
Towards lattice calculations on hundreds of GPUs

Ronald Babich
Boston University &
Pittsburgh Supercomputing Center

Lattice Meets Experiment, Fermilab
October 15, 2011

Overview

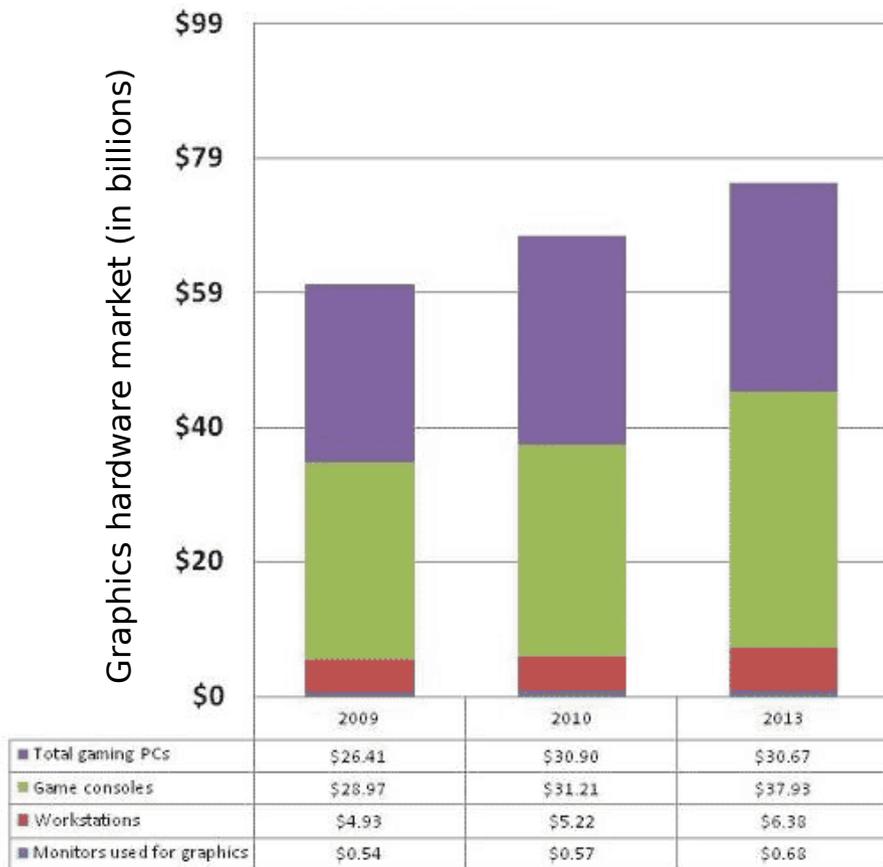
- GPU motivation
- Status of the “QUDA” library
- Strategies and performance
 - Single-GPU / general
 - Multi-GPU
- Outlook: Adaptive geometric multigrid on GPUs

Acknowledgments

- Collaborators and QUDA developers:
 - Kipton Barros (now at LANL)
 - Richard Brower (Boston U.)
 - Michael Clark (Harvard)
 - Justin Foley (Utah)
 - Joel Giedt (Rensselaer)
 - Steven Gottlieb (Indiana)
 - Bálint Joó (Jefferson Lab)
 - James Osborn (Argonne)
 - Claudio Rebbi (Boston U.)
 - Guochun Shi (NCSA)

Graphics processing units (GPUs)

- Video games are driving a multi-billion dollar market for graphics hardware.



(Jon Peddle Research)



GPU computing is leveraging commodity parts, as has long been the case for conventional microprocessors.

Graphics processing units (GPUs)

- We're fortunate that computer graphics is an intrinsically data-parallel problem, like many scientific applications.
- The trend has been toward greater programmability, away from special-purpose hardware.



GPUs for HPC

- It's now fair to say that GPUs are mainstream in high-performance computing.



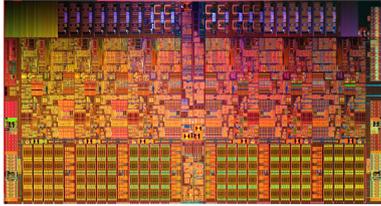
GPUs on the Top 500

- In the latest “Top 500” list (June 2011), three of the top five machines feature GPUs.

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	548352	8162.00	8773.63	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
4	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
5	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 NEC/HP	73278	1192.00	2287.63	1398.61

- But note that rank is determined by the LINPACK benchmark.** For nearly all applications, leveraging so many GPUs in parallel is non-trivial.

A tale of two processors



“Gulftown”

Intel Xeon X5690

6 cores (each with 4-wide SSE unit)

1.17 billion transistors

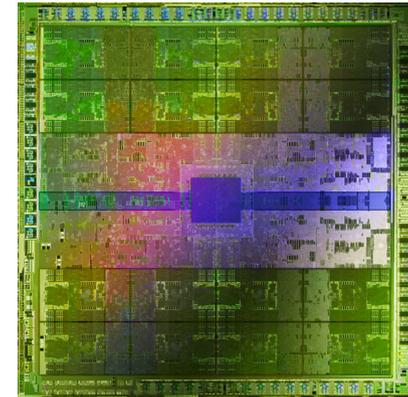
Shared L3 Cache: 12 MB

L1+L2: 6 x (320 KB) = 1920 KB

166 Gflops (SP)

32 GB/s memory bandwidth

up to 288 GB (96 GB is realistic)



“Fermi”

NVIDIA GeForce GTX 480

480 cores

3.0 billion transistors

Shared L2 Cache: 768 KB

L1+SM+Reg: 15 x 192 KB = 2880 KB

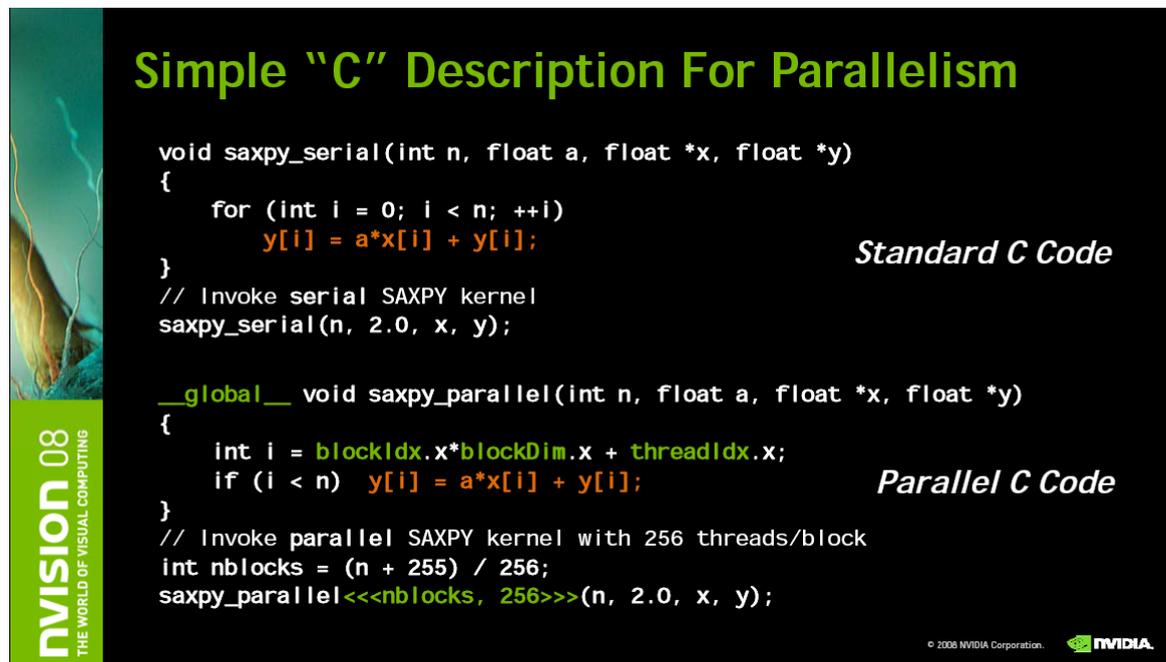
1345 Gflops (SP)

177 GB/s memory bandwidth

1.5 GB (up to 6 GB in Tesla variant)

Programming GPUs

- These days, GPUs are generally programmed by writing functions (“kernels”) to be run on the GPU in C-based languages (**CUDA C/C++** or **OpenCL**). A commercial fortran compiler and layers supporting other languages (e.g., Python, Java) are also available.
- Kernels are executed in parallel, with up to thousands of threads resident at once.



Simple “C” Description For Parallelism

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(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];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

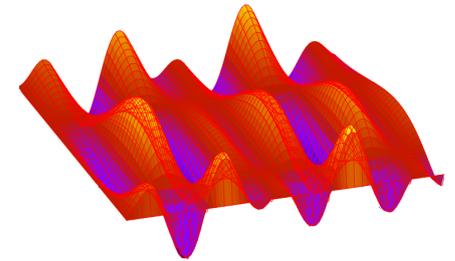
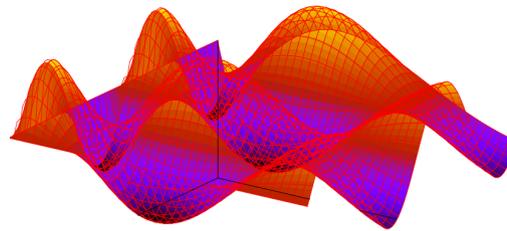
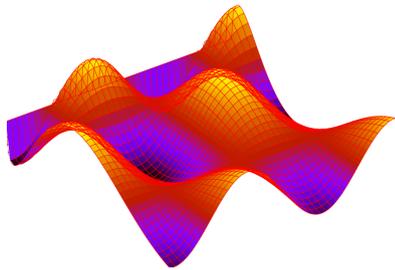
Parallel C Code

VISION 08
THE WORLD OF VISUAL COMPUTING

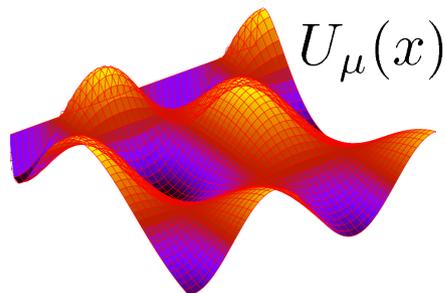
© 2008 NVIDIA Corporation. 

Steps in a lattice calculation

1. Generate an ensemble of gauge field configurations, $\{U_\mu(x)\}$



2. Compute quark propagators in these fixed backgrounds by solving the Dirac equation for various right-hand sides.

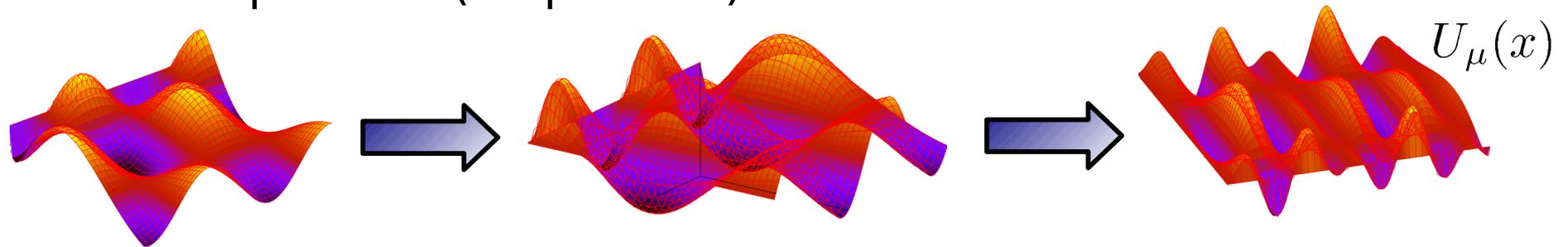


$$D_{ij}^{\alpha\beta}(x, x'; U)\psi_j^\beta(x) = \eta_i^\alpha(x')$$

or “Ax=b”

Configuration generation

- Markov process (sequential)



- Requires $> O(10 \text{ Tflops}) = \text{BlueGene/P, Cray XT5, etc.}$



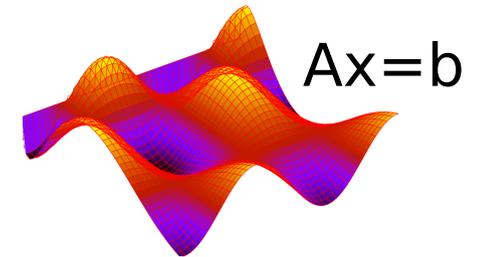
“Intrepid” - Argonne Leadership Computing Facility



“Jaguar” - Oak Ridge Leadership Computing Facility

Computing propagators

- This “analysis” stage is suitable for capacity-type machines but accounts for as many as half the cycles in modern calculations.
- Each job requires tens of cluster nodes . . .



(Clusters dedicated to lattice QCD at Fermilab and Jefferson Lab)

Computing propagators

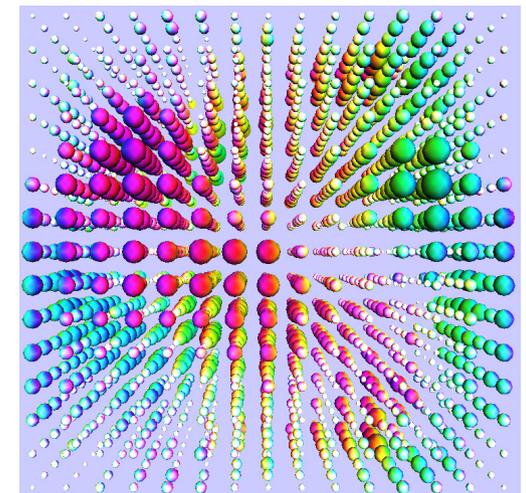
- . . . or a handful of GPUs.



- For smaller lattices, even a single GPU might suffice (**state of the art in 2007-2009**).
- For more typical problems, this *analysis* stage requires $O(10)$ GPUs (**presented at SC'10 last year**).
- Competing with capability machines for *gauge generation* will require the use of $O(100)$ GPUs in parallel (**this talk**).

Krylov solvers

- (Conjugate gradient, BiCGstab, and friends)
- Search for 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
 - ... as well as fast routines for vector addition, inner products, etc. (home-grown “BLAS”)
- **QUDA:** A library for lattice QCD on GPUs
 - <http://lattice.github.com/quda>



QUDA Overview

- “QCD on CUDA” – developed in CUDA C/C++.
- Provides optimized solvers and other routines for the following fermion actions:
 - Wilson and clover-improved Wilson
 - Twisted mass
 - Improved staggered (asqtad/HISQ)
 - Domain wall
- Details, mailing list, and source code repository available here: <http://lattice.github.com/quda>
- Chroma can be built to use the Wilson/clover code with a simple configure flag, likewise for MILC and asqtad/HISQ.
- Straightforward to call directly (e.g., alongside QDP/C)

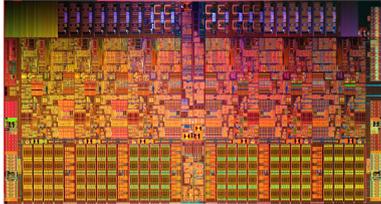
QUDA Capabilities

- **CG** and **BiCGstab** solvers for all actions.
- Wilson, clover, twisted-mass, and improved staggered code includes:
 - **Multi-GPU** support, using either MPI or QMP for communication.
 - **Multi-shift CG** solver.
 - **Domain-decomposed GCR** solver (not proven yet for staggered).
- Improved staggered code also includes:
 - Asqtad link fattening (HISQ in progress).
 - Asqtad fermion force (HISQ in progress).
 - Gauge force for 1-loop improved Symanzik action.

Roadmap / Wish List

- Multi-GPU support for domain wall (in progress)
 - Support for naïve staggered
 - Force terms for actions other than asqtad/HISQ
 - Support for a broader range of gauge groups and representations
 - **Volunteers needed!**
-
- **Note:** The most recent official release (v0.3.2, January 2011) is now pretty stale. QUDA 0.4.0 is coming “any day now.”
 - In the meantime, use the latest development version from the repository: <http://github.com/lattice/quda>

A tale of two processors (reprise)



“Gulftown”

Intel Xeon X5690

6 cores (each with 4-wide SSE unit)

1.17 billion transistors

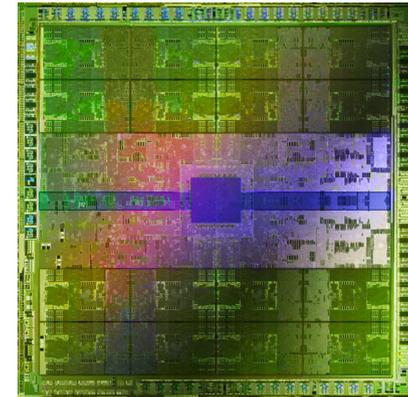
Shared L3 Cache: 12 MB

L1+L2: 6 x (320 KB) = 1920 KB

166 Gflops (SP)

32 GB/s memory bandwidth

up to 288 GB (96 GB is realistic)



“Fermi”

NVIDIA GeForce GTX 480

480 cores

3.0 billion transistors

Shared L2 Cache: 768 KB

L1+SM+Reg: 15 x 192 KB = 2880 KB

1345 Gflops (SP)

177 GB/s memory bandwidth

1.5 GB (up to 6 GB in Tesla variant)

Bandwidth constraints

- Per lattice site, applying the **Wilson-clover** operator (for example) involves 1824 flops while reading/writing 432 floats, corresponding to a **byte/flop ratio of 0.95 in single** precision or **1.90 in double**.
- The basic linear algebra routines are even more memory-bound.
- We're entirely constrained by memory bandwidth. On the GPU, flops are virtually free.

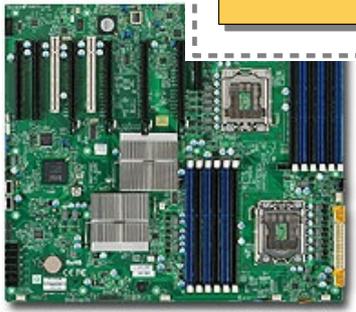
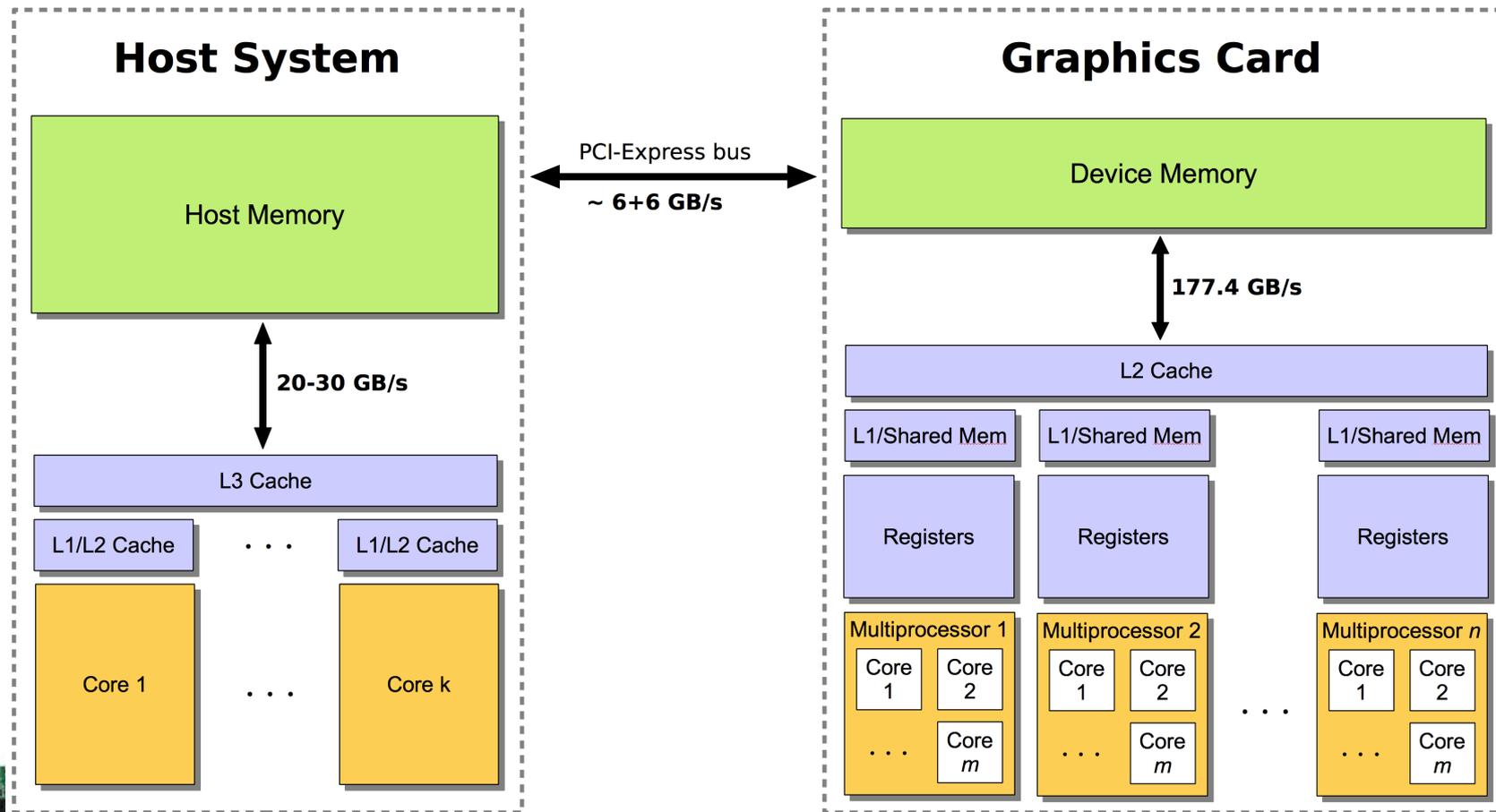
166 Gflops (SP)

32 GB/s memory bandwidth

1345 Gflops (SP)

177 GB/s memory bandwidth

GPU memory hierarchy



(GeForce GTX 480)



Tricks to reduce memory traffic

- Reconstruct SU(3) matrices from 8 or 12 real numbers on the fly, e.g.,

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

P. De Forcrand, D. Lellouch and C. Roiesnel, "Optimizing a lattice QCD simulation program," J. Comput. Phys. 59, 324 (1985).

- Choose a gamma basis with γ_4 diagonal.

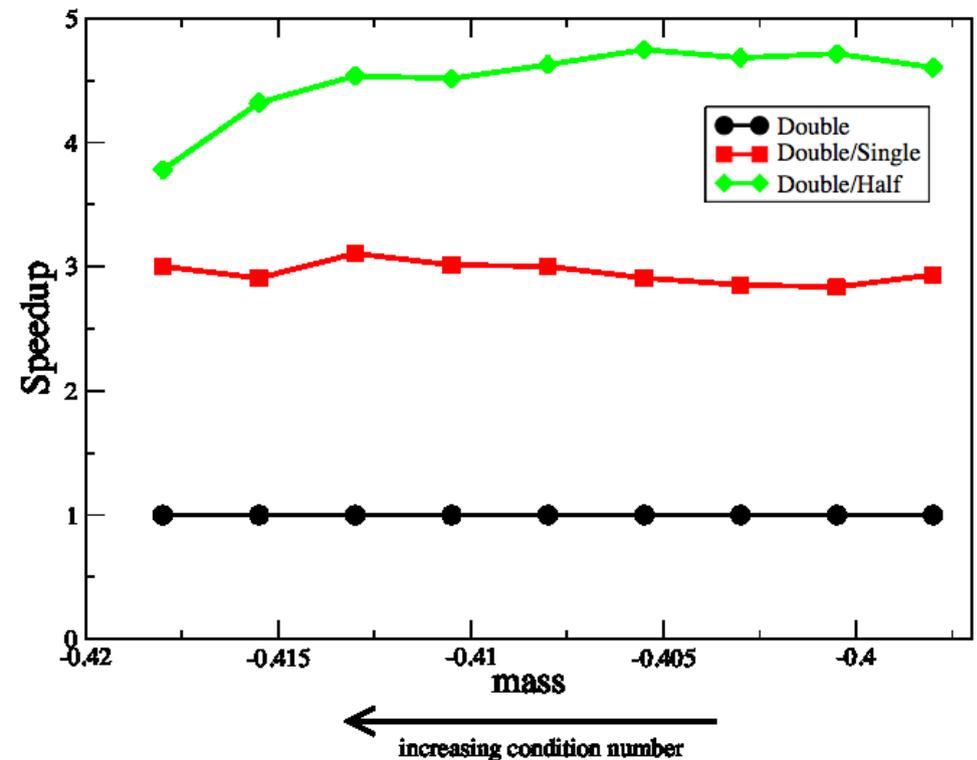
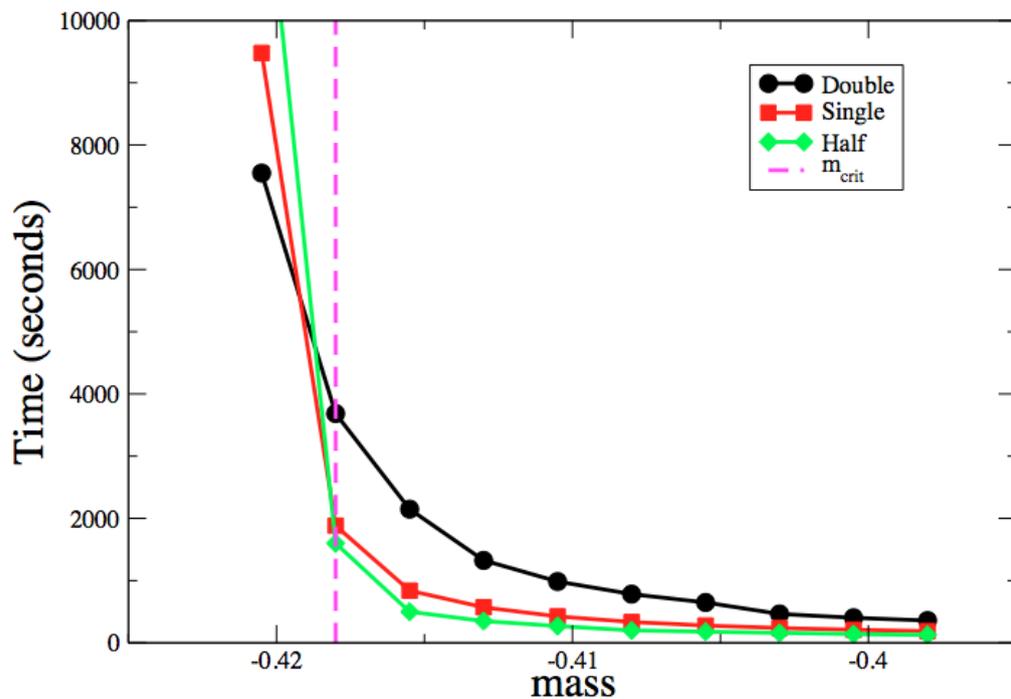
$$P^{\pm 4} = \begin{pmatrix} 1 & 0 & \pm 1 & 0 \\ 0 & 1 & 0 & \pm 1 \\ \pm 1 & 0 & 1 & 0 \\ 0 & \pm 1 & 0 & 1 \end{pmatrix} \longrightarrow \begin{cases} P^{+4} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \\ P^{-4} = \begin{pmatrix} 0 & 0 & 0 & 0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

} similarity transforms on D

- Fix to the temporal gauge (setting gauge links in the t -direction to the identity).

Mixed precision with reliable updates

- Using a mixed-precision solver incorporating “reliable updates” (Clark et al., [arXiv:0911.3191](https://arxiv.org/abs/0911.3191)) with half precision greatly reduces time-to-solution while maintaining double precision accuracy.



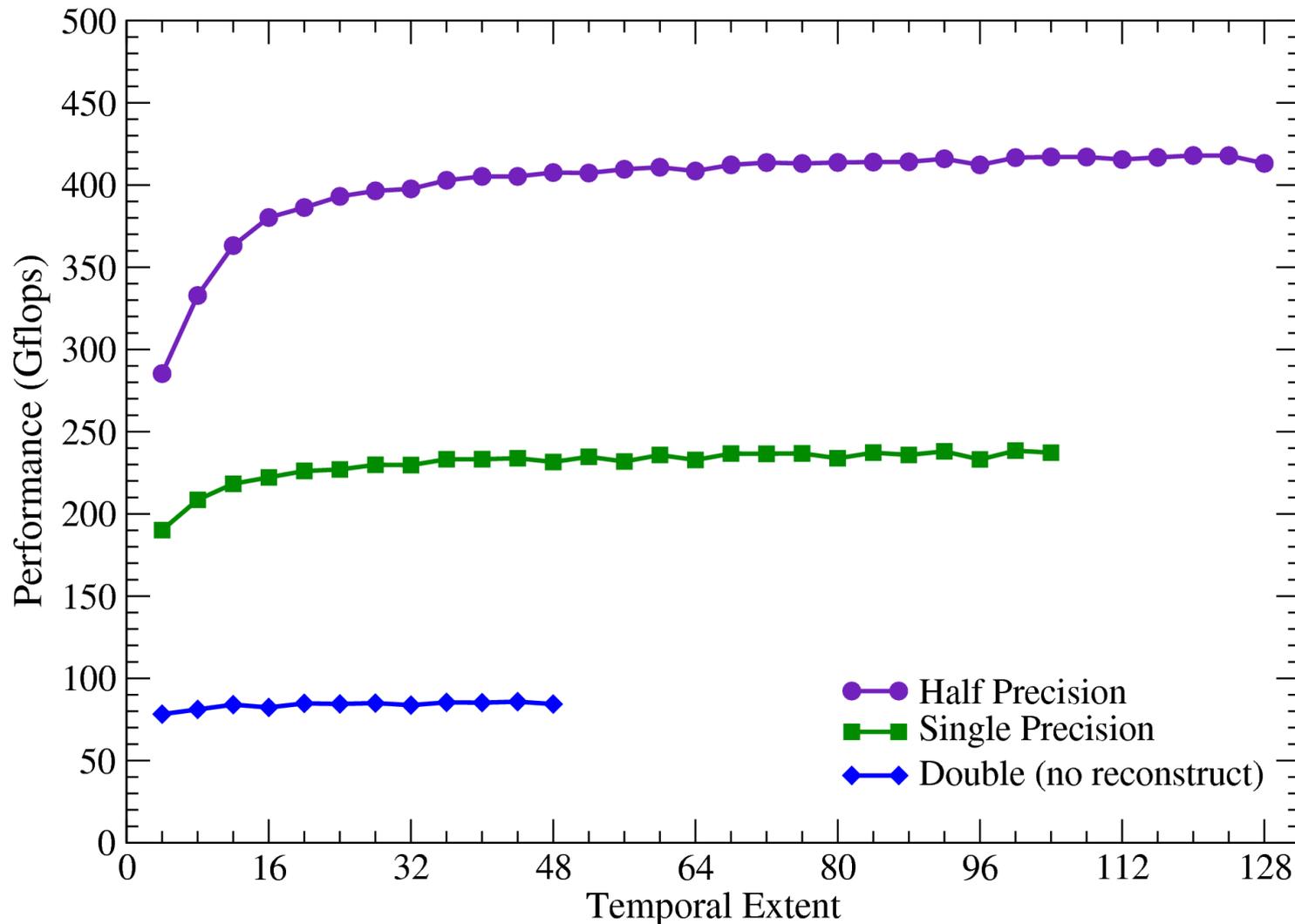
Performance results

- Results are for the even/odd preconditioned clover-improved Wilson matrix-vector product,

$$M = (1 - A_{ee}^{-1} D_{eo} A_{oo}^{-1} D_{oe})$$

- Runs were done on a single GeForce GTX 480.
- For reference, a standard dual-socket node with recent (Westmere) quad-core Xeons would sustain around **20 Gflops** in single precision for a well-optimized Wilson-clover Dslash.
- We'll compare results for double, single, and half precision. In this case, half is a 16-bit quasi-fixed-point implementation, implemented via normalized texture reads.
- The spatial volume is held fixed at 24^3 .

Matrix-vector performance

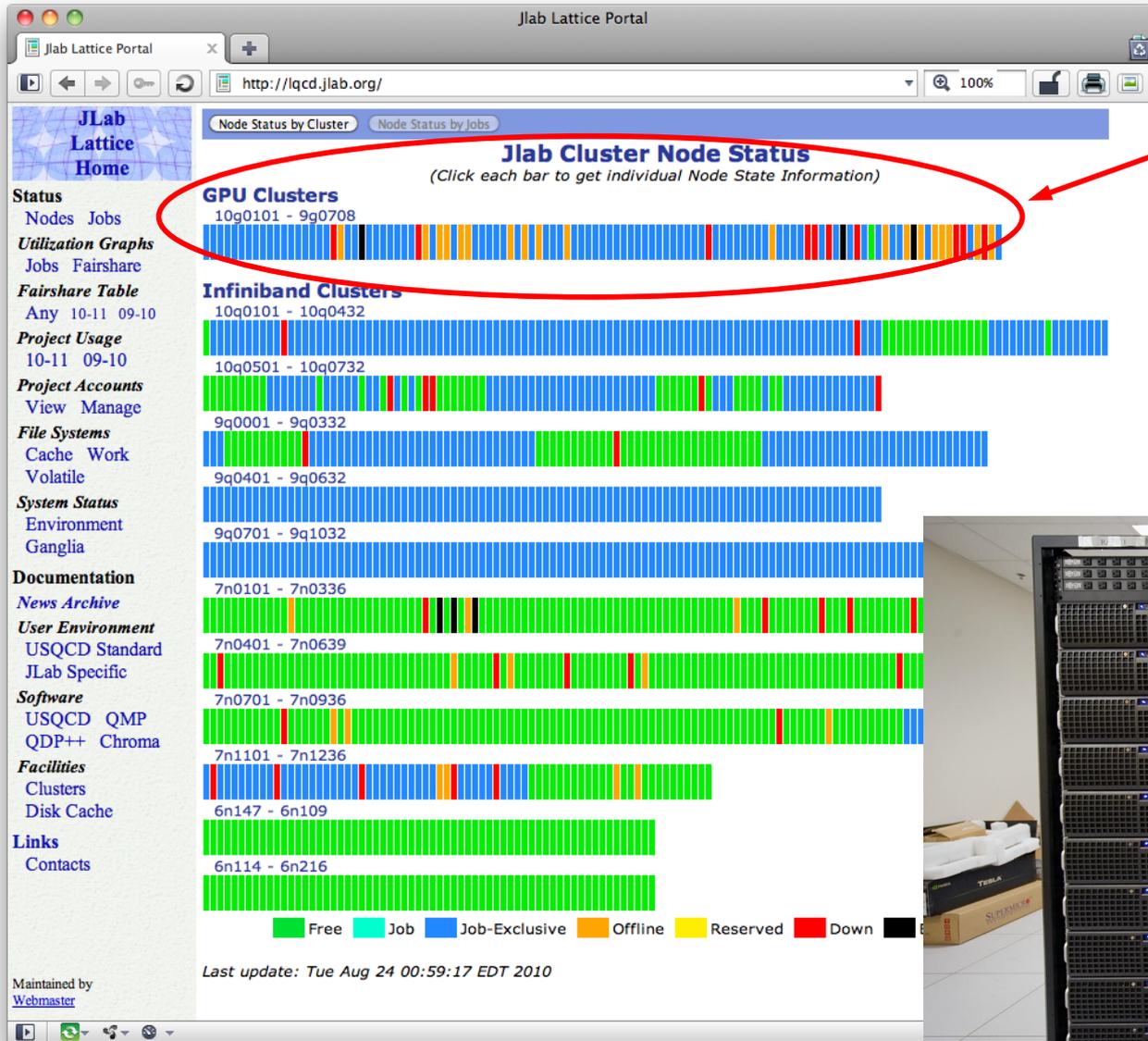


- Single and half performance are about 2.8x and 4.9x higher than double, respectively.

Multi-GPU motivation

- **GPU memory:** For throughput jobs (e.g., computing propagators), it suffices to use the smallest number of GPUs that will fit the job, but often one GPU isn't enough.
- **Host memory:** It's generally most cost-effective to put more than one GPU in a node. These can be used in an embarrassingly parallel fashion (by running multiple separate jobs), but then host memory becomes a constraint.
- **Capability:** We'd like to broaden the range of problems to which GPUs are applicable (e.g., gauge generation).

GPUs are in serious use for “analysis”



~ 500 GPUs dedicated to LQCD at Jefferson Lab



Can they also make a dent here?



"Intrepid" - Argonne Leadership Computing Facility



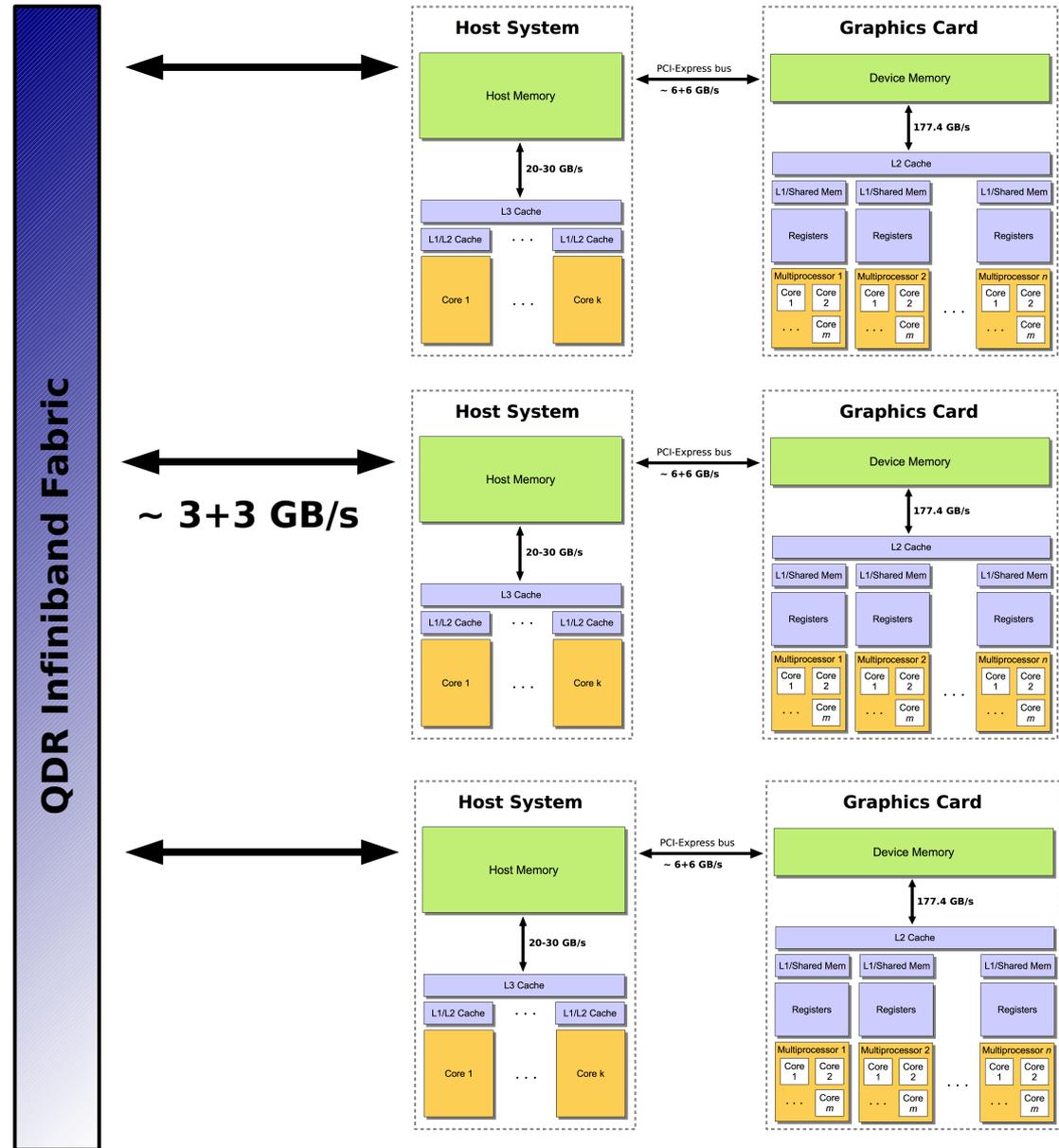
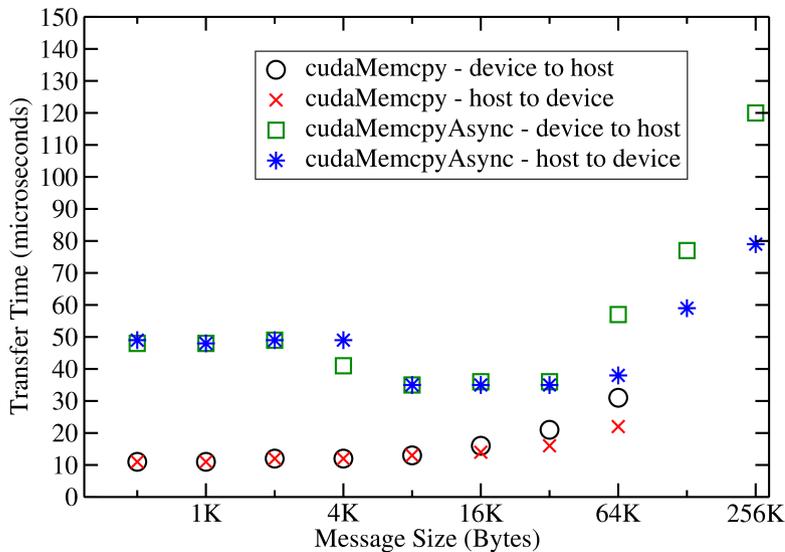
QPACE - NIC Juelich



"Jaguar" - Oak Ridge Leadership Computing Facility

Challenges to scaling up

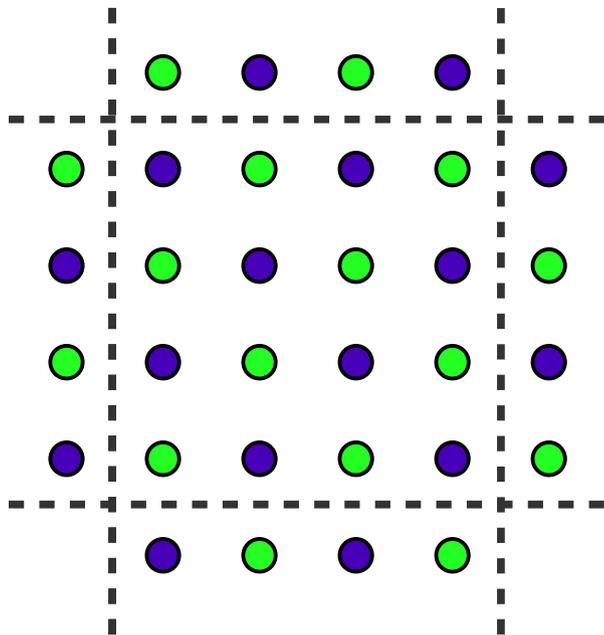
- GPU-to-host and inter-node **bandwidth**
- GPU-to-host and inter-node **latency**



Multi-GPU motivation

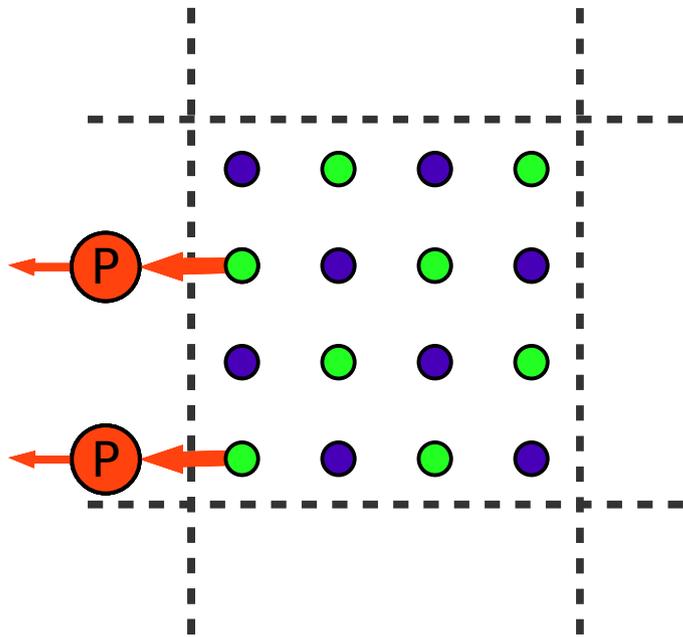
- **GPU memory:** For throughput jobs (e.g., computing propagators), it suffices to use the smallest number of GPUs that will fit the job, but often one GPU isn't enough.
- **Host memory:** It's generally most cost-effective to put more than one GPU in a node. These can be used in an embarrassingly parallel fashion (by running multiple separate jobs), but then host memory becomes a constraint.
- **Capability:** We'd like to broaden the range of problems to which GPUs are applicable (e.g., gauge generation).

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.

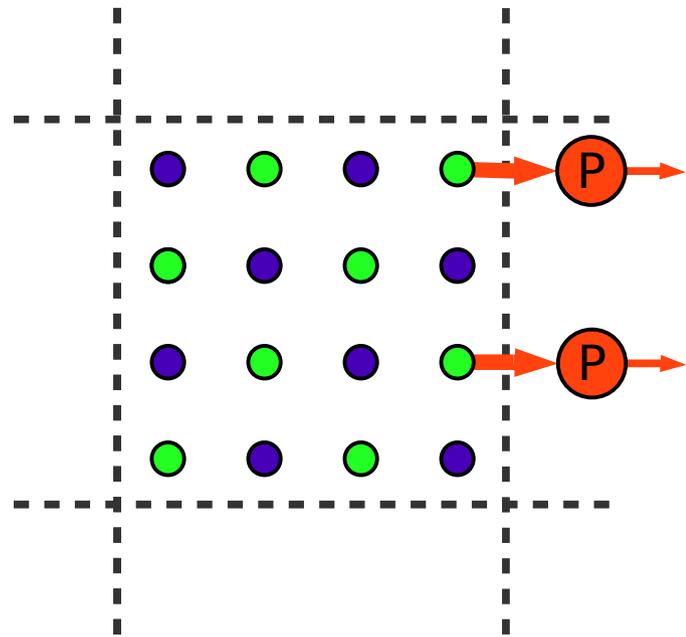
Parallelizing the Dslash



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, a “spin projection” step reduces the amount of data that must be transferred from 24 to 12 floats, at the cost of only 12 adds.

Parallelizing the Dslash

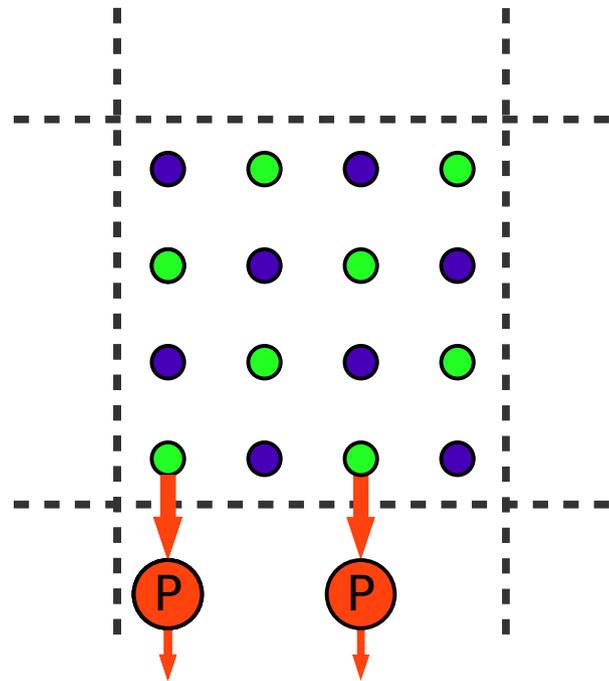


Step 1:

- Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
- As part of the gather kernel, a “spin projection” step reduces the amount of data that must be transferred from 24 to 12 floats, at the cost of only 12 adds.

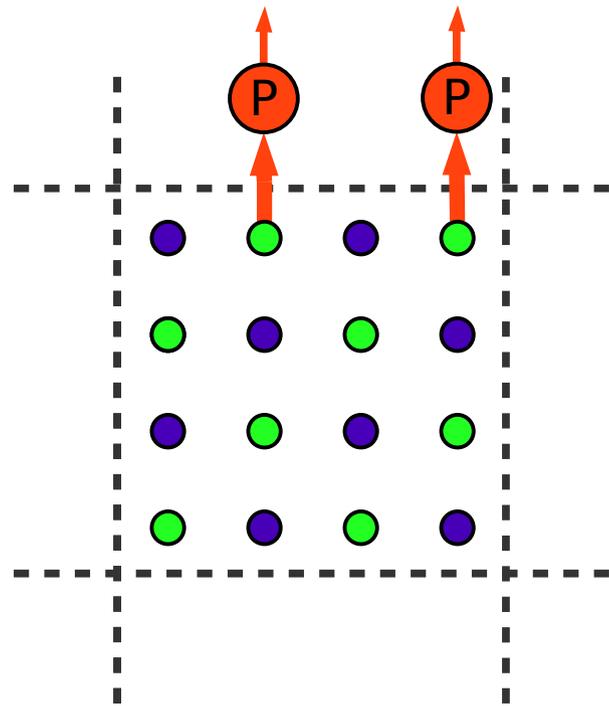
Parallelizing the Dslash

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, a “spin projection” step reduces the amount of data that must be transferred from 24 to 12 floats, at the cost of only 12 adds.

Parallelizing the Dslash



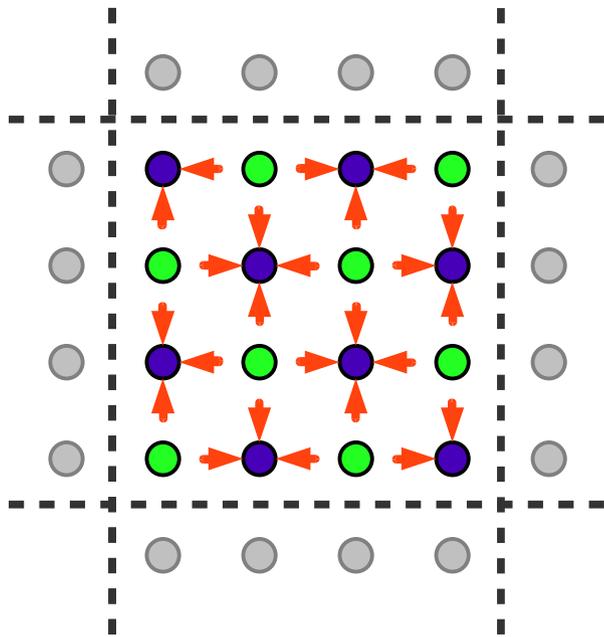
Step 1:

- Gather boundary sites into contiguous buffers to be shipped off to neighboring GPUs, one direction at a time.
- As part of the gather kernel, a “spin projection” step reduces the amount of data that must be transferred from 24 to 12 floats, at the cost of only 12 adds.

Parallelizing the Dslash

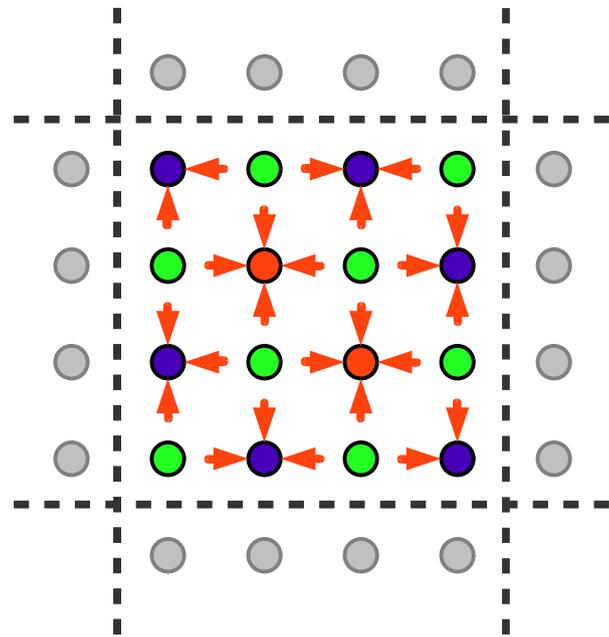
Step 2:

- An “interior kernel” updates all local sites to the extent possible. Sites along the boundary receive contributions from local neighbors.



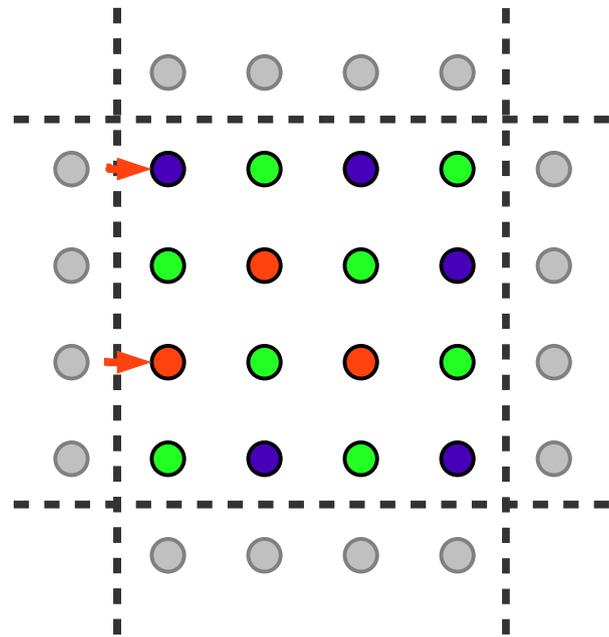
Parallelizing the Dslash

Step 2:



- An “interior kernel” updates all local sites to the extent possible. Sites along the boundary receive contributions from local neighbors.
- To finish off a site, we must apply the clover term (local 12×12 complex matrix-vector multiply). This is done for a given site once contributions from all neighbors have been accumulated.

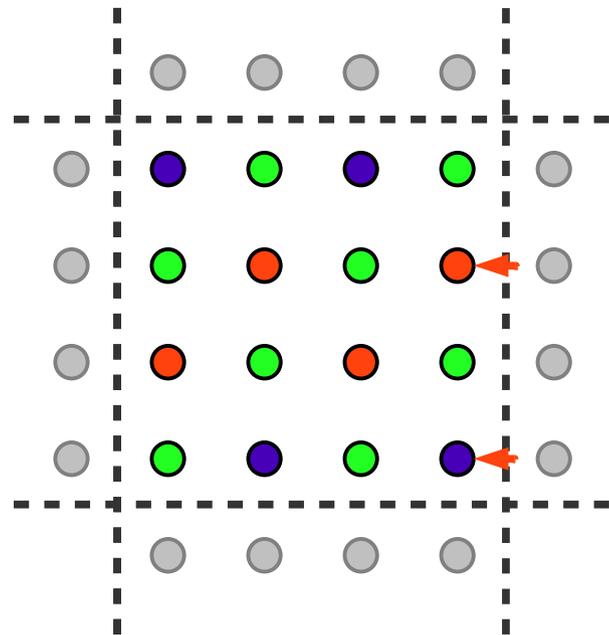
Parallelizing the Dslash



Step 3:

- Boundary sites are updated by a series of kernels (one per direction).
- Note that corner sites (and edges/faces in higher dimensions) introduce a data dependency between kernels, which must therefore execute sequentially.
- A given boundary kernel must also wait for its “ghost zone” to arrive.

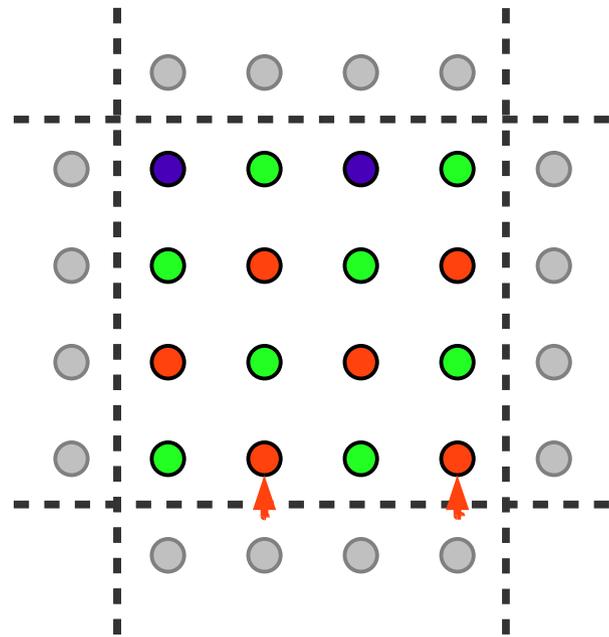
Parallelizing the Dslash



Step 3:

- Boundary sites are updated by a series of kernels (one per direction).
- Note that corner sites (and edges/faces in higher dimensions) introduce a data dependency between kernels, which must therefore execute sequentially.
- A given boundary kernel must also wait for its “ghost zone” to arrive.

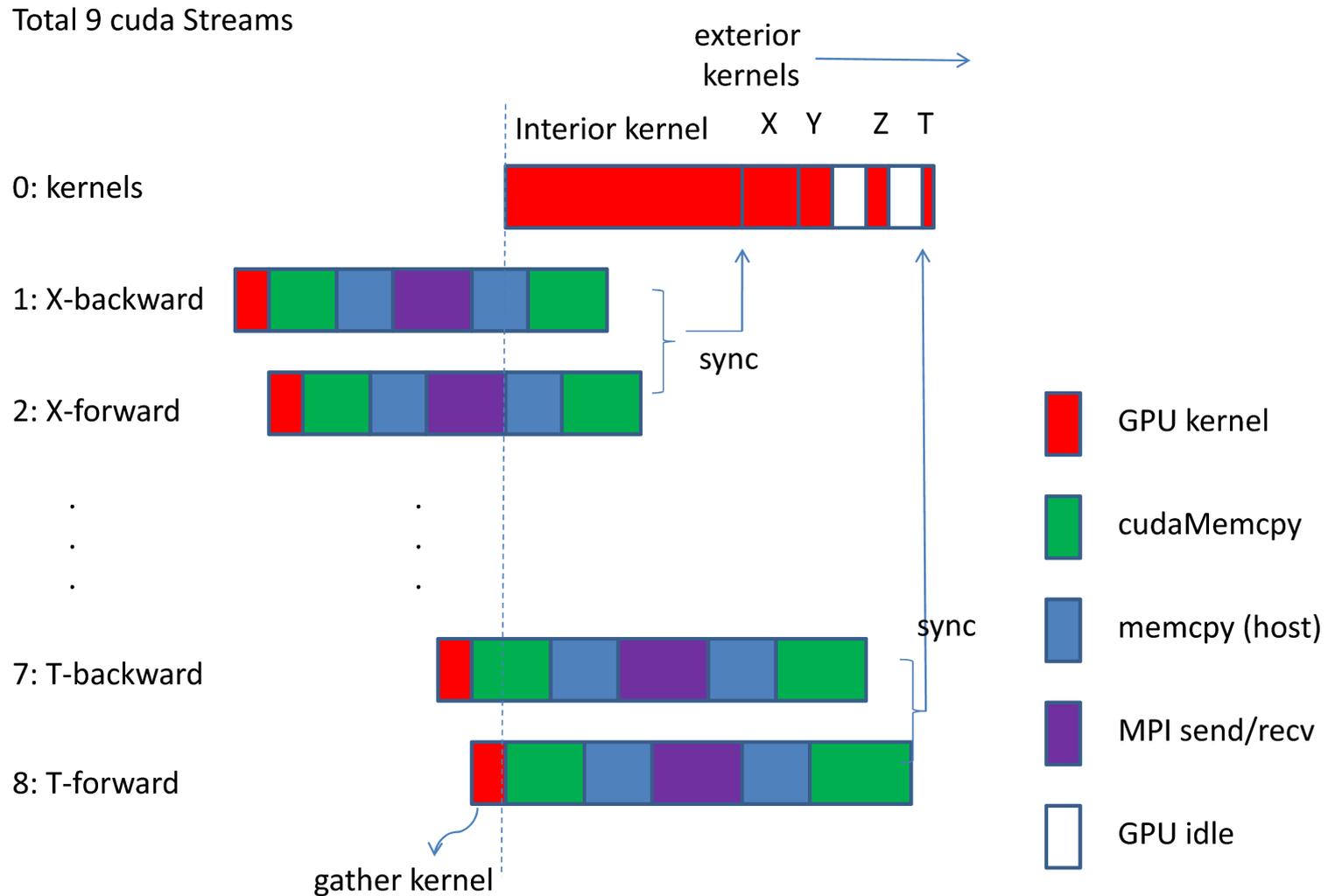
Parallelizing the Dslash



Step 3:

- Boundary sites are updated by a series of kernels (one per direction).
- Note that corner sites (and edges/faces in higher dimensions) introduce a data dependency between kernels, which must therefore execute sequentially.
- A given boundary kernel must also wait for its “ghost zone” to arrive.

Overlapping communications

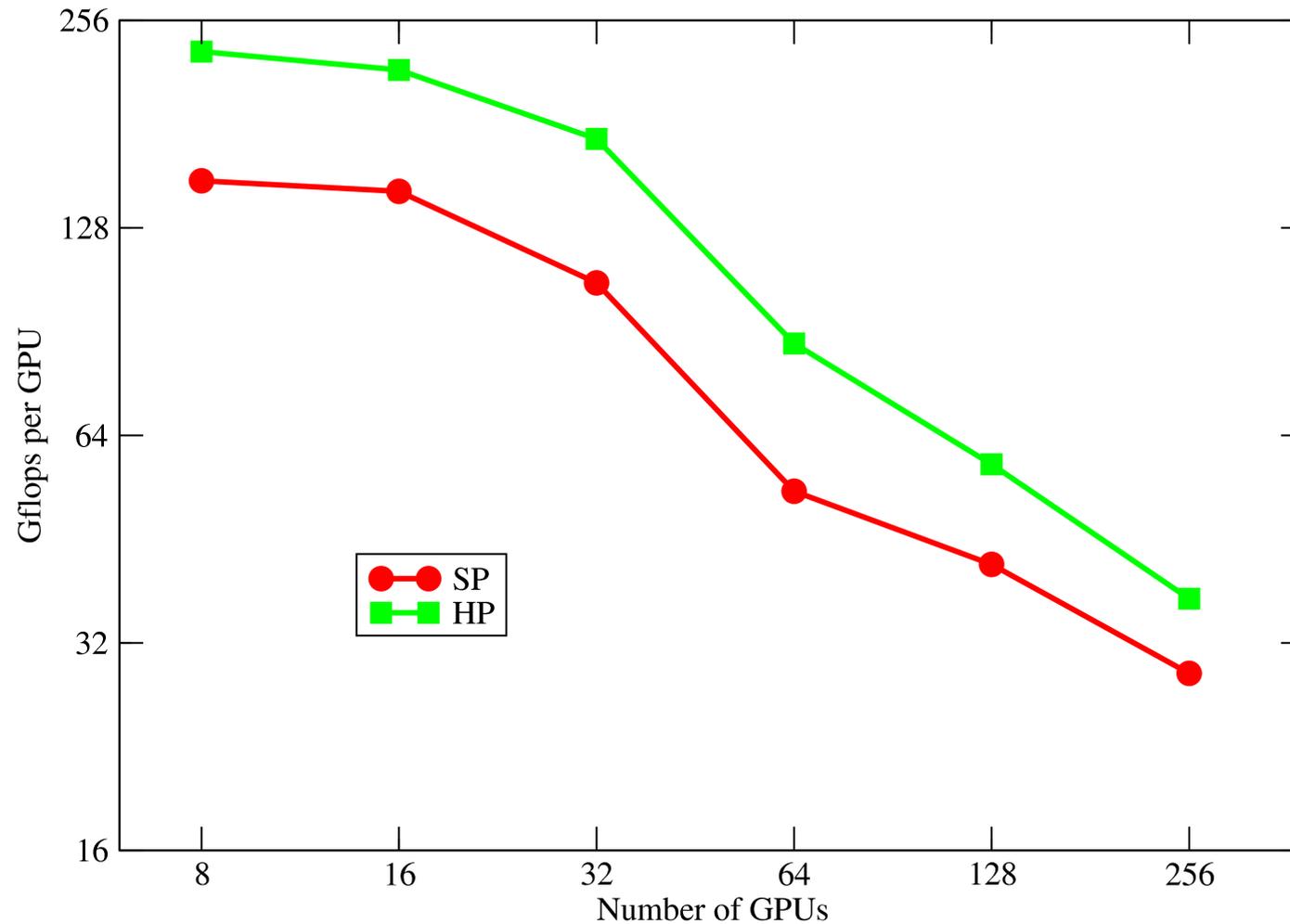


Hardware

- Out test bed is the “Edge” cluster at Lawrence Livermore National Lab:
 - 206 nodes available for batch jobs, interconnected by 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 to the IOH via a switch
 - ECC enabled on the Teslas
 - CUDA 4.0 RC1 (but no GPU-Direct)
 - Driver version 270.27
 - Pre-release version of QUDA 0.4, interfaced to Chroma (an application suite for lattice QCD).

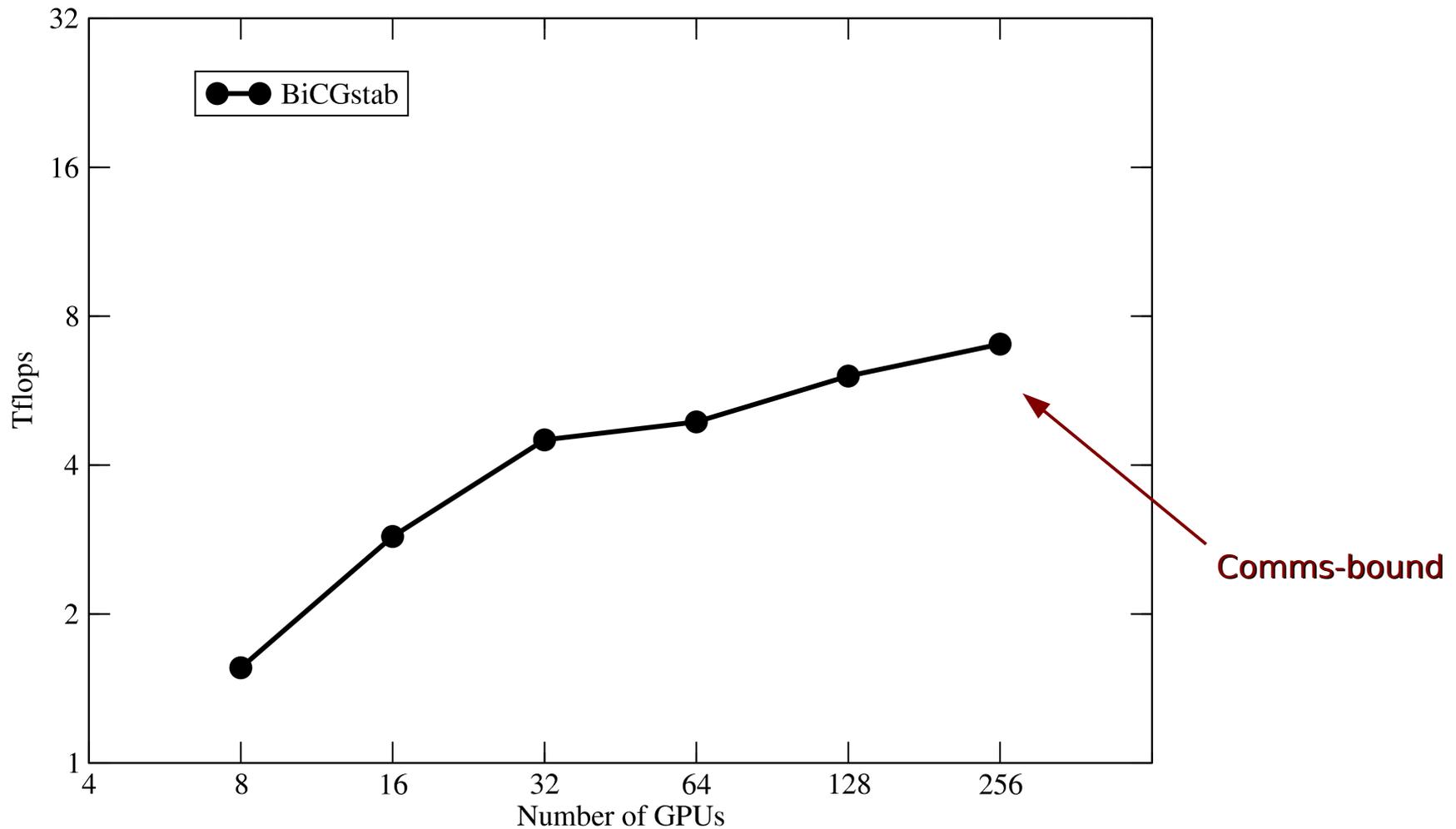
Matrix-vector performance results

$$V = 32^3 \times 256$$



Solver performance

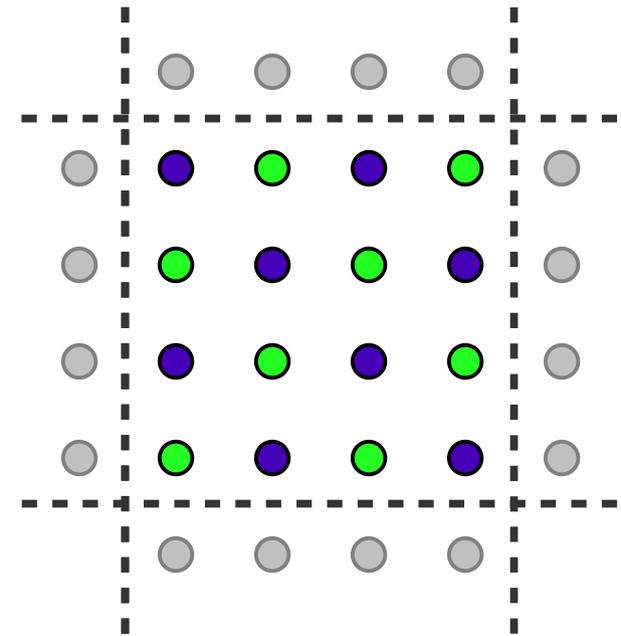
$$V = 32^3 \times 256$$



(BiCGstab, mixed single/half with reliable updates)

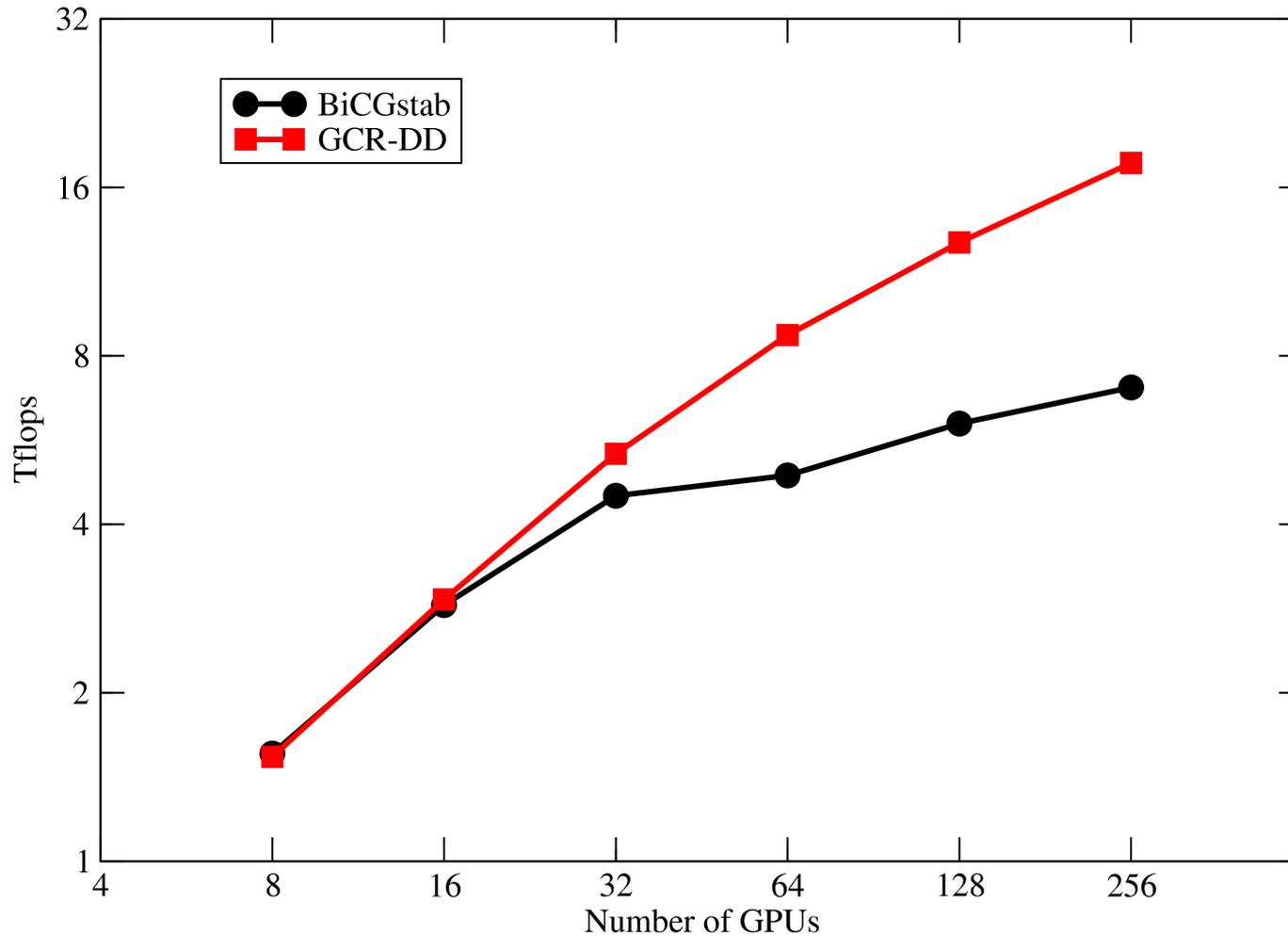
Building a scalable solver

- We need a smarter algorithm, one that takes advantage of the ample compute throughput available while minimizing communication.
- This led us to adopt a domain-decomposition approach by applying an additive Schwarz preconditioner to GCR.
- Most of the work is in the preconditioner, which solves a linear system (to low accuracy via MR) but with Dirichlet boundary conditions between GPUs. In other words, communication is simply turned off in the Dslash.
- Furthermore, this task is well-suited to reduced (e.g., half) precision.



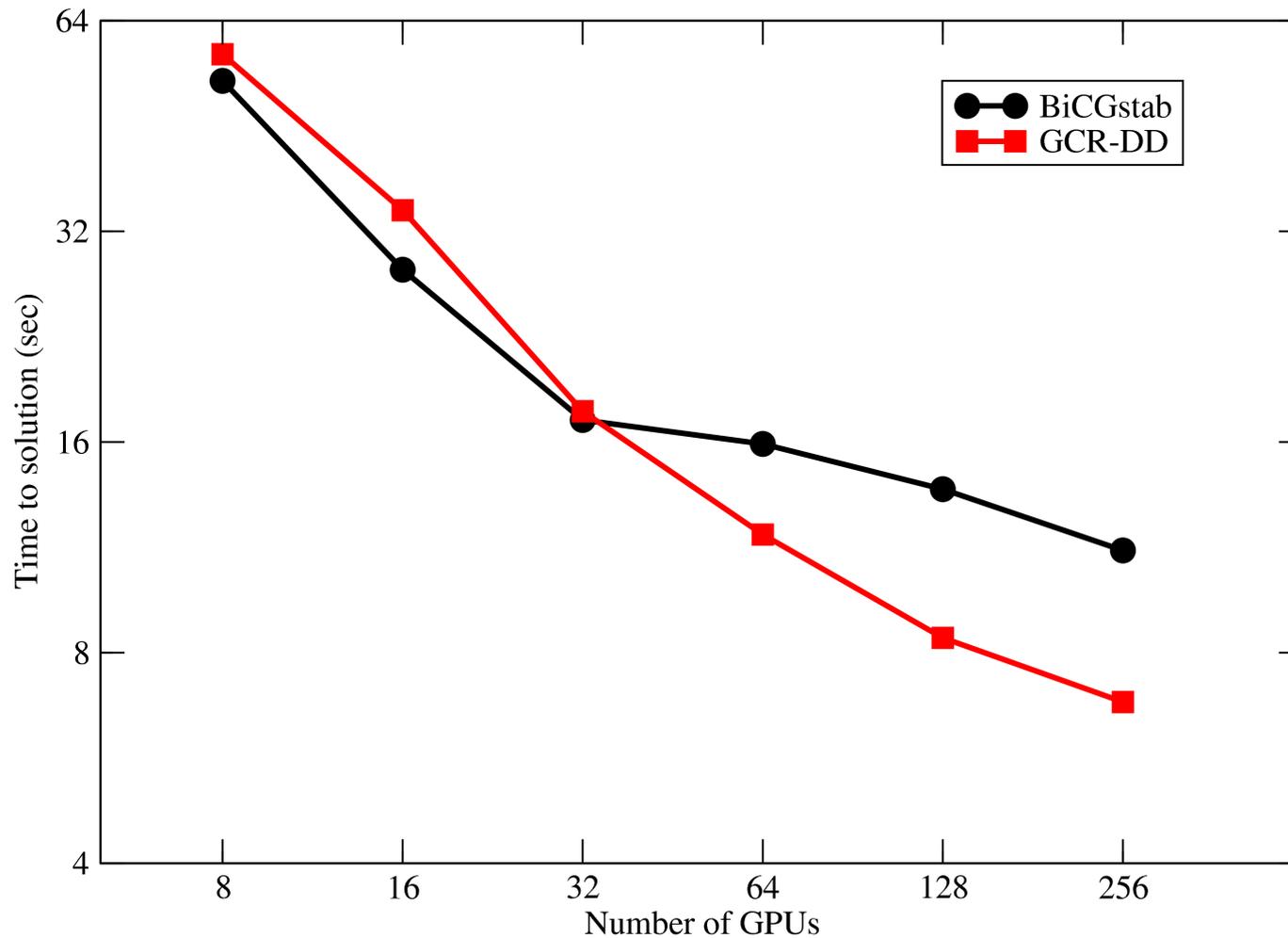
Solver performance

$$V = 32^3 \times 256$$



Solver time to solution

$$V = 32^3 \times 256$$

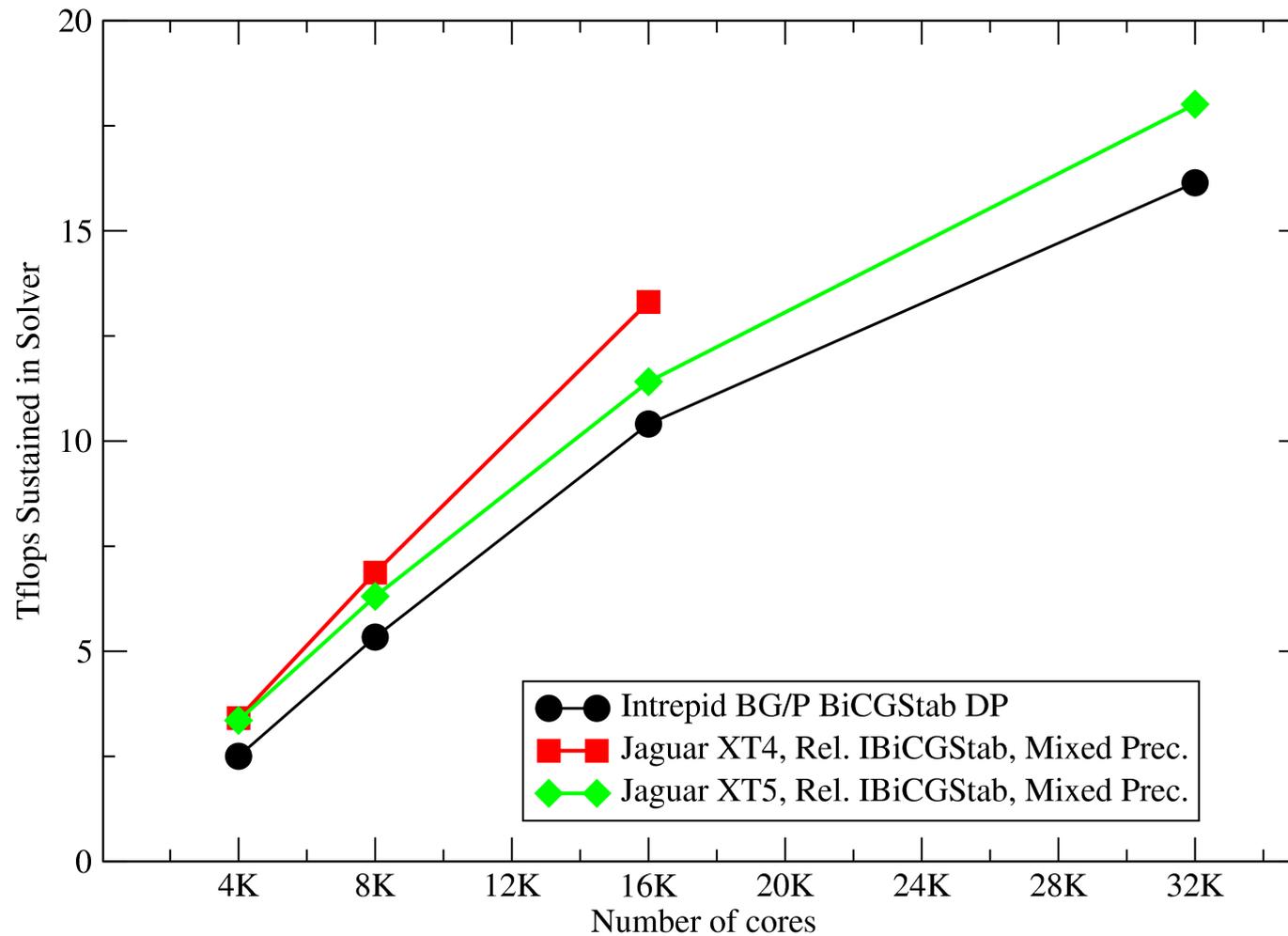


Comparisons

- For a fair comparison, time to solution is the relevant quantity, since algorithms differ in the total number of operations required to reach a given level of accuracy.
- We can define an “effective Tflops” number for GCR-DD, however, in terms of the level of performance that pure single-precision BiCGstab would have to achieve to obtain the same time to solution.
- This yields an effective **9.95 Tflops** for GCR-DD **on 128 GPUs** and **11.5 Tflops on 256 GPUs**.
- This allows us to make rough comparisons to comparable runs on various capability machines:
 - Cray XT4 (Jaguar at Oak Ridge LCF)
 - Cray XT5 (Jaguar PF at Oak Ridge LCF)
 - BlueGene/P (Intrepid at Argonne LCF)

Comparisons

$$V = 32^3 \times 256$$



Multigrid

- A multigrid solver (or preconditioner) works by treating slow-to-converge modes on a succession of coarser grids.

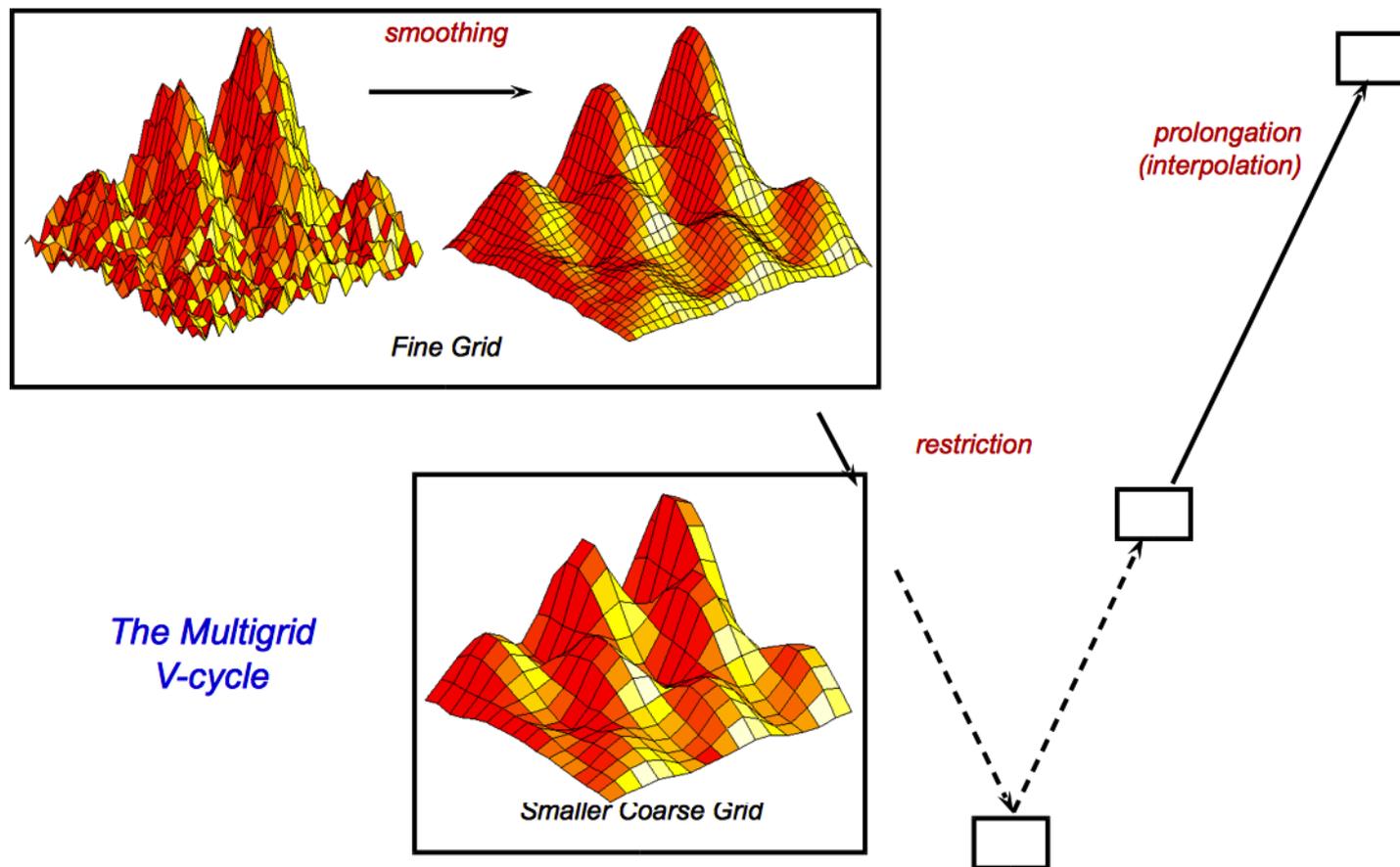
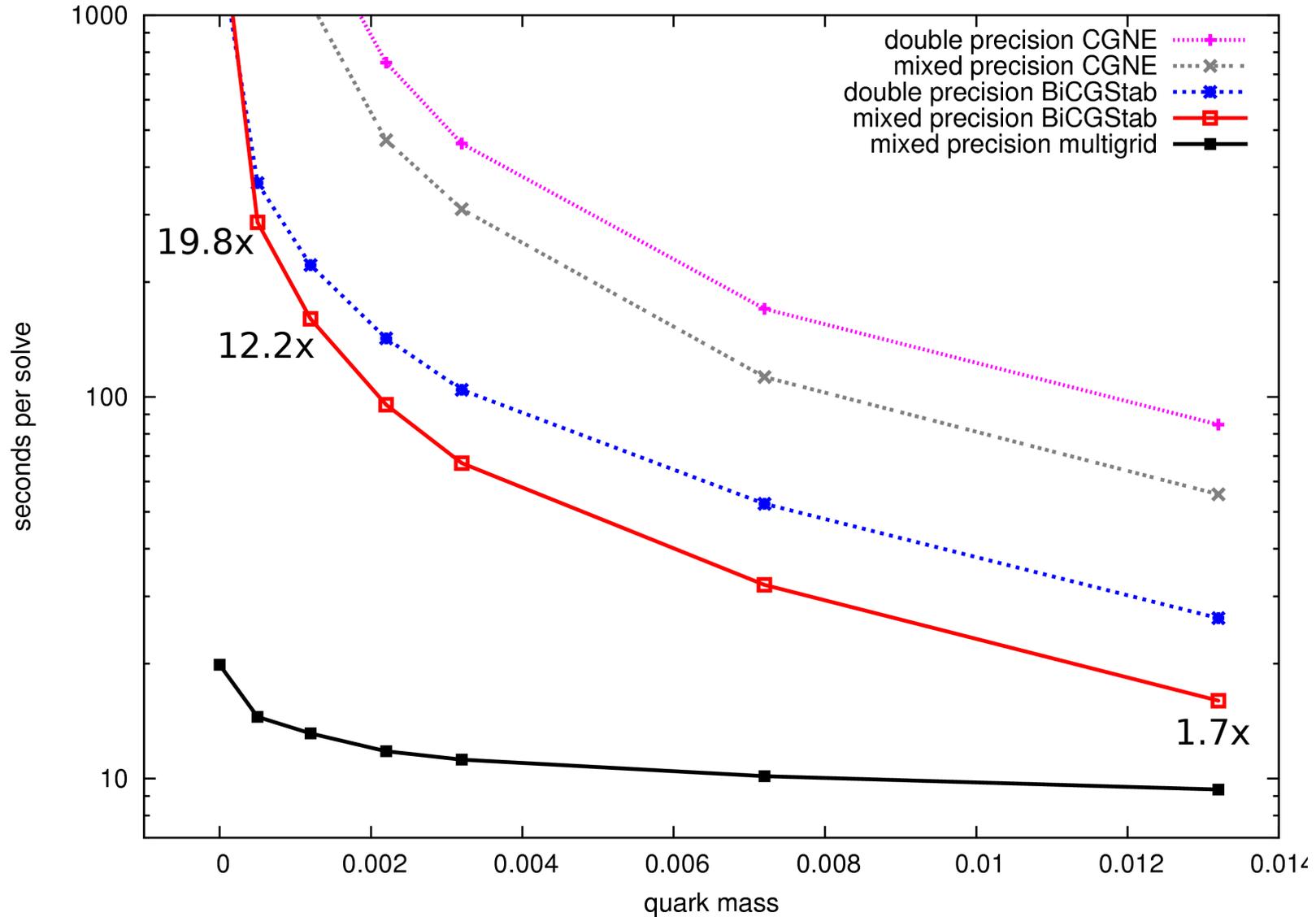


Figure: Rob Falgout (LLNL)

Adaptive geometric multigrid for QCD

- First attempts to apply multigrid to lattice QCD, beginning in the early 90's, mostly came up short.
- Success finally achieved in the last few years by a collaboration of physicists and applied mathematicians.
 - J. Brannick, R. Brower, M. Clark, J. Osborn, C. Rebbi, arXiv:0707.4018
 - R.B., J. Brannick, R. Brower, M. Clark, et al., arXiv:1005.3043
 - J. Osborn, R.B., J. Brannick, R. Brower, M. Clark, S. Cohen, C. Rebbi, arXiv:1011.2775
- “Secret sauce” lies in the construction of the *restriction* and *prolongation* operators that take vectors between grids.
- Optimized code developed for conventional clusters, Blue Genes, etc. (J. Osborn et al.)
- **Results on 1024 cores of BG/P . . .**

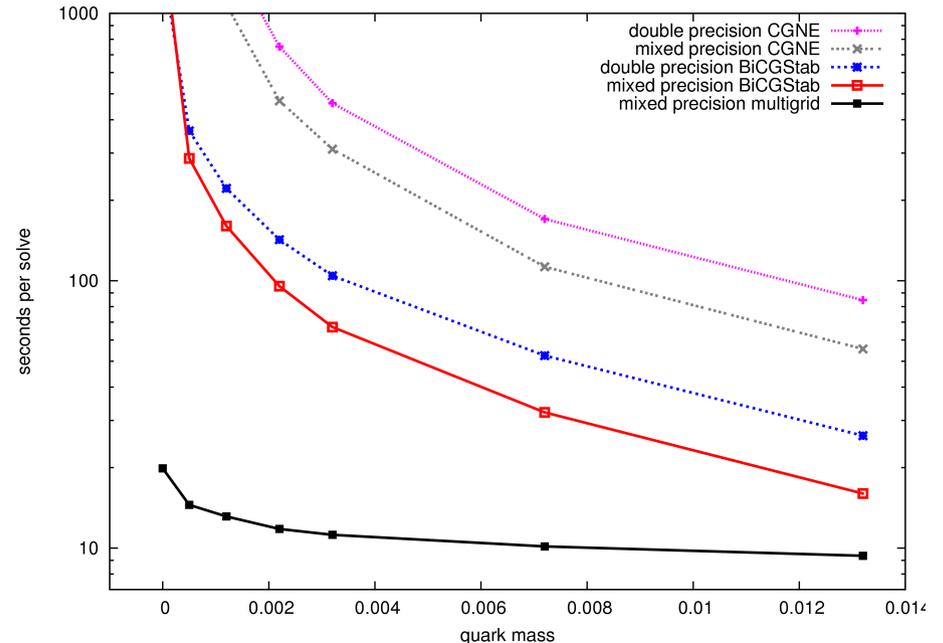
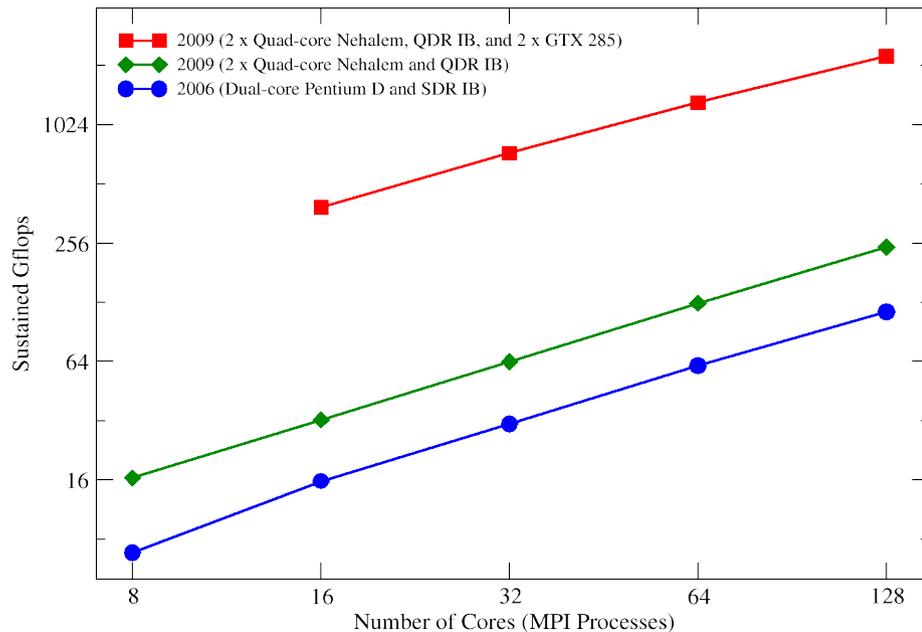
Multigrid results ($32^3 \times 256$)



(Osborn et al.)

Outlook: Multigrid on GPUs

- GPUs clearly win for many workloads in lattice QCD (5-10x improvement in price/performance)
- . . . but multigrid on traditional clusters is competitive (up to 20x over standard solvers at light masses).
- Next step: **MG²**: Multi-GPU multigrid (up to 100x ?).



Multi-GPU x Multigrid = ?

Outlook: Multigrid on GPUs

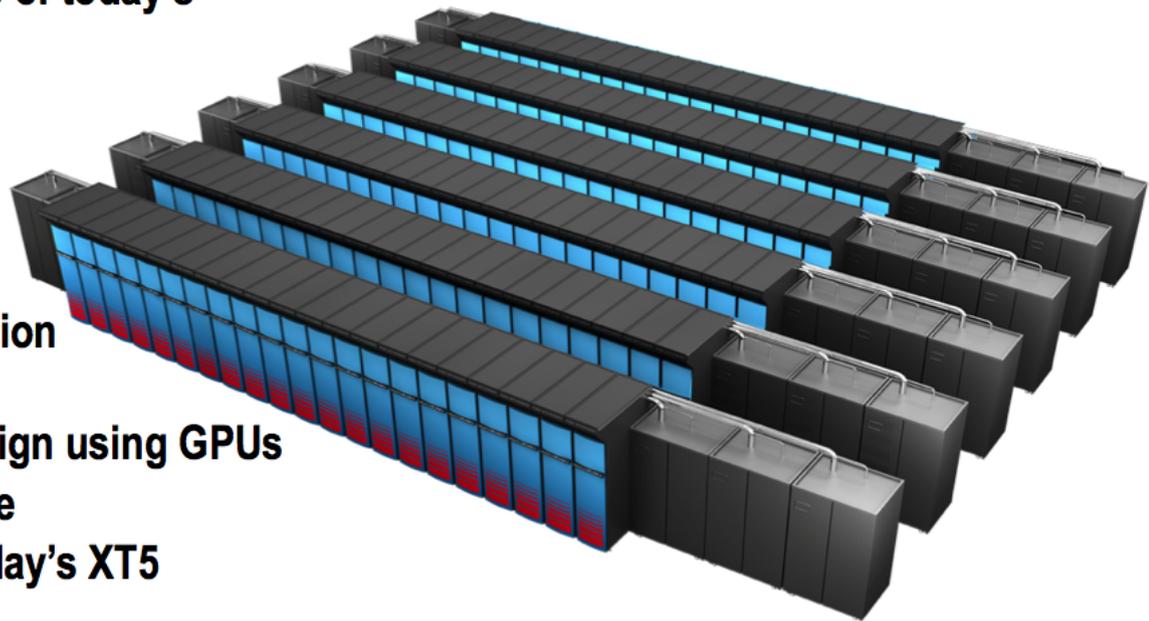
- GPUs clearly win for many workloads in lattice QCD (5-10x improvement in price/performance)
- . . . but multigrid on traditional clusters is competitive (up to 20x over standard solvers at light masses).
- Next step: **MG²**: Multi-GPU multigrid (up to 100x ?).

- Scaling multigrid to hundreds of GPUs will be a challenge, and employing multi-scale methods in gauge generation is an active research topic.
- *but* multigrid solvers will almost certainly work well on tens of GPUs (at least for Wilson/clover).
- **What would you do differently if propagators were 50x cheaper to compute?**

Outlook: Big machines are on the way...

ORNL's "Titan" 20 PF System Goals

- Designed for science from the ground up
- Operating system upgrade of today's Linux Operating System
- Gemini interconnect
 - 3-D Torus
 - Globally addressable memory
 - Advanced synchronization features
- New accelerated node design using GPUs
- 10-20 PF peak performance
 - 9x performance of today's XT5
- Larger memory
- 3x larger and 4x faster file system



Outlook: Big machines are on the way...

Cray XK6 Compute Node

Cray
THE SUPERCOMPUTER COMPANY

XK6 Compute Node Characteristics
AMD Opteron 6200 Interlagos 16 core processor
Tesla X2090 @ 665 GF
Host memory 16 or 32GB 1600 MHz DDR3
Tesla X090 memory 6GB GDDR5 capacity
Gemini high speed Interconnect
Upgradeable to NVIDIA's Kepler many-core processor

The diagram illustrates the internal architecture of a Cray XK6 Compute Node. A central blue interconnect hub, labeled 'CRAY' and 'HT3', connects various components. On the left, an NVIDIA GPU is connected to an AMD Opteron processor via an HT3 interconnect. On the right, an AMD Opteron processor is connected to an NVIDIA GPU via a PCIe Gen2 interconnect. A yellow arrow points from the table of characteristics to the diagram. A 3D coordinate system (X, Y, Z) is shown in the bottom right corner.

Bonus slides

References

- K. Barros, R. Babich, R. Brower, M. A. Clark, and C. Rebbi, “Blasting through lattice calculations using CUDA,” PoS(LATTICE2008) 045 (2008) [arXiv:0810.5365 [hep-lat]].
- M. A. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi, "Solving Lattice QCD systems of equations using mixed precision solvers on GPUs," Comput. Phys. Commun. 181, 1517 (2010) [arXiv:0911.3191 [hep-lat]].
- R. Babich, M. A. Clark, and B. Joó, “Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics,” SC'10.
- G. Shi, S. Gottlieb, A. Torok, and V. Kindratenko, "Accelerating Quantum Chromodynamics Calculations with GPUs," proceedings of SAAHPC'10. [*generalization of the above to staggered fermions*]
- R. Babich, M. A. Clark, B. Joó, G. Shi, R. Brower, and S. Gottlieb, “Scaling lattice QCD beyond 100 GPUs,” SC'11.
- **Prior and contemporary work:** Wuppertal/Budapest group (G.I. Egri et al., 2007), National Taiwan U. (T.W. Chiu et al.), George Washington U. (A. Alexandru et al.), Seoul National U. (W. Lee et al.), Pisa U. (G. Cossu et al.), Hiroshima/Nagoya/KEK, others.

Mixed precision with reliable updates

- In the usual method of *iterative refinement* (or “*defect correction*”), the Krylov subspace is thrown away at every restart:

$$r_0 = b - Ax_0;$$

$$k = 0;$$

while $\|r_k\| > \epsilon$ **do**

 Solve $Ap_{k+1} = r_k$ to precision ϵ^{in} ;

$$x_{k+1} = x_k + p_{k+1};$$

$$r_{k+1} = b - Ax_{k+1};$$

$$k = k + 1;$$

end

- An alternative is “*reliable updates*,” originally introduced to combat residual drift caused by the erratic convergence of BiCGstab: G. L. G. Sleijpen, and H. A. van der Vorst, “Reliable updated residuals in hybrid Bi-CG methods,” *Computing* 56, 141-164 (1996).

Mixed precision with reliable updates

- New (?) idea is to apply this approach to mixed precision.

(Clark et al., arXiv:0911.3191)

$$r_0 = b - Ax_0;$$

$$\hat{r}_0 = r;$$

$$\hat{x}_0 = 0;$$

$$k = 0;$$

while $\|\hat{r}_k\| > \epsilon$ **do**

 Low precision solver iteration: $\hat{r}_k \rightarrow \hat{r}_{k+1}, \hat{x}_k \rightarrow \hat{x}_{k+1};$

if $\|\hat{r}_{k+1}\| < \delta M(\hat{r})$ **then**

$$x_{l+1} = x_l + \hat{x}_{k+1};$$

$$r_{l+1} = b - Ax_{l+1};$$

$$\hat{x}_{k+1} = 0;$$

$$\hat{r}_{k+1} = r;$$

$$l = l + 1;$$

end

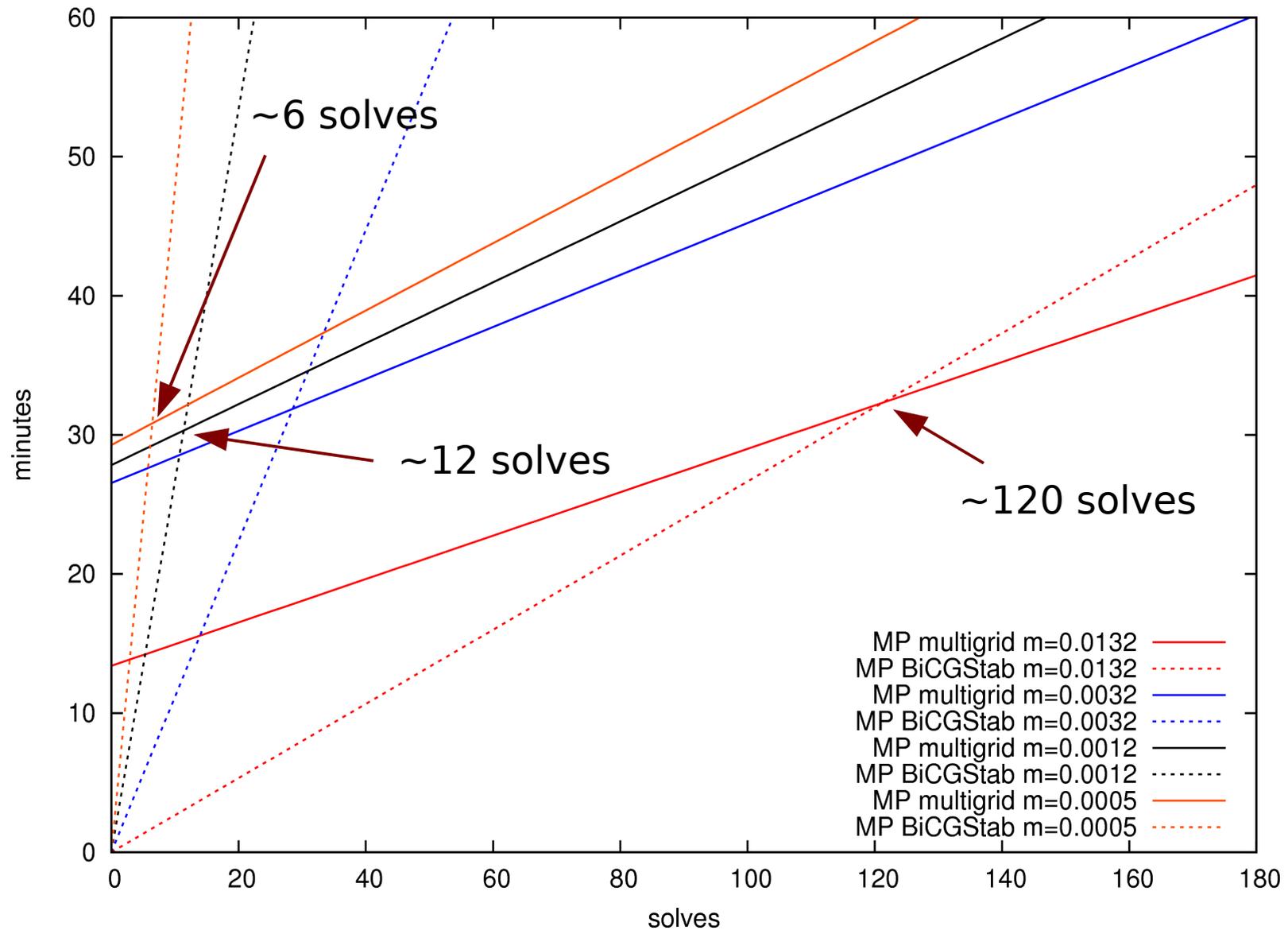
$$k = k + 1;$$

end

- $\hat{}$ denotes reduced precision.
- δ is a parameter determining the frequency of updates.
- $M(\hat{r})$ denotes the maximum iterated residual since the last update.

- Reliable updates seems to win handily at light quark masses (and is no worse than iterative refinement at heavy masses).

Multigrid: Total cost ($32^3 \times 256$)



(Osborn et al.)