New Astronomy 52 (2017) 65–81

Contents lists available at ScienceDirect

New Astronomy

journal homepage: www.elsevier.com/locate/newast

GOTHIC: Gravitational oct-tree code accelerated by hierarchical time step controlling

Yohei Miki [a,b,*], Masayuki Umemura [a,b]

[a] Center for Computational Sciences, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan
[b] CREST, JST, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan

HIGHLIGHTS

- We present a new gravitational octree code on GPU that adopts a block time step.
- It uses adaptive optimizations by monitoring the execution time of each function.
- The code achieves a 3–5 fold acceleration compared to the shared time step method.
- The averaged performance of the code is 10–30% of the theoretical peak performance.

ARTICLE INFO

Article history:
Received 22 June 2016
Revised 28 August 2016
Accepted 22 October 2016
Available online 24 October 2016

Keywords:
N-body simulation
Tree code
Block time step
GPU computing

ABSTRACT

The tree method is a widely implemented algorithm for collisionless *N*-body simulations in astrophysics well suited for GPU(s). Adopting hierarchical time stepping can accelerate *N*-body simulations; however, it is infrequently implemented and its potential remains untested in GPU implementations. We have developed a Gravitational Oct-Tree code accelerated by HIerarchical time step Controlling named `GOTHIC`, which adopts both the tree method and the hierarchical time step. The code adopts some adaptive optimizations by monitoring the execution time of each function on-the-fly and minimizes the time-to-solution by balancing the measured time of multiple functions. Results of performance measurements with realistic particle distribution performed on NVIDIA Tesla M2090, K20X, and GeForce GTX TITAN X, which are representative GPUs of the Fermi, Kepler, and Maxwell generation of GPUs, show that the hierarchical time step achieves a speedup by a factor of around 3–5 times compared to the shared time step. The measured elapsed time per step of `GOTHIC` is 0.30 s or 0.44 s on GTX TITAN X when the particle distribution represents the Andromeda galaxy or the NFW sphere, respectively, with $2^{24} = 16{,}777{,}216$ particles. The averaged performance of the code corresponds to 10–30% of the theoretical single precision peak performance of the GPU.

1. Introduction

Collisionless *N*-body simulations are frequently employed to investigate large scale structure formation and the formation and evolution of gravitational many-body systems such as galaxies. The acceleration of *N*-body particles is given by Newton's equation of motion,

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

where m_i, $\boldsymbol{r}_i$, and $\boldsymbol{a}_i$ are the mass, position, and acceleration of the ith particle of N particles, respectively. The remaining symbols are the gravitational constant G and the Plummer softening parameter ϵ. The latter is commonly adopted in collisionless *N*-body simulations to eliminate divergence due to division by zero. Hereafter, we call the particles which feel and cause gravitational force as *i*- and *j*-*particles*, respectively, and denote their total numbers N_i or N_j.

Employing a large number of *N*-body particles is essential for performing *N*-body simulations that resolve astrophysical phenomena. Since the computational cost of order $O(N_i N_j)$ is too high to investigate realistic phenomena in detail, many earlier studies have attempted to accelerate *N*-body simulations. Widely used algorithms for reducing the amount of computations are the particle-mesh method and the tree method (Hockney and Eastwood, 1988; Barnes and Hut, 1986). The computational complexity of the tree method is $O(N_i \log N_j)$ because the multipole expansion technique significantly reduces the contribution from *j*-particles.

* Corresponding author. Fax: +81298536406.
E-mail address: ymiki@ccs.tsukuba.ac.jp (Y. Miki).

http://dx.doi.org/10.1016/j.newast.2016.10.007

Many N-body simulations adopt a shared time step, and that means all N-body particles share the time step that is required to track the orbital evolution of the particle that evolves its physical quantities in the shortest time span. The timescale of the evolution is not uniform in most astrophysical phenomena; for example, the free-fall time, which is a measure for the timescale of evolution due to self-gravity, scales as the inverse square root of the mass density, and the mass densities have order-of-magnitude differences in typical systems. Therefore, adopting a shared time step causes unnecessary, additional computations to track the evolution of the system. To overcome the situation, a scheme in which every N-body particle has their own individual time step was introduced by Aarseth (1963). Because individual time steps for all particles is not suitable for parallelization, McMillan (1986) proposed the use of block time steps (or sometimes called hierarchical time steps) in which a group of particles has the same time step. Adopting block time steps can reduce the number of computations by reducing N_i.

Exploiting accelerator devices is another approach to reducing the time-to-solution. In the field of numerical astrophysics, a famous accelerator for N-body simulations is the GRAPE ("GRAvity PipE") series (Sugimoto et al., 1990; Ito et al., 1990, 1991, 1993; Fukushige et al., 1991, 2005; Okumura et al., 1993; Makino et al., 1997, 2003; Kawai et al., 2000; Umemura et al., 2012). Its high performance is a result of the pipelined and massively parallel architecture design, which enables massive parallelization of gravitational force calculations. Another widely used accelerator device is the Graphics Processing Unit (GPU), which was originally developed as a processor dedicated to image processing, and is equipped with a large number of computing units (typically a few hundred to a few thousand), suitable for parallel computing. The memory architecture of GPU mainly consists of shared memory and global memory: the former is fast and small on-chip memory ($\sim$ 1 MB per GPU), and the latter is slow and large off-chip memory ($\sim$ 1–10 GB per GPU, but about 100 times slower than the shared memory). Rapid performance improvement of GPUs and the development of General Purpose computing on GPU (GPGPU) have elevated GPUs to be the most attractive accelerators. Moreover, recent demands for power efficient devices strongly support the rapid development of accelerator devices such as GRAPE, GPU, and Intel Xeon Phi.

To promote GPU computing, NVIDIA provides the C/C++ like programming environment named Compute Unified Device Architecture (CUDA: NVIDIA, 2007, 2015). CUDA helps programmers implement GPU codes and optimize them by abstracting actual management of GPU cores and hiding differences among GPUs of various generations. For example, an essential building block of the Fermi generation GPUs is the streaming multiprocessor (SM), which is a group of 32 CUDA cores. In the Kepler generation of GPUs and Maxwell generation of GPUs SM are called SMX and SMM, respectively, and have 192 or 128 CUDA cores. For simplicity, we will refer to this fundamental group of CUDA cores as SM, irrespective of the GPU's generation. The fundamental parallelism in CUDA is thread parallelism, and a bunch of threads is called a block (typically 128–512 threads). Also, a group of blocks is called a grid; the hierarchical structure composed of the thread, block and grid is a key concept in CUDA. CUDA assigns multiple blocks to an SM for hiding latency to access memory and switch threads effectively. Since, in most applications, the number of threads per SM is sufficiently large compared to the number of CUDA cores per SM, all we have to do is to determine the number of threads per block. Through such abstractions of programming and the achieved high performance, GPU computing is now an important domain in high performance computing (HPC) community.

Many earlier studies showed that the tree method efficiently works on GPU(s) (Nakasato, 2012; Ogiya et al., 2013; Bédorf et al., 2012, 2014; Watanabe and Nakasato, 2014). However, none of the studies have coupled their tree method with the block time step on GPU. One difficulty when coupling the block time step with the tree code running on GPU is maintaining performance in the low N_i-regime. As mentioned above, the reduction of the time-to-solution by the block time step is due to the decrease of N_i. However, the performance of massively parallel architectures always drops in the low-number limit because only some of the cores perform any computations while others do not, leading to a waste of computing resources. In the typical implementation of a direct N-body code running on GPU, the critical number of particles required in order not to waste CUDA cores is 10^4 (Miki et al., 2012, 2013). A viable method to decrease the critical number is to adopt ij-parallelization (Nitadori et al., 2006; Nyland et al., 2007; Miki et al., 2012), by which multiple processors calculate the force on a common particle. Miki et al. (2012) showed that ij-parallelization can sustain the high performance of their direct N-body code down to $N \sim 10^3$ on NVIDIA Tesla C2070. An option to activate ij-parallelization may increase the performance of tree code on GPU that adopts the block time step.

In GPU computing, a bunch of threads, 32 threads in the case of CUDA called a warp, always execute the same operation concurrently. If two threads in a warp are forced to execute different operations due to conditional branching, then the threads run both operations. Since there are 32 threads in a warp, this behavior, named "warp divergence", may cause up to 32 times slow down of calculations in the worst case. Therefore, avoiding the warp divergence is one of the key strategies to accelerate calculations using GPU. In a case of the tree code runs on GPU, Ogiya et al. (2013) proposed an algorithm that reduces the warp divergence within the tree traversal and showed it improves the performance. On the other hand, concurrent operations by 32 threads present an opportunity to remove explicit synchronizations within a warp because they are implicitly synchronized. Synchronization is an inevitable operation for parallel computing to proceed properly; however, it often hinders achieving high performance. Hence, removing explicit synchronizations recovers high performance in parallel computing and reduces the time-to-solution. In N-body simulation with direct summation, Miki et al. (2012) demonstrated the benefits of removing explicit synchronizations, especially in the low N runs, where the contribution from synchronization grows.

There is further room for accelerating N-body simulations through automatic performance tuning (auto-tuning). Several examples of auto-tuning accelerating software libraries have been developed in the HPC community (e.g., Whaley et al., 2001; Frigo and Johnson, 2005). The primary purpose of auto-tuning is to provide performance portability on various architectures and to benefit from the rapid performance improvements of architectures without needing to significantly modify optimized codes. Another essential objective of auto-tuning is to ensure the high performance of the code irrespective of input. For example, the performance of sparse matrix-vector multiplications (SpMV) on GPU has a strong dependence on the input sparse matrix (Bell and Garland, 2008). Many studies showed the benefits of auto-tuning for SpMV (Reguly and Giles, 2012; Ashari et al., 2014; Liu and Vinter, 2015; Maggioni and Berger-Wolf, 2016). In astrophysics, Ishiyama et al. (2009); Ishiyama et al. (2012) achieved a good load balance for their massively parallel TreePM code by incorporating on-the-fly measurements for the execution time of each function within the simulation. Just like SpMV, the time-to-solution of the tree method are dependent on the initial data because the particle distribution determines the total number of calculated interactions. Introducing some adaptive features to the tree code would contribute to accelerating N-body simulations by reducing slowdowns in the computation due to the non-uniform particle distribution.

These considerations drove us to develop and test a tree code adopting a block time step that runs on the GPU. The name of the

code is GOTHIC (Gravitational Oct-Tree code accelerated by HIerarchical time step Controlling). The remainder of this paper is organized as follows. Section 2 introduces the implementation and optimizations of GOTHIC using CUDA. Section 3 presents results of performance measurements, and Section 4 contains discussions. Finally, Section 5 summarizes this work.

2. Implementation

This section describes our strategy, implementation, and optimizations in detail. In GOTHIC, all instructions are performed on GPU, just like Bonsai (Bédorf et al., 2012, 2014) to minimize communication between CPU and GPU. Also, all floating-point operations are performed in single precision because this provides sufficient accuracy to follow the time evolution of collisionless systems. Section 2.1 explains how to construct tree structure on GPU, and Section 2.3 presents the algorithm to calculate the gravitational force adopted in GOTHIC. Sections 2.4 and 2.5 introduce additional optimizations aiming to keep performance even in situations not suitable for GPU. Section 2.6 gives information on further optimization to reduce the time-to-solution of GOTHIC, rather the execution time of a specific function. Sections 2.2 and 2.7 present other information required to implement a tree code, and Section 2.8 shows additional tips and issues related to the Kepler generation GPUs.

2.1. Generating tree structure

The space-filling curve based construction of the tree structure, which represents the particle distribution as a logical structure, is performed by the GPU. In this study, we adopt the Peano–Hilbert space-filling curve (Sagan, 2012) to exploit its one-stroke sketch nature, which the more familiar Morton curve does not have. First, the GPU generates the Peano–Hilbert key for all N-body particles in the global memory of the device (see Appendix A for more details). Then, the N-body particles are sorted according to the Peano–Hilbert space-filling curve by using `cub::DeviceRadixSort::SortPairs` function provided in CUB[1] v1.5.1. Using the Peano–Hilbert curve guarantees that the particles near one another in memory space are also near one another in physical space. The relation between memory space and physical space is important when optimizing codes, as shown by Ogiya et al. (2013) for accelerating gravity calculations using the tree structure.

Next, the GPU links the Peano–Hilbert key with the tree structure. The Peano–Hilbert space-filling curve itself has a hierarchical structure. Dividing a cube into eight sub-cubes (i.e., generating an octree structure) corresponds to dividing the Peano–Hilbert key into eight equal parts (or finding seven partitions of the Peano–Hilbert key). Because increasing parallelism is essential to accelerating calculations using many-core architectures such as GPU, we construct the tree structure in a breadth-first manner. Checking multiple tree cells in parallel is possible. However, child cells of all checked cells must have serial numbers to identify them. Calculating prefix sums (Blelloch, 1990) is necessary to tag all tree cells consistently.

When calculating prefix sum within a warp in parallel, the implicit synchronization of 32 threads is an important feature to exploit. Since the warp shuffle instruction is available in GPUs starting with the Kepler generation, our implementation of parallel prefix sum calculation within a warp utilizes the warp shuffle instruction on the Kepler and Maxwell generation GPUs or the shared memory on the Fermi generation GPUs. Repeated executions of a parallel scan within a warp with the appropriate use of `__syncthreads()` and shared memory yield parallel prefix sums within a block. To implement parallel prefix sums within a grid, global synchronization of multiple blocks within a grid is necessary. GOTHIC adopts the GPU lock-free synchronization proposed by Xiao and Feng (2010) as a global synchronization mechanism. In the algorithm, all blocks within a grid must run simultaneously so as not to cause a deadlock. The `__launch_bounds__` qualifier is useful to control the number of concurrent blocks in the case that the register usage limits the number of concurrent blocks per SM. Also, `cudaFuncAttributes::numRegs` obtained by calling the `cudaFuncGetAttributes` function is helpful to judge whether the deadlock will occur just before calling the device function.

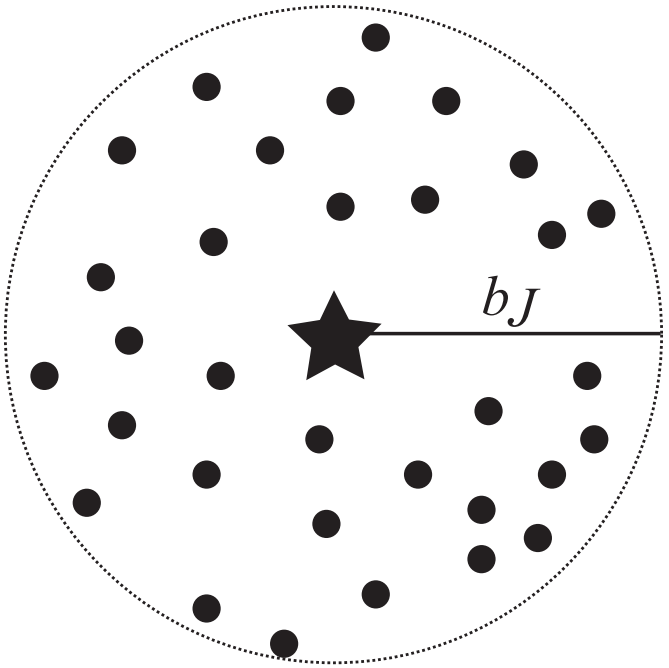

Fig. 1. Definition of a pseudo j-particle. The filled circles and the star indicate locations of real N-body particles and the corresponding pseudo j-particle, respectively. The dotted circle represents the size of the pseudo j-particle b_j.

Since GOTHIC adopts the monopole approximation for gravity calculation between an i-particle with a tree cell, introducing imaginary particles corresponding to tree cells can simplify the implementation of the function to calculate the gravitational force. After the Peano–Hilbert keys are associated with the tree structure, the GPU generates imaginary particles called pseudo j-particles and connect them with tree cells. The pseudo j-particle has information on mass m_j, position $\boldsymbol{r}_j$ and the size b_j; hereafter, the capitalized subscript indicates the index of the pseudo particles. The mass is the total mass of real N-body particles contained in the corresponding tree cell and the position is the center-of-mass of the particles. The size of the pseudo j-particle is defined as the radius of a sphere centered on $\boldsymbol{r}_j$ which can contain all N-body particles contained in the tree cell (see Fig. 1). All physical quantities of the pseudo j-particles must be recalculated at every time step to calculate gravitational force properly.

2.2. Multipole acceptance criterion

If a pseudo j-particle is far, then the gravity from the particle is calculated; if it is near, the tree cell is restricted to the lower level. To judge whether a pseudo j-particle is near or far, the Multipole Acceptance Criterion (MAC) is employed. The most simple MAC is

[1] http://nvlabs.github.io/cub/index.html.

opening angle criterion proposed by Barnes and Hut (1986):

$$\frac{b_J}{d_{ij}} \leq \theta, \tag{2}$$

where d_{ij} is the distance to the particle from an i-particle and θ is an accuracy controlling parameter.

Because the above MAC cannot directly control the accuracy with which the gravitational forces are calculated, more sophisticated MACs have been proposed. The MAC proposed by Warren and Salmon (1993); Salmon and Warren (1994) is as follows:

$$d_{ij} \geq \frac{b_J}{2} + \sqrt{\frac{b_J^2}{4} + \sqrt{\frac{3B_2}{\Delta_{\rm mul}}}}, \tag{3}$$

where $\Delta_{\rm mul}$ is an accuracy controlling parameter and

$$B_2 \equiv \sum_j m_j \left(\boldsymbol{r}_j - \boldsymbol{r}_J\right)^2. \tag{4}$$

The MAC defined by Eq. (3) ensures the required accuracy by monitoring the truncation error of the multipole expansion.

In addition, the acceleration MAC by Springel (2005) given by

$$d_{ij} \geq \left(\frac{G m_J {b_J}^2}{\Delta_{\rm acc} \left|\boldsymbol{a}_i^{\rm old}\right|}\right)^{1/4} \tag{5}$$

also gives the required accuracy, where $\boldsymbol{a}_i^{\rm old}$ is the acceleration of the i-particle in the previous time step and $\Delta_{\rm acc}$ is an accuracy controlling parameter. This MAC directly monitors the acceleration of each i-particle, and gives the appropriate accuracy of the acceleration specified by $\Delta_{\rm acc}$.

The best choice of MAC from the three above must be determined by experiments. In the case of a tree code running on CPU, Nelson et al. (2009) compared the elapsed time of each MAC as a function of achieved accuracy, and concluded that the acceleration MAC was the optimal choice. The performance of the MAC, however, should depend on the implementation of the function which calculates the gravitational acceleration and is optimized for a specific architecture, in our case, GPU. Comparing MACs is, therefore, still necessary for tree codes optimized for GPU and we will provide results of the comparison in Section 3.2.

2.3. Traversing tree structure

Increasing arithmetic intensity leads to performance improvements since hiding the latency to access global memory becomes much easier. To increase the arithmetic intensity in the kernel function, Ogiya et al. (2013) introduced the technique of "vectorization". Ogiya et al. (2013) adopted the depth-first search on-the-fly and the number of i-particles per thread is assumed to be $N_{\rm vec}(\geq 1)$. When judging whether the distance to a pseudo j-particle is far or near, they calculate the distance between the pseudo j-particle and $N_{\rm vec}$ i-particles one by one. A minimum of $N_{\rm vec}$ evaluations of distance is used for the distance judgment. The total number of interactions increases due to the minimum of $N_{\rm vec}$ evaluations; therefore, $N_{\rm vec}$ has some optimal value determined by balancing pros and cons of the effects by the vectorization.

During tree traversal when calculating the gravitational force, warp divergence occurs when some threads in a warp judge the distance to a pseudo j-particle to be sufficiently far while the remainder judge the distance to still be near. Ogiya et al. (2013) introduced "grouping" to reduce the warp divergence. In this step, they group the distance judgment into $N_{\rm grp}$ threads ($N_{\rm grp}$ must be smaller than 32 to utilize the implicit synchronization within a warp) by sharing the minimum distance to a pseudo j-particle from $N_{\rm vec}$ i-particles in $N_{\rm grp}$ threads. Just like $N_{\rm vec}$, there is also an optimal value of $N_{\rm grp}$.

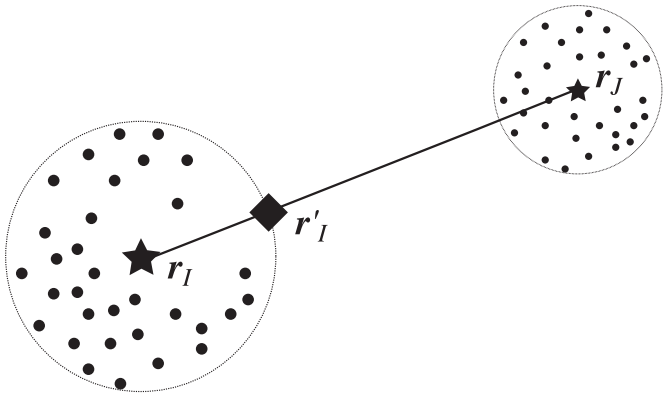

Fig. 2. Sketch of the distance evaluation between i-particles and a pseudo j-particle. Filled stars show positions of a pseudo i-particle ($\boldsymbol{r}_I$) and a pseudo j-particle ($\boldsymbol{r}_J$). Filled circles enclosed by a dotted circle centered on the pseudo i-particle are real i-particles. The filled diamond shows the possible nearest position of i-particles to the pseudo j-particle, $\boldsymbol{r}'_I$. The distance between the pseudo i-particle and pseudo j-particle is measured as $|\boldsymbol{r}_J - \boldsymbol{r}'_I|$.

In Ogiya et al. (2013), $N_{\rm vec}$ distance calculations by $N_{\rm grp}$ threads and $\log_2 N_{\rm grp}$ comparisons to group the judgement in $N_{\rm grp}$ threads are required to judge whether a specific pseudo j-particle is near or far. Here, we modify the vectorization method proposed by Ogiya et al. (2013) by using a breadth-first search. The vectorization in the original form requires $N_{\rm vec}$-times N_i particles for sustained performance. This diminishes the benefits of the block time step. Therefore, we adopt a compromise between an on-the-fly method and an interaction list method. In this method, a small sized interaction list is created in shared memory. Once the size of the interaction list reaches a certain predefined value, we calculate gravitational forces between i-particles and pseudo j-particles in the list and clear the list. By repeating the procedure, the gravity by all j-particles is properly calculated. The arithmetic intensity of the kernel function is determined by the capacity of the interaction list, which depends on the number of threads per block and the cache configuration of the shared memory.

To use the shared memory efficiently and reduce warp divergence, we adopt the grouping almost in the original form. Grouping the interaction list of $N_{\rm grp}$ threads leads to a $N_{\rm grp}$ times bigger list to be stored in the shared memory. We modify the algorithm for grouping the distance judgment to remove $\log_2 N_{\rm grp}$ comparisons as follows. Since the breadth-first search can access queued tree cells in parallel, distance evaluations to multiple pseudo j-particles can be performed at the same time. We introduce a pseudo i-particle shared by $N_{\rm grp}$ threads as shown in Fig. 2. The pseudo i-particles is to include all corresponding real i-particles by defining the appropriate radius b_I. There is some freedom in defining the center of the enclosing sphere $\boldsymbol{r}_I$: for example, the center of the smallest enclosing ball, the center-of-mass of real i-particles, or the geometric center of the enclosing rectangular cuboid (see Appendix B for more detail). The optimal choice to minimize the elapsed time of `GOTHIC` will be determined in Section 3.1 to provide the shortest elapsed time in micro-benchmarks. The distance between the pseudo i-particle and a pseudo j-particle is evaluated as the distance between an imaginary particle and the pseudo j-particle. The imaginary particle is set at the intersection of the surface of the pseudo i-particle with a line connects the pseudo i-particle with the pseudo j-particle, $\boldsymbol{r}'_I$:

$$|\boldsymbol{r}_J - \boldsymbol{r}'_I| \equiv \lambda r_{JI}, \tag{6}$$

where

$$\lambda = \begin{cases} 1 - \frac{b_I}{r_{JI}} & (b_I < r_{JI}), \\ 0 & (b_I \geq r_{JI}). \end{cases} \tag{7}$$

Introducing the pseudo *i*-particle is functionally the same as the vectorization and the grouping by Ogiya et al. (2013) because the distance between the pseudo *i*-particle and the *j*-particle is always smaller than that between all corresponding *i*-particles and the *j*-particle.

When traversing the tree structure in a breadth-first manner, many tree cells must be stored in a large buffer compared to one child cell stored under the depth-first search. The breadth-first search requires additional global memory allocation. Because the total capacity of the global memory on GPU is limited (e.g., 5 GB for NVIDIA Tesla M2090 and K20X with ECC enabled), sophisticated memory management is necessary. In order to allocate as large as possible a chunk of global memory for the buffer, we first query the unused capacity of the global memory using the `cudaMemGetInfo()` function and then allocate the buffer in the global memory. The next problem is the assignment of the buffer to each thread-block. In this study, the capacity of the shared memory sets the upper limit on the number of thread-blocks per SM to two. It determines the maximum number of thread-blocks which can run simultaneously, and we equally divide the buffer into the given number of pieces. The special register `%smid` acquired by the inline PTX function tells the ID of SMs, and is useful to assign unused parts of the buffer to a running thread-block. It should be noted that `%smid` is a volatile variable. Thus, a careful treatment is required to occupy and release the partitioned buffer correctly.

2.4. Splitting particle groups in low dense region

One of the shortcomings of the method introduced in Section 2.3 is an over-computation when *i*-particles in a low dense region are selected as a group of *i*-particles. To avoid this situation, we introduce a critical separation $r_{\rm crit}$ to judge whether to unify *i*-particles into a group or not. If the value is too large or too small, then the elapsed time will become longer due to over-computation or over-splitting of the kernel, respectively. The critical separation $r_{\rm crit}$ must be set carefully to minimize the elapsed time; however, it is impossible to determine the optimal value before the calculation because $r_{\rm crit}$ depends on the particle distribution which evolves in the simulation. This leads us to set $r_{\rm crit}$ through trial-and-error during the simulation. In other words, we apply auto-tuning to determine the optimal value of $r_{\rm crit}$. The strategy we adopt is to search for the optimal $r_{\rm crit}$ by minimizing the GPU time to calculate gravity using Brent's method (Press et al., 2007) and treating the GPU time as a function of $r_{\rm crit}$. Since the optimal value of $r_{\rm crit}$ would also depend on time, some perturbation on $r_{\rm crit}$ is additionally introduced. According to this scheme, $r_{\rm crit}$ automatically evolves to reduce the elapsed time.

2.5. Increasing parallelism in gravity calculation

Maintaining the high performance of the code down to the low N_i-regime is an essential point to achieve high performance with the block time step. However, this is difficult because a lack of parallelism reduces the GPU performance by wasting CUDA cores. The critical number of particles to saturate GPU performance is 10^4 in the case of direct *N*-body calculation (Miki et al., 2012, 2013). Some remedy should be introduced to limit the performance decrease in low N_i-regime. A straightforward remedy is introducing *ij*-parallelization to increase parallelism (Nitadori et al., 2006; Nyland et al., 2007; Miki et al., 2012). In the case of *ij*-parallelization, multiple threads share an *i*-particle and calculate gravity to the particle. As a result, we regain running CUDA cores and GPU performance even in the low N_i-regime.

Introducing *ij*-parallelization requires an implementation of a force accumulation process among multiple threads that share a common *i*-particle. In this work, we have implemented an essentially identical version of the algorithm proposed by Miki et al. (2012). In principle, either synchronization or exclusive control or both are inevitable to sum up the threads' results, and this always impedes the performance improvement in parallel computing. Miki et al. (2012) proposed an algorithm specialized for GPU computing to alleviate the burden of the force accumulation process. They remove explicit synchronization of multiple threads by aggressively utilizing the specification of CUDA that 32 threads in a warp always perform the same operation (implicit synchronization). Therefore, the number of threads that share an *i*-particle, *S*, must satisfy $S \leq 32$.

2.6. Tree rebuild interval

The cost of tree construction, $t_{\rm make}$, is not negligibly small compared to that of tree traversal, $t_{\rm walk}$, and there is no requirement to rebuild the tree structure every time step. Since the particle distribution is almost the same for two time steps in succession, reusing the old tree structure will not deteriorate $t_{\rm walk}$ without additional cost to rebuild the tree structure. The mismatch between the tree structure and the actual particle distribution would increase the execution time, and the timescale of the increase is a function of the time evolution of the particle distribution. There ought to be an optimal interval to rebuild the tree structure and finding it is a task suited to auto-tuning.

The code determines the rebuild interval *n* by guessing the total elapsed time $t_{\rm tot}$. The total elapsed time between the tree constructions is given by

$$t_{\rm tot} = t_{\rm make} + \sum_{i=1}^{n} t_{\rm walk}^{(i)}, \tag{8}$$

where $t_{\rm walk}^{(i)}$ is the execution time to calculate gravity in the *i*th step out of *n* steps which use the same tree structure.

Here, we introduce three toy models, a linear growth model, a power-law growth model, and a parabolic growth model, to guess $t_{\rm walk}^{(i)}$. In the first model, we assume $t_{\rm walk}$ grows as

$$t_{\rm walk}^{(i)} = t_1 + (i-1)\Delta t, \tag{9}$$

where t_1 and Δt are intercept and slope, respectively. The above fitting parameters are determined using the least squared method by monitoring the execution time in every time steps. Then, the total elapsed time is estimated as

$$t_{\rm tot} = t_{\rm make} + n(t_1 - \Delta t) + \frac{n(n+1)}{2}\Delta t. \tag{10}$$

To minimize the mean execution time $t_{\rm mean} \equiv t_{\rm tot}/n$, differentiate $t_{\rm mean}$ with respect to *n*:

$$\frac{d}{dn}\frac{t_{\rm tot}}{n} = -\frac{t_{\rm make}}{n^2} + \frac{\Delta t}{2}. \tag{11}$$

Therefore, the condition to get the extremum is

$$n^2 = \frac{2}{\Delta t} t_{\rm make}. \tag{12}$$

Furthermore, the second derivative with respect to *n* is evaluated as

$$\frac{d^2}{dn^2}\frac{t_{\rm tot}}{n} = \frac{2t_{\rm make}}{n^3}, \tag{13}$$

and is always positive meaning that Eq. (12) always minimizes the mean execution time.

The power-law and the parabolic growth models are shown in Appendix C. The model which gives the smallest reduced χ^2 value,

$$\chi_\nu^2 \equiv \frac{1}{\nu}\sum_{i=1}^{n}\left(\frac{t_{\rm walk,model}^{(i)} - t_{\rm walk,measured}^{(i)}}{\sigma_i}\right)^2, \tag{14}$$

is the most appropriate of the three choices. The degrees of freedom ν is $n-2$ (for the linear or power-law growth model) or $n-3$ (for the parabolic growth model), and we simply assume σ_i is unity.

2.7. Orbit integration

When the block time step is employed, every i-particle has its own time step. Since the adaptive, block time step is employed, we adopt a second-order Runge–Kutta method to integrate the particle orbit. In the prediction step, we update positions and velocities of all j-particles by

$$\boldsymbol{v}_j^{n+1/2} = \boldsymbol{v}_j^n + \frac{\Delta t_j^n}{2}\boldsymbol{a}_j^n, \tag{15}$$

$$\boldsymbol{r}_j^{n+1} = \boldsymbol{r}_j^n + \Delta t_j^n \boldsymbol{v}_j^{n+1/2}, \tag{16}$$

where $\boldsymbol{v}_j^n$ is the velocity of the jth particle at the nth time step, subscripts and superscripts indicate the index of particles and time step, respectively. We then calculate gravity from all j-particles to selected i-particles, and execute the correction step for the chosen i-particles as

$$\boldsymbol{v}_i^{n+1} = \boldsymbol{v}_i^{n+1/2} + \frac{\Delta t_i^n}{2}\boldsymbol{a}_i^{n+1}. \tag{17}$$

Because the above predictor–corrector method is not a symplectic integrator, it does not conserve the pseudo-Hamiltonian unlike the leap-frog method often employed with the shared, fixed time step.

For the comparison cases where the time step is shared and fixed, we adopt a second-order leap-frog method. In this case, orbit integration is performed as

$$\boldsymbol{v}_j^{n+1/2} = \boldsymbol{v}_j^{n-1/2} + \frac{\Delta t_j^n}{2}\boldsymbol{a}_j^n, \tag{18}$$

$$\boldsymbol{r}_j^{n+1} = \boldsymbol{r}_j^n + \Delta t_j^n \boldsymbol{v}_j^{n+1/2}. \tag{19}$$

For fixed shared timesteps, the Runge–Kutta integrator reduces to the leap-frog method.

2.8. Note for Kepler generation GPUs

Kepler generation GPUs support more functions that are useful in performance optimization compared to Fermi generation GPUs. One is warp shuffle instructions, which enable reading registers in other threads within a warp without using the shared memory. Warp shuffle instructions are heavily exploited in the calculation of parallel prefix sums and reductions since it is faster than accessing registers via shared memory. The read-only data cache is another feature to be noted. Just adding the `const __restrict__` qualifier tells the compiler to use a distinct cache in addition to L2 cache of the global memory. It effectively enlarges the capacity of cache and increases effective memory bandwidth.

A warp scheduler has two instruction dispatch units (IDUs) on Kepler generation GPUs (NVIDIA, 2012) while it has only one IDU on Fermi generation GPUs (NVIDIA, 2009). The presence of multiple IDUs within a warp scheduler causes scheduling issues if subsequent operations within a warp have mutual dependencies. Furthermore, Lai and Seznec (2013) reported that bank conflict of registers could occur among four banks on Kepler generation GPUs. Both the scheduling issue and bank conflict of registers cause a slowdown of operations on Kepler generation GPUs. Introducing instruction level parallelism can remove the dependency between subsequent operations and remove the scheduling issue created by multiple IDUs. We examined effects of increasing instruction level parallelism of multiple executions of fused multiply-add (FMA) instructions and direct N-body code without removing bank conflict of registers on NVIDIA Tesla K20. However, the performance did not improve; this suggests that a careful arrangement of registers to prevent bank conflicts is also necessary for further optimization. Because NVIDIA does not provide any tool or framework to arrange registers manually and CUDA shuffles locations of registers, we did not increase instruction level parallelism of subsequent computations or arrange locations of registers. Once such problems originated by hardware are resolved, we can expect the performance of `GOTHIC` to increase on Kepler or Maxwell generation GPUs.

3. Performance measurements of the code

3.1. Configuration of measurements

Performance measurements were done on HA-PACS (Highly Accelerated Parallel Advanced system for Computational Sciences) and a workstation at the University of Tsukuba. HA-PACS is composed of two clusters: the Base Cluster (BC) and the Tightly Coupled Accelerator (TCA). HA-PACS/BC and HA-PACS/TCA is equipped with NVIDIA Tesla M2090 (Fermi generation GPU) and NVIDIA Tesla K20X (Kepler generation GPU), respectively. NVIDIA GeForce GTX TITAN X (Maxwell generation GPU) is installed on the workstation. Table 1 lists the detailed information of the measurement environments. All environments have multiple GPUs, but we use only a single board of GPU on each machine in the measurements below.

Fundamental parameters of the code (e.g., the number of threads per block for each kernel function) are determined by micro-benchmarks for a Navarro–Frenk–White (NFW) sphere (Navarro et al., 1995, 1996), a Plummer sphere (Plummer, 1911), a King sphere (Michie, 1963; Michie and Bodenheimer, 1963; King, 1966) and a Hernquist sphere (Hernquist, 1990). All initial conditions used in this study are generated by the MAny-component Galactic Initial-conditions (MAGI) generator (Miki and Umemura, in preparation). Table 2 summarizes the resultant configuration for functions related to the tree structure. Obviously, optimal values exist for each function (`walkTree` executes tree traversal, `makeTree`, `linkTree`, and `trimTree` build tree structure, `calcMAC` calculates physical quantities of pseudo j-particles, and `genPHkey` translates the position of an i-particle to a Peano–Hilbert key). The adopted enclosing ball for `walkTree` is the efficient bounding sphere (Ritter, 1990) on GTX TITAN X, while M2090 and K20X use the sphere centered on the geometric center of the enclosing rectangular cuboid.

3.2. Measured elapsed time

First, we investigated relations among the accuracy controlling parameters of three MACs (Section 2.2), the resultant accuracy of gravity calculation and the elapsed time on various generations of GPUs. This is similar to the evaluation of a tree code performed by Nelson et al. (2009). Fig. 3 shows the result in the case of an NFW sphere with $2^{23} = 8{,}388{,}608$ particles. The cutoff radii of the density profile and the length of the Plummer softening are $5r_s$ and $r_s/64$, respectively, where the scale length r_s is set to unity. The elapsed time is evaluated as the wall clock time per time step (total number of time steps is fixed to 1024) to include the effects of auto-tuning; it also includes the time required to read/write files and allocate/deallocate memory. The accuracy of the gravity calculation is evaluated as a relative error of acceleration in the tree code $\boldsymbol{a}_i^{\rm tree}$ compared to acceleration in the direct N-body code, $\boldsymbol{a}_i^{\rm direct}$, where the subscript i indicates the index of the N-body particles. Upper and lower panels of the figure present the results for the 99 percentile error and median error, respectively. In other words, the points trace the loci at which 99% (50%) of N-body particles have a smaller error of the acceleration than the plotted

Table 1
Measurement environment.

System	HA-PACS/BC	HA-PACS/TCA	Workstation
Number of nodes	268	64	1
CPU	Intel Xeon E5-2670	Intel Xeon E5-2680 v2	Intel Xeon E5-2640 v3
	8 cores, 2.6 GHz	10 cores, 2.8 GHz	8 cores, 2.6 GHz
	2 sockets per node		2 sockets
RAM	DDR3-1600, 8 channels	DDR3-1866, 8 channels	DDR4-2133, 8 channels
	128 GB per node		64 GB
GPU	NVIDIA Tesla M2090	NVIDIA Tesla K20X	NVIDIA GeForce GTX TITAN X
	512 cores, 1.3 GHz	2688 cores, 732 MHz	3072 cores, 1 GHz
	4 boards per node		2 boards
Video RAM	6 GB (GDDR5, ECC on) per GPU		12 GB (GDDR5) per GPU
C Compiler	icc 15.0.5.223 (gcc 4.4.7 compatibility)		gcc 4.8.5
CUDA Toolkit	7.5.17		

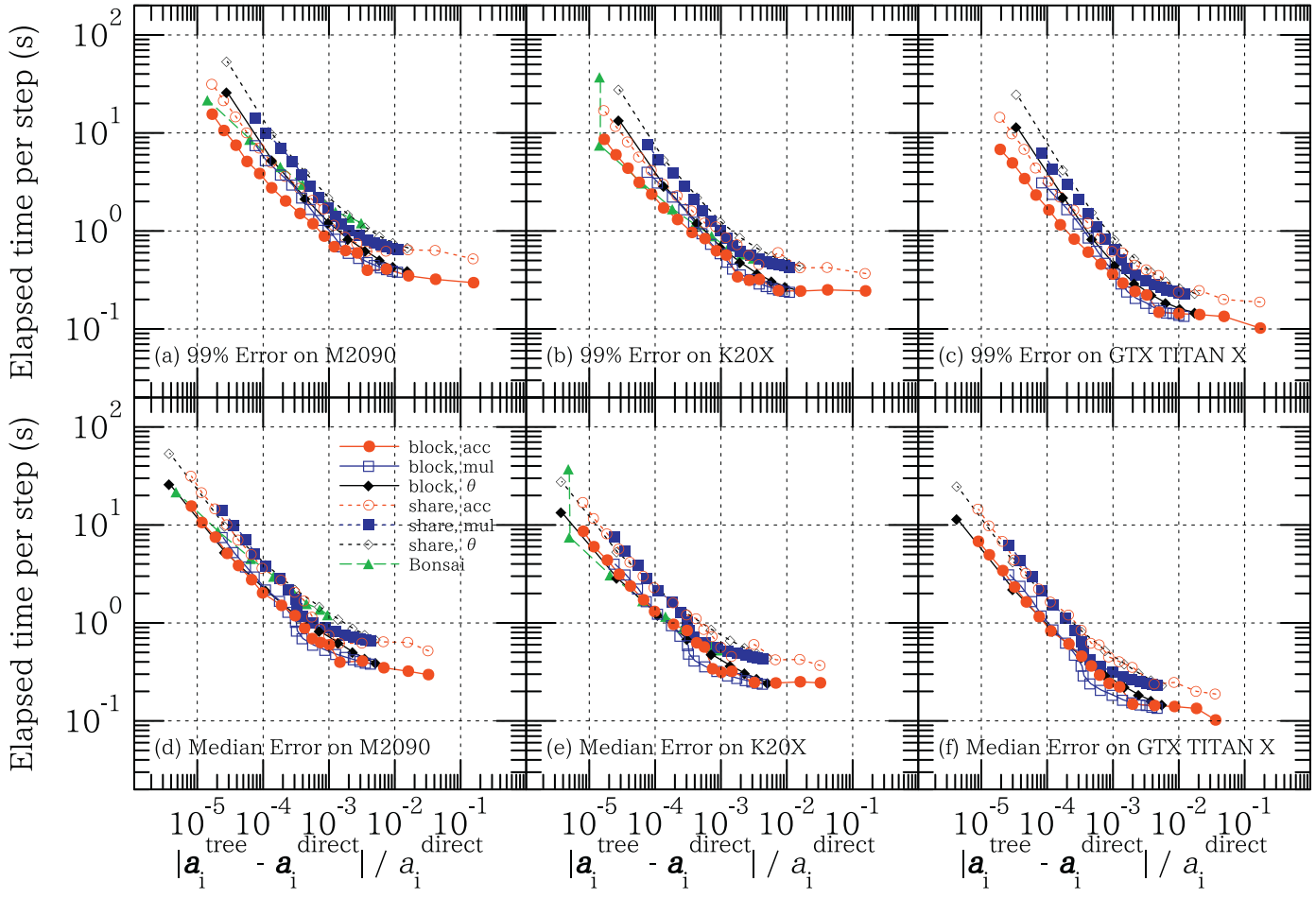

Fig. 3. Elapsed time per step as a function of force accuracy. Distribution of the N-body particles is an NFW sphere with 2^{23} = 8,388,608 particles. Solid and dotted lines with symbols are results of the block time step and shared time step, respectively. Each symbol indicates different MACs: red circles are acceleration MAC (Springel, 2005), blue squares are multipole MAC (Warren and Salmon, 1993), and black diamonds are opening angle (Barnes and Hut, 1986). The green triangles with dashed line show the elapsed time of the public code `Bonsai` (Bédorf et al., 2012, 2014). Values of the accuracy controlling parameters are 2^{-2}, 2^{-3}, ⋯, 2^{-19} for the acceleration MAC and the multipole MAC, 0.9, 0.8, ⋯, 0.1 for the opening angle and `Bonsai` from right to left. Upper and lower panels show the measured elapsed time against 99% error and median error of acceleration as a vector, respectively. Each panel exhibits benchmark results on different GPUs: left (M2090), middle (K20X), and right (GTX TITAN X).

value for each MAC in the upper (lower) panels. The figure clearly reveals the block time step (solid lines) is roughly twice as fast as the shared time step (dotted lines). The block time step with the acceleration MAC (red filled circles with solid line) has the shortest elapsed time in most cases. The multipole MAC (blue squares) is sometimes the optimal choice, especially with lower accuracy, and its performance with higher accuracy is comparable to that of the opening angle (black diamonds).

We have also compared the performance of `GOTHIC` with the public code `Bonsai`[2] (Bédorf et al., 2012, 2014, green triangles) which runs on the Fermi and Kepler generation GPUs. On M2090, the performance measurement with $\theta = 0.1$ for `Bonsai` was not completed because the computation time was too long. In all cases, `GOTHIC` with acceleration MAC and block time step (i.e., fastest configuration) was faster than `Bonsai` except for the case for

[2] https://github.com/treecode/Bonsai.

Table 2
Configuration of thread-blocks.

Function	GPU[a]	T_{tot}[b]	T_{sub}[c]	S[d]	R[e]
walkTree	M2090	256	32	1	63
	K20X	512	32	1	64
	TITAN X	512	32	4	64
makeTree	M2090	128	8		53
	K20X	128	8		49
	TITAN X	128	8		64
linkTree	M2090	128			27
	K20X	256			27
	TITAN X	256			23
trimTree	M2090	128			18
	K20X	128			22
	TITAN X	128			22
calcMAC	M2090	128	32		59
	K20X	128	32		55
	TITAN X	256	32		64
genPHkey	M2090	256			36
	K20X	1024			40
	TITAN X	1024			40

[a] Name of GPU.
[b] Number of threads per block.
[c] Number of threads share operations.
[d] Number of threads share an i-particle.
[e] Register usage per thread.

which the median force error was less than $\sim 10^{-5}$ on K20X. The figure clearly shows that the improvements of GOTHIC with respect to Bonsai are more significant on M2090 compared to K20X. This was expected, because Bédorf et al. (2014) performed sophisticated optimizations focused on the Kepler generation GPUs while we omit some optimizations (see Section 2.8). In other words, the performance improvements of GOTHIC from Bonsai on the Kepler generation of GPUs would increase if we introduced optimizations highly focused on the Kepler generation of GPUs. The typical accuracy for N-body simulations on galactic scales is around 10^{-3}, which corresponds to a value of θ of 0.5–0.7. In such realistic parameter regions, GOTHIC is a few time faster than Bonsai on M2090.

The NFW sphere is not suitable for evaluating effects of the block time step owing to its simple density profile. A more complicated particle distribution having a wider dynamic range in the temporal domain of the orbit evolution of individual particles is be a better choice for performance measurements to examine effects of the block time step. In order to measure the performance in a realistic distribution, we generate a model of the Andromeda galaxy (M31). The mass distribution model of M31 is given by Geehan et al. (2006); Fardal et al. (2007). Its composition is a dark matter halo with an NFW profile (the mass is $8.11 \times 10^{11}\,M_\odot$ and the scale length is 7.63 kpc) with 7,730,866 particles, a stellar bulge with a Hernquist profile (the mass is $3.24 \times 10^{10}\,M_\odot$ and the scale radius is 0.61 kpc) with 308,853 particles, and an exponential disk (the mass is $3.66 \times 10^{10}\,M_\odot$, the scale length is 5.4 kpc, and the scale height is 0.6 kpc) with 348,889 particles. The total number of N-body particles is $2^{23} = 8{,}388{,}608$, the masses of all N-body particles are identical and the Plummer softening length is set to 16 pc. On M2090, a performance measurement with Δ_{mul} of 2^{-19} for GOTHIC with the multipole MAC and the shared time step was not finished due to the limitation of the execution time on HA-PACS. Fig. 4 shows the results of the measurements. Again, the block time step with the acceleration MAC achieves the best performance in most cases. The performance gain of the block time step is significantly greater than that for a pure NFW sphere (Fig. 3). This is because additional components (the bulge and the disk) make the density profile steeper. A steeper density profile means a wider range of time steps of N-body particles since the free-fall time, one of the typical time scales of the system, is proportional to the inverse square root of the volume density. Indeed, the number of time step levels increases from four for the NFW sphere to five for the M31 model. The block time step with the acceleration MAC (red filled circles) achieves the shortest elapsed time in most cases, and is always faster than Bonsai (green triangles). On M2090, the performance measurement with θ of 0.1 for Bonsai was not completed due to exceeding the maximum execution time on HA-PACS. Since the performance improvements from the shared time step are more significant compared to the pure NFW model, the speed increase of GOTHIC compared to Bonsai is greater in the case of the Andromeda galaxy model compared to the NFW model.

3.3. Benefits from block time step

To assess benefits of adopting the block time step in detail, Fig. 5 shows the speed up of the block time step from the shared time step in the case of M31. The block time step results in two times faster completion compared to the shared time in all cases. In galactic scale N-body simulations, the typical value for θ is 0.5–0.7. Corresponding values of Δ_{acc} and Δ_{mul} which give similar accuracy are from 2^{-8} to 2^{-6} and from 2^{-5} to 2^{-2}, respectively (see Fig. 4). For such a typical accuracy, adopting a block time step results in about 2–5 times speed up for all three MACs on M2090, K20X, and GTX TITAN X. The amount of speed up tends to improve with increasing values of the accuracy-controlling parameters (i.e., the decreasing of the accuracy). When increasing the accuracy of gravity calculations, the number of calculations in high density regions increases because many particles are located near each other. Since the speed up of the block time step comes from the reduction of calculations in the low density regions, this increase in calculations weakens the benefits of adopting the block time step.

Hereafter, we regard the block time step with the acceleration MAC as a fiducial configuration, and go into more detail about the results from this configuration. Fig. 6 shows a breakdown of the execution time of various functions during the first 101 steps of the benchmark with $\Delta_{acc} = 2^{-7} = 7.8125 \times 10^{-3}$ on K20X. The initial condition of the system is a model of M31 in dynamical equilibrium with $2^{23} = 8{,}388{,}608$ particles. A slightly slow execution at the first step pushes back the first tree reconstruction to the 26th step; thereafter, the execution times of all functions settle into a regular repeating pattern because the system is in dynamical equilibrium. The execution time for calculating gravity (red circles) is, for the most part, the dominant contribution to the total execution time. For the case of the model of M31, there are three distinct ranges of execution times for calculating gravity; the fast steps with execution times in the range 4×10^{-3} s – 2×10^{-2} s, steps with intermediate execution times in the range 0.15 s – 0.4 s, and the slow steps with execution times of ~ 2 s). We group the steps in these ranges and label them as “FSeq” (fast sequence), “ISeq” (intermediate sequence), and “SSeq” (slow sequence), respectively. The decrease in the number of steps with execution times above 1 s (FSeq) to ten times during the first 101 steps is a consequence of the block time step reducing the number of calculations for slowly moving i-particles. This is the main reason for the acceleration by the block time step. The achieved mean elapsed time per step is 0.33 s, and is a little above 10% of the execution time to calculate gravity in the SSeq. The nearly fixed cost to calculate the position and mass of pseudo j-particles (black crosses), which is 3.2×10^{-2} s, sometimes becomes the most time-consuming function at a given time step. This suggests that further optimization of that function might also accelerate the code. Performing a more precise time integration is also possible without worsening the total elapsed time. For example, one could increase the number of i-particles at the cost of an increase in the execution time to calculate gravity. Unless the increase of the execution time in the

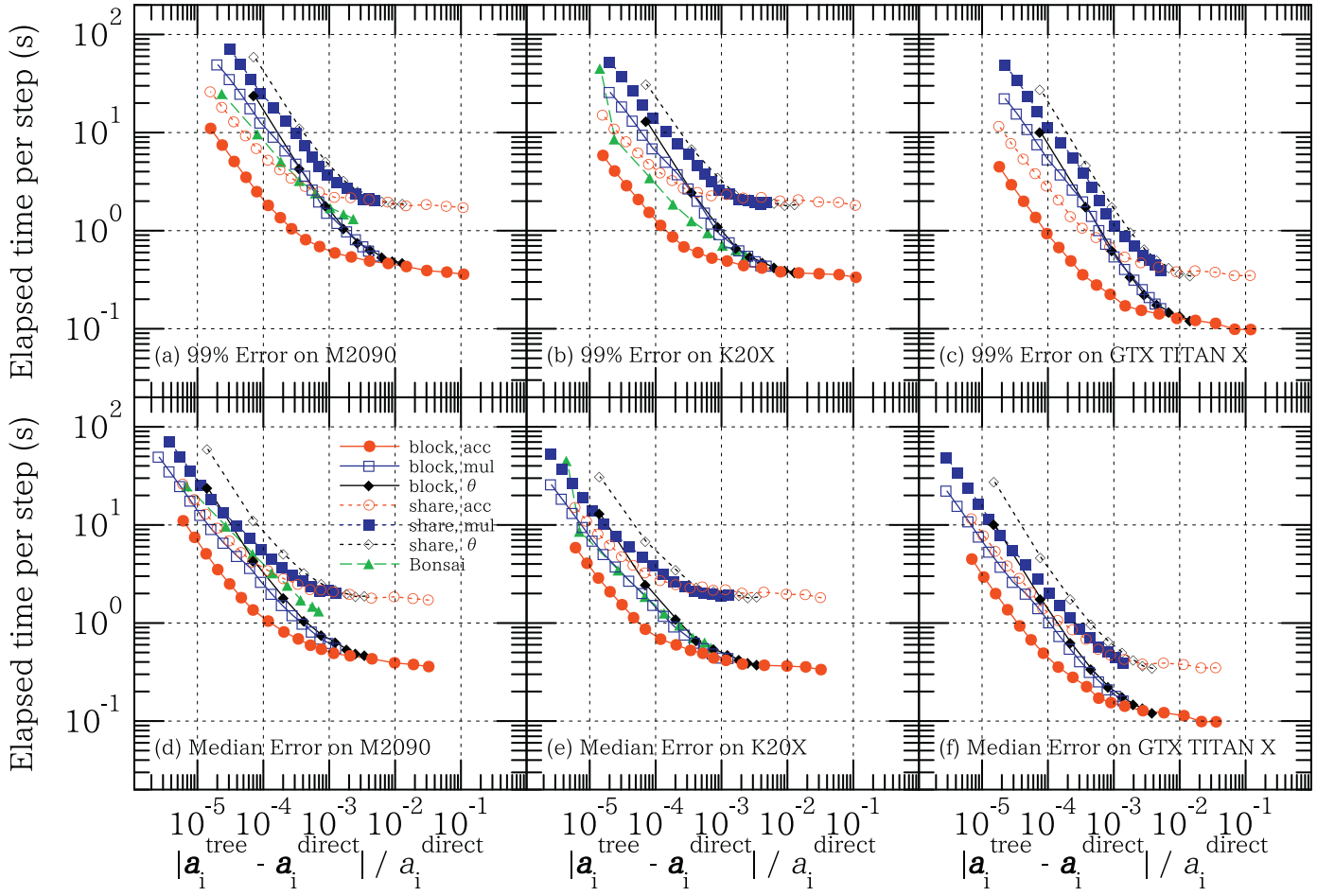

Fig. 4. Elapsed time per step as a function of force accuracy. The distribution of the N-body particles represents the spiral galaxy M31 with 2^{23} = 8,388,608 particles. Symbols, lines and panels are the same as those in Fig. 3.

FSeq is much greater than that the execution time to update j-particles, this would not increase the total elapsed time since the total elapsed time is still dominated by the execution of the SSeq in the gravity calculation is the main reason for the acceleration using the block time step. The costs for correcting the velocity of i-particles (black triangles) roughly fall into three sequences as well, with execution times of 5×10^{-5} s, 2.6×10^{-3} s, and 5×10^{-3} s. This implies that the number of i-particles within each sequence is fairly constant and suggests the scheme is successfully reducing the calculations of gravity for i-particles in the low density regions. The required time to predict position and velocity of j-particles (green crosses) is almost constant at 4.6×10^{-3} s, roughly the same as the slowest sequence for the corrector, in every time step. This is because the number of j-particles is always equal to $N_{\rm tot} = 2^{23}$.

The mean interval between successive tree reconstructions is about 12 steps. The costs of functions related to tree reconstruction (generation and sorting Peano–Hilbert keys, sorting N-body particles using Peano–Hilbert key, tree construction, and split i-particle groups in the low dense region) are almost independent of the particular time step. Because the radix sorting of 32-bit integers with 64-bit keys, which takes about 0.1 s, is the limiting process, further acceleration of the sorting library is essential to reduce the cost to reconstruct tree structures. The execution of the SSeq of the tree traversal and tree reconstruction often form a pair. Because tree construction is an order of magnitude faster than tree traversal, even a tiny increase in the cost to traverse the tree structure is greater than the cost of the tree reconstruction, and thus, the execution of the SSeq of tree traversal becomes a trigger to rebuild the tree structure.

3.4. Dependence on number of N-body particles

To examine the effects of ij-parallelization, we measured elapsed time while varying the number of i-particles, N_i, and keeping the total number of N-body particles fixed at $N_{\rm tot} = 2^{23}$ = 8,388,608. Fig. 7 presents the results for varying number of threads that share a common i-particle, S, on M2090, K20X, and GTX TITAN X. The elapsed time monotonically decreases with N_i. This feature is associated with the reason for the acceleration by the block time step, and roughly scales as N_i^1 if $N_i \gtrsim 10^4/S$ except for $N_i \sim N_{\rm tot}$. The steep increase at $N_i \sim N_{\rm tot}$ for all cases except for $S = 32$ is related to gravity calculations for i-particles in the lowest density regions. As noted in Section 2.4, GOTHIC tends to increase the number of interactions in the low density regions and this causes an increase in the elapsed time. Because $T_{\rm sub}/S = 32/S$ particles share the tree traversal, the steepness of the increase becomes weaker with greater S and vanishes for $S = 32$. Also, particles in the lowest density regions have the longest free-fall time and would have the longest time step; therefore, they would not be selected as i-particles if $N_i < N_{\rm tot}$, and this makes the increase of elapsed time steeper. If further optimizations or another algorithm succeeded in reducing the steep increase of the elapsed time at $N_i \sim N_{\rm tot}$, then the total elapsed time GOTHIC could be significantly decreased.

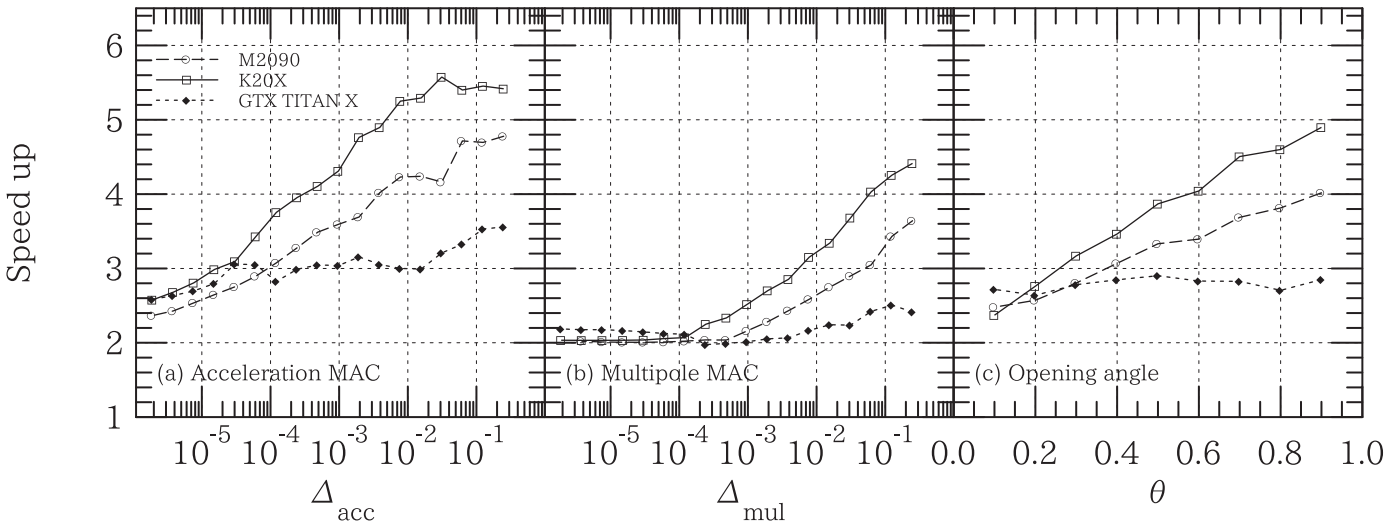

Fig. 5. Speed up of the block time step compared to the shared time step as a function of the accuracy controlling parameter. The particle distribution is M31 by 2^{23} = 8,388,608 particles. The open circles with the dashed line, the open squares with the solid line, and the filled diamonds with the dotted line in each panel show the speed up on M2090, K20X, and GTX TITAN X, respectively. Each panels presents different MACs: (a) acceleration MAC (Springel, 2005), (b) multipole MAC (Warren and Salmon, 1993), and (c) opening angle (Barnes and Hut, 1986).

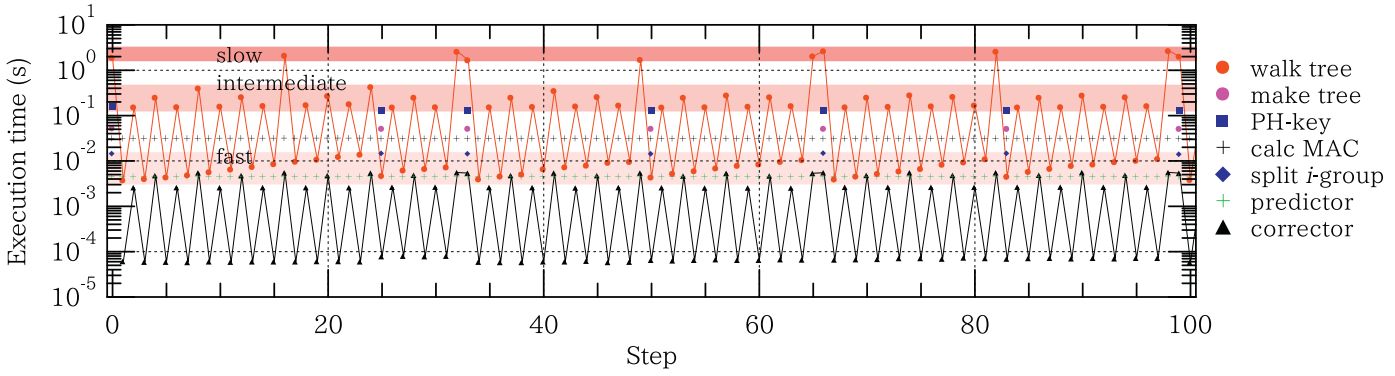

Fig. 6. Execution time of each function on K20X as a function of the time step. The particle distribution is M31 with 2^{23} = 8,388,608 particles. The execution time of the function for gravity calculation (red circles connected by red line), tree construction (magenta circles), generation and sorting Peano–Hilbert keys with particles (blue squares), calculating position and mass of pseudo *j*-particles (black crosses), splitting *i*-particles groups (blue diamonds), predicting *j*-particles' position and velocity (green crosses), and correcting velocity of *i*-particles (black triangles connected by black line) are plotted as a function of time steps. The slow, intermediate, and fast sequences are highlighted by bands in three shades of red.

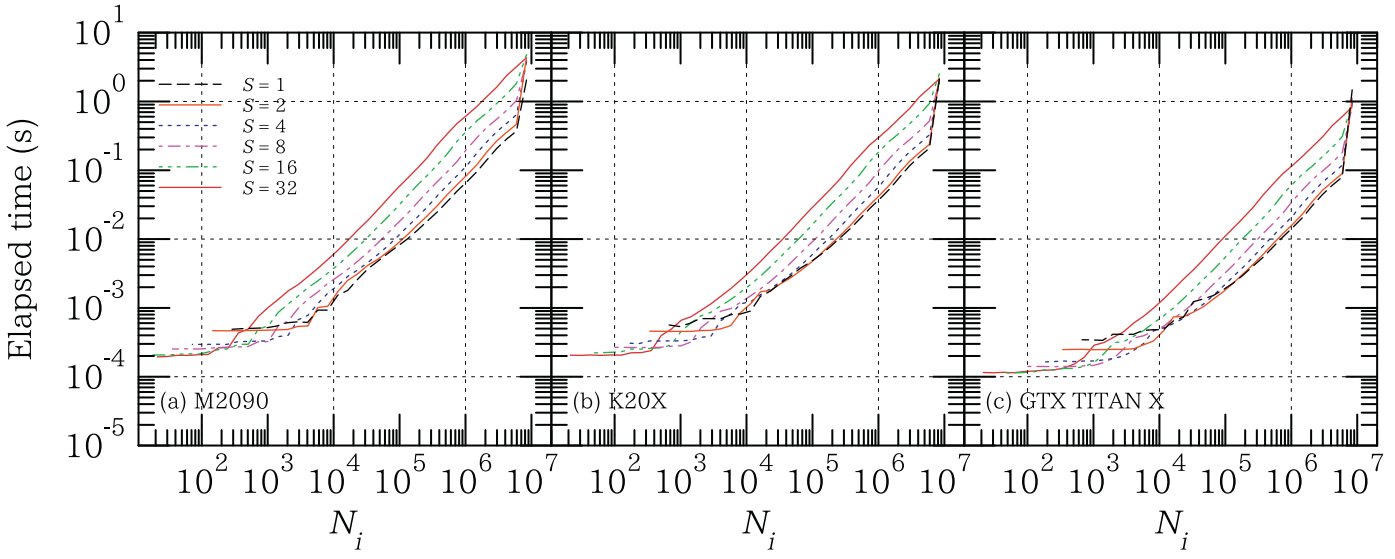

Fig. 7. Dependence on the number of *i*-particles N_i where the total number of *N*-body particles is 2^{23} = 8,388,608. The black dashed line ($S = 1$), the red solid line ($S = 2$), the blue dotted line ($S = 4$), the magenta dot-dashed line ($S = 8$), the green triple-dot-dashed line ($S = 16$), and the brown solid ($S = 32$) line represent the elapsed time for the number of threads that share a common *i*-particle *S*. Each panel reveals results on different generation of GPUs: (a) M2090, (b) K20X, and (c) GTX TITAN X.

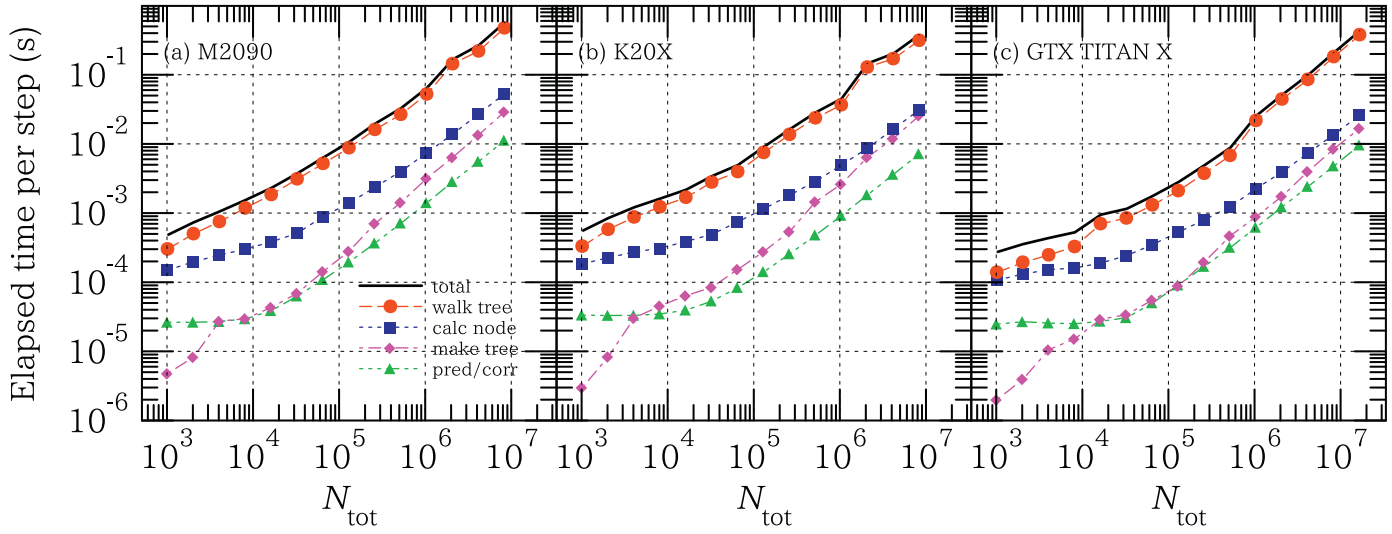

Fig. 8. Breakdown of the elapsed time of GOTHIC as a function of the total number of N-body particles N_{tot}. Each panel shows the elapsed time of functions for gravity calculation (red circles with dashed line), calculating position and mass of pseudo j-particles (blue squares with dotted line), tree construction (magenta diamonds with dot-dashed line), orbit integration (green triangles with triple-dot-dashed line), and sum of them (black solid line). The particle distribution is the M31 model and each panel shows results on different GPUs: (a) M2090, (b) K20X, and (c) GTX TITAN X.

The critical number $10^4/S$, which separates the monotonic decrease with N_i and the constant elapsed time irrespective of N_i, is determined by the number of running CUDA cores. Because the number of thread-blocks per SM is two, the number of threads per block is 256 or 512, and the number of SMs per device is around 20. The number of threads to saturate CUDA cores is given by the product of these three factors and is around 10^4. Introducing ij-parallelization activates S times more threads compared to simple i-parallelization. These two properties result in the critical number being $10^4/S$. The origin and value of the critical number are same for the case of direct summation (Miki et al., 2012).

The dependence of GOTHIC on the number of N-body particles is the final concern we address. Fig. 8 presents the elapsed time as a function of the total number of N-body particles with $\Delta_{acc} = 2^{-7} = 7.8125 \times 10^{-3}$ on M2090, K20X, and GTX TITAN X. Contributions of each function are measured as the elapsed time averaged by 1,024 steps. The number of N-body particles is changed from $2^{10} = 1{,}024$ to $2^{24} = 16{,}777{,}216$. The two-fold greater global memory on GTX TITAN X compared with others enables it to perform N-body simulation with 2^{24} particles that could not run on M2090 or K20X. Traversing the tree structure (red circles with dashed line) always dominates the execution time and scales roughly as N_{tot} if $N_{tot} \gtrsim 10^5$ on all GPUs. It is slightly weaker than the expected scaling of the tree algorithm as $O(N_i \log N_j)$. The scaling gradually becomes worse when decreasing the problem size. In $N_{tot} \lesssim 10^4$, the execution time to calculate the mass, the position, and the size of pseudo j-particles (blue squares with dotted line) approaches a constant floor on each device. Furthermore, the floor value is not negligible compared with the elapsed time to calculate gravity and increases its contribution. Improving the scaling is also necessary to achieve a shorter time-to-solution for $N_{tot} \lesssim 10^4$.

Contributions from tree construction (magenta diamonds with dot-dashed line) and orbit integration (green triangles with triple-dot-dashed line) are comparable for most values of N_{tot} and negligibly small in any case. It should be noted that performance optimization of tree construction is also helpful to decrease the time-to-solution even though its execution time itself is negligible. As stated in Section 2.6, the interval between successive tree constructions is determined by the balance between execution time of tree traversal and construction. Therefore, performance enhancements of the function to update the tree structure can accelerate N-body simulation by decreasing the execution time for calculating gravity. This is a characteristic behavior of GOTHIC due to optimizations affecting multiple functions.

The measured elapsed time per step is 0.47 s (0.58 s), 0.39 s (0.38 s), and 0.14 s (0.21 s) for the M31 model (the NFW sphere) with $2^{23} = 8{,}388{,}608$ particles on M2090, K20X, and GTX TITAN X, respectively. On GTX TITAN X, we ran N-body simulation using $2^{24} = 16{,}777{,}216$ particles and they took 0.30 s and 0.44 s per step for the M31 model and the NFW sphere, respectively. Ogiya et al. (2013) reported that the elapsed time per step of their code was ~ 5 s on M2090 for the NFW sphere with 2^{24} particles. This indicates that the sophisticated algorithms and optimizations adopted in GOTHIC, and performance improvements of GPU achieve more than ten times acceleration of N-body simulations compared to Ogiya et al. (2013).

4. Discussion

The tree method has a better scaling compared to the direct method and is always faster in the high N-regime. However, in the low N-regime, the direct method becomes faster owing to its simplicity. Here, we briefly discuss the transition point at which to switch between the tree method and the direct method. Miki et al. (2013) reported that the execution times for calculating gravity by the direct method with $N = 2^{12} = 4{,}096$ and $N = 2^{13} = 8{,}192$ on M2090 are 9.7×10^{-4} s and 1.9×10^{-3} s, respectively. They are nearly the same as those with GOTHIC (see Fig. 8). Since 10^4 is a sufficiently large number of N-body particles to obtain the sustained performance on M2090, the growth of the elapsed time is proportional to N^2 for $N \gtrsim 10^4$. This implies that the tree method becomes faster than the direct method on GPU for $N \gtrsim 10^4$. Since direct N-body codes on GPU can maintain their $O(N^2)$ scaling down to $\sim 10^3$ through ij-parallelization (Miki et al., 2012), direct N-body codes becomes faster than the tree method in $N \lesssim 10^4$. Furthermore, Miki et al. (2013) adopted the shared time step instead of the block time step, so further speed up of their direct N-body code is possible. In summary, the execution time of GOTHIC is comparable with that of direct N-body codes if $N \sim 10^4$ and becomes shorter the larger the problem size.

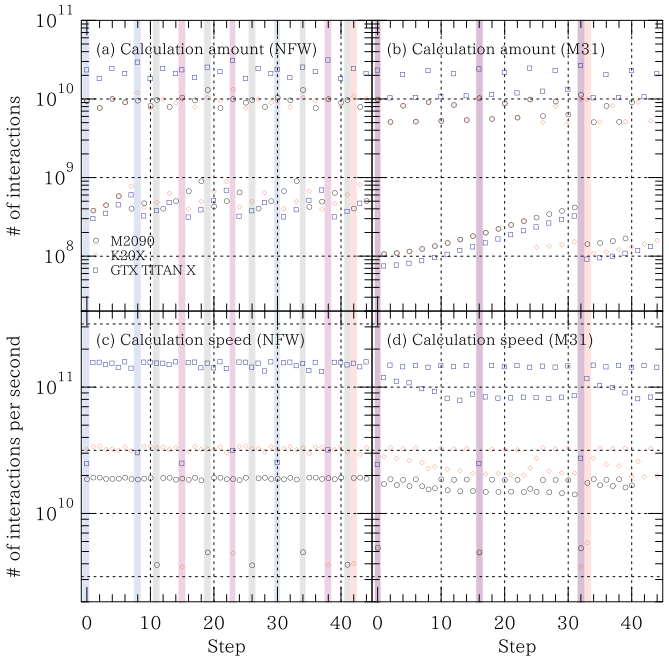

Fig. 9. Measured performance of GOTHIC. The upper and lower panels present the number of interactions and the calculation speed, respectively, as a function of the time step. Different symbols indicate different GPUs: black circles, red diamonds, and blue squares represent M2090, K20X, and GTX TITAN X, respectively. The left panels show results for the NFW model, and the right ones display results for the M31 model, both with $N = 2^{23} = 8{,}388{,}608$. Execution of the slow sequence is highlighted by vertical bands (colored according to GPU).

To estimate the achieved performance of GOTHIC, we have first counted the number of interactions computed in each time step. The counting of interaction pairs is done in a separate run to that of measurements of the elapsed time in order to remove the additional burden of the performance measurements. Fig. 9 shows the measured results as a function of the time step. The directly measured values are the calculated number of interactions in each time step and are shown in Fig. 9a and b. They have similar values on different generations of GPUs. The origin of the differences is differences of the configuration of the kernel function to calculate gravitational force (see Table 2). The gradual increase in the number of interactions with time step in the FSeq is the reason for the growth of the execution time for calculating gravity while using the same tree structure repeatedly. Since rebuilding the tree structure is auto-tuned as described in Section 2.6, the time steps at which the tree is rebuilt will differ depending on the problem or the utilized GPU. The number of interactions calculated per second (Fig. 9c and d) on each GPU is derived by combining independent measurements of the elapsed time. The measured results exhibit clear differences among the three GPUs, reflecting their theoretical peak performance.

The significant difference in each time step is attributable to the block time step. Step by step comparison between the number of interactions and the execution time in each time step reveals two things: (1) the lowest calculation speed is associated with the highest number of interaction pairs (as highlighted by vertical bands in Fig. 9) and (2) the minimum number of interaction pairs does not necessarily result in the highest calculation speed (this is more evident in the M31 model). The SSeq which corresponds to the maximum number of interaction pairs per step includes all i-particles in the lowest density regions, while the ISeq and FSeq, which correspond to the smaller number of interaction pairs per step, do not include i-particles in the lowest density regions. Including i-particles in the lowest density regions drastically increases the number of distance evaluations between a group of i-particles and pseudo j-particles. The remedy for this, introduced in Section 2.4, starts to work at later time steps, and the calculation speed decreases significantly. This is also the case with the steep increase of the elapsed time around $N_i \sim N_{\rm tot}$ observed in Fig. 7. The lowest number of interaction pairs does not lead to a sustained performance in the M31 model either. We find that the highest calculation rate is associated with an intermediate number of interaction pairs.

Conversion from the measured elapsed time to achieved performance requires an assumption about floating-point operation counts per interaction; however, such a conversion is not always rigorous especially in realistic scientific computations. Various values of the floating-point operation counts have been adopted in the literature for collisionless N-body simulations. Examples in studies using GPU(s) are: 20 by Nyland et al. (2007), 26 by Miki et al. (2012, 2013), and 23 by Bédorf et al. (2014), while 38 appears to be the typical value used in astrophysics (Kawai et al., 1999; Hamada and Iitaka, 2007; Nitadori and Makino, 2008; Hamada et al., 2009; Hamada and Nitadori, 2010; Tanikawa et al., 2013). The reason for the differences lies in the estimation of the execution cost of the inverse square root. In this study, we assume that the cost of executing the inverse square root corresponds to the ratio of the throughput of the reciprocal square root to that of addition or multiplication. This is found to be 8, 6, and 4 Flops (floating-point operations) on M2090, K20X, and GTX TITAN X, respectively. It should be noted that an alternative is adopting 4, 3, and 2 Flops on different generations of GPUs (Capuzzo-Dolcetta and Spera, 2013; Bédorf et al., 2014). This choice takes into account the fact that GPUs by NVIDIA support FMA operations and thus can execute 2 Flops per clock cycle. The remaining operations are three subtractions, three multiplications, and seven FMA operations (20 Flops in total), because GOTHIC calculates not only the gravitational force but also the gravitational potential (an FMA operation returns the potential). In summary, we assume that floating-point operation counts per interaction are 28, 26, and 24 Flops, respectively, on M2090, K20X, and GTX TITAN X.

Table 3 summarizes the measured number of interactions calculated per second and the corresponding performance in units of GFlop/s (Giga Floating-point operations per second) for the NFW sphere and the M31 model with $N = 2^{23} = 8{,}388{,}608$ on the three generations of GPUs. The averaged performance over time steps on M2090, K20X, and GTX TITAN X are around 320 GFlop/s, 360 GFlop/s, and 1750 GFlop/s, respectively. They correspond to 10–30% of the theoretical peak performance. The maximum performance on each GPU is around 40%, 20% and 55% of its theoretical peak performance on M2090, K20X, and GTX TITAN X, respectively. Finally, the minimum performance over several time steps drops to less than 10% of the theoretical peak performance except for the M31 model on M2090. This is the case with the highest number of interaction pairs as shown in Fig. 9; i.e., it is equivalent to the performance of the shared time step. This means that the benefit of adopting the block time step lies not only in avoiding unnecessary calculations to follow the time evolution of the system but also in increasing the average calculation speed per time step.

Watanabe and Nakasato (2014) proposed a hybrid tree algorithm to reduce the calculation cost of collisionless N-body simulations applying Particle-Particle Particle-Tree (PPPT) algorithm originally developed by Oshino et al. (2011) for collisional systems. They divided the gravitational force calculation into two steps, short-range and long-range, and reduce the relative frequency of long-range force calculation. Because neglecting small changes of the gravitational field in the distant region does not generate a sig-

Table 3
Achieved performance.

GPU	Model[a]	Number of interactions per second			Achieved performance[b] (GFlop/s)			TPP[c] (GFlop/s)
		Average	Maximum	Minimum	Average	Maximum	Minimum	
M2090	NFW	1.06×10^{10}	1.92×10^{10}	3.87×10^{9}	296	536	108	1332
	M31	1.20×10^{10}	1.86×10^{10}	4.90×10^{9}	336	521	137	
K20X	NFW	1.45×10^{10}	3.40×10^{10}	3.77×10^{9}	377	885	98	3935
	M31	1.34×10^{10}	3.30×10^{10}	3.81×10^{9}	349	859	99	
GTX TITAN X	NFW	6.77×10^{10}	1.59×10^{11}	2.49×10^{10}	1626	3827	598	6611
	M31	7.80×10^{10}	1.50×10^{11}	2.46×10^{10}	1871	3595	590	

[a] Model of initial particle distribution.
[b] One interaction is assumed to correspond to 28, 26 and 24 Flops on M2090, K20X and GTX TITAN X, respectively.
[c] Theoretical peak performance using single precision for each GPU.

nificant error in the force calculations, they succeeded in accelerating the computations without loss of accuracy. They reported a 20% acceleration of the *N*-body simulation for a Plummer sphere; however, the speed up rate probably depends on the distribution of *N*-body particles. In this study, the acceleration by the block time step compared to the shared time step in a Plummer sphere is around 50% for a given typical accuracy while that in the M31 model reaches 500%. This suggests that the hybrid tree algorithm has the potential to accelerate the calculation more than what was reported by Watanabe and Nakasato (2014). Also, combining the hybrid tree algorithm with GOTHIC is possible because the original PPPT algorithm was designed to couple with the individual time step scheme.

There is another unexplored avenue to further accelerate GOTHIC. The block time step introduces an order of magnitude variance of the number of *i*-particles N_i in each time step. As clearly shown in Fig. 7, the optimal value for the number of threads to share an *i*-particle, *S*, depends on N_i. In the current version of GOTHIC, we fix *S* throughout in the simulation to implement the code easily. However, dynamically adjusting the optimal value for *S* in each time step would accelerate the code especially in the low N_i-regime. This sort of auto-tuning is suitable to optimize codes whose performance depend strongly on the inputted problems, and might become a key issue to achieve a good strong scaling in future studies.

Operations for floating point numbers using half precision are supported on current GPUs and are twice as fast as those using single precision on the Pascal generation of GPUs designed for HPC (i.e., GP100 architecture). The number of mantissa bits for half precision is 10 in the IEEE 754-2008 standard. Tanikawa et al. (2013) showed that the approximate inverse square root function with 12 bits accuracy could provide sufficient accuracy for collisionless systems and implemented this in their software library "Phantom-GRAPE", a high-performance direct *N*-body library for CPU. This suggests that the approximate inverse square root function using half precision might also give sufficient accuracy for collisionless *N*-body simulations. Because the inverse square root is the heaviest function in *N*-body simulations, it would accelerate *N*-body simulations further. Even if the accuracy is not sufficient, the Newton–Raphson method can improve the accuracy at only a small cost. Furthermore, adopting arithmetic operations using half precision is promising in the tree method since the distance evaluation stage described in Section 2.3 does not require a precise value of the distance in single precision. Current NVIDIA GPUs support the approximate inverse square root function `rsqrtf()` with at least 21 bits accuracy (NVIDIA, 2015) for variables at single precision and they were found to successfully accelerate collisionless *N*-body simulations (Nyland et al., 2007; Miki et al., 2012, 2013). Exploiting the half precision version of `rsqrtf()`, if it exists, would also increase the performance of realistic scientific computations.

5. Summary

Adopting the tree method is a common way to accelerate collisionless *N*-body simulations in astrophysics, even on GPU. Many earlier studies presented tree codes efficiently running on GPU(s), yet none had coupled their code with the block time step (Nakasato, 2012; Ogiya et al., 2013; Bédorf et al., 2012, 2014; Watanabe and Nakasato, 2014). Since the block time step can also accelerate *N*-body simulations significantly, we have developed a gravitational octree code (GOTHIC), which is accelerated by the block time step. The code adopts the breadth-first search, and runs entirely on GPU, just like `Bonsai` by Bédorf et al. (2012, 2014). The algorithm in the tree traversal is an improved version of the algorithm proposed by Ogiya et al. (2013), which used a depth-first search. GOTHIC also does adaptive optimizations, i.e., auto-tuning, by monitoring the execution time of each function. The optimizations reduce the time-to-solution by balancing the execution time of multiple functions, and using optional *ij*-parallelization to maintain high performance in the low N_i-regime.

The performance of the code is measured on NVIDIA Tesla M2090, K20X, and GeForce GTX TITAN X, which are representative GPUs of the Fermi, Kepler, and Maxwell generation of GPUs, using realistic particle distributions found in astrophysics. The results show that the code with the fiducial configuration (the block time step with the acceleration MAC) achieves around a 3–5 times acceleration compared to the shared time step, and is faster than the public code `Bonsai`. The elapsed time of the code scales roughly as *N* for $N \gtrsim 10^5$; the dependence is slightly weaker than the expected scaling for the tree method, $O(N \log N)$. The averaged performance of the code corresponds to 10–30% of the theoretical peak performance of each GPU. The measured elapsed time per step of GOTHIC is 0.30 s and 0.44 s on GTX TITAN X when the particle distribution represents the Andromeda galaxy and the NFW sphere, respectively, with $2^{24} = 16{,}777{,}216$ particles. The achieved time-to-solution is more than ten times smaller than that achieved in Ogiya et al. (2013). There are still some possibilities for further optimizations that can be explored, for example: (1) adopting a more sophisticated algorithm such as the hybrid tree algorithm proposed by Watanabe and Nakasato (2014), (2) performing deeper optimizations focusing on specific generation of GPUs, (3) auto-tuning of the optimal number of threads *S* in *ij*-parallelization, and (4) utilizing new functions provided by hardware vendors or compilers such as operations in the half precision.

Acknowledgments

YM is particularly grateful to Go Ogiya for fruitful discussions and providing detailed information about his code implementation and performance measurements. YM also appreciates suggestions from Kohji Yoshikawa on code optimizations. YM has benefited from feedback by Takanobu Kirihara on using a beta version

```
PHint encodePeano3D(int Nlev, PHint px, PHint py, PHint pz){
  PHint key = 0;

  for(int jj = Nlev - 1; jj >= 0; jj--){
    /* get xi, yi, and zi */
    PHint xi = (px >> jj) & 1;
    PHint yi = (py >> jj) & 1;
    PHint zi = (pz >> jj) & 1;

    /* turn px, py, and pz */
    px ^= -( xi & ((!yi) |   zi));
    py ^= -((xi & (  yi  |   zi)) | (yi & (!zi)));
    pz ^= -((xi &  (!yi) & (!zi)) | (yi & (!zi)));

    /* append 3bits to the key */
    key |= ((xi << 2) | ((xi ^ yi) << 1) | ((xi ^ zi) ^ yi)) << (3 * jj);

    /* rotate uncyclic (x->z->y->x) */
    if( zi ){      PHint pt = px;      px = py;      py = pz;      pz = pt;    }
    else{
      /* exchange x and z */
      if( !yi ){   PHint pt = px;      px = pz;                  pz = pt;    }
    }
  }
  return (key);
}
```

Listing 1. Implementation of Peano–Hilbert key encoder.

of GOTHIC which improved the performance in realistic simulations. We thank Daisuke Takahashi for providing information on auto-tuning. We would like to express our gratitude to Alexander Y. Wagner for careful reading of the manuscript and comments that improve the paper. Numerical simulations were performed on HA-PACS at the Center for Computational Sciences, University of Tsukuba. The present study was supported by the Japan Science and Technology Agency's (JST) CREST program entitled "Research and Development of Unified Environment on Accelerated Computing and Interconnection for Post-Petascale Era." This research was also supported in part by the Grant-in-Aid for Scientific Research (B) by JSPS (15H03638).

Appendix A. Space-filling curves

Listings 1 and 2 are implementations of the Peano–Hilbert key encoder and decoder, respectively, written in C. The algorithm is an extension to 3D space of the implementation in 2D space by Lam and Shapiro (1994). The generation of Peano–Hilbert keys boils down to the rotation and/or inversion of the fundamental block. Since the rotation and inversion in the 3D space are non-commutative operations, level-by-level encoding/decoding is necessary. The number of logical operations is minimized using the Karnaugh map. The data type `PHint` is `unsigned int` or `unsigned long int` depending on whether the bit length of the key is less than or equal to 30 (the maximum size that fits in a 32-bit integer), respectively.

For comparison, Listing 3 shows how the Morton key generator works up to 63 bit keys. Bédorf et al. (2012) provided Morton key generator in 30 bits based on Raman and Wise (2008). Listing 3 is simply an extension of this to 63 bits. It is much simpler than the Peano–Hilbert key generator; however, it does not have a one-stroke sketch nature.

Appendix B. Comparison of enclosing balls

We have implemented 5 kinds of enclosing ball generators: (1) the smallest enclosing ball (SEB) given by the algorithm proposed by Fischer et al. (2003), (2) the efficient bounding sphere (EBS) proposed by Ritter (1990), (3) the sphere centered on the geometric center of the enclosing rectangular cuboid (GEO), (4) the sphere centered on the center-of-mass of particles (COM), and (5) the smaller of the spheres generated by GEO and COM (CMP). The smaller radius of the enclosing ball mitigates the increase of the number of interactions especially in the low density regions and reduce the elapsed time. From this point of view, the best choice is the SEB, which has the minimum radius. On the other hand, the precise determination of the SEB is a time-consuming process. Therefore, the optimal choice for the generator should be determined by comparing the elapsed times of the code with the various generators. In this section, we summarize the performance of the enclosing ball generators.

First, we compared the radii of each enclosing ball, $r_{\rm ball}$. Fig. B.10 shows amount of radius over-estimation, $r_{\rm ball}/r_{\rm SEB}$, as a function of the radius of the smallest enclosing ball, $r_{\rm SEB}$. After SEB, the EBS method results in the smallest radii; its over-estimation is 5% in most cases and $\sim$ 10% in the worst case as originally claimed by Ritter (1990). The GEO gives somewhat little bigger radii; however, it is smaller than $1.15r_{\rm SEB}$ in most cases. On the other hand, $r_{\rm ball}$ in the COM is much bigger, and it exceeds $1.4r_{\rm SEB}$ in the low density regions (i.e., the region with large $r_{\rm SEB}$); hence, the number of operations executed in the gravity calculations become much greater than other enclosing ball models. The CMP resembles the GEO because the COM predicts larger radii than the GEO in most cases.

Table B.4 lists the costs to generate each enclosing ball on different GPUs. The cost is measured by calling the `clock64()` function within the `_global_` function in the CUDA code and translated into the elapsed time by dividing by the number of concurrent warps and the clock cycle frequency. The elapsed time to generate enclosing balls is always negligibly small compared to that to calculate gravity. The dependence of the elapsed time on the particle distribution is much weaker compared to that of the gravity calculation.

```
void decodePeano3D(int Nlev, PHint key, PHint *rx, PHint *ry, PHint *rz){
  PHint px = 0;
  PHint py = 0;
  PHint pz = 0;

  for(int jj = 0; jj < Nlev; jj++){
    /* get xi, yi, and zi */
    PHint xi = (key >> (3 * jj + 2)) & 1;
    PHint yi = (key >> (3 * jj + 1)) & 1;
    PHint zi = (key >> (3 * jj    )) & 1;

    /* rotate cyclic (x->y->z->x) */
    if( yi ^ zi ){
        PHint pt = px; px = pz; pz = py; py = pt; }
    else{
      /* exchange x and z */
      if( (!xi & !yi & !zi) || (xi & yi & zi) ){
        PHint pt = px; px = pz;           pz = pt; }
    }

    /* turn px , py , and pz */
    PHint mask = ((PHint)1 << jj) - 1;
    px ^= mask & (-( xi & (  yi  |   zi ))                        );
    py ^= mask & (-((xi & ((!yi) | (!zi))) | ((!xi) & yi & zi)));
    pz ^= mask & (-((xi &  (!yi) & (!zi) ) | (        yi & zi)));

    /* append 1 bit to the position */
    px |= ( xi       << jj);
    py |= ((xi ^ yi) << jj);
    pz |= ((yi ^ zi) << jj);
  }
  *rx = px;  *ry = py;  *rz = pz;
}
```

Listing 2. Implementation of Peano–Hilbert key decoder.

```
PHint dilate3D(PHint val){
  val = (val * 0x100000001) & 0x7fff00000000ffff;/*    execute if >= 33 bits */
  val = (val * 0x000010001) & 0x00ff0000ff0000ff;/* 0xff0000ff if <= 30 bits */
  val = (val * 0x000000101) & 0x700f00f00f00f00f;/* 0x0f00f00f if <= 30 bits */
  val = (val * 0x000000011) & 0x30c30c30c30c30c3;/* 0xc30c30c3 if <= 30 bits */
  val = (val * 0x000000005) & 0x1249249249249249;/* 0x49249249 if <= 30 bits */
  return (val);
}
PHint genMorton3D(const PHint ix, const PHint iy, const PHint iz){
  return ((dilate3D(ix) << 2) | (dilate3D(iy) << 1) | (dilate3D(iz)));
}
```

Listing 3. Implementation of Morton key generator.

Table B.4
Computing cost to generate various enclosing balls.

GPU[a]	Model[b]	SEB[c]	EBS[d]	GEO[e]	COM[f]	CMP[g]
M2090	NFW	2.13×10^{-2} s	1.07×10^{-2} s	5.34×10^{-3} s	3.27×10^{-3} s	7.86×10^{-3} s
M2090	M31	2.13×10^{-2} s	1.06×10^{-2} s	5.33×10^{-3} s	3.27×10^{-3} s	7.86×10^{-3} s
K20X	NFW	1.02×10^{-2} s	3.05×10^{-3} s	1.27×10^{-3} s	9.08×10^{-4} s	2.05×10^{-3} s
K20X	M31	1.02×10^{-2} s	3.01×10^{-3} s	1.27×10^{-3} s	9.08×10^{-4} s	2.06×10^{-3} s
TITAN X	NFW	1.06×10^{-2} s	2.04×10^{-3} s	5.09×10^{-6} s	7.45×10^{-4} s	8.43×10^{-4} s
TITAN X	M31	1.06×10^{-2} s	2.00×10^{-3} s	5.09×10^{-6} s	7.49×10^{-4} s	8.69×10^{-4} s

[a] Name of GPU.
[b] Particle distribution models.
[c] Cost to generate the smallest enclosing ball based on Fischer et al. (2003).
[d] Cost to generate the efficient bounding sphere based on Ritter (1990).
[e] Cost to generate the sphere centered on the geometric center of the enclosing rectangular cuboid.
[f] Cost to generate the sphere centered on the center-of-mass of particles.
[g] Cost to generate the smaller sphere of GEO and COM.

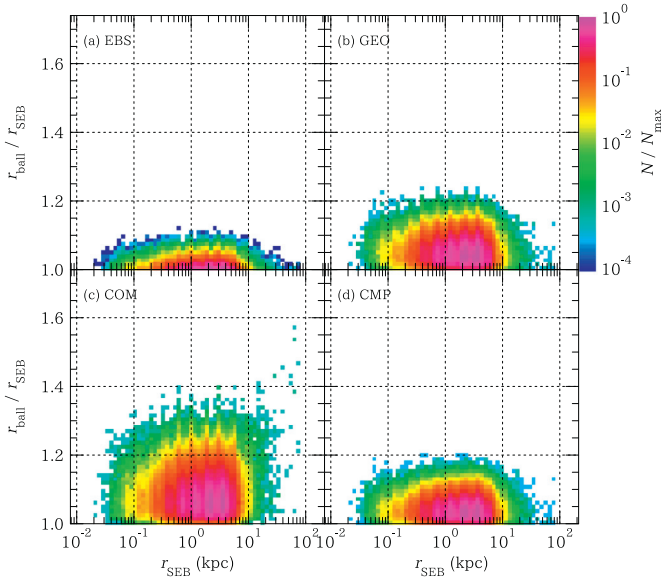

Fig. B.10. Radii of enclosing balls. The horizontal and the vertical axes are the radii of the smallest enclosing ball $r_{\rm SEB}$ and that of an enclosing ball $r_{\rm ball}$ normalized by $r_{\rm SEB}$, respectively. The color map on each panel displays the normalized frequency for different definitions of the pseudo i-particles: (a) the efficient bounding sphere (Ritter, 1990), (b) the sphere centered on the geometric center of the enclosing rectangular cuboid, (c) the sphere centered on the center-of-mass of particles, and (d) the smaller sphere of (b) and (c). The particle distribution is that representing M31 by $2^{23} = 8{,}388{,}608$ particles, and the total number of enclosing balls generated on K20X is 262,144.

Appendix C. Modeling the interval of tree rebuild

In the power-law growth model, the required time to calculate gravity at the ith step is assumed to grow as

$$t_{\rm walk}^{(i)} = r^{i-1} t_1, \tag{C.1}$$

where t_1 and r are the scale factor and the common ratio, respectively. The total elapsed time after n steps is given by

$$t_{\rm tot} = t_{\rm make} + \frac{r^n - 1}{r - 1} t_1. \tag{C.2}$$

The first and the second derivatives of $t_{\rm mean} = t_{\rm tot}/n$ with respect to n are calculated as

$$\frac{d}{dn}\frac{t_{\rm tot}}{n} = -\frac{t_{\rm make}}{n^2} + \frac{(n\ln r - 1)r^n + 1}{n^2(r-1)} t_1, \tag{C.3}$$

$$\frac{d^2}{dn^2}\frac{t_{\rm tot}}{n} = \frac{2t_{\rm make}}{n^3} + \frac{\{1 + (n\ln r - 1)^2\}r^n - 2}{n^3(r-1)} t_1. \tag{C.4}$$

Therefore, the desired condition for rebuilding the tree becomes

$$(n\ln r - 1)r^n = (r-1)\frac{t_{\rm make}}{t_1} - 1, \tag{C.5}$$

if the right hand side of (C.4) is positive. Substituting (C.5) into (C.4) yields the equation

$$\frac{d^2}{dn^2}\frac{t_{\rm tot}}{n} = \frac{1}{n}\frac{(\ln r)^2 r^n}{r-1} t_1, \tag{C.6}$$

which implies that $r > 1$ is the necessary condition to minimize $t_{\rm mean}$.

In the parabolic growth model, we assume

$$t_{\rm walk}^{(i)} = t_1 + (i-1)b + (i-1)^2 a, \tag{C.7}$$

where t_1, a, and b are fitting parameters determined by the least squared method. The total elapsed time after n steps is written as

$$t_{\rm tot} = t_{\rm make} + nt_1 + \frac{n(n-1)}{2} b + \frac{n(n-1)(2n-1)}{6} a. \tag{C.8}$$

The first and the second derivatives of $t_{\rm mean} = t_{\rm tot}/n$ with respect to n are calculated as

$$\frac{d}{dn}\frac{t_{\rm tot}}{n} = -\frac{t_{\rm make}}{n^2} + \frac{b}{2} + \frac{4n-3}{6} a, \tag{C.9}$$

$$\frac{d^2}{dn^2}\frac{t_{\rm tot}}{n} = \frac{2t_{\rm make}}{n^3} + \frac{2a}{3}. \tag{C.10}$$

Equating (C.9) to zero yields the optimal choice as

$$n^2 = \left\{\frac{b}{2} + \frac{4n-3}{6} a\right\}^{-1} t_{\rm make}. \tag{C.11}$$

Putting (C.11) into (C.10) gives the expression of the second derivative at the extremum:

$$\frac{d^2}{dn^2}\frac{t_{\rm tot}}{n} = \frac{b}{n} + \frac{2n-1}{n} a = \frac{b + (2n-1)a}{n}. \tag{C.12}$$

Therefore,

$$(2n-1)a + b \geq 0 \tag{C.13}$$

is the necessary condition to get the shortest time-to-solution.

References

Aarseth, S.J., 1963. Dynamical evolution of clusters of galaxies, I. Mon. Not. R. Astron. Soc. 126, 223.

Ashari, A., Sedaghati, N., Eisenlohr, J., Sadayappan, P., 2014. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs. In: Bode, A., Gerndt, M., Stenström, P., Rauchwerger, L., Miller, B.P., Schulz, M. (Eds.), 2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany, June 10-13, 2014. ACM, pp. 273–282.

Barnes, J., Hut, P., 1986. A hierarchical O(N log N) force-calculation algorithm. Nature 324, 446–449.

Bédorf, J., Gaburov, E., Fujii, M. S., Nitadori, K., Ishiyama, T., Portegies Zwart, S., 2014. 24.77 Pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 GPUs. ArXiv e-prints.

Bédorf, J., Gaburov, E., Portegies Zwart, S., 2012. A sparse octree gravitational N-body code that runs entirely on the GPU processor. J. Comput. Phys. 231, 2825–2839.

Bell, N., Garland, M., 2008. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report. NVIDIA Corporation. NVR-2008-004.

Blelloch, G.E., 1990. Prefix Sums and Their Applications. Technical Report. School of Computer Science, Carnegie Mellon University. CMU-CS-90-190.

Capuzzo-Dolcetta, R., Spera, M., 2013. A performance comparison of different graphics processing units running direct N-body simulations. Comput. Phys. Commun. 184, 2528–2539.

Fardal, M.A., Guhathakurta, P., Babul, A., McConnachie, A.W., 2007. Investigating the Andromeda stream - III. A young shell system in M31. Mon. Not. R. Astron. Soc. 380, 15–32.

Fischer, K., Gärtner, B., Kutz, M., 2003. Fast smallest-enclosing-ball computation in high dimensions. In: Battista, G.D., Zwick, U. (Eds.), Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings. Springer, pp. 630–641.

Frigo, M., Johnson, S.G., 2005. The design and implementation of FFTW3. Proceedings of the IEEE 93 (2). Special issue on "Program Generation, Optimization, and Adaptation".

Fukushige, T., Ito, T., Makino, J., Ebisuzaki, T., Sugimoto, D., Umemura, M., 1991. GRAPE-1A: special-purpose computer for N-body simulation with a tree code. Publ. Astron. Soc. Jpn. 43, 841–858.

Fukushige, T., Makino, J., Kawai, A., 2005. GRAPE-6A: a single-card GRAPE-6 for parallel PC-GRAPE cluster systems. Publ. Astron. Soc. Jpn. 57, 1009–1021.

Geehan, J.J., Fardal, M.A., Babul, A., Guhathakurta, P., 2006. Investigating the andromeda stream - I. Simple analytic bulge-disc-halo model for M31. Mon. Not. R. Astron. Soc. 366, 996–1011.

Hamada, T., Iitaka, T., 2007. The chamomile scheme: an optimized algorithm for N-body simulations on programmable graphics processing units. ArXiv Astrophysics e-prints.

Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K., Taiji, M., 2009. 42 TFlops hierarchical *N*-body simulations on GPUs with applications in both astrophysics and turbulence. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, New York, NY, USA, pp. 62:1–62:12.

Hamada, T., Nitadori, K., 2010. 190 TFlops astrophysical *N*-body simulation on a cluster of GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, Washington, DC, USA, pp. 1–9.

Hernquist, L., 1990. An analytical model for spherical galaxies and bulges. Astrophys. J. 356, 359–364.

Hockney, R.W., Eastwood, J.W., 1988. Computer simulation using particles.

Ishiyama, T., Fukushige, T., Makino, J., 2009. GreeM: massively parallel TreePM code for large cosmological N -body simulations. Publ. Astron. Soc. Jpn. 61, 1319–1330.
Ishiyama, T., Nitadori, K., Makino, J., 2012. 4.45 pflops astrophysical *N*-body simulation on K computer: the gravitational trillion-body problem. In: Hollingsworth, J.K. (Ed.), SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012. IEEE/ACM, p. 5.
Ito, T., Ebisuzaki, T., Makino, J., Sugimoto, D., 1991. A special-purpose computer for gravitational many-body systems: GRAPE-2. Publ. Astron. Soc. Jpn. 43, 547–555.
Ito, T., Makino, J., Ebisuzaki, T., Sugimoto, D., 1990. A special-purpose N-body machine GRAPE-1. Comput. Phys. Commun. 60, 187–194.
Ito, T., Makino, J., Fukushige, T., Ebisuzaki, T., Okumura, S.K., Sugimoto, D., 1993. A special-purpose computer for N-body simulations: GRAPE-2A. Publ. Astron. Soc. Jpn. 45, 339–347.
Kawai, A., Fukushige, T., Makino, J., 1999. $7.0/Mflops astrophysical N-body simulation with treecode on GRAPE-5. In: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (CDROM). ACM, New York, NY, USA.
Kawai, A., Fukushige, T., Makino, J., Taiji, M., 2000. GRAPE-5: a special-purpose computer for N-body simulations. Publ. Astron. Soc. Jpn. 52, 659–676.
King, I.R., 1966. The structure of star clusters. III. Some simple dynamical models. Astron. J. 71, 64.
Lai, J., Seznec, A., 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013. IEEE Computer Society, pp. 4:1–4:10.
Lam, W.M., Shapiro, J.M., 1994. A class of fast algorithms for the Peano–Hilbert space-filling curve. In: Proceedings 1994 International Conference on Image Processing, Austin, Texas, USA, November 13-16, 1994. IEEE, pp. 638–641.
Liu, W., Vinter, B., 2015. CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. In: Bhuyan, L.N., Chong, F., Sarkar, V. (Eds.), Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015. ACM, pp. 339–350.
Maggioni, M., Berger-Wolf, T., 2016. Optimization techniques for sparse matrixvector multiplication on GPUs. J. Parallel Distrib. Comput. 9394, 66–86.
Makino, J., Fukushige, T., Koga, M., Namura, K., 2003. GRAPE-6: Massively-Parallel Special-Purpose Computer for Astrophysical Particle Simulations. Publ. Astron. Soc. Jpn. 55, 1163–1187.
Makino, J., Taiji, M., Ebisuzaki, T., Sugimoto, D., 1997. GRAPE-4: A Massively Parallel Special-Purpose Computer for Collisional N-Body Simulations. Astrophys. J. 480, 432.
McMillan, S.L.W., 1986. The vectorization of small-N integrators. In: Hut, P., McMillan, S.L.W. (Eds.), The Use of Supercomputers in Stellar Dynamics. In: Lecture Notes in Physics, Berlin Springer Verlag, 267, p. 156.
Michie, R.W., 1963. On the distribution of high energy stars in spherical stellar systems. Mon. Not. R. Astron. Soc. 125, 127.
Michie, R.W., Bodenheimer, P.H., 1963. The dynamics of spherical stellar systems, II. Mon. Not. R. Astron. Soc. 126, 269.
Miki, Y., Takahashi, D., Mori, M., 2012. A fast implementation and performance analysis of collisionless N-body code based on GPGPU. Procedia Comput. Sci. 9, 96–105. Proceedings of the International Conference on Computational Science, ICCS 2012.
Miki, Y., Takahashi, D., Mori, M., 2013. Highly scalable implementation of an N-body code on a GPU cluster. Comput. Phys. Commun. 184, 2159–2168.
Miki, Y., Umemura, M., in preparation. MAGI: MAny-component galactic initial-conditions generator.
Nakasato, N., 2012. Implementation of a parallel tree method on a GPU. J. Comput. Sci. 3 (3), 132–141. Scientific Computation Methods and Applications.
Navarro, J.F., Frenk, C.S., White, S.D.M., 1995. Simulations of X-ray clusters. Mon. Not. R. Astron. Soc. 275, 720–740.
Navarro, J.F., Frenk, C.S., White, S.D.M., 1996. The structure of cold dark matter halos. Astrophys. J. 462, 563.
Nelson, A.F., Wetzstein, M., Naab, T., 2009. Vine–a numerical code for simulating astrophysical systems using particles. II. Implementation and performance characteristics. Astrophys. J., Supp. 184, 326–360.
Nitadori, K., Makino, J., 2008. Sixth- and eighth-order Hermite integrator for N-body simulations. New Astron. 13, 498–507.
Nitadori, K., Makino, J., Abe, G., 2006. High-performance small-scale simulation of star clusters evolution on Cray XD1. ArXiv Astrophysics e-prints.
NVIDIA, 2007. NVIDIA CUDA Conpute Unified Device Architecture Programming Guide Version 1.0.
NVIDIA, 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi.
NVIDIA, 2012. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110.
NVIDIA, 2015. CUDA C Programming Guide Version 7.5.
Nyland, L., Harris, M., Prins, J., 2007. Fast *N*-Body Simulation with CUDA.
Ogiya, G., Mori, M., Miki, Y., Boku, T., Nakasato, N., 2013. Studying the core-cusp problem in cold dark matter halos using N-body simulations on GPU clusters. J. Phys. Conf. Series 454 (1), 012014.
Okumura, S.K., Makino, J., Ebisuzaki, T., Fukushige, T., Ito, T., Sugimoto, D., Hashimoto, E., Tomida, K., Miyakawa, N., 1993. Highly parallelized special-purpose computer, GRAPE-3. Publ. Astron. Soc. Jpn. 45, 329–338.
Oshino, S., Funato, Y., Makino, J., 2011. Particle-particle particle-tree: a direct-tree hybrid scheme for collisional N-body simulations. Publ. Astron. Soc. Jpn. 63, 881–892.
Plummer, H.C., 1911. On the problem of distribution in globular star clusters. Mon. Not. R. Astron. Soc. 71, 460–470.
Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., 2007. Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3 edition Cambridge University Press.
Raman, R., Wise, D.S., 2008. Converting to and from dilated integers. IEEE Trans. Computers 57 (4), 567–573.
Reguly, I., Giles, M., 2012. Efficient sparse matrix-vector multiplication on cache-based gpus. In: Innovative Parallel Computing (InPar), 2012, pp. 1–12.
Ritter, J., 1990. Graphics Gems. Academic Press Professional, Inc., San Diego, CA, USA, pp. 301–303. Chapter An Efficient Bounding Sphere.
Sagan, H., 2012. Space-Filling Curves. Springer Science & Business Media.
Salmon, J.K., Warren, M.S., 1994. Skeletons from the treecode closet. J. Comput. Phys. 111, 136–155.
Springel, V., 2005. The cosmological simulation code GADGET-2. Mon. Not. R. Astron. Soc. 364, 1105–1134.
Sugimoto, D., Chikada, Y., Makino, J., Ito, T., Ebisuzaki, T., Umemura, M., 1990. A special-purpose computer for gravitational many-body problems. Nature 345, 33–35.
Tanikawa, A., Yoshikawa, K., Nitadori, K., Okamoto, T., 2013. Phantom-GRAPE: numerical software library to accelerate collisionless N-body simulation with SIMD instruction set on x86 architecture. New Astron. 19, 74–88.
Umemura, M., Susa, H., Hasegawa, K., Suwa, T., Semelin, B., 2012. Formation and radiative feedback of first objects and first galaxies. Prog. Theor. Exp. Phys. 2012 (1), 01A306.
Warren, M.S., Salmon, J.K., 1993. A parallel hashed oct-tree N-body algorithm. In: Proceedings of the 1993 ACM/IEEE conference on Supercomputing. ACM, pp. 12–21.
Watanabe, T., Nakasato, N., 2014. GPU accelerated hybrid tree algorithm for collision-less N-body simulations. ArXiv e-prints.
Whaley, R.C., Petitet, A., Dongarra, J., 2001. Automated empirical optimizations of software and the ATLAS project. Parallel Comput. 27 (1-2), 3–35.
Xiao, S., Feng, W., 2010. Inter-block GPU communication via fast barrier synchronization. In: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp. 1–12.