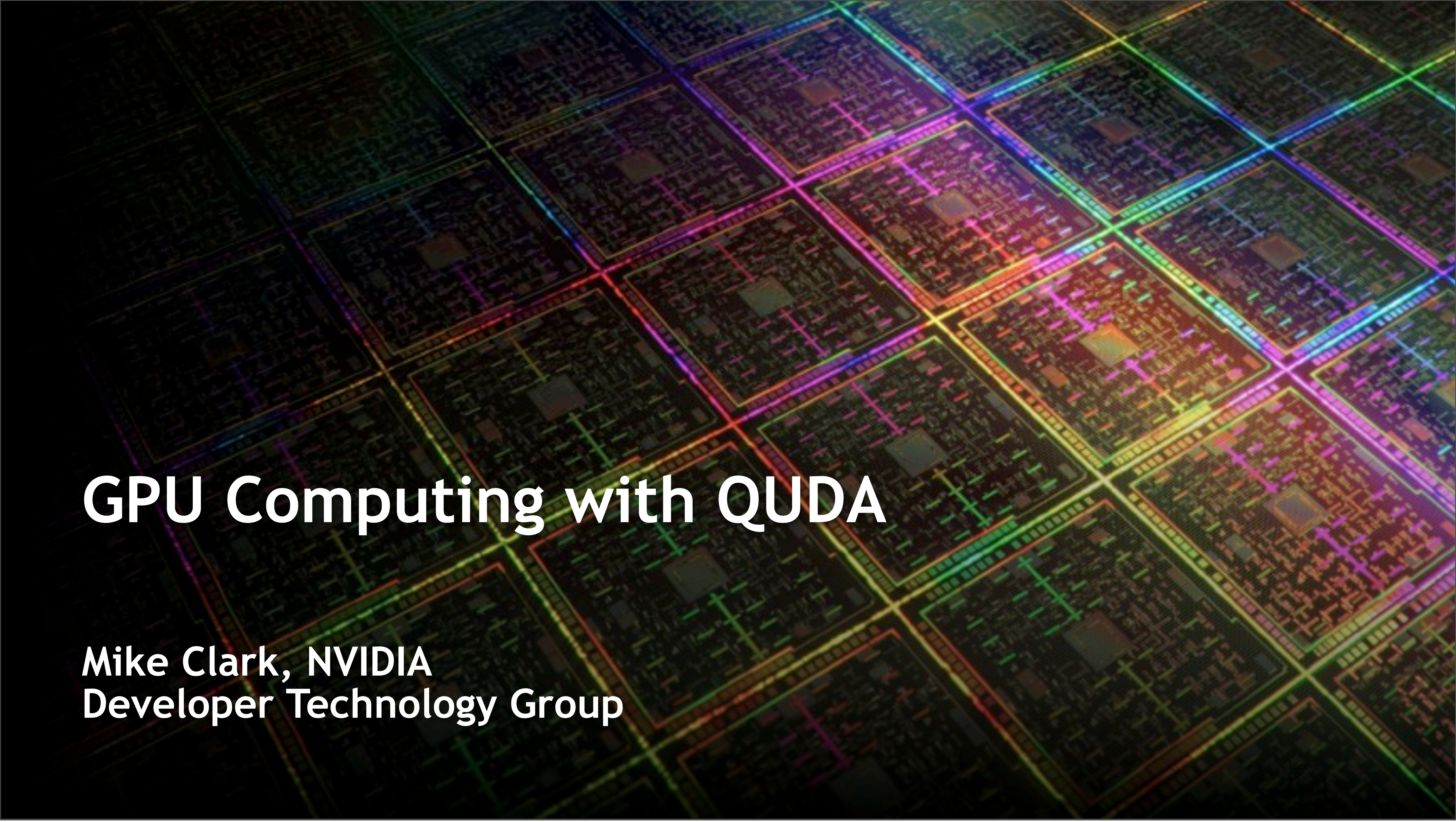


GPU Computing with QUDA

Mike Clark, NVIDIA
Developer Technology Group

QUDA collaborators and developers

- 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 (Jefferson Lab)
- Hyung-Jin Kim (Brookhaven)
- Claudio Rebbi (Boston University)
- Guochun Shi (Google)
- Alexei Strelchenko (FNAL)
- Frank Winter (Jefferson Lab)

Overview

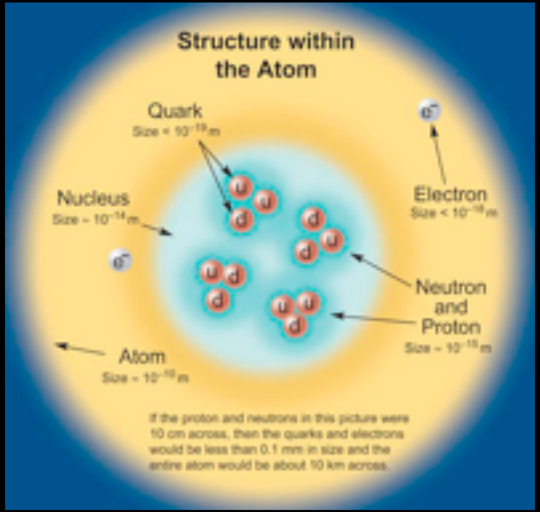
- Quantum Chromodynamics and Lattice QCD
- Motivation
- QUDA Overview
- Interface considerations
- Summary

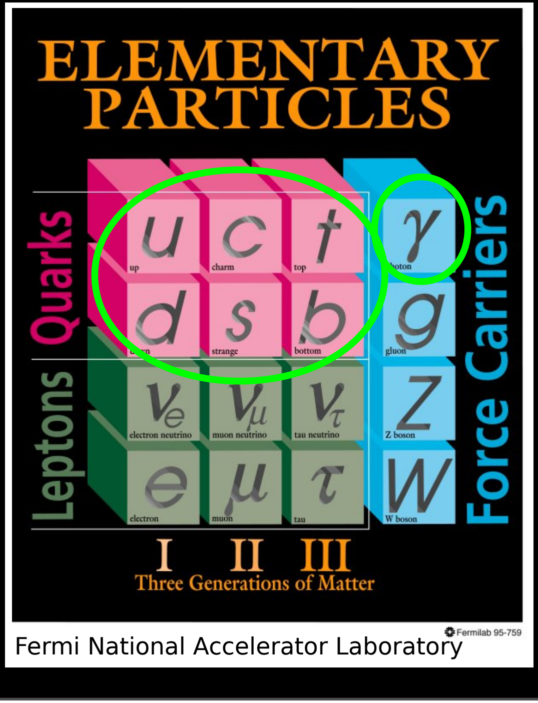
Quantum Chromodynamics

- 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...


Structure within the Atom
Quark
Size < 10^-19 m
Nucleus
Size ~ 10^-14 m
Electron
Size < 10^-18 m
e^-
u
d
Neutron and Proton
Size ~ 10^-15 m
Atom
Size ~ 10^-10 m
If the proton and neutrons in this picture were 10 cm across, then the quarks and electrons would be less than 0.1 mm in size and the entire atom would be about 10 km across.



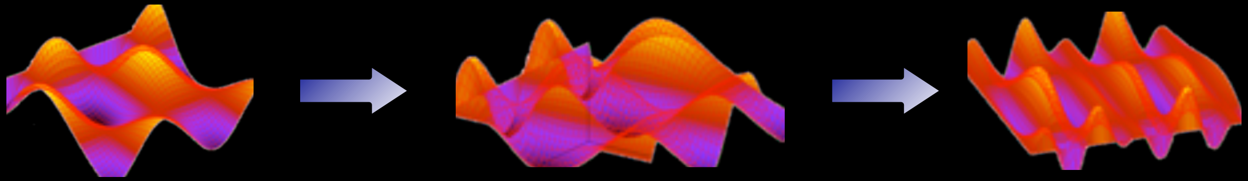
Fermi National Accelerator Laboratory

Lattice Quantum Chromodynamics

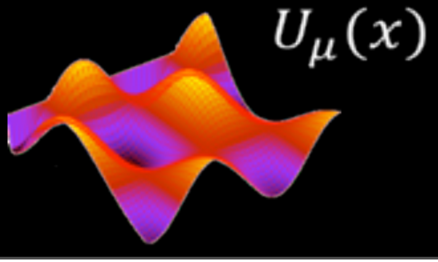
- Theory is highly non-linear ⇒ cannot solve directly
- Must resort to numerical methods to make predictions
- Lattice QCD
  - Discretize spacetime ⇒ 4-d dimensional lattice of size L_x x L_y x L_z x L_t
  - Finitize spacetime ⇒ periodic boundary conditions
  - PDEs ⇒ finite difference equations
- High-precision tool that allows physicists to explore the contents of nucleus from the comfort of their workstation (supercomputer)
- Consumer of 10-20% of North American 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. This requires >O(10 Tflops) sustained for several months (traditionally Crays, Blue Genes, etc.)
   - 50-90% of the runtime is in the solver

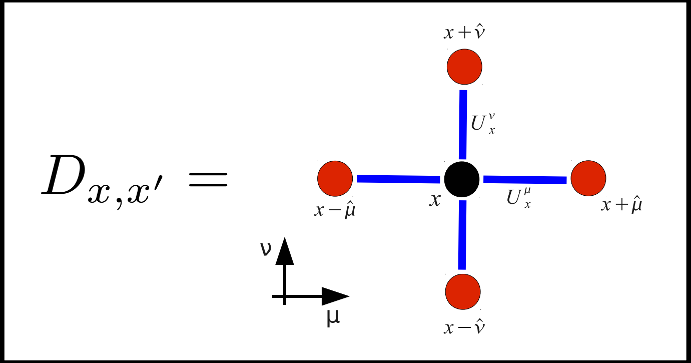


2. "Analyze" the 100s of configurations
   - Can be farmed out, assuming O(1 Tflops) per job
   - 80-99% of the runtime is in the solver



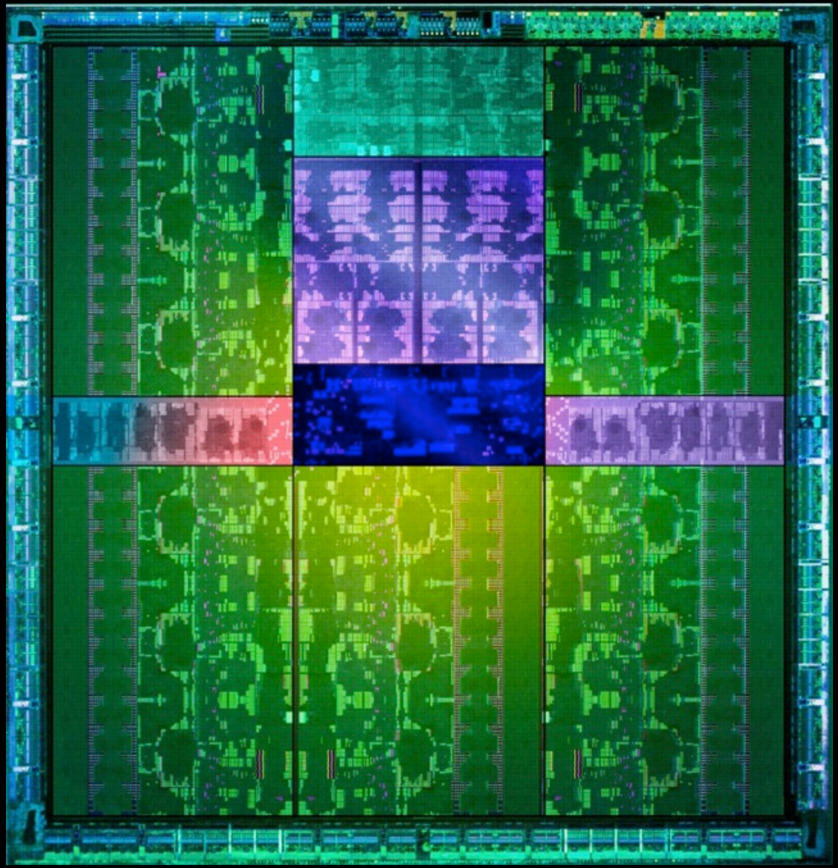
$$D_{ij}^{\alpha\beta}(x,y;U)\psi_j^\beta(y) = \eta_i^\alpha(x)$$

or "$Ax = b$"



Kepler

Fastest, Most Efficient HPC Architecture Ever



SMX ▶ 3x Performance per Watt

Hyper-Q ▶ Easy Speed-up for Legacy MPI Apps

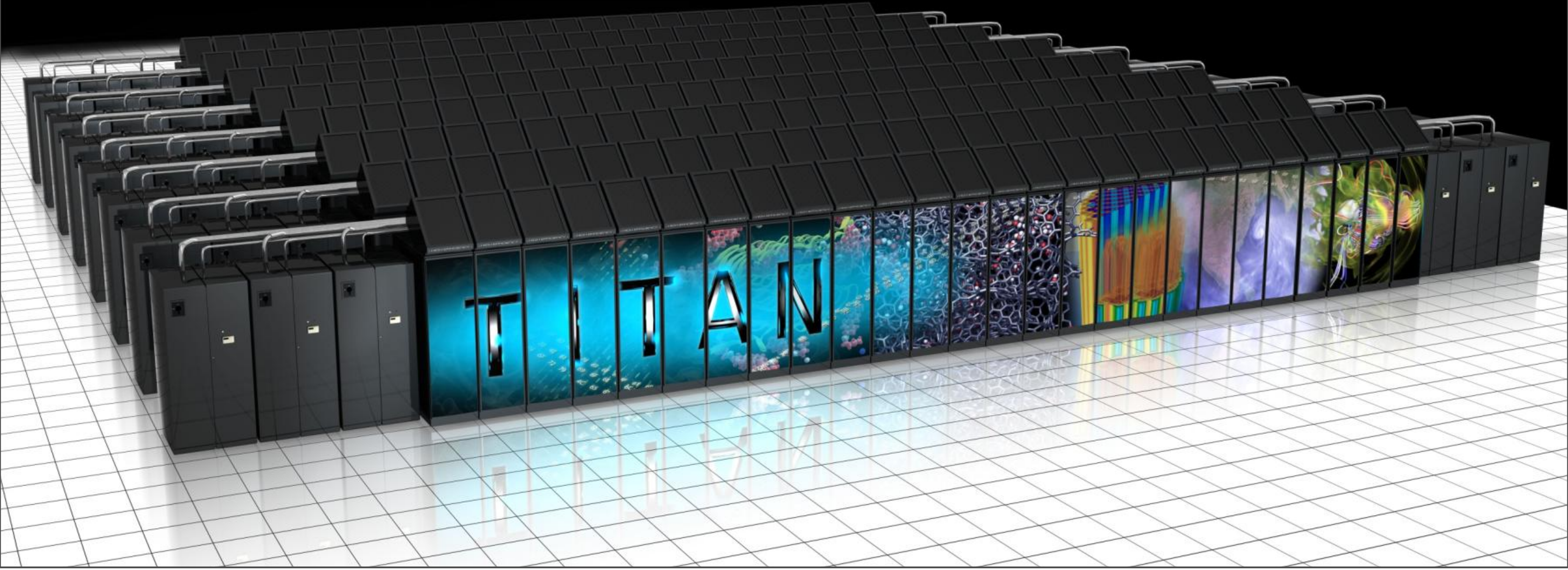
Dynamic Parallelism ▶ Parallel Programming Made Easier than Ever

TITAN: World’s Fastest Supercomputer

18,688 Tesla K20X GPUs

27 Petaflops Peak, 17.59 Petaflops on Linpack

90% of Performance from GPUs



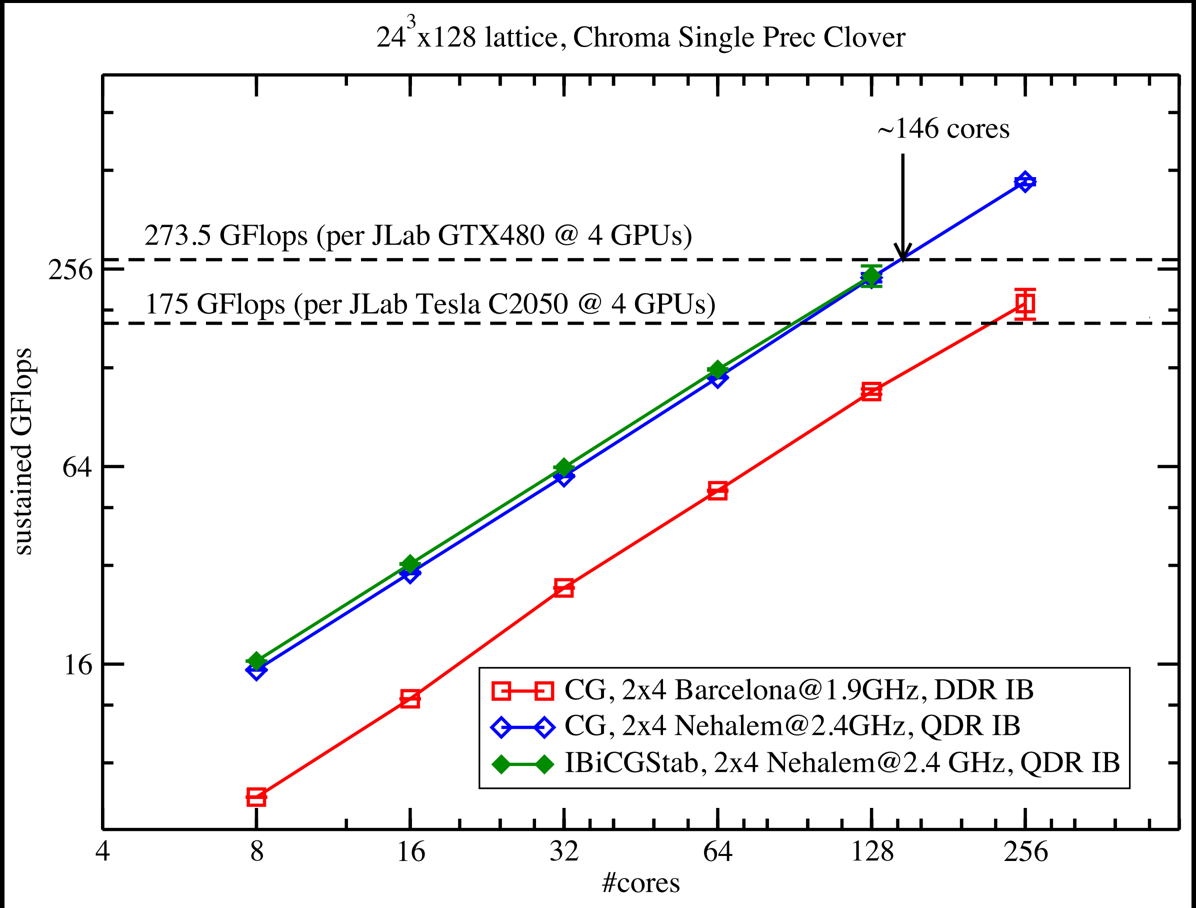
QCD applications

- 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

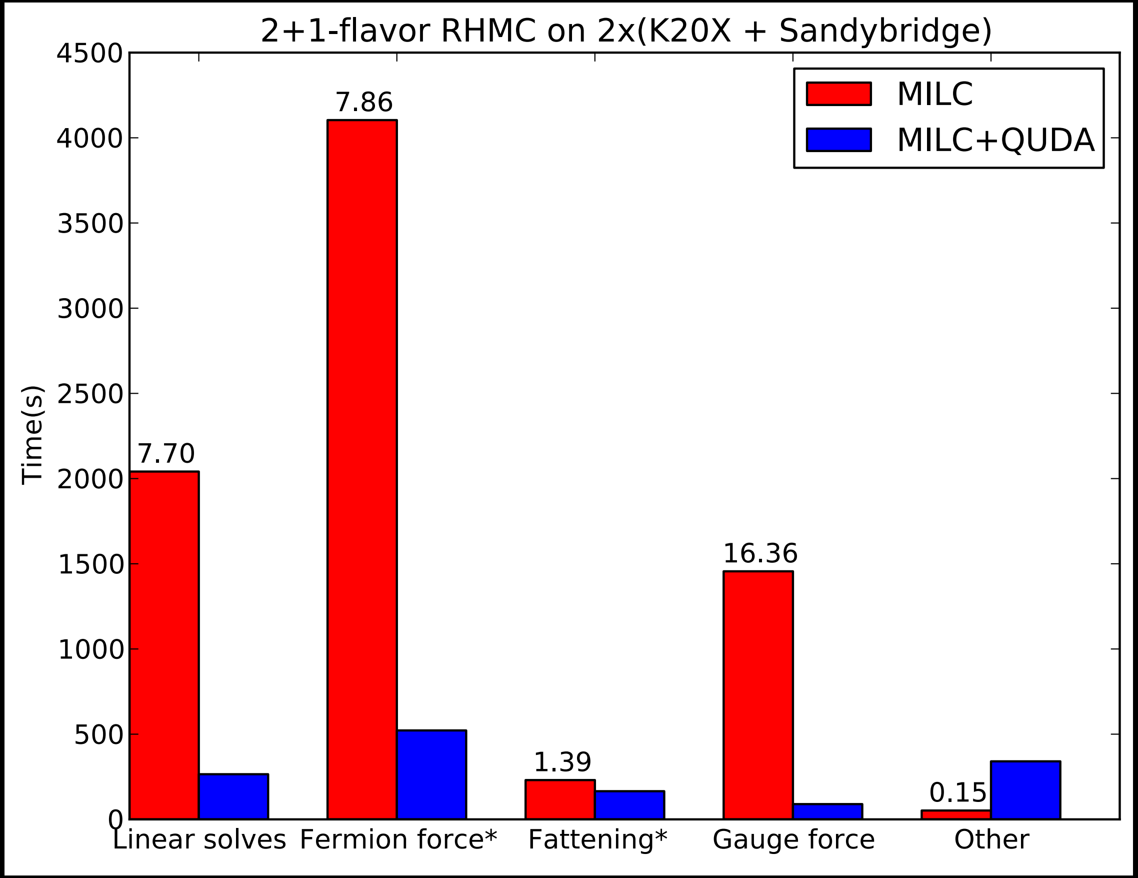
- “QCD on CUDA” – http://lattice.github.com/quda
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, etc.
- Provides:
  - Various solvers for several discretizations, including multi-GPU support and domain-decomposed (Schwarz) preconditioners
  - 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
  - etc.

QUDA Performance - Chroma



- More recent result
  - Complete solver will sustain up to 400 GFLOPS on Kepler
  - 10x speedup vs. Sandy Bridge Xeon

QUDA Performance - MILC



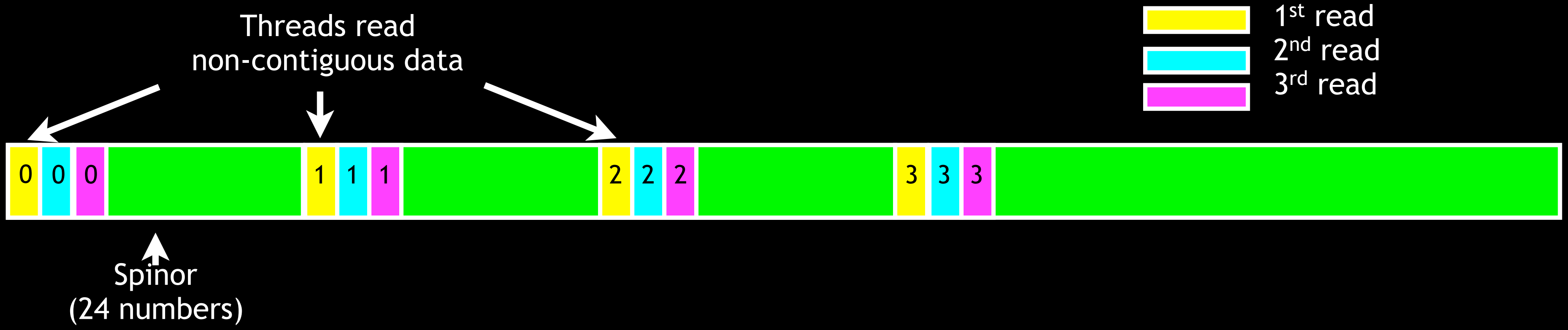
- MILC result on 24^3x64 lattice
  - 5.7x speedup

QUDA - Interfacing Strategy

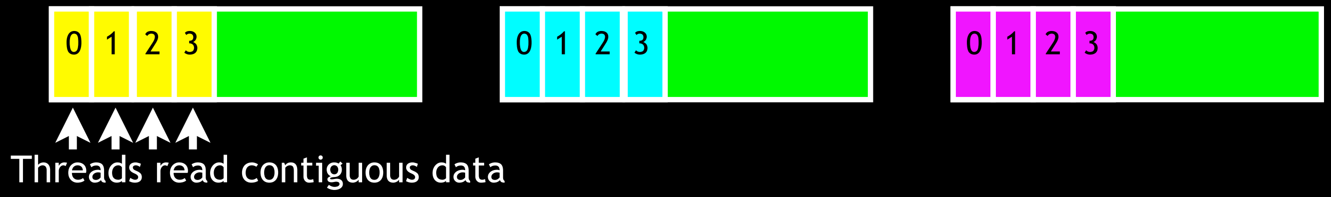
- QUDA designed to accelerate pre-existing LQCD applications
  - Chroma, MILC, CPS, BQCD, etc.
  - Provide an opaque interface
- Interface Design Considerations
  - Field ordering
  - Data residence
  - Multi-GPU
  - Memory management

Field Ordering

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



- GPUs like Structure of Arrays



- QUDA interface deals with all data reordering

- Application remains ignorant

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)
    - Memory-bound operation
    - Deploy double-single and double-half solvers
  - 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

conjugate gradient

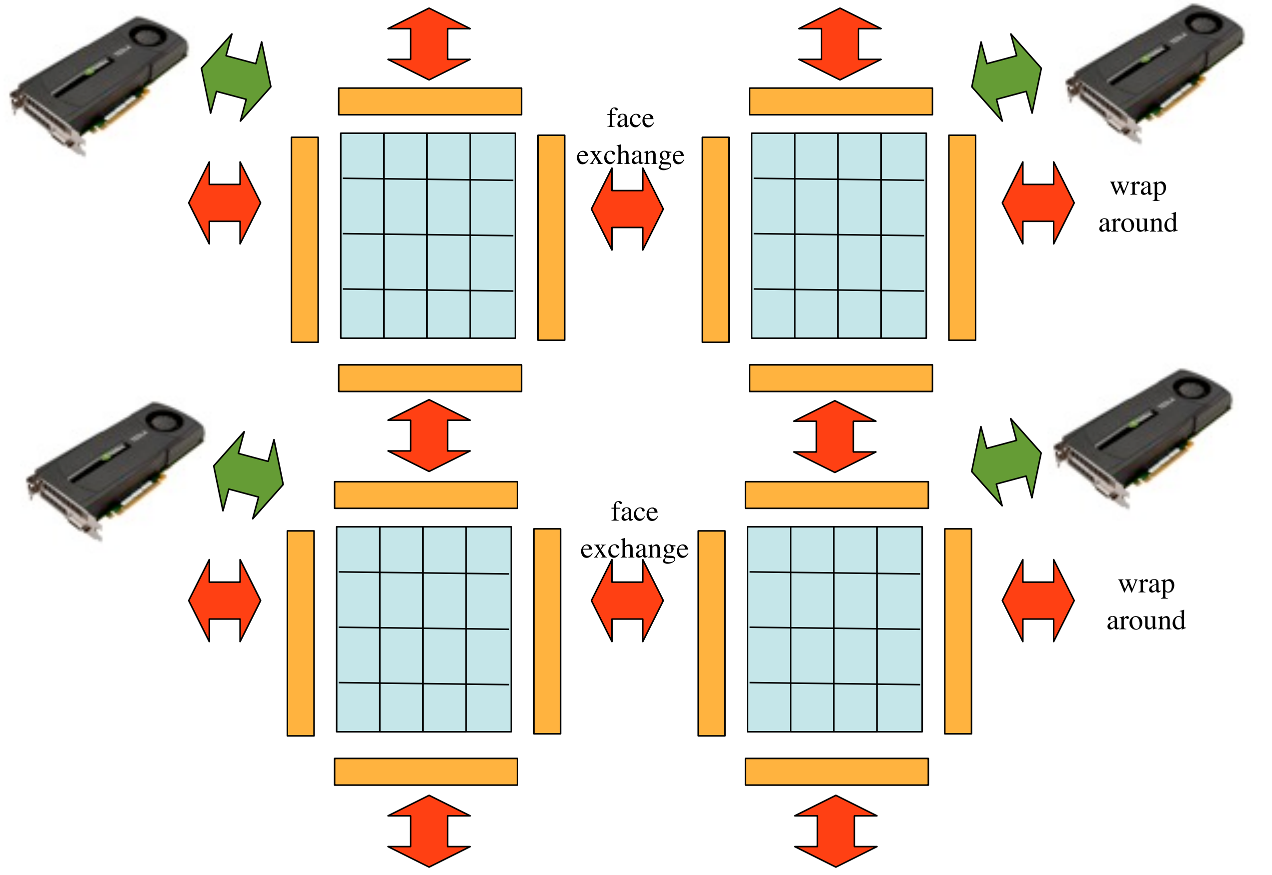
```
while ($|\mathbf{r}_k| > \varepsilon$) {
    $\beta_k = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{r}_{k-1},\mathbf{r}_{k-1})$
    $\mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k\mathbf{p}_k$

    $\alpha = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{p}_{k+1},A\mathbf{p}_{k+1})$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha A\mathbf{p}_{k+1}$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\mathbf{p}_{k+1}$
    $k = k+1$
}
```

mixed-precision

```
while ($|\mathbf{r}_k| > \varepsilon$) {
    $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$
    solve $A\mathbf{p}_k = \mathbf{r}_k$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$
}
```

Multiple GPUs



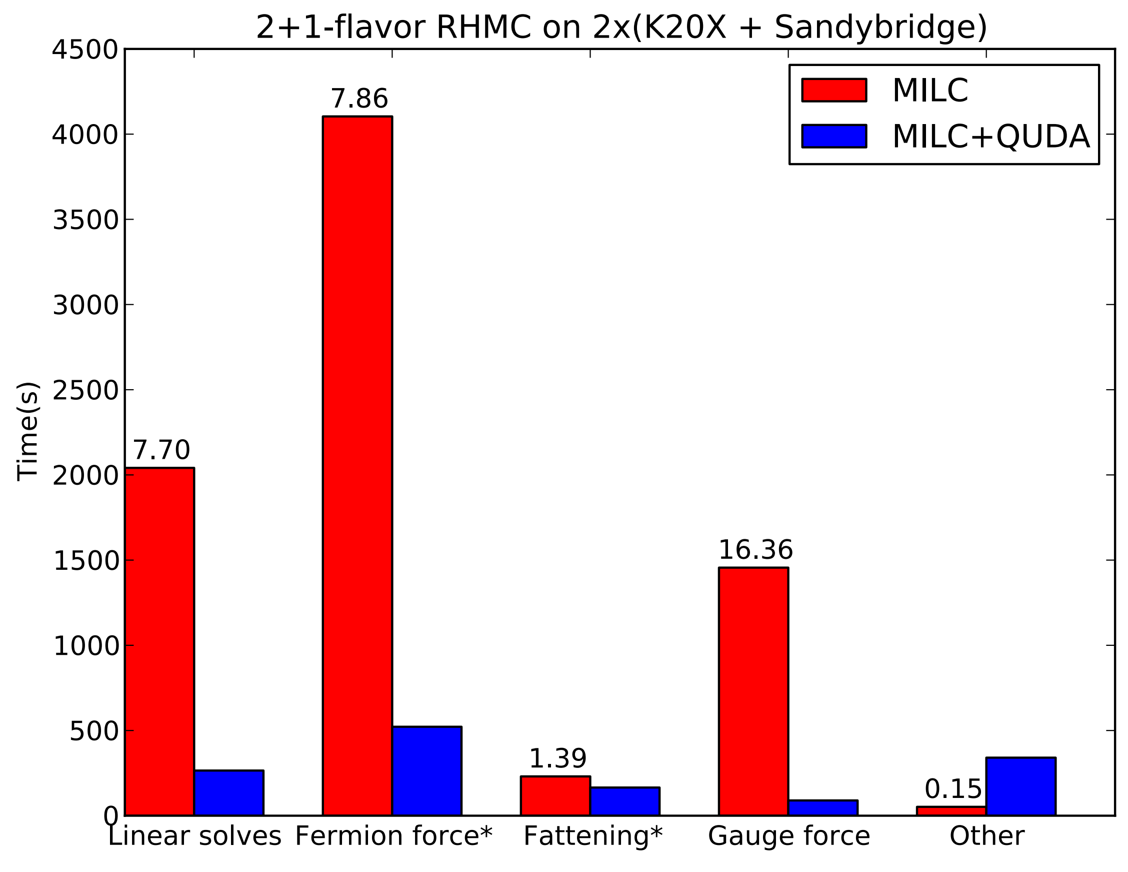
Multiple GPUs

- Many different mechanisms for controlling multiple GPUs
    - MPI processes
    - CPU threads
    - Multiple GPU per thread and do explicit switching
    - Combinations of the above
- QUDA directly supports the simplest: 1 GPU per MPI process
    - Allows partitioning over node with multiple devices and multiple nodes
    - `cudaSetDevice(local_mpi_rank);`
- Any remaining host-code is parallelized using threads
    - This works well for homogenous CPU systems with threaded applications
    - E.g., Chroma and BQCD are fully threaded

Multiple GPUs

- 1 MPI = 1 GPU can be problematic
- Not all LQCD apps are threaded
    - MILC is multi-process only
    - Any work remaining on the CPU only utilizes a single core
- MILC result on 24^3x64 lattice
    - 5.7x net gain in performance
    - But potential >7.7x gain in performance
        - Porting remaining functions

          or
        - Fix host code to run in parallel



Multiple GPUs

- Even threaded CPU applications can have issues
- CPU systems increasingly have NUMA issues
  - e.g., Cray XK7
- GK110 brings a new feature called Hyper-Q
  - Allows multiple MPI processes to share a single GPU (CUDA Proxy)
  - Easily allows full utilization of both CPU and GPU with no app changes
  - Hyper-Q does have some additional latency overhead
- QUDA soon to support directly multiple MPI processes per GPU
  - Communication handled by MPI communicators in the interface
  - Removes CUDA Proxy overhead and works on all GPUs
  - No applications changes required

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

```
#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 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

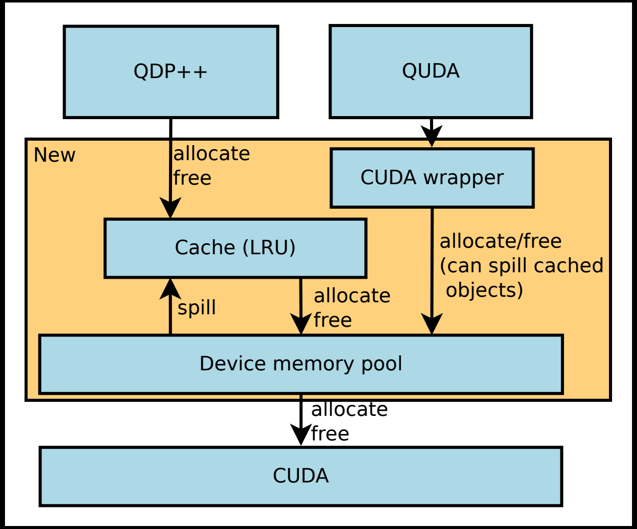
QUDA Low-Level Interface (in development)

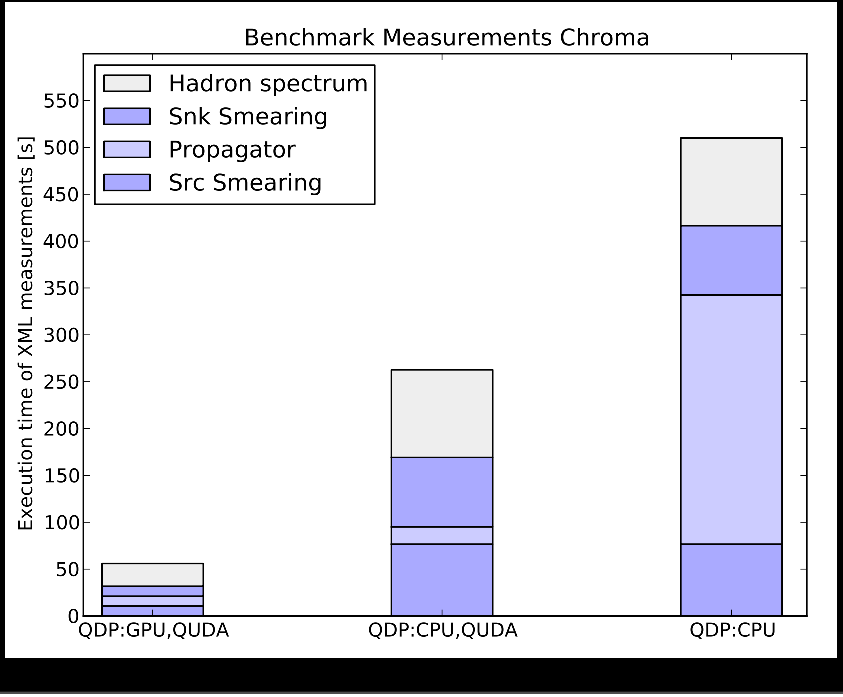
- Possible strawman under consideration

```
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);
etc...
```

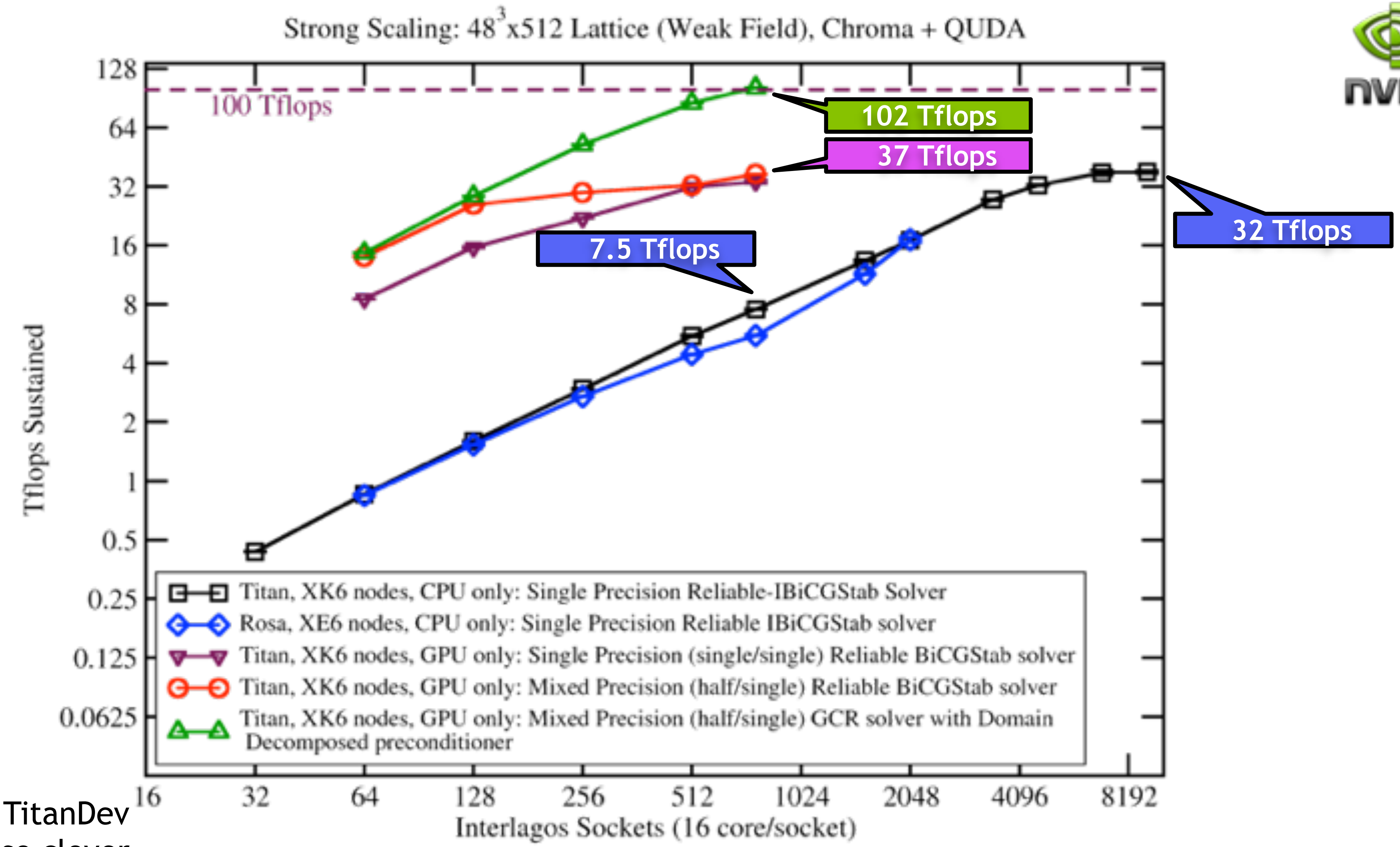
- Here, src, sol, etc. are opaque objects that know about the GPU
- Allows the user to easily maintain data residency
- Users can easily provide their own kernels
- High-level interface becomes a compatibility layer built on top

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

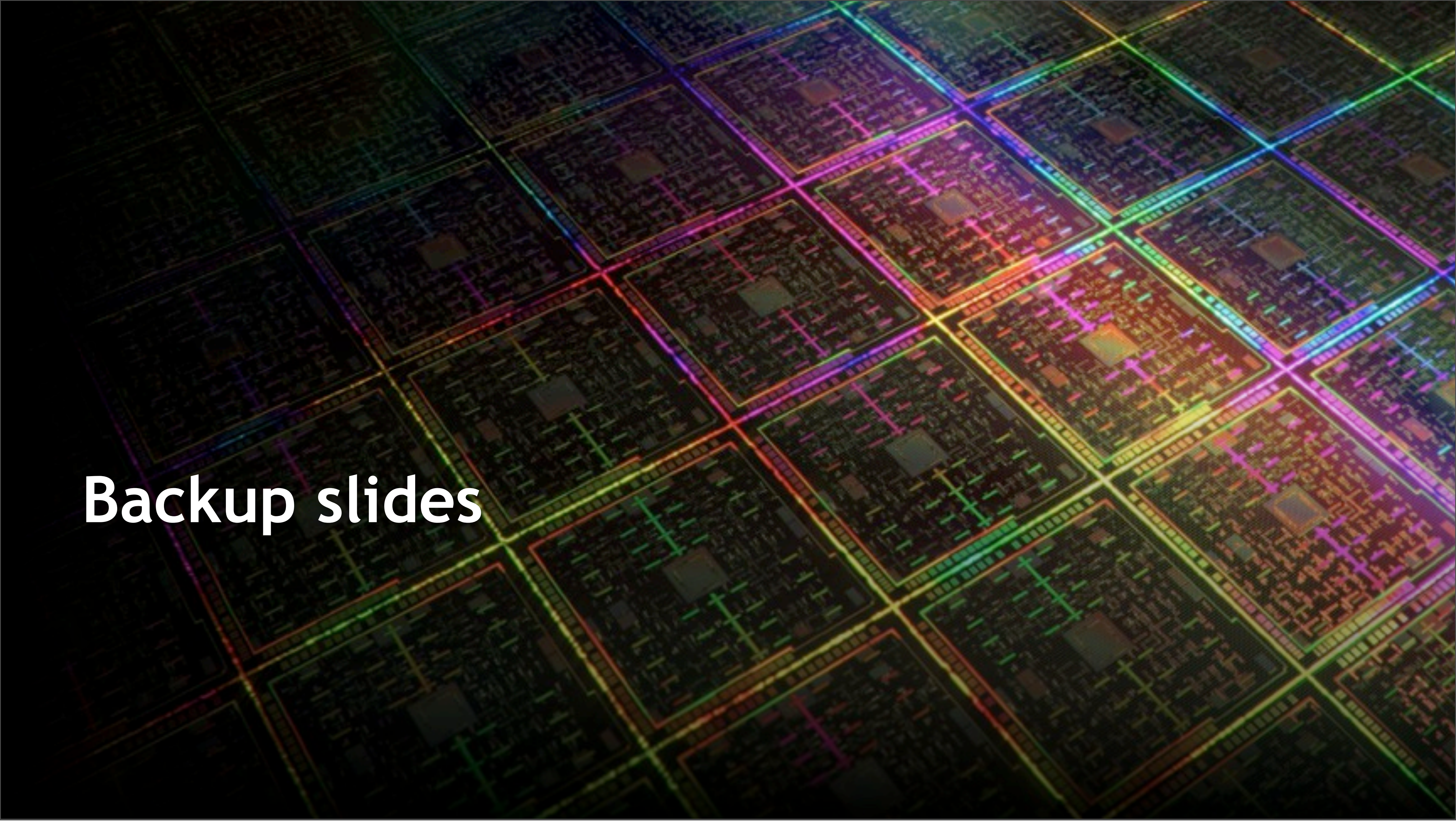


Results from TitanDev

- 48^3x512 aniso clover
- scaling up 768 GPUs

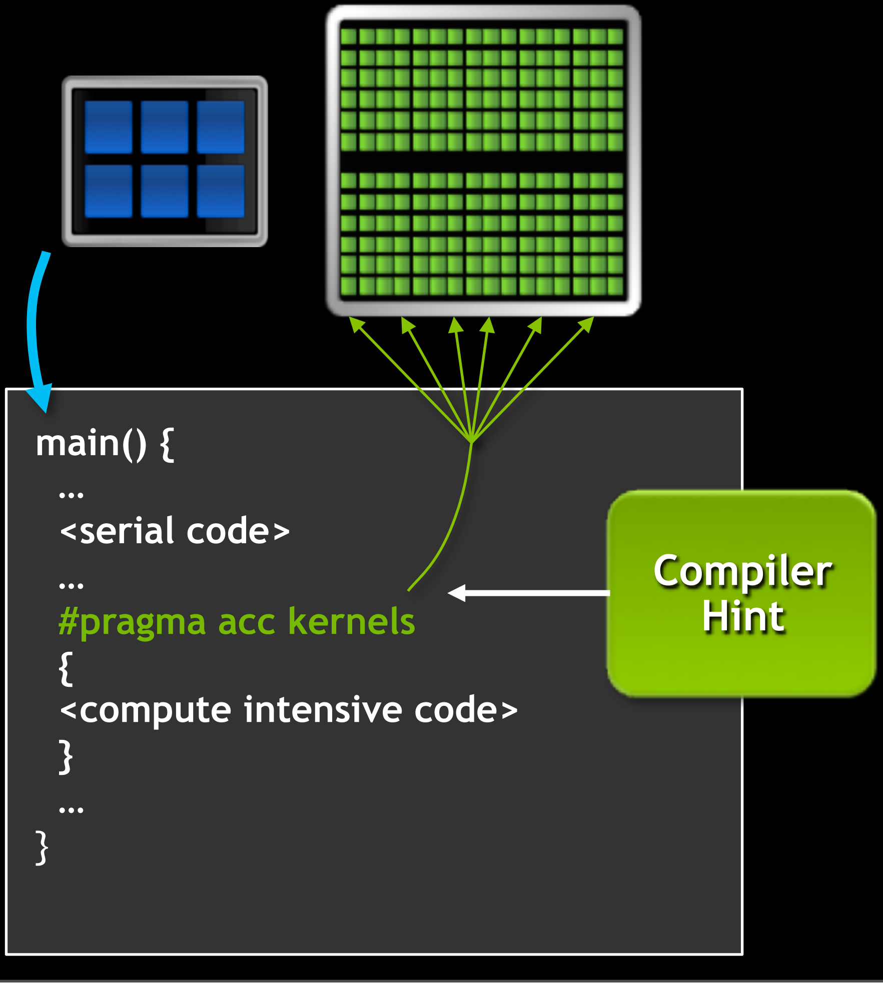
Summary

- Glimpse into the QUDA library
  - GPU library for LQCD applications
- Interface considerations
  - Data ordering
  - Multi-GPU
  - Data residency
- Levels of Interfacing
  - High-level (cpu-side interaction)
  - Lower-level (gpu-side interaction)
  - Tight Integration (common memory pool)
- End result is legacy applications running at large scale on GPUs



Backup slides

OpenACC: Open, Simple, Portable



```
main() {
  …
  <serial code>
  …
  #pragma acc kernels
  {
  <compute intensive code>
  }
  …
}
```

- Open Standard
- Easy, Compiler-Driven Approach
- Portable on GPUs and Xeon Phi

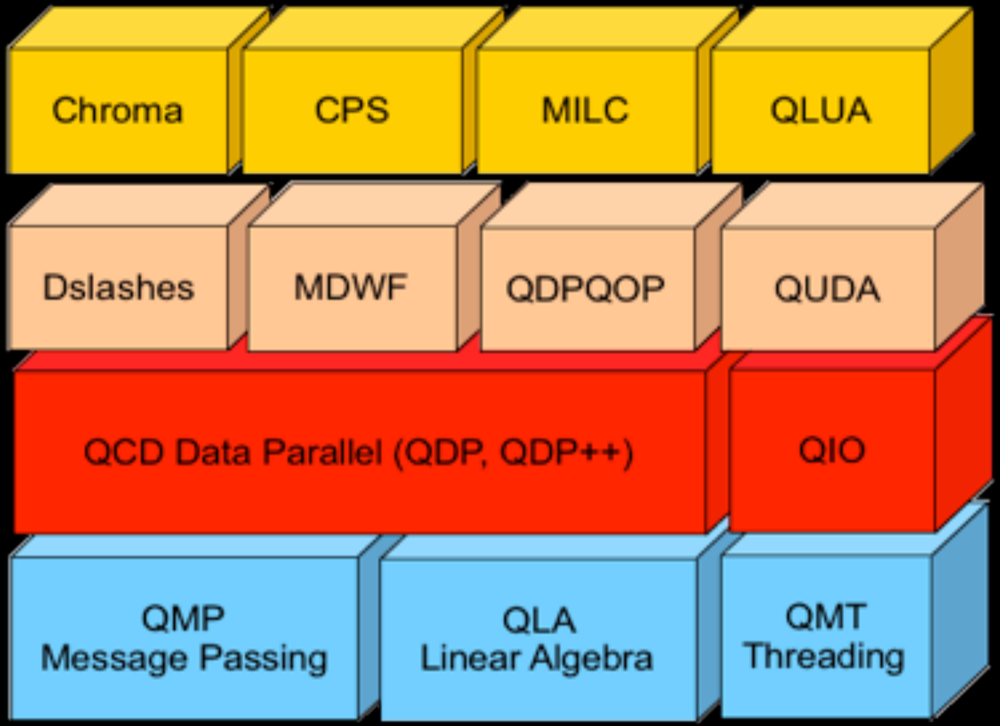
CAM-SE Climate
6x Faster on GPU
2x Faster on CPU only
Top Kernel: 50% of Runtime

Available from: PGI CAPS CRAY

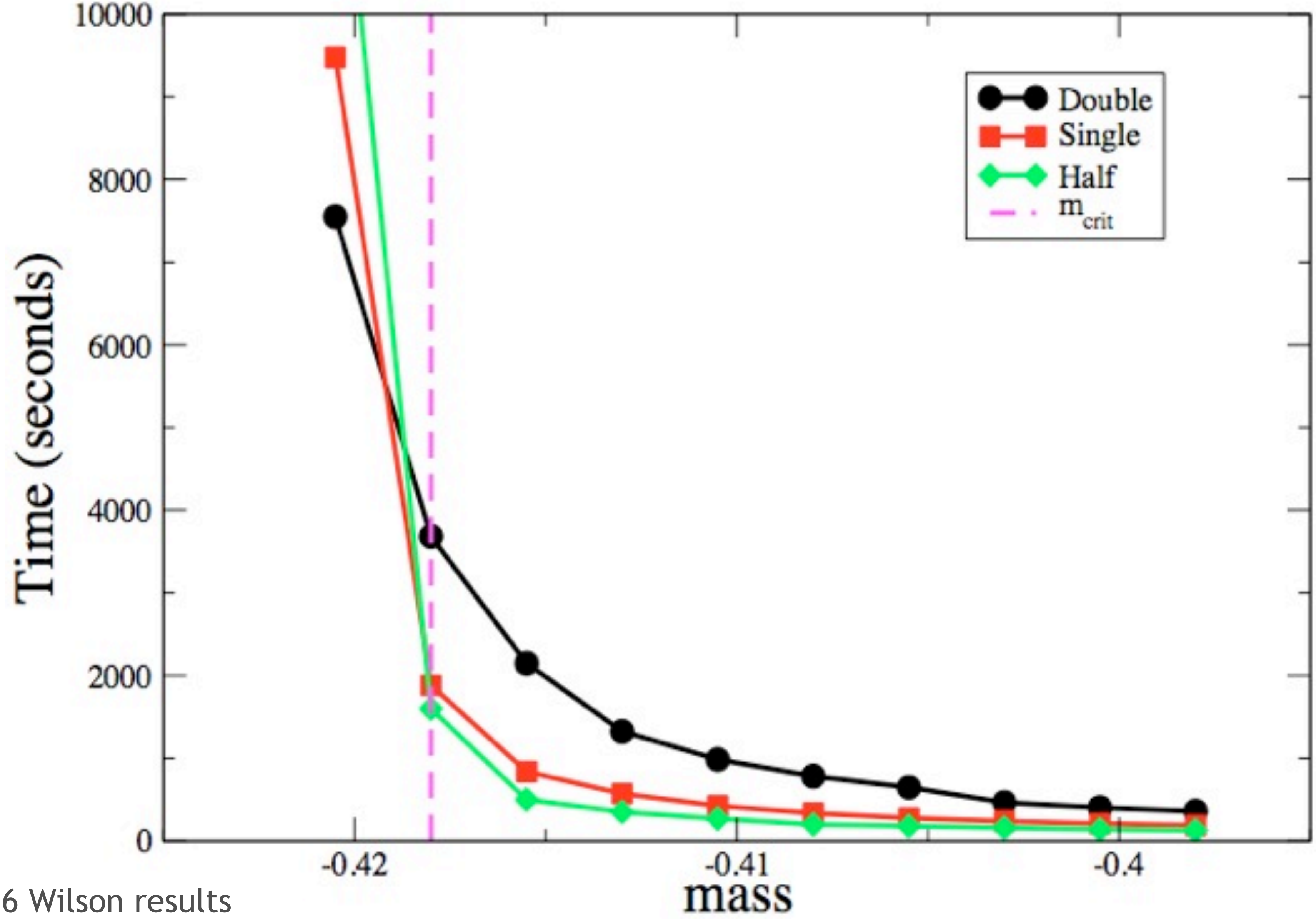
Krylov solvers

- (Conjugate gradients, BiCGstab, and friends)
- Search for the solution to $Ax = b$ in the subspace spanned by $\{b, Ab, A^2b, \dots\}$.
- Upshot:
  - We need fast code to apply A to an arbitrary vector (called the *Dslash* operation in LQCD).
  - ... as well as fast routines for vector addition, inner products, etc. (home-grown “BLAS”)

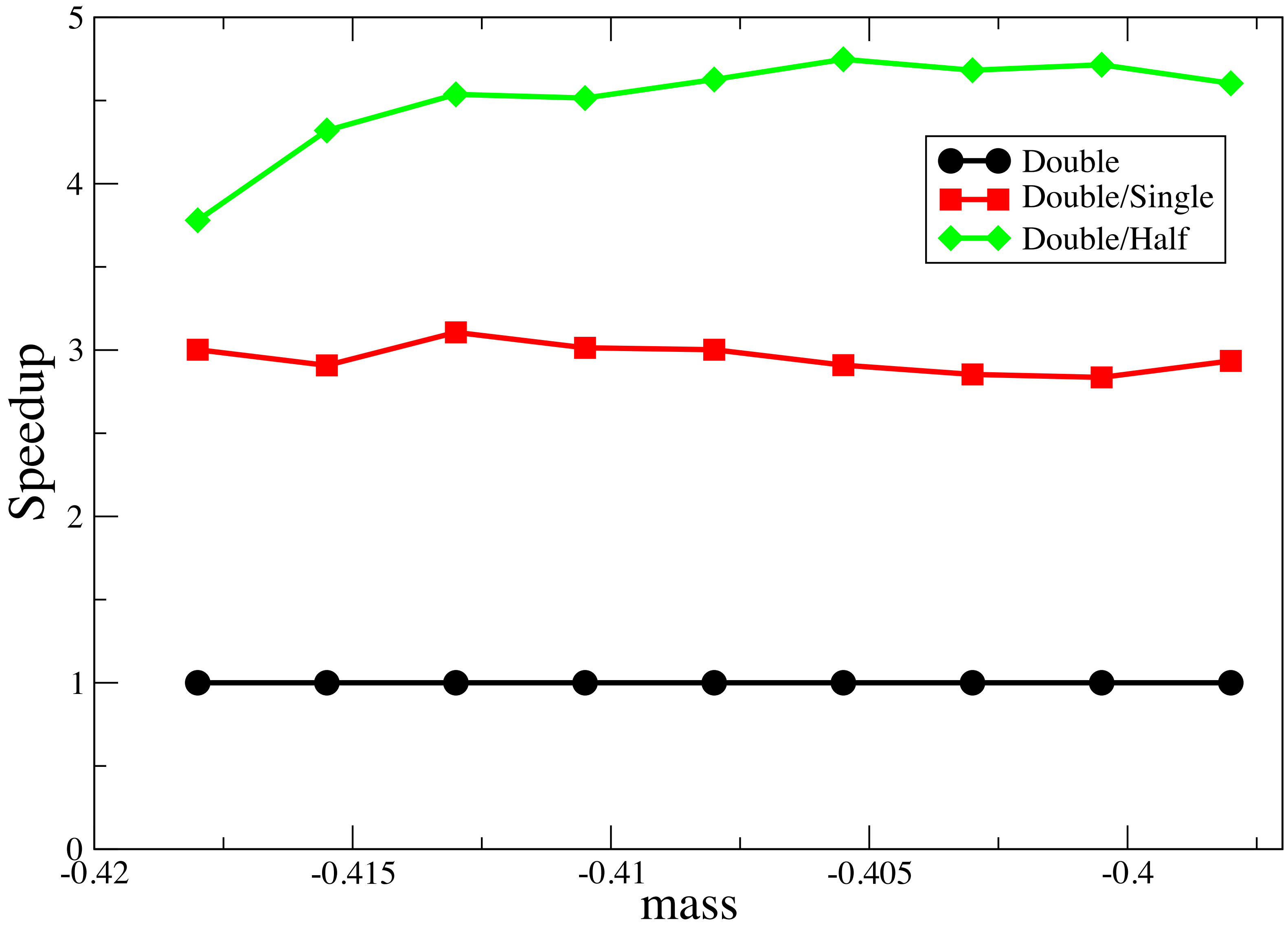
USQCD software stack



(Many components developed under the DOE SciDAC program)

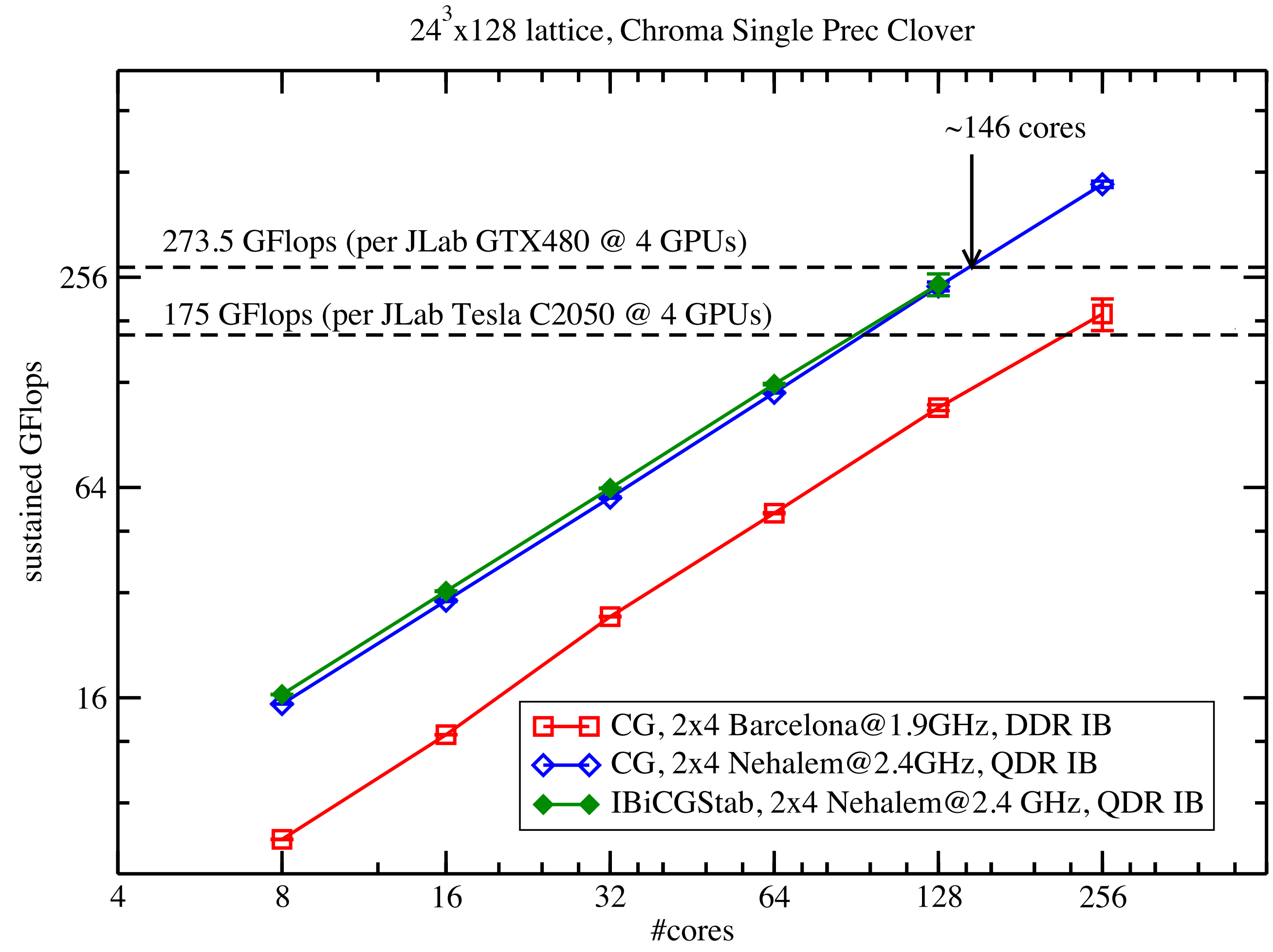


32^3x96 Wilson results
on GTX 280 (for illustration)

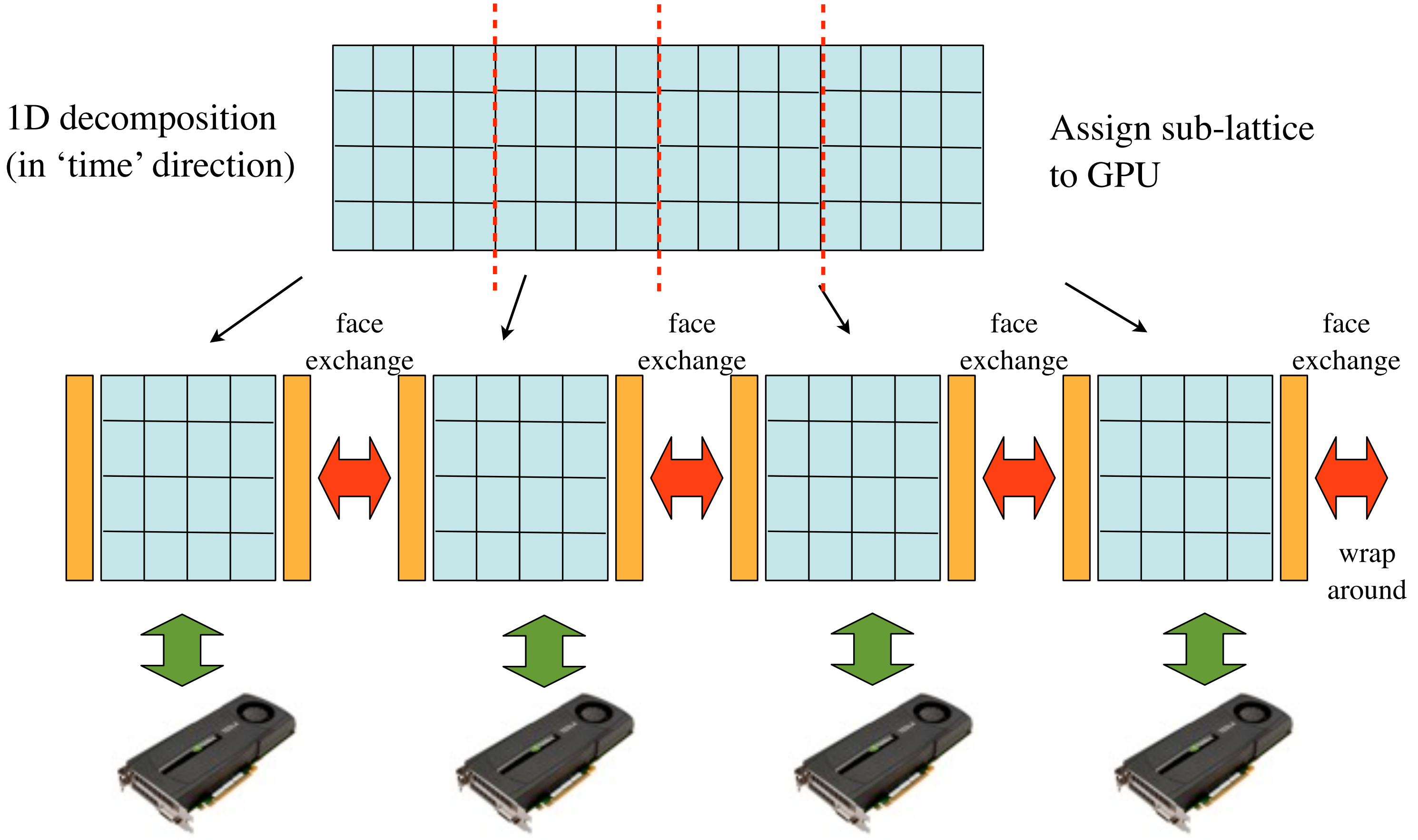


32^3x96 Wilson results
on GTX 280 (for illustration)

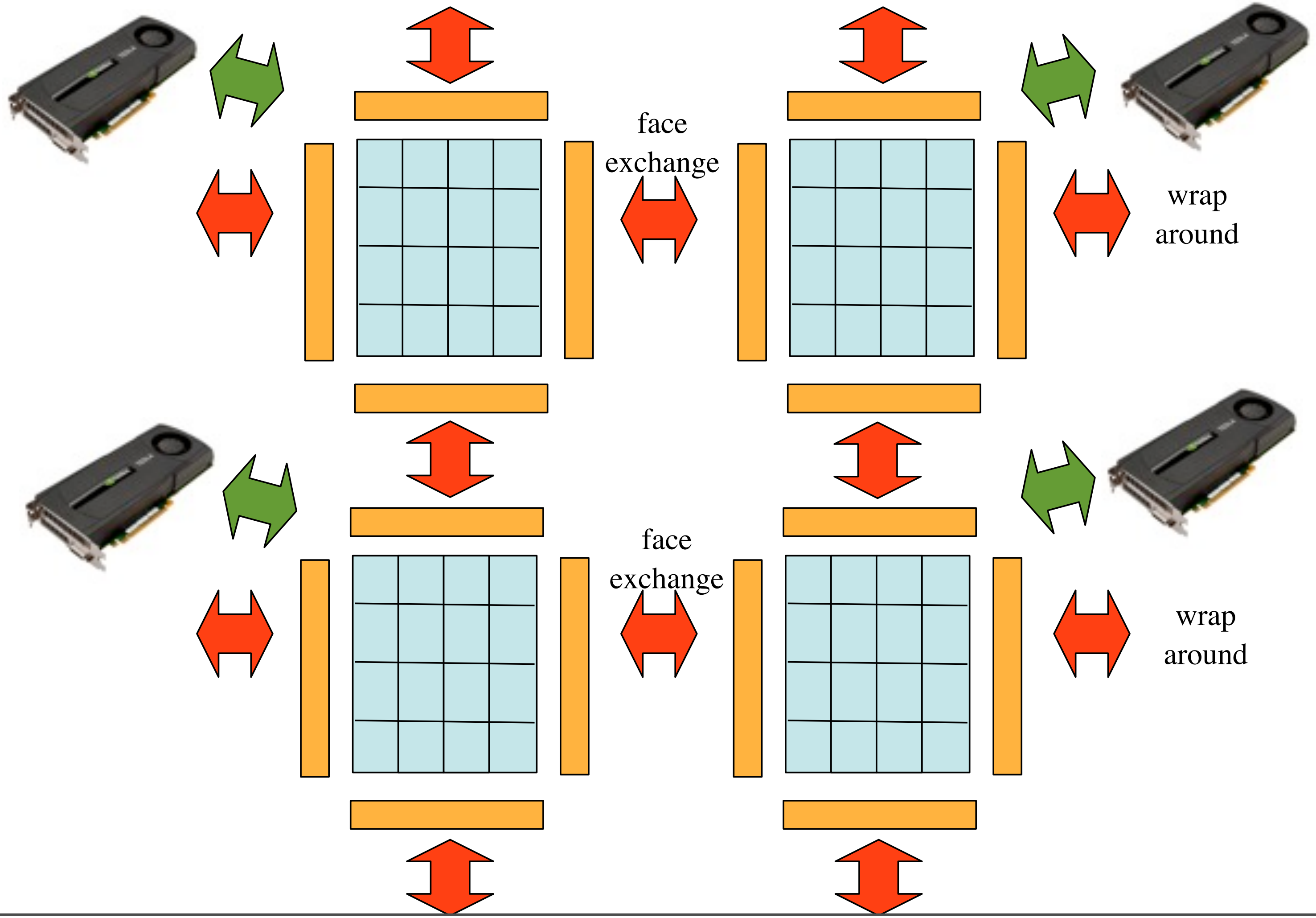
GPUs vs. CPUs


24^3x128 lattice, Chroma Single Prec Clover
~146 cores
273.5 GFlops (per JLab GTX480 @ 4 GPUs)
175 GFlops (per JLab Tesla C2050 @ 4 GPUs)
256
64
16
sustained GFlops
4
8
16
32
64
128
256
#cores
CG, 2x4 Barcelona@1.9GHz, DDR IB
CG, 2x4 Nehalem@2.4GHz, QDR IB
IBiCGStab, 2x4 Nehalem@2.4 GHz, QDR IB

1D Lattice decomposition



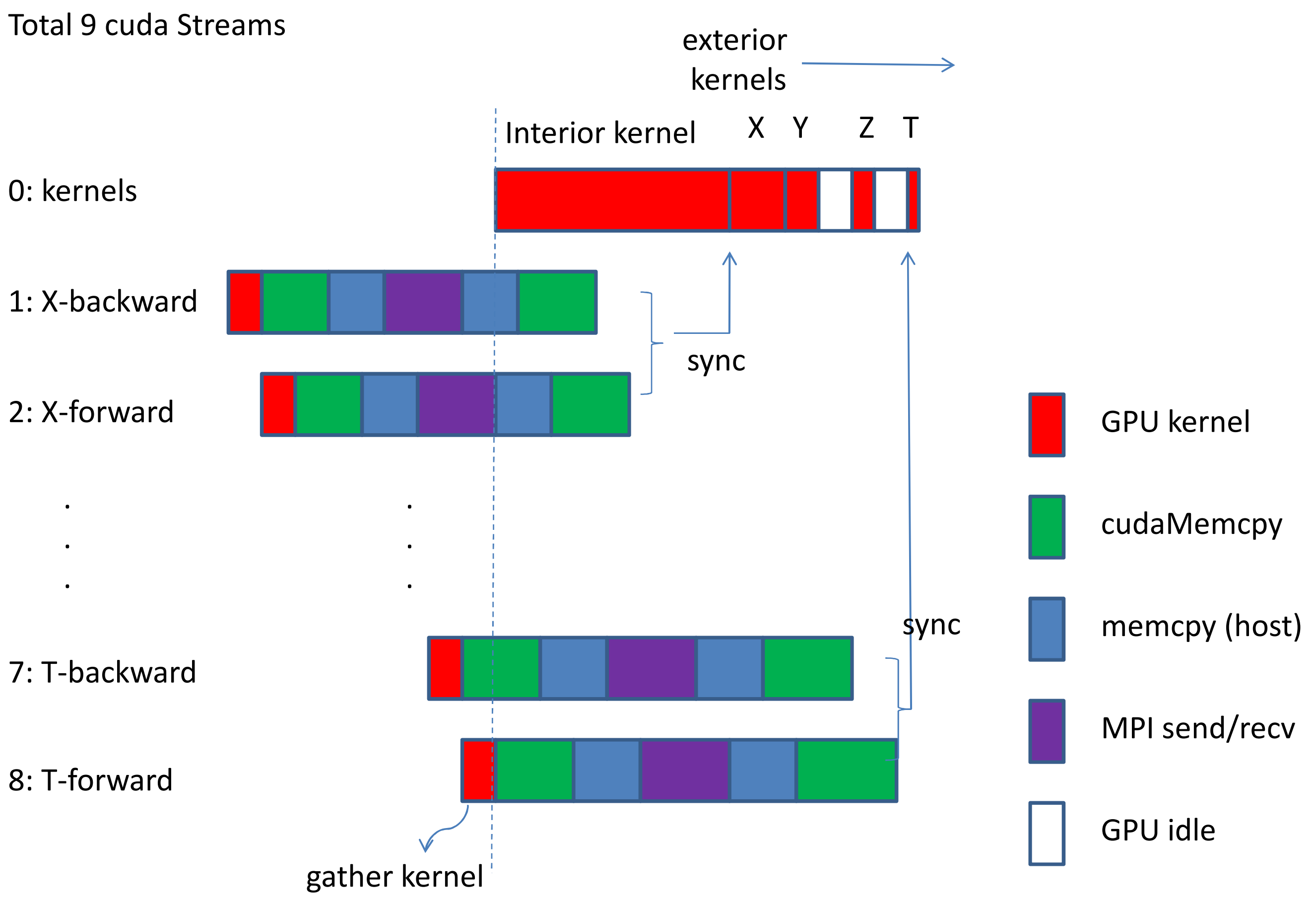
Multi-dimensional lattice decomposition



CUDA Stream API

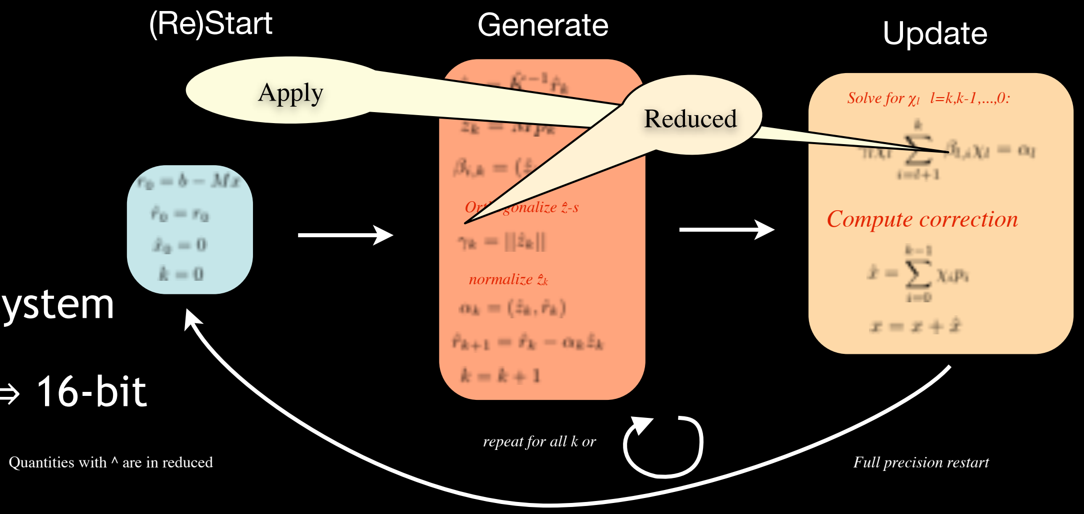
- CUDA provides the stream API for concurrent work queues
  - Provides concurrent kernels and host<->device memcpys
  - Kernels and memcpys are queued to a stream
    - `kernel<<<block, thread, shared, streamId>>>(arguments)`
    - `cudaMemcpyAsync(dst, src, size, type, streamId)`
  - Each stream is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

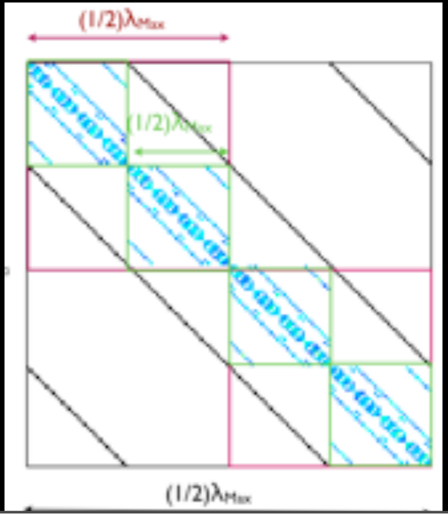
Multi-dimensional Communications Pipeline



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 $\implies$ 16-bit
- Need to use a flexible solver $\implies$ GCR
- Block-diagonal preconditoner impose λ cutoff
- Finer Blocks lose long-wavelength/low-energy modes
  - keep wavelengths of ~ O(Λ_{QCD}^{-1}), Λ_{QCD}^{-1} ~ 1fm
- Aniso clover: (a_s=0.125fm, a_t=0.035fm) $\implies$ 8^3x32 blocks are ideal
  - 48^3x512 lattice: 8^3x32 blocks $\implies$ 3456 GPUs


(Re)Start
Generate
Update
Apply
Reduced
$r_0 = b - Mx$
$\hat{r}_0 = r_0$
$\hat{x}_0 = 0$
$k = 0$
$\hat{z}_k = \hat{K}^{-1}\hat{r}_k$
$\hat{z}_k = \hat{M}\hat{p}_k$
$\beta_{i,k} = (\hat{z}_i, \hat{z}_k)$
Orthogonalize $\hat{z}$-s
$\gamma_k = ||\hat{z}_k||$
normalize $\hat{z}_k$
$\alpha_k = (\hat{z}_k, \hat{r}_k)$
$\hat{r}_{k+1} = \hat{r}_k - \alpha_k \hat{z}_k$
$k = k + 1$
Solve for χ_l l=k,k-1,...,0:
$\gamma_l \chi_l + \sum_{i=l+1}^{k} \beta_{l,i}\chi_i = \alpha_l$
Compute correction
$\hat{x} = \sum_{i=0}^{k-1} \chi_i p_i$
$x = x + \hat{x}$
repeat for all k or
Quantities with ^ are in reduced
Full precision restart

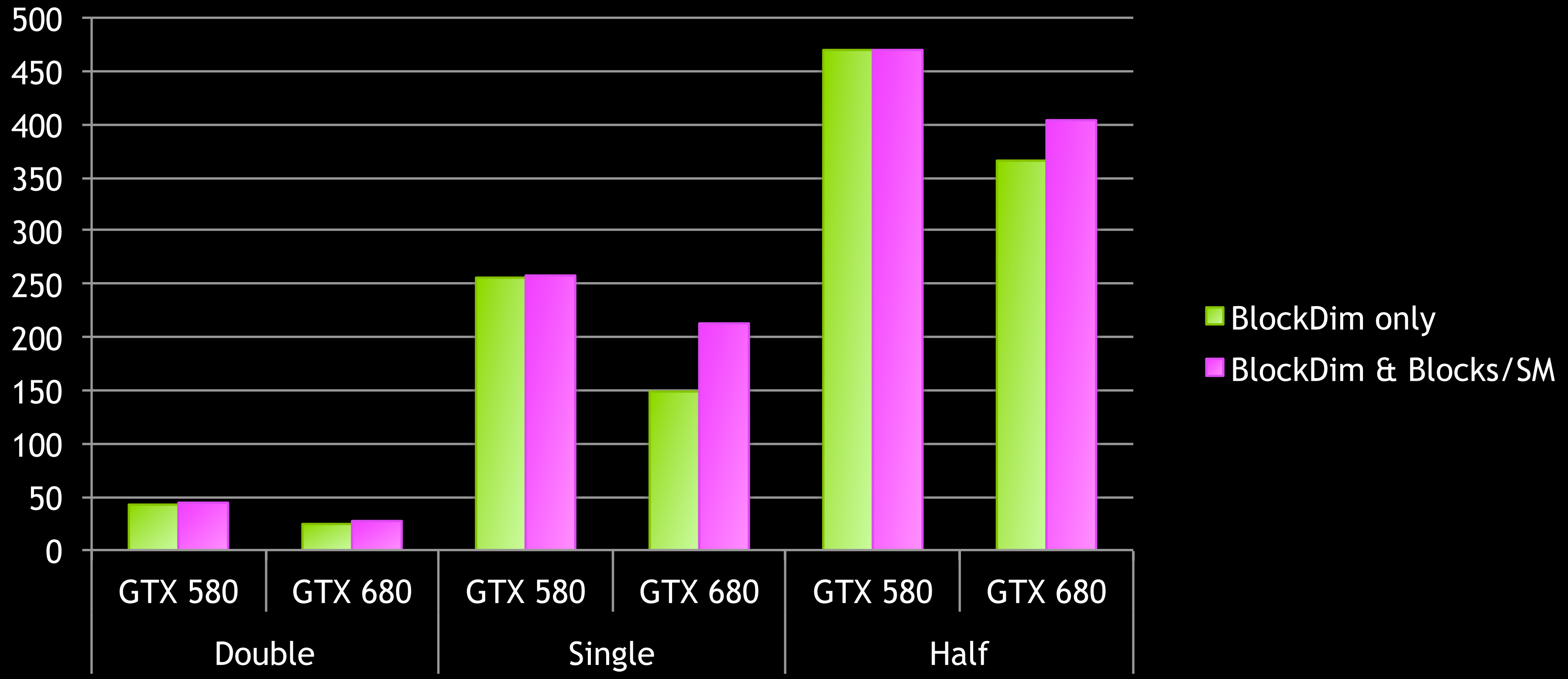


Run-time autotuning

- 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.



Tuesday, April 30, 13

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:

```
myKernelWrapper(a, b, c);
```

- After:

```
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.