Accelerating Quantum Chromodynamics Calculations with GPUs

Guochun Shi, Steven Gottlieb, Aaron Torok, Volodymyr Kindratenko

NCSA & Indiana University
Outline

- Background
- New staggered code
- Implementation in GPU
- Benchmarks
- Production use experience
- Where to get the code
- Future
Background

- NCSA’s Innovative Systems Laboratory
- 8/2009: Prof. Steven Gottlieb sabbatical at NCSA
- We are extending QUDA to staggered quarks
Introduction: Lattice QCD

Four dimensional space-time Lattice QCD.

- Quantum Chromodynamics (QCD) studies the strong force (color force) interaction between quarks and gluons.
- QCD is highly nonlinear problem thus lattice QCD approach is introduced. The computation is performed on 4-D space-time lattice.
Lattice QCD computation on CPU

Time distribution for a run on 2048 XT3 (BigBen) cpus using a $40^3 \times 96$ grid ($5 \times 10^2 \times 6$ per cpu) with $m_l = 0.1m_s$:

<table>
<thead>
<tr>
<th>Activity</th>
<th>time(s)</th>
<th>MF/cpu</th>
<th>per cent</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>2987</td>
<td>530</td>
<td>58.5</td>
</tr>
<tr>
<td>FF</td>
<td>1125</td>
<td>579</td>
<td>22.0</td>
</tr>
<tr>
<td>GF</td>
<td>489</td>
<td>469</td>
<td>9.5</td>
</tr>
<tr>
<td>Fat</td>
<td>442</td>
<td>627</td>
<td>8.7</td>
</tr>
<tr>
<td>Long</td>
<td>24</td>
<td>340</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Input config.</td>
<td>41</td>
<td></td>
<td>&lt;1</td>
</tr>
<tr>
<td>total above</td>
<td>5108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unaccounted</td>
<td>104</td>
<td></td>
<td>1.9</td>
</tr>
<tr>
<td>wallclock</td>
<td>5212</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Two phases in Lattice QCD computation
  - Configuration generation: compute a snapshot of state of the strong force field, Conjugate Gradient (CG), fermion force (FF), gauge force (GF) and link fattening (FAT), see above table for the time distribution in this phase.
  - Analysis: the observables of interest are computed over the configurations. Conjugate Gradient (CG) is even more dominant in this phase.
  - In short, CG is the dominant part, others are important.
## Development status

**MILC applications**

**QUDA GPU library**

<table>
<thead>
<tr>
<th>Description</th>
<th>Status</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CG</strong> Update spinors using the neighboring links and spinors through Conjugate Gradient process</td>
<td>Fully implemented (18, 12 and 8-reconstruct, SP, DP and half precision, mixed precisions)</td>
<td>4</td>
</tr>
<tr>
<td><strong>FAT</strong> Update fatlink using the neighboring links</td>
<td>Implemented 12-reconstruct, single precision case</td>
<td>1</td>
</tr>
<tr>
<td><strong>GF</strong> Update the momentum using the neighboring links</td>
<td>Implemented 12-reconstruct, single precision</td>
<td>1</td>
</tr>
<tr>
<td><strong>FF</strong> Update the momentum using the neighboring links and spinors</td>
<td>Implemented 12-reconstruct, single precision</td>
<td>1</td>
</tr>
</tbody>
</table>
New Staggered Code

• Starting code: QUDA 0.2 from Boston University
• First effort was staggered Dslash for single GPU
• Extended to CG and then multimass CG
• Fat link computation / Gauge Force / Fermion Force
• Wrappers allow call of GPU code from MILC
• Code merged into BU QUDA code which had evolved from initial release
• Multi-GPU code over time dimension works. Multi-GPU development over multiple dimensions in development
• For this talk, I will focus on Staggered Dslash, CG/multimass CG and multi-GPU CG
Lattice QCD: Conjugate Gradient

\[ M\phi = b \]

where \( \phi_{i,x} \) and \( b_{i,x} \) are complex vectors 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 a constant, and the matrix \( D \) (called “D slash”) is given by

\[
D_{x,i,y,j} = \sum_{\mu=1}^{4} (U_{x,\mu}^{F} \delta_{y,x+\mu}^{i} - U_{x-\mu,\mu}^{F}^{+} \delta_{y,x-\mu}^{i,j}) + \sum_{\mu=1}^{4} (U_{x,\mu}^{L} \delta_{y,x+3\mu}^{i,j} - U_{x-3\mu,\mu}^{L}^{+} \delta_{y,x-3\mu}^{i,j})
\]

The linear system (3) is solved using a conjugate gradient method after recasting it in the positive definite form

\[ M^{\dagger}M\phi = M^{\dagger}b. \]

where

\[ M^{\dagger}M = (2ma)^{2}I + D^{\dagger}D \]
Staggered Dslash CPU data layout

- Each site contains:
  - 1 spinor (1x3 complex)
  - 4 fatlink (3x3 complex)
  - 4 longlink (3x3 complex)
  Long link can be reconstructed with 12 or 8 real numbers if it is unitary

- Sites are divided into even and odd sites. For site \((x,y,z,t)\)
  - \((x+y+z+t)\%2 == 0 \Rightarrow \text{even site}\)
  - \((x+y+z+t)\%2 == 1 \Rightarrow \text{odd site}\)

- Total number of sites
  - \(V = \text{dim}X \times \text{dim}Y \times \text{dim}Z \times \text{dim}T\)
  - Half of total sites \(V_h = V/2\)

Spinor

- 6\(\times V\) floats
- Each spinor contains 6 \((3\times2)\) floats

Fatlink

- Array of pointers
- Each link contains 18 floats \((3\times3\times2)\)

Longlink

- Same as fatlink

---

SAAHPC 2010
Staggered Dslash reference implementation

```c
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<Wh*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] + Wh*gaugeSiteSize;
    longlinkEven[dir] = longlink[dir];
    longlinkOdd[dir] = longlink[dir] + Wh*gaugeSiteSize;
}
for (int i = 0; i < Wh; 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);
        }
    }
}
```
Spinor CPU-> GPU mapping

Host memory spinor data layout

Host memory parity spinor data layout

Device memory parity spinor data layout

One spinor

Vh *float2

float2 I0 = tex1DFetch((spinor), sp_idx + 0*Vh);
float2 I1 = tex1DFetch((spinor), sp_idx + 1*Vh);
float2 I2 = tex1DFetch((spinor), sp_idx + 2*Vh);

#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);
Link CPU -> GPU mapping

Host memory links data layout

Intermediate data format

Device memory links data layout

GPU kernel code to read link

#define READ_FAT_MATRIX_12_SINGLE(gauge, dir, idx)
float4 PAT0 = tex1Dfetch((gauge), idx + ((dir/2)*3+0)*Vh);
float4 PAT1 = tex1Dfetch((gauge), idx + ((dir/2)*3+1)*Vh);
float4 PAT2 = tex1Dfetch((gauge), idx + ((dir/2)*3+2)*Vh);
float4 PAT3 = make_float4(0,0,0,0);
float4 PAT4 = make_float4(0,0,0,0);
Dslash GPU Kernel
(only shows computation in +x direction)

#define MAT_V_D(VOUT, U, V)
spino0Lost VOUT[V0U][V] = (V[0][0] * u[0] + V[0][0] * u[0] + V[0][0] * u[0]) + (V[0][0] * u[0] + V[0][0] * u[0] + V[0][0] * u[0]) + (V[0][0] * u[0] + V[0][0] * u[0] + V[0][0] * u[0]);

#define MAT_V_L(VOUT, U, V)
spino0Lost VOUT[V0U][V] = (V[0][0] * u[0] + V[0][0] * u[0] + V[0][0] * u[0]) + (V[0][0] * u[0] + V[0][0] * u[0] + V[0][0] * u[0]) + (V[0][0] * u[0] + V[0][0] * u[0] + V[0][0] * u[0]);

int //diation +X
if (00 < x < 15)
    sig = -1;
else
    sig = 1;

// read gauge matrix from device memory
READ_FAT_MATRIX(FAT_MATRIX, U, ga_idx);
READ_LONG_MATRIX(LONG_MATRIX, U, ga_idx);

// read spinor from device memory
READ_SPINOR_SPINOR(SPINOR_MATRIX, sp_idx, 1, 1, sp_stride);
READ_SPINOR_SPINOR(SPINOR_MATRIX, sp_idx, 3, 1, sp_stride);

// reconstruct gauge matrix
MAT_M(X, U, V, 1, 1);
MAT_M(V, U, V, 1, 1, 0);

o0.re = A0.re;
o0.in = A0.in;
o1.re = A+1.re;
o1.in = A+1.in;
o2.re = A+2.re;
o2.in = A+2.in;
o0.re = B0.re;
o0.in = B0.in;
o1.re = B+1.re;
o1.in = B+1.in;
o2.re = B+2.re;
o2.in = B+2.in;
Dslash optimization technique summary

- Data arrangement in device memory to enable coalesced read and write
- Using textures to read data improves the performance
  - We find mixing texture and direct memory read has better performance
  - The best configuration is different for GT200 and Fermi card. More on this later
- Use padding memory to avoid partition camping
- Loop unroll for computations over eight directions
- Use constant memory to store common used constant variables (X1, X2, X3, X4, X1m1, X2X1mX1, …)
CG in MILC

- Dslash is the most time consuming part in CG
- Other than the dslash, blas routines are also implemented for GPUs
- Mixed precision is also implemented. The bulk of work is done in lower precision, while higher precision is used to compute defect correction. The final solution is as accurate as the higher precision.
- Multimass (multi shift) solver is also implemented. The multimass solvers computes solutions for many equations with different mass but the same Dslash operator.
## Hardware Comparison

<table>
<thead>
<tr>
<th>Type</th>
<th>Cores</th>
<th>BW(GB/s)</th>
<th>SP (GF/s)</th>
<th>DP (GF/S)</th>
<th>RAM(GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX280</td>
<td>240</td>
<td>142</td>
<td>1051</td>
<td>87</td>
<td>1.0</td>
</tr>
<tr>
<td>Tesla C1060</td>
<td>240</td>
<td>102</td>
<td>933</td>
<td>78</td>
<td>4.0</td>
</tr>
<tr>
<td>Fermi GTX480</td>
<td>480</td>
<td>177</td>
<td>1345</td>
<td>168</td>
<td>1.5</td>
</tr>
<tr>
<td>Fermi C2050</td>
<td>448</td>
<td>144</td>
<td>1030</td>
<td>515</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- NSRSC: Fermi C2050
- NCSA: other GPUs
### Performance for different loading method

**GTX480 L1/shared=16/48 KB**  
**Single precision**

<table>
<thead>
<tr>
<th>reading method</th>
<th>reconstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fatlink</td>
<td>12</td>
</tr>
<tr>
<td>Longlink</td>
<td>104</td>
</tr>
<tr>
<td>Spinor</td>
<td>107</td>
</tr>
<tr>
<td>D</td>
<td>110</td>
</tr>
<tr>
<td>D</td>
<td>111</td>
</tr>
<tr>
<td>D</td>
<td>113</td>
</tr>
<tr>
<td>T</td>
<td>113</td>
</tr>
<tr>
<td>T</td>
<td>115</td>
</tr>
<tr>
<td>T</td>
<td>115</td>
</tr>
<tr>
<td>T</td>
<td>104</td>
</tr>
</tbody>
</table>

- **D** = direct read
- **T** = texture read

**Winner**
Performance for different loading method
GTX480 L1/shared=48/16 KB
Double precision

<table>
<thead>
<tr>
<th>reading method</th>
<th>reconstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fatlink</td>
<td>Longlink</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>T</td>
</tr>
<tr>
<td>D</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>D</td>
</tr>
<tr>
<td>T</td>
<td>D</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

D = direct read
T = texture read
* The code does not converge, something is wrong
## Dslash performance on GTX280

<table>
<thead>
<tr>
<th></th>
<th>reconstruct</th>
<th>Dslash (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>12</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>35</td>
</tr>
<tr>
<td>SP</td>
<td>12</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>85</td>
</tr>
<tr>
<td>HP</td>
<td>12</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>109</td>
</tr>
</tbody>
</table>

Lattice size: $24^3 \times 32$
### CG/multimass CG performance on GTX280

<table>
<thead>
<tr>
<th></th>
<th>reconstruct</th>
<th>CG (GFLOPS)</th>
<th>multimass CG (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DP</strong></td>
<td>12</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td><strong>SP</strong></td>
<td>12</td>
<td>98</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>108</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>83</td>
<td>80</td>
</tr>
<tr>
<td><strong>HP</strong></td>
<td>12</td>
<td>123</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>128</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>108</td>
<td>98</td>
</tr>
</tbody>
</table>

$24^3 \times 32$ lattice. Four masses for last column.
GTX280 performance with data movement overhead (GFLOPS)

<table>
<thead>
<tr>
<th></th>
<th>kernel-only</th>
<th>Target speed assuming 100GB/s data transfer limit</th>
<th>Including PCIe overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>98</td>
<td>100</td>
<td>71</td>
</tr>
<tr>
<td>Multi mass CG</td>
<td>92</td>
<td>100</td>
<td>71</td>
</tr>
</tbody>
</table>

- Single precision
- 12-reconstruct
- Measured for 500 iterations
- $24^3 \times 32$ lattice
## Fermi Results (GFLOPS)

<table>
<thead>
<tr>
<th></th>
<th>reconstruction</th>
<th>GTX 280</th>
<th>GTX 480</th>
<th>C2050 ECC</th>
<th>C2050 no ECC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>12</td>
<td>29</td>
<td>31</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>15</td>
<td>16</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>32</td>
<td>50</td>
<td>30</td>
<td>41</td>
</tr>
<tr>
<td>SP</td>
<td>12</td>
<td>92</td>
<td>116</td>
<td>66</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>99</td>
<td>126</td>
<td>72</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>79</td>
<td>104</td>
<td>57</td>
<td>86</td>
</tr>
<tr>
<td>HP</td>
<td>12</td>
<td>77</td>
<td>154</td>
<td>97</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>74</td>
<td>157</td>
<td>101</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>76</td>
<td>131</td>
<td>84</td>
<td>104</td>
</tr>
</tbody>
</table>

- Lattice size $24^3 \times 32$
- The code running this test is multi-GPU version running on one GPU
MULTIGPU benchmarks

- Two kernels used for multiGPU Dslash:
  - Interior kernel
    summing contributions for all directions for t=3,4 ...T-3 and spatial contributions for other boundary t values.
  - Exterior kernel
    adding in the terms in the time direction that need on off-node spinors

- For \(24^3\) spatial size with single precision on GTX280:
  - pack (D2H, 0.29ms, 3.3GB/s)
  - MPI (0.16ms, 6.14GB/s)
  - unpack(H2D, 0.20ms, 4.8 GB/s)
  - only 1.5 GB/s aggregate message rate!
CG performance (GFLOPS) per GPU on AC’s compute nodes (S1070)

<table>
<thead>
<tr>
<th>reconst</th>
<th>1 GPU</th>
<th>2 GPUs</th>
<th>4 GPUs</th>
<th>8 GPUs</th>
<th>12 GPUs</th>
<th>16 GPUs</th>
<th>20 GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>12</td>
<td>22</td>
<td>22</td>
<td>21</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>18</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>21</td>
<td>18</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>SP</td>
<td>12</td>
<td>58</td>
<td>56</td>
<td>43</td>
<td>40</td>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>65</td>
<td>56</td>
<td>40</td>
<td>39</td>
<td>32</td>
<td>33</td>
<td>32</td>
</tr>
<tr>
<td>18</td>
<td>50</td>
<td>50</td>
<td>43</td>
<td>41</td>
<td>35</td>
<td>34</td>
<td>31</td>
</tr>
<tr>
<td>HP</td>
<td>12</td>
<td>61</td>
<td>60</td>
<td>40</td>
<td>33</td>
<td>33</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
<td>59</td>
<td>41</td>
<td>39</td>
<td>36</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>18</td>
<td>61</td>
<td>59</td>
<td>40</td>
<td>40</td>
<td>36</td>
<td>29</td>
<td>32</td>
</tr>
</tbody>
</table>

- Weak scaling, lattice size per GPU $24^3 \times 32$
- Run 3 times and get the best number

- Time breakdown when running with 1 GPU (SP, recon=8)
  - Dlsash time: 3.47(ms)
  - Exchange_walltime: 1.79(ms)
  - internal_kernel: 3.15(ms)
  - boundary_kernel: 0.30(ms)
  - Computation dominant, communicatin time hidden

- Time breakdown when running with 4 GPUs (SP, recon=8)
  - Dlsash time: 4.34(ms)
  - exchange_walltime: 3.94 (ms)
  - internal_kernel: 3.13(ms)
  - boundary_kernel: 0.32(ms)
  - Communication dominant, worsen when we go offnode, probably due to slow CPU, slow Infiniband, PCIe sharing within 2 gpus
CG performance (GFLOPS) per GPU on NERSC’s compute nodes (C2050, no ECC)

<table>
<thead>
<tr>
<th>reconstrct</th>
<th>1 GPU</th>
<th>2 GPUs</th>
<th>4 GPUs</th>
<th>8 GPUs</th>
<th>12 GPUs</th>
<th>16 GPUs</th>
<th>20 GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>12</td>
<td>24</td>
<td>23</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>41</td>
<td>41</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>12</td>
<td>96</td>
<td>94</td>
<td>93</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>86</td>
<td>83</td>
<td>83</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP</td>
<td>12</td>
<td>122</td>
<td>120</td>
<td>116</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>123</td>
<td>120</td>
<td>119</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>104</td>
<td>101</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Weak scaling, lattice size per GPU 24^3x32
- Run 3 times and get the best number

When running with 4 GPUs (SP, recon=8)
- Total dash time = 2.10 (ms)
- exchange_walltime = 1.19 (ms)
- internal_kernel = 1.92 (ms)
- boundary_kernel = 0.17 (ms)
- Total dash time is almost the same as the sum of two kernels. Communication time is completely hidden
Production Experience

- Have been using GPUs for electromagnetic effects, i.e., SU(3) × U(1)
- So far only using ensembles that fit in single GPU memory
- Have analyzed about 4,000 configurations from $20^3 \times 64$ to $28^3 \times 96$.

(Aaron Torok)
  - AC (NCSA), FNAL, Dirac (NERSC), Jlab

- A few constrains on Aaron Torok’s QED application
  - Double precision required
  - GPU mem requirement > 1 GB, only fits into Telsa C1060 → Lower frequency, lower performance
  - The long link is not unitary, only 18 reconstruct is applicable → Higher bandwidth requirement

- CPU: 6.04 node-hr=48.2 core-hr
- GPU: 1.49 node-hr=11.9 core-hr (only 1 core used)
“Electromagnetic Splitting of Charged and Neutral Mesons” (Lattice 2010, Aaron Torok, et al)
Where to get the Code

- Staggered code is integrated with Wilson code, soon to be released (QUDA 0.3) by BU
  - Can be found at http://lattice.bu.edu/quda
- Requires CUDA 3.0 or higher
- MultiGPU code to be released later
Future

- Study of heavy-light mesons might use GPUs. Need to combine both Clover and staggered inverters.
- Although we have asqtad code modules for gauge configuration generation, now generating HISQ configurations, so new code must be added.
- Investigate strong scaling as supercomputers now reaching for petaflop/s performance.
- Essential to decide what other parts of production running can be profitably shifted to GPUs
Acknowledgements

• This work is sponsored by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign (UIUC).

• Other Lattice QCD developers: Ron Balild (Boston University), Mike Clark (Harvard University), Balint Joo (Jefferson National Lab)

• We are thankful for the helpful discussions with Ray Sung and Wen-mei Hwu from ECE department, UIUC, Greg Bauer from NCSA, and John Stone from TCB group, UIUC.