Vol.2018-HPC-166 No.6
2018/9/27

Volta 世代の GPU における重力ツリーコードの性能評価

三木 洋平[1,a)]

概要：Fermi 世代から Pascal 世代までの GPU 向けの最適化がなされている重力ツリーコード `GOTHIC` を，Volta 世代の GPU である Tesla V100 向けに移植し，その性能を評価した．Tesla V100 を用いて性能を測定したところ，$N = 2^{23} = 8\,388\,608$ 粒子で表現したアンドロメダ銀河モデルの計算に要した時間はステップあたり 3.3×10^{-2} s であり，コンパイル時に`-gencode arch=compute_60,code=sm_70` を指定することで約 1.2 倍の性能向上が得られることが分かった．また Pascal 世代の GPU である Tesla P100 と比較すると，1.4–2.2 倍の高速化が達成されることが分かった．得られた 2.2 倍という速度向上率は単精度理論ピーク演算性能比である 1.5 よりも大きい．これは，Volta 世代の GPU において整数演算ユニットが単精度浮動小数点演算ユニットから独立したことによる性能向上だと考えられる．整数演算ユニットの独立によって整数演算と単精度浮動小数点演算の同時実行が可能となり，整数演算の実行時間が単精度浮動小数点演算の実行時間によって隠蔽されることで，理論ピーク演算性能比を越える速度向上率が実現され得る．Tesla V100 上では最大 $N = 25 \times 2^{20} = 26\,214\,400$ 粒子の計算が実行でき，ステップあたりの実行時間は 2.0×10^{-1} s であった．得られた単精度演算性能は 3.5 TFlop/s であり，Tesla V100 の単精度理論ピーク演算性能の 22%にあたる．

1. はじめに

宇宙物理学の研究においては，銀河などの重力多体系の形成・進化過程を詳細に調べるために N 体シミュレーションが用いられている．N 体シミュレーションでは，系を構成する粒子間にはたらく重力を計算し，各粒子の軌道進化を追うことで系の時間発展を求める．軌道積分に必要な加速度は Newton の運動方程式

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

によって与えられる．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 ϵ は，N 体シミュレーションで用いる粒子数 N が現実の系の粒子数よりも少ないために生じる人工的な効果を低減するために導入されており，ゼロ割による発散や自己相互作用を取り除くという役割も果たす．

粒子間の重力相互作用を計算する際に (1) 式をそのまま用いる直接法では計算量が $\mathcal{O}(N^2)$ となり，現実的な実行時間で計算を終えることは難しい．ツリー法 [1] は遠方粒子からの重力を計算する際に多重極展開を用いて計算量を減らす手法であり，演算量は $\mathcal{O}(N \log N)$ となる．アルゴリズム的な加速以外の N 体シミュレーションの高速化手法としては演算加速器の使用があげられ，GPU を用いたツリー法の高速化は多くの成功を収めてきた [2], [4], [12], [17].

著者が開発している `GOTHIC` はツリー法と階層化時間刻み法 [10] を採用した重力ツリーコードであり，CUDA C/C++ で実装されている [12]．`GOTHIC` の実装においては，ツリー構造の構築コストと重力計算の計算コストを監視することで，両者の和が最小化されるような最適なツリー構造の再構築頻度を設定する自動最適化も施されている．また，遠方粒子群からの重力を計算する際に多重極展開を許容するかを決定する判定基準には，重力計算の精度を制御するパラメータ $\varDelta_{\mathrm{acc}}$ に対して

$$\frac{G m_J}{d_{iJ}^2} \left(\frac{b_J}{d_{iJ}}\right)^2 \leq \varDelta_{\mathrm{acc}} \left|\boldsymbol{a}_i^{\mathrm{old}}\right| \quad (2)$$

を満たした時に重力計算を行うという基準 [19], [20] を採用している．ここで，m_J, b_J は遠方粒子群の質量およびサイズ，d_{iJ} は重力を受ける粒子と遠方粒子群の重心との距離であり，$\boldsymbol{a}_i^{\mathrm{old}}$ は 1 ステップ前の時刻における粒子の加速度である．この判定基準を用いると，一般的な判定方法に比べて短い計算時間で同じ精度の計算が実行できることが報告されており [12], [15]，また粒子分布が一様に近い場合においても重力計算の精度を担保しやすいという特長がある．

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

© 2018 Information Processing Society of Japan

NVIDIA 社の HPC 向け GPU の最新世代は Volta 世代であり，Tesla V100 の単精度理論ピーク演算性能は 15.7 TFlop/s と Pascal 世代の Tesla P100 よりも約 1.5 倍高性能である [16]．また詳細なマイクロベンチマークも実施されており，例えば L1 キャッシュのロード・スループットの実測値が Tesla P100 に比べて 3 倍速いことが報告されている [9]．加えて，Tesla V100 は産業技術総合研究所の ABCI や米国オークリッジ国立研究所の Summit といった近年運用を開始した大規模システムの演算加速器として採用されている．したがって，今まで Pascal 世代までの GPU 向けに開発されてきたアプリケーションを Volta 世代向けに移植し，性能最適化を施すことが急務である．特に，2.1 節で取り上げるように Volta 世代ではプログラミングモデルにおいても変更点があるため，実アプリケーションを Volta 世代向けに移植する際の注意事項や性能の振るまいなどを詳細に調べておくことが望ましい．

本研究では，Fermi 世代から Pascal 世代までの各世代の GPU 向けの最適化が施されている重力ツリーコード `GOTHIC` を Volta 世代向けに移植し，その性能評価を行う．2 節において Volta 世代の GPU へのコード移植について紹介し，3 節において性能評価の結果を報告し，4 節では Pascal 世代からの速度向上率や実装方法の違いによる演算性能の変化について議論する．

2. Volta 世代の GPU へのコード移植

本節では，Pascal 世代以前の GPU 向けの実装と Volta 世代の GPU 向け実装の違いと対応方法（2.1 節）と，マイクロベンチマークに基づいた Volta 世代向けの調整（2.2 節）について紹介する．

2.1 Volta 世代における変更点

プログラミングモデルにおける Pascal 世代以前との最大の違いは，ワープ内の暗黙の同期がなくなったことである．Pascal 世代以前においてはワープを構成する 32 スレッドの演算は同時に実行されていたため，同期が必要な処理をワープの中に閉じ込めてあげた際には，明示的に同期命令を発行する必要はなかった．これに対して Volta 世代においては，ワープ内の 32 スレッドの演算が同時に実行されるという保証はなくなった．このため，ワープ内の 32 スレッドを同期する`__syncwarp()` や，ワープ内の 2 ベキ数のスレッドを同期する Cooperative Groups のタイル同期といった新機能が提供されており，これらを用いて明示的に同期処理を行う必要がある．

また，ワープ内の暗黙の同期がなくなった影響として，ワープ分裂が起こった後に明示的な同期処理をいれるまではワープが分裂したまま動作し続けるということがあげられる（[16] の Figs. 20, 22, 23）．Pascal 世代以前においてはワープ分裂が起こるのは条件分岐に関する節の中だけであり，その後はワープ内の 32 スレッドが同時に動作していた．これに対して Volta 世代においては，条件分岐が発生する処理が全て終了しワープ内の 32 スレッドが同時に同じ演算を実行できるようになったとしても，明示的な同期命令を発行するまではワープが分裂したまま動作することがホワイトペーパー [16] の記述から読み取れる．したがって，適切に同期命令を発行しない限りはワープ分裂の持続時間が延びてしまうという性能面への悪影響があるため，Volta 世代への移植は慎重に行う必要がある．こうしたコードの書き換えが困難なユーザのために，`-gencode arch=compute_60,code=sm_70` を指定してコンパイルすることで，Pascal 世代以前と同じくワープ内の 32 スレッドを暗黙のうちに同期された状態で動作させるという救済措置も提供されている．以下，本稿では`-gencode arch=compute_60,code=sm_70` を指定してコンパイルした場合を Pascal モード，通常どおり`-gencode arch=compute_70,code=sm_70` を指定してコンパイルした場合を Volta モードと呼ぶこととする．

ワープ内の暗黙の同期がなくなったことに起因して，ワープシャッフル命令にも変更がなされた．具体的には，今まで提供されてきた`__shfl()` などが非推奨となり，新たに提供された`__shfl_sync()` などを使用することが推奨されている．この新命令では，ワープ内の 32 スレッドのうちワープシャッフル命令に関与するスレッドを指定するために，追加引数であるマスクを渡す必要がある．例えば 32 スレッドあるいは連続する 16 スレッドが同時に`__shfl_sync()` を呼ぶ際には，それぞれ `0xffffffff`, `0xffff` をマスクとして渡せば良い．注意が必要となるのはプログラムの挙動にデータ依存性がある場合であり，例えばレーンサイズが 16 である（16 スレッドの中でシャッフル命令を使う）場合には，1 グループだけがシャッフル命令を呼ぶ場合と同一ワープ内の 2 グループが同時にシャッフル命令を呼ぶ場合とがある．マスクとしては前者の場合には `0xffff`，後者の場合には `0xffffffff` を渡してあげないと正しい挙動とならない．こうした場合には，`__activemask()` を用いて同じ動作をしているスレッドに対応したマスクを動的に取得すれば良い．`GOTHIC` の移植においては，Pascal モードあるいは Volta モードでレーンサイズが 32 である場合においては `0xffffffff` をマスクとして指定し，Volta モードでレーンサイズが 32 未満である場合にはワープシャッフル命令を発行する直前に`__activemask()` を用いて適切なマスクを取得するという実装になっている．

Volta 世代の GPU においてはシェアードメモリと L1 キャッシュの容量の切り替え方法が変更されており，`cudaFuncSetAttribute()` の引数に対象となる関数名と `cudaFuncAttributePreferredSharedMemoryCarveout` およびシェアードメモリへ割当可能な総容量に対する割当比率を整数値で記載すると，CUDA が SM（Streaming

表 1　計算機環境

CPU	Intel Xeon E5-2680 v4 14 cores, 2.4 GHz	IBM POWER9 16 cores, 2.0–3.1 GHz
GPU	NVIDIA Tesla P100 3584 cores, 1.480 GHz HBM2 16 GB	NVIDIA Tesla V100 5120 cores, 1.530 GHz HBM2 16 GB
コンパイラ	icc 18.0.1.163 CUDA 8.0.61	gcc 4.8.5 CUDA 9.2.88

表 2　関数ごとのスレッド数およびレジスタ使用本数

関数	Ttot	Tsub	Volta モード	Pascal モード
walkTree	512	32	64	63
calcNode	128	32	66	56
makeTree	512	8	58	50
predict	512	–	27	27
correct	512	32	25	25

Multiprocessor）あたりのシェアードメモリの容量を 0, 8, 16, 32, 64, 96 KiB のいずれかに設定する．例えば，値を 66 と指定するとシェアードメモリには 96 KiB の 2/3 である 64 KiB が割り当てられる（67 とした際には 96 KiB が割り当てられたため，小数点以下は切り捨てる）．Volta 版の GOTHIC においては，重力計算カーネルのようにシェアードメモリの容量を増やしたい関数では割当比率を 100 に，シェアードメモリを使用しない関数では 0 にするなど，全関数について割当比率をコード内に明示した．

また，CUDA 9 で提供された新機能として Cooperative Groups を用いた複数ブロックをまたいでの全体同期があげられるが，A.1 節に示すように GOTHIC においては性能が低下したため，この機能は使用しないこととした．

2.2　パラメータ調整

ツリーコードの性能は用いる粒子分布によっても変化するため，本研究では現実的な粒子分布としてアンドロメダ銀河を模したモデルを採用する．このモデルは，Fardal らによって構築された銀河モデル [3], [5] に最近の観測に基く恒星ハロー成分 [6], [8] を追加したものであり，ダークマターハロー（Navarro–Frenk–White モデル [14]，$M = 8.11 \times 10^{11}\ M_\odot$, $r_s = 7.63$ kpc），恒星ハロー（Sérsic モデル [18]，$M = 8 \times 10^{11}\ M_\odot$, $r_s = 9$ kpc, $n = 2.2$），バルジ（Hernquist モデル [7]，$M = 3.24 \times 10^{10}\ M_\odot$, $r_s = 0.61$ kpc），銀河円盤（等温・指数関数型 [13]，$M = 3.66 \times 10^{10}\ M_\odot$, $R_s = 5.4$ kpc, $z_s = 0.6$ kpc, $Q_{\min} = 1.8$ [21]）からなる．すべての粒子の質量を等質量とし，MAGI [13] を用いて力学平衡状態にある粒子分布を生成した．

GOTHIC の演算速度は，ブロックあたりのスレッド数 Ttot やコード内でスキャンや縮約演算などを実行する際の最小単位となるスレッド数 Tsub といったパラメータの値によって大きく変化する．本研究では，$N = 2^{23} = 8\,388\,608$ のアンドロメダ銀河モデルを $\Delta_{\rm acc} = 2^{-9} = 1.953125 \times 10^{-3}$ として 4096 ステップだけ計算し，Volta モードにおける実行時間が最小となるパラメータの組み合わせを採用した．性能評価に用いた計算機環境は**表 1** に示した通りであり，また主要な関数におけるスレッド数を**表 2** にまとめた．表中の walkTree は重力計算関数，calcNode はツリーノードの重心位置や質量の計算関数，makeTree はツリー構造の構築関数，predict, correct は予測子・修正子法による時間積分を実行する関数である．また重力計算関数の SM あたりのブロック数については，2, 3, 4 の 3 通りで実験した結果，2 とした場合が最も速かったため，Pascal 世代以前と同様に 2 とした．

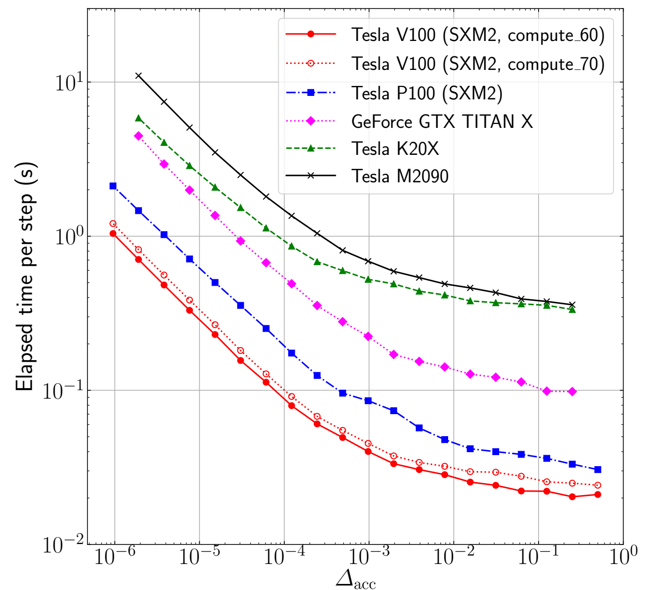

図 1　GOTHIC の実行時間．重力計算の精度を制御するパラメータ $\Delta_{\rm acc}$ の関数として，ステップあたりの実行時間を示した．Maxwell 世代以前の GPU 上での測定結果については [12] において報告済みのデータである．

3.　性能測定の結果

GOTHIC の実行時間を，重力計算の精度を制御するパラメータ $\Delta_{\rm acc}$ の関数として測定した（**図 1**）．測定条件は 2.2 節で行ったマイクロベンチマークと同じであり，性能評価には Tesla P100 および Tesla V100 を用いた（表 1）．また，Maxwell 世代以前の GPU 上での測定結果については [12] において報告済みのものである．典型的な精度である $\Delta_{\rm acc} = 2^{-9} = 1.953125 \times 10^{-3}$ のときのステップあたりの実行時間は，Tesla P100 上で 7.4×10^{-2} s，Tesla V100 上では Volta モードで 3.8×10^{-2} s，Pascal モードで 3.3×10^{-2} s であった．Kepler 世代を除けば $\Delta_{\rm acc}$ に対する計算時間の依存性は GPU の世代に依らずほぼ同じであり，Volta 世代の GPU では Fermi 世代の GPU と同一のアルゴリズムのままで 10 倍以上高速に計算できることが確かめられた．また，Tesla V100 において Volta モードと

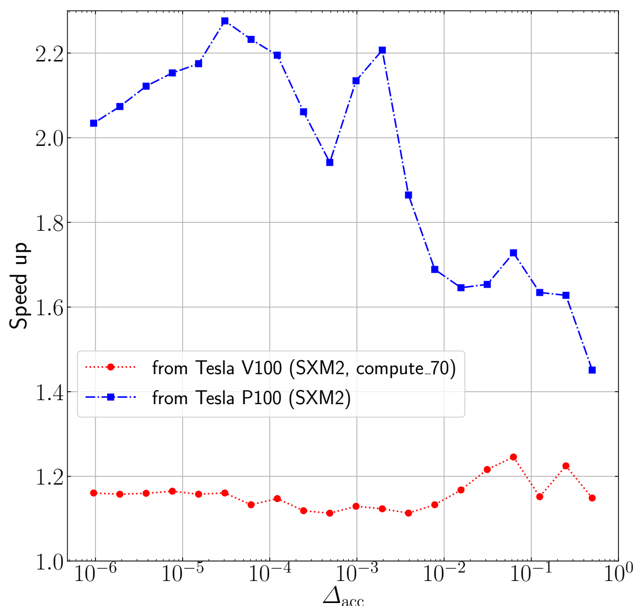

図 2 GOTHIC の速度向上率．Tesla V100 を用いて Pascal モードで動作させた際の計算時間を，Volta モード（赤丸）および Tesla P100（青の四角形）と比較した．

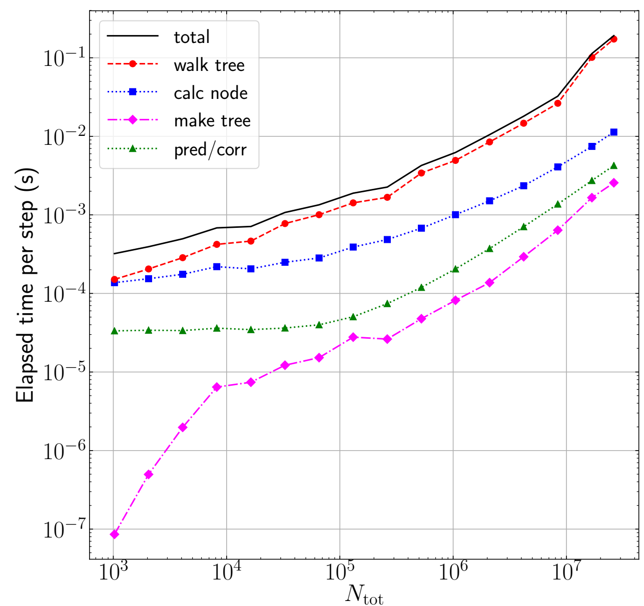

図 3 GOTHIC の実行時間の粒子数依存性．全粒子数 $N_{\rm tot}$ の関数として，$\Delta_{\rm acc} = 2^{-9} = 1.953125 \times 10^{-3}$ における実行時間を示した．

Pascal モードの結果を比較すると，常に Pascal モードの方が高速であることが分かった．以降では，Pascal モードを Tesla V100 使用時の標準設定とする．

Pascal モードにおける測定結果を，Tesla P100 使用時や Volta モードでの測定結果に対する速度向上率として**図 2** に示した．Tesla P100 からは最大で 2.2 倍程度，最低でも 1.4 倍程度の高速化が達成されている（青の四角形）．特に $\Delta_{\rm acc} \lesssim 10^{-3}$ の領域においてはおおむね 2 倍以上の高速化が達成されている．この値は Tesla P100 と Tesla V100 の理論ピーク演算性能比である 1.5 よりも大きな値であり，4.2 節において議論するように整数演算の実行時間が隠蔽されたために実現されたと考えられる．Tesla V100 上での動作モードの比較では，Pascal モードの方が $\Delta_{\rm acc}$ に依らず 1.1–1.2 倍高速であることが分かった（赤丸）．

GOTHIC の実行時間の粒子数依存性を調べるために，重力計算の精度を制御するパラメータの値を $\Delta_{\rm acc} = 2^{-9} = 1.953125 \times 10^{-3}$ で固定し，粒子数を $N = 2^{10} = 1024$ から $N = 25 \times 2^{20} = 26\,214\,400$ まで変化させた際の実行時間を測定した（**図 3**）．この際に，粒子数が少ない領域での測定精度を担保するために，本測定では 65 536 ステップ分の実行時間を測定した．最大の粒子数を用いた $N = 25 \times 2^{20} = 26\,214\,400$ におけるステップあたりの実行時間は 2.0×10^{-1} s であった．Pascal 世代の GPU と同様に，重力計算関数の処理時間（赤丸）が支配的であるが，粒子数が少ない領域においてはツリーノードに関する処理時間（青の四角形）も無視できない．実行時間のスケーリングは Pascal 世代までの GPU 上で得られていた $\propto N$ よりも少し悪くなっているが，ツリーコードの演算量は $\mathcal{O}(N \log N)$ であるため，自然な結果である．

また，Tesla P100 使用時には $N = 30 \times 2^{20} = 31\,457\,280$ 粒子の計算がステップあたり 3.3×10^{-1} s で実行できたが，Tesla V100 では上限の粒子数が減少している．これは，Tesla V100 の SM 数が Tesla P100 に対して 1.4 倍程度増えているため，幅優先ツリーの探索時に使用するバッファ領域の要求量が増加したにも関わらず，メモリ容量がどちらも 16 GB であることが原因である．したがって，32 GB のメモリを搭載したバージョンの Tesla V100 を使用すれば，Tesla P100 よりも多くの粒子数を用いた N 体シミュレーションが実行できると期待される．

4. 議論

図 2 に示した速度向上率について，4.1 節では Pascal モードと Volta モードの違いに，4.2 節では Tesla P100 からの速度向上率に注目して議論する．

4.1 Volta モードからの速度向上率の起源

Tesla V100 上で Pascal モードと Volta モードを比較した結果，Pascal モードの方が 1.2 倍程度高速であった原因を考察する．Pascal モードでは，Volta 世代において導入された各スレッドが独立に動作可能な設定を無効化し，Pascal 世代と同様にワープ内の 32 スレッドが同時に演算を実行することを保証する．このため，`__syncwarp()` 命令や Cooperative Groups を用いてのタイル同期をかける必要がなくなるというメリットがある．しかし，Volta 世代向けの機能を一部無効化するということであるため，コンパイラによる最適化レベルが低下することで性能が劣化

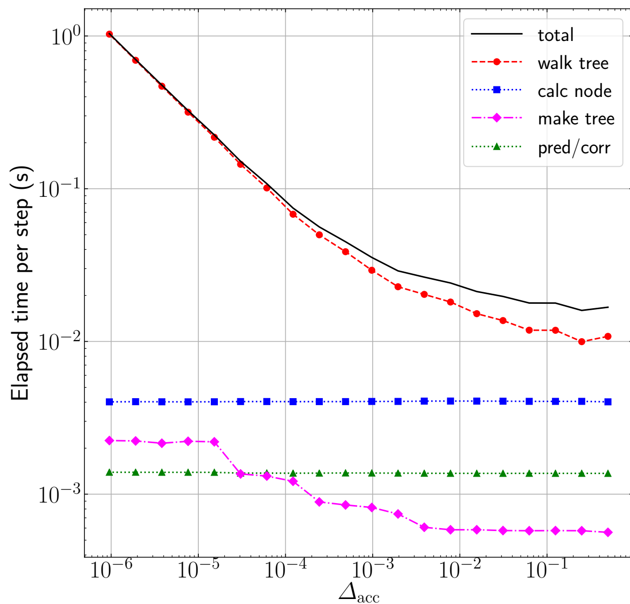

図 4 GOTHIC の実行時間の内訳．重力計算の精度を制御するパラメータ $\mathit{\Delta}_{\mathrm{acc}}$ の関数として，代表的な処理とその総和を示した．赤丸は重力計算，青の四角形はツリーノードの重心や質量の計算，マゼンタの菱形はツリー構造の構築，緑の三角形は予測子・修正子法による時間積分に要する時間を表す．

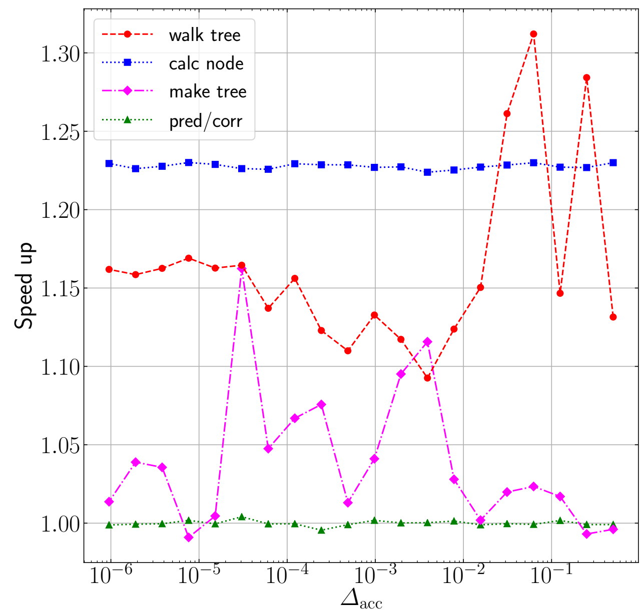

図 5 GOTHIC の関数ごとの速度向上率．Pascal モードの Volta モードからの速度向上率を，重力計算の精度を制御するパラメータ $\mathit{\Delta}_{\mathrm{acc}}$ の関数として示した．

するリスクもある．

GOTHIC 内の関数には，ワープ内の同期処理に対して演算処理が重い関数（重力計算），スキャンやリダクション系の演算が多いために同期処理の影響を受けやすい関数（ツリーノードに関する計算），8 スレッド単位で処理を実行するために`__activemask()` や Cooperative Groups のタイル同期を使用する関数（ツリー構築），ワープ内の同期処理が不要な関数（時間積分）という様々な特性をもった関数が含まれる．そこで，Pascal モードにおける関数ごとの実行時間を測定（**図 4**）し，Volta モードからの速度向上率を調べた（**図 5**）．

図 4 からは，重力計算に要する時間（赤丸）が $\mathit{\Delta}_{\mathrm{acc}}$ に依らず支配的であること，ツリーノードや時間積分の処理時間（それぞれ青の四角形と緑の三角形）は一定であること，ツリー構築に要する時間（マゼンタの菱形）は $\mathit{\Delta}_{\mathrm{acc}}$ を大きくしていくと短くなっていくということが読み取れる．これは，重力計算の精度が高い（$\mathit{\Delta}_{\mathrm{acc}}$ の値が小さい）ほど重力計算に時間がかかること，またツリーノードや時間積分の処理は重力計算の精度とは関係の無い処理であることが要因である．本来はツリー構築も重力計算の精度とは関係のない処理であるが，重力計算の処理時間とツリー構築の処理時間の和を最小化するようにツリー構造の再構築頻度を自動調整しているため，重力計算に要する時間の変化を通じてツリー構築の頻度が変化（具体的には，高精度の場合で 6 ステップに 1 回，低精度の場合で 30 ステップ程度に 1 回）する．重力計算に要する時間が支配的であるため，Volta モードとの比較の際には基本的には重力計算関数の比に注目すれば良いが，$\mathit{\Delta}_{\mathrm{acc}}$ の値が大きい場合にはツリーノードの処理時間の寄与も無視できない．

図 5 に示した Volta モードに対する Pascal モードの関数ごとの速度向上率からは，Volta モードの方が高速な関数はなく，関数によっては Pascal モードの方が 2-3 割高速であることが読み取れる．全体への寄与が最も大きい重力計算関数（赤丸）については，Pascal モードの方が 15%程度高速であり，$\mathit{\Delta}_{\mathrm{acc}} \gtrsim 10^{-2}$ の領域においては速度向上率が上昇する傾向が見える．また，$\mathit{\Delta}_{\mathrm{acc}} \gtrsim 10^{-2}$ においては 23%程度の速度向上を示しているツリーノードの処理時間（青の四角形）の寄与も無視できない．

主にこの 2 つの関数の振る舞いによって，GOTHIC 全体としては Volta モードから Pascal モードに変えることで 20%程度の高速化が達成されることとなる．こうした関数内には，ワープを構成する 32 スレッド内でのリダクション演算やスキャンが含まれるため，`__syncwarp()` が多数回呼ばれている．Pascal モードにおいてはワープ内の 32 スレッドが暗黙のうちに同期されているため，`__syncwarp()` を呼ぶ必要がない．したがって，今回観測された性能の違いは`__syncwarp()` の有無によると考えられ，同期処理の重たい関数であるほど Pascal モードで実行するメリットが大きいことが分かる．

時間積分に関する処理としては，毎ステップ全粒子のデータをアップデートする予測子の計算時間が支配的となるが，この処理には`__syncwarp()` は不要である．このため，Pascal モードと Volta モードどちらにおいても全く同じ処理を実行することとなり，違いが生じるとすればコンパイラによる最適化がどの程度なされるかという点だけで

© 2018 Information Processing Society of Japan

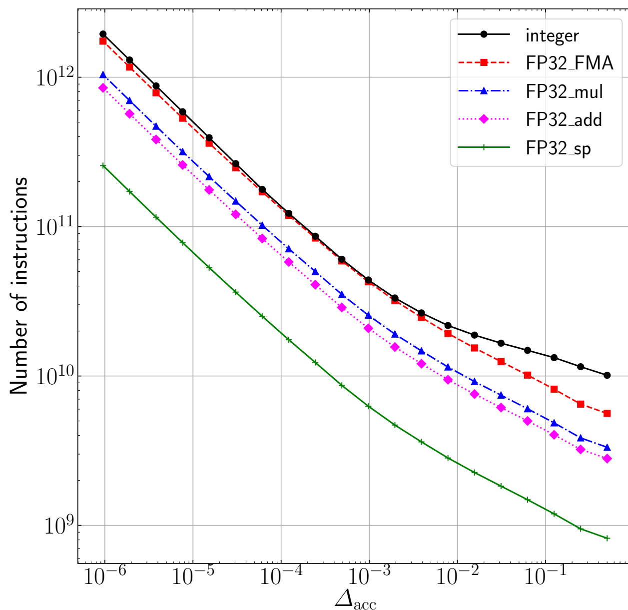

図 6　重力計算関数内で発行されたステップあたりの命令数．緑の十字は，単精度の逆数平方根 `rsqrtf()` にあたる．

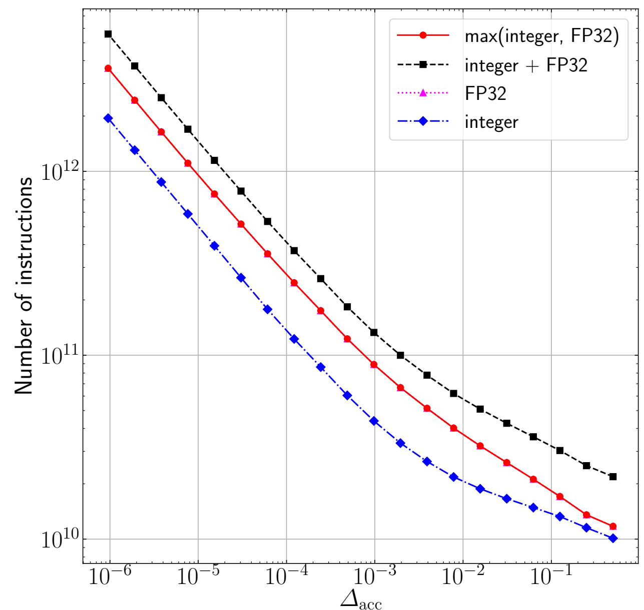

図 7　演算ユニットごとの発行命令数．整数演算（青の菱形）の命令数は単精度浮動小数点演算（マゼンタの三角形）よりも常に少ないため，赤丸で示した整数演算と単精度浮動小数点演算の命令数の最大値は単精度浮動小数点演算のものと一致する．

ある．図 5 に緑の三角形で示したように，両者の実行時間にはほぼ差がない．したがって，Pascal モードと Volta モードの実行時間の違いの最大の要因は `__syncwarp()` による同期コストであり，コンパイラによる最適化レベルに大きな違いはないことが示唆される．

マゼンタの菱形で示したツリー構造の構築関数においては `__activemask()` や Cooperative Groups を用いてのタイル同期を実行しているが，`__syncwarp()` だけを使用する関数と比較して極端に速度向上率が高くなるということはなかった．これは，実行時間の大半を CUB[*1] の `cub::DeviceRadixSort::SortPairs` 関数を用いての radix sort が占めていること，同期についても複数ブロックをまたいでの全体同期が多数回実行されるためにワープ内の同期の寄与が小さくなっているといったことが要因として考えられる．

4.2　Tesla P100 からの速度向上率の起源

本節では，Tesla P100 からの理論ピーク演算性能比を超える速度向上が実現された理由を探る．前節における議論から，$\Delta_{\rm acc}$ の値に依らず実行時間が支配的である重力計算関数についてのみ考える．Tesla V100 においては整数演算ユニットが浮動小数点演算ユニットから独立したため，整数演算と浮動小数点演算のうち実行時間の短い方の処理時間が隠蔽される可能性がある．この処理時間の隠蔽による高速化は演算律速な状況下でしか起こり得ないが，N 体シミュレーションは演算律速な計算の代表例であり，また直接法とは異なり整数演算も多数回発行するツリー法においては十分に起こり得る．

[*1] https://nvlabs.github.io/cub/

重力計算関数内で発行される命令数を取得するため，`nvprof` による測定対象となるメトリックとして `inst_integer`，`flop_count_sp_fma`，`flop_count_sp_add`，`flop_count_sp_mul`，`flop_count_sp_special` を指定し，Tesla V100 上で `GOTHIC` を 256 ステップ実行した．本測定では計算がシリアライズされることで通常実行時と比べて実行時間が大幅に伸びるため，自動最適化機能を無効とした．この際，$\Delta_{\rm acc}$ ごとに別途測定した最適なツリー構造の再構築間隔を用いることで，通常実行時とほぼ同じ処理がなされるようにした．**図 6** にステップあたりの各演算の実行回数を示す．特殊関数ユニットで計算される単精度の逆数平方根演算（緑の十字）のスループットは他の単精度浮動小数点演算に対して 1/4 倍であるが，命令数は 1/4 倍未満であるため，以降ではその実行時間は完全に隠蔽されると仮定して議論を進める．

図 7 に，単精度浮動小数点演算ユニットで実行される FMA（Fused Multiply–Add）命令，加算，乗算の命令数の総和（マゼンタの三角形）と整数演算ユニットで実行される整数演算の命令数（青の菱形）を比較した結果を示す．Pascal 世代以前であれば整数演算と単精度浮動小数点演算は同一の演算ユニットで実行されるため，全体の実行時間は整数演算と単精度浮動小数点演算の命令数の和（黒の四角形）によって決まる．図 7 においては，$\Delta_{\rm acc}$ の値に依らず単精度浮動小数点演算の命令数が整数演算の命令数よりも多いため，両者の最大値（赤丸）は単精度浮動小数点演算の命令数と完全に一致している．したがって，Volta 世代においては命令間の依存性などの要因が無ければ，整数

© 2018 Information Processing Society of Japan

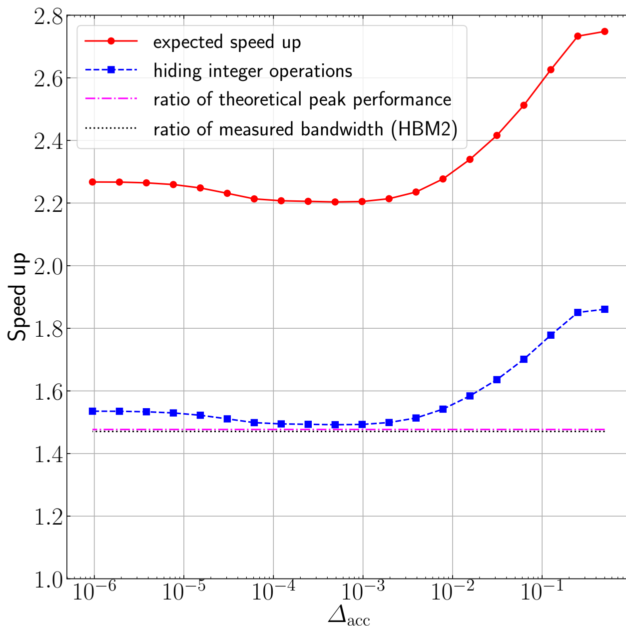

図 8 Telsa P100 から Tesla V100 への速度向上率の見積もり．マゼンタの一点鎖線，黒の点線はそれぞれ理論ピーク演算性能比，メモリバンド幅の実測値の比を，青の四角形は図 7 から見積もられる整数演算の隠蔽による速度向上率を示す．演算律速の場合においては，理論ピーク演算性能比と整数演算の隠蔽両方が寄与するため，両者の積である赤丸が理論予測となる．

演算の処理時間は単精度浮動小数点演算の実行時間によって隠蔽され得る．

ここでは理想的な場合を考え，整数演算の実行時間がまったく隠蔽されない場合（Tesla P100 に対応，図 7 の黒の四角形）と整数演算の実行時間が完全に隠蔽された場合（図 7 の赤丸）を想定して速度向上率を見積もり，**図 8** に示した．整数演算の処理時間の隠蔽による寄与（青の四角形）と理論ピーク演算性能比（マゼンタの一点鎖線）との積が，整数演算の実行時間の隠蔽を考慮した際の速度向上率の理論予測（赤丸）となる．この理論予測は，$\Delta_{\rm acc} \lesssim 10^{-3}$ の領域における速度向上率については図 2 に示した測定結果をよく説明しているが，$\Delta_{\rm acc} \gtrsim 10^{-3}$ における速度向上率の低下までは説明できない．

ところで，ここで測定した浮動小数点演算数と独立に重力計算関数の実行時間を測定することで，重力計算関数単体での演算性能が推定できる．このためには逆数平方根の計算に要する浮動小数点演算数を見積もる必要があるが，ここでは単精度浮動小数点の加算や乗算とのスループット比が 4 であることから，4 Flops 相当の演算であると仮定する [11]．こうして推定した重力計算関数の演算性能は**図 9** のようになり，$\Delta_{\rm acc} \lesssim 10^{-3}$ においては単精度理論ピーク演算性能の 45%にあたる 7 TFlop/s 程度の性能が出ていることが分かる．一方で $\Delta_{\rm acc} \gtrsim 10^{-3}$ の領域においては，$\Delta_{\rm acc}$ を増加させると演算性能も低下するという傾向が見えており，roofline model [22] によく似た結果が得られた．

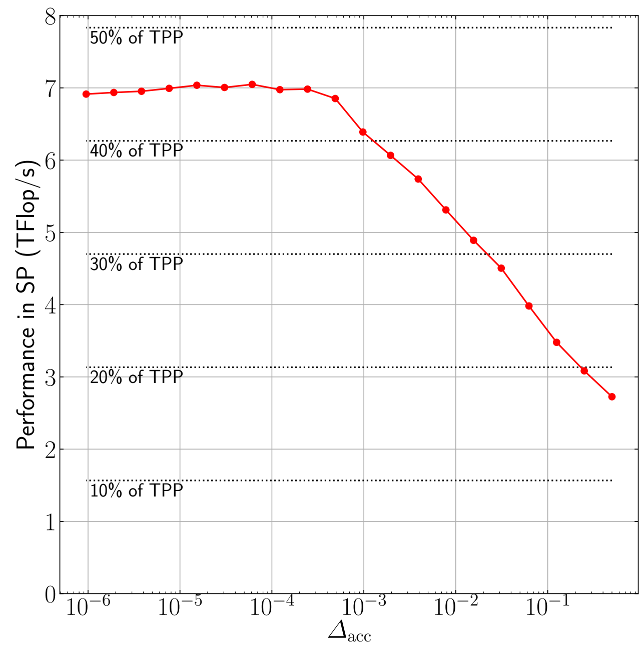

図 9 重力計算関数単体での演算性能．

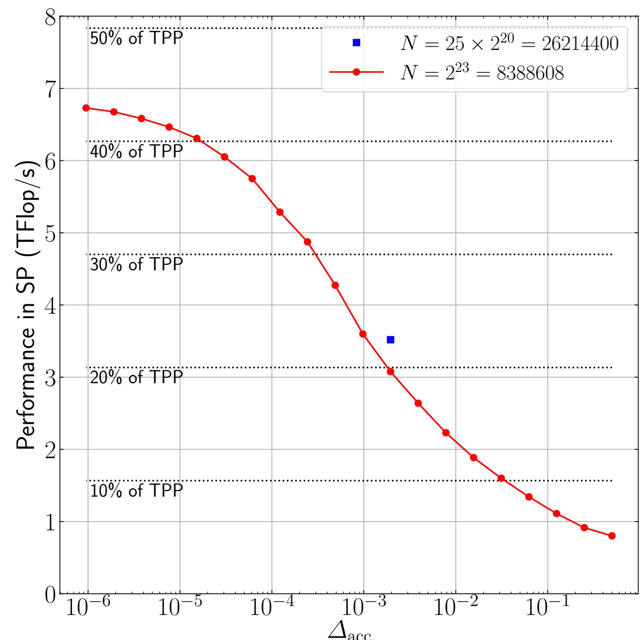

図 10 `GOTHIC` の演算性能．重力計算カーネル内の浮動小数点演算数と全体の実行時間から見積もった演算性能を，重力計算の精度を制御するパラメータ $\Delta_{\rm acc}$ の関数として示した．

ここで，$\Delta_{\rm acc}$ が B/F に対応すると考えれば，$\Delta_{\rm acc} \sim 10^{-3}$ が演算律速とメモリ律速の切り替え点にあたる．すると，$\Delta_{\rm acc} \lesssim 10^{-3}$ においては演算律速であるために整数演算の隠蔽が効いて最大で 2.2 倍の高速化が達成される．一方，$\Delta_{\rm acc} \gtrsim 10^{-3}$ ではメモリ律速であるため，グローバルメモリのメモリバンド幅の実測値の比である 1.4 倍 [9] 程度の高速化が達成される．以上の議論から，図 2 で見られた振る舞いが理解できる．

さらに，得られた浮動小数点演算数を 256 ステップ分の計

算に要した総実行時間で割り，GOTHIC の全体としての演算性能を見積もった結果を図 **10** に示す．図 9 よりも $\Delta_{\rm acc}$ に対する依存性が強くなっているのは，$\Delta_{\rm acc}$ が大きくなるほどツリーノードの処理などの重力計算以外の処理時間が全体に占める割合が多くなってくるためである．得られた単精度浮動小数点演算の演算性能は，$\Delta_{\rm acc} = 2^{-9} = 1.953125 \times 10^{-3}$ のときに，$N = 2^{23} = 8\,388\,608$ において 3.1 TFlop/s（理論ピーク演算性能の 20%），$N = 25 \times 2^{20} = 26\,214\,400$ のときに 3.5 TFlop/s（理論ピーク演算性能の 22%）であった．

5. まとめ

本研究では，Fermi 世代から Pascal 世代までの GPU 向けの最適化がなされていた重力ツリーコード GOTHIC を Volta 世代の GPU である Tesla V100 向けに移植し，その性能を評価した．Volta 世代においては，Pascal 世代までの GPU のプログラミングモデルとは異なり，ワープを構成する 32 スレッド間の暗黙の同期がなくなっている．このため，同期命令を適切な箇所に挿入するという方針でコードを改訂するか，コンパイル時に-gencode arch=compute_60,code=sm_70 を指定することで Pascal 世代以前と同じ動作パターンを強制するかのどちらかの対応をしなければならない．

Tesla V100 を用いて性能を測定したところ，$N = 2^{23} = 8\,388\,608$ 粒子で表現したアンドロメダ銀河モデルの計算に要したステップあたりの実行時間は，同期命令を追加した場合に 3.8×10^{-2} s，Pascal 世代以前と同じ動作パターンの場合に 3.3×10^{-2} s となった．後者の方が高速であるという結果は重力計算の精度には依らず，コンパイル時に-gencode arch=compute_60,code=sm_70 を指定することで約 1.2 倍の性能向上が得られることが分かった．また，Tesla P100 と比較すると Tesla V100 上では 1.4–2.2 倍の高速化が達成されることが分かった．得られた 2.2 倍という速度向上率は単精度理論ピーク演算性能比である 1.5 よりも大きく，これは Volta 世代において整数演算ユニットが浮動小数点演算ユニットから独立したことによる性能向上である．これにより，整数演算と単精度浮動小数点演算の同時実行が可能となり，整数演算の実行時間が単精度浮動小数点演算の実行時間によって隠蔽されることで，理論ピーク演算性能比を越える速度向上率が実現されたと考えられる．Tesla V100 上では最大 $N = 25 \times 2^{20} = 26\,214\,400$ 粒子の計算が実行でき，ステップあたりの実行時間は 2.0×10^{-1} s であった．これは単精度理論ピーク演算性能の 22%である 3.5 TFlop/s という性能にあたる．

謝辞　本研究は，TSUBAME 若手・女性利用者支援制度，学際大規模情報基盤共同利用・共同研究拠点，および，革新的ハイパフォーマンス・コンピューティング・インフラの支援による (課題番号: jh180045-NAH)．NVIDIA Tesla P100 向けの性能最適化および性能測定は，東京工業大学のスパコン TSUBAME3.0 を用いて行った．

参考文献

[1] Barnes, J. and Hut, P.: A hierarchical O(N log N) force-calculation algorithm, *Nature*, Vol. 324, pp. 446–449 (online), DOI: 10.1038/324446a0 (1986).

[2] Bédorf, J., Gaburov, E., Fujii, M. S., Nitadori, K., Ishiyama, T. and Portegies Zwart, S.: 24.77 Pflops on a Gravitational Tree-Code to Simulate the Milky Way Galaxy with 18600 GPUs, *ArXiv e-prints* (2014).

[3] Fardal, M. A., Guhathakurta, P., Babul, A. and McConnachie, A. W.: Investigating the Andromeda stream - III. A young shell system in M31, *Monthly Notices of the Royal Astronomical Society*, Vol. 380, pp. 15–32 (online), DOI: 10.1111/j.1365-2966.2007.11929.x (2007).

[4] Gaburov, E., Bédorf, J. and Portegies Zwart, S.: Gravitational tree-code on graphics processing units: implementation in CUDA, *Procedia Computer Science, volume 1, p. 1119-1127*, Vol. 1, pp. 1119–1127 (online), DOI: 10.1016/j.procs.2010.04.124 (2010).

[5] Geehan, J. J., Fardal, M. A., Babul, A. and Guhathakurta, P.: Investigating the Andromeda stream - I. Simple analytic bulge-disc-halo model for M31, *Monthly Notices of the Royal Astronomical Society*, Vol. 366, pp. 996–1011 (online), DOI: 10.1111/j.1365-2966.2005.09863.x (2006).

[6] Gilbert, K. M., Guhathakurta, P., Beaton, R. L., Bullock, J., Geha, M. C., Kalirai, J. S., Kirby, E. N., Majewski, S. R., Ostheimer, J. C., Patterson, R. J., Tollerud, E. J., Tanaka, M. and Chiba, M.: Global Properties of M31's Stellar Halo from the SPLASH Survey. I. Surface Brightness Profile, *The Astrophysical Journal*, Vol. 760, p. 76 (online), DOI: 10.1088/0004-637X/760/1/76 (2012).

[7] Hernquist, L.: An analytical model for spherical galaxies and bulges, *The Astrophysical Journal*, Vol. 356, pp. 359–364 (online), DOI: 10.1086/168845 (1990).

[8] Ibata, R. A., Lewis, G. F., McConnachie, A. W., Martin, N. F., Irwin, M. J., Ferguson, A. M. N., Babul, A., Bernard, E. J., Chapman, S. C., Collins, M., Fardal, M., Mackey, A. D., Navarro, J., Peñarrubia, J., Rich, R. M., Tanvir, N. and Widrow, L.: The Large-scale Structure of the Halo of the Andromeda Galaxy. I. Global Stellar Density, Morphology and Metallicity Properties, *The Astrophysical Journal*, Vol. 780, p. 128 (online), DOI: 10.1088/0004-637X/780/2/128 (2014).

[9] Jia, Z., Maggioni, M., Staiger, B. and Scarpazza, D. P.: Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking, *ArXiv e-prints* (2018).

[10] McMillan, S. L. W.: The Vectorization of Small-N Integrators, *The Use of Supercomputers in Stellar Dynamics* (Hut, P. and McMillan, S. L. W., eds.), Lecture Notes in Physics, Berlin Springer Verlag, Vol. 267, p. 156 (online), DOI: 10.1007/BFb0116406 (1986).

[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] Miki, Y. and Umemura, M.: GOTHIC: Gravitational oct-tree code accelerated by hierarchical time step controlling, *New Astronomy*, Vol. 52, pp. 65–81 (online), DOI: 10.1016/j.newast.2016.10.007 (2017).

[13] Miki, Y. and Umemura, M.: MAGI: many-component galaxy initializer, *Monthly Notices of the Royal Astro-*

© 2018 Information Processing Society of Japan

nomical Society, Vol. 475, pp. 2269–2281 (online), DOI: 10.1093/mnras/stx3327 (2018).
[14] Navarro, J. F., Frenk, C. S. and White, S. D. M.: Simulations of X-ray clusters, *Monthly Notices of the Royal Astronomical Society*, Vol. 275, pp. 720–740 (online), DOI: 10.1093/mnras/275.3.720 (1995).
[15] Nelson, A. F., Wetzstein, M. and Naab, T.: Vine–A Numerical Code for Simulating Astrophysical Systems Using Particles. II. Implementation and Performance Characteristics, *The Astrophysical Journal Supplement*, Vol. 184, pp. 326–360 (online), DOI: 10.1088/0067-0049/184/2/326 (2009).
[16] NVIDIA: *NVIDIA Tesla V100 GPU Architecture* (2017).
[17] Ogiya, G., Mori, M., Miki, Y., Boku, T. and Nakasato, N.: Studying the core-cusp problem in cold dark matter halos using N-body simulations on GPU clusters, *Journal of Physics Conference Series*, Vol. 454, No. 1, p. 012014 (online), DOI: 10.1088/1742-6596/454/1/012014 (2013).
[18] Sérsic, J. L.: Influence of the atmospheric and instrumental dispersion on the brightness distribution in a galaxy, *Boletin de la Asociacion Argentina de Astronomia La Plata Argentina*, Vol. 6, p. 41 (1963).
[19] Springel, V.: The cosmological simulation code GADGET-2, *Monthly Notices of the Royal Astronomical Society*, Vol. 364, pp. 1105–1134 (online), DOI: 10.1111/j.1365-2966.2005.09655.x (2005).
[20] Springel, V., Yoshida, N. and White, S. D. M.: GADGET: a code for collisionless and gasdynamical cosmological simulations, *New Astronomy*, Vol. 6, pp. 79–117 (online), DOI: 10.1016/S1384-1076(01)00042-2 (2001).
[21] Tenjes, P., Tuvikene, T., Tamm, A., Kipper, R. and Tempel, E.: Spiral arms and disc stability in the Andromeda galaxy, *Astronomy and Astrophysics*, Vol. 600, p. A34 (online), DOI: 10.1051/0004-6361/201629991 (2017).
[22] Williams, S., Waterman, A. and Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures, *Communications of the ACM*, Vol. 52, No. 4, pp. 65–76 (online), DOI: 10.1145/1498765.1498785 (2009).
[23] Xiao, S. and Feng, W.: Inter-block GPU communication via fast barrier synchronization, *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12 (online), DOI: 10.1109/IPDPS.2010.5470477 (2010).

付　録

A.1 Cooperative Groups による全体同期

CUDA 9 において導入された新機能として，Cooperative Groups による複数のブロックをまたいでの全体同期機能があげられる．本研究で扱った `GOTHIC` は Cooperative Groups が導入される以前から開発されていた重力ツリーコードであるので，複数のブロックをまたいでの全体同期については [23] によって提案された GPU lock-free synchronization を用いた実装がなされていた．

以下では，コードの改訂量が少なくすむ関数であるツリーノードの処理関数中の全体同期について，Cooperative Groups を用いたものと [23] によるものを用いた場合とで実行時間を比較する．また，ソースコードが複数ファイルに分割されている際に Cooperative Groups による全体同期機能を用いるためには，オブジェクトファイル生成時に`--device-c` を，コンパイルリンク時に`--device-link --output-file tmp.o` をつけて中間ファイルを生成した上で，最終的にこの中間ファイル tmp.o もリンクするという手順を踏む必要がある．そこで，Cooperative Groups による影響とコンパイル方法の変更による影響とを切り分けるために，元の実装を通常の方法でコンパイルリンクした場合，Cooperative Groups を用いた場合，そして元の実装を Cooperative Groups 向けの方法でコンパイルした場合の 3 通りの結果を比較する．

ステップあたりの実行時間は，元の実装の場合に 4.0×10^{-3} s，Cooperative Groups を用いた場合に 4.9×10^{-3} s，コンパイル方法だけを変えた場合に 4.4×10^{-3} s となった．また，各スレッドが使用するレジスタ数は元の実装の場合に 56 本，Cooperative Groups を用いた場合とコンパイル方法を変えた場合には 64 本となり，後者の場合には SM あたりのブロック数が 9 から 8 に減少していた．コンパイルリンク方法を変更しただけで性能が 1 割低下した要因としては，SM あたりのブロック数が減少したことで occupancy が低下したことが効いていると考えられる．また，この関数の中で全体同期は 21 回発行されていたため，Cooperative Groups を用いた際の全体同期 1 回あたりの追加コストは $\left(4.9 \times 10^{-3} - 4.4 \times 10^{-3}\right)/21 \simeq 2.3 \times 10^{-5}$ s と見積もられる．