創刊 10 周年記念号

oneAPI アプリケーションのパフォーマンス最適化

インテル® oneMKL を使用したモンテカルロ・
シミュレーションの高速化

Issue
40
2020

目次

編集者からのメッセージ

Henry A. Gabb インテル コーポレーション シニア主席エンジニア
HPC と並列コンピューティング分野で長年の経験があり、並列プログラミングに関する記事を多数出版しています。『Developing Multithreaded Applications: A Platform Consistent Approach』の編集者 / 共著者で、インテルと Microsoft* による Universal Parallel Computing Research Centers のプログラム・マネージャーを務めました。

The Parallel Universe 創刊 10 周年

我々はどのような道をたどり、どこへ向かっているのか

本号で The Parallel Universe は創刊 10 周年を迎え、私が編集長に就任してから 3 年が経ちました。名前のとおり、本誌は元々、並列コンピューティングに関連する記事の掲載を目的としていました。並列処理は 10 年前よりもはるかに広く普及していますが、我々のテーマは今でも変わっていません。しかし、時代とともに取り上げるトピックは変わりました。新しいベクトル命令セットは、効率良い命令レベルの並列処理を達成します。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) により、インテルの CPU は実質的にアクセラレーターとして機能します。また、データ・アナリティクスが登場したことで、編集方針を拡張して、並列フレームワーク (Apache Spark* など) や、生産性言語 (Python* や Julia など) によるハイパフォーマンスの実現に関するトピックも取り上げるようになりました。そして、並列コンピューティングの世界もヘテロジニアス並列処理へと拡大しました。

前号で、oneAPI はヘテロジニアス並列コンピューティングにおいて今後重要になると述べました。そのため、本号の最初の 3 つの記事では oneAPI に関連するトピックを取り上げています。注目記事「**GPU-Quicksort**」では、OpenCL* からデータ並列 C++ へ移行する方法をステップごとに詳しく説明します。続く「**oneAPI アプリケーションのパフォーマンス最適化**」と「**ベータ版インテル® oneAPI マス・カーネル・ライブラリーのデータ並列 C++ 使用モデル**」では、インテルのプログラミング・ツールと oneAPI アプリケーションの最適化に関するケーススタディーを紹介し、インテル® oneMKL を利用してインテル® マス・カーネル・ライブラリー (インテル® MKL) 関数をアクセラレーターにオフロードする方法を示します。

OmniSci 社の Venkat Krishnamurthy 氏と Kathryn Vandiver 氏による「**大規模で高速なアナリティクスをインテル® アーキテクチャーで行う**」では、データサイエンスと従来のアナリティクスの統一について述べます。続く「**自動微分を使用した並列計算の新しいアプローチ**」では、Dmitri Goloubentsev 氏 (Matlogica 社) と Evgeny Lakshtanov 氏 (アヴェイロ大学) が、オブジェクト指向のシングルスレッドのスカラーコードを、ベクトル化されたマルチスレッドのラムダ関数に変換するツールについて説明します。

The Parallel Universe 創刊号において、創刊編集長の James Reinders が執筆した「**マルチコア向け並列プログラミングの 8 つのルール**」は、今日でも重要かつ的確な内容であるため、10 周年を記念して再掲載しています。

最後に、これまでにない取り組みですが、書評で本号を締めくくりたいと思います。Oracle Corporation の Ruud van der Pas 氏が、Timothy G. Mattson、Yun (Helen) He、Alice E. Koniges の 3 名により執筆された『**The OpenMP Common Core: Making OpenMP Simple Again**』の書評を寄稿してくれました。最初の OpenMP* 仕様 (1997 年公開) は、驚くべき技術文書でした。簡潔で (わずか 63 ページ)、多数のコード例が掲載されており、OpenMP* を始めるのに役立ちました。その後、OpenMP* 仕様にはベクトル化、スレッドの配置、アクセラレーターへのオフロードをプログラマーが細かく制御できるようにする重要な新機能が追加され、現在では 666 ページにまで増えました。しかし、本のタイトルにある OpenMP* の「Common Core (必修科目)」は、23 年前とほとんど同じです。

コードの現代化、ビジュアル・コンピューティング、データセンターとクラウド・コンピューティング、データサイエンス、システムと IoT 開発、oneAPI を利用したヘテロジニアス並列コンピューティング向けのインテルのソリューションの詳細は、**Tech.Decoded** (英語) を参照してください。

Henry A. Gabb
2020 年 4 月

GPU-Quicksort

OpenCL* からデータ並列 C++ への移行

Robert Ioffe インテル コーポレーション シニア・エクサスケール・パフォーマンス・ソフトウェア・エンジニア

データ並列 C++ (DPC++) は、Khronos SYCL* 標準ベースのヘテロジニアスで移植性の高いプログラミング言語です。このシングルソース・プログラミング言語は、CPU、統合 / ディスクリート GPU、FPGA、その他のアクセラレーターなど、さまざまなプラットフォームをターゲットにすることができます。ここでは、DPC++ で何ができるのか理解するため、重要な OpenCL* アプリケーションである GPU-Quicksort を DPC++ に移行します。目標は、OpenCL* アプリケーションの性能を超えることです。OpenCL* C で汎用アルゴリズムを記述することは非常に困難であり、異なるデータ型を扱うソートのようなアルゴリズムの実装では、これが深刻な問題となります。OpenCL* で記述されたオリジナルの GPU-Quicksort は、符号なし整数をソートします。

ここでは、DPC++ でテンプレートを使用して、複数のデータ型に対応した GPU-Quicksort を実装する方法を示します。そして、DPC++ が移植性に優れていることを示すため、GPU-Quicksort を Windows* と RHEL に移植します。

GPU-Quicksort とは

GPU-Quicksort は、高度に並列化されたマルチコア・グラフィックス・プロセッサー向けに設計された、ハイパフォーマンスなソート・アルゴリズムです。2009 年に、当時スウェーデンのチャルマース工科大学の学生であった Daniel Cederman 氏と Phillippas Tsigas 教授によって開発されました。元々 CUDA* で実装されていましたが、2014 年に筆者がインテル® インテグレーテッド・プロセッサー・グラフィックス上でハイパフォーマンスを実証し、入れ子の並列処理とワークグループ・スキャン関数を使用するため OpenCL* 1.2 と OpenCL* 2.0 で実装し直し、インテルの OpenCL* ドライバーで完全に実装されました。ここでは、GPU-Quicksort の OpenCL* 1.2 実装を DPC++ に移行して、符号なし整数だけでなく、単精度や倍精度の浮動小数点もソートできるように実装を汎用化します。

OpenCL* とは

OpenCL* 1.2 実装をベースに作業を開始します。インテルは、ヘテロジニアス並列システムをプログラミングするための Khronos 標準規格である OpenCL* を、さまざまなオペレーティング・システムとプラットフォームで完全にサポートしています。OpenCL* は以下で構成されます。

- ランタイム
- ホスト API
- デバイス C ベースのプログラミング言語である OpenCL* C

これは OpenCL* の利点でもあり、制限でもあります。利点は、ハイパフォーマンスで移植性の高いヘテロジニアス並列アプリケーションを記述できることです。主な制限は、ホスト側とデバイス側で別々のコードを記述してデバッグする必要があること、およびプログラマーに馴染みのあるテンプレートやその他の C++ 機能を利用できないため、汎用ライブラリーの記述が困難なことです。

データ並列 C++ とは

データ並列 C++ (DPC++) は、Khronos SYCL* を拡張したインテルの実装です。SYCL* 標準は、上記の OpenCL* の制限に対応するように設計されています。DPC++ は次の機能を提供します。

- **シングルソースのプログラミング・モデル。**ホストとデバイスを同じコードベースでプログラミングできます。
- **C++ テンプレートとテンプレート・メタプログラミング。**移植性を損なうことなく、パフォーマンスへの影響を最小限に抑えて、デバイス上でこれらを活用できます。

DPC++ では、プログラマーは CPU、GPU、および FPGA をターゲットにして、アクセラレーター固有のチューニングを行うことができ、OpenCL* よりも確実に改善されています。また、**インテル® VTune™ プロファイラー**や**インテル® Advisor** などのインテル® ソフトウェア・ツールと、GDB でサポートされています。ここでは、DPC++ の特にテンプレート機能を活用します。

開始点 : 2014 年の Windows* アプリケーション

「**OpenCL* 2.0 の GPU-Quicksort: 入れ子の並列処理とワークグループ・スキャン関数**」(英語) にある、OpenCL* 1.2 で実装された GPU-Quicksort を開始点として使用します。このアプリケーションは、Windows* 向けに記述されているため、作業を始める前に、時間を測定するクロスプラットフォーム・コードを追加し、アライメントされたメモリーの割り当て / 解放を Windows* の **`_alligned_malloc/_aligned_free`** から **`aligned_alloc/free`** に変更して、Ubuntu 18.04 に移行しました。

GPU-Quicksort アーキテクチャーについて簡単に見てみましょう。2 つのカーネルで構成されます。

1. **`gqsort_kernel`**
2. **`lqsort_kernel`**

OpenCL* 1.2 で記述されたこれらのカーネルは、ディスパッチャー・コードによって結合され、入力が **`lqsort_kernel`** でソートできる小さなチャンクに分割されるまで繰り返し **`gqsort_kernel`** を呼び出します。ユーザーはアプリケーションで以下を指定できます。

- 測定のためソートを実行する**回数**。
- カーネルを実行する**ベンダーとデバイス**。
- 入力の**サイズ**。
- デバイスの詳細を**表示するかどうか**。

このアプリケーションは、典型的な OpenCL* アーキテクチャーに従って、OpenCL* プラットフォームとデバイスを初期化し、コードをビルドするユーティリティーをサポートします。OpenCL* カーネルとサポート関数は、ユーザー引数を受け取るメイン・アプリケーションとは別のファイルにあり、プラットフォームとデバイスを初期化し、カーネルをビルドし、メモリーを適切に割り当て、バッファーを作成してカーネル引数にバインドし、ディスパッチャー関数を起動します。

データ並列 C++/OpenCL* の相互運用性 : プラットフォームの初期化

最初に、**インテル® oneAPI ベース・ツールキット** (英語) をインストールします。これには、**インテル® oneAPI DPC++ コンパイラー** (英語) が含まれます。**`CL/sycl.hpp`** ヘッダーをインクルードして、DPC++ の冗長性を避けるため名前空間 **`cl::sycl`** を使用して DPC++ への移行を開始します。

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

プラットフォーム、デバイス、コンテキスト、およびキューは、OpenCL* ではなく、簡潔な DPC++ で初期化します。

```
device d(default_selector);
  if (d.is_host()) {
    // This platform has no OpenCL devices
    return;
  }

  queue queue(default_selector, [](cl::sycl::exception_list l) {
    for (auto ep : l) {
      try {
        std::rethrow_exception(ep);
      } catch (cl::sycl::exception e) {
        std::cout << e.what() << std::endl;
      }
    }
  });
```

アプリケーションの残りの部分は OpenCL* ベースであるため、OpenCL* コンテキスト、デバイス、およびキューを取得する必要があります。

```
pOCL->contextHdl = queue.get_context().get();
pOCL->deviceID = queue.get_device().get();
pOCL->cmdQHdl = queue.get();
```

これが最初の反復です。インテル® DPC++ コンパイラーで設定してコンパイルし、実行します。

データ並列 C++: インテルの GPU の選択

最初の反復の欠点は、常にデフォルトのデバイスを選択することです。選択されるデバイスは、インテルの GPU であるとは限りません。インテルの GPU を指定するには、カスタム・デバイス・セレクターを記述する必要があります。

```
class intel_gpu_selector : public device_selector {
 public:
 intel_gpu_selector() : device_selector() {}

 int operator()(const device& device) const override {
   if (device.get_info<info::device::name>().find("Intel") != std::string::npos) {
     if (device.get_info<info::device::device_type>() ==
         info::device_type::gpu) {
       return 50;
     }
   }
   /* Never choose device with a negative score */
   return -1;
 }
};
```

ユーザーが要求した場合、**intel_gpu_selector** を使用してインテルの GPU を選択します。

```
auto get_queue = [&pDeviceStr, &pVendorStr]() {
   device_selector* pds;
   if (pVendorStr == std::string("intel")) {
     if (pDeviceStr == std::string("gpu")) {
         static intel_gpu_selector selector;
     pds = &selector;
   } else if (pDeviceStr == std::string("cpu")) {
     static cpu_selector selector;
     pds = &selector;
   }
 } else {
   static default_selector selector;
   pds = &selector;
 }
   device d(*pds);

   queue queue(*pds, [](cl::sycl::exception_list l) {
     for (auto ep : l) {
       try {
         std::rethrow_exception(ep);
       } catch (cl::sycl::exception e) {
         std::cout << e.what() << std::endl;
       }
     }
   });
   return queue;
 };

 auto queue = get_queue();
```

データ並列 C++: カーネル引数の設定とカーネルの起動

第 3 反復では、DPC++ を使用してカーネル引数を設定し、カーネルを起動します。プログラムのビルドとカーネルの取得は、OpenCL* で行います。**cl::sycl::kernel** オブジェクトを使用してオリジナルの OpenCL* カーネルをラップします。次に例を示します。

```
cl::sycl::kernel sycl_gqsort_kernel(gqsort_kernel, pOCL->queue.get_context());
```

clSetKernelArg メソッドを DPC++ の **set_arg** メソッドに、**clEnqueueNDRange** 呼び出しを **parallel_for** 呼び出しにそれぞれ置き換えます。以下の例は、**gqsort_kernel** ですが、**lqsort_kernel** の変更も非常に似ています。

```
 pOCL->queue.submit([&](handler& cgh) {
      cgh.set_arg(0, db);
      cgh.set_arg(1, dnb);
      cgh.set_arg(2, blocksb);
      cgh.set_arg(3, parentsb);
      cgh.set_arg(4, newsb);

      cgh.parallel_for(nd_range<1>(range<1>(GQSORT_LOCAL_WORKGROUP_SIZE *
(blocks.size())),
                                   range<1>(GQSORT_LOCAL_WORKGROUP_SIZE)),
      sycl_gqsort_kernel);
    });
    pOCL->queue.wait_and_throw();
```

以下のように、1 回の **set_args** 呼び出しですべてのカーネル引数を設定することもできます。

```
cgh.set_args(db, dnb, blocksb, parentsb, newsb);
```

parallel_for も次のように簡潔に指定することができます。

```
cgh.parallel_for(nd_range<>(GQSORT_LOCAL_WORKGROUP_SIZE * blocks.size(),
                            GQSORT_LOCAL_WORKGROUP_SIZE),
```

データ並列 C++: バッファーの作成とアクセスモードの設定

OpenCL* バッファーを DPC++ バッファーに変換します (最初の 2 つは、アライメントされた割り当てで関数に参照渡しされるメモリーをラップしており、残りの 3 つは STL ベクトルから作成されます)。参照渡しのバッファーは、**get_access** メンバー関数の前に **template** キーワードを使用します。必要なアクセス (読み取り、書き込み、あるいは両方) に応じて、バッファーのアクセスモードは異なります。カーネル引数としてバッファーを直接渡すのではなく、バッファーへのアクセサーを渡します。

```
    buffer<T>  d_buffer(d, range<>(size), {property::buffer::use_host_ptr()});
    buffer<T>  dn_buffer(dn, range<>(size), {property::buffer::use_host_ptr()});
…
    buffer<block_record>  blocks_buffer(blocks.data(), blocks.size(),
{property::buffer::use_host_ptr()});
    buffer<parent_record>  parents_buffer(parents.data(), parents.size(),
{property::buffer::use_host_ptr()});
    buffer<work_record>  news_buffer(news.data(), news.size(),
{property::buffer::use_host_ptr()});

    pOCL->queue.submit([&](handler& cgh) {
      auto db = d_buffer.template get_access<access::mode::discard_read_write>(cgh);
      auto dnb = dn_buffer.template get_access<access::mode::discard_read_write>(cgh);
      auto blocksb = blocks_buffer.get_access<access::mode::read>(cgh);
      auto parentsb = parents_buffer.get_access<access::mode::read>(cgh);
      auto newsb = news_buffer.get_access<access::mode::write>(cgh);

      cgh.set_args(db, dnb, blocksb, parentsb, newsb);

      cgh.parallel_for(nd_range<>(GQSORT_LOCAL_WORKGROUP_SIZE * blocks.size(),
                                  GQSORT_LOCAL_WORKGROUP_SIZE),
        sycl_gqsort_kernel);
    });
    pOCL->queue.wait_and_throw();
```

データ並列 C++: プラットフォームとデバイスのプロパティーの照会

OpenCL* では、**clGetPlatformInfo** メソッドと **clGetDeviceInfo** メソッドを使用してプラットフォームとデバイスのプロパティーを照会します。これらの情報の照会には、**get_info<>** メソッドを使用します。次に例を示します。

```
auto max_work_item_dimensions = q.get_device().get_info<info::device::max_work_item_dimensions>();
std::cout << "CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: " << max_work_item_dimensions << std::endl;
```

または、次のようなより複雑な構造のプロパティーを照会することもできます。

```
auto max_work_item_sizes = q.get_device().get_info<info::device::max_work_item_sizes>();
printf ("CL_DEVICE_MAX_WORK_ITEM_SIZES    :     (%5zu, %5zu, %5zu)\n",
                    max_work_item_sizes[0],max_work_item_sizes[1], max_work_item_sizes[2]);
```

OpenCL* カーネルからデータ並列 C++ への移行 - パート 1: gqsort_kernel

ここまでは、DPC++ でプラットフォームとデバイスを初期化し、バッファーとアクセサーを作成してカーネルにバインドし、デバイスでカーネルを起動しました。しかし、カーネルの作成は OpenCL* で行う必要があります。OpenCL* C と **clBuildProgram/clCreateKernel** API を使用して、プログラムをビルドしカーネルを作成します。OpenCL* C カーネルは、実行時にビルドの前にプログラムにロードされる別のファイルに格納されます。これを変更します。2 つのカーネルのうち、簡単な **gqsort_kernel** から作業します。

DPC++ では、ラムダまたはファンクターを使用してカーネルを作成します。通常、小さなカーネルはラムダを使用して作成します。サポート関数を使用する複雑なカーネルでは、ファンクター・クラスを作成したほうが良いでしょう。**gqsort_kernel_class** ファンクターを作成して、後で複数のデータ型をソートできるようにテンプレート化します。

```
template <class T>
class gqsort_kernel_class {
…
};
```

典型的なファンクター・クラスには、パラメーターとして反復 ID (この例では **nd_item<1> id**) を受け取る **void operator()** があります。カーネル本体は **void operator()** にあります。ファンクターには、OpenCL* カーネルのグローバルおよびローカル・メモリー・ポインターと同等の、グローバルおよびローカルアクセサーを受け取るコンストラクターがあります。典型的な DPC++ ファンクターには、グローバルおよびローカルアクセサーの型を定義する **using** 句を持つプリアンブルがあります。**gqsort_kernel** の例では、次のようになります。

```
using blocks_read_accessor = accessor<block_record<T>, 1, access::mode::read,
access::target::global_buffer>;
using parents_read_write_accessor = accessor<parent_record, 1, access::mode::read_write,
access::target::global_buffer>;
using news_write_accessor = accessor<work_record<T>, 1, access::mode::write,
access::target::global_buffer>;
using discard_read_write_accessor =
      accessor<T, 1, access::mode::discard_read_write, access::target::global_buffer>;
using local_read_write_accessor = accessor<uint, 1, access::mode::read_write,
access::target::local>;
```

ファンクターの **private** セクションには、**void operator()** の本体で使用されるすべてのグローバルおよびローカルアクセサーが含まれます。この例では、次のようになります。最初の 5 つは、グローバルバッファーへのアクセサーで、残りはローカルバッファーへのアクセサーです。

```
private:
     discard_read_write_accessor d, dn;
     blocks_read_accessor blocks;
     parents_read_write_accessor parents;
     news_write_accessor news;
     local_read_write_accessor lt, gt, ltsum, gtsum, lbeg, gbeg;
```

gqsort_kernel は、サポート構造体と 2 つのサポート関数 **plus_prescan** と **median** を使用する複雑なカーネルです。これらのサポート関数は、特殊な OpenCL* 関数を使用し、ローカルメモリー配列および変数、ローカルおよびグローバルバリア、アトミック操作を幅広く使用します。これらの要素をすべて DPC++ に変換する必要があります。

関数から開始しましょう。構造体はテンプレート化されているため省略します。スキャン合計を計算する **plus_prescan** 関数は比較的単純なため、ソートを汎用的にする準備としてテンプレート関数にするだけで DPC++ に移行できます。

```
    template <class T>
void plus_prescan(T *a, T *b) {
    T av = *a;
    T bv = *b;
    *a = bv;
    *b = bv + av;
}
```

median 関数は、テンプレート関数にするだけでなく、OpenCL* C の **select** 関数を DPC++ の **cl::sycl::select** 関数に置き換えて、同様のホスト関数と区別するため名前を **median_select** に変更します。

```
template <class T>
T median_select(T x1, T x2, T x3) {
    if (x1 < x2) {
        if (x2 < x3) {
            return x2;
        } else {
            return cl::sycl::select(x1, x3, (uint)(x1 < x3));
        }
    } else {
        if (x1 < x3) {
            return x1;
        } else {
            return cl::sycl::select(x2, x3,(uint)(x2 < x3));
        }
    }
}
```

OpenCL* C では、ローカルメモリー変数と配列をカーネルの本体内で作成して、カーネル引数として渡すことが可能です。しかし、DPC++ では、ファンクターを使用する場合、ファンクターを構築する際にローカル・バッファー・アクセサーを渡します。この例では、すべてのローカルメモリー変数と配列は符号なし整数を格納するため、特殊な **local_read_write_accessor** 型を作成します。

```
using local_read_write_accessor = accessor<uint, 1,
access::mode::read_write, access::target::local>;
```

すべてのローカルメモリー変数を宣言します。

```
local_read_write_accessor
    lt(range<>(GQSORT_LOCAL_WORKGROUP_SIZE+1), cgh),
gt(range<>(GQSORT_LOCAL_WORKGROUP_SIZE+1), cgh),
    ltsum(range<>(1), cgh), gtsum(range<>(1), cgh), lbeg(range<>(1), cgh), gbeg(range<>(1),
cgh);
```

そして、それらをパラメーターとして、グローバル・バッファー・アクセサーとともにファンクター・コンストラクターに渡します。次に、生成されたオブジェクトを **parallel_for** に渡します。

```
auto gqsort = gqsort_kernel_class<T>(db, dnb, blocksb, parentsb, newsb, lt, gt, ltsum,
gtsum, lbeg, gbeg);

      cgh.parallel_for(
        nd_range<>(GQSORT_LOCAL_WORKGROUP_SIZE * blocks.size(),
                   GQSORT_LOCAL_WORKGROUP_SIZE),
        gqsort);
```

この点において、DPC++ は OpenCL* C よりも複雑です。**get_group_id** 関数と **get_local_id** 関数は次のようになります。

```
const size_t blockid = id.get_group(0);
const size_t localid = id.get_local_id(0);
```

ローカルバリアは、

```
barrier(CLK_LOCAL_MEM_FENCE);
```

から以下に変更されます。

```
id.barrier(access::fence_space::local_space);
```

グローバルおよびローカルバリアは、

```
barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
```

から以下に変更されます。

```
id.barrier(access::fence_space::global_and_local);
```

DPC++ のアトミック操作は、OpenCL* C のように洗練されていません。
OpenCL* C では簡潔な以下のコードが

```
psstart = &pparent->sstart;
psend = &pparent->send;
lbeg = atomic_add(psstart, ltsum);
gbeg = atomic_sub(psend, gtsum) - gtsum;
```

DPC++ では次のようになります。

```
          cl::sycl::atomic<uint> psstart_a(multi_ptr<uint,
access::address_space::global_space>(&pparent.sstart));
          cl::sycl::atomic<uint> psend_a(multi_ptr<uint,
access::address_space::global_space>(&pparent.send));
          lbeg[0] = cl::sycl::atomic_fetch_add(psstart_a, ltsum[0]);
          gbeg[0] = cl::sycl::atomic_fetch_sub(psend_a, gtsum[0]) - gtsum[0];
```

DPC++ のアトミック操作は、グローバルまたはローカル・メモリー・ポインターを直接操作できないため、**`cl::sycl::atomic<>`** 変数をアトミック操作を行うために作成しています。

ここまでで、サポート構造体とサポート関数を変換してテンプレート化し、特殊な OpenCL* C 関数を DPC++ に変換しました。また、ローカルアクセサーを持つテンプレート関数を作成し、バリアとアトミック操作を変換しました。

OpenCL* カーネルからデータ並列 C++ への移行 - パート 2: lqsort_kernel

lqsort_kernel の変換も **gqsort_kernel** の変換と似ています。**lqsort_kernel_class** ファンクターを作成して、ローカルメモリー配列と変数、およびバリアを変換します (アトミック操作はありません)。**lqsort_kernel** もサポート関数とサポート構造体を使用します。**gqsort_kernel** で使用される **plus_prescan** と **median_select** に加えて、**lqsort_kernel** にはより複雑な **bitonic_sort** と **sort_threshold** があります。変換後、これらの関数は **lqsort_kernel_class** のメンバー関数になります。DPC++ では反復オブジェクトが必要なバリアの使用により、これらの関数のシグネチャーは変わります。これらの関数は、ローカルおよびグローバル・メモリー・ポインターを使用するため、特別な処理が必要です。そのため、OpenCL* C シグネチャーは、

```
void bitonic_sort(local uint* sh_data, const uint localid)
```

から以下に変わります。

```
void bitonic_sort(local_ptr<T> sh_data, const uint localid, nd_item<1> id)
```

同様に、

```
void sort_threshold(local uint* data_in, global uint* data_out,
                    uint start,  uint end,
                   local uint* temp, uint localid)
```

も以下に変わります。

```
void sort_threshold(local_ptr<T> data_in, global_ptr<T> data_out,
              uint start, uint end,
              local_ptr<T> temp_, uint localid,
              nd_item<1> id)
```

gqsort_kernel と同様に、後で複数のデータ型を扱えるように、**UINT_MAX** マクロを **std::numeric_limits<T>::max()** に置き換えてこれらの関数を変換します。

lqsort_kernel を変換する際に、ローカルメモリーへのポインター (**local uint* sn;** など) は **local_ptr<>** オブジェクト (**local_ptr<T> sn;** など) に置き換えられます。ローカルアクセサーからローカルポインターを取得するため、アクセサーの **get_pointer** メンバー関数を呼び出します。

```
sn = mys.get_pointer();
```

local_ptr<> オブジェクトと **global_ptr<>** オブジェクトはポインター演算を使用するため、**d + d_offset** (**d** はグローバルポインター) は次のようになります。

```
d.get_pointer() + d_offset
```

ローカルメモリー変数は、サイズ 1 のアクセサー (つまり、**gtsum[0]** のようなインデックス 0 の配列アクセス) として変換します。**lqsort_kernel** の変換が完了したら、DPC++ へ完全に移行できますが、この時点では符号なし整数しかソートできません。しかし、サポート構造体とサポート関数、および 2 つのメインカーネルのファンクター・クラスはすでにテンプレート化されているため、複数のデータ型に対応するのは容易です。

データ並列 C++ の利点 : テンプレートと注意事項

DPC++ の真の力は、C++ テンプレートを使用して汎用コードを記述できることです。このセクションでは、GPU-Quicksort を汎用化して、符号なし整数だけでなく、単精度や倍精度の浮動小数点など、ほかの基本データ型もソートできるようにします。前述の **UINT_MAX** から **std::numeric_limits<T>::max()** への変更に加えて、**median_select** 関数を変更する必要があります。**cl::sycl::select** は、第 1 引数と第 2 引数の型のサイズに応じて、異なる第 3 引数を受け取るため、**select_type_selector** 型の特徴クラスを追加します。

```
template <class T> struct select_type_selector;

template <> struct select_type_selector<uint>
{
  typedef uint data_t;
};

template <> struct select_type_selector<float>
{
  typedef uint data_t;
};

template <> struct select_type_selector<double>
{
  typedef ulong data_t;
};
```

これにより、ブール値の比較を **cl::sycl::select** で必要な適切な型に変換することができます。**median_select** は次のようになります。

```
template <class T>
T median_select(T x1, T x2, T x3) {
  if (x1 < x2) {
    if (x2 < x3) {
      return x2;
    } else {
      return cl::sycl::select(x1, x3, typename select_type_selector<T>::data_t(x1 < x3));
    }
  } else {
    if (x1 < x3) {
      return x1;
    } else {
      return cl::sycl::select(x2, x3, typename select_type_selector<T>::data_t(x2 < x3));
    }
  }
}
```

追加の型に対応するには、**select_type_selector** を編集します。これで、**GPUQSort** は GPU で単精度と倍精度の浮動小数点もソートできるようになりました。

Windows* と RHEL への移植

DPC++ の移植性を実証するため、Windows* と RHEL へコードを移植します。RHEL への移植は非常に簡単です。リンク時にインテルの imf 数学ライブラリーを追加するだけです。Windows* への移植には、もう少し手間がかかります。コンパイル時に次の定義を追加します。

```
-D_CRT_SECURE_NO_WARNINGS -D_MT=1
```

倍精度の **cl::sycl::select** は第 3 引数に **unsigned long long** 型 (Linux* では **unsigned long** 型) を必要とすることから、倍精度の **select_type_selector** を次のように変更します。

```
template <> struct select_type_selector<double>
{
#ifdef _MSC_VER
  typedef ulonglong data_t;
#else
  typedef ulong data_t;
#endif
};
```

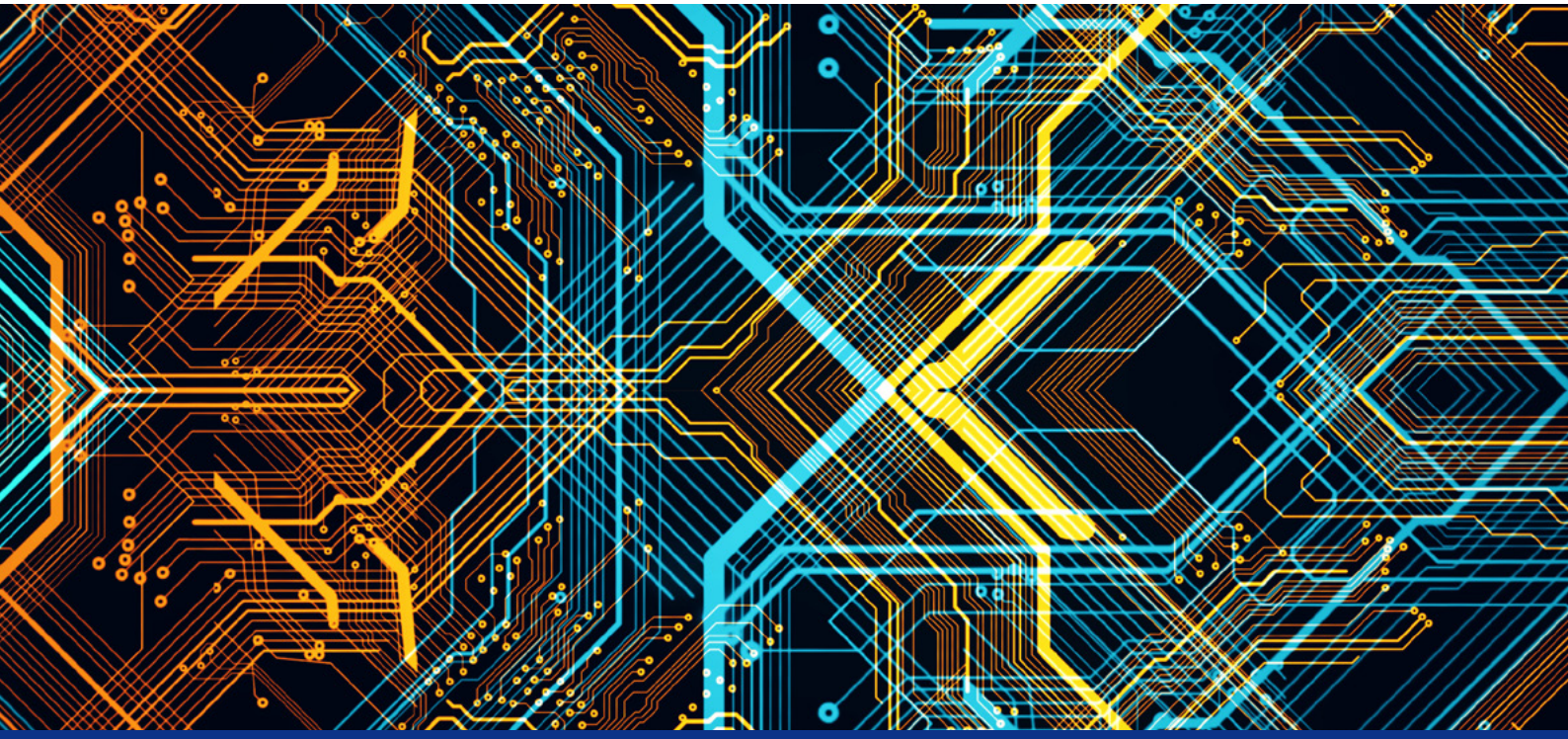
Windows* では、マクロ定義が **std::max** および **std::min** と競合しないように、**max** および **min** を未定義にします。これで、Windows* および RHEL でインテルの GPU を使用して、符号なし整数、単精度浮動小数点、倍精度浮動小数点をソートできます。

今すぐ実践してみましょう

この記事では、GPU-Quicksort をオリジナルの OpenCL* 1.2 から DPC++ へ移行する方法をステップごとに説明しました。ステップごとに、アプリケーションの動作を確認できることが重要です。DPC++ をワークフローに導入することを検討している場合は、小規模から初めて、徐々に追加していくか、時間をかけて完全に移行してください。容易に OpenCL* と DPC++ をコードベースに混在させ、両方の利点を得られます。従来の OpenCL* カーネルをそのまま使用しつつ、DPC++ で開発する新しいコードでは C++ テンプレート、クラス、ラムダを活用できます。Windows* や各種 Linux* ディストリビューションへのコードの移植も容易で、開発プラットフォームを選択できます。さらに、強力なインテルのツールが DPC++ プログラムのデバッグ、プロファイル、解析を支援します。

関連情報

- **Khronos OpenCL*** (英語): ヘテロジニアス・システムの並列プログラミングのオープン・スタンダード
- **Khronos SYCL*** (英語): OpenCL* 向けの C++ シングルソース・ヘテロジニアス・プログラミング
- Daniel Cederman、Philippas Tsigas 著「**GPU-Quicksort: A practical Quicksort Algorithm for Graphics Processors (GPU-Quicksort: グラフィックス・プロセッサー向けの実践的なクイックソート・アルゴリズム)**」(英語)
- Robert Ioffe 著「**GPU-Quicksort in OpenCL 2.0: Nested Parallelism and Work-Group Scan Functions (OpenCL* 2.0 の GPU-Quicksort: 入れ子の並列処理とワークグループ・スキャン関数)**」(英語)
- **インテル® oneAPI ツールキット**
- **この記事で使用したコード** (英語)



oneAPI アプリケーションのパフォーマンス最適化

統一された標準ベースのプログラミング・モデルを最大限に活用

Kevin O' Leary インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

現代のワークロードは驚くほど多様であり、プロセッサー・アーキテクチャーも同様です。すべてのワークロードに最適なアーキテクチャーはありません。パフォーマンスを最大化するには、CPU、GPU、FPGA、および将来のアクセラレーターに展開される、スカラー、ベクトル、行列、および空間 (SVMS) アーキテクチャーを組み合わせる必要があります。インテル® oneAPI 製品は、SVMS にわたってアプリケーションを展開するのに必要なリソースを提供します。この補完的なツールキットのセット (ベース・ツールキットと専門分野向けのアドオン) は、プログラミングを簡素化し、効率と革新の向上に役立ちます。

ベータ版インテル® oneAPI ベース・ツールキット (英語) には、プロファイル、設計、デバッグを支援する高度な解析およびデバッグツールが含まれています。

- **ベータ版インテル® VTune™ プロファイラー** (英語) は、CPU、GPU、および FPGA システムでパフォーマンスのボトルネックを見つけるのに役立ちます。
- **ベータ版インテル® Advisor** (英語) は、ベクトル化、スレッド化、およびアクセラレーターへのオフロードを支援します。
- **ベータ版インテルの GDB 拡張** (英語) は、コードの効率良いデバッグを支援します。

パフォーマンス解析ツール

この記事では、ベータ版インテル® oneAPI ベース・ツールキットに含まれる、ベータ版インテル® Advisor とベータ版インテル® VTune™ プロファイラー、およびその新機能に注目します。

ベータ版インテル® Advisor

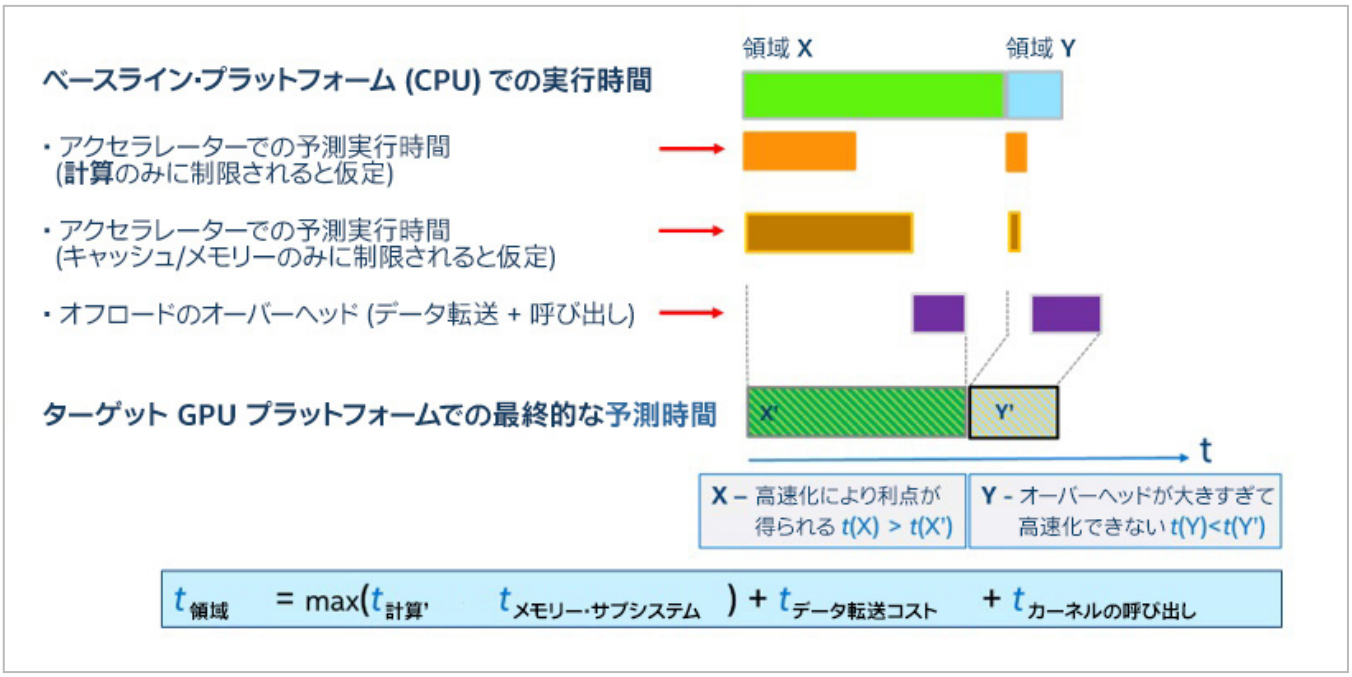
ベータ版インテル® Advisor は、CPU および GPU 上で **DPC++ 言語** (英語) に対応したコードの現代化、プログラミング・アドバイス、パフォーマンス予測を行うように、インテル® Advisor を拡張したものです。C、C++、Fortran、および混合 Python* アプリケーションにおいて、協調設計、パフォーマンス・モデリング、解析、特徴付け機能を提供します。

ベータ版インテル® Advisor には、次のツールが含まれます。

- **オフロード・アドバイザー** : GPU オフロードに最適な候補と適していない領域を特定するのに役立ちます。また、アクセラレーターのパフォーマンス向上やオフロードのオーバーヘッドを予測したり、アクセラレーターのパフォーマンス・ボトルネックをピンポイントで特定することができます。
- **ベクトル化アドバイザー** : 影響の大きい最適化されていないループを特定し、ベクトル化を妨げている要因と安全にベクトル化を強制できる領域を理解するのに役立ちます。
- **スレッド化アドバイザー** : 通常の開発作業を中断することなく、スレッド化設計の選択肢を解析、設計、チューニング、確認するのに役立ちます。
- **ルーフライン解析** : CPU と GPU のパフォーマンスを視覚化して、最大パフォーマンスにどれだけ近づいているかを確認するのに役立ちます。
- (オプション) **ベータ版 oneAPI ベース・ツールキット用インテル® FPGA アドオン** (英語): これらの再構成可能なハードウェア・アクセラレーターをプログラムして、特殊なデータ中心のワークロードを高速化するのに役立ちます (インテル® oneAPI ベース・ツールキットをインストールする必要があります)。

オフロード・アドバイザーのコマンドライン機能を使用することで、ハードウェアが手元になくても、アクセラレーターへ効率良くオフロードするコードを設計できます。コードのパフォーマンスを予測し、データ転送コストと比較できます。再コンパイルは不要です。

ベータ版インテル® Advisor の GPU パフォーマンス評価 (**図 1**) は、限界とボトルネックのパフォーマンス・モデルを使用してスピードアップの上限を予測します。測定された x86 CPU メトリックとアプリケーション特性を入力として受け取り、解析モデルを適用してターゲット GPU 上での実行時間と特性を予測します。



1 **ベータ版インテル® Advisor の GPU パフォーマンス評価**

ルーフライン解析機能は、CPU または GPU コードの計算とメモリーを最適化するのに役立ちます。ボトルネックを特定して、ループやカーネルのパフォーマンスのヘッドルームを決定して、パフォーマンスへの影響が最も大きい最適化を優先します (GPU ルーフライン解析はテクニカルプレビュー機能です)。

ベータ版インテル® VTune™ プロファイラー

ベータ版インテル® VTune™ プロファイラーは、シリアルおよびマルチスレッド・アプリケーションのパフォーマンス解析ツールです。アルゴリズムを解析して、アプリケーションが利用可能なハードウェア・リソースを活用できる場所と方法を特定するのに役立ちます。以下を特定できます。

- アプリケーションやシステム全体で**最も時間を費やしている (ホットな) 関数**
- 利用可能なプロセッサー・リソースを**効率良く利用していないコード領域**
- シーケンシャルとマルチスレッドの両方のパフォーマンスを**最適化するのに最適なコード領域**
- アプリケーションのパフォーマンスに影響する**同期オブジェクト**
- I/O 操作に**アプリケーションが時間を費やしているかどうか、およびその場所と原因**
- **アプリケーションが CPU 依存か GPU 依存か**、および GPU オフロードの効率
- 異なる同期手法、異なるスレッド数、異なるアルゴリズムの**パフォーマンスへの影響**
- **スレッドのアクティビティー**と遷移
- データ共有、キャッシュミス、分岐予測ミスなど、コード中の**ハードウェア関連の問題**

ベータ版インテル® VTune™ プロファイラーは、GPU 解析をサポートする新しい機能も提供しています。

- **GPU オフロード解析** (テクニカルプレビュー機能)
- **GPU 計算 / メディア・ホットスポット解析** (テクニカルプレビュー機能)

GPU オフロード解析 (テクニカルプレビュー機能)

このツールを使用して、プラットフォームの CPU および GPU コアのコード実行を解析し、CPU および GPU アクティビティーを関連付けて、アプリケーションが GPU 依存か CPU 依存かを特定できます。ツールのインフラストラクチャーは、システムのすべてのコアでクロックを自動的に調整するため、統一された時間領域内でいくつかの CPU ベースのワークロードと GPU ベースのワークロードを同時に解析することができます。これにより、以下が可能です。

- アプリケーションが DPC++ または OpenCL* カーネルを**どれくらい効率良く使用しているか特定**できます。
- インテル® メディア SDK タスクの**実行を時系列に解析**できます (Linux* ターゲットのみ)。
- **GPU 利用を調査して**、すべての時点での GPU エンジンのソフトウェア・キューを解析できます。

GPU 計算 / メディア・ホットスポット解析 (テクニカルプレビュー機能)

このツールを使用して、最も時間を費やしている GPU カーネルを解析し、GPU ハードウェア・メトリックに基づいて GPU 利用を特徴付け、メモリー・レイテンシーや非効率なカーネル・アルゴリズムに起因するパフォーマンスの問題を特定し、特定の命令タイプの GPU 命令の頻度を解析します。GPU 計算 / メディア・ホットスポット解析により以下が可能です。

- **GPU 利用率の高い GPU カーネルを調査し**、利用効率を予測し、ストールや低占有率の原因を特定できます。
- 選択した GPU メトリックごとの**アプリケーションのパフォーマンスを時系列に調査**できます。
- **最もホットな DPC++ または OpenCL* カーネルを解析して**、非効率なカーネル・コード・アルゴリズムや適切でないワークアイテムの設定を見つけることができます。

ケーススタディー : ソフトウェア・ツールを使用した oneAPI アプリケーションの最適化

インテル® oneAPI ベース・ツールキットで利用可能ないくつかのツールと機能について分かったところで、例を使って実際に作業してみましょう。このケーススタディーでは、ベータ版インテル® VTune™ プロファイラーとベータ版インテル® Advisor を使用してアプリケーションを最適化します。ルーフライン解析やオフロード・アドバイザーなど、いくつかのベータ版インテル® Advisor の機能を使用して、アプリケーションのボトルネックと、アクセラレーターにオフロードするコード領域を特定します。

行列乗算は、多くのアプリケーションで一般的な操作です。以下は、行列乗算カーネルの例です。

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

アルゴリズムは、3 重の入れ子構造のループで、それぞれの反復で乗算と加算を行います。このようなコードは、計算負荷が高く、多くのメモリーアクセスが行われます。このようなコードの解析にはインテル® Advisor が役立ちます。

ベータ版インテル® Advisor を利用した GPU への移行

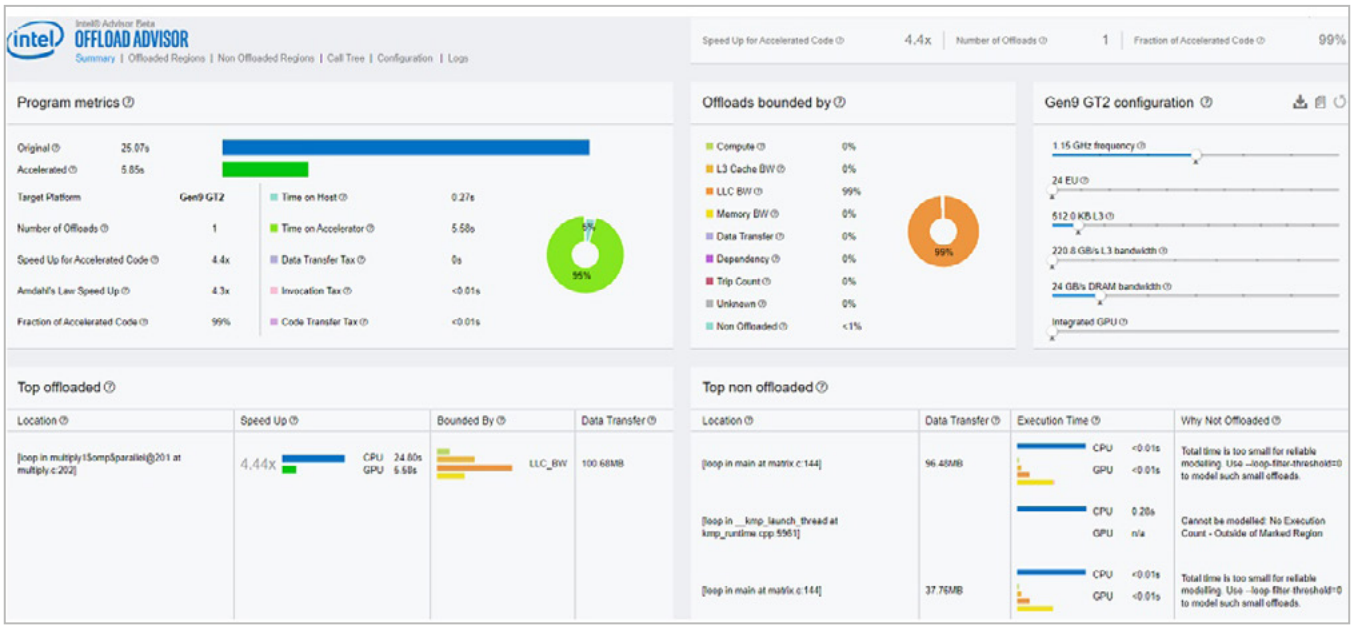
ベータ版インテル® Advisor には、GPU へオフロードすることで利点が得られるコード領域を特定する機能があります。また、GPU で実行した場合のコードのパフォーマンスを予測でき、いくつかの基準に基づいて実験することができます。ベータ版インテル® Advisor で DPC++ コードを解析する場合、2 段階の解析が必要です。

```
$ source $ADVISOR_INSTALL/env/vars.sh

$ advixe-python $APM/collect.py --config gen9 advisor_project
/home/matrix

$ advixe-python $APM/analyze.py --config gen9 advisor_project --out-
dir /home/test/analyze
```

図 2 のスクリーンショットは、CPU 実行時間とアクセラレーター (この例では GPU) で実行した場合の予測時間を示しています。オフロード領域の数とスピードアップも示しています。また、オフロードを制限する要因も分かります。この例では、ラスト・レベル・キャッシュの帯域幅 (LLC BW) によって 99% 制限されています。



2 **CPU 実行時間と予測 GPU 実行時間**

レポートの [Summary] セクションには、以下の情報が表示されます。

- [Program metrics] ペインの**オリジナルの CPU 実行時間**、GPU アクセラレーターでの予測実行時間、オフロード領域の数、スピードアップ。
- **オフロードを制限する要因**。この例では、オフロードはラスト・レベル・キャッシュ (LLC) 帯域幅によって 99% 制限されています。
- GPU へのオフロードにより利点が得られる**上位のオフロードコード領域の正確なソース行**。この例では、オフロードが推奨されるコード領域は 1 つのみです。
- さまざまな理由によりオフロードが推奨されない**上位の非オフロードコード領域の正確なソース行**。この例では、2 つのループは費やされた時間が短すぎて正確にモデル化できず、1 つのループはオフロードコード領域外にあります。

この情報を基に、行列乗算カーネルを DPC++ で記述し直します。

行列乗算カーネルを DPC++ で記述し直す

ベータ版インテル® Advisor は、**図 3** に示すように、オフロード領域の正確なソース行を示します。また、ループの計算時間が短すぎて正確にモデル化できないか、ループがマークされた領域外にあるため、オフロードする必要がないループも示します (**図 4**)。

Top offloaded

Location	Speed Up	Data Transfer	Execution Time	Bounded By
[loop in multiply1ompparallel@179 at multiply.c:180]	6.07x	100.68MB	C... 17.00s GPU 2.80s	L3_BW

3 上位のオフロード領域

Top non offloaded

Location	Data Transfer	Execution Time	Why Not Offloaded
[loop in __kmp_launch_thread at kmp_runtime.cpp:5961]			Cannot be modelled: No Execution Count - Outside of Marked Region
[loop in main at matrix.c:144]			Total time is too small for reliable modelling. Use --loop-filter-threshold=0 to model such small offloads.
[loop in main at matrix.c:144]			Total time is too small for reliable modelling. Use --loop-filter-threshold=0 to model such small offloads.

4 上位の非オフロード領域

次の手順に従って、以下のコード例に示すように、行列乗算カーネルを DPC++ で記述し直します。

1. オフロードデバイスを**選択**します。
2. デバイスキューを**宣言**します。
3. 行列を格納するバッファーを**宣言**します。
4. デバイスキューにワークを**送信**します。
5. 行列乗算を並列に**実行**します。

```
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM])
{
           int i,j,k;

  // Select device
  cl::sycl::gpu_selector device;
  // Declare a deviceQueue
  cl::sycl::queue deviceQueue(device);
  // Declare a 2 dimensional range
  cl::sycl::range<2> matrix_range{NUM, NUM};

  // Declare 3 buffers and Initialize them
  cl::sycl::buffer<TYPE, 2> bufferA((TYPE*)a, matrix_range);
  cl::sycl::buffer<TYPE, 2> bufferB((TYPE*)b, matrix_range);
  cl::sycl::buffer<TYPE, 2> bufferC((TYPE*)c, matrix_range);

  // Submit our job to the queue
  deviceQueue.submit([&](cl::sycl::handler& cgh) {
    // Declare 3 accessprs to our buffers. The first 2 read and the last read_write
    auto accessorA = bufferA.template get_access<sycl_read>(cgh);
    auto accessorB = bufferB.template get_access<sycl_read>(cgh);
    auto accessorC = bufferC.template get_access<sycl_read_write>(cgh);

    // Execute Matrix multiply in parallel over our matrix_range
    // Ind is an index into this range
    cgh.parallel_for<class Matrix<TYPE>>(matrix_range,
                     [=](cl::sycl::id<2> ind) {
            int k;
               for(k=0; k<NUM; k++) {
                  // Perform computation ind[0] is row, ind[1] is col
               accessorC[ind[0]][ind[1]]  += accessorA[ind[0]][k] * accessorB[k][ind[1]];
            }
     });
  });
}
```

ベータ版インテル® VTune™ プロファイラーを利用した GPU 利用の最適化

オフロード・アドバイザーにより CPU カーネルを GPU に移行しましたが、初期の実装は最適とは程遠いものです。ベータ版インテル® VTune™ プロファイラーの GPU オフロード機能を使用して、どれくらい効率良く GPU を利用しているか確認します (**図 5**)。GPU オフロードレポートは、アプリケーションの経過時間が 2.017 秒で、GPU 利用率が 100% であることを示しています。また、行列乗算がホットスポットであることが分かります。

GPU Offload (Preview) GPU Offload (Preview)

INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Graphics | Platform

Elapsed Time: 2.017s

GPU Usage: 47.8%

Use this section to understand whether the GPU was utilized properly and which of the engines were utilized. Identify the amount of gaps in the GPU utilization that potentially could be loaded with some work. This metric is calculated for the engines that had at least one piece of work scheduled to them.

GPU Usage

GPU Usage breakdown by GPU engines and work types.

GPU Engine / Packet Type	GPU Time	GPU Utilization
Render and GPGPU	0.964s	47.8%
Unknown	0.964s	47.8%

*N/A is applied to non-summable metrics.

Packet Queue Depth Histogram

Packet Duration Histogram

Hottest GPU Computing Tasks

This section lists the most active computing tasks running on the GPU, sorted by the Total Time. Focus on the computing tasks flagged as performance-critical.

Computing Task	Total Time	Total Compute Time	Total Transfer Time (f)
Matrix<float>	3.980s	0.961s	3.019s
clEnqueueReadBufferRect	0.000s	0.000s	0.000s

*N/A is applied to non-summable metrics.

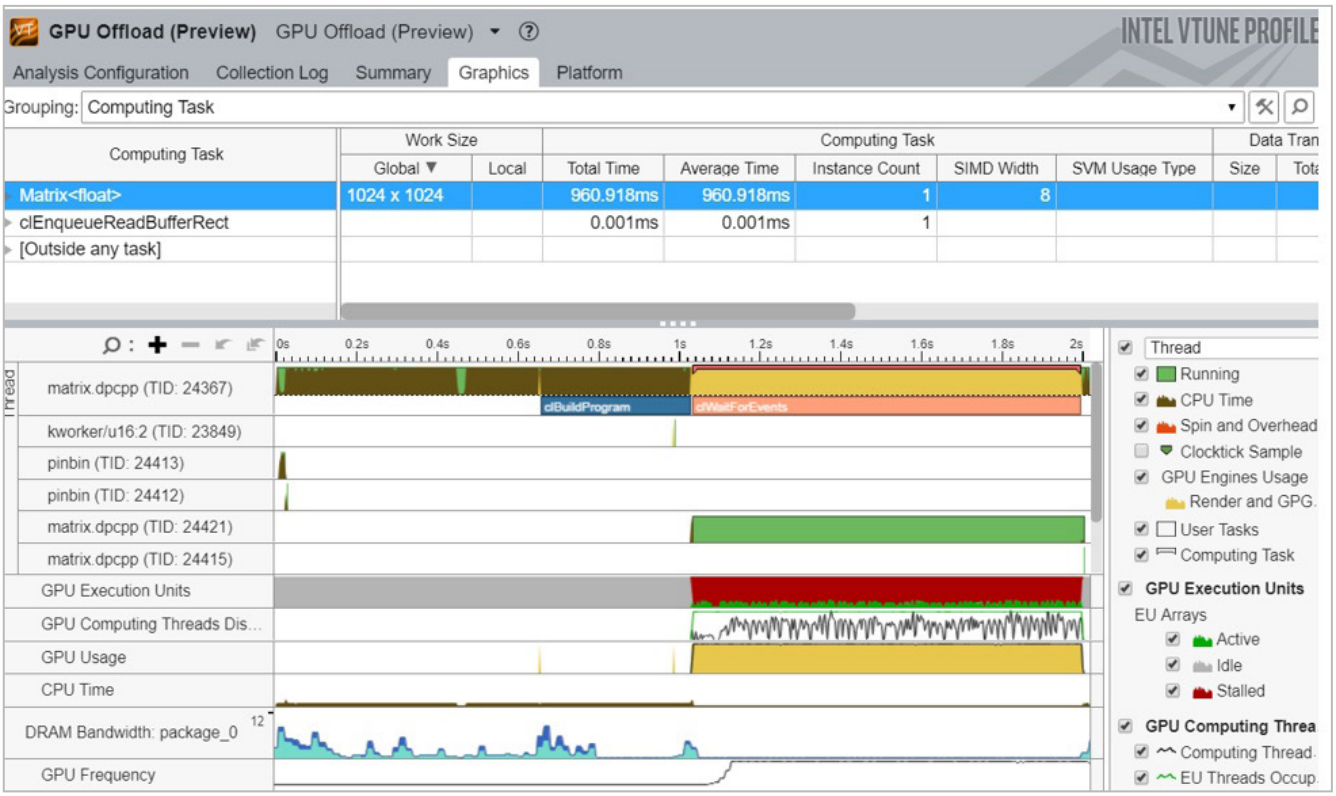
5 **GPU オフロードレポート**

[Graphics] タブや [Platform] タブに切り替えると、詳しい情報を確認できます。ベータ版インテル® VTune™ プロファイラーは、CPU と GPU 間の同期されたタイムラインを表示します。GPU オフロードは、GPU 実行ユニットがストールしていることを示しています (**図 6** のタイムラインの赤いバー)。

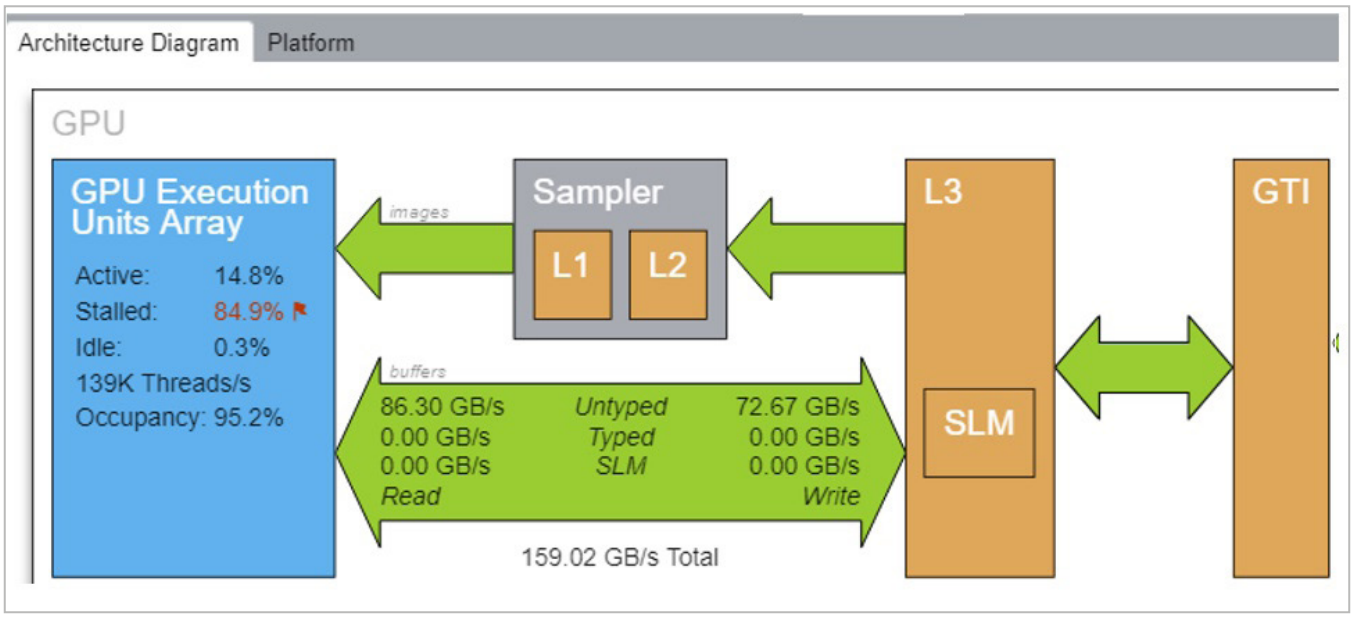
次に、ベータ版インテル® VTune™ プロファイラーの GPU ホットスポット・レポートを実行して、低い GPU 利用率とストールの原因を特定します。GPU ホットスポットの [Graphics] タブをクリックすると、アーキテクチャーの高レベルのダイアグラムが表示されます (**図 7**)。共有ローカルメモリー (SLM) キャッシュを使用していないことが分かります。また、合計で約 159.02GB/ 秒のデータを移動していることが分かります。

次の 2 つの最適化手法を試してみます。

1. 行列のキャッシュ・ブロッキング
2. ローカルメモリーの使用



6 GPU 実行ユニットのストール



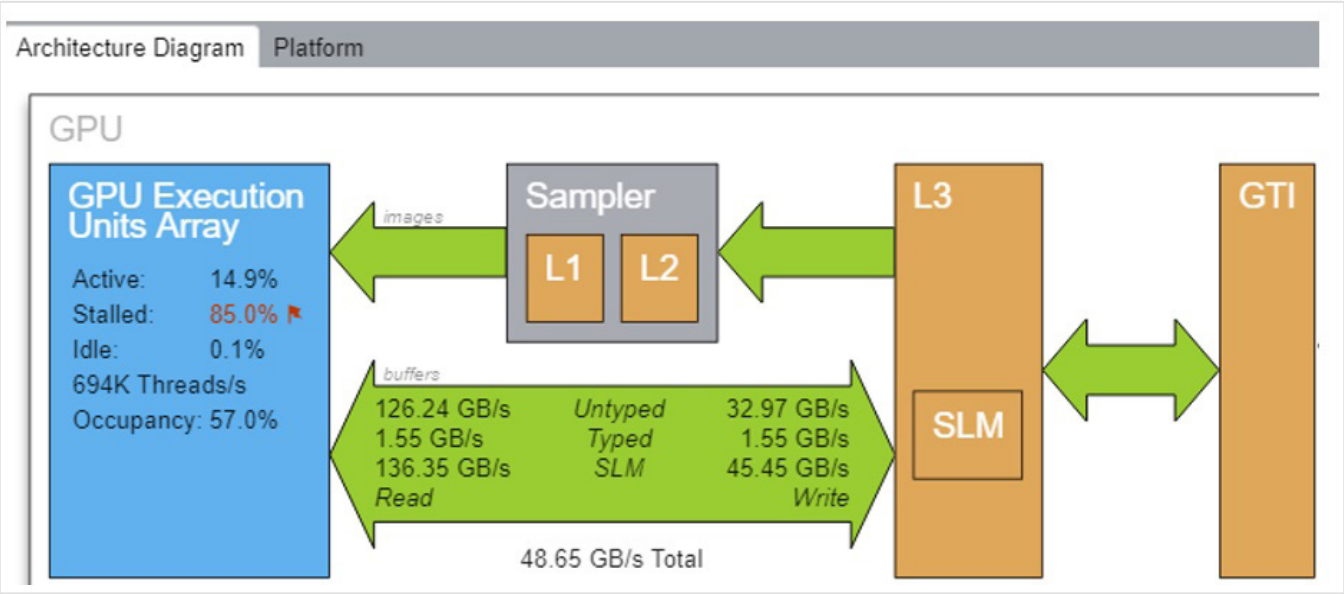
7 メモリー最適化前のアーキテクチャー・ダイアグラム

これらの手法を実装するには、行列をタイルに分割して、SLM キャッシュで個別に処理する必要があります。

```
deviceQueue.submit([&](cl::sycl::handler& cgh) {
    // Declare 3 accessors to our buffers. The first 2 read and the last read_write
    auto accessorA = bufferA.template get_access<sycl_read>(cgh);
    auto accessorB = bufferB.template get_access<sycl_read>(cgh);
    auto accessorC = bufferC.template get_access<sycl_read_write>(cgh);

           //Create matrix tiles
    cl::sycl::accessor<TYPE, 2, cl::sycl::access::mode::read_write, cl::sycl::access::target::local>
aTile(cl::sycl::range<2>(MATRIX_TILE_SIZE, MATRIX_TILE_SIZE), cgh);
    cl::sycl::accessor<TYPE, 2, cl::sycl::access::mode::read_write, cl::sycl::access::target::local>
bTile(cl::sycl::range<2>(MATRIX_TILE_SIZE, MATRIX_TILE_SIZE), cgh);
    // Execute matrix multiply in parallel over our matrix_range
    // ind is an index into this range
    cgh.parallel_for<class Matrix2<TYPE>>(cl::sycl::nd_range<2>(matrix_range,tile_range),
                     [=](cl::sycl::nd_item<2> it) {
            int k;
            const int numTiles = NUM/MATRIX_TILE_SIZE;
            const int row = it.get_local_id(0);
            const int col = it.get_local_id(1);
            const int globalRow = MATRIX_TILE_SIZE*it.get_group(0) + row;
            const int globalCol = MATRIX_TILE_SIZE*it.get_group(1) + col;
            TYPE acc = 0.0;
            for (int t=0; t<numTiles; t++) {
                 const int tiledRow = MATRIX_TILE_SIZE*t + row;
                 const int tiledCol = MATRIX_TILE_SIZE*t + col;
                 aTile[row][col] = accessorA[globalRow][tiledCol];
                 bTile[row][col] = accessorB[tiledRow][globalCol];
                 it.barrier(cl::sycl::access::fence_space::local_space);
                 for(k=0; k<MATRIX_TILE_SIZE; k++) {
                     // Perform computation ind[0] is row, ind[1] is col
                     acc += aTile[row][k] * bTile[k][col];
                 }
                 it.barrier(cl::sycl::access::fence_space::local_space);
            }
            accessorC[globalRow][globalCol] = acc;
     });
  });
```

新しいアーキテクチャー・ダイアグラムは、はるかに効率的です (**図 8**)。SLM を利用して、136.35GB/ 秒を読み取り、45.45GB/ 秒を書き込んでいます。



8 **メモリー最適化後のアーキテクチャー・ダイアグラム**

[Platform] タブ (**図 9**) をクリックして追加のメトリックを確認します。行列は 1024x1024 のグローバルメモリーに格納されていますが、16 x 16 タイルでローカルメモリーを利用しています。新しいメトリックの経過時間は 1.22 秒で、最適化前の 1.64 倍に向上しています。

Grouping: Computing Task

Computing Task	Work Size		Computing Task					
	Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width	SVM Usage Type	Siz
Matrix2<float>	1024 x 1024	16 x 16	187.410ms	187.410ms	1	8		
clEnqueueReadBufferRect			0.002ms	0.002ms	1			
[Outside any task]								

9 **[Platform] タブ**

インテル® Advisor の GPU ルーフライン

行列乗算カーネルの GPU バージョンがハードウェアのパフォーマンスを最大限に引き出せているか確認するため、新しい GPU ルーフライン機能を使用します。ベータ版インテル® Advisor は、インテルの GPU で実行するカーネルのルーフライン・モデルを生成できます。ルーフライン・モデルは、カーネルを特徴付けて、理想的なパフォーマンスからどれくらい離れているか視覚化する非常に優れた機能を提供します。

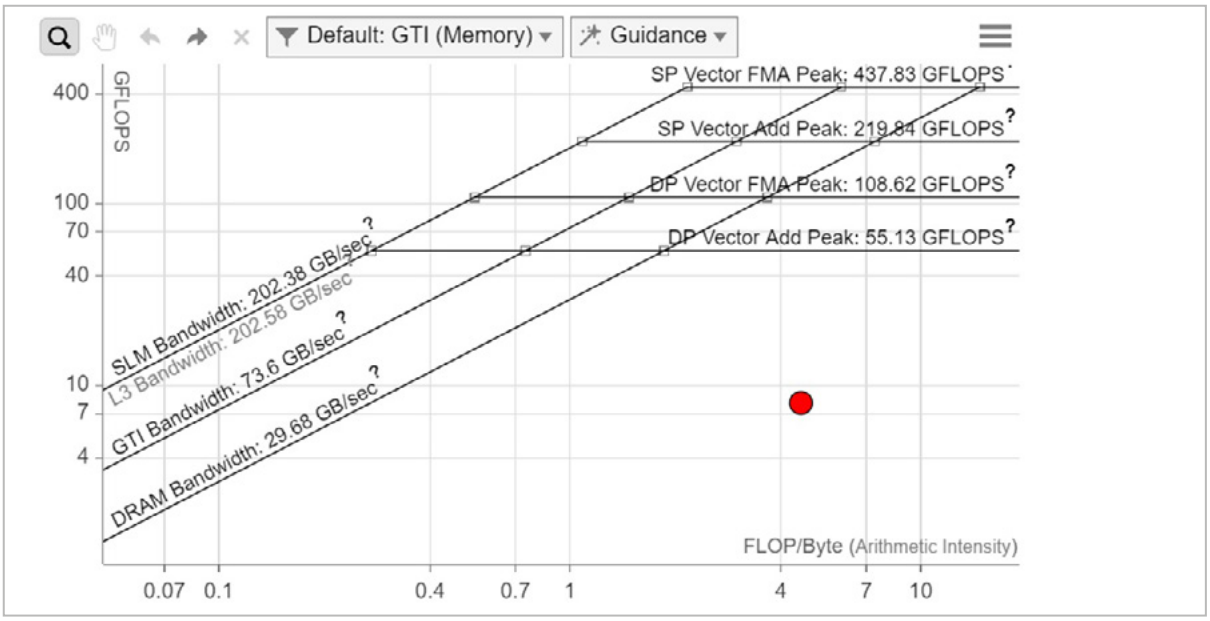
GPU 上のルーフライン・モデルはテクニカルプレビュー機能で、デフォルトでは無効になっています。次の 5 つのステップで有効にできます。

- 最初に、DPC++ コードが GPU 上で正しく動作することを確認します。次のコードを使用して、実行しているハードウェアを確認できます。

```
Cl::sycl::default_selector selector;
Cl::sycl::queue queue(delector);
auto d = queue.get_device();
std::cout<<"Running on
:"<<d.get_info<cl::sycl::info::device::name>()<<std::endl;
```

- これはテクニカルプレビュー機能であるため、次の環境変数を設定して GPU プロファイルを有効にする必要があります。
  export ADVIXE_EXPERIMENTAL=gpu-profiling
- 次に、**--enable-gpu-profiling** オプションを使用してサーベイを実行します。
  advixe-cl -collect survey --enable-gpu-profiling --project-dir <my_project_directory> --search-dir src:r=<my_source_directory> -- ./myapp param1 param2
- **--enable-gpu-profiling** オプションを使用してトリップカウント解析を実行します。
  advixe-cl -collect tripcounts --stacks --flop --enable-gpu-profiling --project-dir <my_project_directory> --search-dir src:r=<my_source_directory> -- ./myapp param1 param2
- ルーフライン・モデルを生成します。
  advixe-cl --report=roofline --gpu --project-dir <my_project_directory> --report-output=roofline.html

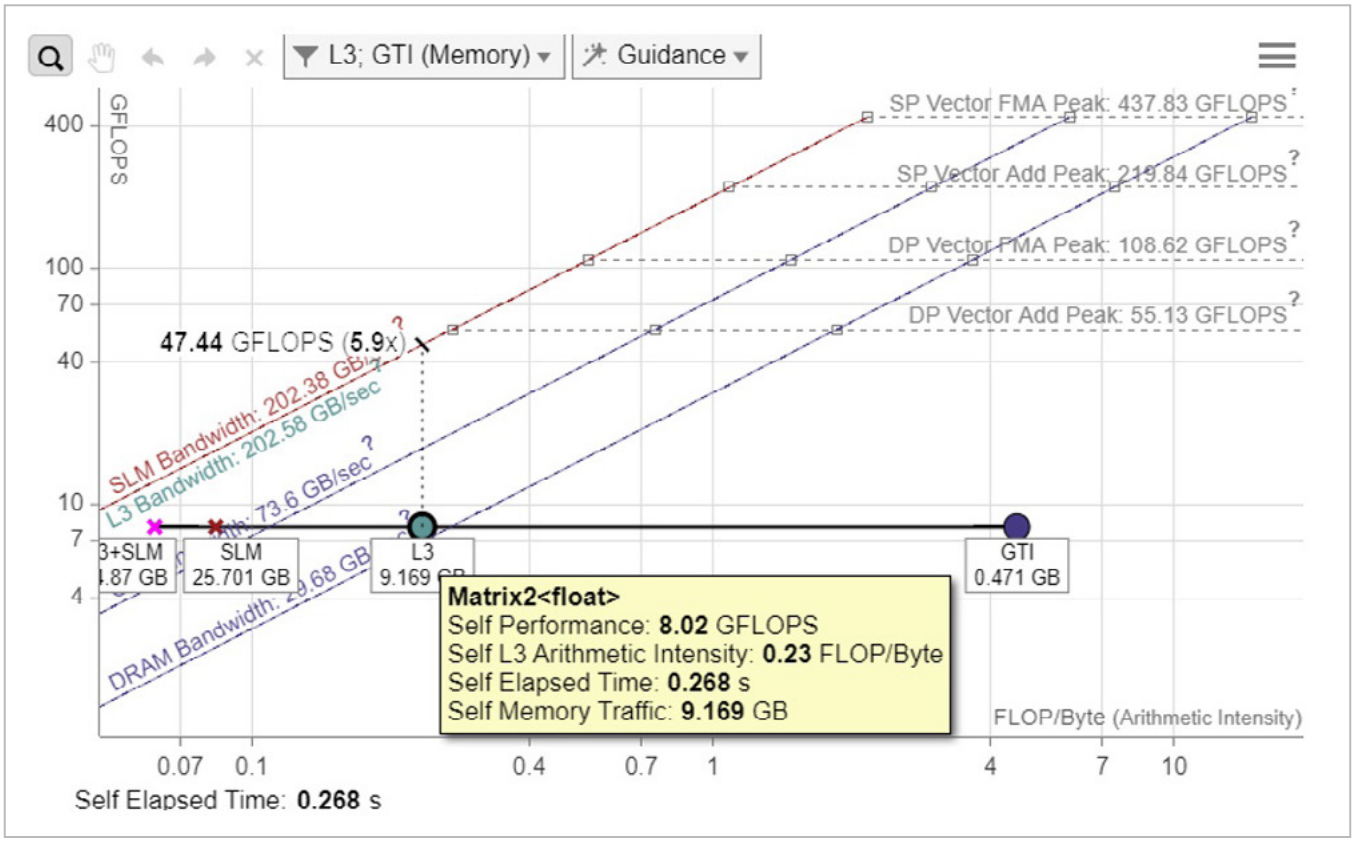
最後のステップを実行すると、ファイル **roofline.html** が生成され、任意のウェブブラウザーで開くことができます (**図 10**)。



10 **ルーフライン・レポート**

演算強度の計算に使用されるメモリー・サブシステムに基づいて異なるドットを表示することもできます(**図 11**)。

11 メモリーレベル



12 ルーフライン・グラフ

図 12 のルーフライン・グラフから、L3 ドットが L3 最大帯域幅に接近していることが分かります。FLOPS を向上するには、さらにキャッシュ利用率を最適化する必要があります。キャッシュ・ブロッキング手法は、メモリーを効率良く利用し、パフォーマンスを向上します。GTI (GPU、CPU、アンコア [LLC]、およびメインメモリー間のトラフィック) ドットは GTI ルーフラインからかなり離れているため、CPU と GPU 間の転送コストは問題になりません。

コードに集中

インテル® oneAPI 製品は、CPU とアクセラレーターに展開される、スカラー、ベクトル、行列、および空間アーキテクチャーでシームレスに動作する、標準の簡素化されたプログラミング・モデルを提供します。インテル® oneAPI 製品を利用することで、ユーザーは最適なマシン命令を生成するメカニズムではなく、コードに集中することができます。

関連情報

- **oneAPI イニシアチブ (英語)**
- **ベータ版インテル® oneAPI ツールキット**
- **ベータ版インテル® Advisor (英語)**
- **ベータ版インテル® VTune™ プロファイラー (英語)**

インテル® oneMKL を使用したモンテカルロ・シミュレーションの高速化

ベータ版インテル® oneAPI マス・カーネル・ライブラリーのデータ並列 C++ 使用モデル

Alina Elizarova インテル コーポレーション マス・アルゴリズム・エンジニア
Pavel Dyakov インテル コーポレーション マス・アルゴリズム・エンジニア
Gennady Fedorov インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

ベータ版インテル® oneAPI マス・カーネル・ライブラリー (ベータ版インテル® oneMKL) (英語) は、工学、科学、金融系アプリケーション向けに拡張された数学ルーチンを開発者およびデータ・サイエンティストに提供します。このライブラリーを使用して、現在および将来の世代のインテル® CPU/GPU 向けにコードを最適化できます。この記事では、ベータ版インテル® oneMKL で追加された乱数ジェネレーターを利用して、モンテカルロ・シミュレーションの例に**データ並列 C++ (DPC++)** (英語) 使用モデルを適用します。

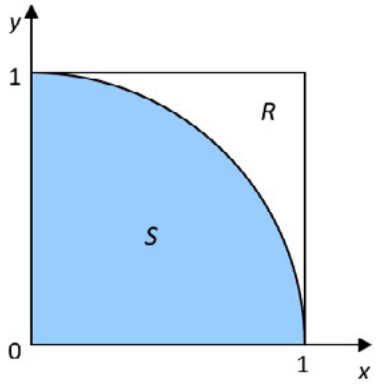
次の 5 つの使用モデルを説明します。

1. リファレンス C++
2. ベータ版インテル® oneMKL DPC++
3. ベータ版インテル® oneMKL DPC++ - 拡張版
4. ヘテロジニアス並列実装
5. 異なるメモリー割り当て手法に基づく実装

最終的に、ホストとデバイス間のデータ転送を最小化することにより、パフォーマンスが大きく向上しました。

数値積分による π の計算

モンテカルロ・シミュレーションは、繰り返しランダム・サンプリングを使用して数値結果を得る計算アルゴリズムです[1]。**図 1** は、モンテカルロ法を使用した π の計算方法を示しています。



1 **モンテカルロ法を使用した π の計算**

単位正方形に内接する象限について考えてみます。セクター **S** の面積は **Area(S) = 1/4 πr² = π/4**、正方形 **R** の面積は **Area(R) = 1** に等しくなります。ポイント **c = (x,y)** を単位正方形 **R** からランダムに選択する場合、**c** がセクター **S** 内にある確率は次のように求められます。

$$Pr(c \in S) = \frac{Area(S)}{Area(R)} = \pi/4$$

n 個のこのようなポイントについて考えます。ここで、**n** は十分な大きさであり、**S** 内に該当するポイントの数 **k** をカウントします。確率 $Pr(c \in S)$ は比率 **k/n** または次のように近似できます。

$$\pi/4 \cong k/n$$

π は次のように近似できます。

$$\pi \cong \frac{4k}{n}$$

大数の法則に従って、**n** の数が大きくなるほど **π** の近似の精度は高くなります。任意の **ε > 0** について、次のようにベルヌーイの定理からより正確な値を求めることができます。

$$Pr\left(\left|\frac{k}{n} - 4\pi\right| \geq \varepsilon\right) \leq \frac{1}{4n\varepsilon^2}$$

テストしたポイント **c** の **x** 座標と **y** 座標が **0 ≤ x ≤ 1** (横座標) および **0 ≤ y ≤ 1** (縦座標) の場合、ポイント **c** は次の場合にセクター **S** 内になります。

$$x^2 + y^2 \leq 1$$

このセクションのまとめ :

1. 各ポイントが **[0,1)** 区間の 2 つの一様分布乱数で表される、**n** 個の 2D ポイントを生成します。
2. セクター **S** 内のポイントの数をカウントします。
3. このページの最初の式を使用して **π** の近似値を計算します。

リファレンス C++ の π 推定の例

n_points 個の 2D ポイントを受け取り上記の計算を行う **estimate_pi** 関数について考えてみましょう。

```
float estimate_pi( size_t n_points ) {
    float estimated_pi;          // Estimated value of Pi
    size_t n_under_curve = 0;    // Number of points fallen under the curve

    // Allocate storage for random numbers
    std::vector<float> x(n_points);
    std::vector<float> y(n_points);

    // Step 1. Generate n_points random numbers
    // 1.1. Generator initialization
    std::default_random_engine engine(SEED);
    std::uniform_real_distribution<float> distr(0.0f, 1.0f);

    // 1.2. Random number generation
    for(int i = 0; i < n_points; i++) {
        x[i] = distr(engine);
        y[i] = distr(engine);
    }

    // Step 2. Count the number of points fallen under the curve
    for ( int i = 0; i < n_points; i++ ) {
        if (x[i] * x[i] + y[i] * y[i] <= 1.0f)
            n_under_curve++;
    }

    // Step 3. Calculate approximated value of Pi
    estimated_pi = n_under_curve / ((float)n_points) * 4.0;
    return estimated_pi;
}
```

この例は、C++ 11 標準の乱数ジェネレーターを使用しています。ステップ 1 では、エンジン (**engine**) と分布 (**distr**) の 2 つのインスタンスを作成して乱数ジェネレーターを初期化します。**engine** はジェネレーターの状態を保持して独立した一様分布の確率変数を提供し、**distr** は統計とパラメーターを使用してジェネレーター出力の変換を表します。この例では、**uniform_real_distribution** は区間 **[a, b)** に一様に分布しているランダムな浮動小数点値を生成します。

ステップ 1.2 では、**engine** を **distr** に渡して、単一の浮動小数点変数 (乱数) を取得します。ループは、ベクトル **x** と **y** に乱数を格納します。ステップ 2 では、位置 **(x, y)** をそれぞれチェックしてセクター **S** 内のポイントの数をカウントし、結果を **n_under_curve** 変数に格納します。最後に、**π** の推定値を計算してメインプログラムに返します (ステップ 3)。

ベータ版インテル® oneMKL DPC++ の π 推定の例

estimate_pi 関数に **cl::sycl::queue** を追加します。DPC++ では、**セレクター・インターフェイス** (英語) を使用して実行するデバイスを選択できます。

```
float estimate_pi(size_t n_points) {
    float estimated_pi;          // Estimated value of Pi
    size_t n_under_curve = 0;    // Number of points fallen under the curve

    // Allocate storage for random numbers
    cl::sycl::buffer<float, 1> x_buf(cl::sycl::range<1>{n_points});
    cl::sycl::buffer<float, 1> y_buf(cl::sycl::range<1>{n_points});

    // Choose device to run on and create queue
    cl::sycl::gpu_selector selector;
    cl::sycl::queue queue(selector);

  std::cout << "Running on: " <<
        queue.get_device().get_info<cl::sycl::info::device::name>() <<
        std::endl;

    // Step 1. Generate n_points random numbers
    // 1.1. Generator initialization
    mkl::rng::philox4x32x10 engine(queue, SEED);
    mkl::rng::uniform<float, mkl::rng::standard> distr(0.0f, 1.0f);

    // 1.2. Random number generation
    mkl::rng::generate(distr, engine, n_points, x_buf);
    mkl::rng::generate(distr, engine, n_points, y_buf);

    // Step 2. Count the number of points fallen under the curve
    auto x_acc = x_buf.template get_access<cl::sycl::access::mode::read>();
    auto y_acc = y_buf.template get_access<cl::sycl::access::mode::read>();
    for ( int i = 0; i < n_points; i++ ) {
        if (x_acc[i] * x_acc[i] + y_acc[i] * y_acc[i] <= 1.0f)
            n_under_curve++;
    }
    // Step 3. Calculate approximated value of Pi
    estimated_pi = n_under_curve / ((float)n_points) * 4.0;
    return estimated_pi;
}
```

cl::sycl::queue はベータ版インテル® oneMKL の関数の入力引数です。ライブラリーのカーネルはこのキューで送信され、デバイス (ホストやアクセラレーター) を切り替えるためにコードを変更する必要はありません。ベータ版インテル® oneMKL では、CPU および GPU デバイスを利用できます。

乱数の格納には、**std::vector** の代わりに **cl::sycl::buffer** を使用します。

```
// Reference:
    std::vector<float> x(n_points);
    std::vector<float> y(n_points);

    // Data Parallel C++:
    cl::sycl::buffer<float, 1> x_buf(cl::sycl::range<1>{n_points});
    cl::sycl::buffer<float, 1> y_buf(cl::sycl::range<1>{n_points});
```

バッファーは、ホスト・アプリケーションとデバイスカーネル間のデータ転送を管理します。**buffer** クラスと **accessor** クラスはメモリー転送を追跡し、異なるカーネル間でデータの一貫性を保証します。

ステップ 1 では、ベータ版インテル® oneMKL の乱数ジェネレーター API で 2 つのエンティティー (乱数ジェネレーターのエンジンと乱数分布) を初期化します。エンジンは、コンストラクターの入力として **cl::sycl::queue** と初期 **SEED** 値を受け取ります。分布 **mkl::rng::uniform** には、出力値の型とエンジンの出力の変換に使用されるメソッドのテンプレート・パラメーター (詳細は『**インテル® oneAPI マス・カーネル・ライブラリー・デベロッパー・リファレンス - C**』(英語) を参照) と分布のパラメーターがあります。

ステップ 1.2 では、**mkl::rng::generate** 関数を呼び出して乱数を取得します。この関数は、以前のステップで作成した分布とエンジン、生成する要素の数、バッファー内の結果のストレージを受け取ります。ベータ版インテル® oneMKL の乱数ジェネレーター API **mkl::rng::generate()** はベクトル化されています。多くの場合、ベクトルバージョンのライブラリー・サブルーチンのほうがスカラーバージョンよりも効率良く実行されます (詳細は「**インテル® MKL ベクトル統計のノート**」(英語) を参照)。

ステップ 2 では、ホストで乱数を後処理し、データにアクセスするバッファーのホストアクセサーを生成します。

```
auto x_acc = x_buf.template get_access<cl::sycl::access::mode::read>();
auto y_acc = y_buf.template get_access<cl::sycl::access::mode::read>();
```

その他のステップは、リファレンス C++ の例と同じです。

ベータ版インテル® oneMKL DPC++ - 拡張版の π 推定の例

前の例のステップ 2 は、DPC++ の並列 STL 関数を使用して最適化できます。このアプローチは、ホストとデバイス間のデータ転送を減らして、パフォーマンスを向上します。ステップ 2 を次のように変更します。

```
auto policy = dpstd::execution::make_sycl_policy<class count>(queue);

  auto x_buf_begin = dpstd::begin(x_buf);
  auto y_buf_begin = dpstd::begin(y_buf);
  auto zip_begin = dpstd::make_zip_iterator(x_buf_begin, y_buf_begin);

  n_under_curve = std::count_if(policy, zip_begin, zip_begin + n_points,
    [](auto p) {
      using std::get;
      float x, y;
      x = get<0>(p);
      y = get<1>(p);
      return x*x + y*y <= 1.0f;
    });
```

zip イテレーターは **count_if** 関数の入力として乱数のペアを提供します。その他のステップは、前のベータ版インテル® oneMKL DPC++ の例と同じです。

統合共有メモリー (USM) ベースのベータ版インテル® oneMKL DPC++ の π 推定の例

DPC++ のポインターベースのメモリー管理には、**cl::sycl::malloc** と **cl::sycl::usm_allocator** の 2 つのアプローチがあります (詳細は**こちら** (英語) を参照)。

cl::sycl::malloc アプローチは、ホストまたはデバイスで直接メモリーを割り当てるか、ホストとデバイスの両方からメモリーにアクセスします。

```
float* x = (float*) cl::sycl::malloc_shared(n_points * sizeof(float),
queue.get_device(), queue.get_context());
```

このメモリー割り当てを使用すると、ポインター演算の長所をそのまま活用できます。このアプローチでは、割り当てたメモリーはすべて解放する必要があります。

cl::sycl::usm_allocator アプローチは、メモリー管理を考慮することなく、標準またはユーザーコンテナーを使用して作業することができます。

```
// Create usm allocator
  cl::sycl::usm_allocator<float, cl::sycl::usm::alloc::shared>
  allocator(queue.get_context(), queue.get_device());
  // Allocate storage for random numbers
  std::vector <float, cl::sycl::usm_allocator<float,
  cl::sycl::usm::alloc::shared>> x(n_points, allocator);
```

乱数の生成は同じ方法で行いますが、**cl::sycl::buffer<float, 1>** の代わりに **float*** を使用します。

```
auto event = mkl::rng::generate(distr, engine, n_points, x.data());
```

各関数は、同期に使用できる **cl::sycl::event** を返します。これらのイベントまたはキュー全体に対して **wait()** 関数または **wait_and_throw()** 関数を呼び出すことができます。**event.wait()** 関数または **queue.wait()** 関数を呼び出すことにより、DPC++ カーネル間のデータ依存関係を手動で制御できます。

乱数を取得してステップ 2 でポイントをカウントするには、ホストアクセサーを作成せずに、ベクトル / ポインターを使用します。

```
for ( int i = 0; i < n_points; i++ ) {
   if (x[i] * x[i] + y[i] * y[i] <= 1.0f)
     n_under_curve++;
 }
```

ベータ版インテル® oneMKL DPC++ の π 推定のヘテロジニアス実行

前の例を変更して、すべての計算をホストで行う代わりに一部の計算をアクセラレーターにオフロードします。API は同じで、**cl::sycl::queue** のデバイスを選択します。異なるデバイスで並列に実行するには、2 つのキューが必要です。

```
// Create queues for Host and GPU
  cl::sycl::queue queue_gpu(cl::sycl::gpu_selector(), exception_handler);
  cl::sycl::queue queue_host(cl::sycl::host_selector(),
  exception_handler);
```

メモリー割り当ても別に行います。CPU 割り当てはホストで直接行うことができます。

```
// Create usm allocators for shared and Host memory allocation
cl::sycl::usm_allocator<float, cl::sycl::usm::alloc::shared>
allocator_gpu(queue_gpu.get_context(), queue_gpu.get_device());
// Allocate storage for random numbers
std::vector <float, decltype(allocator_gpu)> x(n_points, allocator_gpu);
std::vector <float> y(n_points);
```

ベータ版インテル® oneMKL の乱数ジェネレーター・エンジンはデバイスの種類ごとに構築する必要があるため、2 つのオブジェクトが必要です。2 つ目のエンジンは別のシードで初期化するか、**n_points** でオフセットされたシーケンスを継続します (詳細は「**インテル® MKL ベクトル統計のノート**」(英語) を参照)。

```
mkl::rng::philox4x32x10 engine_gpu(queue_gpu, SEED);
mkl::rng::philox4x32x10 engine_host(queue_host, SEED);
mkl::rng::skip_ahead(engine_host, n_points);
```

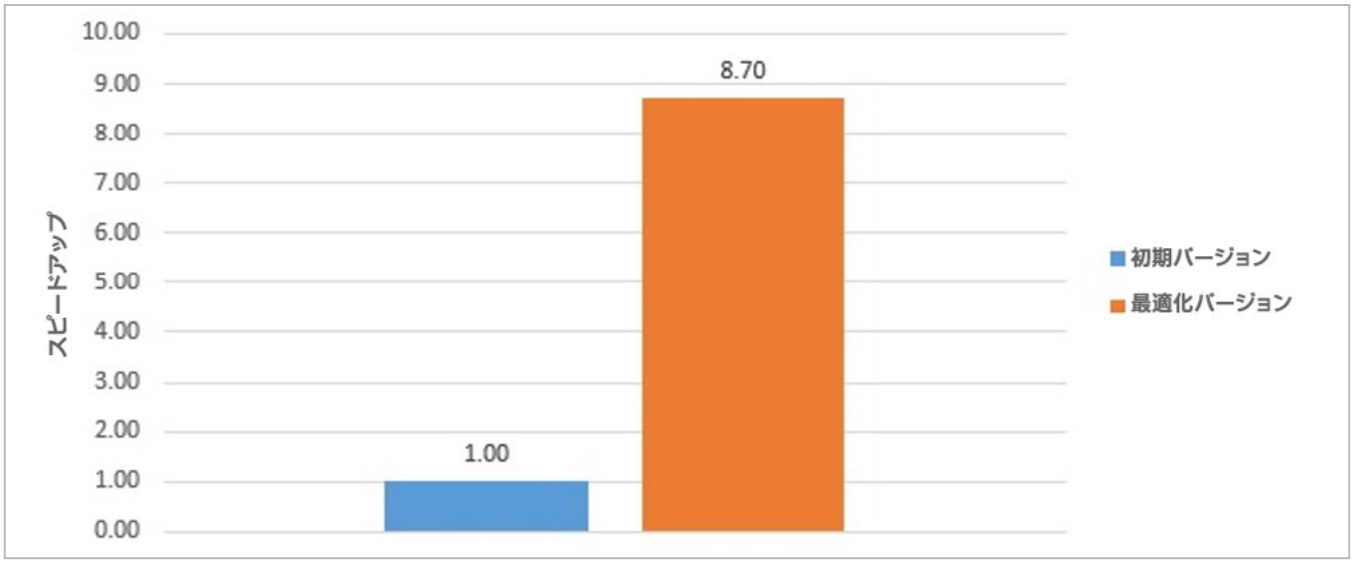
乱数の生成と後処理は前の例と同じように呼び出します。この例では、並列実行ポリシーで **std::count_if** をホストで使用しています。

```
auto event1 = mkl::rng::generate(distr, engine_gpu, n_points, x.data());
auto event2 = mkl::rng::generate(distr, engine_host, n_points,
y.data());
// Wait to finish generation
event1.wait_and_throw();
event2.wait_and_throw();
n_under_curve = std::count_if(std::execution::par,
dpstd::make_zip_iterator(x.begin(),y.begin()),
dpstd::make_zip_iterator(x.end(),y.end()), [](auto p) {
using std::get;
float dx, dy;
dx = get<0>(p);
dy = get<1>(p);
return dx*dx + dy*dy <= 1.0f;
```

このアプローチは、単一 API 内のサポートしているデバイス間でワークを分散し、複数のデバイスで異なるタスクを並列に実行できます。

パフォーマンスの比較

図 2 は、ベータ版インテル® oneMKL DPC++ の例とベータ版インテル® oneMKL DPC++ 拡張版の例を比較したものです。



2 **パフォーマンスの比較**

ハードウェアとソフトウェアの構成 :

- **ハードウェア :** インテル® Core™ i9-9900K プロセッサー @ 3.60GHz、インテル® Gen12LP HD グラフィックス、NEO グラフィックス
- **オペレーティング・システム :** Ubuntu* 18.04.2 LTS
- **ソフトウェア :** ベータ版インテル® oneMKL

シミュレーションのパラメーター :

- **生成した 2D ポイントの数 :** 10^8
- **乱数エンジン :** `mkl::rng::philox4x32x10`
- **乱数分布 :** 単精度一様分布
- **測定範囲 :** `estimate_pi()` 関数の計算部分 (メモリー割り当てのオーバーヘッドを除く)

パフォーマンスの向上

この記事では、ベータ版インテル® oneMKL DPC++ の異なる使用モデルをモンテカルロ・シミュレーションによる **π** の推定に適用しました。リファレンス C++ の例を少し変更し、ベータ版インテル® oneMKL DPC++ の機能と関数を使用することで、異なるサポートデバイスで (ヘテロジニアス実行を含む)、コードを実行することができました。**図 2** に示すように、ホストとデバイス間のデータ転送を減らすことにより、パフォーマンスが大きく向上しました。

参考資料

1. Knuth, Donald E. The Art of Computer Programming, Volume 2, Seminumerical Algorithms, 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

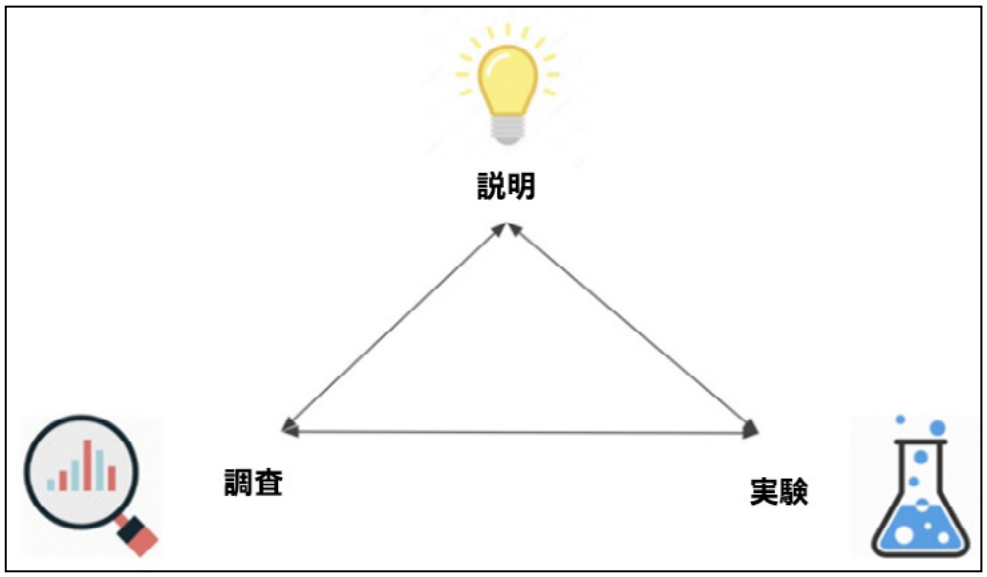
大規模で高速なアナリティクスをインテル® アーキテクチャーで行う

最新のハードウェアで従来のアナリティクスを利用してデータサイエンスを統一

Venkat Krishnamurthy OmniSci プロダクト・バイスプレジデント
Kathryn Vandiver OmniSci プラットフォーム・コア・エンジニアリング シニア・ディレクター

OmniSci は、想像を上回る速さでデータの調査を可能にする最新のハードウェアとソフトウェアの開発を先導するというビジョンを掲げています。このきっかけの 1 つは、実験用プラットフォームである、オープン・データ・サイエンス・スタックです。OmniSci は、そのビジョンとインテルとの共同作業により、最新のハードウェアで従来のアナリティクスを利用して、データサイエンス分野の促進および統一に取り組んでいます。この記事では、データ・サイエンス・アプリケーションとデータ発見の関係について考え、このエコシステムにおける革新を促進するため OmniSci とインテルがどのように協力しているか説明します。

私たちは、人工知能 (AI) が人間の知覚と直観力を急速に拡大する世界に暮らしていることは明らかです。インテルのエンジニアも、この変化と AI の自然な進歩を、人々がデータを通じて世界を理解する方法の一部ととらえています。ビジュアル・アナリティクス・ツール (データの傾向を素早く理解できる人間の視覚に匹敵するものはないため) によるデータの「調査」から、データ・サイエンティストがモデルを構築する「実験」、ビジュアル・アナリティクス・ツールとマシンラーニング手法の両方を組み合わせてシームレスなワークフローで重要な情報を明らかにする「説明」へと、この理解がどのように進むか、「理解のループ」(**図 1**) を考えると分かりやすいでしょう。



1 **理解のループ**

現在のオープン・データ・サイエンス・スタックは、さまざまなエコシステムのオープンソース・イノベーションを利用して作成された、実験用プラットフォームです。

- Python* および PyData スタック (NumPy*/SciPy*)
- pandas*
- matplotlib
- dask
- Numba
- R 言語およびそのエコシステム
- Julia 言語
- Jupyter* および JupyterLab* プロジェクト

これらのコンポーネントは処理のコストを抑え、インタラクティブ・コンピューティング全般、特にデータに基づくストーリーテリングの推進を支援します。例として、**図 2** は、これらのツールを活用して Jupyter* Notebook の内部に表示された、史上初めて撮影に成功したブラックホールの写真を示しています。

2 **ブラックホールの写真 (詳細 (英語))**

OmniSci は、まだ MapD と呼ばれていた 2017 年にこの分野に参加し、その成長に貢献すべきであると認識しました。インメモリー・データベースとデータフレームは基幹技術となり、エンドツーエンド・アナリティクスの準備、前処理、抽出、変換、およびロードフェーズで任意のデータ型のエントリーポイントとしての機能を果たしています。

Python* によるエンドツーエンド・アナリティクス・パイプラインには、pandas* のような機能的なライブラリーがありますが、インテル® アーキテクチャー向けに最適化されていません。つまり、既存および最先端のインテル® ハードウェアの計算能力を利用できる、オープンソースのハイパフォーマンスなフレームワークのニーズがあります。そこで、インテルとの共同作業により、OmniSciDB をそのフレームワークにすることにしました。

pandas* は PyData エコシステムの重要なコンポーネントであり、その API 全体を複製することは逆効果であると判断しました。代わりに、インメモリー・データフレームの分析的な表現を評価する、その強力な表現方法に pandas* の価値があると考えました。OmniSciDB クエリー実行はすでにこのように動作していたため、OmniSciDB を活用できる使い慣れた「Python* 的な」API を見つける (または作成する) ことにしました。

このプロセスの次のステップとして、インテルとともに、オープン・データ・サイエンス・エコシステム内でスケーラブルでハイパフォーマンスなワークフロー実行を可能にする、OmniSci 向けの共通のデータフレームを配置することになりました。

pandas* の作者である Wes McKinney 氏は、pandas* の生産性を活用し、SQL などの言語を使用してより高いレベルのスケーラビリティーを実現することを目標として、**Ibis*** に取り組んでいます。Ibis* は優れたツールであり、特に OmniSci の速度およびスケールとの相性は抜群です。遅延式モデルにより、SQL (および pandas*) でアクセス可能な**バックエンド** (英語) で、複雑なアナリティクスを非常に簡単に実行できます。一貫性のあるデータフレームと Python* バインドを調整することにより、OmniSci と pandas* を使用したスケーラブルなワークフローの作成は、Ibis* を備えたハイパフォーマンスなバックエンドになります。

図 3 は、Ibis* で 14 億 5000 万行のテレマティックス・データセットの集計式をセットアップして評価し、pandas* データフレームを生成している様子を示しています。Ibis* は式を評価して SQL クエリーにコンパイルした後、必要な場合にのみ実行します。VPN 経由でデータセンターの OmniSci サーバーに対して実行した場合でも応答は非常に高速です。

```
[12]: ft.count().execute()

[12]: 1456336982

[20]: daily = ft.REPORT_TIME.truncate('D').name('daily')

[24]: expr = ft.group_by([daily])\
      .aggregate(ft.SPEED.mean())

[26]: expr.compile()

[26]: 'SELECT DATE_TRUNC(DAY, "REPORT_TIME") AS daily, avg("SPEED") AS "mean"\nFROM sfmta_avl_raw\nGROUP BY daily'

[29]: %time expr.execute()

      CPU times: user 288 ms, sys: 5.02 ms, total: 293 ms
      Wall time: 378 ms
[29]:
```

	daily	mean
0	2012-08-15	4.891463
1	2012-08-16	4.742025
2	2012-08-19	4.753195
3	2012-08-20	4.794622
4	2012-08-21	4.813361
...	...	...
1507	2016-11-23	3.622486
1508	2016-11-24	2.158154
1509	2016-12-04	3.523434
1510	2016-12-05	3.578887
1511	2016-12-06	1.785714

1512 rows × 2 columns

3 **Ibis* で 14 億 5000 万行のデータセットを処理**

この新しい Python* API のおかげで、pandas* を使用して Apache Arrow* ベースのデータ・フレーム・メモリーに Ibis* を直接出力することができます。現在は、インテルのデータ・サイエンティストおよびマシンラーニングのエキスパートと協力して、内部結果セット形式を用いた**インテル® Xeon® スケーラブル・プロセッサー**上でのさらなる高速化によりインターフェイスのオーバーヘッドを削減する作業を行っています。

(これはかなりの優れものです。この Ibis* および pandas* でサポートされるバックエンドでは、1 つの API を使用して 1 つの JupyterLab* Notebook の内部でさまざまなデータソースを分析できます。いずれかのバックエンドへの接続を作成するだけで、バックエンドに対して Ibis* の式を実行できます。リモート・バックエンドの結果はデフォルトで pandas* に返されるため、その **pandas* データフレーム** (英語) に Ibis* 接続をラップできます。可能性は無限です。)

行ったり来たり

最後に、**pymapd** (英語) にも取り組むことで、基盤となるプロジェクト (特に Apache Arrow*) の変化のペースにようやく追い付いてきました。データ・サイエンティストは、これらのツールをすべて使用して、pymapd (および Ibis*) の **`load_table`** API を利用して OmniSci にロード可能なデータフレームを作成できるようになりました。また、OmniSci 4.7 では OmniSci Immerse の Visual Data Fusion (VDF) 機能が追加されました。ユーザーは、この機能を利用して、複数のテーブル / ソースのグラフ (現在は組み合わせおよびマルチレイヤー geo グラフ) をセットアップできます。

すべてをまとめる

重要なポイントは、作業の繰り返しを苦にしないことです。各自が独自の機能を開発するのではなく、オープンソース・プロジェクトのコミュニティーで議論した機能をそれぞれ開発しました。OmniSci が Ibis* と pandas* で行った作業は、コミュニティーに関わるすべての人が利用できます。OmniSciDB は 2 年以上前からオープンソースです。ユーザーがさまざまな方法でデータ・サイエンス・ワークフローに OmniSci を簡単に追加できるように、パッケージやインストールも考慮されています。

パッケージはすべて Docker* で行うようにしました。JupyterLab* イメージには、必要なツールがすべて含まれています。Mac* または Linux* ラップトップで、OmniSciDB を含むセットアップをダウンロードして試すことができます。Anaconda* の Python* パッケージ管理を使用しているため、**`conda install -c conda-forge omniscidb-cpu`** コマンドで **`conda-forge`** から OmniSciDB をインストールした後、**`conda install -c conda-forge omnisci-pytools`** コマンドで OmniSci 向けの PyData ツールをインストールできます。あるいは、**OmniSci のダウンロード・ページ** (英語) からビルド済みバージョンの OmniSci を入手することもできます。

過去を振り返り、将来を見据える

ご協力いただいたすべての方々に感謝します。これらの新しい機能はすべて、インテルのエンジニアを含む、オープンソースの協力者の作業と意見がなければ、達成できないものでした。結局のところ、データサイエンスの最先端の手法 (システムを統一すること) はユーザーに役立つものであり、不利益をもたらすものではないと考えています。利用者には、ツールのアセンブリー全体は見せないようにし、情報とその説明は明確にしたほうが良いでしょう。OmniSci は、そのビジョンにより、最新のハードウェアで従来のアナリティクスを利用してデータサイエンス分野の促進および統一に取り組んでいます。

NEWS HIGHLIGHTS

インテルが oneAPI をゲームの世界に — グラフィックス向けレンダリング・ツールキットをデモ

ARNE VERHEYDE TOM'S HARDWARE

バーチャル Game Developers Conference (GDC) 2020 の一部として、インテルはゲームで利用可能なインテル® oneAPI レンダリング・ツールキットについて詳しく説明したプレゼンテーションをオンラインで公開しました。ツールキットには、インテル® Embree、インテル® OSPRay、インテル® Open VKL、インテル® OpenSWR、およびインテル® Open Image Denoise などのライブラリーが含まれます。インテルは、いくつかのライブラリーは GPU でもサポートされる予定であると発表しました。

この記事の続きはこちら (英語) でご覧になれます。 >

自動微分を使用した並列計算の新しいアプローチ

最新のマルチコアシステムで最高のパフォーマンスを達成

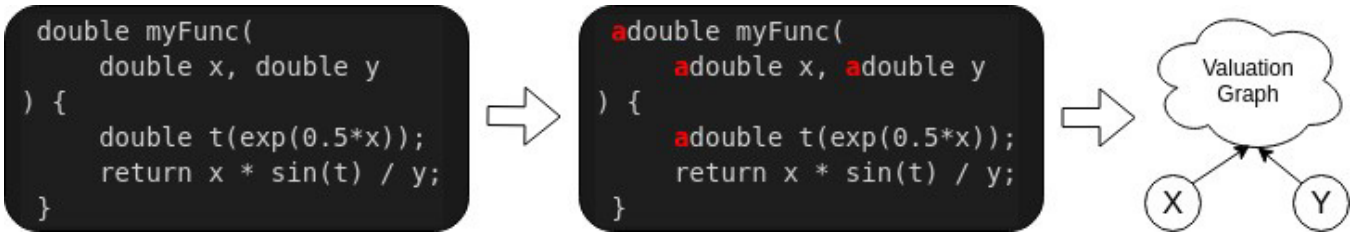
Dmitri Goloubentsev Matlogica 自動随伴微分担当
Evgeny Lakshtanov ポルトガル アヴェイロ大学数学部主席研究員および Matlogica LTD

ハイパフォーマンス・コンピューティングに注目する皆さんが、最初に思い浮かべるのはハイレベルなオブジェクト指向言語ではないでしょう。オブジェクトの抽象化は実行時のペナルティーを伴い、コンパイラーがベクトル化することは困難です。コードをマルチスレッド実行に適応させることは大きな課題であり、多くの場合、生成されるコードの保守に頭を悩ませることになります。

コードのパフォーマンスが重要な部分が局所化されていて、フラット化および安全に並列化できる場合は幸運と言えます。しかしながら、パフォーマンスが重要な問題の多くは、オブジェクト指向プログラミングの抽象化から恩恵を受けることができます。ここでは、SIMD (Single Instruction Multiple Data)、NUMA (Non-Uniform Memory Access) マルチコアシステムで最高のパフォーマンスを実現できるプログラミング・モデルを提案します。

評価グラフ抽出のための演算子のオーバーロード

同じ関数 **F(1)** をデータセット **X[i]** で実行する必要がある問題を考えてみます。例えば、**X[i]** がランダムサンプルで **F(.)** が価格計算関数のモンテカルロ・シミュレーションを見てみましょう (**図 1**)。**演算子のオーバーロード・パターン (利用者定義演算子)** を使用して、**F(.)** で実行されるすべてのプリミティブ演算子を抽出します。



1 **演算子のオーバーロード・パターンの例**

このパターンは**自動随伴微分 (AAD)** ライブラリーでは一般的であり、従来の AAD ライブラリーとは異なり評価グラフを表すデータ構造は構築しません。代わりに、バイナリー・マシンコード命令をコンパイルして、グラフで定義されている評価を複製します。これは、ジャストインタイム (JIT) コンパイルと見なすことができます。ただし、ソースコードは直接操作しません。代わりに、ユーザーのアルゴリズムにより生成された評価グラフをコンパイルします。大規模なデータポイントのセットに **F(.)** を適用する必要があるため、このコードをコンパイルしてすべてのスカラー演算を完全な SIMD ベクトル演算に拡張し、4 つ (インテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2)) または 8 つ (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)) のデータサンプルを並列に処理します。

例で学ぶ

さまざまな抽象的なビジネス・オブジェクトを使用する単純なオプション価格設定フレームワークを見てみましょう。この例では、資産価値をランダムプロセスとしてシミュレートします。

```
template<class vtype>
vtype simulateAssetOneStep(
   const vtype current_value
   , Time current_t
   , Time next_t
   , const BankRate<vtype>& rate
   , const AssetVolatility<vtype>& vol_obj
   , const vtype& random_sample
) {
   double dt = (next_t-current_t);
   vtype vol=vol_obj(current_value, current_t);
   vtype next_value = current_value * (
       1 + (-vol*vol / 2+ rate(current_t))*dt
       +  vol * std::sqrt(dt) * random_sample
   );
   return next_value;
}
```

BankRate クラスと **AssetVolatility** クラスは異なる方法の計算モデル・パラメーターを定義でき、派生クラスの深くで実装できます。この関数はネイティブの **double** 型で使用できます。タイムポイントに沿って適用すると、**t[i]** を使用してオプション満期時の資産価値をシミュレートできます。

```
template<class vtype>
vtype onePathPricing (
   vtype asset
   , double strike
   , const std::vector<Time>& t
   , const BankRate<vtype>& rate_obj
   , const AssetVolatility<vtype>& vol_obj
   , const std::vector<vtype>& random_samples
) {
   for (int t_i = 0; t_i < t.size()-1; ++t_i) {
       asset = simulateAssetOneStep(asset, t[t_i], t[t_i+1], rate_obj,
vol_obj, random_samples[t_i]);
   }
   return std::max(asset - strike, 0.);
}
```

ただし、コンパイラーがコードを効率良くベクトル化できず、ビジネス・オブジェクトが仮想関数呼び出しを含むこともあるため、パフォーマンスは低くなります。AAD ランタイム・コンパイラーを使用すると、関数を実行し、資産変化のランダムパスを 1 つ記録して、満期時にオプション固有の値を計算できます。

```
typedef __m256d mmType;  // __mm256d and __mm512d are supported

   int AVXsize = sizeof(mmType) / sizeof(double);

   aadc::AADCFunctions<mmType> aad_funcs;

   std::vector<idouble> random_samples(num_time_steps, 0.);
   idouble rate(0.03), vol(0.15), asset(100.0);
   aadc::VectorArg random_arg;

   aad_funcs.startRecording();
       // Mark vector of random variables as input only
       markVectorAsInput(random_arg, random_samples, false);

       // Mark rate, initial asset value and volatility as inputs
       aadc::AADCArgument rate_arg(rate.markAsInput());
       aadc::AADCArgument asset_arg(asset.markAsInput());
       aadc::AADCArgument vol_arg(vol.markAsInput());

       BankRate<idouble> rate_obj(rate);
       AssetVolatility<idouble> vol_obj(vol);

       idouble payoff= onePathPricing(asset, strike, t, rate_obj, vol_obj, random_samples);
       aadc::AADCResult payoff_arg(payoff.markAsOutput());
   aad_funcs.stopRecording();
```

この段階では、**func** オブジェクトには、任意の **random_samples** ベクトルを入力として、最終的なペイオフ出力値を生成するため評価を複製する、コンパイル済みのベクトル化されたマシンコードが含まれています。関数オブジェクトは記録後も一定で、実行にはメモリー・コンテキストが必要です。

```
 // allocate memory needed for vectorized execution
shared_ptr<WorkSpace<mmType> > ws = func.createWorkSpace();

mmType mm_total_price(mmSetConst<mmType>(0.0));

// initialize function inputs
ws->val(rate_arg) = mmSetConst<mmType>(init_rate);
ws->val(asset_arg) = mmSetConst<mmType>(init_asset);
ws->val(vol_arg) = mmSetConst<mmType>(init_vol);

// run Monte Carlo loop
for (int mc_i = 0; mc_i < (num_mc_paths / AVXsize); ++mc_i) {
    vector<mmType> avx_random_samples(generateRandomAvxVector());
    // Set function arguments
    ws->setVector(random_samples_arg, avx_random_samples);

    // call the recorded function
    func.forward(ws);
    // get result
   mm_total_price=mmAdd(ws->val(payoff_arg), mm_total_price);
}
// calc avx vector element wise sum
cout << mmSum(mm_total_price) << endl;
```

簡単にマルチスレッド化

効率的で安全なマルチスレッド・コードを作成することは困難です。記録は 1 つの入力サンプルに対してのみ行われ、制御された、安定したシングルスレッド環境で実行されます。ただし、その結果記録される関数はスレッドセーフであり、スレッドごとに割り当てられた個別のワークスペース・メモリーのみ必要とします。これは、マルチスレッド・セーフでないコードをマルチコアシステムで安全に実行できるコードに変換できる、非常に魅力的なプロパティーです。最適な NUMA メモリーの割り当ても容易になります (マルチスレッドの例の完全なコードは、**こちら** (英語) で参照できます)。

自動微分

この手法は、関数を高速化するだけでなく、すべての出力に関連する入力の導関数を計算する随伴関数を作成することもできます。これは、ディープ・ニューラル・ネットワーク (DNN) のトレーニングで使用される誤差逆伝播法 (バックプロパゲーション) アルゴリズムに似ています。DNN のトレーニング・ライブラリーとは異なり、このアプローチはほぼすべての C++ コードで適切に動作します。随伴関数を記録するには、微分に必要な入力変数をマークするだけです。

```
// compute derivative w.r.t. initial value
AADC::Argument asset_arg(asset.markAsDiff());
```

最後に、随伴関数を実行するには、出力の勾配値を初期化して、関数オブジェクトで **`reverse()`** メソッドを呼び出します。

```
ws->diff(payoff_arg) = mmSetConst<mmType>(1.0);

func.reverse(ws);

cout << "Derivative of dPrice/dSpot = " << ws->diff(asset_arg)[0] << endl;
```

最高のパフォーマンスを達成する

ハードウェアは、多くのコア、広いベクトルレジスター、アクセラレーターで並列処理を増やすために進化しています。オブジェクト指向のプログラマーにとって、シングルスレッドのコードを OpenMP* や CUDA* のような既存の並列手法に適応させることは困難です。Matlogica の AADC ツールを使用すると、プログラマーは、オブジェクト指向のシングルスレッドのスカラーコードを、インテル® AVX2/ インテル® AVX-512 に対応したベクトル化、マルチスレッド化されたスレッドセーフなラムダ関数に変換することができます。AADC ツールは、必要なすべての導関数で同じインターフェイスを使用して、随伴法を計算するラムダ関数を生成することもできます。AADC ツールの詳細およびデモバージョンについては、**Matlogica 社のウェブサイト** (英語) を参照してください。

謝辞

Evgeny Lakshtanov は、CIDMA (the Center for Research and Development in Mathematics and Applications) およびポルトガル科学技術財団 (FCT、Fundação para a Ciência e a Tecnologia) によるプロジェクト UIDP/04106/2020 でポルトガルの基金による部分的な支援を受けています。

マルチコア向け並列プログラミングの 8 つのルール

並列化の課題の解決とマルチコアの潜在能力を引き出すのに役立つ一貫したルール

James Reinders The Parallel Universe 創刊編集長および名誉編集長

[編集者注 : この記事は The Parallel Universe 1 号 (2009 年 4 月) に掲載されたものです。前編集長 James Reinders のアドバイスは 11 年経過した現在でも適切なものです。その先見性に敬意を表して、この号で再掲することにしました。]

マルチコア・プロセッサー向けのプログラミングは新しい挑戦です。ここでは、マルチコア・プログラミングで成功するための 8 つのルールを紹介します。

ルール 1: Think Parallel (並列化を考える)

並列処理を意識しながら、すべての問題にアプローチします。並列化の場所を理解し、それを適用する考えを整理します。その他の設計や実装の決定を下す前に、最も良い並列アプローチを決定します。「Think Parallel」並列処理の思考を身につけます。

ルール 2: 抽象化を使用してプログラミングする

並列化を表現するコードの記述に重点を置き、スレッドやプロセッサー・コアを管理するコードの記述を避けます。ライブラリー、OpenMP*、インテル® スレッディング・ビルディング・ブロックなどは、すべて抽象化を使用している例です。ネイティブスレッド (pthreads*、Windows* スレッド、Boost スレッドなど) は使用しないようにしてください。ネイティブ・スレッド・ライブラリーは、並列化のアセンブリー言語です。これらは最も柔軟性がありますが、記述、デバッグ、保守に多大な時間が必要です。スレッド管理やコア管理を行うプログラミングではなく、問題に対応するプログラミングを行えるよう、コードは十分にレベルの高いものである必要があります。

ルール 3: スレッド (core: コア) ではなく、タスク (chore: 仕事) をプログラミングする

スレッドやプロセッサー・コアへのタスクのマッピングは、プログラムの中で明確に分離された処理にしておきます。使用する抽象化は、プログラマーの代わりにスレッド / コア管理を行ってくれるものであることが望ましいでしょう。プログラムには多くのタスク、またはプロセッサー・コア間で自動で処理されるタスク (OpenMP* ループなど) を作成してください。タスクを作成することで、オーバーサブスクリプションを心配することなく、自由に可能な限り多くの処理を作成できます。

ルール 4: 並列処理をオフにするオプションを付けて設計する

デバッグ作業を簡単にするには、並列処理を行わずに実行できるプログラムを作成します。デバッグの際に、最初は並列処理をオンにして実行し、その後オフにすることで、両方の実行で失敗するかどうかを調査できます。プログラムが並列実行されていない場合、一般的な問題は診断がしやすく、従来のツールでもサポートされているため、デバッグが簡単です。並列実行時のみ不具合が生じることが分かれば、追跡中の不具合がどのような種類のものかを特定する手がかりになります。このルールを無視して、シングルスレッドで実行できないプログラムを作成した場合、デバッグ作業に過度の時間を費やすことになります。シングルスレッドの実行はデバッグ用にのみ必要なため、効率的である必要はありません。「Producer-Consumer (生産者 - 消費者)」モデルなど、並行処理が正しく動作しなければならない並列プログラムの作成を避ければ良いだけです。

ルール 5: ロックの使用を避ける

ロックは極力使用しないでください。ロックはプログラムの実行速度を低下させ、スケーラビリティーを減少させ、並列プログラム中の多くの不具合の原因になります。問題の解決には、暗黙的な同期を使用します。明示的な同期が必要な場合は、アトミック操作を使用します。ロックは、最終手段としてのみ使用します。ロックの必要がないプログラムを設計してください。

ルール 6: 並列化に役立つツールとライブラリーを使用する

古いツールで「頑張らない」ようにします。並列化をどのように提示し、そして並列化とどのようにかかわるか、という観点から、ツールに対して批判的になってください。ほとんどのツールは、並列化の準備ができていません。スレッドセーフなライブラリーを探してください。並列化を活用するよう設計されたものが理想的です。

ルール 7: スケーラブル・メモリー・アロケーターを使用する

スレッド化プログラムでは、スケーラブル・メモリー・アロケーターを使用する必要があります。多くのソリューションがありますが、私は、それらすべてが **`malloc()`** よりも優れていると考えます。スケーラブル・メモリー・アロケーターを使用することで、グローバル・ボトルネックが排除され、スレッド間でメモリーを再利用してキャッシュを有効利用し、適切なパーティショニングによってキャッシュラインの共有が回避され、アプリケーションの速度が向上します。

ルール 8: ワークロードに応じてスケーリングするように設計する

年々、プログラムが処理すべき作業は増加していきます。そのための準備が必要です。スケーリングを念頭において設計しておくことで、プロセッサー・コアが増えてもより多くの作業を処理することができます。毎年、インテルのコンピューターの処理能力は向上しています。将来、増加していくワークロードを処理する場合に有利な設計でなければなりません。

マルチコア・プロセッサーから最大限の性能を引き出す

この 8 つのルールでは、スレッド化が至るところで暗黙的に言及されています。ルール 7 のみが、明確にスレッド化について言及したものです。スレッド化が、マルチコアの価値を引き出す唯一の方法ではありません。マルチプログラムやマルチプロセスの実行は、特にサーバー・アプリケーションではよく使用されています。

ここで紹介したルールは、マルチコア・プロセッサーから最大限の性能を引き出すのに役立ちます。**プロセッサー・コアは増え、コア自体の多様性も増しているため、今後 10 年間に、この 8 つのルールのいくつかはその重要性が一層高くなることでしょう。例えば、ヘテロジニアス・プロセッサーや NUMA の到来は、ルール 3 の重要性をより高いものにしています。**[編集者注 : 太字は編集者が追加したものです。James は The Parallel Universe 1 号ですでにヘテロジニアス並列処理について言及していました。]

ここで紹介した 8 つのルールをよく理解し、検討してください。

書評 :『The OpenMP Common Core』

Timothy G. Mattson、Yun (Helen) He、Alice Konigs 共著

Making OpenMP Simple Again

Ruud van der Pas Oracle Corporation シニア主任ソフトウェア・エンジニア

OpenMP* は、共有メモリー並列コンピューティング向けのアプリケーション・プログラミング・インターフェイス (API) を提供します。コンパイラー・ディレクティブ、ライブラリー・ルーチン、環境変数で構成されており、これらはすべて、開発者が並列実行を定義および制御するために使用できます。OpenMP* 仕様は、ハードウェア・アーキテクチャーおよびプログラミング言語のトレンドに対応するため、1997 年から進化を続けています。これは前向きな動きですが、OpenMP* の利用に興味を持っている開発者にとって、現在の機能は多種多様すぎます。Timothy G. Mattson、Yun (Helen) He、Alice Konigs 共著『The OpenMP Common Core』では、「必修科目」のアイデアを定義することにより、この問題に対処できるようにしています。

著者たちが定義した必修科目は、OpenMP* API のコンパクトなサブセットで、新しいユーザーが始めに知るべき情報として理想的です。本書で紹介されている必修科目の機能は OpenMP* を使用して多くのアプリケーションを並列化するには十分ですが、追加の機能が必要なユーザーのために、ほかの資料についての情報も提供されています。本書では、読者に C、C++、または Fortran の基本的なプログラミング・スキルがあることを前提としていますが、並列コンピューティングの知識は必要ありません。

本書は 3 部で構成されています。

1. Setting the Stage (はじめに)
2. The OpenMP Common Core (OpenMP* 必修科目)
3. Beyond the Common Core (必修科目が終わったら)

Setting the Stage (はじめに)

第 1 部では、前提条件について説明し、読者が以降の内容を理解するために必要な、並列コンピューティングに関連する概念を紹介および説明します。現代のさまざまな並列アーキテクチャーの概要も含まれています。OpenMP* の歴史も示されており、現在の OpenMP* がどのようなものか、なぜ必修科目が必要なのかを理解するのに役立ちます。この第 1 部を含めたことにより、並列コンピューティングを初めて行う (特に OpenMP* を初めて使用する) 読者でもすんなりと読むことができるようになっています。ほかの文献を参照する必要はなく、本書を読むだけで十分です。

The OpenMP Common Core (OpenMP* 必修科目)

第 2 部は 120 ページ以上からなり、本書の大部分を占めます。著者たちが OpenMP* 必修科目について議論して定義します。この OpenMP* API のサブセットを定義する概念と構成は非常に分かりやすく説明されています。アプローチは非常に実用的で、例を使用して機能を説明した後、機能を適用する方法を示しています。π の値を近似する数値積分アルゴリズムなど、いくつかの例は、省略することなく示されています。これらの例は、小さくても完全に機能し、シーケンシャル・プログラムを正しい OpenMP* プログラムに変換する方法を説明するのに非常に役立っています。多くの問題点は細部に含まれることから、著者たちは本書全体を通して潜在的な落とし穴を取り上げています。

OpenMP* データモデルは、初心者にとって習得が難しいものの 1 つです。これは主に、シーケンシャル・アプリケーションではデータモデルについて考える必要がないためですが、共有メモリー・プログラミングでは不可欠です。このトピックの内容は素晴らしいとしか言えません。このトピックを読めば、シングルスレッド・プログラムからマルチスレッド・プログラムに移行することの不安と混乱を取り除くことができるでしょう。経験豊富な OpenMP* 開発者にとっても、このセクションは役立つはずです。

本書がほかの書籍と異なる点はもう 1 つあります。多くの OpenMP* 書籍は、並列ループまでしか説明していません。並列アルゴリズムのいくつかのクラスでは、異なる形式の (非同期で動的な) 並列処理が必要です。OpenMP* では、このサポートはタスクの概念を通じて提供されます。本書では、複数のコード例を使用して、OpenMP* タスクを詳細に説明しています。驚いたことに、タスクを含むデータ環境についても非常に分かりやすく説明されています。タスクは簡単に理解できるトピックではありませんが、著者たちは非常にうまく説明しています。

すべての OpenMP* 開発者は、メモリーモデルを正しく理解している必要があります。これは OpenMP* に固有のもので、仕様のかなり複雑な部分です。このトピックを説明している章は、本書の中でも特に優れた部分と言えます。また、共有メモリー並列プログラミングを初めて使用するユーザーにとってもう 1 つの難解なトピックである、メモリーの一貫性についても分かりやすく紹介しています。これらのトピックについて本書より優れた説明を読んだことはありません。

第 2 部は、OpenMP* 必修科目を要約しておしまいです。初めて読んだときは内容が少し冗長ではないかと疑問に思いましたが、本書を通して読み終えるとその疑問は解消しました。読者が以前の章を読み終えていると仮定すると、この第 2 部は優れたリファレンスになるでしょう。リファレンスとして利用することにより、以前の章で取り上げた詳細を調べたり、詳細を見つけたりすることが簡単になります。この第 2 部は、必要な情報がすべて含まれ、情報が結び付くところです。また、著者たちは、ここで特定の機能の使用について微妙な違いを紹介しています。

Beyond the Common Core (必修科目が終わったら)

最後の第 3 部では、OpenMP* 必修科目以外に開発者が必要になる可能性のある機能 (追加の句、ランタイム関数、アトミック操作、ロックなど) について説明します。設計上、収録内容は概要ですが、詳細な情報へのポインターが含まれています。いくつかの機能は必修科目に含まれていないことを不思議に思われるかもしれません。その理由は、すべての開発者がそれらの機能を必要としているわけではなく、アルゴリズムの並列化に固有のものであるためです。

第 3 部の比較的大きなセクションでは、不均等メモリーアクセス (NUMA) の重要なトピックに特化して取り上げています。NUMA はパフォーマンス機能であり、正しい OpenMP* プログラムを記述することとは無関係ですが、本書で取り上げることに異論はありません。NUMA は本書の前半で言及されていますが、表面に多少触れただけです。ここでは、現代のメモリーシステムがどのようなものか、アプリケーションのパフォーマンスにどのように影響するか、詳細に説明しています。OpenMP* は、2013 年にリリースされたバージョン 4.0 仕様から NUMA をサポートしています。ユーザーがデータ配置とスレッド・アフィニティーを制御するために使用できる機能は、いくつかの例を使用して説明されています。

このセクションの最後の部分では、便利なポインターとその他の情報を提供しています。例えば、OpenMP* 仕様を読んで理解する方法を説明しています。仕様は無料で入手できますが、現在の仕様は 600 ページを超えており、かなりの量になります。また、エンドユーザーだけではなくコンパイラー開発者も対象としています。そのため、コンパイラーのエキスパートでなければ、理解するのが難しいかもしれません。ただし、仕様を確認する必要がある場合もあります。このセクションは、読者が自分の取り組み方を見つけるのに役立ちます。

本書は、初心者はもちろん、すべての OpenMP* 開発者にとって非常に役立つものとなっています。経験豊富な開発者は、手元に置いておくに足る十分な情報が全体に散りばめられていることに気付くでしょう。

Software

THE PARALLEL UNIVERSE

JPN/2005/PDF/XL/SPI/ND