一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2022-17,DC2022-17(2022-10)

LLVMにおけるOpenMP GPUオフローディングの性能調査

小島　拓也†

† 東京大学　情報理工学系研究科　113–8656 東京都文京区本郷 7 丁目 3–1
E-mail: †tkojima@hal.ipc.i.u-tokyo.ac.jp

あらまし　GPU などのアクセラレータを用いたアプリケーション実行を加速させるヘテロジニアスコンピューティングが広く普及している．これに伴い, アクセラレータで実行させる計算を効率的に開発する手法が必要となる．OpenMP は共有メモリ型並列環境向けにプログラムを並列化するための API であり, これまでマルチコアプロセッサで広く利用されてきた．その後, バージョン 4.0 からはアクセラレータに計算をオフローディングする機能が追加された．コンパイラ基盤として有名な LLVM では NVIDIA 社および AMD 社の GPU へオフローディングする実装が利用できる．本研究では, これらの実装が同等のアプリケーションを記述した CUDA および OpenCL の実装と比較してどれほどの性能が得られるのかを評価し, 報告する．

キーワード　OpenMP, GPU オフローディング

A Performance Analysis of OpenMP GPU Offloading in LLVM

Takuya KOJIMA†

† Graduate School of Information Science and Technology, The University of Tokyo 7–3–1 Hongo, Bunkyo-ku, Tokyo 113–8656, Japan
E-mail: †tkojima@hal.ipc.i.u-tokyo.ac.jp

1. はじめに

シングルコア CPU からマルチコア CPU へのパラダイムシフトが起きてから十数年の月日が流れ, 半導体微細加工技術の向上鈍化と相まってマルチコア化による性能スケーリングを維持するのは困難になりつつある．そこで, 近年は GPU などのアクセラレータを用いるヘテロジニアスコンピューティングが注目を集めており, とりわけ HPC 分野や機械学習の分野では必須とも言える状況にある．このことは TOP500 にランクインしているスーパーコンピュータの多くが GPU を搭載していることからも窺える [1]．こうした経緯から並列プログラミングモデルに関する様々な技術が革新がもたらされている．

OpenMP は共有メモリ型のマルチコア環境を想定した並列プログラミングモデルである．C/C++ および Fortran の言語拡張 API という位置付けで, ソースコード中の並列化したい箇所に指示文 (ディレクティブ) を与えるだけで並列化が可能である．

一方で, 初めからアクセラレータを想定した並列プログラミングモデルはそれ以前からいくつか存在している．NVIDIA は 2007 年に自社の GPU で汎用計算を可能にする専用コンパイラやランタイムライブラリを含むプログラミング環境 CUDA の提供を開始した．その翌年には Khronos Group よりオープン標準な OpenCL が発表され, GPU だけでなく FPGA など様々なアクセラレータのプログラミング環境として利用されている．ところが, CUDA および OpenCL は低レベル API であるため, アクセラレータが多様化する昨今はその移植性の低さが問題となっている．

このような低レベル API を隠蔽するプログラミングモデルとして OpenACC や SYCL が存在する．OpenACC は OpenMP と同様にディレクティブを用いてアクセラレータで計算させたい処理を指示する．SYCL は OpenCL の仕様を策定している Khronos Group が同様に策定している抽象化レイヤで, C++ のテンプレートなどを用いることでシングルソースで多様なプラットフォームに対応可能な並列プログラミングが可能となる．SYCL の実装例としては Intel oneAPI DPC++, hipSYCL[2], neoSYCL[3] などがいくつか存在している。

こうした経緯から OpenMP は歴史的にマルチコアシステム向けのプログラミングモデルであったが, 2013 年にリリースされたバージョン 4.0 からアクセラレータへのオフローディング機能がサポートされた．

OpenMP は GNU による実装や Intel コンパイラなど多数存在するが, 特にコンパイラ基盤として近年広く利用されている

LLVM[4] に注目すると OpenMP のオフローディング機能が実装されたのは 2017 年にリリースされたバージョン 5.0.0 である．この時点で CUDA をバックエンドとして利用する NVIDA GPU 向けライブラリも併せて提供された．遅れること 2021 年には，AMD GPU 向けに HSA runtime をバックエンドとして利用するライブラリがソースツリーに現れ，バージョン 13.0.0 にて正式にビルドオプションに加わった．このように，OpenMP による GPU へのオフローディング機能は他のプログラミング環境と比較して，登場から月日が浅い．さらに，もともとは共有メモリ型のマルチコア向けの技術であったことから GPU が本来持っている性能を引き出すことができるのかどうかが懸念される．そこで，本研究は CUDA や OpenCL で実装されたアプリケーションと比較して OpenMP によって達成できる性能を評価し，議論を行う．

2. OpenMP による並列プログラミング

本節では OpenMP による並列プログラミングの方法について述べる．OpenMP による並列化の指示で最も簡単なのは parallel 構文と for 構文の結合構文である`#pragma omp parallel for` を for ループの直前に追記する方法である．特に指示しない場合は for ループは均等に分割され，各々が別のスレッドで並列に実行される．Code1 は単精度の浮動小数点数を要素に持つ 2 つのベクトルについて要素ごとに a * X plus Y を計算する処理 (SAXPY) を並列化するコード例である．for 文の終了時に関しては特に指示をしなくても暗黙的にスレッド間でバリア同期が行われる．このように，ディレクティブの挿入だけで簡単に並列化可能であるのが OpenMP の特徴である．

Code 1: マルチコア CPU 向けのコード例

```
1 void saxpy(int N, const float A, float *X,
      float *Y)
2 {
3    #pragma omp parallel for
4    for (i = 1; i < N - 1; i++) {
5       Y[i] = A * X[i] + Y[i];
6    }
7 }
```

2.1 target 構文によるアクセラレータオフローディング

次に，GPU へオフローディングする方法について説明する．Code2 は OpenMP による記述例であり，比較のために CUDA によって同等の計算を行う場合を Code3 に示す．OpenMP に関しては target 構文およびそれと組み合わせる data 構文および teams 構文が重要なディレクティブである．

Code 2: target 構文によるオフローディング例

```
1 void saxpy(int N, const float A, float *X,
      float *Y)
2 {
3    #pragma omp target data map(to:A[0:N]) map(
         tofrom:B[0:N])
4    #pragma omp target teams distribute parallel
         for num_teams(256)
5    for (i = 1; i < N - 1; i++) {
6       Y[i] = A * X[i] + Y[i];
7    }
8 }
```

Code 3: CUDA によるプログラム例

```
1 __global__ void saxpy(int N, float A, float *X,
      float *Y)
2 {
3    int i = blockIdx.x * blockDim.x + threadIdx.x;
4    Y[i] = A * X[i] + Y[i];
5 }
6
7 void host(...)
8 {
9    //...省略: 配列の初期化など
10   float *X_gpu, *Y_gpu;
11   cudaMalloc(&X_gpu, N * sizeof(float));
12   cudaMalloc(&Y_gpu, N * sizeof(float));
13   cudaMemcpy(X_gpu, X, N * sizeof(float),
         cudaMemcpyHostToDevice);
14   cudaMemcpy(Y_gpu, Y, N * sizeof(float),
         cudaMemcpyHostToDevice);
15
16   saxpy<<<N/256,256>>>(N, A, X_gpu, Y_gpu);
17
18   cudaMemcpy(Y, Y_gpu, N * sizeof(float),
         cudaMemcpyDeviceToHost);
19
20   cudaFree(X_gpu);
21   cudaFree(Y_gpu);
22   //...省略
23 }
```

OpenMP では共有メモリ型のプログラミングモデルあるため，特に指示をしなくてもアクセスされる配列などは自動的に共有される．一方で，GPU をはじめとするアクセラレータ (デバイス) に計算をオフローディングする場合は，ホスト (CPU) とデバイスとの間で明示的なデータ転送の指示が必要となる．これを行うのが data 構文であり，続く map 節によって，ホスト上のどのデータをどのようにデバイスへマップするかを指定する．この節ではマップタイプとマップされる変数を指定する．マップタイプには計算開始前にホストからデバイスへの転送だけを行う `to`，計算終了時にデバイスからホストへの転送だけを行う `from`，この両方を行う `tofrom`，中間データ用にメモリ領域だけを確保する `alloc`などがある．この例では配列 X は内容が変更されることがないため，`to`を指定し，配列 Y については `tofrom`を用いている．このように，データ転送についてはプログラマの責任によって行われる．

次に，ループ処理をデバイス上の複数のスレッドに分割する指示を行う．ここで重要になるのはスレッドの階層化である．マルチコア CPU のように単純なスレッドモデルでは階層化されたアーキテクチャを持つ GPU のようなデバイスには適合せず，性能を引き出すことができない．例えば，NVIDA 社の GPU

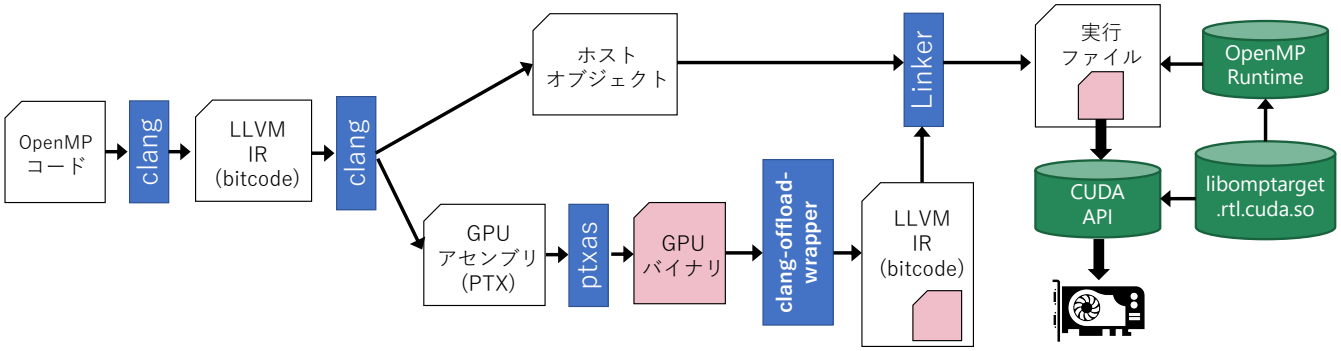


図 1: OpenMP オフローディングのコンパイルフローと実行の仕組み

では Streaming Multiprocessor(SM) と呼ばれるユニットを複数もち, 各 SM は複数の CUDA コアやスケジューラ, ロードストアユニットなどで構成される. SM は Warp と呼ぶ複数のスレッド (現行では 32 スレッド) 単位でスケジュールを行う. CUDA ではスレッドを階層化して, グリッド, スレッドブロックという構造の概念を導入している. Code3 では, 256 スレッドを一つのスレッドブロックとして, スレッド全体を (N/256) ブロックで構成している. 一つのブロックは一つの SM に割り当てられ, 各 SM は 256 スレッドを実行する. このようにして, GPU アーキテクチャに適合するようなスレッドの分割および階層化が施される.

OpenMP の teams 構文はこのようなスレッドの階層化を行うために導入されている. team は複数のスレッドで構成され, teams 構文は League と呼ぶ複数の team を生成する. この階層構造では team 内のスレッド間でバリア同期が行われるが, League 内の team 間ではバリア同期が行われない. つまり, マルチコア CPU 向けの Code1 で用いている parallel 構文は 1 つの team を作っていることになる. 特に, CUDA との対応関係については League はグリッドに, team はスレッドブロックに対応する. ただし, CUDA ではグリッドおよびスレッドブロックが 3 次元的な量で表現できるのに対して League, team については 1 次元的な量でのみ表される. Code2 では num_teams 句によって生成されるチーム数を明示的に指定しているが省略することでランタイムが自動で決定することもできる. また, Code1 のように parallel 構文でもオフローディングすること自体は可能であるが, 実行時間は劇的に遅くなる場合がほとんどである. CUDA による Code3 と比較してコード量は少なく, また CPU 実行の場合と異なるのはプラグマだけなのでソースコードの流用性が極めて高い. 並列化したくない場合は, コンパイル時に OpenMP を有効にするオプションを外せばシングルスレッドのプログラムとしてコンパイルされる. さらに, target 構文において if 節を用いればイテレーション数が小さい場合は CPU で実行し, 十分なイテレーション数がある場合だけ GPU で実行するようにプログラムすることも可能である.

2.2 GPU オフローディングのためのコンパイルフロー

次に, OpenMP オフローディングを含む単一のソースコードからホストとデバイスの両方を利用するプログラムバイナリが生成される仕組みについて, NVIDA 社 GPU 向けの LLVM における実装を例に説明する. 図 1 に LLVM/clang によるコンパイルフローを示す. まず, OpenMP コードは C フロントエンドの clang によってアーキテクチャ非依存の中間表現 LLVM-IR に変換される. このような中間表現を経由し, LLVM は様々なアーキテクチャのアセンブリコードを生成することができる. この特徴を用いて中間表現はホスト CPU で実行されるオブジェクトファイルと NVIDA GPU のアセンブリファイル (PTX ファイル) の両方を生成する. PTX ファイルは CUDA の Toolkit に含まれる PTX アセンブラによって GPU のバイナリへ変換される. この GPU バイナリをホストの実行ファイルに同梱するために, clang-offload-wrapper というユーティリティツールによって LLVM-IR に変換され, これをホストオブジェクトとリンクすることで実行ファイルが完成する.

プログラムの実行時には, まず OpenMP のランタイムライブラリが動的にリンクされ, このランタイムがさらに対応するデバイスのランタイムライブラリをロードする. プログラムが target 構文の領域に到達すると, デバイスのランタイムライブラリは同梱された GPU バイナリのイメージを CUDA の API(cuModuleLoadDataEx ルーチン) を用いて GPU にロードし, 実行する. このようにして CUDA の API は隠蔽され, プログラマはスレッドの分割や, データ転送などを意識するだけで GPU を活用した並列プログラミングが可能となる.

3. ベンチマークによる性能分析

OpenMP による GPU オフローディングの抽象化によってソースコードの流用性が高まり, 生産性の向上が期待できる一方で, きめ細かなチューニングができないことにより得られる本来の性能を発揮できない可能性がある. そこで, 本節ではいくつかのベンチマークを用いて, 同等の計算を行う CUDA および OpenCL と OpenMP による実装を比較する.

3.1 ベンチマーク

評価に用いるベンチマークスイートには PolyBench[5] を用いている. PolyBench のベンチマークは datamining, linear algebra, stencils, medley の 4 つのカテゴリーに分類されており, 様々なパターンのループ処理を計算カーネルとして含む. 問題のスケールは 5 つ用意されており, MINI, SMALL, STANDARD, LARGE, EXTRALARGE の順に問題サイズが大きくなる. また, PolyBench-ACC[6] というアクセラレータ向けの実

表 1: 評価に用いたベンチマークの概要

名前	カテゴリー	内容	データセット
correlation	datamining	NxN の相関行列の計算	$N \in \{512, 1024, 2048, 4096, 8192\}$
2mm	linear algebra	$D = \alpha A \times B \times C + \beta D$ の行列計算 (行列サイズ NxN)	$N \in \{256, 512, 1024, 2048, 4096\}$
syr2k	linear algebra	NxN の対称行列に対するランク 2k の更新	$N \in \{256, 512, 1024, 2048, 4096\}$
lu	linear algebra	NxN 行列に対する LU 分解	$N \in \{512, 1024, 2048, 4096, 8192\}$
fdtd-2d	stencils	平面 NxN 平面に対する 2 次元 FDTD 解析 (反復ステップ数 500)	$N \in \{512, 1024, 2048, 4096, 8192\}$
jacobi-2d	stencils	NxN 行列に対するヤコビ法 (反復ステップ数 20)	$N \in \{256, 512, 1024, 2048, 4096\}$
convolution-3d	stencils	NxNxN 行列に対する 3x3x3 カーネルによる畳み込み演算	$N \in \{64, 128, 256, 384, 512\}$

表 2: 評価に用いる GPU と基本スペック

型番	マイクロアーキテクチャ	CUDA コア/SP 数	定格クロック (MHz)	VRAM
Quadro P620	Pascal	640	1266	GDDR5 4GB
RTX-2080Ti	Turing	4352	1350	GDDR6 11GB
RTX-A4000	Ampere	6144	735<	GDDR6 16 GB
Radeon 5700XT	Navi	2560	1605	GDDR6 8GB

装も公開されており, OpenMP, OpenCL, CUDA, OpenACC, HMPP による記述が利用可能である. 本研究ではこの中から表 1 に示す 7 個のベンチマークを選択し, 評価に用いた.

3.2 実験環境

評価にはベンダーおよび世代, コア数の異なる 4 つの GPU を用いた. 3 つは NVIDIA 社の GPU であり, 1 つは AMD 社の GPU である. コア数に関しては NVIDIA GPU については CUDA コア, AMD GPU については Streaming Processor (SP) の数を示した.

コンパイル環境, ランタイムライブラリについては以下のバージョンを用いた.

- LLVM/clang 14.0.6
- CUDA 11.2
- ROCm 5.0.2

ベンチマーク中の OpenMP による実装はメニーコア向けのプラグマが用いられているため, target temas 構文を用いる記述への変更など最低限の修正のみ行なった. 一方で, NVIDIA GPU で実行する CUDA の実装および AMD GPU で実行する OpenCL の実装に関してはそのままの記述を用いた. 故に, CUDA と OpenCL 自体のチューニングについては考慮しない. ただし, 不必要なデータ転送を行うものがあったため, その部分のみコメントアウトして評価に用いた. いずれのケースも O3 の最適化レベルでコンパイルを行なった.

3.3 計算カーネルの実行時間比較

まず初めに, メモリ間のデータ転送時間などを除く計算カーネルの実行時間を比較する. OpenMP に関してはきめ細かな時間計測を行うにはランタイムライブラリの細工が必要であったため, 本評価では簡易的に target data 構文が対象とするコードブロック内部に時間計測用のコードを埋め込み, target teams 構文で指示されたブロックの実行時間を計測した. 一方で, CUDA についてはカーネル関数の呼び出し前後, OpenCL に関しては clEnqueueNDRangeKernel 関数によるカーネルの実行前後で時間を計測した. 図 2-8 に各ベンチマークでデータセットのサイズを変えて計測した結果を示す. 実時間は対数軸で示し, OpenMP の実行時間を CUDA および OpenCL の実行時間で正規化したグラフも併せて示している.

3.3.1 correlation

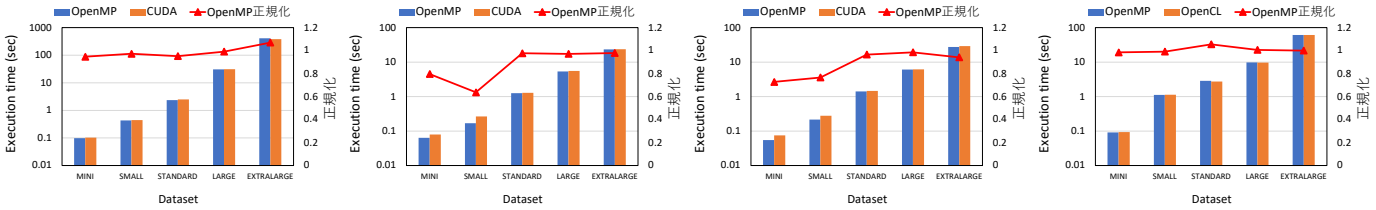
このベンチマークでは OpenMP によるオフローディングは CUDA および OpenCL による実装とほぼ同じ実行時間で完了している. 特に, EXTRALARGE のデータサイズでは RTX-2080Ti や RTX-4000 では CUDA による実装より 2-6% 程度高速に実行できている. 一方で, これ以外のベンチマークでは OpenMP は数倍以上の計算時間を要している. correlation だけがオーバーヘッドが小さい理由に関しては, このベンチマークの一つのスレッド内に 2 重のループが存在するからであると考える. 今回用いたベンチマークの記述では各配列要素で reduction 演算を正しく実行するために, このようなコードが記述されている. しかし, CUDA では SM 内のスレッドで共有可能な shared メモリを用いて reduction 演算を行うのが一般的である. OpenMP の場合, target data 構文にてマップされるのは全てグローバルメモリであり, GPU 内に存在するその他のメモリを活用することはできない. 故に, このようなチューニングを施した場合, OpenMP の実行時間との差は広がると予想される.

3.3.2 2mm

このベンチマークではどの GPU においても OpenMP による実装は CUDA と OpenCL と比較して 2 倍程度の時間を要している. 2 つの計算カーネルで構成されているが, そのどちらも最内ループは reduction 演算となっているため, correlation と同様に 1 つのスレッドがこのループを処理する. ところが, このベンチマークではスレッド内のループは 1 重ループであり, CUDA を用いる場合でも前述のようなチューニングをせずともある程度は効率的に実行できており, OpenMP の場合との性能差が生まれていると推測される.

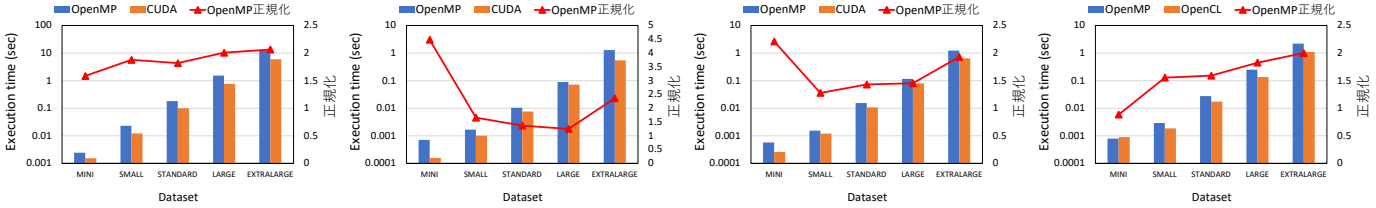
3.3.3 syr2k

いずれの GPU に関してもデータサイズが LARGE の時に,



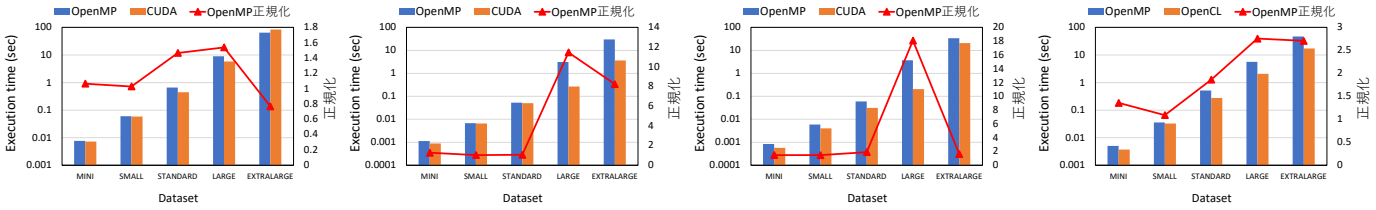
(a) Quadro P620 (b) RTX-2080Ti (c) RTX-A4000 (d) Radeon 5700XT

図 2: correlation による比較結果



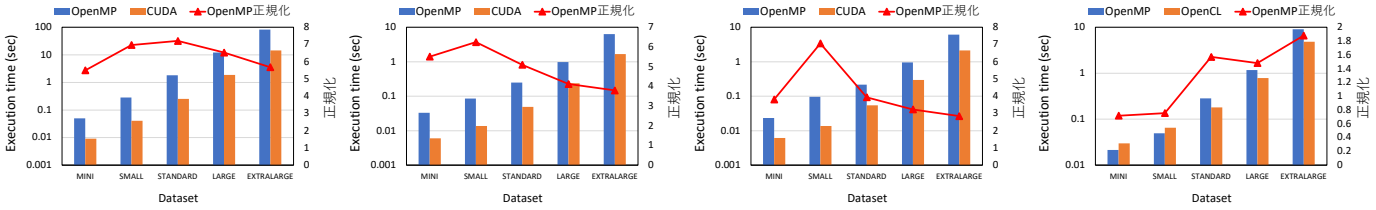
(a) Quadro P620 (b) RTX-2080Ti (c) RTX-A4000 (d) Radeon 5700XT

図 3: 2mm による比較結果



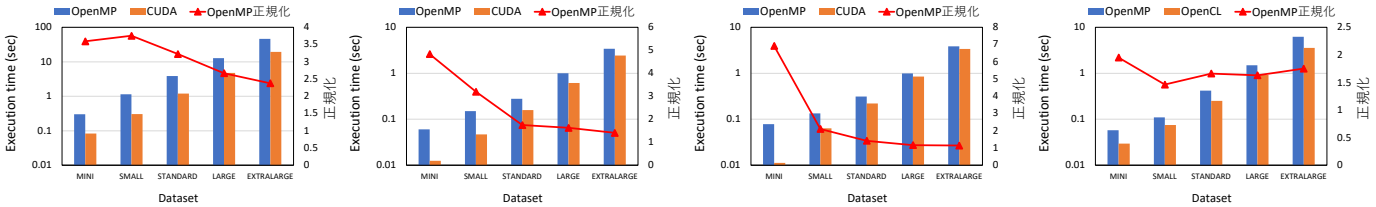
(a) Quadro P620 (b) RTX-2080Ti (c) RTX-A4000 (d) Radeon 5700XT

図 4: syr2k による比較結果



(a) Quadro P620 (b) RTX-2080Ti (c) RTX-A4000 (d) Radeon 5700XT

図 5: lu による比較結果



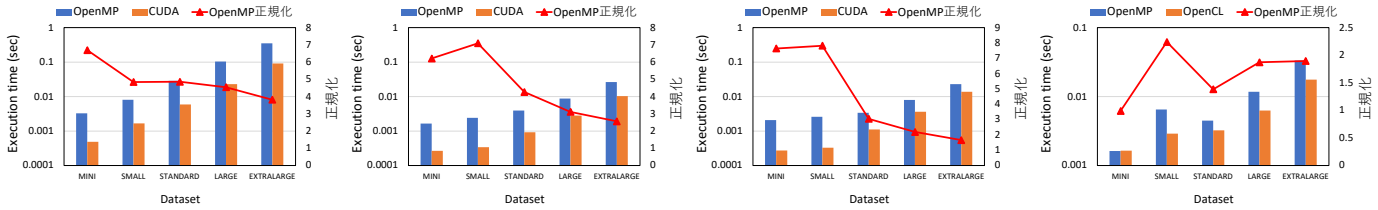
(a) Quadro P620 (b) RTX-2080Ti (c) RTX-A4000 (d) Radeon 5700XT

図 6: fdtd-2d による比較結果

CUDA との実行時間比が最大であり, RTX-A4000 に関しては 18 倍にも及ぶ. 一方で EXTRALARGE になると比率は下がり, Quadro P620 関しては OpenMP の方が実行時間が 15% ほど短縮されている. また, この GPU では LARGE サイズにおても他の GPU と比べて実行時間の増加が穏やかである. この中で最も古い世代の GPU であることを考慮すると, 新しい世代の GPU では CUDA による記述で新しいアーキテクチャを最大限に活用できるコードが生成できるのに対して LLVM ベースの OpenMP ではそれが困難であると推測される.

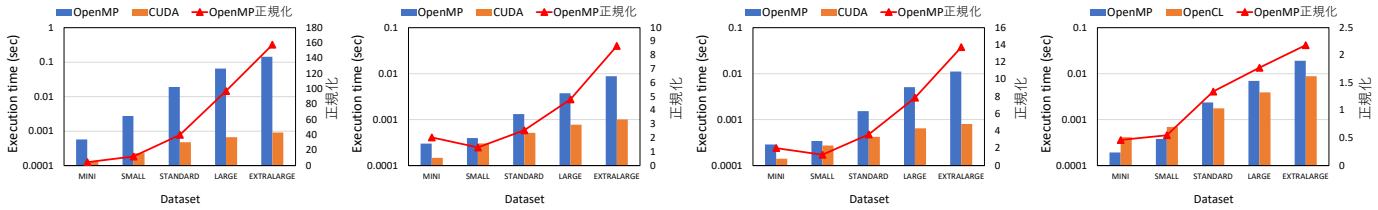
3. 3. 4 lu,fdtd-2d および jacobi-2d

fdtd-2d では, データサイズが大きくなるにつれて OpenMP のオーバーヘッドが小さくなっている. 特に, RTX-2080Ti や RTX-A4000 では最大サイズの EXTRALARGE では 10% 程度のオーバーヘッドで収まっている. これら 3 つのベンチマー



(a) Quadro P620　(b) RTX-2080Ti　(c) RTX-A4000　(d) Radeon 5700XT

図 7: jacobi-2d による比較結果



(a) Quadro P620　(b) RTX-2080Ti　(c) RTX-A4000　(d) Radeon 5700XT

図 8: convolution-3d による比較結果

クでは計算カーネルは反復して実行される．OpenMP の場合，計算カーネルを起動するたびに OpenMP ランタイムのラッパー関数が呼ばれる．ゆえに，データサイズが小さい場合このオーバーヘッドが無視できず，CUDA や OpenCL の場合と比べて相対的に遅くなっていると考えられる．lu や jacobi-2d も fdtd-2d と似た計算パターンであるため，類似の傾向が見られる．

3.3.5　convolution-3d

このベンチマークでは，OpenMP による計算時間のオーバーヘッドが他のベンチマークと比べて極めて大きく，Quadro-P620 ではデータサイズが EXTRALARGE の時におよそ 160 倍の実行時間になっている．他のベンチマークのようにスレッド内の処理に for ループは存在せず，スレッド内の計算が最もシンプルなカーネルであるため，CUDA や OpenCL の実装は shared メモリを用いたチューニングを行わずとも十分に最適化されたコードを生成できていると考えられる．一方で，OpenMP のコンパイルフローで生まれる PTX アセンブリは最適化が不十分であり，その影響が最も強く現れている結果であると考えられる．

ベンチマーク全体と通して，RTX-2080Ti と RTX-A4000 は似た傾向が多く見られた．また，AMD GPU の場合は OpenCL と比べた OpenMP のオーバーヘッドは高々 3 倍であり，NVIDIA GPU で見られるような 10 倍以上にもおよぶ悪化は見られなかった．

4. おわりに

本稿では，近年サポートされた OpenMP における GPU などのアクセラレータへのオフローディング機能の評価を行ない，その結果を報告した．CUDA や OpenCL で記述した場合と同程度の性能が得られるベンチマークもみられたが，これは CUDA および OpenCL の実装が十分に最適化されていないことによるものであると考えられ，それ以外のベンチマークでは数倍から数十倍の実行時間増加が確認された．特に，NVIDA GPU の方がこの影響は大きく，LLVM により生成される PTX アセンブリが CUDA が生成するものと比べても十分に最適化がなされていないと推測される．今後は，プロファイラを用いてこれらの実行時間増加の影響をより詳細に調査し，ラインタイムライブラリや LLVM で標準的に適用される最適化などのうち，GPU 向けには無効にした方が良いものや，GPU に特化した最適化の必要性などに関して明らかにしていく必要がある．

文　献

[1] S. Heldens, P. Hijma, B.V. Werkhoven, J. Maassen, A.S. Belloum, and R.V. Van Nieuwpoort, “The landscape of exascale research: A data-driven literature analysis,” ACM Computing Surveys (CSUR), vol.53, no.2, pp.1–43, 2020.

[2] A. Alpay, B. Soproni, H. Wünsche, and V. Heuveline, “Exploring the possibility of a hipsycl-based implementation of oneapi,” International Workshop on OpenCL, pp.1–12, 2022.

[3] Y. Ke, M. Agung, and H. Takizawa, “neosycl: a sycl implementation for sx-aurora tsubasa,” The International Conference on High Performance Computing in Asia-Pacific Region, pp.50–57, 2021.

[4] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization-IEEE Computer Society, p.75 2004.

[5] L.-N. Pouchet and T. Yuki, “Polybench/c,” 2016.

[6] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to gpu codes,” 2012 innovative parallel computing (InPar)Ieee, pp.1–10 2012.