Vol.2022-HPC-183 No.18
2022/3/18

OpenMPを用いたGPUオフローディングの有効性の評価

河合 直聡[1,a)]　三木 洋平[1]　星野 哲也[1]　塙 敏博[1]　中島 研吾[1,2]

概要：限られた電力，設置面積で最大の性能を得られるスーパーコンピュータシステムを実現するためには，GPU 等の演算加速装置の導入が不可避となりつつある．汎用 CPU 向けに Fortran や C/C++で記述され，OpenMP で並列化されたプログラムを CUDA 等を使って GPU 向けに書き直すことはコストがかかる．OpenMP にはバージョン 4.0 以降は GPU 等でプログラムを実行するオフローディング機能がサポートされている．本研究では，この機能を用いて，演算律速なアプリケーションである N 体計算およびメモリ律速なアプリケーションである ICCG 反復法に適用し，NVIDIA A100，AMD MI100 上での性能評価を実施した．結果，N 体計算では A100 上では CUDA 実装の 58.3%，MI100 上では HIP 実装の 71.9%の演算性能を確認した．また，ICCG 法では Stream Triad ベンチマークで計測したメモリースループットの 88%（A100）と 53%（MI100）を確認した．以上の結果から，OpenMP での GPU オフローディングは，MI100 上での ICCG 法を除いて，実用的な範囲と考える．

1. はじめに

近年の多くのスーパーコンピューターでは，GPU に代表される演算加速機を搭載したヘテロジニアスな環境が採用されている．これは，GPU の電力や単位面積あたりの性能が高いためである．実際に，TOP500 の 10 位までのシステムのうち 7 つ，Green500 に至っては 10 位までのうち 9 つが NVIDIA 製の GPU を搭載したシステムである [1], [2]．また，AMD 製 GPU や Intel 製 GPU を搭載したスーパーコンピューターの導入も発表されており [3], [4], [5]，GPU スーパーコンピューターについても多様化が進んでいくと予想される．

GPU の高い演算性能は千コア以上という大量の演算器を持っていることに由来しており，その性能を十分に発揮するための方法論はマルチコア CPU 向けの最適化手法とは異なる場合もある．したがって共有，分散メモリシステム向けの並列化などを経験しているユーザーであっても，プログラムの GPU 移植・最適化に要するコストは大きい．また，NVIDIA 製 GPU を対象とする場合には専用言語である CUDA を用いることで高性能なプログラムが実装できるが，GPU 内で動作する関数以外にも CPU 側からの GPU 関数の立ち上げ，GPU 内のメモリ管理，CPU-GPU 間のデータ転送などとユーザが記述すべき量は多い[*1]．また，CUDA は NVIDIA 製 GPU のみを対象とした専用言語であるため，他社製 GPU 上で動作しないというベンダーロックインの問題も生じる．

GPU を簡易に使う方法として OpenACC や OpenMP などの指示文ベースの記述方法がある．OpenACC や OpenMP では最も簡単な実装では GPU 化したいループの直前に指示文を挿入するだけで GPU 化が可能となる [6]．必要に応じて CPU-GPU 間のデータ通信や GPU 上でのメモリ確保も記述することで性能向上も期待できる．とりわけ，OpenMP は共有メモリシステム上での並列化のために広く利用されており，OpenMP での GPU へのオフローディングが簡易かつ効率的にできれば，GPU 化の敷居を大きく下げることができる．また，AMD 製 GPU や Intel 製 GPU を対象とした指示文ベースの簡易 GPU 化手法としては，OpenMP によるオフローディングの開発が主に進められている．したがって，OpenACC よりも OpenMP の方がベンダーロックインを避けやすいと言える．

本研究では，OpenMP での GPU オフローディングの有効性を演算およびメモリ律速なアプリケーションを対象に評価を行った．演算律速なアプリケーションとしては宇宙物理学で広く使われる N 体計算を，メモリ律速なアプリケーションとしては連立一次方程式を解く一般的な手法である不完全コレスキー前処理付き共役勾配 (ICCG) 法 [7] を採用した．評価では NVIDIA A100 および AMD MI100

[1] 東京大学 情報基盤センター
Information Technology Center, Kashiwa 277-0882, Japan
[2] 理化学研究所 計算科学研究センター
RIKEN R-CCS
[a)] kawai@cc.u-tokyo.ac.jp

[*1] Unified Memory の利用によって，データ管理に関する実装コストは低減できる．

の両方を用いた．

2. OpenMP を用いた GPU オフローディング

本節では OpenMP による GPU オフローディングについて説明する．Listing 1 に Fortran コードで記述した GPU オフローディングのサンプルコードを示す．

本サンプルコードで`$OMP target` 指示文と`$OMP end target` 指示文で囲まれた部分が GPU へオフロードされる範囲となる．コード内で do/for ループ（Fortran においては do ループ，C/C++では for ループ）で記述された計算を GPU 向けに並列化するためには，do/for ループの直前に指示文を挿入する．その際に挿入できる指示文は主に 2 つあり，1 つは`teams distribute parallel do/for`（Fortran 向けが do，C/C++向けが for），もう 1 つは`loop` である．両方を使用した場合の実装例を Listing 1 に示している．1 つ目の方法では`teams distribute` で teams（CUDA では Block, OpenACC では Gang に相当）単位の並列化を，`parallel do/for` で Thread（OepnACC では Worker）単位の並列化を指示している．場合によっては，`simd` 句を加えることで simd（OpenACC では vector）単位での並列指示も可能である．`loop` 指示節を使う場合には teams, Thread, SIMD のどのレベルで並列化するかなどは基本的にコンパイラ依存となり，コンパイラが最適と考えるレベルでの並列化を行う．また，`loop` 指示節による並列化では CPU と GPU ともに同じコード，同じバイナリで実行できるメリットもある．サンプルコードは 2 重ループの例を示しており，本研究で実際に評価しているアプリケーションでも 2 重ループを含んでいるが，このようなコードに対しては`teams distribute parallel do/for` を，`teams distribute` と `parallel do/for` を分けて付与する．または `loop` 指示節で示すように collapse を指定して並列化する方法も検証している．

Listing 1 OpenMP を用いた GPU オフローディングのサンプルコード．

```
!Copy data from/to device
!$OMP target data map(to:**, from:**, &
!$OMP                 tofrom:**, alloc:**)
!Offloaded inside of target clause
!$OMP target
!$OMP teams distribute
do i = ***
!$OMP parallel do simd
    !parallel "for" in C/C++
    do j = ****
        !Calculation
    enddo
enddo

!$OMP loop collapse(2)
do i = ***
    do j = ****
        !Calculation
    enddo
enddo

!$OMP end target
!$OMP end target data
```

CPU-GPU 間のデータ通信のやり方も 2 通りあり，コンパイラ任せにして記述しないパターンと，ユーザーが明記する方法がある．一般的にコンパイラに任せる方法は，コードの改良が do/for ループへの指示文挿入だけですむ一方で，GPU 上で動作する関数の入口・出口において CPU-GPU 間のデータ通信が発生するために低速になる場合が多い．ただし，Unified Memory を使えば過剰な CPU-GPU 間のデータ通信を削減できるため，`data` 指示文なしであっても性能低下を抑制できる．ユーザーが明記する場合には Listing 1 に示すように `target data` と `end target data` の範囲内で有効な配列を指定し，必要に応じて CPU-GPU 間のデータ移動についても制御する．

3. アプリケーション

本節では OpenMP を用いた GPU オフローディングの評価で使用したアプリケーションについて述べる．

3.1 N 体計算

演算律速なアプリケーションの代表例として，直接法に基づく N 体計算を取り上げる．粒子間に働く重力による i 番目の粒子の加速度 $\boldsymbol{a}_i$ は，粒子の質量 m_i と位置 $\boldsymbol{r}_i$ を用いて

$$\boldsymbol{a}_i = \sum_{j=0, j\neq i}^{N-1} \frac{G m_j (\boldsymbol{r}_j - \boldsymbol{r}_i)}{\left(|\boldsymbol{r}_j - \boldsymbol{r}_i|^2 + \epsilon^2 \right)^{3/2}} \qquad (1)$$

と与えられる．ここで G は重力定数であり，ϵ は Plummer ソフトニングにおけるソフトニング長である．式 (1) において重力を受ける粒子を i-粒子，重力を及ぼす粒子を j-粒子と呼び，それぞれの粒子群に関するコード内の for ループを i-ループおよび j-ループと呼ぶことにする．

直接法に基づく N 体計算においては，i-ループの内部に j-ループがネストした二重ループ構造となり，演算量は $\mathcal{O}(N^2)$ となる．粒子数が少ない場合を除いては i-ループのみを並列化対象とすることで GPU の高い演算性能が発揮でき，先行研究において CUDA を用いた NVIDIA 製 GPU 向け最適化 [8], [9], [10], [11], [12] や HIP による AMD 製 GPU 向け最適化 [12] がなされている．

本研究では，OpenMP を用いた GPU オフローディングによって実現される簡易 GPU 化を試み，NVIDIA A100 および AMD MI100 上での性能を測定する．また，NVIDIA 社が提供する NVIDIA HPC SDK（旧 PGI コンパイラ）においては，OpenACC による GPU オフローディングや C++17 の標準言語による GPU 化もサポートされてい

Listing 2 OpenMP を用いた GPU オフローディング（N 体計算）.

```
#pragma omp target teams distribute parallel
    for simd thread_limit(NTHREADS)
for(int32_t ii = 0; ii < Ni; ii++){
```

Listing 3 OpenACC を用いた GPU オフローディング（N 体計算）.

```
#pragma acc kernels
#pragma acc loop independent vector(NTHREADS)
for(int32_t ii = 0; ii < Ni; ii++){
```

Listing 4 C++17 を用いた GPU オフローディング（N 体計算）.

```
std::for_each_n(std::execution::par, boost::
    iterators::counting_iterator<int32_t>(0),
    Ni, [=](int32_t ii){
```

る [13] ため，A100 上ではこうした手法との比較も行う．ベースとしたコードは [12] において実装した HIP 版の N 体計算コードであり，まずはこのコードを通常の C++実装による CPU コードへと書き替えた．この際に，cuRAND あるいは rocRAND を用いて GPU 上で生成していた初期条件については，C++における標準ライブラリである `std::mt19937` を用いて CPU 上で生成することとした．

OpenMP を用いて i-ループをオフロードする際には，Listing 2 に示したように for ループに `target` 指示文を追加した．スレッドブロックあたりのスレッド数を `NTHREADS` とするようコンパイラに示唆するために，`thread_limit` 指示節を使用した．また，GPU 上に確保すべき配列については `map` 指示文を用いて明示的に指定し，CPU-GPU 間のデータ転送についても `update` 指示文を用いて `hipMemcpy()` などによる実装を置き換えた．本研究では `loop` 指示節を用いた GPU 化も試みたが，NVIDIA HPC SDK 22.1 においては `nvc++`コンパイラはエラーを吐いたため，また ROCm 5.0.0 においては `amdclang++`は `loop` 指示節に未対応であるために，評価対象には含められなかった．

OpenACC を用いて i-ループをオフロードする際には，Listing 3 に示した標準的な実装を採用し，`vector` 指示節を用いてスレッドブロックあたりのスレッド数を `NTHREADS` とするよう示唆した．OpenMP 版と同様に，CPU-GPU 間のデータ転送については指示文を用いて明示的に実装している．

C++17 の標準言語としての機能を用いての GPU 化については，Listing 4 のように for 文をラムダ関数へと置き換えた．コンパイル時に`-stdpar=gpu` を渡すと GPU 向けに，`-stdpar=multicore` を渡すとマルチコア CPU 向けに ISO C++17 で提供される Parallel Algorithms が有効化される．ただし NVIDIA HPC SDK に含まれる `nvc++`においては，`-stdpar=gpu` を渡した際には自動的に CUDA 関連のファイルもインクルードされる挙動となっている．したがって`-stdpar=multicore` 指定を念頭にユーザ側で `float4` 型を `typedef` すると，CUDA 側の定義と衝突しコンパイルエラーとなるため，インクルードガードの設定も必要となる．

3.2 ICCG 反復法

次に，メモリ律速なアプリケーションの代表例として，不完全コレスキー分解前処理付き共役勾配 (ICCG) 法を取り上げる．ICCG 法は疎な係数行列を持つ連立一次方程式を解く代表的な反復法の 1 つであり，大きな条件数の係数行列の問題に対しても他の手法と比較して高い堅牢性を持つため，広く利用されている．ここで取り上げるのは P3D アプリケーション [14] 内の ICCG 法である．P3D では熱拡散方程式を構造格子での有限体積法を用いて離散化しており，導出された連立一次方程式を ICCG は法で解いている．離散化の結果，格子間の依存関係は 7 点ステンシルとよばれる一般的な形状となっており，これに伴って係数行列内の非零要素も 1 行あたり最大 7 個となっている．

ICCG 法の計算は主に，前進後退代入，行列ベクトル積，ベクトルの内積に分けられる．今回の実装では，これらすべてを OpenMP を用いて GPU へオフロードした．Listing 5 に IC 前処理内の前進代入計算を，OpenMP を用いてオフロードしたサンプルを示す．なお，IC 前処理の並列化は Reverse Cuthill-Mckee と Cyclic-Multicoloring(CM-RCM) で行っており，N 色で色分けした場合の，色番号が 1 の前進代入の 1 行あたりの非零要素は対角成分のみ，色番号が 2〜$N-1$ の場合は 4 要素，N の場合は 7 要素となるようになっている．後退代入の場合の非ゼロ要素数はこの逆となる．計算に使用する係数行列の格納形式は，CPU での実績が十分にある Sell-C-σ[15][16] を採用している．Listing 1 に示したのと同様に，外側のループに `teams distribute` を，内側のループに `parallel do` を付与している．なお，`simd` 句に関しては本実装では効果がなかったため，付与していない．この実装に加えて，A100 では `teams distribute` および `parallel do` を `loop` に置き換えた実装も用意した．次に，Listing 6 に Listing 5 から collapse 句を加えた実装のループ制御部のみを示す．ループの collapse により，ループ長が長くなり，条件によっては高速化が期待できる．なお，collapse をするためにはループが完全なネストになっている必要があり，Listing 5 の実装では $ib0$ の計算が collapse の阻害要素となるため，最内ループに移動させている．また，Listing 5 の実装と同様に，`teams distribute parallel do` 句を `loop` に置き換えた実装も用意した．

評価ではさらに `target map` 指示文を挿入せずに，A100 で Unified Memory を有効にした場合と `target map` 指示

Listing 5 前進代入の GPU オフロード (parallel do 版).

```
do ic = 2, NCOLORtot-1
!$OMP target teams distribute private(ib0)
    do ib = range(ic), range(ic+1)-1
        ib0 = (ib-range(ic))*CHUNK
!$OMP parallel do private(i, VAL)
        do is = 1, CHUNK
            i = idx(ic) + ib0 + is
            VAL = Z(i)
            VAL = VAL - AL_pre(ib0+is,1,ip0) &
                      * Z(itemL(ib0+is,1,ip0))
            VAL = VAL - AL_pre(ib0+is,2,ip0) &
                      * Z(itemL(ib0+is,2,ip0))
            VAL = VAL - AL_pre(ib0+is,3,ip0) &
                      * Z(itemL(ib0+is,3,ip0))
            Z(i) = VAL * D(i)
        enddo
    enddo
enddo
```

Listing 6 前進代入の GPU オフロード (collapse 追加版).

```
do ic = 2, NCOLORtot-1
!$OMP target teams distribute parallel do &
!$OMP       collapse(2) private(ib0, i, VAL)
    do ib = range(ic), range(ic+1)-1
        do is = 1, CHUNK
            ib0 = (ib-range(ic))*CHUNK
            i = idx(ic) + ib0 + is
```

文を挿入した場合の評価も行っていく．また，MI100 の方では flang を元に Fortran に対応している状態であり，flang がリリースされてから日が浅いという現状を鑑みて，C 版の実装も用意し評価を行った．

比較対象として N 体計算の場合と同様に OpenACC の実装も用意，評価した．なお，OpenACC の実装は Listing 6 の `loop` 句を `loop independent` に置き換えた形になっている．

4. 結果

4.1 評価環境

表 1 に評価で使用した環境を示す．A100，MI100 ともに同一構成のサーバーに PCI-Express 接続で搭載されており，開発環境は A100 では CUDA 11.6 と NVIDIA HPC SDK 22.1 を，MI100 では ROCm 5.0.0 を使用している．各アプリケーションの実行条件等は次節で述べる．

4.2 N 体計算

本研究で用いた N 体計算コードにおける浮動小数点演算は，全て単精度である．CUDA 実装および HIP 実装においては，GPU 向けの最適化を一切施していない実装を opt0，最適な逆数平方根命令（A100 向けには `rsqrtf()`，MI100 向けには`__frsqrt_rn()`）を使用した実装を opt1，さらにシェアードメモリを使用しなおかつループアンローリングも施した実装を opt2 とした．HIP 実装においては，シェアードメモリを使用する opt2 よりも opt1 の方が高速であった．また，CUDA 実装・HIP 実装ともに命令レベルの並列性を導入すればさらに性能が向上する [12] が，効果は小さい上に簡易 GPU 化コードと比較する上では高度すぎる最適化となるため本研究では適用しない．

GPU を用いたコードにおいては，スレッドブロックあたりのスレッド数が性能を決める重要なパラメータとなる．したがって，指示文経由で示唆するスレッド数を変えた計算を試行することとした．C++17 Parallel Algorithms を用いた GPU 化においては，ユーザ側でスレッド数を示唆する手段が提供されていないため，スレッド数に関する調整は施していない．また，NVIDIA HPC SDK を用いる際には`-Mfpapprox=rsqrt`（または`-Mfpapprox`）や `fastmath` を渡すかどうか，`amdclang++`によってコンパイルする際には最適化オプションとして`-funroll-loops -ffast-math` を渡すかといった点も変えながら比較した（**表 2**）．

粒子数 N を $2^{22} = 4\,194\,304$ とした測定を各パターン 5 回ずつ実施し，一番実行時間が短かった結果を**図 1** および **表 3** に示した．実行性能を算出する際には，相互作用あたりの浮動小数点演算数を 22 Flops と仮定した [12]．

A100 上では，OpenMP，OpenACC，C++17 による簡易 GPU 化手法による性能はほぼ一致しており，GPU 向けの最適化を施していない CUDA 実装（opt0）の 90% を超える性能を示した．一方で，GPU 向けの最適化を施した CUDA 実装（opt2）と比較すると 55% 程度の性能であり，性能差は大きい．

MI100 向けに OpenMP オフローディングを実施した際には HIP 実装の opt0 よりも高い性能が得られており，特に`-funroll-loops -ffast-math` の指定によって 1.85 倍高速化されることが分かった．また，ROCm 4.5.2 使用時には最適化フラグの無指定時に 106 s（3.64 TFlop/s），指定時に 88.0 s（4.40 TFlop/s）であったため，ROCm 5.0.0 へのバージョンアップによってそれぞれ 1.55 倍，2.37 倍高速化された．スレッド数の最適値は 512 であったが，64 スレッドにおいては 135 s（2.87 TFlop/s），128 スレッドにおいては 63.8 s（6.07 TFlop/s），256 スレッドおよび 1024 スレッドにおいては 37.3 s（10.4 TFlop/s）となり，`thread_limit` 指示節を通じて示唆したスレッド数を十分に大きな値にしておくことが重要であった．

AMD 製 GPU 向けに `amdclang++`が提供する OpenMP によって GPU オフロードしたコードにおいては，実行時に環境変数として `LIBOMPTARGET_KERNEL_TRACE=1` を指定することで，スレッドブロック構造を把握できる．これによって，`thread_limit` 指示節で示唆したとおりのスレッド数が使われており，またチーム数（CUDA/HIP 用語におけるブロック数）が 480 であり CU（Compute Unit）数

表 1 性能評価環境.

		A100 搭載サーバ	MI100 搭載サーバ
CPU	モデル	AMD EPYC 7713	AMD EPYC 7713
	構成	64 cores, 2 sockets	64 cores, 2 sockets
	動作クロック	1.5 GHz–3.7 GHz	1.5 GHz–3.7 GHz
GPU	モデル	NVIDIA A100 80GB PCIe	AMD Instinct MI100
	ピーク演算性能 (単精度)	19.5 TFlop/s	23.1 TFlop/s
	メモリバンド幅	1953 GiB/s	1229 GiB/s
開発環境		CUDA 11.6 NVIDIA HPC SDK 22.1	ROCm 5.0.0

表 2 N 体計算コードの実装モデル・使用 GPU ごとのコンパイルコマンド.

使用言語	GPU	コンパイルコマンド
OpenMP	A100	`nvc++ -mp=gpu -target=gpu -gpu=cc80[,fastmath] -O3 [-Mfpapprox[=rsqrt]]`
OpenMP	MI100	`amdclang++ -target x86_64-pc-linux-gnu -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908 -O3 [-funroll-loops -ffast-math]`
OpenACC	A100	`nvc++ -acc -target=gpu -gpu=cc80[,fastmath] -O3 [-Mfpapprox[=rsqrt]]`
C++17	A100	`nvc++ -stdpar=gpu -target=gpu -gpu=cc80[,fastmath] -O3 [-Mfpapprox[=rsqrt]]`
CUDA	A100	`nvcc -gencode arch=compute_80,code=sm_80`
HIP	MI100	`hipcc --offload-arch=gfx908`

の 4 倍である[*2]ことが分かった．HIP 実装においては，各 CU に大量のブロックを割り当てることで CU 内で同時実行可能な演算数を増やし，各種処理のレイテンシの隠蔽につなげている．したがって，スレッド数を 64 とした際にも極端な性能低下は起こっていなかった．OpenMP 版においてスレッド数の増加に伴い性能が大きく向上したという結果は，CU あたりのブロック数が少ないために，主にブロック内のスレッド切り替えによってレイテンシ隠蔽が促進されたためだと考えられる．

OpenMP を用いて GPU オフローディングしたコードは，A100 上では GPU 向けの最適化を施していない CUDA 実装（opt0）の 93.6%，GPU 向けの最適化を施した CUDA 実装（opt2）の 58.3% の性能であった．MI100 上においては GPU 向けの最適化を施していない HIP 実装（opt0）の 196%，GPU 向けの最適化を施した HIP 実装（opt1）の 71.9% の性能であった（最適化フラグ指定時）．用いる GPU に依らずに，GPU 向けの最適化を施していない実装と比較すれば十分な性能が実現できているが，対象 GPU 向けの最適化を施したコードと比較すると著しい性能低下が見られる．

4.3 ICCG 法

ICCG 法で解く問題は立方体形状の領域であり，自由度は $256^3 = 16\,777\,216$ としている．ICCG 法の IC 前処理は 10 色の CM-RCM 法を元に並列化されており，相対残差が 10^{-8} を下回った場合に近似解を得たとして反復計算を停止している．行列およびベクトルのデータ型はすべて倍精度であり，行列の格納形式は 3.2 節で述べたように Sell-C-σ を採用している．Sell-C-σ のパラメータである Chunk size は 128，σ は 1 としている．プログラムのコンパイルに使用したコンパイラおよびオプションは**表 4** に示している．

図 2 には各指示文, 言語の A100, MI100 での実行時間を，**図 3** にはメモリースループットを Stream benchmark[17] の結果とともに示している．各実装のメモリースループの計測は A100 では NVIDIA Nsight Systems（nv-nsight-cu コマンド）を使用して行っており，MI100 では A100 での計測結果から計算時間を加味して算出した．なお，MI100 上での collapse を付与していない C 言語版の実装では，Core Dump を出力して結果を取得できなかったため，未掲載としている．

A100 での計算時間を見ると，Unified Memory を使用した parallel do/for 実装を除いたすべての実装が同程度の計算時間となっている．今回評価した手法においては，loop 指示文を挿入して Unified Memory でデータ転送を行う手法が最も簡易であるが，最も高速な実装である OpenACC の data 指示文ありと比較しても 82%の性能が出ており，メモリースループットは 1238 GiB/s と Stream triad ベンチマークの結果 (1572 GiB/s) の 79%の性能が出ているため，十分に実用的だと考えられる．OpenMP において最も高速なオフロード手法は，Fortran の `loop` 指示文に `data map` 指示文も挿入し，CPU-GPU 間の通信を明示的に行う場合であり，計算時間は OpenACC と比較して 93%，Stream Triad Benchmark と比較して 88%の性能を確認した．

MI100 の結果を見ると，まず C 言語と Fortran の計算時間の差が大きくでており，MI100 での Fortran の使用は，

[*2] ROCm 4.5.2 においてはチーム数が 120 であり，CU 数と一致していた．

表 3 重力計算に要する実行時間および実行性能の実装モデル・使用 GPU ごとの比較.

測定対象	使用 GPU	実行時間	実行性能	opt0 との性能比	最適化実装との性能比
OpenMP	A100	48.8 s	7.93 TFlop/s	0.936	0.546
OpenACC	A100	48.2 s	8.03 TFlop/s	0.948	0.553
C++17	A100	48.3 s	8.02 TFlop/s	0.946	0.552
CUDA (opt0)	A100	45.7 s	8.48 TFlop/s	–	0.583
CUDA (opt1)	A100	28.3 s	13.7 TFlop/s	1.62	0.943
CUDA (opt2)	A100	26.6 s	14.5 TFlop/s	1.71	–
OpenMP (w/o flg)	MI100	68.7 s	5.63 TFlop/s	1.06	0.389
OpenMP (w/ flg)	MI100	37.2 s	10.4 TFlop/s	1.96	0.719
HIP (opt0)	MI100	72.8 s	5.31 TFlop/s	–	0.367
HIP (opt1)	MI100	26.7 s	14.5 TFlop/s	2.73	–

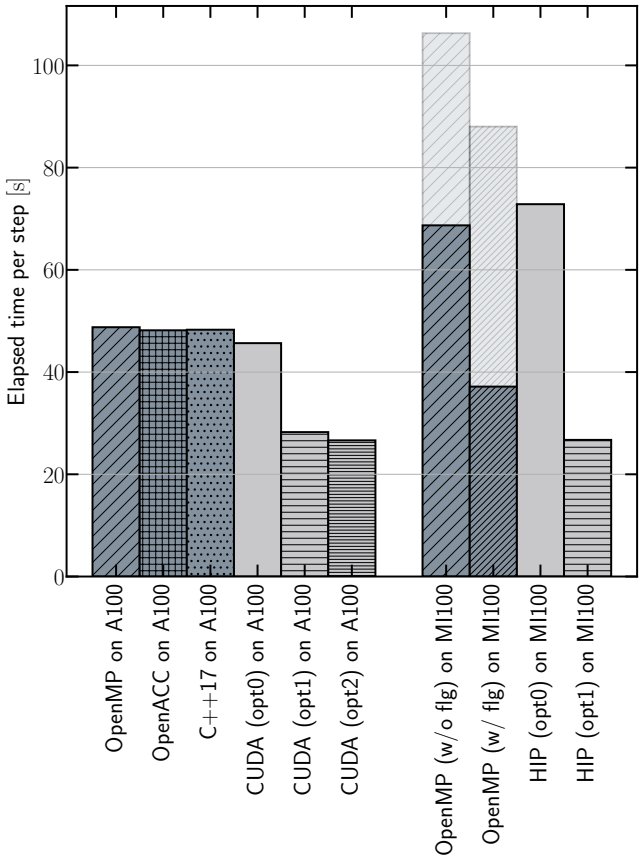

図 1 重力計算に要する実行時間．粒子数 N は $2^{22} = 4\,194\,304$ とした．NVIDIA A100 上では OpenMP, OpenACC, C++17 による簡易 GPU 化版と CUDA を用いた結果を比較した．CUDA 版においては，GPU 向けの最適化を一切施していない実装（opt0），最適な逆数平方根命令（`rsqrtf()`）を使用した実装（opt1），さらにシェアードメモリを使用しなおかつループアンローリングも施した実装（opt2）を用いた測定結果を示した．AMD MI100 上では OpenMP を用いたオフローディング版および HIP を用いた実装の結果を示した．OpenMP 版については最適化フラグ（`-funroll-loops -ffast-math`）の有無および，ROCm 4.5.2 における結果（薄く描画した棒グラフ）を比較した．HIP 版においては，GPU 向けの最適化を一切施していない実装（opt0）と最適な逆数平方根命令（`__frsqrt_rn()`）を使用した実装（opt1）の結果を示した．

現状では非推奨と言える．C 言語の実装は Fortran よりも高速であるが，Stream Triad Benchmark の結果と比較すると 53%と，依然として低い．

5. 議論

5.1 簡易 GPU 化による性能低下の起源

本研究では OpenMP を用いた簡易 GPU 化を試みたが，N 体計算においては CUDA や HIP を用いて十分な最適化を施したコードに匹敵する性能は得られなかった．本節では，今後のコンパイラ開発に示唆を与えるために性能低下の起源を議論しておく．

N 体計算においては逆数平方根の実行時間の寄与が大きいため，適切な逆数平方根演算命令を使用することが重要である．CUDA および HIP を用いた実装の比較によって，単精度浮動小数点演算による N 体計算においては，A100 上では `rsqrtf()`，MI100 上では `__frsqrt_rn()` を用いることで高速化されることが分かっている [12].

A100 向けに `nvc++`を用いてコンパイルする際には，近似命令である `rsqrtf()` の使用を期待して`-Mfpapprox=rsqrt` あるいは`-Mfpapprox`，`fastmath` の付与を試行した．しかし，実際に測定したところ実行時間は変化しなかった．一方で，CUDA 版の opt0 のコードに対して`-Mfpapprox` に相当する`--prec-div=false --prec-sqrt=false` を渡した際には，`rsqrtf()` を明示的に指定した opt1 相当の性能が得られており，`rsqrtf()` の使用につながっていることが確かめられた．本研究における試行ではうまく働かなかったが，`-Mfpapprox` 指定によって簡易 GPU 化コードにおいても `rsqrtf()` が使用されれば opt1 相当の高い演算性能が発揮できると期待される．

MI100 においては，最適化フラグの有無によって性能が大きく変化した．両者の実行ファイルのアセンブリを `llvm-objdump` コマンドを用いて出力したところ，もともと `sqrtss` 命令を用いた処理が実行されていた部分が，最適化フラグの付与によって `rsqrtss` 命令を用いた処理に変わっていることが分かった．つまり，高速な逆数平方根演算命令の使用によって高速化されたということが確かめら

表 4　ICCG 法の実装モデル・使用 GPU ごとのコンパイルコマンド.

使用言語	GPU	コンパイルコマンド
Fortran+OpenMP	A100	`nvfortran -O3 -mp=gpu -gpu=cc80[,managed]`
	MI100	`amdflang -O3 -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa`
C+OpenMP	A100	`nvc++ -O3 -mp=gpu -gpu=cc80[,managed]`
	MI100	`amdclang -O3 -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa`
Fortran+OpenACC	A100	`nvfortran -O3 -acc -ta=tesla,cc80[,managed]`

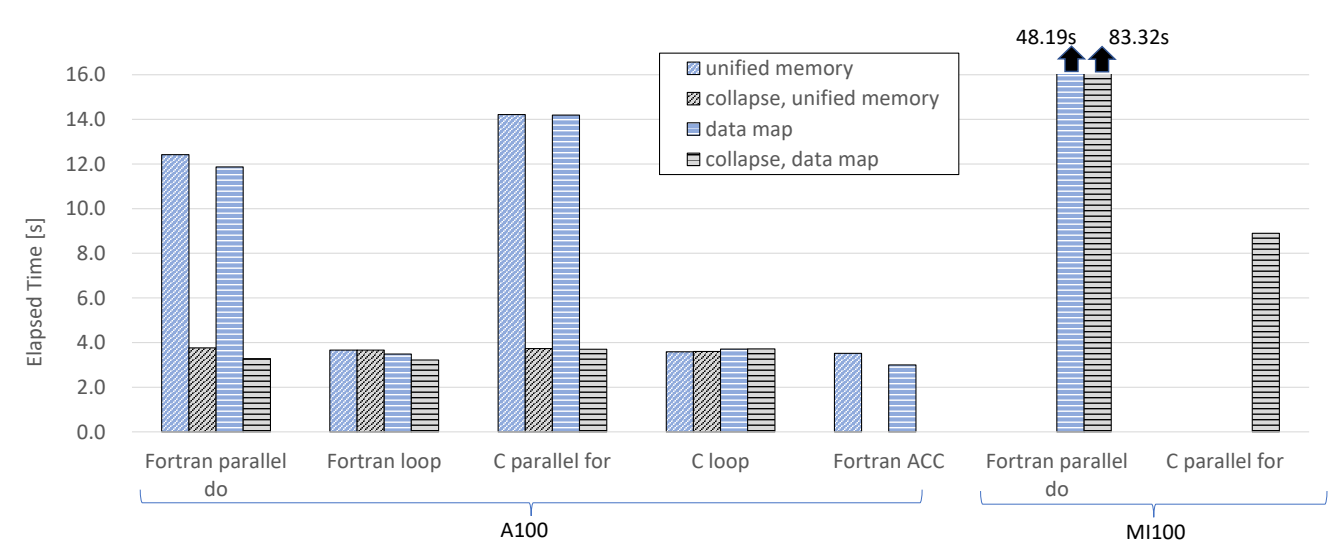

図 2　ICCG 法での近似解を得るまでに要する時間.

れた.

一方で，MI100 上では `std::fma()` を用いて記述した FMA（fused multiply-add）命令が `mulss` と `addss` に分解されて実行されていることも分かった．これは 1 サイクルで 2 Flops 分の演算ができる FMA 命令の処理を，実際には 2 サイクルかけて実行しているということであり，性能低下の要因となる．N 体計算のカーネルにおいては減算 3 回，乗算 3 回，FMA 命令 6 回，逆数平方根 1 回が実行される．[12] による議論から逆数平方根に要するサイクル数を 3 サイクルと仮定すると，FMA 命令を 1 サイクルで実行すれば全部で 15 サイクル，FMA 命令を乗算と加算に展開して 2 サイクルで実行した際には 21 サイクルを要することとなる．したがって OpenMP 版の性能は本来期待される性能の 15/21 ∼ 0.714 倍に低下していると見積もられる．この数字は OpenMP 版と HIP（opt1）の性能比である 0.719（表 3）とよく一致している．また，コンパイル時に `-ffp-contract=fast` も指定した測定を行ったが，FMA 命令の発行にはつながらず性能も変化しなかった.

5.2　NVIDIA HPC SDK が提供する簡易 GPU 化の比較

図 1 および表 3 に示したように，A100 上において OpenMP，OpenACC，C++17 を用いて簡易的に GPU 化したコードの性能はほぼ一致した．ただし，各実装において生成されたバイナリが一致しているわけではないので，`-Minfo` 指定により出力されるコンパイル時のログやプロファイラの出力から分かることをまとめておく.

OpenACC 使用時には，`-Minfo=accel,opt` を指定してメッセージを取得した．スレッドブロックあたりのスレッド数は，`vector` 指示節を通じて示唆した値と一致することが確認できた．一方で `float4` 型の変数については “Local memory used” というメッセージが出力されていた．しかし，プロファイラを用いて確認したところ Local Memory Per Thread は 0 bytes となっていたため，実際にはローカルメモリは使用されていなかった.

OpenMP 使用時には`-Minfo=accel,opt,mp` を指定してメッセージを取得し，対象ループが GPU 上で動作するようにコンパイルされていることが確認できた．スレッドブロックあたりのスレッド数については，`thread_limit` 指示節を用いて示唆したスレッド数ではなく，デフォルトで使用される 128 となっているというメッセージが得られた．ユーザが動作させたいスレッド数を確実に指定する方法がない以上仕方無い側面はあるが，最適なスレッド数を指定できないことによる性能低下は避けられない．また，ローカルメモリを使用したという旨のメッセージは出力されておらず，プロファイラを用いてもローカルメモリが使

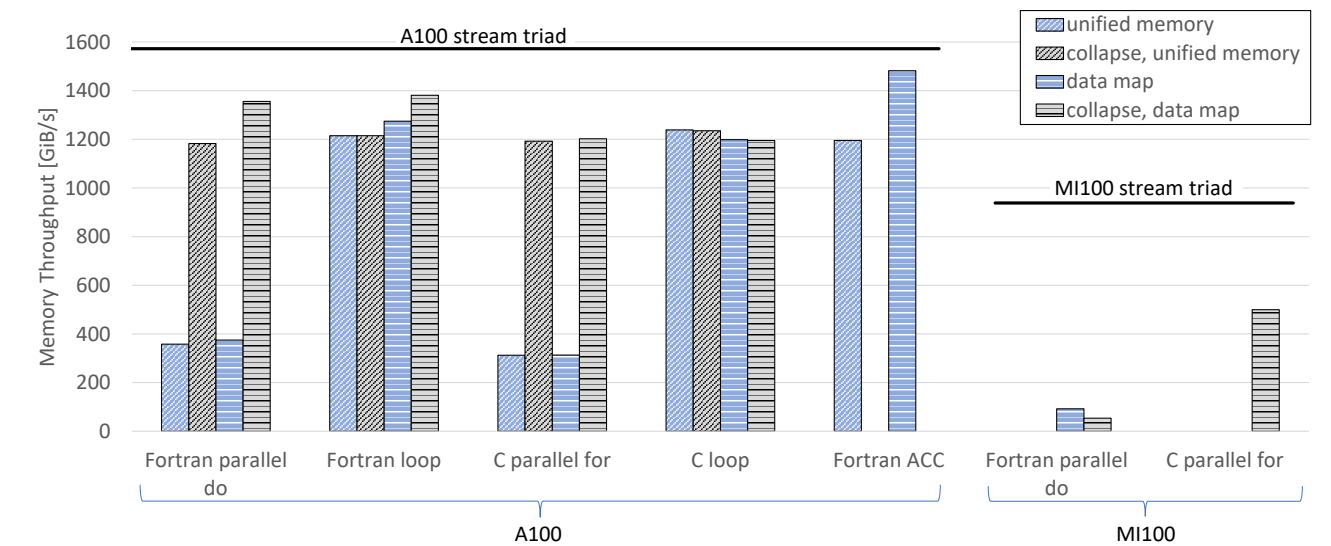

図 3 ICCG 法での各実装のメモリースループット．

用されていないことが確認できた．

C++17 使用時には`-Minfo=accel,opt,stdpar` を指定してメッセージを取得した．OpenMP 同様に対象のラムダ関数が GPU 上で動作するようコンパイルされたことは確認できたが，用いたスレッド数については出力されなかった．プロファイラを用いて実際の動作状況を確認したところ，thrust 配下で動作しており，スレッド数は 256 でありローカルメモリの使用もないことが分かった．

OpenACC 版のコードにおいては 512 スレッド指定時の性能が最も良かった．これに対して C++17 版では 256 スレッド，OpenMP 版では 128 スレッドで動作しており，この違いが性能差の要因であると考えられる．

6. まとめ

本論文では A100 および MI100 上での OpenMP による計算のオフローディングの有効性について，演算律速なアプリケーションである N 体計算と，メモリ律速なアプリケーションである ICCG 法を対象にして検討した．N 体計算の結果では，OpenMP を用いた GPU オフローディングによって，A100 上で CUDA 実装の 54.6%，MI100 上で HIP 実装の 71.9% の性能が達成できた．ICCG 法の結果では，A100 上の `loop` 指示文+Unified memory の最も簡易な実装で，OpenACC 実装の 82%，Stream Benchmark の 77%を達成しており，`loop` 指示文に `data map` 指示文も挿入した実装で，OpenACC 実装の 93%，Stream Benchmark の 88%を達成した．これらの結果から，OpenMP による GPU オフローディングの簡易さも加味すれば，A100 上では実用的なレベルに達しつつあるといえる．一方で，MI100 上でのメモリ律速なアプリケーションでは，十分な性能が発揮できておらず，今後のアップデートに期待がかかる．

謝辞 本研究は JSPS 科研費 JP20H00580, JP20K14517, JP19H05662 および JP18K18059 の助成を受けた．

参考文献

[1] TOP500.org: TOP500 Lists, (online), available from ⟨https://www.top500.org/lists/top500/⟩ (2021).
[2] TOP500.org: Green500 Lists, (online), available from ⟨https://www.top500.org/lists/green500/⟩ (2021).
[3] Oak Ridge National Laboratory: Frontier, (online), available from ⟨https://www.olcf.ornl.gov/frontier/⟩ (2021).
[4] Lawrence Livermore National Laboratory: LLNL and HPE to partner with AMD on El Capitan, projected as world's fastest supercomputer, (online), available from ⟨https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer⟩ (2020).
[5] Argonne Leadership Computing Facility: Aurora, (online), available from ⟨https://www.alcf.anl.gov/aurora/⟩ (2021).
[6] Daley, C. S., Southwell, A., Gayatri, R., Biersdorfff, S., Toepfer, C., Özen, G. and Wright, N. J.: Non-Recurring Engineering (NRE) Best Practices: A Case Study with the NERSC/NVIDIA OpenMP Contract, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, Association for Computing Machinery (2021).
[7] Saad, Y.: *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2nd edition (2003).
[8] Hamada, T. and Iitaka, T.: The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units, *ArXiv Astrophysics e-prints* (2007).
[9] Nyland, L., Harris, M. and Prins, J.: Fast N-Body Simulation with CUDA (2007).
[10] Miki, Y., Takahashi, D. and Mori, M.: A Fast Implementation and Performance Analysis of Collisionless N-body Code Based on GPGPU, *Procedia Computer Science*, Vol. 9, pp. 96 – 105 (online), DOI: http://dx.doi.org/10.1016/j.procs.2012.04.011 (2012).
[11] Miki, Y., Takahashi, D. and Mori, M.: Highly scalable

implementation of an N-body code on a GPU cluster, *Computer Physics Communications*, Vol. 184, pp. 2159–2168 (online), DOI: 10.1016/j.cpc.2013.04.011 (2013).

[12] 三木洋平，塙　敏博：AMD MI100 に向けた N 体計算コードの移植と性能評価，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2021-HPC-182, No. 2, pp. 1–10（オンライン），入手先 ⟨http://id.nii.ac.jp/1001/00214099/⟩ (2021).

[13] Olsen, D., Lopez, G. and Lelbach, B. A.: Accelerating Standard C++ with GPUs Using stdpar, (online), available from ⟨https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/⟩ (2020).

[14] Sakamoto, R., Kondo, M., Fujita, K., Ichimura, T. and Nakajima, K.: The Effectiveness of Low-Precision Floating Arithmetic on Numerical Codes: A Case Study on Power Consumption, *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2020, New York, NY, USA, Association for Computing Machinery, p. 199–206 (2020).

[15] Kreutzer, M., Hager, G., Wellein, G., Fehske, H. and Bishop, A. R.: A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units, *SIAM Journal on Scientific Computing*, Vol. 36, No. 5, pp. C401–C423 (2014).

[16] Kawai, M. and Nakajima, K.: Low/Adaptive Precision Computation in Preconditioned Iterative Solvers for Ill-Conditioned Problems, *International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2022, Association for Computing Machinery, p. 30–40 (2022).

[17] Ben Cumming: Cuda Stream, (online), available from ⟨https://github.com/bcumming/cuda-stream⟩ (2017).