インテル® ISPC ユーザーガイド

このドキュメントは ispc GitHub* に公開されている『Intel® ISPC User's Guide』の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

インテル® インプリシット SPMD プログラム・コンパイラー (インテル® ISPC) は、CPU と GPU 上で実行する SPMD (単一プログラム複数データ) プログラムを記述するためのコンパイラーです。SPMD によるプログラミングはグラフィックスや GPGPU プログラマーにはよく知られており、GPU シェーダー、CUDA*、そして OpenCL* カーネルで使用されています。SPMD の主な考え方は、1 つのデータ要素 (例えば、ピクセルシェーダーの場合はピクセル) を処理するようにプログラムを記述し、ハードウェアとランタイムが異なる入力 (異なるピクセル値など) でプログラムを並列に複数呼び出して実行することです。

インテル® ISPC は以下を目標とします。

- CPU と GPU 上で SPMD プログラムを実行することを望むパフォーマンス志向のプログラマーに、良好なパフォーマンスを発揮する C 言語のバリアントを提供します。
- プログラマーとハードウェアの間に軽量な抽象化レイヤーを提供します。特に、プログラマーがコンパイルされたアセンブリー言語とベースとなるハードウェアへのソースプログラムのマッピングを明確に推論できる実行モデルと、データモデルを持つシリアルプログラム向けの C 言語からの教訓に従います。
- 組込み関数を直接記述するような極めて生産性の低い作業を行うことなく、SIMD (単一プログラム複数データ) ベクトルユニットの計算能力を活用できます。
- C/C++ アプリケーション・コードと SPMD ispc コードが同一プロセッサー上で動作する際の緊密な連携を調査します。それには、2 つの言語間での軽量な関数呼び出し、コピーや再構成なしのポインターによる直接データ共有などが該当します。

システムをご利用された皆様のインテル® ISPC に対するご意見およびご感想をお待ちしています。特にインテル® ISPC を利用され、期待したような効果や結果が得られなかったという方は、是非ご意見をお寄せください。GitHub Discussions (英語) フォーラムに体験やコメントをお寄せいただくか、インテル® ISPC バグトラッカー (英語) にバグや機能的な要望をお送りください。ありがとうございます。

目次

インテル® ISPC の最近の改良

- インテル® ISPC 1.1 における更新
- インテル® ISPC 1.2 における更新
- インテル® ISPC 1.3 における更新
- インテル® ISPC 1.5.0 における更新
- インテル® ISPC 1.6.0 における更新
- インテル® ISPC 1.7.0 における更新
- インテル® ISPC 1.8.2 における更新
- インテル® ISPC 1.9.0 における更新
- インテル® ISPC 1.9.1 における更新
- インテル® ISPC 1.9.2 における更新
- インテル® ISPC 1.10.0 における更新
- インテル® ISPC 1.11.0 における更新
- インテル® ISPC 1.12.0 における更新
- インテル® ISPC 1.13.0 における更新
- インテル® ISPC 1.14.0 における更新
- インテル® ISPC 1.14.1 における更新
- インテル® ISPC 1.15.0 における更新
- インテル® ISPC 1.16.0 における更新
- インテル® ISPC 1.17.0 における更新
- インテル® ISPC 1.18.0 における更新

インテル® ISPC 入門

- インテル® ISPC のインストール
- インテル® ISPC プログラムのコンパイルと実行

インテル® ISPC を使用する

- 基本的なコマンドライン・オプション
- コンパイルターゲットの選択
- 32 ビットと 64 ビット・アドレス・モードの選択
- プリプロセッサー
- デバッグ
- インテル® ISPC へ引数を渡す別の方法

インテル® ISPC 並列実行モデル

- 基本概念: プログラム・インスタンスとプログラム・インスタンスのギャング
- ギャング内の制御フロー
  - 制御フローの例: if 文
  - 制御フローの例: ループ
  - ギャング収束の保証
- 統一データ
  - 統一制御フロー
  - 均一変数と制御フローの変化

- ギャング内のデータ競合
- タスクモデル

インテル® ISPC 言語

- C プログラミング言語との関係
- 字句構造
  - 整数リテラル
  - 浮動小数点リテラル
  - 文字列リテラル
- タイプ
  - 基本タイプとタイプ修飾子
  - uniform と varying 修飾子
  - タイプの新しい名前を定義
  - ポインタータイプ
  - 関数ポインタータイプ
  - 参照タイプ
  - 列挙タイプ
  - ショート・ベクトル・タイプ
  - 配列タイプ
  - 構造体タイプ
    - オペレーターのオーバーロード
  - 配列構造体タイプ
- 宣言と初期化子
- 式
  - 動的メモリー割り当て
  - タイプキャスト
- 制御フロー
  - 条件文: if
  - 条件文: switch
  - 反復文
    - 基本反復文: for、while、do
    - アクティブなプログラム・インスタンスの反復: foreach_active
    - 一意の要素に対する反復: foreach_unique
    - 並列反復文: foreach および foreach_tiled
    - programIndex と programCount による並列反復
  - 非構造化制御フロー: goto
  - Coherent 制御フロー文: cif とフレンド
  - 関数と関数呼び出し
    - 関数オーバーロード
  - 実行マスクの再確立
  - タスク並列実行
    - タスク並列: launch と sync 文
    - タスク並列: ランタイムの要件

- LLVM 組込み関数
- インテル® ISPC 標準ライブラリー
  - データの基本操作
    - 論理および選択操作
    - ビット操作
  - 数学関数
    - 基本数学関数
    - 超越関数
    - 飽和演算
    - 擬似乱数数値
  - 乱数生成
  - 出力関数
  - アサーション
  - コンパイラー最適化のヒント
  - プログラム間のインスタンス操作
    - リダクション
  - スタックメモリー割り当て
  - データ移動
    - メモリーの値設定とコピー
    - パックドロードとストア操作
    - ストリーミング・ロードとストア操作
  - データ変換
    - 構造体配列と配列構造体間のレイアウト変換
    - 半精度浮動小数点の変換
    - sRGB8 への変換
  - システム・プログラミングのサポート
    - アトミック操作とメモリーフェンス
    - プリフェッチ
    - システム情報
- アプリケーションとの相互運用性
  - 相互運用性の概要
  - データレイアウト
  - データのアライメントとエイリアス
  - インテル® ISPC を使用するための既存のプログラミングの再構築
- 製品および性能に関する情報

インテル® ISPC の最近の改良

コンパイラーに対する最近の変更点については、インテル® ISPC ディストリビューションの ReleaseNotes.txt (英語) ファイルを参照してください。

インテル® ISPC 1.1 における更新

インテル® ISPC 1.1 リリースで導入された主な変更点は、言語におけるポインターのファーストクラスのサポートと、新しい並列ループ構造です。この機能を実装するには、言語の構文をいくつか変更する必要がありました。これらの変更により、既存の ispc プログラムを若干変更することが求められました。

以下に関連する言語の変更点を示します。

- reference タイプの構文は C++ の reference 構文と一致するように変更されていますが、reference キーワードは削除されました。reference が行われていると診断メッセージが出力されます。
    - reference タイプの float foo などの宣言は、float &foo に変更する必要があります。
    - reference 修飾子を含む関数宣言の配列パラメーターは、すべて reference を削除する必要があります。例えば、void foo(reference float bar[]) は、単に void foo(float bar[]) とします。
- 配列全体を別の配列に代入するとコンパイルエラーが発生します。
- 大部分の標準ライブラリーが更新され、インデックス・オフセットを使用した参照や配列ではなく、ポインタータイプのパラメーターを受け入れるようになりました。例えば、atomic_add_global() 関数は、以前はアトミックに更新する変数への参照を渡していましたが、ポインターを使用するようになりました。同様に、packed_store_active() 関数は、第 1 パラメーターとして uniform unsigned int[]、第 2 パラメーターとして uniform int のオフセットを受け取るのではなく、第 1 パラメーターは uniform unsigned int へのポインターを受け取ります。
- 参照パラメーターを受け取る関数に varying lvalue タイプを渡すことは、許可されなくなりました。参照は uniform lvalue タイプのみが許可されます。この場合、可変長ポインター・パラメーターを受け取るように関数を変更する必要があります。
- 計算ドメイン、foreach および foreach_tiled をループする新しい反復構造があります。通常の for ループよりも構造がシンプルであることに加え、データを反復処理してプログラムのインスタンスにマッピングする際に、パフォーマンスの利点を得られる可能性が高くなります。これらの詳細は、「並列反復文: foreach と foreach_tiled」を参照してください。

インテル® ISPC 1.2 における更新

インテル® ISPC 1.2 のリリースでは、言語の構文、およびセマンティクスに次の変更が加えられました。

- launch キーワードの構文が整理され、起動する関数呼び出しを山括弧 (< >) で囲む必要がなくなりました。つまり、launch < foo() > ではなく launch foo() のようになります。

- ポインターを使用する際に、ポイントされるデータはデフォルトで uniform タイプになりました。必要に応じて、varying キーワードを使用してポイント先の各種データタイプを指定します。例えば、float *ptr は、均一 float データへの可変ポインターですが、これまでは可変長 float 値へのポインターでした。varying float * を使用して、可変長 float データへの可変ポインターを指定します。
- uniform と varying の詳細と、構造体タイプとの相互作用が整理されました。構造体タイプが宣言されている場合、構造体要素に明示的な uniform または varying 修飾子がない場合、それらは unbound (可変性) があるとみなされます。構造体タイプがインスタンス化されると、バインドされていない可変性要素は親構造体タイプの可変性を継承します。詳細は、「構造体タイプ」を参照してください。
- インテル® ISPC には、「(配列) 構造体配列 (AoSoA または SoA)」メモリーレイアウトのデータをはるかに効率良く使用する新しい言語機能が用意されています。新しい soa<n> 修飾子を構造体に適用して、対応するタイプの n 幅の SoA 配列を指定できます。配列インデックスと SoA タイプの配列を使用したポインター操作は、データにアクセスする際に 2 段階のインデックス計算が自動的に行われます。詳細は、「配列構造体タイプ」を参照してください。

インテル® ISPC 1.3 における更新

このリリースでは、多数の新しい反復構造が追加され、新しい予約語 unmasked、foreach_unique、foreach_active および in が定義されました。これらの名称と同じ変数や関数を持つプログラムは、その名前を変更する必要があります。

インテル® ISPC 1.5.0 における更新

このリリースでは、倍精度浮動小数点定数がサポートされました。倍精度浮動小数点定数は、サフィックス d とオプションの指数部を持つ浮動小数点数です (例: 3.14d、31.4d-1、1.d、1.0d、1d-2 など)。サフィックス d を持たない浮動小数点定数は、単精度の定数として扱われます。

インテル® ISPC 1.6.0 における更新

このリリースでは、オペレーターのオーバーロードがサポートされました。これによりオペレーターがキーワードとなったため、既存のユーザー関数と競合する可能性があります。また、新しい packed_store_active2() 関数が導入されました。これも既存のユーザー関数と競合する可能性があります。

インテル® ISPC 1.7.0 における更新

このリリースには、古いバージョンとの互換性に影響する可能性があるいくつかの変更が含まれています。

- オーバーロードされた関数を選択するアルゴリズムが拡張され、多種類のオーバーロードをカバーできるよう参照タイプが修正されました。同時に、すべての引数を合計した「ベストスコア」関数を使用する従来の方法から、各引数に「ベストスコア」を要求する方法に切り替わりました。最適な関数が存在しない場合、このバージョンは警告を発行します。次のバージョンではエラーになります。簡単な例として、max(int, int) と max(unsigned int, unsigned int) の 2 つの関数があると想定すると、新しい規則では、max(int, unsigned int) を呼び出すときにエラーになります。これは、最良の選択が曖昧であるためです。

- const タイプのポインターを暗黙的に void* へキャストできませんでした。必要であれば明示的にキャストします。
- 生成される .h ファイルで const 修飾子が表示されない問題が修正されました。これにより、出力される .h ファイルで const 修飾子が適切に表示されるようになったため、既存のコードでコンパイルエラーが発生する可能性があります。
- get_ProgramCount() は stdlib から examples/util/util.isph ファイルに移動されました。get_ProgramCount() 関数を使用するには、このファイルをインクルードする必要があります。

インテル® ISPC 1.8.2 における更新

このリリースには、古いバージョンとの互換性に影響する言語の変更は含まれていません。次の点に注意してください。

- uniform タイプのマングリングは可変幅を含まないように変更されたため、複数ターゲットのコンパイルで export 関数の戻りタイプとして uniform タイプへのポインターと uniform 構造体を利用できるようになりました。

インテル® ISPC 1.9.0 における更新

このリリースには、古いバージョンとの互換性に影響する言語の変更は含まれていません。新たにインテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) ターゲット (avx512knl-i32x16) が導入されました。

インテル® ISPC 1.9.1 における更新

このリリースには、古いバージョンとの互換性に影響する言語の変更は含まれていません。新たに インテル® AVX-512 ターゲット (avx512skx-i32x16) が導入されました。

インテル® ISPC 1.9.2 における更新

このリリースには、古いバージョンとの互換性に影響する言語の変更は含まれていません。

インテル® ISPC 1.10.0 における更新

このリリースにはいくつかの言語機能の変更が含まれていますが、互換性には影響しません。これには、新しいストリングストア、64 ビット・タイプの aos_to_soa/soa_to_aos 組込み関数、および #pragma ignore があります。

互換性に影響する変更の 1 つに、short vector タイプのサイズ変更があります。C/C++ とインテル® ISPC 間で short vector タイプをやり取りする場合、注意が必要です。

インテル® ISPC 1.11.0 における更新

このリリースでは、-O1 コンパイルオプションを再定義してサイズが最適化されたため、ビルドシステムの調整が必要になる場合があります。

バージョン 1.11.0 以降の自動生成ヘッダーは、#pragma once を使用します。まれに、C/C++ コンパイラーがサポートしない場合、--no-pragma-once ispc スイッチを使用します。

このリリースでは、新たにインテル® AVX-512 ターゲット (avx512skx-i32x8) が導入されました。これにより生成されるコードは、ZMM レジスターを使用しません。

インテル® ISPC 1.12.0 における更新

このリリースには、古いバージョンとの互換性に影響する次の変更が含まれています。

- noinline キーワードが追加されました。
- 標準ライブラリー関数 rsqrt_fast() と rcp_fast() が追加されました。
- インテル® AVX1.1 (開発コード名 IvyBridge) ターゲットと汎用の開発コード名 KNC および KNL ターゲットが削除されました。開発コード名 KNL のサポートは avx512knl-i32x16 によって継続されます。

このリリースでは、さまざまな変数の静的初期化が導入されていますが、互換性には影響しません。

このリリースには、実験的なクロス OS コンパイルのサポートと、ARM/AARC64 のサポートが導入されています。また、新しい 128 ビットのインテル® AVX2 ターゲット (avx2-i32x4) と開発コード名 Ice Lake クライアントの CPU 定義 (--device=icl) も含まれます。

インテル® ISPC 1.13.0 における更新

このリリースには、古いバージョンとの互換性に影響する次の変更が含まれています。

- bool タイプの表現が、ターゲット固有からブール値ごとの 1 バイトに変更されました。varying bool のサイズはターゲット幅 (バイト単位) であり、unform bool のサイズは 1 です。この定義は C/C++ と互換性があるため、相互運用性が向上します。
- 符号なしタイプのタイプエリアスが追加されました: uint8、uint16、uint32、uint64 および uint。このタイプがサポートされているか確認するには、ISPC_UINT_IS_DEFINED マクロが定義されているかどうかで判定できます。これは、古いバージョンのインテル® ISPC で動作するコードを記述する際に便利です。
- ブール値の引数向けの extract()/insert() と、すべての整数タイプと FP タイプの abs() が、標準ライブラリーに追加されました。

インテル® ISPC 1.14.0 における更新

このリリースには、古いバージョンとの互換性に影響する次の変更が含まれています。

- generic ターゲットが削除されました。代わりにネイティブターゲットを使用してください。

新たに i8 と i16 ターゲットが導入されました: avx2-i8x32、avx2-i16x16、avx512skx-i8x64 および avx512skx-i16x32。

Windows* x86_64 ターゲットで __vectorcall 呼び出し規約がサポートされるようになりました。デフォルトでは無効にされていますが、--vectorcall コマンドライン・スイッチで有効にできます。

インテル® ISPC 1.14.1 における更新

このリリースには、古いバージョンとの互換性に影響する言語の変更は含まれていません。

インテル® ISPC 1.15.0 における更新

このリリースにはいくつかの言語機能の変更が含まれていますが、互換性には影響しません。これには、64 ビット・タイプの packed_[load|store]_active() stdlib 関数と loop unroll プラグマ (#pragma unroll と #pragma nounroll) があります。

インテル® ISPC 1.16.0 における更新

このリリースには、標準ライブラリーにいくつかの新しい関数が追加されており、互換性に影響する可能性があります。

- alloca() - 詳細は、「スタックメモリー割り当て」を参照してください。
- assume() - 詳細は、「コンパイラー最適化のヒント」を参照してください。
- trunc() - 詳細は、「基本数学関数」を参照してください。

言語に LLVM 組込み関数を呼び出す実験的な機能が追加されました。これは、既存のプログラムとの互換性には影響しません。詳細は、「LLVM 組込み関数」を参照してください。

インテル® ISPC 1.17.0 における更新

このリリースでは、f16 サフィックス付きの新しいデータタイプ float16 と浮動小数点リテラルが導入されています。

C/C++ との統一性のため、大文字の X を 16 進プレフィクス (0X) で使用し、大文字の P を 16 進浮動小数点の指数の区切りとして使用できます。例: 0X1P16 など。

インテル® Xe のターゲット、アーキテクチャー、デバイス名の命名が変更されました。

標準ライブラリーに、書き込みを予測してプリフェッチを行う新しい prefetchw_{l1,l2,l3}() 組込み関数が追加されました。

rsqrt(double) と rcp(double) 標準ライブラリー関数の実装に使用されるアルゴリズムがインテル® AVX一-512 で変更されたため、既存のコードに影響する可能性があります。

インテル® ISPC 1.18.0 における更新

インテル® AVX-512 ターゲットで、「ベースタイプ」(または「マスクサイズ」) を含まないように名称が変更されました。互換性のため古い名前は利用できます。新しい名称は、avx512skx-x4、avx512skx-x8、avx512skx-x16、avx512skx-x32、avx512skx-x64 および avx512knl-x16 です。

標準ライブラリーが float16 タイプをサポートするようになりました。ただし、ネイティブ・ハードウェアで float16 がサポートされるターゲットでのみ完全にサポートされることに注意してください。ネイティブでサポートされないハードウェアでのエミュレーションはまだ完全に保証されていませんが、動作する可能性もあります。

コンパイラーは C/C++ コンパイラーと同様に、プリプロセッサーのみを実行する -E オプションを受け入れるようになりました。バグ修正の結果、プリプロセッサーでエラーが発生するとコンパイラーがクラッシュします。以前はクラッシュすることなく、何らかの出力が生成されました (適切な場合もあります)。これは、隔離環境で実行する一部のユーザーには便利な機能であったため (コンパイラー・エクスプローラー (英語) のコンパイル時に検出できないインクルード・ファイルを無視するなど)、この機能を維持するため --ignore-preprocessor-errors オプションが追加されました。

インテル® ISPC 入門

インテル® ISPC のインストール

インテル® ISPC ダウンロード・ページ (英語) には、Windows*、Linux* および macOS* 用のダウンロード可能なビルド済み実行ファイルがあります。もしくは、このページからソースコードをダウンロードして、自身でビルドすることもできます。ソースからインテル® ISPC をビルドする手順に関しては、インテル® ISPC Wiki (英語) を参照してください、

実行ファイルを取得した後、PATH にあるディレクトリーにコピーします。これでインテル® ISPC のインストールは完了です。

インテル® ISPC プログラムのコンパイルと実行

インテル® ISPC がインストールされているディレクトリーの examples/simple に簡単な C++ プログラムでインテル® ISPC を使用するサンプルが用意されています。simple.ispc ファイルを開いてください (以下に同じコードを示します)。

```
export void simple(uniform float vin[], uniform float vout[],
                   uniform int count) {
    foreach (index = 0 ... count) {
        float v = vin[index];
        if (v < 3.)
            v = v * v;
        else
            v = sqrt(v);
        vout[index] = v;
    }
}
```

このプログラムは、vin 配列の値をループ内で出力値の計算を行います。vin の各値が 3 より小さい (2 以下) の場合は値を 2 乗し、3 以上の場合は値の平方根を出力します。

このプログラムでは、関数定義に export キーワードを使用しています。これは、アプリケーション・コードからこの関数が呼び出されることを意味します。パラメーターの uniform 修飾子は、指定された変数が非ベクトル量であることを示します。この概念については、「uniform と varying 修飾子」の節で詳しく説明します。

foreach ループの各反復は、選択されたコンパイルターゲットによって、vin 配列の 4、8、16、32 あるいは 64 要素の入力値を並列に、かつ CPU または GPU の SIMD ハードウェアで効率良く処理します。ここで、変数 index は 0 から index-1 までのすべての値を取ります。配列から変数 v への読み取り後、プログラムは読み取られた値を判定して計算と制御フローを実行します。foreach ループの次の反復が実行される前に、実行中のプログラム・インスタンスの結果が vout 配列に書き込まれます。

サンプルをビルドして実行するには、examples フォルダーに移動して build フォルダーを作成します。そして、cmake -DISPC_EXECUTABLE=<path_to_ispc_binary> ../. を実行します。Linux* と macOS* では、makefile はそのディレクトリーに作成されます。Windows* では、Microsoft* Visual Studio* ソリューションに ispc_examples.sln が作成されます。これで準備ができました、ビルドを開始できます。コンパイル手順の詳細は、次の「インテル® ISPC コンパイラーを使用する」の節で説明します。インテル® ISPC プログラムのコンパイルに加えて、ここではインテル® ISPC コンパイラーはヘッダーファイル simple.h も生成します。このヘッダーファイルには、上記のインテル® ISPC プログラムがコンパイルされる C 呼び出し可能な関数宣言が含まれます。このファイルの関連部分は以下です。

```
#ifdef __cplusplus
extern "C" {
#endif // __cplusplus
    extern void simple(float vin[], float vout[], int32_t count);
#ifdef __cplusplus
}
#endif // __cplusplus
```

生成されたヘッダーファイルを C/C++ コードでインクルードするのは必須ではありませんが (代わりにインテル® ISPC 関数に手動で extern 宣言を使用できます)、関数のシグネチャーが双方で一致しているのを確認するのに役立ちます。

以下は、上記のインテル® ISPC 関数を呼び出すメインプログラム simple.cpp です。

```
#include <stdio.h>
#include "simple.h"

int main() {
    float vin[16], vout[16];
    for (int i = 0; i < 16; ++i)
        vin[i] = i;

    simple(vin, vout, 16);

    for (int i = 0; i < 16; ++i)
        printf("%d: simple(%f) = %f¥n", i, vin[i], vout[i]);
}
```

main() の中ほどにあるインテル® ISPC 関数呼び出しは、通常の関数呼び出しであることに注目してください。そして、C/C++ 関数呼び出しと同じオーバーヘッドがあります。

実行ファイル simple を実行すると、次の出力が表示されます。

```
0: simple(0.000000) = 0.000000
1: simple(1.000000) = 1.000000
2: simple(2.000000) = 4.000000
3: simple(3.000000) = 1.732051
...
```

インテル® ISPC を利用した少し複雑な例として、インテル® ISPC ウェブサイトにあるマンデルブロ集合 (英語) を参照して、アルゴリズムのインテル® ISPC 実装を確認してください。確認後、インテル® ISPC インストール・ディレクトリーの examples にある各種ソースコードの例をご覧になることをお勧めします。

インテル® ISPC を使用する

インテル® ISPC ソースファイルから、アプリケーション・コードにリンク可能なオブジェクト・ファイルを生成するには、次のコマンドを実行します。

```
ispc foo.ispc -o foo.o
```

Windows* では、出力するオブジェクト・ファイル名に foo.obj を指定します。

基本的なコマンドライン・オプション

インテル® ISPC を起動する際に --help オプションを指定すると、利用可能なコマンドライン引数のリストが表示されます。デフォルトでは、コンパイラーは指定されたプログラムファイルをコンパイルし、警告とエラーを示しますが、出力は生成しません。

-o フラグが指定されると、出力ファイルを生成します (デフォルトではネイティブ・オブジェクト・ファイル)。

```
ispc foo.ispc -o foo.obj
```

アセンブリー・ファイルを生成するには、--emit-asm を指定します。

```
ispc foo.ispc -o foo.s --emit-asm
```

LLVM ビットコードを生成するには、--emit-llvm を指定します。LLVM ビットコードをテキスト形式で生成するには、--emit-llvm-text を指定します。

プリプロセッサーのみを実行するには -E フラグを使用します。

```
ispc foo.ispc -E -o foo.i
ispc foo.ispc -E -o foo.ispi
```

このモードでは、出力ファイルが指定されていないと、出力は標準出力に送られます。標準のサフィックス .i または .ispi がプリプロセッサー出力に使用されます。

デフォルトでは、プリプロセッサーがエラーに遭遇するとコンパイルは失敗します。プリプロセッサーのエラーを無視してコンパイルを続行するには、--ignore-preprocessor-errors オプションを指定します。

最適化はデフォルトで有効になっていますが、無効にするには -O0 オプションを指定します。

```
ispc foo.ispc -o foo.obj -O0
```

コマンドラインで -g フラグを指定すると、デバッグシンボルの生成が有効になります。-g フラグは最適化には影響しません。最適化されていないコードをデバッグするには同時に -O0 を指定してください。

-h フラグは、C 呼び出し可能なインテル® ISPC 関数の C/C++ 宣言と引数タイプを含む C/C++を生成するようにインテル® ISPC に指示します。

-D オプションでプリプロセッサーに渡す定義を指定できます。プリプロセッサーは、コンパイル前にプログラム入力に対して実行されます。例えば、-DTEST=1 を指定すると、プリプロセッサー・シンボル TEST がプログラムのコンパイル時に値 1 を持つように定義できます。

コンパイラーは、非効率なコードにコンパイルされるコード構造に対し、パフォーマンスの警告を発行します。これらの警告は、--wno-perf フラグ (または、すべてのコンパイル警告をオフにする --woff) を使用して非表示にできます。さらに、--werror フラグを指定して警告をエラーとして処理するように指示できます。

--pic コマンドライン引数が指定されると、位置に依存しないコード (共有ライブラリーで使用) が生成されます。

コンパイルターゲットの選択

コンパイルターゲットに影響する 4 つのオプションがあります。--arch (ターゲット・アーキテクチャーを設定)、--device (ターゲット CPU/GPU を設定。--cpu とすることもできます)、--target (ターゲット命令セットを設定)、および --target-os (ターゲットのオペレーティング・システムを設定)。

これらのオプションのいずれも指定されていない場合、インテル® ISPC はホスト OS およびコンパイラーが実行されているシステムのアーキテクチャー (x86 システムでは、x86-64 (--arch=x86-64)、ARM システムでは ARM* Neon*) 向けのコードを生成します。

例えば、32 ビット x86 ターゲットにコンパイルするには、コマンドラインで --arch=x86 を指定します。

```
ispc foo.ispc -o foo.obj --arch=x86
```

インテル® Xe-LP プラットフォーム向けにコンパイルするには次を実行します。

```
ispc foo.ispc -o foo.bin --target=xelp-x16 --device=tgllp --emit-zebin
```

現在サポートされるアーキテクチャーは、x86、x86-64、xe32、xe64、arm および aarch64 です。

ターゲット CPU は、デフォルトの命令セットと、コードがチューニングされる CPU アーキテクチャーの両方を決定します。ispc --help オプションで、サポートされるすべての CPU のリストを表示します。デフォルトでは、インテル® ISPC を実行しているシステムの CPU タイプがターゲット CPU となります。

```
ispc foo.ispc -o foo.obj --device=corei7-avx
```

次に、--target でターゲットの命令セットを選択します。ハードウェアでマスキングがサポートされていないターゲットでは、ターゲットの文字列は、[ISA]-i[マスクサイズ]x[ギャングサイズ] の形式になります。例えば、--target=avx2-i32x16 は、インテル® AVX2 命令セット、32 ビットのマスクサイズ、および 64 ビットのギャングサイズでターゲットを指定します。ハードウェア・マスキングをサポートするターゲット (インテル® AVX-512 と GPU ターゲット) では、ターゲットの文字列は、[ISA]-x[ギャングサイズ] の形式になります。例えば、--target=xehpg-x16 は、ターゲット ISA にインテル® X^e-HPG およびギャングサイズ 16 を定義します。

デフォルトでは、ターゲット命令セットは、インテル® ISPC を実行しているシステムでサポートされる命令セットで最もスコアが高いものが選択されます。この場合、コンパイルに使用されたターゲットを示す情報が表示されます。ターゲットを明示的に指定するには、--target スイッチを使用することを推奨します。

サポートされるターゲットの完全なリストを表示するには、--help スイッチを使用して、--target の説明にあるリストに注目してください。また、--support-matrix スイッチを使用すると、サポートされるターゲット、アーキテクチャー、およびターゲット OS を組み合わせた完全なリストが表示されます。

次のターゲット ISA がサポートされます。

ターゲット	説明
avx、avx1	インテル® AVX (2010-2011 年代のインテル® CPU)
avx2	インテル® AVX2 ターゲット (2013 年代のインテル® CPU - 開発コード名 Haswell)
avx512knl	インテル® AVX-512 ターゲット (インテル® Xeon Phi™ チップ – 開発コード名 Knights Landing)
avx512skx	インテル® AVX-512 ターゲット (2013 年以降のインテル® Xeon® CPU)
neon	ARM* Neon*
sse2	インテル® SSE2 (2000 年代初頭の x86 CPU)
sse4	インテル® SSE4 (2008-2010 年代のインテル® CPU)
gen9	第 9 世代インテル® GPU (GEN9)
xehpg	インテル® X^e-HPG GPU
xelp	インテル® X^e-LP GPU

CPU がサポートするベクトル命令セットの詳細は、CPU のマニュアルを参照してください。

マスクサイズは、8、16、32 または 64 ビットですが、ISA とマスクサイズのすべての組み合わせがサポートされているわけではありません。最高のパフォーマンスを得るには、プログラムを構成する最も一般的なデータタイプのサイズに等しいマスクサイズを選択するのが最善です。例えば、ほとんどの計算が 32 ビット浮動小数点値を使用する場合、i32 ターゲットが最適です。8 ビット・データタイプの計算を行う場合、i8 のほうが適しています。

「ギャングサイズ」とプログラム実行への影響に関する詳細は、「基本概念: プログラム・インスタンスとプログラム・インスタンスのギャング」を参照してください。

コンパイルターゲットの命名規則は、2013 年 8 月に変更されています。次の表は、新旧スキームの命名の関係を示しています。

ターゲット	旧名
avx1-i32x8	avx、avx1
avx1-i32x16	avx-x2
avx2-i32x8	avx2
avx2-i32x16	avx2-x2
neon-8	n/a
neon-16	n/a
neon-32	n/a
sse2-i32x4	sse2
sse2-i32x8	sse2-x2
sse4-i32x4	sse4
sse4-i32x8	sse4-x2
sse4-i8x16	n/a
sse4-i16x8	n/a

最後に、--target-os でターゲット・オペレーティング・システムを選択します。ホストによっては、インテル® ISPC が Windows*、Linux*、macOS*、Android*、iOS* および PS4*/PS5* ターゲットをサポートする場合があります。ispc --help を実行して --target-os オプションの出力を見ると、サポートされるターゲットが分かります。デフォルトでインテル® ISPC は、ホスト・オペレーティング・システム向けのコードを生成します。

```
ispc foo.ispc -o foo.obj --target-os=android
```

クロス OS コンパイルは実験段階であることに注意してください。試行して皆さんの経験を送信するか、インテル® ISPC バグトラッカー (英語) にバグや機能要求を提出してください。

32 ビットと 64 ビット・アドレス・モードの選択

デフォルトでは、x86-64 のように 64 ビットのコンパイルターゲットを指定していても、インテル® ISPC はアドレス計算には 32 ビット演算を使用します。この実装アプローチは、計算のコストを抑えることで、パフォーマンスを大幅に向上できます。ポインター自体は、64 ビット・ターゲットの 64 ビット幅に維持されることに注意してください。

インテル® ISPC プログラムが 4GB を超えるメモリーアドレスを指定できるようにするには、コマンドライン引数 --addressing=64 を指定して、アドレス計算に 64 ビット計算を行うことをコンパイラーに指示します。デフォルトの --addressing=32 でコンパイルされたオブジェクト・ファイルと、--addressing=64 でコンパイルされたオブジェクト・ファイルを混在しても安全です。

プリプロセッサー

インテル® ISPC は、プログラムをコンパイルする前に、C プリプロセッサーを自動的に実行します。つまり、インテル® ISPC プログラムは #ifdef、#define などを使用できます。この機能は、--nocpp コマンドライン引数で無効にできます。

プリプロセッサーが実行される前に、多数のプリプロセッサー・シンボルが自動的に定義されます。

事前定義されたプリプロセッサー・シンボルと値

シンボル名	値	説明
ISPC	1	インテル® ISPC コンパイラーがファイルを処理しているのを検出できるようにします
ISPC_TARGET_{NEON, SSE2, SSE4, AVX, AVX2, AVX512KNL, AVX512SKX}	1	コンパイルターゲットに応じていずれかが設定されます
ISPC_POINTER_SIZE	32 または 64	ターゲット・アーキテクチャーのポインターを表現するビット数
ISPC_MAJOR_VERSION	1	インテル® ISPC コンパイラー/言語のメジャーバージョン
ISPC_MINOR_VERSION	13	インテル® ISPC コンパイラー/言語のマイナーバージョン
PI	3.1415926535	数学値
TARGET_WIDTH	ターゲットのベクトル幅 (例: sse2-i32x8 は 8)	静的に変化する初期化向けのコードのバージョンを管理できます
TARGET_ELEMENT_WIDTH	バイト単位で指定された要素の幅 (例:i32 では 4)	静的に変化する初期化向けのコードのバージョンを管理できます
ISPC_UINT_IS_DEFINED	1	インテル® ISPC で uint8/uint16/uint32/uint64 タイプが定義されている場合、このマクロが定義されます (1.13.0 以降)
ISPC_FP64_SUPPORTED	1	double タイプがターゲットでサポートされる場合、このマクロが定義されます
ISPC_LLVM_INTRINSICS_ENABLED	1	LLVM 組込み関数がサポートされる場合、このマクロが定義されます

インテル® ISPC は次の #pragma ディレクティブをサポートします。

#pragma ignore warning ディレクティブは、ソース行のコンパイル警告を無視するようコンパイラーに指示します。

#pragma ignore warning ディレクティブと機能	
#pragma 名	**説明**
#pragma ignore warning(all)	コード行のパフォーマンスの警告を含むすべてのインテル® ISPC コンパイラーの警告をオフにします。
#pragma ignore warning(perf)	コード行のパフォーマンスの警告をオフにします。
#pragma ignore warning	コード行のパフォーマンスの警告を含むすべてのインテル® ISPC コンパイラーの警告をオフにします。

マクロを呼び出す前に #pragma ignore warning を使用すると、展開されたマクロコードからの警告は非表示になります。

#pragma unroll と #pragma nounroll ディレクティブは、コンパイラーにループ展開の最適化のヒントを提供します。このプラグマはループ文の直前に配置します。現在このプラグマは、uniform for と do-while 構造のみをサポートします。

#pragma unroll と #pragma nounroll ディレクティブと機能	
#pragma 名	**使用**
#pragma unroll COUNT	ループを COUNT 回アンロールすることを指示します。パラメーターはオプションで括弧で囲むこともできます。 #pragma unroll (COUNT)
#pragma unroll	可能であれば、ループを完全にアンロールすることを指示します。
#pragma nounroll	ループをアンロールしないことを指示します。

デバッグ

-g コマンドライン・フラグを指定すると、コンパイラーはデバッグシンボルを生成します。デバッグ情報は、Linux* と macOS* では DWARF 形式で出力されます。DWARF のバージョンは、コマンドライン・オプション --dwarf-version={2,3,4} で指定できます。Windows* では CodeView 形式 (PDB ではなく) が使用され、Microsoft* Visual Studio* でネイティブサポートされます。デバッガーでインテル® ISPC プログラムを実行し、ブレークポイントを設定、変数を出力するのは、C/C++ プログラムのデバッグと全く同じです。同様に、インテル® ISPC コードと C/C++ コード間でコールスタックを直接ステップアップおよびダウンできます。

現在デバッグサポートに関連する制限の 1 つは、デバッガーが単一のプログラム・インスタンスではなく、ギャング全体のプログラムにウィンドウを提供することです。この概念は、「プログラム・インスタンスとプログラム・インスタンスのギャング」で紹介します。このように可変変数を表示すると、プログラム・インスタンスごとの値が表示されます。同様に、デバッガーがプログラム・ソースコードのフローをたどるパスは、任意のプログラム・インスタンスが実行した文になります (インテル® ISPC における制御フローの詳細は、「ギャング内の制御フロー」を参照してください)。

デバッグ中は、変数 __mask を使用して、プログラムの現時点の実行マスクを提供できます。

もう 1 つのデバッグ方法は、print 文を使用して printf() 形式のデバッグを行うことです (詳細は、「出力関数」を参照してください)。また、プログラム中の特定のポイントでアプリケーション・コードにコールバックして、必要な変数値を渡して、そこからログを解析する機能も利用できます。

インテル® ISPC へ引数を渡す別の方法

コマンドラインで引数を指定できますが、環境変数 ISPC_ARGS が設定されていると、それは引数に分割され、コマンドラインで指定された引数に追加されます。

また、インテル® ISPC に引数ファイルを渡すこともできます。コマンドラインで @<filename> 形式で引数ファイルを指定し、<filename> が存在して読み取り可能な場合、ファイル内の引数は分割されコマンドラインの引数に置き換えられます。ファイルには @<filename> 形式の引数も許可されます。

ファイルや環境変数が引数に分割される場合、引数がタブや改行を含む 1 つ以上の空白文字で区切られていることに基づいて行われます。引数に空白文字を挿入するために、文字をエスケープしたり引用する方法はありません。

インテル® ISPC 並列実行モデル

インテル® ISPC は C ベースの言語ですが、本質は並列計算向けの言語です。ここで説明するインテル® ISPC の並列実行モデルの詳細を理解することは、インテル® ISPC で効率良く正当なプログラムを作成する上で重要です。

インテル® ISPC は、複数のプロセッサー・コア間でのタスク並列による並列化と、単一コアの SIMD ベクトルレーン間で並列処理する SPMD 並列の 2 つの並列化をサポートしています。この節では SPMD による並列処理に注目しますが、インテル® ISPC におけるタスク並列処理については、この節の最後にある「タスクモデル」を参照してください。

ここでは、インテル® ISPC のコード例を使用してさまざまな概念を説明します。インテル® ISPC と C の関係を考慮すると、それ自体を理解することは自明ですが、言語構文の詳細は「インテル® ISPC 言語」を参照してください。

基本概念: プログラム・インスタンスとプログラム・インスタンスのギャング

C/C++ コードから呼び出されるインテル® ISPC 関数に制御が移行すると、実行モデルがアプリケーションのシリアルモデルからインテル® ISPC の実行モデルに切り替わります。概念的には、複数のインテル® ISPC プログラム・インスタンスが同時に実行を開始します。実行中のプログラム・インスタンスのグループはギャング (*gang*) と呼ばれます (インテル® ISPC は、ギャングで実行されるプログラム・インスタンスの制御フローの一貫性を保証するため、「ギャング・スケジュール」を連想させます。「ギャング収束の保証」で詳しく説明しています。インテル® ISPC のプログラム・インスタンスは、CUDA* の「スレッド」や OpenCL* の「work-item」に類似しており、インテル® ISPC のギャングは CUDA* の「warp」に似ています。

インテル® ISPC プログラムは「暗黙の並列実行」モデルを利用し、プログラム・インスタンスのギャングによって実行される計算を表現します。このモデルでは、インテル® ISPC プログラムは、通常単一のプログラム・インスタンスの動作を記述しますが、それらのギャングは同時に実行されます。この暗黙的モデルは、プログラム可能なグラフィックス・パイプライン、OpenCL* カーネル、および CUDA* のシェーダーで使用されるものと同じです。次のインテル® ISPC 関数の例について考えてみます。

```
float func(float a, float b) {
    return a + b / 2.;
}
```

C 言語では、この関数は 2 つの単精度浮動小数点値の単純な計算を表現します。インテル® ISPC では、この関数はギャング内の各プログラム・インスタンスによって実行される計算を表現します。各プログラム・インスタンスは、変数 a と b に異なる値を持つため、この関数を実行すると、プログラム・インスタンスは異なる結果を生成します。

プログラム・インスタンスのギャングは、インテル® ISPC 関数を呼び出したアプリケーション・コードと同じハードウェア・スレッドとコンテキストで実行を開始します。インテル® ISPC がスレッドを生成したり、コンテキストを切り替えることはありません。一連のプログラム・インスタンスは現在のプロセッサーの SIMD レーンにマッピングされるため、ハードウェア SIMD ユニットの高い利用率とパフォーマンスを達成できます。

ギャング内のプログラム・インスタンスの数は比較的少ないため、実際に実行するハードウェアのネイティブ SIMD 幅の 2 から 4 倍以下になります。したがって、4 レーンのインテル® SSE 命令セットを使用する CPU では 4 または 8 個のプログラム・インスタンスが、8 レーンのインテル® AVX/インテル® AVX2 を使用する CPU では 8 または 16 個、インテル® AVX-512 を実装する CPU では 8、16、32、64 個、またはインテル® GPU では 8 または 16 個のギャングが実行されます。

ギャング内の制御フロー

インテル® ISPC は、ほぼすべての標準的な制御フロー構造をサポートしています。プログラム・インスタンスは、ギャング内のほかのインスタンスと異なるプログラムの実行パスを自由にたどることができます。インテル® ISPC コードの if 文の例について考えてみます。

```
float x = ..., y = ...;
if (x < y) {
   // true 文
}
else {
   // false 文
}
```

一般に、x < y の評価は、ギャング内のプログラム・インスタンスごとに異なる結果をもたらす可能性があります。実行中のプログラム・インスタンスの一部が true の文を実行することもあれば、false の文を実行することもあります。

インテル® ISPC プログラムの複雑な制御フローは、期待どおりに動作し、C 言語で記述される同等のコードがシリアルに実行され、各プログラム・インスタンスの結果を個別に計算した場合と同じように、ギャング内の各プログラム・インスタンスに対し計算します。ただし、ここでは特定の状況下で言語の動作を明確に定義できるよう、制御フローの実行モデルをより正確に定義します。

プログラムカウンターの概念と、プログラムをステップ実行するカウンターの更新方法、および現在のプログラムカウンターで命令を実行するプログラム・インスタンスを示す実行マスクを指定します。プログラムカウンターは、ギャング内のすべてのプログラム・インスタンスによって共有され、次に実行される単一の命令を指します。実行マスクは、現在実行されている

命令が各プログラム・インスタンスにどのように作用するかを示す、プログラム・インスタンスごとのブール値です。つまり、命令文が「すべてオフ」のマスクで実行される場合、その結果は何も反映されません。

アプリケーションによって呼び出されたインテル® ISPC 関数が実行されると、実行マスクは「すべてオン」になり、プログラムカウンターは関数の最初の文の命令を指します。次の 2 つの文は、インテル® ISPC 関数の実行中に必要なプログラムカウンターと実行マスクの動作を示します。

1. プログラムカウンターは関数の保守的な実行パスに対応する一連の値を保持し、プログラム・インスタンスが命令文を実行する場合、プログラムカウンターはその命令文を通過します。
2. プログラムカウンターが通過するそれぞれの命令文で、プログラム・インスタンスがその命令文を実行する場合にのみ、特定のプログラム・インスタンスに対する値が「オン」になるように実行マスクが設定されます。

これらの定義は、コンパイラーにある程度の自由度を与えることに注意してください。例えば、プログラムカウンターは、実行マスクを「すべてオフ」にして一連の命令文を通過できます。これは、目に見える影響がないためです。

非公式ですが、プログラムの制御フローの一貫性については別の場所で説明します。この概念は、ギャング内のプログラム・インスタンスが、ある関数で同じ制御フローをたどる割合を示します (命令文が「ほとんどオン」の実行マスク、または「ほとんどオフ」の実行マスクで実行されるかどうか)。一般に、制御フローの分岐は、異なるプログラム・インスタンスが異なる計算を実行するため、SIMD 効率の低下 (そして、パフォーマンスの低下) につながります。

制御フローの例: if 文

プログラムカウンターと実行マスク間の相互作用を示す具体例として、前節の if 文を表現する、次の疑似コードのコンパイラーの出力を考えてみます。

```
float x = ..., y = ...;
bool test = (x < y);
mask originalMask = get_current_mask();
set_mask(originalMask & test);
if (any_mask_entries_are_enabled()) {
  // true 文
}
set_mask(originalMask & ~test);
if (any_mask_entries_are_enabled()) {
  // false 文
}
set_mask(originalMask);
```

言い換えれば、プログラムカウンターは、true のケースと false のケースの両方の文をステップ実行し、true 文の副作用が false 文を実行するプログラム・インスタンスに影響しないよう実行マスクを設定します (その逆も同様です)。ただし、命令文のブロックは、そのブロックの入り口でマスクが「すべてオフ」になっている場合、実行されません。その後、実行マスクは if 文の直前の値にリストアされます。

制御フローの例: ループ

for、while、do 文は同様に扱われます。プログラムカウンターは、すべてのプログラム・インスタンスがループを抜けるまで、ループを繰り返し実行します。

次のようなループについて考えてみます。

```
int limit = ...;
for (int i = 0; i < limit; ++i) {
    ... ループ本体
}
```

ここで、limit は 1 つを除くすべてのプログラム・インスタンスで値 1 であり、ほかのインスタンスで値 1000 である場合、プログラムカウンターはループ本体を 1000 回通過します。最初は実行マスクがすべてオンになり (すべて for ループに入ると仮定)、残りの 999 回は上限値が 1000 のプログラム・インスタンスを除いてマスクがオフになります。これは、制御フローの一貫性が低いループです。

ループ内の continue 文は、continue 文を実行するプログラム・インスタンスの実行マスクを無効にしてから、ループの残りでプログラムカウンターをステップ続行するか、continue 実行後にすべてのプログラム・インスタンスが無効であれば、ループステップ文にジャンプして処理されます。break 文も同様に扱われます。

ギャング収束の保証

インテル® ISPC 実行モデルは、プログラムカウンターと実行マスクの動作において、プログラム・インスタンスの実行が最大に収束することを保証します。最大収束とは、2 つのプログラム・インスタンスが同じ制御パスをたどる場合、それぞれのプログラム文の同時実行が保証されることを意味します。2 つのプログラム・インスタンスが分岐する制御パスをたどる場合、関数内で可能な限り速く再収束することが保証されます (後に再収束する場合)。[1]

[1] これも、このような保証がない OpenCL* や CUDA* の実装とインテル® ISPC 実行モデルの大きな違いです。

最大収束は、次のように発散的な制御フローが存在する場合、収束することを意味します。

```
if (test) {
  // true
}
else {
  // false
}
```

if 文の評価の前に実行されていたすべてのプログラム・インスタンスは、else ブロックの終了後も実行されることが保証されます。この保証は、各プログラム・インスタンスに固有のプログラムカウンターという概念ではなく、プログラム・インスタンスのギャングに対して単一のプログラムカウンターを持つという概念に由来します。

この特性のもう 1 つの意味は、インテル® ISPC 実装が 8 レーンのギャングを持つ関数を 2 回実行することは違法であり、4 レーンのギャングが毎回 8 レーンのギャングの半分に相当するということです。

次のプログラムを考えてみます。

```
if (programIndex == 0) {
    while (true)  // 無限ループ
        ;
}
print("hello, world¥n");
```

ここでは、プログラムは無限ループして、print 文が実行されることはありません。ギャングの分岐を許容する実行モデルでは、上記の例ですべてのプログラム・インスタンスが無限ループに陥るわけではないため、print 文が実行される可能性があります。

インテル® ISPC における「変化する」関数ポインターもこの影響を受けます。関数ポインターが変化する場合、実行中のすべてのプログラム・インスタンスで異なる値を持つ可能性があります。変化するポインターが呼び出されると、インテル® ISPC は可能な限り実行の収束性を維持する必要があります。生成されたアセンブリー・コードは、実行中のプログラム・インスタンスに一意な関数ポインターの集合を検出し、それぞれを一度だけ呼び出します。これにより、呼び出された際に実行中のプログラム・インスタンスが、その関数ポインター値を持つアクティブなプログラム・インスタンスのセットになります。この場合、変化する関数ポインターが呼び出される順番は定義されていません。

統一データ

uniform 修飾子で宣言された変数は、ギャング全体で共有される単一の値を定義します。対照的に、インテル® ISPC 変数のデフォルト可変修飾子である varying は、ギャング内の各プログラム・インスタンスの個別の格納場所を保持する変数を定義します。構造体で使用する際の uniform と varying における相違点については、「構造体タイプ」の説明を参照してください。

uniform 変数に uniform 値を割り当てることはできますが、uniform 変数に varying 値を割り当てるとエラーになります。uniform 変数への割り当ては実行マスクの影響を受けず、プログラムカウンターのポインターが uniform 代入である文を通過する際に常に適用されます。

統一制御フロー

ギャング全体で共有される変数を uniform として宣言する利点は、ストレージ空間を削減できることです。制御フローを決定する評価条件が均一な (uniform) データに基づいている場合、コンパイラーはその時点で実行中のプログラム・インスタンスがすべて同じフローをたどることを判断でき、制御フローの発散とマスク管理に対処するオーバーヘッドを軽減できます。2 つの制御フロー形式を区別するため、変化する表現式に基づく制御フローを varying 制御フローと呼びます。

例えば、プログラムが指定された (x、y) 座標に隣接するピクセルをループ処理するイメージフィルター操作を考えてみます。

```
float box3x3(uniform float image[32][32], int x, int y) {
    float sum = 0;
    for (int dy = -1; dy <= 1; ++dy)
        for (int dx = -1; dx <= 1; ++dx)
            sum += image[y+dy][x+dx];
    return sum / 9.;
}
```

通常、ギャングのそれぞれのプログラム・インスタンスは、この関数の x と y に異なる値を保持します。ボックスフィルター処理アルゴリズムでは、すべてのプログラム・インスタンスが同じ回数だけ for ループを反復する必要があり、すべてのループが毎回同じ dx と dy の値を持ちます。これらのループで dx と dy が均一な変数として宣言されている場合、インテル® ISPC はさらに効率良いループのコードを生成できます。[2]

[2] この場合、優秀な最適化コンパイラーは、dx と dy がすべてのプログラム・インスタンスに対して同じ値を持つと判断し、最適化されたコードを生成できますが、この最適化はまだインテル® ISPC では実装されていません。

```
for (uniform int dy = -1; dy <= 1; ++dy)
    for (uniform int dx = -1; dx <= 1; ++dx)
        sum += image[y+dy][x+dx];
```

特に、インテル® ISPC は、実行中のプログラム・インスタンスのいずれかが別のループを実行するかどうかを確認するオーバーヘッドを回避できます。代わりに、コンパイラーはすべてのインスタンスが常に同一反復を行うコードを生成できます。

if 文を使用しても同様な利点が得られます。if 文の評価が均一テストで行われる場合、実行中のすべてのプログラム・インスタンスに対し、結果は定義上同一になります。インテル® ISPC は、どちらか一方にジャンプするコードを生成すればよく、両方のケースに対処するコードを実行する必要がないため、オーバーヘッドを回避できます。

均一変数と制御フローの変化

制御フローが変化する場合、if 文の true と false 節の両方が実行され、命令の影響は対応する節を実行するプログラム・インスタンスのみに適用されるようマスクされることを思い出してください。このモデルでは、変化する制御フローのコンテキスト内で均一な変数を変更した場合の影響を定義する必要があります。

一般に、変化する制御フローで均一変数を変更すると、均一変数はギャング内のプログラム・インスタンスのいずれかが特定の実行パスを通過したかどうかに依存する値を持つようになります。次の例について考えてみます。

```
float a = ...;
uniform int b = 0;
if (a == 0) {
    ++b;
    // b は 1
}
else {
    b = 10;
    // b は 10
}
// b が 1 か 10 かは、実行中のギャングの a の値のいずれかが、
// 0 であるかどうかに依存します。
```

ここで、ギャングにまたがる a の値のいずれかが 0 でない場合、if 文実行後の b の値は 10 になります。ただし、if 文の実行を開始した時点で、実行中のプログラム・インスタンスの a の値がすべて 0 である場合、b の値は 1 になります。

ギャング内のデータ競合

プログラム・インスタンスが、ギャング内のほかのプログラム・インスタンスがメモリーに書き込んだ値に依存するようなプログラムを記述するには、あるプログラム・インスタンスの作用が、同じギャング内で動作するほかのプログラム・インスタンスで参照できるタイミングを明確に定義する必要があります。

インテル® ISPC が実装するモデルでは、あるプログラム・インスタンスからの作用の影響は、プログラム内の次のシーケンスポイント以降でギャング内のほかのプログラム・インスタンスから参照できるようになります。[3]

[3] これは、インテル® ISPC と OpenCL* や CUDA* などの SPMD 言語との大きな違いであり、この条件を満たすため barrier() や __syncthreads() などの関数で実行中のプログラム・インスタンス間の同期を取る必要があります。

一般に、シーケンスポイントには、式の完全な終端、関数呼び出しで関数を呼び出す前、関数からのリターン位置、初期化子式の終端などがあります。i のインクリメントと i = i++ の i への代入の間にシーケンスポイントが存在しないことが、C 言語でその式の効果が定義されない理由です。C や C++ のシーケンスポイントの詳細は、シーケンスポイントの Wikipedia ページ (英語) を参照してください。

次の例では、実行中のプログラム・インスタンスごとに 1 つの値を持つ、y の配列を宣言しています。以下の例では、programCount がギャングサイズを示し、整数値 programIndex がゼロから始まる実行中のプログラム・インスタンスにインデックスを提供すると仮定しています。8 つのプログラム・インスタンスが実行されている場合、最初のインスタンスの値は 0、次のインスタンスの値は 1、のように 7 まで続きます。

```
int x = ...;
uniform int tmp[programCount];
tmp[programIndex] = x;
int neighbor = tmp[(programIndex+1)%programCount];
```

このコードでは、実行中のプログラム・インスタンスは、配列 tmp の i 番目の要素が i 番目のプログラム・インスタンスの x の値と等しくなるように、x の値を tmp 配列に書き込みます。次に、プログラム・インスタンスは tmp から neighbor の値をロードし、隣接するプログラム・インスタンスによって書き込まれた値にアクセスします (最後に最初のインスタンスに戻ります)。このコードでは、tmp への書き込みと読み取りがシーケンスポイントで分離されているため、データ競合がありません。

特定のプログラム・インスタンスから別のプログラム・インスタンスへ値を伝達するこのアプリケーションでは、インテル® ISPC の標準ライブラリーに、さらに効率的な組込み関数があります。詳細は、「プログラム間のインスタンス操作」を参照してください。

プログラム・インスタンスのギャング間でデータ競合が発生するコードを記述できます。例えば、次の関数が同じ値の index を持つ複数のプログラム・インスタンスから呼び出された場合、そのいずれかが value 値を array[index] を書き込むかは未定義です。

```
void assign(uniform int array[], int index, int value) {
    array[index] = value;
}
```

また別の例として、配列インデックス i と j の値が一部のプログラム・インスタンスで同じ値を持つ場合、次のような割り当てが行われます。

```
int i = ..., j = ...;
uniform int array[...] = { ... };
array[i] = array[j];
```

同じ場所への読み取りと書き込みの間にシーケンスポイントがないため、プログラムの動作は未定義です。

プログラム・インスタンスは、ギャング内のほかのプログラム・インスタンスによる作用に安全に依存できるというこの規則は、ほかの SPMD 言語で必要なクラスの同期を排除できますが、逆に、異なるギャングサイズで異なる結果を計算するインテル® ISPC プログラムを記述することが可能であることを意味します。

タスクモデル

インテル® ISPC は launch キーワードによる非同期関数呼び出し (タスク処理) メカニズムを提供します (構文は、「タスク並列: launch と sync 文」の節で説明します)。launch で呼び出された関数は、呼び出した関数とは非同期に実行されます。つまり、すぐに実行することも、システム内の別のプロセッサーで同時に実行することもできます。

関数が複数のタスクを起動する場合、タスクが実行される順番に関する保証はありません。さらに、1 つの関数から起動された複数のタスクが同時に実行されることもあります。

タスクを起動した関数は、sync キーワードを使用して、起動された関数との同期を強制できます。同期が行われると、関数は、起動したすべてのタスクが完了するのを待機してから、後続の処理を実行します。sync は、現在の関数によって起動されたタスクのみを待機し、ほかの関数が起動したタスクを待機しないことに注意してください。

タスクを起動した関数がリターンすると、暗黙の同期により起動したすべてのタスクが完了するのを待機してから、呼び出し元の関数にリターンします。この機能は並列合成を行う上で重要です。関数は、2 番目の関数が非同期タスクを起動したか

どうかにかかわりなく、2 番目の関数を呼び出すことができます。どちらの場合も、2 番目の関数がリターンすると、最初の関数はすべての計算が完了したことが保証されます。

インテル® ISPC 言語

インテル® ISPC は、C プログラミング言語の拡張バージョンであり、CPU および GPU 向けのハイパフォーマンス SPMD プログラミングを容易に作成することを可能にする多くの新機能を提供します。インテル® ISPC と C コード間にはわずかな言語構文上の違いがあるだけでなく、インテル® ISPC には並列実行モデルが備わっているため、C コードを再コンパイルしてもインテル® ISPC で正しく並列実行することはできないことに注意してください。正しく動作する C コードをインテル® ISPC に移植するのは、インテル® ISPC プログラムを素早く記述する効率的な方法であるといえます。

ここでは、インテル® ISPC 言語の構文とセマンティクスについて説明します。インテル® ISPC の使い方を理解するには、言語構文と前節の「インテル® ISPC 並列実行モデル」の両方を理解する必要があります。

C プログラミング言語との関係

ここでは、インテル® ISPC と C 言語の違いをまとめています。すでに C 言語に精通している開発者はこの項に注目して、残りの節では新しい言語機能を紹介するトピックだけに注目すると良いでしょう。また、インテル® ISPC の examples/ ディレクトリーにあるアルゴリズムの例でインテル® ISPC と C++ の実装を比較することで、インテル® ISPC と C 言語の関係を理解できるでしょう。

この項の説明は、C89 との比較をベースにしています (これは、カーニハンとリッチーの書籍の第 2 版で説明される C バージョンです)。インテル® ISPC は C99 や C++ からの機能を継承していますが、それらは後述します。

インテル® ISPC は C 言語と同様な次の構文と機能を備えています。

- 式の構文と基本タイプ
- 変数宣言の構文
- if、for、while、do、switch などの制御フロー構造
- 関数ポインター、void*、C 言語の配列とポインターの二重性を含むポインター (配列は、関数への引数としてポインターに変換されます)
- 構造体と配列
- 関数の再起呼び出しのサポート
- ソースファイルの分割コンパイルのサポート
- ||、&&、? : オペレーターによる短絡評価
- プリプロセッサー

インテル® ISPC は C++ と C99 の多くの機能を継承します。

- ブールタイプである bool と、ビルトイン true と false 値
- 参照タイプ (const float &foo など)
- // によるコメントの区切り
- 変数はブロックの先頭だけではなく、ブロック内のどこにでも宣言可能
- for ループの反復変数は、for 文で宣言可能 (例: for (int i = 0;...))

- inline 修飾子は関数がインライン展開されることを示す
- パラメーター・タイプによる関数のオーバーロード
- 16 進浮動小数点定数
- new と delete による動的なメモリー割り当てと解放
- オーバーロードされたオペレーター (オペレーターのオーバーロード) には制限がある

インテル® ISPC では、C89、C99、C++ にはない新しい機能も追加されています。

- foreach と foreach_tiled 並列反復構造 (「並列反復文: foreach および foreach_tiled」を参照)
- ギャング内のプログラム・インスタンスのサブセットに対し反復処理を行う foreach_active と foreach_unique 反復構造 (「アクティブなプログラム・インスタンスの反復: foreach_active」と「一意の要素に対する反復: foreach_unique」を参照)
- タスク並列の言語サポート (「タスク並列実行」の節を参照)
- 実行中のプログラム・インスタンス本体で制御フローの一貫性があることを示す Coherent 制御フロー文 (「Coherent 制御フロー文: cif とフレンド」を参照)
- 豊富な標準ライブラリー (C 言語とは異なる。「インテル® ISPC 標準ライブラリー」を参照)
- ショート・ベクトル・タイプ (「ショート・ベクトル・タイプ」を参照)
- 整数定数をビットベクトルとして指定する構文 (0b1100 は 12 など)

現在インテル® ISPC ではサポートされていませんが、今後サポートされる可能性の高い C89 の機能があります。

- char、short、long (または long double) タイプはなく、int8、int16、int64 ビルトインタイプはある
- 文字定数
- 文字列定数と文字列の配列
- goto 文は部分的にサポートされます (「非構造化制御フロー: goto」を参照)
- union タイプ
- 構造体タイプのビットフィールド・メンバー
- 可変長引数の関数
- リテラル浮動小数点定数 (サフィックスが f でない場合も) は double ではなく float として扱われ、倍精度浮動小数点定数にはサフィックス d を指定する
- volatile 修飾子
- 変数の register ストレージクラス (無視されます)

次の C89 機能は、今後のインテル® ISPC ではサポートされません。

- K&R スタイルの関数宣言
- C 標準ライブラリー
- 8 進整数定数

以下の C89 の予約語はインテル® ISPC でも予約語となっています。

break、case、const、continue、default、do、double、else、enum、extern、float、for、goto、if、int、NULL、return、signed、sizeof、static、struct、switch、typedef、unsigned、void および while。

インテル® ISPC ではさらに以下の予約語を持ちます。

bool、delete、export、cdo、cfor、cif、cwhile、false、float16、foreach、foreach_active、foreach_tiled、foreach_unique、in、inline、noinline、__vectorcall、int8、int16、int32、int64、launch、new、print、uint8、uint16、uint32、uint64、soa、sync、task、true、uniform および varying。

字句構造

インテル® ISPC のトークンは空白とコメントで区切られます。空白文字は、スペース、タブ、キャリッジリターン (CR)/ラインフィード (LF) です。コメントは // で区切り、行末まで続くコメントを示します。また、コメントの開始を /*、終了を */ で区切ることもできます。C/C++ のように、コメントを入れ子にすることはできません。

インテル® ISPC の識別子は、アンダースコア、大文字または小文字で始まり、その後に 0 個以上の文字、数字、およびアンダースコアが続く文字列です。2 つのアンダースコアで始まる識別子は、コンパイラー向けに予約されています。

整数リテラル

整数数値定数は、10 進、16 進、または 2 進数で指定できます。8 進整数定数はサポートされていません。10 進数の定数は、0 から 9 までの 1 つ以上の数字で指定されます。16 進数の定数は、先頭が 0x または 0X で始まり、その後に 0 から 9、a から f、または A から F の 1 つ以上の数字が続きます。バイナリー定数は 0b で始まり、その後に 1 と 0 のシーケンスが続きます。

整数値 "15"を指定するには、次の 3 つの方法があります。

```
int fifteen_decimal = 15;
int fifteen_hex     = 0xf;
int fifteen_binary  = 0b1111;
```

整数の数値定数では、いくつかのサフィックスを指定できます。u で始まる定数は、定数が符号なしであることを示し、ll は 64 ビット整数定数を示します (l は 32 ビット整数定数を示します)。1024、1024*1024 または 1024*1024*1024 の単位は、それぞれサフィックス k、M、および G で表現できます。

```
int two_kb = 2k;   // 2048
int two_megs = 2M; // 2 * 1024 * 1024
int one_gig = 1G;  // 1024 * 1024 * 1024
```

浮動小数点リテラル

インテル® ISPC は、float16、float および double の 3 つの浮動小数点タイプをサポートします。

- float16 は、IEEE 754 半精度 (16 ビット形式) 浮動小数点タイプです。
- float は、IEEE 754 単精度 (32 ビット形式) 浮動小数点タイプです。
- double は、IEEE 754 倍精度 (64 ビット形式) 浮動小数点タイプです。

これら 3 つのタイプの浮動小数点定数は、次のいずれかの方法で指定できます。

- 基数セパレーター付きの 10 進浮動小数点 - 0 から 9 桁以上のゼロシーケンス後に、ピリオドを持ち、以降にゼロ以上の 0 から 9 桁が続くシーケンス。ピリオドの前後には、少なくとも 1 桁の数字が必要です。浮動小数点サフィックスが指定される場合、基数セパレーターはオプションです。
- 指数表記 - 10 進数の基数に e または E、次に任意のプラス/マイナス記号、その後に 10 進数の指数を指定します。
- 16 進浮動小数点定数 - 特定の浮動小数点数をビット精度で表現したものです。0x または 0X プリフィクスで始まり、0 または 1、ピリオド、および仮数の残りを 16 進数で 0～9、a～f、または A～F の数字で表現します。指数部の始まりは、p または P で表わし、その後にオプションでプラス/マイナス記号を記述し、0～9 の数字で指数部の 10 進値を表します。16 進浮動小数点リテラルでは、指数は必須です。

浮動小数点数リテラルのデフォルトタイプは float です。浮動小数点リテラルは、次のサフィックスのいずれかを使用して表現できます。

オペレーター	
サフィックス	**タイプ**
f16 または F16	float16
f または F	float
d または D	double

例:

float タイプの浮動小数点リテラル。

```
float16 two_f16   = 2.0f16;                 // 2.0
float16 pi_f16    = 0x1.92p+1f16;           // 3.1406
float16 neg_f16   = -65520.f16;             // -Inf
float   two_f     = 0x1p+1;                 // 2.0
float   pi_f      = 0x1.921fb6p+1;          // 3.14159274
float   neg_f     = -0x1.ffep+11;           // -4095.0
double  two_d     = 2.0d;                   // 2.0
double  pi_d      = 0x1.921fb54442d18p+1d;  // 3.1415926535897931
double  neg_d     = -0.3333333333333333d;   // -1/3
```

また、e の代わりに d または D をリテラルに使用した指数表記である Fortran double 形式も許可されます。この表記は、倍精度浮動小数点リテラルを表現します。

```
double d1 = 1.234d+3;  // 1234.0d
double d2 = 1.234e+3d; // 1234.0d
```

文字列リテラル

インテル® ISPC における文字列定数は、最初の二重引用符 (") で始まり、改行以外の任意の文字から最後の二重引用符までで表現されます。文字列内では、特殊エスケープシーケンスを使用して特殊文字を表現できます。これらのシーケンスはすべて \ で始まります。

文字列中のエスケープシーケンス。	
\\	バックスラッシュ\
\"	二重引用符: "
\'	一重引用符: '
\a	ベル (アラート)
\b	バックスペース文字
\f	改ページ文字
\n	改行
\r	キャリッジリターン
\t	水平タブ
\v	垂直タブ
\ に 1 つ以上の 0～8 の数字が続く	8 進表記の ASCII 文字
\x に 1 つ以上の 0～9、a～f、A～F の数字が続く	16 進表記の ASCII 文字

インテル® ISPC は文字列データタイプをサポートしていませんが、文字列定数は print() 文の最初の引数として使用できます。

次の識別子が言語キーワードとして予約されています: bool、break、case、cdo、cfor、char、cif、cwhile、const、continue、default、do、double、else、enum、export、extern、false、float、float16、for、foreach、foreach_active、foreach_tiled、foreach_unique、goto、if、in、inline、noinline、int、int8、int16、int32、int64、launch、NULL、print、return、signed、sizeof、soa、static、struct、switch、sync、task、true、typedef、uint、uint8、uint16、uint32、uint64、uniform、union、unsigned、varying、__vectorcall、void、volatile および while。

インテル® ISPC では、次のオペレーターとセパレーターを定義します。

オペレーター	
シンボル	**使用**
=	代入
+、-、*、/、%	算術演算子
&、|、^、!、~、&&、||、<<、>>	論理およびビット単位操作
++、--	プリ/ポストのインクリメント/デクリメント
<、<=、>、>=、==、!=	リレーショナル・オペレーター
*=、/=、+=、-=、<<=、>>=、&=、|=	複合代入オペレーター
?、:	選択オペレーター
;	文セパレーター
,	式セパレーター
.	メンバーアクセス

インテル® ISPC のグループ化には、いくつかのトークンが使用されます。

グループ化トークン	
(,)	式の括弧書き、関数呼び出し、制御フロー構造の区切り指定子。
[,]	配列とショートベクトルの添え字
{, }	複合文

タイプ

基本タイプとタイプ修飾子

インテル® ISPC は静的なタイプ付け言語です。各種コア基本タイプに対応します。

- void: 値がないことを表わす「空」タイプ
- bool: ブール値。true、false またはブール式の値を割り当てできます。
- int8: 8 ビット符号付き整数。
- unsigned int8: 8 ビット符号なし整数。uint8 としても指定できます。
- int16: 16 ビット符号付き整数。
- unsigned int16: 16 ビット符号なし整数。uint16 としても指定できます。
- int: 32 ビット符号付き整数。int32 としても指定できます。
- unsigned int: 32 ビット符号なし整数。unsigned int32、uint32、または uint としても指定できます。
- int64: 64 ビット符号付き整数。
- unsigned int64: 64 ビット符号なし整数。uint64 としても指定できます。
- float16: 16 ビット浮動小数点値
- float: 32 ビット浮動小数点値

- double: 64 ビット倍精度浮動小数点値。

ポインターとメモリーに関連するビルトインタイプも用意されています。

- size_t: オブジェクト (構造体や配列) の最大サイズ。
- ptrdiff_t: 2 つのポインターの差を表現できる十分な範囲表現をもつ整数タイプ。
- intptr_t: ポインター値を表現するのに十分な範囲表現を持つ符号付き整数タイプ。
- uintptr_t: ポインター値を表現するのに十分な範囲表現を持つ符号なし整数タイプ。

異なるタイプ間の暗黙のタイプ変換は、インテル® ISPC によって自動的に行われます。そのため、float タイプの値を int タイプの変数に直接割り当てることができます。タイプが混在するバイナリー算術式では、タイプは次の優先順位に基づいて 2 つのタイプのうち「より汎用的な」タイプに昇格されます。

```
double > uint64 > int64 > float > uint32 > int32 >
    float16 > uint16 > int16 > uint8 > int8 > bool
```

つまり、int64 を double に加算すると、int64 が double に変換されて加算され、double の値が返されます。異なる変換操作が必要な場合、明示的にタイプキャストを行うことができます。変換先のタイプは括弧で囲みます。

```
double foo = 1. / 3.;
int bar = (float)bar + (float)bar;  // 32 ビット単精度加算
```

bool が整数の数値タイプ (int、int64 など) に変換されると、bool 値が true である場合、結果はゼロ以外の値になり、それ以外の値は 0 になります。bool 値が true である場合、整数の数値タイプに変換される保証はありません。

const 修飾子を使用して変数を宣言することで変更を禁止できます。

```
const float PI = 3.1415926535;
```

C 言語と同様に、extern 修飾子を使用して、異なるソースファイルで定義される関数やグローバル変数を参照できます。また、static 修飾子は、現在のスコープ (ファイル内) でのみ参照可能な変数または関数を定義できます。関数で宣言された static 変数の値は、関数呼び出し間で保持されます。

uniform と varying 修飾子

変数に uniform 修飾子がある場合、ギャング内のすべてのプログラム・インスタンスで共有される変数のインスタンスは 1 つのみです。つまり、すべてのプログラム・インスタンスで同じ値になります。uniform 変数は varying 変数よりも必要なストレージが少なくなるのに加え、適用可能な場合、多くのパフォーマンス上の利点をもたらします (「均一な制御フロー」を参照)。可変変数は varying で修飾できますが、varying がデフォルトであるため、修飾しても効果はありません。

「ポインタータイプ」と「タイプキャスト」の節で説明される規則には 2 つの例外があります。

uniform 変数はプログラムの実行時に変更できますが、ギャング全体に対し同じ値を持つ特性を維持する方法でのみ変更できます。したがって、2 つの uniform 変数を加算して結果を uniform 変数に代入することは許可されますが、非 uniform (均一ではない可変) 値を uniform 変数に代入するには違法であり、コンパイル時にエラーとなります。

uniform 変数は、必要に応じて暗黙的にさまざまなタイプに変換されます。

```
uniform int x = ...;
int y = ...;
int z = x * y;  // x は乗算のため varying に変換されます
```

配列自体は uniform でも varying でもありませんが、配列に格納される要素は次のようになります。

```
float foo[10];
uniform float bar[10];
```

最初の宣言はメモリー内の 10 個のギャング幅の float 値に相当し、2 番目の宣言は 10 個の float 値に相当します。

タイプの新しい名前を定義

typedef キーワードは、タイプに名前を付けるために使用できます。

```
typedef int64 BigInt;
typedef float Float3[3];
```

C 言語の構文に従って、上記の宣言では BigInt を int64 タイプ、Float3 を float[3] タイプとして定義します。

また、C 言語と同様に typedef は新しいタイプを作成するのではなく、既存のタイプに新しい名前を付けるだけに使用します。したがって、上記の Float3 パラメーターを受け取ることを宣言する関数に、float[3] タイプの値を渡すことは合法です。

ポインタータイプ

メモリー上のデータへのポインターを使用して、ポインター操作、ポインターによるメモリー値の更新など、C 言語と同様の機能がサポートされます。ほかの基本タイプと同様に、ポインターは uniform でもあれば varying でもあります。

** インテル® ISPC のほかのタイプと同様に、ポインターは明示的に uniform 修飾子が指定されない限り、デフォルトで varying です。ただし、指定されるタイプのデフォルトの可変性は uniform です。** この規則を次の例で説明します。

例えば、次のコードにある ptr 変数は、uniform float 値への可変ポインターです。各プログラム・インスタンスは、個別のポインター値を保持し、*ptr への代入は通常メモリーへの散布 (scatter) を意味します。

```
uniform float a[] = ...;
int index = ...;
float * ptr = &a[index];
*ptr = 1;
```

uniform ポインターは、適切に配置された修飾子によって宣言できます。

```
float f = 0;
varying float * uniform pf = &f;  // varying float への uniform ポインター
*pf = 1;
```

uniform ポインターを宣言するのに uniform 修飾子を使用することは意外であると思われるかもしれませんが、それ自体が const であるポインター (const タイプをポイントするのではない) が、C 言語でどのように宣言されるか思い出してください (宣言を右から左に見ると意味が分かります: varying float への uniform ポインター)。

uniform ポインターが varying データタイプを指す場合、微妙な違いがあります。この場合、各プログラム・インスタンスは、メモリー内の別の場所にアクセスします (ベースとなる varying データタイプ自体が、各プログラム・インスタンスで個別のメモリー位置に配置されているため)。

```
float a;
varying float * uniform pa = &a;
*pa = programIndex;  // (a = programIndex) と同等
```

また、C では配列は暗黙的にポインターに変換されます。

```
float a[10] = { ... };
varying float * uniform pa = a;     // a の最初の要素へのポインター
varying float * uniform pb = a + 5; // a の 5 番目の要素へのポインター
```

どのようなポインタータイプでも、デスティネーションのタイプが uniform ポインターである場合、元のタイプが varying ポインターでない限り、明示的に別のポインタータイプにタイプキャストできます。

```
float *pa = ...;
int *pb = (int *)pa;  // 正当ですが注意してください
```

ほかのタイプと同様に、uniform ポインターは、varying ポインターにタイプキャストできます。

void ポインターは、タイプキャストなしで任意のポインタータイプに割り当て可能です。

```
float foo(void *);
int *bar = ...;
foo(bar);
```

NULL ポインターに対応する特別な NULL 値が定義されています。特殊ケースとして、整数値ゼロは暗黙的に NULL ポインターに変換され、ポインターは条件式で暗黙的にブール値に変換されることがあります。

```
void foo(float *ptr) {
    if (ptr != 0) { // または (ptr != NULL)、または just (ptr)
       ...
```

ポインタータイプから整数タイプへの明示的なタイプキャスト、またはその逆は許可されます。この変換は、関数呼び出しなどでは暗黙的に行われないことに注意してください。

関数ポインタータイプ

C や C++ と同様に、関数へのポインターを取得して利用することもできます。関数ポインタータイプを宣言する構文は、これらの言語と同一です。typedef が最も一般的です。

```
int inc(int v) { return v+1; }
int dec(int v) { return v-1; }

typedef int (*FPType)(int);
FPType fptr = inc;  // vs. int (*fptr)(int) = inc;
```

関数ポインターを指定して、ポインターが指す関数を呼び出すこともできます。

```
int x = fptr(1);
```

関数を呼び出すため、関数アドレスを取得して関数ポインターに割り当てたり、逆参照する必要はありません。

インテル® ISPC データへのポインターと同様に、関数ポインターは uniform または varying のどちらかです。uniform ポインターを介した呼び出しは、ギャング内のすべての実行中のプログラム・インスタンスがターゲット関数を呼び出します。varying 関数ポインターによる呼び出しについては、「ギャング収束の保証」で説明します。

参照タイプ

インテル® ISPC はまた、参照タイプ (C++ の参照タイプと同じ) を提供しており、関数への値の参照渡し、関数からの複数の結果のリターン、既存の変数の変更などが可能になります。

```
void increment(float &f) {
    ++f;
}
```

C++ のように、一度変数にバインドされた参照は、別の変数に再バインドされることはありません。

```
float a = ..., b = ...;
float &r = a;  // a を参照する r を作成
r = b;  // b を a に割り当てますが、r を b に参照させることはありません
```

インテル® ISPC における参照の制限事項として、参照は可変 lvalue にバインドできません。バインドするとコンパイル時にエラーが発生します。これは、vptr が varying ポインタータイプ (言い換えると、ギャング内の各プログラム・インスタンスが個別のポインター値を保持する) である、次のコードで説明できます。

```
uniform float * uniform uptr = ...;
float &ra = *uptr;  // ok
uniform float * varying vptr = ...;
float &rb = *vptr;  // ERROR: *vptr は varying lvalue タイプです。
```

この制限は、参照は内部的に uniform か varying ポインターのいずれかで表現する必要があることに由来します。varying ポインターを選択すると高い柔軟性が得られこの制限も解消されますが、uniform ポインターが必要な一般的なケースではパフォーマンスが低下します。回避策として、varying lvalue の参照が必要な場合、varying ポインターを使用します。

列挙タイプ

enum キーワードは、列挙タイプの名前とその後に続く中括弧で区切られた列挙子 (オプション) のリストで構成され、インテル® ISPC ではユーザー定義の列挙タイプを定義できます。

```
enum Color { RED, GREEN, BLUE };
enum Flags {
    UNINITIALIZED = 0,
    INITIALIZED = 2,
    CACHED = 4
};
```

異なるタイプの enum は、暗黙的に変換しようとするとコンパイル時にエラーが発生しますが、明示的に相互キャストできます。

```
Color c = (Color)CACHED;
```

しかし、列挙子は暗黙的に整数タイプに変換されるため、整数パラメーターを受け取る関数に直接渡すことができ、整数を含む式中でも使用できます。ただし、そのような式の整数結果を列挙タイプの変数に代入するには、明示的に列挙タイプに戻す必要があります。

```
Color c = RED;
int nextColor = c+1;
c = (Color)nextColor;
```

この場合、インクリメント・オペレーターを使用すると明示的なキャストを回避できます。

```
Color c = RED;
++c;  // ここでは c == GREEN
```

ショート・ベクトル・タイプ

インテル® ISPC は、ショートベクトルを定義するためパラメーター化されたタイプをサポートしています。このショートベクトルは、float や int といった基本タイプにのみ使用でき、配列や構造体には適用できません。

注: インテル® ISPC はショートベクトルをプログラムのベクトル化に使用するわけではなく、単に構文上の便宜のため提供しています。ショートベクトを使用または未使用のコードを記述することで、2 つのアプローチ間に顕著なパフォーマンスの差は生じないはずです。

これらのタイプ宣言には、C++ のテンプレートに似た構文が使用されます。

```
float<3> foo;   // 3 つの float のベクトル
double<6> bar;
```

これらのベクトル長は任意ですが、比較的短いベクトルが想定されます。

typedef を使用して、ベクトル長を示す配列のサイズを省略することもできます。

```
typedef float<3> float3;
```

インテル® ISPC は一般的にテンプレートをサポートしていません。特に、ベクトル長はコンパイル時に定数である必要があるだけではなく、ベクトル長をパラメーターとする関数を記述することもできません。

```
uniform int i = foo();
// ERROR: レングスはコンパイル時定数 1 である必要があります
float<i> vec;
// ERROR: ベクトル長のパラメーターを持つ関数は許可されません
float<N> func(float<N> val);
```

これらのショート・ベクトル・タイプに対する算術演算は、ベクトル内の値にコンポーネント単位で適用されます。次の簡単な例について考えてみます。

```
float<3> func(float<3> a, float<3> b) {
    a += b;    // a と b のそれぞれの要素を加算
    a *= 2.;   //  a のすべての要素に 2 を乗算
    bool<3> test = a < b;  // コンポーネント単位の比較
    return test ? a : b;   // それぞれコンポーネントの最小値を返す
}
```

上記のように、スカラータイプがベクトル式で使用されると、対応するベクトル型に自動的に変換されます。この例では、定数 2 は関数の 2 行目の乗算に向けて値 2 の 3 つのベクトルに変換されます。

ほかのショート・ベクトル・タイプ間の変換も期待どおり機能しますが、2 つのベクトルタイプは同じ長さである必要があります。

```
float<3> foo = ...;
int<3> bar = foo;    // ok、要素を int へキャスト
int<4> bat = foo;    // ERROR: 異なるベクトル長
float<4> bing = foo; // ERROR: 異なるベクトル長
```

便宜上、ショートベクトルは個々の要素値のリストを使用して初期化できます。

```
float x = ..., y = ..., z = ...;
float<3> pos = { x, y, z };
```

これらのショート・ベクトル・データ・タイプの個々の要素にアクセスするには、2 つの方法があります。1 つ目は、配列インデックス・オペレーターを使用するものです。

```
float<4> foo;
for (uniform int i = 0; i < 4; ++i)
    foo[i] = i;
```

インテル® ISPC は、構造体メンバーのアクセス・オペレーターのオーバーロードによって、ショートベクトルの最初のいくつかの要素の命名とアクセスに特化したメカニズムを提供します。構文は HLSL の構文に似ています。

```
float<3> position;
position.x = ...;
position.y = ...;
position.z = ...;
```

具体的には、ショート・ベクトル・タイプの最初の要素は .x または .r で、2 番目の要素は .y または .g、3 番目の要素は .z または .b で、4 番目の要素は .w または .a でアクセスできます。ベクトルサイズを超える要素へのインデックス・アクセスは未定義の動作を招き、プログラムがクラッシュする可能性があります。

「スウィズル」向けに拡張された構文を利用して、ほかのショートベクトル値から新しいショートベクトルを作成することもできます。

例:

```
float<3> position = ...;
float<3> new_pos = position.zyx;  // 逆順のコンポーネント
float<2> pos_2d = position.xy;
```

上記のように、単一の要素に代入することは可能ですが、現在、代入式の左辺にスウィズルを使用することはできません。

```
int8<2> foo = ...;
int8<2> bar = ...;
foo.yz = bar;   // Error: 式の左辺には割り当てできません
```

配列タイプ

C や C++ と同様に、任意のタイプの配列を宣言できます。

```
float a[10]; // 10 この可変単精度浮動小数点数の配列
uniform int * varying b[20]; // 均一な int への 20 個の可変ポインターの配列
```

多次元配列は、配列の配列として指定できます。以下は 15 個の float 配列の 5 つの配列を宣言します。

```
uniform float a[5][15];
```

配列のサイズは、配列の初期化リストで決定されますが、配列のサイズはコンパイル時の定数である必要があります。詳細は、次の「宣言と初期化子」を参照してください。1 つの例外は、「サイズのない配列」をパラメーターとして受け取るように関数を宣言できることです。

```
void foo(float array[], int length);
```

最後に、配列の名前は必要に応じて、配列タイプへの均一なポインターに自動で暗黙的に変換されます。

```
uniform int a[10];
int * uniform ap = a;
```

構造体タイプ

集約データ構造は、構造体を使用して生成できます。

```
struct Foo {
    float time;
    int flags[10];
};
```

C++ と同様に、構造体が宣言された後、構造体名を使用してインスタンスを作成できます。

```
Foo f;
```

または、構造体名の前に struct を指定できます。

```
struct Foo f;
```

構造体宣言のメンバーは、それぞれ uniform や varying 修飾子を持つことができますが、レート修飾子を持たない場合もあります。その場合、それらの可変性は最初はバインドされていません。

```
struct Bar {
    uniform int a;
    varying int b;
    int c;
};
```

上記の宣言では C の可変性はバインドされていません。バインドされていない構造体メンバーの可変性は、構造体の定義で解決されます。構造体が uniform の場合、バインドされていないメンバーは uniform であり、構造体が varying の場合、バインドされていないメンバーは varying です。

```
Bar vb;
uniform Bar ub;
```

ここでは、b は varying Bar です (varying がデフォルトの可変性であるため)。Bar が上記のように定義されている場合、vb.a は uniform int です。これは、可変性が Bar タイプの元の宣言でバインドされているためです。同様に、vb.b は varying です。vb が varying であるため、vb.c の可変性も varying です。

同様に、ub.a は uniform、ub.b は varyinmg、そして ub.c は uniform です。

多くの場合、構造体メンバーをアンバインドの可変性で宣言して、uniform と varying の両方ですべてが同じ可変性を持つようにするのは利点があります。特に、構造体が uniform タイプにバインドされたメンバーを持つ場合、構造体配列への可変インデックスは適用できません。次の例について考えてみます。

```
struct Foo { uniform int a; };
uniform Foo f[...] = ...;
int index = ...;
Foo fv = f[index];  // エラー
```

ここで、Foo タイプにバインドされた uniform な可変性を持つメンバーがあります。上記のコードでは、index はプログラム・インスタンスごとに異なる値を持つため、f[index] の値はプログラム・インスタンスごとに Foo::a の異なる値を格納する必要があります。ただし、varying Foo には 1 つのメンバーしかありません。これは、a が Foo の宣言で uniform な可変性で宣言されているためです。そのため、最後の行のインデックス操作はエラーになります。

オペレーターのオーバーロード

インテル® ISPC の構造体タイプのオーバーロード・オペレーターのサポートには制限があります。現在、次のバイナリー・オペレーターのみがサポートされています: *、/、%、+、-、>> および <<。オーバーロード・オペレーターのサポートは C++ 言語に類似しています。struct S のオペレーターをオーバーロードするには、struct S または struct S& タイプの 2 つのパラメーターを受け取り、これらのどちらかのタイプを返すキーワード・オペレーターを使用して関数を宣言および実装する必要があります。

例:

```
struct S { float re, im;};
struct S operator*(struct S a, struct S b) {
    struct S result;
    result.re = a.re * b.re - a.im * b.im;
    result.im = a.re * b.im + a.im * b.re;
    return result;
}

void foo(struct S a, struct S b) {
    struct S mul = a*b;
    print("a.re:   %¥na.im:   %¥n", a.re, a.im);
    print("b.re:   %¥nb.im:   %¥n", b.re, b.im);
    print("mul.re: %¥nmul.im: %¥n", mul.re, mul.im);
}
```

配列構造体タイプ

実行中のプログラム・インスタンスが、連続したメモリーのロードとストアを行えるようにメモリー内にデータを配置できれば、全体のパフォーマンスを向上できます。メモリーアクセスの一貫性を高める方法の 1 つが、メモリー上に「配列構造体」(SOA) 形式で構造体を配置することです。SOA レイアウトの利点については、『インテル® ISPC パフォーマンス・ガイド』の「可能な限り配列構造体レイアウトを使用する」で詳しく説明しています。

インテル® ISPC は、SOA 形式でデータを配置し、言語レベルで 2 つのアクセス機能を提供します。

- 通常の構造体を SOA 形式の構造体に変換する soa キーワード。
- SOA 配列のインデックスを透過的に処理する SOA 配列のインデックス構文。

次の簡単な構造体の宣言について考えてみます。

```
struct Point { float x, y, z; };
```

soa レート修飾子を使用して、この構造体の SOA バリアント配列を宣言できます。

```
soa<8> Point pts[...];
```

Point インスタンスのメモリーレイアウトには SOA 変換が適用され、メモリーには 8 個の x 値と 8 個の y 値が配置されます。次に soa<8> Point の完全な宣言を示します。

```
struct { uniform float x[8], y[8], z[8]; };
```

SOA データの配列がある場合、SOA 配列から適切な値にアクセスする、配列のインデックス操作 (およびポインター操作) を行います。次の例について考えてみます。

```
soa<8> Point pts[...];
uniform float x = pts[10].x;
```

生成されるコードは、2 番目の 8 レーンの SOA 構造体に効率良くアクセスし、そこから 3 番目の x 値をロードします。通常、SOA 配列要素にアクセスするコードは、AOS レイアウトにアクセスするコードと同様に記述できます。

ただし、SOA レイアウトでは、配列の 1 つの要素がメモリー上で連続して配置されないことに注意してください。上記の例では、pts[1].x と pts[1].y が 7 つの float 値で分離されています。

インテル® ISPC の SOA タイプの実装にはいくつかの制限がありますが、今後のリリースで改善される可能性があります。

- soa データを void ポインターにキャストするのは違法です。
- SOA 構造体の参照タイプは違法です。
- SOA 構造体のすべてのメンバーはレート修飾子を持ってはなりません。特に、soa が適用された構造体で明示的に修飾された uniform または varying メンバーを持つことは違法です。

宣言と初期化子

変数は C と同様に宣言され、割り当てられます。

```
float foo = 0, bar[5];
float bat = func(foo);
```

さらに複雑な宣言もあります。

```
void (*fptr_array[16])(int, int);
```

ここで、fptr_array は、戻り値が void で、2 つの int タイプのパラメーターを受け取る関数への 16 個のポインター配列です。

変数が初期化子なしで宣言された場合、値が代入されるまで変数の値は未定義です。未定義の変数を読み取ると未定義の動作になります。

関数外部のファイルスコープで宣言された変数は、すべてグローバル変数です。グローバル変数が static キーワードで修飾されている場合、その変数は定義されたコンパイル単位でのみ参照できます。C/C++ と同様に、関数内で static 修飾された変数は、関数が起動されてもその値が維持されます。

C++ のように、基本ブロックの先頭で変数を宣言する必要はありません。

```
int foo = ...;
if (foo < 2) { ... }
int bar = ...;
```

また、for 文の初期化パートで変数を宣言することもできます。

```
for (int i = 0; ...)
```

可変変数は、中括弧で囲まれた個々の要素の値で初期化できます。値の数はターゲット幅と同一である必要があります。そのため、static varying の初期化は、#if TARGET_WIDTH でガードしない限り、異なる幅を持つターゲットには対応できません。

```
#if TARGET_WIDTH == 4
   varying int bar = { 1, 2, 3, 4 };
#elif TARGET_WIDTH == 8
   varying int bar = { 1, 2, 3, 4, 5, 6, 7, 8 };
#elif TARGET_WIDTH == 16
   ...
#endif
```

配列は、中括弧で囲まれた個々の要素の値で初期化できます。

```
int bar[2][4] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };
```

初期化式を含む配列は、次元を指定せずに宣言できます。この場合、初期化式の「形状」を使用して、配列の次元を決定します。

```
// これは、初期化式により bar[2][4] に相当します
int bar[][] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };
```

構造体は、中括弧で囲まれた個々の要素の値で初期化できます。

```
struct Color { float r, g, b; };
....
Color d = { 0.5, .75, 1.0 }; // r = 0.5, ...
```

構造体配列と構造体内の配列は、通常の構文で初期化できます。

```
struct Foo { int x; float bar[3]; };
Foo fa[2] = { { 1, { 2, 3, 4 } }, { 10, { 20, 30, 40 } } };
// fa[1].bar[2] == 40 など
```

式

式の記述に必要な C のオペレーターがすべて含まれます。ここではすべてを示しませんが、実際に使用されるものを簡単にまとめています。

```
unsigned int i = 0x1234feed;
unsigned int j = (i << 3) ^ ~(i - 3);
i += j / 6;
float f = 1.234e+23;
float g = j * f / (2.f * i);
double h = (g < 2) ? f : g/5;
```

構造体メンバーへのアクセスと配列のインデックス操作も、C 言語と同じように機能します。

```
struct Foo { float f[5]; int i; };
Foo foo = { { 1,2,3,4,5 }, 2 };
return foo.f[4] - foo.i;
```

アドレス取得オペレーター、ポインター逆参照オペレーター、およびポインター・メンバー・オペレーターも機能します。

```
struct Foo { float a, b, c; };
Foo f;
Foo * uniform fp = &f;
(*fp).a = 0;
fp->b = 1;
```

C および C++ と同様の、|| の評価と && の論理オペレーター、および選択オペレーター ?: は、短絡評価です。左辺の値が論理オペレーターの値を決定する場合、右辺は評価されません。例えば、次のコードでは、NUM_ITEMS 以上の index 値は array[index] では評価されません。

```
if (index < NUM_ITEMS && array[index] > 0) {
    // ...
}
```

短絡評価は、生成されるコードに若干のオーバーヘッドを追加する可能性があります。パフォーマンスへの影響のため、短絡評価が適切でない場合は、短絡評価なしでこれらの操作を提供する標準ライブラリーのヘルパー関数を紹介する「論理および選択操作」を参照してください。

動的メモリー割り当て

インテル® ISPC プログラムは、C++ の new と delete オペレーターを使用して、メモリーを動的に割り当て (および解放) できます。

```
int count = ...;
int *ptr = new int[count];
// ptr を使用...
delete[] ptr;
```

上記のコードでは、各プログラム・インスタンスが個別のカウントサイズの uniform int 値の配列を割り当て、そのメモリーの使用後は割り当てを解放します。インテル® ISPC プログラムで new と delete を使用すると、プラットフォームに対応する C ライブラリーのアライメントされたメモリー割り当てルーチン呼び出しとして実装されます (Linux* と macOS* では posix_memalign() と free()、Windows* では _aligned_malloc() と _aligned_free())。そのため、インテル® ISPC の new と delete をペアで使用することを推奨します。C/C++ のメモリー管理関数とはペアにしないでください。

new の uniform と varying の規則は、ポインターが対応する規則に類似することに注意してください (「ポインタータイプ」で説明しています)。具体的には、new の式で特定のレート修飾子が指定されていない場合、デフォルトでは、各プログラム・インスタンスが一意の割り当てを行う varying new を実行します。割り当てられるタイプはデフォルトで uniform です。

ポインターが delete された後に、そのポインターが指していたメモリーにアクセスするのは違法です。削除はプログラム・インスタンスごとに行われます。次のコードを考えてみます。

```
int *ptr = new int[count];
// use ptr
if (count > 1000)
    delete[] ptr;
// ...
```

ここで、count が 1000 より大きなプログラム・インスタンスは、ptr が指す動的に割り当てられたメモリーを delete しますが、1000 以下のプログラム・インスタンスでは delete しません。そのため、プログラム・インスタンスの前者のセットが *ptr にアクセスするのは違法ですが、後者のセットが ptr が指すメモリーにアクセスするのは問題ありません。new が返すポインター値を複数回 delete するのは違法であることに注意してください。

ときには、一連のプログラム・インスタンス全体に対して一度の割り当てを行うと便利な場合があります。new 文を uniform 修飾して、単一のメモリーを割り当てることができます。

```
float * uniform ptr = uniform new float[10];
```

new の通常の呼び出しでは varying ポインター (つまり、プログラム・インスタンスごとに個別に割り当てられた異なるメモリーへのポインター) を返しますが、uniform new では単一の割り当てを行い、uniform ポインターを返します。uniform new では、割り当てられたタイプのデフォルトの可変性は varying であることを思い出してください。つまり、上記のコードは 10 個の varying float 値の配列を割り当てていることになります。

uniform new を使用する場合、微妙な点に注意を払う必要があります。返されたポインターが varying ポインターに格納されると (特定のプログラムでは適切であり便利なことがあります)、varying ポインターが後続の delete 文に渡される可能性があります。これは事実上エラーです。

```
varying float * ptr = uniform new float[10];
// ptr を使用...
delete ptr;  // ERROR: 可変ポインターは削除されました
```

この場合、ptr は、実行中のプログラム・インスタンスごとに一度ずつ (つまり複数回) delete されます。これはエラーです (上記のコードでアクティブなプログラム・インスタンスが 1 つのみである場合を除く)。

new 文を使用する場合、プログラムの必要性に基づいて、new オペレーター自体と割り当てられるデータのタイプの両方で、uniform または varying を適切に使用する必要があります。次の 4 つのメモリー割り当てを考えてみます。

```
uniform float * uniform p1 = uniform new uniform float[10];
float * uniform p2 = uniform new float[10];
float * p3 = new float[10];
varying float * p4 = new varying float[10];
```

float が 4 バイトで、ギャングサイズが 8 つのプログラム・インスタンスである場合、最初の割り当ては 10 個の uniform float 値 (40 バイト) の単一割り当てを示し、2 番目は 10 個の varying float 値 (8 * 4 * 10 = 320 バイト)、3 番目は 10 個の uniform float 値を 8 回割り当て (40 x 8 = 320 バイト)、そして最後はそれぞれ 320 バイトの 8 回の割り当てを行います。

varying データタイプが混在する varying 割り当てを行うことは望ましくないことに留意してください。その場合、各プログラム・インスタンスは、varying float メモリーの個別割り当てを行います。これは、プログラム・インスタンスがそれぞれ varying float の 1 つの要素にしかアクセスしない可能性が高く、無駄であると言えます。これが、割り当てられたタイプがデフォルトでポインターと new 文の両方で統一されている理由の 1 つです。

インテル® ISPC は、C++ のようなコンストラクターやデストラクターをサポートしていませんが、new 文で初期値を設定することは可能です。

```
struct Point { float x, y, z; };
Point *pptr = new Point(10, 20, 30);
```

ここでは、返された Point の x 要素は、10 に初期化されます。一般に、new 文の初期化子を使用して複雑なデータタイプを初期化する際の規則は、「宣言と初期化子」で説明されている変数の初期化子と同じ規則に従います。

タイプキャスト

C 形式のタイプキャスト式は C 言語と同じように機能しますが、バインドされていないタイプはデフォルトで varying としては扱われないという例外があります。

可変性を指定することなくタイプ T にキャストする場合、可変性はキャストされる式のタイプに依存します。つまり、(int) E は、元の式の E と同じ可変性を持ちますが、これは、結果の式が関数の引数として使用されると混乱を招く可能性があります。次の例について考えてみます。

```
float bar(uniform float f);
float bar(varying float f);
float foo(uniform int B) {
   return bar((float)B);
}
```

このコードは、完全な修飾タイプを使用することを提案する次の警告を生成します。

```
警告: uniform タイプ uniform int32 から /*unbound*/ float (可変性は指定されていない) へのタイプキャストは、uniform な可変性になります。

      関数の引数コンテキストでは、予期しない動作を引き起こす可能性があります。
      uniform float へのキャストを推奨します。
```

制御フロー

インテル® ISPC は、if、switch、for、while、do などの C 言語の制御フロー構造の大部分をサポートします。goto のサポートは限定されます。詳細は以下を参照してください。また、その文の制御フローで予期される実行時の一貫性に関するヒントを指示する、C 言語の制御フロー構造のバリアントもサポートされます。さらに、foreach と foreach_tiled による並列ループ構造も提供します。これらについては以降で詳しく説明します。

条件文: if

if 文は C と全く同じように動作します。true ブロックのコードは、条件が true と評価された場合にのみ実行されます。オプションの else 節が指定されている場合、else ブロックのコードは条件が false の場合にのみ実行されます。

```
float x = ..., y = ...;
if (x < 0.)
    y = -y;
else
    x *= 2.;
```

条件文: switch

switch 条件文も利用でき、これも C と同様に動作します。switch 文で使用される式は整数タイプでなければなりません (ただし uniform でも varying でもかまいません)。C と同様に、case コードの終端に break 文がない場合、実行は次の case に「フォールスルー」します。これは、次のコードで分かります。

```
int x = ...;
switch (x) {
case 0:
case 1:
    foo(x);
    /* フォールスルー */
case 5:
    x = 0;
    break;
default:
    x *= x;
}
```

反復文

C/C++ から継承された for、while、do の反復文に加えて、インテル® ISPC はデータを反復処理する特殊な機能を多数提供します。

基本反復文: for、while、do

インテル® ISPC は、C と同じ仕様で for、while、および do ループをサポートします。また、C++ のように for 文内で変数を宣言できます。

```
for (uniform int i = 0; i < 10; ++i) {
  // ループ本体
}
// ここでは i はスコープ内にありません
```

for、while、および do ループでは break 文と continue 文も使用できます。break は現在実行中のループから抜け出しますが、continue はループ本体の残りをスキップしてループの先頭にジャンプする効果があります。

これらのループ構造はすべて、ギャング内の各プログラム・インスタンスに対して独立して実行される効果があることに注意してください。例えば、プログラム・インスタンスの 1 つが continue 文を実行しても、continue 文を実行しなかったほかのプログラム・インスタンスは影響を受けません。

アクティブなプログラム・インスタンスの反復: foreach_active

foreach_active 構造は、アクティブなプログラム・インスタンスでシリアル化するループを指定します。ループ本体は、アクティブなプログラム・インスタンスごとに、そのプログラム・インスタンスのみが一度実行されます。

この構造の例として、各プログラム・インスタンスが独立して、更新中の共有配列へのオフセットを計算するアプリケーションを考えてみます。

```
uniform float array[...] = { ...};
int index = ...;
++array[index];
```

複数のアクティブなプログラム・インスタンスが、index に対し同じ値を計算する場合、上記のコードは未定義の動作となります (詳細は、「ギャング内のデータ競合」を参照)。array[index] のインクリメントは、foreach_active 文内に記述できます。

```
foreach_active (index) {
    ++array[index];
}
```

foreach_active キーワードの後の括弧内に指定された変数名 (ここでは index) を指定すると、その名前の const uniform int64 ローカル変数が宣言され、ループの各反復で実行されるプログラム・インスタンスの programIndex 値を格納します。

上記のコードでは、ループ本体の実行時に同時に実行されるプログラム・インスタンスは 1 つのみであるため、配列の更新は明確に定義されています。この例では、配列を安全に更新する代わりに、標準ライブラリーの「ローカルアトミック」操作を利用できることに注意してください。ただし、ローカルアトミック関数は、複雑なケースでの適用に使用できるとは限りません。

continue 文は foreach_active ループ内でも使用できますが、break 文と return 文は禁止されています。アクティブなプログラム・インスタンスがループ内で処理される順番は定義されていません。

foreach_active の詳細は、『インテル® ISPC パフォーマンス・ガイド』の「foreach_active を効率良く使用する」を参照してください。

一意の要素に対する反復: foreach_unique

可変変数の要素を反復処理して、同じ値を持つサブセットを一緒に処理すると便利です。例えば、値 {1, 2, 2, 1, 1, 0, 0, 0} を持つ可変変数 x を考えてみます。ここで、プログラムは 8 つのプログラム・インスタンスのギャングサイズを持つターゲットで実行されます。x はプログラム・インスタンス全体で 3 つの一意な値を持ちます: 0、1、2。

foreach_unique ループ構造を使用して、これらの一意の値を反復処理できます。次のコードでは、foreach_unique ループ本体が 3 つの一意な値のそれぞれに対し 1 回実行されます。実行マスクは、可変値が処理中の現在の一意な値と一致するプログラム・インスタンスになるように設計されています。

```
int x = ...; // assume {1, 2, 2, 1, 1, 0, 0, 0}
foreach_unique (val in x) {
    extern void func(uniform int v);
    func(val);
}
```

上記の func() は、値 0 で 1 回、値 1 で 1 回、そして値 2 で 1 回の計 3 回呼び出されます。値 0 で呼び出される場合、最後の 3 つのプログラム・インスタンスのみが実行されます。一意な値に対してループを実行する順番は定義されていません。

反復で使用される値を提供する可変式は一度だけ評価され、アトミックタイプ (float、int など)、列挙タイプ、またはポインタータイプである必要があります。反復変数 val は、反復タイプの const uniform タイプの変数です。val はループ内で変更できません。最後に、break 文と return 文はループ本体内での使用は禁止されていますが、continue 文は許可されます。

並列反復文: foreach および foreach_tiled

foreach と foreach_tiled 構造は、整数レンジの多次元ドメインを処理するループを指定します。この役割は、「糖衣構文」だけではなく、インテル® ISPC の並列計算を表現する 2 つの重要な機能の 1 つを提供します。

一般に、foreach または foreach_tiled 文は、カンマで区切られた 1 つ以上の次元指定子を持ちます。各次元は、識別子 = start ... end で指定されます。start は end 以下の符号付き整数値であり、start から end - 1 までのすべての整数値に対する反復を指定します。反復の次元指定は任意ですが、それぞれが異なる値を範囲にまたがります。foreach ループ内では、与えられた識別子が const varying int32 変数として利用できます。実行マスクは、foreach ループの各反復の開始時に "すべてオン" で始まりますが、ループ内の制御フロー構造によって変更されることがあります。

foreach ループで break 文や return 文が使用されると、コンパイル時にエラーが発生します。foreach ループ内で入れ子になった通常の for ループに break 文があるのは合法です。continue 文は foreach ループの中では合法であり、通常の for ループと同じ効果を持ちます。continue 文を実行したプログラム・インスタンスは、反復のループ本体の残りをスキップします。

現在 foreach 文の入れ子は違法ですが、インテル® ISPC の将来のリリースではサポートされる予定です。

具体的な例として次の foreach 構造を考えてみます。

```
foreach (j = 0 ... height, i = 0 ... width) {
    // ループ本体 -- データ要素 (i,j) を処理
}
```

j は 0 から heigth -1 まで、i は 0 から width-1 までの 2 次元領域に対するループを指定します。ループ内では変数 i と j が利用でき、それに応じて初期化されます。

foreach ループは、指定された反復処理の領域が自動的にギャング内のプログラム・インスタンスにマッピングされ、すべてのデータがギャングサイズのチャンクで処理できるようにします。ギャングサイズが 8 であるターゲットに対し、次の foreach ループを実行する例を考えてみます。

```
foreach (i = 0 ... 16) {
    // 要素 i の計算を実行
}
```

このループの有効な実行パスの 1 つは、プログラムカウンターがループ文を 16/8==2 回実行します。1 回目は varying int32 変数で、i はプログラム・インスタンス全体で (0、1、2、3、4、5、6、7) の値を持ち、2 回目は (8、9、10、11、12、13、14、15) の値、つまり、ループ実行の最後まで使用可能なプログラム・インスタンスをすべてのデータにマッピングします。

ただし、反復ドメインの要素が foreach ループが処理する順番を仮定してはなりません。例えば、次のコードの動作は未定義となります。

```
uniform float a[10][100];
foreach (i = 0 ... 10, j = 0 ... 100) {
    if (i == 0)
        a[i][j] = j;
    else
        // エラー: a[i-1][j] が設定されていないと仮定できません
        a[i][j] = a[i-1][j];
```

foreach 文は、通常、反復ドメインの最も内側の次元の連続する要素のセットを選択することで、反復ドメインを細分化します。この分解アプローチは、一貫性のあるメモリー読み取りと書き込みを可能にしますが、ほかの分解よりも制御フローの一貫性が低下する可能性があります。

そのため、foreach_tiled は、すべての次元をコンパクトにプログラム・インスタンスをドメイン内の位置をマッピングする方法で反復ドメインを分解します。例えば、ギャングサイズが 8 のターゲットでは、次の foreach_tiled 文は、毎回 j に 2 要素、i に 4 要素のチャンクで反復領域を処理しようとします。この 2 つの構造のトレードオフについては、『インテル® ISPC パフォーマンス・ガイド』で詳しく説明しています。

```
foreach_tiled (j = 0 ... height, i = 0 ... width) {
    // ループ本体 -- データ要素 (i,j) を処理
}
```

programIndex と programCount による並列反復

foreach と foreach_tiled に加え、ispc はビルトインの programIndex と programCount 変数によって、SPMD プログラム・インスタンスを操作するデータにマッピングする低レベルのメカニズムを提供します。

programIndex は、各プログラム・インスタンスの実行で使用される SIMD レーンのインデックスを提供します。これは、最初のプログラム・インスタンスには値 0 というように、変化する整数です。ビルトイン変数 programCount は、ギャングのインスタンスの総数を示します。これらを組み合わせることで、実行中のプログラム・インスタンスと入力データを一意に対応付けることができます。[4]

[4] programIndex は、OpenCL* の get_global_id() および CUDA* の threadIdx に似ています。

例えば、データ配列に対して何らかの計算を行うインテル® ISPC 関数について考えてみます。

```
for (uniform int i = 0; i < count; i += programCount) {
    float d = data[i + programIndex];
    float r = ....
    result[i + programIndex] = r;
}
```

ここでは、programCount 要素のチャンクでデータを明示的に分割するループを記述しています。ループの各反復では、実行中のプログラム・インスタンスは programIndex を使用して相互に連携し、すべてのデータ要素が確実に処理されるようにします。この例では foreach ループが適しています。foreach は、処理する要素数が programCount によって均等に分割できないデータを自然に処理しますが、上記のループはこのケースを暗黙的に想定しています。

非構造化制御フロー: goto

インテル® ISPC プログラムでは限られた条件下でのみ goto 文を使用できます。具体的には、プログラム・インスタンスが goto 文を実行する場合、すべてのプログラム・インスタンスがその文を実行し、goto に従うとコンパイラーが判断できる場合のみです。

言い換えると、goto 文を囲むスコープ内で「変化する」制御フローを記述するのは違法です。このような状況で goto 文が使用されるとエラーが発生します。

インテル® ISPC プログラムにラベルを記述して goto でジャンプするのは、C と同様です。次のコードは、インダクション変数 i が 0 から 10 になる for ループに相当する goto ベースのコードです。

```
  uniform int i = 0;
check:
  if (i > 10)
      goto done;
  // ループ本体
  ++i;
  goto check;
done:
  // ...
```

Coherent 制御フロー文: cif とフレンド

インテル® ISPC は、標準的な制御フロー構造のすべてにバリアントを提供しており、プログラム実行中の特定の場所で制御フローの一貫性を示すヒントを与えることができます。これらのメカニズムは、実行時に制御フローが実際に一貫性があるかを確認するため、追加のコードを実行する価値があることを示すヒントをコンパイラーに提供します。この場合、単純なコードパスを実行できる可能性があります。

これらの構文の最初の cif は、一貫性があることを期待する if 文です。コード内での cif の使用法は if と全く同じです。

```
cif (x < y) {
    ...
} else {
    ...
}
```

cif は、実行中の SPMD プログラムのほとんどの if 条件文が、すべて同じ結果になると予測されるヒントをコンパイラーに示します。

同様に、cfor、cdo、および cwhile は、ループ反復の開始時にすべてのプログラム・インスタンスが実行されているかどうかを確認します。その場合、「すべてオン」の実行マスクに最適化された特殊なコードパスを実行できます。

関数と関数呼び出し

C と同様に、関数は呼び出される前にインテル® ISPC で宣言する必要がありますが、関数本体が定義される場所の前で前方宣言できます。また、C と同様に、配列は関数に参照渡しされます。再帰的な関数呼び出しは許可されます。

```
int gcd(int a, int b) {
    if (a == 0)
        return b;
    else
        return gcd(b%a, a);
}
```

関数は、可視性と機能に影響するいくつかの修飾子を伴って宣言できます。C/C++ と同様に、関数はデフォルトでグローバルな可視性を持っています。関数に static 修飾子が宣言されている場合、関数は宣言されているファイル内でのみ参照できます。

```
static float lerp(float t, float a, float b) {
    return (1.-t)*a + t*b;
}
```

インテル® ISPC の launch 構造で起動できる関数は、task 修飾子を持つ必要があります。インテル® ISPC のタスク機能に関する詳しい説明は、「タスク並列処理: launch と sync 文」を参照してください。

関数にはマスクされていない修飾子を指定することもできます。この修飾子は、関数実行の開始時にすべてのプログラム・インスタンスをアクティブにする必要があることを示します (または、現在の実行マスクを関数呼び出し側から関数に渡してはならないことを示します)。すべてのプログラム・インスタンスが実行される際に関数が常に呼び出されることが判明している場合、この修飾子を指定するとパフォーマンスがわずかに向上することがあります。マスクされていないプログラムコードの詳細は、「実行マスクの再確立」を参照してください。

C/C++ アプリケーション・コードから呼び出される関数には、export 修飾子が必要です。これにより、インテル® ISPC がコンパイルしたファイルの C/C++ ヘッダーファイルを生成するように指示されている場合、通常の C リンクが行われ、宣言がヘッダーファイルに含まれるようになります。

```
export uniform float inc(uniform float v) {
    return v+1;
}
```

最後に、inline 修飾子が指定された関数は、常にインテル® ISPC によってインライン展開されます。inline 修飾子はヒントではなくインライン展開を強制します。コンパイラーは比較的短い関数の複雑性に応じて臨機応変にインライン展開を行いますが、必須である場合は inline 修飾子を使用します。同様に、noinline 修飾子で定義された関数は、インテル® ISPC がインライン展開することはありません。

関数オーバーロード

関数はパラメーター・タイプでオーバーロードできます。複数の関数定義がある場合、インテル® ISPC は次のモデルによって適切な関数を選択します。2 つのタイプの変換にはそれぞれコストがかかり、インテル® ISPC はコストが最も少ない変換を模索します。インテル® ISPC が変換を検出できない場合、この関数は適切でないことを意味します。そして、インテル® ISPC はすべての引数のコストを合計し、最終的に最もコストが少ない関数を選択します。選択された関数が、ほかの関数のコストよりも大きなコストの引数を持つ場合、これは曖昧なものとして扱われます。タイプの変換コストを小さいものから順番に示します。

1. パラメーターのタイプが正確に一致する場合。
2. 関数パラメーター・タイプが参照であり、パラメーターが一致するのは参照タイプのパラメーターがベースタイプと同等であると見なされる場合。
3. 関数パラメーター・タイプが定数参照であり、パラメーターが一致するのは参照タイプのパラメーターが const 属性を無視してベースタイプと同等であると見なされる場合。
4. 定数属性を除き、パラメーターが正確に一致する場合。[以降定数属性なし]
5. 参照属性を除き、パラメーターが正確に一致する場合。[以降参照属性なし]
6. パラメーターが情報を損失する危険がないタイプ変換のみで一致する場合 (例えば、int16 値を int32 のパラメーター値に変換する)。
7. uniform タイプから varying タイプへの昇格のみでパラメーターが一致する場合。
8. パラメーターが任意のタイプ変換で一致し、可変性を uniform から varying に変更しない場合 (例: int -> float、float -> int など)。
9. uniform タイプから varying タイプまで、拡張や昇格に応じたパラメーターを設定する場合 (6 と 7 の組み合わせ)。
10. パラメーターが任意のタイプ変換で一致し、可変性も uniform から varying に変更できる場合。

- 関数パラメーターが参照タイプで、上記の 2 または 3 が適切でない場合、関数は適切ではありません。
- 上記の 10 が適切でない場合、関数は適切ではありません。

実行マスクの再確立

「関数と関数呼び出し」で説明したように、マスクされていない修飾子で宣言された関数は、関数呼び出し側の実行マスクにかかわりなく、すべてのプログラム・インスタンスが実行されている状態で開始されます。文のブロックを unmasked で囲むと、関数内で同じ効果を得ることができます。

```
int a = ..., b = ...;
if (a < b) {
    // ここでは a < b のプログラム・インスタンスのみが実行されます
    unmasked {
        // ここではすべてのプログラム・インスタンスが実行されます
    }
    // 再度、a < b のプログラム・インスタンスのみが実行されます
}
```

unmasked は次のコードのように、プログラマーが「並列処理の軸を変えたい」、または並列処理を入れ子にしたい場合に有効です。

```
uniform WorkItem items[...] = ...;
foreach (itemNum = 0 ... numItems) {
    // items[itemNum] に対して、さらに処理を行う必要があるか
    // 判断する計算を行います...
    if (/* itemNum を処理する必要があります */) {
        foreach_active (i) {
            unmasked {
                uniform int uItemNum = extract(itemNum, i);
                // uItemNum にプログラム・インスタンスのギャング全体を適用
            }
        }
    }
}
```

一般的に、最初に SPMD による並列処理を適用して、どの項目でさらに処理が必要であるかを判断し、foreach ループでギャングチェックを並列に行います。この場合、以降の処理を必要としない項目に対応するプログラム・インスタンスが非アクティブになり、システム内で対応する計算機能も未使用となるため、それらのサブセットのみが処理を続行すると仮定すると、最初の判断と同じプログラム・インスタンスの foreach ループ内で処理するのは無駄です。

上記のコードでは、foreach_active で処理が必要な項目を順番に処理し、unmasked ですべてのプログラム・インスタンスを再度実行することで、この問題を回避しています。ギャング全体は、各 items[itemNum] に対して実行される計算に適用できます。

unmasked 文の使用には注意が必要です。この文は、プログラムの未定義動作の原因になります。例えば、次のコードについて考えてみます。

```
void func(float);
float a = ...;
float b;
if (a < 0) {
    b = 0;
    unmasked {
        if (b == 0)
            func(a);
    }
}
```

変数 a は特定の値に初期化され、b は初期化されていないため値は未定義です。if 文の評価では b に 0 を代入していますが、これは実行中のプログラム・インスタンス、つまり a < 0 の場合にのみ行われます。実行中のマスクを unmasked で再構築した後に、b を 0 と比較します。この比較は、a < 0 のプログラム・インスタンスに対しては明確に定義されており、true になりますが、そうでないプログラム・インスタンスでは b の値は未定義です。unmasked コードで varying 変数に書き込みを行うと、同様のケースが発生することがあります。

一般に、unmasked ブロック内のコードや、unmasked 修飾子を持つ関数は、外部スコープで宣言された varying 変数にアクセスする際に注意を払う必要があります。

タスク並列実行

インテル® ISPC は 1 つの処理コアの SIMD レーンに対し SPMD による並列処理を行う機能に加え、launch キーワードによる非同期関数呼び出しにより、複数のコアで並列実行を行う機能も備えています。launch で呼び出された関数は、非同期タスクとして実行され、多くの場合システム内の別のコアで実行されます。

タスク並列: launch と sync 文

タスク並列とインテル® ISPC を組み合わせる方法の 1 つとして、C/C++ アプリケーション・コードではタスク並列 (インテル® スレッディング・ビルディング・ブロック、OpenMP*、またはそのほかのタスクシステム) を使用して、タスクで必要に応じてインテル® ISPC を使用してベクトルレーン全体で SPMD 並列を行う方法があります。また、インテル® ISPC はインテル® ISPC コードからのタスクの起動もサポートしています。使用方法については、examples/mandelbrot_tasks の例を参照してください。

タスクとして起動する関数は、task 修飾子を追加して宣言する必要があります。

```
task void func(uniform float a[], uniform int index) {
    ...
    a[index] = ....
}
```

タスクは void をリターンしなければなりません。void でないタスクが定義されると、コンパイル時にエラーが発生します。

task が宣言されると、launch でタスクを起動できます。

```
uniform float a[...] = ...;
launch func(a, 1);
```

関数の launch 文の後、プログラムの実行は非同期に行われます。したがって、関数は、関数内で起動したタスクが書き込んだ値を同期せずにアクセスしてはなりません。関数は、sync 文を使用して起動したすべてのタスクの終了を待機できます。

```
launch func(a, 1);
sync;
// ここで a[] で計算された値を安全に使用できます...
```

また、タスクを起動する関数は、リターンする前に暗黙の同期が自動的に追加されます。そのため、タスクを起動する関数を呼び出す関数は、その関数の非同期計算のタイミングを意識する必要はありません。

launch 文で生成されるタスクは、単一ギャングのワークを持ちます。タスクの開始時に、launch 文の実行時と同じプログラム・インスタンスがそれぞれアクティブおよび非アクティブになります。起動したギャングのすべてのプログラム・インスタンスをアクティブにするには、unmasked 構造を使用できます (「実行マスクの再確立」を参照)。

複数のタスクをまとめて起動するには 2 つの方法があります。1 つは、一度に 1 つのタスクを起動し、そのタスクにパラメーターを渡すことで、全体の計算のどこに相当するか決定する方法です。

```
for (uniform int i = 0; i < 100; ++i)
    launch func(a, i);
```

このコードでは、100 個のタスクが起動され、それぞれが値 i をキーとして何らかの計算を行うと考えられます。一般に、うまく負荷分散を行うには、システム内のプロセッサー・コアよりも多くのタスクを起動すべきですが、タスク・スケジュールと実行のオーバーヘッドが計算を上回るほど多くはないと思われます。

また、単一の launch 文で複数のタスクを起動することもできます。上記の例を次のように単一の launch で書くこともできます。

```
launch[100] func2(a);
```

launch キーワードに角括弧で整数値が与えられると (コンパイル時定数でなくてもかまいません)、その数のタスクがキューに投入され非同期に実行されます。それぞれのタスクの中では、taskIndex と taskCount という 2 つの特殊ビルトイン変数が利用できます。最初の taskIndex は、0 から 1 までの範囲から起動するタスク数を引いたもので、taskCount は起動したタスク数に等しくなります。したがって、次の例では、func2 の実装で taskIndex を使用して、処理する配列要素を決定できます。

```
task void func2(uniform float a[]) {
    ...
    a[taskIndex] = ...
}
```

タスク修飾子を持つ関数内では、taskIndex と taskCount のほかに、threadIndex と threadCount の 2 つのビルトイン変数が提供されます。threadCount は、タスクシステムによって起動されたハードウェア・スレッドの総数を示します。threadIndex は、0 から threadCount-1 までのインデックスであり、現在のタスクを実行しているハードウェア・スレッドに対する一意のインデックスを示します。threadIndex は、現在のスレッドのプライベート・データにアクセスする際に利用できます。並列実行中のアクセスに同期を必要としません。

また、タスク処理システムは、多次元分割 (現在は 3 次元まで) に対応しています。タスクの 3D グリッドを起動するには、例えば x、y、z 次元にそれぞれ N0、N1、N2 個のタスクがあると仮定します。

```
float data[N2][N1][N0]
task void foo_task()
{
   data[taskIndex2][taskIndex1][threadIndex0] = taskIndex;
}
```

次の launch 式を使用します。

```
launch [N2][N1][N0] foo_task()
```

または

```
launch [N0,N1,N2] foo_task()
```

taskIndex の値は、taskIndex0 + taskCount0*(taskIndex1 + taskCount1*taskIndex2) に等しく、taskCount = taskCount0*taskCount1*taskCount2 で、0 から taskCount-1 までの範囲であるとします。launch 式で N1 または N2 が指定されていない場合は 1 とします。最後に、1 次元グリッドのタスクの場合、taskIndex は taskIndex0 と同等であり、taskCount は taskCount0 と同等です。

タスク並列: ランタイムの要件

インテル® ISPC のタスク起動機能を使用する場合、並列タスクの起動と同期を管理する 3 つの関数の C/C++ 実装を提供し、それらは実行ファイルにリンクされる必要があります。これらの関数はどの言語で実装しても構いませんが、C リンクである必要があります (C++ で定義する場合、プロトタイプは extern C ブロック中で宣言する必要があります)。

ユーザーがこれらの関数を提供することで、インテル® ISPC プログラムは、並列処理を管理する既存のタスクシステムを持つソフトウェアとの相互運用が容易になります。インテル® ISPC をマルチスレッドではないシステムで使用し、カスタム実装を記述しない場合、インテル® ISPC に同梱される examples/common/tasksys.cpp ファイルにある関数の実装を利用できます。

独自のタスクシステムを実装する場合の呼び出し要件について以降で説明します。examples/common/tasksys.cpp にあるタスクシステムの例を見ておくのもよいでしょう。独自のタスクシステムを導入していない場合は、この節の残りは読み飛ばしてもかまいません。

以下は、インテル® ISPC でタスクを管理するために必要な 3 つの関数宣言を示します。

```
void *ISPCAlloc(void **handlePtr, int64_t size, int32_t alignment);
void ISPCLaunch(void **handlePtr, void *f, void *data, int count0, int count1, int
count2);
void ISPCSync(void *handle);
```

これら 3 つの関数はすべて、第 1 パラメーターとして不透明ハンドル (または不透明ハンドルへのポインター) を受け取ります。このハンドルでタスクシステムのランタイムは、インテル® ISPC コードの異なる関数からのこれらの関数の呼び出しを区別できます。このように、タスクシステムの実装は、1 つの関数から起動されたタスクのみを効率良くその完了を待機できます。

ISPCLaunch() または ISPCAlloc() のいずれかがインテル® ISPC 関数から最初に呼び出されると、handlePtr パラメーターが指す void* は NULL になります。これらの関数の実装は、*handlerPtr を特定の種類の一意なハンドル値に初期化する必要があります。例えば、関数によって起動されたタスクを記録するため、小さな構造体を割り当てる場合があります。インテル® ISPC コードからこれらの関数の後続を呼び出すと、handlerPtr に同じ値が渡され、*handlerPtr から読み取りを行うと最初の呼び出しで格納された値が取得できます。

関数が exit (または明示的な同期文が実行) されると、*handlerPtr が NULL でない場合は ISPCSync() 呼び出しが実行されます。この場合、ほかの関数のようにポインターではなく、ハンドル値が ISPCSync() に直接渡されます。

ISPCAlloc() 関数は、タスクに渡されるパラメーターを格納するメモリーブロックを割り当てる際に使用します。指定されたサイズとアライメントのメモリーへのポインターが返されます。明示的な ISPCFree() 呼び出しがないことに注意してください。代わりに、インテル® ISPC 関数内で割り当てられたすべてのメモリーは、ISPCSync() が呼び出されたときに解放されます。

ISPCLaunch() は、1 つまたは複数の非同期タスクを起動する際に呼び出されます。インテル® ISPC コード内の launch 文は、生成されたコード内で ISPCLaunch() 呼び出しを行います。関数へのハンドルポインターの後の 3 つのパラメーターは、比較的単純です。void *f パラメーターは、このタスクのワークを実行するために呼び出す関数へのポインターを保持します。そして data は、この関数に渡すデータへのポインターを保持し、count0、count1、count2 は非同期実行のためキューに投入するこの関数のインスタンス数です。つまり、count0、count1、count2 は、それぞれ launch[n2][n1][n0] や launch[n0, n1, n2] など複数タスクの launch 文のパラメーター n0、n1、n2 に相当します。

関数ポインター f のシグネチャーは次のとおりです。

```
void (*TaskFuncPtr)(void *data, int threadIndex, int threadCount,
                    int taskIndex, int taskCount,
                    int taskIndex0, int taskIndex1, int taskIndex2,
                    int taskCount0, int taskCount1, int taskCount2);
```

この関数ポインターがタスクシステムで管理されるハードウェア・スレッドの 1 つから呼び出されると、ISPCLaunch() に渡されたデータポインターが 第 1 パラメーターになります。threadCount は、タスクを実行するために生成されたハードウェア・スレッドの総数を示します。threadIndex は、タスクを実行するハードウェア・スレッドを一意に識別する 0 から threadCount までの整数インデックスです。これらの値を使用して、スレッド・ローカル・ストレージにインデックスを付けることができます。

taskCount の値は、ISPCLaunch() を呼び出した launch 文で起動されたタスクの総数で (taskCount0 * taskCount1 * taskCount2 と等しくなければなりません)、この関数の各呼び出しには、起動された一連のタスクのどのインスタンスが実行されているかを区別するため、taskIndex = taskIndex0 + taskCount0 * (taskIndex1 + taskCount1 * taskIndex2) で、それぞれ 0 から taskCount、taskCount0、taskCount1、taskCount2 間の一意の値の taskIndex、taskIndex0、taskIndex1、taskIndex2 を与える必要があります。

LLVM 組込み関数

インテル® ISPC には、LLVM 組込み関数をインテル® ISPC のソースコードから直接呼び出す実験的な機能が用意されています。この機能は、結果を十分に理解しない限り、製品コードで使用することは推奨できません。具体的には以下です。

- LLVM 組込み関数の可用性と名称は、インテル® ISPC のビルドに使用された LLVM のバージョンに依存しますが、予告なく変更される可能性があります。
- ターゲット CPU で固有の組込み関数が利用可能であるか検証のみを行います。サポートされない組込み関数を使用すると、コンパイラーがクラッシュすることがあります。

LLVM 組込み関数は実験的に使用されており、次のような場合に有用であると考えられます。

- インテル® ISPC が特定の命令生成に失敗した場合、より高いパフォーマンスを得るために必要です。
- 標準ライブラリーや言語自体に上位プリミティブがない場合に必要です。例えば、新しい ISA の拡張機能などが考えられます。

LLVM 組込み関数を使用する有益なケースを見つけた場合、インテル® ISPC のバクトラッカー (英語) に情報をお送りください。

この機能を使用するには、インテル® ISPC に --enable-llvm-intrinsics スイッチを渡す必要があります。構文は通常の関数呼び出しと同様ですが、名称は @ シンボルで開始する必要があります。

例:

```
transpose = @llvm.matrix.transpose.v8f32.i32.i32(matrix, row, column);
```

コンパイル時にこの機能が利用可能であるか検出するには、ISPC_LLVM_INTRINSICS_ENABLED マクロが定義されているかどうか確認します。

インテル® ISPC 標準ライブラリー

インテル® ISPC には標準ライブラリーがあり、インテル® ISPC プログラムをコンパイルする際に自動的に利用できます。標準ライブラリーを無効にするには、コンパイラーに --nostdlib コマンドライン・フラグを渡します。

データの基本操作

論理および選択操作

インテル® ISPC が論理オペレーターと選択オペレーターの評価を短縮する式を思い出してください。(index < count && array[index] == 0) のような式が与えられた場合、array[index] == 0 は index < count が true の場合にのみ評価されます。このプロパティーは、前述のような式を記述する際に、2 番目の式が安全に評価できない可能性があるときに有効です。

このような短絡処理は、生成されるコードにオーバーヘッドをもたらす可能性があります。最初の値を評価し、2 番目の値を評価するコードを条件付きでスキップする追加の操作が必要になります。インテル® ISPC は、両方の式を評価するのが安全

で低コストであることを検出し、生成されたコードで短絡をスキップすることで、コストを軽減しようとします (プログラムから見える動作に変化はありません)。

コンパイラーがこれを検出できなくても、プログラマーが短絡動作を避けたい場合のため、標準ライブラリーではいくつかのヘルパー関数を提供しています。and() と or() は、短絡のない論理 AND と OR 演算を行います。

```
bool and(bool a, bool b)
bool or(bool a, bool b)
uniform bool and(uniform bool a, uniform bool b)
uniform bool or(uniform bool a, uniform bool b)
```

また、ブール条件に基づいて 2 つの値を選択する select() の 3 つのバリアントがあります。条件 cond が true であると t が選択され、そうでなければ f が選択されます。これらは、int8 タイプの select() のバリアントです。

```
int8 select(bool cond, int8 t, int8 f)
int8 select(uniform bool cond, int8 t, int8 f)
uniform int8 select(uniform bool cond, uniform int8 t, uniform int8 f)
```

int16、int32、int64、float、float16 および double タイプのバリアントもあります。

ビット操作

popcnt() のバリアントは、指定された値に設定されたビット数の占有カウントを返します。

```
uniform int popcnt(uniform int v)
int popcnt(int v)
uniform int popcnt(bool v)
```

いくつかの関数では、与えられた値の先頭から何ビットが 0 で、末尾の何ビットが 0 であるかを判定します。これらの関数の符号なしバージョンや、int64 および符号なし int64 タイプを受け取るバリアントもあります。

```
int32 count_leading_zeros(int32 v)
uniform int32 count_leading_zeros(uniform int32 v)
int32 count_trailing_zeros(int32 v)
uniform int32 count_trailing_zeros(uniform int32 v)
```

状況によっては、符号拡張により bool 値を整数に変換するのが便利なときもあります。bool 値が true の場合 (値が 1 になるだけではなく)、整数のビットがすべて 1 になります。sign_extend() 関数は、この機能を提供します。

```
int sign_extend(bool value)
uniform int sign_extend(uniform bool value)
```

また、packmask 関数により、bool の varying 値を整数に変換することもできます。

```
uniform int packmask(bool value)
```

intbits()、float16bits()、floatbits() および doublebits() 関数は、低レベルの浮動小数点ビット操作の実装に使用できます。例えば、intbits() は、指定された float 値のビットごとのコピーである符号なし int を返します。

注: (int) a ではなく C 言語の *((int *)&a) に相当します。

```
float16 float16bits(unsigned int16 a);
uniform float16 float16bits(uniform unsigned int16 a);
float floatbits(unsigned int a);
uniform float floatbits(uniform unsigned int a);
double doublebits(unsigned int64 a);
uniform double doublebits(uniform unsigned int64 a);
unsigned int16 intbits(float16 a);
uniform unsigned int16 intbits(uniform float16 a);
unsigned int intbits(float a);
uniform unsigned int intbits(uniform float a);
unsigned int64 intbits(double a);
uniform unsigned int64 intbits(uniform double a);
```

intbits()、float16bits()、floatbits() および doublebits() 関数は、実行時のコストはありません。これは、指定された値のビットをどのように解釈するかコンパイラーに通知するだけです。これにより、浮動小数点値の低レベルのビット表現を利用した関数を効率良く記述できます。

例えば、標準ライブラリーの abs() 関数は、次のように実装されます。

```
float abs(float a) {
    unsigned int i = intbits(a);
    i &= 0x7fffffff;
    return floatbits(i);
}
```

このコードは、指定された浮動小数点値が正であることを保証するため、上位ビットをクリアします。例えば、インテル® SSE ターゲットで使用する場合、単一の andps 命令にコンパイルされます。

数学関数

標準ライブラリーの数学関数は、比較的標準的な範囲の数学関数を提供します。

超越数学関数の多くでは異なる実装が利用できます。使用する数学ライブラリーは、--math-lib= コマンドライン引数で選択できます。この引数には次の値を指定できます。

- **default**: インテル® ISPC のデフォルトのビルトイン関数。これらにはそれなりの精度があります (例えば、sin は -10pi から 10pi の範囲で最大 1.45e-6 の絶対誤差があります)。
- **fast**: デフォルトのインテル® ISPC 実装より効率的ですが、精度が低いバージョンの関数。
- **svml**: インテルの「ショート・ベクトル・マス・ライブラリー」。これは、インテル® oneAPI DPC++/C++ コンパイラー (icx/icpx) およびインテル® oneAPI C++ コンパイラー・クラシック (icc/icpc) の一部として提供される独自のライブラリーです。いずれかを使用して最終的な実行可能ファイルを生成し、適切なライブラリーにリンクされるようにします。
- **system**: システムの数学ライブラリー。これは、多くのシステムではインテル® ISPC の実装よりも高精度ですが、非効率です。システムの数学関数は一度に 1 つの結果しか生成しないため (ベクトル化されていない)、インテル® ISPC はアクティブなプログラム・インスタンスごとにこれらの関数を一度呼び出す必要があります (ほかの 3 つのオプションではその必要はありません)。

基本数学関数

絶対値関数 abs() に加え、signbits() は与えられた値の符号ビットを抽出し、符号ビットがオン (つまり値が負) の場合は、0x80000000 を、オフの場合は 0 を返します。

```
float16 abs(float a)
uniform float16 abs(uniform float a)
float abs(float a)
uniform float abs(uniform float a)
double abs(double a)
uniform double abs(uniform double a)
int8 abs(int8 a)
uniform int8 abs(uniform int8 a)
int16 abs(int16 a)
uniform int16 abs(uniform int16 a)
int abs(int a)
uniform int abs(uniform int a)
int64 abs(int64 a)
uniform int64 abs(uniform int64 a)
unsigned int16 signbits(float16 x)
uniform unsigned int16 signbits(uniform float16 x)
unsigned int signbits(float x)
uniform unsigned int signbits(uniform float x)
unsigned int64 signbits(double x)
uniform unsigned int64 signbits(uniform double x)
```

float16、float、double タイプには、標準的な丸め関数が用意されています。インテル® SSE またはインテル® AVX をサポートするマシンでは、これらの関数はすべて roundss および roundps 命令のバリアントにマッピングされます。

```
float round(float x)
uniform float round(uniform float x)
float floor(float x)
uniform float floor(uniform float x)
float ceil(float x)
uniform float ceil(uniform float x)
float trunc(float x)
uniform float trunc(uniform float x)
```

rcp() は 1/v の近似値を計算します。アーキテクチャーによって誤差の範囲は異なります。

```
float rcp(float v)
uniform float rcp(uniform float v)
```

インテル® ISPC はまた、ニュートン・ラフソン法を使用しない、低精度の float 向けの rcp() も用意しています。

```
float rcp_fast(float v)
uniform float rcp_fast(uniform float v)
```

すべてのインテル® ISPC 標準タイプには、最小および最大関数の標準セットがあります。また、これらの関数は対応する組込み関数にマッピングされます。

```
float min(float a, float b)
uniform float min(uniform float a, uniform float b)
float max(float a, float b)
uniform float max(uniform float a, uniform float b)
unsigned int min(unsigned int a, unsigned int b)
uniform unsigned int min(uniform unsigned int a,
                         uniform unsigned int b)
unsigned int max(unsigned int a, unsigned int b)
uniform unsigned int max(uniform unsigned int a,
                         uniform unsigned int b)
```

clamp() 関数は、値を指定された範囲にクランプします。それらの実装は、min() と max() をベースとするため非常に効率的です。

```
float clamp(float v, float low, float high)
uniform float clamp(uniform float v, uniform float low,
                    uniform float high)
unsigned int clamp(unsigned int v, unsigned int low,
                   unsigned int high)
uniform unsigned int clamp(uniform unsigned int v,
                           uniform unsigned int low,
                           uniform unsigned int high)
```

isnan() 関数は、指定された値が浮動小数点値であるかどうかをテストします。

```
bool isnan(float16 v)
uniform bool isnan(uniform float16 v)
bool isnan(float v)
uniform bool isnan(uniform float v)
bool isnan(double v)
uniform bool isnan(uniform double v)
```

8 ビットおよび 16 ビットのデータ幅を操作する関数も多数用意されています。これらは、それらをサポートするターゲットで演算を実行する特殊な命令にマップされます。avg_up() は 2 つの値の平均値を計算し、その平均値が 2 つの整数の中間にある場合は切り上げを行います (つまり、(a+b+1)/2 を計算します)。

```
int8 avg_up(int8 a, int8 b)
unsigned int8 avg_up(unsigned int8 a, unsigned int8 b)
int16 avg_up(int16 a, int16 b)
unsigned int16 avg_up(unsigned int16 a, unsigned int16 b)
```

avg_down() は、2 つの値の平均値を計算して切り捨てを行います (つまり、(a+b)/2 を計算します)。

```
int8 avg_down(int8 a, int8 b)
unsigned int8 avg_down(unsigned int8 a, unsigned int8 b)
int16 avg_down(int16 a, int16 b)
unsigned int16 avg_down(unsigned int16 a, unsigned int16 b)
```

超越関数

値の平方根は sqrt() 関数で計算できます。利用可能であれば、ハードウェアの平方根組込み関数にマップされます。近似逆平方根 1/sqrt(v) は、rsqrt() によって計算できます。rcp() と同様に、この関数呼び出しのエラーはアーキテクチャーによって異なります。

```
float sqrt(float v)
uniform float sqrt(uniform float v)
float rsqrt(float v)
uniform float rsqrt(uniform float v)
```

インテル® ISPC はまた、ニュートン・ラフソン法を使用しない、低精度の float 向けの rsqrt() も用意しています。

```
float rsqrt_fast(float v)
uniform float rsqrt_fast(uniform float v)
```

インテル® ISPC は、三角関数の標準的な各種呼び出しを提供します。

```
float sin(float x)
uniform float sin(uniform float x)
float cos(float x)
uniform float cos(uniform float x)
float tan(float x)
uniform float tan(uniform float x)
```

対応する逆関数も利用できます。

```
float asin(float x)
uniform float asin(uniform float x)
float acos(float x)
uniform float acos(uniform float x)
float atan(float x)
uniform float atan(uniform float x)
float atan2(float y, float x)
uniform float atan2(uniform float y, uniform float x)
```

正弦と余弦の両方が必要である場合、sincos() 関数は、それぞれ個別の関数を 2 回呼び出すよりも効率良く両方を計算します。

```
void sincos(float x, varying float * uniform s, varying float * uniform c)
void sincos(uniform float x, uniform float * uniform s,
          uniform float * uniform c)
```

通常の指数関数および対数関数が提供されます。

```
float exp(float x)
uniform float exp(uniform float x)
float log(float x)
uniform float log(uniform float x)
float pow(float a, float b)
uniform float pow(uniform float a, uniform float b)
```

メモリー上の浮動小数点表現を操作する低レベルの専用関数がいくつか用意されています。標準の数学ライブラリーと同様に、ldexp() は値 x を 2^n 倍し、frexp() は正規化した仮数を直接返して、正規化した指数を pw2 パラメーターの 2 のべき乗を返します。

```
float ldexp(float x, int n)
uniform float ldexp(uniform float x, uniform int n)
float frexp(float x, varying int * uniform pw2)
uniform float frexp(uniform float x,
                 uniform int * uniform pw2)
```

float16、float、double タイプには、超越関数が用意されています。

飽和演算

インテル® ISPC の標準ライブラリーでは、すべての整数タイプの飽和 (オーバーフローは不可) 加算、減算、乗算、および除算が提供されます。

```
int8 saturating_add(uniform int8 a, uniform int8 b)
int8 saturating_add(varying int8 a, varying int8 b)
unsigned int8 saturating_add(uniform unsigned int8 a, uniform unsigned int8 b)
unsigned int8 saturating_add(varying unsigned int8 a, varying unsigned int8 b)

int8 saturating_sub(uniform int8 a, uniform int8 b)
int8 saturating_sub(varying int8 a, varying int8 b)
unsigned int8 saturating_sub(uniform unsigned int8 a, uniform unsigned int8 b)
unsigned int8 saturating_sub(varying unsigned int8 a, varying unsigned int8 b)

int8 saturating_mul(uniform int8 a, uniform int8 b)
int8 saturating_mul(varying int8 a, varying int8 b)
unsigned int8 saturating_mul(uniform unsigned int8 a, uniform unsigned int8 b)
unsigned int8 saturating_mul(varying unsigned int8 a, varying unsigned int8 b)

int8 saturating_div(uniform int8 a, uniform int8 b)
int8 saturating_div(varying int8 a, varying int8 b)
```

```
unsigned int8 saturating_div(uniform unsigned int8 a, uniform unsigned int8 b)
unsigned int8 saturating_div(varying unsigned int8 a,varying unsigned int8 b)
```

飽和演算関数には、上記の int8 タイプのほかに、int16 タイプ、int32 タイプ、int64 タイプの値をサポートするバージョンもあります。

擬似乱数数値

インテル® ISPC 標準ライブラリーでは、簡単な乱数発生器が提供されています。RNG の状態は、RNGState 構造体のインスタンスで管理され、シードは seed_rng() で提供されます。

```
struct RNGState;
void seed_rng(varying RNGState * uniform state, varying int seed)
void seed_rng(uniform RNGState * uniform state, uniform int seed)
```

例えば、RNGState state; seed_rng(&state, 1); のように、すべてのプログラム・インスタンスに同じ可変シード値を指定すると、ギャング内のすべてのプログラム・インスタンスに同じ疑似乱数のシーケンスが返されます。この動作が望ましくない場合、programIndex の値を seed に追加するか、seed が各プログラム・インスタンスに対して一意の値を持つようにします。

RNG にシードを渡した後、random() 関数で疑似乱数の unsigned int32 の値を、frandom() 関数で疑似乱数の float 値を取得できます。

```
unsigned int32 random(varying RNGState * uniform state)
float frandom(varying RNGState * uniform state)
uniform unsigned int32 random(RNGState * uniform state)
uniform float frandom(uniform RNGState * uniform state)
```

乱数生成

近年の CPU (インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge 以降など) は、真の乱数生成をサポートするものがあります。いくつかの標準ライブラリーでは、この機能を利用できます。

```
bool rdrand(uniform int32 * uniform ptr)
bool rdrand(varying int32 * uniform ptr)
bool rdrand(uniform int32 * varying ptr)
```

プロセッサーが乱数を生成するのに十分なエントロピーを保持していない場合、この関数は失敗し false を返します。プロセッサーが乱数生成に成功すると、ランダム値がポインターに格納され、true が返されます。この関数は一般に次のように使用し、成功するまで繰り返し呼び出す必要があります。

```
int r;
while (rdrand(&r) == false)
    ;// 空のループ本体
```

rdrand() には、上記の int32 のほかに、int16、float、int64 の値を返すバージョンもあります。

インテル® AVX2 より古いターゲット向けにコンパイルする場合、rdrand() 関数は常に false を返します。

出力関数

インテル® ISPC には、プログラム実行中に値を表示する簡単な print 文があります。次のインテル® ISPC プログラムでは、print 文の使い方が 3 つ示されています。

```
export void foo(uniform float f[4], uniform int i) {
    float x = f[programIndex];
    print("i = %, x = %¥n", i, x);
    if (x < 2) {
        ++x;
        print("added to x = %¥n", x);
    }
    print("last print of x = %¥n", x);
}
```

いくつか注意すべき点があります。この関数は、printf ではなく print と呼ばれます (C 言語とは異なります)。次に、この関数に渡されるフォーマット文字列は、対応する値を表示する位置を示すパーセント記号を 1 つだけ使用します。フォーマット・オペレーターのタイプと渡されるデータのタイプを一致させる必要はありません。ただし、printf が提供するような豊富なデータ・フォーマット機能 (%.10f. のような形式) は、現在使用できません。

f パラメーターに渡された float 配列 (0、1、2、3) と i パラメーターに値 10 を使用してこの関数を呼び出すと、4 レーンのコンパイルターゲットでは次の出力が得られます。

```
i = 10, x = [0.000000,1.000000,2.000000,3.000000]
x = [1.000000,2.000000,((2.000000)),((3.000000))] へ加算
x = [1.000000,2.000000,2.000000,3.000000] の最後の出力
```

可変変数を表示する場合、現在実行されていないプログラム・インスタンスの値は、非アクティブなプログラム・インスタンスを示す二重括弧内に表示されます。非アクティブなプログラム・インスタンスの要素には、ガベージ値が含まれていることがありますが、状況によってはその値を確認すると便利なことがあります。

アサーション

インテル® ISPC 標準ライブラリーには、assert() 文をインテル® ISPC プログラムコードに追加するメカニズムが含まれています。C の assert() のように、asset() 関数は単一のブール式を引数として受け取ります。実行時に式が false と評価された場合、診断エラーメッセージが表示され、abort() 関数が呼び出されます。

可変量のデータで呼び出された場合、呼び出された時点で実行中のプログラム・インスタンスのいずれかで式が false と評価されると、アサーションがトリガーされます。したがって、次のようなコードを考えてみます。

```
int x = programIndex - 2;  // (-2, -1, 0, ... )
if (x > 0)
    assert(x > 0);
```

実行中のいずれのプログラム・インスタンスでも条件が true でないため、asset() 文はトリガーされません。この assert() 文が if の外側にあれば、トリガーされます。

コンパイルするファイル内のアサーションをすべて無効にするには (最適化したリリースビルドなど)、--opt=disable-assertions コマンドライン引数を指定します。

コンパイラー最適化のヒント

インテル® ISPC 標準ライブラリーには、assume() 文をインテル® ISPC プログラムコードに追加するメカニズムが含まれています。assume() 関数は、単一の uniform ブール式を引数として受け取ります。この式は true であると想定され、可能であればこの情報は最適化に使用されます。

assume() 文で使用される条件ではコードは生成されず、ランタイムチェックは行われません。コンパイラーは、この情報を最適化のヒントとしてのみ使用します。

以下にこの機能の基本例を示します。

```
inline uniform int bar1(uniform int a, uniform int b) {
    if (a < b)
        return 2;
    return 5;
}
uniform int foo1(uniform int a, uniform int b) {
    assume(a < b);
    return bar1(a, b);
}
```

assume() ヒントによりコンパイラーは、コンパイル時に bar1() で a < b を解決して 2 を返すことが可能になるため分岐を排除できます。

```
inline void bar2(uniform int * uniform a) {
    if (a != NULL) {
        a[2] = 9;
    }
 }
 void foo2(uniform int a[]) {
    assume(a != NULL);
    bar2(a);
 }
```

assume() ヒントによりコンパイラーは、コンパイル時に bar2() で a != NULL を解決できるため分岐を排除できます。

```
int foo3(uniform int a[], uniform int count) {
    int ret = 0;
    assume(count % programCount == 0);
    foreach (i = 0 ... count) {
        ret += a[i];
    }
    return ret;
}
```

assume() ヒントは、コンパイル時にコンパイルカウントが programCount の倍数であることを通知します。これにより、foreach のリマインダーループを排除できます。

```
typedef float<TARGET_WIDTH> AlignedFloat;
unmasked void foo4(uniform float Result[], const uniform float Source1[], const
uniform unsigned int Iterations)
{
     assume(((uniform uint64)((void*)Source1) & (32 * TARGET_WIDTH)-1) == 0);
     assume(((uniform uint64)((void*)Result) & (32 * TARGET_WIDTH)-1) == 0);
     uniform AlignedFloat S1;
     S1[programIndex] = Source1[programIndex];
     const uniform AlignedFloat R = S1;
     Result[programIndex] = R[programIndex];
}
```

assume() ヒントは、ロードとストアで使用されるメモリー位置がアライメントされていることをコンパイラーに通知します。これにより、コンパイラーは非アライメント命令ではなくアライメント命令を生成できます。

プログラム間のインスタンス操作

インテル® ISPC プログラムは、異なるデータ要素に対して独立した計算を表現するために使用されます (真のデータ並列処理など)。プログラム・インスタンスが協調して結果を計算できると便利なこともあります。ここで説明するレーン間の操作は、ギャング内で実行されているプログラム・インスタンス間の通信向けのプリミティブを提供します。

lanemask() 関数は、現在実行中の SPMD プログラム・インスタンスを示す符号化した整数を返します。i 番目のビットは、i 番目のプログラム・インスタンスのレーンが現在アクティブである場合に設定されます。

```
uniform int lanemask()
```

あるプログラム・インスタンスからほかのすべてのプログラム・インスタンスに値をブロードキャストするには、broadcast() 関数を利用できます。index で指定されたプログラム・インスタンスの value パラメーターの値を、実行中のすべてのプログラム・インスタンスにブロードキャストします。

```
int8 broadcast(int8 value, uniform int index)
int16 broadcast(int16 value, uniform int index)
int32 broadcast(int32 value, uniform int index)
int64 broadcast(int64 value, uniform int index)
float16 broadcast(float16 value, uniform int index)
float broadcast(float value, uniform int index)
double broadcast(double value, uniform int index)
```

rotate() 関数を使用すると、各プログラム・インスタンスは、offset ステップ分離れた特定の値を検出することができます。例えば、8 レーンのターゲットにおいて、value が実行中のプログラム・インスタンスのギャング全体で (1、2、3、4、5、6、7、8) の値を持つ場合、rotate(value - 1) によって、最初のプログラム・インスタンスは値 8 を、2 番目は 1、3 番目は 2 を取得するように実行されます。提供される offset 値は正でも負でもよく、ギャングサイズよりも大きい可能性もあります (有効なオフセットを確実にするためマスクされます)。

```
int8 rotate(int8 value, uniform int offset)
int16 rotate(int16 value, uniform int offset)
int32 rotate(int32 value, uniform int offset)
int64 rotate(int64 value, uniform int offset)
float16 rotate(float16 value, uniform int offset)
float rotate(float value, uniform int offset)
double rotate(double value, uniform int offset)
```

shift() 関数を使用すると、各プログラム・インスタンスは、offset ステップ分離れた特定の値を検出することができます。これは rotate() に似ていますが、値は循環してシフトされません。必要に応じて 0 がシフトインされます。

```
int8 shift(int8 value, uniform int offset)
int16 shift(int16 value, uniform int offset)
int32 shift(int32 value, uniform int offset)
int64 shift(int64 value, uniform int offset)
float16 shift(float16 value, uniform int offset)
float shift(float value, uniform int offset)
double shift(double value, uniform int offset)
```

shuffle() 関数は、プログラム・インスタンス間の値を並べ替える (シャッフルする) 2 つのバリアントを提供します。最初のバージョンでは、permutation 値には、value 値を取得するプログラム・インスタンスを指定します。permutation に指定する値は、0 からギャングサイズまでの値でなければなりません。

```
int8 shuffle(int8 value, int permutation)
int16 shuffle(int16 value, int permutation)
int32 shuffle(int32 value, int permutation)
int64 shuffle(int64 value, int permutation)
float16 shuffle(float16 value, int permutation)
float shuffle(float value, int permutation)
double shuffle(double value, int permutation)
```

2 番目の shuffle() バージョンは、指定された 2 つの値を連結した拡張ベクトルの並べ替えを行います。つまり、permutation の要素 0 の値は、value0 の最初の要素に対応し、ギャングサイズ の 2 倍から 1 を引いた値が value1 の最後の要素に対応します。

```
int8 shuffle(int8 value0, int8 value1, int permutation)
int16 shuffle(int16 value0, int16 value1, int permutation)
int32 shuffle(int32 value0, int32 value1, int permutation)
int64 shuffle(int64 value0, int64 value1, int permutation)
float16 shuffle(float16 value0, float16 value1, int permutation)
float shuffle(float value0, float value1, int permutation)
double shuffle(double value0, double value1, int permutation)
```

最後に、SIMD レーンの値を抽出および設定するプリミティブ操作があります。この節で説明した、broadcast、rotate、shift および shuffle のすべての操作は、これらのルーチンから実装できますが、一般的にはそれほど効率的ではありません。これらのルーチンは、上記にないリダクションやレーン間の通信を実装するのに便利です。可変値が指定されると、extract() 関数は i 番目の要素を uniform 値として返します。

```
uniform bool extract(bool x, uniform int i)
uniform int8 extract(int8 x, uniform int i)
uniform int16 extract(int16 x, uniform int i)
uniform int32 extract(int32 x, uniform int i)
uniform int64 extract(int64 x, uniform int i)
uniform float16 extract(float16 x, uniform int i)
uniform float extract(float x, uniform int i)
uniform double extract(double x, uniform int i)
```

同様に、insert () は、x の i 番目の要素を値 v に置き換えた値を返します。

```
bool insert(bool x, uniform int i, uniform bool v)
int8 insert(int8 x, uniform int i, uniform int8 v)
int16 insert(int16 x, uniform int i, uniform int16 v)
int32 insert(int32 x, uniform int i, uniform int32 v)
int64 insert(int64 x, uniform int i, uniform int64 v)
float16 insert(float16 x, uniform int i, uniform float16 v)
float insert(float x, uniform int i, uniform float v)
double insert(double x, uniform int i, uniform double v)
```

リダクション

実行中のプログラム・インスタンス全体の条件を評価する多数のルーチンが用意されています。例えば、any() は指定された値 v が実行中の SPMD プログラム・インスタンスのいずれかで true であれば true、all() はすべてに対して true であれば true を返し、none() は v が常に false であれば true を返します。

```
uniform bool any(bool v)
uniform bool all(bool v)
uniform bool none(bool v)
```

また、プログラム・インスタンス間でさまざまなリダクション計算を実行できます。例えば、アクティブなプログラム・インスタンスのそれぞれで指定された値を reduce_add() 関数で加算できます。

```
uniform int16 reduce_add(int8 x)
uniform unsigned int16 reduce_add(unsigned int8 x)
uniform int32 reduce_add(int16 x)
uniform unsigned int32 reduce_add(unsigned int16 x)
uniform int64 reduce_add(int32 x)
uniform unsigned int64 reduce_add(unsigned int32 x)
uniform int64 reduce_add(int64 x)
uniform unsigned int64 reduce_add(unsigned int64 x)

uniform float16 reduce_add(float16 x)
uniform float reduce_add(float x)
uniform double reduce_add(double x)
```

また、実行中のすべてのプログラム・インスタンスに対して、指定された値の最小値を求める関数を使用できます。

```
uniform int32 reduce_min(int32 a)
uniform unsigned int32 reduce_min(unsigned int32 a)
uniform int64 reduce_min(int64 a)
uniform unsigned int64 reduce_min(unsigned int64 a)

uniform float16 reduce_min(float16 a)
uniform float reduce_min(float a)
uniform double reduce_min(double a)
```

アクティブなプログラム・インスタンスに対し、指定された可変変数の最大値を計算する同等の関数も利用できます。

```
uniform int32 reduce_max(int32 a)
uniform unsigned int32 reduce_max(unsigned int32 a)
uniform int64 reduce_max(int64 a)
uniform unsigned int64 reduce_max(unsigned int64 a)

uniform float16 reduce_max(float16 a)
uniform float reduce_max(float a)
uniform double reduce_max(double a)
```

最後に、指定された値が、実行中のすべてのプログラム・インスタンスで同じ値を持つか確認する reduce_equal() 関数も利用できます。

```
uniform bool reduce_equal(int32 v)
uniform bool reduce_equal(unsigned int32 v)
uniform bool reduce_equal(int64 v)
uniform bool reduce_equal(unsigned int64 v)

uniform bool reduce_equal(float16 v)
uniform bool reduce_equal(float v)
uniform bool reduce_equal(double)
```

また、これらの関数のバリアントとして、値がすべてのプログラム・インスタンスで同じ場合に uniform 値を返すものもあります。『インテル® ISPC パフォーマンス・ガイド』に、メモリーアクセスのパフォーマンスを向上するバリアントを応用する説明があります。

```
uniform bool reduce_equal(int32 v, uniform int32 * uniform sameval)
uniform bool reduce_equal(unsigned int32 v,
                          uniform unsigned int32 * uniform sameval)
uniform bool reduce_equal(int64 v, uniform int64 * uniform sameval)
uniform bool reduce_equal(unsigned int64 v,
                          uniform unsigned int64 * uniform sameval)

uniform bool reduce_equal(float16 v, uniform float16 * uniform sameval)
uniform bool reduce_equal(float v, uniform float * uniform sameval)
uniform bool reduce_equal(double, uniform double * uniform sameval)
```

いずれのプログラム・インスタンスも実行されていないときに呼び出されると、reduce_rqual() は false を返します。

また、プログラム・インスタンス間の値の scan を計算する関数も多数用意されています。例えば、exclusive_scan_add() 関数は、プログラム・インスタンスごとに、先行するすべてのプログラム・インスタンスで指定された値の合計を計算します。インテル® ISPC で現在利用可能なスキャンはすべて排他的スキャンです。つまり、特定の要素に対して計算された値には、その要素の値は含まれません。C コードでは、配列に対する排他的加算スキャンは次のように実装できます。

```
void scan_add(int *in_array, int *result_array, int count) {
    result_array[0] = 0;
    for (int i = 1; i < count; ++i)
        result_array[i] = result_array[i-1] + in_array[i-1];
}
```

インテル® ISPC では、加算、ビット単位の論理積、ビット単位の論理和のスキャン関数が利用できます。

```
int32 exclusive_scan_add(int32 v)
unsigned int32 exclusive_scan_add(unsigned int32 v)
float16 exclusive_scan_add(float16 v)
float exclusive_scan_add(float v)
int64 exclusive_scan_add(int64 v)
unsigned int64 exclusive_scan_add(unsigned int64 v)
double exclusive_scan_add(double v)
int32 exclusive_scan_and(int32 v)
unsigned int32 exclusive_scan_and(unsigned int32 v)
int64 exclusive_scan_and(int64 v)
unsigned int64 exclusive_scan_and(unsigned int64 v)
int32 exclusive_scan_or(int32 v)
unsigned int32 exclusive_scan_or(unsigned int32 v)
int64 exclusive_scan_or(int64 v)
unsigned int64 exclusive_scan_or(unsigned int64 v)
```

最初のプログラム・インスタンスの戻り値は、exclusive_scan_add と exclusive_scan_or では 0 になり、exclusive_scan_and ではすべてのビットが 1 に設定されます。

排他的スキャンを使用して、プログラム・インスタンスからコンパクトな出力バッファーに可変長の出力を生成する方法は、FAQ (英語) で説明されています。

スタックメモリー割り当て

インテル® ISPC 標準ライブラリーには、スタックにメモリーを割り当てる際に使用できる alloca() 関数が含まれます。

```
void * uniform alloca(uniform size_t size);
```

alloca() 関数は、呼び出し元のスタックフレームに size バイトの空間を割り当てます。alloca() を呼び出した関数が呼び出し元にリターンすると、この一時的な空間は自動的に解放されます。

データ移動

メモリーの値設定とコピー

メモリーブロックをコピーしたり、メモリー内の値を初期化する関数がいくつか用意されています。memcpy は、C 標準ライブラリーの同名のルーチンによって、メモリーの src 位置から始まる指定された count バイト数を、2 つのメモリー領域が重複しないことを保証される dst 位置にコピーします。バッファーが重複する可能性がある場合、memmove を使用してデータをコピーすることもできます。

```
void memcpy(void * uniform dst, void * uniform src, uniform int32 count)
void memmove(void * uniform dst, void * uniform src, uniform int32 count)
void memcpy(void * varying dst, void * varying src, int32 count)
void memmove(void * varying dst, void * varying src, int32 count)
```

これらの関数には、uniform と varying ポインターの両方を受け取るものがあることに注意してください。また、sizeof(float) と sizeof(uniform float) は異なる値を返すため、プログラマーは count 値の計算に注意が必要です。

メモリーの値を初期化するには、memset ルーチンを利用できます。C 標準ライブラリーの同名の関数と同様な動作を期待できます。これは、指定された位置から始まるメモリーから、指定されるバイト数を、指定された値に設定します。

```
void memset(void * uniform ptr, uniform int8 val, uniform int32 count)
void memset(void * varying ptr, int8 val, int32 count)
```

これらの関数には、操作するメモリーのバイト数として 64 ビット値を受け取るバリアントもあります。

```
void memcpy64(void * uniform dst, void * uniform src, uniform int64 count)
void memcpy64(void * varying dst, void * varying src, int64 count)
void memmove64(void * uniform dst, void * uniform src, uniform int64 count)
void memmove64(void * varying dst, void * varying src, int64 count)
void memset64(void * uniform ptr, uniform int8 val, uniform int64 count)
void memset64(void * varying ptr, int8 val, int64 count)
```

パックドロードとストア操作

標準ライブラリーは、アクティブなプログラム・インスタンスのリニアなメモリー位置の値を読み書きするルーチンも提供します。packed_load_active() 関数は、指定された位置から始まる連続した値をロードし、実行中のプログラム・インスタンスごとに 1 つの連続した値をロードして、そのプログラム・インスタンスの val 変数にストアします。ロードされた値の総数を返します。

```
uniform int packed_load_active(uniform int * uniform base,
                               varying int * uniform val)
uniform int packed_load_active(uniform unsigned int * uniform base,
                               varying unsigned int * uniform val)
```

同様に、packed_store_active() 関数は、packed_store_active() 呼び出しを実行した各プログラム・インスタンスの val 値をストアし、指定された位置から連続して結果を保存します。ストアされた値の総数を返します。

```
uniform int packed_store_active(uniform int * uniform base,
                                int val)
uniform int packed_store_active(uniform unsigned int * uniform base,
                                unsigned int val)
```

また、packed_store_active2() 関数は、出力配列に 1 つ余分に要素を書き込む (ただし、packed_store_active() と同じ値を返す) 以外は全く同じ署名と同じセマンティクスを持っています。これらの関数は、サポートされるほとんどのターゲットで異なる分岐のない実装を提案しており、通常は (常にではありませんが) packed_store_active() よりも良いパフォーマンスを示します。使用する前に、特定のターゲット・ハードウェアでユーザーのシナリオで関数のパフォーマンスをテストすることを推奨します。

これらの関数の使用方法として、次のコードに packed_store_active() の例を示します。

```
uniform int negative_indices(uniform float a[], uniform int length,
                             uniform int indices[]) {
    uniform int numNeg = 0;
    foreach (i = 0 ... length) {
        if (a[i] < 0.)
            numNeg += packed_store_active(&indices[numNeg], i);
    }
    return numNeg;
}
```

この関数は、浮動小数点数の配列 a を受け取り、その長さは length パラメーターで指定されます。この関数はまた、少なくとも length と同じ長さの出力配列 indices を受け取ります。次に、a のすべての要素をループし、ゼロ未満の要素ごとに、その要素のオフセットを indices 配列に格納します。負の値の総数を返します。例えば、入力配列 a[8] = { 10, -20, 30, -40, -50, -60, 70, 80 } が与えられた場合、4 つの負の値のカウントを返し、indices[] の最初の 4 要素を a[i] が 0 より小さい配列のインデックスに対応する値 { 1, 3, 4, 5 } に初期化します。

ストリーミング・ロードとストア操作

標準ライブラリーは、ストリーミング・ロードとストリーミング・ストア操作を行うルーチンを提供します。この実装は、ストリーミングだけでなく、非テンポラルな操作としても機能します。uniform 変数または varying 変数のロードやストアによって、使用するルーチンが異なります。

ストリーミング・ストアの種類を以下に示します。

varying 変数から配列にストアする場合:

```
void streaming_store(uniform unsigned int8 a[], unsigned int8 vals)
void streaming_store(uniform int8 a[], int8 vals)
void streaming_store(uniform unsigned int16 a[], unsigned int16 vals)
void streaming_store(uniform int16 a[], int16 vals)
void streaming_store(uniform unsigned int a[], unsigned int vals)
void streaming_store(uniform int a[], int vals)
void streaming_store(uniform unsigned int64 a[], unsigned int64 vals)
void streaming_store(uniform int64 a[], int64 vals)
void streaming_store(uniform float16 a[], float16 vals)
void streaming_store(uniform float a[], float vals)
void streaming_store(uniform double a[], double vals)
```

uniform 変数から配列にストアする場合:

```
void streaming_store(uniform unsigned int8 a[], uniform unsigned int8 vals)
void streaming_store(uniform int8 a[], uniform int8 vals)
void streaming_store(uniform unsigned int16 a[], uniform unsigned int16 vals)
void streaming_store(uniform int16 a[], uniform int16 vals)
void streaming_store(uniform unsigned int a[], uniform unsigned int vals)
void streaming_store(uniform int a[], uniform int vals)
void streaming_store(uniform unsigned int64 a[], uniform unsigned int64 vals)
void streaming_store(uniform int64 a[], uniform int64 vals)
void streaming_store(uniform float16 a[], uniform float16 vals)
void streaming_store(uniform float a[], uniform float vals)
void streaming_store(uniform double a[], uniform double vals)
```

ストリーミング・ロードの種類を以下に示します。

配列から varying 値をロード:

```
varying unsigned int8 streaming_load(uniform unsigned int8 a[])
varying int8 streaming_load(uniform int8 a[])
varying unsigned int16 streaming_load(uniform unsigned int16 a[])
varying int16 streaming_load(uniform int16 a[])
varying unsigned int streaming_load(uniform unsigned int a[])
varying int streaming_load(uniform int a[])
varying unsigned int64 streaming_load(uniform unsigned int64 a[])
varying int64 streaming_load(uniform int64 a[])
varying float16 streaming_load(uniform float16 a[])
varying float streaming_load(uniform float a[])
varying double streaming_load(uniform double a[])
```

配列から uniform 値をロード:

```
uniform unsigned int8 streaming_load_uniform(uniform unsigned int8 a[])
uniform int8 streaming_load_uniform(uniform int8 a[])
uniform unsigned int16 streaming_load_uniform(uniform unsigned int16 a[])
uniform int16 streaming_load_uniform(uniform int16 a[])
uniform unsigned int streaming_load_uniform(uniform unsigned int a[])
uniform int streaming_load_uniform(uniform int a[])
uniform unsigned int64 streaming_load_uniform(uniform unsigned int64 a[])
uniform int64 streaming_load_uniform(uniform int64 a[])
uniform float16 streaming_load_uniform(uniform float16 a[])
uniform float streaming_load_uniform(uniform float a[])
uniform double streaming_load_uniform(uniform double a[])
```

データ変換

構造体配列と配列構造体間のレイアウト変換

アプリケーションでは、メモリーに「構造体配列」形式でデータを格納することが多く見受けられます。C/C++ コードでは便利ですが、このレイアウトでは、インテル® ISPC プログラムが「配列構造体」形式でデータを配置するよりも効率が悪くなる可能性があります。このトピックの詳細は、『インテル® ISPC パフォーマンス・ガイド』の「可能な場合は配列構造体レイアウトを使用する」を参照してください。

標準ライブラリーには、この 2 つの形式を効率的に変換する関数が用意されており、アプリケーションを変更して「配列の構造レイアウト」を使用することができない場合に対応できます。3 次元 (x,y,z) の位置データを C の配列に並べたものを考えてみましょう。

```
// C++ コード
float pos[] = { x0, y0, z0, x1, y1, z1, x2, ... };
```

インテル® ISPC プログラムでは、(x,y,z) 値の集合を読み込み、それに基づいて計算を行う場合があります。以下にその自然な表現をします。

```
extern uniform float pos[];
uniform int base = ...;
float x = pos[base + 3 * programIndex];     // x = { x0 x1 x2 ... }
float y = pos[base + 1 + 3 * programIndex]; // y = { y0 y1 y2 ... }
float z = pos[base + 2 + 3 * programIndex]; // z = { z0 z1 z2 ... }
```

メモリーアクセスが不規則になり、パフォーマンスが低下します。また、aos_to_soa3() 標準ライブラリー関数を使用することもできます。

```
extern uniform float pos[];
uniform int base = ...;
float x, y, z;
aos_to_soa3(&pos[base], &x, &y, &z);
```

このルーチンは、与えられたオフセットの配列からギャングサイズの 3 倍の値をロードし、3 つの異なる結果を返します。この関数の int32、int64、float および double のバリアントがあります。

```
void aos_to_soa3(uniform float a[], varying float * uniform v0,
               varying float * uniform v1, varying float * uniform v2)
void aos_to_soa3(uniform int32 a[], varying int32 * uniform v0,
               varying int32 * uniform v1, varying int32 * uniform v2)
void aos_to_soa3(uniform double a[], varying double * uniform v0,
               varying double * uniform v1, varying double * uniform v2)
void aos_to_soa3(uniform int64 a[], varying int64 * uniform v0,
               varying int64 * uniform v1, varying int64 * uniform v2)
```

計算が完了すると、対応する関数がインテル® ISPC 可変変数の SoA 値から逆変換し、配列の指定されたオフセットに値を書き戻します。

```
extern uniform float pos[];
uniform int base = ...;
float x, y, z;
aos_to_soa3(&pos[base], &x, &y, &z);
// x、y、z の計算を実行
soa_to_aos3(x, y, z, &pos[base]);
void soa_to_aos3(float v0, float v1, float v2, uniform float a[])
void soa_to_aos3(int32 v0, int32 v1, int32 v2, uniform int32 a[])
void soa_to_aos3(double v0, double v1, double v2, uniform double a[])
void soa_to_aos3(int64 v0, int64 v1, int64 v2, uniform int64 a[])
```

これらの関数は、現在のプログラムの実行マスクを考慮しないことに注意してください。それらギャングサイズの 3 倍を無条件にリード/ライトします。したがって、反復回数がプログラム・カウントの整数倍でない場合、aos_to_soa3() は入力データの終端を越えて読み込み、soa_to_aos3() は出力データの終端を越えて書き込みます。この場合、メモリー破壊を回避するために、以下のいずれかの方法を取ることができます。

- データバッファーのサイズが programCount の倍数であることを確認し、リード/ライトのオーバーフローがメモリー破壊を起こさないようにします。
- メインループでは、反復回数が programCount の倍数になるようにマスクし、残りの反復回数に対して手動で「リマインダー」ループ (gather/scatter を使用) を追加します。

また、AoS と SoA レイアウト間で 4 レーンの値や 2 レーンの値を変換する関数のバリアントがあります。つまり、aos_to_soa4() は、r0 g0 b0 a0 r1 g1 b1 a1 ... のように配置されたメモリー内の AoS データを、r0 r1...、g0 g1...、b0 b1...、という値の 4 つの可変変数に変換します。そして a0 a1...、の指定されたオフセットの配列からギャングサイズ値の合計 4 倍を読み取ります。

```
void aos_to_soa4(uniform float a[], varying float * uniform v0,
               varying float * uniform v1, varying float * uniform v2,
               varying float * uniform v3)
void aos_to_soa4(uniform int32 a[], varying int32 * uniform v0,
               varying int32 * uniform v1, varying int32 * uniform v2,
               varying int32 * uniform v3)
void soa_to_aos4(float v0, float v1, float v2, float v3, uniform float a[])
void soa_to_aos4(int32 v0, int32 v1, int32 v2, int32 v3, uniform int32 a[])
```

また、これらの関数の 2 レーンのバリアントもサポートされています。

```
void aos_to_soa2(uniform float a[], varying float * uniform v0,
               varying float * uniform v1)
void aos_to_soa2(uniform int32 a[], varying int32 * uniform v0,
               varying int32 * uniform v1)
void soa_to_aos2(float v0, float v1, uniform float a[])
void soa_to_aos2(int32 v0, int32 v1, uniform int32 a[])
```

半精度浮動小数点の変換

IEEE 16 ビット浮動小数点フォーマットの変換関数があります。インテル® ISPC では float16 データタイプが言語と標準ライブラリーで完全にサポートされていますが、これは float16 データタイプをハードウェアがサポートするターゲット上でのみであることに注意してください。以下の関数は、メモリー上の半精度データとの変換を容易にするもので、主にハードウェアに float16 のネイティブサポートがないターゲットでの使用を対象としています。

これらを使用するには、半精度データを int16 にロードし、half_to_float() 関数で 32 ビット浮動小数点値に変換する必要があります。値を半精度形式でメモリーに格納するため、float_to_half() 関数は、与えられた float に最も近い 16 ビットを半精度で返します。

```
float half_to_float(unsigned int16 h)
uniform float half_to_float(uniform unsigned int16 h)
int16 float_to_half(float f)
uniform int16 float_to_half(uniform float f)
```

また、これらの関数には、浮動小数点数の無限大、「NaN (数ではない)」、非正規化された数を正しく処理することを気にしない、より高速なバージョンもあります。これらは上記の関数より高速ですが、精度は落ちます。

```
float half_to_float_fast(unsigned int16 h)
uniform float half_to_float_fast(uniform unsigned int16 h)
int16 float_to_half_fast(float f)
uniform int16 float_to_half_fast(uniform float f)
```

sRGB8 への変換

sRGB 色空間は、グラフィックスやイメージングの多くのアプリケーションで使用されています。詳細は、Wikipedia の sRGB (英語) のページを参照してください。インテル® ISPC 標準ライブラリーには、浮動小数点タイプの色値を sRGB 空間の 8 ビット値に変換する 2 つの関数が用意されています。

```
int float_to_srgb8(float v)
uniform int float_to_srgb8(uniform float v)
```

システム・プログラミングのサポート

アトミック操作とメモリーフェンス

アトミックメモリー操作の標準セットは、標準ライブラリーで提供されますが、uniform と varying タイプの両方、および「ローカル」と「グローバル」のアトミック操作を行うバリアントもあります。

ローカルアトミックは、ギャング内のプログラム・インスタンス間でアトミック操作を提供しますが、複数のギャングや異なるハードウェア・スレッドでのメモリー操作にまたがっては動作しません。それらが必要な理由を理解するため、ヒストグラムの計算で、各プログラムがある値をどのバケットに入れるかを計算し、それに対応するカウンターをインクリメントする処理を考えてみます。コードが次のように記述されている場合、プログラムの動作は未定義です。

```
uniform int count[N_BUCKETS] = ...;
float value = ...;
int bucket = clamp(value / N_BUCKETS, 0, N_BUCKETS);
++count[bucket];  // ERROR: 競合した場合の動作は未定義
```

複数のプログラム・インスタンスが同じ値のバケットに対応する値を保持する場合、インクリメントの影響は不定です。「ギャング内のデータ競合」の説明を参照してください。この場合、count[bucket] を更新するプログラム・インスタンスとその値を読み取るほかのプログラム・インスタンスの間にシーケンスポイントはありません。

この場合、atomic_add_local() 関数を使用することができます。ローカルアトミックは、プログラム・インスタンスのギャング間でアトミックであり、期待される結果が計算されます。

```
...
int bucket = clamp(value / N_BUCKETS, 0, N_BUCKETS);
atomic_add_local(&count[bucket], 1);
```

この 32 ビット整数のアトミック加算ルーチンのバリアントを使用します。

```
int32 atomic_add_local(uniform int32 * uniform ptr, int32 delta)
```

このルーチンのセマンティクスはアトミックな加算関数として典型的なものです。このポインターはメモリー上の 1 つの位置 (すべてのプログラム・インスタンスで同じもの) を指し、実行中の各プログラム・インスタンスでは、ptr が指す場所に格納されている値にそのプログラム・インスタンスの値 delta がアトミックに加算され、古い値がこの関数から返されます。

注意すべきことは、加算される値のタイプが uniform 整数であるのに対し、インクリメント量と戻り値が変化することです。つまり、この呼び出しのセマンティクスは、実行中のプログラム・インスタンスが個々のデルタ値でアトミック操作を行い、その戻り値として前の値を取得するものです。実行中のプログラム・インスタンスに対するアトミックは、任意の順序で発行できます。つまり、programIndex 順に発行されることは保証されません。

グローバルアトミックは、ローカルアトミックよりも強力であり、ギャング内のプログラム・インスタンスの両方にアトミックであり、また異なるギャングや異なるハードウェア・スレッドにまたがってもアトミックです。以下に、上記で使用したアトミックのグローバルバリアントを示します。

```
int32 atomic_add_global(uniform int32 * uniform ptr, int32 delta)
```

複数のプロセッサーが同時に同じメモリー位置にアトミックな加算を行う場合、加算はハードウェアによってシリアル化され、最終的に正しい結果が計算されます。

以下に、これらの関数の int32 タイプの宣言を示します。また、符号なし int32 や int64 の値を受け取る int64 の同等なバリアントもあります。

```
int32 atomic_add_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_subtract_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_min_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_max_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_and_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_or_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_xor_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_swap_{local,global}(uniform int32 * uniform ptr, int32 value)
```

float と double タイプのサポートもあります。ローカルアトミックの場合、論理演算以外はすべて利用できます。ここには示されませんが、これらに対応する double のバリアントもあります。

```
float atomic_add_local(uniform float * uniform ptr, float value)
float atomic_subtract_local(uniform float * uniform ptr, float value)
float atomic_min_local(uniform float * uniform ptr, float value)
float atomic_max_local(uniform float * uniform ptr, float value)
float atomic_swap_local(uniform float * uniform ptr, float value)
```

グローバルアトミックの場合、これらのタイプのアトミックスワップのみが利用できます。

```
float atomic_swap_global(uniform float * uniform ptr, float value)
double atomic_swap_global(uniform double * uniform ptr, double value)
```

最後に、"swap" (ただし、ほかのアトミックは使用できません) は、ポインタータイプで使用できます。

```
void *atomic_swap_{local,global}(void * * uniform ptr, void * value)
```

オペランドに uniform な値を受け取り、uniform な結果を返すアトミックのバリアントもあります。これは、プログラム・インスタンスごとに実行されるのではなく、プログラム・インスタンスのギャング全体に対して実行される単一のアトミックに相当します。

```
uniform int32 atomic_add_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_subtract_{local,global}(uniform int32 * uniform ptr,
                                             uniform int32 value)
uniform int32 atomic_min_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_max_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_and_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_or_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_xor_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_swap_{local,global}(uniform int32 * uniform ptr,
                                         uniform int32 newval)
```

また、ポインターにおいても同様です。

```
uniform void *atomic_swap_{local,global}(void * * uniform ptr,
                                         void *newval)
```

アトミック関数を意図したように使用するように注意してください。次のコードについて考えてみます。

```
extern uniform int32 counter;
int32 myCounter = atomic_add_global(&counter, 1);
```

このように、実行中のプログラムの各インスタンスがカウンターを 1 つずつインクリメントし、カウンターの古い値を取得するようなコードを書くこともできます (例えば、結果を配列の一意の位置に格納する場合など)。しかし、上記のコードでは、atomic_add_global() の 2 番目のバリアントが呼び出され、カウンターに追加する uniform int 値を取得し、1 つのアトミック操作のみが実行されます。カウンターは 1 つだけインクリメントし、すべてのプログラム・インスタンスは同じ値を返します (uniform int32 の戻り値が varying int32 に暗黙的に変換されるためです)。例えば、次のように記述すると、必要とする atomic add 関数が呼ばれるようになります。

```
extern uniform int32 counter;
int32 myCounter = atomic_add_global(&counter, (varying int32)1);
```

これらの各アトミック関数には、可変ポインターを受け取る 3 つ目のバリアントがあります。これにより、各プログラム・インスタンスはメモリー内の別の場所にアトミック操作を発行できます。それらの一部または全部が、メモリー内の同じ位置を指していたとしても、適切な結果が返されます。

```
int32 atomic_add_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_subtract_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_min_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_max_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_and_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_or_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_xor_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_swap_{local,global}(uniform int32 * varying ptr, int32 value)
```

および

```
void *atomic_swap_{local,global}(void * * ptr, void *value)
```

アトミックな「比較と交換」関数もあります。比較と交換は、val の値を compare とアトミックに比較します。一致する場合は、newval を val に割り当てます。いずれの場合も val の古い値が返されます。ほかのアトミック操作と同様に、この関数には符号なしの 64 ビットバージョンもあります。さらに、float、double および void* のバリアントもあります。

```
int32 atomic_compare_exchange_{local,global}(uniform int32 * uniform ptr,
                                             int32 compare, int32 newval)
uniform int32 atomic_compare_exchange_{local,global}(uniform int32 * uniform ptr,
                                  uniform int32 compare, uniform int32 newval)
```

インテル® ISPC には、メモリーバリアをコードに挿入する標準ライブラリー・ルーチンもあります。これにより、バリアが発行される前のすべてのメモリーのリードとライトが、バリアが発行される前に完了することが保証されます。マルチスレッド・コードにおけるメモリーバリアの必要性と使用法については、「Linux* カーネルのメモリーバリア」(英語) に関するドキュメントを参照してください。

```
void memory_barrier();
```

このバリアは、ギャング内のプログラム・インスタンス間のリードとライトの調整には必要がないことに注意してください。異なるコアで実行される複数のハードウェア・スレッド間で調整する場合にのみ必要です。ギャング全体のメモリーのリードとライトで保証される順番については、「ギャング内のデータ競合」を参照してください。

プリフェッチ

標準ライブラリーには、プロセッサーのキャッシュにデータをプリフェッチするさまざまな関数があります。現代の CPU には自動プリフェッチ機能が備わっており、必要になるデータを事前にキャッシュにプリフェッチできますが、ハイパフォーマンス・アプリケーションでは、この機能が有効である場合があります。

例えば、次のコードでは、配列の項目を反復処理する際に、プロセッサーの L1 キャッシュにデータをプリフェッチする方法を示しています。

```
uniform int32 array[...];
for (uniform int i = 0; i < count; ++i) {
    // array[i] の計算を実行
    prefetch_l1(&array[i+32]);
}
```

標準ライブラリーには、L1、L2、L3 キャッシュにプリフェッチを行うルーチンが用意されています。これには、prefetch_nt() というバリアントもありますが、これはプリフェッチされる値が複数回利用されることがないことを示します (したがって、キャッシュから排出される可能性も高くなります)。さらに、これらの関数には、uniform と varying ポインタータイプの両方を受け取るバージョンもあります。

```
void prefetch_{l1,l2,l3,nt}(void * uniform ptr)
void prefetch_{l1,l2,l3,nt}(void * varying ptr)
```

標準ライブラリーには、書き込みを想定して L1、L2、L3 キャッシュにプリフェッチを行うルーチンが用意されています。

```
void prefetchw_{l1,l2,l3}(void * uniform ptr)
void prefetchw_{l1,l2,l3}(void * varying ptr)
```

システム情報

高精度のハードウェア・クロック・カウンターの値は、clock() ルーチンで返され、その値はプロセッサー・サイクルごとに 1 つずつ増加します。したがって、プログラム実行中の異なる時点で clock() が返す値の差分を取ることで、その間のサイクル数が分かります。

```
uniform int64 clock()
```

clock() はプロセッサーのパイプラインをフラッシュすることに注意してください。これには 100 サイクルほどのコストがかかるので、細かい精度の測定を行う場合は、clock() を呼び出すコストを測定して、報告された結果からその値を差し引くと良いでしょう。

また、システムで利用可能な CPU コア数を求めるルーチンも用意されています。

```
uniform int num_cores()
```

この値は、システムのプロセッサー数に応じて並列タスクの分解粒度を適合させるのに有用です。

アプリケーションとの相互運用性

インテル® ISPC の主な目的の 1 つは、C/C++ アプリケーション・コードとインテル® ISPC で記述された並列コード間の相互運用を容易にすることです。ここでは、この仕組みの詳細といくつかの問題点について説明します。

相互運用性の概要

「インテル® ISPC プログラムのコンパイルと実行」で説明されているように、C/C++ コードからインテル® ISPC コードを呼び出すのは比較的容易です。最初に、呼び出されるすべてのインテル® ISPC 関数を export キーワードで定義します。

```
export void foo(uniform float a[]) {
    ...
}
```

この関数は、次の C の呼び出し可能関数に対応します。

```
void foo(float a[]);
```

(「uniform と varying 修飾子」で説明したように、uniform タイプは C/C++タイプの単一インスタンスに対応します。)

関数呼び出しでアプリケーションからインテル® ISPC に渡される変数と、アプリケーションとインテル® ISPC 間でグローバル変数を共有することができます。これを可能にするには、グローバル変数を通常のように宣言し (インテル® ISPC またはアプリケーション・コードで)、他方に extern 宣言を追加します。

例えば、次のインテル® ISPC コードを考えます。

```
// ispc コード
uniform float foo;
extern uniform float bar[10];
```

同等の C++ コード:

```
// C++ コード
extern "C" {
  extern float foo;
  float bar[10];
}
```

foo と bar グローバル変数の両方は、双方でアクセスできます。インテル® ISPC は関数とグローバルに C リンクを使用するため、C++ では extern “C” 宣言が必要であることに注意してください。

インテル® ISPC コードはまた、C/C++ にコールバックできます。インテル® ISPC では、呼び出すアプリケーションの関数を extern "C" 修飾子で宣言する必要があります。

```
extern "C" void foo(uniform float f, uniform float g);
```

C++ とは異なり、extern "C" では宣言する複数の関数を中括弧で区切らないため、インテル® ISPC から呼び出す複数の C 関数は、次のように宣言する必要があります。

```
extern "C" void foo(uniform float f, uniform float g);
extern "C" uniform int bar(uniform int a);
```

extern "C" リンクで宣言された関数をオーバーロードするのは違法です。インテル® ISPC はエラーを通知します。
extern "C" リンクで宣言された関数は、__vectorcall 修飾子を加えることで、Windows* の __vectorcall 呼び出し規約に準拠できます。

```
extern "C" __vectorcall void foo(uniform float f, uniform float g);
```

__vectorcall は、extern "C" 関数宣言と Windows* でのみ使用できます。

実行中のプログラム・インスタンスのギャングに対し、1 つの関数呼び出しだけが C++ に戻ります。さらに、プログラム・インスタンスのいずれも呼び出しを行わない場合、関数は C/C++ のコールバックを行いません。次のコードを考えてみます。

```
uniform float foo = ...;
float x = ...;
if (x != 0)
    foo = appFunc(foo);
```

appFunc() は、実行中のプログラム・インスタンスの 1 つ以上が、x != 0 を true と評価した場合にのみ呼び出されます。アプリケーション・コードが、実行中のプログラム・インスタンスのいずれかを呼び出すか判断する場合、アクティブな SIMD レーンを示すマスクを関数に渡すことができます。

```
extern "C" float appFunc(uniform float x,
                         uniform int activeLanes);
```

関数が次のように呼び出されると想定します。

```
...
x = appFunc(x, lanemask());
```

この時点で、コードの最初のプログラム・インスタンスが実行されている場合 activeLanes パラメーターの値はビット 0 が 1 になり、2 番目のインスタンスではビット 1 が 1 になります。lanemask() 関数については、「プログラム間のインスタンス操作」で説明します。アプリケーション・コードは、次にように記述できます。

```
float appFunc(float x, int activeLanes) {
    for (int i = 0; i < programCount; ++i)
        if ((activeLanes & (1 << i)) != 0) {
            // i 番目の SIMD レーンの計算を実行
        }
}
```

場合によっては、ギャングに対し 1 つの呼び出しを行うのではなく、プログラム・インスタンスに対して 1 つの呼び出しを行うことが適切です。例えば、次のコードはスカラー・パラメーターを受け取る既存の数学ライブラリー・ルーチンをどのように呼び出すかを示しています。

```
extern "C" uniform double erf(uniform double);
double v = ...;
double result;
foreach_active (instance) {
    uniform double r = erf(extract(v, instance));
    result = insert(result, instance, r);
}
```

このコードでは、アクティブなプログラム・インスタンスごとに erf() を一度呼び出し、プログラム・インスタンスの v の値を渡して、その結果をインスタンスの result に格納します。

データレイアウト

一般にインテル® ISPC は、構造体タイプやそのほかの複雑なデータタイプを C/C++ と同じようにメモリーに配置します。C/C++ コートとインテル® ISPC コード間の相互参照を容易にするため、構造体のレイアウトを一致させることが重要です。

インテル® ISPC と C/C++ 間でデータを共有する多くの複雑性は、インテル® ISPC コードとアプリケーション・コード間のデータ構造の調整から生じます。インテル® ISPC コードで共有構造体を宣言してから、生成されたヘッダーファイルを調査すると便利です (C/C++ に相当するものがあります)。次の構造体の例を考えてみます。

```
// ispc コード
struct Node {
   int count;
   float pos[3];
};
```

export される関数へのパラメーターで均一な Node 構造体が使用されている場合、インテル® ISPC によって生成されるヘッダーファイルには次のような宣言が含まれます。

```
// C/C++ コード
struct Node {
   int count;
   float pos[3];
};
```

varying タイプのサイズは、プログラム・インスタンスのギャングのサイズに依存するため、インテル® ISPC には export 修飾子を持つ関数のパラメーターで varying タイプを使用することに関する制限があります。インテル® ISPC は、パラメーターがポインタータイプでない限り、export された関数へのパラメーターが varying タイプを持つことを禁止しています。つまり、varying float は許可されませんが、varying *float uniform (varying float への uniform ポインター) は許可されます。プログラマーは、varying データへのポインターを介してアクセスされるデータが正しい構成になるよう注意する必要があります。

同様に、アプリケーションと共有される構造体タイプがポインターを持つこともできます。

```
// C コード
struct Foo {
    float *foo, *bar;
};
```

インテル® ISPC では、対応する構造体宣言は次のようになります。

```
// ispc コード
struct Foo {
    float * uniform foo, * uniform bar;
};
```

エクスポートされる関数に、可変構造体タイプへのポインターがあると、生成されるヘッダーファイルの定義は、次のようになります (8 レーンの SIMD の場合)。

```
// C/C++ コード
struct Node {
  int count[8];
  float pos[3][8];
};
```

マルチターゲット・コンパイルの場合、インテル® ISPC は複数のヘッダーファイルと、複数のサイズに対応した定義を持つ「汎用」ヘッダーファイルを生成します。エクスポートされた関数内の変数へのポインターは、void* に書き換えられます。実行時に、インテル® ISPC のディスパッチ機能がこれらのポインターを適切なタイプにキャストします。プログラマーは、programCount の値を返すエクスポートされた関数を作成するだけで、インテル® ISPC によって実行時に使用されるギャング幅を決定するメカニズムを、C/C++ コードに提供できます。このような関数の例は、インテル® ISPC の配布物の examples/util/util.isph ファイルに含まれています。

インテル® ISPC は、uniform (均一な) ショート・ベクトル・タイプを、最初の要素がマシンの自然なベクトルにアライメントされるようメモリーに格納します (インテル® SSE ターゲットでは 16 バイトなど)。つまり、これらのタイプは、コンパイルターゲットによってレイアウトが異なります。そのため、C/C++ のアプリケーション・コードから uniform なショート・ベクトル・タイプにアクセスするのは避けるべきです。

データのアライメントとエイリアス

アプリケーションからインテル® ISPC プログラムにポインターを渡す際に、遵守すべき 2 つの制約があります。

1 つは、インテル® ISPC に渡される配列の先頭要素のメモリーを読み取ることが正当であることです。これは当然のことですが、アプリケーションから呼び出されるインテル® ISPC 関数に NULL ポインターを渡すことは違法であることを意味します。

2 つ目は、インテル® ISPC プログラム内のポインターと参照は別名であってはならないことです。インテル® ISPC は、異なるポインターが同じメモリー位置を指すことがないと想定します。これは、初期値が同じである、またはプログラム実行時の配列インテックス操作が原因です。

このエイリアシングの制約は、関数への参照パラメーターにも適用されます。次のような関数を考えてみます。

```
void func(int &a, int &b) {
    a = 0;
    if (b == 0) { ... }
}
```

この場合、同じ変数を func() に渡してはなりません。これはエイリアシングのもう 1 つの例です。呼び出し元が関数 func(x, x) として呼び出す場合、エイリアシングが行われないとコンパイラーの要件によって、if 文が true と評価されるとは限りません。

将来のインテル® ISPC では、ポインターがエイリアス化される可能性を示すメカニズムが実装されます。

インテル® ISPC を使用するための既存のプログラミングの再構築

インテル® ISPC は、既存のプログラムコードに最小限の修正を加えることで SPMD 並列処理を組み込むことができるように設計されています。C/C++ とインテル® ISPC のコード間でメモリーやデータ構造を共有する機能、インテル® ISPC と C/C++ 間で実行を直接移行できる機能は、それらを達成する重要な機能です。また、これらはインテル® ISPC を採用するプログラムへの段階的な移行も容易にし、最も計算量の多い領域をインテル® ISPC コードに変換し、それ以外はそのまま残すこともできます。

あるコード領域をインテル® ISPC で実行するようにしたい場合、次の課題は計算をどのように並列化するか決定することです。一般的に、大量の計算が内部で行われるループ (ray ごと、pixel ごとなど) がある場合、それらを SPMD 計算形式にマッピングすると、良い効果が得られます。

SPMD プログラム・インスタンスへの正確な計算のマッピングは、慎重に選択する必要があります。この選択は、gather/scatter メモリーアクセスとコヒーレント・メモリー・アクセスの組み合わせに影響する場合があります。詳細は、『インテル® ISPC パフォーマンス・ガイド』のトピックを参照してください。この決定は、実行中の SPMD プログラム・インスタンス全体の制御フローの一貫性に影響する可能性があり、パフォーマンスにも影響します。一般に、SPMD プログラム・インスタンス全体で同様の計算を行う傾向があるワークのグループを構成すると、パフォーマンスが向上します。

製品および性能に関する情報

性能は、使用状況、構成、その他の要因によって異なります。詳細については、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。