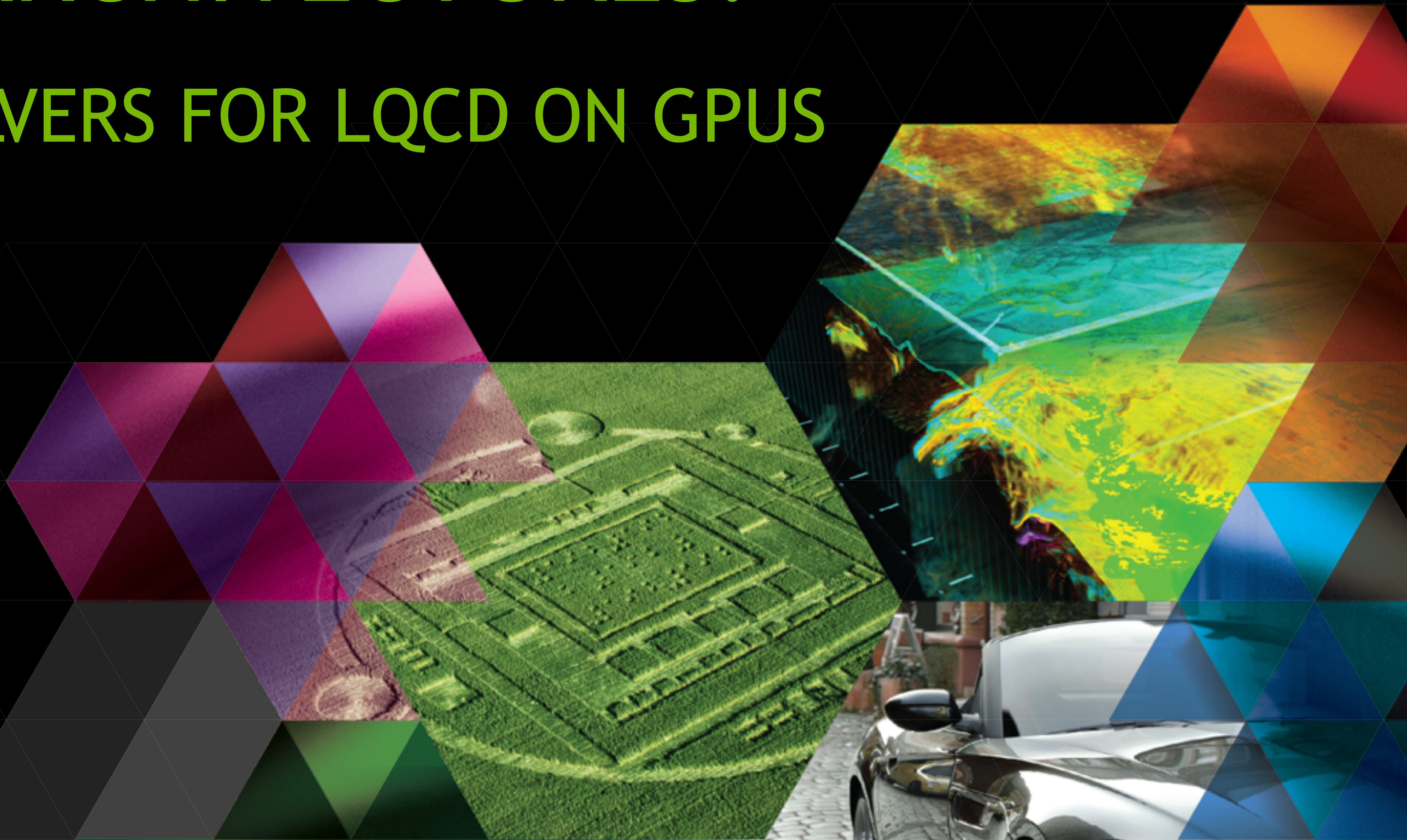
GPU TECHNOLOGY CONFERENCE

HIERARCHICAL ALGORITHMS ON HETEROGENEOUS ARCHITECTURES:

ADAPTIVE MULTIGRID SOLVERS FOR LQCD ON GPUS

M Clark
NVIDIA



Contents

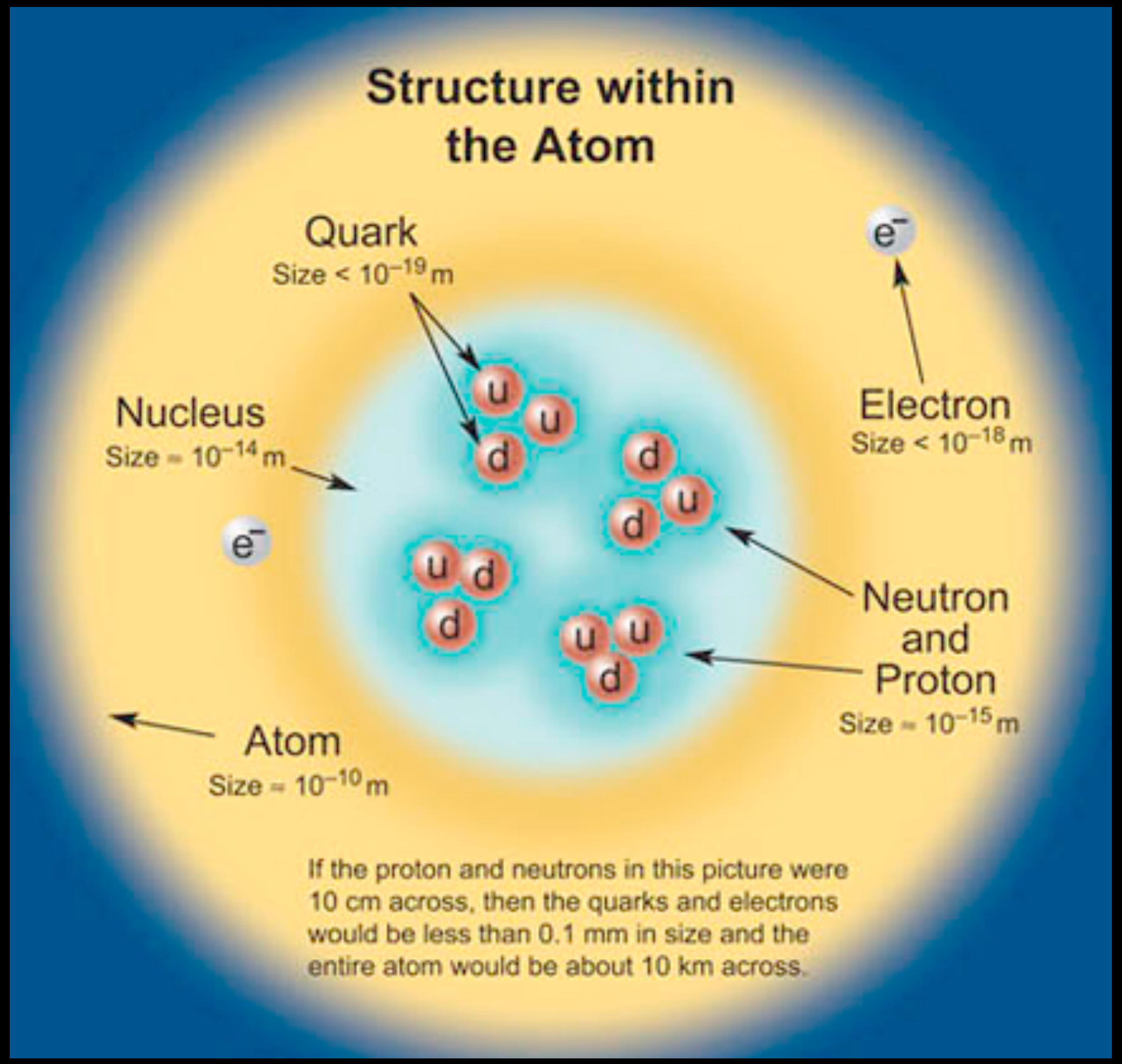
- Introduction to Lattice QCD
- Introduction to GPUs
- QUDA Library
- Multigrid on Heterogeneous Architectures
- Summary

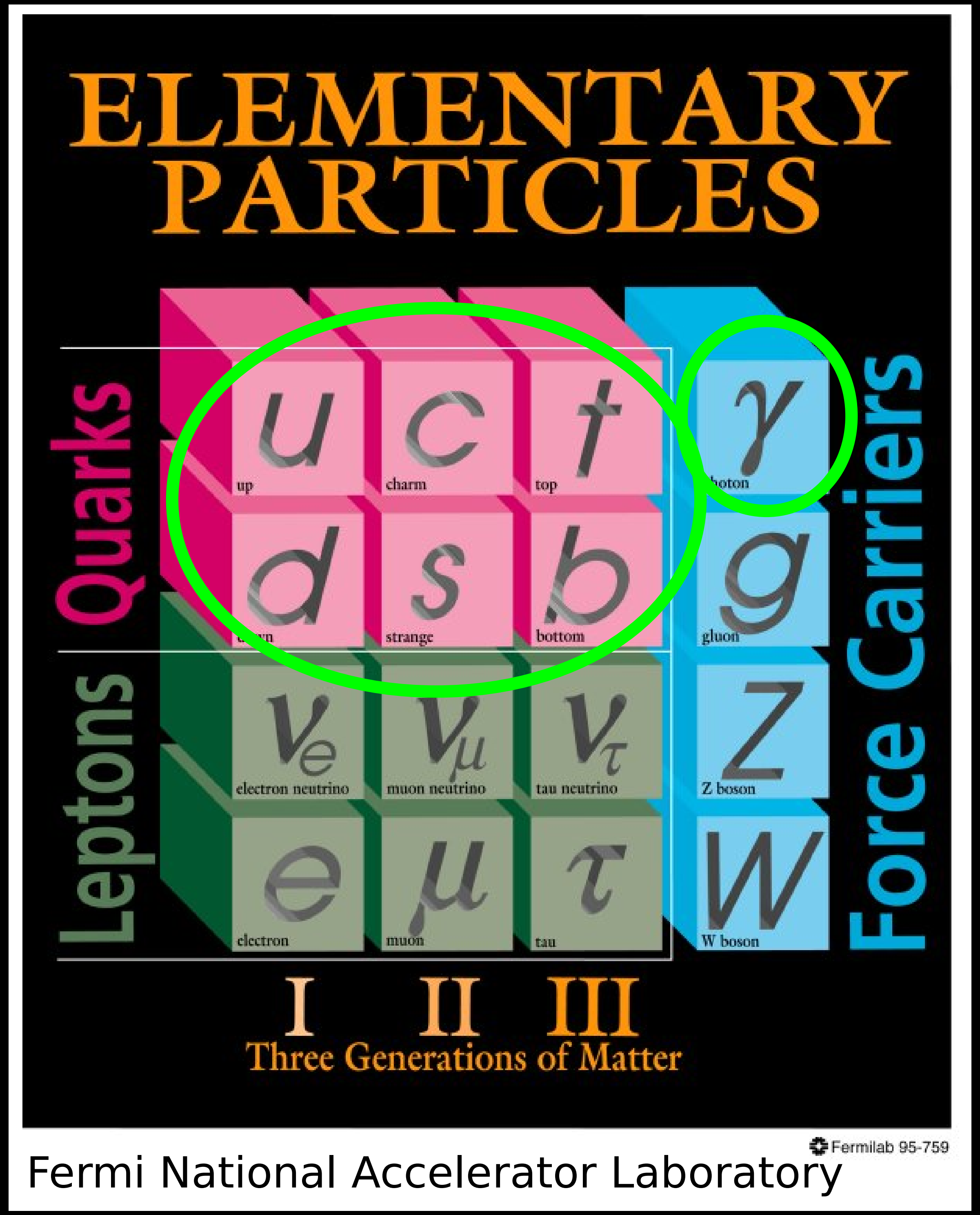
Quantum Chromodynamics

- 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, lots of equations etc.

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

…but


Structure within the Atom
Quark
Size < 10^-19 m
Electron
Size < 10^-18 m
Nucleus
Size ≈ 10^-14 m
Neutron and Proton
Size ≈ 10^-15 m
Atom
Size ≈ 10^-10 m
u
d
e⁻
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.

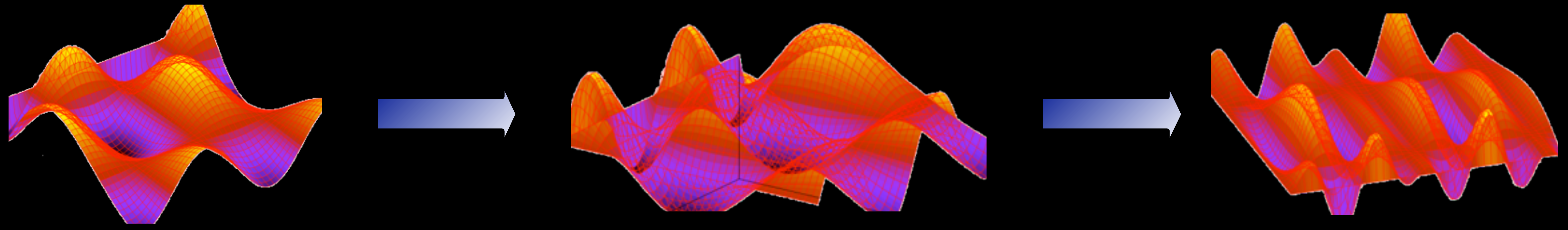


Lattice Quantum Chromodynamics

- 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 North American (public) supercomputer cycles

Steps in a lattice QCD 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

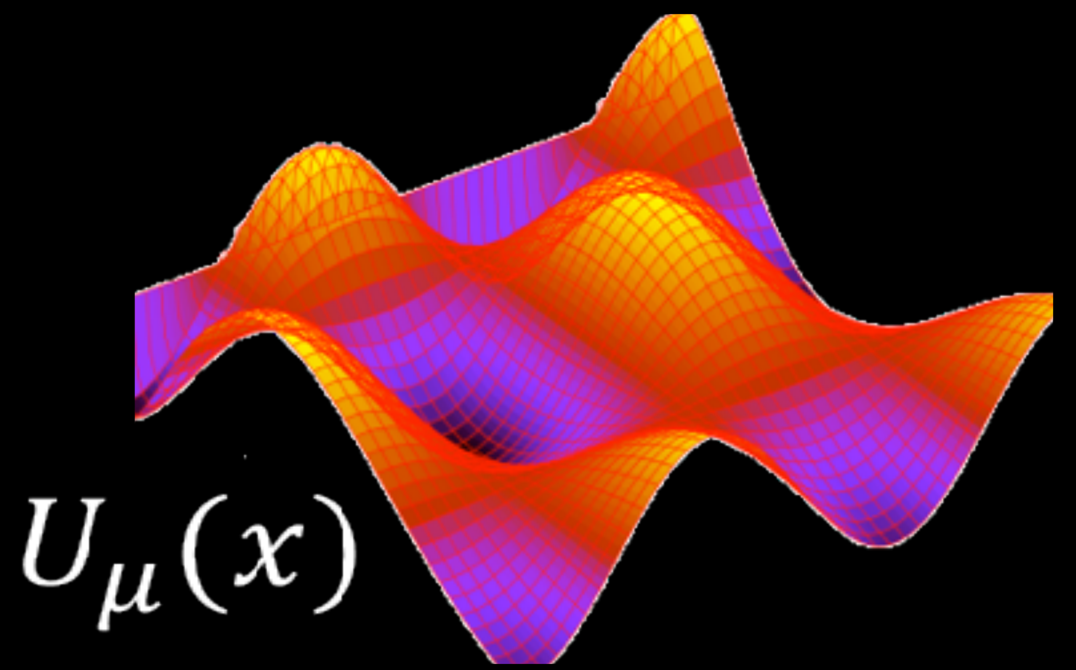


$$D_{ij}^{\alpha\beta}(x,y;U)\psi_j^{\beta}(y) = \eta_i^{\alpha}(x)$$

or “$Ax = b$”

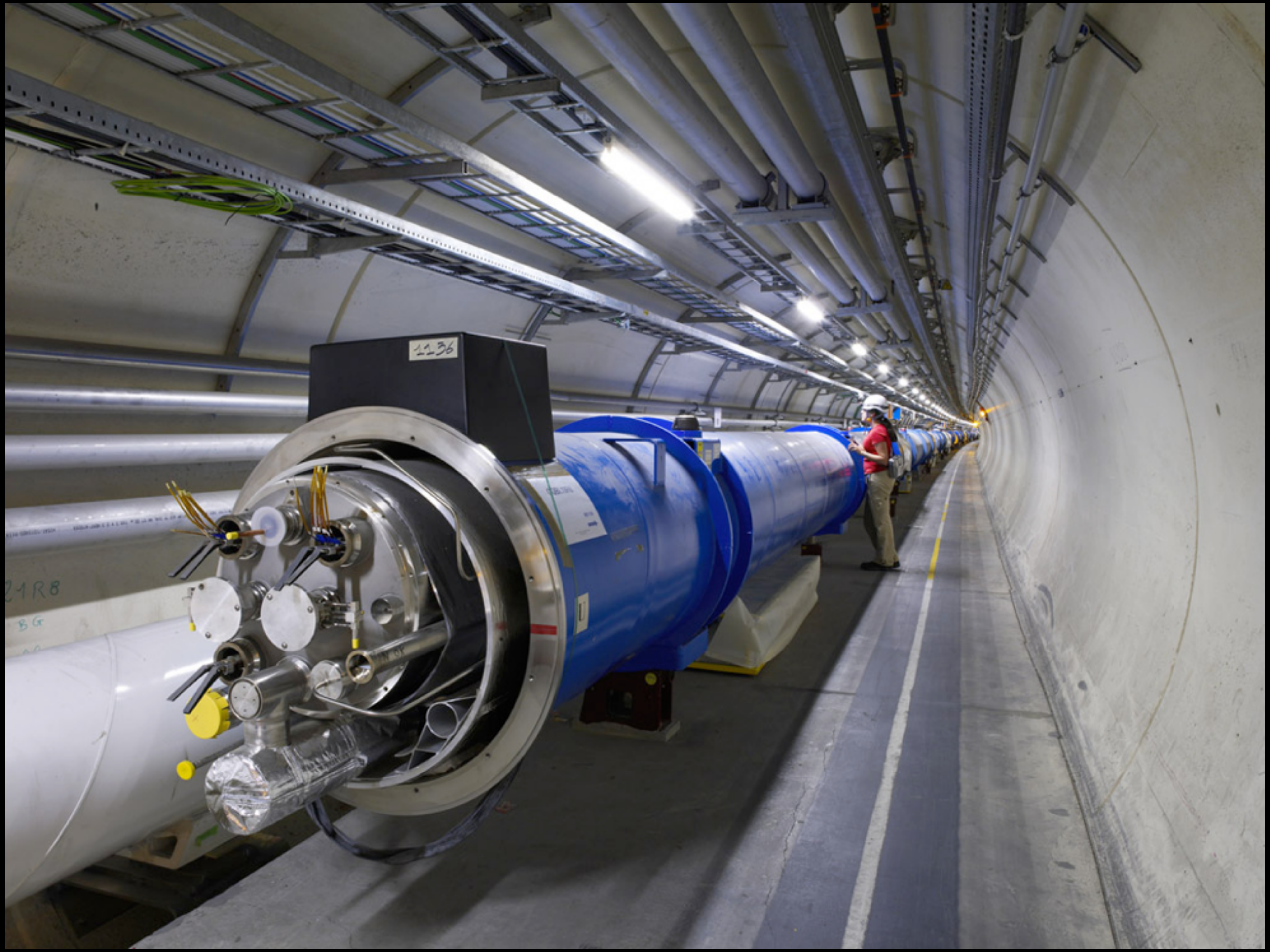
2. “Analyze” the configurations
   - Can be farmed out, assuming O(1 Tflops) per job.
   - 80-99% of the runtime is in the linear solver

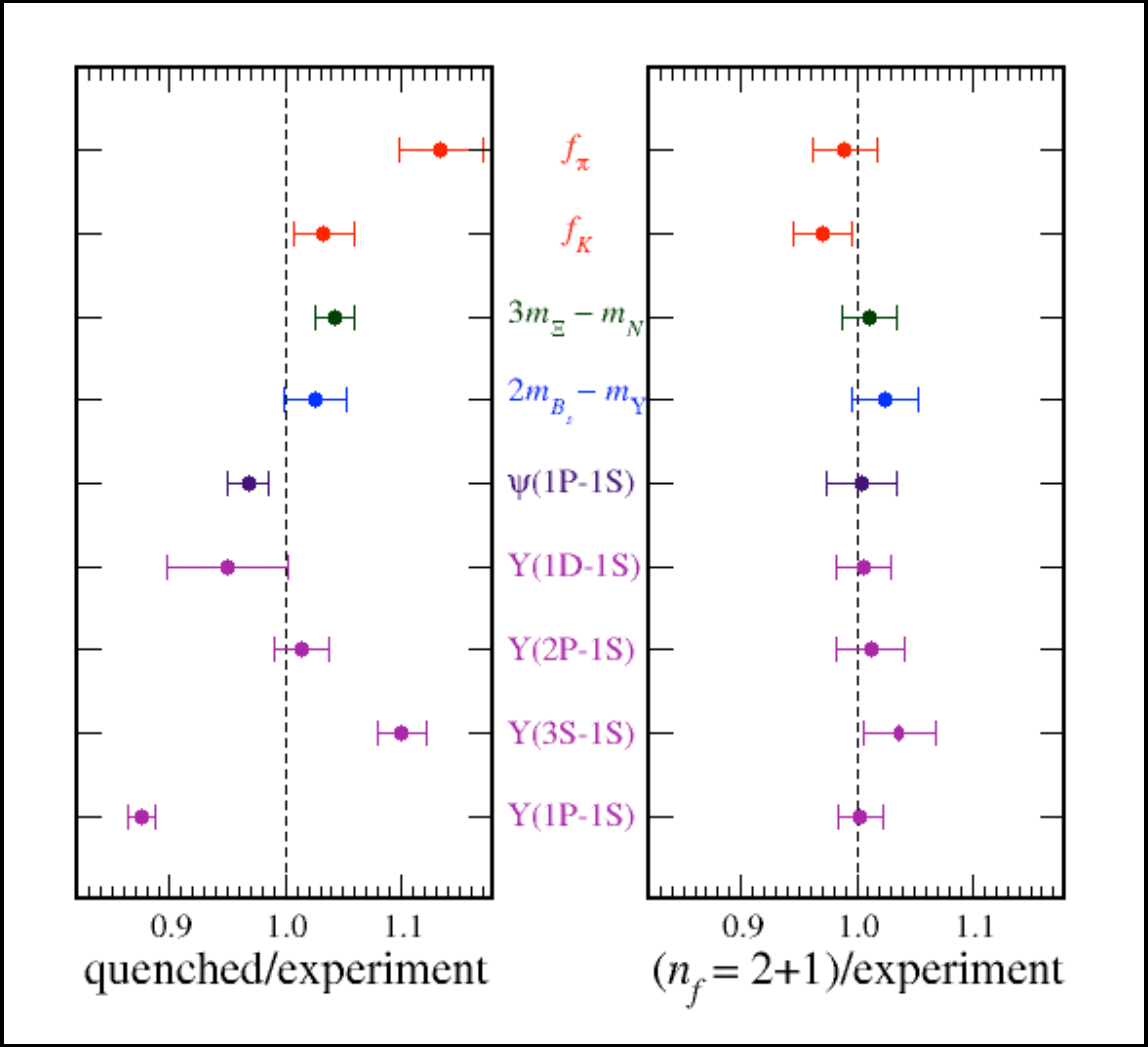
     Task parallelism means that clusters reign supreme here



D. Weintroub

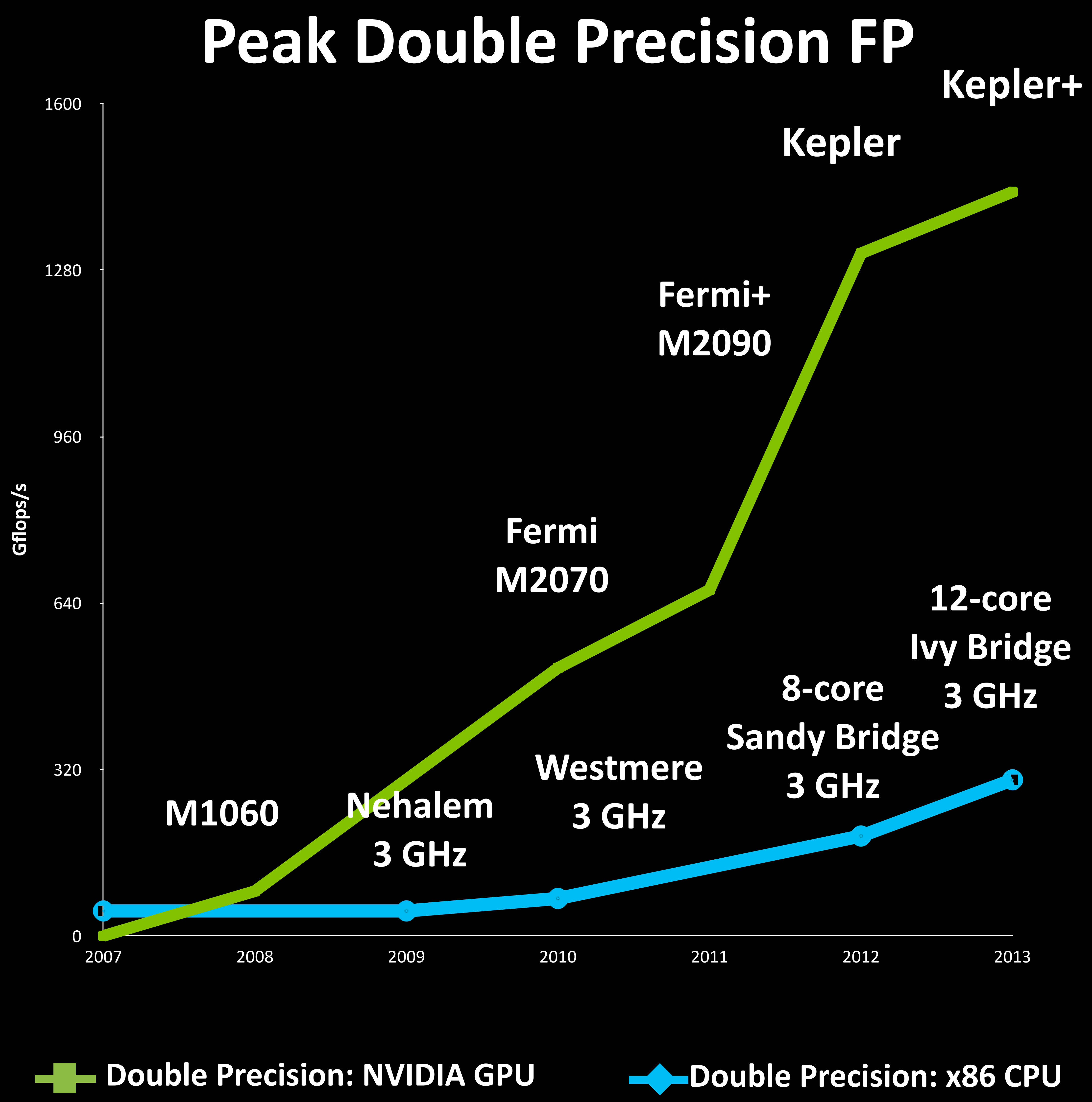
D. Weintroub

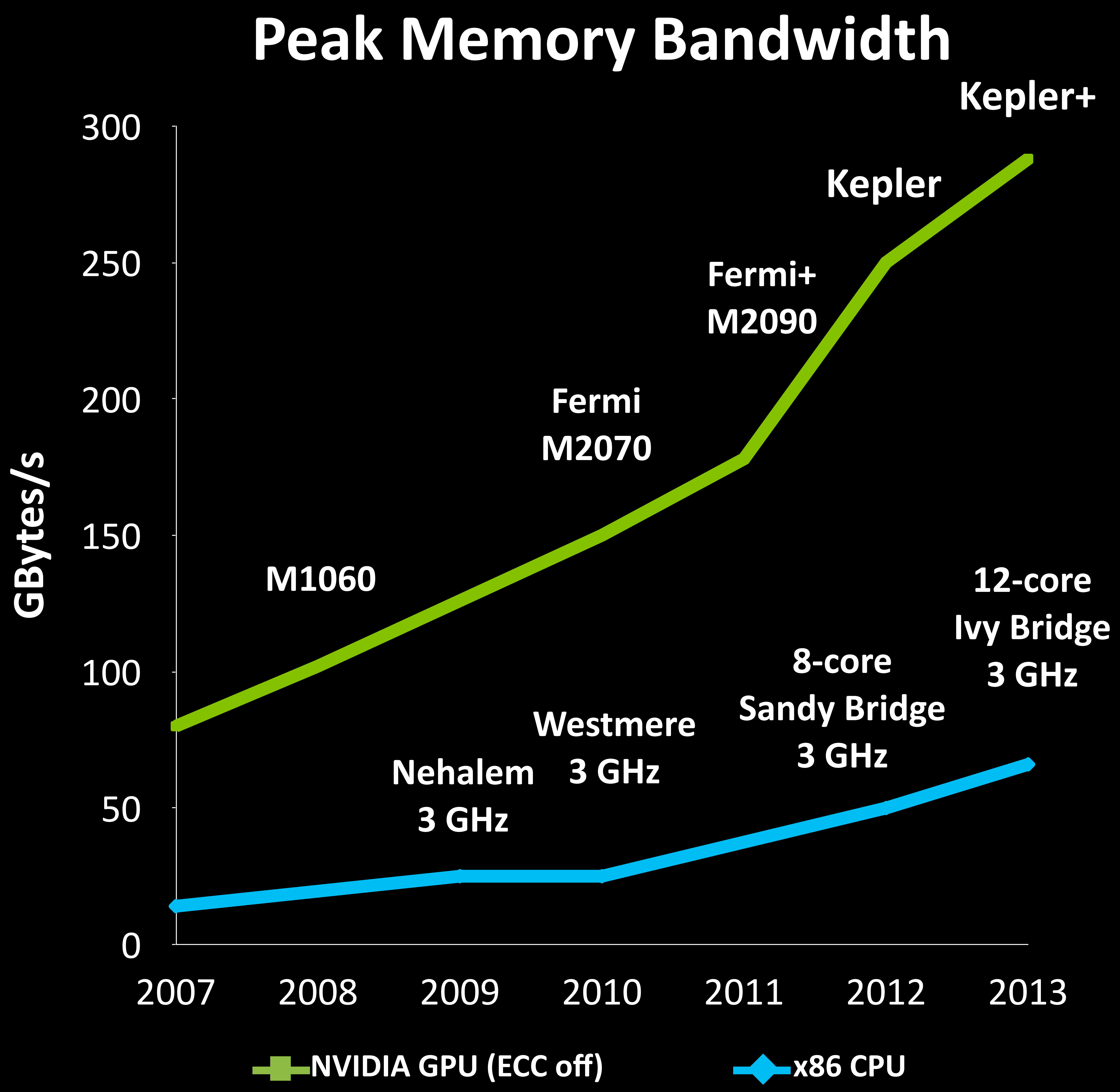



f_π
f_K
$3m_\Xi - m_N$
$2m_{B_s} - m_\Upsilon$
ψ(1P-1S)
Υ(1D-1S)
Υ(2P-1S)
Υ(3S-1S)
Υ(1P-1S)
0.9
1.0
1.1
quenched/experiment
0.9
1.0
1.1
$(n_f = 2+1)$/experiment

Davies *et al*

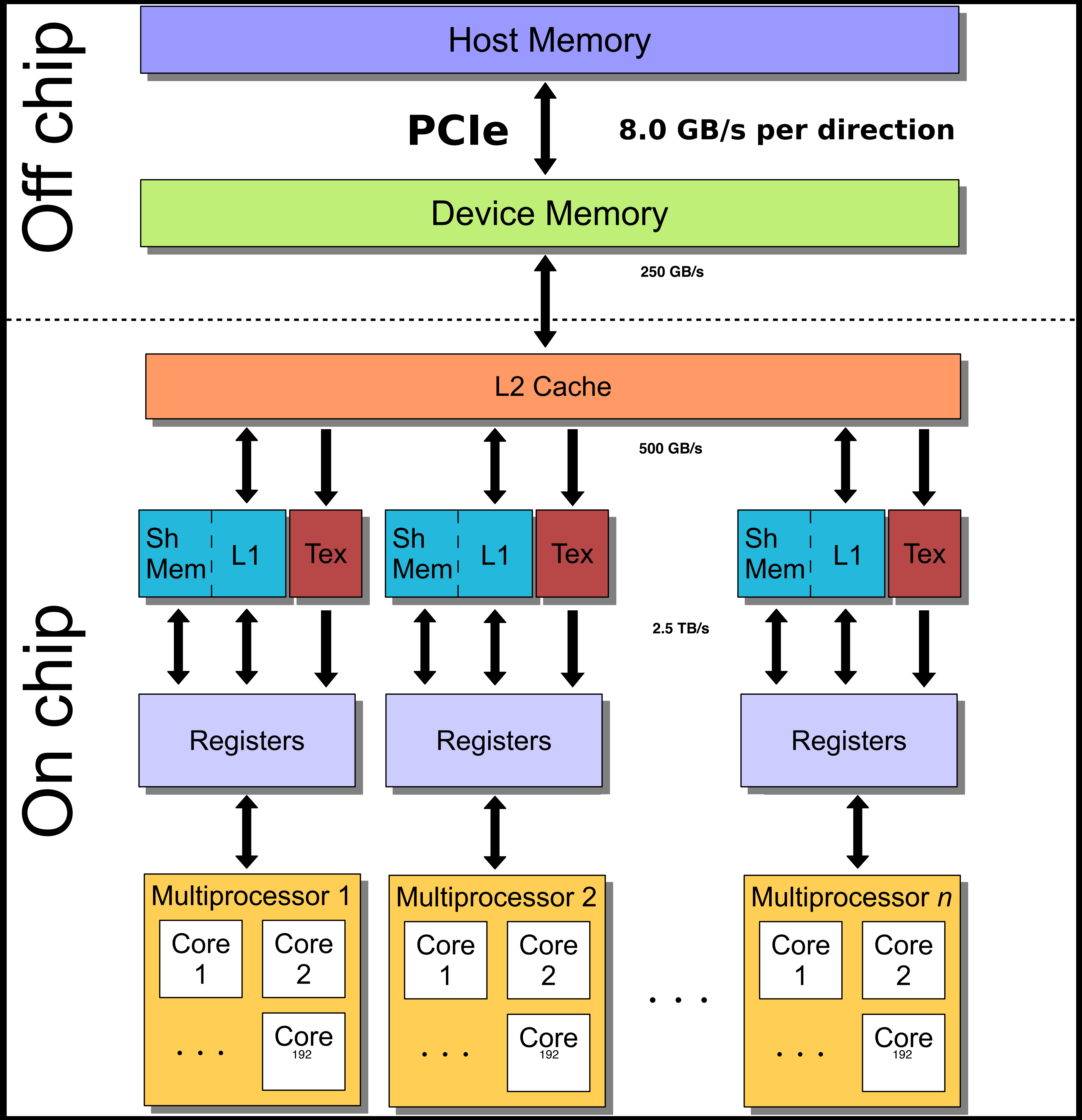
The March of GPUs





What is a GPU?

- Kepler K20X (2012)
  - 2688 processing cores
  - 3995 SP Gflops peak
- Effective SIMD width of 32 threads (warp)
- Deep memory hierarchy
- As we move away from registers
  - Bandwidth decreases
  - Latency increases
- Programmed using a thread model
  - Architecture abstraction is known as **CUDA**
  - Fine-grained parallelism required
- Diversity of programming languages
  - CUDA C/C++/Fortran
  - OpenACC, OpenMP 4.0
  - Python, etc.



TITAN

LQCD applications

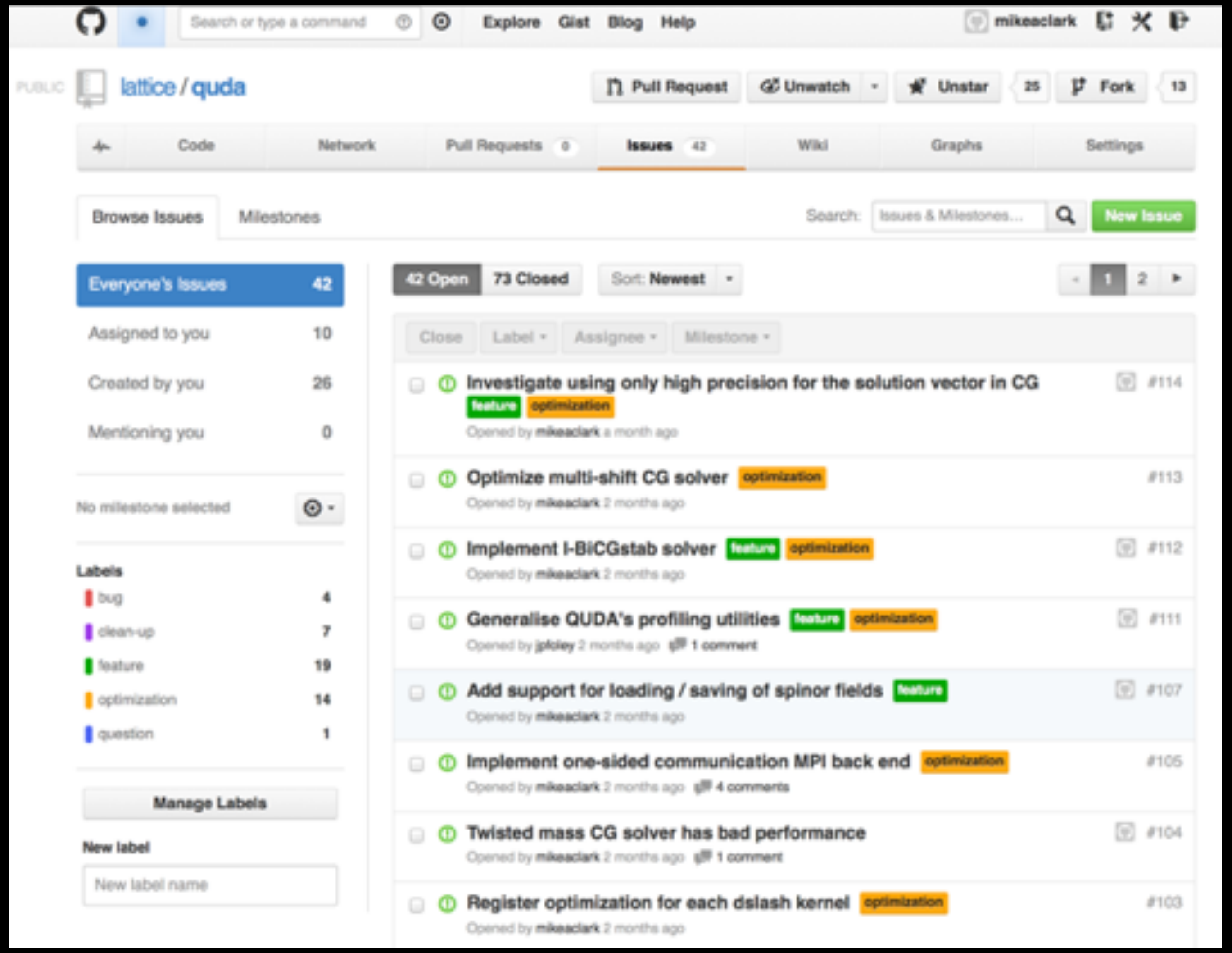
- Some examples
  - MILC (FNAL, Indiana, Arizona, Utah)
    - strict C, MPI only
  - CPS (Columbia, BNL, Edinburgh)
    - C++ (but no templates), MPI and partially threaded
  - Chroma (Jlab, Edinburgh)
    - C++ expression-template programming, MPI and threads
  - BQCD (Berlin QCD)
    - F90, MPI and threads
- Each application consists of 100K-1M lines of code
- Porting each application not directly tractable

Enter QUDA

- "QCD on CUDA" – http://lattice.github.com/quda
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, etc.
- Provides:
  - Various solvers for all major fermonic discretizations, with multi-GPU support
  - Additional performance-critical routines needed for gauge-field generation
- Maximize performance / Minimize time to science
  - 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
  - Multigrid solvers for optimal convergence new!

QUDA is community driven

- Ron Babich (NVIDIA)
- Kip Barros (LANL)
- Rich Brower (Boston University)
- Michael Cheng (Boston University)
- MAC (NVIDIA)
- Justin Foley
- Joel Giedt (Rensselaer Polytechnic Institute)
- Steve Gottlieb (Indiana University)
- Bálint Joó (Jlab)
- Hyung-Jin Kim (BNL)
- Jian Liang (IHEP)
- Claudio Rebbi (Boston University)
- Guochun Shi (NCSA -> Google)
- Alexei Strelchenko (Cyprus Institute -> FNAL)
- Alejandro Vaquero (Cyprus Institute)
- Frank Winter (UoE -> Jlab)
- Yibo Yang (IHEP)



Everyone's Issues	42
Assigned to you	10
Created by you	26
Mentioning you	0

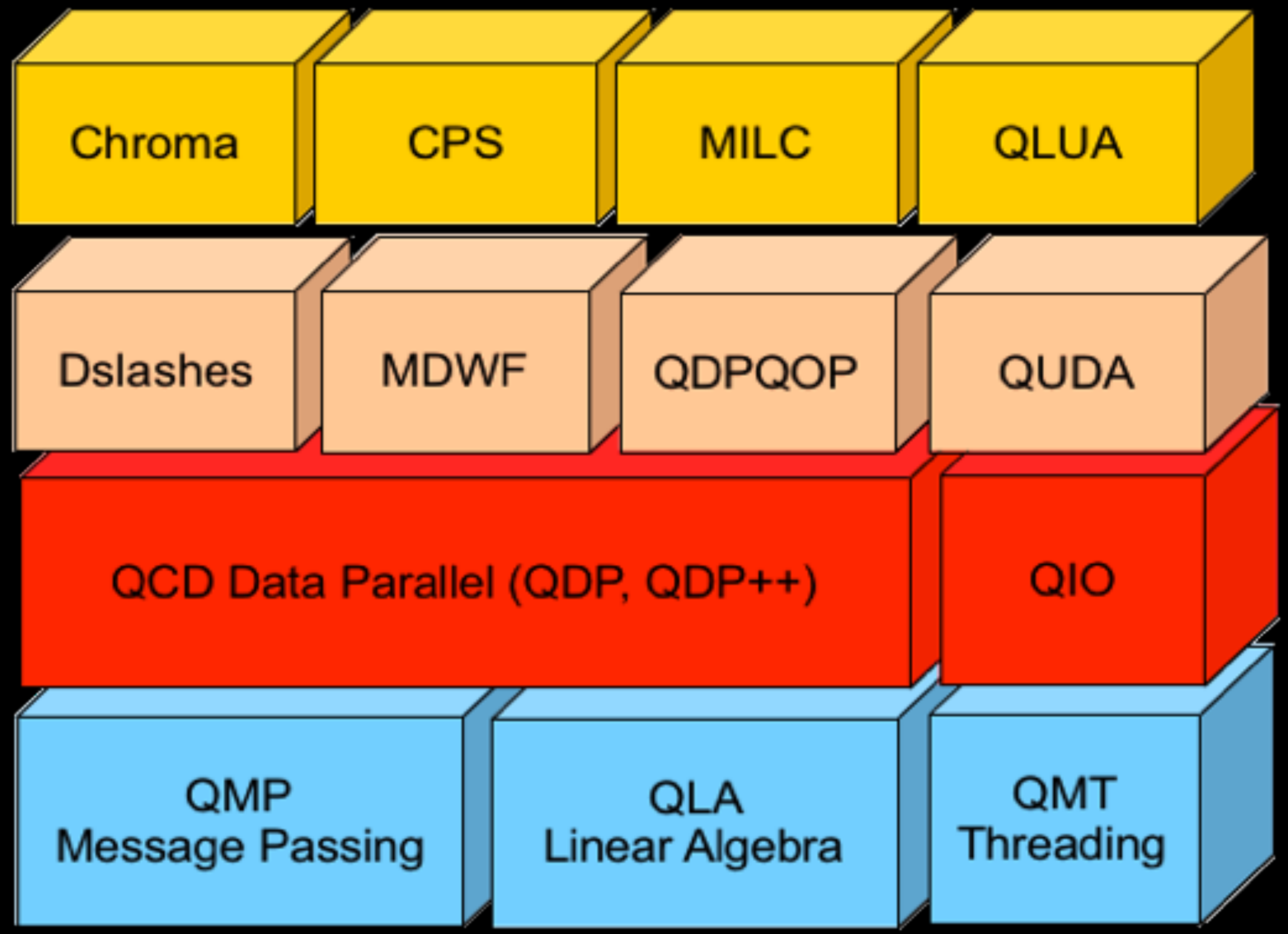
No milestone selected

Labels

bug	4
clean-up	7
feature	19
optimization	14
question	1

42 Open 73 Closed Sort: Newest 1 2

- Investigate using only high precision for the solution vector in CG feature optimization #114
  Opened by mikeaclark a month ago
- Optimize multi-shift CG solver optimization #113
  Opened by mikeaclark 2 months ago
- Implement I-BiCGstab solver feature optimization #112
  Opened by mikeaclark 2 months ago
- Generalise QUDA's profiling utilities feature optimization #111
  Opened by jpfoley 2 months ago 1 comment
- Add support for loading / saving of spinor fields feature #107
  Opened by mikeaclark 2 months ago
- Implement one-sided communication MPI back end optimization #105
  Opened by mikeaclark 2 months ago 4 comments
- Twisted mass CG solver has bad performance #104
  Opened by mikeaclark 2 months ago 1 comment
- Register optimization for each dslash kernel optimization #103
  Opened by mikeaclark 2 months ago



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)

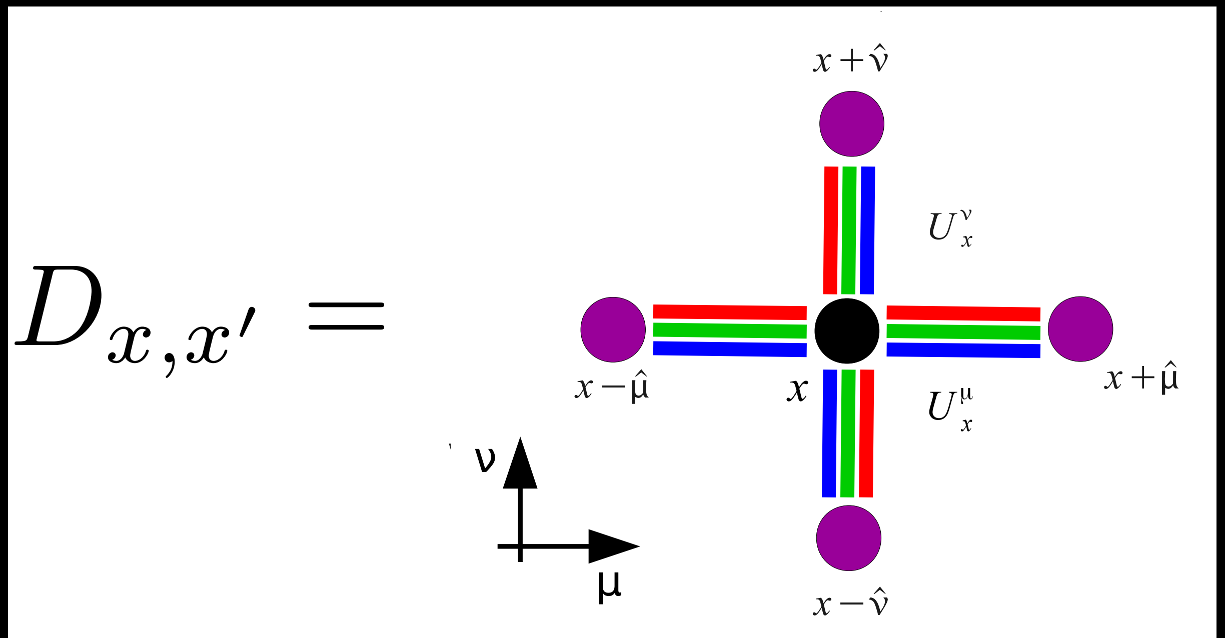
$$
\begin{aligned}
M_{x,x'} &= -\frac{1}{2}\sum_{\mu=1}^{4}\left(P^{-\mu}\otimes U_x^{\mu}\,\delta_{x+\hat{\mu},x'} \;+ P^{+\mu}\otimes U_{x-\hat{\mu}}^{\mu\dagger}\,\delta_{x-\hat{\mu},x'}\right) + (4+m+A_x)\delta_{x,x'} \\
&\equiv -\frac{1}{2}D_{x,x'} + (4+m+A_x)\delta_{x,x'}
\end{aligned}
$$

m quark mass parameter

  - 4-d nearest neighbor stencil operator acting on a vector field
- Eigen spectrum is complex (typically real positive)

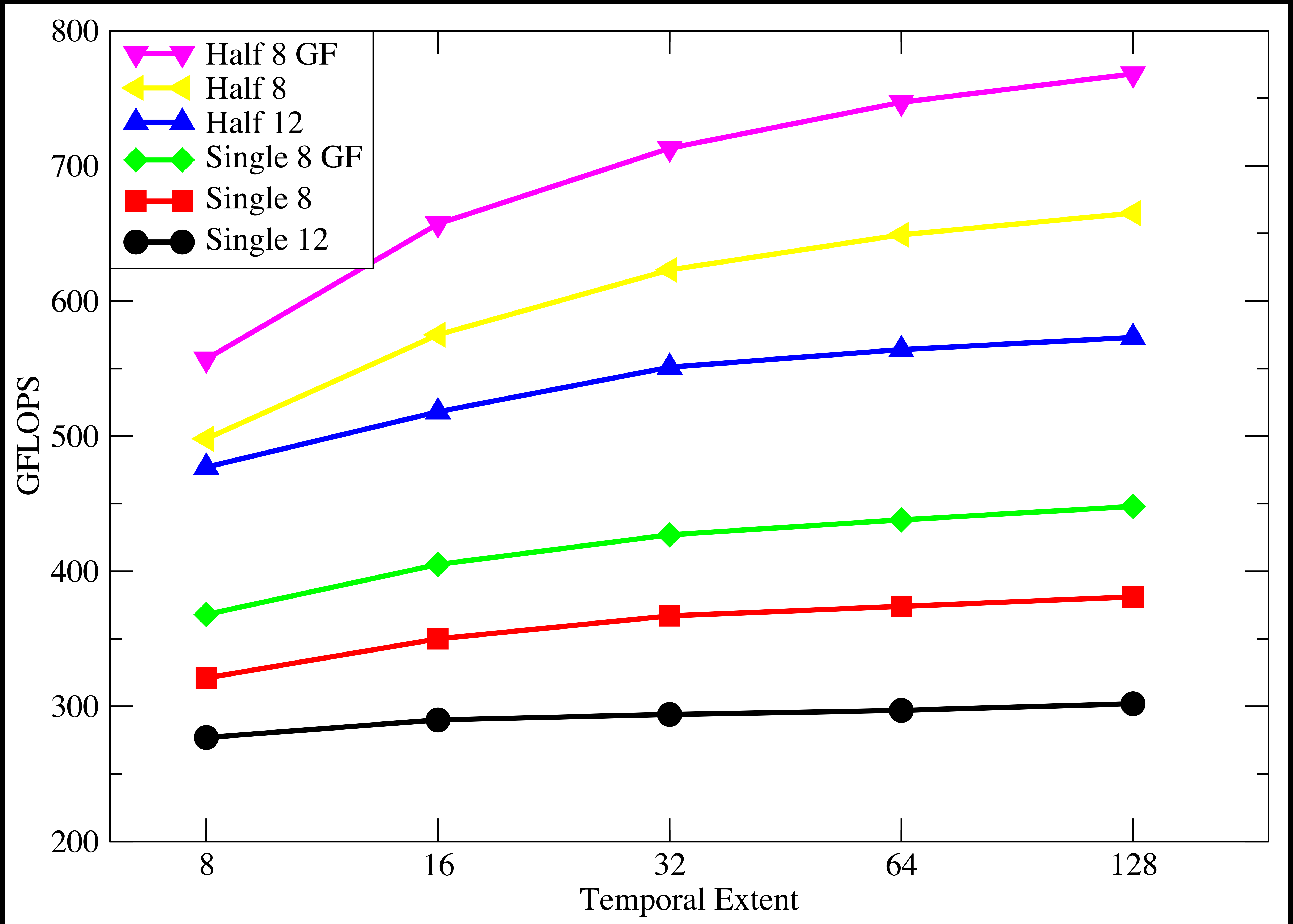
Mapping the Dirac operator to CUDA

- 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 => $3.3\text{x}10^6$ 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)
  - Similarity transforms to increase operator sparsity
  - Use 16-bit fixed-point representation
    - No loss in precision with mixed-precision solver
    - Almost a free lunch (small increase in iteration count)



Tesla K20X	
Gflops	3995
GB/s	250
AI	16

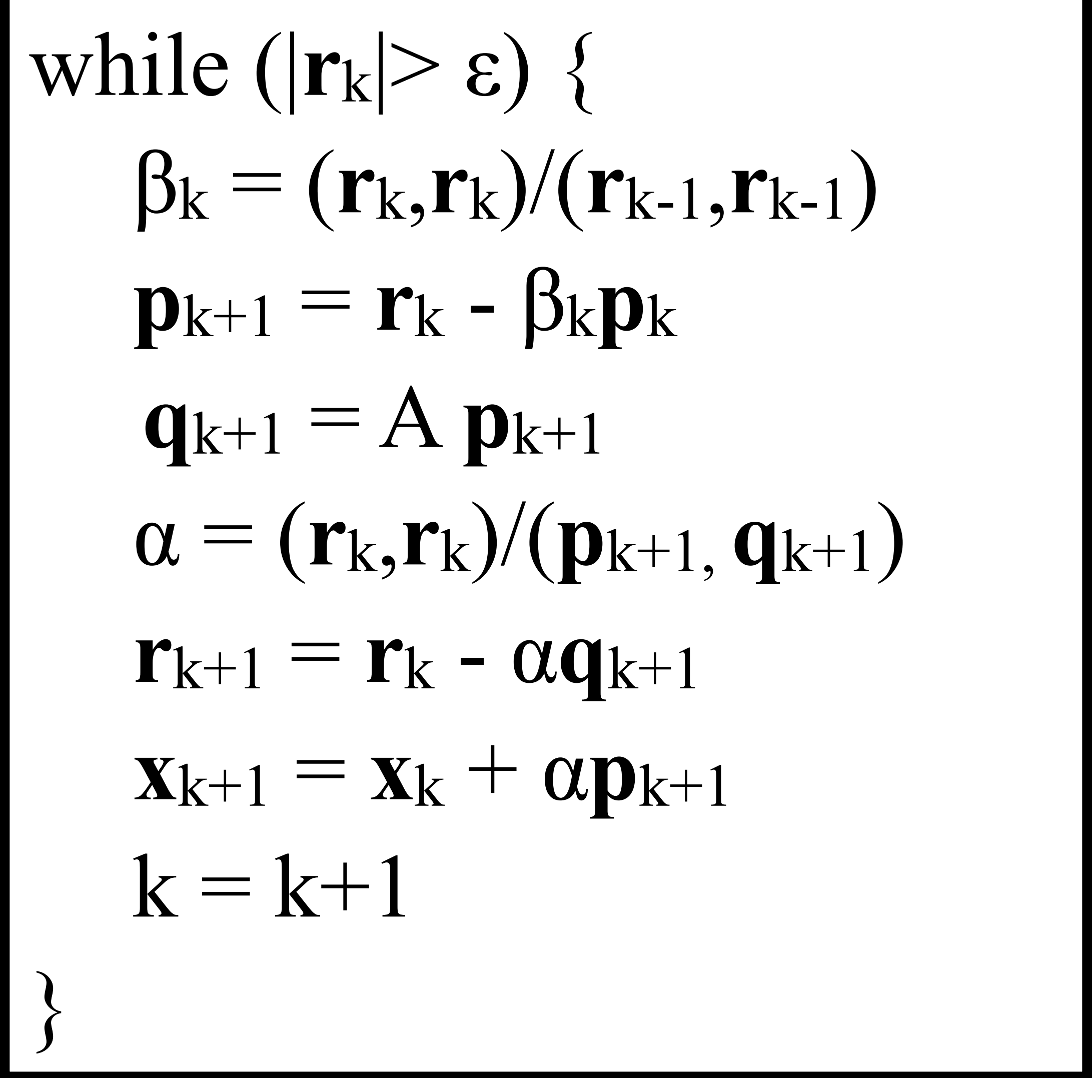
Kepler Wilson-Dslash Performance



Wilson Dslash
K20X performance
V = 24^3xT

Linear Solvers

- Nature of eigen-spectrum constrains which solver choice
  - CGNE / CGNR
  - BiCGstab
  - GMRES
- Condition number inversely proportional to mass
  - Light (realistic) masses are highly singular
- Entire solver algorithm must run on GPUs
  - Time-critical kernel is the stencil application (SpMV)
  - Also require BLAS level-1 type operations



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

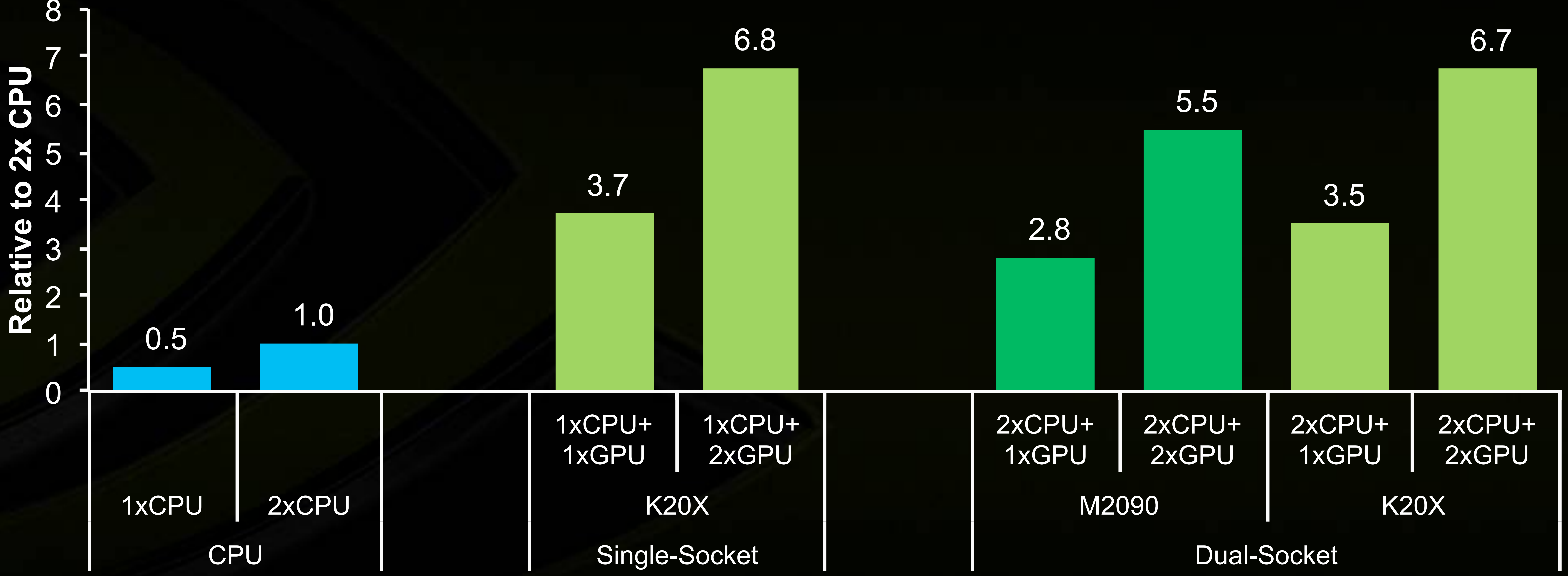
conjugate gradient

Chroma Benchmark with QUDA

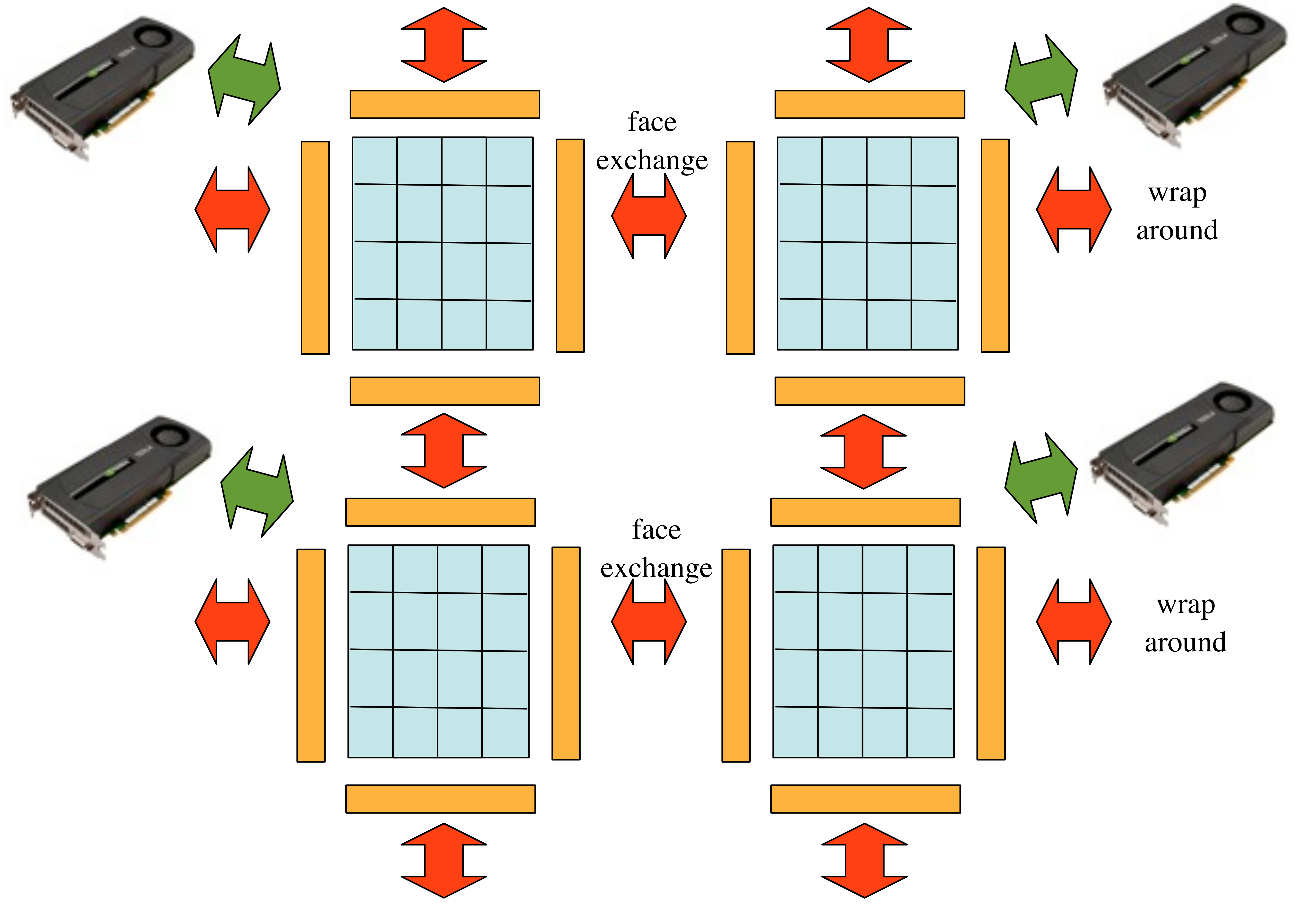
Chroma

24^3x128 lattice

Relative Performance (Propagator) vs. E5-2687w 3.10 GHz Sandy Bridge



Multi-GPU Implementation



- Scalable multi-GPU solver required
  - cuda streams to overlap comms and compute
  - Packing kernels for contiguous data for MPI
  - Utilize GPU Direct for low-latency inter-GPU communication

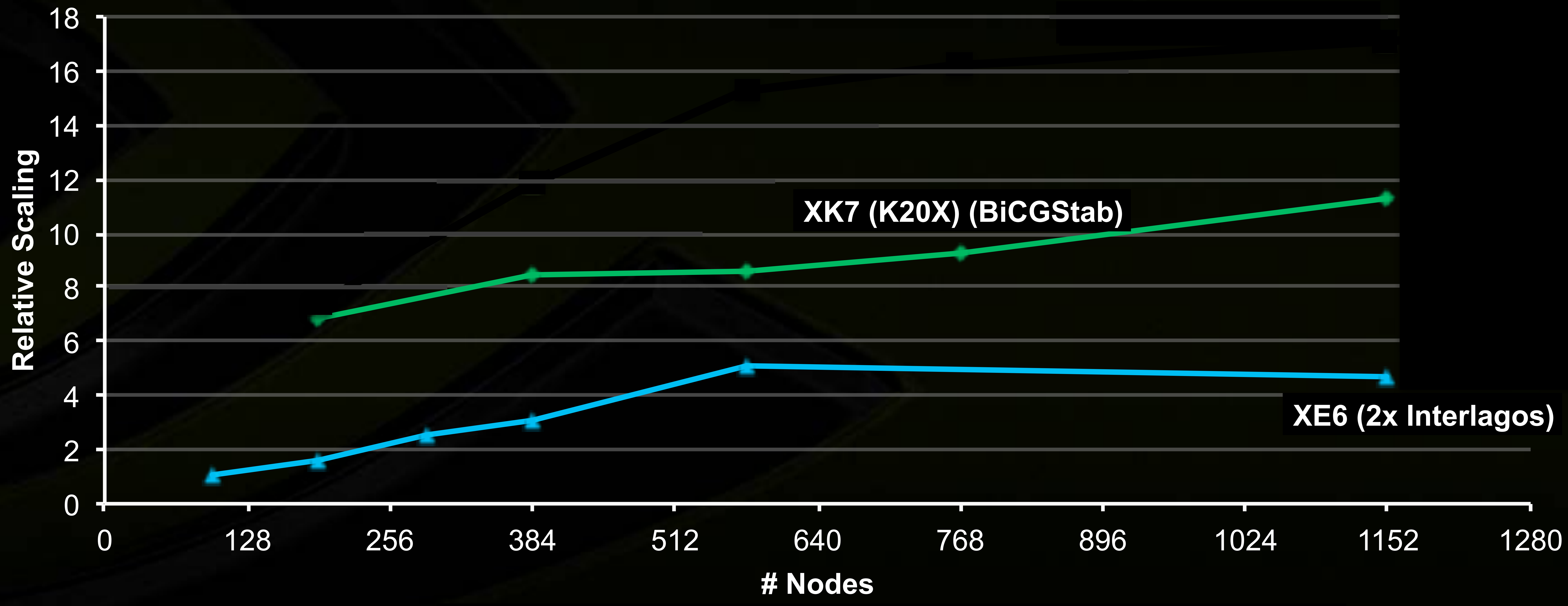
Strong Scaling Chroma with DD

Chroma

48^3x512 lattice

Relative Scaling (Application Time)

“XK7” node = XK7 (1x K20X + 1x Interlagos)
“XE6” node = XE6 (2x Interlagos)



Communication-Reducing Algorithms

- Reduce inter-node communication *and* synchronization
  - Inter-node communication comes from face exchange
  - Synchronization comes from global sums
- Utilize domain-decomposition techniques, e.g., Additive Schwarz

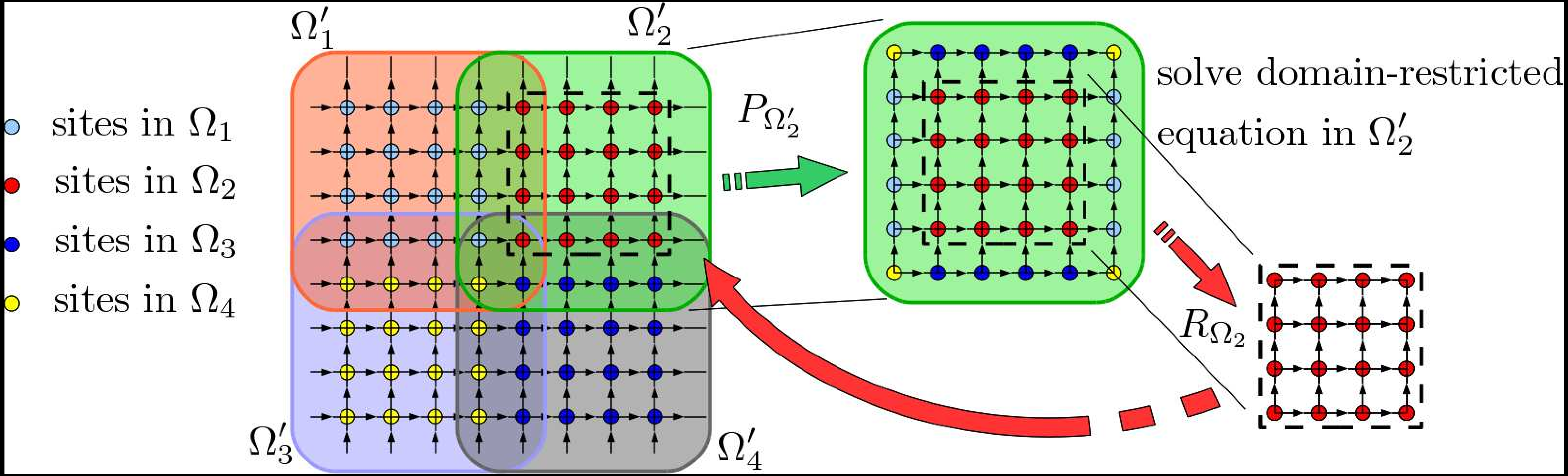
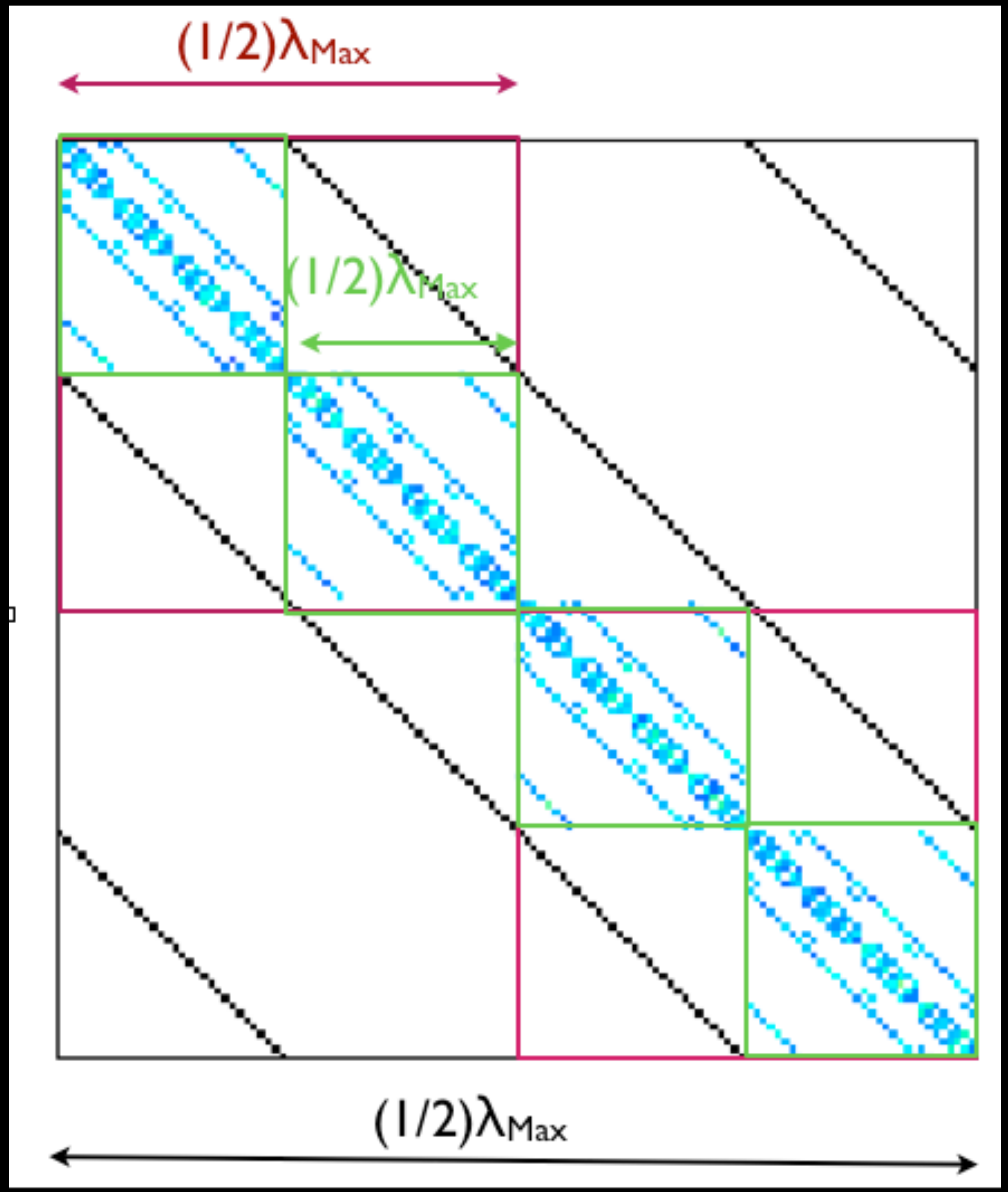


figure taken from Osaki and Ishikawa

Communication-Reducing Algorithms

- Non-overlapping blocks - simply switch off inter-node comms
- Preconditioner is a gross approximation
  - Use an iterative solver to solve each domain system
  - Only block-local sums required
  - Require only ~10 iterations of domain solver $\Longrightarrow$ 16-bit precision
  - Need to use a flexible solver $\Longrightarrow$ GCR
- Block-diagonal preconditioner impose λ cutoff
  - Limits scalability of algorithm
  - In practice, non-preconditioned part becomes source of Amdahl



GPU TECHNOLOGY CONFERENCE

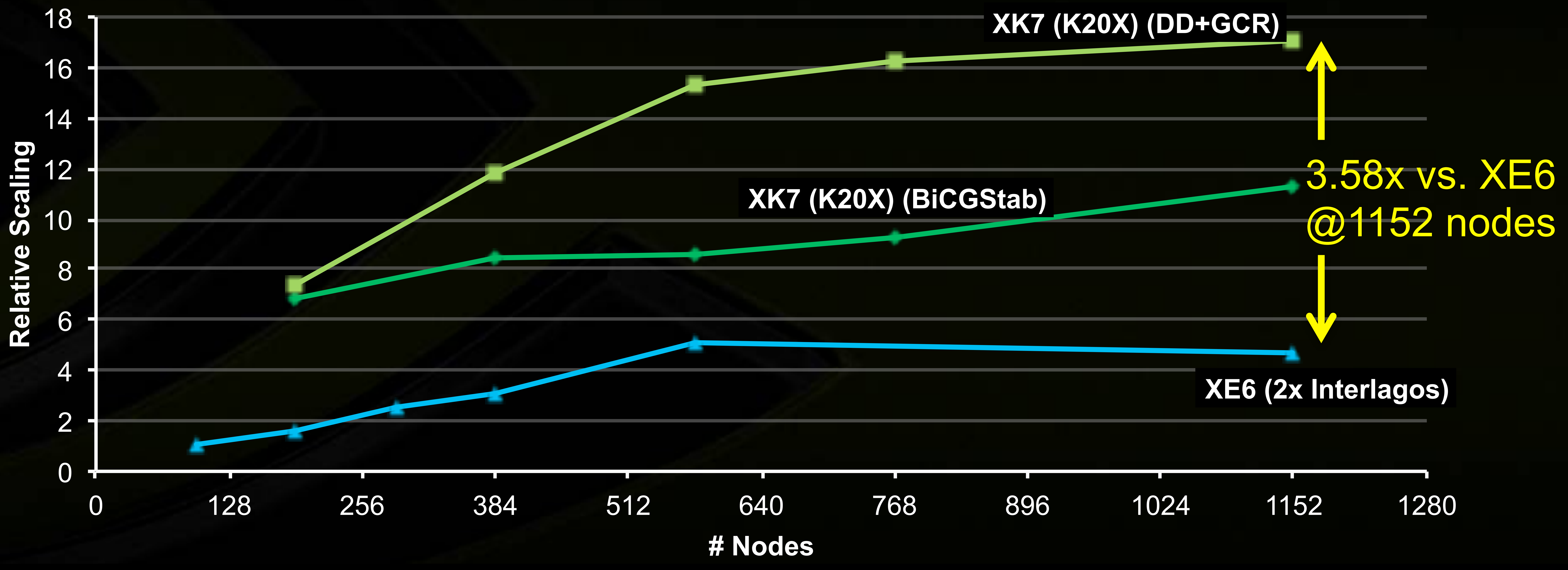
Strong Scaling Chroma with DD

Chroma

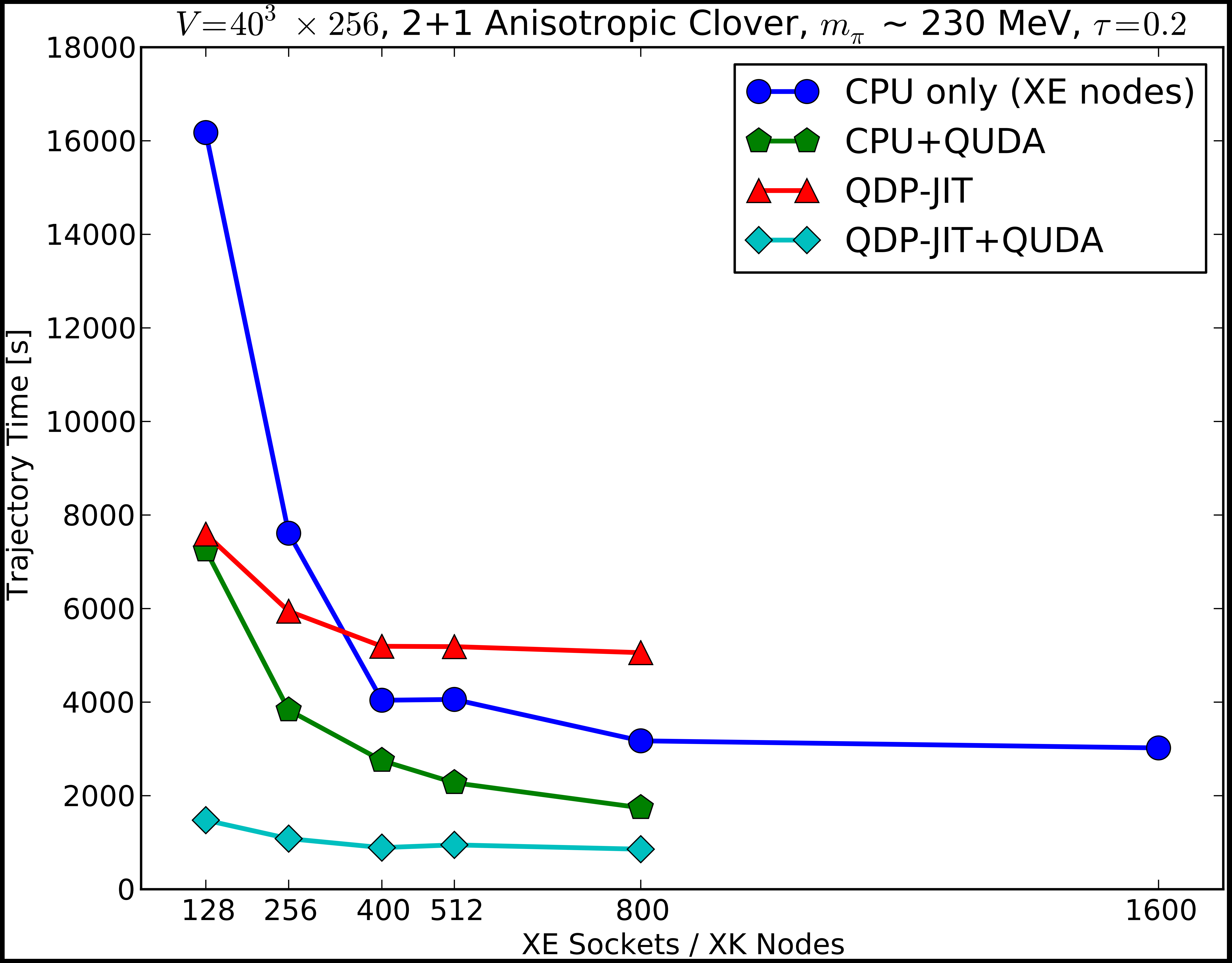
48^3x512 lattice

Relative Scaling (Application Time)

“XK7” node = XK7 (1x K20X + 1x Interlagos)
“XE6” node = XE6 (2x Interlagos)

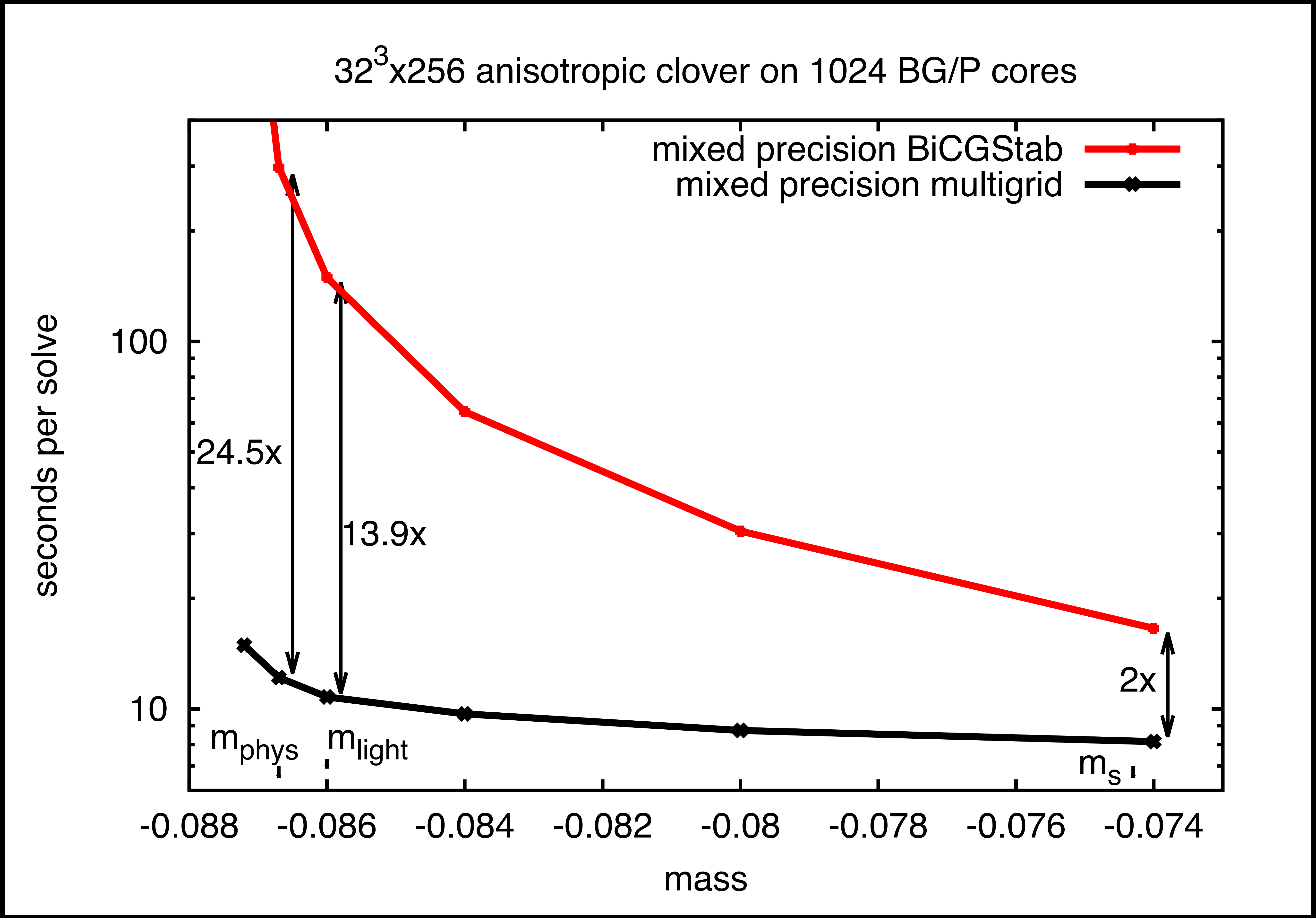


Full Gauge Generation with Chroma



Winter, Clark, Joo and Edwards, IPDPS 2014

Adaptive Geometric Multigrid



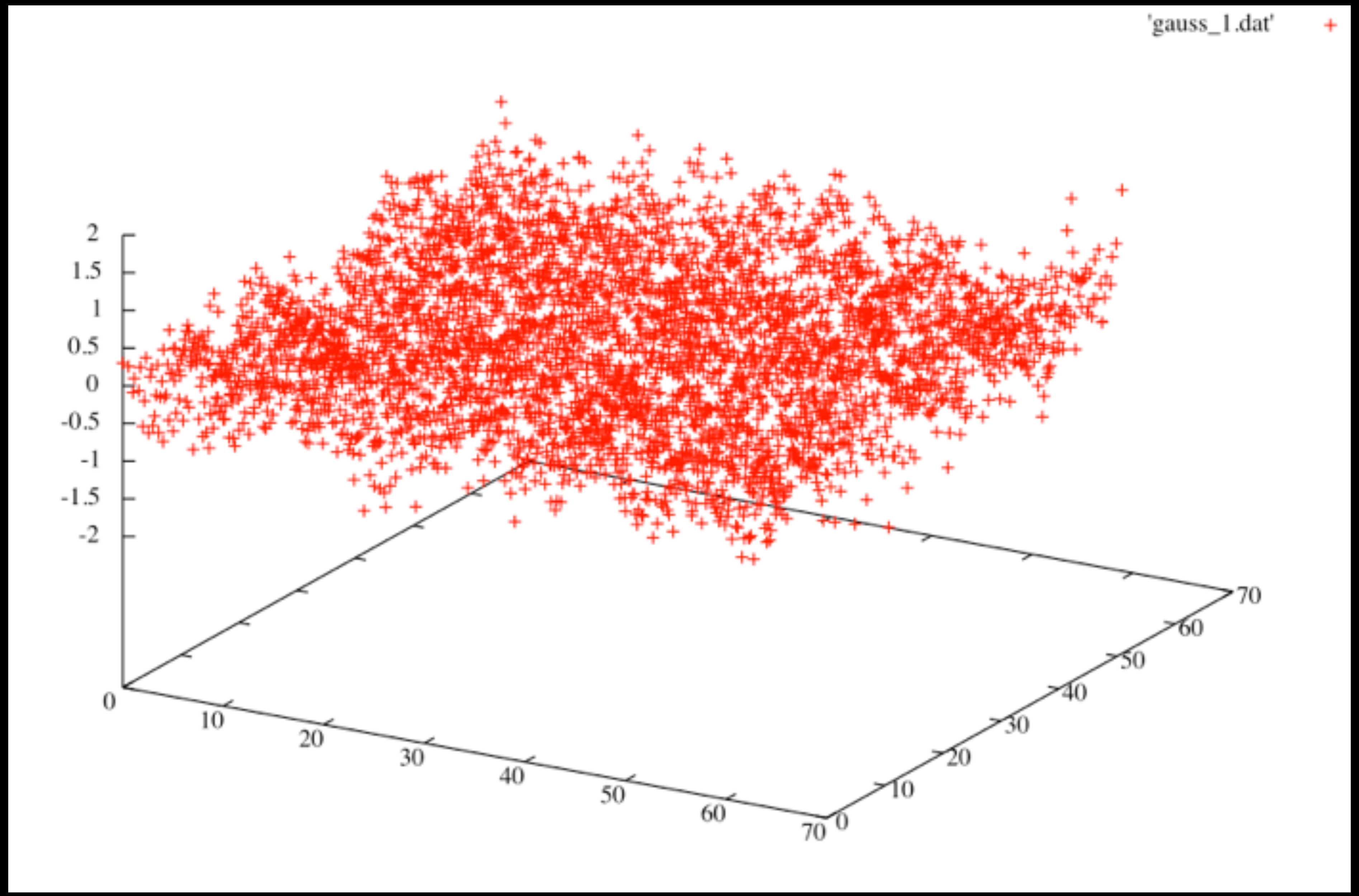
Osborn *et al 2011*

Introduction to Multigrid

- Preconditioner is a gross approximation
- 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 $e_i = -x_i$

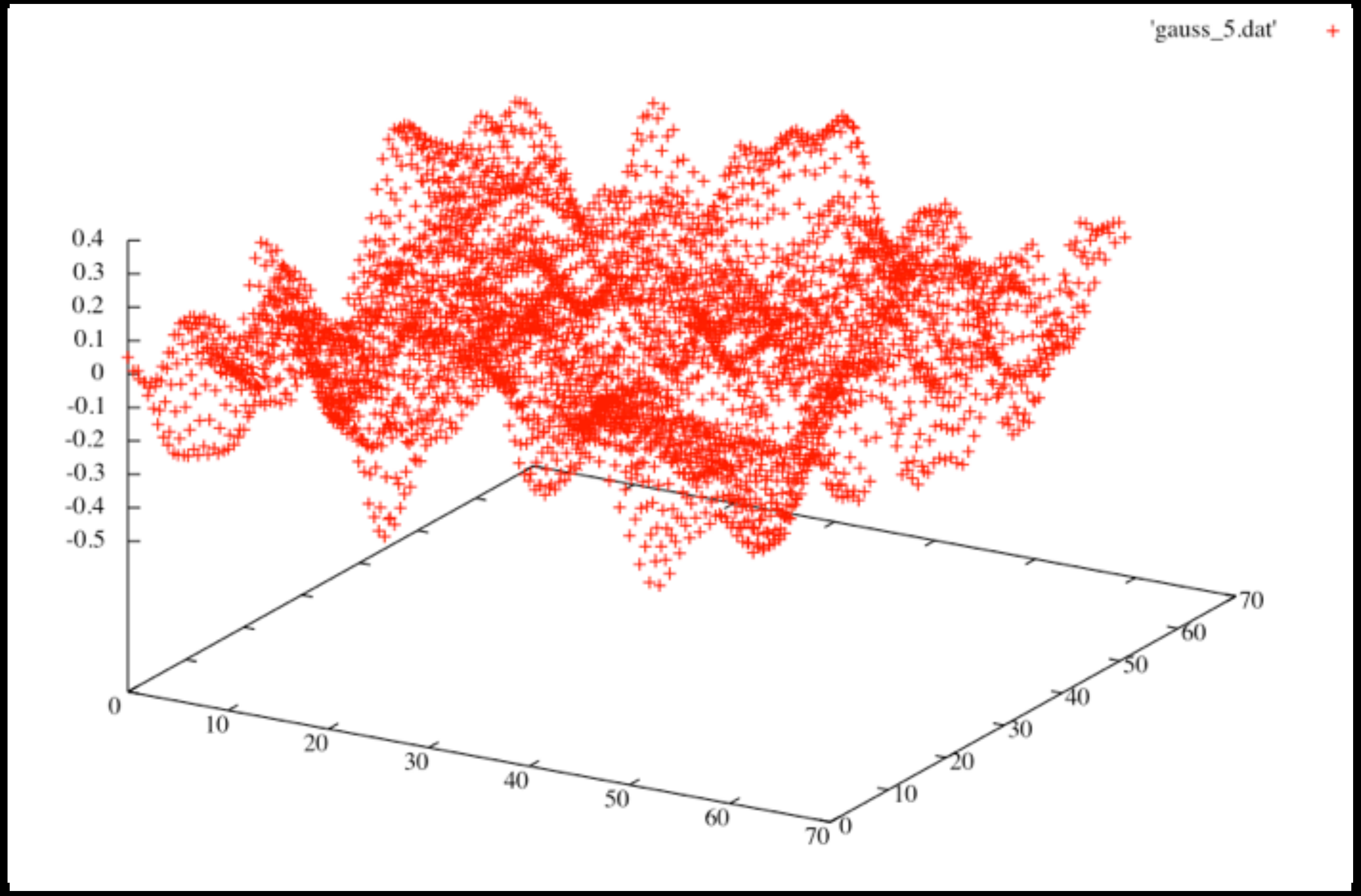
Introduction to Multigrid

- Preconditioner is a gross approximation
- 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 $e_i = -x_i$



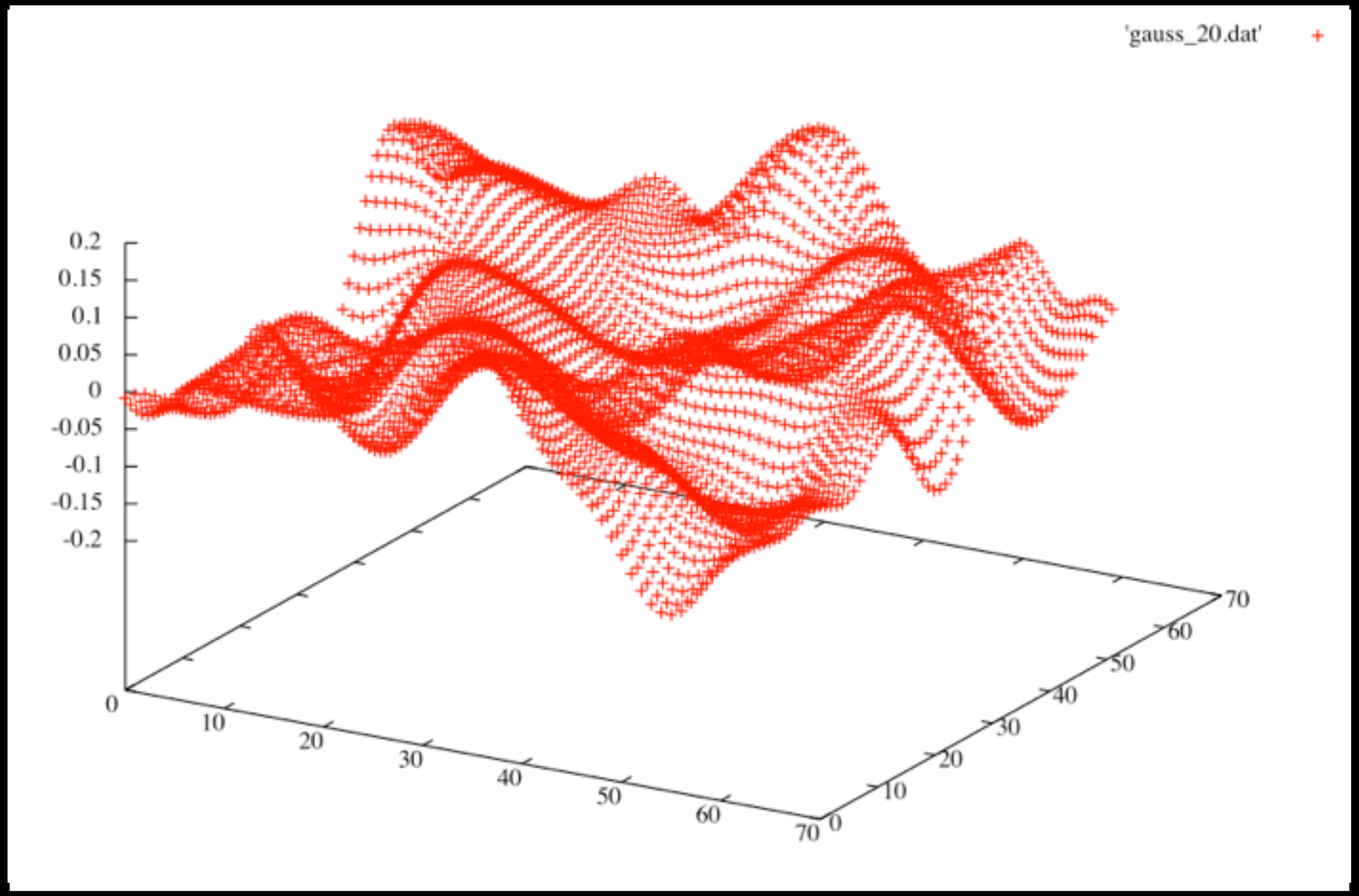
Introduction to Multigrid

- Preconditioner is a gross approximation
- 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 $e_i = -x_i$



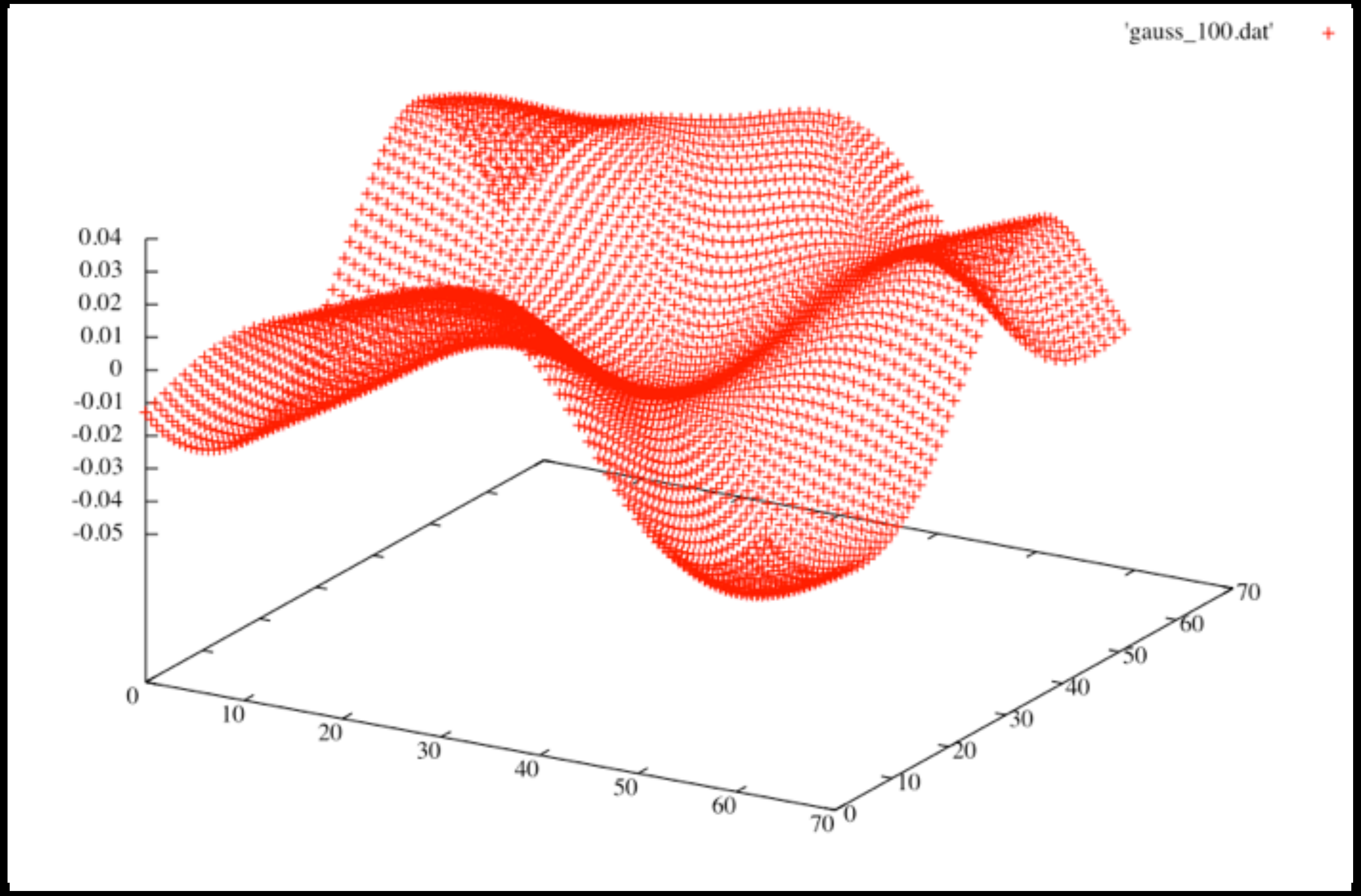
Introduction to Multigrid

- Preconditioner is a gross approximation
- 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 $e_i = -x_i$



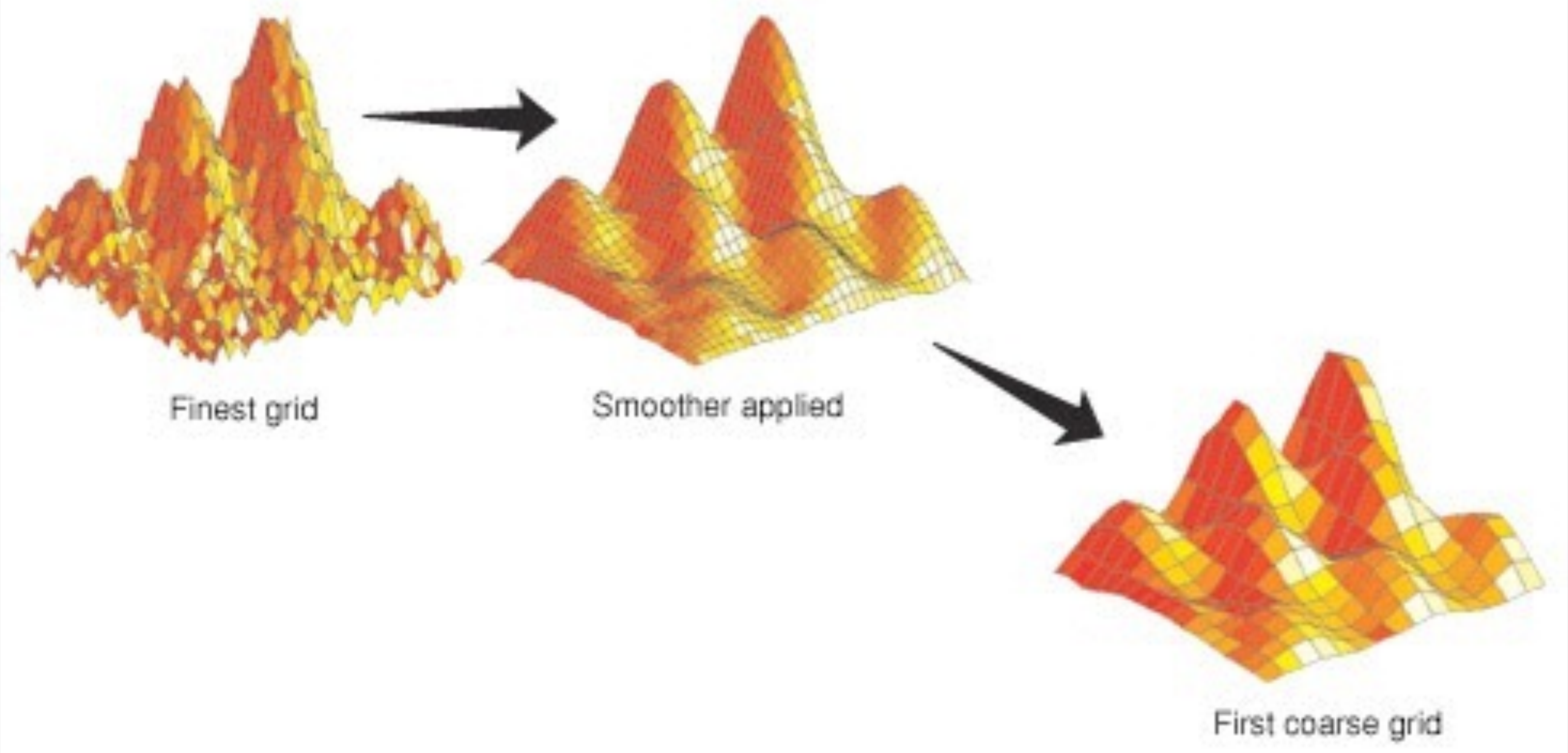
Introduction to Multigrid

- Preconditioner is a gross approximation
- 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

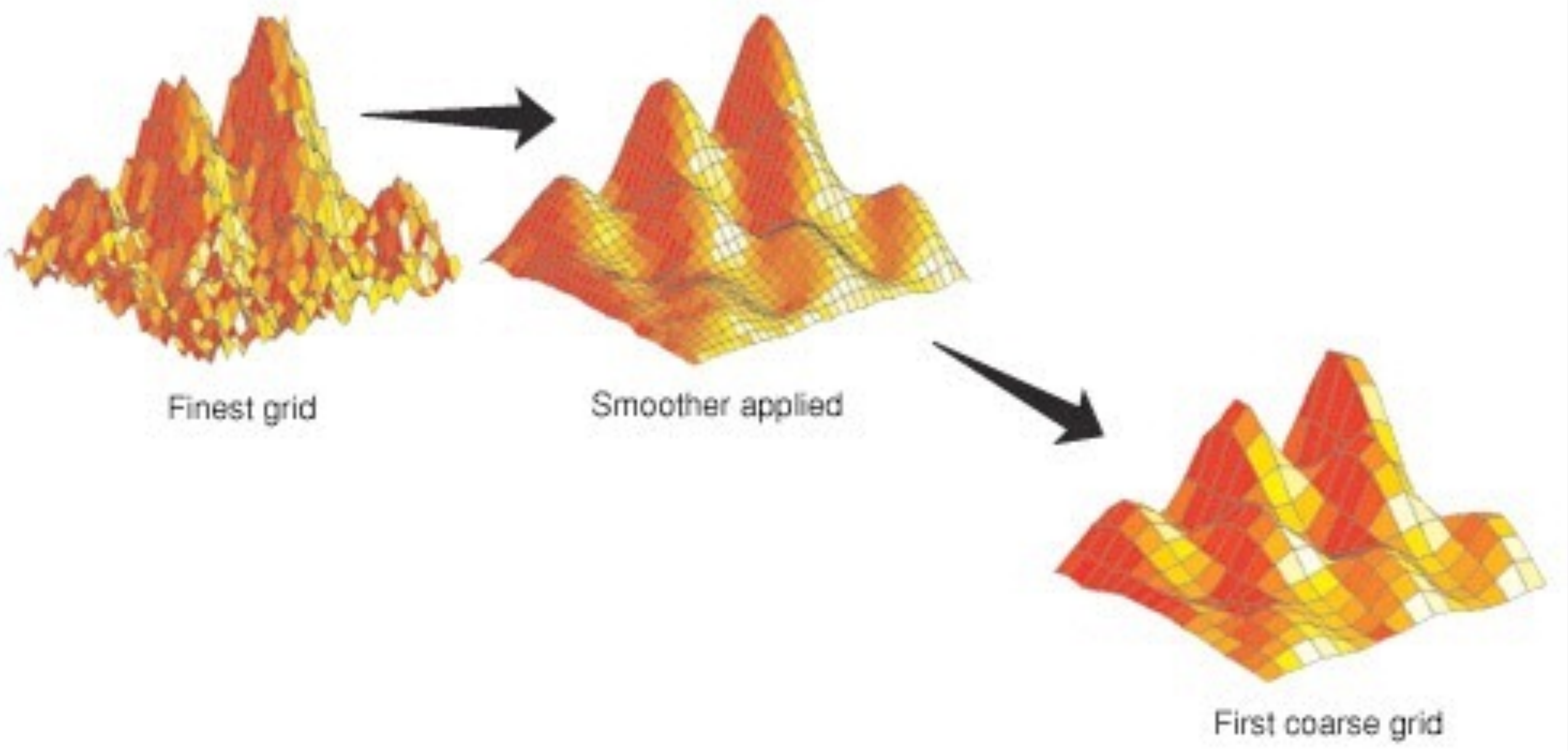
- Low frequency error modes are smooth
- Can accurately represent on coarse grid



- Low frequency on fine => high frequency on coarse
- Relaxation effective agin on coarse grid
- Interpolate back to fine grid

Multigrid V-cycle

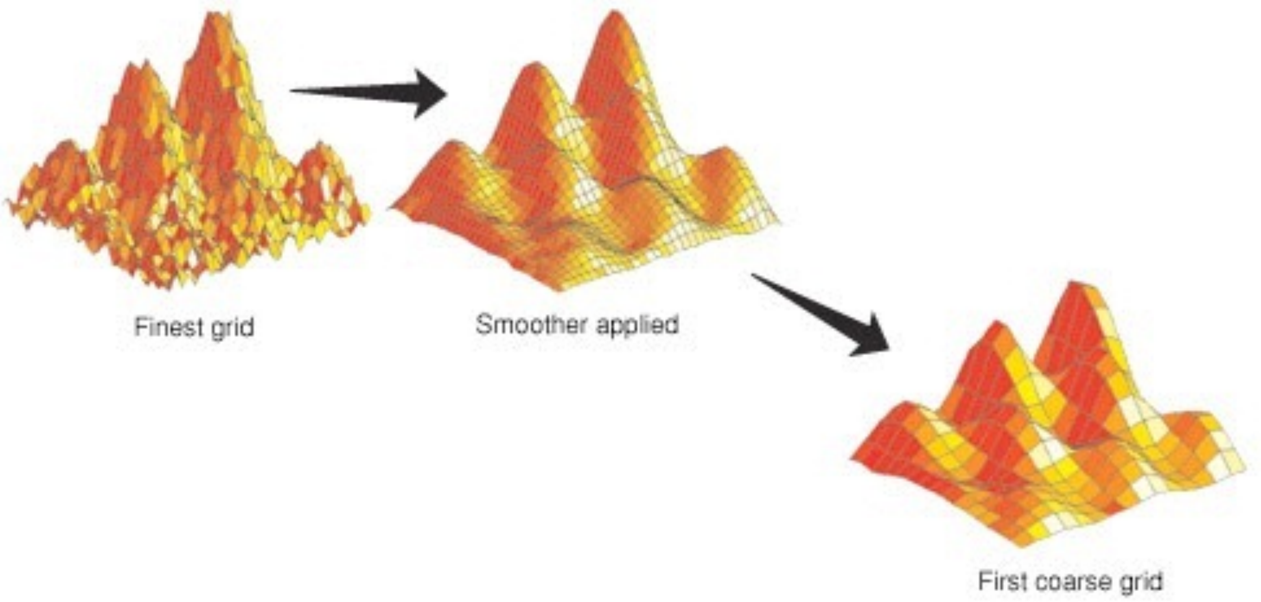
- Solve
  1. Smooth
  2. Compute residual
  3. Restrict residual
  4. Recurse on coarse problem
  5. Prolongate correction
  6. Smooth
  7. If not converged, goto 1



- 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



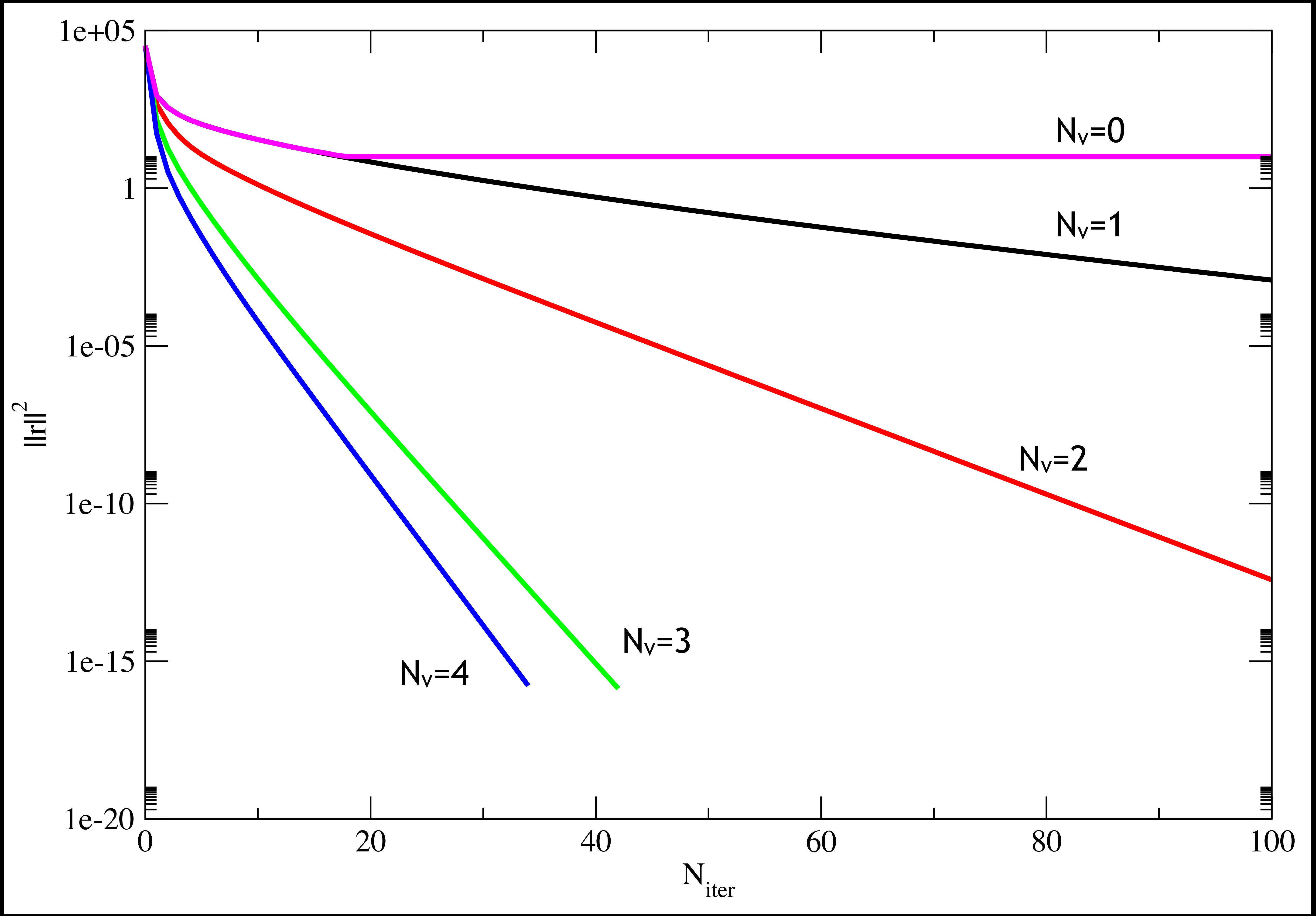
- Setup

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

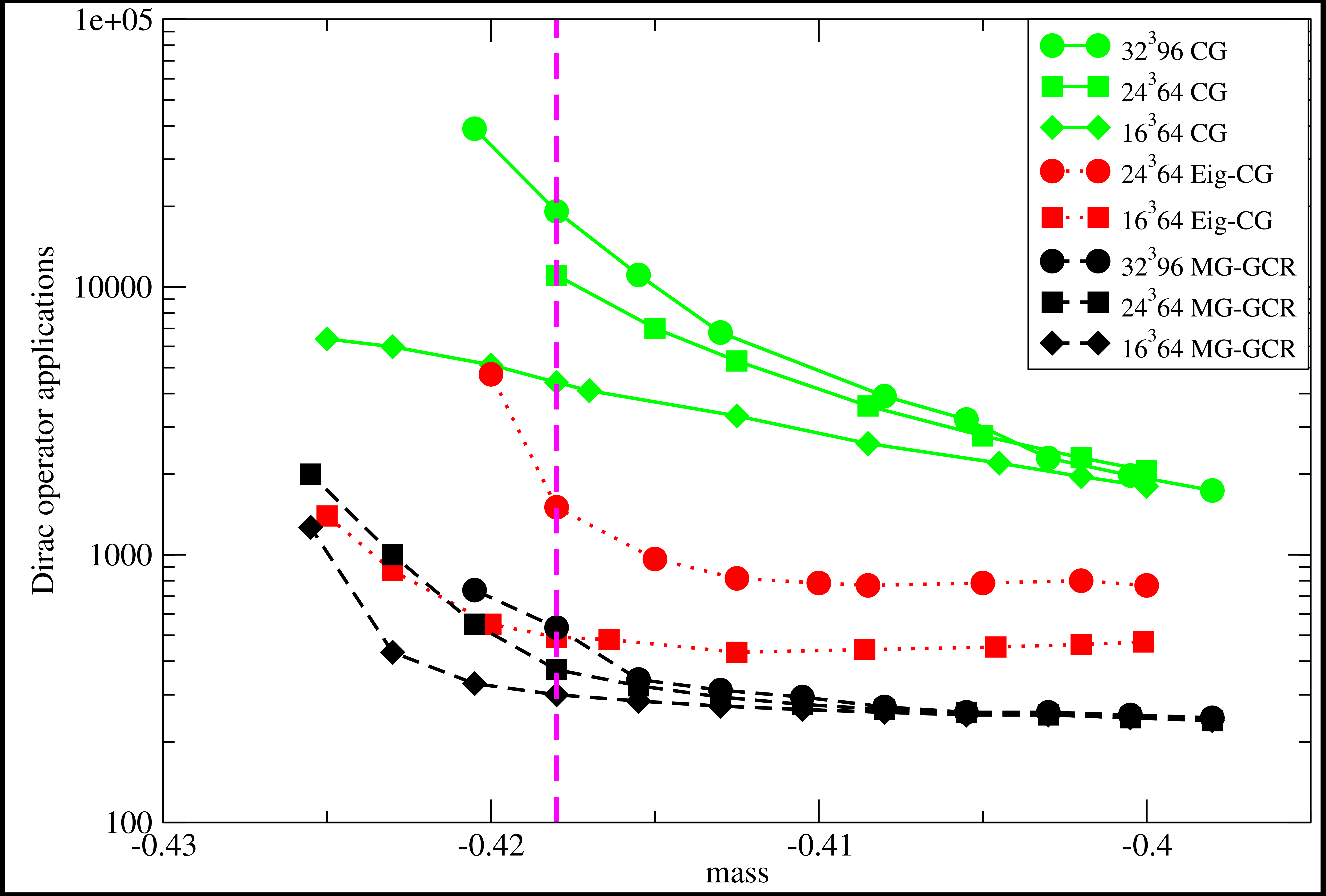
Adaptive Geometric Multigrid

GPU TECHNOLOGY CONFERENCE

Adaptive Geometric Multigrid



Adaptive Geometric Multigrid



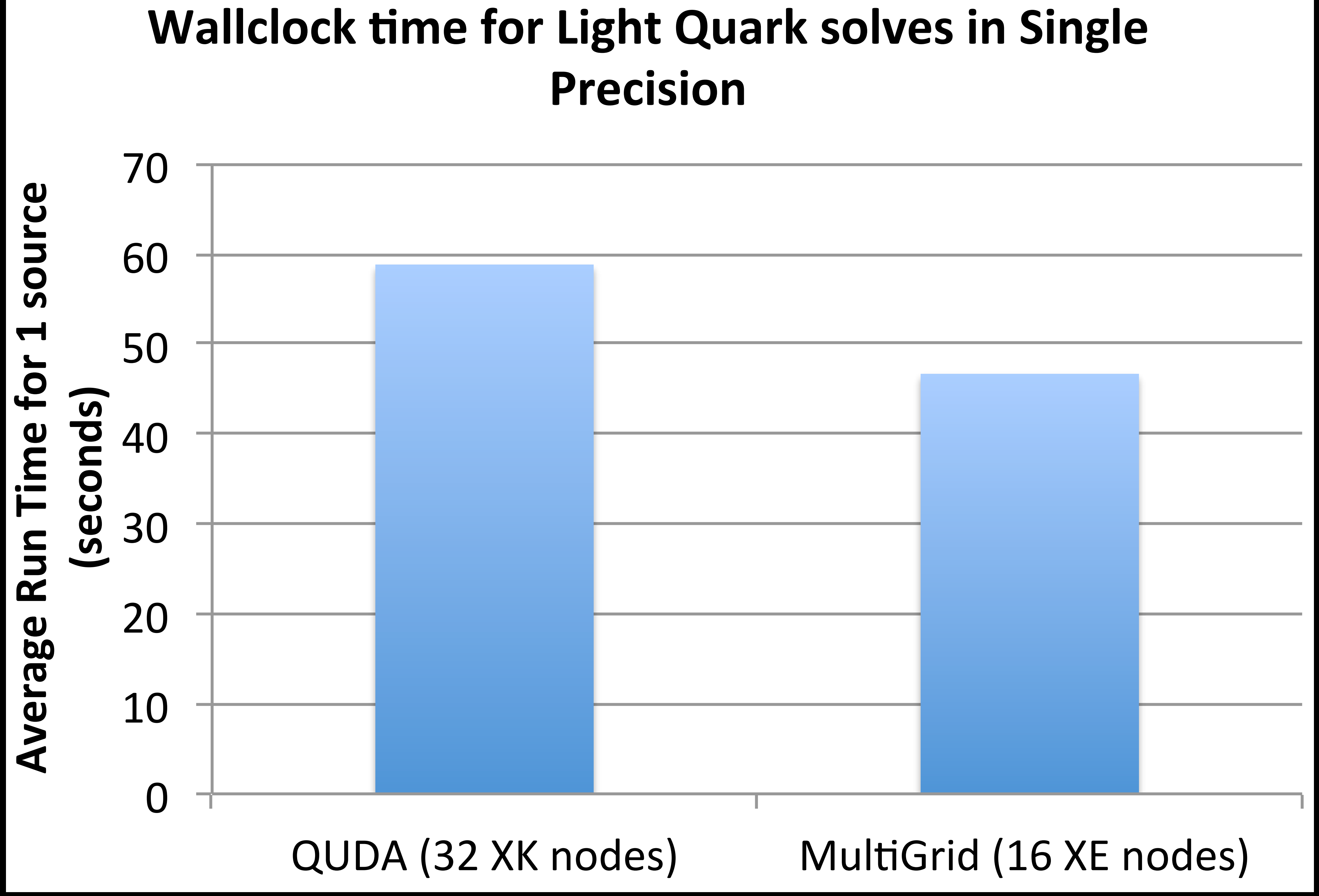
240 vectors

20 vectors

Babich *et al* 2010

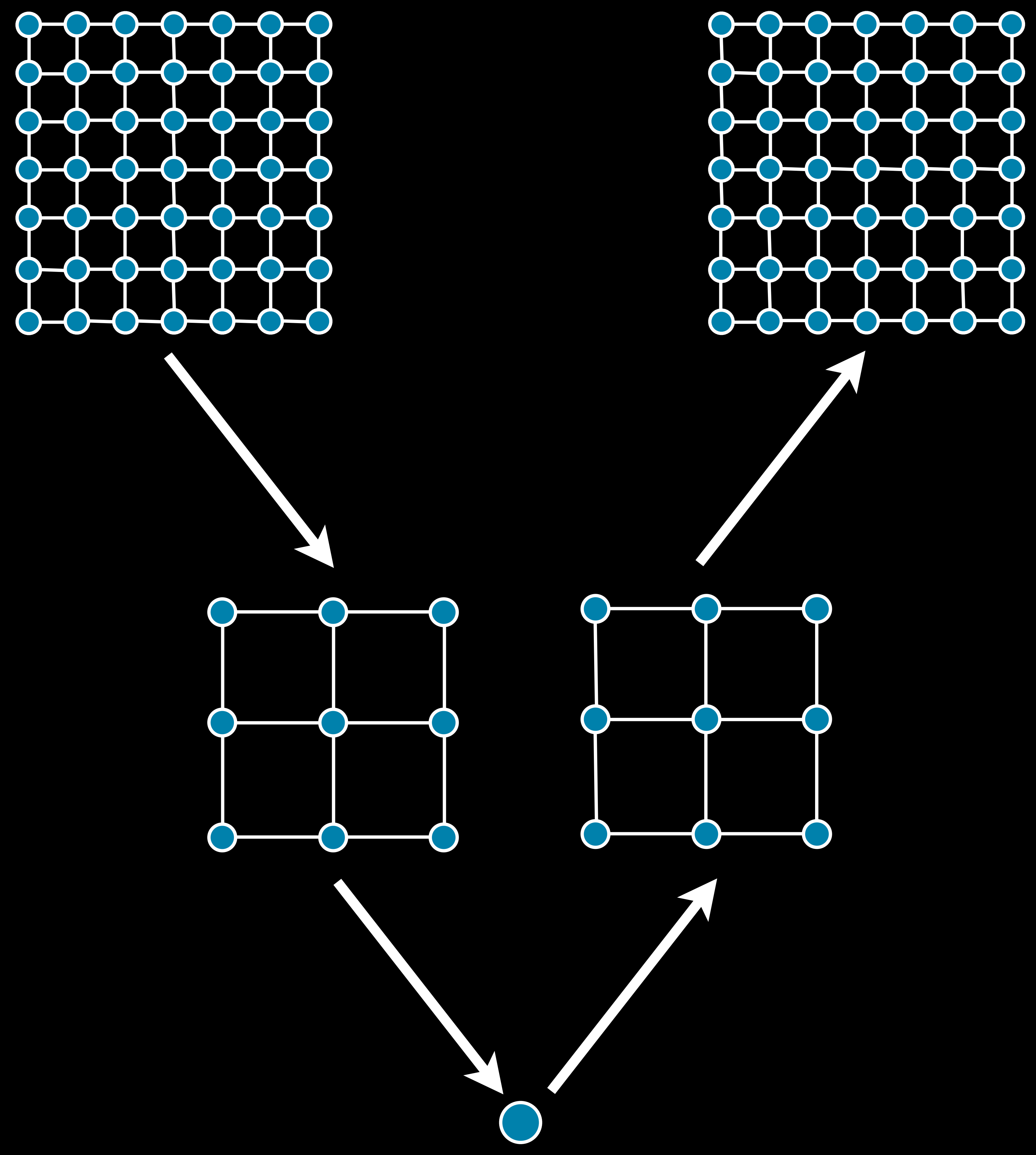
Motivation

- A CPU running the optimal algorithm surpasses a highly tuned GPU sub-optimal algorithm
- For competitiveness, MG on GPU is a must
- Seek multiplicative gain of architecture and algorithm



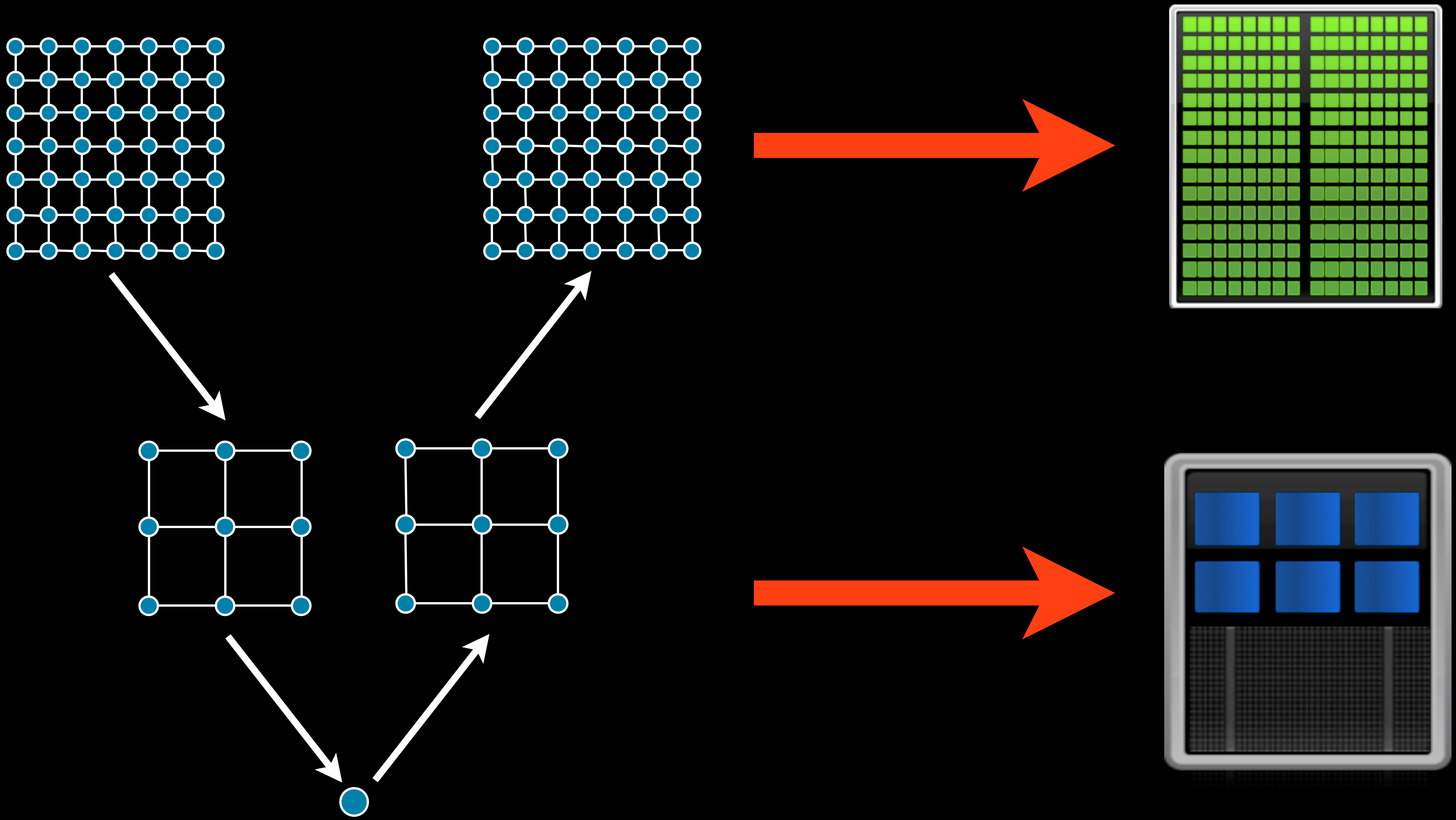
Chroma propagator benchmark
Figure by Balint Joo
MG Chroma integration by Saul Cohen
MG Algorithm by James Osborn

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 decomposes problem into throughput and latency parts

Hierarchical algorithms on heterogeneous architectures



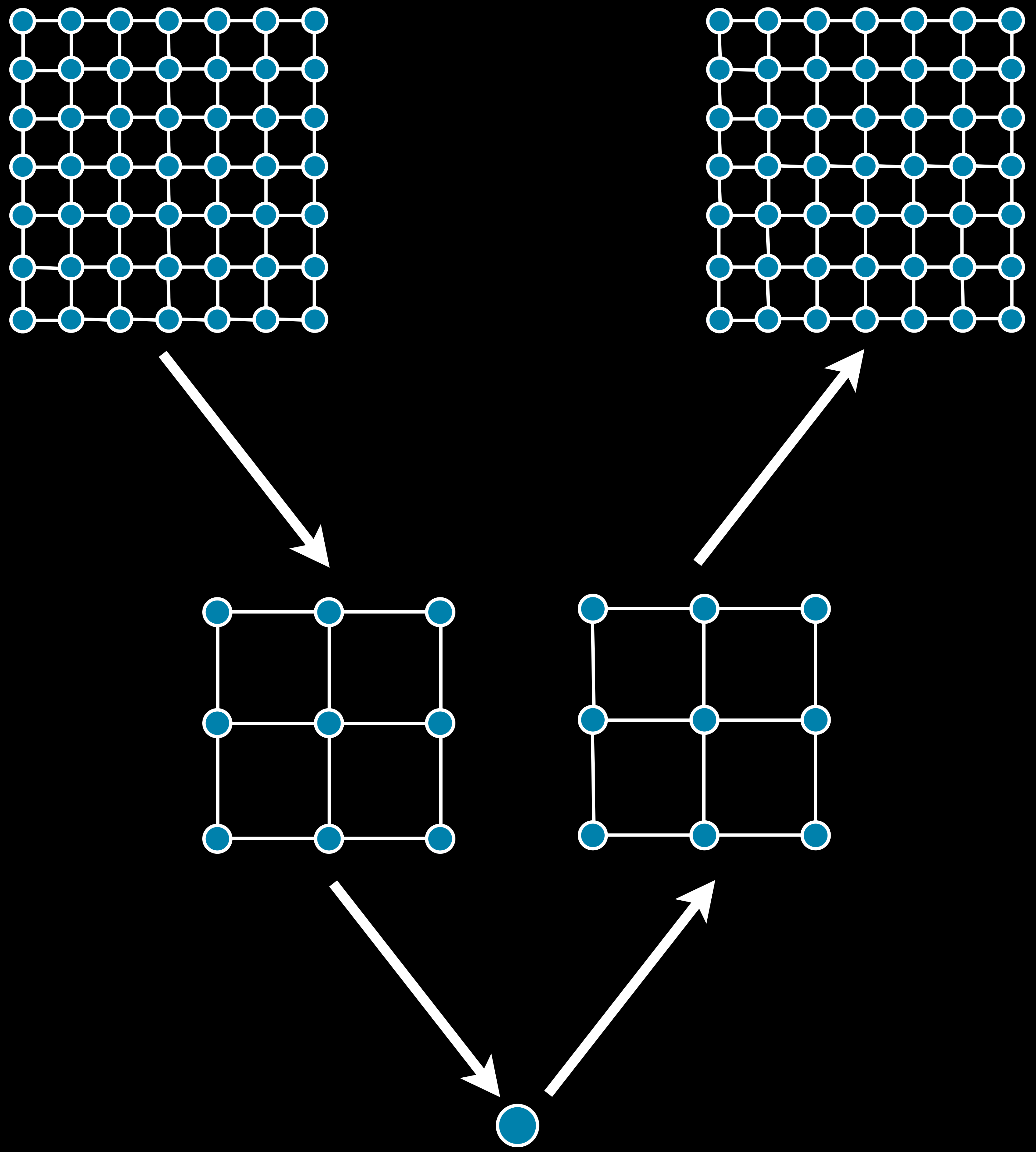
GPU

Thousands of cores for parallel processing

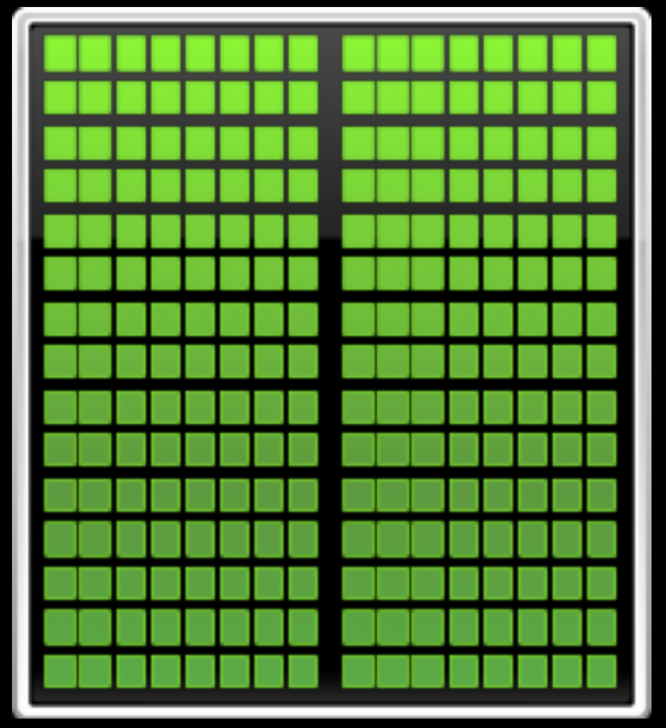
CPU

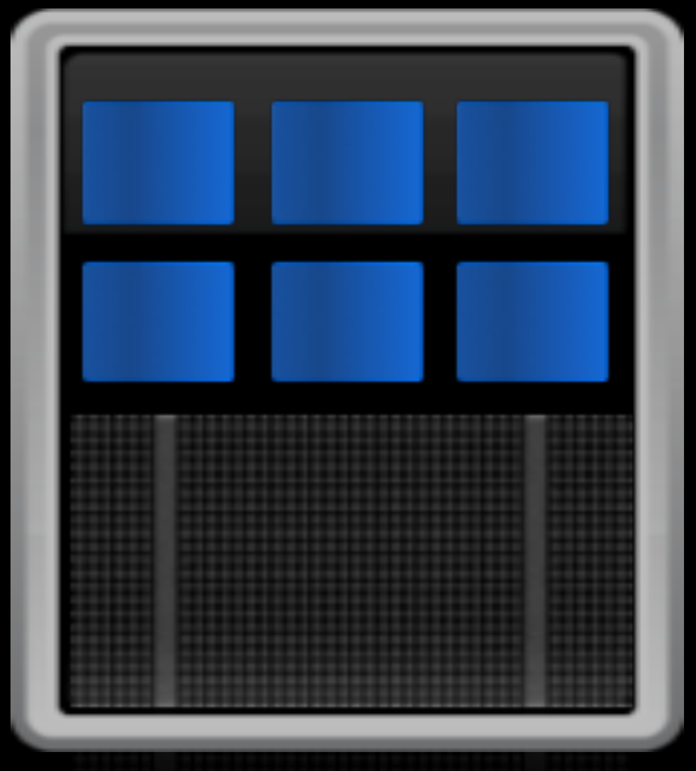
Few Cores optimized for serial work

Heterogeneous Updating Scheme

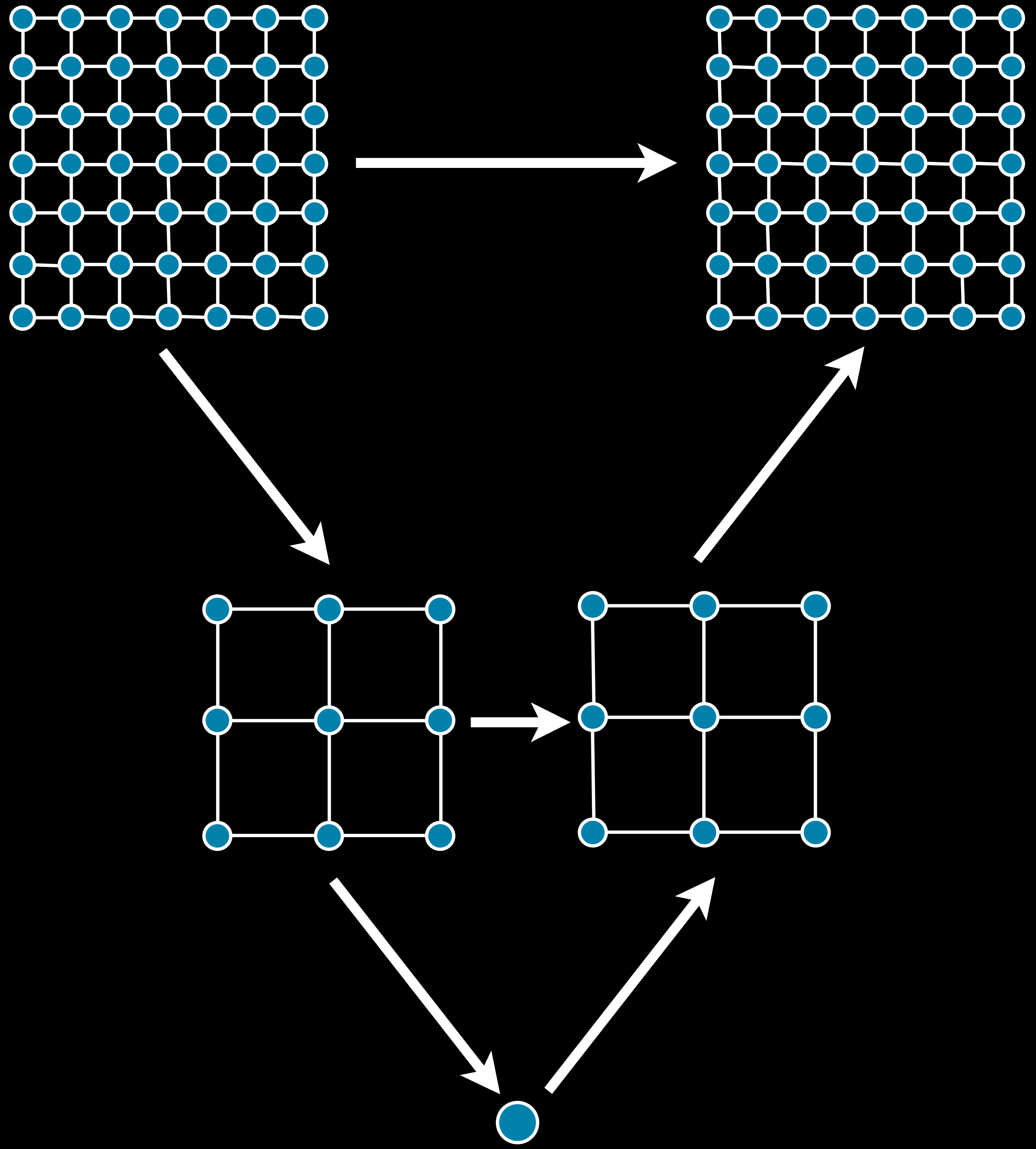


- Multiplicative MG is necessarily serial process
  - Cannot utilize both GPU and CPU simultanesouly

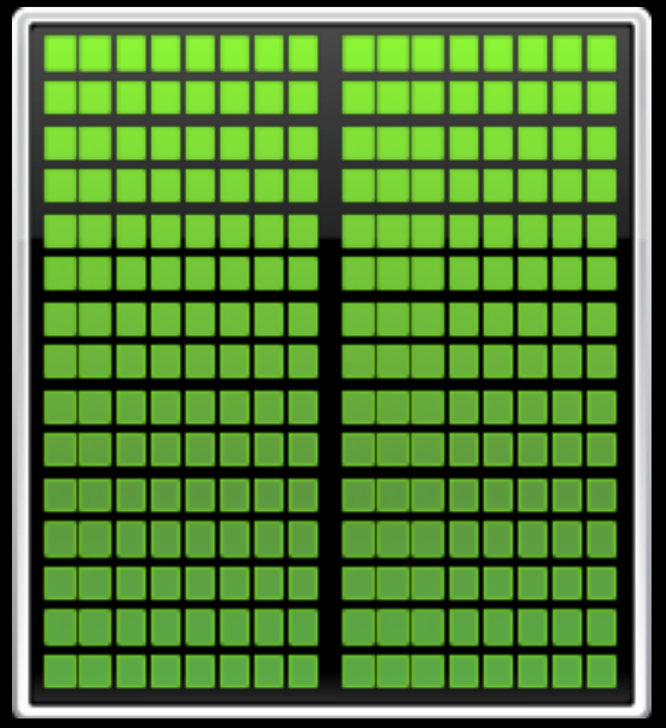


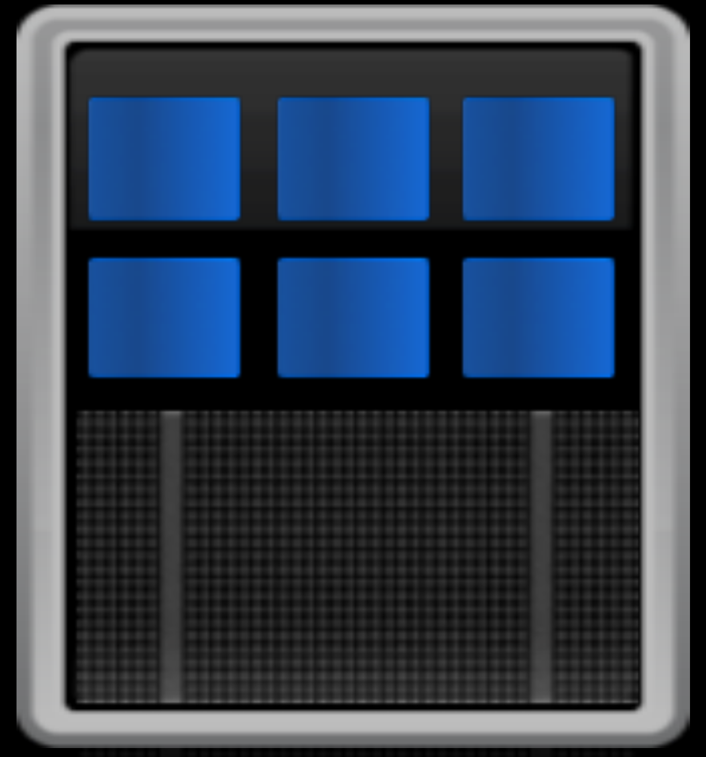


Heterogeneous Updating Scheme



- Multiplicative MG is necessarily serial process
  - Cannot utilize both GPU and CPU simultanesouly
- Additive MG is parallel
  - Can utilize both GPU and CPU simultanesouly
- Additive MG requires accurate coarse-grid solution
  - Not amenable to multi-level
  - Only need additive correction at CPU<->GPU level interface
- Accurate coarse-grid solution maybe cheaper than serialization / synchronization



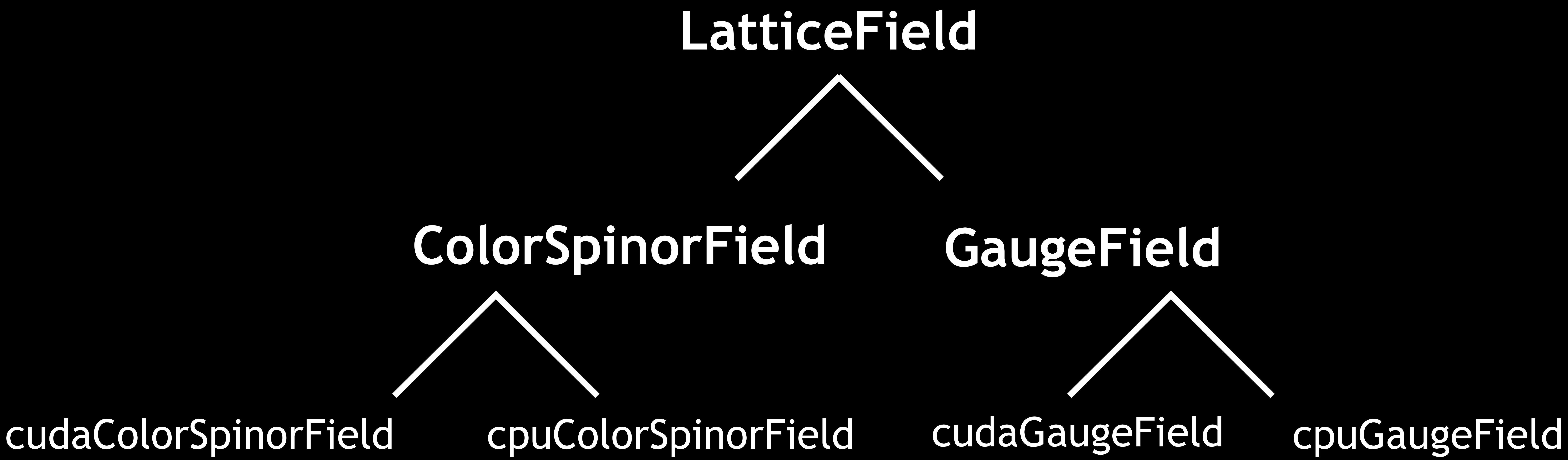


Design Goals

- Performance
  - LQCD typically reaches high % peak peak performance
  - Brute force can beat the best algorithm
- Flexibility
  - Deploy level *i* on either CPU or GPU
  - All algorithmic flow decisions made at runtime
  - Autotune for a given *heterogeneous* architecture
- (Short term) Provide optimal solvers to legacy apps
  - e.g., Chroma, CPS, MILC, etc.
- (Long term) Hierarchical algorithm toolbox
  - Little to no barrier to trying new algorithms

Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



LatticeField

ColorSpinorField

GaugeField

cudaColorSpinorField

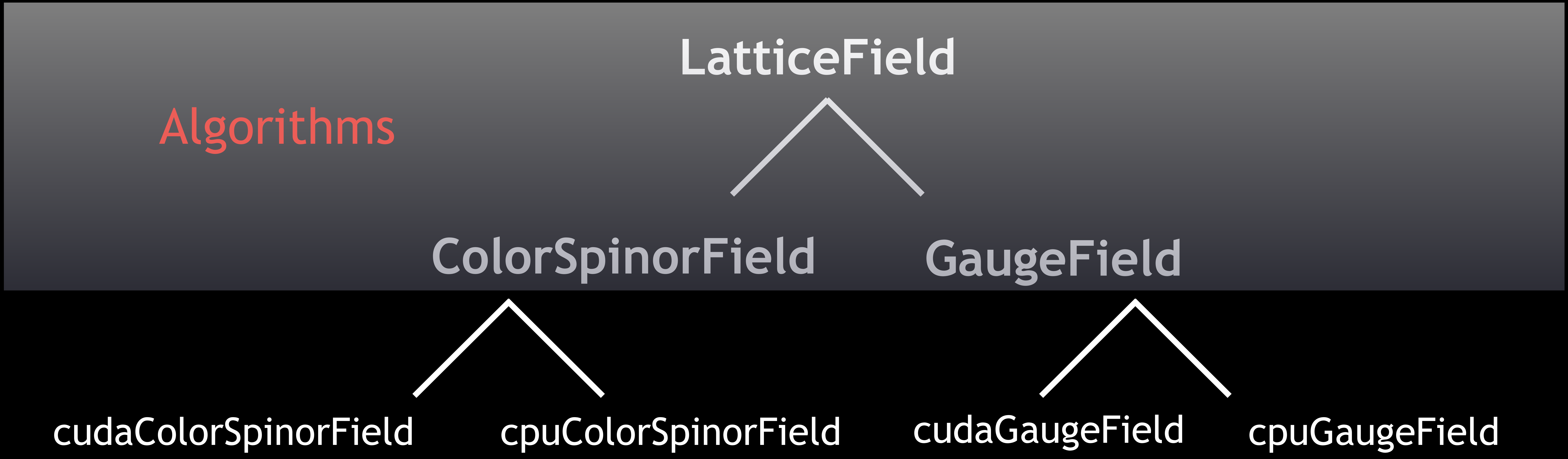
cpuColorSpinorField

cudaGaugeField

cpuGaugeField

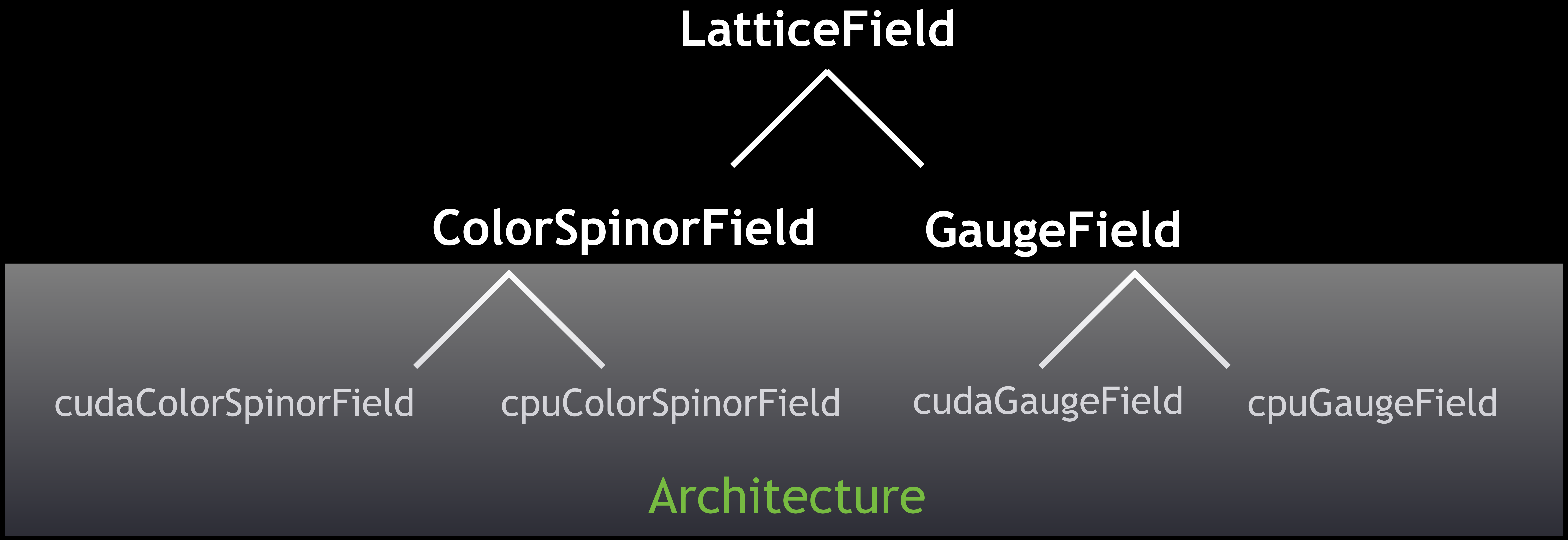
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



LatticeField

ColorSpinorField

GaugeField

cudaColorSpinorField

cpuColorSpinorField

cudaGaugeField

cpuGaugeField

Architecture

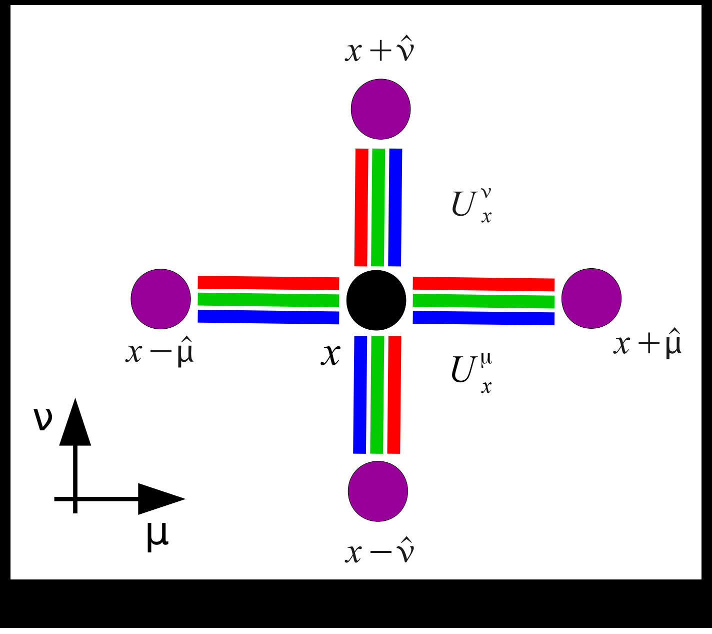
Multigrid and QUDA

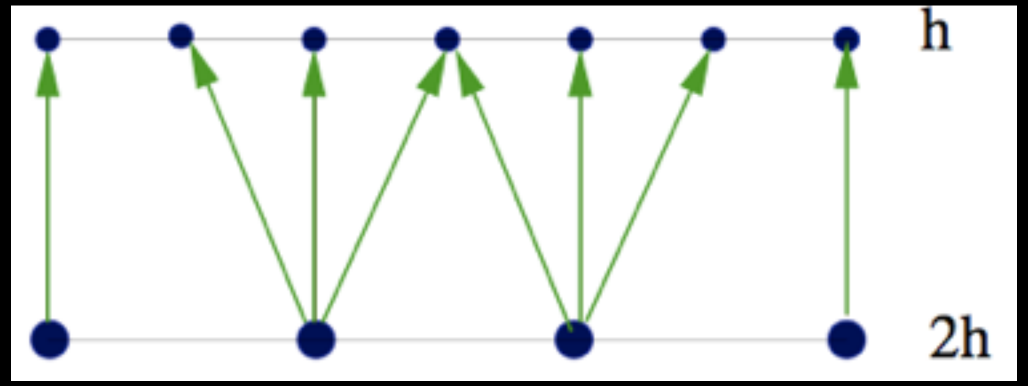
- Algorithms are straightforward to write down
- QUDA Multigrid V-cycle source:

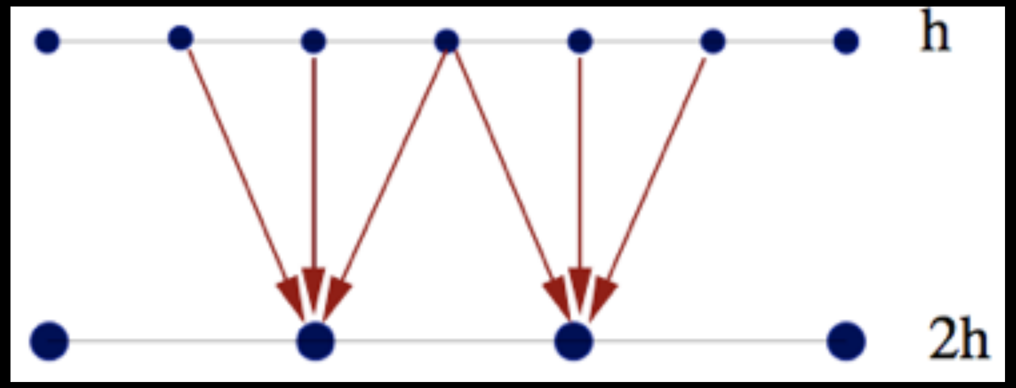
```
void MG::operator()(ColorSpinorField &x, ColorSpinorField &b) {

  if (param.level < param.Nlevel) {
    (*presmoother)(x, b);         // do the pre smoothing

    transfer->R(*r_coarse, *r);         // restrict to the coarse grid

    (*coarse)(*x_coarse, *r_coarse);        // recurse to the next lower level

    transfer->P(*r, *x_coarse);        // prolongate back to this grid

    (*postsmoother)(x,b);         // do the post smoothing

  } else {
    (*coarsesolver)(x, b);  // do the coarse grid solve
  }

}
```

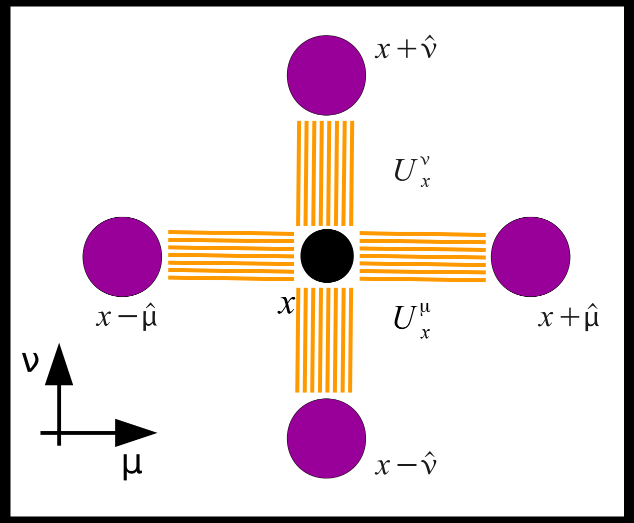
Ingredients for Parallel Adaptive Multigrid

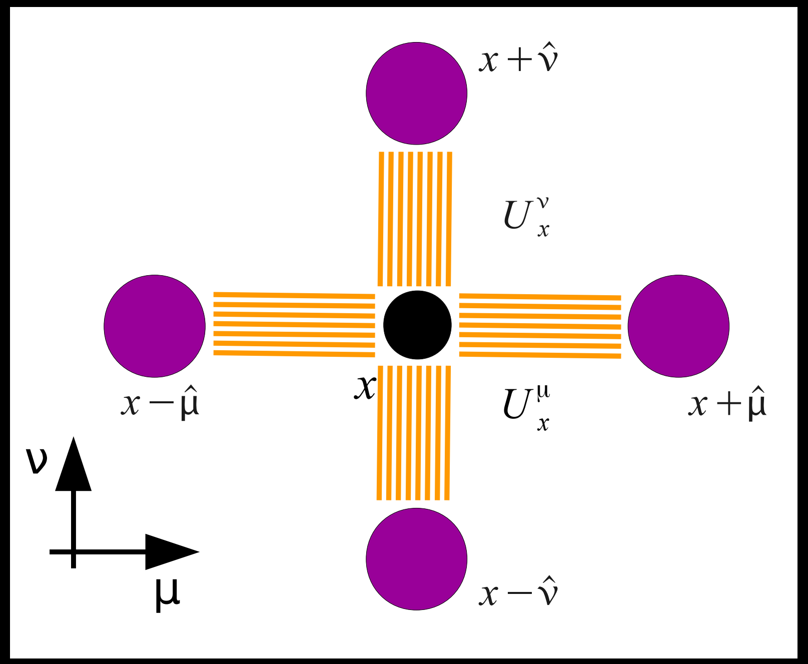
- Prolongation construction (setup)
  - Block orthogonalization of null space vectors
  - Sort null-space vectors into block order (locality)
  - Batched QR decomposition
- Smoothing (relaxation on a given grid)
  - Repurpose the domain-decomposition preconditioner
- Prolongation
  - interpolation from 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
  - direct solve on coarse grid
  - (near) serial algorithm











Parallel Implementation

- Coarse operator looks like a Dirac operator
  - Link matrices have dimension N_v x N_v (e.g., 20 x 20)

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

- Fine vs. Coarse grid parallelization
  - Coarse grid points have limited thread-level parallelism
  - Highly desirable to parallelize over fine grid points where possible
- Parallelization of internal degrees of freedom?
  - Color / Spin degrees of freedom are tightly coupled (dense matrix)
  - Each thread loops over color / spin dimensions
  - Rely on instruction-level parallelism for latency hiding
- Parallel multigrid uses common parallel primitives
  - Reduce, sort, etc.
  - Use CUB parallel primitives for high performance

Writing the same code for two architectures

- Use C++ templates to abstract arch specifics
  - Load/store order, caching modifiers, precision, intrinsics
- CPU and GPU almost identical
  - CPU and GPU kernels call the same functions
  - Index computation (for loop -> thread id)
  - Block reductions (shared memory reduction and / or atomic operations)

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 here:
field order, cache modifiers, textures

platform independent stuff goes here
99% of computation goes here

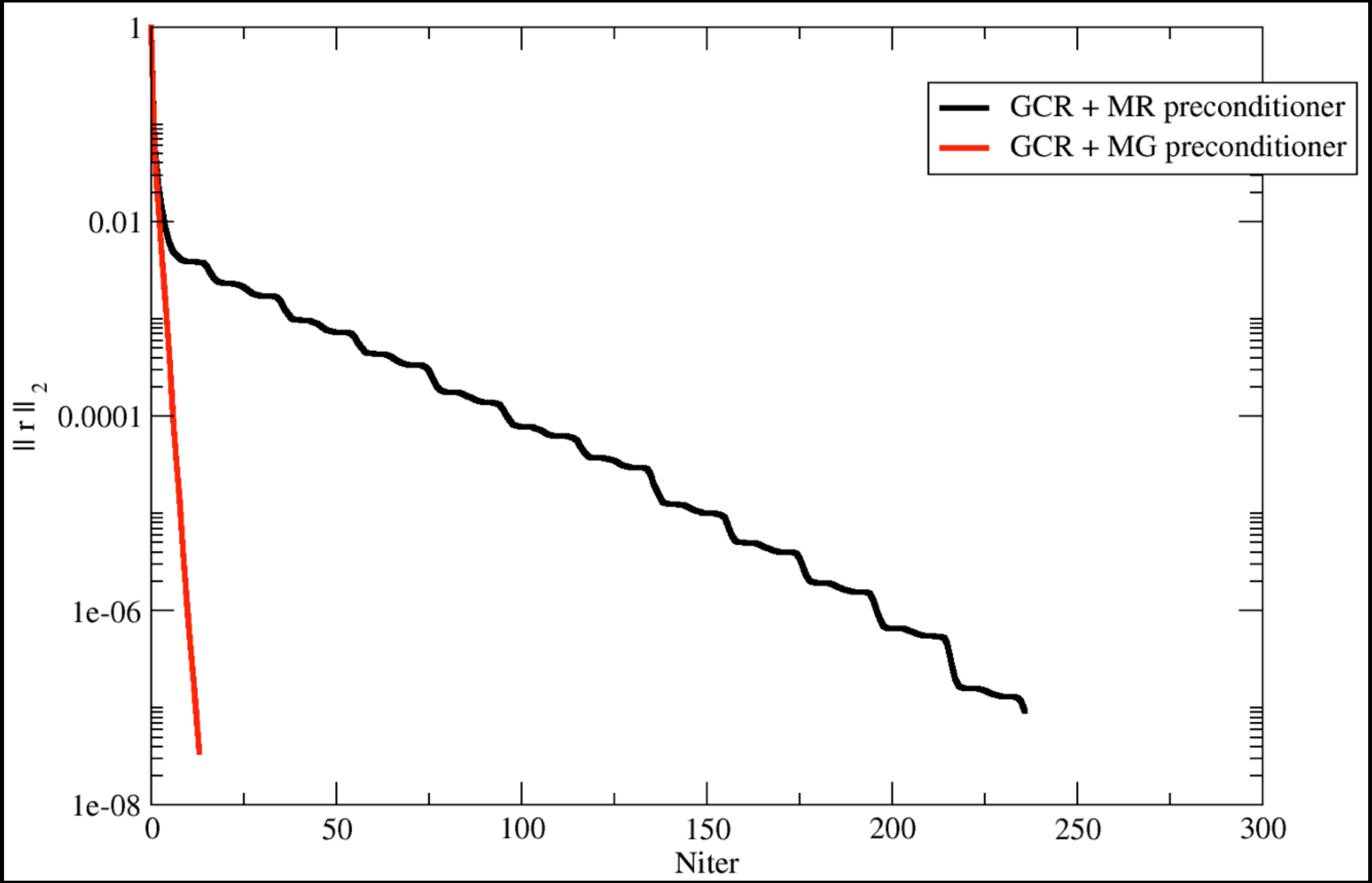
```
template<…> void fooCPU(Arg &arg) {
  arg.sum = 0.0;
#pragma omp for
  for (int x=0; x<size; x++)
    arg.sum += bar<…>(arg, x);
}
```

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);
}
```

CPU

GPU



GCR + MR preconditioner
GCR + MG preconditioner
1
0.01
0.0001
1e-06
1e-08
|| r ||$_2$
0
50
100
150
200
250
300
Niter

Current Status

- Wilson multigrid fully numerically verified
  - Consistent with results from QCDMG (Babich *et al* 2010)
- Framework still slow
  - Small speedup observed versus BiCGstab (~1.5x)
  - Host code not optimized at all (serial)
  - GPU <-> CPU transfers not optimal
  - Optimal code requires heavy degree of templating (compilation and link time is increasingly a problem)
- Early observations
  - Using 16-bit precision for smoothing does not affect convergence
  - Coarse-grid solve can be poorly conditioned thus requiring single precision

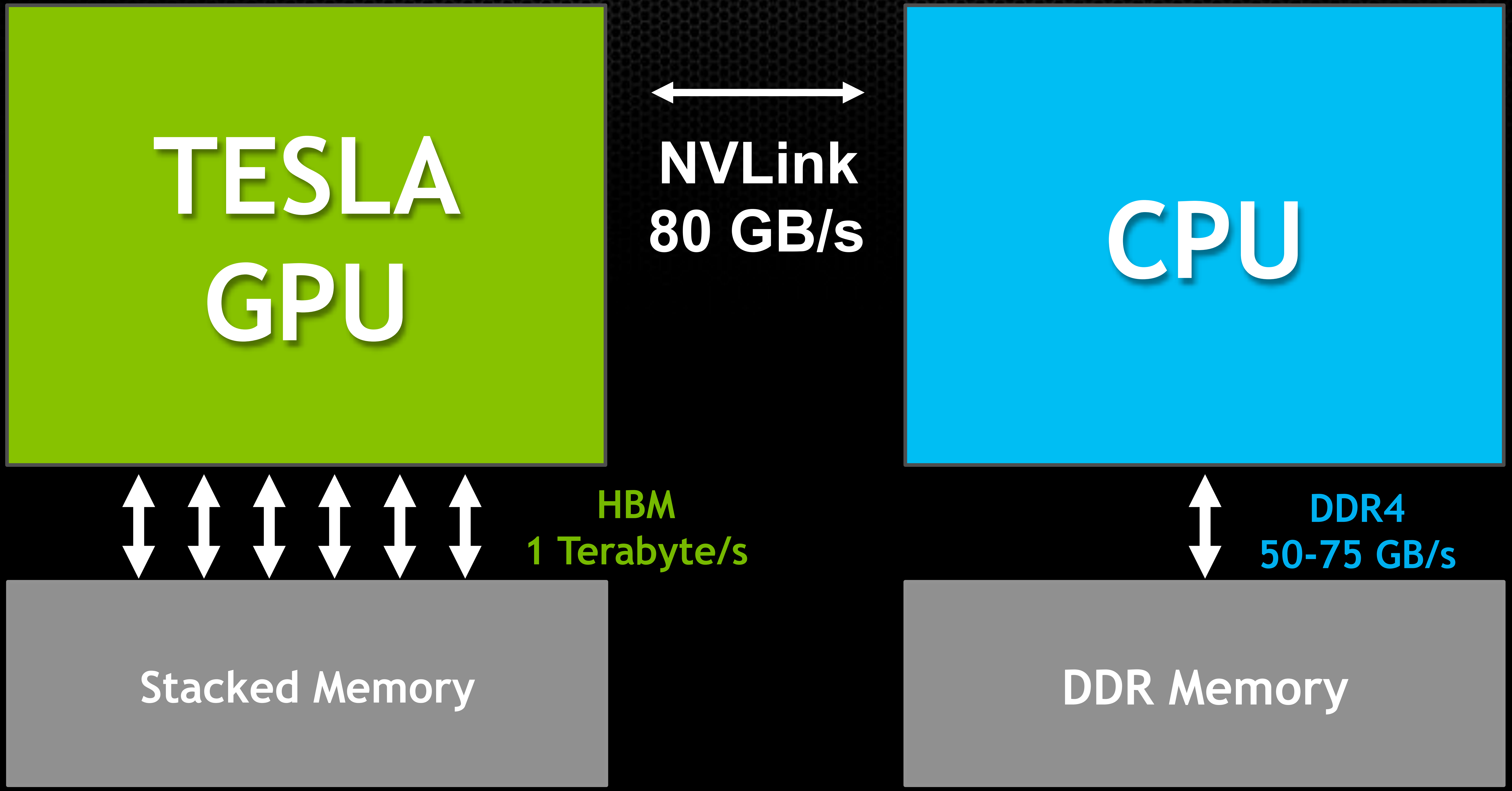
Next Steps

- Optimize
  - E.g., kernel fusion, CPU OpenMP/vectorization
  - read/write directly to/from CPU memory
- Add support for clover coarsening and put into production asap
- Strong scaling
- Algorithm research
  - Precision investigation
  - Coarse-grid solvers (direct vs. indirect)
  - Staggered multigrid
  - Comparison of traditional versus *heterogeneous update*

Hierarchical Algorithm Toolbox

- Exploit closer coupling of precision and algorithm
  - QUDA designed for complete run-time specification of precision at any point in the algorithm
  - Currently supports 64-bit, 32-bit, 16-bit
  - Is 128-bit or 8-bit useful at all for hierarchical algorithms?
- Domain-decomposition (DD) and multigrid
  - DD solvers are effective for high-frequency dampening
  - Overlapping domains likely more important at coarser scales
- Real goal is developing asynchronous solvers for future heterogeneous architectures

Heterogeneous Computing in 2016



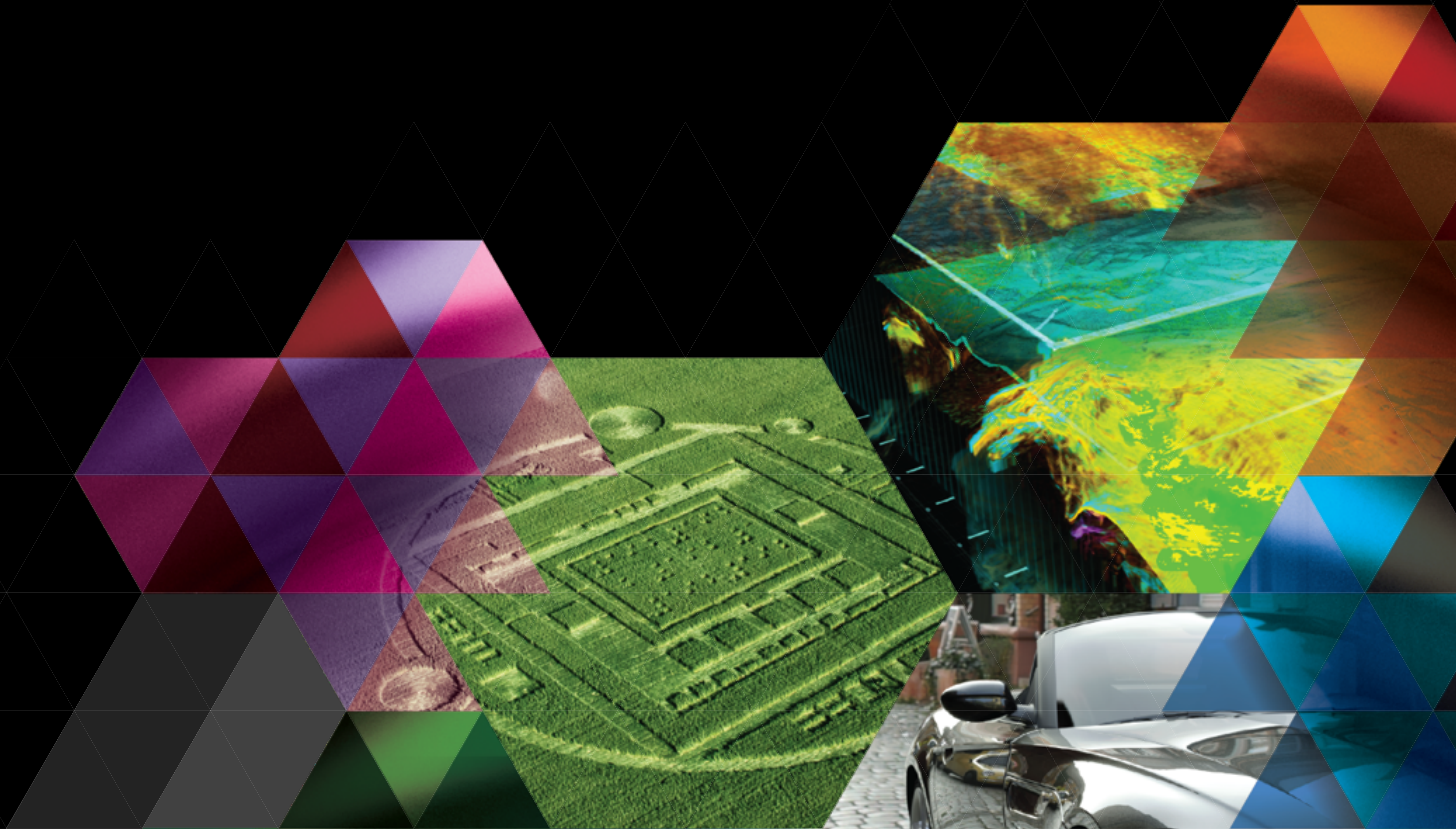
Summary

- Introduction to QUDA library
- Production library for GPU-accelerated LQCD
  - Scalable linear solvers
  - Coverage for most LQCD algorithms
- Current research efforts focused on adaptive multigrid algorithms
  - All of the nitty gritty details worked out
  - Now time for fun
- Hierarchical *and* heterogeneous algorithm research toolbox
  - Aim for scalability *and* optimality
- Lessons today are relevant for Exascale preparation

GPU TECHNOLOGY CONFERENCE

NVIDIA

BACK UP SLIDES



The compilation problem…

- Tightly-coupled variables should be at the register level
- Dynamic indexing cannot be resolved in register variables
  - Array values with indices not known at compile time spill out into global memory (L1 / L2 / DRAM)

```
template <typename ProlongateArg>
 __global__ void prolongate(ProlongateArg arg, int Ncolor, int Nspin) {
   int x = blockIdx.x*blockDim.x + threadIdx.x;
   for (int s=0; s<Nspin; s++) {
     for (int c=0; c<Ncolor; c++) {
     …
     }
   }
 }
```

The compilation problem…

- All *internal* parameters must be known at *compile* time
  - Template over every possible combination O(10,000) combinations
    - Tensor product between different parameters
    - O(10,000 combinations) *per* kernel
  - Only compile necessary kernel at runtime

```
template <typename Arg, int Ncolor, int Nspin>
 __global__ void prolongate(Arg arg) {
   int x = blockIdx.x*blockDim.x + threadIdx.x;
   for (int s=0; s<Nspin; s++) {
     for (int c=0; c<Ncolor; c++) {
    …
     }
   }
 }
```

- JIT support could help here…

QUDA Roadmap

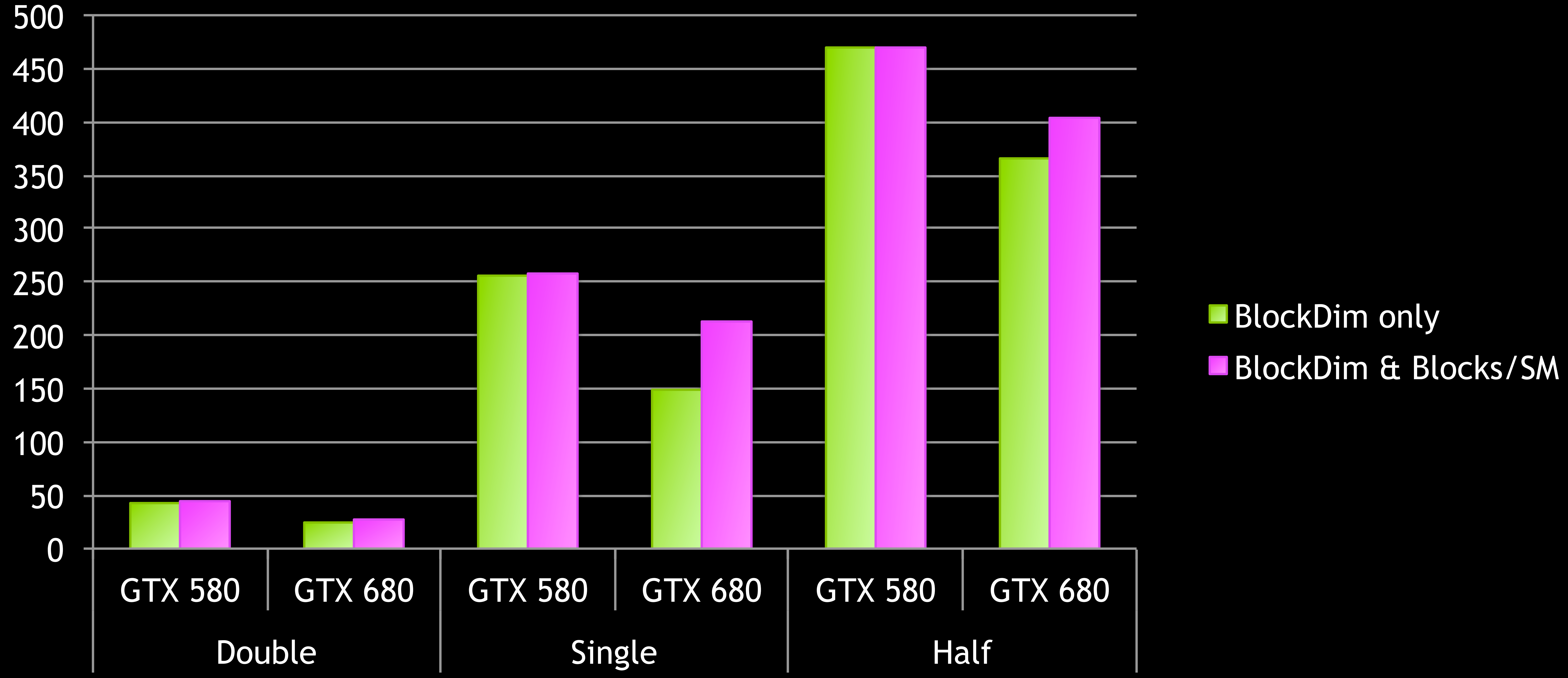
- 0.6.x
  - Long-link computation
  - Reconstruct 9/13 support for HISQ fermions
  - Google test API for stronger unit tests (QUDA now in CUDA regression suite)
- 0.7.0
  - Twisted-clover and Mobius fermions
  - EigCG solver
  - Better strong scaling
  - Stabilized mixed-precision CG
  - Clover field computation, inversion and force terms
- 0.8.0
  - Adaptive multigrid
  - Optimized dslash (essentially untouched since 2009)
  - s-step solvers
- Taking requests (and more importantly volunteers!)

Run-time autotuning

- Motivation:
  - —Kernel performance (but not output) strongly dependent on launch parameters:
    - gridDim (trading off with work per thread), blockDim
    - blocks/SM (controlled by over-allocating shared memory)
- Design objectives:
  - —Tune launch parameters for all performance-critical kernels at run-time as needed (on first launch).
  - —Cache optimal parameters in memory between launches.
  - —Optionally cache parameters to disk between runs.
  - —Preserve correctness.

Auto-tuned “warp-throttling”

- Motivation: Increase reuse in limited L2 cache.



Run-time autotuning: Implementation

- Parameters stored in a global cache:

```
static std::map<TuneKey, TuneParam> tunecache;
```

- TuneKey is a struct of strings specifying the kernel name, lattice volume, etc.
- TuneParam is a struct specifying the tune blockDim, gridDim, etc.
- Kernels get wrapped in a child class of Tunable (next slide)
- tuneLaunch() searches the cache and tunes if not found:

```
TuneParam tuneLaunch(Tunable &tunable, QudaTune enabled,
QudaVerbosity verbosity);
```

Run-time autotuning: Usage

- Before:

```
myKernelWrapper(a, b, c);
```

- After:

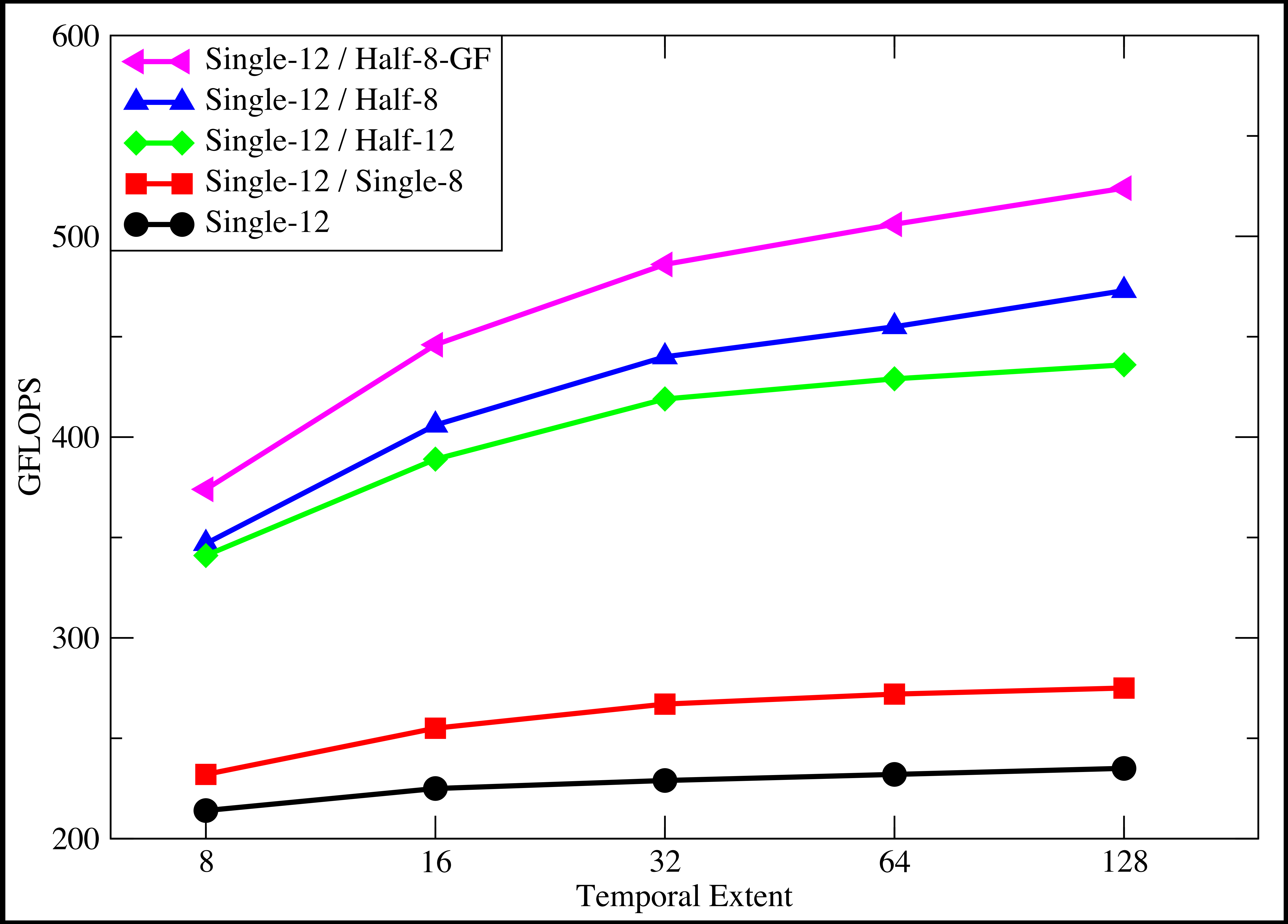
```
MyKernelWrapper *k = new MyKernelWrapper(a, b, c);
k->apply();  // <-- automatically tunes if necessary
```

- Here MyKernelWrapper inherits from Tunable and optionally overloads various virtual member functions (next slide).
- Wrapping related kernels in a class hierarchy is often useful anyway, independent of tuning.

Virtual member functions of Tunable

- Invoke the kernel (tuning if necessary):
  - apply()
- Save and restore state before/after tuning:
  - preTune(), postTune()
- Advance to next set of trial parameters in the tuning:
  - advanceGridDim(), advanceBlockDim(), advanceSharedBytes()
  - advanceTuneParam() // simply calls the above by default
- Performance reporting
  - flops(), bytes(), perfString()
- etc.

Kepler Wilson-Solver Performance



Wilson CG
K20X performance
V = 24^3xT

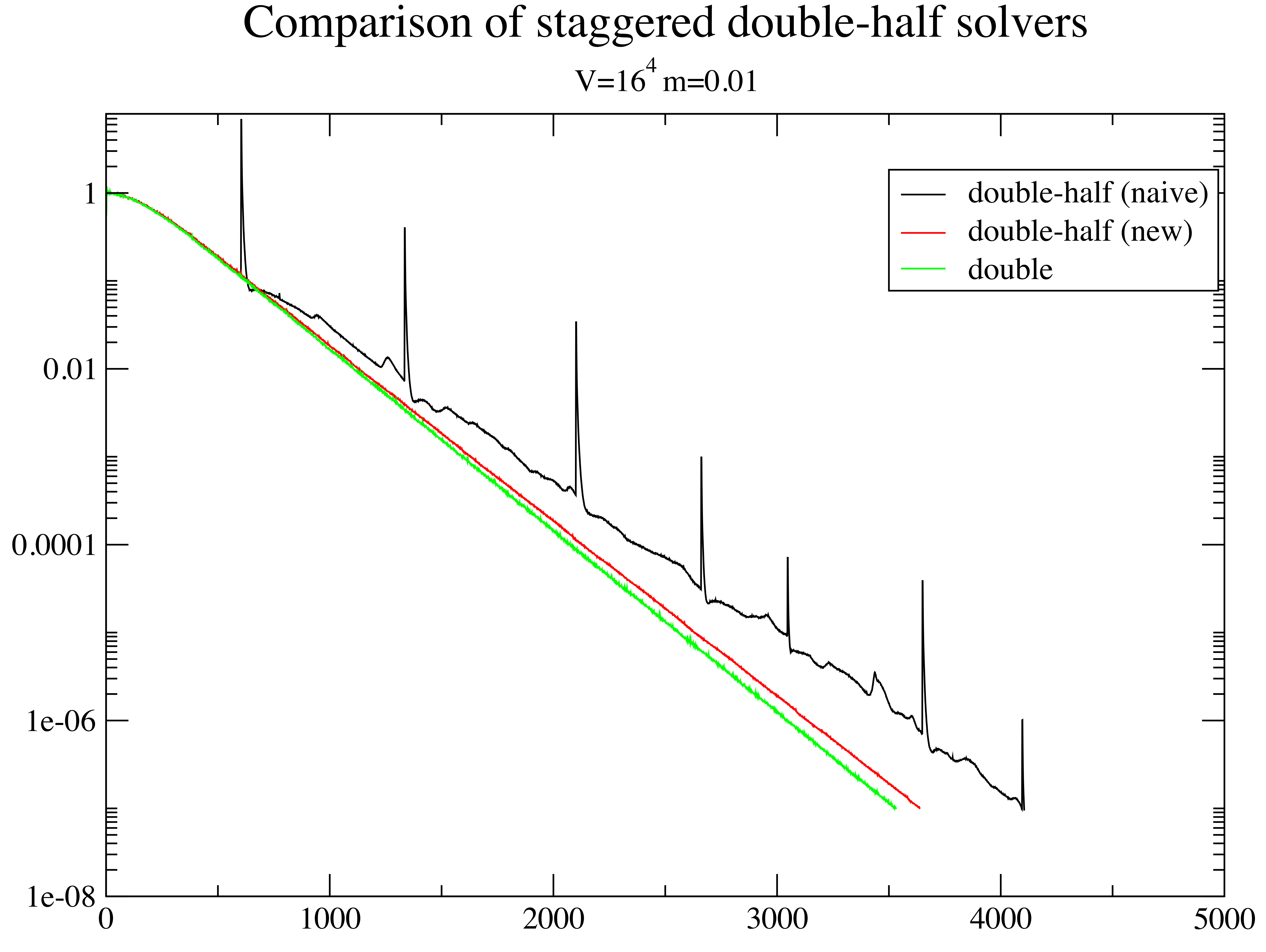
(Stable) Mixed-precision CG

- CG convergence relies on gradient vector being orthogonal to residual
  - Re-project when injecting new residual
- α chosen to minimize $|e|_A$
  - True irrespective of precision of p, q, r
  - Solution correction is truncated if we keep low precision x
  - Always keep solution vector in high precision
- β computation relies on $(r_i,r_j) = |r_i|^2 \delta_{ij}$
  - Not true in finite precision
  - Polak-Ribière formula is equivalent and self-stabilizing through local orthogonality

$$\beta_k = \alpha(\alpha(q_k,q_k) - (p_k,q_k))/(\mathbf{r}_{k-1},\mathbf{r}_{k-1})$$

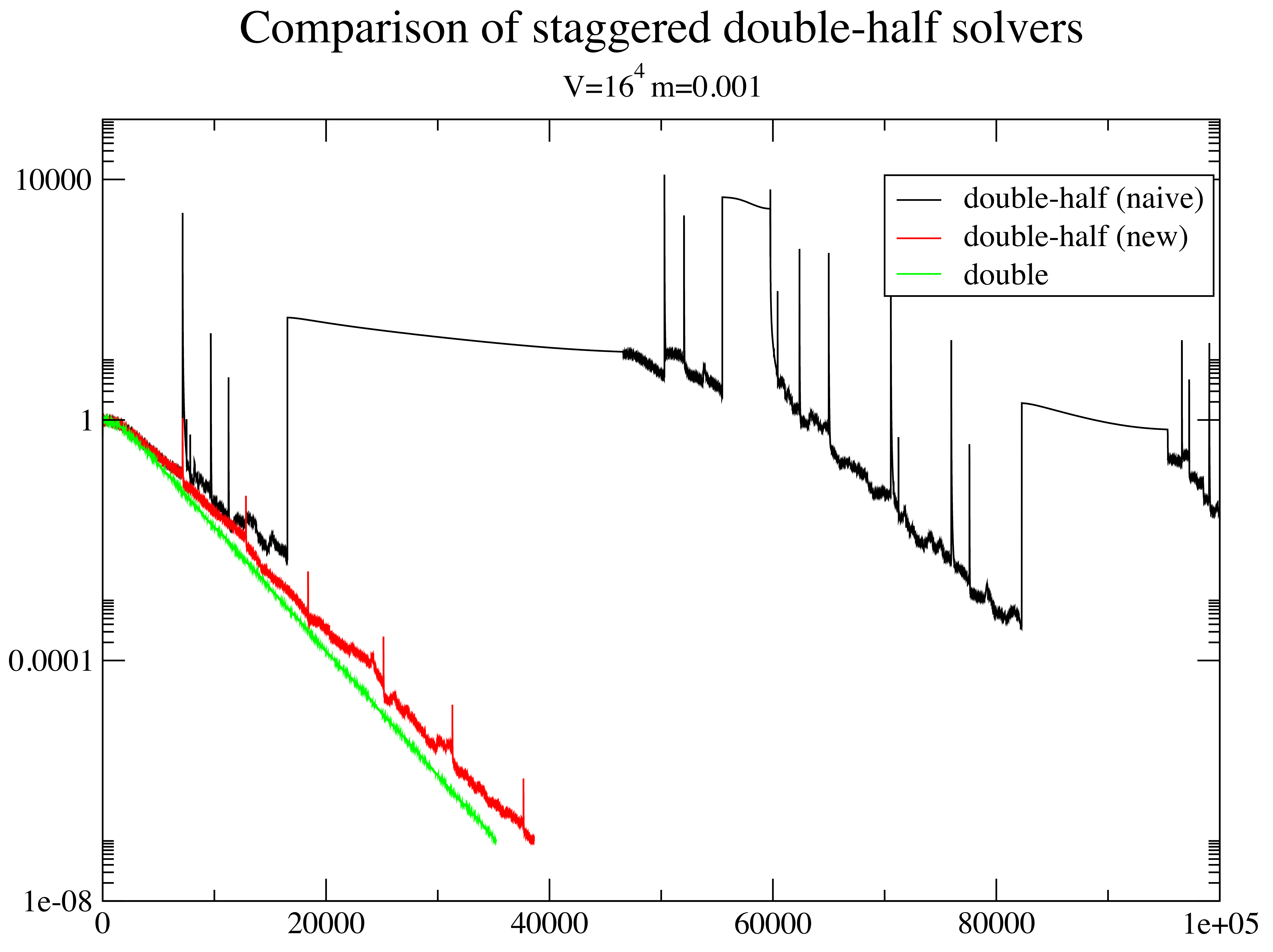
- Further improvement possible
  - Mining the literature on fault-tolerant solvers…

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



Comparison of staggered double-half solvers

$V=16^4$ m=0.01

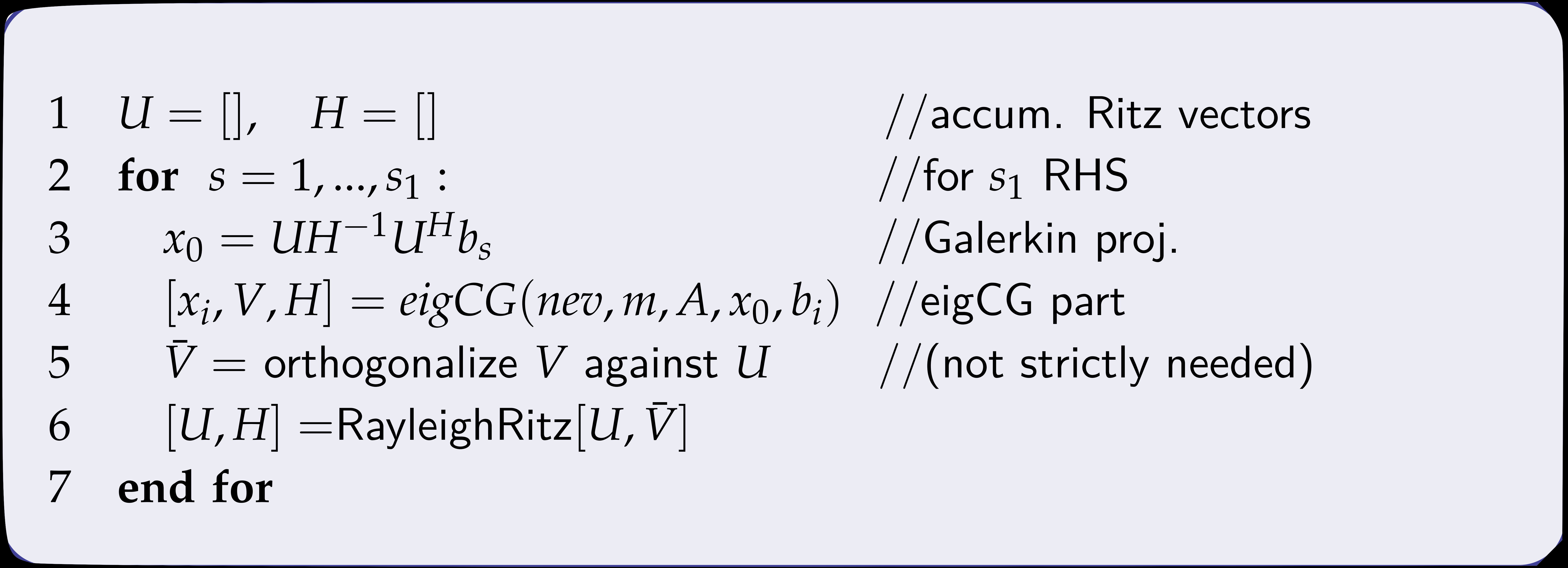


Comparison of staggered double-half solvers

V=16^4 m=0.001

Deflation Algorithms in QUDA

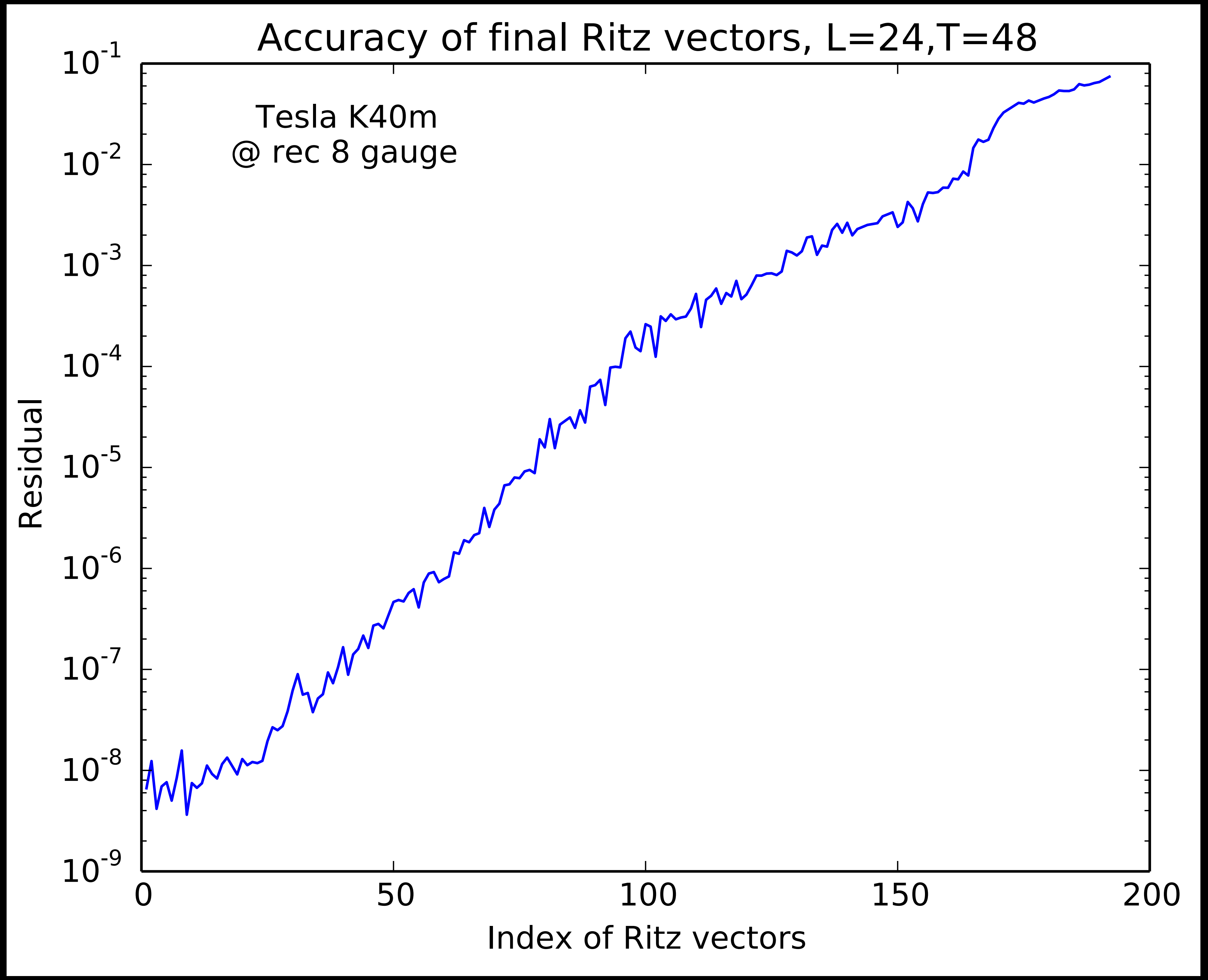
- EigCG implemented in QUDA (Alexei Strelchenko)



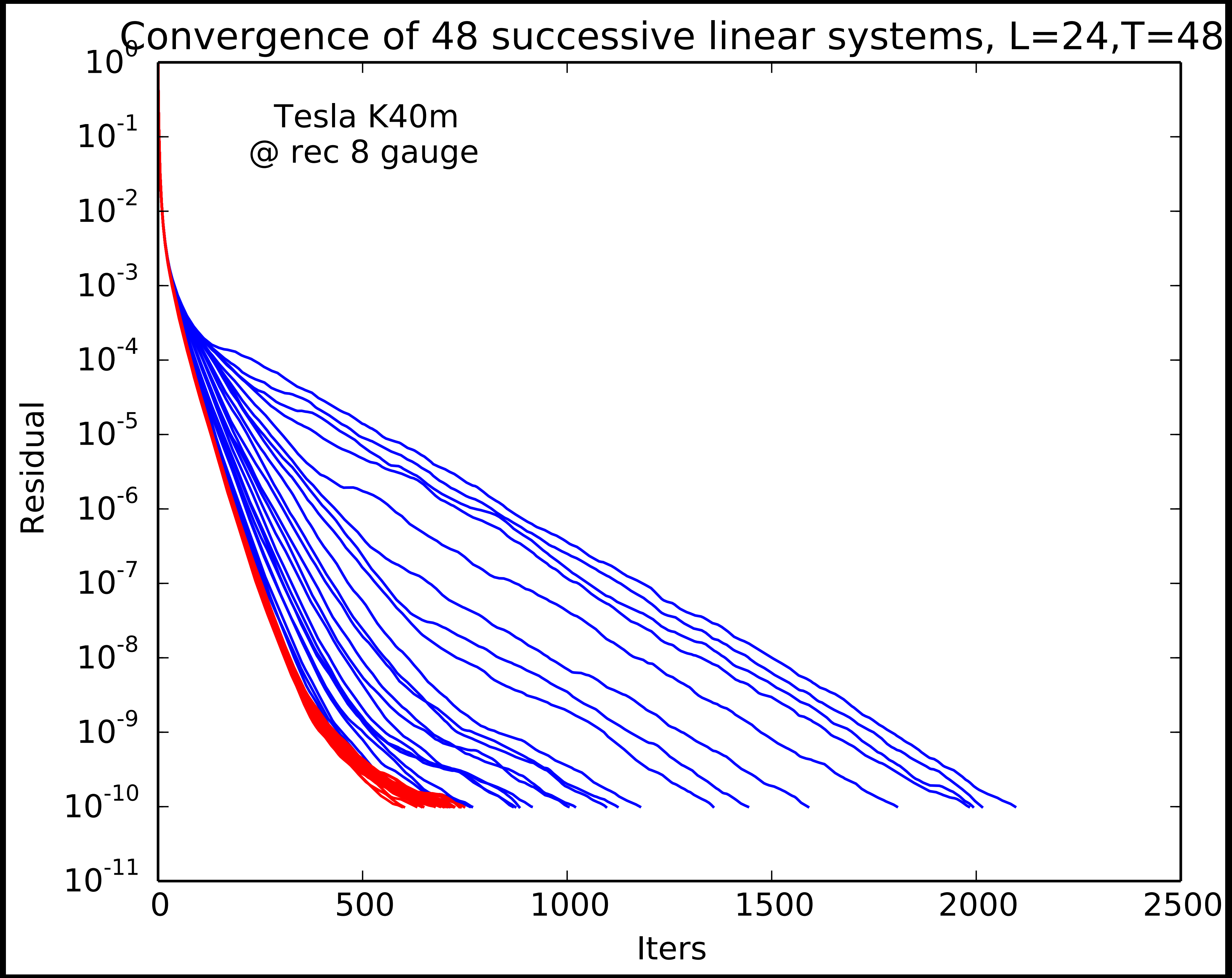
```
$U = [\,], \quad H = [\,]$                          //accum. Ritz vectors
for $s = 1, ..., s_1$ :                             //for $s_1$ RHS
    $x_0 = UH^{-1}U^H b_s$                          //Galerkin proj.
    $[x_i, V, H] = eigCG(nev, m, A, x_0, b_i)$      //eigCG part
    $\bar{V}$ = orthogonalize $V$ against $U$       //(not strictly needed)
    $[U, H]$ =RayleighRitz$[U, \bar{V}]$
end for
```

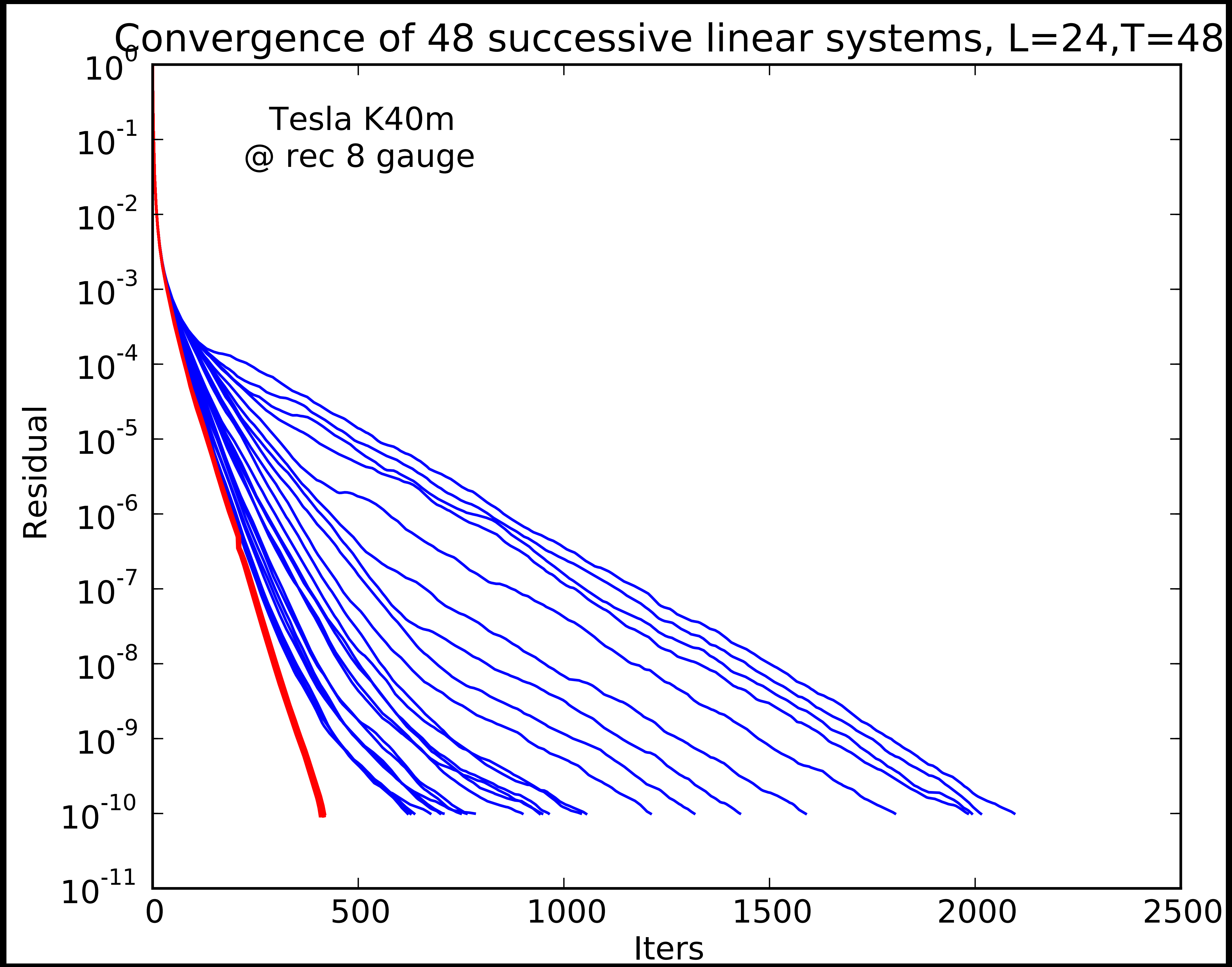
Deflation Algorithms in QUDA

- Use MAGMA library for required LAPACK functionality
- Memory not a problem
  - EigCG only works on subsets
  - Cache full set on CPU
- Extensible eigenvector solver framework for future solvers
  - EigBiCG
  - GMRES-DR
  - etc.



Deflation Algorithms in QUDA





degenerate twisted mass 24^3x48, κ = 0.161231, μ = 0.0085

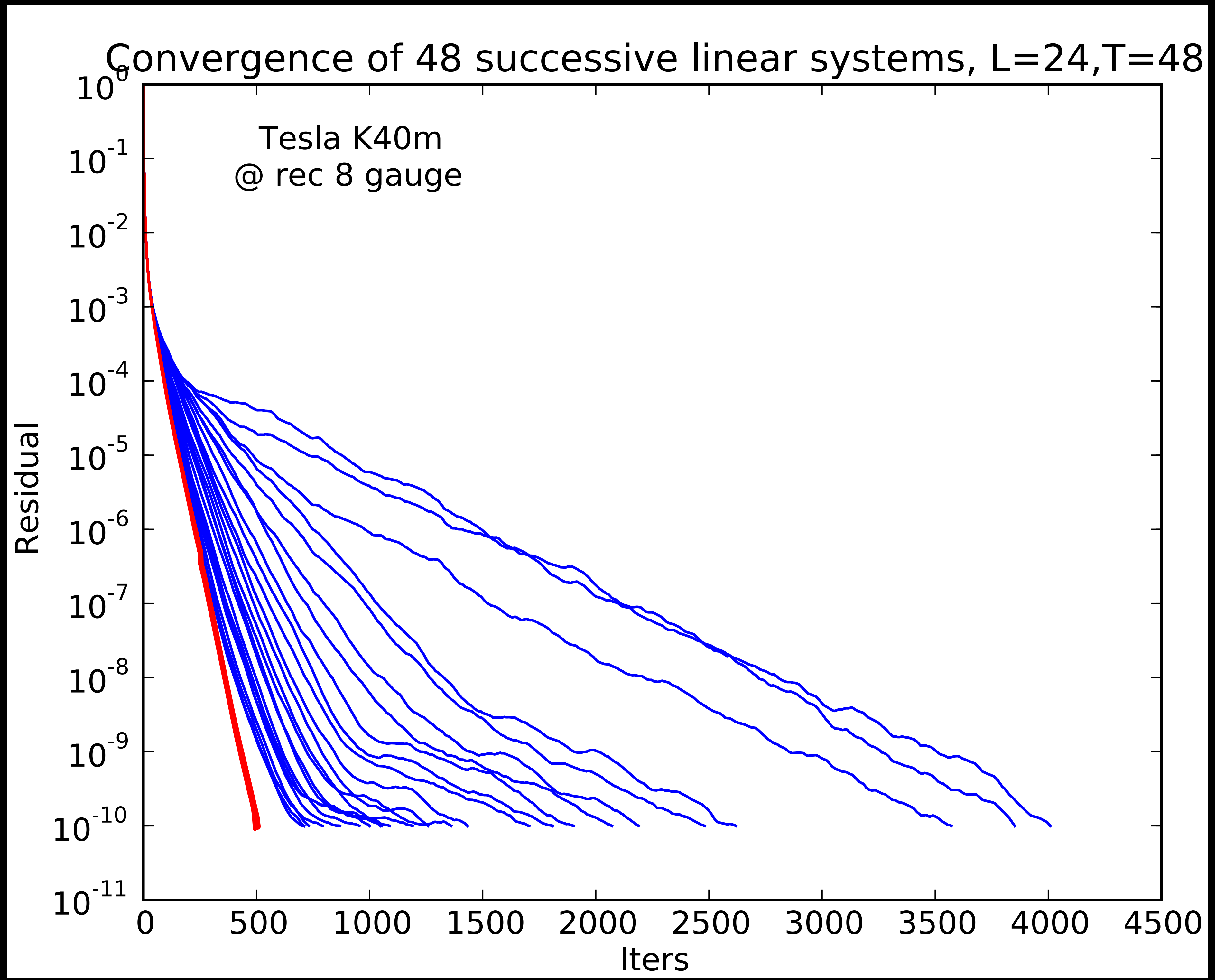
Mixed-Precision Deflation Algorithms

- Mixed-precision CG
  - Precision-truncated residual is ignorant of low modes
  - This can causes breakdown in CG recurrence relations
  - Ameliorated by using reliable updates (and other methods)
- EigCG phase seems to need double precision
  - Loss of precision in finding Ritz vectors results in very poor eigenvector set
- Deflated CG is hugely stabilized once low modes projected out
  - double-half solvers now completely stable at light quark mass
  - e.g. degenerate twisted mass 24^3x48, κ = 0.161231, μ = 0.0040

Non-deflated double-single CG: 15 sec
Non-deflated double-half CG: (does not converge)
InitCG double-single initCG: 2.42 sec
InitCG double-half initCG: 1.84 sec
Achieved speedup ~8X for initCG (combination of algorithm and precision)

Mixed Precision Deflation Algorithms

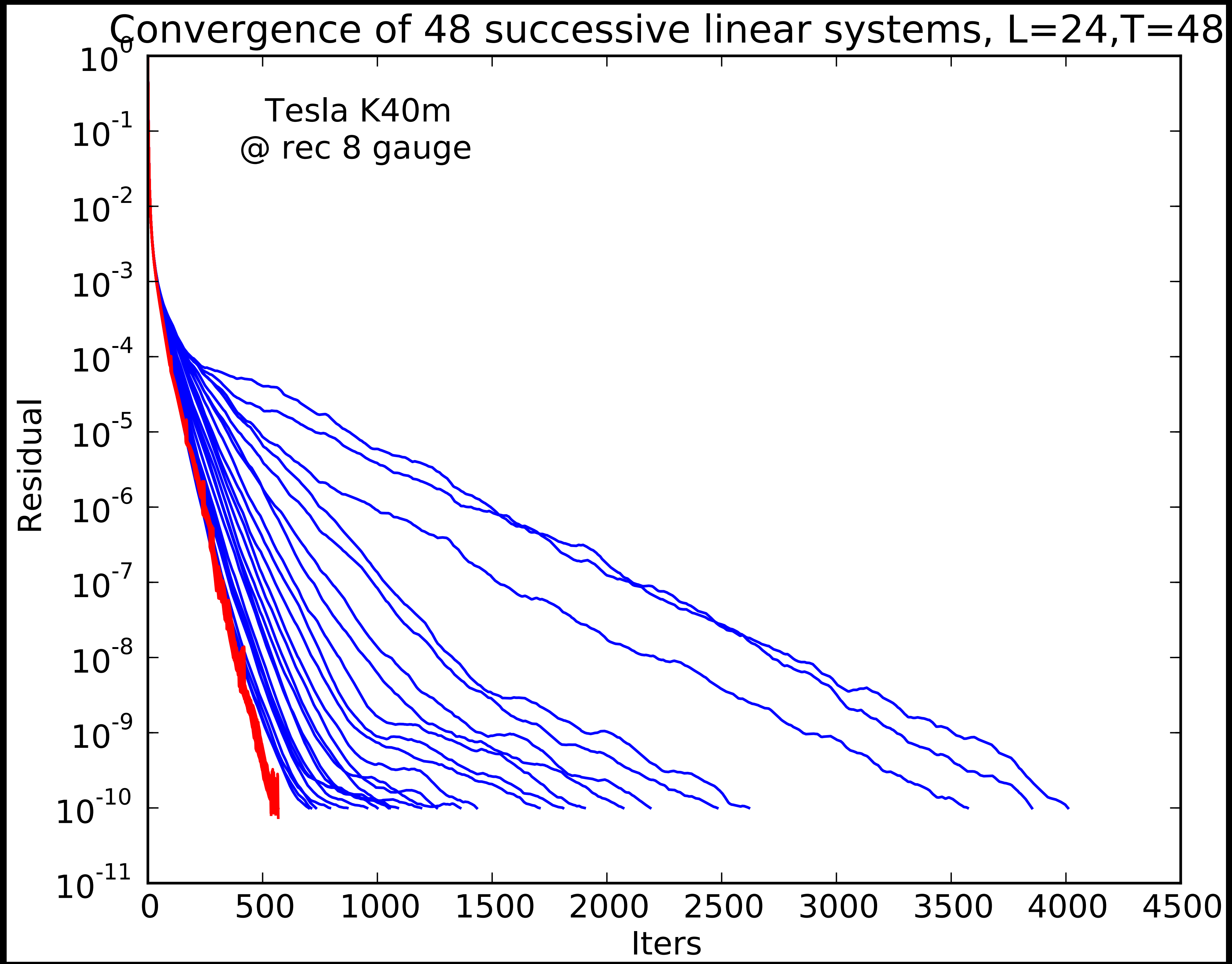
- EigCG seems to need high precision (double)



Convergence of 48 successive linear systems, L=24,T=48

Tesla K40m
@ rec 8 gauge

double-single



Convergence of 48 successive linear systems, L=24,T=48

Tesla K40m
@ rec 8 gauge

double-half

degenerate twisted mass 24^3x48, κ = 0.161231, μ = 0.0040

The Future of GPUs

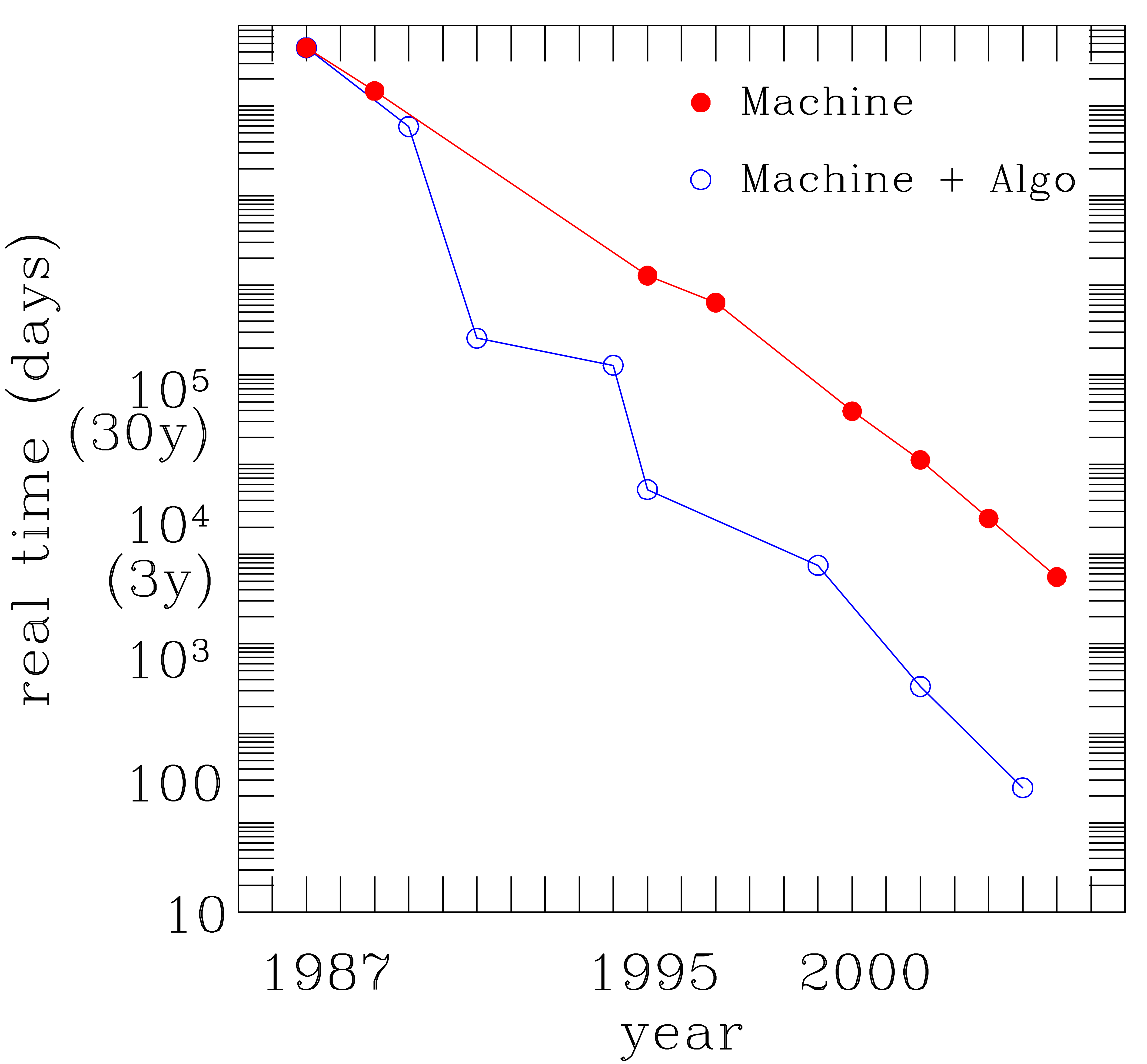
- GPUs viable because of multi $B gaming market
- Coming to an end anytime soon?

CLOSE TO NEW RECORD
500 points to beat KMagnus

Reach The Factory Roof

The Future of GPUs

- Each photo-realistic image takes ~2 seconds
- Photo-realistic imagery requires ~200x faster
- Add physics
  - Rigid body mechanics
  - Computational fluid dynamics (smoke, water, wind)
  - Hair
  - etc.
- GPUs aren’t slowing down anytime soon


● Machine
○ Machine + Algo
real time (days)
10^5
(30y)
10^4
(3y)
10^3
100
10
1987
1995
2000
year