GRAPE-DR: 2-Pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing

Junichiro Makino
Center for Computational Astrophysics, National Astronomical Observatory of Japan, 2-21-1 Ohsawa, Mitaka, Tokyo 181-8588, Japan
makino@cfca.jp

Kei Hiraki
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo, Tokyo 133-0033, Japan
hiraki@is.s.u-tokyo.ac.jp

Mary Inaba
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo, Tokyo 133-0033, Japan
mary@is.s.u-tokyo.ac.jp

ABSTRACT

We describe the GRAPE-DR (Greatly Reduced Array of Processor Elements with Data Reduction) system, which will consist of 4096 processor chips each with 512 cores operating at the clock frequency of 500 MHz. The peak speed of a processor chip is 512Gflops (single precision) or 256 Gflops (double precision).

The GRAPE-DR chip works as an attached processor to standard PCs. Currently, a PCI-X board with single GRAPE-DR chip is in operation. We are developing a 4-chip board with PCI-Express interface, which will have the peak performance of 1 Tflops. The final system will be a cluster of 512 PCs each with two GRAPE-DR boards. We plan to complete the final system by early 2009.

The application area of GRAPE-DR covers particle-based simulations such as astrophysical many-body simulations and molecular-dynamics simulations, quantum chemistry calculations, various applications which requires dense matrix operations, and many other compute-intensive applications.

1. INTRODUCTION

SIMD parallel processing is a very old idea, with several successful implementations such as Illiac-IV[2], ICL/AMT DAP, Goodyear MPP[8], TMC CM-1/2[6], and INFN/Quadrics APE-100, APEmille and apeNEXT. These are large machines made of up to 64K processors, each with its own local memory. Except for the APE machines which were designed for LQCD calculation, all of these machines were built before 1990.

There is another form of the SIMD architecture. Almost all recent microprocessors have some form of SIMD processing units, with 4 or more arithmetic units. These include VIS of SPARC, Altivec of PowerPC, MVI of Alpha, 3DNow! of AMD and MMX and SSEx of Intel x86.

Though both of these two architectures are called "SIMD", the actual hardware implementation and programming models are completely different. In the former case of large-scale SIMD parallel machines, each processing element has its own memory and the address generation unit for it, and they are connected through some routing network. In the latter case, the instruction set of a single processor is extended to handle a single long word as a vector of multiple short words. Thus, essentially only the arithmetic units are duplicated, and they are connected to single memory unit (or single L1 cache) through a single datapath.

The former architecture is for large machines made of a number of processing chips, each with one or a few processors. The latter is for a single processor chip. With present-day technology we could in principle integrate thousands or more of processors used in machines like CM-1 into a single chip, and yet SIMD extensions currently use just a few arithmetic units.

We have completed the development of a large-scale SIMD processor chip, which we call GRAPE-DR (Greatly Reduced Array of Processor Elements with Data Reduction). A single GRAPE-DR chip, fabricated with TSMC 90nm process, integrates 512 processors, each with double-precision floating-point arithmetic units. It operates with 500MHz clock frequency and the peak performance of a single chip is 512 Gflops (single precision) or 256 Gflops (double precision). We are currently building a large parallel machine with these GRAPE-DR chips. The final machine will be completed by early 2009, and its peak performance will be 2 Pflops (single precision) or 1 Pflops (double precision).

In this paper, we describe the idea behind the GRAPE-DR processor, its detailed architecture, the programming model, measured performance on a few applications, application areas, and future plans.

In section 2, we discuss the limitation of past and present SIMD architectures and in sections 3 and 4 we describe how GRAPE-DR architecture solved the problems. In section 5 we describe the details of the GRAPE-DR chip and in section 6 current status of hardware development. Finally, in section 7 we compare the GRAPE-DR architecture with

SC07 November 10-16, 2007, Reno, Nevada, USA

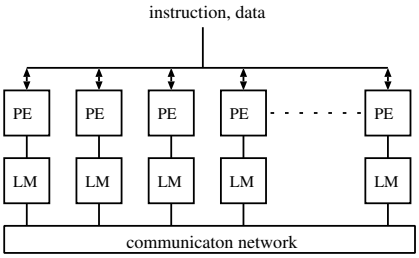

Figure 1: Structure of a traditional large-scale SIMD processor

those of several other "many-core" processors.

2. LIMITATIONS OF LARGE-SCALE SIMD MACHINES

As we described in the previous section, large-scale SIMD machines have become extinct in early 1990s. One can think of many reasons, but one crucial one, in our opinion, is simply that with traditional SIMD architecture it has become impossible to make effective use of the advance of the semiconductor technology.

Figure 1 shows the basic architecture of an SIMD processor. A number of processor elements (PEs), each with its local memory (LM), execute the same instruction. They can communicate with each other through some communication network. Many machines had simple mesh connections. Some machines had more complex networks such as the hypercube.

The main advantage of an SIMD machine compared to an MIMD machine is that many processors can share single control logic for instruction fetch, decode etc. Thus, one can make larger number of processors from the same amount of transistors or silicon real estate, and until late 1980s one could make large-scale SIMD machines with number of processors much larger than that of MIMD machines for similar production cost.

However, such an advantage was lost in early 1990s, when a single LSI became large enough to integrate complex microprocessors with fully pipelined floating-point arithmetic units. At that point, the bandwidth that floating-point arithmetic units required became much more than that of the external memory, and even for a single arithmetic unit a cache memory (or a complex hierarchy of caches) became necessary. If even a single arithmetic unit requires a cache, it is clear that increasing the number of arithmetic units does not improve the overall performance. In other words, memory bandwidth limits the performance of an SIMD processor chip. Moreover, the SIMD architecture implies that the instruction stream must be supplied from an external control unit and broadcasted to the entire machine, which is even more difficult than offering the bandwidth to the local memory. So the instruction bandwidth is also a problem.

To summarize, it has become impossible to design a large scale SIMD machine, with reasonably high clock frequency and reasonably large number of processor elements integrated into a chip. This is the basic reason why large-scale SIMD machines have become extinct.

The current SIMD extensions of microprocessors solved the latter problem of the instruction bandwidth by limiting the range of arithmetic units which works in the SIMD fashion to a single processor core in a single chip, but did not really solved the memory bandwidth problem. In current SIMD extensions, SIMD execution units access the data in the L1 cache through multi-word load/store instructions. Thus, as far as the data is in the L1 cache, high performance can be achieved. However, if the data is not in the L1 cache, the performance is limited by the bandwidth of the memory (or L2 or higher level cache), which is much less than that of the L1 cache. Current microprocessors thus have only a modest number of floating point units, even for the case of multi-core chips.

However, the requirement for memory bandwidth depends critically on the type of applications, and there are a number of important applications which require rather small memory bandwidth. One example is particle-based simulations. In particle-based simulations, one particle interact with many other particles at each timestep. Thus, calculation cost per particle is fairly large, while the amount of data for one particle is tiny. The extreme example is astrophysical many body simulations with the $O(N^2)$ direct interaction calculation algorithm, where N is the number of particles. With the direct interaction calculation, the gravitational force on one particle is calculated as the sum of all $N-1$ forces from other particles. Therefore, we can use various blocking techniques to reduce the requirement for memory bandwidth. In the case of astrophysical many-body simulations with $O(N \log N)$ or $O(N)$ methods, calculation cost is much smaller, but we can still use blocking techniques. The same is true for classical molecular dynamics simulations.

Another example is quantum chemistry calculations, where the calculation of two-electron integrals and the diagonalization of dense matrices are among the most costly operations. Both are compute-intensive operations on small amount of data. In general, most operations on dense matrices can be rewritten in such a way that the matrix-matrix multiplications become the most time-consuming part, and it is easy to reduce the memory bandwidth requirement of matrix-matrix multiplications by blocking.

It is probably more useful to list applications which require very high memory bandwidth and thus not suitable for our SIMD architecture. One is large scale hydrodynamics simulations (CFD) on structured grid and relatively simple low-order explicit integration schemes. Another example is applications which requires FFT of very large dataset, such as CFD with spectrum methos or plane-wave expansion methods for band-structure calculations. In both cases, the number of arithmetic operations per memory access is intrinsically small.

For many other applications, we can reduce the memory bandwidth requirement by various techniques, and we might be able to use an SIMD processor with large number of floating point units and small memory bandwidth effectively. In the next section, we describe the architecture of such an SIMD processor chip we developed.

3. GREATLY REDUCED ARRAY OF PROCESSOR ELEMENTS

Figure 2 shows the basic architecture we discuss in this paper. It consists of a number of processing elements (PEs),

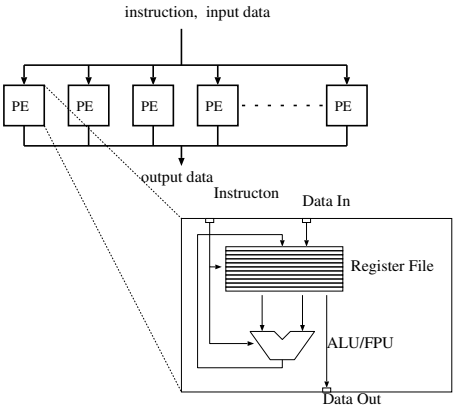

Figure 2: Basic structure of an SIMD processor

each of which consists of an FPU and a register file. They all receive the same instruction from outside the chip, and perform the same operation.

Compared to the classic SIMD architecture such as that of TMC CM-2, the main difference are the followings.

a) PEs do not have large local memories.

b) There is no communication network between PEs.

We introduce these two simplifications so that a large number of PEs can be integrated into a single chip. If we want to have a large memory connected to each PE, the only economical way is to attach DRAM chips. However, once we decide to use external memory chips, it becomes very difficult to integrate large number of processors into a chip. Consider an example of a chip with 100 processors, each with one arithmetic unit. If the clock frequency is 1 GHz, the peak speed of the chip is 100 Gflops. If we want to add the memory units which can supply one word per clock cycle to these 100 processors, we need the memory bandwidth of 800 GB/s, or around 100 times more than that of the latest microprocessors, which is clearly not practical. If we eliminate the local memory of processors, we can integrate very large number of processors into a chip.

A communication network is not very expensive, as far as it is limited into a single chip. A two-dimensional mesh network would be quite natural, for physically two-dimensional array of PEs on a single silicon chip. However, such a two-dimensional network poses a very hard problem, if we try to extend it to multi-chip systems. (see figure 3) With current and near-future VLSI technology, it is possible to integrate more than 1000 PEs to a single chip, each with fully pipelined FPUs. Thus, a 2D array will have the dimensions of 32 by 32, and the minimum number of external links necessary to construct a 2D network is $32 \times 4 = 128$. If we want to have, say, 16 wires per link, the total number of pins necessary is 2,048. To make such a large number of pins work with a reasonable data rate is not impossible, but very costly.

If we eliminate the inter-PE communication network right from the beginning, we have no problem in constructing multi-chip systems, since PEs in different chips need not be connected.

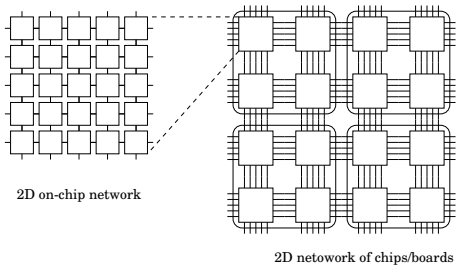

Figure 3: On-chip two-dimensional network (left) and its extension to multiple chips and multiple boards (right)

Thus, this simple architecture has two advantages. First, we can integrate a very large number of PEs into a single chip. Second, it is easy to construct a system with multiple chips. As a result, we can construct a system with very high peak performance.

Important question here is if any real application can actually take advantage of this architecture. We consider several examples and extend the basic architecture in the next section.

Note that this SIMD processor works as an attached processor to general-purpose CPU. Since the on-chip memory is limited to just the register files of PEs, the SIMD processor itself cannot run any application which requires large amount of memory. Thus, we need to move only the part of the application program which can be efficiently done on the SIMD processor. This of course means there will be communication overhead.

Before we proceed to the next section, we need a name for the proposed architecture. Since the main difference between the proposed architecture and previous SIMD architecture is the removal of elements like local memory and inter-PE network, we call this architecture Greatly Reduced Array of Processor Elements, or GRAPE. The similarity of this name to the GRAPE for astronomical N-body simulations[9] is a pure coincidence.

4. MODIFICATION TO THE BASIC ARCHITECTURE

4.1 Particle-based simulations

In many particle-based simulations such as classical molecular dynamics or astrophysical N-body simulations, the most expensive part of calculation is the evaluation of particle-particle interactions. In general, it has the form

$$f_i = \sum_{j \neq i} g(x_i, x_j), \tag{1}$$

where x_i denotes the variables associated with particle i, $g(x_i, x_j)$ is some generalized "force" from particle j to particle i, and f_i is the total "force" on particle i. At least formally, the summation is taken over all particles in the system. Therefore the calculation cost is $O(N^2)$, where N is the total number of particles in the system. In some cases the interaction is of short-range nature and we can apply some cut-off length. If the interaction is long-ranged, we might be able to use approximate algorithms such as FMM[4] or

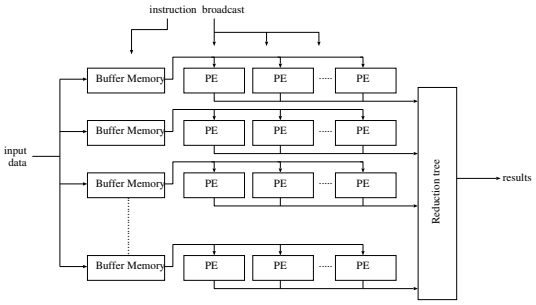

Figure 4: Modified SIMD architecture

Barnes-Hut tree[1].

In these cases, however, the most expensive part is still the evaluation of equation (1). The basic SIMD structure we discussed in the previous section is quite suited to calculations of this type. In the simplest case, we first load data of particles on which we calculate the interaction to the registers of PEs. In other words, we first write one x_i to each PE. Then we broadcast one x_j to all PEs and let them calculate the force from this particle j to their particles. We repeat sending particles x_j until all particles are sent, and then read the calculated results f_i in PEs. Remaining calculations such as the time integration of particles are done on the host computer which controls the SIMD processor.

If the number of particles is much smaller than the number of PEs, the efficiency would become low. Even when the total number of the particles is large, if the interaction is short-ranged, the number of particles with which one particle physically interact can be much smaller than the number of PEs.

This problem can be solved in many different ways. One possibility is to organize the processors into blocks, as shown in figure 4.

In this modified architecture, PEs are organized into blocks, each with small buffer memory. These blocks are connected to a reduction network. The host computer can either write data to individual buffer memories or broadcast the same data to all buffer memories. In this way, PEs in different blocks can calculate the forces from different particles. In addition, the reduction network allows multiple PEs in different blocks to calculate the force on the same particle from different particles. Thus, the efficiency for small-N systems or short-range force is greatly improved. In the following, we call these blocks of PE as broadcast blocks (BBs) and the buffer memory as broadcast memory (BM).

Note that the hardware cost of the buffer memory and reduction network is very small, since their cost is proportional to the number of blocks, which is a small fraction of the number of PEs.

4.2 Dense Matrix operations

For any dense-matrix algorithms, the basic operation is matrix multiplication $C = AB$. Thus, the key question is if our proposed architecture can achieve reasonable performance for matrix multiplication.

We consider the modified architecture discussed in the previous section. In this architecture, it is easy to implement parallel matrix multiplication. In the following, we assume that the number of PEs in a group and the number of groups in a chip is both n. Thus, a chip has n^2 PEs. The basic idea is to block-subdivide the matrix A into $n \times n$ sub-matrices in the same way as in the standard Canon's algorithm and load them to each PE. Then we take one column of B and divide it to n pieces, and send these pieces to the broadcast memories. We then calculate $c = Ab$ on each PE on each block. By taking summation over blocks, we obtain one row of C.

To be more specific, let us assume that A is a square matrix of size nm. We subdivide A into submatrices A_{ij} of size m, where m should be small enough that m^2 words can fit to the local memory of each PE. Here, A_{ij} is stored to PE i of BB j. We consider the multiplication of single column of B, which is a vector b with length mn. The vector b is again divided into n pieces, and for each j, b_j is broadcasted to all PEs of BB j. Then, PE i of BB j performs the multiplication $A_{ij}b_j$. The results are partial sum of c_i, for each of PE i of all BBs. Thus, by taking the summation of these partial c_i sums over all BBs, actual sum c_i is calculated. Summation is done by the reduction network.

4.3 Two-electron integral

The evaluation of two-electron integrals is simply a rather long calculation from small number of input data, resulting in essentially a single number, and a very large number of them can be calculated in parallel. Thus, original simple architecture without reduction network is enough. Since the modified architecture can be used to emulate the original one, the modified architecture can be used.

4.4 Greatly Reduced Array of Processor Elements with Data Reduction

We have changed the basic SIMD architecture by making the blocks of PEs and adding the buffer memories and the reduction network. Therefore, we call this architecture GRAPE-DR, or Greatly Reduced Array of Processor Elements with Data Reduction.

5. DESIGN OF THE GRAPE-DR CHIP

In this section, we overview the design of the first chip based on the GRAPE-DR architecture, the GRAPE-DR chip. It integrates 512 simple processing elements (PEs), organized into 16 broadcast blocks. Each PE can do one floating-point addition and one multiplication in single precision per clock cycle, or one addition and one multiplication in double precision in every two clock cycles. The clock frequency is 500MHz and the theoretical peak performance is 512Gflops in single precision and 256 Gflops in double precision.

5.1 PE architecture

Since each PE is programmable, we have to define its instruction set architecture, in other words, how we write programs for PEs. We designed the PE architecture so that the hardware is simple and yet can achieve high performance.

From the hardware point of view, PE must be fully pipelined, with fixed number of stages which does not depend on the data. This requirement of fixed number of stages comes from the SIMD nature that all PEs must proceed in lockstep. For simplicity, we construct PEs so that all operations, integer or floating-point, have the same number of stages.

With this pipeline processor of fixed stages, the result of one operation becomes available after a fixed number of stages, independent of the types of the operations executed. A simple way to take advantage of this predictability is to define a vector instruction set, with the vector length same as the number of pipeline stages. In this way, each vector instruction can use the result of the previous instruction, and we can eliminate the need for the instruction scheduling. Additional advantage of the vector-mode instruction is that the communication bandwidth for the instruction stream is reduced by the factor same as the vector length. In our first implementation, we use the vector length of four.

When we use this vector instruction set for the calculation of particle-particle interaction, the simplest way to write the program is to use one PE to calculate the forces on four particles (here four is the vector length). This means the effective number of PEs, from software point of view, becomes larger by a factor of four, and we probably want to make the number of blocks introduced in the previous section larger by the same factor, to keep the effective number of particles to be processed in parallel small. Note that this can be achieved without increasing the communication bandwidth, since the communication bandwidth depends only on the total processing speed and the number of i-particles.

The use of the vector instruction set also implies the size of the register file must be large enough in order to keep all intermediate data. However, for many applications the requirement is anyway small. Therefore the impact of the vector mode on the size of the register file is rather small.

The fact that we adopted the vector mode instruction and reduced the bandwidth requirement for the instruction stream means that we need not be too clever in reducing the length of the instruction word. So we adopted the horizontal microcode itself as the instruction word. An instruction word consists of all the necessary control bits for all components, and these control bits are directly supplied to the hardware, after adequate numbers of delay stages.

Figure 5 shows the architecture of a PE of the GRAPE-DR chip. A PE consists of a floating-point adder, a floating-point multiplier, an integer ALU, a three-port general-purpose register file (GP reg), a single-port local memory, and an additional dual-port working register (T register). The T register is used store temporary values. The local memory augments the size of the register file. The address generator for the local memory supports the indirect addressing, by arrowing the content of the T register to be used as the address of the local memory. It also supports constant-stride access during vector operations. Storing of the results to memory units (GP reg and local memory) can be controlled by mask registers. Mask registers can store the flag output of the integer ALU and the floating-point adder.

Each PE has two fixed inputs, PEID and BBID. PEID gives the index of PE within its broadcast block, while BBID gives the index of the broadcast block itself. Using these fixed-number inputs and mask registers, we can control individual PEs independently.

The broadcast memory is dual-ported. With the current GRAPE-DR design, the data in the broadcast memory can be written directly to all of GP register, T register and the local memory, while only the data in the GP register can be transferred to to the broadcast memory.

The basic data format is a 72-bit floating-point format,

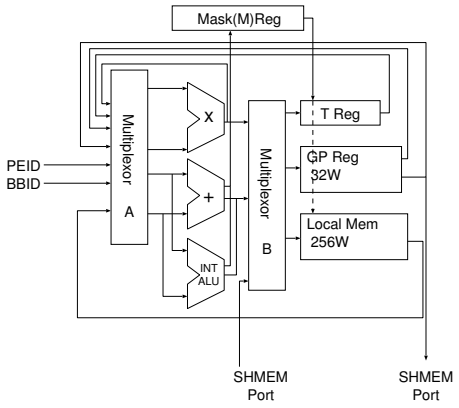

Figure 5: Structure of a Processor Element

with 1-bit sign, 11-bit exponent and 60-bit mantissa. We call this format double-precision. It also supports single-precision format with 24-bit mantissa. The integer ALU operates on 72-bit integers. The floating-point adder unit also work in 72-bit double-precision data, but it has the flag to round the output to single-precision format. Also, it has the flag to handle unnormalized numbers, for both the input and output.

The implementation of the floating-point multiplier is a bit complex. The multiplier is designed to handle input of up to 50-bit mantissa (not 60 bits). The reason is that for many applications accuracy of addition/subtraction is more critical than that of multiplication. Therefore we used 60-bit mantissa for addition/subtraction and 50-bit mantissa for multiplication.

The actual multiplier array has one 50-bit port (port A) and one 25-bit port (port B), and generate 75-bit output. This 75-bit result is then rounded to either 60-bit or 24-bit format. Thus, multiplication of two single-precision words can be done with throughput of one result per cycle. For double-precision multiplication, we first perform the multiplication between port-A input and upper 25 bits of port-B input, and then perform the multiplication between port-A output and lower 25 bits of port-B input, and then add then using the floating-point adder unit. Thus, while the double-precision multiplication is performed, floating-point adder unit is occupied for half of the time.

The integer ALU can perform most of basic integer arithmetic and logical operations, including shift operations.

5.2 Chip architecture

Figure 6 shows the overall architecture. We so far showed PEs as forming a two-dimensional grid, but from hardware point of view it is more appropriate to regard the structure as a two-level hierarchy. The chip consists of multiple broadcast blocks (BBs) of PEs. Each block consists of PEs and a broadcast memory (BM). All BBs receive the same data and instruction from outside the chip. The outputs of BBs are reduced by the reduction network.

All communication to and from PEs are through BMs. To write a data to one PE, first we write that data to the BM, and then transfer it to PE's memory (or registers). To read out the data in a PE, first we let that PE to transfer the data to BM, and then use the reduction network to output

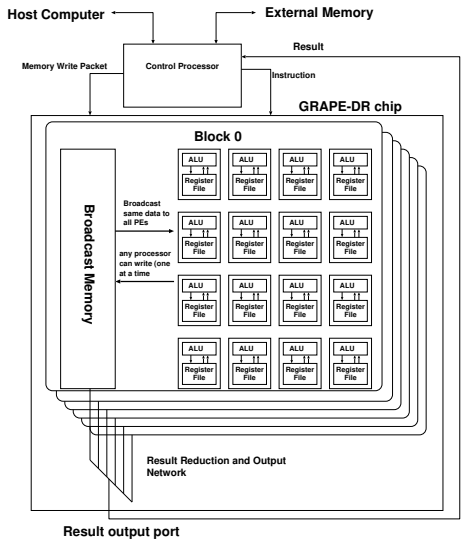

Figure 6: Overall Architecture of the GRAPE-DR chip

the data.

The reduction network has the binary tree structure, and each tree node has the floating-point adder and integer ALU of the same design as those of PEs. Thus, we can apply many different reduction operations, such as summation, multiplication, max, min, and, or etc.

5.3 Programming GRAPE-DR

In the case where we use this new GRAPE-DR processor as the replacement of traditional special-purpose GRAPE chip for particle-particle interaction calculation, programming is not a very large issue. We can simply write down the microcode for the gravitational force calculation, and communication library routines etc. This is not much different from the softwares necessary for traditional GRAPE hardwares. The only difference is that we also need to write the microcode, which is just several tens of lines.

For other kind of particle-particle interactions, the development process is quite similar to that for gravitational force calculation, and much of the communication library codes can be recycled. Thus, it would be more efficient to let some software generate the communication library from higher-level specifications, in the way similar to the PGPG system [5]. Also, writing the horizontal microcode by hand is hard, even for just a few lines. We have developed a simple symbolic assembly language, in which the program is written in a more or less human-readable way. Compiler languages are also under development [7].

Complete examples of the assembly language code and compiler language code is given in Appendix.

5.4 Implementation details of the GRAPE-DR chip

In this section, we summarize the characteristics of the first hardware based on the GRAPE-DR architecture, the GRAPE-DR chip. The GRAPE-DR chip was fabricated using 90 nm process and integrates 512 processors. The general-purpose register has 32 words and local memory 256 words.

One BB consists of 32 PEs and a 1024-word dual-port BM. One chip consists of 16 blocks, and the input port of the chip can accept one double-precision word per clock cycle. The throughput of the output port is one word per every two clock cycles.

The chip operates on the clock cycle of 500 MHz and offers the peak speed of 512 Gflops for single-precision operations and 256 Gflops for double-precision operations. Input data bandwidth is 4 GB/s and output 2 GB/s.

5.5 Parallel GRAPE-DR system

So far, we have discussed the design of a single chip. In practice, in order to achieve a reasonable performance, it is necessary to use many of these chips for one application. In other words, we need to discuss how to construct a large parallel system.

Here, we continue the approach we used for previous GRAPE hardwares[3]. The GRAPE-DR hardware will be designed as a relatively small attached processor for UNIX/Linux running workstations or PCs, and large parallel systems will be constructed just by assembling large PC clusters in which each node is connected to small GRAPE-DR hardware. This approach has many practical advantages. Most important one is that we can keep the speed ratio between the PC host computer and GRAPE-DR relatively small (around a factor of 1000 or less), which greatly simplifies the requirement for the application software. Parallelization is handled on the side of the host computer and GRAPE-DR would not have any special hardware/software to support parallelization.

One GRAPE-DR card will house 4 processor chips, each with its own off-chip memory. The data transfer speed between the host and GRAPE-DR card can be the bottleneck, but current fast interface standards like 8-lane PCI-Express would offer reasonable bandwidth, at least for the current GRAPE-DR chip.

We plan to complete a 4096-chip system by early 2009. It will have the theoretical peak performance of 2 Pflops for single precision and 1 Pflops for double precision. Most likely, it will be a 512-node system each with two GRAPE-DR cards.

6. DEVELOPMENT STATUS

6.1 Hardware

We finished the physical design of the GRAPE-DR chip by the end of 2005, and received the first sample chips in May 2006. Top panel of figure 7 shows the top-level layout image of the chip. Each white square is one PE. The die size is 18mm by 18mm.

We have developed the GRAPE-DR test board (see the bottom photo of figure 7), which houses one GRAPE-DR chip around the same time and confirmed the operation of the chip with both the test vectors and for real applications. The test board consists of one GRAPE-DR chip, one FPGA chip (Altera Stratix II), and one memory chip. The interface to the host is PCI-X, and we used the IP core from PLDA. Figure 8 shows the block diagram of the GRAPE-DR test board.

We have finished the development of the second board with PCI-Express interface and large on-board memory with DDR2 DRAM.

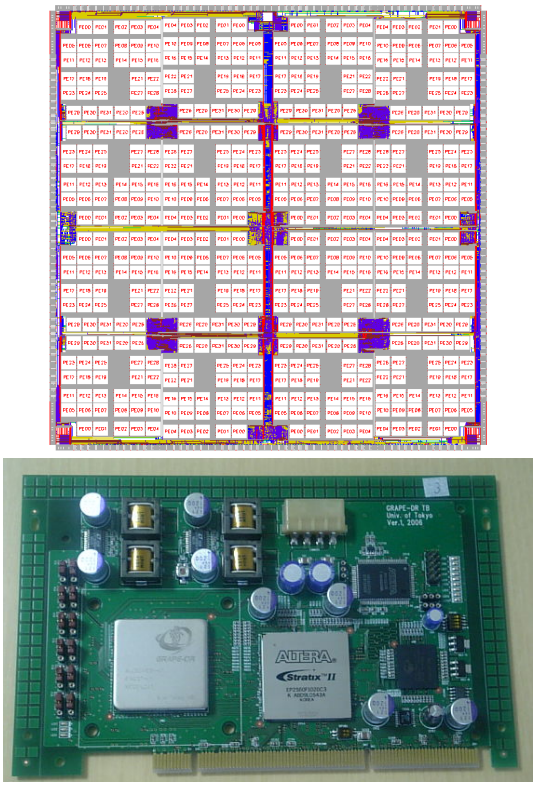

Figure 7: GRAPE-DR chip layout (top) and GRAPE-DR test board (bottom)

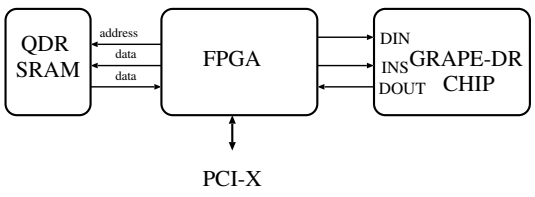

Figure 8: GRAPE-DR chip test board block diagram.

Table 1: Applications tested on the hardware

application	assembly code steps	asymptotic speed	measured speed
simple gravity	56	174 Gflops	50 Gflops
gravity and time derivative	95	162 Gflops	—
vDW force	102	100 Gflops	—

The measured maximum power consumption of the GRAPE-DR chip was 65W.

6.2 Software

So far, we have implemented the following applications

- Gravitational N-body calculation (simple one and that for Hermite integration scheme)
- Molecular dynamics calculation with van der Waals potential
- Parallel integration of three-body problems
- Matrix multiplications
- Simplified two-electron integral calculation

The first two have been actually run on the hardware. Table 1 lists the number of instructions in the loop body, estimated asymptotic performance of the single-chip board when we ignore the communication between the host and the board, and actual measured performance of the board (only for one gravity code).

For gravitational force calculation, around 50Gflops was measured for integration of 1024-body system. Currently, we use the on-chip memory of FPGA as the on-board memory, which limits the size of the memory. For larger number of particles, the performance close to the peak could be achieved, even with current relatively slow PCI-X interface. Since the purpose of this test board is to test the chip itself, we have not tried much to achieve very high performance. For the same reason, we do not give the measured performance for two other codes.

Other applications have been tested on software chip simulator. We plan to run these applications after a second board with large on-chip memory becomes available, since they do require large memory for either data (matrix multiplication) or code (others).

7. DISCUSSION

7.1 Comparison with previous SIMD machines

Since all PEs on GRAPE-DR execute the same instruction stream, it belongs to the SIMD architecture. There have been a number of projects to develop large-scale parallel computers for scientific computing based on SIMD architecture. Examples are Illiac IV, Goodyear MPP, ICL DAP, TMC CM-2, Maspar MP-1, IFN/Quadrics APE machines.

There are, however, several important differences between the design of GRAPE-DR and that of previous SIMD machines. The first one is the size of the local memory for PEs. All previous SIMD machines are designed so that the main

body of the application program runs on the SIMD processor array. Therefore the combined capacity of the memories local to PEs must be large enough to run actual applications. However, this requirement for large local memory makes the efficient design of an SIMD processor array practically impossible with the current VLSI technology, as we discussed in section 2. Except for APE machines, SIMD machines listed above were designed and built before 1990, which was the time when it become possible to integrate multiple floating-point units into a single chip.

In the case of GRAPE-DR, we solved this problem of the local memory bandwidth by limiting the size of the local memory so that it can be integrated into one chip. This means that the way the application program use the GRAPE-DR hardware is different from that for previous SIMD processors. Main body of the data resides on the main memory of the host computer, and only some relatively small chunks of data are exchanged between the host and GRAPE-DR. This also means quite different requirement for the software. In the case of previous SIMD machines, the entire application code must be run on the SIMD processor array. Thus, either the users or the compilers must generate the program which runs on a particular SIMD architecture.

In the case of GRAPE-DR, however, we move only the most compute-intensive part of a given application to GRAPE-DR, and all remaining part of the application code still runs on the host computer. Thus, we do not need to develop a good compiler or rewrite the entire application code by hand.

Another difference is that the architecture of GRAPE-DR is not strictly SIMD, since multiple GRAPE-DR boards connected to different host computers can run different programs. Thus, even though the chip-level architecture is SIMD, the system-level architecture is distributed-memory MIMD. Thus, we can use parallel programs developed for PC clusters, by replacing the most compute-intensive part to the calls to library routines implemented on GRAPE-DR.

The design of ClearSpeed CX600 is quite similar to GRAPE-DR. It consists of 96 PEs, each with integer ALU, FMUL, FADD, integer MAC, 5-port register file and 6KB of memory. All PEs receive the same instruction from a scalar processor integrated into the chip. Compared to GRAPE-DR, the main difference is the lack of the support for the hierarchical structure (broadcast memory and reduction network). Since the number of PEs in the CX chip is still relatively small, the reduction network might not be crucial for the application performance. The CX600 chip has the die size of 15mm by 15mm and fabricated using IBM Cu-11 130 nm process. Its peak speed for matrix multiplication is 25 Gflops. With the first implementation of the GRAPE-DR architecture, we achieved 256 Gflops double-precision speed for matrix multiplication with 512 PEs using 90nm process.

Recent GPUs with the so-called "Unified Shader" architecture, in particular nVidia GeForce 8800, can be used as GPGPU (General-Purpose GPU), in the way rather similar to GRAPE-DR processor chip. The peak performance numbers of GeForce 8800 and GRAPE-DR chip are rather similar. The former can perform 128 single-precision multiplications and 128 multiply-and-add operations also in single precision, at the clock speed of 1.35GHz. Thus, the theoretical peak performance is 518 Gflops. The peak performance of a GRAPE-DR chip is 512 Gflops. The transistor count of GeForce 8800 is 681M, while that of GRAPE-DR is 450M. Both are manufactured using TSMC 90nm process. From the viewpoint of an application developer, the largest difference between GPU and GRAPE-DR is the lack of the external memory with high bandwidth in the case of GRAPE-DR. This difference of course limit the application area to some extent, but many applications do not need such memory.

One important difference is the power consumption. GeForce 8800 can consume as much as 150W, while the maximum power consumption of a GRAPE-DR chip is 65W. This difference comes from the differences in the clock frequency and the die size. In other words, the design of GRAPE-DR is significantly more efficient than that of a GPU with unified-shader architecture. This difference in the efficiency is likely to increase in future generations, since GPUs will most likely become more flexible, in other words less efficient in the use of transistors.

7.2 On-chip communication network or off-chip memory bandwidth

One unusual design decision we made for GRAPE-DR is the design of the communication network for PEs, or lack of it. Almost every SIMD design has nearest-neighbor 2-D grid communication network, and some successful designs had network of higher dimensions (3D on APE and 14D on CM). On the other hand, the baseline GRAPE-DR architecture has only the broadcast/reduction network.

For particle-based simulations or dense-matrix operations, this lack of inter-PE communication network is no problem. Previous generations of special-purpose GRAPE systems did not have inter-PE communication network. Thus, as far as we use GRAPE-DR as the programmable replacement of old GRAPE systems, we do not need inter-PE network.

Here we discuss if addition of the inter-PE communication network would improve the performance of some applications. We consider FFT and explicit hydro code on regular grid.

The GRAPE-DR chip can perform multiple FFT operations of up to around 512 points, with the efficiency of around 10%. With some on-chip network, we may be able to use multiple PEs to perform FFT of a long vector. However, the increase in the performance is rather limited, since even if we do 1M-points FFT, the computation/communication ratio becomes only a factor two bigger.

A more straightforward way to improve the efficiency is to increase the off-chip communication bandwidth. With fast serial interfaces like XDR, it is not too expensive to connect the GRAPE-DR chip, its local memory and host processor with the link speed exceeding 10 GB/s. In this way, it is not impossible to achieve the efficiency much higher than that of the current GRAPE-DR chip.

For hydro calculations, on-chip communication network would be useful, if all variables can fit into the local memory of the chip. However, in practice that would be unlikely, and performance would be be limited by the off-chip communication bandwidth. Thus, it seems more practical to increase the off-chip communication bandwidth.

At least for these two examples, on-chip communication network does not help, and increasing the off-chip communication bandwidth is more useful.

Acknowledgments

The authors thank Toshiyuki Fukushige, Yoko Funato, Piet Hut, Toshikazu Ebisuzaki, and Makoto Taiji for discussions related to this work. The GRAPE-DR chip design was done in collaboration with IBM Japan and Alchip company. We than Ken Namura, Mitsuru Sugimoto, and many others from these two companies. The design of the control processor on the prototype board was done by Takeshi Fujino. This research is partially supported by the Special Coordination Fund for Promoting Science and Technology (GRAPE-DR project), Ministry of Education, Culture, Sports, Science and Technology, Japan.

Appendix: Example of assembly-level programming

Currently, the software system for GRAPE-DR is quite similar to the PROGRAPE/PGR system, which is designed to use FPGA as coprocessors. In both cases, the underlining idea is to use them in the way similar to GRAPE systems. In other words, they are designed to use the hardware to evaluate equation (1). The assembly language source has the following three sections

- Variable declaration section, in which we define the above x_i, x_j, f_i and other working variables.
- Initialization section, which gives the assembly language code for initialization.
- Loop section, which gives the definition of function g.

The following is the complete listing of assembly-language description for gravitational force calculation

$$\boldsymbol{a}_i = -\sum m_j \frac{\boldsymbol{r}_i - \boldsymbol{r}_j}{(|\boldsymbol{r}_i - \boldsymbol{r}_j|^2 + \epsilon_j^2)^{3/2}} \tag{2}$$

```
 1: var vector long xi   hlt  flt64to72
 2: var vector long yi   hlt  flt64to72
 3: var vector long zi   hlt  flt64to72
 4: bvar long xj         elt  flt64to72
 5: bvar long yj         elt  flt64to72
 6: bvar long zj         elt  flt64to72
 7: bvar long vxj xj
 8: bvar short mj        elt  flt64to36
 9: bvar short eps2      elt  flt64to36
10: var short  lmj
11: var short  leps2
12: var vector long accx rrn  flt72to64 fadd
13: var vector long accy rrn  flt72to64 fadd
14: var vector long accz rrn  flt72to64 fadd
15: var vector long pot  rrn  flt72to64 fadd
16: loop initialization
17: vlen 4
18: uxor $t $t $t
19: upassa $ti $ti $lr40v
20: upassa $t $t   $lr48v
21: upassa $t $t   $lr56v
22: upassa $t $t   pot
23: loop body
24: vlen 3
25: bm vxj $lr0v
26: vlen 1
27: bm mj lmj
28: bm eps2 leps2
29: vlen 4
30: nop
31: fsub $lr0 xi $r6v  $t
32: fsub $lr2 yi $r10v ;  fmul $ti $ti $t
33: fsub $lr4 zi $r14v
34: fmul $r10v $r10v $r18v ; fadd $t leps2 $t
35: fmul $r14v $r14v $r22v
36: fadd $t $r18v $t
37: fadd $ti $r22v $r18v $t
38: ulsr $ti     il"60"  $t $lr22v
39: ulsr $ti     il"1"   $t
40: uadd $lr22v     $ti  $t
41: usub hl"9fd" $ti     $t
42: ulsl $ti     il"60"  $lr30v
43: moi 1
44: uand il"1" $lr22v
45: moi 0
46: uand $r18v h"000ffffff" $t
47: uor  $ti h"3ff000000" $t
48: fmul $ti f"0.57" $t
49: fsub f"1.57" $ti $t
50: mi 1
51: fmul f"1.414" $ti $t
52: mi 0
53: nop
54: fmul $t $lr30v $t $r22v
55: fmul $r18v $r18v $r26v $t
56: fmul $r18v $ti   $r26v $t
57: fmul $ti f"0.5" $r26v
58: fmul $r22v $r22v $t
59: fmul $ti $r26v $t
60: fsub f"1.5" $ti $t
61: fmul $r22v $ti $t $r22v
62: fmul $ti $ti $t
63: fmul $ti $r26v $t
64: fsub f"1.5" $ti $t
65: fmul $r22v $ti $t $r22v
66: fmul $ti $ti $t
67: fmul $ti $r26v $t
68: fsub f"1.5" $ti $t
69: fmul $r22v $ti $t $r22v
70: fmul $ti $ti $t
71: fmul $ti $r26v $t
72: fsub f"1.5" $ti $t
73: fmul $r22v $ti $t $r22v
74: fmul $ti $ti $t
75: fmul $ti $r26v $t
76: fsub f"1.5" $ti $t
77: fmul $r22v $ti $t
78: fmul lmj $ti $t $r22v
79: fmul $r6v $ti $t ; upassa pot pot $lr0v
80: fadd $lr40v $ti $lr40v accx
81: fmul $r10v $r22v $t
82: fadd $lr48v $ti $lr48v accy
83: fmul $r14v $r22v $t
84: fadd $lr56v $ti $lr56v accz
85: fmul $r18v $r22v $t
86: fadd $lr0v $ti pot
```

Lines 1-3 defines x_i, which is, in this case the three position variables `xi`, `yi`, and `zi`. The keyword `hlt` means that it is for x_i. Keywords `elt` and `rrn` are for x_j data and f_i data. Keywords like `flt64to72` specifies the type of for-

mat conversion to be done in the interface hardware. Lines 16 to 22 are initialization. Lines 16 and 17 are assembler directives.

The `vlen` directive means the instructions will be performed for four clock cycles. As we described in section 5.1, the instructions are sent to the chip only once in every four clock cycles. The `vlen` directive specifies the actual number of vector length for the subsequent instructions.

Lines 18 to 22 specifies actual instructions. Instructions have three-address format, with `operation source1 source2 destination` syntax, except for the `bm` instruction (lines 25, 27 etc) which has `operation source destination` format. However, as in the case of line 55, we can specify multiple destinations when that is useful. Also, immediate values can be specified as is used in lines 57, 60 etc. The `bm` instruction move the data between the broadcast memory and local memory (or register) of PEs. Here, we do not give the full description of the instructions, but any operation starting with "u" is unsigned integer operation, and `fmul` and `fadd` are floating point multiplication and addition. Operands of the form `$[l]r`*nn*`[v]` such as `$lr0v` are register specification, and `$t` is the T register (see figure 5). The variables declared in the first section have static addresses in the local memory. Thus, the instruction `fsub $lr2 yi $r10v ; fmul $ti $ti $t` means "subtract the vector variable `yi` from long scalar register 2 and store the result as short vector register starting at address 10, and calculate the square the result of the previous instruction (in the T register) and store that to the T register.

This example is rather complex and long, because we calculate $x^{-3/2}$ using Newton iteration. Initial guess is made in lines 38 to 54, and Newton iteration (5 times) is done in lines 55 to 77.

From this description, the assembler generates interface functions to send x_i and x_j data and let the GRAPE-DR hardware run. The following C-code fragments is the definition of these functions and structs used by them.

```
struct SING_hlt_struct0{
  double xi;
  double yi;
  double zi;
};
struct SING_hlt_vector_struct0{
  double xi[4];
  double yi[4];
  double zi[4];
};
struct SING_elt_struct0{
  double xj;
  double yj;
  double zj;
  double mj;
  double eps2;
};

struct SING_result_struct{
  double accx;
  double accy;
  double accz;
  double pot;

};
struct SING_result_vectorstruct{
  double accx[8];
  double accy[8];
  double accz[8];
  double pot[8];

};
void SING_grape_init();
int SING_send_i_particle(struct
                         SING_hlt_struct0 *ip,
                         int n);
int SING_send_elt_data0(struct
                        SING_elt_struct0 *ip,
                        int index_in_EM);
int SING_grape_run(int n);
int SING_get_result(struct
                    SING_result_struct *rp);
```

The application program can use GRAPE-DR by calling the five functions defined above. This is rather similar to using GRAPE hardware. Communications between the host and GRAPE-DR are performed through two "send" functions and one "get" function.

At the first sight, one might think this programming model is far too specialized to the calculation of particle-particle interaction. It turned out, however, that with a few extensions this model is applicable to a wide range of applications, such as parallel integration of three-body problem, matrix multiplication, evaluation of two-electron integrals etc.

We have developed a compiler which generates the assembly code for the same gravitational force calculation from the following description.

```
/VARI  xi, yi, zi
/VARJ  xj, yj, zj, mj, e2;;
/VARF  fx, fy, fz;
dx = xi - xj;
dy = yi - yj;
dz = zi - zj;
r2 = dx*dx + dy*dy + dz*dz + e2;
r3i= powm32(r2);
ff = mj*r3i;
fx += ff*dx;
fy += ff*dy;
fz += ff*dz;
```

Currently, the code generated by this compiler is not very optimized. We are working on this issue.

8. REFERENCES

[1] J. Barnes and P. Hut. A hiearchical o(nlogn) force calculation algorithm. *Nature*, 324:446–449, 1986.

[2] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh and D.L. Slotnick, Theąąlliac IV system, Proc. IEEE, vol. 60, no. 4, April 1972, pp. 369-388

[3] T. Fukushige, J. Makino, and A. Kawai. GRAPE-6A: A Single-Card GRAPE-6 for Parallel PC-GRAPE Cluster Systems. *Publ. Astr. Soc. Japan*, 57:1009–1021, Dec. 2005.

[4] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, December 1987.

[5] T. Hamada, T. Fukuishige, and J. Makino. Pgpg: An automatic generator of pipeline design for programmable grape systems. *Publ. Astr. Soc. Japan*, 57:799–813 2005.

[6] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge, Massachusetts, 1985.

[7] T. Nishikawa, M. Inaba, and K. Hiraki. flat-c: An implementation of c language for highly-parallel computers (in japanese). In *Proceedings of SWoPP2005*. IPSJ, 2005.

[8] J. L. Potter. *The Massively Parallel Processor*. The MIT Press, Cambridge, Massachusetts, 1985.

[9] D. Sugimoto, Y. Chikada, J. Makino, T. Ito, T. Ebisuzaki, and M. Umemura. A special-purpose computer for gravitational many-body problems. *Nature*, 345:33–35, 1990.