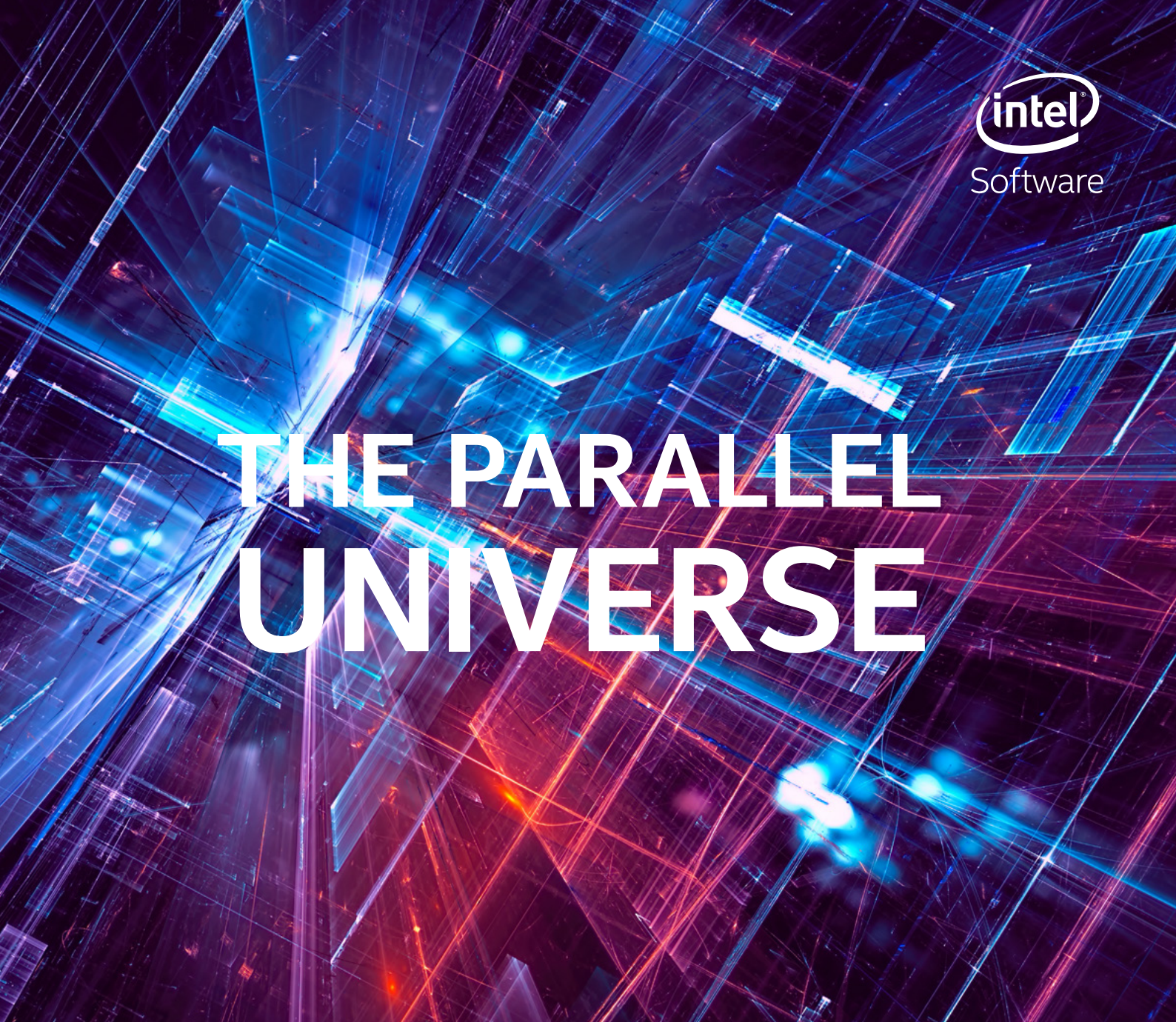


インテル® プロセッサー・グラフィックス・アーキテクチャー向けの DPC++

インテル® コンパイラーの最適化レポートを最大限に活用する

OpenMP* ベクトル化サポートの謎に迫る

Issue
41
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。

本号は自宅のオフィスからお届けします

何が重要かについて考える

前号の「編集者からのメッセージ」は、テキサス州オースティンにあるインテルのオフィスで執筆しました。あれから実に多くのことが起こりました。新型コロナウイルス (COVID-19) の急速な感染拡大により、日常が大きく様変わりし、何が重要なのかについて考えさせられました。そして、病気と闘うため、ウイルス学と計算科学の頭脳が集結されました。インテルは、パンデミックとの闘いを支援するため、**COVID-19 ハイパフォーマンス・コンピューティング・コンソーシアム** (英語) に参加しています。このコンソーシアムは、連邦政府、産業界、学術界のリーダーたちが無償で計算時間やリソースを提供する官民一体のユニークな取り組みです。私は、この取り組みに協力できることを誇りに思います。

COVID-19 の話はこれぐらいにしておいて、コンピューティングについてお話ししましょう。現代のベクトル CPU は実質的にアクセラレーターです (「**oneAPI: Why Should You Care?**」(英語) の Erik Lindahl 教授のコメントをご覧ください)。ベクトル処理の活用については本誌でもたびたび取り上げてきましたが、多くのインテル® CPU には汎用計算処理に使用可能なグラフィックス・プロセッサーが統合されています。この低消費電力アクセラレーターへの計算のオフロードについて、本号の注目記事「**インテル® プロセッサー・グラフィックス・アーキテクチャー向けの DPC++**」は、ステップバイステップのケーススタディーを提供します。

OpenMP* についての 2 つの記事、「**ヘテロジニアス・アーキテクチャーのプログラミングに OpenMP* オフロードを使用する**」と「**OpenMP* SIMD サポートの謎に迫る**」はそれぞれ、アクセラレーターへのオフロードとベクトル化ディレクティブの利用方法を示します。ベクトル化は通常、パフォーマンスを大幅に向上させる安価な方法であり、優れたコンパイラーは手間のかかる作業の多くを行ってくれます。「**インテル® コンパイラーの最適化レポートを最大限に活用する**」は、コンパイラーが計算集約型ループを効率良く最適化できるように支援するのに役立ちます。

次に、分散メモリー型並列処理と MPI パフォーマンス・チューニングに注目します。計算と通信をオーバーラップさせてレイテンシーを隠蔽することは、長い間 MPI プログラミングの常識となっていましたが、20 年以上前に「**Where's the Overlap? An Analysis of Popular MPI Implementations**」(英語) が出版されて以来、パフォーマンスの利点を実証することは困難でした。「**MPI-3 の非ブロッキング集合操作による通信レイテンシーの隠蔽**」(Parallel Universe 33 号) と「**インテル® MPI ライブラリーの MPI-3 非ブロッキング I/O 集合操作**」(Parallel Universe 35 号) が出版されたことで疑念が和らぎました。「**HPC アプリケーションの計算と通信のオーバーラップ**」は、通信レイテンシーを隠蔽することによるパフォーマンスの利点について、さらに詳しく実証しています。「**インテル® MPI ライブラリーのチューニングを使用した HPC クラスターの効率化**」は、クラスターの運用効率を向上する方法を紹介します。

最後に、**OmniSci** (英語) 社のプロダクト・マネージメント・バイスプレジデントである Venkat Krishnamurthy 氏は、「**データ・サイエンス・アプリケーションを次のレベルへ**」で大規模なデータ・アナリティクスと、データサイエンスと従来のアナリティクスの融合について述べています。

以上が本号で取り上げている記事です。次号までにパンデミックが終息に向かうことを願っています。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 7 月

インテル® プロセッサー・グラフィックス・アーキテクチャー向けの DPC++

計算集約型コードをインテル® GPU へオフロードする方法

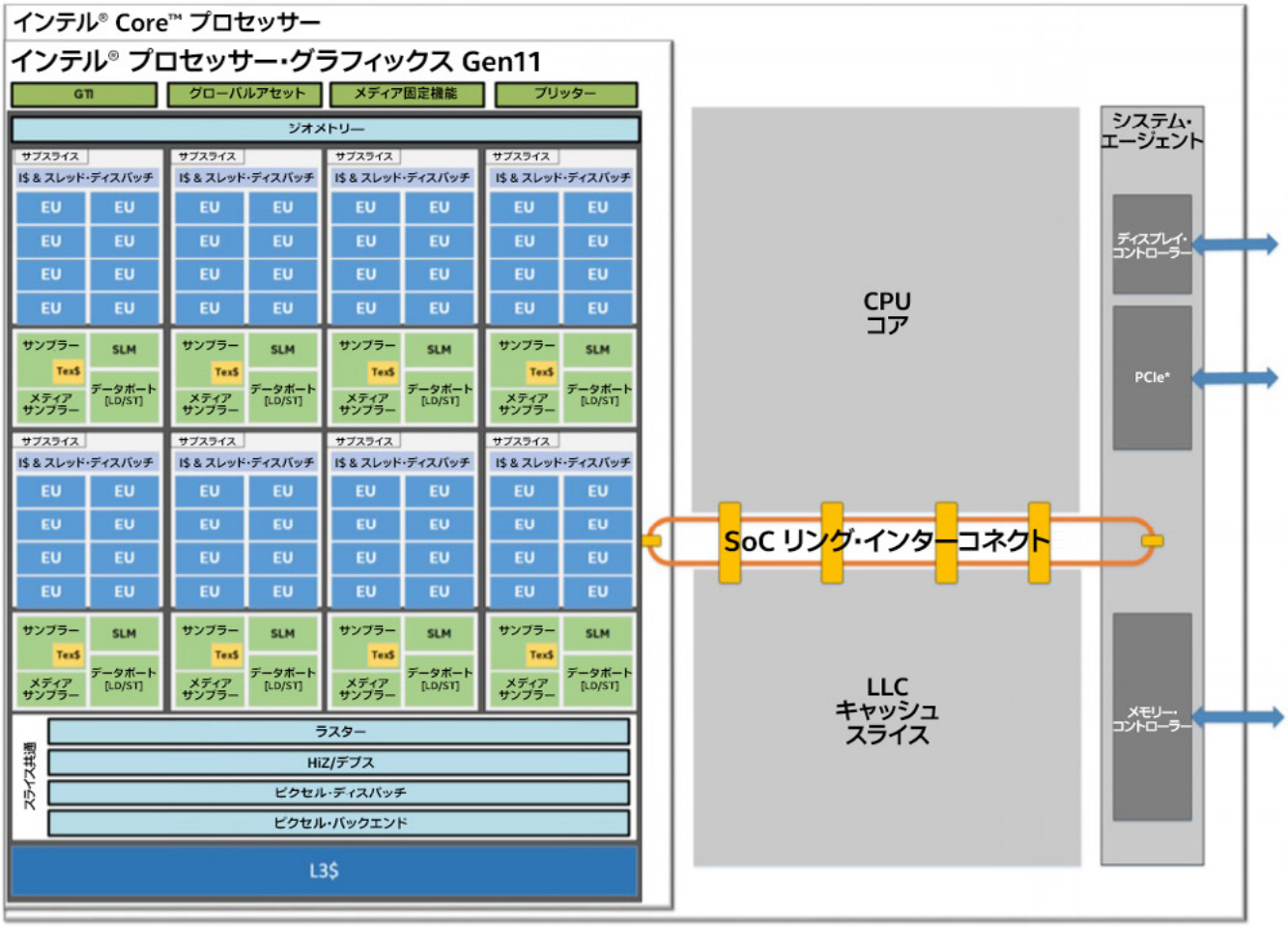
Rama Malladi インテル コーポレーション グラフィックス・パフォーマンス・モデリング・エンジニア

インテル® プロセッサー・グラフィックス・アーキテクチャーは、インテルのシステムオンチップ (SoC) 製品にグラフィックス、計算、メディア、およびディスプレイ機能を提供するインテル® テクノロジーです。インテル® プロセッサー・グラフィックス・アーキテクチャーは、略称「Gen」(世代を意味する Generation の省略形) として知られています。「Gen」の後にはアーキテクチャーの各リリースに対応するバージョン番号が続きます。例えば、インテル® プロセッサー・グラフィックス・アーキテクチャーの最新リリースは Gen11 です。長年にわたって進化したインテル® プロセッサー・グラフィックス・アーキテクチャーは、優れたグラフィックス (3D レンダリングとメディア・パフォーマンス) と最大 1 TFLOPS (1 秒間に 1 兆回の浮動小数点操作) のパフォーマンスを達成する汎用計算処理能力を提供します。

この記事では、インテル® プロセッサー・グラフィックス **Gen9** (英語) および **Gen11** (英語) アーキテクチャーの汎用計算処理と、**インテル® oneAPI ベース・ツールキット** (英語) の**データ並列 C++ (DPC++)** (英語) を使用してこれらをプログラムする方法を調査します。具体的には、DPC++ を使用した 2 つの Gen アーキテクチャーのプログラミングとパフォーマンスを示すケーススタディーを見ていきます。

Gen 別のインテル® プロセッサー・グラフィックス・アーキテクチャーの概要

インテル® プロセッサー・グラフィックスは、インテル® CPU と統合された、電力効率に優れたハイパフォーマンスなグラフィックスとメディア・アクセラレーターです。統合 GPU は、最終レベルキャッシュ (LLC) を CPU と共有することで、低レイテンシー、高帯域幅で細粒度のコヒーレントなデータ共有を可能にします。**図 1** は、Gen11 グラフィックスを搭載した SoC です。オンダイ統合により、ディスクリート・グラフィックス・カードよりもはるかに低い消費電力を実現しています。

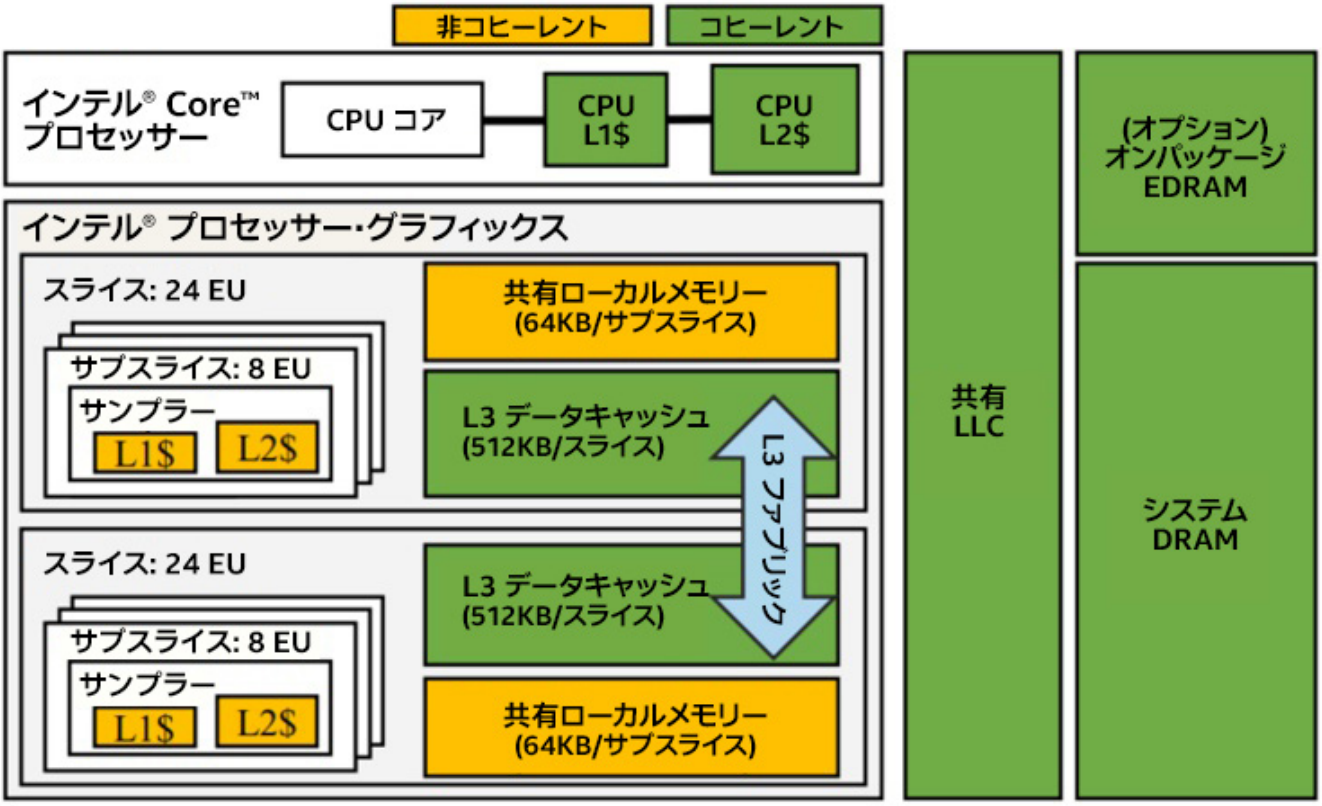


1 **インテル® プロセッサー・グラフィックス Gen11 SoC (CPU SoC の一部)**

図 2 は、Gen9 GPU アーキテクチャーの簡単なブロック図です。GPU には多数の実行ユニット (EU) があり、それぞれ SIMD (Single Instruction, Multiple Data) 計算を実行できます。8 つの EU が 1 つのサブスライスとなり、各サブスライスには以下があります。

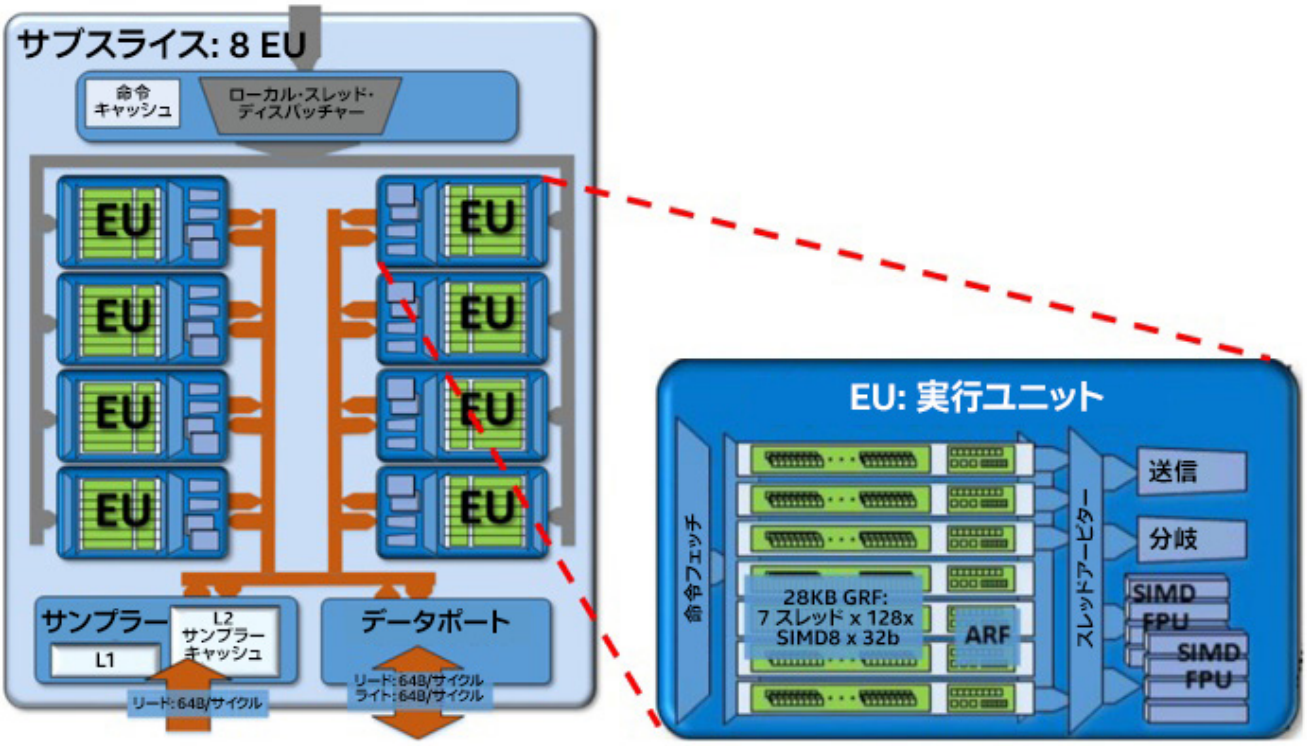
- 命令キャッシュ
- L1/L2 サンプラーキャッシュ
- メモリーロード / ストア・ユニット・ポート

サブスライスの集合がスライスを形成し、共有 L3 キャッシュ (CPU とコヒーレント) とバンク共有ローカルメモリー (SLM) とで構成されます。インテル® インテグレーテッド GPU には、1 つ以上のスライスがあります。そのような構成では、L3 はインターコネクト・ファブリックを介して複数のスライスに接続されます。



2 **Gen9 インテル® GPU アーキテクチャー**

図 3 は、Gen9 アーキテクチャーの EU の詳細です。EU ごとに最大 7 スレッドをサポートしており、各スレッドには 128 個の SIMD8 32 ビット・レジスターがあります。EU は、**サイクルごとに最大 4 つの命令** (英語) を発行できます (インテル® GPU のアーキテクチャーの詳細とベンチマークについては、**こちら** (英語) を参照)。例えば、ハードウェアのピーク理論 GFLOPS は、**(EU の数) * (EU ごとの SIMD ユニットの数) * (SIMD ユニットごとの 1 サイクルあたりの FLOPS 数) * (周波数 GHz)** として計算できます。



3 **サブスライスと EU アーキテクチャーの詳細**

GPU などのデバイスをプログラミングする場合、最高のパフォーマンスを引き出すには、利用可能なハードウェア機能に対応した言語構造が必要です。いくつかの API がありますが、ここでは oneAPI に注目します。

oneAPI と DPC++

oneAPI は、さまざまなアクセラレーターや複数の世代のハードウェアにわたって移植性とパフォーマンスを提供する、オープンで無料の標準ベースのプログラミング・モデルです。oneAPI に含まれる DPC++ は、さまざまなハードウェア・ターゲットにわたってコードの再利用を可能にするコア・プログラミング言語です。詳細は、以前の記事「**インテル® oneAPI を使用したヘテロジニアス・プログラミング**」(Parallel Universe 39 号) を参照してください。DPC++ には以下が含まれます。

- ホストデバイスのメモリー管理を容易にする**統合共有メモリー機能**
- ベクトル化を支援する **OpenCL*** 形式の NDRange サブグループ
- 汎用 / 関数ポインターの**サポート**
- その他の**多くの機能**

この記事では、CUDA* コードを DPC++ に移行するケーススタディーを紹介します。

ケーススタディー : インテル® プロセッサー・グラフィックス上での計算カーネルの実行

電波天文学の画像処理に広く使用されている **Hogbom Clean 画像処理アルゴリズム** (英語) について考えてみます。この画像処理アルゴリズムには 2 つのホットスポットがあります。

- **`Find Peak`**
- **`SubtractPSF`**

説明を簡潔にするため、ここでは **`Find Peak`** のパフォーマンスに注目します。オリジナルの実装は、C++、OpenMP*、CUDA*、および OpenCL* で記述されていました。ホスト CPU は、GPU が利用可能な場合、CUDA* と OpenCL* カーネルを GPU にオフロードします (CUDA* は NVIDIA* GPU にのみ計算をオフロードする独自のアプローチです)。**図 4** と **5** はそれぞれ、ホストコードとデバイスコードを示します。

```
cudaMalloc((void **) &d_peak, nBlocks * sizeof(Peak));
// Find Peak - CUDA Code
d_findPeak<<<nBlocks, findPeakWidth>>>(d_image, size, d_peak);
cudaMemcpy(&peaks, d_peak, nBlocks * sizeof(Peak), cudaMemcpyDeviceToHost);
...
Peak p;
for (int i = 0; i < nBlocks; ++i) {
  if (abs(peaks[i].val) > abs(p.val)) {
    p.val = peaks[i].val;
    p.pos = peaks[i].pos;
```

4 **Find Peak のホストコード : C++、CUDA***

```
__shared__ float maxVal[findPeakWidth];
__shared__ size_t maxPos[findPeakWidth];
const int column = threadIdx.x + (blockIdx.x * blockDim.x);
...
for (int idx = column; idx < size; idx += 4096) {
  if (abs(image[idx]) > abs(maxVal[threadIdx.x])) {
    maxVal[threadIdx.x] = image[idx];
    maxPos[threadIdx.x] = idx;
  }
}
__syncthreads();
if (threadIdx.x == 0) {
  absPeak[blockIdx.x].val = 0.0;
  ...
  for (int i = 0; i < findPeakWidth; ++i) {
    if (abs(maxVal[i]) > abs(absPeak[blockIdx.x].val)) {
      absPeak[blockIdx.x].val = maxVal[i];
...
```

5 **Find Peak のデバイスコード : CUDA***

CUDA* から DPC++ への移行は、手動で行うことも、**インテル® DPC++ 互換性ツール (インテル® DPCT)** (英語) を使用して行うこともできます。インテル® DPCT は、CUDA* プログラムから DPC++ への移行を支援します (**図 6** と **7**)。インテル® DPCT を使用するには、インテル® oneAPI ベース・ツールキットと NVIDIA* CUDA* ヘッダーが必要です。次のコマンドを実行するだけで、インテル® DPCT を起動して **`example.cu`** ファイルを移行できます。

```
dpct example.cu
```

複数の CUDA* ファイルを含むアプリケーションを移行する場合、インテル® DPCT の **`--in-root`** オプションでプログラムのソースの場所を指定し、**`--out-root`** オプションで移行したコードの出力先を指定します。アプリケーションが **`make`** や **`cmake`** を使用する場合、**`intercept-build`** を使用して移行することを推奨します。このコマンドは、コンパイラーの呼び出し (C++ ホストコードと CUDA* デバイスコードの入力ファイル名と関連するコンパイラー・オプション) を含む、コンパイル・データベース・ファイル (**`.json`** ファイル) を作成します。

```
intercept-build make
dpct -p=<path to .json file> --out-root=dpct_output ...
```

Hogbom Clean CUDA* コードを DPC++ へ移行する場合、CUDA* カーネルを含む **`HogbomCuda.cu`** ファイルに対してインテル® DPCT を呼び出すか、**`intercept-build`** を使用できます。デフォルトでは、移行したコードのファイル拡張子は **`dp.cpp`** になります。

移行した DPC++ コード (**図 6 - 9**) とオリジナルの CUDA* コード (**図 4** と **5**) を比較してみます。

```
dpct HogbomCuda.cu --out-root=MigratedCode --cuda-include-path=<CUDA-Headers>
OR
intercept-build make
dpct -p=compile_commands.json --out-root=MigratedCode --cuda-include-path=<CUDA-Headers>
```

```
// Find peak - DPC++ Compatibility Tool Host Code Generated
static Peak findPeak(const float* d_image, size_t size)
{
  d_peak = (Peak *)sycl::malloc_device( nBlocks * sizeof(Peak), dpct::get_current_device(),
                                        dpct::get_default_context());
  ...
  dpct::get_default_queue().submit([&](sycl::handler &cgh) {
    sycl::accessor<float, 1, sycl::access::mode::read_write, sycl::access::target::local>
                  maxVal_acc_ct1(sycl::range<1>(1024 /*findPeakWidth*/), cgh);
    sycl::accessor<size_t, 1, sycl::access::mode::read_write, sycl::access::target::local>
                  maxPos_acc_ct1(sycl::range<1>(1024 /*findPeakWidth*/), cgh);

    cgh.parallel_for(
    sycl::nd_range<3>(sycl::range<3>(1, 1, nBlocks)*sycl::range<3>(1, 1, findPeakWidth),
                      sycl::range<3>(1, 1, findPeakWidth)),
                      [=](sycl::nd_item<3> item_ct1) {
                          d_findPeak(d_image, size, d_peak, item_ct1,
                                     maxVal_acc_ct1.get_pointer(),
                                     maxPos_acc_ct1.get_pointer());
                  });
  });
  dpct::get_default_queue().memcpy(&peaks, d_peak, nBlocks * sizeof(Peak)).wait();
  ...
}
```

6 インテル® DPCT を使用して移行した Find Peak DPC++ ホストコード

```
CUDA Host Code
---------------
// device kernel call by host.
d_image and size are read inside
the d_findPeak kernel code.
Output is written to d_peak.

d_findPeak<<<nBlocks,
findPeakWidth>>>(d_image, size,
d_peak);
```

```
DPC++ Host Code
---------------
dpct::get_default_queue().submit([&](
sycl::handler &cgh) { ...
  cgh.parallel_for( ...
    d_findPeak(d_image, size, d_peak,
        item_ct1,
        maxVal_acc_ct1.get_pointer(),

maxPos_acc_ct1.get_pointer());
  ...); // device queue, parallelism
});
```

7 Find Peak の CUDA* ホストコードと移行した DPC++ ホストコードの比較

```
void d_findPeak(const float* image, size_t size, Peak* absPeak, sycl::nd_item<3> item_ct1,
                float *maxVal, size_t *maxPos)
{
  const int column = item_ct1.get_local_id(2) +
                     (item_ct1.get_group(2) * item_ct1.get_local_range().get(2));

  maxVal[item_ct1.get_local_id(2)] = 0.0;
  maxPos[item_ct1.get_local_id(2)] = 0;

  for (int idx = column; idx < size; idx += 4096)
  {
    if (sycl::fabs(image[idx]) > sycl::fabs(maxVal[item_ct1.get_local_id(2)]))
    {
      maxVal[item_ct1.get_local_id(2)] = image[idx];
      maxPos[item_ct1.get_local_id(2)] = idx;
    }
  }

  item_ct1.barrier();
  if (item_ct1.get_local_id(2) == 0) {
    absPeak[item_ct1.get_group(2)].val = 0.0;
    absPeak[item_ct1.get_group(2)].pos = 0;
    for (int i = 0; i < findPeakWidth; ++i) {
      if (sycl::fabs(maxVal[i]) > sycl::fabs(absPeak[blockIdx.x].val)) {
        absPeak[item_ct1.get_group(2)].val = maxVal[i];
        absPeak[item_ct1.get_group(2)].pos = maxPos[i];
      }
    }
  }
}
```

8 インテル® DPCT を使用して移行した Find Peak の DPC++ デバイスコード

```
CUDA Kernel
------------
__global__
void d_findPeak (const float* image, size_t
size, Peak* absPeak) // device fn.


__shared__ float maxVal[findPeakWidth];

maxVal[threadIdx.x] = ... // local write


__syncthreads();

absPeak[blockIdx.x].v... // global write
```

```
DPC++ Migrated Kernel
---------------------
void d_findPeak (const float* image,
size_t size, Peak* absPeak,
sycl::nd_item<3> item_ct1,
float *maxVal, size_t *maxPos)

float *maxVal // Passed as function param

maxVal[item_ct1.get_local_id(2)] = ... //
nd_range accessor

item_ct1.barrier(); // synchronization

absPeak[item_ct1.get_group(2)].v...
```

9 Find Peak の CUDA* カーネルと移行した DPC++ デバイスカーネルの比較

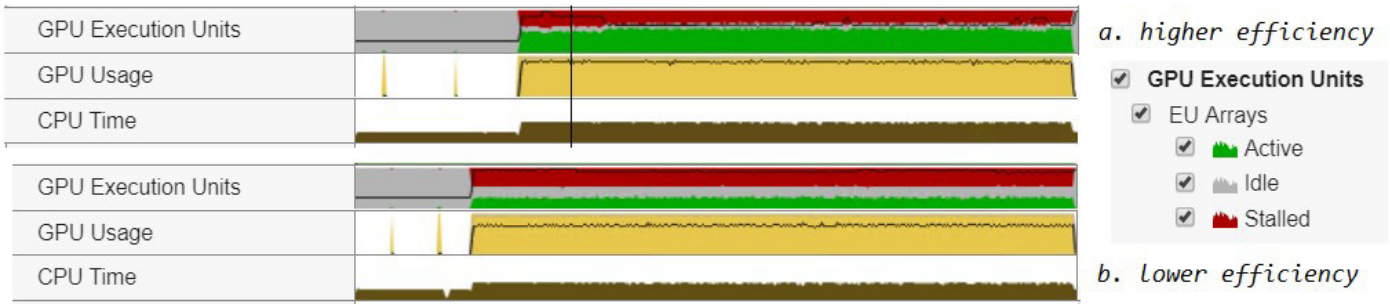
DPC++ コードの重要な点として、**SYCL*** キューを使用したデバイスコードの呼び出し、デバイスコードを実行するラムダ関数ハンドラー、マルチスレッド実行の **parallel _ for** 構造 (オプション) があります。ここで移行した DPC++ コードは、統合共有メモリー (USM) プログラミング・モデルを使用しているにもかかわらず、デバイスカーネルが読み書きするデータをデバイス上のメモリーに割り当てています。そのため、ホストとデバイス間で明示的なデータコピーが必要です。代わりに、共有メモリーに割り当てることで、ホストとデバイスの両方からアクセスして更新することができます。SYCL* のバッファーとアクセサーを使用してデータ転送を行う非 USM コードは、ここには表示されていません。

インテル® DPCT によって移行されたコードは、現在のデバイスを決定し、そのデバイス用のキューを作成します (**get_current_device()** と **get_default_queue()** を呼び出します)。DPC++ コードを GPU へオフロードするには、**sycl::gpu_selector** パラメーターを使用してキューを作成する必要があります。処理するデータは、デバイス上だけでなく、GPU 上で実行するカーネルからも利用できるようにすべきです。GPU との間でコピーされるデータの次元とサイズは、**sycl::range**、**sycl::nd_range** で指定します。インテル® DPCT を使用する場合、CUDA* コードの各ソース行は同等の DPC++ コードに移行されます。**Find Peak** のデバイス・カーネル・コード (**d_findPeak**) の場合、(CUDA* コードから) 生成された DPC++ コードは、ほぼ 1 対 1 で対応付けられます。したがって、インテル® DPCT は、迅速に移行してプロトタイプを生成できる非常に強力なツールと言えます。移行した DPC++ コードと CUDA* コードの比較を **図 7** と **9** に示します。

インテル® DPCT を使用してコードを DPC++ へ移行したら、次に正当性と効率を確認します。インテル® DPCT は、プリプロセッサー・ディレクティブ変数を値に置換することがあります。これを手動で元に戻す必要があります。また、CUDA* の **threadId.x** から同等の **nd_range** アクセサーへの置換など、移行したコードで修正が示されている場合、コンパイルエラーが発生することがあります。Hogbom Clean アプリケーション・コードには正当性チェッカーがあります。これを利用して、GPU 上で実行した DPC++ コードの結果とホスト CPU 上で実行したベースライン C++ 実装の結果を比較して正当性をチェックします。

次に、GPU 上で移行した DPC++ コードの利用率 (EU 占有率、キャッシュの利用状況、SP または DP FLOPS) とホストとデバイス間のデータ転送を解析することで、移行した DPC++ コードの効率を特定します。ワークグループのサイズ / 範囲の次元などのパラメーターは、GPU 利用率に影響します。これは、Hogbom Clean アプリケーションの **Find Peak** では **nBlocks** と **findPeakWidth** です。

図 10 は、パフォーマンスへの影響とチューニングの可能性を示すため、**nBlocks** の値を **24** と **4** に設定して収集したパフォーマンス・プロファイルです。**findPeakWidth** を **256** に設定して、GPU プロファイルをサポートする**インテル® VTune™ プロファイラー**でプロファイルしました。CUDA* を使用する NVIDIA* GPU で最適なパラメーター設定は、DPC++ コードを実行するインテル® GPU では最適でないため、インテル® DPCT を使用する場合、明示的なチューニングが必要になります。**表 1** は、Gen9 (48 EU) 上で収集したメトリックです。



10 **Gen9 上の Hogbom Clean プロファイル (nBlocks = (a) 24 と (b) 4 の場合)**

表 1. Find Peak ホットスポットの Gen9 GPU 上でのパフォーマンス・メトリック

関数	グローバルサイズ	ローカルサイズ	実行時間 (秒)	インスタンス	%GPU アレイ			CPU 利用率 %	L3 シェーダー帯域幅 (GB/秒)
					アクティブ	ストール	アイドル		
d_find Peak	6,144	256	A: 3.61	1,001	36.5	48.8	14.7	8.6	19.5
	1,024	256	B: 8.65	1,001	13.1	52.5	34.4	1.7	8.3

GPU 利用率と効率の最適化に加えて、ホストとデバイス間のデータ転送もチューニングすべきです。Hogbom Clean アプリケーションは、**`Find Peak`** を複数回呼び出しており、**`SubtractPSF`** カーネルとカーネルで使用されるデータはデバイス上に配置できます。そうすることで、ホストとデバイス間で再割り当てやコピーが不要になります (データ転送と USM に関連する最適化については、今後の記事で取り上げます)。

優れたアルゴリズムを記述する

インテル® プロセッサー・グラフィックス・アーキテクチャーと DPC++ の機能を理解することは、より優れたアルゴリズムと移植性の高い実装を記述するのに役立ちます。この記事では、アーキテクチャーの詳細を説明し、DPC++ 構文とインテル® DPCT を使用したケーススタディーを紹介しました。インテル® GPU 上で最高のパフォーマンスを達成するには、特にインテル® DPCT を使用する場合は、カーネル・パラメーターをチューニングすることが重要です。最新のインテル® ハードウェアとソフトウェアでアプリケーションを開発、テスト、実行できる**インテル® DevCloud** (英語) を使用することを推奨します。

ヘテロジニアス・アーキテクチャーのプログラミングに OpenMP* オフロードを使用する

アクセラレーターへのオフロードに OpenMP* ディレクティブを使用する方法

Jose Noudohouenou インテル コーポレーション ソフトウェア・エンジニア
Nitya Hariharan インテル コーポレーション アプリケーション エンジニア

OpenMP* は、バージョン 4.0 からアクセラレーター・オフロードをサポートしています。**ベータ版 インテル® oneAPI ベース・ツールキット** (英語) は、インテル® C/C++ および Fortran コンパイラーで OpenMP* オフロードをサポートします。この記事は、OpenMP* オフロード・ディレクティブの使い方を示します。OpenMP* のいくつかのプラグマを説明し、プログラムのコンパイル方法を示し、サンプル・アプリケーションを使用して OpenMP* のアクセラレーター・オフロード向けに必要なコード変更について述べます。また、**インテル® VTune™ プロファイラー**と**インテル® Advisor** を使用して、パフォーマンスのホットスポットを見つけ、コードを最適化する方法を紹介します。

OpenMP* オフロード・ディレクティブ

OpenMP* の **declare target** ディレクティブは、アクセラレーター・デバイス上の関数、サブルーチン、または変数をマップしてアクセス可能にします。**target map** ディレクティブは、デバイス環境にデータをマップして、デバイス上でオフロードコードを実行します。**map** 節のマップタイプは、ホストとデバイス間のデータマップを指定します。

- **to:x** は、デバイス上でデータを読み取り専用にします。
- **from:x** は、デバイス上でデータを書き込み専用にします。
- **tofrom:x** は、ホストとデバイスの両方でデータを読み書きできるようにします。

図 1 のコードは、次の 4 種類のオフロードを実行する SAXPY の例です。

- **DEFAULT**
- **TEAMS**
- **THREADS**
- **TEAM_THREADS**

それぞれ異なる OpenMP* ディレクティブを使用します。**DEFAULT** は、基本の OpenMP* オフロード・ディレクティブを使用します。**TEAMS** は、**num_teams** 節を使用してスレッドチームを生成します。チームとスレッドの数は指定しません。これらは実行時に選択されます。**THREADS** は **TEAMS** に似ています。**thread_limit** 節でチームごとのスレッド数を指定します。**TEAM_THREADS** は、チームの数とチームごとのスレッド数の両方を指定します。

```
void saxpy(int n, float a, float *restrict x, float *restrict y) {

#ifdef DEFAULT
        #pragma omp target parallel for map(to: x[0:n], a) map(tofrom: y[0:n])

#elif TEAMS
        #pragma omp target teams distribute parallel for map(to: x[0:n], a)
map(tofrom: y[0:n])

#elif THREADS
        #pragma omp target teams distribute parallel for map(to: x[0:n], a)
map(tofrom: y[0:n]) thread_limit(112)
#elif TEAM_THREADS
        #pragma omp target teams distribute parallel for map(to: x[0:n], a)
map(tofrom: y[0:n]) num_teams(4) thread_limit(112)
        #else
                #pragma omp parallel for
        #endif
                for (int i = 0; i < n; i++)
                        y[i] = a*x[i] + y[i];}
```

1 **4 種類の OpenMP* オフロードを使用する SAXPY コードの例**

SAXPY のそれぞれのオフロードは、メインプログラム内で通常のルーチンとして呼び出すことができます。**DEFAULT** は次のようにコンパイルできます。

```
$icx -O3 -std=c99 -g -no-ipo -fiopenmp -fopenmp-targets=spir64 -
D__STRICT_ANSI__ -DDEFAULT main.c -o saxpy_default
```

-fiopenmp オプションは、LLVM で OpenMP* の変換を有効にします。**-fopenmp-targets=spir64** オプションは、GPU デバイス向けに x86 + SPIR64 ファットバイナリーを生成するようコンパイラーに指示します (コンパイルプロセスの詳細は、「**ベータ版インテル® C++ コンパイラーおよびベータ版インテル® Fortran コンパイラー向けの GPU への OpenMP* オフロード導入**」の「C++ および Fortran の PVC ディレクティブのサポート強化」または『**インテル® oneAPI プログラミング・ガイド**』の「プログラミング・インターフェイス」>「コンパイルモデル」を参照)。

ほかの種類のオフロードをコンパイルするには、**-D** オプションを使用します (例えば、**TEAMS** の場合は **-DTEAMS** を使用します)。次のようにプログラムを実行します。

```
$./saxpy_default 1000000
```

ケーススタディー : 量子色力学

このセクションでは、OpenMP* オフロードプラグマを **MILC アプリケーション** (英語) に追加します。ここでは、インテル® Iris® Pro グラフィックス 580 (Gen9) GPU が統合されたインテル® Core™ i7-6770HQ CPU 2.60GHz を搭載した NUC マシンを使用します。Gen9 は 72 個の実行ユニット (EU) を備えており、3 つのスライス、スライスごとに 3 つのサブスライス、サブスライスごとに 8 つの EU があります。L3 キャッシュのサイズは 3 * 512KB = 1.5MB で、LLC のサイズは 8MB (CPU コアと共有) です。すべての MILC 実行は **wx12t12.in** 入力ファイルを使用します。

ここでは、OpenMP* オフロードを有効にすることに注目します。最初に、インテル® Advisor を使用してコードをプロファイルし、パフォーマンスのホットスポットを見つけます。**表 1** は、ホスト CPU 上の上位 4 つのホットスポットです。これらのコード領域は Gen9 へのオフロード候補です。それぞれのホットスポットに **TEAM_THREADS** の OpenMP* プラグマを追加します。追加後のコードを**図 2** から **5** に示します。

表 1. ベータ版インテル® Advisor のオフロード・アドバイザーで検出された上位の MILC ホットスポット

場所	ホスト上での合計時間
dslash_fn_field_specialompparallel_for@802 at dslash_fn_dblstore.c:802 (dslash_codelet2)	19.57%
dslash_fn_field_specialompparallel_for@704 at dslash_fn_dblstore.c:704 (dslash_codelet1)	18.66%
ks_multicg_offset_field_cpuompparallel_for@529 at ks_multicg_offset.c:529(multicg_offset_codelet2)	5.86%
ks_multicg_offset_field_cpuompparallel_for@445 at ks_multicg_offset.c:445(multicg_offset_codelet1)	4.56%

```
void dslash_codelet1(//pass loop indices, source and destination matrices)
{
  #pragma omp target teams distribute parallel for private(//private variables)
map(to: //list of read only variables) map(tofrom://list of read+write variables)
num_teams(72) thread_limit(112)

    for( i=loopstart; i<loopend; i++){

       //Set indices in x, y and z directions
       //calls to matrix routines
       }
}
```

2 **MILC dslash_codelet1**

表 2 は、Gen9 上で各 MILC コードを実行したインテル® VTune™ プロファイラーの GPU オフロード解析結果です。ホットスポットとともに、コード領域で使用された SIMD 幅 (**S_{width}**、8 または 16) が表示されています。この情報を利用して、最適な **thread_limit** 値になるようにコードをチューニングできます。**num_teams** 値は EU の数に設定します。**thread_limit** 値の設定には、ループストライドに関する情報が必要です。ループストライドが 1 の場合、最適な **thread_limit** 値は EU ごとのハードウェア・スレッド数 **($N_{threads}$) * S_{width}** になります。ストライドが 1 よりも大きい場合、**thread_limit** 値はストライド以上の **S_{width}** の最初の倍数になります。ローカル・ワーク・サイズ (LWS) とグローバル・ワーク・サイズ (GWS) は、以下の式で決定されます。

$$LWS = S_{width} * N_{threads}$$
$$GWS = LWS * \text{number of EUs}$$

Gen9 には EU ごとに 7 つのスレッドがあり、合計 72 個の EU があります。LWS は、**multicg_offset_codelet1** では 56 ですが、それ以外のコードでは 112 です。GWS は、**dslash_codelet1**、**dslash_codelet2**、および **multicg_offset_codelet2** では 112 * 72 で、**multicg_offset_codelet1** では 56 * 31 です。**multicg_offset_codelet1** には、リダクション操作があり、負荷の低い計算処理を並列に実行する複数のスレッドを同期する必要があります。同期コストを軽減するため、このコードを実行する EU の数は少なくした方が良いでしょう。また、効率に関しても、**num_teams** を 31 に設定すると、**multicg_offset_codelet1** カーネルやループ・インスタンスを実行する (初期割り当ての) 総スレッド数の割合が 97.23% と高くなります。「呼び出し回数」カラムはコードが呼び出された回数を示し、「GPU メモリー帯域幅」はコードごとのデータの読み書き速度を示します。これらの特性はすべてホットスポットのパフォーマンスに影響します。

```
void dslash_codelet2(//pass loop indices, source and destination matrices)
{
   //local variables and su3 matrix

#pragma omp target teams distribute parallel for private(//private variables)
map(to: //read-only variables) map(tofrom: //read-write data) num_teams(72)
thread_limit(112)

    for( i=loopstart; i<loopend; i++){

//Set indices in x, y and z directions
//calls to matrix routines
  }}
```

3 **MILC dslash_codelet2**

```
void multicg_offset_codelet1(//pass loop indices, source and destination
matrices)
{
  #pragma omp target teams distribute parallel for private(i,j) map(to: //read
only  data) map(tofrom: //read-write data) num_teams(31) thread_limit(56)
reduction(+:rsq)
   {
for( i=loopstart; i<loopend; i++){
        for(j=0;j<num_offsets_now;j++)
        {
          //call scalar_mult_add routine;
        }

//post loop computation
      }
  }
 *rsq_=rsq;
}
```

4 **MILC multicg_offset_codelet1**

表 2. インテル® VTune™ プロファイラーによる MILC TEAM_THREADS オフロードの GPU オフロード解析結果

ホットスポット	S_{width}	GWS	LWS	呼び出し回数	GPU メモリー帯域幅 (GB/秒)	
					リード	ライト
dslash_codelet1	16	8,064	112	74,084	2.073	0.020
dslash_codelet2	16	8,064	112	74,083	1.993	0.021
multicg_offset_codelet1	8	1,736	56	74,082	15.167	6.048
multicg_offset_codelet2	16	8,064	112	36,352	3.990	1.835

```
void multicg_offset_codelet2(//pass loop indices, source and destination
matrices)
{
#pragma omp target teams distribute parallel for private(i,j) map(to: //rad-
only data) map(tofrom: //read-write data) num_teams(72) thread_limit(112)

for( i=loopstart; i<loopend; i++){
        for(j=0;j<num_offsets_now;j++) {
          //calls to add/multiply routines
      }
      ind=j_low+i;
      offload_su3vec_copy(&pm1[ind],cg_p+i); }
}
```

5 **MILC multicg_offset_codelet2**

最適なパフォーマンスの実現

インテル® oneAPI ベース・ツールキットで提供される OpenMP* のオフロードサポートは、アプリケーションの計算負荷の高い領域をアクセラレーターへ容易にオフロードできるようにします。ここで紹介した手法に従って、オフロードに適したコード領域を特定し、適切な OpenMP* オフロードプラグマを使用してオフロードし、プロファイルとチューニングにより最適なパフォーマンスを実現できます。

ARTICLE HIGHLIGHTS

インテル® oneAPI DevCloud を使用した COVID-19 関連の医用画像処理の実験に関する論文

データ・サイエンティストは、インテル® DevCloud でインテル® AI アナリティクス・ツールキットを使用して、胸部スキャンから COVID-19 を検出し、人命救助に貢献しています。

この論文はこちら (英語) からご覧になれます。>

OpenMP* ベクトル化サポートの謎に迫る

OpenMP* の SIMD ディレクティブを使用してコンパイラーにベクトル化するコードを伝える方法

Clay P. Breshears 博士 Omics Data Automation, Inc. 主席エンジニア

OpenMP* は、pthread* などの明示的なスレッド・ライブラリーよりも使いやすく、分かりやすい、コンパイラー主導のスレッド化の標準を提供するため、20 年以上前に誕生しました。スレッドの生成と破棄、管理、およびスレッドへの計算の割り当てに関する多くの操作は、コンパイラー・ディレクティブに抽象化されました。現代の CPU には、データの複数の部分に対して同じ命令を実行できるベクトルレジスターが備わっています。そのような命令レベルの並列処理は、パフォーマンスと電力効率を大幅に向上します。しかし、スレッドレベルの並列処理と同様に、

コンパイラーが適切にベクトル化できるように手助けが必要になることがあります。プログラマーがコンパイラーにベクトル化可能な計算を知らせることができるように、SIMD (Single-Instruction, Multiple Data) ディレクティブと節が OpenMP* に追加されました。

単純なベクトル化の例

図 1 のコードは、**x** が **0** から **1** の区間について関数 **4.0 / (1.0 + x²)** の曲線下面積を計算して、**π** を求めます。コードで使用される一般的な処理は、数値積分またはリーマン和として知られています。

```
#include <stdio.h>
#include <omp.h>

size_t num_rects = 10000000;

int main(int argc, char* argv[])
{
        double x, width, sum=0.0;
        size_t i;
        double area;
        width = 1./(double)num_rects;
        for (i=0; i<num_rects; ++i)
        {
                x = (i + .5)*width;
                sum += 4.0/(1.+ x*x);
        }
        area = width * sum;
        printf("The value of PI is %15.12f\n",area);
        return 0;
}
```

1 **リーマン和のシリアルコード**

線形代数の授業を思い出してみてください。x 軸の固定値間の曲線下面積を推定する中点の法則というものがあったでしょう。区間は均等な幅の帯に分割され各帯の中点における関数の値が計算されます。各帯の幅に中点の関数値 (高さ) を掛けて、帯 (そのほとんどは曲線の下にある) の面積を求めます。区間のすべての帯の面積の合計は、実際の数値解に近い値になります。

選択する帯の幅によっては、帯の一部は曲線の外側となり、曲線の内側の一部は帯に含まれません。しかし、帯の幅を狭くしたり、帯の数を増やすことで、推定値の精度が上がります。

図 1 のループは、ループ・イテレーター `i` を介して値のセットを一度に 1 つずつ生成します。変数 `x` は、関数値 (中点における帯の高さ) の計算に使用される中点の値を保持します。関数値は合計 `sum` に集計されます。ループが完了したら、すべての帯の幅 (固定値 `width`) に高さの合計を掛けて面積を計算します (面積の計算は、ループの各反復で高さを `sum` に加算する前に行うこともできますが、ここでは最適化のため、ループの反復回数分実行する代わりに、ループの外側で一度だけ実行しています)。

帯の面積を計算するループの各反復は独立しているため、OpenMP* を使用してループをスレッド化し、各スレッドで計算された部分解を合計して解を求めることもできますが、この記事の目的はこのループをベクトル化する方法を示すことであるため、ここでは取り上げません。

ループ本体は、各ループ反復で生成される値の範囲に対して、同じ計算を繰り返し実行します。`x` を計算するため加算と乗算を行い、`sum` に集計する値を計算するため乗算、加算、および除算を行います。これらの値のセットをベクトルレジスターにロードできれば、より少ない命令サイクルで計算を実行できます。このループは、各反復で 6 つの算術演算を実行し、これを 1,000 万回繰り返しています。4 つまたは 8 つの値を 1 つのベクトルレジスターにパックして、1 つの命令でそれらの値すべてに対して算術演算を実行することで、実行される命令数は 6,000 万スカラー命令から 1,500 万または 750 万ベクトル命令に減ります。

OpenMP* SIMD ディレクティブの簡単な使用例

ベクトルレジスターに複数の値をロードする必要があるため、基本的な OpenMP* SIMD ディレクティブの `omp simd` を使用して、複数の値を「生成」し、それぞれに対して同じ計算を実行するループをベクトル化します。**図 2** は、**図 1** のループに OpenMP* ディレクティブを追加したものです。このディレクティブは、ループ内の計算をスカラー操作からベクトル操作に変換できることをコンパイラーに知らせます。

```
#pragma omp simd private(x), reduction(+:sum)
        for (i = 0; i < num_rects; ++i)
        {
                x = (i + .5)*width;
                sum += 4.0/(1.+ x*x);
        }
```

2 **OpenMP* の omp simd ディレクティブを使用してループをベクトル化する例**

各反復 (と対応するループ・イテレーター値 `i`) は、論理的な反復に「バンドル」され、ループ本体の処理は、対応する SIMD ハードウェアを介してバンドルされた反復それぞれに対して実行されます。OpenMP* では、これらのバンドルされた反復を「SIMD チャンク」と呼びます。SIMD チャンクに配置されるシリアル反復の数は、以下の要因に依存します。

- 実行時のハードウェアと命令セットのベクトル演算サポート
- 操作する値のサイズ
- その他の実装固有の要因

表 1 は、リーマン和コードの 2 つのバージョンの実行時間です。ベクトル化によってパフォーマンスが向上することは明らかです。

表 1. リーマン和コードのシリアルバージョンとベクトルバージョンの実行時間

帯の数	シリアル実行時間 (秒)	ベクトル実行時間 (秒)
10,000,000	0.155458	0.023405
100,000,000	1.45208	0.119641
1,000,000,000	14.1644	1.13109
10,000,000,000	141.749	1.13109
100,000,000,000	1417.66	112.581

SIMD ディレクティブにいくつかの節を追加できます。必要に応じて、上記の例に示すように、**private** 節と **reduction** 節を使用できます。このほかにも、コンパイラーが効率良いコードを生成できるようにメモリーのレイアウト / アクセスパターンに関するヒントを提供する節 (後述する **safelen** など) を SIMD ディレクティブに追加できます。ここではすべての利用可能な節を取り上げません。詳細は、OpenMP* 仕様を確認してください。

インテル® コンパイラーを使用する利点

OpenMP* の SIMD サポートについて詳しく述べる前に、**表 1** の結果について説明します。インテル® コンパイラーは長年にわたって改良されており、ベクトル・ハードウェアの変更やインテル® CPU で現在サポートされているベクトル命令についての詳しい知識が活かされています。そのため、多くのケースで OpenMP* SIMD ディレクティブによるヒントがなくても、**図 2** のループがベクトル化可能であることを認識し、コードをベクトル化できます。テストでは、インテル® コンパイラーがループを自動的にベクトル化しないように、最適化を無効にして (**-O0** オプションを指定して) シリアルコードをコンパイルしました。

インテル® コンパイラーの最適化レポートは、ベクトル化と並列化の最適化に関して、コンパイラー独自の考察を提供します。ベクトル化できると仮定したループがベクトル化されなかった場合、最適化レポートでコンパイラーがベクトル化しなかった理由を確認し、コードを見直してベクトル化されるように変更できます。例えば、コードをリファクタリングして操作を効率良い順序に並べ替えたり、OpenMP* の SIMD ディレクティブと節を追加して安全にベクトル化できることをコンパイラーに伝えます。

図 3 は、ベクトル化に関するコンパイラーの考察を得るため、オリジナルのシリアルコード (**図 1**) をコンパイルする際に使用したコマンドです。

[訳者注 : コンパイラー・メッセージは、日本語版のコンパイラーがインストールされている環境では日本語で表示されます。]

```
$> icc -qopt-report=2 -qopt-report-phase=vec pi.cc
icc: remark #10397: optimization reports are generated in *.optrpt
files in the output location
```

3 **ベクトル化に関する最適化レポートを出力するコンパイルコマンド**

このコマンドを実行すると、**図 4** の情報を含む **pi.optrpt** ファイルが生成されます。

```
$> cat pi.optrpt
Intel(R) Advisor can now assist with vectorization and show optimization
  report messages with your source code.
See "https://software.intel.com/en-us/intel-advisor-xe" for details.

Begin optimization report for: main(int, char **)

    Report from: Vector optimizations [vec]

LOOP BEGIN at pi.cc(16,4)
   remark #15410: vectorization support: conversion from int to float will be
emulated    [ pi.cc(17,10) ]
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at pi.cc(16,4)
<Remainder loop for vectorization>
   remark #15410: vectorization support: conversion from int to float will be
emulated    [ pi.cc(17,10) ]
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at pi_omp.cc(16,4)
<Remainder loop for vectorization>
LOOP END
```

4 **出力されたベクトル化レポート**

実際にコンパイルしたコードでは、行 16 は **for** ループで、行 17 は **x** の代入です (イテレーター **i** を **double** にキャストしてみましたが、エミュレーションに関するコメントはなくなりませんでした)。つまり、コンパイラーは、プログラマーの助けがなくても、安全にベクトル化できるコードを識別できます。

ループがベクトル化可能であると考えられる場合、OpenMP* SIMD ディレクティブを追加し、**-qopenmp** オプションを指定してコンパイルし、コンパイラーからの出力を確認してみてください。SIMD ディレクティブを追加してもコンパイラーがループを安全にベクトル化できない場合、**図 5** のように警告メッセージと行番号が示されます。

[訳者注 : インテル® コンパイラーは、**omp simd** が記述された場合、**-qopenmp** オプションが指定されなくてもデフォルトでプラグマまたはディレクティブを有効にします。そのため、**omp simd** によるベクトル化のパフォーマンスへの影響を調査するには、**-qno-openmp-simd** オプションを指定して **omp simd** を無効にします。]

```
$> icc -qopenmp -o nosimd no_simd.cc
no_simd.cc(17): warning #15552: loop was not vectorized with "simd"
```

5 **ベクトル化できないコードに関するコンパイラー出力**

この場合、**-qopt-report** オプションを使用してコンパイラーの最適化レポートを出力し、コンパイラーがコードをベクトル化できない理由を確認します。この情報を基に、以下の点を考慮してコードを再評価します。

- リファクタリングして障害を排除する
- 安全にベクトル化できることをコンパイラーに伝えるため、ディレクティブに節を追加する
- その他の最適化の可能性についてループを調査する

[編集者注 : コンパイラーの最適化レポートと提供される情報の詳細は、本号の記事「**インテル® コンパイラーの最適化レポートを最大限に活用する**」を参照してください。]

safelen 節

図 6 の関数について考えてみます。

```
void scale(float* A, float* B, const float scale, size_t s, size_t e)
{
   for (int i = s; i < e; ++i) {
      A[i] += B[i] * scale;
   }
}
```

6 **ベクトル化候補の別のループ**

このコードは、2 つの浮動小数点配列 (**A**、**B**)、スケール係数 (**scale**)、および第 1 配列の開始位置と終了位置 (**s**、**e**) を受け取り、第 1 配列の要素に第 2 配列の対応する要素のスケーリングされた値を加算して更新する、浮動小数点積和演算です。通常、配列 **A** と配列 **B** がオーバーラップしなければベクトル化できます。

B が **A** の一部であっても、インデックス値が大きい場合 (つまり、配列 **A** の中で現在処理されている要素よりも後にある場合)、コードにはアンチ依存関係または Write after Read (WAR) 依存関係が存在します。例えば、**B** が **A** を 2 要素オフセットしたものである場合、ループ内部をサイズ 4 の SIMD チャンクにできます。

```
A[k  ] += A[k+2] * scale
A[k+1] += A[k+3] * scale
A[k+2] += A[k+4] * scale
A[k+3] += A[k+5] * scale
```

A[k+2] から **A[k+5]** の現在値はベクトルレジスターにコピーされ、スケール係数を掛けてから、ベクトルレジスターにロードされている **A[k]** から **A[k+3]** の現在値に加算されるため、問題なくベクトル化できます。

3 つ目のケースは、**B** が **A** の負のオフセットである場合です。コードに真の依存関係または Read after Write (RAW) 依存関係が存在し、ベクトル化すると問題が生じる可能性があります。オフセットが **-2** の場合について考えてみます。上記の SIMD チャンクは次のようにアンワインドできます。

```
A[k  ] += A[k-2] * scale
     A[k+1] += A[k-1] * scale
     A[k+2] += A[k  ] * scale
     A[k+3] += A[k+1] * scale
```

更新された (正しい) 値が書き込まれる前に **A[k]** と **A[k+1]** のオリジナルの値が読み取られる問題があることは明らかです。この場合、オフセットの最小値が分かっていれば、**safelen** 節を使用して SIMD チャンクの安全な最小長をコンパイラーに知らせることができます。例えば、**A** と **B** が同じ配列の一部であり、2 つの配列の間の要素オフセットが常に 12 以上の場合、**図 6** のオリジナル関数は**図 7** に示すように OpenMP* を使用してベクトル化できます。これにより、コンパイラーはベクトル長が 12 要素以下であれば安全にベクトルコードを生成できます。

```
void scale(float* A, float* B, const float scale, size_t s, size_t e)
{
   #pragma omp simd safelen(12)
   for (int i = s; i < e; ++i) {
      A[i] += B[i] * scale;
   }
}
```

7 **安全なベクトル長をコンパイラーに知らせるため safelen 節を使用**

インテル® コンパイラーでオリジナルの **scale()** 関数 (**図 6**) をコンパイルすると、コードのベクトルバージョンが生成されます。これは、コードに含まれるループを安全にベクトル化できるとコンパイラーが見なしたためです。しかし、真の依存関係が存在する可能性があるため、コンパイラーはループのシリアルバージョンも生成します。実行時に **A** と **B** のパラメーターのアドレスを比較して、適切なバージョンのコードが実行されます。

リーマン和の例に戻る

前述のとおり、インテル® コンパイラーは、安全であると判断した場合はベクトルコードを生成し、そうでない場合はシリアルコードを生成します。しかし、ユーザー定義関数の呼び出しを含むループの場合はどうでしょうか？関数がインライン展開されていない場合、コンパイラーはループを安全にベクトル化できるかどうか分かりません。

図 8 と **9** は、リーマン和のコードを再構成して 2 つのファイルに分割したものです。**図 8** の **pi_func.cc** は、面積を求める曲線の範囲内の **i** 番目の中点の関数値を計算します。**図 9** の **pi_driver.cc** は、リーマン和のドライバーコードの本体です。

```
double pi_func(size_t i, double width)
{
   double x = (i + .5)*width;
   double fVal = 4.0/(1.+ x*x);
   return fVal;
}
```

8 **オリジナルの例から別のファイルに切り分けた計算部分**

BLOG HIGHLIGHTS

GPU-Quicksort: OpenCL* からデータ並列 C++ への移行

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。

この記事の続きはこちらでご覧になれます。>

```
#include <stdio.h>

extern pi_func(size_t i, double width);
size_t num_rects = 10000000;

int main(int argc, char* argv[])
{
        double width, sum=0.0;
        size_t i;
        double area;
        width = 1./(double)num_rects;
        for (i=0; i<num_rects; ++i) {
           sum += pi_func(i, width);
        }
        area = width * sum;
        printf("The value of PI is %15.12f\n",area);
        return 0;
}
```

9 **ループ内からユーザー定義関数を呼び出すドライバーコード**

この方法でコードを構築する利点の 1 つは、計算部分を任意の計算関数に置換できるため、ドライバーコードを汎用性の高いリーマン和アルゴリズムにできることです。**print** 文を変更して、関数名をより汎用的なものに更新すれば、使用する任意の計算関数をドライバーにリンクするだけで処理できます。

pi_func.cc ファイル (**図 8**) にループを検出できないため、コンパイラーは計算をベクトル化できません。ループにユーザー定義の **pi_func()** 呼び出しが含まれるため (**図 9**)、コンパイラーはコードをベクトル化できるかどうか判断できません。実際、ベクトル化に関する最適化レポートを生成するオプションを指定して **pi_func.cc** ファイルをコンパイルすると、空のレポートが生成されます。

declare simd ディレクティブを使用して、プログラマーが安全であることが分かっているコードのベクトル化をコンパイラーに指示できます。最初に、**図 10** に示すように、**pi_func()** 関数にプラグマを追加します。

```
#pragma omp declare simd
double pi_func(size_t i, double width)
{
   double x = (i + .5)*width;
   double fVal = 4.0/(1.+ x*x);
   return fVal;
}
```

10 **declare simd を追加して関数がベクトル化可能であることを示す**

これをドライバー・コード・ファイルの extern 宣言でも行います (図 11)。最後に、オリジナルの例と同様の方法で、pi_func() 呼び出しを含むループがベクトル化可能であることを知らせます。

```
#pragma omp declare simd
extern pi_func(size_t i, double width);
. . .
     #pragma omp simd, reduction(+:sum)
     for (i=0; i<num_rects; ++i) {
         sum += pi_func(i, width);
       }
```

11 **OpenMP* SIMD ディレクティブを追加した図 9 の関連コード行**

pi_func() コードをコンパイルすると、ベクトルバージョン **vec_pi_func()** が生成されます。ドライバー関数内でループ本体は、関数のベクトルバージョン **vec_pi_func()** を呼び出すように変更されます。基本的に、ハードウェアとデータでサポートされるベクトル長が 4 の場合、**pi_func()** の 4 つの呼び出し (パラメーター **i**、**i+1**、**i+2**、および **i+3**) はベクトルバージョン **vec_pi_func()** の 1 つの呼び出しに置換され、4 つの値はベクトル実行のためベクトルレジスターにコピーされます。

表 2 は、関数値の計算を別のファイルに分割したリーマン和コードの 2 つのバージョンの実行時間です。ベクトルバージョンは、**declare simd** 節によりベクトル化されます。この例のベクトルバージョンは、シリアルバージョンよりも高速ですが、オリジナルのベクトルバージョンほど高速ではありません。

表 2. ユーザー定義関数を含むリーマン和コードのシリアルバージョンとベクトルバージョンの実行時間

帯の数	シリアル実行時間 (秒)	ベクトル実行時間 (秒)
10,000,000	0.03790	0.02247
100,000,000	0.32864	0.18099
1,000,000,000	3.23194	1.74271
10,000,000,000	32.3014	17.4235
100,000,000,000	323.560	173.926

実行時間の短縮

同時に実行可能な計算については、そのような実行をサポートするハードウェア上でベクトルバージョンを使用することで、同じマシンサイクル内で実際に複数の計算を行って実行時間を短縮できます。これは並列コードでも同様です (**図 1** のオリジナルのリーマン和コードの **`for`** ループをスレッド化およびベクトル化してみてください)。

単純なケースでは、インテル® コンパイラーは適切なコンパイラー最適化レベルで計算ループを自動的にベクトル化できます。インテル® コンパイラーの最適化レポートは、ベクトル化されたものとされなかったもの (およびその理由) を提供します。コンパイラーの保守的な判断によってベクトル化されなかった場合、コードを再構成するか、OpenMP* の SIMD ディレクティブと節を追加して安全にベクトル化できることをコンパイラーに知らせることができます。SIMD プラグマと節の一覧は、コンパイラーでサポートされている OpenMP* バージョンのドキュメントを確認してください。

コンパイルと実行の詳細

- すべてのコンパイルと実行は**インテル® DevCloud** (英語) 上で実施されました。
- 使用したインテル® コンパイラーのバージョンは 19.1 20200306 です。
- 実行は、**インテル® Xeon® Platinum プロセッサー**と 384GB のメモリーを搭載したシステム上で、次のコマンドで起動した対話型セッションを使用して行いました。
  `qsub -I -l nodes=1:plat8153:ppn=2 -d`

インテル® コンパイラーの最適化レポートを最大限に活用する

コンパイラーで高速なコードを作成する

Mayank Tiwari インテル コーポレーション テクニカル・コンサルティング・エンジニア
Rama Malladi インテル コーポレーション グラフィックス・パフォーマンス・モデリング・エンジニア

アプリケーションのパフォーマンスを向上するには、通常、アルゴリズムとアーキテクチャーの 2 種類の最適化を行います。多くのプログラマーは、コンピューター・サイエンスのカリキュラムで教わる、アルゴリズムの最適化は熟知していることでしょう。しかし、アーキテクチャーの最適化はそれほど容易なものではありません。ハードウェアをできるだけ効率良く使用するには、アーキテクチャーの知識に加えて、コンパイラー、プロファイラー、ライブラリーなどのツールによるサポートが必要です。

最新のコンパイラーは強力であり、多くのコード変換と最適化を適用してアプリケーションのパフォーマンスを向上できます。また、パフォーマンスを向上するコードのチューニングについてプログラマーにフィードバックを提供することもできます。

この記事では、C、C++、Fortran コンパイラーの機能である、最適化レポートについて説明します。各レポートは、コンパイルされたオブジェクト / バイナリーファイルとともに生成されるテキストファイルであり、コンパイラーにより生成されたコードの詳細な情報を得ることができます。レポートには、ループ変換、ベクトル化、インライン展開、その他の最適化に関する情報が含まれています。

最適化レポートの生成

Linux* および macOS* で最適化レポートを生成するには、**-qopt-report[=n]** コンパイラー・オプションを使用します。Windows* では、**/Qopt-report[:n]** を使用します。**n** はオプションで、レポートの詳細レベルを示します。有効な値は、**0** (レポートなし) から **5** (詳細) です。レベル **n=1** から **n=5** の各レベルでは、前のレベルのすべての情報と、レベルに応じた追加の情報が含まれます。ここでは、レベル **n=5** を使用して生成されたレポートに含まれる、インテル® C/C++ コンパイラーにより行われたループの最適化に注目します (今後の記事で、ほかの最適化レポートのトピックも取り上げる予定です)。

ループ変換

ループは、プログラミング言語で最も重要な制御フロー文の 1 つです。小さな最適化により、多くの命令に影響を与え、全体のパフォーマンスを向上できることから、ループは重要な最適化の候補です。コンパイラーは、特にループをターゲットとして、ループアンロール、ループ融合、ループ交換などの変換を行います。コンパイラー・レポートですべての変換を確認して、パフォーマンスを向上する指示をコンパイラーに送ることができます。

図 1 は、次元 (**size * size**) の行列乗算 **matrixA**、**matrixB**、**matrixC** のコードです。ここでは、**size** を **256** に設定して、**matrixA** と **matrixB** に 1 から 10 のランダムな要素を格納しています。最初に、インテル® C++ コンパイラー 19.1 を使用して、このコードをコンパイルします。

Windows*:

```
icl matrix1.cpp -o matrix1 /Qopt-report=5 /O1
```

Linux*:

```
icpc matrix1.cpp -o matrix1 -qopt-report=5 -O1
```

バイナリー **matrix1** の、詳細レベル **n=5** および最適化レベル **O1** でコンパイラーにより提供されるすべての情報を含む最適化レポート **matrix1.optrpt** が生成されます [訳者注 : 記事内のコンパイラー・メッセージは、日本語版のコンパイラーがインストールされている環境では日本語で表示されます]。

```
56: for(int i = 0; i < size; i++){
57:   for(int j = 0; j < size; j++){
58:     matrixC[i][j] = 0;
59:   }
60: }
61: clock_t t1, t2;
62: t1 = clock();
63: for(int i = 0; i < size; i++){
64:   for(int j = 0; j < size; j++){
65:     for(int k = 0; k < size; k++){
66:       matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
67:     }
68:   }
69: }
70: t2 = clock();
71: cout << (double)(t2 - t1)/CLOCKS_PER_SEC << " seconds" << endl;
```

1 **行列乗算コード**

ここで生成されたコンパイラーの最適化レポートを、理解しやすいように分割して説明します。最適化レポートで最初に目にするのは、コンパイラーのバージョンとコンパイラー・オプションのセットです。

```
Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64,
Version 19.1.0.166 Build 20191121
Compiler options: /o:m1 /Qopt-report=5 /O1
```

この後に、インライン展開のメトリックが続きます (今後の記事で取り上げる予定です)。**Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]** セクションで、コンパイラーにより行われたループの最適化を確認します。**/O1** オプションを使用してコードをコンパイルしたため、コンパイラーはループ関連の最適化を無効にし、最適化レポートには次のメッセージが表示されます。

```
remark #15320: routine skipped: loop optimizations disabled
```

最適化レポートの残りの部分には、コード生成の最適化、インライン展開、プロシージャー間の最適化 (IPO)、オフロード (存在する場合) の情報が含まれます。

図 1 のコードを実行して実行時間を測定します。今回使用したテストシステムで、**/O1** コンパイラー・オプションを使用した場合の実行時間は約 15 ミリ秒でした。次に、デフォルトの **/O2** コンパイラー・オプションを使用します。

Windows*:

```
icl matrix1.cpp -o matrix1 /Qopt-report=5 /O2
```

Linux*:

```
icpc matrix1.cpp -o matrix1 -qopt-report=5 -O2
```

/O2 を使用すると、最適化レポートにいくつかの情報が追加されます。まず、コンパイラー・オプションが **/O2** になります。

```
Compiler options: /o:m1 /Qopt-report=5 /O2
```

そして、ループの最適化レポートが追加されます。例えば、行 56 と 57 のループは畳み込み / 融合され、**memset()** に置換されています。**matrixC** はゼロに設定されます。**memset()** は **vector** や **rep move** 命令を使用して配列を初期化するため、**for** ループよりも効率的です。

```
LOOP BEGIN at C:\Users\Documents\Optimization_Reports\matrix1.cpp(56,2)
remark #25420: Collapsed with loop at line 57
remark #25408: memset generated

LOOP BEGIN at C:\Users\Documents\Optimization_Reports\matrix1.cpp(57,3)
remark #25421: Loop eliminated in Collapsing
```

さらにスクロールすると、行 63 の最も外側のループに対して次の最適化リマークが示されています。

```
LOOP BEGIN at C:\Users\Documents\Optimization_Reports\matrix1.cpp(63,2)
remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
```

この最適化リマークは、コンパイラーによる非常に優れたループ変換を示しています。ループ交換またはループの入れ子の交換と呼ばれるこの最適化は、内部ループと外部ループを交換します。ループ変数が配列のインデックスとして使用されていると、この変換により、場合によっては参照の局所性 (キャッシュヒット) が向上します。リマーク (1 2 3) --> (1 3 2) は、コンパイラーが行 63、64、65 の実行を 63、65、64 に変更したことを示しています。これは、行列乗算が **図 2** のように行われるようになったことを意味します。

```
63: for(int i = 0; i < size; i++)
65:   for(int k = 0; k < size; k++)
64:     for(int j = 0; j < size; j++)
66:       matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
```

2 **コンパイラーによるループ交換が適用された後の行列乗算コード**

テストシステムでは、**/O2** オプションを使用した場合の実行時間は約 4 ミリ秒でした。**/O1** オプションと比較して、約 4 倍のスピードアップを達成できたことになります。これは、コンパイラーの最適化 (例えば、キャッシュの局所性) が優れていることを示しています。**/O3** コンパイラー・オプションを使用すると、より積極的な最適化が適用され、コードで追加のループ変換が行われます。更新された最適化レポートには、次の情報が含まれます。

```
LOOP BEGIN at C:\Users\Documents\PUM\Optimization_Reports\matrix1.cpp(65,4)
remark #25440: unrolled and jammed by 4 (pre-vector)
```

/O3 では、コンパイラーは行 65 のループを 4 回アンロールしてジャムします。ループアンロールおよびジャムは、ループ本体のコードを増やし、ループの反復回数を減らします。プロセッサーの使用率が高くなり、ループ実行の条件付き操作とインクリメント操作の数が減るため、多くの場合、パフォーマンスが向上します。ただし、**/O3** 最適化後の実行時間は約 4 ミリ秒であり、このコードサンプルでは大きな効果が得られませんでした。

ループの入れ替え

コンパイラーがループの入れ替えを適用する別のコードを見てみましょう。この変換では、ループ本体を複製することにより、ループ内の条件文 (**図 3**) が **`if-else`** 構文に変更されます。変換されたループは、コンパイラーが最適化しやすくなります。

```
10:    int i, c, A[1000], B[1000];
11:    c = rand()%2;
12:    for (i = 0; i < 1000; i++) {
13:      A[i] += B[i];
14:      if (c)
15:        B[i] = 0;
16:    }
```

3 **ループの入れ替えの最適化を適用する前のコード**

`c` が初期化されていない場合、またはランダムな値で初期化されている場合、コンパイラーは 2 つのループを生成します。

1. **`if`** 条件のループ
2. **`else`** 条件のループ

最適化レポートの、条件文がループからホイストされたことを示す以下のリマークから、この最適化を確認できます。

```
LOOP BEGIN at
C:\Users\Documents\Optimization_Reports\loop_unswitching.cpp(12,2)
<Predicate Optimized v1> remark #25422: Invariant Condition at line 14 hoisted
out of this loop.
```

図 4 に変換後のコードを示します。

```
if (c) {
    for (i = 0; i < 1000; i++) {
      A[i] += B[i];
      B[i] = 0;
    }
  } else {
    for (i = 0; i < 1000; i++) {
      A[i] += B[i];
    }
  }
```

4 **ループ本体にループの入れ替えの最適化を適用**

c が **0** または **1** に初期化されると、値はコンパイル時に解決されるため、新しいバージョンは生成されません。演習として確認してください。

ループ分割

コンパイラーによるループ最適化の別の例として、ループ分割があります。ループ本体が大きく、独立した文が含まれる場合、コンパイラーはループを複数のループに分割できます。**#pragma distribute point** を使用して、この最適化を強制できます。このプラグマがループの内部に配置されると、コンパイラーはコードのその位置でループを分割しようとします。コンパイラーは、ループ本体のすべての依存関係を無視します。**図 5** に、ループ分割のコードサンプルを示します。

```
18: for (int i = 0; i < 1000; i ++) {
19:a[i] = a[i] + i; b[i] = b[i] + i;
20:   c[i] = c[i] + i;     d[i] = d[i] + i;
21: #pragma distribute point
22:   e[i] = e[i] + i;  f[i] = f[i] + i;
23:   g[i] = g[i] + i;  h[i] = h[i] + i;
24: }
```

5 **ループ本体にループ分割の最適化を適用**

コンパイラー・レポートでは、ループが 2 つのループに分割され、ループの詳細が両方のチャンクに含まれることがわかります。

```
LOOP BEGIN at
C:\Users\Documents\Optimization_Reports\loop_distribution.cpp(18,2)
<Distributed chunk1>
   remark #25426: Loop Distributed (2 way)
   remark #25428: Distributed for large ii at line 18
```

ループのマルチバージョニング

ループのマルチバージョニングは、ベクトル化とスカラーによる結果の不一致が疑われる場合にコンパイラーが実行できる最適化です。コンパイラーは、スカラーループとベクトル化されたループを生成し、ランタイムループを挿入して、ベクトル化されたループの結果の動作確認を行います。コンパイラーは、両方のループの結果が同じで、ベクトル化によるパフォーマンスの向上を予測できる場合にのみ、ループをベクトル化します。プログラマーは、潜在的な不一致の問題を解決する (例えば、エイリアシングの問題や依存関係を削除する) か、ループを強制的にベクトル化することにより、このマルチバージョニングを抑制できます。これらの状況では、マルチバージョニングの最適化は適用されません。

```
15:    for(int i = 0; i < 1000; i++){
16:      for(int j = 0; j < 1000; j++){
17:        B[i][j] = C[i][j] + D[i][j];
18:        E[i][j] = B[i][j-1] * 2.0;
19:      }
20:    }
```

6 **ループ本体にループのマルチバージョニングの最適化を適用**

図 6 のコードでは、データの依存関係が仮定されるため、コンパイラーは、コンパイラー・レポートで示されているように、ループのマルチバージョニングの最適化を行います。

```
LOOP BEGIN at
C:\Users\Documents\Optimization_Reports\loop_multiversioning.cpp(15,2)
<Multiversioned v1>
   remark #25420: Collapsed with loop at line 16
   remark #25228: Loop multiversioned for Data Dependence
```

関数のインライン展開

最近のコンパイラーでは、関数のインライン展開による最適化も行われます。例えば、数学関数 **abs()**、**sin()**、**cos()** は、コンパイラーによりインライン展開されます。これらの関数呼び出しを含むループはベクトル化できないため、コンパイラーはこれらの関数呼び出しをインライン展開に置き換えます。この最適化は、最適化レポートにも示されます (下記を参照)。

```
-> INLINE (MANUAL): (18,5) abs(double) (isz = 0) (sz = 7)
-> INLINE (MANUAL): (18,5) sin<int, void>(int) (isz = 2) (sz = 9)
-> INLINE (MANUAL): (18,5) cos<int, void>(int) (isz = 2) (sz = 9)
```

ここで、**sz** はルーチンのサイズ、**isz** はインライン展開されたルーチンのサイズです。**isz** は、インライン展開によりコードが肥大化することを示します。ルーチンのサイズが小さいほど、インライン展開される可能性が高くなります。

コード生成、高レベル、およびループの最適化

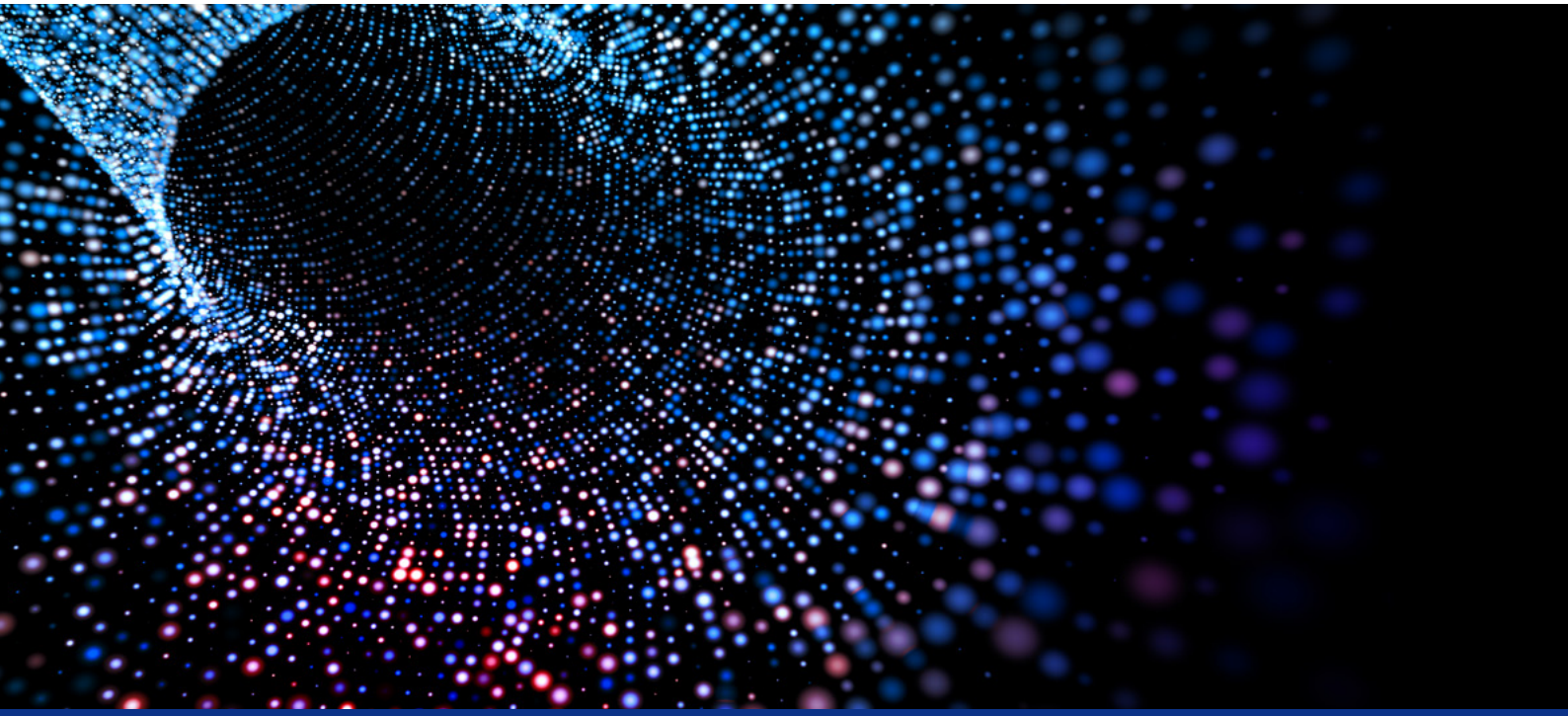
この記事では、C/C++ コンパイラーにより行われるループの最適化について、最適化レポートを使用して説明しました。命令スケジューリング、レジスター割り当て、データフローの解析とチューニングなどのコード生成の最適化は、この記事では取り上げていません。コンパイラーの最適化レポートを調べて、パフォーマンスを向上するさまざまな推奨事項を試してみてください。

参考資料

1. **インテル® C++ コンパイラー 19.1 デベロッパー・ガイドおよびリファレンス (英語)**
2. **インテル® C++ コンパイラー**
3. Engineering a Compiler by Keith Cooper and Linda Torczon.

oneAPI で複数のアーキテクチャーにわたる
多様なワークロードに妥協のないパフォーマンスを提供

詳細 >



HPC アプリケーションの計算と通信のオーバーラップ

レイテンシーを隠蔽して MPI パフォーマンスを向上

Fabio Baruffa 博士 インテル コーポレーション シニア・ソフトウェア・アプリケーション・エンジニア

大規模な HPC システムで MPI アプリケーションをスケーリングするには、効率的な通信パターンが必要です。可能な限り、アプリケーションは通信と計算をオーバーラップさせて通信レイテンシーを隠蔽する必要があります。以前の記事で、非ブロッキング通信と I/O 集合操作をスピードアップするため**インテル® MPI ライブラリー**で提供される非ブロッキング集合通信を実装する方法を述べました (**MPI-3 の非ブロッキング集合操作による通信レイテンシーの隠蔽**および**インテル® MPI ライブラリーの MPI-3 非ブロッキング I/O 集合操作**を参照)。これらのスピードアップは、

インテル® MPI ライブラリーで通信と計算の並列実行を可能にする**非同期処理スレッドのサポート** (英語) によりもたらされます。この記事では、非ブロッキングのポイントツーポイント通信、主に **MPI_Isend** 関数と **MPI_Irecv** 関数に注目して、ドメイン分割コードの効率的なゴーストセル交換メカニズムにより MPI アプリケーションの通信レイテンシーを効率良く隠蔽する方法を説明します。

ポイントツーポイント・オーバーラップのベンチマーク測定

通信と計算をオーバーラップするインテル® MPI ライブラリーの機能をテストするため、単純なピンポン・ベンチマークを実装しました。計算はスリープ関数によりシミュレートされ、通信は非ブロッキング送信 (**Isend**) および受信 (**Irecv**) を使用します。これは、**一般向け非同期 MPI** (英語) で Wittmann ほかにより使用されたアプローチに似ています。

最初のタスクが **MPI_Isend** により通信を初期化して、時間 T_{cpu} の間スリープして計算をシミュレートし、そして **MPI_Wait** をポストする一方で、2 番目のタスクは **MPI_Irecv** をポストするようにベンチマークを設定します。十分な統計データを収集するため、この操作を複数回実行します (このケースでは、100 回反復しています)。以下にコードを示します。

```
while(t_delay <= max_delay) {
   start_time = MPI_Wtime();
      for(i=0; i<ITERATIONS; ++i) {
       if(my_rank ==0 ) {
         MPI_Isend (send_buf, ..., &send_req);
            Sleep (t_delay);
         MPI_Wait(&send_req, &stat);
        } else {
         MPI_Irecv (recv_buf, …, &recv_req);
         MPI_Wait(&recv_req, &stat);
        }
      }
   stop_time = MPI_Wtime();
   t_delay += delay_step;
}
```

ベンチマークの合計時間 T_t と計算部分の時間 T_{cpu} を測定します。非同期処理でない場合、合計時間 $T_t = T_{cpu} + T_c$ で、T_c は通信時間です。非同期通信が有効な場合、ベンチマーク結果の合計時間は $T_t = \max(T_{cpu}, T_c)$ となり、計算と通信のオーバーラップが示されます。

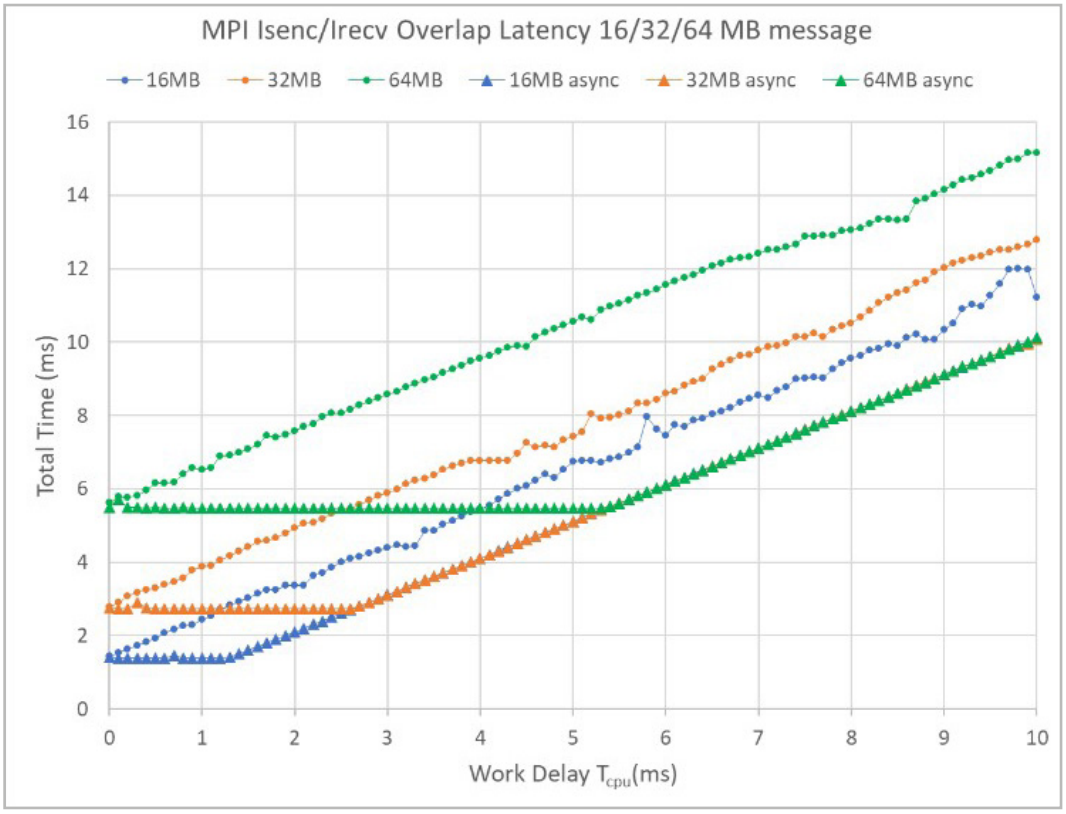
インテル® MPI ライブラリーの **release_mt** バージョンを使用して非同期処理スレッド機能を有効にするには、次の環境変数を設定する必要があります。

```
$ export I_MPI_ASYNC_PROGRESS=1
```

(詳細は、『**インテル® MPI ライブラリー・デベロッパー・ガイド (Linux* 版)**』(英語) および『**インテル® MPI ライブラリー・デベロッパー・リファレンス (Linux* 版)**』(英語) を参照。)

デフォルトでは、MPI タスクごとに 1 つの処理スレッドが生成されます。

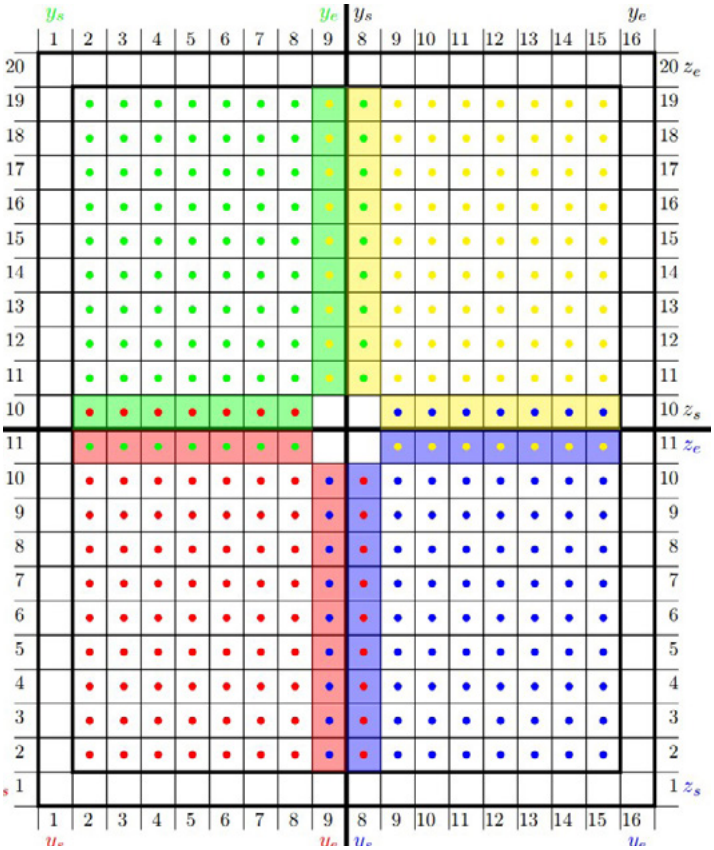
図 1 は、ピンポン・ベンチマークの結果です。16MB (青の線) から 64MB (緑の線) まで、**Isend**/**Irecv** 通信の異なるメッセージサイズを色分けしています。MPI 通信のプロトコルにより、ブロッキング通信でも、小さなサイズのメッセージが送られる可能性があります。同じ色の 2 つの線は、非同期処理が有効な場合と有効でない場合に対応しています (三角のマーカーと円のマーカー)。通信と計算のオーバーラップの効果は、16MB async、32MB async、64MB async の線から明確に確認できます。水平部分は、合計時間が通信オーバーヘッドで占められていることを示しています。実際、計算時間はこの部分には全く含まれていません。一方、(x 軸で表される) 計算時間が長い場合、合計時間は計算で占められます。通信は完全に隠蔽されており、$T_t = T_{cpu}$ から計算と通信のオーバーラップは 100% になります。



1 **異なるメッセージサイズのオーバーラップ・レイテンシーと動作時間。この実験では、ノードごとに 1 つの MPI プロセスと 2 つのノードを使用。**

ドメイン分割におけるゴーストセル交換

非同期処理スレッドのパフォーマンスの利点を詳しく調べるため、ゴーストセル交換メカニズムを使用する一般的なドメイン分割アプリケーションの動作を調べます。**図 2** は、4 つのブロックに分割された 2 次元ドメインの例です。各ブロックは、異なる MPI プロセス (異なる色で表示) に割り当てられています。色付きのドットを含む白いセルは、各プロセスが個別に計算するセルです。各サブドメインの境界の色付きのセルは、隣接する MPI プロセスと交換するゴーストセルです。



2 **2 次元ドメイン分割の例**

ベンチマーク・アプリケーションは、各方向で 1 要素のステンシルの 2D ドメイン分割を使用して、3 次元の拡散熱方程式を解きます。一般的な更新スキームは、次のように要約できます。

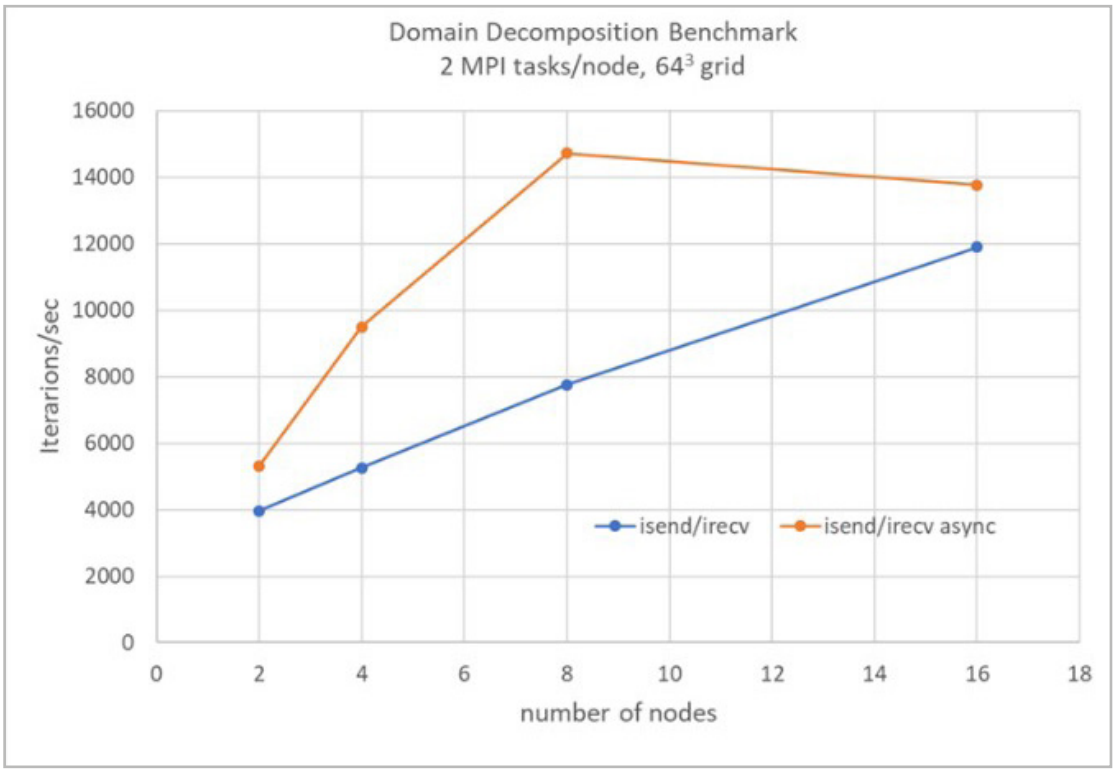
- ゴーストセルのデータを送信バッファーにコピーします。
- `Isend`/`Irecv` 呼び出しによりハロー交換を行います。
- 計算 (パート 1): ドメインの内部フィールドを更新します。

- **MPI_Waitall**
- 受信バッファーのデータをゴーストセルにコピーします。
- 計算 (パート 2): ハローセルを更新します。
- N 回繰り返します。

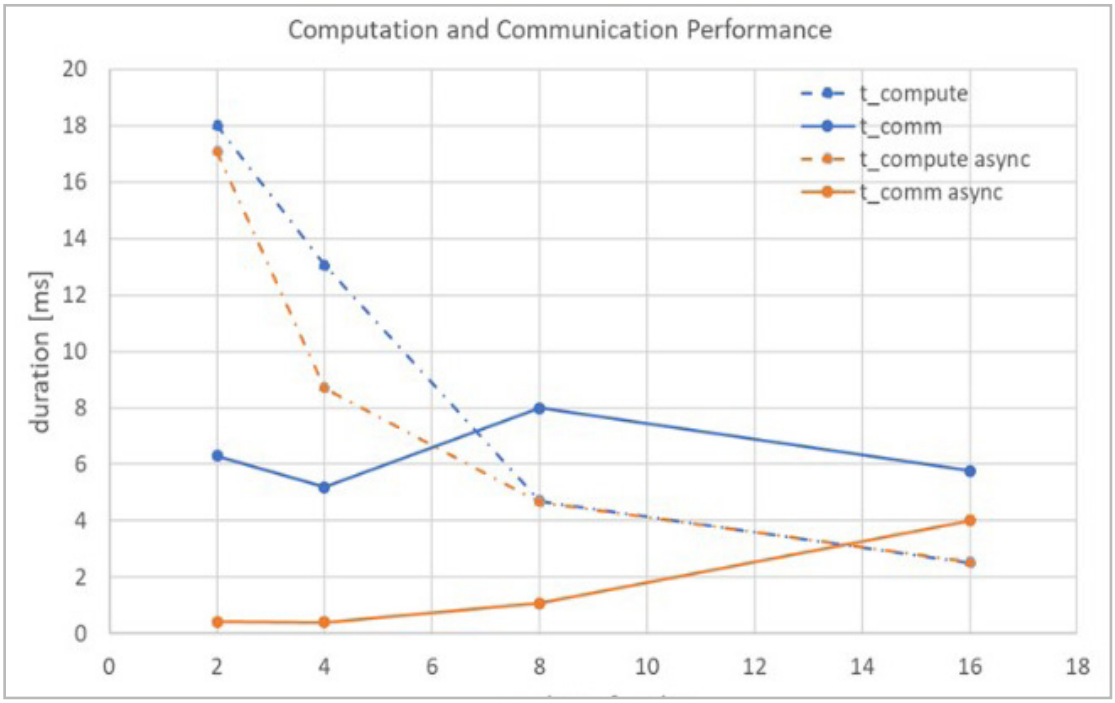
計算は、シミュレーションの実行に使用される MPI プロセスの数によるストロング・スケーリングを示します。コピーステップは、交換に使用するデータをバッファーに格納するために必要です。シミュレーション時間の点からは、コピーのオーバーヘッドは無視できます。ハロー交換は、ゴーストセルの通信をインスタンス化するために使用されます。内部フィールドの計算中は使用されません。そのため、ハロー交換に関しては内部フィールドの更新を非同期で実行できます。分かりやすいように、ドメイン外部の境界は無視しています。

ストロング・スケーリング・ベンチマーク

マルチノード・システムで、リソースの増加とともに 1 秒あたりの反復数がどのように変化するか測定しました。各ノードは、2 ソケットの**インテル® Xeon® Gold 6148 プロセッサー**・ベースのシステムです。**図 3** は、問題サイズを 64 x 64 x 64 グリッドに固定してノードの数を増やしたストロング・スケーリングの結果です。青の線とオレンジの線の差は、インテル® MPI ライブラリーの非同期処理スレッド実装を使用して通信と計算をオーバーラップした場合のパフォーマンスの向上を示しています。8 ノードで最大 1.9 倍パフォーマンスが向上しました。この結果は計算時間と通信時間から確認できます (**図 4**)。青の線は標準 MPI に対応し、オレンジの線は非同期処理スレッド実装に対応しています。計算時間と通信時間は、それぞれ破線と実線で表されています。ノードの数が増えると、MPI プロセスごとの計算が小さくなり、アプリケーションの通信オーバーヘッドが大きくなるため、メリットはそれほどありません。



3 **1 秒あたりの計算された総反復数とシステムのノード数。数値が高いほうが良い。グリッドサイズ : 64 x 64 x 64。ノードごとに 2 つの MPI プロセス。**



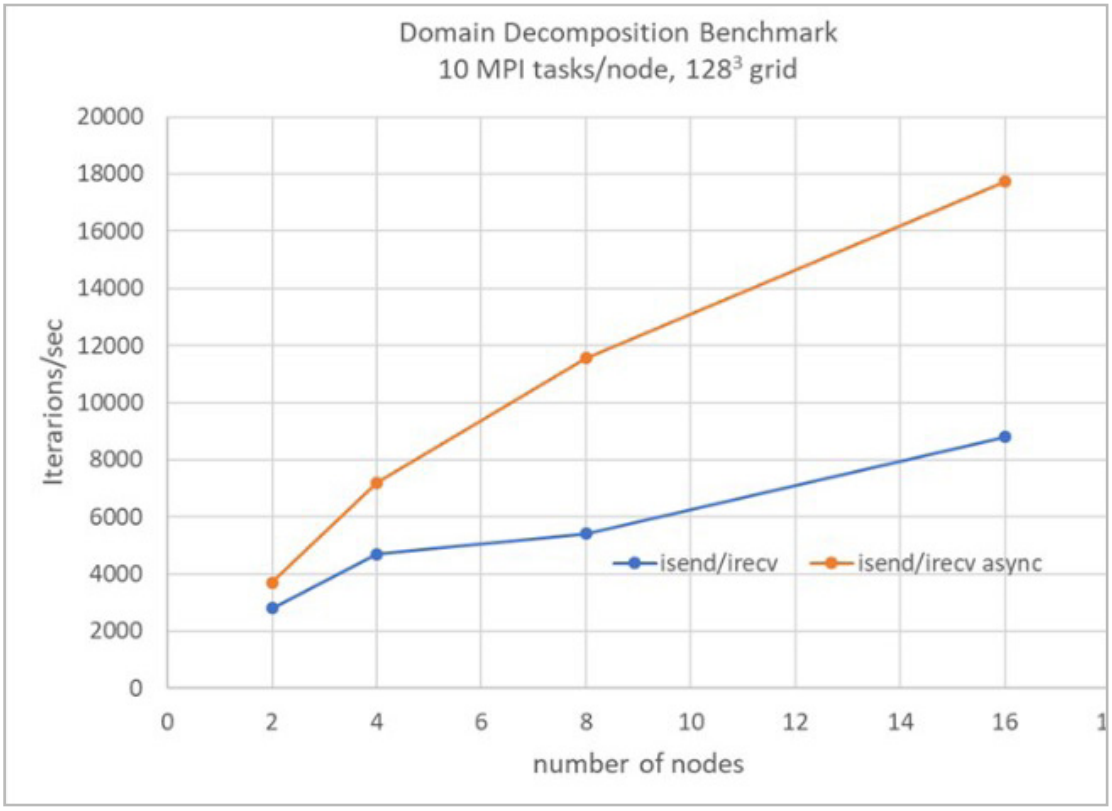
4 **計算時間と通信時間の測定結果。数値が低いほうが良い。グリッドサイズ : 64 x 64 x 64。ノードごとに 2 つの MPI プロセス。**

ノードごとの MPI プロセスの数を増やす

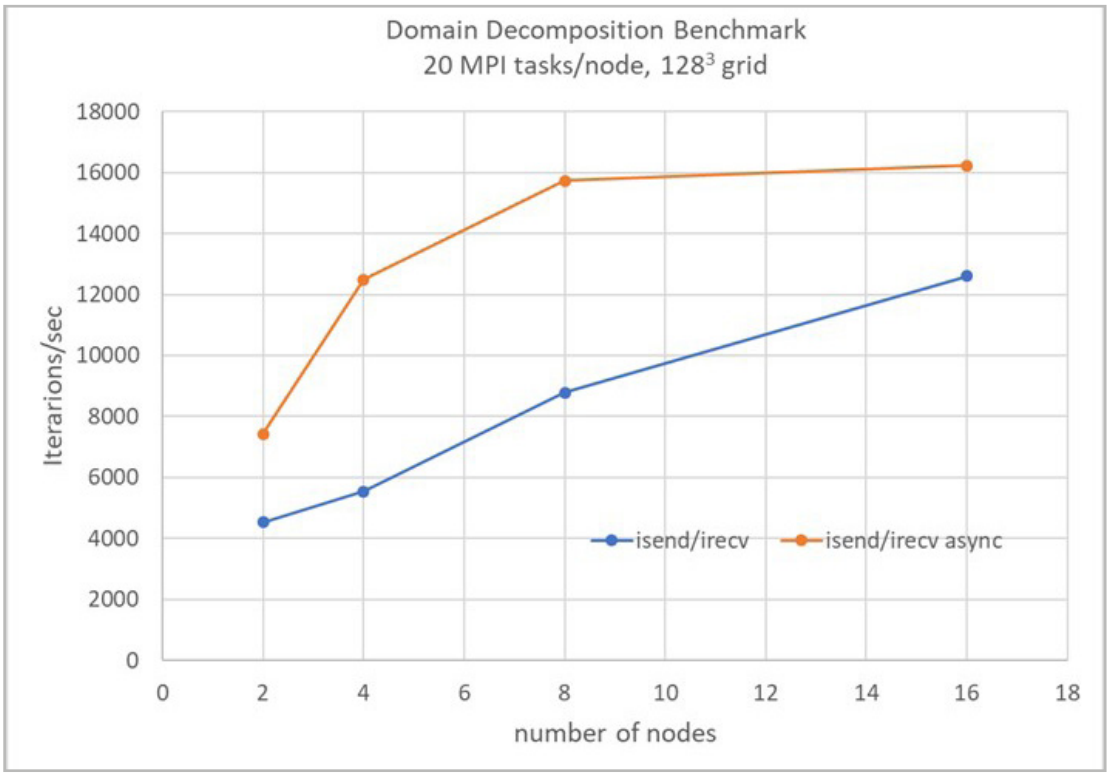
ノードごとの MPI プロセスの数が多い場合は、ゴーストセルの通信とオーバーラップする十分な計算を行うため、グリッドサイズを 128 x 128 x 128 に増やします。**図 5** と**図 6** は、1 秒あたりの反復数と (ノードごとの MPI プロセスの数が 10 および 20 の場合の) ノード数を示しています。MPI プロセスの数が多い場合、利用可能なリソースのバランスをとる必要があります。インテル® MPI ライブラリーでは、次の環境変数を使用して、MPI プロセスと非同期処理スレッドを特定の CPU コアに設定できます。

```
$ export I_MPI_PIN_PROCESSOR_LIST=<list of cores>
$ export I_MPI_ASYNC_PROGRESS_PIN=<list of cores>
```

これらの環境変数を使用して、対応する MPI プロセスの近くに非同期スレッドを設定すると、最良のパフォーマンスが得られます。どちらの場合もスピードアップし、ノードごとに 10 の MPI プロセス、16 ノードでは、最大 2.2 倍スピードアップしました。



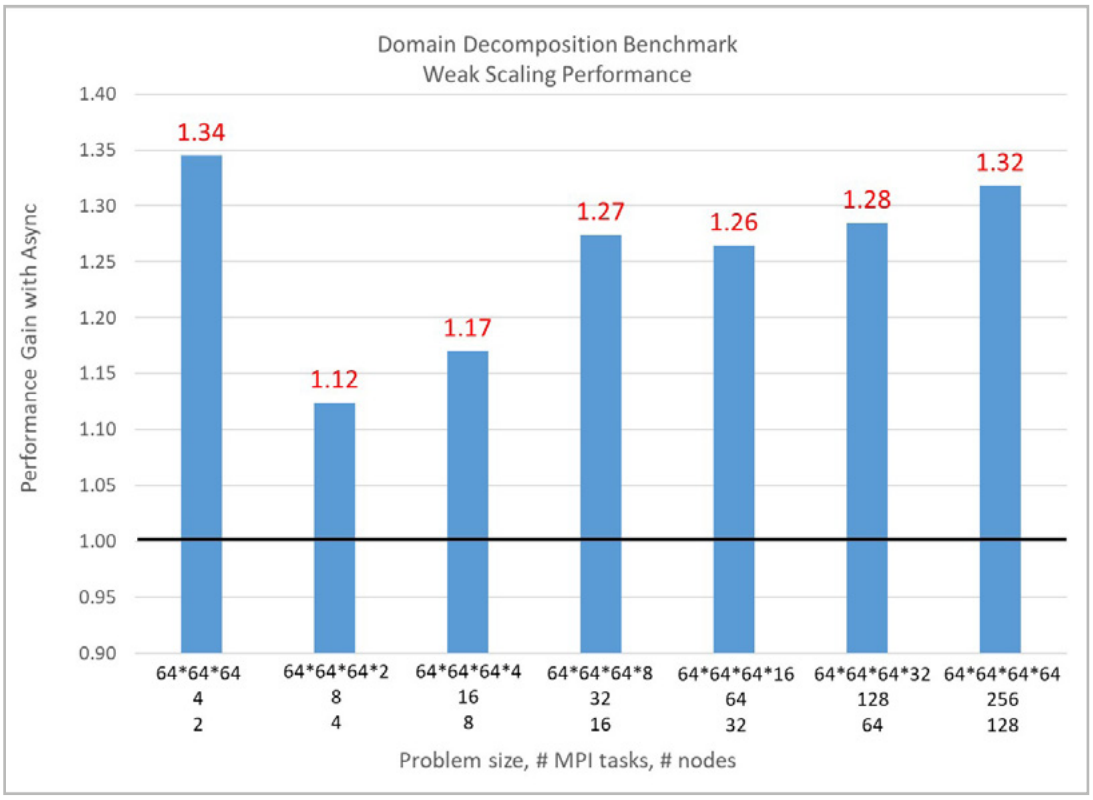
5 **1 秒あたりの計算された総反復数とシステムのノード数。数値が高いほうが良い。グリッドサイズ : 128 x 128 x 128。ノードごとに 10 の MPI プロセス。リソースのバランスをとるため、export I_MPI_PIN_PROCESSOR_LIST=1,5,9,13,17,21,25,29,33,37 および export I_MPI_ASYNC_PROGRESS_PIN=0,4,8,12,16,20,24,28,32,36 に設定し、対応する MPI プロセスの近くに非同期スレッドを設定。**



6 **1 秒あたりの計算された総反復数とシステムのノード数。数値が高いほうが良い。グリッドサイズ : 128 x 128 x 128。ノードごとに 20 の MPI プロセス。リソースのバランスをとるため、export I_MPI_PIN_PROCESSOR_LIST=1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39 および export I_MPI_ASYNC_PROGRESS_PIN=0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38 に設定し、対応する MPI プロセスの近くに非同期スレッドを設定。**

ウィーク・スケーリング・ベンチマーク

最後のパフォーマンス・テストでは、ウィーク・スケーリングの調査を行いました。**図 7** は、通信と計算をオーバーラップしたパフォーマンス向上の結果です。グリッドサイズを 2 倍にして、ノード数を最大 128 まで増やしています。計算サイズとノードごとの MPI プロセスの数は 4 つのプロセスで一定です。パフォーマンスは全体的に大きく向上し、最も大きなテストで最大 32% パフォーマンスが向上しました。



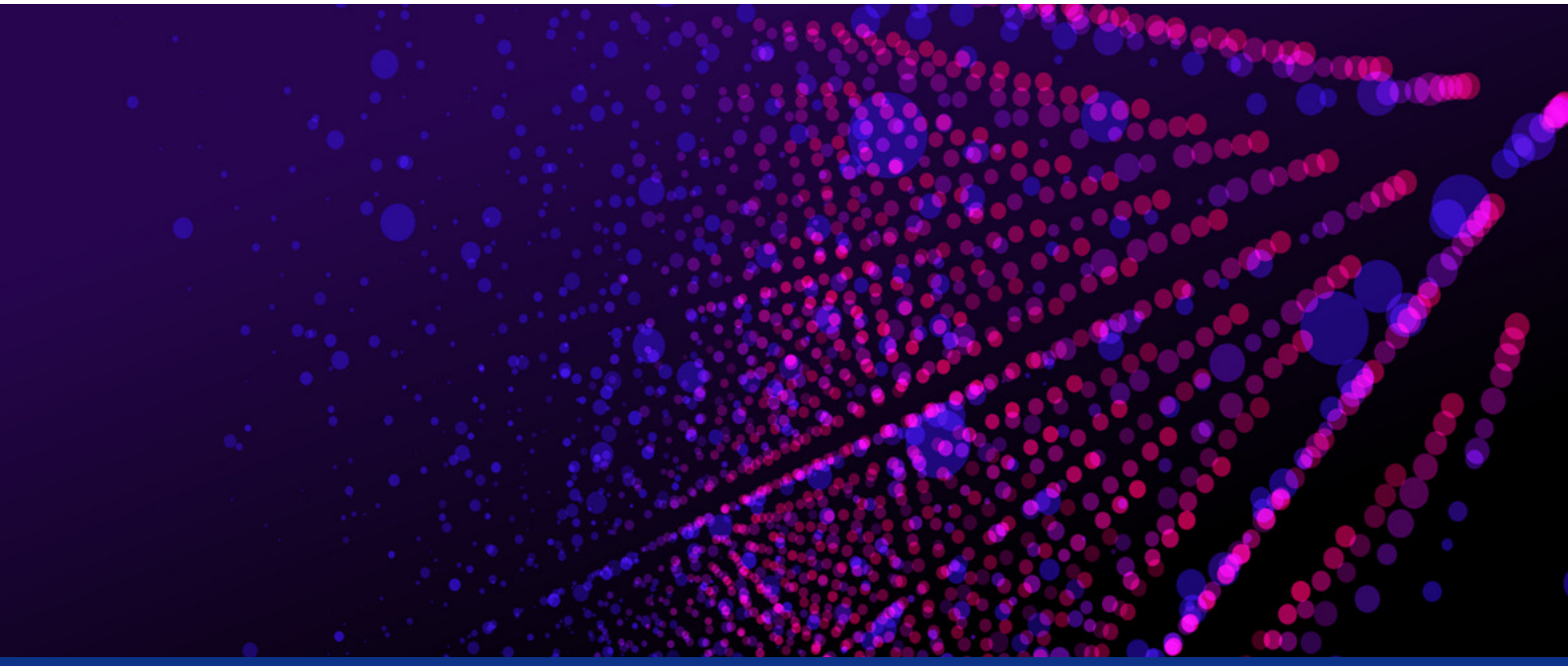
7 **ウィーク・スケーリング・ベンチマークのパフォーマンスの向上。各バーは非同期処理スレッドを使用した場合のスピードアップを示す。数値が高いほうが良い。**

速度とスケーリングの向上

この記事では、インテル® MPI ライブラリーが提供する非同期処理スレッド機能を使用して、非ブロッキングのポイントツーポイント MPI 関数の通信と計算のオーバーラップの可能性を調査および解析しました。ゴーストセル交換メカニズムを使用する一般的なドメイン分割コードで、この実装を使用するパフォーマンスの利点を示しました。非ブロッキング **`Isend`**/**`Irecv`** スキームを使用した同様の通信パターンを含むアプリケーションでは、同様のスピードアップとスケーリング動作が得られると考えています。

Configuration: Testing by Intel as of Feb. 14, 2020. Node configuration: 2x Intel® Xeon® Gold 6148 Processor CPU @ 2.40GHz, 20 cores per CPU, 96 GB per node, nodes connected by the Intel® Omni-Path Fabric (Intel® OP Fabric), Intel® MPI Library 2019 update 4, release_mt, asynchronous progress control enabled via export I_MPI_ASYNC_PROGRESS=1, export I_MPI_ASYNC_PROGRESS_THREADS=1. Intel' s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.


intel
Software
将来に向けた
取り組み
技術的なスキルアップ、
エキスパートからの回答、
新しい開発分野への挑戦。
これらすべてにインテルの
無料の技術ウェビナー
(オンデマンド) が役立ちます。
トピックを選択 (英語) >
コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation またはその子会社の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
© Intel Corporation



インテル® MPI ライブラリーのチューニングを使用した HPC クラスターの効率化

コードを変更しないでパフォーマンスを向上

Dr. Amarpal Singh Kapoor インテル コーポレーション テクニカル・コンサルティング・エンジニア
Marat Shamshetdinov インテル コーポレーション ソフトウェア開発エンジニア

HPC クラスターの調達には、パフォーマンスと経済的な考慮が必要になります。しかし、調達すれば終わりではありません。HPC クラスターを効率良く運用することも、環境および経済的な理由から同じように重要であり、クラスターのライフサイクルを通して続く継続的なプロセスです。

クラスターの運用効率を向上するにはさまざまな方法がありますが、この記事では、**インテル® MPI ライブラリー**で利用可能なチューニング・ユーティリティーを使用してコストを削減する方法を取り上げます。これらのユーティリティーは、設定をチューニングして、クラスターで MPI ジョブの解決にかかる時間を短縮するのに役立ちます。クラスター全体およびアプリケーション固有のチューニングを行うことができます。これらのユーティリティーを使用するためソースコードを変更する必要はありません。これらのユーティリティーはすべて、インテル® MPI ライブラリーから利用できます。

ここでは、4 つのチューニング・ユーティリティーについて簡単に説明します。

- MPItune
- Fast Tuner
- Autotuner
- **`mpitune_fast`**

中でも、Autotuner と次の点に注目します。

- 自動チューニングの**推奨手順**
- **インテル® VTune™ プロファイラー**のアプリケーション・パフォーマンス・スナップショット (APS) を使用して**候補のアプリケーションを選択する手法**
- **インテル® MPI Benchmarks (IMB)** (英語) の**パフォーマンス向上の結果**

今後の記事では、この調査を実際の HPC アプリケーションに拡張する予定です。

バージョン情報

この記事では、2 つのインテル ® ソフトウェア・ツールを使用します。

1. インテル® MPI ライブラリー
2. インテル® VTune™ プロファイラー

これらのツールはどちらも**インテル® Parallel Studio XE** および**インテル® oneAPI ツールキット**の一部として利用できます (**表 1**)。

表 1. この記事で取り上げたツールを含むパッケージ

ツール	インテル® MPI ライブラリー	インテル® VTune™ プロファイラー
インテル® Parallel Studio XE Professional Edition	–	√
インテル® Parallel Studio XE Cluster Edition	√	√
インテル® oneAPI ベース・ツールキット	–	√
インテル® oneAPI HPC ツールキット	√	–

この記事では、**インテル® oneAPI ベース・ツールキット** (英語) および**インテル® oneAPI HPC ツールキット** (英語) を使用しました (記事執筆時点の最新バージョンは 2021.1.0 beta07 です)。ここで取り上げたチューニング・ユーティリティーは、インテル® Parallel Studio XE のユーザーが利用できるものと同等です。すべてのテストは、**インテル® oneAPI DevCloud** (英語) の**インテル® Xeon® Gold 6128 プロセッサー** (デュアルソケット、ソケットあたり 12 コア、インテル® ハイパースレッディング・テクノロジー有効) を搭載したイーサネット接続ベースのクラスターを使用して行いました。

チューニング・ユーティリティー

インテル® MPI ライブラリーの開発チームは、ノード内およびノード間でメッセージを転送しつつ、クラス最高のレイテンシーと帯域幅を維持するためチューニングに多くの労力を費やしています。そのため、インテル® MPI ライブラリーはそのままでも優れていますが、次のように、さらにチューニングを行うことでメリットが得られるケースもあります。

- テストされていないランク数 (**-n**) とノードあたりのランク数 (**-ppn**) の組み合わせ
- 2 の累乗でないメッセージサイズ
- 新しいネットワーク・トポロジー
- テストされていないインターコネクト

これらのケースでは、追加のチューニングを行うことでパフォーマンスを向上できます。クラスターで最も時間を費やしているアプリケーションについて、そのコードを変更することなくパフォーマンスの向上を達成することは、それがわずかな向上であっても、サービスのライフサイクル全体で見れば、大幅なコスト削減、負荷の削減、ジョブのスループットの向上につながります。

MPItune と Fast Tuner は、2018 年以前からインテル® MPI ライブラリーに含まれています。MPItune には大きなパラメーター空間があり、次のような変数の最適値を探します。

- **I_MPI_ADJUST_<opname>** ファミリー
- **I_MPI_WAIT_MODE**
- **I_MPI_DYNAMIC_CONNECTION**
- **I_MPI_*_RADIX** 変数
- その他のオプション、ファブリック、合計ランク、ノードごとのランクの異なる値

チューニング手法は O(**n**) で、**n** は可能なテスト構成の数です。このアプローチにはコストがかかります。MPItune は、(クラスター管理者による) クラスター全体のチューニングと (非特権ユーザーによる) アプリケーション固有のチューニングの両方に使用できます。

Fast Tuner は、MPItune の大きなオーバーヘッドを克服します。MPItune に基づいて、Fast Tuner は、アプリケーションの初期プロファイルを生成してターゲット MPI 関数のリストを作成することから開始します。この後に、Fast Tuner が実際のアプリケーションを再度起動することはありません。代わりに、チューニングされた構成を生成する代用として IMB を使用し、チューニングのオーバーヘッドを軽減します。

インテル® MPI ライブラリーの 2019 バージョンでは、さらに 2 つのチューニング・ユーティリティーが追加されました。

1. **Autotuner**
2. **mpitune_fast**

Autotuner は、チューニング範囲を環境変数の **I_MPI_ADJUST_<opname>** に限定して、チューニングを現在の構成のみ (ファブリック、ランク数、ノードあたりのランク数など) に制限します。Autotuner は対象のアプリケーションを実行します。Fast Tuner のように代用ベンチマークに依存する必要はありません。Autotuner は、アプリケーションで同じサイズの異なるコミュニケーターについて独立したチューニング・データを生成することもできます。Autotuner の主な特徴は、オーバーヘッドのないスマートな選択ロジックです。ほかのユーティリティーと異なり、個別にチューニングを実行する必要はありませんが、場合によっては、オプションで実行してパフォーマンスを向上することもできます。

最後のユーティリティー、**mpitune_fast** を使用すると、インテル® MPI ライブラリーを自動的に設定して、自動チューニングを有効にして起動し、クラスター・チューニング向けに構成できます。このツールは、適切な Autotuner 環境で IMB を繰り返し起動し、クラスター構成のチューニング・パラメーターを含むファイルを生成します。チューニング・ファイルを生成した後、**I_MPI_TUNING_BIN** 環境変数を使用して、インテル® MPI ライブラリーを使用するクラスターのすべてのアプリケーションのデフォルトとして設定できます。**mpitune_fast** は、Slurm および LSF ワークロード・マネージャーをサポートしています。ジョブが割り当てられたホストを自動的に定義して、起動を行います。**mpitune_fast** を起動する最も簡単な方法を次に示します。

```
$ mpitune_fast -f ./hostfile
```

追加のオプションを使用して、チューニングの実行をカスタマイズできます。

```
$ mpitune_fast -ppn 8,4,2,1 -f ./hostfile -c alltoall,allreduce,barrier
```

オプションの一覧を確認するには、**-h** オプションを使用します。

```
$ mpitune_fast -h
```

この記事の残りの部分では、Autotuner について取り上げます。

Autotuner でチューニングするアプリケーションを選択する

このステップはオプションです。アプリケーションのパフォーマンス特性が分かっている場合は、このステップをスキップしてかまいません。分かっていない場合は、高レベルのアプリケーション・メトリックを取得するように設計された低オーバーヘッドのツール、APS を使用します。

APS プロファイルを生成するには、次のコマンドを実行します。

```
$ source /opt/intel/vtune/<version>/apsvars.sh
$ mpiexec.hydra -n X -ppn Y -f hostfile aps ./test1
```

MPI run コマンドでは、アプリケーション名の前に **aps** を指定していることに注意してください。詳細な MPI メトリックを収集するには、別の環境変数を設定する必要があります。

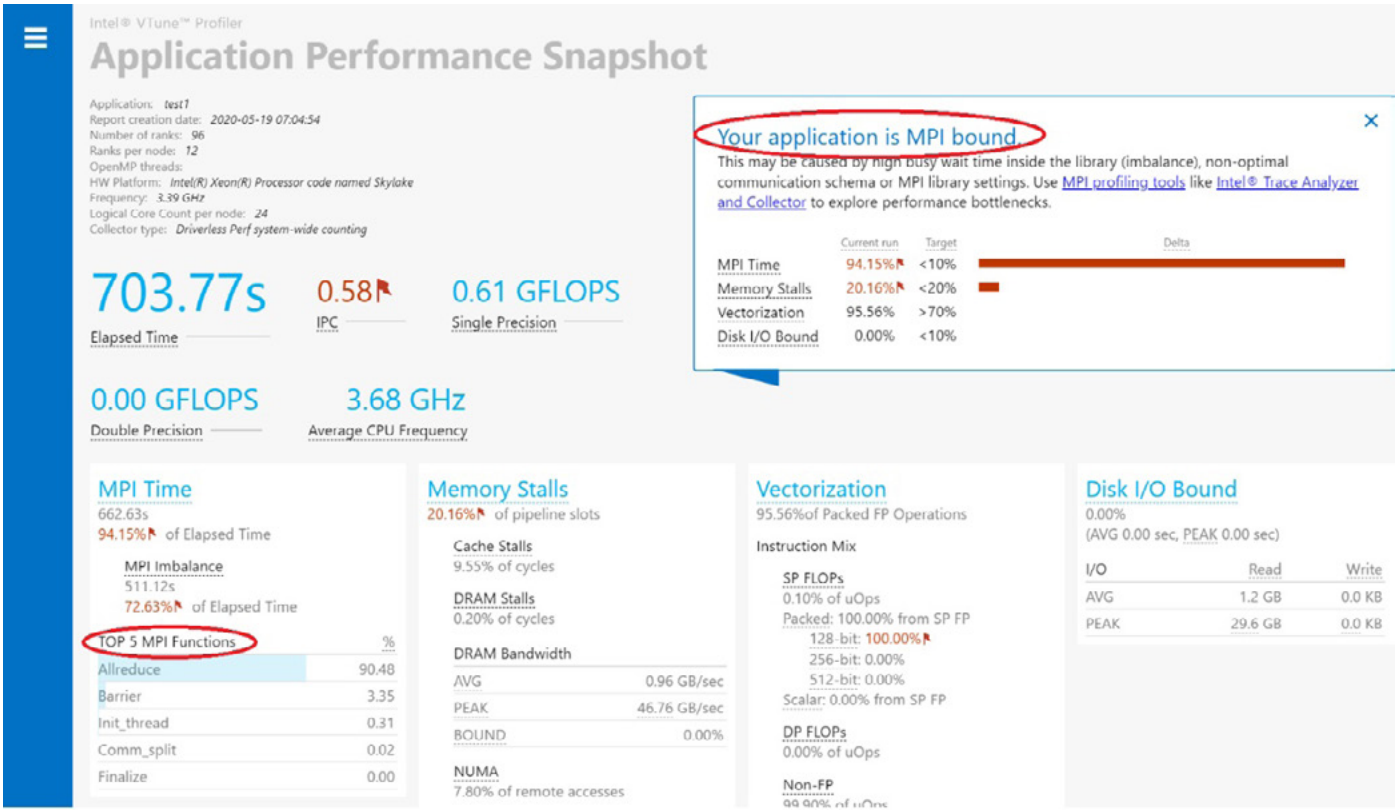
```
$ APS_STAT_LEVEL=5 mpiexec.hydra -n X -ppn Y -f hostfile aps ./test1
```

このコマンドを実行すると、**aps_result_<postfix>** という名前の結果フォルダーが生成されます。

次のコマンドは、画面のテキスト出力に加えて、(HTML 形式の) レポートを生成します。

```
$ aps --report=/path/to/aps_result_<postfix>
```

APS は、アプリケーションが MPI 依存で、Allreduce が最も時間を費やしている MPI 関数であることをレポートしています (**図 1**)。集合通信関数で最も時間が費やされているため、このアプリケーションは Autotuner によるチューニングの良い候補と言えます。非ブロッキング集合操作から生じる待機操作が多くの時間を費やしている状況も適しています。Autotuner は現在、ポイントツーポイント通信のチューニングは行っていません。



1 **APS のレポート**

Autotuner は、コミュニケーター固有のチューニングを行うこともできます。同じサイズでランクの構成が異なる 2 つのコミュニケーターは、Autotuner から個別のチューニング・データを受け取ります。場合によっては、最も時間を費やしているコミュニケーターのみにチューニングを制限することもできます。この場合、APS プロファイルを収集するときに追加の環境変数 (**APS_STAT_LEVEL=3** および **APS_COLLECT_COMM_IDS=1**) を使用してコミュニケーターの ID データを収集します。その後、次のコマンドを使用してコミュニケーターの ID を表示します。

```
$ aps-report aps_result_<postfix> -lFE
```

MPI 呼び出しの数とメッセージサイズは、考慮すべき重要なパラメーターです。特定の MPI 操作、メッセージサイズ、コミュニケーターについて、最適な実装を選択するには (少なくともその MPI 操作で) 利用可能な実装の数と同じ数の MPI 呼び出しが必要です。呼び出し数が増えると、それに応じて累計される全体的なゲインも増えます。フィルターされていない関数のサマリーを出力するには、**-lFE** オプションを **-fF** に変更します。フィルターされていないメッセージサイズのサマリーを出力するには、**-mDPF** に変更します。ほかにもいくつかのオプションを MPI パラメーターとして指定できます (詳細は、『**APS ユーザーガイド** 』(英語) の「Analysis Charts (解析レポート)」セクションを参照)。

Autotuner のステップ

前述したように、Autotuner を使用してアプリケーションをチューニングするにはコードを変更する必要はありません。さらに、この手法はコンパイラーのデバッグ情報に依存しないため、再コンパイルを行うことなく既存のバイナリーを使用できます。Autotuner を使用してアプリケーションをチューニングするには、以下のステップを実行します。

ステップ 1: チューニング・データの生成と使用

このステップは必須で、チューニング・データの生成と使用が含まれます。

```
$ I_MPI_TUNING_MODE=auto:cluster mpiexec.hydra -n X -ppn Y -f hostfile ./binary
```

I_MPI_TUNING_BIN_DUMP をユーザー定義のファイル名に設定して、生成されたチューニング・データを (以降の実行で使用する) ファイルに保存することもできます。

```
$ I_MPI_TUNING_MODE=auto:cluster I_MPI_TUNING_BIN_DUMP=tuning.dat
mpiexec.hydra -n X -ppn Y -f hostfile ./binary
```

環境変数で **export** 文を使用していないことに注意してください。これは、特に検証ステップも実行する場合、自動チューニング・ワークフローのコンテキストで適切に動作します。

このステップを実行するだけで、パフォーマンスが向上する可能性があります。このステップでは、チューニング・データの生成とは別に、チューニング・データの適用も行われることに注意してください。

ステップ 2: チューニング・データの検証

このステップはオプションで、ステップ 1 のパフォーマンス・ゲインを向上します。このステップでは、ステップ 1 で (**I_MPI_TUNING_BIN** を使用して) 生成したチューニング・ファイルを使用します。

```
$ I_MPI_TUNING_BIN=tuning.dat mpiexec.hydra -n X -ppn Y -f hostfile
./binary
```

ステップ 2 で **I_MPI_TUNING_MODE=auto:cluster** を有効にすると、追加のオーバーヘッドが発生します。このオーバーヘッドは、誤って **export** 文を使用している場合に発生する可能性があります。これが、ステップ 1 で **export** 文を使用しなかった理由です。Autotuner の制御ロジックをさらに制御するには、**環境変数の I_MPI_TUNING_AUTO_* ファミリー** (英語) を介して追加のノブを使用します。

パフォーマンス・ゲインの実証

このセクションでは、前のセクションの方法を IMB に拡張し、達成したスピードアップを示します。

ベースライン

```
$ mpiexec.hydra -n 96 -ppn 12 -f hostfile ./IMB-MPI1 allreduce -iter
1000 -iter_policy off -npmin 96 -time 100000
```

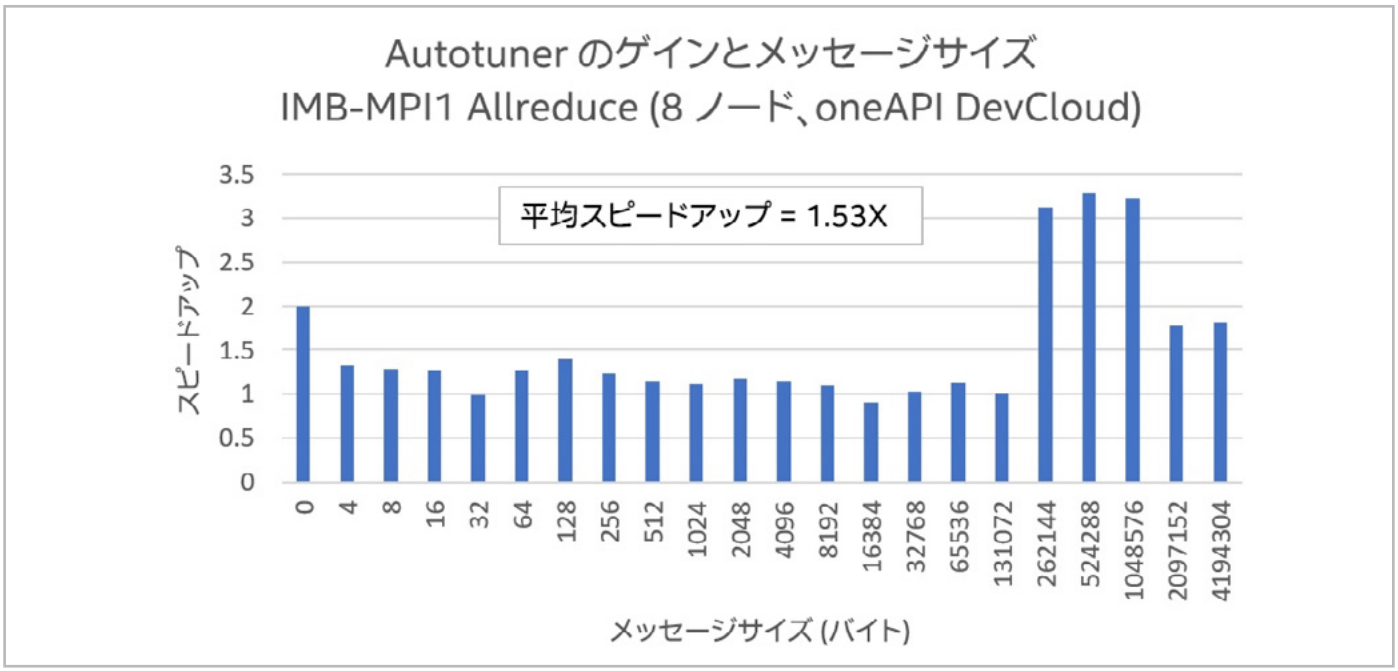
ここでは、**-iter 1000** オプションを使用して、すべてのメッセージサイズの繰り返し回数を 1,000 に設定しています。**-iter_policy off** オプションは、デフォルトの反復ポリシーを無効にして大きなメッセージサイズの繰り返し回数を制限します。**-npmin 96** オプションは MPI ランク数を 96 に制限します (デフォルトでは、**IMB-MPI1** は 2 の累乗の増分で指定されたランク数に達するため、設定を変更しました)。**-time 100000** は、**IMB-MPI1** がタイムアウトする前にサンプルごとに 100,000 秒待機します。

ステップ 1

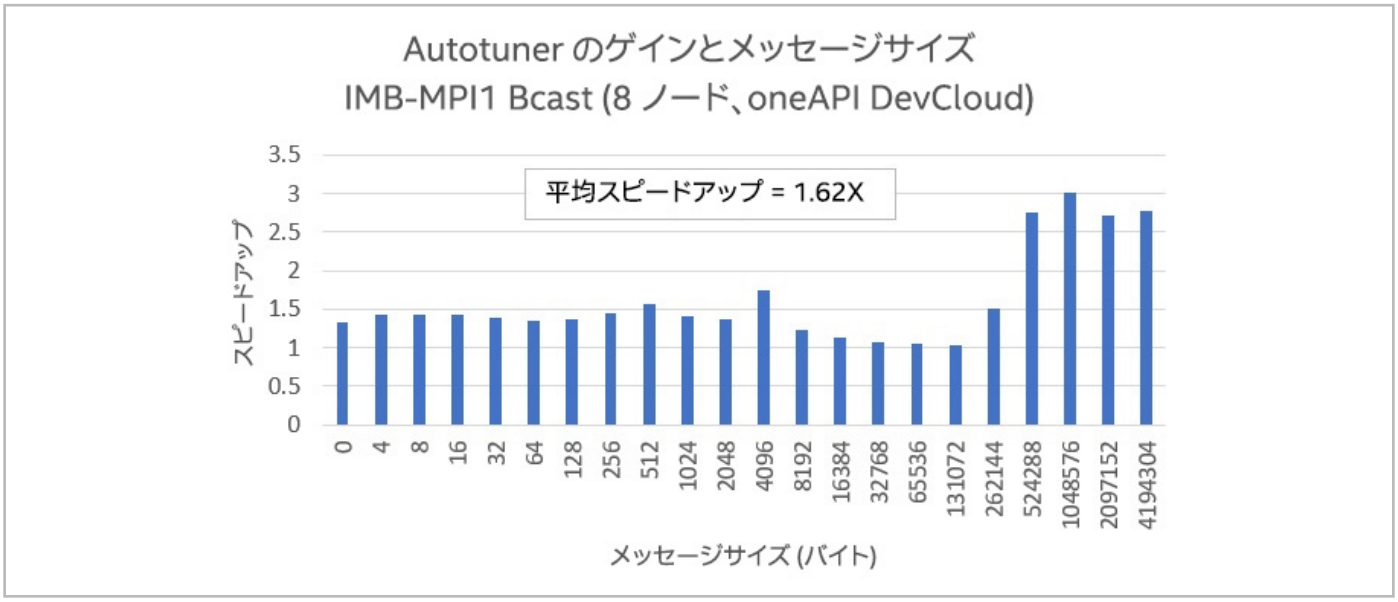
```
$ I_MPI_TUNING_MODE=auto:cluster I_MPI_TUNING_BIN_DUMP=allreduce1.dat
mpiexec.hydra -n 96 -ppn 12 -f hostfile ./IMB-MPI1 allreduce
-iter 1000 -iter_policy off -npmin 96 -time 100000
```

ステップ 2

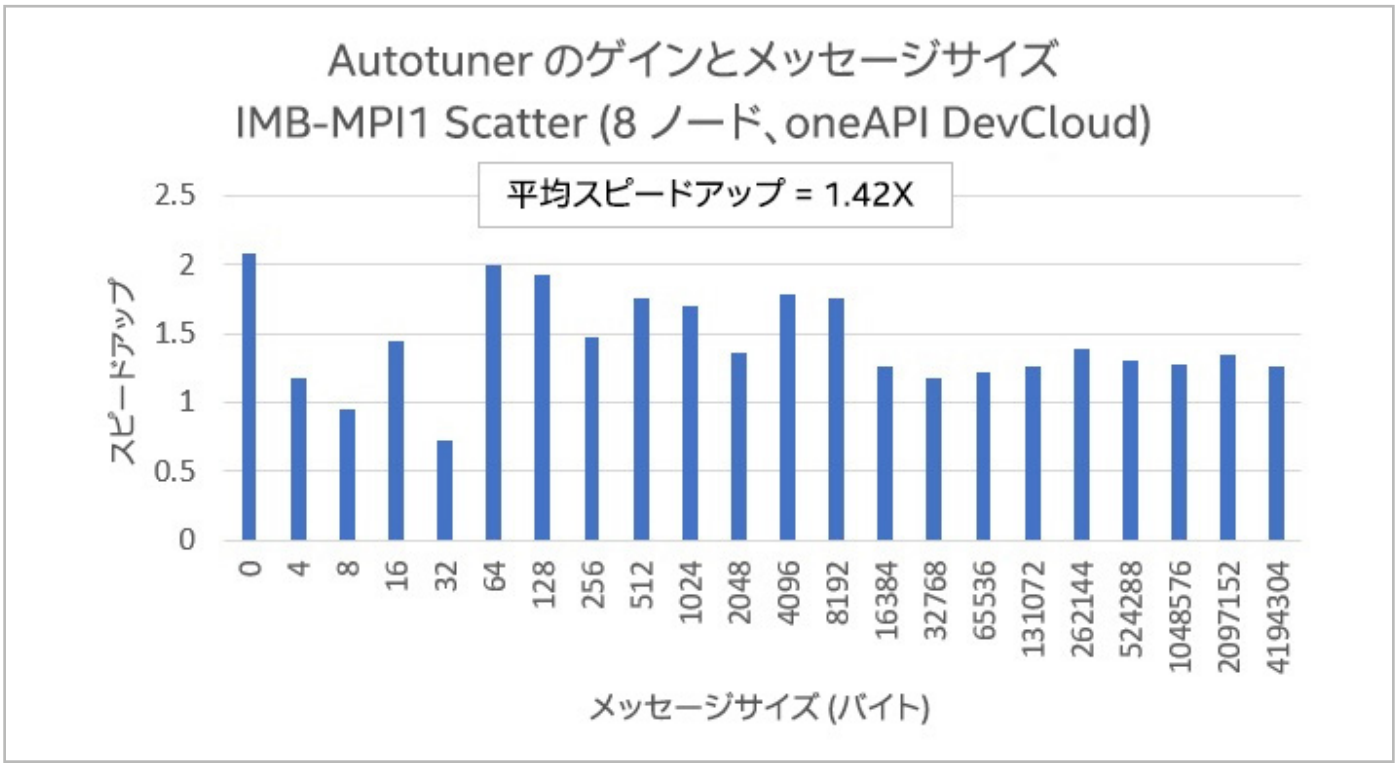
```
$ I_MPI_TUNING_BIN=allreduce1.dat mpiexec.hydra -n 96 -ppn 12 -f
hostfile ./IMB-MPI1 allreduce -iter 1000 -iter_policy off -npmin 96 -
time 100000
```



2 **Allreduce の Autotuner のゲイン**



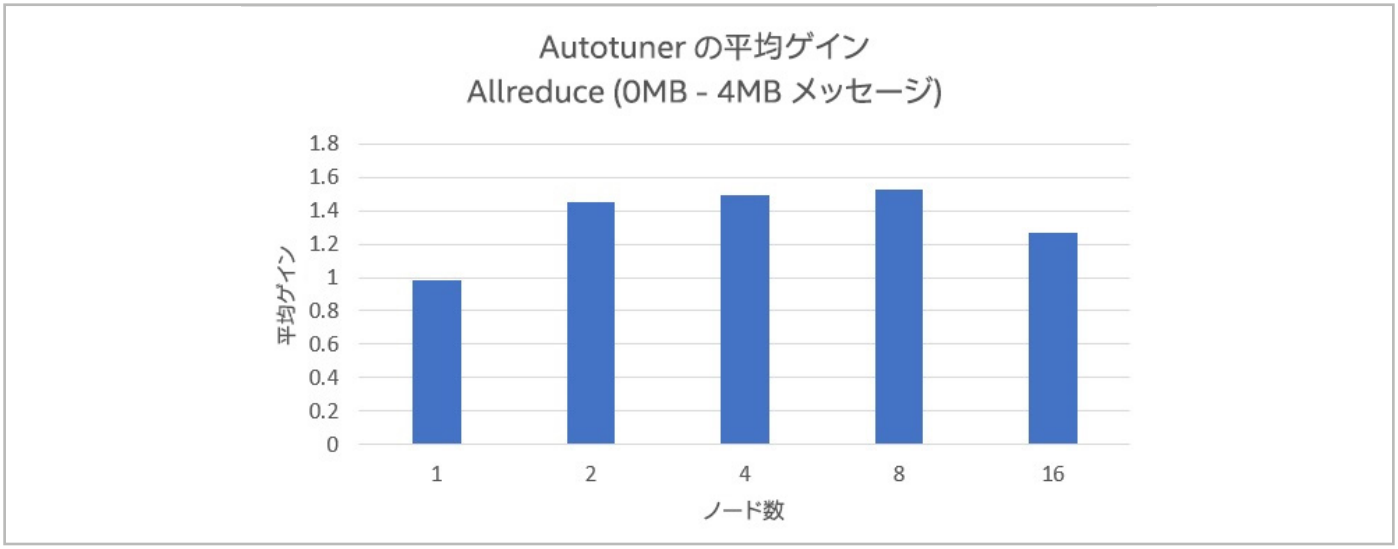
3 **Bcast の Autotuner のゲイン**



4 **Scatter の Autotuner のゲイン**

図 2 から **図 4** に示されているように、多くのメッセージサイズでパフォーマンスが大幅に向上しました。Allreduce、Bcast、Scatter の 0MB から 4MB メッセージサイズの平均スピードアップは、それぞれ 1.53 倍、1.62 倍、1.42 倍でした。スピードアップの計算は、ベースラインとステップ 2 の IMB の平均時間 (`t_avg`) メトリックに基づいて行われました。

図 5 は、さまざまなノード数 (1 から 16) での Allreduce の Autotuner のゲインを示しています。ゲインは持続可能で、0MB から 4MB メッセージサイズの平均スピードアップは 1.34 倍でした。一般に、ノード数が増えると、Autotuner のゲインも増えると予想されます。



5 **Autotuner のゲインとノード数**

コードを変更せずにパフォーマンスを向上

この記事では、インテル® MPI ライブラリー 2019 で利用可能な 4 つのチューニング・ユーティリティーを紹介しました。**表 2** は、5 つの項目についてのこれらのユーティリティーの評価です。緑は良い、オレンジはどちらでもない、赤は悪いを表しています。

表 2. インテル® MPI ライブラリーのチューニング・ユーティリティーと能力

パラメーター/チューニング・ユーティリティー	MPItune	Fast Tuner	Autotuner	mpitune_fast
チューニングのオーバーヘッド				
使いやすさ				
アプリケーション・チューニング				
マイクロベンチマーク・チューニング				
プロダクション環境での採用				

mpitune_fast はクラスター全体のチューニングに適した最初のステップとして機能しますが、アプリケーション・レベルのチューニングには Autotuner を使用します。アプリケーションで Autotuner の恩恵が得られるかどうか理解するには、そのアプリケーションを特徴付けることが重要です。APS はこの機能を果たします。時間の大部分を集合操作に費やしているアプリケーション (ブロッキングと非ブロッキングの両方) で最も恩恵が得られると予想されます。

BLOG HIGHLIGHTS

複数のアーキテクチャーにわたる DPC++ のデータ管理 : oneAPI を使用した実行の順序

データ並列 C++ (DPC++) データ管理ブログのパート 1 では、oneAPI プログラミング・モデルなどの、ハードウェア・アクセラレーターを含むクロスアーキテクチャー・システムをターゲットにする場合のデータの移動と制御について説明しました。統合共有メモリー (USM) とバッファーはどちらも、プログラマーが DPC++ (oneAPI のクロスアーキテクチャー言語) で移植性とパフォーマンスの目標を達成するのに役立ちます。このブログでは、DPC++ カーネルの実行の相対的な順序を詳しく説明します。

この記事の続きはこちら (英語) でご覧になれます。 >

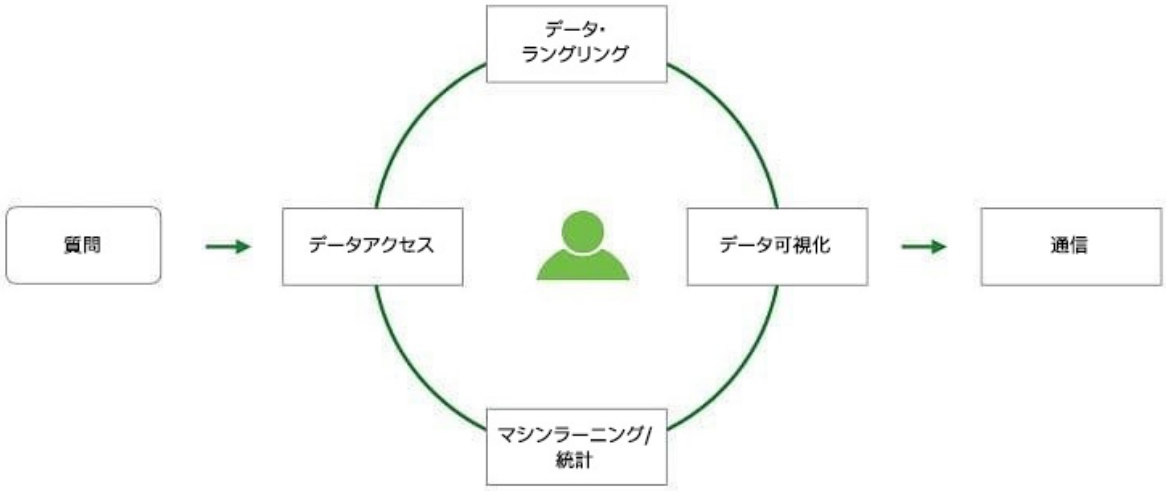
データ・サイエンス・アプリケーションを次のレベルへ

スケーラビリティー、パフォーマンス、柔軟性

Venkat Krishnamurthy OmniSci プロダクト・マネージメント・バイスプレジデント

この 10 年間、テクノロジーの展望は主にデータ自身により推進されてきたことは明らかです (**データと AI の状況 2019** (英語) を参照)。純粋な応用科学であれ産業であれ、人間の試みのあらゆる分野で、問題を解決する主な方法として、我々はデータを収集して使用しています。その結果、データサイエンスが重要な分野として注目を集めるようになりました。増え続けるデータセットから有用な情報を整理して引き出せることは重要なスキルセットであり、データ・サイエンティストがより大きくより詳しいデータを扱えるようにさまざまなツールと手法が登場しました。

データ・サイエンティストの一般的なワークフローは、基本的に図 1 のような反復プロセスです。

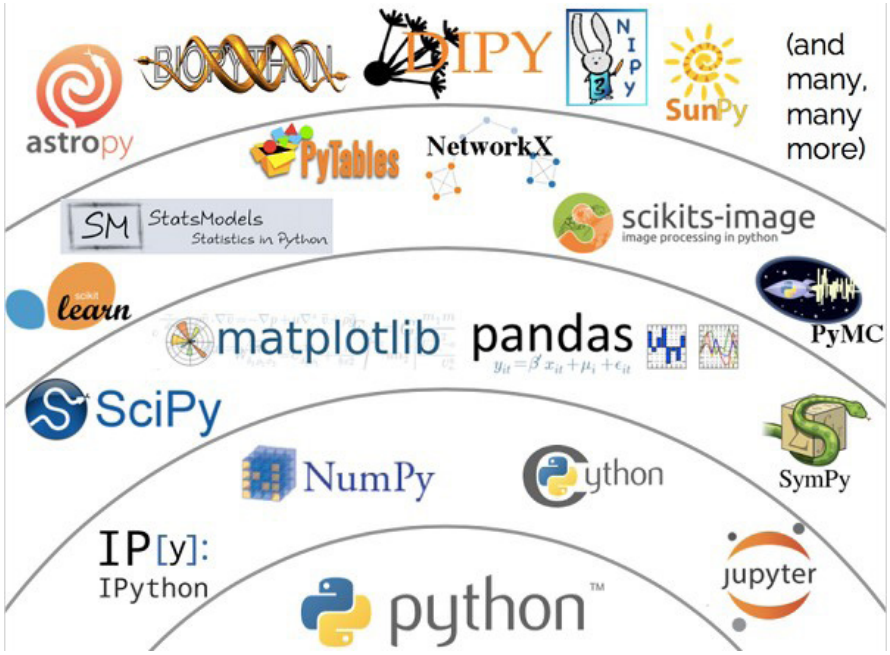


1 **データ・サイエンス・ワークフロー**

データ・サイエンティストは、(職業プログラマーではなくデータ・サイエンティストにとっての) 使いやすさと統計および数値計算向けライブラリーを広範にサポートするエコシステムの組み合わせとして、Python* および R エコシステムを長年支持してきました。近年、データサイエンスの重要なサブフィールドとしてディープラーニングと AI が出現したことで、これらのエコシステムにも多くの機能が追加されています。特に Python* は、マシンラーニングと AI ワークフローで広く普及しています。

Python* の世界では、しばらくの間、PyData スタック (図 2) が最も完全で人気の高いデータ・サイエンス・ツールのセットでした。N 次元配列データの最も低いレベルの数値計算 (NumPy) から始まり、このスタックは、科学計算 (SciPy*)、表形式 / リレーショナル・データ解析 (pandas)、シンボリック計算 (SymPy) の一連のレイヤーを提供します。

スタックの上位には、可視化 (Matplotlib、Altair*、Bokeh)、マシンラーニング (scikit-learn)、グラフ・アナリティクス (NetworkX など) に特化したライブラリーも用意されています。AstroPy や BioPython などのドメイン固有のツールキットは、これらのレイヤー上に構築され、オープンツールのディープでリッチなエコシステムを提供します。これらに加えて、Jupyter* プロジェクトは、インタラクティブ・コンピューティングのアイデア全般、特にデータに基づくストーリーテリングの推進に大きく貢献しました。多くのデータ・サイエンティストは、デフォルトの開発環境として Jupyter* を使用し、仮説とモデルを作成して調査しています。



2 **Python* データサイエンス (PyData) スタック**

OmniSci: 最新のハードウェアを活用してアナリティクスを高速化

OmniSci (旧 MapD) では、2013 年以降、HPC の手法を使用して分析 SQL とデータ可視化を同時に高速化しました。オープンソースの OmniSciDB SQL エンジンは、複数のアイデアを結集したものです。

- メモリー階層の効率的な利用による I/O アクセラレーション
- 分析 SQL カーネル向けの LLVM ベースの JIT コンパイル
- 大規模なインサイチュ・データの可視化
- マシンラーニングやディープラーニングなどのアウトオブコアのワークフローとの効率的なデータ交換

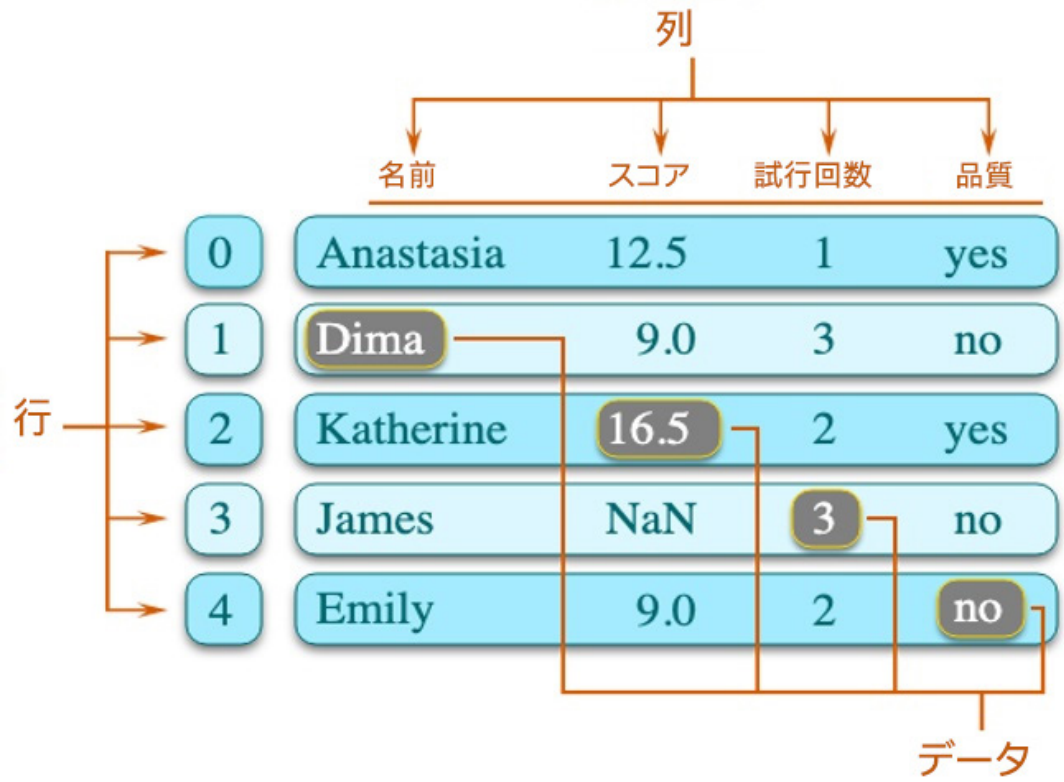
これらのアイデアを結集することにより、GPU などのハードウェア・アクセラレーターと最新のベクトル CPU の両方で、分析 SQL クエリーを 2 桁から 3 桁高速化できます。

OmniSci は当初、分析 SQL とデータ可視化の問題に取り組んでいましたが (OmniSciDB エンジンは 2017 年にオープンソース化)、主要な統合とインターフェイスを備えたオープンな PyData エコシステムにさらなる価値を提供できることに気付きました。我々は、データ・サイエンティストが PyData スタック内のプログラム・ワークフロー、つまり Jupyter* の内部で作業して、(未加工の SQL ではなく) 使い慣れた API を使用してデータを操作できるようにしようと考えました。これを念頭に、NumPy と SciPy* の作者の Travis Oliphant 氏により設立された Quansight Labs と協力して、Python* API のレイヤーのセットでコア OmniSciDB エンジンのスケーラビリティーとパフォーマンスを活用するオープン・データ・サイエンス・スタックを実現しました。

データフレームと関連する問題

その際に、分析データ構造に関連する、基本的なデータ・サイエンス・ワークフローでいくつかの問題に遭遇しました。テンソル (マシンラーニングやディープラーニングの主要なデータ構造であるデータの多次元配列) に相当するデータフレーム (**図 3**) は、テーブルの既存のリレーショナル・データベース・パラダイムに密接にマップされる、おそらくアナリティクスで最も一般的に使用されるデータ構造です。

データ・サイエンティストはあらゆる種類の表形式データからデータフレームを作成します。データフレーム・ライブラリーは Python* にも存在し、最も人気の高いのは pandas です。R (これらの起源) や Julia などの新しい言語でも同様です。



3 **データフレームの構造**

問題は、エコシステム全体のアナリティクス・ツールの数が急増するとともに、データフレームの実装数も増加することです。例えば、Apache Spark* は非常に人気の高い解析処理エンジンおよびプラットフォームです。多くのデータフレームの機能を提供するデータフレーム API (Spark* の分散データセット) を備えていますが、pandas とも、R とも異なります。OmniSci では、これらの API はすべて非常に柔軟であるにもかかわらず、データ・サイエンティストが非常に大規模なデータセットのインタラクティブな調査を行うことを困難にする、スケーラビリティーとパフォーマンスの問題に悩まされていることが分かりました。

例えば、Spark* では非常に大きな分散データフレームを扱えますが、Spark* は何よりもまず分散システム向けに設計されている (そして Java* 仮想マシンでの実行には追加のオーバーヘッドが発生する) ため、計算エンジンは真にインタラクティブな (1 秒未満の) クエリーを処理できません。一方、pandas はリッチで強力な API ですが、操作が Python* で実装されており、インタープリター環境での実行により大幅なオーバーヘッドが発生するため、スケーラビリティーの問題があります。

最後に、言語やエコシステム (JVM、Python*、R) にわたる既知の交換の問題があるため、さまざまな計算環境に分散できる完全なワークフローを構築することは困難で非効率的です。例えば、Spark* は、Python*/pandas などのローカル・コンピューティング環境での詳細な解析のため、大規模なデータセットを操作および形成するワークフローの初期段階においてよく使用されます。最近まで、これによるスケーラビリティーを制限する交換と変換のオーバーヘッドが発生していました。幸いにも、このニーズに対応するデファクト・スタンダードとして Apache Arrow* が登場しました。しかし、異なるフレームワーク間のデータ交換での採用はまだ進んでいません。

データサイエンスへの取り組みを始めて間もなくしてから、OmniSci は Ibis に参加しました (**図 4**)。pandas の作者の Wes McKinney 氏が開発に携わっている API である Ibis は、OmniSci などの大規模なデータ処理およびストレージシステムと Python* データ・サイエンス・スタックをつなぐ興味深い方法を提供します。この方法では、PyData スタックとデータストアの世界をつなぐ、いくつかの主要なレイヤーが提供されます。Ibis プロジェクトのウェブサイトによれば、次のコンポーネントが含まれます。

- 構造化データに対して構成可能で再利用可能なアナリティクスを可能にする、アナリティクス向けに特別に設計された **pandas に似たドメイン固有の言語 (DSL)** (Ibis 表記)。SQL SELECT クエリーで表現できることは Ibis で記述できます。
- HDFS およびその他のストレージシステムへの**統合ユーザー・インターフェイス**。
- 複数の SQL システムをターゲットとする**拡張可能なトランスレーター・コンパイラー・システム**。

4 **データサイエンスとデータストアをつなぐ API の Ibis**

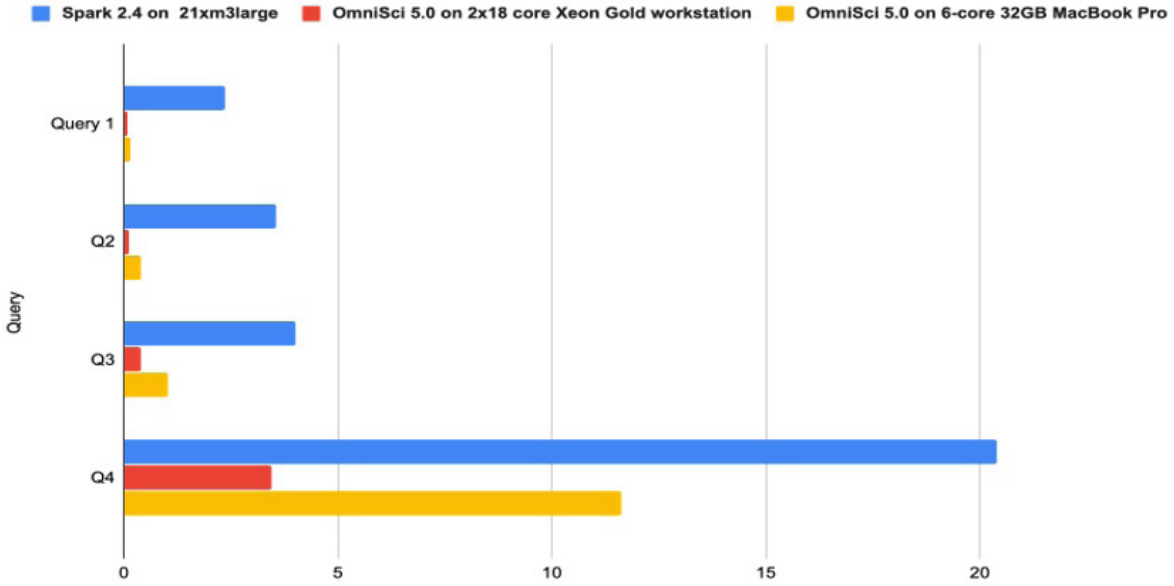
MySQL*、Postgres*、BigQuery*、OmniSci、Spark*、Clickhouse などを含む SQL システムをサポートし、新しいバックエンドを簡単に追加できます。

また、宣言型の Python* 可視化フレームワーク、Altair* で構築されるオープンデータの可視化にも投資しました。Altair* は、データの可視化に最新の宣言型のユーザー・インターフェイス・フレームワーク、Vega および Vega-Lite を利用します。Quansight と協力して Ibis と Altair* を統合することにより、ソースシステムからデータを移動することなく、非常に大きなデータセットを可視化して調査できるようになりました。

OmniSci とインテル : データサイエンスにおける連携を強化

2019 年に、これらの統合の最初のバージョンをリリースしました。これらもすべて、各プロジェクト内でオープンソース化されています。その際に、オープンソースの OmniSciDB エンジンの能力とパフォーマンスを目にしたインテルと、非常に有益な共同作業を開始することになりました。我々はともに、データフレーム中心の分析ワークフローを高速化するため、共通の目標として、その機能とデータ・サイエンス・ツールの橋渡しとなる方法を検討しています。インテルのチームは、エコシステムの能力を示すリファレンス・プラットフォームとして OmniSci を選択しました。詳細なシステムとチューニングの専門知識を備えたインテルのチームは、この目標に向けて大きく貢献しています。

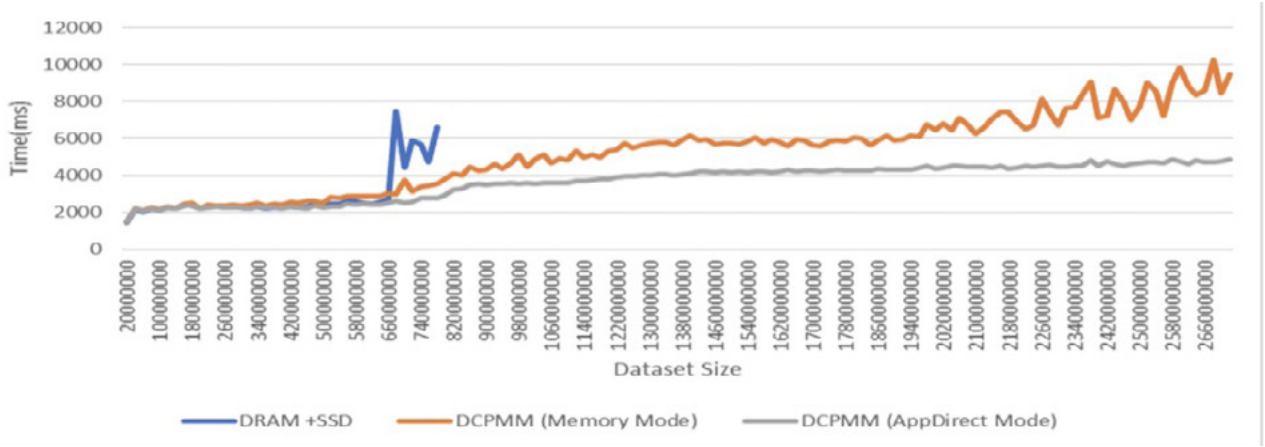
チームは、**インテル® VTune™ プロファイラー**などのインテルのプロファイリング・ツールを使用して、インテル® CPU ファミリーで OmniSciDB 実行 (**インテル® スレッディング・ビルディング・ブロック**とロックの使用を含む) のパフォーマンス・チューニングと最適化を行うべき領域を特定しました。OmniSci は、タクシー乗車ベンチマークのすべてのクエリーで、Spark* クラスターを大幅に上回っています (**図 5**)。



5 **インテル® CPU で OmniSci を使用したタクシー乗車ベンチマーク。x 軸は各クエリーの完了時間 (秒) を示す。このベンチマークの詳細はこちら (英語) を参照。**

チームは、最初にデータをインポートすることなく、OmniSci のインメモリーデータ取り込みに使用できるハイパフォーマンスな Arrow* ベースの取り込みコンポーネントも提供しました。目標は、OmniSci のストレージシステムを使用せずに、CSV や Apache Parquet* などのスタティック・ファイルからデータフレームを作成するのに似たワークフローをサポートすることです。

現在は、エンジンのスケールとパフォーマンスを活用しながら Python* ベースのデータ・サイエンス・ワークフローに簡単に組み込むことができるように、OmniSciDB エンジンをライブラリーにコンポーネント化する作業を行っています。また、OmniSci のストレージ・サブシステムを最適化して**インテル® Optane™ DC パーシステント・メモリー・モジュール** (**図 6**) を活用する作業も行っています。最初のベンチマーク結果は非常に有望であり、OmniSci がハードウェア・フットプリントを削減しつつ、非常に大規模なデータセットをサポートできる可能性が示されました。



6 **インテル® Optane™ DC パーシステント・メモリーで OmniSciDB を使用した予備スケーリングの結果**

OmniSci の高速データ・レンダリング・パイプラインを活用するため、**oneAPI** (英語) を使用して新しいハードウェア、特に**インテルの Xe GPU** をターゲットにすることも検討しています。これにより、OmniSciDB エンジンだけでなく、OmniSci スタック全体を、データセンターからデスクトップやラップトップまで、あらゆるインテル® プラットフォームで実行できるようになります。

データ・サイエンティストにとってのメリット

この共同作業は、データ・サイエンティストにもれなく大きなメリットをもたらします。初めて、非常に効率的なハードウェア・フットプリントで数十億行のデータを含むデータセットに対して大規模な分析計算を実行できるようになります。このスケーラビリティー、パフォーマンス、熟知性の組み合わせにより、インテルと OmniSci の共同作業は、データレイクやデータ・ウェアハウスなどの大規模なデータ処理およびストレージシステムを補完するハイパフォーマンス・データ・サイエンス環境として魅力的です。

テラバイトまでのデータセットは、ラップトップでインタラクティブに分析および可視化できます。デスクトップ・クラスのシステムでは、最大 10TB 以上を分析できます。Arrow* を使用すると、**インテル® データ・アナリティクス・アクセラレーション・ライブラリー**などのマシンラーニング・ライブラリーをワークフローにシームレスに統合できます。

スケーラビリティー、パフォーマンス、柔軟性

OmniSci とインテルはともに、データ・サイエンティストに魅力的な新しいプラットフォームを提供します。オープンな、ハードウェアを考慮した、ハイパフォーマンスなアナリティクス・エンジンの機能とインテル® テクノロジー・エコシステムの能力を統合することにより、データ・サイエンス・ワークフローのスケーラビリティー、パフォーマンス、柔軟性において大きなメリットが得られることが示されています。

手順 (英語) に従って、Docker* を実行している Mac* または Linux* ラップトップで OmniSciDB をダウンロードして試すことができます。OmniSci について学び、データ・サイエンス・ワークフローを構築するのに役立つ関連情報を次に示します。

- **データサイエンスにおける OmniSci の利用についてのブログ (英語)**
- **データサイエンスにおける OmniSci のオンデマンド・セミナー (英語)**
- **Todd Mostak 氏による OmniSci アーキテクチャーの説明 (英語)**

VIDEO HIGHLIGHTS

ヘテロジニアス・コンピューティングへのコラボレーション

業界や学界のリーダーが、ヘテロジニアス・プログラミングをさらに進めるための業界全体のイニシアチブとしての oneAPI の重要性と、その可能性を実現するためにどのように貢献しているかについて話します。

視聴する (英語) >

Software

THE PARALLEL UNIVERSE

JPN/2008/PDF/XL/MKTG/SU