The March of GPUs

Peak Double Precision FP

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

Peak Memory Bandwidth

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

Double Precision:
- NVIDIA GPU
- x86 CPU

Monday, August 5, 13
Stunning Graphics Realism

Lush, Rich Worlds

Incredible Physics Effects

Core of the Definitive Gaming Platform
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

Compute-Intensive Functions

Rest of Sequential CPU Code

CPU

Monday, August 5, 13
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

Monday, August 5, 13
3 Ways to Accelerate Applications

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
- Vector Signal Image Processing
- GPU Accelerated Linear Algebra
- Matrix Algebra on GPU and Multicore
- NVIDIA cuFFT
- IMSL Library
- Sparse Linear Algebra
- Building-block Algorithms
- C++ Templated Parallel Algorithms

Monday, August 5, 13
OpenACC Directives

Program myscience

... serial code ...

!$acc kernels

do k = 1,n1

do i = 1,n2

... parallel code ...

endo
dendo

!$acc end kernels

... 

End Program myscience

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original Fortran or C code

Monday, August 5, 13
<table>
<thead>
<tr>
<th>Category</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical analytics</td>
<td>MATLAB, Mathematica, LabVIEW</td>
</tr>
<tr>
<td>Fortran</td>
<td>OpenACC, CUDA Fortran</td>
</tr>
<tr>
<td>C</td>
<td>OpenACC, CUDA C</td>
</tr>
<tr>
<td>C++</td>
<td>Thrust, CUDA C++</td>
</tr>
<tr>
<td>Python</td>
<td>PyCUDA, Copperhead</td>
</tr>
<tr>
<td>C#</td>
<td>GPU.NET</td>
</tr>
</tbody>
</table>
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)
The QUDA Library
QCD applications

• Some examples
  – MILC (FNAL, Indiana, Arizona, 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
  – While porting application is possible (e.g., using OpenACC)
  – All applications have the same common algorithms (e.g., solver)
Enter QUDA

• “QCD on CUDA” - [http://lattice.github.com/quda](http://lattice.github.com/quda)
  – Written in C / C++ / Python

• 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
  – Wilson, clover, twisted mass, HISQ, ASQTAD, staggered, dwf, mobius
  – 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
QUDA is community driven

- Ron Babich (NVIDIA)
- Kip Barros (LANL)
- Rich Brower (Boston University)
- Michael Cheng (Boston University)
- 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 (FNAL)
- Alejandro Vaquero (Cyprus Institute)
- Frank Winter (Jlab)
- Yibo Yang (IHEP)
QUDA Mission Statement

• QUDA is
  – a library enabling legacy applications to run on GPUs
  – open source so anyone can join the fun
  – 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
Mapping the Wilson Dslash to CUDA

- Assign a single space-time point to each thread
  - $V = XYZT$ 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
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
    \[
    \begin{pmatrix}
    a_1 & a_2 & a_3 \\
    b_1 & b_2 & b_3 \\
    c_1 & c_2 & c_3
    \end{pmatrix}
    \]
    \[
    \begin{pmatrix}
    a_1 & a_2 & a_3 \\
    b_1 & b_2 & b_3
    \end{pmatrix}
    c = (a \times b)^{*}
    \]

- Additional 384 flops per site

- Also have an 8-number parameterization (requires sin/cos and sqrt)

- 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%

GeForce GTX Titan
> 1 TFLOPS

Gigaflops

Temporal Extent
Krylov Solver Implementation

- Complete solver **must** be on GPU
  - Transfer b to GPU (reorder)
  - Solve $Mx=b$
  - Transfer $x$ to CPU (reorder)
- Entire algorithm 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

while ($|r_k| > \varepsilon$) {
    $\beta_k = (r_k, r_k)/(r_{k-1}, r_{k-1})$
    $p_{k+1} = r_k - \beta_k p_k$
    $\alpha = (r_k, r_k)/(p_{k+1}, A p_{k+1})$
    $r_{k+1} = r_k - \alpha A p_{k+1}$
    $x_{k+1} = x_k + \alpha p_{k+1}$
    $k = k+1$
}
Kepler Wilson-Solver Performance

K20X CG performance

\[ V = 24^3 xT \]

Wilson-Clover is \( \pm 10\% \)

BiCGstab is -10\%

---

Monday, August 5, 13
Multi-dimensional lattice decomposition

[Diagram of multi-dimensional lattice decomposition with NVIDIA logo]
#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 default interface provides a simple view for the outside world
  • C or Fortran
  • Host applications simply pass cpu-side or gpu-side pointers (new!)
  • QUDA takes care of all field reordering and data copying
  • No GPU code in user application
  • User application specifies multi-node topology

• Limitation
  • No control over memory management
  • No external opaque gpu objects
  • Low-level interface under development

• Limitation
  • No control over memory management
  • No external opaque gpu objects
  • Low-level interface under development
Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma
48^3x512 lattice
Relative Scaling (Application Time)

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

# Nodes
0 128 256 384 512 640 768 896 1024 1152 1280

Relative Scaling
0 2 4 6 8 10 12 14 16 18

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

3.58x vs. XE6 @1152 nodes
MILC on QUDA

- RHMC Gauge generation benchmark
  - Volume = $24^3 \times 64$
  - QUDA: solver, forces, fat link
  - MILC: long link, outer product
- Single node performance
  - 2 CPUs vs. 2 GPUs
  - 6x net gain in performance
  - But potential >8x gain in performance
  - Porting remaining functions (J. Foley)
    - Long link this week
- MILC + QUDA used in production for HotQCD

![MILC GPU Performance](image)
Future Directions

• LQCD coverage (avoiding Amdahl)
  – Remaining terms for optimal gauge generation
  – Contractions
  – Eigenvector solvers (EigCG and Arnoldi probably first)
  – Gauge fixing
  – Heatbath / over relaxation algorithms?

• QUDA Application?
  – Collect QUDA routines into a self-contained (R)HMC application

• Performance
  – Scaling
  – Learning from today’s lessons (software and hardware)
Conclusions

• Introduction to QUDA
• Optimal performance through bleeding-edge techniques
• Legacy Applications ready for accelerators
• Still more of work to do
  – New developers welcome
• Lessons today are relevant for Exascale preparation
Backup slides
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
Chroma (Lattice QCD) – High Energy & Nuclear Physics

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

- GPUDirect RDMA will radically improve strong scaling
  – Coming in soon in QUDA
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
Multi-dimensional Communications Pipeline

Total 9 cuda Streams

0: kernels

1: X-backward

2: X-forward

... ...

7: T-backward

8: T-forward

interior kernel

exterior kernels

Gather kernels for all directions are launched after all gather kernels finishes, overlapping completely with the communications.

The interior kernel is executed in the GPU at the beginning so that the communications in all dimensions are likely to exceed the interior kernel run time. The accumulation of communication over multiple dimensions is likely to exceed the interior kernel run time, therefore must be placed in the same stream and be synchronous with the interior kernel.

While the interior kernel can be overlapped with communication as well as overlapping the data movement, the communications in the exterior kernels dominate the computations and any reduction in communication cost is likely to improve the performance.

Communication cost dominates the computations and any reduction in communication cost is likely to improve the performance. CUDA streams are extensively used to overlap computations with communication. Communication is likely to improve the performance as well as overlapping the data movement.

Communications between multiple nodes PCIe data transfers, host memcpy and inter-node communications. PCIe data transfers and multiple stages of communications. One stream is used for interior and exterior kernels, one for gathering and exchanging spinors in the forward direction and the other in the backward direction.

Two streams per dimension are used for gather kernels; one for gathering and exchanging spinors in the forward direction and the other in the backward direction.

Overall dslash performance leading to the idle GPU seen Fig 3, thus degrading the performance. The two host memcpy are required due to the fact that different communication components have different mapping schemes, but the different data mapping makes it impossible and one has to choose the fastest changing index and T the slowest changing index for the mapping scheme.

Such directives in local spinor region still follows the T slowest changing index reading and writing of the destination spinors, which is local to the ghost spinor and all gauge field and spinor access are coalesced. The use of memory padding avoids the GPU memory partitioning problem and further improves the performance.

The gauge field consists of 18 floating point numbers per site when no reconstruction is employed. The gauge field has data dependency with each other and must be executed that because of the spinors in the corners the exterior kernels source spinors. It is also clear from the above description since most of the data traffic comes from the gauge field and exterior kernels to minimize the coalesced access penalty.

Xi Yi Z slowest rkD to okD mapping schedule with Xki Yki Zk one or another. We choose to compute our index using the complete coalesced access impossible and one has to choose to okD mapping scheme. Such directives in local spinor region still follows the T slowest running index and all data traffic comes from the gauge field and exterior kernels.

However, in the Xi Yi Z exterior kernels, the ghost spinor camping problem is still an issue and further improves the performance.
Adaptive Multigrid

32^3 x 256 anisotropic clover on 1024 BG/P cores

Osborn et al, arXiv:1011.2775

Monday, August 5, 13
Hierarchical algorithms on heterogeneous architectures

GPU
Thousands of cores for parallel processing

CPU
Few Cores optimized for serial work
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.

---

**Monday, August 5, 2013**
Run-time autotuning: Implementation

- Parameters stored in a global cache:
  
  ```
  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:
  
  ```
  TuneParam tuneLaunch(Tunable &tunable, QudaTune enabled, QudaVerbosity verbosity);
  ```
Run-time autotuning: Usage

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

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

- Here \text{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.
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
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
QUDA Low-Level Interface (in development)

• Possible strawman under consideration

```c
lat = QUDA_new_lattice(dims, ndim, lat_param);
u = QUDA_new_link_field(lat, gauge_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(solution, source, u, solver);
QUDA_save_site_field(solution, 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
Domain Decomposition

(Re)Start

Quantities with ^ are in reduced precision

Apply Preconditioner: reduced precision inner solve

Generate Subspace

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

Orthogonalize \( \hat{z}_s \)

\[ \alpha_k = (\hat{z}_k, \hat{r}_k) \]
\[ \hat{r}_{k+1} = \hat{r}_k - \alpha_k \hat{z}_k \]
\[ k = k + 1 \]

Update Solution

Reduced Precision

Solve for \( y = k - 1, \ldots, 0 \):

\[ \gamma_{XI} \sum_{i=k+1}^{k} \beta_{i,I} x_I = \alpha_i \]

Compute correction for \( x \):

\[ \hat{x} = \sum_{i=0}^{k-1} \chi_i p_i \]
\[ x = x + \hat{x} \]

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

Full precision restart if not converged

Monday, August 5, 13
• 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 \( \implies \) 16-bit
  – Need to use a flexible solver \( \implies \) GCR
• Block-diagonal preconditioner impose \( \lambda \) cutoff
• Finer Blocks lose long-wavelength/low-energy modes
  – keep wavelengths of \( \sim O(\Lambda_{QCD}^{-1}) \), \( \Lambda_{QCD}^{-1} \sim 1 \text{fm} \)
• Aniso clover: \( (a_s=0.125 \text{fm}, a_t=0.035 \text{fm}) \implies 8^3 \times 32 \text{ blocks are ideal} \)
  – 48^3 \times 512 \text{ lattice}: 8^3 \times 32 \text{ blocks} \implies 3456 \text{ GPUs}
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)