Vol.2021-HPC-182 No.2
2021/12/6

AMD MI100に向けたN体計算コードの移植と性能評価

三木 洋平[1,a)]　塙 敏博[1]

概要：国内・海外を問わず今まで導入・運用されてきたGPUスパコンはそのほとんどがNVIDIA製GPUを搭載してきたが，今後はAMD製GPUやIntel製GPUを搭載したGPUスパコンの普及も予想される．したがって将来導入されるGPUスパコンがNVIDIA製GPU，AMD製GPU，Intel製GPUのいずれを搭載したとしても対応できるようなソフトウェア開発・最適化を進めていかなければならない．本研究ではNVIDIA製GPU向けにCUDAで実装された既存のN体計算コードを，AMD製GPU上でも動作可能となるようにROCm/HIPに移植した上で性能評価を行った．AMD製GPUであるMI100向けの性能最適化としては逆数平方根の計算に`__frsqrt_rn()`命令を用いることが重要であり，相互作用あたりの浮動小数点演算数として単精度22Flopsを仮定すると，最大14.6TFlop/sという高い演算性能を発揮した．NVIDIA製GPUであるA100上では最大14.5TFlop/sであり，両GPUの発揮した性能はほぼ同じであった．またA100上でCUDA版コードとHIP版コードの性能差を測定したところ，性能差は存在しない，つまりHIP実装によるオーバーヘッドは存在しないか十分に小さいということが分かった．

1. はじめに

スーパーコンピュータにGPUが演算加速器として搭載されるようになって10年程度が経過し，最新のTOP500ではGPUを搭載したスパコンのシステム数が100を越えている[25]．東京大学情報基盤センターが運用するWisteria/BDEC-01のデータ・学習ノード群Aquariusもこうした GPU搭載型スパコンであり，NVIDIA A100を全360枚搭載している[26], [28], [29]．他にもPerlmutter[17], ABCI[27]など多数の例があるが，演算加速器搭載型スパコンの多くはNVIDIA製GPUを搭載しており，例外はMN-Coreを搭載したMN-3[23]，AMD Vega 20を搭載したSugon TC6000[*1]などの数システムに限られる．したがって，GPUスパコン向けにコードを開発・性能最適化する際にはNVIDIA製GPUのことだけを考えれば良いという状況が長く続いていた．

今後数年間のうちに導入されるスパコンの情報が発表され，NVIDIA製ではないGPUを搭載したシステムが複数導入されることが分かった．特に象徴的なアメリカのエクサスケール・スパコンにおいては，Frontier[22]およびEl Capitan[13]はAMD製GPUを，Aurora[4]はIntel製GPU（Ponte Vecchio）を搭載することが発表されている．一方でAlps[7]のようにNVIDIA製GPUを搭載することが発表されたシステムもあり，今後のGPUスパコンは多様化が進んでいくと予測される．したがって今まで進めてきたNVIDIA製GPU向けのソフトウェア開発・性能最適化に加えて，AMD製GPUやIntel製GPUもカバーできるような研究開発も行い，今後多様化するであろうGPU環境への準備を進めていくことが重要である．

NVIDIA製GPU向けのプログラミング手法としては，CUDAによる実装やOpenACCを用いた指示文ベースの実装およびその組み合わせが主流である．高性能を得るために詳細な最適化を施したい場合にはCUDAを用いた実装が，CPUコードを少ない手間でGPU化したい場合には指示文ベースの実装が適している．AMD製GPUやIntel製GPUを対象とした開発環境を考える際にも，この2つの異なる需要を両方満たせるかという観点が重要であろう．AMDが提供する開発環境ROCm（Radeon Open Compute）[3]やIntelが提供する開発環境oneAPI[11]は，各社製のGPUに加えてNVIDIA製GPUにも対応できるよう開発が進められている．両環境ともにOpenMPのアクセラレータ指示文をサポートしており，これに加えてAMDはHIP（Heterogeneous-Compute Interface for Portability），IntelはData Parallel C++を用いたより高度な演算加速器向けの開発環境を提供している．

本研究では，NVIDIA製GPU向けにCUDAを用いて実装された既存のN体計算コードをAMD製GPU上でも動作可能となるようROCm/HIPに移植し，その性能評価をAMD製GPUとNVIDIA製GPU両方の上で行った．2節

[1] 東京大学 情報基盤センター
[a)] ymiki@cc.u-tokyo.ac.jp

[*1] `https://www.top500.org/system/179801/`

Listing 1 __global__関数の立ち上げ方法.

```
// CUDA
kernel<<<dim3(gridDim), dim3(blockDim), dShmem
    , stream>>>(var0, var1);

// HIP
hipLaunchKernelGGL(kernel, dim3(gridDim), dim3
    (blockDim), dShmem, stream, var0, var1);
```

において AMD 製 GPU 向けの開発環境を紹介し，AMD MI100 と NVIDIA A100 の簡単な比較（3 節）の後で N 体計算の実装および性能最適化内容を紹介する（4 節）．その後 5 節において性能評価結果を報告し，6 節では詳細な測定やサイクル数に基づいて実行性能について議論する．

2. AMD 製 GPU 向けの開発環境 ROCm

AMD 製 GPU 向けの開発環境として，ROCm が提供されている [3]．GPU 向けのプログラミングモデルとしては HIP および OpenMP を用いた指示文ベースの GPU 化がサポートされている．LLVM ベースのコンパイラである ROCmCC コンパイラが提供されており，HIP，OpenMP，OpenCL に対応している．ROCm の中にはコンパイラ以外にも GPU 向けに最適化されたライブラリなども含まれており，ROCm は NVIDIA の CUDA Toolkit や HPC SDK，Intel の oneAPI に相当する開発環境と言える．

本研究では，CUDA と同レベルに詳細な実装が可能となる HIP に注目する．HIP は C++に基づいて開発されており，AMD 製 GPU と NVIDIA 製 GPU 両方で動作するコードを実装できる．AMD 製 GPU 上でのランタイムは ROCclr（Radeon Open Compute Common Language Runtime）であり，HIP はヘッダファイル及び HIP-Clang コンパイラによって構築されたライブラリを提供する．NVIDIA 製 GPU 上では，HIP の API を CUDA における API へと変換するためのヘッダファイルを提供し，バイナリは CUDA のランタイムを用いて実行される．以下では HIP と CUDA の違い（2.1 節）および CUDA から HIP への移植ツール（2.2 節）について紹介する．

2.1 HIP と CUDA の違い

メモリ確保などのホスト上で実行する関数や事前設定されている変数については，概ね cuda*となっているものを hip*と置き換えれば良い．具体的には cudaMalloc()，cudaMemcpy()，cudaDeviceSynchronize() に対応して hipMalloc()，hipMemcpy()，hipDeviceSynchronize() が提供されている．ただし，全ての関数や変数がこの対応関係に従うことは保証されない．例えば cudaMallocHost() に対応する関数は hipHostMalloc() であり，hipMallocHost() の使用は非推奨となっている．

CUDA においてデバイス上で動作する関数につける修飾子である__global__や__device__は HIP においても同じである．HIP において__global__関数を呼び出す際には，hipLaunchKernelGGL を用いる．実装例は Listing 1 に示したとおりであり，dShmem はシェアードメモリを動的に確保する際の容量，stream はストリームの ID である．CUDA における実装では dShmem および stream はデフォルト値の 0 としておく場合に限って省略可能だが，HIP による実装では末尾の引数ではないため省略不可能である．

CUDA においてデバイス関数を実装する際に使う変数である threadIdx.x や warpSize などは，HIP においても同じ変数名のまま提供されている．シェアードメモリを使用するには，デバイス関数内での配列宣言時に__shared__修飾子をつければよく，これも CUDA と同じである．スレッドブロック内の全スレッドを同期するための__syncthreads() も CUDA と同様に使える．ただし AMD 製 GPU（のうち HPC を対象とした CDNA 系）においては warpSize の値が 64 となっており，NVIDIA 製 GPU における 32 とは異なっているため，この点には注意が必要である．

HIP においては，CUDA 9 で導入された Independent Thread Scheduling という動作モードは存在しない．これは AMD の GPU において，ワープ内の処理が暗黙の内に同期された状態で実行されるため，ユーザがワープ内のスレッドを明示的に同期する場面がないことを示している．NVIDIA の GPU においても，この動作モードを無効化した方が高速になる場合もある [16], [30] ため，実装の容易さ・性能向上という観点で利点であると考えられる．一方で NVIDIA の GPU を使用する場合には，ワープ内のスレッドを明示的に同期するために__syncwarp() を発行する場面はある．Independent Thread Scheduling に対応しない HIP には__syncwarp() も存在しないため，この場合には部分的に CUDA を用いて記述することとなる．

ワープシャッフル命令は HIP においても提供されているが，Independent Thread Scheduling がないことに由来して，__shfl() や__shfl_xor() などの関数が提供されている．CUDA 9 以降においてはマスク値とともに使用するワープシャッフル命令である__shfl_sync() などを使うため，前述した warpSize の値の違いもあわせて対象 GPU ごとに注意深く実装する必要がある．

NVIDIA Ampere 世代及び CUDA 11 において warp reduce 命令や cuda/pipeline 経由での memcpy_async() 命令といった新機能が導入された．こうした CUDA における新機能はまだ HIP では対応されておらず，AMD 製 GPU においてハードウェア的な機能追加がなされるかも含めて今後の開発動向を注視する必要がある．

GPU は継続的に開発されてきたハードウェアであり，世代が進むごとに提供される機能も増えてきた．このため NVIDIA の GPU 向けにコード開発をする際には#if __CUDA_ARCH__ >= 800 などとするマクロ制御を

Listing 2 NVIDIA 製 GPU 向けの HIP 環境変数の設定.

```
setenv  HIP_COMPILER    nvcc
setenv  HIP_PLATFORM    nvidia
setenv  HIP_RUNTIME     cuda
```

利用し，複数世代の GPU 上で動作可能であり，新機能に対応している場合には活用するという実装がなされてきた．HIP における実装では，ワープシャッフル命令が使えるデバイスを対象としてコンパイルした際には__HIP_ARCH_HAS_WARP_SHUFFLE__が define されるというように，機能ごとにフラグが設定される．したがって，使いたい機能が提供されているかだけを条件判定に用いるため，CUDA よりも直観的に実装できる構成となっている．

NVIDIA 製 GPU と AMD 製 GPU のどちらを対象とするかによって，提供されている機能や最適化方針が変わってくるため，コードの切り替えは必須となる．このためには，動作させるプラットフォームに応じて自動的に定義されるマクロ__HIP_PLATFORM_NVIDIA__および__HIP_PLATFORM_AMD__を用いれば良い．例えば, NVIDIA 製 GPU 上で動作する際には__syncwarp() 命令の発行が必要とされる場面があり，__HIP_PLATFORM_NVIDIA__が define されている場合にのみ__syncwarp() を記述すればこれを実現できる．

最後に HIP で実装したコードをコンパイルするコマンドだが，MI100 向けには hipcc --offload-arch=gfx908 source.cpp とすれば良い．ここで gfx908 は MI100 のコード名であり，mygpu コマンドで調べることもできる．同じコードを A100 向けにコンパイルする際には，hipcc -gencode arch=compute_80,code=sm_80 source.cpp とする．このコマンドは CUDA コードを A100 向けにコンパイルする際のコマンドのうち，nvcc を hipcc に置き換えただけのものである．NVIDIA 製 GPU 向けの HIP 環境を構築した際の手順によっては，コンパイル時のバックエンドに AMD 製 GPU 向けの環境を使おうとしてエラーとなる場合がある．今回用いた A100 搭載サーバにおいては，Listing 2 に示した環境変数の指定を追加し，常に適切なバックエンドが使用されるように設定した．

2.2 Hipify: CUDA から HIP への変換ツール

既存の CUDA コードから HIP への変換を支援するためのツールとして hipify が提供されている．この移植ツール群には hipify-perl や hipify-clang といった複数のツールが含まれている．このうち hipify-perl は単純な文字列変換（例えば cuda を hip に置き換えるなど）を行う簡易的なツールである．もう一方の hipify-clang はよりコンパイラに近いツールであり，マクロ関数などもチェックする，変換失敗時にはエラーを出力するといった複数の利点がある．

表 1 AMD MI100 と NVIDIA A100 の比較.

	MI100（PCIe）	A100 (40GB PCIe)
単精度理論ピーク性能	23.1 TFlop/s	19.5 TFlop/s
FP32 コア数	7680	6912
	(= 120 × 64)	(= 108 × 64)
動作周波数	1.502 GHz	1.41 GHz
グローバルメモリ容量	HBM2 32 GB	HBM2 40 GB
メモリバンド幅	1228.8 GB/s	1555 GB/s
L2 キャッシュ容量	8 MB	40 MB
シェアードメモリ容量	64 KB	164 KB
熱設計電力（TDP）	300 W	250 W

本研究においては，既に CUDA で実装済みであった複数バージョンの N 体計算コードを hipify-clang を用いて HIP に変換するという手順を取った．この際のコマンドは hipify-clang *.cu *.cuh --cuda-path=/cuda/path -I/include/path といったものであり，多くのコードの変換に成功した．本研究において自動変換に失敗したのは，以下の 2 通りである．

(1) マクロスイッチを用いたコード
HIP への変換時にはマクロスイッチ#ifdef ... #else ... #endif 中の#ifdef などを無視するようであり，const をつけた変数を複数回定義している旨のエラーを出力して変換に失敗した．HIP 変換後のコードにおいて有効化するフラグを切り換える場合を想定すれば仕方ない振る舞いであり，ここではマクロスイッチを用いないコードを複数用意してそれぞれを変換した後で，両者をマージした．

(2) CUDA 版ライブラリと ROCm 版ライブラリで不整合があった場合
HIP への変換は成功したように見えるものの，hipcc での AMD の GPU 向けコンパイルは通らずに実行できないという問題が起こった．GPU 上での初期条件生成関数中で cuRAND を使用した CUDA 実装を，hipify-clang は hipRAND/rocRAND を使用する HIP コードに変換した．この際に，cuRAND と hipRAND/rocRAND において定義されたマクロ名が異なるという不整合があったため，変換後の HIP コードは一部不正なものであった．名前の不整合があったのは主に Mersenne Twister 関連であり，適宜修正することで NVIDIA の GPU でも AMD の GPU でも正しく動作するようになった．

3. AMD MI100 と NVIDIA A100 の比較

表 1 に，AMD Instinct MI100 アクセラレーター [1] と NVIDIA A100 Tensor コア GPU[19] の比較を示す．NVIDIA A100 には，フォームファクターについては SXM4 版と PCIe 版，グローバルメモリについては HBM2 版と HBM2e 版で全 4 通りの製品があるが，ここでは PCIe 版

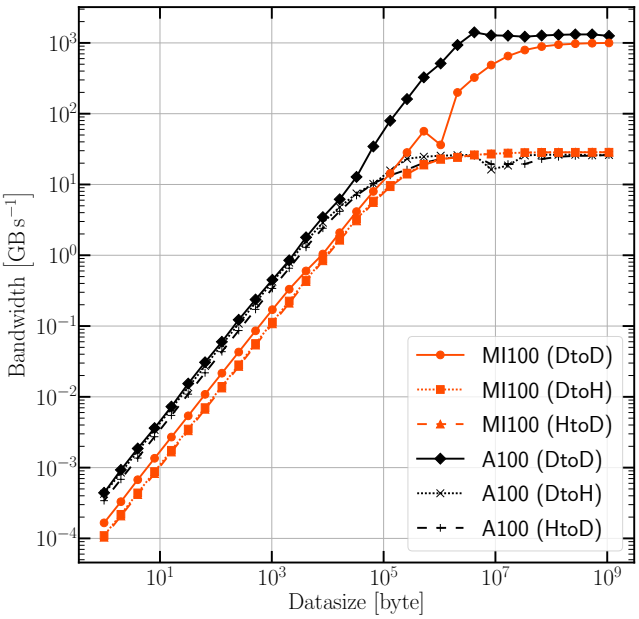

図 1 MI100 と A100 のメモリバンド幅.

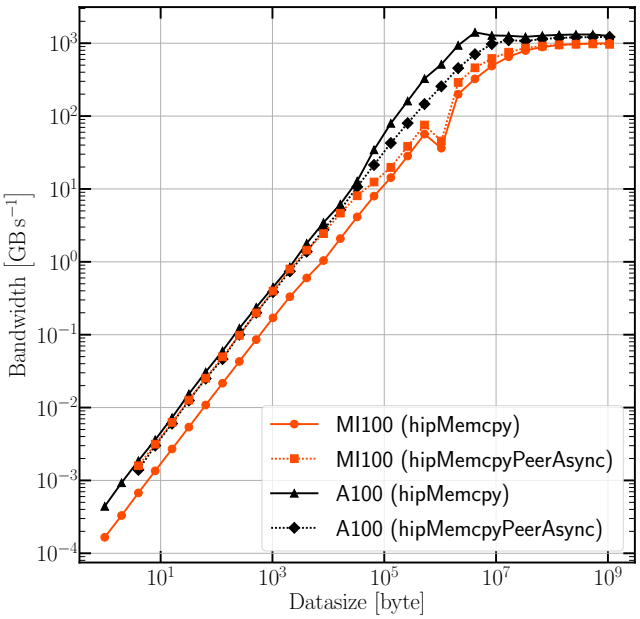

図 2 MI100 と A100 のメモリバンド幅（hipMemcpy と hipMemcpyPeerAsync の比較）.

かつ HBM2 版のみが提供されている MI100 と最も近いモデルである 40GB PCIe を比較対象とした．以下に見るように，AMD MI100 と NVIDIA A100 は同等の性能を有するものの，浮動小数点演算性能については MI100 が，メモリ性能については A100 が優位である．

単精度浮動小数点演算コアについては，MI100 については 64 コアからなる Computing Unit（以下 CU）が 120，A100 については 64 個の CUDA コアからなる Streaming Multiprocessor（以下 SM）が 108 搭載されており，MI100 の方が 1 割コアが多い．動作周波数についても MI100 の方が高く，単精度浮動小数点演算の理論ピーク性能は MI100 が 23.1 TFlop/s，A100 が 19.5 TFlop/s と MI100 の方が高い．

メモリ性能については，MI100 が HBM2 を 4 スタック搭載するのに対し A100 は 5 スタック搭載しており，容量・メモリバンド幅ともに A100 の方が優位である．また L2 キャッシュの容量やシェアードメモリの最大容量を見ても A100 の方が高く，特に L2 キャッシュの容量差が顕著である．

MI100 と A100 のメモリバンド幅を測定した結果を，図 1 及び図 2 に示す[*2]．GPU 内においては，ピークメモリバンド幅の太い A100 の方が高性能となった．MI100 においては，データサイズが 1 MiB となる点においてメモリバンド幅が極端に落ち込んでいるが，この理由は不明である．またデータサイズが小さい領域においては，MI100 では hipMemcpy と hipMemcpyPeerAsync の性能差が顕著である．ホスト・デバイス間通信のピークメモリバンド幅は MI100 の方が A100 よりも太いものの，ほぼ同等の性能を示した．またデータサイズが小さい領域においては，A100 のバンド幅の方が MI100 のものよりも太いという結果になった．

[*2] 本測定時には，CUDA 11.5 及び ROCm 4.5.0 を用いた．

表 2 性能評価環境.

	MI100 搭載サーバ	A100 搭載サーバ
CPU	AMD EPYC 7662 64 cores, 2 sockets 2.0 GHz–3.3 GHz	AMD EPYC 7662 64 cores, 2 sockets 2.0 GHz–3.3 GHz
GPU	AMD Instinct MI100	NVIDIA A100 40GB PCIe
開発環境	ROCm 4.3.0	ROCm 4.3.0, CUDA 11.4

4. N 体計算コードの実装と最適化

直接法を用いた N 体計算では，粒子間に働く重力計算

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

に計算時間のほとんどを費す．ここで m_i, $\boldsymbol{r}_i$, $\boldsymbol{a}_i$ はそれぞれ i 番目の粒子の質量，位置，加速度であり，また G は重力定数である．重力ソフトニングとしては Plummer ソフトニングを採用し，ϵ がソフトニング長である．(1) 式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，それぞれの粒子数を N_i, N_j と書くこととする．

NVIDIA 製 GPU を対象とした実装は多数あり，十分な知見がたまっている [8], [14], [15], [21]．Listing 3 に HIP を用いての重力計算カーネルのベースライン実装の一部を示すが，ここで示した部分については CUDA 実装と厳密に同じコードとなっている．

以降では，適切な逆数平方根計算命令の指定（4.1 節），シェアードメモリの使用（4.2 節），命令レベルの並列性の導入（4.3 節）といった最適化手法を紹介する．動作テスト及び性能測定については，表 2 に示した環境を用いた．両

Listing 3 ベースライン実装

```
__global__ void calc_acc(const int Ni, float4
    *ipos, float4 *iacc, const int Nj, float4
    *jpos, const float eps){
    const int ii = blockIdx.x * blockDim.x +
        threadIdx.x;
    float4 pi = ipos[ii];
    pi.w = eps * eps;
    float4 ai = {0.0f, 0.0f, 0.0f, 0.0f};

    for(int jj = 0; jj < Nj; jj++){
        const position pj = jpos[jj];

        float4 rji;
        rji.x = pj.x - pi.x;
        rji.y = pj.y - pi.y;
        rji.z = pj.z - pi.z;
        const float r2 = fmaf(rji.z, rji.z,
            fmaf(rji.y, rji.y, fmaf(rji.x, rji
            .x, pi.w)));
        rji.w = 1.0f / sqrtf(r2);
        rji.w *= rji.w * rji.w;
        rji.w *= pj.w;

        ai.x = fmaf(rji.x, rji.w, ai.x);
        ai.y = fmaf(rji.y, rji.w, ai.y);
        ai.z = fmaf(rji.z, rji.w, ai.z);
    }
    iacc[ii] = ai;
}
```

表 3 逆数平方根の計算方法と N 体計算の演算性能の関係.

逆数平方根の計算方法	MI100	A100 (HIP)
`1.0f / sqrtf(r2)`	5.326 TFlop/s	8.289 TFlop/s
`rsqrtf(r2)`	11.25 TFlop/s	13.51 TFlop/s
`__frsqrt_rn(r2)`	14.38 TFlop/s	8.744 TFlop/s

環境のホスト部分の構成は同じであり，搭載している GPU だけが異なるため，両 GPU の比較に適している．

4.1 逆数平方根の計算方法

重力計算カーネル内で必要となる浮動小数点演算は，減算 3 回，乗算 3 回，FMA（Fused Multiply-Add）命令 6 回と，逆数平方根の計算 1 回である．このうち逆数平方根の計算は実装方法が複数あり，なおかつ N 体計算全体の実行時間への寄与も大きいため，特に重要である．

言語仕様に依らない実装としては，`1.0f / sqrtf(val)` として，平方根を計算した後に除算を計算するという方法がある．CUDA においては `rsqrtf(val)` や `__frsqrt_rn(val)` といった逆数平方根を直接計算する命令が提供されており，これらは HIP においても同様に使用できる．NVIDIA の GPU 上では，`rsqrtf(val)` の最大 ulp = 2 であり，`__frsqrt_rn(val)` は IEEE-754 compliant である [20]．HIP においては `__frsqrt_rn(val)` はデバイス上でのみ実行可能な組み込み関数として提供されている [2]．

逆数平方根の計算方法とスレッドブロックあたりのスレッド数を変えて N 体計算の演算性能を測定した結果を表 3 に示す．粒子数 N は $N = 2^{22} = 4\,194\,304$ で固定し，各設定において 10 回測定した最高値を示した．相互作用あたりの浮動小数点演算数としては本稿を通じて 22 Flops を仮定しており，この仮定の妥当性については 6.2 節において議論する．MI100 においては `__frsqrt_rn(val)` 使用時に 14.4 TFlop/s となり，最適化前に比べて 2.7 倍高速になることが分かった．この速度向上率については，6.3 節において議論する．A100 においては `rsqrtf(val)` 使用時が 13.5 TFlop/s と最も高速であり，CUDA 使用時と同じ結果であった．

4.2 シェアードメモリの使用

NVIDIA の GPU をターゲットとして N 体計算を高速化する際には，シェアードメモリの使用が有効である．ただし A100 においては，L2 キャッシュの容量増加などの改良によって，シェアードメモリ未使用でも高い性能が発揮できるようになっている．

シェアードメモリを使用する際においては，シェアードメモリに対するロード・ストア命令をワープ単位で処理する場合とスレッドブロック単位で処理する場合の 2 通りが考えられる．前者はワープ内の暗黙の同期を利用した実装であり明示的な同期命令を発行する必要はないが，後者については `__syncthreads()` 命令を適切に発行してシェアードメモリ上のデータの一貫性をユーザが保証しなければならない．一方でグローバルメモリからのロード量は後者の実装の方が少なくなることもあり，直接法を採用した N 体計算コードにおいては後者の実装の方が高速となる．

4.3 命令レベルの並列性の導入

Kepler 世代以降の NVIDIA の GPU では，各ワープスケジューラに複数の命令ディスパッチユニットが搭載されている．このため，連続する命令が互いに独立に実行できるよう実装することで性能が向上することがある．ただし，実際の動作はアウト・オブ・オーダー実行であるため，命令レベルの並列性の導入が必ず性能向上につながるという保証はない．

本研究では，実行命令を 2^n 個（$n = 1,\ 2,\ 3, 4$）の連続した命令に展開するマクロ関数を実装し，N_j に関するループに対して命令レベルの並列性を導入した．

4.4 最適化手法の選択とパラメータ設定

ここまで紹介してきた最適化手法を適用するかやスレッド数などのパラメータの値を設定するため，各構成における測定を粒子数 $N = 2^{22} = 4\,194\,304$ において行った．HIP 版・CUDA 版ともにまずは 3 通りの逆数平方根を用いたベースライン実装の測定を行い適切な逆数平方根を決

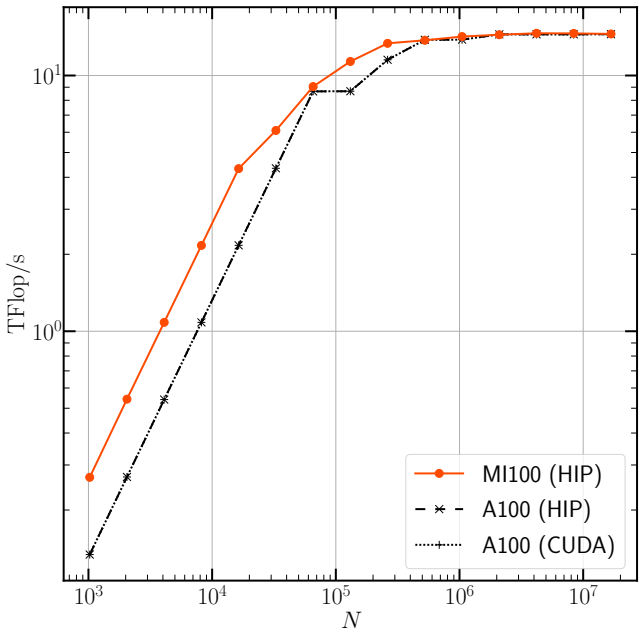

図 3 N 体計算コードの単精度浮動小数点演算性能．相互作用あたりの浮動小数点演算数を 22 Flops と仮定し，測定した単精度浮動小数点演算性能を粒子数 N に対してプロットした．A100 上での測定結果は，HIP 版（破線）と CUDA 版（点線）での性能がよく一致しているため，ほぼ重なって表示されている．

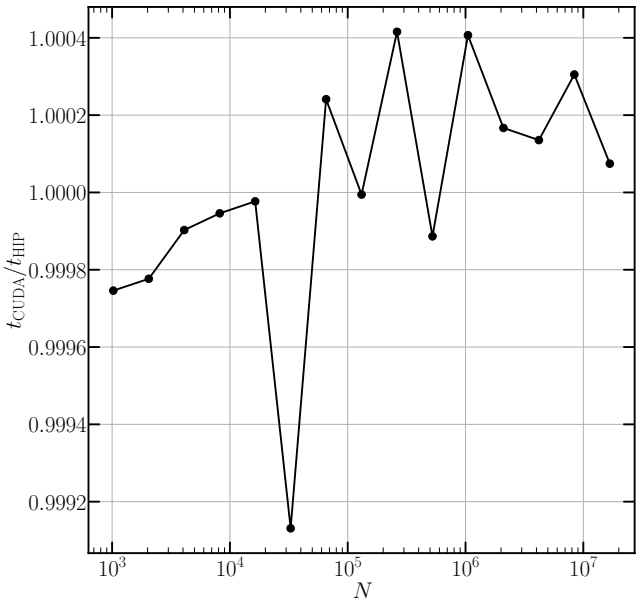

図 4 HIP 実装と CUDA 実装における性能比較．A100 上で測定した CUDA 版の実行時間 t_{CUDA} と HIP 版の実行時間 t_{HIP} の比を，粒子数 N に対してプロットした．

定した（4.1 節）．その後命令レベルの並列性を導入した実装について測定した．以降の測定では全てシェアードメモリを利用し，同期単位をブロック単位とするものとワープ単位とするもの，ダブルバッファ実装とする最適化の適用・不適用，命令レベルの並列性の導入・非導入の計 8 通りのバージョンを試行した．また，CUDA 版においては `memcpy_async()` を用いた実装も実験した．

この一連の測定において最高性能を発揮した設定を最終構成として採用した．MI100 向けの実装では逆数平方根は `__frsqrt_rn(val)` で計算し，スレッドブロックあたりのスレッド数を 256，命令レベルの並列性は 4 とした．シェアードメモリを使用しない方が高速であり，またこれに付随してループアンローリングも適用していない．A100 向けの実装では逆数平方根は `rsqrtf(val)` で計算し，スレッドブロックあたりのスレッド数を 1024，命令レベルの並列性は 4 とした．シェアードメモリはブロック単位で使用するシングルバッファ版とし，ループアンローリングを適用しないコードが一番高速であった．また，本構成は CUDA 版と HIP 版で共通であった．

5. 性能評価

ほとんどの演算器が待機状態になる問題サイズから GPU の演算器を使い切り演算性能を飽和させられるだけの十分な問題サイズまでをカバーした性能評価を行うため，粒子数 N は 1024 から 16 777 216 まで 2 倍ずつ増加させることとした．各粒子数における測定は 100 回行い，その最高値を図 3 に示した．また，中央値も含めた測定結果を表 A·1 に記載した．

得られた最高性能は MI100 において 14.6 TFlop/s，A100 において 14.5 TFlop/s であり，両 GPU でほぼ同じ値が得られた．また，A100 を用いた測定においては，HIP 版と CUDA 版両方の結果を示しているが，両者は非常によく一致している．

粒子数が少ない領域での振る舞いを議論する際には，スレッドブロックあたりのスレッド数と CU 数（または SM 数）が重要である．MI100 では CU 数が 120 でありスレッド数を 256 としたので，$N < N_{\mathrm{fill}} \equiv 120 \times 256 = 30720$ においては全てのコアを動作させられない．この領域では粒子数を増やすほど動作するコア数が増えるため，実行性能が粒子数 N に比例する．A100 においては SM 数が 108 でありスレッド数を 1024 としたので，粒子数に対するスケーリングが変化する点は $N_{\mathrm{fill}} = 108 \times 1024 = 110592$ である．

粒子数 N が N_{fill} を越えた直後においては，スレッドブロックの演算処理を 1 度だけ処理する CU/SM と 2 度処理する CU/SM が混在し，実行時間が不連続に増えることとなる．MI100 においては $N = 32\,768$，A100 においては $N = 131\,072$ における測定結果が該当する．この実行時間の段階的増加は粒子数 N が N_{fill} の整数倍を超過する度に起こるが，粒子数 N が増加するほどパイプラインの段数が増え，実行性能はより滑らかに飽和性能へと近づいていく．この振る舞いの詳細については，6.1 節で議論する．

図 3 からは A100 上において CUDA 版と HIP 版の性能差は読み取れないが，定量的に比較するために CUDA 版での実行時間 t_{CUDA} と HIP 版での実行時間 t_{HIP} の比を図

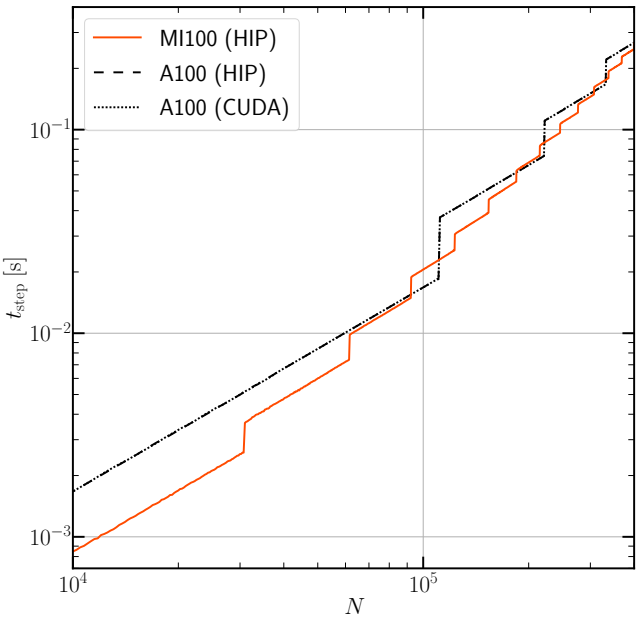

図 5 粒子数が中程度の領域における実行時間．GPU 内の CU/SM を埋め切るための粒子数 N_{fill} は，本研究の構成の下では MI100 において 30 720，A100 において 110 592 である．粒子数 N が N_{fill} 未満から N_{fill} の数倍程度となる領域において，重力計算関数の実行時間 t_{step} を測定しプロットした．

4 に示した．それぞれの実行時間としては，100 回測定した結果の中央値を用いた．この図から $t_{\mathrm{CUDA}}/t_{\mathrm{HIP}}$ がほぼ 1 となっており，両者に性能差がないことが分かる．したがって，HIP を用いることによるオーバーヘッドはない，あるいはオーバーヘッドがあったとしても十分に小さいために今回の測定では検出できなかったと結論される．

6. 議論

6.1 中間領域での振る舞い

前節で報告した性能測定の結果，$N < N_{\mathrm{fill}}$ では実行性能が粒子数 N に比例して増加，$N \gg N_{\mathrm{fill}}$ において飽和性能が得られることが分かり，またこの結果は MI100 と A100 で共通であった．一方で，中間領域である $N \sim N_{\mathrm{fill}}$ における振る舞いには MI100 と A100 で違いが見られた．A100 においては，$N = 131\,072$ において一旦実行性能の増加が止まるという鞍点状の構造が見られ，さらに粒子数を増やすと実行性能も増加していった．これに対して MI100 では，$N = 32\,768$ において実行性能の伸びが若干ゆるやかになるものの，A100 に比べてなめらかに実行性能が増加していく．

こうした振る舞いをより詳細に調べるため，粒子数が N_{fill} 付近となる領域に注目し，粒子数をスレッド数刻みで増やす追加測定を行った．各粒子数における測定を 100 回行い，ステップあたりの実行時間 t_{step} の最小値を図 5 に示す．

実行時間 t_{step} の粒子数依存性には，N_i, N_j それぞれに対する依存性がある．図 5 を見ると，実行時間 t_{step} が概ね N に比例して増加するトレンドの上に，$\lceil N/N_{\mathrm{fill}} \rceil$ の値が変化したタイミングで実行時間が不連続に変化するという振る舞いが乗っていることが読み取れる．前者の比例関係は N_j の増加によるループ長の延長，後者の振る舞いは N_i の増加によるスレッドブロック数増大が直接的な要因である．不連続に計算時間が増大する頻度が異なるのは，N_{fill} の値が MI100 では 30 720，A100 では 110 592 と約 4 倍異なるためである．

表 4 A100 上での性能測定結果（$N_{\mathrm{fill}} = 110\,592$）．

N	t_{step} [s]	実行時間比	$\lceil N/N_{\mathrm{fill}} \rceil$
110 592	1.86×10^{-2}	2.00	1
111 616	3.71×10^{-2}		2
221 184	7.43×10^{-2}	1.49	2
222 208	1.11×10^{-1}		3
331 776	1.67×10^{-1}	1.32	3
332 800	2.21×10^{-1}		4
442 368	2.97×10^{-1}	1.24	4
443 392	3.68×10^{-1}		5

表 5 MI100 上での性能測定結果（$N_{\mathrm{fill}} = 30\,720$）．

N	t_{step} [s]	実行時間比	$\lceil N/N_{\mathrm{fill}} \rceil$
30 720	2.59×10^{-3}	1.40	1
30 976	3.63×10^{-3}		2
61 440	7.41×10^{-3}	1.33	2
61 696	9.82×10^{-3}		3
92 160	1.49×10^{-2}	1.27	3
92 416	1.89×10^{-2}		4
122 880	1.49×10^{-2}	1.20	4
123 136	1.89×10^{-2}		5

図 5 では実行時間の増大率が MI100 よりも A100 の方が大きくなっている．定量的に比較するため，$\lceil N/N_{\mathrm{fill}} \rceil$ の値が変化する直前・直後での実行時間とその比を，A100（表 4）と MI100（表 5）それぞれについてまとめた．A100 については，$\lceil N/N_{\mathrm{fill}} \rceil$ の境界をまたいだ点の実行時間の増え方が 2/1, 3/2, 4/3, 5/4 倍となっており，$\lceil N/N_{\mathrm{fill}} \rceil$ の段数比に対応している．MI100 についても同様の評価をすると，実行時間の増方が $\lceil N/N_{\mathrm{fill}} \rceil$ の段数由来のものに比べて小さいことが分かる．これは CU を共有する複数のブロックにおける演算処理・メモリアクセスが同時進行することで実行時間の一部が隠蔽されているためだと考えられる．実行時間の隠蔽度合いが異なる理由としては MI100 と A100 のハードウェア側の違い以外にも，スレッドブロックあたりのスレッド数や，シェアードメモリの使用・不使用に由来する `__syncthreads()` 命令の発行・未発行といった実装上の違いも考えられる．

6.2 浮動小数点演算数に関する仮定の妥当性

本研究では，相互作用あたりの浮動小数点演算数として 22 Flops を仮定した．N 体計算の性能評価において

は，伝統的に 38 Flops という値が仮定されることが多い [8], [9], [10], [12], [18], [24] ので，ここで 22 Flops という仮定の妥当性を議論しておく．

相互作用あたりの演算量は減算 3 回，乗算 3 回，FMA 命令 6 回，逆数平方根 1 回である．減算および乗算は 1 Flop の演算を 1 サイクルで実行，FMA 命令については 2 Flops の演算を 1 サイクルで実行する．A100 上で実行した逆数平方根演算 `rsqrtf()` についてはスループットの値が示されている [20] ため，FMA 命令との比から 4 サイクル実行に当たることが分かる．ここで，逆数平方根演算の浮動小数点演算数として，サイクル数の比とそろえて 4 Flops であると仮定する．これより相互作用あたりの演算量は 22 Flops と見積もられることになる．他には逆数平方根の浮動小数点演算数を 2 Flops と仮定する流儀 [5], [6] もあり，これはサイクル数の比から浮動小数点演算数を見積もる際に FMA 命令の演算量 2 Flops を基準としている．この場合には相互作用あたりの演算量は 20 Flops と見積もられることになる．

MI100 向けに同じ精度の見積もりをするための根拠となる資料は見つけられなかった．そこで，MI100 上で `clock64()` 関数を用いて各命令の実行に要するサイクル数を測定した．測定結果は `1.0f / sqrtf(val)` では 1.47×10^2，`rsqrtf(val)` では 39.3，`__frsqrt_rn(val)` では 16.5 となり，`__frsqrt_rn(val)` の値が最小である．同様にして FMA 命令についても `clock64()` 関数を用いて測定すると 5.00 となったので，この比をサイクル数の見積もりとする．つまり，`1.0f / sqrtf(val)` は 29, `rsqrtf(val)` では 8，`__frsqrt_rn(val)` では 3 サイクルかかると考え，この値を Flops 値としても使用する．MI100 で最終的に用いた命令は `__frsqrt_rn(val)` であるため，3 サイクルで 3 Flops という値となり A100 の場合とは一致しない．このため，MI100 上では相互作用あたりの浮動小数点演算数を 21 Flops と仮定する方がもっともらしい．しかし本研究では性能値同士を直接比較するため，逆数平方根の浮動小数点演算数として 4 Flops という共通の値を A100 と MI100 両方に仮定することとした．

6.3 到達可能な性能値の見積もり

本節ではメモリアクセスのレイテンシや演算命令実行のストールといった要素を無視し，浮動小数点演算に要する実行時間だけから N 体計算の実行性能の上限値を見積もり，本研究で得られた測定結果と比較する．

A100 においては 1 相互作用を計算する浮動小数点演算に 16 サイクルを要する．GPU の理論ピーク演算性能は全ての演算が FMA 命令であった場合として計算されているので，16 サイクルであれば 32 Flops が上限値となる．したがって，N 体計算については理論ピーク演算性能の $22/32 \sim 69\%$ が到達可能な上限値となる．A100 の単精度理論ピーク演算性能は 19.5 TFlop/s であり，本研究で得られた 14.51 TFlop/s は理論ピークの 74% に当たる．A100 上では逆数平方根演算 `rsqrtf()` が SFU という CUDA コアとは異なる演算器で計算され，実行時間の一部が隠蔽されるために理論値である 69% よりも高い数値が得られる．逆数平方根の計算時間が完全に隠ぺいされたと仮定すると，12 サイクルで 22 Flops を処理することになる．この時，N 体計算は理論ピーク演算性能の $22/(12 \times 2) \sim 92\%$ まで達成可能ということになり，本研究で得られた理論ピークの 74% という結果はここでの見積もりと矛盾しない．

MI100 においては，相互作用あたりの浮動小数点演算には 15 サイクルを要する（6.2 節）．これに対応する上限値は 30 Flops であり，本研究では相互作用あたり 22 Flops を仮定しているため，理論ピーク演算性能の $22/30 \sim 73\%$ が到達可能な上限値となる．本研究で得られた 14.63 TFlop/s という結果は MI100 の単精度理論ピーク演算性能 23.1 TFlop/s の 63% にあたる．これは到達可能な上限値の 86% にあたり，よく最適化されたコードであると言える．

また，MI100 上では 3 種類の逆数平方根演算の実行サイクル数も測定済みである．これより，4.1 節における簡易測定結果についても議論できる．1 組分の相互作用を計算するのに要するサイクル数は，逆数平方根として `1.0f / sqrtf(val)` を用いると 41, `rsqrtf(val)` では 20, `__frsqrt_rn(val)` では 15 サイクルと見積もられる．したがって，`1.0f / sqrtf(val)` での実行性能を基準とすれば，`rsqrtf(val)` 使用時には 2.1 倍，`__frsqrt_rn(val)` 使用時には 2.7 倍の性能が得られると期待される．表 3 から同様の比を計算すると，`rsqrtf(val)` 使用時には `1.0f / sqrtf(val)` 使用時の 2.1 倍，`__frsqrt_rn(val)` 使用時には `1.0f / sqrtf(val)` 使用時の 2.7 倍となっており，本節における見積もりとも整合する．

7. まとめ

今まで導入・運用されてきた GPU スパコンはそのほとんどが NVIDIA 製 GPU を搭載してきたが，AMD 製 GPU や Intel 製 GPU を搭載する GPU スパコンの導入が多数発表されている．したがって今後導入される GPU スパコンを利活用するには，AMD 製 GPU や Intel 製 GPU を対象としたコード開発・性能最適化も重要になってきた．そこで本研究では，AMD 社が提供する開発環境である ROCm/HIP を用いて，NVIDIA 製 GPU 上でも AMD 製 GPU 上でも動作可能な N 体計算コードを実装し，性能最適化を施した上で性能評価を行った．

AMD MI100 向けの性能最適化としては，逆数平方根の計算に `__frsqrt_rn(val)` を用いることが重要であった．相互作用あたりの浮動小数点演算数として単精度 22 Flops を仮定すると，AMD MI100 上では最大 14.6 TFlop/s, NVIDIA A100 上では最大 14.5 TFlop/s となり，両 GPU はほぼ同等

の高い演算性能を発揮できることが確かめられた．NVIDIA A100 上では CUDA 版コードと HIP 版コードの両方が動作するためその性能差を測定したところ，性能差は存在しない，つまり HIP 実装によるオーバーヘッドは存在しないか十分に小さいということが分かった．また詳細な性能分析も行い，両 GPU 上での性能測定結果の違いの多くが CU/SM 数（MI100 は 120 CUs 搭載，A100 は 108 SMs 搭載）とスレッドブロックあたりのスレッド数の違いで説明できることが分かった．

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

参考文献

[1] Advanced Micro Devices, Inc.: *AMD CDNA ARCHITECTURE* (2020).
[2] Advanced Micro Devices, Inc.: *AMD ROCm HIP Programming Guide* (2021).
[3] Advanced Micro Devices, Inc.: ROCm - Open Source Platform for HPC and Ultrascale GPU Computing, (online), available from ⟨https://github.com/RadeonOpenCompute/ROCm⟩ (2021).
[4] Argonne Leadership Computing Facility: Aurora, (online), available from ⟨https://www.alcf.anl.gov/aurora/⟩ (2021).
[5] Bédorf, J., Gaburov, E., Fujii, M. S., Nitadori, K., Ishiyama, T. and Zwart, S. P.: 24.77 Pflops on a Gravitational Tree-Code to Simulate the Milky Way Galaxy with 18600 GPUs, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014* (Damkroger, T. and Dongarra, J. J., eds.), IEEE Computer Society, pp. 54–65 (online), DOI: 10.1109/SC.2014.10 (2014).
[6] Capuzzo-Dolcetta, R. and Spera, M.: A performance comparison of different graphics processing units running direct N-body simulations, *Computer Physics Communications*, Vol. 184, pp. 2528–2539 (online), DOI: 10.1016/j.cpc.2013.07.005 (2013).
[7] Centro Svizzero di Calcolo Scientifico: Alps, (online), available from ⟨https://www.cscs.ch/computers/alps/⟩ (2021).
[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] Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K. and Taiji, M.: 42 TFlops hierarchical *N*-body simulations on GPUs with applications in both astrophysics and turbulence, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, ACM, pp. 62:1–62:12 (online), DOI: http://doi.acm.org/10.1145/1654059.1654123 (2009).
[10] Hamada, T. and Nitadori, K.: 190 TFlops Astrophysical *N*-body Simulation on a Cluster of GPUs, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, IEEE Computer Society, pp. 1–9 (online), DOI: http://dx.doi.org/10.1109/SC.2010.1 (2010).
[11] Intel Corporation: oneAPI Programming Model, (online), available from ⟨https://www.oneapi.io/⟩ (2019).
[12] Kawai, A., Fukushige, T. and Makino, J.: $7.0/Mflops astrophysical N-body simulation with treecode on GRAPE-5, *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, New York, NY, USA, ACM, (online), DOI: http://doi.acm.org/10.1145/331532.331598 (1999).
[13] Lawrence Livermore National Laboratory: LLNL and HPE to partner with AMD on El Capitan, projected as world's fastest supercomputer,（オンライン），入手先 ⟨https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer⟩ (2020).
[14] 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: 10.1016/j.procs.2012.04.011 (2012).
[15] 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).
[16] Miki, Y.: Gravitational Octree Code Performance Evaluation on Volta GPU, *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, ACM, pp. 62:1–62:10 (online), DOI: 10.1145/3337821.3337845 (2019).
[17] National Energy Research Scientific Computing Center: Perlmutter, (online), available from ⟨https://www.nersc.gov/systems/perlmutter/⟩ (2021).
[18] Nitadori, K. and Makino, J.: Sixth- and eighth-order Hermite integrator for N-body simulations, *New Astronomy*, Vol. 13, pp. 498–507 (online), DOI: 10.1016/j.newast.2008.01.010 (2008).
[19] NVIDIA Cooporation: *NVIDIA A100 Tensor Core GPU Architecture* (2020).
[20] NVIDIA Corporation: *CUDA C++ Programming Guide Version 11.4* (2021).
[21] Nyland, L., Harris, M. and Prins, J.: Fast *N*-Body Simulation with CUDA (2007).
[22] Oak Ridge National Laboratory: Frontier, (online), available from ⟨https://www.olcf.ornl.gov/frontier/⟩ (2021).
[23] Preferred Networks: PFN's Supercomputers, (online), available from ⟨https://projects.preferred.jp/supercomputers/#mn-3a⟩ (2021).
[24] Tanikawa, A., Yoshikawa, K., Nitadori, K. and Okamoto, T.: Phantom-GRAPE: Numerical software library to accelerate collisionless N-body simulation with SIMD instruction set on x86 architecture, *New Astronomy*, Vol. 19, pp. 74–88 (online), DOI: 10.1016/j.newast.2012.08.009 (2013).
[25] TOP500.org: TOP500 Lists (June 2021), (online), available from ⟨https://www.top500.org/lists/top500/2021/06/⟩ (2021).
[26] 中島研吾, 塙 敏博, 下川辺隆史, 伊田明弘, 芝 隼人, 三木洋平, 星野哲也, 有間英志, 河合直聡, 坂本龍一, 近藤正章, 岩下武史, 八代 尚, 長尾大道, 松葉浩也, 荻田武史, 片桐孝洋, 古村孝志, 鶴岡 弘, 市村 強, 藤田航平: 「計算・データ・学習」融合スーパーコンピュータシステム「Wisteris/BDEC-01」の概要, 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-179, No. 1, pp. 1–7 (2021).
[27] 産業技術総合研究所：ABCI システムの概要，

（オンライン），入手先 ⟨https://docs.abci.ai/ja/system-overview/⟩ (2021).
[28] 東京大学情報基盤センター：Wisteria/BDEC-01 スーパーコンピュータシステムの紹介，（オンライン），入手先 ⟨https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/system.php⟩ (2021).
[29] 塙　敏博，中島研吾，下川辺隆史，芝　隼人，三木洋平，星野哲也，河合直聡，似鳥啓吾，今村俊幸，工藤周平，中尾昌広：「計算・データ・学習」融合スーパーコンピュータシステム Wisteris/BDEC-01 の性能評価，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-180, No. 22, pp. 1–8 (2021).
[30] 三木洋平：NVIDIA A100 における重力ツリーコードの性能評価，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-180, No. 24, pp. 1–7 (2021).

付　　録

A.1　性能評価結果の最高値と中央値

5 節において実行した性能評価において，各粒子数ごと 100 回の測定において記録した最高性能 R_{max} および中央値 R_{med} を表 A·1 に示した．

表 A·1　性能測定結果の最高値と中央値．

N	MI100		A100 (HIP)		A100 (CUDA)	
	R_{max} [TFlop/s]	R_{med} [TFlop/s]	R_{max} [TFlop/s]	R_{med} [TFlop/s]	R_{max} [TFlop/s]	R_{med} [TFlop/s]
1024	0.2685	0.2647	0.1339	0.1338	0.1339	0.1339
2048	0.5427	0.5356	0.2694	0.2692	0.2694	0.2693
4096	1.083	1.075	0.5406	0.5405	0.5406	0.5405
8192	2.166	2.155	1.083	1.083	1.083	1.083
16 384	4.324	4.307	2.168	2.168	2.168	2.168
32 768	6.097	6.075	4.338	4.312	4.338	4.316
65 536	9.053	9.022	8.666	8.593	8.675	8.591
131 072	11.35	11.25	8.679	8.679	8.679	8.679
262 144	13.36	13.26	11.52	11.50	11.52	11.50
524 288	13.73	13.62	13.79	13.78	13.79	13.78
1 048 576	14.20	14.11	13.81	13.80	13.81	13.79
2 097 152	14.43	14.38	14.49	14.47	14.48	14.47
4 194 304	14.63	14.58	14.48	14.47	14.48	14.47
8 388 608	14.61	14.48	14.49	14.46	14.46	14.45
16 777 216	14.54	14.50	14.51	14.50	14.51	14.50