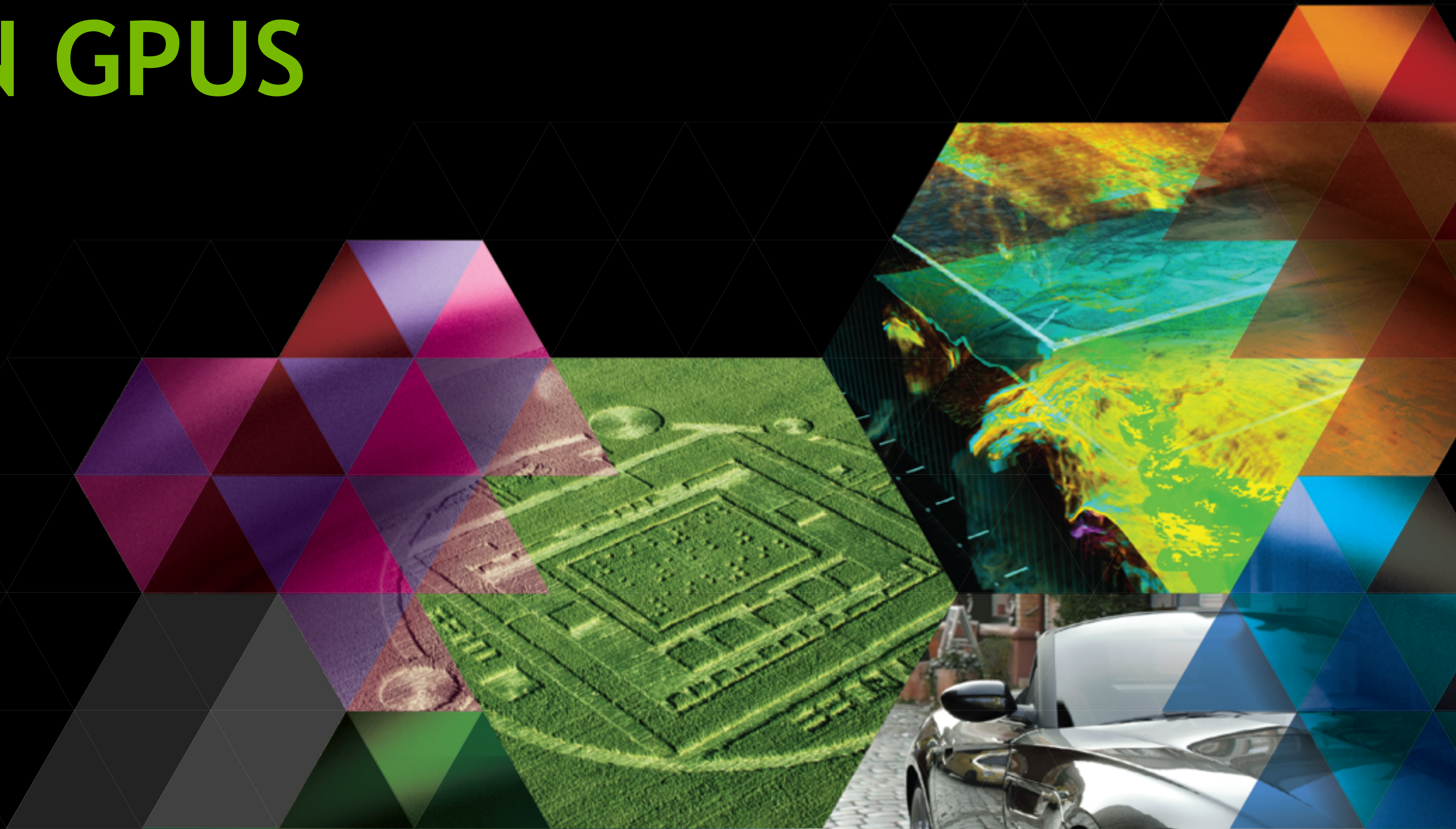
Lattice 2014

ADAPTIVE MULTIGRID SOLVERS FOR LQCD ON GPUS

M Clark
with
Michael Cheng and Rich Brower
(Boston University)



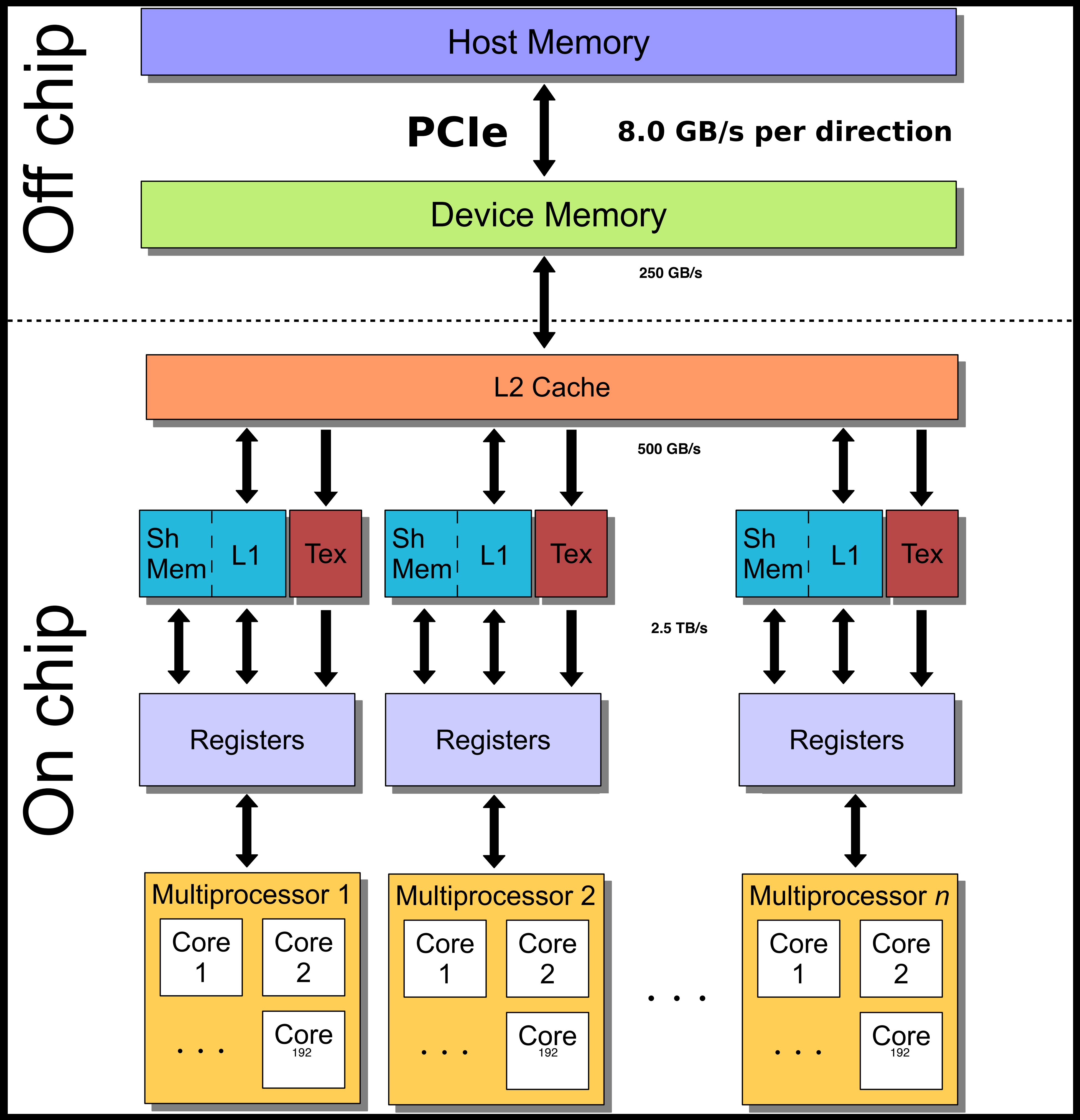
Lattice 2014

Contents

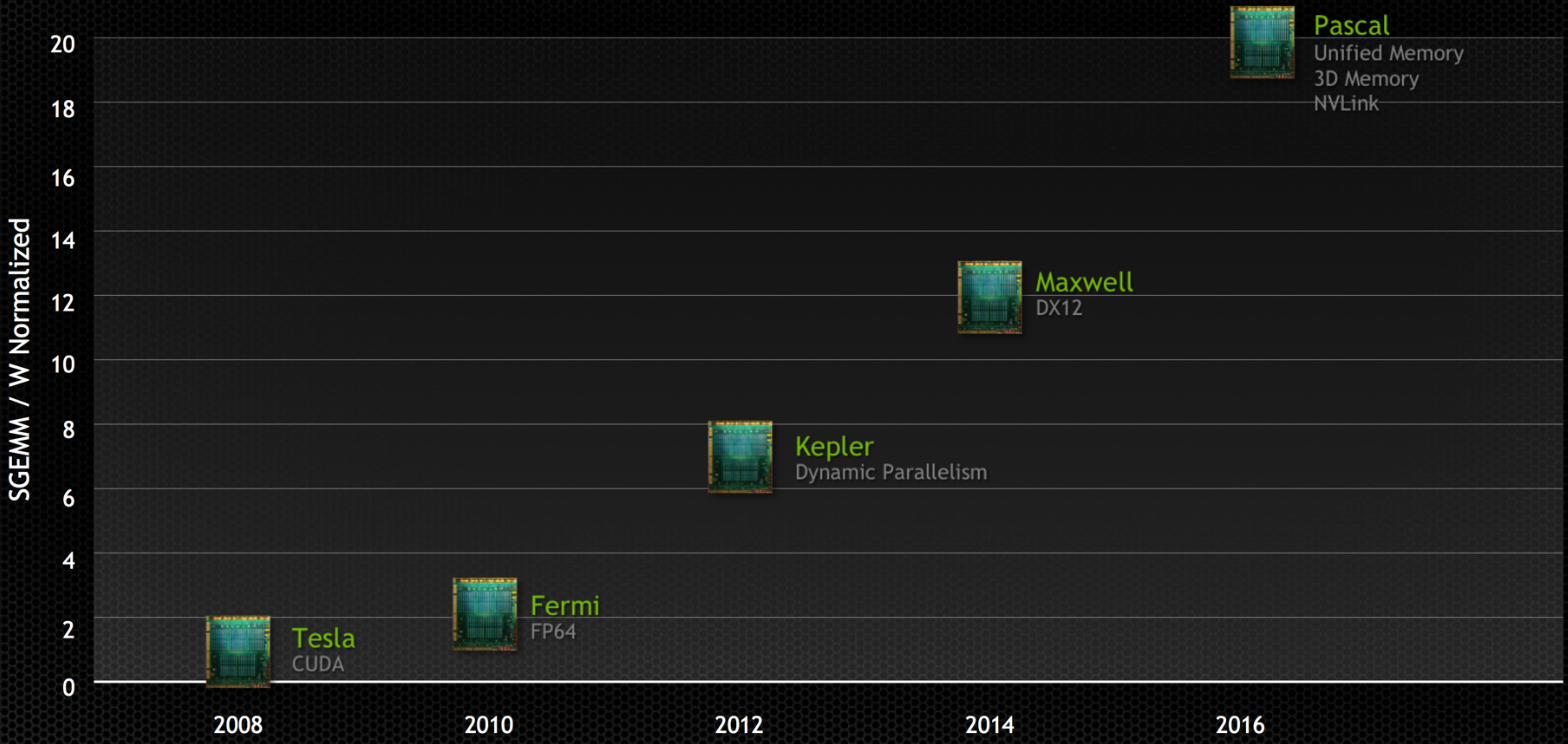
- GPU Computing + QUDA
- Multigrid
- Heterogeneous Multigrid
- Summary

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.



Strong CUDA GPU Roadmap




TITAN

Introducing QUDA

- "QCD on CUDA" – http://lattice.github.com/quda
  - Open source effort with 20+ contributors
- 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 (Lanczos and EigCG) new!
  - Multigrid solvers for optimal convergence new!

Linear Solvers

- QUDA supports a wide range of linear solvers
  - CG, BiCGstab, GCR, Multi-shift solvers, etc.
- As well as domain decomposition preconditioners
  - Additive/Multiplicative Schwarz, overlapping domains
- Together with almost all fermion actions under the sun
  - Wilson, Wilson-clover
  - Twisted mass, degenerate and non degenerate twisted mass
  - Twisted with a clover term
  - HISQ, ASQTAD, naive staggered
  - Domain wall, mobius
- Condition number inversely proportional to mass
  - Light (realistic) masses are highly singular
  - Naive Krylov solvers suffer from critical slowing down at decreasing mass

while ($(|\mathbf{r}_k| > \varepsilon)$) {

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

$\mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k\mathbf{p}_k$

$\mathbf{q}_{k+1} = A\,\mathbf{p}_{k+1}$

$\alpha = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{p}_{k+1}, \mathbf{q}_{k+1})$

$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha\mathbf{q}_{k+1}$

$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\mathbf{p}_{k+1}$

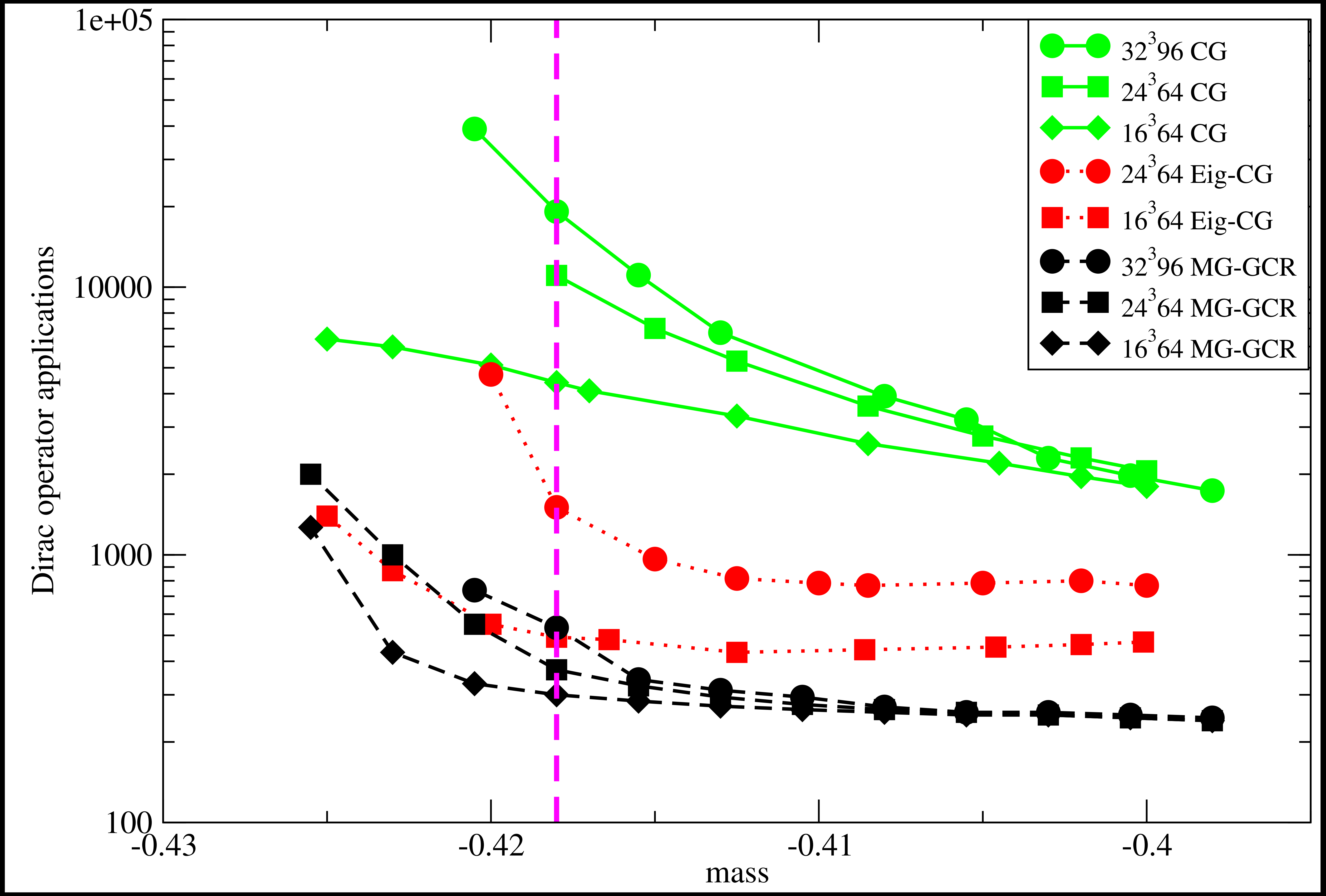
$k = k+1$

}

conjugate gradient

Lattice 2014

Adaptive Geometric Multigrid

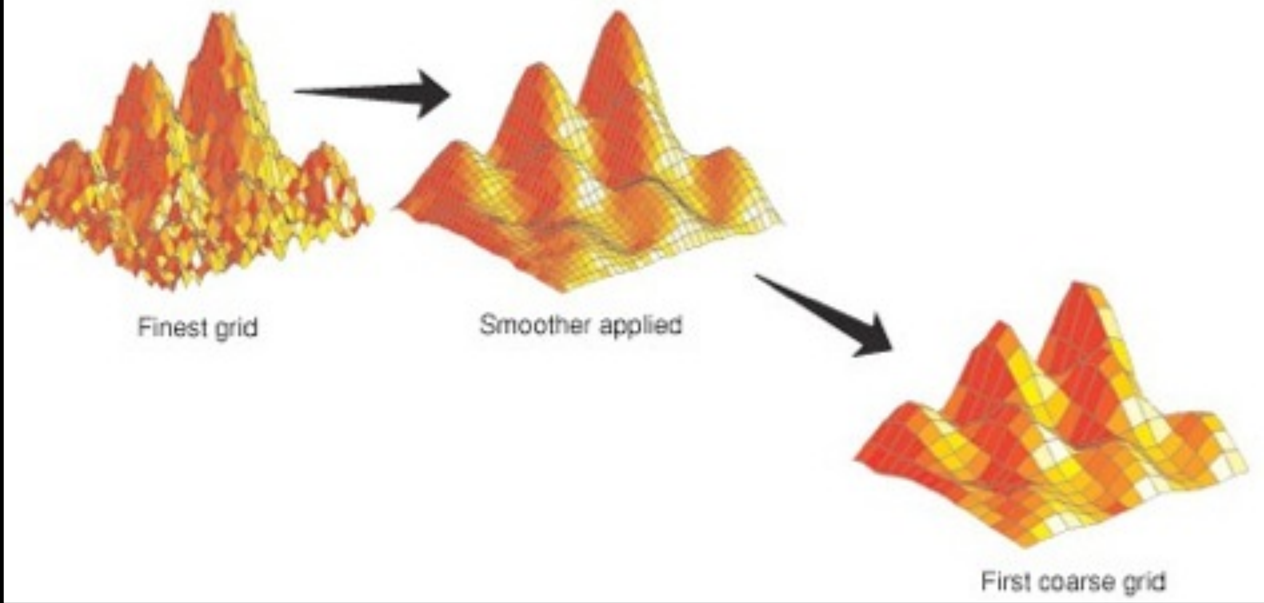


240 vectors

20 vectors

Babich *et al* 2010

Adaptive Geometric Multigrid



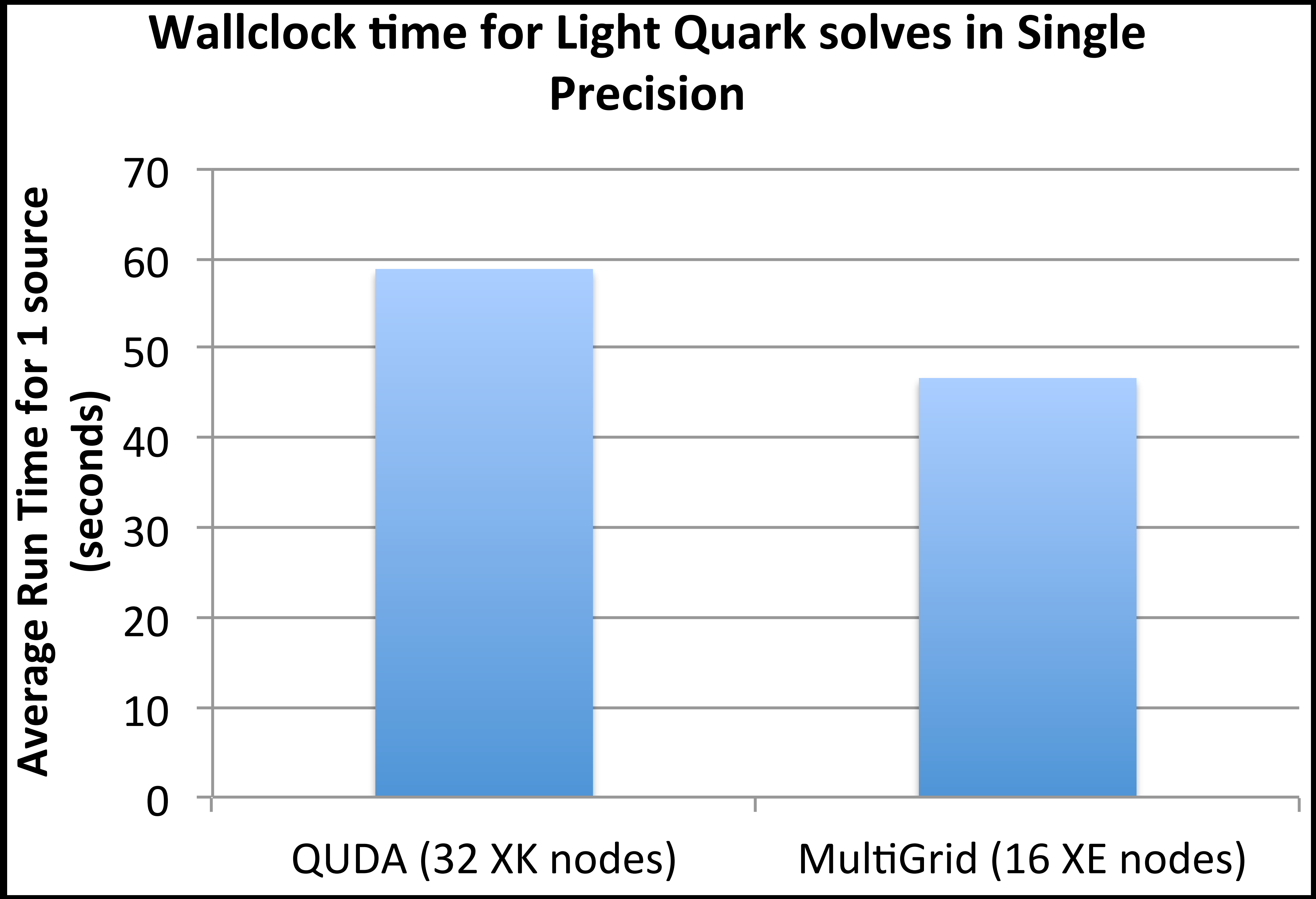
- Adaptively find candidate null-space vectors
  - Dynamically learn the null space and use this to define the prolongator
  - Algorithm is self learning
- Setup
  1. Set solver to be simple smoother
  2. Apply current solver to random vector $v_i = P(D)\, \eta_i$
  3. If convergence good enough, solver 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

Hierarchical algorithms for LQCD

- Adaptive Geometric Multigrid for LQCD
  - Based on adaptive smooth aggregation (Brezina *et al* 2004)
  - Low modes have weak-approximation property => locally co-linear
  - Apply fixed geometric coarsening (Brannick *et al* 2007, Babich *et al* 2010)
- Clover Multigrid (Osborn *et al* 2010)
  - Apply multigrid to the even/odd system
- Domain decomposition multigrid (Frommer *et al* 2012)
  - Use Schwarz Alternating Procedure as smoother for improved scalability
- Inexact Deflation (Lüscher 2007)
  - Equivalent to adaptive “unsmoothed” aggregation
  - Local coherence = Weak-approximation property
  - Uses an additive correction vs. MG’s multiplicative correction
- Domain-wall Multigrid / Deflation (Cohen et al 2012, Boyle 2013)
  - Apply to normal operator for positivity

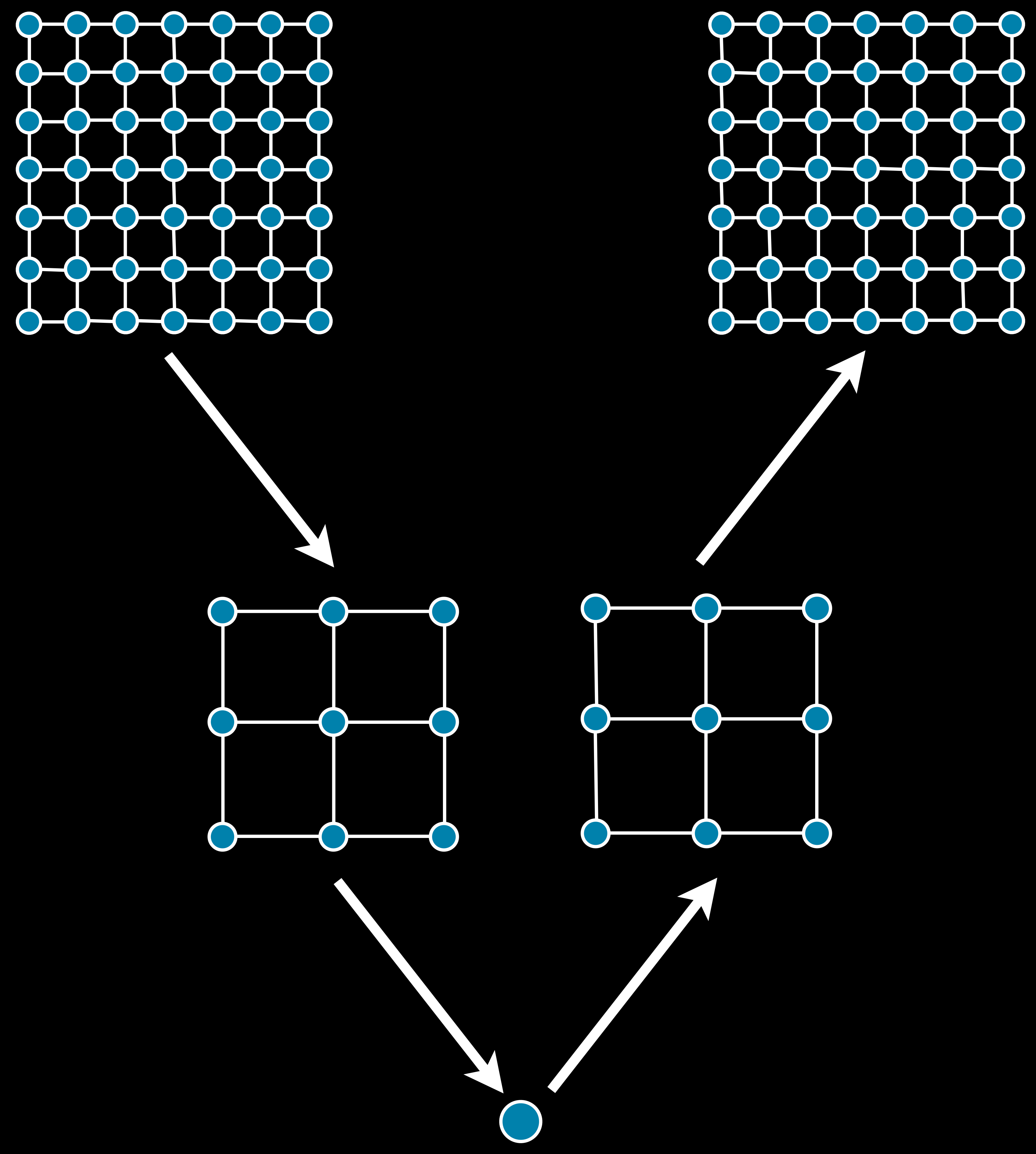
Motivation

- A CPU running the optimal algorithm can surpass a highly tuned GPU naive algorithm
- For competitiveness, MG on GPU is a must
- Seek *multiplicative gain* of architecture and algorithm
- Multigrid speedup expected to be > 10x



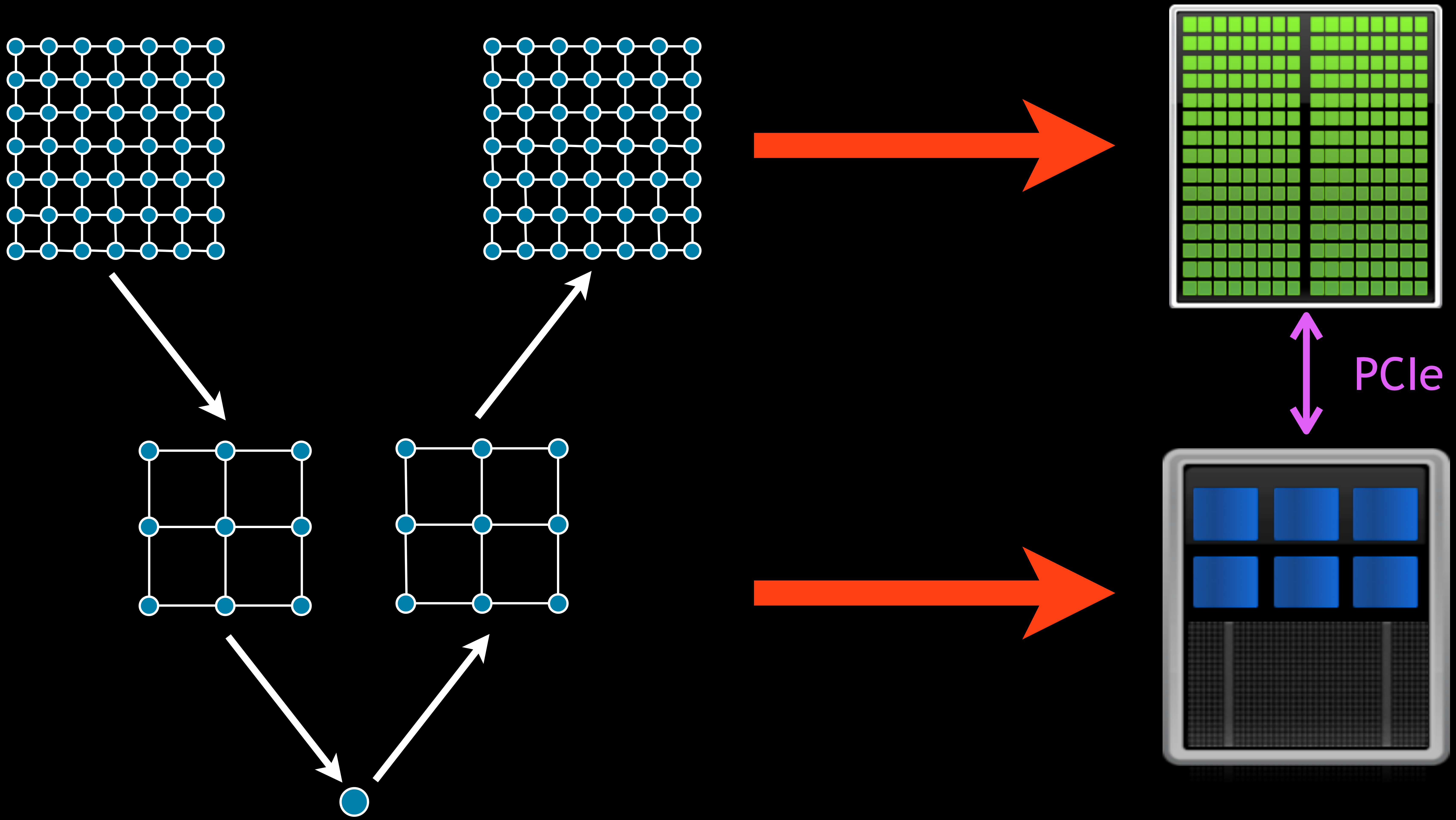
Chroma propagator benchmark
Figure by Balint Joo
MG Chroma integration by Saul Cohen
MG Algorithm by James Osborn

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



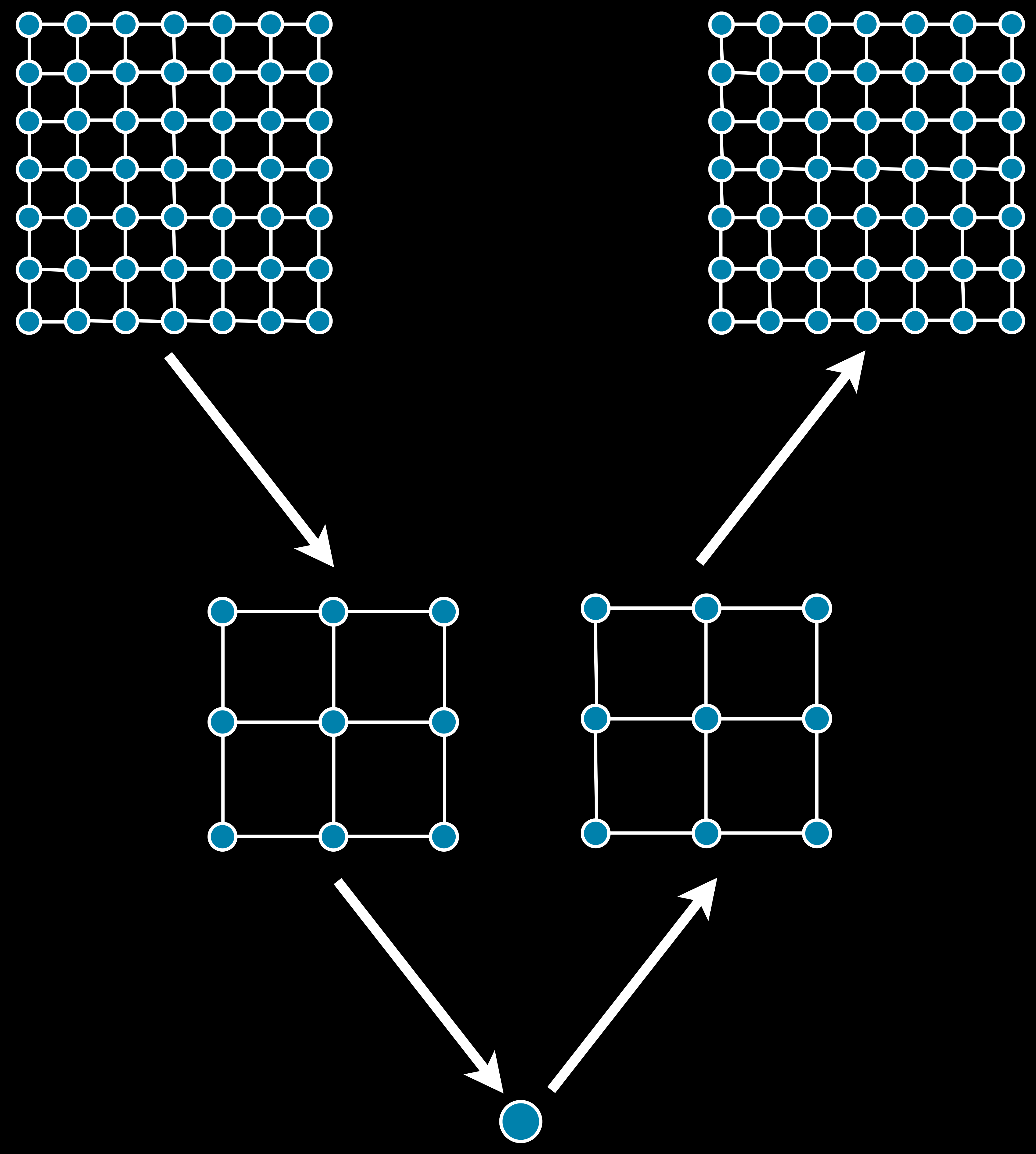
GPU

Thousands of cores
for parallel processing

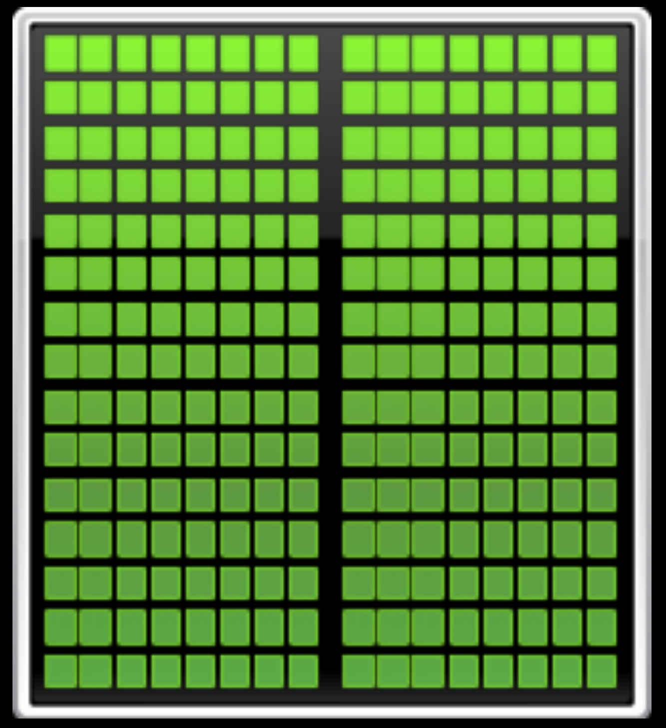
CPU

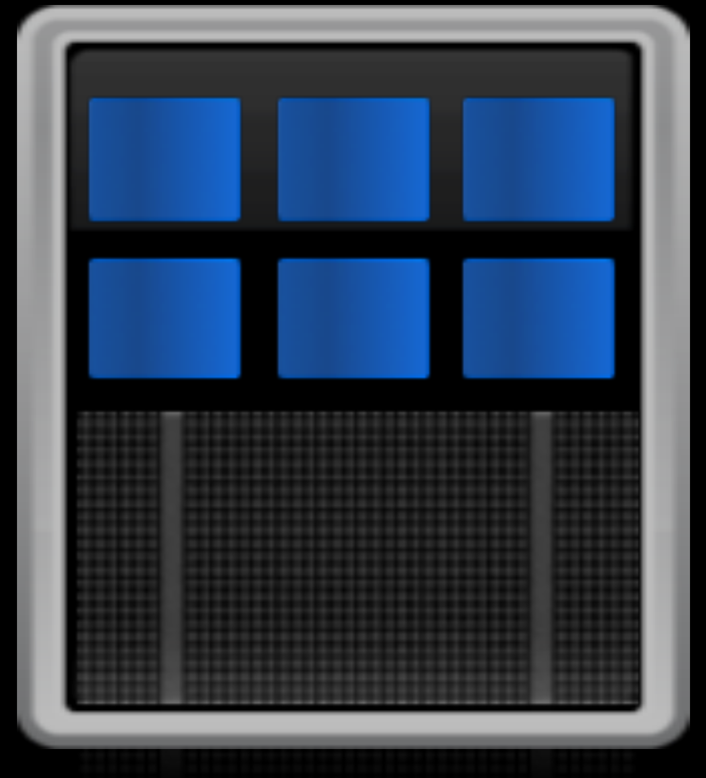
Few Cores optimized
for serial work

Heterogeneous Updating Scheme

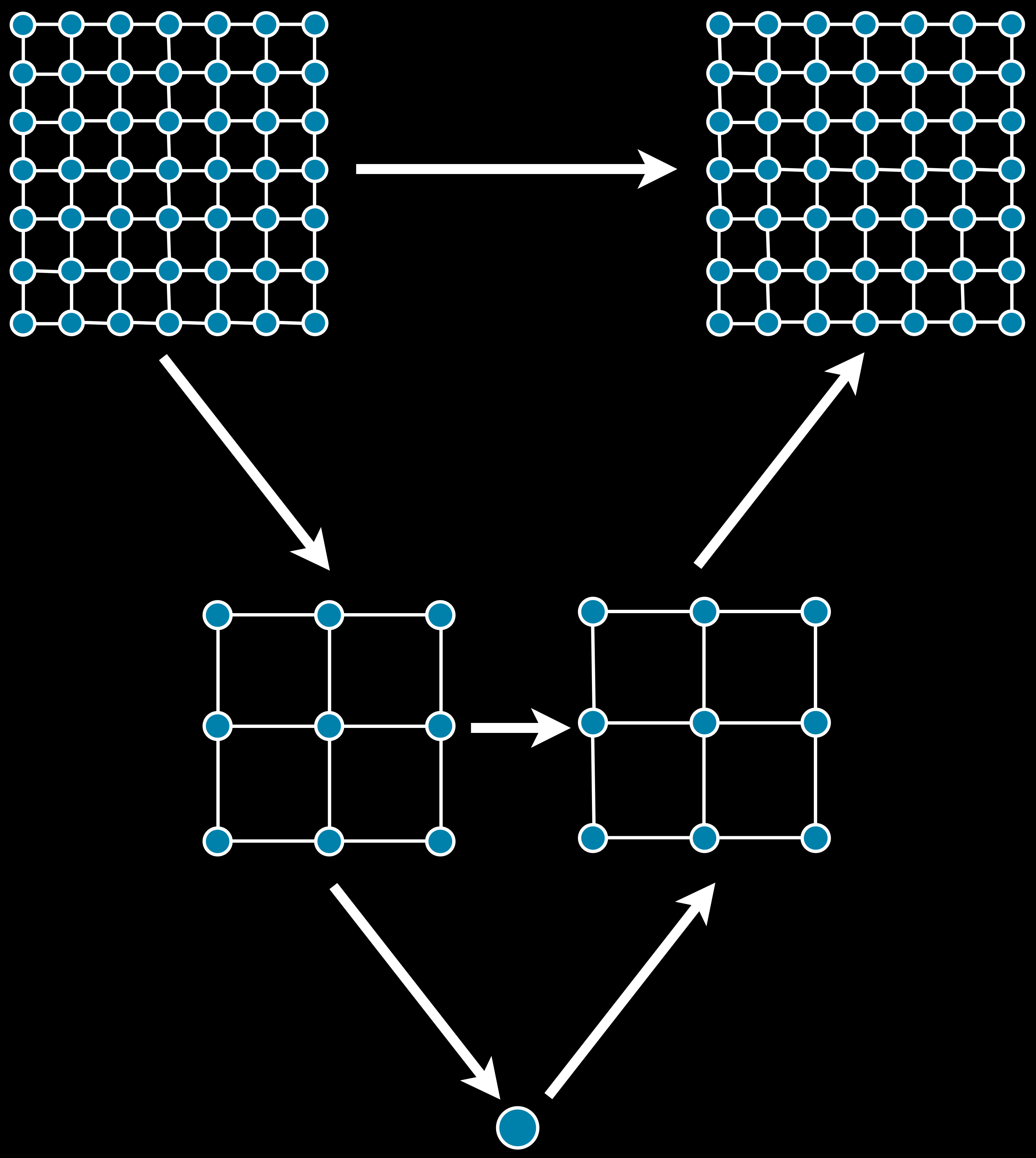


- Multiplicative MG is necessarily serial process
  - Cannot utilize both GPU and CPU simultanesouly

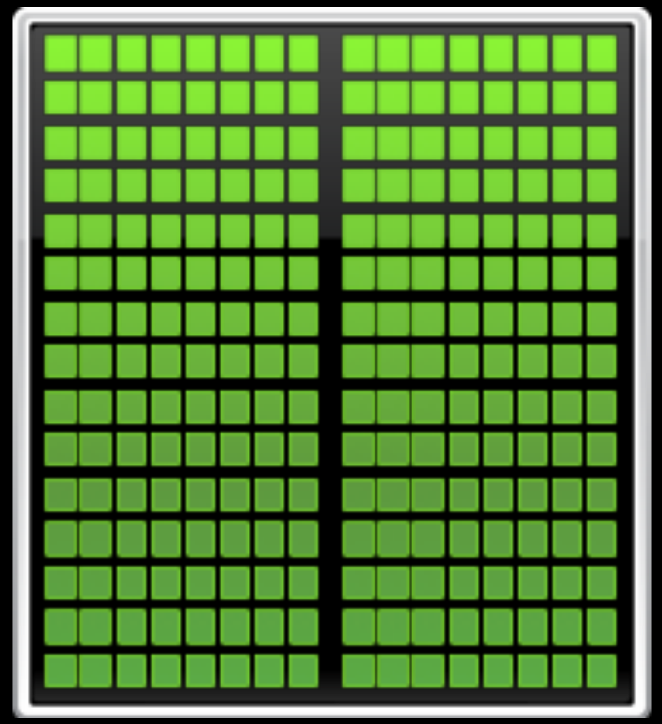


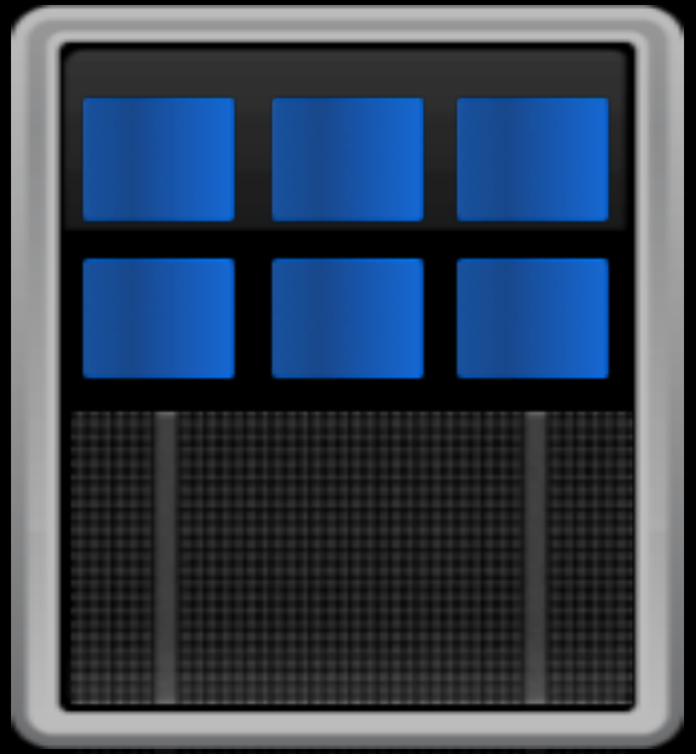


Heterogeneous Updating Scheme



- 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
- Heterogeneous Multigrid may actually *improve* strong scaling
  - Already doing DD preconditioner
  - Coarse-grid correction is almost free



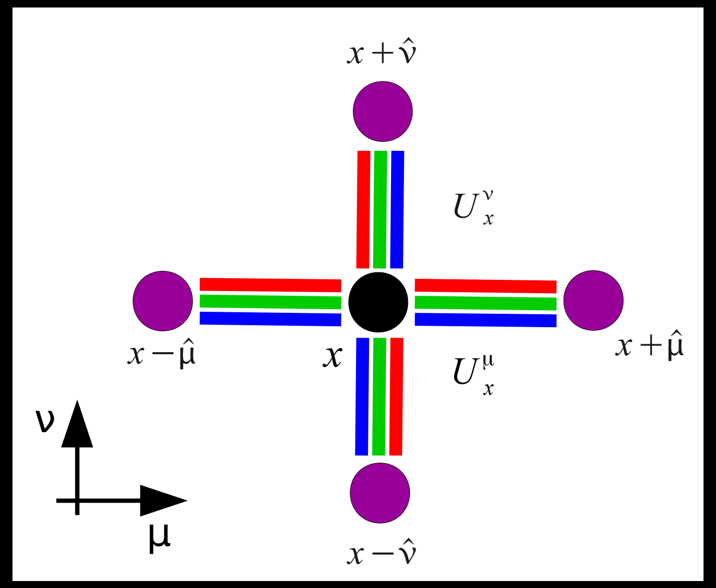


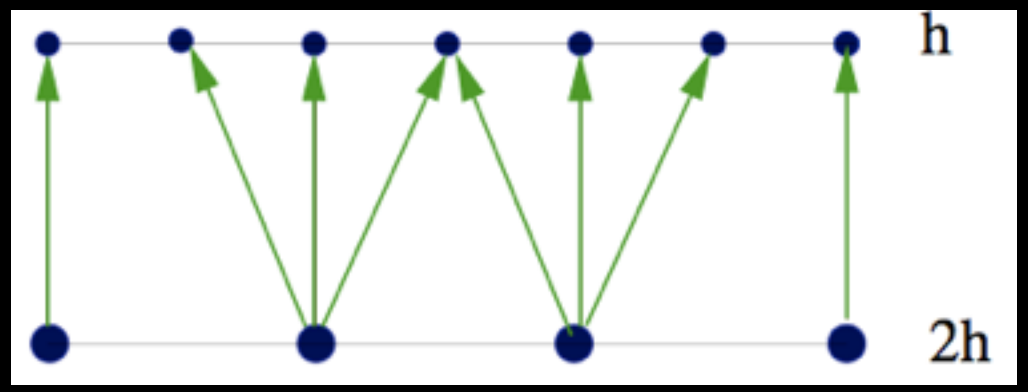
Design Goals

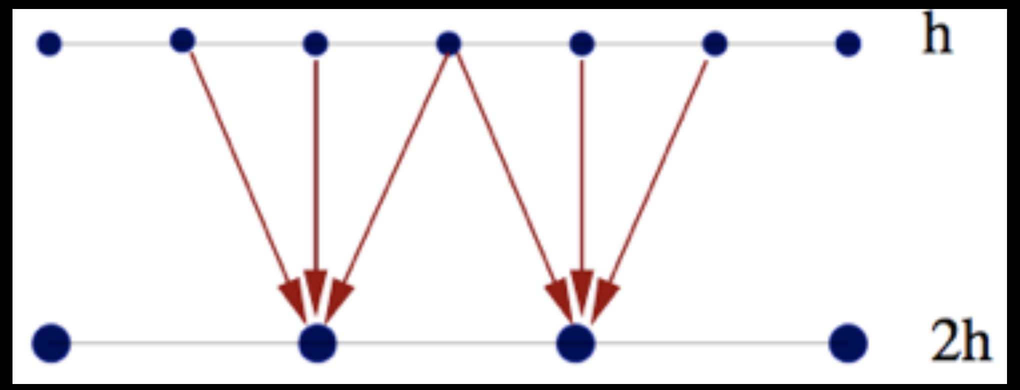
- Performance
  - LQCD typically reaches high % peak peak performance
  - Brute force can beat the best algorithm
  - Multigrid must be optimized to the same level
- Flexibility
  - Deploy level *i* on either CPU or GPU
  - All algorithmic flow decisions made at runtime
  - Autotune for a given *heterogeneous* 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 implementing new algorithms

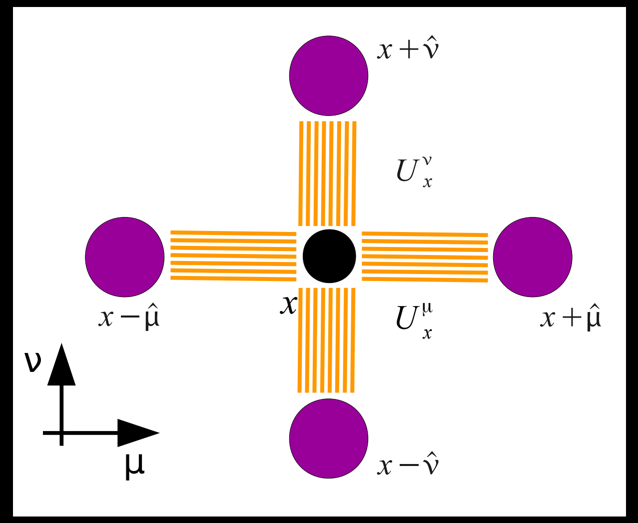
Ingredients for Parallel Adaptive Multigrid

- 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

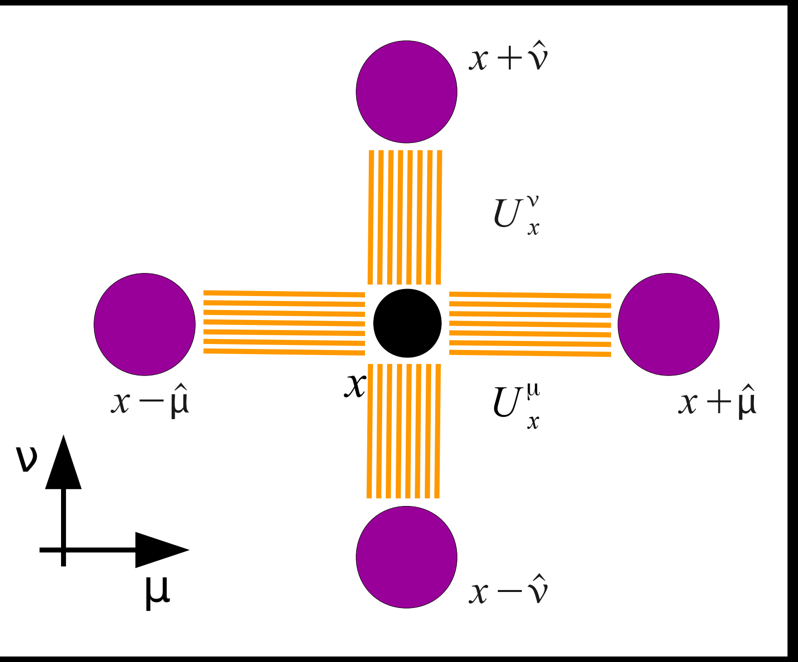








Parallel Implementation



- Coarse operator looks like a Dirac operator
  - Link matrices have dimension N_v x N_v (e.g., 24 x 24)

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

- 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

```
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);
}
```

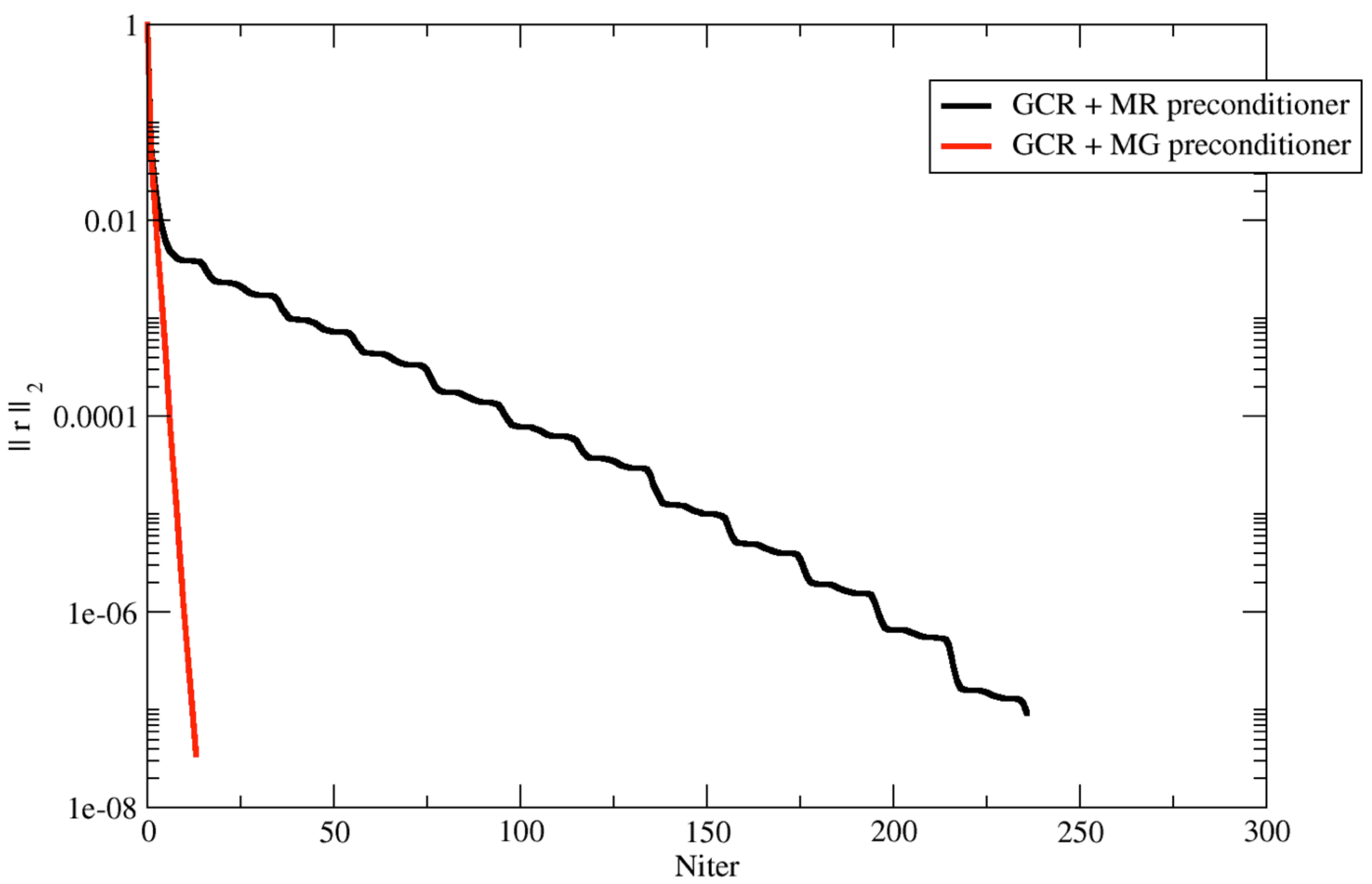
platform specific parallelization
GPU: shared memory
CPU: OpenMP, vectorization

```
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

Lattice 2014



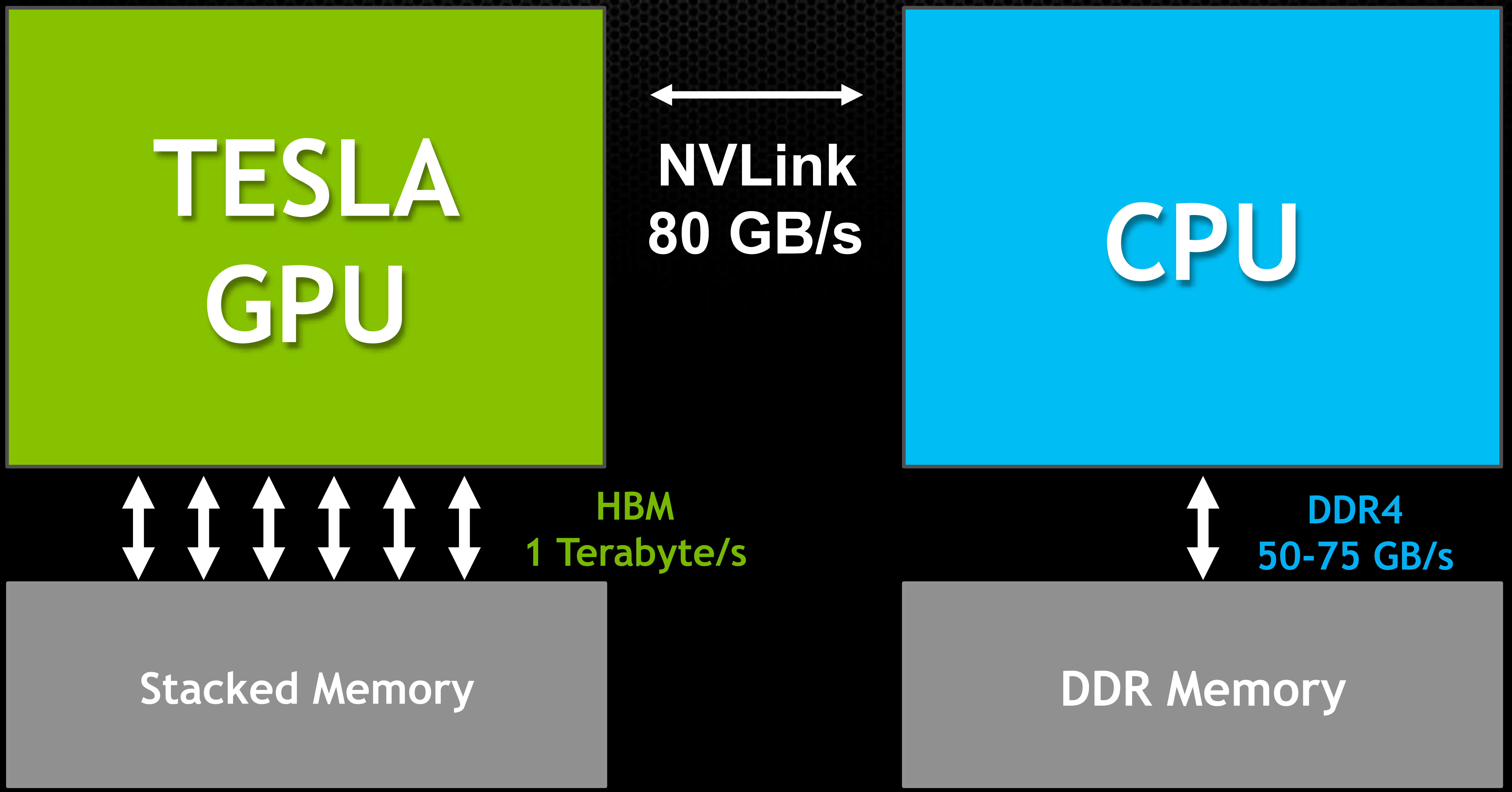
Current Status

- Wilson multigrid fully numerically verified
  - Consistent with results from QCDMG (Babich *et al* 2010)
- Framework still slow
  - Host code not optimized at all
  - 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
  - Spin coarsening strategies and use of Laplace modes
  - Coarse-grid solvers (direct vs. indirect)
  - Staggered multigrid
  - Comparison of traditional versus *heterogeneous update*
- Real goal is developing asynchronous solvers for future heterogeneous architectures

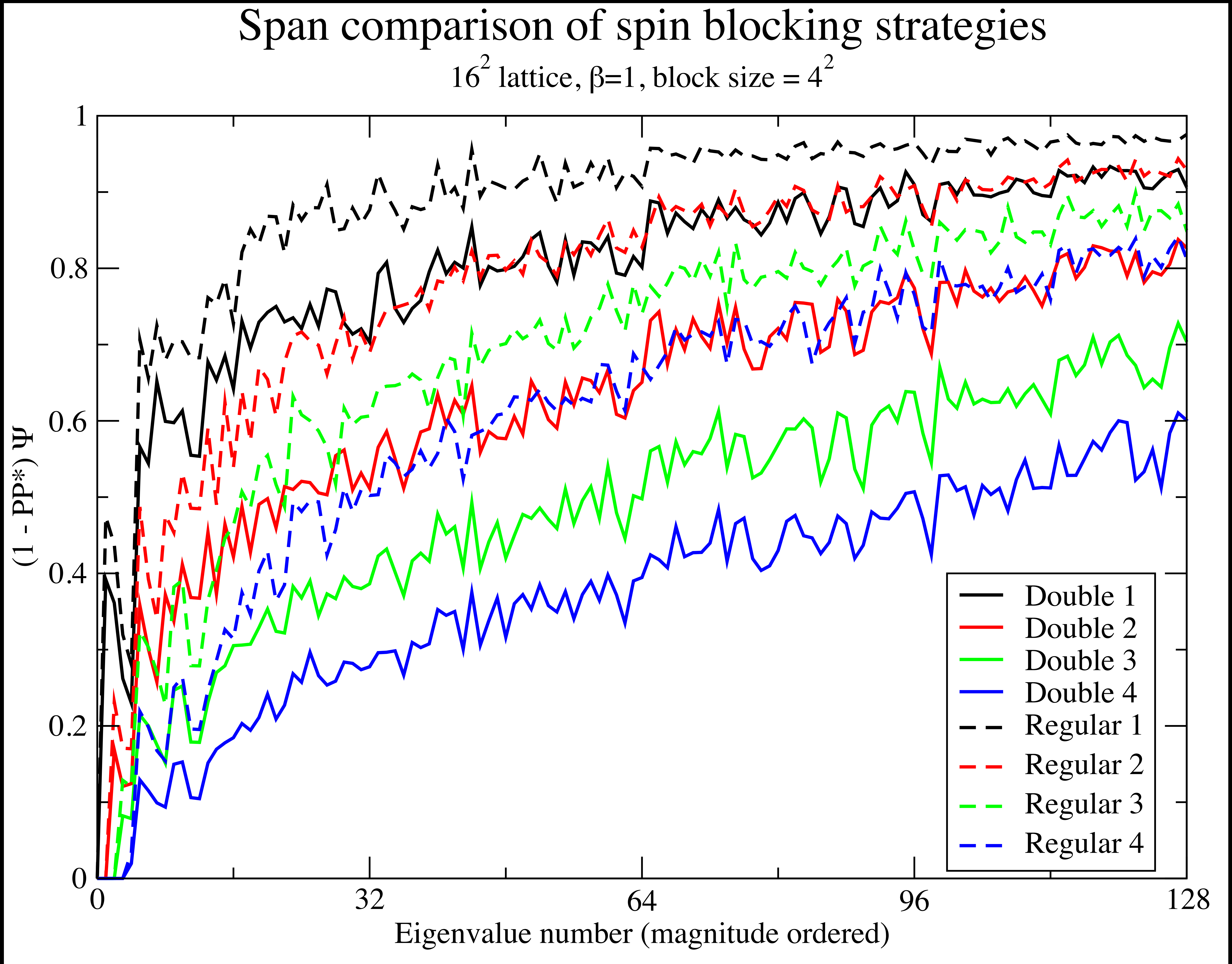
Heterogeneous Computing in 2016



Summary

- Overview of Multigrid in QUDA project
- Framework essentially complete (barring clover)
- Efforts now focussed on optimization
- Then can *finally* return to numerics
- Hierarchical *and* heterogeneous algorithm research toolbox
  - Aim for scalability *and* optimality
- Lessons today are relevant for future architecture preparation

Lattice 2014



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?

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++) {
    …
     }
   }
 }
```

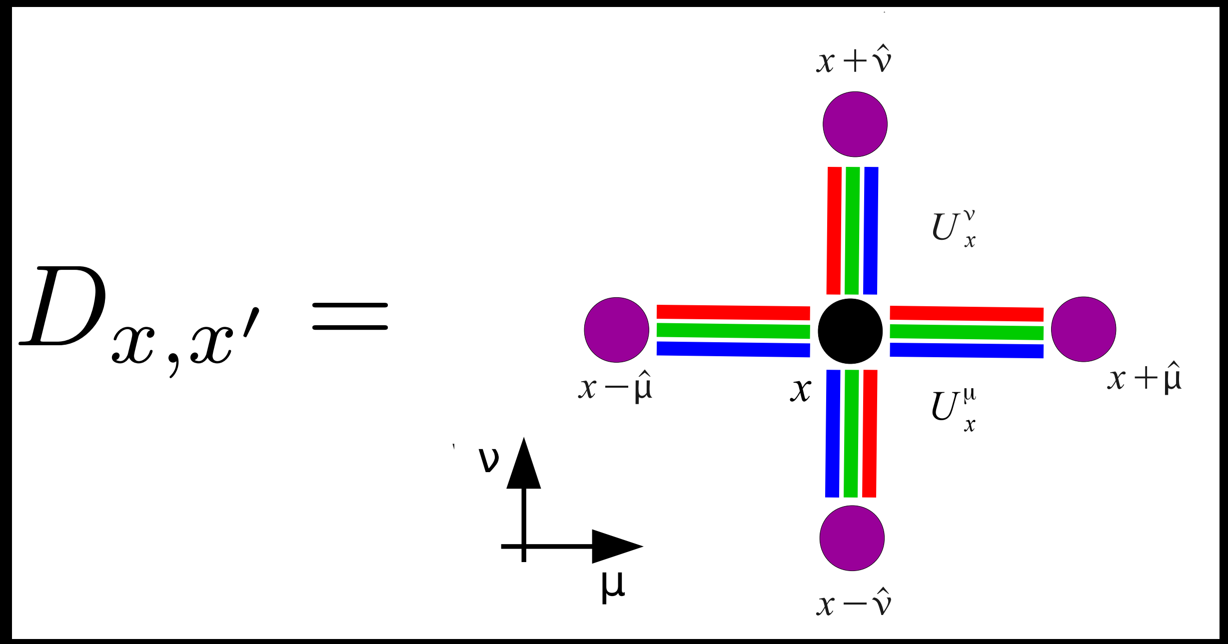
The compilation problem…

- 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 compilation will fix this

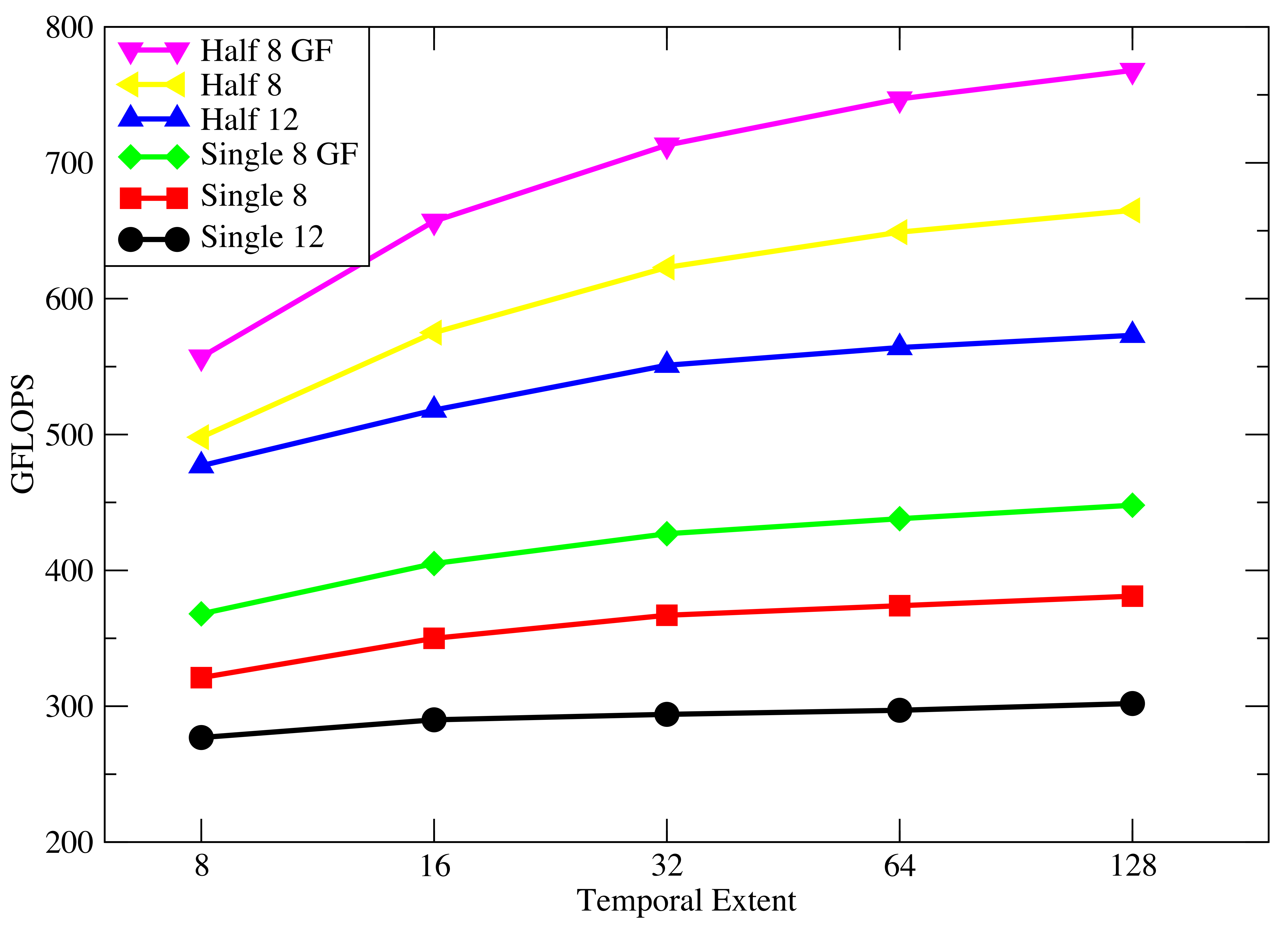
Mapping the Dirac operator to CUDA



- 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)
  - 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)

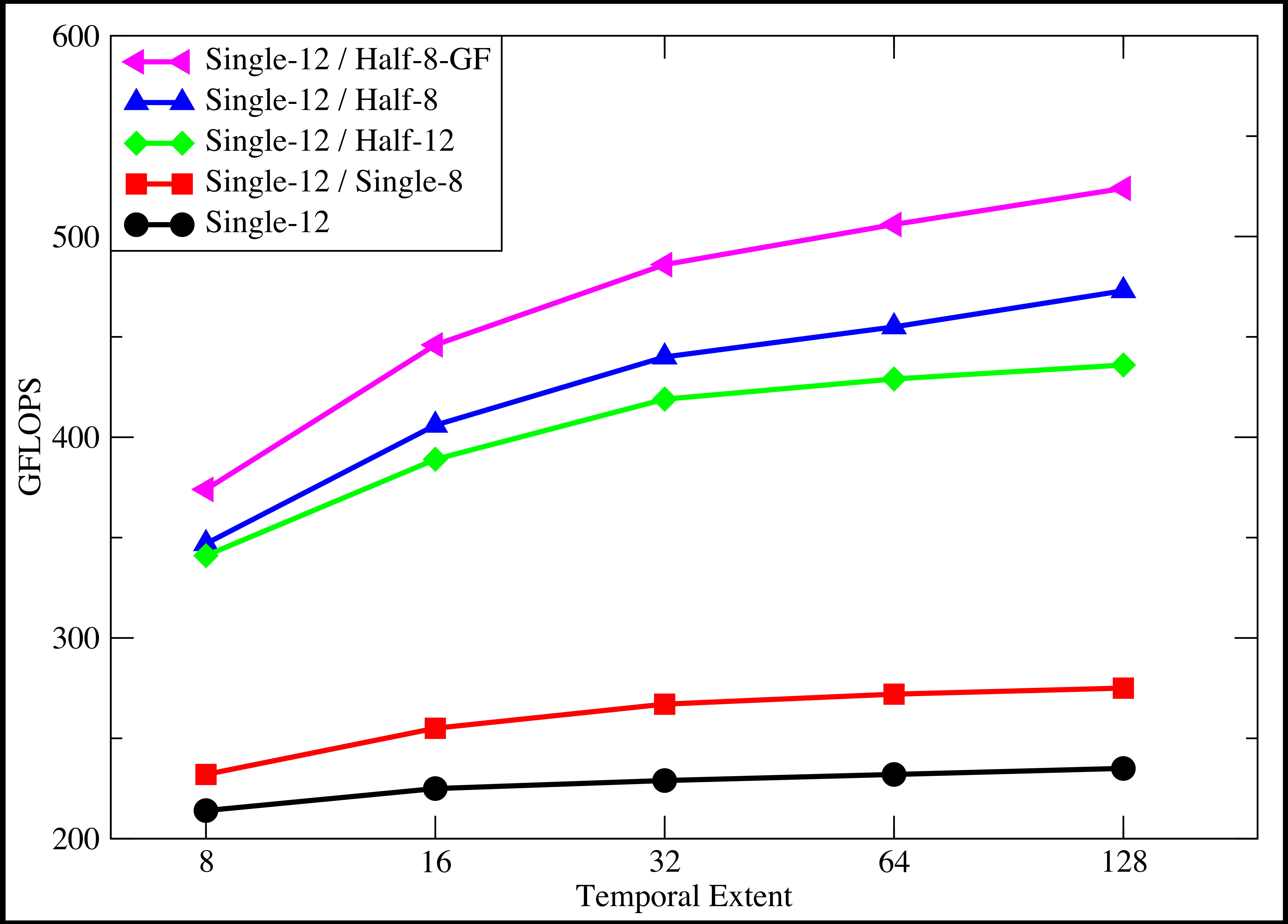
Tesla K20X	
Gflops	3995
GB/s	250
AI	16

Kepler Wilson-Dslash Performance



Wilson Dslash
K20X performance
$V = 24^3 x T$

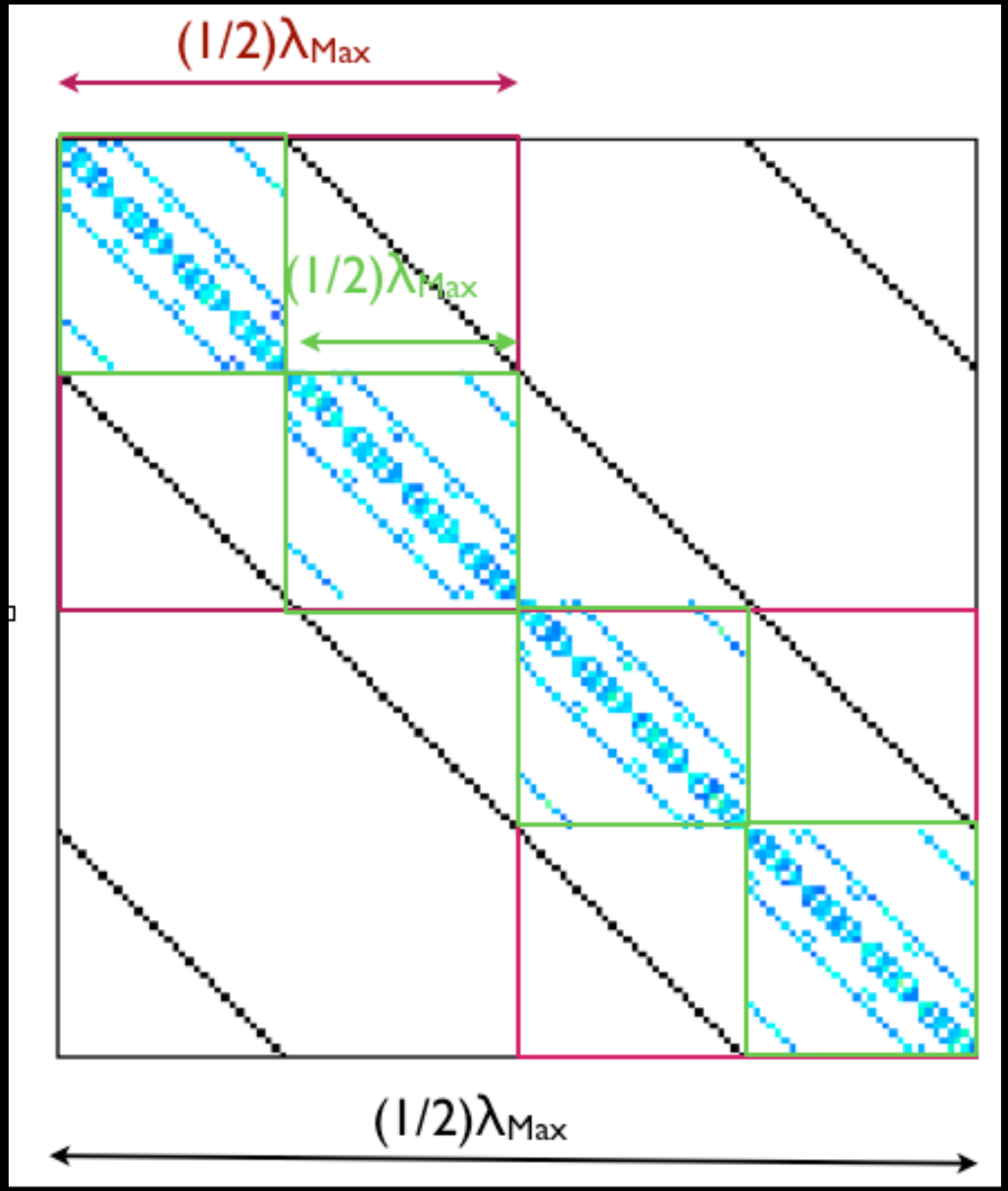
Kepler Wilson-Solver Performance



Wilson CG
K20X performance
$V = 24^3 x T$

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



Strong Scaling Chroma with DD

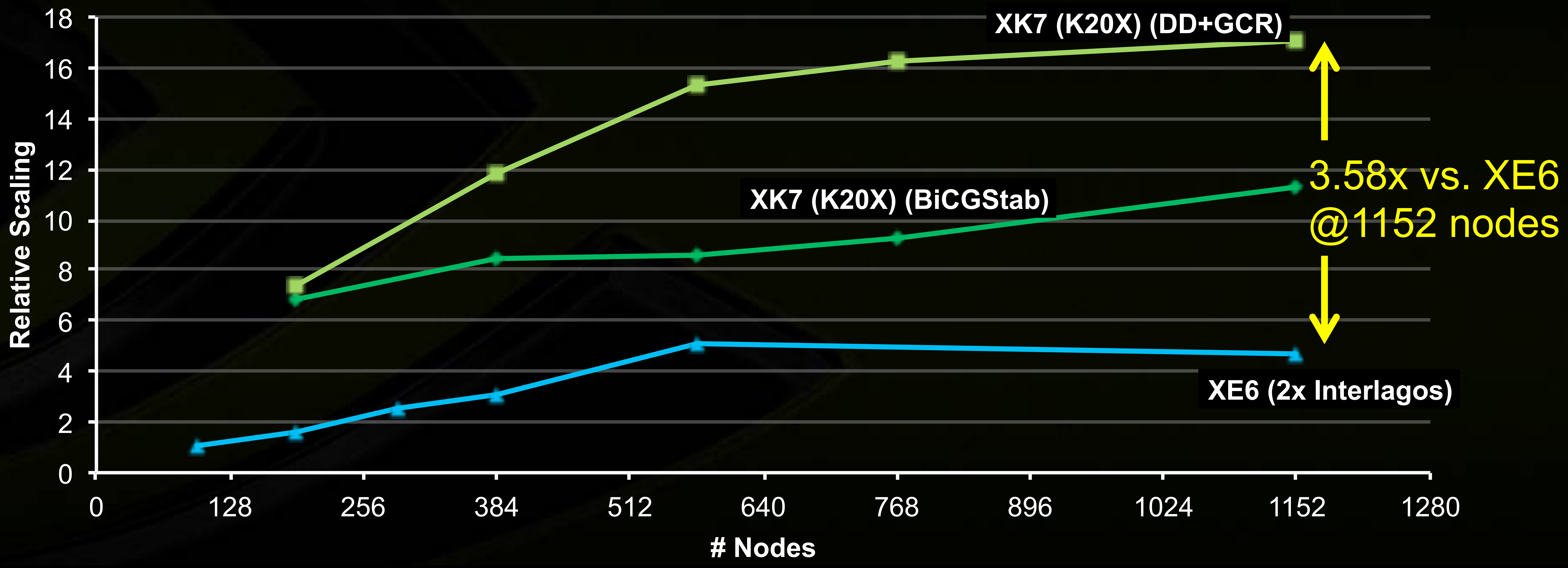
Chroma

48^3x512 lattice

Relative Scaling (Application Time)

"XK7" node = XK7 (1x K20X + 1x Interlagos)

"XE6" node = XE6 (2x Interlagos)



Lattice 2014

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

Strong GPU Roadmap