2012 Symposium on Application Accelerators in High Performance Computing

The Fat-Link Computation On Large GPU Clusters for Lattice QCD

Guochun Shi*, Ronald Babich[†], Michael A. Clark[‡], Bálint Joó[§], Steven Gottlieb[¶], Volodymyr Kindratenko*
*National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana, IL 61801, USA
[†]Center for Computational Science, Boston University, Boston, MA 02215, USA
[‡]Harvard-Smithsonian Center for Astrophysics, Cambridge, MA 02143, USA
[§]Thomas Jefferson National Accelerator Facility, Newport News, VA 23606, USA
[¶] Department of Physics, Indiana University, Bloomington, IN 47405, USA

***Abstract*—Graphics Processing Units (GPU) are becoming increasingly popular in high performance computing due to their high performance, high power efficiency and low cost. In this paper, we present results of an effort to implement the fat-link computation — an important component of many lattice quantum chromodynamics (LQCD) calculations — on GPU clusters using the QUDA framework. Two implementations, one similar to the original CPU algorithm in the MILC code and one based on the idea of reduced communication by redundant computations, are presented and their relative advantages are discussed. In strong-scaling tests on up to 384 GPUs on Longhorn and 256 GPUs on Keeneland GPU clusters, where the CPU core to GPU ratio is 4:1 in both clusters, we achieved up to 11.4x and 8.7x node speedup when running on the two GPU clusters, respectively.**

***Keywords*-Quantum Chromodynamics, Lattice QCD, GPU, MILC, QUDA, CUDA**

I. Introduction

The rapidly increasing performance of graphics processing units (GPUs) and the introduction of CUDA that improves the programmability of GPUs for scientific computation has led to a significant effort devoted to efficient numerical computation for lattice quantum chromodynamics (LQCD) using GPUs [1]. LQCD is the method of choice for nonperturbative calculations of the effects of Nature's strong force. Further, such calculations consume a significant fraction of the cycles provided by many supercomputing centers. Finding solutions of the Dirac equation, a large sparse system of linear equations, takes the vast majority of the time in a typical LQCD calculation. For generating new configurations of the gauge fields, this generally takes well over 50% of the time, while for measuring physical quantities on stored configurations, it can take almost the entire time. Efficient implementation of other parts of the code have not been as widely reported, though important components have been implemented on a single GPU.

In this work, we describe multi-GPU implementations of other essential routines for gauge field generation. We consider the asqtad prescription of the Dirac operator discretization, focusing on the challenges of implementing the *fat-link* computation [2].

II. Lattice QCD

Quantum chromodynamics describes the interaction of *quarks* and *gluons*. Quarks are fundamental fermions, that carry a *color* charge (analogous to the electric charge of Quantum Electrodynamics (QED)). The gluons are the force mediators of QCD, analogous to the photon of QED. Because the strength of the interaction is much stronger in QCD that in QED, the perturbation theory that works so well for QED is of limited applicability in QCD, requiring an alternative non-perturbative approach.

In LQCD, the quark and gluon fields are defined on a 4-dimensional periodic lattice. The quarks are defined on the grid points and the gluons are defined on the links joining grid points. This procedure regulates the theory as there are no longer an infinite number of dynamical variables. The quarks have a color index and in some formulations a spin index. The variables that describe the quark at each site may be referred to as a *color-spinor*, or just a *spinor* for short. We refer to the complete lattice vector as a spinor field. The gluon or gauge fields consist of $SU(3)$ matrices that reside on the links between all nearest neighbor lattice sites. Thus, each such matrix is referred to as a link matrix. At each site, we store four links that point to the nearest neighbors in the positive x, y, z, and t directions, respectively, as shown in Figure 1. Each site's links in the negative directions are the conjugate transpose of the links that point to the site from corresponding neighbors in the negative directions.

A. The Asqtad Dirac Matrix

The fundamental interactions of QCD, those taking place between quarks and gluons, are encoded in the quark-gluon interaction differential operator known as the Dirac operator. A proper discretization of the Dirac operator for lattice QCD requires special care. As in many PDE solvers, the derivatives are replaced by finite differences. Thus, on the lattice, the Dirac operator becomes a large sparse matrix M and the calculations require many solutions to systems of linear equations given by

$$Mx = b. \tag{1}$$

Computationally, the brunt of the numerical work in LQCD for both the gauge generation and analysis phases involves

CFP1225P-ART/12 $26.00 © 2012 IEEE
DOI 10.1109/SAAHPC.2012.10

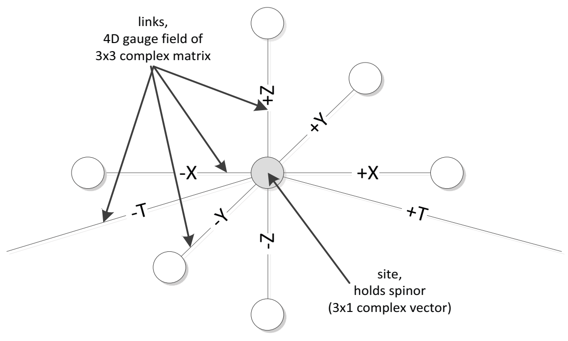


Figure 1. 4-d spacetime lattice.

solving such linear systems.

The staggered matrix is a central-difference discretization of the Dirac operator, wherein the infamous fermion doublers (which arise due to the red-black instability of the central difference approximation) are removed through "staggering" the spin degrees of freedom onto neighboring lattice sites. This essentially reduces the number of spin degrees of freedom per site from four to one, reducing the computational burden significantly.[1] This transformation, however, comes at the expense of increased discretization errors and breaks the so-called taste flavor symmetry. To reduce these discretization errors, the gauge field that connects nearest neighboring sites on the lattice is *smeared*, which essentially is a local averaging of the field. There are many prescriptions for this averaging; and here we employ the popular asqtad procedure. The errors are further reduced through the inclusion of neighboring spinors three links away from the central point in the derivative approximation. When acting in a complex-valued vector space that is the tensor product of a 4-dimensional discretized Euclidean spacetime and *color* space, the asqtad matrix is given by

$$\begin{aligned} M_{x,x'} &= -\frac{1}{2}\sum_{\mu=1}^{4}\big(\hat{U}_x^{\mu}\,\delta_{x+\hat{\mu},x'} - \hat{U}_{x-\hat{\mu}}^{\mu\dagger}\,\delta_{x-\hat{\mu},x'} + \\ &\quad \check{U}_x^{\mu}\,\delta_{x+3\hat{\mu},x'} - \check{U}_{x-3\hat{\mu}}^{\mu\dagger}\,\delta_{x-3\hat{\mu},x'}\big) + m\delta_{x,x'} \\ &\equiv -\frac{1}{2}D_{x,x'} + m\delta_{x,x'}. \end{aligned} \tag{2}$$

Here indices x and x' are spacetime indices (the color indices have been suppressed for brevity), $\delta_{x,y}$ is the Kronecker delta and m is the quark mass parameter. There are two gauge fields present: $\hat{U}_x^{\mu}$ is the *fat* gauge field produced from locally averaging U_x^{μ}, and $\check{U}_x^{\mu}$ is the *long* gauge field produced by taking the product of the links $U_x^{\mu}U_{x+\hat{\mu}}^{\mu}U_{x+2\hat{\mu}}^{\mu}$. While both of these fields are functions of the original field U_x^{μ}, in practice these fields are pre-calculated before the application of $M_{x,x'}$ since iterative solvers will require the application of $M_{x,x'}$ many hundreds or thousands of times. The Kogut-Susskind sign factors and any factors necessary for anti-periodic boundary conditions in time have been absorbed into the matrices $\hat{U}$ and $\check{U}$.

[1]For the staggered discretization we consider here, although there are no spin degrees of freedom, we still refer to the fermion variables as spinors.

B. LQCD calculations

The LQCD computation can be divided into two phases. In the first phase, the gauge configuration generation phase, many ensembles of gauge fields are generated using Monte Carlo simulation, the algorithm of choice being variants of either Hybrid Molecular Dynamics (HMD) or Hybrid Monte Carlo (HMC). In the second phase, the analysis phase, these ensembles are analyzed and the physical quantities of interest are computed. In both stages, the solution of a large system of linear equations is the dominant computation; this so-called Dirac solver is typically a conjugate gradient (CG) or other Krylov solver.

In Table I, we give the distribution of the time spent in each stage of the computation for gauge field generation on a production job run several years ago. Since the Dirac solver accounts for 58.5% of the time, any GPU accelerated LQCD computation must consider more than just the solver to avoid an Amdahl's-law slowdown from the remaining computational routines.

Computation	time(s)	percentage
Dirac solver	2987	58.5
Fermion force	1125	22.0
Gauge force	489	9.5
Fat link	442	8.7
Long link	24	< 1
IO	41	< 1
Total	5108	98.1
Unaccounted	104	1.9
Wall clock	5212	

Table I
TIME DISTRIBUTION FOR ASQTAD GAUGE GENERATION RUN ON 2048 CORES OF A CRAY XT3 (BIGBEN) (VOLUME = $40^3 \times 96$, $m_l = 0.1m_s$).

III. RELATED WORK

The first efforts in employing GPUs for Lattice QCD were reported in [3] where graphics APIs were used. Since the first release of CUDA in 2007, many further efforts in porting Lattice QCD solvers to GPUs have been reported [4], [5], [6], [7]. In [8], [9], a complete RHMC algorithm for staggered fermions was ported to NVIDIA GPUs, and over two orders of magnitude speedup was reported comparing one GPU to one CPU core. However, the algorithm only runs on a single GPU.

Our GPU coding efforts are centered around an open-source library called QUDA [10], where we developed a mixed precision solver [11], [12] and later parallelized it by partitioning the grid over space and time dimensions and ran on large number of GPUs [13], [14], [15]. We use the same framework to port the fat-link computation onto GPUs[16]. To the authors' best knowledge, this is the first effort to port the fat-link computation so that it can use a large number of GPUs with partitioning over space and time dimensions.

IV. Fat-Link Algorithm

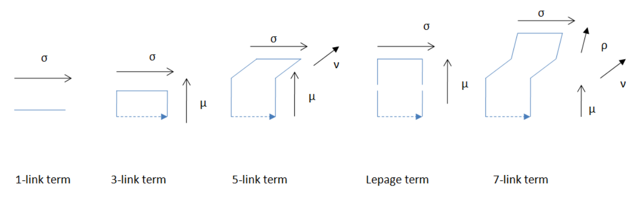


Figure 2. The various terms that contribute to the fat link in the σ direction. σ, μ, ν and ρ represent different spacetime directions.

The fat link can be computed by summing over the contributions from various terms with different weights, namely the 1-link, 3-link, 5-link, Lepage and 7-link terms, as shown in Fig. 2, where σ, μ, ν, and ρ are different directions. Each term is computed by multiplying the links along the path, and accumulated over all possible combinations of directions. For example, when we are computing the X direction fat link ($\sigma = X$), the μ direction for the 3-link term includes $+Y$, $-Y$, $+Z$, $-Z$, $+T$ and $-T$. Notice that a higher link term includes some redundant computation of lower link terms. For example, the middle three links in the 5-link term form a 3-link term and the 5-link term can be realized in two 3-term computation steps, as shown in Figure 3. The CPU version of the implementation, used in MILC [17], takes this into account and avoids the redundant computation by repeatedly calling a 3-link term computing function and accumulating the results into a temporary data structure called a *staple*, which is also a 3×3 complex matrix.

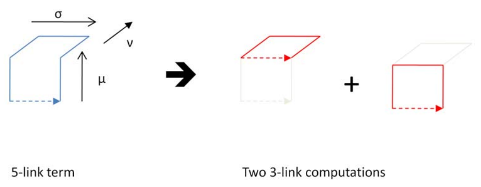
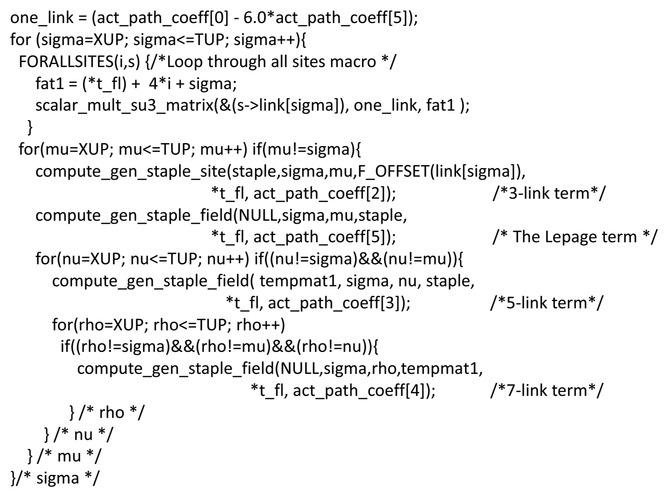


Figure 3. The 5-link term decomposed as two 3-link computations.

The pseudocode for the algorithm is given in Figure 4. In function *compute_gen_staple_site* the 3-link term is computed in the μ for loop and accumulated into the fat link with



```
one_link = (act_path_coeff[0] - 6.0*act_path_coeff[5]);
for (sigma=XUP; sigma<=TUP; sigma++){
  FORALLSITES(i,s) {/*Loop through all sites macro */
    fat1 = (*t_fl) + 4*i + sigma;
    scalar_mult_su3_matrix(&(s->link[sigma]), one_link, fat1 );
  }
  for(mu=XUP; mu<=TUP; mu++) if(mu!=sigma){
    compute_gen_staple_site(staple,sigma,mu,F_OFFSET(link[sigma]),
                                          *t_fl, act_path_coeff[2]);              /*3-link term*/
    compute_gen_staple_field(NULL,sigma,mu,staple,
                                          *t_fl, act_path_coeff[5]);              /* The Lepage term */
    for(nu=XUP; nu<=TUP; nu++) if((nu!=sigma)&&(nu!=mu)){
      compute_gen_staple_field( tempmat1, sigma, nu, staple,
                                          *t_fl, act_path_coeff[3]);              /*5-link term*/
      for(rho=XUP; rho<=TUP; rho++)
        if((rho!=sigma)&&(rho!=mu)&&(rho!=nu)){
          compute_gen_staple_field(NULL,sigma,rho,tempmat1,
                                          *t_fl, act_path_coeff[4]);              /*7-link term*/
        } /* rho */
      } /* nu */
    } /* mu */
}/* sigma */
```

Figure 4. Pseudocode for the fat-link computation.

a coefficient. At the same time, the accumulated 3-link term sum is written into the staple, which is used for the Lepage term and 5-link terms in the ν for loop calling a similar 3-link term computing function *compute_gen_staple_field*. The 7-link term is computed similarly in the ρ for loop. The 3-link term computing functions, *compute_gen_staple_site* and *compute_gen_staple_field*, given two positive directions computes upper and lower 3-link path contribution for the fat link, as shown in Figure 5, with the difference that the first one takes all the links from the site link, while the other takes the middle link from the previously computed staple.

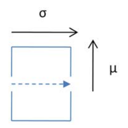


Figure 5. The 3-link computing functions, *compute_gen_staple_site* and *compute_gen_staple_field* compute the contributions from both the upper and lower 3-link paths. The former uses the site link for the σ-direction and the latter uses the staple.

In CPU code, the 4-d links are mapped to a 1-d array in memory. To efficiently use the CPU cache, the links are mapped to even and odd data segments with the even data in the first half of the memory region and the odd data in the second half. The oddness of a site is determined by the oddness of $(x+y+z+t)$, where x, y, z, t are the coordinates for the site. The index of a site within the even or odd region is determined by the coordinates in the following formula

$$\text{index} = \frac{1}{2}(x + y * X + z * X * Y + t * X * Y * Z) \quad (3)$$

where x, y, z, t are coordinates and X, Y, Z are the grid dimensions. All of the site links, fat links and the staples have a similar data structure as described above, with the only difference being that there are four site links or fat links per site, corresponding to four directions while there is only one staple per site, corresponding to the direction that we are computing.

V. Data Layout and Single-GPU Implementation

While computing the fat link, the site link will be read many times and the fat link itself will be read and written once for each kernel. To efficiently run the computation on the GPU, the data layout in the GPU memory must enable coalesced access to the device memory, both in reading and in writing. However, this objective cannot be achieved with the CPU data layout, assuming each site is assigned to a thread, since each thread needs to load its own matrix of 18 words. Coalesced access can be enabled, however, by splitting the 18 words for each link into small chunks and arranging them in such a way that in each load all threads in a warp are loading a contiguous memory segment. Shown in Figure 6 is an example for the single-precision case, where the 18 single-precision floats are split into 9 float2 values that are stored ($\frac{1}{2}V + pad$) apart, where $V = XYZT$. This ensures that for each load instruction in the GPU kernel, shown in Figure 7, all threads in a warp will end up loading 32 float2 data items from a contiguous memory region. The pad is added to make the stride an uneven number to avoid the partition-camping problem that affects certain GPUs [18], first proposed in [11]. As we will discuss in the next section, the padding memory region is also a natural fit for storing ghost-face data from neighboring nodes.

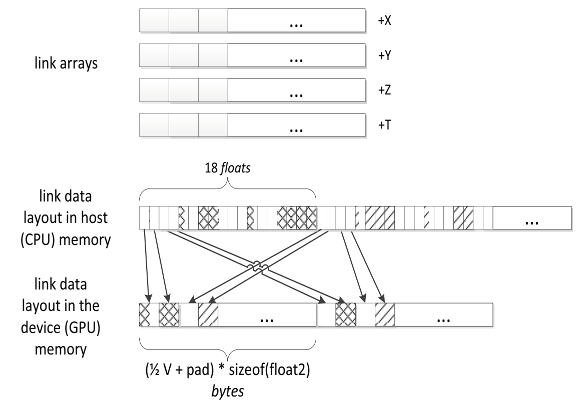
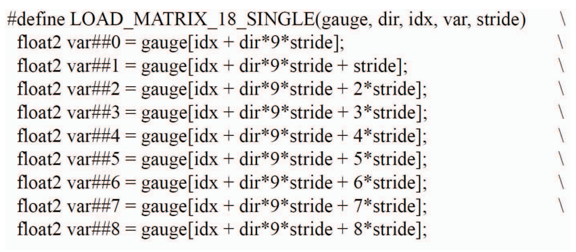

link arrays
+X
+Y
+Z
+T
18 *floats*
link data layout in host (CPU) memory
link data layout in the device (GPU) memory
(½ V + pad) * sizeof(float2) *bytes*

Figure 6. Data layout in GPU device memory.

Since we expect the fat-link computation to be limited by the device memory bandwidth, any transformations to reduce the memory traffic are desirable. Given that the links are special unitary matrices, $U_x^\mu \in SU(3)$, we can reconstruct



```
#define LOAD_MATRIX_18_SINGLE(gauge, dir, idx, var, stride)      \
  float2 var##0 = gauge[idx + dir*9*stride];                     \
  float2 var##1 = gauge[idx + dir*9*stride + stride];            \
  float2 var##2 = gauge[idx + dir*9*stride + 2*stride];          \
  float2 var##3 = gauge[idx + dir*9*stride + 3*stride];          \
  float2 var##4 = gauge[idx + dir*9*stride + 4*stride];          \
  float2 var##5 = gauge[idx + dir*9*stride + 5*stride];          \
  float2 var##6 = gauge[idx + dir*9*stride + 6*stride];          \
  float2 var##7 = gauge[idx + dir*9*stride + 7*stride];          \
  float2 var##8 = gauge[idx + dir*9*stride + 8*stride];
```

Figure 7. Link-reading code used in GPU kernels; `idx` is the thread index and corresponds to the space-time index, `dir` is the direction the link points and `stride` is the stride in device memory between each value read in.

the third row from the cross product of the first two rows, thereby reducing memory loads from 18 numbers to 12 at the expense of additional computation. It is further shown in [19] that links can be reconstructed from 8 independent numbers. We have implemented the "12-reconstruct" and the "no-reconstruct" (or "18-reconstruct") cases. The 12-reconstruct method has a positive impact on performance in single precision, but for double precision this is less clear, and depends on the double-precision capability of the GPU under consideration, as will be discussed in the performance section. The GPU implementation of the fat-link computation maintains the same high level looping structure, shown in Figure 4, with the two functions *compute_gen_staple_site* and *compute_gen_staple_field* implemented as GPU kernels. The thread block size is set to 64 and the total number of threads is equal to the total number of sites, where each thread computes the upper and lower 3-link path's contribution to the fat link. The site link and fat link are read and written in a coalesced way, and the link-link (matrix-matrix) multiplication is fully unrolled. Some constants, such as lattice dimensions, are stored in constant memory. Since shared memory is not explicitly used, on NVIDIA's Fermi architecture the 64 KiB L1/shared memory is set to 48 KiB/16 KiB, preferring L1 cache. Some shared memory is used to store links, thus reducing register usage.

VI. Multi-GPU Implementation

A. Approach 1: Standard Implementation

Our initial approach to implementation is to do exactly what the fat-link computation in MILC does; i.e., the high-level loop code structure remains the same, and the neighboring site links and temporary staples are exchanged between neighboring nodes in order to compute the upper and lower 3-link contributions to the fat link in the *compute_gen_staple_site* and *compute_gen_staple_field* functions. Since only the nearest neighbors are used in the two functions, we only need to exchange the first and last 3-d faces between neighboring nodes in each direction where the lattice is partitioned. The site link does not change throughout the whole fat-link computation, and therefore the ghost-face exchange for site links can be done only once at the beginning of the computation. The temporary staple, however, is updated many times within the loops and needs to be exchanged every time it is recomputed. For the site link, 2-d face ghost data from a diagonal neighbor is also needed, due to the fact that the third link needed for the lower staple can be from the diagonal neighbor in $(+1, -1)$ for (σ, μ) directions, shown in Figure 5. It can be determined that a total of 12 (4×3 because σ and μ are different directions) diagonal neighbors' ghost data are needed. The staple does not need the diagonal neighbors' ghost data because it is always the middle link in the the 3 link computation when it is used. As discussed earlier in the data layout section, the padding memory region, added to the link and staple data structure to avoid partition camping, can also be used to store the ghost-face data from the neighbors.

The site link boundary data exchange happens only once at the beginning of the computation, when the site-link data is in the host memory. Each node communicates to its 8 direct neighbors one 3-d face worth of ghost site link data and to 12 diagonal neighbors one 2-d face worth of data each. Since the data in the X, Y, Z dimensions are not contiguous (unlike data in the T dimension), the 3-d or 2-d data are first copied to a contiguous block of memory before they can be communicated. In receiving all the ghost-face data from its neighbors, each node copies the ghost data to the end of the local data and moves the whole combined array to GPU memory. It then launches a GPU kernel to do the desired format converting, as will be discussed in a later section. The site link data remains in the GPU memory throughout the entire computation.

The boundary data exchange for the staple is similar to the site link, except the source is from the GPU instead of the CPU memory and it happens only among 8 direct neighbors. A gather GPU kernel is used to gather the boundary data for X, Y and Z directions (while T direction data is already contiguous in GPU memory). Once the data is gathered, it is copied to CPU memory and exchanged among neighbors using MPI. Upon receiving the ghost-staple data, it is copied to the GPU memory and put in the padding/ghost region. With the advent of GPU Direct support in CUDA 4.0, copies between Infiniband and CUDA pinned memory spaces are not necessary in the process.

An opportunity to overlap communication and computation exists since the dependence is only on the boundary data. To overlap communication and computation, the staple computations can be separated into an interior kernel, which computes the interior staples and 8 exterior kernels which compute the boundary staples. The corner sites among multiple boundaries are computed multiple times. Note that all the ghost staples needed for this staple computation has already been received from the previous staple computation, thus we can start the exterior kernels, which use the ghost staples, at the very beginning of the staple computation. For the very first ghost staple computation, all the data are from local site links and ghost site links, which are both available. The exterior kernels are launched first, followed by the gather kernels, then the communications and the interior kernel simultaneously. A total of 9 streams are used, as shown in the Figure 8.

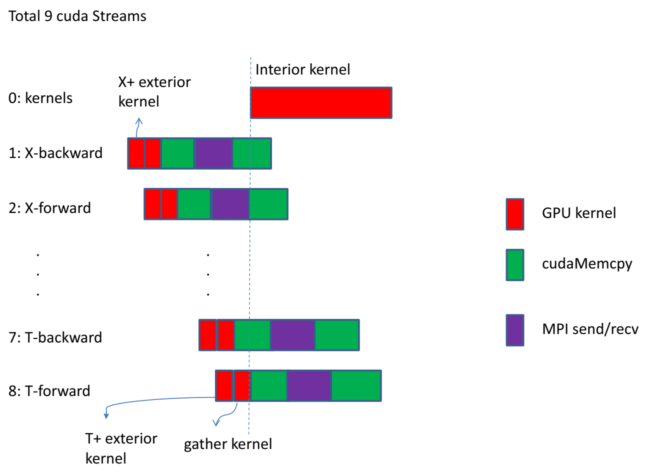


Figure 8. Overlapping communication with computation in the standard approach to computing fat links.

Finally, the 3-link term GPU kernels are modified accordingly to read the ghost-face data for its neighbor site link and staple accesses.

B. Approach 2: Extended-Volume Implementation

Some time ago, one of us noticed that the performance of the CPU version of the gauge-force computation was limited because of the large number of gathers required. The solution, implemented by S. Basak, was to pregather all the off-node links needed in the entire computation so that all subsequent references to links stored at neighboring sites (which might reside off-node) could be satisfied by a local copy of the data. That resulted in greatly improved performance and helped to inspire the approach taken here.

It is clear from Figure 2 that we can compute all contributions to the fat link without exchanging the staple, if we have all the ghost site links at the beginning and compute the otherwise ghost staple locally. Due to the presence of the Lepage term, two 3-d ghost faces are needed in both positive and negative directions in X, Y, Z, and T. In the rest of this section X, Y, Z, and T, will be used to denote a direction or the extent of the grid *local* to each GPU. The local volume thus increases from $XYZT$ to $(X+4)(Y+4)(Z+4)(T+4)$, thus increasing the GPU memory requirement.

1) Ghost site link exchange: All the ghost data for the extended volume reside on up to 80 ($3^4 - 1 = 80$) neighboring nodes. Communicating with all these neighbors separately could be a huge overhead. We complete the communication phase by sending/receiving up to 8 messages in total. First all nodes exchange with their neighbors in the $-X$ and $+X$ directions, with the $x = 0, 1$ and $x = X - 2, X - 1$ 3-d faces, respectively. Each 3-d face is of size YZT. Next, each node communicates with its neighbors in the $-Y$ and $+Y$ directions, with $y = 0, 1$ and $y = Y - 2, Y - 1$ 3-d faces. However, the 3-d face size is $(X + 4)ZT$, with the local data filling the $2 \leq x < X + 2$ range in the message, and ghost data from X direction communication filling the range $x = 0, 1$ and $x = X + 2, X + 3$. Once the Y communication is done, each node communicates with its Z neighbors, with the Z 3-d face size $(X + 4)(Y + 4)T$. The local data fills the range $2 \leq x < X + 2$, $2 \leq y < Y + 2$ and the previous ghost data from X and Y directions fill the rest. Finally each node communicates with its $-T$ and $+T$ directions' neighbors with message size $(X+4)(Y+4)(Z+4)$. In this way we construct a complete $(X+4)(Y+4)(Z+4)(T+4)$ extended volume containing all local data and ghost data from neighbors with only 8 messages. When a certain dimension is not partitioned, the data exchange does not happen although the node still needs to copy local data into the extended volume as if they are from neighbors.

2) Data format and GPU kernels with extended volume: In the extended-volume approach, all the local and ghost site links from neighbor nodes are copied into an extended data region. The site-link data is then mapped to GPU memory using a strategy similar to that illustrated in Figure 6, only with a $+4$ size increase in each dimension. The staple data structure is similarly constructed with a $+4$ increased size. The fat link data structure retains its original dimensions since there is no ghost fat link involved. The high level code structure remains the same as in Figure 4.

There are two types of staple computation calls. The first computes the staple, adds the contribution of the fat link, and saves the staple for future use. For this type of function call, we need to invoke the kernel with total number of threads equal to $(X+2)(Y+2)(Z+2)(T+2)$. Each thread is mapped to a set of coordinates (x, y, z, t), where $0 \leq x < X + 2$, $0 \leq y < X+2$, $0 \leq z < Z+2$, and $0 \leq t < T+2$. Threads on the boundaries, i.e. $x = 0, X + 1$ or $y = 0, Y + 1$ or $z = 0, Z + 1$ or $t = 0, T + 1$, compute the staples and save them while the inner threads compute a staple, add it to the fat link with the appropriate weight and save the staples as well. In computing the index of a given site link or staple from the coordinates, the sizes of the various dimensions are first increased by 4. The second type of call only computes the contribution to fat link; no staple storage is needed. In this case, only $XYZT$ threads are launched and mapped to coordinates in original dimension sizes. The first and the third staple calls in Figure 4 belong to the first type and the second and the fourth staple calls are the second type. In the extended-volume approach, the index calculations for the neighboring sites in the path are straightforward since they will stay in the extended volume and no special boundary treatment is necessary. All the staple and fat-link contribution computations are local and efficient, as will be shown in the performance section.

VII. Interface and Link-Format Change Kernels

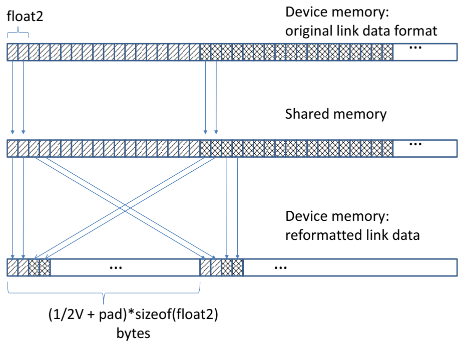


Figure 9. The GPU kernel to change the link data from CPU format to the desired GPU format.

The GPU computation of the fat link is interfaced with MILC through the library *libquda.a*, where the function *load_fatlinks* from MILC is replaced by the GPU function *load_fatlinks_gpu*. The site link data, originally scattered in different memory locations in MILC, are gathered into a contiguous memory region, and the format of the site-link data is transformed from the contiguous 18-number data format to the strided format consisting of 9 float2 numbers (for example). The format change was originally done on the CPU before sending the data to the GPU, but we found the overhead of doing so to be significant. To reduce the overhead, the CPU format data is copied to the device memory as-is, and then a GPU kernel performs the format conversion. Figure 9 shows an example of how the link data is shuffled efficiently to the desired format in the GPU kernel. The source link data is loaded into the shared memory as a coalesced read, where all threads cooperate to read data in a contiguous memory region. In the write-out stage, each thread is responsible for writing out a single link, e.g., 9 float2 numbers, and the write to the device memory is coalesced as well, although the read access in shared memory at this stage is striped. Similarly, after the computation is done, the fat link needs to change from the GPU format to the CPU format, and a similar kernel is used to do the format conversion on the GPU before the fat link is copied back to host memory.

VIII. Performance and Discussion

A. Hardware description

For this study, we use two GPU clusters: Longhorn at the Texas Advanced Computing Center and the Keeneland Initial Delivery (ID) system at the National Institute for Computational Sciences. The Longhorn cluster has 256 compute nodes, each with 8 Intel Nehalem cores (2.5 GHz) and 2 NVIDIA Quadro FX5800 cards. The Quadro FX5800 is a high-end professional graphics part based on the GT200 architecture, with similar compute capability to the Tesla C1060 (intended for compute-only applications). The Keeneland ID system consists of 120 nodes, each with dual hex-core Intel Westmere CPUs and 3 NVIDIA Tesla M2070 cards, based on the GF100 ("Fermi") architecture. Both clusters utilize QDR Infiniband as the interconnect. In this study, we run scaling tests from 32 to 384 GPUs on Longhorn and 64 to 256 GPUs on Keeneland. We also perform single-GPU runs on GeForce GTX480 cards installed in the AC cluster at NCSA.

B. *Single-GPU performance With artificially enabled communication*

We benchmarked the fat-link computation performance on a single GTX480, in both single and double precision, with or without 12-reconstruct, with a lattice volume of $16^3 \times 32$. Although it is a single-GPU run, we can induce the communications from the node to itself by partitioning the input lattice in any given dimensions, thus executing the multi-GPU code path on a single GPU. The performance results of partitioning different dimensions are shown in Figure 10. When there is no communication, the standard approach performs better than the extended-volume approach. This is expected because when there is no partitioning, the standard approach does not incur any communication overhead nor does it involve any redundant computation. The second approach, however, still needs to copy the local volume data to fill in the $+4$ extended volume. It also incurs additional kernel cost because of the additional ghost staple computation. When we enable the partition in the T direction, the performance for both approaches drops but in the standard approach it drops much faster although it is still better than the performance in the extended-volume approach. Eventually when the ZT, YZT, and $XYZT$ directions are partitioned, the performance for the extended-volume approach is better than the standard approach, indicating that the increased communication cost outweighs the cost of additional computations. We also observe that in both approaches the 12-reconstruct method is slightly faster than no-reconstruct method, with a larger performance gain in single precision. The tradeoff of increased flop count for lower bandwidth requirements is more beneficial for single precision than double precision due to the abundant single precision compute throughput available on the GTX 480 in comparison to double precision.

C. *Multi-GPU performance*

For our strong-scaling runs, we use version 7.6.3 of the MILC code running the *su3_rmd* application from the *ks_imp_dyn* directory with a lattice volume of $64^3 \times 192$.

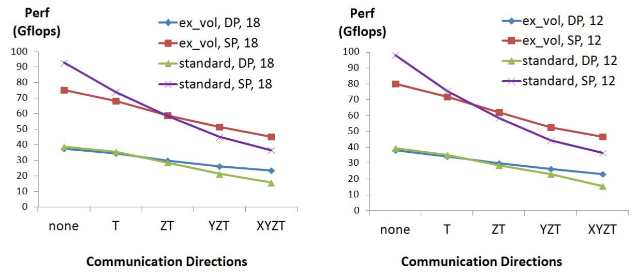


Figure 10. Single GTX480 GPU performance with communication enabled in various dimensions. The left diagram shows the single-precision (SP) and double-precision (DP) performance for the standard and extended-volume ("ex_vol") approaches, and the right diagram shows the same set of measurements with 12-reconstruct. The horizontal axis labels indicate in which dimensions communication is enabled. The lattice volume is $16^3 \times 32$.

We always report the performance for the second fat-link computation since the first one involves high CUDA and QUDA library initialization overhead. Performance for various combinations of options are measured, including

- Two implementations: one CPU-like implementation, tagged as *standard*. and the second extended-volume approach, tagged as *ex_vol*.
- Two precisions: both single (tagged as *SP*) and double precision (tagged as *DP*).
- Two link-reconstruction methods: 12-reconstruct (tagged as 12) and no-reconstruct (tagged as 18).
- Various choices of partitioned dimensions: the 4-d lattice can be partitioned in one or more dimensions. We choose to partition either in all *X, Y, Z, T* (denoted as *XYZT*), or in *Y, Z, T*(denoted as *YZT*), or in *Z, T*(denoted as *ZT*). We do not try partitioning the problem in T only because it won't work well due to the small local time dimension.
- Various numbers of GPUs: we benchmark from 32 to 384 GPUs on Longhorn and from 64 to 256 GPUs on Keeneland.

1) Strong-Scaling Runs on Longhorn: The strong-scaling performance is shown in Figure 11. It can be clearly seen that the extended-volume approach exhibits better performance than the standard approach in all cases, indicating the benefits of reducing communication by performing redundant computation on the GPUs. It is also observed that in the double-precision case, the 12-reconstruct method is usually slower than no-reconstruct, while for single precision, it is the other way around. The extended-volume approach also exhibits better performance scaling than the standard approach. Three data points at 32 GPUs are missing for the double-precision extended-volume approach, because those runs require more than 4 GB of memory per GPU, and thus fail. This reflects the increased memory requirement

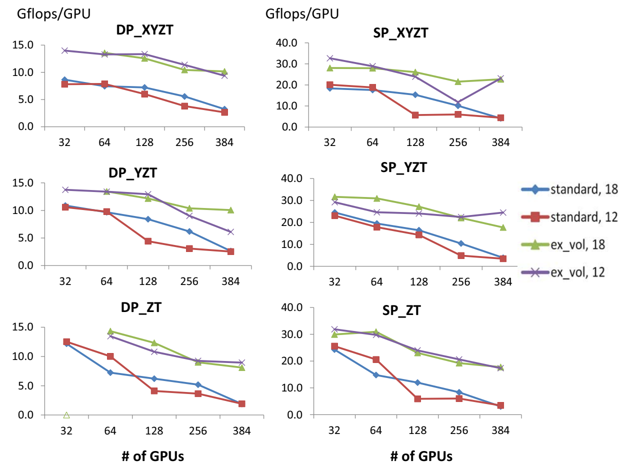


Figure 11. Fat-link strong-scaling performance on Longhorn. The diagrams on the left are for double precision, while those on the right are for single precision. The upper, middle and lower rows of diagrams show performance results for partitioning in dimensions *XYZT*, *YZT* and *ZT*, respectively. Within each diagram, we report performance for the standard and extended-volume approach, as well as the 12-reconstruct and no (18) reconstruct cases.

due to the +4 size increase in all dimensions. However, when using the fat-link computation together with the GPU Dirac solver [15], we expect the solver would reach the GPU memory bottleneck before the fat-link computation does even for the extended-volume algorithm.

# of nodes	16	32	64	128	192
# of CPU cores	128	256	512	1024	1536
DP perf/core(Gflops)	0.75	0.76	0.56	0.34	0.34
SP perf/core(Gflops)	1.4	1.1	1.0	0.56	0.54

Table II
THE CPU STRONG-SCALING PERFORMANCE ON LONGHORN.

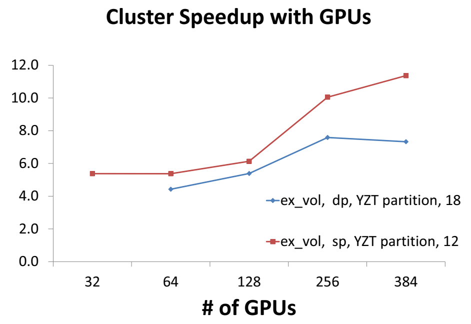


Figure 12. The best cluster speedup on Longhorn as a function of number of GPUs. The GPU performance is obtained by running two GPUs and two CPU cores per node, while the CPU performance number is obtained by running eight CPU cores per node.

We also benchmarked the same lattice with the same number of nodes, using all the CPU cores but no GPUs. For these runs, we use the original MILC 7.6.3 program and partition the lattice in all dimensions, the same way MILC usually does in its production runs. The performance using 16 to 192 nodes (corresponding to 32 to 384 GPUs) is shown in Table II. It can be seen that the performance per CPU also decreases significantly as the number of nodes increases. The speedup from using the GPUs actually increases as the number of nodes increases (see Figure 12), due to the worse scaling of the CPU performance. The best cluster speedup is 8.3x for double precision and 11.4x for single precision, when running the extended-volume implementation with the *YZT* partitioning scheme.

2) *Strong-Scaling Runs on Keeneland:* On Keeneland, we ran the fat-link computation with 64, 128 and 256 GPUs. When running on 32 GPUs, we encountered some MPI-related errors and were unable to resolve the issue in time for this publication. Since there are 3 GPUs per node, we use the minimum number of nodes required, i.e., 22, 43 and 86 nodes, and leave the extra one or two GPUs idle on the last node. The performance is shown in Figure 13.

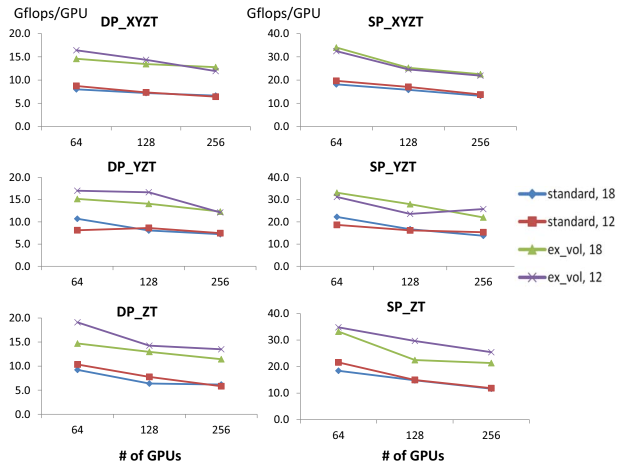


Figure 13. Fat-link strong-scaling performance on Keeneland. The diagrams on the left are for double precision, while those on the right are for single precision. The upper, middle and lower rows of diagrams show performance results for partitioning in dimensions *XYZT*, *YZT* and *ZT*, respectively. Within each diagram, we report performance for the standard and extended-volume approach, as well as 12-reconstruct and no (18) reconstruct cases

It is again demonstrated that the extended-volume approach shows superior performance in all cases and the scaling efficiency is similar for both implementations. In most cases, the 12-reconstruct variant shows better performance than the corresponding no-reconstruct (18) case. This is in contrast with Longhorn, where the 12-reconstruct is slower in most cases with double precision. We attribute this difference to the much improved double-precision peak performance of the Tesla M2070 ("Fermi") card. We also notice that per-GPU performance on Keeneland is higher than on

Longhorn. For example, the best performance for 64 GPU runs with the extended-volume approach is 14.3 Gflops/GPU for double precision and 30.9 Gflops/GPU for single precision on Longhorn, while they are 19.1 Gflops/GPU and 34.8 Gflops/GPU on Keeneland, representing an increase of 34% and 13% for double and single precision, respectively.

We also run the CPU benchmark on the same number of nodes on Keeneland. Since there are 12 cores per node and the input lattice is not divisible by the total number of cores, we ran 256 out of 264, 512 out of 516 and 1024 out of 1032 cores, respectively. As shown in Table III, at the 256-core level, the Keeneland performance is similar to the Longhorn performance, but on 512 and 1024 cores, the Keeneland performance scales much better. Because of the good CPU performance scaling, the best node speedup with GPUs gradually declines, as shown in Figure 14. The best cluster speedup is 8.7x for double precision and 7.7x for single precision, when running the extended-volume implementation with the *ZT* partitioning scheme and 12-reconstruct.

# of nodes	22	43	86
# of CPU cores used	256	512	1024
DP perf/core(Gflops)	0.74	0.73	0.68
SP perf/core(Gflops)	1.13	1.11	1.06

Table III
CPU STRONG-SCALING PERFORMANCE ON KEENELAND.

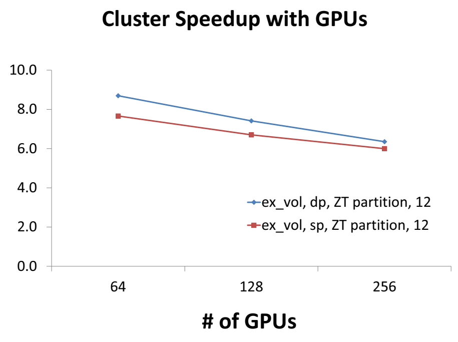


Figure 14. The best cluster speedup on Keeneland when using GPUs. The GPU performance is obtained by running three GPUs and three CPU cores per node, while the CPU performance number is obtained by running twelve CPU cores per node.

3) Kernel performance and overhead: It is worth noting that the current performance that the MILC code reports includes all the communication overheads, including the communication with all neighbors, the copying of site-link data from host to GPU memory at the beginning and the copying of the computed fat-link data to the host memory. In the future, when we move other components such as gauge force and fermion force computations onto the GPUs, it will be possible to keep all the data in GPU memory, and thus the fat-link copy from the GPU memory to host memory can be eliminated. While the communication of site-link data with neighbors will still be necessary, the cost can be amortized with the gauge force and fermion force computation. As shown in Table IV, a large percentage of the time is the overhead, and the kernel-only performance is two to three times faster than than the current performance MILC sees, indicating a great potential for performance improvement when other components are moved onto GPUs in the future.

Machine	Precision	Perf w/ overhead (Gflops)	Kernel Perf only (Gflops)
Longhorn	DP	13.4	28.9
	SP	30.9	79.0
Keeneland	DP	15.2	43.8
	SP	33.1	93.8

Table IV
PERFORMANCE OF THE FAT-LINK COMPUTATION WITH AND WITHOUT COMMUNICATION OVERHEAD FOR 64-GPU RUNS WITH THE EXTENDED-VOLUME APPROACH AND *YZT* PARTITIONING SCHEME.

IX. CONCLUSIONS AND FUTURE WORK

In this work, we ported the asqtad fat-link computation, needed by the MILC code, to GPU clusters using the QUDA framework. We showed that the algorithm that removes redundant computation and runs efficiently on conventional systems without GPUs does not work well on GPU-enabled clusters. We demonstrated the new extended-volume approach, which reduces communication by performing redundant computations and shows superior performance and better scaling. With the extended-volume approach, we achieved significant speedups compared to the CPU-only runs: up to 11x on Longhorn and 8x on Keeneland, where the CPU core to GPU ratio is 4:1 for both clusters. We believe the performance can be further improved in the future as more components are ported to GPUs.

There is a great motivation to port all the key code components necessary for gauge field generation to run in multi-GPU mode. These components are the linear solver, the fat-link computation, the gauge force and the fermion force computations. We presented the work for the solver in [15] and for the fat-link computation in this paper. The logical next step will be porting the gauge and fermion forces to GPU clusters. We already have written single-GPU ports for these two forces, and the performance is very encouraging. We are also extending the code to use a two-level link fattening scheme called HISQ [20], [21], which will be used extensively in production running by MILC.

ACKNOWLEDGMENTS

We gratefully acknowledge the use of the Longhorn cluster at the Texas Advanced Computing Center and the Keeneland ID cluster at the National Institute for Computational Sciences. The authors would like to thank Justin

Foley and Carleton DeTar of the University of Utah for helpful discussions. This work was supported in part by NSF grants OCI-0946441, OCI-1060012, OCI-1060067, and PHY-0555234, as well as DOE grants DE-FC02-06ER41439, DE-FC02-06ER41440, DE-FC02-06ER41443, DE-FG02-91ER40661, and DE-FG02-91ER40676. BJ additionally acknowledges support under DOE grant DE-AC05-06OR23177, under which Jefferson Science Associates LLC manages and operates Jefferson Lab. GS is funded through the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign. The U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce this manuscript for U.S. Government purposes.

References

[1] M. A. Clark, "QCD on GPUs: cost effective supercomputing," *PoS*, vol. LATTICE2009, p. 003, 2009.

[2] A. Bazavov, D. Toussaint, C. Bernard, J. Laiho, C. DeTar *et al.*, "Nonperturbative QCD simulations with 2+1 flavors of improved staggered quarks," *Rev.Mod.Phys.*, vol. 82, pp. 1349–1417, 2010.

[3] 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*, vol. 177, no. 8, pp. 631 – 639, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/B6TJ5-4P0084K-2/2/9acb2259093f6609d741c654a00e666e

[4] Y. Osaki and K.-I. Ishikawa, "Domain Decomposition method on GPU cluster," *PoS*, vol. LATTICE2010, p. 036, 2010.

[5] H.-J. Kim and W. Lee, "Multi GPU Performance of Conjugate Gradient Algorithm with Staggered Fermions," *PoS*, vol. LATTICE2010, p. 028, 2010.

[6] 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*, vol. LATTICE2010, p. 030, 2010.

[7] A. Alexandru, C. Pelissier, B. Gamari, and F. Lee, "Multi-mass solvers for lattice QCD on GPUs," 2011.

[8] C. Bonati, G. Cossu, M. D'Elia, and A. Di Giacomo, "Staggered fermions simulations on GPUs," *PoS*, vol. LATTICE2010, p. 324, 2010.

[9] C. Bonati, G. Cossu, M. D'Elia, and P. Incardona, "Staggered fermions simulations on GPUs," *preprint number KEK-CP-255, IFUP-TH/2011-14*, 2011.

[10] http://lattice.github.com/quda, 2011.

[11] 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.*, vol. 181, pp. 1517–1528, 2010.

[12] S. Gottlieb, G. Shi, A. Torok, and V. Kindratenko, "QUDA programming for staggered quarks," *PoS*, vol. LATTICE2010, p. 026, 2010.

[13] 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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.40

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

[15] 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*, ser. SC '11. IEEE Computer Society, 2011.

[16] G. Shi, S. Gottlieb, and V. V. Kindratenko, "MILC on GPUs," *NCSA Technical Report*, 2010.

[17] M. L. Collaboration, C. Bernard *et al.*, "The MILC Code," http://www.physics.indiana.edu/~sg/milc/, 2010.

[18] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in CUDA," *NVIDIA Technical Report*, 2009.

[19] B. Bunk and R. Sommer, "An eight parameter representation of SU(3) matrices and its application for simulating lattice QCD," *Comput. Phys. Commun.*, vol. 40, pp. 229–232, 1986.

[20] E. Follana *et al.*, "Highly improved staggered quarks on the lattice, with applications to charm physics," *Phys.Rev.*, vol. D75, p. 054502, 2007.

[21] A. Bazavov *et al.*, "Scaling studies of QCD with the dynamical HISQ action," *Phys.Rev.*, vol. D82, p. 074501, 2010.