# Lattice simulation on graphics cards

Győző I. Egri[a], Zoltán Fodor[ab], Christian Hoelbling[a], Sándor D. Katz[b], Dániel Nógradi[*a] and Kálmán K. Szabó[a]

[a]*Institute for Theoretical Physics, Eötvös University, Hungary*
[b]*Department of Physics, University of Wuppertal, Germany*
*E-mail:* nogradi@physik.uni-wuppertal.de

The speed, bandwidth and cost characteristics of today's PC graphics cards make them an attractive target as general purpose calculational platforms. I will outline the advantages and challenges of performing lattice simulations on such a hardware architecture and compare it with the traditional CPU based platforms. Sample code is also given.

*XXIVth International Symposium on Lattice Field Theory*
*July 23-28, 2006*
*Tucson, Arizona, USA*

*Speaker.

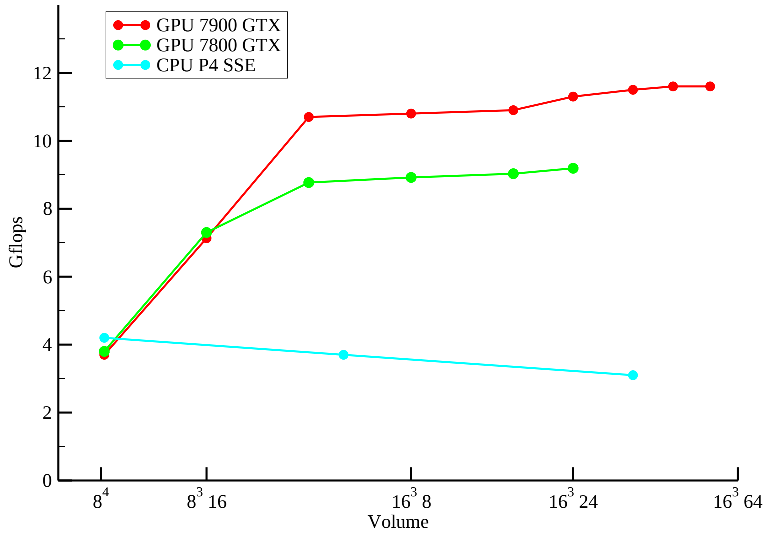

**Figure 1:** Wilson matrix multiplication performance for various lattice volumes on Nvidia 7900 GTX, 7800 GTX cards and a 3 GHz Pentium 4 using SSE optimization.

## 1. Introduction

The goal of every lattice field theorist is to use a calculational platform that maximises the performance/price ratio. In this paper a competitive but so far unused and unappreciated (at least in the lattice community) architecture will be introduced.

So far the only available option was the usage of CPU-based platforms may it be individual PC's, PC clusters or dedicated supercomputers such as QCDOC or BlueGene/L. The actual calculational task in all of these solutions is done by CPU's which significantly vary in terms of features but share essentially the same architecture. In recent years a rapidly developing specialized architecture emerged from the graphics industry, Graphical Processing Units or GPU's which took over some of the calculational tasks of the CPU. These chips are designed to fulfill the needs of a graphics oriented audience (gamers, designers, etc.) and hence were specialized to the kind of task this set of users most frequently need i.e. graphical processing. However the complexity of this task grew to a level that general programmability of the chips was also required. The end product of this evolution is a high performance chip optimized for SIMD floating point operations on large vectors that can be utilised for general purpose calculations such as lattice field theory.

The price of the current top models is around $500 and considering figure 1. for sustained performance it becomes clear that GPU's can be very cost effective while achieving much higher performances than CPU's. For reference we give some numbers from figure 1. for the newer Nvidia 7900 GTX card: 11.6 Gflops sustained performance on a $16^3 \times 60$ lattice using the Wilson kernel. Another good reason for investigating graphics hardware is the fact that performance increase for CPU's has slowed down in the last 5-6 years whereas the growth for GPU's is still a steep exponential thus it is expected that in the future the gap between CPU's and GPU's will continue to grow [1].

Much more details on general purpose programming of graphics hardware than what is covered here can be found in [1], see also [2] and a tutorial at [3].

## 2. Architecture

In order to understand the basics of GPU programming and the techniques of writing efficient code the architecture of modern GPU's will be described briefly below; for more details see [1].

The native data type is a 2-dimensional array called a texture. In graphics applications such as games the content of a texture is typically displayed on the screen after a set of transformations on it. On modern cards these transformations can be (almost) freely programmed which allows us to code for instance lattice applications. Each texture has a width and height and consist of width $\times$ height number of pixels. There are various types of textures but in our lattice application we use one which contains 4 floating point numbers per pixel which are the RGBA (red, green, blue, alpha) colour channels.

In any calculation a set of textures is used as input and another set of textures are calculated as output. The calculation is such that the outgoing pixels can be calculated independently and can not refer to each other. This constraint allows for massively parallel execution of the code, i.e. a large number of outgoing pixels (as well as the 4 RGBA values if possible) are computed at the same time using fragment processors. This feature together with the two facts that (1) 24 or more fragment processors are included in a single modern GPU and (2) the availability of a very high memory bandwidth (51.2 GB/s for Nvidia 7900 GTX) is the source of the high performance of current chips.

The question might still arise how it is possible that the growth rate is much higher for GPU's than CPU's since both chips are the product of the same semiconductor industry. Part of the answer is that in a CPU many transistors are busy with non-computational tasks such as branch prediction, caching, etc, while in a GPU almost all transistors are calculating [1].

There are other components in a modern graphics card besides the fragment processors such as vertex processors but we did not find these useful for our lattice code.

## 3. Programming

In order to exploit the unusual but powerful architecture of GPU's a somewhat unusual programming model must be used. In this section the basics of GPU programming will be introduced with emphasis on the parts that are needed for a lattice application.

The basic software framework for interacting with the GPU from the CPU is OpenGL [4]. It is a graphics library containing functions and macros that can be used to manage textures on the GPU. For instance OpenGL function calls create, delete, bind, select textures as well as set their properties. The core OpenGL library has many extensions typically invented by the chip vendors that are thus tailored to a specific hardware. Once such an extension becomes wide spread it may become part of the OpenGL core specification. A general OpenGL extension registry can be found in [5] while the Nvidia specific extensions are available from [6].

Another important segment of GPU programming is the graphics driver that comes with the actual cards and should be provided by the chip vendors in order to exploit all hardware features. Unfortunately most high quality drivers are closed source. Needless to say that developers like lattice field theorists using GPU's for general purpose calculations would benefit immensely from some degree of openness regarding both the hardware and the driver.

The drivers are important in that every OpenGL call or any instruction from the CPU is given to the driver which then decides what exactly is passed to the GPU. For instance it may optimize a set of calls into simpler calls if possible. Usually such an optimization can make a drastic effect on performance thus the usage of the latest drivers (specific to the given chip) is recommended. Since from the two major vendors, ATI and Nvidia, only Nvidia offers high quality drivers for Linux we are using exclusively Nvidia hardware.

The third and perhaps the most important segment of GPU programming is the actual computation on the textures, i.e. the code that specifies how a set of outgoing textures are computed from a set of incoming textures. This kind of code is typically called a pixel shader and can be written in a number of ways. The crudest option is the usage of a GPU-specific assembly-like language. A higher-level alternative is a C-like language invented by Nvidia called Cg [7] while there are some even higher-level tools like Brook/BrookGPU [8] or Sh [9]. In our lattice code we used Cg.

## 4. The lattice on a GPU

We currently have production code for both dynamical overlap (Wilson) and staggered fermion formulations. In this section the basic layout of the overlap code will be outlined.

If the lattice dimensions are $N_t$, $N_x$, $N_y$ and $N_z$ then each texture will have width $N_tN_x$ and height $N_yN_z$ thus a lattice site will correspond to a pixel.

The gauge links in a given direction are given by 9 complex or 18 real numbers per site. These 18 numbers fit into 5 textures since each pixel can store 4 floating point numbers (the RGBA colour channels) in the type of textures we use. In fact the 5 textures can store $5 \cdot 4 = 20$ numbers so along with the 18 links components there are 2 extra slots. These can be used to store the 2-dimensional texture location of its nearest neighbour in the given direction. Thus over all the gauge links are stored in 20 textures, 5 for each direction.

A Wilson vector has 4 Dirac and 3 colour components per site making up 24 real components. These can be stored in 6 textures.

In the HMC algorithm one needs to calculate the force which is an anti-hermitian traceless matrix for each direction. There are 8 independent real components per direction, thus over all the force is stored in 8 textures.

Once the storage layout is determined as above one only needs to write pixel shaders in Cg to do the actual computations such as multiplying by the Wilson kernel, calculating the force, etc. In our overlap code all fermionic computations are done on the GPU while in the staggered code even parts of the gauge update is on the GPU. This was necessary because the gauge part started to become the bottle neck as the time spent on this became a larger and larger portion of the total execution time as we optimized the fermionic GPU code.

## 5. Present difficulties and future prospects

There are a number of unpleasant features and limitations at present that unavoidably effect anyone attempting to do lattice simulations on a GPU.

At present each of our cards are running separately and there is no communication between the nodes. We hope to introduce some solution for communication either between several cards within one node or between cards in separate nodes. A second issue is that the current top Nvidia model is the 7900 GTX which has only 512 MB memory. This will certainly change in the future.

Regarding precision, most GPU's at present are only capable of handling single precision numbers which is however not a problem for two reasons. In a lattice simulation single precision is often enough and also when it is not one can still use the double precision available on the CPU. For example in a conjugate gradient algorithm one can restart the inverter and thus do thousands of iterations quickly on the GPU in single precision and only do a some small number of double precision calculations on the CPU. In this way the end result will be correct to double precision however the majority of the instructions were done in single precision. It is likely that the support for double precision calculations will be substantially enhanced in the future.

It is somewhat of an inconvenience that there is very limited support for conditional statements and loops and they drastically lower the performance. However we did not find this to be a serious limitation in our code as we completely eliminated every conditional statement and loop from the pixel shaders (Cg code).

In practice a further slight annoyance is the lack of simple debugging as the programmer does not have direct access to the GPU memory only through the driver. It is recommended to use the built-in debugging features of OpenGL and Cg.

As to the future, the hardware, corresponding driver and OpenGL extension improvements on the side of the vendors is rather unpredictable except for the obvious trend to improve the existing features (more memory, more bandwidth, faster clock speed, etc.). An example of a useful recently added OpenGL extension is the Pixel Buffer Object extension or PBO [10]. This allows for asynchronous readbacks from the GPU and saves one data copy from the driver controlled memory to the programmer controlled memory. In our lattice code we found it very advantageous to use this new extension because it considerably increases the GPU - CPU bandwidth.

## 6. Sample code

A full working code that performs the operation $z_i = x_i + y_i$ on the GPU for three $4NM$ sized single precision arrays is described below. In order to use the code the following software must be installed on a Linux system: glew, freeglut, Cg and an approriate driver.

As mentioned in section 3. each data array $x, y$ and $z$ will be stored in textures. Since their size was chosen to be $4NM$ the size of the textures will be $N \times M$ since each pixel can store 4 floating point numbers.

First we give a Cg program to add two textures and put the result into a third texture. The code can be downloaded from [11] and we assume it is saved under `add.cg`.

Notice that the line containing the addition adds 4 floating point numbers in one go since each texture lookup, texRECT( ... ), results in a structure containing all the 4 numbers stored in a pixel.

Once the pixel shader is ready the actual C program containing all OpenGL calls necessary to create the textures, load the initial data $x$ and $y$ from the CPU memory to the GPU memory, run the above shader and retrieve the result from the GPU memory back into CPU memory, finally comparing the result of the addition with the same result obtained on the CPU should be written. This C code can be downloaded from [12] and may be saved as `add.c`.

Once both `add.cg` and `add.c` are ready `add.c` can be compiled in the usual way and then should be linked with the following options `-lCg -lCgGL -lGL -lglut -lGLEW -lpthread`. The resulting executable should be run from the directory where `add.cg` is located. If no error occurs the output will indicate that indeed the calculation on the GPU and on the CPU was the same.

## Acknowledgements

We would like to thank Dominik Göddeke for enlightening discussions on the ins and outs of general purpose GPU programming. In addition László Szirmay-Kalos and Tamás Umenhoffer are kindly acknowledged for discussions in the early stages that greatly helped kick starting our project.

## References

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn and T.J. Purcell in Eurographics 2005, State of the Art Reports, 21-51, http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=844

[2] ed. Matt Pharr, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley Professional, 2005

[3] Dominik Göddeke, General Purpose GPU Tutorial, http://www.mathematik.uni-dortmund.de/%7egoeddeke/gpgpu/tutorial.html

[4] M. Segal, K. Akeley, The OpenGL Graphics System: A Specification, http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf

[5] OpenGL Extension Registry, http://oss.sgi.com/projects/ogl-sample/registry/

[6] NVIDIA OpenGL Extension Specifications, http://developer.nvidia.com/object/nvidia_opengl_specs.html

[7] Cg Specification, Reference Manual, Users Manual, http://developer.nvidia.com/object/cg_toolkit.html

[8] Brook, http://www.gpgpu.org/w/index.php/Brook; BrookGPU, http://graphics.stanford.edu/projects/brookgpu/

[9] Sh, http://libsh.org/

[10] Nvidia Technical Brief, Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL, http://download.nvidia.com/developer/Papers/2005/Fast_Texture_Transfers/Fast_Texture_Transfers.pdf

[11] The pixel shader in Cg, http://pos.sissa.it//archive/conferences/032/034/LAT2006_034_a1.cg

[12] The main C code, http://pos.sissa.it//archive/conferences/032/034/LAT2006_034_a2.c