An Application Specific Processor for Montecarlo Simulations

G. Danese, F. Leporati, M. Bera, M. Giachero, N. Nazzicari, A. Spelgatti
Dip. di Informatica e Sistemistica – University of Pavia – via Ferrata, 1
Tel. +39 0382 985350 – Fax +39 0382 985373 – e-mail: francesco.leporati@unipv.it

Abstract

In this paper we describe an Accelerator based on FPGA technology and interfaced to an external host computing system through standard bus connections; it is conceived to accelerate double precision floating point operations, present in the energy calculation of Montecarlo (MC) Metropolis particle system simulations. The Accelerator plays the role of coprocessor giving a speed-up factor equal to 4, with respect last generation PCs. The proposed solution is based on COTS components and shows good characteristics of scalability in terms of clock frequency, memory capability and number of computing units. Moreover, the Accelerator can be also conceived as a part of a computing system made up by a cluster of accelerated workstations.

1. Introduction

Since their birth, computers have been employed in simulation of physical phenomena. Basically, two kinds of simulations are often used: deterministic and statistical. The former ones try to foresee the evolution of the system by calculating some relevant physical equations. The latter ones force the evolution of the system, through a random modification of some internal parameters and by evaluating the feasibility of the change imposed through a suitable cost function which should reproduce the answer of the system (i. e. the evolution of the system energy like in a Montecarlo simulation). This approach, however, requires long computation times even though in the last years the computing power of a typical processing system is considerably increased: in fact if more and more quick computers are nowadays available, also more and more complex problems are dealt with, demanding long simulations to be thoroughly studied [1-2]. In this sense, scientists usually follow three approaches:

- using supercomputers often optimising particular operations; [3]
- running applications on clusters of workstations, so realising a global system into which the overall computing power could ideally be the sum of the single ones; [4-6]
- implementing dedicated hardware systems (accelerators) able to speed up those operations which represent the core of the calculations done. These systems are then embedded in PC or workstations which drive their activity and manage the results [7-9].

Moreover a hybrid approach consisting in clusters of accelerated workstations is becoming more and more diffused.

The project illustrated in this paper belongs to this last category, aiming at the developing a double precision floating point accelerator based on FPGAs, to be inserted inside a PC. This accelerated machine will be the first element of a cluster dedicated to particle systems simulations, consisting of commodity components (mother-boards, disks, network controllers and I/O interfaces). Each Processing Element will be equipped with an FPGA-based accelerator board charged to execute the heaviest routines of the implemented applications. This architectural choice is due to the wide use of double precision floating point operations made in physical simulations, since they require to appreciate also small variations of the physical parameters under analysis. While a general purpose computer could employ several clock cycles to execute this kind of calculations, a special purpose system can process these operations in up to one clock cycle. Moreover, a further sped-up can be achieved if a cluster of accelerated system is realised .

The Accelerator has been implemented onto a commercial FPGA board (Stratix Pro kit from Altera) interfaced with a PC through an Ethernet connection and features a speed-up equal to 4 running MC simulations on few thousands particle systems with respect to a 3 GHz P4 processor.

15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)
0-7695-2784-1/07 $20.00 © 2007 IEEE

2. Simulation of dipolar systems

We implemented a Montecarlo-Metropolis simulation [10] of a system of spins interacting with an external applied field (due to their tensorial polarizability) and among themselves (through Induced Dipole-Induced Dipole interactions), spatially located on a *cubic* lattice, free of rotating but not of translating in the space. Computationally, most of the effort is devoted to the problem of calculating the local field $\mathbf{E}_{i,loc}$ at the i-th lattice site, which is necessary to evaluate the induced dipole on each particle. Since $\mathbf{E}_{i,loc}$ is the sum of the external field $\mathbf{E}_0$ and of the dipolar fields of the neighboring particles, the calculation involves intricate implicit equations, which we computationally processed through iterative refreshing of the whole set of the electric dipoles induced on the spins. Since this is a necessary step towards calculating the total energy of the system, such dipole refreshing iterations had to be performed at every Monte Carlo move.

On the other hand, simulations take unacceptably long times even on the most recent and powerful workstations ranging from a few days up to some weeks depending on the size of the simulated system. The core of the computation is, in fact, the evaluation of the energy since, according to the implemented algorithm, equilibrium in a system with N particles is reached through a sequence of moves, carried out by randomly selecting a spin, changing its orientation through a random angular displacement and by evaluating the corresponding change in the energy. Each move can be accepted or rejected depending on the variation of the energy ΔU associated with it. We simulated lattice systems with particles ranging from a few hundreds up to 100.000 considering only first neighbor interactions, i. e. the interaction between each spin and the six closest ones in the X+, X-, Y+, Y-, Z+, Z- directions.

3. Overview of the system architecture

To carry on our project, we chose a commercial Altera board, which was conceived as a development system able to work independently of the PC. The board is equipped with a Stratix 1S40 FPGA component on which a 32 bit RISC CPU, called Nios, is implemented; this processor is programmable using C language and it is supplied with basic libraries that allow an easy handling of the following on board devices: 2 MB Ram, 8 MB Flash Memory, 16 MB Compact Flash Memory, 100 Mb/s Ethernet Interface, 2 Serial ports.

The Nios processor plays the role of computing supervisor, monitoring (external memory data storing, interfacing …) and managing possible errors during the elaboration. The Nios processor consists of a fixed point calculation unit with 128 registers which is also charged with the calculation of memory addresses. Its peculiarity is in its parametric configurability through which the user can modify data and instruction cache sizes, boot address, register number and data bus width. Moreover, the processor is able to manage external interrupt request from those peripherals which can be connected through a dedicated bus. Thanks to this feature, we are able to connect the accelerating units, memory handler and interface block to the Nios.

The size of FPGA is large enough to host two accelerating units (fig. 1), working concurrently. The board is connected to the external world through two channels: a serial port and the Ethernet. The first one works as input/output terminal for the Nios: through it we can test, download and execute the Nios code. The Ethernet

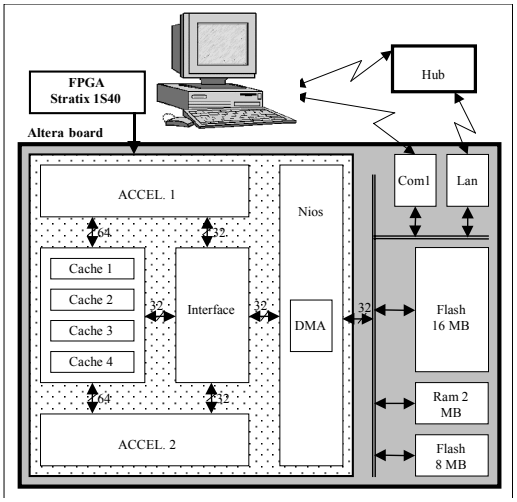

Fig. 1 – The complete elaboration system

interface is used to transfer input data and results.

Three areas were mapped onto the 8 MB Flash memory: the first contains the default configuration file for the FPGA; the second area contains a user custom configuration file and the third is available for other purposes: for instance, the Nios code may be placed here, to be executed at the boot.

The 2 MB memory is used by the Nios to store code and data of the user program and for system use; the external 16 MB flash memory must contain data for the elaboration; finally, four high speed

caches grant a constant data flow to the accelerating units. At each step of elaboration, a data sub-set is transferred in the first two caches for the elaboration; in the meanwhile, the other caches are filled with new data coming from the 16 MB memory. At the end of the step, caches are swapped and elaboration can continue; at the same time data on the unused cache are stored in the 16 MB Memory. The transfers between 16 MB memory and caches are carried out through DMA managed by the Nios processor. At the end of the elaboration, all needed data from a 16 MB memory are retrieved and results sent to the PC.

All the devices present on the board work at a 100 MHz clock frequency which is not the maximum one but an acceptable trade off between the speed of the elaboration and the correct timing of the devices.

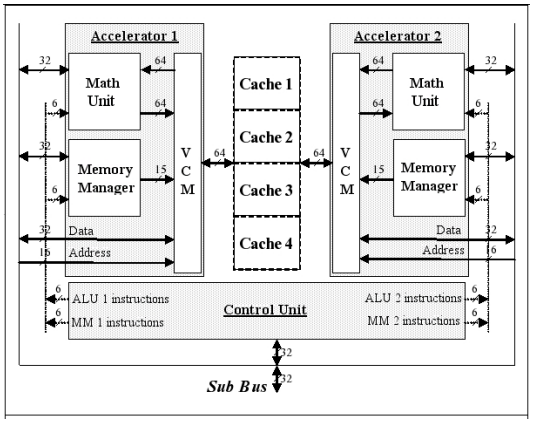

Fig. 2 – Architecture of the computational

The high-level architecture of the computational unit is shown in fig. 2. The main functional elements are:

- 2 accelerating units, working independently;
- a Cache Memory (4 banks), which can store input data and results for the two accelerating units;
- a Control Unit.

On the bottom part of the figure a bus devoted to communication between Accelerator and Nios processor ("sub bus") is represented.

3.1 Math Unit architecture

The Math Unit functional core is a double-precision floating-point ALU, which integrates both an adder and a multiplier operating in a parallel fashion. Both devices are pipelined (9 stages for the adder and 15 for the multiplier) so that high clock rates are achievable. Note that, in the expected applications, accurate coding can minimize the negative effects of such latency.

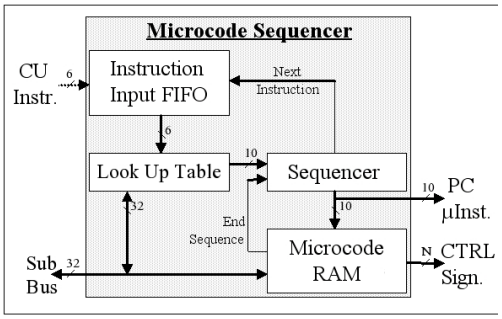

Fig. 3 – The architecture of the microcode sequencer

Together with the adder and the multiplier, the ALU also contains 3 register banks, each able to store 4 double-precision floating point numbers. The banks are each tied to a particular purpose (one is for input data, one for adder results and one for multiplier results).

Like in many similar applications, to make computing elements and storage space independent, a FIFO memory for both inputs and outputs is implemented (there are two FIFO queues on the output since arithmetic results are separated from logical ones).

The ALU operations are encoded in 37-bit words, able to simultaneously trigger:

- either a sum or a comparison;
- a multiplication;
- a data fetch;
- 3 write operations to the internal register banks;
- the output of a result.

To achieve better performance with our specific task, the operands of the adder can optionally be multiplied by [-2, -1, 2] for the first operand, and [-1, -0.5, 0.5] for the second one. In a similar way, the multiplier result can be doubled, halved or negated without extra clock cycles.

Since feeding the op-codes would require a large and mostly wasted bandwidth (the code is essentially cyclic, so that the same op-codes are executed over and over again) the code sequences are stored in a Microcode Sequencer. This device stores the program sequences in an internal RAM and associates to them a 6-bit op-code (this is much like having a CPU with a micro-programmed control unit whose code can be changed by the application to define a custom instruction set).

3.2 The Memory Manager

The Math Unit itself has no addressing capabilities in either input or output channels, so every memory I/O operation must be managed by an external device. The Memory Manager is deemed to that task. The Memory Manager is a device, conceived for a specific application class: those where most computations are performed on data logically organized in three-dimensional matrixes. Decoupling the allocation issues from the computing algorithm, the Memory Manager computes the memory addresses from semantic-level inputs, such as addresses in the matrix domain (X-Y-Z coordinates) or offsets between elements (the matrix is supposed to be cyclic, so that e.g. the leftmost element in a row is adjacent to the rightmost element in the same row). This is of extreme importance, since otherwise the same code would require at least a recompilation to be executed on matrixes with different sizes.

3.3 Sequencers

Inside the Accelerator, there are different sequencers. Since these are rather similar to each other, we'll describe one of them, the Microcode Sequencer, shown in fig. 3.

As already stated, the Microcode Sequencer converts a single 6-bit opcode into a sequence of custom microinstructions. The logic required to achieve this goal is somewhat simple.

All microcode sequences are stored in a RAM area, shown in the lower right corner of the figure. The last instruction of a sequence is marked with an "end sequence" flag, while the first one is referenced through a look-up table addressed by the current op-code. A FIFO queue stores the program to be executed (which is a sequence of 6-bit op-codes). The "Sequencer Control" block acts as a control unit and generates the address for the Microcode RAM.

When a new instruction is to be executed, the "Sequencer Control" gets the first microcode address from the LUT. Then, until the last micro-instruction is reached, it increments the Microcode RAM address so that the micro-instructions are properly generated.

4. Programming the Accelerator

As previously mentioned, the instruction sets for the ALU and for the Memory Manager are fully programmable to target the particular application for which the Accelerator was used. Each instruction consists on a microinstruction sequence which specifies what operation should be performed within a clock cycle. Since microinstructions are binary words, programming the Accelerator is not an easy task, therefore a set of tools was conceived and realized, to simplify sequence writing and testing, introducing some automatic mechanisms. In the following these tools are briefly described.

4.1 The sequence generation tool for the Accelerator

With "Accelerator sequence" we mean a group of 38 bit microinstructions that defines an instruction for the Accelerator. Instead of writing the binary profile of each microinstruction, we defined a pseudo-assembly language that describes the sequences and a tool that translates them into code. As seen before, the Accelerator can perform, in each clock cycle, various operations (sum/comparison between floating point values, a product between floating point values, 3 write operations, data read operation, result write operation).

The syntax we defined follows these rules:

- every microinstruction ends with a semicolon
- an asterisk indicates the end of a microinstruction and the beginning of another
- a sequence of microinstructions must be comprised within brackets
- a sequence of microinstructions must have a name in the first line after the opening bracket, and within double quotes
- proper keywords were defined to declare the begin and the end of a program.

Five types of operations were conceived. In particular, sum and multiplication support two important features:

- if it is specified only one operand, the other one is automatically set to the proper constant value (0 or 1);
- on each operand a multiplicative constant (clearly related to Montecarlo calculations), selected between five, can be applied.

4.2 The software development environment

The development environment we realized to program the Accelerator consists of three main parts: a translator, a post-processor and a simulator. The translator and the post processor are Perl

scripts, while the simulator is a Visual Basic program.

The main function of the translator is to convert a hand written program into 38-bit words binary code, which can be processed by the Accelerator. The name "translator" was chosen since the program is not an assembler (there is not a one to one correspondence between source code instruction and the executed instruction) nor a compiler (it manages a very low level written source code). A post-processor was realized to perform "general" tasks, like setting the *end-of-sequence* bit in the last instruction of each sequence, determining the number of sequences and the operation count per each sequence, reordering information to create a file containing both binary and source code for the simulation.

Finally, a simulator was developed to check the semantic correctness of the algorithm implementation, without merging it into the software structure managing the entire system (Memory Manager, Virtual Cache Manager, and so on). Three principal settings influence the way it works:

- *symbolic vs. numeric*: when symbolic mode is active, input data are treated as symbols, and the produced output is a string that contains a literal expression formally calculated. Otherwise, when numeric mode is active, inputs are used as numeric data and also outputs are numeric.
- *automatic vs. manual*: in automatic mode, simulator gets all inputs from a specified file, which also contains the list of sequences to be executed. In manual mode, there are two different cases: in symbolic mode, simulator generates a sequence of symbols (e.g. n0, n1, n2, ...); in numeric mode a pop-up window asks the user for every single data.
- *continue vs. stop:* at end of sequence: in the former mode execution continues until every input datum is used, in the latter mode execution stops at every end of sequence.

Combining these options two principal execution modes (*step by step* and *run*) can be settled.

The simulator displays some information to the user:

- the next microinstruction to be executed, both in binary format and in source code;
- the sequence number and the microinstruction number;
- data contained in register banks;
- data contained in multiplier and adder pipelines;
- output data;
- the number of clock cycles since the beginning of execution.

Finally, all the described tools were "merged" into an Integrated Development Environment which allows to use them in an automatic way and in particular it:

- provides a user-friendly graphical interface to simplify code writing, translation and simulation;
- provides an easy way to call functions and menus, preventing the user from dealing with complex syntactical details;
- allows source code editing;
- shows runtime errors and warnings;
- simplifies the creation of new projects, managing the generation of all necessary files.

4.3 The sequence generation tool for the Memory Manager

The Accelerator operates on data streams, so it was necessary to develop a Memory Manager to fetch data from memory and store results back. Again, for sake of generality and reusability, we conceived a pseudo-assembly language, to easily write sequences for Memory Manager.

We provided the Memory Manager with less support than the Accelerator (e.g.: no graphical interface, no simulator), but we implemented a tool able to automatically generate Memory Manager sequences starting from the Accelerator ones.

The developed Memory Manager language consists on five different microinstructions:

- INIT [X,Y,Z]: it defines the dimensions of the matrices containing dipole coordinates and moments (it is clearly related to Montecarlo-Metropolis algorithm implementation);
- GVC [16 bits word]: it defines the cache configuration and the synchronization between the Accelerator and the host processor;
- R [address]: it reads data from memory and sends them to the Accelerator;
- W [address]: it reads data from the Accelerator and writes it into memory;
- Pointer Modification [pointer, new value]: it sets a new value for the specified pointer.

At each clock cycle the Memory Manager can modify a pointer and also performs a Read/Write operation. To indicate the two operations which are performed in parallel, the syntax uses the symbol "--", as shown below:

data read/write -- modify pointer

Physics simulations are often data-intensive and data are used in a very regular way. In particular, in our project, the dipole scanning patterns bring to a fully predictable use of memory. This allows us to implement an automatic sequence generation for the Memory manager. The following generation chain was thus implemented:

1) a C program generates diagonal scanning patterns for a generic lattice. Patterns are expressed through indexes that indicate dipole positions in the simulation box with periodic boundary conditions. The program generates three different files for input data, output data and pointer management;

2) a Perl script elaborates the Accelerator sequences by extracting I/O operations and generating a file that contains filtered instructions and some additional information:

3) another Perl script generates the Memory Manager sequences joining information on input data, output data and Accelerator operations. The script checks the Accelerator instructions on a one by one basis and identifies input/output ones; then, it reads proper information from the correct file and creates a Memory Manager instruction (a R/W one), that will be put in a resulting file:

4) a third Perl script adds to this file, the pointer modification instructions:

5) in the last step the translator generates the binary profile of the instructions to be executed.

4.4 Control Unit language

No flow control tools are provided in the Accelerator sequence set: this task is given to Control Unit. We developed a simple language that allows to implement loops, conditional or unconditioned jumps, comparisons and counters, and a translator to get binary code from source code. A Control Unit program is usually a very short code quite rarely modified.

4.5 Other instructions

In addition to previously mentioned instructions, an Execute instruction was implemented to allow the Accelerator to perform one or more sequences. There are four independent units on board that can execute a sequence: two ALUs and two Memory Managers. Each of these units has a LUT addressed with a six bit word that contains all the addresses of the stored sequences. The Execute instruction features the following syntax:

IST [MM2][ALU2][MM1][ALU1]

Each parameter is a 7 bit word: one bit indicates if that unit is activated, the other six bits are the address of the sequence that will be executed.

Finally, instructions were developed to manage counters (initialization and working) which are used to evaluate the performance in terms of elaboration times, for synchronization and timing purposes.

5. Energy evaluation implementation

In the previous section we described the architecture of the Accelerator trying to highlight how the design aims to make units working in parallel. In this section we illustrate how the physical application is implemented to introduce a further level of parallelism in the elaboration, while reducing the need for data communication between the two accelerating units and the main processor.

Since the 85% of the calculations done in a Montecarlo-Metropolis simulation concerns the evaluation of the energy due to particle interactions, the architecture implemented onto the FPGA is mainly dedicated to that portion of the program.

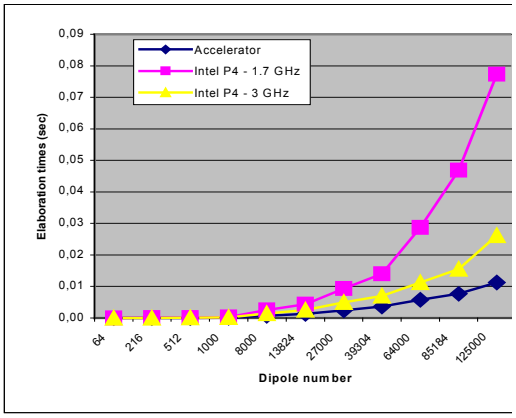

Fig. 4 – Computing times for the energy calculations performed on the FPGA based accelerator and on 2 Intel P4 processors

The global energy in the system is given by the sum of single dipole energies which are, in turn, due to the proper energy of the dipole plus that coming from the interactions with first neighbor dipoles (determining Induced Dipole – Induced Dipole interactions). Each dipole is represented by the three components of its moment and its energy is due to a linear combination of these terms with those of the other neighboring dipoles.

6. Results

The entire system has been tested by executing Montecarlo simulations of different size lattices (4<ND<50, where ND is the number of dipoles on each side of the cubic lattice).

Performance has been evaluated in terms of speed-up with respect the execution of the same simulation on Intel P4 processors with 1GB Ram memory and in terms of FPGA occupation. Simulation code was written in C language and optimized using the Microsoft Visual C++ environment. The Accelerator elaboration times were measured by means of the clock counters implemented in the interface between Nios and the coprocessor previously described.

In fig. 4 we show the execution times of the energy calculation which is repeatedly executed 8*2*N*10000 times where 8 is the coefficient responsible for the interaction settlement (equilibration) and N is the dipole number: this gives reason of the high computational load which can lead (for big particle systems, e. g. 100000 dipoles) to wait months for results, if the simulation should be performed on a PC. In the figure, the computing times obtained for the same calculations executed on Intel P4 processors with 3 GHz and 1.7 GHz frequency respectively are shown together with those obtained by using the Accelerator. The shape of the plots clearly depicts the quadratic dependence on the particle number, since the energy is due to the interactions among all the particles. The speed-up factor (defined as the ratio between the computing times of the 2 Intel processors and that of the Accelerator) is presented in fig. 5, and is increasing for the 1.7 GHz processor due to cache effect, while for the most performing Intel processor (3 GHz) sets around 2. Considering that in the FPGA we used, other 2 accelerating units could be implemented, we can reasonably state that a speed-up factor equal to 4 can be achieved in case of a "full" implementation on the FPGA component we chose (Stratix EP1S40). Further speed-up could be obtained if other components of the Altera's family (EP1S120 or Stratix2) should be employed.

The cost of each board we bought was nearly $ 1200 (mid 2004): this represents an important indication when predicting trade-off between a cluster of workstations with respect to a cluster of FPGA based accelerators. In practice, our work indicates that each FPGA unit gives a computational power 4 times greater, only doubling costs with respect to a computational unit in a PC cluster, providing the scientist with a COTS desktop computing system on which he/she can run simulations. Finally, these estimations could be further enforced if an analogous work would be performed on Stratix2 evaluation kits now available from Altera.

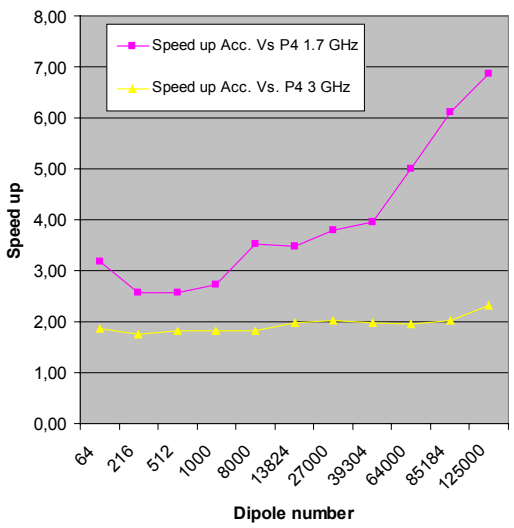

Fig. 5 – Speed-up of the FPGA based accelerator with respect the P4 Intel processors

7. Conclusions

The implemented prototype of the accelerated computer dedicated to the simulation of interacting particle systems satisfies the conditions discussed in the introduction.

The cost of the prototype amounts to about $1,200 per node, low enough to justify personal interactive use. At present, the computing system is not commercially available, but the assembling phase does not require particular technologies, so other prototypes can be easily built.

The interface services are satisfactory and can be easily integrated with other functions specific to any particular application.

The computing system is interesting both because of its performance, which is comparable to that offered by supercomputing services if more accelerators should be assembled together, and because of its ease of use.

Our work does not want to enter in competition with other solutions but it clearly indicates that the realization of cluster of accelerators is an interesting way to achieve computing performance with low costs like shown by [11] and [12].

A further improvement could be achieved by a full custom ASIC implementation of the Accelerator which is not justified at a prototyping level while it allows a large scale manufacturing with reduced costs. This would make available several computing units connected in cluster fashion by means of a point to point network, providing the user with a great computing power like in the case of the Grape project.

References

1. J. DONGARRA et al.: "*High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions*", IEEE Comp. in Science & Engin., vol. 7(2), Mar-Apr 05, pp. 51-59.
2. P. MARSH: "*High performance horizons*"; Computing & Control Engineering Journal, vol. 15(6), December-January 2004/2005, pp. 42-48.
3. D. G. FEITELSON: "*The supercomputer industry in light of the Top500 data*"; IEEE Comp. in Science & Engin., vol. 7(1), Jan-Feb 05, pp. 42-47.
4. M. K. GOBBERT: "*Configuration and Performance of a Beowulf Cluster for Large-Scale Scientific Simulations*"; IEEE Comp. in Science & Engin., vol. 7(2), March-April 2005, pp. 14-26.
5. "*Special Issue on GRID Computing*"; Proceedings of the IEEE, vol. 93(3), March 2005, pp. 692-697.
6. B. BOGHOSIAN et al., "*Scientific applications of grid computing*", IEEE Comp. in Science & Engin., vol. 7(5), Sept.-Oct. 2005, pp. 10-13.
7. http://www-zeuthen.desy.de/ape/html/.
8. T. FUKUSHIGE, P. HUT, J. MAKINO: "*High-Performance Special Purpose Computers in Science*"; IEEE Comp. Science and Engin. vol. 2, 1999, pp. 12-13.
9. JERRAYA, W. WOLF: "*Multiprocessor systems on chip*"; Morgan Kaufmann, 2005.
10. N. METROPOLIS et al, "*Equation of state calculations by fast computing methods*"; Journal of Chem. Physics, vol. 21, June 1953, pp. 11-16.
11. A. CRUZ et al., "*A Special Purpose Computer for spin glass models*", Computer Physics Comm., vol. 133, n° 2-3, 2001, pp. 165-176
12. BELLETTI F. et al., "*An adaptive FPGA computer",* IEEE Comp. in Science & Engin., vol. 8(1), January-February 2006, pp. 41-49.

0-7695-2784-1/07 $20.00 © 2007 IEEE