Porting MILC to GPU: Lessons learned

Dylan Roeh
Jonathan Troup
Guochun Shi
Volodymyr Kindratenko

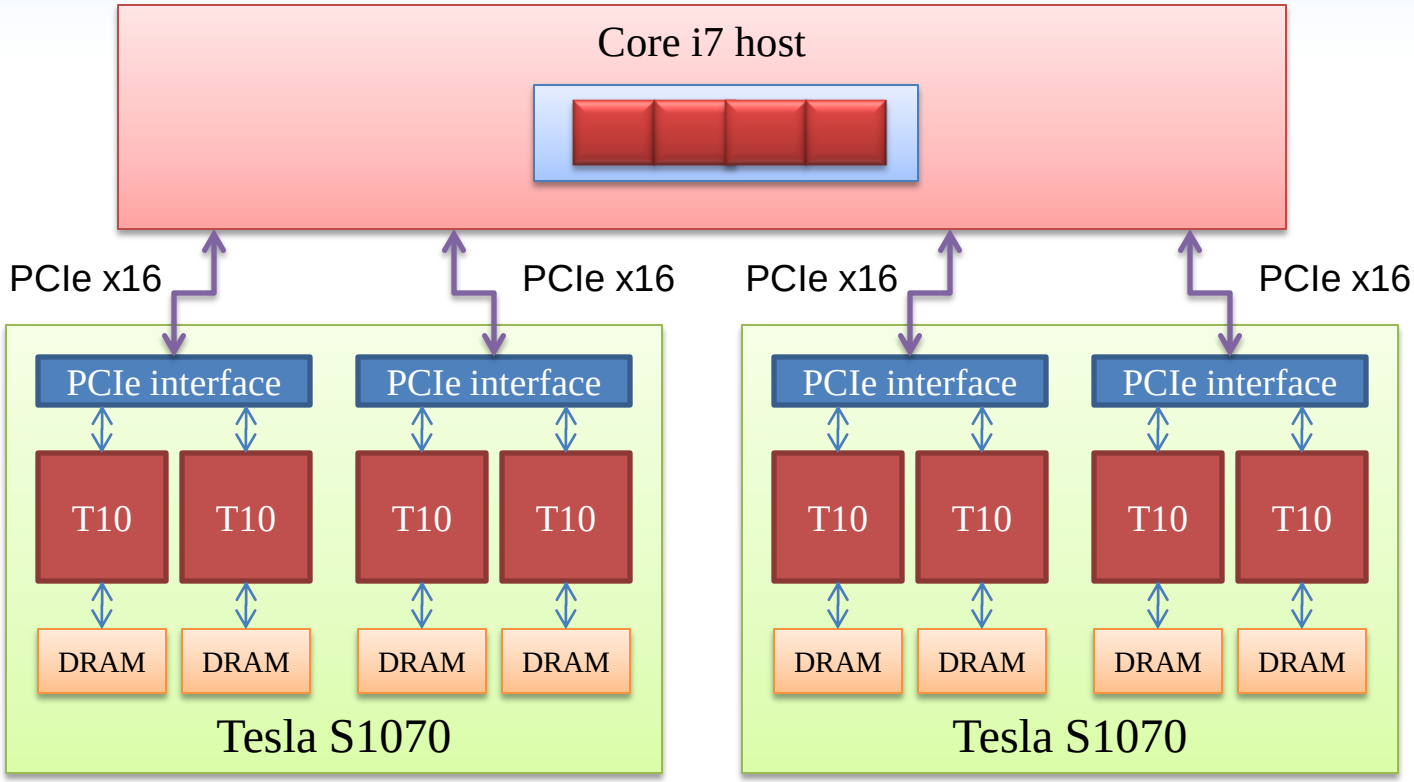
Innovative Systems Laboratory

National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

GPU Clusters at NCSA

- **Lincoln**
  - Production system available via TeraGrid HPC allocation
    - Nodes: 192
    - CPU cores: 1536
    - Accelerator Units (S1070): 96
    - Total GPUs: 384
    - CPU cores/GPU ratio: 4
- **AC**
  - Experimental system managed by ISL
    - Nodes: 32
    - CPU cores: 128
    - Accelerator Units (S1070): 32
    - Total GPUs: 128
    - CPU cores/ GPU ratio: 1

New Experimental Cluster Node



- CPU cores (Intel core i7): 4
- Accelerator Units (S1070): 2
- Total GPUs: 8
- CPU cores/GPU ratio: 0.5
- Theoretical peak PCIe bandwidth: 18 GB/s

- CPU Memory: 24 GB
- GPU Memory: 32 GB

Goals/Approach

- **Goals**
  - Understand what is involved in implementing MILC on GPU
  - Understand GPU cluster hardware requirements for MILC
- **Approach**
  - Port few kernels using PGI x64+GPU compiler
  - Port few kernels using native NVIDIA CUDA C compiler

Portland Group PGI x64+GPU compiler

- **Uses OpenMP-like directives to identify portions of code for moving to GPU**
  - No need for programming in CUDA C
  - Programmer is still responsible for parallelizing the code
- **Supports C, C++, and Fortran(!)**
- **Supports 64-bit Linux and latest NVIDIA GPUs**

Target code

```
for (i=0;i < sites_on_node; i++) {
        mult_su3_na(…, …, …)
}
```

```
void mult_su3_na(su3_matrix *a,
su3_matrix *b, su3_matrix *c )
{
register int i,j,k;
register complex x,y;
   for(i=0;i<3;i++)for(j=0;j<3;j++) {
      x.real=x.imag=0.0;
      for(k=0;k<3;k++) {
         CMUL_J(a->e[i][k] , b->e[j][k] , y);
         CSUM( x , y );
      }
      c->e[i][j] = x;
   }
}
```

Target code

```
#pragma acc region
for (i=0;i < sites_on_node; i++) {
            mult_su3_na(…, …, …)
}
```

➔PGI compiler does not like structures

```
void mult_su3_na(su3_matrix *a,
su3_matrix *b, su3_matrix *c )
{
register int i,j,k;
register complex x,y;
   for(i=0;i<3;i++)for(j=0;j<3;j++) {
      x.real=x.imag=0.0;
      for(k=0;k<3;k++) {
         CMUL_J(a->e[i][k] , b->e[j][k] , y);
         CSUM( x , y );
      }
      c->e[i][j] = x;
   }
}
```

Modified code

```
#pragma acc region
for (i=0;i < sites_on_node; i++){
            mult_su3_na(…, …, …)
}
```

Remove structures and make it pure pointers

➔PGI compiler does not like pointers either

```
inline void mult_su3_na_plain(float* a, float* b, float* c)
{
 int i,j,k;
 for (i =0;i < 3;i++)
  for (j=0;j< 3;j++) {
   float real_part=0;
   float imag_part=0;
   for (k=0;k <3; k++) {
    float real_a  = *(a + i*6+ k*2);
    float imag_a = *(a+ i*6+k*2+1);
    float real_b = *(b + j*6+k*2);
    float imag_b = *(b + j*6+k*2 +1);
    real_part += real_a*real_b - imag_a* imag_b;
    imag_part += real_a*imag_b + imag_b*real_a;
   }
   float* real_p = c + i*6+j*2;
   float* imag_p = c + i*6+j*2 + 1;
   *real_p = real_part;
   *imag_p = imag_part;
  }
}
```

Modified code

Copy data into a continuous memory region

```
float matrix_buf_a[MAX_FLOAT_COUNT];
float matrix_buf_b[MAX_FLOAT_COUNT];
float matrix_buf_c[MAX_FLOAT_COUNT];
for(i=0;i < sites_on_node; i++){
 memcpy( &matrix_buf_a[18*i],
        &oprod_along_path[length-ilink][i],
        sizeof(su3_matrix));
    memcpy( &matrix_buf_b[18*i],
        &mats_along_path[ilink][i], sizeof(su3_matrix));
    memcpy( &matrix_buf_c[18*i], &mat_tmp0[i],
        sizeof(su3_matrix));
}

#pragma acc region
  for(i=0;i < sites_on_node; i++){
        //unroll completely
}
```

Fully unroll the function

```
int ii =0; int jj = 0;
real_a = matrix_buf_a[ii*6 +     18*i];
imag_a = matrix_buf_a[ii*6 + 1 + 18*i];
real_b = matrix_buf_b[jj*6 +     18*i];
imag_b = matrix_buf_b[jj*6 + 1 + 18*i];

real_part += real_a*real_b - imag_a* imag_b;
imag_part += real_a*imag_b + imag_a* real_b;

real_a = matrix_buf_a[ii*6 + 1+ 1 + 18*i];
imag_a = matrix_buf_a[ii*6 + 3 + 18*i];
real_b = matrix_buf_b[jj*6 + 2 + 18*i];
imag_b=  matrix_buf_b[jj*6 + 3 + 18*i];
real_part += real_a*real_b - imag_a*imag_b;
imag_part += real_a*imag_b + imag_a*real_b;

real_a  = matrix_buf_a[ii*6+ 4+18*i];
imag_a = matrix_buf_a[ii*6+4+1+18*i];
real_b = matrix_buf_b[jj*6+4+18*i];
imag_b = matrix_buf_b[jj*6+4 +1+18*i];
real_part += real_a*real_b - imag_a* imag_b;
imag_part += real_a*imag_b + imag_a*real_b;

matrix_buf_c[ii*6+jj*2 + 18*i]= real_part;
matrix_buf_c[ii*6+jj*2 + 1 + 18*i]= imag_part;

ii=0;jj=1;    .....
```

Conclusions with PGI x64+GPU compiler

- **Significant code modifications are necessary**
  - To work around compiler limitations
  - To parallelize the code
- **The benefits are not entirely clear**
  - Even though there is no need to write in CUDA C, code modifications are very similar
  - Compiler transformations are not always visible/obvious

Implementing kernels in CUDA C

- **Use part of the fermion forces kernel as an example (fermion_force_fn_multi.c)**
- **Two approaches**
  - Port the entire kernel
  - Implement 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

Implementing kernels in CUDA C

- **First approach: port the entire kernel**
  - Does not particularly increase the FLOPS/bytes ratio
  - Involves some significant possibility of branching

- **Second approach: implement parallel SU(3) operations**
  - May be an easier strategy to move the code to GPU, but
    - One have to deal with the low FLOPS/bytes ratio in many of the operations

Possible ways forward

- **Consider packed SU(3) representations rather than the 3x3 complex matrix representation**
  - Given the first two rows of the matrix (6 complex numbers, or 12 real numbers), one can reconstruct the third by taking the cross product of the conjugates of those rows.
    - FLOPS are cheap on the GPU!
  - They can also be packed as 8 real numbers, using a method described in the paper by Bunk and Sommer.
  - This will require to adapt MILC to use packed SU(3) matrices all the way through, and offload any significant loop of SU(3) operations to the GPU.

Possible ways forward

- **Switch storage of SU(3) matrices from an array-of-structs to a struct-of-arrays**
  - To allow for easy coalesced global memory reads/writes.
  - To eliminate additional data manipulation on the CPU before it is shipped to the GPU memory
    - but converting the data inside of fermion_force_fn_multi will reduce or eliminate the performance advantage of offloading some of the loops to the GPU

Conclusions

- **Porting MILC "as is" to GPU is not the right strategy because of the low FLOPS/byte ratio throughout many MILC kernels**
  - And the shear number of small kernels!
- **A more fruitfully approach is to start redesigning essential algorithms in the GPU-friendly manner**
  - Issues to consider include data layout and algorithms selection to maximize FLOPS/byte ratio