HIERARCHICAL ALGORITHMS ON HETEROGENEOUS ARCHITECTURES:
ADAPTIVE MULTIGRID SOLVERS FOR LQCD ON GPUs

M Clark
NVIDIA
with R Brower and M Cheng
Contents

- Introduction to Lattice QCD
- QUDA Library
- Multigrid on Heterogeneous Architectures
- Summary
Quantum Chromodynamics

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

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

...but
Lattice Quantum Chromodynamics

- Theory is highly non-linear ⇒ cannot solve directly
- Must resort to numerical methods to make predictions
- Lattice QCD
  - Discretize spacetime ⇒ 4-d dimensional lattice of size $L_x \times L_y \times L_z \times L_t$
  - Finitize spacetime ⇒ periodic boundary conditions
  - PDEs ⇒ 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
- 90% of the computational cost is spent in the sparse linear solver

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

or "Ax = b"
LQCD 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
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 / Minimize time to science
  – Exploit physical symmetries to minimize memory traffic
  – Mixed-precision methods
  – Autotuning for high performance on all CUDA-capable architectures
  – Domain-decomposed (Schwarz) preconditioners for strong scaling
  – Multigrid solvers for optimal convergence  new!
The Dirac Operator

- Quark interactions are described by the Dirac operator
  - First-order PDE acting with a background field
  - Large sparse matrix

- 4-d nearest neighbor stencil operator acting on a vector field

- For solving $Ax=b$, use an iterative linear solver
  - CG, BiCGstab, etc.

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

\[
\equiv -\frac{1}{2} D_{x,x'} + \left( 4 + m + A_{x} \right) \delta_{x,x'}
\]

$D_{x,x'}$ is the clover matrix

$P_{\mu}$ is the Dirac spin projector matrices

$SU(3)$ QCD gauge field (link matrices) (3x3 color space)

$A$ is the clover matrix (12x12 spin $\otimes$ color space)

$m$ quark mass parameter
Mapping the Dirac operator to CUDA

- Finite difference operator in LQCD is known as Dslash
- Assign a single space-time point to each thread
  - $V = XYZT$ threads, e.g., $V = 24^4 \Rightarrow 3.3 \times 10^6$ threads
- 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)
- QUDA reduces memory traffic
  - Exact SU(3) matrix compression (18 $\Rightarrow$ 12 or 8 real numbers)
  - Similarity transforms to increase operator sparsity
  - Use 16-bit fixed-point representation
    - No loss in precision with mixed-precision solver
- Use CUDA streams and MPI for multi-GPU


**Chroma**

48³x512 lattice

Relative Scaling (Application Time)

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

“XE6” node = XE6 (2x Interlagos)

- XK7 (K20X) (DD+GCR)
- XK7 (K20X) (BiCGStab)
- XE6 (2x Interlagos)

3.58x vs. XE6 @1152 nodes
Adaptive Geometric Multigrid

Osborn et al 2011
Introduction to Multigrid

- Preconditioner is a gross approximation
- Stationary iterative solvers effective on high frequency errors
- Minimal effect on low frequency error
- Example
  - Free Laplace operator in 2d
  - $Ax = 0$, $x_0 = \text{random}$
  - Gauss Seidel relaxation
  - Plot error $e_i = -x_i$
Introduction to Multigrid

- Low frequency error modes are smooth
- Can accurately represent on coarse grid

- Low frequency on fine $\Rightarrow$ high frequency on coarse
- Relaxation effective again on coarse grid
- Interpolate back to fine grid
Multigrid V-cycle

• Solve
  1. Smooth
  2. Compute residual
  3. Restrict residual
  4. Recurse on coarse problem
  5. Prolongate correction
  6. Smooth
  7. If not converged, goto 1

- Multigrid has optimal scaling
  - \( O(N) \) Linear scaling with problem size
  - Convergence rate independent of condition number
Adaptive Geometric Multigrid

![Graph showing Dirac operator applications across different mass values and geometric multigrid methods. The graph includes lines for different vector counts: 20, 240 vectors, and Babich et al 2010 notation.](image)
Motivation

- A CPU running the optimal algorithm surpasses a highly tuned GPU sub-optimal algorithm
- For competitiveness, MG on GPU is a must
- Seek multiplicative gain of architecture and algorithm
The Challenge of Multigrid on GPU

• 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
• Multigrid decomposes problem into throughput and latency parts
Hierarchical algorithms on heterogeneous architectures

- GPUs: Thousands of cores for parallel processing
- CPUs: Few Cores optimized for serial work
Design Goals

• Performance
  – LQCD typically reaches high % peak peak performance
  – Brute force can beat the best algorithm

• 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
Ingredients for Parallel Adaptive Multigrid

- Prolongation construction (setup)
  - Batched QR decomposition
- Smoothing (relaxation on a given grid)
  - Repurpose the domain-decomposition preconditioner
- Prolongation
  - one-to-many mapping
- Restriction
  - many-to-one mapping
- Coarse Operator construction (setup)
  - Batched (small) dense matrix multiplication
- Coarse grid solver
  - direct solve on coarse grid
- Fine vs. Coarse grid parallelization
  - Coarse grid points have limited thread-level parallelism
  - Highly desirable to parallelize over fine grid points where possible
- Parallel multigrid uses common parallel primitives
  - Reduce, sort, etc.
  - Use CUB parallel primitives for high performance
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
  – CPU and GPU kernels call the same functions
  – Index computation (for loop -> thread id)
  – Block reductions (shared memory reduction and / or atomic operations)
Writing the same code for two architectures

template<...> __host__ __device__ Real bar(Arg &arg, int x) {
    // do platform independent stuff here
    complex<Real> a[arg.length];
    arg.A.load(a);
    ...
    // do computation
    arg.A.save(a);
    return norm(a);
}

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

platform independent stuff goes here
99% of computation goes here

CPU

GPU
Current Status

- Adaptive multigrid algorithm fully numerically verified
  - Consistent with previous results (Babich et al 2010)
- Framework still slow
  - Host code not optimized at all
  - GPU <-> CPU transfers not optimal
  - Optimal code requires heavy degree of templating (compilation and link time is increasingly a problem)
- Early observations
  - Using 16-bit precision for smoothing does not affect convergence
  - Coarse-grid solve can be poorly conditioned thus requiring single precision
Next Steps

- Optimize
  - E.g., kernel fusion, CPU OpenMP/vectorization
  - read/write directly to/from CPU memory
- Strong scaling
- Algorithm research
  - Precision investigation
  - Coarse-grid solvers (direct vs. indirect)
  - Investigate different update strategies:
    - multiplicative vs heterogeneous

• Real goal is developing asynchronous solvers for future heterogeneous architectures
Heterogeneous Computing in 2016

TESLA GPU

NVLink 80 GB/s

Stacked Memory

HBM 1 Terabyte/s

CPU

DDR4 50-75 GB/s

DDR Memory
Summary

• Introduction to QUDA library
• Production library for GPU-accelerated LQCD
  – Scalable linear solvers
  – Coverage for most LQCD algorithms
• Current research efforts focused on adaptive multigrid algorithms
  – All of the nitty gritty details worked out
  – Now time for fun
• Hierarchical \textit{and} heterogeneous algorithm research toolbox
  – Aim for scalability \textit{and} optimality
• Lessons today are relevant for Exascale preparation