にとって便利なメンタルモデルですが、SYCL や CUDA を使用してデータ並列カーネルを実行する唯一の方法ではありません。このメンタルモデルが機能しなくなるケースの 1 つは、*独立スレッド・スケジュール*と呼ばれる機能をサポートする新しい CUDA デバイスを使用する場合です。

独立スレッド・スケジュールを備えた CUDA デバイスでは、個々の CUDA スレッドはグループとしてではなく独立して進行します。これらの追加の前方進行保証により、強力な前方進行保証がなければ SYCL デバイスでは正しく実行されないコードパターンを CUDA デバイスで安全に実行できるようになります。`__shfl_xor_sync` 関数の `_sync` サフィックスは、関数が同期をすることを明確に示し、32 ビットマスクを使用して同期する CUDA スレッドを指定するため CUDA に追加されました。

前方進行保証は SYCL コミュニティーで活発に議論されているトピックであり、SYCL の将来のバージョンでは、デバイスの前方進行能力を決定するクエリーと、カーネルの前方進行要件を指定するプロパティーが追加される可能性が高いと思われます。ただし、現時点では、CUDA から移植された構文的には正しい SYCL プログラムが、独立スレッド・スケジュールの影響で、すべての SYCL デバイスで正しく実行されない可能性があることに注意してください。

バリア

最後に、知っておくべき微妙な実行モデルの違いは、CUDA の `__syncthreads` 関数と SYCL の `group_barrier` 関数との比較です。CUDA の `__syncthreads` 関数はスレッドブロック内の*終了していない*すべての CUDA スレッドを同期しますが、SYCL の `group_barrier` 関数はワークグループ内の*すべての*ワーク項目を同期します。つまり、`__syncthreads` を呼び出す前に一部の CUDA スレッドが早期に終了した場合、CUDA カーネルは正常に実行されますが、図 21-7 に示すような SYCL カーネルが正常に実行される保証はありません。

```
std::cout << "WARNING: May deadlock on some devices!\n";
q.parallel_for(nd_range<1>{64, 64}, [=](auto item) {
   int id  = item.get_global_id(0);
   if (id  >= count) {
      return;     // 早期 exit
   }
   Group_barrier(item.get_group());(item.get_group());
   buffer[id] = id;
 }).wait();
```

図 21-7. SYCL バリア・デッドロックの可能性

この修正は簡単です。範囲チェックを `group_barrier` の後に移動するか、この場合は、`group_barrier` を完全に削除できます。ただし、これは常に当てはまるわけではなく、他のカーネルでは、すべてのワーク項目が常に `group_barrier` に到達するか、または常に `group_barrier` をスキップするように再構築することが必要である場合があります。

メモリーモデル

基本的に、CUDA と SYCL はどちらも弱い順序のメモリーモデルを使用します。幸いなことに、CUDA カーネルを SYCL に移行するときに注意する必要があるメモリーモデルの違いはわずかです。

バリア

デフォルトでは、CUDA `__syncthreads` バリア関数と SYCL `group_barrier` バリア関数は、SYCL `group_barrier` に渡されるグループがワークグループであると仮定すると、メモリーモデルに同じ効果をもたらします。同様に、CUDA `__syncwarp` バリア関数は、SYCL `group_barrier` に渡されるグループがサブグループであると仮定すると、SYCL `group_barrier` バリア関数と同じ効果を持ちます。

SYCL `group_barrier` は、バリアに対する `fence_scope` を指定するオプションのパラメーターを受け入れますが、ほとんどの場合これは省略できます。`group_barrier` には、`memory_scope::device` などの広いスコープを渡すことができますが、これは通常必要ではなく、SYCL `group_barrier` のコストは CUDA `__syncthreads` バリアよりも高くなる可能性があります。

```
q.parallel_for(nd_range<1>{16, 16}, [=](auto item) {
    // __syncthreads または
    // this_thread_block().sync() と同等:
    group_barrier(item.get_group());

    // __syncwarp または
    // tiled_partition<32>(this_thread_block()).sync() と同等:
    group_barrier(item.get_sub_group());
 }).wait();
```

図 21-8. CUDA と SYCL の同等のバリア操作

図 21-8 のコードは、CUDA と SYCL の同等のバリア構文を示しています。`this_thread_block` と `tiled_partition` を使用する新しい CUDA 協調グループ構文には、SYCL `group_barrier` にさらに近い同期関数があることに注目してください。これは、CUDA と SYCL 2020 の新しいバージョンがますます類似してきたことを示すもう 1 つの例です。

アトミックとフェンス

CUDA と SYCL はどちらも同様のアトミック操作をサポートしていますが、バリアと同様に、知っておくべき重要な違いがいくつかあります。最も重要な違いは、デフォルトのアトミック*メモリー順序*に関するものです。

多くの CUDA プログラムは、atomicAdd のように、アトミック関数がメモリーへのポインターを受け取る、古い C に似たアトミック構文で記述されています。これらのアトミック関数は*緩和されたアトミック*であり、デバイススコープで動作します。これらのアトミック関数には、atomicAdd_system や atomicAdd_block など、異なるスコープで動作するサフィックス付きバージョンもありますが、これらは一般的ではありません。

SYCL アトミック構文は少し異なり、C++20 の std::atomic_ref に基づいています（SYCL atomic_ref クラスの詳細と、std::atomic_ref との比較については、第 19 章を参照してください）。SYCL アトミックを CUDA atomicAdd 関数と同等にしたい場合、図 21-9 に示すように、SYCL atomic_ref が同様の memory_ order::relaxed メモリー順序と memory_scope::device スコープを持つように宣言する必要があります。

```
q.parallel_for(count, [=](auto id) {
  // SYCL atomic_ref は、atomic_ref タイプの一部として
  // デフォルトの順序とデフォルトのスコープを指定する必要があります。
  // CUDA atomicAdd の動作と一致させるには、
  // デバイススコープを持つ緩和されたアトミックが必要です：
  atomic_ref<int, memory_order::relaxed,
             memory_scope::device>
       aref(*buffer);

  // メモリー順序が指定されていない場合は、
  // デフォルトが使用されます：
  aref.fetch_add(1);

  // アトミック操作の一部としてメモリーの順序と
  // スコープを指定することもできます：
  aref.fetch_add(1, memory_order::relaxed,
                    memory_scope::device);
});
```

図 21-9. CUDA と SYCL の同等のアトミック操作

新しい CUDA コードでは、CUDA C++ 標準ライブラリーの `cuda::atomic_ref` クラスが使用される場合があります。`cuda::atomic_ref` クラスは SYCL の `atomic_ref` クラスに似ていますが、いくつか重要な違いもあるので注意が必要です:

- CUDA `atomic_ref` の場合、スコープはオプションですが、指定されていない場合はデフォルトでシステム全体になります。SYCL `atomic_ref` は、すべてのケースでアトミックスコープを指定する必要があります。
- CUDA `atomic_ref` のデフォルトのアトミック順序は無条件に順次一貫性がありますが、
- SYCL `atomic_ref` では異なるデフォルトのアトミック順序が指定される場合があります。デフォルトのアトミック順序を指定することで、SYCL コードはより簡潔になり、アトミック順序が順次一貫性以外の場合でも `+=` などの演算子を使用できるようになります。

コードまたはアルゴリズムにアトミック性が必要である場合、留意すべき点が 1 つあります。一部のアトミック操作とアトミックスコープは SYCL 仕様では必須ではなく、すべての SYCL デバイスでサポートされているわけではない可能性があります。これは CUDA デバイスにも当てはまりますが、SYCL デバイスには多様性があるため、SYCL では特に覚えておくことが重要です。SYCL デバイスのプロパティーを照会する方法の詳細については第 12 章を参照してください。SYCL デバイスまたはコンテキストでサポートされる可能性のあるアトミック機能ついては第 19 章を参照してください。

その他の違い

ここでは、CUDA コードを SYCL に移植する際に留意すべきその他の違いについて説明します。

item クラスとビルトイン変数

CUDA と SYCL の大きなスタイル上の違いの 1 つは、カーネル・インスタンスが N 次元並列インデックス空間内の位置を識別する方法です。第 4 章で説明したように、すべての SYCL カーネルでは、item、nd_item、id、または並列インデックス空間内のワーク項目を識別する整数引数が必要です。
item クラスと nd_item クラスは、グローバル範囲、ローカル範囲、およびワーク項目が属するさまざまなグループなど、並列インデックス空間自体に関する情報を照会するのにも使用できます。

CUDA カーネルには、並列インデックス空間内の CUDA スレッドを識別する引数は含まれていません。代わりに、CUDA スレッドは、blockIdx や threadIdx などのビルトイン変数を使用して並列インデックス空間内の場所を識別し、gridDim や blockDim などのビルトイン変数を使用して並列インデックス空間自体に関する情報を表します。協調グループを使用する新しい CUDA カーネルは、this_thread_block などのビルトイン関数を呼び出すことで、特定の協調グループを暗黙的に構築することもできます。

これは通常、記述されるコードも機能に影響しない構文の違いのみですが、呼び出された関数がワーク項目のインデックスを知る必要がある場合など、SYCL カーネルが呼び出された関数に item または nd_item を渡すケースがあることを意味します。

コンテキスト

CUDA と SYCL のもう 1 つの概念的な違いは、SYCL のコンテキストです。SYCL コンテキストは、SYCL デバイスセットの SYCL アプリケーションの状態を格納するオブジェクトであることを思い出してください。たとえば、SYCL コンテキストには、メモリー割り当てやコンパイルされたプログラムに関する情報が格納される場合があります。コンテキストは SYCL アプリケーションの重要な概念です。それは、単一の SYCL アプリケーションが、複数のバックエンド API を使用して、複数のベンダーのデバイスをサポートする可能性があるためです。

ほとんどの場合、SYCL プログラムはコンテキストの存在をまったく認識せず、本書のほとんどのサンプルプログラムではコンテキストを作成したり、操作していません。ただし、プログラム内で暗黙的または明示的に追加の SYCL コンテキストを作成する場合は、異なる SYCL コンテキストを持つコンテキスト固有の SYCL オブジェクトを使用しないように注意する必要があります。複数のコンテキストを不用意に使用すると、SYCL カーネルをコンテキストごとに 1 回ずつ複数回コンパイルするなど、プログラムの実行が非効率になる可能性があります。最悪の場合、SYCL オブジェクトをコンテキスト間で混在させることで、未定義の動作が発生し、プログラムが移植できなくなったり、一部のバックエンドやデバイスで不適切に実行される可能性があります。

CUDA にもコンテキストの概念がありますが、CUDA コンテキストは低レベルの CUDA ドライバー API によってのみ公開されることに注意してください。ほとんどの CUDA プログラムは、コンテキストを作成したり操作することはありません。

エラーチェック

考慮すべき最後の違いは、エラーチェックとエラー処理に関するものです。CUDA は C 言語の伝統を受け継いでいるため、CUDA のエラーは CUDA 関数呼び出しからのエラーコードとして返されます。ほとんどの CUDA 関数は、失敗したエラーコードは、関数へのパラメーターが正しくないなど、エラーを返す関数内のエラーを示します。ただし、`cudaDeviceSynchronize` などの CUDA 関数では、エラー値によってデバイスで発生した非同期エラーが返されることもあります。

SYCL には同期エラーと非同期エラーがありますが、どちらのタイプのエラーも SYCL 関数からの戻り値ではなく、SYCL 例外を使用して報告されます。SYCL におけるエラー検出とエラー処理の詳細については、第 5 章を参照してください。

SYCL にはまだない CUDA の機能

これまで、CUDA と SYCL の両方に機能が存在するが、表現方法が異なるケースについて説明してきました。ここでは、(現時点では) CUDA にはあるが、SYCL には同等のものがないいくつかの機能について説明します。ここにはすべてを網羅するリストはありませんが、SYCL への移行時に労力が必要になる可能性のある、CUDA アプリケーションで一般的に使用される機能の一部を説明します。

ベンダー固有の機能は、標準仕様の拡張であるか、完全にベンダー固有の API で定義されているかに関わりなく、標準化プロセスの重要な部分であることに注意してください。ベンダー固有の機能は、重要な実装の経験をもたらし、機能が改良されて標準化に組み込まれる前にその価値を証明できるようにします。これらの機能の多くは、SYCL 標準に組み込むためにすでに積極的に開発されており、一部は標準の拡張機能としてすでに利用可能になっている可能性があります。

参加しましょう!

ユーザーや開発者からのフィードバックは、標準化プロセスのもうひとつの重要な部分です。新しい機能のアイデアがある場合、または拡張機能や別の API の機能が有益だと思った場合は、ぜひ参加を検討してください。SYCL はオープン・スタンダードであり、多くの SYCL 実装はオープンソースであるため、成長を続ける SYCL コミュニティーに簡単に参加できます。

グローバル変数

プログラマーは早い段階でグローバル変数を使用しないように指示されますが、場合によってはグローバル変数が適切であることもあります。便利な定数、ルックアップ・テーブル、またはデータ並列カーネルを実行するすべてのワーク項目がアクセスしてその他の値を格納するため、グローバル変数を使用することを選択できます。

CUDA は、異なるアドレス空間にある、異なる存続期間を持つグローバル変数をサポートします。たとえば、CUDA プログラムは、各デバイスに固有のグローバルメモリー空間に `__device__` グローバル変数を宣言できます。これらのグローバル変数は、ホストによって設定またはホストから読み取られ、カーネルを実行しているすべての CUDA スレッドからアクセスされます。CUDA プログラムは、CUDA 共有メモリー空間で `__shared__` グローバル変数を宣言することもできます（これは、SYCL ローカルメモリーで宣言された変数と同等であることに注意してください）。この変数は、CUDA ブロックごとに一意であり、そのブロック内の CUDA スレッドによってのみアクセスできます。

SYCL はデバイスコード内のグローバル変数をまだサポートしていませんが、同様の機能を提供する拡張機能が開発中です。

協調グループ

この章の前半で説明したように、CUDA の最近のバージョンでは、バリアやシャッフル関数などの集合操作に、代替構文を提供する協調グループがサポートされています。SYCL のグループ・オブジェクトとグループ・アルゴリズム・ライブラリーは、CUDA 協調グループと多くの類似点がありますが、いくつかの重要な違いも残っています。

最大の違いは、SYCL グループ関数は現在、定義済みの SYCL ワークグループおよびサブグループ・クラスでのみ機能しますが、CUDA 協調グループはより柔軟であることです。たとえば、CUDA プログラムは、既存のグループを小さなグループのセットに分割する固定サイズの `tiled_partition` グループを作成したり、CUDA ワープ内の現在アクティブな CUDA スレッドのセットを `coalesced_group` として表現する場合があります。

CUDA プログラムでは、さらに、ワークグループよりも大きな協調グループを作成することもできます。たとえば、CUDA プログラムは、グリッド内のすべての CUDA スレッド（つまり、グローバル範囲内のすべてのワーク項目）を表す `grid_group` を作成したり、*スレッド・ブロック・クラスター*内のすべての CUDA スレッドを表す `cluster_group` を作成する場合があります。

これらの新しい、より大きなグループを効果的に使用するには、グリッド内のすべての CUDA スレッドが連携できるようにするため、またはスレッド・ブロック・クラスターの次元を指定するために、特別なホスト API 関数を使用して CUDA カーネルを起動する必要があります。

SYCL はまだ CUDA のすべての協調グループタイプをサポートしていませんが、SYCL に追加のグループタイプを追加する拡張機能が開発中です。SYCL 2020 でのグループ・オブジェクトとグループ・アルゴリズムの導入により、SYCL はこの機能をサポートできるようになりました。

行列乗算ハードウェア

この項で説明する最後の機能は、*行列乗算*ハードウェア（*行列乗算累算（MMA）*ハードウェア、*テンソルコア*、または*シストリック・アレイ*とも呼ばれる）へのアクセスです。これらはすべて、多くの人工知能（AI）ワークロードの鍵となる行列乗算演算を高速化するために特別に設計された専用ハードウェア・エンジンの別名です。これらのワークロードをカスタマイズする場合、最高のパフォーマンスを実現するには、データ並列カーネル内の行列乗算ハードウェアにアクセスできることが重要です。

CUDA は、*ワープ行列乗算および累算（WMMA）*関数で行列乗算ハードウェアへのアクセスを提供します。これらの関数により、ワープ内の CUDA スレッド（サブグループ内のワーク項目と同等）が協調して、小さな行列タイルで行列の乗算と累算の操作を実行できるようになります。これらの行列タイルの要素は、一部のデバイスやアルゴリズムでは 32 ビットの `float` または 64 ビットの `double` にできますが、より一般的には、8 ビットの `char`、16 ビットの `half`、または `bfloat16s`（`bf16`）などの特殊な AI 型である低精度の型が使用されます。

CUDA と SYCL はどちらも、行列乗算ハードウェアのサポートを積極的に進化させています。これは、さまざまなベンダーが最初にベンダー固有のメカニズムを介してベンダー固有の機能をサポートし、その後機能が改良され、共通のベスト・プラクティスが標準に追加される方法の良い例です。

移植ツールとテクニック

幸いなことに、アプリケーションを CUDA から SYCL に移行する場合、手動で行う必要はなく、ツールを使用して移行の一部を自動化できます。ここでは、移行を支援するツールとテクニックについて説明します。

dpct と SYCLomatic を使用したコードの移行

この項では、DPC++ 互換性ツール（dpct）と関連するオープンソースの SYCLomatic ツールについて説明します。ここで説明する概念は SYCLomatic にも同様に適用されますが、dpct を使用して CUDA サンプルを SYCL に自動的に移行します。

図 21-10 は、移行する単純な CUDA サンプルの重要な部分を示しています。このサンプルは、バッファーのブロックを反転します。これは実際に有用なサンプルではありませんが、CUDA 共有メモリーのグローバル変数、バリア、デバイスクエリー、メモリーの割り当てと初期化、カーネル・ディスパッチ、基本的なエラーチェックなど、自動移行ツールが処理する興味深いケースが含まれています。

```
__shared__ int scratch[256];
__global__ void Reverse(int* ptr, size_t size) {
  auto gid = blockIdx.x * blockDim.x + threadIdx.x;
  auto lid = threadIdx.x;

  scratch[lid] = ptr[gid];
  __syncthreads();
  ptr[gid] = scratch[256 - lid - 1];
}

int main() {
  std::array<int, size> data;
  std::iota(data.begin(), data.end(), 0);

  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp, 0);
  std::cout << "Running on device: " << deviceProp.name << "\n";

  int* ptr = nullptr;
  cudaMalloc(&ptr, size * sizeof(int));
  cudaMemcpy(ptr, data.data(), size * sizeof(int),
                        cudaMemcpyDefault);
  Reverse<<<size / 256, 256>>>(ptr, size);
  cudaError_t  result = cudaDeviceSynchronize();
  if (result != cudaSuccess) {
     std::cout << "An error occurred!\n";
  }
  // ...
```

図 21-10. 自動的に移行する簡単な CUDA プログラム

dpct の実行

これは単純な例であるため、dpct を呼び出して、移行する CUDA ソースファイルを渡すだけです。複雑なシナリオでは、アプリケーションのビルドプロセスの一部として dpct を呼び出して、移行する CUDA ソースファイルを識別します。詳しい情報と追加のトレーニング資料については、この章の最後にあるリンクを参照してください。

サンプルの CUDA ソースファイルで dpct を実行すると、図 21-11 のような出力が表示されることがあります。この出力から、いくつかのことが分かります。まず、ファイルが正常に処理されました。これは素晴らしいことです。ただし、dpct が移行できなかったケースを示す警告がいくつかあります。この例では、3 つの警告はすべて、CUDA と SYCL 間のエラーチェックの違いが原因です。私たちのプログラムでは、dpct はプログラムがエラーを生成しない場合に正しく動作する SYCL コードを生成できましたが、エラーチェックを移行することはできませんでした。

エラーチェックの警告は、dpct や SYCLomatic などの移行ツールがすべてを移行できないことを示す良い例です。移行の問題に対処するため、移行されたコードを確認して調整したり、保守性、移植性、パフォーマンスの観点から移行された SYCL コードを改善することが予想されます。

```
$ dpct source_file.cu
NOTE: Could not auto-detect compilation database for file 'source_file.cu' in
'/path/to/your/file' or any parent directory. The directory "dpct_output" is
used as "out-root"
Processing:
/path/to/your/file/source_file.cu/path/to/your/file/source_file.cu:38:5:
warning: DPCT1001:0: The statement could not be removed.
       std::cout << "An error occurred!\n";
       ^
/path/to/your/file/source_file.cu:37:3: warning: DPCT1000:1: Error handling
if-stmt was detected but could not be rewritten.
   if (result != cudaSuccess) {
   ^
/path/to/your/file/source_file.cu:36:24: warning: DPCT1003:2: Migrated API
does not return error code. (*, 0) is inserted. You may need to rewrite this
code.
   cudaError_t result = cudaDeviceSynchronize();
                        ^
Processed 1 file(s) in -in-root folder "/path/to/your/file"
```

図 21-11. この CUDA プログラムを移行したときのサンプル dpct 出力

この例では、移行されたコードをそのまま使用できます。図 21-12 は、DPC++ を使用して移行したコードをコンパイルする方法を示しています。NVIDIA GPU をサポートするコンパイラーを使用して、移行したプログラムをインテル GPU、インテル CPU、NVIDIA GPU で正常に実行できることがわかります。移行したプログラムを異なるデバイスを備えた別のシステムで実行する場合、出力が異なったり、選択したデバイスがシステムに存在しないと実行に失敗することがあるので注意してください。

```
$ icpx -fsycl -fsycl-targets=spir64,nvptx64-nvidia-cuda \
        migrated.cpp -o migrated
$ ./migrated
Running on device: Intel(R) UHD Graphics 770 Success.
$ ONEAPI_DEVICE_SELECTOR=opencl:cpu ./migrated
Running on device: 12th Gen Intel(R) Core(TM) i9-12900K Success.
$ ONEAPI_DEVICE_SELECTOR=ext_oneapi_cuda:gpu ./migrated
Running on device: NVIDIA GeForce RTX 3060
Success.
```

図 21-12. 移行された CUDA プログラムのコンパイルと実行

dpct 出力の検査

移行された出力を調べると、この章で説明した多くの違いが dpct によって処理されていることがわかります。たとえば、図 21-13 に示す移行された SYCL カーネルでは、`__shared__` グローバル変数 `scratch` がローカル・メモリー・アクセサーに変換され、カーネルに渡されていることがわかります。また、組み込み変数 `blockIdx` と `threadIdx` が `nd_item` クラスのインスタンスの呼び出しに置き換えられ、連続する次元の異なる規則が適切に処理されたこともわかります（`threadIdx.x` が `item_gt1.get_local_id(2)` の呼び出しに置き換えられるなど）。

```
void Reverse(int *ptr, size_t size,
              const  sycl::nd_item<3> &item_ct1,
              int  *scratch) {
   auto gid =
        item_ct1.get_group(2) * item_ct1.get_local_range(2) +
        item_ct1.get_local_id(2);
   auto lid = item_ct1.get_local_id(2);

   scratch[lid] = ptr[gid];
   item_ct1.barrier(sycl::access::fence_space::local_space);
   ptr[gid] = scratch[256 - lid - 1];
}
```

図 21-13. CUDA から移行された SYCL カーネル

また、図 21-14 に示す移行されたデバイスのクエリーなど、いくつかの dpct ユーティリティー関数を使用して、dpct がホストコードの違いの一部を処理したこともわかります。これらのヘルパー関数は、移行されたコードでのみ使用されます。移植性と保守性を考慮すると、新たな開発では標準の SYCL API を直接使用することを推奨します。

```
dpct::device_info deviceProp;
dpct::dev_mgr::instance().get_device(0).get_device_info(
     deviceProp);
std::cout << "Running on device: "
          << deviceProp.get_name() << "\n";
```

図 21-14. CUDA から移行された SYCL デバイス名を照会

ただし、一般に、dpct が生成する SYCL コードは可読性が高く、CUDA コードと移行された SYCL コード間のマッピングは明確です。追加の手動での編集が必要になることがありますが、dpct や SYCLomatic などの自動化ツールを使用すると、時間を節約し、移行中のエラーを軽減できます。

まとめ

この章では、CUDA をサポートする SYCL コンパイラーを使用して、アプリケーションを CUDA から SYCL に移行し、CUDA デバイスを含むあらゆる SYCL デバイスでアプリケーションを実行する方法について説明しました。

まず、用語は別として、CUDA プログラムと SYCL プログラムの間にある多くの類似点を調べることから始めました。CUDA と SYCL は基本的に同じカーネルベースの並列処理アプローチを使用しており、実行モデルとメモリーモデルも似ているため、CUDA プログラムを SYCL に移行するのは比較的簡単であることがわかりました。
また、CUDA と SYCL に構文上または動作上の微妙な違いがあるいくつかの点についても調査しました。そのため、CUDA アプリケーションを SYCL に移行する際には、これらの違いに留意しておくとよいでしょう。
さらに、CUDA にはあるが SYCL にはない（まだ!）いくつかの機能についても説明し、ベンダー固有の機能が標準化プロセスの重要な部分である理由についても説明しました。

最後に、移行プロセスの一部を自動化するいくつかのツールを調査し、dpct ツールを使用して単純な CUDA の例を SYCL に自動的に移行しました。ツールがコードの大部分を自動的に移行し、機能的に正しく読みやすいコードを生成する様子を確認しました。より複雑なアプリケーションでは、追加の確認と修正が必要になる場合がありますが、移行後は、移行した SYCL の例を別の SYCL デバイスで実行できました。

関連情報

CUDA コードを SYCL に移行するのは人気のトピックであり、詳細を学習できるリソースが他にも多数あります。著者が有用であると判断した 2 つのリソースを以下に示します:

- CUDA から SYCL への移行方法を示す一般情報とチュートリアル（tinyurl.com/cuda2sycl）（https://www.isus.jp/products/dpct/cuda-sycl-migration-workflow/）（日本語）（https://www.isus.jp/ai/porting-ai-codes-from-cuda-to-sycl-part1/）（日本語）
- DPC++ 互換性ツールの入門ガイド（tinyurl.com/startDPCpp）（https://www.isus.jp/wp-content/uploads/dpct/2025/get started guide/index.html）（日本語）

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

エピローグ　SYCL の将来の方向性

今こそ、SYCL を使用した C++ プログラミングのすべてを理解したという安らぎと平穏を感じてください。すべてのピースが埋まりました。

これまでの章のサンプルコードでは、標準の SYCL 2020 機能を使用して、多様なハードウェアで実行できるように努めてきました。拡張機能（相互運用性や FPGA 固有の拡張機能など）を使用した箇所については、その旨を明記しています。ただし、このエピローグに示されている将来を見据えたコードは、2023 年半ばの時点ではどのコンパイラーでもコンパイルできません。

このエピローグでは、将来について推測します。私たちの水晶玉を読むのは少々難しいかもしれませんし、このエピローグにはいかなる保証もありません。この本の初版で私たちが予測したいくつかの機能は実現しましたが、実現しなかったものもあります。

このエピローグでは、私たちが期待している今後の SYCL 機能と DPC++ 拡張機能を少しだけ紹介します。このエピローグに示されているサンプルコードがコンパイルできることは保証されません。一部は、本書の出版後にリリースされたコンパイラーと既に互換性がある可能性がありますが、その他は、構文を少し調整しないとコンパイルできない可能性があります。一部の機能は拡張機能としてリリースされるか、SYCL の将来のバージョンに組み込まれる可能性がありますが、その他の機能は無期限に実験的な機能のままである可能性があります。この本に関連付けられている GitHub リポジトリー内のサンプルコードは、進化するにつれて新しい構文に合わせて更新される可能性があります。
同様に、本書にも訂正があり、随時追加される可能性があります。

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2

これら 2 つの場所（コード・リポジトリーと書籍の正誤表 - リンクは第 1 章の冒頭にあります）で更新を確認することを推奨します。

C++11、C++14、C++17 とのより緊密な連携

SYCL と C++ の緊密な連携を維持することには 2 つの利点があります。まず、SYCL が標準 C++ の最新かつ最高の機能を活用して、開発者の生産性を向上できるようになります。次に、SYCL で導入された異種プログラミング機能が C++ の将来の方向性に良い影響を与える可能性が高まります。

SYCL 1.2.1 は C++11 をベースとしており、SYCL 2020 のインターフェイスに対する最大の改善点の多くは、C++14（ジェネリック・ラムダなど）と C++17（クラス・テンプレート引数推論（CTAD）など）で導入された言語機能によってのみ可能となりました。SYCL と C++ はさらに緊密になっていくと予想され、すでにいくつかの刺激的な取り組みが進行中です。

C++ 標準テンプレート・ライブラリー（STL）には、第 17 章で説明した並列パターンに対応するアルゴリズムがいくつか含まれています。STL のアルゴリズムは通常、反復子のペアで指定されるシーケンスに適用され、C++17 以降では、順次実行か並列実行かを指定する*実行ポリシー*引数をサポートします。
この標準では、実装で独自の実行ポリシーを定義することも許可されており、第 18 章で説明する oneAPI DPC++ ライブラリー（oneDPL）は、このカスタム実行ポリシーを活用して、アルゴリズムを SYCL デバイスで実行できるようにします。その結果、異種デバイスのプログラミングにおける生産性の高いアプローチが実現します。アプリケーションを STL アルゴリズムのみで表現できる場合、oneDPL を使用すると、SYCL カーネルコードを 1 行も記述することなく、システムのアクセラレーターを利用できるようになります。STL アルゴリズムが特定の SYCL 概念（バッファーなど）とどのように相互作用するか、必要なすべての標準ライブラリー・クラス（`std::complex`、`std::atomic` など）をデバイスコードで使用できるようにするにはまだ未解決の問題がありますが、oneDPL はホストコードとデバイスコードを統合する長い道のりの第一歩です。

C++20、C++23 以降の機能の採用

SYCL 仕様は、使用する機能が多くのコンパイラーでサポートされてるように、意図的に C++ の後追いとなっています。しかし、SYCL 委員会のメンバー（その多くは ISO C++ 委員会にも参加）は、C++ の将来のバージョンがどのように開発されるか注意深く見守っています。

ここで説明する C++ や SYCL の機能のうち、まだ仕様として確定していないものを採用するのは早急であるかもしれません。機能は標準化される前に大幅に変更される可能性があります。それでも、将来の SYCL プログラムの外観や動作に影響する可能性のある機能がいくつか議論されており、それはまた議論する価値があります。

SYCL 2020 の一部の機能は C++20 から得たもの（例: `std::atomic_ref`）であり、その他の機能は `sycl::` 名前空間に*事前採用*されています（例: `std::bit_cast`、`std::span`）。SYCL の時期公式リリースに向けて、C++20 との整合性をさらに高め、その最も有用な部分を組み込む予定です。例えば、C++20 では、`std::latch` と `std::barrier` の形式で追加のスレッド同期ルーチンが導入されました。同様のインターフェイスを使用してデバイス全体のバリアを定義する方法については、すでに第 19 章で説明しましたが、新しい C++20 構文のコンテキストでサブグループ・バリアとワークグループ・バリアを再検討する意味があるかもしれません。

C++23 の最も興味深い機能の 1 つは、ポインターの多次元配列構文と、基礎となるデータへのアクセスを制御する拡張ポイントとしての `AccessorPolicy` の両方を提供する、データの非所有ビューである `mdspan` です。

これらのセマンティクスは SYCL アクセサーのセマンティクスと似ており、mdspan を使用すると、図 EP-1 に示すように、バッファーと USM 割り当ての両方でアクセサーのような構文を使用できるようになります。

```
queue q;
constexpr int N = 4;
constexpr int M = 2;
int* data = malloc_shared<int>(N * M, q);

stdex::mdspan<int, N, M> view{data};
q.parallel_for(range<2>{N, M}, [=](id<2> idx) {
      int i = idx[0];
      int j = idx[1];
      view(i, j) = i * M + j;
}).wait();
```

図 EP-1. mdspan を使用して USM ポインターにアクセサーのようなインデックスを付加

SYCL が mdspan を正式にサポートするのは時間の問題でしょう。その間、関心のある読者は、Kokkos プロジェクトの一部として利用可能なオープンソースの製品品質のリファレンス実装を試してみることをお勧めします。

SPMD と SIMD プログラミングの混在

C++ に提案されているもう 1 つの興味深い機能は、C++ での明示的なベクトル並列処理向けの移植可能なインターフェイスを提供する std::simd クラス・テンプレートです。このインターフェイスを採用すると、第 11 章で説明したベクトル型の 2 つの異なる使用法（プログラマーの利便性のためのベクトル型の使用と、忍者プログラマーによる低レベルのパフォーマンス・チューニングのためのベクトル型の使用）を明確に区別できるようになります。同じ言語で SPMD と SIMD 両方のプログラミング・スタイルがサポートされていることから、カーネルが使用するスタイルをどのように宣言すればよいのか、また同じカーネル内でスタイルを組み合わせて使用できるのかなど、興味深い疑問も生じます。

私たちは、この質問に対する潜在的な解決策として、DPC++ 拡張機能（sycl_ext_oneapi_invoke_simd）の開発を始めました。これは新しい invoke_simd 関数（std::invoke をモデル化）を提供し、開発者が SPMD カーネル内部から明示的にベクトル化（SIMD）されたコードを呼び出すことを可能にします。invoke_simd の呼び出しは、2 つのプログラミング・スタイルで暗示される 2 つの実行モデル間の明確な境界として機能し、それらの間でデータがどのように流れるか定義します。
図 EP-2 のコードは、スカラーとベクトル（simd）引数の組み合わせを受け取る関数を呼び出す、invoke_simd の使用法の単純な例を示しています。

```
// 関数は 1 つのベクトル引数（x）と 1 つのスカラー引数（n）
// を必要とします
simd<float, 8> scale(simd<float>, 8> x, float n) {
  return x * n:
}

q.parallel_for(…, sycl::nd_item<1> it)
   [[sycl::reqd_sub_group?size(8)]] {
  // SIMD コードでは、各作業項目は独自の x 変数と n 変数
  // を持ちます
  float x = …;
  float n = …;

  // サブグループ内のワークアイテムを使用して
  // SIMD 関数（スケール）を呼び出します。
  // 各ワークアイテムの x 値は simd<float, 8> に結合されます。
  // n の値は、すべてのワークアイテムで
  // 同じ（均一）になるように定義されます。
  // 返される simd<float, 8> はアンパックされます
  sycl::sub_group sg = it.get_sub_group();
  float y = invoke_simd(sg, scale, x, uniform(n));
});
```

図 EP-2. SPMD カーネルから SIMD 関数を呼び出す簡単な例

invoke_simd によるアプローチにはいくつかの利点があります。まず、予期せぬ驚きがあってはなりません。異なる実行モデルを持つ関数は明示的に呼び出され、ユーザーにはデータを転送する方法を記述する責任があります。第 2 に、このメカニズムにより、きめ細かな特殊化が可能になります。

つまり、SPMD コードの残りを破棄することなく、明示的にベクトル化された数行のコード（パフォーマンス・チューニング用など）を記述できます。最後に、拡張が容易です。`invoke_simd` 自体は、単純なオーバーロードを介して新しいグループや新しい引数マッピングをサポートするように拡張でき、同様の `invoke_*` 関数を導入して、異なるコンテキスト（SYCL 以外の言語で記述されたコードなど）との相互運用性を処理できます。

アドレス空間

SYCL 2020 による汎用アドレス空間サポートの導入により、使用しているメモリーの種類を意識せずに通常の C++ ポインターを使用できるようになり、多くのコードが大幅に簡素化される可能性があります。多くの最新アーキテクチャーは、汎用アドレス空間のハードウェア・サポートを提供しているため、通常の C++ ポインターを使用するコードは多様なマシンで動作し、パフォーマンスのオーバーヘッドは最小限に抑えられると期待できます。

ただし、汎用アドレス空間のサポートが複雑になる（古い、またはより特殊な目的の）アーキテクチャーもあります。一部のハードウェアでは、異なる種類のメモリーにアクセスするため異なる命令が使用される場合があり、コンパイラーはコンパイル時に明確的なアドレス空間を認識する必要があります（つまり、正しい命令を生成する必要があります）。
汎用アドレス空間を表現できない SYCL バックエンドも存在する可能性があります（OpenCL 1.2 など）。SYCL 2020 は、アドレス空間を推論する一連の推論規則を通じて、このようなハードウェアやバックエンドに対応しています。

アドレス空間推論規則は SYCL 1.2.1 から継承されており、SYCL 2020 仕様には、これらの規則が SYCL の将来のバージョンで再検討される注記が含まれています。本書の執筆時点では、これらの規則がどのように変更されるか正確には不明ですが、SYCL の長期的な考え方は明確です。ほとんどの場合、アドレス空間の管理を気にすることなく、コンパイラーとハードウェアが適切に処理することを信頼する必要があります。

特殊化メカニズム

また、対象デバイスのプロパティー（*アスペクト*）（デバイスの種類、特定の拡張機能のサポート、ワークグループのローカルメモリーのサイズ、コンパイラーによって選択されたサブグループのサイズなど）に基づいてカーネルを特化できるようにコンパイル時のクエリーを導入する計画もあります。
このようなクエリーには、現在 C++ には存在しない特殊な種類の定数式を表します。ホストコードがコンパイルされるときは必ずしも `constexpr` ではありませんが、ターゲットデバイスが認識されると `constexpr` になります。

この“デバイス定数式”のコンセプトを公開する正確なメカニズムはまだ設計中です。これは、SYCL 2020 で導入された特殊化定数機能に基づいて構築され、図 EP-3 に示すコードと同様の外観と動作になることが期待されます。

```
h.parallel_for(range{1}, [=](id<1> idx) {
   if_device_has<aspect::cpu>([&]() {
       /* CPU に特化したコード */
       out << "On a CPU!" << endl;
   }).else_if_device_has<aspect::gpu>([&]() {
       /* GPU に特化したコード */
    out << "On a GPU!" << endl;
   });
});
```

図 EP-3. カーネルコンパイル時にデバイスの側面に基づいてカーネルコードを特殊化

コンパイル時のプロパティー

SYCL では、*プロパティー・リスト*をコンストラクターに渡すことで、特定のクラス（バッファー、アクセサーなど）の動作を変更できます。これらのプロパティーは非常に強力ですが、コンストラクターに渡されるプロパティーは実行されるまで不明であるため、その効果は制限されます。コンパイル時に特定のプロパティーを宣言できるようにすると、

実行時のチェック数が減り、特定のプロパティーが存在する場合にコンパイラーがホストコードとデバイスコードの両方を特殊化できるようになるため、パフォーマンスが大幅に向上する可能性があります。

DPC++ コンパイラーは、コンパイル時プロパティー（`sycl_ext_oneapi_properties`）の実験的な拡張機能をサポートしており、既にさまざまな拡張機能が有効になっています:

- アドレス空間を超えた情報を付加したポインター。これは、`sycl::multi_ptr`（`sycl_ext_oneapi_annotated_ptr`）の将来に影響する可能性があります
- カーネル構成コントロールは、C++ 属性を置き換え、ライブラリーのみの SYCL 実装の機能を向上できます（`sycl_ext_oneapi_kernel_properties`）
- メモリー動作とアクセス制御の説明（`sycl_ext_oneapi_device_global`、`sycl_ext_oneapi_prefetch`）

コンパイル時プロパティーに関する初期の経験は非常に良好で、その潜在的な使用例は増え続けています。適用範囲が広いことを考慮すると、将来の SYCL 仕様でコンパイル時プロパティーのいくつかが採用されることを期待しています。

まとめ

SYCL はすでに大きな注目を集めていますが、これはほんの始まりに過ぎません。私たち（コミュニティー）の先はまだ長く、異種プログラミングのベストプラクティスを抽出し、パフォーマンス、移植性、生産性のバランスを実現する新しい言語機能を設計するには、継続的な努力が必要です。

皆さんの支援が必要です。C++（またはその他のプログラミング言語）で使用する機能が SYCL にない場合は、お問い合わせください。共に SYCL を使用した C++ の将来の方向性を形作ることができます。

関連情報

- Khronos SYCL Registry、www.khronos.org/registry/SYCL
- H. Carter Edwards et al., “mdspan: A Non-Owning Multidimensional Array Reference”、wg21.link/p0009
- D. Hollman et al., “Production-Quality mdspan Implementation”、github.com/kokkos/mdspan
- Intel DPC++ Compiler Extensions、tinyurl.com/syclextend

オープンアクセス この章は Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International の条件に従ってライセンスされています。

（http://creativecommons.org/licenses/by/4.0/）では、元著者とソースに適切なクレジットを与え、Creative Commons ライセンスへのリンクを提供し、ライセンスされた素材を変更したかどうかを示せば、あらゆるメディアや形式での非営利目的の使用、共有、配布、複製が許可されます。

本書に掲載されている画像やその他の第三者の素材は、素材のクレジットラインに別途記載がない限り、本のクリエイティブ・コモンズ・ライセンスの対象となります。資料が本書のクリエイティブ・コモンズ・ライセンスに含まれておらず、意図する使用が法定規制で許可されていないか、許可された使用を超える場合は、著作権所有者から直接許可を得る必要があります。

訳者あとがき

本書の初版が 2020 年に出版されたとき翻訳を検討しましたが、SYCL 2020 の仕様が間もなく公開され、一部内容がそぐわない部分が出てきたため、本書の更新を見守っていました。この第 2 版が出版され、本書の内容も SYCL 2020 の仕様に合わせて更新されたため、新たに翻訳の検討を始めました。本書の翻訳が完成したのは、2026 年 4 月ですが、インテル社の oneAPI ツールキットもバージョン 2025 が出荷されており、間もなく 2026 バージョンがリリースされると思われます。

インテル® DPC++/C++ コンパイラー 2025.2 までは、対応する NVIDIA と AMD GPU 向けのプラグインが Codeplay 社から提供されていましたが、コンパイラーの 2025.3 以降は対応プラグインが提供されなくなりました。オープンソース版から自身でビルドすれば利用できます。このプラグインを利用すると、本書でも紹介しているマルチベンダー、マルチアーキテクチャー向けの検証を簡単に行えましたが、まことに残念なことです。

SYCL 2020 の仕様もリビジョン 11 まで更新されています。本書の脚注でも紹介していますが、インテル® DPC++/C++ コンパイラー 2024.1 からは、正式に SYCL 2020 をサポートするコンパイラーとなりました。SYCL や oneAPI に関連する書籍や記事は iSUS（http://www.isus.jp）で公開中ですので、本書と合わせてご覧ください。

最後に、日本語の査読を快く受けれ入れてくれた池井さんに感謝します。

2026 年 3 月
iSUS 編集部
すがわら　きよふみ

インデックス

A

加速型と異種システム、1
アクセラレーター・デバイス
GPU（グラフィックス処理ユニット（GPU））*を参照*
事前（AOT）コンパイル、316
`all_of_group`、242
ジーン・アムダール、10
アムダールの法則、10
匿名ラムダ、346 – 347
`any_of_group`、242
アプリケーション・プログラミング・インターフェイス（API）、264
　バックエンドの相互運用性*を参照*
AOS（構造体配列）構造体、442 – 444
アスペクト、52
　デバイスアスペクト非同期*を参照*
　エラー、139
　ホストプログラムの実行、35
　キュー（アウトオブオーダー）、21
アトミック操作
`atomic_fence`、542
`atomic_fence_order_` 機能、541
`atomic_fence_scope_` 機能、541
`atomic_memory_order_` 機能、540
`atomic_memory_scope_` 機能、541
`atomic_ref` クラス、543
`cl::sycl::atomic` クラス（SYCL 2020 で非推奨）、543
`cl::sycl::atomic`（SYCL 2020 で非推奨）、543
データ競合、524
デバイス全体
　同期、553–556
`std::atomic` クラス、543
`std::atomic_ref` クラス、544 – 548
統合共有メモリー、550

B

バックエンドの相互運用性、559
　バックエンド、560
　`get_native` 関数、564 – 566
　`interop_handle`、566

© 2023 Intel Corporation.
J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-9691-2

バックエンドの相互運用性（*続き*）
カーネル、569 – 574
主な利点、562
低レベル API 機能、576
バリア同期、530
バッファー、67、72、78
アクセサー、78 – 80、180
アクセスターゲット、190
デダクション・タグ、195、197
get_access メソッド、192
モード、190
名前—アクセスデータ、198
プレースホルダー・アクセサー、191、197
コーディングの嗜好、181
コマンドグループ（CG）、207
myDoubles 配列、183
プロパティー、186
context_bound、188
use_host_ptr、187
use_mutex、187
set_final_data メソッド、189
set_write_back メソッド、189
USM（統合共有メモリー（USM）*を参照*）

C

キャッシュ一貫性のある非均一メモリーアクセス（cc-NUMA）、419
中央処理装置（CPU）アーキテクチャー、420
cc-NUMA システム、419 – 421
ハードウェア・スレッド、421
マルチコア、419 – 421
マルチコア・プロセッサー、417
並列処理、417
パフォーマンス、418 – 419
SIMD 命令
ハードウェア、422 - 428
ベクトル化、436 – 448
ソケット、421
スレッドレベル
並列処理、428 – 436
クラス・テンプレート引数デダクション (CTAD)、181
クロック周波数、500 – 501
cl::sycl::atomic（SYCL 2020 では非推奨）、543
コード実行、ホストデバイス集合関数*を参照*、通信ブロードキャスト関数、241
排他的/包括的スキャン、244
機能、241
permute_group_by_xor、245
shift_group_left、244
シャッフル関数、243 – 246
サブグループ、245
投票関数、242
XOR 操作、244
コマンドグループ（CG）アクション、203
記述、202、203
イベントベースの依存関係、203
実行、213

線形依存チェーンバッファー/アクセサー、207、208
イベント、207
インオーダー・キュー、205、206
タスク実行、204
“Y” パターンアクセサー、212
イベント、210
インオーダー・キュー、208、209
アウトオブオーダー・キュー、209、210
通信
バリア関数同期、223 – 225
集合関数、225、241 – 246
行列乗算カーネルの暗黙的/明示的キャッシュ、229
サブグループ
ブロードキャスト関数、237
集合関数、237 – 239
定義、235
ND-range parallel_ for、240 – 241
同期、236
ワークグループ・ローカル・メモリー
コンパイラーの最適化、226
info::device::local_mem_type、227
行列乗算カーネル、227 – 231
メモリー・サブシステム、226
ND-range カーネル、231 – 235
コンパイル時のプロパティー、612 – 613
並行性と並列性、28 – 29
コンテキスト、157、319、320、576
copy メソッド、214
C++ プログラミング、5
非同期、20
並行性と並列性 22、28 – 29
データ並列プログラミング、6
デッドロック、12
主要属性、14
CUDA コードの移行、581
プラットフォーム・モデル、321 – 322
アルゴリズムの設計が不十分、22
移植性/パフォーマンス 移植性、26 – 28
std::memcpy、19 – 22
C++ 標準ライブラリー
CPU/GPU/FPGA カバレッジ、508、509
クロスアーキテクチャー 移植性、510
DPC++ コンパイラー、509
メモリーモデル、535
std::swap、507、508
SYCL デバイス、507
CUDA
移行（CUDA コードの移行*を参照*）
興味深い、303
詳細列挙コード、301 – 302
get_info<>、300 – 301
has() インターフェイス、303

カスタム・メモリー・システムのメモリーアクセス、495、497
最適化、496
ステージ、495、496
静的結合、497
構造体、497

D

明示的データ管理、70
暗黙的、71
ローカルアクセス、69
並列処理/データフィード、67
データ移動
アクセサー、215
アプリケーション・パフォーマンス、213
コマンドグループ、215
明示的、167 – 169、213
グラフ（グラフのスケジュール*を参照*）
ホストと共有割り当て、215
暗黙的、214
データ移行、169
データ移動、170
細粒度
制御、172 – 174
ホスト・アプリケーション、169
memcpy、169
移行、170 – 171
プリフェッチ/mem_advise、173
共有割り当て、169
memcpy メソッド、167、214
プリフェッチ操作、215
update_host メソッド、214
データ並列処理アーキテクチャー、98
データ並列カーネル
C++ クラス、109
説明的な
プログラミング、105
実行空間、105 – 106
id クラス、110 – 111
item クラス、111 – 112
行列乗算、
108、109
操作、105
parallel_for
関数、107– 109
range クラス、109 – 110
簡略化された定義、110
SPMD プログラミング
モデル、105
デバイス固有
最適化、98
ハードウェア・リソース、98
階層データ、104
ホスト/デバイスコード、102 – 103
カーネルベース
プログラミング、101
カーネル形式、103 – 105
機能、132
フローチャート、130、131
機能性、130
言語機能、102
ループとカーネル、99 – 101
複数次元
カーネル、101 – 102
ND-range、104

パフォーマンス/移植性/生産性、99
プログラマーの生産性、98
シーケンシャル・セマンティクス、99
2 次元範囲、102
ワーク項目
データと実行範囲、129
多対一マッピング、128 – 129
マッピング計算、128
1 対 1 マッピング、128
データ並列プログラミング、13
デッドロック、21
デバッグプロセス
コンパイラー・オプション、324
デッドロック、324
ホスト/デバイス、323
キューのプロファイル、330 – 334
ランタイムエラー、327 – 330
sycl::stream、326
トレースおよびプロファイル・ツール、334 – 335
デバッグ技術、21
デバイスアスペクト、51、296、297
デバイスコード、34
デバイス定数式、611
デバイス情報
all_devices_have_v<aspect>、308
any_device_has_v<aspect>、308
コンパイル時のプロパティーとランタイム、308
正当性
デバイスキュー、305 – 306
フェッチ・パラメーター、304
パラメーター、304
カスタム・デバイス・セレクター、298
記述子、303
列挙メソッド
アスペクト、296 – 297
興味深い,300 – 302
カスタムデバイスセレクター、298 – 299
device クラス、300
デバイスクエリーメカニズム、300
デバイスセレクター、294
get_info<>、300 – 301
実装、293
get_info plus has()、303
カーネル特殊化、296
出力プログラム、301
好ましい解決策、299
堅牢なアプリケーション、296
try-catch、294、295
カーネル情報記述子、303
カーネル特殊化、309 – 311
プロパティー、291
特殊化定数、308
SYCL 仕様、290
テンプレート化されたカーネル、308
チューニング/最適化カーネルクエリー、308
デバイス選択、36、289、291、293、298

デバイス全体の同期、553
アトミック参照、555
実装、553
ND-range、554
食事する哲学者の問題、21
倍精度 A×X+Y
(DAXPY)、17–18
サンプルコードのダウンロード、313
SYCL コンパイラーのダウンロード、313
DPC++ コンパイラー、313、503
oneAPI DPC++ ライブラリー
(oneDPL)*を参照*
DPC++ 互換性ツール (dpct)
コンパイルと実行
プログラム、601 – 603
CUDA プログラム、599
ヘルパー関数、602 – 603
コードの移行、598 – 602
出力プロセス、601 – 602
DPC++ ライブラリー (oneDPL)、606

E

エラー処理
アプリケーション戦略、140 – 149
非同期
定義、138
検出/レポート、141
デバイス、149、150
異種
プログラミング、136
ホストプログラム/タスクグラフ
実行、137
主な機能、135
安全性、135 – 136
`std::function`、145
`std::terminate`、142
同期
定義、137
タイプ、136 – 138
実行ポリシー、606

F

ファットバイナリー、315
フェンス用メモリーモデル、223
フィールド・プログラマブル・ゲート
アレイ (FPGA) ビルディング・ブロック
要素、498
ルックアップ・テーブル、498
数学エンジン、498
オフチップ・ハードウェア、499
オンチップメモリー、499
ルーティング・ファブリック、499、500
カスタム・メモリー・システム、495 – 497
データ・フロー・アーキテクチャー、462
定義、451
カーネルが消費するチップ、459 – 460
ループ開始間隔
増分乱数、485、486
反復
プロセス、486、487
反復、485
ステージ、487、488
パイプライン並列処理

逆方向通信、480
効率的な利用、474
生成機能、477
ループ伝搬データ
依存関係、479 – 481
ループ開始
間隔、483 – 488
ND-range 実行
モデル、475 – 478
空間
実装、481 – 483
ステージ、472 – 474
連続反復、482
ワーク項目、475
パイプ
自動選択、493
ブロック化/非ブロック化、494
FIFO、489、490
情報、495
カーネル、491、492
モジュール設計/アクセス、490
パラメーター化、492、493
プロパティー、489
タイプ、491
ランタイムデバイス
事前、470 – 471
fpga_selector、467
SYCL、452
細粒度の特殊化、610
先入れ先出し（FIFO）、489、490
ファンクター、255 – 258
将来の方向性、605

G

汎用アドレス空間、610
グラフィックス処理ユニット（GPU）
グローバルメモリーのアクセス、405 – 409
キャッシュ/メモリー、385
コンパイル手順、317
計算依存、405
実行リソース、384
固定関数、384
レイテンシーを隠匿、398 – 399
高レベルのビルディング・ブロック、384–385
命令ストリーム、399
メモリー依存、405
占有率、399
カーネルのオフロード
抽象化、400
コスト、403 – 404
デバイスメモリー/リモートメモリー/ホストメモリー、404
SYCL ランタイム・ライブラリー、400
最適化
数学関数、413
小さなデータ型、412、413
パフォーマンス、383
SIMD 命令、391
単純なプロセッサー
利点、386
機能、386
行列乗算、386 – 388、391
oneMKL プロジェクト、387

グラフィックス処理ユニット（GPU）（*続き*）
並列処理、389 – 391
プロセッサー・リソース、390
タスクカーネル、388
トレードオフ、386
特殊化機能/拡張機能、414
サブグループ集合関数、412
ワークグループ・ローカル・メモリー、409 – 411
グラフ・スケジュール
CG（コマンドグループ*参照*）
データ依存関係、202
データ移動、213 – 216
ホスト同期、216 – 218
グループ・アルゴリズム、115、366
ジョン・グスタフソン、10

H

Hello, world!、6–7
異種システム、1
ホストコード、33
ホストデバイス `cpu_selector_v`、45
開発/デバッグ/展開、43 – 46
デバイス・コード・クラス、61
ホストタスク、63 – 65
送信、56
タスクグラフ、55 – 57
デバイス選択、51
アスペクト、51 – 53
メカニズム、54
GPU、45 – 50
`queue` クラス、58
キュー、37 – 42
単一ソースファイル
デバイスコード、34–35
ホストタスク、63

I

id クラス、110
イメージ、72
インオーダー・キュー、84
`invoke_simd`、609
`item` クラス、111

J

ジャストインタイム（JIT）コンパイル、316

K

カーネル
アクセラレーター・タイプ、250
出力・データアクセス
初期化、336
利点/欠点、249
事前（AOT）、259
正当性、306 – 307
ダイレクト・プログラミング、370
列挙メソッド、296
明示的/選択的オブジェクト、262

相互運用性、264
API 定義オブジェクト、569、570
非 SYCL ソース
言語、571 – 574
set_arg() と set_args()
インターフェイス、570
ソース/中間
表現、569
SPIR-V、573
ジャストインタイム (JIT)、259
カーネル・オブジェクト/カーネルバンドル、259
行列乗算、227 – 231
メモリーモデル、528
名前付き関数オブジェクト
定義、255–258
要素、256
operator() 関数、256 – 258
オプション属性、258
オブジェクト・ファイル、259
コンパイル済みカーネルバンドル、260
カーネルの照会、263
表現、249
チューニング/最適化、308

L

ラムダ式、23
匿名、26
匿名/名前なし関数
オブジェクト/クロージャー、251
キャプチャー・リスト、23
データ並列処理、102、103
デモコード、25
要素、252 – 255
関数オブジェクト、26
識別、254 – 255
暗黙的/明示的、254
カーネル、249
名前付き関数オブジェクト、256 – 259
パラメーター、24
テンプレート・パラメーター、254
名前なしパラメーター、255
ライブラリー
ビルトイン関数
ホストとデバイス、504
ホスト/デバイス、504
sycl::、506
C++ 標準ライブラリー、507 – 510
SYCL 実装、503

M

マップパターン、351 – 352、370
数学配列型 (marray)、271–273
marray クラス、267、271、273
マスクと予測、394
行列乗算と累算
(MMA)、597
メモリー (一貫性) モデル
acquire-release 順序付け、533
アトミック/非アトミック
操作、531 – 532
アトミック操作、534
atomic_ref クラス、531
バリア関数
同期、223

メモリー（一貫性）モデル（*続き*）
バリア/メモリー
フェンス、529 – 530
C++ スレッド、524
コア概念、525
データ競合/
同期、526 – 529
機能/能力、523
異種システム、524
命令/マイクロオペレーション、526
カーネル、528
メモリーモデル（
メモリーモデル*を参照*）
メモリー順序、532 – 534
並列アプリケーション
開発、527
並べ替え操作、532
シーケンシャル実行、526
SYCL 仕様、525
メモリーモデル
アトミックとフェンス
操作、540
`atomic_fence` 関数、542
`atomic`（アトミック操作）
バリアとフェンス*を参照*、542
C++ 標準ライブラリー、535
コンセプト、534
開発アプローチ、541
言語機能、535
ロード操作、538
`memory_order` 列挙
クラス、536 – 538
`memory_scope` 列挙クラス、539 – 541
デバイス機能の照会、540 – 542
SYCL メモリーモデル、536
CUDA コードの移行、579
C++ コード、581
協調グループ、596
機能、595
グローバル変数、595
グループ・アルゴリズム、596
行列乗算
ハードウェア、597 – 599
メモリーモデル、589
アトミック操作、590 – 592
等価なバリア、590
フェンス、591 – 593
その他の違い
コンテキスト、593
エラーチェック/エラー処理、594
`item` クラスとビルトイン変数、592、593
複数ターゲットと単一デバイスターゲット、579 – 581
ポーティング・ツール/テクニック、598 – 602
実世界の SYCL コード、581
類似点/相違点、583
バリア・デッドロック、589
連続次元、585
実行モデル、584
前方進行
保証、588

独立したスレッドのスケジュール、588
インオーダーとアウトオブオーダー、584 - 686
ワークグループ・サイズ（ワープサイズ）、587
標準化プロセス、595
用語、583
スレッド・ブロック・クラスター、596
複数アーキテクチャー・バイナリー、315
複数翻単位、344 – 345
multi_ptr、612

N

N 次元範囲（ND-range）通信/同期、221
データ並列カーネル、360
デバイス全体同期、554
get_global_id() 関数、123
get_local_id() 関数、123
group クラス、124 – 126
group_barrier 関数、233
カーネル形式
実行範囲、113 – 114
明示的、113
前方進行の保証、116
実装定義、114
規範構造、113
シャッフル操作、118
SIMD レーン、120
サブグループ、113、117 – 120
タスクベースの並列処理モデル、117
ベクトル化、117
ワークグループ、113、115 – 117
ワーク項目、113、115、116
ローカルアクセサー、231 – 232
nd_item クラス、124 – 125
nd_range クラス、122、123
parallel_for、121
パイプライン並列処理、475 – 478
sub_group クラス、126 – 127
同期関数、232 – 233
2 次元範囲、222
ワークグループ/ワーク項目、221 – 222、233 – 235
nd_range クラス、122
N-次元範囲（ND-range）*を参照*
none_of_group、242

O

oneAPI DPC++ ライブラリー（oneDPL）、510
binary_search、516
エラー処理モデル、520
ポリシーとホスト側反復子、515
オンラインのリソース、313
アウトオブオーダー・キュー、84、92

P

パック、377 – 379
パックパターン、358 – 359
parallel_for、107
並列パターン
　ダイレクト・プログラミング
　　ギャザー/スキャッター
　　操作、377
　　マップパターン、370
　　パック、377 – 379
　　リダクション・パターン、373 – 374
　　ステンシルパターン、371 – 373
　ハードウェア・デバイス、350
　高レベルの概要、350
　マップパターン、351 – 352
　パック、358
　プロパティー、349
　リダクション、356 – 358
　スキャン、356 – 357
　ステンシル、352 – 354
　STL アルゴリズム、367
　アンパック、359
　ベンダー提供のライブラリー、370
並列プログラミング
　ジーン・アムダール/ジョン・グスタフソン、10
　データ並列プログラミング、13
　異種システム、11
　Think Parallel（並列に考える）、9
　　データ並列性*を参照*、
パフォーマンスの移植性 98、290
プラットフォーム・モデル
　利点、320
　コンパイル処理
　　事前/ジャストインタイム、319
　　オフロードバンドラー/アンバンドラー、319
移植性、98、317
条件付きとマスク、394
printf と sycl::stream、326
プロファイル・キュー、330

Q

キュー
　定義、37 – 39
　デバイスセレクター、41
　device_selector クラス、41 – 42
　インオーダーとアウトオブオーダー、21
　メンバー関数、39
　複数キュー、40、41
　アウトオブオーダーとインオーダー、21
　プロファイル、330
　ワーク実行、39、40

R

競合状態、19
range クラス、109
書き込み後の読み取り（RAW）、202
reducer クラス、363
リダクション、354
reduction クラス、361
リダクション・ライブラリー、360

リダクション・パターン、
356 – 358、373 – 374
関連情報オンライン、313

S

スケーリング、11
スキャンパターン、356 – 357
Single instruction, multiple data
(SIMD)、98、608
アクション、285
中央処理装置ハードウェア・スレッド、
423
命令レベル
並列処理、423
複数レベルの並列処理、425
複数の計算、422
並列処理
ハードウェア、425
パフォーマンスの利点、422
STREAM トライアド
プログラム、425 – 427
スレッドレベル並列処理、424
ベクトル化、436 – 448
x86 アーキテクチャー、422
便利なタイプ、281、283
ハードウェア命令、271
メンタルモデル、269
ベクトル、269
Single Program Multiple Data
(SPMD)、105、393、105
プログラミング・モデル、269
単一ソース、31
特殊化メカニズム、611
SPMD/SIMD プログラミング
スタイル、608 – 610
標準テンプレート・ライブラリー
(STL)、606
`std::simd`、608
ステンシルパターン、352 – 354、371 –
373
ストリーム (`sycl::stream`)、326
配列構造体 (SOA)、443
サブグループ、113
送信、39
`SYCL_EXTERNAL`、344
SYCLomatic ツール、598 – 602
SYCL 標準、1、2、313
同期
アクセサー、217
バリア関数、223 – 225
デバイス全体、553
イベント、216
グラフ実行、216
ND-range カーネル、232 – 233
キュー・オブジェクト、216
サブグループ通信、236
スレッドレベル並列処理、431
`use_mutex`、217
ベクトル、438
同期、139
エラー、139
エラー処理
例外のキャッチ、144
定義、137
サブバッファー、139

同期（*続き*）
sycl::exception、143
try-catch 構造、143 – 144
未処理の C++
例外、141
キュー（インオーダー）、21

T

タスクグラフのデータ
コマンドグループ、84
データ依存関係、86
depends_on() メソッド、85
分離した依存関係、82
イベント、85
実行、80
ホストデバイス、55 – 57
読み取り後の読み取り（RAR）シナリオ、88
スレッドレベルの並列処理アフィニティー内部、431 – 434
要素、430
並列処理を活用、435
マッピング、430
メモリー、435 – 436
parallel_for カーネルコード、428、429
SYCL プログラム、430
TBB パーティショナー、434
ワークグループのスケジュール、430
throw_asynchronous、148
タイミングとプロファイル、330
トレースおよびプロファイル・ツールインターフェイス、334
翻訳単位、31、344、345

U

統合共有メモリー（USM）、67、92、153
利点、72
割り当てタイプ、73
アトミック操作、550
特性、154
通信、225 – 227
データ初期化、165 – 166
データ初期化と移動、165
定義、154
デバイス割り当て、154、157
ホスト割り当て、155
malloc、72
メモリー割り当て、156
aligned_malloc 関数、164
C++ アロケーター・スタイル、158、162 – 163
C++ スタイル、158、160 – 161
コンテキスト・オブジェクト、157
C スタイル、158 – 159
割り当て開放、164 – 165
malloc 関数、158
new/malloc/allocators、156
memset 関数、166
移動（データ移動*を参照*）
クエリー、174 – 177

共有割り当て、155
統合仮想アドレス、73
ユニットストライド/最高速の次元移動、585 – 586
アンパックパターン、359、379 – 380

V

vec クラス、267, 273
ベクトル
便利なタイプ、268
アドレスエスケープ、284
コンパイラー、281
ハードウェア
実装、280
ハードウェア命令、282
暗黙的、282
カーネル実行、281
メモリーアクセス、283
並列処理、281、282
SIMD/SPMD
命令、281
ワーク項目、282
CPU SIMD
AOS（構造体の配列）構造体、442 – 444
計算の複雑性、448
データタイプのインパクト、444 – 446
デスティネーション・レジスター、441
実行モデル、437 – 440
ギャザー/スキャッター
命令、445
ハードウェア、436 – 448
命令ストリーム、436
マスクとコスト、440 – 442
single_task, 446–448
サブグループ・バリア、438
ユニットストライド・ベクトル、444
ワーク項目/ワークグループ、438
データ収集、268 – 269
要素/要素タイプ、274
明示的コード、269
暗黙的、270
命令/クロックサイクル、270
相互運用性/バックエンド・ネイティブ関数、276
load() メンバー関数、274
ロード/ストア操作、274 – 276
メモリーレイアウト,281
NumElements
パラメーター、274
スカラー操作、267
SIMD マッピング、269
SIMD タイプ、284 – 285
store() メンバー関数、274
サブグループ・バリアと
シャッフル、269 – 271
swizzled_vec__ クラス、279
スウィズル操作、276 – 279
スウィズル、273
SYCL 1.2.1 仕様、268
vec クラス、274
ワーク項目、270
仮想関数、17
vote 関数、242

W、X

待機オブジェクト、7、19、20、39
wait_and_throw、39、148
ワープ行列乗算と加算（WMMA）、597
ウェブサイト、313
ワークグループ、113
ワークグループのバリア、115
ワークグループ・ローカル・メモリー、115
ワーク項目、113
読み取り後の書き込み（WAR）、202

Y、Z

“Y” パターン、205、211