PROCEEDINGS OF SCIENCE

Lattice SU(2) on GPU's

Nuno Cardoso
CFTP, Instituto Superior Técnico
E-mail: nunocardoso@cftp.ist.utl.pt

Pedro Bicudo
CFTP, Instituto Superior Técnico
E-mail: bicudo@ist.utl.pt

We discuss the CUDA approach to the simulation of pure gauge Lattice SU(2).
CUDA is a hardware and software architecture developed by NVIDIA for computing on the GPU. We present an analysis and performance comparison between the GPU and CPU with single precision. Analysis with single and multiple GPU's, using CUDA and OPENMP, are also presented. In order to obtain a high performance, the code must be optimized for the GPU architecture, i.e., an implementation that exploits the memory hierarchy of the CUDA programming model.
Using GPU texture memory and minimizing the data transfers between CPU and GPU, we achieve a speedup of 200× using 2 NVIDIA 295 GTX cards relative to a serial CPU, which demonstrates that GPU's can serve as an efficient platform for scientific computing. With multi-GPU's we are able, in one day computation, to generate 1 000 000 gauge configurations in a 48^4 lattice with $\beta = 6.0$ and calculate the mean average plaquette.
We present results for the mean average plaquette in several lattice sizes for different β. Finally we present results for the mean average Polyakov loop at finite temperature.

The XXVIII International Symposium on Lattice Field Theory - LAT2010
July 14-19 2010
Villasimius, Sardinia, Italy

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence. http://pos.sissa.it/

1. Introduction

In the last years, the developments in computer technology made graphics processing units (GPU's) available to everyone as a powerful coprocessor in numerical computations. Currently many-core GPU's such as NVIDIA GTX and Tesla GPU's can contain more than 500 processor cores on one chip in the recent architecture, Fermi architecture. However, GPU's are known to be hard to program, since coalescing of parallel memory is a critical requirement for achieve the maximum performance.

In this paper, we explore the limits of GPU computing in generating SU(2) pure lattice configurations using the heat bath method and performing some measurements in this configurations like the mean average plaquette and the Polyakov loop in single precision. We test the performance using 1 GPU, 2 GPU's and 4 GPU's. We use two NVIDIA 295 GTX cards, each card has two GPU's inside.

This paper is divided in 5 sections. In section 2, we present a little description in how to generate lattice SU(2) configurations, in section 3 we present the Cuda programming model, in section 4 we present the GPU performance over one CPU core as well as results for the mean average plaquette and Polyakov loop for different β and lattice sizes. Finally, in section 5 we present the conclusions.

2. Lattice SU(2)

A matrix U $\in$ SU(2) can be written as:

$$\mathrm{U} = a_0 \mathbb{1} + i\mathrm{a}\cdot\boldsymbol{\sigma}$$

with

$$a^2 = a_0^2 + \mathrm{a}^2 = 1$$

This condition defines the unitary hyper-sphere surface S^3,

$$\mathrm{dU} = \frac{1}{2\pi^2}\delta\left(a^2-1\right)\mathrm{d}^4a$$

and

$$\mathrm{Tr}\,\mathrm{U} = 2a_0, \qquad \mathrm{UU}^\dagger = \mathbb{1}, \qquad \det\mathrm{U} = 1$$

To generate SU(2) pure gauge configurations, [1, 2], we need to calculate the staple, V the sum of the six products of neighboring links that interact with a particular link, in the heat bath method. The distribution to be generated for every single link is,

$$\mathrm{d}P(\mathrm{U}) \propto \exp\left[\frac{1}{2}\beta\,\mathrm{Tr}\left(UV\right)\right]$$

Using a useful property of SU(2) elements, saying that any sum of them is proportional to another SU(2) element

$$\tilde{\mathrm{U}} = \frac{V}{\sqrt{\det V}} = \frac{V}{k}$$

PoS(Lattice 2010)024

where $\tilde{\mathrm{U}}$ is another SU(2) element and the invariance of the group measure

$$\mathrm{d}P\left(\mathrm{U}\tilde{\mathrm{U}}^{-1}\right) \propto \exp\left[\frac{1}{2}\beta k \mathrm{Tr}\,\mathrm{U}\right]\mathrm{dU} = \exp\left[\beta k a_0\right]\frac{1}{2\pi^2}\delta\left(a^2-1\right)\mathrm{d}^4 a$$

We generate a random number for $a_0 \in [-1,1]$ with distribution

$$P\left(a_0\right) \sim \sqrt{1-a_0^2}\exp\left(\beta k a_0\right)$$

once the a's are obtained in this way, the new link is updated,

$$\mathrm{U}' = \mathrm{U}\tilde{\mathrm{U}}^{-1}$$

3. Cuda Programming Model

NVIDIA Compute Unified Device Architecture (CUDA), [3, 4] is a general purpose parallel computing architecture with a novel parallel programming model and instruction set architecture. C for CUDA exposes the CUDA programming model as an abstraction of GPU parallel architecture using a minimal set of extensions to the C language by allowing programmers to define C functions, called kernels.

In general, unlike CPU, a GPU has more transistors dedicated to data processing than to data caching and flow control. A basic building block of NVIDIA GPUs is a multiprocessor with 8 cores, up to 16384 32-bit registers, 16KB memory shared between 1024 co-resident threads (a multiprocessor executes a block of up to 16 warps, comprising of up to 32 threads, simultaneously). Conditional execution is possible, but groups of 32 threads (a thread warp) must execute the same instruction in SIMD fashion. With up to 240 cores (30 multiprocessors) and memory bandwidth up to 102 GBps, the latest generation of GPUs offers extremely cost-effective computational power not only for visualization but also for general purpose scientific computations.

NVIDIA GPU memory model is highly hierarchical and there exist per-thread local memory, per-thread-block shared memory and device memory which aggregates global, constant and texture memory allocated to a grid, an array of thread blocks. A thread executes a kernel, GPU program, and communicates with threads in the same thread block via high-bandwidth low-latency shared memory. Generally, optimizing the performance of CUDA applications could involve optimizing data access patterns to these various memory spaces. Each of the memory space has certain performance characteristics and constraints. To have an efficient implementation is necessary to consider carefully CUDA memory spaces, specifically, local and global memories which are not cached and have high access latencies. The CPU, host, can read and write global memory but not to shared memory.

A kernel is a function called from CPU that runs on the GPU. A CUDA kernel is executed by an array of threads, all threads run the same code and each thread has an ID that it uses to compute memory addresses and make control decisions. A kernel launches a grid of thread blocks. Threads within a block cooperate via shared memory, however threads in different blocks cannot cooperate.

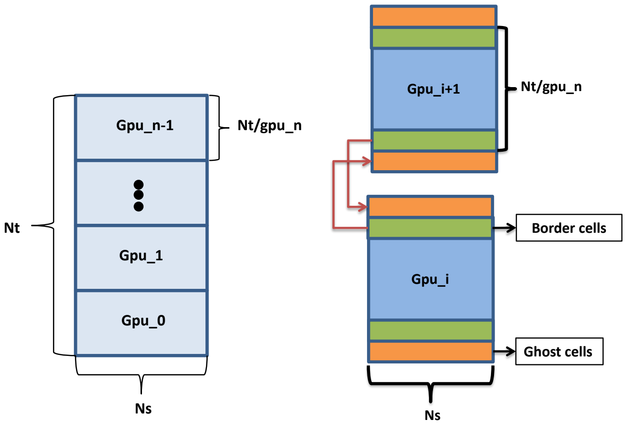
PoS(Lattice 2010)024

4. Results

We implement our code in CUDA language to run in one GPU or in several GPU's. The code was tested in two NVIDIA 295 GTX cards, each card has two GPU's inside. Due to the data access patterns and the limited amount of the shared memory available, we use texture memory to load data. Note that the texture memory can only be read inside the kernels and updated between kernels. The lattice array is a `float4` type array and the four sites of the `float4` are used to store the a_0, a_1, a_2 and a_3. All the C/CUDA operators (plus, multiplication, trace, determinant, and so on) was developed using the SU(2) generators.

The CUDA code implementation consists of 7 kernels, one kernel to generate and store random numbers, this part is based on [5], two kernels to the type of the initialization of the lattice, cold start or hot start, a kernel to perform the heat bath method, a kernel to compute the plaquette at each lattice site, a kernel to compute the Polyakov loop at each lattice site and a kernel to perform the sum over all the lattice sites for the plaquette and/or Polyakov loop.

The Multi-GPU part was implemented using CUDA and OPENMP, each CPU thread controls one GPU. Each GPU computes $Ns \times \frac{Nt}{num.gpu's}$, the total length of the array in each GPU is $Ns \times (\frac{Nt}{num.gpu's} + 2)$, see figure 1. At each iteration, the links are calculated separately by even and odd sites and direction, μ, before calculate the following direction, the border cells need to be exchanged between each GPU. In order to exchange border cells between GPU's it is necessary to copy these cells to CPU memory and then synchronize each CPU thread with the command `#pragma omp barrier` before updating the GPU memory, ghost cells.

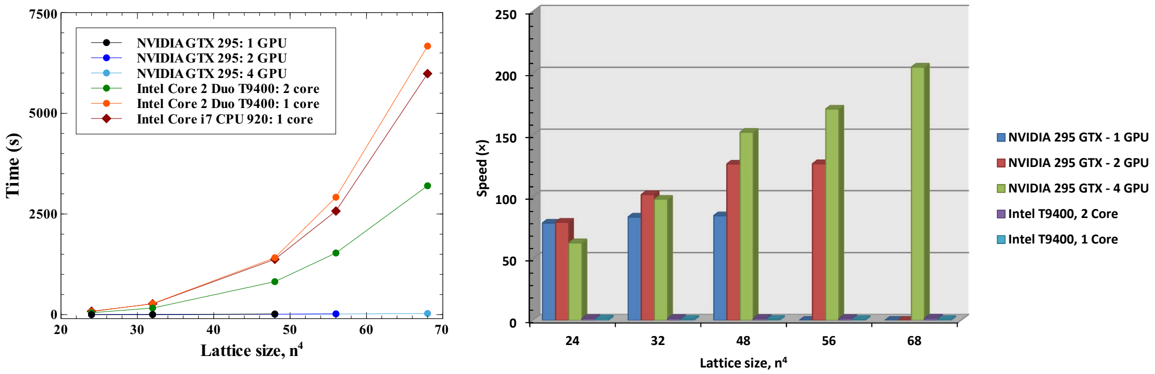


(a) (b)

Figure 1: Schematic view of the lattice size part in each GPU.

4.1 GPU Performance

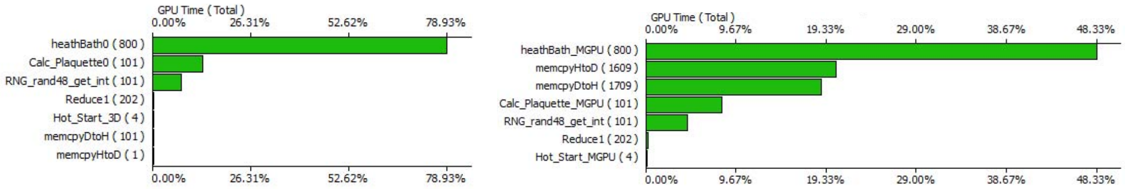
In order to test the GPU performance, we measure the execution time for the CUDA code implementation in one, two and four GPU's and the serial code in CPU (Intel Core i7 CPU 920 and Mobile DualCore Intel Core 2 Duo T9400) for different lattice sizes at $\beta = 6.0$, hot start initialization of the lattice, 100 iterations of the heat bath method and the calculation of the mean average plaquette at each iteration, see figure 2. In figure 2a, we show the execution time and in figure 2b the speed performance with the Intel Core i7 CPU 920 with only one core. With NVIDIA Visual Profiler it is possible to analyze the percentage taken by each kernel of the code, see figure 3.



(a) Execution time.

(b) Speed over Intel Core i7 CPU 920 with 1 core

Figure 2: Execution time, (a), and speed performance over one core CPU, (b), for different lattice sizes at $\beta = 6.0$, hot start initialization of the lattice, 100 iterations of the heat bath method and the calculation of the mean average plaquette at each iteration



(a) Summary Plot with a single GPU for a 32^4 lattice

(b) Summary Plot with 4 GPU's for a 56^4 lattice

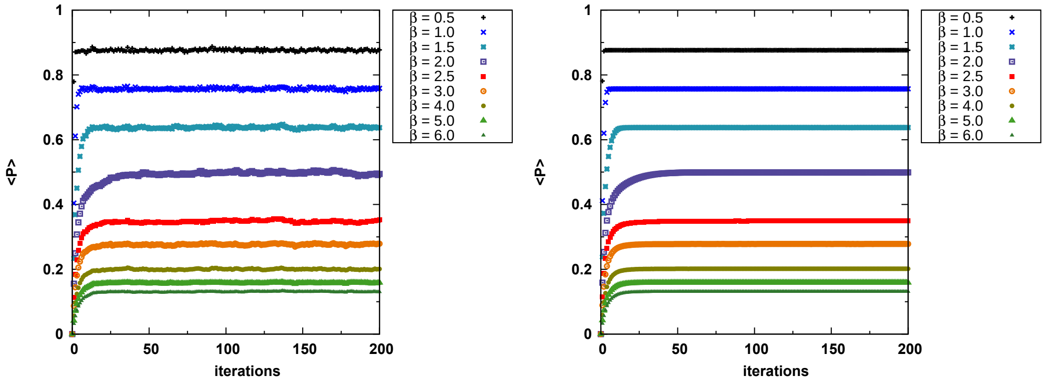
Figure 3: Performance with NVIDIA Visual Profiler. memcpyHtoD means memory transfers from Host (CPU) to Device (GPU) memory; memcpyDtoH means memory transfers from Device to Host memory; Hot_Start* is the kernel to initialize the link of the lattice site with a random SU(2) matrix; heatbath* is the kernel for the heat bath method (the total number is given by the number of iterations (100) times the number of directions (4) times two (first the even then the odd sites are computed which gives 800); Calc_Plaquette* is the kernel for compute the plaquette for every lattice site; Reduce1 is the kernel to perform the sum of the plaquette in the entire lattice; RNG_rand48_get_int is the kernel to generate random numbers

4.2 Mean Average Plaquette and Polyakov Loop

In this section we present the mean average plaquette at each heat bath iteration, figure 4, and the mean average plaquette for each β for 5000 configurations, 5 with different lattice sizes.

In figure 6 we present the mean average Polyakov loop for several β and temporal lattice sizes, 6a, and the mean average Polyakov loop versus the temperature (T/T_c), 6b, for 200 000 heat bath iterations.

PoS(Lattice 2010)024



(a) Lattice size 8^4

(b) Lattice size 48^4

Figure 4: Mean plaquette per iteration for different β and two lattice sizes, 8^4 and 48^4, with cold start initialization and 200 iterations of the heat bath method.

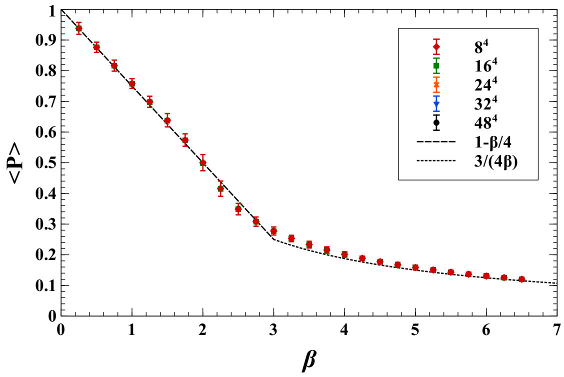


Figure 5: β dependence of the average plaquette from Monte Carlo simulation for several lattice sizes (data points) and analytic predictions (denoted by dashed lines).

5. Conclusions

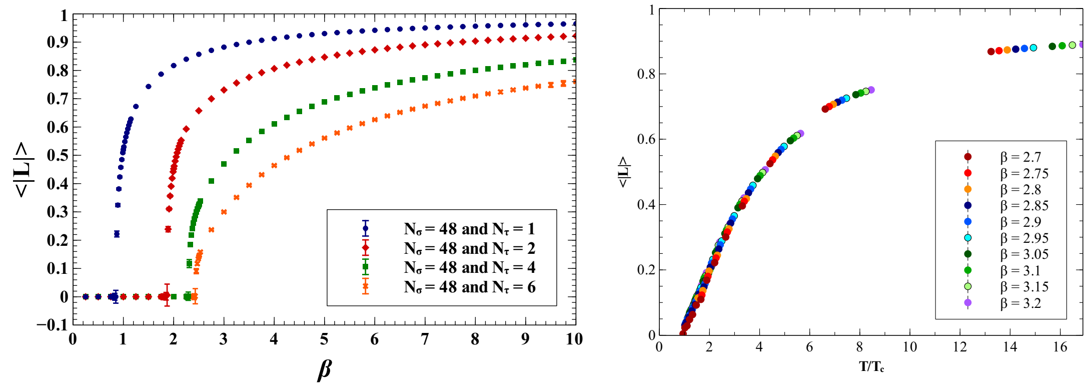
With 2 NVIDIA GTX 295, we were able to obtain more than 200× the performance over the Intel Core i7 CPU 920 with 1 core for a 68^4 lattice using single precision.

It's not possible to generate SU(2) configurations using only GPU shared memory due to the limited amount of shared memory available. The limited number of registers also affects the GPU performance. Using texture memory in this problem, we were able to achieve the high performance.

The occupancy and performance of the GPU's is strongly linked to the number of threads per block, registers per thread, shared memory per block, memory access, read and writing, patterns. To maximize performance it is necessary to ensure that the memory access is coalesced and to minimize copies between GPU and CPU memories.

Using texture memory there is no problems with coalesced memory, it cannot be updated inside kernels but only updated between kernels, it is very useful for complicated problems where memory access (read/write) cannot be achieved in a coalesced pattern. However, the hardware doesn't support double precision as texture format, it is possible to use `int2` and `__hiloint2double`

PoS(Lattice 2010)024



(a) Mean average Polyakov loop versus β. β dependence of the average Polyakov loop from Monte Carlo simulation for different temporal lattice sizes.

(b) Mean average Polyakov loop versus T/T_c.

Figure 6: Mean average Polyakov loop

and cast it to double. This was not implemented in the code.

NVIDIA Visual Profiler is a useful tool to help measure execution time, performance, number of registers and shared memory used by each kernel and other functions.

Although NVIDIA 295 GTX supports double precision, the performance is much slower because it has only one double precision unit for each multiprocessor. We expect that the new generation of the NVIDIA cards, the Fermi architecture, is better for double precision calculations with 8× the peak double precision floating point performance over previous ones.

Acknowledgments

This work was financed by the FCT contracts POCI/FP/81933/2007, CERN/FP/83582/2008, PTDC/FIS/100968/2008 and CERN/FP/109327/2009. We thank Marco Cardoso and Orlando Oliveira for useful discussions.

References

[1] M. Creutz, Monte Carlo study of quantized SU(2) gauge theory, Phys. Rev. D21, 2308-2315 (1980).

[2] M. Creutz, Confinement and lattice gauge theory, Physica Scripta 23, 973-977 (1981).

[3] `http://www.nvidia.com/object/cuda_home_new.html`

[4] NVIDIA, NVIDIA CUDA$^{\mathrm{TM}}$ Programming Guide, Version 3.0, 02/20/2010.

[5] van Meel, J. A., Arnold, A., Frenkel, D., Portegies Zwart, S. & Belleman, R. G. (2008). Harvesting graphics power for MD simulations. Molecular Simulation, 34(3), 259-266. doi:10.1080/08927020701744295

PoS(Lattice 2010)024