GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

REVOLUTIONIZING LATTICE QCD PHYSICS WITH HETEROGENEOUS MULTIGRID

Kate Clark, April 6th 2016

PRESENTED BY
NVIDIA

CONTENTS

Introduction to LQCD

QUDA Library

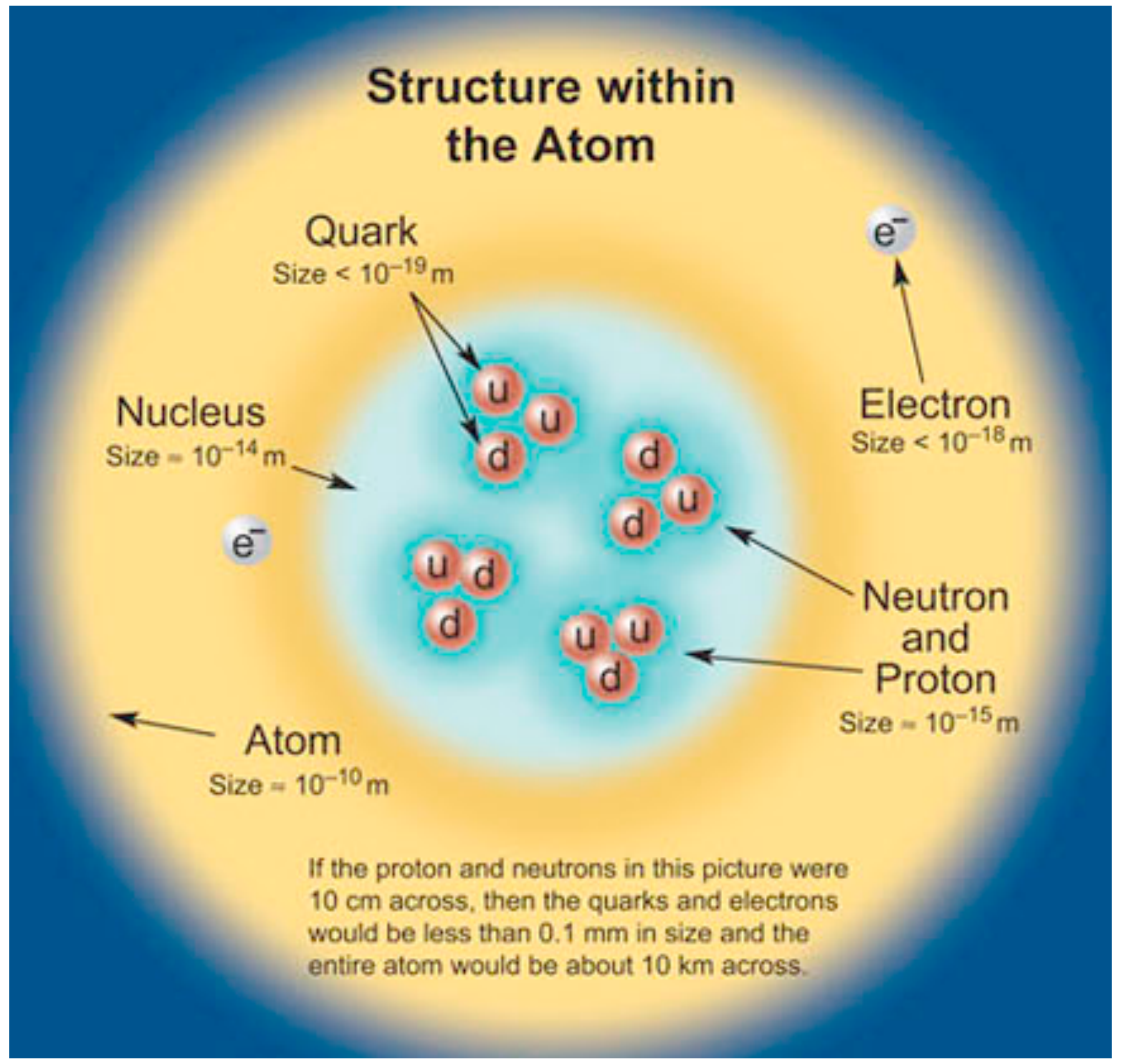
Adaptive Multigrid

QUDA Multigrid

Results

Conclusion

QUANTUM CHROMODYNAMICS


Structure within the Atom
Quark
Size < 10^{-19} m
Nucleus
Size ≈ 10^{-14} m
Electron
Size < 10^{-18} m
e⁻
u
d
Neutron and Proton
Size ≈ 10^{-15} m
Atom
Size ≈ 10^{-10} m
If the proton and neutrons in this picture were 10 cm across, then the quarks and electrons would be less than 0.1 mm in size and the entire atom would be about 10 km across.

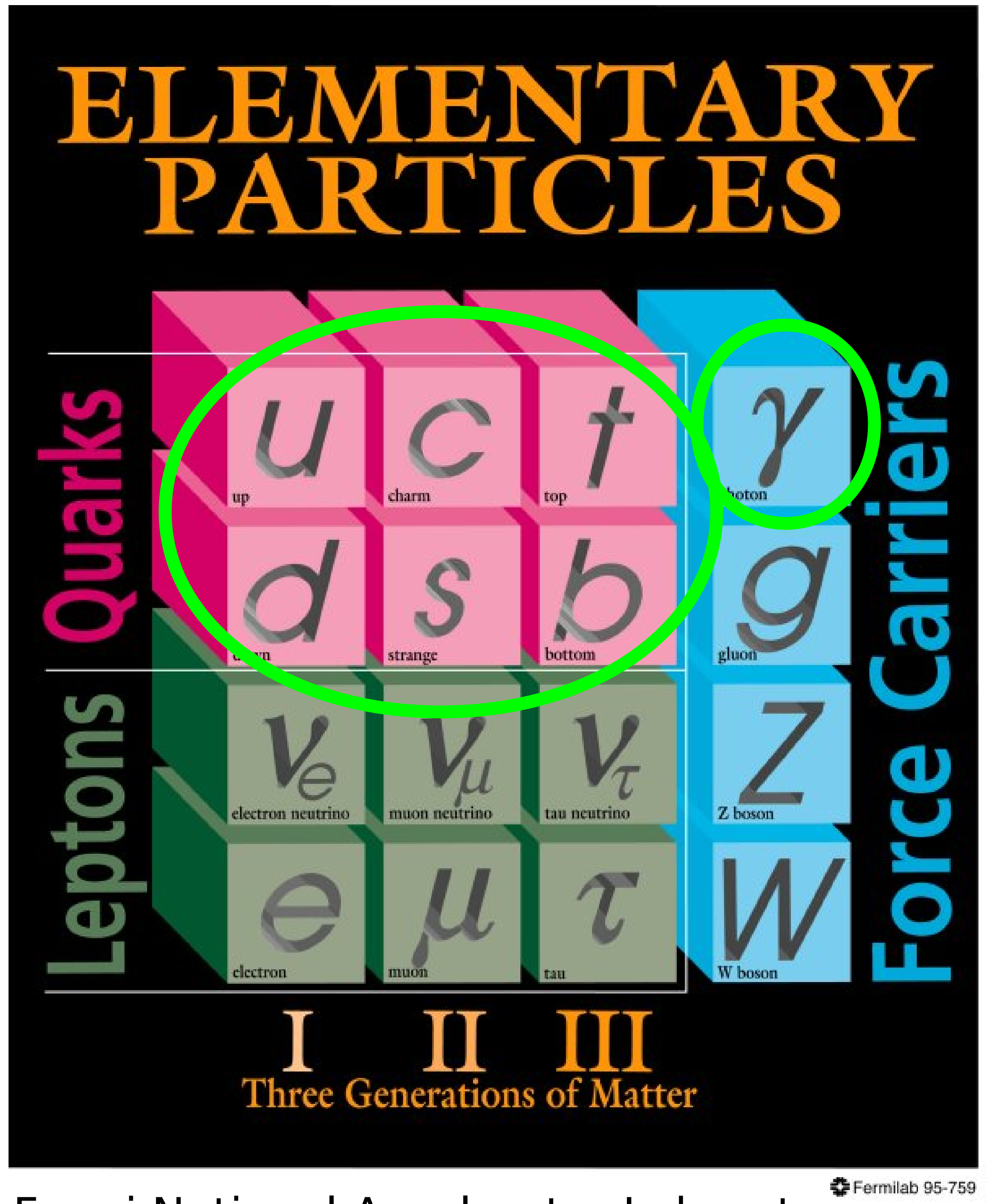
The strong force is one of the basic forces of nature
(along with gravity, em and weak)

It's what binds together the quarks and gluons in the proton and the neutron (as well as hundreds of other particles seen in accelerator experiments)

QCD is the theory of the strong force
It's a beautiful theory…

$$\langle\Omega\rangle = \frac{1}{Z}\int [dU]\, e^{-\int d^4x\, L(U)}\, \Omega(U)$$

…but



Fermi National Accelerator Laboratory

LATTICE QUANTUM CHROMODYNAMICS

Theory is highly non-linear ⇒ cannot solve directly

Must resort to numerical methods to make predictions

Lattice QCD

- Discretize spacetime ⇒ 4-d dimensional lattice of size L_x x L_y x L_z x L_t
- Finitize spacetime ⇒ periodic boundary conditions
- PDEs ⇒ finite difference equations

High-precision tool that allows physicists to explore the contents of nucleus from the comfort of their workstation (supercomputer)

Consumer of 10-20% of public supercomputer cycles

STEPS IN AN LQCD CALCULATION

1. Generate an ensemble of gluon field (“gauge”) configurations

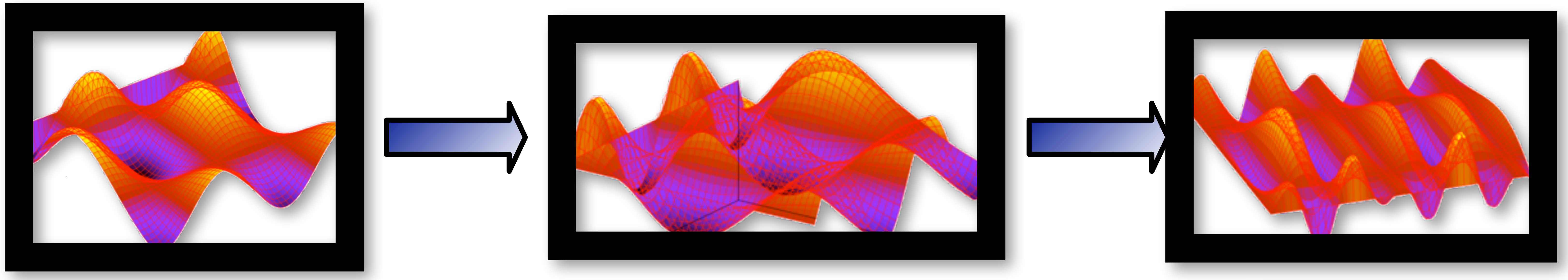
    Produced in sequence, with hundreds needed per ensemble

    Strong scaling required with O(100 TFLOPS) sustained for several months

    50-90% of the runtime is in the linear solver

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

or *Ax = b*

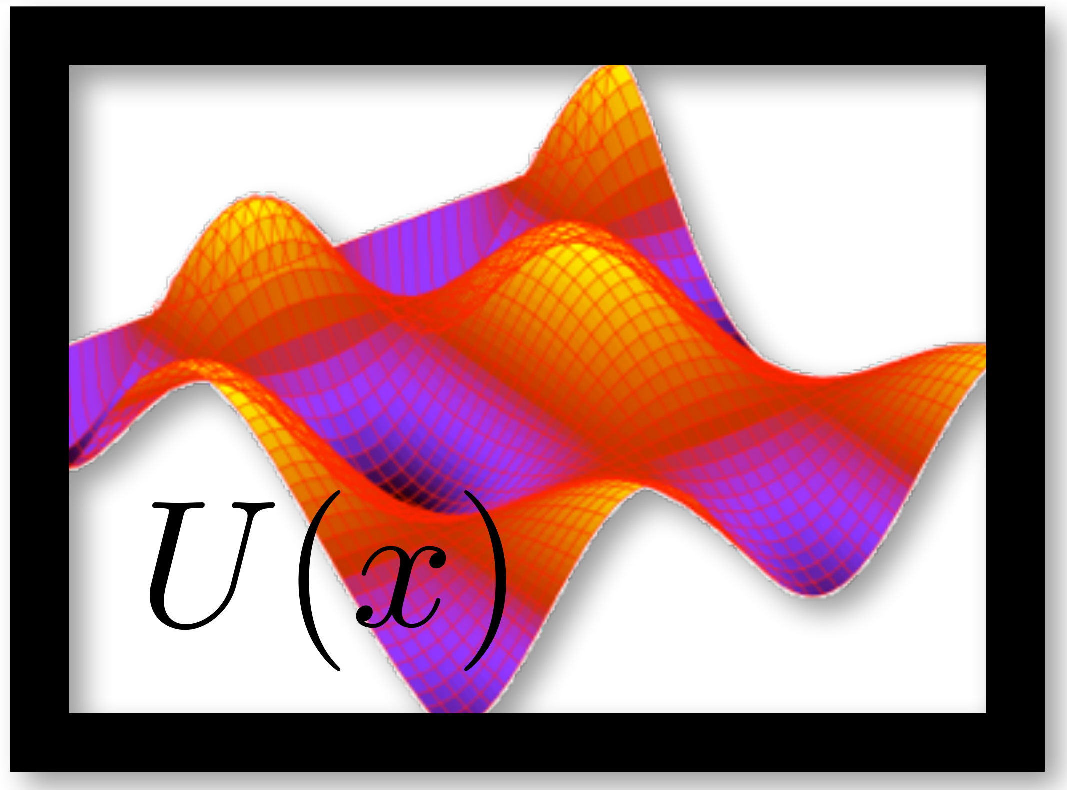


2. “Analyze” the configurations

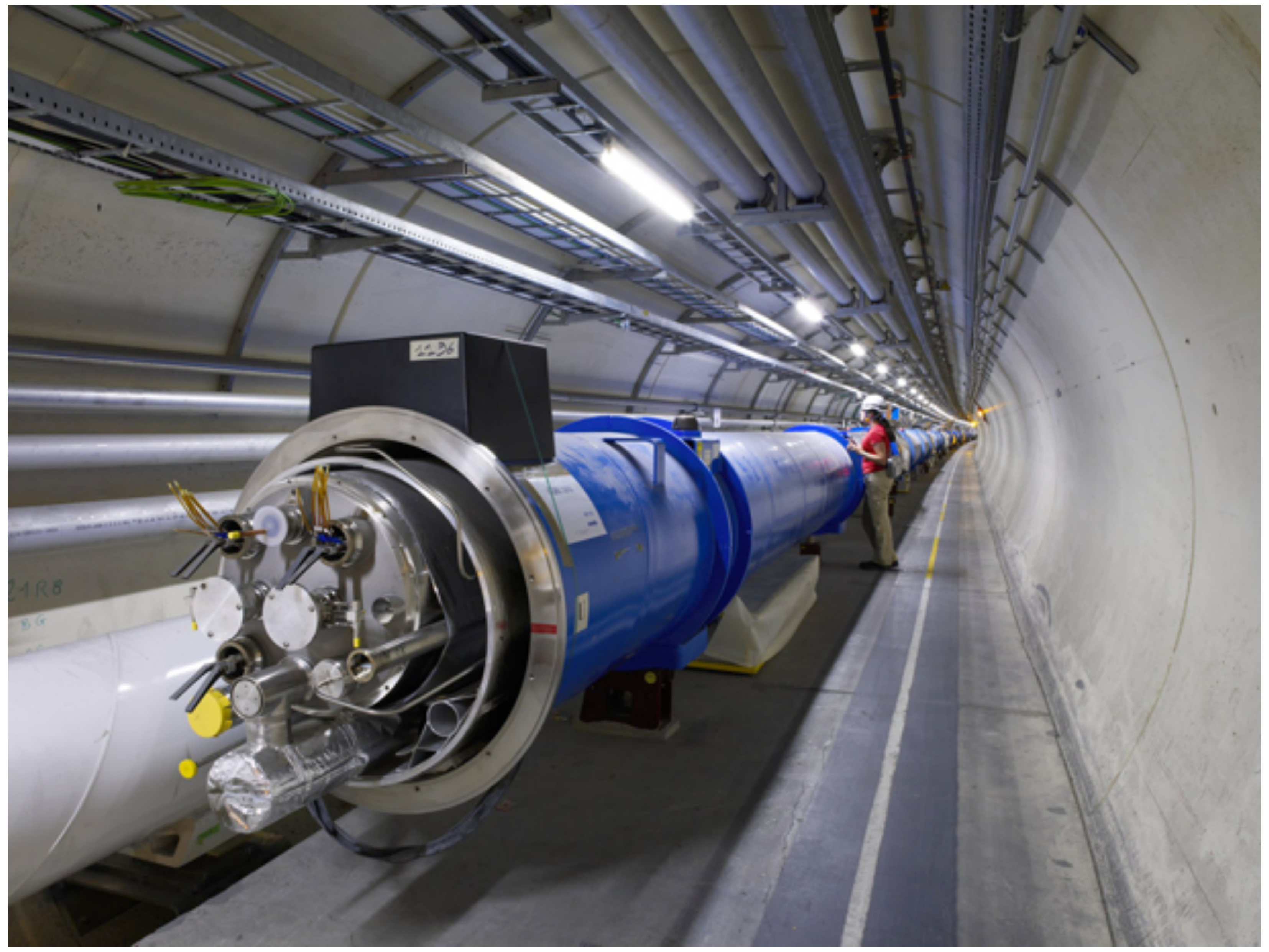
    Can be farmed out, assuming O(1 TFLOPS) per job.

    80-99% of the runtime is in the linear solver

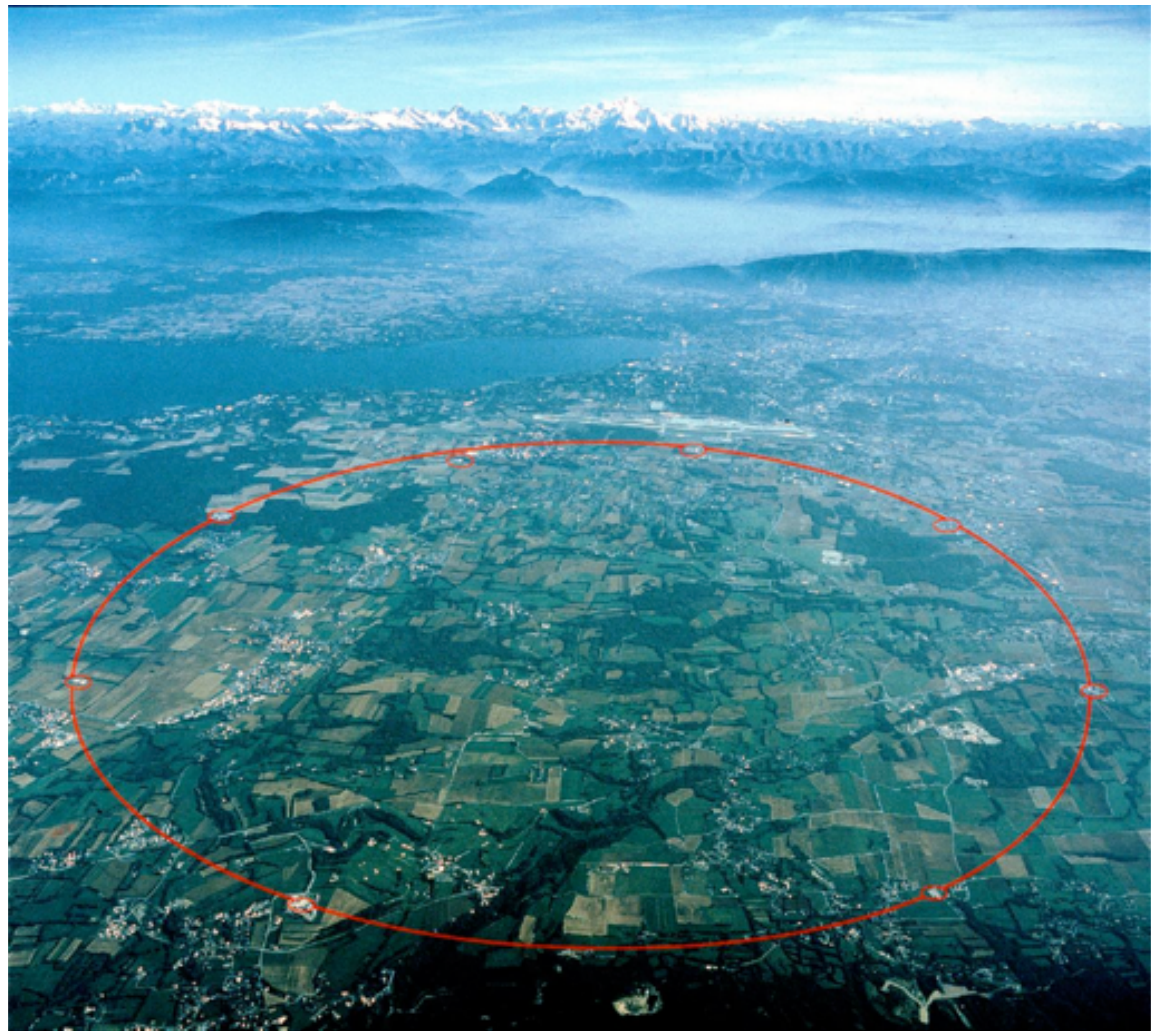
    Task parallelism means that clusters reign supreme here

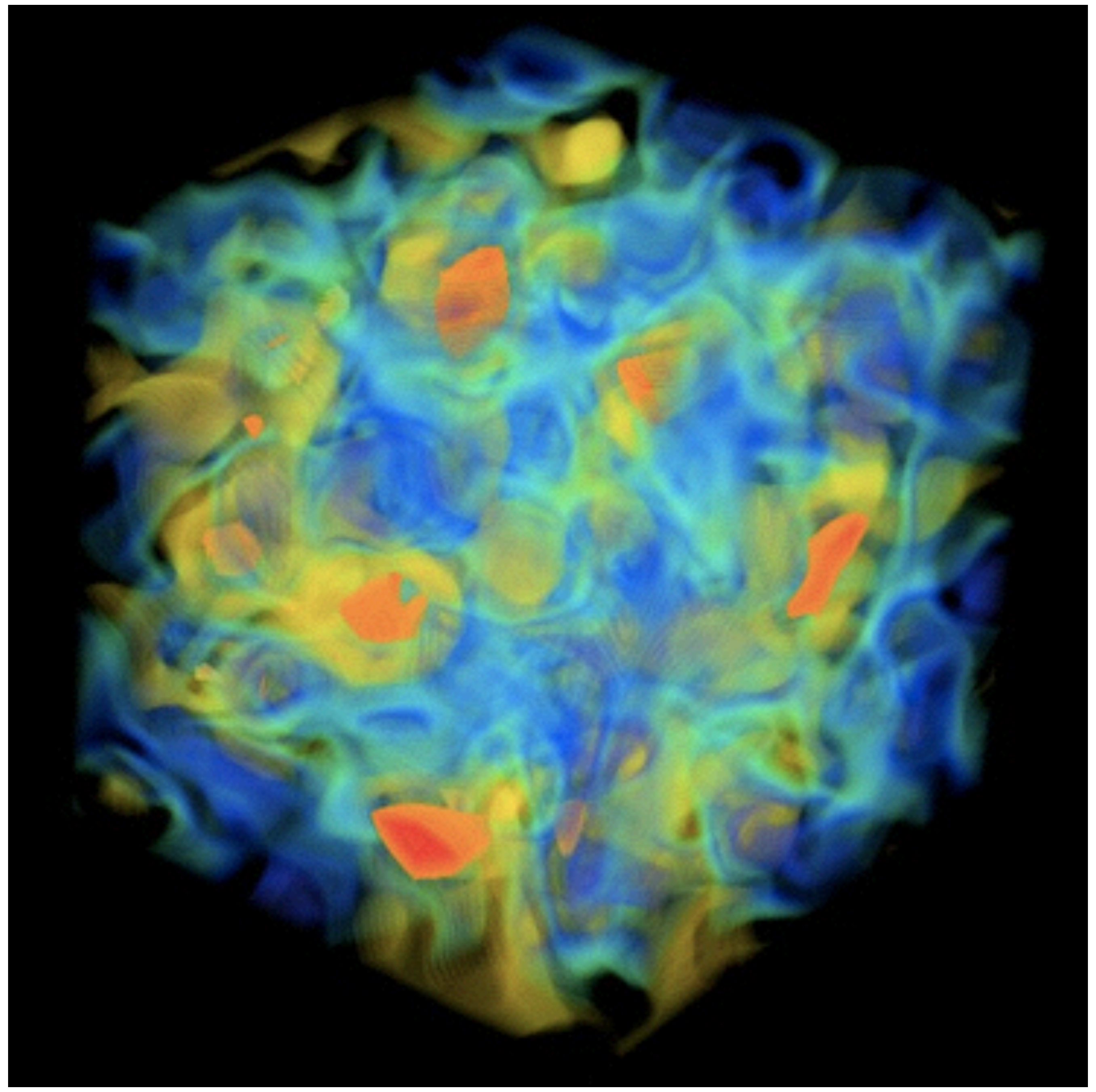


Large Hadron Collider

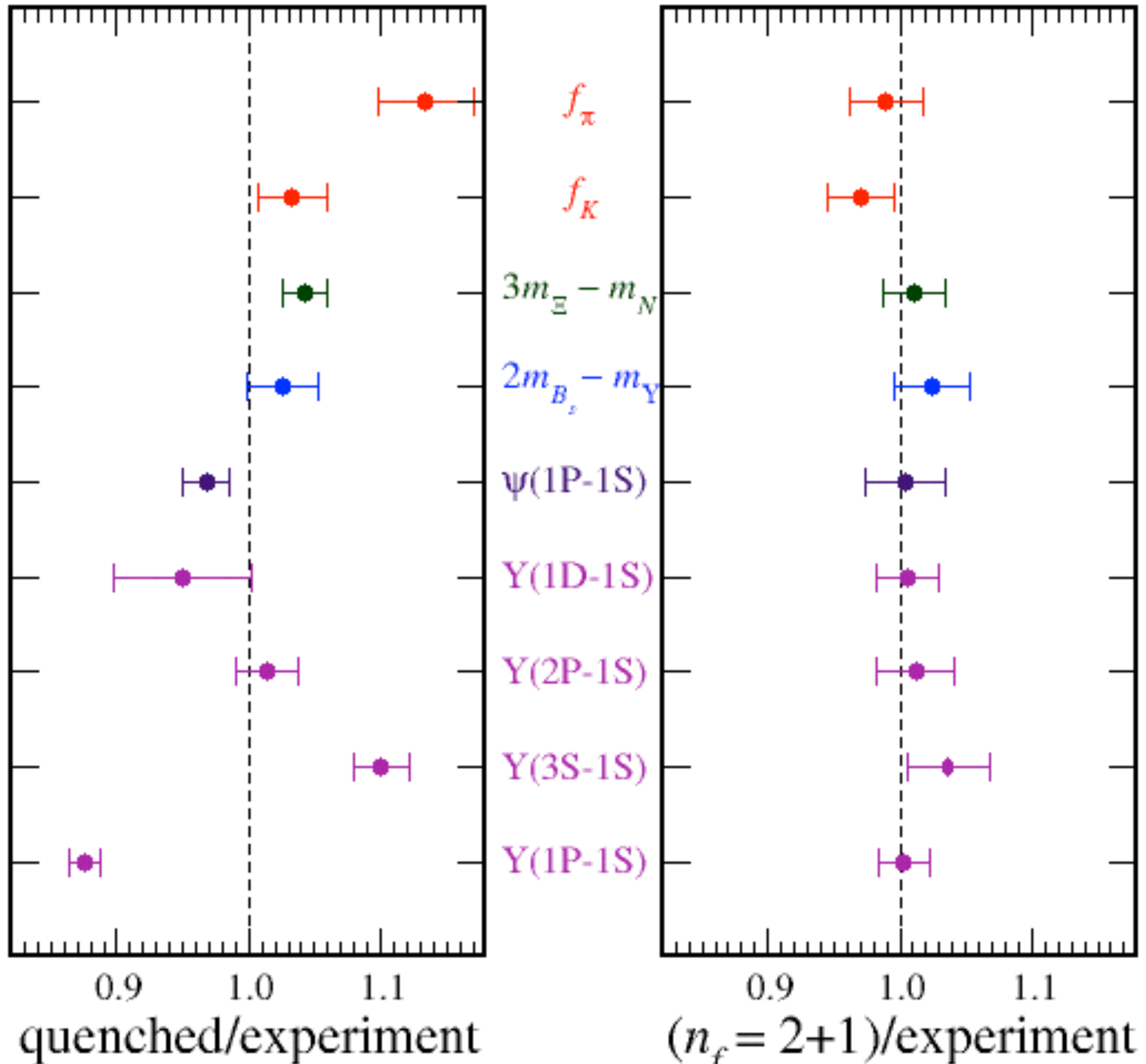


Large Hadron Collider





Brookhaven National Laboratory



Davies *et al*

QUDA

- “QCD on CUDA” – http://lattice.github.com/quda (open source)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, etc.
  - Latest release 0.8.0 (8th February 2016)
- 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)
  - Multigrid solvers for optimal convergence
- A research tool for how to reach the exascale

TITAN

QUDA COLLABORATORS

(multigrid collaborators in green)

Ron Babich (NVIDIA)
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)
Dean Howarth (Rensselaer Polytechnic Institute)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Claudio Rebbi (Boston University)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Alejandro Vaquero (INFN)
Mathias Wagner (NVIDIA)
Frank Winter (Jlab)

THE DIRAC OPERATOR

Quark interactions are described by the Dirac operator

- First-order PDE acting with a background field
- Large sparse matrix

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

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

A is the clover matrix (12x12 *spin*⊗*color* space)

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

m quark mass parameter

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

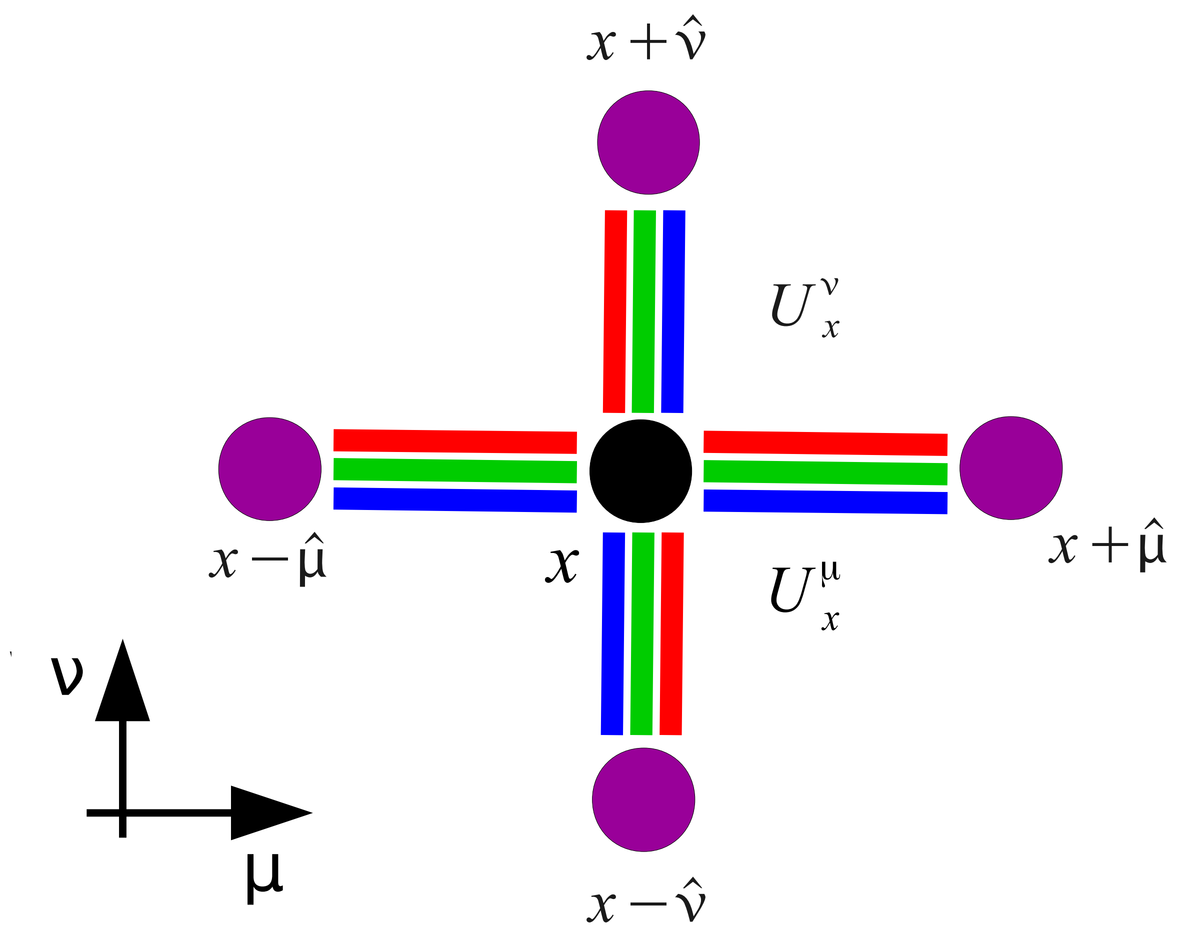
4-d nearest neighbor stencil operator acting on a vector field

Eigen spectrum is complex (typically real positive)

MAPPING THE DIRAC OPERATOR TO CUDA

- Finite difference operator in LQCD is known as Dslash
- Assign a single space-time point to each thread
  - V = XYZT threads, e.g., V = 24^4 => 3.3×10^6 threads
- Looping over direction each thread must
  - Load the neighboring spinor (24 numbers x8)
  - Load the color matrix connecting the sites (18 numbers x8)
  - Do the computation
  - Save the result (24 numbers)
- Each thread has (Wilson Dslash) 0.92 naive arithmetic intensity
- QUDA reduces memory traffic
  - Exact SU(3) matrix compression (18 => 12 or 8 real numbers)
  - Use 16-bit fixed-point representation with mixed-precision solver

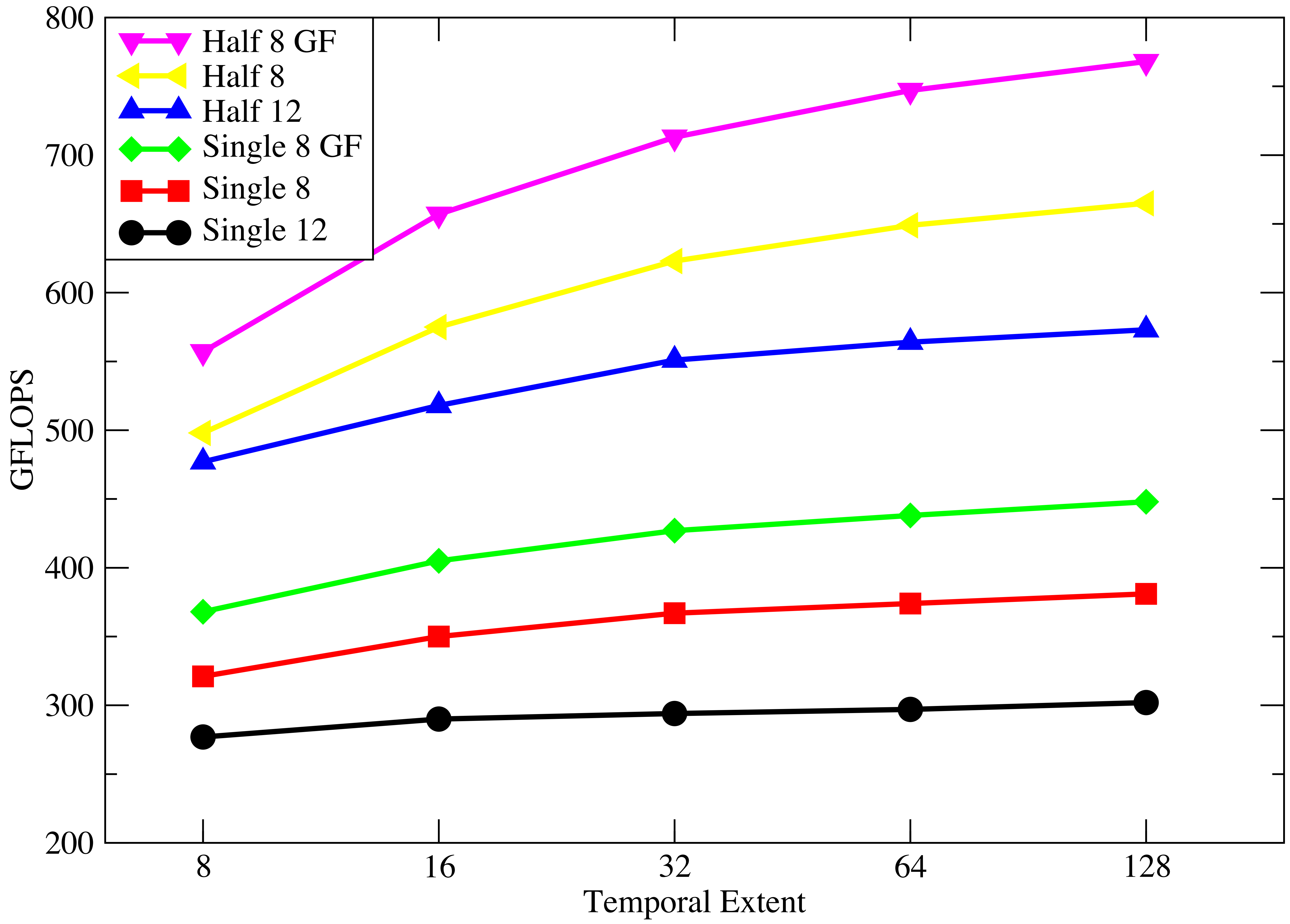
$$D_{x,x'} =$$



GPU TECHNOLOGY CONFERENCE

WILSON-DSLASH PERFORMANCE

K20X, ECC on, V = 24^3xT



LINEAR SOLVERS

QUDA supports a wide range of linear solvers

- CG, BiCGstab, GCR, Multi-shift solvers, etc.

Condition number inversely proportional to mass

- Light (realistic) masses are highly singular
- Naive Krylov solvers suffer from critical slowing down at decreasing mass

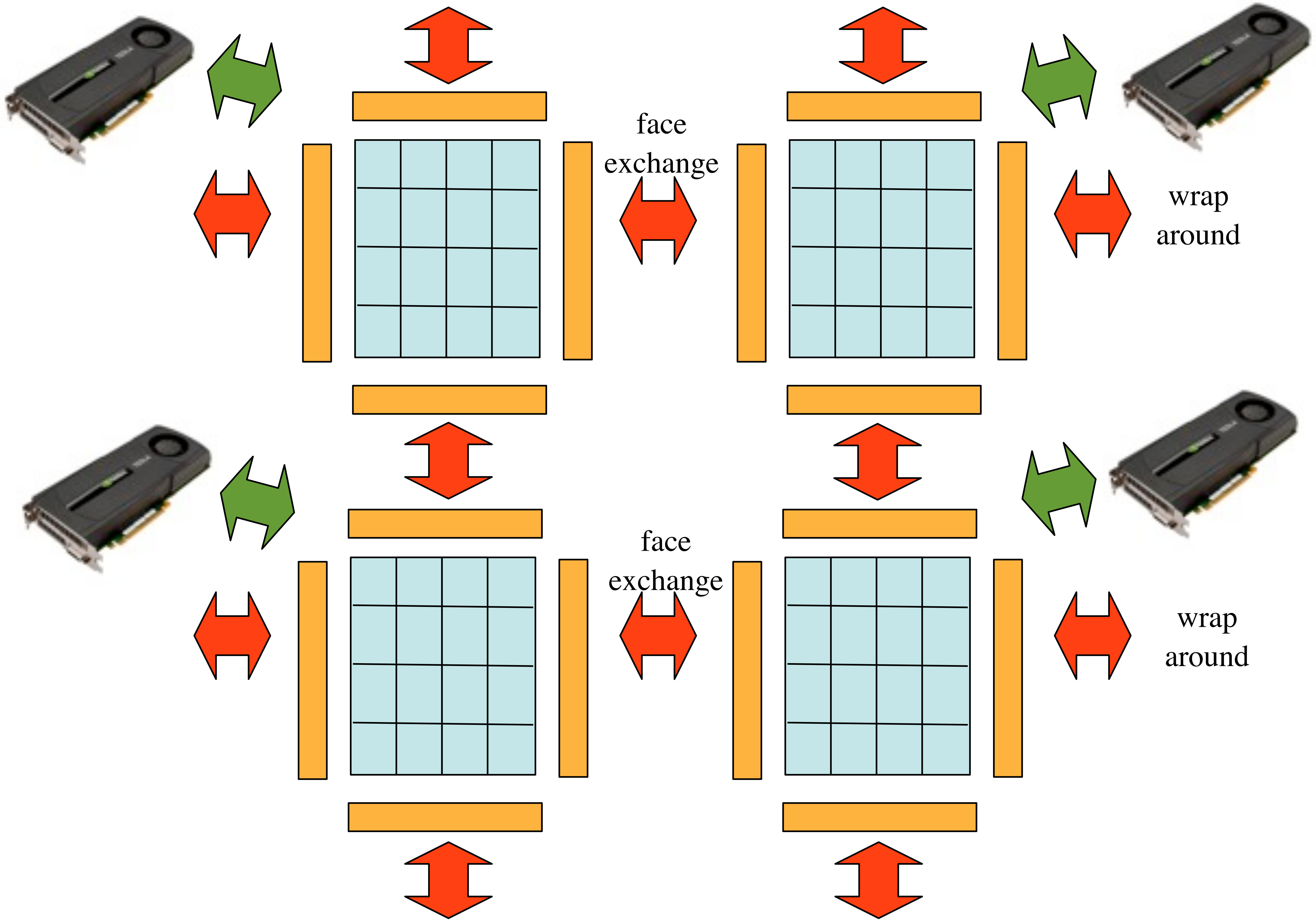
Entire solver algorithm must run on GPUs

- Time-critical kernel is the stencil application
- Also require BLAS level-1 type operations

```
while ($|\mathbf{r}_k| > \varepsilon$) {
    $\beta_k = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{r}_{k-1},\mathbf{r}_{k-1})$
    $\mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k\mathbf{p}_k$
    $\mathbf{q}_{k+1} = A\ \mathbf{p}_{k+1}$
    $\alpha = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{p}_{k+1}, \mathbf{q}_{k+1})$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha\mathbf{q}_{k+1}$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\mathbf{p}_{k+1}$
    $k = k+1$
}
```

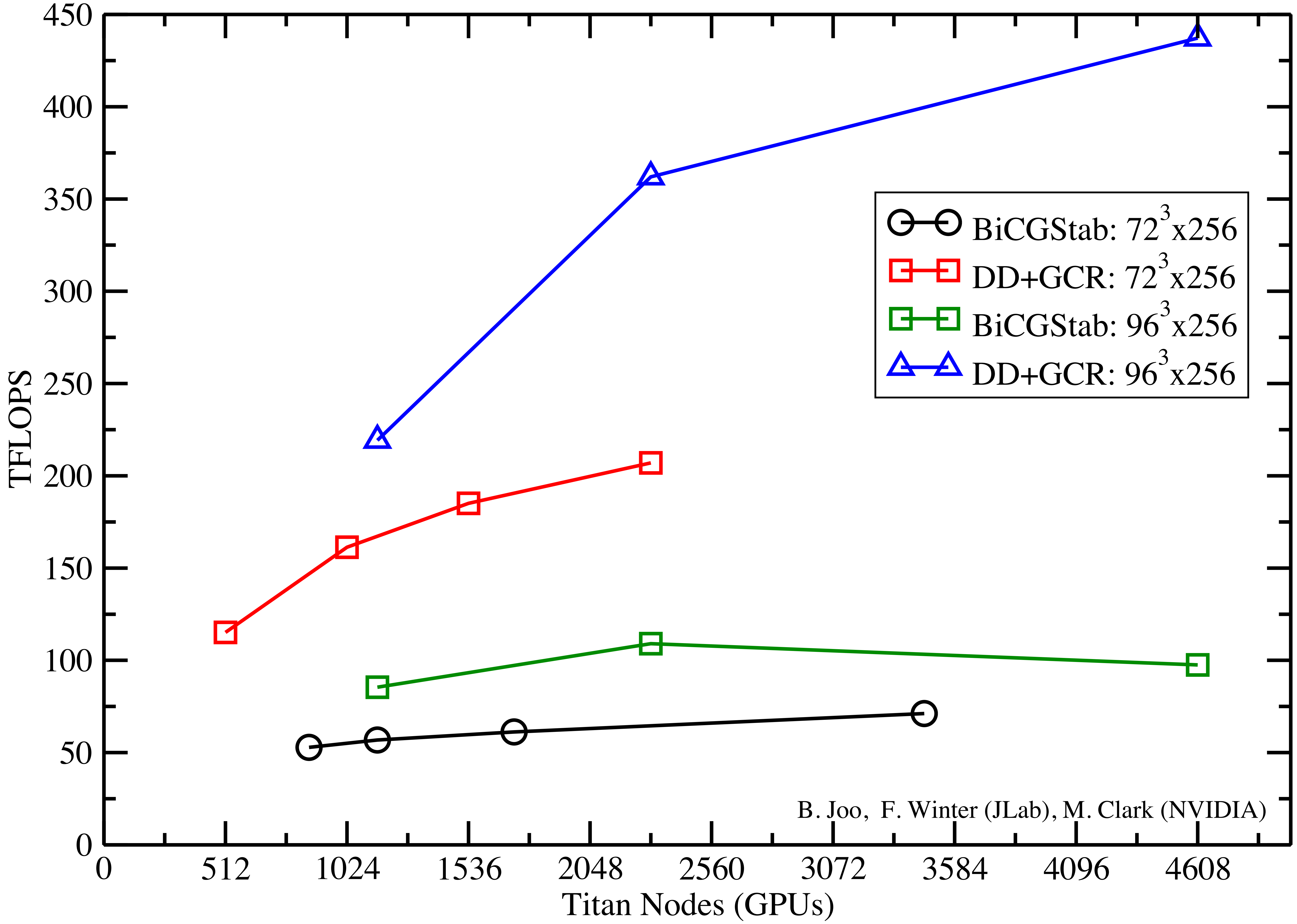
conjugate gradient

MULTI-GPU DECOMPOSITION



STRONG SCALING

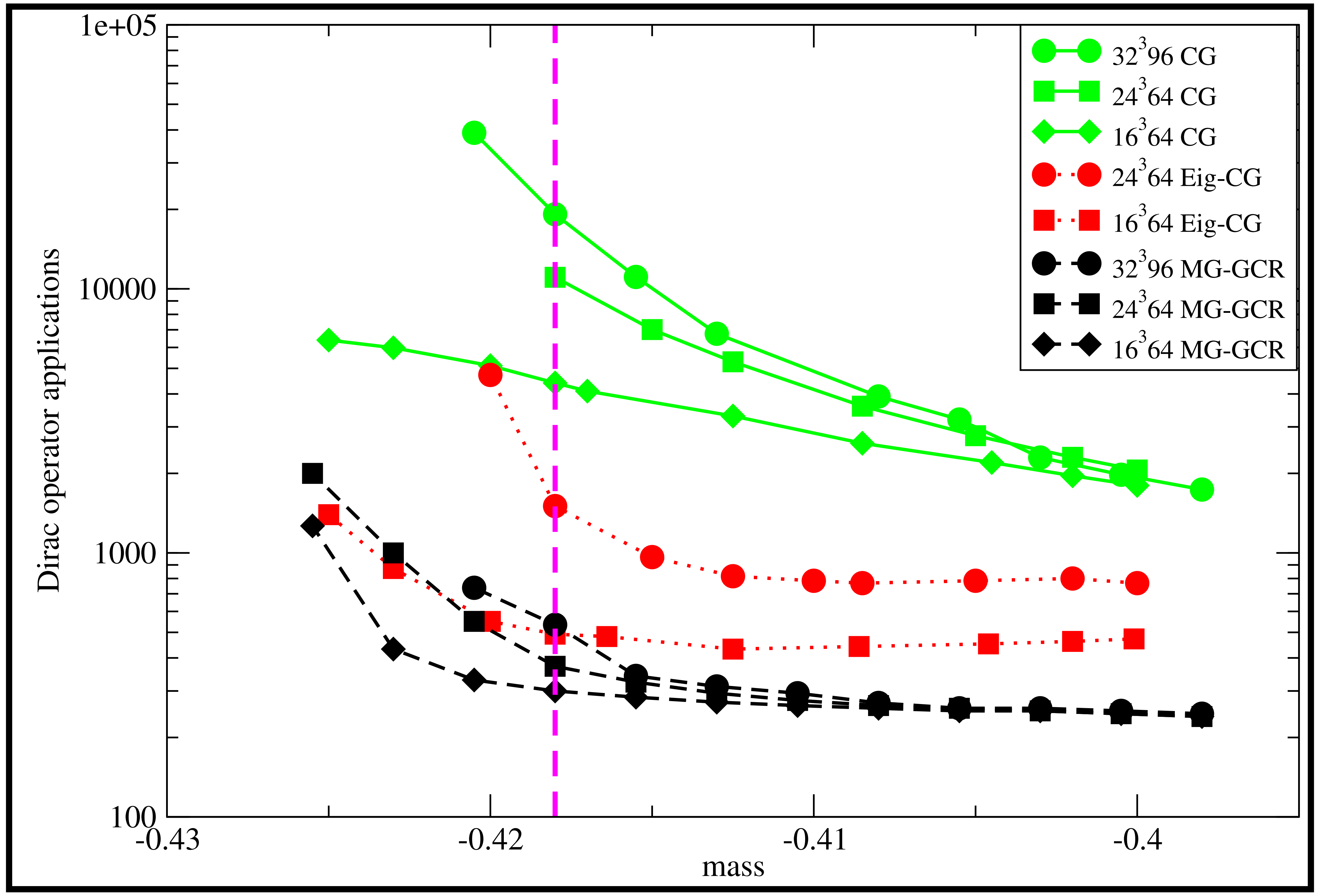
Chroma running on Titan with QUDA



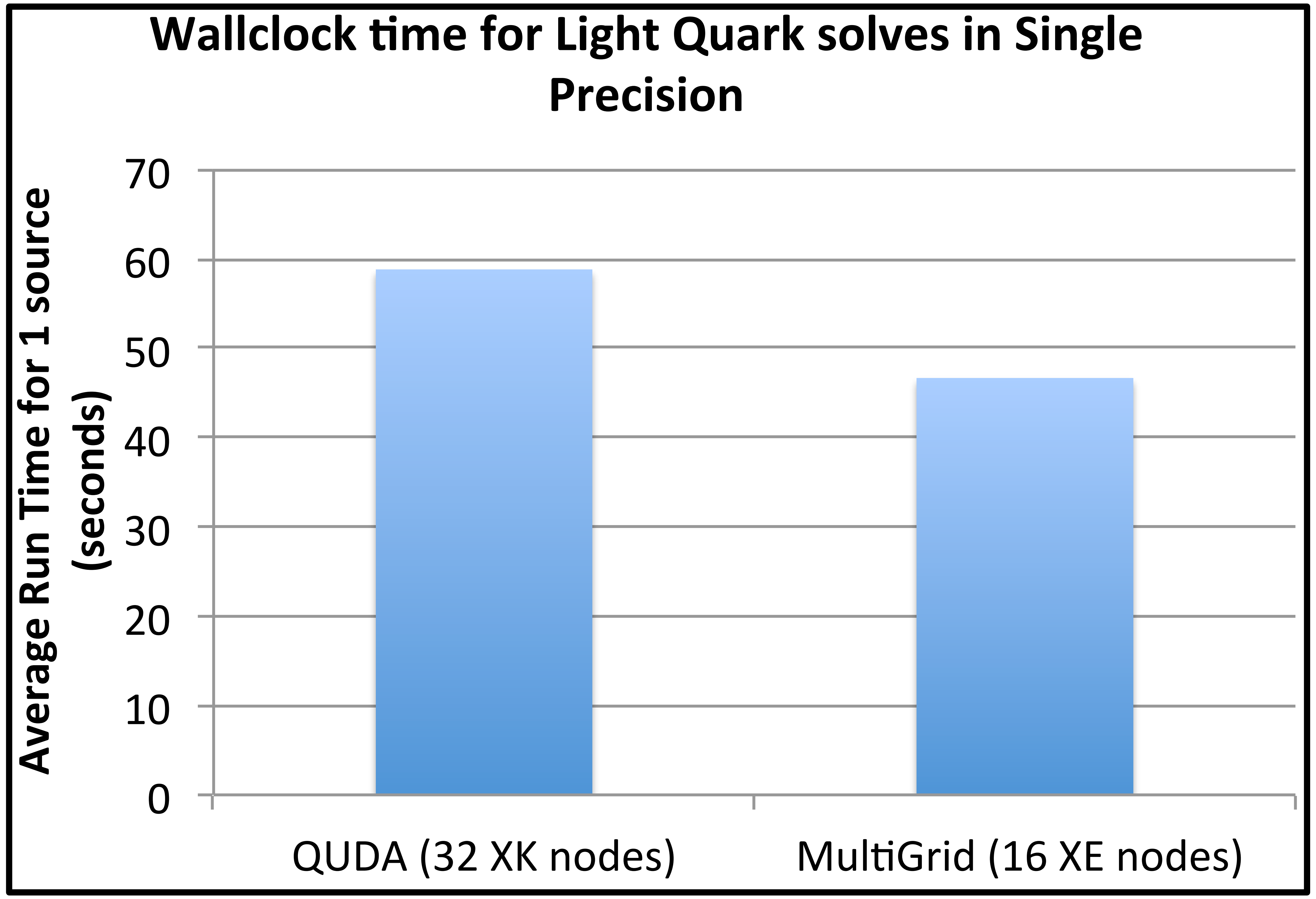
B. Joo, F. Winter (JLab), M. Clark (NVIDIA)

ADAPTIVE MULTIGRID

WHY MULTIGRID?



Babich *et al* 2010



Chroma propagator benchmark
Figure by Balint Joo
MG Chroma integration by Saul Cohen
MG Algorithm by James Osborn

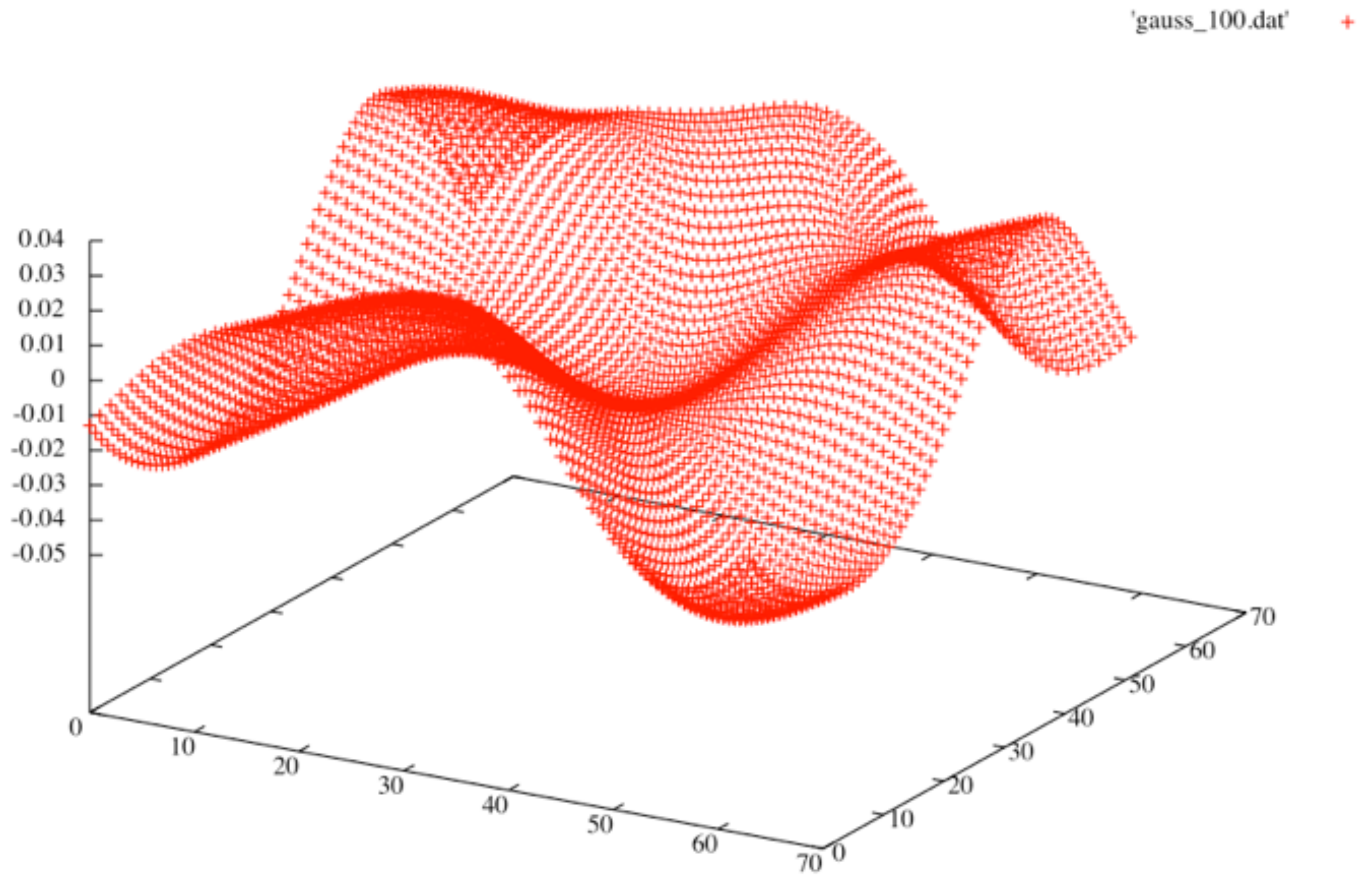
INTRODUCTION TO MULTIGRID

Stationary iterative solvers effective on high frequency errors

Minimal effect on low frequency error

Example

- Free Laplace operator in 2d
- $Ax = 0$, x_0 = random
- Gauss Seidel relaxation
- Plot error $e_i = -x_i$



INTRODUCTION TO MULTIGRID

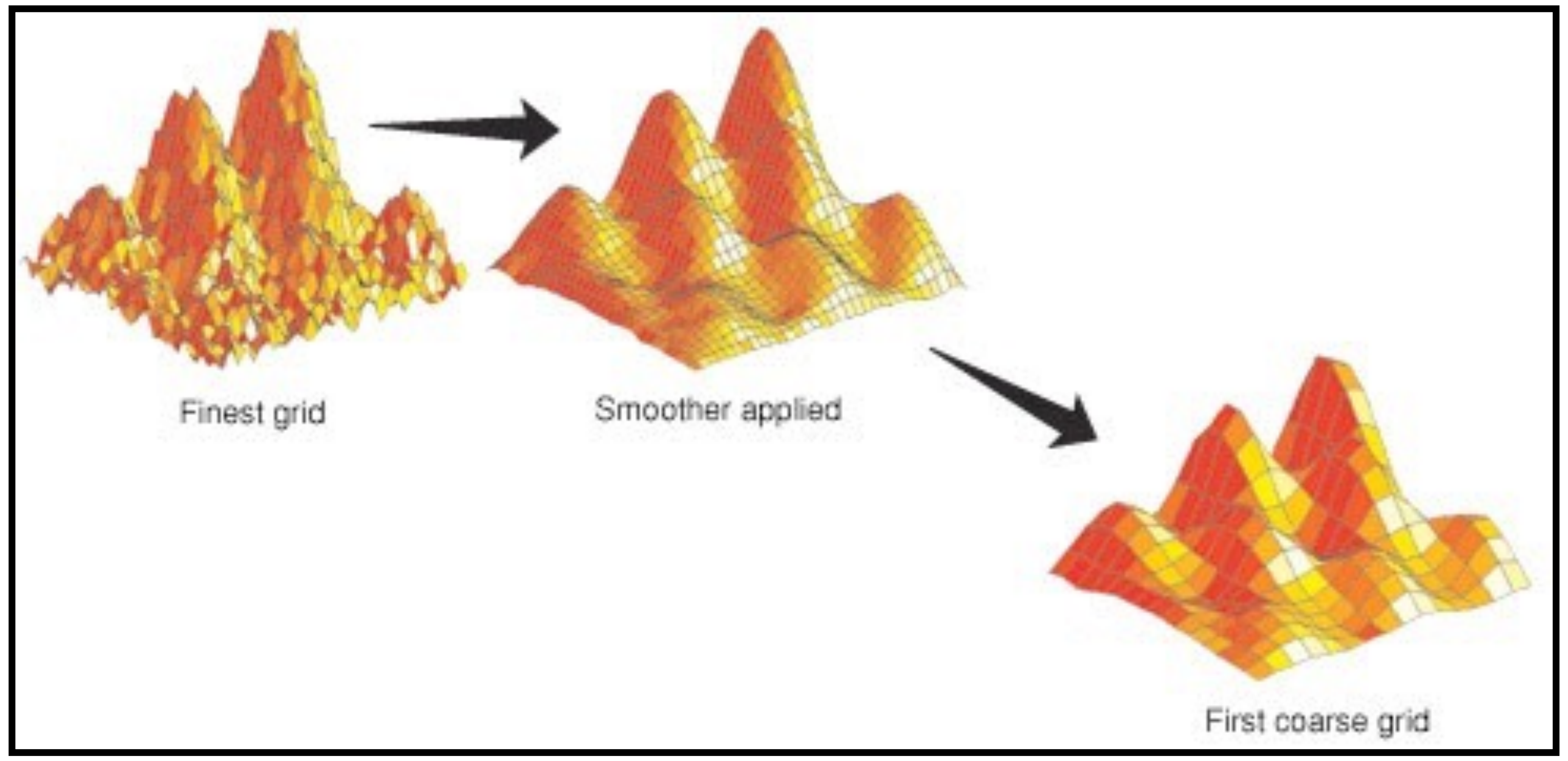
Low frequency error modes are smooth

Can accurately represent on coarse grid

Low frequency on fine
  => high frequency on coarse

Relaxation effective agin on coarse grid

Interpolate back to fine grid



Falgout

MULTIGRID V-CYCLE

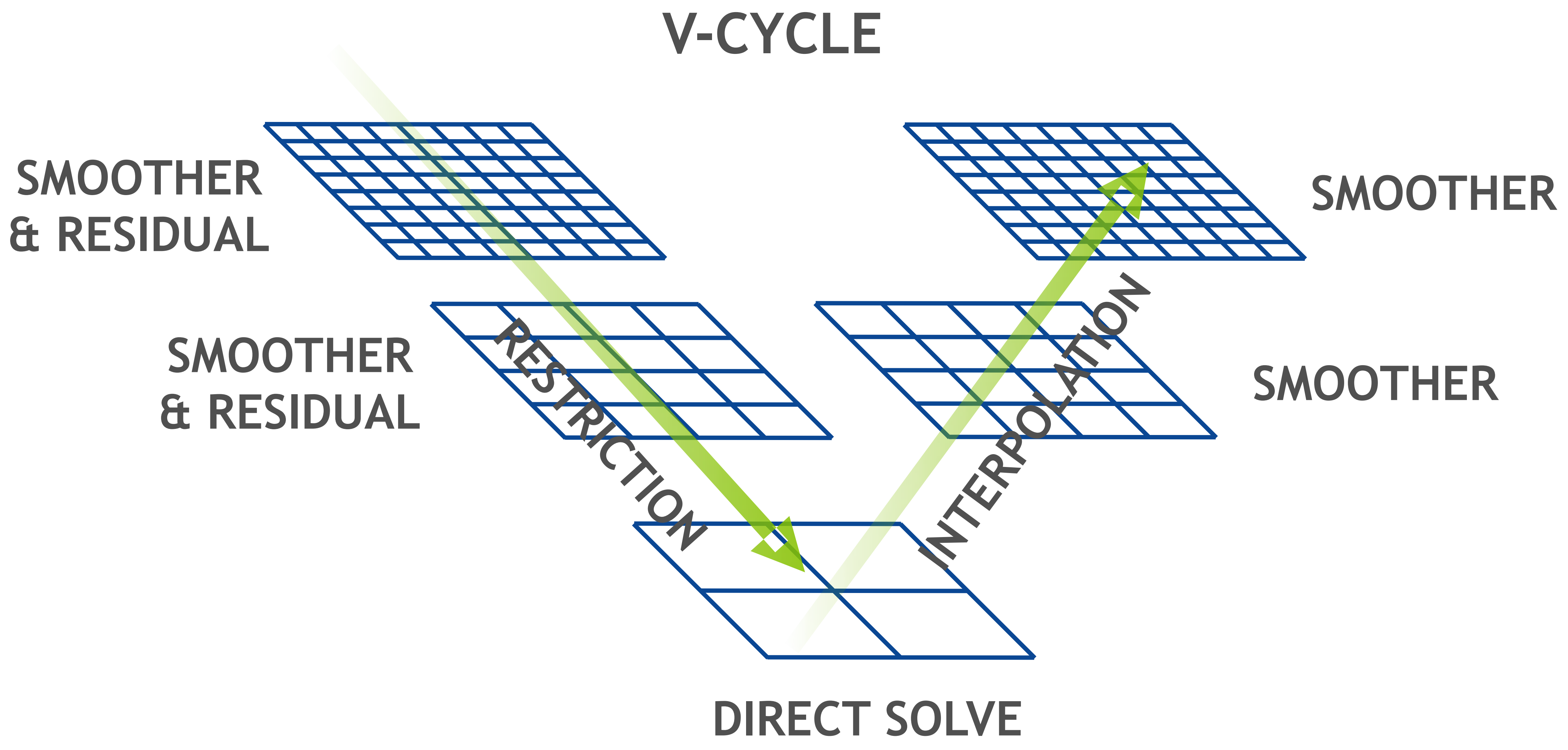
Solve

1. Smooth
2. Compute residual
3. Restrict residual
4. Recurse on coarse problem
5. Prolongate correction
6. Smooth
7. If not converged, goto 1

Multigrid has optimal scaling

- *O(N)* Linear scaling with problem size
- Convergence rate independent of condition number

For LQCD, we do not know the null space components that need to be preserved on the coarse grid



ADAPTIVE GEOMETRIC MULTIGRID

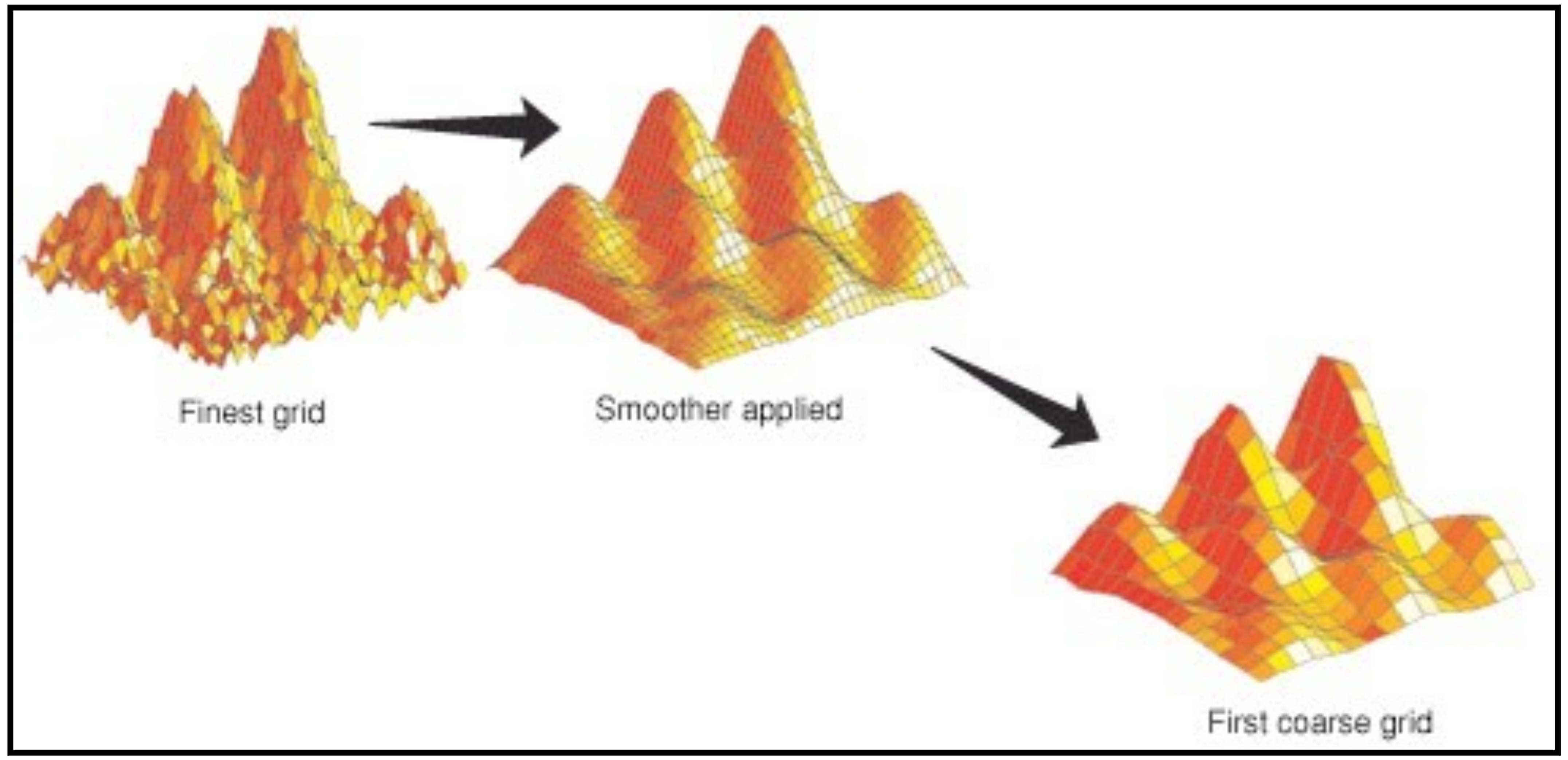
Adaptively find candidate null-space vectors

- 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

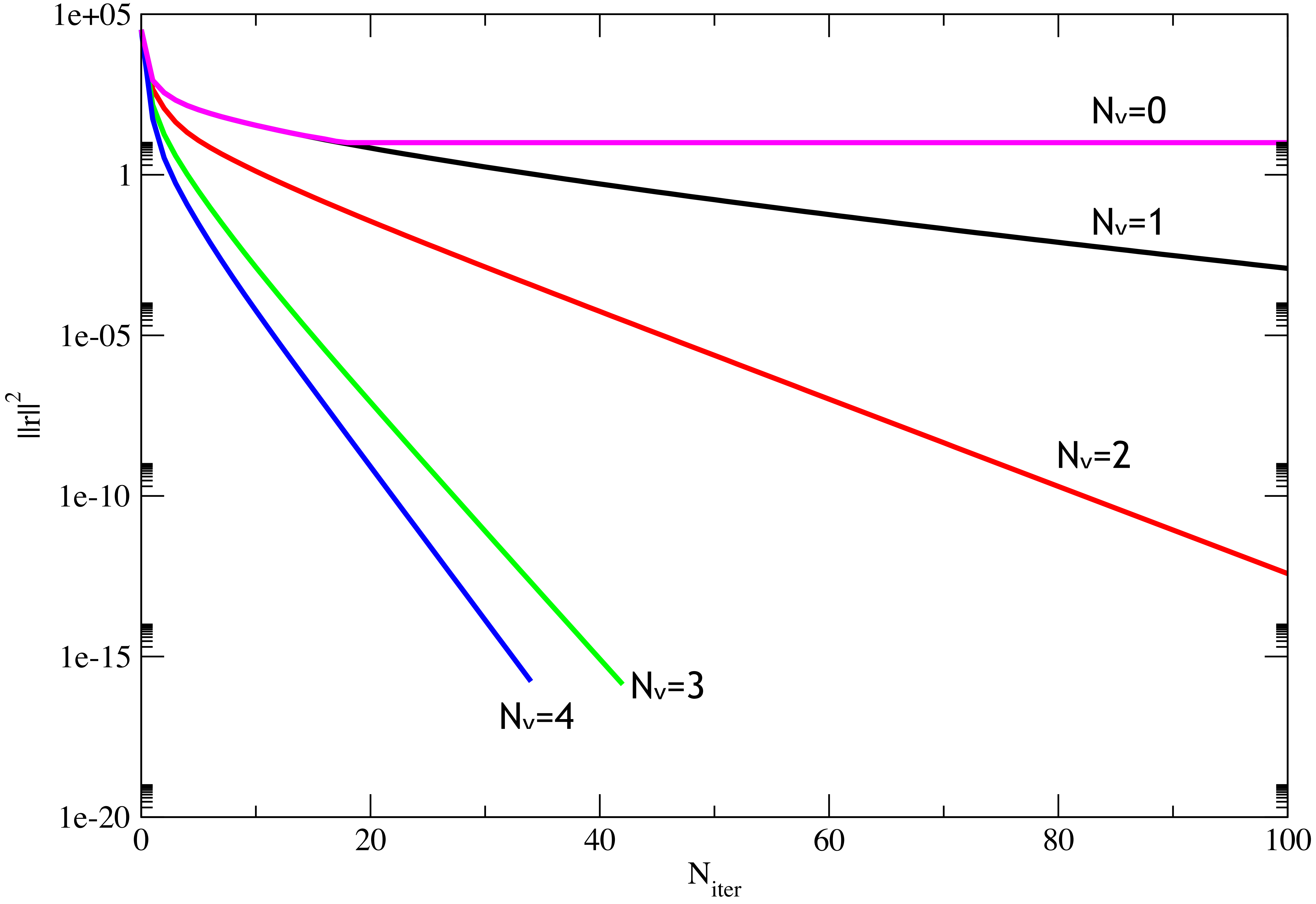
Babich *et al* 2010



Falgout

ADAPTIVE GEOMETRIC MULTIGRID

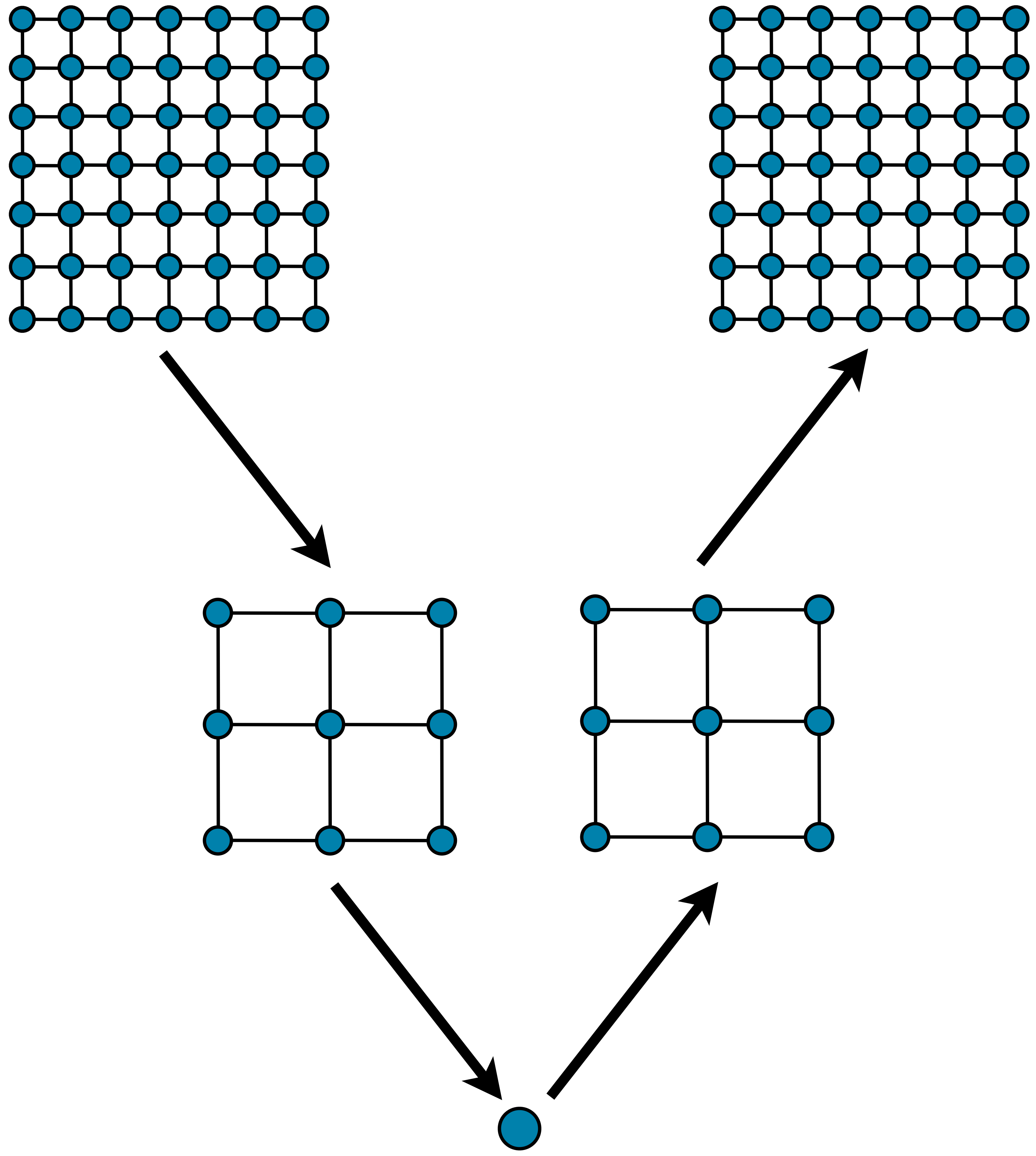
4-d Laplace operator



Typically 20-30 vectors needed to capture Dirac null space

MULTIGRID ON GPUS

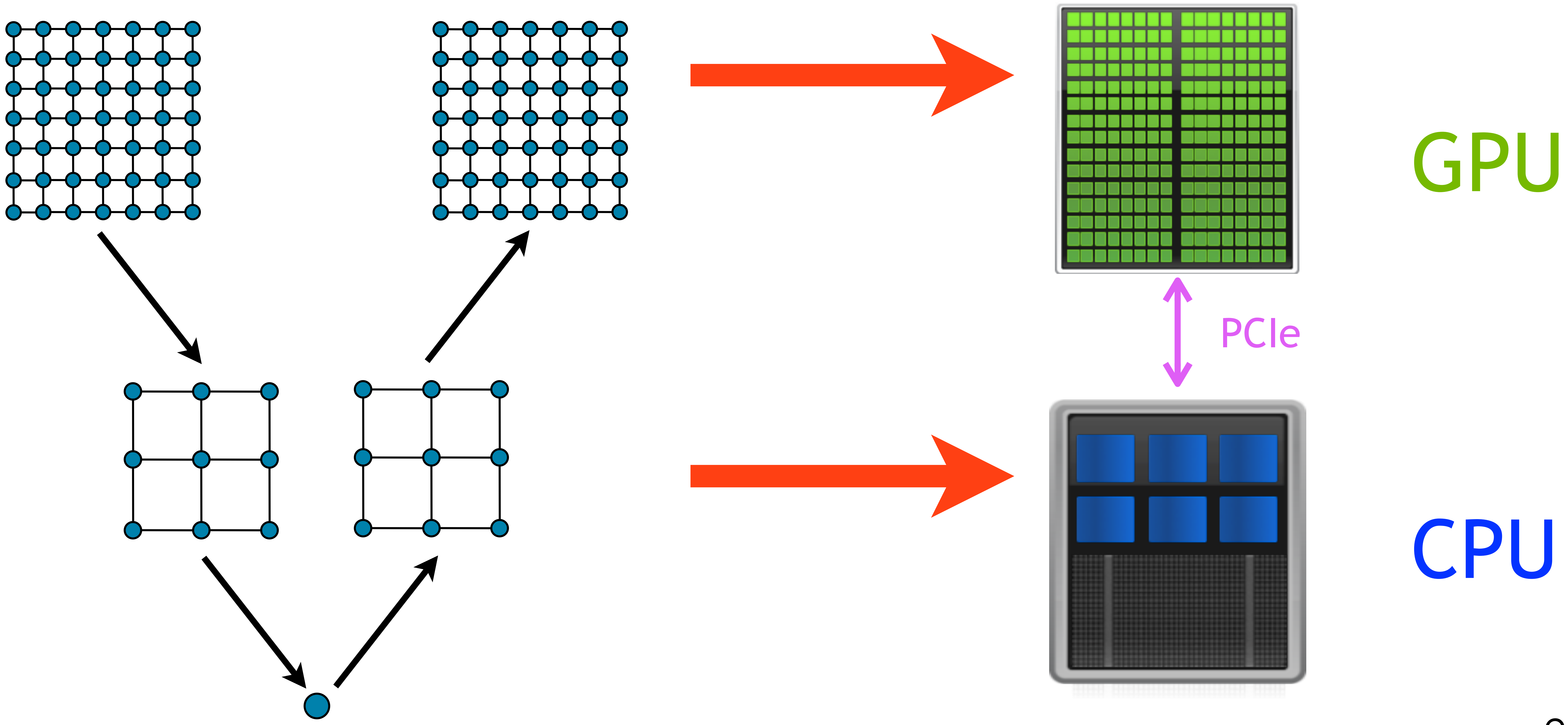
THE CHALLENGE OF MULTIGRID ON GPU



- 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

THE CHALLENGE OF MULTIGRID ON GPU



DESIGN GOALS

Performance

- LQCD typically reaches high % peak peak performance
- Brute force can beat the best algorithm
- Multigrid must be optimized to the same level

Flexibility

- Deploy level *i* on either CPU or GPU
- All algorithmic flow decisions made at runtime
- Autotune for a given *heterogeneous*

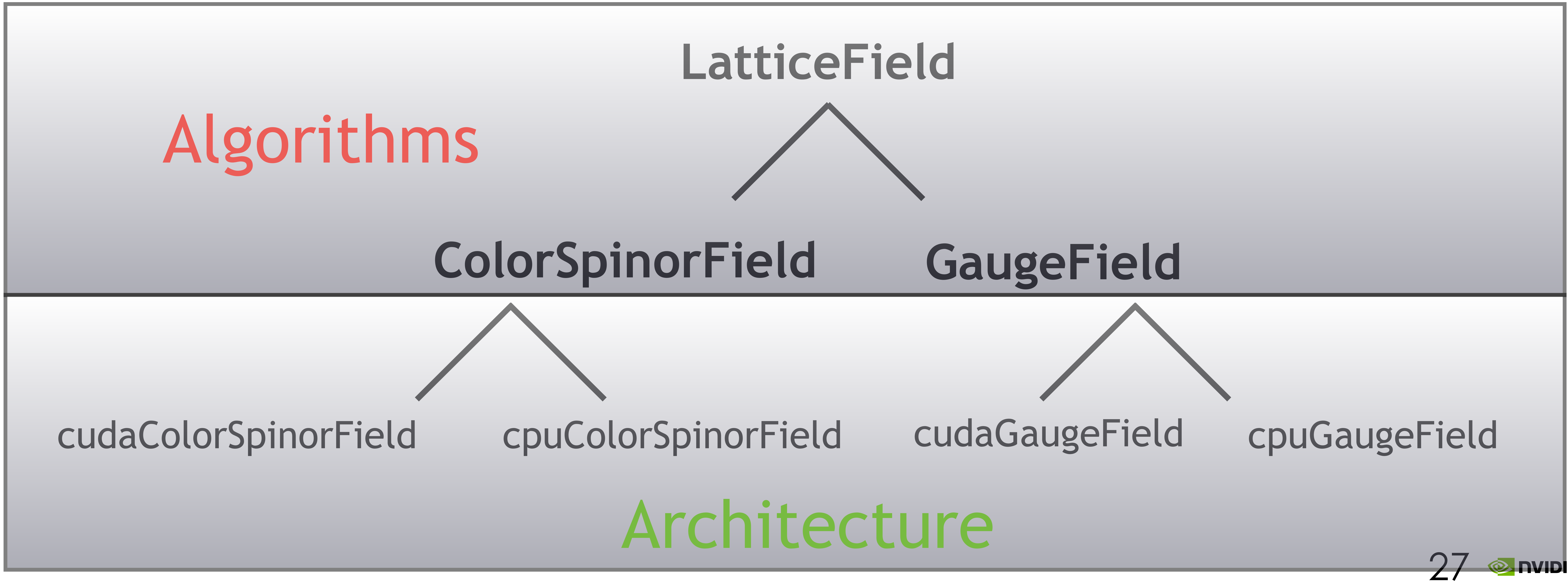
(Short term) Provide optimal solvers to legacy apps

- Initial target analysis computations, e.g., 100,000 linear solves per linear system
- Focus on final solver performance

(Long term) Hierarchical algorithm toolbox

MULTIGRID AND QUDA

QUDA designed to abstract algorithm from the heterogeneity



WRITING THE SAME CODE FOR TWO ARCHITECTURES

```
template<…> __host__ __device__ Real bar(Arg &arg, int x) {
  // do platform independent stuff here
  complex<Real> a[arg.length];
  arg.A.load(a);

  … // do computation

  arg.A.save(a);
  return norm(a);
}
```

platform specific load/store hidden here:
field order, cache modifiers, textures

platform independent stuff goes here
99% of computation goes here

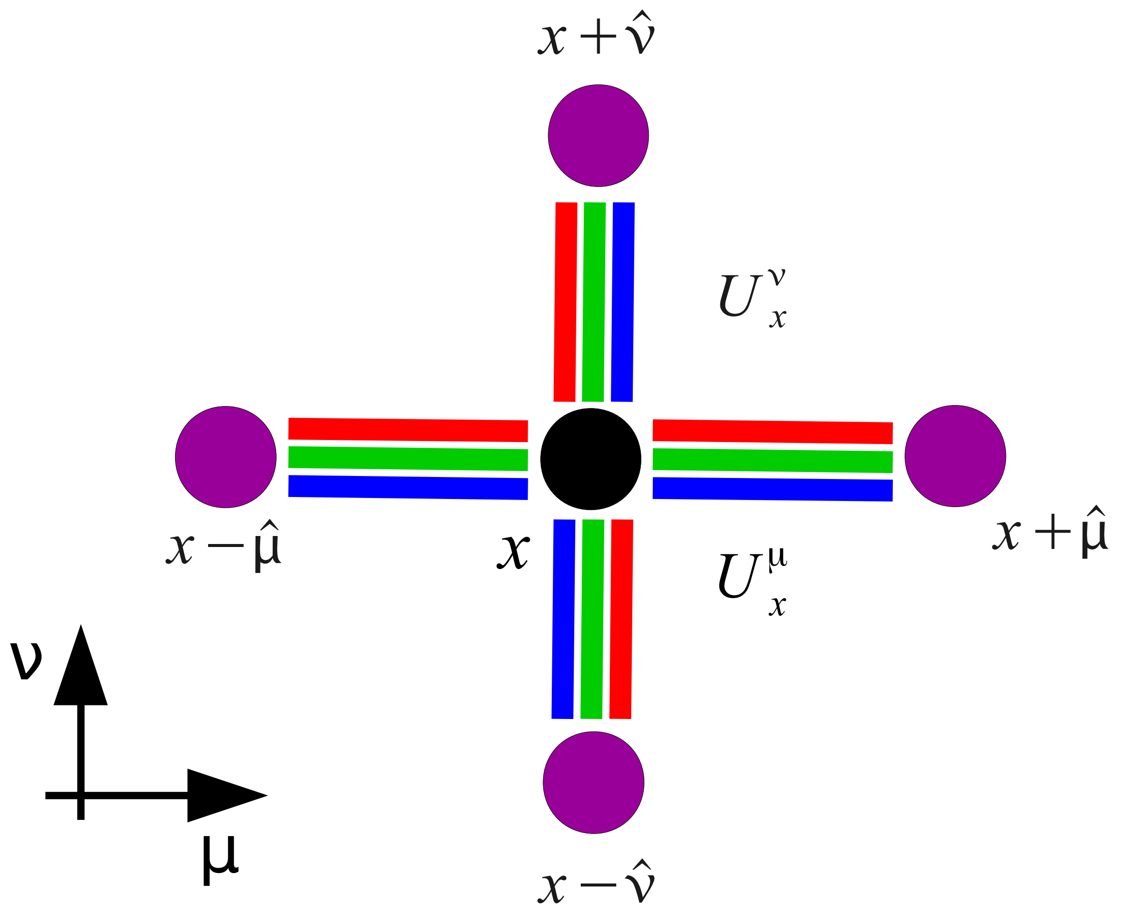
```
template<…> void fooCPU(Arg &arg) {
  arg.sum = 0.0;
#pragma omp for
  for (int x=0; x<size; x++)
    arg.sum += bar<…>(arg, x);
}
```

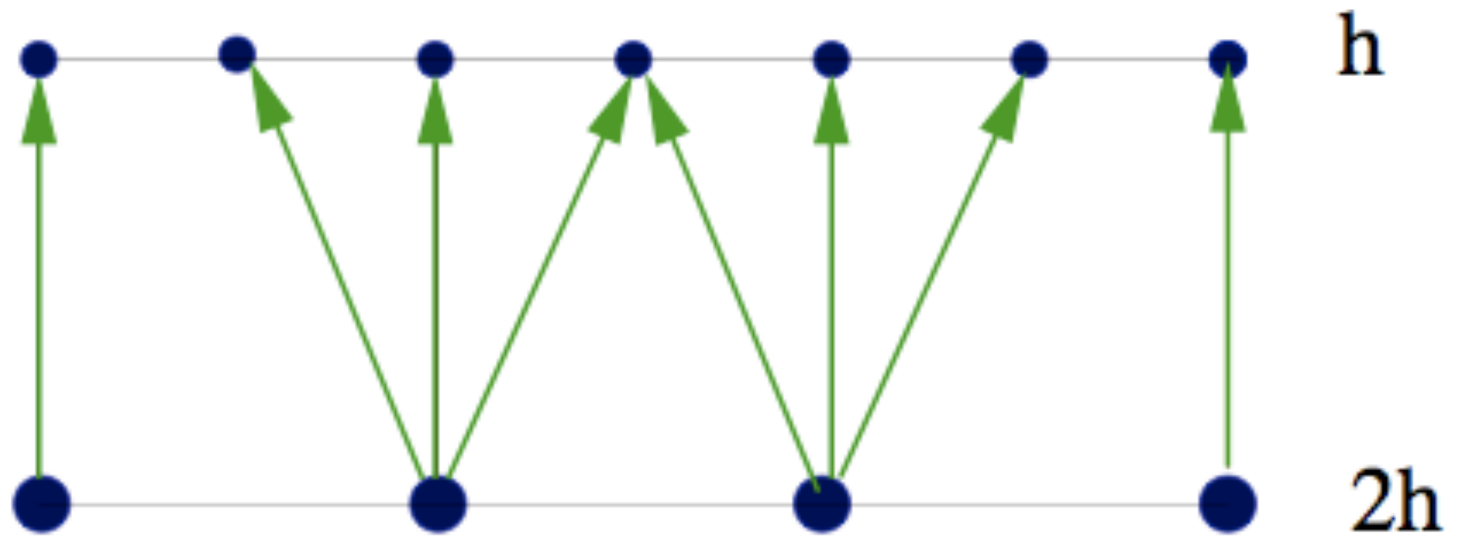
platform specific parallelization
GPU: shared memory
CPU: OpenMP, vectorization

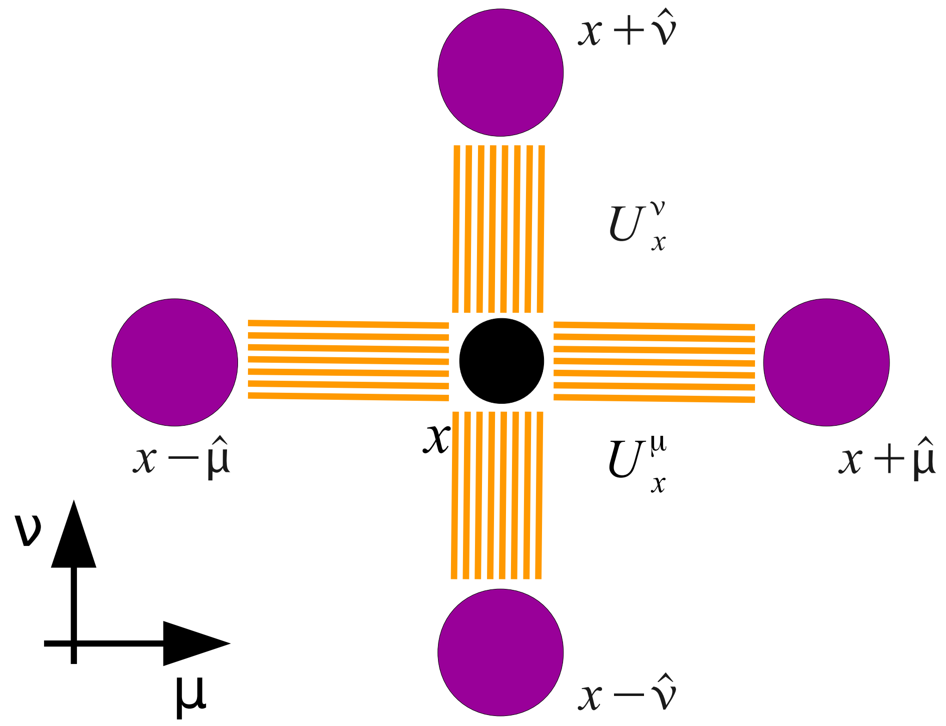
```
template<…> __global__ void fooGPU(Arg arg) {
  int tid = threadIdx.x + blockIdx.x*blockDim.x;
  real sum = bar<…>(arg, tid);
  __shared__ typename BlockReduce::TempStorage tmp;
  arg.sum = cub::BlockReduce<…>(tmp).Sum(sum);
}
```

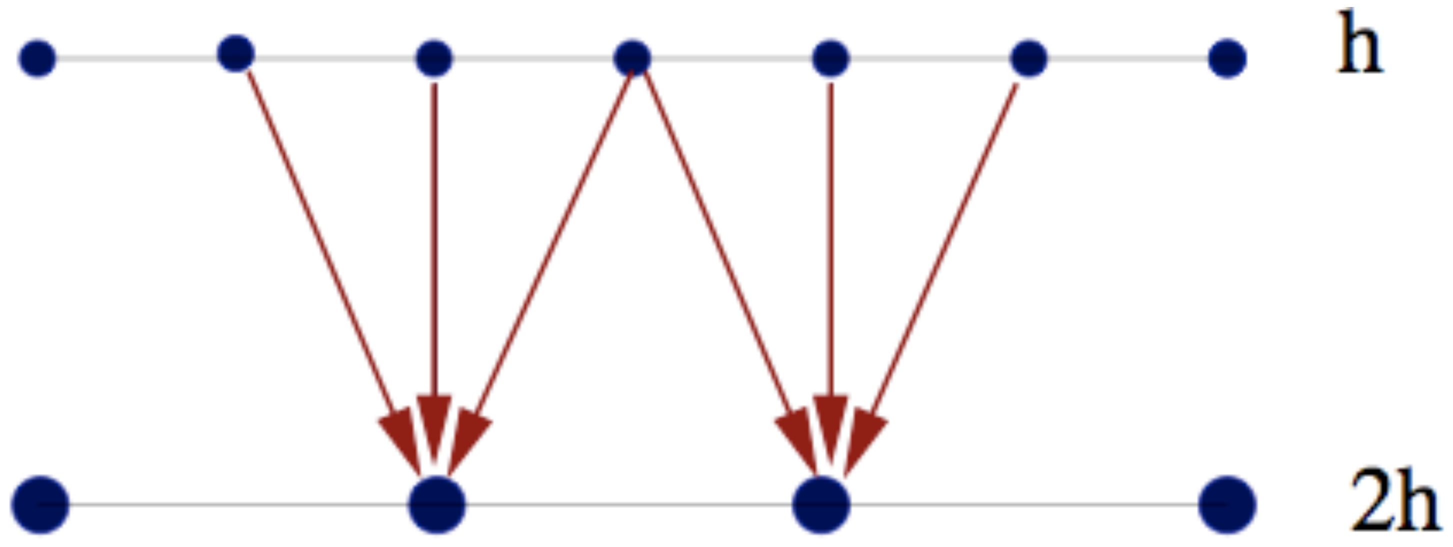
INGREDIENTS FOR PARALLEL ADAPTIVE MULTIGRID

- Prolongation construction (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

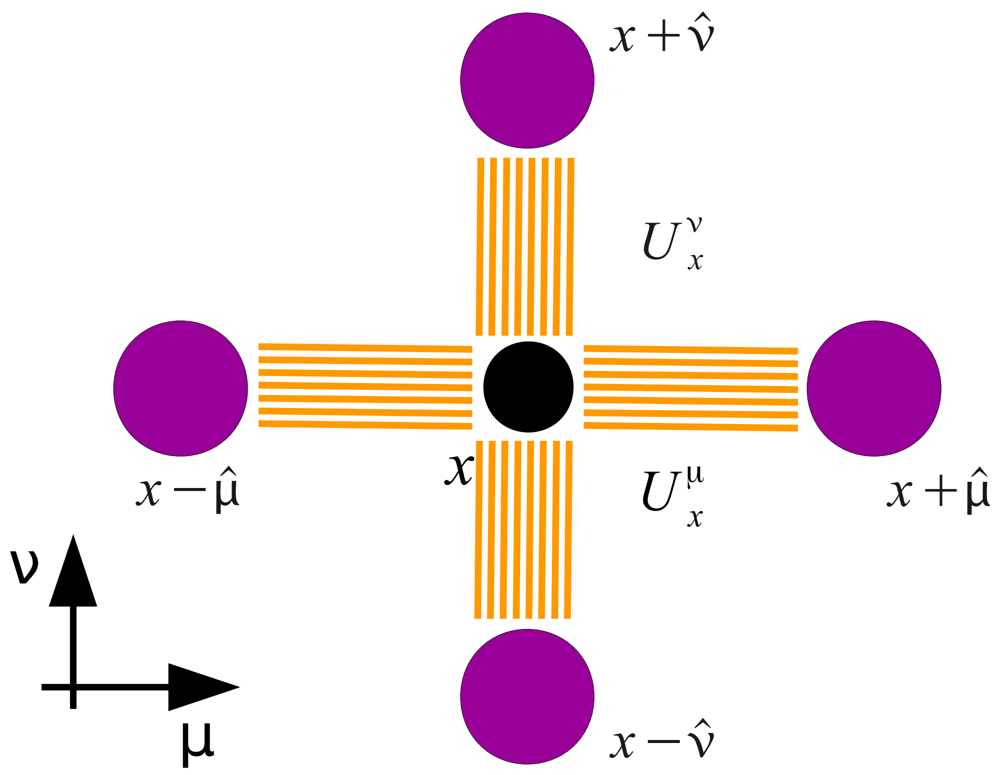








COARSE GRID OPERATOR



- 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

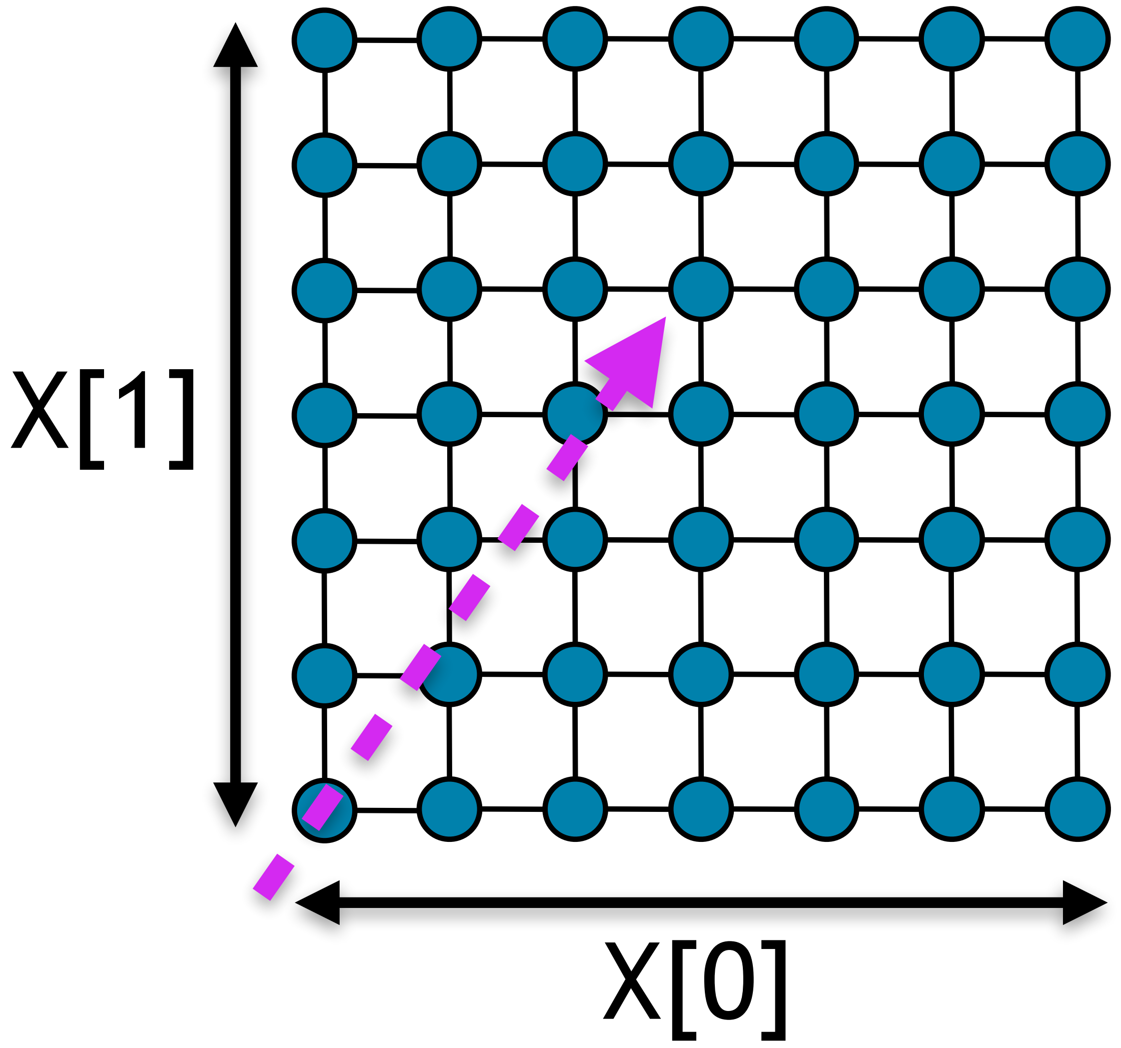
- Current GPUs have up to 3072 processing cores

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

GRID PARALLELISM

Thread x dimension maps to location on the grid

```
__device__ void grid_idx(int x[], const int X[])
{
  // X[] holds the local lattice dimension
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  int za = (idx / X[0]);
  int zb =  (za / X[1]);
  x[1] = za - zb * X[1];
  x[3] = (zb / X[2]);
  x[2] = zb - x[3] * X[2];
  x[0] = idx - za * X[0];
  // x[] now holds the thread coordinates
}
```



MATRIX-VECTOR PARALLELISM

Each stencil application is a sum of matrix-vector products

Parallelize over output vector indices (parallelization over color and spin)

Thread y dimension maps to vector indices

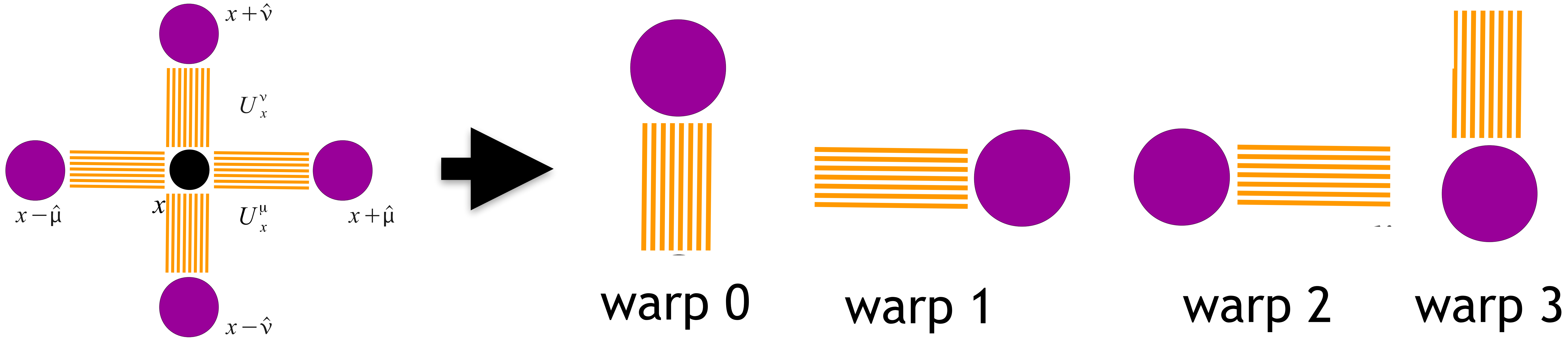
Up to 2 x N_v more parallelism

thread y index

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

```
template<int Nv>
__device__ void color_spin_idx(int &s, int &c)
{
  int yIdx = blockDim.y*blockIdx.y + threadIdx.y;
  int s = yIdx / Nv;
  int c = yIdx % Nv;
  // s is now spin index for this thread
  // c is now color index for this thread
}
```

STENCIL DIRECTION PARALLELSIM



Partition computation over stencil direction and dimension onto different threads

Write result to shared memory

Synchronize

dim=0/dir=0 threads combine and write out result

Introduces up to 8x more parallelism

```
__device__ void dim_dir_idx(int &dim, int &dir)
{
  int zIdx = blockDim.z*blockIdx.z + threadIdx.z;
  int dir = zIdx % 2;
  int dim = zIdx / 2;
  // dir is now the fwd/back direction for this thread
  // dim is now the dim for this thread
}
```

DOT PRODUCT PARALLELIZATION I

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

Partition dot product between threads in the same warp

Use warp shuffle for final result

Useful when not enough grid parallelism to fill a warp

```
const int warp_size = 32; // warp size
const int n_split = 4; // four-way warp split
const int grid_points = warp_size/n_split; // grid points per warp
complex<real> sum = 0.0;
for (int i=0; i<N; i+=n_split)
  sum += a[i] * b[i];

// cascading reduction
for (int offset = warp_size/2; offset >= grid_points; offset /= 2)
  sum += __shfl_down(sum, offset);
// first grid_points threads now hold desired result
```

DOT PRODUCT PARALLELIZATION II

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

Partition dot product computation within a thread

Hide dependent arithmetic latency within a thread

More important for Kepler then Maxwell / Pascal

```
const int n_ilp = 2; // two-way ILP
complex<real> sum[n_ilp] = { };
for (int i=0; i<N; i+=n_ilp)
  for (int j=0; j<n_ilp; j++)
    sum[j] += a[i+j] * b[i+j];

complex<real> total = static_cast<real>(0.0);
for (int j=0; j<n_ilp; j++) total += sum[j];
```

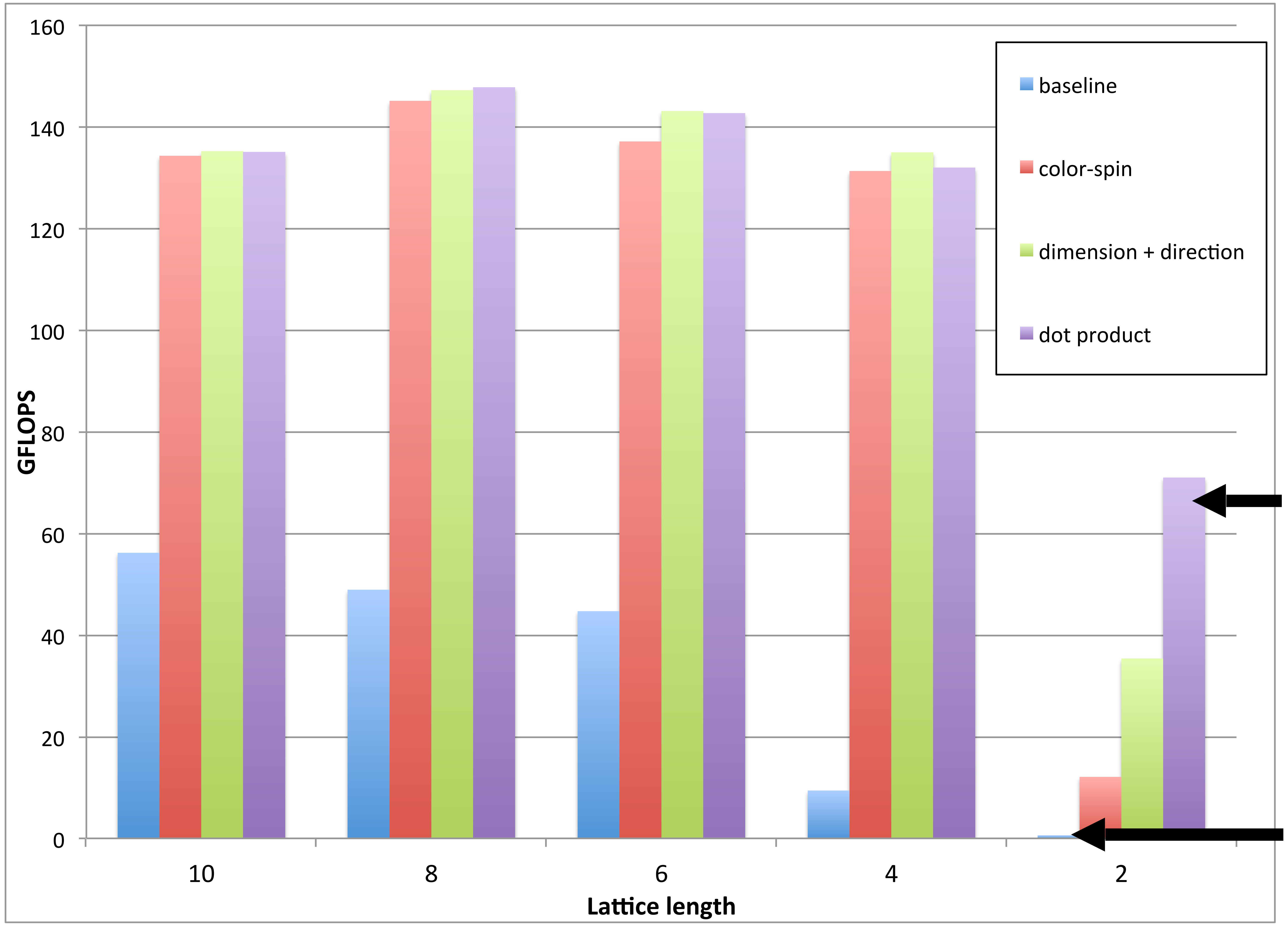
Degree of ILP exposed

Multiple computations with no dependencies

Compute final result

COARSE GRID OPERATOR PERFORMANCE

Tesla K20X (Titan), FP32, N = 24

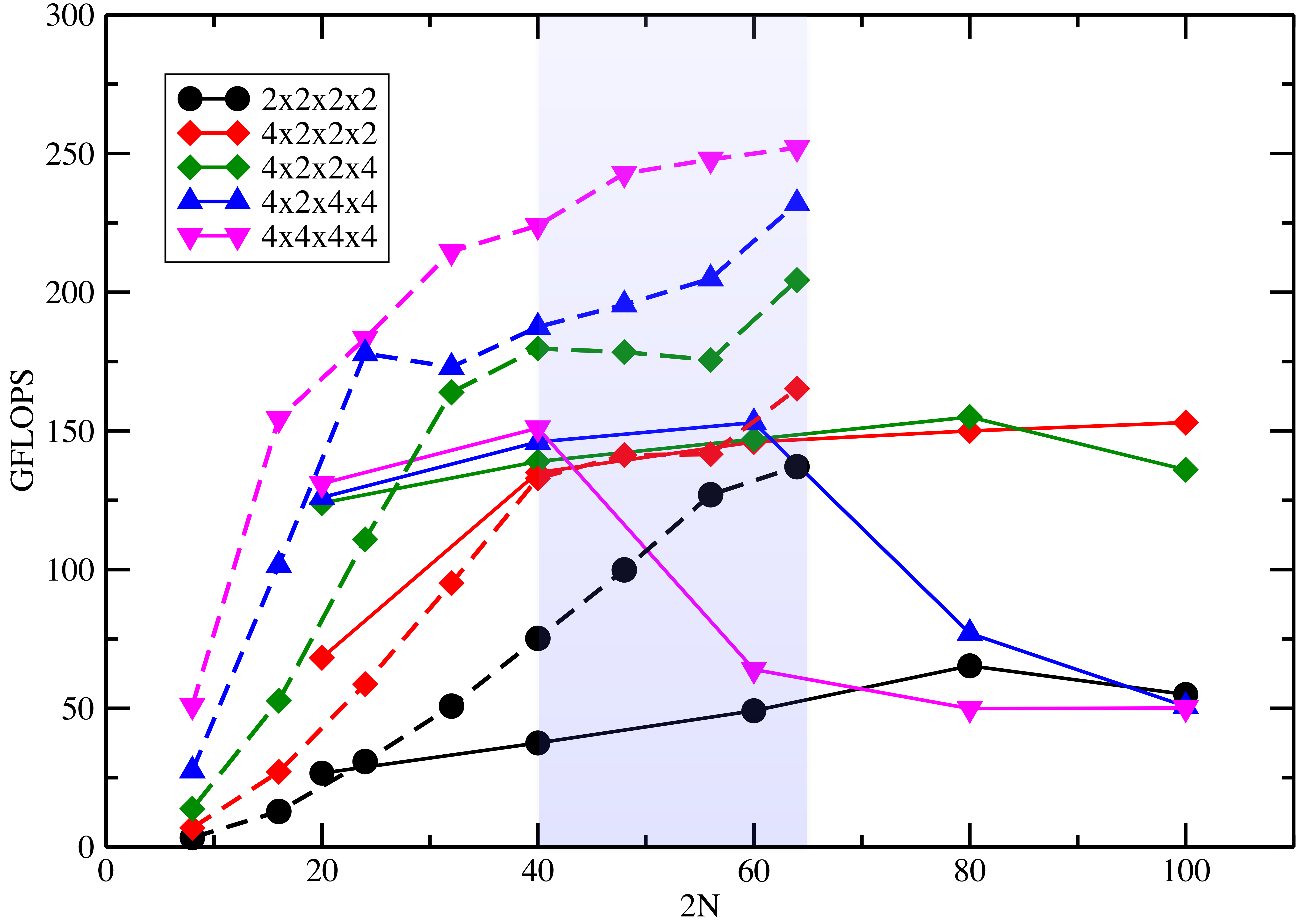


24,576-way parallel

16-way parallel

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

RESULTS

MULTIGRID VERSUS BICGSTAB

Compare MG against the best traditional clover Krylov solver

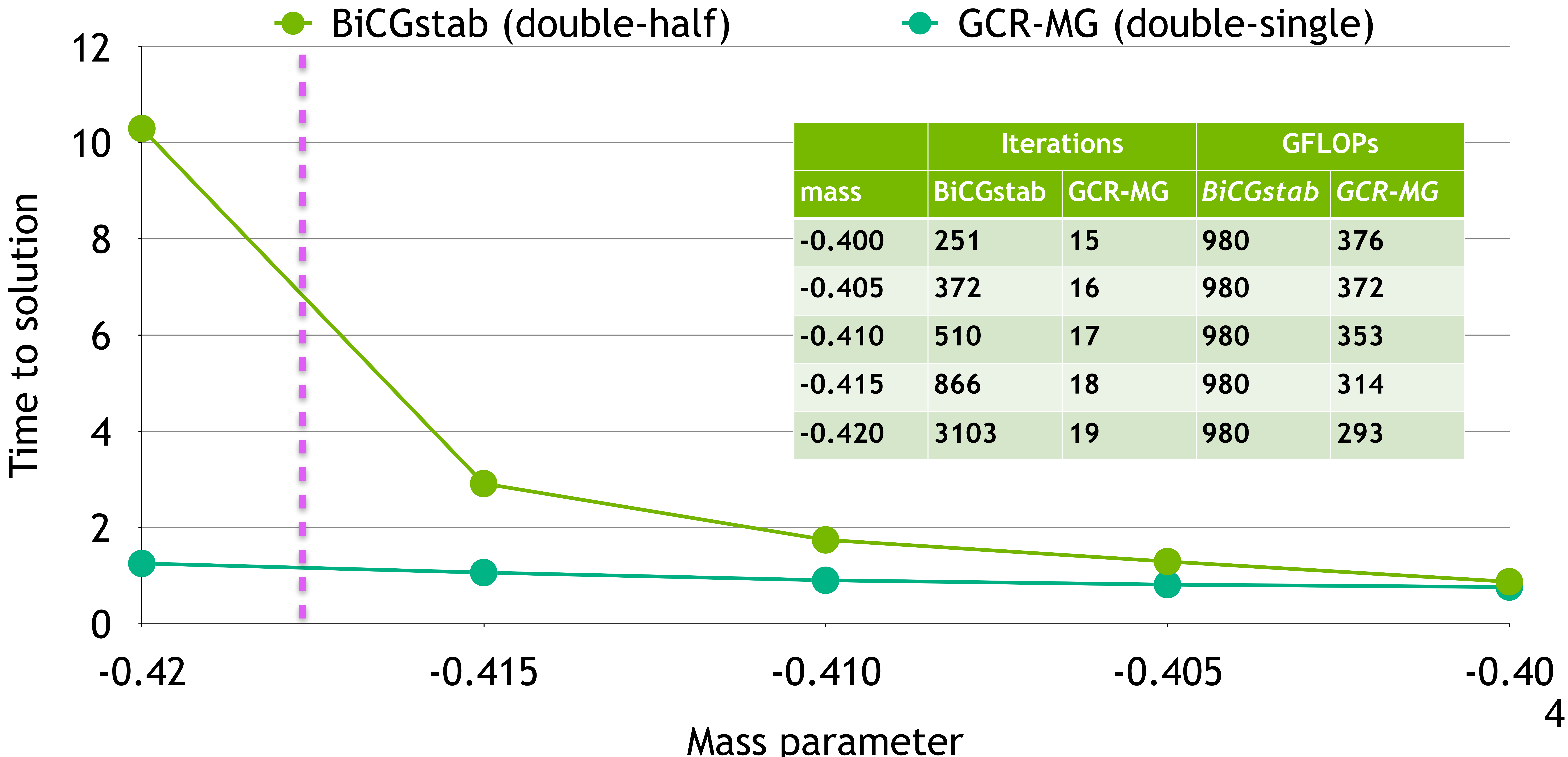
- BiCGstab in double/half precision
- 12/8 reconstruct
- Red-black preconditioning

Adaptive Multigrid algorithm

- GCR outer solver wraps 3-level MG preconditioner
- GCR restarts done in double, everything else in single
- 24 or 32 null-space vectors on fine grid
- Minimum Residual smoother
- Red-black preconditioning on each level
- GCR coarse-grid solver

MULTIGRID VERSUS BICGSTAB

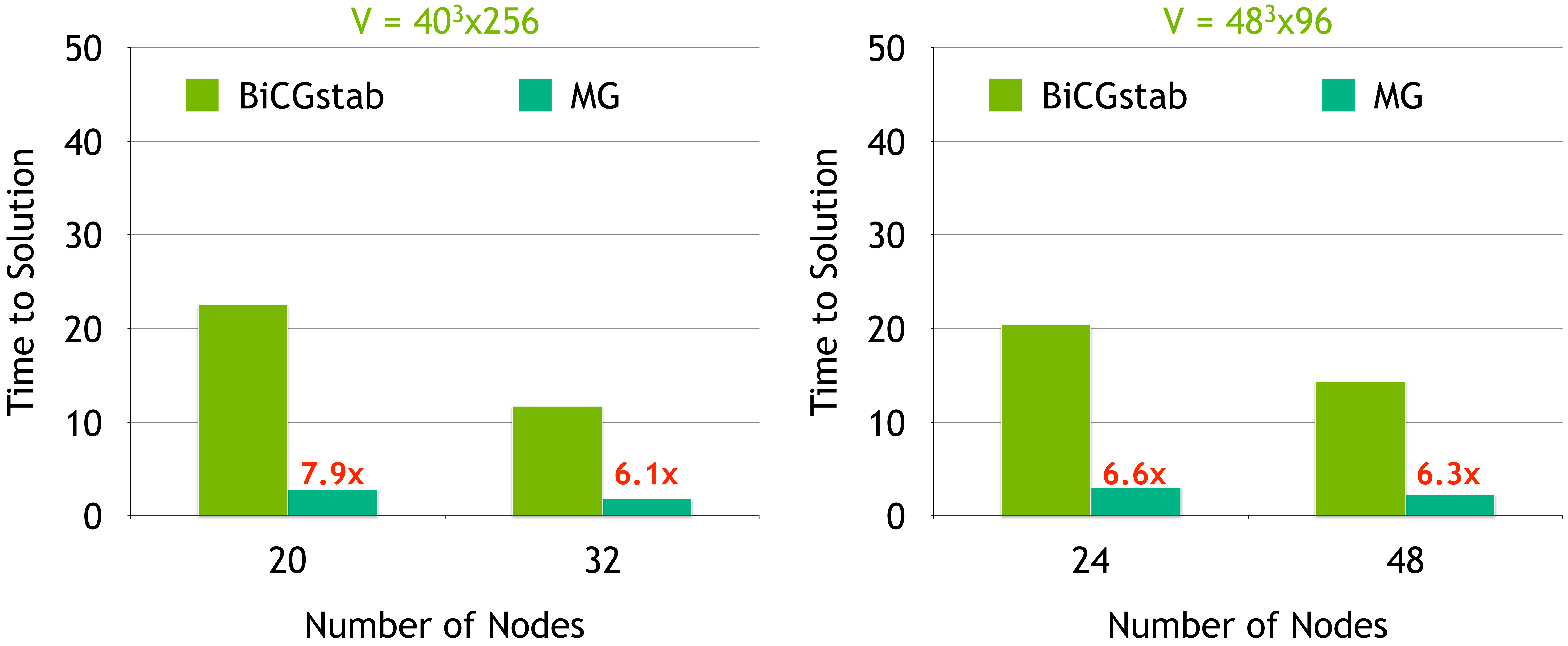
V = 24^3x64, single workstation (3x M6000)



	Iterations		GFLOPs	
mass	BiCGstab	GCR-MG	*BiCGstab*	*GCR-MG*
-0.400	251	15	980	376
-0.405	372	16	980	372
-0.410	510	17	980	353
-0.415	866	18	980	314
-0.420	3103	19	980	293

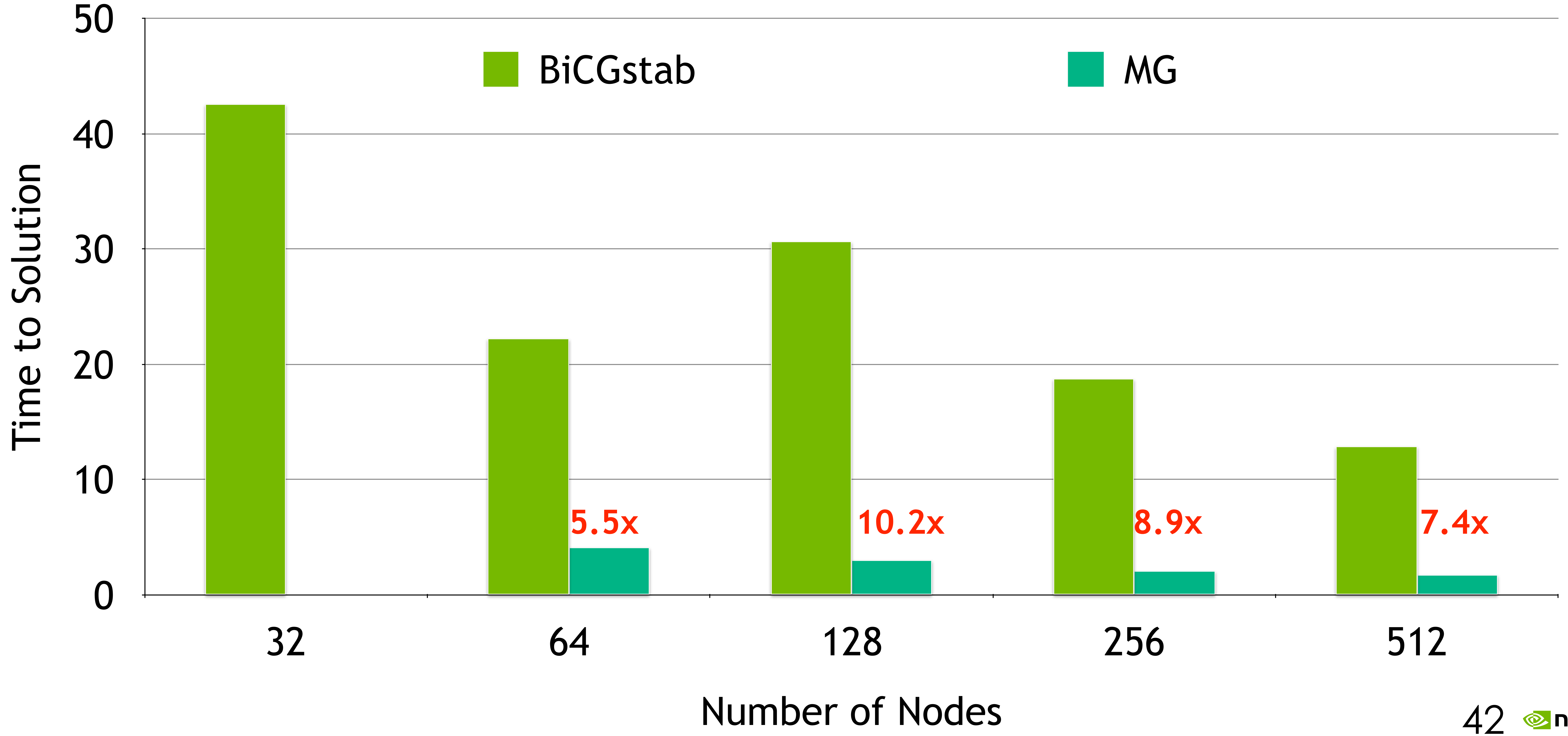
MULTIGRID VERSUS BICGSTAB

Strong scaling on Titan (K20X)



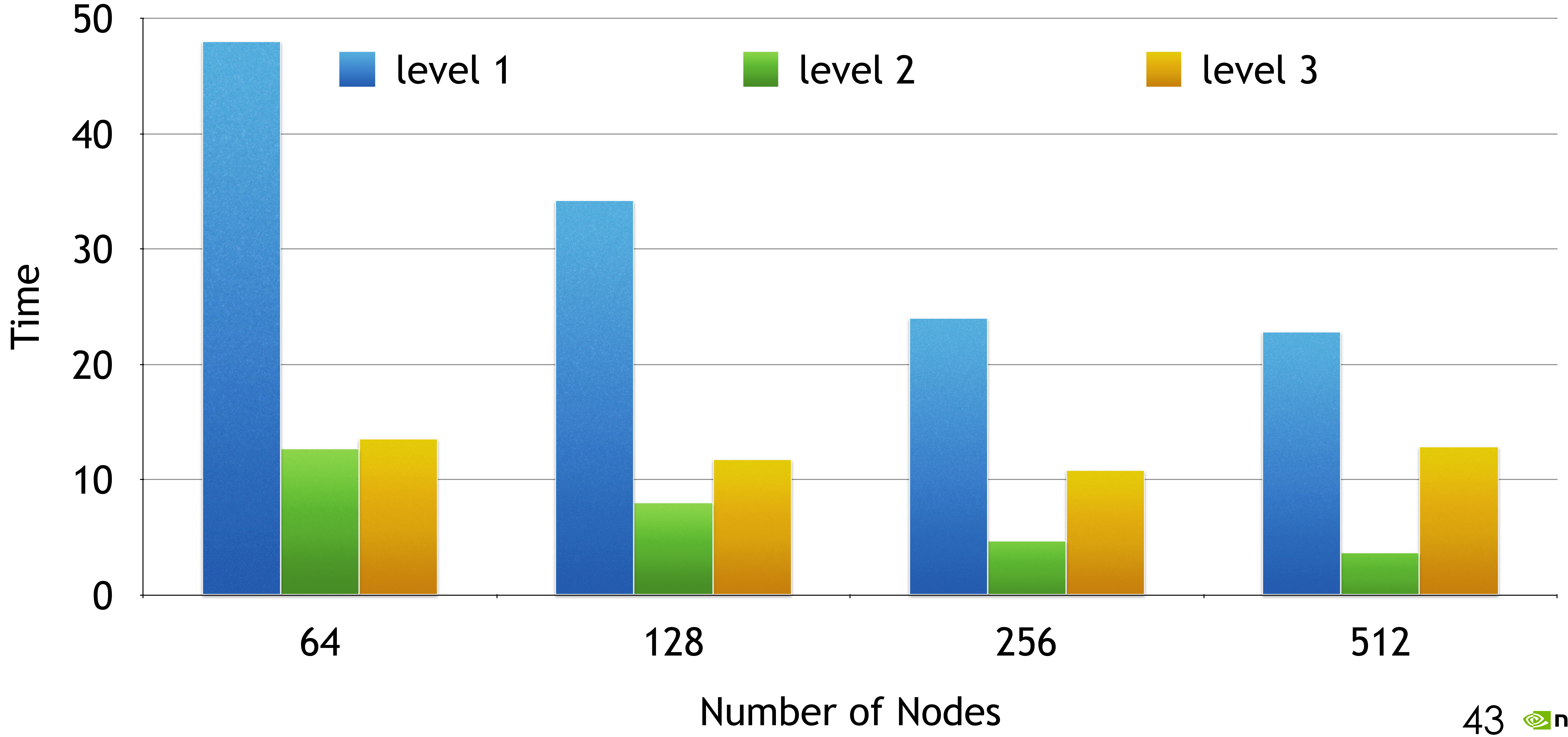
MULTIGRID VERSUS BICGSTAB

Strong scaling on Titan (K20X), V = 64^3x128

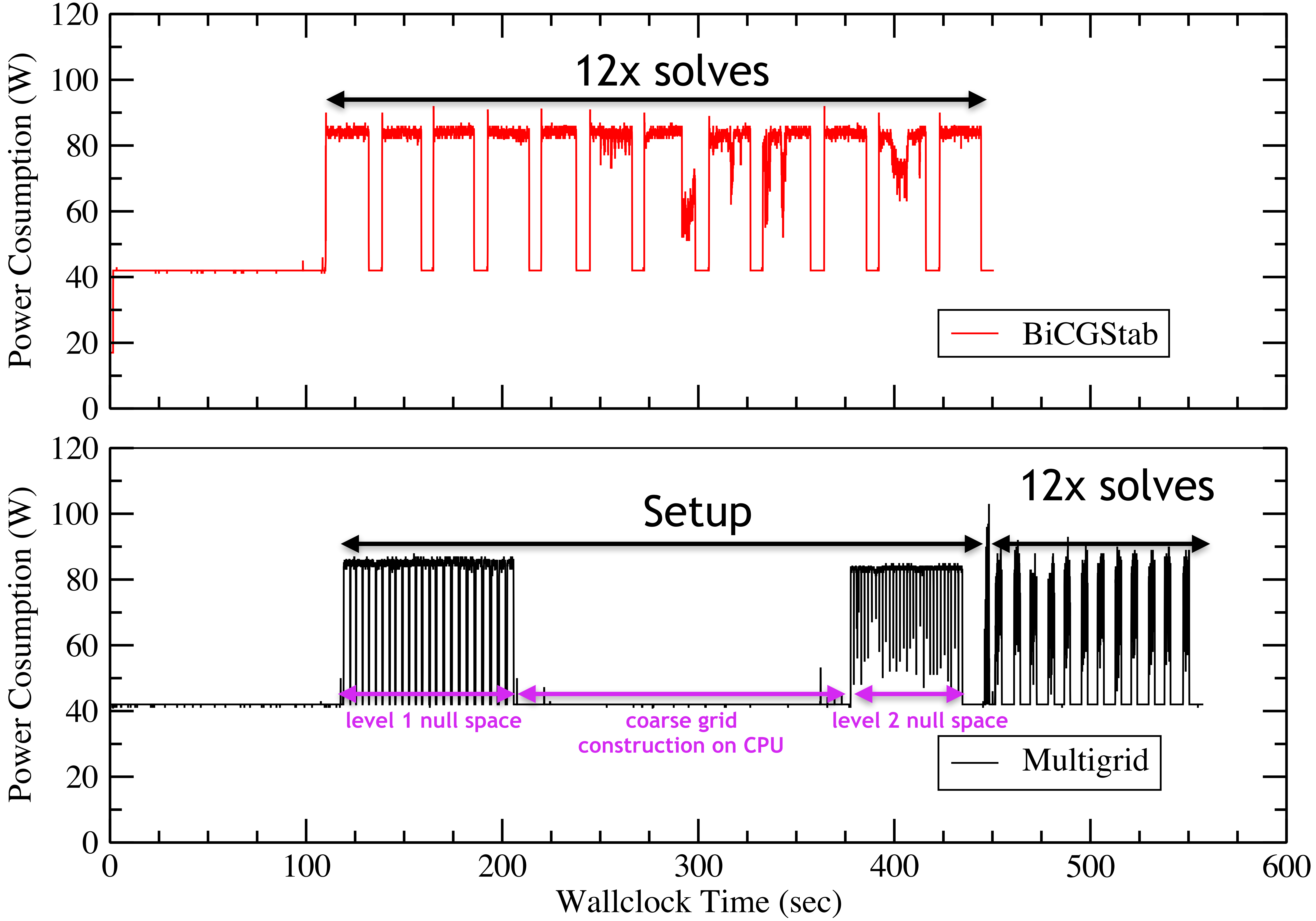


MULTIGRID TIMING BREAKDOWN

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



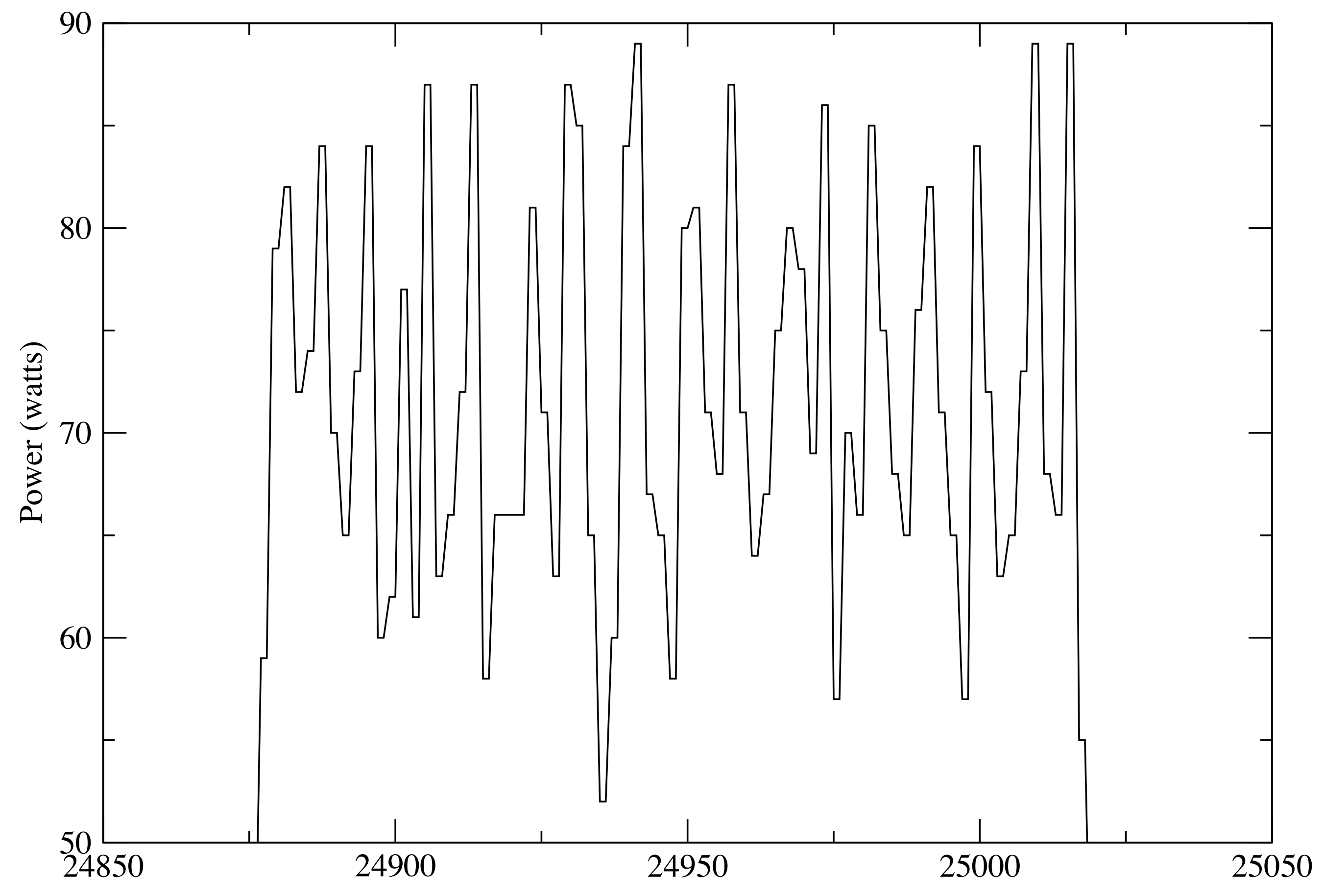
POWER EFFICIENCY



BiCGstab average power
~ 83 watts per GPU

MG average power
~ 72 watts per GPU

MG consumes less power and 10x faster



MULTIGRID FUTURE WORK

Absolute Performance tuning, e.g., half precision on coarse grids

Strong scaling improvements:

- Combine with Schwarz preconditioner
- Accelerate coarse grid solver: CA-GMRES instead of GCR
- More flexible coarse grid distribution, e.g., redundant nodes

Investigate off load of coarse grids to the CPU

- Use CPU and GPU simultaneously using additive MG

Full off load of setup phase to GPU

CONCLUSIONS AND OUTLOOK

Multigrid algorithms LQCD are running well on GPUs

- Up to 10x speedup

Fine-grained parallelization was key

- Importance of fine-grained parallelization will only increase
- Fine-grained parallelism applicable to all geometric stencil-type problems

Future consider heterogeneous multigrid

GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

THANK YOU

JOIN THE CONVERSATION

#GTC16

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join

PRESENTED BY

NVIDIA.