Lattice QCD Algorithms at the Exascale

M. Clark, NVIDIA
Contents

• 1 minute introduction to GPUs
• 2 minute introduction to Lattice QCD
• QUDA Library
  – Solver Algorithms
• Current Research
  – Adaptive Multigrid
  – Abstracting algorithms from architecture
• Future Work
The March of GPUs

Peak Double Precision FP

- M1060
- Nehalem 3 GHz
- Westmere 3 GHz
- Sandy Bridge 3 GHz
- Ivy Bridge 3 GHz

Peak Memory Bandwidth

- M1060
- Westmere 3 GHz
- Sandy Bridge 3 GHz
- Ivy Bridge 3 GHz

Gflops/s

GB/s

NVIDIA GPU (ECC off)

x86 CPU
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 (public) 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
   - 50-90% of the runtime is in the linear solver

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 \[ "Ax = b" \]

\[ U_\mu(x) \]
QCD applications

• Some examples
  – MILC (FNAL, Indiana, Arizona, Utah)
    • strict C, MPI only
  – CPS (Columbia, BNL, Edinburgh)
    • C++ (but no templates), MPI and partially threaded
  – Chroma (Jlab, 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)
Enter QUDA

• “QCD on CUDA” - 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 all major fermionic discretizations, with multi-GPU support
  — Additional performance-critical routines needed for gauge-field generation
• Maximize performance
  – Exploit physical symmetries to minimize memory traffic
  – Mixed-precision methods
  – Autotuning for high performance on all CUDA-capable architectures
    • Branched and used elsewhere
  – Cache blocking
  – Domain-decomposed (Schwarz) preconditioners for strong scaling
QUDA is community driven

- Ron Babich (NVIDIA)
- Kip Barros (LANL)
- Rich Brower (Boston University)
- Michael Cheng (Boston University)
- MAC (NVIDIA)
- Justin Foley (University of Utah)
- Joel Giedt (Rensselaer Polytechnic Institute)
- Steve Gottlieb (Indiana University)
- Bálint Joó (Jlab)
- Hyung-Jin Kim (BNL)
- Jian Liang (IHEP)
- Claudio Rebbi (Boston University)
- Guochun Shi (NCSA -> Google)
- Alexei Strelchenko (Cyprus Institute -> FNAL)
- Alejandro Vaquero (Cyprus Institute)
- Frank Winter (UoE -> Jlab)
- Yibo Yang (IHEP)
Mapping the Dslash to CUDA

- Finite difference operator in LQCD is known as Dslash
  - QUDA implements 11 different discretization variants
- Assign a single space-time point to each thread
  - \( V = XYZT \) threads, e.g., \( V = 24^4 \to 3.3 \times 10^6 \) threads
  - Fine-grained parallelization
  - Each thread has (Wilson Dslash) 0.92 naive arithmetic intensity
- QUDA exploits domain knowledge to reduce memory traffic
  - Exact SU(3) matrix compression (18 \to 12 \text{ or } 8 \text{ real numbers})
  - Similarity transforms to increase operator sparsity
  - Use 16-bit fixed-point representation
    - No loss in precision with mixed-precision solver
    - Almost a free lunch (small increase in iteration count)

\[
D_{x,x'} = \begin{pmatrix}
  1 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 1
\end{pmatrix}
\]

**Tesla K20X**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</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>
Kepler Wilson-Dslash Performance

\[ V = 24^3 \times T \text{ K20X Dslash} \]
Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma
24³x128 lattice
Relative Performance (Propagator) vs. E5-2687w 3.10 GHz Sandy Bridge

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Relative to 2x CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xCPU</td>
<td>0.5</td>
</tr>
<tr>
<td>2xCPU</td>
<td>1.0</td>
</tr>
<tr>
<td>1xCPU+1xGPU</td>
<td>3.7</td>
</tr>
<tr>
<td>1xCPU+2xGPU</td>
<td>6.8</td>
</tr>
<tr>
<td>2xCPU+1xGPU</td>
<td>2.8</td>
</tr>
<tr>
<td>2xCPU+2xGPU</td>
<td>5.5</td>
</tr>
<tr>
<td>2xCPU+1xGPU</td>
<td>3.5</td>
</tr>
<tr>
<td>2xCPU+2xGPU</td>
<td>6.7</td>
</tr>
</tbody>
</table>

CPU
Single-Socket

K20X
Dual-Socket

M2090

Domain Decomposition

- Reduce inter-node communication and synchronization
- Utilize domain-decomposition techniques, e.g., Additive Schwarz
  - Non-overlapping blocks - simply switch off inter-node 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 λ cutoff
  - Limits scalability of algorithm
  - In practice, non-preconditioned part becomes source of Amdahl, limiting scalability
Chroma (Lattice QCD) – High Energy & Nuclear Physics

**Chroma**

48\(^3\)x512 lattice

Relative Scaling (Application Time)

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

“XE6” node = XE6 (2x Interlagos)

---

![Graph showing relative scaling for different nodes](image)
Current Research
Adaptive Geometric Multigrid

Osborn et al, arXiv:1011.2775

- Adaptively find candidate null-space vectors
  - Dynamically learn the null space and use this to define the prolongator
  - Algorithm is self learning
- Optimal algorithm
  - Linear scaling with V
  - Insensitive to condition number
The Challenge of Multigrid on GPU

- For competitiveness, MG on GPU is a must
- GPU requirements very different from CPU
  - Each thread is slow, but $O(10,000)$ threads per GPU
- Fine grids run very efficiently
  - High parallel throughput problem
- Coarse grids are worst possible scenario
  - More cores than degrees of freedom
  - Increasingly serial and latency bound
  - Little’s law ($\text{bytes} = \text{bandwidth} \times \text{latency}$)
  - Amdahl’s law limiter
Hierarchical algorithms on heterogeneous architectures

GPU
Thousands of cores for parallel processing

CPU
Few Cores optimized for serial work
Design Goals

• Flexibility
  – Deploy level $i$ on either CPU or GPU
  – All algorithmic flow decisions made at runtime
  – Autotune for a given heterogeneous architecture

• (Short term) Provide optimal solvers to legacy apps
  – e.g., Chroma, CPS, MILC, etc.

• (Long term) Hierarchical algorithm toolbox
  – Little to no barrier to trying new algorithms
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity

**Diagram**: Diagram showing the hierarchy of fields and algorithms, including:
- LatticeField
  - ColorSpinorField
    - cudaColorSpinorField
    - cpuColorSpinorField
  - GaugeField
    - cudaGaugeField
    - cpuGaugeField
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity

Diagram:
- LatticeField
  - ColorSpinorField
    - cudaColorSpinorField
    - cpuColorSpinorField
  - GaugeField
    - cudaGaugeField
    - cpuGaugeField
- Architecture
Writing the same code for two architectures

- Use C++ templates to abstract arch specifics
  - Load/store order, caching modifiers, precision, intrinsics
- CPU and GPU almost identical
  - Index computation (for loop -> thread id)
  - Block reductions (shared memory reduction and / or atomic operations)

```cpp
// CPU: OpenMP, vectorization

template<...> void fooCPU(Arg &arg) {
    arg.sum = 0.0;
    #pragma omp for
    for (int x=0; x<size; x++)
        arg.sum += bar<...>(arg, x);
}

// GPU: shared memory

template<...> __global__ void fooGPU(Arg arg) {
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    real sum = bar<...>(arg, tid);
    __shared__ typename BlockReduce::TempStorage tmp;
    arg.sum = cub::DeviceReduce<...>(tmp).Sum(sum);
}
```

```cpp
// platform specific parallelization here

platform specific load/store here:
field order, cache modifiers, textures

platform specific parallelization here
99% of code goes here
```

```
platform independent stuff goes here

99% of code goes here
```

```cpp
// platform independent stuff goes here

99% of code goes here

```

```cpp
// platform specific load/store here:

field order, cache modifiers, textures

```

```
CPU

```

```
GPU

```
Current Status

- First multigrid solver working in QUDA
- Some components still on CPU only

<table>
<thead>
<tr>
<th></th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine grid operator</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Block Orthogonalization</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Prolongator</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Restrictor</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Construct coarse gauge field</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Coarse grid operator</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Vector BLAS</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Designed to interoperate with J. Osborn’s *qopqdp* implementation
  - Can verify algorithm correctness, and share null space vectors
Future Directions
Scalability

- Only scratched the surface of domain-decomposition algorithms
  - Overlapping blocks
  - Alternating boundary conditions
  - Multiplicative Schwarz
  - Precision truncation

- Global sums are source of Amdahl
  - New algorithms are required
  - S-step CG / BiCGstab, etc.

- One-sided communication
  - MPI-3 expands one-sided communications
  - Cray (and others) have hardware support
  - Ultimate goal: asynchronous solver algorithms?
QUDA as a Hierarchical Algorithm Tool

• Lots of interesting questions to be explored
• Exploit closer coupling of precision and algorithm
  – QUDA designed for complete run-time specification of precision at any point in the algorithm
  – Currently supports 64-bit, 32-bit, 16-bit
  – Is 128-bit or 8-bit useful at all for hierarchical algorithms?
    • long double observed to reduce solver iterations on x86
• Domain-decomposition (DD) and multigrid
  – DD approaches likely vital for strong scaling
  – DD solvers are effective for high-frequency dampening
  – Overlapping domains likely more important at coarser scales
Summary

• Introduction to QUADA library
• Production library for GPU-accelerated LQCD
  – Scalable linear solvers
  – Coverage for most LQCD algorithms
• Current research efforts focused on adaptive multigrid algorithms
  – Most of the nitty gritty details worked out
  – Now time for fun
• Hierarchical and heterogeneous algorithm research toolbox
  – Hope for scalability and optimality
• Lessons today are relevant for Exascale preparation
Backup slides
The Need for Just-In-Time Compilation

• Tightly-coupled variables should be at the register level
• Dynamic indexing cannot be resolved in register variables
  – Array values with indices not known at compile time spill out into global memory (L1 / L2 / DRAM)

```cpp
template <typename ProlongateArg>
__global__ void prolongate(ProlongateArg arg, int Nspin, int Ncolor) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    for (int s=0; s<Nspin; s++) {
        for (int c=0; c<Ncolor; c++) {
            ...
        }
    }
}
```
The Need for Just-In-Time Compilation

• Possible solutions
  – Template over every possible $N_v \otimes$ precision for each MG kernel
  – One thread per colour matrix row (inefficient for $N_v \mod 32 \neq 0$)
  – Only compile necessary kernel at runtime

```cpp
template <typename ProlongateArg, int Ncolor, int Nspin>
__global__ void prolongate(ProlongateArg arg) {
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  for (int s=0; s<Nspin; s++) {
    for (int c=0; c<Ncolor; c++) {
      ...
    }
  }
}
```

• JIT support will be coming in CUDA x.y
  – Final performant implementation will likely require this
Heterogeneous Updating Scheme

- Multiplicative MG is necessarily serial process
  - Cannot utilize both GPU and CPU simultaneously

Additive MG is parallel
- Can utilize both GPU and CPU simultaneously
- Additive MG requires accurate coarse-grid solution
- Not amenable to multi-level
- Only need additive correction at CPU<->GPU level interface

Accurate coarse grid solution maybe cheaper than serialization / synchronization
Heterogeneous Updating Scheme

- Multiplicative MG is necessarily a serial process
  - Cannot utilize both GPU and CPU simultaneously
- Additive MG is parallel
  - Can utilize both GPU and CPU simultaneously
- Additive MG requires accurate coarse-grid solution
  - Not amenable to multi-level
  - Only need additive correction at CPU<->GPU level interface
- Accurate coarse grid solution maybe cheaper than serialization / synchronization
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:**
  ```cpp
  myKernelWrapper(a, b, c);
  ```

- **After:**
  ```cpp
  MyKernelWrapper *k = new MyKernelWrapper(a, b, c);
  k->apply(); // <-- automatically tunes if necessary
  ```

- Here *MyKernelWrapper* inherits from 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.