



The QUDA library for lattice QCD on GPUs

M A Clark, NVIDIA
Developer Technology Group



Outline



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

Collaborators and QUDA 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)
- Claudio Rebbi (Boston University)
- Guochun Shi (NCSA -> Google)
- Alexei Strelchenko (Cyprus Institute -> FNAL)
- Frank Winter (The University of Edinburgh)



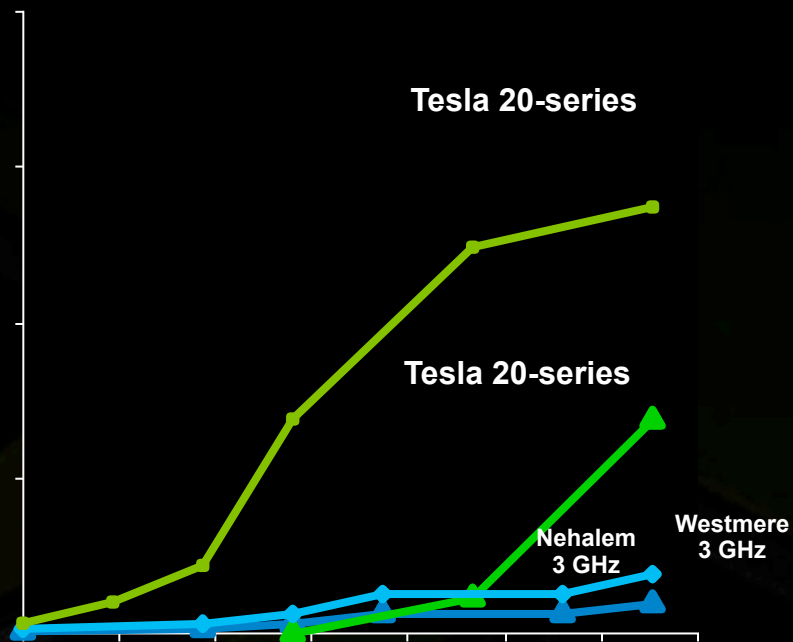
**MORE THAN JUST INNOVATIVE.
GAME-CHANGING.**

EXPERIENCE THE GEFORCE® GTX 690.

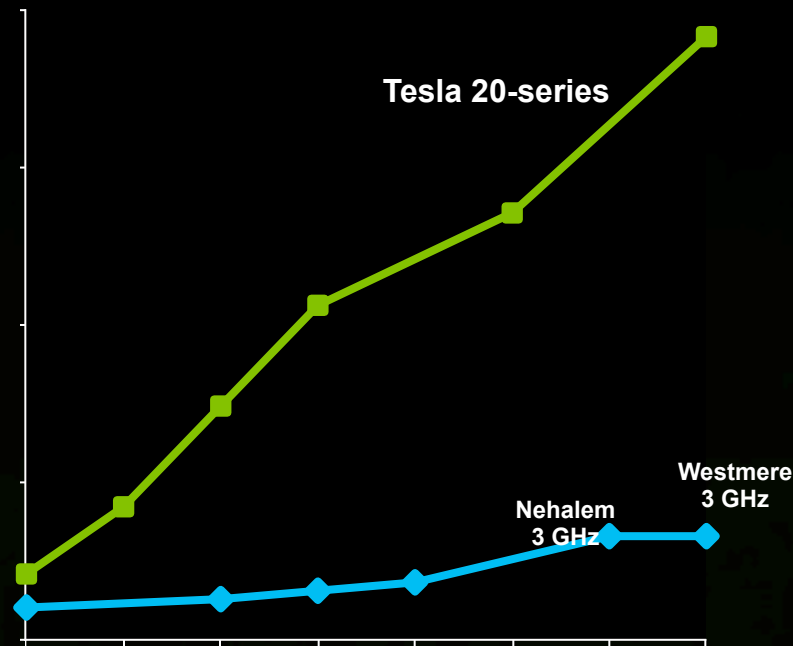
Why GPU Computing?



GFlops/sec



GBytes/sec

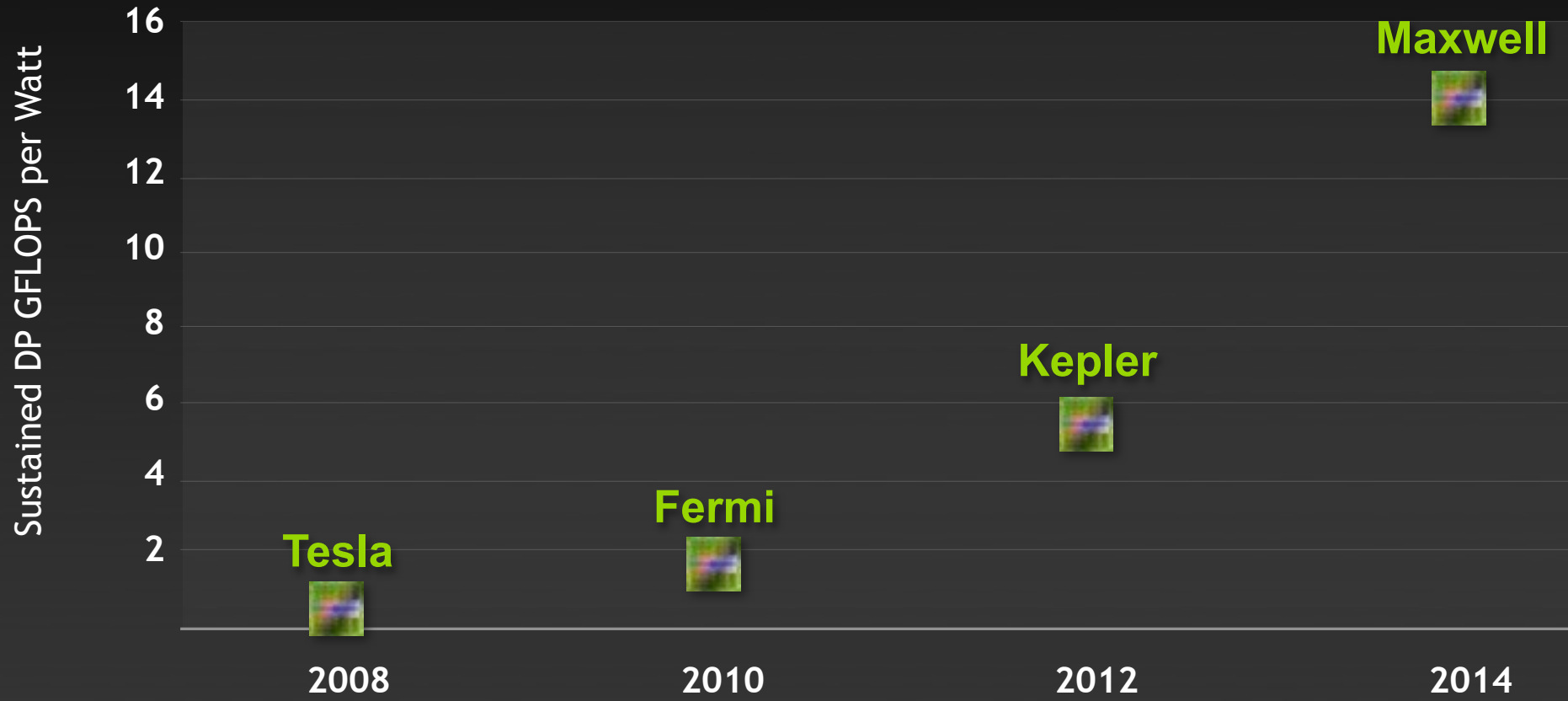


- Single Precision: NVIDIA GPU
- ▲ Double Precision: NVIDIA GPU
- ◆ Single Precision: x86 CPU
- ▲ Double Precision: x86 CPU

- NVIDIA GPU ECC off
- ◆ X86 CPU

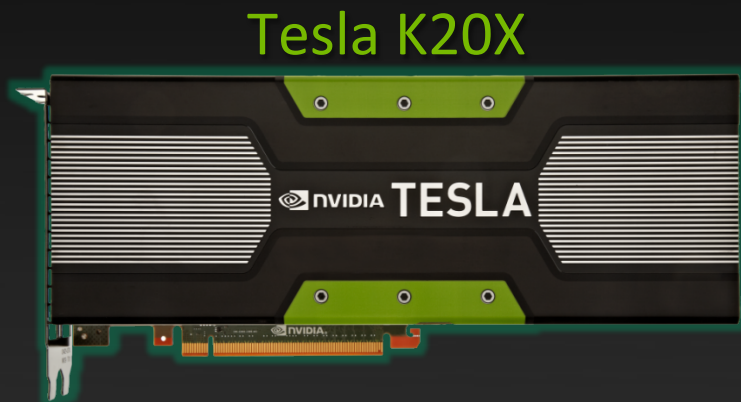


NVIDIA GPU Roadmap: Increasing Performance/Watt



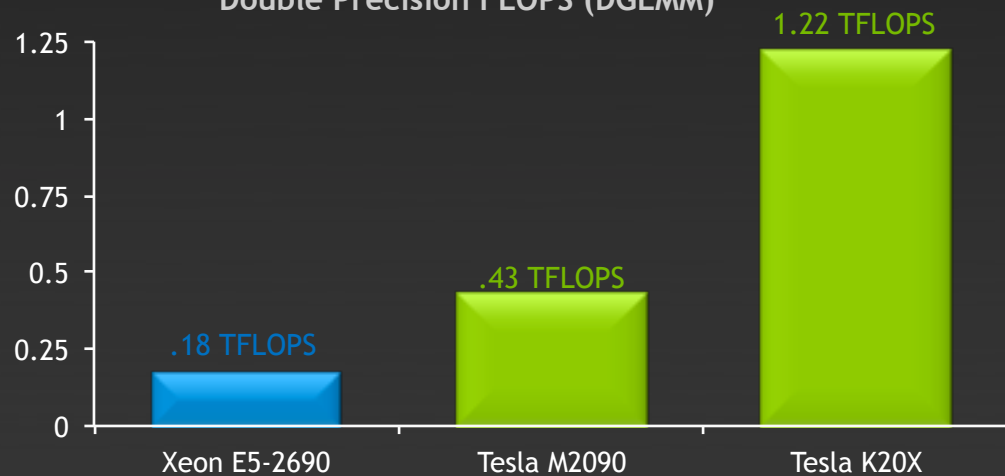
Tesla K20 Family : World's Fastest Accelerator

>1TFlop Perf in under 225W



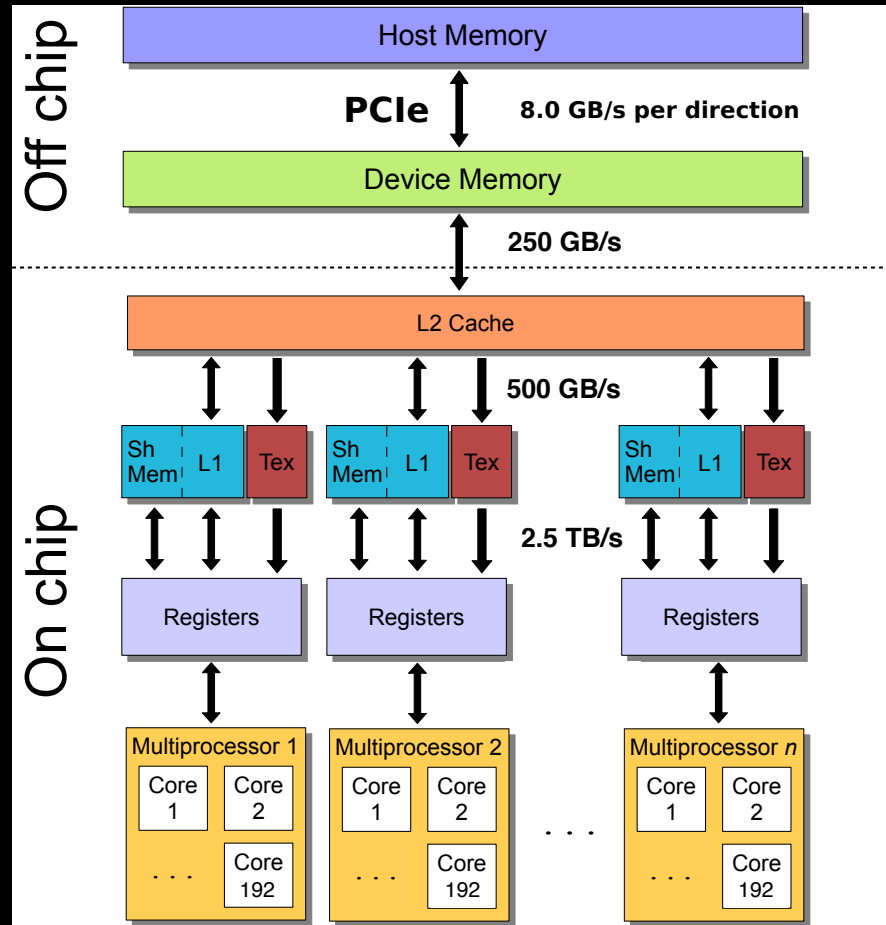
Tesla K20X

Double Precision FLOPS (DGEMM)



	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



Id software ©

Stunning Graphics Realism

Lush, Rich Worlds



Crysis © 2006 Crytek / Electronic Arts



Incredible Physics Effects



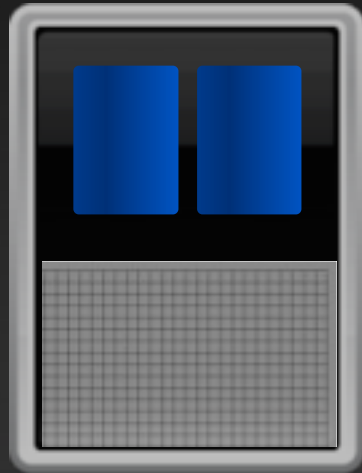
Core of the Definitive Gaming Platform

Hellgate: London © 2005-2006 Flagship Studios, Inc. Licensed by NAMCO BANDAI Games America, Inc.

Full Spectrum Warrior: Ten Hammers © 2006 Pandemic Studios, LLC. All rights reserved. © 2006 THQ Inc. All rights reserved.

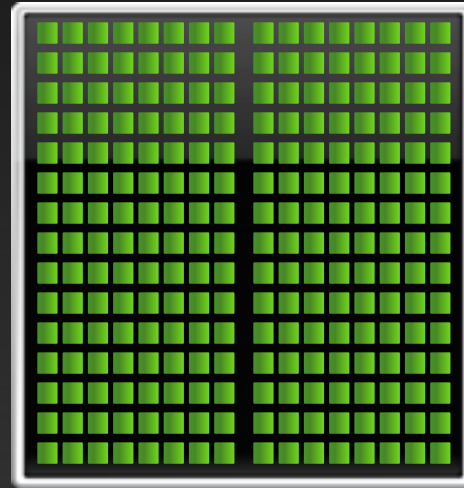
GPGPU Revolutionizes Computing

Latency Processor + Throughput processor



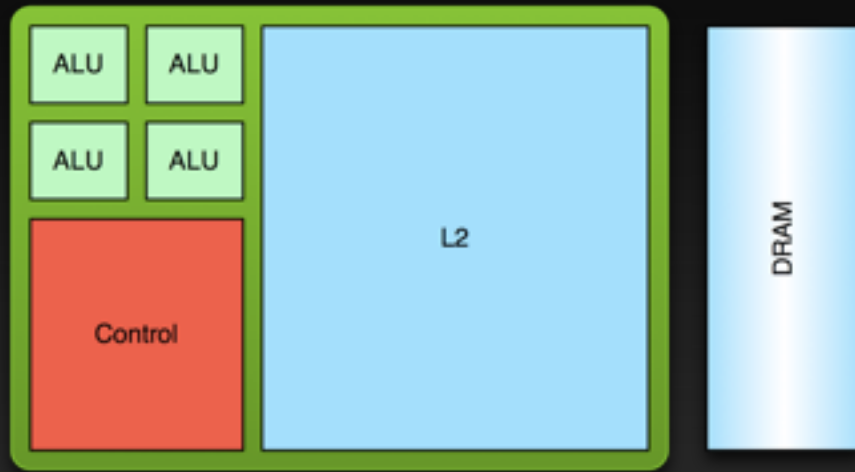
CPU

+



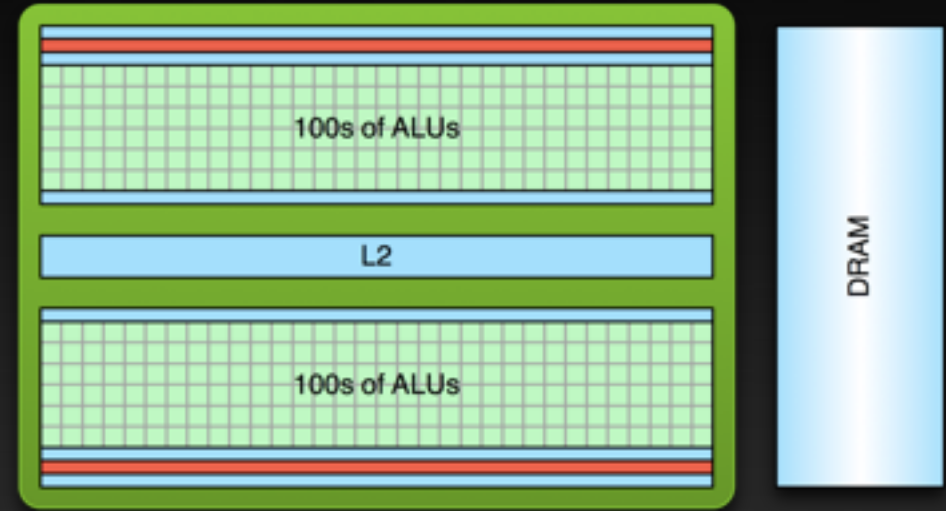
GPU

Low Latency or High Throughput?



CPU

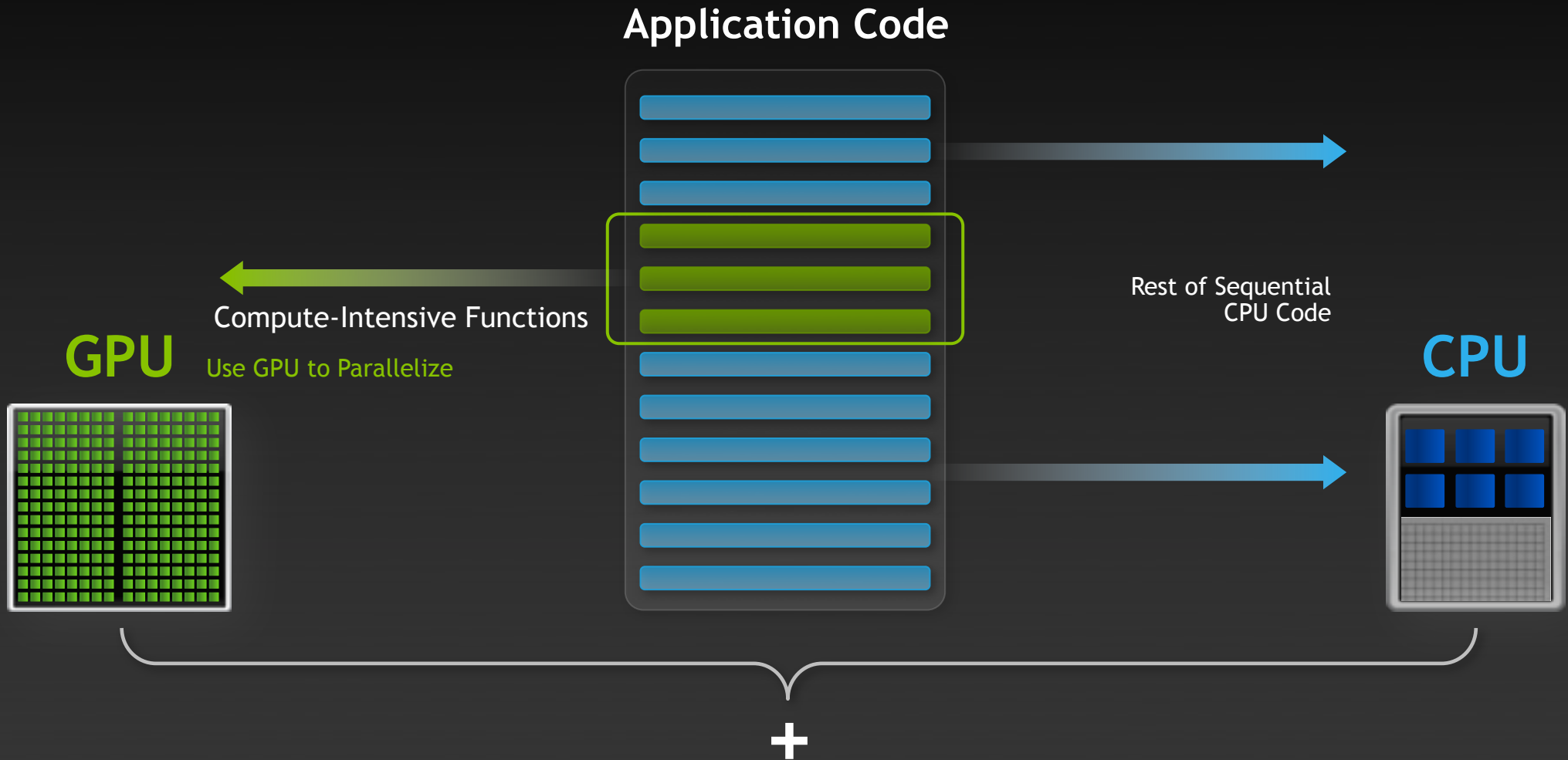
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

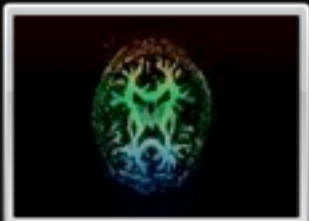


GPU

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

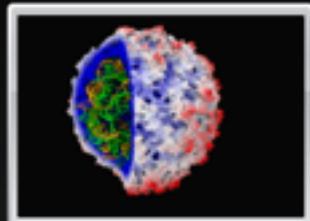
Small Changes, Big Speed-up





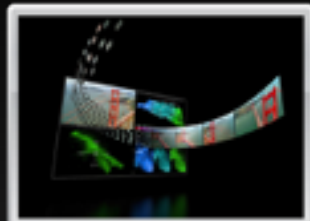
146X

Medical Imaging
U of Utah



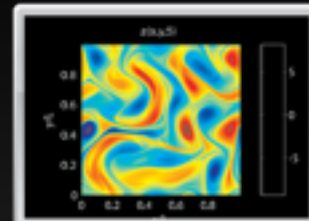
36X

Molecular Dynamics
U of Illinois, Urbana



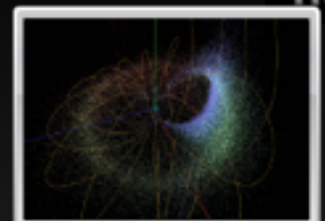
18X

Video Transcoding
Elemental Tech



50X

Matlab Computing
AccelerEyes



100X

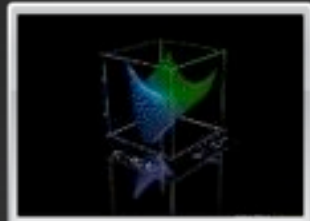
Astrophysics
RIKEN

GPUs Accelerate Science



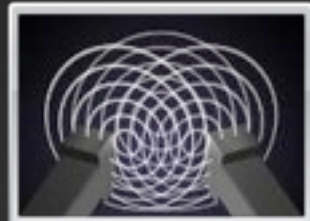
149X

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

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

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

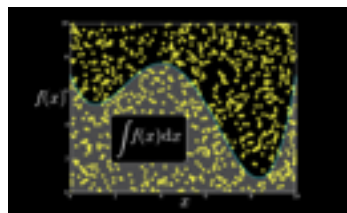
Maximum
Performance

GPU Accelerated Libraries

“Drop-in” Acceleration for your Applications



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



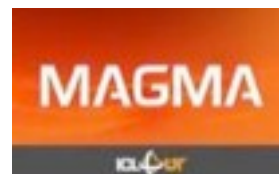
NVIDIA NPP



Vector Signal
Image Processing



GPU Accelerated
Linear Algebra



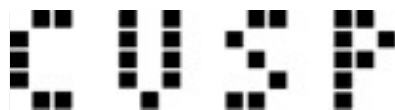
Matrix Algebra on
GPU and Multicore



NVIDIA cuFFT



IMSL Library



Sparse Linear Algebra

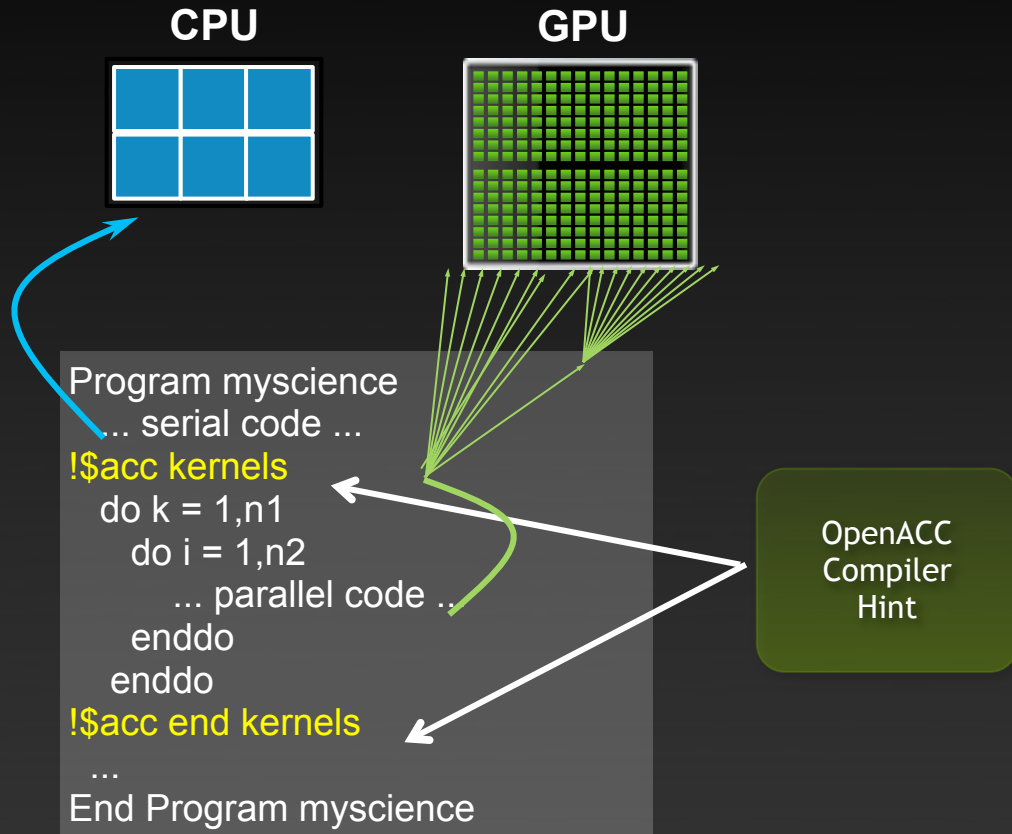


Building-block Algorithms



C++ Templated
Parallel Algorithms

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

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);
```

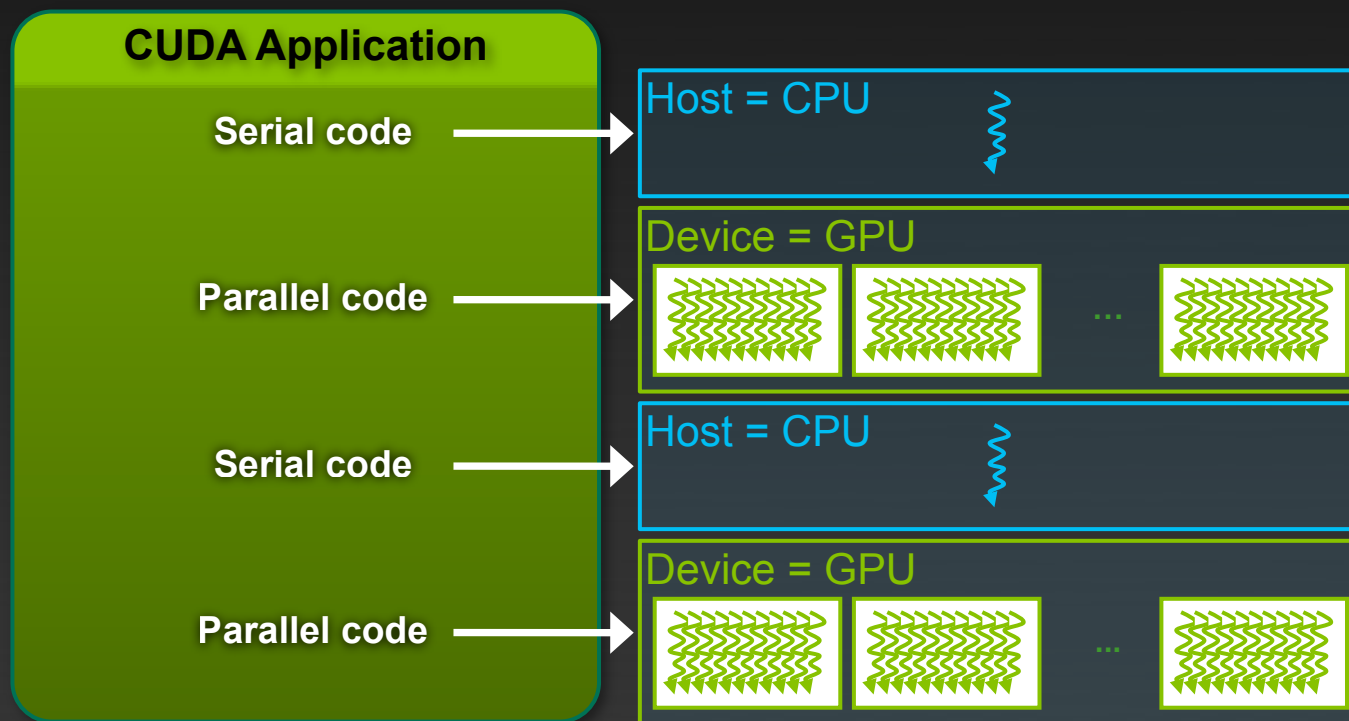
```
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

<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**)





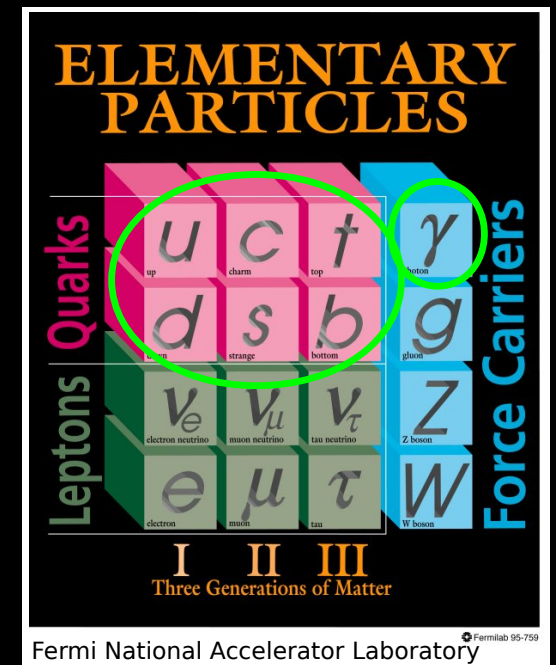
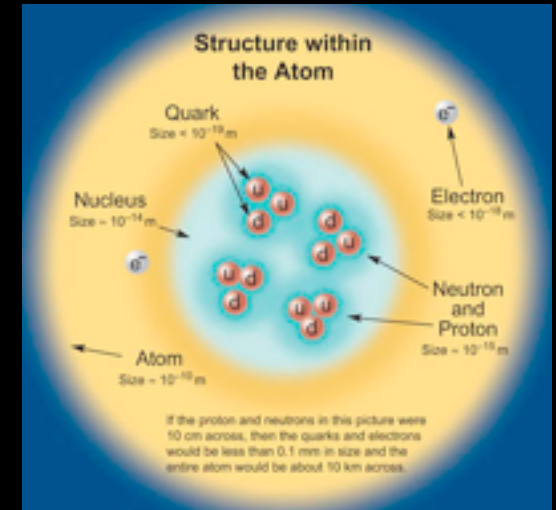
Quantum Chromodynamics

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...



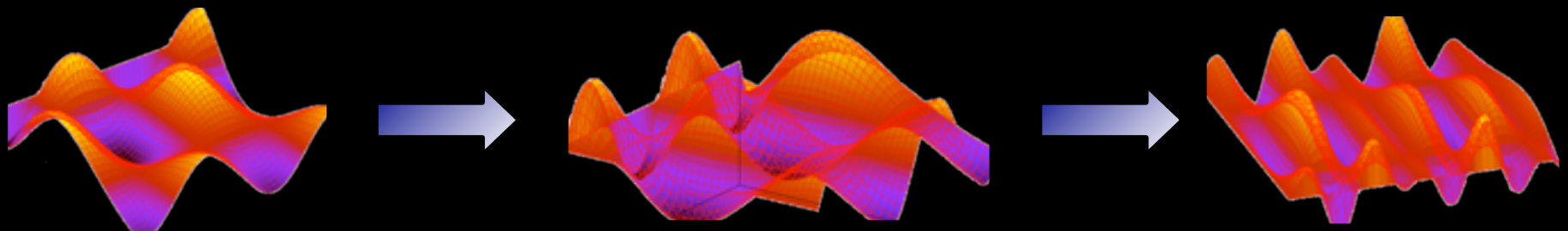
Lattice Quantum Chromodynamics

- Theory is highly non-linear \Rightarrow cannot solve directly
- Must resort to numerical methods to make predictions
- Lattice QCD
 - Discretize spacetime \Rightarrow 4-d dimensional lattice of size $L_x \times L_y \times L_z \times L_t$
 - Finitize spacetime \Rightarrow periodic boundary conditions
 - PDEs \Rightarrow finite difference equations
- High-precision tool that allows physicists to explore the contents of nucleus from the comfort of their workstation (supercomputer)
- Consumer of 10-20% of North American supercomputer cycles

Steps in a lattice QCD calculation

1. Generate an ensemble of gluon field (“gauge”) configurations

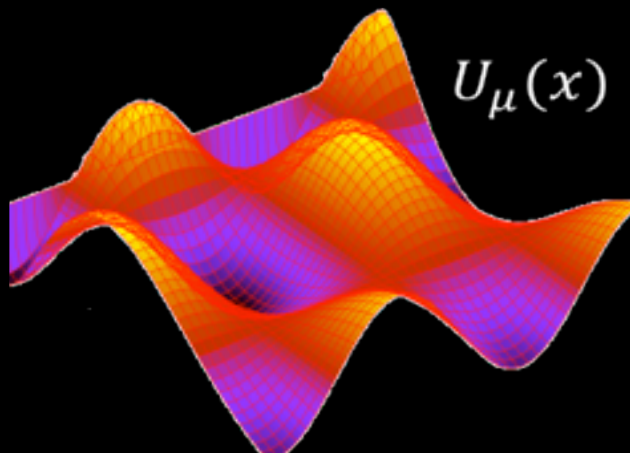
- Produced in sequence, with hundreds needed per ensemble. This requires $> O(10 \text{ Tflops})$ sustained for several months (traditionally Crays, Blue Genes, etc.)
- 50-90% of the runtime is in the linear solver



Steps in a lattice QCD calculation

2. “Analyze” the configurations

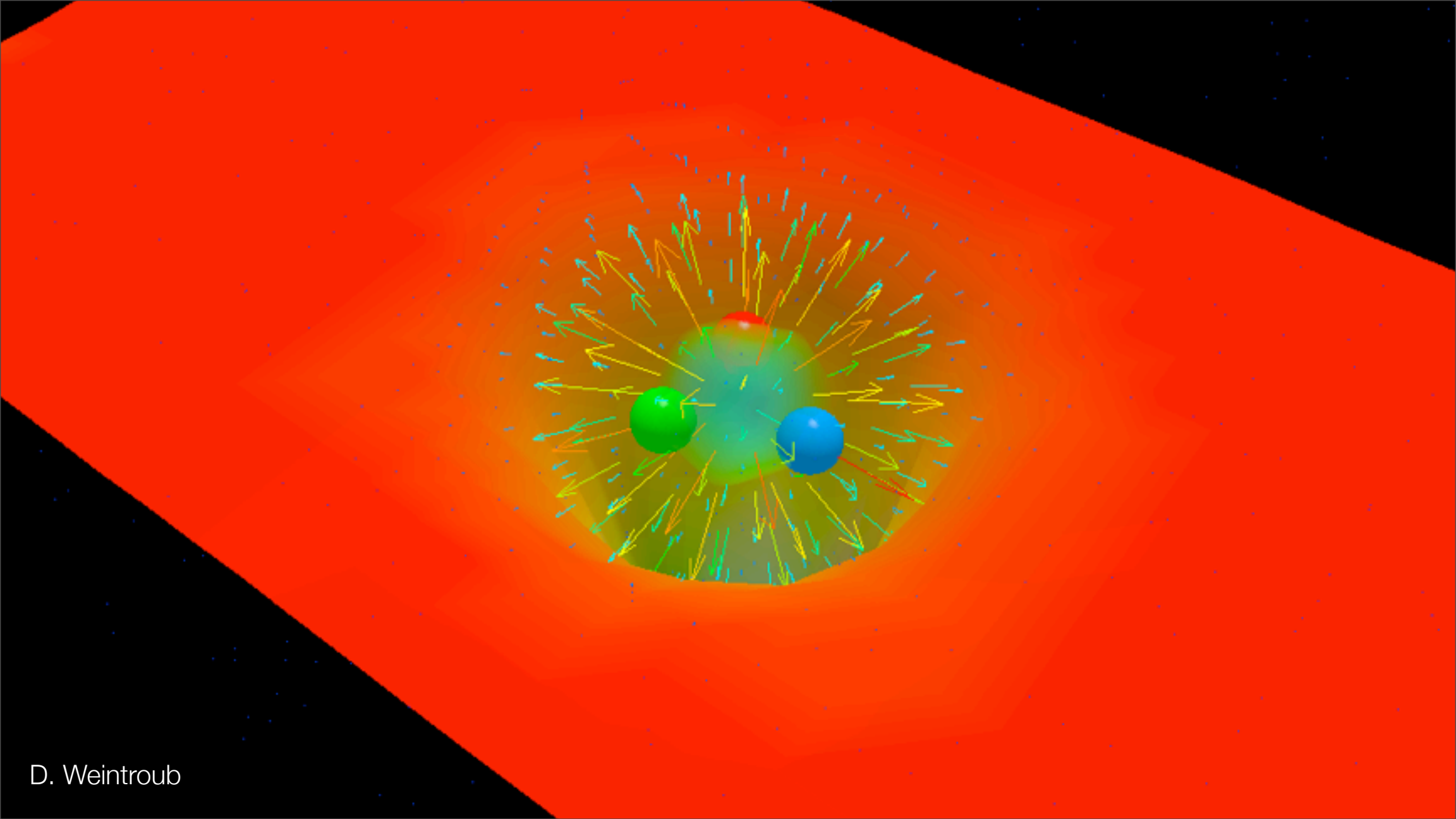
- Can be farmed out, assuming **O(1 Tflops)** per job.
- **80-99% of the runtime is in the linear solver**
Task parallelism means that clusters reign supreme here



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

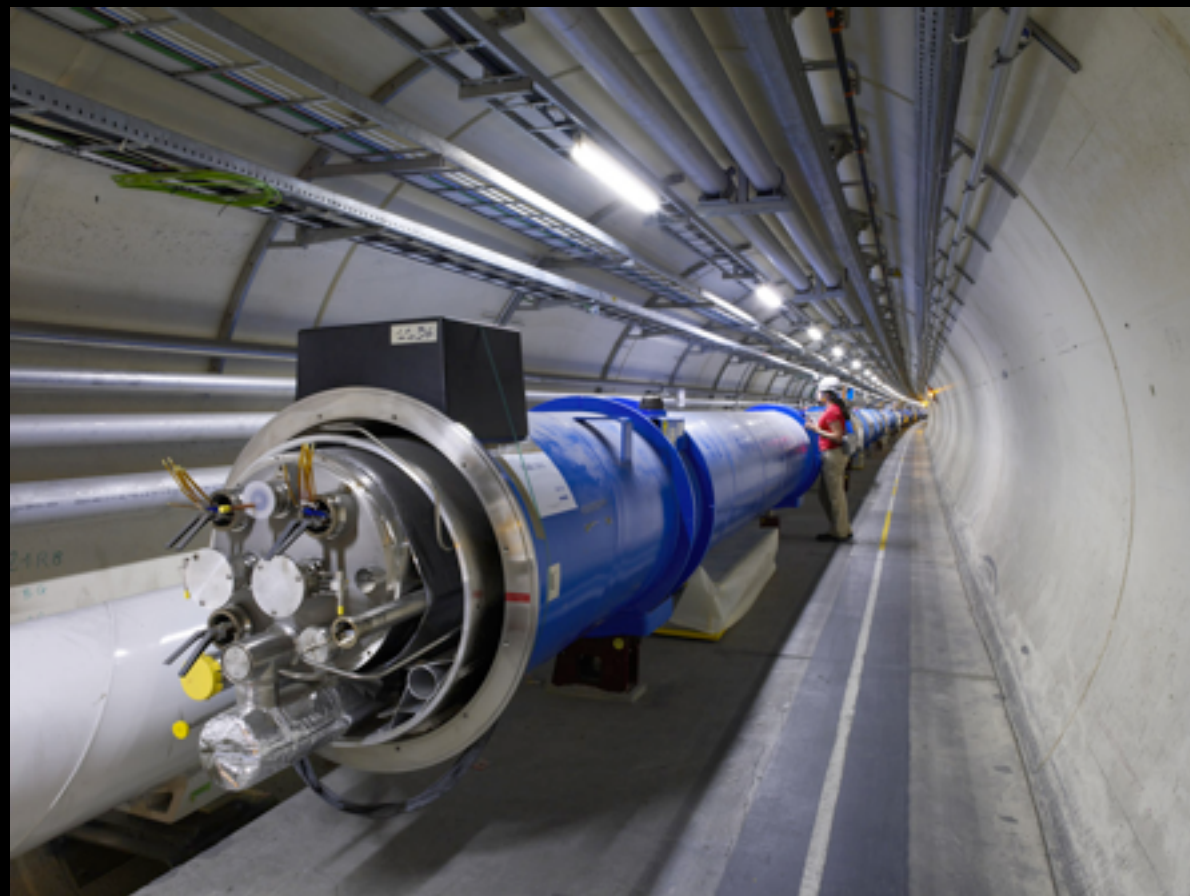
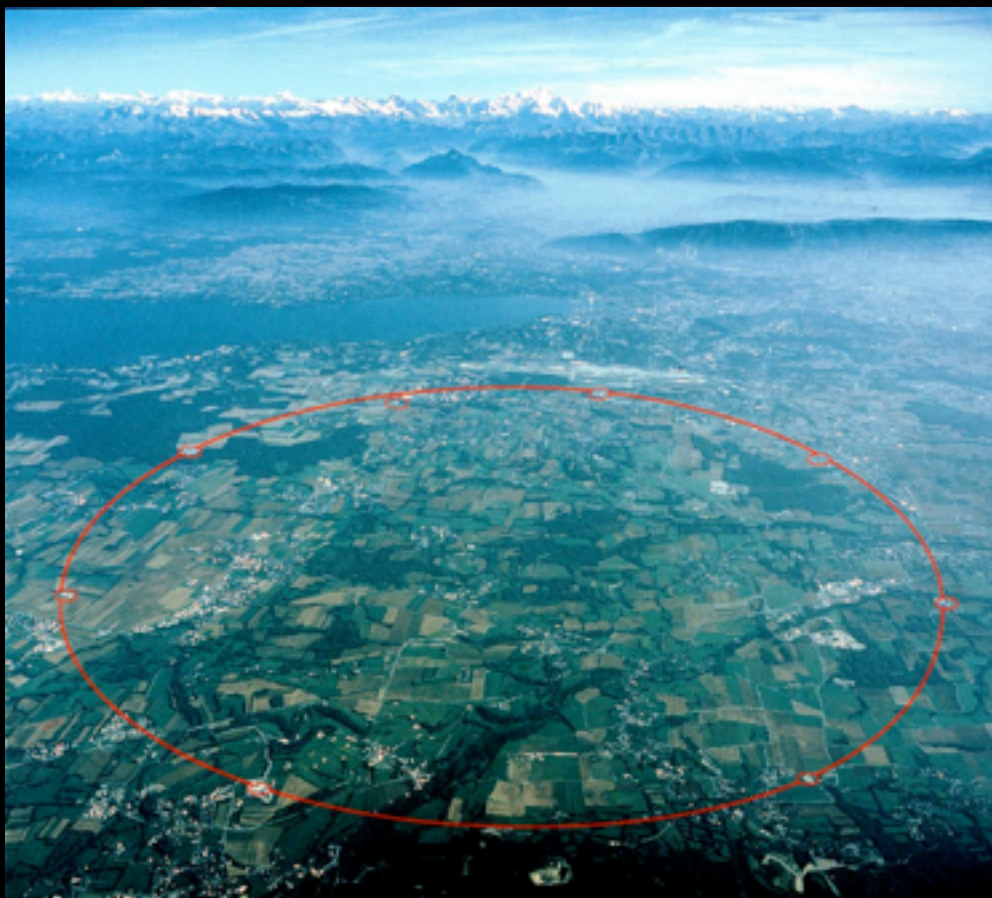
or “ $Ax = b$ ”

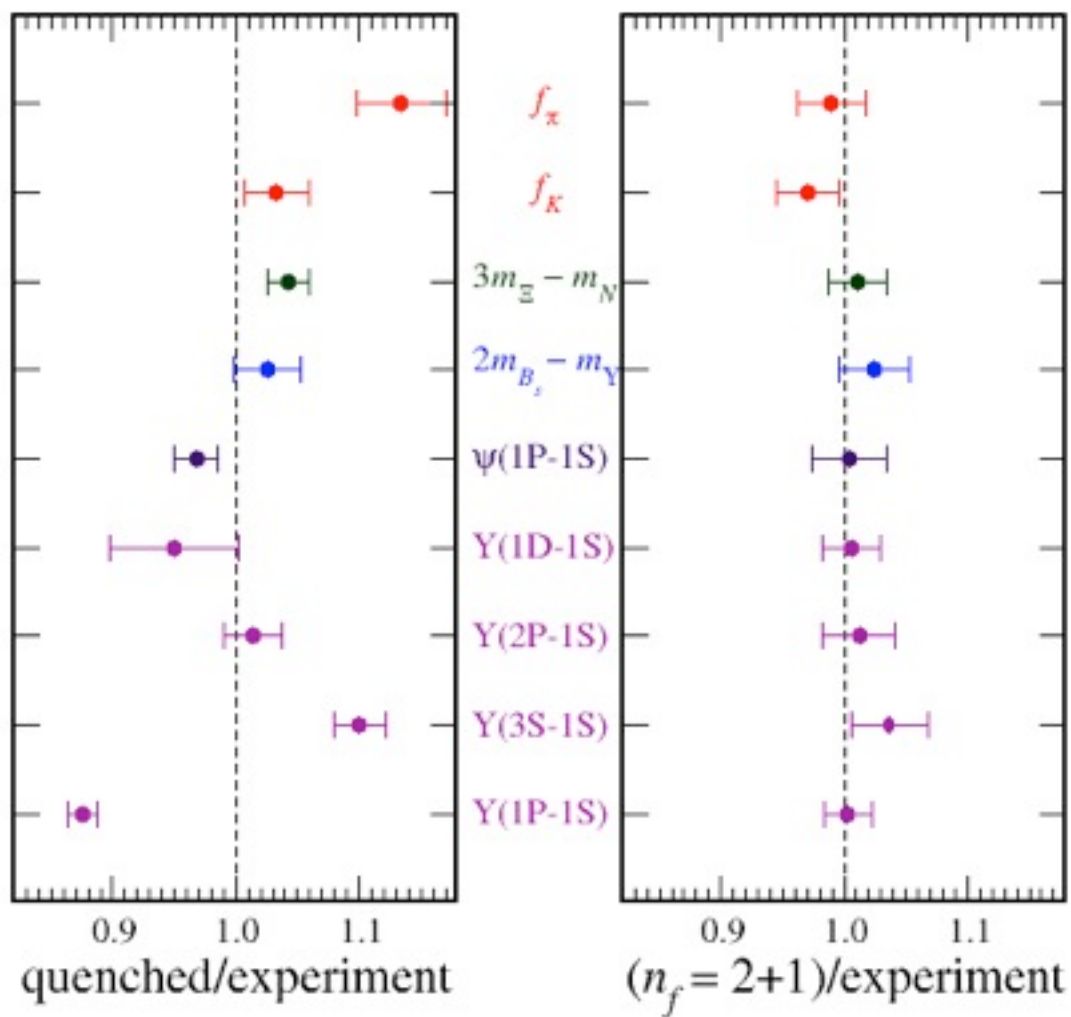
D. van der Vort



D. Weintroub

Tuesday, June 18, 13





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

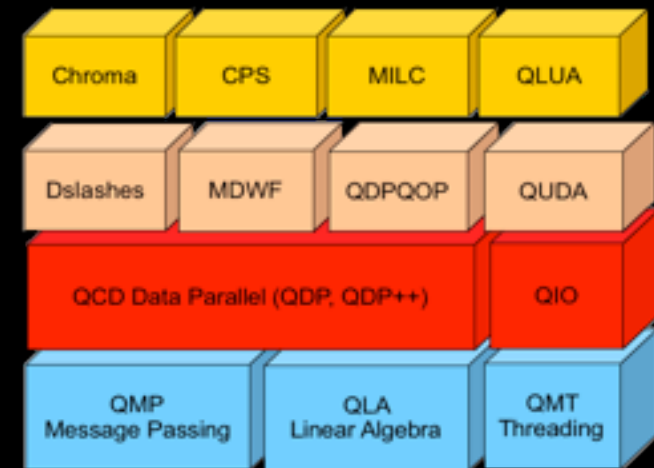
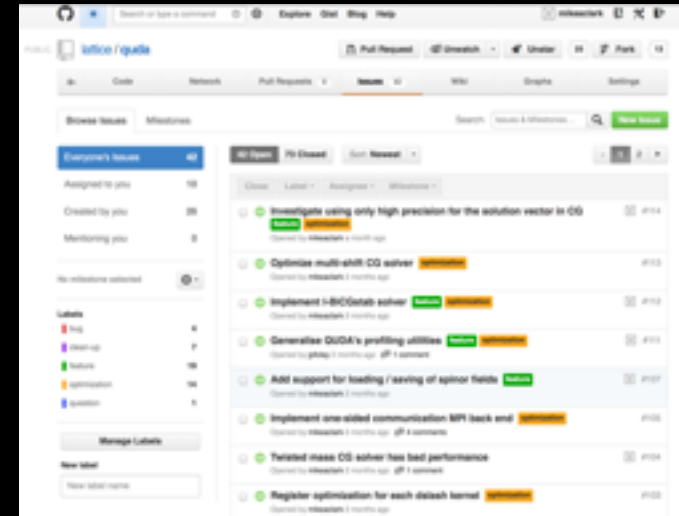
Tuesday, June 18, 13

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

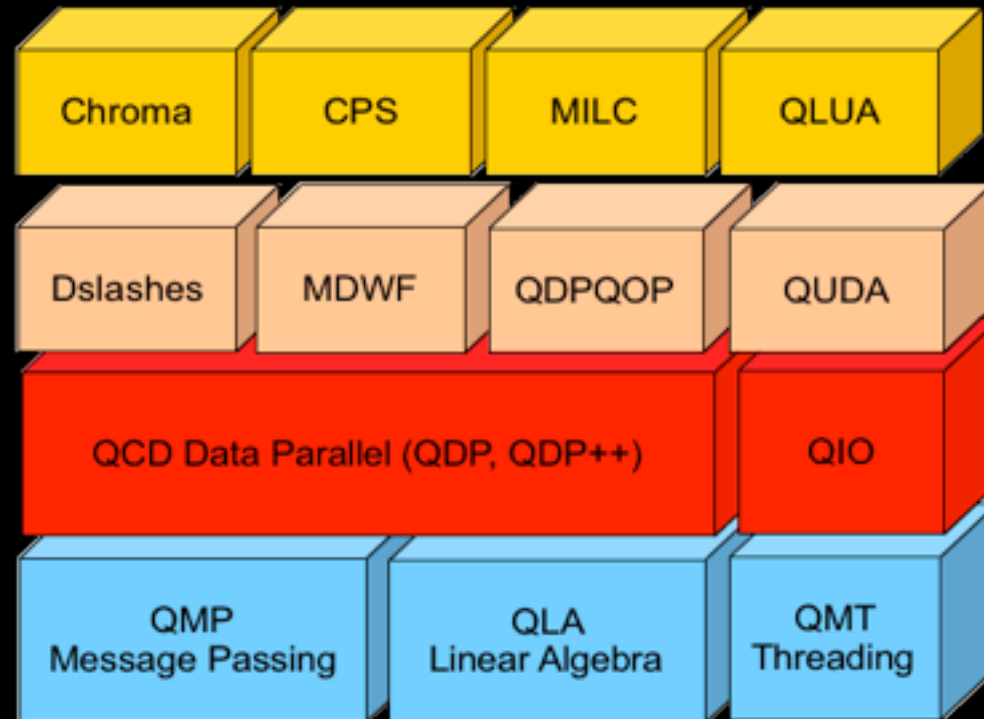
- Ron Babich (NVIDIA)
- Kip Barros (LANL)
- Rich Brower (Boston University)
- Justin Foley (University of Utah)
- Joel Giedt (Rensselaer Polytechnic Institute)
- Steve Gottlieb (Indiana University)
- Bálint Joó (Jlab)
- Claudio Rebbi (Boston University)
- Guochun Shi (NCSA -> Google)
- Alexei Strelchenko (Cyprus Institute -> FNAL)
- Frank Winter (UoE -> Jlab)



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
 - Free from the restrictions of DSLs, e.g., multigrid in QDP

USQCD software stack



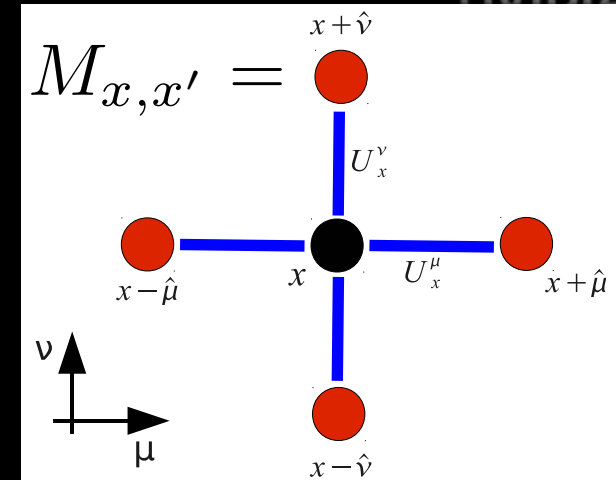
(Many components developed under the DOE SciDAC program)



BYDIA

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 $\in SU(3)$)
- Typically use Krylov solvers to solve $Mx = b$
 - Performance-critical kernel is the SpMV
- Stencil application:
 - Load neighboring spinors, multiply by the inter-connecting link matrix, sum and store



Wilson Matrix

Dirac spin projector matrices
(4x4 spin space)

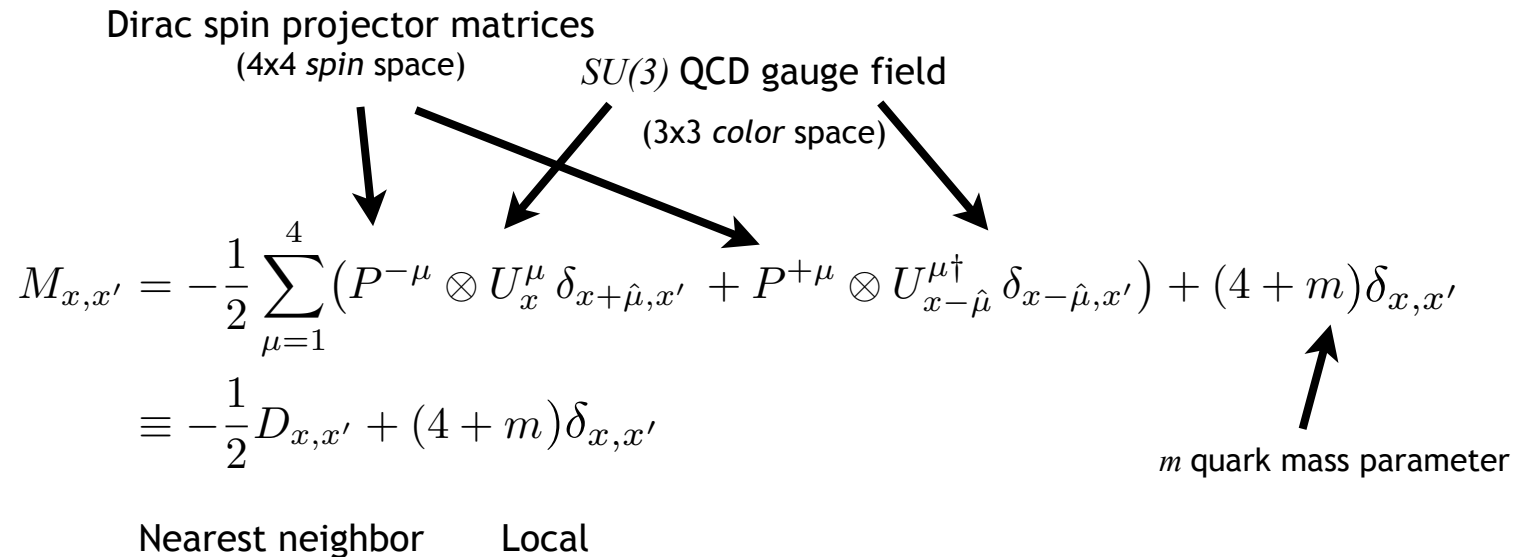
$SU(3)$ QCD gauge field
(3x3 color space)

$$M_{x,x'} = -\frac{1}{2} \sum_{\mu=1}^4 (P^{-\mu} \otimes U_x^\mu \delta_{x+\hat{\mu},x'} + P^{+\mu} \otimes U_{x-\hat{\mu}}^{\mu\dagger} \delta_{x-\hat{\mu},x'}) + (4 + m) \delta_{x,x'}$$

$$\equiv -\frac{1}{2} D_{x,x'} + (4 + m) \delta_{x,x'}$$

Nearest neighbor Local

m quark mass parameter



Wilson Matrix

Dirac spin projector matrices
(4x4 spin space)

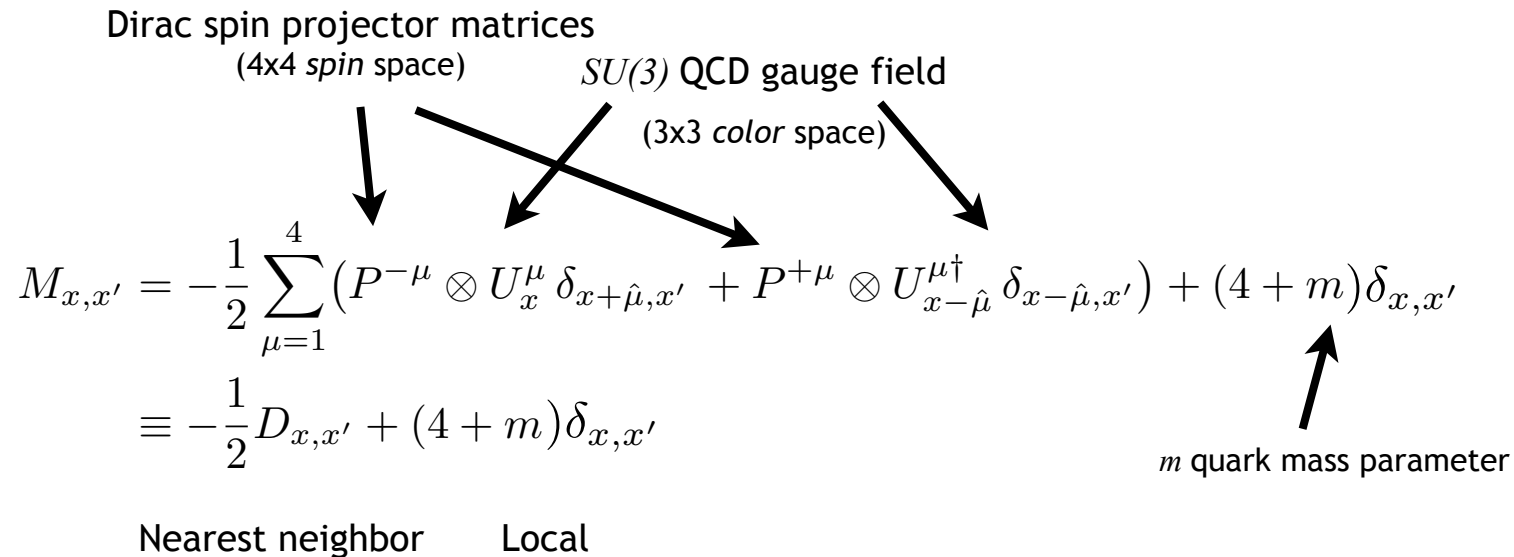
$SU(3)$ QCD gauge field
(3x3 color space)

$$M_{x,x'} = -\frac{1}{2} \sum_{\mu=1}^4 (P^{-\mu} \otimes U_x^\mu \delta_{x+\hat{\mu},x'} + P^{+\mu} \otimes U_{x-\hat{\mu}}^{\mu\dagger} \delta_{x-\hat{\mu},x'}) + (4 + m)\delta_{x,x'}$$

$$\equiv -\frac{1}{2} D_{x,x'} + (4 + m)\delta_{x,x'}$$

Nearest neighbor Local

m quark mass parameter

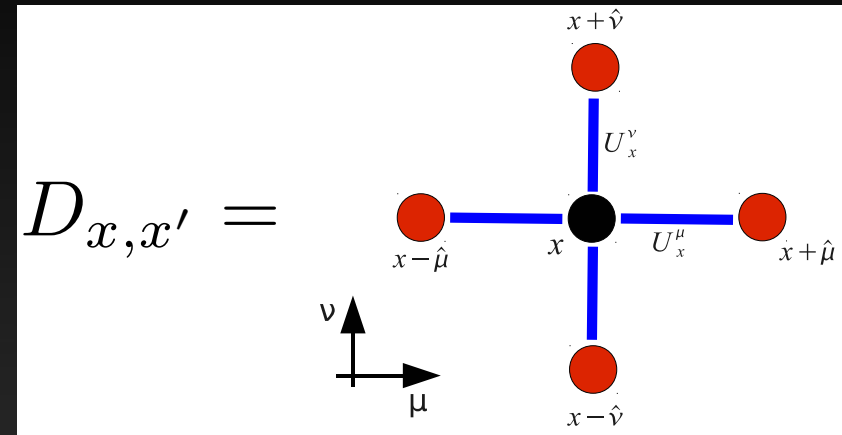


4d nearest-neighbor stencil operator acting on a vector field

Mapping the Wilson Dslash to CUDA



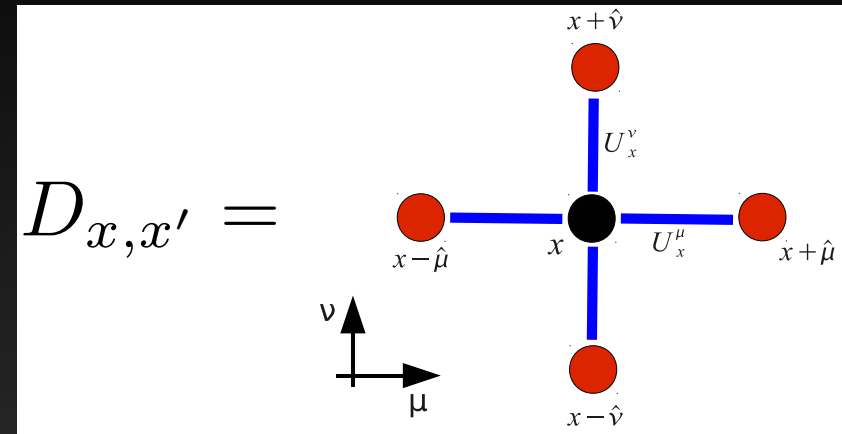
- Assign a single space-time point to each thread
 - $V = XYZT$ threads
 - $V = 24^4 \Rightarrow 3.3 \times 10^6$ threads
 - Fine-grained parallelization
- Looping over direction each thread must
 - Load the neighboring spinor (24 numbers x8)
 - Load the color matrix connecting the sites (18 numbers 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



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

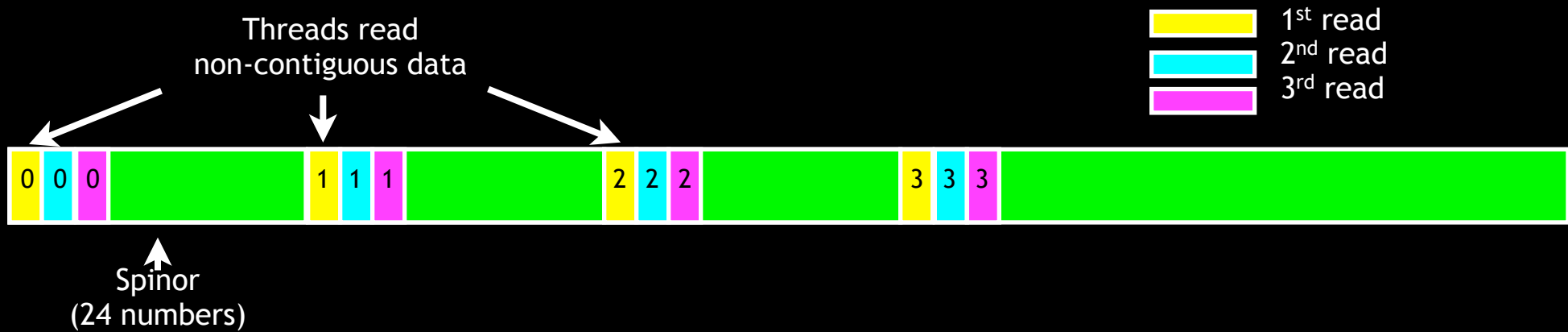


Tesla K20X	
Gflops	3995
GB/s	250
AI	16

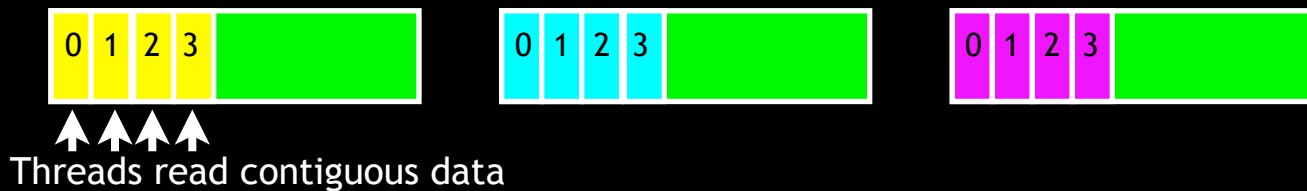
bandwidth bound

Field Ordering

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



- GPUs like Structure of Arrays



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

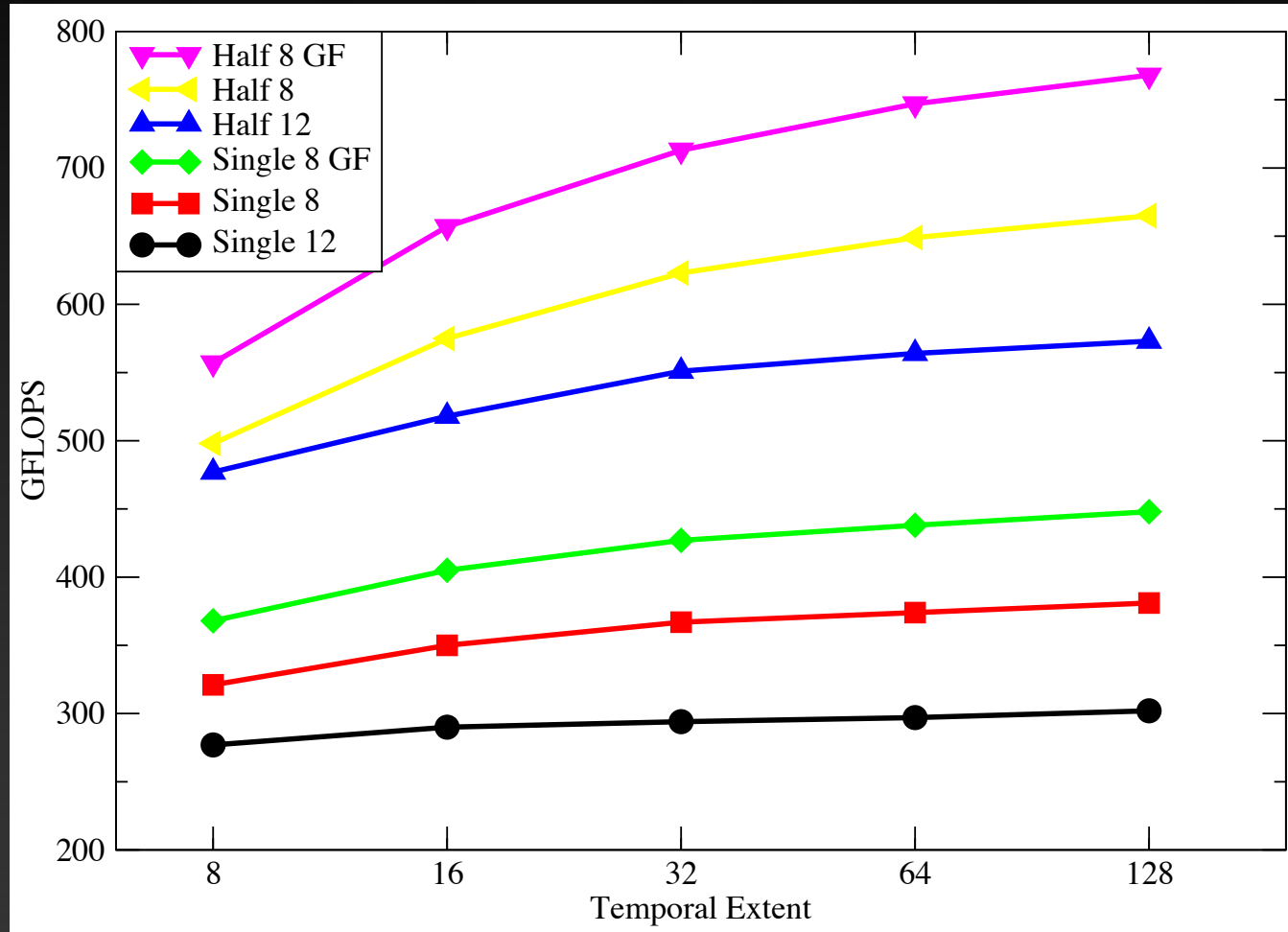
Reducing Memory Traffic

- SU(3) matrices are all unitary complex matrices with $\det = 1$
 - 12-number parameterization: reconstruct full matrix on the fly in registers

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \longrightarrow \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix} \mathbf{c} = (\mathbf{a} \times \mathbf{b})^*$$

- 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 CG performance
 $V = 24^3 \times T$
Wilson-Clover is $\pm 10\%$

GeForce GTX Titan
> 1 TFLOPS

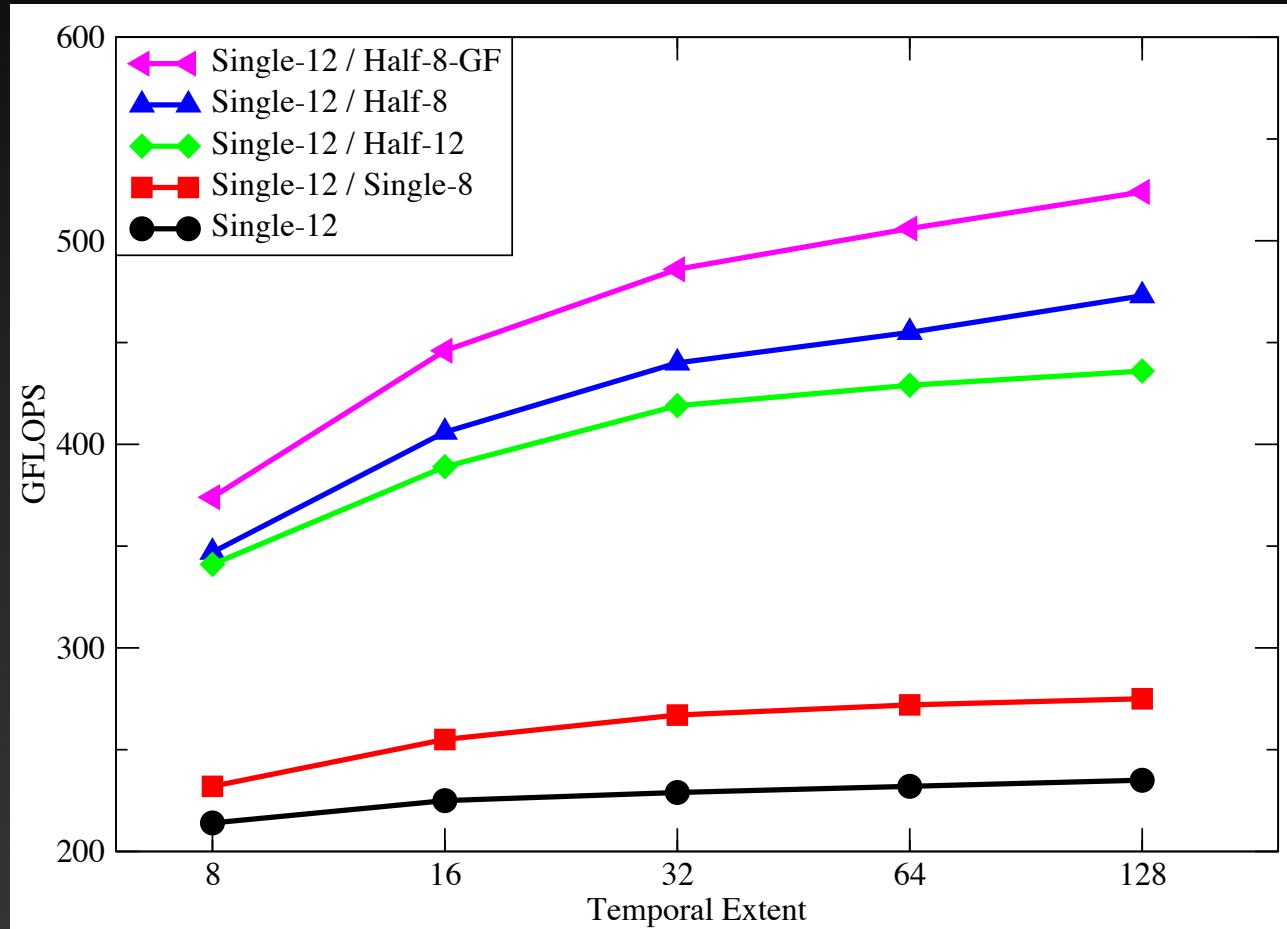
Krylov Solver Implementation

- 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

```
while ( $|r_k| > \epsilon$ ) {  
     $\beta_k = (r_k, r_k) / (r_{k-1}, r_{k-1})$   
     $p_{k+1} = r_k - \beta_k p_k$   
  
     $\alpha = (r_k, r_k) / (p_{k+1}, A p_{k+1})$   
     $r_{k+1} = r_k - \alpha A p_{k+1}$   
     $x_{k+1} = x_k + \alpha p_{k+1}$   
     $k = k+1$   
}
```

Kepler Wilson-Solver Performance

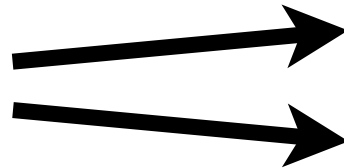


K20X performance
 $V = 24^3 \times T$
Wilson-Clover is $\pm 10\%$
BiCGstab is -10%

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**

High precision
mat-vec and
accumulate



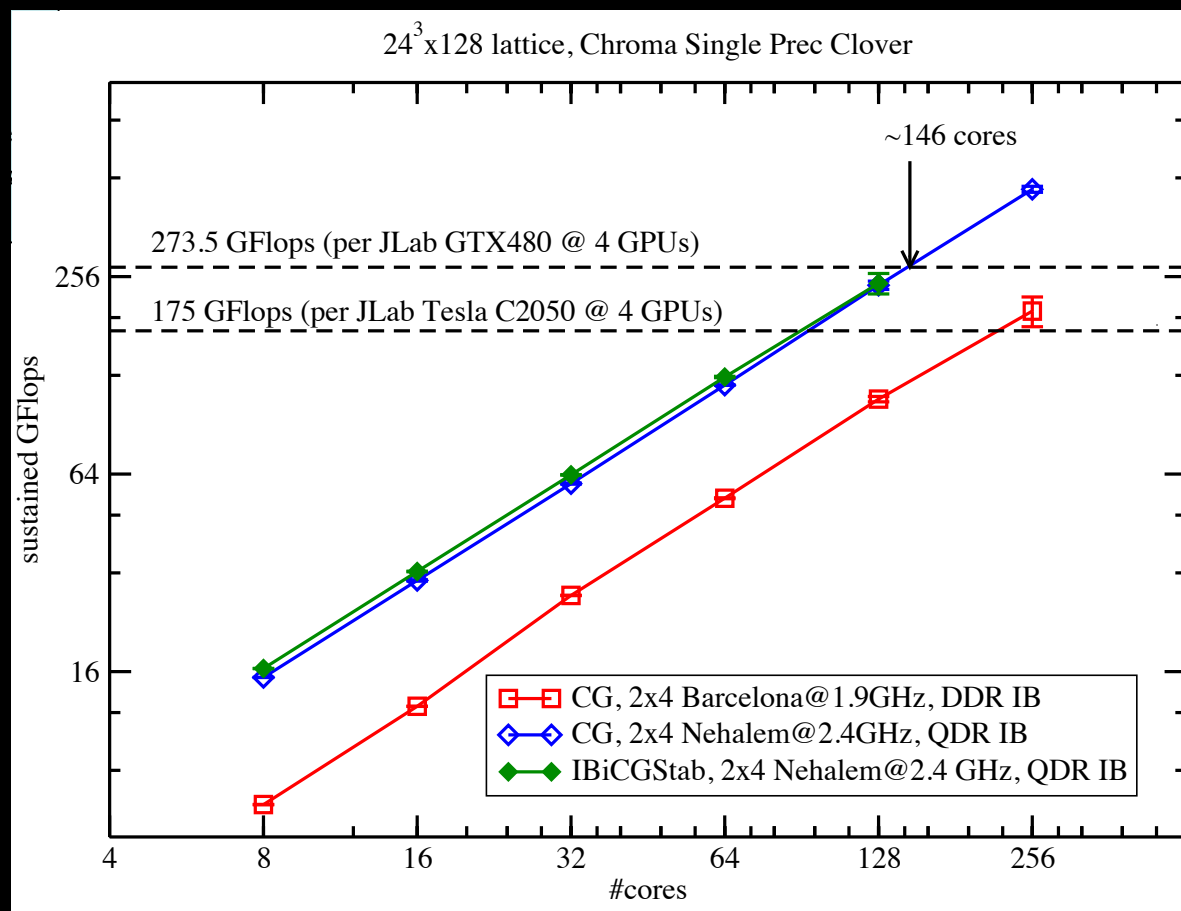
```
while ( | $\mathbf{r}_k$ | >  $\epsilon$  ) {  
     $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$   
    solve  $A\mathbf{p}_k = \mathbf{r}_k$   
     $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$   
}
```



Inner low
precision solve

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

QUDA Performance - Chroma



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



Supercomputing with QUDA

Tuesday, June 18, 13



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



BLUE WATERS
SUSTAINED PETASCALE COMPUTING

Tsubame 2.0, Tianhe,
Blue Waters, etc.

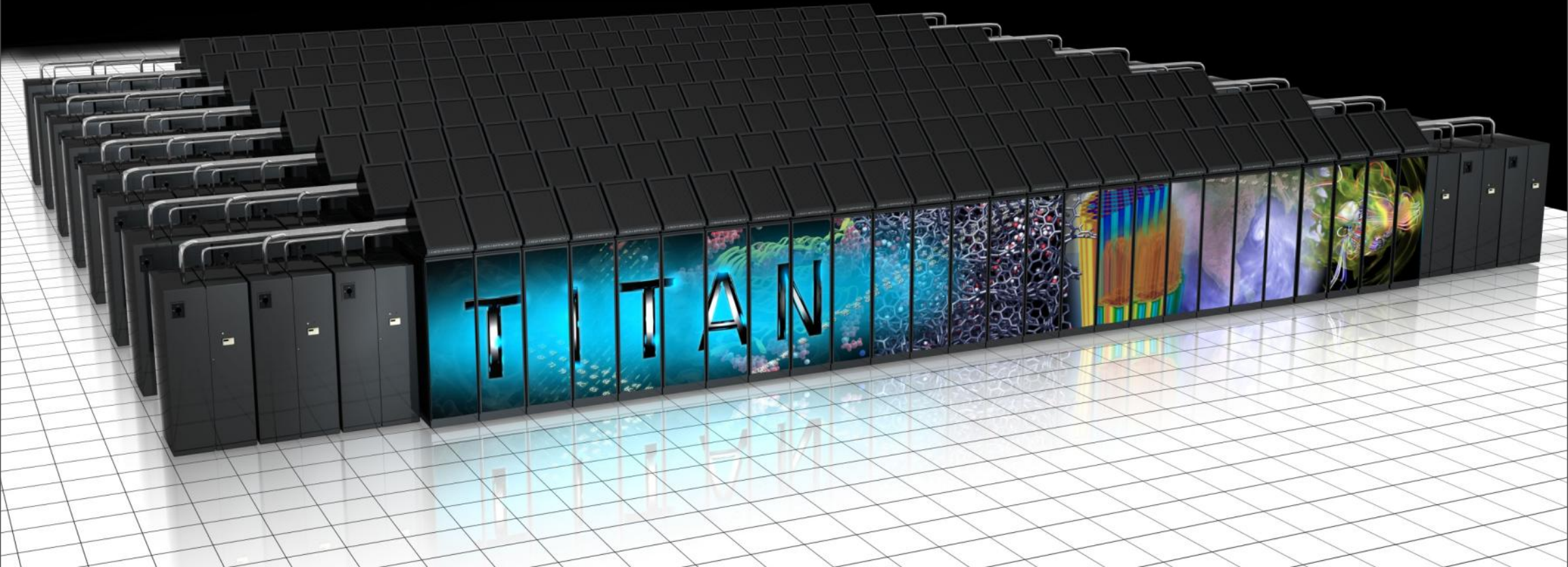
TITAN: World's Fastest Supercomputer



18,688 Tesla K20X GPUs

27 Petaflops Peak, 17.59 Petaflops on Linpack

90% of Performance from GPUs



Multiple GPUs

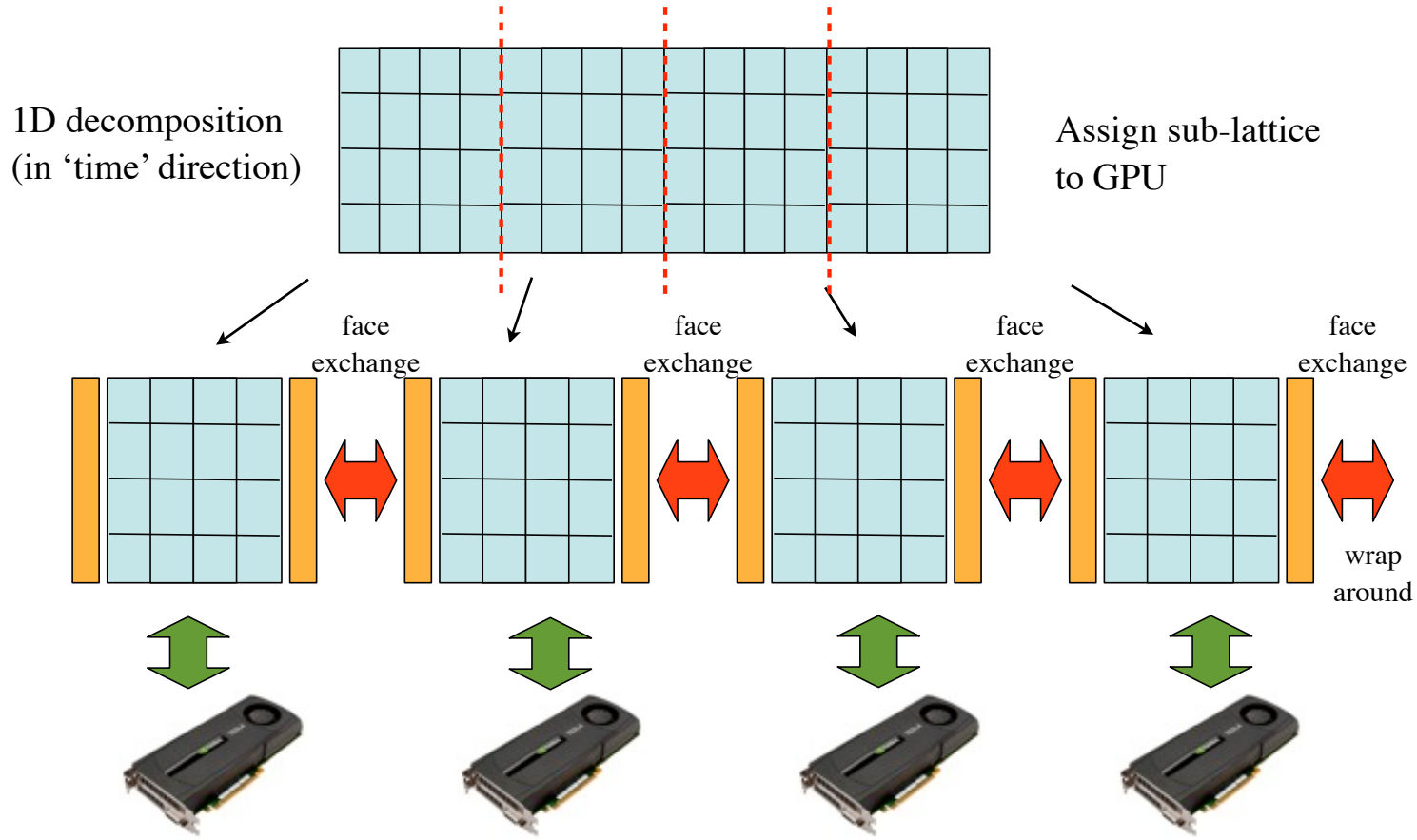


- Many different mechanisms for controlling multiple GPUs
 - MPI processes
 - CPU threads
 - Multiple GPU per thread and do explicit switching
 - Combinations of the above
- QUDA uses the simplest: 1 GPU per MPI process
 - Allows partitioning over node with multiple devices and multiple nodes
 - `cudaSetDevice(local_mpi_rank);`

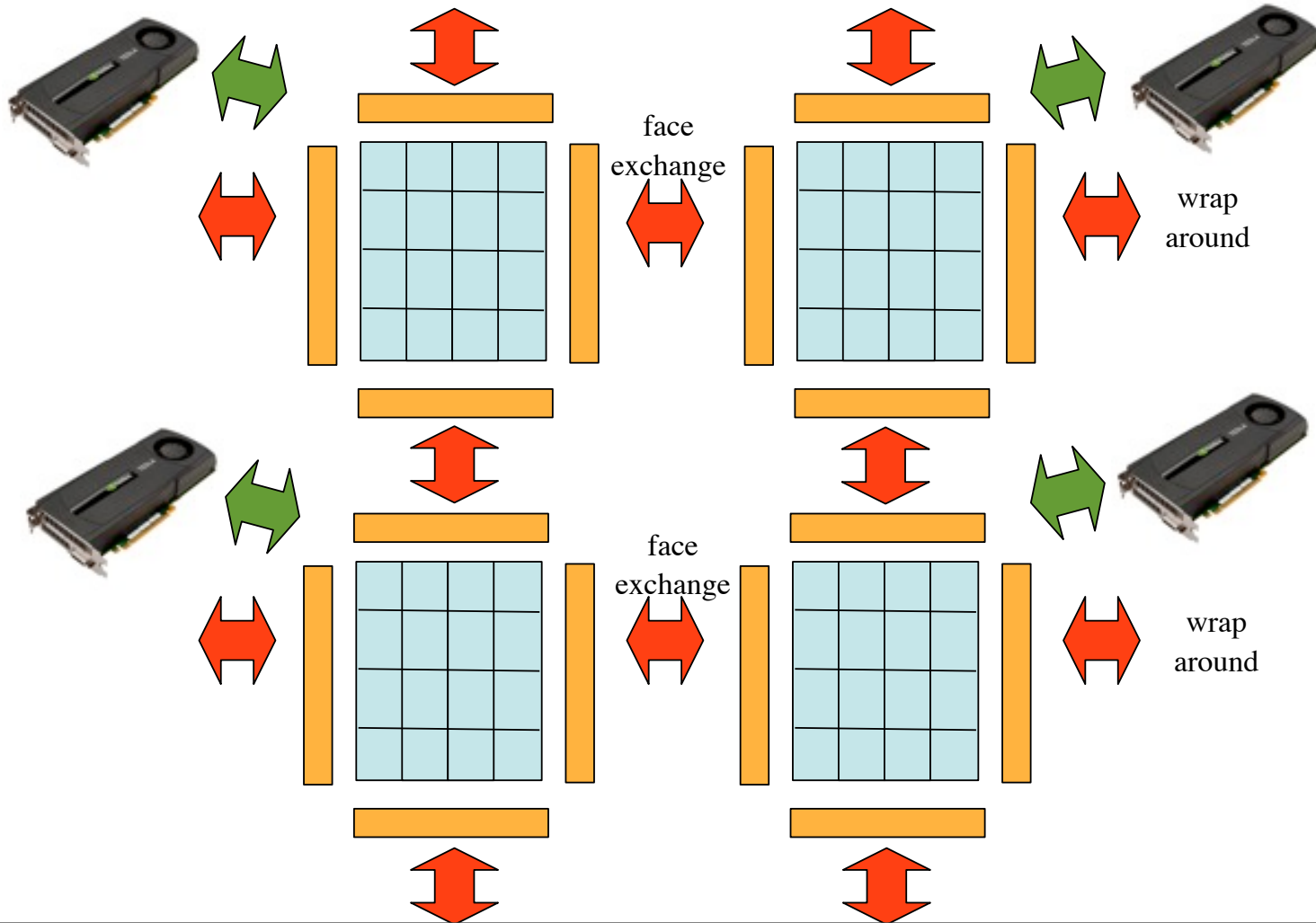
CUDA Stream API

- 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



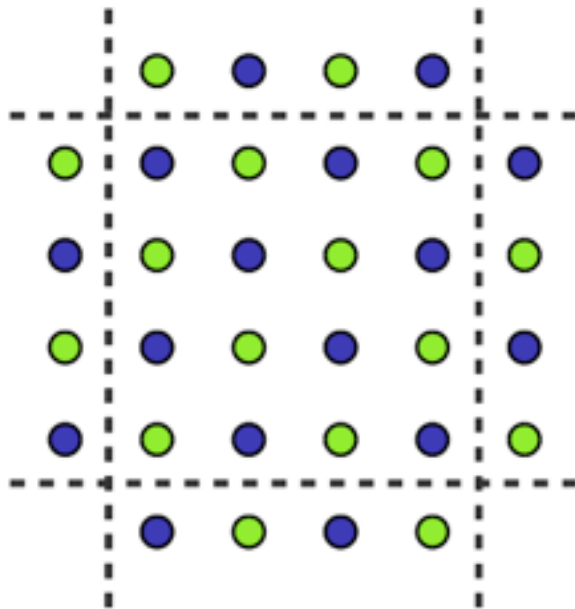
Multi-dimensional lattice decomposition



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

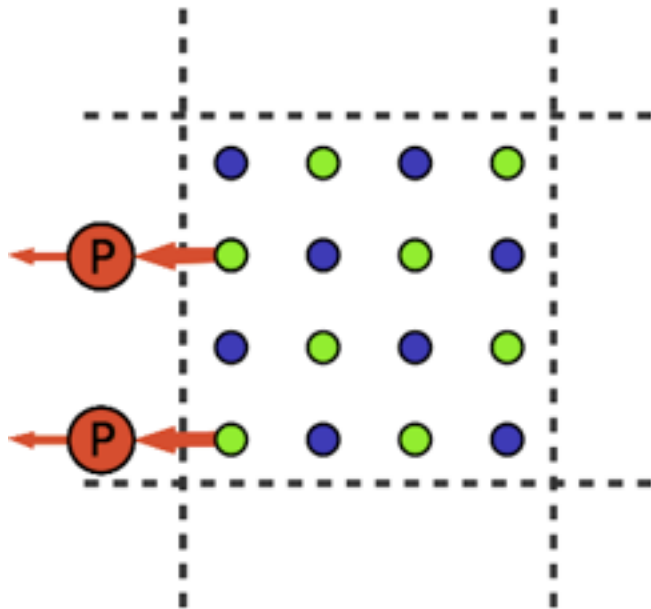
Multi-dimensional Kernel Computation



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

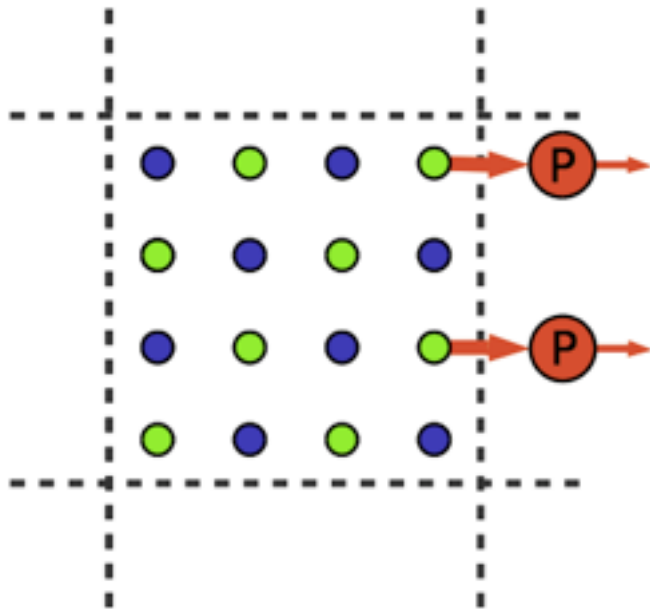
Multi-dimensional Kernel Computation



Step 1

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

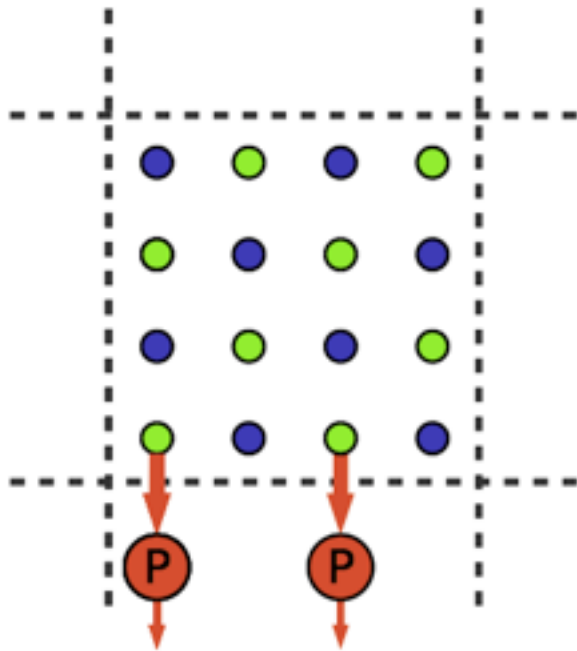
Multi-dimensional Kernel Computation



Step 1

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

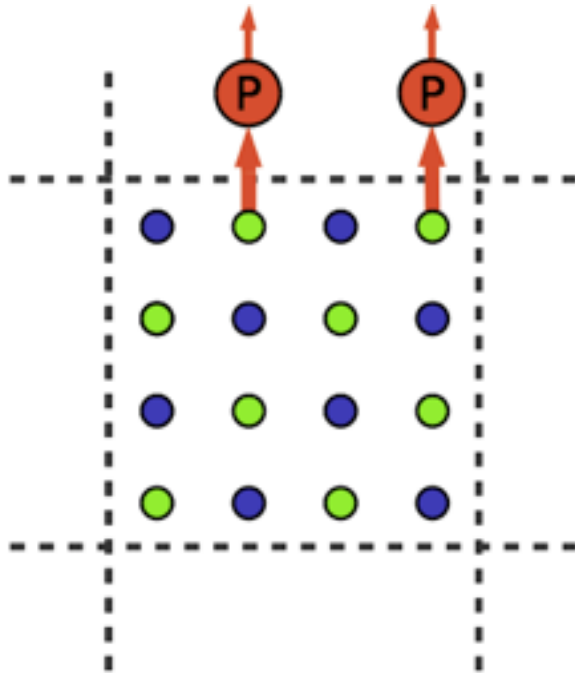
Multi-dimensional Kernel Computation



Step 1

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

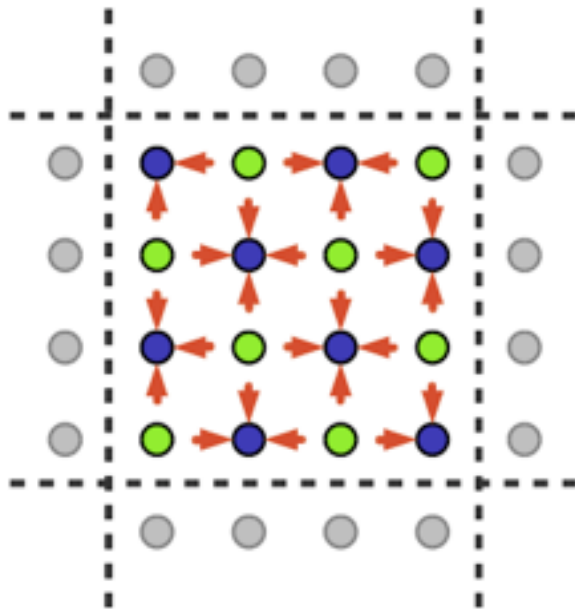
Multi-dimensional Kernel Computation



Step 1

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

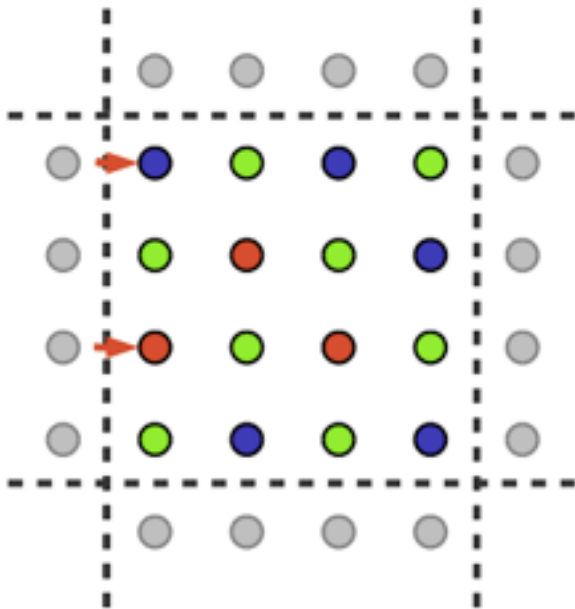
Multi-dimensional Kernel Computation



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



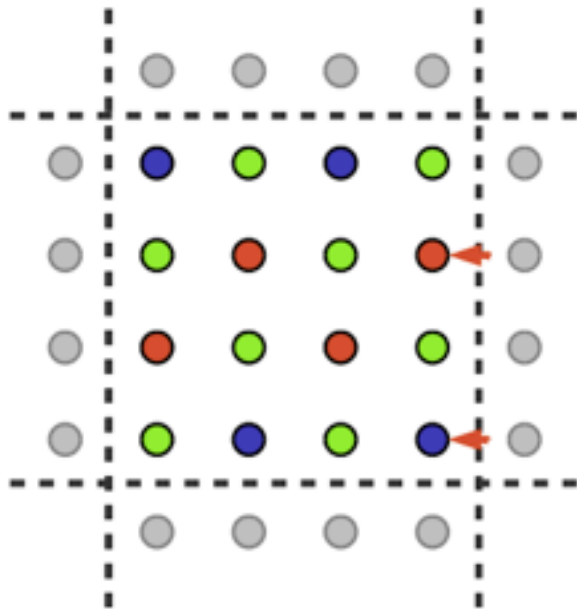
Step 3

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



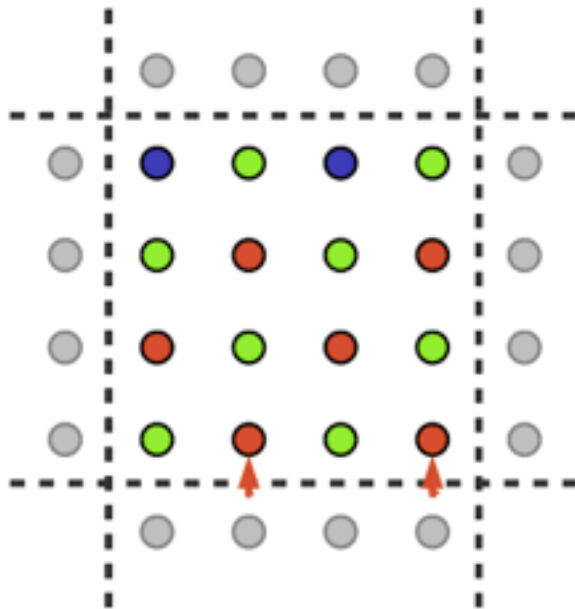
Step 3

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



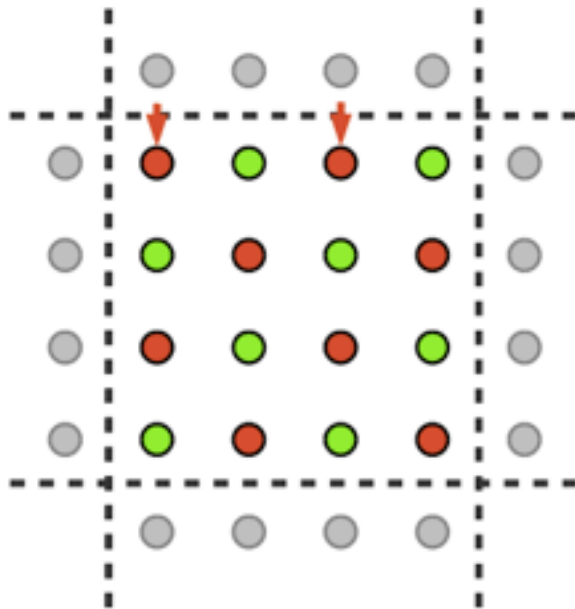
Step 3

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



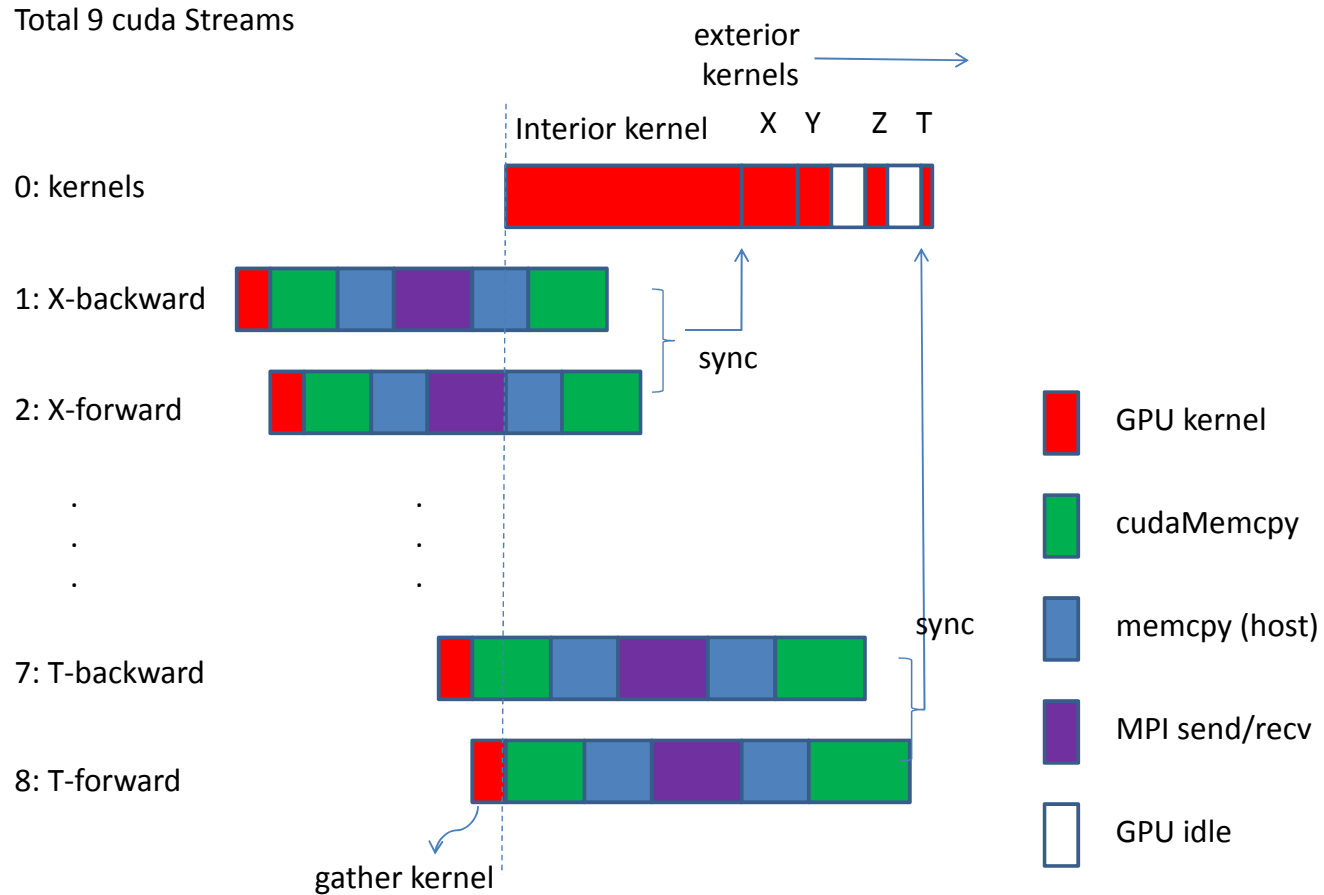
Step 3

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

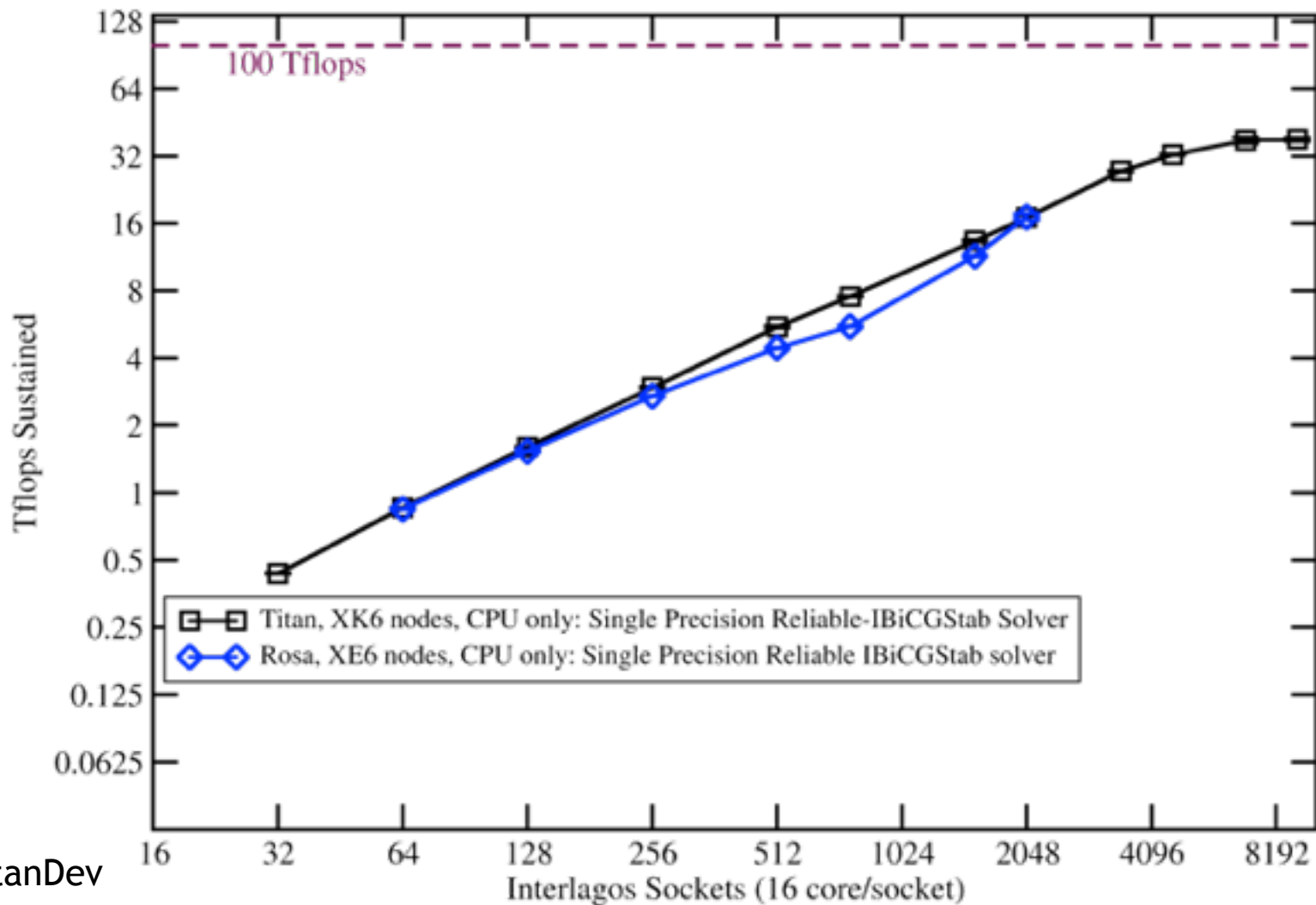
A given boundary kernel must wait for its ghost
zone to arrive

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

Multi-dimensional Communications Pipeline

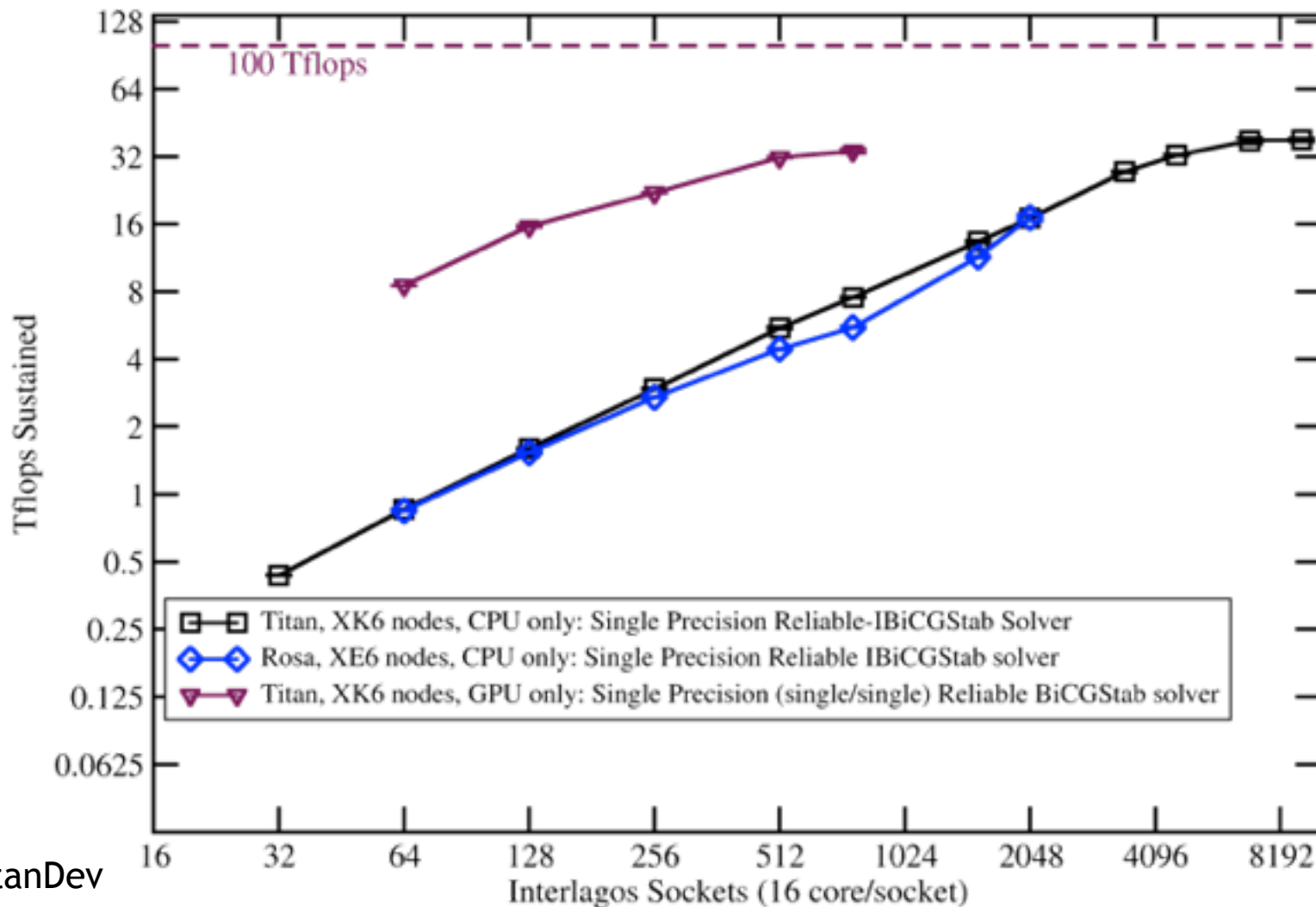


Strong Scaling: $48^3 \times 512$ Lattice (Weak Field), Chroma + QUDA



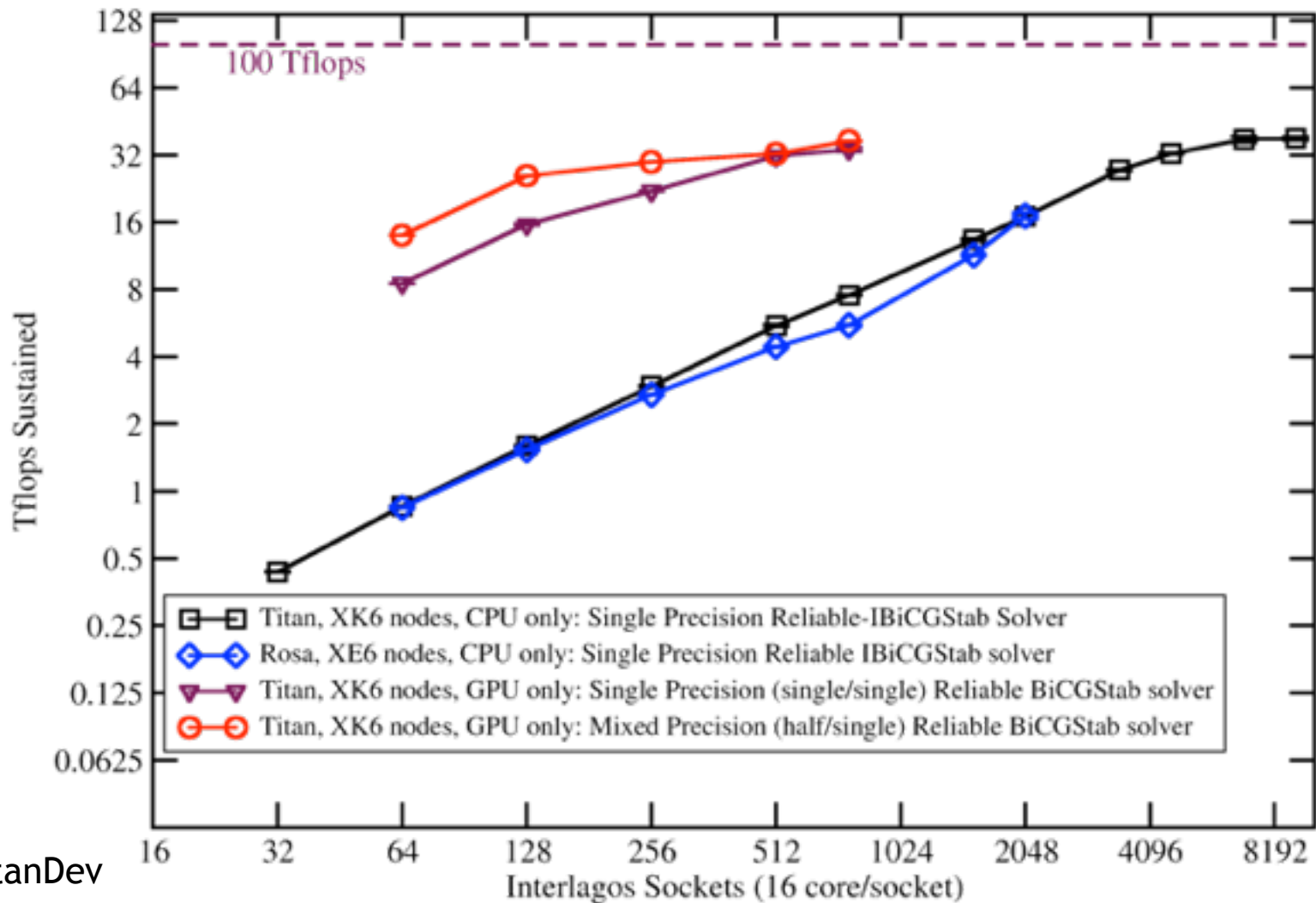
Results from TitanDev
- Clover propagator
- $48^3 \times 512$ aniso clover
- scaling up 768 GPUs

Strong Scaling: $48^3 \times 512$ Lattice (Weak Field), Chroma + QUDA



Results from TitanDev
- Clover propagator
- $48^3 \times 512$ aniso clover
- scaling up 768 GPUs

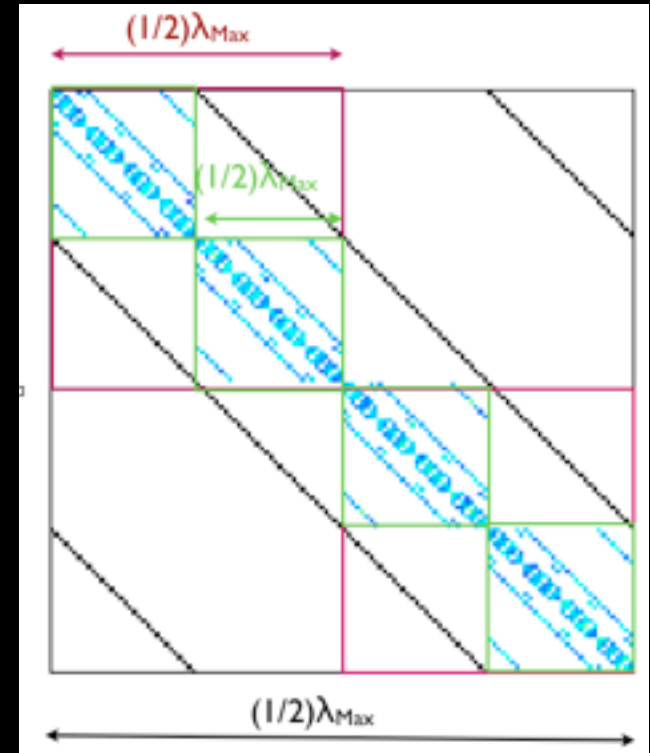
Strong Scaling: $48^3 \times 512$ Lattice (Weak Field), Chroma + QUDA



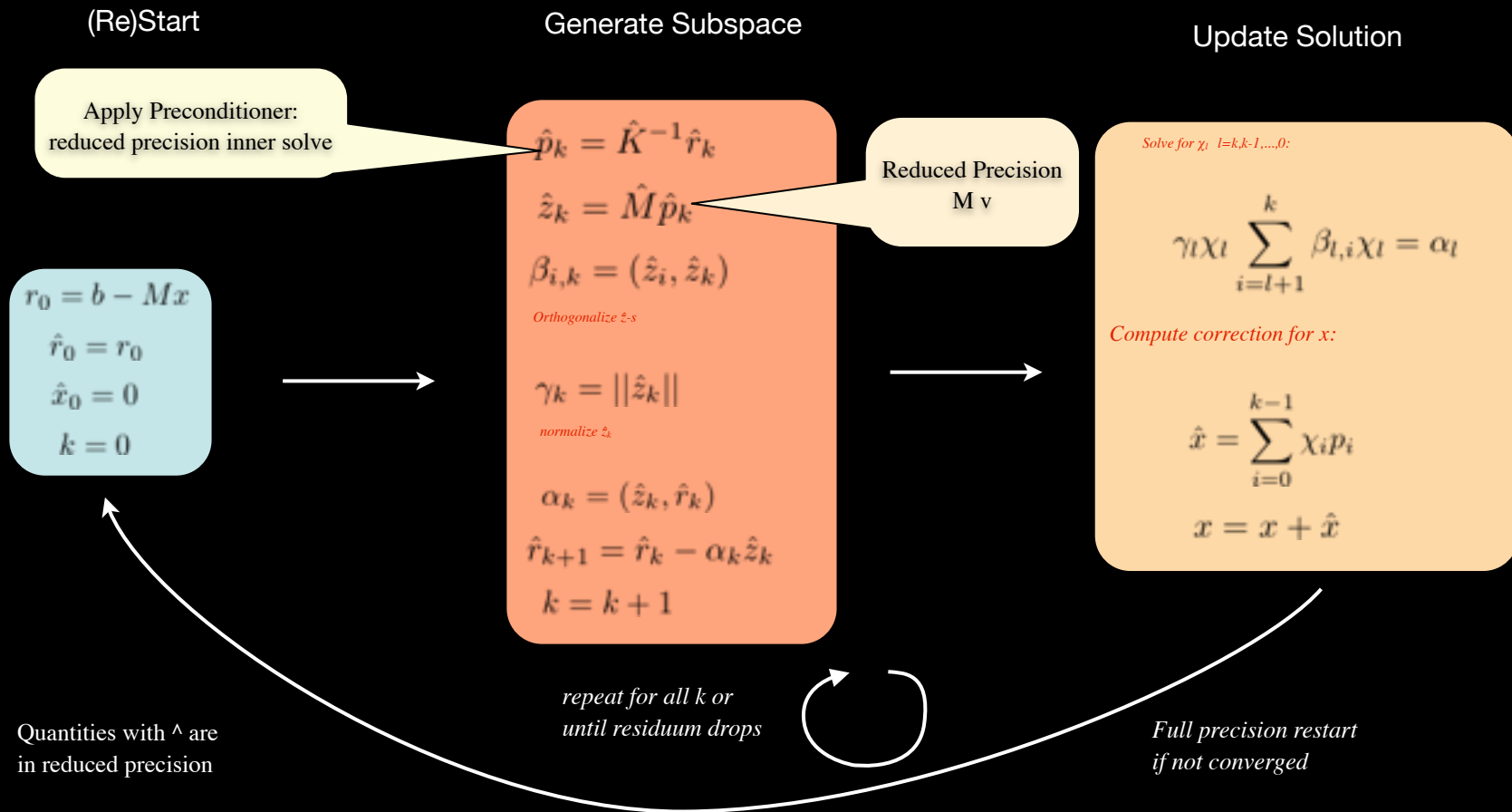
Results from TitanDev
- Clover propagator
- $48^3 \times 512$ aniso clover
- scaling up 768 GPUs

Domain Decomposition

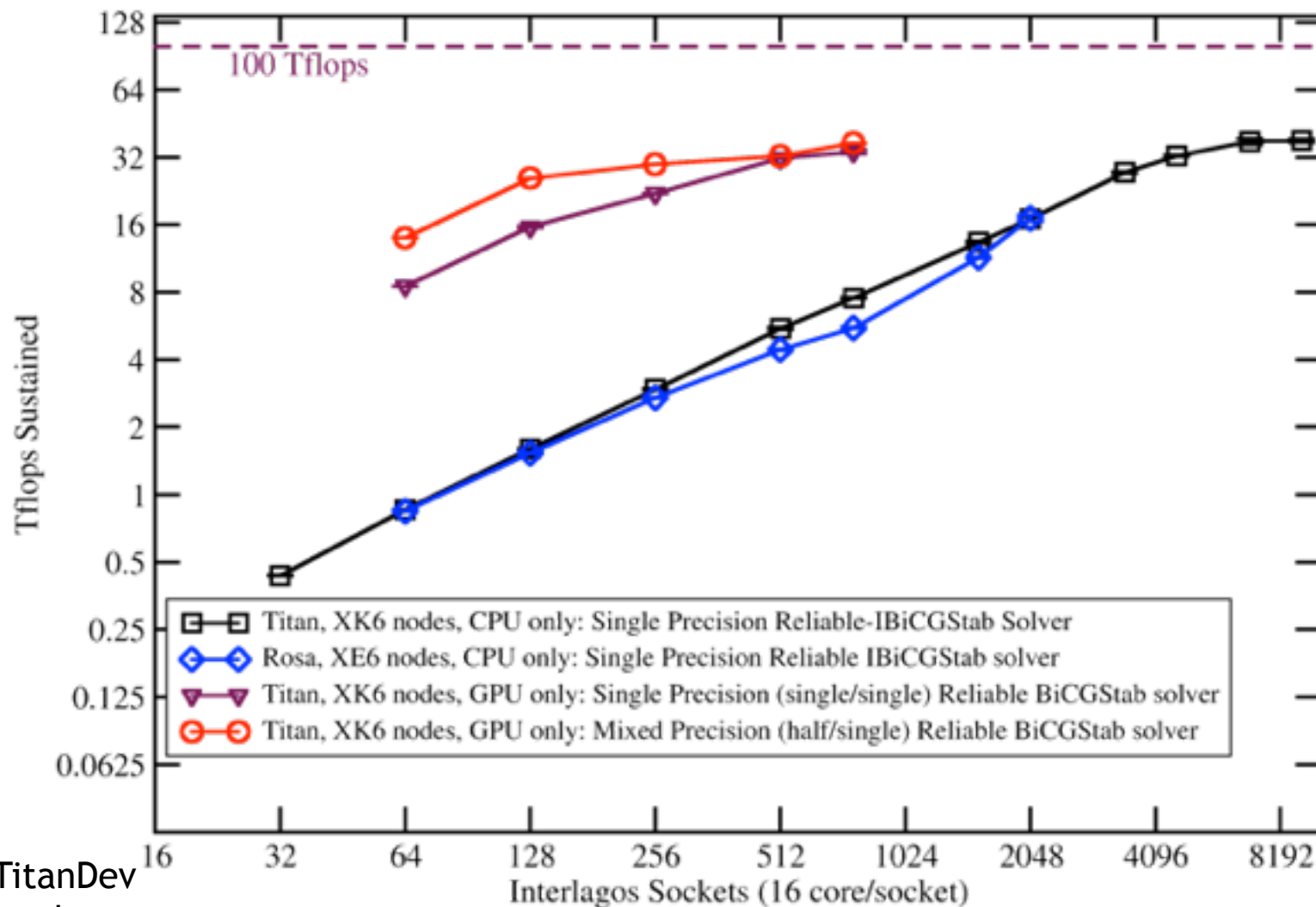
- Non-overlapping blocks - simply have to switch off inter-GPU communication
- Preconditioner is a gross approximation
 - Use an iterative solver to solve each domain system
 - Require only 10 iterations of domain solver \Rightarrow 16-bit
 - Need to use a flexible solver \Rightarrow GCR
- Block-diagonal preconditioner impose λ cutoff
- Finer Blocks lose long-wavelength/low-energy modes
 - keep wavelengths of $\sim O(\Lambda_{\text{QCD}}^{-1})$, $\Lambda_{\text{QCD}}^{-1} \sim 1\text{fm}$
- Aniso clover: ($a_s=0.125\text{fm}$, $a_t=0.035\text{fm}$) \Rightarrow $8^3 \times 32$ blocks are ideal
 - $48^3 \times 512$ lattice: $8^3 \times 32$ blocks \Rightarrow 3456 GPUs



Domain Decomposition

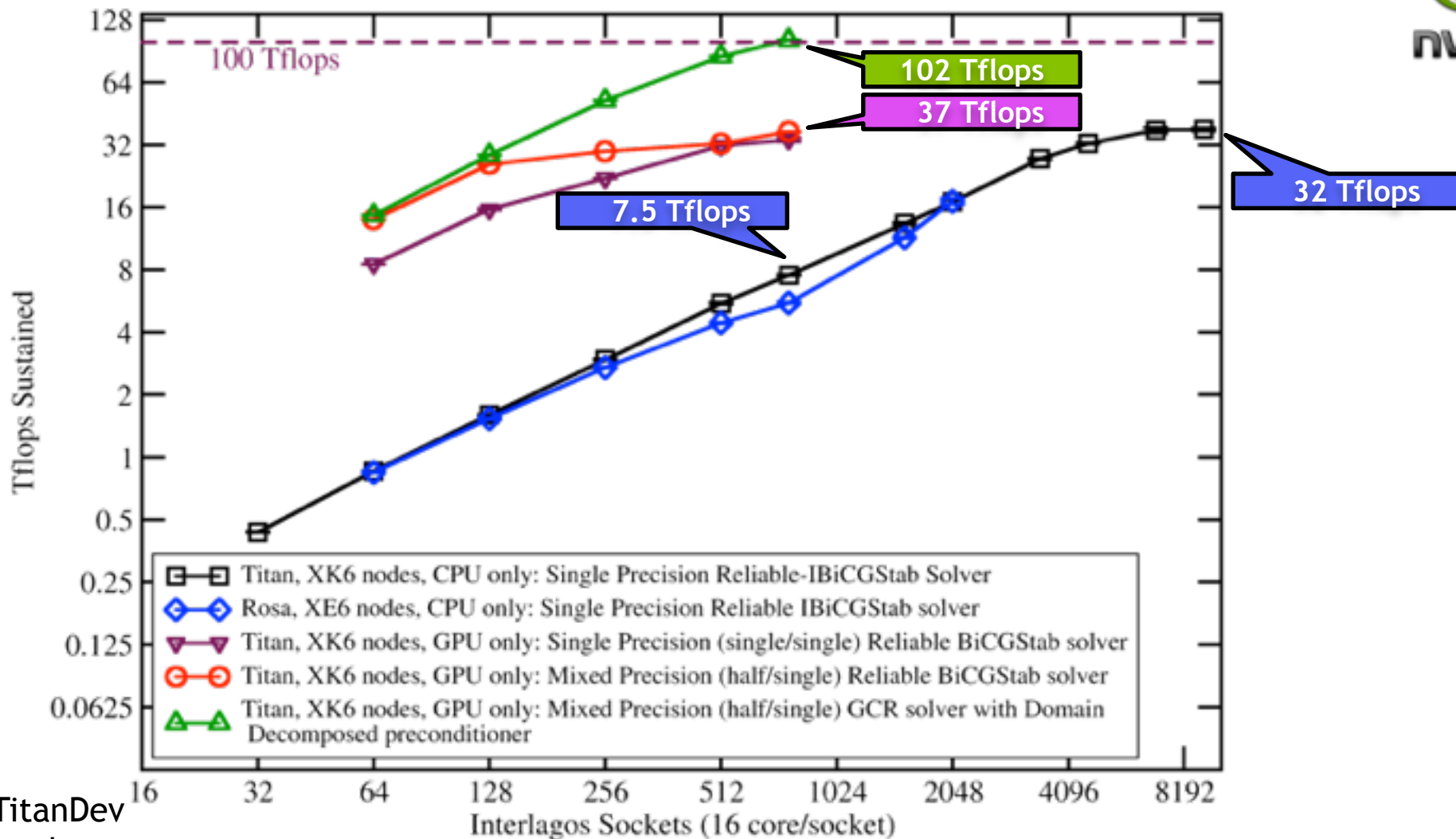


Strong Scaling: $48^3 \times 512$ Lattice (Weak Field), Chroma + QUDA



Results from TitanDev
 - $48^3 \times 512$ aniso clover
 - scaling up 768 GPUs

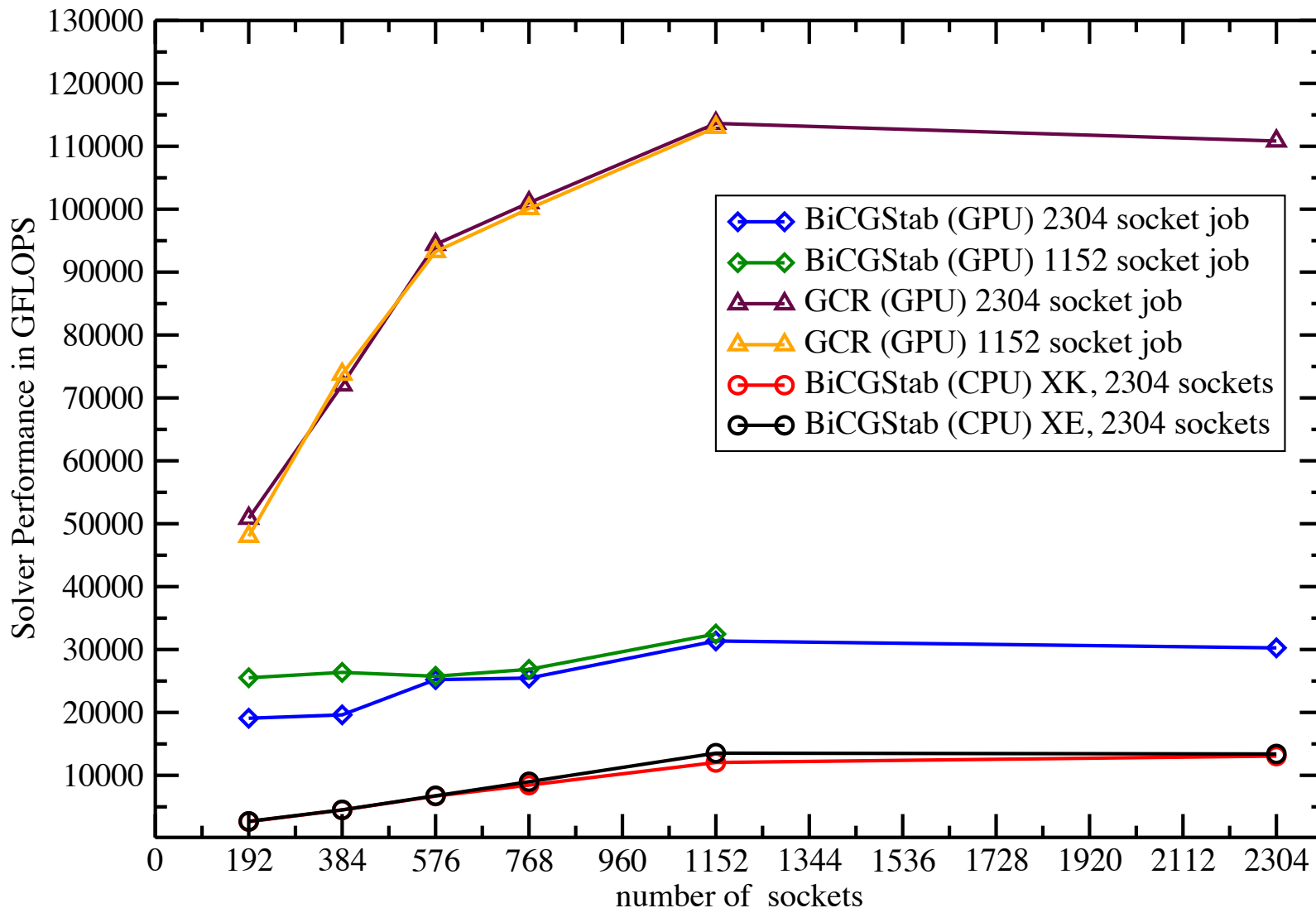
Strong Scaling: $48^3 \times 512$ Lattice (Weak Field), Chroma + QUDA



Results from TitanDev
 - $48^3 \times 512$ aniso clover
 - scaling up 768 GPUs



Blue Waters, $V=48^3 \times 512$, $m_q = -0.0864$, (attempt at physical m_π)

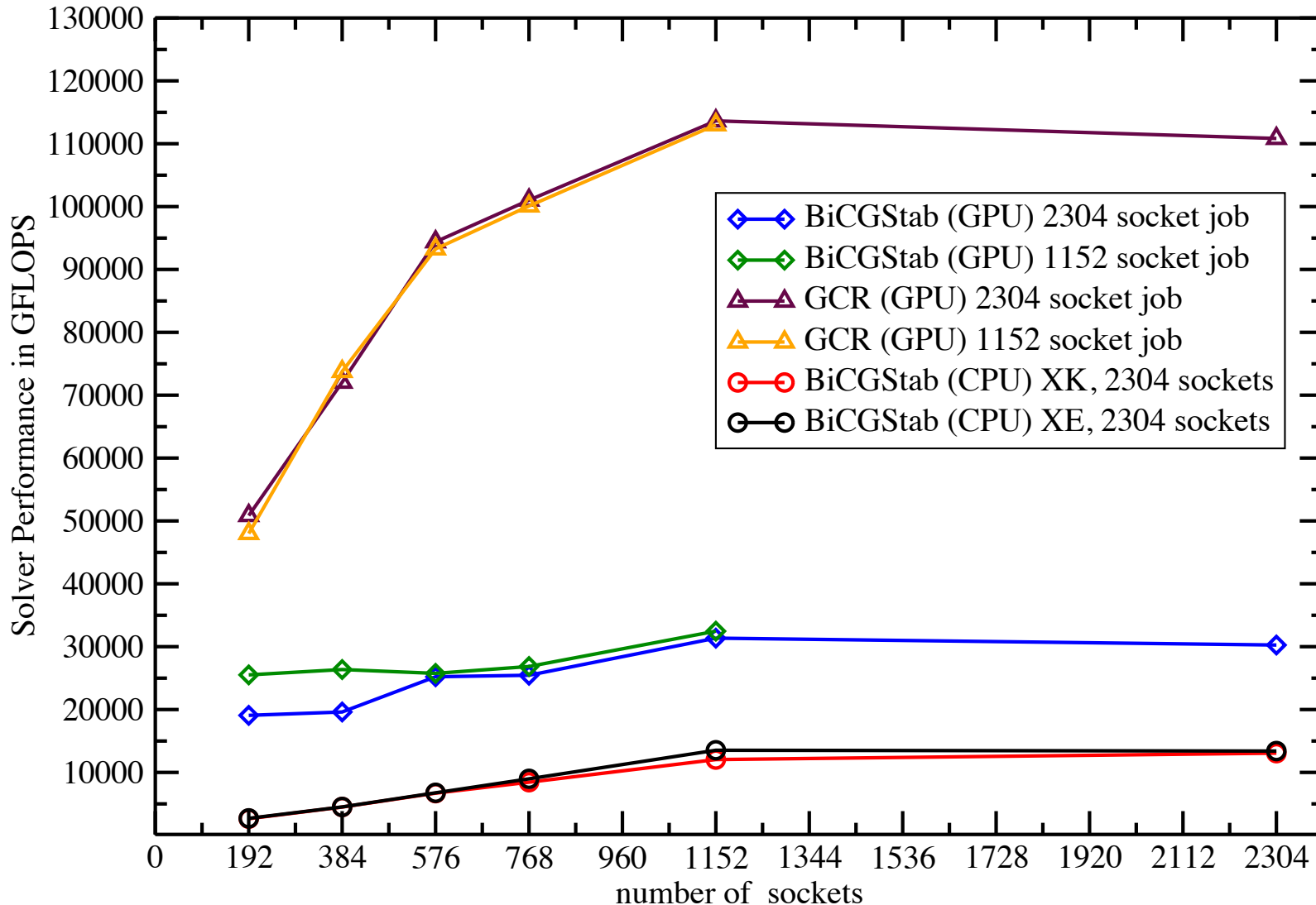


Chroma + QUDA
Clover propagator

Nodes: 192-2304
FLOPS: 19x-7.66x
Solver: 11.5x-4.62x
Whole app: 7.33x-3.35x



Blue Waters, $V=48^3 \times 512$, $m_q = -0.0864$, (attempt at physical m_π)



Chroma + QUDA
Clover propagator

Nodes: 192-2304
FLOPS: 19x-7.66x
Solver: 11.5x-4.62x
Whole app: 7.33x-3.35x

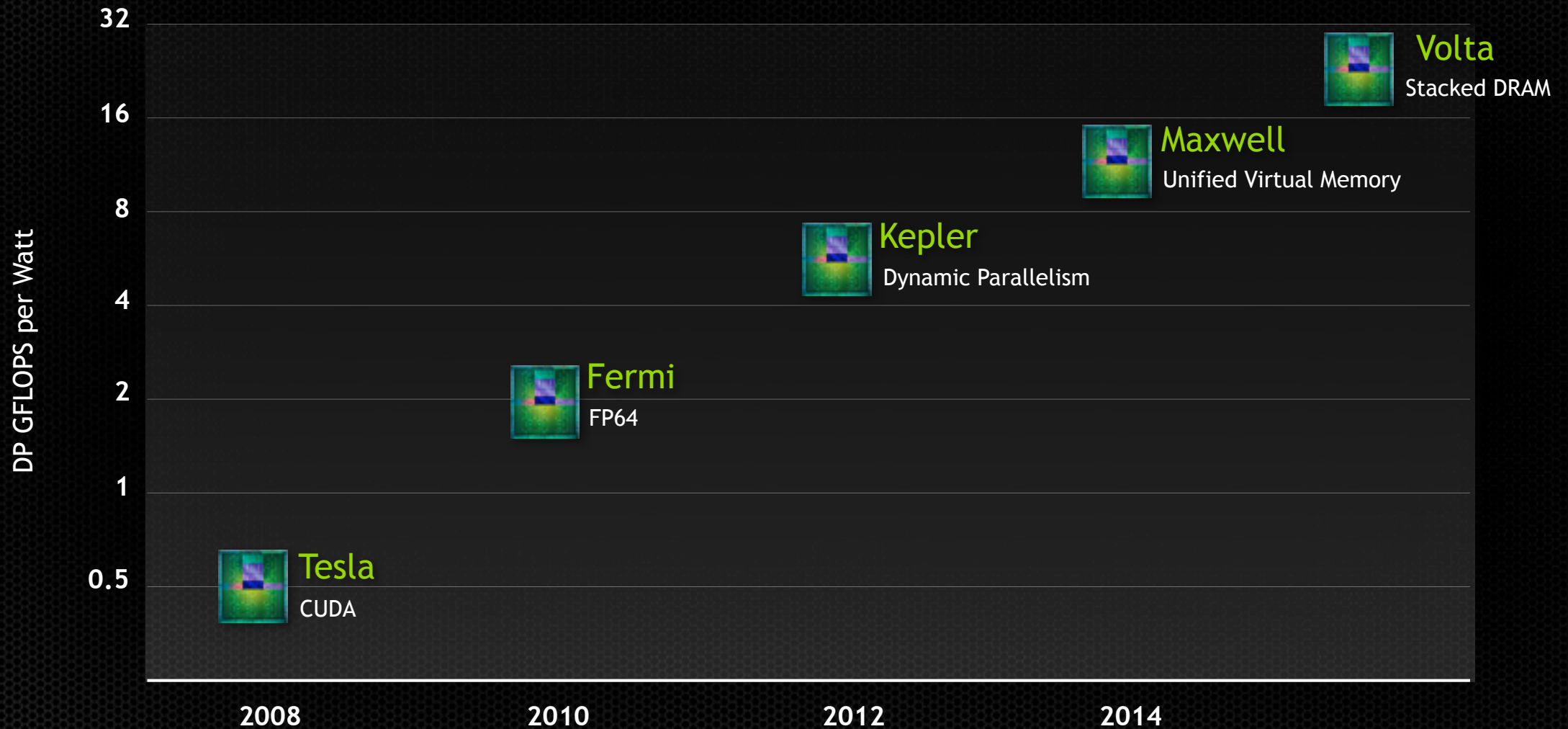
with the addition
of QDP-JIT,
Chroma is ready

An aerial photograph of a city grid, where the streets are highlighted with vibrant, multi-colored lines in shades of blue, green, yellow, orange, red, and purple. The lines are thicker and more prominent than the surrounding streets, creating a futuristic or data-driven aesthetic. The overall scene is set against a dark background, making the glowing lines stand out.

Future Directions

Tuesday, June 18, 13

GPU Roadmap

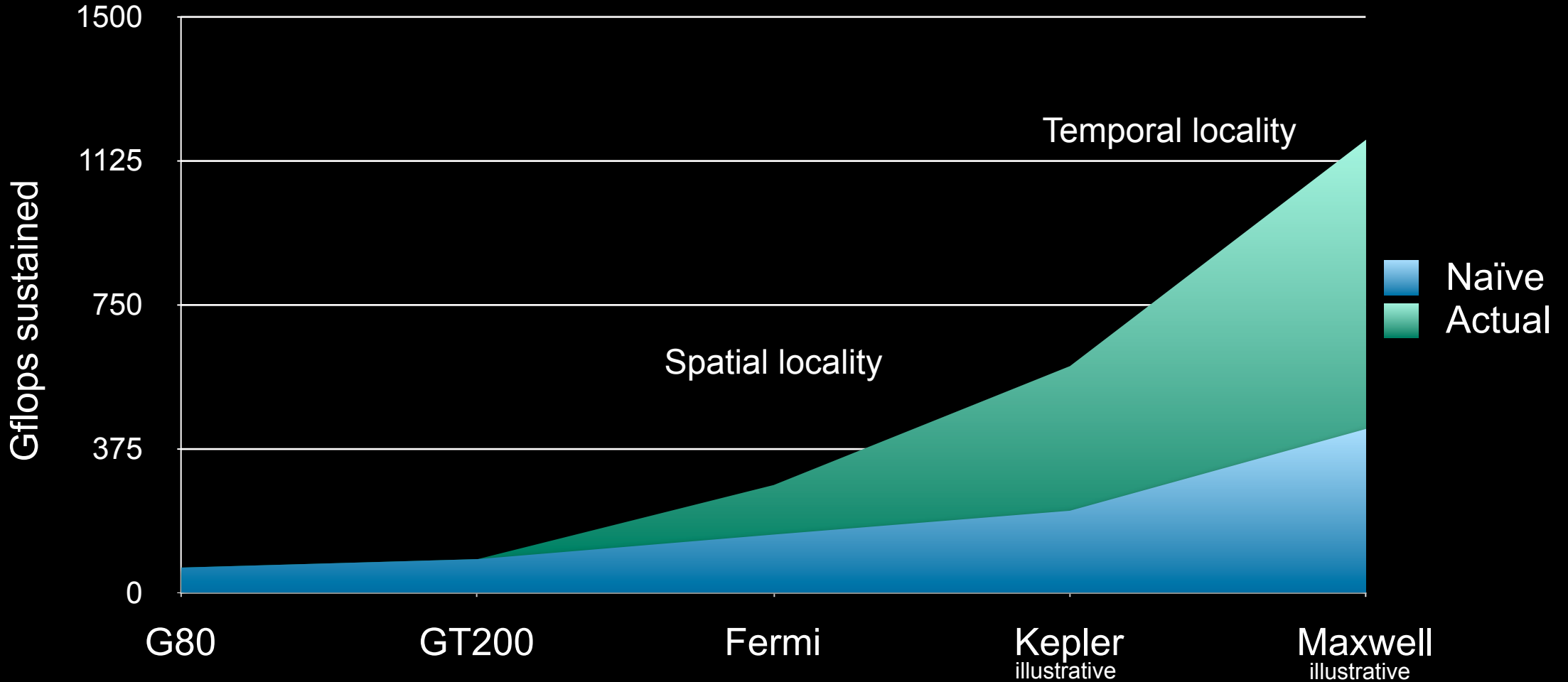


Future Directions

- LQCD coverage (avoiding Amdahl)
 - Remaining force terms needed for gauge generation
 - Contractions
 - Eigenvector solvers
- Solvers
 - Adaptive Multigrid
 - Scalability
- Performance
 - Locality
 - Learning from today's lessons (software and hardware)

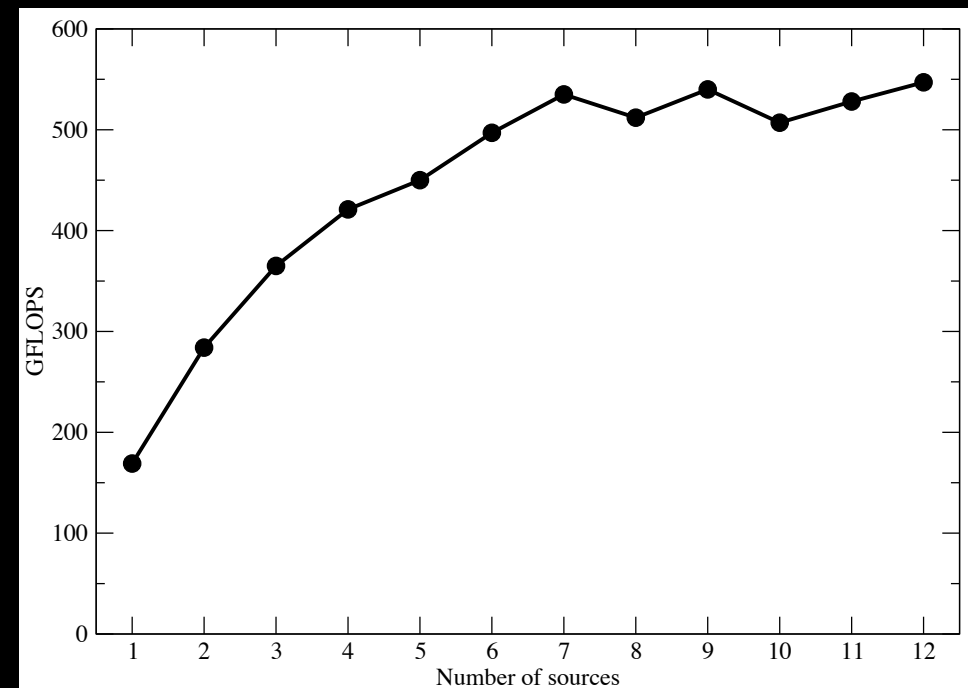
Exploiting Locality

Wilson SP Dslash Performance with GPU generation



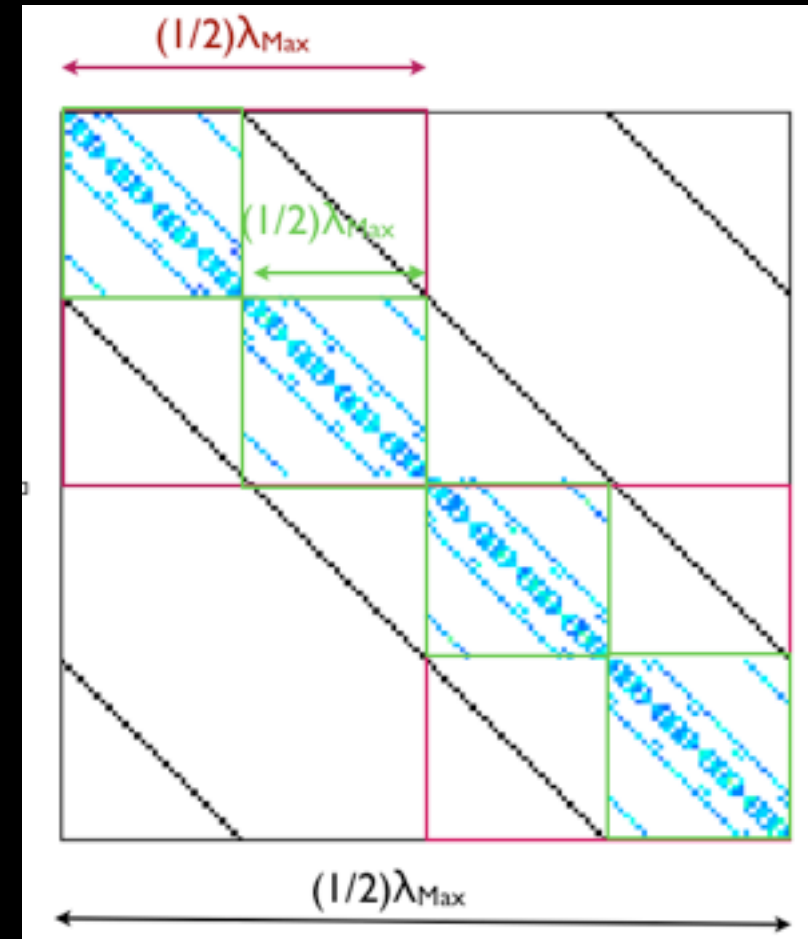
Future Directions - Locality

- 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



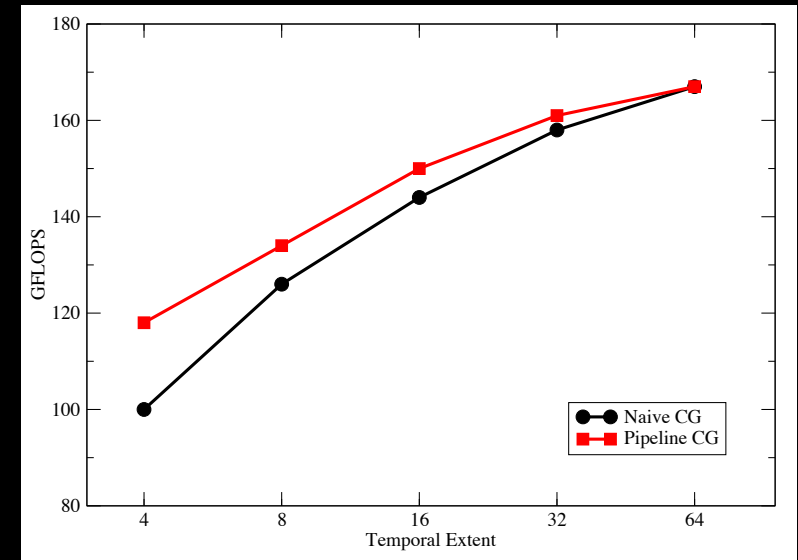
Future Directions - Communication

- 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

- 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
 - Think parallel
 - Algorithmic innovation may be required
 - Lessons today are relevant for Exascale

Where can I learn more?

- You can learn more:
 - CUDA Programming Guide
 - CUDA Zone - tools, training, webinars and more
<http://developer.nvidia.com/cuda>
 - Increasing number of books available





Backup slides

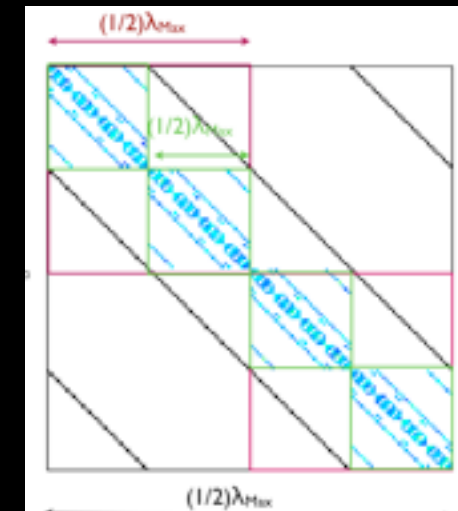
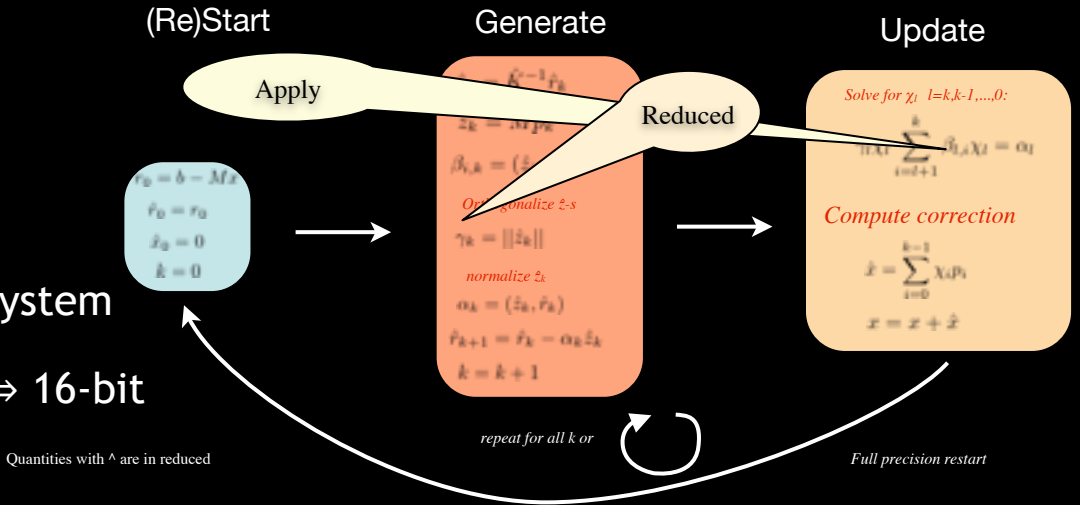
Memory Coalescing

- 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	

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

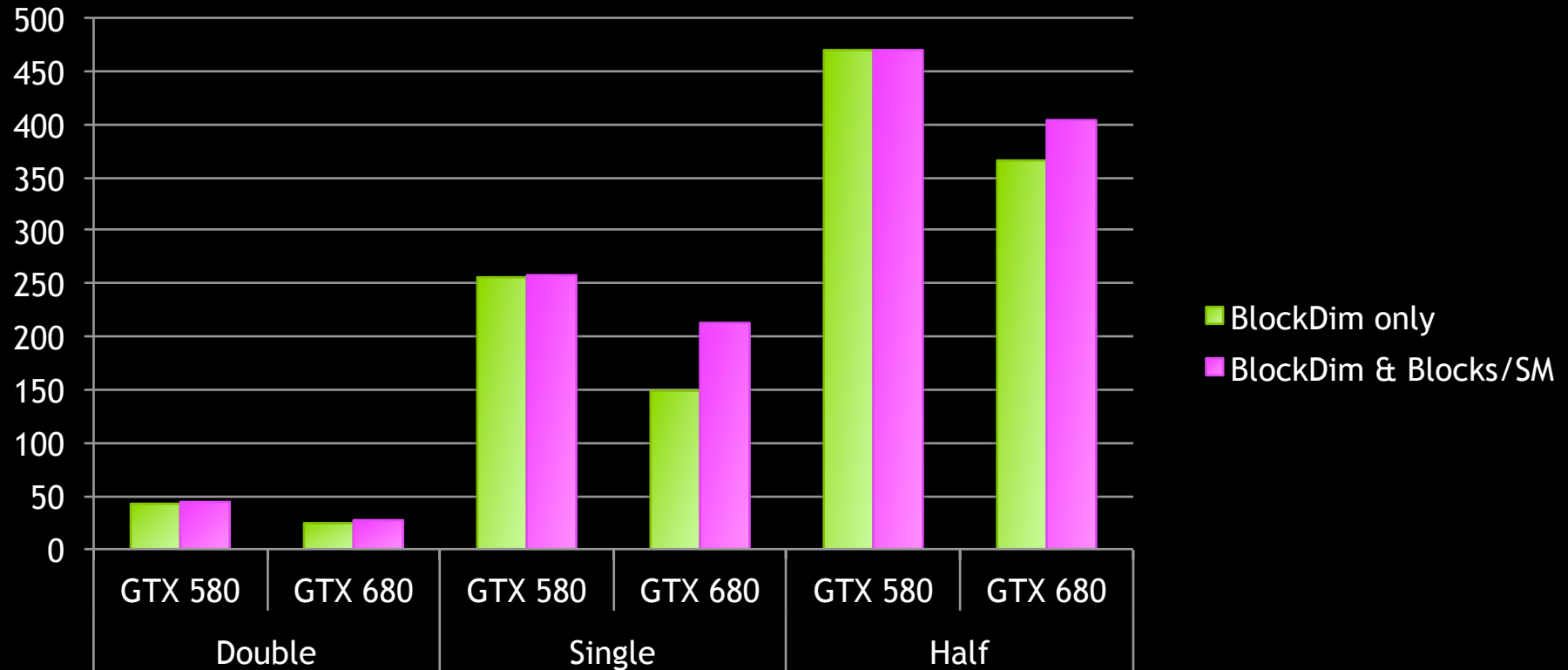


Run-time autotuning

- 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.

OpenACC execution Model



- The OpenACC execution model has three levels:
 - *gang*, *worker* and *vector*
- This is supposed to map to an architecture that is a collection of Processing Elements (PEs)
 - Each PE is multithreaded and each thread can execute vector instructions
- For GPUs one possible mapping could be **gang=block**, **worker=warp**, **vector=threads in a warp**
 - Depends on what the compiler thinks is the best mapping for the problem

Mapping OpenACC to CUDA threads and blocks



```
#pragma acc kernels
  for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```



16 blocks, 256 threads each

```
#pragma acc kernels loop gang(100) vector(128)
  for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```



100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using kernels

```
#pragma acc parallel num_gangs(100) vector_length(128)
{
  #pragma acc loop gang vector
  for( int i = 0; i < n; ++i ) y[i] += a*x[i]; }
```



100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using parallel

Mapping OpenACC to CUDA threads and blocks



```
#pragma acc parallel num_gangs(100)
{
    for( int i = 0; i < n; ++i ) y[i] += a*x[i]; }
```



100 thread blocks, each with apparently 1 thread, each thread redundantly executes the loop

```
#pragma acc parallel num_gangs(100)
{
    #pragma acc loop gang
    for( int i = 0; i < n; ++i ) y[i] += a*x[i]; }
```



compiler can notice that only 'gangs' are being created, so it might decide to create threads instead, say 2 thread blocks of 50 threads.

Mapping OpenACC to CUDA threads and blocks



```
#pragma acc kernels loop gang(100) vector(128)
  for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

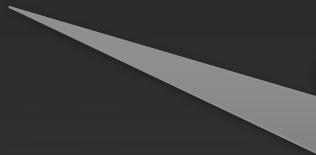


100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using kernels

```
#pragma acc kernels loop gang(50) vector(128)
  for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```



50 thread blocks, each with 128 threads. Each thread does two elements worth of work



Doing multiple iterations per thread can improve performance by amortizing the cost of setup

Mapping multi dimensional blocks and grids to OpenACC

- A nested for loop generates multidimensional blocks and grids

```
#pragma acc kernels loop gang(100), vector(16)
    for( ... )
#pragma acc loop gang(200), vector(32)
    for( ... )
```

100 blocks tall
(row/Y direction)

16 thread tall
block

200 blocks wide
(column/X direction)

32 thread wide
block