QUDA: QCD on GPUs

Mike Clark, NVIDIA
Developer Technology Group

Overview

- QUDA Overview
- Single-GPU Wilson solver
- Multi-GPU strategy and performance
- Getting into QUDA

QUDA overview

- "QCD on CUDA" – http://lattice.github.com/quda
- Effort started at Boston University in 2008, now in wide use as the GPU backend for Chroma, MILC, and various home-grown codes.
- Provides:
  - Various solvers for several discretizations, including multi-GPU support and domain-decomposed (Schwarz) preconditioners.
  - Additional performance-critical routines needed for gauge field generation.
- Contributors welcome!

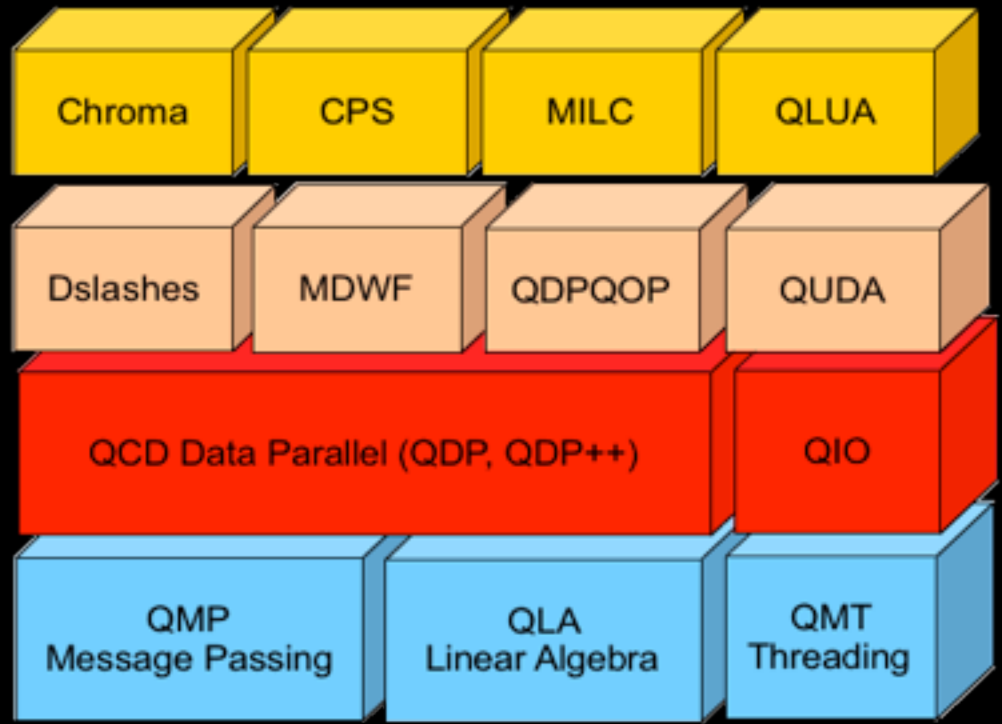
QUDA overview

- Implements most discretized Dirac operators
  - Wilson
  - Wilson-Clover
  - Twisted mass
  - Improve staggered (ASQTAD and HISQ)
  - Domain Wall

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)

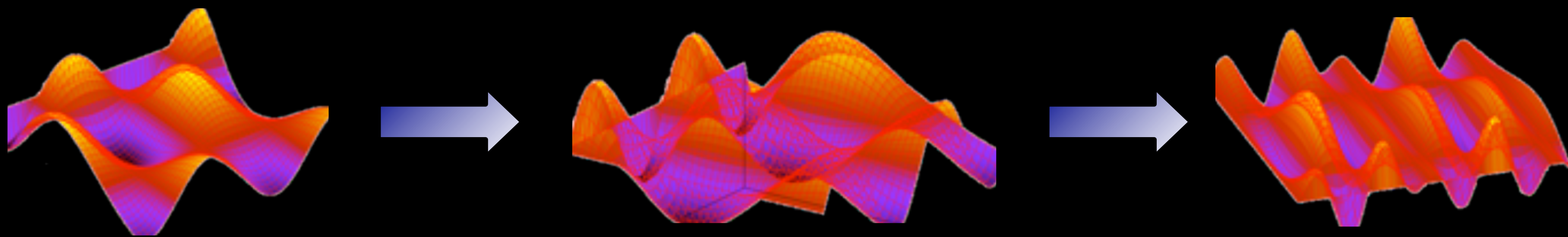
USQCD software stack



(Many components developed under the DOE SciDAC program)

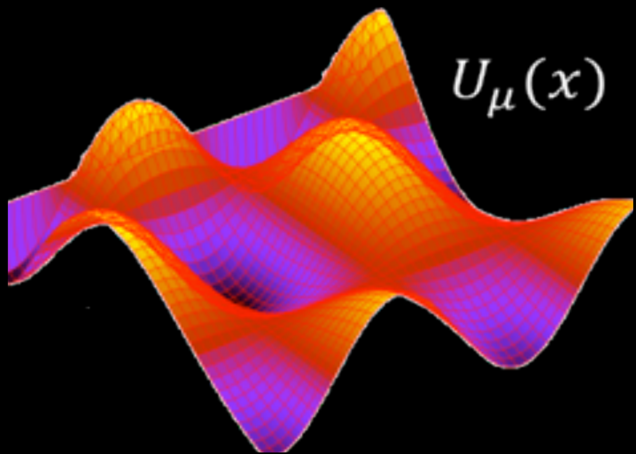
Steps in a lattice QCD calculation

1. Generate an ensemble of gluon field ("gauge") configurations.
   - Produced in sequence, with hundreds needed per ensemble. This requires > O(10 Tflops) sustained for several months (traditionally Crays, Blue Genes, etc.)
   - 50-90% of the runtime is in the solver.



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 solver.
     GPUs have gained a lot of traction here.



$U_\mu(x)$

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

or "$Ax = b$"

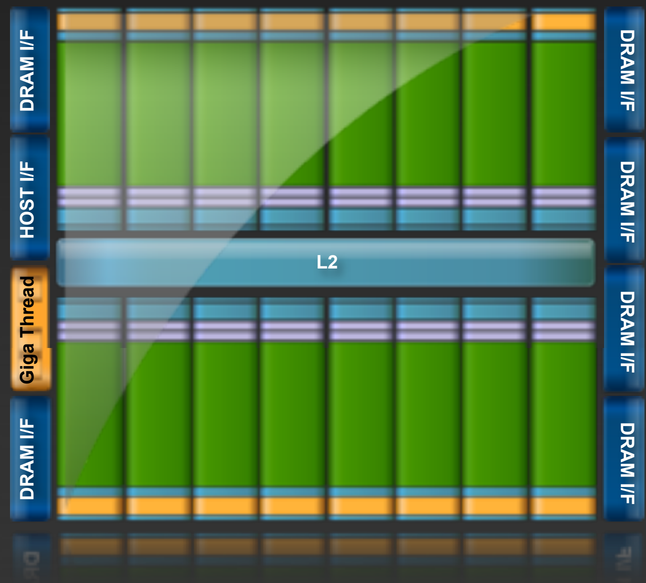
Krylov solvers

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

Thursday, August 23, 12

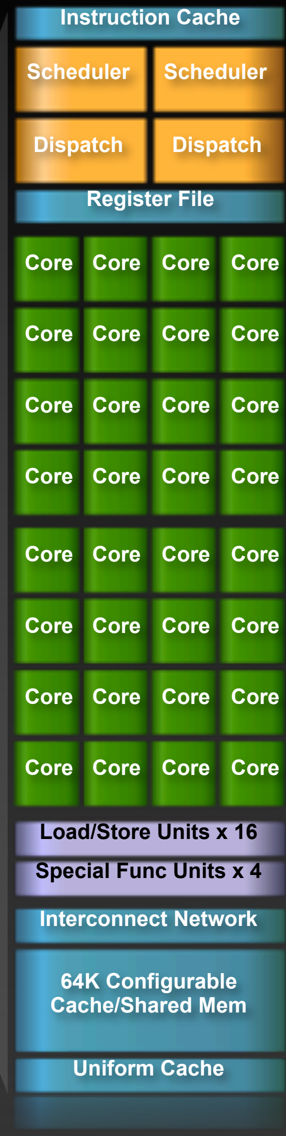
GPU Architecture: Two Main Components

- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB
  - Bandwidth currently up to 177 GB/s for Quadro and Tesla products
  - ECC on/off option for Quadro and Tesla products
- Streaming Multiprocessors (SMs)
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



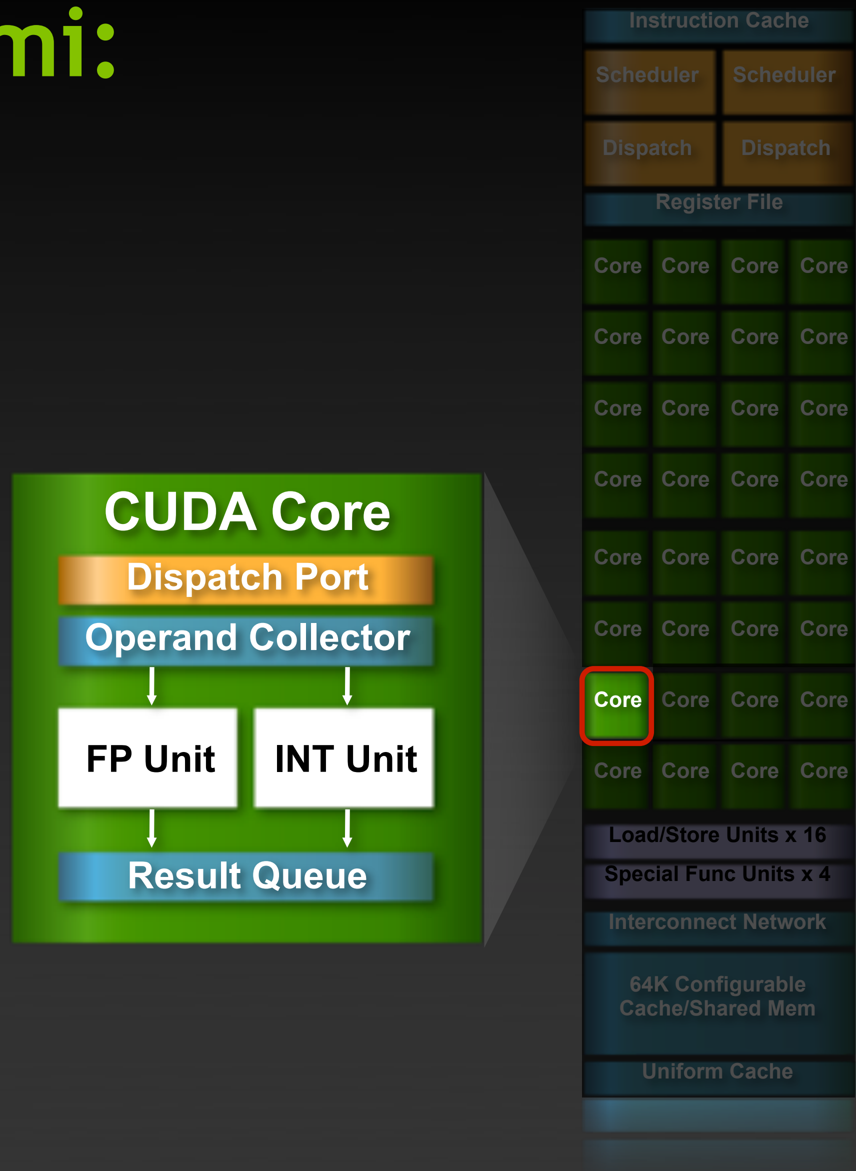
GPU Architecture - Fermi: Streaming Multiprocessor (SM)

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers
- 63 registers-per-thread limit
  - Exceeding this will cause variables to spill into gmem



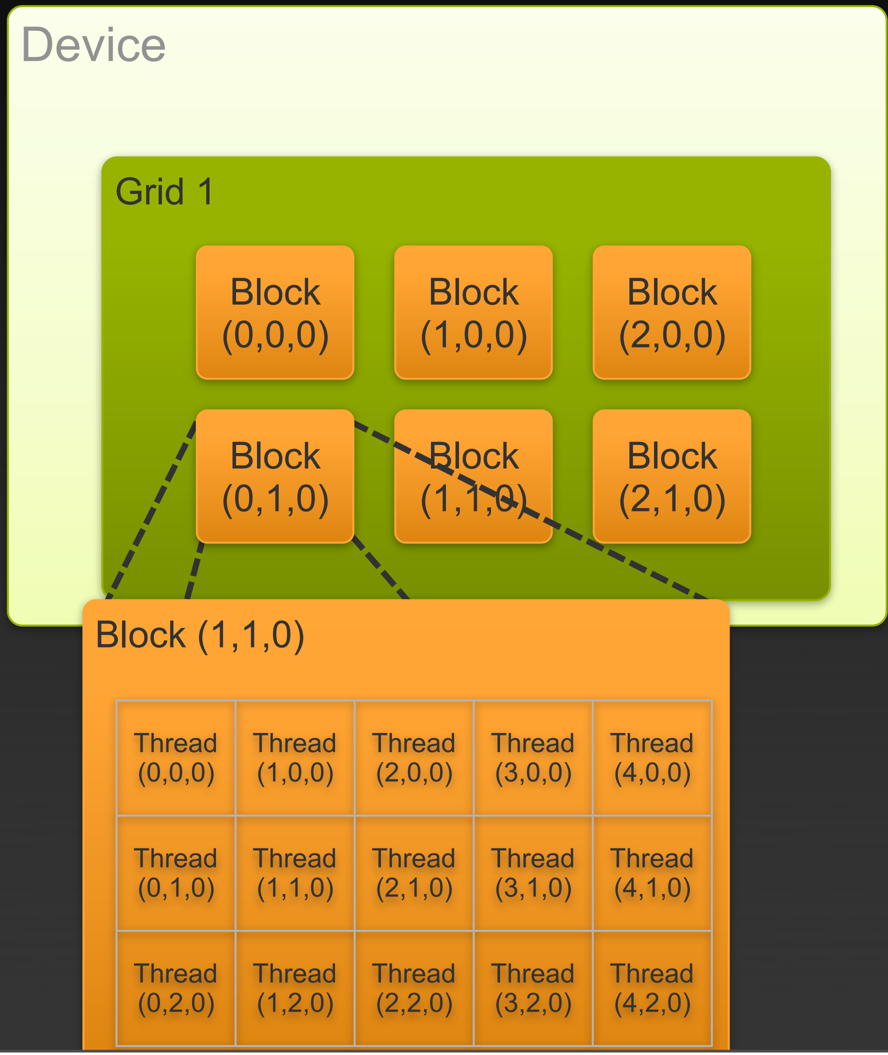
GPU Architecture - Fermi: CUDA Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit



GPU Kernels

- A parallel function that runs on the GPU is called a kernel
- A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
- Built-in variables used to identify threads:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`



3

CUDA C

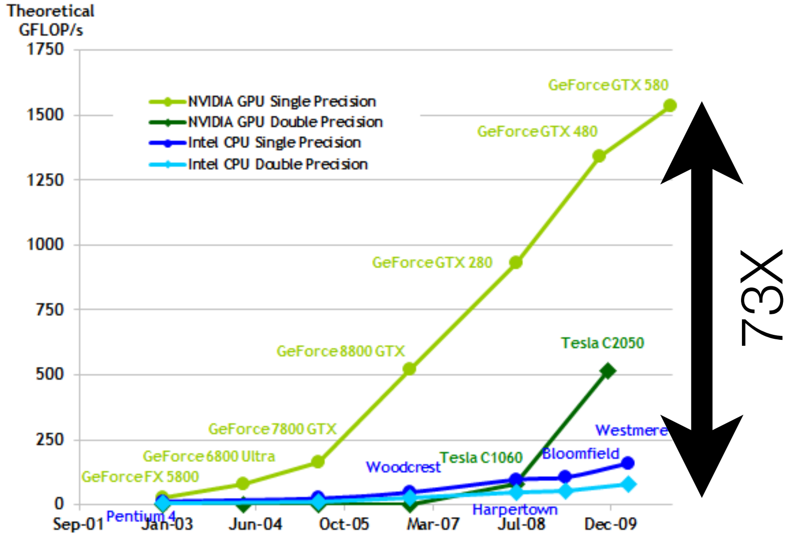
Standard C

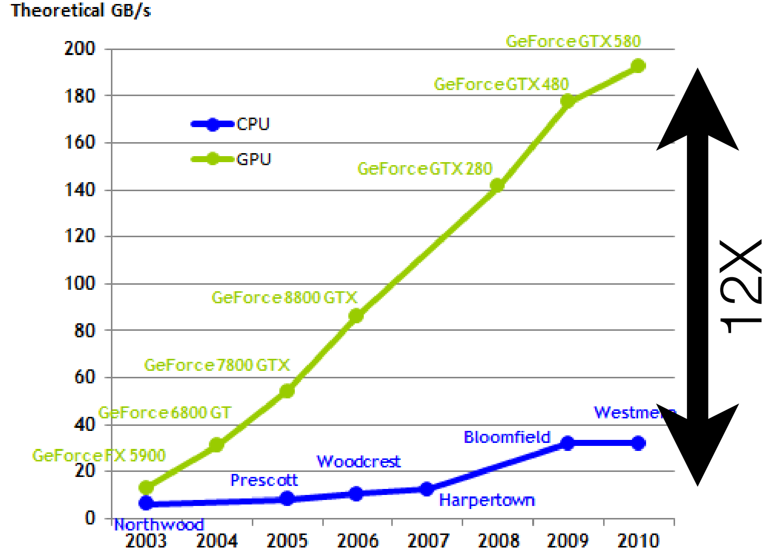
```c
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

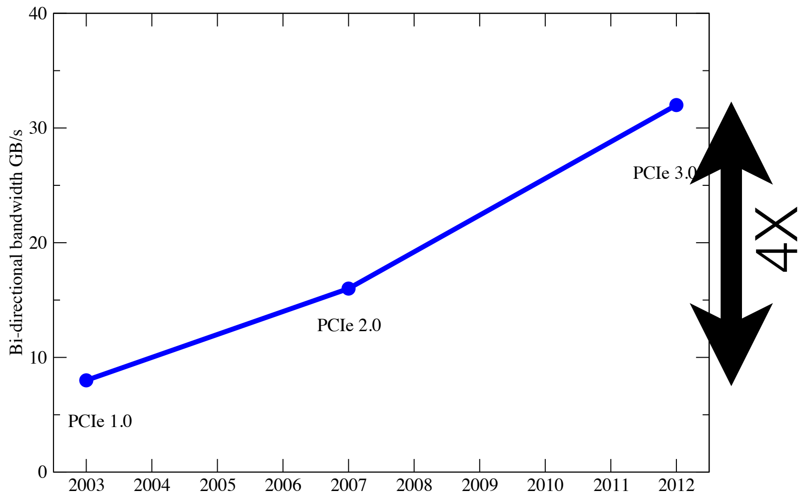
Parallel C

```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

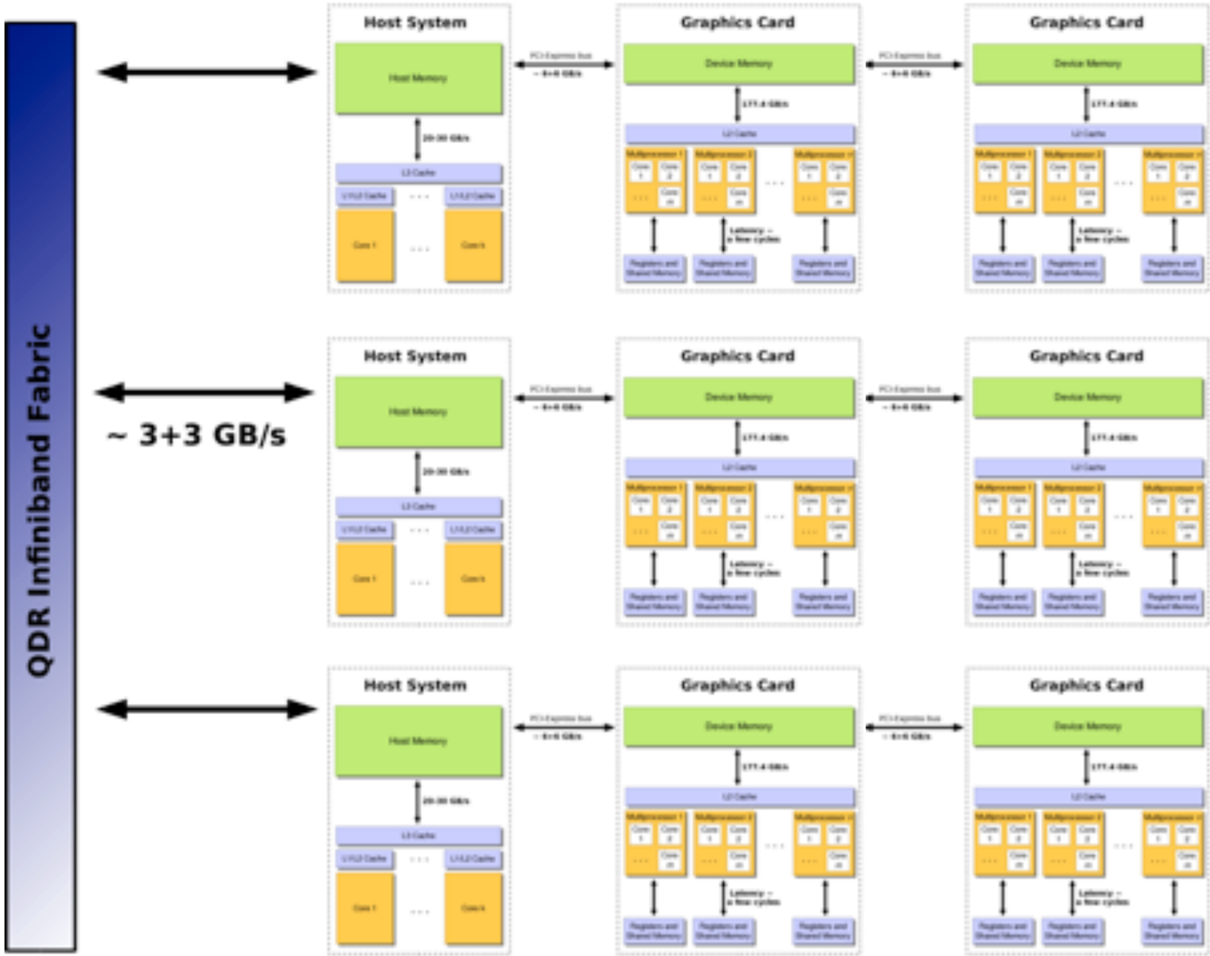






- Disparity worse with every generation
- All architectures have this problem
- Processors get wider
- Memory hierarchy gets deeper

Memory Hierarchy



Single GPU Wilson Solver

Krylov Solver Implementation

- Complete solver **must** be on GPU
  - Transfer b to GPU
  - Solve Mx=b
  - Transfer x to CPU
- Time-critical kernel is the mat-vec
  - Applying the Dirac operator to a spinor field
- Also require BLAS level-1 type operations
  - AXPY operations: b += ax - just like yesterday's vector addition
  - NORM operations: c = (b,b)

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

conjugate gradient

QUDA - General Strategy

- Assign a single space-time point to each thread -> V = XYZT threads
  - Map 4-d space-time index to a 1-d thread index

```
int gindex = threadIdx.x + blockIdx.x*blockDim.x
```

  - Reverse mapping obtained from modular arithmetic

```
gindex = (((t*Z+z)*Y+y)*X+x
```

  - $V = 24^4 \Rightarrow 3.3\times10^6$ threads
  - Fine-grained parallelization
- Maximize performance
  - Field reordering
  - Exploit physical symmetries
  - Mixed-precision methods

Wilson Matrix

Dirac spin projector matrices
(4x4 *spin* space)

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

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

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

m quark mass parameter

Nearest neighbor   Local

Wilson Matrix

Dirac spin projector matrices
(4x4 *spin* space)

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

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

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

m quark mass parameter

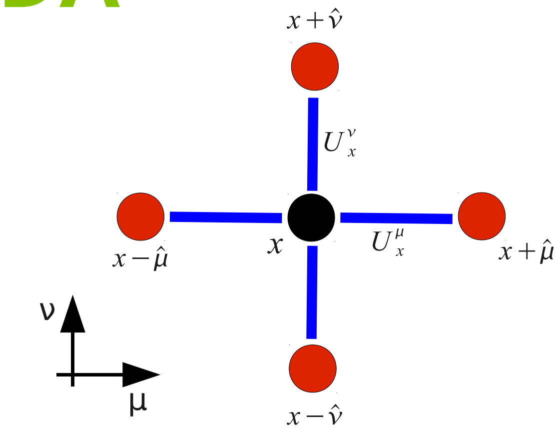
Nearest neighbor   Local

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

Mapping the Wilson Dslash to CUDA

- 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)
- Minimum resources required
  - 12 + 18 + 24 = 54 registers
  - Fermi supports 63x 32-bit registers per thread
- Arithmetic intensity
  - 1320 floating point operations per site
  - 1440 bytes per site (single precision)
  - 0.92 naive arithmetic intensity

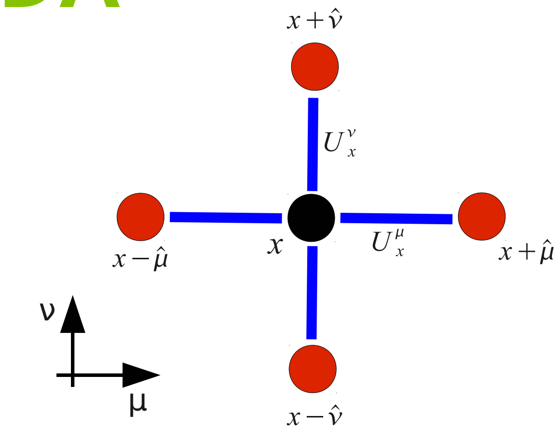
$$D_{x,x'} =$$



Mapping the Wilson Dslash to CUDA

- 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)
- Minimum resources required
  - 12 + 18 + 24 = 54 registers
  - Fermi supports 63x 32-bit registers per thread
- Arithmetic intensity
  - 1320 floating point operations per site
  - 1440 bytes per site (single precision)
  - 0.92 naive arithmetic intensity

$$D_{x,x'} =$$



Tesla M2090	
Gflops	1333
GBytes/s	177
AI	7.5

bandwidth bound

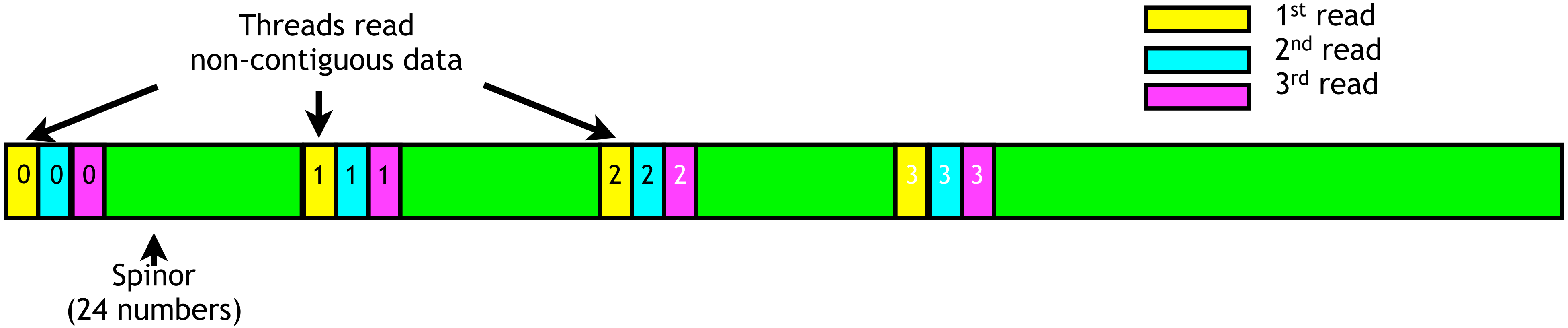
Memory Coalescing

- 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

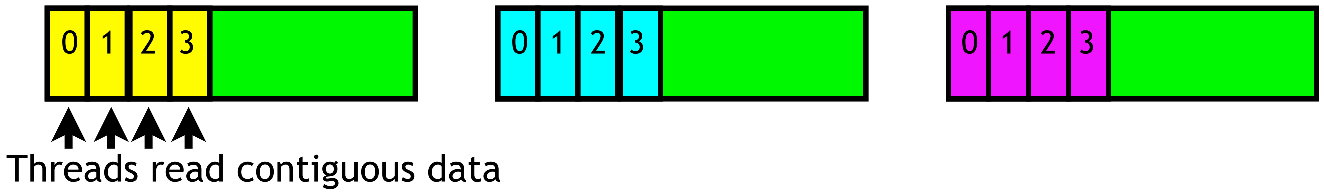
type	32-bit	64-bit	128-bit
int	int	int2	int4
float	float	float2	float4
double		double	double2
char	char4		
short	short2	short4	

Field Ordering

- Typical CPU spinor field ordering: array of spinors (V x 24 floats)



- Reorder fields for coalescing: 6V x float4



- Similar reordering required for color matrices: 3V x float4

- 16-bit uses short4, 64-bit uses double2

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

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

    - Additional 384 flops per site
  - 8 number parameterization

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \longrightarrow \begin{pmatrix} \arg(a_1) & \arg(c_1) & \mathrm{Re}(a_2) & \mathrm{Im}(a_2) \\ \mathrm{Re}(a_3) & \mathrm{Im}(a_3) & \mathrm{Re}(b_1) & \mathrm{Im}(b_1) \end{pmatrix}$$

    - Additional 856 flops per site
- Gauge fix to unit gauge field along T-dimension

Reducing Memory Traffic

- Impose similarity transforms to increase sparsity
    - Globally change Dirac matrix basis

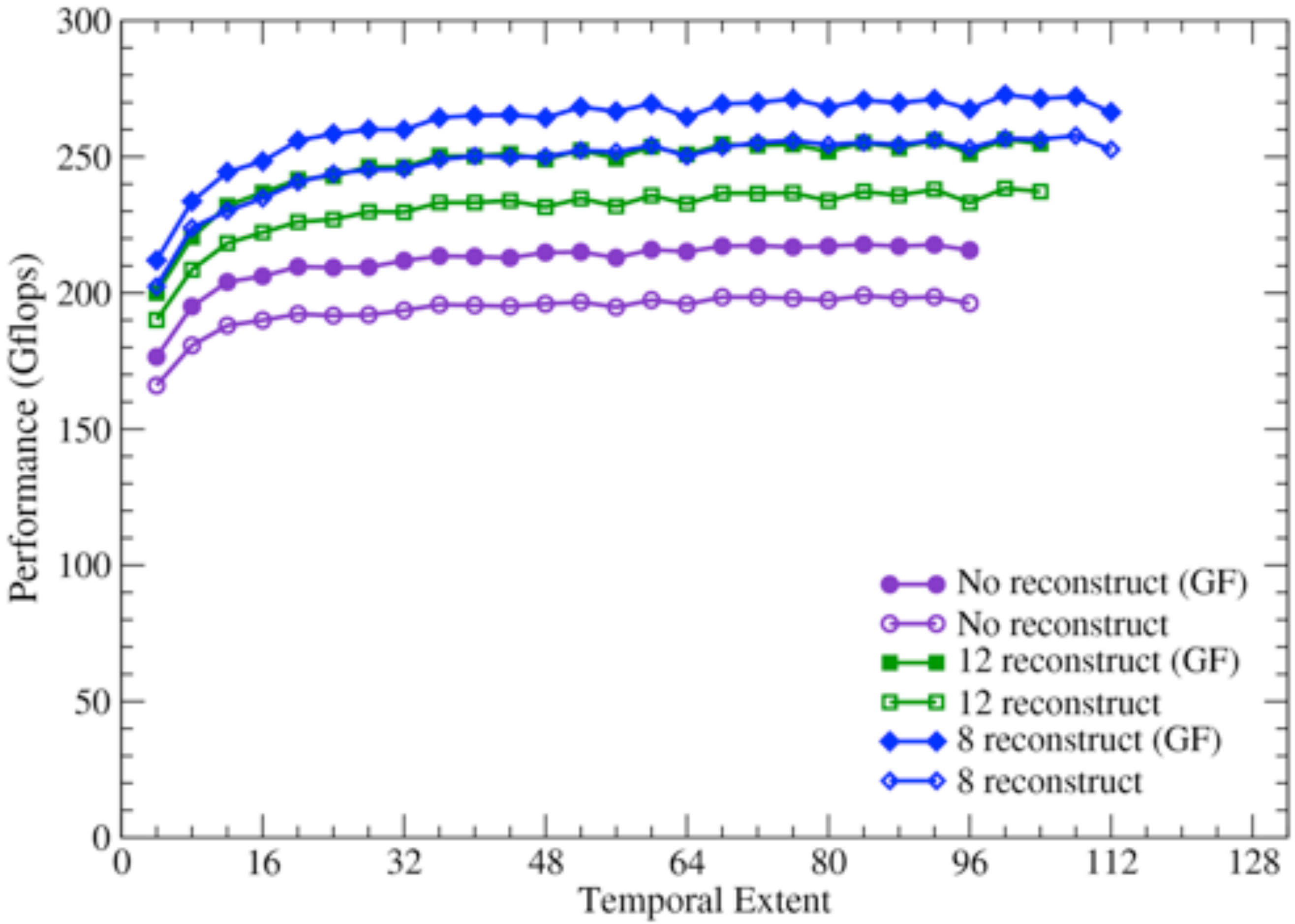
$$P^{\pm 4} = \begin{pmatrix} 1 & 0 & \pm 1 & 0 \\ 0 & 1 & 0 & \pm 1 \\ \pm 1 & 0 & 1 & 0 \\ 0 & \pm 1 & 0 & 1 \end{pmatrix} \longrightarrow P^{+4} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} P^{-4} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

- (Advanced) Still memory bound - Can further reduce memory traffic by truncating the precision
    - Use 16-bit fixed-point representation

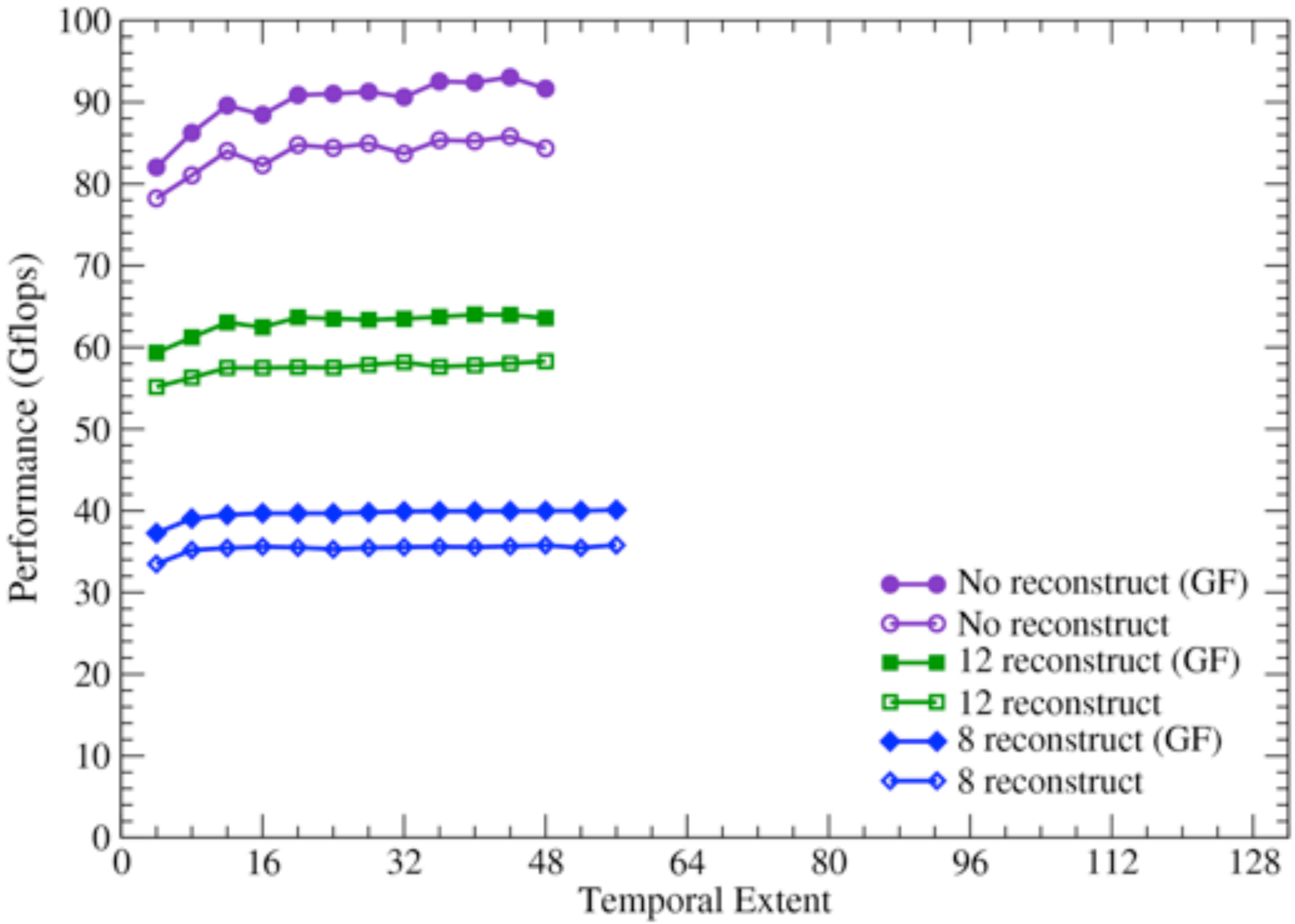
Wilson-Dslash Performance

- For illustration only; not our latest and greatest
- Runs were done on a single Fermi GTX 480 (~M2090)
- Typical single-node performance on Westmere
  - ~25 Gflops for typical optimized production code
  - ~50 Gflops when highly optimized (Smelyanskiy et al)
- Hold spatial lattice dimensions fixed 24^3, vary temporal extent
  - Demonstrates the need for minimum problem size to hide latencies

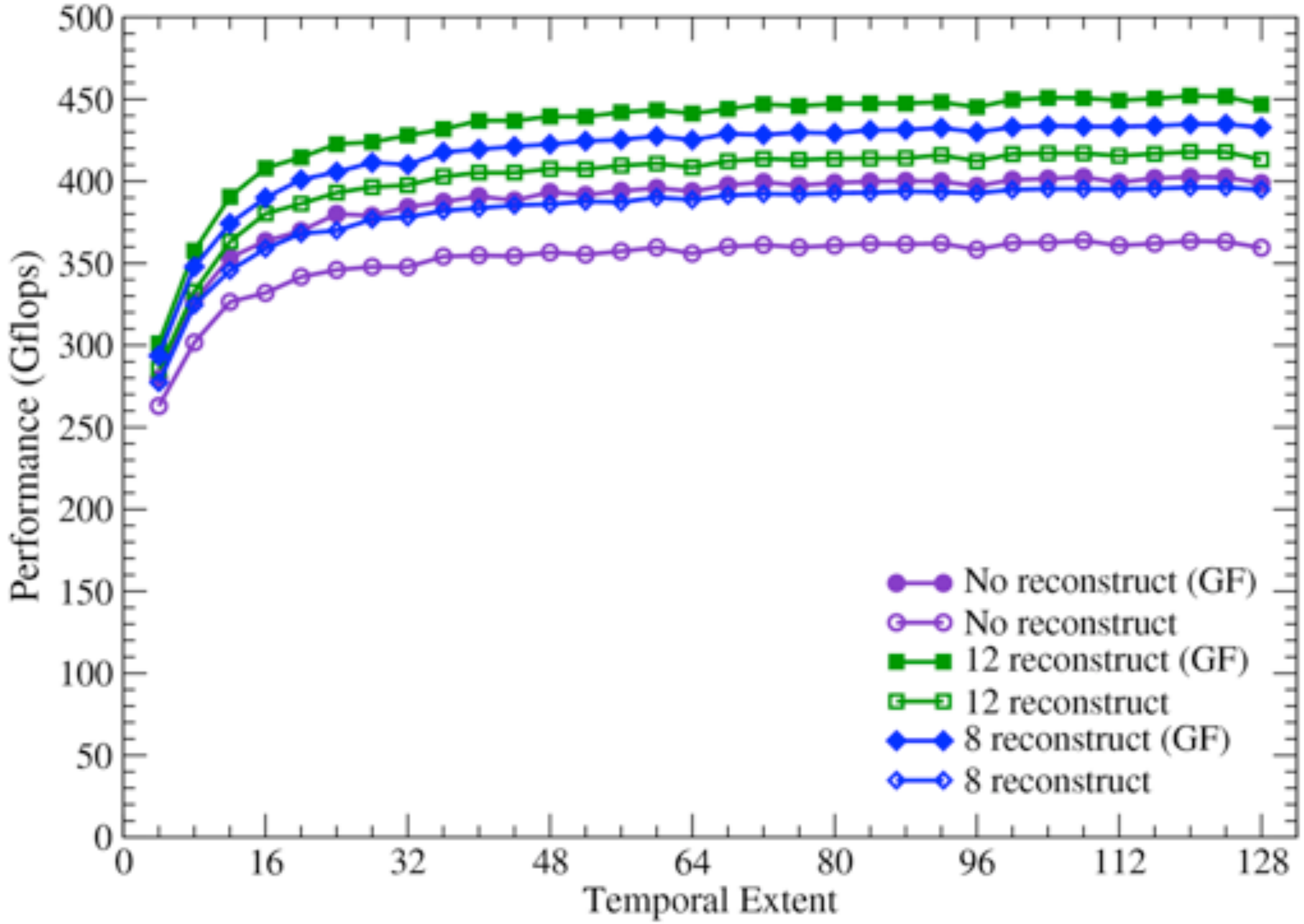
Wilson performance - single precision



Wilson performance - double precision

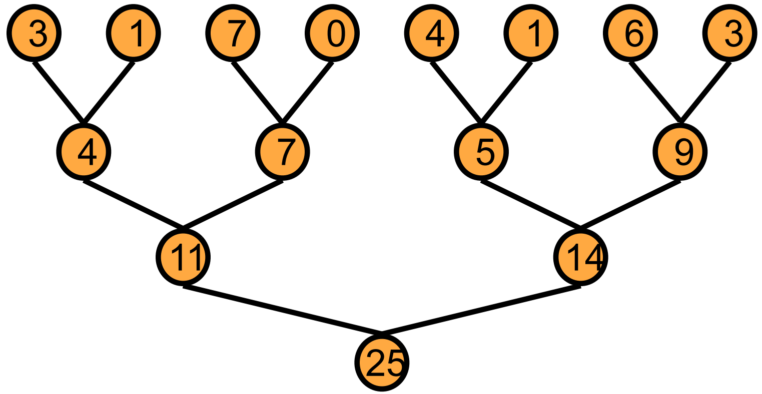


Wilson performance - half precision



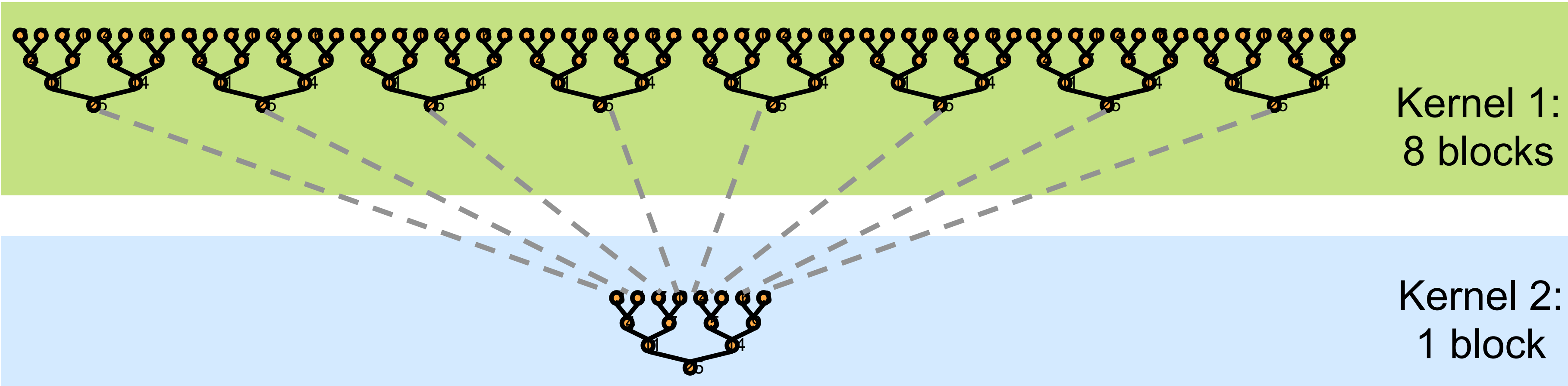
Parallel Reduction

- Common and important data parallel primitive in solvers
- Tree-based approach used within each thread block
  - Use shared memory to communicate within thread blocks

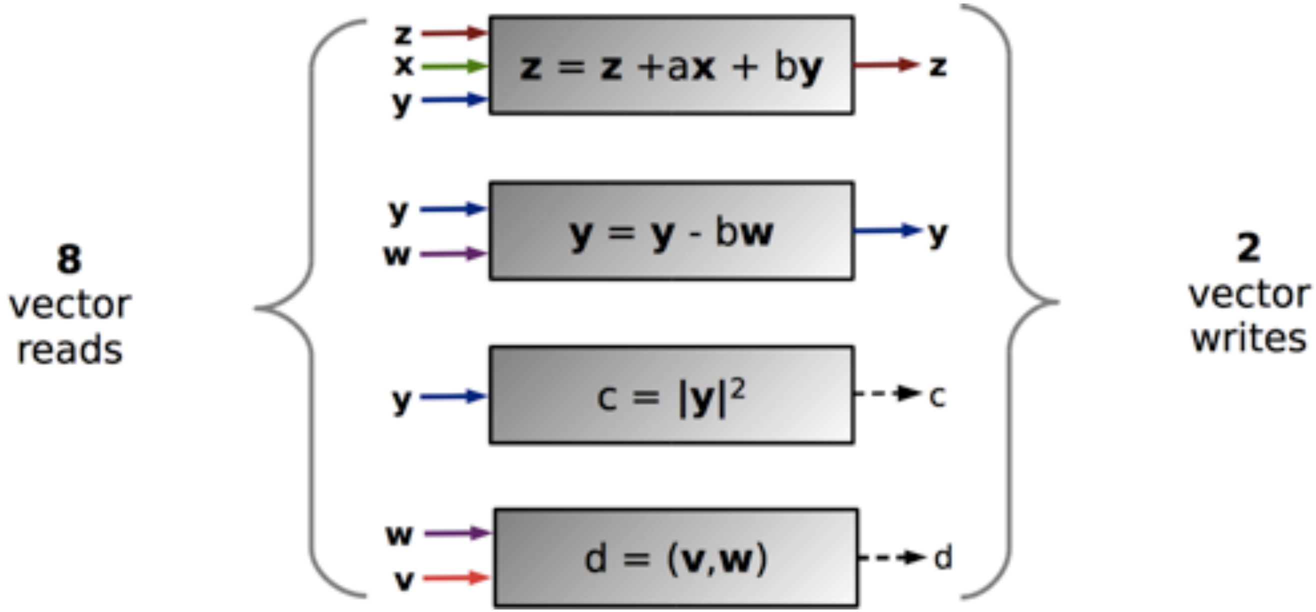


Parallel Reduction

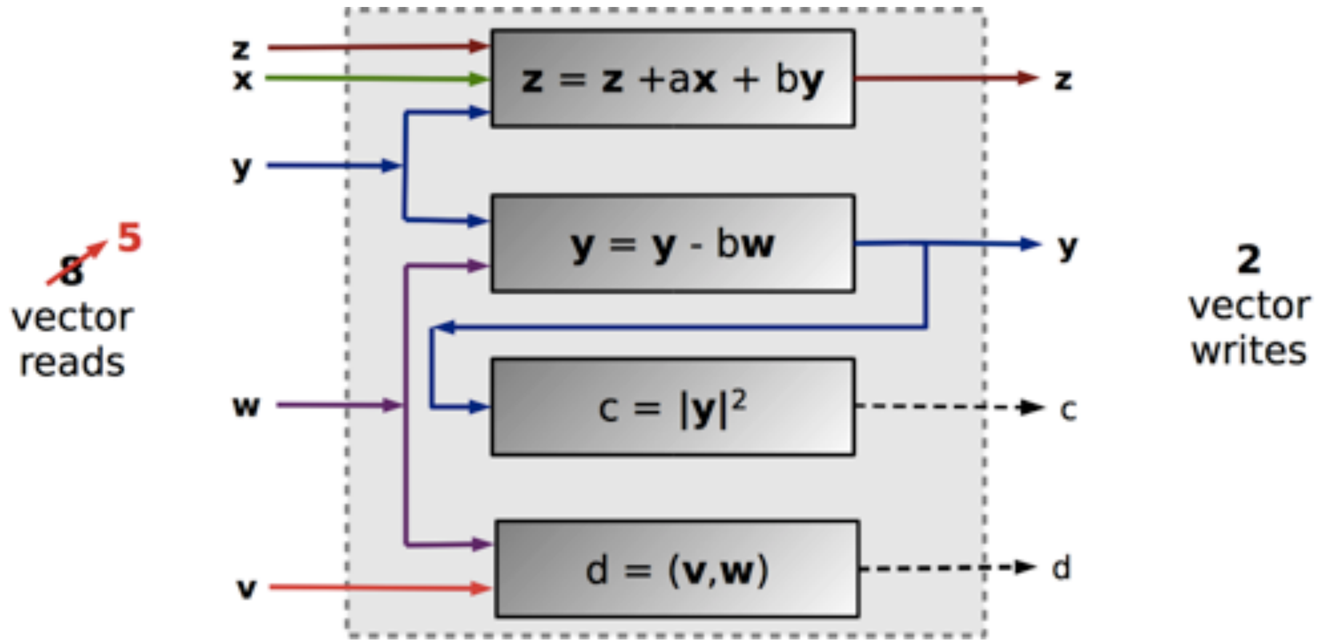
- Avoid global sync by decomposing computation into multiple kernel invocations



Optimizing the Solver: Kernel Fusion



Optimizing the Solver: Kernel Fusion



Mixed-Precision Solvers

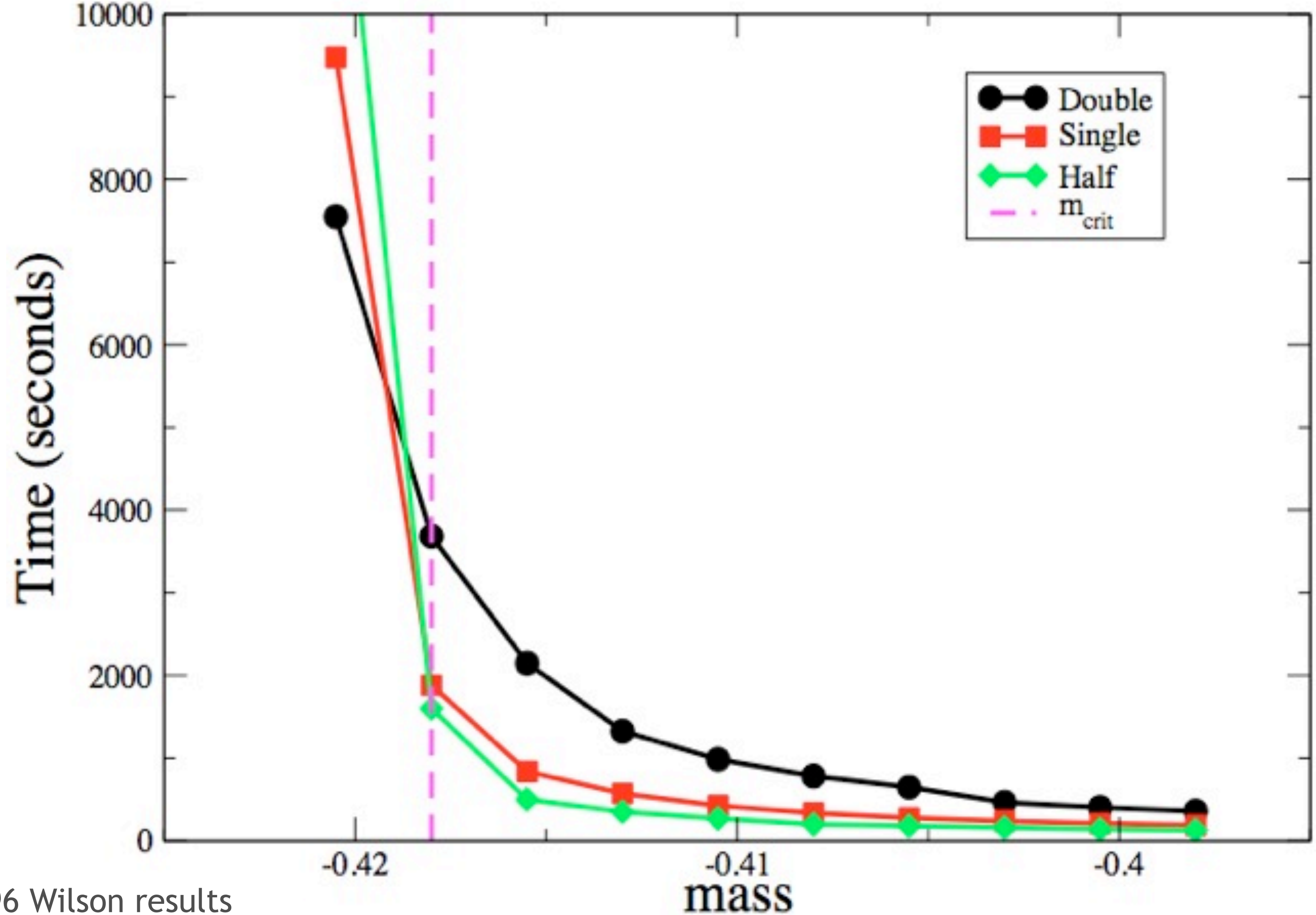
- Often require solver tolerance beyond limit of single precision
- But single and half precision much faster than double
- Use mixed precision
  - e.g. defect-correction

High precision mat-vec and accumulate

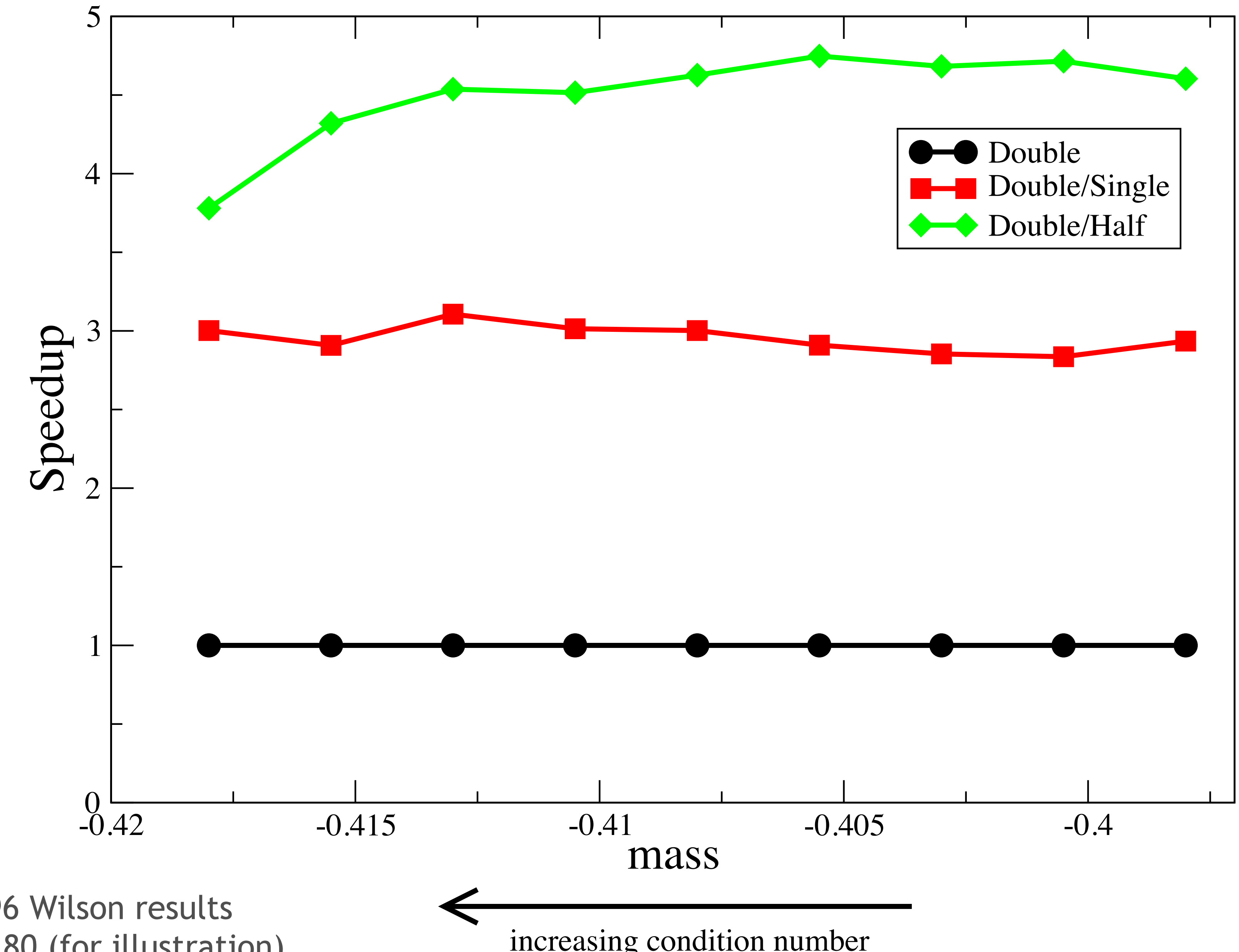
```
while (|rk|> ε) {
    rk = b - Axk
    solve Apk = rk
    xk+1 = xk + pk
}
```

Inner low precision solve

- QUDA uses Reliable Updates (Sleijpen and Van der Worst 1996)
- Almost a free lunch
  - Iteration count increases

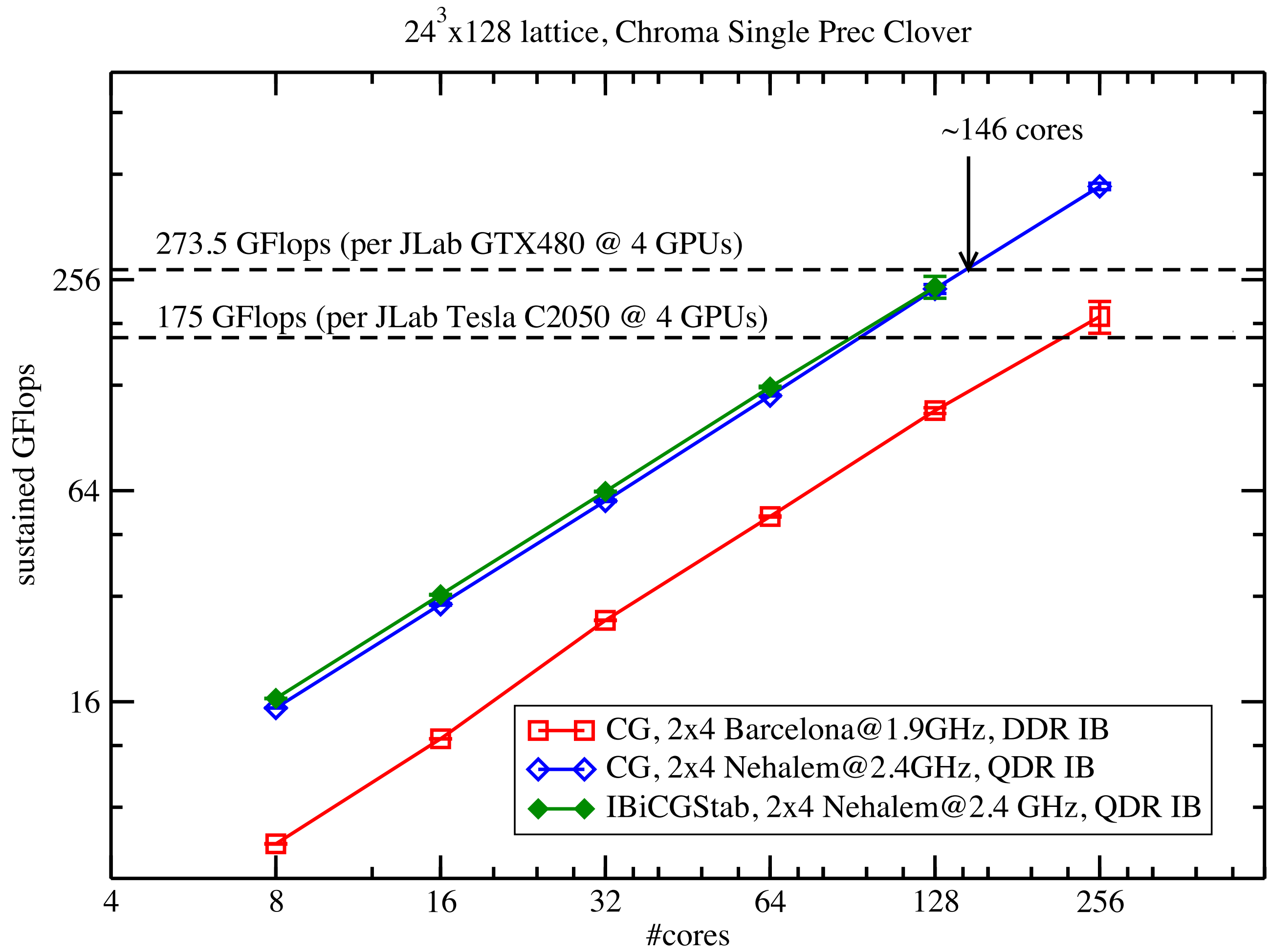


32^3x96 Wilson results
on GTX 280 (for illustration)

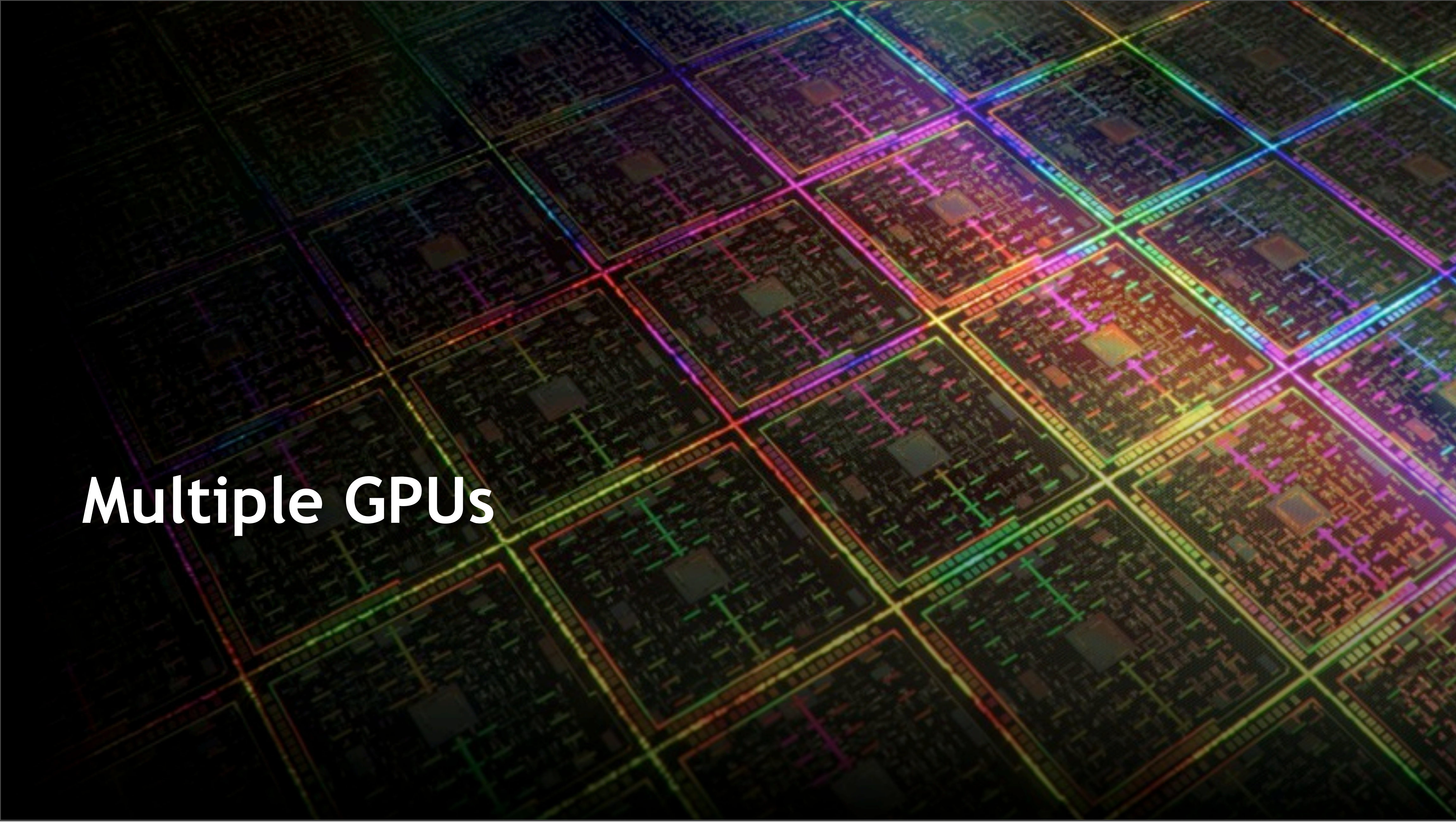


32^3x96 Wilson results
on GTX 280 (for illustration)

GPUs vs. CPUs



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


Multiple GPUs

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
  - Can GPUs replace traditional super computers?

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
  - Can GPUs replace traditional super computers?

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
  - Can GPUs replace traditional super computers?

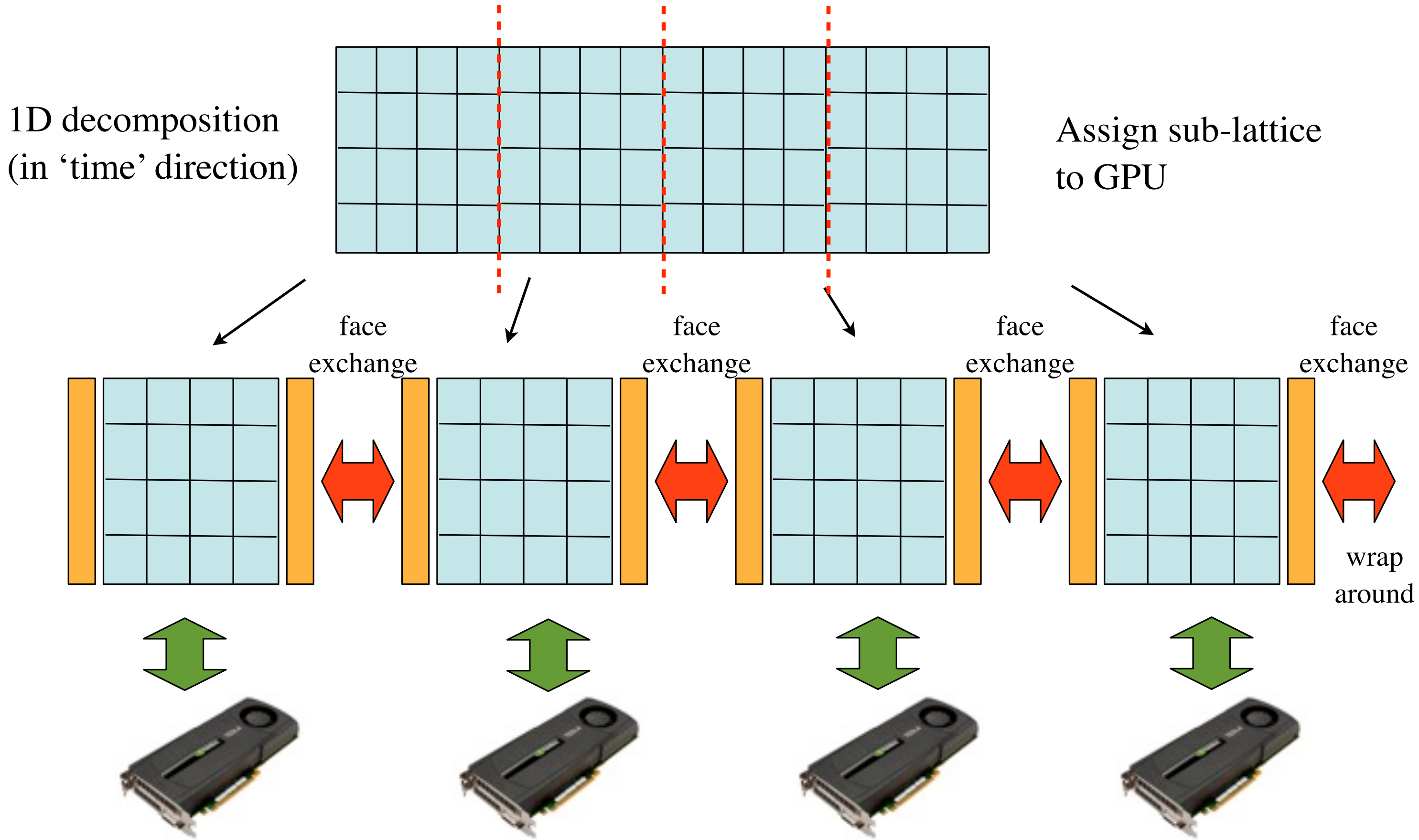
Multiple GPUs

- 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);`

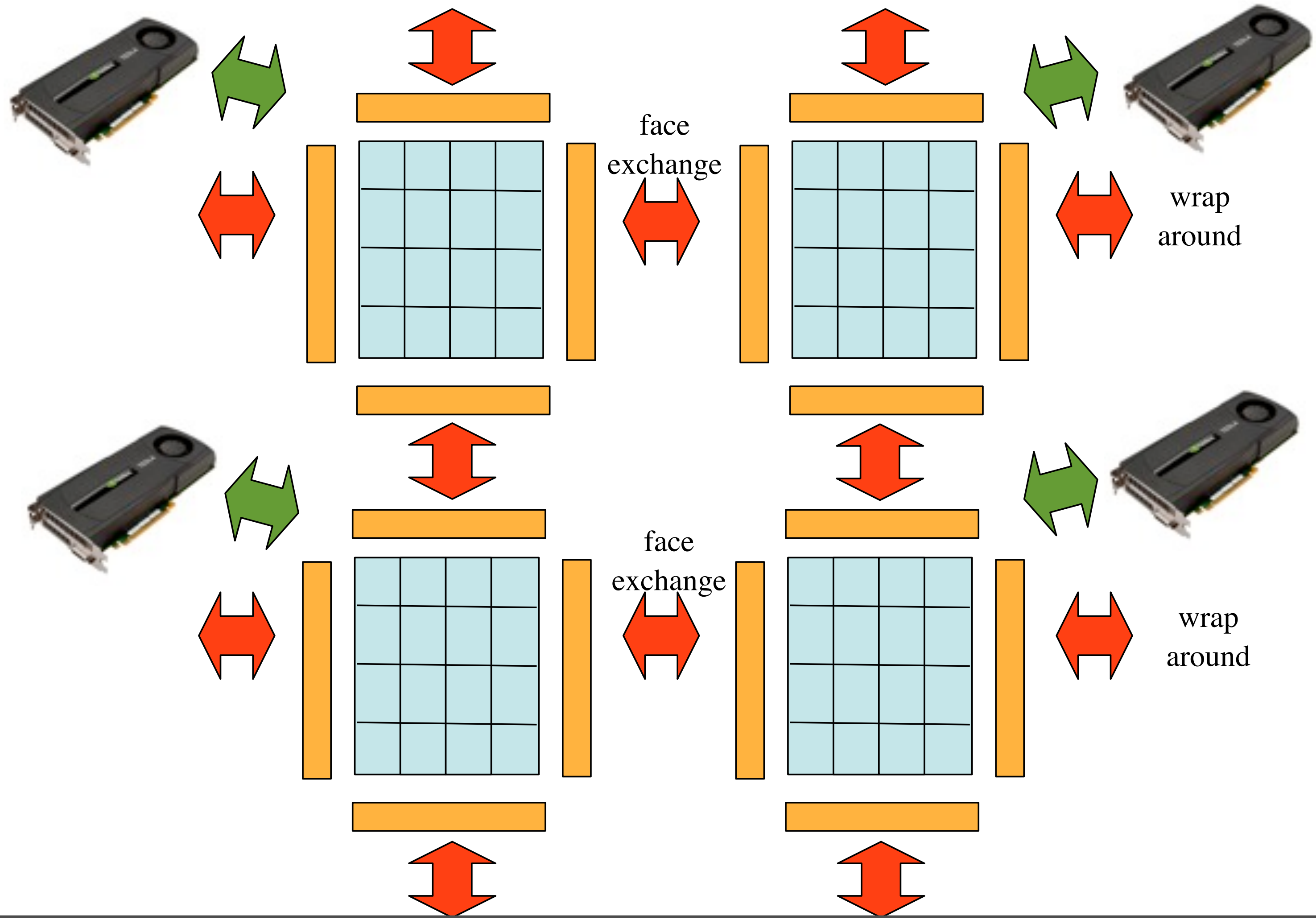
CUDA Stream API

- 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



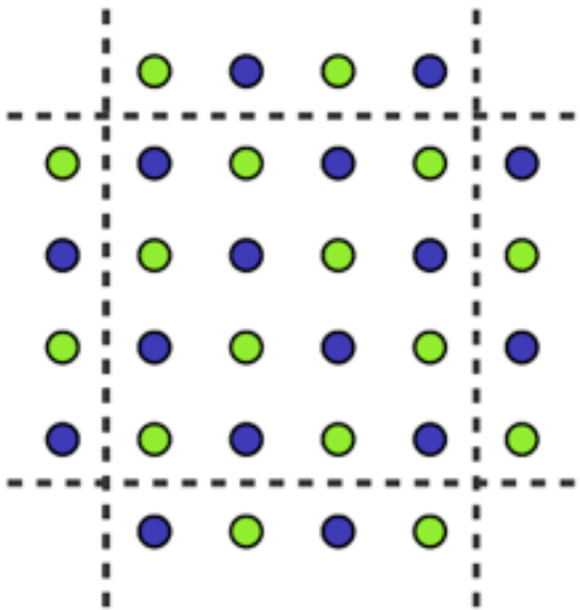
Multi-dimensional lattice decomposition



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

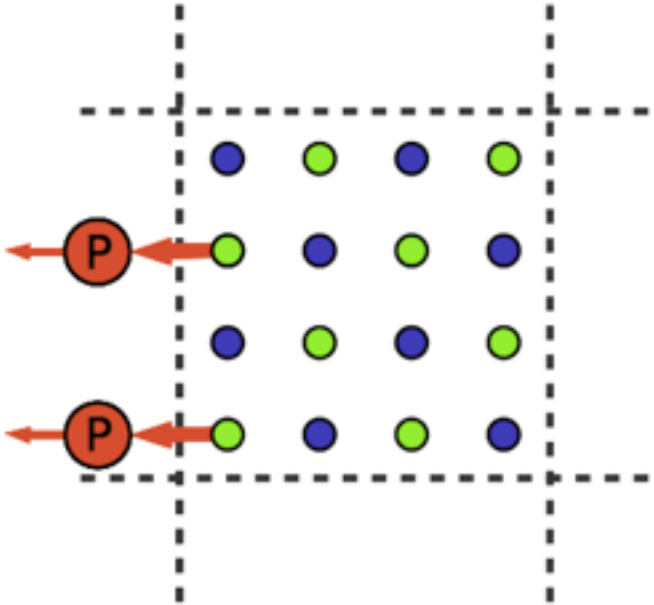
Multi-dimensional Kernel Computation



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

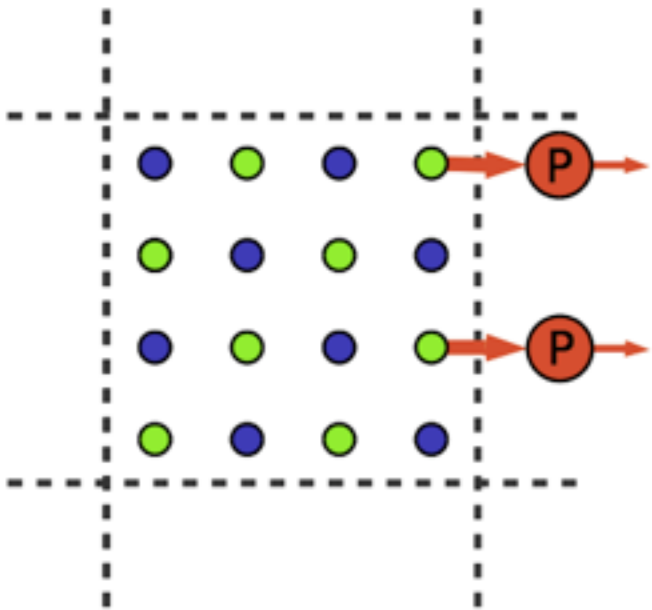
Multi-dimensional Kernel Computation



Step 1

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

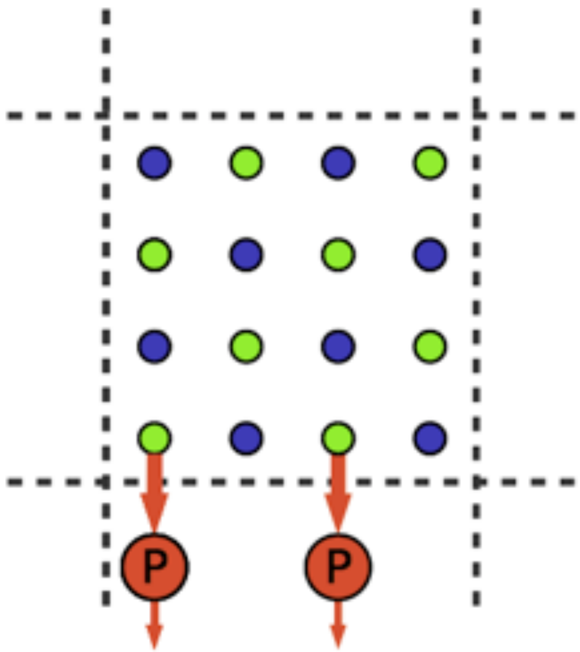
Multi-dimensional Kernel Computation



Step 1

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

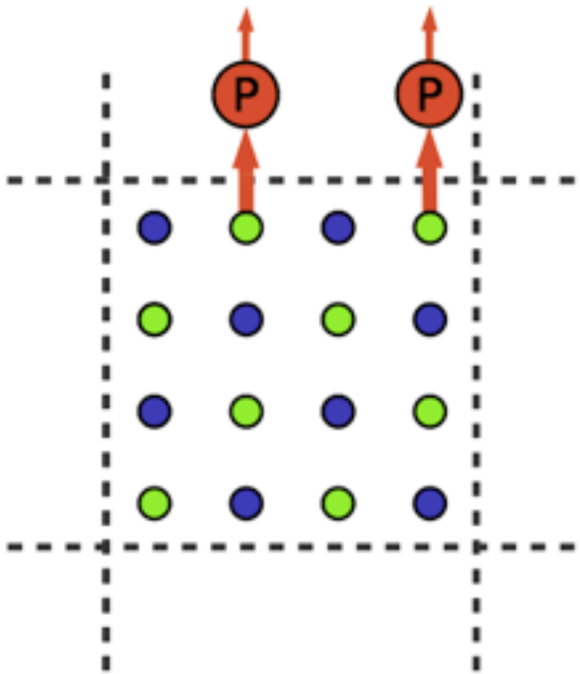
Multi-dimensional Kernel Computation



Step 1

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

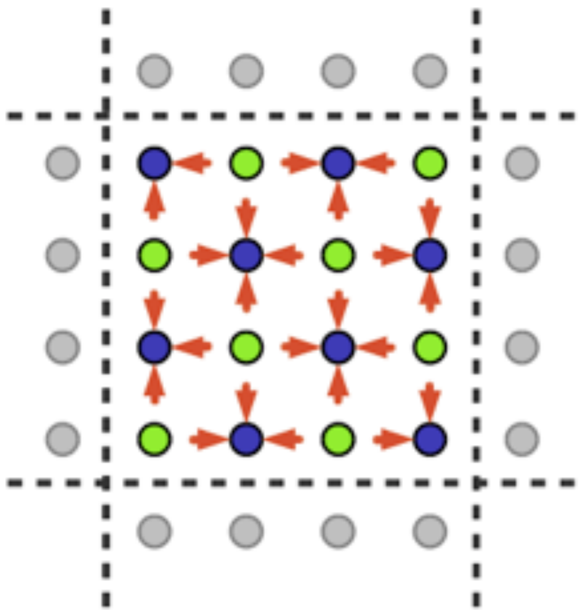
Multi-dimensional Kernel Computation



Step 1

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

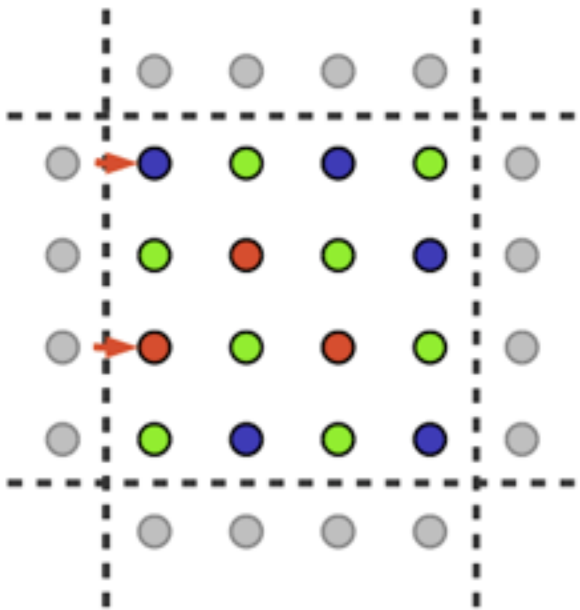
Multi-dimensional Kernel Computation



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



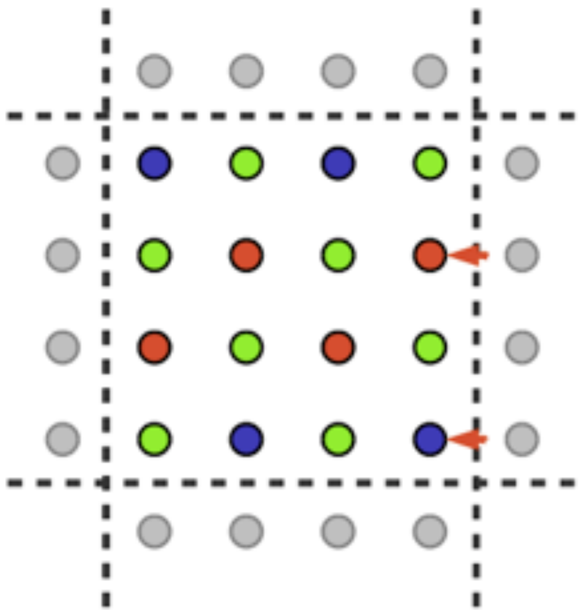
Step 3

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



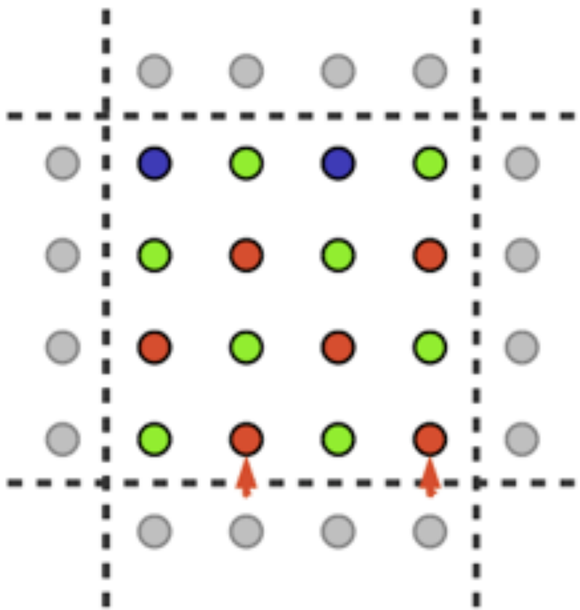
Step 3

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



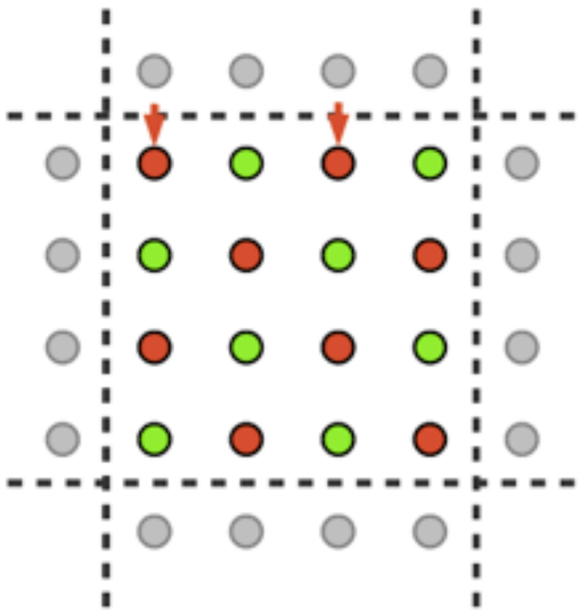
Step 3

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



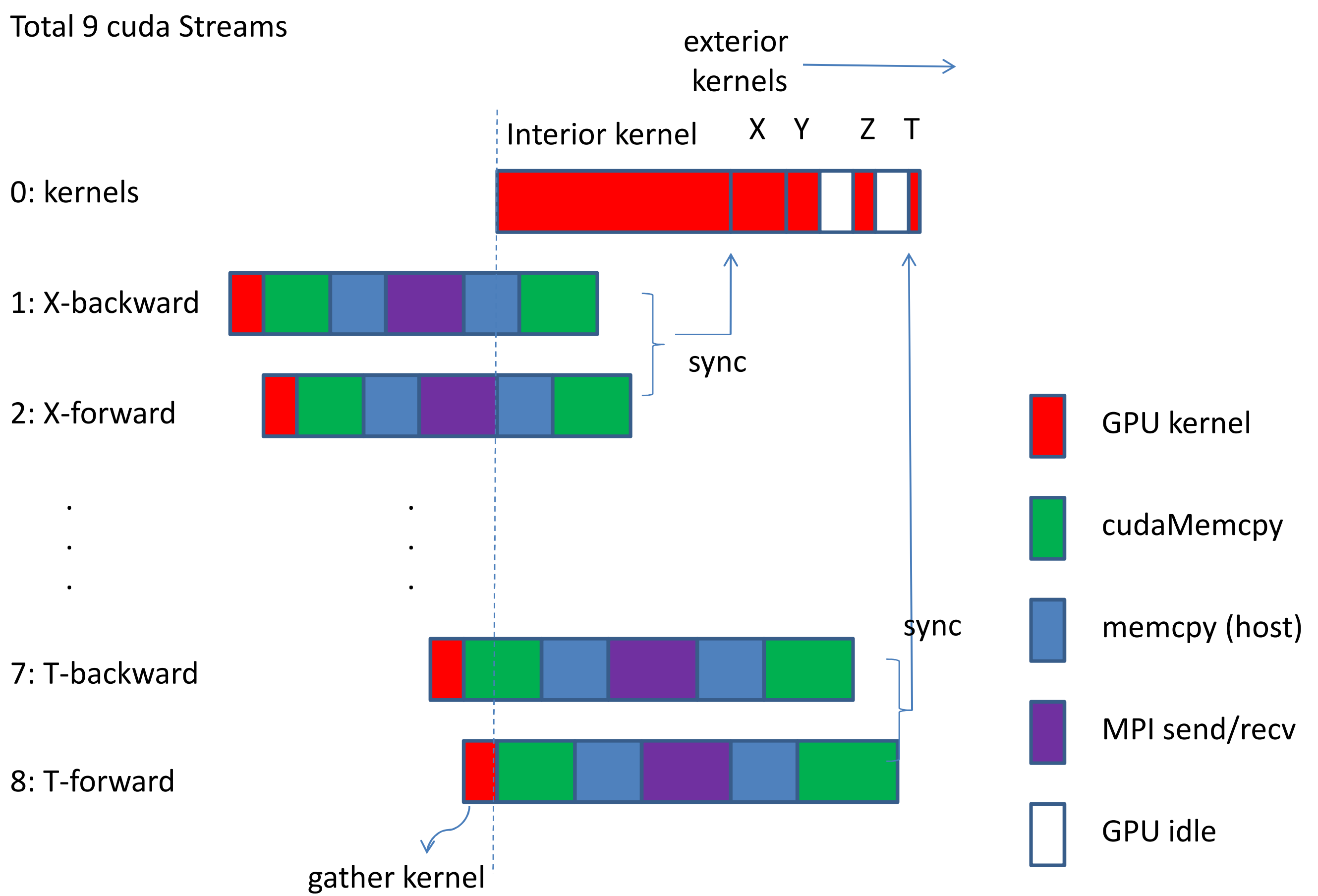
Step 3

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

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

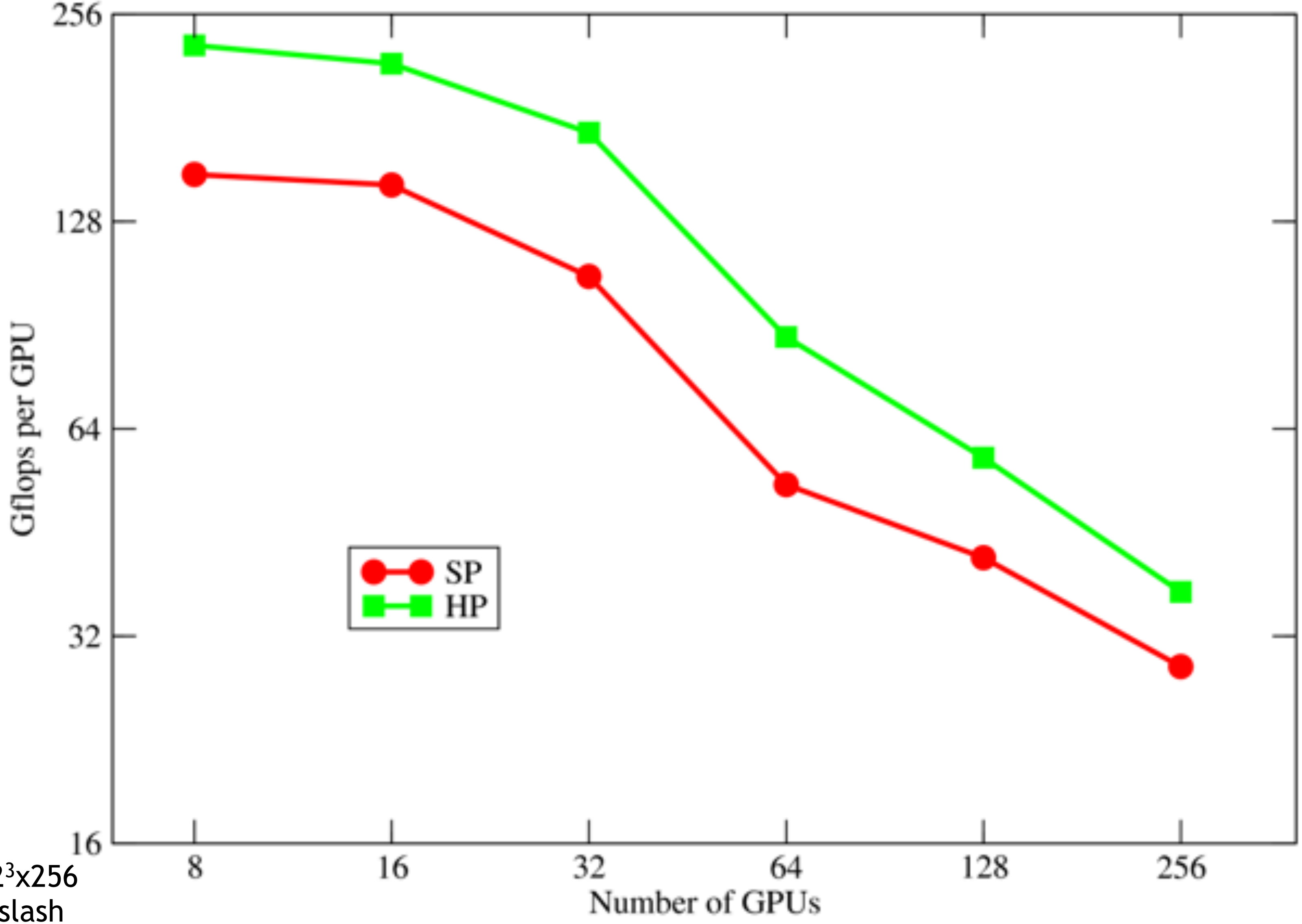
Note in higher dimensions corner sites have a race condition - serialization of kernels required

Multi-dimensional Communications Pipeline

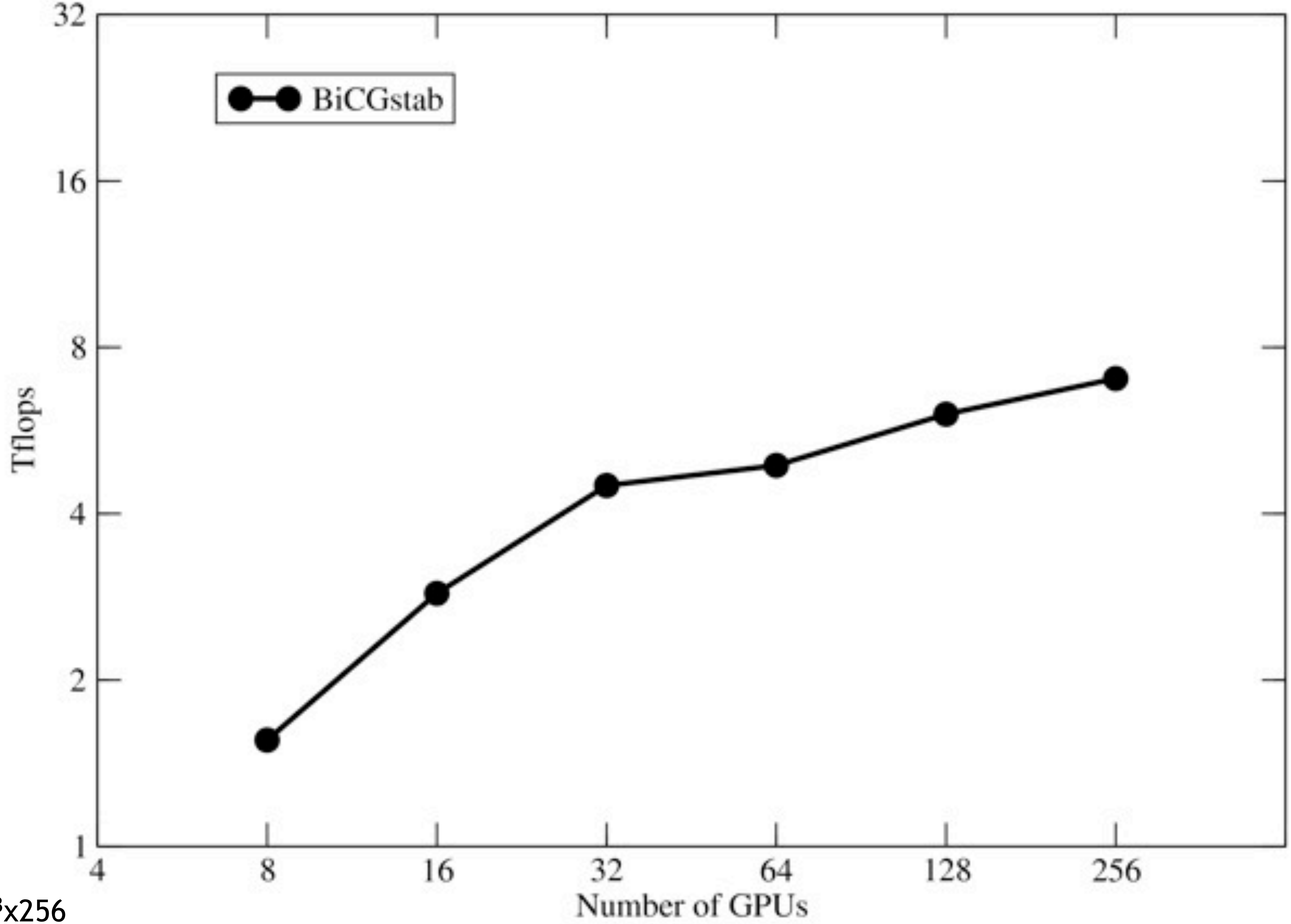


Performance results

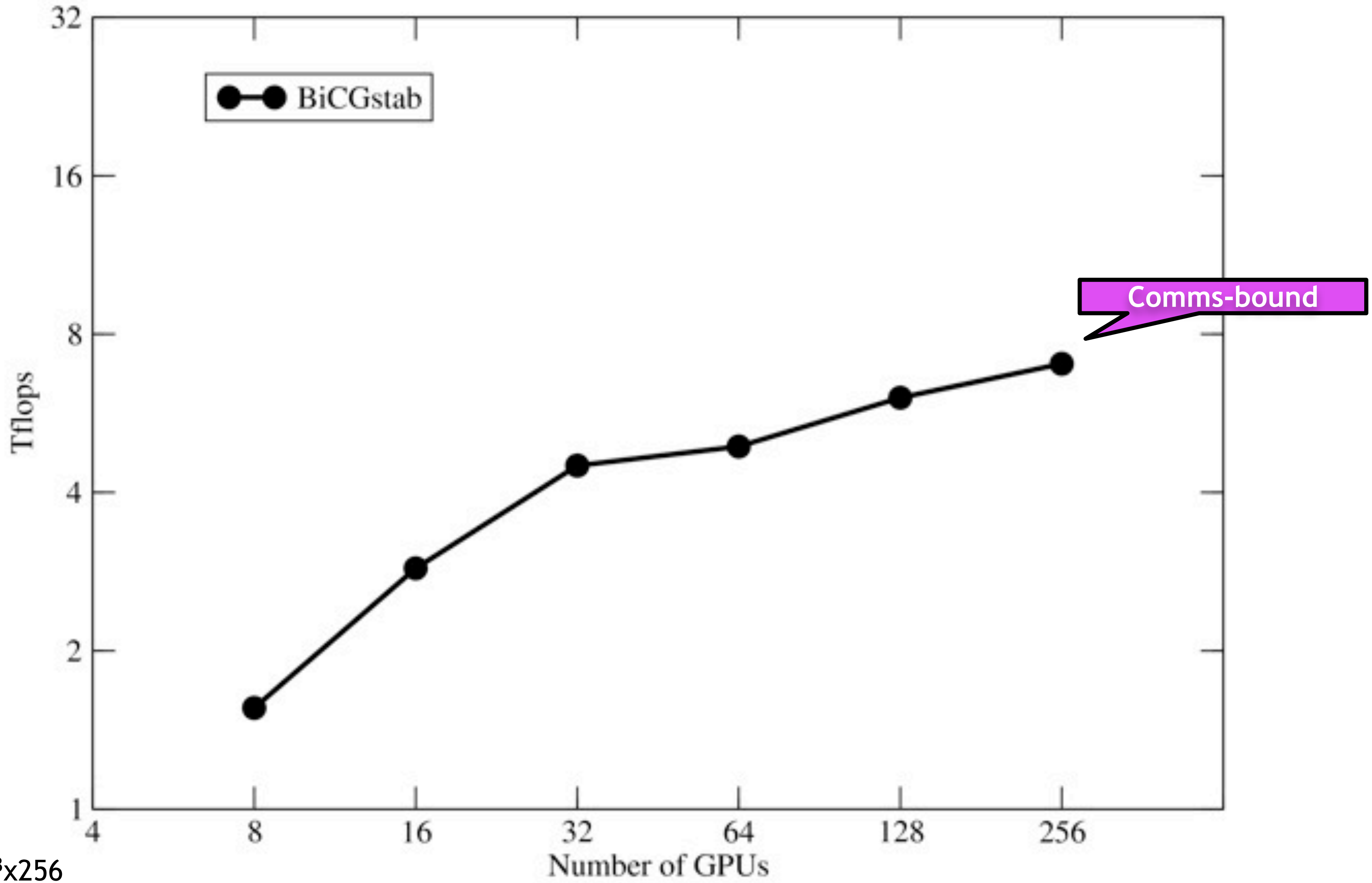
- Results presented at SC'11 (not taking advantage of more recent optimizations).
- Test Bed: "Edge" at LLNL
  - 206 nodes available for batch jobs, with QDR infiniband
  - 2 Intel Xeon X5660 processors per node (6-core Westmere @ 2.8 GHz)
  - 2 Tesla M2050 cards per node, sharing 16 PCI-E lanes via a switch
  - ECC enabled
  - CUDA 4.0



Strong scaling 32^3x256
Wilson-clover dslash



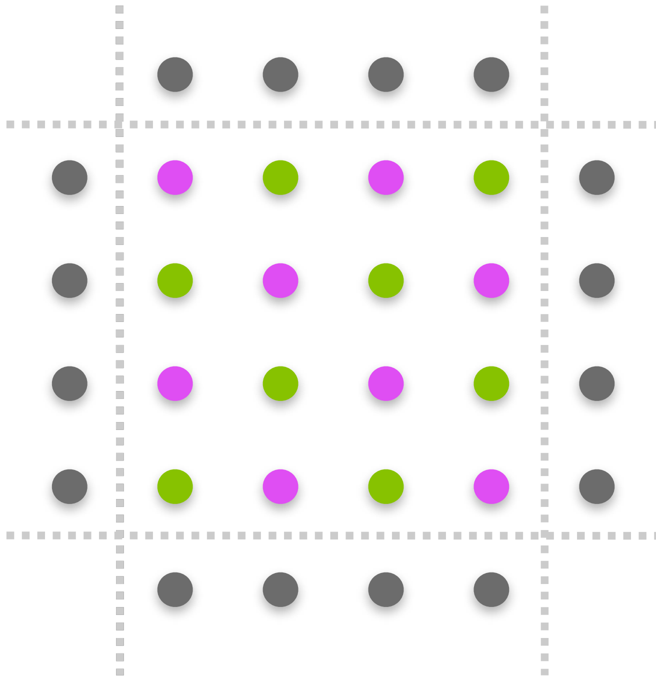
Strong scaling 32^3x256
Wilson-clover BiCGstab



Strong scaling 32^3x256
Wilson-clover BiCGstab

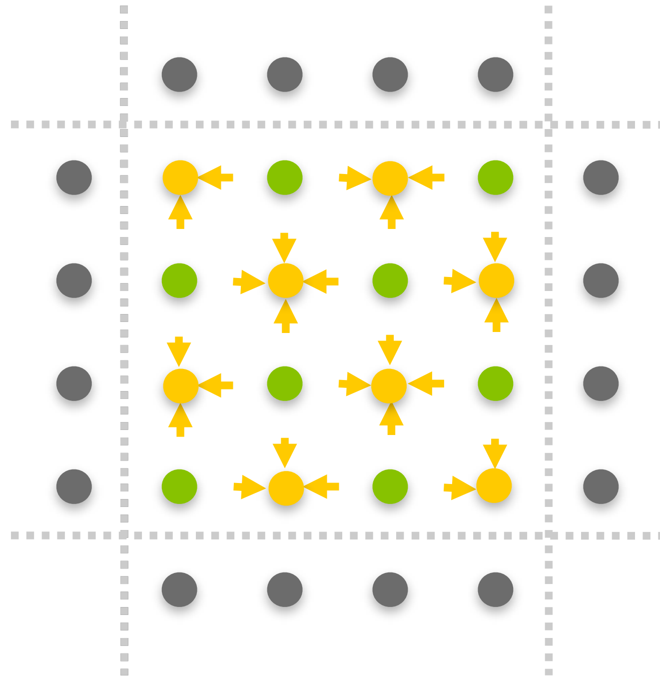
Building a scalable solver

- Inter-GPU communication hurts, so let's avoid it.
- In the strong-scaling regime, we employ a solver with a domain-decomposed preconditioner.
- Most of the flops go into the preconditioner, where communication is turned off.
- Half precision is perfect here.
- Iteration count goes up, but it's worth it.

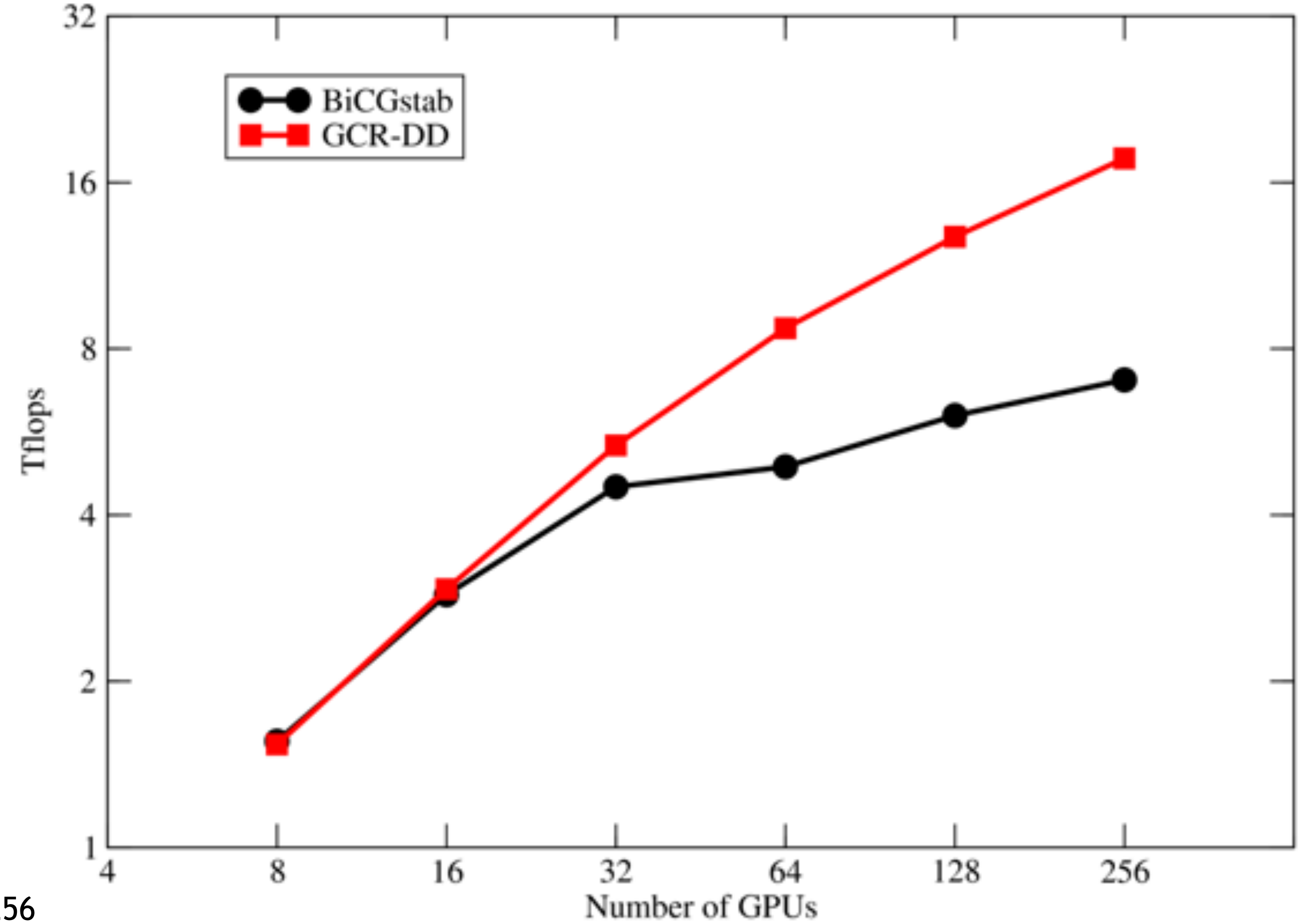


Building a scalable solver

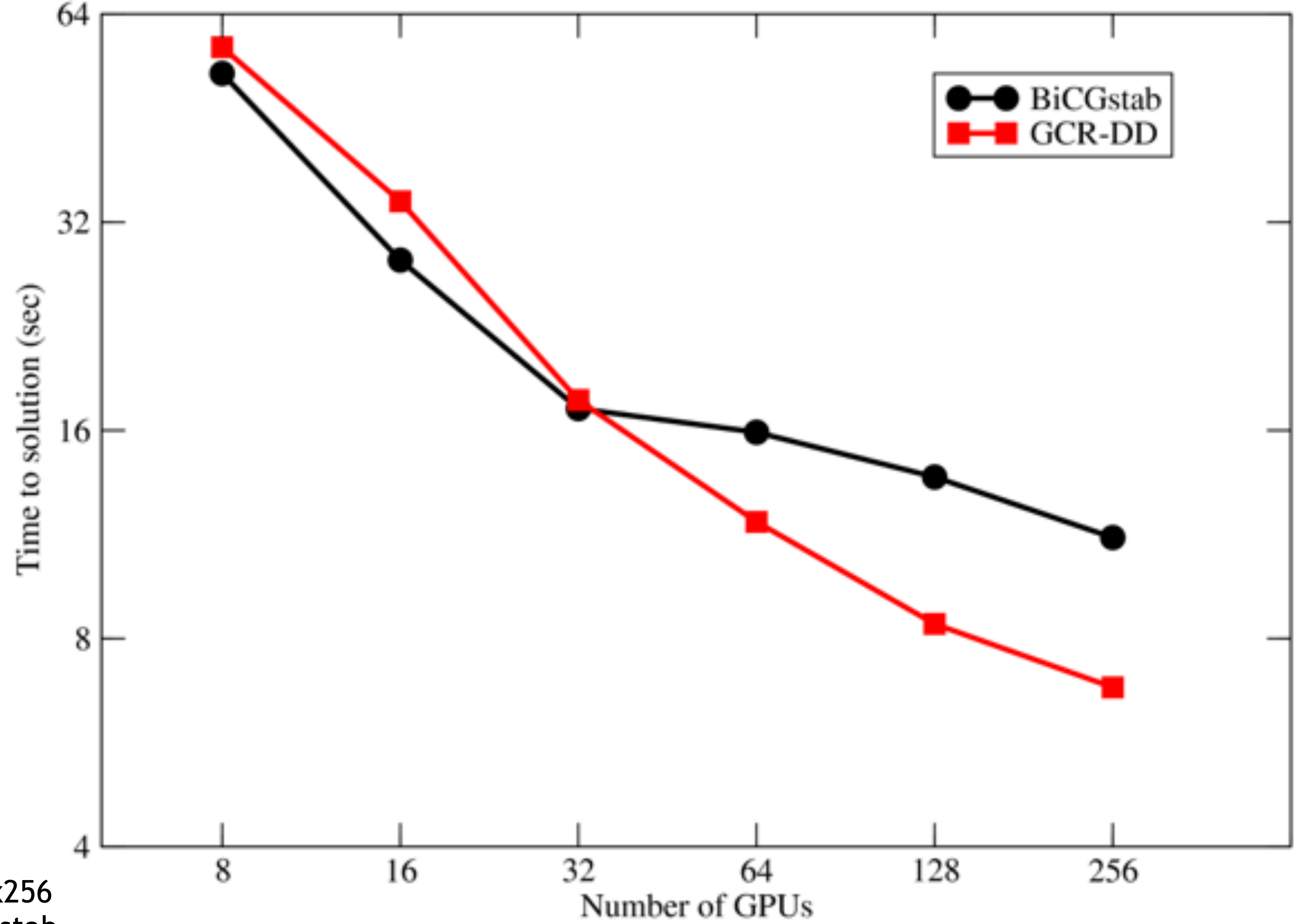
- Inter-GPU communication hurts, so let's avoid it.
- In the strong-scaling regime, we employ a solver with a domain-decomposed preconditioner.
- Most of the flops go into the preconditioner, where communication is turned off.
- Half precision is perfect here.
- Iteration count goes up, but it's worth it.



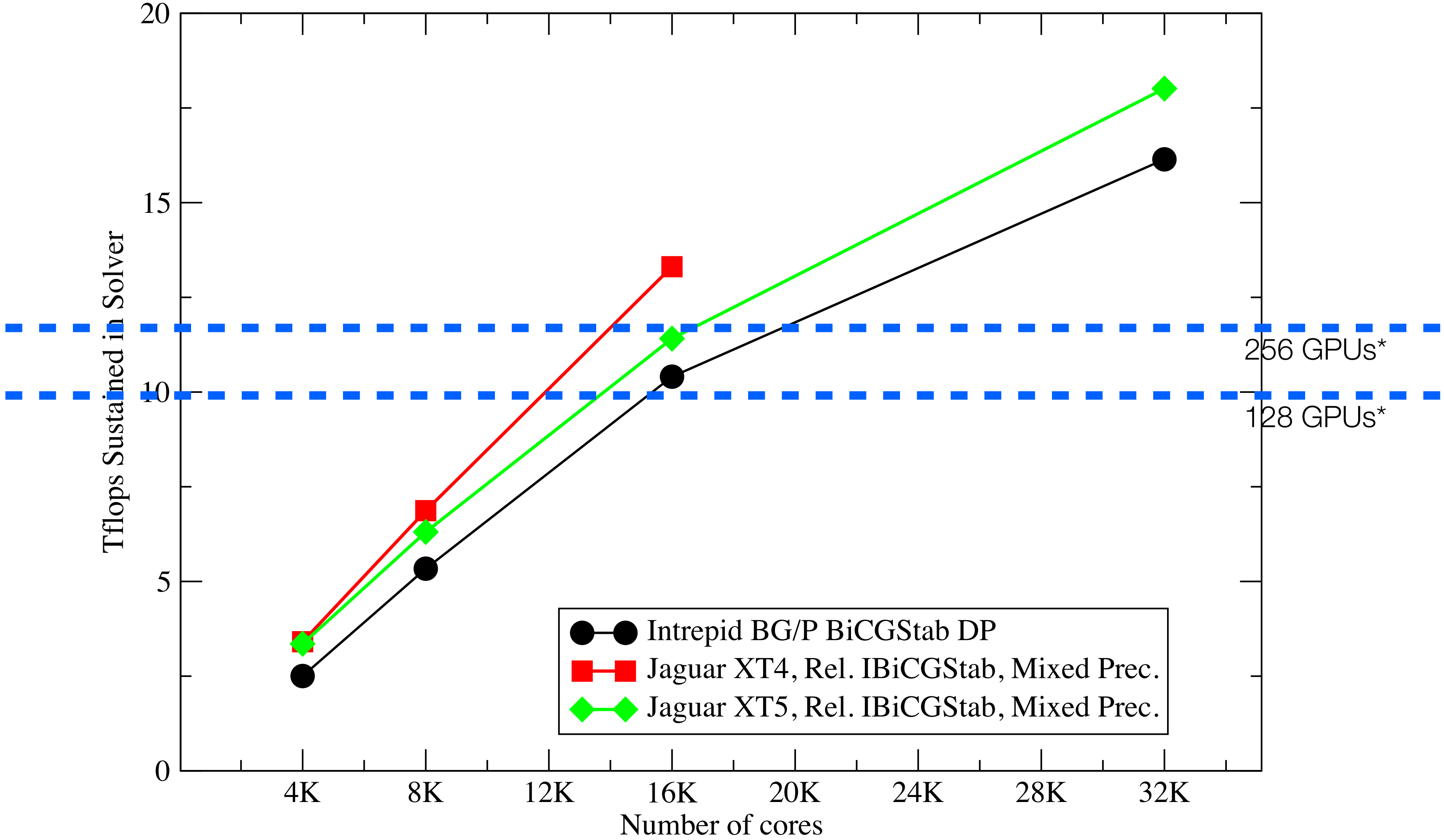
Done!



Strong scaling 32^3x256
Wilson-clover BiCGstab



Strong scaling 32^3x256
Wilson-clover BiCGstab



*GPU Tflops scaled according solver iterations

This is the future of capability computing...

Tsubame 2.0
4224 GPUs

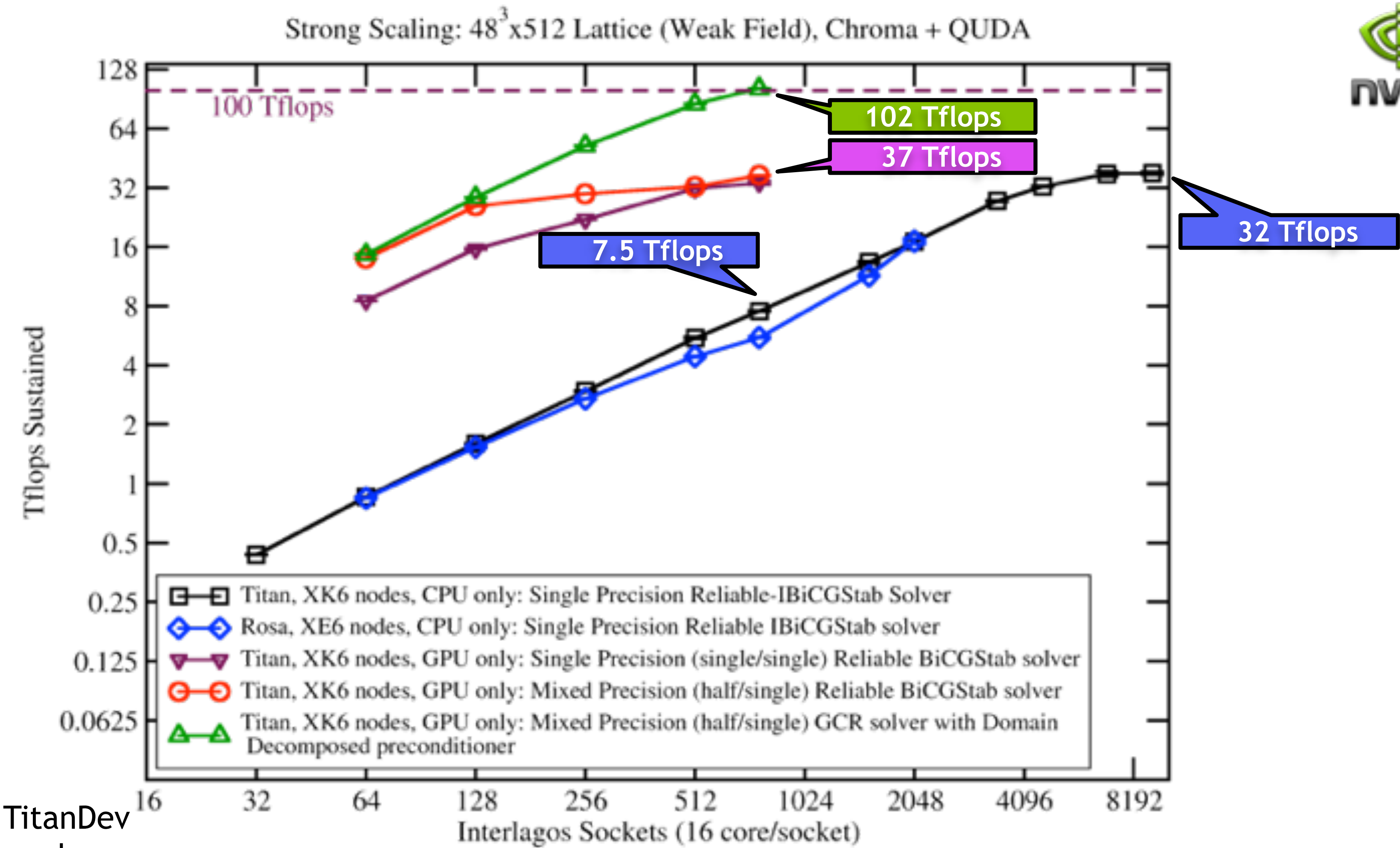
Tianhe-1A
7168 GPUs

coming soon...

Titan
>20 Petaflops
18,688 GPUs

Strong scaling on TitanDev (Cray XK6)

- 960 nodes, each with:
  - 1 Tesla X2090
  - 1 Opteron (16-core/8-module “Interlagos”)
- Cray Gemini interconnect
- Development platform in anticipation of Titan



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

What haven’t we covered?

- Non-solver kernels required for HMC
  - Gauge force, fermion force, link fattening
- Advanced optimizations
  - Using shared memory for cache blocking
  - Autotuning
  - Texture cache and half precision
  - and lots more

HMC timing breakdown

Time distribution for a run on 2048 XT3 (BigBen) cpus using a $40^3 \times 96$ grid ($5 \times 10^2 \times 6$ per cpu) with $m_l = 0.1 m_s$:

Activity	time(s)	MF/cpu	per cent
CG	2987	530	58.5
FF	1125	579	22.0
GF	489	469	9.5
Fat	442	627	8.7
Long	24	340	<1
Input config.	41		<1
total above	5108		
unaccounted	104		1.9
wallclock	5212		

Work in progress

- Gauge field generation on GPUs, for 2 different discretizations & applications:
  - Improved staggered in MILC
  - Wilson and Wilson-clover in Chroma (leveraging Frank Winter's QDP-JIT framework)
- Adaptive geometric multigrid on GPUs
  - GPUs give 5-10x in price/performance
  - Multigrid has the potential to give another 10x (at least for Wilson and Wilson-clover) at light quark masses.

Getting into QUDA

Using QUDA

- QUDA designed to accelerate pre-existing LQCD applications
  - Chroma, MILC, CPS, BQCD
- Solo QUDA workflow possible
  - tests directory includes linear solver examples
  - Gauge fields loaded through QIO
  - tests main use is for self contained correctness checking

Using QUDA

- QUDA provides a simple C interface for the outside world
    - Host applications simply pass cpu-side pointers
    - QUDA takes care of all field reordering and data copying
- Both a blessing and curse

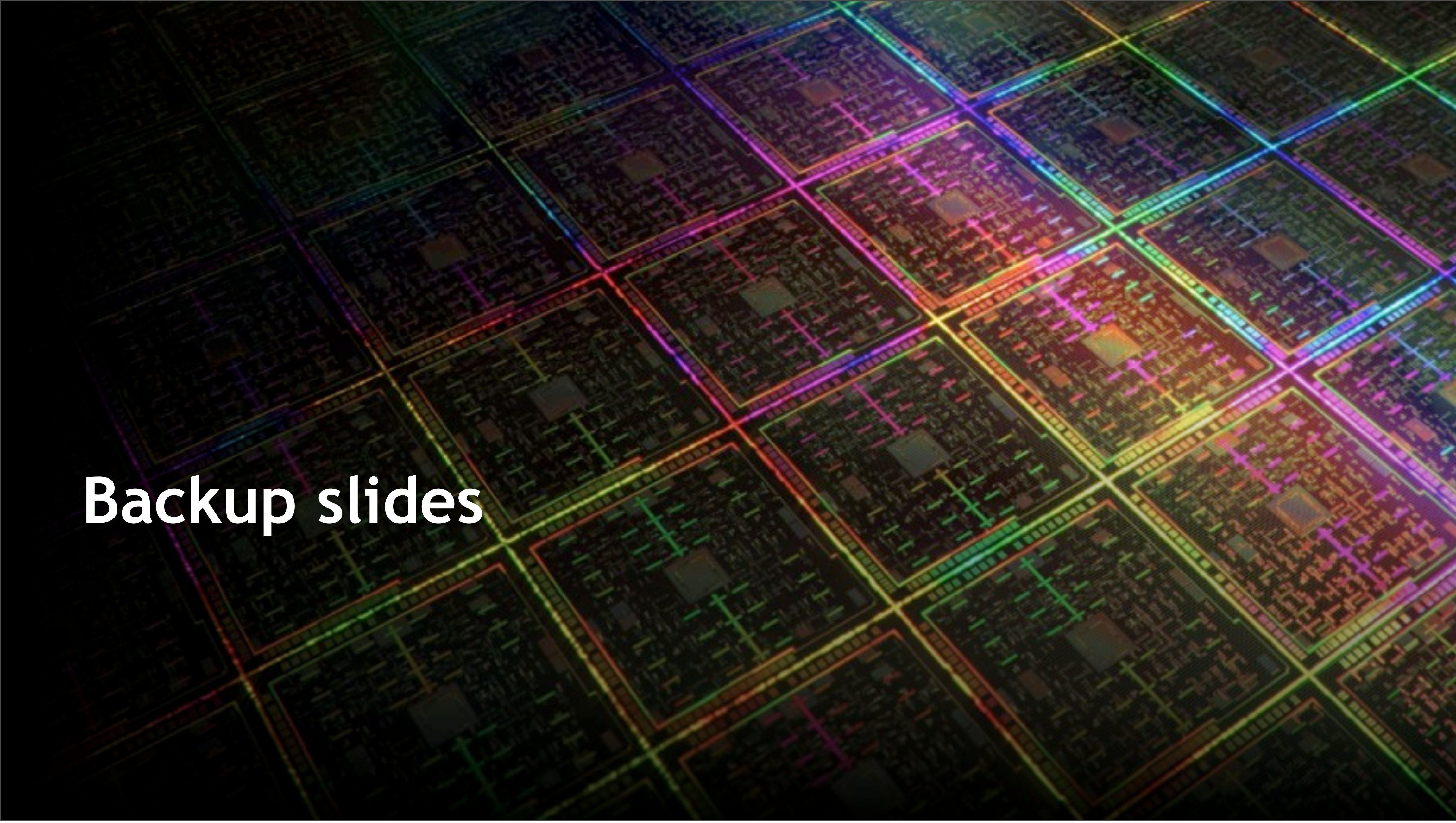
```
#include <quda.h>

int main() {

  // initialize the QUDA library
  initQuda(device);

  // load the gauge field
  loadGaugeQuda((void*)gauge, &gauge_param);

  // perform the inversion
  invertQuda(spinorOut, spinorIn, &inv_param)

  // free the gauge field
  freeGaugeQuda();

  // finalize the QUDA library
  endQuda();

}
```

Getting Involved with QUDA

- QUDA is open source
  - All development done in github
- Features requests are welcome
- More developers are even more welcome

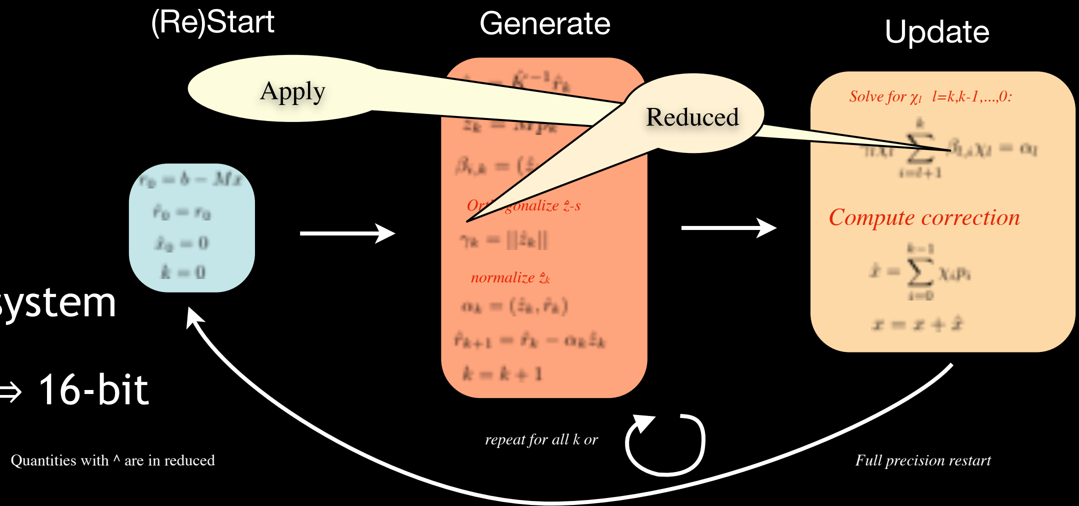
Summary

- Glimpse into the QUDA library
  - Implementing the Dslash
  - Multi-GPU considerations
- Possible take-home messages
  - Start experimenting with writing code with GPUs
    - CUDA C/C++, OpenACC, it doesn't matter
  - Using GPUs + QUDA as a black box to accelerate physics
  - Looking deeper into QUDA
    - contact me mclark@nvidia.com

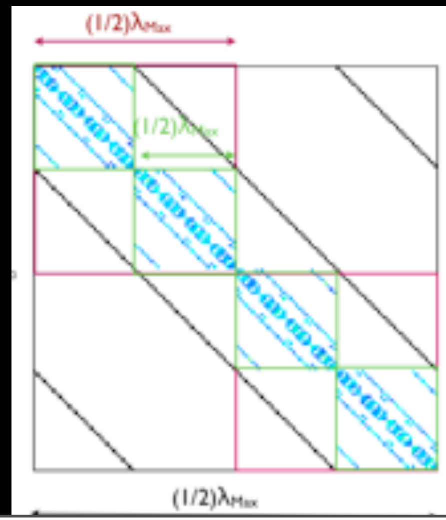


Backup slides

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