ELSEVIER
Available online at www.sciencedirect.com
SciVerse ScienceDirect
Procedia Computer Science 9 (2012) 96 – 105
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

A Fast Implementation and Performance Analysis of Collisionless *N*-body Code Based on GPGPU

Yohei Miki[a,b,*], Daisuke Takahashi[c], Masao Mori[d]

[a]*Graduate School of Pure and Applied Sciences, University of Tsukuba*
[b]*Graduate School of Systems and Information Engineering, University of Tsukuba*
[c]*Faculty of Engineering, Information and Systems, University of Tsukuba*
[d]*Faculty of Pure and Applied Sciences, University of Tsukuba*

Abstract

We have implemented a fast collisionless *N*-body code which runs on GPU, the peak performance of the code reaches 767 GFLOPS (corresponds to 74 % of theoretical peak performance for our measurement environment) under an assumption of computational cost is 26 floating-point operations per interaction. Our implementation is 1.7 times faster than CUDA SDK in maximum case (for low *N* region) due to our proposal algorithm of force accumulation without synchronization. Detailed performance analysis clarifies that the performance metrics of collisionless *N*-body simulations on GPU are only two quantities: first one is the number of running streaming multiprocessors and another is the clock cycle ratio of latency to access global memory and operations to calculate gravitational interaction.

Keywords: *N*-body simulations, GPGPU, optimization

1. Introduction

In astrophysics, collisionless *N*-body simulation is one of the most powerful tool to investigate structure formation of large scale structure, formation and evolution history of stellar systems such as galaxies. The fundamental equation of *N*-body simulation is Newton's equation of motion expressed as

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

where G is the gravitational constant, m_i, $\boldsymbol{x}_i$ and $\boldsymbol{a}_i$ are mass, position, and acceleration of i-th particle out of N particles, respectively. The gravitational softening parameter ϵ, introduced to avoid divergence due to division by zero, eliminates self-interaction when calculating gravitational force. The amount of computation is proportional to the number of i-particles, N_i, and the number of j-particles, N_j. We use the word i-particles, and j-particles to

*Corresponding author
Email addresses: `ymiki@ccs.tsukuba.ac.jp` (Yohei Miki), `daisuke@cs.tsukuba.ac.jp` (Daisuke Takahashi), `mmori@ccs.tsukuba.ac.jp` (Masao Mori)

doi:10.1016/j.procs.2012.04.011

denote particles feel gravitational force, and particles cause gravitational force, respectively. Since a large number of N-body particles is necessary to investigate astrophysical phenomena in detail, many earlier studies have tackled with achieving fast implementation of N-body code. Fast algorithms, such as the particle-mesh method and the tree method, have been proposed to reduce the amount of computation [1, 2]. The computational complexity for the tree method is $O(N \log N)$, since the multipole expansion technique reduce the contribution from N_j. In contrast, accepting individual time step can also reduce the amount of computation by reducing N_i. Of course, there is another approach to reduce computational time; that is usage of accelerator. The most famous and one of the most successive accelerator for gravitational many-body systems is GRAPE ("Gravity PipE") series [3, 4]. The GRAPE system can perform fast calculation of gravitational interaction due to their design of pipelined and massively parallelized architecture. That is to say, accelerated calculation using GRAPE is due to massive parallelization of force calculation.

In recent days, GPU (Graphics Processing Units) becomes one of the most attractive accelerator due to development of GPGPU (General Purpose computing on GPU). Collisionless N-body simulation is one of the most successive example. Many earlier studies reported that high performance can be achieved by massive parallelization about i-particles [5, 6, 7, 8]. Implementation of CUDA SDK, based on [6], can achieve high performance of 716 GFLOPS (Giga FLoating-point Operations Per Second). Hamada et al. (2009) reported their implementation is slightly (about 6 %) faster than CUDA SDK [7]. However, there was no discussion about the origin of their difference. Their implementation can achieve high performance in high N region, however, achieving high performance in low N region is also essential to combine the tree-method or the individual time steps since the benefits of such algorithms comes from reduced N. Therefore, keeping high performance in low N region is essential, to achieve high performance in case of collaborating with such fast algorithms. For such purpose, CUDA SDK supports two-dimensional parallelization, combination of parallelization about i-particles and j-particles, of calculating gravitational force [6]. In fact, the implementation of CUDA SDK succeeded to achieve high performance in low N-region. However, there is a possibility to develop performance due to their poor implementation of force accumulation process. Two-dimensional parallelization also provide force accumulation process, since multiple threads calculate acceleration of a common i-particle, thus results of calculation must be accumulated by all corresponding threads. In such process, synchronization or exclusive control is necessary to account whole threads' results appropriately.

In this work, we propose a new technique to accumulate acceleration without synchronization in section 2. In addition, we have optimized our N-body code as introduced in section 3 and report measured performance of our implementation in section 4. Furthermore, also at the view point of computational science, collisionless N-body calculation is noteworthy due to its compute intensive aspect: calculations of $O(N^2)$ against for data transfer of $O(N)$. A detailed analysis of such characteristic calculation can contribute to optimize other GPGPU applications of compute intensive problems. Therefore, we provide a detailed performance analysis based on clock cycles in section 5.

2. Algorithm: Force Accumulation without Synchronization

In general, synchronization and exclusive control prevent obtaining high performance in parallel computing. For GPGPU, the cost of synchronization and exclusive control also high due to its characteristics as a many-core architecture. The accumulation process implemented in CUDA SDK is as follows: 1) whole threads store their own results to shared memory, 2) whole threads are synchronized by `__syncthreads()`, 3) a representative thread adds loaded data from shared memory to its own result. That is to say, CUDA SDK uses synchronization and exclusive control, both of them can significantly decrease the performance for small N, which is the parameter region they tried to improve performance.

Here, we propose a new technique to accumulate gravitational force without synchronization or usage of atomic operations. Our proposal makes use of the following two features: 1) the target array has 4 components (3 components for gravitational acceleration, and another one for gravitational potential), 2) 32 threads in a warp are synchronized implicitly since the threads in a warp perform operations at the same time. Therefore, accumulating gravitational acceleration is possible by shifting component to be accumulated, if a warp contains all relating threads.

Figure 1 and Code 1 show the above method to accumulate gravitational acceleration for a case of 4 threads calculate an i-particle, the most straightforward case. Here, we introduce our method in detail for the case. The 4 threads which labeled by an index `jj = 0, 1, 2, 3` have individual array "`src[4]`" to store their own results of calculations, and these results must be unified to shared array "`dst[4]`". At the first step, the threads store the data from `src[jj]` to `dst[jj]` (the first step of Figure 1, the third line of Code 1). On the next step, exclusive OR (`XOR`)

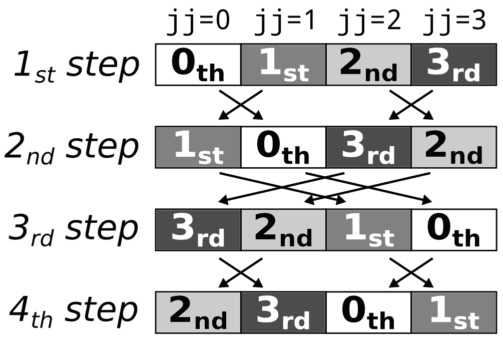

Figure 1: Basic image of accumulating gravitational acceleration without explicit synchronization

between `jj` and unity is performed, and the data stored in `src[jj]` is added to `dst[jj]` (the second step of Figure 1, the fourth line of Code 1). The operation `XOR` with unity is taken to flip the lowest bit of the index `jj` with the smallest cost to shift the index of the array to be updated (execution of logical operation for 32-bit integer needs only 1 cycle for GPUs of compute capability 2.0). In the forthcoming step, `src[jj]` is added to `dst[jj]` after performing `XOR` between `jj` and two. At the end of the step, each thread updates the one remained component, evaluated by taking `XOR` between `jj` and unity.

```
1 float src[4];
2 __shared__ float dst[4];
3          dst[jj] = src[jj];
4 jj ^= 1; dst[jj] += src[jj];
5 jj ^= 2; dst[jj] += src[jj];
6 jj ^= 1; dst[jj] += src[jj];
```

Code 1: Source code for accumulating gravitational acceleration without explicit synchronization (4 threads version)

We have implemented the above algorithm of four cases (2, 4, 8, 16 threads share an *i*-particle) on our collisionless *N*-body code.

3. Optimization of the Innermost Loop

Our implementation of the kernel function to calculate interactions is similar to CUDA SDK [6], except for some additional optimizations.

The most influential difference is calculation process of $\mathrm{r}_{ji}{}^2 + \epsilon^2$ (Code 2). In both implementation, a `float3` type variable `rji`, a `float` type variable `eps2`, and a `float` type variable `r2` store the displacement vector $\mathrm{r}_{ji} \equiv \mathrm{x}_j - \mathrm{x}_i$, ϵ^2, and result of $\mathrm{r}_{ji}{}^2 + \epsilon^2$ calculated as Code 2, respectively.

```
1 /* Implementation of CUDA SDK */
2 float r2 = rji.x * rji.x + rji.y * rji.y + rji.z * rji.z;
3 r2 += eps2;
4 /* Our Implementation */
5 float r2 = eps2 + rji.x * rji.x + rji.y * rji.y + rji.z * rji.z;
```

Code 2: Difference between CUDA SDK and our implementation

Above two source codes look like almost the same, however, generated instruction sets are quite different. For implementation of CUDA SDK, 1 multiplication and 2 fused multiply-add (FMA) operations are performed at the line 2, and 1 addition is performed at the line 3. Thus, its computational cost corresponds to 4 clock cycles according to CUDA C Programming Guide [9]. On the other hand, only 3 FMA operations are performed using 3 clock cycles in our implementation. Therefore, our implementation would be faster than CUDA SDK. The most important point of this optimization is that the innermost loop includes the calculation of `r2`; thus, this small care directly increase performance.

In addition, we set ”L1 cache preferred” since experiments exhibit slight performance increase compared with ”shared memory preferred” in most cases. In addition, we increase the number of unroll counts up to 128, four times greater value than that of CUDA SDK.

Finally, we must take two additional treatments not to decrease performance. Our accumulation algorithm needs to access array using an index calculated from thread index, thus the array `src` is placed in local memory, much slower than register. This is because ”Arrays for which it cannot determine that they are indexed with constant quantities” is likely to place in local memory [9]. To avoid performance decrease due to using slow local memory within the innermost loop, we have decided to copy data of `src` to a new temporary array just before the accumulation process. Furthermore, usage of 128-bit memory accesses instead of 32-bit memory accesses would contribute to performance increase; therefore, we store data of acceleration in an array of union ”`aligned4float`” defined as shown in Code 3.

```
typedef union __align__(16) {
  float4 r4;
  float val[4];
} aligned4float;
```

Code 3: Definition of union

By using the union, the CUDA compiler knows that the array of acceleration is 128-bit aligned, thus 128-bit memory accesses can be performed. Furthermore, components stored in the union can be indicated using an index as Code 1. This property is essential to accumulate distributed information in multiple threads without synchronization, and the structure `float4` provided by CUDA does not have the property.

4. Performance Measurements

We have taken performance measurements of our implementation and CUDA SDK using an environment listed on Table 1. In performance measurements, we measure executing time of calculating gravitational acceleration with

CPU	AMD Opteron 6128 (2.0 GHz, 8 cores)
RAM	8 GB (DDR 3)
OS	CentOS 5.5 (x86_64)
GPU	Tesla C2070 (1.15 GHz, 448 CUDA cores)
Video RAM	6 GB (GDDR 5, ECC on)
Compiler	gcc 4.1.2 (-O3), nvcc 4.0
CUDA SDK	CUDA SDK 4.0.17

Table 1: Evaluation environment

direct summation. In other words, communication time between the CPU and the GPU via PCI Express is not included. We study dependency of performance on N_i, N_j, and thread-block structure. Parameter region of N is N_i, N_j = 256, 512, 1024, $\cdots$, 1048576. In our implementation, number of threads per block $T_{\rm tot}$ is always 256, same with CUDA SDK, so only one parameter $T_{\rm sub}$, represents a number of threads which share an i-particle, determine the thread-block structure. That is to say, if $T_{\rm sub} = 2$ then 2 threads calculate acceleration of the common i-particle and a block calculate acceleration of 128 i-particles. We have measured performance for $T_{\rm sub}$ = 1, 2, 4, 8, and 16.

In order to compare the performance of our implementation and that of CUDA SDK under the same condition, we have to take three additional treatments. At first, the gravitational constant G is assumed to be unity in our implementation as same as CUDA SDK. In addition, we omit calculation about gravitational potential in our implementation, not included in CUDA SDK, as the second treatment. In our implementation, we have separately implemented function for orbit integration of N-body particles and gravitational interaction. This separate implementation is a desirable feature for easily changing implementation of time integration scheme (e.g. leap-frog integrator using fixed time step, or Runge-Kutta method using adaptive time step and so on). Thus, we separate the function in contrast to the all-in-one implementation, calculation of gravitational force and orbit integration is performed in a kernel function, of CUDA SDK. Since this difference of implementation strategy leads the difference of computational amount and an unfair comparison, we omit time integration of position and velocity in our performance measurements as the third treatment. The effect of the first and third treatments on our performance measurements is expected to be negligibly small due to its smallness of computational amount ($O(N)$). On the other hand, effects of omitting the calculation of gravitational potential cannot be negligible due to its computational complexity of $O(N^2)$. However, the additional execution time corresponds to only 1 clock cycle, since the calculation can be performed by one additional FMA operation.

We show results of performance measurements about our implementation for $N_i = N_j$ case in Figure 2. The

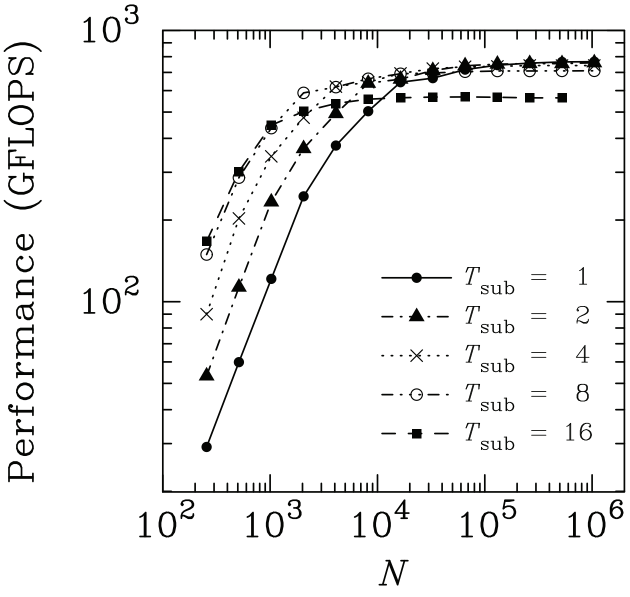

Figure 2: Measured performance of our implementation against N

horizontal axis is the number of N-body particles, and the vertical axis shows the performance under the assumption of one interaction corresponds to 26 floating-point operations. Filled circles with a full line, triangles with a dot-dashed line, crosses with a dotted line, open circles with a triple-dot-dashed line, and squares with a dashed line show the performance of $T_{\rm sub} = 1$, 2, 4, 8, and 16, respectively. The figure shows some clear behavior as follows: 1) the performance increase in low N region looks like proportional to N, 2) the measured performance saturate in the large N region, 3) critical N which determine transition point of performance dependency on N tends to decrease with increasing of $T_{\rm sub}$, and 4) the sustained performance of $T_{\rm sub} = 16$ is much lower than that of other $T_{\rm sub}$. The achieved peak performance of 767 GFLOPS for $N_i = N_j = 1048576$, $T_{\rm sub} = 1$ corresponds to 74 % of C2070's theoretical peak performance for single-precision floating-point operations.

To evaluate performance improvement from CUDA SDK, we show speed up of our implementation from CUDA SDK against the number of N-body particles in Figure 3. At the left panel in Figure 3, filled circles with a full line, triangles with a dot-dashed line, crosses with a dotted line, open circles with a triple-dot-dashed line, and squares with a dashed line show the performance improvement of our implementation from CUDA SDK for $T_{\rm sub} = 1$, 2, 4, 8, and 16, respectively (same symbols and lines with Figure 2). The figure clearly shows that performance of high $T_{\rm sub}$ in low N region increase drastically, and speed up is greater than 5% in all N, $T_{\rm sub}$. At the right panel in Figure 3, we compare our implementation and CUDA SDK for the fastest $T_{\rm sub}$ and plot using triangles with a dot-dashed line. The fastest $T_{\rm sub}$ is $T_{\rm sub}$ which can achieve the best performance for given N (e.g. 16 for $N = 256$, 1 for $N = 1048576$). To evaluate performance increase due to optimization about Code 2, we modify the original implementation of CUDA SDK and plot speed up of the optimized version from the original version of CUDA SDK using circles with a full line. The performance increase of our implementation reach to 69.5 % in maximum, 7.0 % in minimum from the original version of CUDA SDK, and that of the optimized CUDA SDK is 5.9 % in maximum. The figure shows that our implementation is always faster than the optimized version of CUDA SDK, and performance for low N region

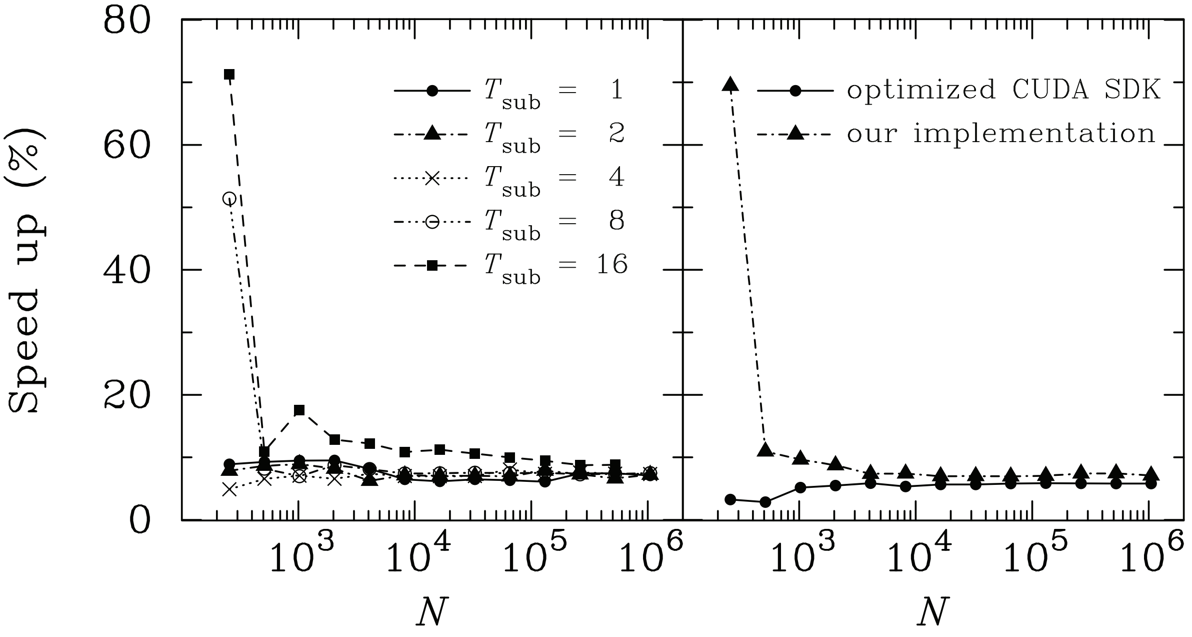

Figure 3: Speed up from slightly modified version of CUDA SDK 4.0.17

drastically increase.

5. Performance Analysis Based on Clock Cycles

In this section, we present a detailed performance analysis based on clock cycles needed to calculate gravitational interaction.

First of all, we must point out that the most basic evaluation cannot explain results of performance measurements in section 4. For $T_{\rm sub} = 1$, calculation of gravitational acceleration needs execution of a load instruction from shared memory using 128-bit memory access, 3 subtractions, 3 multiplications, 6 FMAs, and one `rsqrtf()`, corresponds to 24 clock cycles, per interaction in our implementation. Therefore, C2070 can calculate $1.15 \times 448/24 \simeq 21.47$ billion interaction per second, so expected execution time is 51.2 seconds for $N_i = N_j = 1048576$, corresponds to 558 GFLOPS. Thus, expected performance is much lower than the measured performance of 767 GFLOPS. This large discrepancy is due to over-wrapped execution of two different instructions at the same time. Therefore, we must evaluate the number of clock cycles per interaction $C_{\rm cal}$ using results of performance measurements to investigate effects due to such over-wrapped execution.

To evaluate $C_{\rm cal}$ and other quantities related to performance precisely, we estimate total clock cycles to complete calculation of gravitational interaction $C_{\rm tot}$. For simplicity, we assume N_i, and N_j are multiples of $T_{\rm tot}$ in below analysis.

To start the computation, there is a certain cost to start kernel function $C_{\rm ker}$. In addition, position data of i-particle must be loaded from global memory before calculating interaction. At the time, high latency L about 400 – 800 clock cycles occur [9]. Data transfer time from global memory is negligibly small than L due to GPU's wide memory band width of 144 GB/s. Once each thread stores position data of i-particle in the registers, then force calculation loop start. To achieve high performance, we divide the force calculation loop in the following two steps: position data of $T_{\rm tot}$ j-particles are copied from global memory to shared memory at the first step, then threads calculate gravitational interaction among $T_{\rm tot}/T_{\rm sub}$ i-particles and $T_{\rm tot}$ j-particles at the second step. This calculation loop is performed $N_j/T_{\rm tot}$ times to deal all j-particles. Thus, execution of force calculation in a block needs clock cycles of

$$C_{\rm int} = \frac{N_j}{T_{\rm tot}} \times \left(L + \frac{T_{\rm tot}}{T_{\rm sub}} C_{\rm cal}\right) \times \frac{T_{\rm tot}}{N_{\rm core}} = \frac{N_j}{N_{\rm core}} \left(L + \frac{T_{\rm tot}}{T_{\rm sub}} C_{\rm cal}\right), \tag{2}$$

where $N_{\rm core}$ is the number of CUDA cores per Streaming Multiprocessor (SM), 32 for GPUs of compute capability 2.0. The term $T_{\rm tot}/N_{\rm core}$ comes from the fact that the warp schedulers automatically divide execution of computation in $T_{\rm tot}/N_{\rm core}$ groups due to the limited number of CUDA cores. Furthermore, if one SM contains multiple blocks, then the latency L can be hidden by over-wrapping of data transfer from global memory and calculation of interaction among i-particles and j-particles. For such case, $C_{\rm int}$ should be decrease to

$$C_{\rm hid} = \frac{N_j}{N_{\rm core}} \max\left(L, \frac{T_{\rm tot}}{T_{\rm sub}} C_{\rm cal}\right). \tag{3}$$

At the end of computation, $T_{\rm sub}$ threads accumulate acceleration data of i-particle to a thread if needed, and resultant data are transferred to global memory in L clock cycles. Clock cycles to complete the accumulation process $C_{\rm acc}$ is a function of $T_{\rm sub}$. For our implementation explained in section 2, $C_{\rm acc}(T_{\rm sub}) = 0$, 17, 20, 34 and 45 for $T_{\rm sub} = 1$, 2, 4, 8 and 16, respectively. To summarize the above analysis, clock cycles to complete computation of a block, $C_{\rm block}$, is expressed as

$$C_{\rm block} = \begin{cases} C_{\rm int} + C_{\rm ker} + 2L + C_{\rm acc}(T_{\rm sub}) & \equiv C_{\rm block,\,int}, \quad \text{for } B_{\rm SM} = 1, \\ C_{\rm hid} + \left[C_{\rm ker} + 2L + C_{\rm acc}(T_{\rm sub})\right]/B_{\rm SM} & \equiv C_{\rm block,\,hid}, \quad \text{for } B_{\rm SM} > 1, \end{cases} \tag{4}$$

where $B_{\rm SM}$ is the number of blocks assigned to an SM at the same time. The factor of $1/B_{\rm SM}$ represents an effect of over-wrapped memory transfer time caused by the existence of multiple blocks in an SM.

To evaluate $C_{\rm tot}$ using $C_{\rm block}$, we must evaluate how many times blocks repeat the loop about computation. The total number of blocks, $B_{\rm tot}$, is expressed as $N_i/(T_{\rm tot}/T_{\rm sub}) = N_i T_{\rm sub}/T_{\rm tot}$, therefore, the number of the loop, N_{tot}, is represented using the number of SMs, $N_{\rm SM}$, as

$$N_{\rm tot} = {\rm ceil}\left(\frac{B_{\rm tot}}{N_{\rm SM}}\right) = {\rm ceil}\left(\frac{T_{\rm sub} N_i}{T_{\rm tot} N_{\rm SM}}\right). \tag{5}$$

If $B_{\rm SM} \geq 2$, then some part of $N_{\rm tot}$ loops compute for $C_{\rm hid}$ clock cycles, not $C_{\rm int}$. The number of such loops is $N_{\rm hid} = {\rm floor}(N_{\rm tot}/B_{\rm SM})$, and the number of remain loops is $N_{\rm rem} = N_{\rm tot} - B_{\rm SM} N_{\rm hid}$, therefore, $C_{\rm tot}$ is expressed as

$$\begin{aligned} C_{\rm tot} &= N_{\rm hid} \times (B_{\rm SM} C_{\rm block,\,hid}) + N_{\rm rem} \times C_{\rm block,\,int} & (6)\\ &= N_{\rm hid} \times \left[\frac{B_{\rm SM} N_j}{N_{\rm core}} \max\left(L, \frac{T_{\rm tot}}{T_{\rm sub}} C_{\rm cal}\right) + C_{\rm ker} + 2L + C_{\rm acc}(T_{\rm sub})\right] \\ &\quad + N_{\rm rem} \times \left[\frac{N_j}{N_{\rm core}} \left(L + \frac{T_{\rm tot}}{T_{\rm sub}} C_{\rm cal}\right) + C_{\rm ker} + 2L + C_{\rm acc}(T_{\rm sub})\right]. & (7) \end{aligned}$$

Hereafter, we estimate unknown parameters $C_{\rm cal}$, L, and C_{ker} using results of performance measurements.

To begin with, we evaluate the maximum $B_{\rm SM}$ of our implementation. The number of available registers and the capacity of shared memory per SM, 32768 and 16 KB for C2070 with L1 cache preferred option, determine the $B_{\rm SM}$. All threads use 27 registers in maximum, therefore, $B_{\rm SM} \leq 32768/(27 \times 256) = 4.7$. Since the capacity of memory to store position is 16 byte (4 elements of single-precision floating point numbers), each block uses 8 KB to store the position data of 256 particles. Therefore, $B_{\rm SM} = 2$ for our implementation due to the limited capacity of shared memory. Here, we estimate $C_{\rm cal}$ as a demonstration of how to use Equation 7. For $N_i \gg T_{\rm tot} N_{\rm SM} = 3584$, $N_{\rm hid}$ grows much greater than $N_{\rm rem}$ and the contribution of the second term of the Eq. 7 to the $C_{\rm tot}$ becomes negligibly small. Furthermore, if $N_j \gg N_{\rm core}/B_{\rm SM}$, then execution time for the inside of the interaction loop becomes much greater than that for the outside of the loop, thus the contribution of the term $C_{\rm ker} + 2L + C_{\rm acc}(T_{\rm sub})$ becomes negligible small. To assess the contribution of $C_{\rm cal}$ to the $C_{\rm tot}$ precisely, $T_{\rm sub}$ should be small to hide contribution of the latency L by increasing arithmetic intensity $T_{\rm tot} C_{\rm cal}/(T_{\rm sub} L)$. Therefore, $N_i = N_j = 1048576$ and $T_{\rm sub} = 1$ is the most suitable case in our performance measurements (section 4): estimation of $C_{\rm tot}$ and the corresponding execution time in the case is $2.45 C_{\rm cal}$ billion clock cycles and $2.13 C_{\rm cal}$ seconds, respectively. Measured execution time of our implementation is 37.3 seconds as listed on the top line of the Table 2; therefore, $C_{\rm cal}$ is estimated to be 17.5 clock cycles.

Continuously, we estimate clock cycles of the latency L accompanied with accesses to global memory. To evaluate L, $T_{\rm sub}$ must be 16. This is because, that is the only case of L is greater than calculation of $T_{\rm tot} C_{\rm cal}/T_{\rm sub} = 16 \times 17.5 =$

N_i	N_j	T_{sub}	note	execution time (sec.)
1048576	1048576	1	`rsqrtf()` is performed	37.3
524288	524288	16	—	12.7
256	256	16	—	1.02×10^{-5}
524288	256	16	—	8.25×10^{-3}
1048576	1048576	1	8 multiplications are performed instead of `rsqrtf()`	46.3

Table 2: Measured execution time to estimate clock cycles

280 clock cycles, even if L is around 400 clock cycles of the minimum value appeared in CUDA C Programming Guide [9]. For the case of $N_i = N_j = 524288$ and $T_{\mathrm{sub}} = 16$, measured execution time is 12.7 seconds (Table 2) and clock cycles and corresponding time predicted by Equation 7 are $38.3L$ million clock cycles and $33.3L$ milliseconds, respectively. Therefore, L corresponds to 381 clock cycles in our implementation.

At the end of a series of estimation, we estimate C_{ker}. To evaluate the contribution of the C_{ker} to the C_{tot} precisely, N_j/N_{core} and $T_{\mathrm{tot}}C_{\mathrm{cal}}/T_{\mathrm{sub}}$ should be small to minimize the contribution from inside of the interaction loop. $N_j = 256$, $T_{\mathrm{sub}} = 16$ is the desired case, then C_{tot} is derived to be $N_{\mathrm{hid}} \times (18L + 45 + C_{\mathrm{ker}})$ clock cycles using Equation 7. Results of performance measurements say C_{ker} is 4827 and 1206 clock cycles for $N_i = 256$ and 524288, respectively. The above estimated cost to start kernel function C_{ker} using two measurements differs. Therefore, we can only say that C_{ker} is order of 10^3 clock cycles, or several micro-seconds. The estimated clock cycles of C_{cal}, L, and C_{ker} are 17.5, 380, and order of 1000 clock cycles, respectively.

Here, we quantitatively discuss the reason why C_{cal} is 17.5 clock cycles, not evaluated value for our implementation of 24 clock cycles. The answer is due to encapsulation of execution time by over-wrapped execution of `rsqrtf()` and accesses to shared memory with other instructions at the same time.

First of all, we discuss effects delivered by usage of `rsqrtf()` function. The special function units for single-precision floating-point transcendental functions perform the `rsqrtf()` function [9]; thus, CUDA cores can perform simple floating-point arithmetic operations, such as addition multiplication, at the same time. To quantify the degree of the over-wrapped execution with `rsqrtf()` and other operations, we have taken an experiment: we measure performance when 8 multiplications are performed instead of `rsqrtf()` to operate 8 clock cycles. The results listed on Table 2 clearly exhibit over-wrapped execution `rsqrtf()` with other operations reduce the execution time. If the throughput of `rsqrtf()` is 8 clock cycles as listed on CUDA C Programming Guide [9], then measured execution time of `rsqrtf()` and 8 multiplications must be the same. However, the measured results of both case clearly different, and the difference of $46.3 - 37.3 = 9.0$ seconds means the effect of over-wrapped execution of `rsqrtf()` and other operations. The execution time of 9 seconds corresponds to 4.2 clock cycles, it means that 4.2 clock cycles of 8 clock cycles are hidden when `rsqrtf()` is performed. This is one of the reason why higher performance could achieve by using built-in functions performed by special function units, such as `__sinf()` and `__log2f`.

Furthermore, we can evaluate how much time to access shared memory is hidden, because there is only one chance to perform the over-wrapped execution of data transfer from shared memory and calculations. The previous basic estimation, assumed no over-wrapping of multiple instructions, says 51.2 seconds is necessary to complete calculation of gravitational interaction for $N_i = N_j = 1048576$ as mentioned before. In contrast, measured execution time to perform data transfer from shared memory and calculation of 20 clock cycles is only 46.3 seconds as listed on Table 2. Therefore, 2.3 clock cycles, derived from the difference of 4.9 seconds, of 4 clock cycles is the hidden clock cycle due to over-wrapping of data transfer from shared memory and calculations. Such over-wrapping reduce the execution time of $4.2 + 2.3 = 6.5$ clock cycles, and C_{cal} becomes 17.5 clock cycles, much shorter than the original value of 24 clock cycles.

To clarify the origin of the performance improvement from CUDA SDK appeared in section 4, we re-examine the Equation 7 in below.

First of all, let us consider the origin of performance improvement of 7.0 % in minimum. It should not be related to the force accumulation process, because the results of $T_{\mathrm{sub}} = 1$ also show the performance improvement. In addition, dependency of the speed up from CUDA SDK on the number of N-body particles at the left panel in Figure 3 is quite weak, except for $T_{\mathrm{sub}} = 8$ and 16 for low N. Therefore, the origin of the performance increase is considered to be due to the term $C_{\rm cal}$. Our optimization concerning about $C_{\rm cal}$ is only about Code 2 which reduce execution time corresponds to 1 clock cycle, thus let us evaluate the effect of the optimization. $C_{\rm cal}$ is 17.5 clock cycles for our implementation, and 18.5 clock cycles is the expected value for that of CUDA SDK. The effect of this optimization is evaluated to be $18.5/17.5 \simeq 1.06$; thus, performance increase of this optimization is considered to be 6 %, quite close value to the 7.0 %. Of course, this speed up of 6 % also explains of the optimized version of CUDA SDK shown at the right panel in Figure 3. The origin of performance increase from CUDA SDK reported by Hamada et al. (2009) would be the same with the above optimization. For GPUs of compute capability 1.x, 4 and 8 clock cycles are need to perform `rsqrtf()` and load from the shared memory, respectively. As a result, the total number of clock cycles for computing interaction is same with the case of compute capability 2.0. If we assume the same rate of clock cycles is hidden by over-wrapping with other operations for older GPUs, then the computation needs 17.3 clock cycles per interaction. The performance increase of $636\text{GFLOPS}/598.5\text{GFLOPS} \approx 1.063$ is fairly close to $(17.3 + 1)/17.3 \simeq 1.058$, therefore, the main reason of the increase is considered to be due to this optimization as same with this work.

Now, let us consider the parameter region of small N to evaluate effects of our force accumulation technique without synchronization. The term concerning the force accumulation process, $C_{\rm acc}(T_{\rm sub})$ of Equation 7, has no dependence on N_j, so effects of the term become significant only for low N_j case. Thus, it is natural to consider that the reduced clock cycles of $C_{\rm acc}(T_{\rm sub})$ is the origin of performance increase appeared in low N region of Figure 3. To quantify the difference with CUDA SDK, we must evaluate the cost of synchronization, but it is not easy. According to the CUDA C Programming Guide, the throughput for `__syncthreads()` is "16 operations per clock cycle", thus 2 clock cycles are necessary to perform `__syncthreads()` since $N_{\rm core}$ is 32. The operation `__syncthreads()` must be executed by whole warps within a block; consequently, the execution time becomes $T_{\rm tot}/\texttt{warpSize} = 8$ times greater. However, the above estimation is valid only for the most lucky case of the whole warps have been already synchronized, and the necessary clock cycles can grow much bigger without upper limit and reduce opportunities to over-wrapped execution of multiple instructions, if at least one warp delays with other warps. For the above reason, we omit a quantitative comparison between our implementation and CUDA SDK. In our implementation, execution time of $C_{\rm acc}(T_{\rm sub})$ is negligibly small compared with that of $2L + C_{\rm ker}$ in any case. The significant performance increase of our implementation appeared in Figure 3 suggest that the execution time of $C_{\rm acc}(T_{\rm sub})$ for CUDA SDK, which include synchronization and exclusive control, would not be negligible small in contrast to our implementation. According to the Equation 7, the contribution of $C_{\rm acc}(T_{\rm sub})$ to the total execution time becomes smaller with N increasing, it means the benefits of the optimization about force accumulation process are not available for the large N. This is because the control parameter is $B_{\rm SM}N_j/N_{\rm core}$, so "large N" means sufficiently large N to neglect the contribution of $C_{\rm acc}(T_{\rm sub})$ to the total execution time. In fact, degree of the performance improvement decreases with N increasing in Figure 3.

At the end of this section, we explain trends appeared in the Figure 2. As we mentioned in section 4, our implementation has five trends: 1) the performance increase in low N region looks like proportional to N, 2) the performance increase saturate in the large N region, 3) critical N which determine the transition point of the performance dependency on N tends to decrease with increasing of $T_{\rm sub}$, 4) the sustained performance decreases gradually with $T_{\rm sub}$ increase, and 5) that of $T_{\rm sub} = 16$ is much lower than that of any other $T_{\rm sub}$. At first, the performance increase in low N region is due to the dependence on the term $N_{\rm hid}$ of Equation 7. Since $N_{\rm hid}$ is approximately expressed as $T_{\rm sub}N_i/(T_{\rm tot}N_{\rm SM}B_{\rm SM}) = T_{\rm sub}N_i/7168$, the increase of N does not mean increase of $N_{\rm hid}$ for the region of $N_i \leq 7168/T_{\rm sub}$. Therefore, the execution time is proportional to N_j only while the amount of computation is proportional to N_iN_j; consequently, the performance increase is proportional to N. Such performance increase continues as long as $N_{\rm hid}$ is less than a few, which corresponds to the condition not to waste 14 SMs of a GPU. When $N_{\rm hid}$ sufficiently grows up, then the performance reaches the peak performance (the second trend). The condition to achieve the peak performance of GPU is that $N_{\rm hid} \gtrsim 7168/T_{\rm sub}$, therefore, the third trend appears due to the term of $1/T_{\rm sub}$. The origin of the fourth trend is considered to be a decrease of arithmetic intensity $T_{\rm tot}C_{\rm cal}/(T_{\rm sub}L) \simeq 12/T_{\rm sub}$ in the interaction loop to calculate gravitational interaction. The over-wrapping of the data transfer from global memory and calculation of gravitational interaction becomes easier for the higher value of the arithmetic intensity (i.e. lower $T_{\rm sub}$). Since increasing degree of the over-wrapping would contribute for the increase of performance, that would be the origin of the fourth trend. If the arithmetic intensity becomes lower than unity, then the latency due to accessing global memory suppress the performance. This is the reason why the sustained performance of $T_{\rm sub} = 16$ is much lower than any other $T_{\rm sub}$.

6. Discussion and Summary

Here, we discuss estimation of floating-point operation count per interaction. Historically, a variety number of operation counts have been assumed to evaluate performance of collisionless N-body simulations. The floating-point operations count of 30 is assumed to evaluate performance of GRAPE series for collisionless systems [3, 4], 20 is GPGPU code by CUDA SDK [6], and 38 is the most frequently assumed value using various architectures [10, 5, 11, 7, 8]. Different estimations about computational cost of the inverse square root are the origin of the above difference (e.g. counting of 20 floating-point operations as an inverse square root operation leads to the assumption of 38 floating-point operations per interaction). Since 8 clock cycles are need to perform `rsqrtf()` for GPUs of compute capability 2.0 [9] whereas addition or multiplication need only 1 clock cycle to perform, the computational cost of `rsqrtf()` corresponds to 8 floating-point operations. Therefore, 26 floating-point operations per interaction, assumed value in this work, is the most plausible estimation for GPUs of compute capability 2.0. Of course, this estimation of floating-point operations does not affect the speed up ratio appeared in Figure 3 because the overall factor is canceled out.

Furthermore, we would like to mention about impacts of our implementation to N-body simulations performed in studies of astrophysics. At least, 7.0 % of computational time for direct N-body simulations is reduced as shown in Figure 3. Since the speed up rate is much higher in the low N region, the impact would become more powerful if effective number of N-body particles is reduced by combining with tree method. However, this is only our expectation, so quantitative verification of this speculation is an important future work.

We have implemented a fast collisionless N-body code which runs on GPU, the peak performance of 767 GFLOPS. For the sake of our optimizations, we can achieve the performance increase of 7.0 % in minimum and 69.5 % in maximum (due to reduction of computation and improving the algorithm for force accumulation, respectively) from CUDA SDK. Our detailed performance analysis shows that the following two quantities determine the performance of collisionless N-body simulations: first one is the number of running streaming multiprocessors and another is the clock cycle ratio of latency for accessing global memory and operations to calculate gravitational interaction.

Acknowledgments

We thank Go Ogiya for fruitful discussion in detail, and Taisuke Boku for his useful comments. Numerical Simulations have been performed with hexa cluster at the Center for Computational Sciences, University of Tsukuba. This work was partially supported by the program of the Pre-Strategic Initiatives, University of Tsukuba and by the Grant-in-Aid for Scientific Research (A)(21244013).

References

1. R. W. Hockney, J. W. Eastwood, Computer simulation using particles, 1988.
2. J. Barnes, P. Hut, A hierarchical O(N log N) force-calculation algorithm, Nature 324 (1986) 446–449. doi:10.1038/324446a0.
3. S. K. Okumura, J. Makino, T. Ebisuzaki, T. Fukushige, T. Ito, D. Sugimoto, E. Hashimoto, K. Tomida, N. Miyakawa, Highly Parallelized Special-Purpose Computer, GRAPE-3, Publications of the Astronomical Society of Japan 45 (1993) 329–338.
4. A. Kawai, T. Fukushige, J. Makino, M. Taiji, GRAPE-5: A Special-Purpose Computer for N-Body Simulations, Publications of the Astronomical Society of Japan 52 (2000) 659–676.
5. T. Hamada, T. Iitaka, The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units, ArXiv Astrophysics e-printsarXiv:arXiv:astro-ph/0703100.
6. L. Nyland, M. Harris, J. Prins, Fast N-Body Simulation with CUDA (2007).
7. T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, M. Taiji, 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, SC '09, ACM, New York, NY, USA, 2009, pp. 62:1–62:12. doi:http://doi.acm.org/10.1145/1654059.1654123.
8. T. Hamada, K. Nitadori, 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, SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–9. doi:http://dx.doi.org/10.1109/SC.2010.1.
9. Nvidia, NVIDIA CUDA C Programming Guide Version 4.0 (2011).
10. A. Kawai, T. Fukushige, J. Makino, $7.0/Mflops astrophysical N-body simulation with treecode on GRAPE-5, in: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '99, ACM, New York, NY, USA, 1999. doi:http://doi.acm.org/10.1145/331532.331598.
11. K. Nitadori, J. Makino, Sixth- and eighth-order Hermite integrator for N-body simulations, New Astronomy 13 (2008) 498–507. doi:10.1016/j.newast.2008.01.010.