The QUDA library for lattice QCD on GPUs

M A Clark, NVIDIA
Developer Technology Group
Outline

- Introduction to GPU Computing
- Lattice QCD
- QUDA: QCD on CUDA
- Supercomputing with QUDA
- Future Directions
- Summary
The March of GPUs

**Peak Double Precision FP**

- M1060
- Nehalem 3 GHz
- Westmere 3 GHz
- Fermi M2070
- Fermi+ M2090
- Kepler

**Peak Memory Bandwidth**

- M1060
- Fermi M2070
- Fermi+ M2090
- 8-core Sandy Bridge 3 GHz
- Kepler

Double Precision:
- NVIDIA GPU
- x86 CPU

Memory Bandwidth:
- NVIDIA GPU (ECC off)
- x86 CPU
Stunning Graphics Realism

Lush, Rich Worlds

Incredible Physics Effects

Core of the Definitive Gaming Platform
Tesla K20 Family: World’s Fastest Accelerator
>1TFlop Perf in under 225W

<table>
<thead>
<tr>
<th></th>
<th>Tesla K20X</th>
<th>Tesla K20</th>
</tr>
</thead>
<tbody>
<tr>
<td># CUDA Cores</td>
<td>2688</td>
<td>2496</td>
</tr>
<tr>
<td>Peak Double Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak DGEMM</td>
<td>1.32 TF</td>
<td>1.17 TF</td>
</tr>
<tr>
<td>Peak Single Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak SGEMM</td>
<td>1.22 TF</td>
<td>1.10 TF</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>250 GB/s</td>
<td>208 GB/s</td>
</tr>
<tr>
<td>Memory size</td>
<td>6 GB</td>
<td>5 GB</td>
</tr>
<tr>
<td>Total Board Power</td>
<td>235W</td>
<td>225W</td>
</tr>
</tbody>
</table>
The Kepler Architecture

- Kepler K20X
  - 2688 processing cores
  - 3995 SP Gflops peak (665.5 fma)
  - Effective SIMD width of 32 threads (warp)

- Deep memory hierarchy
  - As we move away from registers
    - Bandwidth decreases
    - Latency increases
  - Each level imposes a minimum arithmetic intensity to achieve peak

- Limited on-chip memory
  - 65,536 32-bit registers, 255 registers per thread
  - 48 KiB shared memory
  - 1.5 MiB L2
GPGPU Revolutionizes Computing

Latency Processor + Throughput processor

CPU + GPU
Low Latency or High Throughput?

**CPU**
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

**GPU**
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
Small Changes, Big Speed-up

Application Code

GPU

Use GPU to Parallelize

CPU

Compute-Intensive Functions

Rest of Sequential CPU Code
GPUs Accelerate Science

- Medical Imaging, U of Utah
- Molecular Dynamics, U of Illinois, Urbana
- Video Transcoding, Elemental Tech
- Matlab Computing, AccelerEyes
- Astrophysics, RIKEN

- Financial Simulation, Oxford
- Linear Algebra, Universidad Jaime
- 3D Ultrasound, Techniscan
- Quantum Chemistry, U of Illinois, Urbana
- Gene Sequencing, U of Maryland
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration

- OpenACC Directives
  - Easily Accelerate Applications

- Programming Languages
  - (C/C++, Fortran, Python, …)
  - Maximum Performance
GPU Accelerated Libraries
“Drop-in” Acceleration for your Applications

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL
- CULA|tools
- MAGMA
- NVIDIA cuFFT
- Rogue Wave Software
- CUSP
- IMSL Library
- Sparse Linear Algebra
- Building-block Algorithms
- C++ Templated Parallel Algorithms
OpenACC Directives

Program myscience
... serial code ...
$acc kernels
doo k = 1,n1
doo i = 1,n2
... parallel code ...
enddo
enddo
$acc end kernels
... End Program myscience

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs
GPU Programming Languages

- **Numerical analytics**: MATLAB, Mathematica, LabVIEW
- **Fortran**: OpenACC, CUDA Fortran
- **C**: OpenACC, CUDA C
- **C++**: Thrust, CUDA C++
- **Python**: PyCUDA, Copperhead
- **C#**: GPU.NET
void saxpy(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;

cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);

Anatomy of a CUDA Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements (GPU parallel functions are called Kernels)
Quantum Chromodynamics
Quantum Chromodynamics

- The strong force is one of the basic forces of nature (along with gravity, em and the weak force)

- It’s what binds together the quarks and gluons in the proton and the neutron (as well as hundreds of other particles seen in accelerator experiments)

- QCD is the theory of the strong force

- It’s a beautiful theory, lots of equations etc.

\[
\langle \Omega \rangle = \frac{1}{Z} \int [dU] e^{-\int d^4x L(U)} \Omega(U)
\]

...but...
Lattice Quantum Chromodynamics

- Theory is highly non-linear $\Rightarrow$ cannot solve directly
- Must resort to numerical methods to make predictions
- Lattice QCD
  - Discretize spacetime $\Rightarrow$ 4-d dimensional lattice of size $L_x \times L_y \times L_z \times L_t$
  - Finitize spacetime $\Rightarrow$ periodic boundary conditions
  - PDEs $\Rightarrow$ finite difference equations
- High-precision tool that allows physicists to explore the contents of nucleus from the comfort of their workstation (supercomputer)
- Consumer of 10-20% of North American supercomputer cycles
Steps in a lattice QCD calculation

1. Generate an ensemble of gluon field ("gauge") configurations
   - Produced in sequence, with hundreds needed per ensemble
   - Strong scaling required with $O(10-100 \text{ Tflops})$ sustained for several months (traditionally Crays, Blue Genes, etc.)
   - 50-90% of the runtime is in the linear solver
Steps in a lattice QCD calculation

2. “Analyze” the configurations
   - Can be farmed out, assuming $O(1 \text{ Tflops})$ per job.
   - 80-99% of the runtime is in the linear solver
     Task parallelism means that clusters reign supreme here

$$D_{ij}^{\alpha\beta}(x, y; U)\psi_j^\beta(y) = \eta_i^\alpha(x)$$

or “$Ax = b$”
Davies et al
QCD applications

• Some examples
  – MILC (FNAL, Indiana, Tuscon, Utah)
    • strict C, MPI only
  – CPS (Columbia, Brookhaven, Edinburgh)
    • C++ (but no templates), MPI and partially threaded
  – Chroma (Jefferson Laboratory, Edinburgh)
    • C++ expression-template programming, MPI and threads
  – BQCD (Berlin QCD)
    • F90, MPI and threads

• Each application consists of 100K-1M lines of code
• Porting each application not directly tractable
  – OpenACC possible for well-written code “Fortran-style” code (BQCD, maybe MILC)
Enter QUDA

- “QCD on CUDA” - [http://lattice.github.com/quda](http://lattice.github.com/quda)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, etc.
- Provides:
  - Various solvers for several discretizations, including multi-GPU support and domain-decomposed (Schwarz) preconditioners
  - Additional performance-critical routines needed for gauge field generation
- Maximize performance
  - Exploit physical symmetries
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Cache blocking
QUDA is community driven

- Developed on github
  - http://lattice.github.com/quda

- Open source, anyone can join the fun

- Contributors
  - Ron Babich (NVIDIA)
  - Kip Barros (LANL)
  - Rich Brower (Boston University)
  - Justin Foley (University of Utah)
  - Joel Giedt (Rensselaer Polytechnic Institute)
  - Steve Gottlieb (Indiana University)
  - Bálint Joó (Jlab)
  - Hyung-Jin Kim (BNL)
  - Claudio Rebbi (Boston University)
  - Guochun Shi (NCSA -> Google)
  - Alexei Strelchenko (FNAL)
  - Frank Winter (UoE -> Jlab)
USQCD software stack

(Many components developed under the DOE SciDAC program)
QUDA High-Level Interface

- QUDA default interface provides a simple view for the outside world
  - C or Fortran
  - Host applications simply pass cpu-side pointers
  - QUDA takes care of all field reordering and data copying
  - No GPU code in user application

- Limitations
  - No control over memory management
  - Data residency between QUDA calls not possible
  - QUDA might not support user application field order

```c
#include <quda.h>

int main() {
    // initialize the QUDA library
    initQuda(device);

    // load the gauge field
    loadGaugeQuda((void*)gauge, &gauge_param);

    // perform the linear solve
    invertQuda(spinorOut, spinorIn, &inv_param);

    // free the gauge field
    freeGaugeQuda();

    // finalize the QUDA library
    endQuda();
}
```
QUDA Mission Statement

• QUDA is
  – a library enabling legacy applications to run on GPUs
  – evolving
    • more features
    • cleaner, easier to maintain
  – a research tool into how to reach the exascale
    • Lessons learned are mostly (platform) agnostic
    • Domain-specific knowledge is key
    • Free from the restrictions of DSLs, e.g., multigrid in QDP
Solving the Dirac Equation

- Solving the Dirac Equation is the most time consuming operation in LQCD
  - First-order PDE acting on a vector field
  - On the lattice this becomes a large sparse matrix $M$
    - Radius 1 finite-difference stencil acting on a 4-d grid
    - Each grid point is a 12-component complex vector (spinor)
    - Between each grid point lies a 3x3 complex matrix (link matrix $\in$ SU(3))
  - Typically use Krylov solvers to solve $M x = b$
    - Performance-critical kernel is the SpMV
- Stencil application:
  - Load neighboring spinors, multiply by the inter-connecting link matrix, sum and store
Wilson Matrix

Dirac spin projector matrices

\( \text{(4x4 spin space)} \)

\( M_{x,x'} = -\frac{1}{2} \sum_{\mu=1}^{4} (P^{-\mu} \otimes U_{x}^{\mu} \delta_{x+\mu,x'} + P^{+\mu} \otimes U_{x-\mu}^{\mu} \delta_{x-\mu,x'}) + (4 + m) \delta_{x,x'} \)

\( \equiv -\frac{1}{2} D_{x,x'} + (4 + m) \delta_{x,x'} \)

\( m \) quark mass parameter

4d nearest-neighbor stencil operator acting on a vector field
Mapping the Wilson Dslash to CUDA

- Assign a single space-time point to each thread
  - \( V = \text{XYZT threads} \)
  - \( V = 24^4 \Rightarrow 3.3 \times 10^6 \) threads
  - Fine-grained parallelization
- Looping over direction each thread must
  - Load the neighboring spinor (24 numbers x8)
  - Load the color matrix connecting the sites (18 numbers x8)
  - Do the computation
  - Save the result (24 numbers)
- Arithmetic intensity
  - 1320 floating point operations per site
  - 1440 bytes per site (single precision)
  - 0.92 naive arithmetic intensity

\[
D_{x,x'} = \begin{pmatrix}
U_x & & & \\
& U_x & & \\
& & & U_x \\
& & & \\
U_x & & & \end{pmatrix}
\]

Tesla K20X

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gflops</td>
<td>3995</td>
</tr>
<tr>
<td>GB/s</td>
<td>250</td>
</tr>
<tr>
<td>AI</td>
<td>16</td>
</tr>
</tbody>
</table>

**bandwidth bound**
Field Ordering

- CPU codes tend to favor Array of Structures but these behave badly on GPUs

Threads read non-contiguous data

Spinor (24 numbers)

- GPUs (and AVX / Phi) like Structure of Arrays

Threads read contiguous data

- QUDA interface deals with all data reordering
- Application remains ignorant
Reducing Memory Traffic

- SU(3) matrices are all unitary complex matrices with det = 1
  - 12-number parameterization: reconstruct full matrix on the fly in registers

\[
\begin{pmatrix}
a_1 & a_2 & a_3 \\
b_1 & b_2 & b_3 \\
c_1 & c_2 & c_3 \\
\end{pmatrix}
\rightarrow
\begin{pmatrix}
a_1 & a_2 & a_3 \\
b_1 & b_2 & b_3 \\
c_1 & c_2 & c_3 \\
\end{pmatrix}
\]

- Additional 384 flops per site
- Also have an 8-number parameterization (requires sin/cos and sqrt)
  - Additional 856 flops per site
- Impose similarity transforms to increase sparsity
- Still memory bound - Can further reduce memory traffic by truncating the precision
  - Use 16-bit fixed-point representation
  - No loss in precision with mixed-precision solver
  - Almost a free lunch (small increase in iteration count)
Kepler Wilson-Dslash Performance

K20X Dslash performance
V = 24^3 x T
Wilson-Clover is ±10%
Krylov Solver Implementation

- Complete solver **must** be on GPU
  - Transfer $b$ to GPU (reorder)
  - Solve $Mx=b$
  - Transfer $x$ to CPU (reorder)
- Entire algorithms must run on GPUs
  - Time-critical kernel is the stencil application (SpMV)
  - Also require BLAS level-1 type operations
    - e.g., AXPY operations: $b += ax$, NORM operations: $c = (b,b)$
    - Roll our own kernels for kernel fusion and custom precision

```c
while (|r_k| > ε) {
    β_k = (r_k, r_k) / (r_{k-1}, r_{k-1})
    p_{k+1} = r_k - β_k p_k
    α = (r_k, r_k) / (p_{k+1}, A p_{k+1})
    r_{k+1} = r_k - α A p_{k+1}
    x_{k+1} = x_k + α p_{k+1}
    k = k + 1
}
```

conjugate gradient
Kepler Wilson-Solver Performance

K20X CG performance

\[ V = 24^3 \times T \]
Mixed-Precision Solvers

- Often require solver tolerance beyond limit of single precision
- But single and half precision much faster than double
- Use mixed precision
  - e.g. defect-correction

QUDA uses Reliable Updates (Sleijpen and Van der Worst 1996)
- Almost a free lunch
  - Small increase in iteration count

```plaintext
while (|r_k| > \varepsilon) {
  r_k = b - Ax_k
  solve Ap_k = r_k
  x_{k+1} = x_k + p_k
}
```
Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma
24^3 × 128 lattice
Relative Performance (Propagator) vs. E5-2687w 3.10 GHz Sandy Bridge

<table>
<thead>
<tr>
<th></th>
<th>1xCPU</th>
<th>2xCPU</th>
<th>K20X Single-Socket</th>
<th>2xCPU+1xGPU</th>
<th>2xCPU+2xGPU</th>
<th>M2090 Dual-Socket</th>
<th>2xCPU+1xGPU</th>
<th>2xCPU+2xGPU</th>
<th>K20X Dual-Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative to 2x CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xCPU</td>
<td>0.5</td>
<td>1.0</td>
<td></td>
<td>3.7</td>
<td>6.8</td>
<td>2.8</td>
<td>3.5</td>
<td>3.5</td>
<td>6.7</td>
</tr>
<tr>
<td>2xCPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Supercomputing with QUDA
The need for multiple GPUs

• Only yesterday’s lattice volumes fit on a single GPU
• More cost effective to build multi-GPU nodes
  • Better use of resources if parallelized
• Gauge generation requires strong scaling
  • 10-100 TFLOPS sustained solver performance
Supercomputing means GPUs

Tsubame 2.0, Tianhe 1A, Blue Waters, etc.
TITAN: World’s Most Efficient Supercomputer

18,688 Tesla K20X GPUs
27 Petaflops Peak, 17.59 Petaflops on Linpack
90% of Performance from GPUs
Multiple GPUs

- Many different mechanisms for controlling multiple GPUs
  - MPI processes
  - CPU threads
  - Multiple GPU per thread and do explicit switching
  - Combinations of the above
- QUDA uses the simplest: 1 GPU per MPI process
  - Allows partitioning over node with multiple devices and multiple nodes
  - `cudaSetDevice(local_mpi_rank);`
  - In the future likely will support many-to-one or threads
CUDA Stream API

• CUDA provides the stream API for concurrent work queues
  • Provides concurrent kernels and host<->device memcpys
  • Kernels and memcpys are queued to a stream
    • \texttt{kernel<<<block, thread, shared, streamId>>>(arguments)}
    • \texttt{cudaMemcpyAsync(dst, src, size, type, streamId)}
  • Each stream is an in-order execution queue
  • Must synchronize device to ensure consistency between streams
    • \texttt{cudaDeviceSynchronize()}
• QUDA uses the stream API to overlap communication of the halo region with computation on the interior
1D Lattice decomposition

1D decomposition (in ‘time’ direction)

Assign sub-lattice to GPU

face exchange

face exchange

face exchange

face exchange

wrap around

Friday, January 28, 2011
Multi-dimensional lattice decomposition
Multi-dimensional Ingredients

• Packing kernels
  – Boundary faces are not contiguous memory buffers
  – Need to pack data into contiguous buffers for communication
  – One for each dimension

• Interior dslash
  – Updates interior sites only

• Exterior dslash
  – Does final update with halo region from neighbouring GPU
  – One for each dimension
Multi-dimensional Kernel Computation

2-d example
- Checkerboard updating scheme employed, so only half of the sites are updated per application
  - Green: source sites
  - Purple: sites to be updated
  - Orange: site update complete
Multi-dimensional Kernel Computation

Step 1
- Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
Multi-dimensional Kernel Computation

Step 1

- Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
Step 1

- Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
Multi-dimensional Kernel Computation

Step 1

- Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
Multi-dimensional Kernel Computation

Step 2

An “interior kernel” updates all local sites to the extent possible. Sites along the boundary receive contributions from local neighbors.
Multi-dimensional Kernel Computation

Step 3

Boundary sites are updated by a series of kernels - one per direction.

A given boundary kernel must wait for its ghost zone to arrive.

Note in higher dimensions corner sites have a race condition - serialization of kernels required.
Multi-dimensional Kernel Computation

Step 3

Boundary sites are updated by a series of kernels - one per direction.

A given boundary kernel must wait for its ghost zone to arrive.

Note in higher dimensions corner sites have a race condition - serialization of kernels required.
Multi-dimensional Kernel Computation

Step 3

Boundary sites are updated by a series of kernels - one per direction.

A given boundary kernel must wait for its ghost zone to arrive

Note in higher dimensions corner sites have a race condition - serialization of kernels required
Multi-dimensional Kernel Computation

Step 3

Boundary sites are updated by a series of kernels - one per direction.

A given boundary kernel must wait for its ghost zone to arrive

Note in higher dimensions corner sites have a race condition - serialization of kernels required
has data dependency with each other and must be executed.

It is also clear from the above description that because of the spinors in corners, the exterior kernels source spinors. The ghost spinors from the Ti neighbor boundaries are computed in the exterior kernel for T dimension using the ghost spinor and further improves the performance. The interior kernel computes any contributions to the boundary spinors that does not involve with ghost spinors. The interior kernel computes the space contribution for this spinor as well as the negative T direction's contribution for this spinor. The positive T direction's contribution for this spinor will be computed in the exterior kernel for T dimension using the ghost spinor and further improves the performance.

The use of memory padding avoids the GPU memory partition camping problem and further improves the performance. However, in the X, Y, Z exterior kernels, the ghost spinor share ghost gauge fields from the Ti neighbor boundaries. Since spinors in the corners belong to multiple boundaries, for the interior kernel so that it computes the full results for the interior spinors and the partial results for spinors in the boundary-update kernels are coalesced to the extent possible. The memory accesses in both interior and exterior kernels are coalesced to the extent possible that memory accesses in both interior and exterior kernels are coalesced to the extent possible.

The two host memcpy are required due to the fact that memory and the GPU direct technology is not readily available in the existing GPU clusters. We expect these extra memory and the GPU direct technology is not readily available in the existing GPU clusters. The two host memcpy are required due to the fact that memory access in the communication is likely to improve the performance. When communicating over multiple dimensions, the communication cost dominates the computations and any reduction of the communication components is likely to exceed the interior kernel run time. One extra stream is used for interior and exterior kernels, but the different data mapping makes the different mapping schemes but the different mapping schemes make the different mapping schemes.

CUDA streams are extensively used to overlap computational tasks with the communication. The accumulation of communication over multiple dimensions is likely to exceed the interior kernel run time. The gathering and exchanging of spinors in the ghost data, the interior kernel and other exterior kernels are computed in the same stream and are synchronized with the communication in the corresponding dimension. Therefore, the communication and data transfer are ordered on the GPU so as to enable the use of CUDA streams. One extra stream is used for interior and exterior kernels. The gather kernels for all directions are launched in Fig. 4. The gather kernels for all directions are launched. One extra stream is used for interior and exterior kernels. The gather kernels for all directions are launched.

Communications Pipeline

Multi-dimensional CUDA SDK has an interesting GPU to GPU direct memory copy. The recent availability of CUDA SDK allows host and GPU data copies to be removed in the future when better support is made available in the existing GPU clusters. We expect these extra memory and the GPU direct technology is not readily available in the existing GPU clusters. The two host memcpy are required due to the fact that memory access in the communication is likely to improve the performance. When communicating over multiple dimensions, the communication cost dominates the computations and any reduction of the communication components is likely to exceed the interior kernel run time. One extra stream is used for interior and exterior kernels, but the different data mapping makes the different mapping schemes but the different mapping schemes make the different mapping schemes.

CUDA streams are extensively used to overlap computational tasks with the communication. The accumulation of communication over multiple dimensions is likely to exceed the interior kernel run time. The gathering and exchanging of spinors in the ghost data, the interior kernel and other exterior kernels are computed in the same stream and are synchronized with the communication in the corresponding dimension. Therefore, the communication and data transfer are ordered on the GPU so as to enable the use of CUDA streams. One extra stream is used for interior and exterior kernels. The gather kernels for all directions are launched in Fig. 4. The gather kernels for all directions are launched. One extra stream is used for interior and exterior kernels. The gather kernels for all directions are launched.
Results from TitanDev
- Clover propagator
- $48^3 \times 512$ aniso clover
- scaling up 768 GPUs
Results from TitanDev
- Clover propagator
- $48^3 \times 512$ aniso clover
- scaling up 768 GPUs
Results from TitanDev
- Clover propagator
- $48^3 \times 512$ aniso clover
- scaling up 768 GPUs

Strong Scaling: $48^3 \times 512$ Lattice (Weak Field), Chroma + QUDA

100 Tflops

Tflops Sustained vs. Interlagos Sockets (16 core/socket)
Domain Decomposition

- Non-overlapping blocks - simply have to switch off inter-GPU communication
- Preconditioner is a gross approximation
  - Use an iterative solver to solve each domain system
  - Require only 10 iterations of domain solver □ 16-bit
  - Need to use a flexible solver □ GCR
- Block-diagonal preconditioner impose $\lambda$ cutoff
- Smaller blocks lose low frequency modes
  - keep wavelengths of $\sim O(\Lambda_{QCD}^{-1})$, $\Lambda_{QCD}^{-1} \sim 1$fm
- Aniso clover: ($a_s=0.125$fm, $a_t=0.035$fm) □ $8^3\times32$ blocks are ideal
  - $48^3\times512$ lattice: $8^3\times32$ blocks □ 3456 GPUs
Results from TitanDev
- 48^3 x 512 aniso clover
- scaling up 768 GPUs
Results from TitanDev
- $48^3 \times 512$ aniso clover
- scaling up 768 GPUs
Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma
48^3\times 512 lattice
Relative Scaling (Application Time)

“XK7” node = XK7 (1x K20X + 1x Interlagos)
“XE6” node = XE6 (2x Interlagos)

3.58x vs. XE6 @1152 nodes
Clover Propagator Benchmark on Titan: Strong Scaling, QUDA+Chroma+QDP-JIT(PTX)

- BiCGStab: $72^3 \times 256$
- DD+GCR: $72^3 \times 256$
- BiCGStab: $96^3 \times 256$
- DD+GCR: $96^3 \times 256$

B. Joo, F. Winter (JLab), M. Clark (NVIDIA)
HISQ RHMC with QUDA

<table>
<thead>
<tr>
<th>Routine</th>
<th>Single</th>
<th>Double</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-shift solver</td>
<td>156.5</td>
<td>77.1</td>
<td>157.4</td>
</tr>
<tr>
<td>Fermion force</td>
<td>191.2</td>
<td>97.2</td>
<td></td>
</tr>
<tr>
<td>Fat link generation</td>
<td>170.7</td>
<td>82.0</td>
<td></td>
</tr>
<tr>
<td>Gauge force</td>
<td>194.8</td>
<td>98.3</td>
<td></td>
</tr>
</tbody>
</table>

Absolute performance (36^4 lattice)

QUDA vs. MILC (24^364)
MILC on QUDA

- Gauge generation on 256 BW nodes
  - Volume = 96$^3 \times 192$
  - QUDA: solver, forces, fat link
  - MILC: long link, momentum exp.

- MILC is multi-process only
  - 1 GPU per process
  - 4x net gain in performance
  - But potential >5x gain in performance
    - Porting remaining functions
    - or
    - Fix host code to run in parallel

(Mixed-) Double-precision Rational HMC on a 96$^3 \times 192$ lattice on Titan

Justin Foley, University of Utah

Many-GPU calculations in Lattice QCD
MILC on QUDA

Preliminary strong scaling on Titan (V = 96^3x192)
Future Directions
Future Directions

• LQCD coverage (avoiding Amdahl)
  – Remaining components needed for gauge generation
  – Contractions
  – Eigenvector solvers

• Solvers
  – Scalability
  – Optimal solvers (e.g., adaptive multigrid)

• Performance
  - Locality
  - Learning from today’s lessons (software and hardware)
QUDA - Chroma Integration

- Chroma is built on top of QDP++
  - QDP++ is a DSL of data-parallel building blocks
  - C++ expression-template approach
- QUDA only accelerates the linear solver
- QDP/JIT is a project to port QDP++ directly to GPUs (Frank Winter)
  - Generates ptx kernels at run time
  - Kernels are JIT compiled and cached for later use
  - Chroma runs unaltered on GPUs
- QUDA has low-level hooks for QDP/JIT
  - Common GPU memory pool
  - QUDA accelerates time-critical routines
  - QDP/JIT takes care of Amdahl
Exploiting Locality
Wilson SP Dslash Performance with GPU generation

Gflops sustained

- Spatial locality
- Temporal locality

Naïve
Actual

G80
GT200
Fermi
Kepler
Maxwell

illustrative
illustrative
Future Directions - Communication

• Only scratched the surface of domain-decomposition algorithms
  – Disjoint additive
  – Overlapping additive
  – Alternating boundary conditions
  – Random boundary conditions
  – Multiplicative Schwarz
  – Precision truncation
Future Directions - Latency

• Global sums are bad
  – Global synchronizations
  – Performance fluctuations

• New algorithms are required
  – S-step CG / BiCGstab, etc.
  – E.g., Pipeline CG vs. Naive

• One-sided communication
  – MPI-3 expands one-sided communications
  – Cray Gemini has hardware support
  – Asynchronous algorithms?
    • Random Schwarz has exponential convergence

![Graph showing performance comparison between Naive CG and Pipeline CG](image)
Future Directions - Precision

• Mixed-precision methods have become de facto
  – Mixed-precision Krylov solvers
  – Low-precision preconditioners

• Exploit closer coupling of precision and algorithm
  – Domain decomposition, Adaptive Multigrid
  – Hierarchical-precision algorithms
  – 128-bit <-> 64-bit <-> 32-bit <-> 16-bit <-> 8-bit

• Low precision is lossy compression
• Low-precision tolerance is fault tolerance
Summary

• Introduction to GPU Computing and LQCD computation
• Glimpse into the QUDA library
  – Exploiting domain knowledge to achieve high performance
  – Mixed-precision methods
  – Communication reduction at the expense of computation
  – Enables legacy QCD applications ready for accelerators
• GPU Supercomputing is here now
  – Algorithmic innovation may be required
  – Today’s lessons are relevant for Exascale
Backup slides
QUDA Interface Extensions

• Allow QUDA interface to accept GPU pointers
  – First natural extension
  – Remove unnecessary PCIe communications between QUDA function calls

• Allow user-defined functors for handling field ordering
  – User only has to specify their field order
  – Made possible with device libraries (CUDA 5.0)

• Limitations
  – Limited control of memory management
  – Requires deeper application integration
**QUDA Low-Level Interface** (in development)

- Possible strawman under consideration

```c
lat = QUDA_new_lattice(dims, ndim, lat_param);
source = QUDA_new_site_field(lat, spinor_param);
solution = QUDA_new_site_field(lat, spinor_param);
QUDA_load_link_field(u, host_u, gauge_order);
QUDA_load_site_field(source, host_source, spinor_order);
QUDA_solve(source, u, solver);
QUDA_save_site_field(source, host_solution, spinor_order);
QUDA_destroy_site_field(source);
```

- Here, src, sol, etc. are opaque objects that know about the GPU
- Allows the user to easily maintain data residency
- Users can easily provide their own kernels
- High-level interface becomes a compatibility layer built on top
QUDA - Chroma Integration

- Chroma is built on top of QDP++
  - QDP++ is a DSL of data-parallel building blocks
  - C++ expression-template approach
- QDP/JIT is a project to port QDP++ directly to GPUs (Frank Winter)
  - Generates ptx kernels at run time
  - Kernels are JIT compiled and cached for later use
  - Chroma runs unaltered on GPUs
- QUDA has low-level hooks for QDP/JIT
  - Common GPU memory pool
  - QUDA accelerates time-critical routines
  - QDP/JIT takes care of Amdahl
Low Latency or High Throughput?

- **CPU** architecture must **minimize latency** within each thread
- **GPU** architecture **hides latency** with computation from other thread warps

![Diagram showing GPU Stream Multiprocessor and CPU core with T1 through T4 and W1 through W4 showing processing, waiting, and context switch phases.](image-url)
Memory Coalescing

- To achieve maximum bandwidth, threads within a warp must read from consecutive regions of memory.
  - Each thread can load 32-bit, 64-bit or 128-bit words.
  - CUDA provides built-in vector types.

<table>
<thead>
<tr>
<th>type</th>
<th>32-bit</th>
<th>64-bit</th>
<th>128-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>int2</td>
<td>int4</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float2</td>
<td>float4</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>double</td>
<td>double2</td>
</tr>
<tr>
<td>char</td>
<td>char4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>short2</td>
<td></td>
<td>short4</td>
</tr>
</tbody>
</table>
Run-time autotuning

- **Motivation:**
  - Kernel performance (but not output) strongly dependent on launch parameters:
    - `gridDim` (trading off with work per thread), `blockDim`
    - `blocks/SM` (controlled by over-allocating shared memory)

- **Design objectives:**
  - Tune launch parameters for all performance-critical kernels at run-time as needed (on first launch).
  - Cache optimal parameters in memory between launches.
  - Optionally cache parameters to disk between runs.
  - Preserve correctness.
Auto-tuned “warp-throttling”

- Motivation: Increase reuse in limited L2 cache.
Run-time autotuning: Implementation

- Parameters stored in a global cache:
  
  ```cpp
  static std::map<TuneKey, TuneParam> tunecache;
  ```

- **TuneKey** is a struct of strings specifying the kernel name, lattice volume, etc.

- **TuneParam** is a struct specifying the tune blockDim, gridDim, etc.

- Kernels get wrapped in a child class of **Tunable** (next slide)

- **tuneLaunch()** searches the cache and tunes if not found:
  
  ```cpp
  TuneParam tuneLaunch(Tunable &tunable, QudaTune enabled, QudaVerbosity verbosity);
  ```
Run-time autotuning: Usage

- **Before:**
  
  \[\text{myKernelWrapper}(a, \ b, \ c);\]

- **After:**
  
  \[\text{MyKernelWrapper}\ *k = \text{new MyKernelWrapper}(a, \ b, \ c);\]
  \[k->\text{apply}(); \quad // \quad <\-- \text{automatically tunes if necessary} \]

- **Here** \text{MyKernelWrapper} \ inherits from \text{Tunable} \ and \ optionally \ overloads \ various \ virtual \ member \ functions \ (next \ slide).

- **Wrapping related kernels in a class hierarchy is often useful anyway, independent of tuning.**
Virtual member functions of Tunable

- Invoke the kernel (tuning if necessary):
  - apply()

- Save and restore state before/after tuning:
  - preTune(), postTune()

- Advance to next set of trial parameters in the tuning:
  - advanceGridDim(), advanceBlockDim(), advanceSharedBytes()
  - advanceTuneParam() // simply calls the above by default

- Performance reporting
  - flops(), bytes(), perfString()

- etc.
Domain Decomposition

(Re)Start

Apply Preconditioner: reduced precision inner solve

Generate Subspace

\[ \hat{r}_k = \hat{K}^{-1} \hat{r}_k \]
\[ \hat{z}_k = \hat{M} \hat{p}_k \]
\[ \beta_{i,k} = (\hat{z}_i, \hat{z}_k) \]
\[ \gamma_k = ||\hat{z}_k|| \]
\[ \alpha_k = (\hat{z}_k, \hat{r}_k) \]
\[ \hat{r}_{k+1} = \hat{r}_k - \alpha_k \hat{z}_k \]
\[ k = k + 1 \]

Orthogonalize \( \hat{z}_s \)

Update Solution

Reduced Precision Solve for \( y \):
\[ \gamma l\chi_l \sum_{i=l+1}^{k} \beta_{i,l} \chi_l = \alpha_l \]

Compute correction for \( x \):
\[ \hat{x} = \sum_{i=0}^{k-1} \chi_i p_i \]
\[ x = x + \hat{x} \]

Quantities with \(^\wedge\) are in reduced precision

repeat for all \( k \) or until residuum drops

Full precision restart if not converged
Future Directions - Locality

• Where locality does not exist, let’s create it
  – E.g., Multi-source solvers
  – Staggered Dslash performance, K20X
  – Transform a memory-bound into a cache-bound problem
  – Entire solver will remain bandwidth bound
The High Cost of Data Movement

Fetching operands costs more than computing on them.

- 64-bit DP: 20 pJ
- 256-bit access: 8 kB SRAM, 50 pJ
- DRAM Rd/Wr: 500 pJ
- Efficient off-chip link: 1 nJ