LETTER *Special Section on Reconfigurable Systems*

Acceleration of the Fast Multipole Method on FPGA Devices

Hitoshi UKAWA[†a)], ***Nonmember and*** **Tetsu NARUMI**[††b)], ***Member***

SUMMARY The fast multipole method (FMM) for N-body simulations is attracting much attention since it requires minimal communication between computing nodes. We implemented hardware pipelines specialized for the FMM on an FPGA device, the GRAPE-9. An N-body simulation with 1.6×10^7 particles ran 16 times faster than that on a CPU. Moreover the particle-to-particle stage of the FMM on the GRAPE-9 executed 2.5 times faster than on a GPU in a limited case.

key words: *FPGA, fast multipole method, special-purpose pipeline, N-body simulation*

1. Introduction

The N-body simulation is one of the main focus areas in high performance computing, since it requires huge computational power. Without a sophisticated algorithm, the computational cost to calculate the gravity between N particles is $O(N^2)$, while that of the fast multipole method (FMM) is only $O(N)$ [1], [2]. Recently, the FMM's low communication requirement between computing nodes has become a hot topic since the parallel efficiency of the fast fourier transform often degrades with a large number of processors [3], [4]. Combining an accelerator, such as a graphics processing unit (GPU), with the FMM is one of the promising approaches to speed up the simulation further, as in exaFMM [5] and gemsFMM [6]. However, there is limited dedicated hardware for the FMM probably owing to the complexity thereof. We implemented a special-purpose pipeline for the FMM on a field programmable gate array (FPGA) using a high-level language, and obtained better performance compared with both CPU and GPU implementations.

2. Fast Multipole Method

FMM, proposed by Rokhlin in 1985, is an efficient algorithm for N-body simulations [1], [2] that reduces the computational complexity of N-body problems from $O(N^2)$ to $O(N)$. In a direct method, mutual forces are calculated between all pairs in the system. However, in the FMM, multipoles are made from multi particles before calculation, and the mutual force is calculated between multipoles rather than between particles. FMM is composed of six stages: P2M (particle to multipole), M2M (multipole to multipole), M2L (multipole to local), L2L (local to local), L2P (local to particle), and P2P (particle to particle). Before starting these stages, the simulation box is divided into eight subboxes and an octree is constructed hierarchically to handle multipoles efficiently. First, a multipole is calculated from neighboring particles in the P2M stage. The radius of the multipole expansion is enlarged in the M2M stage, and then the multipole expansion is converted to a local expansion by calculating the mutual forces between multipoles in the M2L stage. The radius of the local expansion is reduced in the L2L stage, and each particle's potential is calculated from the local expansion in the L2P stage. The contribution of distant particles is calculated during these five stages. Finally, the potential between neighboring particles is calculated directly in the P2P stage.

There are two parameters in the FMM: the opening angle, θ, and the order of multipole expansion, P. With a smaller θ, the tree becomes deeper and the calculation complexity of the M2L stage increases, while that of the P2P stage decreases. Using a larger θ, results in a shallower tree while the complexity of M2L decreases, and that of P2P increases.

3. Implementation

We used the GRAPE-9, a special-purpose computer for N-body simulations, as an FPGA device, since it can be reconfigured easily using the pipeline generator for programmable GRAPE, generation 2 (PGPG2) software described later.

3.1 GRAPE-9

We used the GRAPE-9 model800 [7]. A schematic of the GRAPE-9 system is shown in Fig. 1. The GRAPE-9 model800 has eight cards called processor chips, each of which has one Altera Cyclone IV GX FPGA and 2GB DDR2 memory. Since the GRAPE-9 was designed primarily to calculate gravity, pipelined circuits for this purpose was initially written on the FPGA. However users can implement original pipelines for the GRAPE-9 using the PGPG2 software [8], which means that a special-purpose accelerator can be constructed.

Manuscript received May 5, 2014.
Manuscript revised September 1, 2014.
Manuscript publicized November 19, 2014.

[†]The author is with the University of Tsukuba, Tsukuba-shi, 305–8577 Japan.
[††]The author is with the University of Electro-Communications, Chofu-shi, 182–8585 Japan.
a) E-mail: ukawa@hpcs.cs.tsukuba.ac.jp
b) E-mail: narumi@cs.uec.ac.jp

DOI: 10.1587/transinf.2014RCL0002

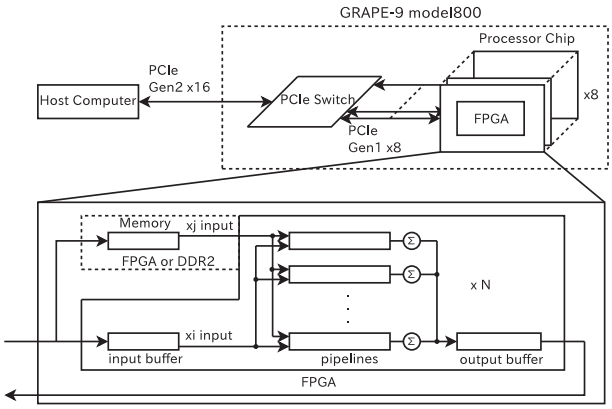

Fig. 1 Overview of a GRAPE-9 model800 system.

The GRAPE-9 with PGPG2 can not execute arbitrary programs but it can execute the following equation:

$$A_i = \sum_j^N f(x_i, x_j) \tag{1}$$

where, A_i is the ith result, $f()$ is the function between two particles, and x_n is the position of the nth particle. The $f()$ is converted into one pipelined circuit by the PGPG2, and is connected to the circuit to execute summation. These two circuits construct one pipeline. Therefore one summation in Eq. (1) is executed on one pipeline. When an FPGA has four pipelines, four different data, *i.e.*, x_i, x_{i+1}, x_{i+2}, x_{i+3}, are stored on the registers of different pipelines. The data of x_j is stored on the DDR2 memory or the block RAM in the FPGA. Once the calculation is started, the same x_j data is broadcasted to all the pipelines at every clock, and the pipeline circuits calculate $f(x_i, x_j)$, $f(x_{i+1}, x_j)$, $f(x_{i+2}, x_j)$, and $f(x_{i+3}, x_j)$, respectivey. After accumulating for all the x_j data, four results, *i.e.* A_i, A_{i+1}, A_{i+2}, and A_{i+3}, are transferred to the host computer. This cycle is repeated for all the x_i data.

One P2P pipline circuit that compiled by the PGPG2 uses 1,307 logic elements, and 72 pipelines are placed in one FPGA. So, if four cards are used, 288 pipelines in total are placed in the GRAPE-9. This means that 288 gravitational potentials are calculated at one time. About M2L, one M2L pipeline use 9,681 logic elements, and eight pipelines are placed in one FPGA. The number, eight, is limited by the algorithm to calculate the M2L stage not by the amount of logic elements in the FPGA.

3.2 PGPG2

The PGPG2 is developed based on PGPG [9] by the same company as the GRAPE-9. It automatically generates pipelined VHDL code and C libraries to control the GRAPE-9 from the source code written in its own language, PG2. Basically, users do not have to write the VHDL codes because PGPG2 automatically generates it.

PG2 code consist of a signal part, an equation part, and an output part. The PG2 code of the P2P stage is shown in Fig. 2. Lines 1 to 4 are signal part, which defines input and output signals. For I/O signals, one has to specify three tokens: data types, signal names, and signal types. Data types are int or float. As shown in the figure, an arbitrary bit length for the mantissa and exponent parts of the floating-point as well as the integer values can be specified. Signal types are fout, ipin or jpin. Fout means output, ipin means x_i input data, jpin means x_j input data. Lines 6 to 9 are equation part, which calculate the potential energy between x_i and x_j particles. Explicit declaration for inner variables like 'dx,' 'r2,' 'inr,' and 'mr' is not needed. The operation '=' means the connection of right and left signals. Other operations are the same as C language. Line 11 is output part, which converts the results to integers and sums them up. Output signals have to have operation '+=' that means summation. Vector variables can be described in PG2 as 'xi[3].' Multiplication between vectors like line 7 is converted to an inner product. The pg_pow() is the library function defined in PG2, which calculates the power function. Fig. 3 shows a part of the PG2 code of the M2L stage. It generates

```
 1  int64         pout      fout; // potential
 2  int32         xi[3]     ipin; // ip
 3  int32         xj[3]     jpin; // jp
 4  float18.9     mj        jpin; // src of ip
 5
 6  dx  = (float18.9)(xj - xi);
 7  r2  = dx * dx;
 8  inr = pg_pow(r2, -1, 2);
 9  mr  = inr * mj;
10
11  pout += (int64)(mr << 45);
```

Fig. 2 PG2 code of the P2P stage.

```
 1  r2 = ddx * ddx;                          // r^2
 2  inr = (flag) ? 0 : pg_pow(r2, -1, 2);   // 1/r
 3  inr2 = 0 - inr * inr;                    // -(1/r)^2
 4  inr3 = inr * inr2;                       // -(1/r)^3
 5  inr5 = 3 * inr3 * inr2;                  //  (1/r)^5
 6
 7  c0  = inr;             // 0th coefficient
 8
 9  c1  = ddx[0] * inr3;   // 1th coefficient
10  c2  = ddx[1] * inr3;
11  c3  = ddx[2] * inr3;
12
13  c4  = ddx[0] * ddx[0] * inr5 + inr3; // 2th coefficient
14  c5  = ddx[0] * ddx[1] * inr5;
15  c6  = ddx[0] * ddx[2] * inr5;
16  c7  = ddx[1] * ddx[1] * inr5 + inr3;
17  c8  = ddx[1] * ddx[2] * inr5;
18  c9  = ddx[2] * ddx[2] * inr5 + inr3;
19
20  /* 0th local coefficient */
21  L[0] += (int64)((M[0]*c0+M[1]*c1+M[2]*c2+M[3]*c3) << lshift);
22
23  /* 1th local coefficient */
24  L[1] += (int64)((M[0]*c1+M[1]*c4+M[2]*c5+M[3]*c6) << lshift);
25  L[2] += (int64)((M[0]*c2+M[1]*c5+M[2]*c7+M[3]*c8) << lshift);
26  L[3] += (int64)((M[0]*c3+M[1]*c6+M[2]*c8+M[3]*c9) << lshift);
27
28  /* 2th local coefficient */
29  L[4] += (int64)((M[0]*c4) << lshift);
30  L[5] += (int64)((M[0]*c5) << lshift);
31  L[6] += (int64)((M[0]*c6) << lshift);
32  L[7] += (int64)((M[0]*c7) << lshift);
33  L[8] += (int64)((M[0]*c8) << lshift);
34  L[9] += (int64)((M[0]*c9) << lshift);
```

Fig. 3 PG2 code of the M2L stage.

```
1: jmsize = MAX MEMORY SIZE
2: npipe = NUMBER OF PIPELINES
3: for each leaf cells of octree do
4:    j, i = 0
5:    while j < n_j do
6:       transfer jmsize sets of x_j data to GRAPE-9
7:       while i < n_i do
8:          transfer npipe sets of x_i data to GRAPE-9
9:          run the GRAPE-9
10:         get result from GRAPE-9
11:         i = i + npipe
12:      end while
13:      for i = 0 to n_i do
14:         summation of ith result
15:      end for
16:      j = j + jmsize
17:   end while
18: end for
```

Fig. 4 Pseudo code of the P2P stage on a host computer.

the pipeline to calculate the multipole to the local operation for P=2. Line 7 to 18 calculate variables of 'c0' to 'c9,' coefficients of M2L. The variables of 'M[0]' to 'M[3]' are multipole coefficients. Then line 20 to 34 calculate local coefficients from variables 'c0' and 'M[0]' to 'c9' and 'M[3].' Fig. 4 shows how the host computer controls the GRAPE-9 in the P2P stage. In lines 6 and 8, the host computer sends the data to the pipeline, and in line 10, it receives the results. In line 9 the signal to start the calculation is sent to the pipeline. These operations are performed by special APIs generated by the PGPG2. Basically, the execution flow of other stages is similar to Fig. 4, although the parameters are different.

4. Result

First, we ran the N-body simulation code using the FMM on a CPU and the GRAPE-9, and compared the results of each stage as shown in Fig. 5. One processor chip of the GRAPE-9 was used, while the CPU was an Intel core i3 3220 3.30GHz processor. The number of particles was 10^6, the max tree depth was 5, and the expansion order was 2. The GRAPE-9 is faster than the CPU in the M2L and P2P stages, but slower in the other stages. This is because the M2L and P2P stages have much higher computational complexity than the other stages, and the GRAPE-9 is more efficient under these conditions. For both the CPU and GRAPE-9, the M2L and P2P stages dominate the total calculation time. Therefore, we used both the GRAPE-9 and CPU in the next experiment, *i.e.*, M2L and P2P were executed on the GRAPE-9, and the other stages on the CPU. The P2P and M2L stages used four and one processor chips, respectively. The total execution times are shown in Fig. 6. 'GRAPE-9' denotes the combination of CPU and GRAPE-9, and 'Direct' means direct calculation without the FMM

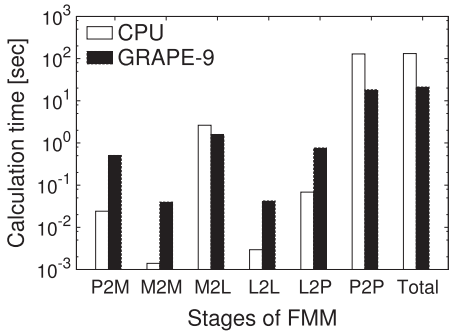

Fig. 5 Calculation time of each stage on a CPU and GRAPE-9.

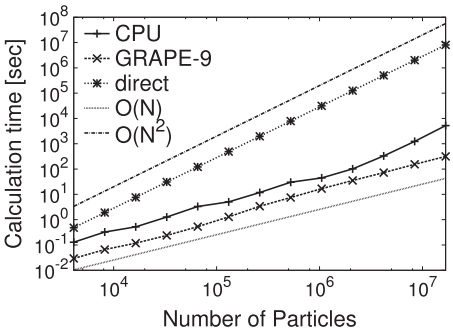

Fig. 6 Total execution time of FMM on CPU and GRAPE-9.

on the CPU. As shown in Fig. 6, the calculation time of the direct method scales as $O(N^2)$, while both FMMs scale as $O(N)$. Comparing the execution of the FMM on the combined CPU and GRAPE-9 with that on only the CPU shows that the combination gets consistently better performance under all conditions, culminating in 16 times faster with 1.6×10^7 particles in this experiment.

Next, we compared the P2P stage on GRAPE-9 with that on a GPU, confirming GRAPE-9 executes faster than the GPU. The target GPU device, an NVIDIA GeForce GT 240, was specifically chosen because it uses similar technology and electrical power to the GRAPE-9. The GT 240 uses 40-nm technology and its maximum power consumption is 70W, while the FPGA in the GRAPE-9 uses 60-nm technology and consumes 60W, with four cards. Therefore the GT 240 should be faster from a technology and power consumption perspective. In terms of price, GRAPE-9 is much more expensive than GT240. GRAPE-9 is four thousand dollar with four cards, while GT 240 is about one hundred dollar. The software running on the GPU is gemsFMM [6], developed by Yokota. Four processor chips were used in this calculation. Comparative results for the GRAPE-9 and the GPU executing the P2P stage are shown in Fig. 7. The calculation speed of the GPU is maximal with 3×10^4 particles or more. This is because the peak speed of this GPU is about 160 GFLOPS. With 10^5 particles, the calculation speed of the GRAPE-9 reaches about 350 GFLOPS, but thereafter, it declines sharply and then increases. The reason for these up and down performance swings is that the optimal depth of the octree changes depending on the number of particles.

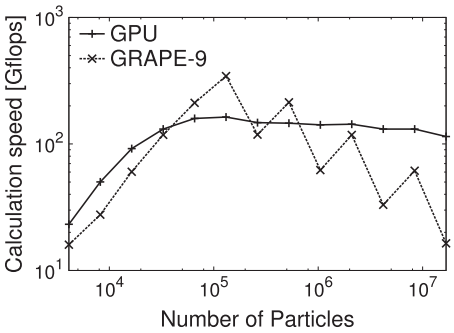

Fig. 7 Calculation speed of the P2P on a GPU and GRAPE-9.

For example, if the number of particles exceeds 10^5, the total execution time of FMM can be reduced by increasing the tree depth. If the tree depth is increased one level, the calculation cost of P2P becomes eight times smaller. However, with the GRAPE-9, the efficiency decreases since the communication time between the GRAPE-9 and host computer becomes a bottleneck. Current version of PGPG2 cannot access randomly to the memory. The x_j data always broadcasted contignuously from the first address. So the host computer often have to transfer the same x_j data for neighboring particles to the GRAPE-9 every time P2P stage started. In fact, the same particle data must be transferred 125 times from host computer to the GRAPE-9 because of this reason.

5. Conclusion

We implemented the FMM on an FPGA device, the GRAPE-9. The calculation speed of our system is faster than that on the CPU. Under specific conditions, it is also faster than on a GPU.

To achieve better performance by reducing communication of x_j data between the GRAPE-9 and the host computer, new functions need to be added to PGPG2 to handle the cell-index method, which accesses particles by cells. This will obviate the need to transfer the same particle data repeatedly.

Moreover, using the new GRAPE-9 would be useful for further acceleration. The GRAPE-9 model5000, released in November 2013, has 16 processor chips, and it includes a new version of FPGA, the Altera Cyclone V GX. Using multiple GRAPE-9 computers is also future work.

References

[1] V. Rokhlin, "Rapid solution of integral equations of classical potential theory," Comp. Phys., vol.60, pp.187–207, Sept. 1985.
[2] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulation," Comp. Phys., vol.135, pp.280–292, Aug. 1997.
[3] R. Yokota, L.A. Barba, T. Narumi, and K. Yasuoka, "Petascale turbulence simulation using a highly parallel fast multipole method on GPUs," Comp. Phys. Communs., vol.184, pp.445–455, March 2013.
[4] A. Arnold, F. Fahrenberger, C. Holm, O. Lenz, M. Bolten, H. Dachsel, R. Halver, I. Kabadshow, F. Gahler, F. Heber, J. Iseringhausen, M. Hofmann, M. Pippig, D. Potts, and G. Sutmann, "Comparison of scalable fast methods for long-range interactions," Phys. Rev. E, vol.88, 063308, Dec. 2013.
[5] R. Yokota, "An FMM based on dual tree traversal for many-core architectures," Algorithm & Comp. Tech., vol.7, no.3, pp.301–324, Sept. 2013.
[6] R. Yokota, "gemsFMM," https://sites.google.com/site/rioyokota/software/gemsfmm, accessed April 20, 2014.
[7] T. Fukushige and A. Kawai, "GRAPE-9 model800," K&F Computing Research, http://www.kfcr.jp/grape9-e.html, accessed April 20, 2014.
[8] T. Fukushige and A. Kawai, "Pipeline-designing utility PGPG2 (for GRAPE-9)," K&F Computing Research, http://www.kfcr.jp/grape9-e.html, accessed April 20, 2014.
[9] T. Hamada, T. Fukushige, and J. Makino, "PGPG: An automatic generator of pipeline design for programmable GRAPE systems," Publications of the Astronomical Society of Japan, vol.57, no.5, pp.799–813, Oct. 2005.