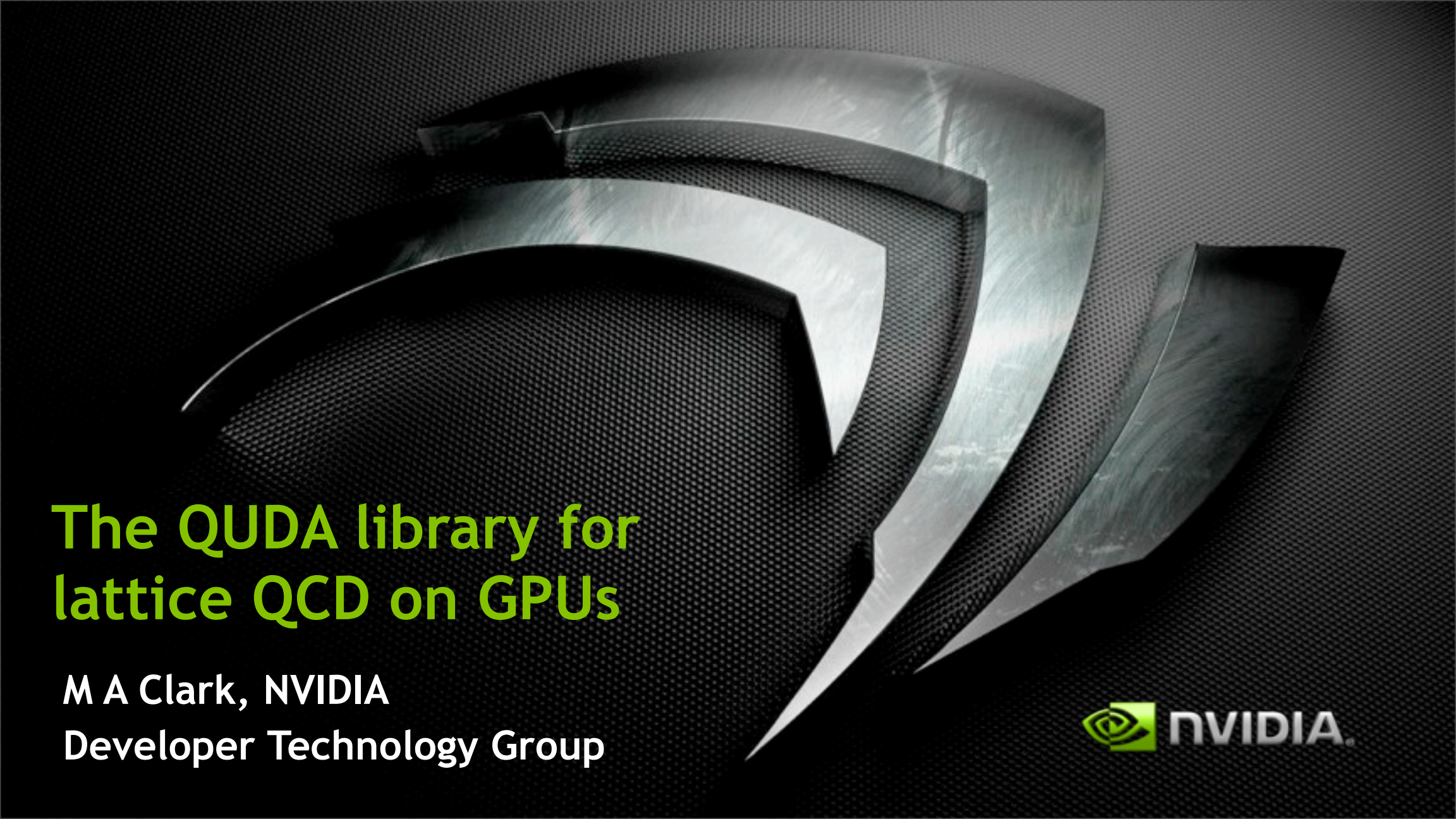


The QUDA library for lattice QCD on GPUs

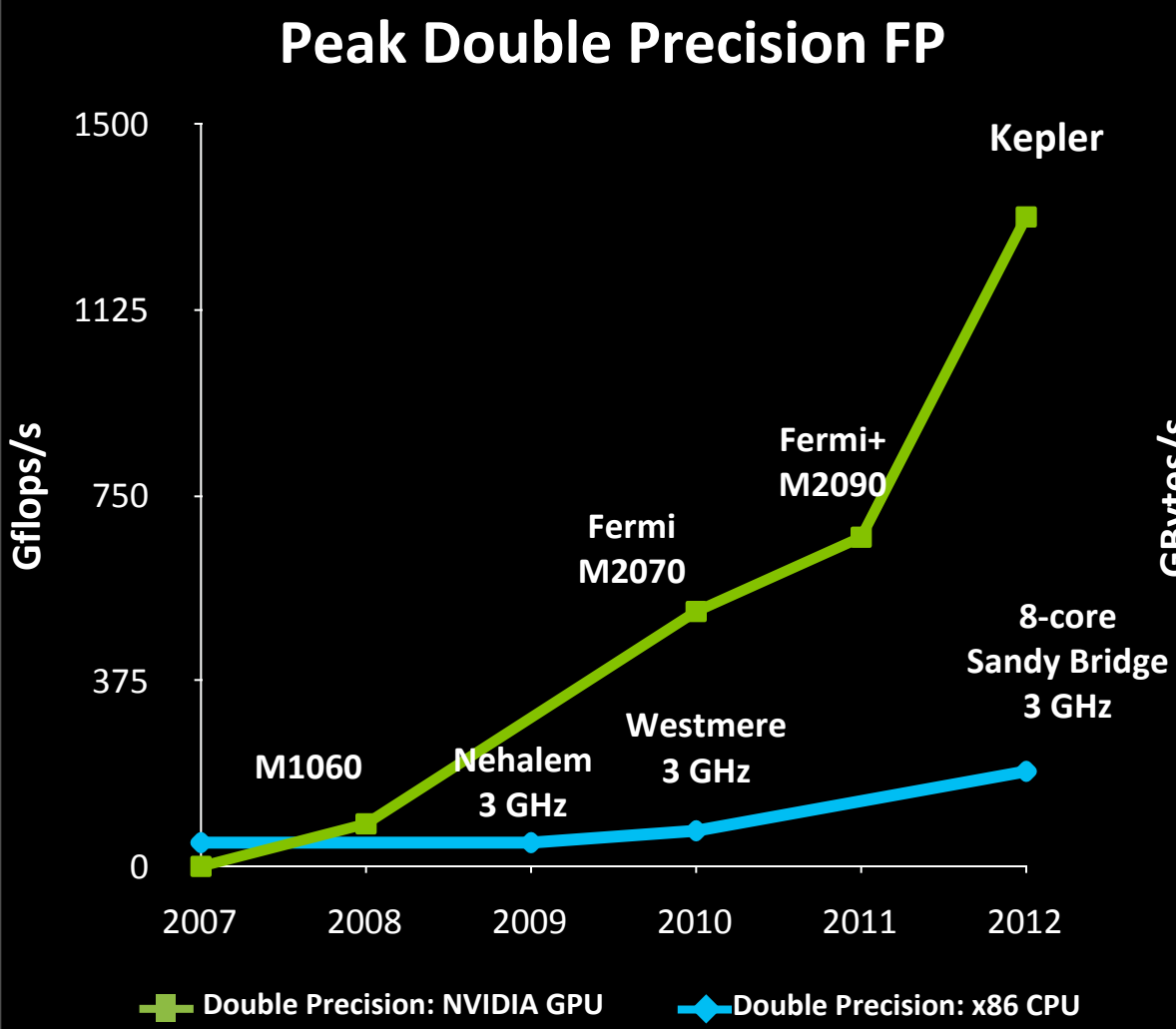
M A Clark, NVIDIA
Developer Technology Group

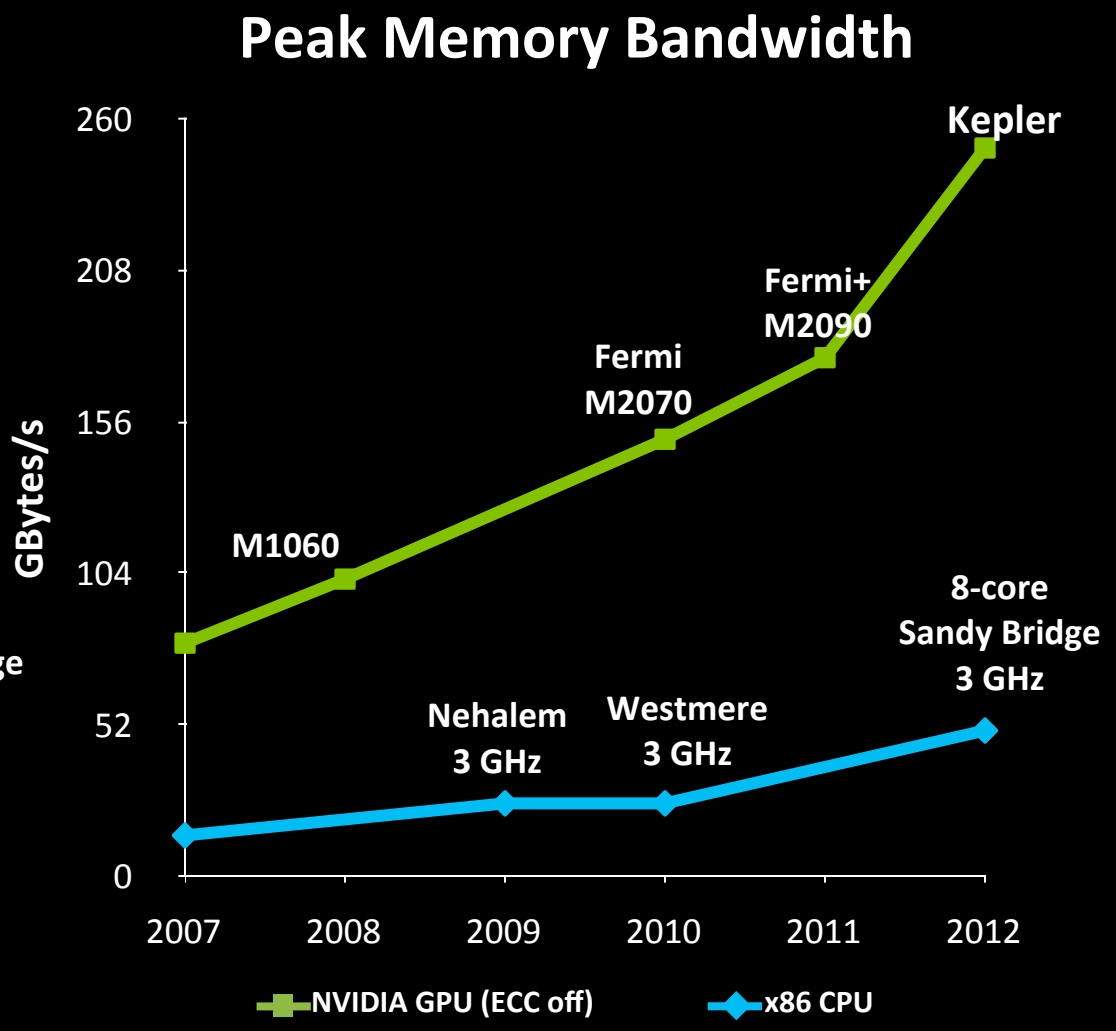
NVIDIA

Outline

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

The March of GPUs





Stunning Graphics Realism

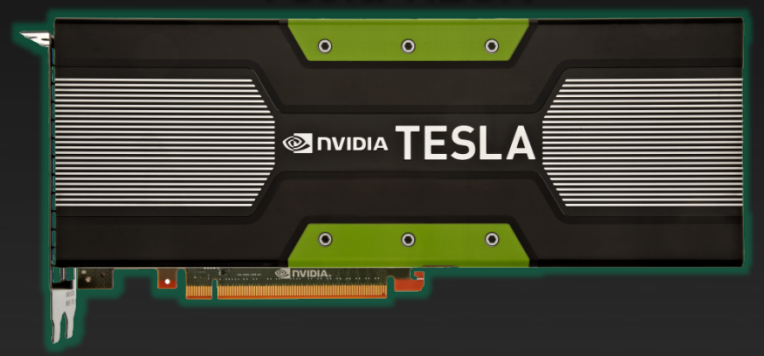
Lush, Rich Worlds

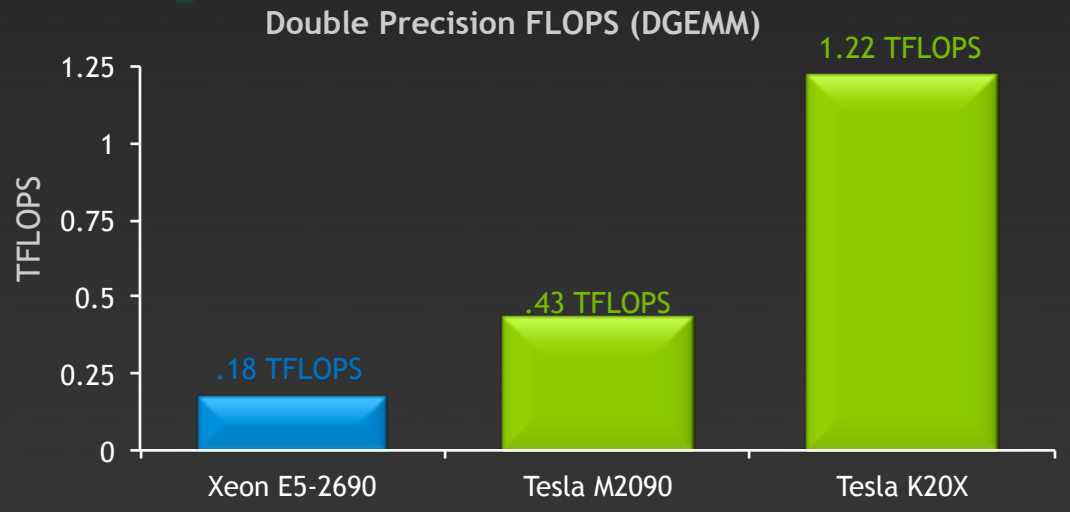
Incredible Physics Effects

Core of the Definitive Gaming Platform

Tesla K20 Family : World's Fastest Accelerator
>1TFlop Perf in under 225W

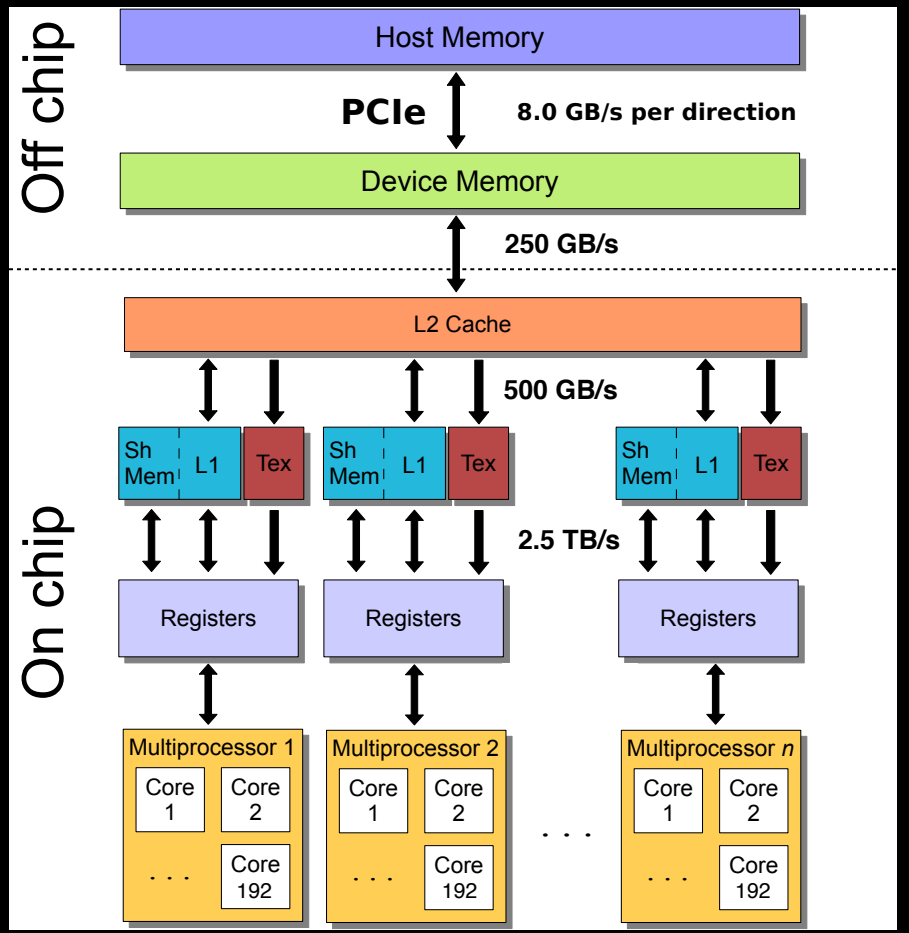
Tesla K20X





	Tesla K20X	Tesla K20
# CUDA Cores	2688	2496
Peak Double Precision Peak DGEMM	1.32 TF 1.22 TF	1.17 TF 1.10 TF
Peak Single Precision Peak SGEMM	3.95 TF 2.90 TF	3.52 TF 2.61 TF
Memory Bandwidth	250 GB/s	208 GB/s
Memory size	6 GB	5 GB
Total Board Power	235W	225W

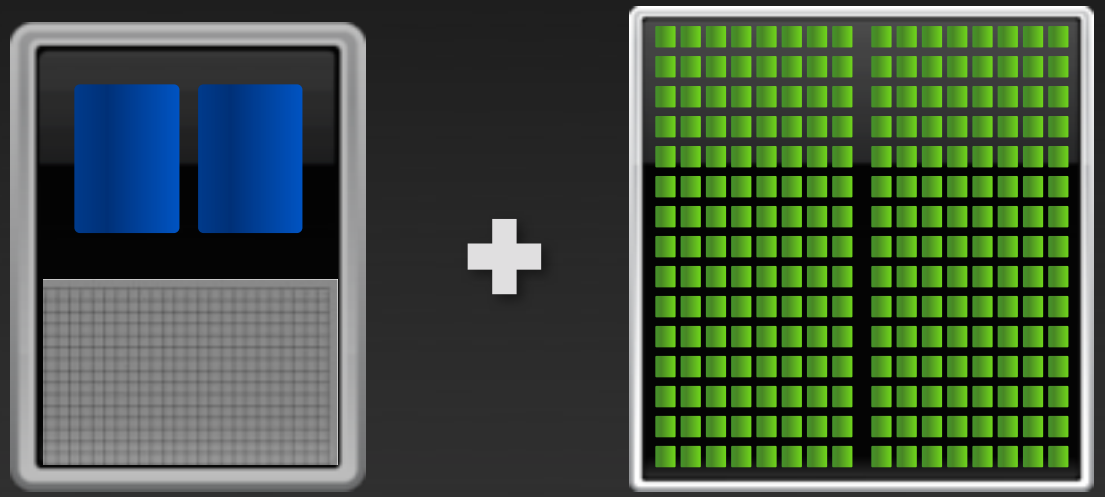
The Kepler Architecture



- 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

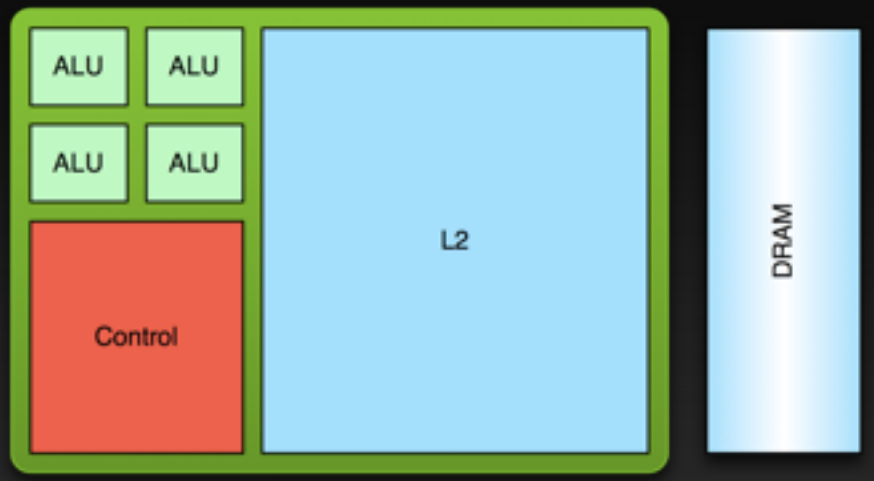
GPGPU Revolutionizes Computing

Latency Processor + Throughput processor



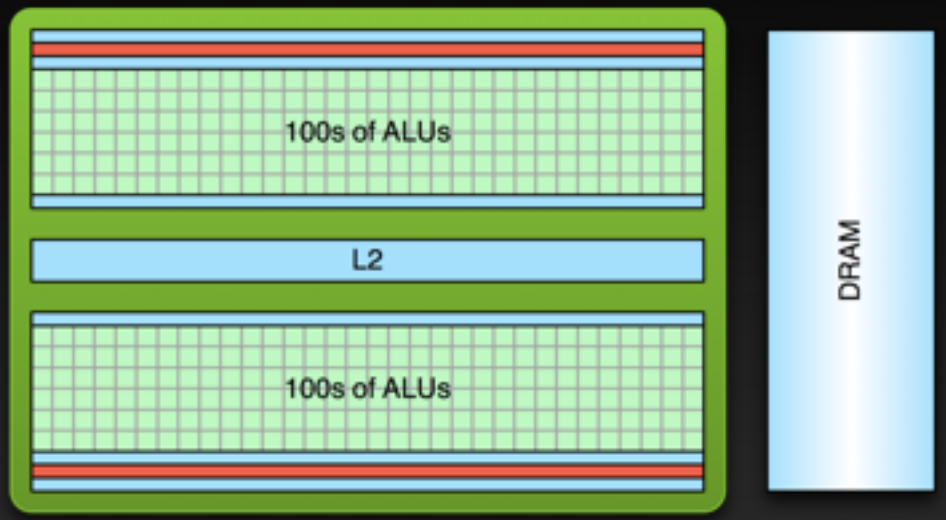
CPU GPU

Low Latency or High Throughput?



CPU

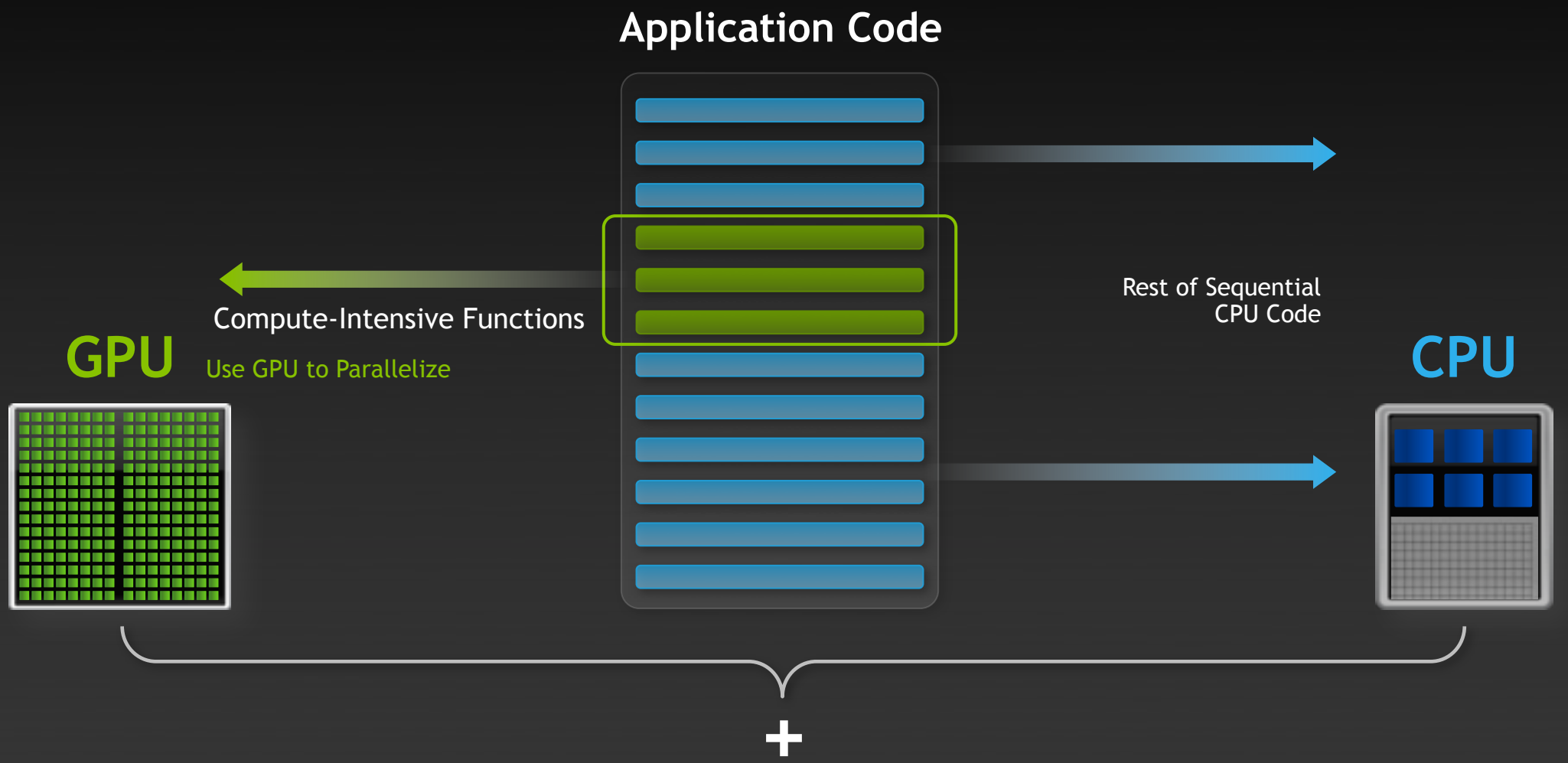
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution



GPU

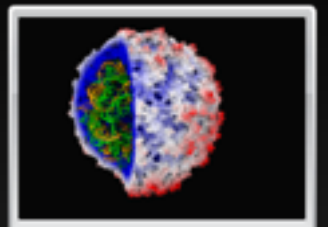
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

Small Changes, Big Speed-up



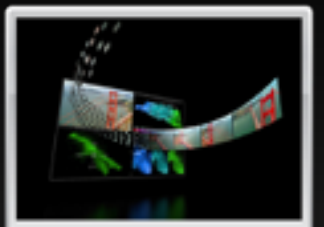
146X

Medical Imaging
U of Utah



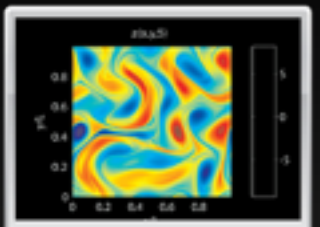
36X

Molecular Dynamics
U of Illinois, Urbana



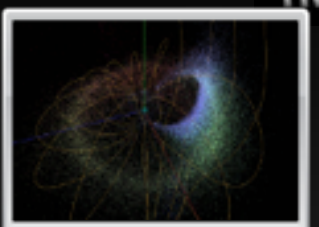
18X

Video Transcoding
Elemental Tech



50X

Matlab Computing
AccelerEyes



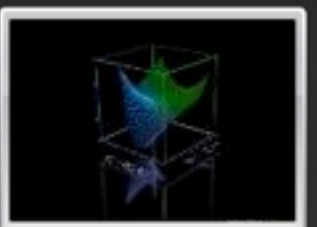
100X

Astrophysics
RIKEN

GPUs Accelerate Science

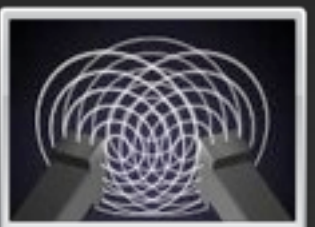
149X

Financial Simulation
Oxford



47X

Linear Algebra
Universidad Jaime

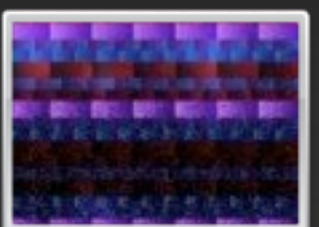


20X

3D Ultrasound
Techniscan

130X

Quantum Chemistry
U of Illinois, Urbana



30X

Gene Sequencing
U of Maryland

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC Directives

Programming Languages
(C/C++, Fortran, Python, …)

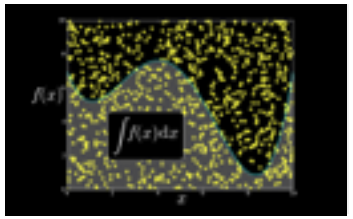
"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Performance

GPU Accelerated Libraries
"Drop-in" Acceleration for your Applications

NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

CULA | tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore

NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

CUSP

Sparse Linear Algebra

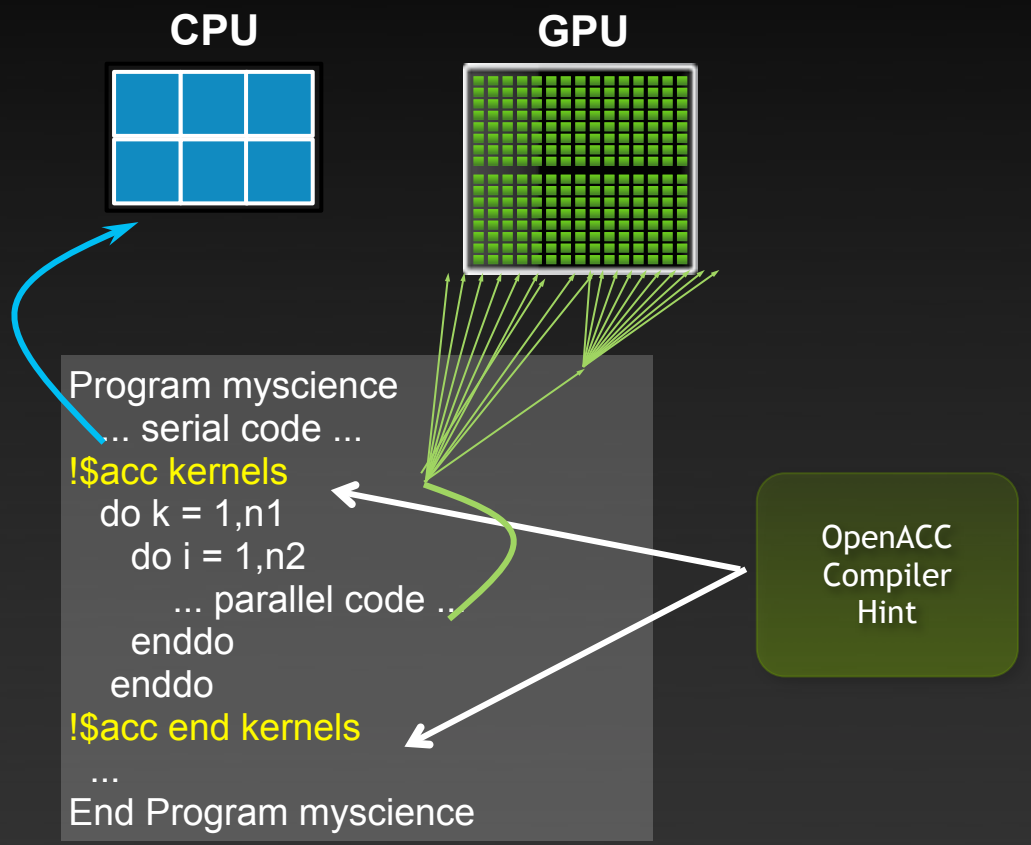
ArrayFire

Building-block Algorithms

Thrust

C++ Templated Parallel Algorithms

OpenACC Directives



Your original
Fortran or C code

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

CUDA C

Standard C

```
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);
```

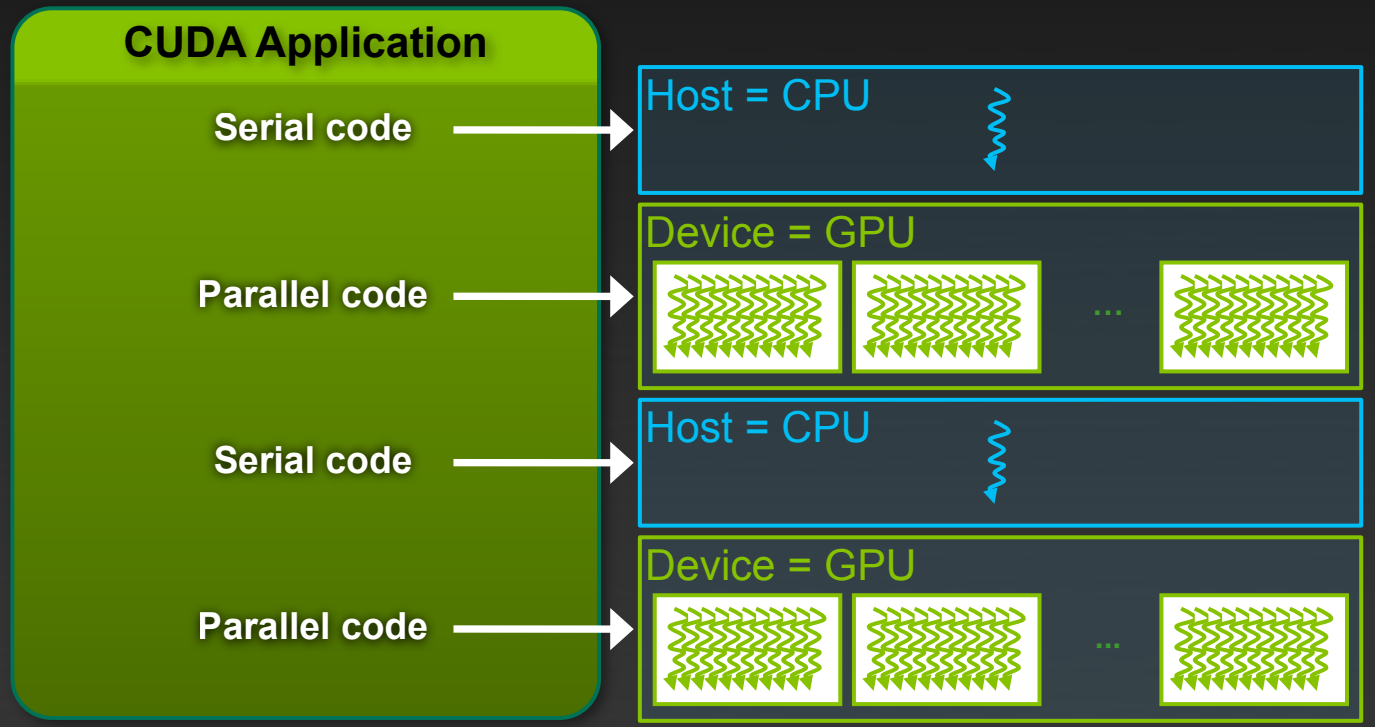
Parallel C

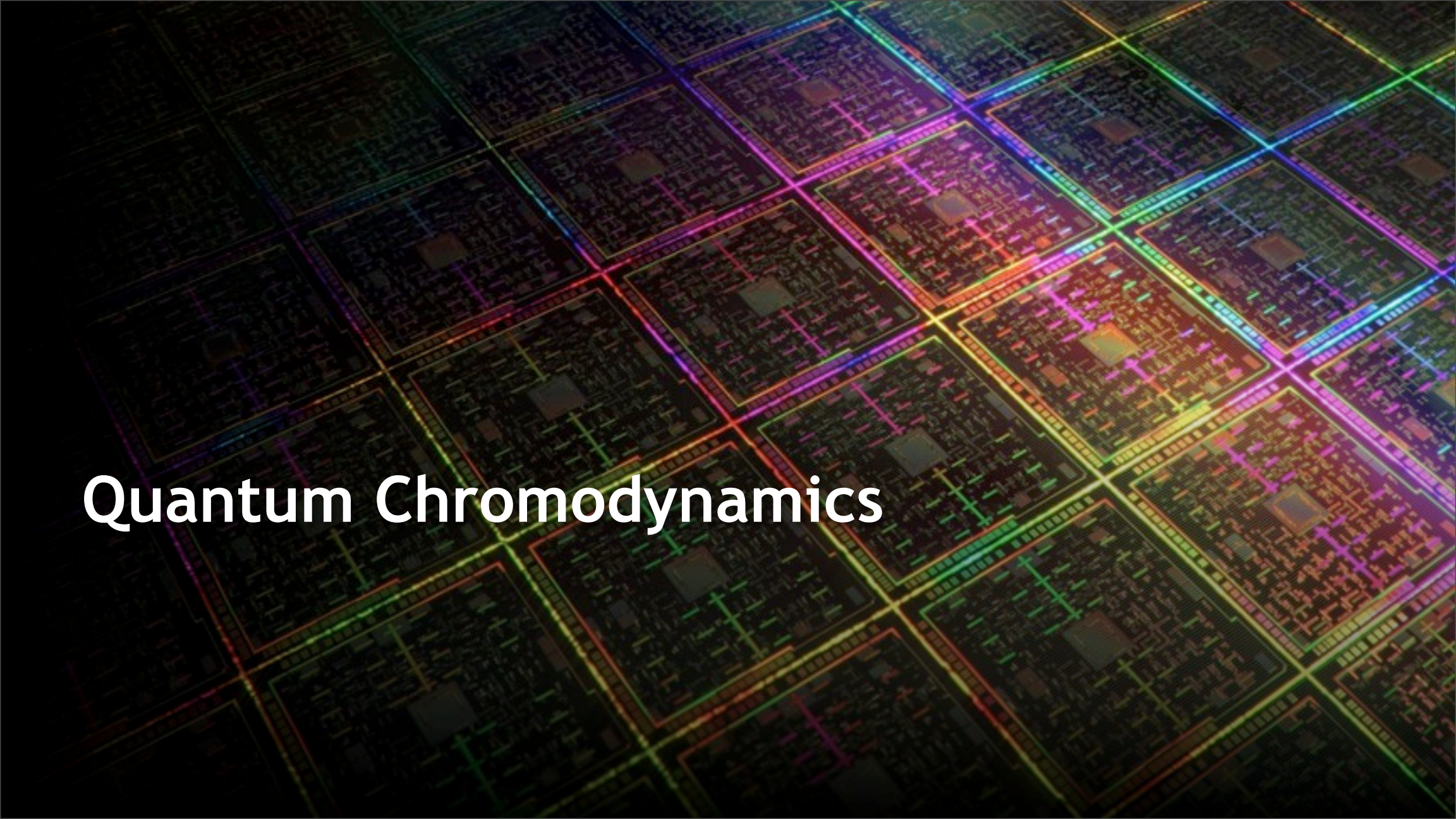
```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

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);
```

http://developer.nvidia.com/cuda-toolkit

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)



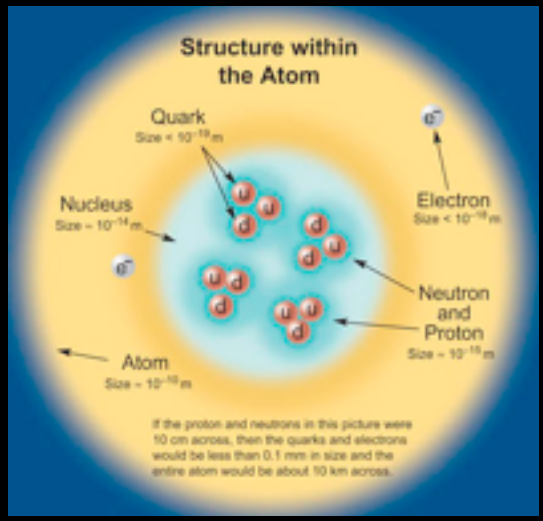

Quantum Chromodynamics

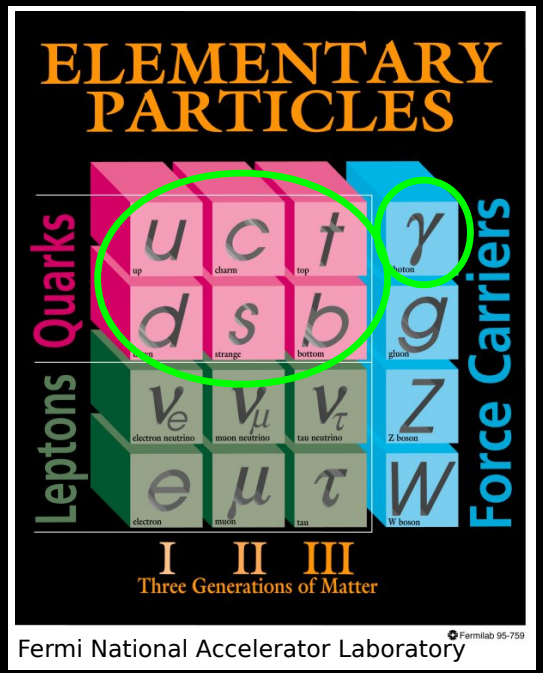
Quantum Chromodynamics

- 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...



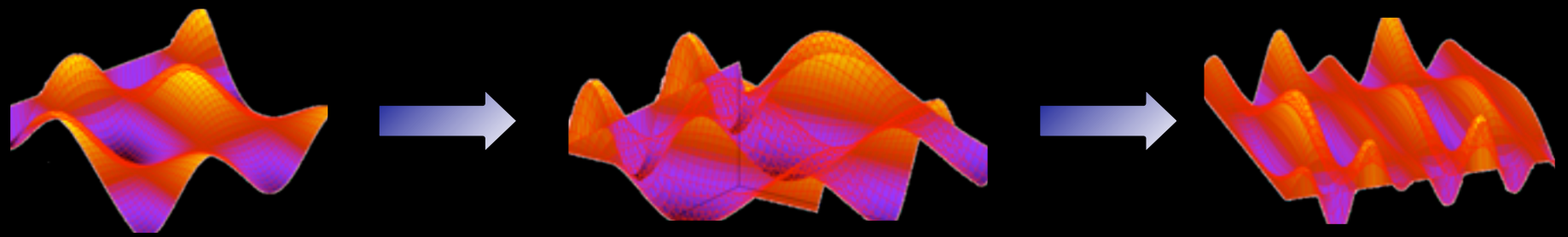


Lattice Quantum Chromodynamics

- Theory is highly non-linear ⇒ 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

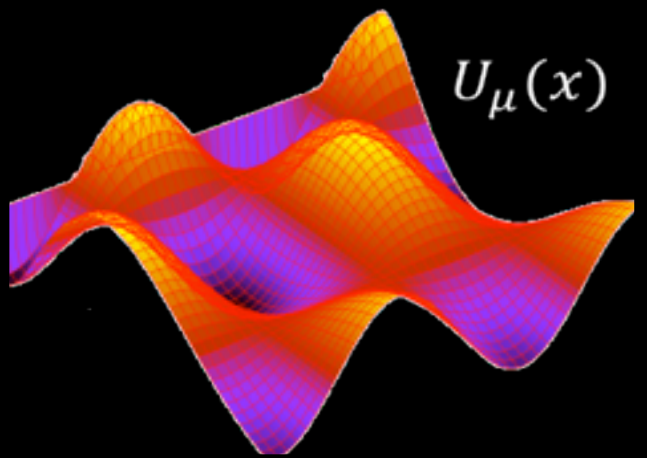
Steps in a lattice QCD calculation

1. Generate an ensemble of gluon field (“gauge”) configurations
   - Produced in sequence, with hundreds needed per ensemble
   - Strong scaling required with O(10-100 Tflops) sustained for several months (traditionally Crays, Blue Genes, etc.)
   - 50-90% of the runtime is in the linear solver



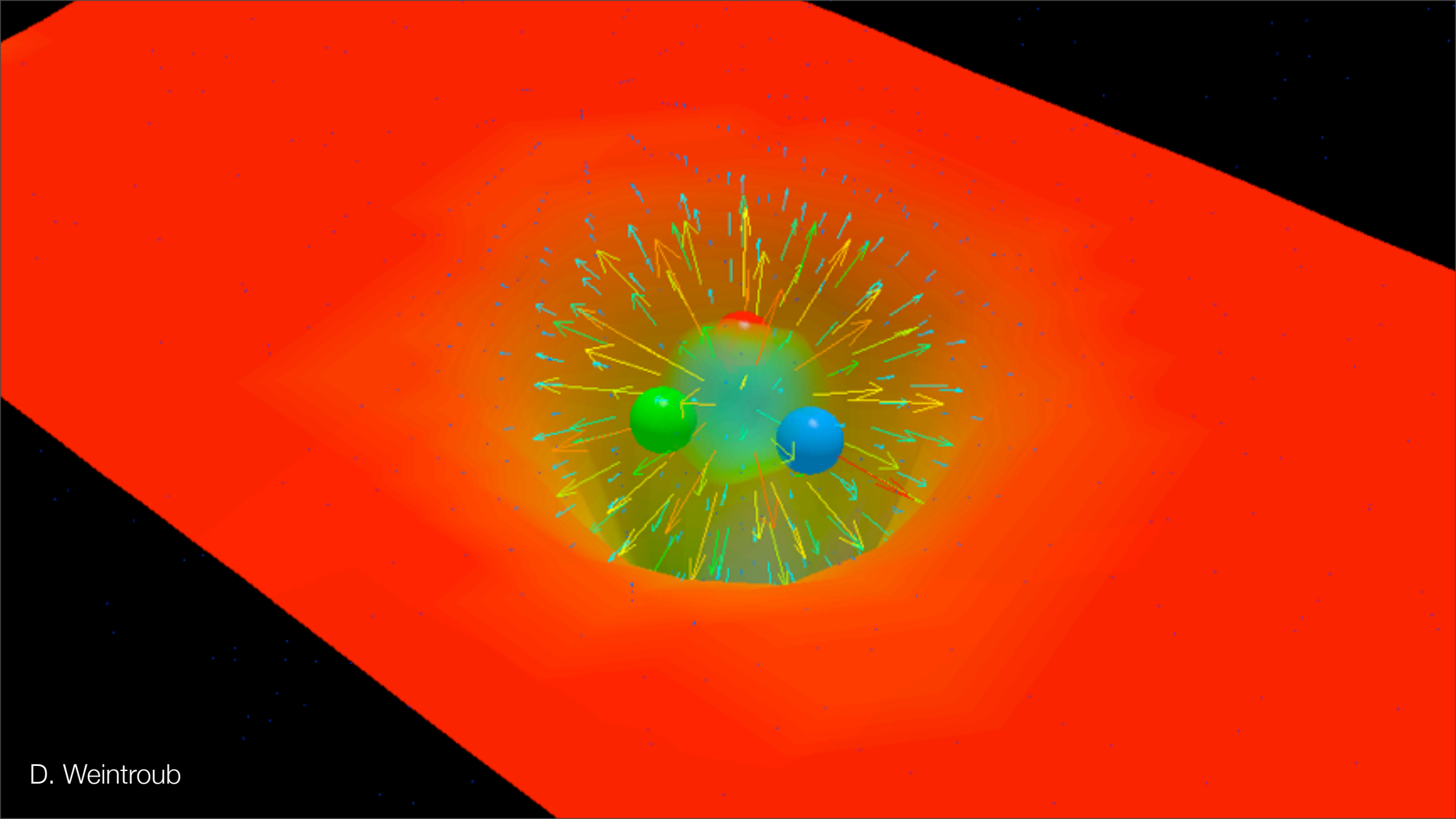
Steps in a lattice QCD calculation

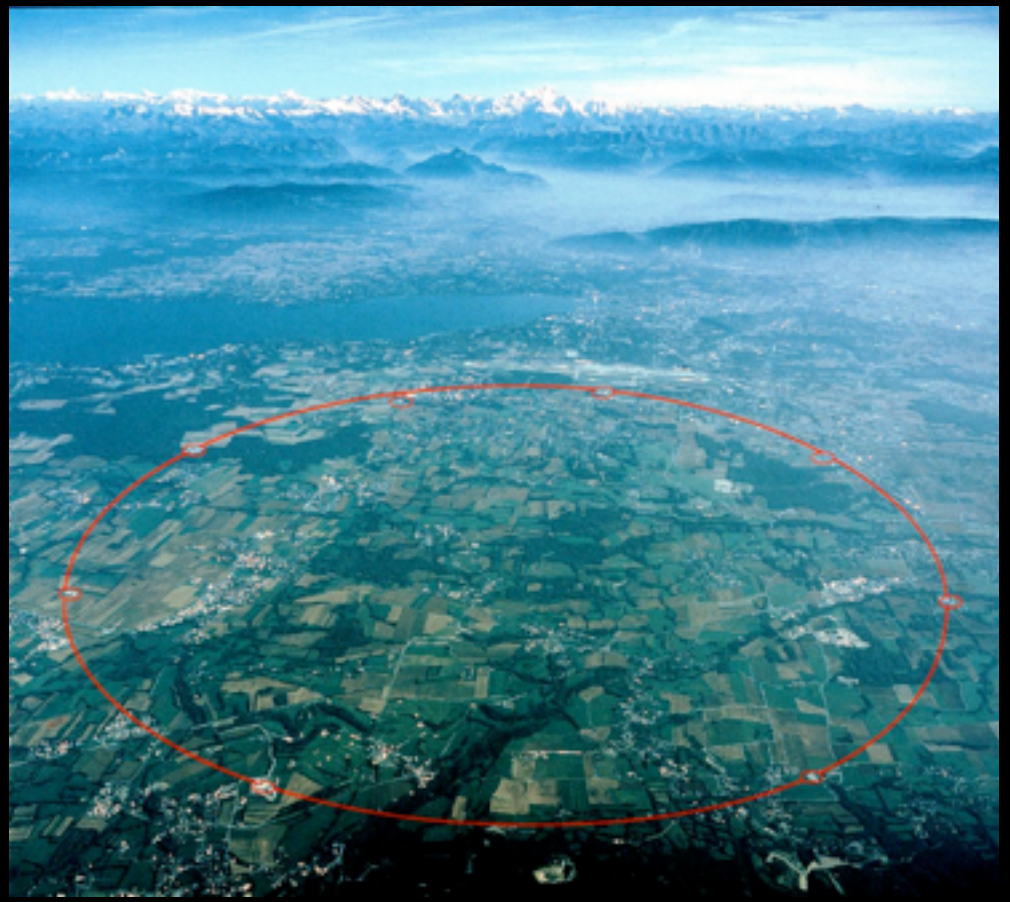
2. “Analyze” the configurations
   - Can be farmed out, assuming O(1 Tflops) per job.
   - 80-99% of the runtime is in the linear solver
     Task parallelism means that clusters reign supreme here

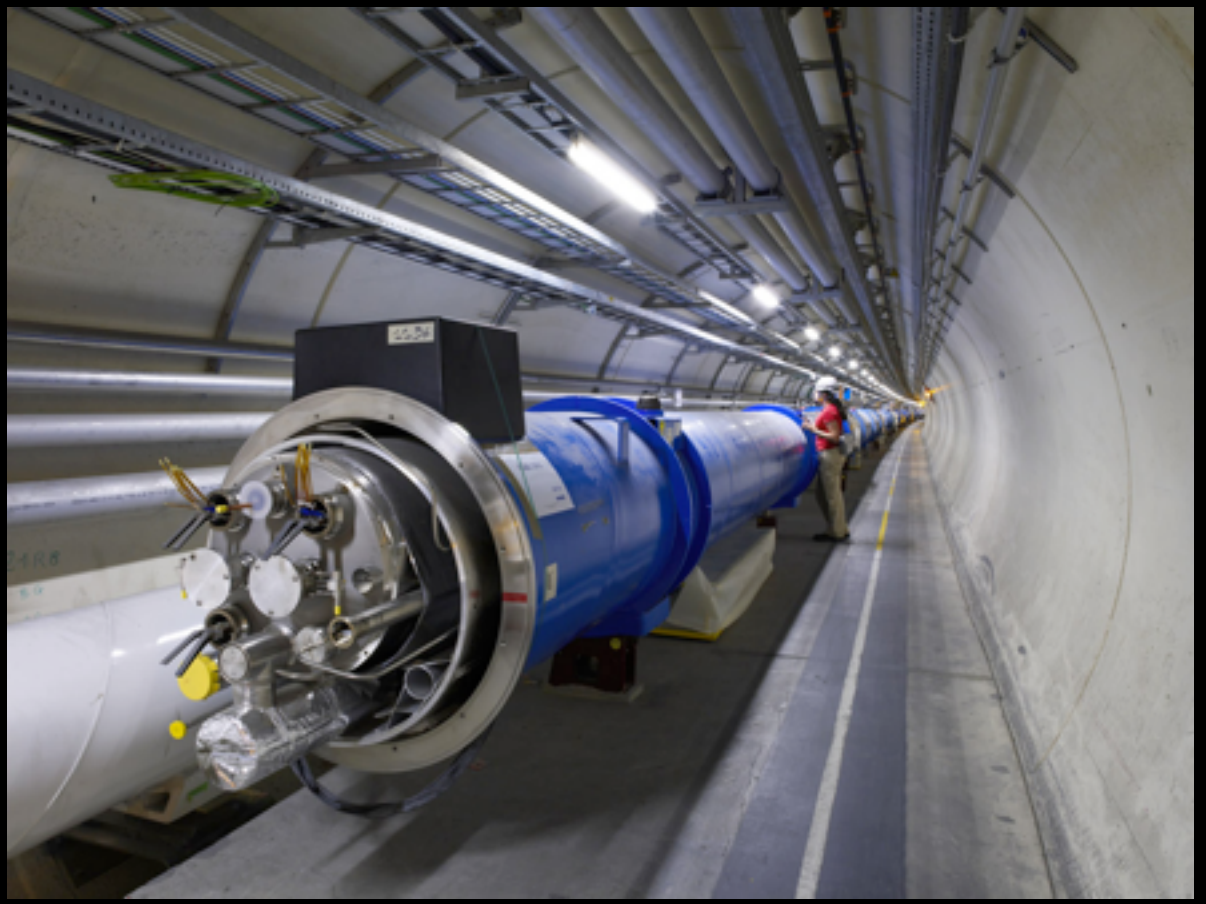


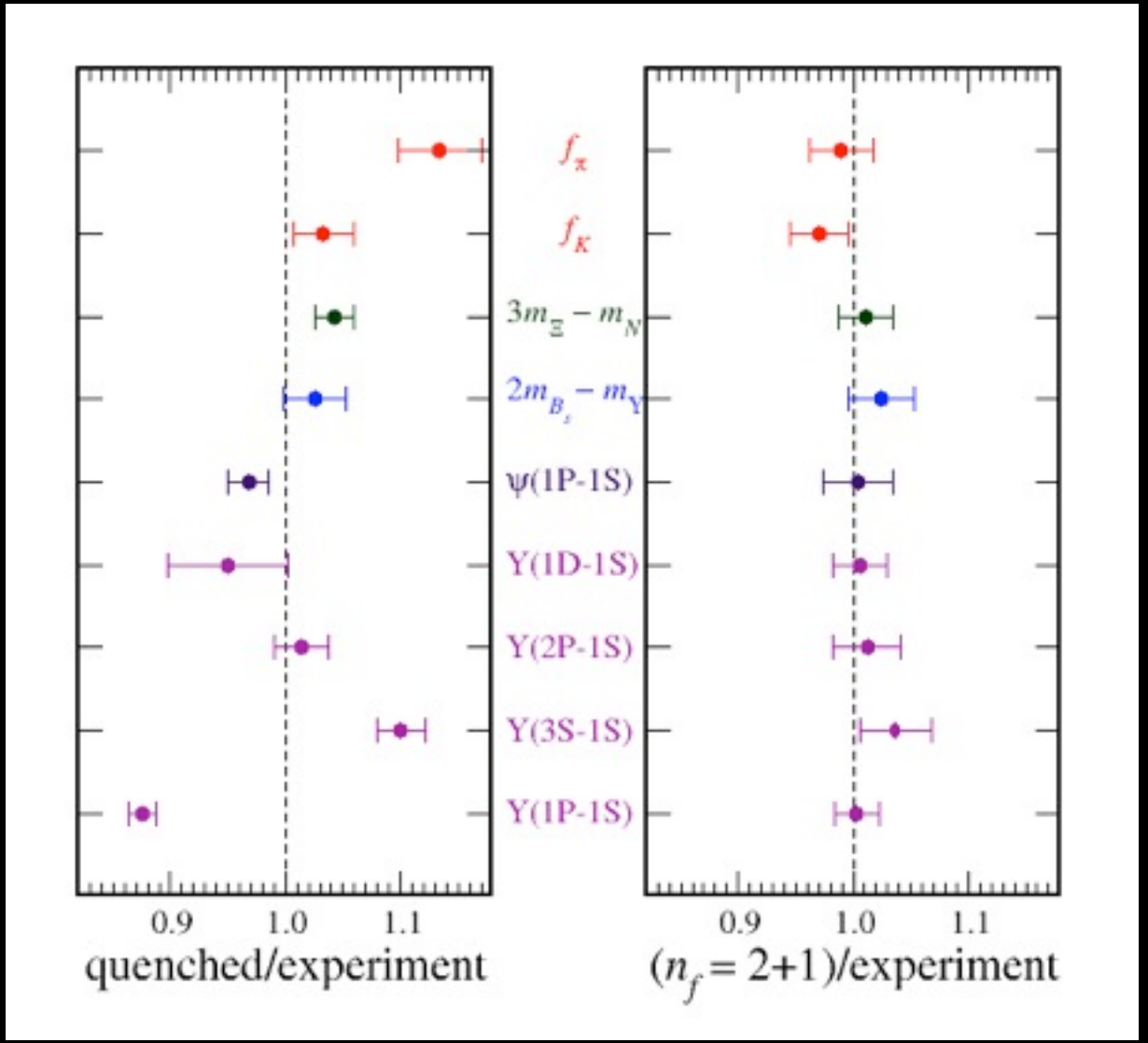
$$D_{ij}^{\alpha\beta}(x,y;U)\psi_j^\beta(y) = \eta_i^\alpha(x)$$

or “$Ax = b$”


D. Weintroub



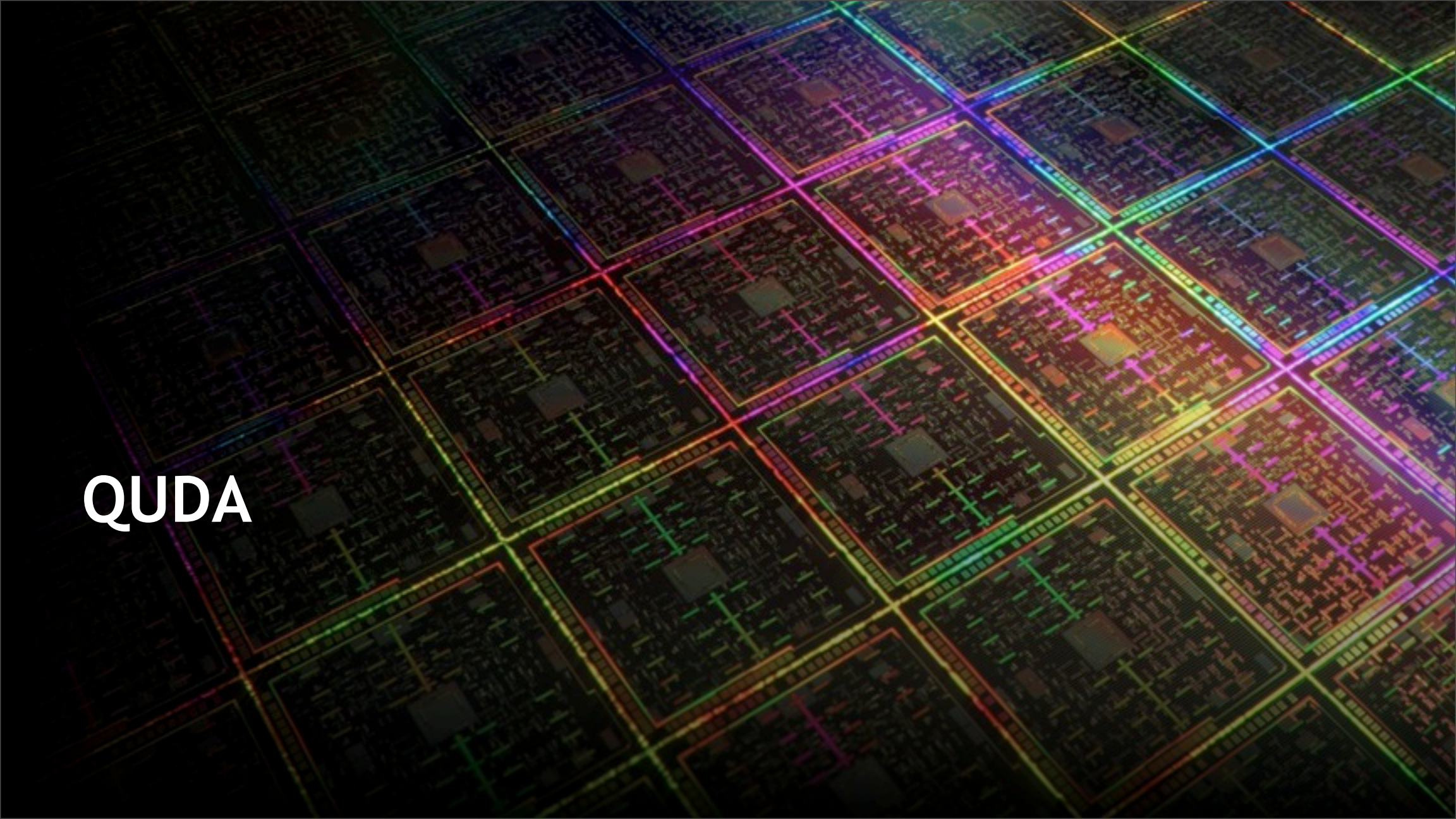



f_π
f_K
$3m_\Xi - m_N$
$2m_{B_s} - m_\Upsilon$
ψ(1P-1S)
Υ(1D-1S)
Υ(2P-1S)
Υ(3S-1S)
Υ(1P-1S)
0.9
1.0
1.1
quenched/experiment
0.9
1.0
1.1
(n_f = 2+1)/experiment

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)

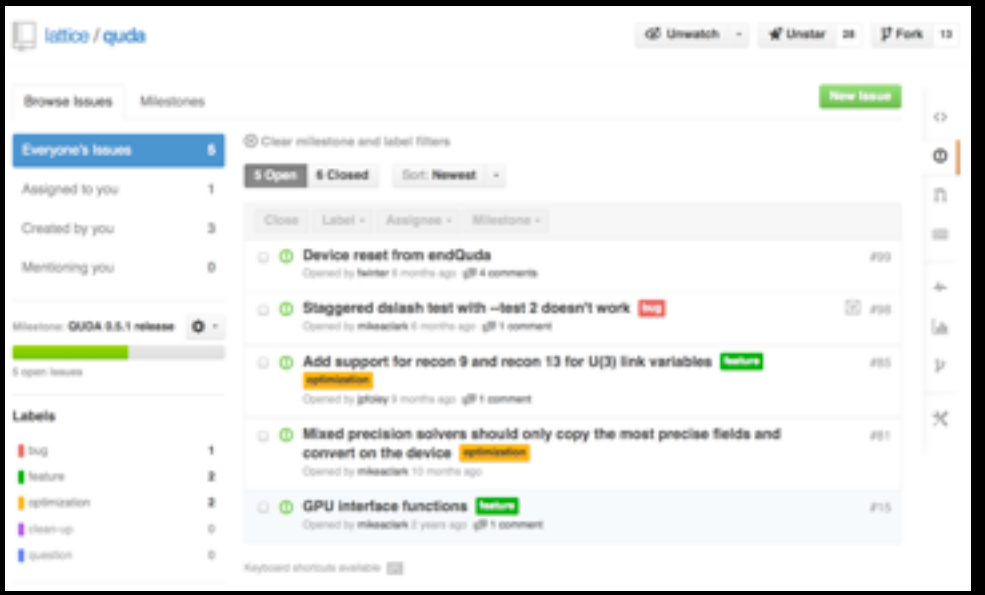

QUDA

Enter QUDA

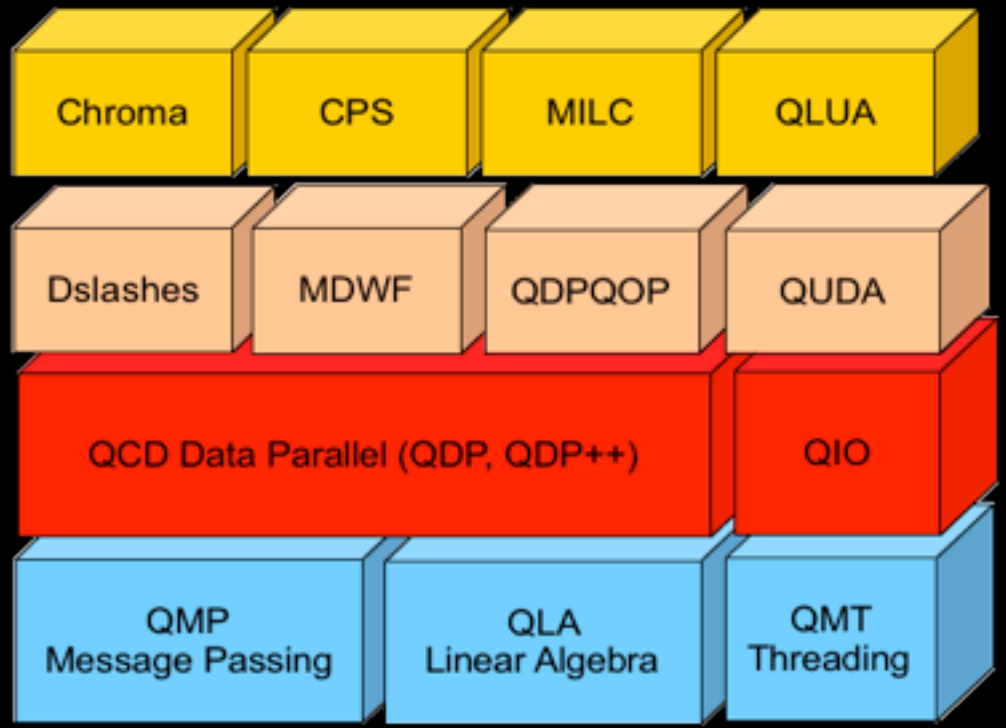
- "QCD on CUDA" – http://lattice.github.com/quda
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, etc.
- Provides:
  - Various solvers for several discretizations, including multi-GPU support and domain-decomposed (Schwarz) preconditioners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

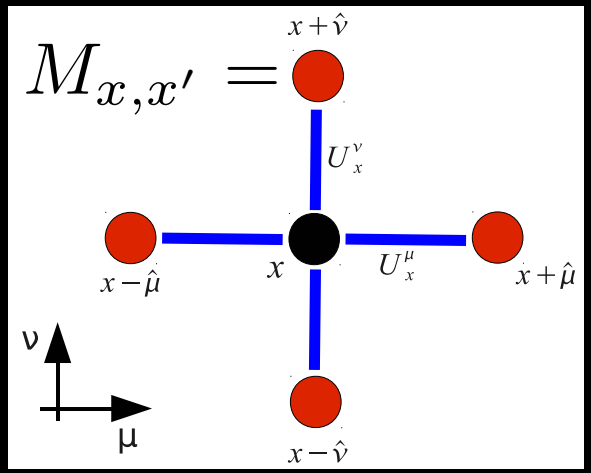
- QUDA default interface provides a simple view for the outside world
  - C or Fortran
  - Host applications simply pass cpu-side pointers
  - QUDA takes care of all field reordering and data copying
  - No GPU code in user application
- Limitations
  - No control over memory management
  - Data residency between QUDA calls not possible
  - QUDA might not support user application field order

```
#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

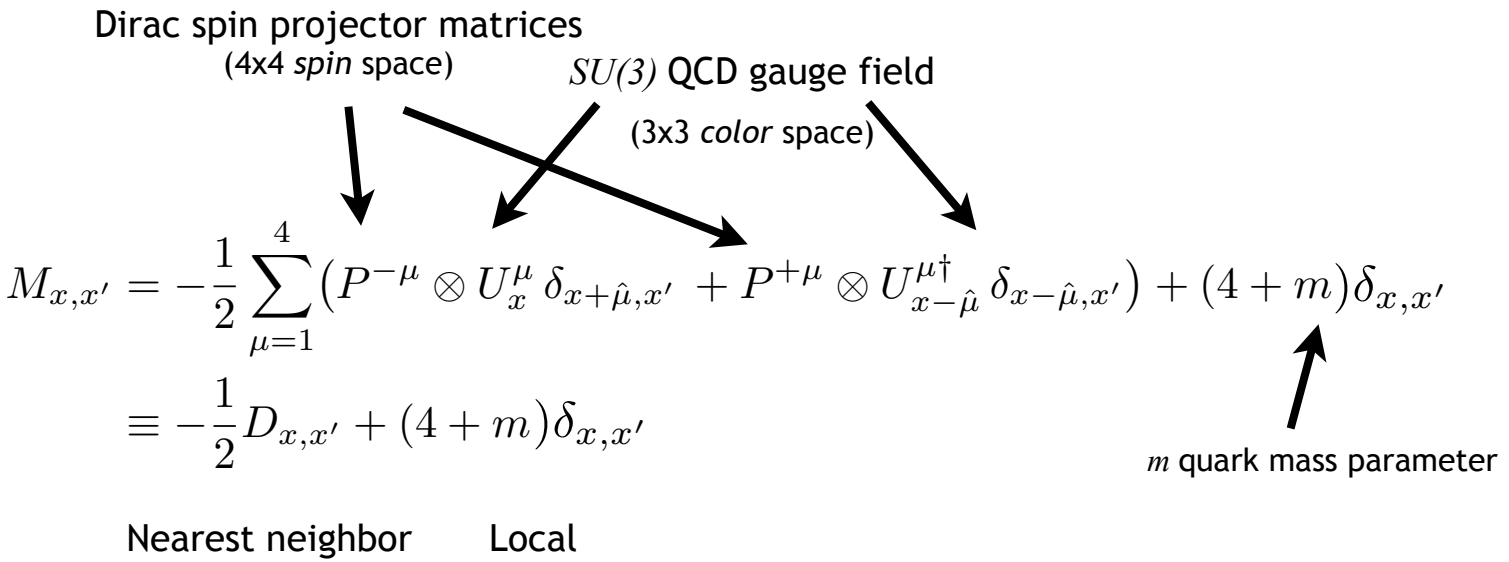
- QUDA is
  - a library enabling legacy applications to run on GPUs
  - evolving
    - more features
    - cleaner, easier to maintain
  - a research tool into how to reach the exascale
    - Lessons learned are mostly (platform) agnostic
    - Domain-specific knowledge is key
    - Free from the restrictions 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 ∈ 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

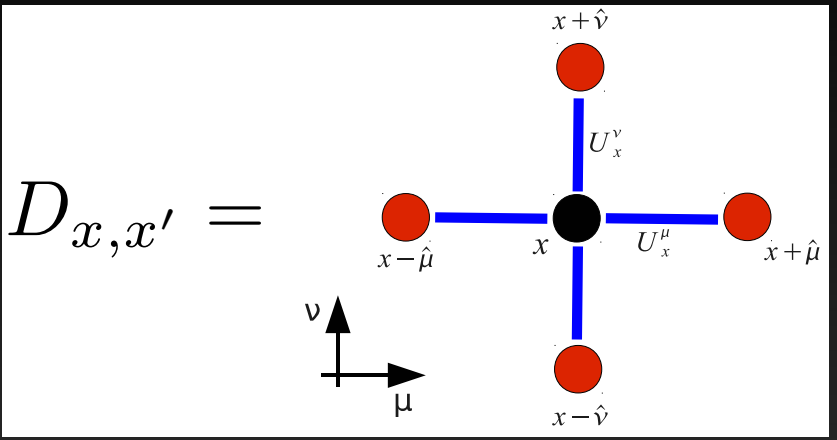
Wilson Matrix



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 = XYZT threads
  - V = 24^4 => 3.3×10^6 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

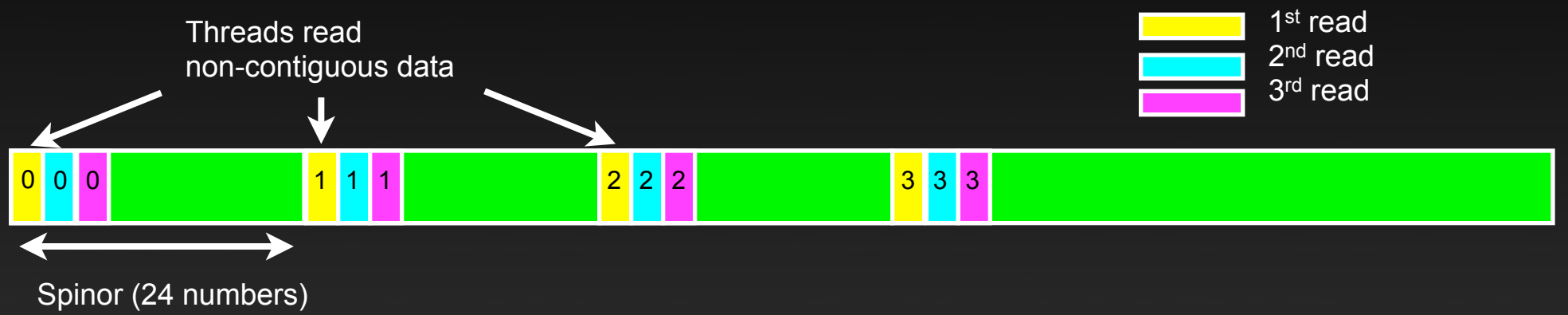


Tesla K20X	
Gflops	3995
GB/s	250
AI	16

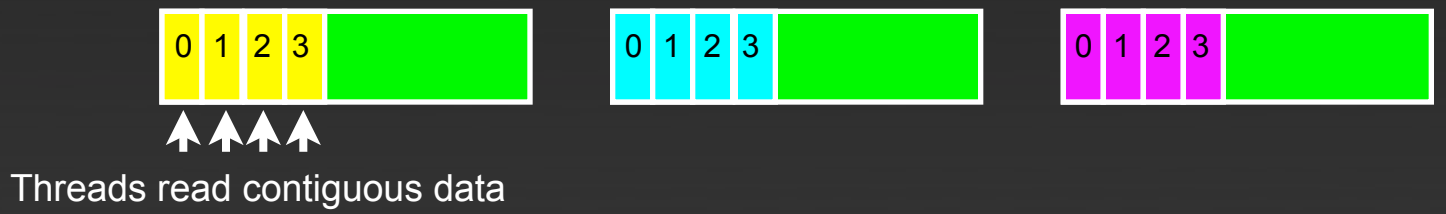
bandwidth bound

Field Ordering

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



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



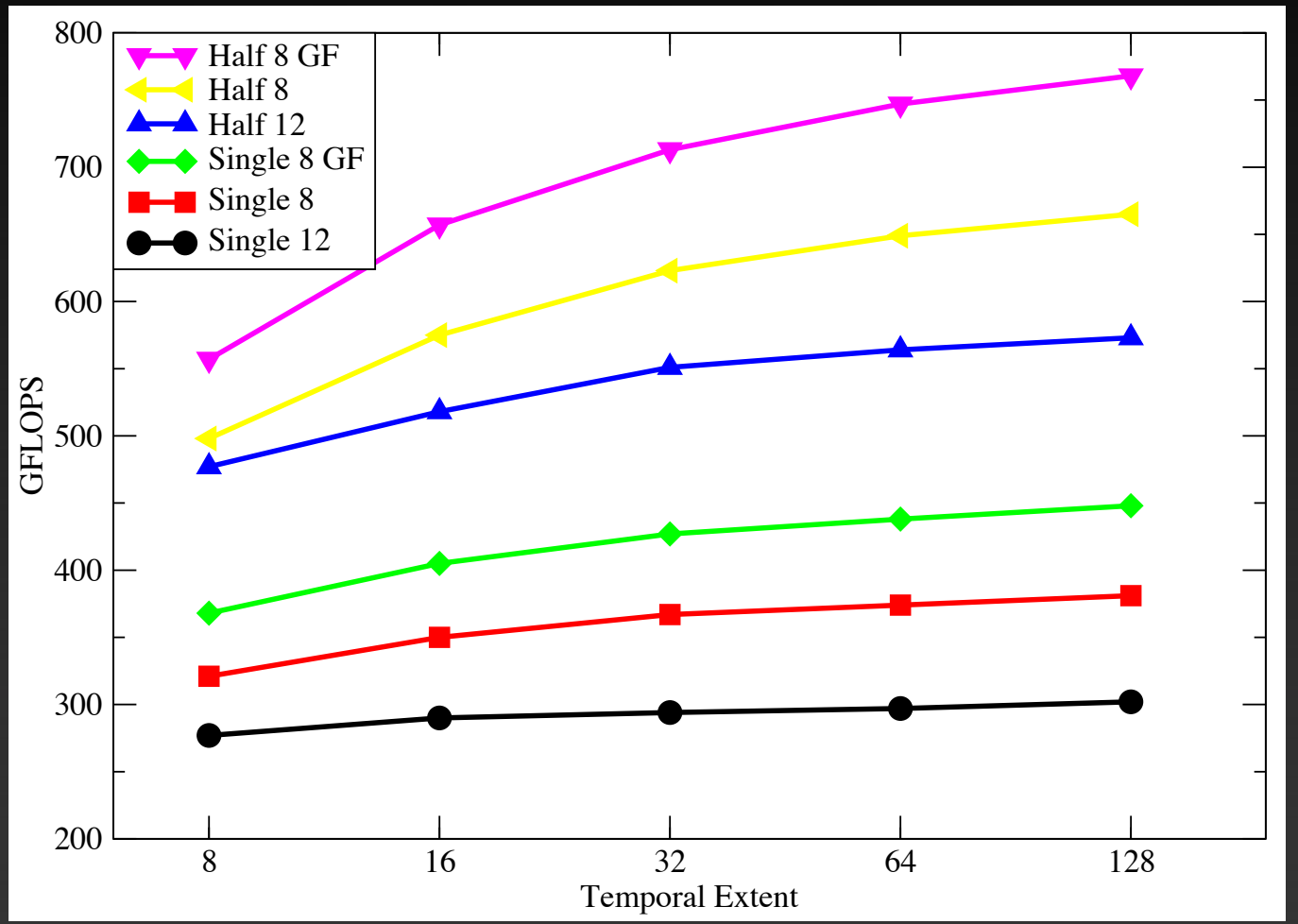
- QUDA interface deals with all data reordering
- Application remains ignorant

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

    - 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
$V = 24^3 \times T$
Wilson-Clover is ±10%

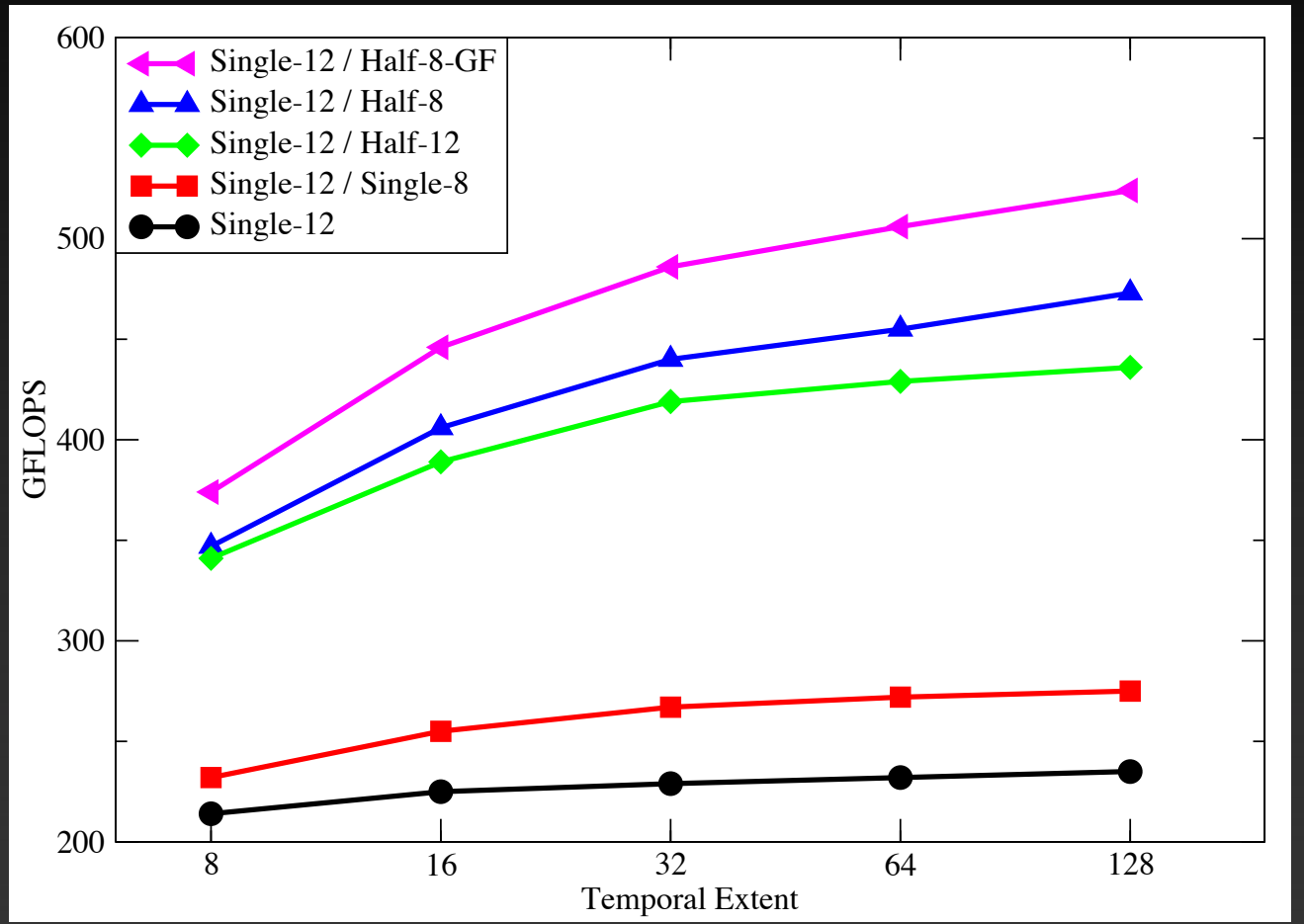
Krylov Solver Implementation

- 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

conjugate gradient

$$
\begin{aligned}
&\text{while } (|\mathbf{r}_k| > \varepsilon) \ \{ \\
&\quad \beta_k = (\mathbf{r}_k, \mathbf{r}_k)/(\mathbf{r}_{k-1}, \mathbf{r}_{k-1}) \\
&\quad \mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k \mathbf{p}_k \\
&\quad \\
&\quad \alpha = (\mathbf{r}_k, \mathbf{r}_k)/(\mathbf{p}_{k+1}, A\mathbf{p}_{k+1}) \\
&\quad \mathbf{r}_{k+1} = \mathbf{r}_k - \alpha A \mathbf{p}_{k+1} \\
&\quad \mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \mathbf{p}_{k+1} \\
&\quad k = k+1 \\
&\}
\end{aligned}
$$

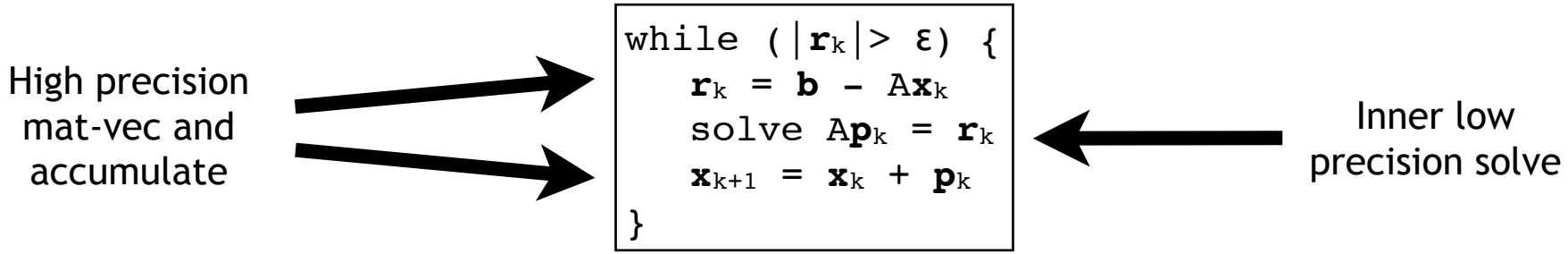
Kepler Wilson-Solver Performance



K20X CG performance
$V = 24^3 \times T$

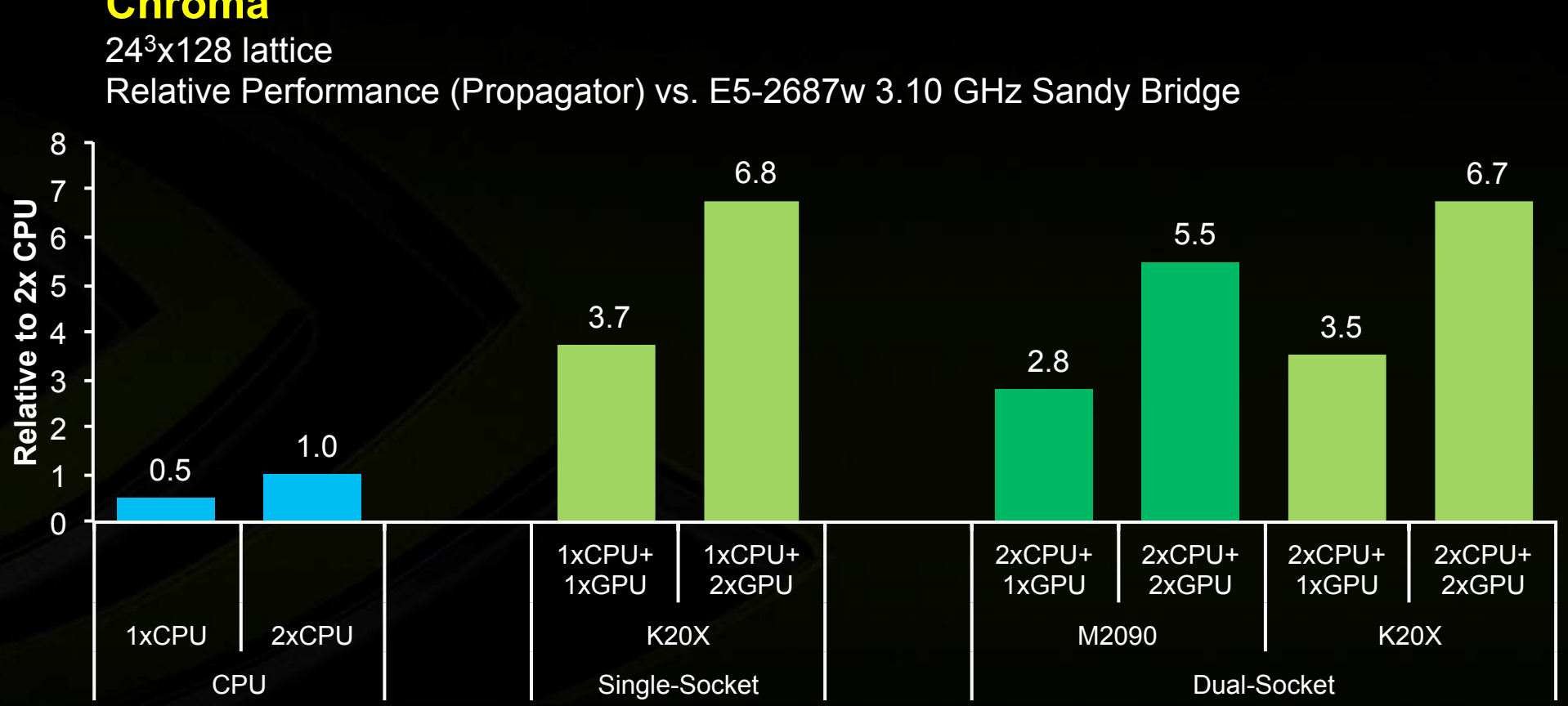
Mixed-Precision Solvers

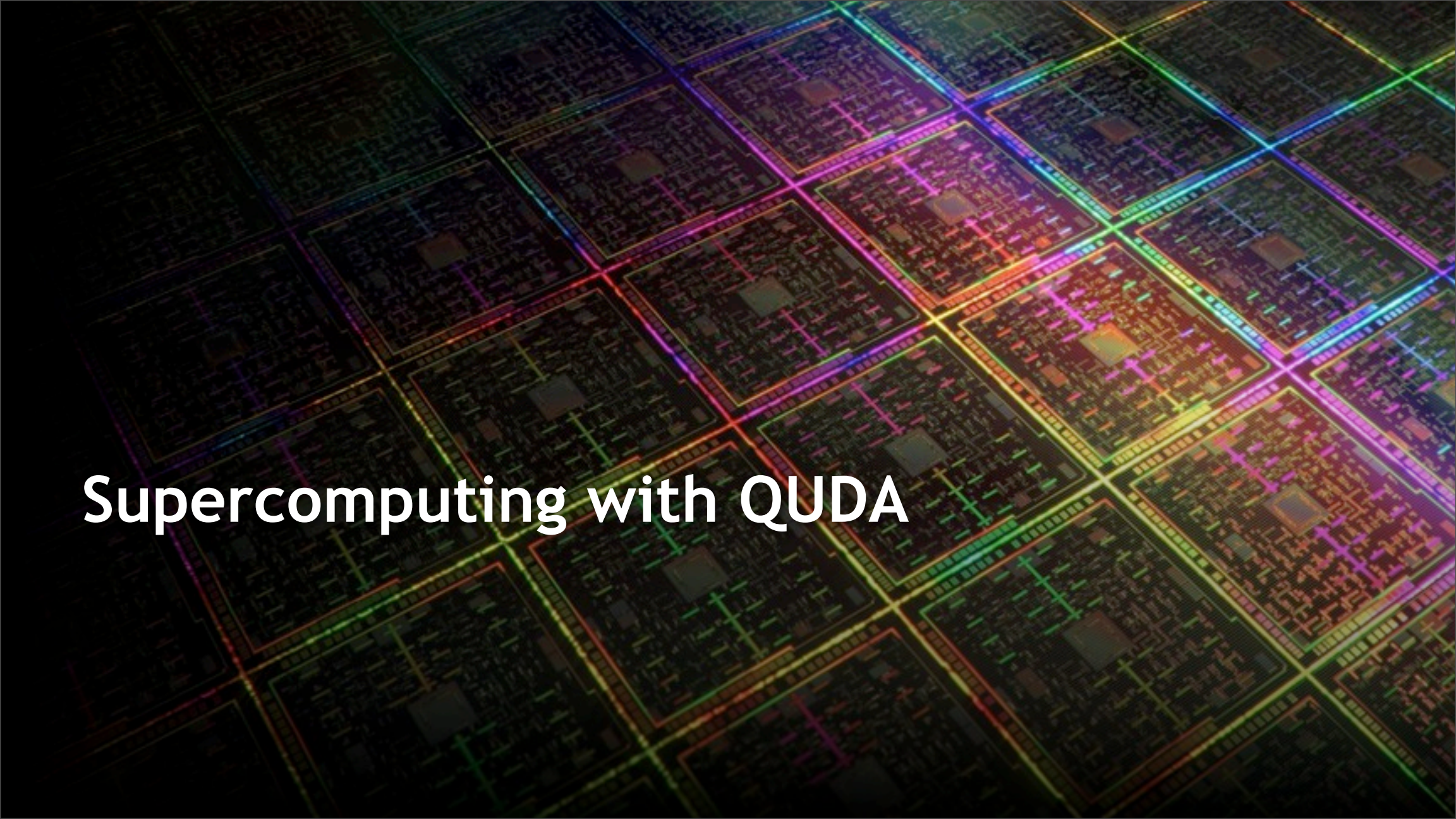
- Often require solver tolerance beyond limit of single precision
- But single and half precision much faster than double
- Use mixed precision
  - e.g. defect-correction



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

Chroma (Lattice QCD) – High Energy & Nuclear Physics




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

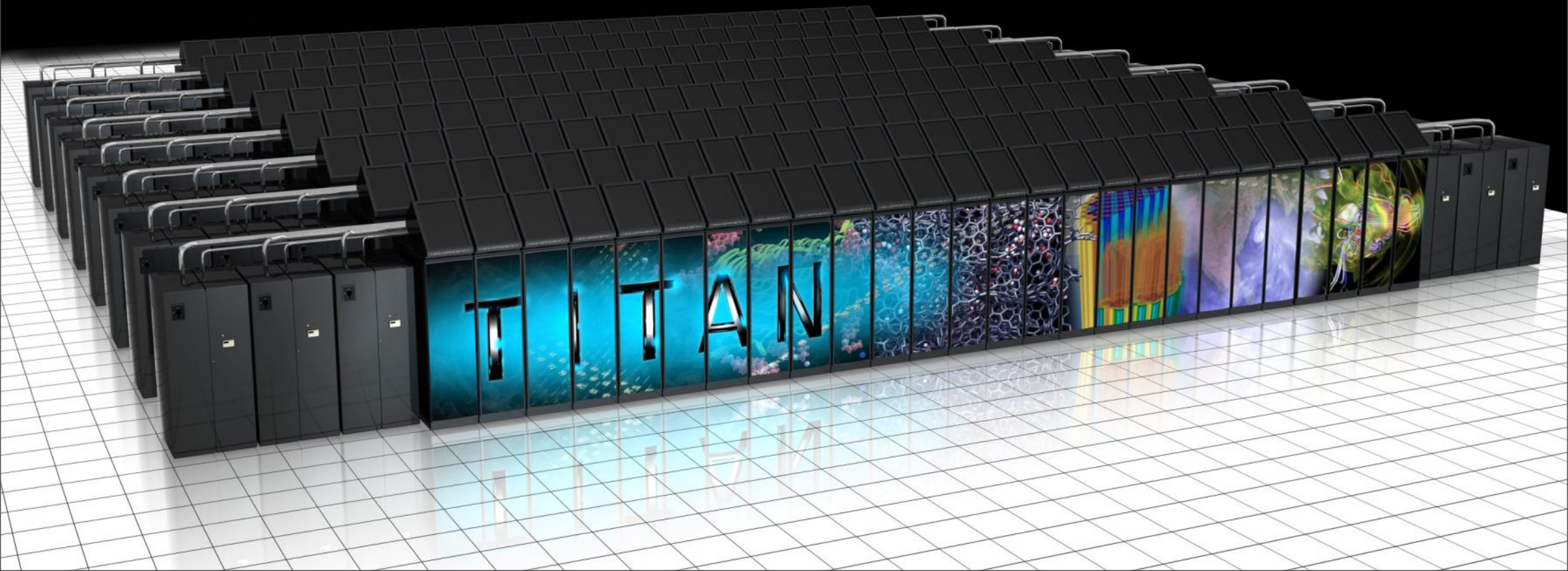
Tsubame 2.0, Tianhe 1A,
Blue Waters, etc.

TITAN: World's Most Efficient Supercomputer

18,688 Tesla K20X GPUs

27 Petaflops Peak, 17.59 Petaflops on Linpack

90% of Performance from GPUs



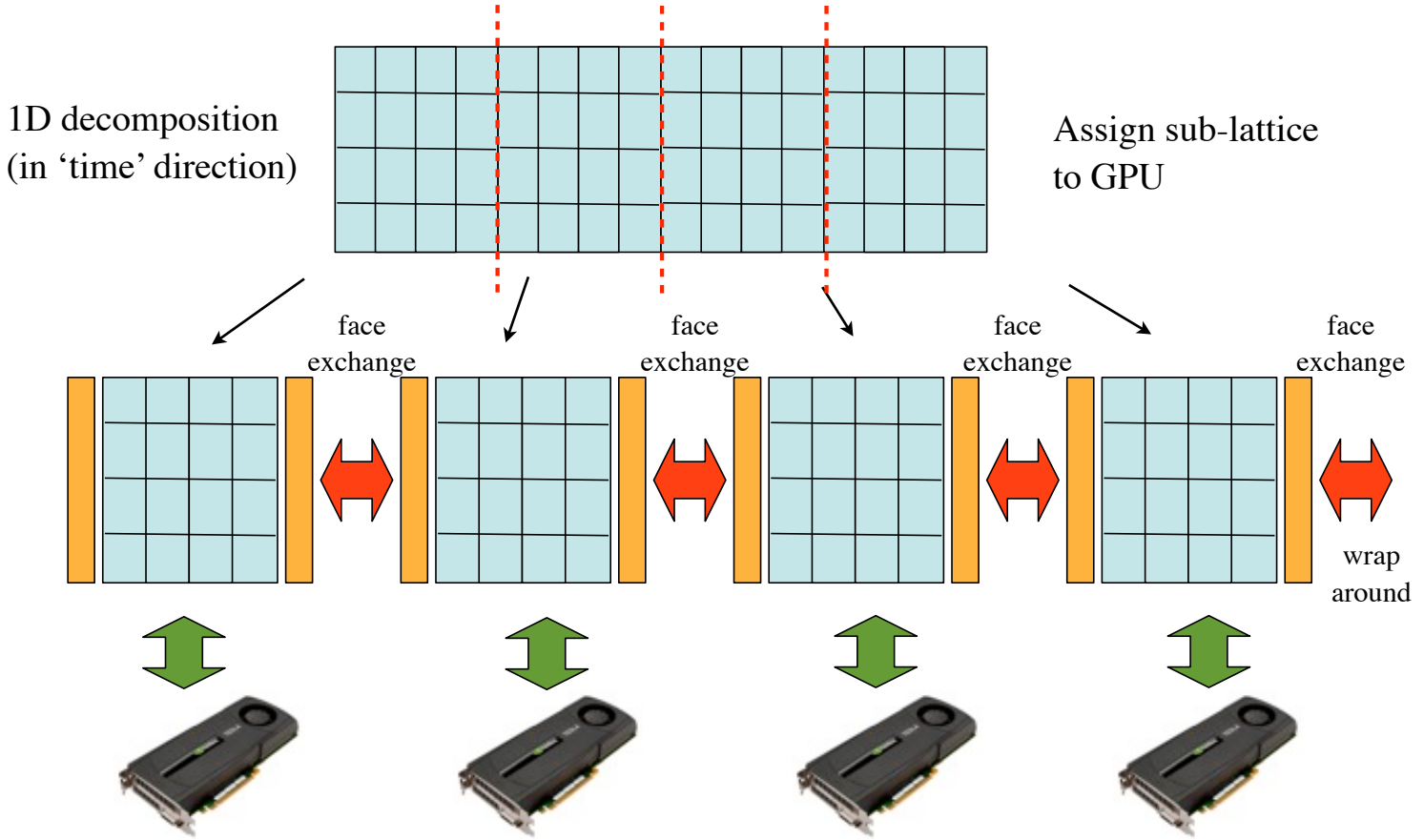
Multiple GPUs

- 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

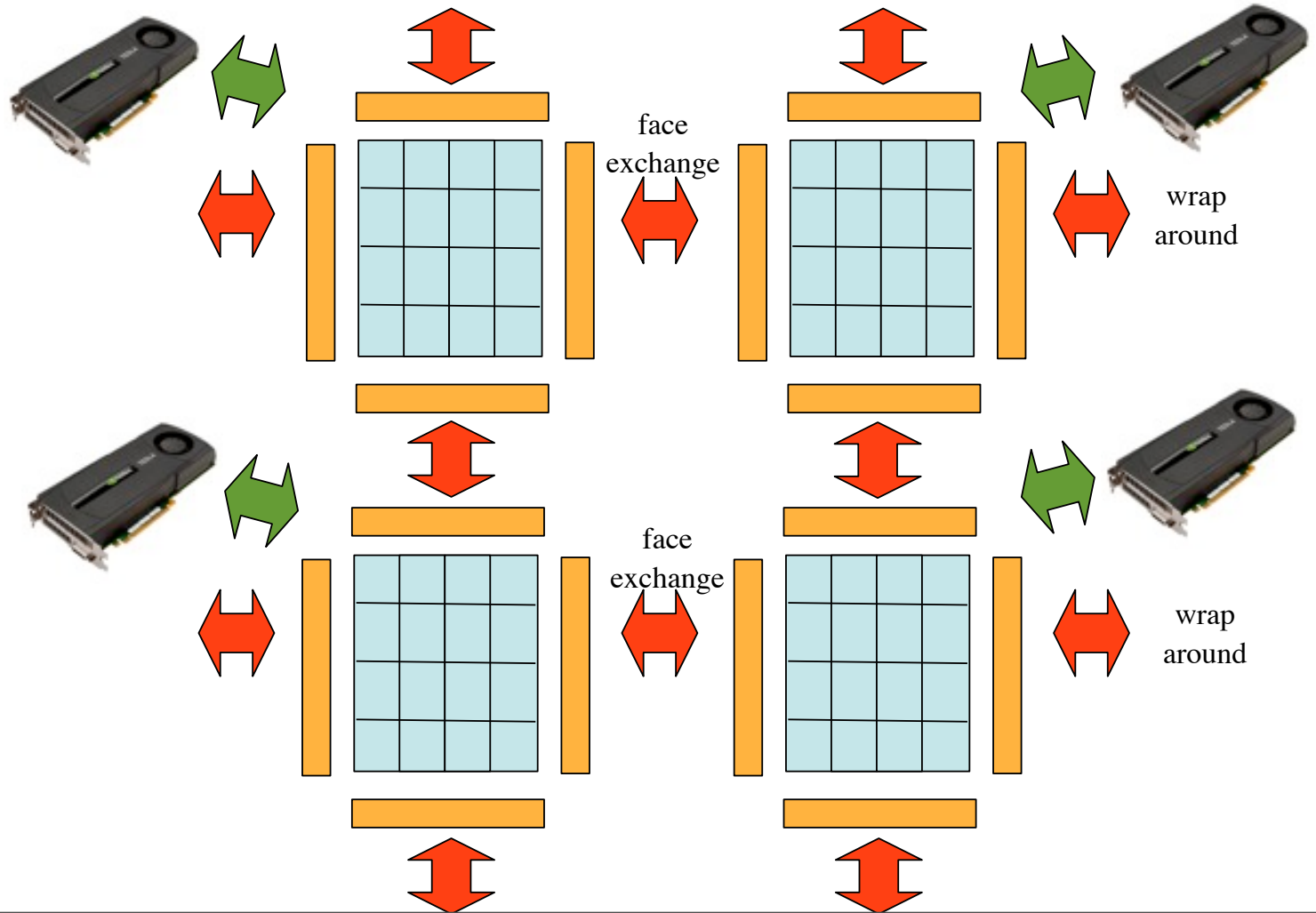
CUDA Stream API

- CUDA provides the stream API for concurrent work queues
  - Provides concurrent kernels and host<->device memcpys
  - Kernels and memcpys are queued to a stream
    - `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

1D Lattice decomposition



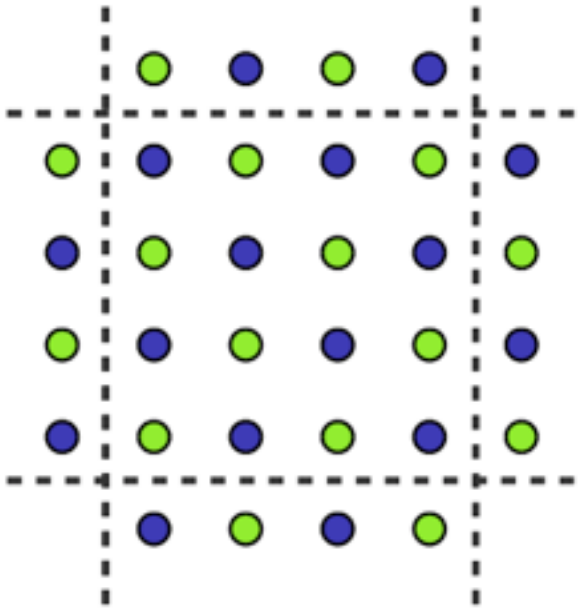
Multi-dimensional lattice decomposition



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

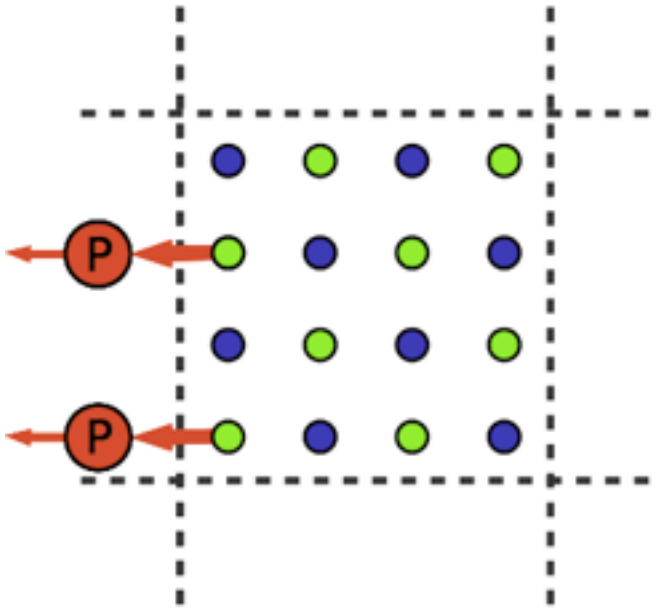
Multi-dimensional Kernel Computation



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

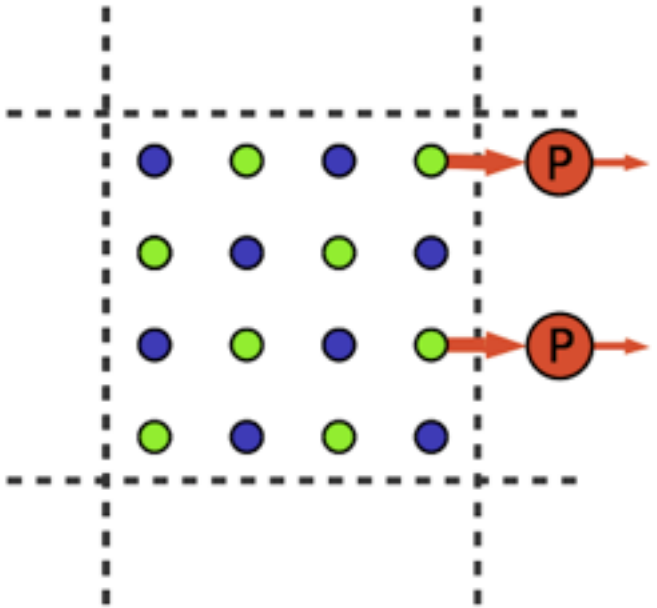
Multi-dimensional Kernel Computation



Step 1

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

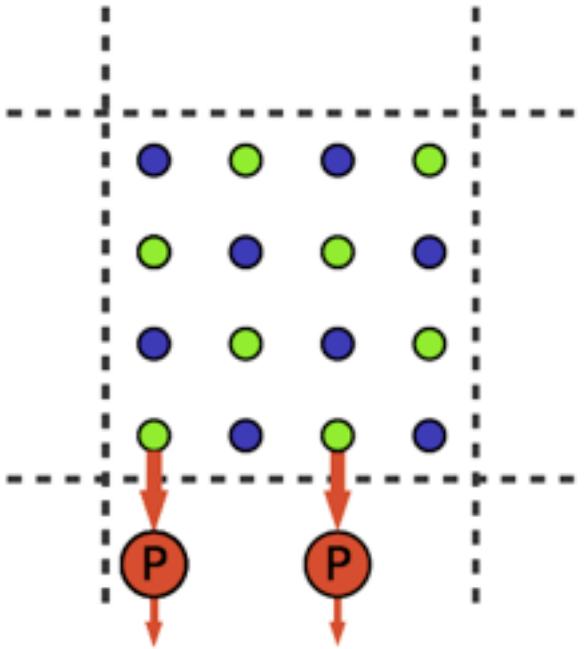
Multi-dimensional Kernel Computation



Step 1

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

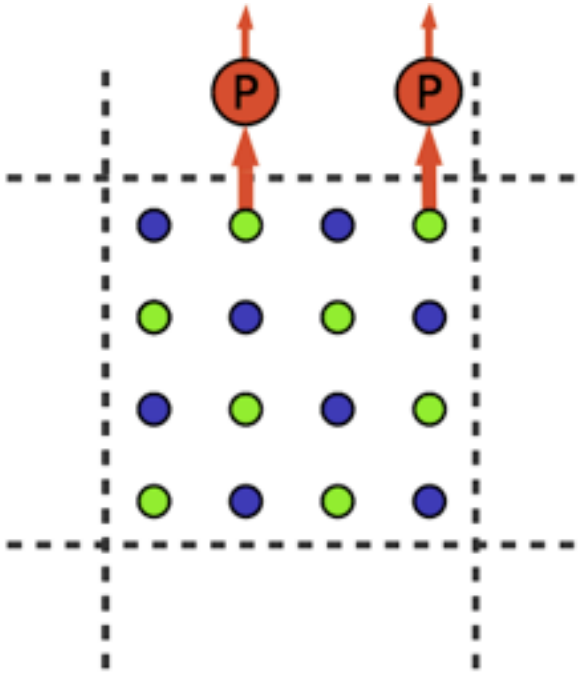
Multi-dimensional Kernel Computation



Step 1

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

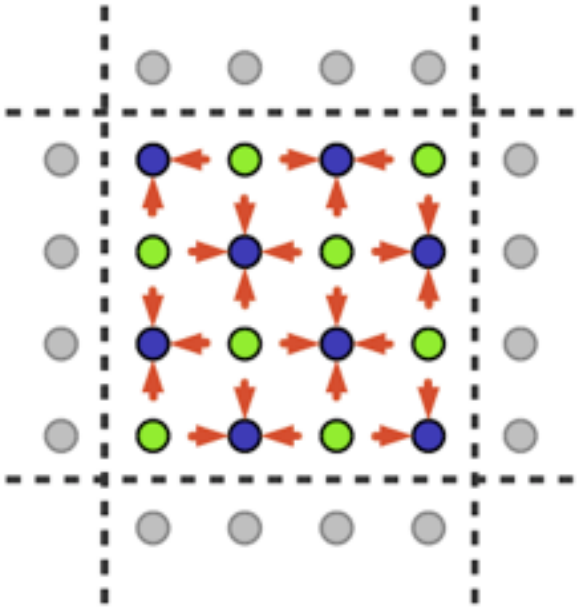
Multi-dimensional Kernel Computation



Step 1

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

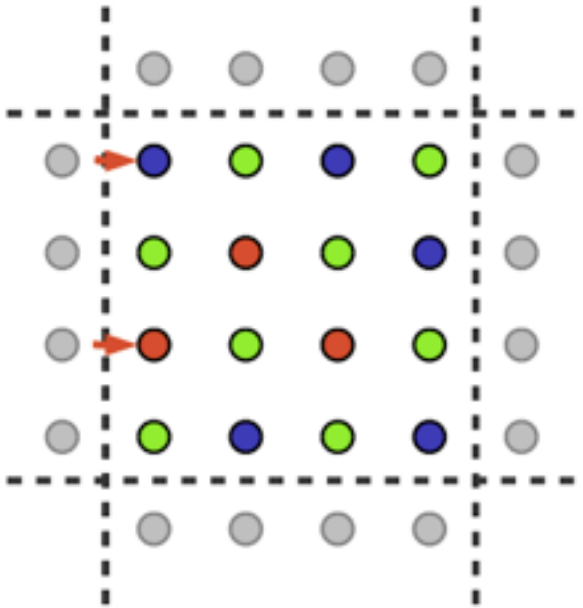
Multi-dimensional Kernel Computation



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



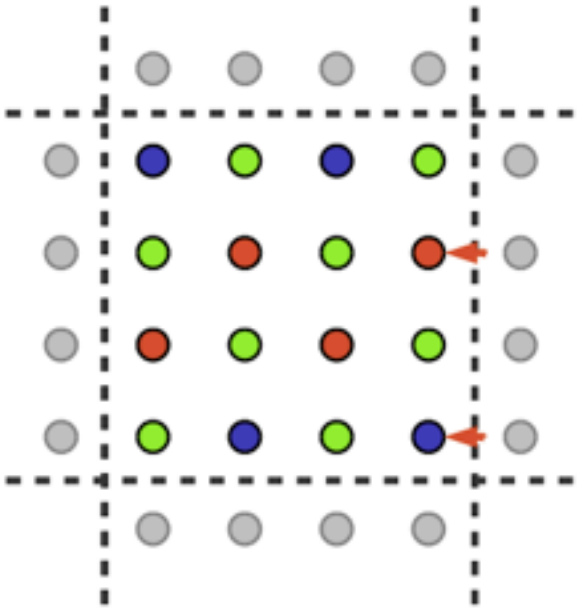
Step 3

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



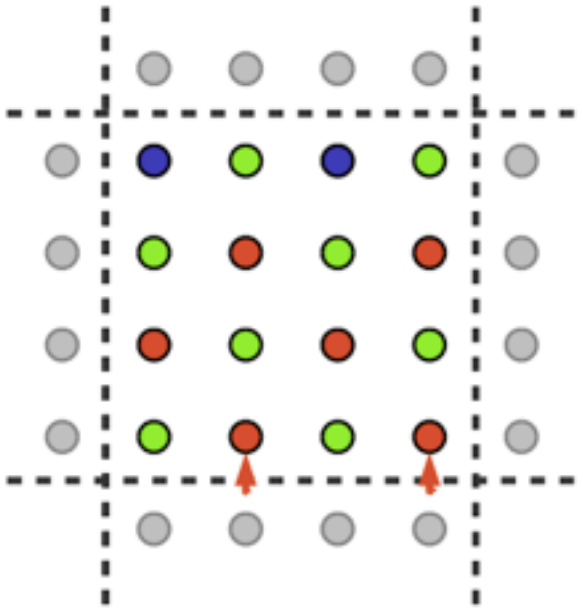
Step 3

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



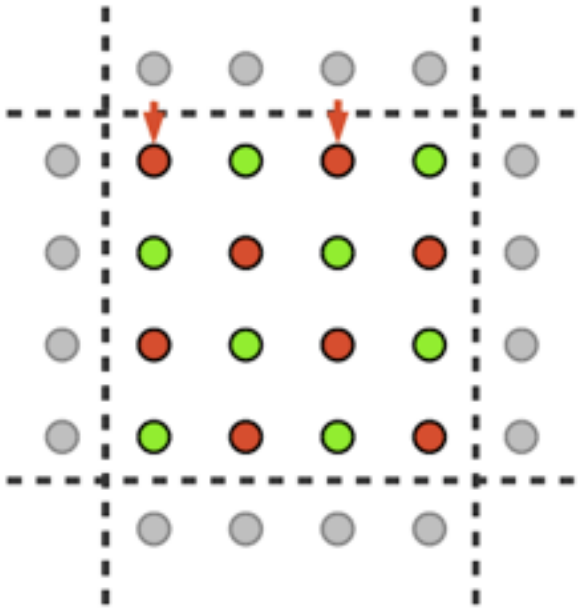
Step 3

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



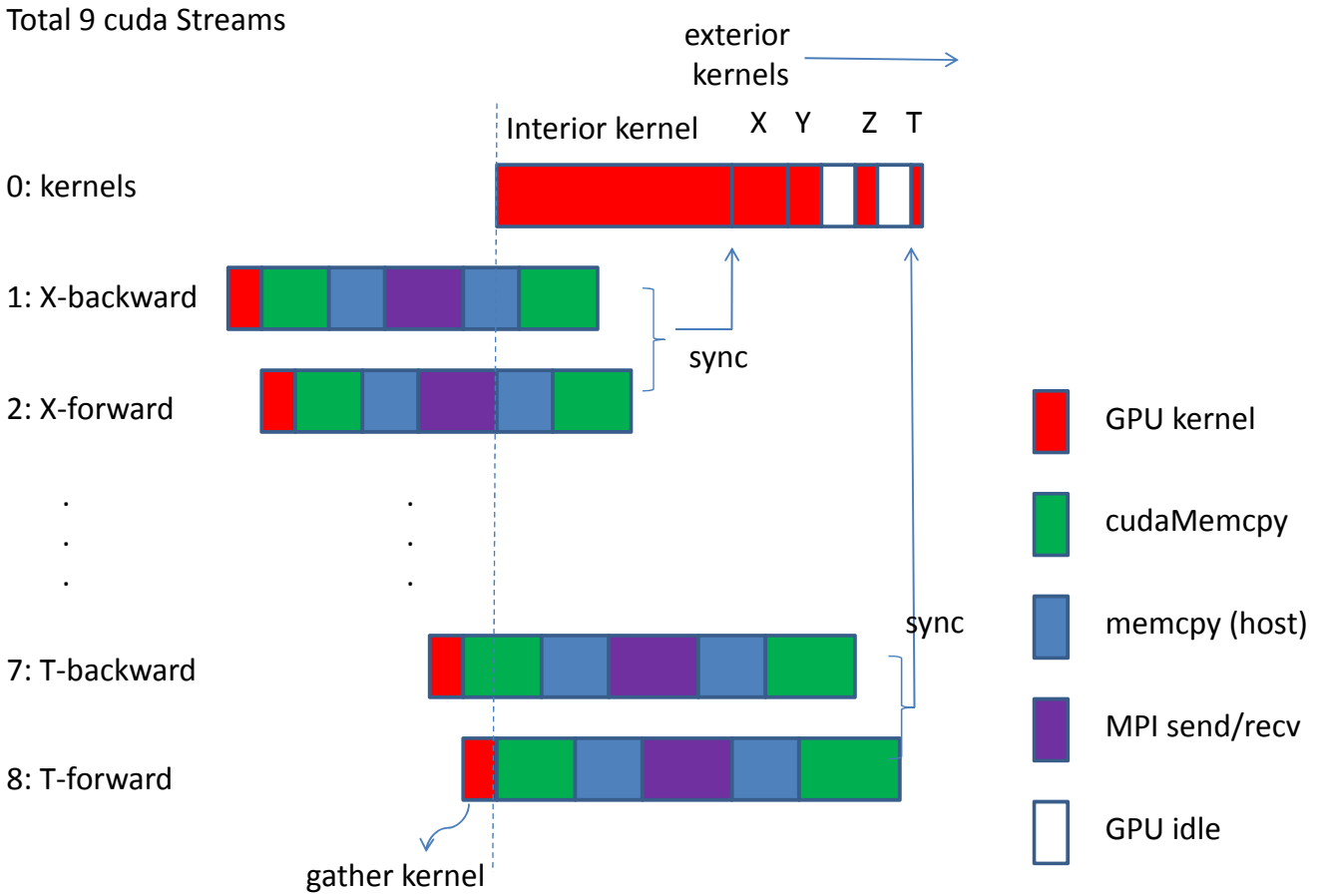
Step 3

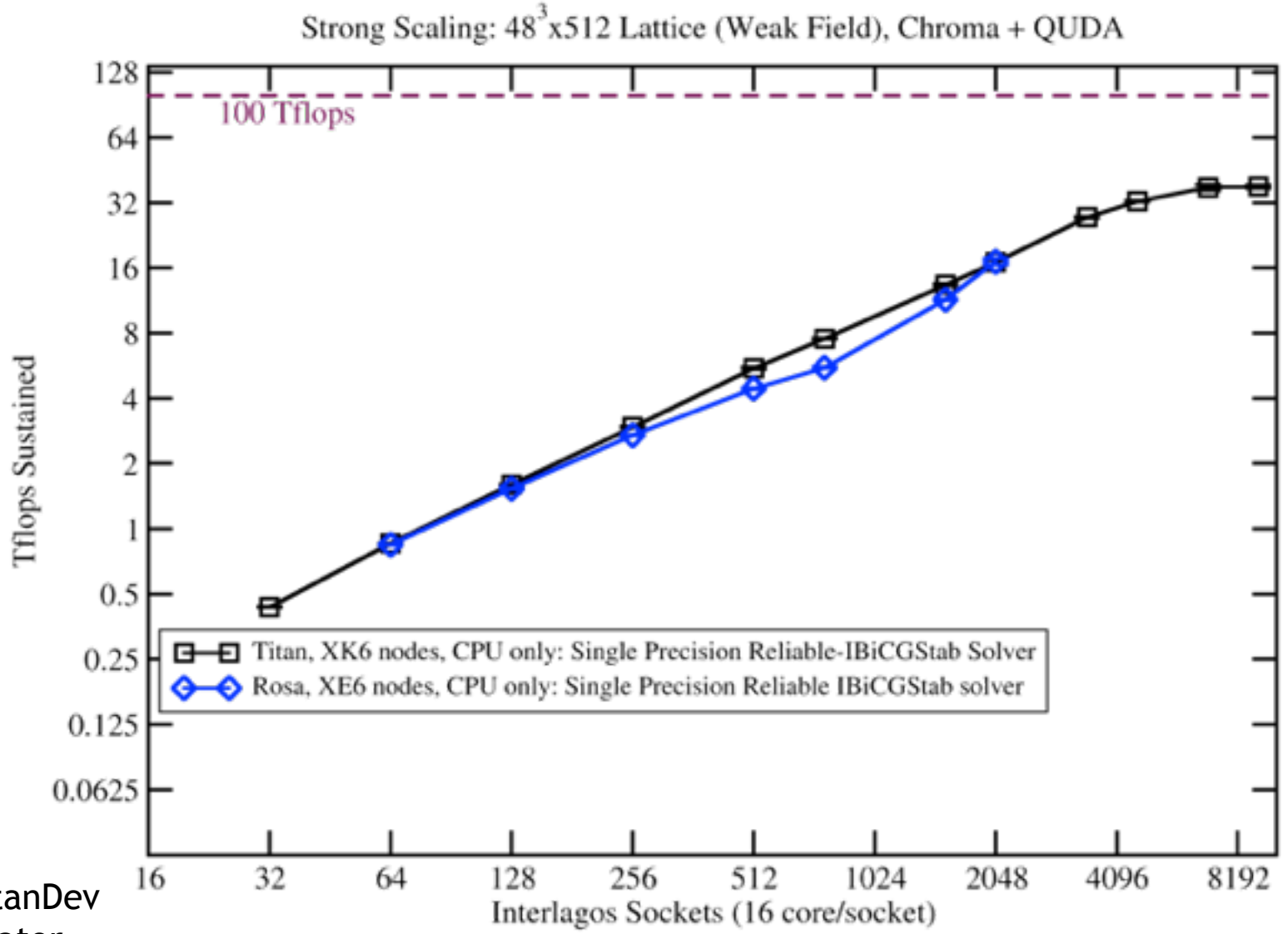
Boundary sites are updated by a series of kernels - one per direction.

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

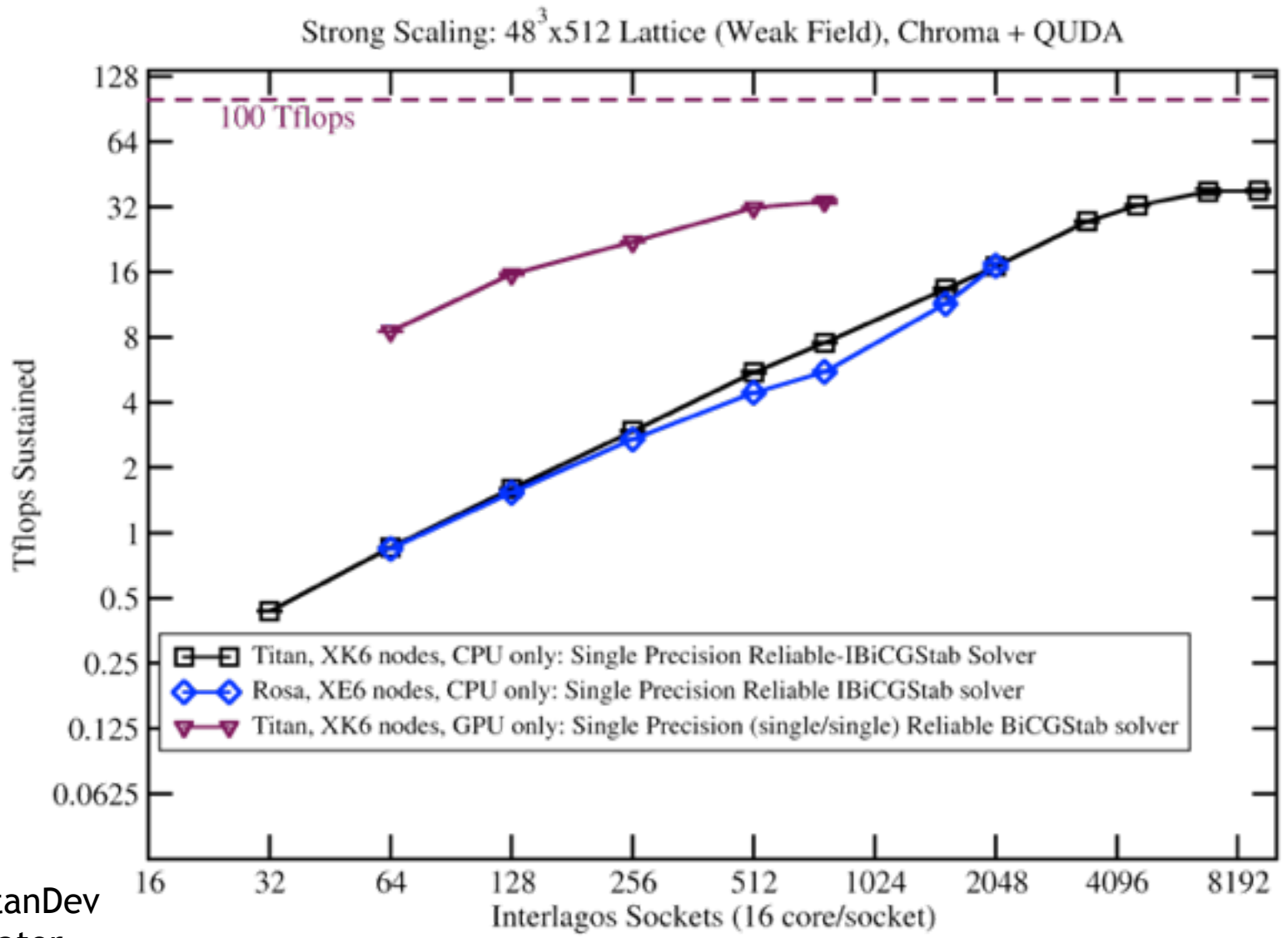
Note in higher dimensions corner sites have a race condition - serialization of kernels required

Multi-dimensional Communications Pipeline

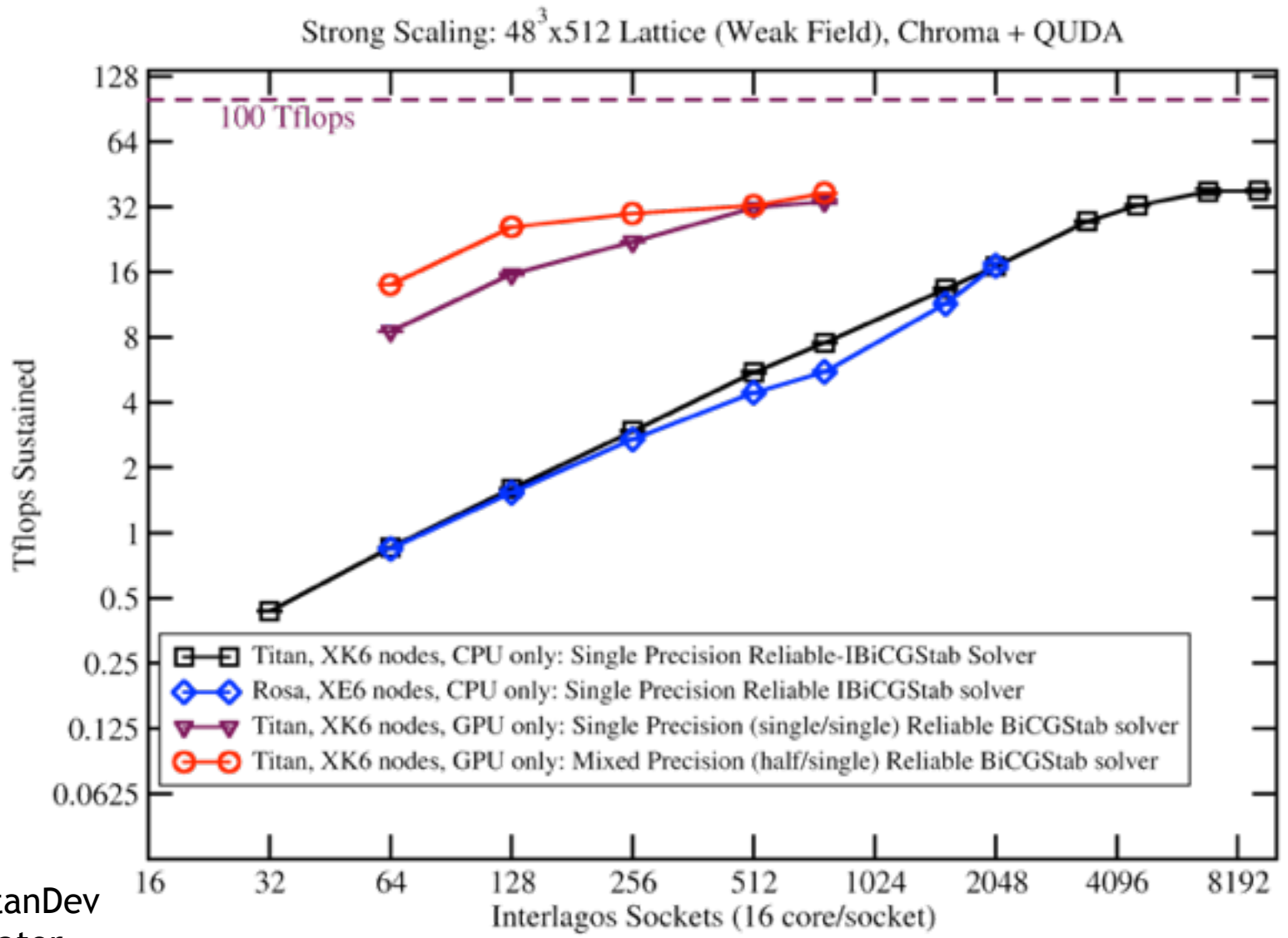




Results from TitanDev
- Clover propagator
- 48^3x512 aniso clover
- scaling up 768 GPUs



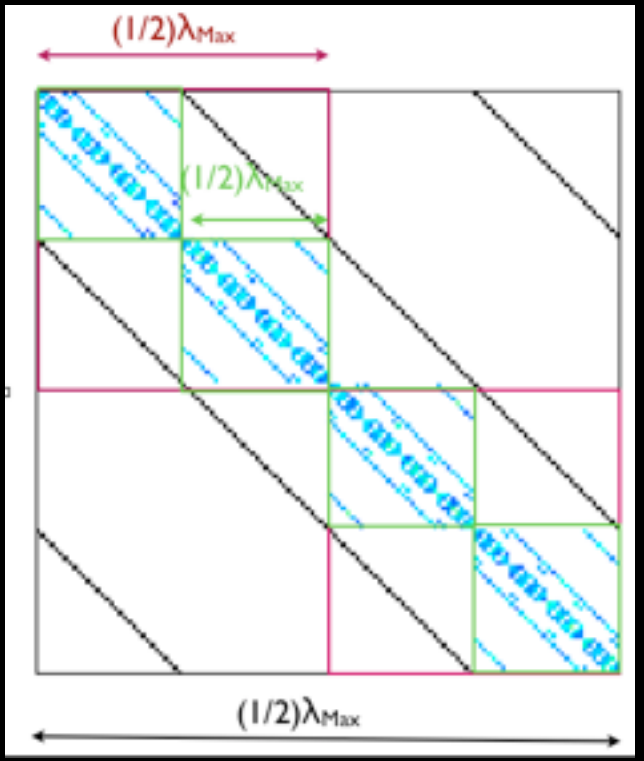
Results from TitanDev
- Clover propagator
- 48^3x512 aniso clover
- scaling up 768 GPUs

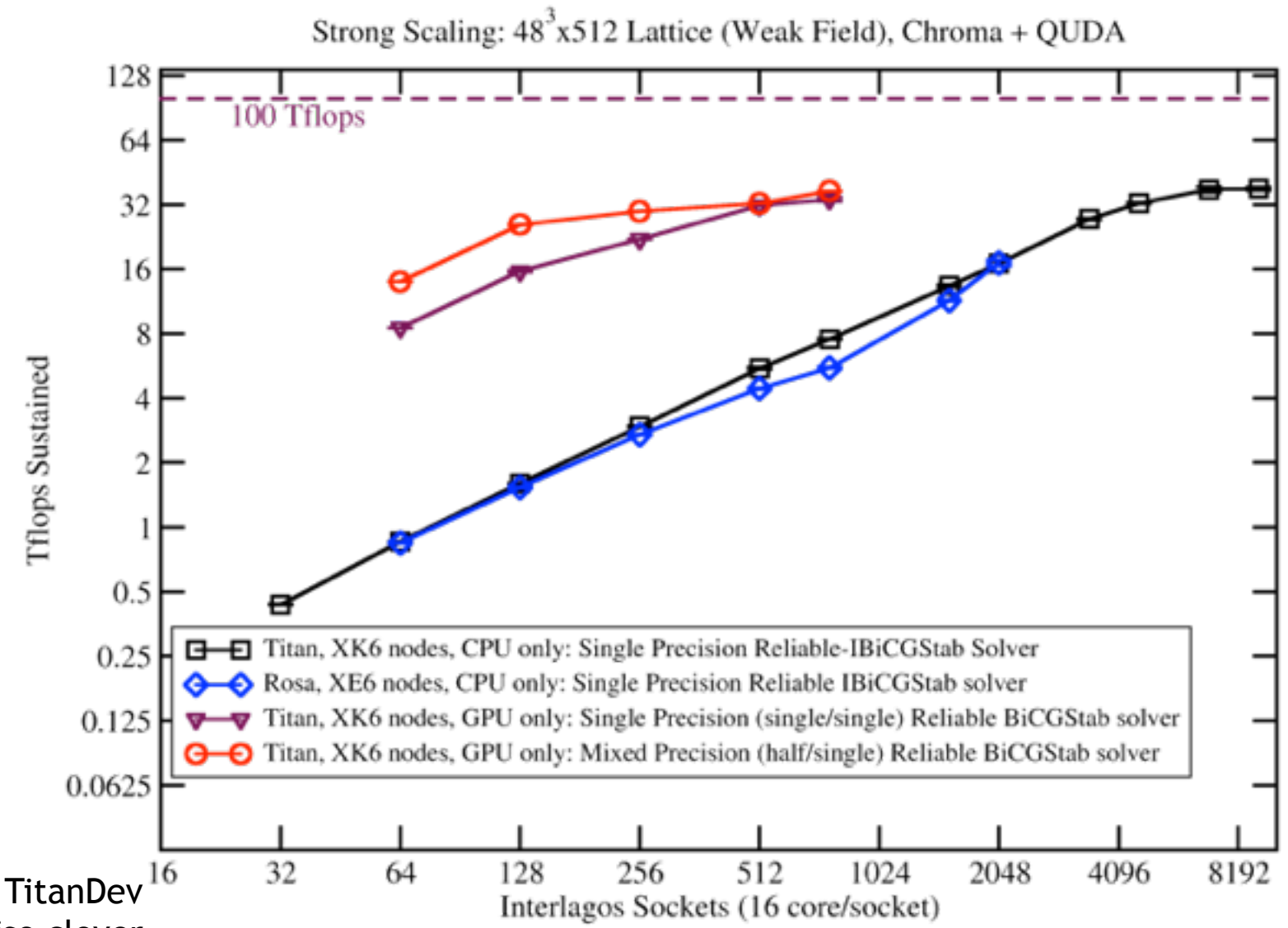


Results from TitanDev
- Clover propagator
- 48^3x512 aniso clover
- scaling up 768 GPUs

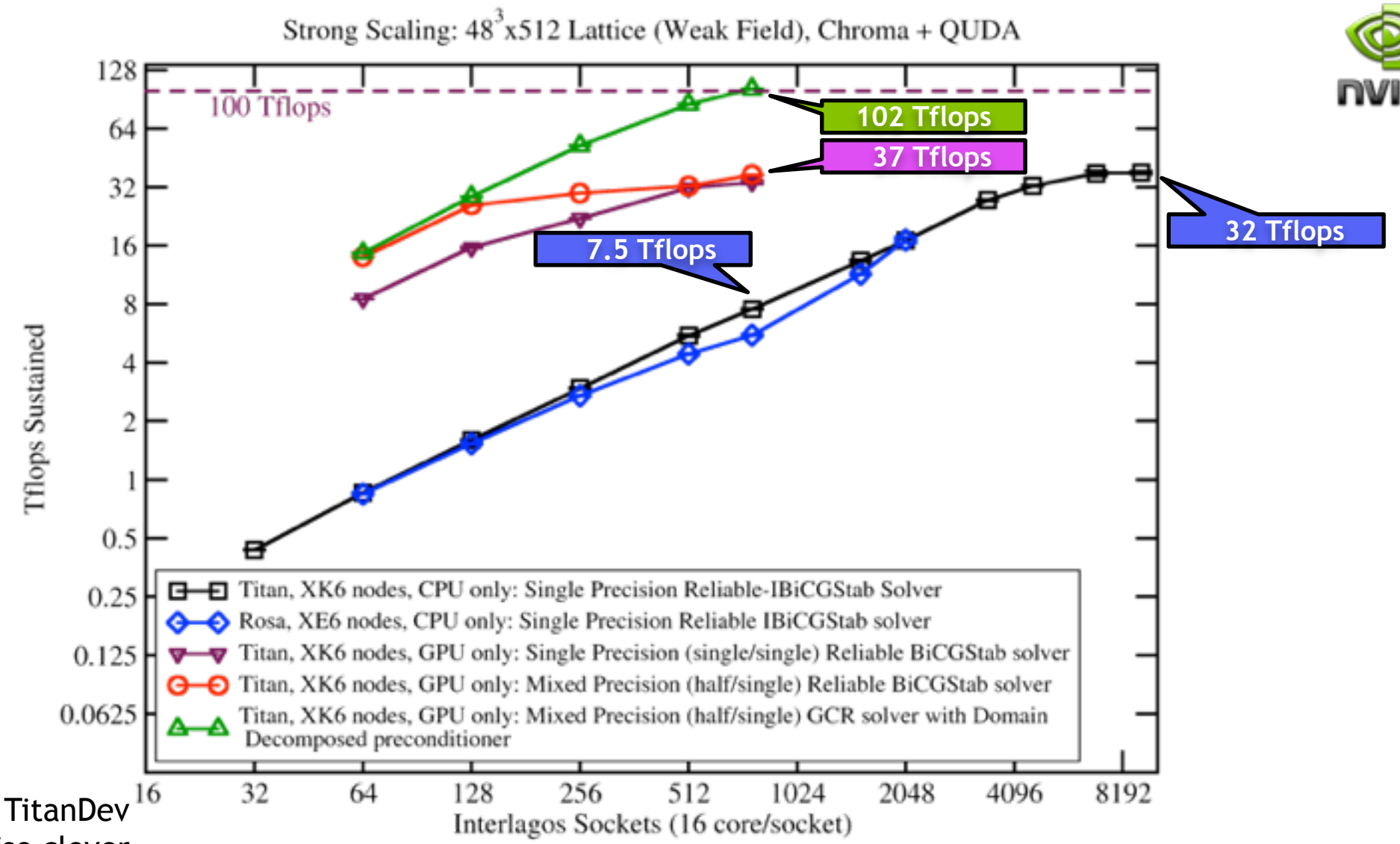
Domain Decomposition

- 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 preconditoner impose λ cutoff
- Smaller blocks lose low frequency modes
  - keep wavelengths of ~ $O(\Lambda_{QCD}^{-1})$, $\Lambda_{QCD}^{-1} \sim 1fm$
- Aniso clover: (a_s=0.125fm, a_t=0.035fm) ▫ 8^3x32 blocks are ideal
  - 48^3x512 lattice: 8^3x32 blocks ▫ 3456 GPUs





Results from TitanDev
- 48^3x512 aniso clover
- scaling up 768 GPUs



Results from TitanDev
- 48^3x512 aniso clover
- scaling up 768 GPUs

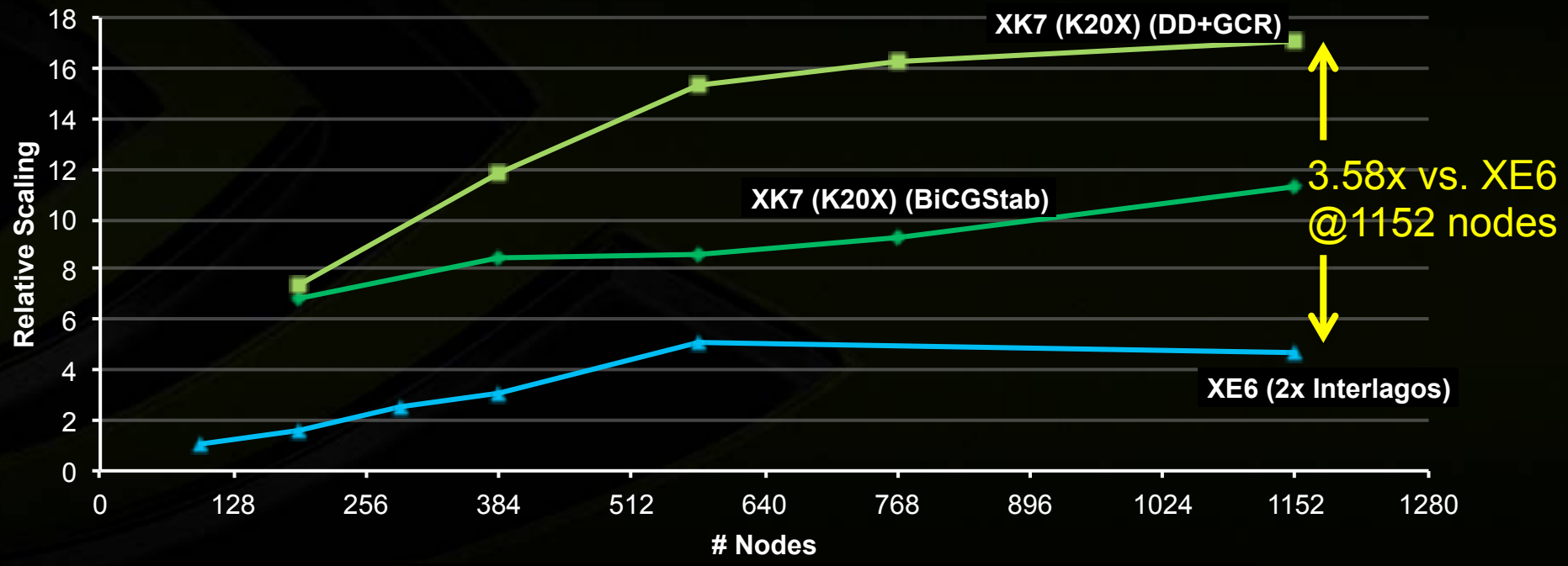
Chroma (Lattice QCD) – High Energy & Nuclear Physics

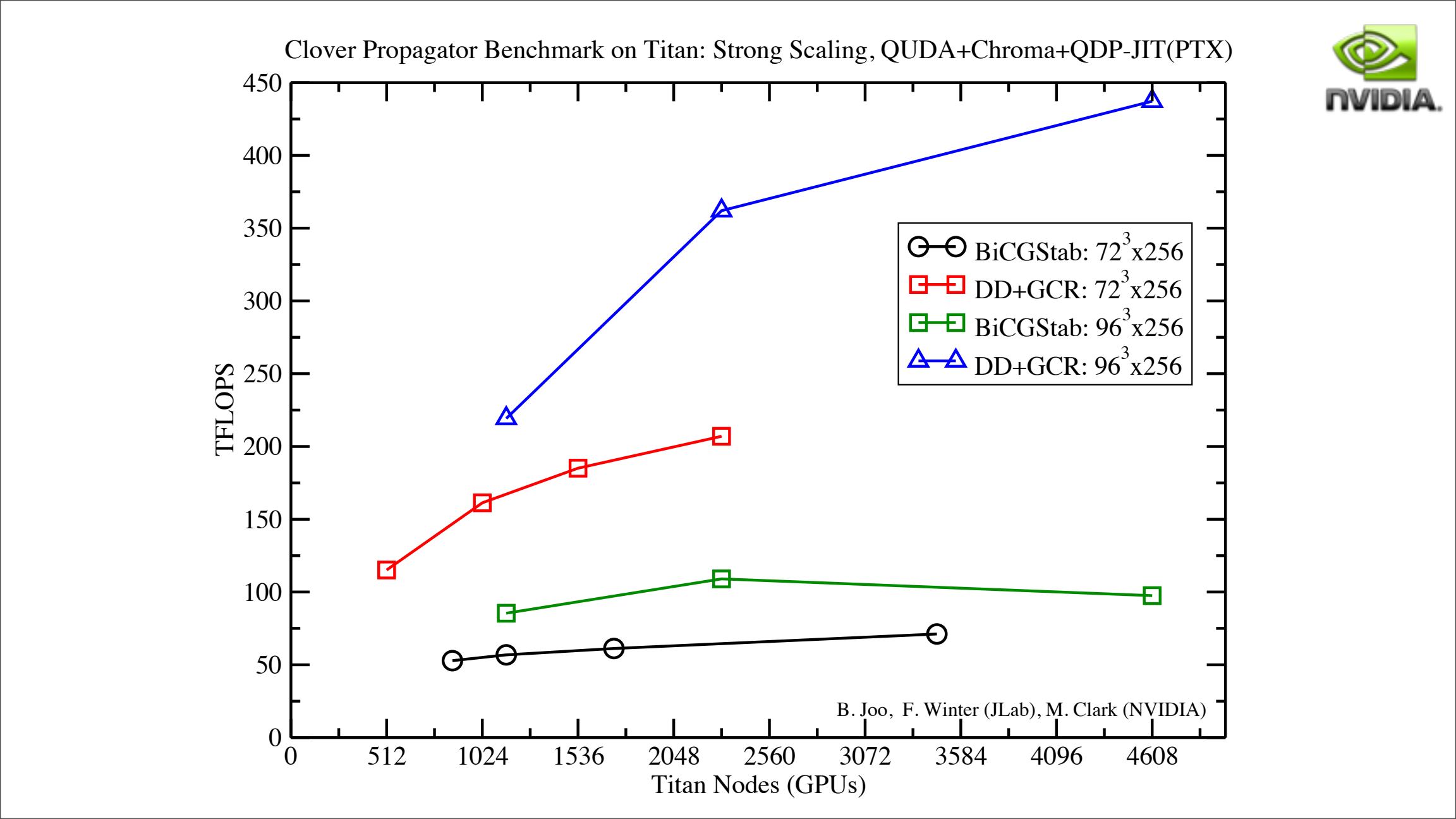
Chroma

48^3x512 lattice

Relative Scaling (Application Time)

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

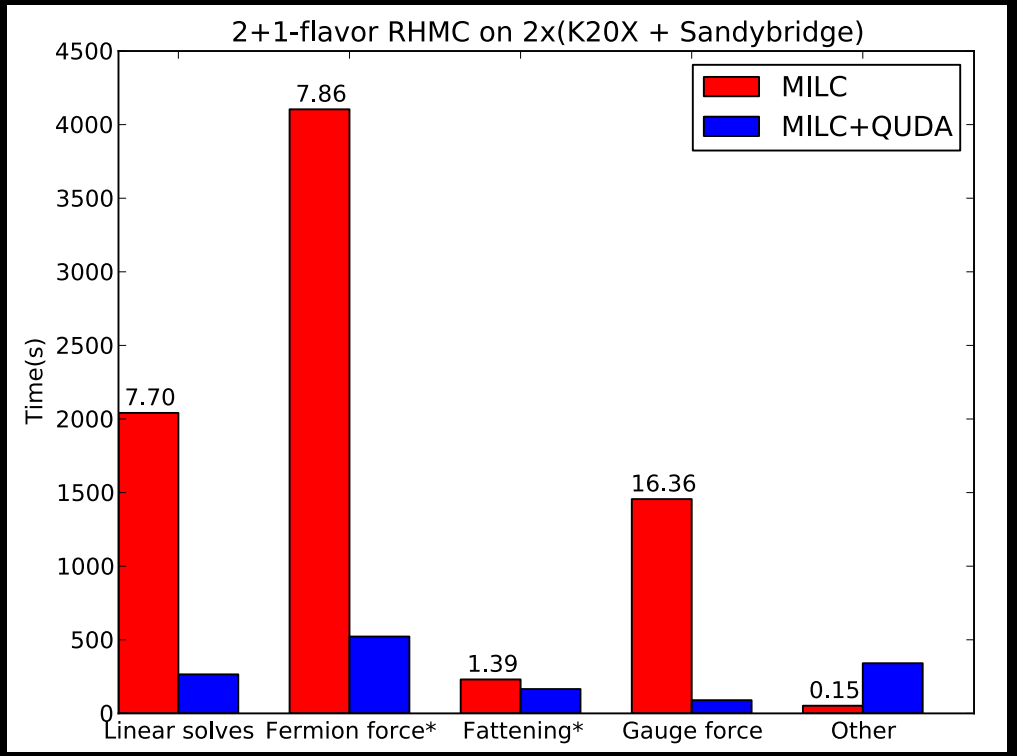



Clover Propagator Benchmark on Titan: Strong Scaling, QUDA+Chroma+QDP-JIT(PTX)
TFLOPS
Titan Nodes (GPUs)
0
50
100
150
200
250
300
350
400
450
0
512
1024
1536
2048
2560
3072
3584
4096
4608
BiCGStab: 72^3x256
DD+GCR: 72^3x256
BiCGStab: 96^3x256
DD+GCR: 96^3x256
B. Joo, F. Winter (JLab), M. Clark (NVIDIA)
NVIDIA

HISQ RHMC with QUDA

Routine	Single	Double	Mixed
Multi-shift solver	156.5	77.1	157.4
Fermion force	191.2	97.2	
Fat link generation	170.7	82.0	
Gauge force	194.8	98.3	

Absolute performance (36^4 lattice)

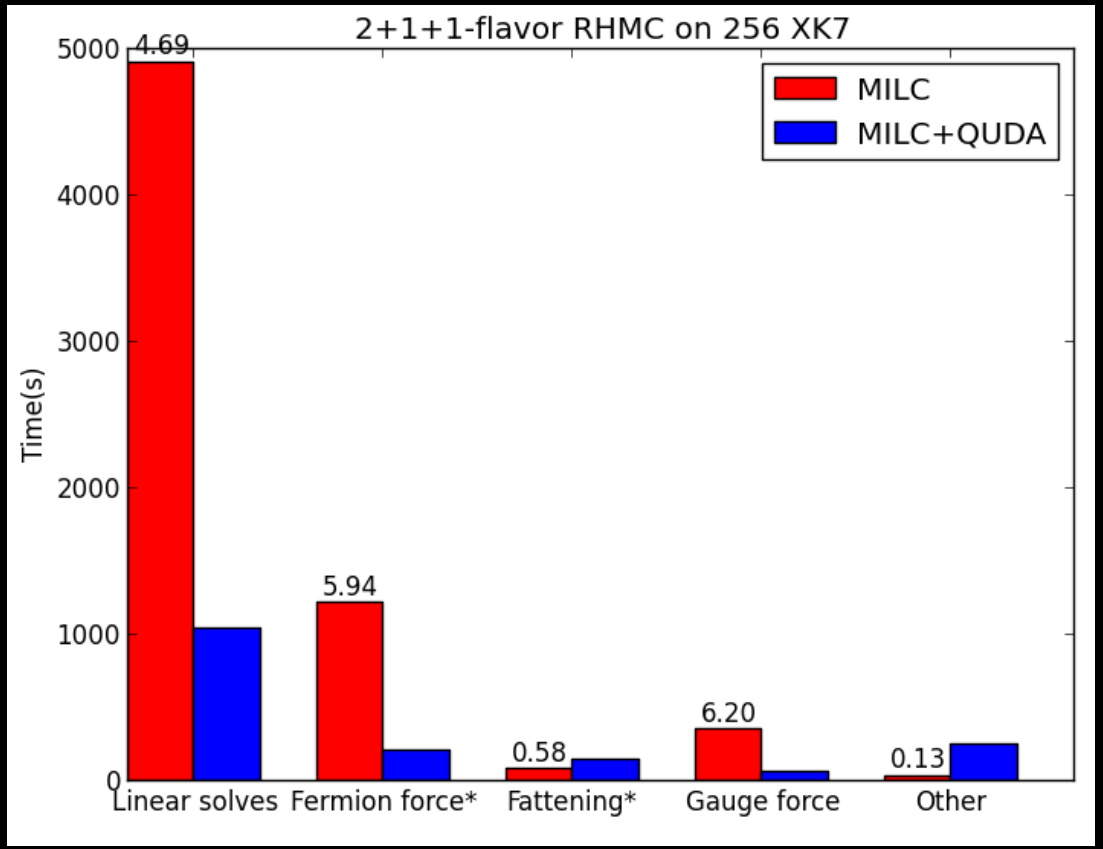


QUDA vs. MILC ($24^3 64$)

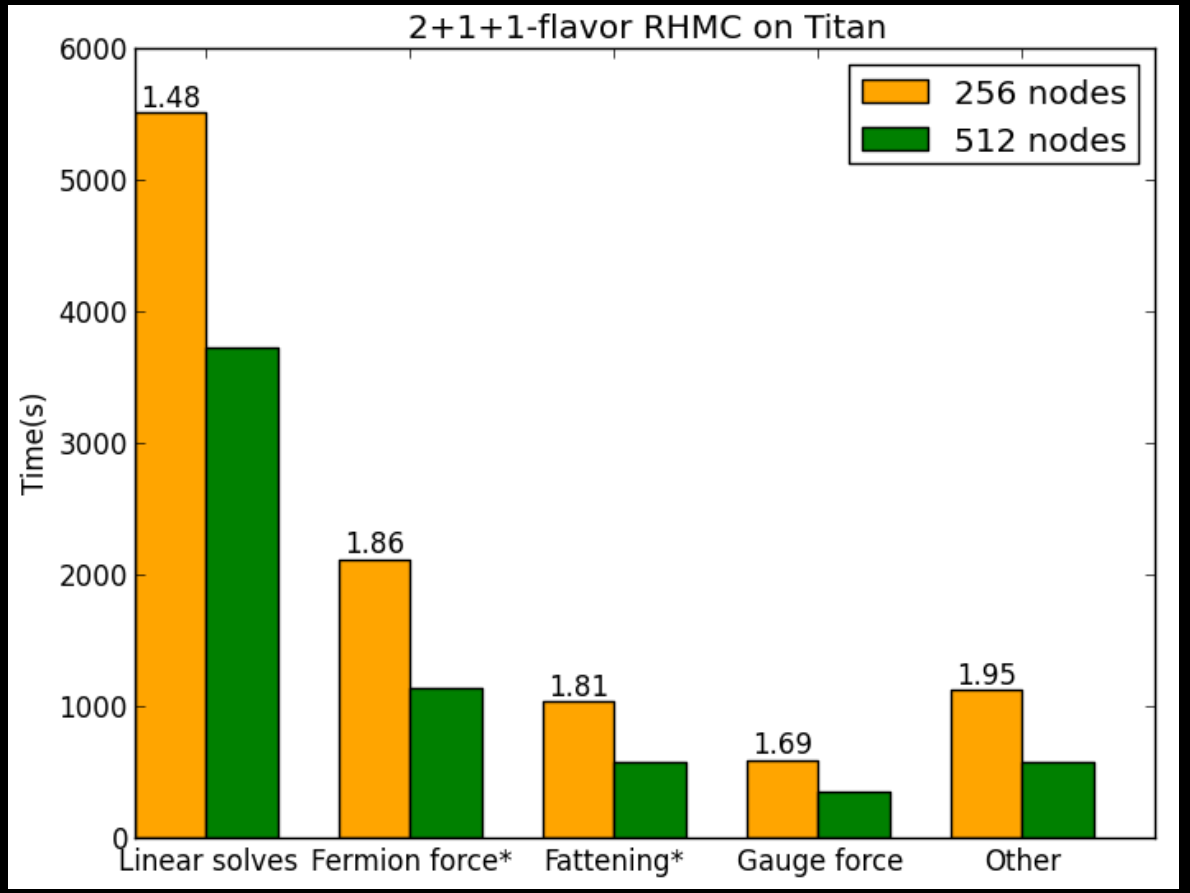
MILC on QUDA

- Gauge generation on 256 BW nodes
  - Volume = 96^3x192
  - 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



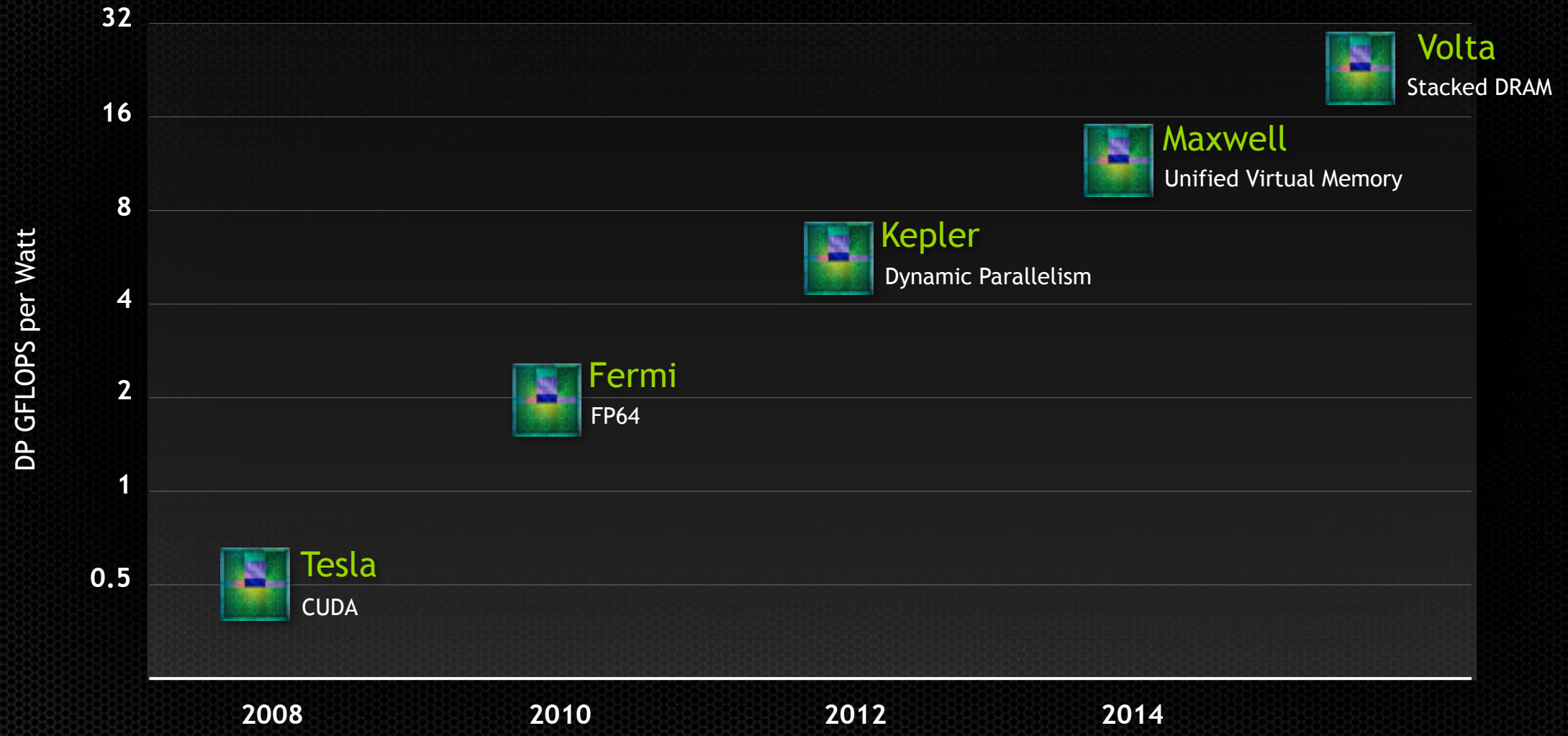
MILC on QUDA



Preliminary strong scaling on Titan ($V = 96^3 \times 192$)

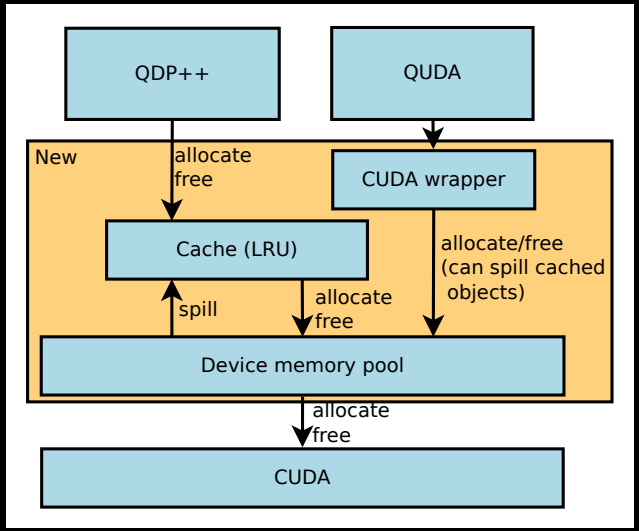

Future Directions

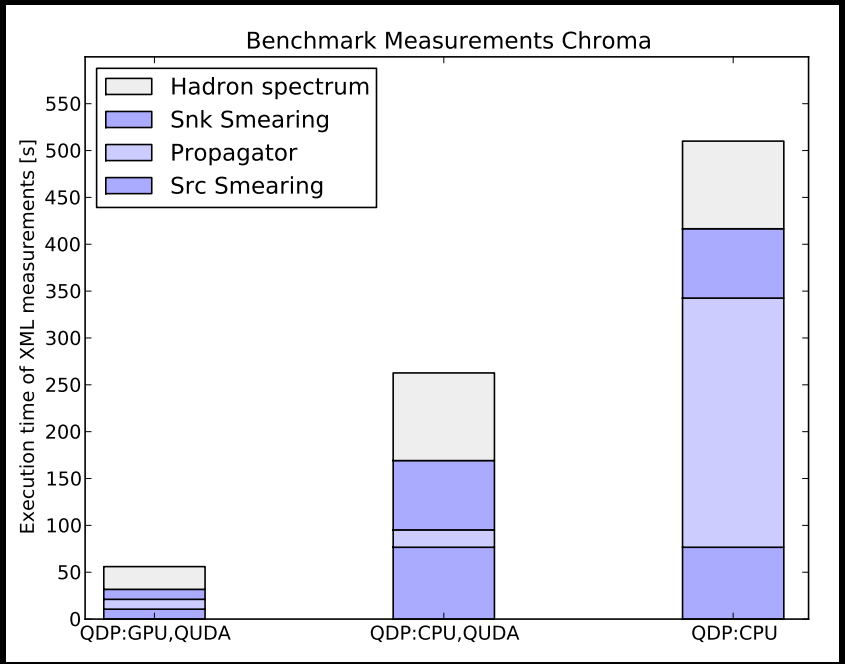
GPU Roadmap


DP GFLOPS per Watt
32
16
8
4
2
1
0.5
2008
2010
2012
2014
Tesla
CUDA
Fermi
FP64
Kepler
Dynamic Parallelism
Maxwell
Unified Virtual Memory
Volta
Stacked DRAM

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)



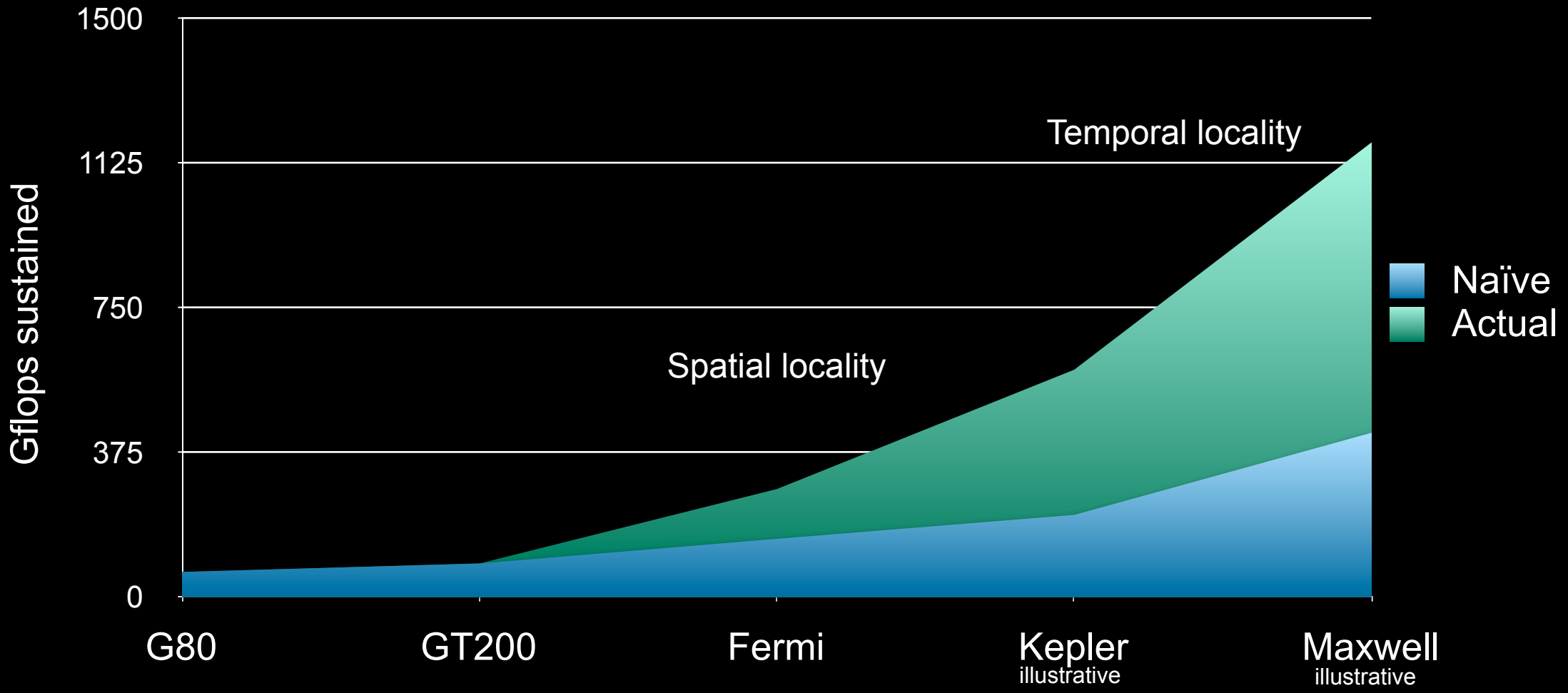


QUDA - Chroma Integration

- 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

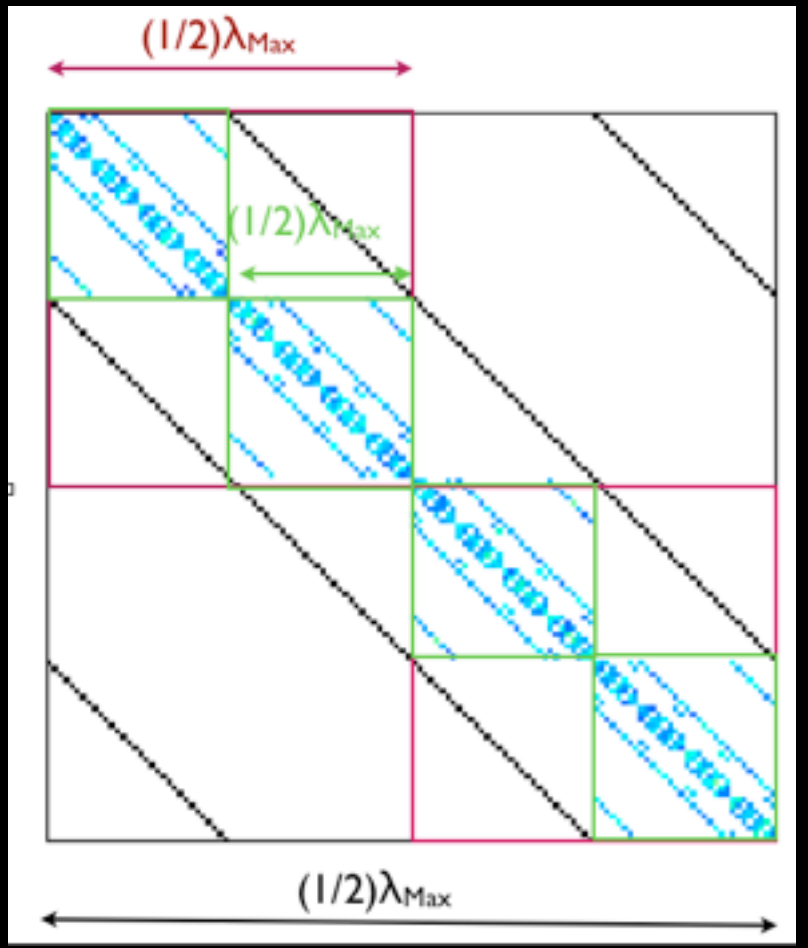
Exploiting Locality

Wilson SP Dslash Performance with GPU generation



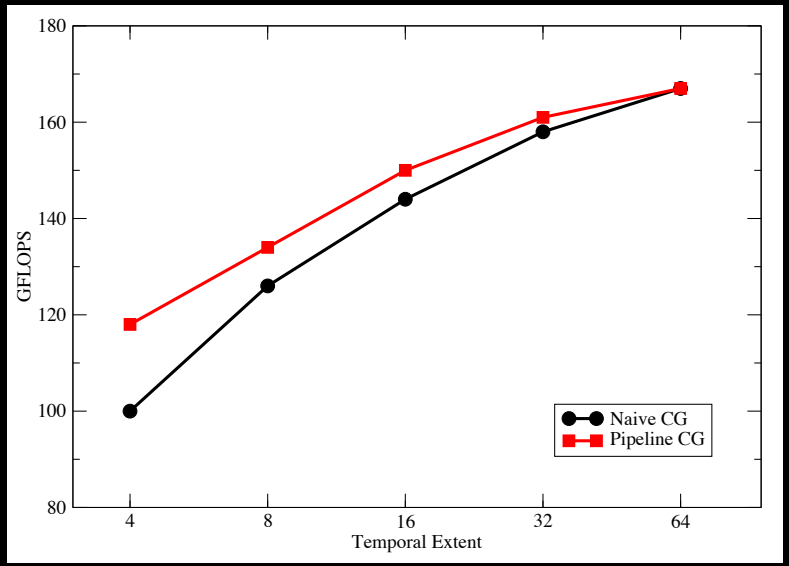
Future Directions - Communication

- 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



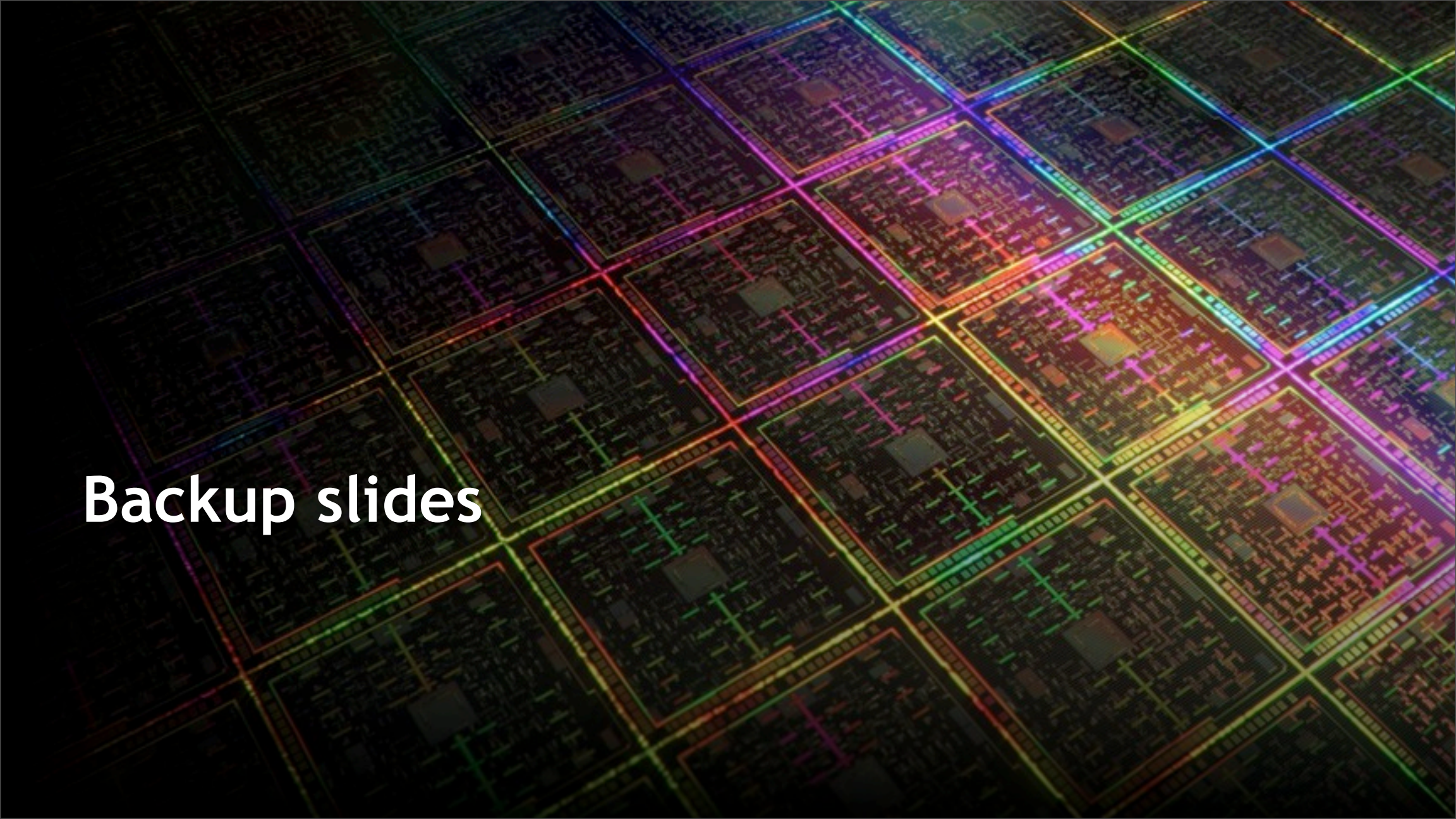
Future Directions - Precision

- 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

mclark at nvidia dot com

- 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

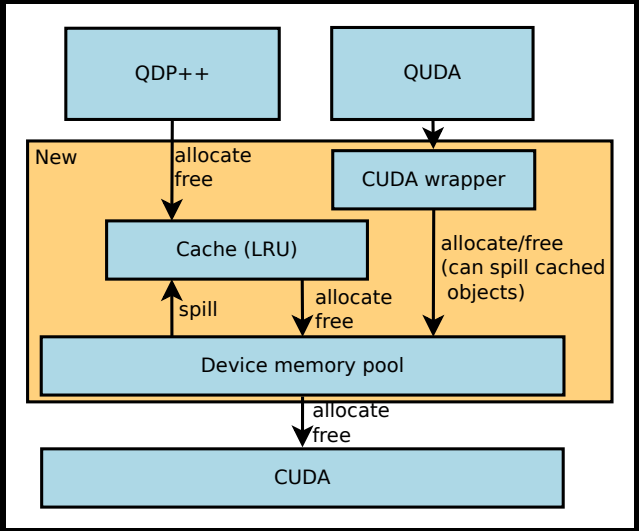
- Allow QUDA interface to accept GPU pointers
  - First natural extension
  - Remove unnecessary PCIe communications between QUDA function calls
- Allow user-defined functors for handling field ordering
  - User only has to specify their field order
  - Made possible with device libraries (CUDA 5.0)
- Limitations
  - Limited control of memory management
  - Requires deeper application integration

QUDA Low-Level Interface (in development)

- Possible strawman under consideration

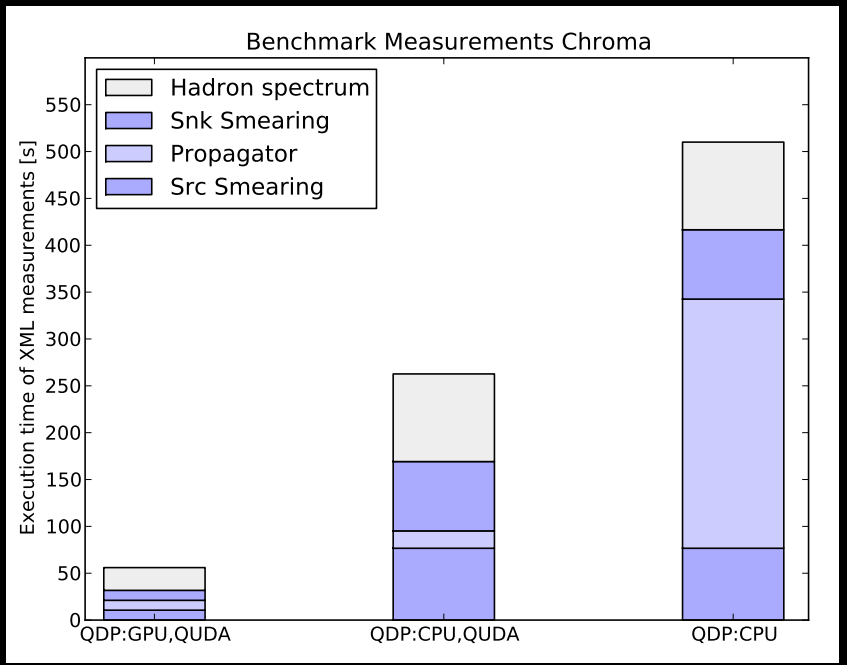
```
lat = QUDA_new_lattice(dims, ndim, lat_param);
u = QUDA_new_link_field(lat, gauge_param);
source = QUDA_new_site_field(lat, spinor_param);
solution = QUDA_new_site_field(lat, spinor_param);
QUDA_load_link_field(u, host_u, gauge_order);
QUDA_load_site_field(source, host_source, spinor_order);
QUDA_solve(solution, 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



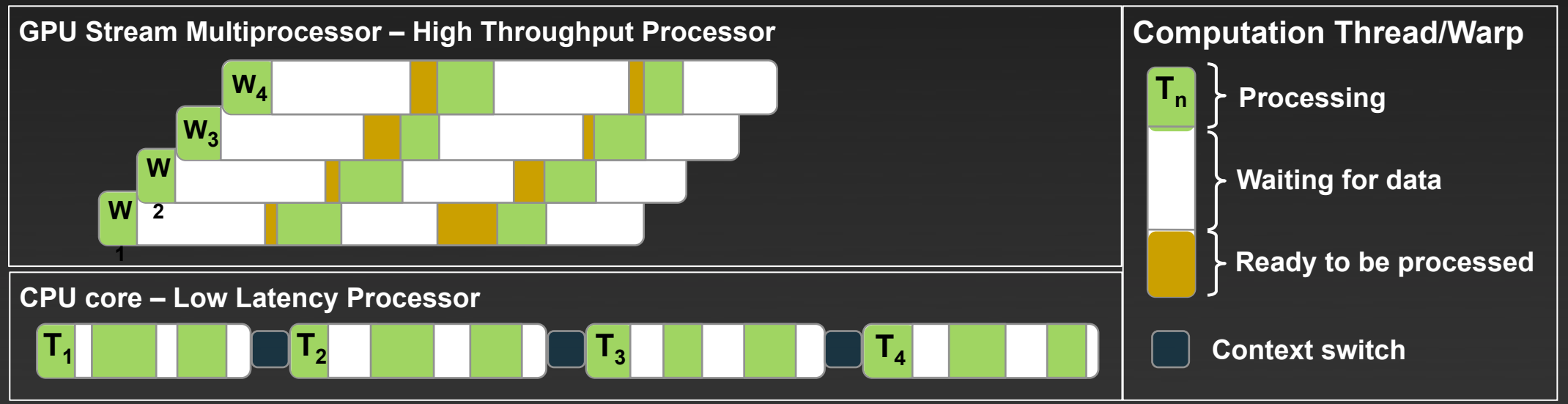
QUDA - Chroma Integration

- 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



Low Latency or High Throughput?

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



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

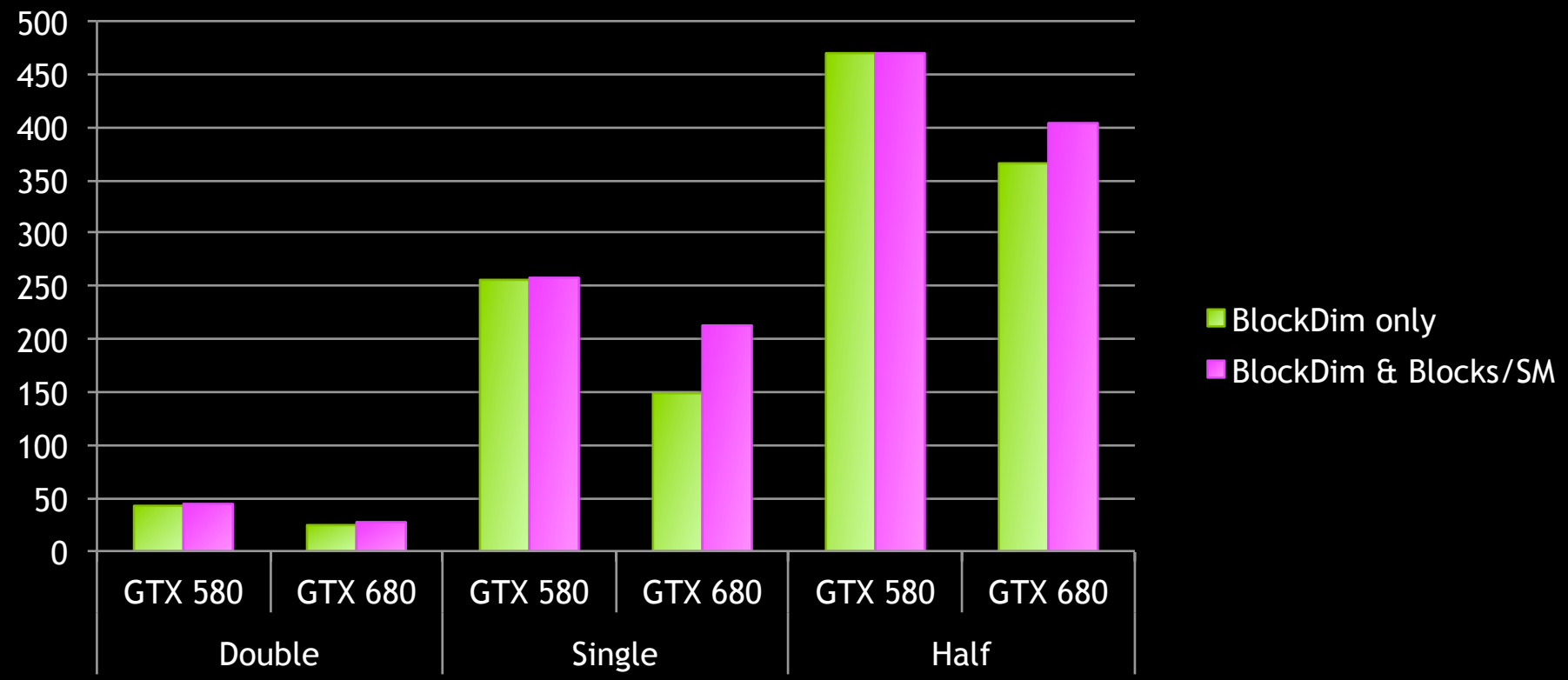
type	32-bit	64-bit	128-bit
int	int	int2	int4
float	float	float2	float4
double		double	double2
char	char4		
short	short2	short4	

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:

```
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:

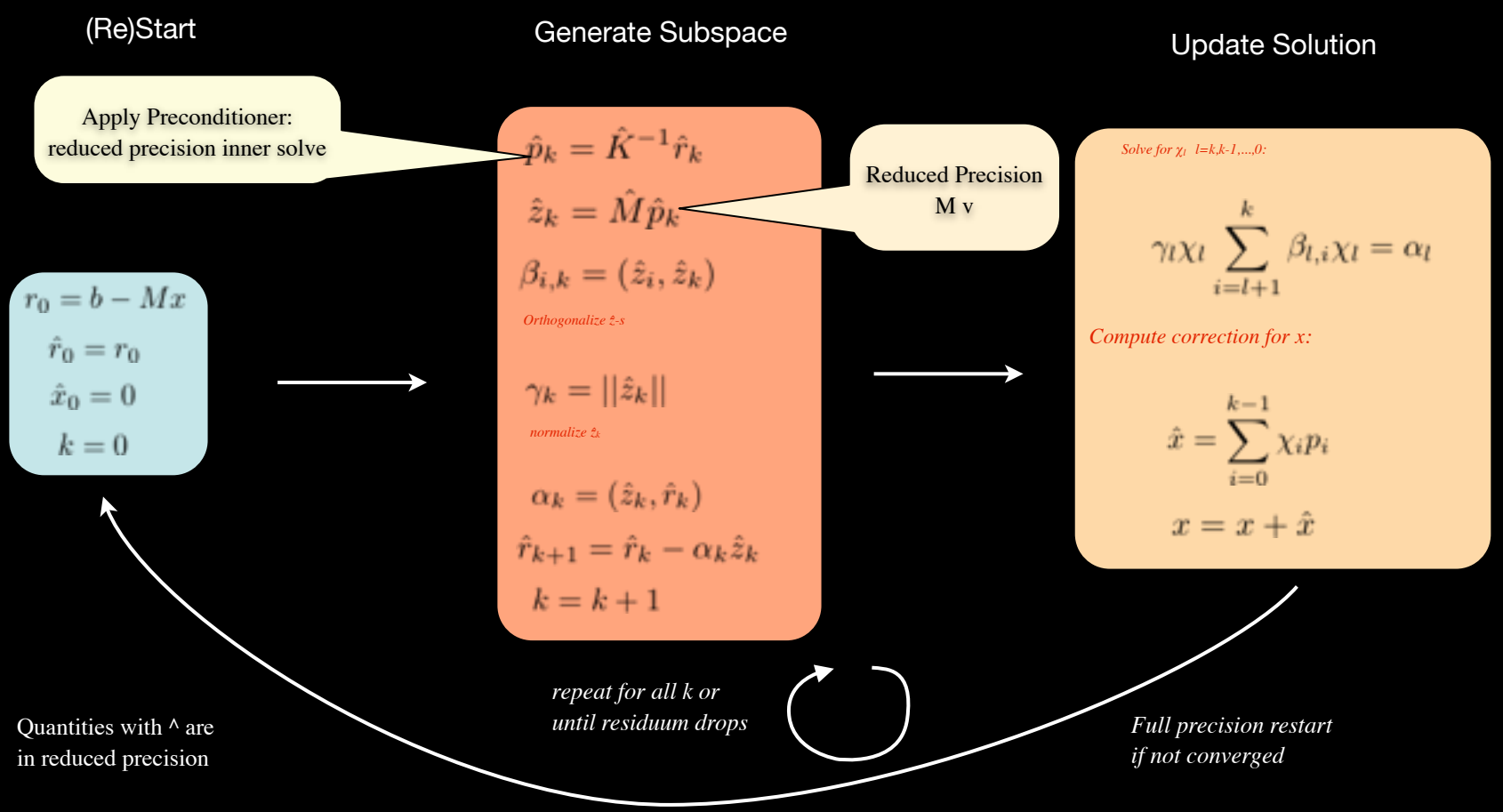
```
MyKernelWrapper *k = new MyKernelWrapper(a, b, c);
k->apply();  // <-- automatically tunes if necessary
```

- 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

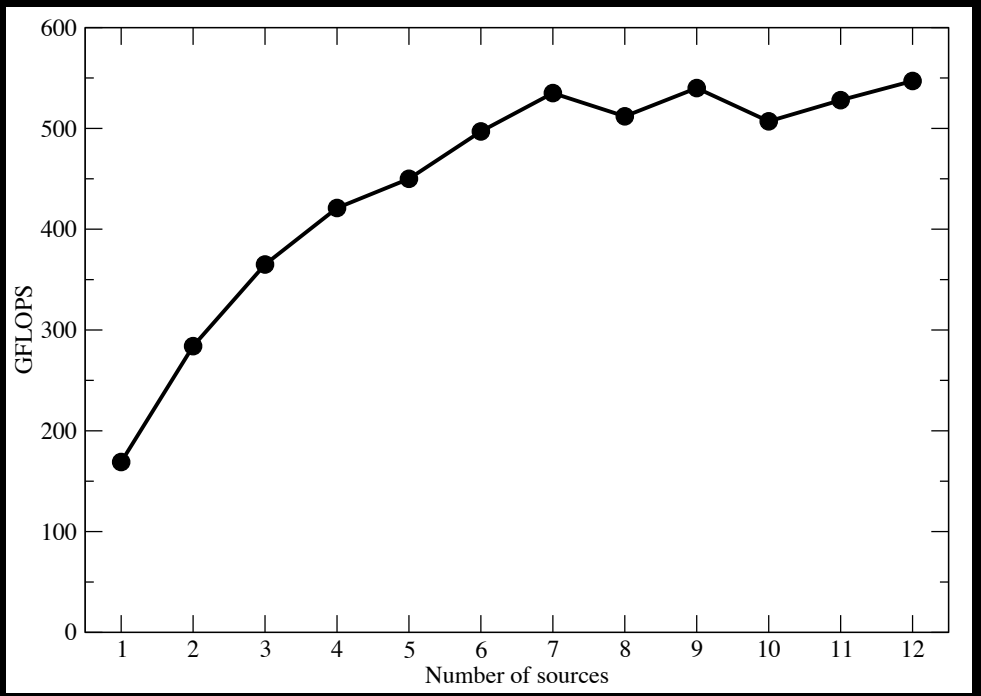
- Invoke the kernel (tuning if necessary):
  - apply()
- Save and restore state before/after tuning:
  - preTune(), postTune()
- Advance to next set of trial parameters in the tuning:
  - advanceGridDim(), advanceBlockDim(), advanceSharedBytes()
  - advanceTuneParam() // simply calls the above by default
- Performance reporting
  - flops(), bytes(), perfString()
- etc.

Domain Decomposition



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