Blasting Through Lattice Calculations using CUDA

Kipton Barros, Ronald Babich, Richard Brower, Michael A. Clark, and Claudio Rebbi
Boston University

ABSTRACT

Modern graphics hardware is well suited to highly parallel numerical tasks and provides significant cost and performance benefits. Graphics hardware vendors are now making available development tools to support high performance computing. NVIDIA'S CUDA platform, in particular, offers direct access to graphics hardware through a programming language similar to C. Using the CUDA platform we have implemented a Dirac-Wilson operator which runs at an effective 68 Gigaflops on the Tesla C870 GPU. The recently released GTX 280 GPU runs this same code at 92 Gigaflops and we expect improvement pending code optimization.

1 Introduction

- Moore's law, the doubling of computer power every year and a half, can no longer be sustained with speed-up at the individual processor level. Multiple processor and/or multiple cores per processor are required.
- Even with multiple processors, memory bandwidth is a serious limitation, preventing massively parallel applications from achieving more than a small percentage of peak performance.
- Programmable graphics processors, originally developed to perform simple operations on a large number of data elements in parallel take advantage of multiple cores and very high memory bandwidth to deliver astounding performance on fine-grained massively parallel applications.

Multiple cores and very high memory bandwidth are the key.

2 The NVIDIA HPC Graphics Cards

- NVIDIA's Tesla line of graphics cards

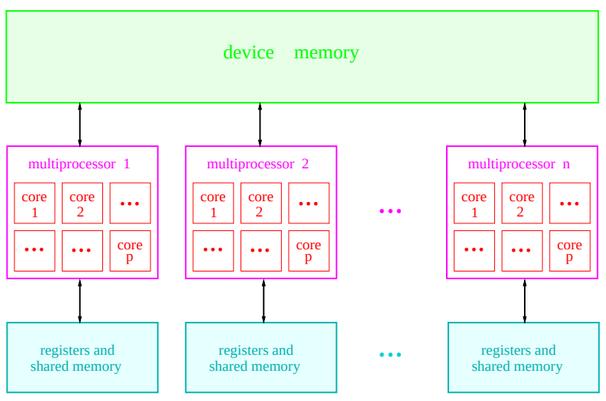
  The Tesla line of graphics cards is designed specifically for high performance computing (HPC.) [1] In fact, these Tesla cards do not have a video output function, so their sole capability is general purpose computing. There are two generations of the Tesla: the C870 and the C1060 cards. The latter will be released in August 2008.

- NVIDIA's consumer line of graphics cards

  NVIDIA'S core market is graphics cards which are designed to accelerate 3D rendering, primarily video games. There are, roughly speaking, two generations of consumer level graphics cards. The flagship products of the previous and current generation are, respectively, the 8800 GTX and the GTX 280. These cards directly parallel the Tesla line: the 8800 GTX corresponds to the Tesla C870 and the GTX 280 corresponds to the Tesla C1060. The consumer line of graphics cards has higher bandwidth than the Tesla line but has less total device memory.

- Hardware architecture

  All the cores on the card live on a single GPU (Graphics processing unit) and are grouped into multiprocessors. For example the GPU in the Tesla C870 card contains 16 multiprocessors with 8 cores each. All cores in a multiprocessor execute the same instruction, although they can address different data. All cores in a multiprocessor have access to local registers and to a very fast shared memory. All cores on the card can access a device memory, which is common to the card. The card can of course communicate with the host computer and read from and write to the memory of the host, but access to the host computer memory is much slower than access to the device memory. The global device memory has a high latency and best performance is achieved when groups of 16 threads access a contiguous region of memory. Two read-only caches are available: a small but very fast constant cache, and a texture cache for global device reads with spatial locality.



- Specifications and comparison

Card	Cores	Bandwidth (GiB/s)	GFLOPS	Device Memory
8800 GTX	128	86.4	518.0	768 MB
Tesla C870	128	76.8	518.4	1.5 GB
GTX 280	240	141.7	933	1 GB
Tesla C1060	240	102	~ 933	4 GB

3 The CUDA Programming Model

- "CUDA (Compute Unified Device Architecture) is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device" (from NVIDIA's CUDA Programming guide.) [2, 3]
- CUDA uses a programming language which is an extension of C, with a special compiler, nvcc, as well as a hardware driver and a runtime library. Higher level libraries are also provided.
- The main idea of CUDA is that it spawns a very large number of threads, which execute in parallel. For example, in a LGT application one might have one thread for each lattice site (or each site of the odd or even sublattices.) The user controls the number, organization and memory usage of the threads by an extension of the C function calls. For example `dslashKernel <<< gridDim, blockDim, N_BT>>> (args);` will invoke the function `dslashKernel(args)`, which will execute on many individual threads.
- Threads are grouped into thread blocks and the collection of thread blocks is called a grid. The instruction above tells the GPU to launch a kernel using `gridDim` blocks, each containing `blockDim` threads. `N_BT` tells the compiler to allocate N_BT bytes of shared memory for each block. This shared memory is explicitly managed by the programmer and allows rapid communication for threads within a thread block. This is possible because each thread block executes entirely on a single multiprocessor and can be programmatically synchronized. In addition to shared memory, threads have access to local registers and also the global device memory. Communication between thread blocks is only possible between kernel invocations, and occurs through device memory.
- The GPU will dynamically schedule thread blocks to be executed on the multiprocessor. It is desirable to have a high multiprocessor occupancy: that is, to have many threads executing in parallel. This allows, for example, hiding the high device memory latency. Each thread block has fixed register and shared memory resource requirements, and decreasing these requirements can improve the multiprocessor occupancy. We have found that an occupancy of 192 threads per multiprocessor is near to optimal on the C870 card.
- Conditional execution is possible, but groups of 32 threads (a thread warp) must execute the same instruction in SIMD fashion.

4 Wilson Inverter: Main features

- The power of GPUs has already been harvested to speed up LGT calculations (cfr. [4]), but the task is made much easier with CUDA, which also helps to write more efficient code.
- We implemented code, authored by Kipton Barros, which uses CUDA to solve the Dirac-Wilson equation on the lattice. We use an odd-even preconditioned conjugate gradient solver. The conjugate gradient driver runs on the host computer and calls CUDA code to implement the action of the Dslash operator (the Wilson-Dirac operator acting on the spinor variables at the even sites) or its Hermitian conjugate.
- The action of Dslash and Dslash† on a spinor field is the most compute intensive part of the code, and this is run on the GPU. With CUDA we spawn one thread for each site in the (even or odd) sublattice. The thread are executed in highly parallel fashion, achieving a remarkable performance.
- The execution speed is limited by the memory bandwidth between device memory and multiprocessor memory. Memory bandwidth limitation is a common feature of most lattice QCD applications. Here the very large memory bandwidth in the GPU helps achieve a high sustained performance. Also, we structured the calculation in a manner which helps reduce the amount of transferred data (we mostly use 32 bit precision, limiting double precision calculations to global sums in the conjugate gradient driver, we gauge fix to the temporal gauge, we store only two columns of the gauge field variables, reconstructing the third on the fly, etc.), and maximizes the overlap of computation and communication.
- The global sums in the conjugate gradient driver are also implemented on the GPU, by parallel reduction.
- Here we are mostly concerned in the implementation and performance of the Dslash and Dslash† operators. The overall Dirac-Wilson solver requires a large volume of data transfer between the host computer memory and the memory of the GPU and therefore cannot achieve as high a performance as the code for Dslash. Still, we find that the whole solver runs at well over 80% of the speed of just its Dslash component.
- The actual CUDA code was produced by using the "Scala Programming Language" as a higher level code generator. [5]

5 Wilson Inverter: Example Code

Preliminary declarations:

```
#include <stdlib.h>
#include <stdio.h>
```

"qcd.h" contains the size of the lattice etc.:

```
#include "qcd.h"
#define BLOCK_DIM (64) // threads per block
#define GRID_DIM (Nh/BLOCK_DIM) // Nh threads in total
#define SPINOR_BYTES (Nh*spinorSiteSize*sizeof(float))
#define PACKED_GAUGE_BYTES (4*Nh*packedGaugeSiteSize \
      *sizeof(float))
```

Associate variables with the read-only texture cache for better performance:

```
texture<float4, 1, cudaReadModeElementType> gauge0Tex;
texture<float4, 1, cudaReadModeElementType> gauge1Tex;
texture<float4, 1, cudaReadModeElementType> spinorTex;
```

Declarations of `dslashKernel` and `dslashDaggerKernel` as multi-threaded functions (the function bodies are contained in separate files for convenience):

```
__global__ void
dslashKernel(float4* g_out, int oddBit) {
   #include "dslash_core.cu"
}
 __global__ void
dslashDaggerKernel(float4* g_out, int oddBit) {
   #include "dslash_dagger_core.cu"
}
```

Example of loading a spinor field on one of the two (even or odd) half-lattices from the host computer to the global memory of the GPU:

```
CudaPSpinor loadParitySpinor(float *spinor) {
   CudaPSpinor ret;
   cudaMalloc((void**)&ret, SPINOR_BYTES);
   float4 *packed = (float4*) malloc(SPINOR_BYTES);
```

Pack spinors on odd or even sites, on host:

```
   packParitySpinor(packed, spinor);
   cudaMemcpy(ret, packed, SPINOR_BYTES,\
      cudaMemcpyHostToDevice);
   free(packed);
   return ret;
}
```

The code in `dslashKernel`, included from the file dslash_core.cu:

```
#define SHARED_FLOATS_PER_THREAD 25
#define SHARED_BYTES (BLOCK_DIM* \
      SHARED_FLOATS_PER_THREAD *sizeof(float))
```

`I0.x I0.y I0.z I0.w` are the components of a variable local to the thread, which are aliased to i00_re etc. for convenience of coding:

```
#define i00_re I0.x
#define i00_im I0.y
#define i01_re I0.z
#define i01_im I0.w
...
```

This is how the thread variable is read from device memory, making use of the texture cache:

```
#define READ_SPINOR(spinor) \
   float4 I0 = tex1Dfetch((spinor), sp_idx + 0*Nh); \
   float4 I1 = tex1Dfetch((spinor), sp_idx + 1*Nh); \
...
```

Later we invoke:

```
   READ_SPINOR(spinorTex);
```

And this is a segment from the action of the Dirac operator. The neighboring spins are gauge transported to the site and added to the output spinor:

```
// project spinor into half spinors
float a0_re = +i00_re+i30_im;
float a0_im = +i00_im-i30_re;
float a1_re = +i01_re+i31_im;
float a1_im = +i01_im-i31_re;
float a2_re = +i02_re+i32_im;
float a2_im = +i02_im-i32_re;

float b0_re = +i10_re+i20_im;
float b0_im = +i10_im-i20_re;
float b1_re = +i11_re+i21_im;
float b1_im = +i11_im-i21_re;
float b2_re = +i12_re+i22_im;
float b2_im = +i12_im-i22_re;

// read gauge matrix from device memory
READ_GAUGE_MATRIX(gauge0Tex);
```

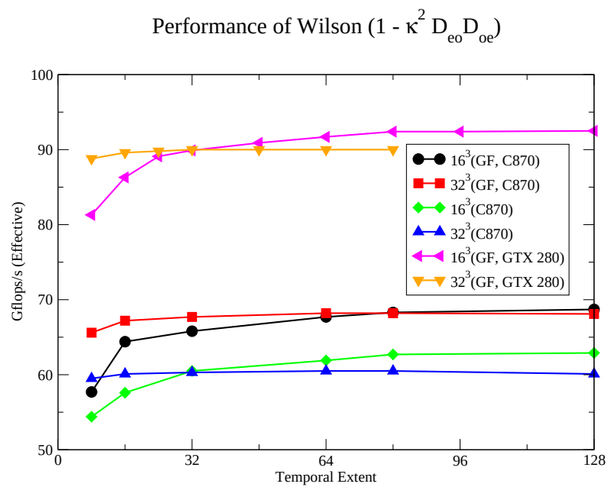
... continued on next box

Wilson Inverter: Example Code continued ...

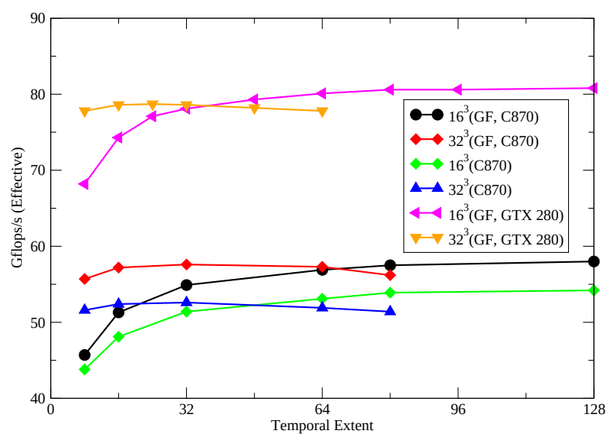
```
// multiply row 0 by half spinors
{
float A_re = + (g00_re * a0_re - g00_im * a0_im \
)     +(g01_re * a1_re - g01_im * a1_im) \
    +(g02_re * a2_re - g02_im * a2_im);
float A_im = + (g00_re * a0_im + g00_im * a0_re) \
    +(g01_re * a1_im + g01_im * a1_re) \
    +(g02_re * a2_im + g02_im * a2_re);
float B_re = + (g00_re * b0_re - g00_im * b0_im) \
    +(g01_re * b1_re - g01_im * b1_im) \
    +(g02_re * b2_re - g02_im * b2_im);
float B_im = + (g00_re * b0_im + g00_im * b0_re) \
    +(g01_re * b1_im + g01_im * b1_re) \
    +(g02_re * b2_im + g02_im * b2_re);
o00_re += +A_re;
o00_im += +A_im;
o10_re += +B_re;
o10_im += +B_im;
o20_re += -B_im;
o20_im += +B_re;
o30_re += -A_im;
o30_im += +A_re;
}

// multiply row 1 by half spinors

...
```

5 Performance

- Both the even-odd preconditioned Wilson-Dirac operator and the full CG inverter have been benchmarked for a varierty of different volumes. The (GF) label on the plots signify that the gauge fixing trick was used, in this case the GFlop/s number reported are the effective numbers.



- The Wilson Matrix-Vector operation performance is only weakly volume dependent: for all but the smallest volumes the performance sustains above 60 GFlop/s on the C870 and around 90 GFlop/s on the GTX 280. The gauge fixing trick employed to reduce the bandwidth results in around a 10% improvement.



- The CG performance is of course reduced compared to the Wilson kernel, however, for reasonable sized volumes, the GPU code still sustains over 50 GFlop/s and 80GFlop/s on the C870 and GTX 280 respectively performance.
- GPU code performance is more than an order of magnitude greater than typical SSE optimized implementations (Wilson Matrix-Vector < 5 GFlop/s on a 3.0GHz Xeon processor). In addition the scaling with increasing volume is relatively constant, compared to CPU implementations which drastically fall in performance as the local volume is increased out of the cache.
- At 80 GFlops/s sustained CG performance, and $\$650$ per GPU board, the GTX 280 card represents a cost of $0.8¢$/ MFlop/s (excluding host computer cost).

Acknowledgments

This work was supported in part by US DOE grants DE-FG02-91ER40676 and DE-FC02-06ER41440 and NSF grants 0221680, PHY-0427646, and OCI-0749300.

References

[1] See for example http://www.nvidia.com/object/tesla_c870.html

[2] "NVIDIA CUDA Programming Guide", http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[3] Cyril Zeller, "NVIDIA Tutorial CUDA", http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf

[4] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi and K. K. Szabo, "Lattice QCD as a video game," Comput. Phys. Commun. **177**, 631 (2007) [arXiv:hep-lat/0611022].

[5] "The Scala Programming Language", http://www.scala-lang.org/