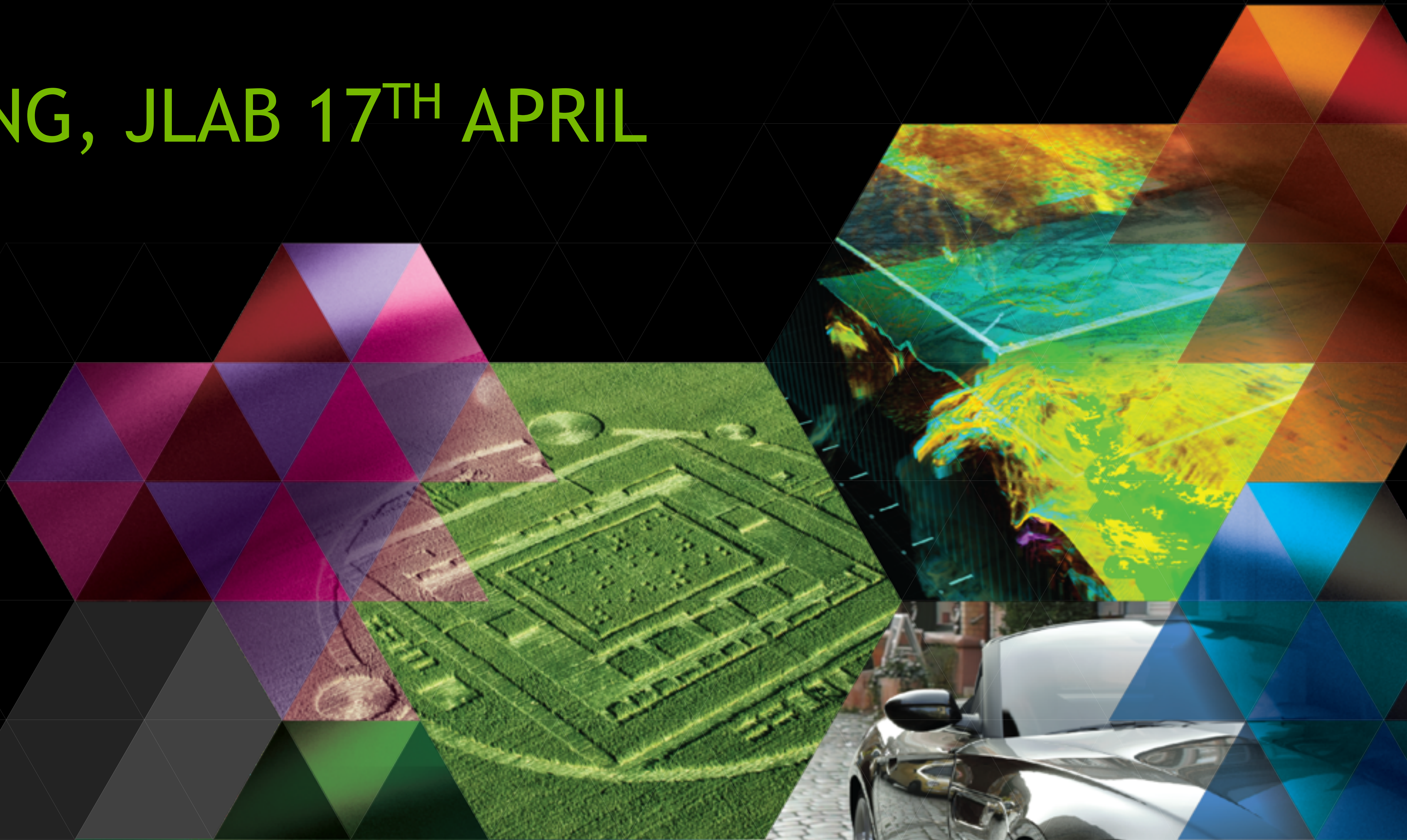
STATE OF QUDA

USQCD SOFTWARE MEETING, JLAB 17TH APRIL

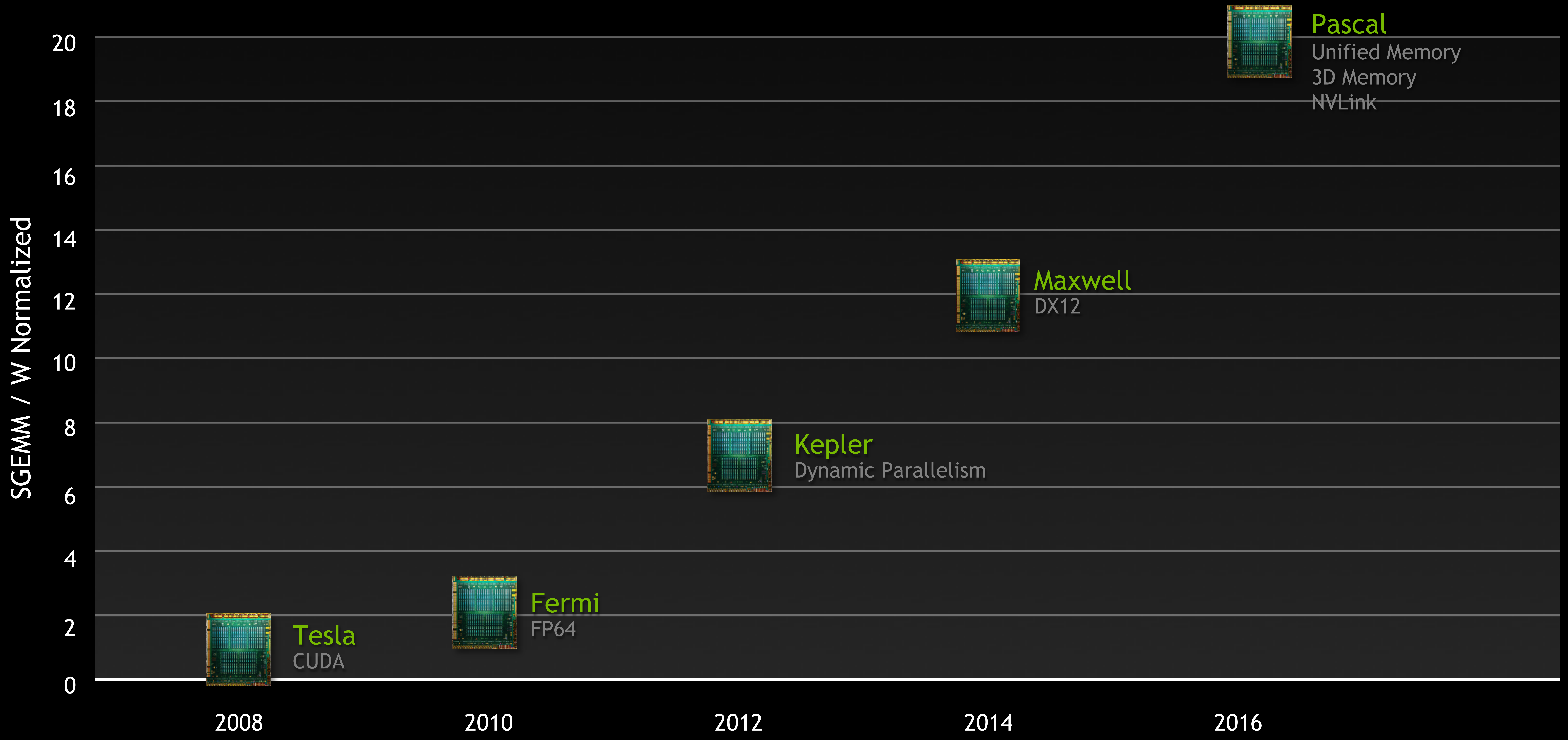


M Clark
NVIDIA

Contents

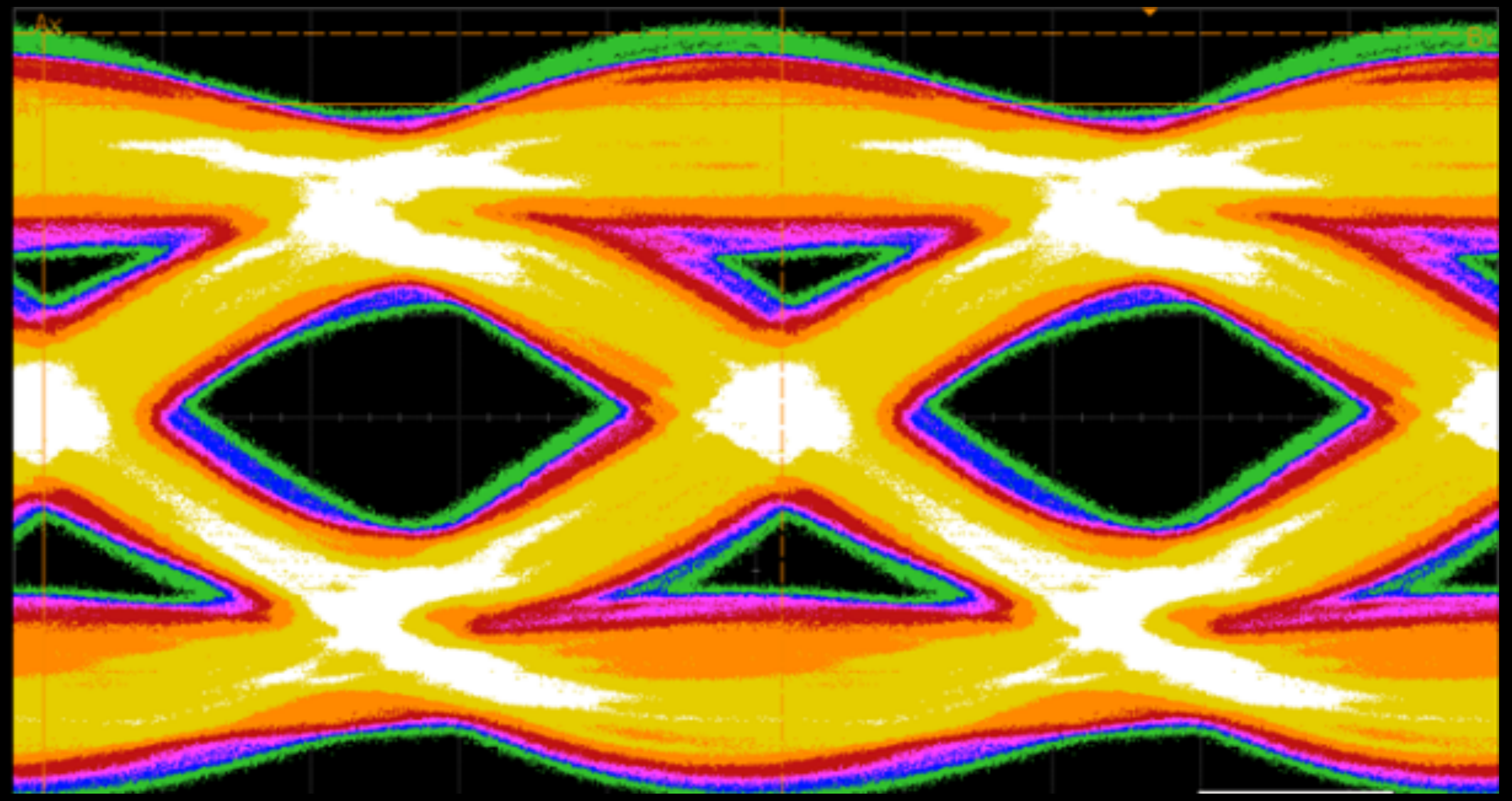
- NVIDIA update
- QUDA current Status
- Strong scaling
- Domain decomposition
- Multigrid

CUDA Roadmap

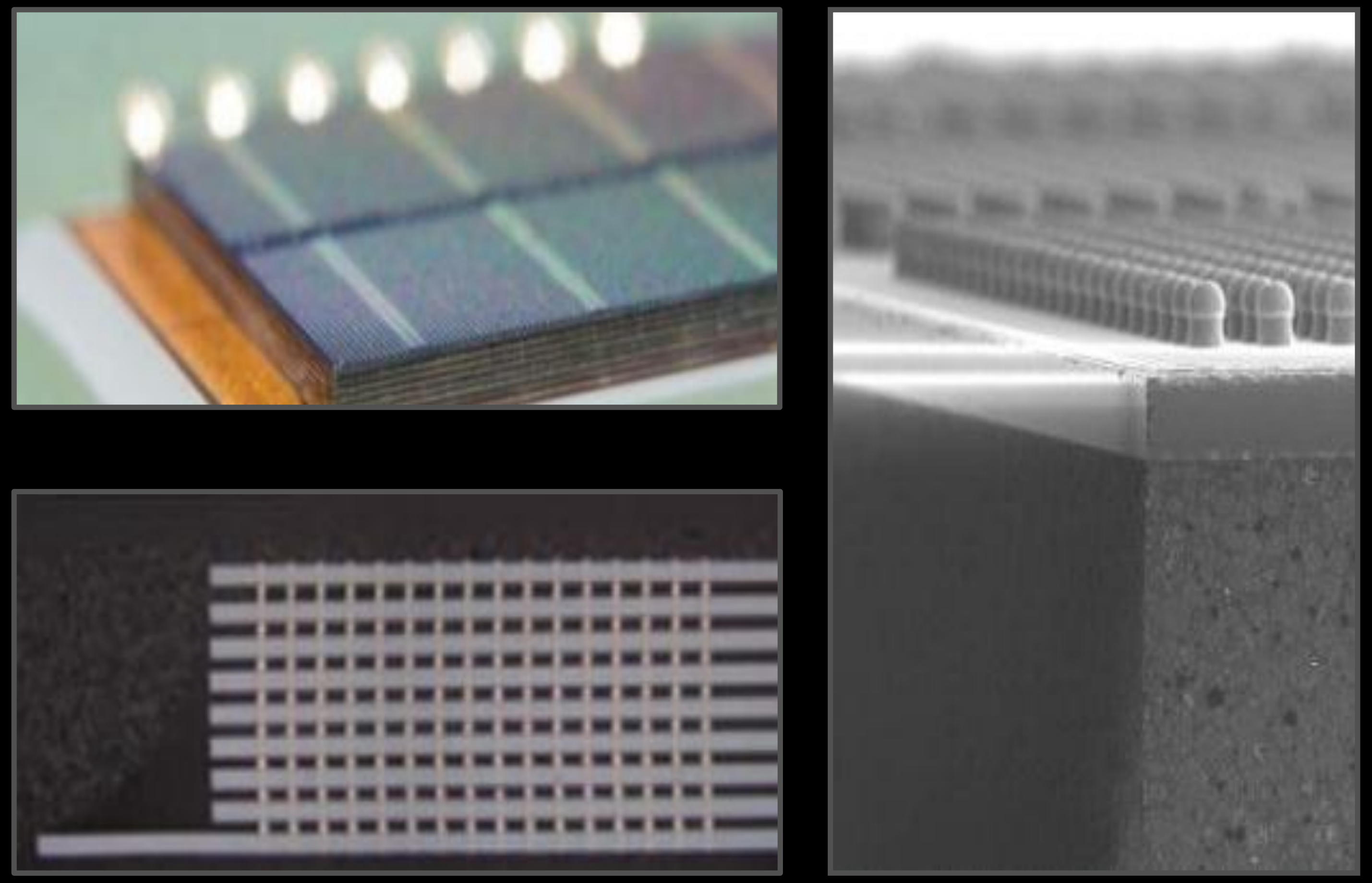


Introducing NVLINK and Stacked Memory

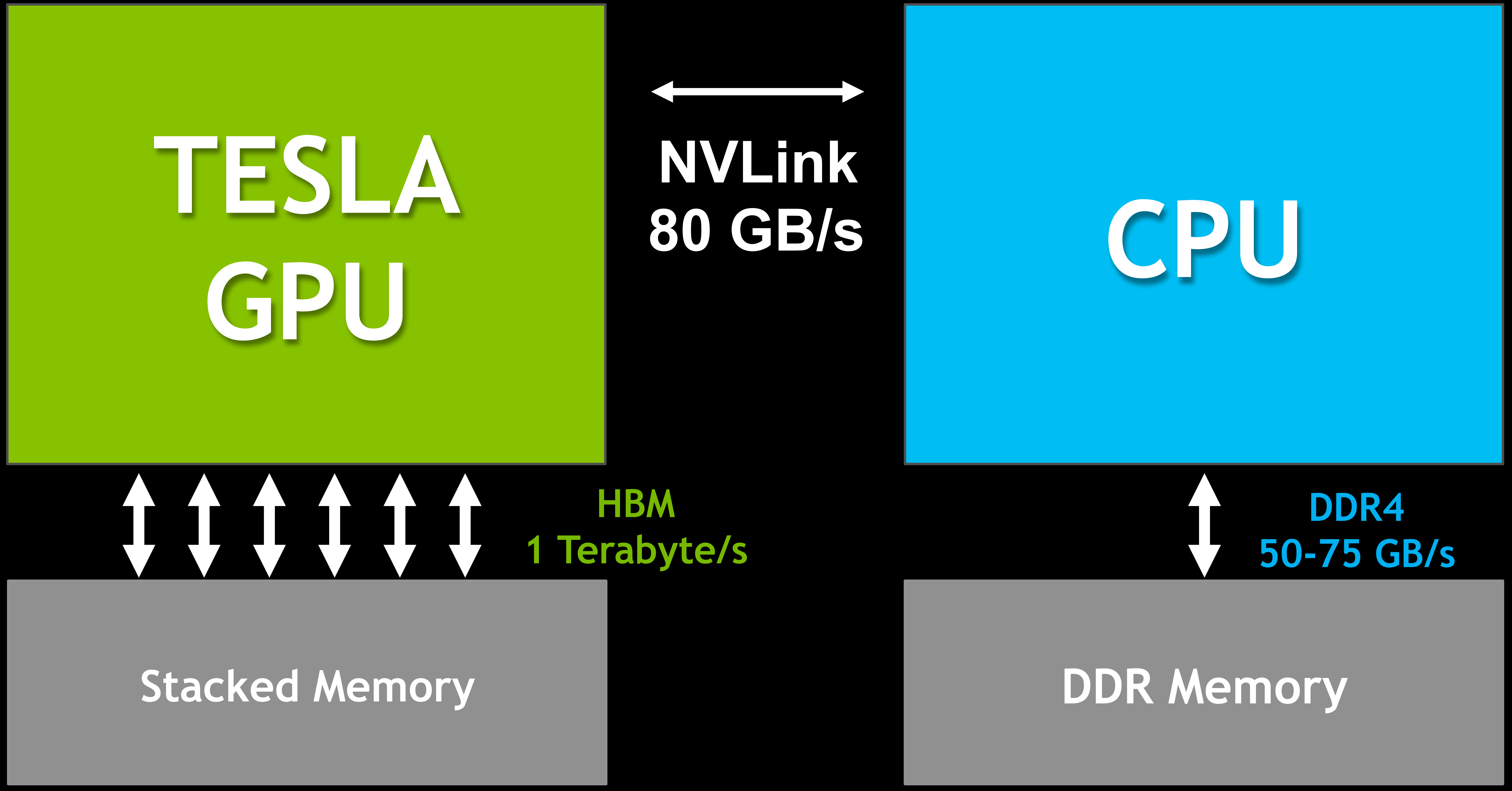
- NVLINK
  - GPU high speed interconnect
  - 80-200 GB/s
  - Planned support for POWER CPUs



- Stacked Memory
  - 4x Higher Bandwidth (~1 TB/s)
  - 3x Larger Capacity
  - 4x More Energy Efficient per bit



NVLINK Enables Data Transfers at the Speed of CPU Memory



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, TIFR, 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
  - – Eigenvector solvers new!
  - – Multigrid solvers for optimal convergence new!

QUDA Community

- 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)
- Gregory Petropoulos (Boulder)
- Claudio Rebbi (Boston University)
- Guochun Shi (NCSA -> Google)
- Alexei Strelchenko (FNAL)
- Alejandro Vaquero (Cyprus Institute)
- Frank Winter (Jlab)
- Yibo Yang (IHEP)

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!)

Linear Solvers

- QUDA now has a wide choice of linear solvers
  - CGNE / CGNR
  - BiCGstab
  - GCR
  - Minimum Residual
  - Steepest Descent
  - PCG
- 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

Mixed-precision solvers

- QUDA has had mixed-precision from the get go
- Almost a free lunch where it works well
  - Mixed double-single and double-half BiCGstab (wilson / clover)
  - 2 Tflops sustained in workstation (4 GPUs)
- Did not work well for CG (staggered / twisted mass / dwf)
  - double-single has increased iteration count
  - double-half non convergent
- Why is this?
  - CG recurrence relations much more intolerant
  - BiCGstab noisy as hell anyway
- Need to make CG more robust
  - Make double-half work
  - Less polishing in mixed-precision multi-shift solver

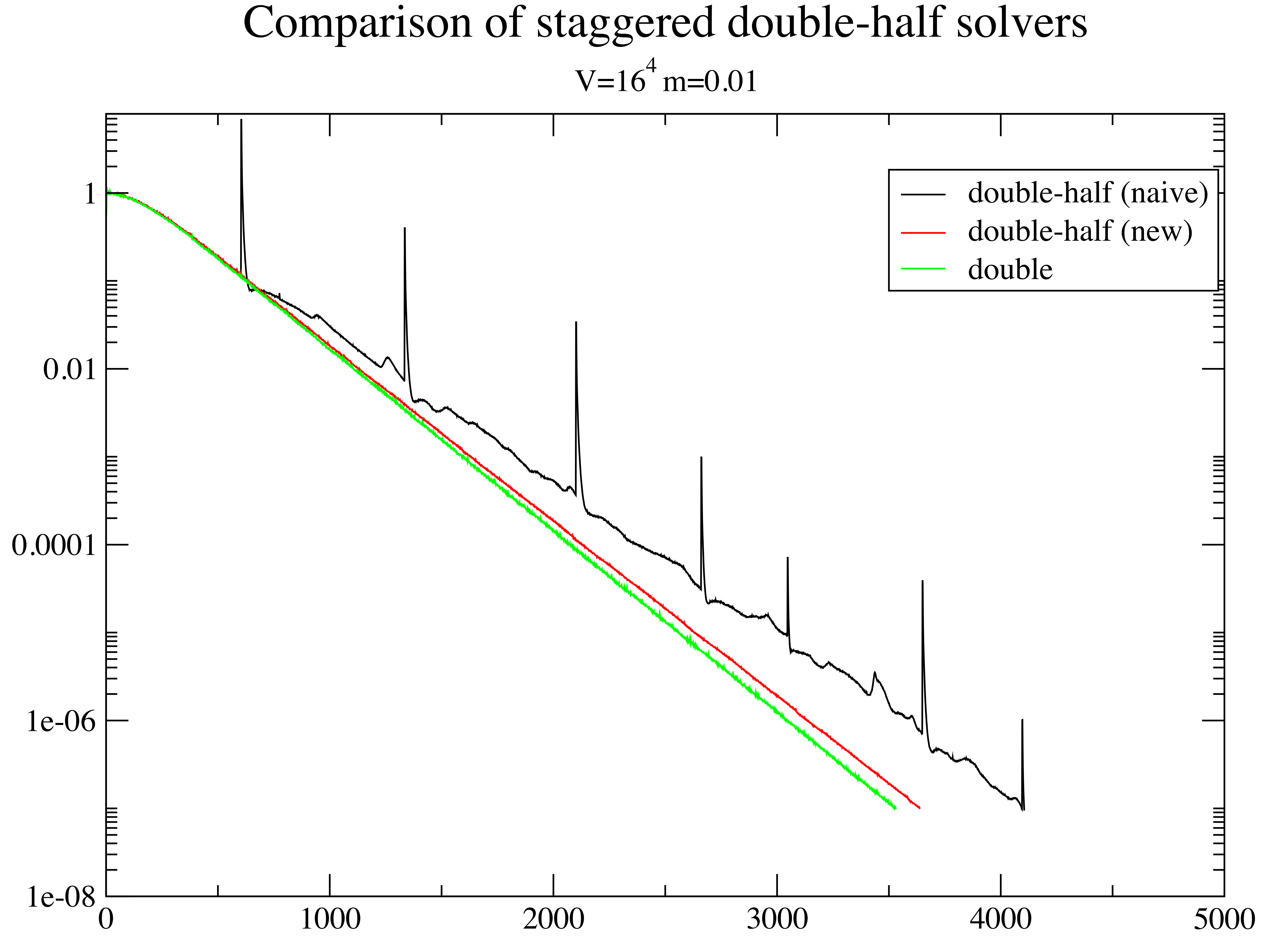
(Stable) Mixed-precision CG

- 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 x in low precision
  - 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

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

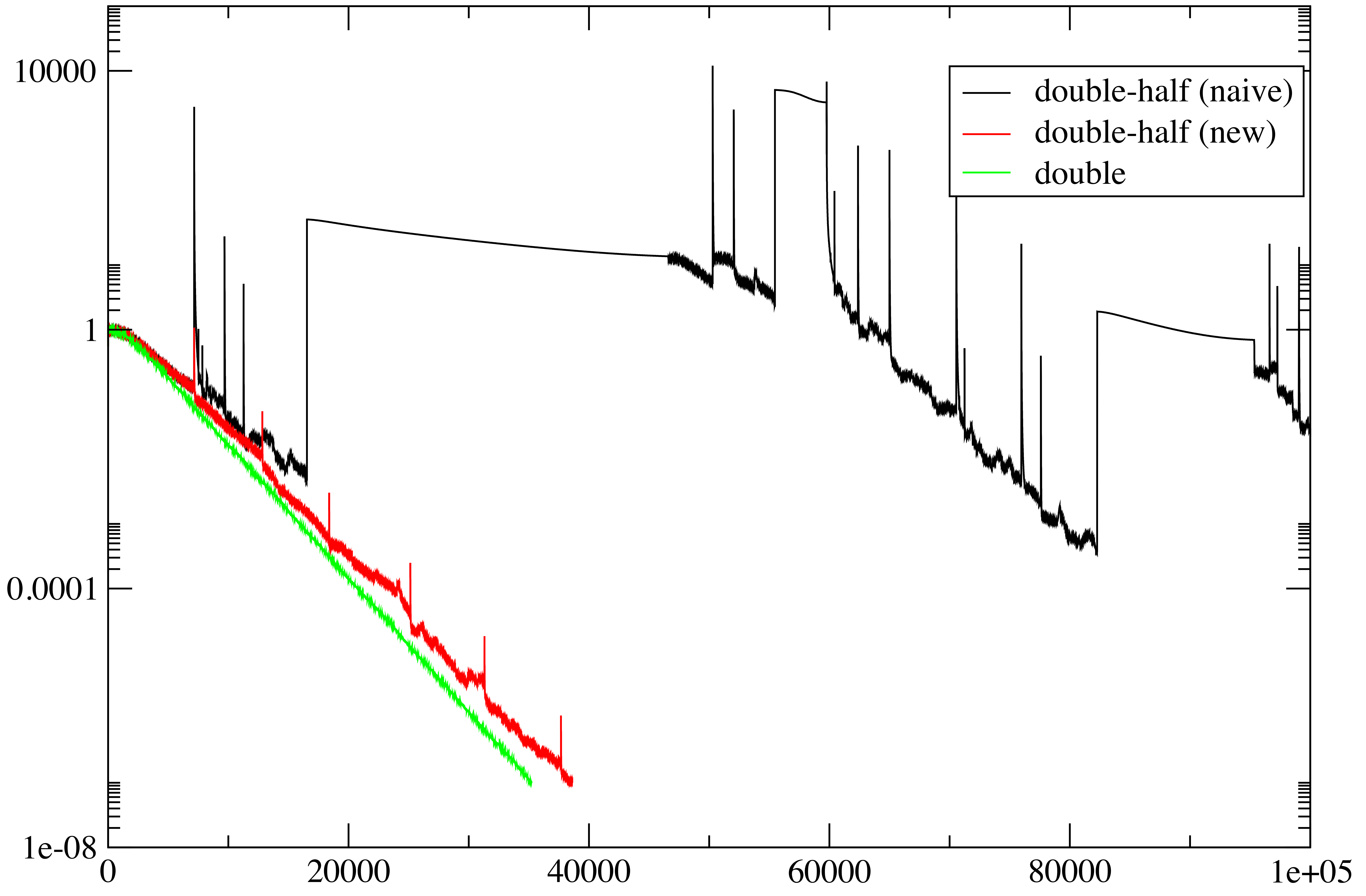
- Further improvement possible
  - Mining the literature on fault-tolerant solvers…

```
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
1
0.01
0.0001
1e-06
1e-08
0
1000
2000
3000
4000
5000
double-half (naive)
double-half (new)
double

Comparison of staggered double-half solvers

$V=16^4$ m=0.001



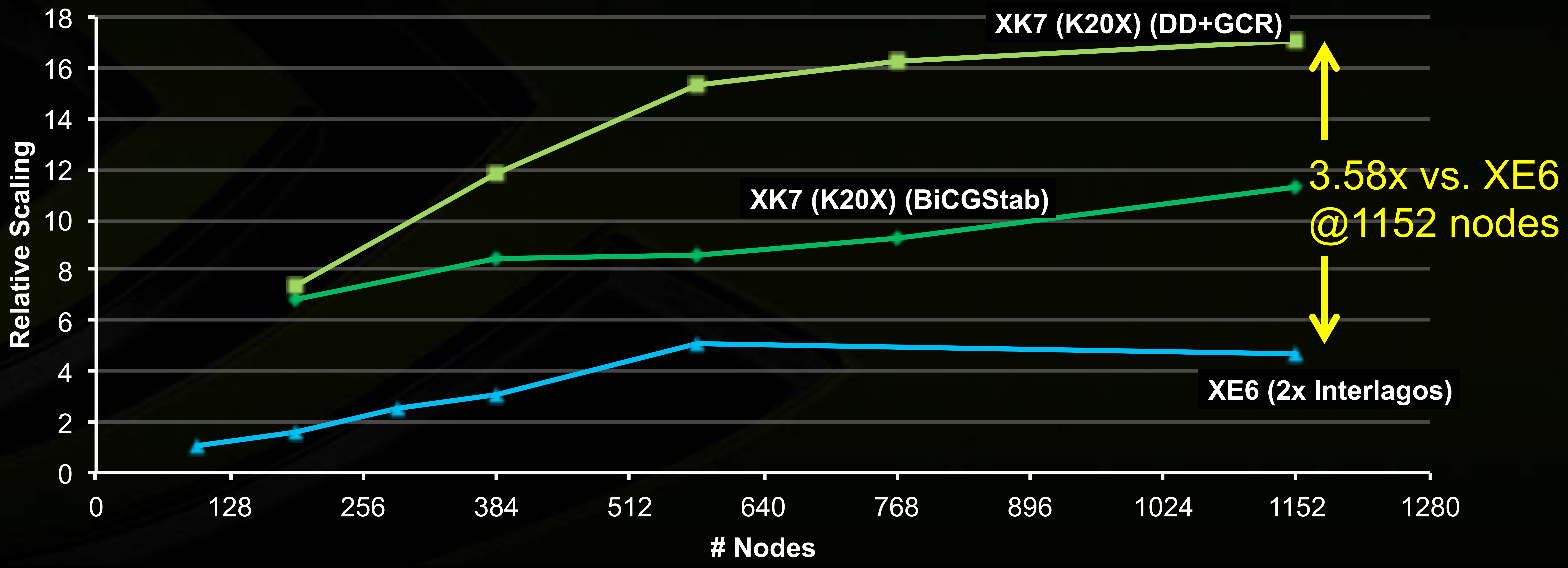
Strong Scaling Chroma

Chroma

48^3x512 lattice

Relative Scaling (Application Time)

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

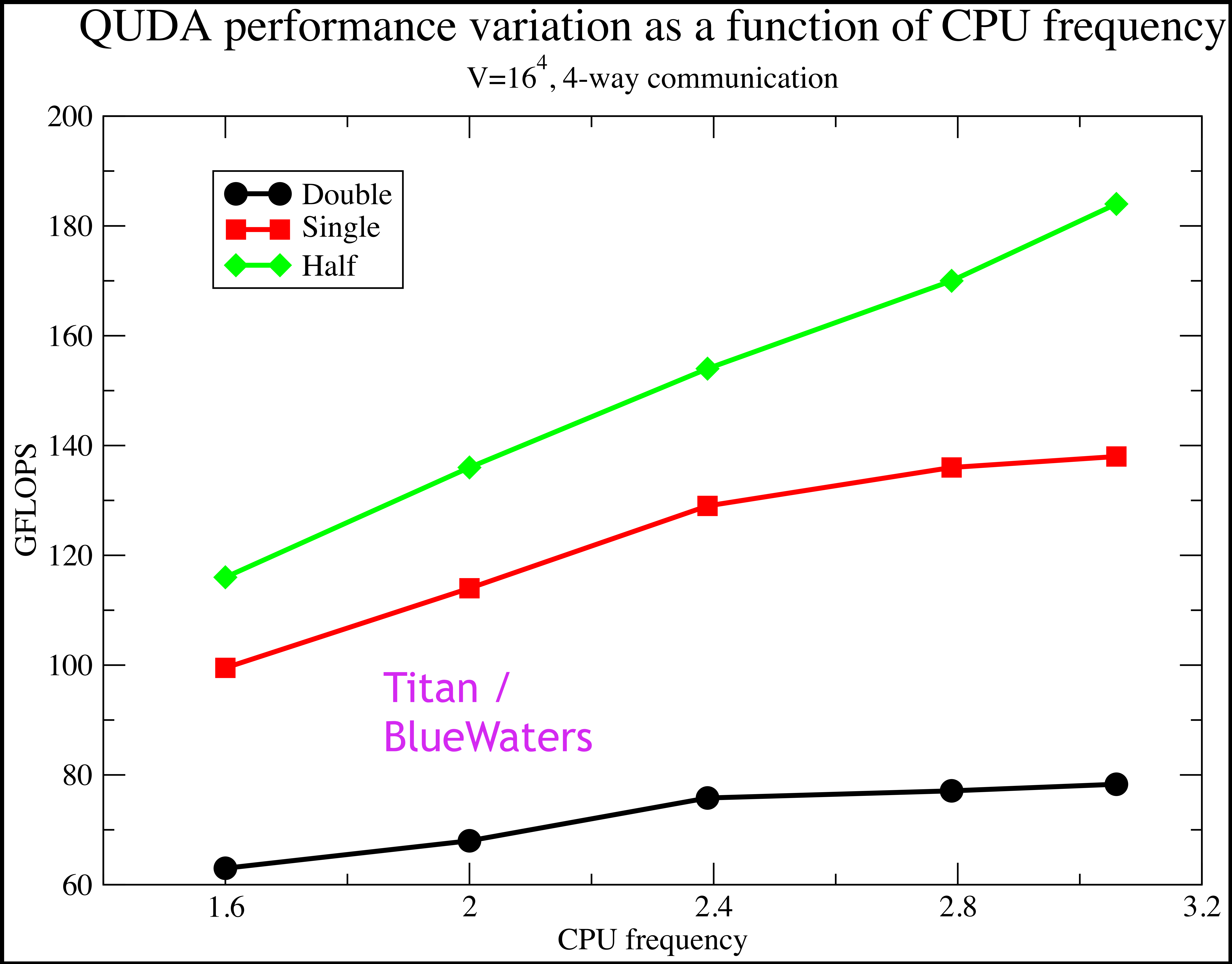


What are the strong scaling limiters?

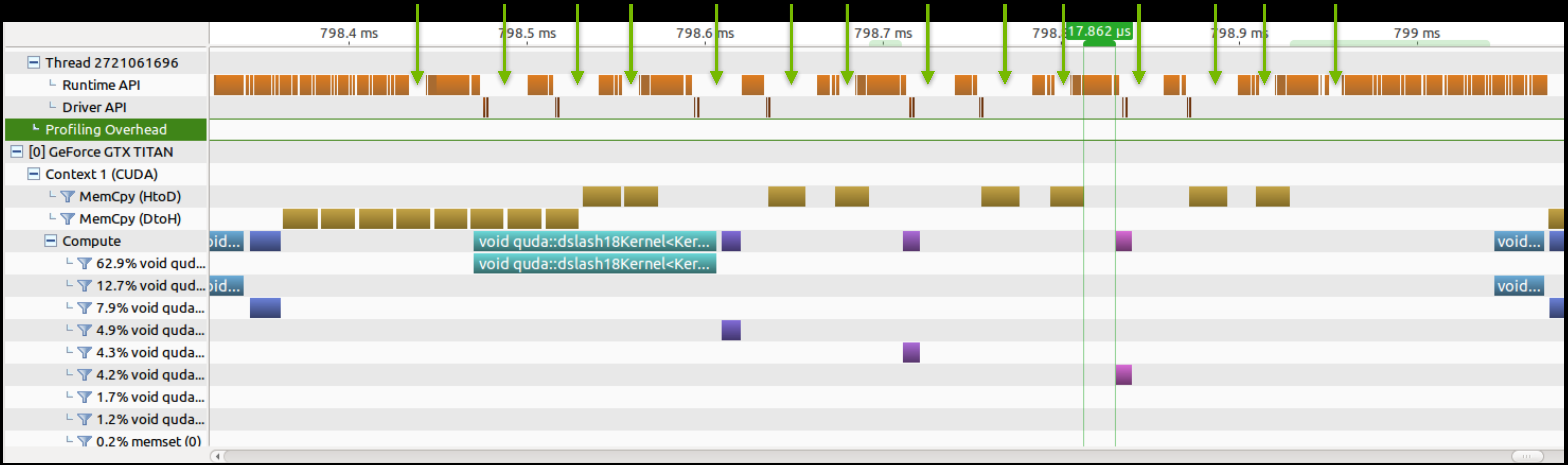
- Factors that will limit strong scaling
  - Network bandwidth
  - Network latency
  - PCIe bandwidth
  - PCIe latency
  - GPU utilization
  - GPU launch latency
- Never really looked at strong scaling for 2+ years…
- Can we do better without new algorithms?

Time to revisit…

- PCIe generation 3
  - Almost doubling in PCIe bandwidth
  - No improvement
- GPUDirect RDMA
  - Reduces end-to-end latency by a factor of three
  - No improvement
- Stream priorities for better kernel concurrency
  - No improvement
- What is the performance limiter?

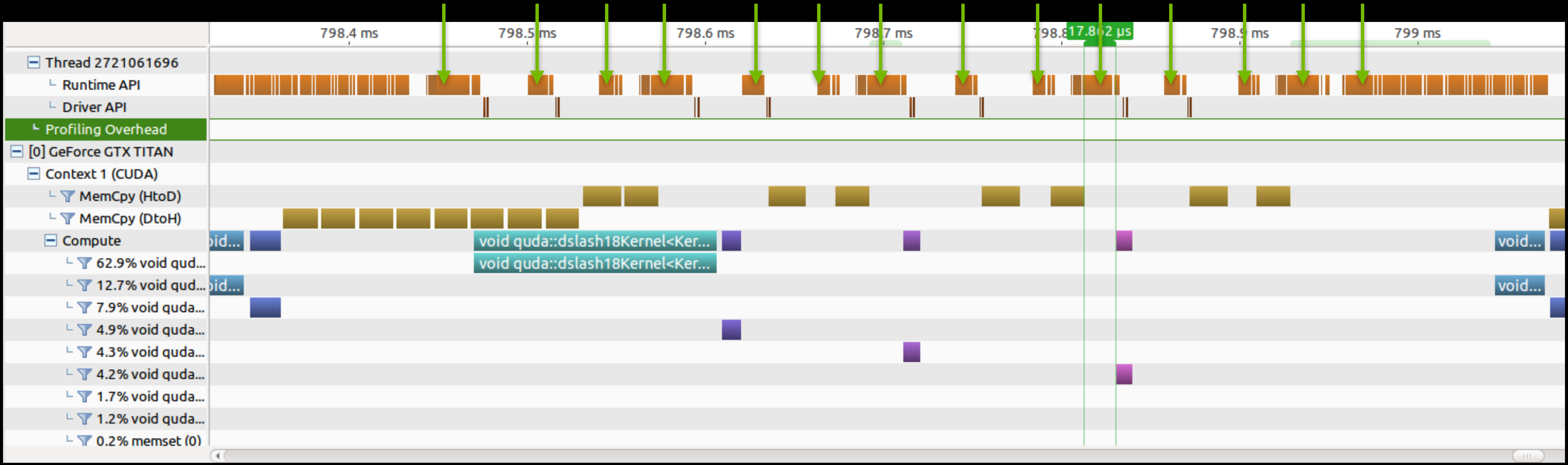

QUDA performance variation as a function of CPU frequency
V=16^4, 4-way communication
GFLOPS
CPU frequency
200
180
160
140
120
100
80
60
1.6
2
2.4
2.8
3.2
Double
Single
Half
Titan /
BlueWaters

Delving deeper…



- Large white spaces means cycles are being wasted on the CPU
  - But nothing's running on the CPU is it?

Delving deeper…



- Kernel launch is way bigger than expected
  - Expect 4 us, observing 8-17 us

Revisiting QUDA’s run-time tuning

- 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)
- 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
  - tuneLaunch() searches the cache and tunes if not found:

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

- As the cache increases, the tune cache lookup becomes costly…

Tuning the tuning

- Before

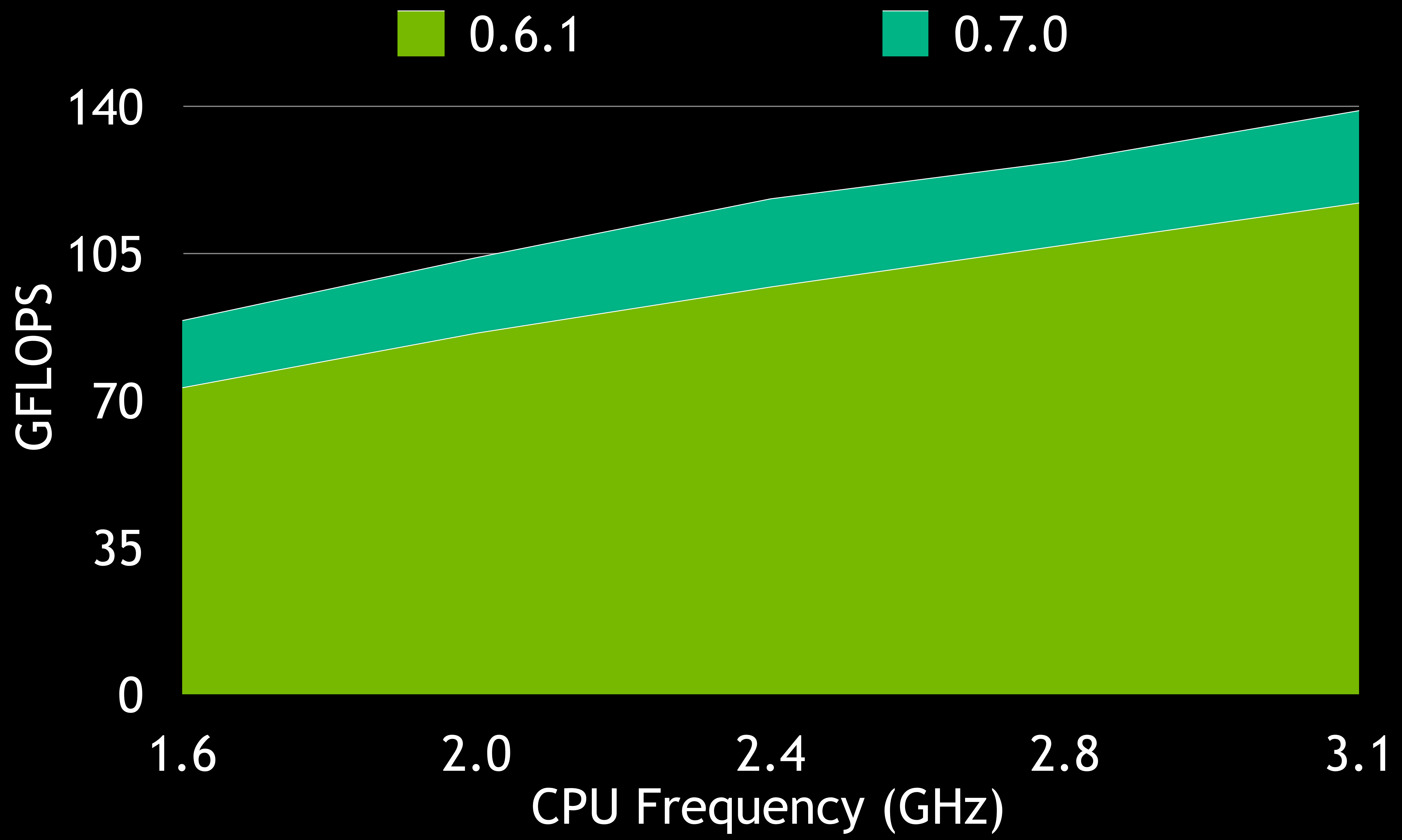
	Actual time spent (us)
TuneKey create	*4.50*
Check if entry exists	*0.60*
TuneParam lookup	*0.55*
Check launch parameters	*0.03*

- Optimizations
  - std::string -> char*
  - Cache string parameters
  - Replace count and operator[] with find

	Actual time spent (us)
TuneKey create	*0.20*
TuneParam lookup	*0.25*
Check launch parameters	*0.03*

Very quick payback

- Preliminary result from a few hours of work
  - 16^4 wilson dslash, 4-way comms



More work to do

- Expect to be able to further half the tune cache look up
  - Completely negligible compared to other overheads
- Kernel launch overheads still double what should be expected
  - Dslash kernel launch takes 9 us
    - <1 us spent in tune cache lookup
    - 8 us spent in cudaLaunch (expect < 4us)
  - Initial investigation suggests that std::map is blowing out the cache hampering subsequent kernel launch
  - Investigating more cache friendly alternatives (e.g., Loki)
  - Hope to get effective kernel launch close to SOL
- Better scaling on Titan / Blue Waters without new algorithms
- Expect PCIe gen 3, GPUDirect, etc. will have significant benefit

Reworking the Dslash Communications

- Work being done by Justin Foley
  - Double buffering QMP/MPI receive buffers for pre posting
  - Threading the QUDA dslash to further reduce CPU load, E.g.,
    - thread 0: pack kernel, initiate communications
    - thread 1: interior kernel, boundary kernels
  - One-sided MPI for reduced inter-node latency
  - Broad GPU Direct RDMA support
  - More kernel fusion

- Native communications layer
  - PEACH2 support (Tsukuba)
  - IB VERBS support (Jiri Kraus - Julich)
  - Cray uGNI ? (Cray interested in supporting this)

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

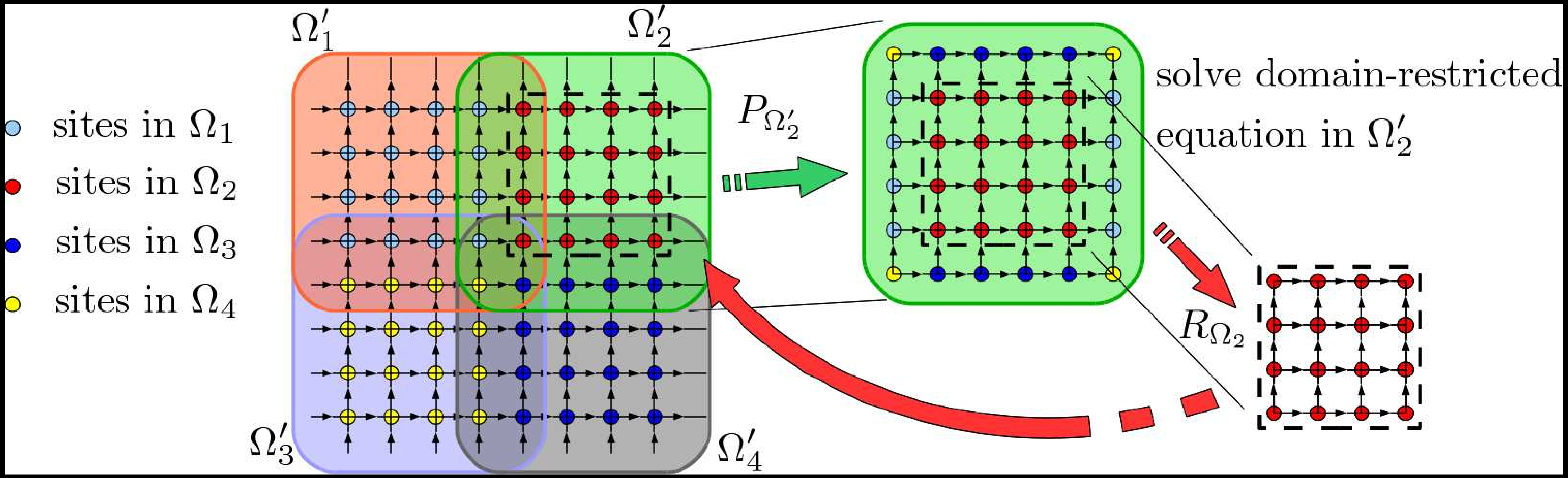
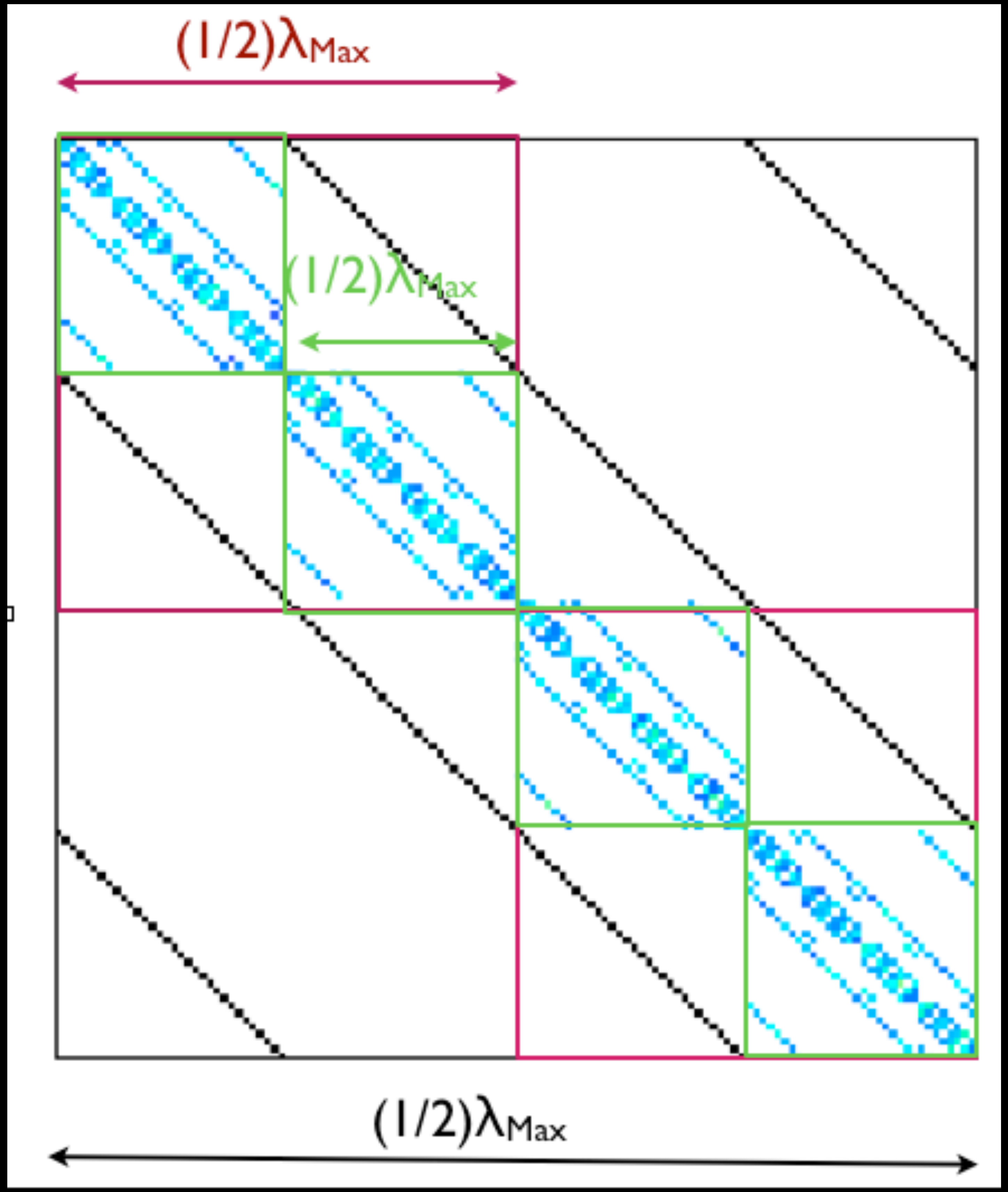


figure taken from Osaki and Ishikawa

- Utilize s-step solvers to suppress global sums
  - Allows for the introduction of tower-kernels to decrease local comms

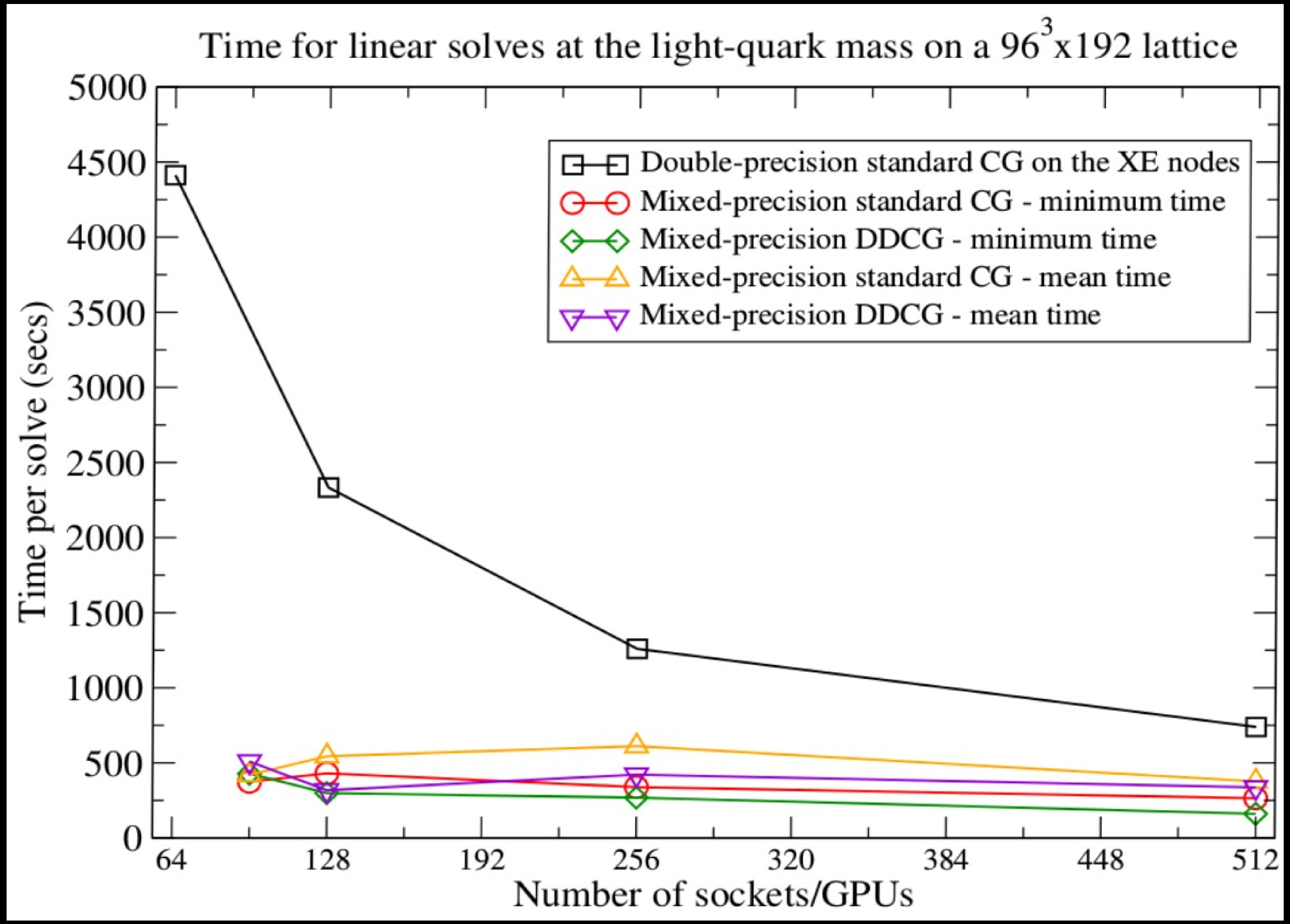
Communication-reducing Algorithms

- 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

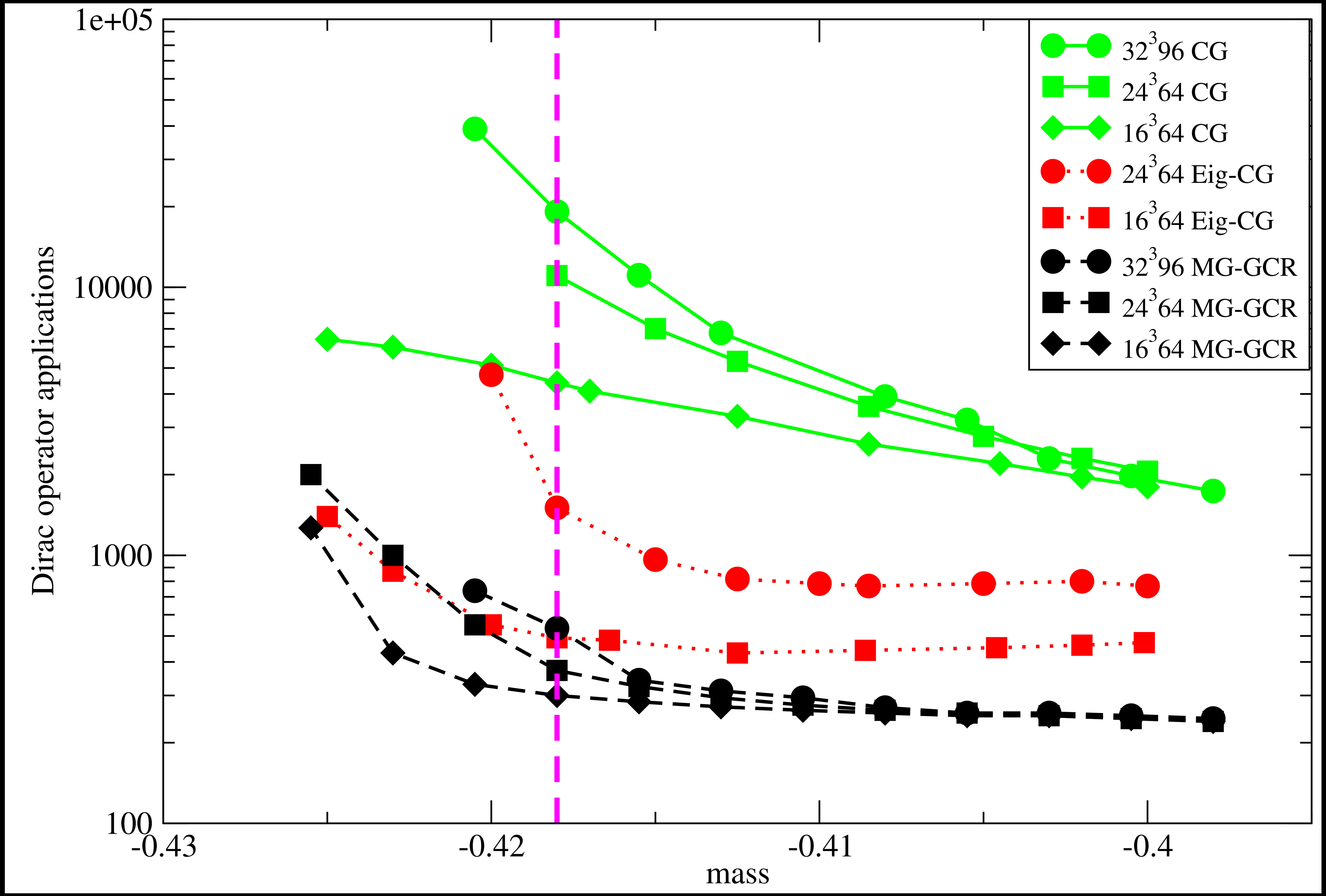


Communication-reducing Algorithms

- QUDA now support for overlapping blocks (Justin)
  - Motivated for staggered solvers
  - Checked into 0.7 branch
- Staggered DD solvers
  - Communication reduced by 4x
  - Not much actual speedup
- Need to revisit this with reworked tuning engine



Adaptive Geometric Multigrid



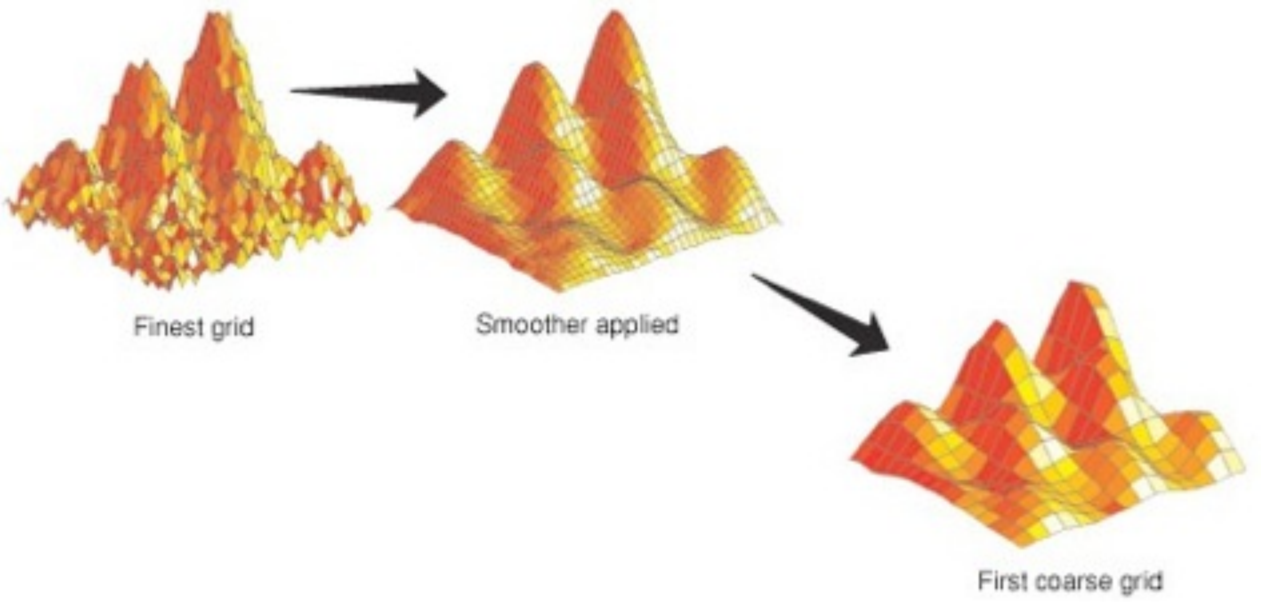
240 vectors

20 vectors

Babich *et al* 2010

Adaptive Geometric Multigrid

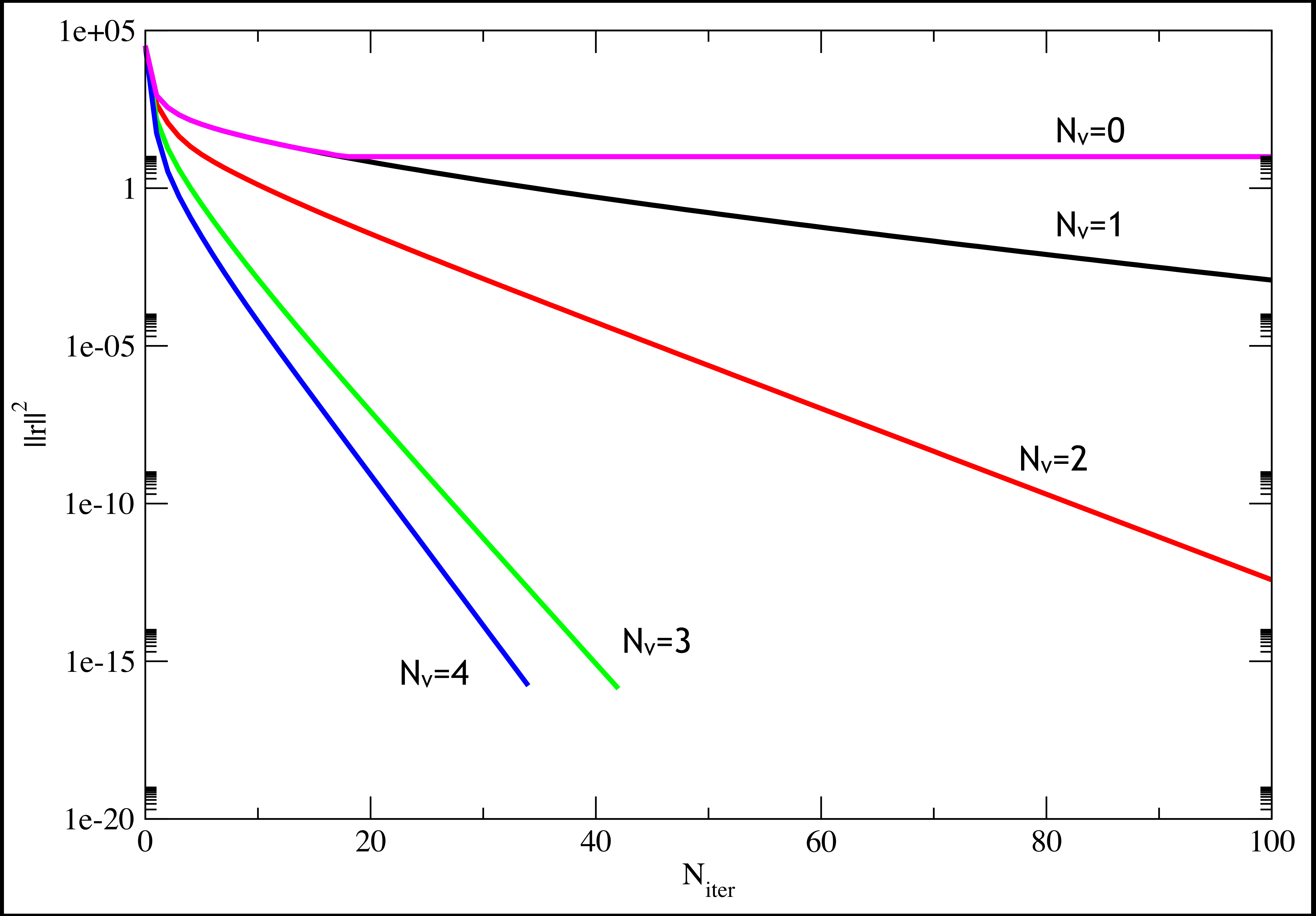
- Adaptively find candidate null-space vectors
  - Dynamically learn the null space and use this to define the prolongator
  - Algorithm is self learning



- Setup

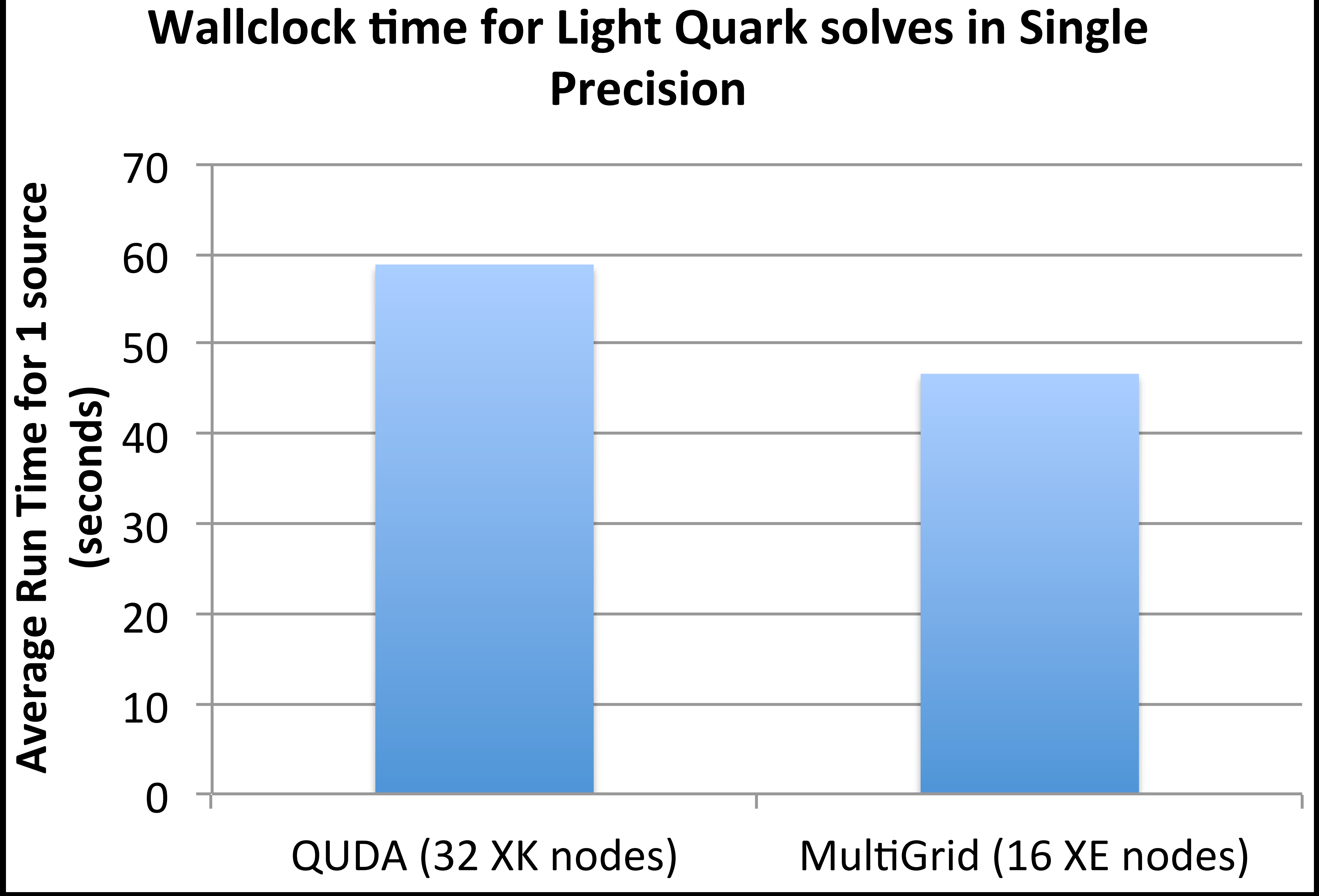
1. Set solver to be simple smoother
2. Apply current solver to random vector $v_i = P(D)\, \eta_i$
3. If convergence good enough, solver setup complete
4. Construct prolongator using fixed coarsening $(1 - P\, R)\, v_k = 0$
   ➡ Typically use 4^4 geometric blocks
   ➡ Preserve chirality when coarsening $R = \gamma_5 P^\dagger \gamma_5 = P^\dagger$
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



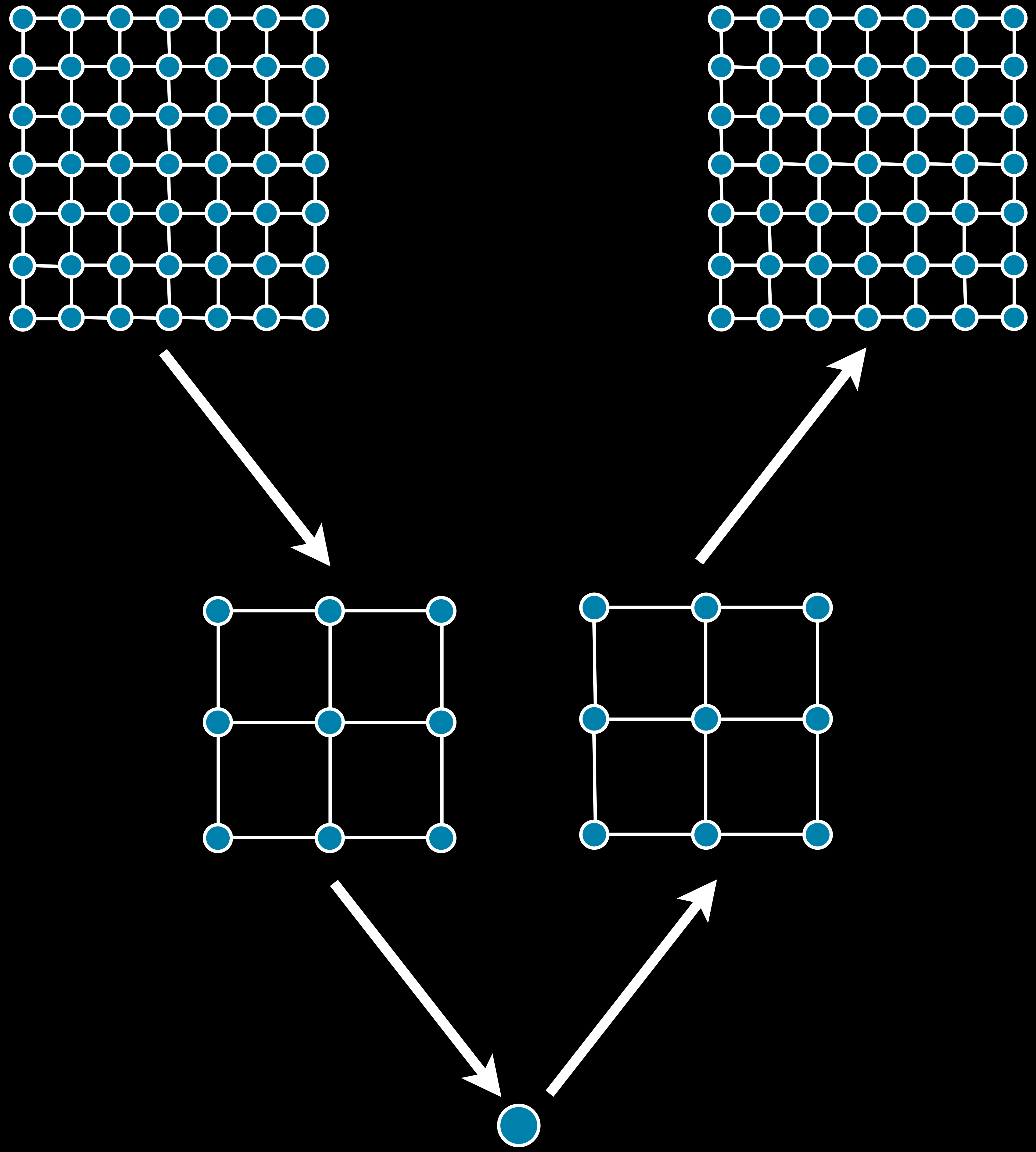
Motivation

- 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



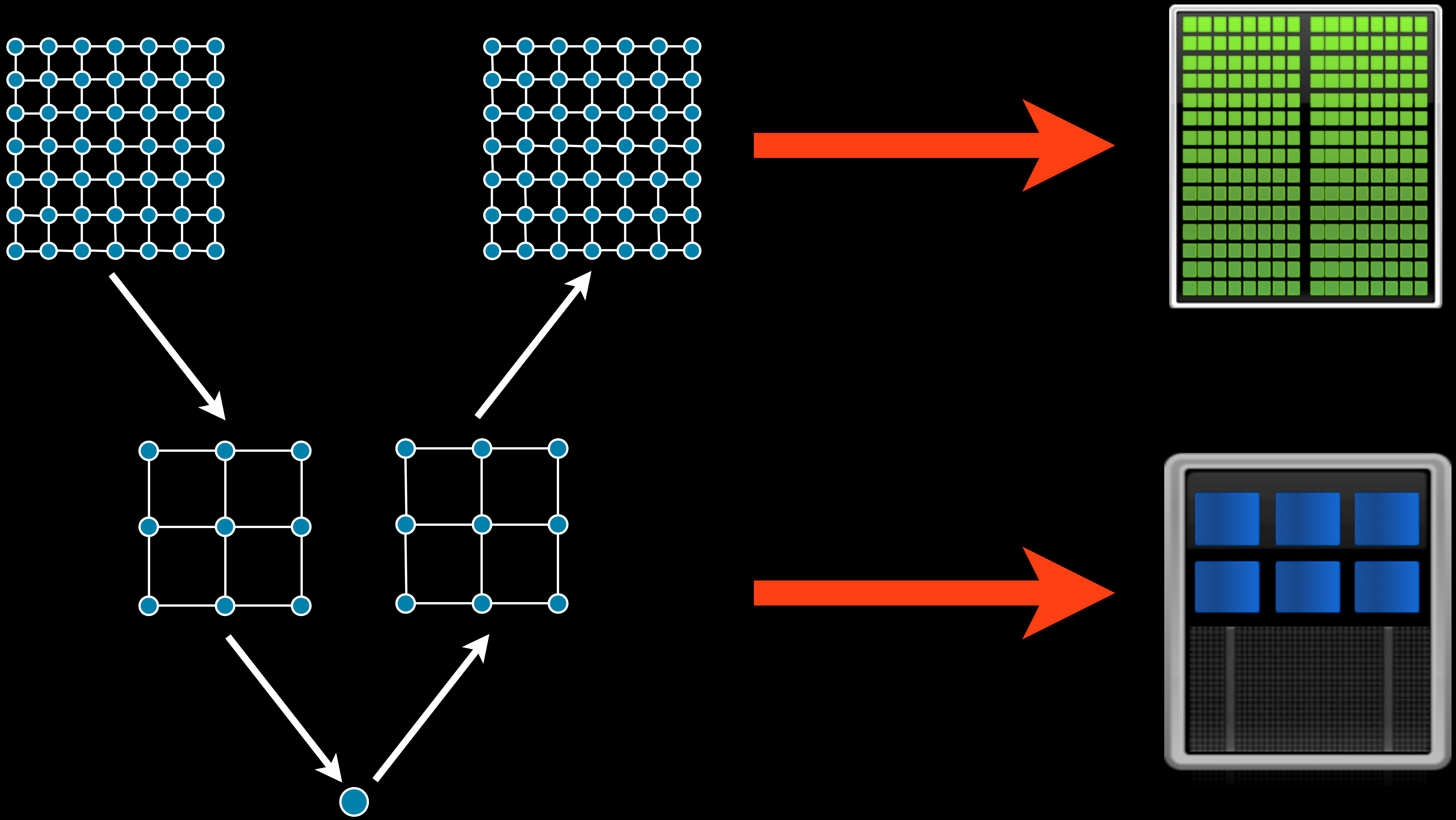
Chroma propagator benchmark
Figure by Balint Joo
MG Chroma integration by Saul Cohen
QOPQDP MG by James Osborn

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

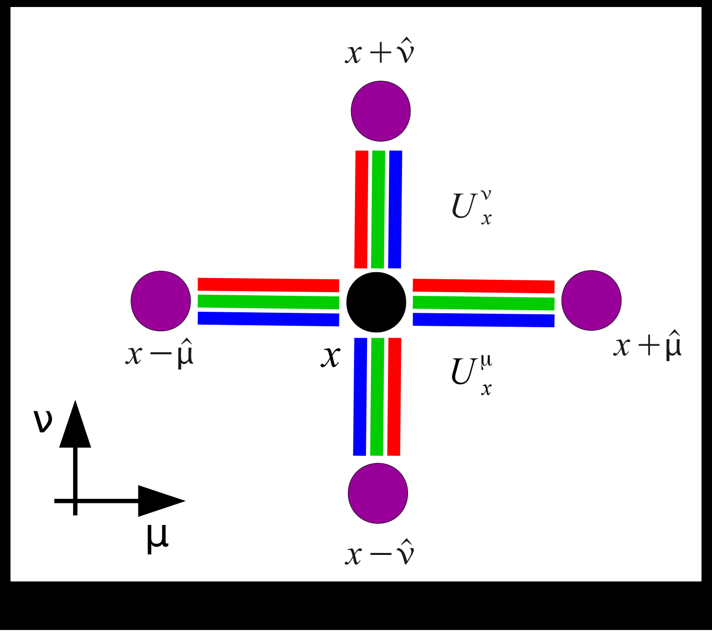
Thousands of cores for parallel processing

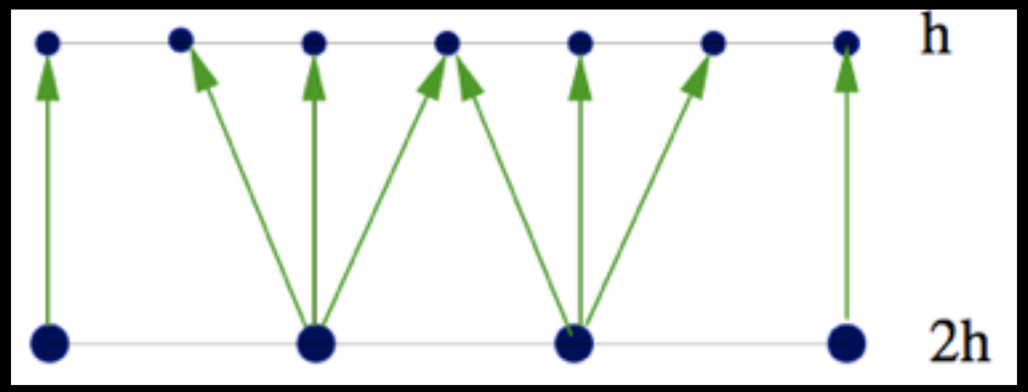
CPU

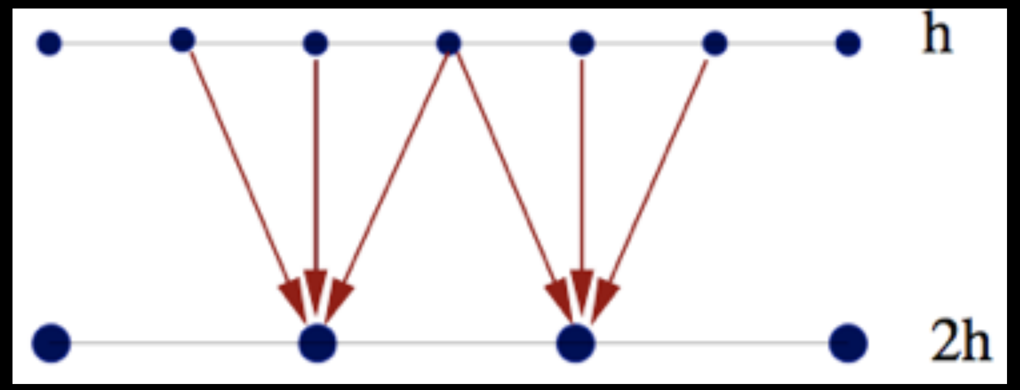
Few Cores optimized for serial work

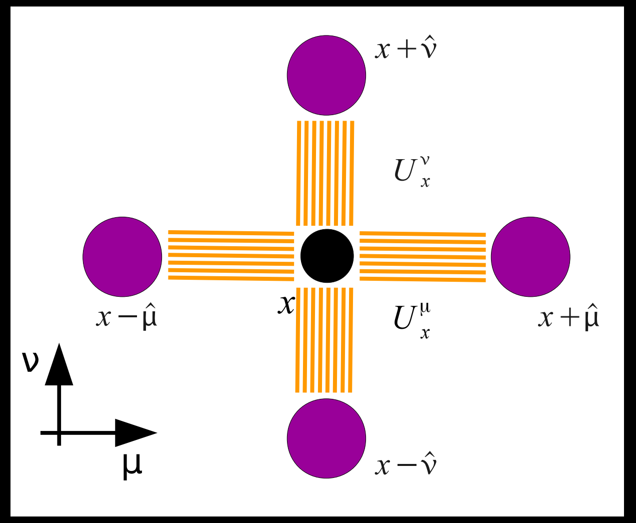
Ingredients for Parallel Adaptive Multigrid

- 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







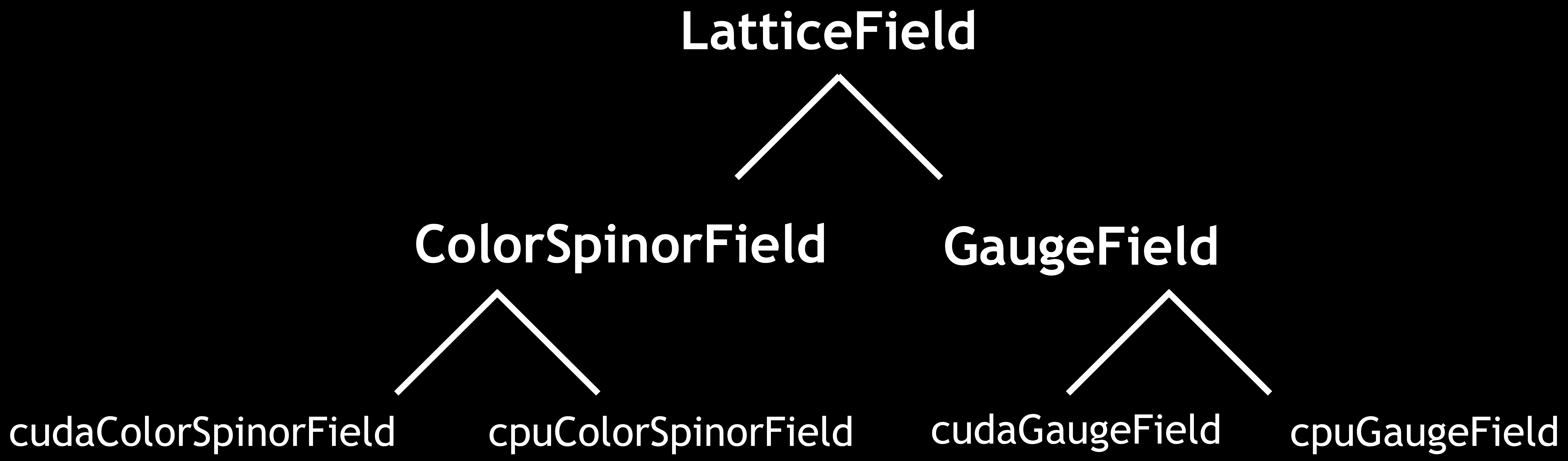


Design Goals

- 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

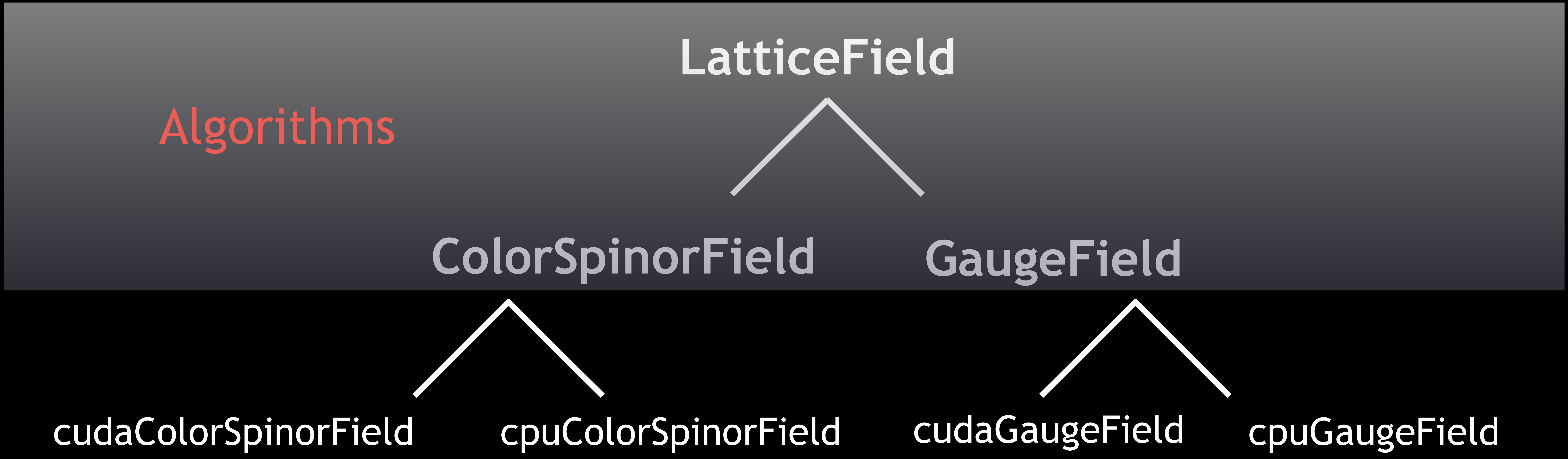
cpuColorSpinorField

cudaGaugeField

cpuGaugeField

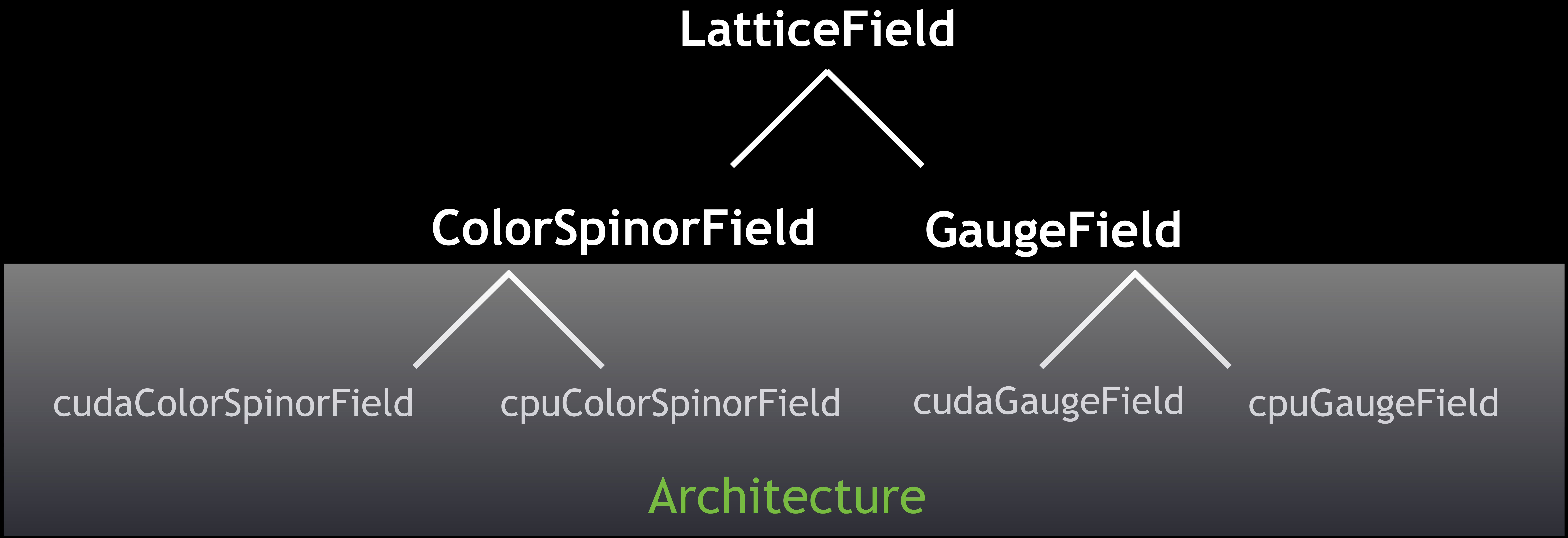
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity

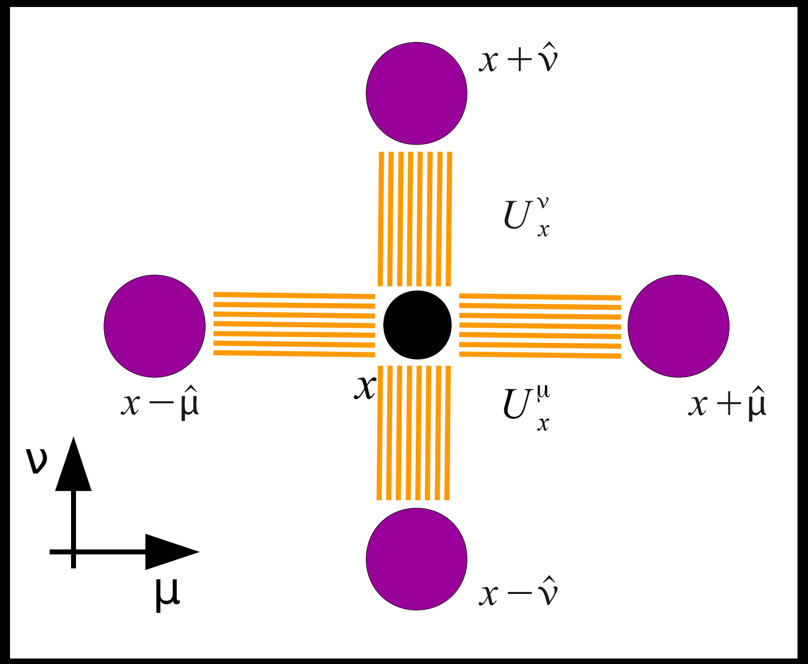


Multigrid and QUDA

- 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
  }

}
```

Parallel Implementation



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

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

- 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

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…

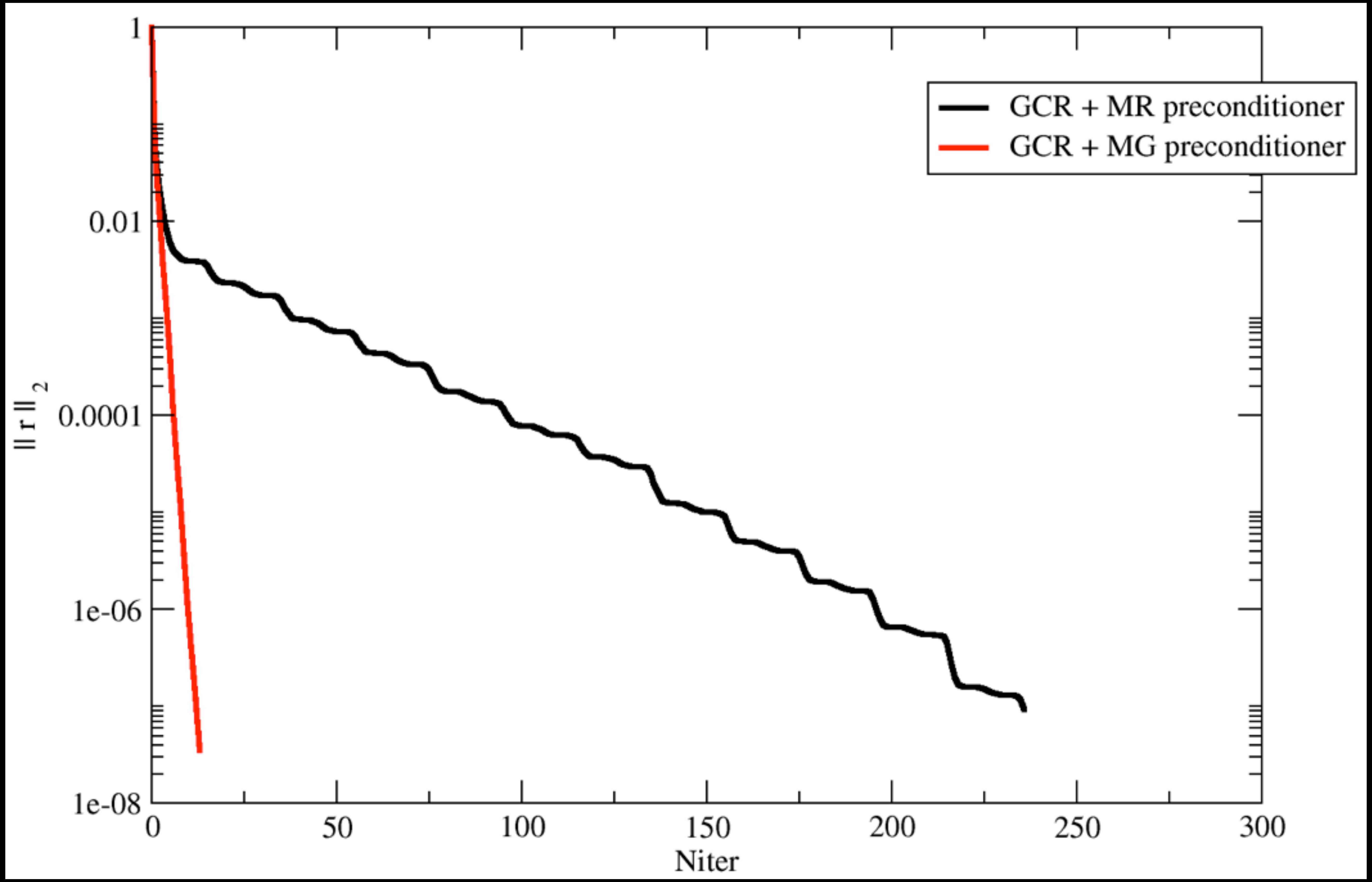
- All *internal* parameters must be known at *compile* time
  - Template over every possible combination O(10,000) combinations
    - Tensor product between different parameters
    - O(10,000 combinations) *per* kernel
  - Only compile necessary kernel at runtime

```
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 would help here…

Current Status

- Framework is working



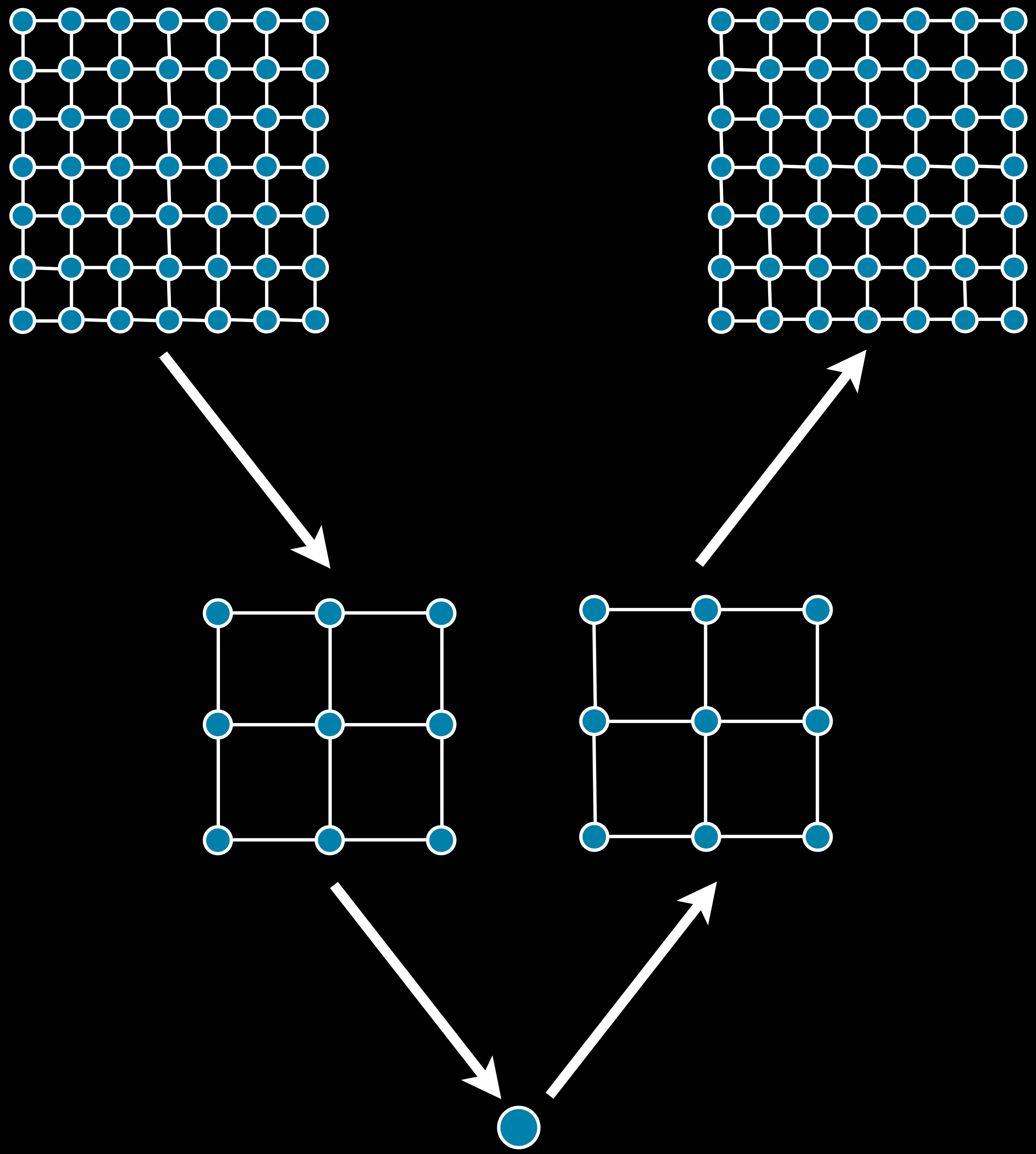
Fine grid on GPU
Coarse grid on CPU

Runtime decision as to where each component is running

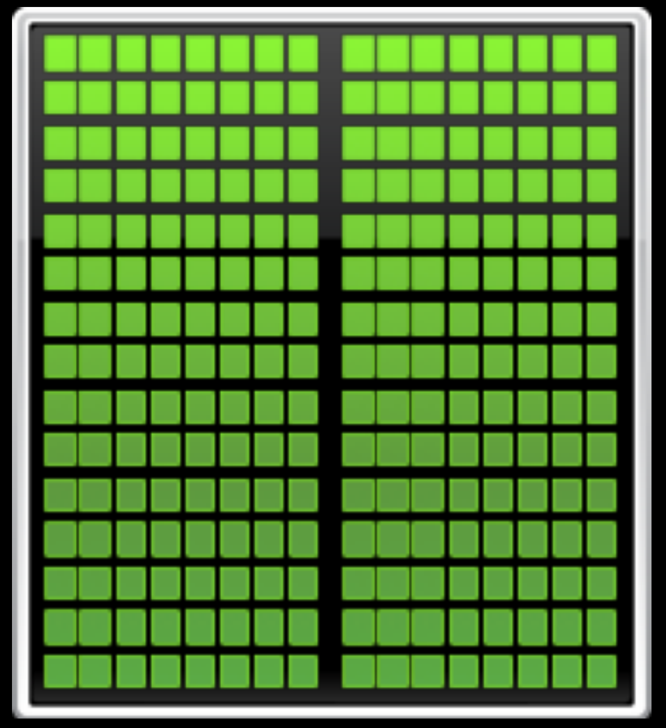
Hierarchical Algorithm Toolbox

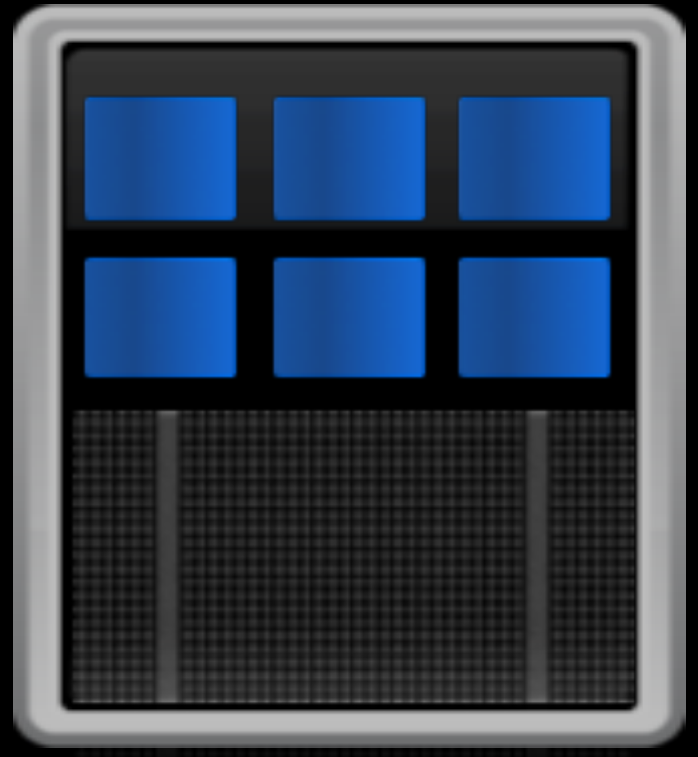
- Real goal is to produce scalable and optimal solvers
- Exploit closer coupling of precision and algorithm
  - QUDA designed for complete run-time specification of precision at any point in the algorithm
  - Currently supports 64-bit, 32-bit, 16-bit
  - Is 128-bit or 8-bit useful at all for hierarchical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

Heterogeneous Updating Scheme

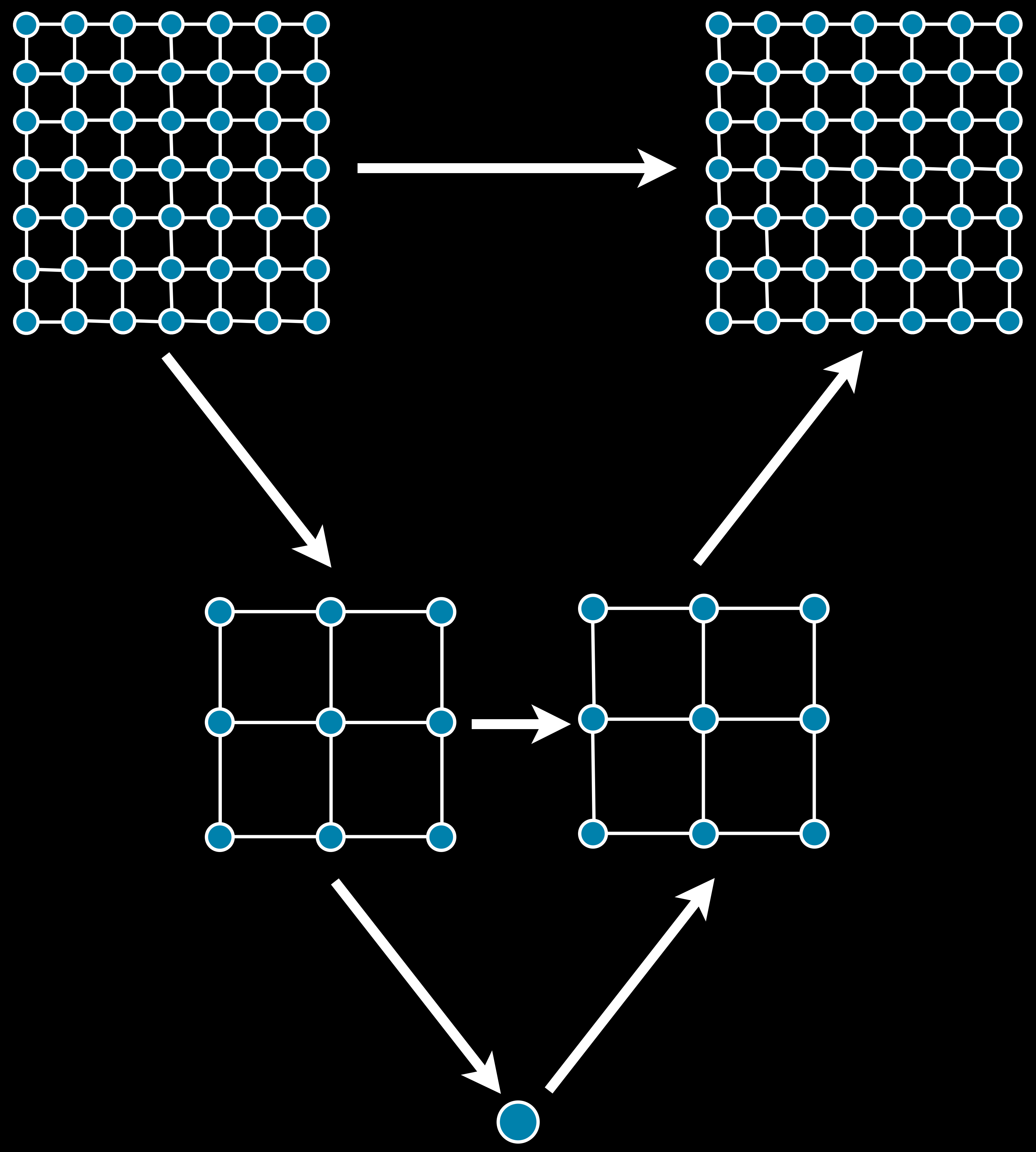


- Multiplicative MG is necessarily serial process
  - Cannot utilize both GPU and CPU simultanesouly

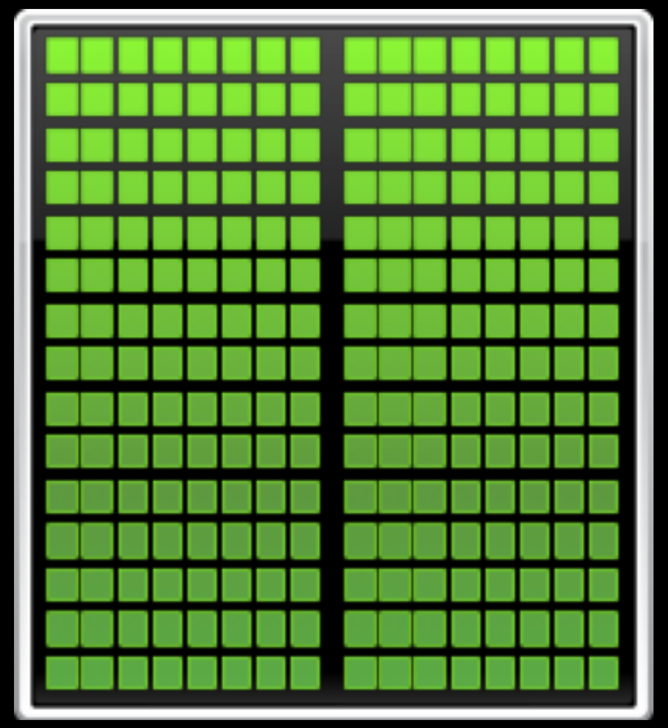


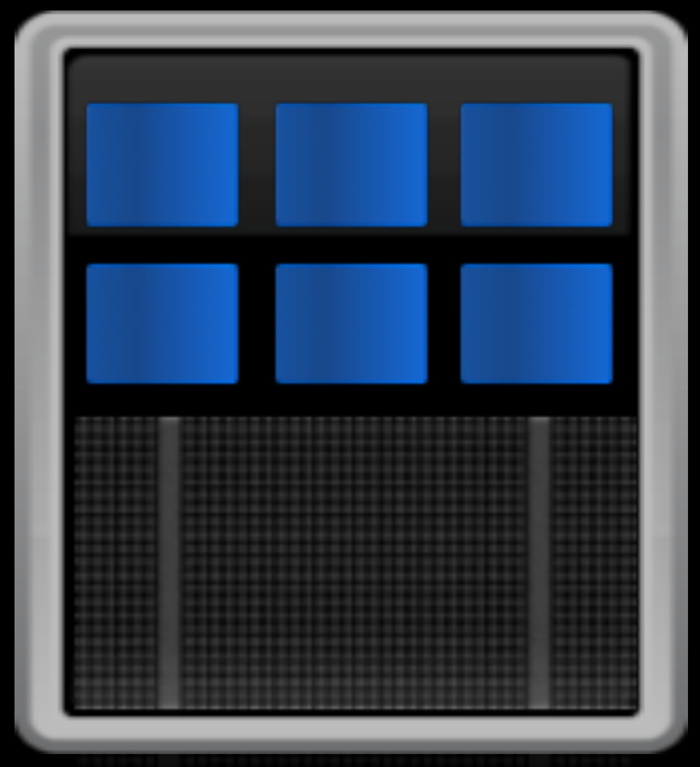


Heterogeneous Updating Scheme



- 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

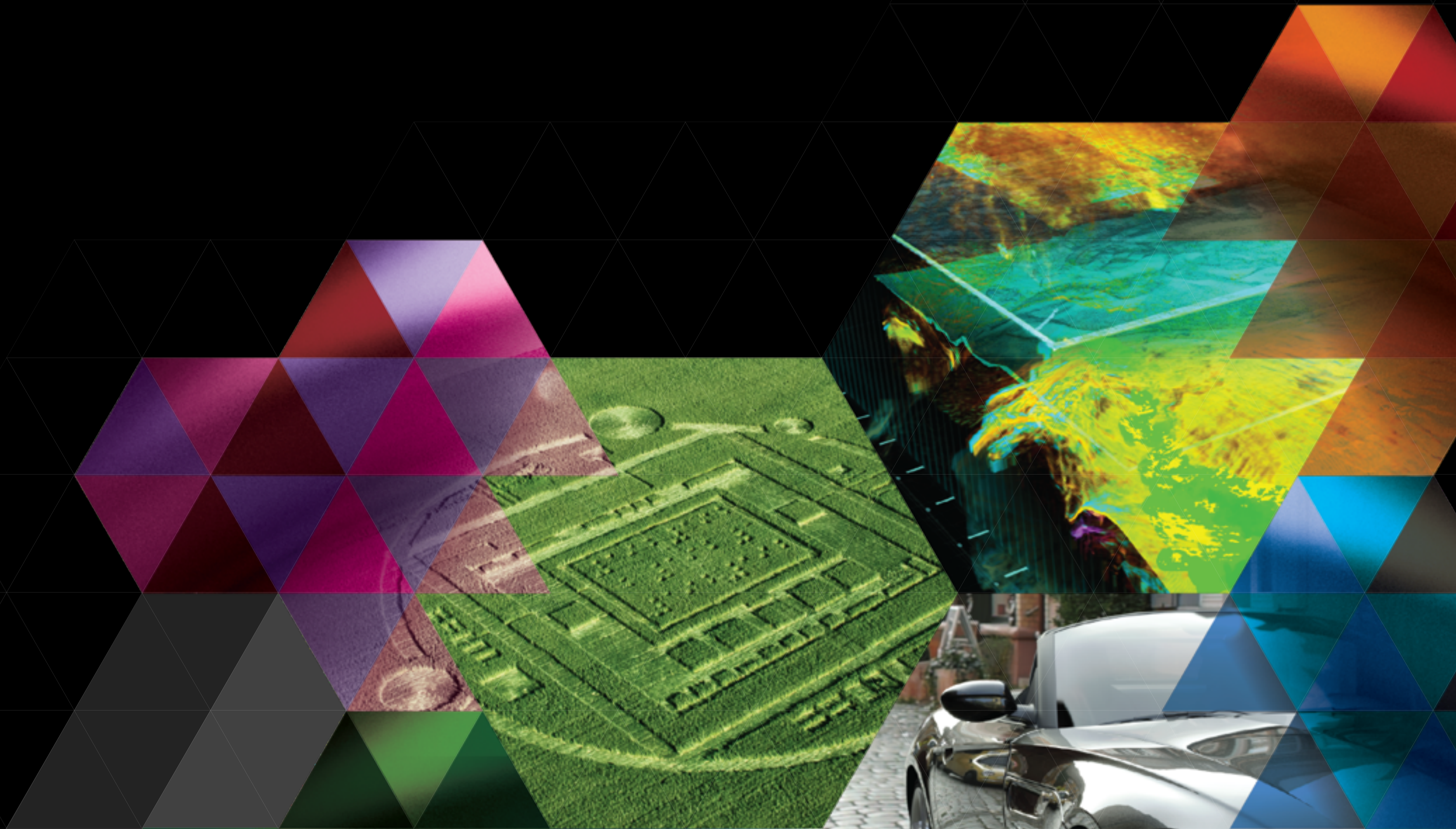




Summary

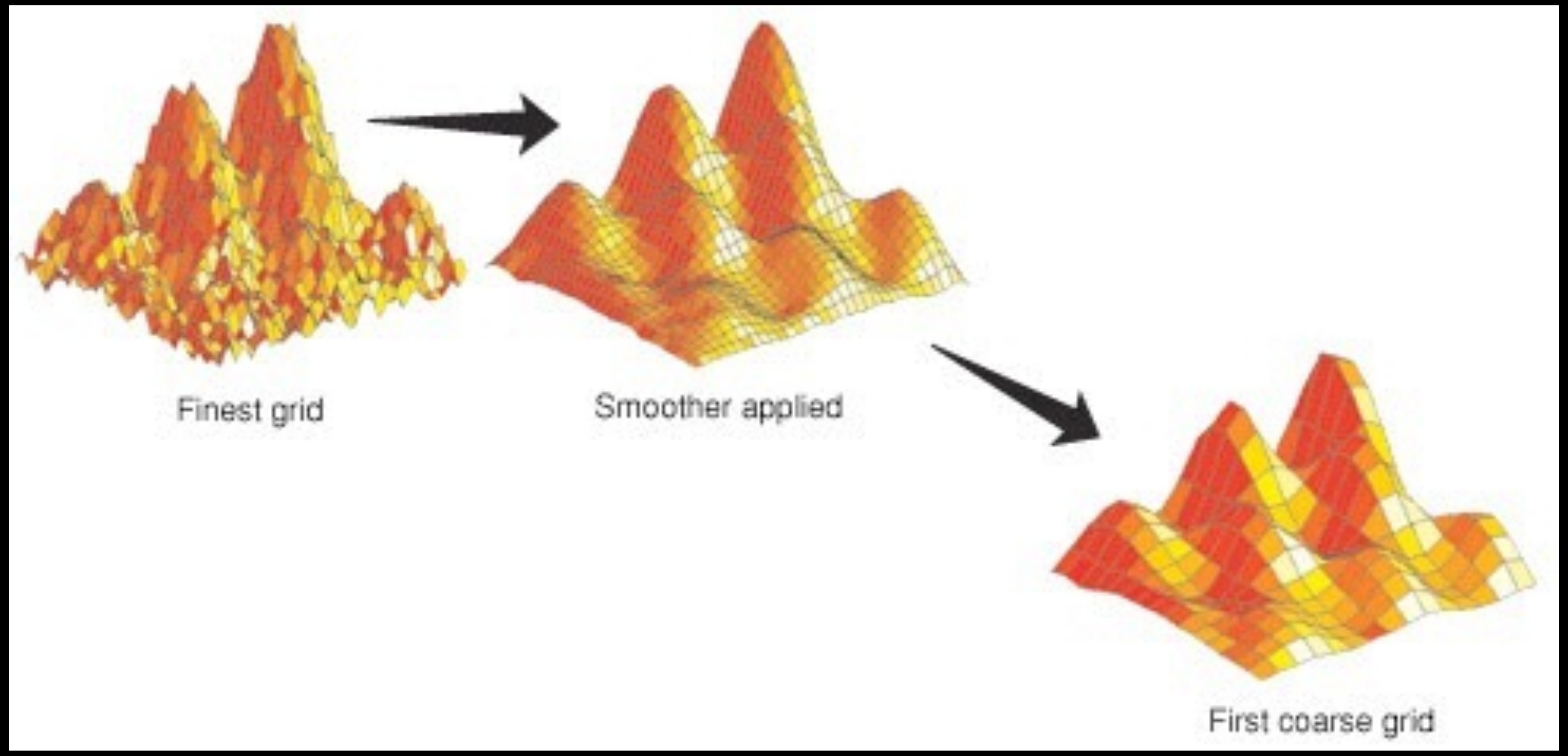
- QUDA is reaching critical mass
  - Coverage for most LQCD algorithms
- Production library for GPU-accelerated LQCD
  - Scalable linear solvers
- Last couple of years have focussed on broad coverage
- Refocusing on linear solvers
  - Strong scaling
  - Optimal solvers
- Hierarchical *and* heterogeneous algorithm research toolbox
  - Aim for scalability *and* optimality
- Lessons today are relevant for Exascale preparation

BACK UP SLIDES



Introduction to Multigrid

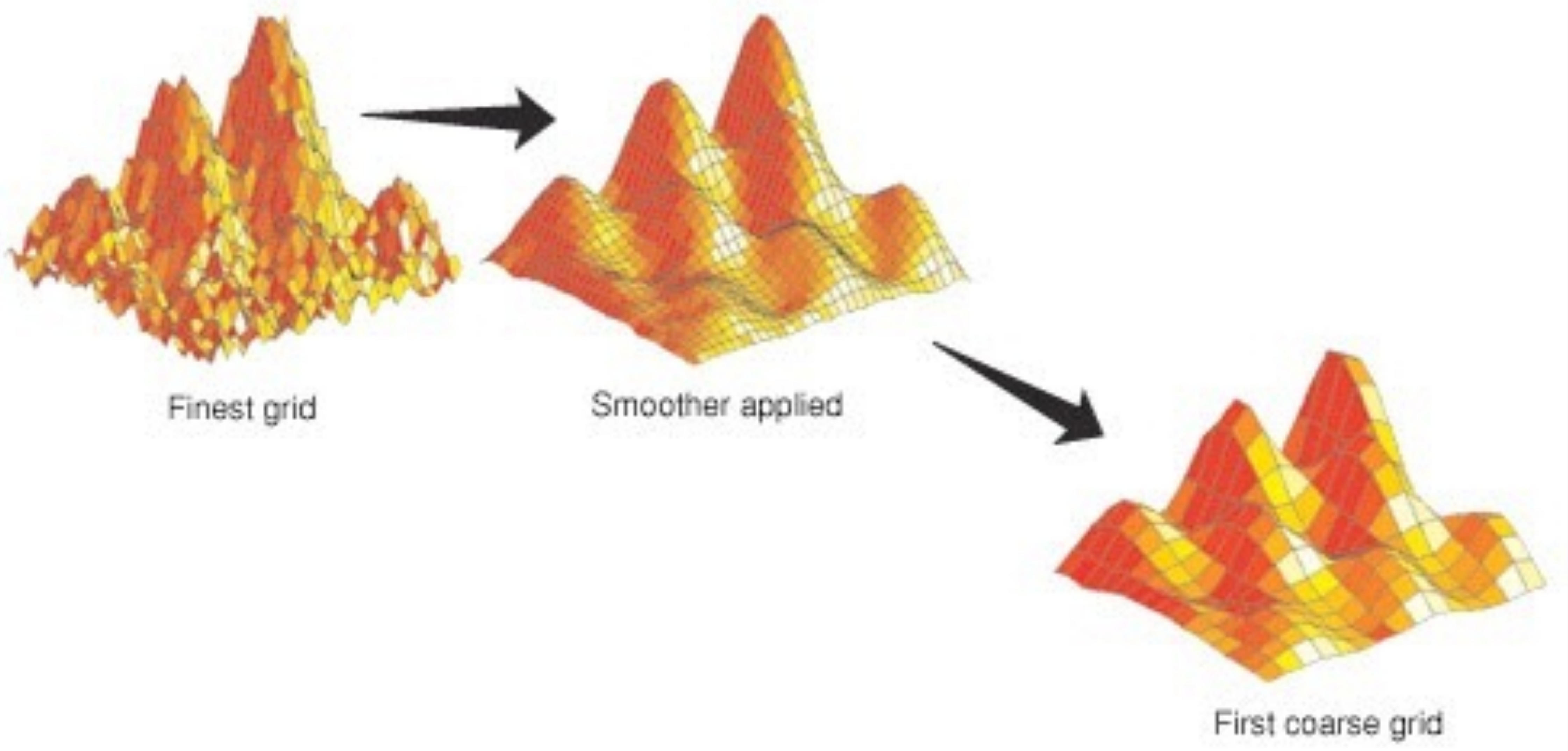
- Low frequency error modes are smooth
- Can accurately represent on coarse grid



- 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

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
```

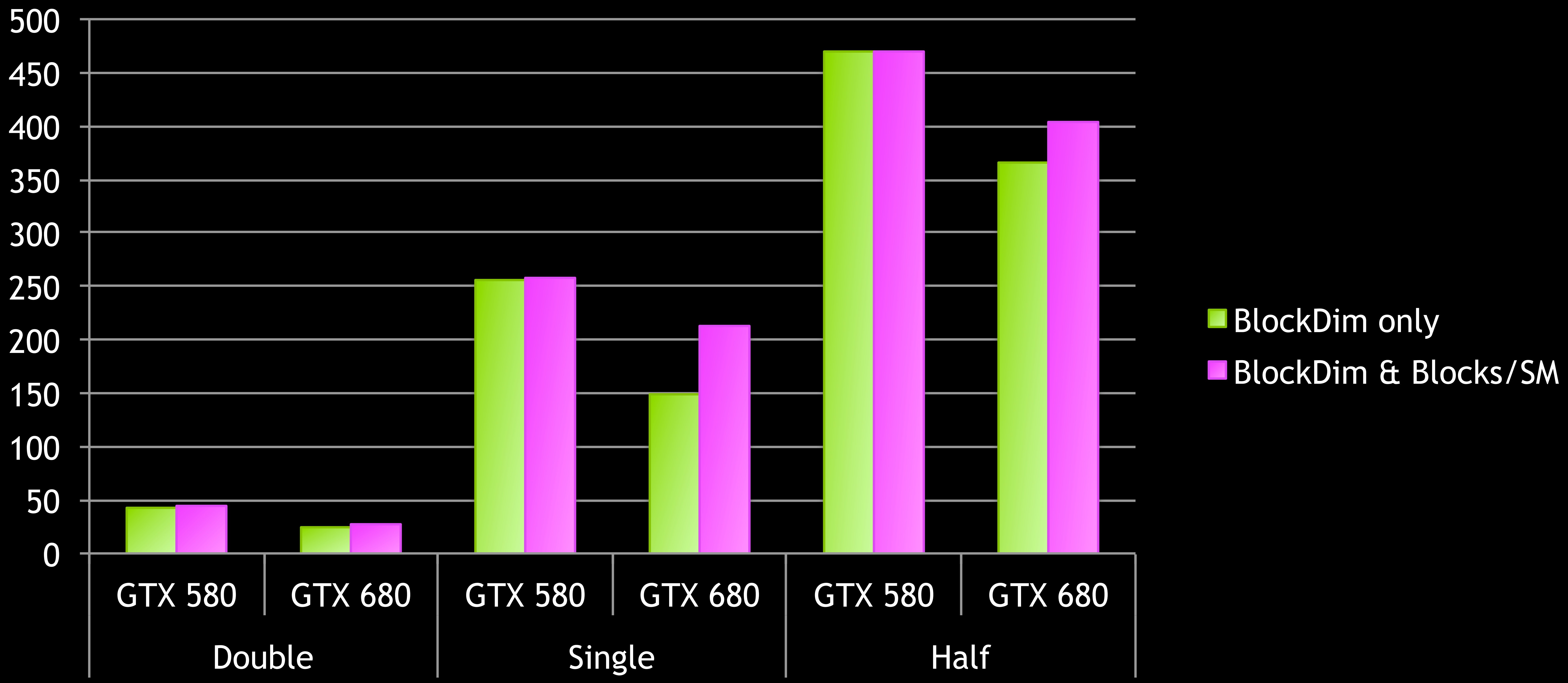
- Here MyKernelWrapper inherits from Tunable and optionally overloads various virtual member functions (next slide).
- Wrapping related kernels in a class hierarchy is often useful anyway, independent of tuning.

Virtual member functions of Tunable

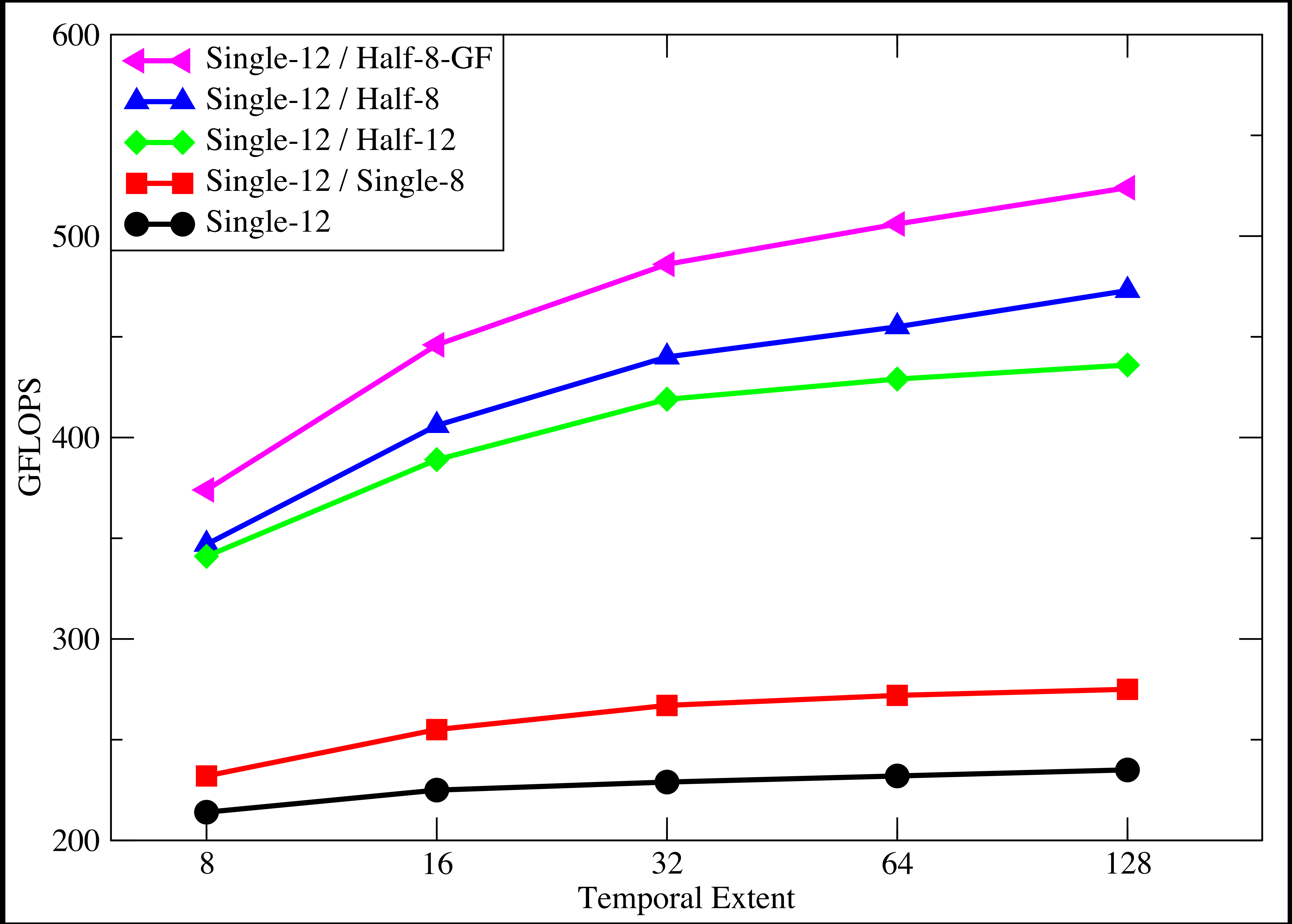
- Invoke the kernel (tuning if necessary):
  - —apply()
- Save and restore state before/after tuning:
  - —preTune(), postTune()
- Advance to next set of trial parameters in the tuning:
  - —advanceGridDim(), advanceBlockDim(), advanceSharedBytes()
  - —advanceTuneParam() // simply calls the above by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

Auto-tuned “warp-throttling”

- Motivation: Increase reuse in limited L2 cache.

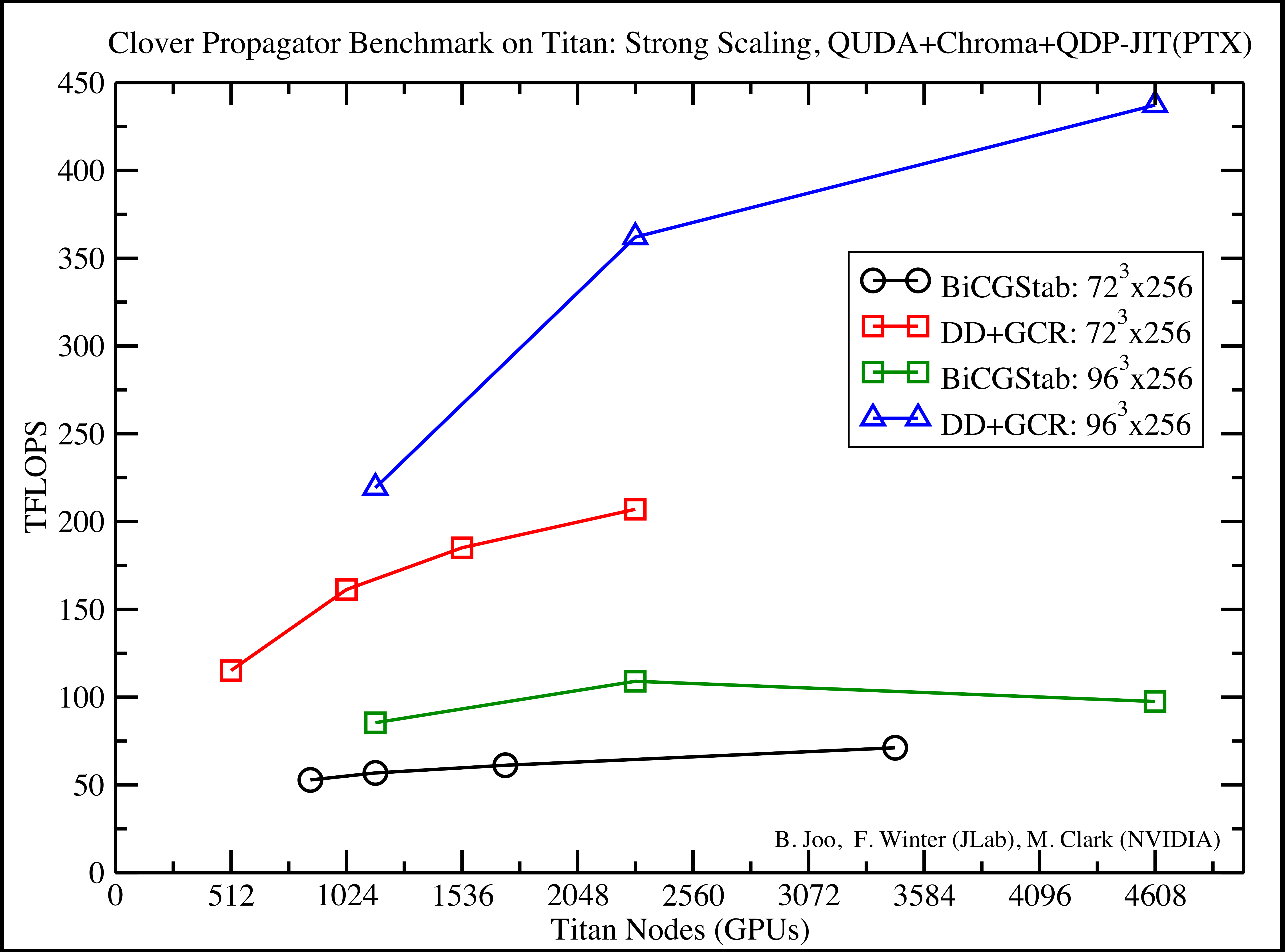


Kepler Wilson-Solver Performance



Wilson CG
K20X performance
V = 24^3xT

Extreme Scaling


Clover Propagator Benchmark on Titan: Strong Scaling, QUDA+Chroma+QDP-JIT(PTX)
TFLOPS
Titan Nodes (GPUs)
BiCGStab: 72^3x256
DD+GCR: 72^3x256
BiCGStab: 96^3x256
DD+GCR: 96^3x256
B. Joo, F. Winter (JLab), M. Clark (NVIDIA)