

# Molecular Dynamics and Integral Equation on GPU

Toshiaki IITAKA

RIKEN (The Institute of Physical and Chemical Research)  
Wako, Saitama 351-0198, Japan, 20051211  
tiitaka@riken.jp,  
WWW home page: <http://www.iitaka.org/>

**Abstract.** We report our experience of accelerating numerical calculations by Graphic Processing Unit (GPU) for the case of Molecular Dynamics simulation. The sustained performance of one GPU chip (GForce7800GTX) was found to be about 15 Gflops or 70 times of CPU calculation. This performance is compared with that of chips specially designed for molecular dynamics calculation. We show also how the algorithm for molecular dynamics calculation can be applied to solving integral equations.

## 1 Introduction

Molecular Dynamics (MD) [1] is a simulation method for studying the physical properties of  $N$  particles by solving numerically the equation of motion in classical dynamics or the Newton's equation (1) with the position and velocity of the particles at time  $t = 0$ .

$$m_i \frac{d^2 \mathbf{r}_i}{dt^2} = \mathbf{F}_i \quad (1)$$

where  $m_i$  and  $\mathbf{r}_i$  are mass and position of the  $i$ -th particle where  $i = 1, 2, \dots, N$ .  $\mathbf{F}_i$  is the total force acting on the  $i$ -particle, which is the sum of  $\mathbf{f}_{ij}$ , the force acting on the  $i$ -th particle from  $j$ -th particle,

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{f}_{ij}. \quad (2)$$

In the system of charged point particles, the relevant force is the Coulomb force

$$\mathbf{f}_{ij} = \mathbf{n}_{ij} \frac{q_i q_j}{4\pi\epsilon_0 r^2} \quad (3)$$

where  $q_i$  is the charge,  $\epsilon_0$  is the dielectric constant of vacuum,  $r = |\mathbf{r}_j - \mathbf{r}_i|$  is the distance between the two particles and  $\mathbf{n}_{ij} = (\mathbf{r}_j - \mathbf{r}_i)/r$  is the normalized vector directing from  $i$ -particle to  $j$ -particle. Force due to the Lennard-Jones potential for the system of inert gas, such as argon, is expressed by

$$\mathbf{f}_{ij} = \mathbf{n}_{ij} \epsilon_{ij} \left( -2 \left( \frac{C_{ij}}{r} \right)^{14} + \left( \frac{C_{ij}}{r} \right)^8 \right) \quad (4)$$

In the case of astronomical problem (N-body problem), the relevant force is the gravitational force

$$\mathbf{f}_{ij} = \mathbf{n}_{ij} \frac{Gm_i m_j}{r^2} \quad (5)$$

where  $m_i$  is the mass of the particle and  $G$  is the Gravitational constant.

In solving the equation of motion (1) with the force, (2)-(5), the most computationally demanding part is the calculation of force (3)-(5) and its summation (2). These equations have two particle indices,  $i$  and  $j$ , which implies  $O(N^2)$  arithmetics. Once the total force on  $i$ -particle is known, the other part of the calculation, i.e. solving the ordinary differential equation (1) is much easier task of  $O(N)$  arithmetics. Here we used the notation  $O(N)$  and  $O(N^2)$  to indicate the arithmetic cost is proportional to the number of particles linearly and quadratic, respectively. Our strategy for attacking this problem is to use GPU as a coprocessor for calculating  $O(N^2)$  part of calculation. The  $O(N)$  part may be calculated either on GPU or CPU depending on the problem. In a similar concept, coprocessors for the  $O(N^2)$  force evaluation have been designed and produced for long time [2].

The organization of this article is as follows. In section 2, we review the preceding study of MD on GPU and our present results. In section 3, application to solving integral equation is discussed. In section 4, the performance of GPU is compared with that of specially designed chip for MD. All calculation was performed on DELL Dimension 9100 personal computer equipped with NVIDIA 7800 GTX GPU card and Windows XP Professional.

## 2 Molecular Dynamics on GPU

### 2.1 data stream scheme

By now, several calculations of MD on GPU have been reported. Nyland et al. [5] reported 4096 body simulation with  $290 \times 10^6$ (pair/second)  $\times 25$ (flop/pair)  $\approx 7$ (Gflops). Mark Harris reported N-body simulation of 8192 body system with  $480 \times 10^6$ (pair/second)  $\times 25$ (flop/pair)  $\approx 12$ (Gflops) [6], and  $672 \times 10^6$ (pair/second)  $\times 25$ (flop/pair)  $\approx 17$ (Gflops) [7]. These implementations of the force calculation is based on the parallelism of particle *pair*,  $(i, j)$ . They build a  $N \times N$  force matrix (texture),  $\mathbf{f}_{i,j}$ . The Cg code for this calculation is shown in Table 1. The float4 vector iPosMass contains three dimensional position vector and mass of  $i$ -particle  $(\mathbf{r}_i, m_i)$ .

Then the total force acting on  $i$ -particle,  $F_i$ , is calculated by summing up with  $j$ -particles (2) by using binary tree summation. This scheme is performed very efficiently in parallel on GPU because it has  $N^2$ -parallelism in the most heavy force calculation and  $N$ -parallelism in relatively light calculation of summation. A minor drawback of this algorithm is that the size of  $\mathbf{f}_{i,j}$  table is limited to 2048 by the maximum texture size. For more than 2048 particles, the particles should be divided into blocks of 2024 particles and the total force should be calculated block by block.

```

float4 force(float2 ij      : WPOS,
             uniform sampler2D pos) : COLOR0
{
// Pos texture is 2D, not 1D, so we need to
// convert body index into 2D coords for pos tex
float4 iCoords = getBodyCoords(ij);
float4 iPosMass = texture2D(pos, iCoords.xy);
float4 jPosMass = texture2D(pos, iCoords.zw);
float3 dir = iPos.xyz - jPos.xyz;
float r2 = dot(dir, dir);
        dir = normalize(dir);
return dir * g * iPosMass.w * jPosMass.w / r2;

```

**Table 1.** Mark Harris' code

## 2.2 naive loop scheme

To avoid this blocking of particles, we introduced the loop flow, which is available from fp40 profile, into the program. Now the algorithm is the same as that of conventional CPU calculation. Since the factor of  $m_i G$  is common for all  $j$ -particle, it is multiplied to  $\mathbf{F}_i$  after force summation. A Cg program for this algorithm is shown in Table 2. The  $O(N^2)$  texture memory for  $\mathbf{f}_{ij}$  is replaced with a register for  $\mathbf{F}_i$ . Therefore number of particles for one block is now  $2048 \times 2048$ , which is sufficiently large. Further, this scheme should work faster than Harris' because it has simpler structure and less switching of FBO's. Surprisingly, we found that this program works only at small  $N \sim 60$  but the GPU starts to give crazy result as  $N$  becomes larger for unknown reason (maybe some timing problem?). Therefore we had to adopt 'loop scheme with blocking' as described in the next subsection.

## 2.3 loop scheme with blocking

The Cg program for loop scheme with blocking is shown in Table 3. In this scheme the number of  $j$ -particle processed in a single shading is fixed to  $128 \times 32$ , which is determined by trials and errors to somehow optimize the efficiency. For blocking, we had to introduce another texture to keep partial sum  $\mathbf{F}_i$  named 'OldForce', and its switching with Output texture for each block. On one hand, if we took the number equal to the number of  $i$ -particle, the scheme reduces to the naive loop scheme of the previous subsection, which should be most efficient in theory but breaks down at large  $N$  in practice. On the other hand, if we take the number equal to one, the number of switching buffers becomes  $N$  and the scheme becomes very inefficient. One of the most time consuming part of the calculation in this scheme (as well as other schemes) is the texture search. The time spent for the texture search can be estimated by replacing 'texRECT(texture,j)' with something like 'float4(0.001\*j,1,1,1)'. The calculation time reduced from 0.45 sec to 0.36 sec for 16384 particles, that means about 20 % of time is consumed for

```

struct Output {
    float4 color : COLOR;
};

Output main(
    float2 i : TEXCOORD0,
    uniform int mx,
    uniform int my,
    uniform samplerRECT texture):COLOR
{
    Output OUT;
    float r,rr,k,l;
    float2 j;
    float3 ri,rj,rij,force;
    const float eps2=1e-6;
    force=0;
    ri = texRECT(texture,i);
    for(l=0.5; l <my; ++l){
    for(k=0.5; k <mx; ++k){
        j=float2(k,l);
        rj = texRECT(texture,j);
        rij= rj-ri;
        rr = eps2+dot(rij,rij);
        r = sqrt(rr);
        force += rij/(r*rr) ;
    };
    };
    OUT.color.xyz=force;
    return OUT;
}

```

**Table 2.** Simple loop scheme

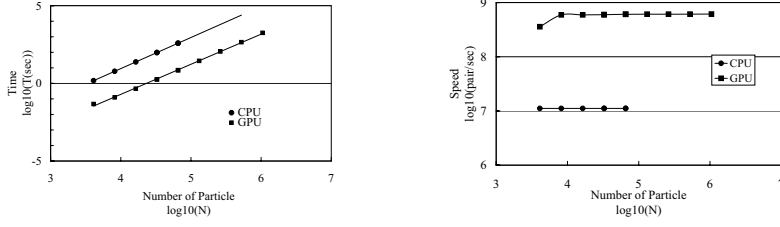
texture search. The present result indicates that MD calculation on GPU is also cash-limited as in the case of matrix-matrix multiplication [8].

The modification of the shading code to the Lennard-Jones potential is straightforward (Table 3). The fourth component of  $\mathbf{r}_i$  may contain the type of particle when number of types are larger than one. Using numerical table for interpolating the potential is, in general, a good method for calculating complex potential [2–4]. However, the improvement of calculation time for GPU turned out minor because large part of time is spent for the texture search of particle positions and potential lookup table.

```
// Lennard-Jones Potential
struct Output {
    float4 color : COLOR;
};

Output main(
    float2 i : TEXCOORD0,
    uniform int k,
    uniform samplerRECT OldForce,
    uniform samplerRECT texture):COLOR
{
    Output OUT;
    const float c=1.0;
    float r,rr,lj,q,q2,q4,l,m;
    float2 j;
    float3 rij,force;
    float4 ri,rj;
    ri = texRECT(texture,i);
    force = 0;
    for(m=0.5;m<32;++m){
    for(l=0.5;l<128;++l){
        j=float2(l,m+k);
        rj = texRECT(texture,j);
        rij=rj.xyz-ri.xyz;
        rr= clamp(dot(rij,rij),0.25,64.);
        q=c/rr;
        q2=q*q;
        q4=q2*q2;
        force += rij*(-2*q*q2+1)*q4;
    }
    }
    OUT.color.xyz = force + (texRECT(OldForce,i)).xyz;
    return OUT;
}
```

**Table 3.** loop scheme with blocking



**Fig. 1.** Time for force calculation

### 3 Integral equation

The numerical methods for force calculation can be applied to solving integral equations iteratively. The Fredholm equations of the second kind is to find a unknown function  $g(t)$  that satisfies

$$g(t) = \lambda \int_a^b K(t, s)g(s)ds + h(t) \quad (6)$$

where  $h(t)$  and  $K(t, s)$  are given functions. According to the Nystrom method [9], the integral is approximated by some quadrature rule such as Gauss-Legendre rule

$$g(t_i) = \lambda \sum_{j=1}^N w_j K(t_i, s_j)g(s_j) + h(t_i) \quad (7)$$

where  $w_j$  are weights of quadrature rule and  $s_j$  and  $t_i$  are quadrature points. Introducing an approximate solution of  $g(s_j)$  into the right hand side, we obtain the improved solution on the left hand side. By iterating this process until the solution converges, we can obtain the solution of (7). By defining the *force* as  $f_{ij} = w_j K(t_i, s_j)g(s_j)$ , the iteration formula can be calculated on GPU in the similar way as (2). A typical example of the Fredholm equation of the second kind in quantum mechanics is the Lippmann-Schwinger equation

$$\phi(\mathbf{r}) = \phi_0(\mathbf{r}) + \int d\mathbf{r}' G_0(\mathbf{r}, \mathbf{r}')V(\mathbf{r}')\phi(\mathbf{r}') \quad (8)$$

where the first term on the right hand side is the plane wave of incident electron and the second term is the wave function scattered by the potential  $V(\mathbf{r})$ , while  $G_0$  represents the Green's function of free electron. The numerical result of such a calculation will be presented elsewhere.

## 4 Comparison with GRAPE systems

The idea of accelerating numerical calculation by specially designed hardware is as old as the history of modern computers. One of the most successful attempts may be the GRAPE (GRAVity PipE) project for N-body problems. The latest chips for molecular dynamics are MDGRAPE2 [3] and MDGRAPE3 [4]. The comparison of data format and performance between GPU (GF7800GTX), MDGRAPE2, and MDGRAPE3 are listed in Table 4 and Table 5.

Table 4 shows that the position of particles is represented by 32 bit float in GPU, while 40 bit fixed in MDGRAPEs for easy implementation of periodic boundary condition. Since 32 bit float has 22 digit of relative accuracy corresponding to  $1\mu\text{m}$  in 1 m, both representation seem to have sufficient accuracy. The total force is represented by 32 bit float in GPU, 64 bit float in MDGRAPE2, and 80 bit fixed. In MDGRAPEs, high precision representations are used in order to avoid round off errors in the summation of force  $\sum_j \mathbf{f}_{ij}$ . 32 bit float in GPU might be not sufficiently accurate to avoid round off errors in some situations. However, adopting the tree method for force summation [5–7], which is believed more stable against round off error, may improve the situation. More detailed studies of round off error in various environments are awaited.

Table 5 shows that the performance of each chip is compared. The performance of GPU (GF7800GTX) with block loop algorithm is comparable to, but slightly lower than 17 Gflops of Harris’ algorithm [7]. The performance of GPU is higher than MDGRAPE2 chip and lower than MDGRAPE3. Considering the low price of GPU board (\$600), its cost/performance is estimated much lower than GRAPEs.

Another advantage of using GPU for MD and integral equation is that GPU is programmable. Therefore the force in MD and the kernel of integral equation can be complicated formula, while the force used with MDGRAPEs are restricted to coulomb/gravitational force and the tabulated ‘central’ two-body force.

Symbol	Data	GF7800GTX	MDGRAPE-2[3]	MDGRAPE-3[4]
$r_i$	position	32 bit float	40 bit fixed	40 bit fixed
$\mathbf{f}_{ij}$	force	32 bit float	32 bit float	32 bit float
$\mathbf{F}_i$	total force	32 bit float	64 bit float	80 bit fixed
	others	32 bit float	32 bit float	32 bit float

**Table 4.** comparison of data format

## 5 Summary

Molecular dynamics calculation implemented on GPU with algorithm of M. Harris has performance comparable to MDGRAPEs, specially designed chips for MD

Data	GF7800GTX	MDGRAPE-2[3]	MDGRAPE-3[4]
# of pipeline	32	4	20
clock (MHz)	400	100	460
peak (Gflops)	150	16.4	300
peak (Gpair/sec)	6	0.4	9
sustained (Gflops)	15	3.75	NA
sustained (Gpair/sec)	0.6	0.09	NA

**Table 5.** comparison of performance

calculation, and better cost/performance. GPU implementation has advantages in flexible programmability that allows more complex force/kernel functions than central two-body forces. GPU can calculate not only the force (2)-(5) but also the time-evolution (1). The issue of round off error in force accumulation should be examined more in details before applied to real scientific calculation.

## References

1. J.M. Haile, *Molecular Dynamics Simulation*, (John Wiley & Sons, New York) 1992.
2. J. Makino, and M. Taiji, *Scientific simulations with special-purpose computers – the GRAPE systems*, (John Wiley & Sons, Chicester) 1998.
3. R. Susukita, T. Ebisuzaki, B.G. Elmegreen, H. Furusawa, K. Kato, A. Kawai, Y. Kobayashi, T. Koishi, G.D. McNiven, T. Narumi, K. Yasuoka, *Comp. Phys. Commun.* **155**, 115 (2003).
4. M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, A. Konagaya, in Proceedings of SC’03, November 15-21, 2003, Phoenix Arizona, USA.
5. L. Nyland, M. Harris, and J. Prins, *N-body simulations on a GPU*, In Proceedings of the ACM Workshop on General-Purpose Computation on Graphics Processors (Aug. 2004).
6. Mark Harris, "GPGPU: General-Purpose Computation on GPUs", Game Developers Conference 2005. ([http://developer.nvidia.com/object/gdc\\_2005\\_presentations.html](http://developer.nvidia.com/object/gdc_2005_presentations.html)).
7. Mark Harris, "GPGPU: General-Purpose Computation on GPUs", SIGGRAPH 2005 GPGPU COURSE. (<http://www.gpgpu.org/s2005/>).
8. K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication", *Graphics Hardware 2004*, pp. 133-138.
9. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in Fortran 77*. Cambridge University Press, 1994.