S-B1-5

Magnetohydrodynamics Simulations on GPU Clusters with Automatic Programming

GPUクラスタとプログラム自動生成による磁気流体シミュレーション

Takayuki Muranushi
村主　崇行

Yukawa Institute for Theoretical Physics, Kyoto University
Kitashirakawa Oiwakecho, Sakyo-ku, Kyoto 606-8502 Japan,
京都大学基礎物理学研究所／白眉センター　〒606-8502　京都市左京区北白川追分町

We are developing a new programming language named Paraiso. Its goal is to generate, from mathematical notations of the algorithms, explicit solvers of partial differential equations on massively parallel and heterogeneous computers. Current version of Paraiso can generate multicore fluid simulators that are as fast as hand-written ones, and can translate such codes to GPU programs without any change. It can also optimize such codes based on benchmarking and simulated evolution. Please have a look at `http://paraiso-lang.org/wiki/` for more detail.

1. Background

The good news is that our computers are getting even faster and faster. The bad news is that they're getting harder and harder to program. To change a single-CPU program for example to a program that uses hundreds of GPUs, you need to add a lot of lines that handle parallelism and communications. Since such codes are results of logical necessity that the simulated algorithms are the same, can we automate the process?

Parallel programming languages are studied for a long time, but we not yet see a decisive success. Just one example was a High Performance Fortran, a very promising approach to introduce a high-level parallelism in Fortran but, as is summarized by the project leader [1], it failed. Perhaps seeking for a single general-purpose parallel programming language is not a good idea, because parallel programs need to exploit the parallelism inherent in each problem domain. We may need a whole range of different approaches instead [2].

FFTW [3], ATLAS [4], and SPIRAL [5] are examples of successful domain-specific code-generation and auto-tuning framework, specialized for fast Fourier transformation, linear algebra, and signal processing, respectively. Paraiso tries to achieve the similar role for the domain of partial differential equations solvers. Similar projects are Liszt [6] and Physis [7].

2. Orthotope Machine and its Instruction Set

The Orthotope Machine (c.f. Fig. 1), the central concept of Paraiso, is a virtual machine much like vector computers. Orthotope Machine consists of

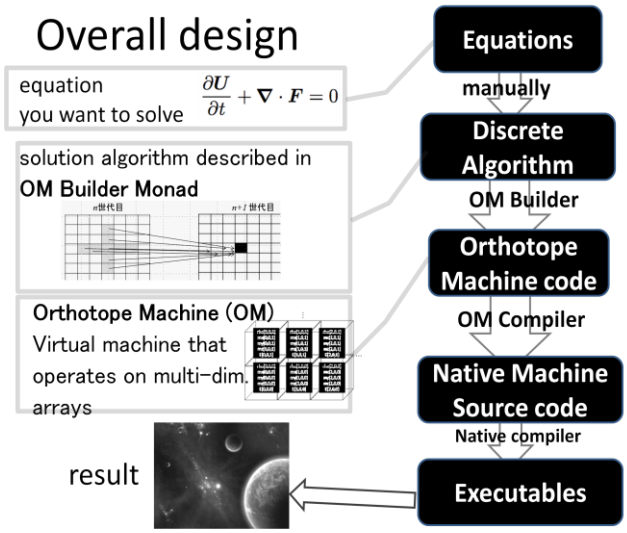

Fig.1. The overall design of Paraiso.

registers; each register is a multidimensional array of infinite size. Arithmetic instructions work parallelly on these array registers, like adding or multiplying every pair of element at the same array index. Other instruction loads from neighbour cells, thus shifts the entire array by some constant vector. There is no intention of buiding a real hardware: Orthotope Machine is a thought object to construct dataflow graphs representing parallel computation.

Table 1 is the list of all instruction supported by the Orthotope Machine. The instruction set is sufficient to implement major portion of uniform-mesh explicit partial differential equations solvers. Real programs encode far more diverse concepts (e.g. various boundary conditions) than the instruction set encompass. How far we can treat them formally is future challenge.

Table I. The instruction set of the Orthotope Machine

	Intensity (arb. units)	T_e (eV)
`Imm`	load constant value	
`Load`	read from named array	
`Store`	write to named array	
`Reduce`	array to scalar value	
`Broadcast`	scalar to array	
`Shift`	copy each cell to neighbourhood	
`LoadIndex`	get coordinate of each cell	
`LoadSize`	get array size	
`Arith`	various mathematical operations	

3. Builder Monad

Paraiso programs are written in terms of Builder Monads. Builder monads are elementary code generators, and by combining them we get larger code generators.

Math operation such as addition, multiplication, interpolation and time/space derivatives, can be defined between builder monads. Operations are defined in a commutative way; the addition of two code generators is a code generator that generates a program that calculates the sum of the two results of the two programs that would have been generated by former two code generators. Builder monads can also become elements of mathematical structures such as complex numbers and tensors.

All these combined you can program Orthotope Machine so easily. You describe the algorithm you want to generate using tensor equations. Then if you re-interpret the equations as equations of builder monads, the system of equations automatically becomes a code generator that generates the simulator for the system.

4. Code Generation

The Orthotope Machine program is then translated to native codes such as C++ and CUDA (Fig. 2). Starting from a parallel dataflow graph that encompasses the entire program, Paraiso holds great freedom of program transformation. The built-in analyzer as well as user can add annotations to the dataflow graph, changing which data to keep on memory, which data to re-compute, how the algorithm is separated into subroutines. Just for example, a hydrodynamics solver written in Paraiso has 2^{2000} possible implementations.

At the moment, Paraiso can generate OpenMP and CUDA program for multicore CPUs as well as GPUs On 8-core CPU. The speed of OpenMP version almost matches that of hand-written simulators widely used. CUDA version is 10 times faster than them and generated without changing a line. By adding just 1 or 2 lines of Annotation by hand, we can make radical changes the code, resulting in 10 times improvement in the performance. Paraiso is able to search for optimal annotation via method based on Exchange Monte Carlo [8] and genetic algorithms. The experiment is ongoing as I'm writing this, and I observe additional 30% optimization. Since this is a PLASMA conference, I hope I have time to try writing MHD codes by the conference date.

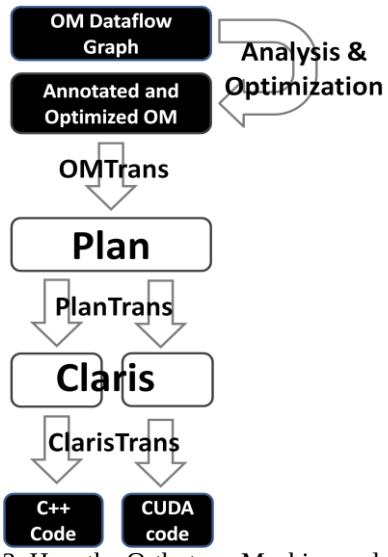

Fig.2. How the Orthotope Machine codes are translated into native codes.

Acknowledgments

This project was partially supported by JST, CREST through its research program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing."

References

[1] K.Kennedy, C.Koelbel, and H.Zima: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22

[2] S.Jones: Functional Programming eXchange 2011, *"Managing parallelism: embrace diversity, but control side effects"*

[3] M.Frigo and S.Johnson: Proc. IEEE **93** (2), 216–231 (2005)

[4] R.Whaley, A.Petitet and J.Dongarra: Parallel Computing, **27**(1-2) pp.3--35, 2001.

[5] M.Püschel et al: Proceedings of the IEEE special issue on *"Program Generation, Optimization, and Adaptation,"* Vol. 93, No. 2, 2005, pp. 232-275

[6] Z.DeVito et al: ACM/IEEE Supercomputing (2011)

[7] N.Maruyama, T.Nomura, K.Sato and S.Matsuoka: Supercomputing (2011) pp. 1--12

[8] Hukushima, Nemoto; J. Phys. Soc. Jpn. **65** (1996) No.6,pp1604-1608