International Journal of Computational Methods
Vol. 8, No. 3 (2011) 561–581

DOI: 10.1142/S0219876211002708

World Scientific
www.worldscientific.com

FAST QUASI DOUBLE-PRECISION METHOD WITH SINGLE-PRECISION HARDWARE TO ACCELERATE SCIENTIFIC APPLICATIONS

TETSU NARUMI

Department of Computer Science
University of Electro-Communications
1-5-1 Chofugaoka, Chofu, Tokyo 182-8585, Japan
narumi@cs.uec.ac.jp

TSUYOSHI HAMADA

Department of Computer and Information Sciences
Nagasaki University, Nagasaki 852-8521, Japan
hamada@cis.nagasaki-u.ac.jp

KEIGO NITADORI

High-Performance Molecular Simulation Team
RIKEN Advanced Science Institute
Yokohama 230-0046, Japan
keigo@riken.jp

RYUJI SAKAMAKI* and KENJI YASUOKA†

Department of Mechanical Engineering
Keio University, Yokohama 223-8522, Japan
**sakamaki@z6.keio.jp*
†*yasuoka@mech.keio.ac.jp*

Received 30 October 2010
Accepted 15 February 2011

The recent commodity hardware, such as the Cell processor in PLAYSTATION 3 or GeForce GTX280 GPU, has much more peak performance in single-precision than that of CPU of PCs, while the double-precision performance of these computers is comparable to it. Even though a quasi double-precision method achieves comparable accuracy to double-precision, the performance of it is relatively low. In this paper, the quasi double-precision method is modified, and it can deliver more performance than that by native double-precision of the GPU. Calculation of a Mandelbrot set and molecular dynamics (MD) simulation are used to test the method.

Keywords: Mixed-precision method; graphics processing unit; molecular dynamics simulation; quasi double-precision method.

1. Introduction

Precision of arithmetic operations in computer simulation is one of the crucial issues to obtain reasonable results. However, during the past decade most simulations have used 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. In this case, the calculation speed of single-precision is twice as fast as that of double-precision.

In the past three or four years, some processors with completely different performances for different precisions have been developed. For example, the Cell processor [Pham *et al.* (2005)] in PLAYSTATION 3 (PS3) [SONY Comp. Entert. Corp. (2010)] has a 179 Gflops single-precision peak performance, and only 17 Gflops in double-precision. GeForce GTX280, one of the latest GPUs from NVIDIA Corp. [2010], has 933 Gflops single-precision performance, and only 78 Gflops in double-precision. Although the single-precision performances of these processors exceed the latest Intel CPU, such as Xeon X7460 (128 Gflops) [Intel Corp. (2010)], the double-precision performance is comparable to the CPU (64 Gflops). To achieve maximum performance on these processors, it is necessary to use as many single-precision arithmetic operations as possible. Actually, most GPU codes use only single-precision [Owens *et al.* (2008); Hwu *et al.* (2009)]. The challenge in using single-precision operations is the reduction of cancellation or round-off errors in the calculation results.

One promising approach to get the merit of the high performance of single-precision hardware is to use a mixed-precision arithmetic operation. It has been used for performing matrix operations [Göddeke *et al.* (2007); Buttari *et al.* (2007)]. In the iterative loop for solving matrix equations, they used single-precision operations for the most computationally heavy part. Other parts are performed with double-precision as higher accuracy is required to assure 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 or the GPU.

However, mixed-precision approach has limitations. Georgescu and Okuda [2009] report that the mixed-precision algorithm does not work well when the condition number of the matrix is very large, such as 10^8. Moreover, there are many algorithms in which mixed-precision algorithm cannot be applied. All the noniterative algorithms do not fit to it.

Another approach to get the merit of the high performance of single-precision hardware for getting further accuracy than single-precision is quasi double-precision method [Göddeke *et al.* (2007); Graça and Defour (2006); Payne and Hitz (2006); Thall (2006)]. In the quasi double-precision method, a pair of single-precision variables is used for expressing a value to increase the accuracy. Over ten times additional operations are needed to calculate an addition, a subtraction, and a

multiplication with the quasi double-precision method. Therefore, even though there is large difference between single- and double-precision performances in GPUs, the native double-precision calculation was comparable to or faster than that by quasi double-precision method [Georgescu and Okuda (2009)]. It means that there is no merit to use quasi double-precision in terms of performance.

In this paper, we propose two modified methods of quasi double-precision that deliver more performance than that by native double-precision hardware on the GPU. One variation is for the calculation of a Mandelbrot set, which is a simple example of using additions, subtractions, and multiplications in the most heavy loop. This technique can be used when the required accuracy is in between single- and double-precisions. Another variation is for molecular dynamics (MD) simulations. This technique can be used when the required dynamic range is not so large.

MD simulation is a powerful technique to investigate the physical property of condensed matter at an atomic level [Karplus and McCammon (2002)]. Atoms in a system interact with each other via the Coulomb, van der Waals, and bonding forces. Among these forces, two-body interactions with Coulomb and van der Waals forces are dominant when the number of atoms in a system becomes large. Therefore, for the large systems, the computational cost of MD simulations can become overwhelming. To enhance the performance of the MD simulation code, single-precision operations are extensively used. For example, GROMACS [Spoel *et al.* (2005)], one of the fastest MD simulation codes with a single thread, uses SSE unit in Intel CPUs to enhance the performance. NAMD [Stone *et al.* (2007); Rodrigues *et al.* (2008)], 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 to calculate single-precision force on GPUs [Stone *et al.* (2007); Rodrigues *et al.* (2008); Anderson *et al.* (2008); van Meel *et al.* (2008); Harvey *et al.* (2009)], the accuracy of the accumulation of forces performed is not 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 cancellation or rounding off error [Hamada and Iitaka (2007)], a more accurate precision should be used. To satisfy the request, quasi double-precision method has been used with GPUs for gravitational N-body simulations [Nitadori (2007); Hamada (2008)]. The method is also used for other applications, such as Mandelbrot set calculation [Thall (2006)], finite element method [Göddeke *et al.* (2007)], and boundary element method [Takahashi (2008); Takahashi and Koketsu (2006)]. Since the current GPU supports native double-precision operations, the quasi double-precision has little meaning if its performance is lower than that by the native double-precision. In this paper, we propose a faster and comparably accurate method for accumulating forces against native double-precision.

The paper is organized as follows. In Sec. 2, quasi double-precision, which had been previously used, is described. In Sec. 3, the proposed method is explained in detail. In Sec. 4, new method is applied to Mandelbrot set calculation, which is a simple example of using additions, subtractions, and multiplications in the most heavy loop. In Sec. 5, a simple MD simulation with van der Waals particles is performed with the new method. Our accumulation method of forces on single-precision hardware showed better performance than that by double-precision hardware and more accuracy compared with that by single-precision calculation. Summary and comments are presented in Sec. 6.

2. Quasi Double-Precision Method

In this section, quasi double-precision method, a well-known method to increase accuracy by using a pair of variables to express one value, is described.

Knuth [1969] and Dekker [1971] proposed a method to sum two values without losing 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:

$$x_S + y_S = z_S + zz_S, \tag{1}$$

where z_S and zz_S are:

$$\begin{aligned} z_S &= fl_S(x_S + y_S), \\ w_S &= fl_S(z_S - x_S), \\ zz_S &= fl_S(y_S - w_S). \end{aligned} \tag{2}$$

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

$$\left| \frac{zz_S}{z_S} \right| < 2^{-n_S}, \tag{3}$$

where $n_S = 24$ is the number of mantissa bits of single-precision defined in the IEEE754 standard.

In the following pages, we call this combination of higher and lower expressions a *pair expression*. Even though the supported dynamic range is narrower, 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.

`add_knuth_and_dekker1` in Fig. 1 shows the program for Eq. (2) in C language. Here, "`SS z`" is a variable for a *pair expression*: `z.hs` $= z_S$ and `z.ls` $= zz_S$. Dekker also proposed a modified method, which does not depend on the condition, $|x_S| \geq |y_S|$, as described `add_knuth_and_dekker2` in Fig. 1. We call these algorithms as Knuth and Dekkter's trick.

By using the trick, addition, subtraction, and multiplication can be written as Fig. 2. We call these routines as "full quasi double-precision" routines since they

```
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;
}

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;
}
```

Fig. 1. Knuth and Dekker's trick in C language to add two single-precision variables without losing accuracy. `SS` holds a pair expression. `add_knuth_and_dekker1` can be used when $|\texttt{xs}| \geq |\texttt{ys}|$. `add_knuth_and_dekker2` can be used for all the cases.

```
SS dsadd_full(SS a, SS b)
{
  float rs;
  SS z, c;

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

  return c;
}

SS dssub_full(SS a, SS b)
{
  SS c;
  float t1 = a.hs - b.hs;
  float e = t1 - a.hs;
  float t2 = ((-b.hs - e) + (a.hs - (t1 - e))) \
                                  + a.ls - b.ls;
  c = add_knuth_and_dekker1(t1, t2);

  return c;
}

SS dsmul_full(SS a, SS b)
{
  SS sa, sb, c, t, d;
  float cona, conb, c2;

  cona  = a.hs * 8193.0f;
  conb  = b.hs * 8193.0f;
  sa.hs = cona - (cona - a.hs);
  sb.hs = conb - (conb - b.hs);
```

Fig. 2. Addition, subtraction, and multiplication with a pair expression using Knuth and Dekker's trick. `__fmul_rn(a,b)` is a special command for multiplication (`a * b`) with NVIDIA's GPU to prevent from fusing addition and multiplication. These routines are basically the same as DSFUN90 library [Bailey *et al.* (2010)], which are used in an NVIDIA's sample program `Mandelbrot`.

```
34    sa.ls = a.hs - sa.hs;
35    sb.ls = b.hs - sb.hs;
36    c.hs  = __fmul_rn(a.hs, b.hs);
37    c.ls  = (((sa.hs * sb.hs - c.hs) + sa.hs * sb.ls) \
38                        + sa.ls * sb.hs) + sa.ls * sb.ls;
39
40    c2 = __fmul_rn(a.hs, b.ls) + __fmul_rn(a.ls, b.hs);
41    t  = add_knuth_and_dekker2(c.hs, c2);
42    t.ls += c.ls + __fmul_rn(a.ls, b.ls);
43
44    d     = add_knuth_and_dekker1(t.hs, t.ls);
45
46    return d;
47  }
```

Fig. 2. (*Contuinued*)

guarantee the maximum accuracy of a *pair expression* with two single-precision variables. Similar routines are used in a DSFUN90 library [Bailey *et al.* (2010)], as well as an NVIDIA's sample program, `Mandelbrot`. `__fmul_rn(a,b)` is a special command for multiplication (`a * b`) with NVIDIA's GPU to prevent from fusing addition and multiplication. When fusing occurs, the accuracy of multiplication differs from the IEEE754 standard.

3. Fast Quasi Double-Precision Method

In this section, fast quasi double-precision method is newly described.

3.1. *Fast addition, subtraction and multiplication*

Figure 3 shows the simplified routine for addition, subtraction, and multiplication to perform quasi double-precision operations faster. The differences between the full quasi double-precision routines (Fig. 2) are summarized in Tables 1 and 2.

The first idea of simplifying the quasi double-precision method is to skip the normalization step at the last stage. For example, `add_knuth_and_dekker1` is skipped in the `dsadd_F1`. This is because the normalization step is not so sensitive to accuracy. We can skip the normalization step with the cost of some accuracy.

The second step is to use `add_knuth_and_dekker1` instead of `add_knuth_and_dekker2` assuming that the first operand of the routine is larger than the second operand. To ensure this for addition and subtraction, we add a large value to the first operand beforehand, then subtract it afterwards. Details of the procedure will be presented in Sec. 4.1.

In the case of multiplication, some reduction of operations can be done without losing accuracy. First, `add_knuth_and_dekker2` (line 41 of Fig. 2) of `dsmul_full` can be changed to `add_knuth_and_dekker1` (line 58 of Fig. 3) of `dsmul_F12` because the first operand is bigger than the second operand. Moreover, the operation of extracting the preceding 12 bits from a single-precision variables (lines 30–33 of Fig. 2) in `dsmul_full` can be simplified as in lines 48–49 of Fig. 3 using NVIDIA's special commands (`__int_as_float` and `__float_as_int`).

To further accelerate quasi double-precision multiplication, we reduce the number of mantissa bits for multiplication from 48 to 36 bits. This actually reduces the

```
SS dsadd_F1(SS a, SS b)
{
  SS c;

  c     = add_knuth_and_dekker2(a.hs, b.hs);
  c.ls += a.ls + b.ls;

  return c;
}

SS dsadd_F23(SS a, SS b)
{
  SS c;

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

  return c;
}

SS dssub_F1(SS a, SS b)
{
  SS c;

  c.hs = a.hs - b.hs;
  float e = c.hs - a.hs;
  c.ls = ((-b.hs - e) + (a.hs - (c.hs - e))) \
                                + a.ls - b.ls;
  return c;
}

SS dssub_F23(SS a, SS b)
{
  SS c;

  c.hs = a.hs - b.hs;
  float e = c.hs - a.hs;
  c.ls = (-b.hs - e) + a.ls - b.ls;

  return c;
}

SS dsmul_F12(SS a, SS b)
{
  SS sa, sb, c, t, d;
  float c2;
  sa.hs = __int_as_float(__float_as_int(a.hs) \
                                        & 0xfffff000);
  sb.hs = __int_as_float(__float_as_int(b.hs) \
                                        & 0xfffff000);
  sa.ls = a.hs - sa.hs;
  sb.ls = b.hs - sb.hs;
  c.hs  = __fmul_rn(a.hs, b.hs);
  c.ls  = (((sa.hs * sb.hs - c.hs) + sa.hs * sb.ls) \
                    + sa.ls * sb.hs) + sa.ls * sb.ls;
  c2 = __fmul_rn(a.hs, b.ls) + __fmul_rn(a.ls, b.hs);
  c.ls  = c.ls + c2 + __fmul_rn(a.ls, b.ls);
  d     = add_knuth_and_dekker1(c.hs, c.ls);

  return d;
}

SS dsmul_F3(SS a, SS b)
{
  SS sa, sb, t, d;
  sa.hs = __int_as_float(__float_as_int(a.hs) \
                                        & 0xfffff000);
  sb.hs = __int_as_float(__float_as_int(b.hs) \
                                        & 0xfffff000);
  sa.ls = (a.hs - sa.hs)+a.ls;
  sb.ls = (b.hs - sb.hs)+b.ls;
  t.hs = sa.hs * sb.hs;
  t.ls = sa.hs * sb.ls + sb.hs * sa.ls;
  d    = add_knuth_and_dekker1(t.hs, t.ls);
  d.ls = d.ls + sa.ls * sb.ls;

  return d;
}
```

Fig. 3. Proposed routines for addition, subtraction, and multiplication, which are simplified from Fig. 2. The numbers of operations are summarized in Tables 1 and 2.

Table 1. Number of operations for addition and subtraction.

Subroutine	Operation	Normalization of the last stage	First operand must be larger than the second operand	Number of operations
`dsadd_full`	+	yes	no	11
`dsadd_F1`	+	no	no	8
`dsadd_F23`	+	no	yes	5
`dssub_full`	−	yes	no	11
`dssub_F1`	−	no	no	8
`dssub_F23`	−	no	yes	5

Table 2. Number of operations for multiplication.

Subroutine	Operation	Reduction of operations with the same accuracy	Normalization at the last stage	Reduction of number of mantissa bits	Number of operations
`dsmul_full`	×	no	yes	no	32
`dsmul_F12`	×	yes	no	no	22
`dsmul_F3`	×	yes	no	yes	15

accuracy around three orders of magnitude, but the number of operations needed to one quasi double-precision multiplication becomes about half of the original method as described in Table 2.

3.2. *Fast accumulation*

In this section, fast quasi double-precision method for accumulation is described. In the MD simulation, the single-precision calculation of pairwise force is accurate enough [Amisaki *et al.* (1995); Narumi *et al.* (2008)]. However, the accumulation of each force should be carefully performed because cancellation or rounding off error occurs when just single-precision is used for summing up forces as described in Sec. 1. The target problem is to sum up single-precision values into an accumulated value. Using quasi double-precision is useful to prevent from the cancellation and rounding off errors. Here, we assume that we already have an accumulated value by a *pair expression*.

In the following paragraphs, we first explain two previous methods for accumulation. Then, we propose a new method for it.

3.2.1. *Full quasi double-precision*

The full implementation of the quasi double-precision addition `dsadd_full` was used for accumulation with GPU [Graça and Defour (2006); Takahashi (2008); Takahashi and Koketsu (2006)]. `b.ls=0` is assumed in `dsadd_full` in Fig. 2 because pairwise force is calculated with a single-precision variable.

3.2.2. *Nitadori's trick*

We have already proposed a more simplified method, Nitadori's trick [Nitadori (2007); Nitadori (2009)], where only four operations are necessary per accumulation, while full quasi double-precision method requires ten operations. The algorithm of Nitadori's trick is similar to `dsadd_F23`, but `b.ls=0` is also assumed. To ensure the Nitadori's trick work properly, we assume two conditions. The first condition is that the accumulated value should be larger than a value that is going to be added, so that `add_knuth_and_dekker1` can be used instead of `add_knuth_and_dekker2`. The other condition is that the target application should not be sensitive to the small decrease in accuracy in a *pair expression* because it skips the normalization step. 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 numbers of values are added, much larger error could be generated by this method. However, in the collision-less N-body simulation, the error does not seem to be a problem [Hamada (2008)]. For other gravitational N-body simulations, detailed analysis will be presented elsewhere. In the MD simulation, this error has a notable effect in some cases.

3.2.3. *Narumi's trick*

In this paper, we propose a more accurate method based on Nitadori's trick, which we call Narumi's trick. The idea behind it is to use fixed-point numbers instead of floating-point numbers. In MD simulation, the required dynamic range of each pairwise force is not so large. There is a certain limit of the size of a force that an atom receives because it basically vibrates on the balanced position. Too small size of a force does not cause an error because it almost vanishes at the time integration operation of atom positions. Therefore, the fixed-point-based algorithm can be used for this part.

Figure 4 shows the routines for Narumi's trick. First, quasi double-precision structure, `SI`, is defined. This structure houses a single-precision value (`float hs`) and a 32-bit integer (`int li`). `hs` and `li` correspond to the higher and lower parts of a *pair expression*, respectively. `LOWER_FACTOR` is the factor to be multiplied to a floating-point value to be converted into 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 to 0 at the most significant bits (MSBs) of the lower part of a *pair expression* (`li`). The preceding zeros guarantee the nonoverflow of `li` even when `add_narumi` is called 128 (=`LOWER_LOOP` = `1 << LOWER_SHIFT`) times. The reason for using a 32-bit integer instead of single-precision is that a 32-bit integer has 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 parts of the *pair expression* is changed, the accuracy of accumulation is not deteriorated as in the Nitadori's trick.

```
#define LARGE_SHIFT    15
#define LARGE          (float)(3 << (LARGE_SHIFT-1))
#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)
typedef struct {
    float hs;
    int   li;
} SI;

SI initialize_narumi(void)
{
  SI a;

  a.hs = LARGE;
  a.li = 0;

  return a;
}

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;
}

SI update_narumi(SI a)
{
  SI b;

  b.hs = a.hs + (float)(a.li & (MASK(LOWER_SHIFT) \
              << (32-LOWER_SHIFT))) * LOWER_FACTOR_1;
  b.li = a.li & MASK(32-LOWER_SHIFT);

  return b;
}

double copy_narumi(SI a)
{
  return (a.hs-LARGE) + a.li * (double)LOWER_FACTOR_1;
}
```

Fig. 4. Proposed method (Narumi's trick) for accumulation. `LARGE` must be set to a large enough number to satisfy $|$`a.hs`$| \geq |$`ys`$|$ in `add_narumi`, i.e., each pairwise force should be smaller than `(1<<LARGE_SHIFT)` at least.

Figure 5 shows the block diagram of the *pair expression* of Narumi's trick. The reason `LARGE` is set to `3<<14` instead of `1<<15` is that the position of a decimal point and the sign are fixed for $|$`fs`$[$`k`$]| <$`(1<<14)`. With this method, the decimal points of the higher and lower parts of the *pair expression* are fixed, which means that the *pair expression* is equivalent to a 48-bit integer as shown in Fig. 5. The accuracy of the *pair expression* is $2^{-48} \simeq 10^{-15}$, and it is the same as that of full quasi double-precision method. Note that the accuracy becomes lower when `LARGE` is too large compared to each force.

An example of using Narumi's trick in calculating van der Waals force between atoms is described in Fig. 6. Within four routines in Fig. 4, `copy_narumi` must be run at CPU, while other three routines can be run with a single-precision GPU.

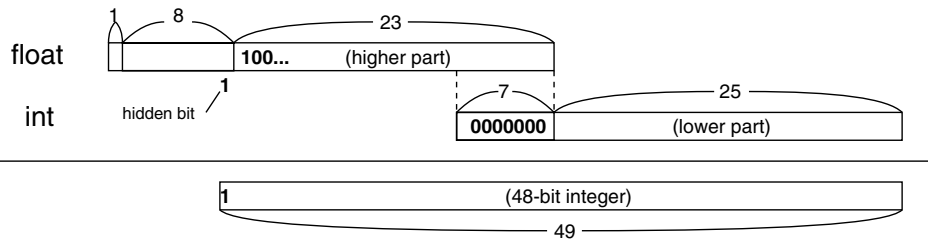

Fig. 5. Block diagram of a *pair expression* in Narumi's trick. A `float` and an `int` are used, and they correspond to a 48-bit integer.

```
 1  void pairwiseforce(float ris[3], float rjs[3],
 2                     float fs[3])
 3  {
 4    int k;
 5    float ds[3], rs, qs, ps;
 6
 7    for(k=0; k<3; k++) ds[k] = ris[k] - rjs[k];
 8    rs = ds[0]*ds[0] + ds[1]*ds[1] + ds[2]*ds[2];
 9    if(rs >= 0.25f){
10      rs = 1.0f / rs;
11      qs = rs * rs * rs;
12      ps = rs * qs * (2.0f * qs - 1.0f);
13      for(k=0; k<3; k++) fs[k] = ps * ds[k];
14    }
15    else{
16      for(k=0; k<3; k++) fs[k] = 0.0f;
17    }
18  }
19
20  void calcforce_one(float ri[3],
21             float rj[][3], int nj, SI a[3])
22  {
23    int j, jj, k;
24    float fs[3];
25
26    for(k=0; k<3; k++)     a[k] = initialize_narumi();
27    for(jj=0; jj<nj; jj+=LOWER_LOOP){
28      for(j=jj ;j-jj<LOWER_LOOP && j<nj; j++){
29        pairwiseforce(ri, rj[j], fs);
30        for(k=0;k<3;k++) a[k] = add_narumi(a[k], fs[k]);
31      }
32      for(k=0;k<3;k++)   a[k] = update_narumi(a[k]);
33    }
34  }
35
36  void calcforce(int ni, int nj, float ri[][3],
37             float rj[][3], double fd[][3])
38  {
39    int i, k;
40    SI (*a)[3]=(SI (*)[3])malloc(sizeof(SI)*ni*3);
41
42    for(i=0; i<ni; i++){
43      calcforce_one(ri[i], rj, nj, a[i]);
44    }
45    for(i=0; i<ni; i++){
46      for(k=0;k<3;k++) fd[i][k] = copy_narumi(a[i][k]);
47    }
48
49    free(a);
50  }
```

Fig. 6. Example code of using proposed method (Narumi's trick) for calculating van der Waals force.

As shown in line 32 of Fig. 6, `update_narumi()` is called every `LOWER_LOOP`. Therefore, the operations of `update_narumi` are negligible. Moreover, the multiplication in line 31 in Fig. 4 can be eliminated if pairwise force is already multiplied by `LOWER_FACTOR`. Therefore, the number of operations needed for Narumi's trick is five including conversion from float to int in line 31 of Fig. 4.

4. Mandelbrot Set Calculation

In this section, numerical error analysis and performance of calculation of Mandelbrot set are described as well as a basic equation and subroutines.

4.1. *Basic equation and subroutine*

The Mandelbrot set is a set that satisfies $|Z_\infty| < 2$, where Z_n is defined as:

$$Z_{n+1} = Z_n^2 + C, \quad Z_0 = 0. \tag{4}$$

Z_n and C are complex numbers. Figure 7 is the snapshot of the Mandelbrot set.

The most inner loop of the Mandelbrot set calculation with a *pair expression* is shown in Fig. 8. `CalcMandelbrot_Q` uses full quasi double-precision method listed in Fig. 2, while `CalcMandelbrot_F1`, `CalcMandelbrot_F2`, and `CalcMandelbrot_F3` use fast quasi double-precision method listed in Fig. 3. The name of routines and the number of operations for the most inner loop are summarized in Table 3.

In `CalcMandelbrot_F2` and `CalcMandelbrot_F3`, `OFFSET` value is added before addition and subtraction to ensure that the first operand is larger than the second operand in order to use `add_knuth_and_dekker1` instead of `add_knuth_and_dekker2`. Then, `OFFSET` is subtracted after addition or subtraction. `OFFSET` is set 3.0 in the paper. In `CalcMandelbrot_F3`, the multiplication is performed with lower accuracy to enhance the performance.

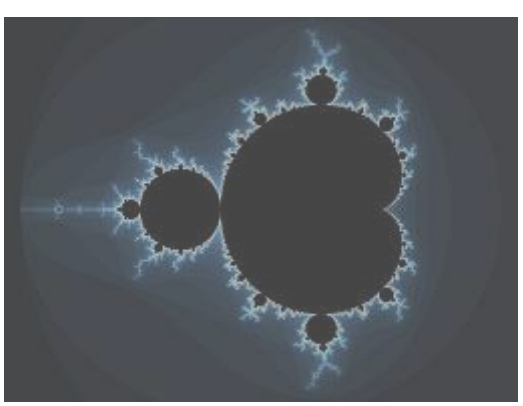

Fig. 7. Screenshot of Mandelbrot set sample program without any modification from the original code by NVIDIA (`Mandelbrot`).

```
 1  int CalcMandelbrot_Q(SS xPos, SS yPos, int crunch)
 2  {
 3    SS xx, yy, sum;
 4    int i = crunch;
 5    SS x = xPos;
 6    SS y = yPos;
 7
 8    yy  = dsmul_full(y, y);     // yy = y * y
 9    xx  = dsmul_full(x, x);     // xx = x * x
10    sum = dsadd_full(xx, yy);   // sum = xx + yy
11    while (--i && (sum.hs + sum.ls < 4.0f)) {
12      y = dsmul_full(x, y);     // y = x*y*2.0f + yPos
13      y = dsadd_full(y, y);
14      y = dsadd_full(y, yPos);
15
16      x = dssub_full(xx, yy);   //  x = xx - yy + xPos
17      x = dsadd_full(x, xPos);
18
19      yy  = dsmul_full(y, y);   // yy = y * y
20      xx  = dsmul_full(x, x);   // xx = x * x
21      sum = dsadd_full(xx, yy); // sum = xx + yy
22    }
23
24    return i;
25  }
26
27  int CalcMandelbrot_F1(SS xPos, SS yPos, int crunch)
28  {
29    SS xx, yy, sum;
30    int i = crunch;
31    SS x = xPos;
32    SS y = yPos;
33
34    yy  = dsmul_full(y, y);     // yy = y * y
35    xx  = dsmul_full(x, x);     // xx = x * x
36    sum = dsadd_full(xx, yy);   // sum = xx + yy
37    while (--i && (sum.hs + sum.ls < 4.0f)) {
38      y = dsmul_F12(x, y);      // y = x*y*2.0f + yPos
39      y.hs *= 2.0f; y.ls *= 2.0f;
40      y = dsadd_F1(y, yPos);
41
42      x = dssub_F1(xx, yy);     //  x = xx - yy + xPos
43      x = dsadd_F1(x, xPos);
44
45      yy  = dsmul_F12(y, y);    // yy = y * y
46      xx  = dsmul_F12(x, x);    // xx = x * x
47      sum = dsadd_F1(xx, yy);   // sum = xx + yy
48    }
49
50    return i;
51  }
52
53  int CalcMandelbrot_Fx(SS xPos, SS yPos, int crunch)
54  {
55    SS xx, yy, xPos2, yPos2, offset={OFFSET, 0.0f};
56    int i = crunch;
57    SS x = xPos;
58    SS y = yPos;
59
60    yPos2 = dsadd_F23(offset, yPos);//yPos2=OFFSET+yPos
61    xPos2 = dsadd_F23(offset, xPos);//xPos2=OFFSET+xPos
62
63    yy  = dsmul_full(y, y);     // yy = y * y
64    xx  = dsmul_full(x, x);     // xx = x * x
65    while (--i && (xx.hs + yy.hs < 4.0f)) {
66      y = dsmul_Fx(x, y);    // y=yPos2+x*y*2.0f-OFFSET
67      y.hs *= 2.0f; y.ls *= 2.0f;
68      y = dsadd_F23(yPos2, y); y.hs -= OFFSET;
69
```

Fig. 8. Programs for Mandelbrot set calculation. `Mandelbrot_F2` and `Mandelbrot_F3` are expressed with the same subroutine named as `Mandelbrot_Fx`. The difference of then is the subroutines for multiplication which is (`dsmul_Fx`) listed in Table 3. Number of operations in a while loop (lines 12–21, 38–47, and 66–74) is summarized in Table 3. `OFFSET` is set 3.0 in the paper.

```
70      x = dsadd_F23(xPos2, xx); // x=xPos2+xx-yy-OFFSET
71      x = dssub_F23(x, yy); x.hs -= OFFSET;
72
73      yy  = dsmul_Fx(y, y);     // yy = y * y
74      xx  = dsmul_Fx(x, x);     // xx = x * x
75    }
76
77    return i;
78  }
```

Fig. 8. (*Continued*)

Table 3. Mandelbrot calculation routine.

Subroutine	Subroutine for addition	Subroutine for subtraction	Subroutine for multiplication	Number of operations in a loop
`Mandelbrot_Q`	`dsadd_full`	`dssub_full`	`dsmul_full`	151
`Mandelbrot_F1`	`dsadd_F1`	`dssub_F1`	`dsmul_F12`	92
`Mandelbrot_F2`	`dsadd_F23`	`dssub_F23`	`dsmul_F12`	85
`Mandelbrot_F3`	`dsadd_F23`	`dssub_F23`	`dsmul_F3`	64

4.2. *Accuracy*

In Fig. 9 the average relative error, S_{err}, of Mandelbrot calculation is plotted against the calculation kernel described in Table 4. Here, S_{err} is the average error over pixels described as:

$$S_{err} = \left\langle \left| \frac{S_{\text{method}} - S_D}{S_D} \right| \right\rangle , \tag{5}$$

where S_{method} is $|Z_{100}|$ with the specified method, and S_D is $|Z_{100}|$ with double-precision calculation. Therefore, `crunch` in Fig. 8 is set to 100 and no comparison against `4.0f` in the `while` statement is performed for accuracy calculation. The number of pixels for averaging is 3.1×10^5 since the window size is 640×480. The accuracy of Kernel F1 is comparable to that of Kernel Q, which uses full quasi double-precision method. The accuracy of Kernel F2 is a little worse than that of Kernel F1. The accuracy of Kernel F3 is worse against F2 around three orders of magnitude as it reduces 12-bit of mantissa for internal multiplication. The accuracy is just in the middle of single- and double-precision. Figure 10 compares the image by single-precision (Kernel S) and fast quasi double-precision (Kernel F3). Kernel F3 produces a sharper image than that by single-precision.

4.3. *Performance*

Figure 11 compares the performance of the kernels. Kernels F2 and F3 are faster than that of double-precision (Kernel D). Therefore, a single-precision GPU is enough for this calculation if required accuracy is enough like Fig. 10. Especially, proposed Kernel F3 is about 50% faster than native double-precision (Kernel D) though its accuracy is just in the middle of single- and double-precision.

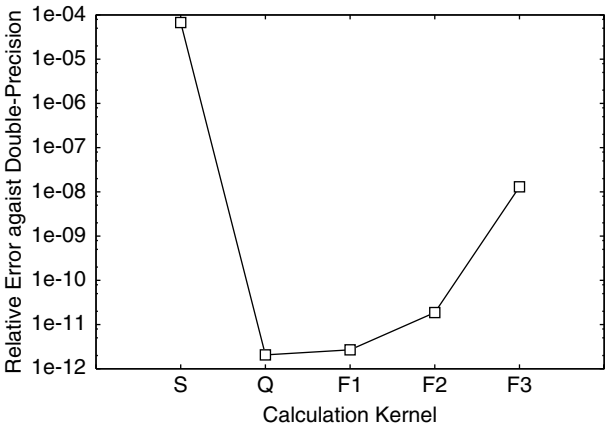

Fig. 9. Average relative error against double-precision for Mandelbrot calculation. Kernels are described in Table 4. The accuracy of the proposed kernel F3 is just in the middle of single- and double-precision.

Table 4. Calculation kernel. Kernel Q, F1, F2, and F3 are described in Fig. 8 and Table 3.

Kernel	Precision of hardware	Description
S	Single	All operations are performed with single-precision
D	Double	All operations are performed with double-precision
Q	Single	Full quasi double-precision method is used
F1	Single	Simple optimization on quasi double-precision method
F2	Single	Reduction of accuracy on additions
F3	Single	Reduction of accuracy on additions and multiplications

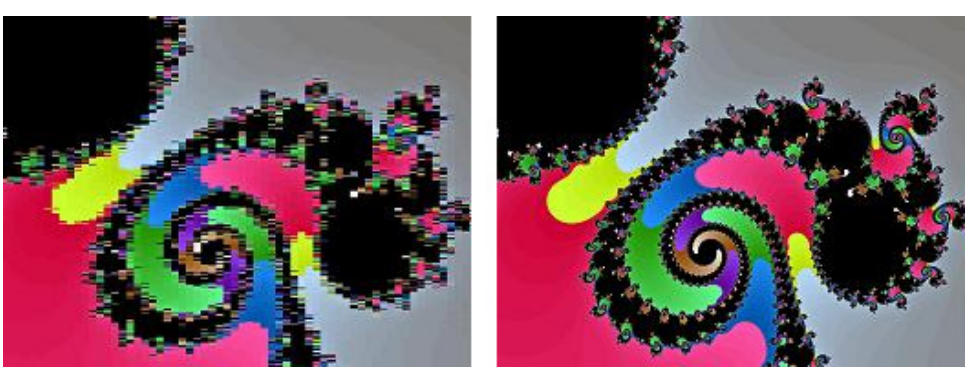

Fig. 10. Comparison of images generated with all single-precision (method S; left) and proposed methods (method F3; right). Other methods (D, Q, F1, and F2) produce similar images as method F3. Initial position is `(xOff, yOff)` $= (-0.7677000147655,\ 0.09478599420880)$. 3.2×10^{12} times close-up image against Fig. 7.

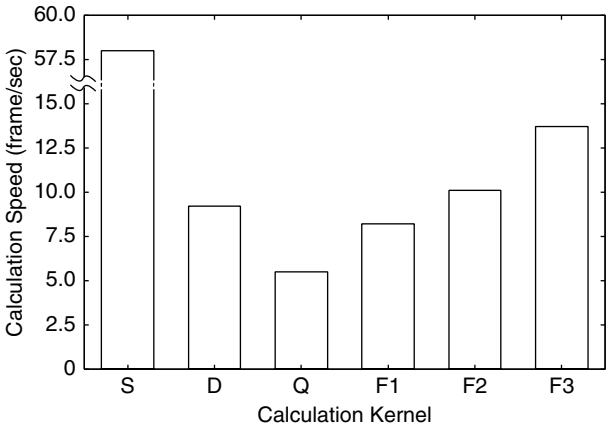

Fig. 11. Performance of GTX280 GPU for Mandelbrot calculation with different kernels described in Table 4. Proposed kernel F3 is 50% faster than kernel D, native double-precision calculation.

5. MD Simulation

In this section, numerical error analysis of Narumi's trick for van der Waals force calculation in MD simulation is presented, followed by the description of its performance.

Two systems are used to test the proposed method numerically. One is a small system with 1,024 particles (System 1), which is used to test accuracy. The other is a bigger system with 65,536 particles (System 2), which is used to achieve 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. Particle positions are defined randomly for both systems. System 1 has only an atom type, while System 2 has 16 atom types, which 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.

5.1. *Accuracy*

5.1.1. *Average error in force*

Figure 12 shows the average error of forces for the various methods. Methods are summarized in Table 5. The average error of forces is calculated as follows:

$$f_{\mathrm{err}} = \frac{\sum_{i=1}^{N} |\vec{F}_{\mathrm{method}_i} - \vec{F}_{\mathrm{double}_i}|}{\sum_{i=1}^{N} |\vec{F}_{\mathrm{double}_i}|}, \tag{6}$$

where $\vec{F}_{\mathrm{double}_i}$ is the force on the ith particle with double-precision calculation (method C), $\vec{F}_{\mathrm{method}_i}$ is the force of a specified method, and N is the total number of particles in the system. As shown in Fig. 12, method D produces reasonably low

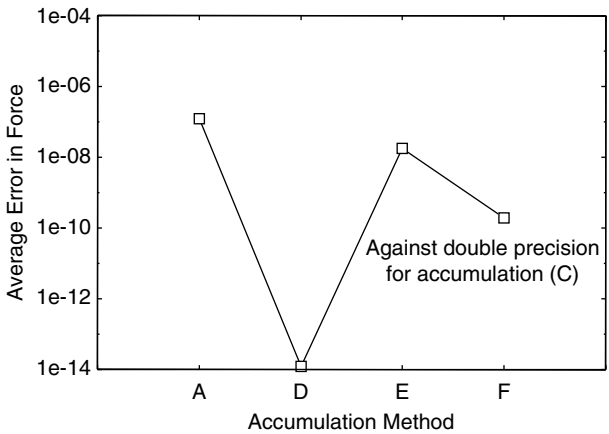

Fig. 12. Average error in force against accumulation with double-precision (method C) is plotted for different accumulation methods, which are summarized in Table 5.

Table 5. Accumulation methods.

Method	Pairwise force	Accumulation	Number of operations for accumulation	9800GTX	GTX280
A	Single-precision	Single-precision	1	○	○
B	Double-precision	Double-precision	1	×	○
C	Single-precision	Double-precision	1	×	○
D	Single-precision	Full quasi double-precision	10	○	○
E	Single-precision	Nitadori's trick	4	○	○
F	Single-precision	Narumi's trick	5	○	○

error as expected. The point is that method F is much more accurate than method E only with one additional operation.

5.1.2. *Average offset in force*

Figure 13 shows the average offset of forces against various methods. Methods are summarized in Table 5. Average offset of forces is calculated as follows:

$$f_{\text{offset}} = \frac{|\sum_{i=1}^{N} \vec{F}_{\text{method}_i}|}{\sum_{i=1}^{N} |\vec{F}_{\text{method}_i}|}. \tag{7}$$

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 that 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

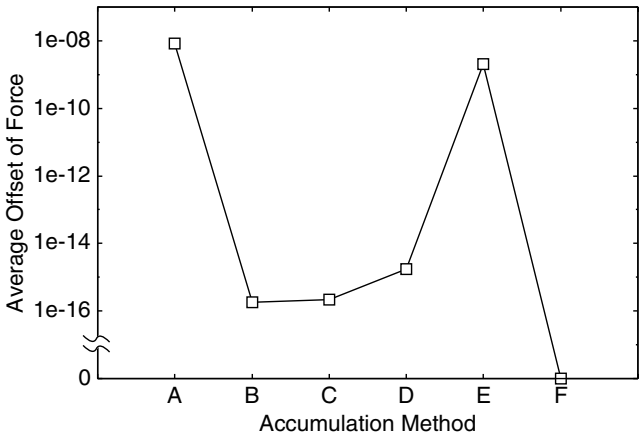

Fig. 13. Average offset of forces for System 1 against accumulation methods. Proposed method F has no offset.

in the system fixed in the range of the numerical error for no additional energy is introduced in the system. Methods B, C, D, and F satisfy the sufficiently small offset in forces. Note that the proposed method F has no offset in forces, while the method B with full double-precision has some offset. This is why the proposed method uses fixed-point numbers instead of floating-point numbers. Fixed-point numbers guarantee the arbitrary order of summation for the same result.

5.2. *Performance*

Figure 14 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 5. Methods B and C are not applicable to 9800GTX because it does not 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 performance of method A, with all single-precision, is comparable to the other literature [van Meel *et al.* (2008)]. They calculated the Coulomb and van der Waals forces between similar number of particles without cutoff. Their calculation time is 0.75 s/step, while ours is 0.56 s/step. Consider that they calculated around 20% of additional operations for the Coulomb force and used 15% slower GPU card, these timings are almost equivalent.

From Fig. 14, the proposed method F is 60% faster than the method C, which is the accumulation with double-precision. Method F is slightly slower than Nitadori's

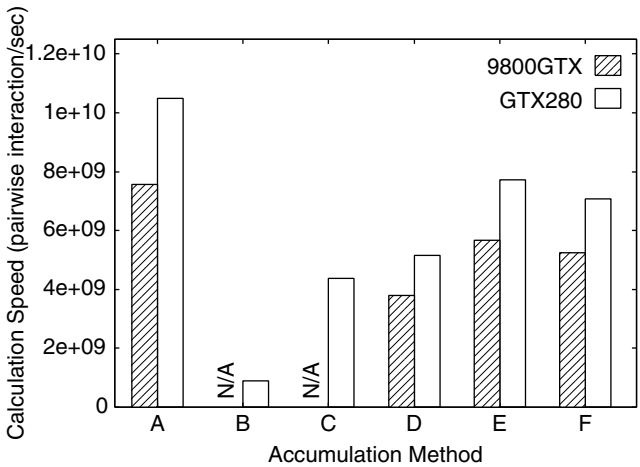

Fig. 14. Performance of 9800GTX and GTX280 GPUs for van der Waals force calculation for System 2 with different types of methods. Proposed method F is 60% faster than the method C, accumulation with double-precision.

method E. These results show that the support of double-precision on the latest GPU is not so useful for van der Waals calculations since the method F can be executed without double-precision.

6. Concluding Remarks

We proposed a simplified method of quasi double-precision for addition, subtraction, and multiplication. We can get faster operation with the method with the cost of accuracy when calculating Mandelbrot set with a GPU.

For MD simulation, 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 is no offset in forces for the proposed method.

This algorithm would also be useful for other processors, such as ATI's GPU or a Cell processor, because both have large differences in the speed of single- and double-precision operations. However, the effectiveness of the fast quasi double-precision method might be lower with the ATI's GPU, since it has higher ratio of double-precision performance compared with NVIDIA's. The Narumi's trick can be easily implemented to a Cell processor even though it does not support "round to nearest" rounding-mode because Narumi's trick is basically a fixed-point method. The single-precision operation with the Cell processor supports only "round to zero" mode, which is not compatible with the most common rounding mode of IEEE754 standard. This drawback of the Cell processor is not so influential to Narumi's trick.

References

Amisaki, T. *et al.* [1995] Error evaluation in the design of a special-purpose processor that calculates nonbonded forces in molecular dynamics simulations, *J. Comput. Chem.* **16**(9), 1120–1130.

Anderson, J. A., Lorenz, C. D. and Travesset, A. [2008] General purpose molecular dynamics simulations fully implemented on graphics processing units, *J. Comput. Phys.* **227**(10), 5342–5359.

Bailey, D. H. *et al.* [2010] High-Precision Software Directory, http://crd.lbl.gov/~dhbailey/mpdist/.

Buttari, A. *et al.* [2007] Mixed precision iterative refinement techniques for the solution of dense linear systems, *Int. J. High Perform. Comput. Appl.* **21**, 457–466.

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

Georgescu, S. and Okuda, H. [2009] Mixed precision in Krylov Solvers on GPU, *Proc. Comput. Eng. Conf.*, Tokyo, Japan, **14**, 281–282.

Göddeke, D., Strzodka, R. and Turek, S. [2007] Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, *Int. J. Parallel Emergent Distrib. Syst.* **22**(4), 221–256.

Graça, G. D. and Defour, D. [2006] Implementation of float–float operators on graphics hardware, *In RNC7: 7th Conference on Real Numbers and Computers*, pp. 23–32.

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

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

Harvey, M. J., Giupponi, G. and De Fabritiis, G. [2009] ACEMD: Accelerating biomolecular dynamics in the microsecond time scale, *J. Chem. Theory Comput.* **5**(6), 1632–1639.

Hwu, W.-M. *et al.* [2009] Compute unified device architecture application suitability, *IEEE Comput. Sci. Eng.* **11**(3), 16–26.

Intel Corp. [2010] http://www.intel.com/products/processor/xeon7000/.

Karplus, M. and McCammon, J. A. [2002] Molecular dynamics simulations of biomolecules, *Nat. Struct. Biol.* **9**, 646–652.

Knuth, D. E. [1969] *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA.

Narumi, T. *et al.* [2008] Accelerating molecular dynamics simulations on PLAYSTATION 3 using 'Virtual-GRAPE' programming model, *SIAM J. Sci. Comput.* **30**(6), 3108–3125.

Nitadori, K. [2007] *High-Accuracy N-body Simulations on GPU*, Poster in AstroGPU 2007, Princeton, USA.

Nitadori, K. [2009] New approaches to high-performance N-body simulations — high-order integrator, new parallel algorithm, and efficient use of SIMD hardware, *Doctor's Thesis*, University of Tokyo, Japan.

NVIDIA Corp. [2010] http://www.nvidia.com/.

Owens, J. D. *et al.* [2008] GPU computing, *Proc. IEEE* **96**(5), 879–899.

Payne, B. R. and Hitz, M. A. [2006] Implementation of residue number systems on GPUs, *In ACM SIGGRAPH 2006 Research posters*, New York, NY, p. 57.

Pham, D. *et al.* [2005] The design and implementation of a first-generation CELL processor, *Proc. Solid-State Circuits Conf.* **1**, 184–592.

Rodrigues, C. I. *et al.* [2008] GPU acceleration of cutoff pair potentials for molecular modeling applications, *Proceedings of the 2008 Conference on Computing Frontiers*, pp. 273–282.

SONY Computer Entertainment Inc. [2010] http://www.playstation.com/.

Spoel, D. V. D. *et al.* [2005] Gromacs: fast, flexible, and free, *J. Comput. Chem.* **26**(16), 1701–1718.

Stone, J. E. *et al.* [2007] Accelerating molecular modeling applications with graphics processors, *J. Comput. Chem.* **28**, 2618–2640.

Takahashi, T. and Koketsu, K. [2006] A high performance computing using graphics boards in boundary element methods, *Proceedings of Keisan-Kougaku-Kouenkai*, 11, Osaka, Japan.

Takahashi, T. [2008] Hardware acceleration for boundary element methods, *Harvard/Riken GPGPU Symposium 2008*, Cambridge, US.

Thall, A. [2006] Extended-precision floating-point numbers for GPU computation, *In ACM SIGGRAPH 2006 Research posters*, New York, NY, p. 52.

van Meel, J. A. *et al.* [2008] Harvesting graphics power for MD simulations, *Mol. Simulat.* **34**(3), 259–266.