PROCEEDINGS OF SCIENCE

Application of Many-core Accelerators for Problems in Astronomy and Physics

Naohito Nakasato*University of Aizu, Japan

E-mail: nakasato@u-aizu.ac.jp

Recently, many-core accelerators are developing so fast that the computing devices attract researchers who are always demanding faster computers. Since many-core accelerators such as graphic processing unit (GPU) are nothing but parallel computers, we need to modify an existing application program with specific optimizations (mostly parallelization) for a given accelerator. In this paper, we describe our problem-specific compiler system for many-core accelerators, specifically, GPU and GRAPE-DR. GRAPE-DR is another many-core accelerators device that is specially targeted scientific applications. In our compiler, we focus a compute intensive problem expressed as two-nested loop. Our compiler asks a user to write computations in the inner-most loop. All details related to parallelization and optimization techniques for a given accelerator are hidden from the user point of view. Our compiler successfully generates the fastest code ever for astronomical N-body simulations with the performance of 2.6 TFLOPS (single precision) on a recent GPU. Another successful application on both GPU and GRAPE-DR is the evaluation of a multi-dimensional integral in quadruple precision. The program generated by our compiler runs at a speed of 15 QD-GFLOPS on GPU and 4 QD-GFLOPS on GRAPE-DR. The performance obtained so far is more than 50-200 times faster than a conventional CPU.

13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research
February 22-27, 2010
Jaipur, India

*Speaker.

PoS(ACAT2010)015

1. Introduction

The rise of many-core accelerators such as Cell and GPU opens a new way of high performance computing. An important question is what types of practical applications are efficient on many-core accelerators. Our simple answer is a problem that requires high compute density. Even in Cell and GPUs with the external memory bandwidth at $\sim$ 100 GB s^{-1} or more, the memory-wall problem is quite severe since the raw performance of these processors ($>$ a several 100 GFLOPS) is very high compared to the memory bandwidth. Applications that allow repeated reuse of data or applications with high compute density are the most efficient on many-core accelerators (and also on general purpose CPUs).

A well-known example of this type of applications is a many-particle simulation. In astronomical many-particle simulations, the most time consuming part is the evaluation of mutual force between particles:

$$f_i = \sum_{j=1}^{N} \frac{m_j(x_i - x_j)}{(|x_i - x_j|^2 + \varepsilon^2)^{3/2}}, \tag{1.1}$$

where x_i, m_i, ε are position of a particle, the mass, and a parameter that prevents division by zero, respectively. Given a number of particles N, this force evaluation requires $O(N^2)$ complexity but other part of the simulations, such as orbit integration, requires only $O(N)$ complexity. It was shown that one can do the evaluation of mutual force very efficiently with an accelerator device called GRAPE (GRAvity piPE) [1, 2]. This $O(N^2)$ direct summation force evaluation is fundamental to the modeling of dense star clusters that are collisional system. Note that there is an $O(N\log N)$ method [3] for collision-less system like galaxy and cosmological simulations but the method is not applicable to star clusters.

The GRAPE system is widely used in astronomical community. It is a specially designed computing system to calculate Newtonian gravity between particles expressed in Eq.(1.1). In GRAPE system, all calculations except computation of Eq.(1.1) are done on a host computer that controls GRAPE system. The host computer sends x_i and m_i to GRAPE and receives results f_i. In other words, only the most computing intensive part of many-particle simulations is computed on the specially developed component. Apparently, this same work division technique is applicable to a system with GPU ([4] and many others). Note $O(N^2)$ direct summation force evaluation is very effective for many-core accelerators because in this evaluation each element of data is reused $O(N)$ times.

A class of applications with high compute density is dense matrix multiplication. To compute square matrix ($N \times N$) multiplication, we require $2N^3$ operations with a naive implementation. In other words, each element of data is reused $O(N^{0.5})$ times. Although compute density of the dense matrix multiplication is not as high as the force evaluation, it is effective to utilize many-core accelerators. In the recent TOP500 benchmark[1], which heavily rely on the dense matrix multiplication, two systems with many-core accelerators were spotted on 2nd (Cell) and 5th (GPU).

In this paper, we report our implementation of $O(N^2)$ force evaluation scheme on many-core accelerators. Furthermore, we apply our $O(N^2)$ scheme to an evaluation of the Feynman path integral arises in the particle physics. A direct computation of the Feynman path integral is numer-

[1] November 2009

ically unstable with double precision (DP) operations because of its divergent nature. So a solution to this difficulty is that we compute the integral with quadruple precision (QP) operations [5]. If we implement QP operations with the emulation scheme by [6] and [7], which utilizes DP units for emulation, one QP variable is expressed as a sum of two DP variables so that one QP addition and multiplication requires 20 and 23 DP operations, respectively. Thus, it is expected that the peak performance of QP operations is at least 20 times slower than its DP performance. Thus, the evaluation of the integral is a highly time consuming task. We show that many-core accelerators are effective to evaluate a simple one-loop integral with the QP emulation scheme.

2. Architecture of Many-core Accelerators

In this section, we briefly describe many-core accelerators we used in the present work.

2.1 GPU: Cypress Architecture

The Cypress GPU from AMD/ATi is the company's latest GPU with many enhancements for general purpose computing on GPU (GPGPU). It has 1600 arithmetic units (called a stream core), each of which is capable of executing single precision floating-point (FP) multiply-add. At the time of writing, the fastest Cypress processor is running at 850 MHz and offers a peak performance of $1600 \times 2 \times 850 \times 10^6 = 2.71$ Tflops.

Moreover, these units are organized hierarchically as follows. At one level higher from the stream cores, a five-way very long instruction word (VLIW) unit called a thread processor (TP) that consists of four simple stream cores and one transcendental stream core. Therefore, one Cypress processor has 320 TPs. The TP can execute either at most five single-precision/integer operations, four simple single-precision/integer operations with one transcendental operation, or double-precision operations by combinations of the four stream cores. A unit called a SIMD engine consists of 16 TPs. At the top level, there are 20 SIMD engines, a controller unit called an ultra-threaded dispatch processor, and other units such as units for graphic processing, memory controllers and DMA engines. An external memory attached to the Cypress is 1 GB GDDR5 memory with a bus width of 256 bit. It has a data clock rate at 4800 MHz and offers us a bandwidth of 153.6 GB sec^{-1}.

We program the Cypress GPU through an assembly like language called IL (Intermediate Language). The IL is like a virtual instruction set for GPU from AMD/ATi. With IL, we have full control of every VLIW instructions. A code written in IL is called a compute kernel.

2.2 GRAPE-DR Architecture

GRAPE-DR (Greatly Reduced Array of Processor Elements with Data Reduction) is a specially developed many-core accelerator for applications in Astronomy. It has 1024 FP arithmetic units. A half of the units are double precision (DP) addition units and another are SP multiplication units. Logically, we program 512 add/mul units in SIMD-way to do useful calculations. GRAPE-DR is designed to optimize to compute a force summation like Eq.(1.1) as schematically shown in Figure 1 (see [8] for detailed internal structure of GRAPE-DR). With clock speed of 380 MHz, a performance of one GRAPE-DR chip is 195 GFLOPS in DP operations and 390 GFLOPS in SP operations, respectively. The GRAPE-DR is programmable but not fully programmable unlike

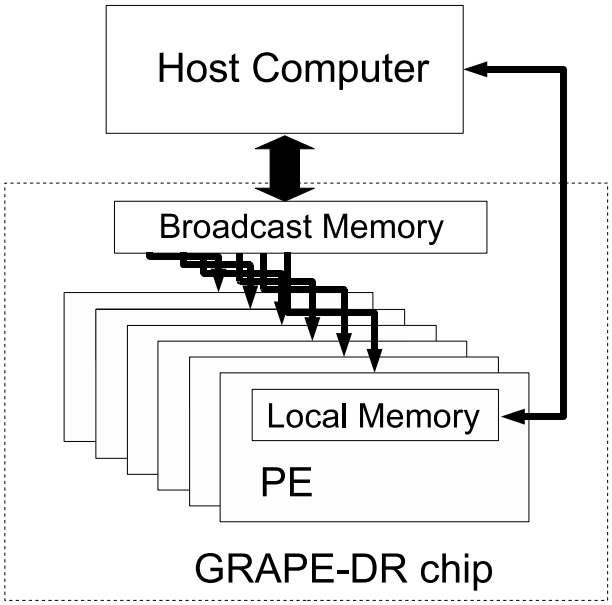

Figure 1: Schematic view of the GRAPE-DR system. It consists of two parts: a host computer and GRAPE-DR chip(s). GRAPE-DR chip has 512 processing elements (PE). A PE is a unit of computing components in GRAPE-DR which has own local memory (LM) and arithmetic units (DP add, SP mul and integer ALU). Every 32 PE is grouped to constitute a broadcast block (BB) unit. Each BB unit has a memory component (broadcast memory; BM) that is shared by all 32 PEs, namely, all PE simultaneously read a same data. When we view this memory read operation from BB, it is a broadcast of a data from BM to all 32 PEs. Note in the figure, we depict 1 BB unit that hosts 8 PEs.

other many-core accelerators such Cell and GPUs. Its architecture and memory system is simplified to support only limited types of applications. Many-particle simulations, which are considered to be compute intensive, are very efficiently executed on GRAPE-DR.

2.3 A Compute Model of Many-core Accelerator

After the introduction of new GPUs like Cypress, it turns out that the GRAPE-DR is very similar to such recent GPUs. In the present work, we treat GRAPE-DR and Cypress GPU are logically same system as described below (see Figure 1). Based on this compute model, we have developed a special compiler system for GRAPE-DR and Cypress GPU. Reference [9] could be consulted for a detailed description of our compiler system.

Our logical many-core accelerator has far many FP arithmetic units working in SIMD-way. Each arithmetic unit has own local memory. The local memory corresponds to general purpose registers on each TP in the Cypress and registers and LM on each PE in the GRAPE-DR, respectively. In the GRAPE-DR, all PEs are connected to the broadcast memory that is shared by all PEs. Main purpose of the broadcast memory is that all PEs can load the same data efficiently. The Cypress has similar shared memory components but in the present work we do not use the shared memory components. Instead, we regard a read cache memory as replacement of the broadcast memory. The cache memory on the Cypress works effectively like the broadcast memory.

```
for i = 0 to N-1
  s[i] = 0
  for j = 0 to N-1
    s[i] += f(x[i], x[j])
```

Figure 2: A simple nested loop to computer a general force calculation.

3. $O(N^2)$ force summation on many-core accelerators

In this section, we describe how we use many-core accelerators to compute a general force summation. It is expressed with the following equation:

$$s_i = \sum_j^N F(a_i, b_i, c_j, d_j ...), \tag{3.1}$$

where F is a function that evaluates a value from input variables $a_i, b_i, c_j, ...$ and s_i is a summation result. If input variables $a_i, b_i, c_j, ...$ are vector components of a position of a particle and mass of the particle, this equation is reduced to gravity force equation in Eq.(1.1). Also, for a given quadrature, we can do a numerical integration with this formulation by regarding $a_i, b_i, ...$ as integration points and function F as integrand. Another trivial application is to compute a complicated function for a very large number of times. This typically arises in a Monte Carlo integration scheme. In this case, we do not take summation over variables.

Suppose we implement a program to compute the general force summation like Eq.(3.1). This can be simply calculated by a nested loop as shown in Figure 2 (a). To map this nested loop on many-core accelerators, we unroll the outer loop as shown in Figure 3 (b) and assign computations of each inner loop for `x[i]` to processors on an accelerator. The loop unrolling is a standard technique on general purpose CPUs to enhance compute density by reducing required memory bandwidth and also by latency hiding for arithmetic units. If we unroll the outer loop by n ways, the number of times `x[j]` loaded is reduced by a factor of n. On a general purpose CPU, n is limited to 4 - 8 at most due to a small number of registers (typically $\sim$ 16 - 128 in DP words). However, many-core accelerators we consider here have more than 100 FP arithmetic units and each arithmetic unit has 32 - 128 registers. So we can regard an aggregate number of registers is 1000 - 5000. Therefore we can unroll the loop by roughly 200-500 ways provided that a many-core accelerator has memory component shared by all arithmetic units or cache memory. This greatly reduced required memory bandwidth is the key to efficiently utilize many-core accelerators. Specifically, in the example here, `x[i]` is reused repeatedly whole time during the inner loop and each `x[j]` is used once during the inner loop but it is shared by n logical processors. The GRAPE-DR is designed to be optimized to this unrolling technique. In the case of GRAPE-DR, all `x[]` are stored on the broadcast memory (BM). The BM broadcasts each `x[j]` in each iteration for the inner loop.

With the Cypress GPU, it is best to utilize 4-vector SIMD unit as much as possible for gaining maximum performance. So one way to make 4-vector SIMD unit on each TP busy is to unroll the inner loop of force calculation in 4 ways as shown in Figure 4.

```
for i = 0 to N-1 each 4
  s[i] = s[i+1] = s[i+2] = s[i+3] = 0
  for j = 0 to N-1
    s[i]   += f(x[i],   x[j])
    s[i+1] += f(x[i+1], x[j])
    s[i+2] += f(x[i+2], x[j])
    s[i+3] += f(x[i+3], x[j])
```

Figure 3: Unroll i-loop in 4 ways. We assign computations of each inner loop for different i to four different processors on a many-core accelerator.

```
for i = 0 to N-1 each 4
  s[i] = s[i+1] = s[i+2] = s[i+3] = 0
  for j = 0 to N-1 each 4
    for k = 0 to 3
      s[i  ] += f(x[i  ], x[j+k])
      s[i+1] += f(x[i+1], x[j+k])
      s[i+2] += f(x[i+2], x[j+k])
      s[i+3] += f(x[i+3], x[j+k])
```

Figure 4: Unroll both i-loop and j-loop in 4 ways

4. Astronomical Application

We did an experiment to implement the force calculation loop Eq.(1.1) with the two schemes shown in Figure 3 and 4 (see [10] for details). Precisely, we have implemented conventional equations expressed as

$$
\begin{aligned}
p_i &= \sum_{j=1,j\neq i}^{N} p(x_i,x_j,m_j) = \sum_{j=1,j\neq i}^{N} \frac{m_j}{(|x_i-x_j|^2+\varepsilon^2)^{1/2}}, \\
f_i &= \sum_{j=1,j\neq i}^{N} f(x_i,x_j,m_j) = \sum_{j=1,j\neq i}^{N} \frac{m_j(x_i-x_j)}{(|x_i-x_j|^2+\varepsilon^2)^{3/2}},
\end{aligned}
\tag{4.1}
$$

where p_i and f_i are potential and force for a particle i, and x_i, m_i, ε are position of a particle, the mass, and a parameter that prevents division by zero, respectively. In the most inner loop, by simultaneously evaluating functions p and f, we require 22 arithmetic operations, which include one square root and one division, to compute an interaction between particle i and j. Since previous authors starting from [11] used a conventional operational count for evaluation of f_i and p_i, we also adopt the conventional counts of 38 throughout the paper. In Figure 5, we plot a computing speed of our optimized IL code for computing Eq.(4.1) as a function of N. We have obtained

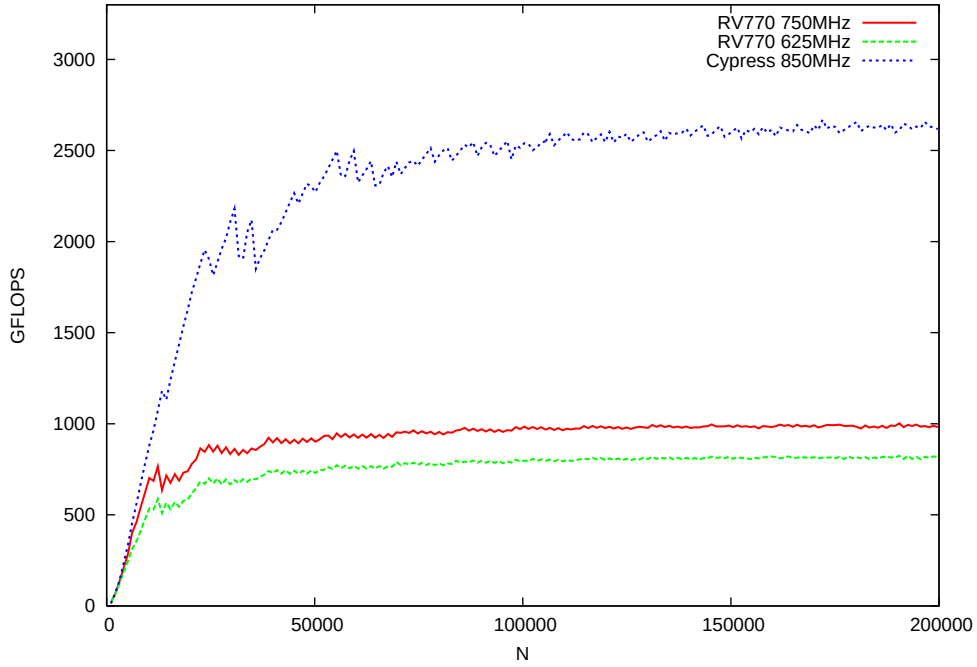

Figure 5: A performance of our $O(N^2)$ force evaluation scheme on various GPUs. RV770 is an old generation GPU architecture with 160 TPs.

$\sim$ 2600 GFLOPS at $N > 100000$ on Cypress GPU running at 850 MHz. As far as we know, the performance we obtained is fastest ever with one GPU chip.

5. One-loop Integral

A simple example of such integral is an one-loop integral expressed as

$$\begin{aligned} I &= \int_0^1 dx \int_0^{1-x} dy \int_0^{1-x-y} dz F(x,y,z), \\ F(x,y,z) &= D(x,y,z)^{-2} \\ D &= -xys - tz(1-x-y-z) + (x+y)\lambda^2 \\ &\quad +(1-x-y-z)(1-x-y)m_e^2 \\ &\quad +z(1-x-y)m_f^2. \end{aligned} \tag{5.1}$$

Here, s and t are parameters and m_e, and m_f are physical constants. And λ is a fictitious photon mass that is supposed to be zero so that accurate evaluation of this integral is actually very hard due to its divergent nature [5]. In [5], they have reported that a combination of a multi-dimensional integration scheme and an extrapolation scheme on λ [12] is necessary to tackle to this problem.

If we adopt the double exponential integral scheme, this integration is reduced to the three nested summation, which requires $\sim 27N^3$ operations where N is a number of integration points in one direction. Practically, given s and t, we need to evaluate the integral repeatedly for ~ 20 times due to the extrapolation scheme. Accordingly, with $N = 1024$, a total number of required QP operations for one evaluation is $\sim 6 \times 10^{11}$. Furthermore, we need to evaluates the integral with

	Add	Mul	Div
Cypress	21	25	53
GRAPE-DR	21	41	199

Table 1: A number of required DP operations to emulate a given QP emulation on Cypress GPU and GRAPE-DR.

```
LMEM xx, yy, cnt4;
BMEM x30_1, gw30;
RMEM res;
CONST tt, ramda, fme, fmf, s, one;

zz = x30_1*cnt4;
d = -xx*yy*s-tt*zz*(one-xx-yy-zz)+(xx+yy)*ramda**2 +
        (one-xx-yy-zz)*(one-xx-yy)*fme**2+zz*(one-xx-yy)*fmf**2;

res += gw30/d**2;
```

Figure 6: Source code for our compiler system to compute the integral in (5.1).

different combination of s and t. The number of combination is as large as $\sim 10^6$. And there are many other integrals, each of which corresponds to a specific condition. The good news is that the QP emulation scheme is expected to be efficient on many-core accelerators due to its intrinsic nature. That is one QP addition requires 4 DP variables as input and executes 20 DP operations to obtain 2 DP variables. In other words, the QP emulation scheme is also quite compute intensive in addition to many-particle simulations.

We have developed the QP emulation library for both the Cypress GPU and GRAPE-DR. Table 1 shows a number of DP operations to emulate each DD arithmetic operations on Cypress GPU and GRAPE-DR. In terms of the number of DP operation counts, the Cypress GPU is more efficient than GRAPE-DR. We apply our $O(N^2)$ force evaluation scheme to compute this nested summation. Given x and y, we compute the inner-most summation of Eq.(5.1) with our scheme. We compute a several hundred combinations of x and y in parallel with a many-core accelerator. With our QP emulation library, the Cypress GPU computes 320 combinations of x and y in parallel for instance.

The definition of the integrand in the original Fortran code is written as 2 lines. The source code for our compiler system to computer Eq.(5.1) is written as 9 lines including definitions of variables that is denoted as `LMEM`, `BMEM` and, `RMEM` (Figure 6). Each of these lines defines input LM variables, BM variables, and output LM variables. From this input source code, our compiler generates assembly code for Cypress GPU and GRAPE-DR. The source code contains 26 add/mul operations and 1 div operation. For Cypress GPU, the generated assembly language contains 319 VLIW instructions, which includes instruction related to data load and a loop operation. Since an

	$N = 256$	$N = 512$	$N = 1024$	$N = 2048$	clock
GRAPE-DR	0.21	1.21	7.83	55.1	380
Cypress	0.05	0.29	1.94	14.2	850
Core i7	7.39	59.0	472	—	2670

Table 2: The measured elapsed time to compute the integral in Eq.(5.1) with different integration points. The last column indicates clock speed of processors in MHz.

expected number of VLIW instructions for 27 QP operations is roughly $26 \times (21+25)/2+53 = 651$ operations, the Cypress VLIW architecture is very effective to the QP emulation scheme. For GRAPE-DR, the generated assembly language contains 1226 instructions, which includes 40 nop operations. In other words, a percentage of time during which arithmetic units are stalled is as small as 3 %.

In Table 2, we present the elapsed time to compute the integral in Eq.(5.1) with different N. We show the elapsed time for Cypress GPU running at 850 MHz and GRAPE-DR chip running at 380 MHz. Additionally, we present the elapsed time with a conventional CPU (note this is a result with single core of Core i7 running 2.67 GHz). Depending on the size of N, the measured computing speeds are 13.0, 15.5, and 16.9 QD-GFLOPS for $N = 512$, 1024, and 2048, respectively, for Cypress GPU. For comparison, in case of $N = 1024$, we have obtained 3.83 QD-GFLOPS with GRAPE-DR and 63.7 QD-MFLOPS with Core i7 CPU. Here, we assume the total number of QD operations is $28N^3$ (one division is equivalent to two add/mul operations). The Cypress GPU shows impressive performance gain (> 200 times) compared to the conventional CPU. The GRAPE-DR also shows good performance gain (~ 60 times). Additional advantage of GRAPE-DR is its low power consumption. Nominal power consumption of three architectures is ~ 200 W for Cypress GPU, ~ 60 W for GRAPE-DR, and $\sim 130/4 \sim 33$ W for Core i7 CPU. So, both many-core accelerators show roughly similar performance per watt with this particular problem.

6. Conclusion

In this paper, we introduce a newly developed compiler system for high performance computing using many-core accelerators. Accelerators are effective on specific problems that share a certain pattern of calculations. Specifically, they are suited to calculations which allow repeated reuse of data, and a calculation with high compute density.

Our novel programming model for such calculations is simple but sufficient to implement a several important compute intensive applications such as many-particle simulations and the double exponential integral scheme. More precisely, our compiler system generates highly efficient codes for many-core accelerators Cypress GPU and GRAPE-DR. We have obtained ~ 2.6 TFLOPS with $O(N^2)$ force evaluation for application in astronomy. Also, we have shown that the QD emulation scheme is well suited to the many-core accelerators and very power efficient. With Cypress GPU, we have obtained ~ 15 QD-GFLOPS that is more than 200 times faster than a conventional CPU with single thread. Combined with the scheme proposed in [5], our compiler system is effective to

utilize a desktop computer equipped with GPU/GRAPE-DR for computing more precise integral value.

This work was supported in part by the Grant-in-Aid of the Ministry of Education (No. 20105005 and 21244020).

References

[1] D. Sugimoto, Y. Chikada, J. Makino, T. Ito, T. Ebisuzaki, and M. Umemura, "*A Special-Purpose Computer for Gravitational Many-Body Problems*," *Nature*, vol. 345, pp. 33–35, 1990.

[2] J. Makino and M. Taiji, *Scientific Simulations with Special-Purpose Computers — The GRAPE Systems*. New York: John Wiley and Sons, 1998.

[3] J. Barnes and P. Hut, "*A Hierarchical O(NlogN) Force-Calculation Algorithm*," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[4] L. Nyland, M. Harris, and J. Prins, "*Fast N-Body Simulation with CUDA*," in *GPU Gems3*. New York, NY: Addison-Wesley, 2007, pp. 677–696.

[5] F. Yuasa, E. de Doncker, J. Fujimoto, N. Hamaguchi, T. Ishikawa, and Y. Simizu, "*Precise Numerical Evaluation of the Scalar One-Loop Integrals with the Infrared Divergence*," in *Proceedings of the ACAT workshop*, 2007, pp. 446–449.

[6] D. Kunuth, *The Art of Computer Programming vol.2 Seminumerical Algorithms*, 1st ed. Reading, Massachusetts: Addison Wesley, 1998.

[7] T. Dekker, "*A Floating-Point Technique for Extending the Available Precision*," *Numerische Mathematik*, vol. 18, pp. 224–242, 1971.

[8] J. Makino, "*Specialized Hardware for Supercomputing*," *SciDAC Review*, pp. 54–65, 2009.

[9] N. Nakasato and J. Makino, "*A Compiler for High Performance Computing With Many-Core Accelerators*," in *IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–9.

[10] K. Fujiwara and N. Nakasato, "Fast Simulations of Gravitational Many-body Problem on RV770 GPU," 2009, extended undergraduate thesis (University of Aizu 2008). [Online]. Available: http://jp.arxiv.org/abs/0904.3659

[11] M. Warren, J. Salmon, D. Becker, M. Goda, T. Sterling, and G. Winckelmans, in *Proceedings of Supercomputing '97*, 1997.

[12] P. Wynn, "*On the convergence and stability of the epsilon algorithm*," *SIAM Journal of Mathematical Physics*, vol. 3, pp. 91–122, 1966.