QCDNA 2014

GPU ALGORITHMS FOR LATTICE QCD

M Clark
NVIDIA

QCDNA 2014

Contents

- GPU Computing
- QUDA Library
- Mixed-Precision Solvers
- Strong scaling algorithms
- Eigenvectors Solvers
- Multigrid
- Summary

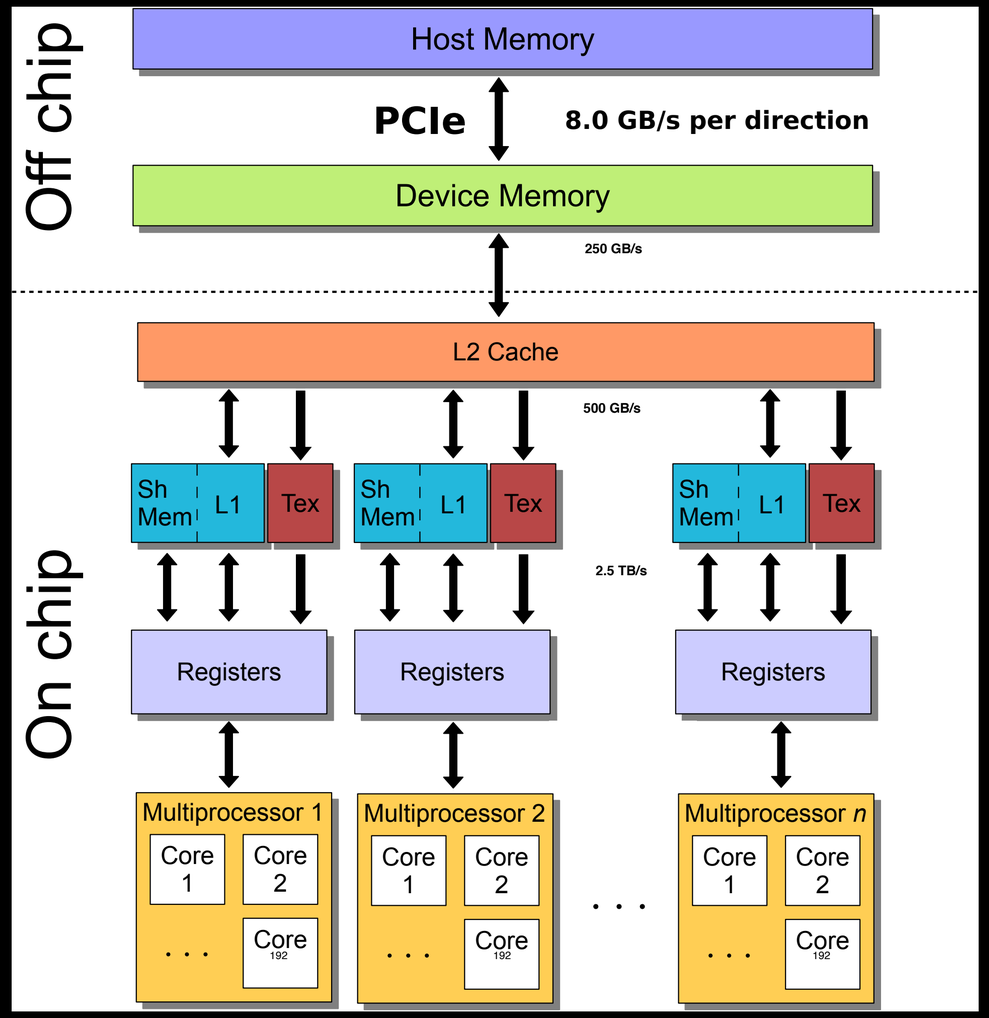
From this…

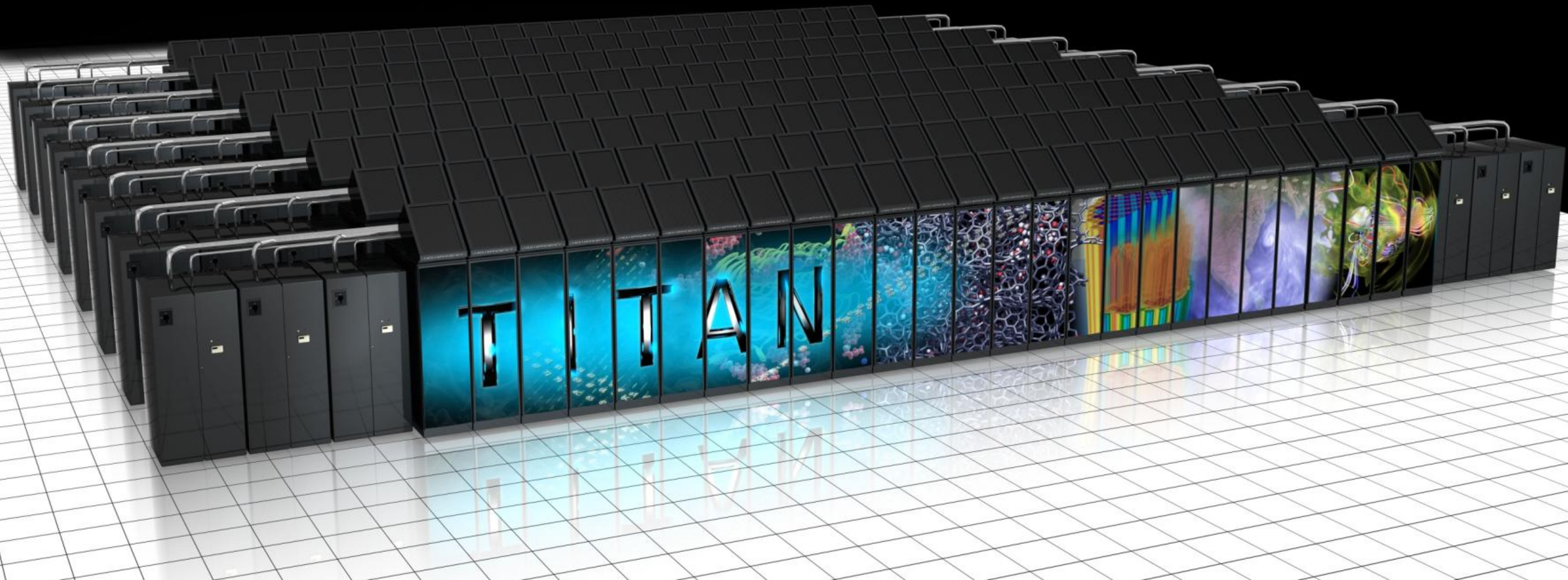
...to this

QCDNA 2014

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

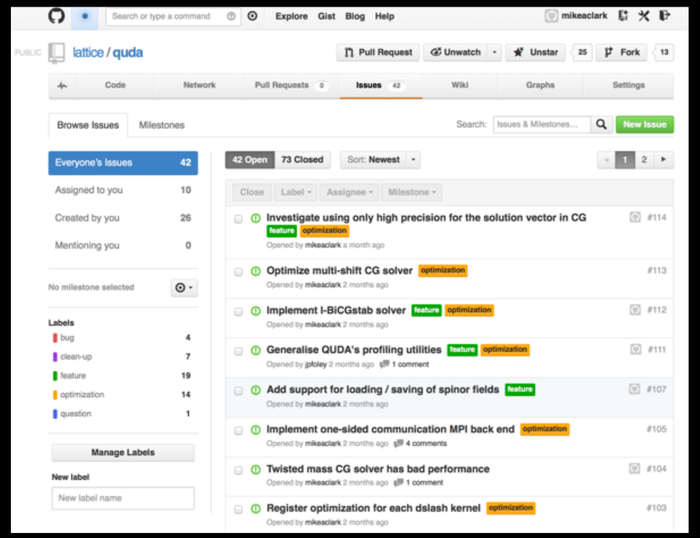
Enter QUDA

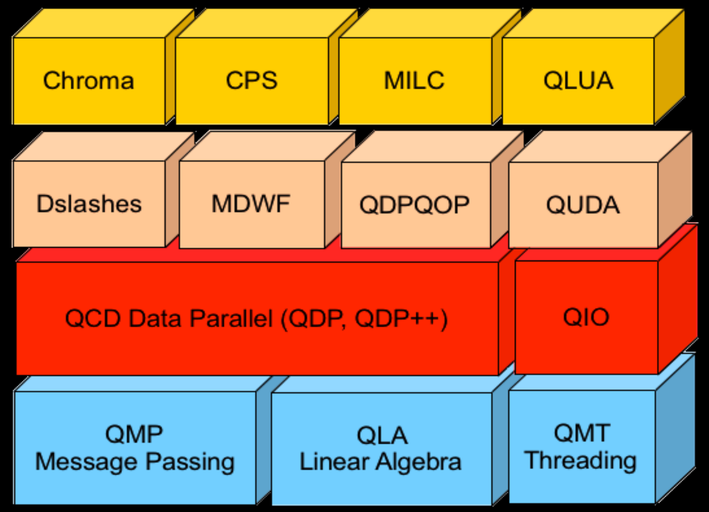
- "QCD on CUDA" – http://lattice.github.com/quda
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, etc.
- Provides:
  - Various solvers for all major fermonic discretizations, with multi-GPU support
  - Additional performance-critical 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!

QCDNA 2014

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)

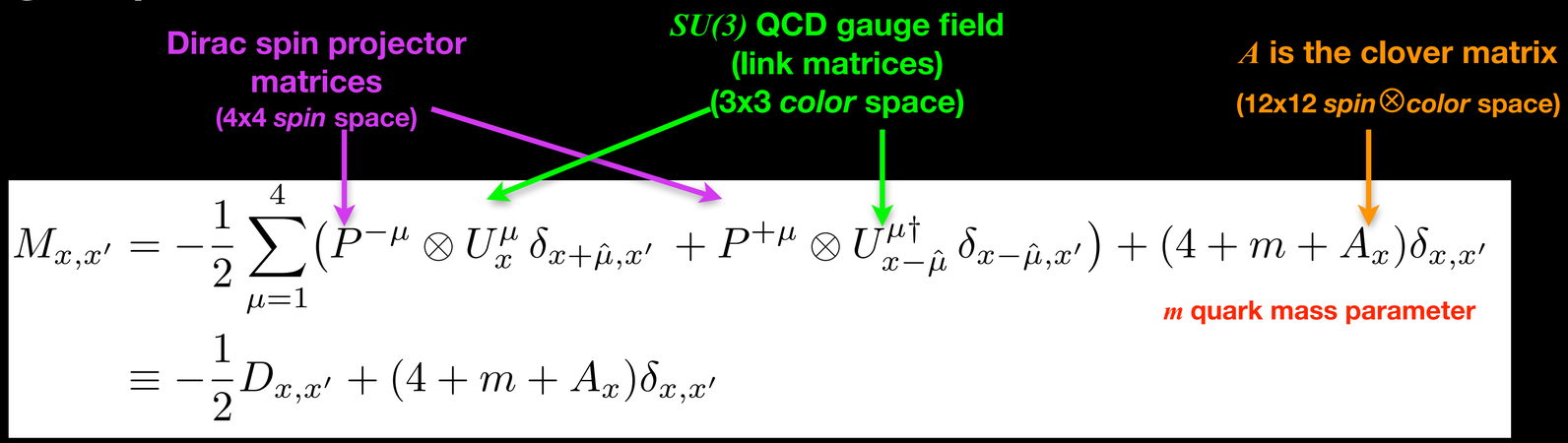




QCDNA 2014

The Dirac Operator

- Quark interactions are described by the Dirac operator
  - First-order PDE acting with a background field
  - Large sparse matrix



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

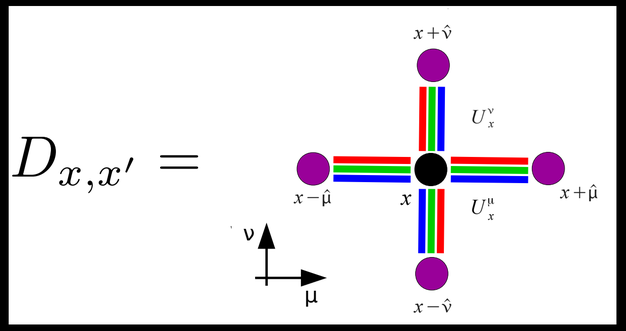
$$\equiv -\frac{1}{2}D_{x,x'} + (4+m+A_x)\delta_{x,x'}$$

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

QCDNA 2014

Mapping the Dirac operator to CUDA

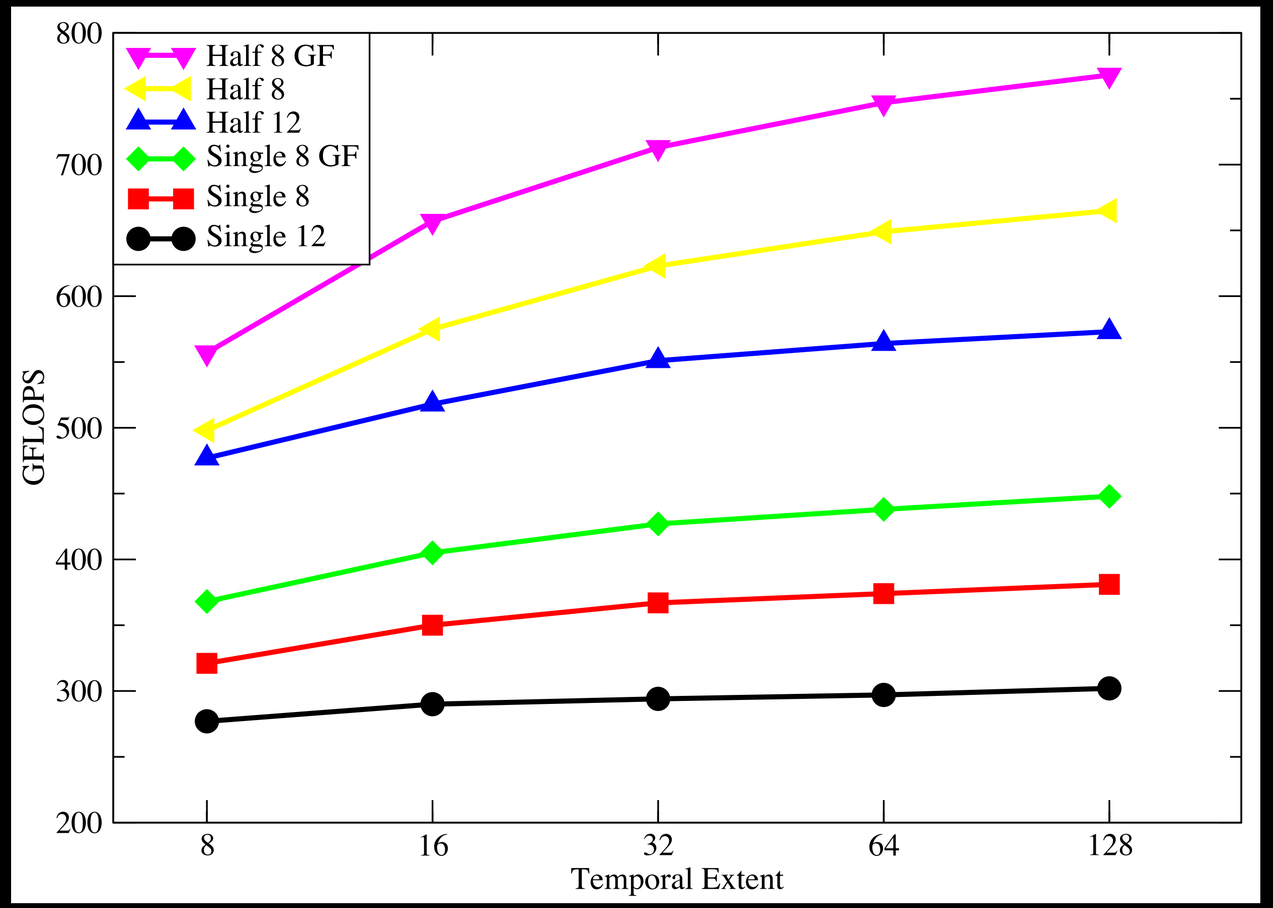
- Finite difference operator in LQCD is known as Dslash
- Assign a single space-time point to each thread
  - V = XYZT threads, e.g., V = 24^4 => $3.3\text{x}10^6$ threads
- Looping over direction each thread must
  - Load the neighboring spinor (24 numbers x8)
  - Load the color matrix connecting the 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

QCDNA 2014

Kepler Wilson-Dslash Performance



Wilson Dslash
K20X performance
$V = 24^3xT$

QCDNA 2014

Linear Solvers

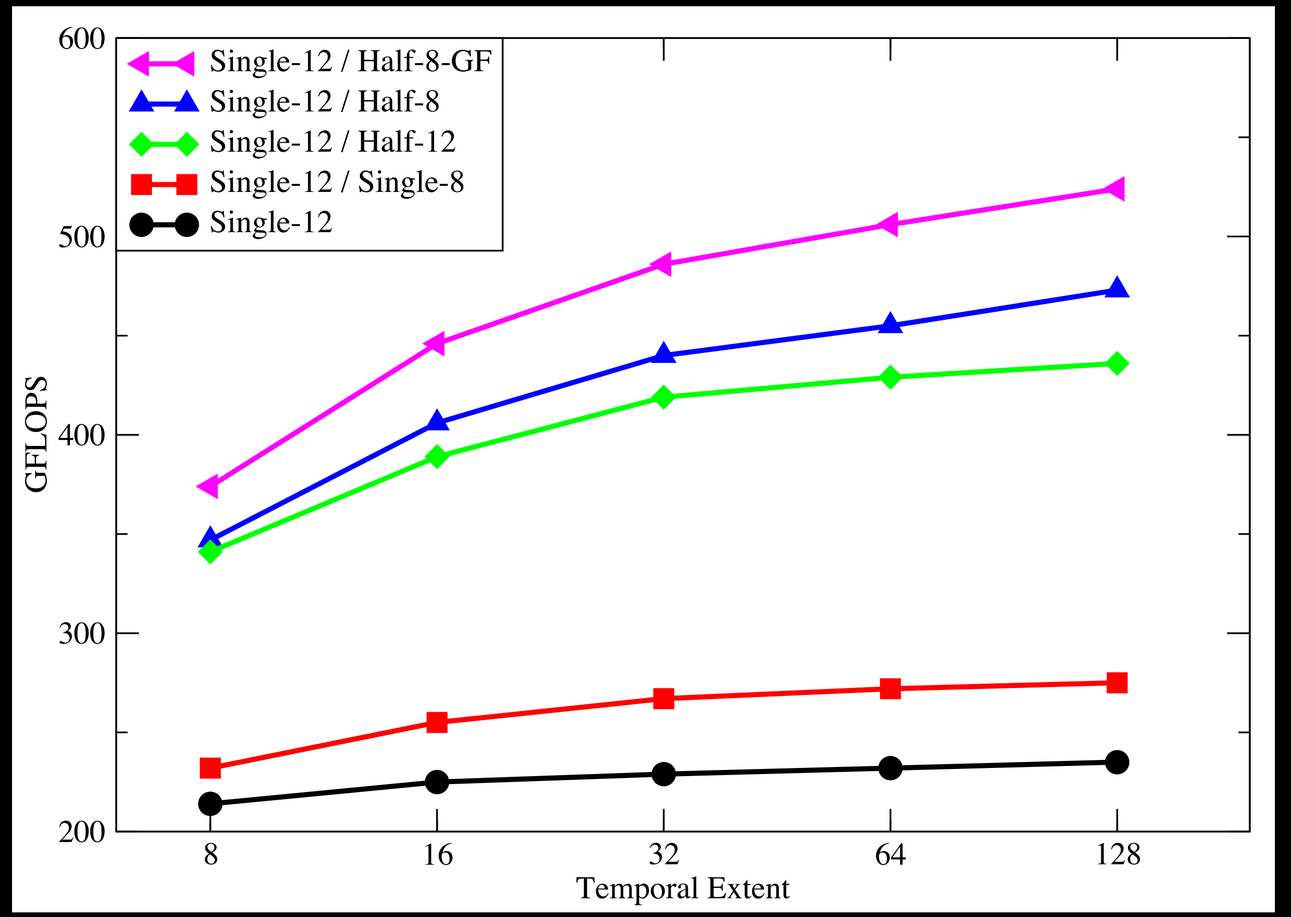
- Nature of eigen-spectrum constrains which solver choice
  - CGNE / CGNR
  - BiCGstab
  - GMRES
- Condition number inversely proportional to mass
  - Light (realistic) masses are highly singular
- Entire solver algorithm must run on GPUs
  - Time-critical kernel is the stencil application (SpMV)
  - Also require BLAS level-1 type operations
- BLAS is becoming the Amdahl's law of naive linear solvers
  - Global sums are expensive
  - BLAS are bandwidth bound
  - Can rectify through e.g., s-step methods and/or preconditioners

$$
\begin{aligned}
&\text{while } (|\mathbf{r}_k| > \varepsilon) \ \{ \\
&\quad \beta_k = (\mathbf{r}_k, \mathbf{r}_k)/(\mathbf{r}_{k-1}, \mathbf{r}_{k-1}) \\
&\quad \mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k \mathbf{p}_k \\
&\quad \mathbf{q}_{k+1} = A\,\mathbf{p}_{k+1} \\
&\quad \alpha = (\mathbf{r}_k, \mathbf{r}_k)/(\mathbf{p}_{k+1}, \mathbf{q}_{k+1}) \\
&\quad \mathbf{r}_{k+1} = \mathbf{r}_k - \alpha \mathbf{q}_{k+1} \\
&\quad \mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \mathbf{p}_{k+1} \\
&\quad k = k+1 \\
&\}
\end{aligned}
$$

conjugate gradient

QCDNA 2014

Kepler Wilson-Solver Performance



Wilson CG
K20X performance
$V = 24^3 \times T$

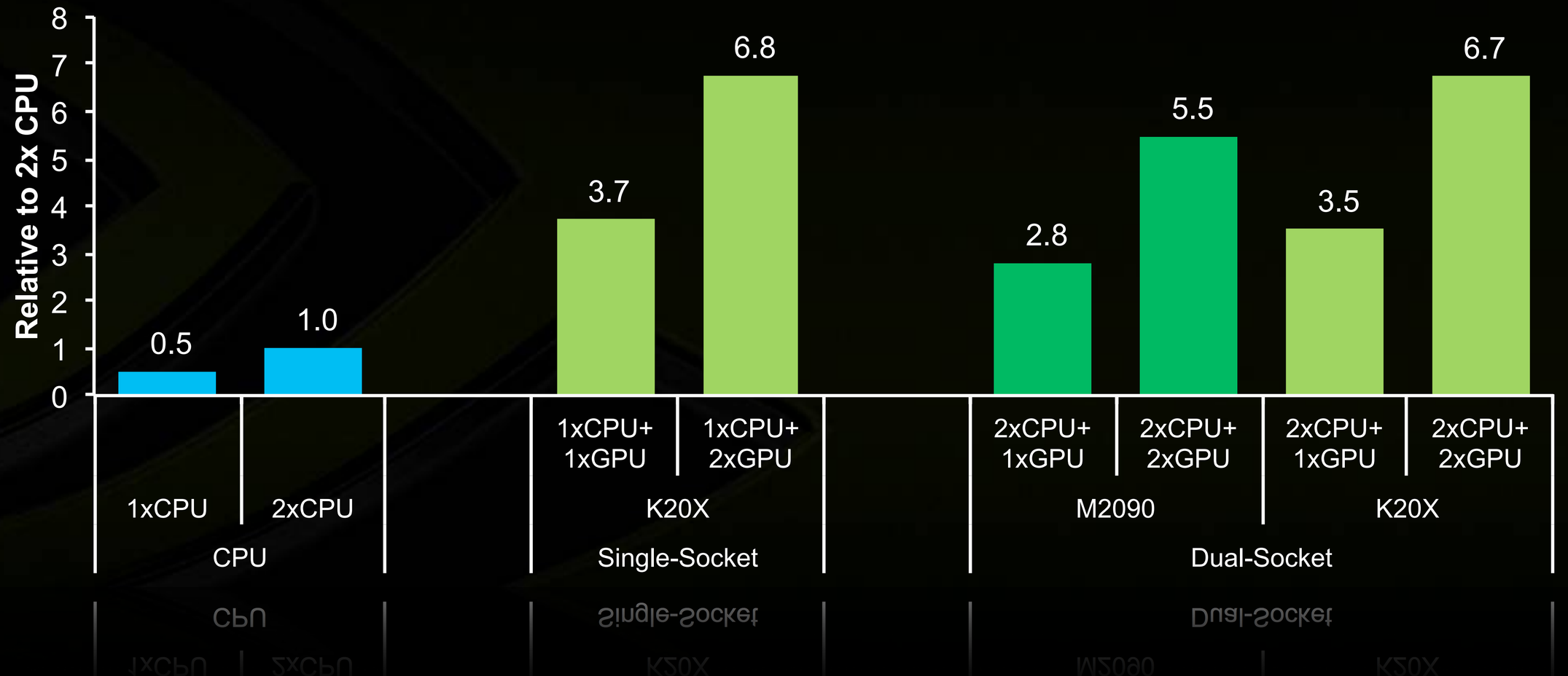
QCDNA 2014

Chroma Benchmark with QUDA

Chroma

24^3x128 lattice

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



QCDNA 2014

Mixed-precision solvers

- QUDA has had mixed-precision from the get go
- Almost a free lunch where it works well (wilson/clover)
  - Residual injection / reliable updates mixed-precision BiCGstab
  - 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

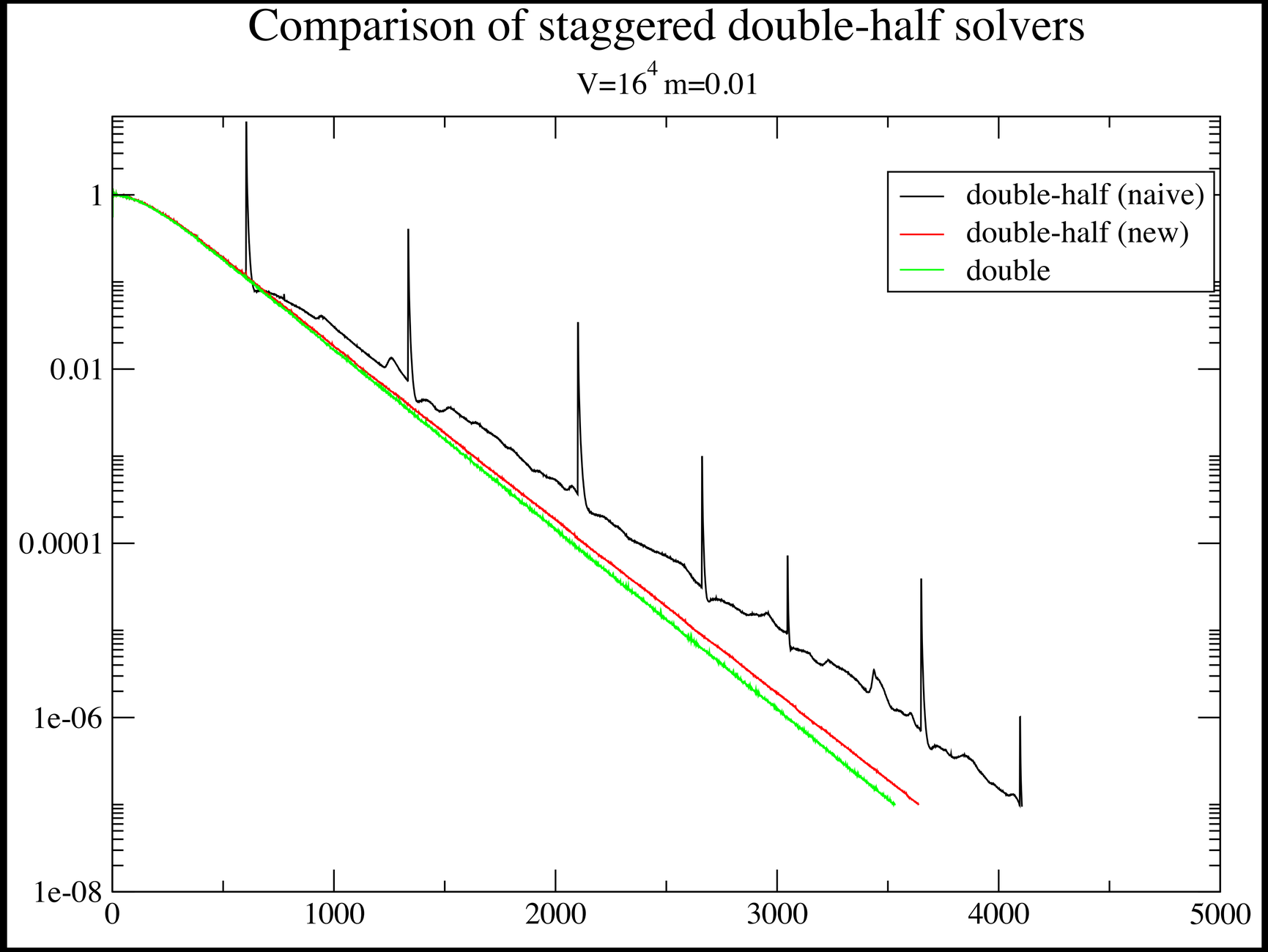
- CG convergence relies on gradient vector being orthogonal to residual
  - Re-project when injecting new residual
- α chosen to minimize $|e|_A$
  - True irrespective of precision of p, q, r
  - Solution correction is truncated if we keep low precision x
  - Always keep solution vector in high precision
- 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

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

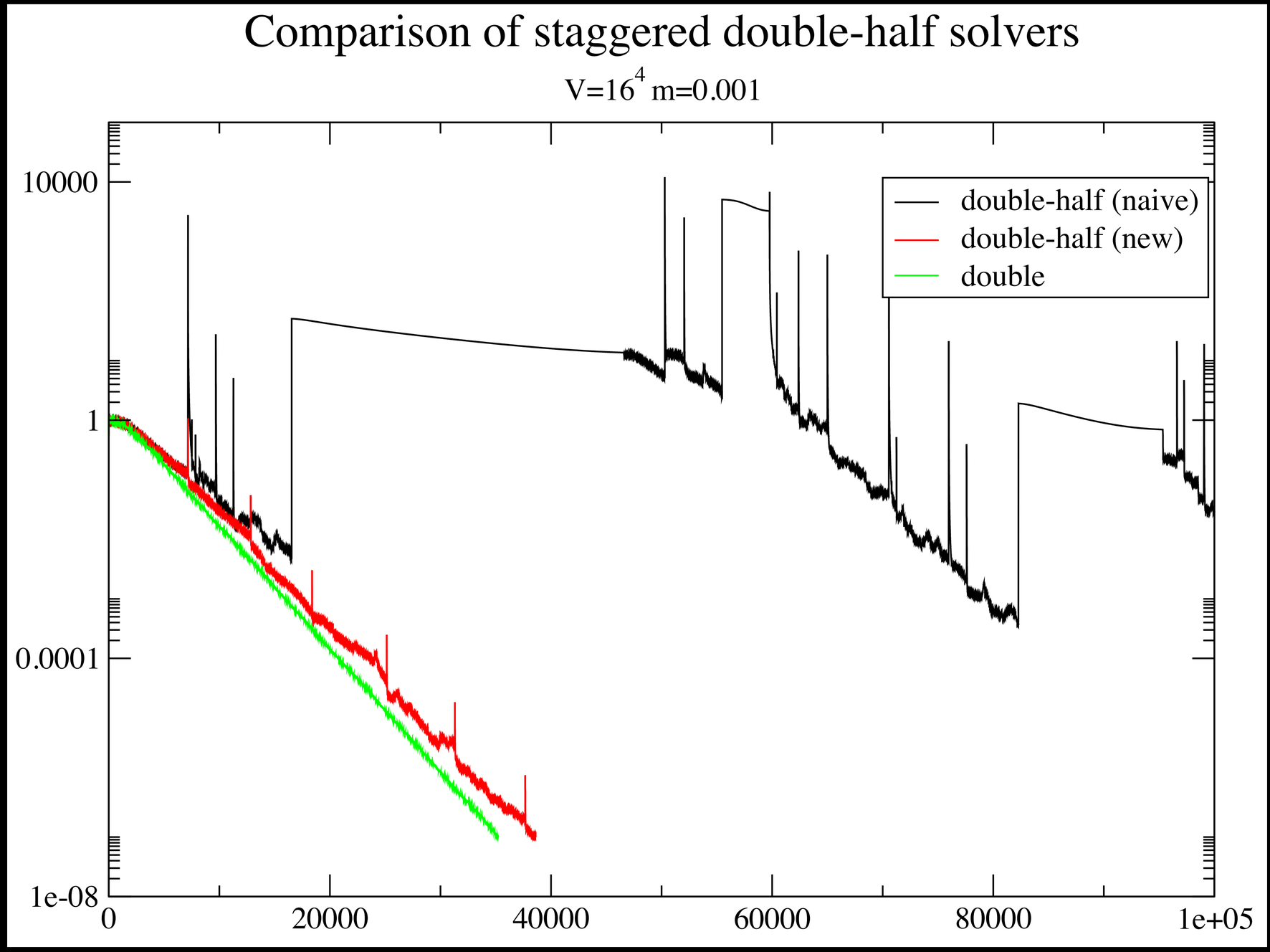
- Further improvement possible
  - Mining the literature on fault-tolerant solvers…

```
while (|r_k|> ε) {
    β_k = (r_k,r_k)/(r_k-1,r_k-1)
    p_k+1 = r_k - β_k p_k
    q_k+1 = A p_k+1
    α = (r_k,r_k)/(p_k+1, q_k+1)
    r_k+1 = r_k - α q_k+1
    x_k+1 = x_k + α p_k+1
    k = k+1
}
```

QCDNA 2014

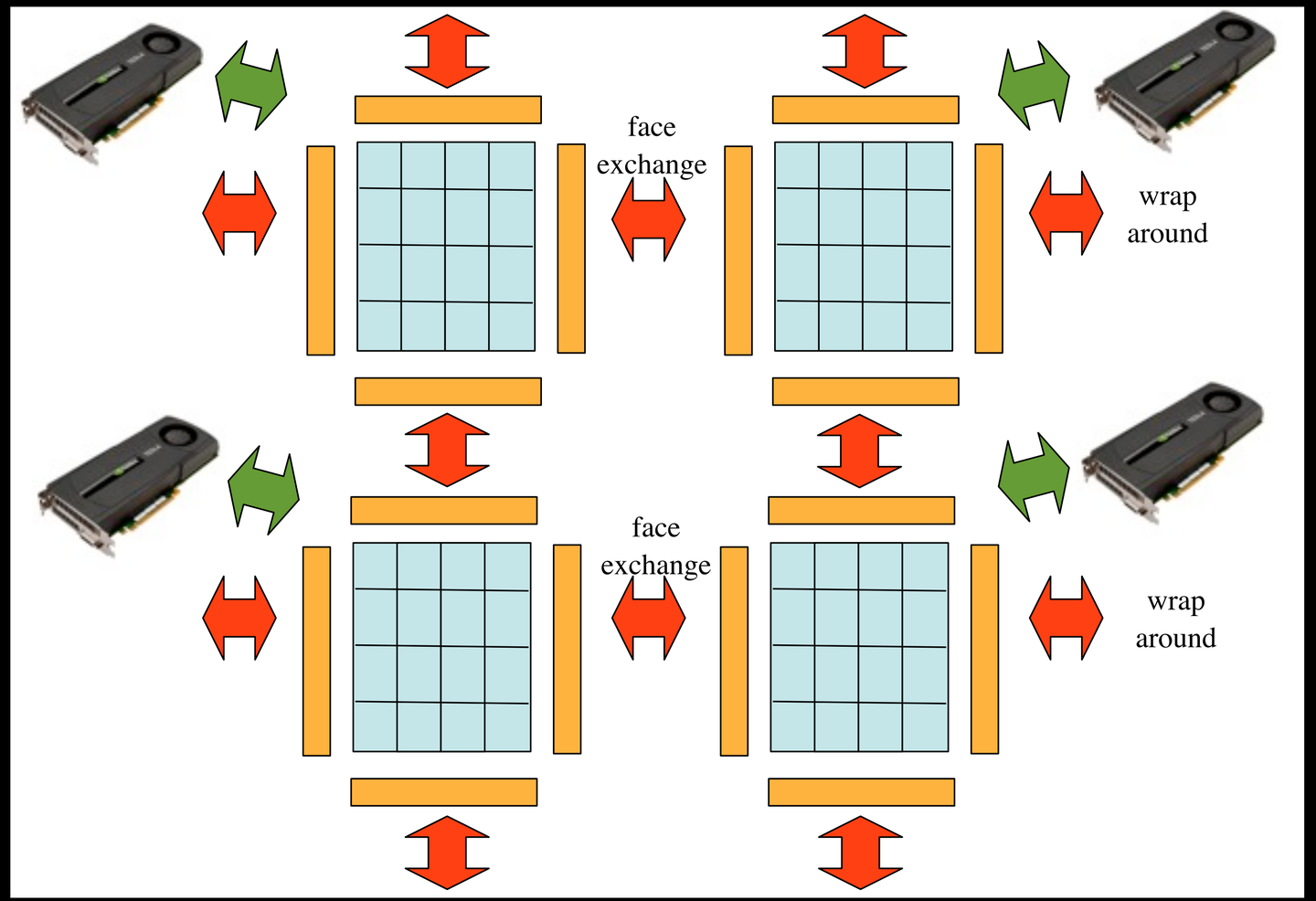

Comparison of staggered double-half solvers
V=16^4 m=0.01
double-half (naive)
double-half (new)
double
1
0.01
0.0001
1e-06
1e-08
0
1000
2000
3000
4000
5000

QCDNA 2014


Comparison of staggered double-half solvers
V=16^4 m=0.001
10000
1
0.0001
1e-08
0
20000
40000
60000
80000
1e+05
double-half (naive)
double-half (new)
double

QCDNA 2014

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

QCDNA 2014

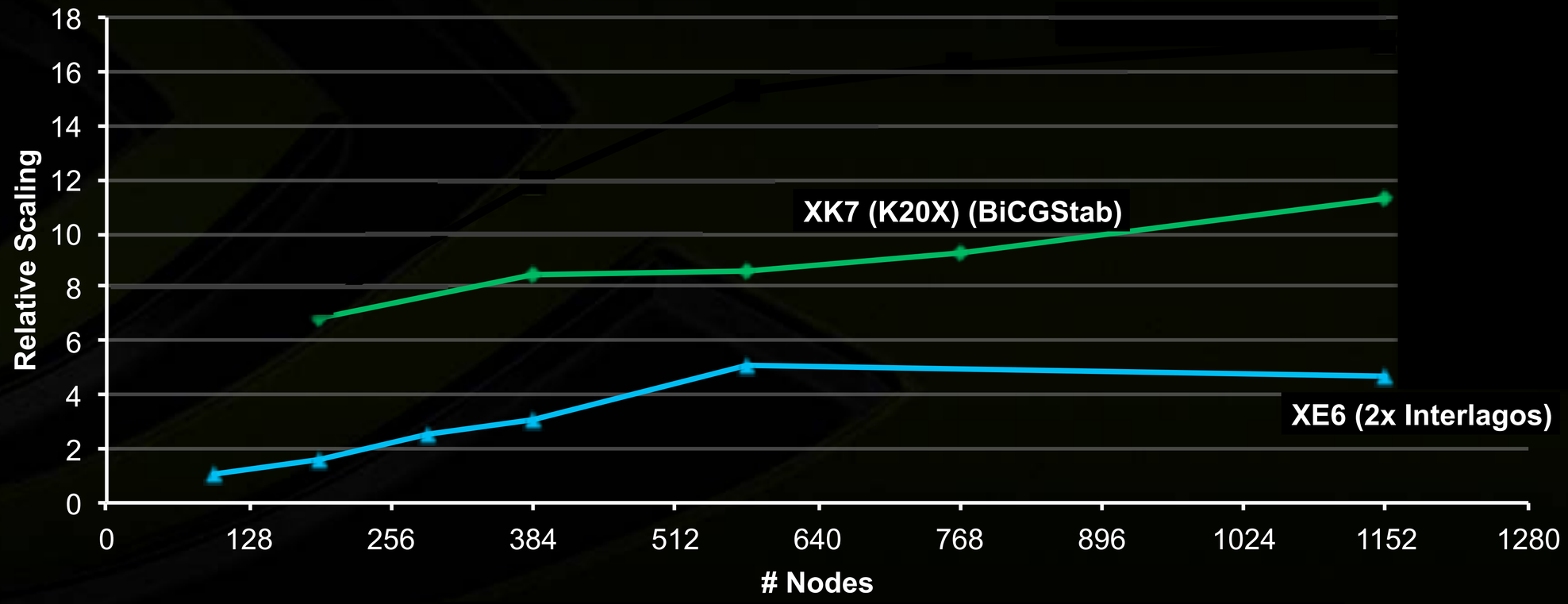
Strong Scaling Chroma with DD

Chroma

48^3x512 lattice

Relative Scaling (Application Time)

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



QCDNA 2014

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

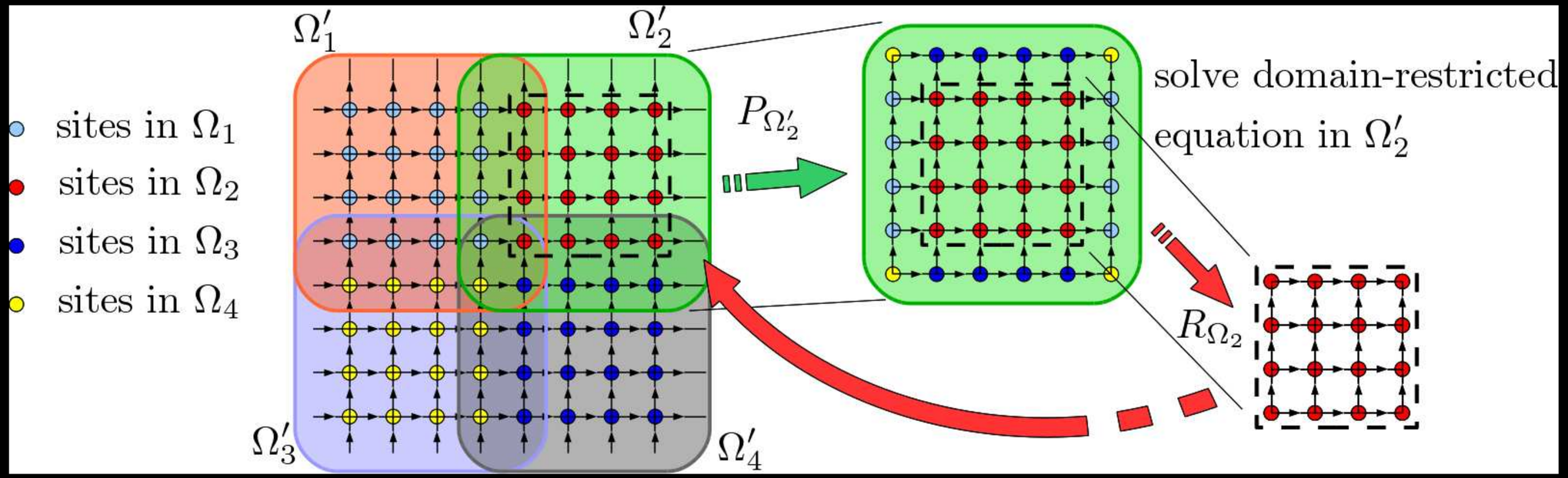
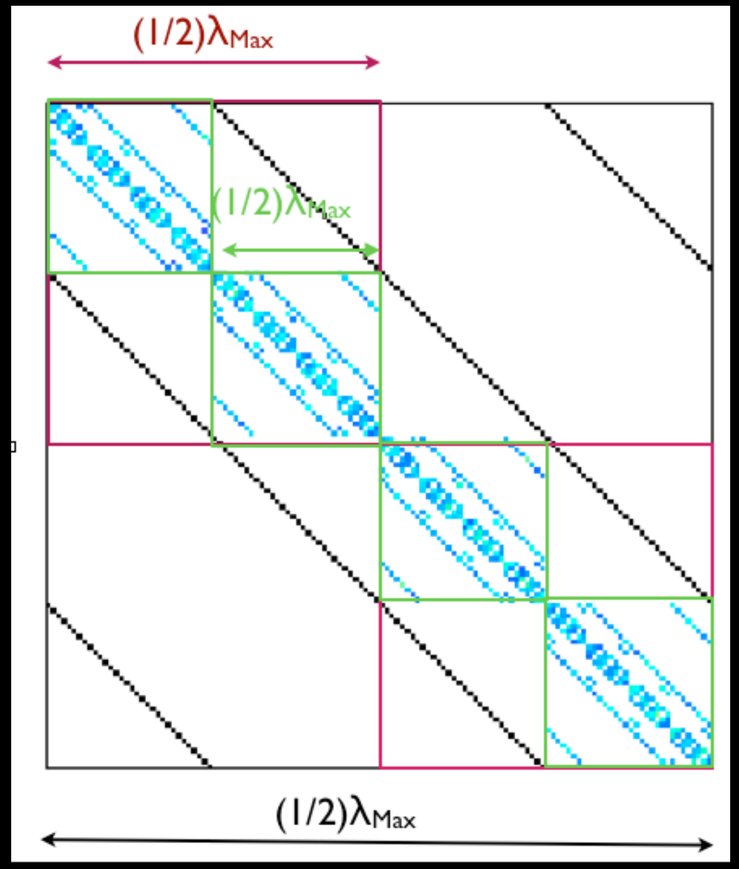


figure taken from Osaki and Ishikawa

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



QCDNA 2014

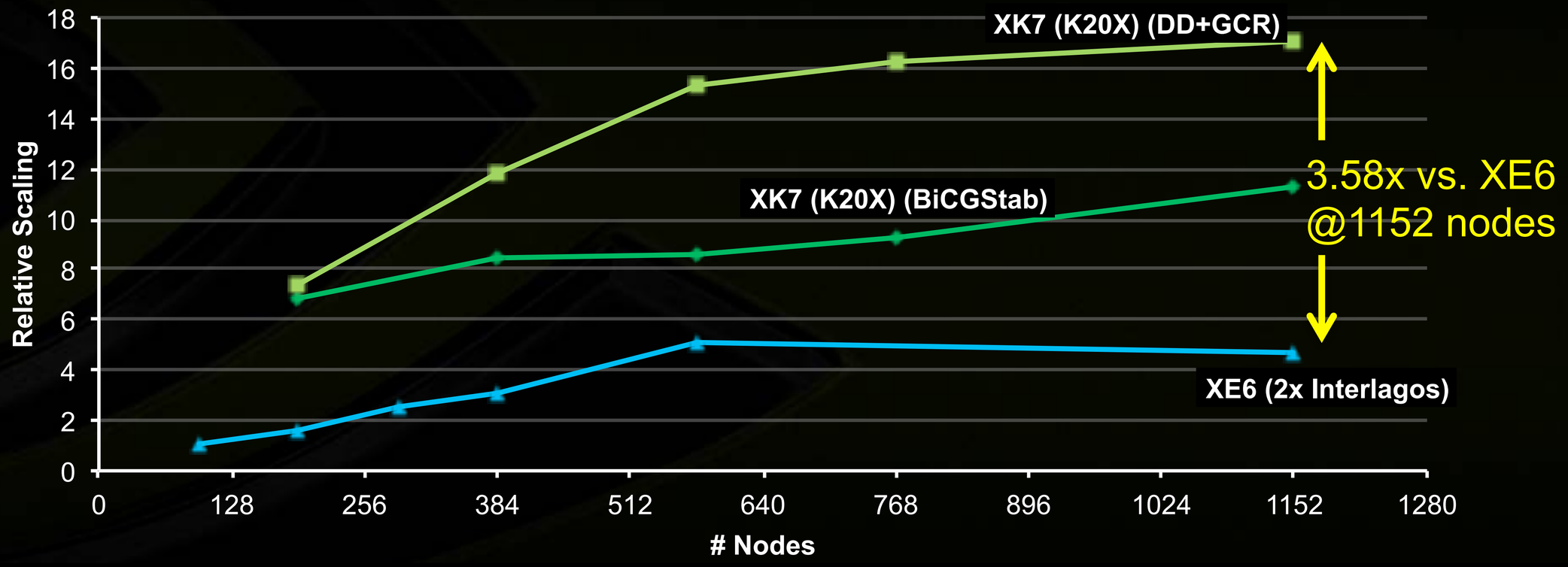
Strong Scaling Chroma with DD

Chroma

48^3x512 lattice

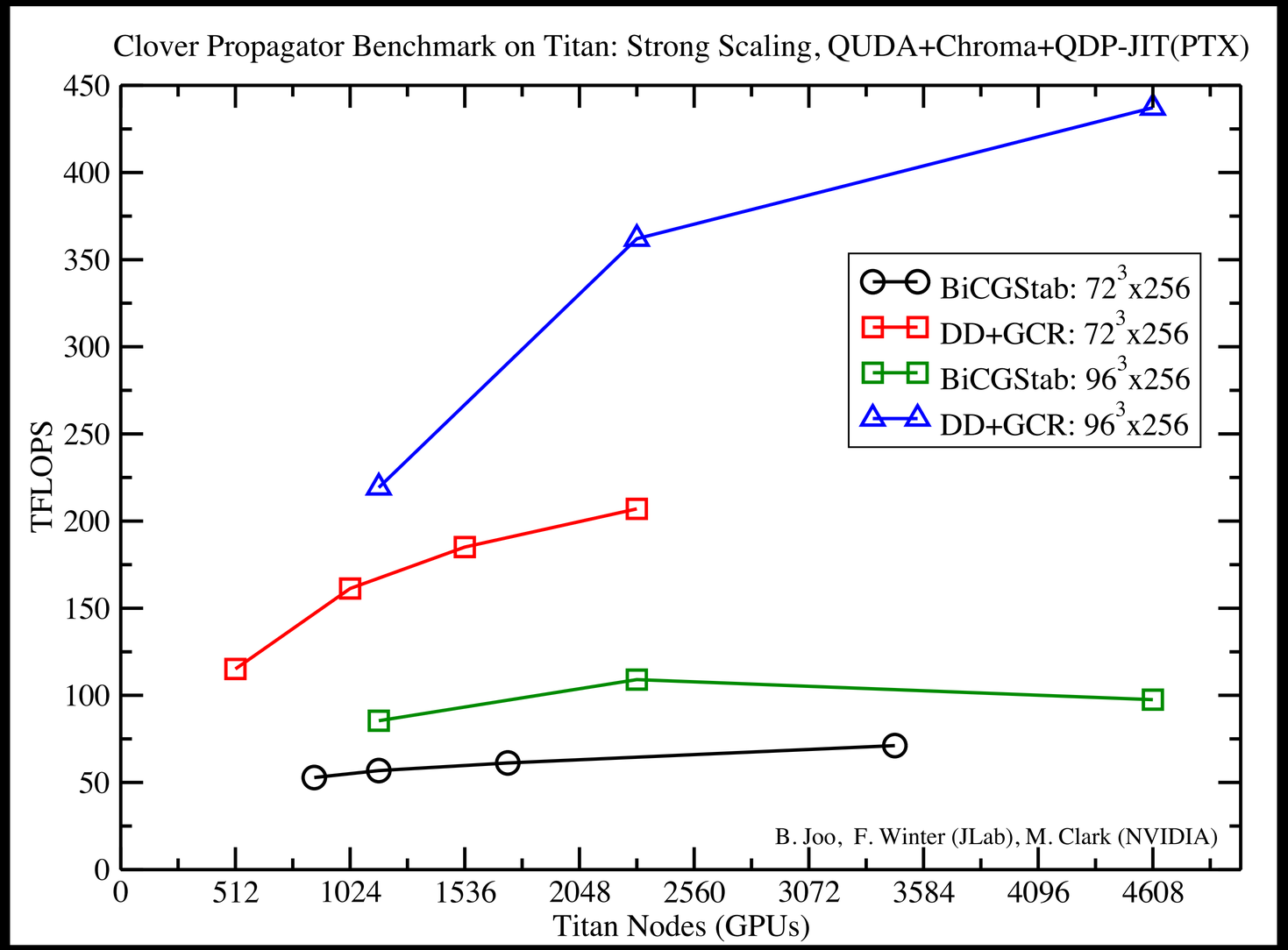
Relative Scaling (Application Time)

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



QCDNA 2014

Extreme Scaling



QCDNA 2014

Deflation Algorithms in QUDA

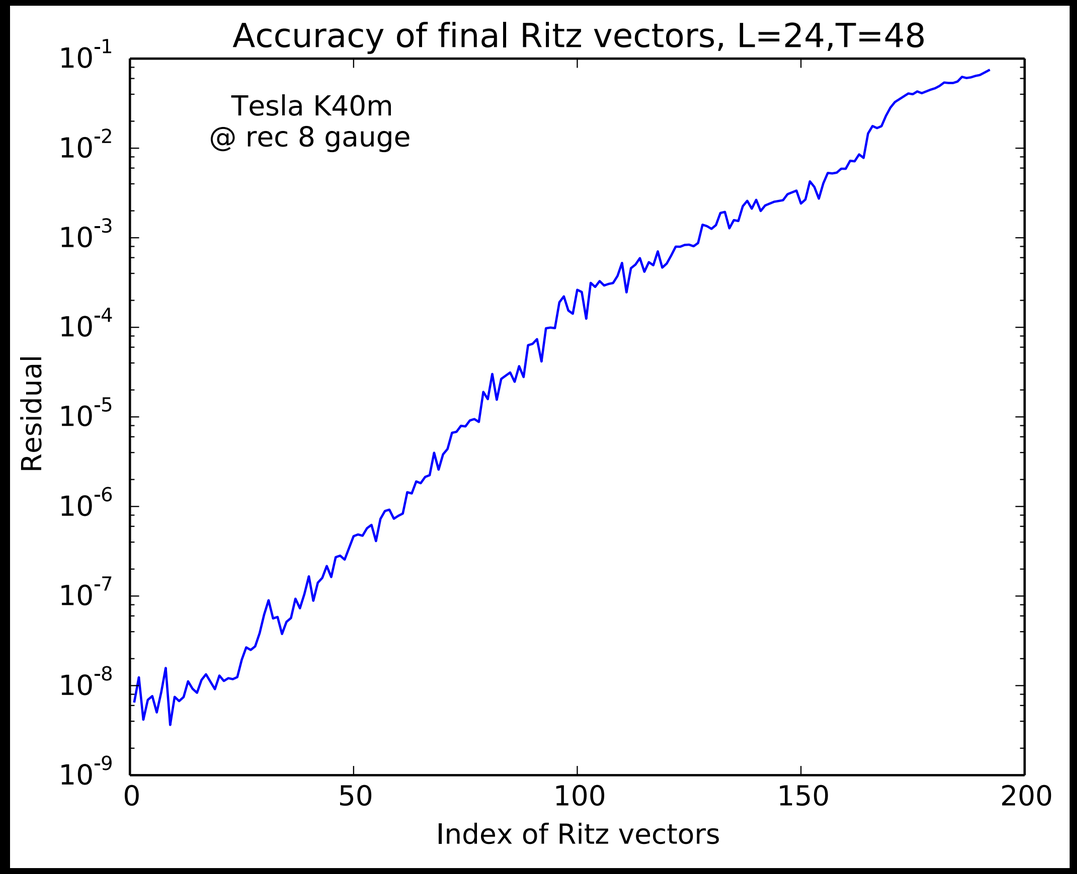
- EigCG implemented in QUDA (Alexei Strelchenko)

$U = [], \quad H = []$ //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

QCDNA 2014

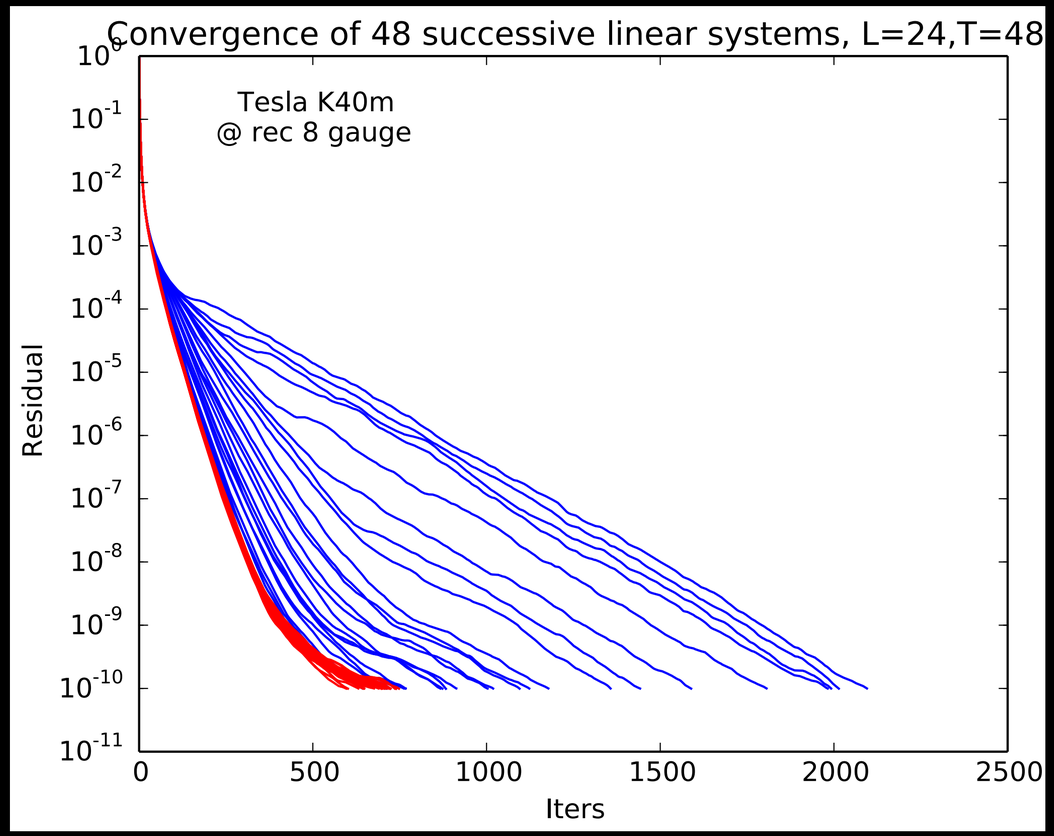
Deflation Algorithms in QUDA

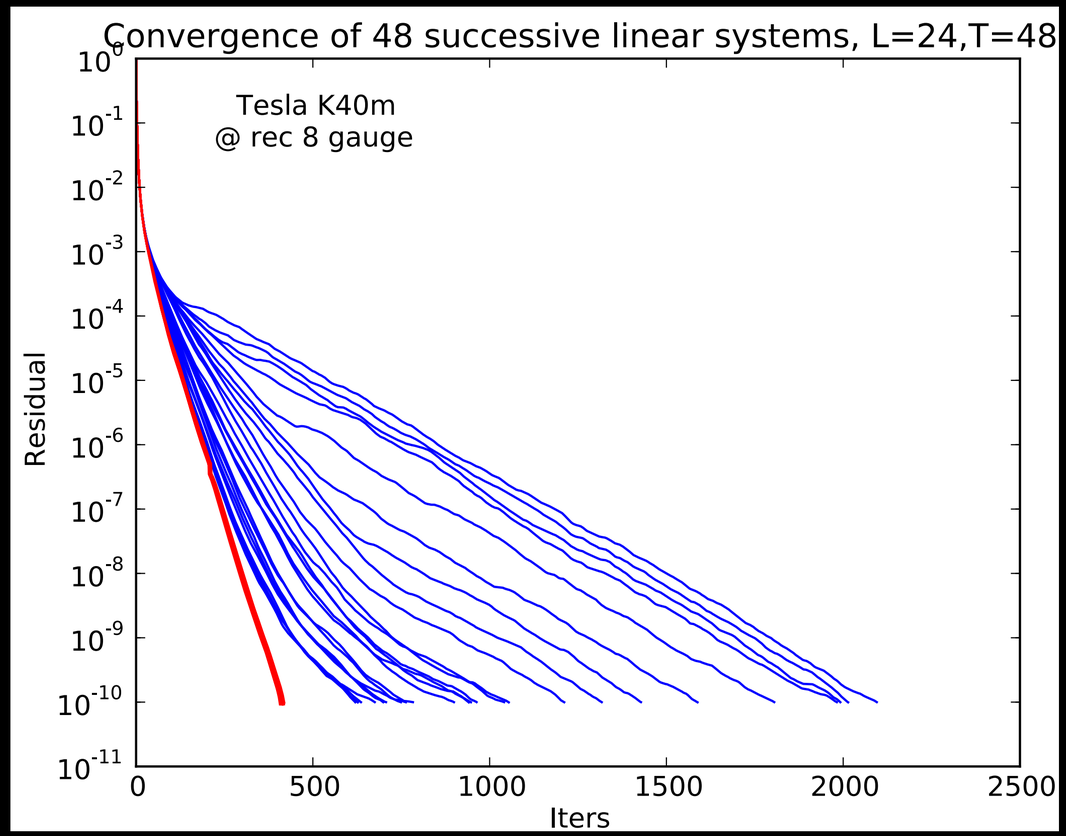
- Use MAGMA library for required LAPACK functionality
- Memory not a problem
  - EigCG only works on subsets
  - Cache full set on CPU
- Extensible eigenvector solver framework for future solvers
  - EigBiCG
  - GMRES-DR
  - etc.



QCDNA 2014

Deflation Algorithms in QUDA





degenerate twisted mass 24^3x48, $\kappa = 0.161231$, $\mu = 0.0085$

QCDNA 2014

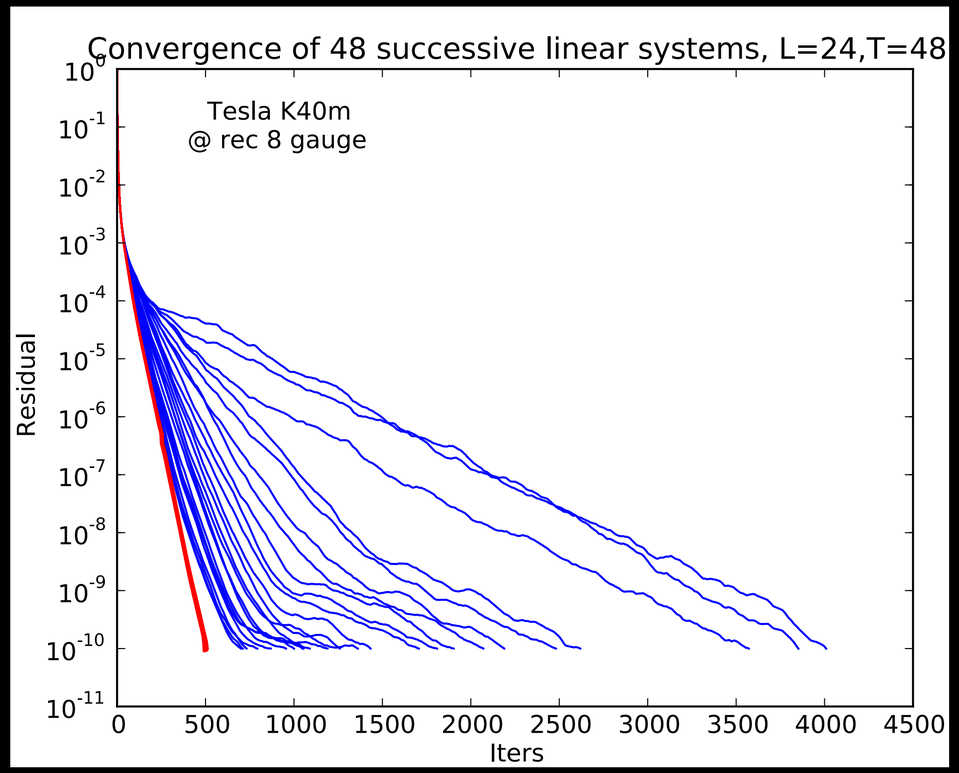
Mixed-Precision Deflation Algorithms

- 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, $\kappa = 0.161231$, $\mu = 0.0040$

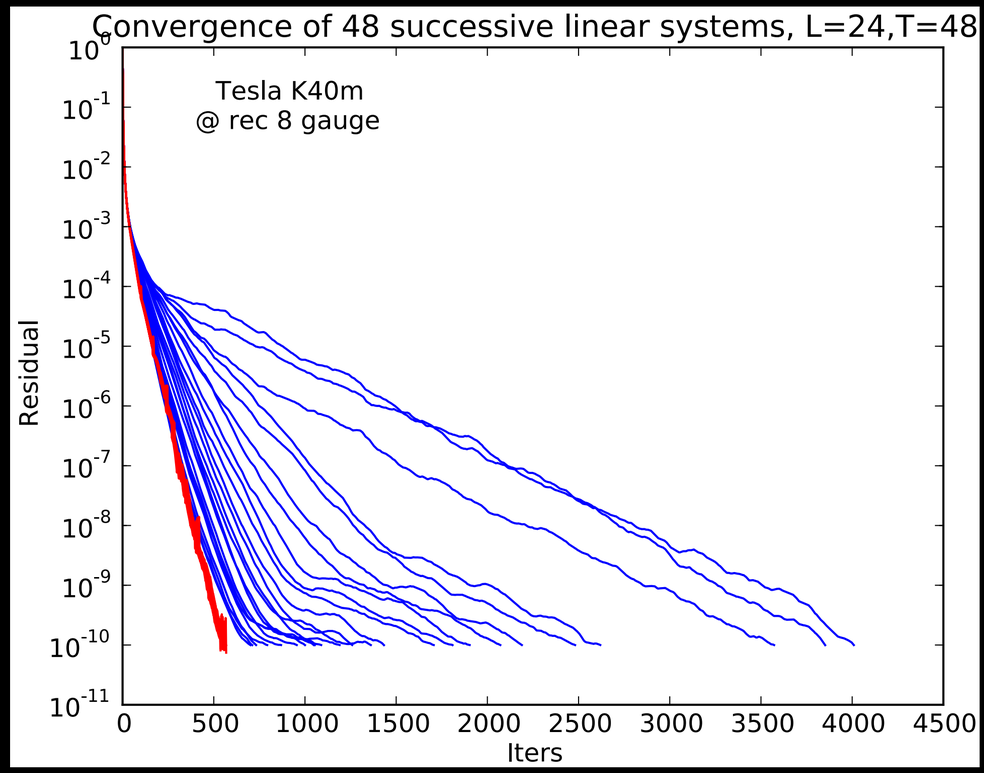
Non-deflated double-single CG: 15 sec
Non-deflated double-half CG: (does not converge)
InitCG double-single initCG: 2.42 sec
InitCG double-half initCG: 1.84 sec
Achieved speedup ~8X for initCG (combination of algorithm and precision)

QCDNA 2014

Mixed Precision Deflation Algorithms



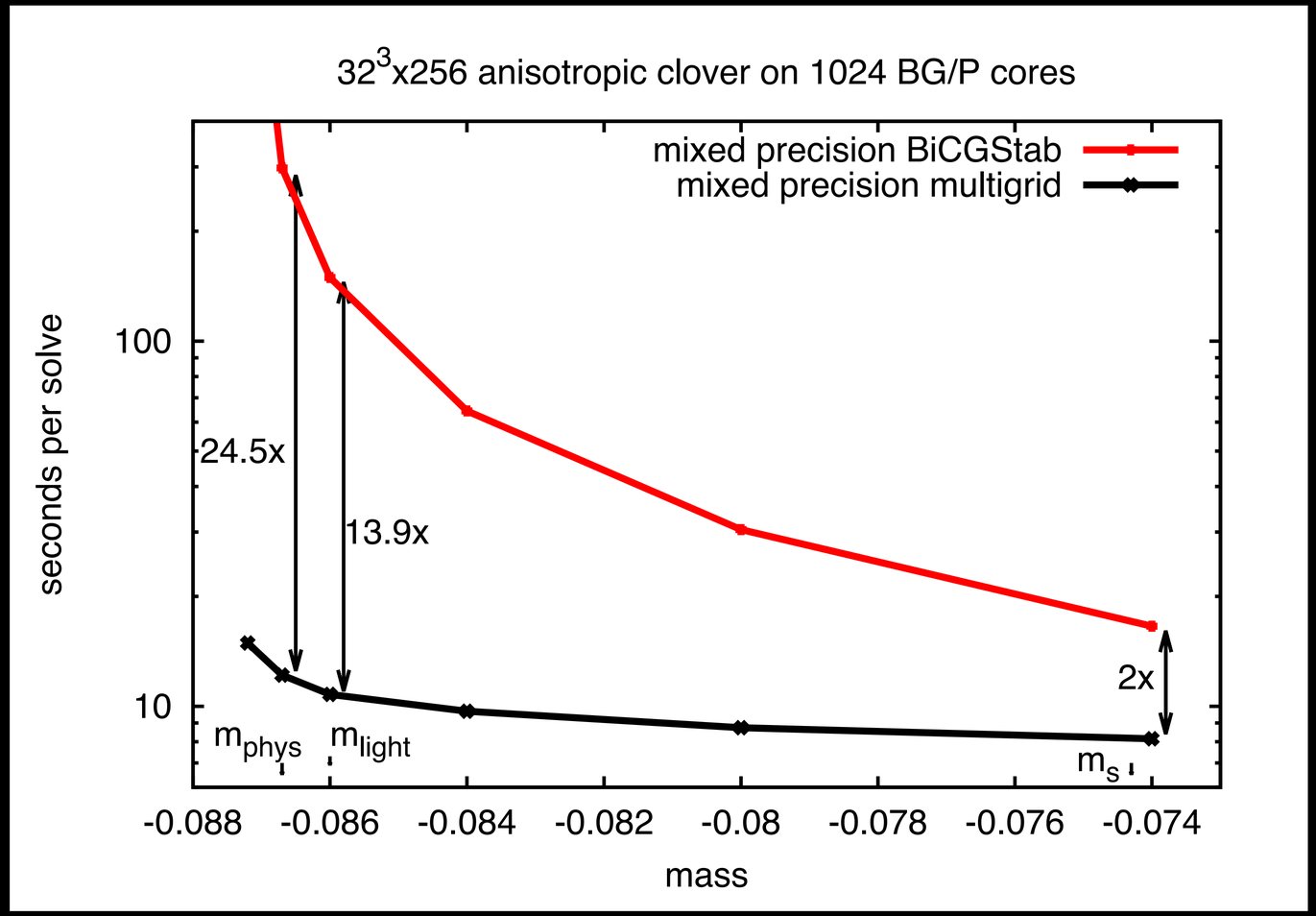
double-single



double-half

degenerate twisted mass 24^3x48, $\kappa = 0.161231$, $\mu = 0.0040$

Adaptive Geometric Multigrid

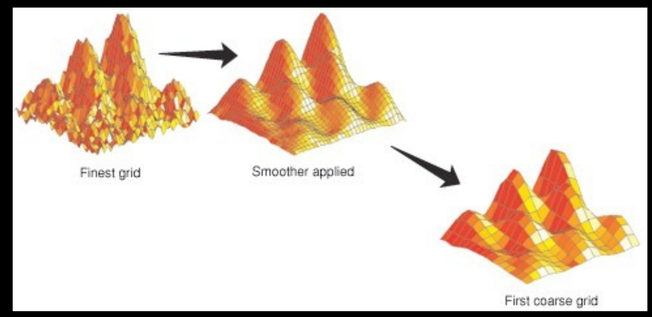


Osborn *et al 2011*

QCDNA 2014

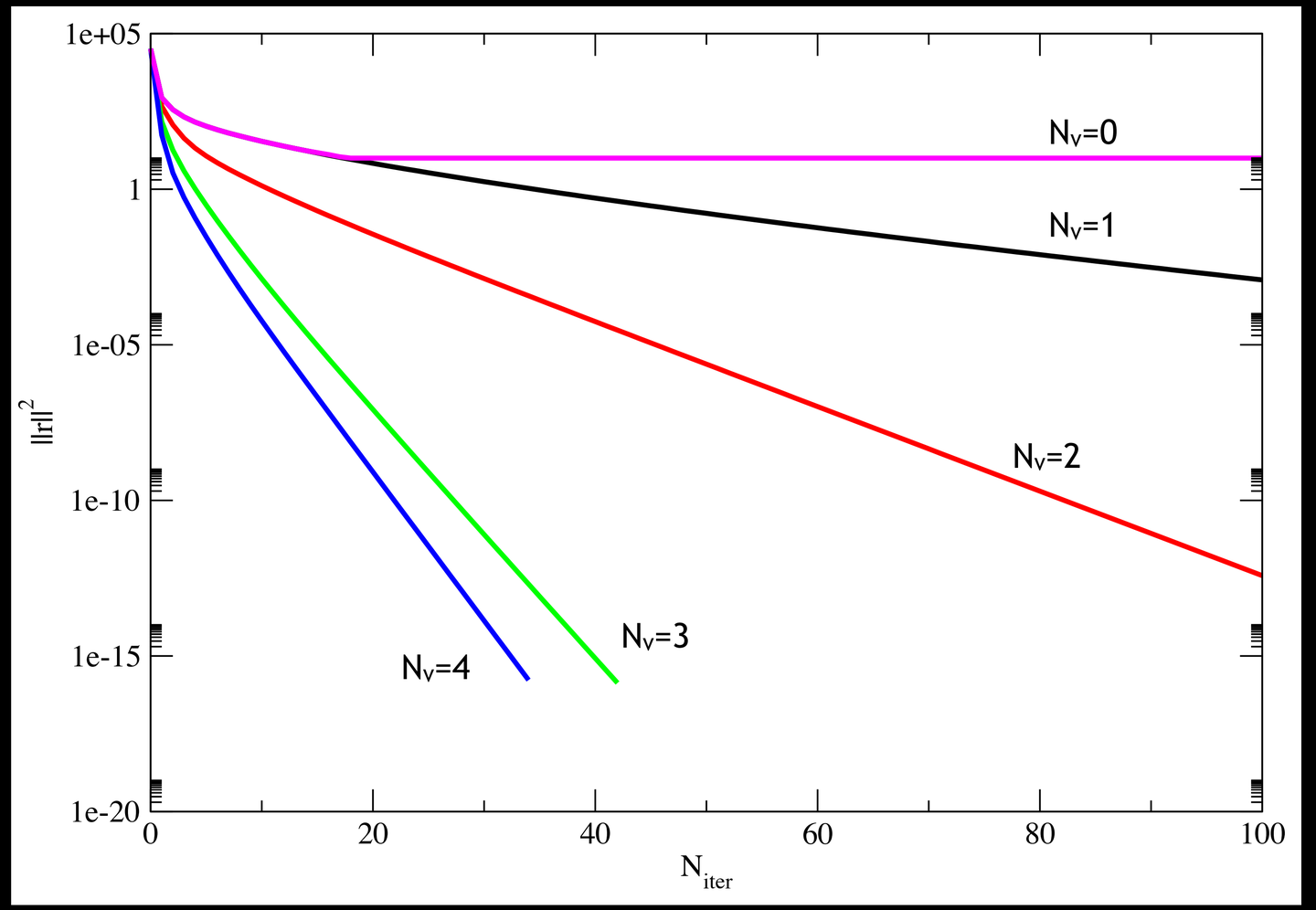
Adaptive Geometric Multigrid

- Adaptively find candidate null-space vectors
  - Dynamically learn the null space and use this to define the prolongator
  - Algorithm is self learning
- Setup
  - Set solver to be simple smoother
  - Apply current solver to random vector $v_i = P(D)\ \eta_i$
  - If convergence good enough, solver setup complete
  - Construct prolongator using fixed coarsening $(1 - P\ R)\ v_k = 0$
    - ➡ Typically use 4^4 geometric blocks
    - ➡ Preserve chirality when coarsening $R = \gamma_5\ P^\dagger\ \gamma_5 = P^\dagger$
  - Construct coarse operator ($D_c = R\ D\ P$)
  - Recurse on coarse problem
  - Set solver to be augmented V-cycle, goto 2

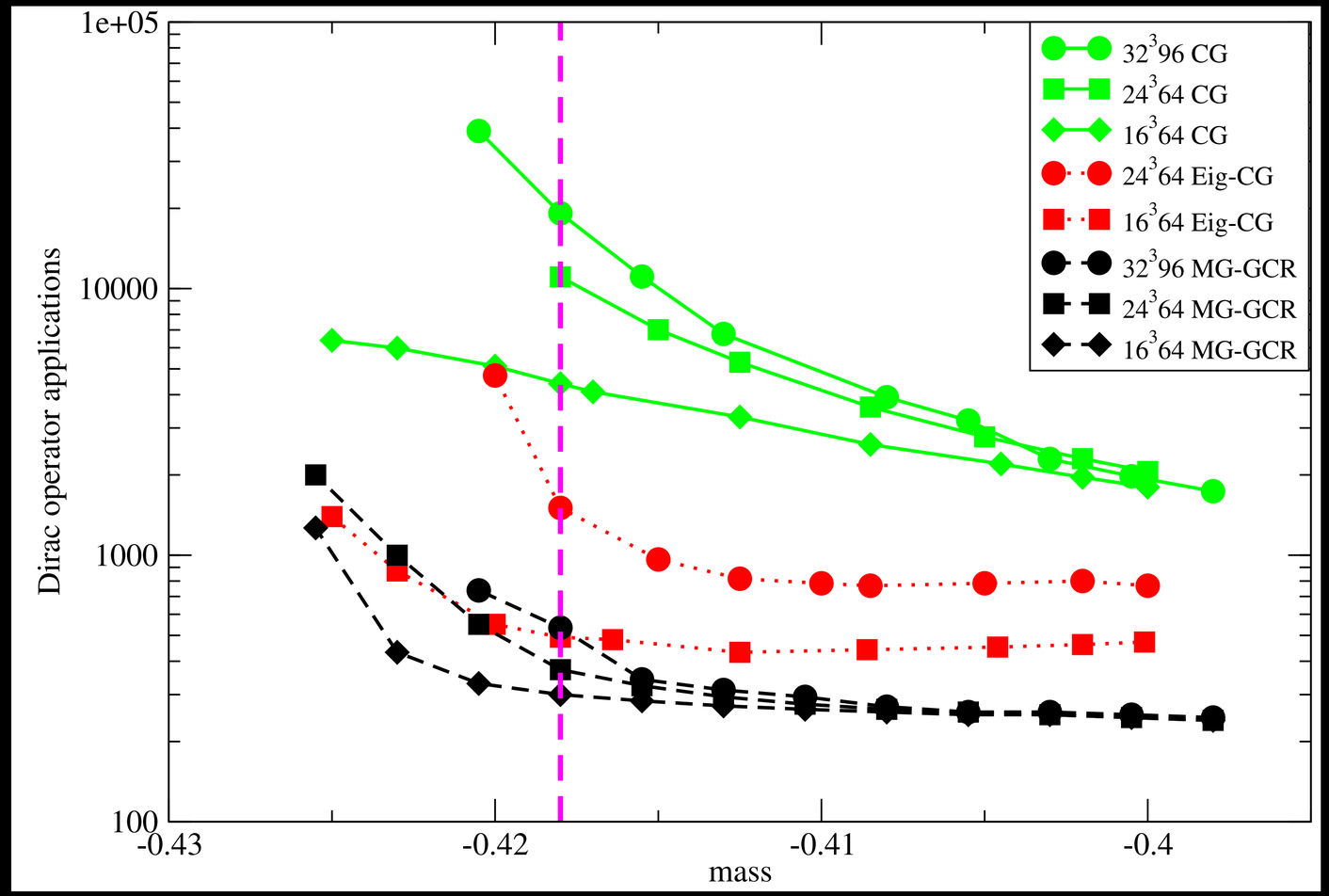


QCDNA 2014

Adaptive Geometric Multigrid



Adaptive Geometric Multigrid



240 vectors

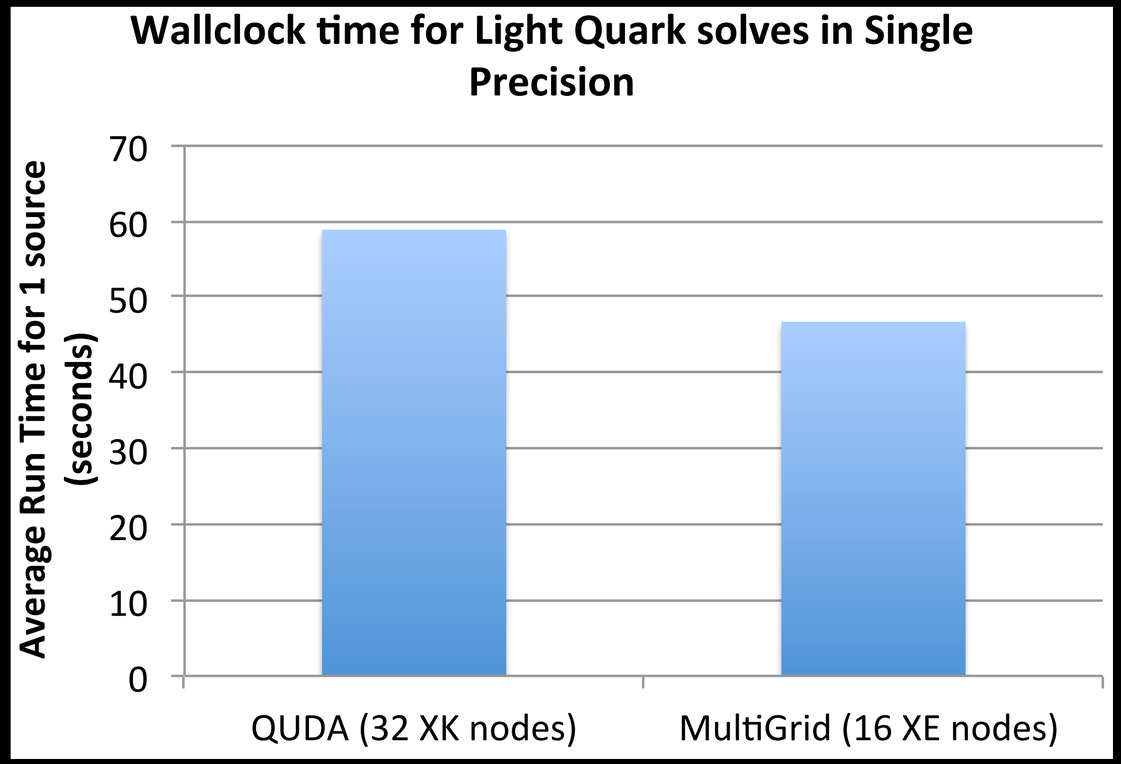
20 vectors

Babich *et al* 2010

QCDNA 2014

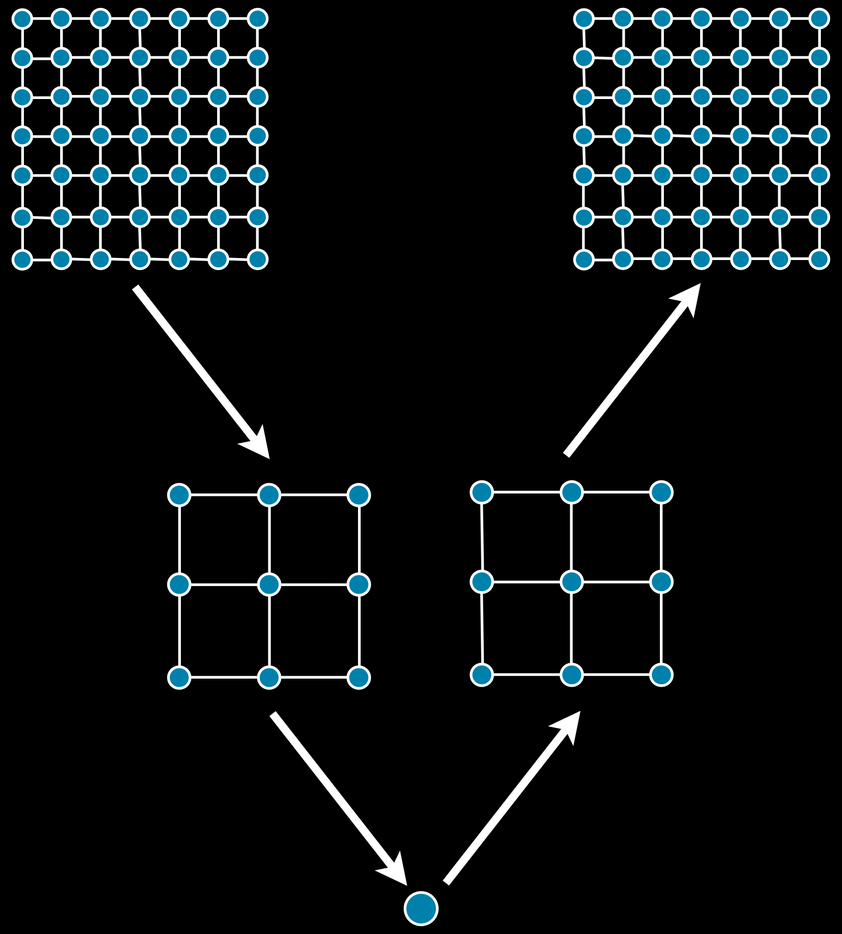
Motivation

- 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

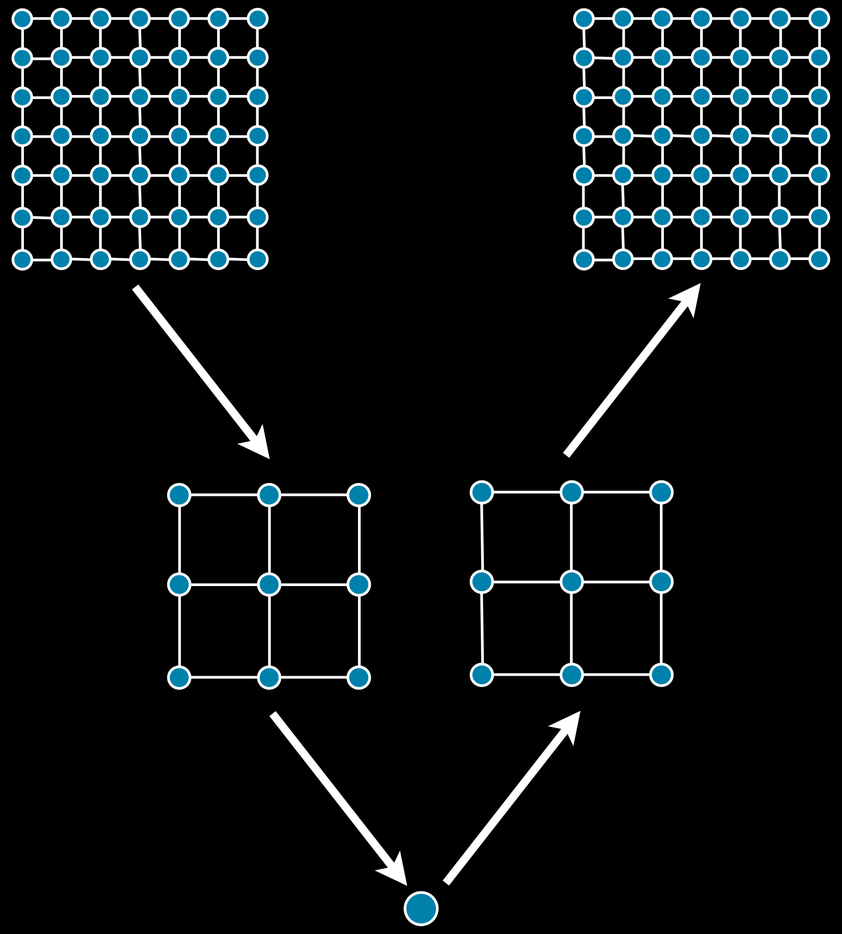
The Challenge of Multigrid on GPU

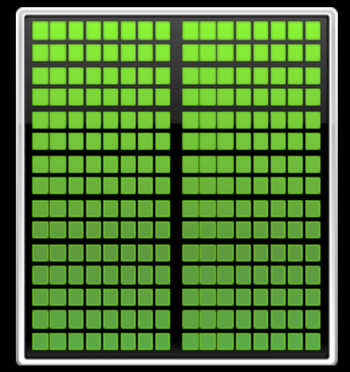


- GPU requirements very different from CPU
  - Each thread is slow, but O(10,000) threads per GPU
- Fine grids run very efficiently
  - High parallel throughput problem
- Coarse grids are worst possible scenario
  - More cores than degrees of freedom
  - Increasingly serial and latency bound
  - Little's law (bytes = bandwidth * latency)
  - Amdahl's law limiter
- Multigrid decomposes problem into throughput and latency parts

QCDNA 2014

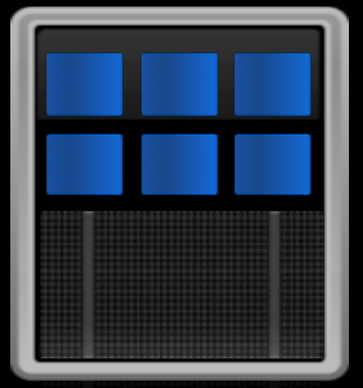
Hierarchical algorithms on heterogeneous architectures





GPU

Thousands of cores for parallel processing



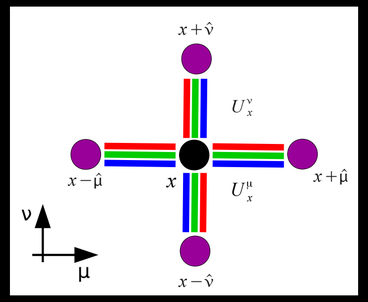
CPU

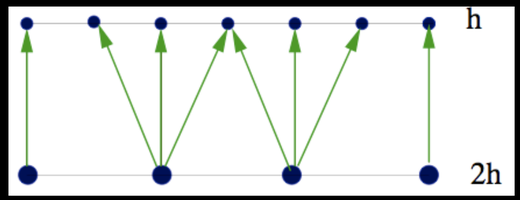
Few Cores optimized for serial work

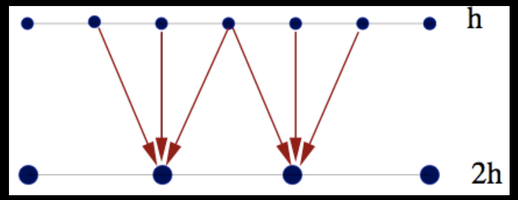
QCDNA 2014

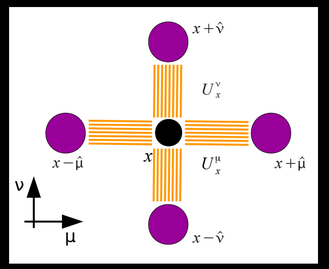
Ingredients for Parallel Adaptive Multigrid

- 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







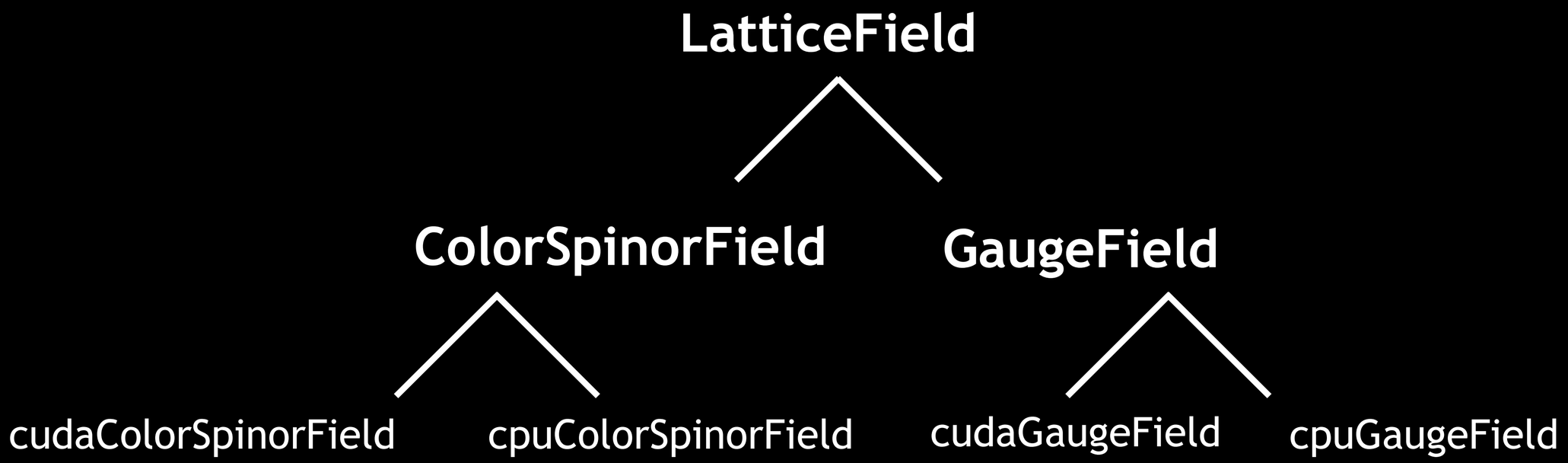


Design Goals

- 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

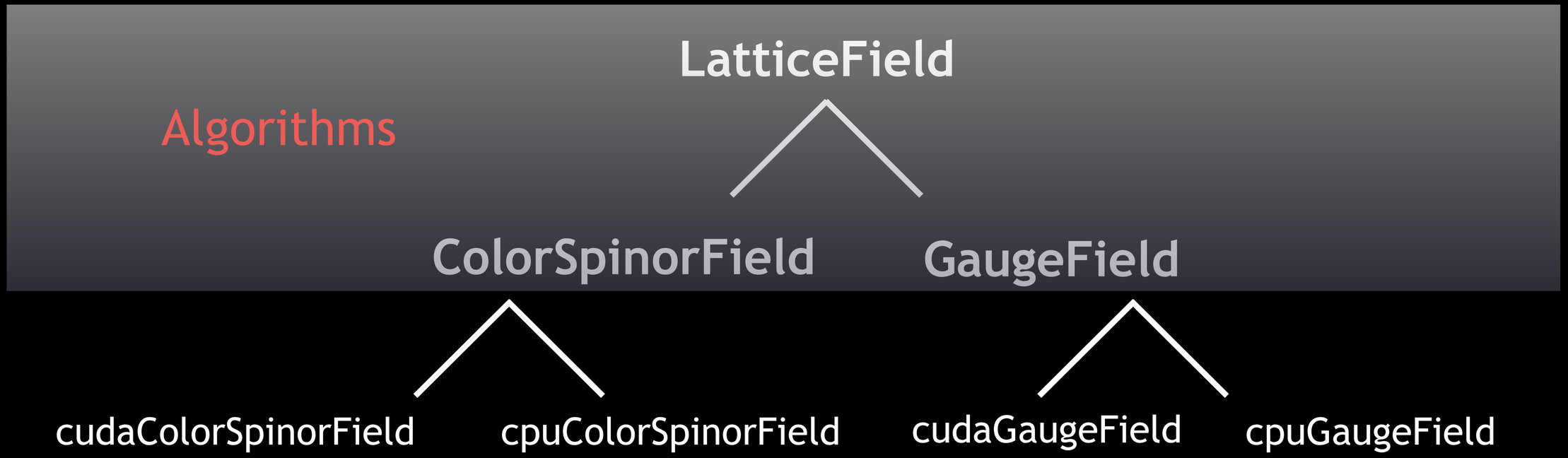
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



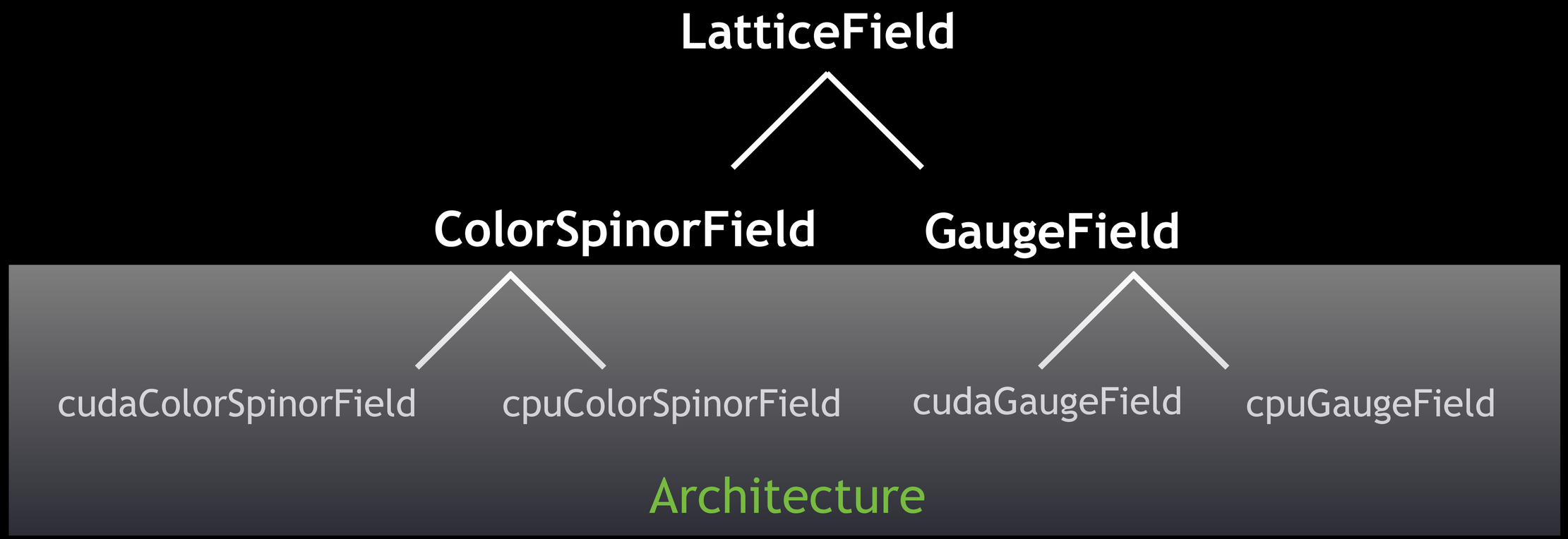
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



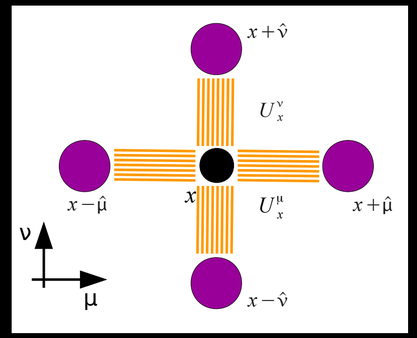
Multigrid and QUDA

- 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
  }

}
```

QCDNA 2014

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

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

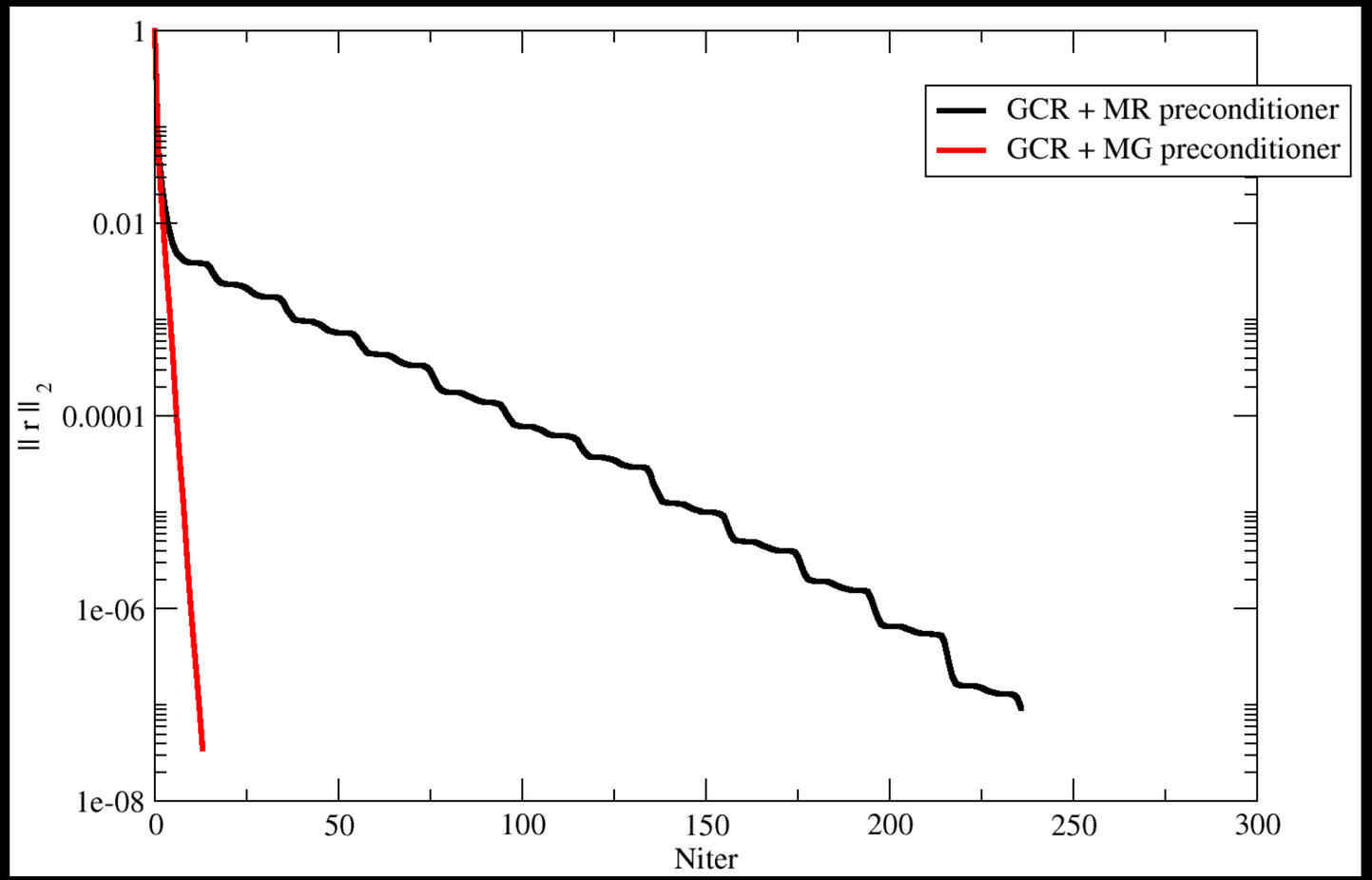
```
template <typename Arg, int Ncolor, int Nspin>
 __global__ void prolongate(Arg arg) {
   int x = blockIdx.x*blockDim.x + threadIdx.x;
   for (int s=0; s<Nspin; s++) {
     for (int c=0; c<Ncolor; c++) {
     …
     }
   }
 }
```

- JIT compilation will fix this

QCDNA 2014

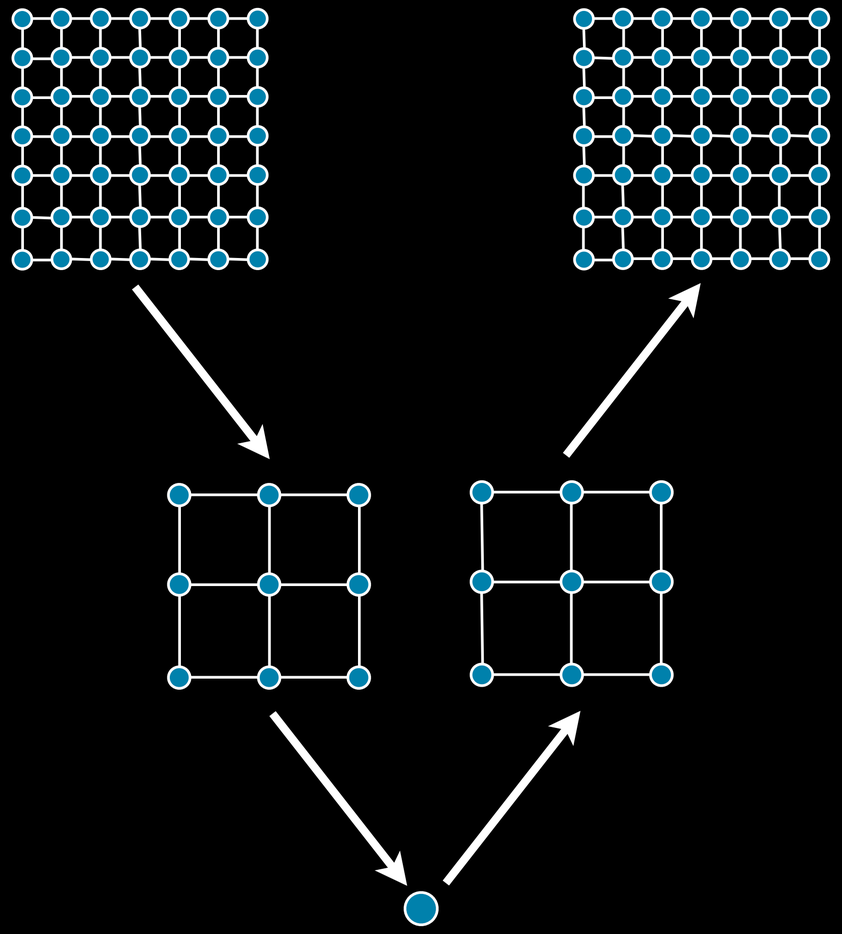
Current Status

- Framework is working but still slow

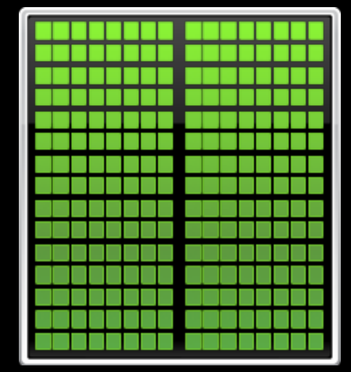


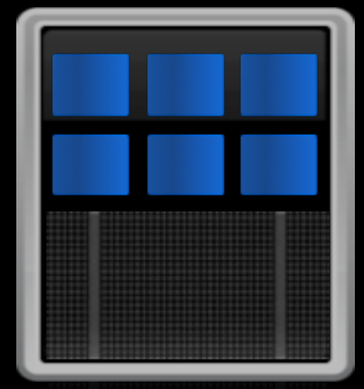
Fine grid on GPU
Coarse grid on CPU

Heterogeneous Updating Scheme

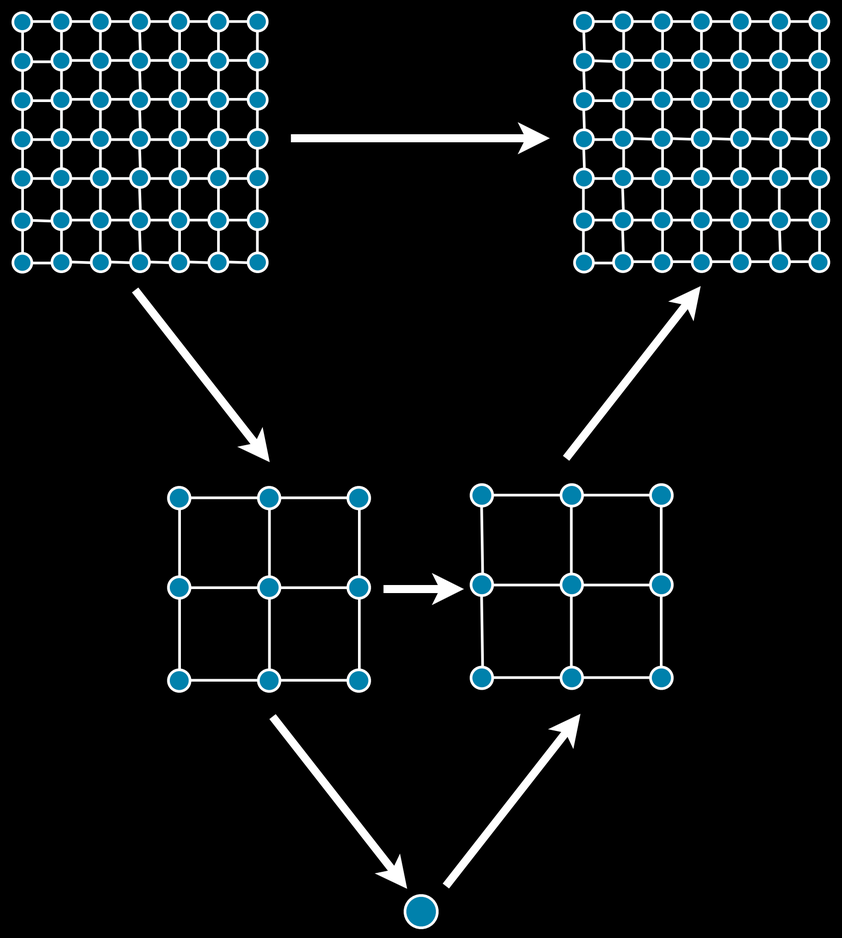


- Multiplicative MG is necessarily serial process
  - Cannot utilize both GPU and CPU simultanesouly

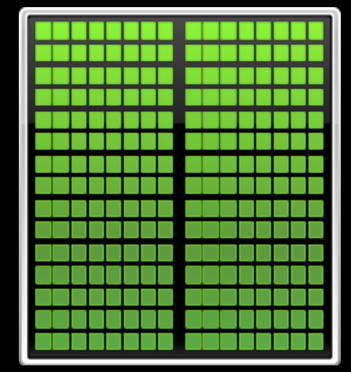


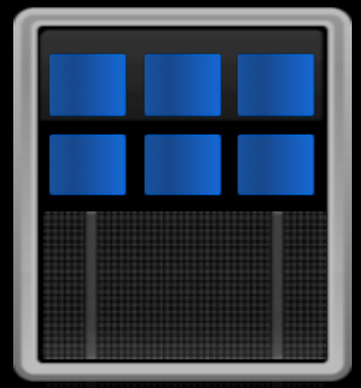


Heterogeneous Updating Scheme



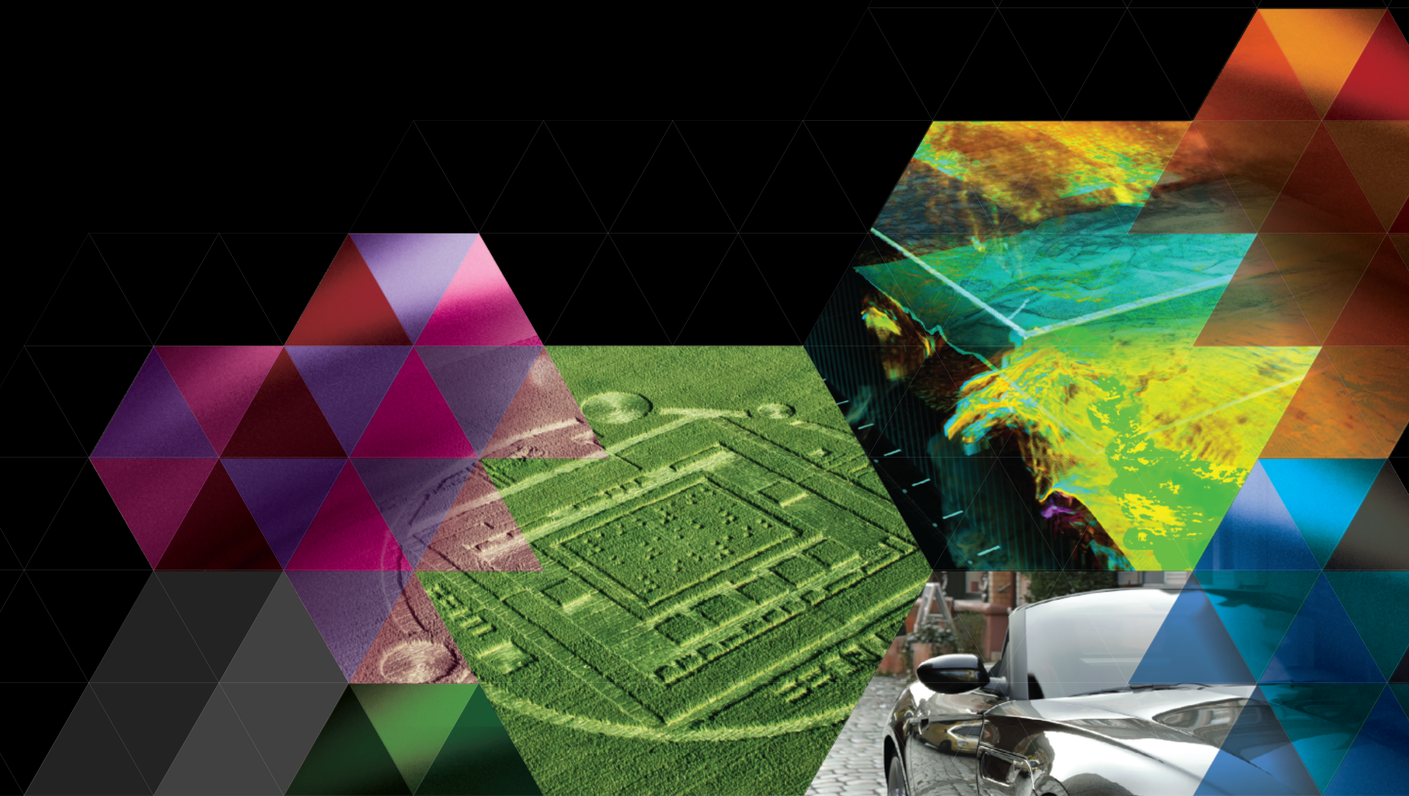
- Multiplicative MG is necessarily serial process
  - Cannot utilize both GPU and CPU simultanesouly
- Additive MG is parallel
  - Can utilize both GPU and CPU simultanesouly
- Additive MG requires accurate coarse-grid solution
  - Not amenable to multi-level
  - Only need additive correction at CPU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



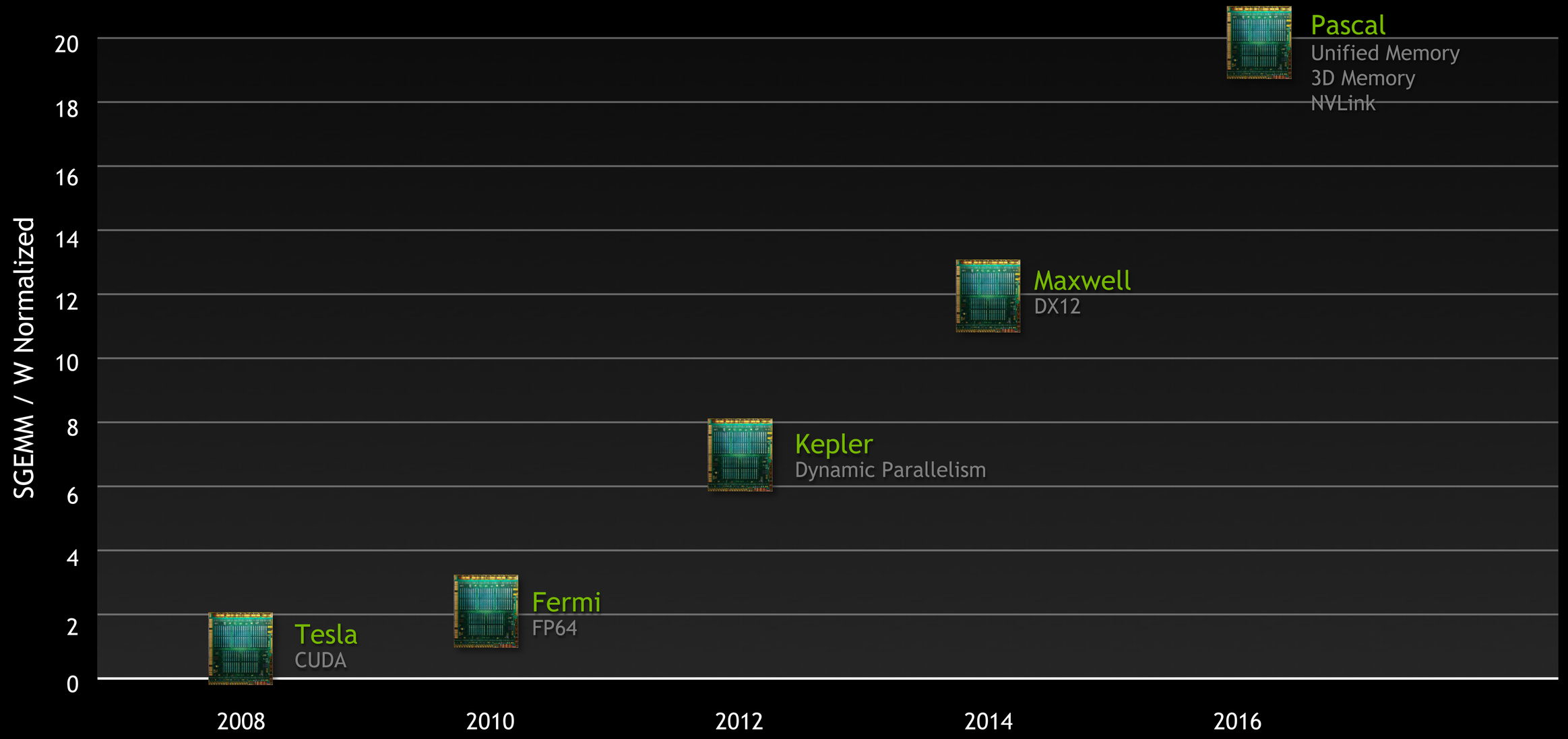


QCDNA 2014

FUTURE DIRECTIONS…



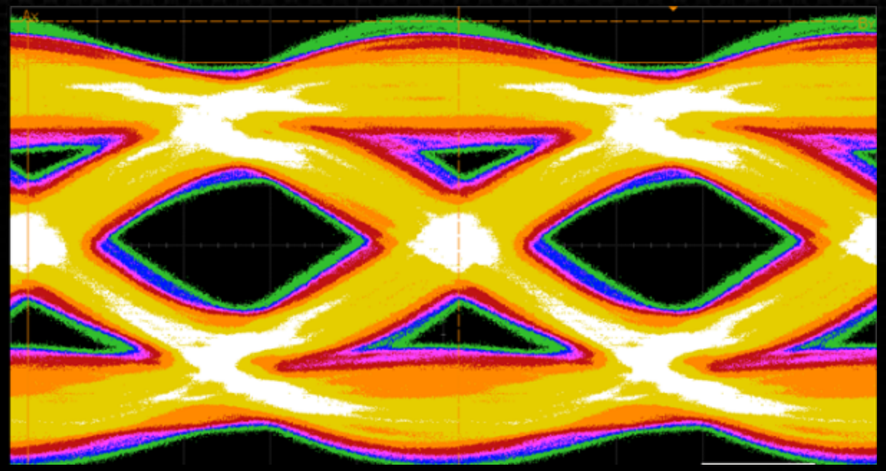
Strong GPU Roadmap



Introducing NVLINK and Stacked Memory

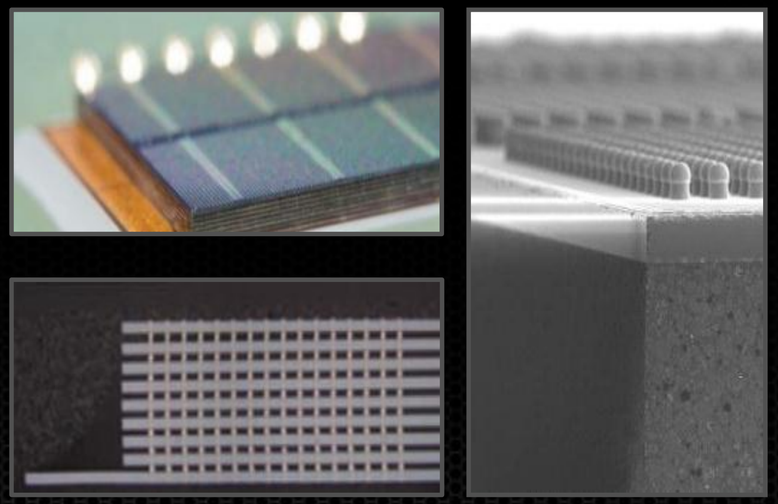
NVLINK

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

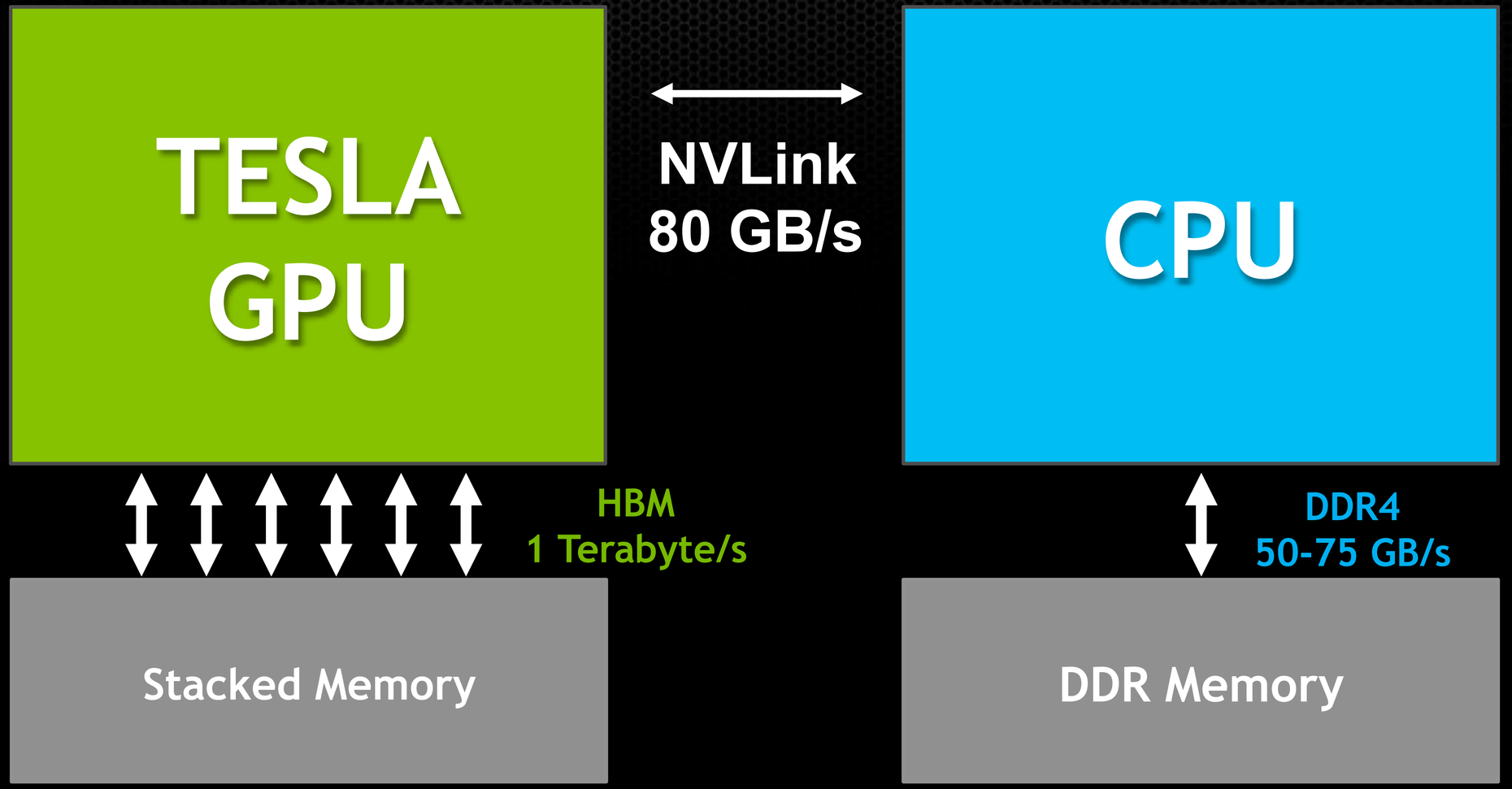


Stacked Memory

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



NVLink Enables Data Transfer At Speed of CPU Memory



The Future of GPUs

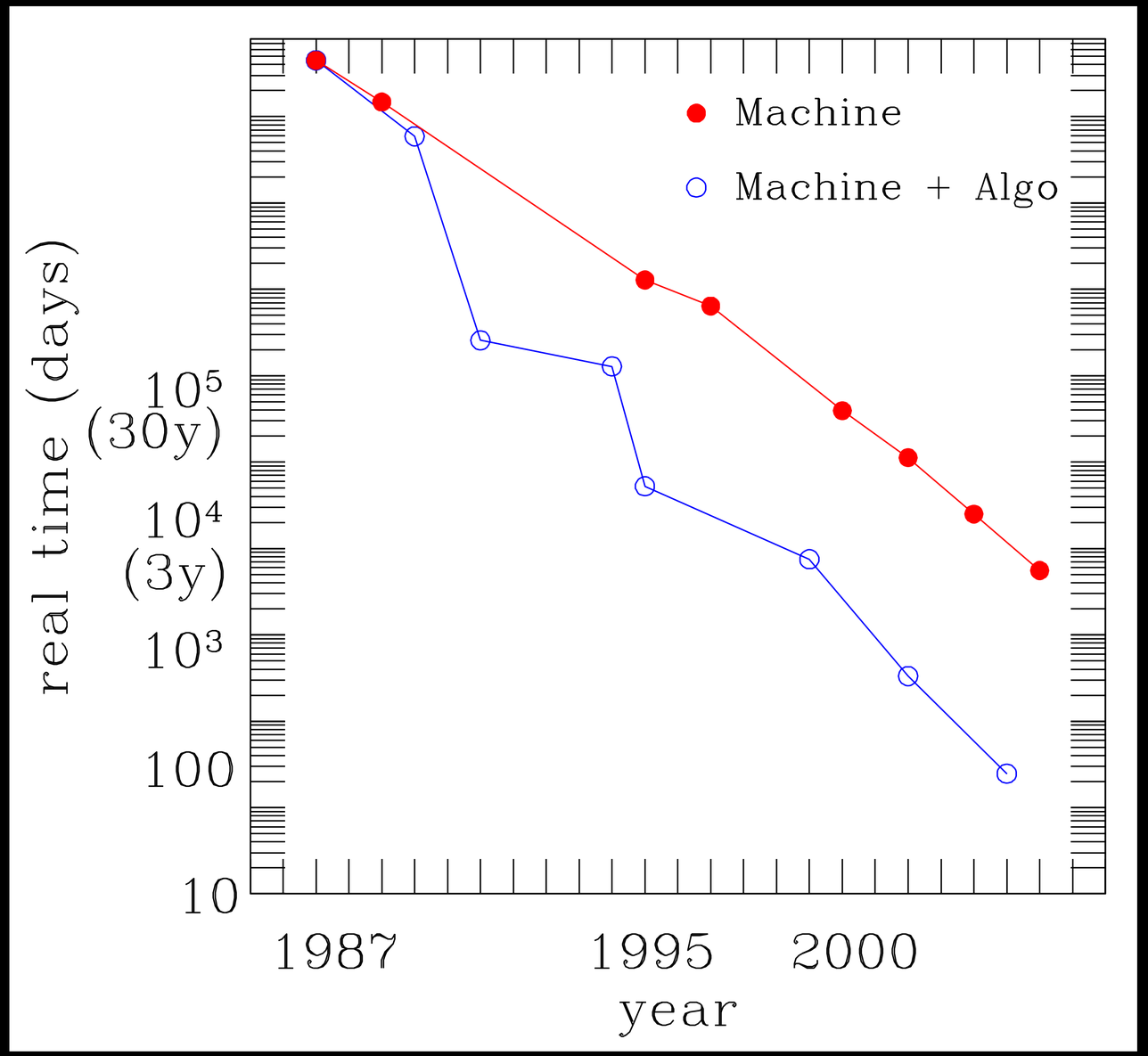
- GPUs viable because of multi $B gaming market
- Coming to an end anytime soon?


CLOSE TO NEW RECORD
500 points to beat KMagnus
Reach The Factory Roof
R RELOAD
1 / 36
SINGLE
1
+100

The Future of GPUs

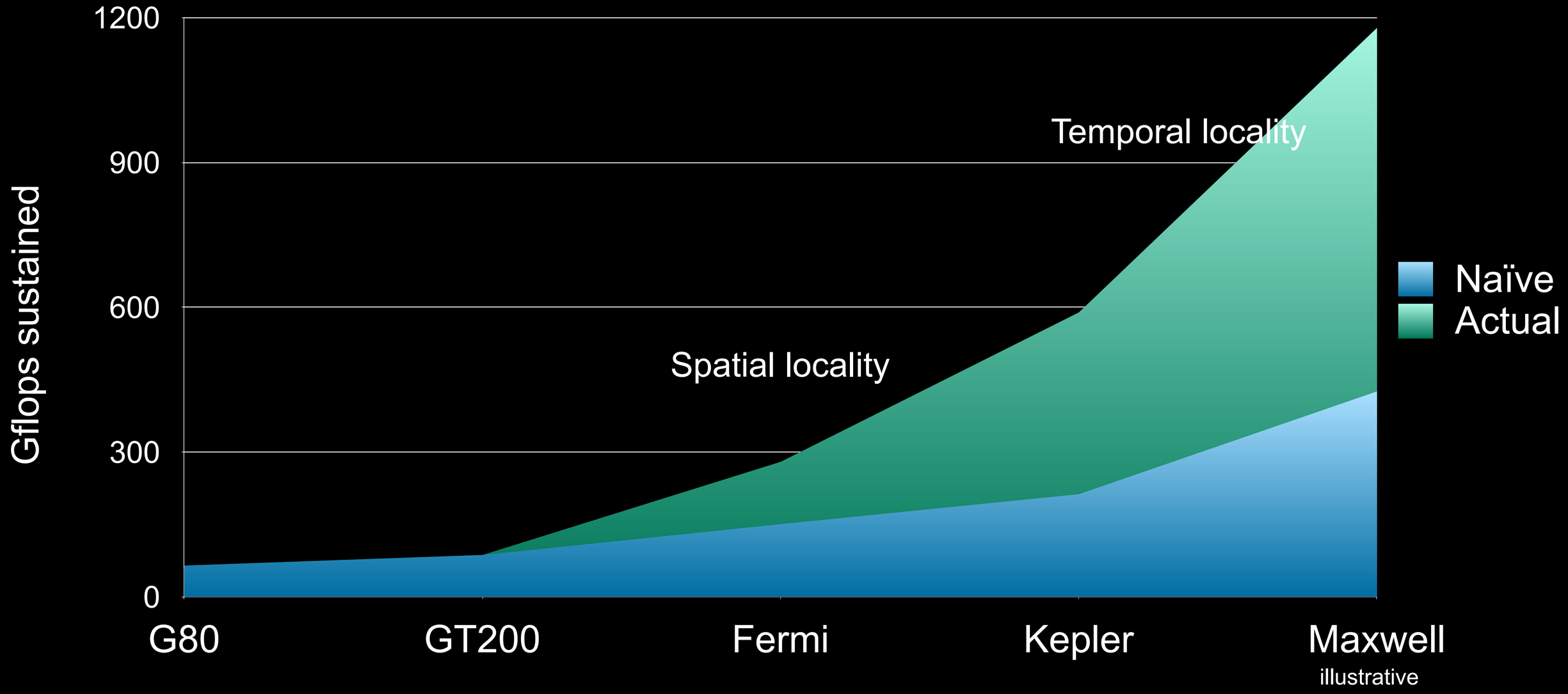
- Each photo-realistic image takes ~2 seconds
- Photo-realistic imagery requires ~200x faster
- Add physics
  - Rigid body mechanics
  - Computational fluid dynamics (smoke, water, wind)
  - Hair
  - etc.
- GPUs aren’t slowing down anytime soon

QCDNA 2014


Machine
Machine + Algo
real time (days)
10⁵
(30y)
10⁴
(3y)
10³
100
10
1987
1995
2000
year

Exploiting Locality

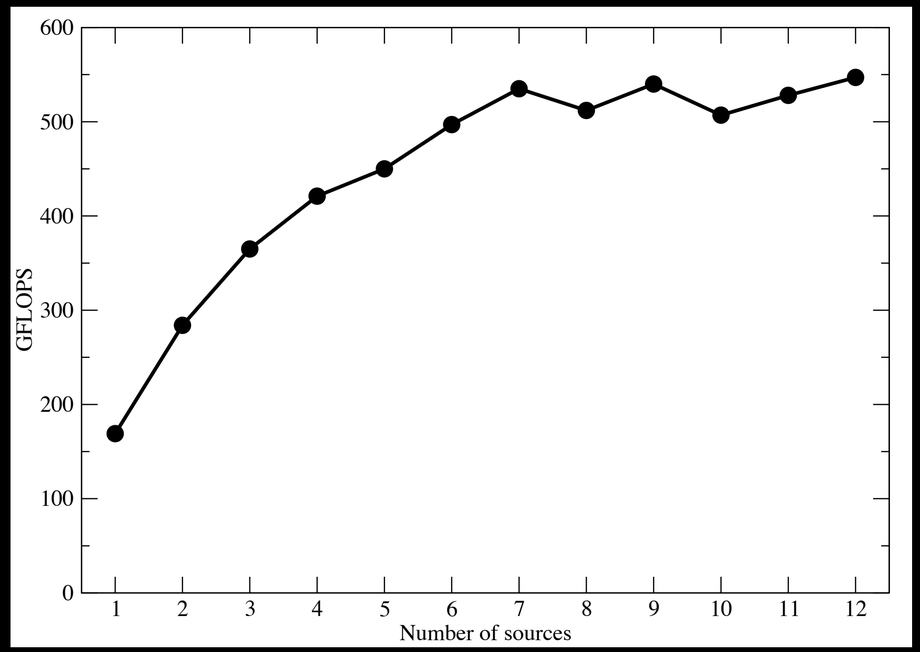
Wilson SP Dslash Performance with GPU generation



QCDNA 2014

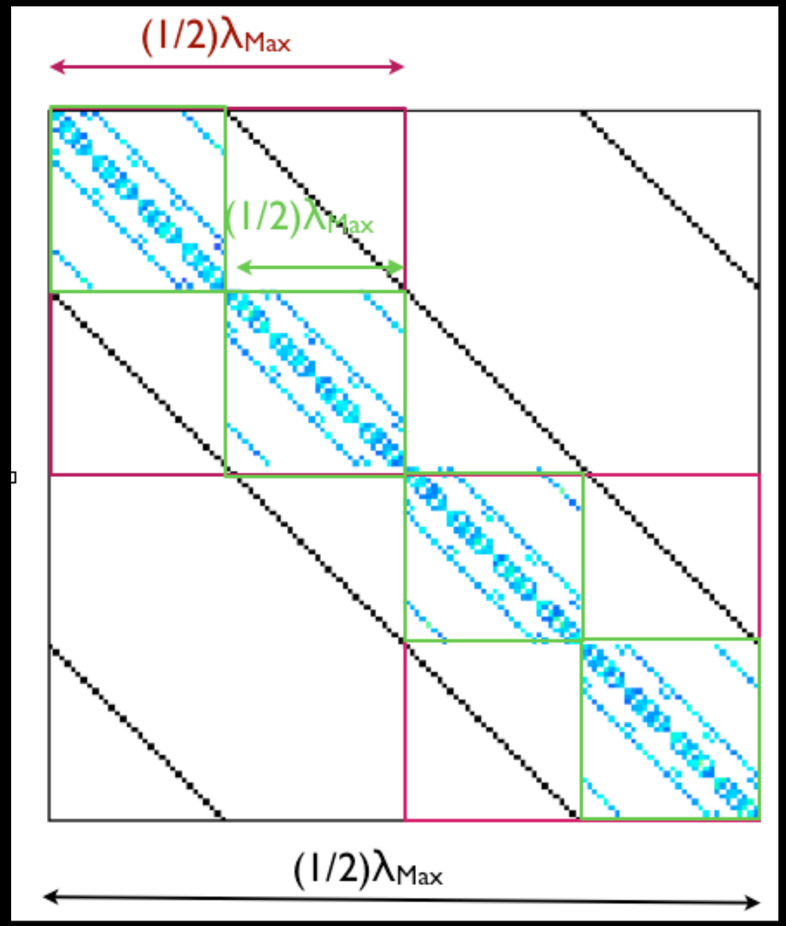
Future Directions - Locality

- Where locality does not exist, let's create it
  - E.g., Multi-source solvers
  - Staggered Dslash performance, K20X
  - Transform a memory-bound into a cache-bound problem
  - Entire solver will remain bandwidth bound



Future Directions - Communication

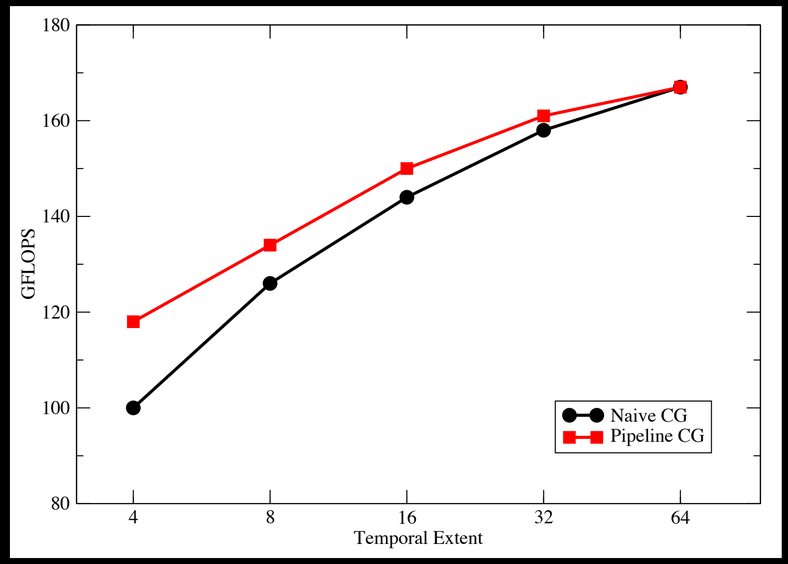
- Only scratched the surface of domain-decomposition algorithms
  - Disjoint additive
  - Overlapping additive
  - Alternating boundary conditions
  - Random boundary conditions
  - Multiplicative Schwarz
  - Precision truncation



QCDNA 2014

Future Directions - Latency

- Global sums are bad
  - Global synchronizations
  - Performance fluctuations
- New algorithms are required
  - S-step CG / BiCGstab, etc.
  - E.g., Pipeline CG vs. Naive
- One-sided communication
  - MPI-3 expands one-sided communications
  - Cray Gemini has hardware support
  - Asynchronous algorithms?
    - Random Schwarz has exponential convergence



QCDNA 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?

QCDNA 2014

Summary

- Introduction to QUDA library
- Production library for GPU-accelerated LQCD
  - Scalable linear solvers
  - Coverage for most LQCD algorithms
- Efforts now focussed on strong scaling optimal algorithms
  - Domain decomposition
  - Eigenvector solvers
  - Adaptive multigrid
  - Mixed precision
- Hierarchical *and* heterogeneous algorithm research toolbox
  - Aim for scalability *and* optimality
- Lessons today are relevant for Exascale preparation

QCDNA 2014

BACK UP SLIDES