CLOVER HMC AND STAGGERED MULTIGRID ON SUMMIT AND VOLTA

Kate Clark, July 25th 2018

PRESENTED BY
NVIDIA.

OUTLINE

Clover HMC Multigrid
- Setup acceleration
- Results
- Improving strong scaling

Staggered Multigrid
- HISQ Algorithm
- Results

with
- Bálint Joó
- Arjun Gambhir
- Mathias Wagner
- Evan Weinberg
- Frank Winter
- Boram Yoon

with
- Rich Brower
- Alexei Strelchenko
- Evan Weinberg

QUDA

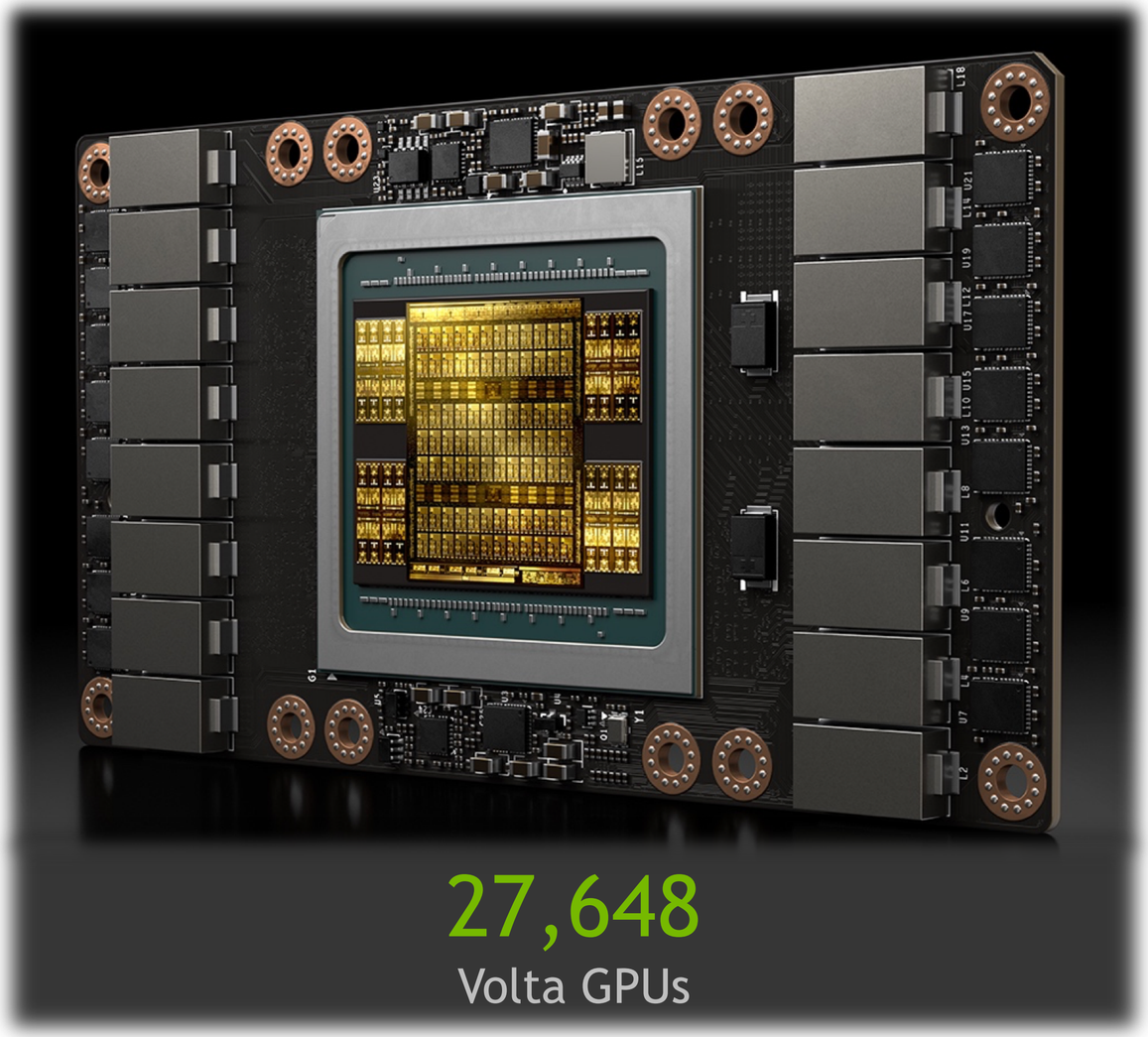
- "QCD on CUDA" – http://lattice.github.com/quda (open source, BSD license)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, tmLQCD, etc.
- Provides:
  - Various solvers for all major fermionic discretizations, with multi-GPU support
  - 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 (additive 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

NVIDIA POWERS WORLD'S FASTEST SUPERCOMPUTER

Summit Becomes First System to Scale the 100 Petaflops Milestone

122 PF
HPC

3 EF
AI



27,648
Volta GPUs

HMC MULTIGRID

STARTING POINT

2+1 flavour Wilson-clover fermions with Stout improvement running on Chroma

Physical parameters:

V = 64^3x128, ml=-0.2416, ms=-0.2050, a~0.09 fm, m_π~170 MeV

Performance measured relative to prior pre-MG optimal approach

- Essentially the algorithm that has been run on Titan 2012-2016
- 3 Hasenbusch ratios, with heaviest Hasenbusch mass = strange quark
  - Represented as 1 + 1 + 1 using multi-shift CG (pure double precision)
- 2-flavour solves: GCR + Additive Schwarz preconditioner (mixed precision)
- All fermions on the same time scale using MN5FV 4^{th} order integrator

Benchmark Time: 1024 nodes of Titan = 4006 seconds

https://github.com/JeffersonLab/qdp-jit

CHROMA + QDP-JIT/LLVM

QDP-JIT/PTX: implementation of QDP++ API for NVIDIA GPUs by Frank Winter (arXiv:1408.5925)

- Chroma builds unaltered and offloads evaluations to the GPU automatically
- Direct device interface to QUDA to run optimized solves

Prior publication covers earlier with direct PTX code generator

- Now use LLVM IR code generator and can target any architecture that LLVM supports

Chroma/QDP-JIT: Clover HMC in production on Titan and newer machines

Latest improvements:

- Caching of PTX kernels to eliminate overheads
- Faster startup times making the library more suitable for all jobs

WHY HMC + MULTIGRID?

HMC typically dominated by solving the Dirac equation

However, much more challenging than analysis

- Few solves per linear system
- Can be bound by heavy solves (c.f. Hasenbusch mass preconditioning)

Build on top of pre-existing QUDA MG (arXiv:1612.07873)

Multigrid setup must run at speed of light since little scope for amortizing

Reuse and evolve multigrid setup where possible

MULTIGRID SETUP

Generate null vectors (BiCGStab, CG, etc. acting on homogenous system)

$$Ax_k = 0, \;\; k = 1 \ldots N, \quad \rightarrow \quad B = (x_1 x_2 \ldots x_n)$$

Block Orthogonalization of basis set

$$B^i = Q^i R^i = V^i B^i_c \qquad B = \sum_i B^i, V = \sum_i V^i$$

QR decomposition over each block

Coarse-link construction (Galerkin projection $D_c = P^\dagger D P$)

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

$$Y^{+\mathrm{b}}_\mu(\hat{x}) = \sum_{x \in \hat{x}} V^\dagger(x) P^{+\mu} U_\mu(x) A^{-1}(y) V(y) \delta_{x,y+\mu} \delta_{\hat{x},\hat{y}+\mu}$$

“backward link”

$$Y^{-\mathrm{f}}_\mu(\hat{x}) = \sum_{x \in \hat{x}} V^\dagger(x) A^{-1}(x) P^{-\mu} U_\mu(x) V(y) \delta_{x,y+\mu} \delta_{\hat{x},\hat{y}+\mu}$$

“forward link”

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

“coarse clover”

HMC MULTIGRID ALGORITHM

Use the same null space for all masses (setup run on lightest mass)

- We use CG to find null-space vectors

Evolve the null space vectors as the gauge field evolves (Lüscher 2007)

- Update the null space when the preconditioner degrades too much on lightest mass

Parameters to tune

- Refresh threshold: at what point do we refresh the null space?
- Refresh iterations: how much work do we do when refreshing?

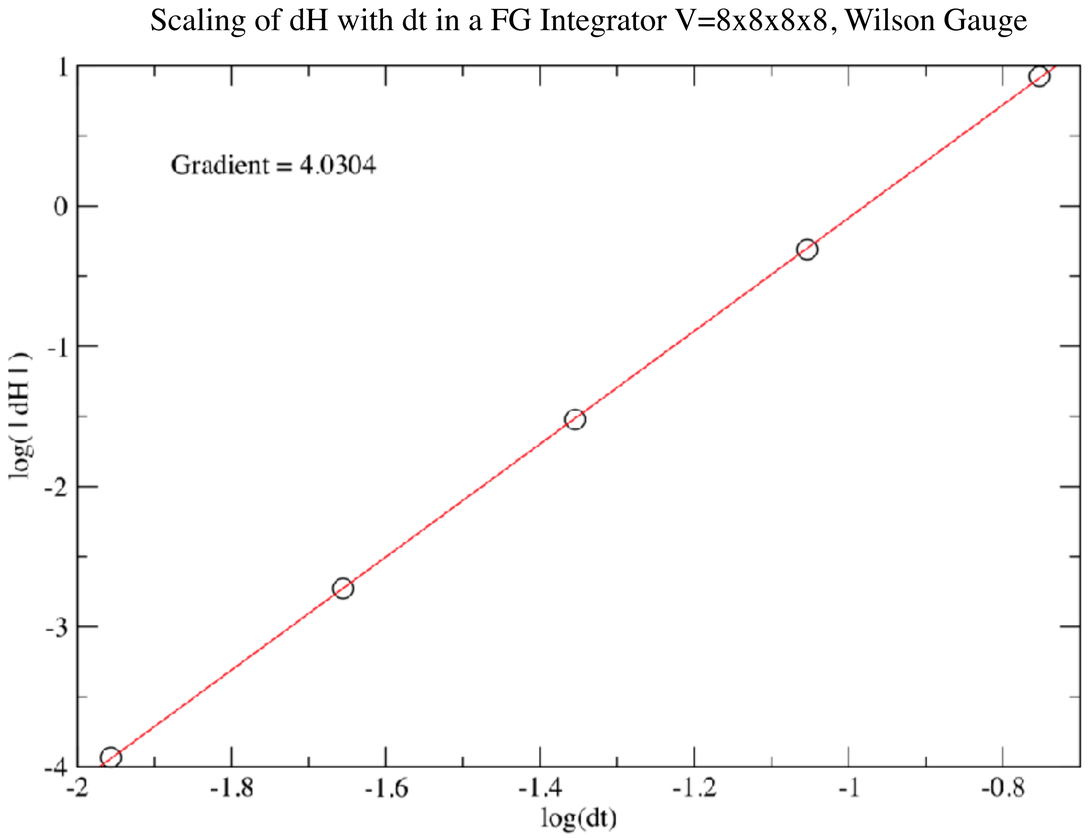
FORCE GRADIENT INTEGRATOR

Standard 4th order integrator following Omelyan requires 5 force evaluations per step (4MN5FV)

Omelyan 2nd order integrator requires 2 force evaluations per step

Force gradient integrator (Clark, Kennedy, Silva) possible with 3 force evaluations + 1 auxiliary force gradient evaluation (Yin and Mawhinney)

- Saves on solves compared to 4MN5FV
- 4th order so volume scaling of cost is $V^{9/8}$



OPTIMIZATION AND TUNING STEPS

(far from exclusive)

Replace GCR+DD with GCR-MG

- Made Hasenbusch terms cheaper so add extra Hasenbsuch term and retuned
- Put heaviest fermion doublet onto the fine (gauge) time scale

Optimize mixed-precision multigrid method:

- 16-bit precision wherever it makes sense (null space, coarse link variables, halo exchange)

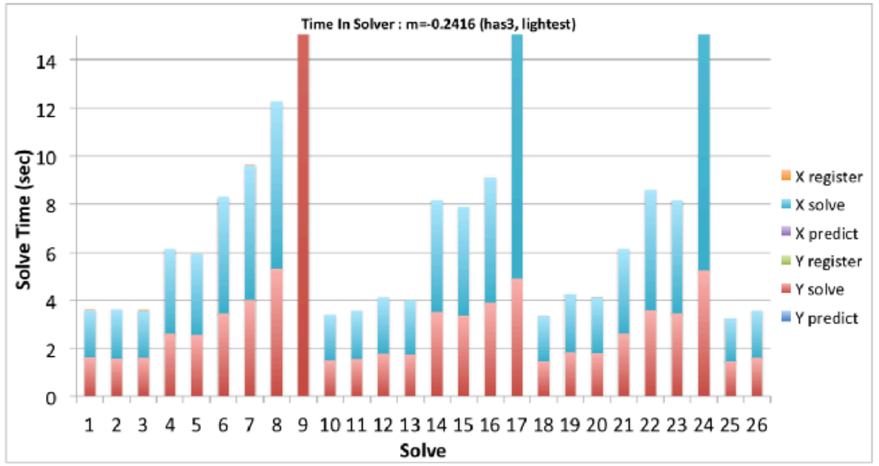
Volta 4x faster than Pascal for key setup routines: use multigrid for all 2-flavour solves

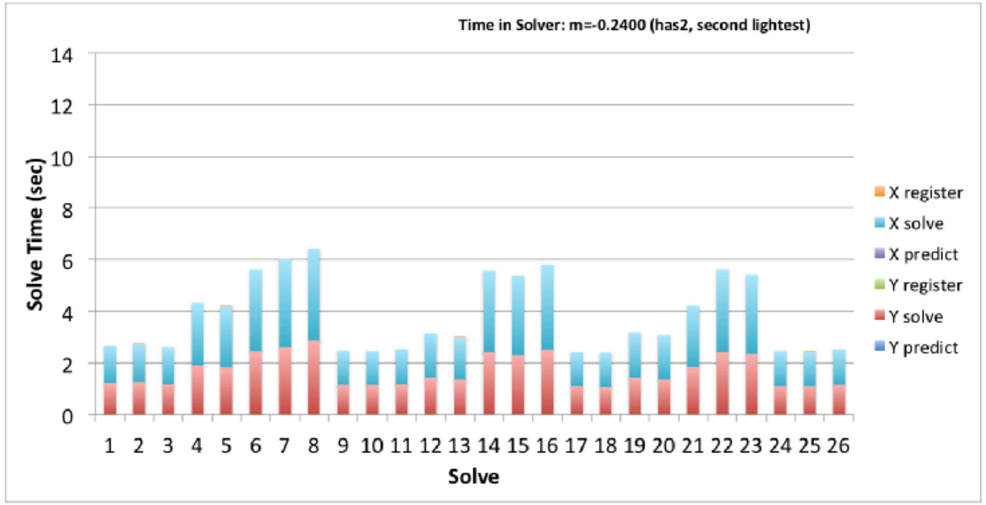
Replaced MN5FV integrator with Force Gradient integrator, tuned number of steps

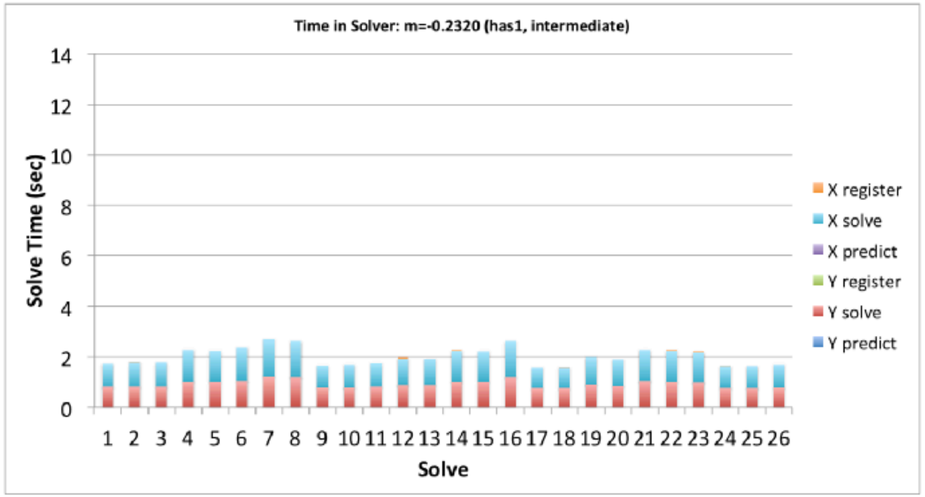
Multi-shift CG is expensive (no multigrid - yet…)

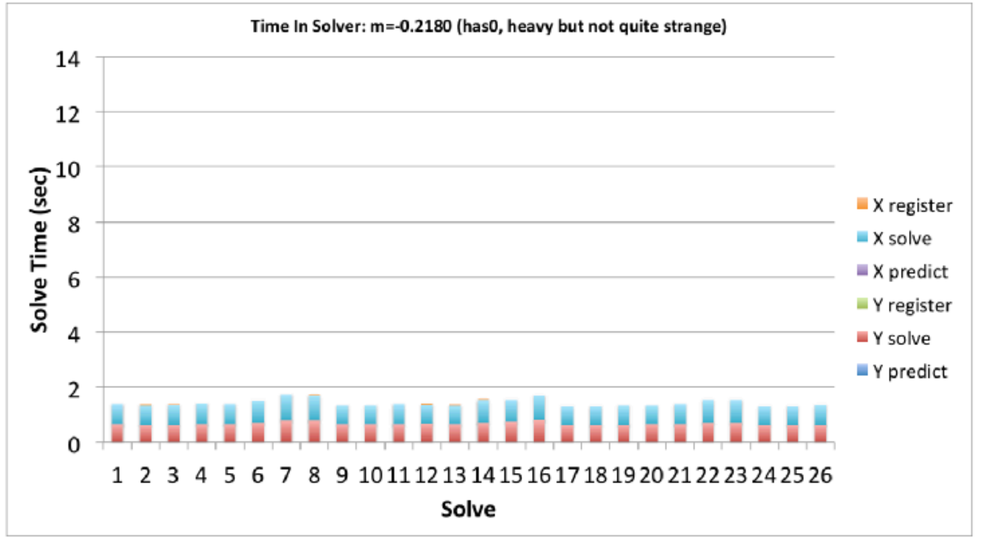
- Replace pure fp64 multi-shift CG with mixed-precision multi-shift CG and refinement: 1.5x faster

NULL-SPACE EVOLUTION

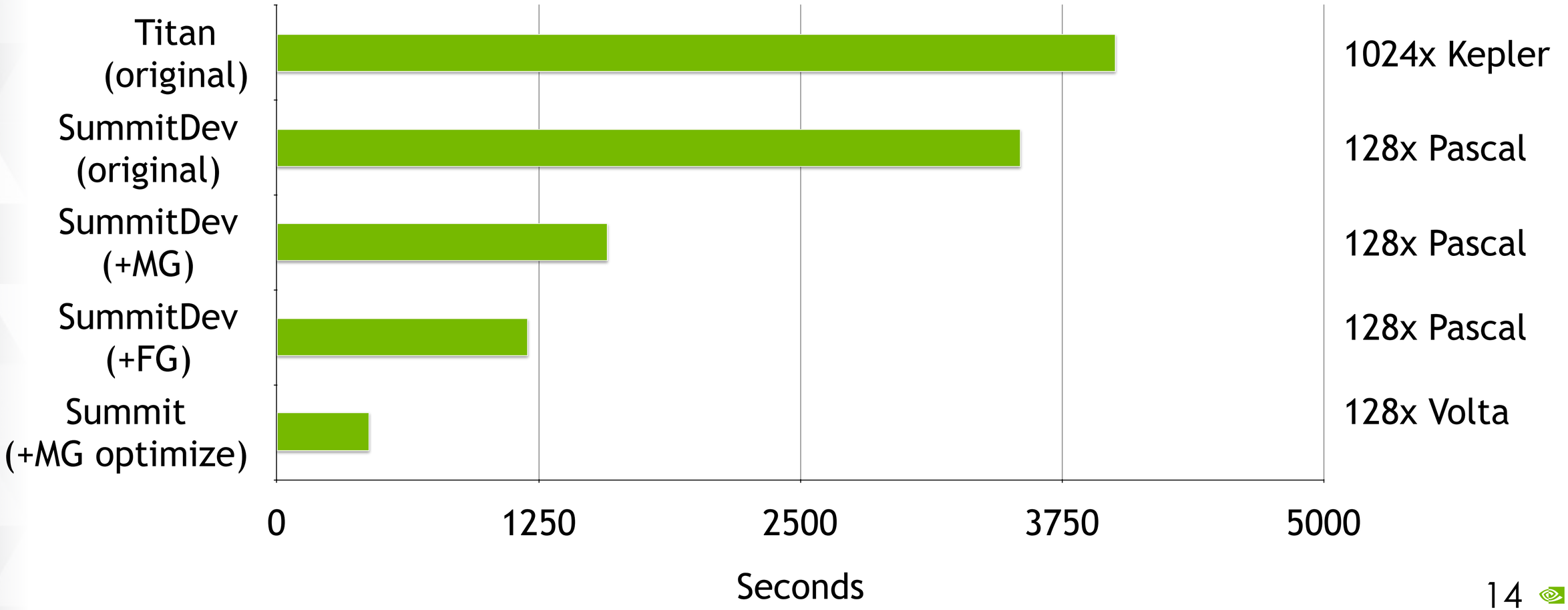




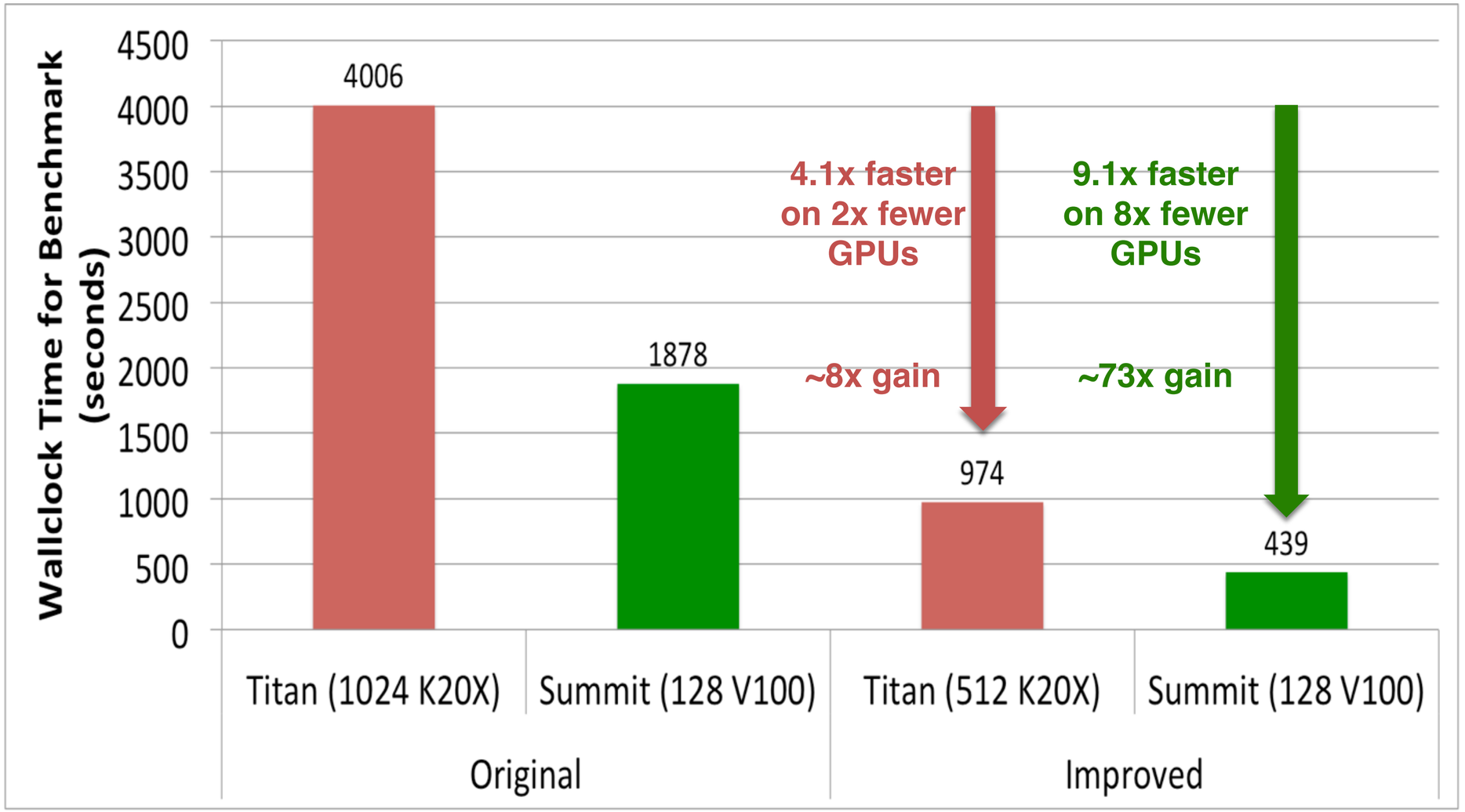




HMC SPEEDUP PROGRESSION



LATEST RESULTS



WORK IN PROGRESS TO GET TO >100X

Network bandwidth limited for halo exchange on Summit

- Deploy 8-bit precision for halo exchange in smoother
- Close to 2x reduction in nearest-neighbor network traffic
- Initial testing shows negligible effect on convergence

Latency limited by global reductions

- Replace MR smoother and bottom GCR solver with communication avoiding GCR (CA-GCR)
- >6x decrease in number of global reductions
- >20% speedup on workstation, ~~expect much bigger gain on 100s GPUs~~ **40% speedup at Titan 512 nodes**

Use multi-rhs null-space generation, e.g., 24x CG => 1x block CG on 24 rhs

Cannot coarsen beyond 2^4 coarse grid points per MPI process presenting hard limit on scaling

HMC MULTIGRID SUMMARY

2018 Chroma gauge generation close to 100x increase in throughput vs 2016

- Multigrid solver
- Force gradient integrator and MD tuning
- Titan -> Summit (Kepler to Volta)

Work continues to further improve this…

STAGGERED MULTIGRID

STAGGERED MULTIGRID

Last year we presented our work on developing a staggered MG algorithm in 2-d

We have now extended this to 4-d and implemented it in QUDA

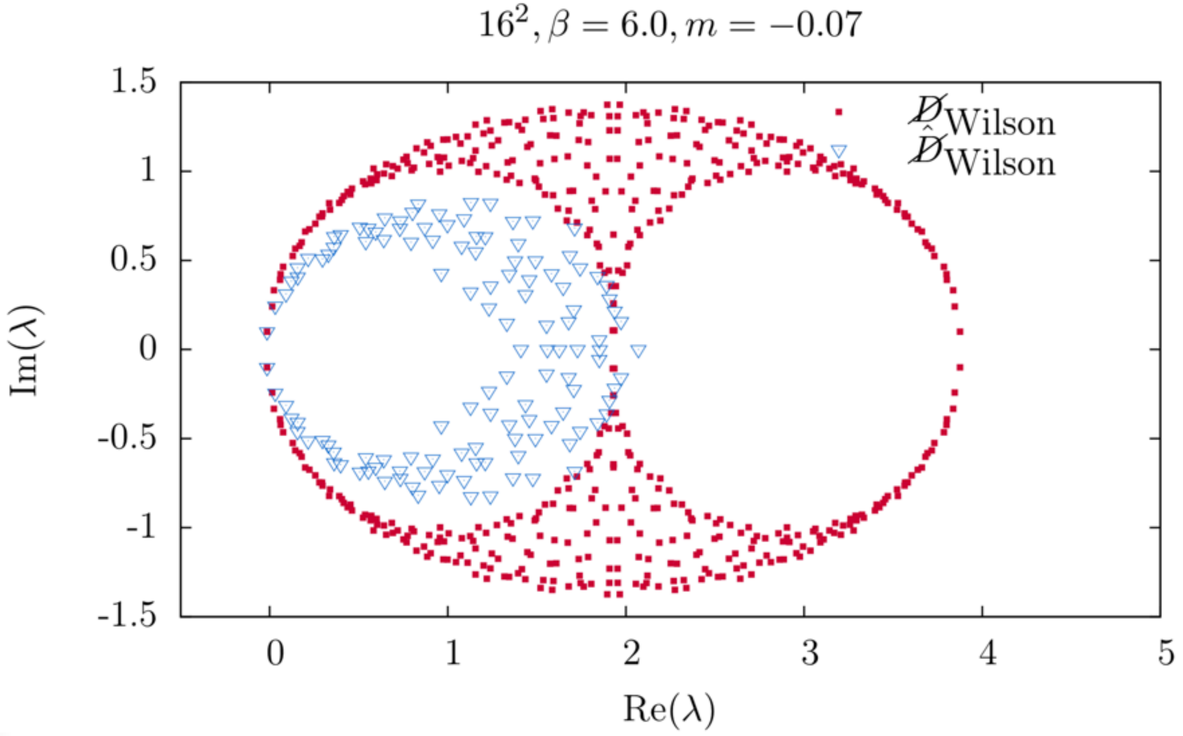
How well does this work?

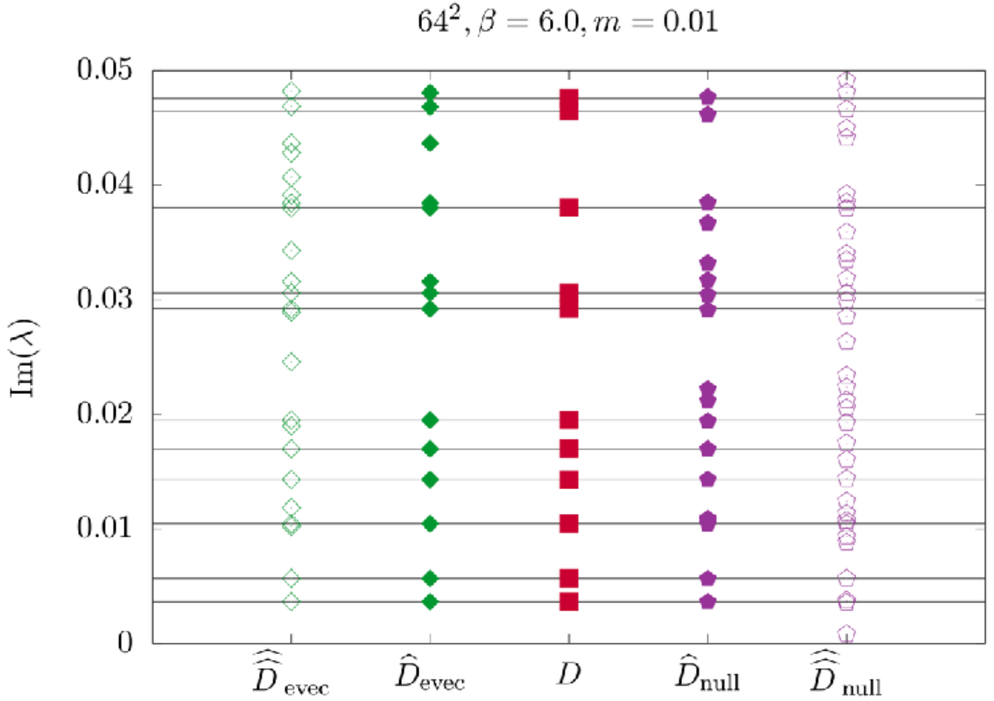
arXiv:1801.07823

WHAT MAKES STAGGERED MG HARD?

Naïve Galerkin projection does not work

- Spurious low modes on coarse grids
- System gets worse conditioned as we progressively coarsen





Compare to Wilson MG which preserves low modes with no cascade

arXiv:1801.07823

OUR SOLUTION

Staggered fermions distribute d fermions over 2^d sites

Each 2^d block is a supersite

or flavour representation or Kahler-Dirac block (arXiv:0509026 Dürr)

$$\mathcal{S} = b^4 \sum_{X,\mu} \bar{q}(X) \left[\nabla_\mu \left(\gamma_\mu \otimes 1 \right) - \frac{b}{2} \triangle_\mu \left(\gamma_5 \otimes \tau_\mu \tau_5 \right) + m \left(1 \otimes 1 \right) \right] q(X)$$

$$\equiv b^4 \sum_{X,\mu} \bar{q}(X) \left[\not{D} + m \right] q(X)$$

$$\left(\nabla_\mu q \right)(X) = \frac{q(X + b\hat{\mu}) - q(X - b\hat{\mu})}{2b}$$

$$\left(\triangle_\mu q \right)(X) = \frac{q(X + b\hat{\mu}) - 2q(X) + q(X - b\hat{\mu})}{b^2}$$

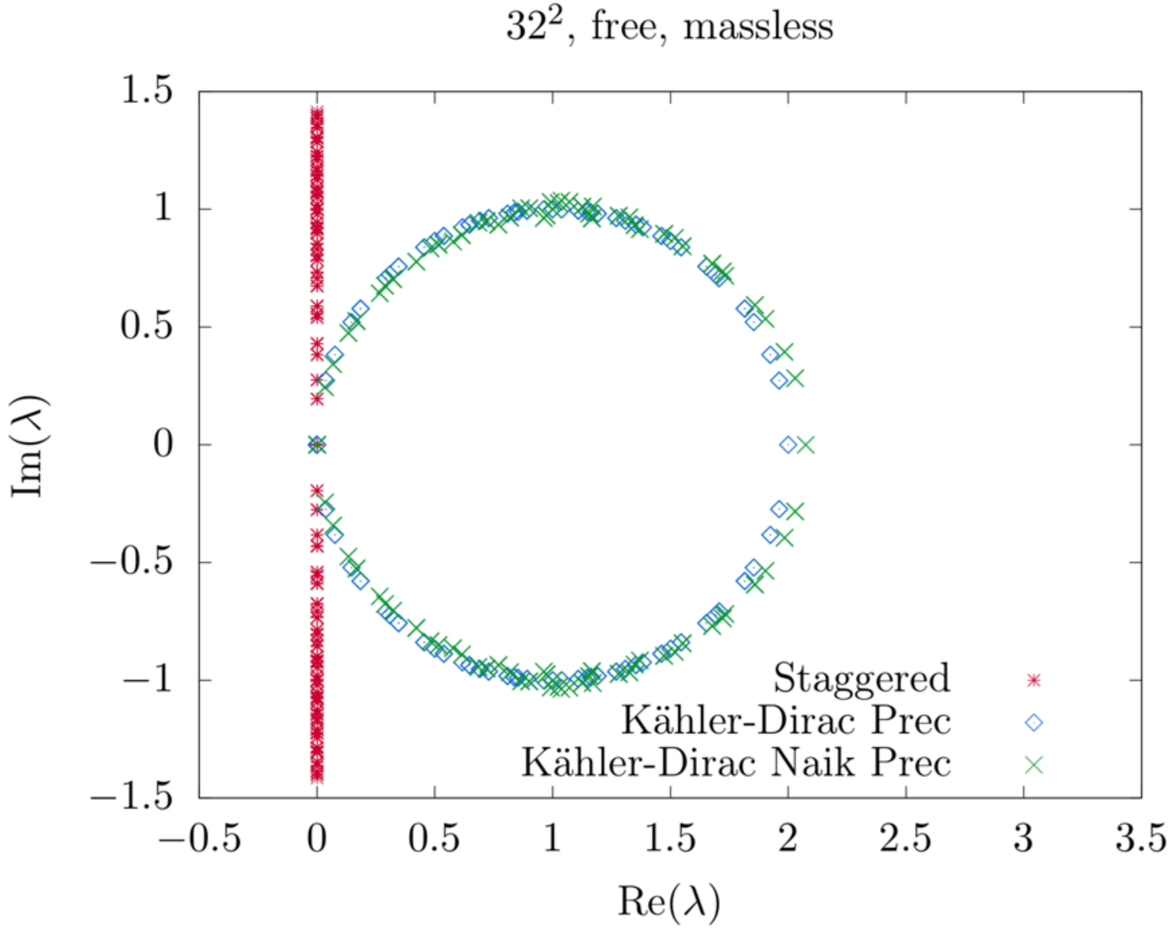
arXiv:1801.07823

OUR SOLUTION

Transform into Kahler-Dirac form through unitary transformation

$$\begin{pmatrix} m & 0 & -\frac{1}{2}U_x(2\vec{n}) & -\frac{1}{2}U_y(2\vec{n}) \\ 0 & m & -\frac{1}{2}U_y^\dagger(2\vec{n}+\hat{x}) & \frac{1}{2}U_x^\dagger(2\vec{n}+\hat{y}) \\ \frac{1}{2}U_x^\dagger(2\vec{n}) & \frac{1}{2}U_y(2\vec{n}+\hat{x}) & m & 0 \\ \frac{1}{2}U_y^\dagger(2\vec{n}) & -\frac{1}{2}U_x(2\vec{n}+\hat{y}) & 0 & m \end{pmatrix}$$

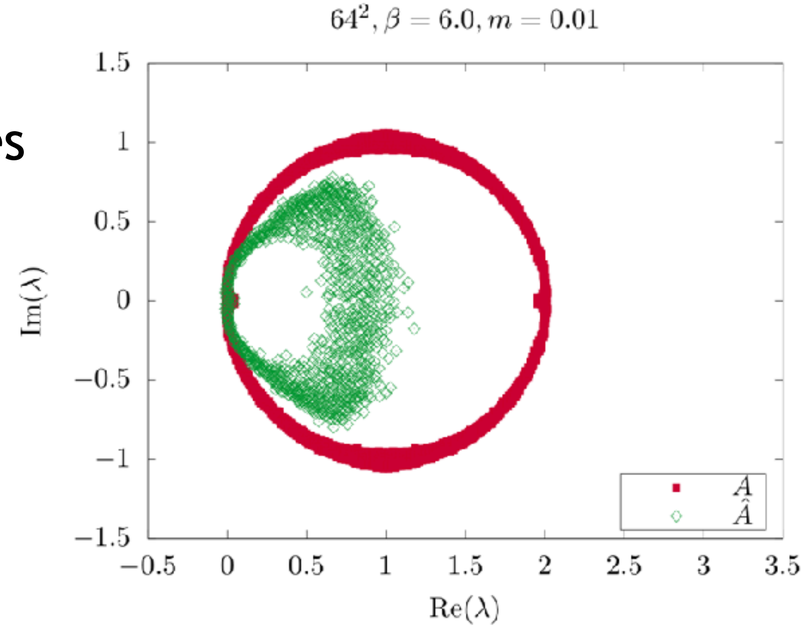
“Precondition” the staggered operator by the Kahler-Dirac block

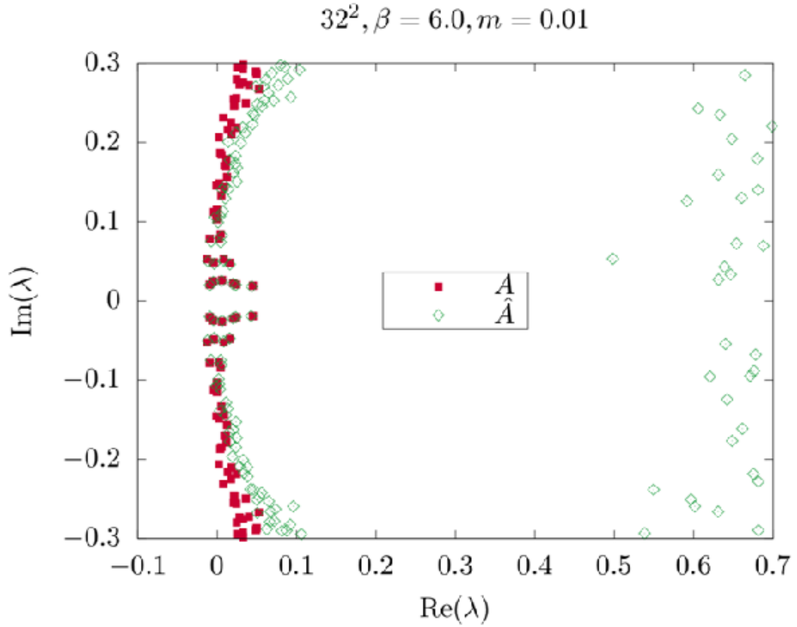


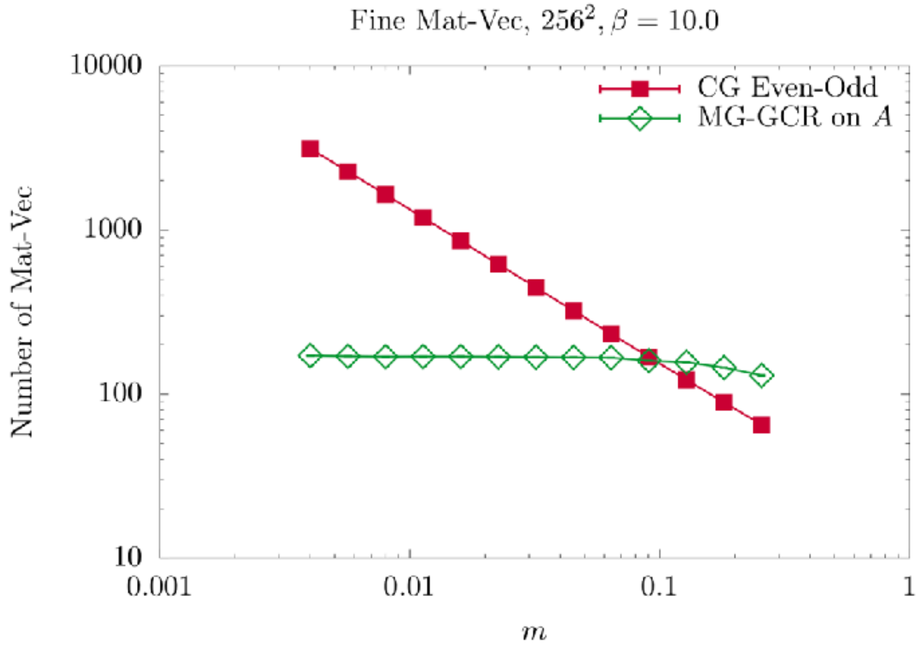
NVIDIA.

arXiv:1801.07823

No spurious low modes as we coarsen







Removal of critical slowing down

GOING TO 4D AND HISQ FERMIONS

Block-preconditioned operator is no longer an exact circle

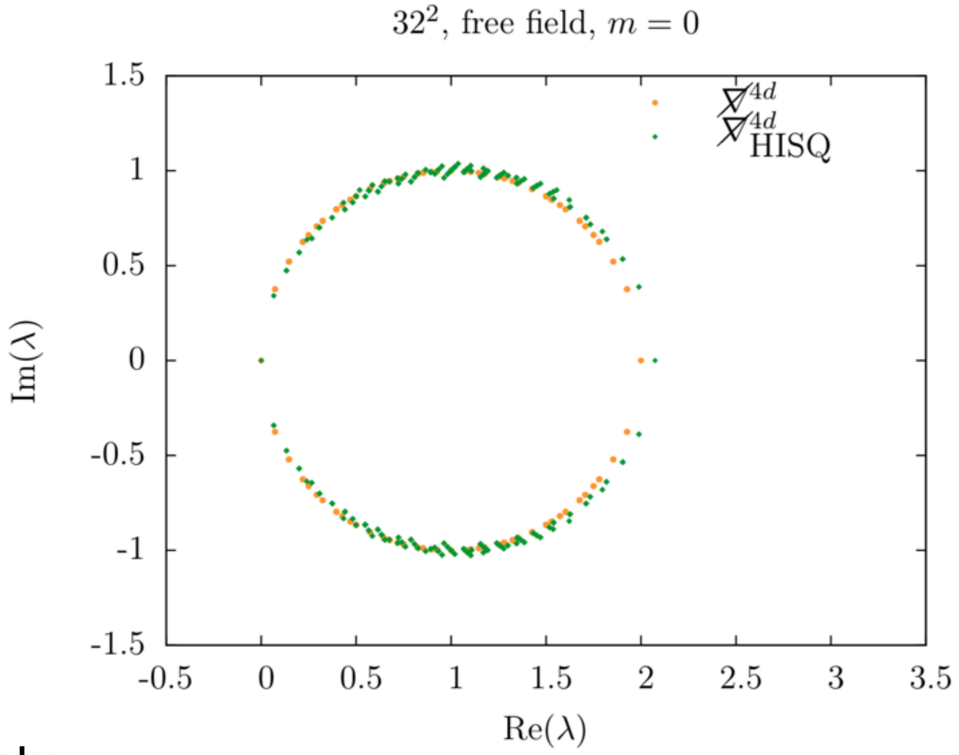
Prescription is *almost* identical to 2-d method

Drop Naik contribution from block preconditioner

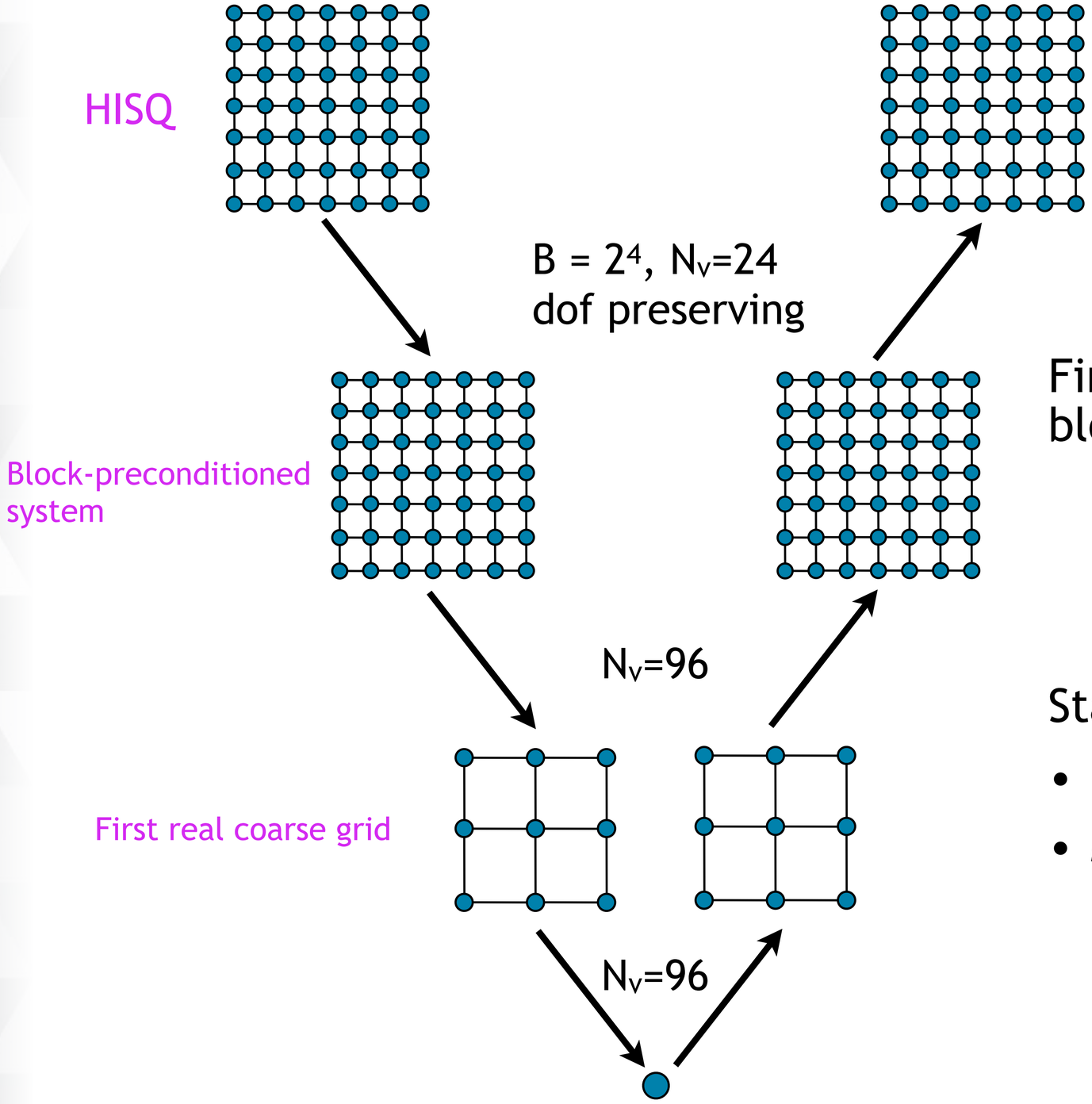
- No longer a unitary transformation
- No longer an exact Schur complement

Iterate between HISQ operator and block-preconditioned system

- Effectively apply MG to fat-link truncated HISQ operator only



HISQ MG ALGORITHM



First “coarsening" is transformation to block-preconditioned system

Staggered has 4-fold degeneracy

- Need 4x null space vectors (N_v=24 -> 96)
- Much more memory intensive

HISQ MG RESULTS

Very preliminary

SU(3) pure-gauge with V = 32^3x64 and V = 48^3x96, a variety of β

All tests come from running on QUDA running Prometheus cluster

- 16 GPUs for 32^3x64, 96 GPUs for 48^3x96

Setup:

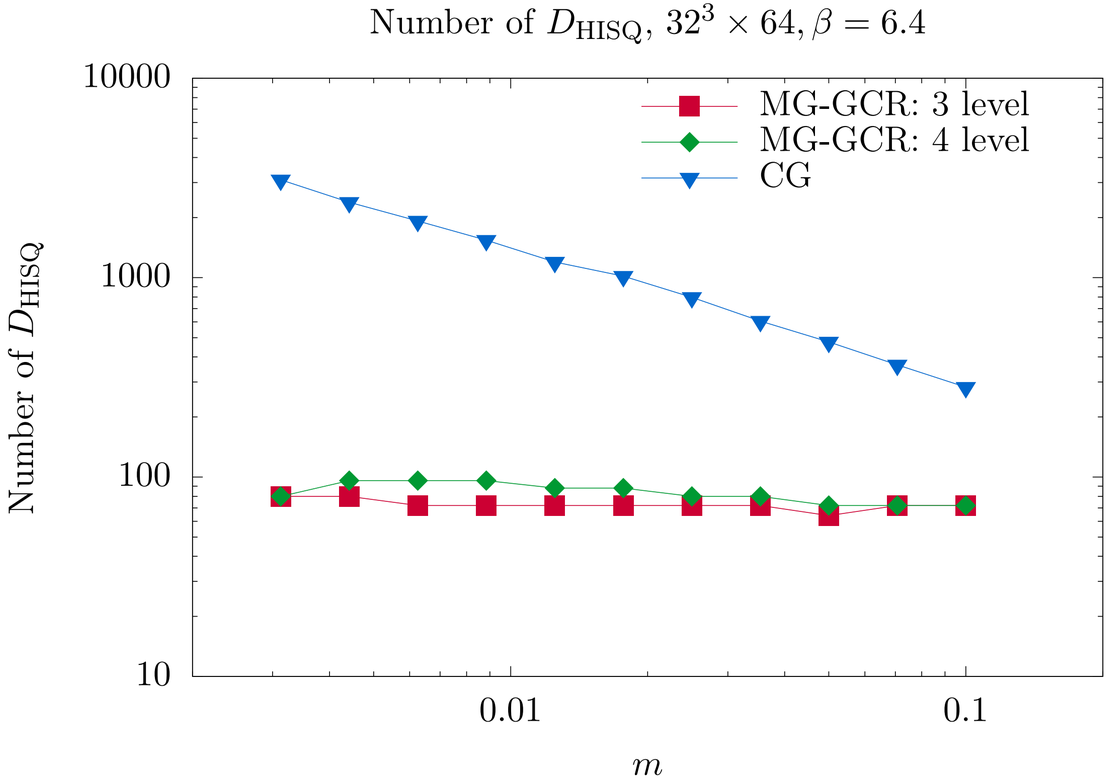
- CGNR
- tolerance 10^{-5}

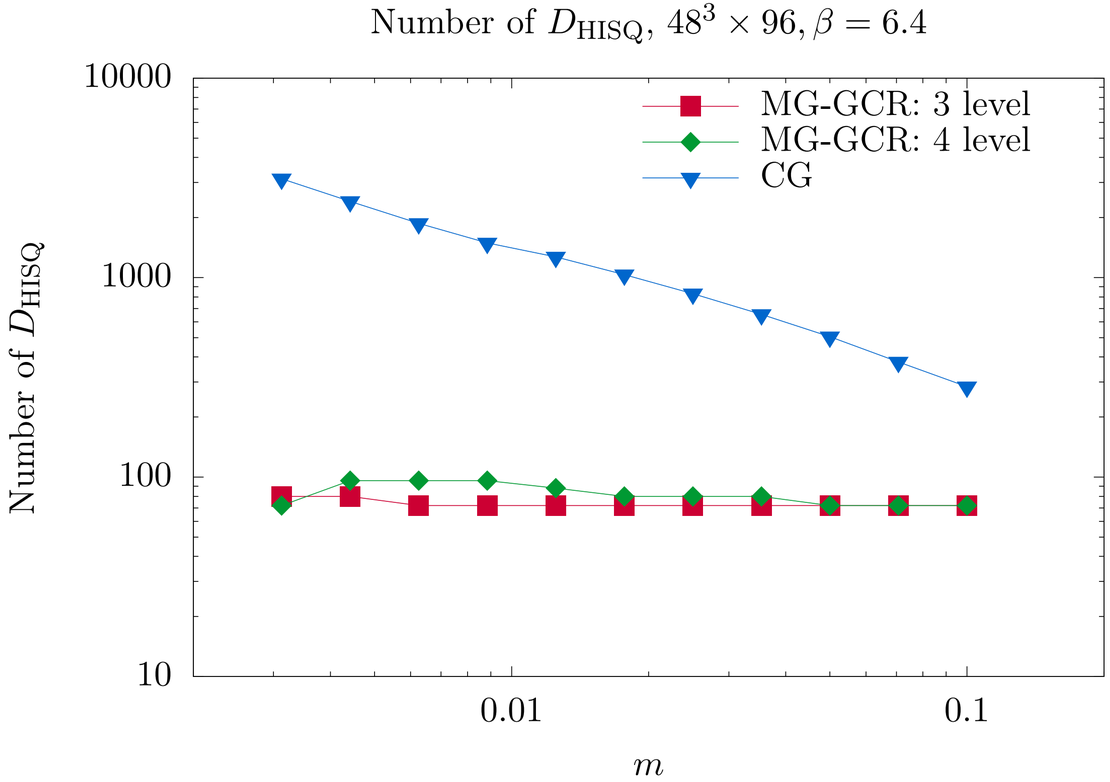
Solver Parameters

	Solver	*Smoother*	*Volume*		*tol*
			Small	*Large*	
level 1	*GCR*	*CA-GCR(0,6)*	*32^3x64*	*48^3x96*	*10^{-12}*
level 2	*GCR*	*CA-GCR(0,6)*	*16^3x32*	*24^3x48*	*0.05*
level 3	*GCR*	*CA-GCR(0,6)*	*8^3x16*	*8^3x24*	*0.25*
level 4	*CGNE*	*-*	*4^3x8*	*4^3x24*	*0.25*

FINE GRID

Level 1

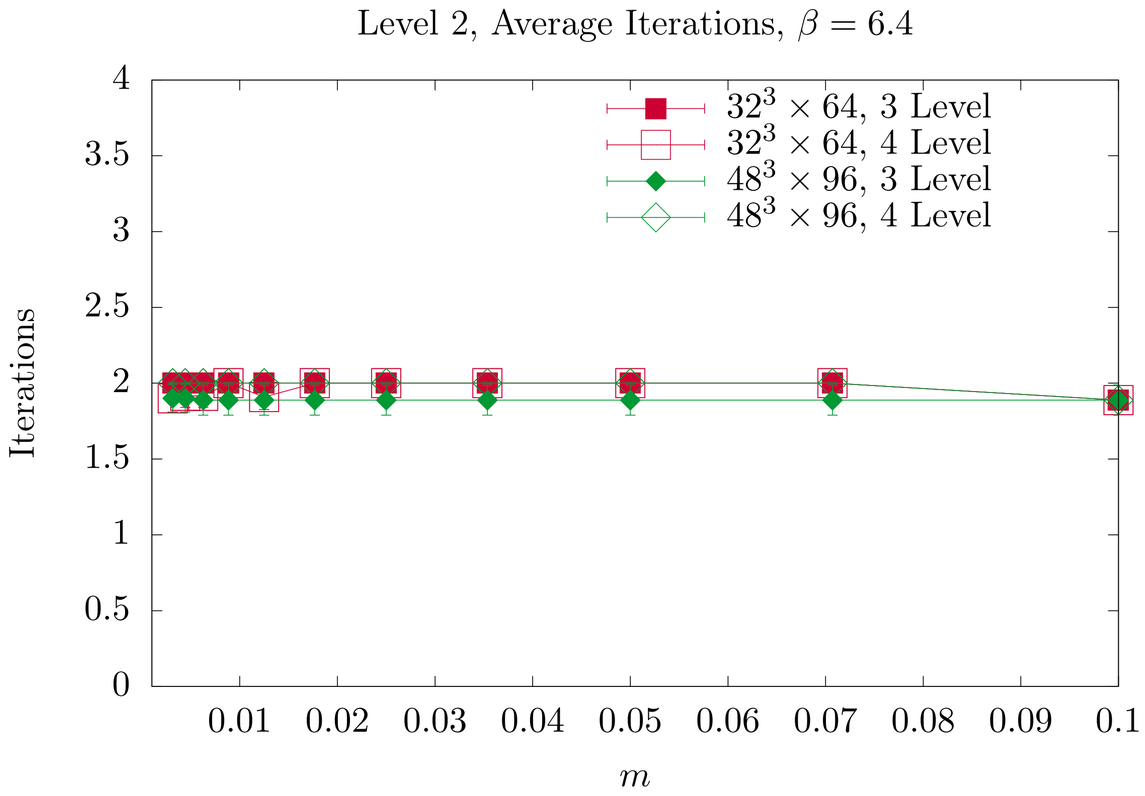


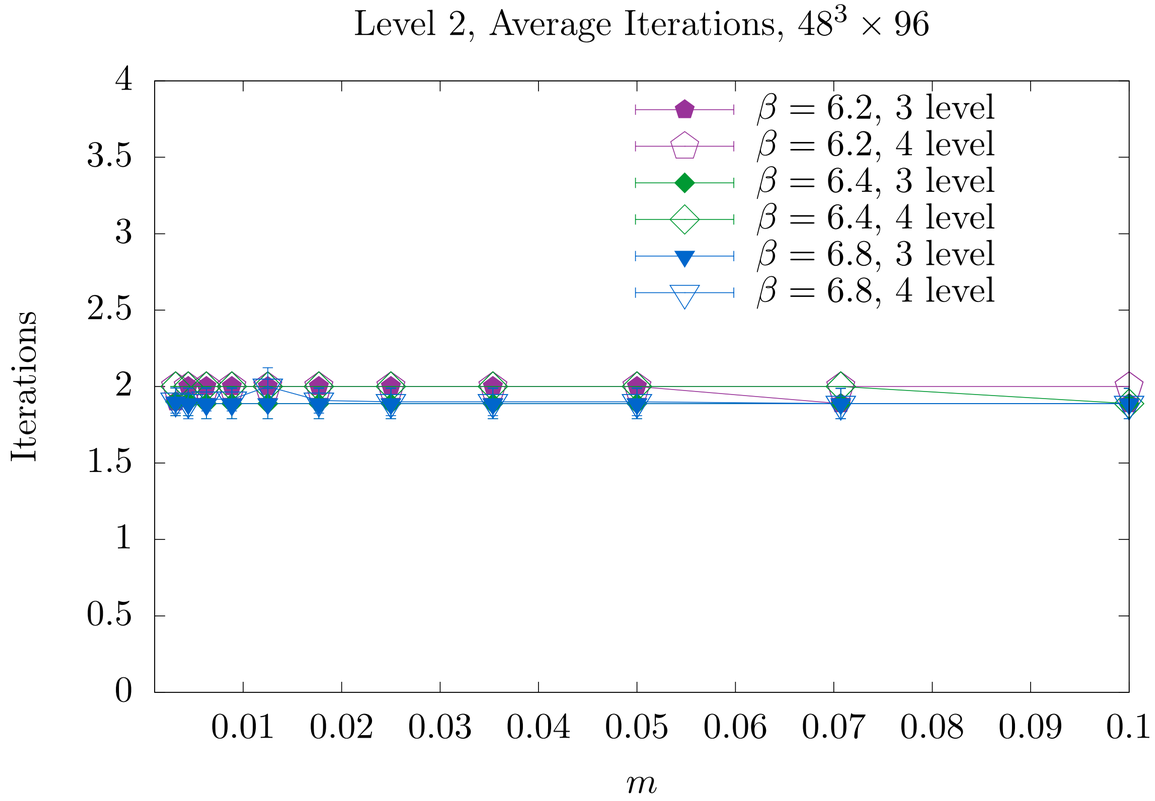


Zero quark mass dependence in fine grid

BLOCK PRECONDITIONER

Level 2

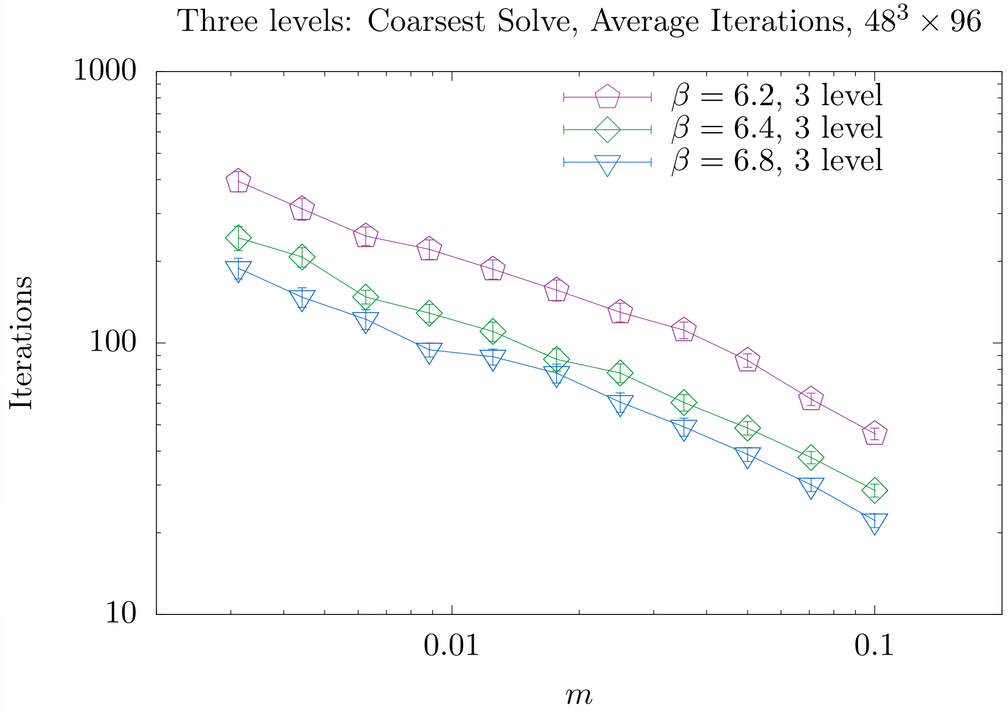


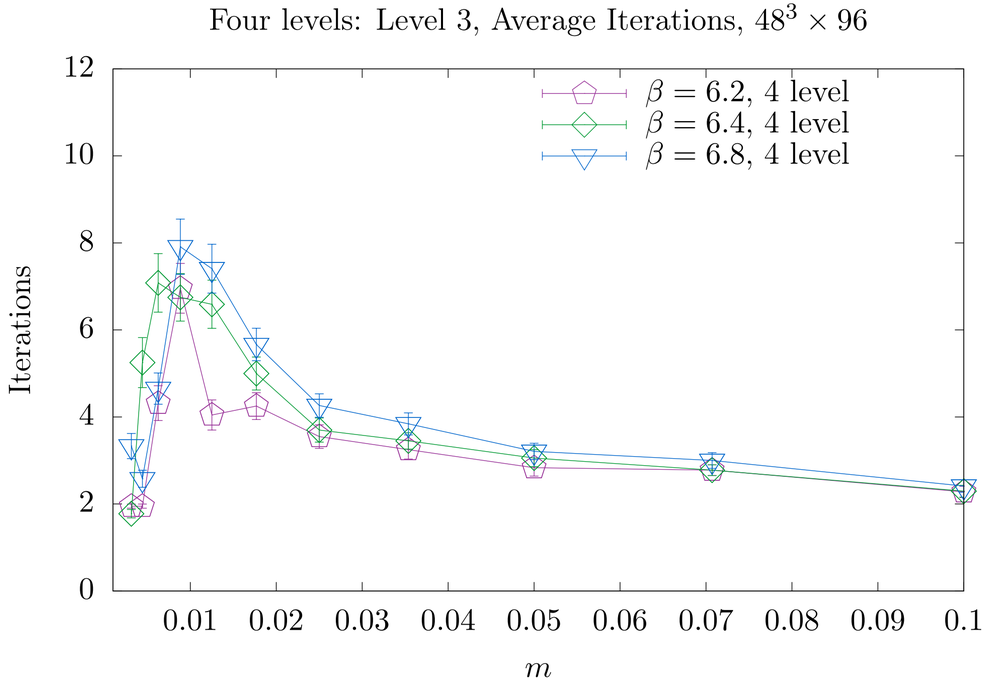


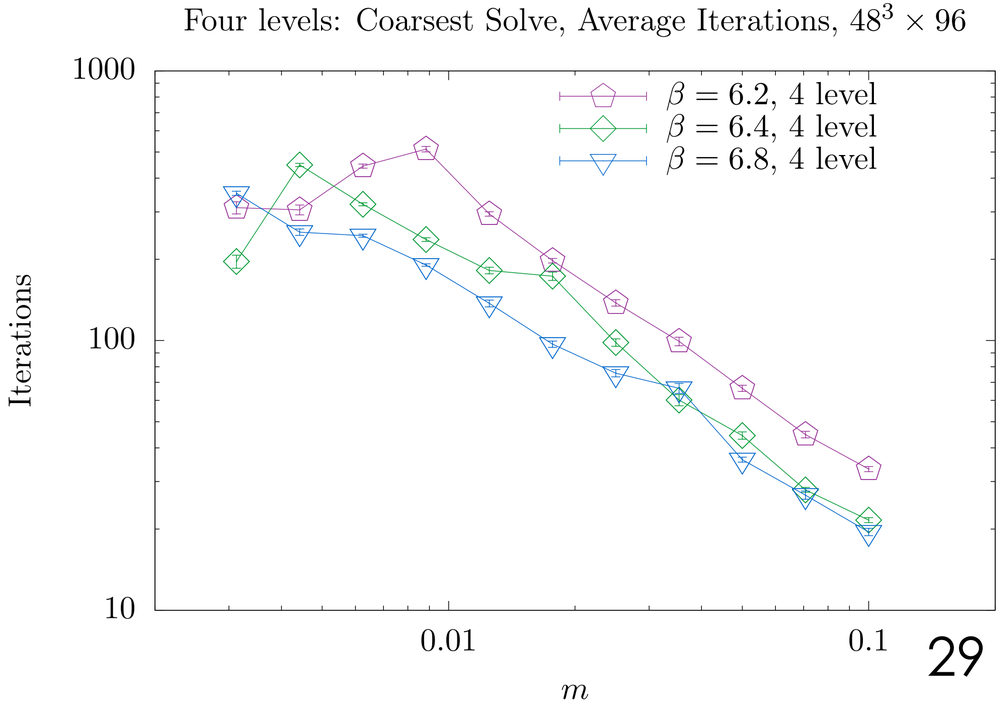
Zero quark mass dependence in block preconditioner

COARSE GRIDS

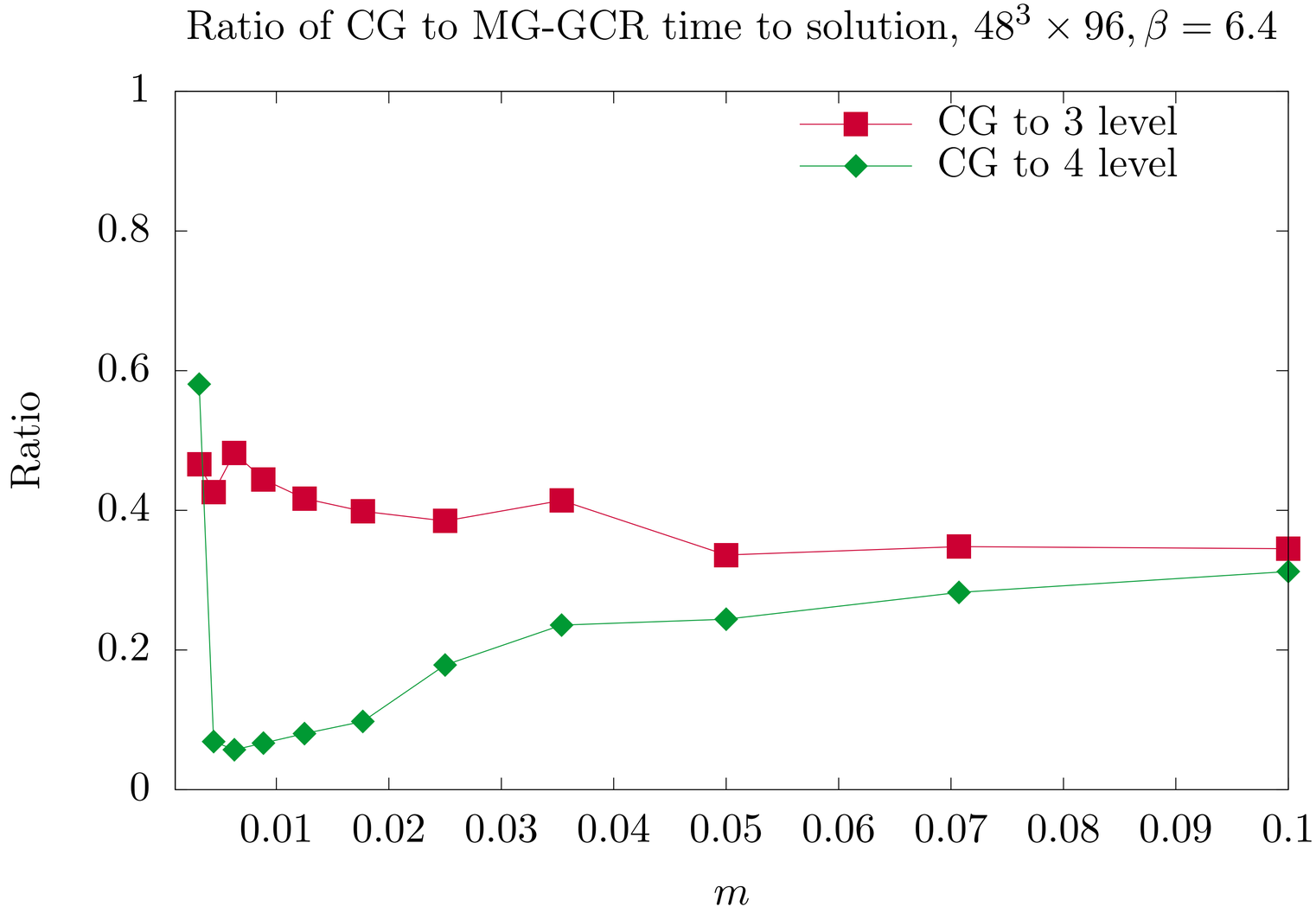
Levels 3 and 4





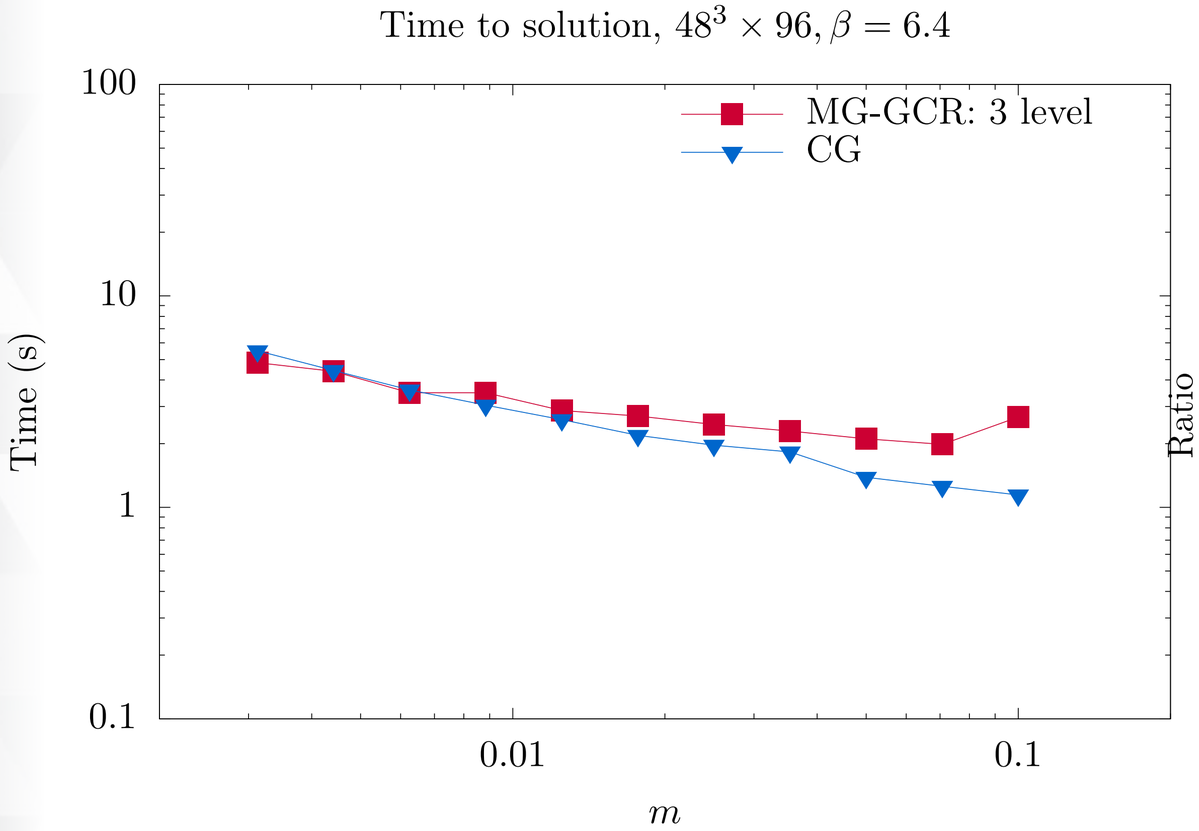


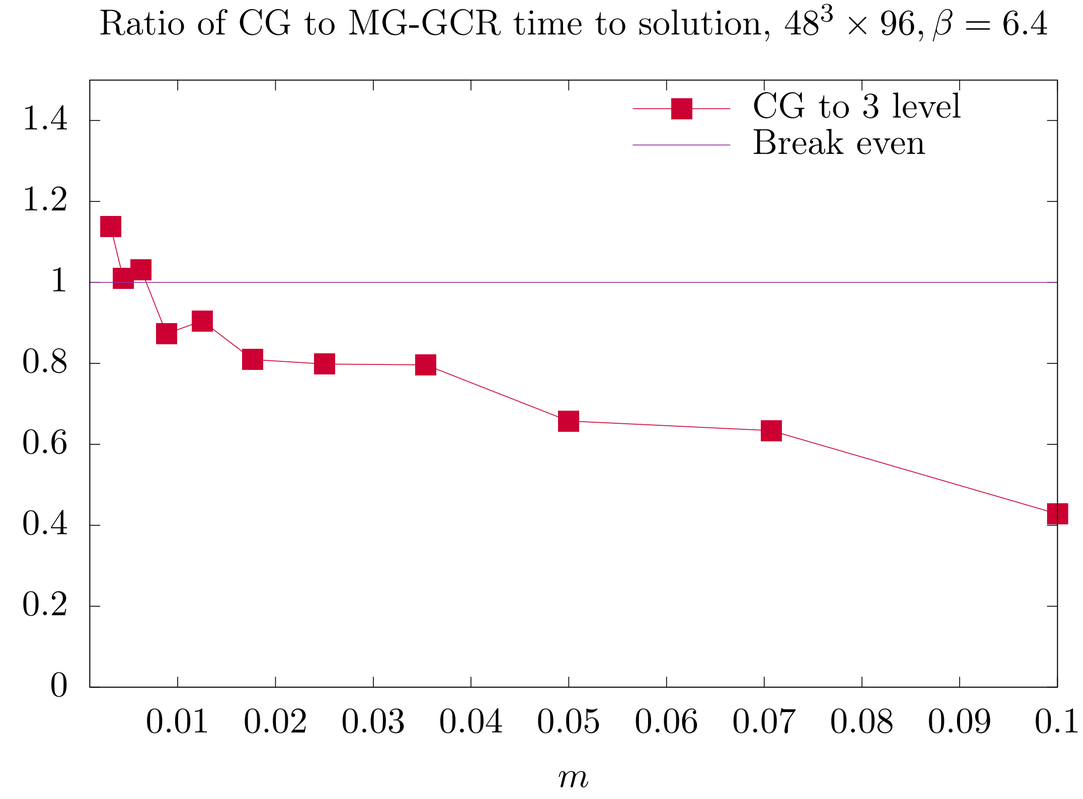
SPEEDDOWN


Ratio of CG to MG-GCR time to solution, $48^3 \times 96, \beta = 6.4$
CG to 3 level
CG to 4 level
Ratio
1
0.8
0.6
0.4
0.2
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
m

SPEEDUP!

switch on even-odd preconditioning





STAGGERED MULTIGRID SUMMARY

Our 2-d staggered multigrid algorithm works in 4-d with HISQ fermions

- Removal of mass dependence from the fine grid and block preconditioner
- No need to include Naik contribution when coarsening

Not much actual speedup yet…

Next steps

- More robust adaptive setup to deal large null space required
- Better approach to bottom solver (deflation, direct solve, etc.)

BACKUP

U.S. BUILT TWO FLAGSHIP SUPERCOMPUTERS

Powered by the Tesla Platform

100-300 PFLOPS Peak

10x in Scientific App Performance

IBM POWER9 CPU + NVIDIA Volta GPU

NVLink High Speed Interconnect

49 TFLOPS per Node, 4608 Nodes

Major Step Forward on the Path to Exascale

COMMUNICATION-AVOIDING GCR

source vector b, solution vector x

while (i<N) {

p_{i+1} <- A*p_i // build basis (N mat-vecs)

$q_i = p_{i+1}$

}

// minimize residual solving (one “blas-3” reduction)

$\psi = (q, q)^{-1} (q, b)$

// update solution vector (one “blas-2” kernel)

$x = \Sigma_k \psi_k p_k$

Similar to CA-GMRES (see Mark Hoemmen’s thesis)

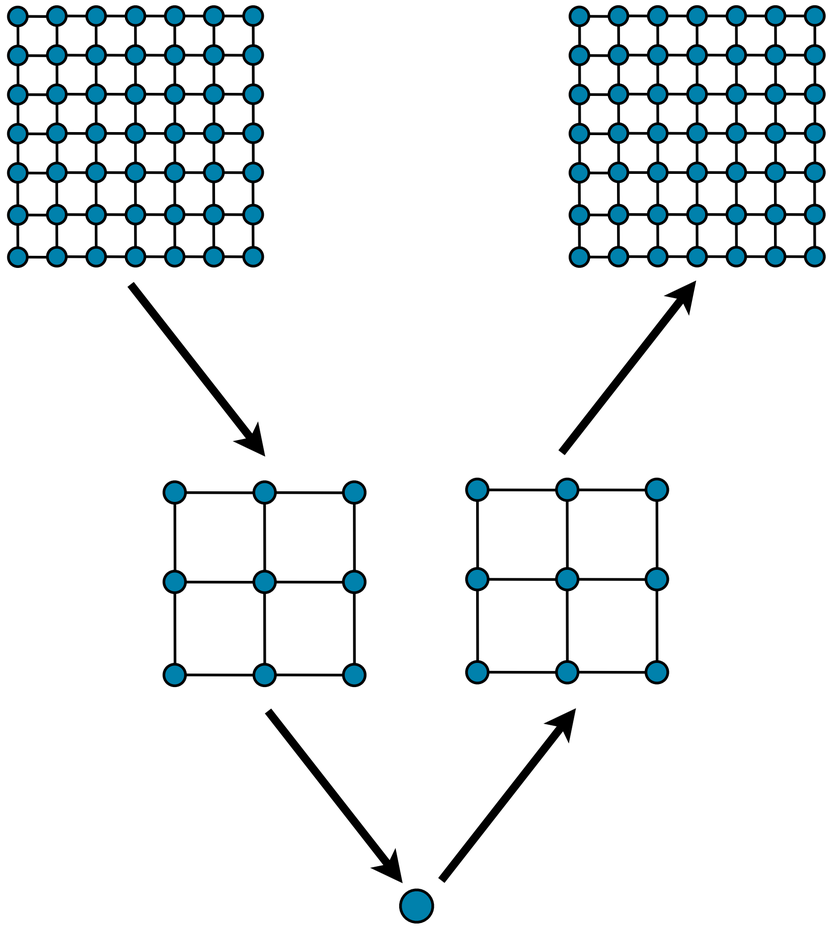
GCR(N) uses modified Gram Schmidt to orthonormalize the basis at every step

- Hence N(N-1)/2 reductions

Instead use classical Gram Schmidt and orthonormalize every N steps

- One reduction every N steps

GLOBAL SYNCHRONIZATIONS IN LQCD MG



Example GCR-MG

- 24x24x24x64 Wilson lattice
- Running at critical point

MR(0,8) smoother with GCR coarse grid solver

- 980 reductions to reach convergence

MR(0,8) smoother, with pipelined GCR

- 829 reductions to reach convergence

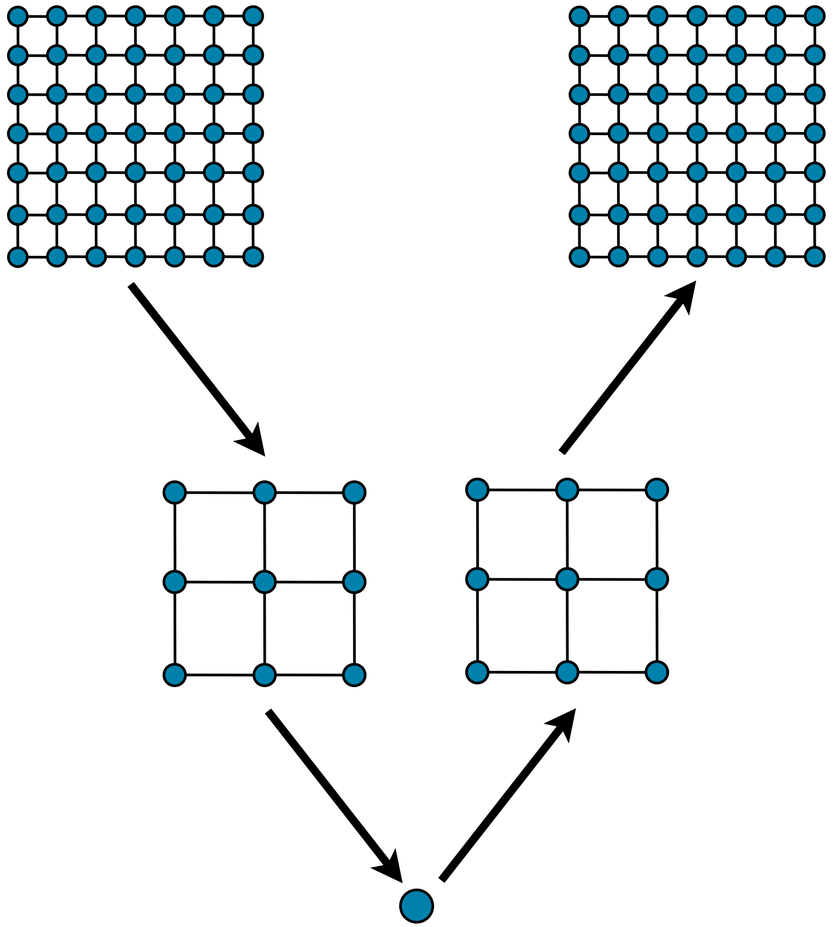
CA-GCR(0,8) for smoother and coarse-grid

- 153 reductions to reach convergence
- >6x reduction in reductions

20% faster on a single workstation

How much faster on Titan / Summit?

GLOBAL SYNCHRONIZATIONS IN LQCD MG



Non-Hermitian system

- No guarantee of convergence
- Use a K-cycle for solver stability

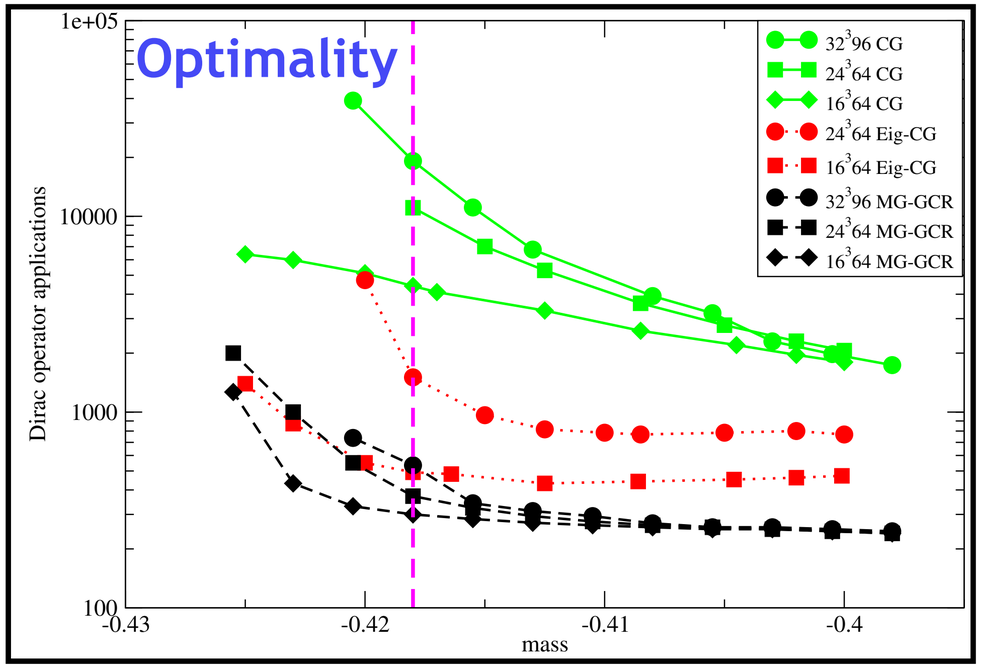
GCR solver deployed at every level

- N(N+1)/2 reductions required

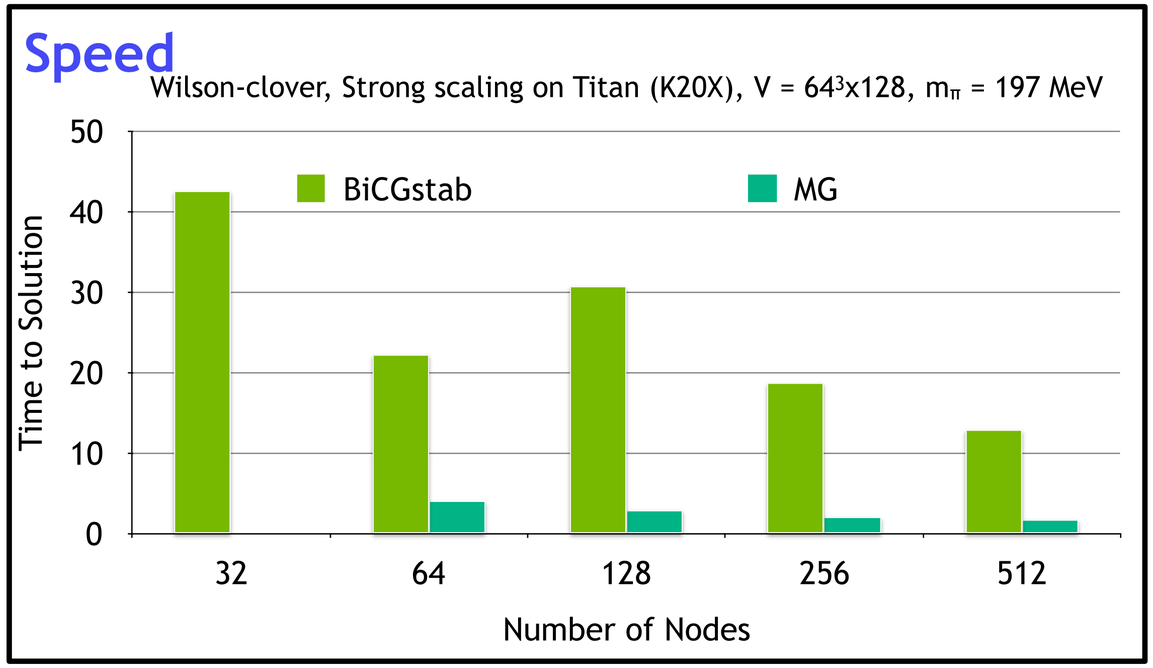
Use MR as a smoother

- N reductions required

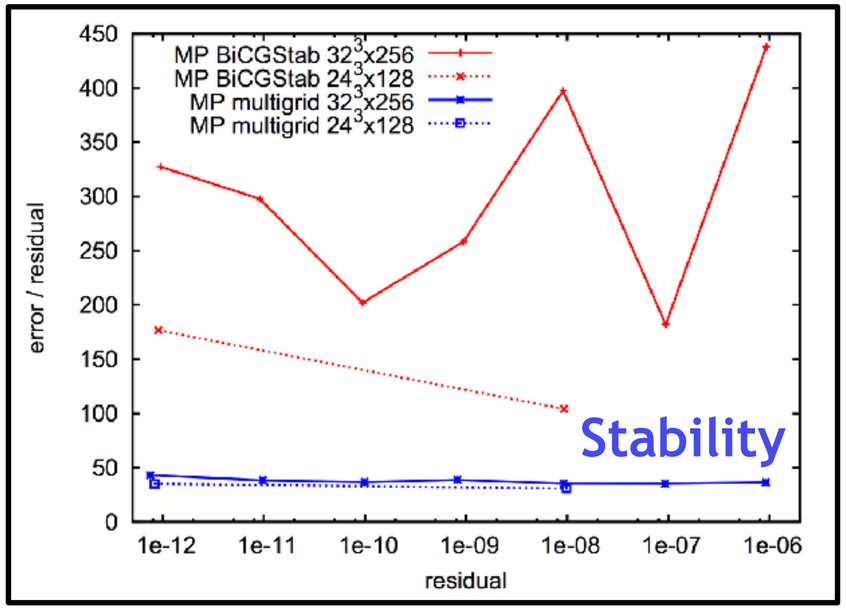
WHY MULTIGRID?



Babich *et al* 2010

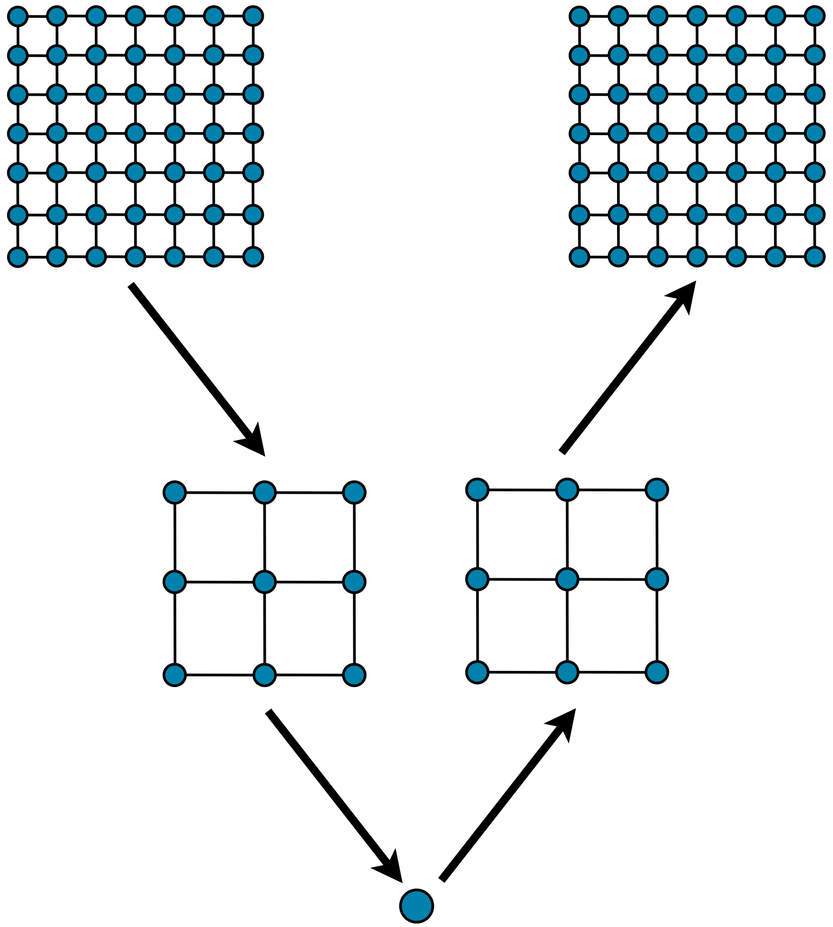


Clark *et al* (2016)



Osborn *et al* 2010

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

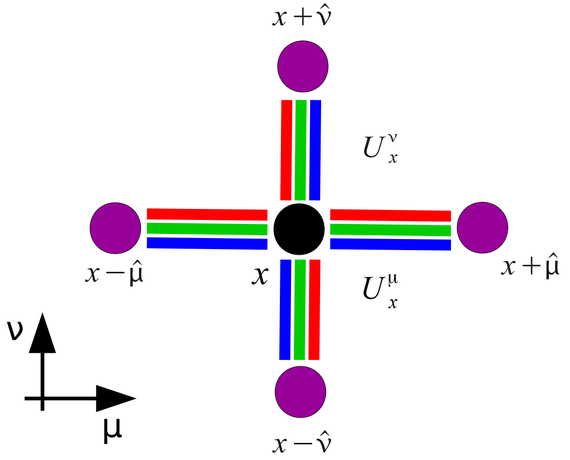
Coarse grids are worst possible scenario

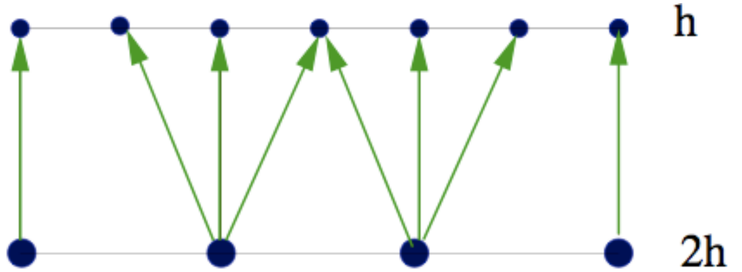
- More cores than degrees of freedom
- Increasingly serial and latency bound
- Little's law (bytes = bandwidth * latency)
- Amdahl's law limiter

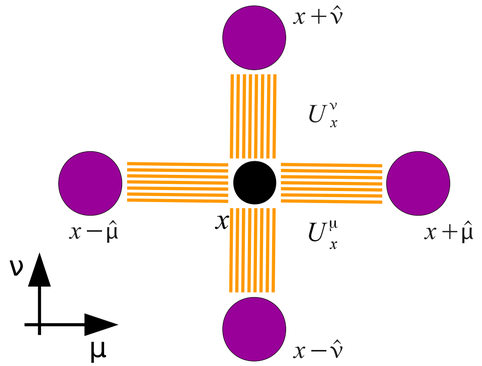
Multigrid exposes many of the problems expected at the Exascale

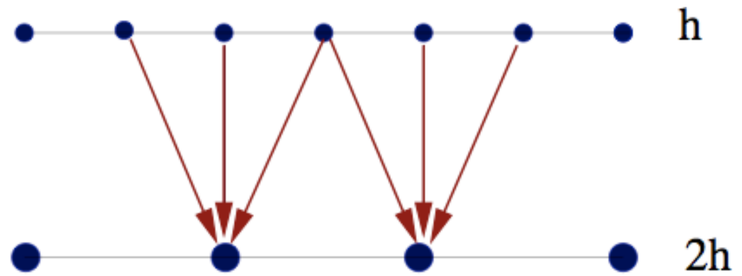
INGREDIENTS FOR PARALLEL ADAPTIVE MULTIGRID

- 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

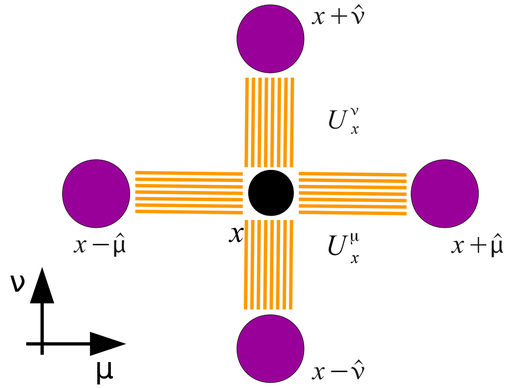








COARSE GRID OPERATOR

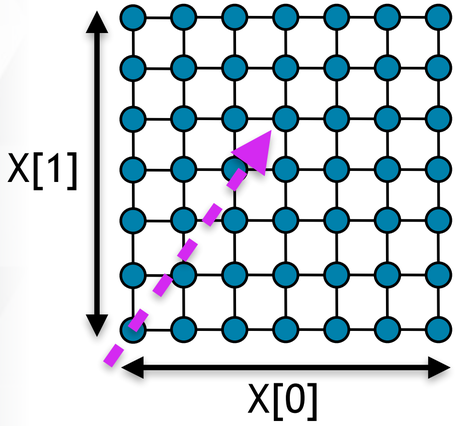


- Coarse operator looks like a Dirac operator (many more colors)
  - Link matrices have dimension $2N_v$ x $2N_v$ (e.g., 48 x 48)

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

- 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 cores
- Need to consider finer-grained parallelization
  - Increase parallelism to use all GPU resources
  - Load balancing

SOURCE OF PARALLELISM



1. Grid parallelism

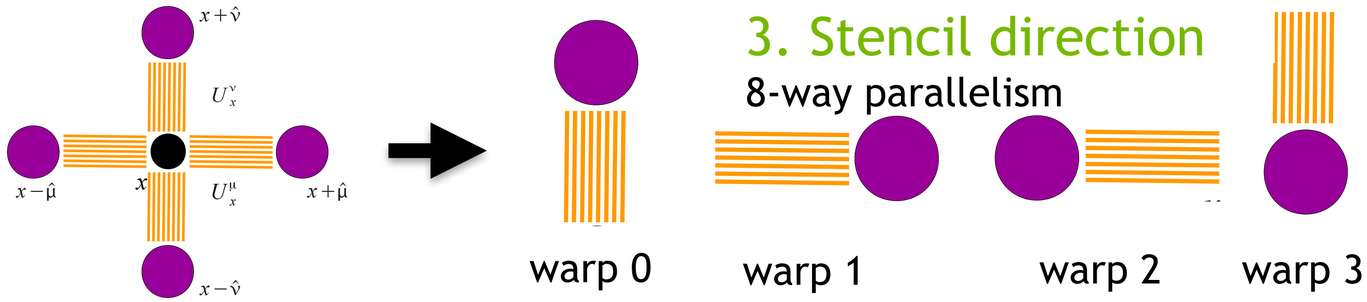
Volume of threads

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 parallelism (spin * color)



3. Stencil direction

8-way parallelism

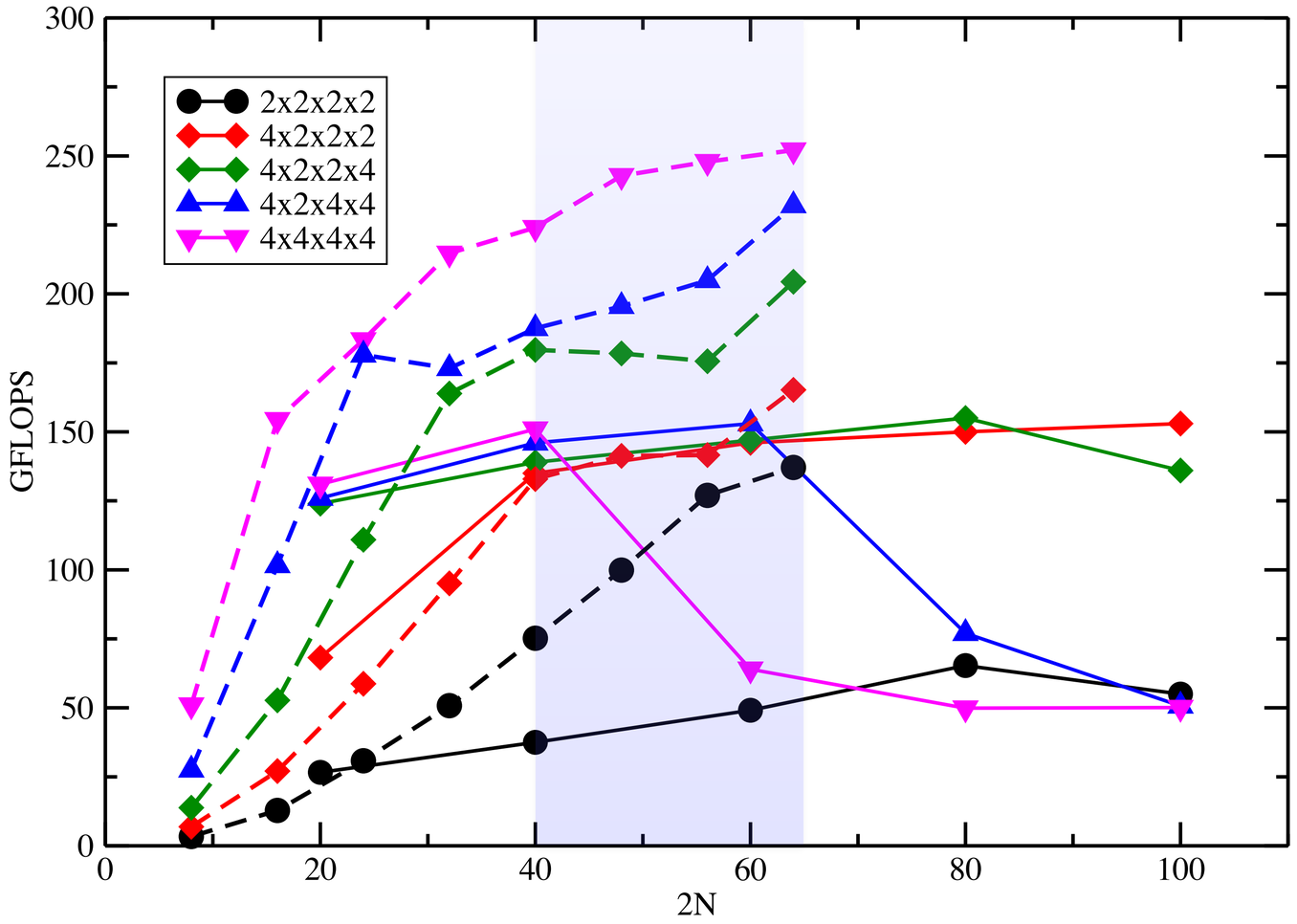
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \Rightarrow \begin{pmatrix} a_{00} & a_{01} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} + \begin{pmatrix} a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_2 \\ b_3 \end{pmatrix}$$

4. Dot-product partitioning

4-way parallelism

COARSE GRID OPERATOR PERFORMANCE

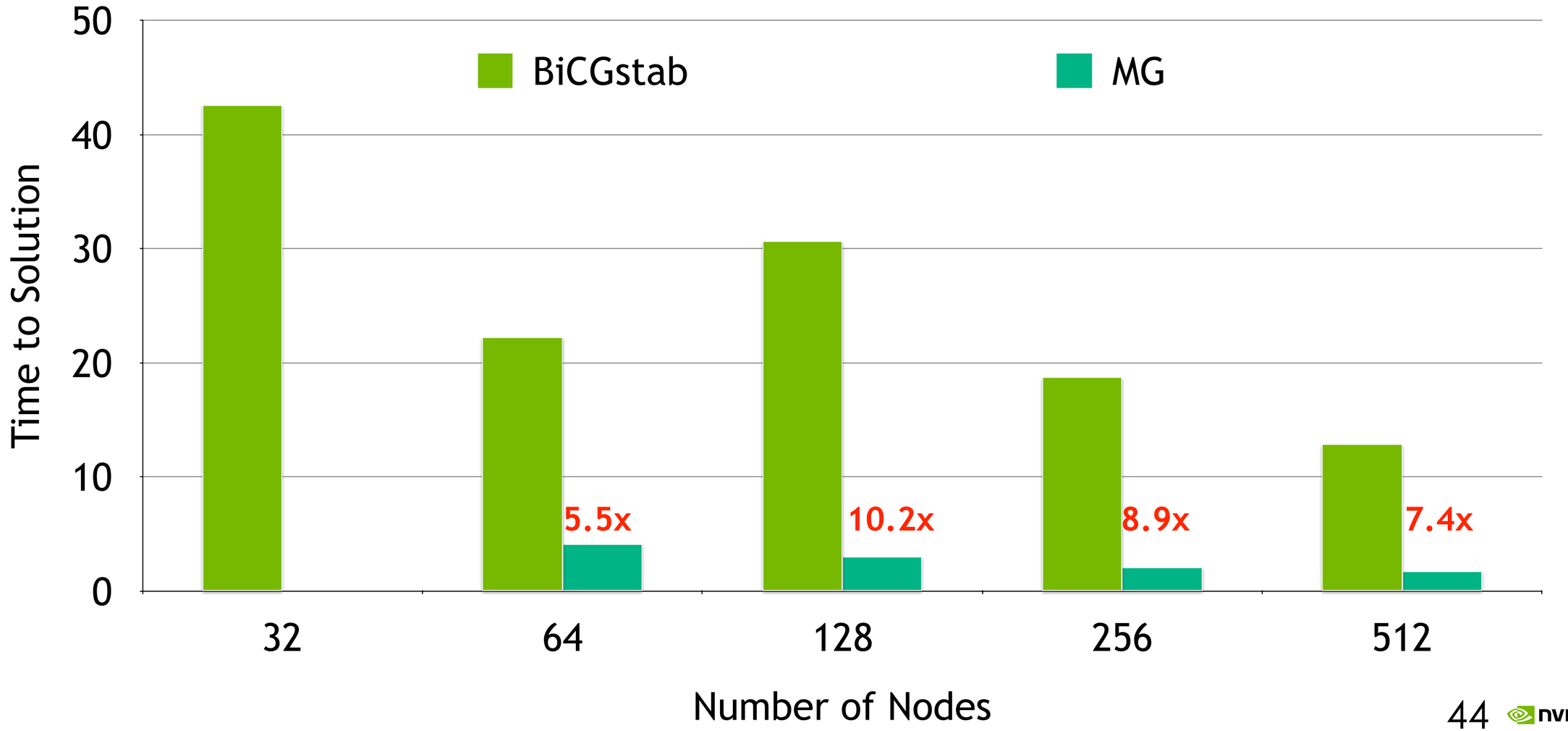
8-core Haswell 2.4 GHz (solid line) vs M6000 (dashed lined), FP32



- 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

MULTIGRID VERSUS BICGSTAB

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

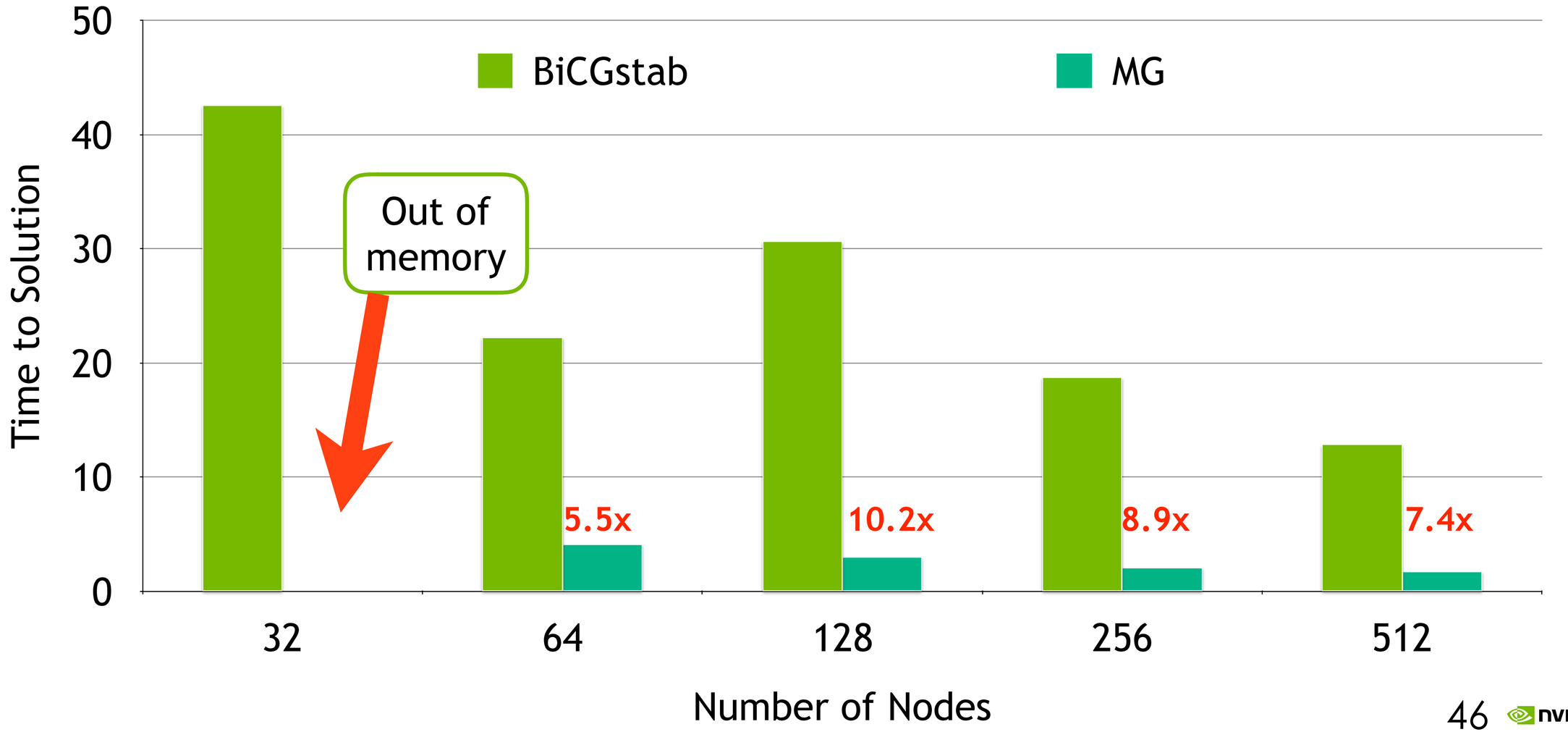


PRIOR OUTSTANDING ISSUES

- 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 = 32^3x 16 per P100 for clover solver
  - Forces us to strong scale more than we might like

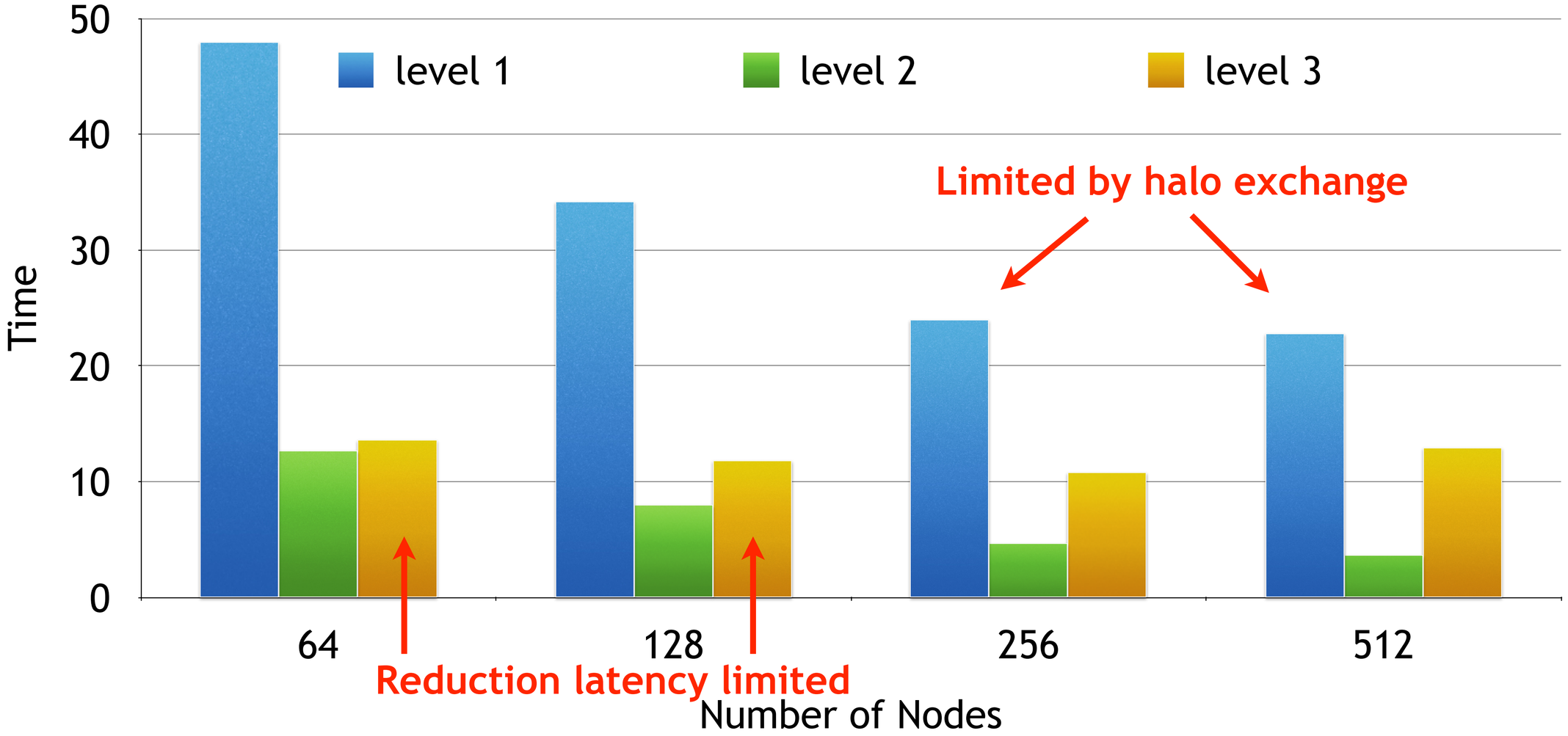
MULTIGRID VERSUS BICGSTAB

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



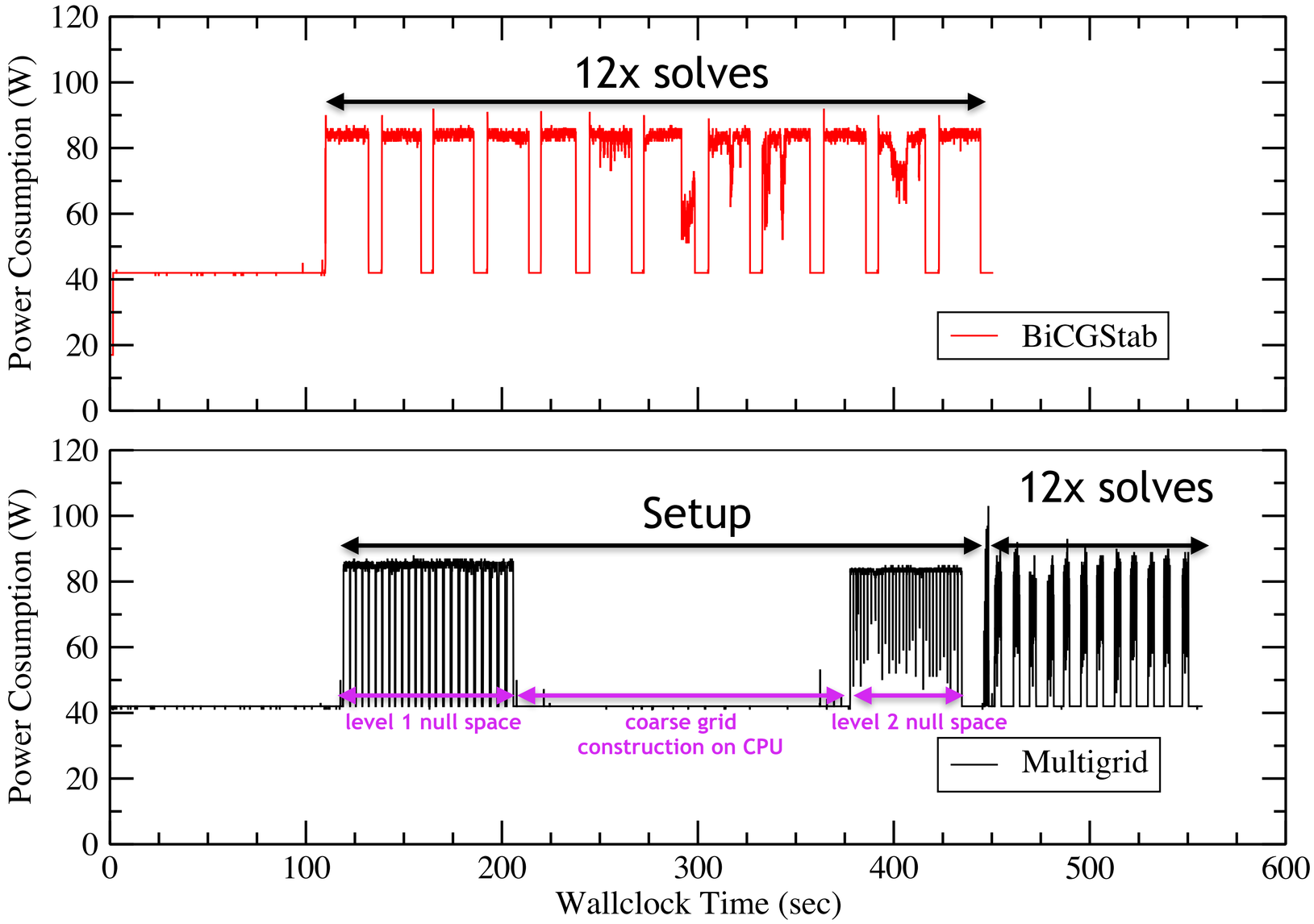
MULTIGRID TIMING BREAKDOWN

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

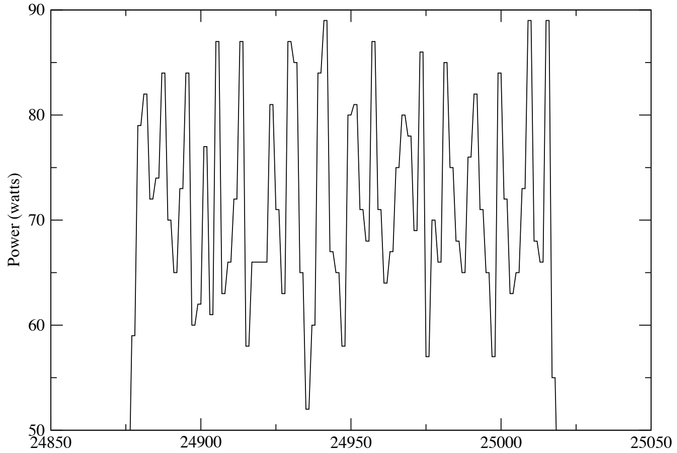
POWER EFFICIENCY



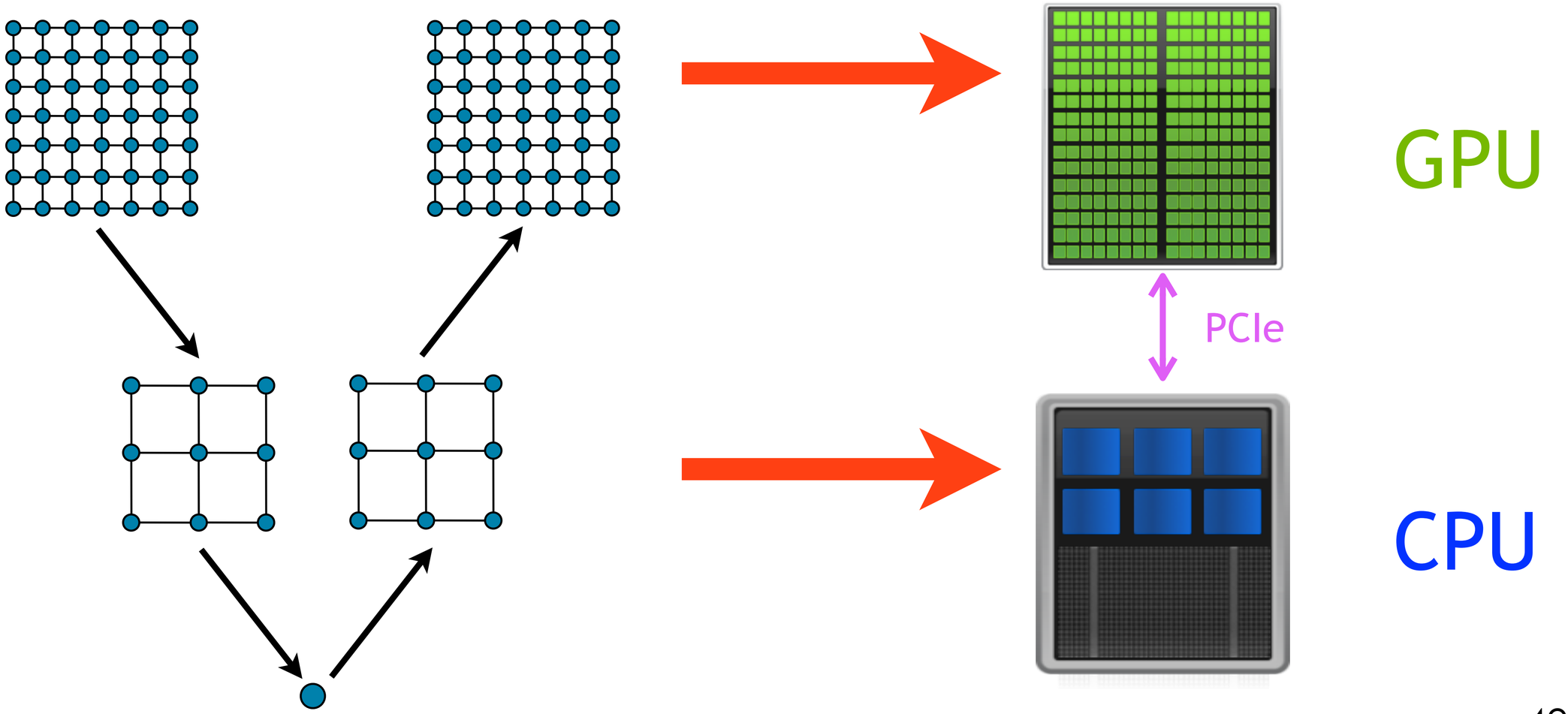
BiCGstab average power
~ 83 watts per GPU

MG average power
~ 72 watts per GPU

MG consumes less
power and 10x faster

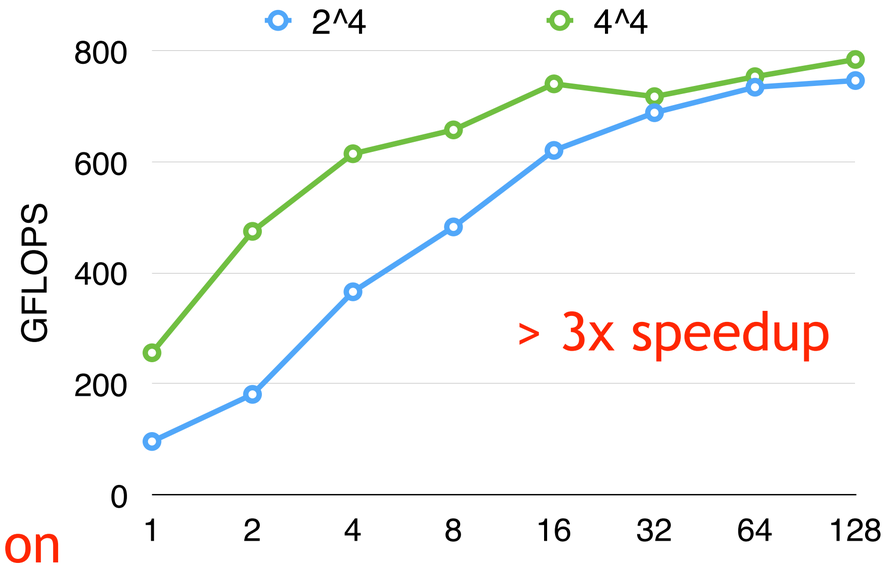


HIERARCHICAL ALGORITHMS ON HETEROGENEOUS ARCHITECTURES



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

NVIDIA

ADAPTIVE GEOMETRIC MULTIGRID

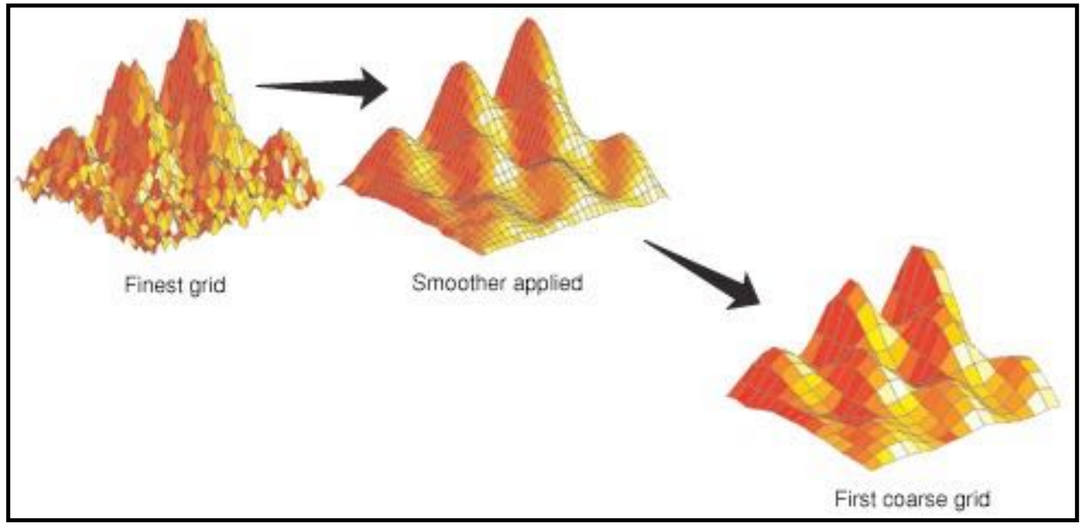
Adaptively find candidate null-space vectors

- 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

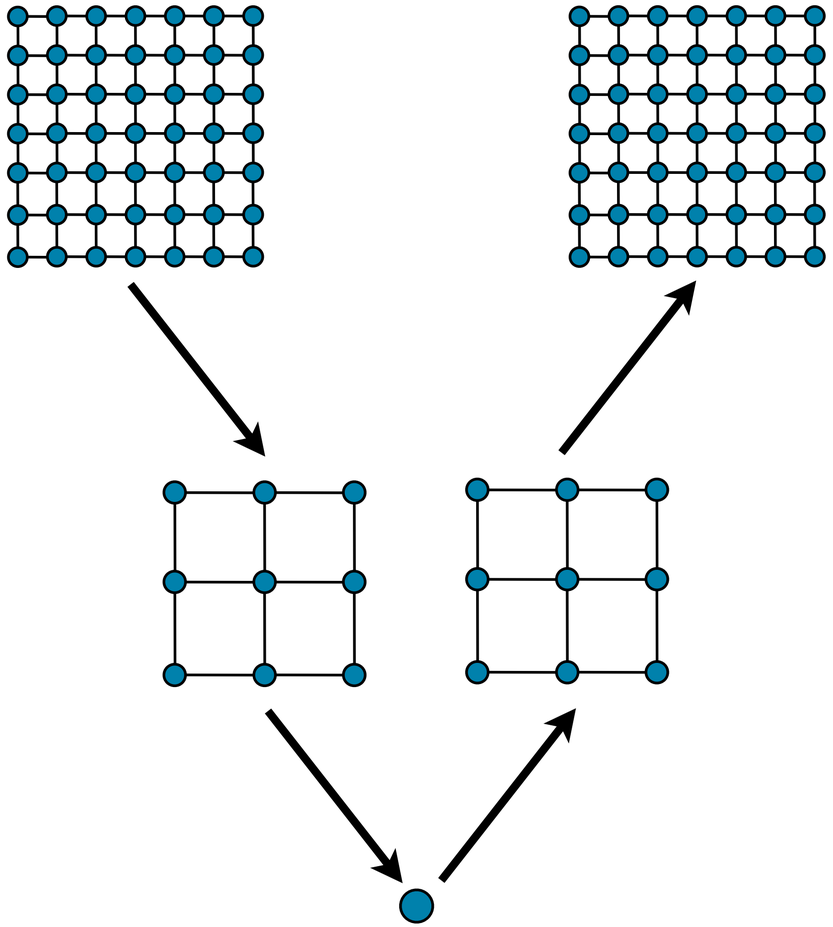
Babich *et al* 2010



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

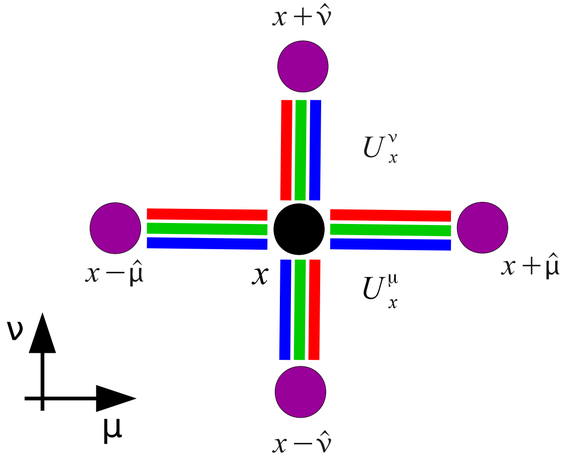
Coarse grids are worst possible scenario

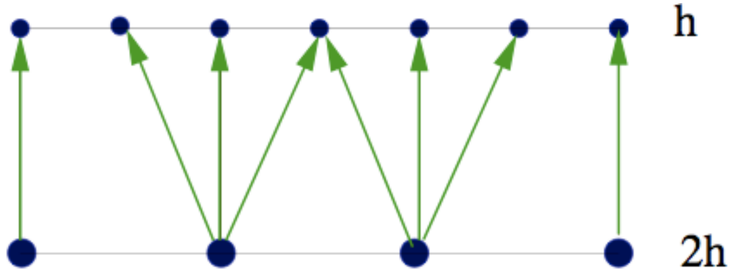
- More cores than degrees of freedom
- Increasingly serial and latency bound
- Little's law (bytes = bandwidth * latency)
- Amdahl's law limiter

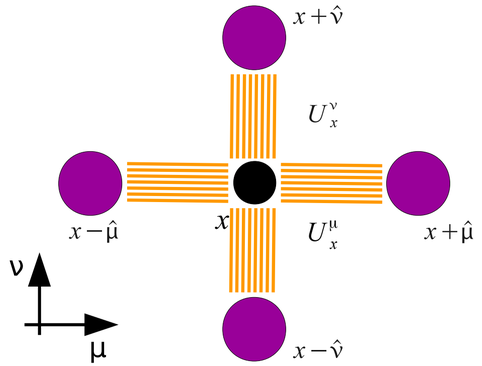
Multigrid exposes many of the problems expected at the Exascale

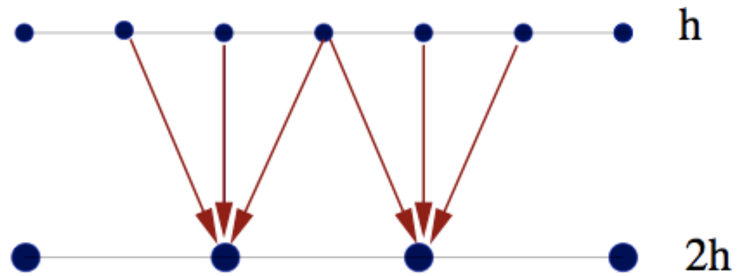
INGREDIENTS FOR PARALLEL ADAPTIVE MULTIGRID

- 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

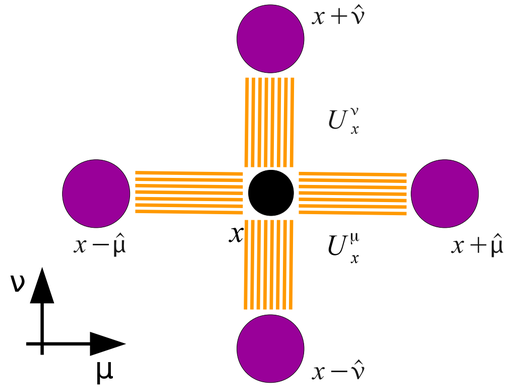








COARSE GRID OPERATOR

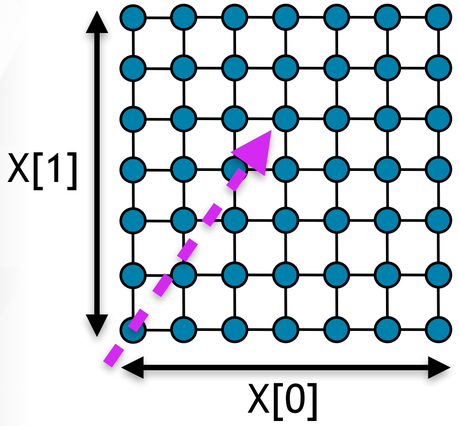


- Coarse operator looks like a Dirac operator (many more colors)
  - Link matrices have dimension $2N_v$ x $2N_v$ (e.g., 48 x 48)

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

- 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

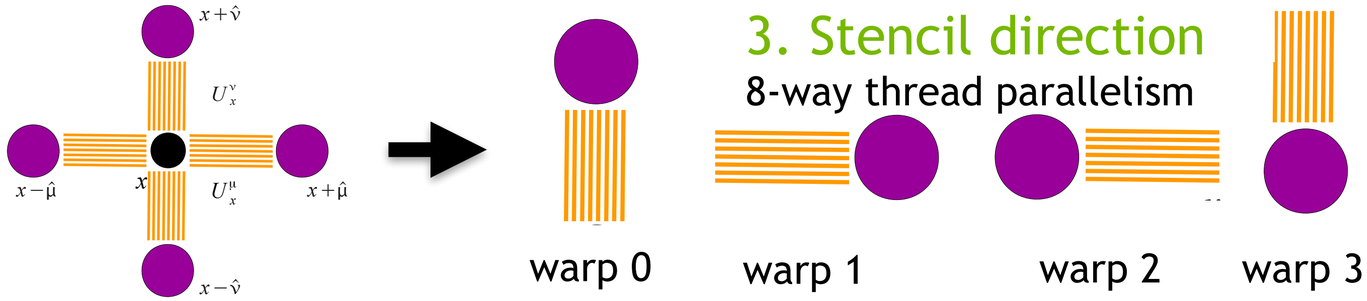
Volume of threads

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

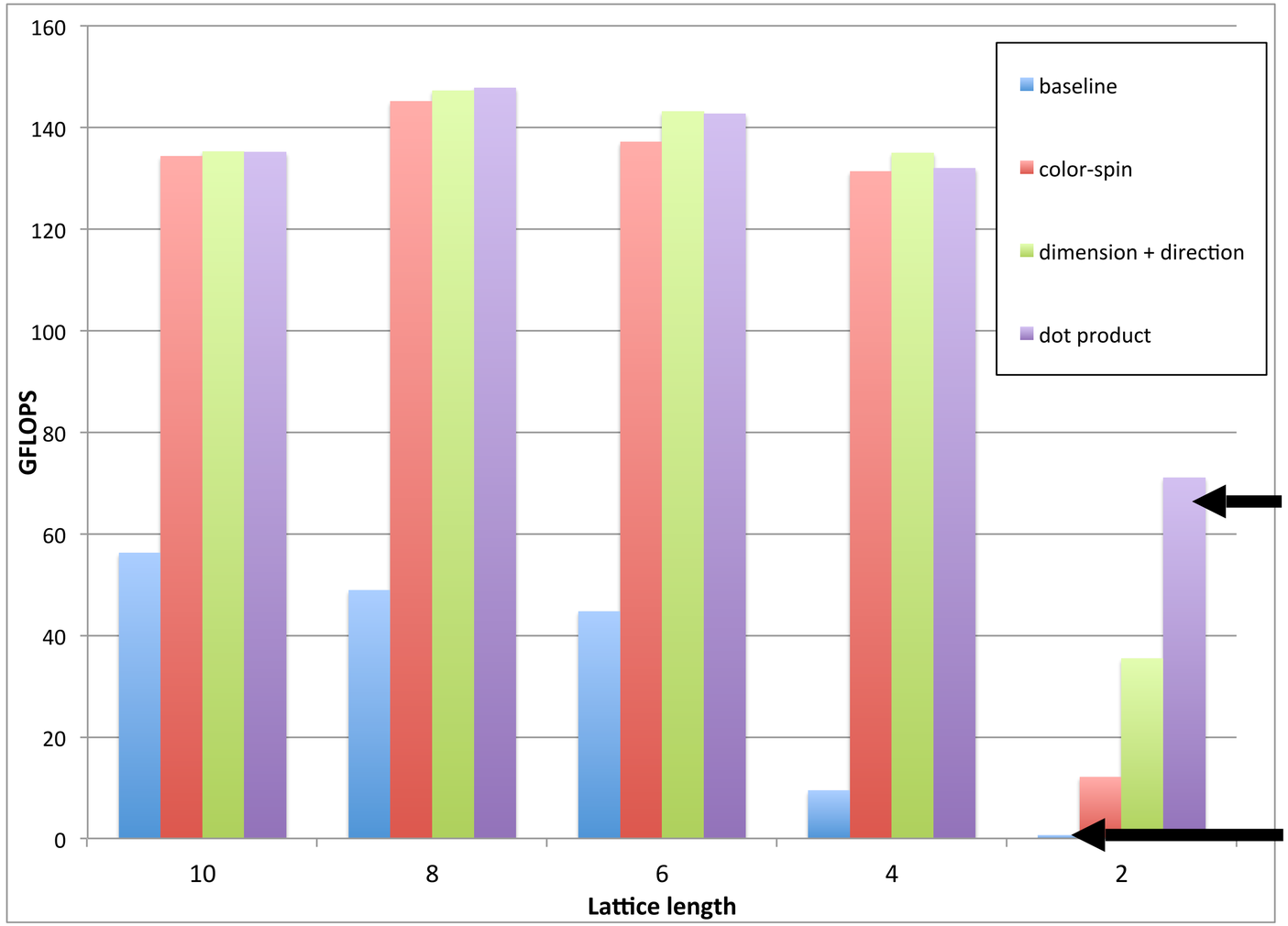
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \Rightarrow \begin{pmatrix} a_{00} & a_{01} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} + \begin{pmatrix} a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_2 \\ b_3 \end{pmatrix}$$

4. Dot-product partitioning

4-way thread parallelism + ILP

COARSE GRID OPERATOR PERFORMANCE

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



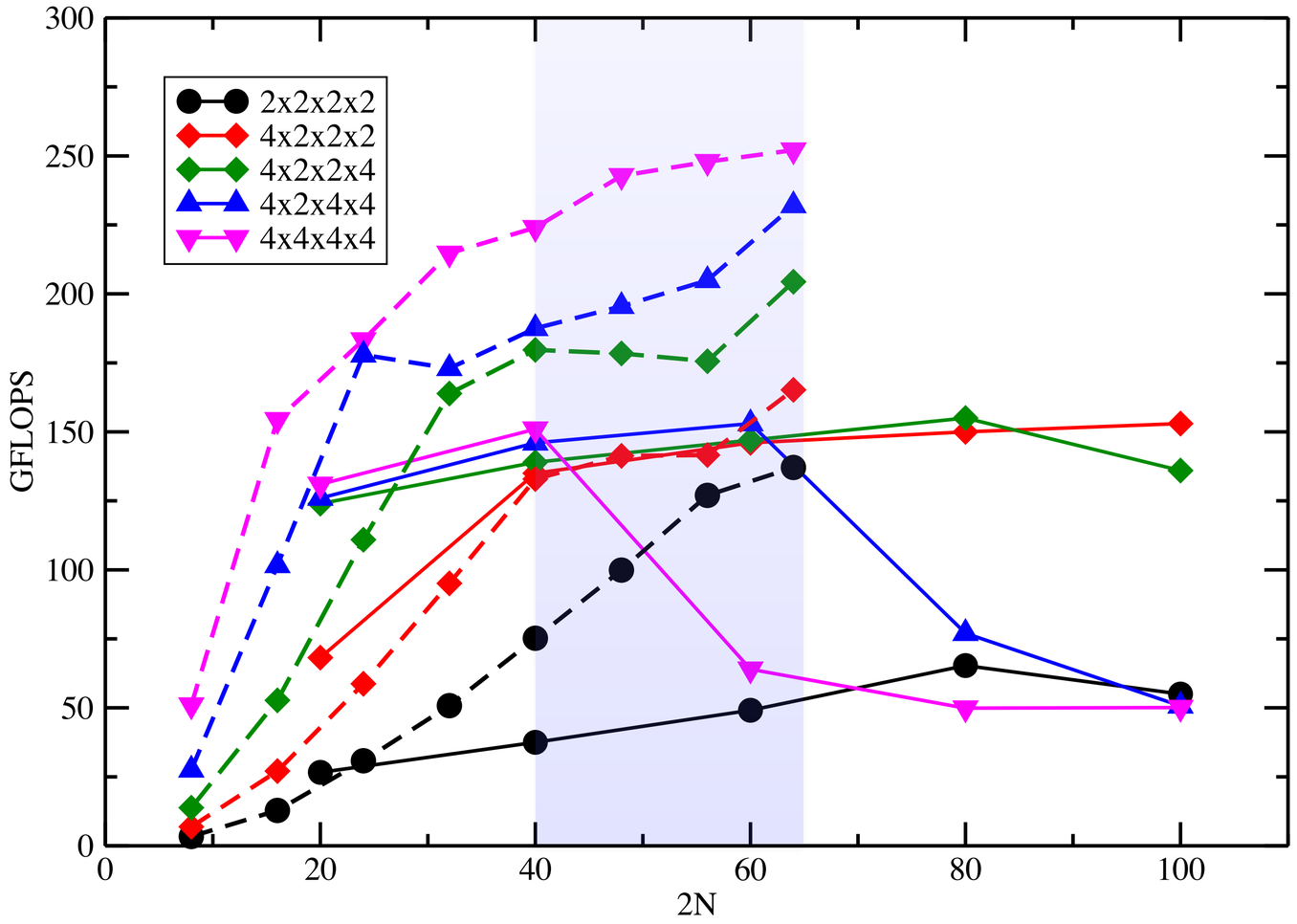
24,576-way parallel

16-way parallel

NVIDIA

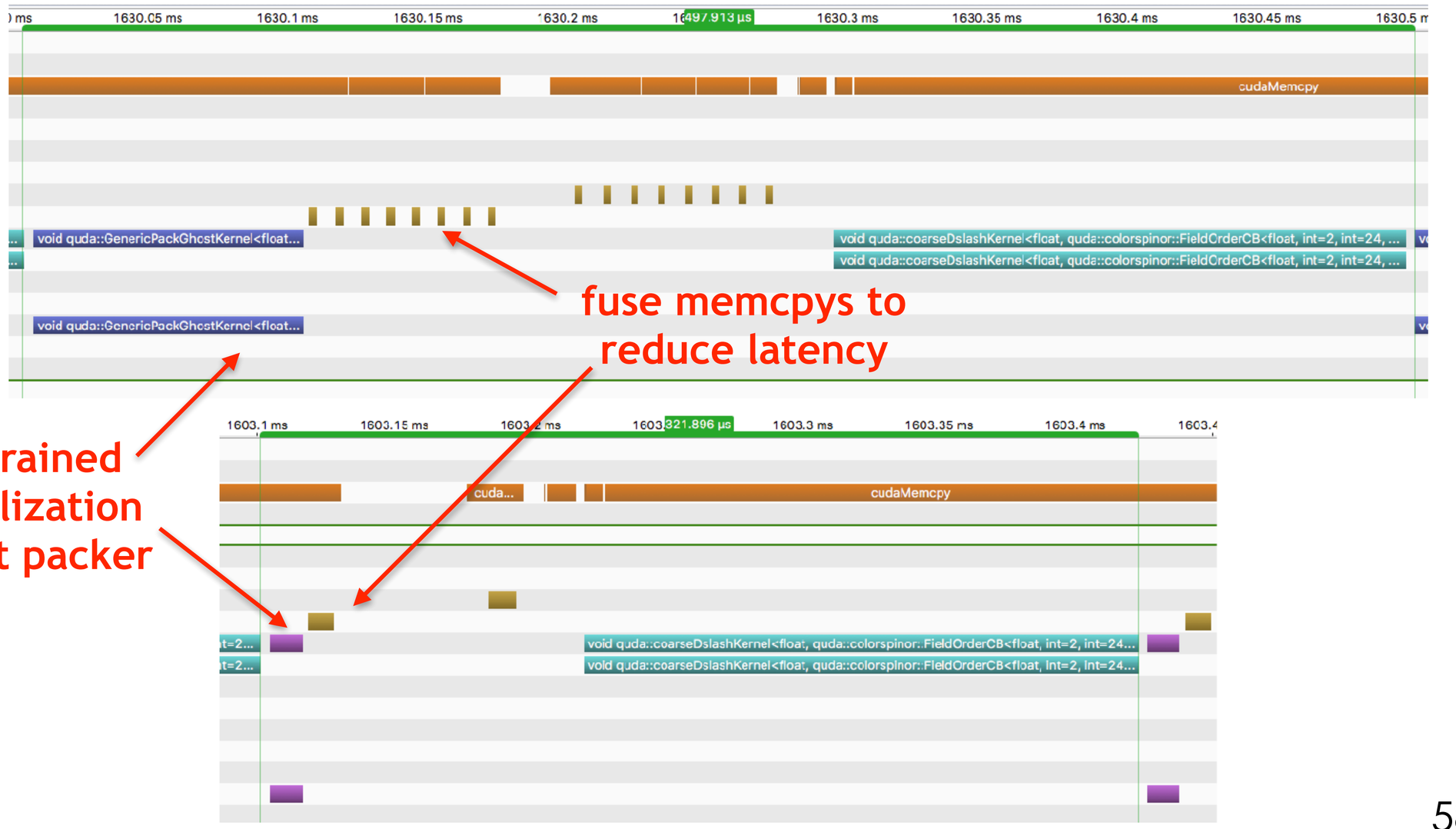
COARSE GRID OPERATOR PERFORMANCE

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



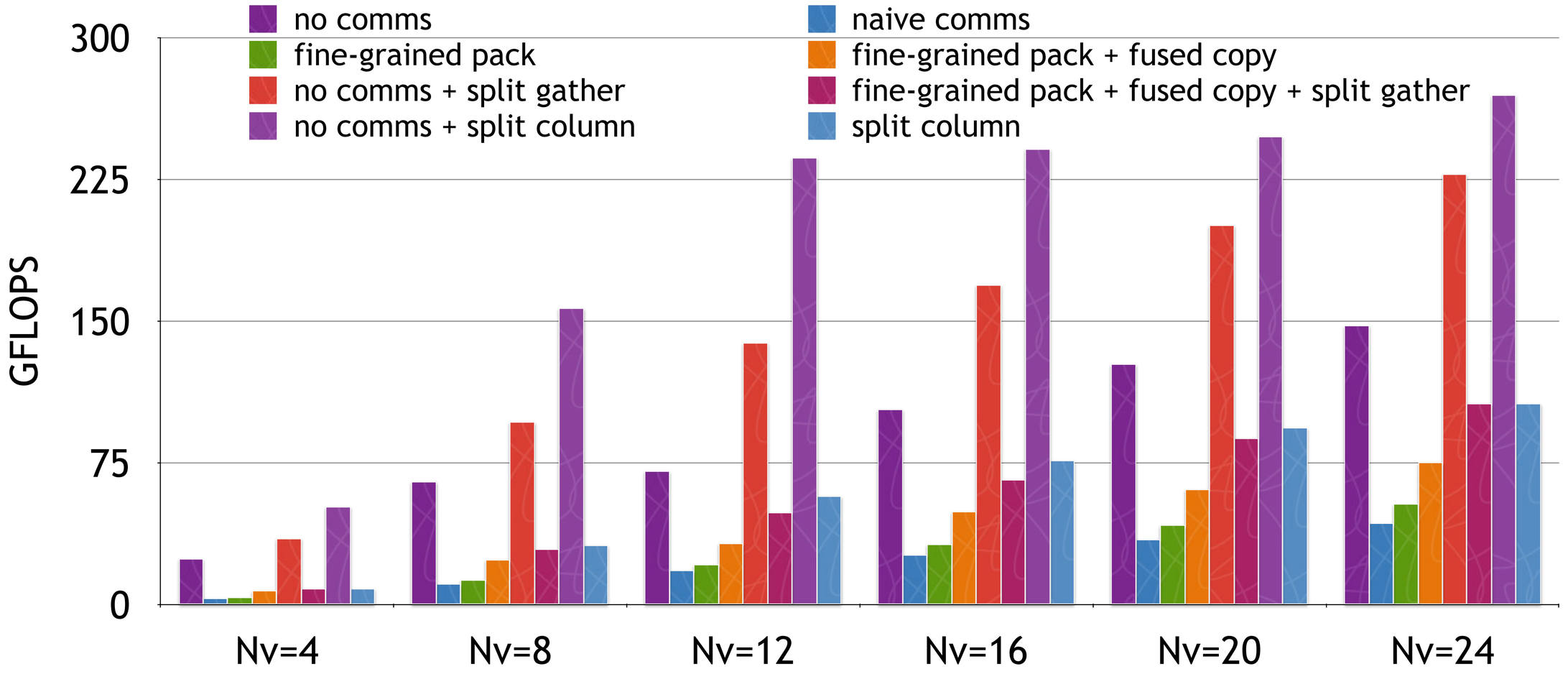
- Autotuner finds optimum degree of parallelization
  - Larger grids favor less fine grained
  - Coarse grids favor most fine grained
- GPU is nearly always faster than CPU
- Expect in future that coarse grids will favor CPUs
- For now, use GPU exclusively

IMPROVING STRONG SCALING



IMPROVING STRONG SCALING

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

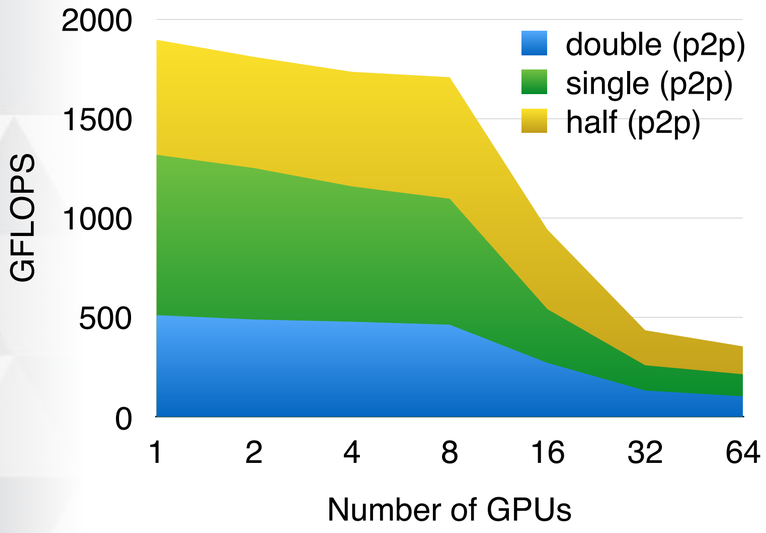


GPU DIRECT RDMA

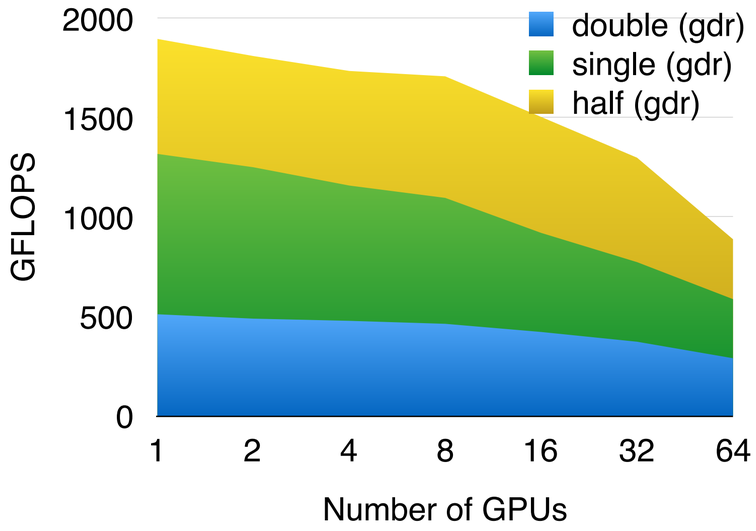
QUDA now has first-class support for GPU Direct RDMA

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

24^4 per GPU weak scaling on Saturn V

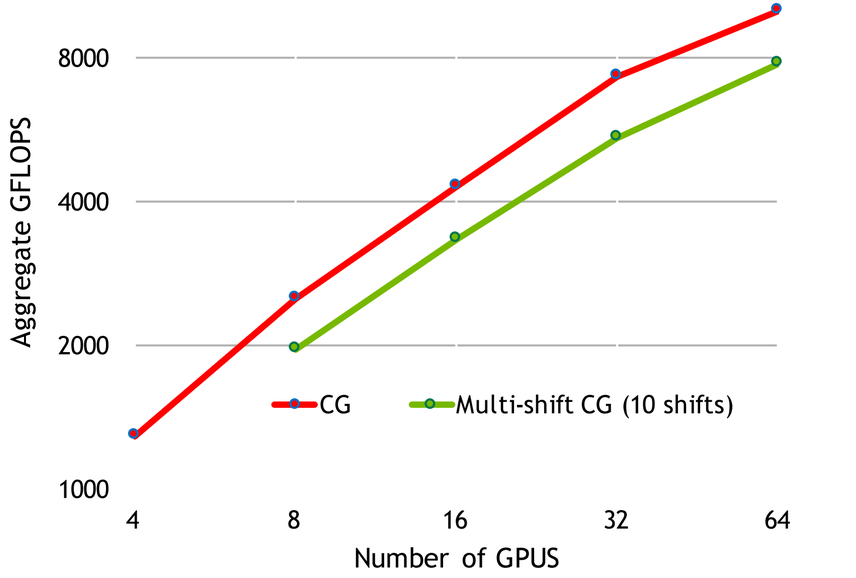


without GDR



with GDR

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



DOMAIN-DECOMPOSITION SMOOTHERS

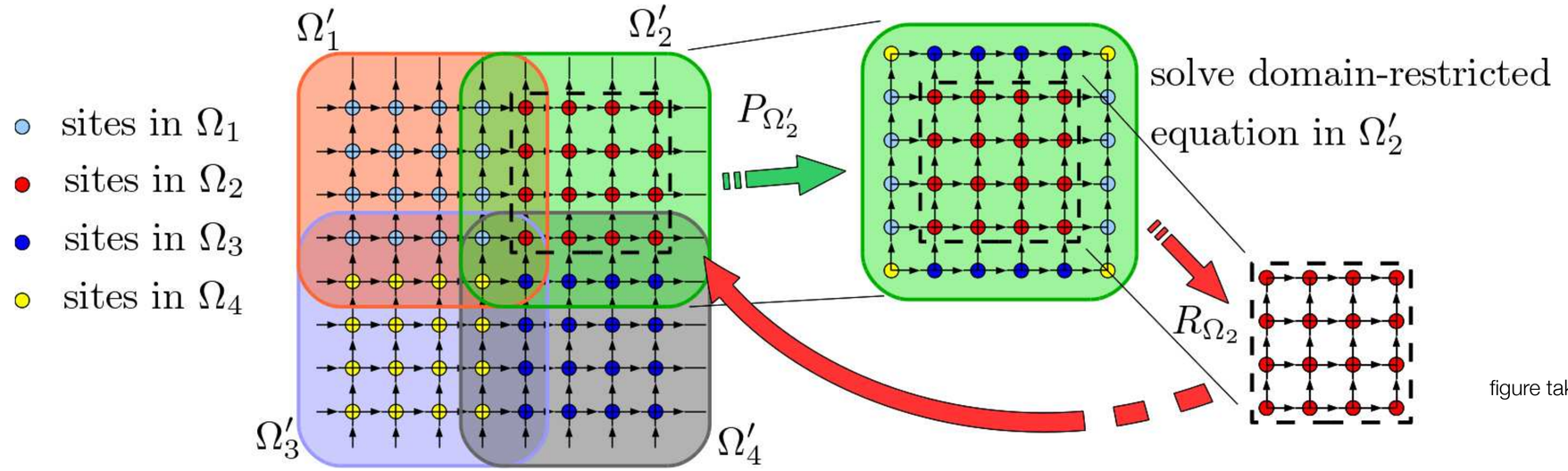


figure taken from Osaki and Ishikawa

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

QUDA now has support for both additive and multiplicative Schwarz smoothing

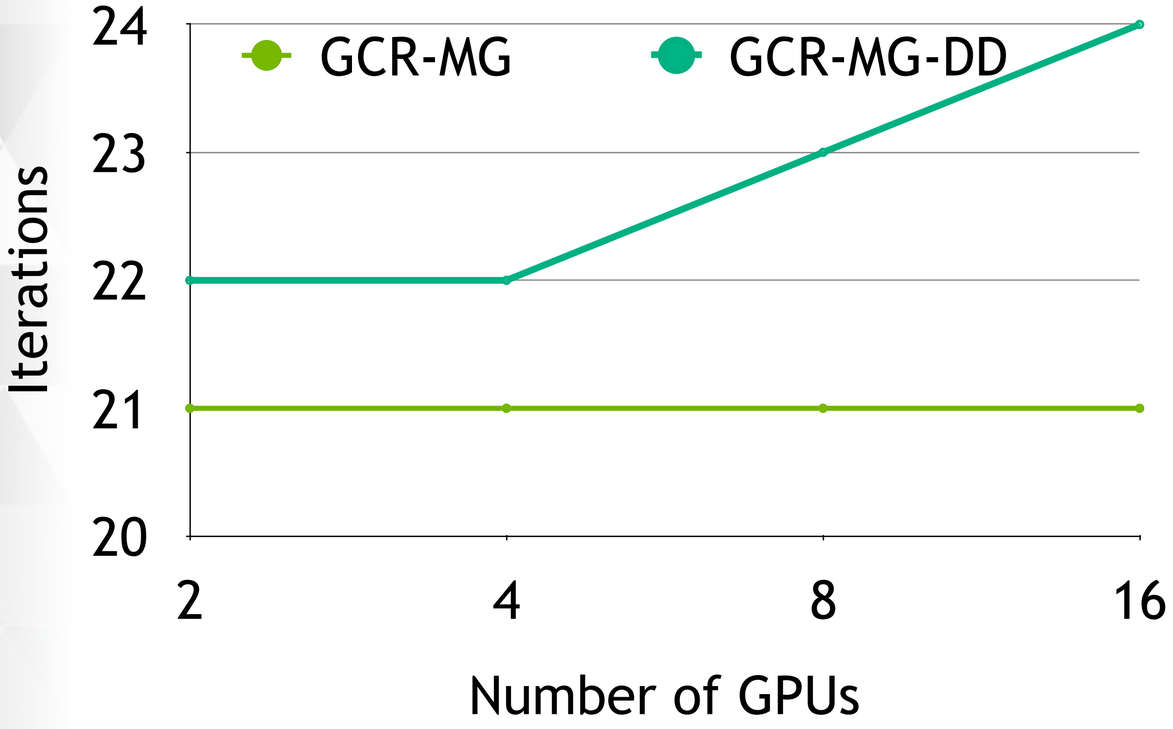
Enable at any level and / or combine with even/odd preconditioning at any level

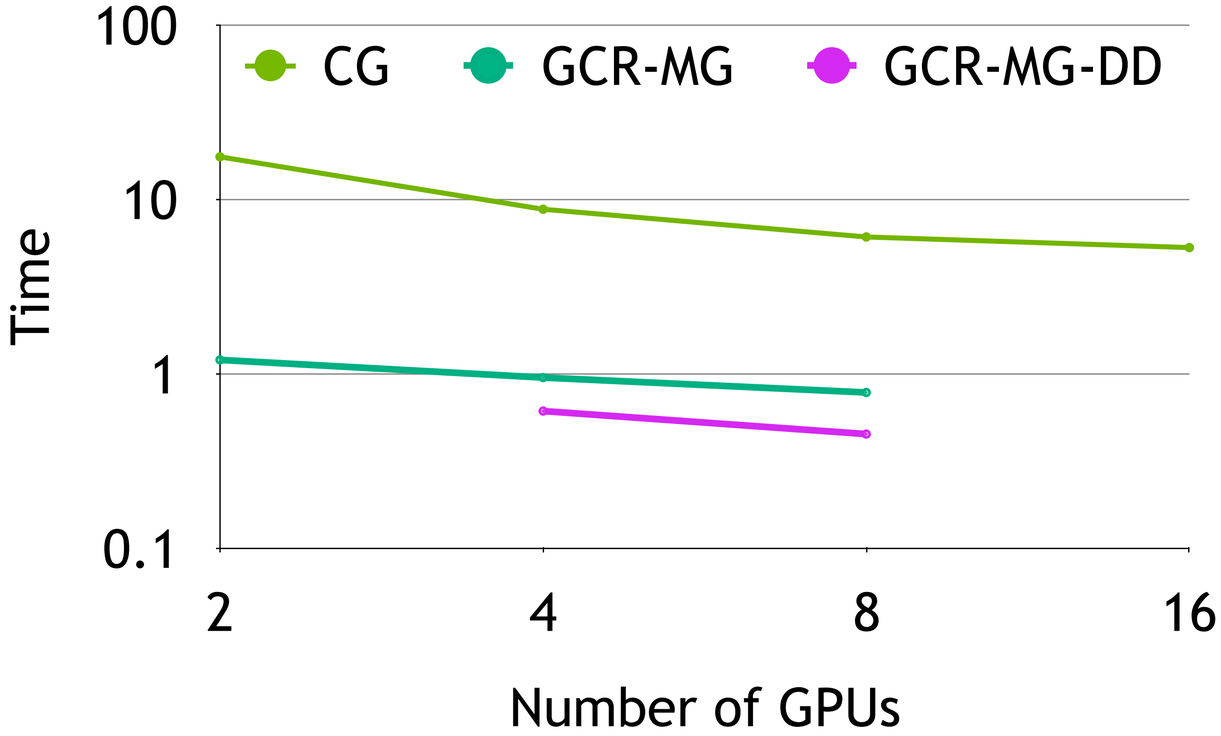
Dramatic reduction in communication important on systems with weak networks

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





MULTIGRID AT THE EXASCALE

Machine	*Titan*	*Summit*	*Summit++*
Volume	64^3x128	128^3x256	256^3x512
1st coarse grid	16^3x32	32^3x64	64^3x128
2nd coarse grid	8^3x16	16^3x32	32^3x64
3rd coarse grid		8^3x16	16^3x32
3rd coarse grid			8^3x16

Computers are getting wider not faster

- Increasing the problem size means running on more cores
- Coarse grids will be running on subset of the nodes at same speed

Multigrid reverts to N log N

MULTIGRID (HMC) AT THE EXASCALE

Communication reducing algorithms more critical than ever

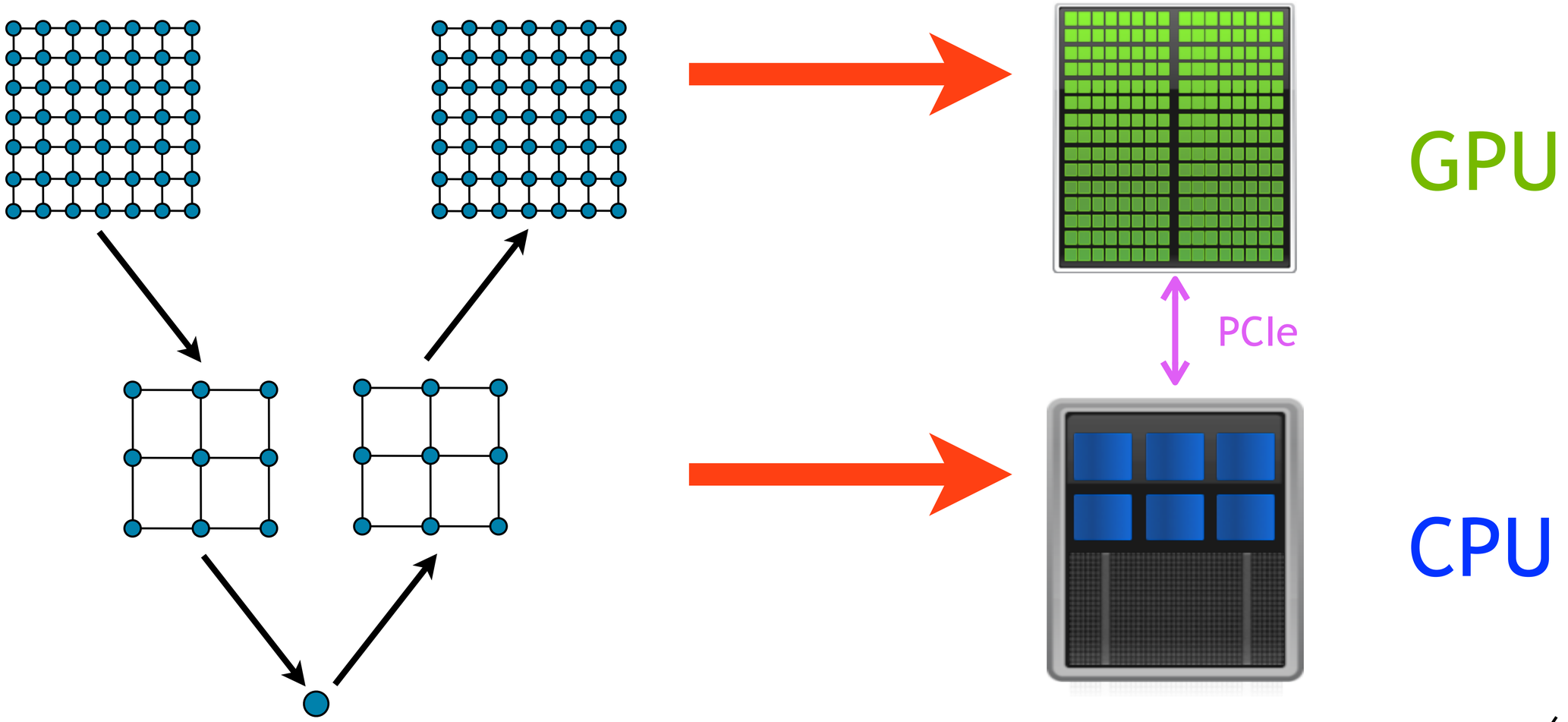
- Memory traffic
- Latency and synchronization

Acceleration of coarse grid solves ever more critical

Heterogeneous multigrid

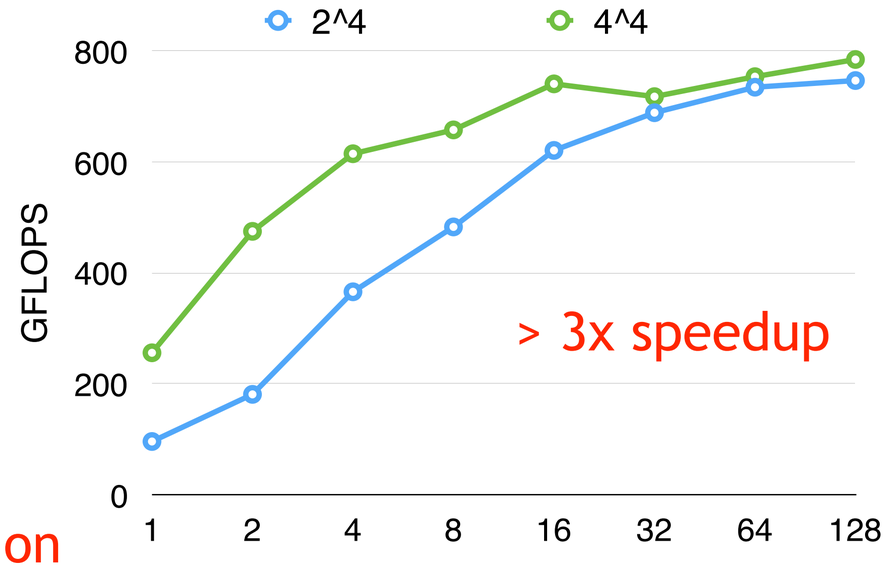
Task mixing?

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

16-BIT FIXED-POINT FOR COARSE GRIDS

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

Computation always done in FP32

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

WRITING ALGORITHMS IN FIXED POINT

```
/**
   Calculates the matrix UV^{s,c'}_mu(x) = \sum_c U^{c}_mu(x) * V^{s,c}_mu(x+mu)
   Where: mu = dir, s = fine spin, c' = coarse color, c = fine color
*/
template<bool from_coarse, typename Float, int dim, QudaDirection dir, int fineSpin, int fineColor,
         int coarseSpin, int coarseColor, typename Wtype, typename Arg>
__device__ __host__ inline void computeUV(Arg &arg, const Wtype &W, int parity, int x_cb, int ic_c) {

  int coord[5];
  coord[4] = 0;
  getCoords(coord, x_cb, arg.x_size, parity);

  complex<Float> UV[fineSpin][fineColor];

  for(int s = 0; s < fineSpin; s++) {
    for(int c = 0; c < fineColor; c++) {
      UV[s][c] = static_cast<Float>(0.0);
    }
  }

  if ( arg.comm_dim[dim] && (coord[dim] + 1 >= arg.x_size[dim]) ) {
    int nFace = 1;
    int ghost_idx = ghostFaceIndex<1>(coord, arg.x_size, dim, nFace);

    for(int s = 0; s < fineSpin; s++) {  //Fine Spin
      for(int ic = 0; ic < fineColor; ic++) { //Fine Color rows of gauge field
        for(int jc = 0; jc < fineColor; jc++) {  //Fine Color columns of gauge field
          UV[s][ic] += arg.U(dim, parity, x_cb, ic, jc) * W.Ghost(dim, 1, (parity+1)&1, ghost_idx, s, jc, ic_c);
        }
      }
    }

  } else {
    int y_cb = linkIndexP1(coord, arg.x_size, dim);

    for(int s = 0; s < fineSpin; s++) {  //Fine Spin
      for(int ic = 0; ic < fineColor; ic++) { //Fine Color rows of gauge field
        for(int jc = 0; jc < fineColor; jc++) {  //Fine Color columns of gauge field
          UV[s][ic] += arg.U(dim, parity, x_cb, ic, jc) * W((parity+1)&1, y_cb, s, jc, ic_c);
        }
      }
    }
  }


  for(int s = 0; s < fineSpin; s++) {
    for(int c = 0; c < fineColor; c++) {
      arg.UV(parity,x_cb,s,c,ic_c) = UV[s][c];
    }
  }


} // computeUV
```

Apply gauge field to set of null-space vectors

(Single precision variant)

Let’s see what changes to for 16-bit variant

WRITING ALGORITHMS IN FIXED POINT

```
/**
   Calculates the matrix UV^{s,c'}_mu(x) = \sum_c U^{c}_mu(x) * V^{s,c}_mu(x+mu)
   Where: mu = dir, s = fine spin, c' = coarse color, c = fine color
*/
template<bool from_coarse, typename Float, int dim, QudaDirection dir, int fineSpin, int fineColor,
         int coarseSpin, int coarseColor, typename Wtype, typename Arg>
__device__ __host__ inline void computeUV(Arg &arg, const Wtype &W, int parity, int x_cb, int ic_c) {

  int coord[5];
  coord[4] = 0;
  getCoords(coord, x_cb, arg.x_size, parity);

  complex<Float> UV[fineSpin][fineColor];

  for(int s = 0; s < fineSpin; s++) {
    for(int c = 0; c < fineColor; c++) {
      UV[s][c] = static_cast<Float>(0.0);
    }
  }

  if ( arg.comm_dim[dim] && (coord[dim] + 1 >= arg.x_size[dim]) ) {
    int nFace = 1;
    int ghost_idx = ghostFaceIndex<1>(coord, arg.x_size, dim, nFace);

    for(int s = 0; s < fineSpin; s++) {  //Fine Spin
      for(int ic = 0; ic < fineColor; ic++) { //Fine Color rows of gauge field
        for(int jc = 0; jc < fineColor; jc++) {  //Fine Color columns of gauge field
          UV[s][ic] += arg.U(dim, parity, x_cb, ic, jc) * W.Ghost(dim, 1, (parity+1)&1, ghost_idx, s, jc, ic_c);
        }
      }
    }

  } else {
    int y_cb = linkIndexP1(coord, arg.x_size, dim);

    for(int s = 0; s < fineSpin; s++) {  //Fine Spin
      for(int ic = 0; ic < fineColor; ic++) { //Fine Color rows of gauge field
        for(int jc = 0; jc < fineColor; jc++) {  //Fine Color columns of gauge field
          UV[s][ic] += arg.U(dim, parity, x_cb, ic, jc) * W((parity+1)&1, y_cb, s, jc, ic_c);
        }
      }
    }
  }


  for(int s = 0; s < fineSpin; s++) {
    for(int c = 0; c < fineColor; c++) {
      arg.UV(parity,x_cb,s,c,ic_c) = UV[s][c];
    }
  }


} // computeUV
```

Apply gauge field to set of null-space vectors

(Fixed-point variant)

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

WRITING ALGORITHMS IN FIXED POINT

```
/**
   Calculates the matrix UV^{s,c'}_mu(x) = \sum_c U^{c}_mu(x) * V^{s,c}_mu(x+mu)
   Where: mu = dir, s = fine spin, c' = coarse color, c = fine color
*/
template<bool from_coarse, typename Float, int dim, QudaDirection dir, int fineSpin, int fineColor,
         int coarseSpin, int coarseColor, typename Wtype, typename Arg>
__device__ __host__ inline void computeUV(Arg &arg, const Wtype &W, int parity, int x_cb, int ic_c) {

  int coord[5];
  coord[4] = 0;
  getCoords(coord, x_cb, arg.x_size, parity);

  complex<Float> UV[fineSpin][fineColor];

  for(int s = 0; s < fineSpin; s++) {
    for(int c = 0; c < fineColor; c++) {
      UV[s][c] = static_cast<Float>(0.0);
    }
  }

  if ( arg.comm_dim[dim] && (coord[dim] + 1 >= arg.x_size[dim]) ) {
    int nFace = 1;
    int ghost_idx = ghostFaceIndex<1>(coord, arg.x_size, dim, nFace);

    for(int s = 0; s < fineSpin; s++) {  //Fine Spin
      for(int ic = 0; ic < fineColor; ic++) { //Fine Color rows of gauge field
        for(int jc = 0; jc < fineColor; jc++) {  //Fine Color columns of gauge field
          UV[s][ic] += arg.U(dim, parity, x_cb, ic, jc) * W.Ghost(dim, 1, (parity+1)&1, ghost_idx, s, jc, ic_c);
        }
      }
    }

  } else {
    int y_cb = linkIndexP1(coord, arg.x_size, dim);

    for(int s = 0; s < fineSpin; s++) {  //Fine Spin
      for(int ic = 0; ic < fineColor; ic++) { //Fine Color rows of gauge field
        for(int jc = 0; jc < fineColor; jc++) {  //Fine Color columns of gauge field
          UV[s][ic] += arg.U(dim, parity, x_cb, ic, jc) * W((parity+1)&1, y_cb, s, jc, ic_c);
        }
      }
    }
  }


  for(int s = 0; s < fineSpin; s++) {
    for(int c = 0; c < fineColor; c++) {
      arg.UV(parity,x_cb,s,c,ic_c) = UV[s][c];
    }
  }


} // computeUV
```

```
/**
 * Read-only complex-member accessor function.  The last
 * parameter n is only used for indexed into the packed
 * null-space vectors.
 * @param x 1-d checkerboard site index
 * @param s spin index
 * @param c color index
 * @param v vector number
 */
__device__ __host__ inline const complex<Float> operator()
  (int parity, int x_cb, int s, int c, int n=0) const
{
  complex<short> tmp = v[accessor.index(parity,x_cb,s,c,n)];
  return scale_inv*complex<Float>(static_cast<Float>(tmp.x),
                                  static_cast<Float>(tmp.y));
}
```

```
/**
   @brief Assignment operator with complex number instance as input
   @param a Complex number we want to store in this accessor
*/
__device__ __host__ inline void operator=(const complex<Float> &a) {
  if ( !fixed ) { // not fixed point
    v[idx] = complex<storeFloat>(a.x, a.y);
  } else { // we need to scale and then round
    v[idx] = complex<storeFloat>(round(scale * a.x), round(scale * a.y));
  }
}
```

16-BIT FIXED-POINT FOR COARSE GRIDS

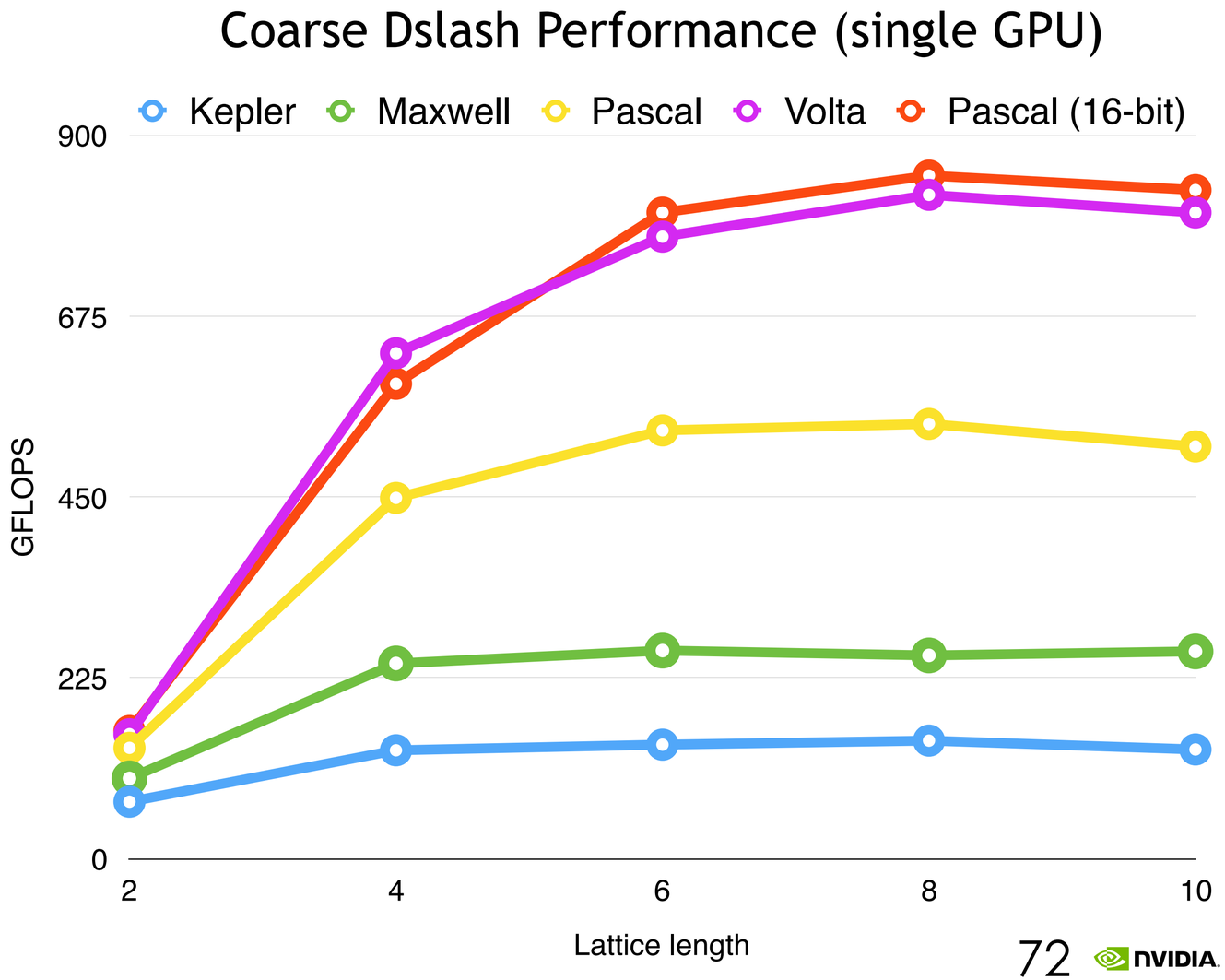
16-bit is like running with a new GPU!

Coarse-link setup kernels 1.8x faster

Restriction and Prolongation 1.8x faster

33% reduction in peak memory

Absolutely zero effect on multigrid convergence



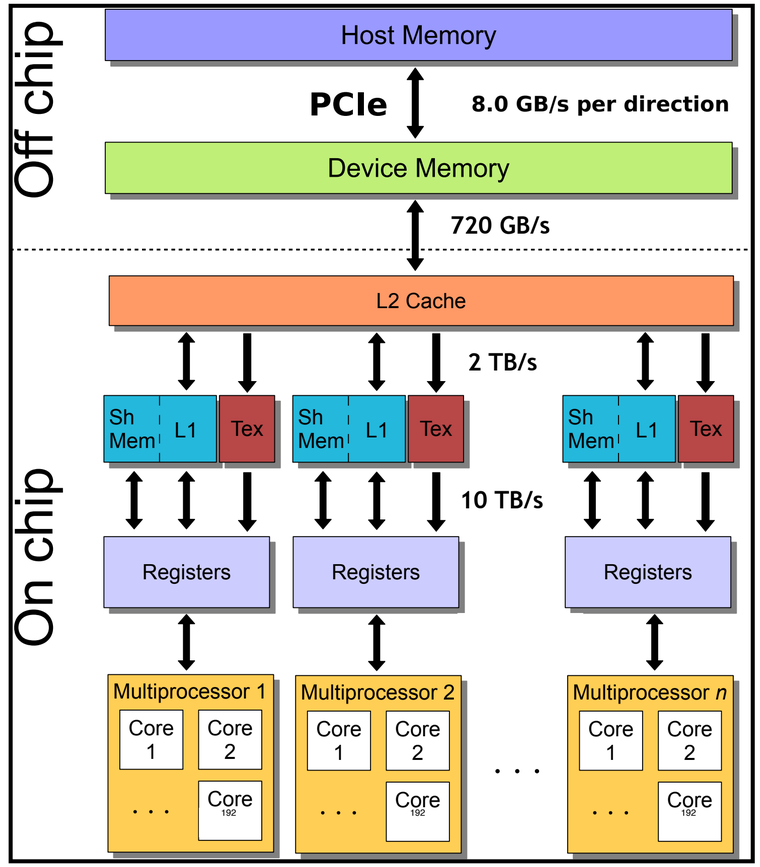
BLOCK ORTHOGONALIZATION

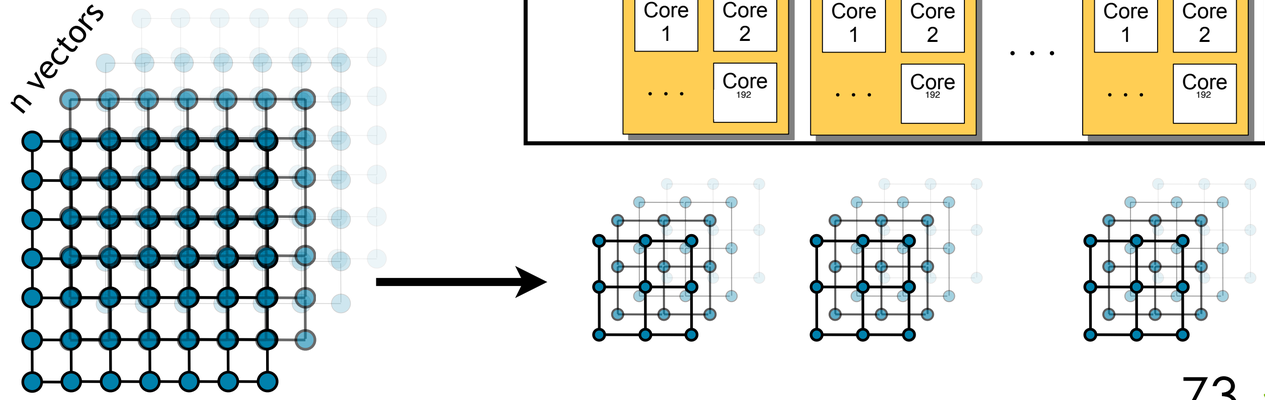
Forms the block orthonormal basis upon which we construct the coarse grid

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

Assign each multigrid aggregate to a CUDA thread block

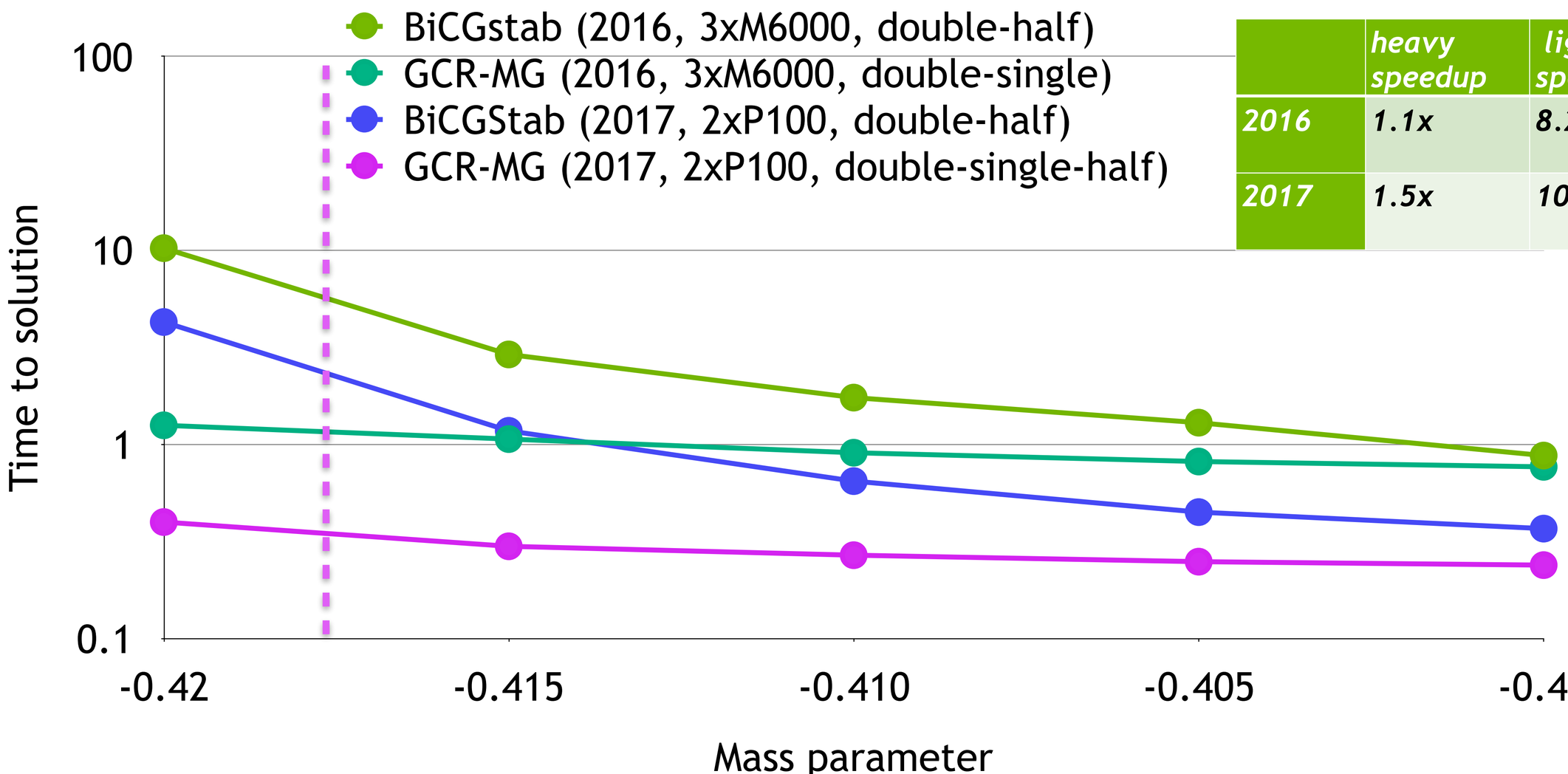
- All reductions are therefore local to a CUDA thread block
- Do the full block orthonormalization in a single kernel
- Minimizes total memory traffic





MULTIGRID VERSUS BICGSTAB

Wilson, V = 24^3x64, single workstation



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

COARSE-LINK CONSTRUCTION

Recipe

1. Compute required intermediate $T = U A^{-1} V$
   - 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

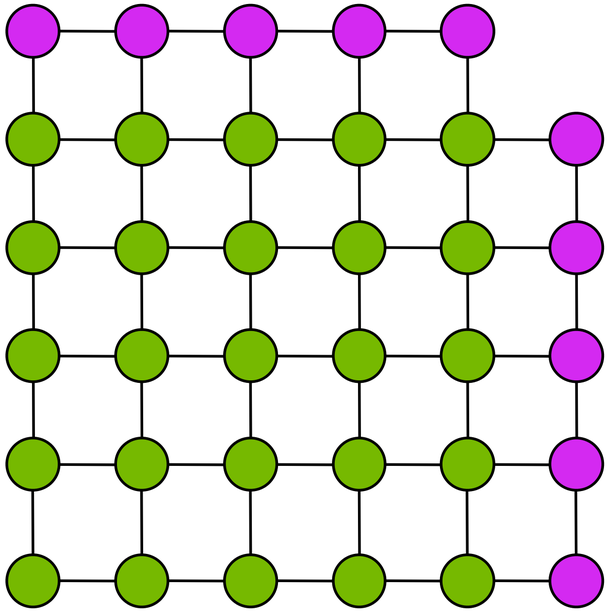
1. Compute required intermediate $T = UA^{-1}V$
   - Multi-RHS matrix-vector => matrix-matrix operation
   - High efficiency on parallel architectures

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



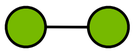
Employ fine-grained parallelization

- fine-grid geometry
- coarse-grid color

Each thread computes its assigned matrix elements

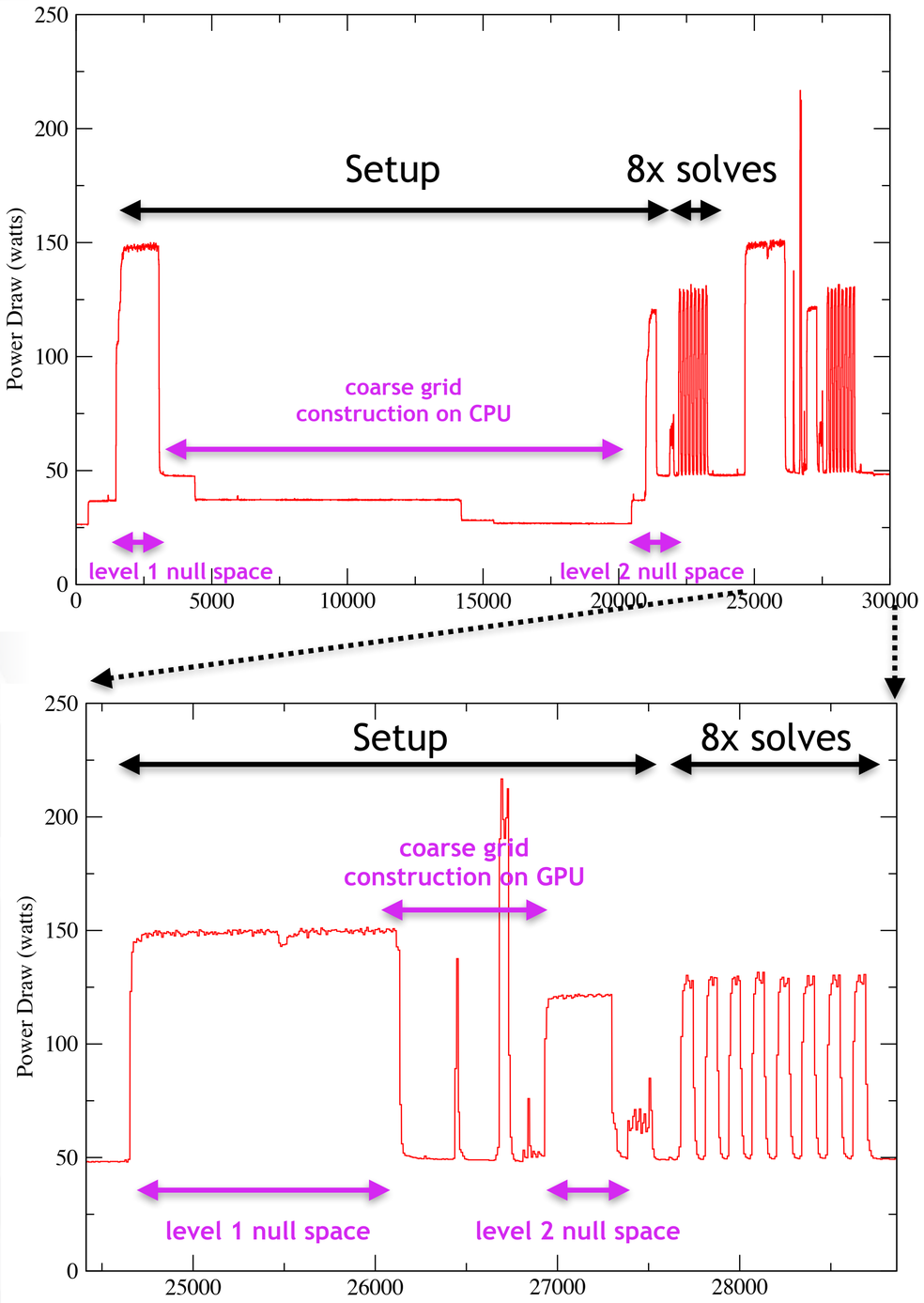
Atomically update the relevant coarse link field depending on thread location

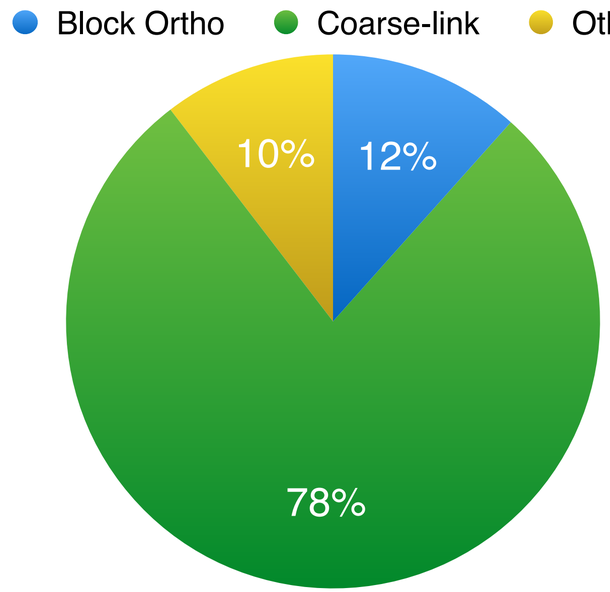
$$Y = \sum$$ 

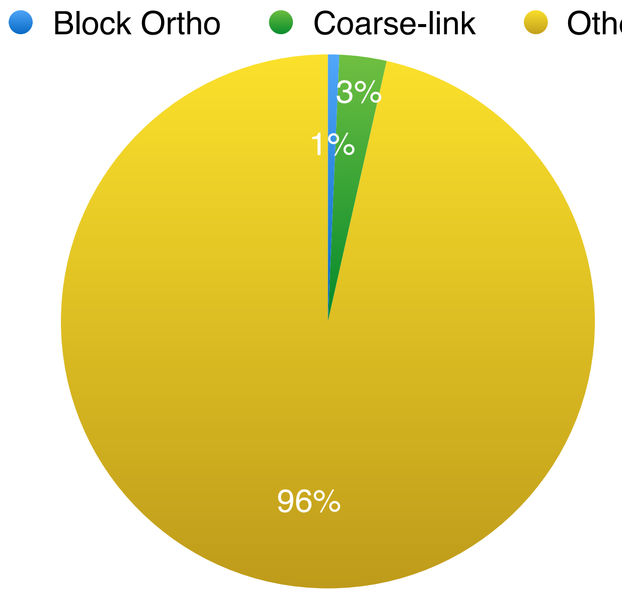
$$X = \sum$$ 

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