Lattice QCD on Graphics Processing Units?

Zhaofeng Liu

LPT, INRIA-Futurs(Orsay), PRISM(Versailles), IRISA/INRIA(Rennes), CAPS-Entreprise

June 14, 2007

Outline

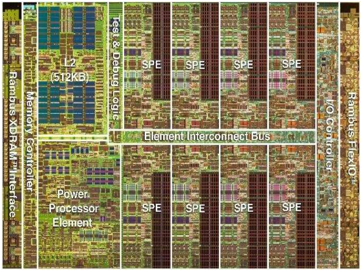
- Background
  - Graphics Processing Units(GPU)
  - HMC c-code of ETMC
- Some first attempts
  - profiling
  - distribution of Hopping_Matrix()
  - puting on GPUs

Full Lattice QCD is expensive

- Fermions are expensive. Need to solve linear equation $D_{lattice}\eta = \phi$. For a $16^3 \times 32$ lattice, $D_{lattice}$: $\sim$ 1 million $\times$ 1 million matrix.
- apeNEXT, QCDOC(predecessor of BlueGene/L): ($\sim$ 10 Teraflops) supercomputers optimized for LQCD calculations.

To "solve QCD" numerically needs "cheap" multi Petaflops machines.

- Multicore chips.
- CPU+coprocessors (e.g., IBM Cell: PPE+SPE)
  CPU+Graphics Processing Units(GPU)

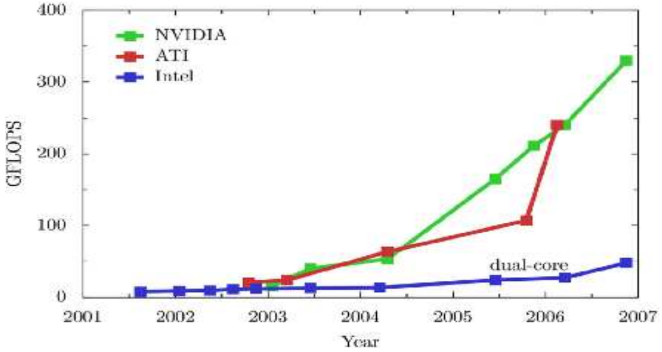


Graphics Processing Units(GPU)

http://developer.download.nvidia.com/compute/cuda/0_81/NVIDIA_CUDA_Programming_Guide_0.8.2.pdf

- Powerful and inexpensive. For example,
  NVIDIA GeForce 7900 GTX($378 as of Oct.2006)
  51.2 GB/sec memory bandwidth.
  NVIDIA 8800 GTX($599) **330 GFLOPS** measured.
  ATI Radeon X1900 XTX(similar price)
  can sustain a measured **240 GFLOPS**.
  Compare to the SSE units of a dual-core 3.7 GHz Intel Pentium Extreme Edition 965
  8.5 GB/sec and **25.6 GFLOPS** theoretical peak.
- Programmable. High level languages have emerged.
- The computational capabilities of GPUs grow fast.

GPU



- For the moment only single precision.
- Effective General-Purpose GPU programming is not simply a matter of learning a new language.
- Few if any full-featured debuggers and profilers.
- Poor bandwidth between CPU and GPU.

GPGPU and LQCD on GPUs

- General-Purpose computing on the GPU (GPGPU) has become interesting.
  - http://GPGPU.org : general discussion, programming help, etc.
  - fluid dynamics simulations, visual simulation of cloud dynamics, etc.
- G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi and K. K. Szabo, "Lattice QCD as a video game", arXiv:hep-lat/0611022.
  - $N_x \times N_y \times N_z \times N_t$ lattice $\rightarrow N_x N_y \times N_z N_t$ texture.
  - a lattice site $\rightarrow$ a pixel.
- Use GPUs as accelerators. Move time consuming parts of LQCD simulation codes to GPUs.

HMC c-code of ETMC

- HMC — Hybrid Monte Carlo Algorithm.
- ETMC — European Twisted Mass Collaboration.
- Dynamical simulation with two quark flavors. Include fermion determinant by pseudo-fermions

$$\mathit{detQ} \propto \int \mathrm{d}\phi \mathrm{d}\phi^\dagger \exp\big(-\phi^\dagger \frac{1}{Q}\phi\big)$$

- The gauge action includes a plaquette term (1×1 Wilson loops) and a planar rectangular term (1×2 Wilson loops).
- The code can also compute quark propagators (invert).

Profiling of the HMC code

- Intel Itanium 2 processor, no SSE (Streaming SIMD Extensions).
- compile with icc.
- gprof hmc_tm(invert) gmon.out

lattice	Hopping_Matrix()	get_rectangle_staples()	total
hmc_tm			
4^4	∼31%	∼52%	∼83%
$6^3 \times 4$	∼40%	∼45%	∼85%
$8^3 \times 16$	∼58%	∼26%	∼84%
$16^3 \times 32$	∼74%	∼8%	∼82%
invert			
$16^3 \times 32$	∼81%	—	—

Hopping_Matrix.c

- The HMC algorithm requires computations like $\eta = Q^{-1}\phi$.
- It's done by using, say, the Conjugate Gradient method, which requires multiplications of Q with vectors.
- Hopping_Matrix() finishes the "hopping" part in the multiplications:

$$l(x) = \kappa \sum_{y} \sum_{\mu} \delta_{x,y-\mu}(1+\gamma_\mu)U_\mu(x)\ k(y).$$

  Here $U_\mu(x)$ is the gauge field. l and k are spinors. When y is odd/even, x is even/odd on the lattice.
- There are two outer loops in Hopping_Matrix(). One goes over y, the other goes over x.

Hopping_Matrix.c (cont'd)

$$l(x) = \kappa \sum_{y} \sum_{\mu} \delta_{x,y-\mu}(1+\gamma_\mu)U_\mu(x)\ k(y).$$

- In each loop, there is an inner loop for the four directions: $\mu = \hat{t}, \hat{x}, \hat{y}, \hat{z}$.
- The gauge fields $U_\mu(x)$ are in the order they are accessed to allow for continuous memory access.
- As a first step, we are trying to put Hopping_Matrix() on GPUs since it is the most time consuming part.

Distribution of Hopping_Matrix.c

$$l(x) = \kappa \sum_{y} \sum_{\mu} \delta_{x,y-\mu}(1+\gamma_{\mu})U_{\mu}(x)\ k(y).$$

- Distributing the loops in Hopping_Matrix() makes it possible to put Hopping_Matrix() on several GPUs and makes it simpler to optimize the code for GPUs.
- At the same time, by re-organizing the distributed loops in different ways, we may find an optimized way which fastens the code.
- When re-organizing the distributed loops, the gauge fields $U_{\mu}(x)$ should be re-ordered to allow for continuous memory access.
- Spinor fields used for storing intermediate results should also be re-ordered for optimization in memory access.

Distribution of Hopping_Matrix.c (cont'd)

$$l(x) = \kappa \sum_{\mu} \sum_{y} \delta_{x,y-\mu}(1+\gamma_{\mu}) U_{\mu}(x)\ k(y).$$

- The loop over y or x is distributed along four directions $\mu = \hat{t}$, $\hat{x}$, $\hat{y}$, $\hat{z}$. i.e., for each direction, there is a loop over y or x.
- After re-ordering the gauge fields and spinor fields correspondingly, we find some improvement on the Intel Itanium 2 processor (no SSE, compile with icc).

$16^3 \times 32$	before(seconds)	after(seconds)	improvement
hmc_tm	$\sim$ 20300	$\sim$ 16100	~20%
invert	$\sim$ 13600	$\sim$ 10600	~20%

hmc_tm: one trajectory.
invert: one quark propagator with source at 0.

Put Hopping_Matrix() on a GPU

$$l(x) = \kappa \sum_{\mu} \sum_{y} \delta_{x,y-\mu}(1+\gamma_\mu)U_\mu(x)\ k(y).$$

The basic scheme is:

1. Beginning of the program: initialize GPU environnement and put data from gauge fields $U_\mu(x)$ on the GPU.
2. In Hopping_Matrix():
    - put $k(y)$ on GPU
    - compute loop of y
    - permute intermediate spin fields
    - compute loop of x
    - get back $l(x)$

    GPUs excel at processing data in two-dimensional arrays, but are limited when processing one-dimensional arrays.
3. End of program: cleanup GPU stuff.

Things going on and to do

- One version is finished, which uses Cg, a traditionnal shading language created by Nvidia. This is on GeForce7 series. $\sim 8\times$ faster for executing Hopping_Matrix() once comparing with a pentium machine. Quite encouraging.
- Another version is under developing, which uses Cuda to use the new possibilities of the GeForce8 series.
- Performance analysis for the above GPU versions.
- Performance analysis for the codes using the distributed Hopping_Matrix() (with or without SSE). (More precise analysis than using 'gprof'.)
- Is there any improvement on a Pentium4 machine with SSE2 using a distributed Hopping_Matrix()?