MILC on GPUs

Gushing Shi, Steven Gottlieb, Volodymyr Kindratenko

1 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 applications can be accelerated on the Cell Broadband Engine [3]. In this work, we investigate how this code can take advantage of the newly emerging GPU computing paradigm.

There are four main parts of the code that are responsible for over 98% of the overall execution time [2]: 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].

We investigated three different approaches to MILC GPU implementation: 1) PGI compiler based port of the code to NVIDIA GPUs, 2) porting kernels "as is" using CUDA C, and 3) rewriting essential computational QCD algorithms in CUDA C and interfacing them with the MILC code base. In this section we briefly outline the findings from these three approaches and in the sections that follow we provide an in-depth implementation description for the third approach, which turned out to be the most fruitful.

1.1 PGI GPU compiler-based approach

The PGI Compiler is designed to take sequential C or FORTRAN code and automatically convert it to code which can then be sent to the GPU, in an effort to reduce the need for explicit memory management in GPU code. We tried to implement mult_su3_na subroutine in fermion_force_fn_multi calculations using this approach.

First, we ran into numerous compiler bugs since the PGI compiler still was in beta at the time. Second, we discovered numerous compiler limitations, such as no support for structures, so the code was rewritten using pointers in place of structures, which are partially supported. The code was further re-written to use a large array instead of pointers or structs, as the compiler needed everything to be in contiguous memory to send it to the GPU automatically. The relevant MILC macros were also replaced with explicit code. The function call was replaced by the body of the function that was called, effectively inlining mult_su3_na. Also, the two inner loops were completely unrolled. With these very significant code transformations, the compiler was able to generate a GPU implementation of the innermost loop. However, the increased overhead appears to have made the code run slightly slower. Overall, the PGI

compiler still requires a fair amount of rewriting of the code and still seems to require some amount of explicit memory management.

1.2 Direct kernel port

The initial plan was to offload the entirety of the loop starting at line 241 in fermion_force_fn_multi.c to the GPU. However, the student was not able to get this working. We are now not sure this approach was a good idea as it does not particularly increase the FLOPS/bytes ratio and involves some significant possibility of branching. More on this work can be found in [4].

More recently we tried to write parallel SU(3) operations (scalar multiply-add, multiply, adjoint, projector) with the end goal of replacing many of the FOREVEN, FORODD, and FORALL inner loops. We have not gotten far enough in this approach to have results, but it seems that (short of a total rewrite) this is a reasonable strategy for accelerating MILC, although with limitations on the possible performance improvements.

1.3 Complete rewrite of essential computational algorithms

We have implemented Conjugate Gradient (CG) solver and Fat Link computation kernel from scratch and interfaced these implementations with the MILC code base. In the following sections we present these implementations and performance results. Our starting point for this work was the Boston University implementation of Wilson-Dirac sparse matrix-vector product and CG solver [6]. More on the CG solver implementation can be found in [5].

2 Conjugate Gradient Implementation

2.1 Staggered Dslash implementation

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

$$D_{j,y;i,x} = \sum_{\mu=1}^{4} \left(U_{j,i,x,\mu}\, \delta_{x,y+\hat{\mu}} - U^{\dagger}_{j,i,x,\mu}\, \delta_{y,x+\hat{\mu}} \right)$$

MILC uses staggered Dslash operator whereas Boston University provided Wilson-Dirac Dslash operator. Thus, the rest of this report deals with the staggered Dslash operator GPU implementation.

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 (Figure 1). The opposite link for a site is

defined as 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. Reference CPU implementation of the staggered Dslash operator is shown in Figure 2.

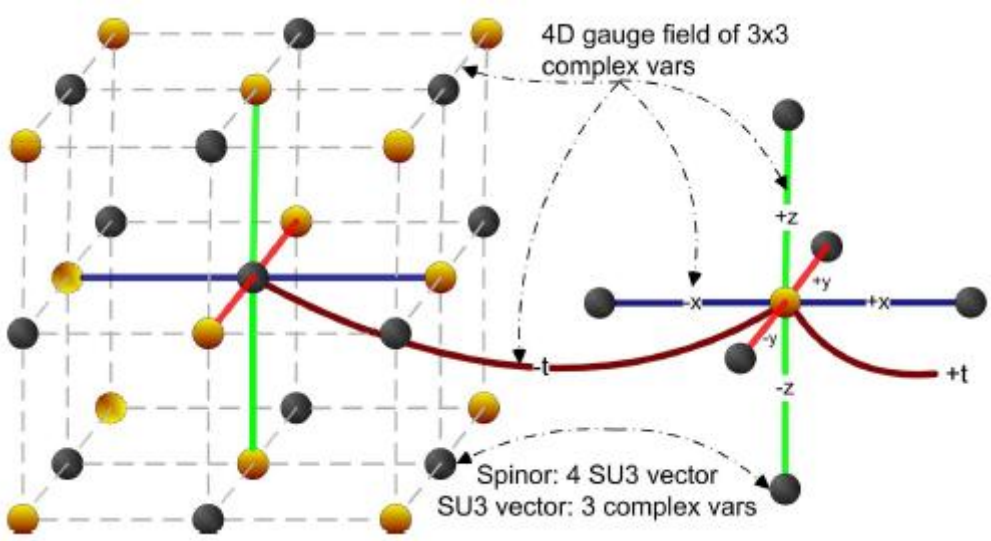


Figure 1. Four-dimensional lattice QCD. Picture is taken from *K. Ibrahim, F. Bodin, Efficient SIMDization and Data Management of the Lattice QCD Computation on the Cell Broadband Engine*.

```
template <typename sFloat, typename gFloat>
void dslashReference_st(sFloat *res, gFloat **fatlink, gFloat** longlink, sFloat *spinorField, int oddBit, int daggerBit)
{
    for (int i=0; i<Vh*1*3*2; i++) res[i] = 0.0;

    gFloat *fatlinkEven[4], *fatlinkOdd[4];
    gFloat *longlinkEven[4], *longlinkOdd[4];

    for (int dir = 0; dir < 4; dir++) {
        fatlinkEven[dir] = fatlink[dir];
        fatlinkOdd[dir] = fatlink[dir] + Vh*gaugeSiteSize;
        longlinkEven[dir] =longlink[dir];
        longlinkOdd[dir] = longlink[dir] + Vh*gaugeSiteSize;
    }
    for (int i = 0; i < Vh; i++) {
        for (int dir = 0; dir < 8; dir++) {
            gFloat* fatlnk = gaugeLink_st(i, dir, oddBit, fatlinkEven, fatlinkOdd, 1);
            gFloat* longlnk = gaugeLink_st(i, dir, oddBit, longlinkEven, longlinkOdd, 3);

            sFloat *first_neighbor_spinor = spinorNeighbor(i, dir, oddBit, spinorField, 1);
            sFloat *third_neighbor_spinor = spinorNeighbor(i, dir, oddBit, spinorField, 3);

            sFloat gaugedSpinor[spinorSiteSize];

            if (dir % 2 == 0){
                su3Mul(gaugedSpinor, fatlnk, first_neighbor_spinor);
                sum(&res[i*spinorSiteSize], &res[i*spinorSiteSize], gaugedSpinor, spinorSiteSize);
                su3Mul(gaugedSpinor, longlnk, third_neighbor_spinor);
                sum(&res[i*spinorSiteSize], &res[i*spinorSiteSize], gaugedSpinor, spinorSiteSize);
            }
            else{
                su3Adjmul(gaugedSpinor, fatlnk, first_neighbor_spinor);
                sub(&res[i*spinorSiteSize], &res[i*spinorSiteSize], gaugedSpinor, spinorSiteSize);
                su3Adjmul(gaugedSpinor, longlnk, third_neighbor_spinor);
                sub(&res[i*spinorSiteSize], &res[i*spinorSiteSize], gaugedSpinor, spinorSiteSize);
            }
        }
    }
}
```

Figure 2. Reference CPU implementation of the Dslash operator.

Each site contains one spinor, 4 fat links and 4 long links. Each spinor is 1x3 complex vector, requiring 6 bytes for single-precision implementation. Each link, fat link or long link, is a 3x3 complex matrix, thus requiring 72 bytes of storage for single-precision implementation. The long link is unitary, and thus can be reconstructed using 12-reconstruct or 8-reconstruct [7]; the fat link does not have this property. 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 is shown in Figure 3.

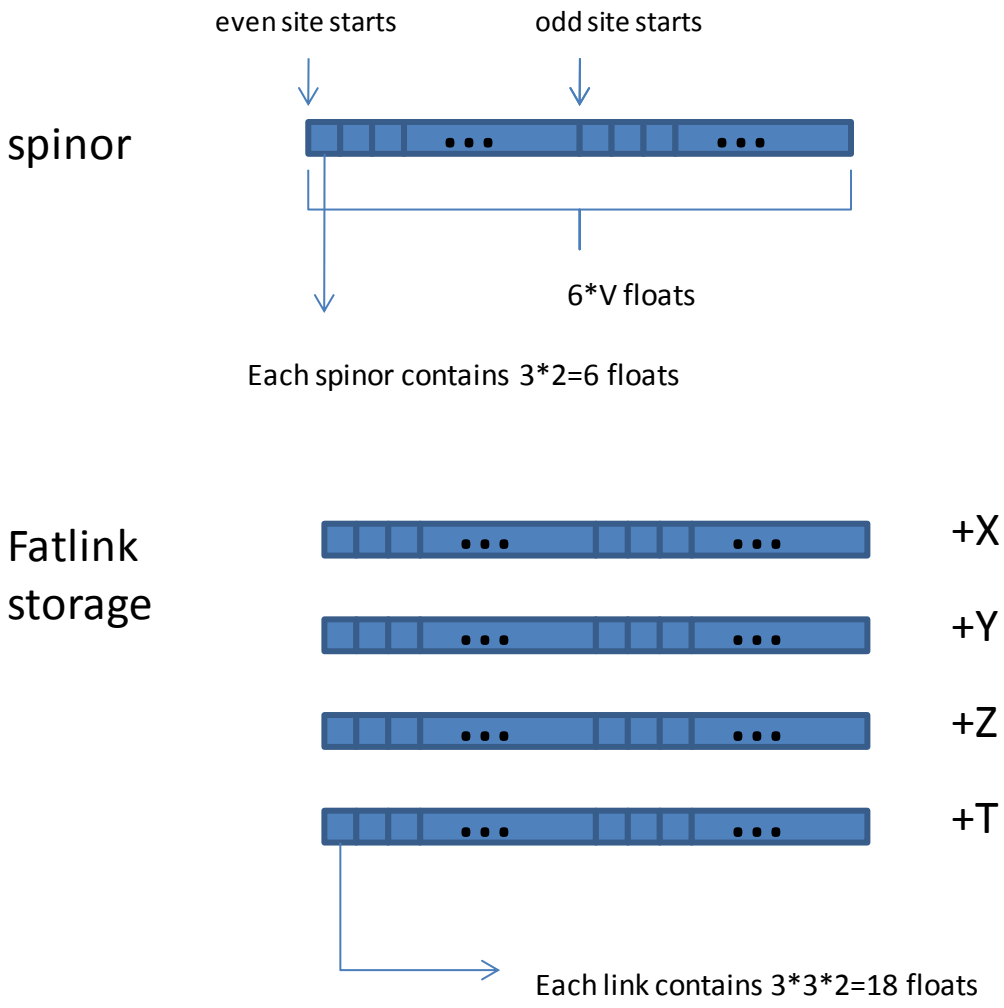
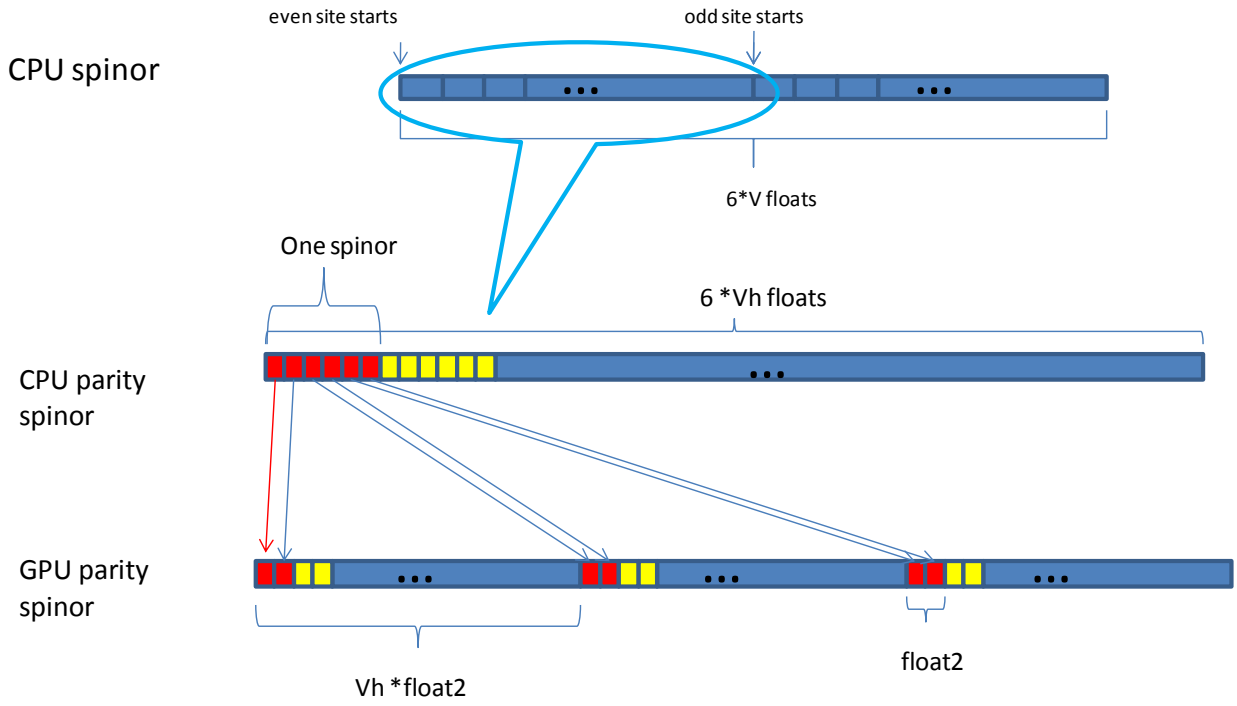


Figure 3. Data layout in memory.

When implementing the Dslash operation on the GPU, it is important to note that the computation/byte ratio is close to 1. 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. We achieve this by aligning data in 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 stripe of Vh so that when all threads are reading neighboring spinors in the same odd/even segment, their reads can be coalesced (Figure 4). The data is read through texture, which results in better performance as compared to the device memory.



GPU kernel code to read spinor

```
#define READ_SPINOR_SINGLE(spinor)                  \
  float2 I0 = tex1Dfetch((spinor), sp_idx + 0*Vh);  \
  float2 I1 = tex1Dfetch((spinor), sp_idx + 1*Vh);  \
  float2 I2 = tex1Dfetch((spinor), sp_idx + 2*Vh);
```

Figure 4. Data layout in the device memory for spinors and GPU code to access the data via texture unit.

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 accessing 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 5 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 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.

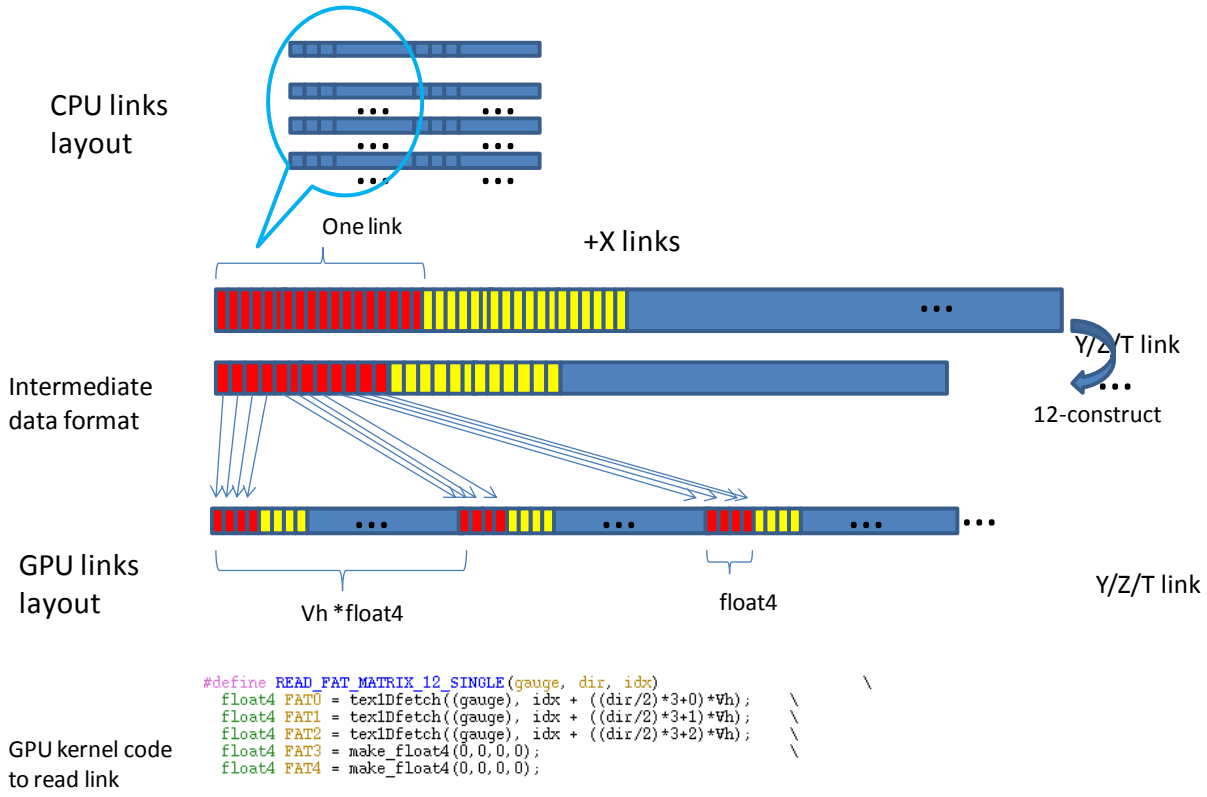


Figure 5. Data layout for the 12-reconstruct for the long link matrix.

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. Such an implementation is nontrivial and is a subject of further work.

Here we only described 12-reconstruct and single-precision computations. Implementations of the 8-reconstruct and double-precision computations are similar. The half precision is implemented as well using short2/short4 data types with normalization vector. More details on these mixed-precision implementations can be found in [5].

2.2 CG

The Conjugate Gradient procedure 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 achieved. The Dslash as well as 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 the overall performance is substantially improved.

2.3 Multi-mass Solver

Occasionally 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 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.

2.4 Interface to MILC

The CG and multi-mass solvers are implemented as a stand-alone library. To interface them with MILC program, 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.

2.5 Dslash and CG performance

Here we report results obtained on GTX280 NVIDIA GPU using lattice size $24 \times 24 \times 24 \times 32$. As shown in Table 1, with decreasing precision the performance increases. For single precision, changing from 12-reconstruct to 8-reconstruct, the bandwidth requirement decreases hence the performance increases. However, in case of double precision, the peak performance for GTX280 is only 77 GFLOPS and 8-reconstruct introduced extra computations, hence the effective performance decreases although the bandwidth requirement decreases.

	Double	Single	Half
12-reconstruct	29.7 GFLOPS	86.7 GFLOPS	136.7 GFLOPS
8-reconstruct	16.7 GFLOPS	95.8 GFLOPS	127.9 GFLOPS

Table 1. Dslash operation performance.

Table 2 shows performance from CG solver for different precision and reconstruct techniques. We achieve as high as 86 GFLOPS for single precision and nearly 30 GFLOPS for double precision. In mixed precision mode, we can achieve 107 GFLOPS for double precision accuracy and 118 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 increases.

The multi-mass solver performance is close to the corresponding CG solver performance. As we discussed before, the mixed precision does not work for multi-mass solver.

Spinor	Link	Reconstruct	Spinor sloppy	Link sloppy	Recon sloppy	Performance (GFLOPS)
double	double	12	double	double	12	28.7
double	double	12	single	single	8	82.3
double	double	12	half	half	12	107.5
single	single	8	single	single	8	86.1
single	single	8	half	half	12	118.7
half	half	12	half	half	12	120.0

Table 2. CG solver performance.

3 Fat Link computation

3.1 GPU Implementation

There is one fat link in each direction for each site. The fat links are computed from the unitary links in one site's neighbors. It involves several terms: 1-link term, 3-link term, 5-link term, lapage term, and 7-link term. The 1-link term is trivial to compute. Given two different directions, the 3-link is computed as the sum of upper staple and lower staple as shown in Figure 6. The staple is computed as the matrix product in the route.

Once 3-link term is calculated, the 5-link term is computed in the same way the 3-link term is computed, with the second matrix in the route being the previously computed 3-link term. The lapage term can be computed similar to the 5-link term. Similarly, the 7-link term can be computed based on the previously computed 5-link term. This way, redundant computations are removed. This method is implemented in the original MILC code base.

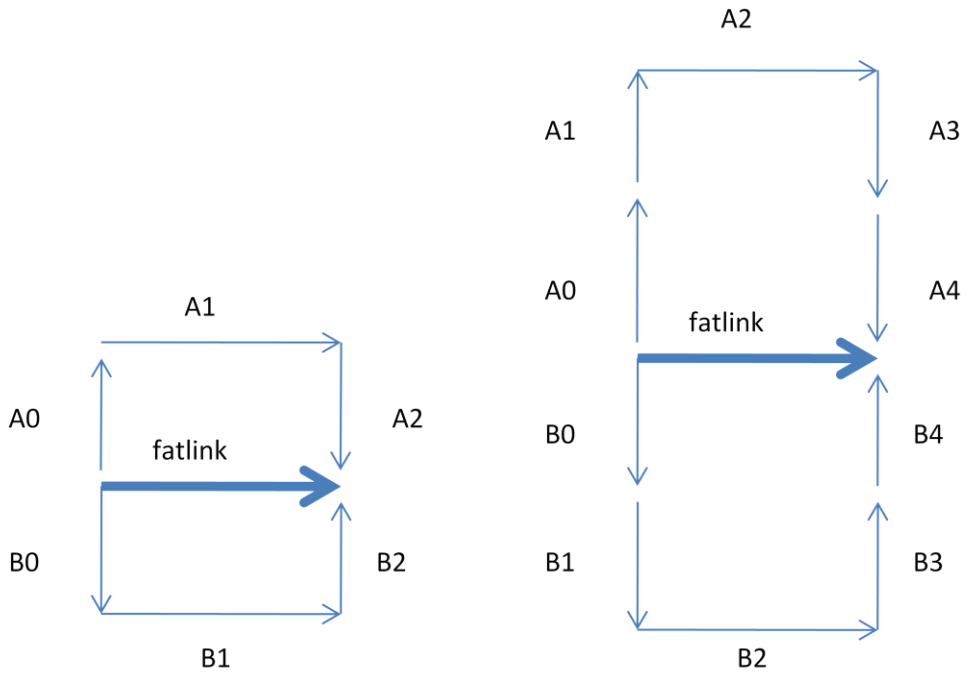


Figure 6. Computation of 3- (left) and 5-link (right) terms.

It is easy to see that these primitive links are reused heavily in computing different fat links. A careful count shows that each link is reused 78 times in computing all the fat links. In porting the code to GPU, we attempted to store links data in shared memory for sharing between multiple threads. Links can then be loaded once and can be reused by many threads. However, to compute a fat link, we need to load four neighbor sites in each direction $(+2, -2)$, which indicates we will need load all links from 5^4 neighbors for a single thread. This requires 120 KB shared memory for a single thread even when using single-precision and 12-reconstruct which is well above the shared memory size (64KB per SM) on current GPU architecture. Therefore, we currently do not use shared memory in computing fat links.

The site links are represented by unitary matrixes and therefore we can use the 12-reconstruct to reduce the bandwidth usage. We use float4 data type to store site links. Both staple and fat links are not unitary and we use nine float2 values to store them. The values in each link are stored with stride Vh so that we ensure coalesced device memory access.

Although we could implement all terms (except the trivial 1-term) computation in one kernel, we wrote several similar kernels in order to avoid some branch instructions. The kernels are further separated to even and odd kernels to compute even and odd sites separately. Register usage in these kernels is a big problem as our initial implementation required 115 registers, resulting on the GPU with occupancy of only 13%. We were able to reduce the he register usage to 85 via a careful manual tuning, improving the GPU occupancy to 19%. Further improvements that resulted in 64 registers (corresponding to 25% occupancy) have not yielded better performance. Mixed use of global and shared memory and reading through texture helped to improve the performance slightly.

3.2 GPU performance

Based on preliminary analysis, assuming 12-reconstruct for each link, we should achieve 168 GFLOPS if we can sustain 100 GB/s memory bandwidth. For fat link computation, we only implemented single precision and 12-reconstruct. Performance for different versions is shown in table 3. The measurements were obtained running the code on GTX280 GPU. Our latest version performs close to the 100 GB/s projected memory bandwidth resulting in 168 GFLOPS. 8-reconsuct and double precision implementations are to be done later.

	First revision	Second revision	Third revision
Optimization techniques used		Reduced register use, increased occupancy	Mixed memory: reading from device memory directly and reading through texture
Performance	96 GFLOPS	125 GFLOPS	156 GFLOPS

Table 3. Fat link kernel performance.

4 Conclusions and future work

So far we have only implemented two out of four major computational algorithms used in MILC. Performance achieved for these two kernels varies significantly depending on the data type used, 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 86.1 GFLOPS on the GPU and just under 3 GFLOPS on the 2.8 GHz Intel Xeon core, or about ~29x speedup. Similarly, Fat link kernel performance is 156 GFLOPS on the GPU vs. 2.9 GFLOPS on the CPU, or ~53x speedup.

The remaining two algorithms to be ported to the GPU are Fermion and Gauge force.

Also, 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 a good overall performance. This is still a subject of further analysis and implementation.

5 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.

6 Appendix

6.1 Conjugate Gradient GPU Implementation

Subroutines to compute CG and Dslash. dslashSS12Kernel is one of several GPU kernels.

```
/*****************Conjugate Graident ************************
 * The main loop on CPU
 * The dslash and other blas operations are done on GPU
 */

int invertCgCuda_milc_parity(ParitySpinor x, ParitySpinor source, FullGauge fatlinkPrecise,
                                        FullGauge longlinkPrecise, FullGauge fatlinkSloppy,
                                        FullGauge longlinkSloppy, ParitySpinor tmp,
                                        QudaInvertParam *perf, double mass, int oddBit)
{
  ......
```

```
 PRINTF("%d iterations, r2 = %e\n", k, r2);
  stopwatchStart();
  while (r2 > stop && k<perf->maxiter) {
            struct timeval t0, t1;
            gettimeofday(&t0, NULL);
            dslashCuda_st(tmp_sloppy, fatlinkSloppy, longlinkSloppy, p, 1 - oddBit, 0);
            dslashAxpyCuda(Ap, fatlinkSloppy, longlinkSloppy, tmp_sloppy, oddBit, 0, p, msq_x4);
            pAp = reDotProductCuda(p, Ap);
            alpha = r2 / pAp;
            r2_old = r2;
            r2 = axpyNormCuda(-alpha, Ap, r_sloppy);

            // reliable update conditions
            rNorm = sqrt(r2);
            if (rNorm > maxrx) maxrx = rNorm;
            if (rNorm > maxrr) maxrr = rNorm;
            int updateX = (rNorm < delta*r0Norm && r0Norm <= maxrx) ? 1 : 0;
            int updateR = ((rNorm < delta*maxrr && r0Norm <= maxrr) || updateX) ? 1 : 0;

            if (!updateR) {
               beta = r2 / r2_old;
               axpyZpbxCuda(alpha, p, x_sloppy, r_sloppy, beta);
            } else {
               axpyCuda(alpha, p, x_sloppy);
               if (x.precision != x_sloppy.precision) copyCuda(x, x_sloppy);
               dslashCuda_st(tmp, fatlinkPrecise, longlinkPrecise, x, 1 - oddBit, 0);
               dslashAxpyCuda(r, fatlinkPrecise, longlinkPrecise, tmp, oddBit, 0, x, msq_x4);

               r2 = xmyNormCuda(b, r);
               if (x.precision != r_sloppy.precision) copyCuda(r_sloppy, r);
               rNorm = sqrt(r2);

               maxrr = rNorm;
               rUpdate++;

               if (updateX) {
                        xpyCuda(x, y);
                        zeroCuda(x_sloppy);
                        copyCuda(b, r);
                        r0Norm = rNorm;

                        maxrx = rNorm;
                        xUpdate++;
               }

               beta = r2 / r2_old;
               xpayCuda(r_sloppy, beta, p);
            }
            gettimeofday(&t1, NULL);
            k++;
            PRINTF("%d iterations, r2 = %e, time=%f\n", k, r2,TDIFF(t1, t0));
  }
  if (x.precision != x_sloppy.precision) copyCuda(x, x_sloppy);
  //y is the solution
  xpyCuda(y, x);

  ......
}


/* The dslash operator entry function in CPU*/

void dslashCuda_st(ParitySpinor out, FullGauge cudaFatLink, FullGauge cudaLongLink,
                           ParitySpinor in, int parity, int dagger)
{
```

```
    if (!initDslash) {
        initDslashCuda(cudaFatLink);
    }
    checkSpinor(in, out);
    checkGaugeSpinor(in, cudaFatLink);

    if (in.precision == QUDA_DOUBLE_PRECISION) {
        dslashDCuda(out, cudaFatLink, cudaLongLink, in, parity, dagger);
    } else if (in.precision == QUDA_SINGLE_PRECISION) {
        dslashSCuda(out, cudaFatLink, cudaLongLink, in, parity, dagger);
    } else if (in.precision == QUDA_HALF_PRECISION) {
        dslashHCuda(out, cudaFatLink, cudaLongLink, in, parity, dagger);
    }
}

void
dslashSCuda(ParitySpinor res, FullGauge flink, FullGauge llink, ParitySpinor spinor,
            int oddBit, int daggerBit)
{
    dim3 gridDim(res.volume/llink.blockDim, 1, 1);
    dim3 blockDim(llink.blockDim, 1, 1);
    bindFatLongLinkTex(flink, llink, oddBit);
    int spinor_bytes = res.length*sizeof(float);
    cudaBindTexture(0, spinorTexSingle, spinor.spinor, spinor_bytes);

    int shared_bytes = blockDim.x*SHARED_FLOATS_PER_THREAD*sizeof(float);

    if (llink.precision == QUDA_DOUBLE_PRECISION) {
        if (llink.reconstruct == QUDA_RECONSTRUCT_12) {
            if (!daggerBit) {
                dslashDS12Kernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            } else {
                dslashDS12DaggerKernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            }
        } else {
            if (!daggerBit) {
                dslashDS8Kernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            } else {
                dslashDS8DaggerKernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            }
        }
    } else if (llink.precision == QUDA_SINGLE_PRECISION) {
        if (llink.reconstruct == QUDA_RECONSTRUCT_12) {
            if (!daggerBit) {
                dslashSS12Kernel<<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            } else {
                dslashSS12DaggerKernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            }
        } else {
            if (!daggerBit) {
                dslashSS8Kernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            } else {
                dslashSS8DaggerKernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            }
        }
    } else {
        if (llink.reconstruct == QUDA_RECONSTRUCT_12) {
            if (!daggerBit) {
                dslashHS12Kernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            } else {
                dslashHS12DaggerKernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            }
        } else {
            if (!daggerBit) {
                dslashHS8Kernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
            } else {
```

```
                        dslashHS8DaggerKernel <<<gridDim, blockDim, shared_bytes>>> ((float2 *)res.spinor, oddBit); CUERR;
                    }
                }
    }

}

/* one of dslash kernel in GPU
 * (single precision spinor, single precision links, 12 reconstruct for long link)
 */


__global__ void dslashSS12Kernel(float2*, int )
{
    ....

    int sid = blockIdx.x*blockDim.x + threadIdx.x;
    int z1 = FAST_INT_DIVIDE(sid, X1h);
    int x1h = sid - z1*X1h;
    int z2 = FAST_INT_DIVIDE(z1, X2);
    int x2 = z1 - z2*X2;
    int x4 = FAST_INT_DIVIDE(z2, X3);
    int x3 = z2 - x4*X3;
    int x1odd = (x2 + x3 + x4 + oddBit) & 1;
    int x1 = 2*x1h + x1odd;
    int X = 2*sid + x1odd;
    int sign;

    {
                //direction: +X

                if(x4%2 ==1){
                    sign = -1;
                }else{
                    sign =1;
                }
                int sp_idx_1st_nbr = ((x1==X1m1) ? X-X1m1 : X+1) >> 1;
                int sp_idx_3rd_nbr = ((x1 > (X1 -4)) ? X -X1 +3 : X+3) >> 1;

                int ga_idx = sid;

                // read gauge matrix from device memory
                READ_FAT_MATRIX(FATLINK0TEX, 0, ga_idx);
                READ_LONG_MATRIX(LONGLINK0TEX, 0, ga_idx);

                // read spinor from device memory
                READ_1ST_NBR_SPINOR(SPINORTEX, sp_idx_1st_nbr);
                READ_3RD_NBR_SPINOR(SPINORTEX, sp_idx_3rd_nbr);

                // reconstruct gauge matrix
                RECONSTRUCT_GAUGE_MATRIX(0, long, ga_idx, sign);

                MAT_MUL_V(A, fat, i);
                MAT_MUL_V(B, long, t);

                o00_re += A0_re;
                o00_im += A0_im;
                o01_re += A1_re;
                o01_im += A1_im;
                o02_re += A2_re;
                o02_im += A2_im;

                o00_re += B0_re;
                o00_im += B0_im;
                o01_re += B1_re;
                o01_im += B1_im;
```

```
            o02_re += B2_re;
            o02_im += B2_im;
    }

    ... //compute in direction -x, +y, -y, +z, -z, +t, -t
            //Write result into device memory

}

#define READ_FAT_MATRIX_18_SINGLE(gauge, dir, idx)              \
    float2 FAT0 = tex1Dfetch((gauge), idx + ((dir/2)*9+0)*Vh);      \
    float2 FAT1 = tex1Dfetch((gauge), idx + ((dir/2)*9+1)*Vh);      \
    float2 FAT2 = tex1Dfetch((gauge), idx + ((dir/2)*9+2)*Vh);      \
    float2 FAT3 = tex1Dfetch((gauge), idx + ((dir/2)*9+3)*Vh);      \
    float2 FAT4 = tex1Dfetch((gauge), idx + ((dir/2)*9+4)*Vh);      \
    float2 FAT5 = tex1Dfetch((gauge), idx + ((dir/2)*9+5)*Vh);      \
    float2 FAT6 = tex1Dfetch((gauge), idx + ((dir/2)*9+6)*Vh);      \
    float2 FAT7 = tex1Dfetch((gauge), idx + ((dir/2)*9+7)*Vh);      \
    float2 FAT8 = tex1Dfetch((gauge), idx + ((dir/2)*9+8)*Vh);

#define READ_LONG_MATRIX_12_SINGLE(gauge, dir, idx)                             \
    READ_GAUGE_MATRIX_12_SINGLE(gauge, dir, idx, LONG)

#define READ_GAUGE_MATRIX_12_SINGLE(gauge, dir, idx, var)   \
    float4 var##0 = tex1Dfetch((gauge), idx + ((dir/2)*3+0)*Vh);    \
    float4 var##1 = tex1Dfetch((gauge), idx + ((dir/2)*3+1)*Vh);    \
    float4 var##2 = tex1Dfetch((gauge), idx + ((dir/2)*3+2)*Vh);    \
    float4 var##3 = make_float4(0,0,0,0);                                       \
    float4 var##4 = make_float4(0,0,0,0);

#define RECONSTRUCT_GAUGE_MATRIX_12_SINGLE(dir, gauge, idx, sign)   \
    ACC_CONJ_PROD(gauge##20, +gauge##01, +gauge##12);                           \
    ACC_CONJ_PROD(gauge##20, -gauge##02, +gauge##11);                           \
    ACC_CONJ_PROD(gauge##21, +gauge##02, +gauge##10);                           \
    ACC_CONJ_PROD(gauge##21, -gauge##00, +gauge##12);                           \
    ACC_CONJ_PROD(gauge##22, +gauge##00, +gauge##11);                           \
    ACC_CONJ_PROD(gauge##22, -gauge##01, +gauge##10);                           \
    if (1){float u0 = coeff_f*sign;                                             \
            gauge##20_re *=u0;gauge##20_im *=u0; gauge##21_re *=u0; gauge##21_im *=u0; \
            gauge##22_re *=u0;gauge##22_im *=u0;}


#define READ_1ST_NBR_SPINOR_SINGLE(spinor, idx)     \
  float2 I0 = tex1Dfetch((spinor), idx + 0*Vh);   \
  float2 I1 = tex1Dfetch((spinor), idx + 1*Vh);   \
  float2 I2 = tex1Dfetch((spinor), idx + 2*Vh);

#define READ_3RD_NBR_SPINOR_SINGLE(spinor, idx)     \
  float2 T0 = tex1Dfetch((spinor), idx + 0*Vh);   \
  float2 T1 = tex1Dfetch((spinor), idx + 1*Vh);   \
  float2 T2 = tex1Dfetch((spinor), idx + 2*Vh);
```

6.2 Fat Link GPU Implementation

As an example, llfat MILC function, llfat_cuda, is provided that uses several GPU kernels. Source code of one such kernel, computeGenStapleFieldEvenKernel, is provided as well.

```
void llfat_cuda(void* fatLink, void* siteLink,
        FullGauge cudaFatLink, FullGauge cudaSiteLink,
        FullStaple cudaStaple, FullStaple cudaStaple1,
        QudaGaugeParam* param, void* _act_path_coeff)
{
```

```
float* act_path_coeff = (float*) _act_path_coeff;
int volume = param->X[0]*param->X[1]*param->X[2]*param->X[3];
dim3 gridDim(volume/param->blockDim,1,1);
dim3 halfGridDim(volume/(2*param->blockDim),1,1);
dim3 blockDim(param->blockDim , 1, 1);

cudaBindTexture(0, siteLink0TexSingle, cudaSiteLink.even, cudaSiteLink.bytes);
cudaBindTexture(0, siteLink1TexSingle, cudaSiteLink.odd, cudaSiteLink.bytes);

cudaBindTexture(0, fatLink0TexSingle, cudaFatLink.even, cudaFatLink.bytes);
cudaBindTexture(0, fatLink1TexSingle, cudaFatLink.odd,  cudaFatLink.bytes);

computeFatLinkKernel<<<gridDim, blockDim>>>((float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                    (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd);

for(int dir = 0;dir < 4; dir++){
    for(int nu = 0; nu < 4; nu++){
      if (nu != dir){
          //even
          siteComputeGenStapleEvenKernel<<<halfGridDim, blockDim>>>((float2*)cudaStaple.even, (float2*)cudaStaple.odd,
                                                (float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                                (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd,
                                                dir, nu,
                                                act_path_coeff[2]);
          //odd
          siteComputeGenStapleOddKernel<<<halfGridDim, blockDim>>>((float2*)cudaStaple.even, (float2*)cudaStaple.odd,
                                               (float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                               (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd,
                                               dir, nu,
                                               act_path_coeff[2]);

          cudaBindTexture(0, muLink0TexSingle, cudaStaple.even, cudaStaple.bytes);
          cudaBindTexture(0, muLink1TexSingle, cudaStaple.odd, cudaStaple.bytes);
          //even
          computeGenStapleFieldEvenKernel<<<halfGridDim, blockDim>>>((float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                                (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd,
                                                (float2*)cudaStaple.even, (float2*)cudaStaple.odd,
                                                dir, nu,
                                                act_path_coeff[2]);
          //odd
          computeGenStapleFieldOddKernel<<<halfGridDim, blockDim>>>((float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                               (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd,
                                               (float2*)cudaStaple.even, (float2*)cudaStaple.odd,
                                               dir, nu,
                                               act_path_coeff[2]);
          cudaUnbindTexture(muLink0TexSingle);
          cudaUnbindTexture(muLink1TexSingle);

          for(int rho = 0; rho < 4; rho++){
            if (rho != dir && rho != nu){
                cudaBindTexture(0, muLink0TexSingle, cudaStaple.even, cudaStaple.bytes);
                cudaBindTexture(0, muLink1TexSingle, cudaStaple.odd, cudaStaple.bytes);

                //even
                computeGenStapleFieldSaveEvenKernel<<<halfGridDim, blockDim>>>((float2*)cudaStaple1.even, (float2*)cudaStaple1.odd,
                                                        (float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                                        (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd,
                                                        (float2*)cudaStaple.even, (float2*)cudaStaple.odd,
                                                        dir, rho,
                                                        act_path_coeff[3]);
                //odd
                computeGenStapleFieldSaveOddKernel<<<halfGridDim, blockDim>>>((float2*)cudaStaple1.even, (float2*)cudaStaple1.odd,
                                                       (float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                                       (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd,
                                                       (float2*)cudaStaple.even, (float2*)cudaStaple.odd,
                                                       dir, rho,
```

```
                                                    act_path_coeff[3]);
                cudaUnbindTexture(muLink0TexSingle);
                cudaUnbindTexture(muLink1TexSingle);

                for(int sig = 0; sig < 4; sig++){
                  if (sig != dir && sig != nu && sig != rho){

                    cudaBindTexture(0, muLink0TexSingle, cudaStaple1.even, cudaStaple1.bytes);
                    cudaBindTexture(0, muLink1TexSingle, cudaStaple1.odd,  cudaStaple1.bytes);
                    //even
                    computeGenStapleFieldEvenKernel<<<halfGridDim, blockDim>>>((float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                                        (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd,
                                                        (float2*)cudaStaple1.even, (float2*)cudaStaple1.odd,
                                                        dir, sig,
                                                        act_path_coeff[3]);
                    //odd
                    computeGenStapleFieldOddKernel<<<halfGridDim, blockDim>>>((float4*)cudaSiteLink.even, (float4*)cudaSiteLink.odd,
                                                       (float2*)cudaFatLink.even, (float2*)cudaFatLink.odd,
                                                       (float2*)cudaStaple1.even, (float2*)cudaStaple1.odd,
                                                       dir, sig,
                                                       act_path_coeff[3]);
                    cudaUnbindTexture(muLink0TexSingle);
                    cudaUnbindTexture(muLink1TexSingle);
                  }
                }//sig
              }
            }//rho
          }
        }//nu
    }

    CUERR;
    return;
}



__global__ void
computeGenStapleFieldEvenKernel(float4* sitelink_even, float4* sitelink_odd,
                        float2* fatlink_even, float2* fatlink_odd,
                        float2* mulink_even, float2* mulink_odd,
                        int mu, int nu, float mycoeff)
{
    float4 A0, A1, A2, A3, A4;
    float4 B0, B1, B2, B3, B4;
    float2 BB0, BB1, BB2, BB3, BB4, BB5, BB6, BB7, BB8;
    float4 C0, C1, C2, C3, C4;
    float4 TEMPA0, TEMPA1, TEMPA2, TEMPA3, TEMPA4;
    float2 FAT0, FAT1, FAT2, FAT3, FAT4, FAT5, FAT6, FAT7, FAT8;

    int mem_idx = blockIdx.x*blockDim.x + threadIdx.x;
    int odd_bit=0;

    int z1 = FAST_INT_DIVIDE(mem_idx, X1h);
    int x1h = mem_idx - z1*X1h;
    int z2 = FAST_INT_DIVIDE(z1, X2);
    int x2 = z1 - z2*X2;
    int x4 = FAST_INT_DIVIDE(z2, X3);
    int x3 = z2 - x4*X3;
    int x1odd = (x2 + x3 + x4 + odd_bit) & 1;
    int x1 = 2*x1h + x1odd;
    int X = 2*mem_idx + x1odd;

    int sign =1;
```

```
  int new_mem_idx;


  /* Upper staple */
  /* Computes the staple :
   *              mu (BB)
   *           +-------+
   *      nu   |       |
   *       (A) |       |(C)
   *           X       X
   *
   */

  /* load matrix A and reconstruct*/
  LOAD_EVEN_SITE_MATRIX_12_SINGLE(nu, mem_idx, A);
  LLFAT_RECONSTRUCT_MATRIX_12_SINGLE(nu, mem_idx, sign, a);

  /* load matrix BB*/
  COMPUTE_NEW_IDX_PLUS(nu, X);
  LOAD_ODD_MULINK_MATRIX_18_SINGLE(0, new_mem_idx, BB);

  MULT_SU3_NN(a, bb, tempa);

  /* load matrix C and reconstruct*/
  COMPUTE_NEW_IDX_PLUS(mu, X);
  LOAD_ODD_SITE_MATRIX_12_SINGLE(nu, new_mem_idx, C);
  LLFAT_RECONSTRUCT_MATRIX_12_SINGLE(nu, new_mem_idx, sign, c);

  MULT_SU3_NA_TEST(tempa, c);

  /***************lower staple****************
   *
   *             X      X
   *         nu  |      |
   *          (A)|      | (C)
   *             +-------+
   *              mu (B)
   *
   ******************************************/

  /* load matrix A and reconstruct*/
  COMPUTE_NEW_FULL_IDX_MINUS(nu);
  LOAD_ODD_SITE_MATRIX_12_SINGLE(nu, (new_mem_idx>>1), A);
  LLFAT_RECONSTRUCT_MATRIX_12_SINGLE(nu, (new_mem_idx>>1), sign, a);

  /* load matrix B*/
  LOAD_ODD_MULINK_MATRIX_18_SINGLE(0, (new_mem_idx>>1), BB);
  MULT_SU3_AN(a, bb, b);

  /* load matrix C and reconstruct*/
  COMPUTE_NEW_IDX_PLUS(mu, new_mem_idx);
  LOAD_EVEN_SITE_MATRIX_12_SINGLE(nu, new_mem_idx, C);
  LLFAT_RECONSTRUCT_MATRIX_12_SINGLE(nu, new_mem_idx, sign, c);

  MULT_SU3_NN_TEST(b, c);

  LLFAT_ADD_SU3_MATRIX(b, tempa, b);

  LOAD_EVEN_FAT_MATRIX_18_SINGLE(mu, mem_idx);
  SCALAR_MULT_ADD_SU3_MATRIX(fat, b, mycoeff, fat);

  WRITE_FAT_MATRIX(fatlink_even, mu,  mem_idx);

  return;
}
```

```
#define COMPUTE_NEW_IDX_PLUS(mydir, idx) do {                           \
      switch(mydir){                                                    \
      case 0:                                                           \
        new_mem_idx = ( (x1==X1m1)?idx-X1m1:idx+1)>> 1;                 \
        break;                                                          \
      case 1:                                                           \
        new_mem_idx = ( (x2==X2m1)?idx-X2X1mX1:idx+X1) >> 1;            \
        break;                                                          \
      case 2:                                                           \
        new_mem_idx = ( (x3==X3m1)?idx-X3X2X1mX2X1:idx+X2X1) >> 1;      \
        break;                                                          \
      case 3:                                                           \
        new_mem_idx = ( (x4==X4m1)?idx-X4X3X2X1mX3X2X1:idx+X3X2X1) >> 1; \
        break;                                                          \
      }                                                                 \
  }while(0)


#define COMPUTE_NEW_FULL_IDX_MINUS(mydir) do {                          \
      switch(mydir){                                                    \
      case 0:                                                           \
        new_mem_idx = ( (x1==0)?X+X1m1:X-1);                    \
        break;                                                          \
      case 1:                                                           \
        new_mem_idx = ( (x2==0)?X+X2X1mX1:X-X1);                \
        break;                                                          \
      case 2:                                                           \
        new_mem_idx = ( (x3==0)?X+X3X2X1mX2X1:X-X2X1);                  \
        break;                                                          \
      case 3:                                                           \
        new_mem_idx = ( (x4==0)?X+X4X3X2X1mX3X2X1:X-X3X2X1);            \
        break;                                                          \
      }                                                                 \
  }while(0)


#define LOAD_MATRIX_12_SINGLE(gauge, dir, idx, var)do{                  \
      int start_pos = idx + dir*Vhx3;                           \
      var##0 = gauge[start_pos];                                \
      var##1 = gauge[start_pos + Vh];                           \
      var##2 = gauge[start_pos + Vhx2];                         \
  }while(0)

#define LOAD_MATRIX_12_SINGLE_TEX(gauge, dir, idx, var)do{              \
      int start_pos = idx + dir*Vhx3;                           \
      var##0 = tex1Dfetch(gauge, start_pos);                          \
      var##1 = tex1Dfetch(gauge, start_pos + Vh);               \
      var##2 = tex1Dfetch(gauge, start_pos + Vhx2);                   \
  }while(0)


#if (SITE_MATRIX_LOAD_TEX == 1)
#define LOAD_EVEN_SITE_MATRIX_12_SINGLE(dir, idx, var) LOAD_MATRIX_12_SINGLE_TEX(siteLink0TexSingle, dir, idx, var)
#define LOAD_ODD_SITE_MATRIX_12_SINGLE(dir, idx, var) LOAD_MATRIX_12_SINGLE_TEX(siteLink1TexSingle, dir, idx, var)
#else
#define LOAD_EVEN_SITE_MATRIX_12_SINGLE(dir, idx, var) LOAD_MATRIX_12_SINGLE(sitelink_even, dir, idx, var)
#define LOAD_ODD_SITE_MATRIX_12_SINGLE(dir, idx, var) LOAD_MATRIX_12_SINGLE(sitelink_odd, dir, idx, var)
#endif



#define LLFAT_RECONSTRUCT_MATRIX_12_SINGLE(dir, idx, sign, var)         \
  ACC_CONJ_PROD_ASSIGN(var##20, +var##01, +var##12);                    \
  ACC_CONJ_PROD(var##20, -var##02, +var##11);                           \
  ACC_CONJ_PROD_ASSIGN(var##21, +var##02, +var##10);                    \
```

```
  ACC_CONJ_PROD(var##21, -var##00, +var##12);                          \
  ACC_CONJ_PROD_ASSIGN(var##22, +var##00, +var##11);                   \
  ACC_CONJ_PROD(var##22, -var##01, +var##10);                          \
  var##20_re *=sign;var##20_im *=sign; var##21_re *=sign; var##21_im *=sign; \
  var##22_re *=sign;var##22_im *=sign;



#define LOAD_MATRIX_18_SINGLE(gauge, dir, idx, var)do{                 \
      int start_pos= idx + dir*Vhx9;                                   \
      var##0 = gauge[start_pos];                                  \
      var##1 = gauge[start_pos + Vh];                                  \
      var##2 = gauge[start_pos + Vhx2];                                \
      var##3 = gauge[start_pos + Vhx3];                                \
      var##4 = gauge[start_pos + Vhx4];                                \
      var##5 = gauge[start_pos + Vhx5];                                \
      var##6 = gauge[start_pos + Vhx6];                                \
      var##7 = gauge[start_pos + Vhx7];                                \
      var##8 = gauge[start_pos + Vhx8]; } while(0)

#define LOAD_MATRIX_18_SINGLE_TEX(gauge, dir, idx, var)do{              \
      int start_pos= idx + dir*Vhx9;                                   \
      var##0 = tex1Dfetch(gauge, start_pos);                           \
      var##1 = tex1Dfetch(gauge, start_pos + Vh);                      \
      var##2 = tex1Dfetch(gauge, start_pos + Vhx2);                    \
      var##3 = tex1Dfetch(gauge, start_pos + Vhx3);                    \
      var##4 = tex1Dfetch(gauge, start_pos + Vhx4);                    \
      var##5 = tex1Dfetch(gauge, start_pos + Vhx5);                    \
      var##6 = tex1Dfetch(gauge, start_pos + Vhx6);                    \
      var##7 = tex1Dfetch(gauge, start_pos + Vhx7);                    \
      var##8 = tex1Dfetch(gauge, start_pos + Vhx8); } while(0)


#if (MULINK_LOAD_TEX == 1)
#define LOAD_EVEN_MULINK_MATRIX_18_SINGLE(dir, idx, var) LOAD_MATRIX_18_SINGLE_TEX(muLink0TexSingle, dir, idx, var)
#define LOAD_ODD_MULINK_MATRIX_18_SINGLE(dir, idx, var) LOAD_MATRIX_18_SINGLE_TEX(muLink1TexSingle, dir, idx, var)
#else
#define LOAD_EVEN_MULINK_MATRIX_18_SINGLE(dir, idx, var) LOAD_MATRIX_18_SINGLE(mulink_even, dir, idx, var)
#define LOAD_ODD_MULINK_MATRIX_18_SINGLE(dir, idx, var) LOAD_MATRIX_18_SINGLE(mulink_odd, dir, idx, var)
#endif

#if (FATLINK_LOAD_TEX == 1)
#define LOAD_EVEN_FAT_MATRIX_18_SINGLE(dir, idx) LOAD_MATRIX_18_SINGLE_TEX(fatLink0TexSingle, dir, idx, FAT)
#define LOAD_ODD_FAT_MATRIX_18_SINGLE(dir, idx) LOAD_MATRIX_18_SINGLE_TEX(fatLink1TexSingle, dir, idx, FAT)
#else
#define LOAD_EVEN_FAT_MATRIX_18_SINGLE(dir, idx) LOAD_MATRIX_18_SINGLE(fatlink_even, dir, idx, FAT)
#define LOAD_ODD_FAT_MATRIX_18_SINGLE(dir, idx)  LOAD_MATRIX_18_SINGLE(fatlink_odd, dir, idx, FAT)
#endif

#define MULT_SU3_NN(ma, mb, mc)                                        \
  mc##00_re =                                                    \
      ma##00_re * mb##00_re - ma##00_im * mb##00_im +            \
      ma##01_re * mb##10_re - ma##01_im * mb##10_im +            \
      ma##02_re * mb##20_re - ma##02_im * mb##20_im;             \
  mc##00_im =                                                    \
      ma##00_re * mb##00_im + ma##00_im * mb##00_re +            \
      ma##01_re * mb##10_im + ma##01_im * mb##10_re +            \
      ma##02_re * mb##20_im + ma##02_im * mb##20_re;             \
  mc##10_re =                                                    \
      ma##10_re * mb##00_re - ma##10_im * mb##00_im +            \
      ma##11_re * mb##10_re - ma##11_im * mb##10_im +            \
      ma##12_re * mb##20_re - ma##12_im * mb##20_im;             \
  mc##10_im =                                                    \
      ma##10_re * mb##00_im + ma##10_im * mb##00_re +            \
      ma##11_re * mb##10_im + ma##11_im * mb##10_re +            \
      ma##12_re * mb##20_im + ma##12_im * mb##20_re;             \
```

```
mc##20_re =                                     \
    ma##20_re * mb##00_re - ma##20_im * mb##00_im +     \
    ma##21_re * mb##10_re - ma##21_im * mb##10_im +     \
    ma##22_re * mb##20_re - ma##22_im * mb##20_im;      \
mc##20_im =                                     \
    ma##20_re * mb##00_im + ma##20_im * mb##00_re +         \
    ma##21_re * mb##10_im + ma##21_im * mb##10_re +         \
    ma##22_re * mb##20_im + ma##22_im * mb##20_re;      \
mc##01_re =                                     \
    ma##00_re * mb##01_re - ma##00_im * mb##01_im +     \
    ma##01_re * mb##11_re - ma##01_im * mb##11_im +     \
    ma##02_re * mb##21_re - ma##02_im * mb##21_im;      \
mc##01_im =                                     \
    ma##00_re * mb##01_im + ma##00_im * mb##01_re +         \
    ma##01_re * mb##11_im + ma##01_im * mb##11_re +         \
    ma##02_re * mb##21_im + ma##02_im * mb##21_re;      \
mc##11_re =                                     \
    ma##10_re * mb##01_re - ma##10_im * mb##01_im +     \
    ma##11_re * mb##11_re - ma##11_im * mb##11_im +     \
    ma##12_re * mb##21_re - ma##12_im * mb##21_im;      \
mc##11_im =                                     \
    ma##10_re * mb##01_im + ma##10_im * mb##01_re +         \
    ma##11_re * mb##11_im + ma##11_im * mb##11_re +         \
    ma##12_re * mb##21_im + ma##12_im * mb##21_re;      \
mc##21_re =                                     \
    ma##20_re * mb##01_re - ma##20_im * mb##01_im +     \
    ma##21_re * mb##11_re - ma##21_im * mb##11_im +     \
    ma##22_re * mb##21_re - ma##22_im * mb##21_im;      \
mc##21_im =                                     \
    ma##20_re * mb##01_im + ma##20_im * mb##01_re +         \
    ma##21_re * mb##11_im + ma##21_im * mb##11_re +         \
    ma##22_re * mb##21_im + ma##22_im * mb##21_re;      \
mc##02_re =                                     \
    ma##00_re * mb##02_re - ma##00_im * mb##02_im +     \
    ma##01_re * mb##12_re - ma##01_im * mb##12_im +     \
    ma##02_re * mb##22_re - ma##02_im * mb##22_im;      \
mc##02_im =                                     \
    ma##00_re * mb##02_im + ma##00_im * mb##02_re +         \
    ma##01_re * mb##12_im + ma##01_im * mb##12_re +         \
    ma##02_re * mb##22_im + ma##02_im * mb##22_re;      \
mc##12_re =                                     \
    ma##10_re * mb##02_re - ma##10_im * mb##02_im +     \
    ma##11_re * mb##12_re - ma##11_im * mb##12_im +     \
    ma##12_re * mb##22_re - ma##12_im * mb##22_im;      \
mc##12_im =                                     \
    ma##10_re * mb##02_im + ma##10_im * mb##02_re +         \
    ma##11_re * mb##12_im + ma##11_im * mb##12_re +         \
    ma##12_re * mb##22_im + ma##12_im * mb##22_re;      \
mc##22_re =                                     \
    ma##20_re * mb##02_re - ma##20_im * mb##02_im +     \
    ma##21_re * mb##12_re - ma##21_im * mb##12_im +     \
    ma##22_re * mb##22_re - ma##22_im * mb##22_im;      \
mc##22_im =                                     \
    ma##20_re * mb##02_im + ma##20_im * mb##02_re +         \
    ma##21_re * mb##12_im + ma##21_im * mb##12_re +         \
    ma##22_re * mb##22_im + ma##22_im * mb##22_re;
```