Tuning And Understanding MILC Performance In Cray XK6 GPU Clusters

Guochun Shi
National Center for Supercomputing Applications
University of Illinois
Urbana, IL 61801
gshi@ncsa.illinois.edu

Steven Gottlieb
Department of Physics
Indiana University
Bloomington, IN 47405
sg@indiana.edu

Michael T. Showerman
National Center for Supercomputing Applications
University of Illinois
Urbana, IL 61801
mshow@ncsa.illinois.edu

ABSTRACT

Graphics Processing Units (GPU) are becoming increasingly popular in high performance computing due to their high performance, high power efficiency, and low cost. Lattice QCD is one of the fields that has successfully adopted GPUs and scaled to hundreds of them. In this paper, we report our Cray XK6 experience in profiling and understanding performance for MILC, one of the Lattice QCD computation packages, running on multi-node Cray XK6 computers using a domain specific GPU library called QUDA.

QUDA is a library for accelerating Lattice QCD computations on GPUs. It started at Boston University and has evolved into a multi-institution project. It supports multiple quark actions and has been interfaced to many applications, including MILC and Chroma. The most time consuming part of lattice QCD computation is a sparse matrix solver and QUDA supports efficient Conjugate Gradient (CG) and other solvers. By partitioning in the 4-D space time domain, the solvers in the QUDA library enable the applications to scale to hundreds of GPUs with high efficiency. The other computationally intensive components, such as link fattening, gauge force and fermion force computations, are also being ported to GPUs.

Keywords

Lattice QCD, GPU, Krylov solvers, MILC

1. INTRODUCTION: LATTICE QCD AND MILC

Quantum Chromodynamics (QCD) is the quantum field theory that describes the strong interaction among subatomic particles. Lattice QCD (LQCD) is a formulation of the theory that discretizes the four dimensional space-time continuum and allows numerical calculation of the strong force. This is the most successful approach for dealing with the strong interaction in the low energy regime where perturbation theory does not work. Lattce QCD calculations have been consuming a significant portion of the supercomputer time in the US, Europe, and Asia. LQCD is one of the major applications that demands exaflop-speed machines. MIMD Lattice Computation (MILC) is one of the freely available LQCD computation packages, and it has proved popular with US and international researchers. It is imperative that we enable lattice QCD codes to run efficiently on upcoming supercomputers, including the GPU-accelerated machines.

Typical computations are separated into two stages. The first stage is the gauge generation stage, in which snapshots of the QCD gauge fields are computed using Monte Carlo methods. This is run as only a few parallel streams. The second stage is the analysis stage, in which the gauge fields generated in the first stage are used to calculate observables of physical interest. Each stored snapshot (or configuration) can be run as an independent job. Different researchers can use the configurations for different projects, so the second stage can result in thousands of independent moderate size jobs. In the first stage, usually over 50% of the calculation time is spent on the fermion solver, while the rest of the time is spent on so called fatlink, gauge force, and fermion force computations. For example, in a production run on Argonne National Lab's BlueGene/P when generating $64^3 \times 144$ lattice in 8,192 cores, the run time distribution is shown in Table 1.

Table 1: Distribution of time in a production run

Computation	Percentage
CG	53%
FF	27%
GF	9%
fat	10%

In contrast, in the analysis stage, the solver totally dominates the computation. Therefore in both stages, the solver is always the most important computation and optimizing its efficiency receives the most attention. We reported our work on porting the solver into many GPUs in [1]. In this work, we benchmark and profile LQCD code on Cray XK6 nodes and analyze how the features in the XK6 node benefit the application and where potential bottlenecks could come from in the future.

Lattice QCD solves the space-time 4D linear system $M\phi = b$ where $\phi_{i,x}$ and $b_{i,x}$ are complex variables carrying a color

cug2012 April 29 - May 3, 2012, Stuttgart, Germany

index $i = 1, 2, 3$ and a four-dimensional lattice coordinate x. The matrix is given by $M = 2maI + D$ where I is the identity matrix, $2ma$ is a constant related to the quark mass, and the matrix D (called "Dslash" operator) is given by

$$D_{x,i;y,j} = \Sigma_{\mu=1}^{4}(U_{x,\mu}^{Fi,j}\delta_{y,x+\hat{\mu}} - U_{x-\hat{\mu},\mu}^{F\dagger i,j}\delta_{y,x-\hat{\mu}}) + \Sigma_{\mu=1}^{4}(U_{x,\mu}^{Li,j}\delta_{y,x+3\hat{\mu}} - U_{x-3\hat{\mu},\mu}^{L\dagger i,j}\delta_{y,x-3\hat{\mu}})$$

2. GPU ARCHITECTURE: FERMI GPUS

Early GPUs were fixed function non-programmable graphics only accelerators. Over the years, GPUs have become more powerful in terms of raw compute capability. They have also become more flexible in programmability. NVIDIA introduced a completely programmable shader GPU, and in 2007, Compute Unified Device Architecture (CUDA) was introduced by NVIDIA, with which a programmer can write a general C-like program and execute it in NVIDIA GPUs without using graphics APIs.

GPUs follow a different design philosophy from that of CPUs, that makes them suited for massively parallel data-intensive computing, while not fit for general purpose computing. In GPUs, many compute units (Streaming Processors (SP) in NVIDIA's terminology) are packed together to provide massive raw computing power. For example, there are up to 512 streaming processors in a Fermi GPU and we expect there will be more in the upcoming Kepler GPUs. Each SP is capable of executing one MAD-type single precision floating point instruction per cycle, or two flops. Double precision execution speed is half of that in single precision. With the Fermi architecture, 32 streaming processors are packed together, forming a Streaming Multiprocessor (SM). Each SM contains a scratch pad (called shared memory in NVIDIA's terminology), texture, and dispatching units. Sixteen such streaming multiprocessors form the whole GPU.

There are multiple layers of memory in the GPU. Starting closest to the compute unit (streaming processor), there are registers, shared memory (L1 cache), L2 caches, and global memory. The access to these memories are in order of increasing latency and decreasing bandwidth. The global memory can be bound to texture units and accessed through texture calls. Since texture has its own cache, this usually helps improve application performance. There is also a small amount of constant memory, 64 KB in case of Fermi, that can be used to store constants, lookup tables, etc.

Matching the hardware is the CUDA programming environment. In CUDA programming, the compute or data intensive computation part is rewritten using thousands or tens of thousands of threads, called a kernel. In each kernel, many threads are grouped together and called thread blocks. All the threads in a thread block execute the same code, and run on one SM. Thread blocks should be completely independent, as there is no mechanism for threads in different thread blocks to communicate with each other. Threads within a single thread block can coordinate through the shared memory, and they can be synchronized using *__syncthreads()*. Multiple thread blocks can occupy the same SM and the scheduler can switch to different warps without extra overhead. The ratio of the number of thread blocks to the maximum possible number of thread blocks in a SM, called occupancy, is determined by the amount of resources such as registers and shared memory used by the thread blocks and is usually a good indicator of how well one is using the GPU.

The GPUs in Cray XK6 compute nodes are Tesla X2090s. This Fermi GPU has 512 cores and 6 GB memory with ECC protection. The peak performance is 1.3 Teraflops in single precision and 665 Gflops in double precision. The peak bandwidth is 177 GB/s when ECC is off and 155 GB/s when ECC is on. The GPU is connected to the host with a PCIe 2.0 bus. The hosts are connected to each other with Cray's Gemini Interconnect, which provides 1–2 microseconds of latency in point to point messages and 20 GB/s peak injection bandwidth. The Cray XK6 architecture is shown in Fig. 1 [1].

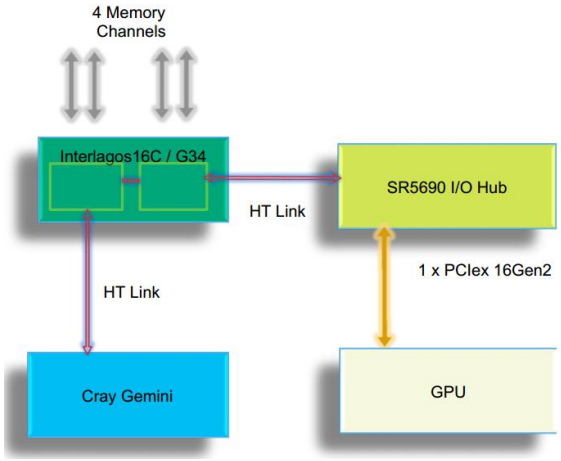

Figure 1: The Cray XK6 node architecture.

It is worth noting that there are two CPU dies in the same socket, each die with its own memory controller. These two dies appear as different numa nodes to the OS. As discussed later, the communication performance will be different when a GPU process is bound to different numa nodes.

3. OVERVIEW OF RELATED WORK

Lattice QCD calculations on GPUs were originally reported in [2] where the immaturity of using GPUs for general purpose computation necessitated the use of graphics APIs. Since the advent of CUDA in 2007, there has been rapid uptake by the LQCD community (see [3] for an overview). More recent work includes [4], which targets the computation of multiple systems of equations with Wilson fermions where the systems of equations are related by a linear shift. Solving such systems is of great utility in implementing the overlap formulation of QCD. This is a problem we target in the staggered-fermion solver below. The work in [5] targets the domain-wall fermion formulation of LQCD. In [6], multi-GPU CG implementation and performance is reported for staggered fermions. This work concerns the QUDA library [7], which we describe in Sec. 4 below.

Most work to date has concerned single-GPU LQCD implementations. Beyond the multi-GPU parallelization of

[1] http://www.olcf.ornl.gov/wp-content/uploads/2012/01/TitanWorkshop2012_Day1_AMD.pdf

QUDA [8, 9] and the work in [10] which targets a multi-GPU implementation of the overlap formulation, there has been little reported in the literature, though we are aware of other implementations which are in production [11].

4. THE QUDA LIBRARY

4.1 General Introduction

QUDA[2] is a library for performing calculations in lattice QCD on graphics processing units (GPUs) using NVIDIA's "C for CUDA" API. It supports multiple quark actions including Wilson, clover-improved Wilson, twisted mass, improved staggered, and domain wall. The initial development was started in Boston University [12] and now it is developed by individuals at multiple institutions. The library supports solvers (both CG and BiCGStab), fatlink, fermion force and gauge force computations. QUDA is a standalone library and can be linked with any existing Lattice QCD software packages. So far MILC and Chroma have been using QUDA to do analysis computations (but not gauge configuration generation).

4.2 Data Structure And Data Layout In GPU Memory

In lattice QCD, we mainly focus on two data structures, the spinor field and the gauge field. Each element of a spinor field is a length three complex vector (6 floating point numbers) and each element of a gauge field is a 3×3 complex matrix (18 floating point numbers). On a four-dimensional space time lattice, there is one spinor element at each of the grid points (called a "site"). The gauge fields live on the links connecting nearest neighbor grid points, and are sometimes just called links. For each site, there are four links that go to its neighbors in $+X, +Y, +Z, +T$ directions, and there are four links pointing to this site from its neighbors from $-X, -Y, -Z, -T$ directions. The spinor and all the four links pointing to the positive direction's neighbors are stored according to the coordinates of this site. All the sites are categorized as even or odd sites according to whether the sum of x, y, z, and t is even or odd. All even data (spinor or gauge field) are stored in the first half of the array and the odd data are stored in the second half. The mapping of 4-D coordinates to 1-D array is the following

$$idx = \frac{t * XYZ + z * XY + y * X + x}{2}$$

where X, Y, and Z are the dimension's sizes, x, y, z, t are the coordinates of a site.

In CPU memory, all 6 floating numbers for a spinor or 18 floating numbers for a gauge field are stored contiguously in memory. This is well suited for a CPU's large cache architecture. However, in a GPU, this will not result in coalesced memory access. In order to enable coalesced access, the spinor or the gauge field are split into 3 float2 (double2) or 9 float2 (double2) numbers, respectively, and stored as depicted in Figs. 2 and 3. In this way, the threads in the same warp are able to read or write data stored consecutively, thus enabling coalesced access to achieve near maximum bandwidth.

In many cases, the 18 numbers in the gauge field are not independent. For example, in a SU(3) matrix, the last row

[2]https://github.com/lattice/quda

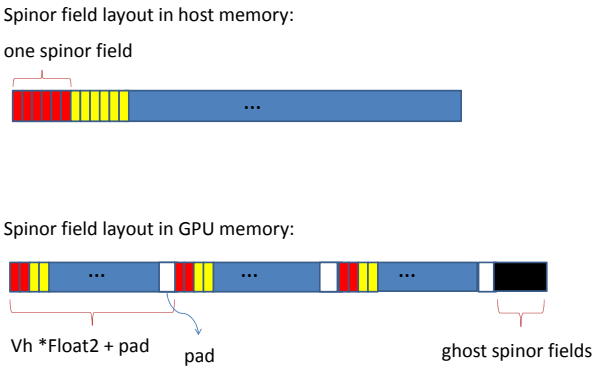

Figure 2: Spinor field layout in host and GPU memory for the staggered discretization (consisting of six floating point numbers per site). Here Vh is half the local volume of the lattice, corresponding to the number of sites in an even/odd subset. Layout for the clover-improved Wilson discretization is similar, wherein the spinor field consists of 24 floating point numbers per site.

can be constructed using the outer product of the first two rows, thus reducing storage requirement to 12 numbers. Furthermore, it was shown in [13] that an SU(3) matrix can be reconstructed using only eight numbers. In these cases, we can store and load only 12 or 8 numbers for a gauge field, then reconstruct the entire 18 number gauge field on the fly. In this way, we can trade extra computations for a reduced bandwidth requirement, potentially improving the performance. For single precision, the gauge field in these two cases are split into three float4 (12 numbers case) or two float4 (8 number case) instead of a number of float2 so that we can achieve better bandwidth. Texture units are used when possible to optimize read speed.

4.3 Parallelization Over Multiple GPUs

In parallelizing over multiple GPUs, the 4-D space time lattice is partitioned and each GPU gets a local subvolume of the 4-D space time lattice. Each GPU needs to communicate with its neighbors in the processor grid to get the 3 slices of three-dimensional data in order to do the dslash operation for the local spinor data. The gauge field data remains constant during the whole solution process, thus it is exchanged only once at the beginning. The ghost gauge field is placed in the ghost/padding regions shown in Fig. 3 and proper indices are computed in the kernel to use the ghost data. The spinor field, however, changes in each iteration of the solution process thus must be exchanged among GPUs each iteration. The ghost data for the spinor is placed at the end of the local data buffer instead of the padding region due to the requirement for computing norms with spinors.

Exchanging spinor data among GPUs involves several steps in each direction: packing, gathering, inter-process communication through MPI or QMP, and scattering. The QMP, or "QCD message-passing" standard, was originally developed to provide a simplified subset of communication primitives

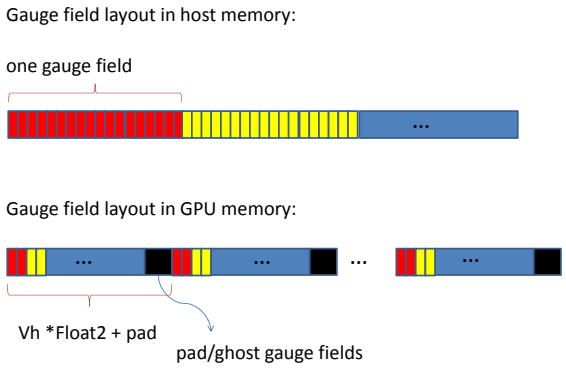

Figure 3: Gauge field layout in host and GPU memory. The gauge field consists of 18 floating point numbers per site (when no reconstruction is employed) and is ordered on the GPU so as to ensure that memory accesses in both interior and boundary-update kernels are coalesced to the extent possible.

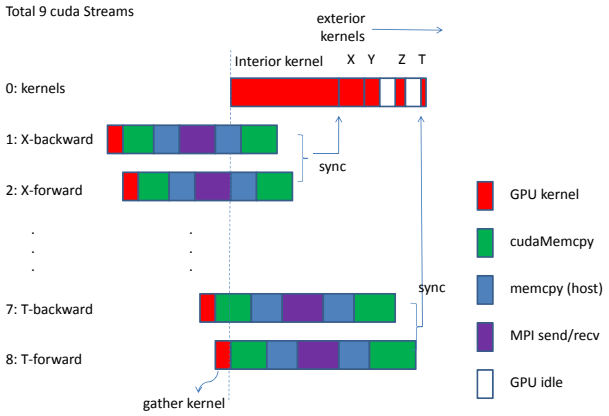

Figure 4: Use of CUDA streams in the application of the Dirac operator, illustrating the multiple stages of communication. A single stream is used for the interior and exterior kernels, and two streams per dimension are used for gather kernels, PCIe data transfer, host memory copies, and inter-node communication.

most used by LQCD codes, allowing for optimized implementations on a variety of architectures, including purpose-built machines that lack MPI. Packing involves launching a GPU kernel to collect the boundary data for that direction into a contiguous GPU buffer. The data is then copied from the GPU to the host memory (gathering); after that, the data is exchanged among the neighboring processes and finally the data is copied to the ghost area of the spinor in the GPU (see Fig. 2). In each direction, these actions must happen in sequence but for different directions, these steps can overlap with each other thus reducing the overall run time. To enable the overlap, one CUDA stream is used for each direction.

The dslash computation in the GPU is separated into an interior kernel and multiple exterior kernels. The updating of interior spinors does not require ghost spinors. Therefore, they can be updated without waiting for the ghost data. Once the ghost data in a direction is collected, an exterior kernel for that direction is launched to finish computing the contribution from the boundary spinors. We use eight CUDA streams for the eight directions ($+X$, $+Y$, $+Z$, $+T$, $-X$, $-Y$, $-Z$, $-T$) and one extra stream for interior and exterior kernels. The overall dslash execution timeline is shown in Fig. 4.

4.4 CG, Mixed Precision and Multi-mass CG Solvers

Following the efficient implementation of the dslash operation in the GPU, we developed Conjugate Gradient (CG) for all precisions: double precision, single precision and half precision, and all the variations: no reconstruct (18 numbers for a gauge field), 12-reconstruct, 8-reconstruct gauge fields. Mixed precision solvers [7] are also supported, where the majority of the work is done in lower precision while the residual is updated using higher precision later so that the final solution is as accurate as the higher precision. We also developed a multi-shift (multi-mass) CG solver based on the algorithm in Ref. [14] and a variation to solve multi-shift systems using a mixed precision solver.

Table 2: Measured peak bandwidth

communication	measured peak (GB/s)
h2d	5.7
d2h	6.5
bidir optimal numa	11.3
bidir suboptimal numa	9.1

5. THE PERFORMANCE RESULTS

5.1 Bandwidth across the PCIe Connection

To better understand the GPU performance, we benchmarked the communication bandwidth over PCIe. The host to device (h2d), device to host (d2h) and bi-directional bandwidth are shown in Fig. 5 with different message sizes. The peak bandwidth is reported in Table 2. The bi-directional bandwidth is measured by sending data from host to device and from device to host at the same time. Since h2d and d2h are the same for different numa settings, we only report one. However, the bi-directional bandwidth shows a difference for large messages. This indicates that there can be a performance penalty when the d2h and h2d communication happen simultaneously if the process is running on the suboptimal numa node.

5.2 The Dslash Timeline And The solver's Performance

Dslash operation is the most time consuming operation in the CG solver, and the performance of a CG solver is determined by how effectively we can execute the dslash operation. Here we use the $U(1)$ code that computes the elec-

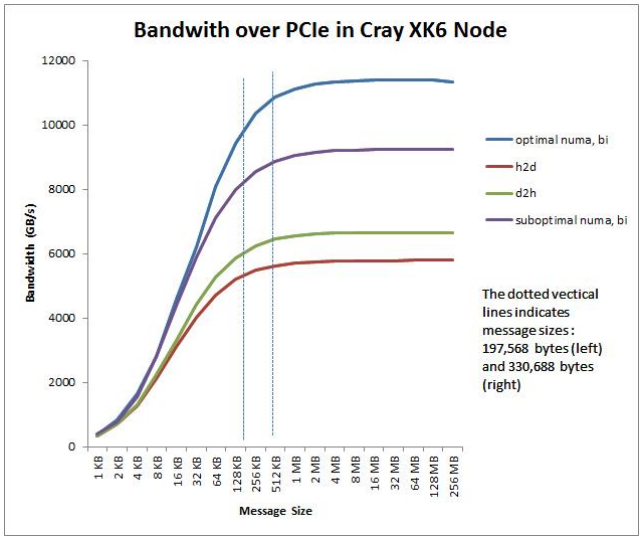

Figure 5: The h2d, d2h and bi-directional bandwidth in a Cray XK6 GPU node

tromagnetic contribution to the meson mass [15]. The test is set up to run on 16 GPUs, with the input lattice size $28^3 \times 96$, 7 masses and different numa settings. The solver is the double/single mixed precision multi-mass solver. The problem is partitioned over Y/Z/T dimensions in the grid of $2 \times 2 \times 4$, with local volume size $28 \times 14 \times 14 \times 24$. In each iteration, the spinor's boundary data is exchanged among its immediate neighbors in six directions $+Y, -Y, +Z, -Z, +T, -T$.

The system we use in this experiment is the Early Science System (ESS) for Blue Waters. The ESS contains 48 Cray XE6 cabinets and a small number of XK6 GPU nodes, representing about 15% of the final Blue Waters system [3]. On the ESS we use CUDA 4.0 for the GPUs and Cray MPICH2 version 5.4.2 for inter-node communication. One typical dslash execution timeline in ESS is shown in Fig. 6, with optimal or suboptimal numa. In both cases, the interior kernel time exceeds the communication time. The asynchronous nature of the communication channel makes the execution in the communication channel efficient since there is no fixed order of execution among different channels. It can be seen in the diagram that all the communication phases, including d2h, inter-node MPI communication and h2d, are well overlapped with each other in different communication channels. While in both cases, the interior kernel runs long enough to cover the communication time, it is clear that with the suboptimal numa setting communication takes longer and is much closer to the interior kernel run time. It is worth noting that while the h2d and d2h performance for different numa nodes are virtually identical, the combined performance is not, and the message sizes in our test cases (197,568 bytes for the T direction, and 338,688 bytes for the Y and Z directions) are large enough to be in the region where the bi-directional PCIe performance degrades with suboptimal numa binding (see Fig. 5). It is shown in the suboptimal numa figure that the gather phase in the $Y-$ channel is starting to overlap with the scatter phase in the $T+$ phase,

[3]More info about ESS can be found in http://www.ncsa.illinois.edu/News/Stories/BW_ESS/

with which the numa effects for bi-directional bandwith will influence the performance. The numa node also has a visible impact on the MPI performance as shown in the orange bar in Figure 6. Overall this explains why the communication channel binding to the suboptimal numa cores shows longer communication time.

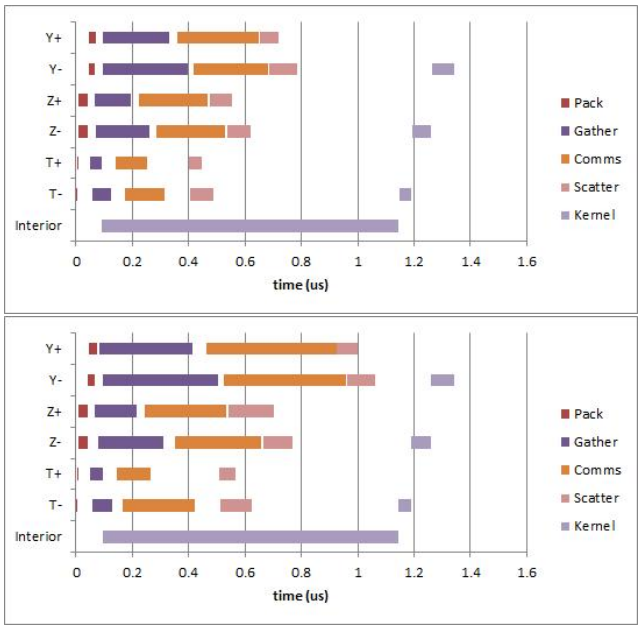

Figure 6: The execution timeline for one dslash operation with different numa settings. Each horizontal represents a CUDA stream handling one directions' communication and/or kernels. In both cases the interior kernel completely overlaps the communications in other directions. The upper diagram represents the case with optimal numa mapping while the bottom one represents the suboptimal numa mapping case.

The achieved bandwidth results are shown in Table 3. One noticeable point is that the bandwidth in the scatter phase is higher than those in the gather phase in general. This is due to the fact the gather for all communication channels happen in a relatively small time window, and they compete with each other for PCIe bandwidth, while the scatter in different communication channels happens at different times, and each one gets nearly full bandwidth. It is not surprising that the first d2h communication (T+, gather) and the last h2d (Y+, scatter), achieve the best bandwidth performance because there is no contention for the first or the last PCI communications.

The final solver performance for both cases in ESS, shown in Table 4, are very close to each other. As mentioned earlier, the interior kernel completely covers the communication channel's execution, thus shielding the internal communication differences. Due to limited availability of GPU nodes, we are not able to test scaling with more than 16 GPUs. However, we expect the solver's performance may degrade significant when the local volume decreases thus reducing the interior kernel run time.

As a comparison, we did similar runs on the Dirac GPU

cluster in NERSC[4]. Dirac is a 50 node GPU cluster connected with QDR infiniband. Each GPU node contains two Intel 5530 2.4 GHz, 5.86 GT/sec QPI Quad core Nehalem processors each with 8 MB of cache (8 cores per node) and 24 GB DDR3-1066 registered ECC memory. Forty four nodes contain one C2050 GPU per node, and the rest contain various numbers of different GPUs. For this experiment, we used 16 C2050 GPU nodes and benchmarked the same problem. We used CUDA 4.1 for GPUs and openMPI 1.4.2 with gcc in Dirac for inter-node communications. The dslash profile is shown in Fig. 7. It can be seen that the interior kernel is only slightly slower in Dirac than in ESS. This makes sense since we have C2050 in Dirac, while we have the full blown Fermi GPU X2090 in ESS. A noticeable difference is that at NERSC the overall communication time is more than the interior kernel, therefore, the GPU is idle for a significant fraction of the time, thus degrading the performance. One interesting point to notice regarding the Dirac cluster is that while the MPI communication for different channels started at different times, they end almost at the same time. We speculate that this might be due to a fair-share policy in the MPI implementation. This is particularly bad for our application because it slows down every channel, and it creates contention for the host to device communication bandwidth, as can been seen in the scatter time in Fig. 7. The solver's reported performance is shown in Table 4. The ESS GPU nodes are about 40% faster than the GPU nodes in Dirac cluster for this particular run. It is worth noting that we did not report the optimal numa or suboptimal numa performance for Dirac, as our measurements show no difference between the different numa settings.

[4]http://www.nersc.gov/users/computational-systems/dirac/

Table 3: The achieved bandwidth in the different communication stages. The message size in T direction is $197,568$ bytes while it is $338,688$ bytes in Y and Z directions.

numa node	Comm	Gather	MPI	Scatter
optimal numa	T-	2.9	2.8	2.3
	T+	4.9	3.6	4.2
	Z-	1.8	2.8	4.1
	Z+	2.6	2.8	4.2
	Y-	1.1	2.6	3.4
	Y+	1.4	2.3	5.0
suboptimal numa	T-	2.8	1.5	1.7
	T+	4.5	3.3	3.4
	Z-	1.5	2.2	3.2
	Z+	2.3	2.4	2.1
	Y-	0.8	1.6	3.4
	Y+	1.0	1.5	4.8

Table 4: The CG performance

Machine	GFLOPS/GPU
ESS with optimal numa	51.1
ESS with suboptimal numa	50.3
Dirac	36.1

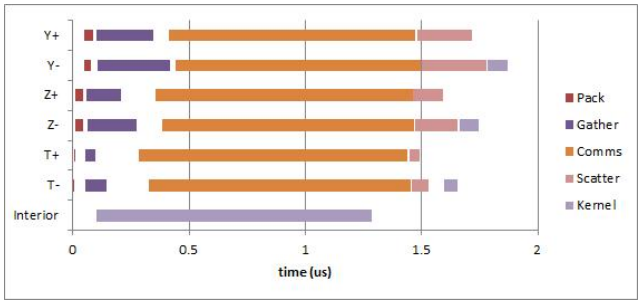

Figure 7: The execution timeline in a typical dslash operation in the Dirac GPU cluster at NERSC.Each horizontal represents a CUDA stream handing one directions' communication and/or kernels. In this run the communication channels run longer time than the interior kernel, indicating the performance will hurt because the GPUs sit idle for some period of time during the iteration

6. CONCLUSIONS AND FUTURE WORK

This paper demonstrates and explains the dslash and the conjugate Gradient solver's performance when running a MILC application on multiple Cray XK6 GPU nodes, parallelizing in multiple dimensions. Our results show how the relative runtime of the communication channels and the interior kernel dictates the final performance. The result also shows that while there is virtually no difference in d2h and h2d for the different numa cores, the combined bandwidth is not the same, and this is reflected in the increased communication time in the dslash operation. While the current runs show little difference with different numa bindings, the performance difference is expected to be more significant when the problem is scaled up to more GPUs and the interior kernel runs in less time than the communication channels. We also compared the performance between ESS and Dirac GPU cluster from NESRC and show how the better GPUs and inter-node communications help the ESS achieve better performance.

Our future work includes two directions: one is to push the scaling so that we can run the solver in thousands of GPUs efficiently with real world work loads; the other direction is to port all the components, including the fatlink, the gauge force and the fermion force computation into GPUs and parallelize them over multiple GPUs so that we can run the entire gauge generation in many GPUs. We are actively pursuing both goals with other members of the QUDA development team.

7. ACKNOWLEDGMENTS

Part of this research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work was partially supported by Department of Energy grants DE-FC02-06ER41443 and DE-FG02-91ER40661. We thank Robert Gottlieb for reading the manuscript.

8. REFERENCES

[1] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower,

and S. Gottlieb, "Scaling Lattice QCD beyond 100 GPUs," in *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. IEEE Computer Society, 2011.

[2] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nógrádi, and K. K. Szabó, "Lattice QCD as a video game," *Computer Physics Communications* **177** no. 8, (2007) 631 – 639, arXiv:0611022 [hep-lat].

[3] M. A. Clark, "QCD on GPUs: cost effective supercomputing," *PoS* **LATTICE2009** (2009) 003.

[4] A. Alexandru, C. Pelissier, B. Gamari, and F. Lee, "Multi-mass solvers for lattice QCD on GPUs," arXiv:1103.5103 [hep-lat].

[5] TWQCD Collaboration, T.-W. Chiu, T.-H. Hsieh, Y.-Y. Mao, and K. Ogawa, "GPU-Based Conjugate Gradient Solver for Lattice QCD with Domain-Wall Fermions," *PoS* **LATTICE2010** (2010) 030, arXiv:1101.0423 [hep-lat].

[6] K. Hyung-Jin and L. Weonjong, "Multi GPU Performance of Conjugate Gradient Algorithm with Staggered Fermions," *PoS* **LATTICE2010** (2010) 028.

[7] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, "Solving Lattice QCD systems of equations using mixed precision solvers on GPUs," *Comput. Phys. Commun.* **181** (2010) 1517–1528, arXiv:0911.3191 [hep-lat].

[8] R. Babich, M. A. Clark, and B. Joó, "Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11. IEEE Computer Society, Washington, DC, USA, 2010. arXiv:1011.0024 [hep-lat].

[9] G. Shi, S. Gottlieb, A. Torok, and V. V. Kindratenko, "Design of MILC lattice QCD application for GPU clusters," in *IPDPS*. IEEE, 2011.

[10] A. Alexandru, M. Lujan, C. Pelissier, B. Gamari, and F. X. Lee, "Efficient implementation of the overlap operator on multi- GPUs," arXiv:1106.4964 [hep-lat].

[11] S. Borsáni, "Thermodynamics from accelerated architectures." http://crunch.ikp.physik.tu-darmstadt.de/gpu2011/Talks/Borsanyi_Darmstadt_GPU.pdf, 2011.

[12] K. Barros, R. Babich, R. Brower, M. A. Clark, and C. Rebbi, "Blasting through lattice calculations using CUDA," *PoS* **LATTICE2008** (2008) 045.

[13] B. Bunk and R. Sommer, "An eight parameter representation of SU(3) matrices and its application for simulating lattice QCD," *Computer Physics Communications* **40** no. 2-3, (June, 1986) 229–232.

[14] B. Jegerlehner, "Multiple mass solvers," *Nuclear Physics B - Proceedings Supplements* **63** no. 1-3, (April, 1998) 958–960.

[15] A. Torok, S. Basak, A. Bazavov, C. Bernard, C. DeTar, E. Freeland, W. Freeman, S. Gottlieb, U. Heller, J. Hetrick, V. Kindratenko, J. Laiho, L.Levkova, M. Oktay, J. Osborn, G. Shi, R. Sugar, D. Toussaint, and R. V. Water, "Electromagnetic splitting of charged and neutral mesons," *PoS* **LATTICE2010** (2010) .