ADVANCES IN ADAPTIVE MULTIGRID ALGORITHM FOR QUDA

Kate Clark, June 19th 2017

OUTLINE

Multigrid in QUDA

Accelerating the Setup

Mixed Precision

Twisted-mass Multigrid

Strong Scaling

QUDA

- “QCD on CUDA” – http://lattice.github.com/quda (open source, BSD license)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, tmLQCD, etc.
- Provides:
  - Various solvers for all major fermionic discretizations, with multi-GPU support
  - Additional performance-critical routines needed for gauge-field generation
- Maximize performance
  - Exploit physical symmetries to minimize memory traffic
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Domain-decomposed (Schwarz) preconditioners for strong scaling
  - Eigenvector and deflated solvers (Lanczos, EigCG, GMRES-DR)
  - Multi-source solvers
  - Multigrid solvers for optimal convergence
- A research tool for how to reach the exascale

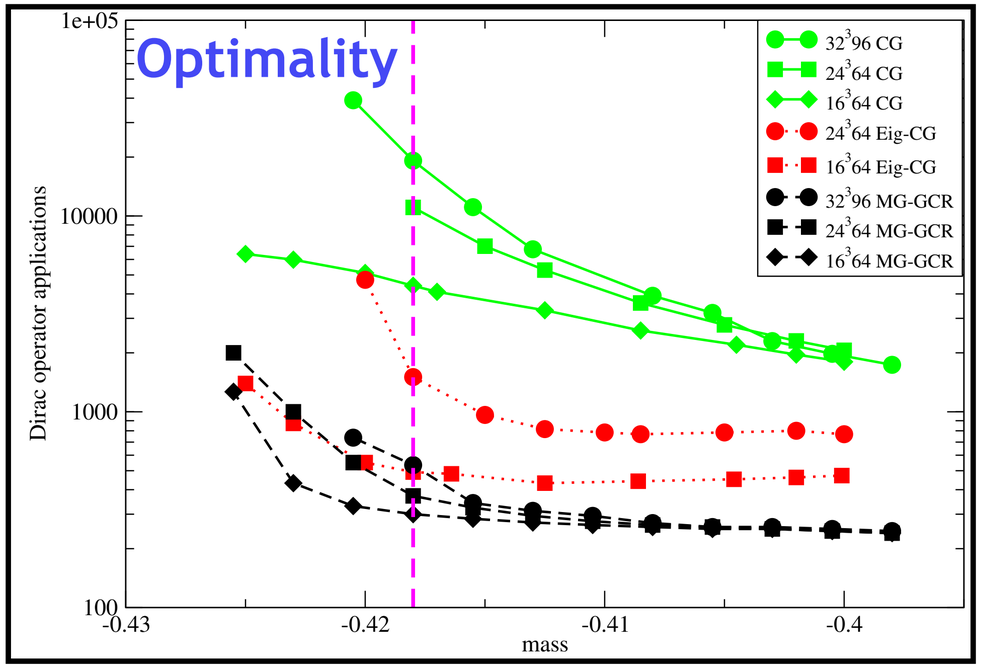
QUDA CONTRIBUTORS

Multigrid collaborators in green

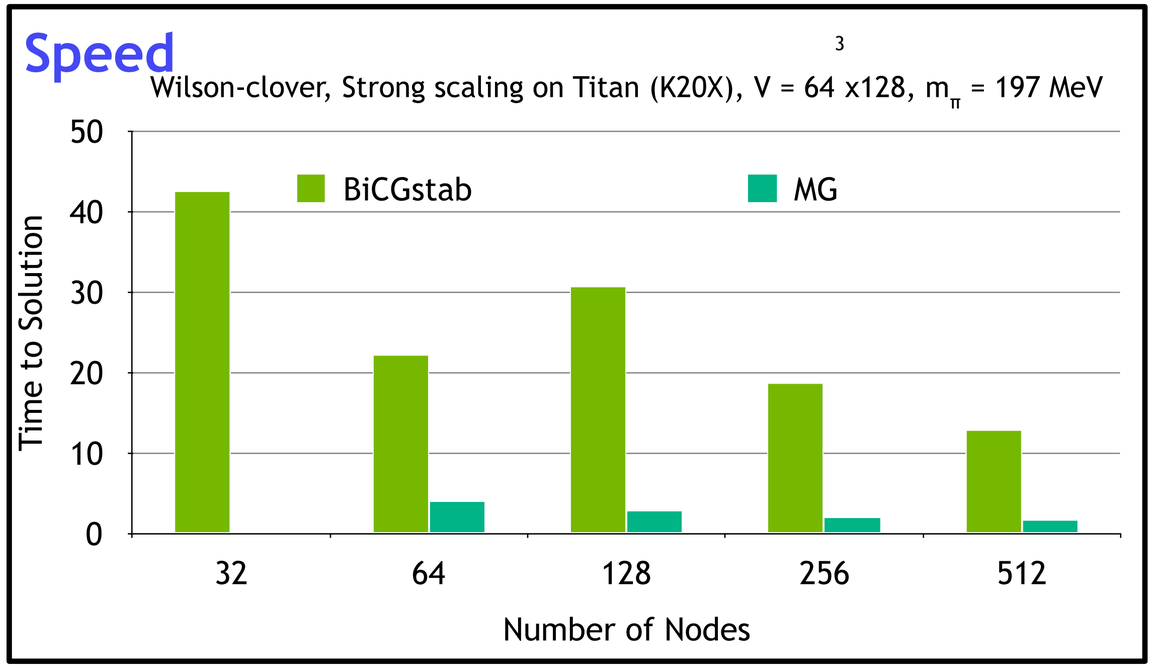
Ron Babich (NVIDIA)
Simone Bacchio (Cyprus)
Michael Baldhauf (Regensburg)
Kip Barros (LANL)
Rich Brower (Boston University)
Nuno Cardoso (NCSA)
Kate Clark (NVIDIA)
Michael Cheng (Boston University)
Carleton DeTar (Utah University)
Justin Foley (Utah -> NIH)
Joel Giedt (Rensselaer Polytechnic Institute)
Arjun Gambhir (William and Mary)
Steve Gottlieb (Indiana University)
Kyriakos Hadjiyiannakou (Cyprus)
Dean Howarth (Temple)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Bartek Kostrzewa (Bonn)
Claudio Rebbi (Boston University)
Hauke Sandmeyer (Bielefeld)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Alejandro Vaquero (INFN)
Mathias Wagner (NVIDIA)
Evan Weinberg (BU)
Frank Winter (Jlab)

MULTIGRID IN QUDA

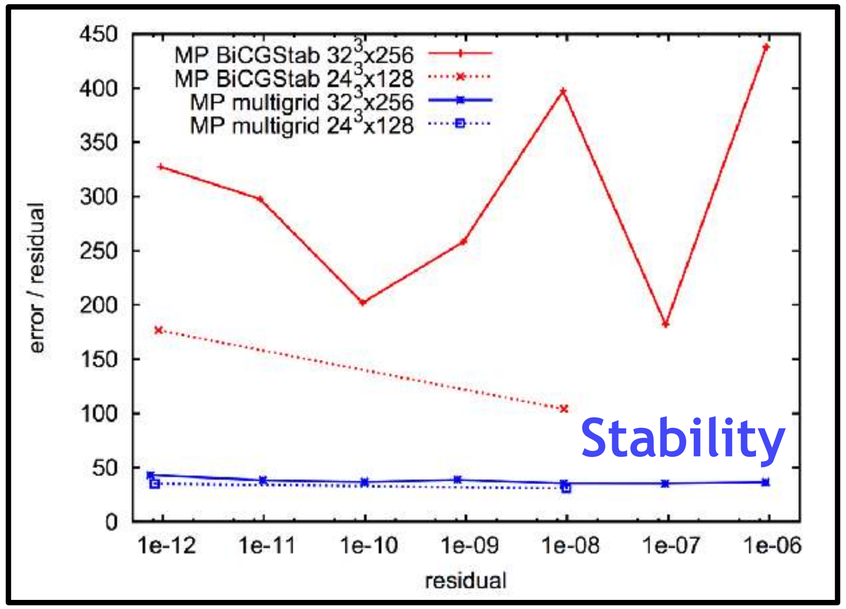
WHY MULTIGRID?



Babich *et al* 2010



Clark *et al* (2016)



Osborn *et al* 2010

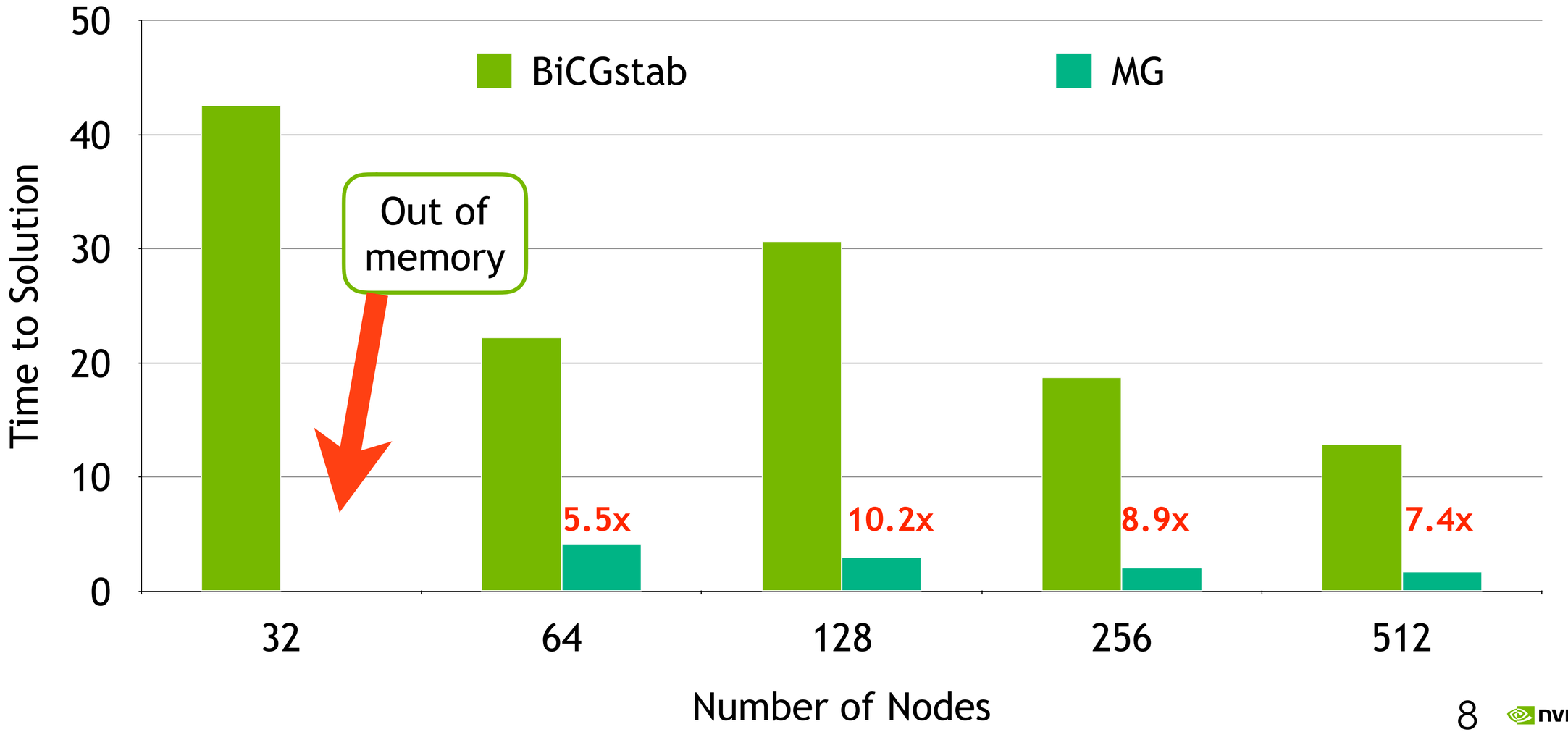
NVIDIA

PRIOR OUTSTANDING ISSUES

- Setup phase partially done on CPU (coarse-link construction and block orthogonalization)
  - Prevents use of MG with HMC
- 16-bit precision only supported on fine grid
  - Coarse operator more expensive relative to fine grid than it should be
- Strong scaling limitations:
  - Use of GCR with modified Gram-Schmidt means reductions dominate (cf Titan scaling breakdown)
  - Halo exchange of smoothers limit the strong scaling
- Memory overhead put limit of $V = 32^3 \times 16$ per P100 for clover solver
  - Forces us to strong scale more than we might like

MULTIGRID VERSUS BICGSTAB

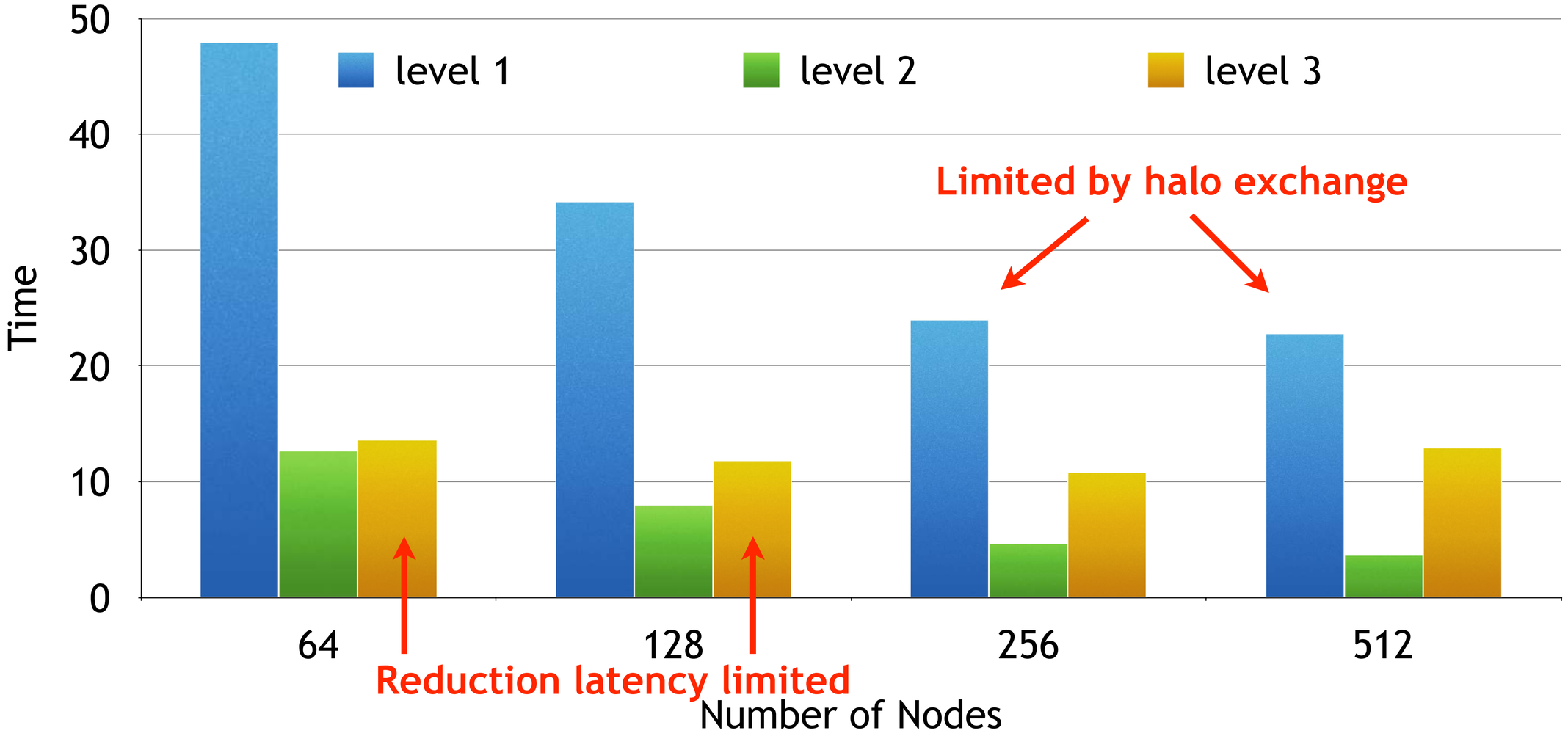
Wilson-clover, Strong scaling on Titan (K20X), V = 64^3x128, m_π = 197 MeV



NVIDIA

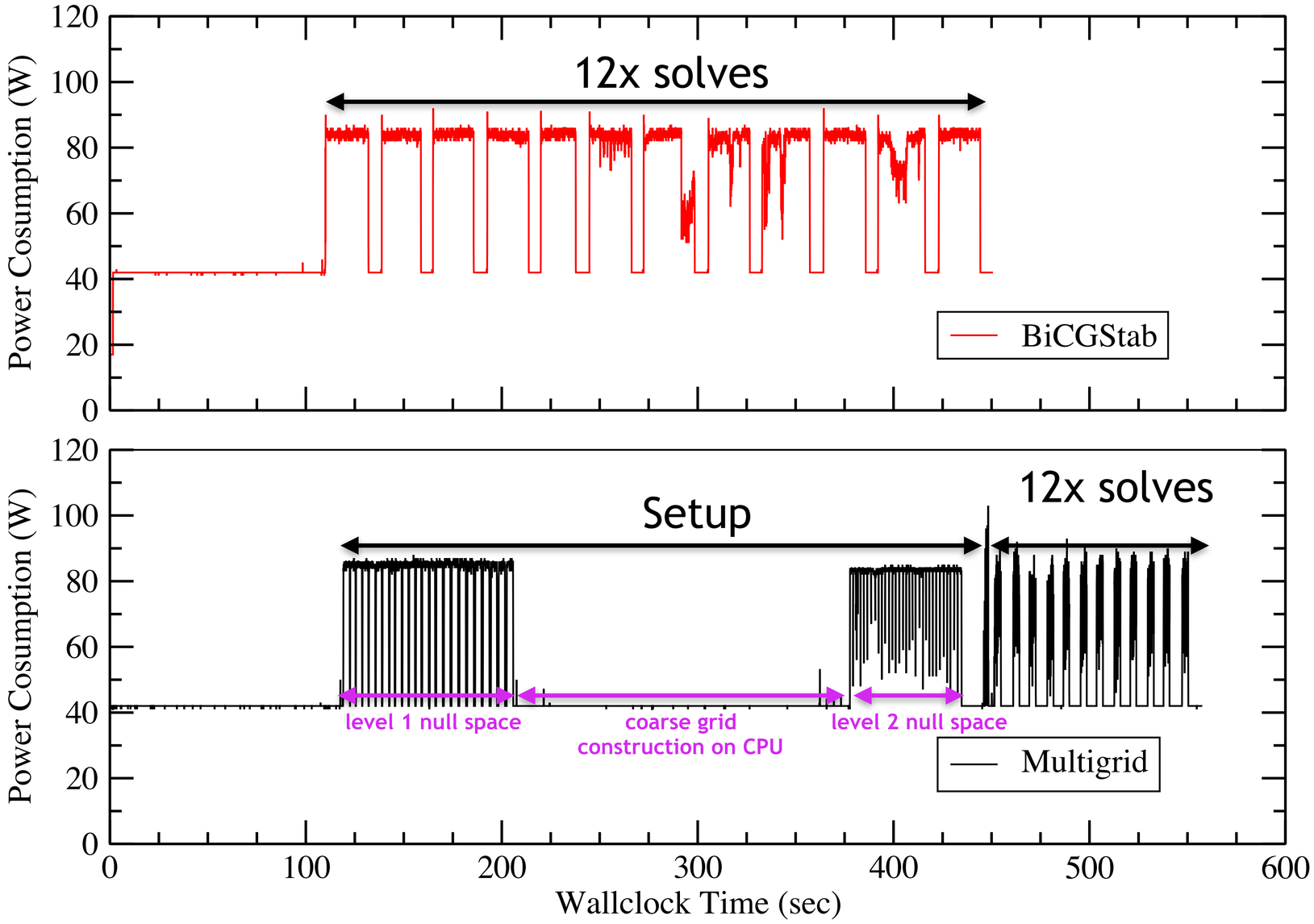
MULTIGRID TIMING BREAKDOWN

Wilson-clover, Strong scaling on Titan (K20X), V = 64^3x128, 12 linear solves



Credit to Don Maxwell @ OLCF for helping with Power measurements on Titan

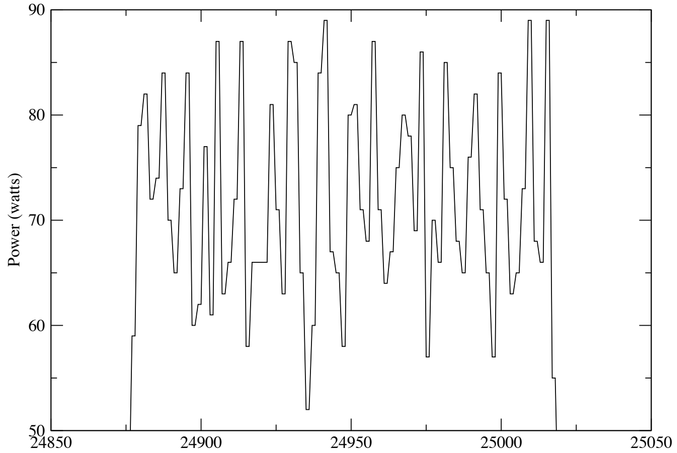
POWER EFFICIENCY



BiCGstab average power
~ 83 watts per GPU

MG average power
~ 72 watts per GPU

MG consumes less power and 10x faster



SETUP ACCELERATION FOR HMC

MULTIGRID SETUP

Essential steps

- Generate null vectors (BiCGStab, CG, etc.)
- Block Orthogonalization - was on CPU
- Coarse-link construction - was on CPU

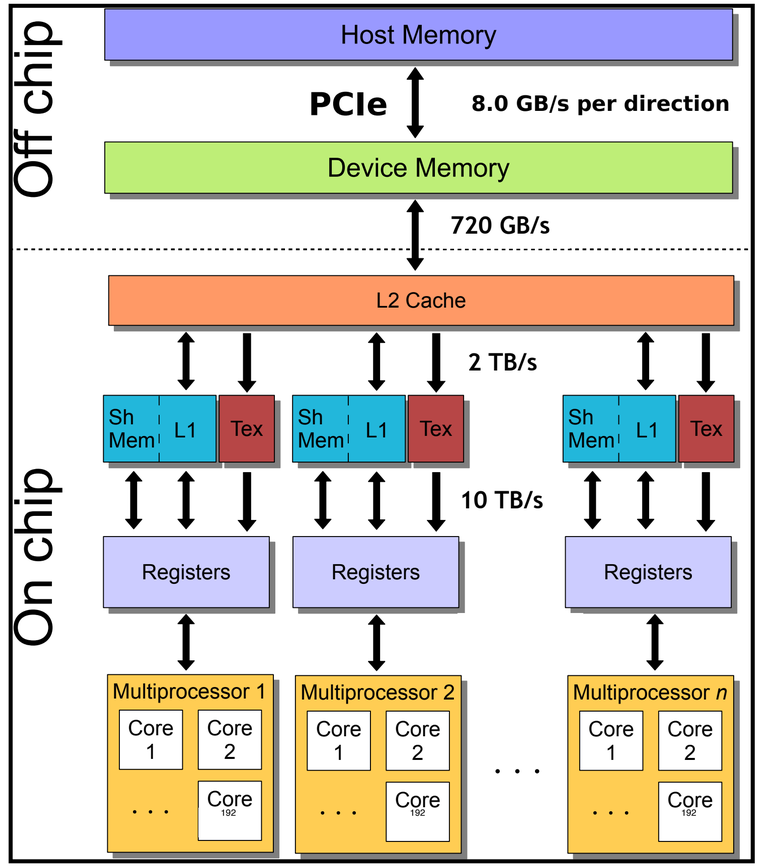
BLOCK ORTHOGONALIZATION

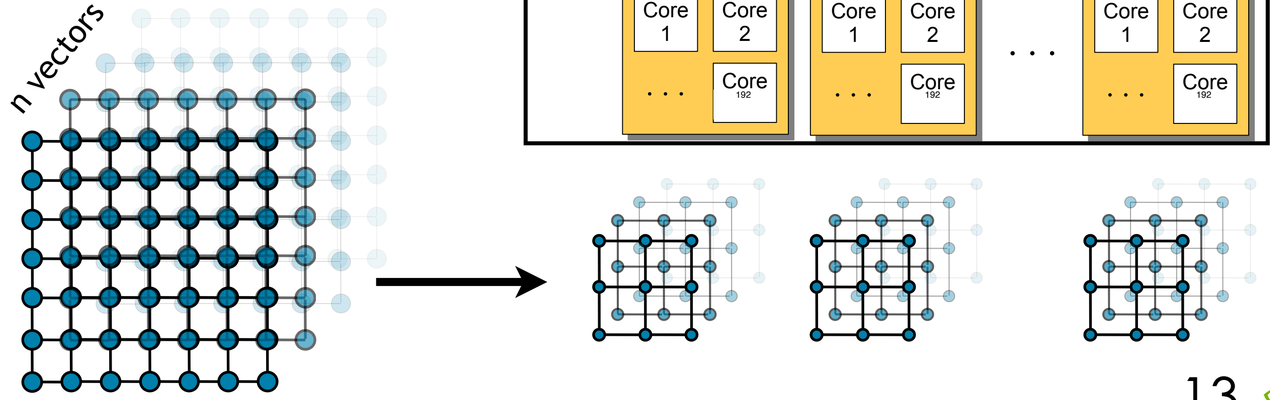
Forms the block orthonormal basis upon which we construct the coarse grid

QR on the set of null-space vectors within each multigrid aggregate

Assign each multigrid aggregate to a CUDA thread block

- All reductions are therefore local to a CUDA thread block
- Do the full block orthonormalization in a single kernel
- Minimizes total memory traffic





COARSE-LINK CONSTRUCTION

Galerkin projection of the Dirac operator onto the coarse grid, $D_c = P^\dagger D P$

E.g., for even-odd preconditioned clover we need to evaluate

$$D_c = -\sum_\mu \left[Y_\mu^{-\mathrm{f}}(\hat{x}) + Y_\mu^{+\mathrm{b}\dagger}(\hat{x}-\mu) \right] + X\delta_{\hat{x},\hat{y}}$$

$$Y_\mu^{+\mathrm{b}}(\hat{x}) = \sum_{x\in\hat{x}} V^\dagger(x) P^{+\mu} U_\mu(x) A^{-1}(y) V(y) \delta_{x,y+\mu}\delta_{\hat{x},\hat{y}+\mu} \quad \text{“backward link”}$$

$$Y_\mu^{-\mathrm{f}}(\hat{x}) = \sum_{x\in\hat{x}} V^\dagger(x) A^{-1}(x) P^{-\mu} U_\mu(x) V(y) \delta_{x,y+\mu}\delta_{\hat{x},\hat{y}+\mu} \quad \text{“forward link”}$$

$$X(\hat{x}) = \sum_{x\in\hat{x},\mu} V^\dagger(x) \left(P^{+\mu} U_\mu(x) A^{-1}(y) + A^{-1}(x) P^{-\mu} U_\mu(x) \right) V(y) \delta_{x,y+\mu}\delta_{\hat{x},\hat{y}} \quad \text{“coarse clover”}$$

COARSE-LINK CONSTRUCTION

Recipe

1. Compute required intermediate $T = UA^{-1}V$
   - Multi-RHS matrix-vector => matrix-matrix operation
   - High efficiency on parallel architectures

2. Compute coarse link matrix $V^{\dagger}P^{+}T$
   - Naive intermediate has fine-grid geometry and coarse-grid degrees of freedom
   - E.g., 16^4 fine grid with 48 degrees of freedom per site => ~18 GB per direction

3. Sum contribution to Y or X as needed

COARSE-LINK CONSTRUCTION

Recipe

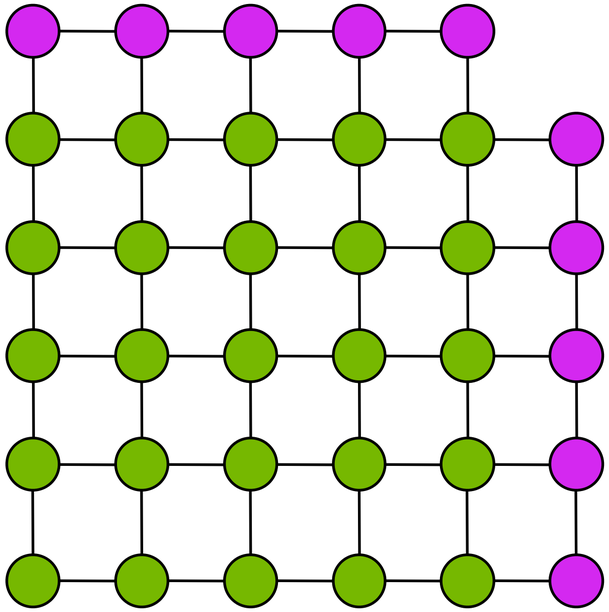
1. Compute required intermediate $T = UA^{-1}V$
   - Multi-RHS matrix-vector => matrix-matrix operation
   - High efficiency on parallel architectures

2. Compute coarse link matrix $V^{\dagger}P^{+}T$

Need a single fused computation to avoid intermediate

3. Sum contribution to Y or X as needed

COARSE-LINK CONSTRUCTION



Employ fine-grained parallelization

- fine-grid geometry
- coarse-grid color

Each thread computes its assigned matrix elements

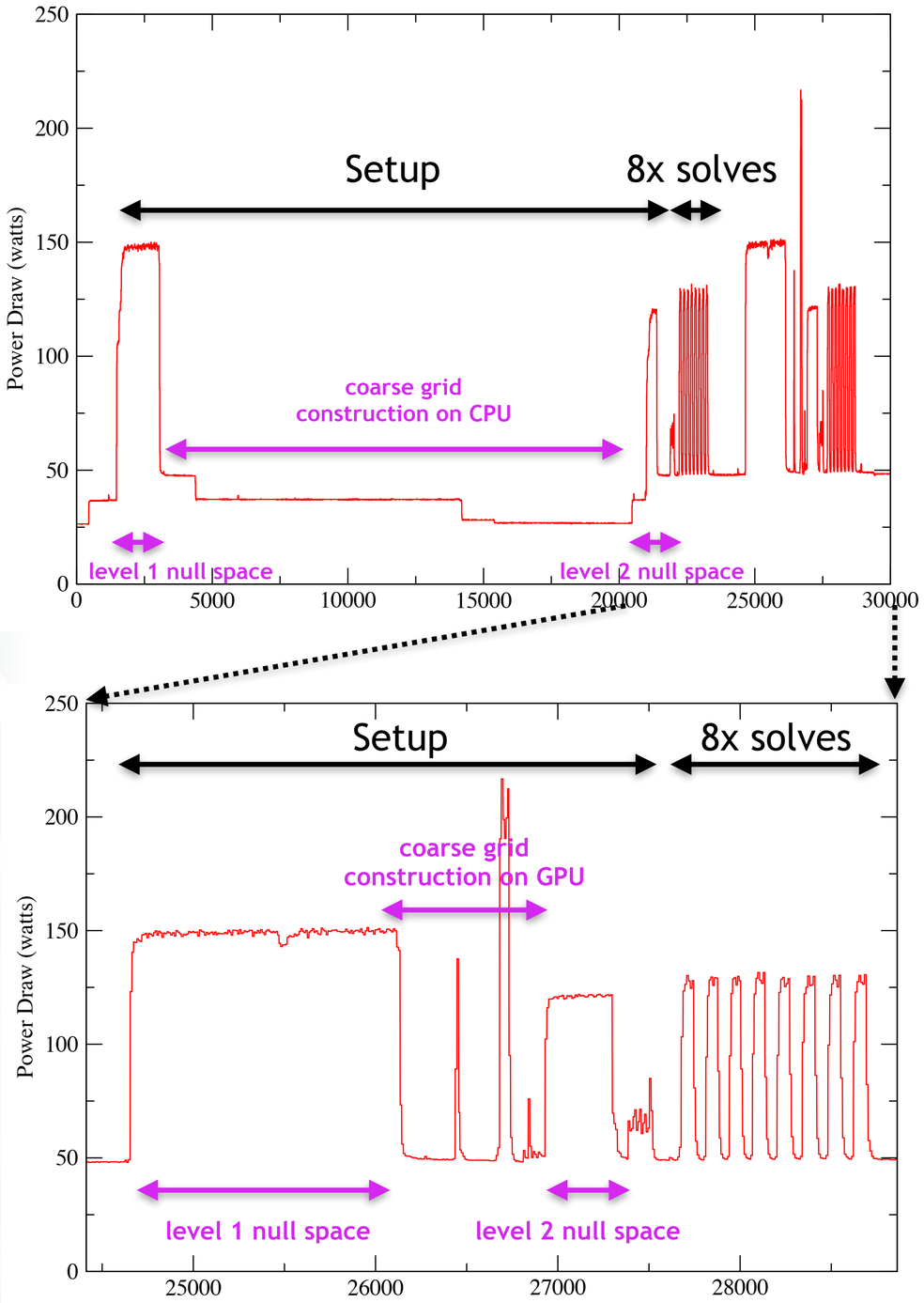
Atomically update the relevant coarse link field depending on thread location

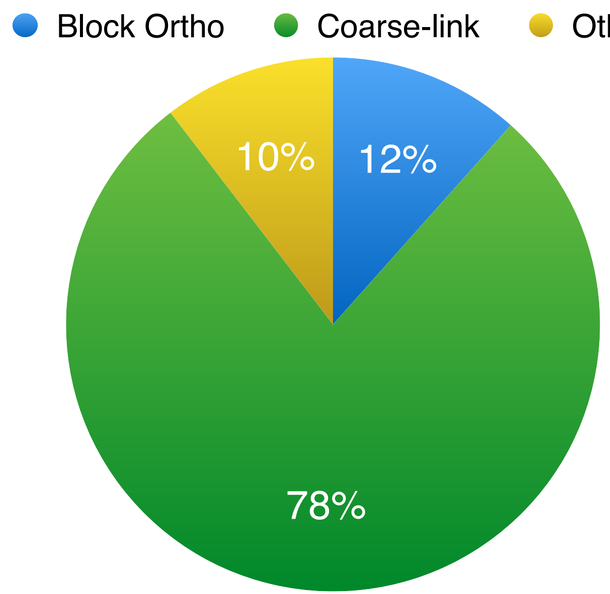
$$Y = \sum$$ 

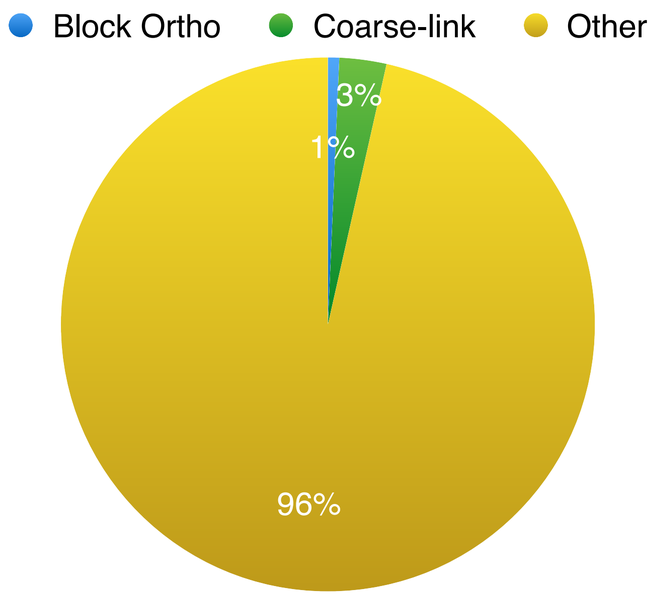
$$X = \sum$$ 

Finally, neighbour exchange boundary link elements

RESULTS







Null-space finding now dominates the setup process

Coarse-link construction runs at ~0.5-1 TFLOPS (P100)

Further factor of 2-3x improvement available if needed

HMC MULTIGRID

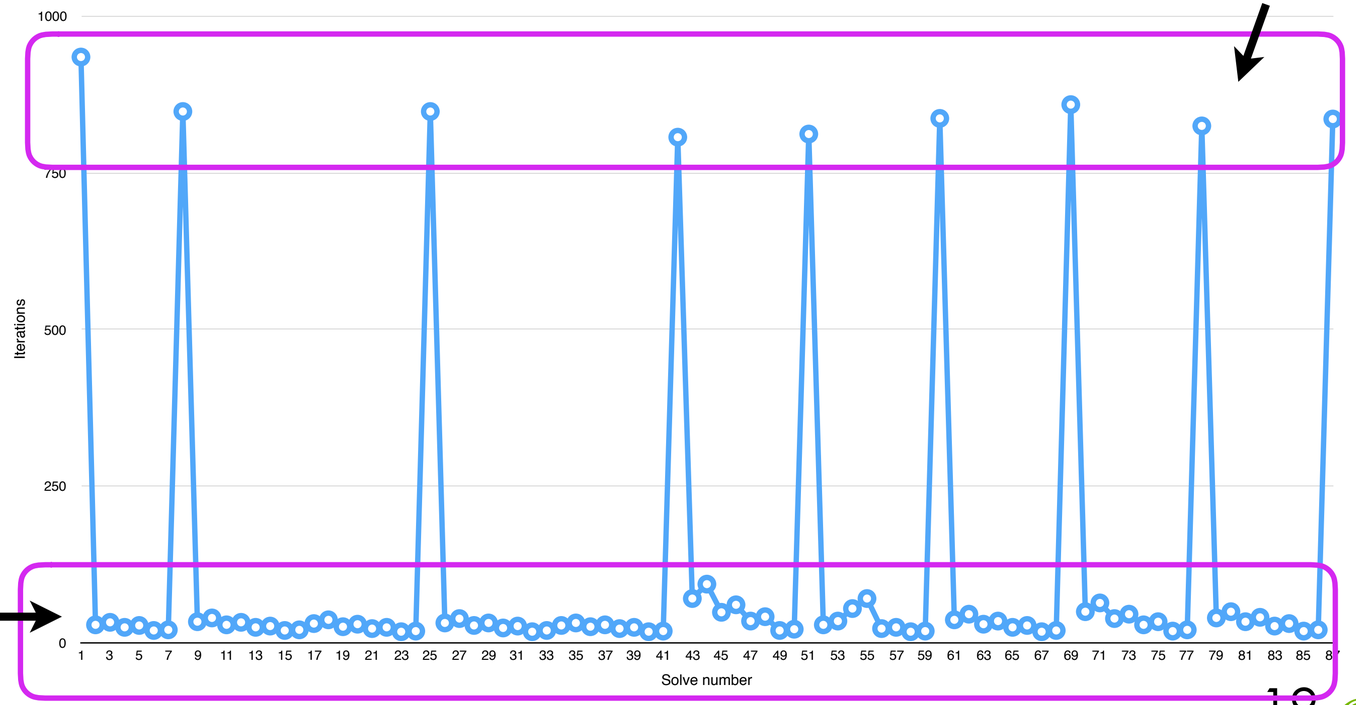
Exposed `updateMultigridQuda` interface to the outside world

Allows for MG state to be refreshed and evolved along MD trajectory

Plugged into Chroma HMC…

Hot off the press from Titan

V = 64^3x128, Clover (R)HMC



MIXED PRECISION

16-BIT FIXED-POINT FOR COARSE GRIDS

QUDA uses 16-bit precision as a memory traffic reduction strategy

Actually uses “block float” format

- Uses 16-bit fixed point per grid point with single float to normalize
- CG / BiCGStab has ~10% hit in iteration count for overall ~1.7x speedup vs FP32

Initial implementation of Multigrid did not support 16-bit precision on coarse grids

- Was not immediately obvious how to marry block float with fine-grain parallelization
- FP16 a possibility, but range is limiting

16-BIT FIXED-POINT FOR COARSE GRIDS

Solution is simple: use global fixed point

- ➡ null-space vectors
- ➡ coarse-link construction temporaries
- ➡ coarse-link matrices

already block orthonormal

estimate max element to set scale,
e.g., $|U\,V|_{max} \sim |U|_{max}\,|V|_{max}$

Leave vector fields in FP32 since coarse operator is never bound by vector-field traffic

All fixed-point <-> float conversion hidden in QUDA-field accessors

- No changes to any kernel code
- Set scale of field prior to writing to it, then all read/write access is opaque

16-BIT FIXED-POINT FOR COARSE GRIDS

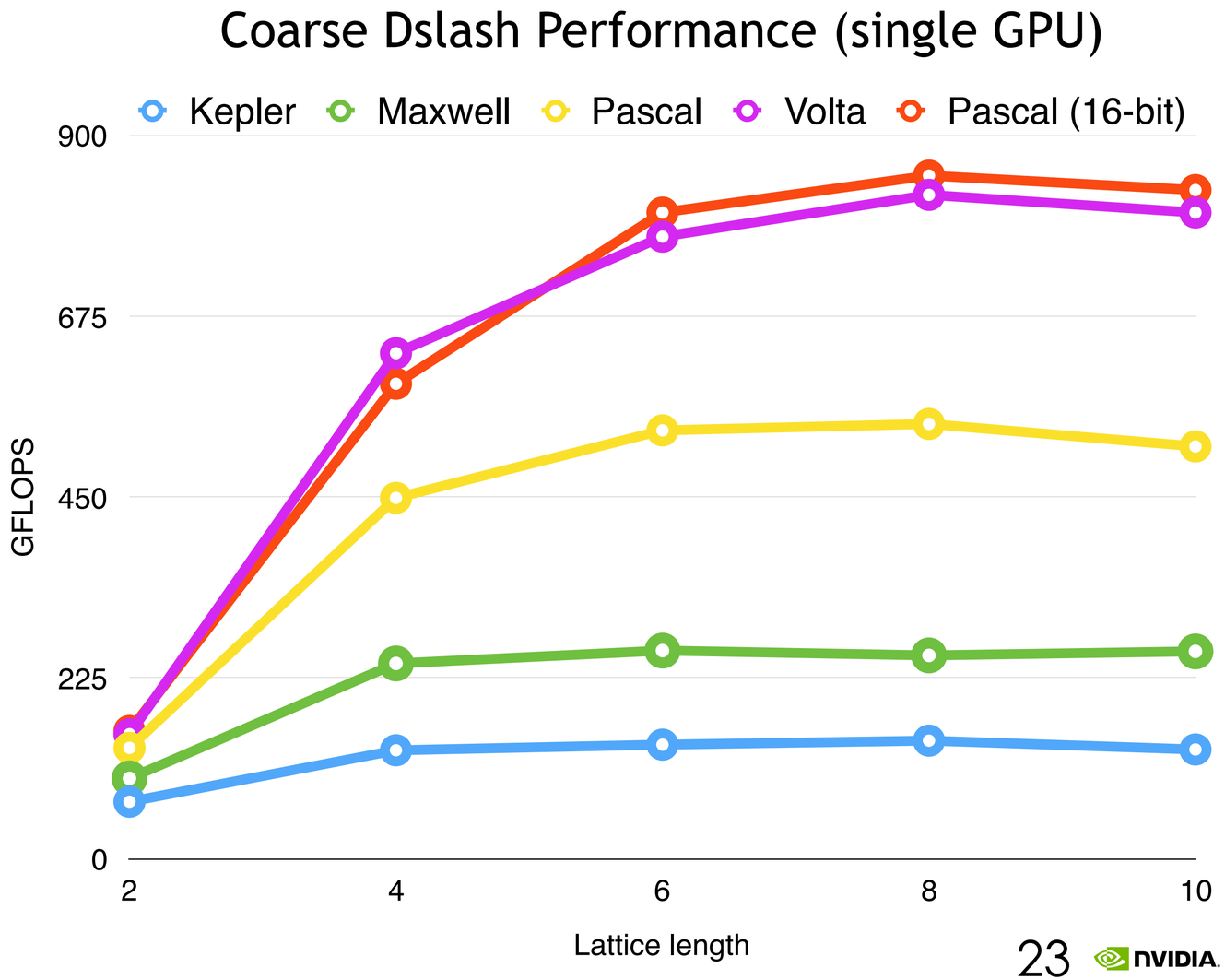
16-bit is like running with a new GPU!

Coarse-link setup kernels 1.8x faster

Restriction and Prolongation 1.8x faster

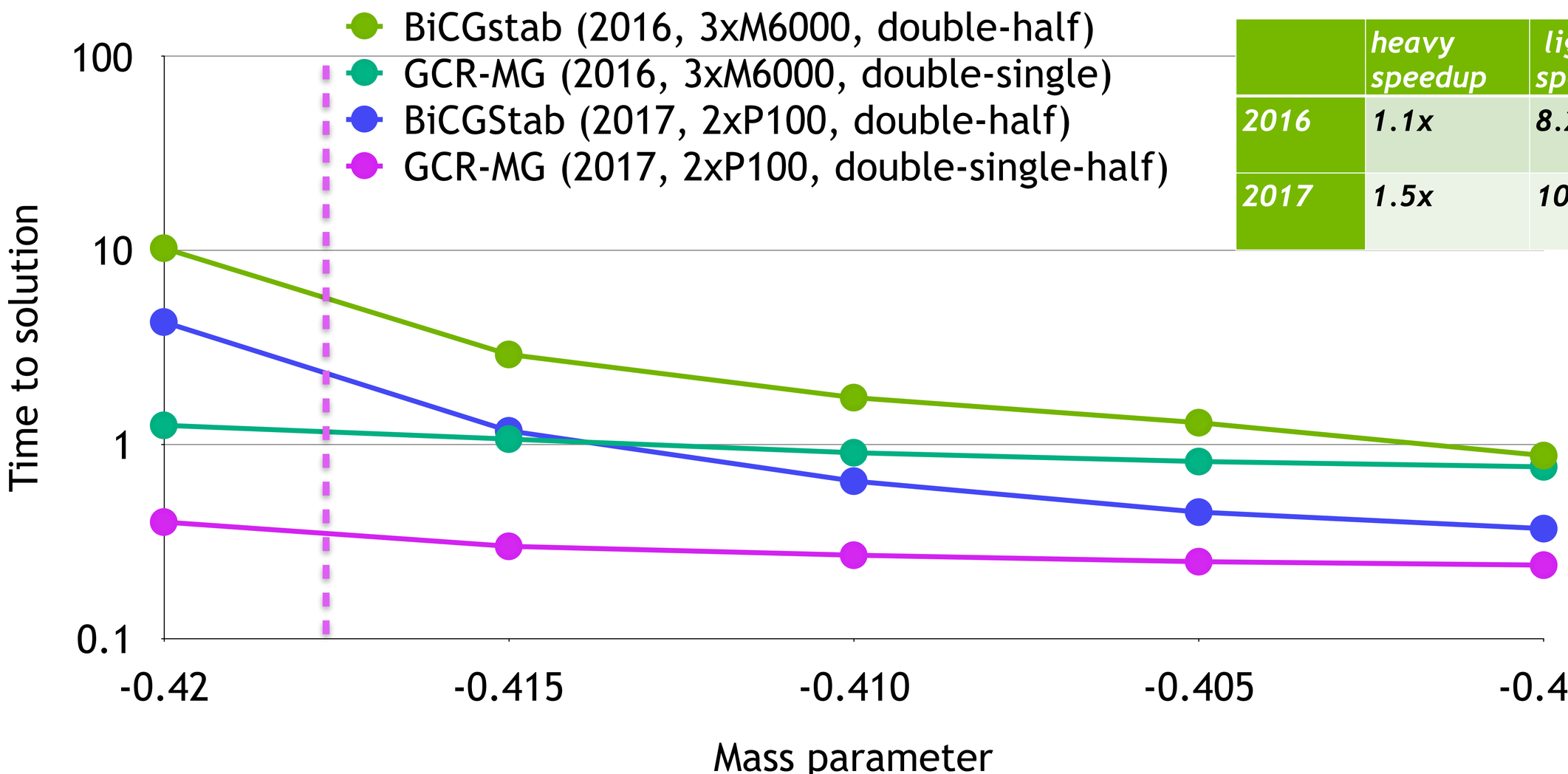
33% reduction in peak memory

Absolutely zero effect on multigrid convergence



MULTIGRID VERSUS BICGSTAB

Wilson, V = 24^3x64, single workstation



	heavy speedup	*light speedup*
2016	*1.1x*	*8.2x*
2017	*1.5x*	*10.7x*

TWISTED-MASS MULTIGRID

MULTIGRID FOR TWISTED MASS

QUDA has optional “mu scaling” on coarse grids for faster coarse-grid convergence (Alexandrou *et al*, 2016)

When solving the even-odd preconditioned system additional twist introduced onto Schur-compliment

Due to BiCGStab stagnation / divergence with twisted-mass fermions, use CG as setup solver

Twisted-clover, $V = 32^3 \times 64$, kappa = 0.1372938, csw = 1.57551, $\mu_{sea} = 0.006$, 2xP100

	Iterations		Time per solve (s)	
μ	CG	GCR-MG	CG	GCR-MG
0.006	2587	23	17.62	1.19
0.005	3076	23	20.96	1.21
0.004	3865	24	26.26	1.29
0.003	5126	25	34.87	1.37
0.002	7457	27	50.68	1.55
0.001	13836	32	93.97	1.82
0.0009	14701	32	99.82	1.79
0.0008	15443	33	104.86	1.85
0.0007	16343	34	110.96	1.93
0.0006	18292	35	124.13	1.97
0.0005	19001	36	129.02	2.04

MULTIGRID FOR TWISTED MASS

MG vs CG vs deflated CG

Compared to CG

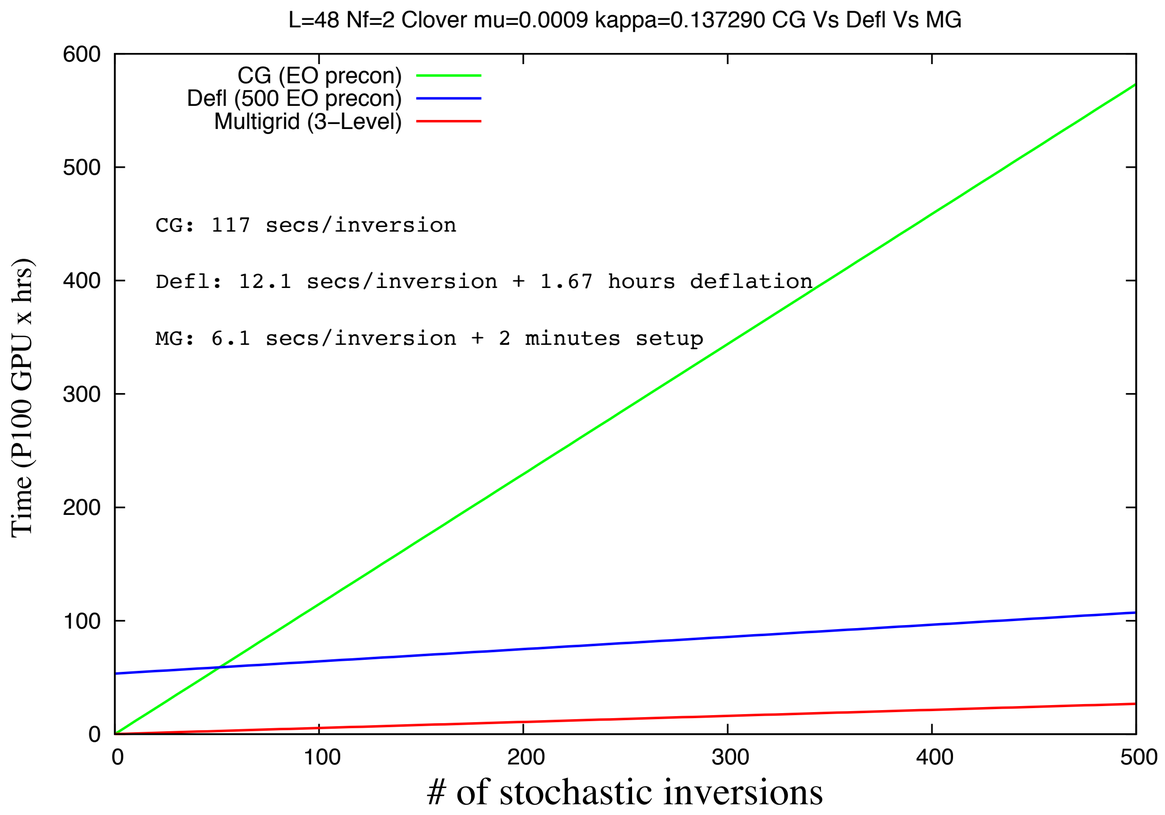
- 19x speedup

Compared to deflation

- MG has lower memory overhead
- Much reduced setup time
- 2x faster time per solve

(Note these results do not include 16-bit)

32x P100 (Piz Daint)



STRONG SCALING

PIPELINED GCR

Initially used GCR with modified Gram-Schmidt

- Numerically stable
- O(N) reductions per step

Would like to use classical Gram Schmidt

- O(1) reductions per step
- Numerically unstable

Solution: use “pipelined GCR” with O(N/pipeline) reductions per step

- Interpolate between classical Gram Schmidt and modified Gram Schmidt
- pipeline = 1: modified Gram Schmidt
- pipeline = N: ordinary Gram Schmidt

In practice pipeline = 8 suppresses reduction cost with zero stability impact

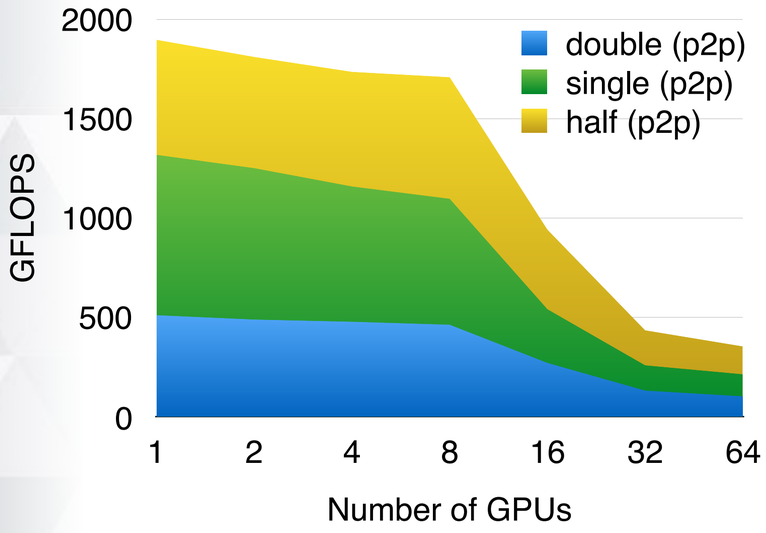
Utilizes multi-BLAS technology developed for block solvers (see talk by Mathias Wagner)

GPU DIRECT RDMA

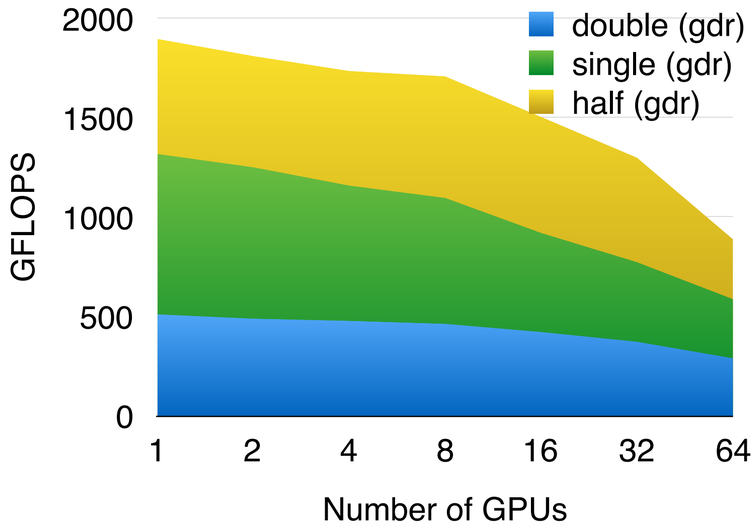
QUDA now has first-class support for GPU Direct RDMA

- Direct GPU <-> NIC communication on systems that support it
- Dramatic improvement in inter-node scaling

24^4 per GPU weak scaling on Saturn V

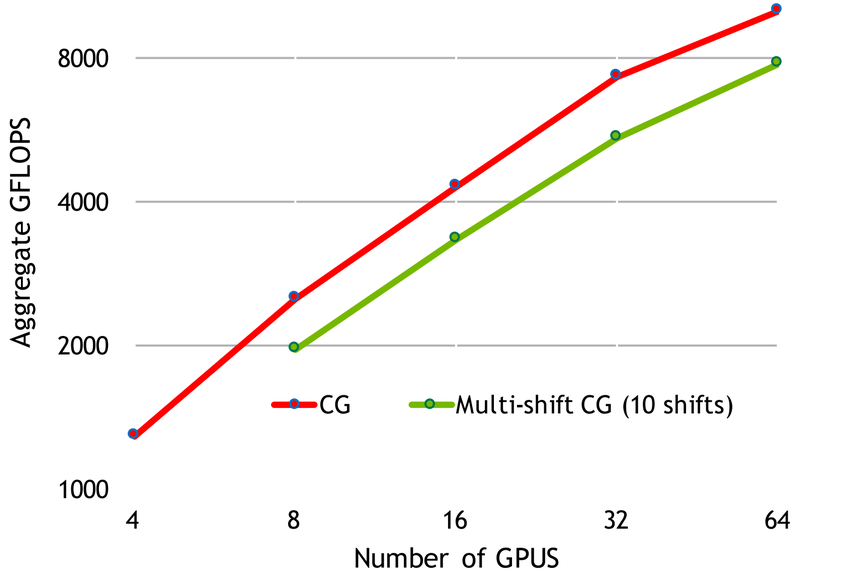


without GDR



with GDR

$48^3 96$ strong scaling on Saturn V (DP)



DOMAIN-DECOMPOSITION SMOOTHERS

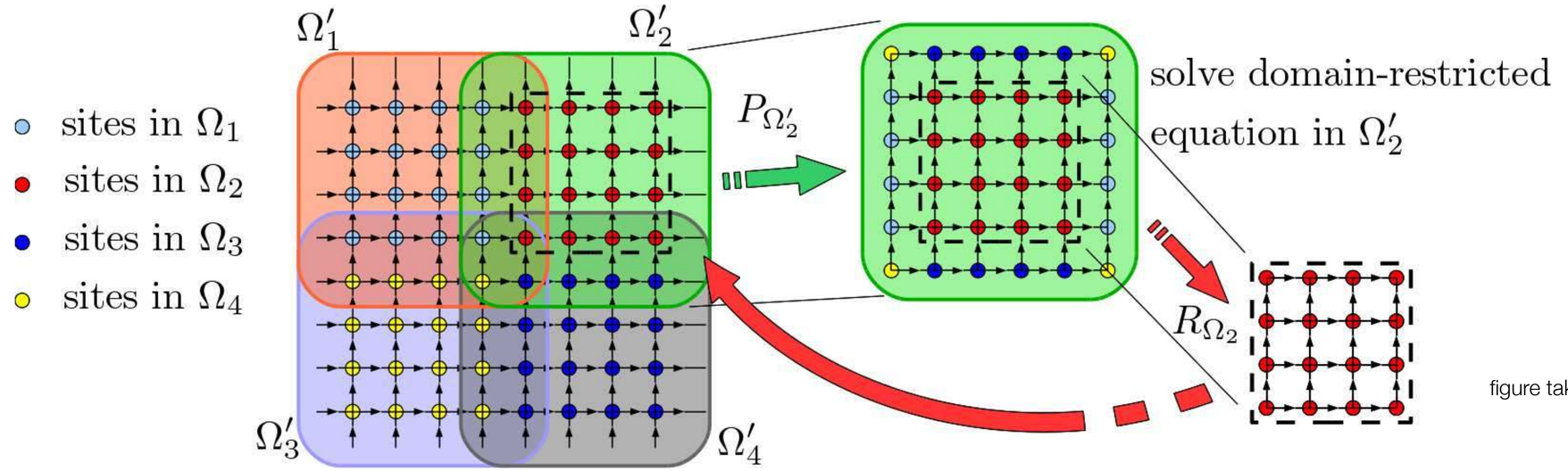


figure taken from Osaki and Ishikawa

Domain-decomposition smoothers are effective smoothers for QCD MG (Frommer *et al*)

QUDA now has support for both additive and multiplicative Schwarz smoothing

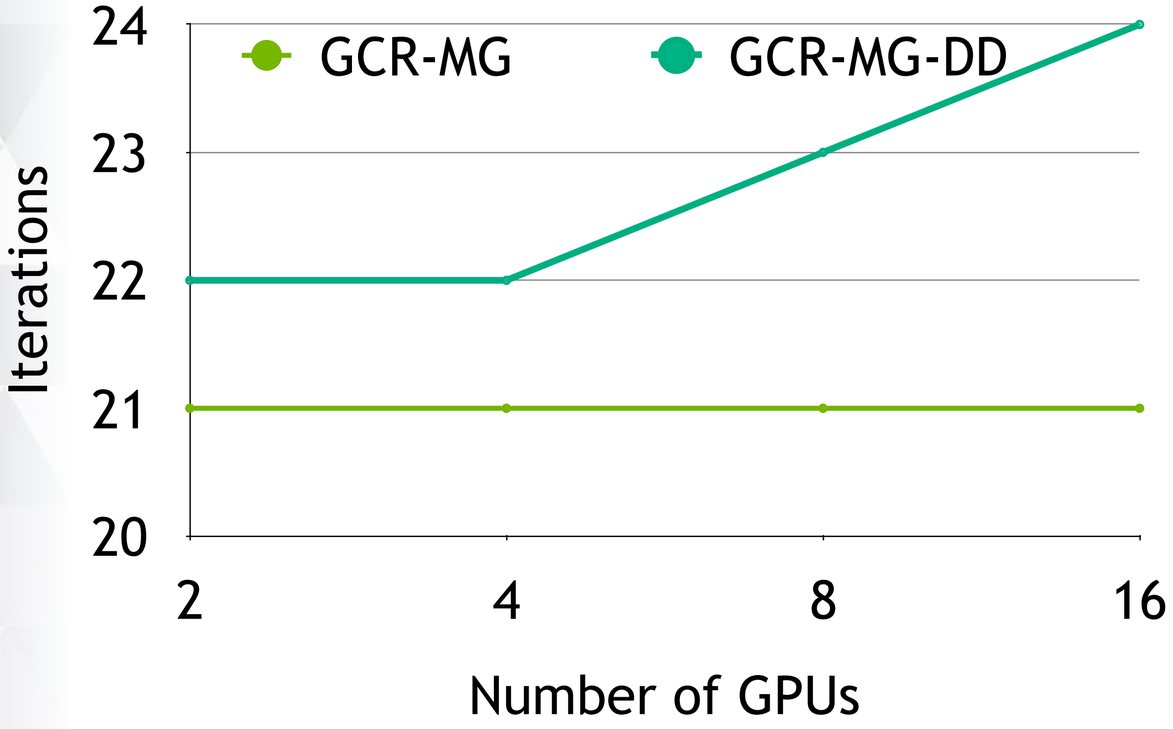
Enable at any level and / or combine with even/odd preconditioning at any level

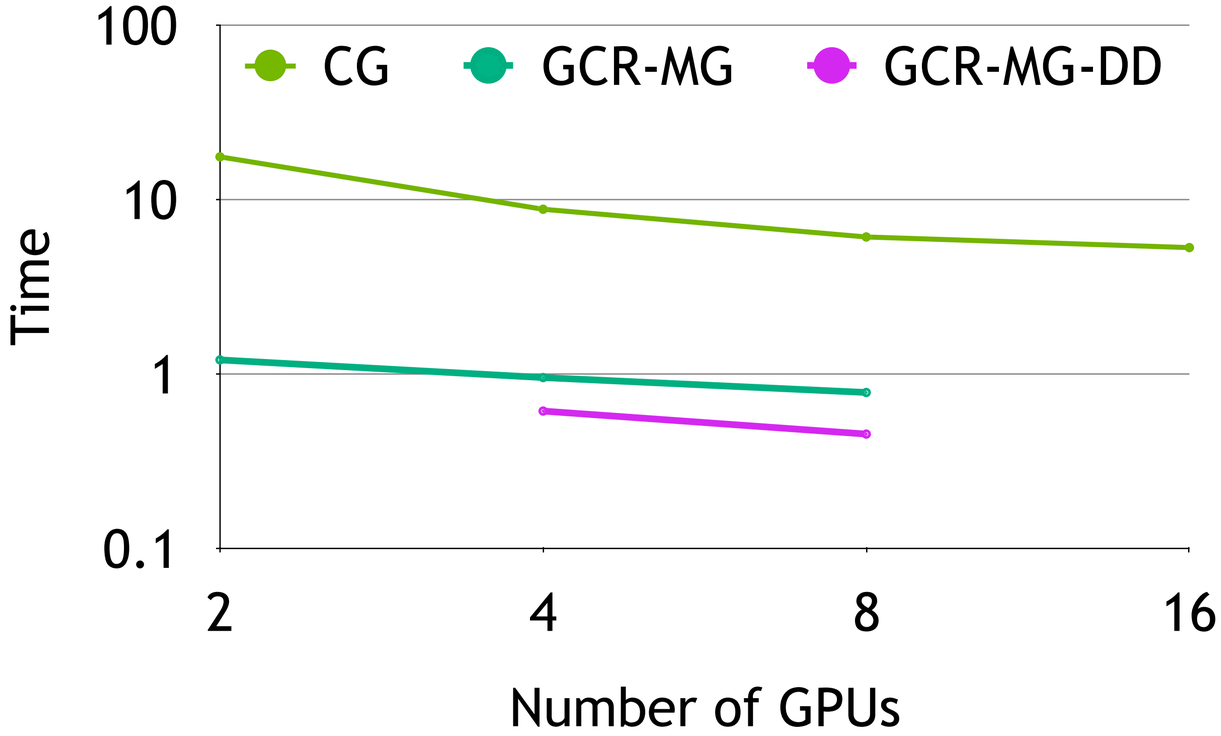
Dramatic reduction in communication important on systems with weak networks

E.g., Piz Daint vs. Saturn V

INITIAL SCHWARZ RESULTS

Twisted-clover, V = 32^3x64, κ = 0.1372938, csw = 1.57551, μ = 0.006, Piz Daint
Additive Schwarz smoother





SUMMARY

QUDA Multigrid has significantly improved in the past 12 months

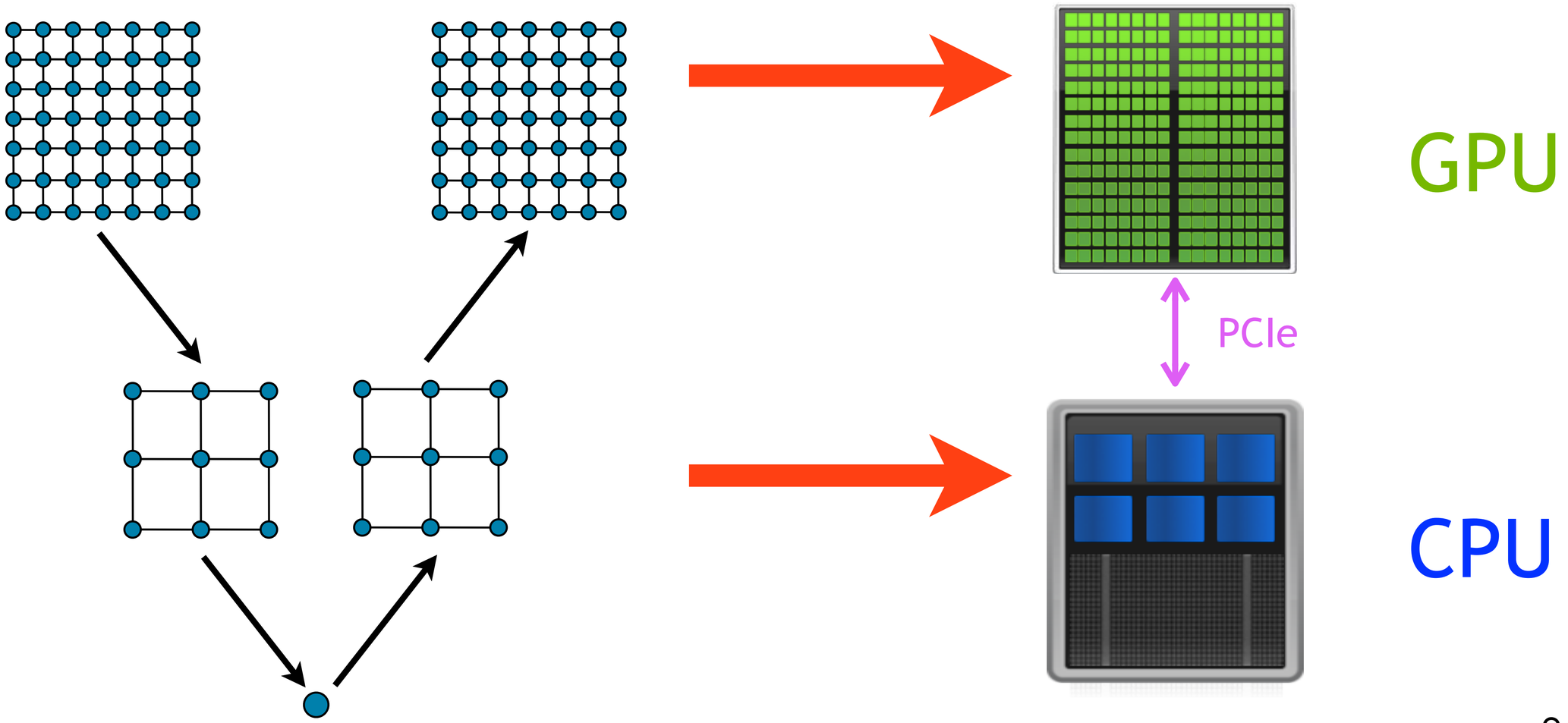
- Dramatically reduced setup time has enabled HMC-MG (in Chroma)
- First-class mixed-precision implementation ($V=32^3x64$ per GPU now possible)
- Much improved strong scaling
- Full support for twisted-mass multigrid

Ongoing and next projects

- Reducing setup cost
- Staggered Multigrid (see Evan Weinberg's talk)
- Domain-wall multigrid: just started working on this

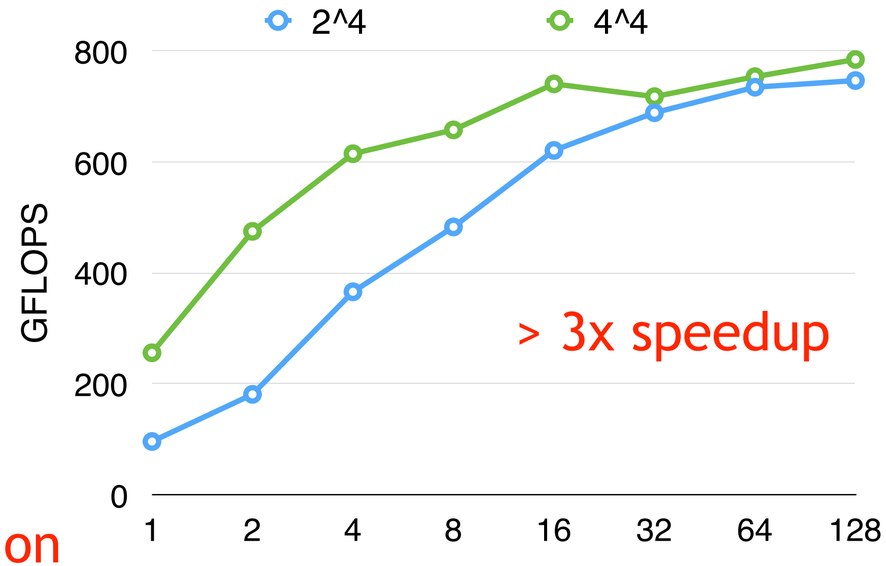
BACKUP

HIERARCHICAL ALGORITHMS ON HETEROGENEOUS ARCHITECTURES



MULTI-SRC SOLVERS

- Multi-src solvers increase locality through link-field reuse
- Multi-grid operators even more so since link matrices are 48x48
  - Coarse Dslash / Prolongator / Restrictor
- Coarsest grids also latency limited
  - Kernel level latency
  - Network latency
- Multi-src solvers are a solution
  - More parallelism
  - Bigger messages



Coarse dslash on M6000 GPU vs #rhs

ADAPTIVE GEOMETRIC MULTIGRID

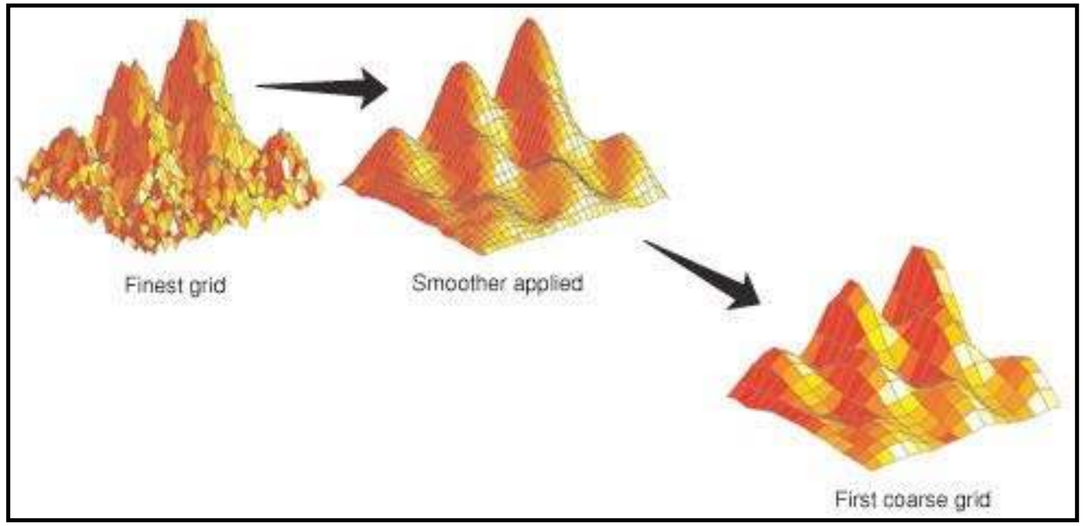
Adaptively find candidate null-space vectors

- Dynamically learn the null space and use this to define the prolongator
- Algorithm is self learning

Setup

1. Set solver to be simple smoother
2. Apply current solver to random vector $v_i = P(D)\ \eta_i$
3. If convergence good enough, solver setup complete
4. Construct prolongator using fixed coarsening $(1 - P\ R)\ v_k = 0$
   - ➡ Typically use 4^4 geometric blocks
   - ➡ Preserve chirality when coarsening $R = \gamma_5\ P^\dagger\ \gamma_5 = P^\dagger$
5. Construct coarse operator ($D_c = R\ D\ P$)
6. Recurse on coarse problem
7. Set solver to be augmented V-cycle, goto 2

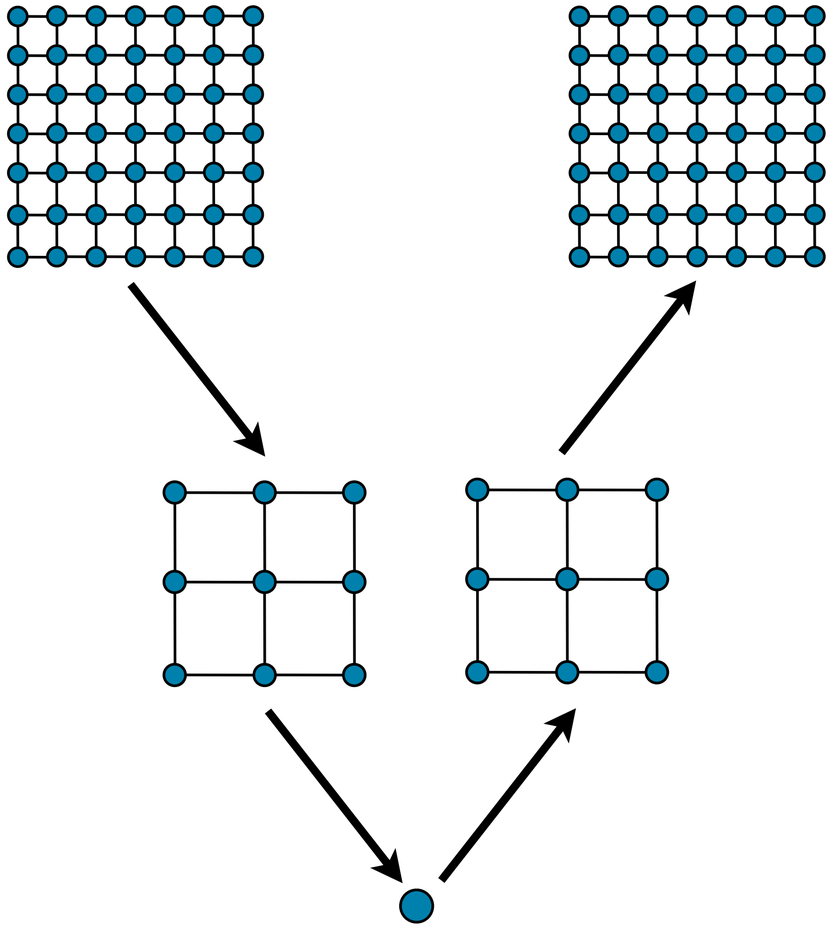
Babich *et al* 2010



Falgout

see also Inexact Deflation (Lüscher, 2007)
Local coherence = weak approximation theory

THE CHALLENGE OF MULTIGRID ON GPU



GPU requirements very different from CPU

- Each thread is slow, but O(10,000) threads per GPU

Fine grids run very efficiently

- High parallel throughput problem

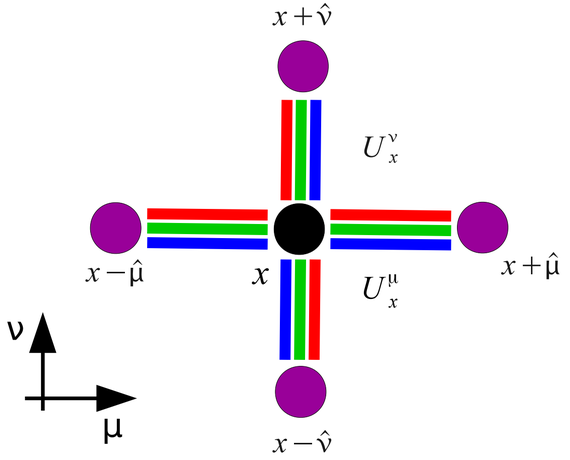
Coarse grids are worst possible scenario

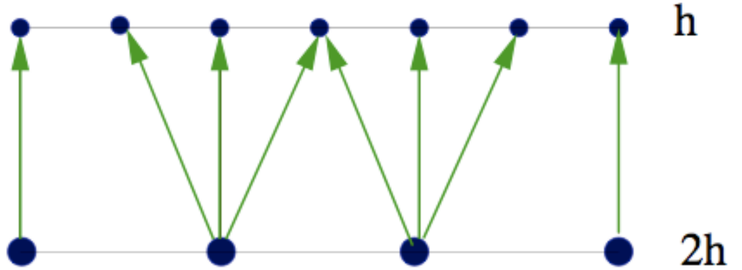
- More cores than degrees of freedom
- Increasingly serial and latency bound
- Little's law (bytes = bandwidth * latency)
- Amdahl's law limiter

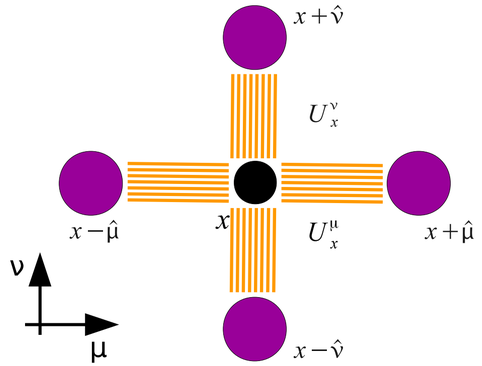
Multigrid exposes many of the problems expected at the Exascale

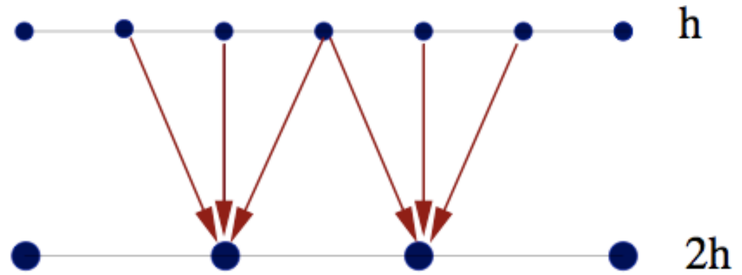
INGREDIENTS FOR PARALLEL ADAPTIVE MULTIGRID

- Multigrid setup
  - Block orthogonalization of null space vectors
  - Batched QR decomposition
- Smoothing (relaxation on a given grid)
  - Repurpose existing solvers
- Prolongation
  - interpolation from coarse grid to fine grid
  - one-to-many mapping
- Restriction
  - restriction from fine grid to coarse grid
  - many-to-one mapping
- Coarse Operator construction (setup)
  - Evaluate $R\,A\,P$ locally
  - Batched (small) dense matrix multiplication
- Coarse grid solver
  - Need optimal coarse-grid operator

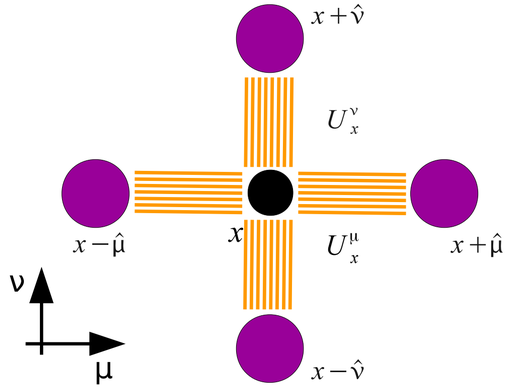








COARSE GRID OPERATOR

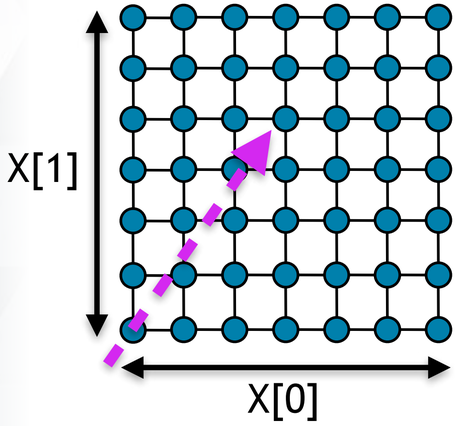


- Coarse operator looks like a Dirac operator (many more colors)
  - Link matrices have dimension $2N_v$ x $2N_v$ (e.g., 48 x 48)

$$\hat{D}_{\mathbf{i}\hat{s}\hat{c},\mathbf{j}\hat{s}'\hat{c}'} = -\sum_{\mu}\left[Y^{-\mu}_{\mathbf{i}\hat{s}\hat{c},\mathbf{j}\hat{s}'\hat{c}'}\delta_{\mathbf{i}+\mu,\mathbf{j}} + Y^{+\mu\dagger}_{\mathbf{i}s\hat{c},\mathbf{j}s'\hat{c}'}\delta_{\mathbf{i}-\mu,\mathbf{j}}\right] + \left(M - X_{\mathbf{i}\hat{s}\hat{c},\mathbf{j}\hat{s}'\hat{c}'}\right)\delta_{\mathbf{i}\hat{s}\hat{c},\mathbf{j}\hat{s}'\hat{c}'}.$$

- Fine vs. Coarse grid parallelization
  - Fine grid operator has plenty of grid-level parallelism
    - E.g., 16x16x16x16 = 65536 lattice sites
  - Coarse grid operator has diminishing grid-level parallelism
    - first coarse grid 4x4x4x4= 256 lattice sites
    - second coarse grid 2x2x2x2 = 16 lattice sites
- Current GPUs have up to 3840 processing elements
- Need to consider finer-grained parallelization
  - Increase parallelism to use all GPU resources
  - Load balancing

SOURCE OF PARALLELISM



1. Grid parallelism

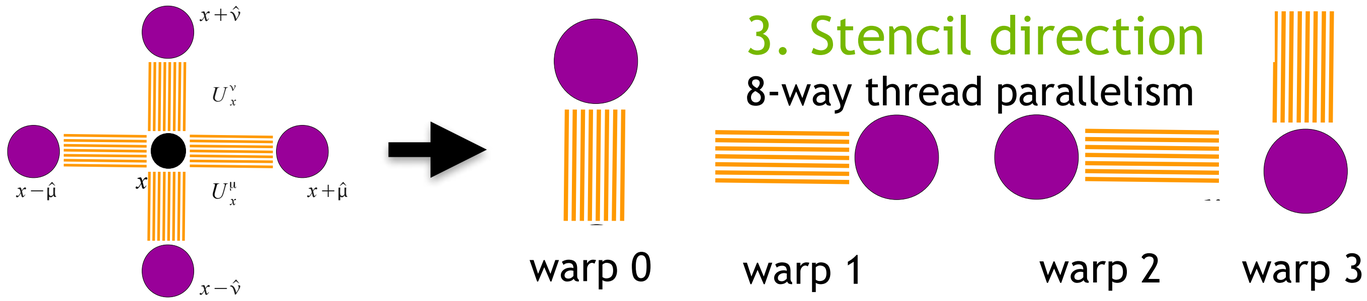
Volume of threads

thread y index

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} += \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

2. Link matrix-vector partitioning

2 N_{vec}-way thread parallelism (spin * color)



3. Stencil direction

8-way thread parallelism

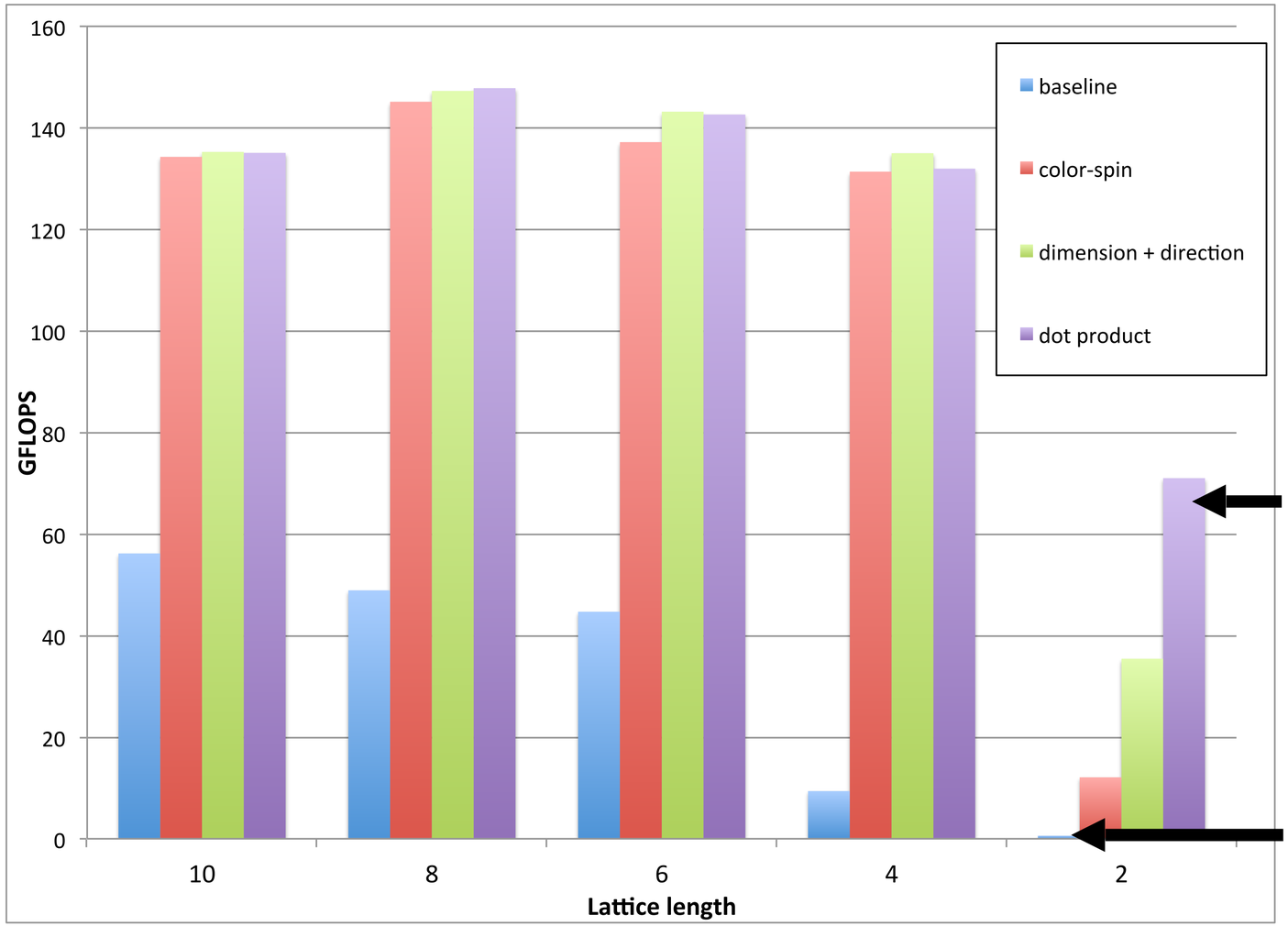
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \Rightarrow \begin{pmatrix} a_{00} & a_{01} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} + \begin{pmatrix} a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_2 \\ b_3 \end{pmatrix}$$

4. Dot-product partitioning

4-way thread parallelism + ILP

COARSE GRID OPERATOR PERFORMANCE

Tesla K20X (Titan), FP32, N_{vec} = 24



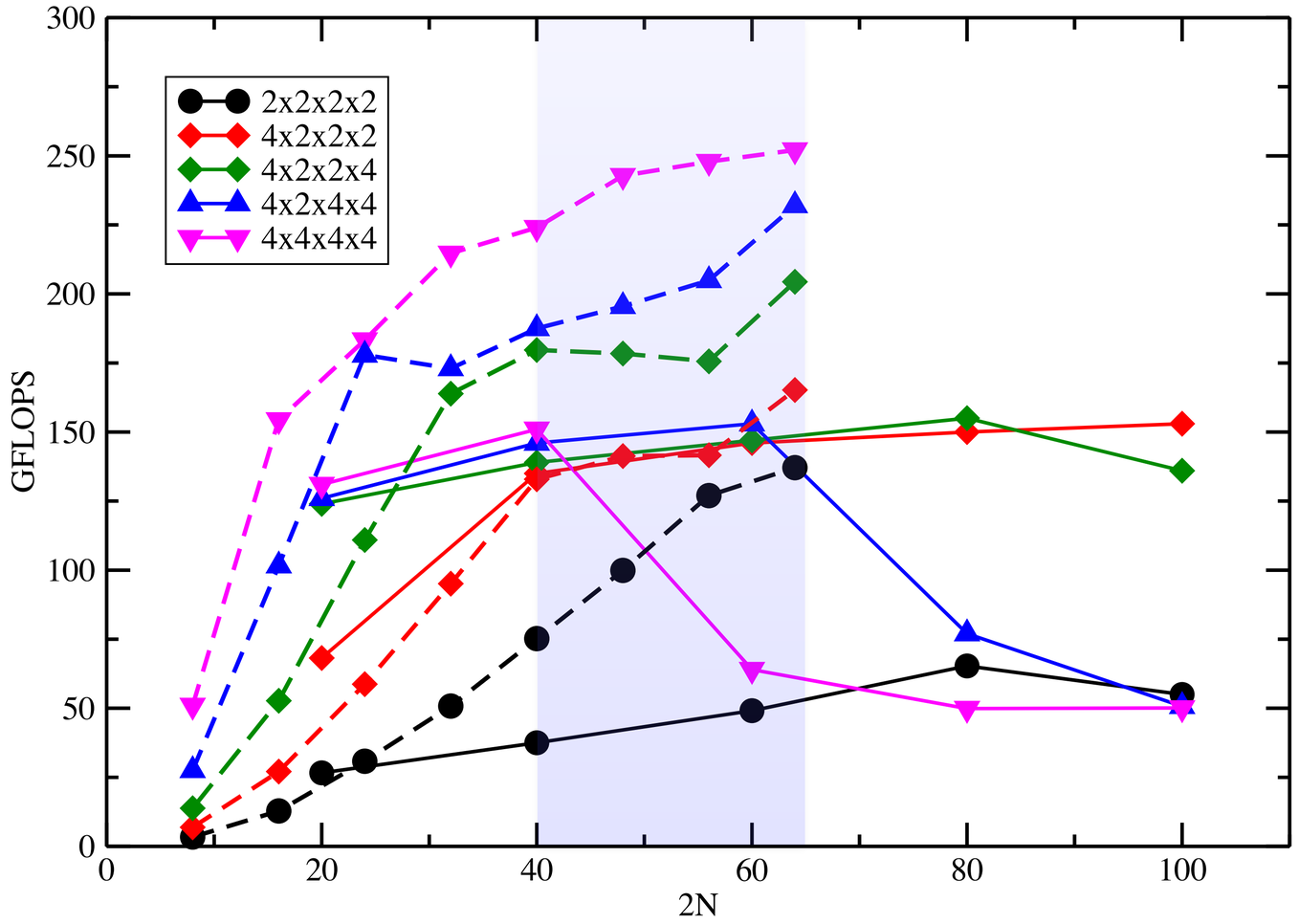
24,576-way parallel

16-way parallel

NVIDIA

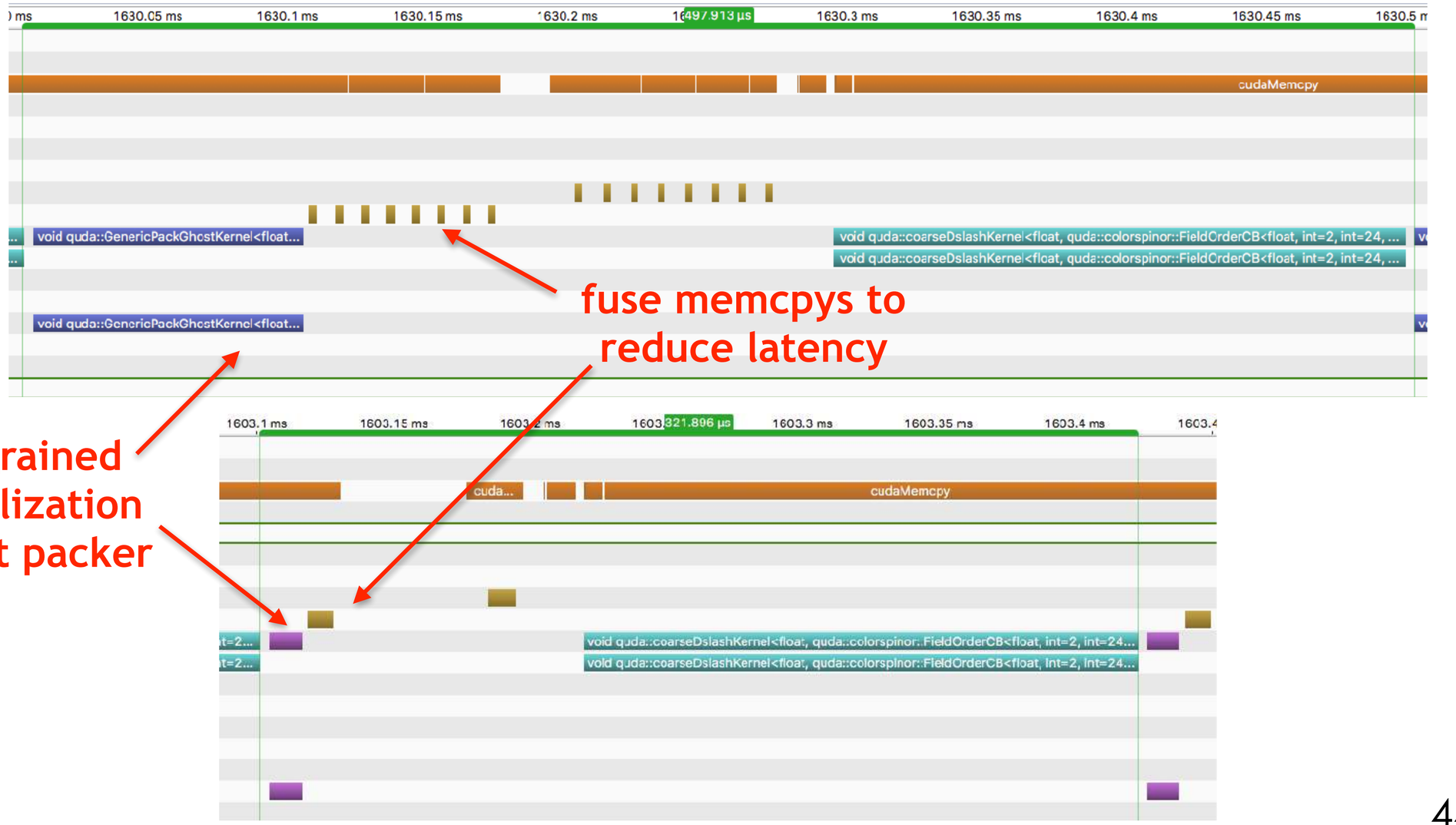
COARSE GRID OPERATOR PERFORMANCE

8-core Haswell 2.4 GHz (solid line) vs M6000 (dashed lined), FP32



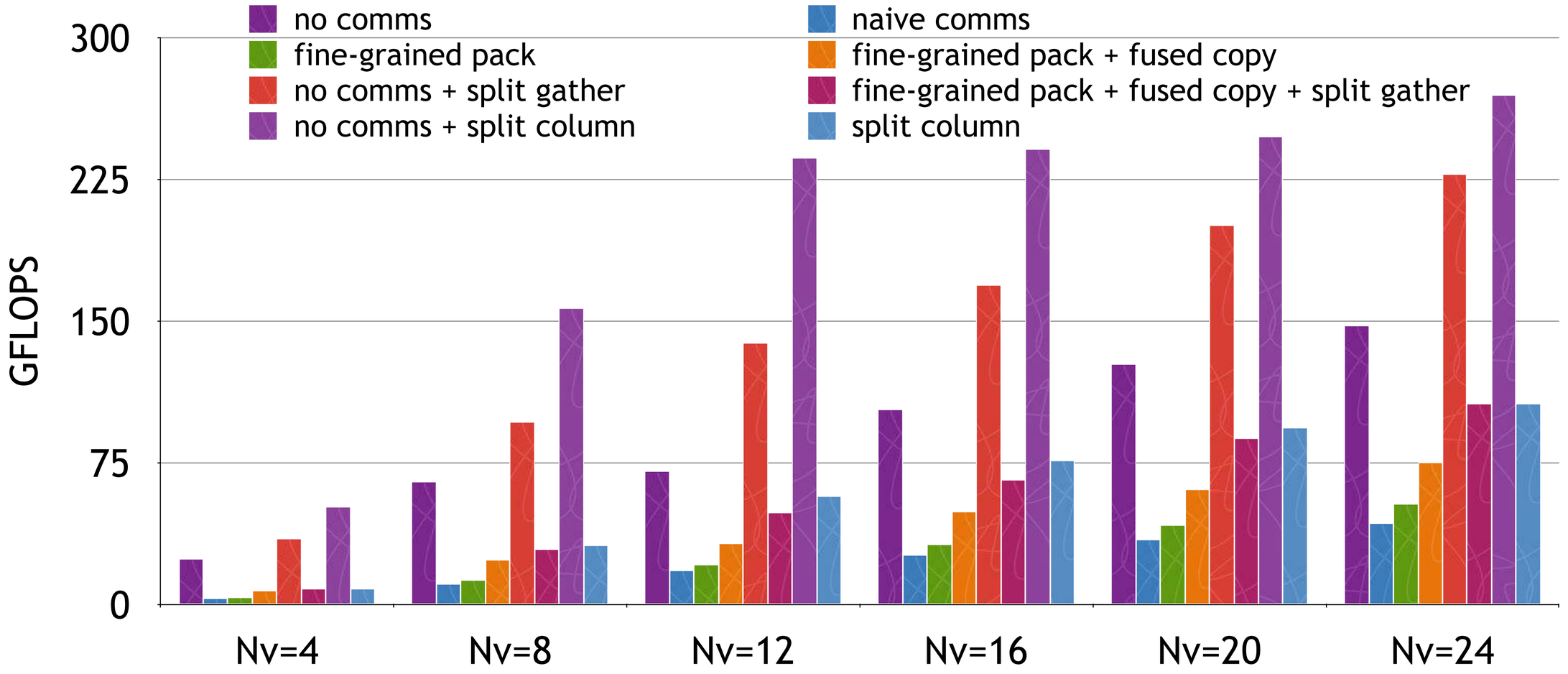
- Autotuner finds optimum degree of parallelization
  - Larger grids favor less fine grained
  - Coarse grids favor most fine grained
- GPU is nearly always faster than CPU
- Expect in future that coarse grids will favor CPUs
- For now, use GPU exclusively

IMPROVING STRONG SCALING



IMPROVING STRONG SCALING

$V_{coarse} = 4^4$, 8-way communication, FP32, Quadro M6000



NVIDIA