jh230037

大規模アプリケーションの高性能な実用的アクセラレータ対応手法

下川辺　隆史（東京大学）

概要　近年、電力と設置面積の制約のもと、最大限に計算性能を高くするために、多くのスパコンで GPU などの演算加速装置が導入されている。本課題では、スパコンで動作している CPU アプリケーションを演算加速装置を搭載したスパコンへ移植し、その移植の方法を確立することを目的とする。様々な分野の研究者が開発した CPU アプリケーションを移植することを念頭に、演算加速装置で高い性能を達成することを目指しながらも、専用言語を多用せず、標準的・汎用的手法を用いて移植を実現する。実用的な移植方法として、OpenMP、OpenACC、do concurrent などの活用を検討する。本年度は、海洋大循環モデル COCO を対象として do concurrent を用いて GPU 化を進めた。また、C++で記述された拡散方程式を対象に性能可搬性プログラミングモデルである SYCL を用いて GPU 移植を行い、従来手法と同等の性能がでることを示した。

1. 共同研究に関する情報

⑴ 共同利用・共同研究を実施している拠点名

東京大学 情報基盤センター

名古屋大学 情報基盤センター

大阪大学 サイバーメディアセンター

(2) 課題分野

大規模計算科学課題分野

(3) 共同研究分野(HPCI 資源利用課題のみ)

超大規模数値計算系応用分野

(4) 参加研究者の役割分担

- 下川辺 隆史（東京大学）：移植コード開発と最適化
- 額田 彰（筑波大学）：移植コード開発と最適化
- 古村 孝志（東京大学）：OpenSWPC に関する助言
- 羽角 博康（東京大学）：COCO に関する助言
- 川崎 高雄（東京大学）：COCO に関する助言
- 山岸 孝輝(高度情報科学技術研究機構)：COCO に関する助言
- 朴 泰祐（筑波大学）：演算加速装置に関する助言
- 高橋 大介（筑波大学）：演算加速装置に関する助言
- 三木 洋平（東京大学）：演算加速装置に関する助言
- 塙 敏博（東京大学）：大規模計算に関する助言
- 中島 研吾（東京大学）：計算科学に関する助言
- 星野 哲也（名古屋大学）：演算加速装置に関する助言
- 勢見 達将（筑波大学）：移植コード開発
- 佐久間 大我（東京大学）：移植コード開発
- Ziheng Yuan（東京大学）：移植コード開発

2. 研究の目的と意義

近年、電力と設置面積の制約のもと、最大限に計算性能を高くするために、多くのスパコンで GPU などの演算加速装置が導入されている。今後この傾向はさらに加速すると予想され、将来のスパコンでは、大多数のアプリケーションで高い性能を引き出すために演

算加速装置の利用が必須となる。このような中、東京大学情報基盤センターと筑波大学計算科学研究センターと共同で設置する最先端共同 HPC 基盤施設（JCAHPC）では、Oakforest-PACS（OFP）の後継機（OFP-II）を 2025 年 1 月に稼働予定であり、その OFP-II においても演算加速装置を導入する方向で検討を進めている。OFP をはじめとした両センターが運営に関わるスパコンでは、CPU に最適化された多数のアプリケーションが実行されている。OFP-II の稼働開始に向け、これらのアプリケーションを演算加速装置を導入した OFP-II へ効率的に移植していく指針の確立とその手法の構築が必須である。

本研究課題では、スパコンで動作している CPU アプリケーションを演算加速装置を搭載したスパコンへ移植し、その移植の方法を確立することを目的とする。様々な分野の研究者が開発した CPU アプリケーションを移植することを念頭に、演算加速装置で高い性能を達成することを目指しながらも、高性能計算分野でない研究者も難なく扱えるように、演算加速装置専用の開発言語を多用せず、標準的・汎用的手法を用いて移植を実現する。

本研究課題は 3 年計画で進めており、今年度はその 2 年目である。昨年度は、標準的な並列化手法が実用に耐えうるか、どの程度の性能が達成できるかを検証することに重点をおき、地震波伝播・強震動シミュレーションコード OpenSWPC を Fortran の標準並列化構文 DO CONCURRENT を用いて GPU 化した。ノード間通信は CUDA などの GPU 専用言語を用いる従来手法より性能が低下することが判明したが、これは昨年度課題で CUDA Fortran の機能を利用して明示的にデバイスメモリを確保した手法を導入することで解決した。

本年度は、当初の計画通り GPU 移植の対象としているもう一つのアプリケーションである海洋大循環モデル COCO の GPU 化を進める。

3. 当拠点の公募型共同研究として実施した意義

本研究課題は、既存のアプリケーションを演算加速装置へ移植し、その手法を汎用的な方法として確立する。これにはアプリケーション分野と高性能計算分野の専門家による密な連携が必須である。

アプリケーション分野の専門家として COCO の開発者、OpenSWPC の開発者が参画する。アプリケーションのコードや計算に必要なデータの提供とともに、データ構造に関する知見を提供する。また、アプリケーションの将来性を考えると、移植したコードはそれぞれのコミュニティで継続して開発できることが重要となる。これにはアプリケーション分野からの視点は必須であり、この役割を担う。高性能計算分野からは演算加速装置への最適化の専門家が参画し、コードの原型を保ちつつ、最大の性能を引き出す手法を検討する。我々はこれまでに CPU と GPU 両方で実行可能な性能可搬性が高いフレームワークの開発経験があり、この知見を生かす。このように様々な専門家が密に連携することにより、実用に耐えうる汎用的な移植手法の確立を目指す。

本研究課題は、演算加速装置として NVIDIA A100 GPU を対象とし、様々なスパコンにおける移植可能性を検証する計画であり、東京大学の Wisteria-Aquarius と大阪大学の SQUID (GPU)は本研究を遂行する上で最適な計算環境である。

このように本研究は当拠点公募型共同研究として計算環境を効果的に利用し、共同研究を遂行することで着実に成果を出すことができている。

4. 前年度までに得られた研究成果の概要

本研究課題は継続課題で、昨年度（2022 年度）から当公募型共同研究として実施している。本課題では、二つの実アプリケーション COCO と OpenSWPC を GPU 化する。昨年度は、コード規模が比較的小さい OpenSWPC の GPU 化を進めた。OpenSWPC は Fortran で記述され、OpenMP+MPI のハイブリッド並列化が既に行われている。検討の結果、Fortran の標準並列化構文 DO CONCURRENT で GPU 化できることがわかり、GPU 移植をおこなった。その結果、東京大学の Wisteria-Aquarius ノードを用いると、1 GPU で 8 CPU 程度の計算性能を達成できた。しかし、弱スケーリングであっても GPU 数が増えると、通信による性能劣化が顕著となることがわかり、その一因は意図しない CPU と GPU 間の通信であることが判明した。これは MPI と標準並列化構文を併用するアプリケーションにおいて、普遍的な課題であり、本研究課題では、2023 年度の前半にこの通信の高性能化を進める。

5. 今年度の研究成果の詳細

今年度は当初の計画通り、海洋大循環モデル COCO(https://ccsr.aori.u-tokyo.ac.jp/~hasumi/COCO/) の GPU 移植を進めた。また、これまで Fortran で開発されたアプリケーションの GPU 化に注力してきたが、当初の計画に加えて、C++で記述されたアプリケーションを対象に標準的・汎用的手法を用いて GPU 移植に関する研究を開始した。これを通して、C++言語で開発されたアプリケーションについて GPU への移植方法を検討した。

以下では、COCO の GPU 移植の状況と汎用的手法である SYCL を用いた C++で記述された拡散方程式コードの GPU 移植の性能評価について述べる。

5.1 DO CONCURRRENT 構文による COCO の GPU 化

海水・海氷熱循環モデル COCO の GPU 化を行った。このソフトウェアは多くの物理モデルを統合している大規模なアプリケーションである。COCO に対して今回用いた GPU 化手法は Fortran の標準並列化構文 do concurrent を用いるものである。COCO のコードの GPU 化対象部分は全て do 文となっているため、do concurrent や OpenACC,OpenMP などで十分高い性能が達成できることが期待できる。大規模なコードゆえ GPU 化の際にバグが混入する可能性も考えられるため、do concurrent と unified memory を利用する方法を選択した。

COCO では水平方向の XY 軸のインデックスを 1 つにマージしたデータ構造を採用しており、並列実行可能な do ループのほとんどは XY 軸と Z 軸の二重ループになっていて、以下の例のように Z 軸ループが外側である。

```
do k=kstr,kstr+kz-1
  do ij=ijtstr,ijtend
    処理
  end do
end do
```

XY 軸方向のループは常に並列実行可能であるが、Z 軸方向については並列実行可能な処理と不可能な処理の両方がある。並列実行可能な場合は二重ループをまとめて do concurrent 構文に変換する。外側ループが並列化できない場合に対しては内側ループだけ do concurrent 構文にすることで対応可能であるが、k ループの反復回数だけ GPU カーネルが呼び出されることになり効率が悪い。当該箇所については二つのループを入れ替えることが可能となっており、ループを入れ替えて外側になった ij ループを do concurrent 構文とすることによって GPU カーネルの呼び出し回数を削減でき、またメモリアクセスも削減することができる。

このような大規模アプリケーションの場

合は do concurrent 構文に書き換える編集コストも無視できない。OpenACC や OpenMP による GPU 化ではディレクティブの行を do 文の上に追加するだけであり、二重ループや三重ループでは collapse(2)、collapse(3)がつくこと以外は共通であるため編集コストは非常に低い。これに対して do 文を do concurrent 構文に変更する作業は文字単位での編集となり、また多重ループのマージには大幅な修正が必要となる。特に COCO のようにそれぞれのループの反復回数が微妙に異なるケースでは、コピー&ペーストに頼らず慎重に修正を行わなければいけない。このような差異を考えると、状況によって推奨される GPU 化手法は変わってくると言える。第三者が開発したコードを熟練者が GPU 化する場合には OpenACC が最適である。一方で自ら開発したコードを初心者が GPU 化する場合には Fortran の標準並列化構文は学習コストが極めて低く、ミスも起こりにくい。

ユーザにとっては GPU 化の目的は実行時間の短縮にのみあり、如何に高速に実行できるかが重要である。OpenACC や OpenMP、do concurrent を用いた場合では生成される GPU カーネルの性能に大きな差はないことは昨年度の OpenSWPC でも確認できている。COCO の GPU カーネルはメモリアクセスがボトルネックであるものが多く、より差が出にくいと考えられる。主として実行時間に影響するのはホストとデバイス間でのデータ転送である。do concurrent のように unified memory を利用する場合には関連する全ての処理を GPU 化することによってデータ転送が最小になる。自動的に unified memory が割り当てられるのは allocatable 属性がついている変数に限られる。固定サイズの配列等はこの属性がついていないため、allocatable 属性を追加して後で確保するように修正する必要がある。これはモジュール変数だけでなくサブルーチン内のローカル変数にも適用される。

MPI 通信に関するバッファには unified memory は適していない。高い通信性能を確保するためにはホストメモリやデバイスメモリ等、他のメモリを利用する必要がある。COCO の通信は一つのファイル、モジュールに集約されており、通信の前後で通信バッファに Pack/Unpack するという処理になっている。そこで通信バッファをデバイスメモリに変更し、Pack/Unpack を行う do concurrent 構文に OpenACC の data deviceptr ディレクティブで当該配列変数がデバイスポインタであることを明示するという方法で対応する。

```
!$acc data deviceptr(sdbfx1)
do concurrent(k=1:kdim,j=1:ny,i=1:nx)
  sdbfx1(i,j,k)=q1(i+istr-1,j+jstr-1,k)
end do
!$acc end data
```

その他の通信には Allreduce と Broadcast があるが、いずれも配列でデータではないため特にメモリの種類を調整していない。

以上のような方向性で GPU 化は定型化された作業となる。GPU 化されたコードはコンパイルには問題ないが、実行時にエラーが発生している。CPU で実行されている部分で Segmentation fault が発生しており、その分析を続けている。やや特殊な実装を用いている通信部分を一般的な unified memory を利用するコードに戻しても解消しない。デバッグ用のコードを追加していくと Segmentation fault が発生する場所が変わることから、何らかの理由で配列を格納するメモリが破壊されていることが疑われる。大規模アプリケーションであるためこの手の不具合の原因の特定は非常に難しいが、GPU 化を段階的に進めることができる unified memory の利点を生かして原因箇所を探していく予定である。

5.2 汎用的手法を用いた C++コードの GPU 移植

今年度は当初の計画に加えて、C++向け性能可搬性プログラミングモデルとして SYCL について評価を行った。

SYCL は、OpenCL を開発している Khronos Group により提供されている、ロイヤリティフリーのクロスアーキテクチャ向け抽象化レイヤーである。異種デバイスを一つのアプリケーションで、最新の ISO C++を用いて記述される。

まず SYCL のコードは、ホストデバイスで動作するホストコードとオフロード先のデバイスで動作するデバイスコードに分けられる。ホストコードで、オフロード先のデバイスを選択する Queue クラスを作成、Queue を submit することで、デバイスにオフロードする。デバイスコードは、下記の参考コードにおける、h.parallel for のラムダ式で表される部分となる。SYCL で 利用できる代表的なカーネルは、SIMD 的な並列処理を サポートする parallel for kernel（コード内 : parallel_for）と、パイプラインによる並列処理サポートする single task（コード内 : single_task）の 2 つである。

```
// キューの作成
queue q(d_selector, exception_handler);

buffer<int> a_buf(a.data());
buffer<int> b_buf(b.data());
buffer<int> sum_buf(sum.data());

// キューをデバイスに送り、データの移動
// やデバイスでの処理を書く。
q.submit([&](handler &h){
accessor a_vec(a_buf, h);
accessor b_vec(b_buf, h);
accessor sum_vec(sum_buf, h);

h.parallel_for(n,[=](nd_item<1> i){
    sum[i] = a[i] + b[i];
});
});
```

ここでは GPU にオフロードすることを考え、 parallel_for を採用した。

parallel_for では、ループ範囲の指定に 2 次元、3 次元の配列を指定できる。また、ループ範囲の指定には、nd_range クラスを使用することができる。 Nd_range クラスは、(global_range, local_range) で構成され、global_range はループ範囲を表し、local_range は各スレッドに割り当てられる work-group(CUDA の thread-block に対応）の大きさを表す。

Local_range の全体や各次元のサイズは、デバイス毎に指定できる大きさに制限があり、最適な local_range は異なる。コンパイル時に決定する必要はなく、実行時に値を渡すことができるため、CPU と GPU でコードを共有することが可能である。

ここでは、SYCL を用いて 3 次元の拡散方程式を実装し、性能評価を行う。

実行環境は、東京大学情報基盤センターが運用する Wisteria-Aquarius 上で行った。CPU は Intel Xeon Platinum 8360Y、GPU は NVIDIA A100 が搭載されており、SYCL コードのコンパイラーは Intel oneAPI DPC++/C++ Compiler 2023.0.0 である。各測定では、グリッドサイズ global_range は 計算領域サイズ(nx,ny,nz)と同じとする。測定は計算領域サイズ(128,128,128)、(256,256,256)、(512,512,512) の 3 つに対して行われる。先述した nd_range を利用しており、デバイス毎の制約は今回の環境の場合、CPU は local_range のサイズが 8192 まで、GPU は local_range の各次元 のサイズが 64 までとなっている。

まず、CPU での実行について述べる。Local_range に関しては、global_range が

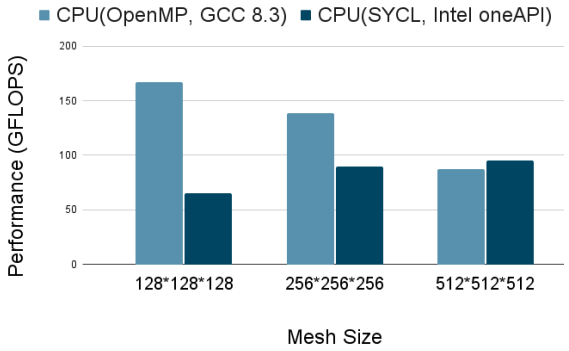

図 1 CPU 上での拡散方程式の実行性能

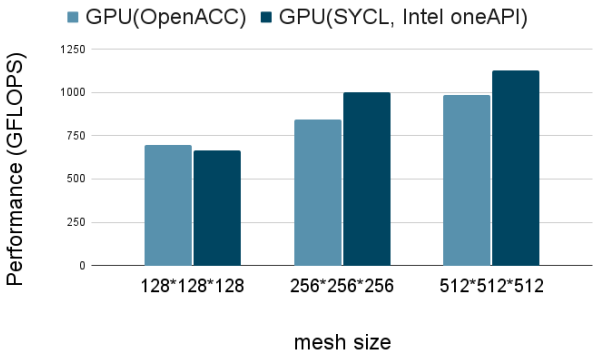

図 2 GPU 上での拡散方程式の実行性能

(128,128,128) の場合は (128,64,1)、(256,256,256) の場合は (256,32,1)、(512,512,512) の場合は (512,8,1) とした。CPU の性能は、OpenMP のコードと比較した。比較に用いた OpenMP のコードは、GCC8.3 でコンパイルしている。Intel Compiler も OpenMP には対応しているが、動作を試したところ GCC の場合よりも性能が低下したため、今回は GCC を選択した。

図 1 が CPU の場合の性能測定結果である。計算領域が小さい場合、つまり global_range が (128,128,128) のときには、SYCL では OpenMP の 3 分の 1 程度の性能にとどまった。ただ、計算領域が大きくなるとその差は縮まり、global_range が (512,512,512) の場合には、SYCL が OpenMP よりも高い性能を達成している。

次に、GPU での実行について述べる。Local_range は全ての計算で(64, 4, 1)とする。GPU の性能は、OpenACC のコードと比較した。コンパイル環境は NVIDIA HPC SDK Version 22.7 である。

図 2 は GPU での性能結果を表す。この性能測定結果を見ると、計算領域サイズが (128,128,128) の場合には、SYCL の性能が OpenACC の性能にほぼ匹敵している。それよりも計算領域サイズが大きくなると SYCL による実装の方が OpenACC による実装よりも性能が高くなっている。

6. 進捗状況の自己評価と今後の展望

今年度は、当初の計画通り、COCO の GPU 化を進めた。COCO は昨年度 GPU 化した OpenSWPC と比べて様々な計算処理が必要となる大規模で複雑なコードである。Fortran の標準並列化構文 do concurrent を用いて GPU 化を進めた。GPU 化されたコードはコンパイルには問題ないが、実行時に CPU で実行されている部分で Segmentation fault のエラーが発生しており、その分析を続けている。大規模アプリケーションであるためこの手の不具合の原因の特定は非常に難しく、GPU 化を段階的に進めることができる unified memory の利点を生かして原因箇所を調査中である。

並行して、当初の計画では Fortran で記述されたアプリケーションを主な対象としていたが、C++で記述されたアプリケーションにおいて標準的・汎用的手法を用いて GPU 移植する方法について検討を開始した。具体的には、拡散方程式のコードについて、SYCL を用いた GPU 移植についてコード開発と性能検証を開始した。SYCL で実装したコードを CPU および GPU で実行し、それぞれ OpenMP による実装および OpenACC による実装と同程度の性能を達成することを確認した。

来年度の前半は、COCO のエラーが発生している原因箇所を特定し、問題解決を図る。次

に、シミュレーションのデータ入出力を性能向上する手法を開発し、これを GPU へ移植中の COCO へ導入する。unified memory は GPU 化において高い生産性を実現するが、細かいメモリ管理ができないため、意図しない CPU と GPU 間の通信により、異なるノード間の GPU 間通信は性能劣化する。シミュレーションのデータ入出力は一般にノード間の GPU 間通信を伴うため、同様に性能劣化する。昨年度までの研究課題で、データを GPU または CPU メモリに固定することでこの通信劣化を回避する方法を開発した。これを応用しデータ入出力の性能を向上する方法を開発する。

並行して、C++言語で記述された格子ボルツマン法による流体アプリケーションを std::execution とユニファイドメモリを利用し GPU 化を行う。Fortran と同様に、C++でも意図しない CPU と GPU 間の通信による通信劣化が発生することがわかっており、C++に対応した性能向上手法を開発する。また、SYCL を用いた流体コードの GPU への移植を行う。SYCL で実装した流体アプリケーションの性能評価を進め最適化手法を検討する。

来年度の後半は、前半に引き続き、流体アプリケーションの最適化を進める。これまでに開発した地震波伝播・強震動シミュレーションコード OpenSWPC (https://github.com/tktmyd/OpenSWPC)、海洋大循環モデル COCO および流体アプリを最新の NVIDIA H100 GPU で実行し性能評価を行う。必要に応じて、H100 向けの最適化手法を開発する予定である。

7. 研究業績

(1) 学術論文 （査読あり）

該当なし

(2) 国際会議プロシーディングス （査読あり）

該当なし

(3) 国際会議発表（査読なし）

[1] Ziheng Yuan, Takashi Shimokawabe, “Accelerating lattice Boltzmann method with GPU and C++ standard parallelization”, 10th International Congress on Industrial and Applied Mathematics, Tokyo, Japan, August 2023.

[2] Tatsumasa Seimi, Akira Nukada, “Performance Evaluation of OpenSWPC using Various GPU Programming Methods”, International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia) 2024, January 2024.(Poster)

[3] Ziheng Yuan, Takashi Shimokawabe, “Accelerating Lattice Boltzmann method with C++ standard language parallel algorithm”, International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia) 2024, January 2024.(Poster)

(4) 国内会議発表（査読なし）

[4] 佐久間 大我、下川辺 隆史、大森 拓郎, “oneAPI を用いた様々なデバイス上でのステンシル計算の実装”, 第 28 回計算工学講演会, つくば, 2023 年 6 月.

[5] Ziheng Yuan, Takashi Shimokawabe, “Accelerating lattice Boltzmann method simulation with GPU computation using C++ standard language parallelism”, 第 28 回計算工学講演会, つくば, 2023 年 6 月.

(5) 公開したライブラリなど

該当なし

(6) その他（特許，プレスリリース，著書等）

[5] Ziheng Yuan, Best Student Poster Awards 3rd place, International Conference on High Performance Computing

in Asia-Pacific Region (HPCAsia) 2024,
January 2024.