GPU TECHNOLOGY CONFERENCE

Unraveling the mysteries of quarks with hundreds of GPUs

Ron Babich
NVIDIA

Collaborators and “QUDA” developers

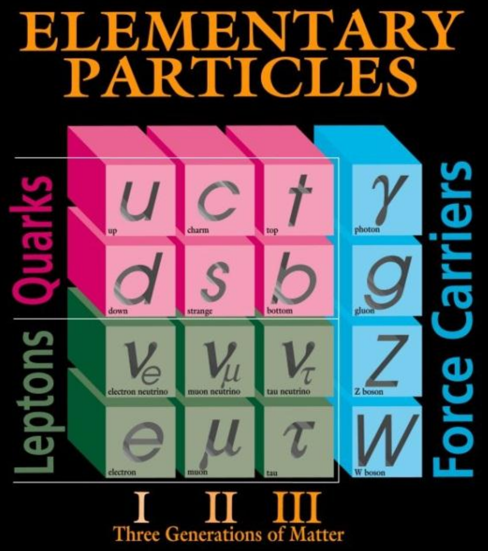
- Kip Barros (LANL)
- Rich Brower (Boston University)
- Mike Clark (NVIDIA)
- Justin Foley (University of Utah)
- Joel Giedt (Rensselaer Polytechnic Institute)
- Steve Gottlieb (Indiana University)
- Bálint Joó (Jefferson Lab)
- Claudio Rebbi (Boston University)
- Guochun Shi (NCSA)
- Alexei Strelchenko (Cyprus Institute)
- Frank Winter (University of Edinburgh)

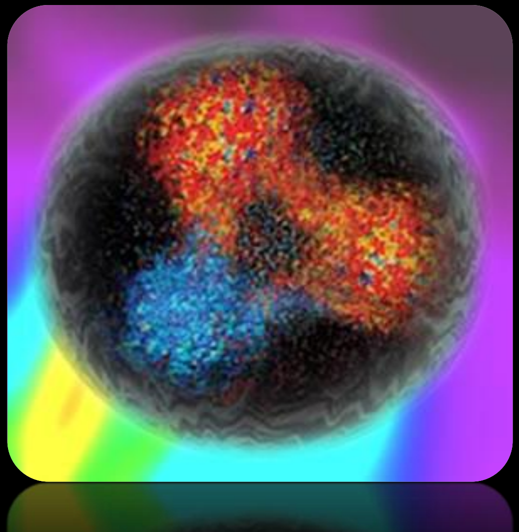
Overview

- Scientific motivation
- Lattice QCD as a computational problem
- Single-GPU strategies, optimizations, and performance
- Multi-GPU strategy and performance
- Scaling on TitanDev (up to 768 GPUs)
- Outlook

Quarks and gluons

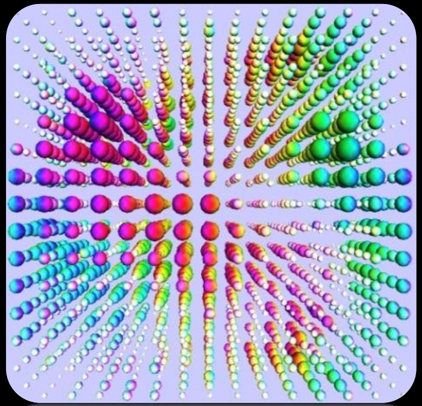
- The proton and neutron are not fundamental.
- They're made of 3 quarks each, plus force-carrying particles called gluons (and other quarks that pop into and out of existence from the quantum vacuum).
- Bound together by the strong force, one of the 4 known forces of nature.
  - Others are gravity, electromagnetism, and the weak force.





QCD and lattice QCD

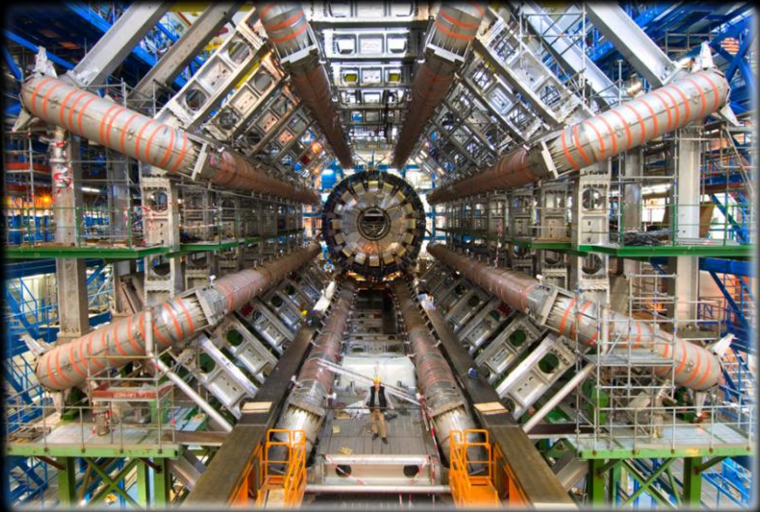
- Quantum Chromodynamics (QCD) is the theory that describes the interactions of quarks and gluons.
- We know the equations, but solving them is hard.
- Lattice QCD is the only known *ab initio* method.
- Key idea is to replace spacetime with a 4D grid and sample the configurations of quark and gluon fields.
- More samples → smaller statistical errors
- Most of the runtime is spent in linear solvers involving a local (radius 1 or radius 3) stencil operator.

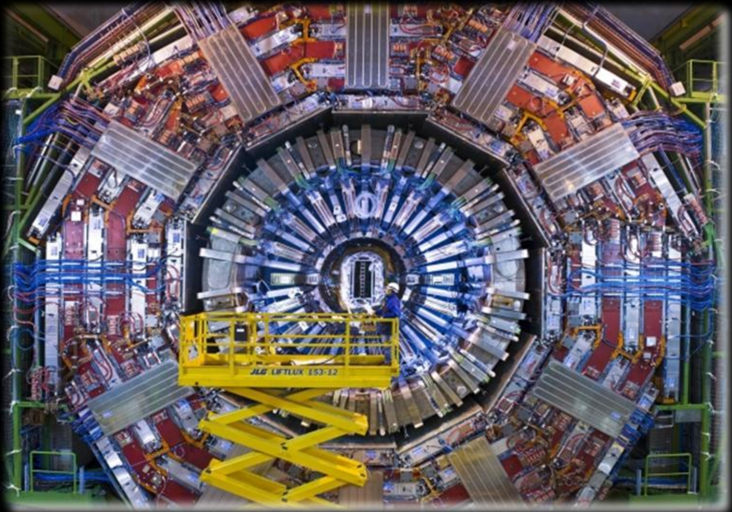


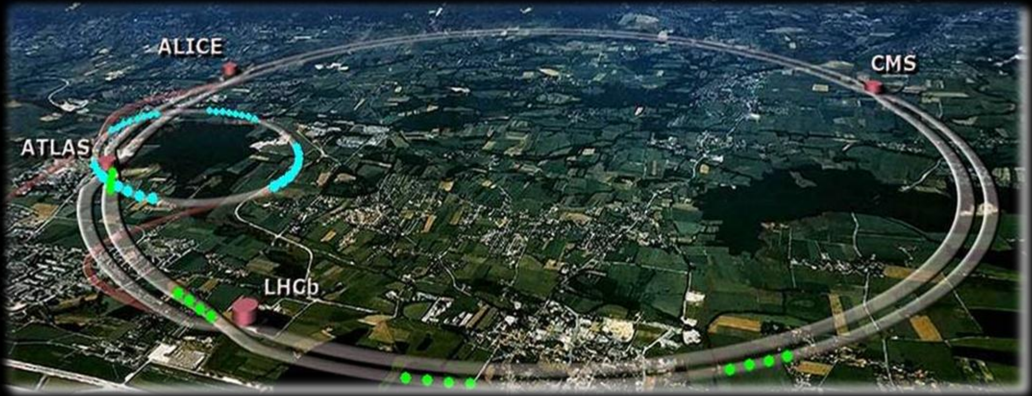
Questions for LQCD to answer

- What's the mass of the proton (without cheating)?
  - The masses of the quarks contribute only a few percent…
- What's the mass of a given short-lived particle? (glimpsed only briefly at accelerator experiments)
- What's the internal structure of these particles? Parameters feed into experiments attempting to discover:
  - the origin of dark matter in the universe (10x the density of visible matter)
  - why there's more matter (us) than antimatter
- What was the universe like in the first μs after the big bang?

New “strong dynamics” at the LHC?



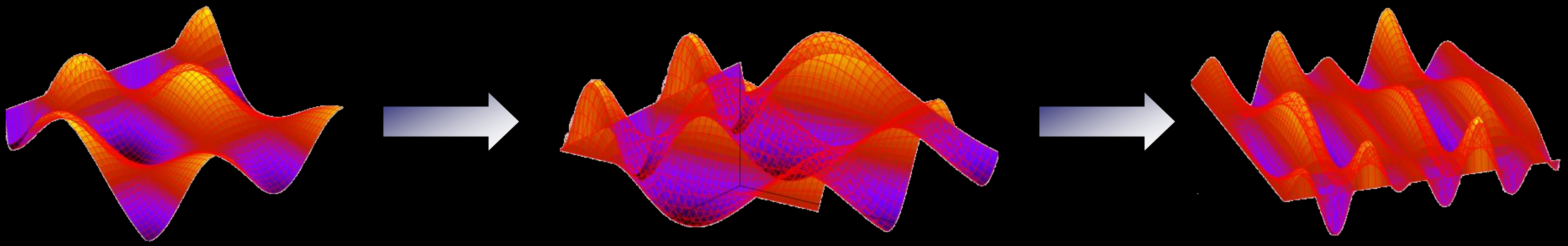




LHC Live: http://op-webtools.web.cern.ch/op-webtools/vistar/vistars.php?usr=LHC3

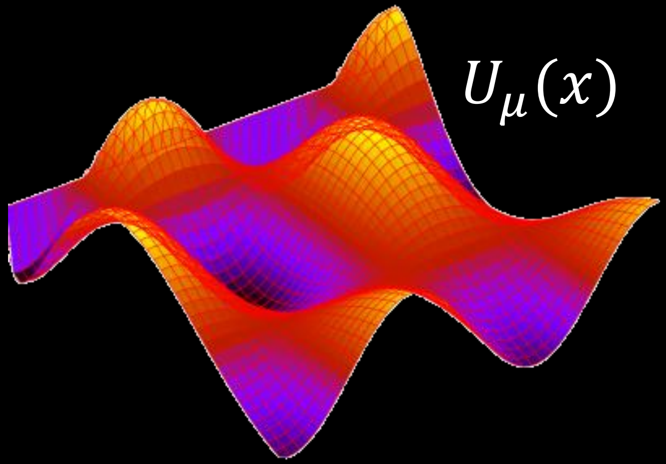
Steps in a lattice QCD calculation

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.

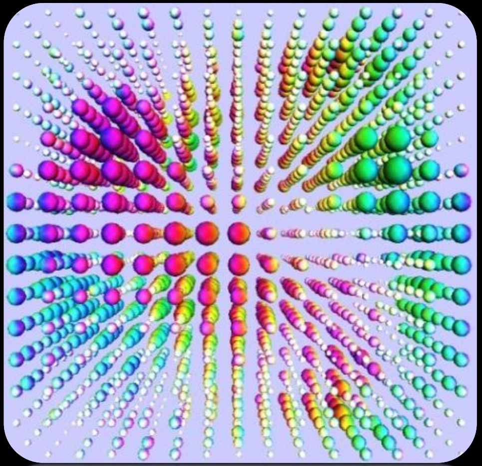


$$D_{ij}^{\alpha\beta}(x,y;U)\psi_j^\beta(y) = \eta_i^\alpha(x)$$

or "$Ax = b$"

Krylov solvers

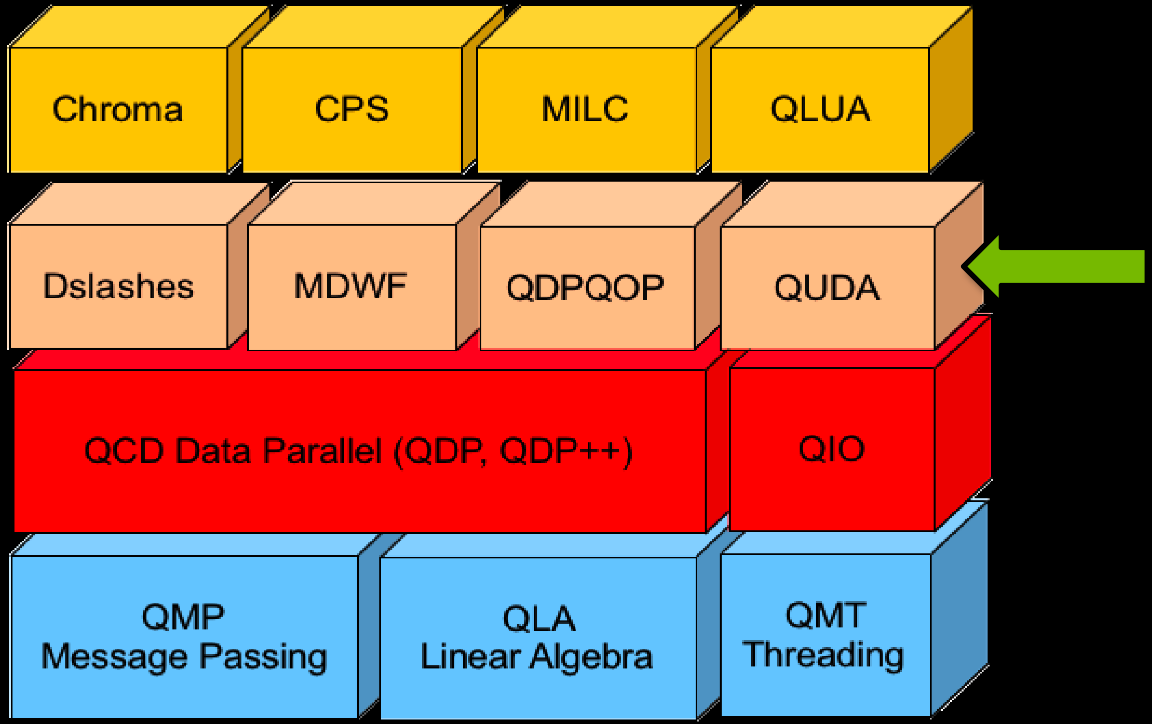
- (Conjugate gradients, BiCGstab, and friends)
- Search for the solution to $Ax = b$ in the subspace spanned by $\{b, Ab, A^2b, \dots\}$.
- Upshot:
  - We need fast code to apply A to an arbitrary vector (called the *Dslash* operation in LQCD).
  - ... as well as fast routines for vector addition, inner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



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!

USQCD software stack



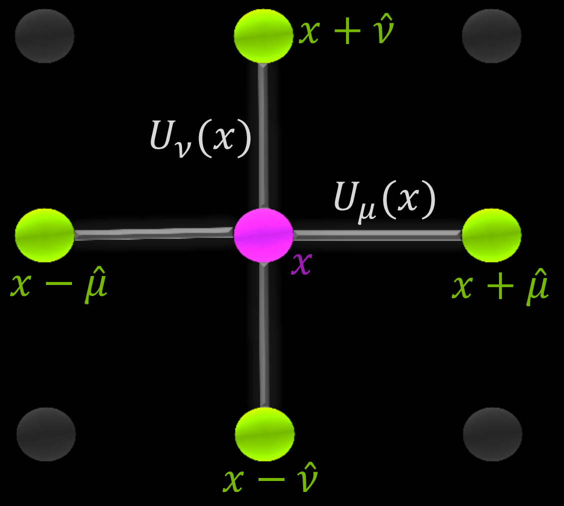
(Many components developed under the DOE SciDAC program)

Our "A": The Wilson-clover operator

- One of several common discretizations:

$$D(x,y) = -\frac{1}{2}\sum_{\mu=1}^{4}\left[P_\mu^- \otimes U_\mu(x)\delta(x+\hat{\mu},y) + P_\mu^+ \otimes U_\mu(x-\hat{\mu})\delta(x-\hat{\mu},y)\right] + A(x)\delta(x,y)$$

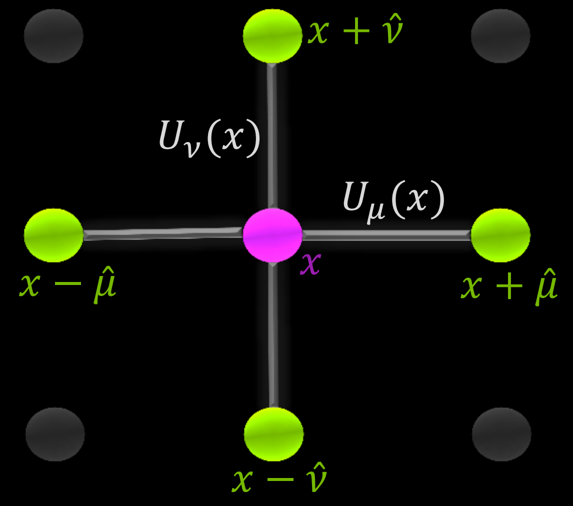
- 9-point stencil in 4 dimensions
- $P_\mu^\pm$ are 4x4 projection matrices acting in "spin" space, with entries ∈ $\{0, \pm 1, \pm i\}$ (never explicitly stored).
- $U_\mu(x)$ are fields of 3x3 complex matrices acting in "color" space.
- $A(x)$ is a field of 12x12 complex matrices.
- Altogether, our vector consists of 12 complex numbers per site.



$$\begin{array}{ccccc} (spin) & \otimes & (color) & \otimes & (spacetime) \\ 4 & \times & 3 & \times & N_x N_y N_z N_t \end{array}$$

We're bandwidth-bound

- Per lattice site, this matrix-vector product involves
  - 1824 flops
  - 432 floats in/out
- Byte/flop ratio
  - = 0.95 in single precision
  - = 1.90 in double
- Linear algebra is even worse.
  - Byte/flop = 12 for $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{c}$ in single precision



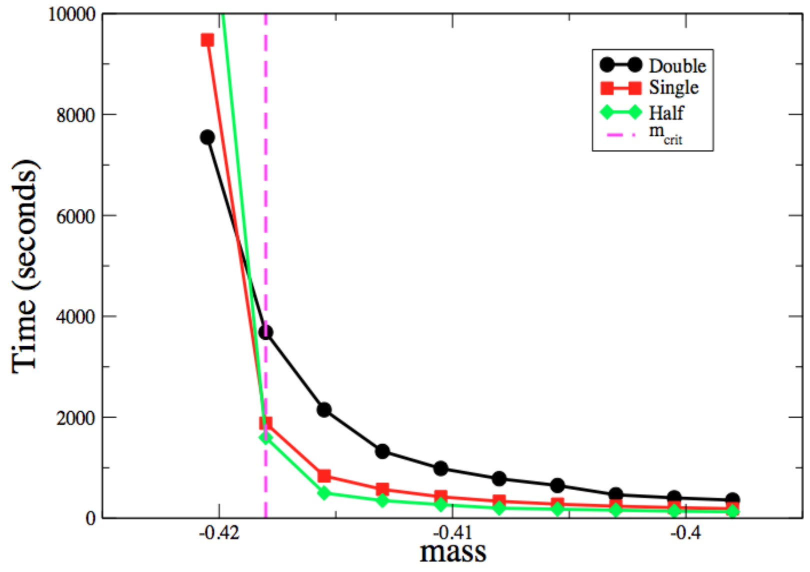
Strategies (details to follow)

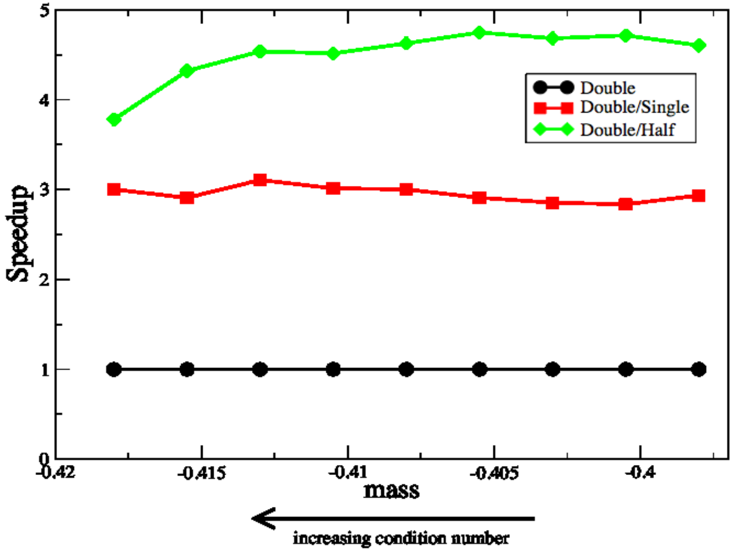
- Reduce memory traffic:
  - Recompute data on the fly
  - Take advantage of symmetries of the matrix to increase sparsity
  - Fuse kernels where ever possible
  - Aggressively employ mixed-precision solvers
- Auto-tune launch parameters for all performance-critical kernels:
  - Thread-block and grid dimensions
  - Number of thread blocks (by over-allocating shared memory)
- Block data in shared memory and L2.

Mixed precision with reliable updates

- Mixed-precision solver with “reliable updates” does most work in half precision, but maintains double-precision accuracy.

Results on GTX 280 (for illustration)



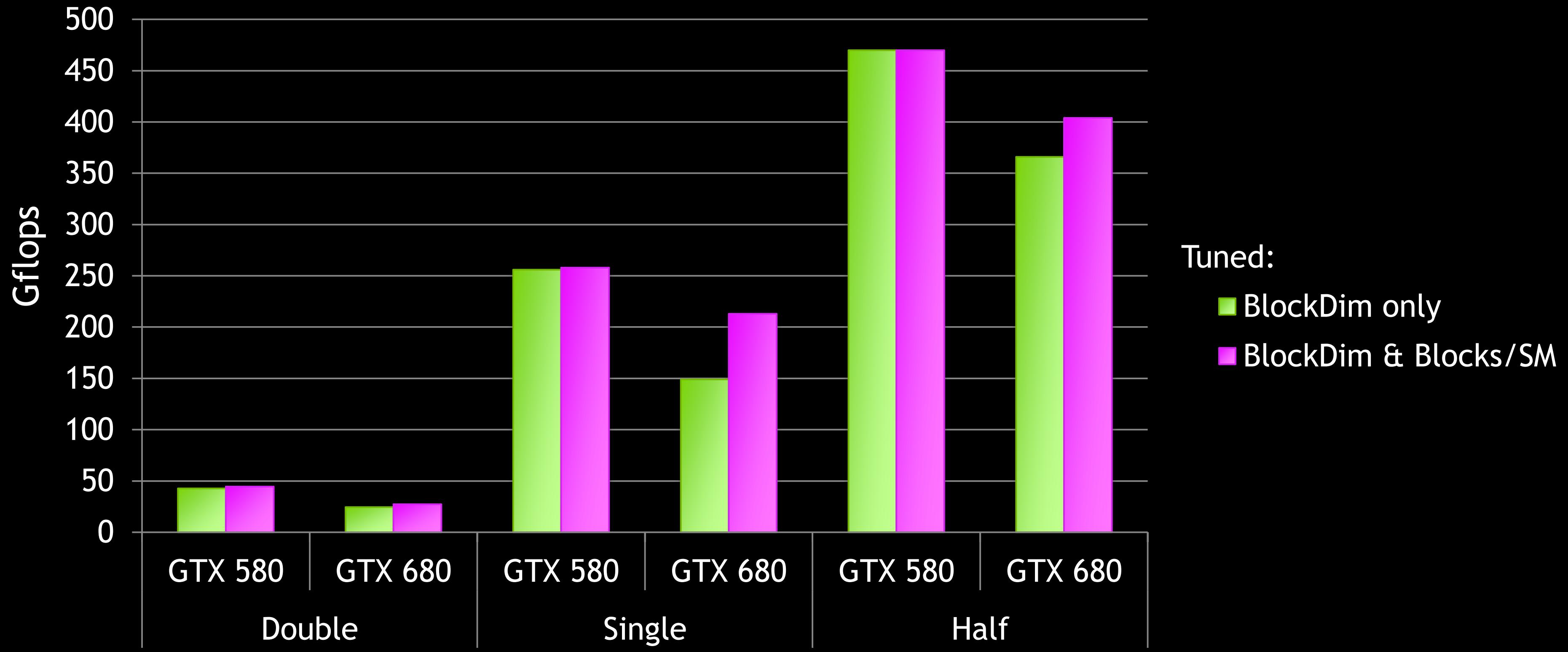


Run-time autotuning

- 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.

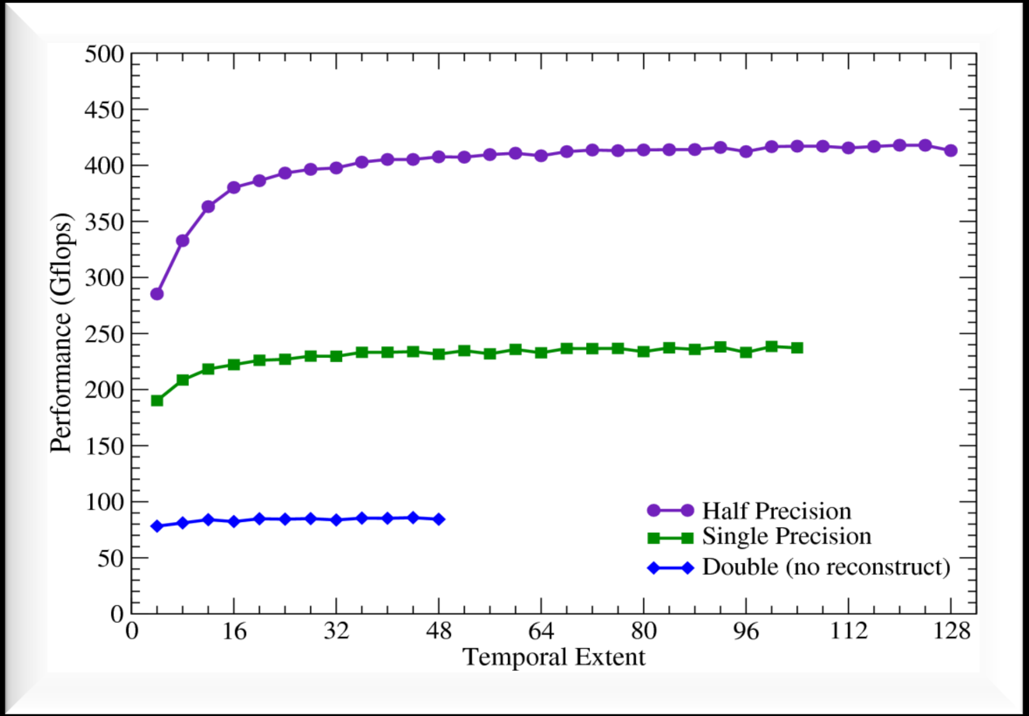
Virtual member functions of Tunable

- 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.

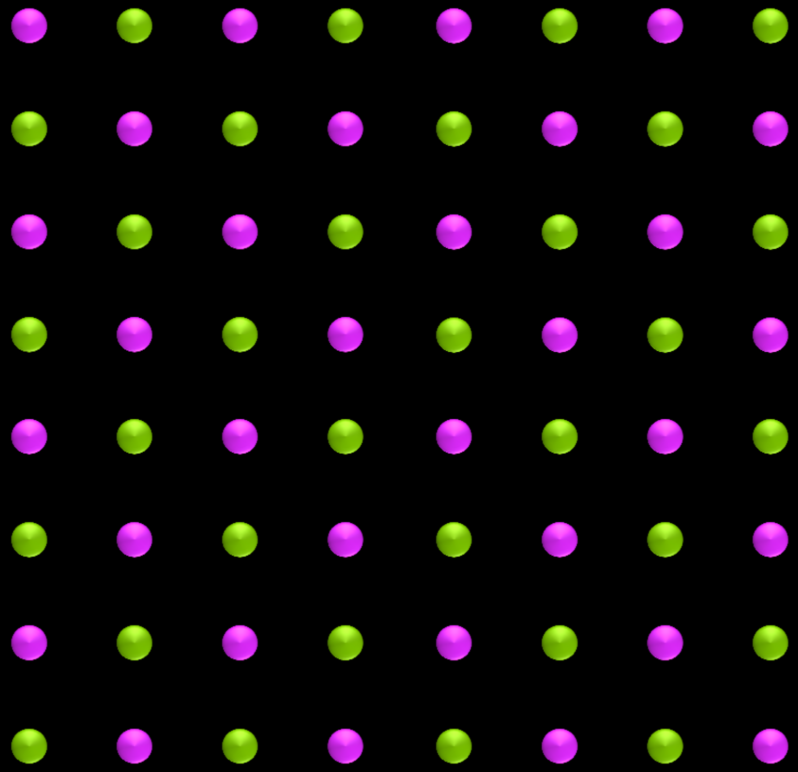
GPU TECHNOLOGY CONFERENCE

Matrix-vector performance

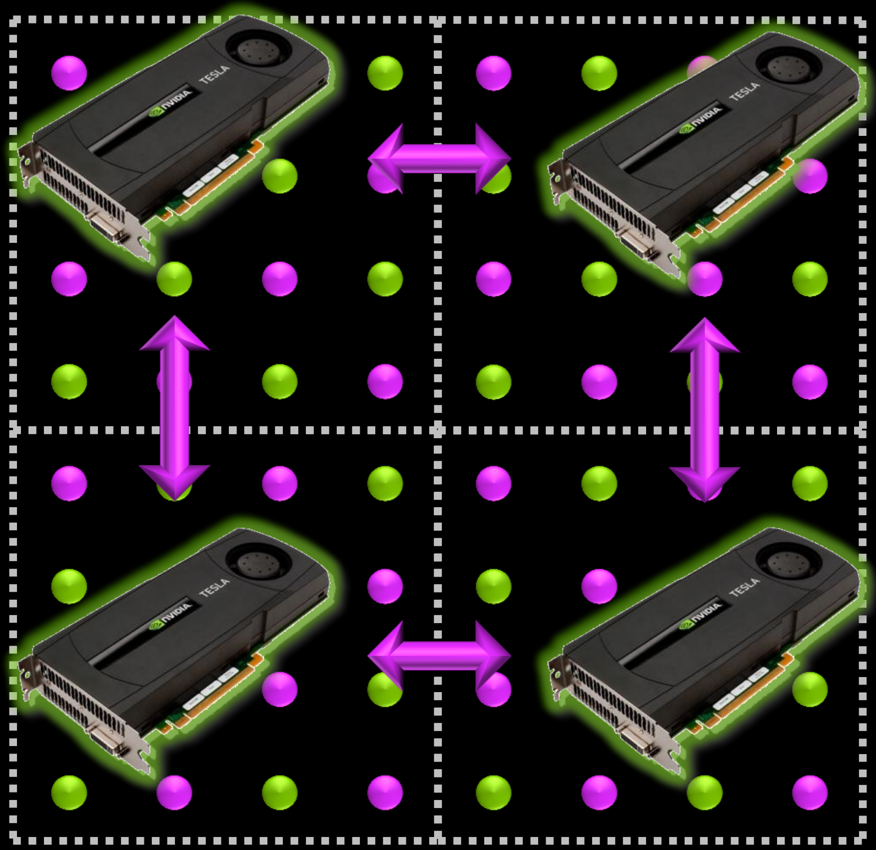
- For illustration; not our latest and greatest.
- Runs were done on a single GTX 480 (≈ Tesla M2090)
- Typical single-precision performance on a dual-Westmere node for comparison:
  - ≈ 25 Gflops for typical (optimized) production code
  - ≈ 50 Gflops might be possible following Smelyanskiy et al. (Intel, 2011)
- Spatial volume held fixed at $N_x N_y N_z = 24^4$



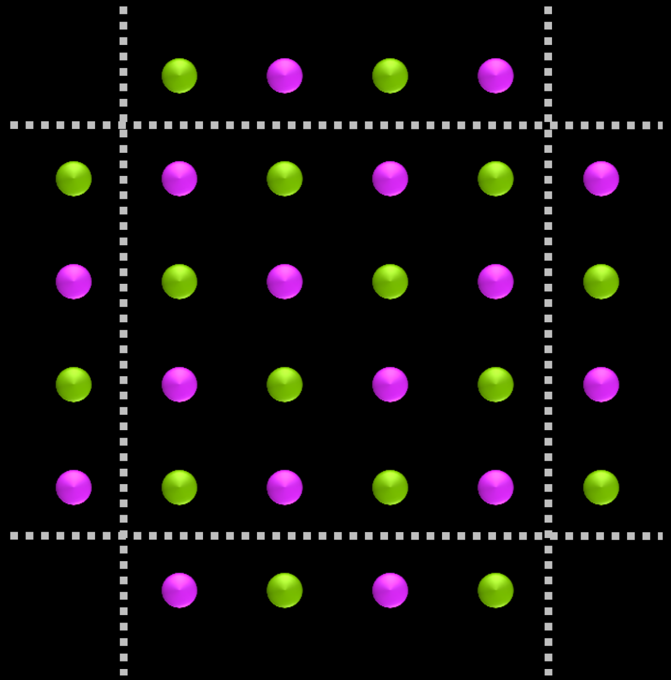
Parallelizing the Dslash



Parallelizing the Dslash



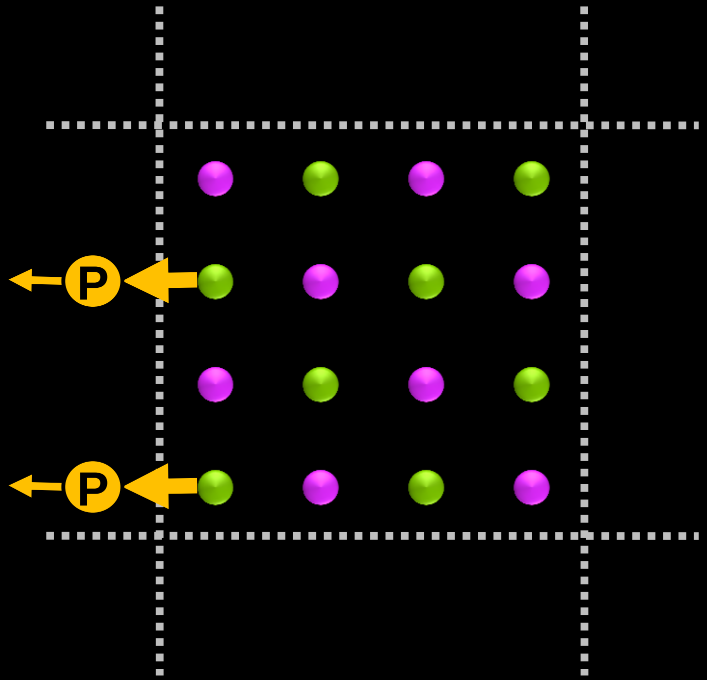
Parallelizing the Dslash



- For illustration, consider a 2D problem with a 4^2 local volume.
- Because we employ even/odd (red/black) preconditioning, only half the sites will be updated per "Dslash" operation.
- We'll take these to be the purple sites.

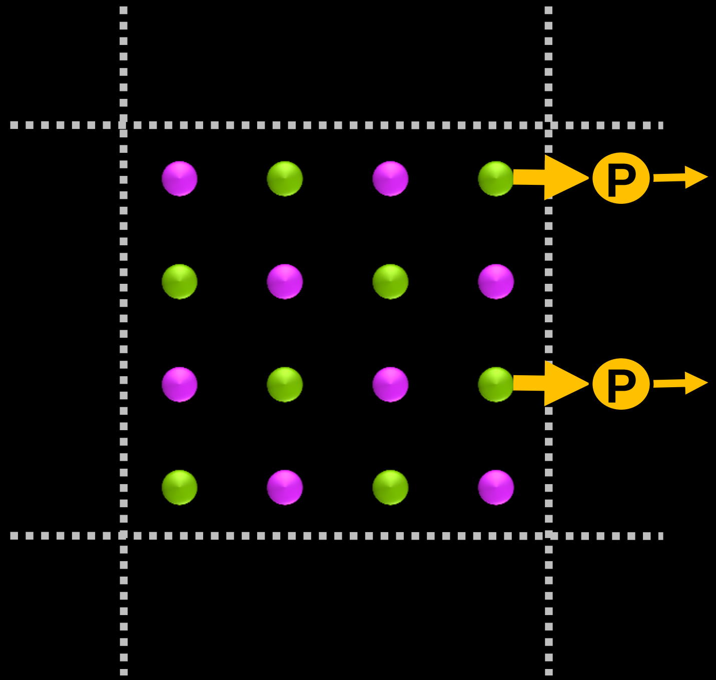
GPU TECHNOLOGY CONFERENCE

Parallelizing the Dslash



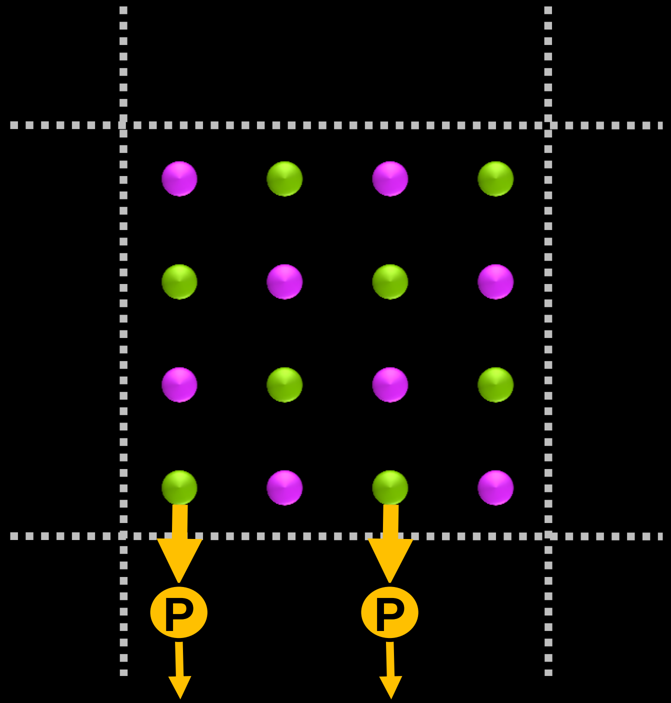
- Step 1:
  - Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
  - As part of the gather kernel, perform a "spin projection" step:
    - Reduces 24 floats → 12 floats
    - Costs only 12 adds

Parallelizing the Dslash



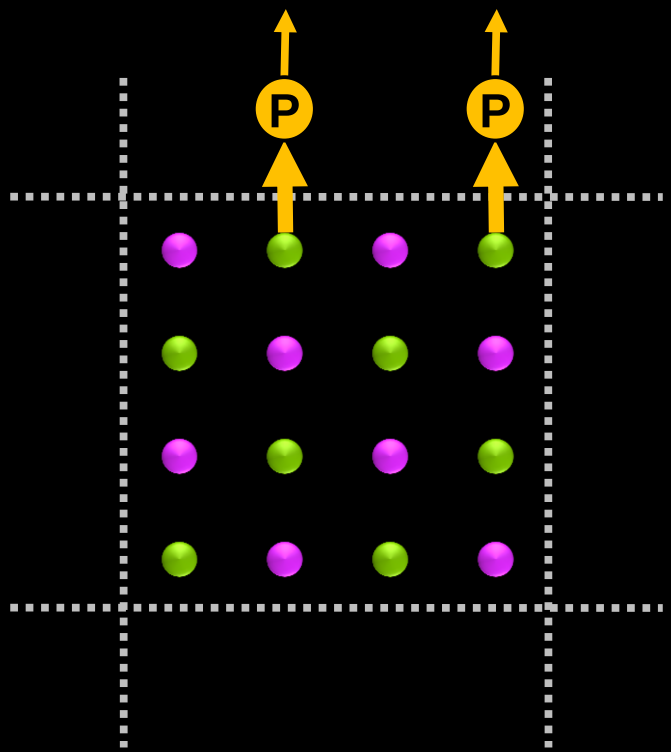
- Step 1:
  - Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
  - As part of the gather kernel, perform a "spin projection" step:
    - Reduces 24 floats → 12 floats
    - Costs only 12 adds

Parallelizing the Dslash



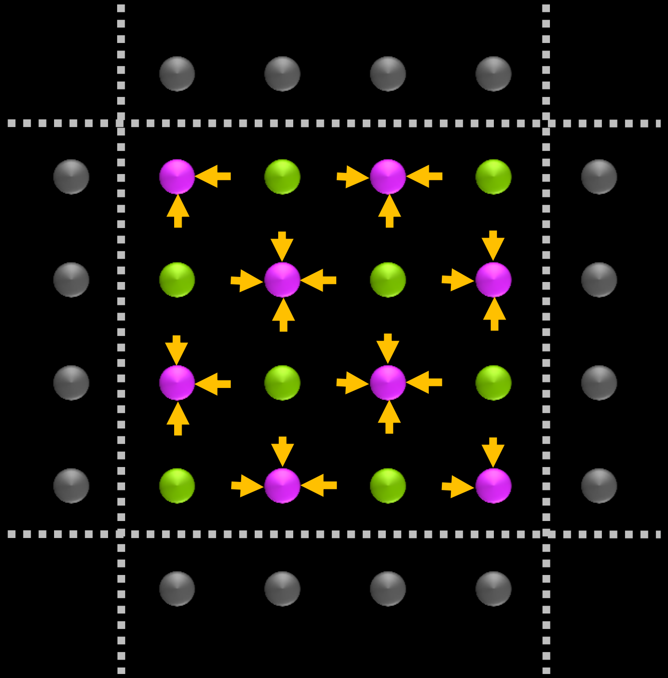
- Step 1:
  - Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
  - As part of the gather kernel, perform a "spin projection" step:
    - Reduces 24 floats → 12 floats
    - Costs only 12 adds

Parallelizing the Dslash



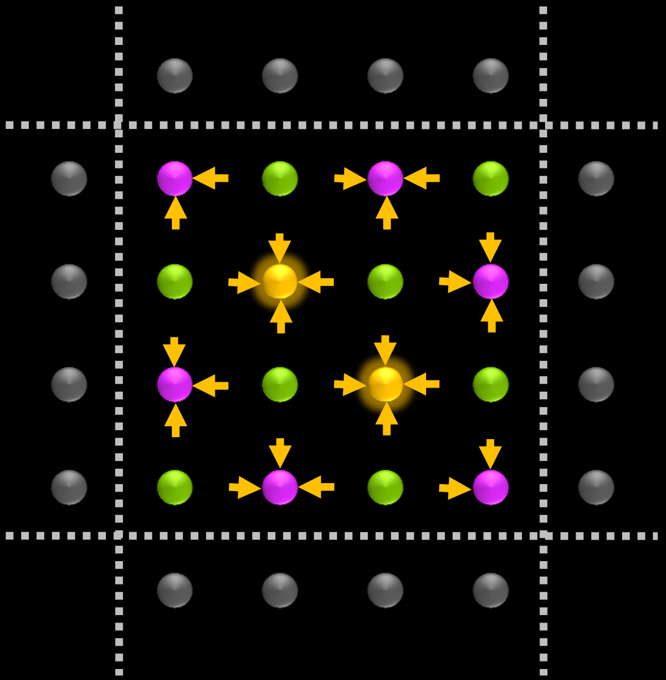
- Step 1:
  - Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
  - As part of the gather kernel, perform a "spin projection" step:
    - Reduces 24 floats → 12 floats
    - Costs only 12 adds

Parallelizing the Dslash



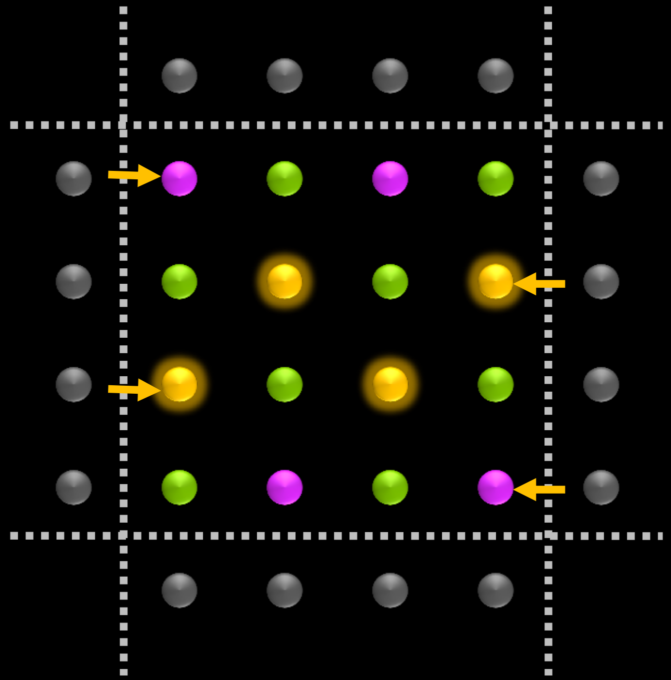
- Step 2:
  - An "interior kernel" updates all sites to the extent possible.
  - Sites along the boundary receive contributions from local neighbors.

Parallelizing the Dslash



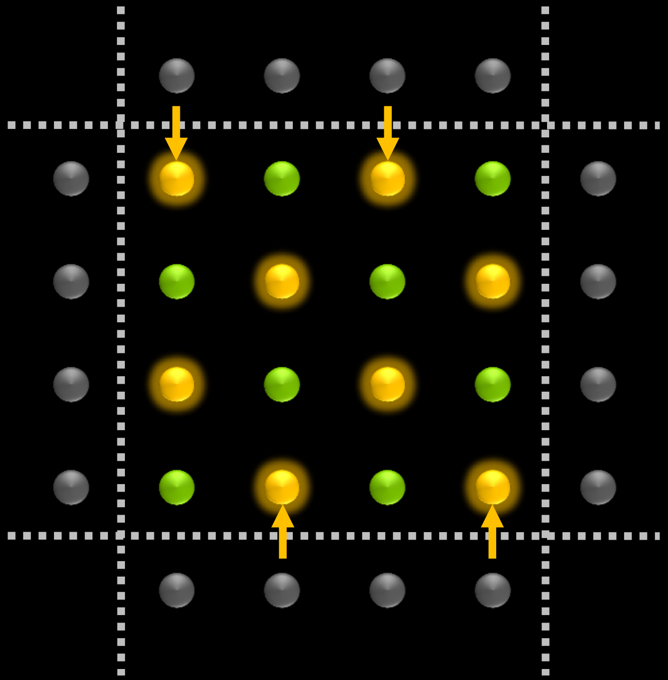
- Step 2:
  - An "interior kernel" updates all sites to the extent possible.
  - Sites along the boundary receive contributions from local neighbors.
  - Finishing off a site requires a local (12x12 complex) matrix-vector multiply. This is done ASAP.

Parallelizing the Dslash



- Step 3:
  - Boundary sites are updated by a series of kernels, one per dimension.
  - Corner sites (and edges/faces) introduce a data dependency between kernels, so we execute them sequentially.
  - A given boundary kernel must also wait for its “ghost zone” to arrive.

Parallelizing the Dslash

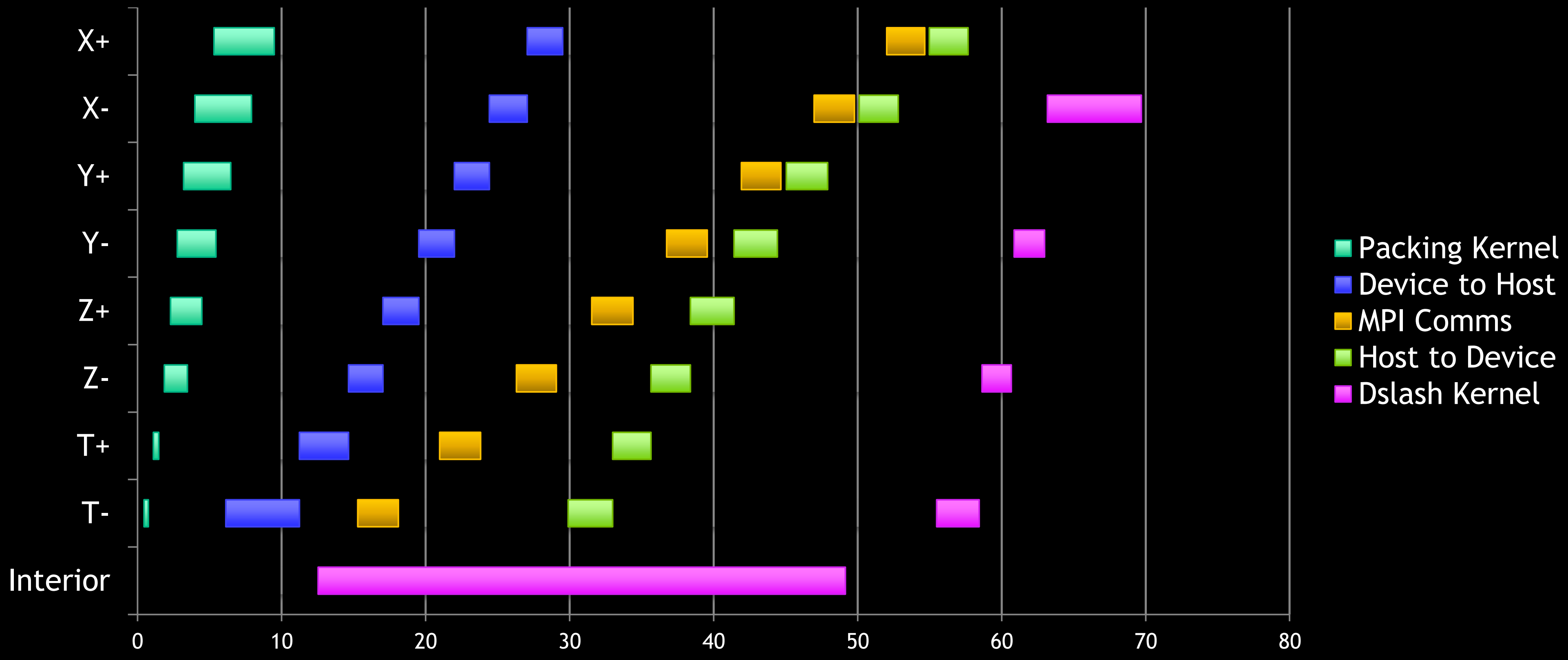


Done!

- Step 3:
  - Boundary sites are updated by a series of kernels, one per dimension.
  - Corner sites (and edges/faces) introduce a data dependency between kernels, so we execute them sequentially.
  - A given boundary kernel must also wait for its “ghost zone” to arrive.

Overlapping comms & compute

- Multi-GPU timings for 4 Tesla C2050 cards in a box.

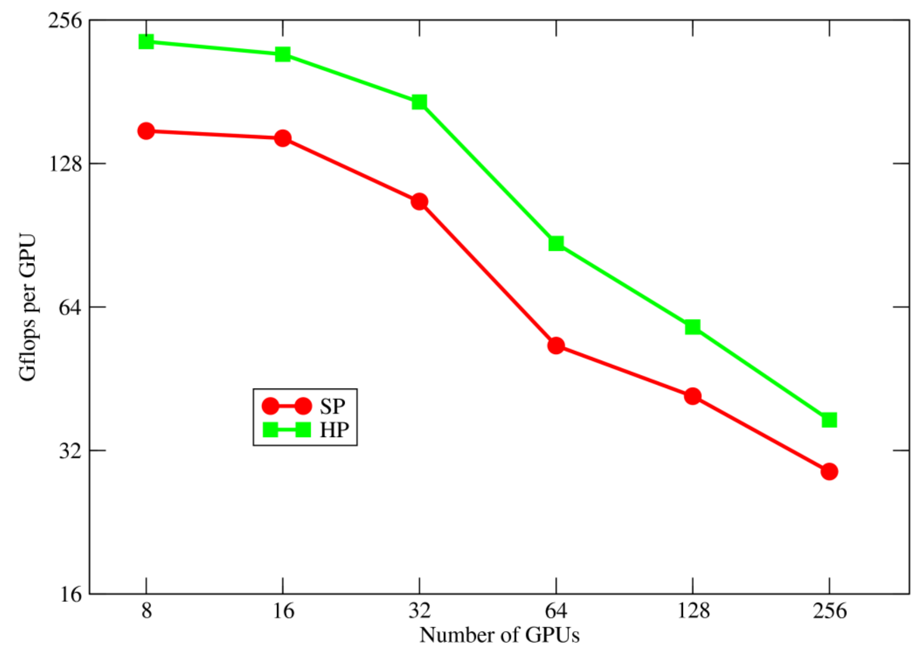


Performance results

- 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 RC1 (but no GPU-Direct)

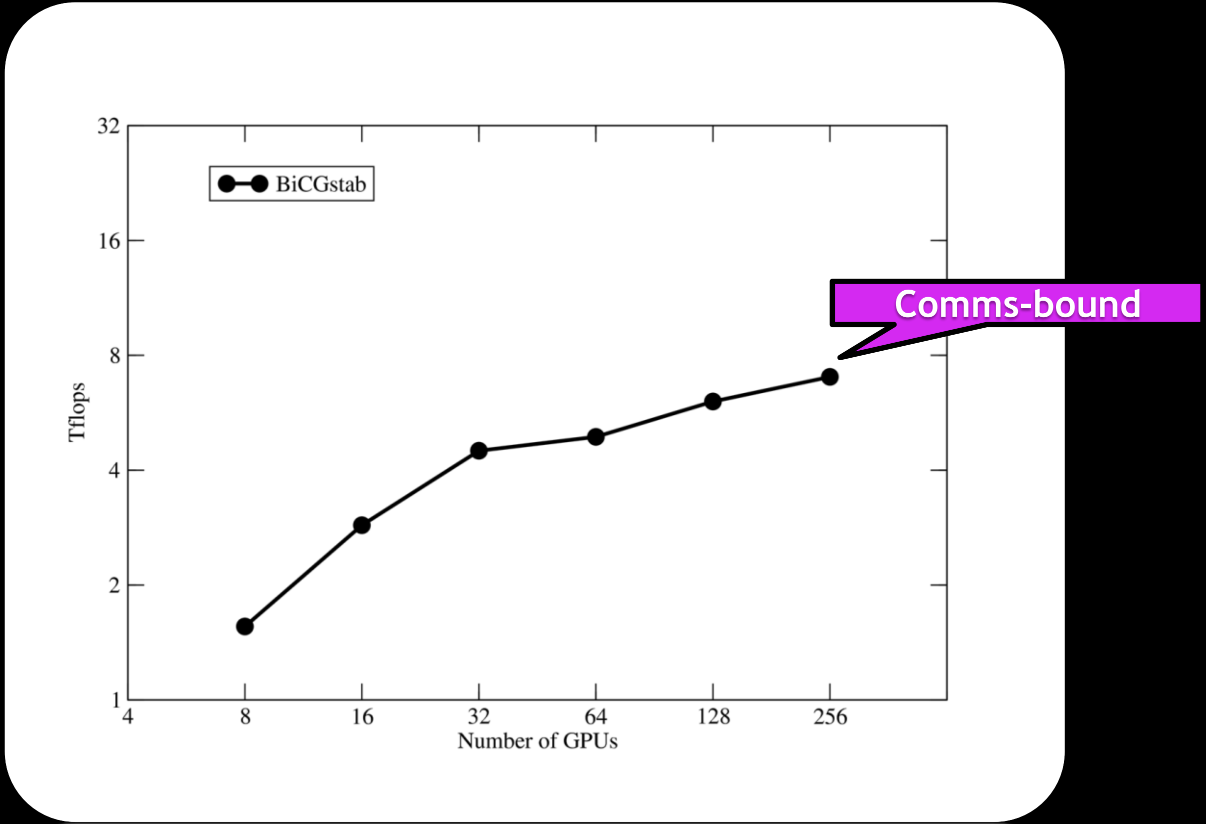
Matrix-vector performance

- Strong-scaling with global volume $32^3 \times 256$



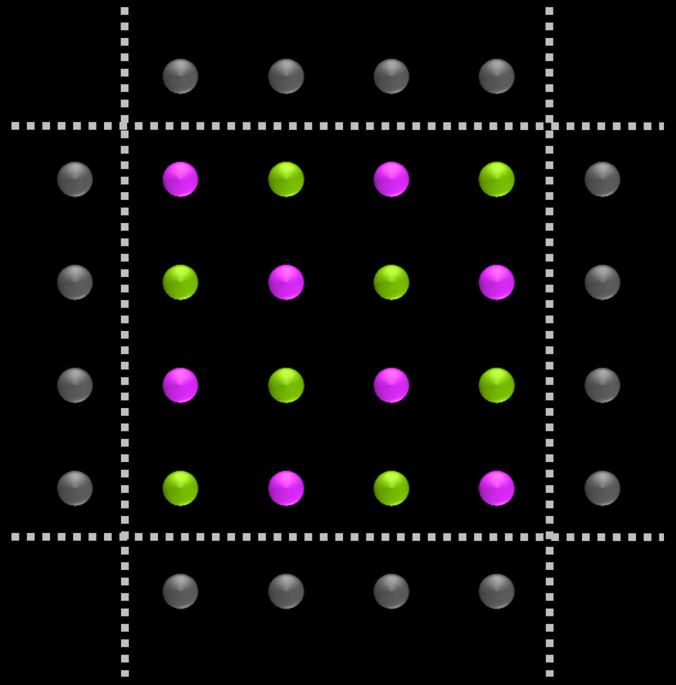
Solver performance

- BiCGstab (mixed single/half) strong scaling, $V = 32^3 \times 256$



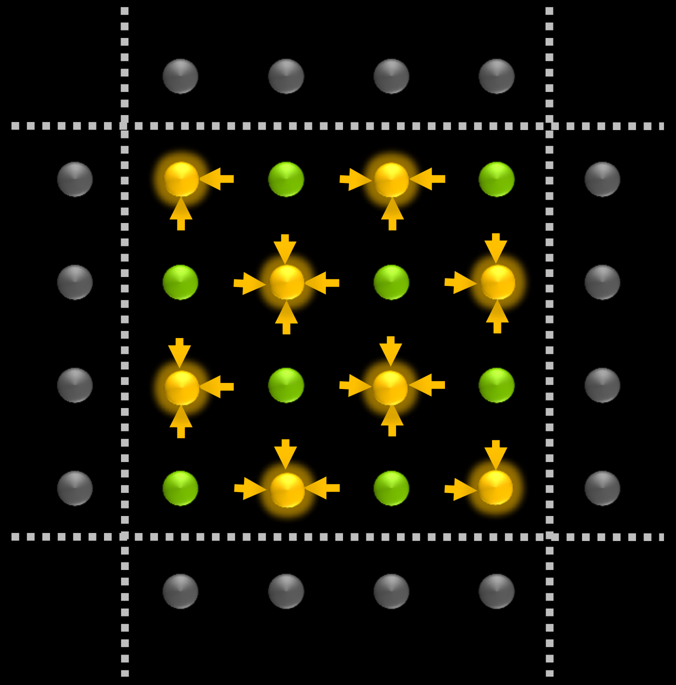
Building a scalable solver

- 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

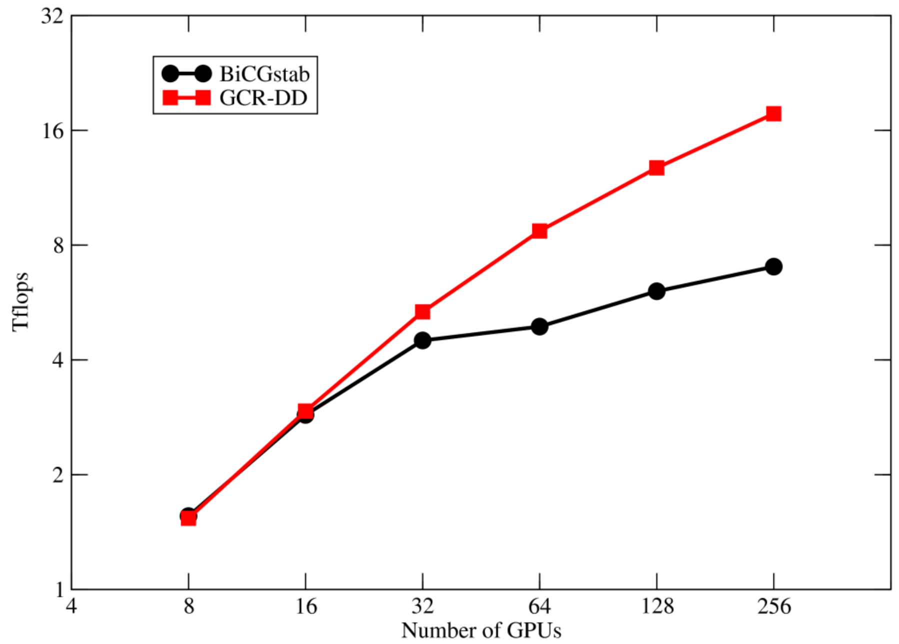
- Inter-GPU communication hurts, so let’s avoid it.
- In the strong-scaling regime, we employ a solver with a domain-decomposed preconditioner.
- Most of the flops go into the preconditioner, where communication is turned off.
- Half precision is perfect here.
- Iteration count goes up, but it’s worth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



Done!

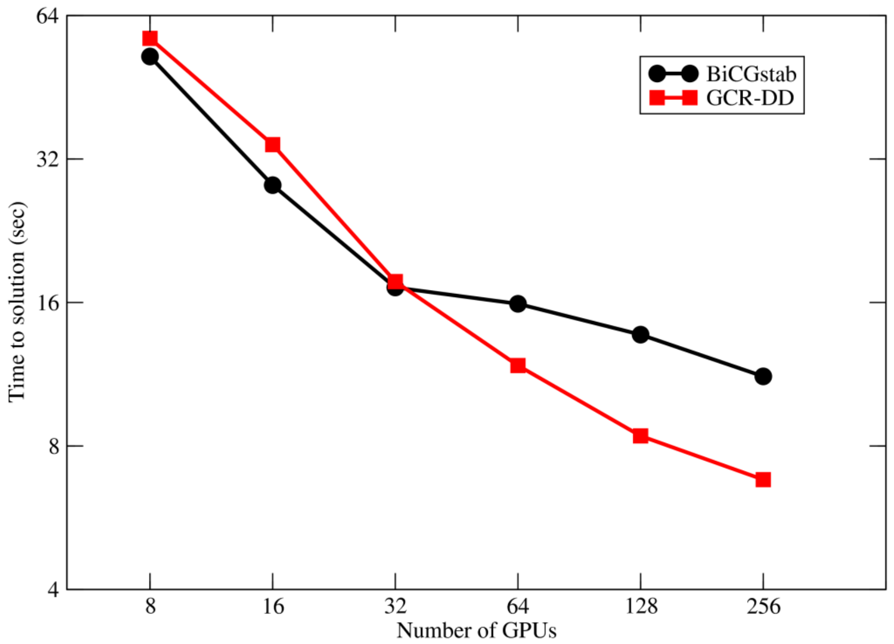
Solver performance (reprise)

- BiCGstab vs. GCR-DD strong scaling, $V = 32^3 \times 256$



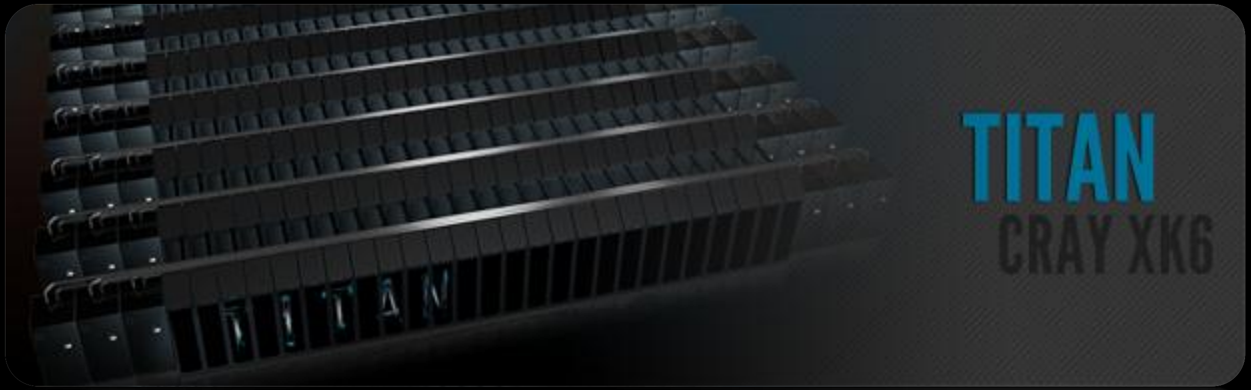
Solver time to solution

- BiCGstab vs. GCR-DD strong scaling, $V = 32^3 \times 256$

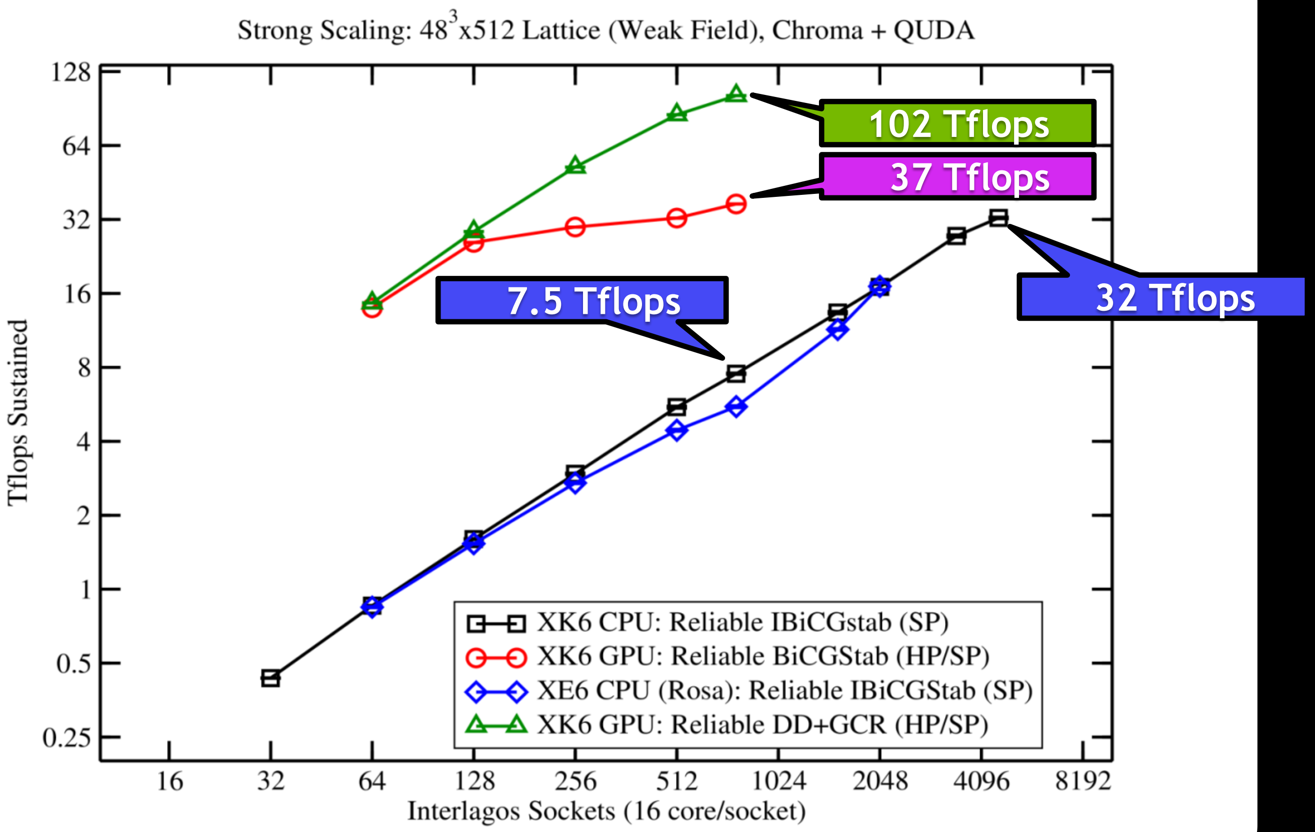


Strong scaling on TitanDev (Cray XK6)

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



Strong scaling on TitanDev (Cray XK6)



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.