SYCL* 2020 で追加された 5 つの優れた機能

SYCL* プログラミング言語が進化

James Brodman インテル コーポレーション 主席エンジニア
John Pennycook インテル コーポレーション ソフトウェア・イネーブリング & 最適化アーキテクト

SYCL* 2020 は、アクセラレーターの活用を考えている C++ プログラマーにとって、エキサイティングなアップデートです。私たち 2 人は、SYCL* 仕様、SYCL* に関する書籍（英語）、SYCL* を LLVM に実装する DPC++ オープンソース・プロジェクト（英語）に貢献する機会に恵まれました。ここでは、SYCL* 2020（英語）で追加された新機能の中で、私たちが特に気に入っているものを紹介します。これは、Khronos を代表してではなく、インテルのエンジニアである私たちの意見として提供します。

SYCL*

SYCL* は、C++ にヘテロジニアス・プログラミングのサポートをもたらす Khronos の標準規格です。2020 年末に SYCL* 2020 仕様が確定し、それ以降、コンパイラーのサポートが拡大しています（実装については、Khronos のウェブサイト（英語）を参照してください）。

SYCL* については、「C++ のヘテロジニアスな将来の考察」や、sycl.tech（英語）に掲載されている多数のリソースなど、多くの場所で取り上げられています。簡単に言えば、SYCL* は、「C++ でヘテロジニアス・プログラミングを可能にし、ベンダーやアーキテクチャーを超えた移植性を実現するにはどうすればよいか？」という重要な課題に取り組んでいます。

コミュニティーからの強力なフィードバックにより、SYCL* 2020 は、さまざまなマルチベンダーとマルチアーキテクチャーに対応したエキサイティングな新機能を備えています。この記事では、これらの新機能とその目的について説明します。

5 つの優れた機能

SYCL* 2020 の重要な目標は、SYCL* を ISO C++ と整合させることであり、これには 2 つの利点があります。1 つは、C++ プログラマーにとって SYCL* が自然であることを保証します。もう 1 つは、SYCL* が、ほかの C++ ライブラリーや ISO C++ 自体にも影響を与える、ヘテロジニアス・プログラミングに対するマルチベンダー、マルチアーキテクチャー・ソリューションとして機能できるようになります。

SYCL* 2020 の構文変更の多くは、ベース言語を C++11 から C++17 へ更新したことに伴うもので、開発者はクラス・テンプレート引数推定（CTAD）（英語）や推定ガイドなどの機能を利用できるようになりました。しかし、多くの新機能も追加されています。この記事では、SYCL* 2020 の新機能の中から 5 つを取り上げ、それらが重要な理由を説明します。

1. **バックエンド**は、OpenCL* 以外の言語 / フレームワークで構築された SYCL* 実装への扉を開き、SYCL* がより多様なハードウェアをターゲットにすることを可能にします。
2. **統合共有メモリー（USM）**は、ポインターベースのアクセスモデルで、SYCL* 1.2.1 のバッファー / アクセサーモデルの代替となるものです。
3. **リダクション**は、一般的なプログラミング・パターンであり、SYCL* 2020 は「組込み」ライブラリーによってこれを高速化します。
4. **グループ・ライブラリー**は、協調的なワークアイテムの抽象化を提供し、（ベンダーに関係なく）ハードウェア機能と整合性を取ることで、アプリケーションのパフォーマンスとプログラマーの生産性を向上させます。
5. **アトミック参照**は C++20 の `std::atomic_ref` と整合性があり、C++ メモリーモデルをヘテロジニアス・デバイスに拡張します。

これらの機能は、オープンで、マルチベンダー、かつマルチアーキテクチャーの SYCL* エコシステムを確立し、C++ プログラマーが現在および将来のヘテロジニアス・コンピューティングの可能性を十分に引き出すのに役立ちます。

1. バックエンド

バックエンドの導入により、SYCL* 2020 は OpenCL* 以外の言語 / フレームワークで構築された実装への扉を開きます。その結果、名前空間は `cl::sycl::` から `sycl::` に短縮され、SYCL* ヘッダーファイルは `<CL/sycl.hpp>` から `<sycl/sycl.hpp>` になりました。

これは表面的な変更ではなく、SYCL* にとって重大な意味を持ちます。引き続き OpenCL* 上に実装できますが (そして多くの実装がそうしていますが)、汎用バックエンドのサポートにより、SYCL* はより多様なヘテロジニアス API とハードウェアをターゲットにできるプログラミング・モデルに変わりました。SYCL* は、C++ アプリケーションとベンダー固有のライブラリーとの間の「接着剤」として機能するようになり、開発者はより簡単に、しかもコードを変更することなく、さまざまなプラットフォームをターゲットにできます。

SYCL* 2020 は真にオープンで、クロスアーキテクチャー、かつクロスベンダーを実現

オープンソースの DPC++ コンパイラー・プロジェクト（英語）は、LLVM（clang）で SYCL* 2020 を実装し、この柔軟性を活かして NVIDIA*、AMD*、およびインテルの GPU をサポートします。SYCL* 2020 は真にオープンで、クロスアーキテクチャー、かつクロスベンダーを実現します（**図 1**）。

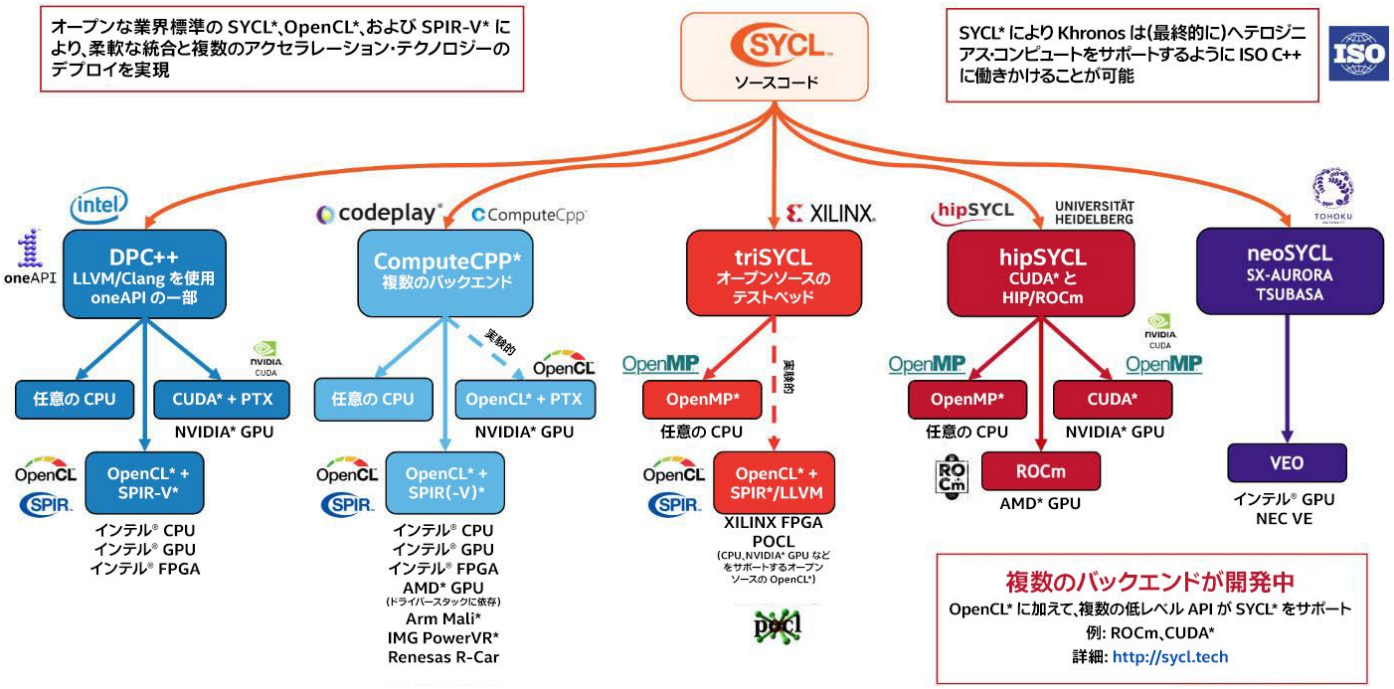

図 1. https://www.khronos.org/sycl/（英語）**の複数のバックエンドをターゲットとする SYCL* 実装**

2. 統合共有メモリー

デバイスによっては、ホスト（CPU）と統一されたメモリービューをサポートできます。SYCL* 2020 では、これを**統合共有メモリー（USM）**と呼び、SYCL* 1.2.1 のバッファー / アクセサーモデルに代わるポインターベースのアクセスモデルを可能にします。

USM を使ったプログラミングには、2 つの重要な利点があります。第一に、USM はホストとデバイスで単一の統合アドレス空間を提供します。USM 割り当てへのポインターはデバイス間で一貫しており、カーネルに直接引数として渡すことができます。これにより、既存のポインタベースの C++ および CUDA* コードの SYCL* への移行が大幅に

簡素化されます。第二に、USM はデバイス間で自動的に移行する共有割り当てを可能にし、プログラマーの生産性を向上させ、C++ コンテナー（`std::vector` など）や C++ アルゴリズム（**図 2** に示すように インテル® oneDPL（英語）を介して）との互換性を提供します。

```
sycl::usm_allocator<int, sycl::usm::alloc::shared> alloc(q.get_context(),
                                                         q.get_device());
std::vector<int, decltype(alloc)> vec(n, alloc);

auto policy = oneapi::dpl::execution::make_device_policy(q);
std::fill(policy, vec.begin(), vec.end(), 0);
```

図 2. 書籍で紹介している例**（英語）の C++ コンテナーとアルゴリズムでの USM の使用**

3 種類の USM 割り当ては、プログラマーに必要なデータ移動の制御を提供します。デバイス割り当てでは、プログラマーがアプリケーションのデータ移動を完全に制御できます。ホスト割り当ては、データの使用頻度が低く、移動のコストが見合わない場合、またはデータのサイズがデバイスのメモリーに収まらない場合に有効です。共有割り当ては、使用される場所に自動的に移行され、パフォーマンスと生産性の両方の利点をもたらします。

3. リダクション

SYCL* 2020 のリダクションに対するアプローチは、P0075（英語）の提案や、Kokkos（英語）および RAJA（英語）ライブラリーで実装されている機能など、ほかの C++ リダクションを参考にしています。

`reducer` クラスと `reduction` 関数を使用することで、SYCL* カーネルのリダクション・セマンティクスの変数表現が大幅に簡素化されます。また、コンパイル時にリダクション・アルゴリズムを指定するように実装できるため、多くのベンダーのさまざまなデバイスでハイパフォーマンスが得られます。

SYCL* 2020 のリダクションがもたらす改善の実例として、ブリストル大学で開発された人気の高い BabelStream（英語）ベンチマークを挙げることができます。BabelStream には、カーネル内のすべてのワークアイテムの浮動小数点総和を計算するシンプルなドット積カーネルが含まれています。SYCL* 1.2.1 バージョン（英語）は 43 行で、特定のアルゴリズム（ワークグループのローカルメモリーのツリー・リダクション）により記述されます。また、デバイスに最適な work-group サイズは、ユーザーが選択する必要があります（**図 3**）。SYCL* 2020 バージョン（英語）は、わずか 20 行と短いだけでなく、アルゴリズムや work-group サイズの選択を実装に委ねることで、パフォーマンスの移植性を高めています（**図 4**）。

```
template <class T>
T SYCLStream<T>::dot()
{
  queue->submit([&](handler &cgh)
  {
    auto ka    = d_a->template get_access<access::mode::read>(cgh);
    auto kb    = d_b->template get_access<access::mode::read>(cgh);
    auto ksum = d_sum->template get_access<access::mode::write>(cgh);

    auto wg_sum = accessor<T, 1, access::mode::read_write,
access::target::local>(range<1>(dot_wgsize), cgh);

    size_t N = array_size;
    cgh.parallel_for<dot_kernel>(nd_range<1>(dot_num_groups*dot_wgsize, dot_wgsize),
[=](nd_item<1> item)
    {
      size_t i = item.get_global_id(0);
      size_t li = item.get_local_id(0);
      size_t global_size = item.get_global_range()[0];

      wg_sum[li] = 0.0;
      for (; i < N; i += global_size)
        wg_sum[li] += ka[i] * kb[i];

      size_t local_size = item.get_local_range()[0];
      for (int offset = local_size / 2; offset > 0; offset /= 2)
      {
        item.barrier(cl::sycl::access::fence_space::local_space);
        if (li < offset)
          wg_sum[li] += wg_sum[li + offset];
      }

      if (li == 0)
        ksum[item.get_group(0)] = wg_sum[0];
    });
  });

  T sum = 0.0;
  auto h_sum = d_sum->template get_access<access::mode::read>();
  for (int i = 0; i < dot_num_groups; i++)
  {
    sum += h_sum[i];
  }

  return sum;
}
```

図 3. BabelStream のドット積カーネルの SYCL* 1.2.1 バージョン

```
template <class T>
T SYCLStream<T>::dot()
{
  queue->submit([&](sycl::handler &cgh)
  {
    sycl::accessor ka {d_a, cgh, sycl::read_only};
    sycl::accessor kb {d_b, cgh, sycl::read_only};

    cgh.parallel_for(sycl::range<1>{array_size},
      sycl::reduction(d_sum, cgh, std::plus<T>(), sycl::property::reduction::initiali
ze_to_identity{}),
      [=](sycl::id<1> idx, auto& sum)
      {
        sum += ka[idx] * kb[idx];
      });
  });

  sycl::host_accessor result {d_sum, sycl::read_only};
  return result[0];
}
```

図 4. BabelStream のドット積カーネルの SYCL* 2020 バージョン

4. グループ・ライブラリー

SYCL* 2020 は、SYCL* 1.2.1 の work-group 抽象化機能を拡張し、新しい sub-group 抽象化機能とグループベース・アルゴリズムのライブラリーを提供します。

`sub_group` クラスは、カーネル内で「一緒に」実行される協調的なワークアイテムのセットを表し、異なるベンダーのハードウェア機能に移植可能な抽象化を提供します。DPC++ コンパイラーでは、sub-group は常に、インテル® アーキテクチャーの SIMD ベクトル化、NVIDIA* アーキテクチャーの「warps」、AMD* アーキテクチャーの「wavefronts」といった重要なハードウェア概念にマッピングされ、SYCL* アプリケーションの低レベルのパフォーマンス・チューニングを可能にします。

ISO C++ との整合性のもう 1 つの例として、SYCL* 2020 では C++17 のグループベース・アルゴリズムの `all_of`、`any_of`、`none_of`、`reduce`、`exclusive_scan`、および `inclusive_scan` が追加されています。各アルゴリズムは異なるスコープでサポートされており、SYCL* 実装では、work-group や sub-group の並列処理を使用して、これらの関数の高度にチューニングされた協調的なバージョンを提供できます。

SYCL* 2020 のグループ・ライブラリーは、より多くのグループタイプと、より幅広いグループベース・アルゴリズムの基盤となります。

5. アトミック参照

C++20 では、アトミック参照（`std::atomic_ref`）で型をラップする機能が導入され、アトミック操作が大きく進化しました。SYCL* 2020 では、この設計を採用して拡張し（`sycl::atomic_ref` として）、アドレス空間とメモリースコープをサポートして、多様なヘテロジニアス・コンピューティングに完全に対応するアトミック参照の実装を実現しました。

SYCL* 2020 では、ISO C++ を拡張することについて慎重に検討し、パフォーマンスを犠牲にすることなく移植可能なプログラミングを可能にするには、メモリースコープの概念が必須であると考えました。ヘテロジニアス・システムには、複雑なメモリー階層があります（**図 5**）。

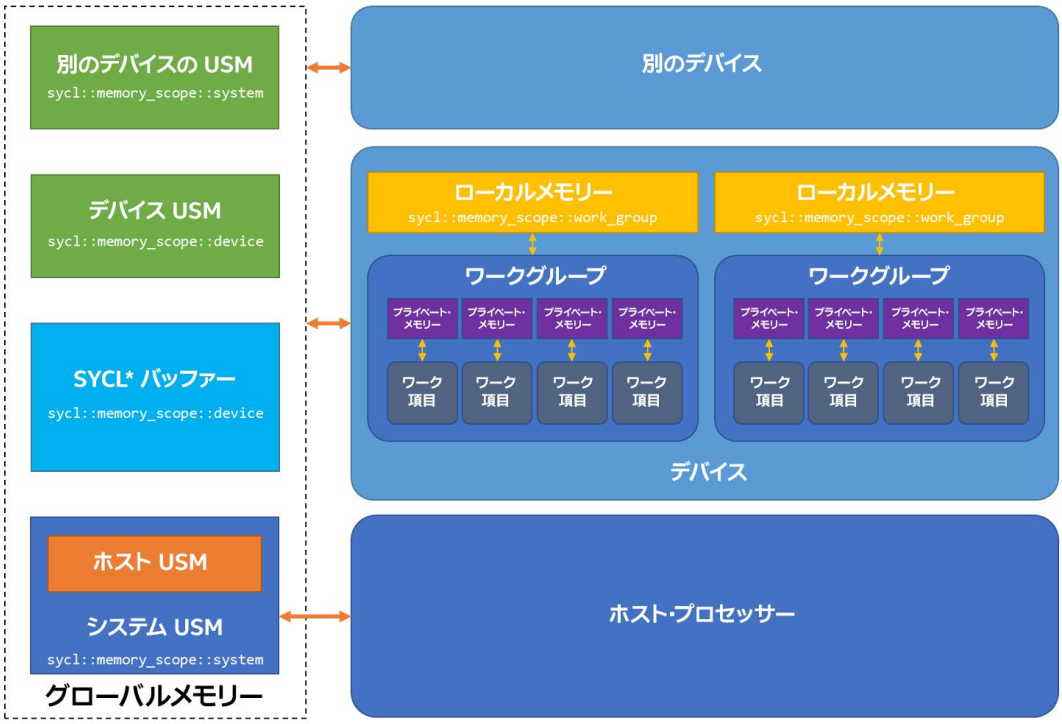

図 5. メモリースコープを使用することで、アトミック参照で一貫性が必要なメモリーを指定し、メモリーの更新を「見る」ことができるワークアイテムやデバイスをきめ細かく制御可能

メモリーモデルとアトミック操作は複雑なため、SYCL* ではできるだけ多くのデバイスをサポートできるように、すべてのデバイスで C++ メモリーモデルを完全にサポートすることを要求していません。むしろ、SYCL* はさまざまなデバイス機能を提供しています。これは、あらゆるベンダーのデバイスにオープンであることのもう 1 つの素晴らしい例です。

SYCL* 2020 の先 : ベンダー拡張

SYCL* 2020 でより多くのバックエンドとハードウェアがサポートされたことで、ベンダーによる拡張機能の開発が促進されています。これらの拡張機能は、それを必要とするデバイスに実用的なソリューションを提供し、将来の SYCL* 標準の方向性を示すイノベーションを可能にします。標準化プロセスにおいて拡張は重要な役割を果たします。この記事で取り上げたいくつかの機能は、DPC++ コンパイラー・プロジェクトで検討された拡張機能に由来するものです。

ここでは、SYCL* 2020 のベンダー拡張機能として DPC++ コンパイラー・プロジェクトでサポートされている 2 つの新機能を簡単に説明します。

カーネルスコープでのグループローカル・メモリー

SYCL* 1.2.1 は、ローカルアクセサーを介してグループローカル・メモリーをサポートしています。グループローカル・メモリーは、カーネルの外側で宣言され、カーネルの引数として取得される必要があります。OpenCL* や CUDA* などの言語のプログラマーにとって、これは不自然に感じられるかもしれません。そこで、グループローカル・メモリーをカーネル関数内で宣言できるようにする拡張機能（英語）が設計されました。この変更により、カーネルはより自己完結的になり、（コンパイル時にローカルメモリーのサイズが分かっている場合）コンパイラーの最適化にも影響を与えることができます。

FPGA 固有の拡張機能

DPC++ コンパイラー・プロジェクトでインテル® FPGA を使用できるようになりました。この拡張機能またはそれに近いものは、すべてのベンダーの FPGA に移植可能であると考えています。FPGA は重要なアクセラレーターであり、この先駆的な取り組みが、ほかのベンダーの拡張プロジェクトとともに、将来の SYCL* 標準に反映されることを期待しています。

FPGA セレクターの追加により、FPGA ハードウェアまたは FPGA エミュレーション・デバイスを簡単に利用できるようになりました。これは、FPGA をターゲットとするソフトウェア開発者にとって重要な考慮事項である、迅速なプロトタイプ作成を可能にします。FPGA の LSU 制御は、FPGA のロード / ストア操作をチューニングし、特定の方法でグローバル・メモリー・アクセスの実装を構成するように明示的に要求できます。FPGA メモリーチャネル経由で FPGA 設計をチューニングできるように、外部メモリーバンク（DDR チャネルなど）を持つデータの配置制御が追加されています。FPGA レジスターにより、ハイパフォーマンスな FPGA パイプラインを実現する主要なチューニング制御が有効になります。

まとめ

ヘテロジニアスが普及し、ハードウェアの選択肢はますます多様化し、より優れたパフォーマンスとワット当たりのパフォーマンスを追求するものが多くなっています。この傾向は、SYCL* のようなオープンで、マルチベンダー、かつマルチアーキテクチャーのプログラミング・モデルの必要性をさらに高めます。

この記事では、SYCL* 2020 の移植性とパフォーマンスの移植性を可能にする 5 つの新機能を紹介しました。SYCL* 2020 により、C++ プログラマーはヘテロジニアス・コンピューティングの可能性を十分に活用できるようになります。

詳細は、sycl.tech（英語）を参照してください。オンライン・チュートリアル、SYCL* 書籍（無料の PDF（英語））へのリンク、現在の SYCL* 標準仕様（英語）へのリンクがあります。

[訳者注 : SYCL* 2020 リビジョン 4 の日本語参考訳が iSUS で公開されています。]