Accelerating Quantum Chromodynamics Calculations with GPUs

Guochun Shi[†], Steven Gottlieb[†‡], Aaron Torok[‡], Volodymyr Kindratenko[†]
[†]National Center for Supercomputing Applications, University of Illinois, Urbana, IL, USA
[‡]Department of Physics, Indiana University, Bloomington, IN, USA
gshi@ncsa.illinois.edu, gs@indiana.edu, amtorok@indiana.edu, kindr@ncsa.illinois.edu

***Abstract*—We present a CUDA C implementation of the Conjugate Gradient (CG) and multi-mass CG solver from the MILC quantum chromodynamics package to speedup improved staggered quarks computations on NVIDIA GPUs. The implementation is built on the QUDA package from Boston University.**

Keywords- quantum chromodynamics; MILC; GPU

I. INTRODUCTION

The MIMD Lattice Computation (MILC) [1] code, a Quantum Chromodynamics (QCD) application used to simulate four-dimensional SU(3) lattice gauge theory, is one of the largest compute cycle users at many supercomputing centers. Previously, we have investigated how one of MILC's applications can be accelerated on the Cell Broadband Engine [3]. In this work, we discuss how this code can take advantage of newly emerging Graphics Processing Units (GPUs).

There are four main parts of the application that are responsible for over 98% of the execution time: Conjugate Gradient (CG) solver (over 58%), Fermion force (FF) (over 22%), Gauge force (GF) (about 10%), and "fat links" (about 9%). All these kernels achieve between 1 and 3.5 GFLOPS per CPU core on a CPU system [2]. In this short paper, we describe how the CG solver is re-implemented to work on GPUs. Our starting point for this work was the Boston University (BU) implementation of the Wilson-Dirac sparse matrix-vector product and CG solver [6]. We have extended the BU GPU code to include the case of improved staggered quarks, or staggered Dslash operator, used in MILC.

II. CONJUGATE GRADIENT SOLVER

A. Staggered Dslash kernel

Lattice QCD solves the space-time 4-D linear system $M\phi = b$ where $\phi_{i,x}$ and $b_{i,x}$ are complex variables carrying a color index $i = 1,2,3$ and a four-dimensional lattice coordinate x. The matrix M is given by $M = 2maI + D$ where I is the identity matrix, $2ma$ is constant, and the matrix D (called "Dslash") is given by

$$\mathrm{D}_{\mathrm{x,i;y,j}} = \sum_{\mu=1}^{4}\left(\mathrm{U}_{x,\mu}^{F\,i,j}\delta_{\mathrm{y,x+\hat{\mu}}} - \mathrm{U}_{x-\hat{\mu},\mu}^{F\,\dagger\,i,j}\delta_{\mathrm{y,x-\hat{\mu}}}\right) + \sum_{\mu=1}^{4}\left(\mathrm{U}_{x,\mu}^{L\,i,j}\delta_{\mathrm{y,x+3\hat{\mu}}} - \mathrm{U}_{x-3\hat{\mu},\mu}^{L\,\dagger\,i,j}\delta_{\mathrm{y,x-3\hat{\mu}}}\right)$$

where U^F and U^L incorporate the staggered phase factors.

Improved staggered quarks such as asqtad and HISQ require both first and third nearest neighbor terms in the Dirac operator. We call the corresponding links *fat* links and *long* links. In the Dslash operation, all links are read-only and only spinors are updated. There are 4 fat links and four long links for each site, one in each x, y, z, t directions. The opposite link for a site is defined as conjugate transpose of the positive link in its negative neighbor in the same direction. In order to update each site, we need to read the spinors and the positive or negative links in all the positive and negative directions.

Each site contains one spinor, 4 fat links and 4 long links. Each spinor is a three-element complex vector, requiring 6 floats for single-precision implementation. Each link, fat link or long link, is a 3x3 complex matrix, thus requiring 18 floats of storage for single-precision implementation. It is well know that a 3x3 unitary matrix is completely determined by the elements of its first two rows (12 real elements). In fact, it actually determined by only 8 real numbers [7]. The BU QUDA library defines corresponding compression techniques called 12-reconstruct and 8-reconstruct, respectively. We can use these techniques for the long links (even when staggered phases are included), but not for the fat links, as they are not unitary. Finally the even and odd quantities are stored in the first and second half of a memory region to take advantage of the fact that when even spinors are updated, all its first neighbors and third neighbors are on odd sites and vice versa. The data layout in memory used in MILC is shown in Figure 1.

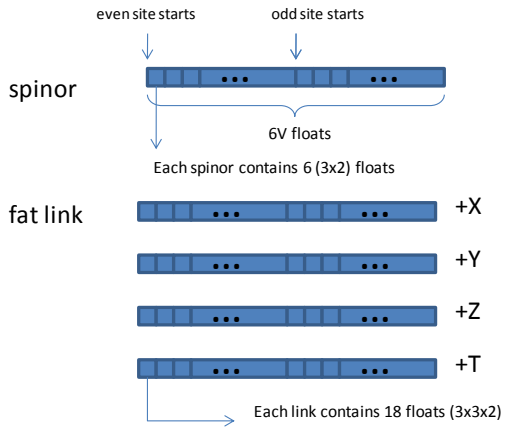

Figure 1. Data layout in host memory for spinor (top) and fat link (bottom).

When implementing the Dslash operation on the GPU, it is important to note that the computation/byte ratio is close to one for single precision. This indicates that the performance of the GPU implementation will be limited by the GPU memory bandwidth. Therefore, it is critical to ensure full memory bandwidth utilization, which ultimately requires coalesced access to the device memory. In long link 12/8 reconstruction, the bandwidth requirement is reduced by performing more computation. This is desirable in single and half precision while counter-productive in double precision, as we will show later. We achieve the coalesced access by aligning data in GPU device memory as follows.

For spinors, which are represented by six 32-bit floating point numbers, we use three float2 values. The three float2 values are stored in memory in a stripe of $Vh + pad$ so that when all threads are reading neighboring spinors in the same odd/even segment, their reads can be coalesced (Figure 2). The pad is introduced to avoid the partition camping problem [9] and is usually one half of temporal face size (x1*x2*x3/2). The data is read either directly from device memory or through texture and we found that a combination of both achieves better performance than loading everything in either way. Specifically, when the fat link is loaded directly and the long link and spinors are loaded through texture the best performance is achieved.

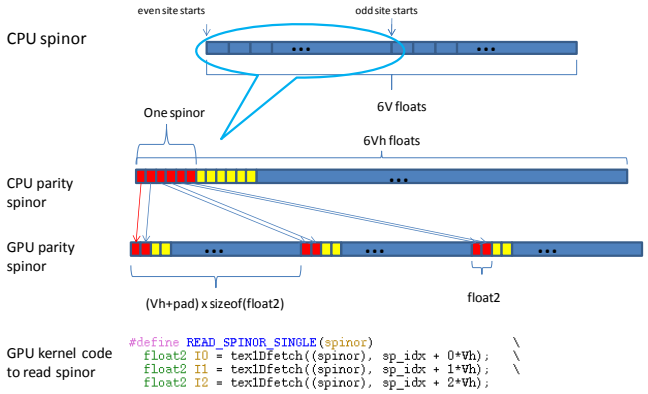

Figure 2. Data layout in the device memory for spinors and GPU code to access the data via texture unit.

The fat link is stored as a 3x3 complex matrix and it is not unitary. Therefore no reconstruction can be done with it and we have to load it completely. We store fat link as nine float2 values. The access method is similar to the one used for spinors except it uses nine float2 values instead of three.

The long link matrix is unitary and thus we can use data compression in the form of 12-reconstruct or 8-reconstruct to restore the full matrix on the fly from a subset of it. Figure 3 shows the data layout for the 12-reconstruct for the long link matrix. The compressed matrix is stored as three float4 values in memory. Each float4 value is stored with stripe of $Vh + pad$ to enable coalesced access. The 4th and 5th float4 are filled with zeros when reading the long link and will later on be computed based on the first three float4 values.

In the Dslash operation, links are not shared and thus opportunities for data reuse are not available. Each spinor, on the other hand, is used 16 times. However, since the spinors are small compared to links, the benefit of reusing them across multiple threads is not as great. Data access requirement for a 12-reconstruct is *spinors read* + *spinor write* + *fat link read* + *long link read*, or $8 \times 6 \times 2 + 6 + 8 \times 18 + 8 \times 12 = 342$ words. If we can reuse the same spinor to the maximum, it is easy to see that data access requirement becomes $(1 \times 6 + 6) + 8 \times 18 + 8 \times 12 = 256$ words, leading to 26.3% memory bandwidth improvement. However, in order to make the sharing data reuse work, we need to rearrange the data so that threads in each thread block compute on neighboring sites in 4D space.

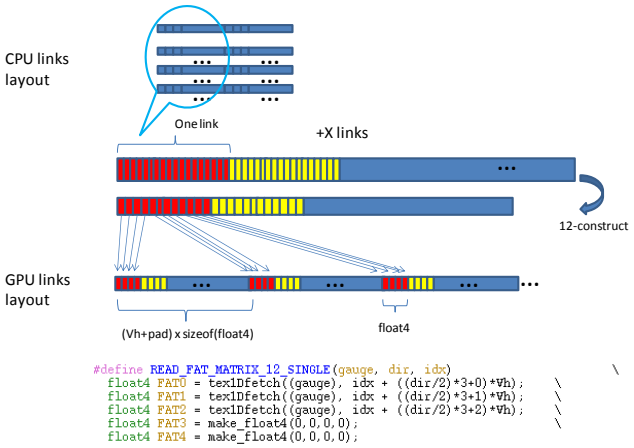

Figure 3. Data layout for the 12-reconstruct for the long link matrix.

Here we only describe 12-reconstruct and single-precision (SP) computations. Implementations of the 8-reconstruct, 18 (no) reconstruct, and double-precision (DP) computations are similar. The half precision is implemented as well using short2/short4 data types with a normalization vector. More details on the mixed-precision implementations can be found in [5].

B. CG

The CG solver is used in MILC to solve the system of linear equations. We implemented parts of the CG subroutine to work on the GPU. The main loop is still executed on the CPU, but all link and spinor data reside in the device memory until convergence to the desirable level is achieved or the maximum number of iterations is executed. The Dslash and other vector operations are executed on the GPU.

Mixed precision is also implemented. The bulk of the work is done using lower precision and only when it reaches a certain threshold, the solution is updated using higher precision. This way, the final solution is sufficiently accurate and overall performance is substantially improved.

C. Multi-mass Solver

Often we need to solve the same linear system for different masses. There are algorithms developed for this case and implemented in MILC CPU code base. The procedure starts with solving the system for one mass and then reusing coefficients, solutions, residues, etc., for other masses. We have ported this procedure to the GPU as well

with one caveat: The mixed precision approach does not work for the multi-mass solver. For the multi-mass solver to be mathematically correct, the residuals of the shifted system must coincide. This constrains us to using zero initial guess for all shifts. When using mixed precision with reliable updates, every time a high precision update takes place, this condition is violated. Hence the multi-shift solver no longer converges for the shifted systems.

D. Interface to MILC

The CG and multi-mass solvers are implemented as a stand-alone library. To interface them with MILC, we have implemented several glue subroutines that perform data conversion/alignment and call GPU-based subroutines. The correctness is verified by comparing the GPU results with the CPU results obtained by the original MILC program. The MILC program can switch to CPU or GPU based routines by defining different macros during the compilation.

E. Dslash and CG performance

Here we report results obtained on a GTX280 NVIDIA GPU using a lattice size of 24×24×24×32. As shown in Table I, with decreasing precision the performance increases. For single precision, changing from 18-reconstruct to 12-reconstruct then to 8-reconstruct, the bandwidth requirement decreases hence the performance increases. However, in the case of double precision, the peak performance for GTX280 is only 77 GFLOPS and 12 and 8-reconstruct introduced extra computations, hence the effective performance decreases although the bandwidth requirement decreases.

TABLE I. DSLASH OPERATION PERFORMANCE

	DP (GFLOPS)	SP (GFLOPS)	Half (GFLOPS)
18-reconstruct	35	85	109
12-reconstruct	32	98	119
8-reconstruct	16	104	128

TABLE II. CG SOLVER PERFORMANCE

Spinor	Link	Recon-struct	Spinor sloppy	Link sloppy	Recon sloppy	Perfor-mance GFLOPS
DP	DP	18	DP	DP	18	33
DP	DP	18	SP	SP	8	91
DP	DP	18	half	half	8	110
SP	SP	8	SP	SP	8	100
SP	SP	8	half	half	8	120
half	half	8	half	half	8	124

Table II shows the performance of the CG solver for different precision and reconstruct techniques. We achieve as high as 100 GFLOPS for single precision and 33 GFLOPS for double precision. In mixed precision mode, we can achieve 110 GFLOPS for double precision accuracy and 120 GFLOPS for single precision accuracy, with "sloppy" computation in half precision. However, as we use sloppy precision to compute, the number of iterations it takes to converge in CG solver increases. Despite this, the overall performance is still better due to the improved Dslash performance with the sloppy precision. The multi-mass solver performance is close to the CG solver performance.

Input spinors, links, and the resulting spinor solution are moved in and out of the GPU memory through PCIe interface. However, this data movement cost is well amortized as it takes hundreds or even thousands of iterations to converge.

III. CONCLUSIONS AND FUTURE WORK

Performance achieved by the CG kernel depends on the data type used and lattice size, but at the same time it is significantly higher than the performance per CPU core on a conventional CPU system. For example, single-precision floating-point implementation of the CG solver achieves 100 GFLOPS on the GPU and just under 3 GFLOPS on the 2.8 GHz Intel Xeon core, or about 33x speedup.

We already implemented the remaining three most time-consuming kernels from MILC and we have done a preliminary analysis of the MILC implementation with regards to multi-GPU and multi-node parallelization strategy. The challenge lays in the parallel CG solver implementation. There are multiple global scatter/gather operations in the current MILC implementation that require data movement between GPUs on multiple compute nodes, which is detrimental to sustaining good overall performance. This is still a subject of further analysis and implementation.

ACKNOWLEDGMENT

This work was sponsored by the Institute for Advanced Computing Applications and Technologies (IACAT) and utilized the AC cluster [8] at the National Center for Supercomputing Applications at the University of Illinois.

REFERENCES

[1] The MIMD Lattice Computation (MILC) Collaboration, http://www.physics.utah.edu/~detar/milc/

[2] S. Gottlieb, Early User Experience—MILC (Lattice QCD), http://denali.physics.indiana.edu/~sg/ncsa_multicore.pdf

[3] G. Shi, V. Kindratenko, S. Gottlieb, The bottom-up implementation of one MILC lattice QCD application on the Cell blade, International Journal of Parallel Programming, vol. 37, no. 5, pp. 488-507, 2009.

[4] D. Roeh, J. Troup, G. Shi, V. Kindratenko, Porting MILC to GPU: Lessons learned, Workshop on using GPUs for LQCD, August 19-21 2009, Thomas Jefferson National Accelerator Facility, Newport News, Virginia, http://www.ncsa.illinois.edu/~kindr/projects/hpca/files/jlab_QCD_on_GPU_presentation.pdf

[5] G. Shi, GPU Implementation of CG solver for MILC, November 2009, internal presentation, http://www.ncsa.illinois.edu/~kindr/projects/hpca/files/GPU_CG_presentation.pdf

[6] M. A. Clark, R. Babich, K. Barros, R. C. Brower, C. Rebbi, Solving Lattice QCD systems of equations using mixed precision solvers on GPUs, http://arxiv.org/abs/0911.3191

[7] B. Bunk, R. Sommer, An 8 parameter representation of SU(3) matrices and its application for simulating lattice QCD, Computer Physics Communications, vol. 40, no. 2-3, p. 229-232, 1986.

[8] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu, GPU Clusters for High-Performance Computing, Proc. IEEE International Conference on Cluster Computing, Workshop on Parallel Programming on Accelerator Clusters, Dec. 2009, doi: 10.1109/CLUSTR.2009.5289128.

[9] G. Ruetsch and P. Micikevicius, Optimizing matrix transpose in CUDA, NVIDIA Technical Report (2009).