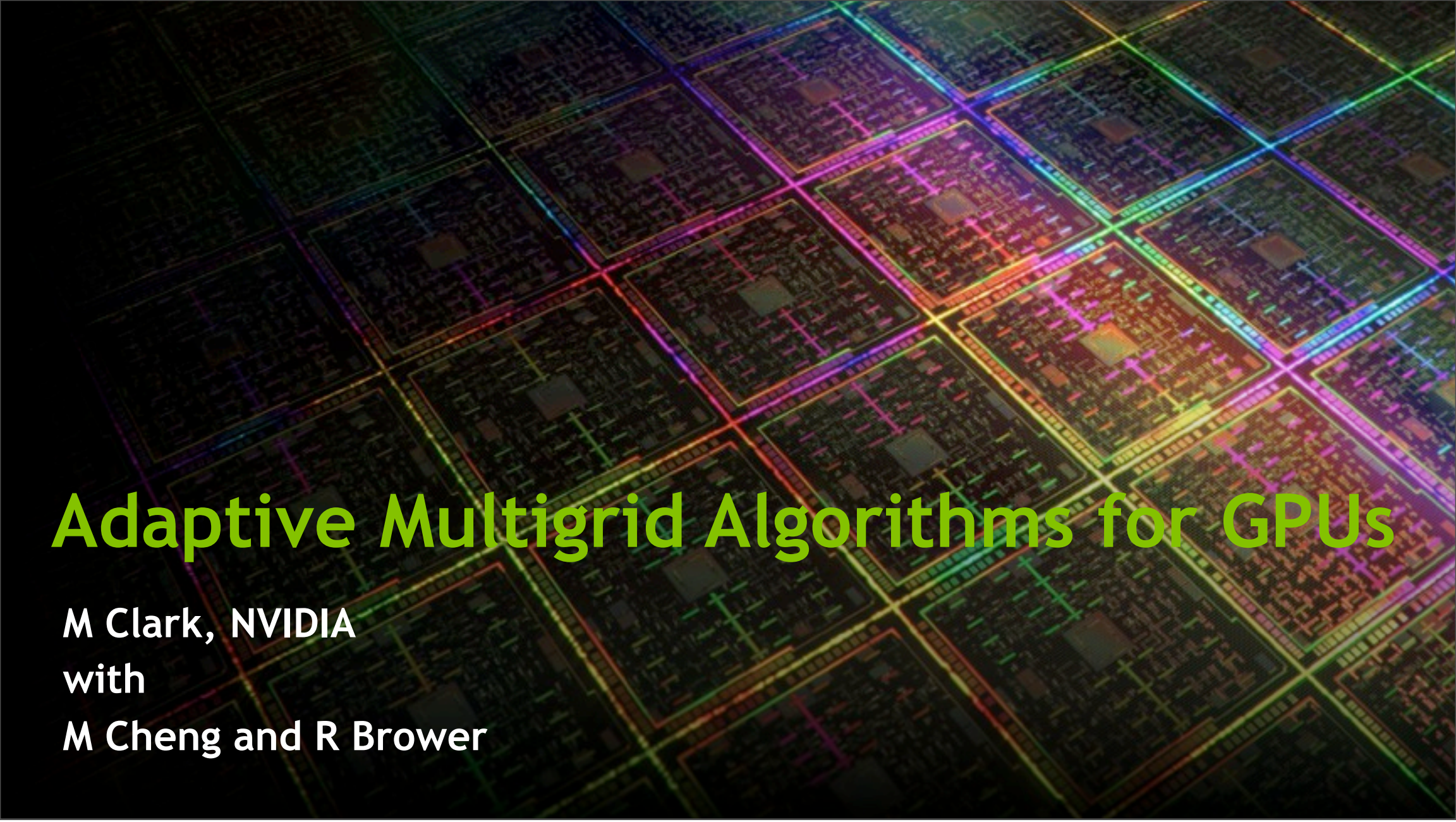


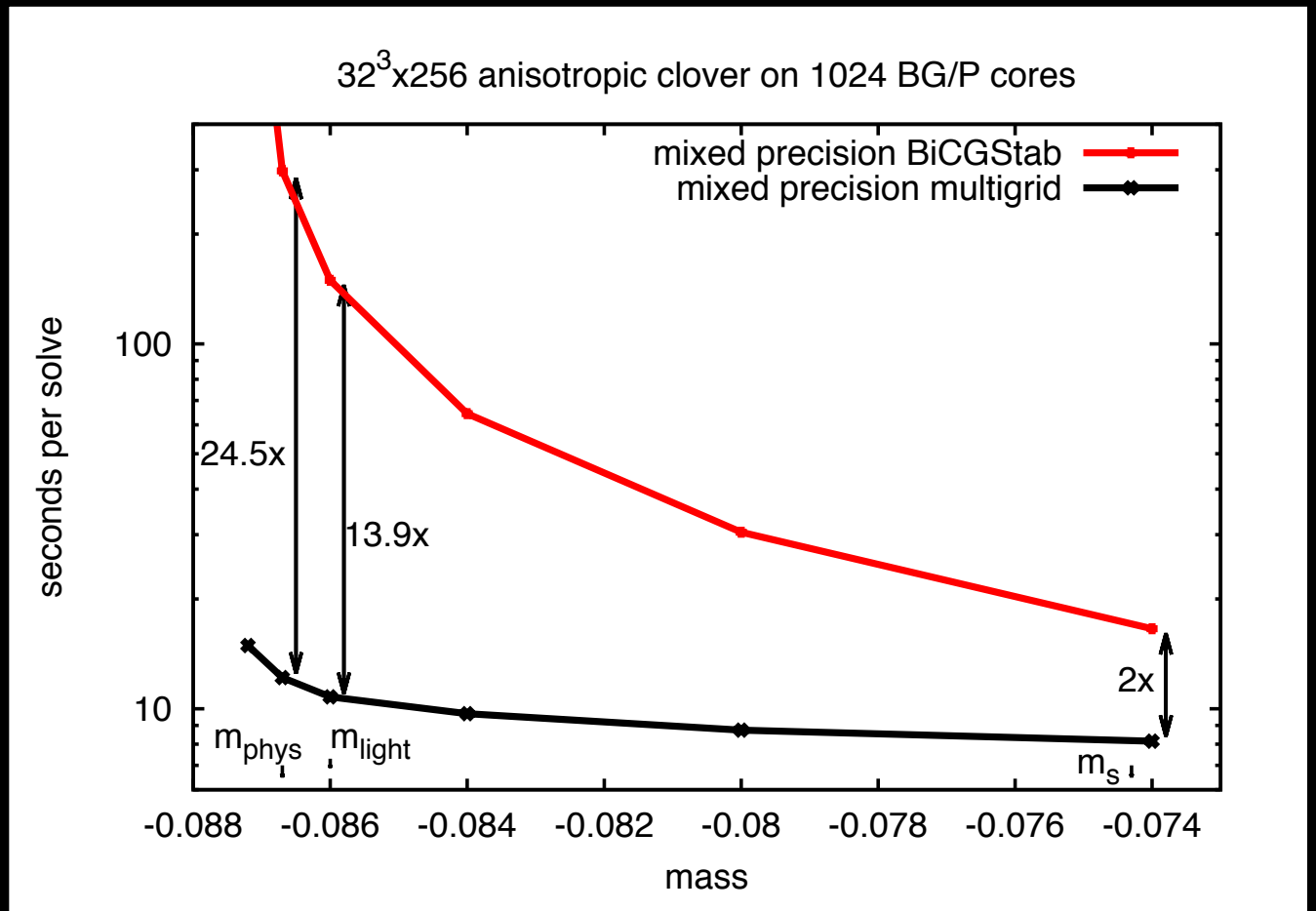
Adaptive Multigrid Algorithms for GPUs

M Clark, NVIDIA
with
M Cheng and R Brower

Contents

- Adaptive Multigrid
- QUDA Library
- Implementation
- *Extremely Preliminary* Results
- Future Work

Adaptive Multigrid



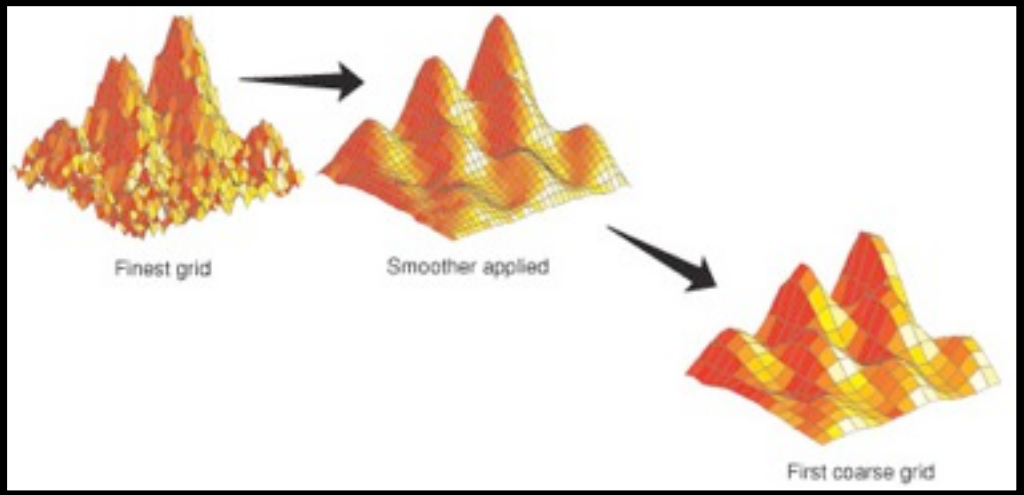
Osborn *et al*, **arXiv:1011.2775**

Hierarchical algorithms for LQCD

- Adaptive Geometric Multigrid
  - 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)
  - see also Frommer *et al* 2012
- Inexact Deflation (Lüscher 2007)
  - Equivalent to adaptive "unsmoothed" aggregation
  - Local coherence = Weak-approximation property
  - Uses an additive correction vs. MG's multiplicative correction
- Residual reduced by a constant per iteration
  - Convergence in O(1) iterations, O(*N*) per iteration
  - O(*N*) total solution cost

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

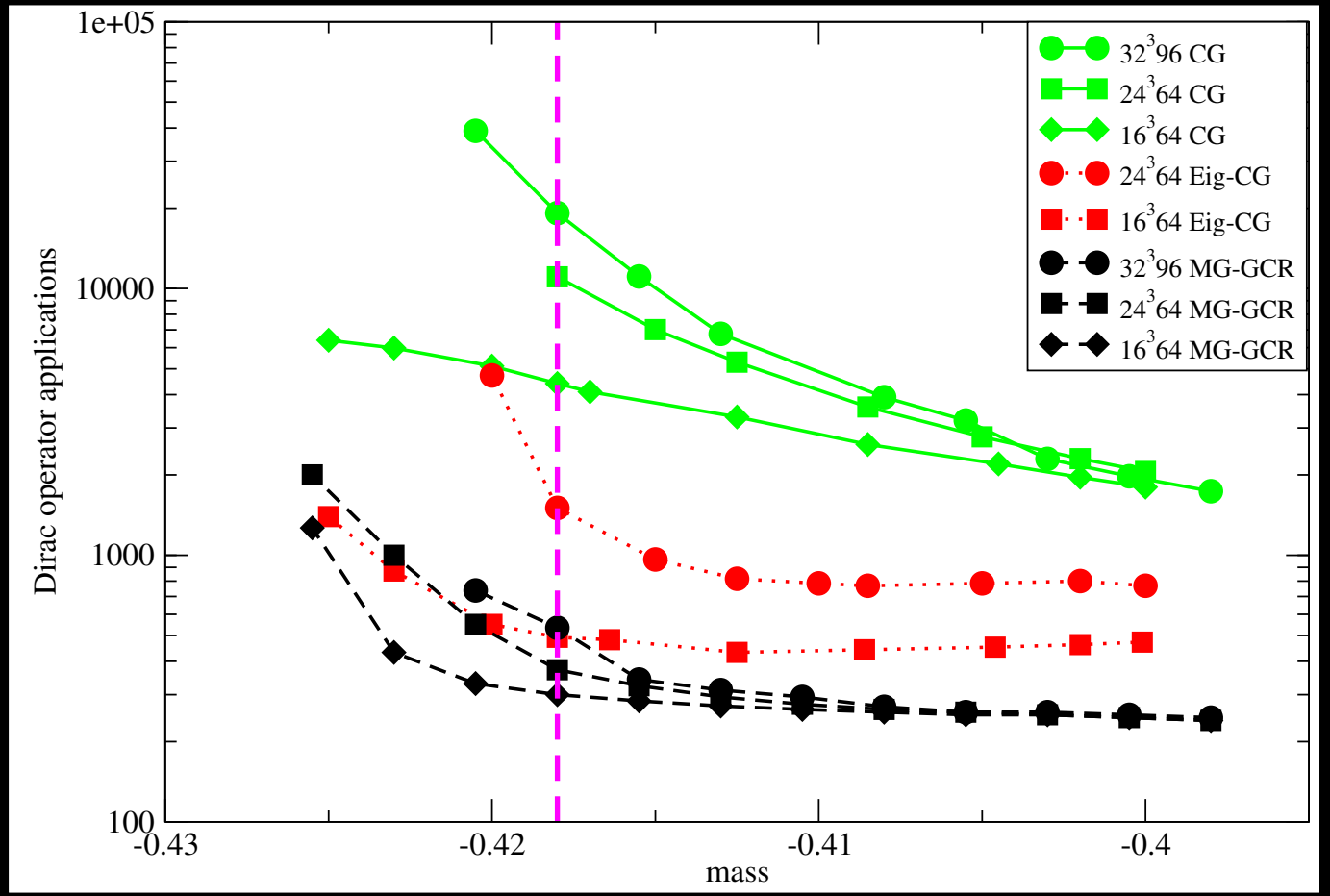


- Typically use multigrid as a preconditioner for a Krylov method
- For LQCD, we do not know the null space components that need to be preserved on the coarse grid

Adaptive Geometric Multigrid

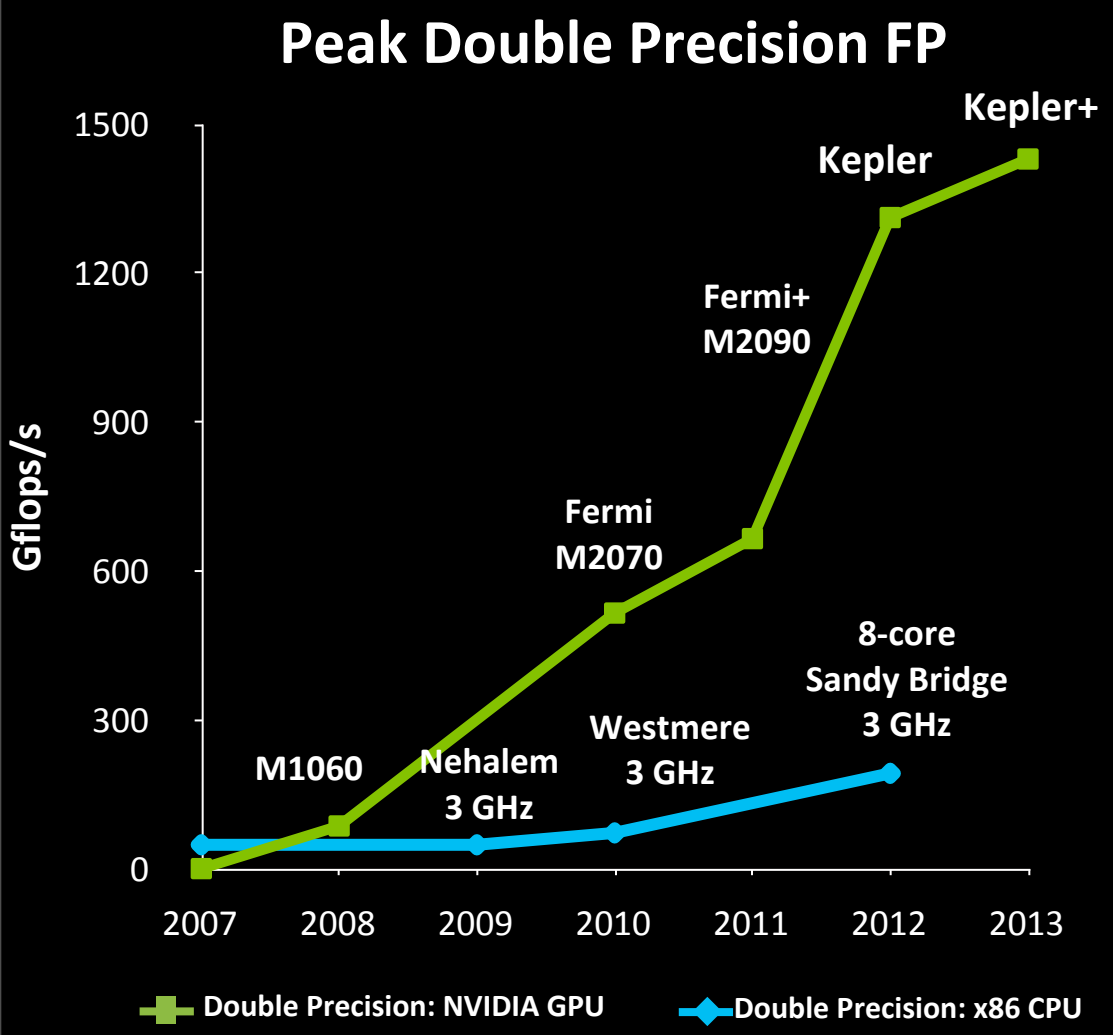
- Adaptively find candidate null-space vectors
  - Dynamically learn the null space and use this to define the prolongator
  - Algorithm is self learning
- Setup
  1. Set solver to be simple smoother
  2. Apply current solver to random vector $v_i = P(D)\ \eta_i$
  3. If convergence good enough, solver 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 ($D_c = P^\dagger\ D\ P$)
  6. Recurse on coarse problem
  7. Set solver to be augmented V-cycle, goto 2

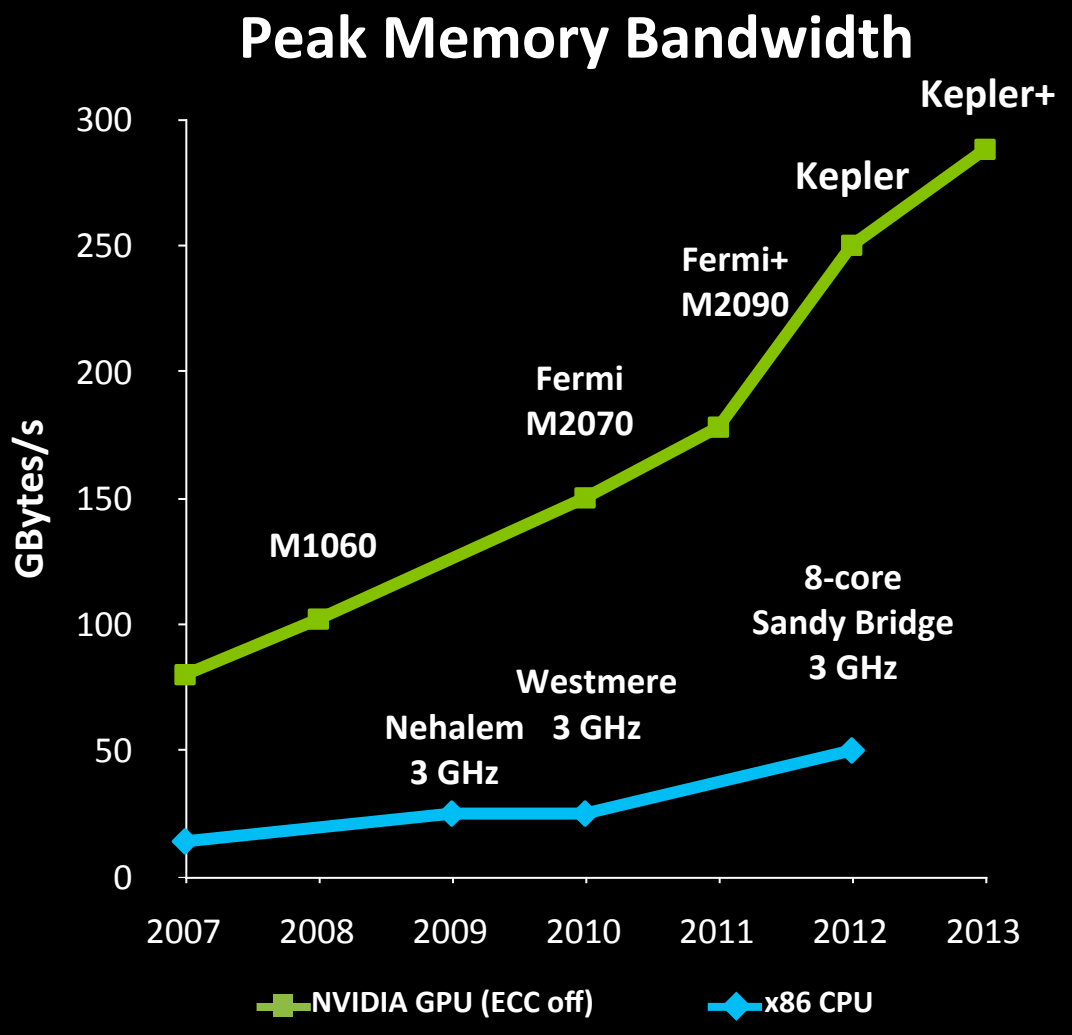
Motivation



Phys. Rev. Lett. 105, 201602 (2010)

The March of GPUs





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 several discretizations, including multi-GPU support and domain-decomposed (Schwarz) preconditioners
  - Additional performance-critical routines needed for gauge-field generation
- Maximize performance
  - Exploit physical symmetries
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Cache blocking

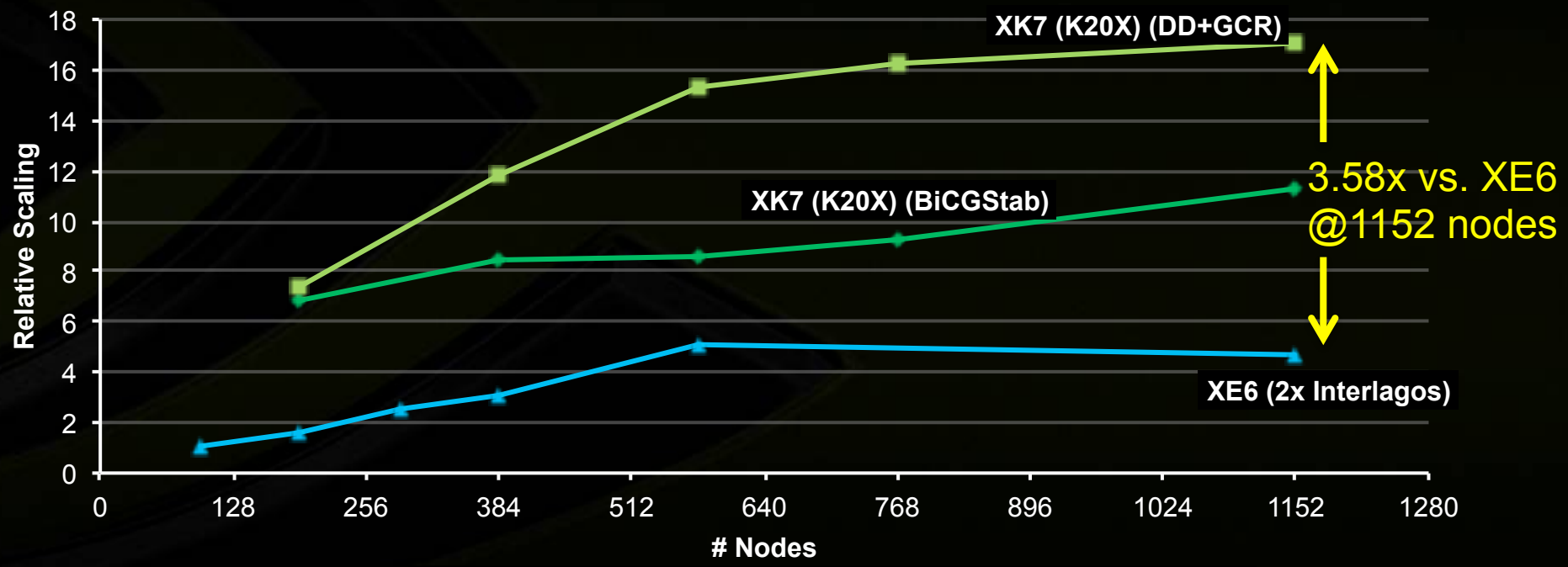
Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma

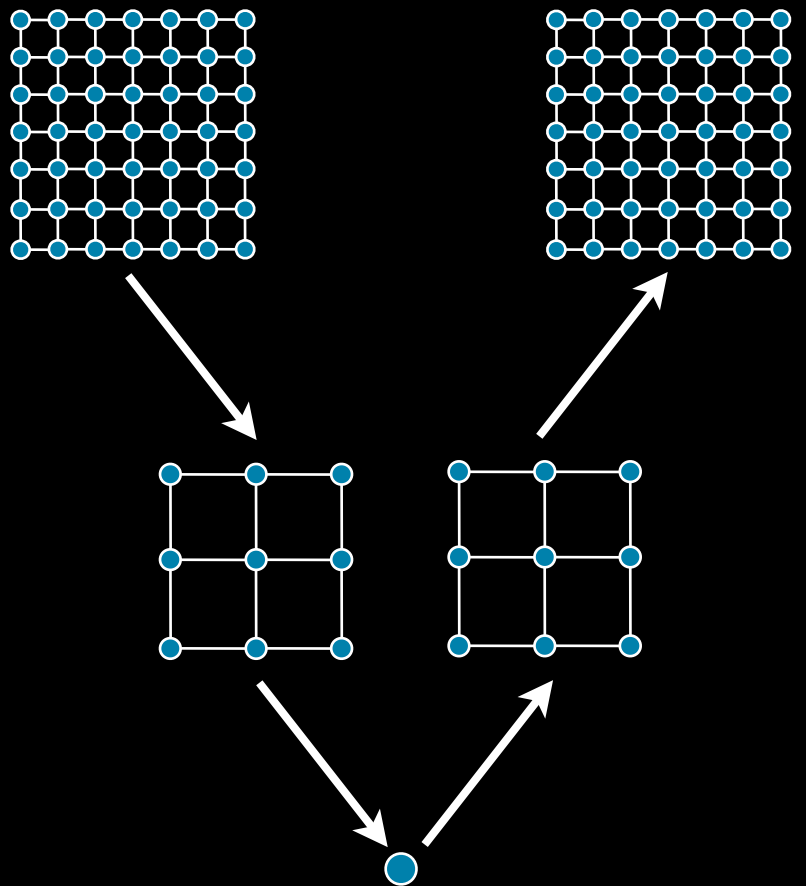
48^3x512 lattice

Relative Scaling (Application Time)

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

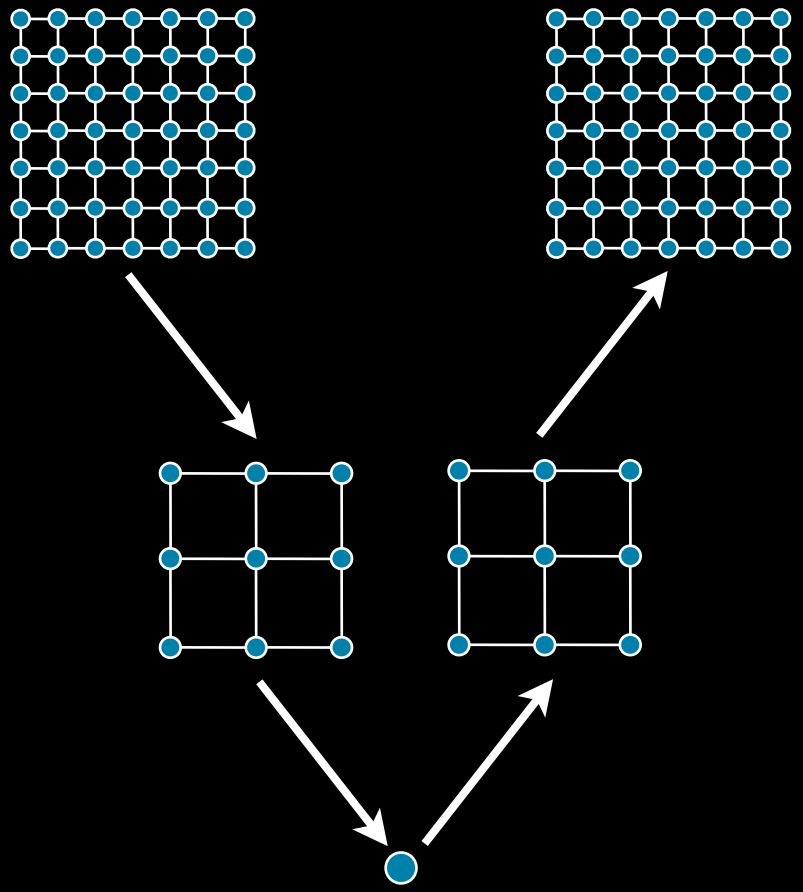


The Challenge of Multigrid on GPU



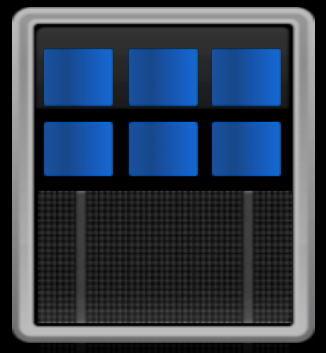
- For competitiveness, MG on GPU is a must
- GPU requirements very different from CPU
  - Each thread is slow, but O(10,000) threads per GPU
- Fine grids run very efficiently
  - High parallel throughput problem
- Coarse grids are worst possible scenario
  - More cores than degrees of freedom
  - Increasingly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

Hierarchical algorithms on heterogeneous architectures



GPU

Thousands of cores
for parallel processing



CPU

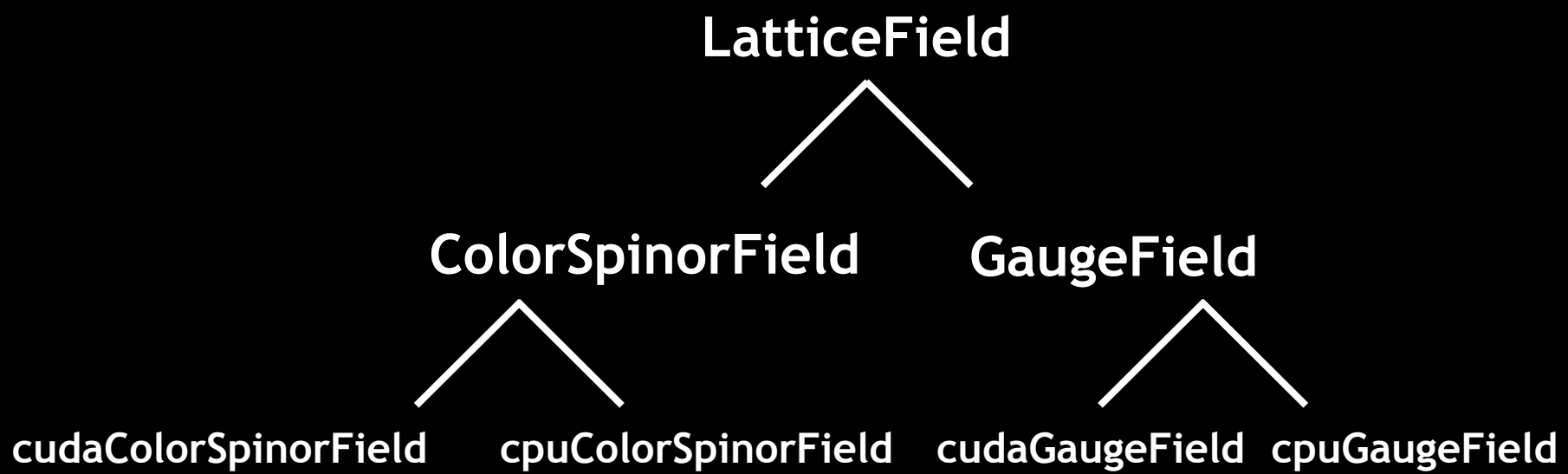
Few Cores optimized
for serial work

Design Goals

- 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

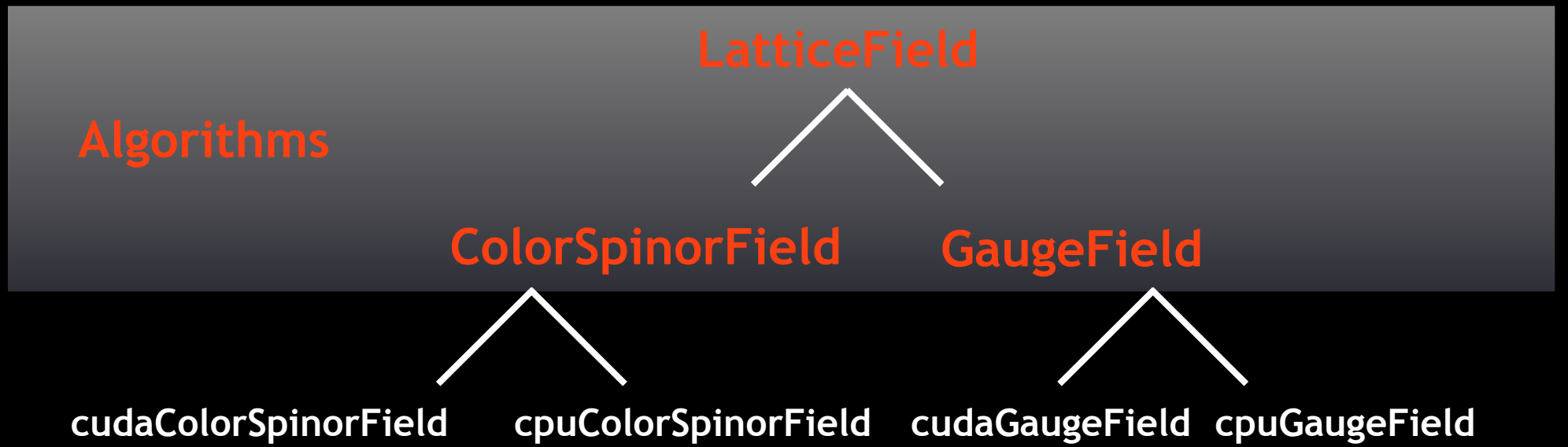
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



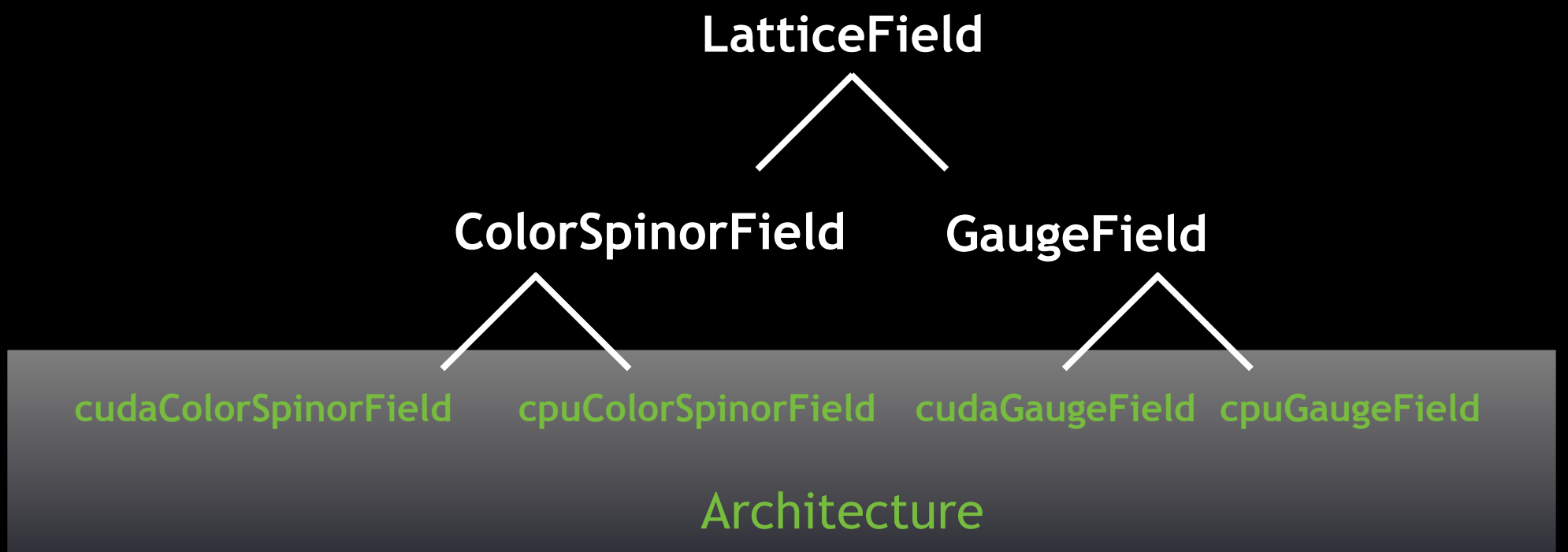
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



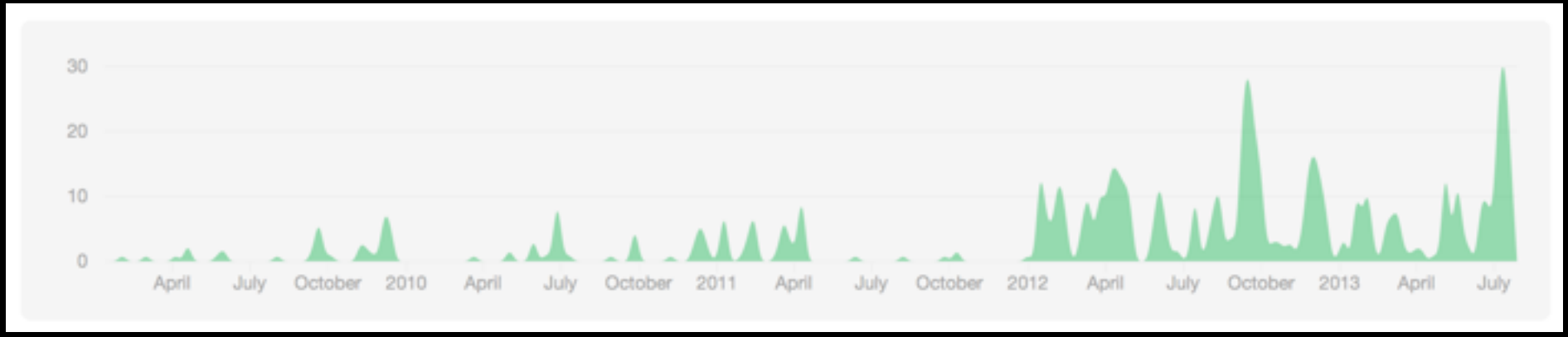
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



Multigrid and QUDA

- While envisaged to be fairly abstract
  - Rarely implemented like this in practice
  - Product of rapid development by different developers
- Adding multigrid required a lot of work
  - Improves maintainability of QUDA across the board



Writing the same code for two architectures

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

```
// Applies the grid prolongation operator (coarse to fine)
 template <class FineSpinor, class CoarseSpinor>
 void prolongate(FineSpinor &out, const CoarseSpinor &in,
const int *geo_map, const int *spin_map) {

   for (int x=0; x<out.Volume(); x++) {
     for (int s=0; s<out.Nspin(); s++) {
       for (int c=0; c<out.Ncolor(); c++) {
         out(x, s, c) = in(geo_map[x], spin_map[s], c);
       }
     }
   }

 }
```

CPU

```
// Applies the grid prolongation operator (coarse to fine)
  template <class FineSpinor, class CoarseSpinor>
  __global__ void prolongate(FineSpinor out, const
CoarseSpinor in, const int *geo_map, const int *spin_map) {

    int x = blockIdx.x*blockDim.x + threadIdx.x;
    for (int s=0; s<out.Nspin(); s++) {
      for (int c=0; c<out.Ncolor(); c++) {
        out(x, s, c) = in(geo_map[x], spin_map[s], c);
      }
    }

  }
```

GPU

Current Status

- First multigrid solver working in QUDA as of last Friday
- Some components still on CPU only

	GPU	CPU
Fine grid operator	✓	
Block Orthogonalization		✓
Prolongator	✓	✓
Restrictor	✓	✓
Construct coarse gauge field		✓
Coarse grid operator		✓
Vector BLAS	✓	✓

- Designed to interoperate with J. Osborn's *qopqdp* implementation
  - Can verify algorithm correctness, and share null space vectors

Very preliminary two-level results

level 0 $24^3 \times 128$
level 1 $8^3 \times 16$
D on GPU
D_c, P, R on CPU

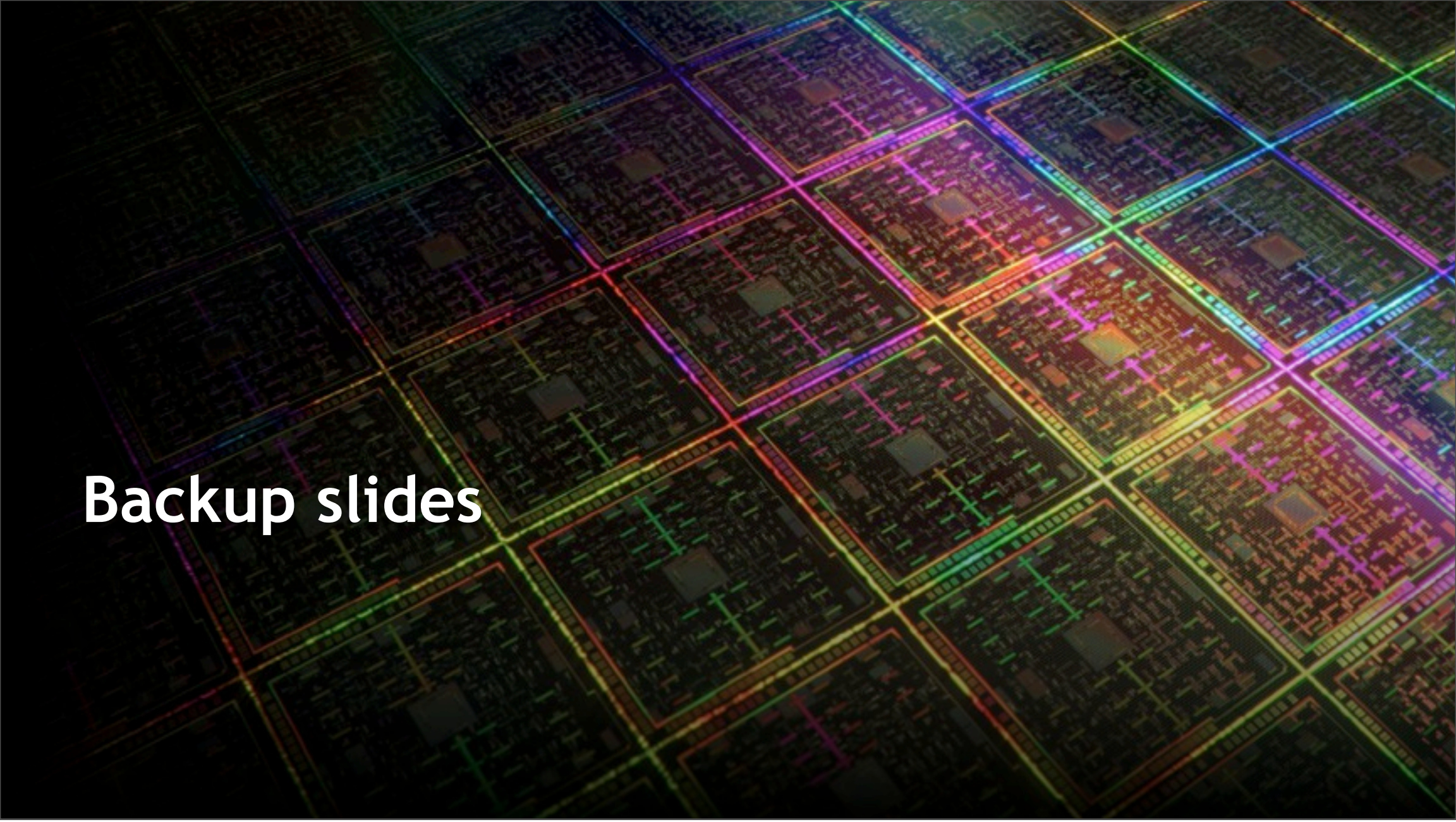
QUDA as a Hierarchical Algorithm Tool

- Lots of interesting questions to be explored
- Exploit closer coupling of precision and algorithm
  - QUDA designed for complete run-time specification of precision at any point
  - 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 approaches likely vital for strong scaling
  - DD solvers are effective for high-frequency dampening
  - Overlapping domains likely more important at coarser scales

Summary

mclark at nvidia dot com

- Introduction to multigrid on QUDA
- Basic framework complete, proof of concept
- Still *lots* of work to do
  - Most of the nitty gritty details worked out
  - Now time for fun
- Beta testing for end of year
  - Chroma Wilson / Wilson-clover support first
- Lessons today are relevant for Exascale preparation

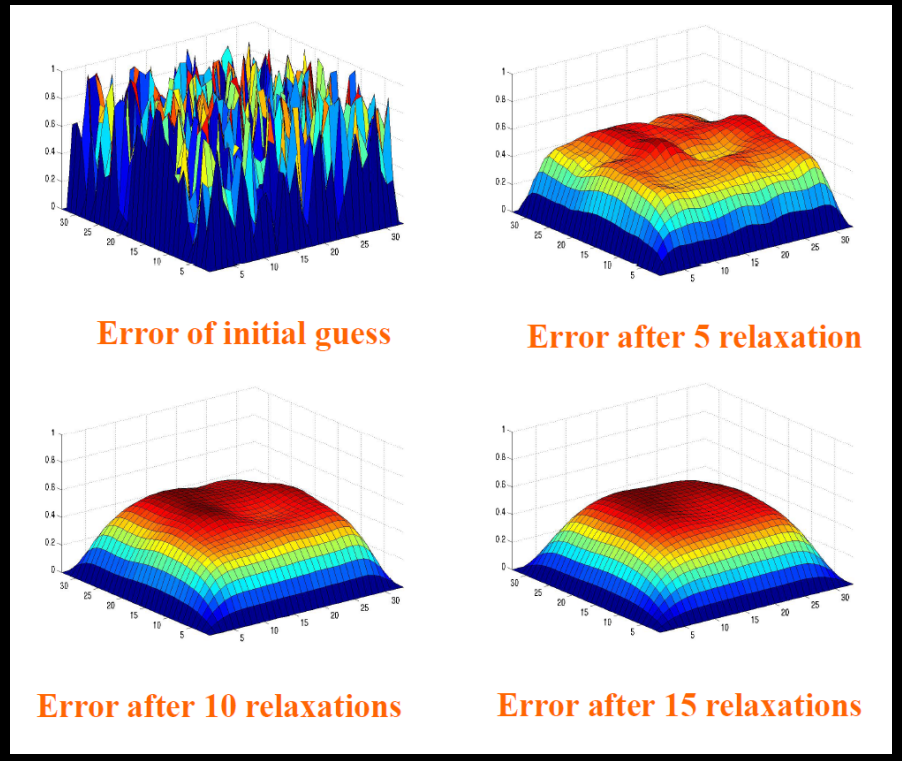

Backup slides

Failure of Geometric Multigrid for LQCD

2-d Laplace operator error
with Gauss-Seidel interation

2-d U(1) Wilson-Dirac operator
after 200 Gauss-Seidel iterations

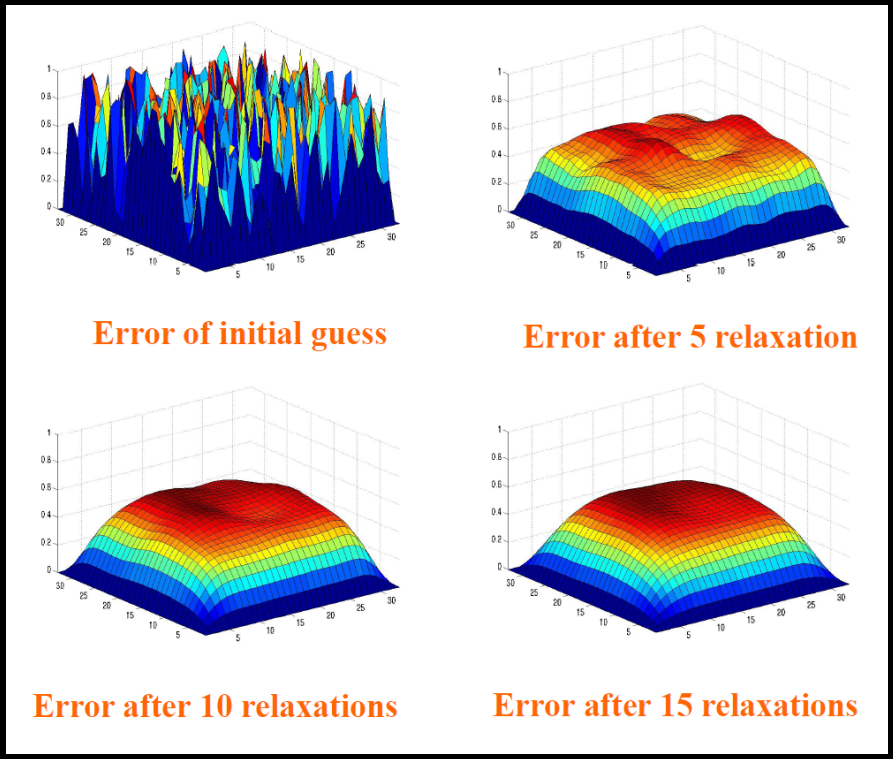
Failure of Geometric Multigrid for LQCD

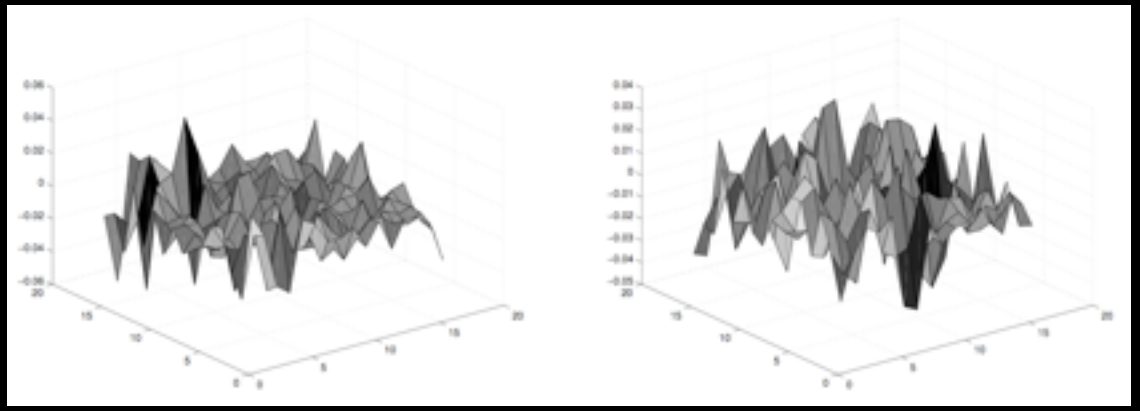


2-d Laplace operator error
with Gauss-Seidel interation

2-d U(1) Wilson-Dirac operator
after 200 Gauss-Seidel iterations

Failure of Geometric Multigrid for LQCD





2-d Laplace operator error
with Gauss-Seidel interation

2-d U(1) Wilson-Dirac operator
after 200 Gauss-Seidel iterations

The Need for Just-In-Time Compilation

- 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)

```
// Applies the grid prolongation operator (coarse to fine)
  template <class FineSpinor, class CoarseSpinor>
  __global__ void prolongate(FineSpinor out, const CoarseSpinor in, const
int *geo_map, const int *spin_map) {

    int x = blockIdx.x*blockDim.x + threadIdx.x;
    for (int s=0; s<out.Nspin(); s++) {
      for (int c=0; c<out.Ncolor(); c++) {
         out(x, s, c) = in(geo_map[x], spin_map[s], c);
      }
    }

  }
```

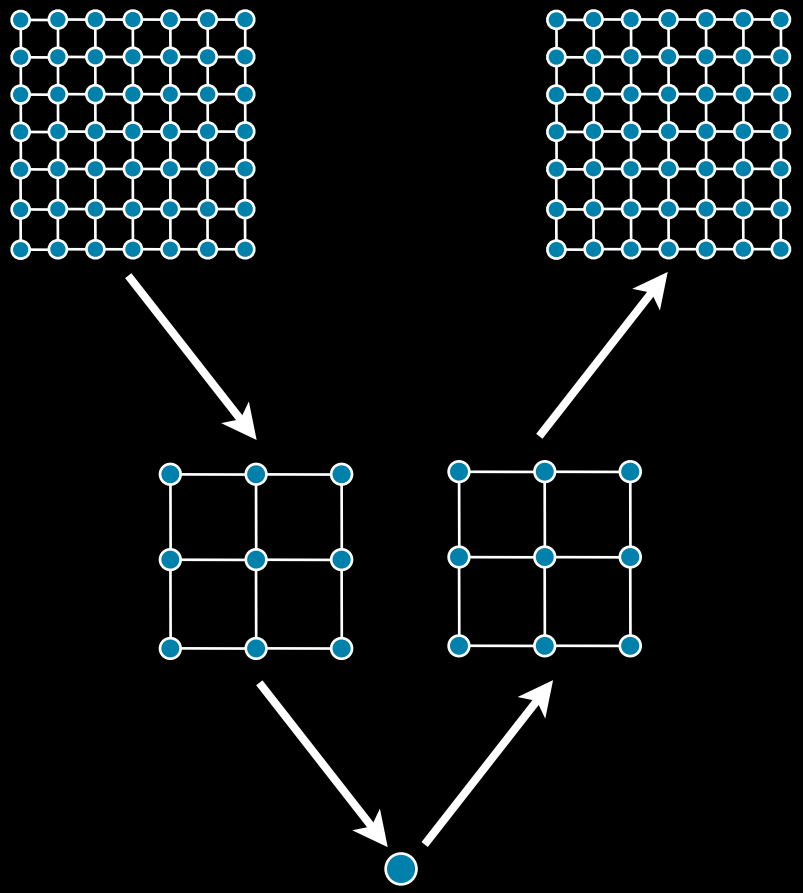
The Need for Just-In-Time Compilation

- Possible solutions
  - Template over every possible $N_v \otimes$ precision for each MG kernel
  - One thread per colour matrix row (inefficient for N_v mod 32 $\neq$ 0)
  - Compile necessary kernel at runtime

```
// Applies the grid prolongation operator (coarse to fine)
  template <class FineSpinor, class CoarseSpinor, int Ncolor, int Nspin>
  __global__ void prolongate(FineSpinor out, const CoarseSpinor in, const
int *geo_map, const int *spin_map) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    for (int s=0; s<Nspin; s++) {
      for (int c=0; c<Ncolor; c++) {
        out(x, s, c) = in(geo_map[x], spin_map[s], c);
      }
    }
  }
```

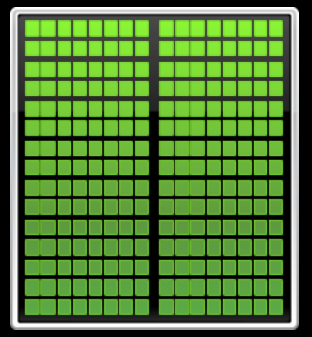
- JIT support will be coming in CUDA 6.x
  - Final performant implementation will likely require this

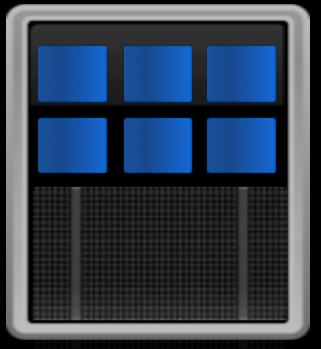
Heterogeneous Updating Scheme



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

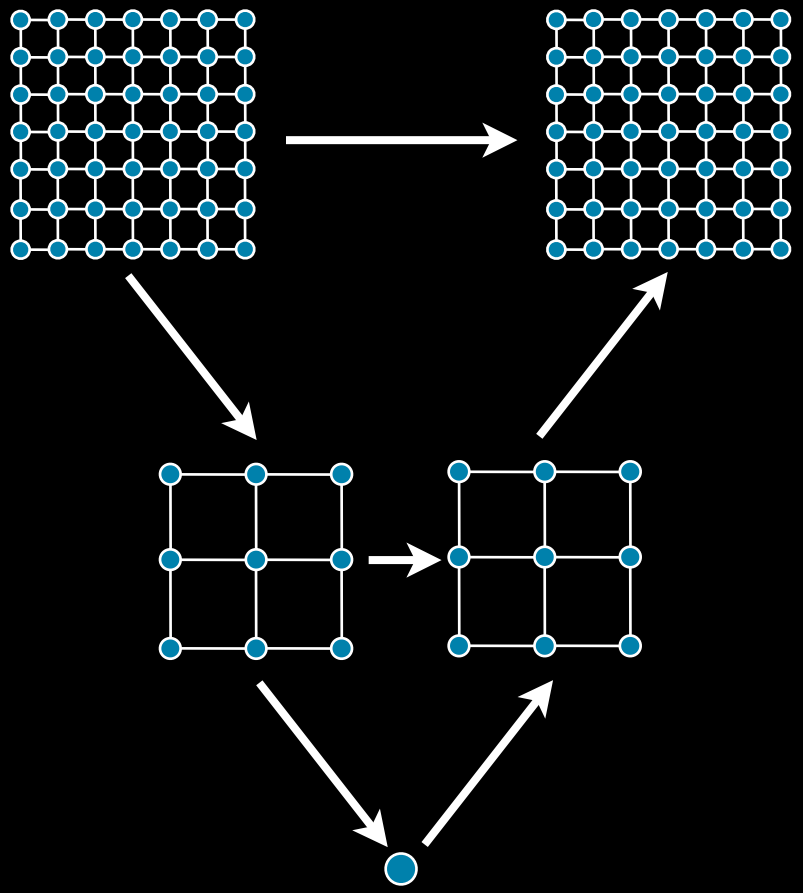
GPU





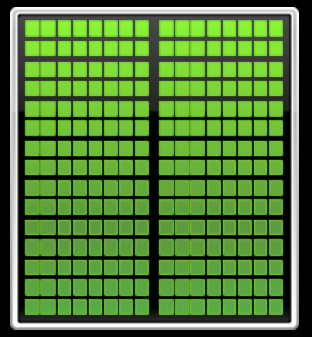
CPU

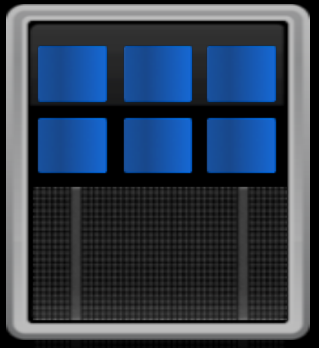
Heterogeneous Updating Scheme



- 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

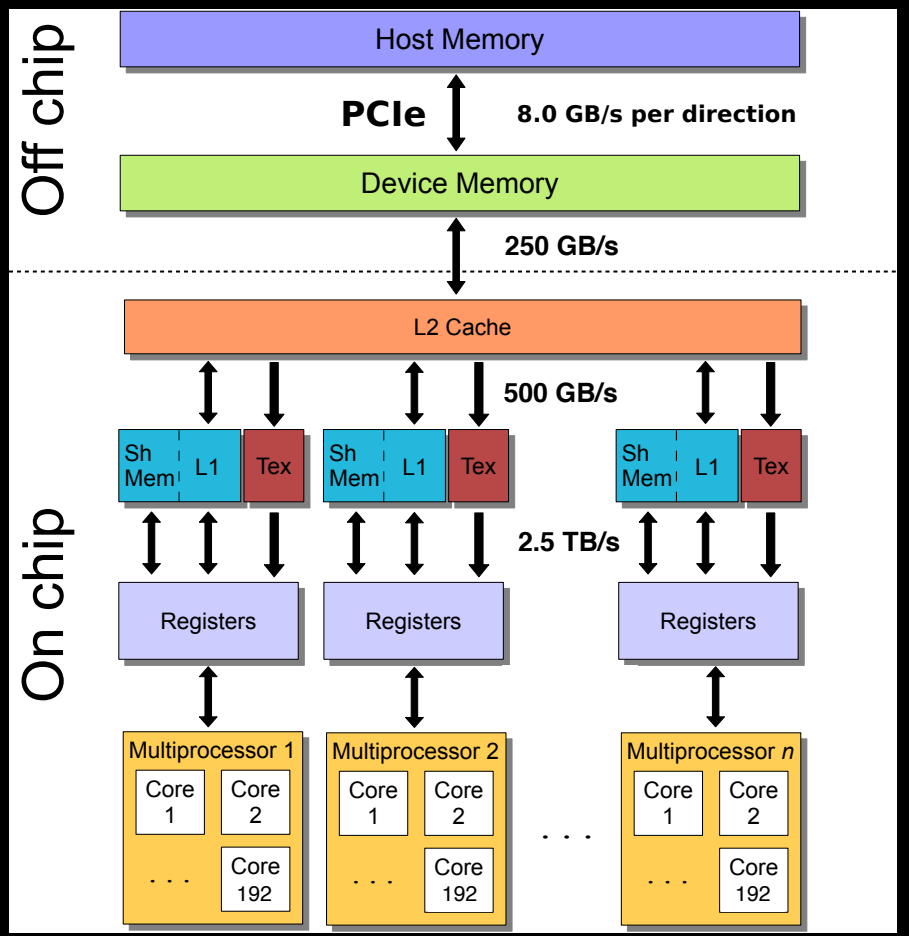
GPU





CPU

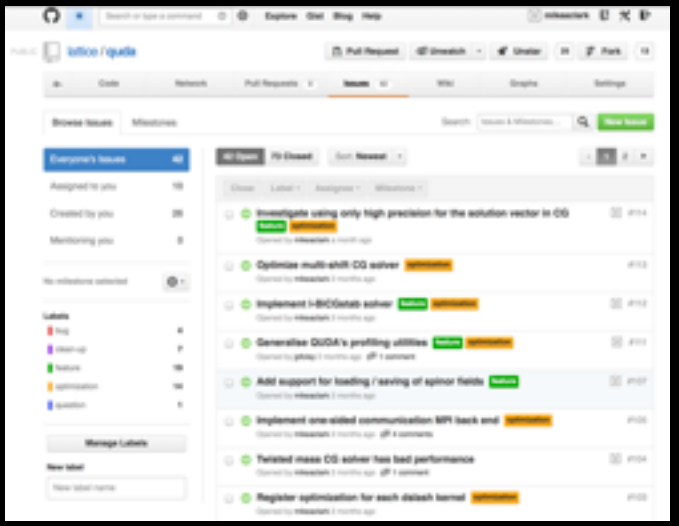
The Kepler Architecture

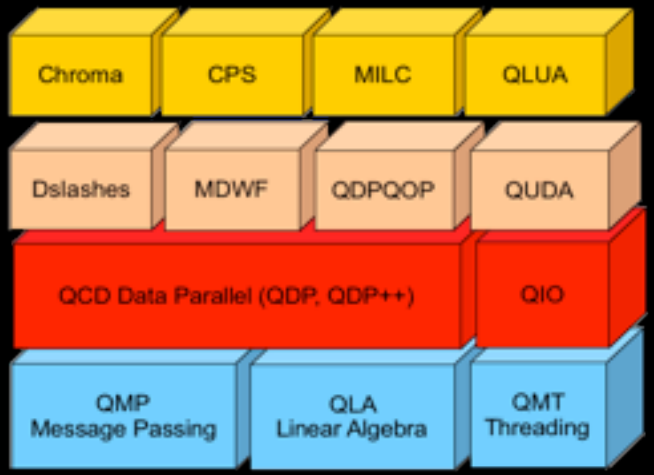


- Kepler K20X
  - 2688 processing cores
  - 3995 SP Gflops peak (665.5 fma)
  - Effective SIMD width of 32 threads (warp)
- Deep memory hierarchy
  - As we move away from registers
    - Bandwidth decreases
    - Latency increases
  - Each level imposes a minimum arithmetic intensity to achieve peak
- Limited on-chip memory
  - 65,536 32-bit registers, 255 registers per thread
  - 48 KiB shared memory
  - 1.5 MiB L2

QUDA is community driven

- Ron Babich (NVIDIA)
- Kip Barros (LANL)
- Rich Brower (Boston University)
- Michael Cheng (Boston University)
- Justin Foley (University of Utah)
- 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)

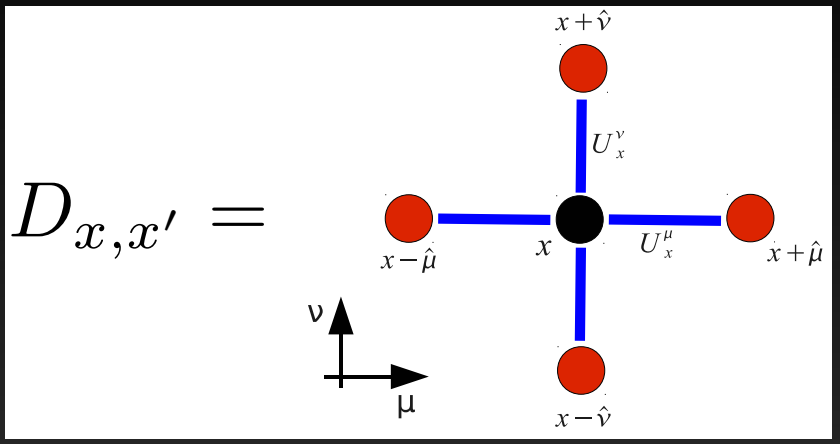




QUDA Mission Statement

- QUDA is
  - a library enabling legacy applications to run on GPUs
  - open source so anyone can join the fun
  - evolving
    - more features
    - cleaner, easier to maintain
  - a research tool into how to reach the exascale
    - Lessons learned are mostly (platform) agnostic
    - Domain-specific knowledge is key
    - Free from the restrictions of DSLs, e.g., multigrid in QDP

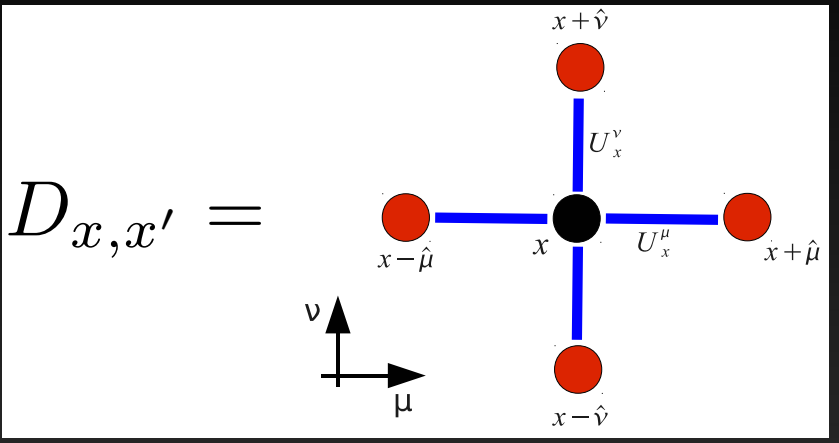
Mapping the Wilson Dslash to CUDA



- Assign a single space-time point to each thread
  - V = XYZT threads
  - V = 24^4 => 3.3×10^6 threads
  - Fine-grained parallelization
- 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)
- Arithmetic intensity
  - 1320 floating point operations per site
  - 1440 bytes per site (single precision)
  - 0.92 naive arithmetic intensity

Mapping the Wilson Dslash to CUDA

- Assign a single space-time point to each thread
  - V = XYZT threads
  - V = 24^4 => 3.3×10^6 threads
  - Fine-grained parallelization
- 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)
- Arithmetic intensity
  - 1320 floating point operations per site
  - 1440 bytes per site (single precision)
  - 0.92 naive arithmetic intensity



Tesla K20X	
Gflops	3995
GB/s	250
AI	16

bandwidth bound

Krylov Solver Implementation

- Complete solver **must** be on GPU
  - Transfer b to GPU (reorder)
  - Solve Mx=b
  - Transfer x to CPU (reorder)
- Entire algorithms must run on GPUs
  - Time-critical kernel is the stencil application (SpMV)
  - Also require BLAS level-1 type operations
    - e.g., AXPY operations: b += ax, NORM operations: c = (b,b)
    - Roll our own kernels for kernel fusion and custom precision

conjugate gradient

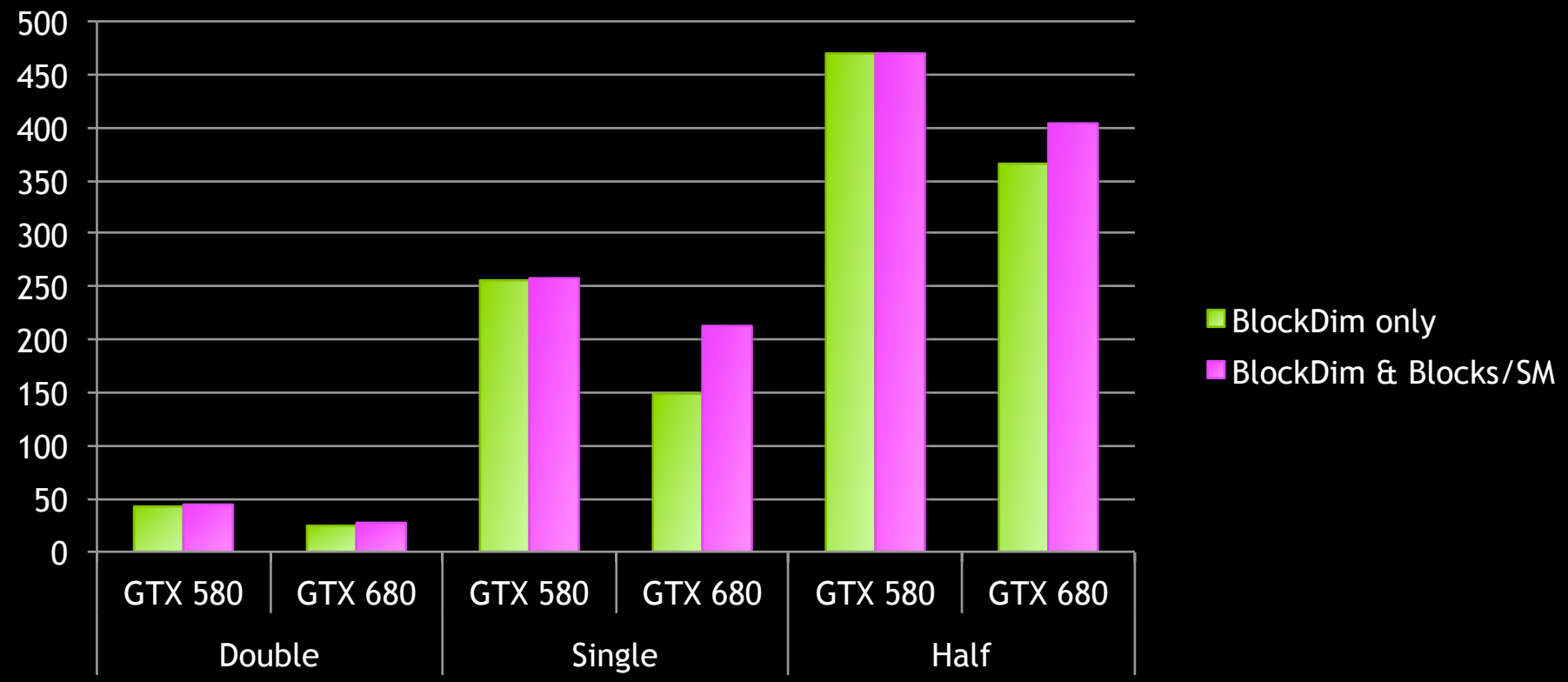
$$
\begin{aligned}
&\text{while } (|\mathbf{r}_k| > \varepsilon) \ \{ \\
&\quad \beta_k = (\mathbf{r}_k, \mathbf{r}_k)/(\mathbf{r}_{k-1}, \mathbf{r}_{k-1}) \\
&\quad \mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k \mathbf{p}_k \\
&\quad \alpha = (\mathbf{r}_k, \mathbf{r}_k)/(\mathbf{p}_{k+1}, A\mathbf{p}_{k+1})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

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:

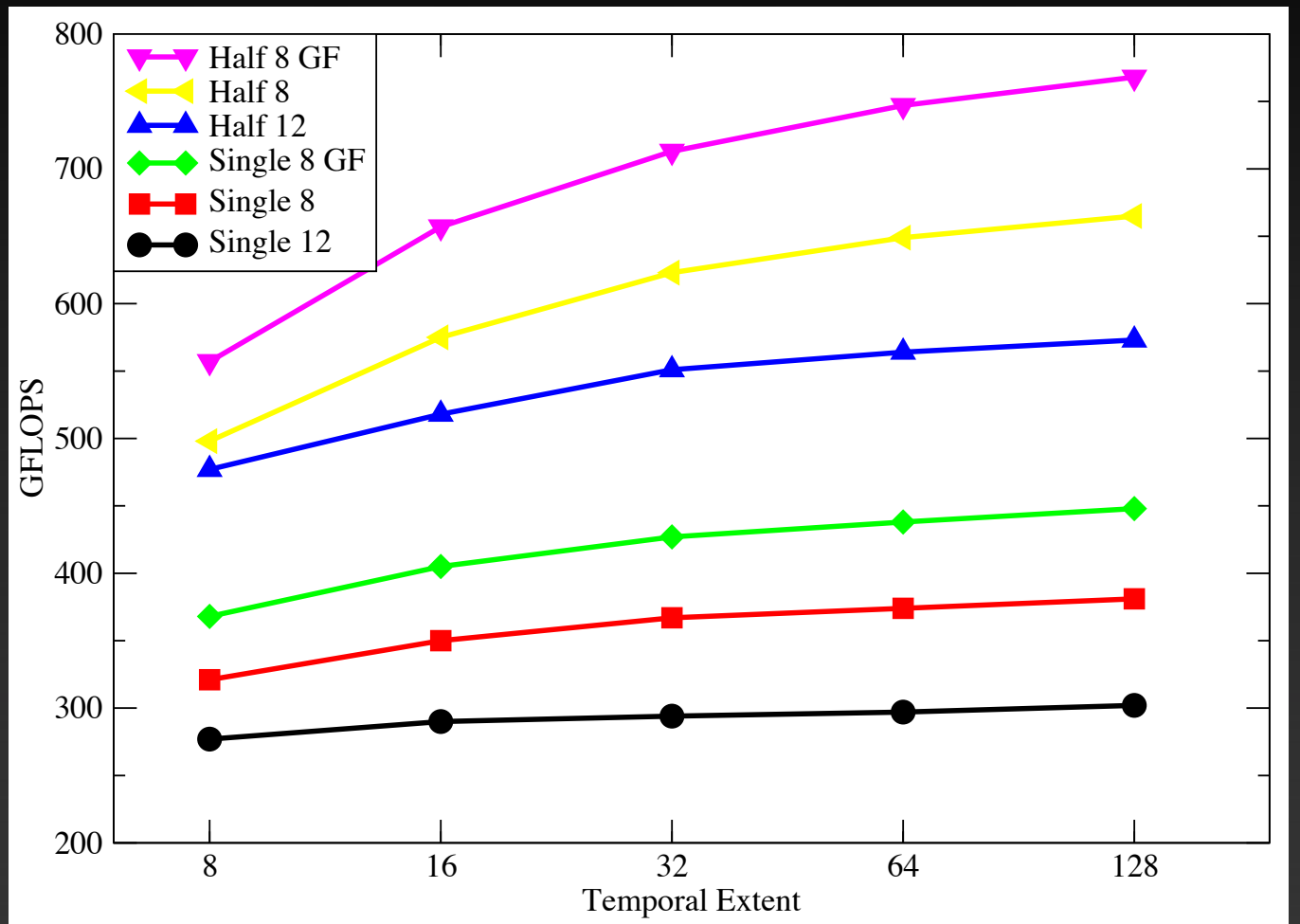
```
MyKernelWrapper *k = new MyKernelWrapper(a, b, c);
k->apply();  // <-- automatically tunes if necessary
```

- 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

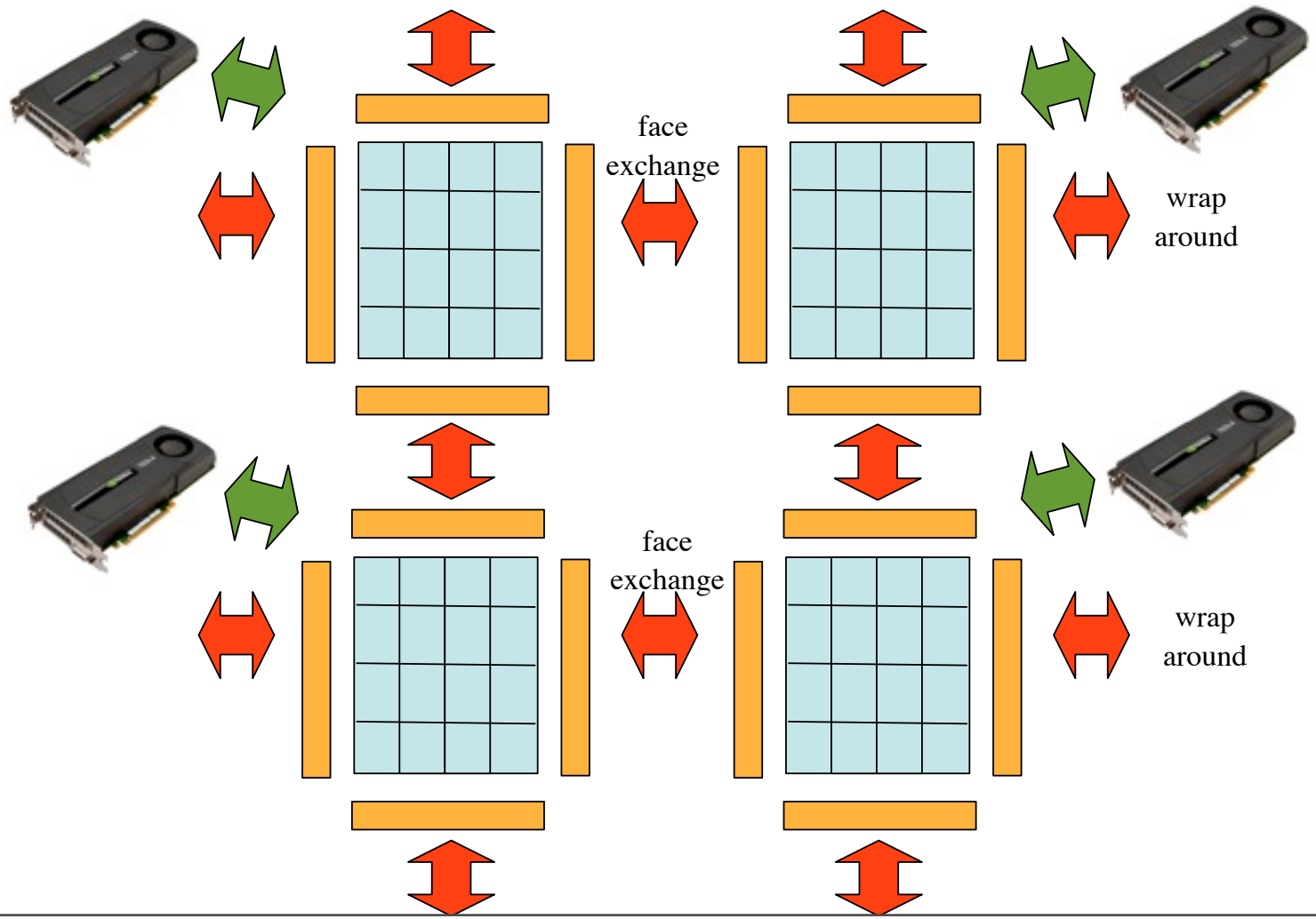
- Invoke the kernel (tuning if necessary):
  - apply()
- Save and restore state before/after tuning:
  - preTune(), postTune()
- Advance to next set of trial parameters in the tuning:
  - advanceGridDim(), advanceBlockDim(), advanceSharedBytes()
  - advanceTuneParam() // simply calls the above by default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

Kepler Wilson-Dslash Performance



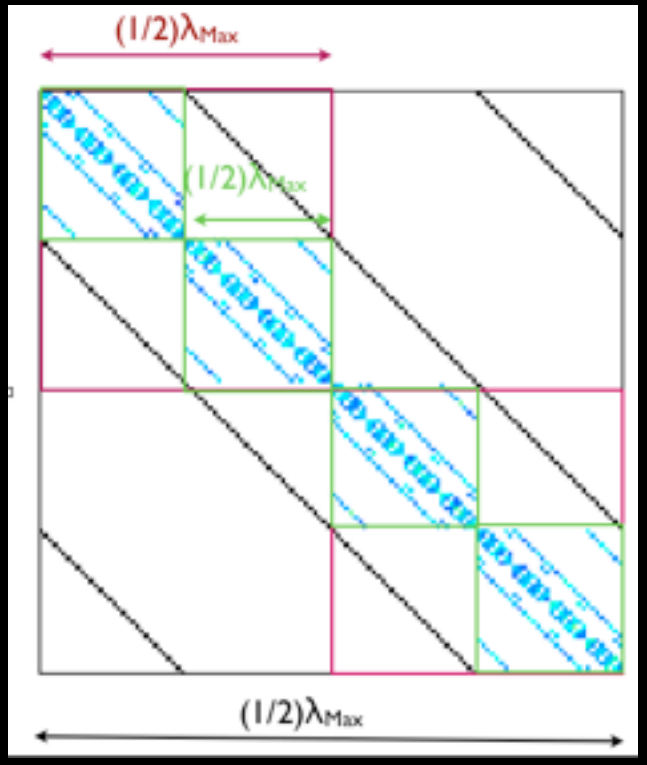
V = 24^3xT K20X Dslash

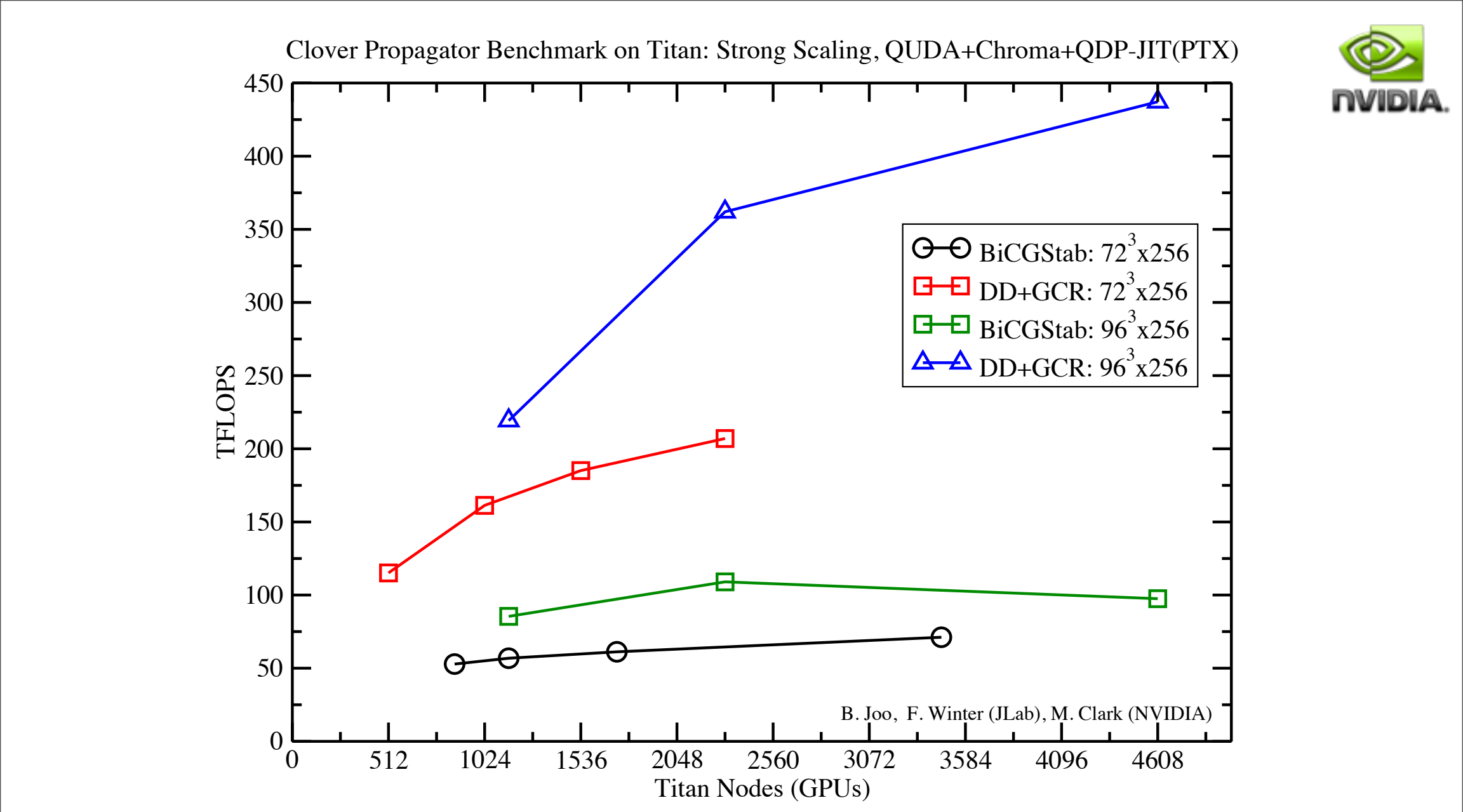
Multi-dimensional lattice decomposition



Domain Decomposition

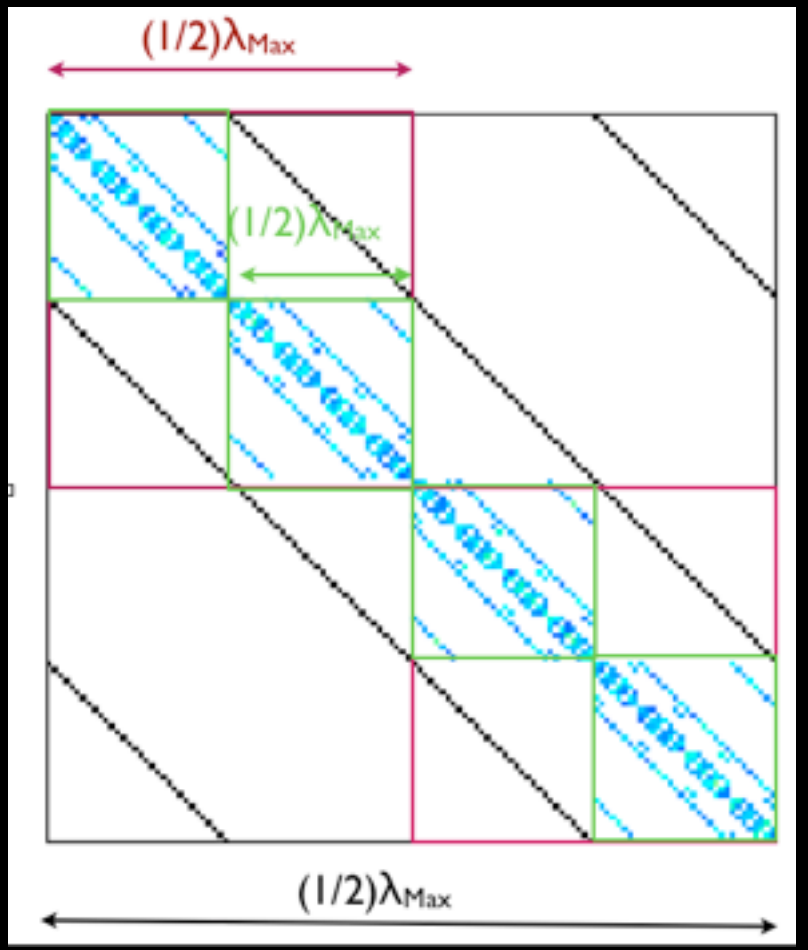
- Non-overlapping blocks - simply have to switch off inter-GPU communication
- Preconditioner is a gross approximation
  - Use an iterative solver to solve each domain system
  - Require only 10 iterations of domain solver $\Longrightarrow$ 16-bit
  - Need to use a flexible solver $\Longrightarrow$ GCR
- Block-diagonal preconditoner impose λ cutoff
- Finer Blocks lose long-wavelength/low-energy modes
  - keep wavelengths of ~ $O(\Lambda_{QCD}^{-1})$, $\Lambda_{QCD}^{-1} \sim 1fm$
- Aniso clover: (a_s=0.125fm, a_t=0.035fm) $\Longrightarrow$ 8^3x32 blocks are ideal
  - 48^3x512 lattice: 8^3x32 blocks $\Longrightarrow$ 3456 GPUs



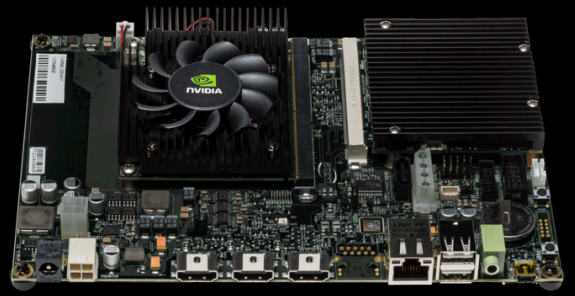

Clover Propagator Benchmark on Titan: Strong Scaling, QUDA+Chroma+QDP-JIT(PTX)
TFLOPS
Titan Nodes (GPUs)
0
50
100
150
200
250
300
350
400
450
0
512
1024
1536
2048
2560
3072
3584
4096
4608
BiCGStab: 72^3x256
DD+GCR: 72^3x256
BiCGStab: 96^3x256
DD+GCR: 96^3x256
B. Joo, F. Winter (JLab), M. Clark (NVIDIA)

Future Directions - Communication

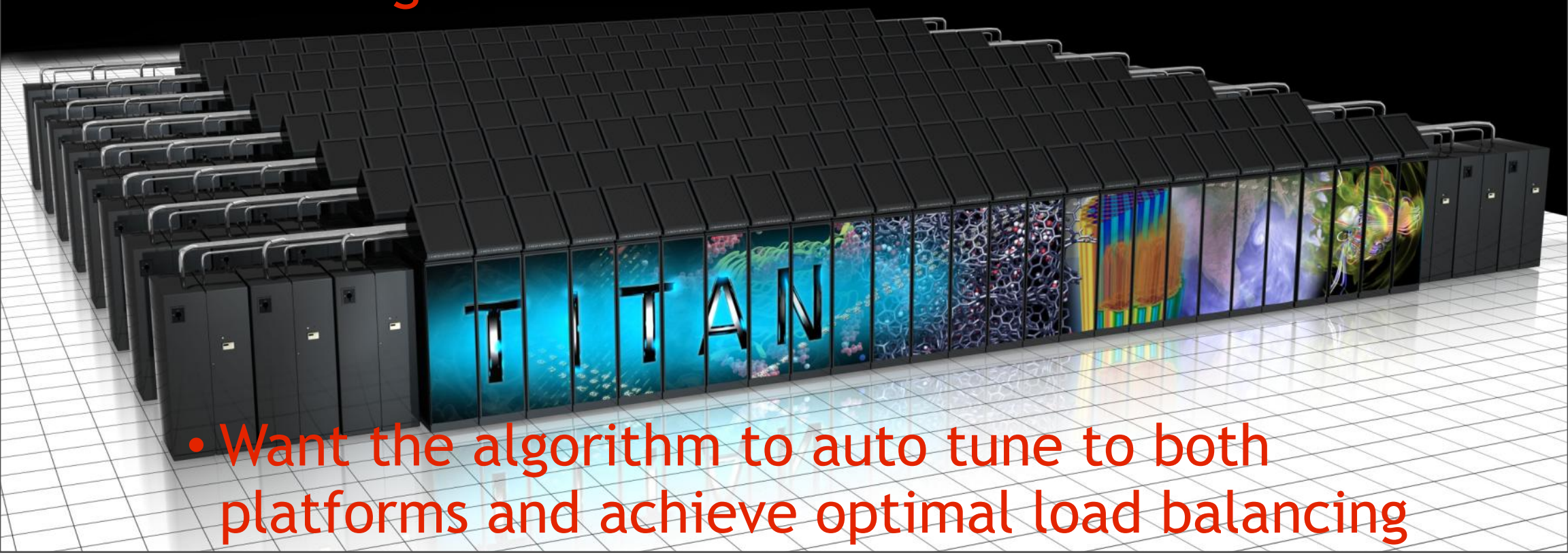
- Only scratched the surface of domain-decomposition algorithms
  - Disjoint additive
  - Overlapping additive
  - Alternating boundary conditions
  - Random boundary conditions
  - Multiplicative Schwarz
  - Precision truncation
  - Random Schwarz



Flexibility



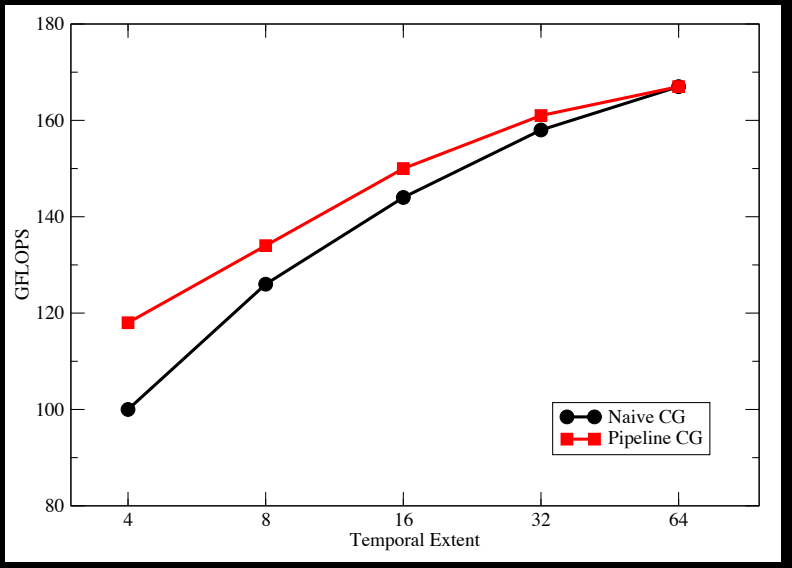
- There is lots of variation is what constitutes heterogenous



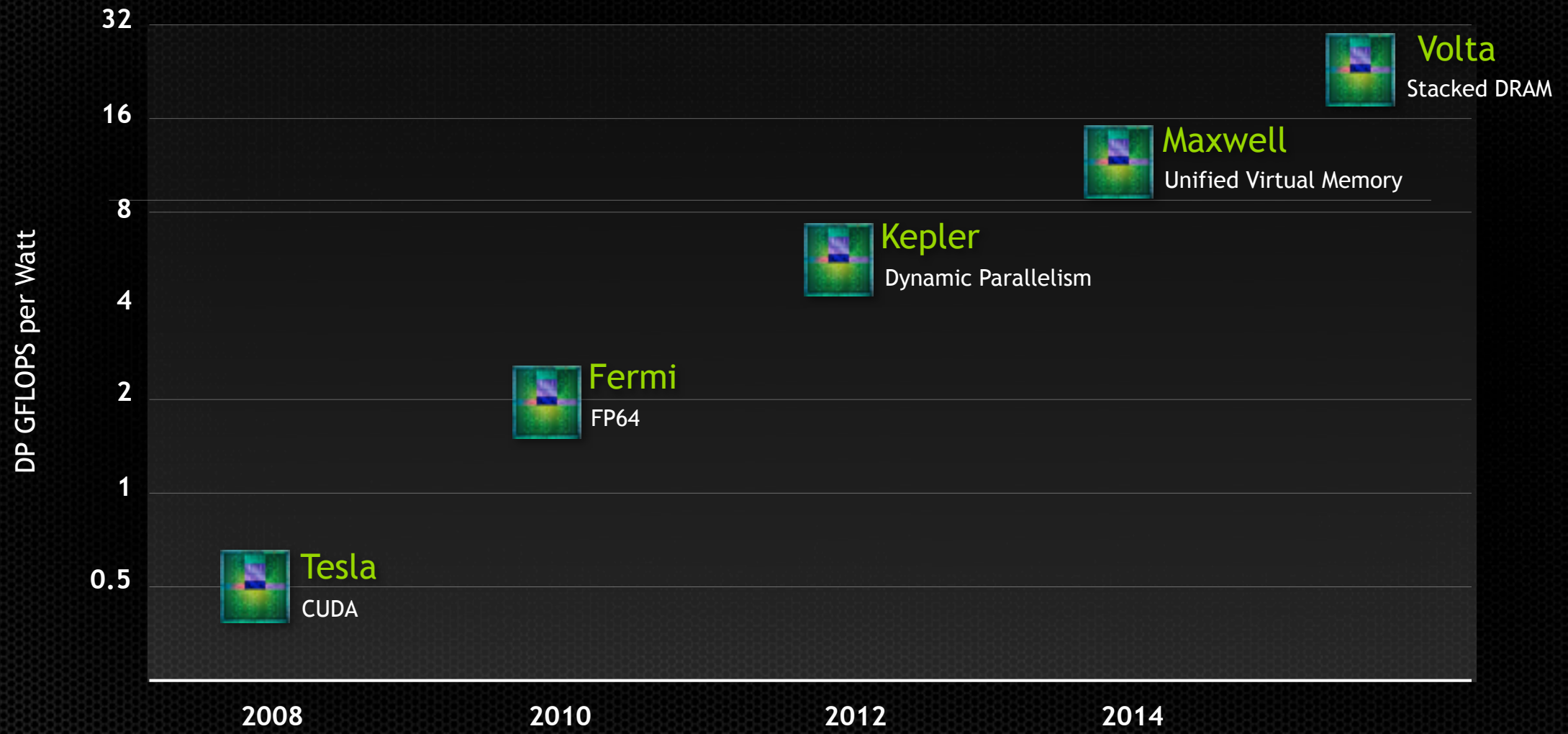
- Want the algorithm to auto tune to both platforms and achieve optimal load balancing

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



GPU Roadmap


DP GFLOPS per Watt
32
16
8
4
2
1
0.5
2008
2010
2012
2014
Tesla
CUDA
Fermi
FP64
Kepler
Dynamic Parallelism
Maxwell
Unified Virtual Memory
Volta
Stacked DRAM

Future Directions - Precision

- Mixed-precision methods have become de facto
  - Mixed-precision Krylov solvers
  - Low-precision preconditioners
- Exploit closer coupling of precision and algorithm
  - Domain decomposition, Adaptive Multigrid
  - Hierarchical-precision algorithms
  - 128-bit <-> 64-bit <-> 32-bit <-> 16-bit <-> 8-bit
- Low precision is lossy compression
- Low-precision tolerance is fault tolerance