The Nitadori's Trick: High-Performance Quasi Double-Precision Method using Single-Precision Hardware for N-body Simulations with GPUs

Abstract

The recent advancement of computer technology in commodity hardware enables us to perform scientific simulations on them. For example, the Cell processor in the PLAYSTATION 3 (PS3) has a 179 Gflops peak performance in single precision, which exceeds the fastest CPU for PCs. Moreover, GeForce GTX280, the latest Graphics Processing Unit (GPU) by NVIDIA, has 933 Gflops peak performance in single precision. However, the double precision performance of these computers is not so good: a PS3 has 17 Gflops and a GTX280 has 78 Gflops, which is comparable to current CPUs for PCs. Using mixed-precision arithmetic is the key to extract the potential of these hardware. In this paper, a new algorithm to use mixed-precision arithmetic for the accumulation part of forces in Molecular Dynamics (MD) simulations is proposed. The algorithm achieves comparable accuracy to that of double precision with single-precision hardware by quasi double-precision method. Especially, no offset of accumulated forces is the key feature of the method, which enhance the stability of the MD simulation. The speed of the quasi double-precision calculation is in between those of double and single precisions.

1 Introduction

Precision of arithmetic operations in computer simulation is one of the crucial issues for obtaining reasonable results. However, during the past decade most simulations have used the double precision as a default since it is usually accurate enough, and CPUs perform double precision arithmetic operations with similar performance to single precision. One exceptional case is when you use a short vector, Single Instruction Multiple Data (SIMD) unit, such as Streaming SIMD Extensions (SSE) in x86 CPU or AltiVec in PowerPC CPU. For this case, the calculation speed of single precision is twice as fast as that of double precision with these units.

In the past two or three years, some processors having completely different performances for different precisions have been developed. For example, the Cell processor [1] in PLAYSTATION 3 (PS3) [2] has a 179 Gflops single precision peak performance, and only 17 Gflops in double precision. GeForce GTX280, the latest GPU from NVIDIA [3], has 933 Gflops single precision performance, and only 78 Gflops in double precision. Though the single precision performances of these processors exceed the latest Intel CPU, such as Xeon X7460 (128 Gflops) [4], the double precision performance is comparable to the CPU (64 Gflops). In order to achieve maximum performance on these processors, it is necessary to use as many single precision arithmetic operations as possible. The challenge in using single precision operations is the reduction of round-off errors in the calculation results. Using mixed-precision arithmetic operations is a promising approach to achieve high performance and high accuracy at the same time.

Buttari et al. [5] used mixed-precision for performing matrix operations. In the iterative loop for solving matrix equations, they used single precision operations for the most computationally heavy parts. Other parts are performed with double precision because higher accuracy is required for assuring the convergence of the iterative solution. By using this technique, they achieved the same results as the double precision with much higher performance on the Cell processor.

MD simulation is a powerful technique to investigate the physical property of condensed mater at an atomic level [9]. Atoms in a system interact with each other via the Coulomb, van der Waals, and bonding forces. Among these forces, two body interactions with the Coulomb and van der Waals forces are dominant when the number of atoms in a system becomes large. Therefore the computational cost of MD simulations can become overwhelming for large systems. To enhance the performance of the MD simulation code, many techniques have been used on the basis of mixed-precision. For example, GROMACS [10], one of the fastest MD simulation codes with a single thread, uses SSE unit in Intel CPUs to enhance the performance. NAMD [11, 12], one of the fastest parallel MD codes, supports GPU to enhance the performance. In both codes, the two body force calculations are performed in single precision to extract the maximum performance of the hardware.

In spite of the previous efforts of single precision force calculation on GPUs [11, 12, 13], the accumulation of forces is not performed accurate enough, because the hardware did not support double precision at that time. Since summing up many forces from surrounding atoms with single precision operations often cause a rounding off error [14], a more accurate precision, such as double precision, should be used. Takahashi et al. [6, 7] used a mixed-precision for accelerating the boundary element method with a GPU. Nitadori [18] and Hamada et al. [8] also used mixed-precision for gravitational N-body simulations with a GPU. Both of them achieved quasi double-precision by using a pair of single-precision variables to increase the accuracy at the force accumulation part.

In this paper we propose a new method to perform the accumulation of forces with comparable accuracy to double precision using single-precision hardware, and the speed of the method is in between those of double and single precisions. In section 2, the previous idea of using quasi double-precision is described. In section 3, the proposed method is explained in detail. The accuracy and the performance of the method are presented in section 4. In section 5, summary and comments are shown.

2 Quasi Double-Precision Method

In this section, the method to increase the accuracy of summation is described. Here, we concentrate on the situation that a single precision value is added to an accumulated value, which is expressed in various formats.

2.1 Error in floating point expression

When a number, x, is expressed by a floating point value, $fl(x)$, the following condition is satisfied,

$$\frac{fl(x)}{x} < 2^{-n}, \tag{1}$$

where n denotes the number of mantissa bits including a hidden bit. This equation shows the range of the error of the floating point expression. If the double and single precision expression of x is expressed as $fl_D(x)$ and $fl_S(x)$ respectively, the errors in these expressions are:

$$\begin{aligned} \frac{fl_D(x)}{x} &< 2^{-n_D} \simeq 10^{-16}, & (2)\\ \frac{fl_S(x)}{x} &< 2^{-n_S} \simeq 10^{-7}, & (3) \end{aligned}$$

since the number of mantissa bits of double and single precision are $n_D = 53$ and $n_S = 24$ respectively in the IEEE754 standard. 10^{-7} error of single precision is often insufficient for target applications. One easy solution is to use double precision, but the other is to use multiple single precision variables to express one value. We will use x_D or x_S instead of $fl_D(x)$ or $fl_S(x)$ hereafter for simplicity.

2.2 Knuth & Dekker's trick

Knuth [15] and Dekker [16] proposed a method to sum two values without losing so much accuracy. If two single precision values, x_S and y_S, satisfy $|x_S| \geq |y_S|$, their sum can be expressed by the following expression:

$$x_S + y_S = z_S + zz_S, \tag{4}$$

where z_S and zz_S are:

$$\begin{aligned} z_S &= fl_S(x_S + y_S),\\ w_S &= fl_S(z_S - x_S),\\ zz_S &= fl_S(y_S - w_S). \end{aligned} \tag{5}$$

```c
typedef struct {
    float hs;
    float ls;
} SS;

SS add_knuth_and_dekker1(float xs, float ys)
{
    float ws;
    SS z;

    z.hs = xs   + ys;
    ws   = z.hs - xs;
    z.ls = ys   - ws;

    return z;
}
```

Figure 1. Knuth & Dekker's trick in C language when $|x_S| \geq |y_S|$.

```c
SS add_knuth_and_dekker2(float xs, float ys)
{
    float ws, vs, z1s, z2s;
    SS z;

    z.hs = xs   + ys;
    ws   = z.hs - xs;
    z1s  = ys   - ws;
    vs   = z.hs - ws;
    z2s  = vs   - xs;
    z.ls = z1s  - z2s;

    return z;
}
```

Figure 2. Knuth & Dekker's trick in C language for arbitrary x_S and y_S.

Here $fl_S(x+y)$ and $fl_S(x-y)$ denote the single precision floating point addition and subtraction respectively. z_S and zz_S can be understood as the higher and lower part of the result respectively, as they satisfy the following equation:

$$|\frac{z_S}{zz_S}| < 2^{-n_S}. \tag{6}$$

In the following paper, we call this combination of higher and lower expressions a *pair expression*. The error in the *pair expression*, $2^{-2n_S} = 2^{-48} \simeq 10^{-15}$, is comparable to double precision and much lower than single precision. Figure 1 shows the program for Equation (5) in C language. Here '`SS z`' is a variable for a *pair expression*: $\texttt{z.hs} = z_S$ and $\texttt{z.ls} = zz_S$.

Dekker also proposed a modified method which does not need the condition, $|x_S| \geq |y_S|$, as follows:

$$\begin{aligned} z_S &= fl_S(x_S + y_S), \\ w_S &= fl_S(z_S - x_S), \\ z1_S &= fl_S(y_S - w_S), \\ v_S &= fl_S(z_S - w_S), \\ z2_S &= fl_S(v_S - x_S), \\ zz_S &= fl_S(z1_S - z2_S). \end{aligned} \tag{7}$$

Figure 2 shows the program for Equation (7).

2.3 Takahashi & Iitaka's trick

Knuth & Dekker's trick is often used when much more accuracy is required than the hardware supported precision. One example is the DSFUN90 library [17] which uses multiple single-precision arithmetic. Knuth & Dekker's trick has already been used on GPUs. In this paper, we refer to Takahashi & Iitaka's trick [6, 7].

```
SS add_takahashi_and_iitaka(SS a, float ys)
{
  float ws, vs, z1s, z2s, rs, ts;
  SS z, b;

  z.hs = a.hs + ys;
  ws   = z.hs - a.hs;
  z1s  = ys   - ws;
  vs   = z.hs - ws;
  z2s  = vs   - a.hs;
  z.ls = z1s  - z2s;
  rs   = a.ls + z.ls;
  b.hs = z.hs + rs;
  ts   = b.hs - z.hs;
  b.ls = rs   - ts;

  return b;
}
```

Figure 3. Takahashi & Iitaka's trick in C language.

The target problem is to sum up single precision values into an accumulated value. Here we assume that we already have an accumulated value by a *pair expression*, a_S and aa_S, where a_S and aa_S are the higher and lower part of the accumulated result respectively. Then the summation of y_S and a *pair expression* of a_S and aa_S are performed by the following procedure in Takahashi & Iitaka's trick.

$$a_S + y_S + aa_S = b_S + bb_S, \tag{8}$$

where

$$\begin{aligned}
z_S &= fl_S(a_S + y_S),\\
w_S &= fl_S(z_S - a_S),\\
z1_S &= fl_S(y_S - w_S),\\
v_S &= fl_S(z_S - w_S),\\
z2_S &= fl_S(v_S - a_S),\\
zz_S &= fl_S(z1_S - z2_S),\\
r_S &= fl_S(aa_S + zz_S),\\
b_S &= fl_S(z_S + r_S),\\
t_S &= fl_S(b_S - z_S),\\
bb_S &= fl_S(r_S - t_S),
\end{aligned} \tag{9}$$

Figure 3 shows their method in C language, which requires 10 operations (3 summations and 7 subtractions). Here '`SS a`' holds a *pair expression* of a_S and aa_S, and '`SS b`' does of b_S and bb_S.

Their method can be rewritten by using Knuth & Dekker's trick as shown in Figure 4. The method is composed of three parts. In the first part (line 6 in Figure 4), Knuth & Dekker's trick (7) is used. In the second part (line 7 in Figure 4), lower part of *pair expressions* are added with each other. In the third part (line 8 in Figure 4), the method expressed in Equation (5) can be used. The reason is that $|z_S| \geq |r_S|$ is satisfied since r_S is the sum of the lower part of *pair expressions* and z_S is the higher part.

3 High-Performance Quasi Double-Precision Method

In this section, a proposed method for accumulating forces with quasi double-precision is presented.

3.1 Nitadori's trick

We have proposed a simplified method, Nitadori's trick [18], where only 4 operations are needed per summation, while Takahashi & Iitaka's trick costs 10 operations. Figure 5 shows the program in C language.

```
SS add_takahashi_and_iitaka(SS a, float ys)
{
  float rs;
  SS z, b;

  z  = add_knuth_and_dekker2(a.hs, ys);
  rs = a.ls + z.ls;
  b  = add_knuth_and_dekker1(z.hs, rs);

  return b;
}
```

Figure 4. Takahashi & Iitaka's trick in C language with Knuth & Dekker's subroutines.

```
SS add_nitadori(SS a, float ys)
{
    SS b;

    b    = add_knuth_and_dekker1(a.hs, ys);
    b.ls = a.ls + b.ls;

    return b;
}
```

Figure 5. Nitadori's trick in C language with Knuth & Dekker's subroutines.

In the method, two conditions are assumed. The first condition is that the accumulated value should be larger than a value which is going to be added. Therefore, the method of Equation (5) can be used instead of Equation (7). The other condition is that the target application should not be sensitive to the small decrease of the accuracy in a *pair expression*. For example, when 128 values are added to the accumulated value by this method, the lower part becomes larger and larger. Then the error in the *pair expression* is in the worst case 128 times larger than 2^{-48}, which is the normal accuracy of the single precision *pair expression*. When larger number of values are added, much larger error might be generated by this method. However in collision-less N-body simulation, the error does not seem to be a problem [8]. For other gravitational N-body simulations, detailed analysis will be presetnted elsewhere. In the MD simulation, this error has a notable effect in some cases.

3.2 Modified Nitadori's trick

Figure 6 shows the program to calculate a force (`fd`) by accumulating `nj` pairwise forces (`fs`) with Nitadori's trick. When `LARGE` is set to `0.0`, the first condition of the Nitadori's trick must be satisfied to obtain the correct result. This means that the absolute value of the accumulated value must be larger than that of each force ($|\mathtt{a[k].hs}| \geq |\mathtt{fs[k]}|$). However, when `LARGE` is set to be a large number, such as `(3<<20)`$\simeq 3 \times 10^6$, the first condition is always satisfied as each force is always smaller than `LARGE`. Note that if `LARGE` is set too large, the accuracy of the accumulation becomes lower because of the rounding off. We call this method modified Nitadori's trick.

A sample code to calculate pairwase force is shown in Figure 7. In the actual code used in Section 4, each particle has its own atom type and it is highly optimized to the GPU. `calcvdw` is a sample code to show how to call accumulation routine.

3.3 Proposed method : Narumi's trick

Figure 8 shows the proposed method expressed by C language. This method is based on the Nitadori's trick but prevents the second condition in the Nitadori's trick.

First, quasi double-precision struct, `SI`, is defined. This struct houses a single precision value (`float hs`) and a 32-bit integer (`int li`). `hs` and `li` correspond to the higher and lower part of a *pair expression* respectively. `LOWER_FACTOR` is the factor to be multiplied to a floating point value to convert it to an integer. `LOWER_FACTOR_1` is the factor to be multiplied to an integer to convert it back to floating point expression.

`LOWER_SHIFT` specifies the number of bits to be set 0 at the MSBs (Most Significant Bits) of the lower part of a *pair expression* (`li`). The preceding 0s guarantee the non-overflow of `li` even when `add_narumi` is called 128 (=`LOWER_LOOP=1<<LOWER_SHIF`
times. The reason for using 32-bit integers instead of single precision is that 32-bit integers have more resolution compared to single precision when a decimal point is fixed. Because of this expression, even when the ratio between the higher and lower part of the *pair expression* is changed, the accuracy of accumulation does not decrease as in the Nitadori's trick.

```
 1 #if 1 // change to 0 for modified Nitadori’s
 2       // and Narumi’s methods
 3 #define LARGE        0.0f
 4 #else
 5 #define LARGE_SHIFT  21
 6 #define LARGE  \
 7         (float)(3 << (LARGE_SHIFT-1))
 8 #endif
 9
10 void accumulate_nitadori(float ri[3],
11        float rj[][3], int nj, double fd[3])
12 {
13   int j, k;
14   SS a[3];
15   float fs[3];
16
17   for(k=0; k<3; k++){
18     a[k].hs = LARGE;
19     a[k].ls = 0.0f;
20   }
21   for(j=0; j<nj; j++){
22     pairwiseforce(ri, rj[j], fs);
23     for(k=0; k<3; k++){
24       a[k] = add_nitadori(a[k], fs[k]);
25     }
26   }
27   for(k=0;k<3;k++){
28     fd[k]  = a[k].hs - LARGE;
29     fd[k] += a[k].ls;
30   }
31 }
```

Figure 6. Accumulation of forces by Nitadori’s trick. For modified Nitadori’s trick, line 1 must be changed from `#if 1` **to** `#if 0`**, and** `LARGE_SHIFT` **must be set to a large enough number to satisfy** `LARGE`$\geq$ $|$`fs[k]`$|$**.**

```
void pairwiseforce(float ris[3], float rjs[3],
                   float fs[3])
{
  int k;
  float ds[3], rs, qs, ps;

  for(k=0; k<3; k++) ds[k] = ris[k] - rjs[k];
  rs = ds[0]*ds[0] + ds[1]*ds[1] + ds[2]*ds[2];
  if(rs != 0.0f){
    rs = 1.0f / rs;
    qs = rs * rs * rs;
    ps = rs * qs * (2.0f * qs - 1.0f);
    for(k=0; k<3; k++) fs[k] = ps * ds[k];
  }
  else{
    for(k=0; k<3; k++) fs[k] = 0.0f;
  }
}

void calcvdw(int ni, int nj, float ri[][3],
             float rj[][3], double fd[][3])
{
  int i;

  for(i=0; i<ni; i++){
    accumulate_nitadori(ri[i], rj, nj, fd[i]);
    // accumulate_narumi(ri[i], rj, nj, fd[i]);
  }
}
```

Figure 7. Calculatioin of van der Waals forces with (modified) Nitadori’s method.

```
typedef struct {
    float hs;
    int   li;
} SI;

#define LOWER_SHIFT  7
#define LOWER_LOOP   (1 << LOWER_SHIFT)
#define LOWER_FACTOR   \
        (float)(1LL << (23 - LARGE_SHIFT \
                      + 32 - LOWER_SHIFT))
#define LOWER_FACTOR_1 \
        ( 1.0f / LOWER_FACTOR )
#define MASK(n)          ((0x1<<(n)) -1)

SI add_narumi(SI a, float ys)
{
  SS b;
  SI c;

  b    = add_knuth_and_dekker1(a.hs, ys);
  c.hs = b.hs;
  c.li = a.li+(int)(b.ls * LOWER_FACTOR);

  return c;
}

void accumulate_narumi(float ri[3],
       float rj[][3], int nj, double fd[3])
{
  int j, jj, k;
  SI a[3];
  float fs[3];

  for(k=0; k<3; k++){
    a[k].hs = LARGE;
    a[k].li = 0;
  }
  for(jj=0; jj<nj; jj+=LOWER_LOOP){
    for(j=jj;j-jj<LOWER_LOOP && j<nj;j++){
      pairwiseforce(ri, rj[j], fs);
      for(k=0; k<3; k++){
        a[k] = add_narumi(a[k], fs[k]);
      }
    }
    for(k=0; k<3; k++){
      a[k].hs += (float)(a[k].li &
       (MASK(LOWER_SHIFT)<<(32-
       LOWER_SHIFT)))*LOWER_FACTOR_1;
      a[k].li &= MASK(32-LOWER_SHIFT);
    }
  }
  for(k=0; k<3; k++){
    fd[k] = a[k].hs - LARGE;
    fd[k]+= a[k].li * (double)LOWER_FACTOR_1;
  }
}
```

Figure 8. Accumulation of forces by proposed method. `LARGE` **is set to be a large enough number to satisfy $|\mathtt{a[k].hs}| \geq |\mathtt{fs[k]}|$.** `LOWER_SHIFT` **is set to such as 7 to prevent the second condition of Nitadori's trick.**

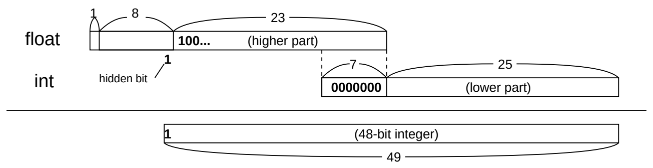

Figure 9. Block diagram of a *pair expression* **in Narumi's trick. A** `float` **and an** `int` **are used, and they correspond to a 48-bit integer.**

3.4 Accuracy of Narumi's trick

Figure 9 shows the block diagram of the *pair expression* of Narumi's trick. The reason `LARGE` is set to `3<<20` instead of `1<<21` is that the position of a decimal point and the sign are fixed for $|$`fs[k]`$|$ $<$`(1<<20)`. With this method the decimal points of the higher and lower part of the *pair expression* are fixed, which means that the *pair expression* is equivalent to a 48-bit integer as shown in Figure9. The accuracy of the *pair expression* is$2^{-48} \simeq 10^{-15}$, which is the same as the Takahashi &Iitaka's trick. Note that, similar to the modified Nitadori's trick, the accuracy becomes lower when `LARGE` is large compared to each force.

3.5 Number of operations in Narumi's trick

The number of operations per summation is 6 in Narumi's trick. 3 is used for Knuth & Dekker's trick. 3 is for the line 22: `c.li=a.li+(int)(b.ls * LOWER_FACTOR)`. The multiplication in this line can be deleted when each force is already multiplied by changing parameters to be `LOWER_FACTOR` times larger. The conversion from `float` to `int` also takes one operation. The operations in the lines 46-49 are negligible since these lines are performed every `LOWER_LOOP` times. Therefore, the total number of operations per a summation is 5 instead of 4 in the Nitadori's trick or 10 in the Takahashi & Iitaka's trick. When `LOWER_LOOP` is small (`LOWER_SHIFT` is small), the operations for lines 46-49 has some impact on the performance.

4 Numerical Test

In the following part, numerical error analysis of the proposed method is presented. Then its performance is described.

4.1 Test systems

Two systems are used to numerically test the proposed method. One is a protein system (System 1), which is used for testing accuracy. The other is a simple system with Lennard Jones particles (System 2), which is used for performance benchmark. For both systems, only the two body interaction of the van der Waals force is evaluated without cutoff. Other forces, such as the Coulomb and bonding forces, are not calculated. When the two body interaction is calculated on a molecule, forces on the bonded part on the molecule are skipped. One way to do this, is to mask the bonded interaction in the force summation loop. One needs to check the mask bit in the most internal loop, which causes a performance drop since the mask bit differs for each thread. Another way is to first calculate the two body interaction on all combinations in a system and then subtract the bonded part afterwards. The first one is called 'Exclude on the fly' and the other is called 'Exclude afterwards', in this paper. 'Exclude on the fly' algorithm is used in NAMD with GPU [11].

System 1 contains a protein, Scytalone dehydratase (2,715 atoms), and 208 water molecules. Total number of atoms is 3,339, and AMBER [19] force field is used to calculate the forces on atoms. System 2 contains 65,536 particles, and 16 atom types are randomly assigned to each particle. Parameters for the Lennard Jones potential are also randomly generated. Note that atom types and these parameters do not influence the performance. Exclusion of the bonded part is not needed.

4.2 Average error in force

Figure 10 shows the average error of forces for the various methods. Methods are summarized in Table 1. The average error of forces is calculated as follows:

$$f_{\text{err}} = \frac{\sum_{i=1}^{N} |\vec{F}_{\text{method}_i} - \vec{F}_{\text{double}_i}|}{\sum_{i=1}^{N} |\vec{F}_{\text{double}_i}|}, \tag{10}$$

where $\vec{F}_{\text{double}_i}$ is the force on the i-th particle with full double precision calculation (method G), $\vec{F}_{\text{method}_i}$ is the force of a specified method, and N is the total number of particles in the system. The average error of forces is similar when the 'Exclude on the fly' algorithm is used. As shown in our previous paper [20], average error of forces of 10^{-3} or larger is not sufficient for a stable MD simulation. Therefore, methods A and D are not useful with the 'Exclude afterwards' algorithm.

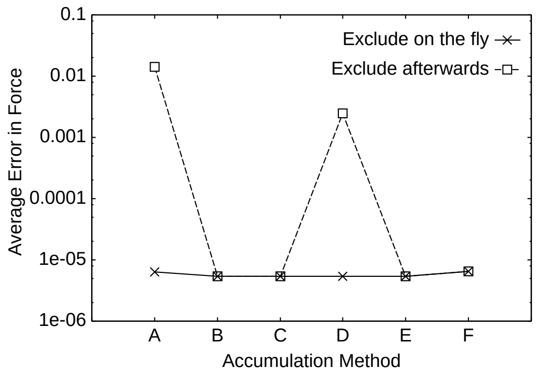

Figure 10. Average error of forces for System 1 against accumulation methods.

4.3 Average offset in force

Figure 11 shows the average offset of forces against various methods. Methods are summarized in Table 1. Average offset of forces is calculated as follows:

$$f_{\text{offset}} = \frac{|\sum_{i=1}^{N} \vec{F}_{\text{method}_i}|}{\sum_{i=1}^{N} |\vec{F}_{\text{method}_i}|}. \tag{11}$$

The numerator of the equation, sum of forces, should be zero if there is no numerical error in the calculation because of the Newton's third law. The average offset of forces is important for the stability of the time integration of the particle positions. If there is a big offset in the forces, it means the Newton's third law is not satisfied, and the energy in the system tends to increase even for the microcanonical ensemble. The microcanonical ensemble should keep the total energy in the system fixed in the range of the numerical error because no additional energy is introduced in the system. With the 'Exclude on the fly' algorithm, methods B, C, D, F, and G satisfy the sufficiently small offset in forces, while with the 'Exclude afterwards' algorithm, only methods B, C, F, and G satisfy this condition. Note that the proposed method F has no offset in forces, while the method G with full double precision has some offset. This is why the proposed method uses fixed point instead of floating point. Fixed point guarantees the arbitrary order of summation for the same result.

4.4 Performance

Figure 12 shows the calculation speed (number of pairwise interaction per second) of van der Waals forces with 9800GTX and GTX280 GPUs with different types of methods shown in Table 1. Method B is not applicable to 9800GTX because it doesn't support double precision. For this performance benchmark, a relatively large system is calculated without cutoff though usual MD simulation for Lennard Jones particles uses a cutoff to reduce the calculation cost. The reason is to remove as many overheads as possible to measure the speed for the internal of the force calculation loop with different accumulation methods.

The proposed method F is faster than the method B, which is the accumulation with double precision. Method F is slightly slower than Nitadori's method D. These results show that the double precision support of the latest GPU is not so useful for van der Waals calculations since the method F can be executed without double precision.

5 Concluding Remarks

The proposed method (Narumi's trick) has an accuracy equivalent to that of a double precision calculation for the summation part with single-precision hardware. Notably, there are no offset in forces for the proposed method. There is no difference in the accuracy with 'Exclude on the fly' and 'Exclude afterwards' algorithms for the proposed method. This is useful for achieving high performance on the GPU, since the force calculation part becomes simple when the 'Exclude afterwards' algorithm is used. For example, we can divide the force calculation into two parts: two body interaction with GPU and exclusion on CPU. As the exclusion part is not so heavy but a little complicated compared with the two body interaction,

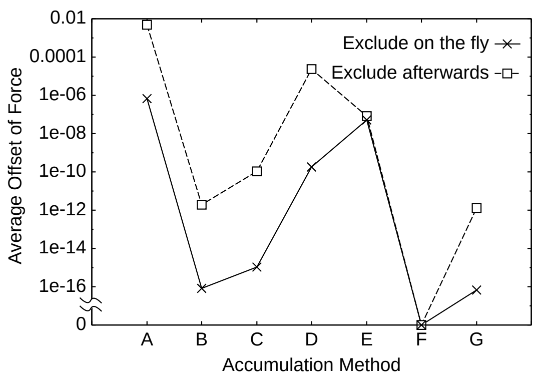

Figure 11. Average offset of forces for System 1 against accumulation methods. Proposed method F has no offset.

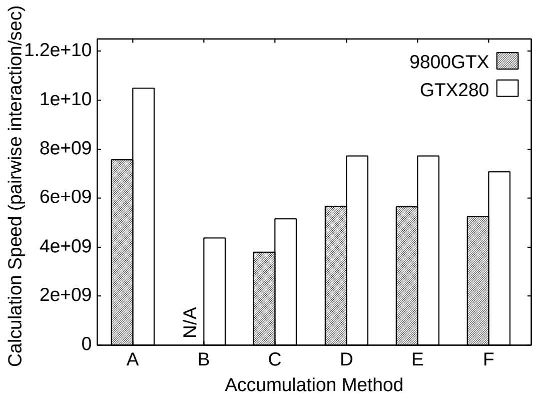

Figure 12. Performance of 9800GTX and GTX280 GPUs for van der Waals force calculation for System 2 with different types of methods. Proposed method F is faster than the method B, accumulation with double precision.

Table 1. Accumulation methods

Method	Pairwise force	Accumulation
A	Single precision	Single precision
B	Single precision	Double precision
C	Single precision	Takahashi & Iitaka's trick
D	Single precision	Nitadori's trick
E	Single precision	Modified Nitadori's trick
F	Single precision	Proposed method (Narumi's trick)
G	Double precision	Double precision

the CPU is suitable for calculating it. The two body interaction part without exclusion is simple and easy to accelerate on the GPU. Therefore, the proposed method would be useful for the actual MD codes on GPUs in the near future in terms of both performance and accuracy.

Acknowledgement

Part of this study was supported by the Core Research for Evolution Science and Technology (CREST) of the Japan Science and Technology Corporation (JST) and by the Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Young Scientists (B), H20, 2070031.

References

[1] D. PHAM, S. ASANO, M. BOLLIGER, M. N. DAY, H. P. HOFSTEE, C. JOHNS, J. KAHLE, A. KAMEYAMA, J. KEATY, Y. MASUBUCHI, M. RILEY, D. SHIPPY, D. STASIAK, M. SUZUOKI, M. WANG, J. WARNOCK, S. WEITZEL, D. WENDEL, T. YAMAZAKI, K. YAZAWA, *The design and implementation of a first-generation CELL processor*, in Proceedings of Solid-State Circuits Conference, 1, (2005), pp 184–592.

[2] `http://www.playstation.com/` (Oct. 2008).

[3] `http://www.nvidia.com/` (Oct. 2008).

[4] `http://www.intel.com/products/ processor/xeon7000/` (Oct. 2008).

[5] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, "Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems", *International Journal of High Performance Computing Applications*, 21, (2007), pp 457-466.

[6] Toru Takahashi, "Hardware Acceleration for Boundary Element Methods", Harvard/Riken GPGPU Symposium 2008, Cambridge, US, (Mar. 2008).

[7] Toru Takahashi and Kazuki Koketsu, "A high performance computing using graphics boards in boundary element methods", in Proceedings of Keisan-Kougaku-Kouenkai, vol. 11, Osaka, Japan, (Jun. 2006; in Japanese).

[8] T. Hamada, "Simulation of Stellar Systems with GPUs", *in Proceedings of the Forum on Advanced Scientific Simulation - Astrophysics*, Fukuoka, Japan, (2008), `http://onokoro.cc.kyushu-u.ac.jp/forum08/ talk/Hamada.pdf` (Oct. 2008; in Japanese).

[9] M. Karplus and J. A. McCammon, "Molecular dynamics simulations of biomolecules", *Nat. Struct. Biol.*, 9, (2002), pp 646-52.

[10] D. VAN DER SPOEL, E. LINDAHL, B. HESS, G. GROENHOF, A. E. MARK, AND H. J. BERENDSEN *Gromacs: fast, flexible, and free*, J. Comput. Chem., 26 (16), (2005), pp 1701–1718.

[11] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors", *Journal of Computational Chemistry*, 28, (2007), pp 2618-2640.

[12] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W. W. Hwu, "GPU acceleration of cutoff pair potentials for molecular modeling Applications", *in Proceedings of the 2008 Conference on Computing Frontiers*, (2008), pp 273-282.

[13] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units", *Journal of Computational Physics*, 227, 10, (2008), pp 5342-5359.

[14] T. Hamada and T. Iitaka, "The Chamomile Scheme: An optimized algorithm for N-body simulations on programmable graphics processing units", *ArXiv Astrophysics e-prints*, astro-ph/073100, (2007).

[15] D. E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, Addison-Wesley (1969).

[16] T. J. Dekker, "A floating-point technique for extending the available precision", *Numer. Math.*, 18 (1971), pp 224-242.

[17] David H. Bailey, Yozo Hida, Karthik Jeyabalan, Xiaoye S. Li, and Brandon Thompson, "High-Precision Software Directory", `http://crd.lbl.gov/˜dhbailey/ mpdist/` (Oct. 2008).

[18] K. Nitadori, "High-accuracy N-body simulations on GPU", *Poster in AstroGPU 2007*, Princeton, (2007).

[19] D. A. Case, T. A. Darden, T. E. Cheatham III, C. L. Simmerling, J. Wang, R. E. Duke, R. Luo, K. M. Merz, B. Wang, D. A. Pearlman, M. Crowley, S. Brozell, V. Tsui, H. Gohlke, J. Mongan, V. Hornak, G. Cui, P. Beroza, C. Schafmeister, J. W. Caldwell, W. S. Ross, P. A. Kollman, Amber 8.0, University of California San Francisco (2004).

[20] T. Narumi, S. Kameoka, M. Taiji, and K. Yasuoka, "Accelerating molecular dynamics simulations on PLAYSTATION 3 using 'Virtual-GRAPE' programming model", *SIAM Journal on Scientific Computing*, in press.