A Compiler for High Performance Computing With Many-Core Accelerators

Naohito Nakasato [1] and Jun Makino [2]

[1] *University of Aizu, Department of Computer Science and Engineering*
nakasato@u-aizu.ac.jp

[2] *National Astronomical Observatory of Japan*
makino@cfca.jp

***Abstract*—We introduce a newly developed compiler for high performance computing using many-core accelerators. A high peak performance of such accelerators attracts researchers who are always demanding faster computers. However, it is difficult to create an efficient implementation of an existing serial program for such accelerators even in the case of massively parallel problems. While existing parallel programming tools force us to program every details of an implementation from loop-level parallelism to 4-vector SIMD operations, our novel approach is that given a compute intensive problem expressed as a nested loop, the compiler only ask us to define a compute kernel inside the inner-most loop. We observe that input variables appeared in the kernel is classified into two types; invariant during the loop and variables updated in each iteration. The compiler let us to specify either type of the inputs so as it pick a predefined optimal way to process them. The compiler successfully generates the fastest code ever for many-particle simulations with the performance of 500 GFLOPS (single precision) on RV770 GPU. Another successful application is the evaluation of a multi-dimensional integral. It runs at a speed of 5 - 7 GFLOPS (quadruple precision) on both GRAPE-DR and GPU.**

I. Introduction

A. Programming Many-Core Accelerators

The rise of many-core accelerators such as Cell, GPU, and commodity multicore CPU with SIMD units opens a new way of high performance computing but also poses a serious problem of how to program highly parallel floating-point (FP) units on these accelerators.

Roughly speaking, until recently there have been two conceptual levels of parallel programming. (1) vector processing and (2) distributed parallel processing. In 70's - mid 80's, we had to modify our serial programs into the form suitable for vector processing to get the highest performance. At that time, an important programming optimization was to vectorize loops with user directives. After the introduction of massively parallel processor (MPP) in late 80's and commodity PC cluster in mid 90's, we were forced to use a new programming concept that is the explicit parallel programming with message passing. A important question at that time was how to write scalable parallel programs which worked for 10 - 1000 processors with explicit message passing. By now, the number of cores in current big MPPs have gone beyond more than 100,000 (the Jaguar system at Oak Ridge National Laboratory has 150,152 cores). We now face the same problem for 10,000 - 100,000 processors.

Many-core accelerators add new levels to the parallel programming. One is a short vector processing and another is MIMD processing for 100 - 1000 FP units.

The short vector processing is actually 4-vector SIMD operations in Cell, GPUs, and SSE. In these processors, logically all arithmetic operations are performed as 4-vector operations in single precision (SP). To obtain a good performance, we need to modify our programs so that they do computations in 4-vectors. This vector processing is quite different from traditional vector processing because usually hand optimization in assembly language is necessary.

The combination of the short vector and MIMD processing requires us to rethink/restructure completely existing numerical algorithms if we want to utilize many-core accelerators.

To make the problem even worse, each accelerators have their own programming environment with slight differences in details. For example, from the point of view of their architecture, we can logically consider G80/G90/GT200b GPU from Nvidia and RV770 GPU from AMD/ATi to be very similar. They both consist of a sea of FP arithmetic units, which are connected with high speed memory, running in MIMD way. However, each company offers its own programming environment for general purpose GPU computing (GPGPU). While OpenCL or another language/library might offer a unification of the new two-level parallel programming in the near future, we expect that an unified approach is not always a right answer. In this paper, we introduce an alternative way of programming many-core accelerators. Specifically, we present a programming model for accelerators and parallel compiler based on it.

An important questions here 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}, the memory-wall problem is quite severe since the raw performance of these processors ($\sim$ a several 100 GFLOPS) is very high compared to the memory bandwidth. Applications which 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.

B. A Successful Accelerator for Many-particle Simulations

In astronomical many-particle simulations, the most time consuming part is the evaluation of mutual force between particles:

$$\boldsymbol{f}_i = \sum_{j=1}^{N} \frac{m_j(\boldsymbol{x}_i - \boldsymbol{x}_j)}{(|\boldsymbol{x}_i - \boldsymbol{x}_j|^2 + \epsilon^2)^{3/2}}, \quad (1)$$

where $\boldsymbol{x}_i$, 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). In GRAPE system, all calculations except computation of Eq.(1) are done on a host computer that controls GRAPE system. The host computer sends $\boldsymbol{x}_i$ and m_i to GRAPE and receives results $\boldsymbol{f}_i$. In other words, only the most computing intensive part of many-body simulations is computed on the specially developed component. Inside the GRAPE system, Eq.(1) is computed by force computing pipelines, each of which has more than 50 arithmetic units. The most recent GRAPE-6 chip has six such force computing pipelines. Thus, in total, one GRAPE-6 has more than 300 arithmetic units and provides 30 GFLOPS with only 13 W. This is still very efficient in spite of a fact that GRAPE-6 was developed 10 years ago. A recent commodity 4-core CPU (say, Intel Core i7) that consumes 100W provides at most $\sim$ 30 GFLOPS [4].

C. A Many-Core Accelerator: GRAPE-DR

In a new GRAPE system after GRAPE-6, instead of developing enhanced version of GRAPE-6 chip, GRAPE-DR (Greatly Reduced Array of Processor Elements with Data Reduction) chip, which is not fixed function for gravity but programmable, has been designed in 2005 followed by system construction in 2006-2009. At 2009 after the introduction of new GPUs designed for GPGPU, it turns out that GRAPE-DR is very similar to the recent GPUs. Like GPUs, it is also a many-core accelerator with 1024 FP arithmetic units. A half of the units are double precision (DP) addition units and another are SP multiplication units. Logically, one can program 512 add/mul units in SIMD way to do useful calculations (see [5] 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 system is programmable but not fully programmable unlike 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. Our programming model is originally designed for this type of computation. However, we found that the same programming model is useful for an evaluation of a multi-dimensional integral that has divergent nature.

D. A Difficult Problem: The Loop Integral

An evaluation of the Feynman path integral I arises in the particle physics. A simple example of such integral is an one-loop integral expressed as

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

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 [6]. They have reported that a combination of a multi-dimensional integration scheme and an extrapolation scheme on λ [7] along with at least quadruple precision (QP) operations is necessary to tackle to this problem. If we adopt the double exponential integral scheme, this integration is reduced to the three nested summation loop, 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 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.

If we implement QP operations with the scheme by [8] and [9], which utilizes DP units for emulation, a 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 one-loop integral is a highly time consuming task when we do it as a serial program.

A 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.

E. Plan of the Paper

Organization of our paper is as follows. In section 2, we summarize performance metric of commodity general purpose CPU and many-core accelerators with a special emphasis on the dependence of performance on numerical precision. All accelerators we consider are very good at SP operations but their DP performance is disappointing. Detailed analysis reveals that their QP performance again is remarkable. Section 3 is devoted to the explanation of our computing model. In section 4, we describe our compiler for many-core accelerators. Finally, we present performance results on GRAPE-DR and GPU obtained with our compiler in section 5 followed by our conclusion in section 6.

II. Performance Comparison of SP, DP, QP Operations on many-core accelerators

In this section, we briefly summarize performance characteristics of several many-core accelerators. Here, we make a special emphasis on the dependence of performance on the numerical precision. As a basis for comparison, we start the discussion with the performance of commodity general purpose CPUs.

We summarize the performance characteristic for a general purpose CPU and many-core accelerators in Table I.

A. Recent Multicore CPUs

In the last a few years, the performance of commodity general purpose CPUs are improving mainly through the increase of the number of cores on a chip. These CPUs are used not only for personal use but also as work horse of HPC clusters. The latest example of such CPUs is Intel Core i7 (Nehalem architecture). It has four cores running at 2.67 - 3.2 GHz and each core is capable of running two threads simultaneously. One can utilize each core for up to 4 SP operations in one cycle while the DP performance is one half of the SP performance. Accordingly, one Core i7 chip at 3.2 GHz offers the peak performances of 51.2 (SP) and 25.6 (DP) GFLOPS.

As already noted, theoretically, the estimated QP performance on a processor is no more than 5 % of its DP performance. We wrote routines in C language that do QP add and mul operations and have done a naive benchmark test. In this test, we have measured performance and latency of each operation on Core i7 processor running at 2.67 GHz. We have compiled the test program using gcc version 4.1.2 along with "-O2" optimization option. It turns out that the performance of QP add/mul is $\sim 73/58$ MFLOPS and QP add and mul operations require 36 and 46 clocks, respectively. We have inspected generated object codes and found that (a) add and mul QP routines use 34 and 37 instructions, respectively, (b) both routines use SSE (addsd/subsd/mulsd) instructions for DP operations, (c) the allocated number of registers in add and mul routines is 9 and 8, respectively. Additional instructions for both cases are mainly movsd instructions that move data between SSE registers. Note we used only one core and did not utilize SSE2 feature in this test so the expected peak QP performance is $2.67/20 \sim 134$ MFLOPS provided that each routine requires 20 DP operations. However, our test reveals that both operations require roughly two times longer latency than such ideal case. It seems that the main factor that causes the lower QP performance is the pipeline stall due to dependent operations. Precisely, both pipeline latency of FP operations (addsd/mulsd requires 3/5 clocks) and additional movsd instructions (latency 1 clock) affect total latency of QP routines.

B. Many-Core Accelerators

Many-core accelerators that we consider share same nature. Their SP performance is superior to DP performance. However, the ratio between DP and SP performance depends on architecture and implementation of a processor. It spreads from 0.5 for PowerXCell 8i and GRAPE-DR to 0.1 for Cell Broadband Engine, GT200b, and multiply on RV770. From these numbers, these accelerators can be divided into two groups. PowerXCell 8i, which is an variant of Cell Broadband Engine with the DP performance enhancement, and GRAPE-DR have been designed and developed for HPC applications while others with the lower ratio are for interactive processing of graphics. From a view point of architecture, we see that GRAPE-DR and GPUs are similar. They have more than 200 arithmetic units with relatively low clock cycle. On the other hand, both Cell products run at higher clock cycle with less than 50 units. We note that an important difference between GRAPE-DR and others is that external memory bandwidth is rather small (~ 4 GB s^{-1}) compared to that of GPUs and Cells (75 - 100 GB s^{-1}). GRAPE-DR is designed to do scientific applications that do not require high memory bandwidth.

We can estimate the QP performance of those accelerators using a method similar to the previous section. It is also true that at most an estimated QP performance is 5 % of their DP performance. Also, a further performance decline is expected due to the pipeline stall but it is quite difficult to estimate without a description of detailed architecture except for GRAPE-DR. For GRAPE-DR, we have implemented QP emulation codes as shown in later sections. Since GRAPE-DR executes FP operations completely in-order, we can calculate the fraction of time during which arithmetic units stall. In a real application with QP operations by GRAPE-DR presented in 4.3, the fraction is not significant. We estimate the peak performance of QP performance on GRAPE-DR is ~ 6.7 GFLOPS, which is 15 times faster than the estimated peak performance of Core i7. Also, the estimated peak performance of QP operations on RV770 is ~ 6.5 GFLOPS.

III. Computing Model

Based on experiences with GRAPE systems, we developed a simple computing model for GRAPE-DR.

	SP	DP	QP	$N_{\rm unit}^{\rm SP}$	$N_{\rm unit}^{\rm DP}$	clock	note
CPU	51	26	0.43	16	8	3.2	Core i7 SSE2/3
Cell1	200	20		24	6	3.2	Cell Broadband Engine
Cell2	230	109		32	16	3.2	PowerXCell 8i
GPU1	933	78		240	28	1.5	GT200b
GPU2	1200	240	6.5	800	128	0.75	RV770 (DP add)
GRAPE-DR	390	195	6.7	512	256	0.38	SING

TABLE I
SP, DP, QP PERFORMANCE AND CHARACTERISTICS OF A GENERAL PURPOSE CPU AND MANY-CORE ACCELERATORS ARE SUMMARIZED. A UNIT OF FLOATING POINT SPEED FOR SP, DP, QP ARE GFLOPS. QP SPEED SHOWN IN THE FOURTH COLUMN FOR CPU IS A RAW ESTIMATED PEAK VALUE BUT THE NUMBERS FOR GRAPE-DR AND RV770 ARE AN ESTIMATED PEAK SPEED EXPLAINED IN SECTION 2.2. THE FIFTH AND SIXTH COLUMNS SHOW A NUMBER OF SP AND DP ARITHMETIC UNITS, RESPECTIVELY. THE SEVENTH COLUMN INDICATES THE CLOCK CYCLE IN GHZ.

A. Massively Parallel Force Calculation

A general force calculation loop is expressed with the following equation:

$$s_i = \sum_j^N F(a_i, b_i, c_j, d_j ...), \tag{3}$$

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). 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.

B. A Problem of Programming the Force Calculation

At the first sight, one might think that such parallel problem is easily implemented on any parallel computers. That is not really true. In our experience of programming on various parallel computers, even for a simplest force calculation loop like Eq.(1), we always need to explicitly specify which part of our serial code can be parallelized to a compiler with detailed tweaks if we try to obtain a performance close to the peak speed. Two insights on programming of parallel problems we obtained so far are (1) there is no automagical way to do it and (2) given a parallel computer, there are best programming practises how to organize data and loop to get the best performance.

We believe the problem of existing environment/softwares/languages for parallel programming is the wrong level of abstraction. Most of softwares fall between following two extremes. At one extreme, the abstraction is rather high level such that loop-level parallelization adopted in compilers for vector computes and OpenMP. In another extreme, the abstraction is so low level such that we need to specify every 4-vector operations in short vector processing, e.g. assembly programming. For GPUs, both Nvidia and AMD/ATi offer programming environments (C for CUDA and Brook+) that fuse these two extremes like Chimera. A critical lack in existing parallel programming system in regarding our purpose of implementing a force calculation like Eq.(3) on many-core accelerators is that it seems usage pattern of data is not considered at all.

C. Our Solution

Suppose we implement a program to compute a general force calculation loop like Eq.(3). This can be simplely calculated by a nested loop as shown in Figure 1 (a). To map this nested loop on many-core accelerators, we unroll the outer loop as shown in Figure 2 (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. 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.

Actually, GRAPE-DR is designed to optimize to this unrolling technique as schematically shown in Figure 4. 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 `j`. This structure is the most efficient to do this type of computations.

In principle, an ideal compiler should be aware of the differences of data reuse. Practically, we let a user to specify a type of variables and our compiler automatically select the best practices on a given platform. In our compiler, we put details of how to organize/transfer/manupilate data on an accelerator and precise usage of a sea of FP units behind the scene so that we only need to worry about data sending to/receiving from the device.

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

Fig. 1. A simple nested loop to computer a general force calculation.

```
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])
```

Fig. 2. Unroll i-loop in 4 ways

IV. A Compiler for many-core accelerators

In this section, we describe the implementation of our compiler and present a brief summary of the syntax of a domain specific language we propose for describing compute kernels. At the end of this section, we show how our compiler is effective on many-core accelerators GRAPE-DR and GPU RV770.

A. Programming Model

In our programming model, a user need to write a separate compute kernel in our domain specific language. It is a user's task to extract a compute intensive loop in an own numerical code for off-loading to many-core accelerators. Also the user needs to modify the corresponding part of the original code to add calls to a several interface routines that control and handle data transfer to/from many-core accelerators. We think this explicit modification of the existing code is better approach in concerning performance because the user can easily optimize a way of data transfer. How data is transfered between a host computer and many-core accelerators is highly depending on a given problem.

In our language, we explicitly specify input (variables that are in the right hand side of Eq.(3)) and operations between those variables (precise definition of function F) to obtain

```
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])
```

Fig. 3. Unroll both i-loop and j-loop in 4 ways

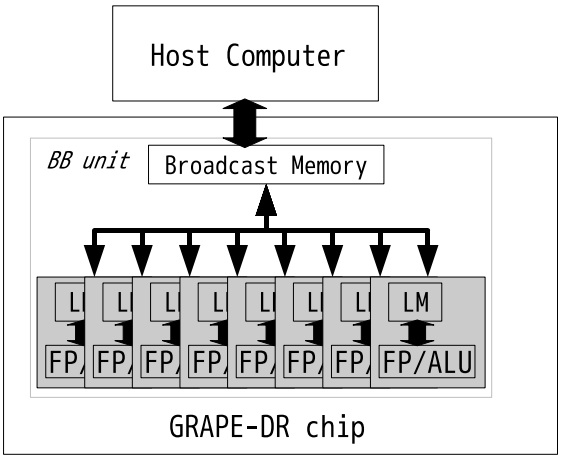

Fig. 4. Schematic view of the GRAPE-DR system. It consists of two parts: a host computer and GRAPE-DR chip(s). GRAPE-DR chip has 512 processing elements (PE). A PE is a unit of computing components in GRAPE-DR which has own local memory (LM) and arithmetic units (DP add, SP mul and integer ALU). Every 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.

output results (variables that are in the left hand side of Eq.(3)). Precisely, we divide the inputs into two types. One type is a *local variable* that is invariant during the inner loop. Another type is a *broadcast variable* that is different in each iteration. In Eq.(1), the local variables are components of a position vector $\boldsymbol{x}_i$ and ϵ and the broadcast variables are components of a position vector $\boldsymbol{x}_j$. The names of two types of inputs are originated from architecture of GRAPE-DR.

This language is easy to map to any many-core accelerators although it is originally designed to obtain a good performance for GRAPE-DR. As shown later, our compiler works very well for GPUs. We expect that our proposed programming model also shows good performance on a general purpose multicore CPU and new architectures such as Intel Larrabee.

B. Implementation

Our compiler consists of three stages; (a) frontend, (b) optimization on an internal representation and (c) backends that generate a target code.

In our system, a user explicitly specify input/output variables and arithmetic operations (hereafter it is called a compute kernel) on input variables. A main functions of the frontend is to parse the input source code and generate the corresponding abstract syntax tree (AST), and then to translate the AST into an internal presentation. At the same time, semantic information of variables is stored separately.

As the internal representation, we employed Low Level Virtual Machine (LLVM) [10] that is an open source compiler infrastructure actively developed. LLVM defines RISC-like instruction set (LLVM language) as a low-level code representation and offers various optimization modules on the

LLVM language. Moreover, it supports target code generation modules (both static and Just-In-Time code generators), C & C++ frontend, and utility functions for compiler development. We implemented our parser as a frontend to the LLVM infrastructure so that the input arithmetic operations and function definitions are translated into the LLVM language.

LLVM enable us to easily do following two complicated tasks. One is to do a several straight-forward optimizations that include constant propagation, common expression elimination, unrolling of loops, and inlining of functions. Another more important task is to employ the static code generator for x86 architecture to generate a verification object file. An example usage of the verification object file is as follows. Suppose we try to implement a part of our own program with our compiler, we first extract that part and write the input source to our compiler. After processing, the frontend and optimization module of the compiler generate the corresponding object files in both LLVM language and x86 machine language. Then one can replace the part in the original program with a call to the the generated x86 object file and test whether desirable results are obtained. This step verifies that (1) the input source code is correct and (2) the frontend and optimization passes work correctly.

Finally, the generated object file in LLVM language is converted to a target code at backends of the compiler. Since the generated object file is already optimized, backends simply translate each instruction into the corresponding instruction of the target. A unique feature of our compiler is that one can specify a target many-core accelerator and numerical precision (namely SP, DP and QP) as a compile option. The current version of the compiler supports following backends: SP, DP and QP for GRAPE-DR, SP, DP and QP for RV770.

In addition, we partially support mixed precision operations. This feature is useful when we need high precision operations only in a part of a whole calculation. For instance, in the gravity force equation Eq.(1), a calculation of a relative vector between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ is critical in terms of numerical accuracy. Given a pair of particles that are very close each other, the calculation involves possible a loss of trailing digits. A way to handle the problem, is that we use QP operations in the calculation of relative vectors and then convert the results into DP variables and we use DP operations in the remaining calculations. This method can greatly reduce a total number of operations.

C. Syntax of Our Language

Here, we present an example source code and describe how it works. Suppose we compute the following integral with the Simpson quadrature.

$$S = \int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}. \tag{4}$$

The corresponding source code is shown in Figure 5. The code consists of three parts; the definition of input/output variables (line 1 - 3), the definition of arithmetic operations (line 5 - 11), and the definition of functions (line 13-14 and 17-19). We specify the types of variables with a line starting with `LMEM`, `BMEM` and, `RMEM`, each of which defines input LM variables, BM variables, and output LM variables. Lines 5 - 11 represents the compute kernel of the code. One can use temporary variable as necessary without definitions. Functions are defined as a block starting with `function`. A return value of a function is a value in a last expression. Here, functions `sqrt` and `integrand` are user defined functions. On the other hand, the function `rsqrt` is a system function that computes the inverse of the square root.

An accumulation operation += is special and only valid for output LM variables. This operation implies that a compute kernel is processed as the iteration of inner-loop so that the output LM variable accumulate general force computed from input LM variables and BM variables. Every iteration, BM variables are properly updated. This accumulation mode is used for implementing the force calculating loop like Eq.(1). The sample source code is shown in Figure 6. As a result, in the example shown in Figure 5, we can not use operation += to accumulate the Simpson quadrature in the loop block defined in lines 7 - 10. Currently, a loop block only accepts fixed number of iterations so that the loop block is completely unrolled. Finally, the result of numerical integration are stored in the output LM variable `result` at line 11.

For each type of variables, we prepare corresponding interface routines. Specifically, we have routines that write input data to `LMEM` variables and read output data from `RMEM` variables and so on. For simplicity, each variables in any type is indexed in order of appearance in the compute kernel. In the source code shown in Figure 5, `LMEM` variables `a` and `b` are indexed `0` and `1`, respectively and `RMEM` variable `result` is indexed `0`. The index is used to specify a variable when we use the I/O routines along with other parameters such as number of data etc. Also, we prepare a few routines to control execution of the code on many-core accelerators.

D. Backends

After the processing at the frontend and optimization stages, a sequence of instruction in LLVM language are generated. At this point, the instructions only consist of basic arithmetic operations (addition, subtraction and multiply) and a few system functions (division and inverse of square root). A task of a backend is to translate instructions in LLVM language into a target code In both GRAPE-DR and RV770 backend, each LLVM instruction is implicitly converted to a corresponding 4-vector SIMD instruction. This implicit treatment is helpful for a user since the user does not have to describe every detail of SIMD operations.

1) GRAPE-DR: First, we introduce important characteristics of GRAPE-DR relevant to target code generation:

- It has 512 processor elements (PE).
- Each PE has a DP addition unit, a SP multiply unit and an integer arithmetic unit.
- FP units work as 4-vector SIMD operation
- Each PE has two-read/one-write register file (32 words in DP), one-read/one-write local memory (256 words in

```
 1 LMEM a, b;
 2 BMEM dx;
 3 RMEM result;
 4
 5 sum = 0.5*integrand(a) + 0.5*integrand(b);
 6 x = a;
 7 for(i, 1, 99) {
 8   x = x + dx;
 9   sum = sum + integrand(x);
10 }
11 result = sum*dx;
12
13 function sqrt(x) {
14   res = x*rsqrt(x);
15 }
16
17 function integrand(x) {
18   res = sqrt(1 - x**2);
19 }
```

Fig. 5. A source code for the numerical integration Eq.(4). The numbers at the beginning of each line indicate the line number.

```
LMEM xi, yi, zi, e2;
BMEM xj, yj, zj, mj;
RMEM ax, ay, az;

dx = xj - xi;
dy = yj - yi;
dz = zj - zi;

r1i = rsqrt(dx**2 + dy**2 + dz**2 + e2);
af = mj*r1i**3;

ax += af*dx;
ay += af*dy;
az += af*dz;
```

Fig. 6. A source code for the force calculation loop Eq.(1)

DP), and a temporary register.

- A write to the temporary register is also connected to a shortcut path for using the value at the next instructions.
- All instructions are executed in-order.

The backend for GRAPE-DR generates an assembly code for GRAPE-DR assembler. Thus, a separate invocation of the assembler with the generated assembly code is required to obtain machine code for GRAPE-DR hardware.

If the specified precision option is SP or DP, the backend directly translates simple arithmetic instructions (add, sub, mul) with corresponding GRAPE-DR instructions. Division (div) is treated separately as a combination of a reciprocal operation followed by a multiplication. Another backend's task is the calculation of living period of variables and register allocation. Our allocation strategy is simple. If living period is one, we assign the temporary register, and otherwise we assign register file as much as possible and if the register spills, we assign the local memory to remaining variable. At the final stage of the backend, it inserts idle (nop) operations to accommodate memory access coordination.

Generation of QP code for GRAPE-DR is somewhat different. Since QP arithmetic operations are emulated using DP operations, each basic operation is replaced with a sequence of the emulation code. Our QP emulation code for GRAPE-DR requires 21, 41 and 199 DP operations for QP add/sub, QP mul, and QP div operations, respectively. Note since GRAPE-DR only support SP multiply, the QP mul emulation code requires additional steps compared to an optimal sequence with DP multiply. Accordingly, the backend for GRAPE-DR QP code replaces four basic instructions with a corresponding code sequence. In QP code generation, we assign the local memory to all variables. Registers are reserved as temporary variables for QP emulations.

In either cases, the backend together with the assembler generates a skeleton code to be used with application programs. The skeleton code consists of API calls to initialize, send data to, execute the code with, and receive data from GRAPE-DR hardware.

2) RV770: The backend for RV770 works similarly but it's tasks are rather simple because the backend generates not architecture dependent code for RV770 but a target independent code that is called intermediate language (IL). IL is a part of the software development environment for GPU from AMD/ATi named compute abstraction layer (CAL). CAL is responsible for translating the IL code into the machine code of RV770. Therefore IL behaves like a virtual instruction set for GPU from AMD/ATi. With this methodology, our backend does not have to care about detailed architecture of GPU and possible future updates in architecture. Since CAL is also responsible for physical register allocation, our backend does not require register allocation.

Generation of SP/DP code for RV770 proceeds with a similar as SP/DP code generation for GRAPE-DR. One critical difference between GRAPE-DR and RV770 is internal structure of arithmetic units. A RV770 chip has 800 SP arithmetic units with capacity of multiply-add operation in one cycle. Internally, 800 SP units are divided into 160 groups where each group consists of a combination of 4-vector SIMD unit and a unit capable of special function like div, sin, cos etc. To maximize performance, it is best to utilize 4-vector SIMD unit as much as possible. So one way to make 4-vector SIMD unit busy is to unroll the inner loop of force calculation in 4 ways as shown in Figure 3.

We did an experiment to implement the force calculation loop Eq.(1) with the two schemes shown in Figure 2 and 3 (see [11] for details). In the following, they are called I-unrolling and IJ-unrolling schemes, respectively. In this experiment, we used RV770 running at 750 MHz. Highest computing speed obtained so far is ~ 200 GFLOPS with the I-unrolling scheme. This speed is comparable to previously reported speed of another GPU [12]. On the other hand, we obtained ~ 500 GFLOPS, which is about half of theoretical speed of the

system, with the IJ-unrolling scheme. Here, we assume that one force calculation requires 20 SP operations. The result with the latter scheme is the fastest ever reported with one GPU chip. Our backend for SP code for RV770 generates a code with the IJ-unrolling scheme by its default.

For QP code generation, we implement QP emulation code in IL code. Our QP emulation code in IL requires 20, 23 and 38 DP operations for QP add/sub, QP mul, and QP div operations, respectively. In generating QP code, the backend simply replaces each basic operations with a call to a corresponding emulation code.

V. Performance Results

Finally, we report a current status of the compiler and an obtained performance of sample applications. At the time of writing, not all supported backend is fully optimized so that here we report only interesting performance results.

A. Gravity in SP

Our first result is the performance of the force calculation loop Eq.(1) using SP on RV770. Our compiler generates a very efficient code from the source code shown in Figure 6. As already noted, we obtained about 50 % of the theoretical peak performance with the generated code. Actually, the peak performance is only available if all SP operations are multiply-accumulate operations. Except for matrix multiplication, such situation does not arise in practice. Therefore, we believe one half of the peak performance is quite high.

B. Feynman One-loop Integral in QP

Our next result is the evaluation of the Feynman one-loop integral shown in Eq.(2). We adopt the double exponential integration scheme to implement the one-loop integral. The definition of the integrand in the original Fortran code is written as 2 lines. With our language, the number of lines is 9 including definitions of variables. We test the integral on both GRAPE-DR and RV770. After processing with the GRAPE-DR QP backend, the generated assembly language contains 1079 instructions, which includes 34 nop operations. In other words, a percentage of time during which arithmetic units are stalled is as small as 3 %.

We run this code on one GRAPE-DR chip running at 380 MHz and obtain the following results; depending on the size of N, the measured computing speeds are $\sim 2.67, 3.85, 4.80$ and 5.46 GFLOPS for $N = 256$, 512, 1024, and 2048, respectively. For RV770, we do same tests and obtain $\sim 6.43, 7.14, 7.46$ and 7.57 GFLOPS for $N = 256$, 512, 1024, and 2048, respectively, with one RV770 chip running at 750 MHz. From our tests with SP operations described in the last section, we expect that the IJ-unrolling scheme is also effective in this case. Further investigations will be required to how performance is affected by precise details of the generated code.

C. Hermite scheme in QP and mixed QP and DP

Our final application is calculation of Eq.(1) using QP operations. Evaluation of gravity force in QP is desirable to test three body scattering processes, which are thought to be frequently occurred in star clusters and planetesimal. Also, QP evaluation of gravity will be a necessary tool in future simulations of a star cluster with high fraction of binary stars In a such star cluster, smallest scale length is $\sim 10^6$ cm while the size of typical star cluster is $\sim 10^{21}$ cm. Apparently, direct evaluation of gravity in DP brings large error. Here, we report the implementation of the Hermite scheme, which is a standard integration scheme in modeling a star cluster [13]. The Hermite scheme require us to compute a time derivative of acceleration in addition to Eq.(1).

We generated the QP code, which computes both acceleration and its time derivative, for RV770 and did benchmark tests. The measured computing speeds with one RV770 chip running at 750 MHz are $\sim$ 4.40, 4.68, and 6.31 GFLOPS for $N = 1024$, 2048, and 4096, respectively. For the first time, our compiler enable us to do astronomical many-particle simulations in QP with a feasible computing speed.

In this particular problem, a gain using mixed QP and DP operation is attractive. We modified the original code to use QP operations only in the calculation of relative vectors and the summation of the acceleration. Accordingly, only 6 add operations in the whole calculation are done with QP. Note the whole operations in this case consist of 21 add, 19 mul, 1 div and 1 sqrt operations. The measured computing speeds with the same configuration are $\sim$ 10.3, 19.8, and 27.8 GFLOPS for $N = 1024$, 2048, and 4096, respectively. Even with this modification, we see negligible change in integration error. This result is quite promising to use our compiler for simulating the evolution of a highly dense star cluster.

VI. Conclusion

In this paper, we introduce a newly developed compiler 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. The moderate level of abstraction combined with an explicit specification of type of variables enable our compiler to generate highly efficient codes for many-core accelerators GRAPE-DR and RV770 GPU. Another novelty of the compiler is that it is possible to choose a desirable numerical precision, SP, DP, QP or combination of those precisions. The performance results of SP and QP operations on GRAPE-DR and RV770 GPU obtained so far are remarkable. We conclude that our programming model and compiler is highly effective to many-core accelerators.

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] K. Nitadori, J. Makino, and P. Hut, "Performance tuning of N-body codes on modern microprocessors: I. Direct integration with a hermite scheme on x86_64 architecture," *New Astronomy*, vol. 12, pp. 169–181, Dec. 2006.
[5] J. Makino, "Specialized Hardware for Supercomputing," *SciDAC Review*, pp. 54–65, 2009.
[6] 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.
[7] P. Wynn, "On the convergence and stability of the epsilon algorithm," *SIAM Journal of Mathematical Physics*, vol. 3, pp. 91–122, 1966.
[8] D. Kunuth, *The Art of Computer Programming vol.2 Seminumerical Algorithms*, 1st ed. Reading, Massachusetts: Addison Wesley, 1998.
[9] T. Dekker, "A Floating-Point Technique for Extending the Available Precision," *Numerische Mathematik*, vol. 18, pp. 224–242, 1971.
[10] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
[11] 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
[12] 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.
[13] J. Makino and S. J. Aarseth, "On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems," *Publication of Astronomical Society of Japan*, vol. 44, pp. 141–151, Apr. 1992.