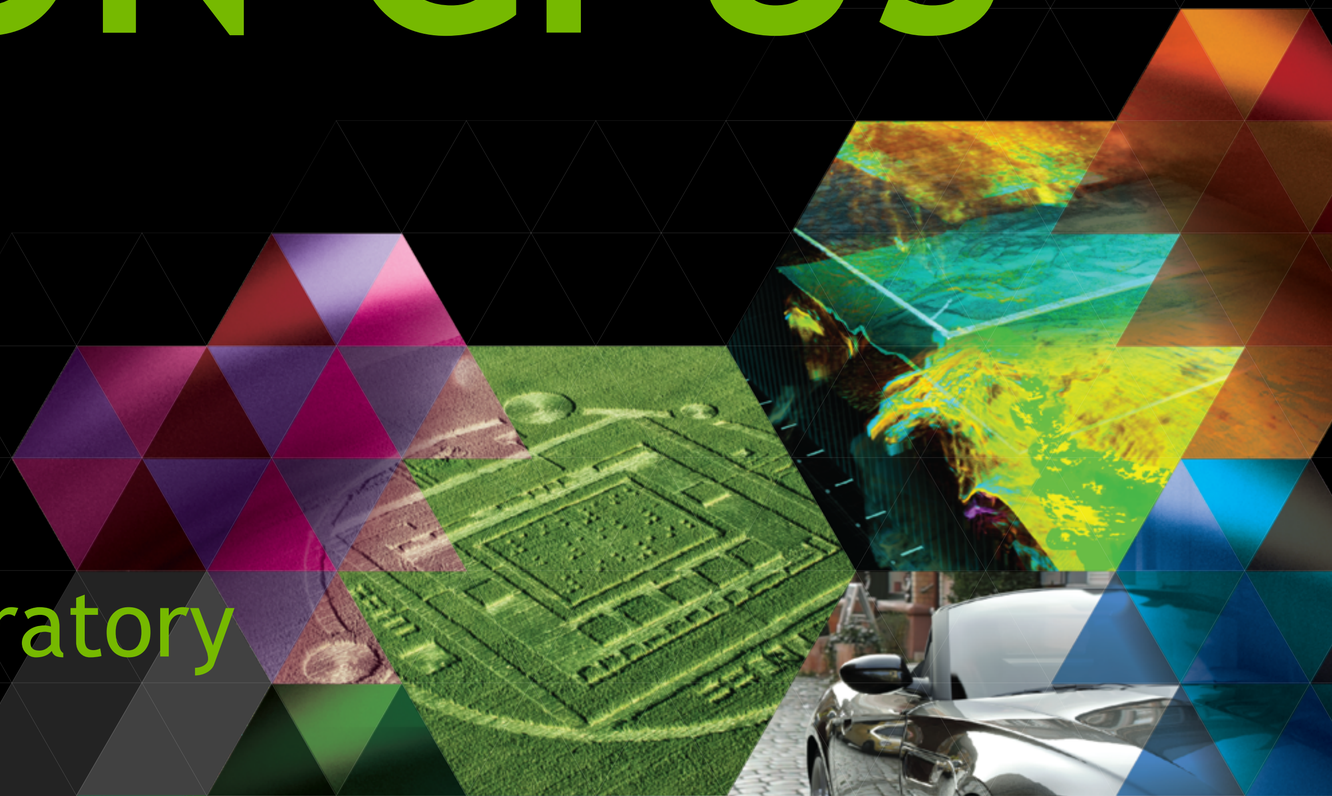
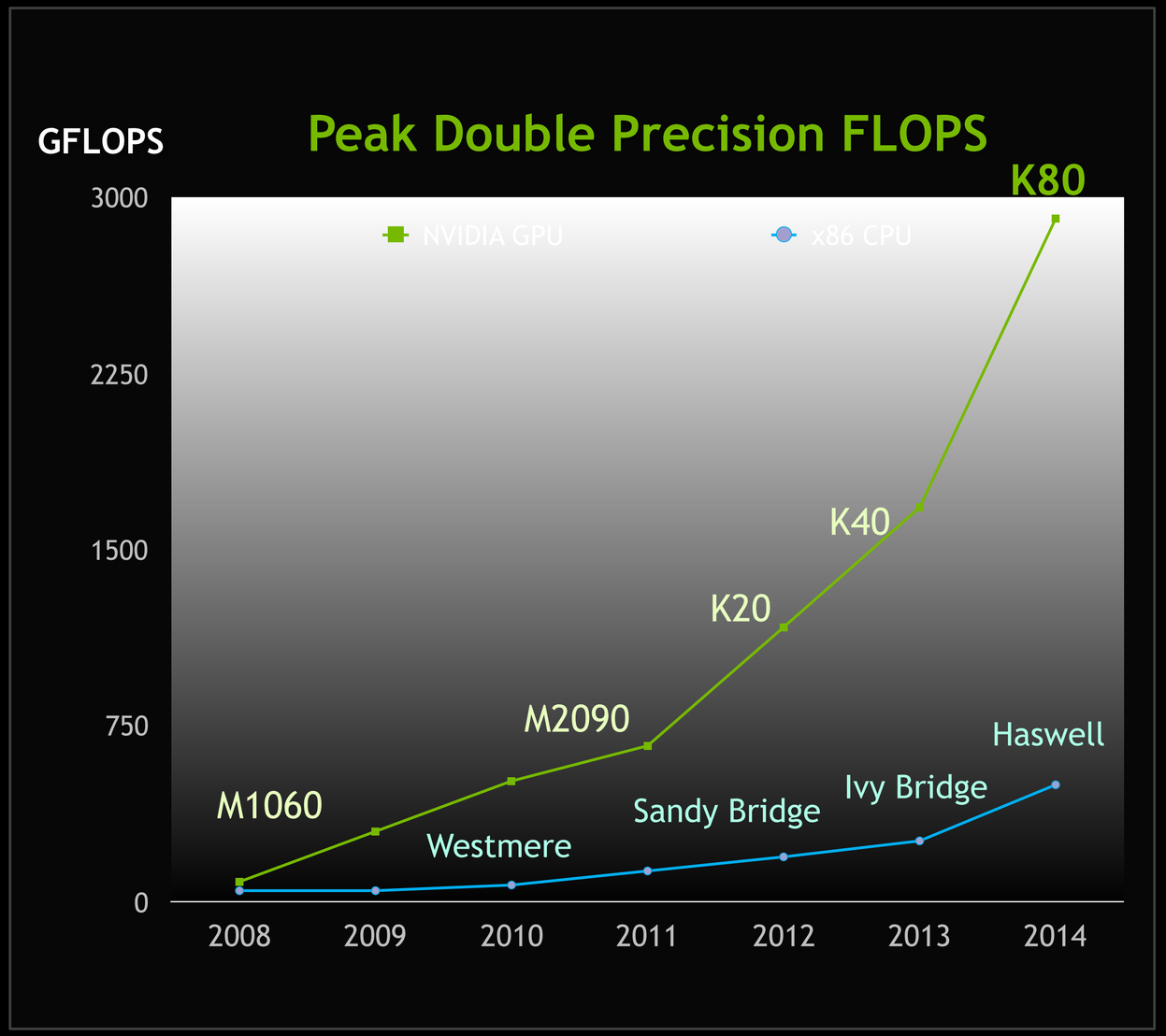
QUDA ON GPUS

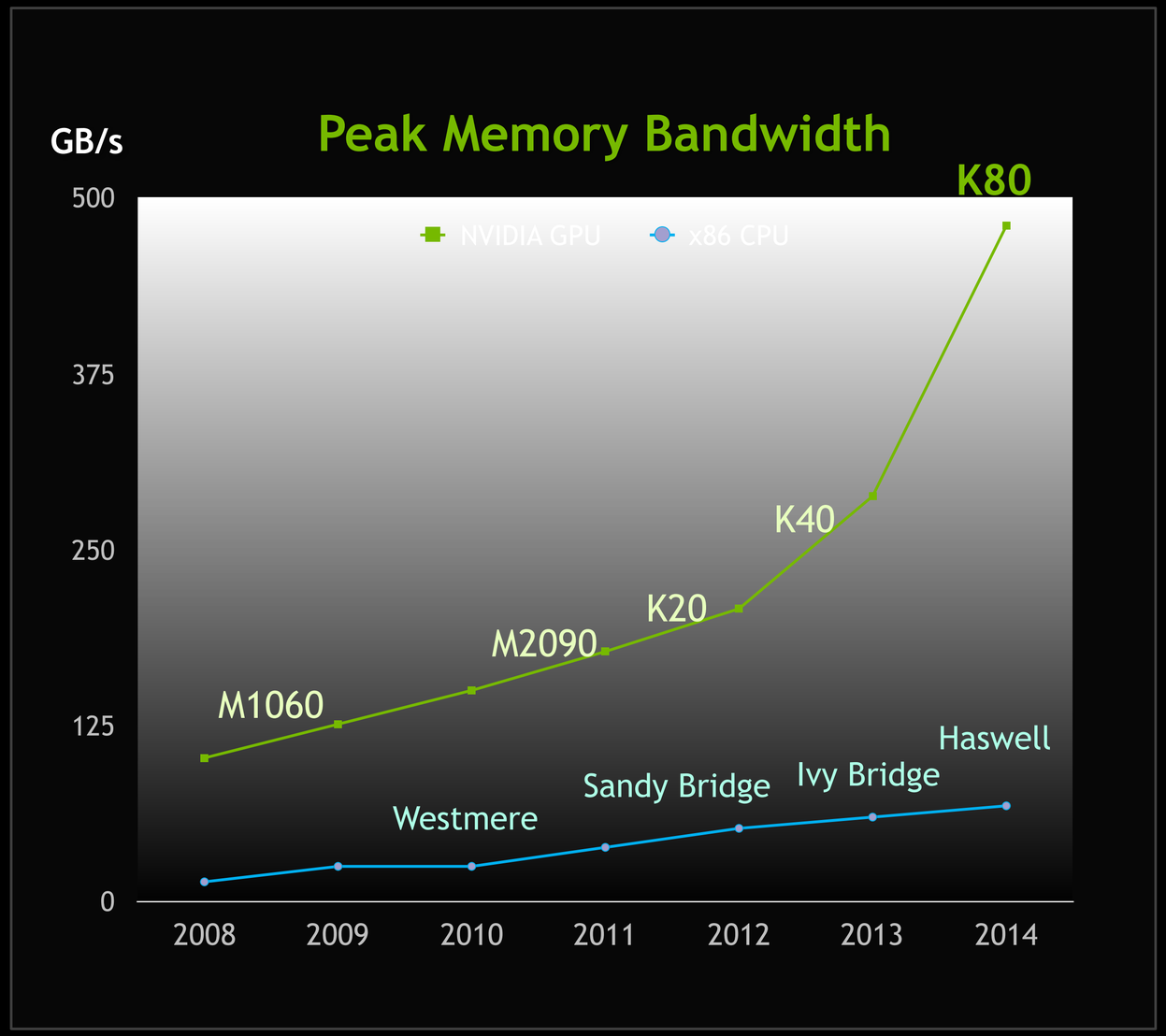


March 2016
Brookhaven National Laboratory

Contents

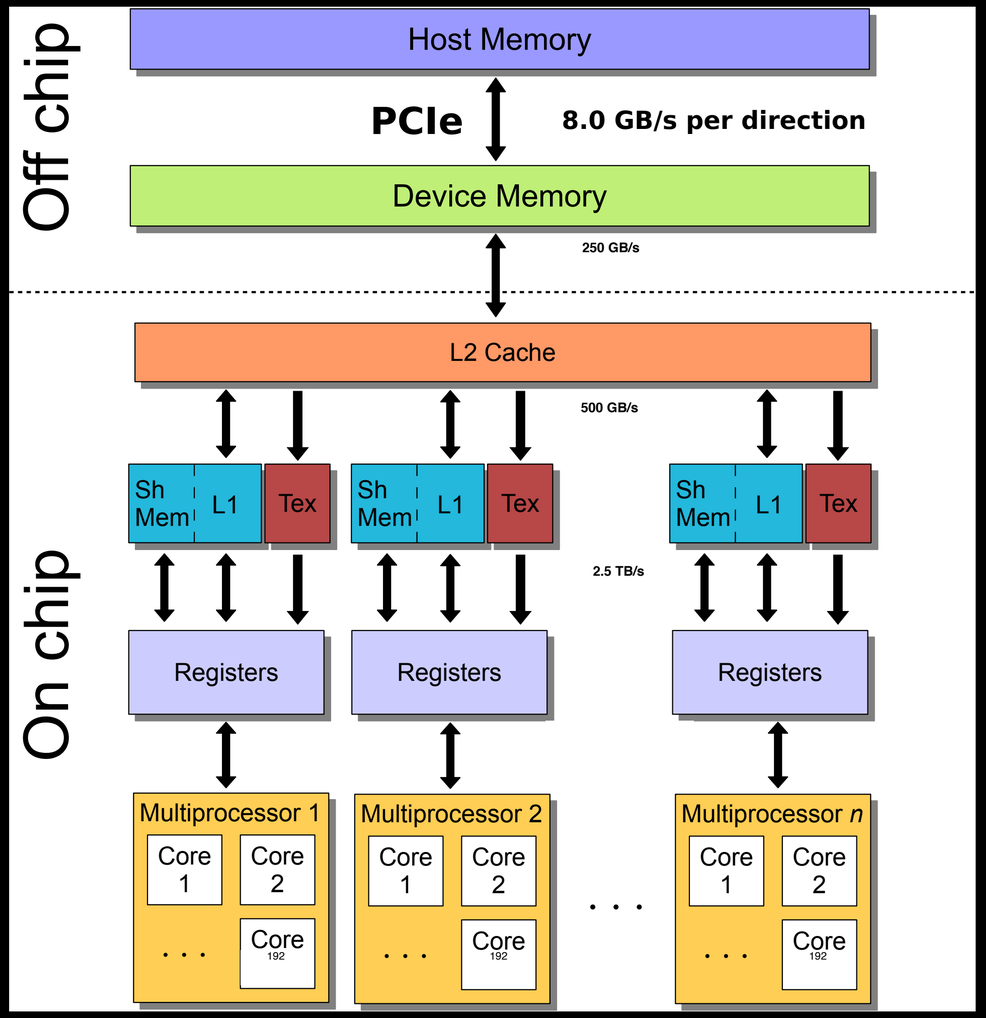
- GPUs
- QUDA
- Multigrid and Hierarchical Algorithms
- Exascale

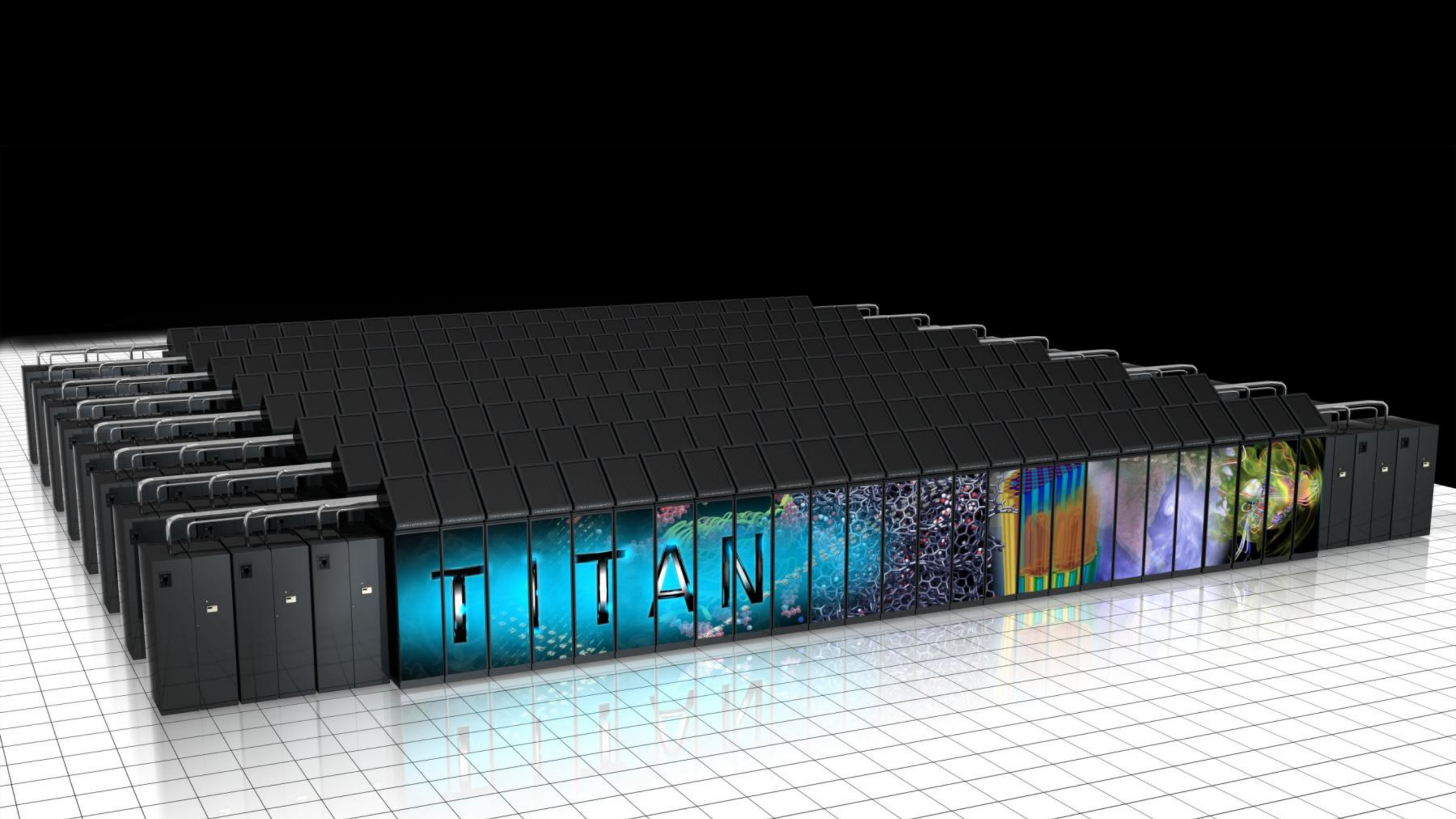

GFLOPS
Peak Double Precision FLOPS
3000
2250
1500
750
0
NVIDIA GPU
x86 CPU
K80
K40
K20
M2090
M1060
Haswell
Ivy Bridge
Sandy Bridge
Westmere
2008
2009
2010
2011
2012
2013
2014


GB/s
Peak Memory Bandwidth
500
375
250
125
0
NVIDIA GPU
x86 CPU
K80
K40
K20
M2090
M1060
Haswell
Ivy Bridge
Sandy Bridge
Westmere
2008
2009
2010
2011
2012
2013
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

QCD on GPUs

QUDA

- "QCD on CUDA" – http://lattice.github.com/quda (open source)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, **MILC**, TIFR, tmQCD, etc.
  - Latest release 0.8.0 (8th February 2016)
- Provides:
  - Various solvers for all major fermionic discretizations, with multi-GPU support
  - Additional performance-critical routines needed for gauge-field generation
  - Gauge fixing, pure gauge evolution, link smearing, etc.
- Maximize performance
  - 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 and deflated solvers (Lanczos, EigCG, GMRES-DR)
  - Multigrid solvers for optimal convergence
- A research tool for how to reach the exascale

In the QUDA Pipeline

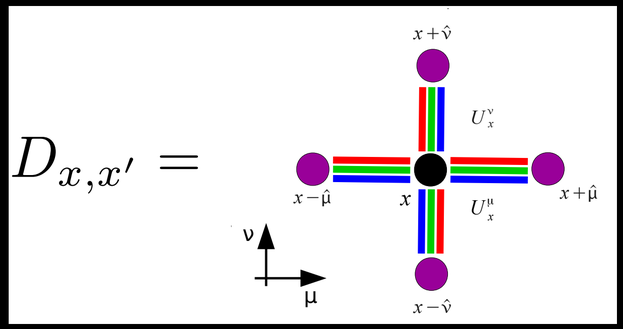
- Dramatically improved 4-d domain-wall performance
- Drastically improved strong scaling
  - Intra node - direct communication between GPUs (no MPI)
  - GPU Direct Async - NIC and GPU can synchronize with no CPU
- Improved deflation algorithms
- Stout smearing
- Multi-right-hand-side solvers
- Communication avoiding solvers (s-step)
- Improved reduction support (including quad-precision)

QUDA collaborators

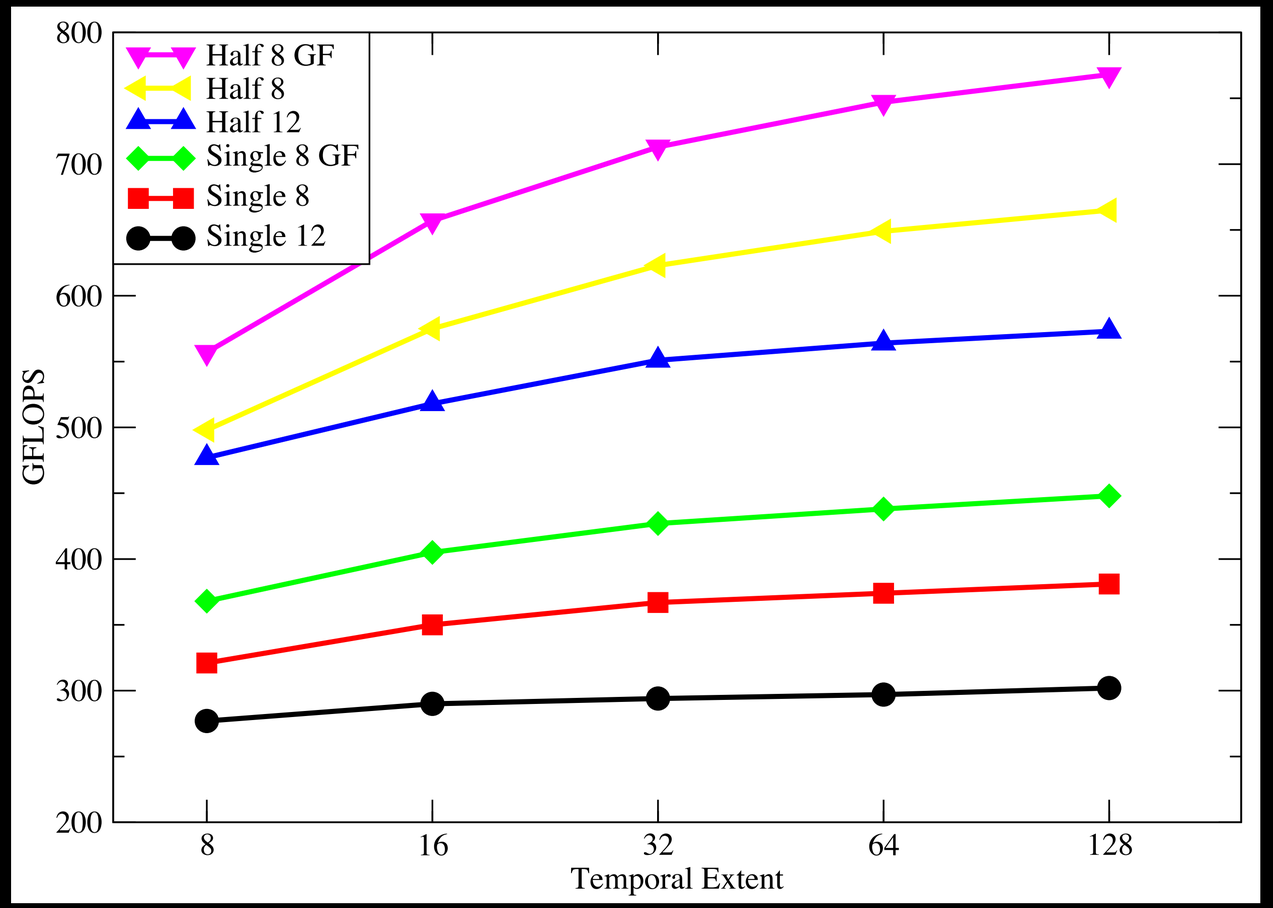
- Ron Babich (NVIDIA)
- Michael Baldhauf (Regensburg)
- Kip Barros (LANL)
- Rich Brower (Boston University)
- Nuno Cardoso (NCSA)
- Michael Cheng (Boston University)
- Carleton DeTar (Utah University)
- Justin Foley (Utah -> NIH)
- Joel Giedt (Rensselaer Polytechnic Institute)
- Steve Gottlieb (Indiana University)
- Bálint Joó (Jlab)
- Hyung-Jin Kim (BNL -> Samsung)
- Claudio Rebbi (Boston University)
- Guochun Shi (NCSA -> Google)
- Mario Schröck (INFN)
- Alexei Strelchenko (FNAL)
- Alejandro Vaquero (INFN)
- Mathias Wagner (NVIDIA)
- Frank Winter (Jlab)

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)



Kepler Wilson-Dslash Performance



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

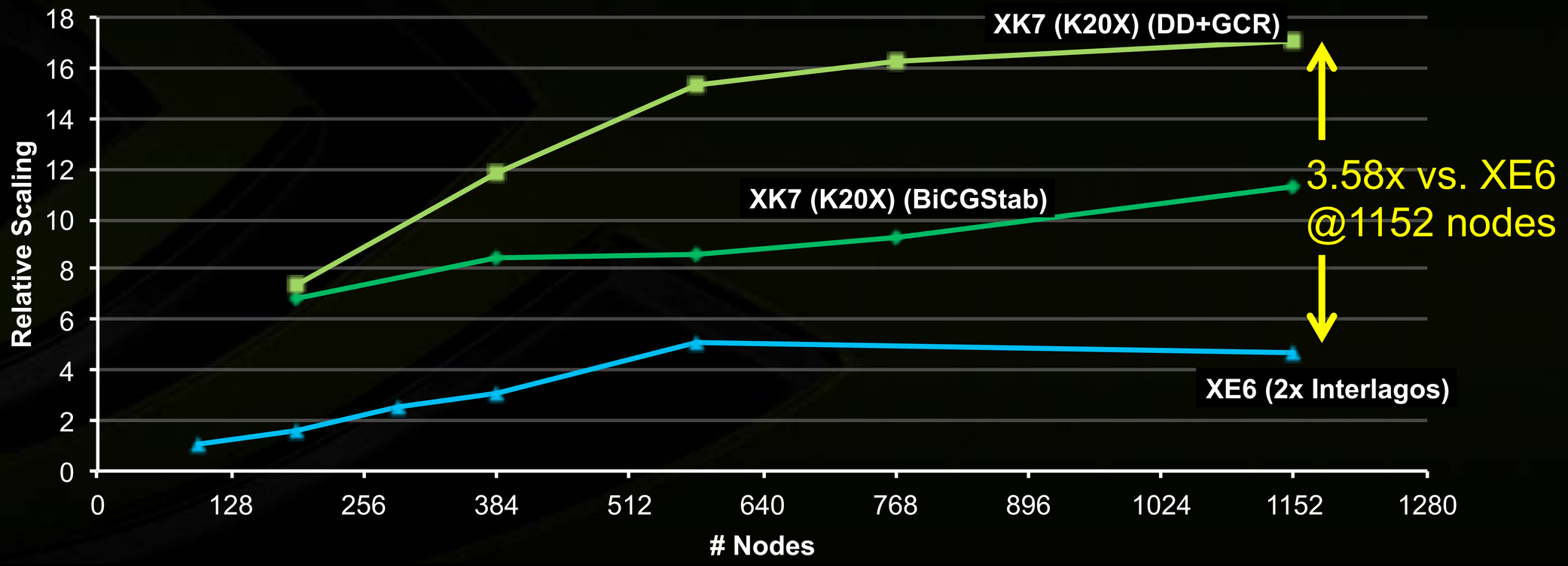
Strong Scaling Chroma with DD

Chroma

48^3x512 lattice

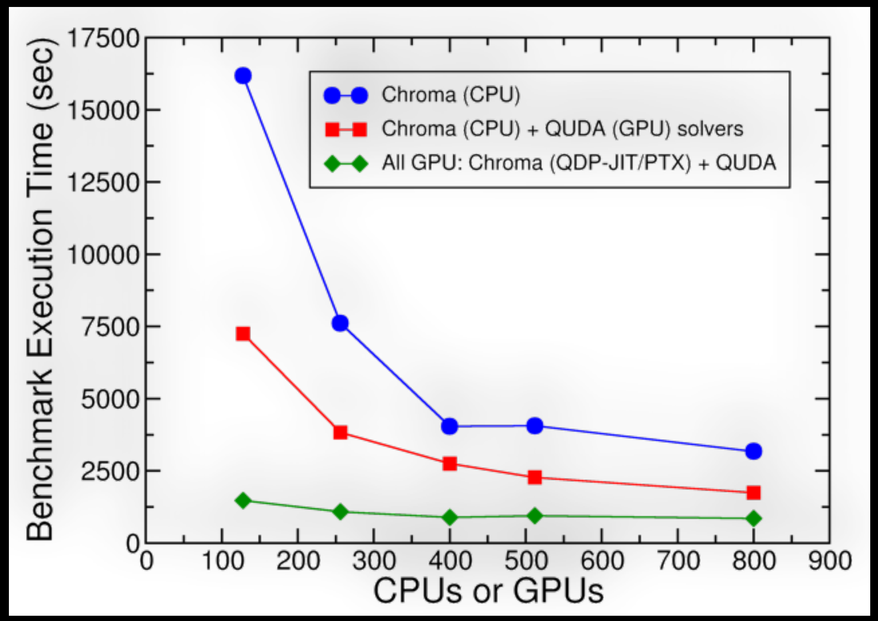
Relative Scaling (Application Time)

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



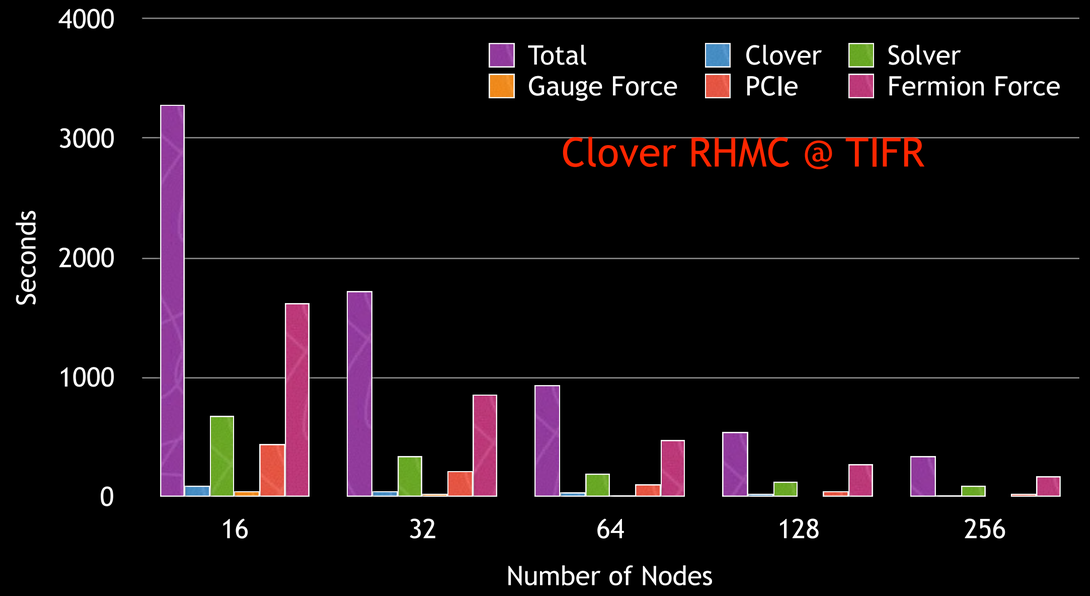
QCD on GPUs

Wide deployment on GPU clusters



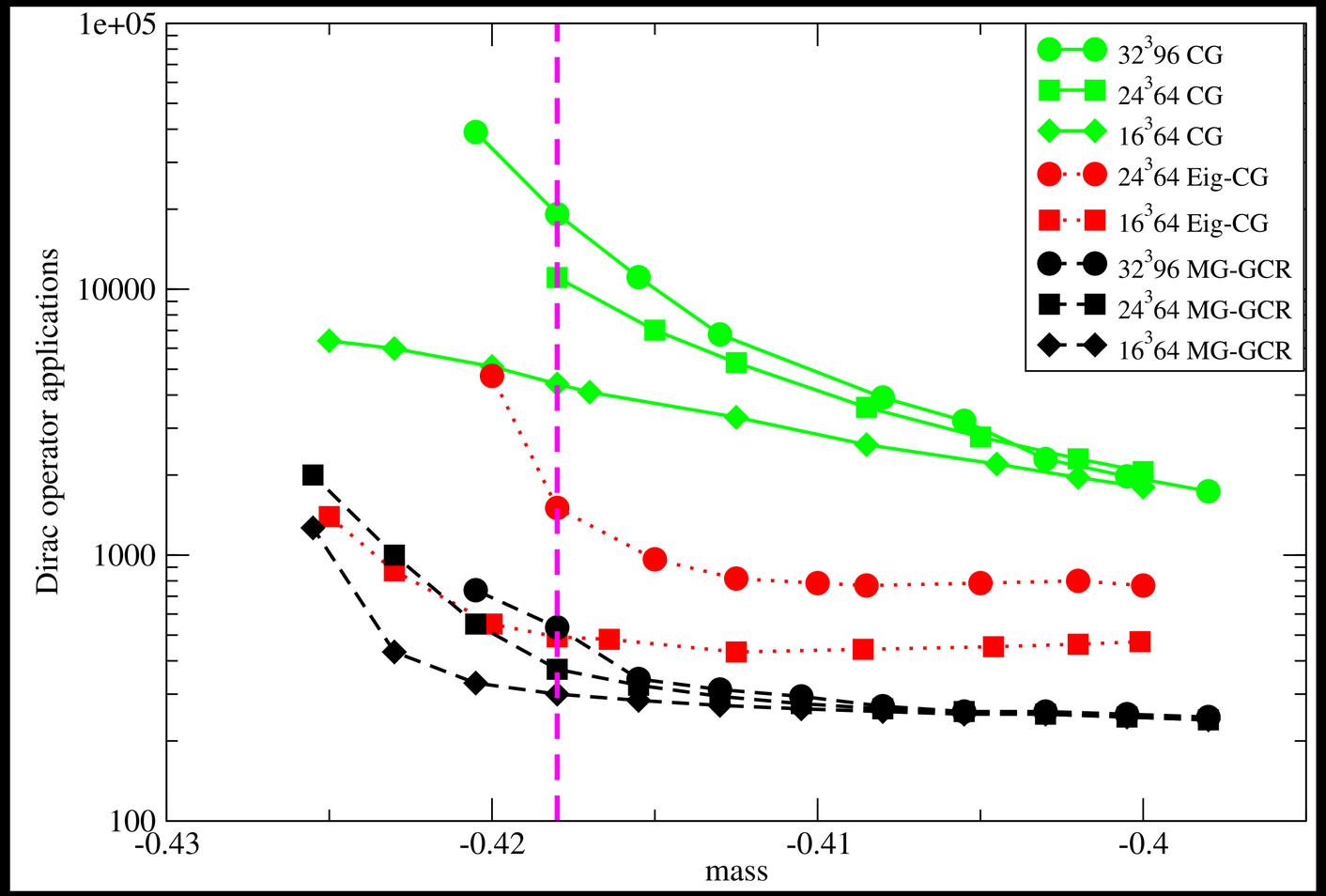
- Clover RHMC on Titan and Blue Waters (Chroma)
- Staggered RHMC @ FNAL (MILC)
- Nuclear physics @ Jlab (Chroma)

- Clover RHMC running @ TIFR (MILC)
- Thermodynamics @ TIFR
- Twisted-mass @ INFN
- etc.



MULTIGRID AND HIERARCHICAL ALGORITHMS

Why Multigrid?



240 vectors

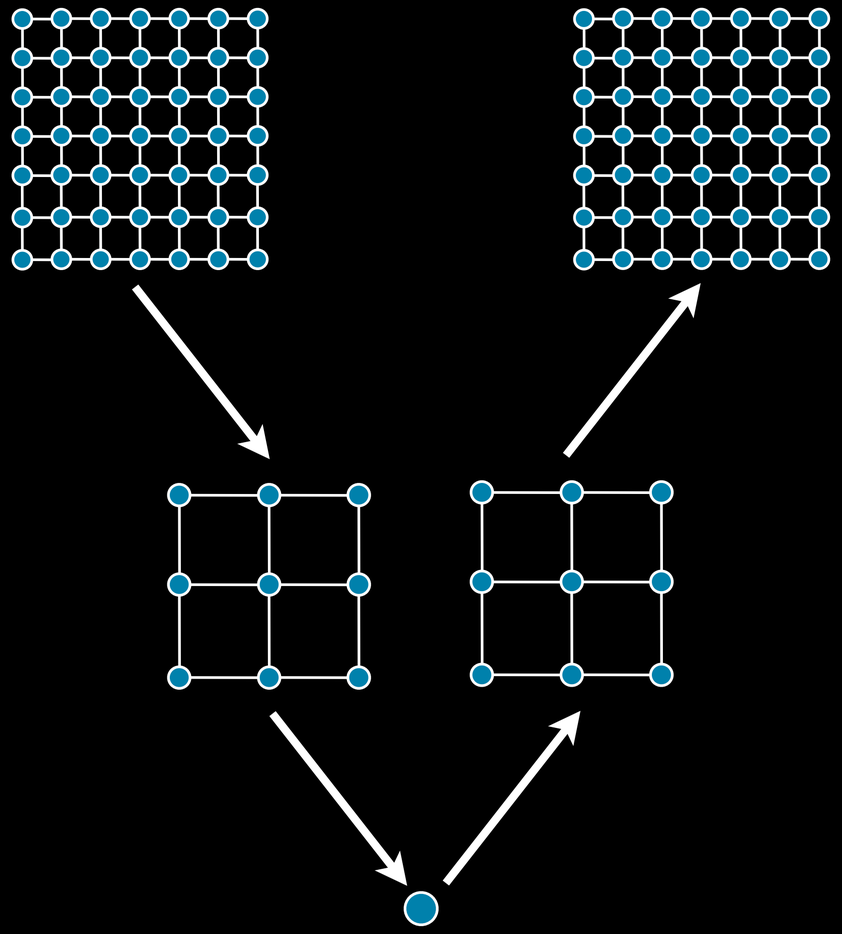
20 vectors

Babich *et al* 2010

Hierarchical algorithms for LQCD

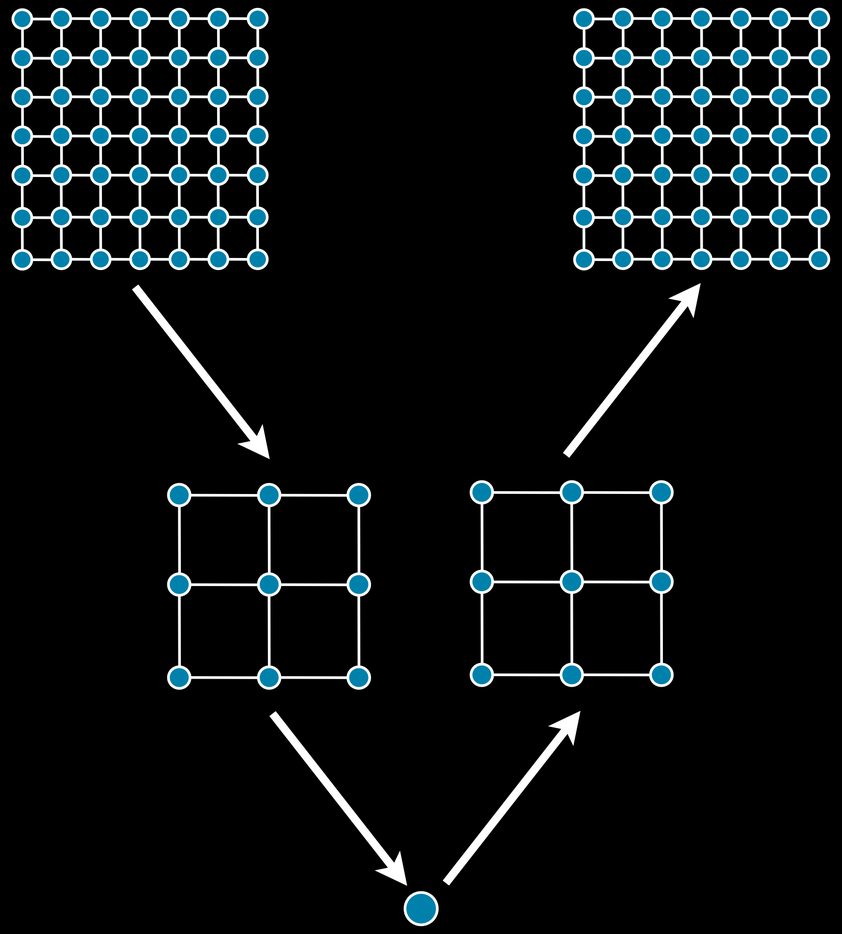
- Hierarchical algorithms have revolutionized LQCD computation
- Adaptive Geometric Multigrid for LQCD
  - Based on adaptive smooth aggregation (Brezina *et al* 2004)
  - Low modes have weak-approximation property => locally co-linear
  - Apply fixed geometric coarsening (Brannick *et al* 2007, Babich *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

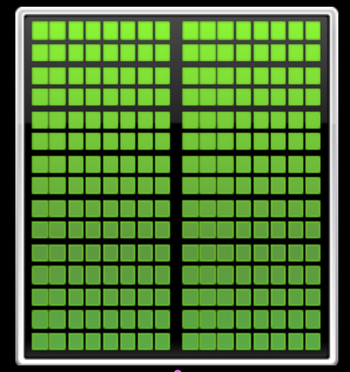
The Challenge of Multigrid on GPU



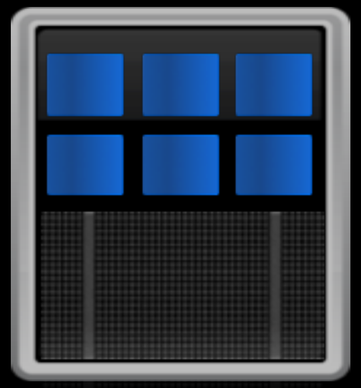
- GPU requirements very different from CPU
  - Each thread is slow, but O(10,000) threads per GPU
- Fine grids run very efficiently
  - High parallel throughput problem
- Coarse grids are worst possible scenario
  - More cores than degrees of freedom
  - Increasingly serial and latency bound
  - Little's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

Hierarchical algorithms on heterogeneous architectures





PCIe



GPU

Thousands of cores for parallel processing

CPU

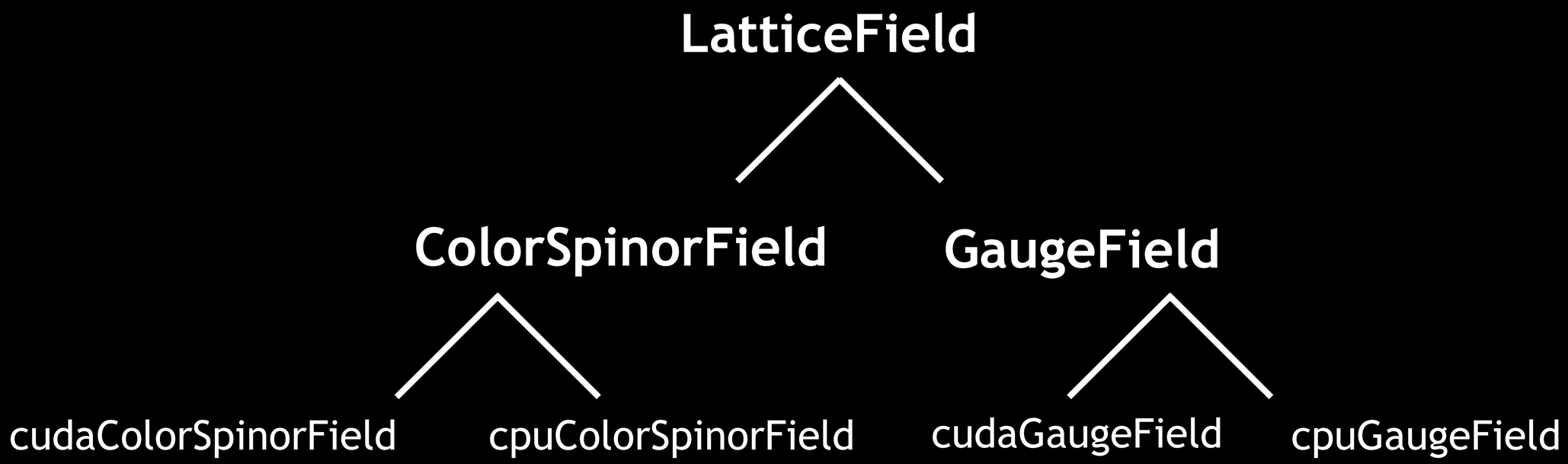
Few Cores optimized for serial work

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*
- (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

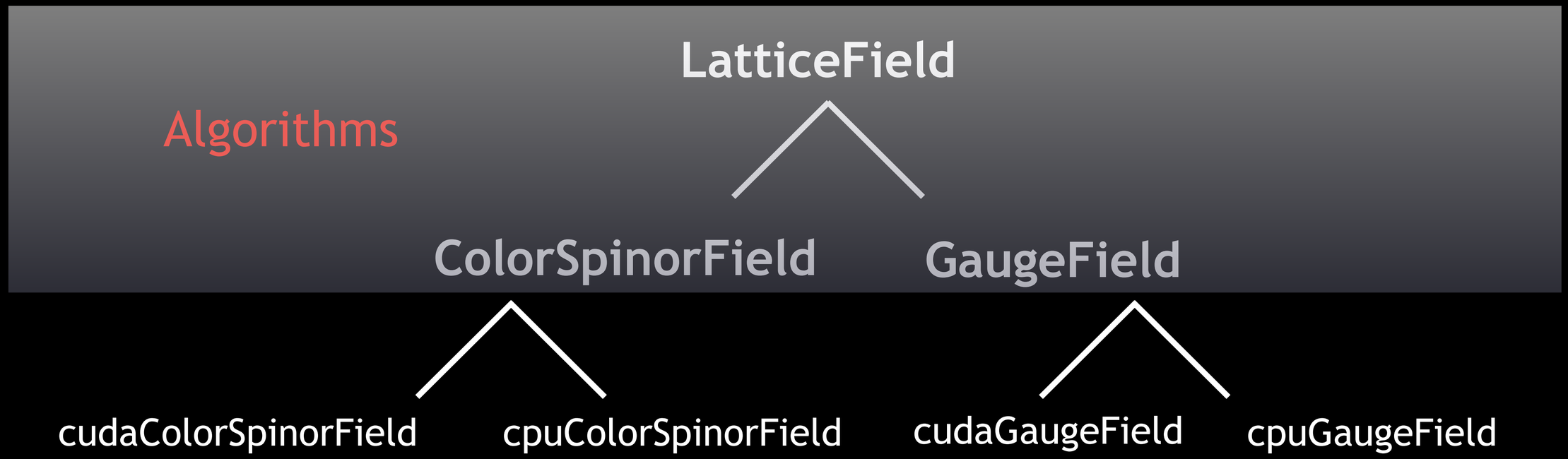
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



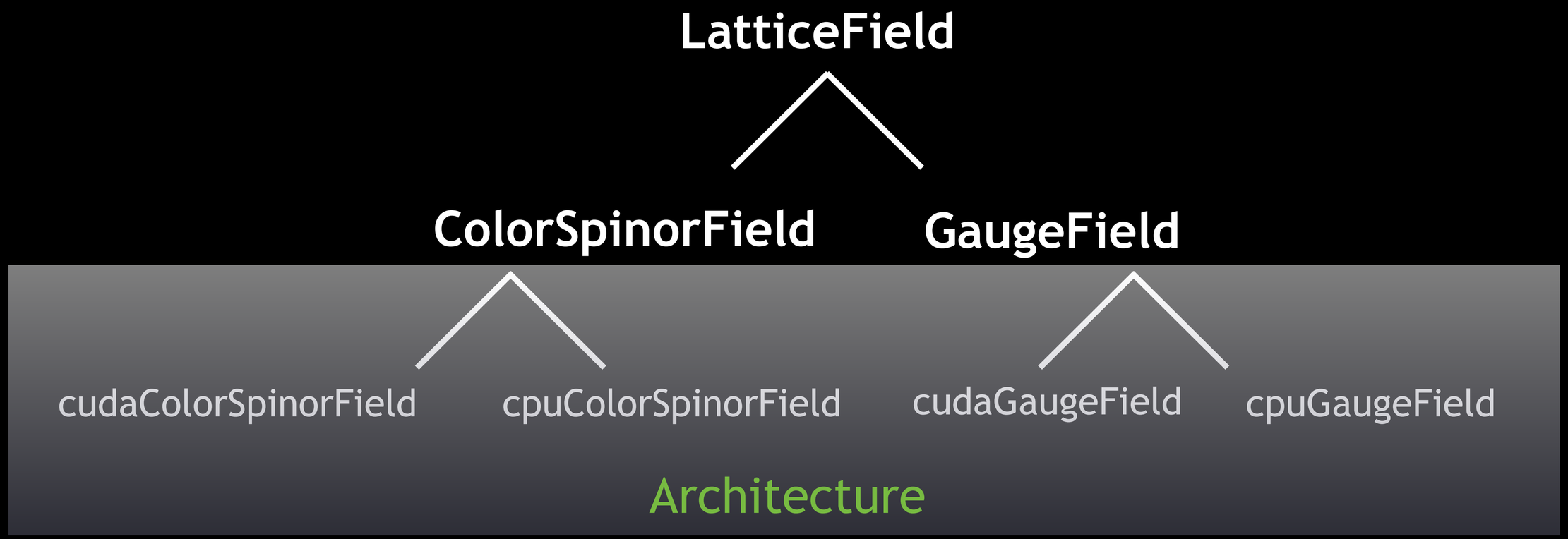
Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



Multigrid and QUDA

- QUDA designed to abstract algorithm from the heterogeneity



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 hidden 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

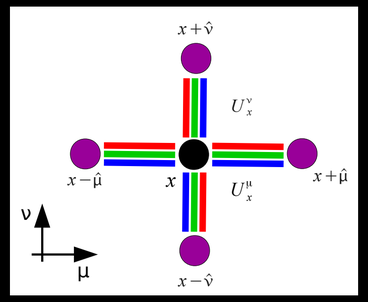
```
template<…> __global__ void fooGPU(Arg arg) {
  int tid = threadIdx.x + blockIdx.x*blockDim.x;
  real sum = bar<…>(arg, tid);
  __shared__ typename BlockReduce::TempStorage tmp;
  arg.sum = cub::BlockReduce<…>(tmp).Sum(sum);
}
```

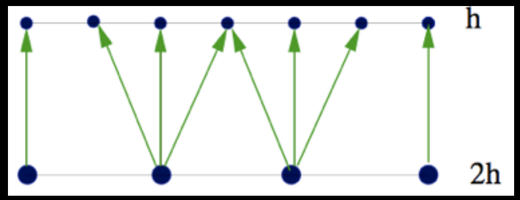
CPU

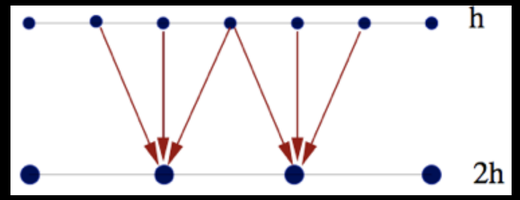
GPU

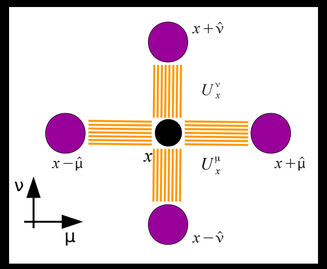
Ingredients for Parallel Adaptive Multigrid

- Prolongation construction (setup)
  - Block orthogonalization of null space vectors
  - Batched QR decomposition
- Smoothing (relaxation on a given grid)
  - Repurpose existing solvers (DD 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

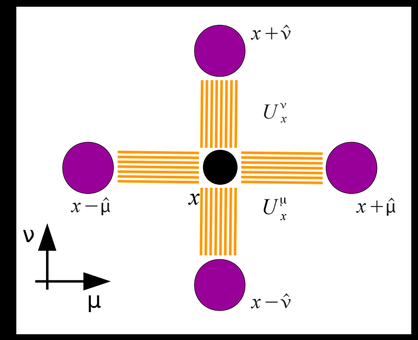








Coarse Grid Operator Analysis



- Coarse operator looks like a Dirac operator
  - Link matrices have dimension $2N_v$ x $2N_v$ (e.g., 48 x 48)

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

- Fine vs. Coarse grid parallelization
  - Fine grid operator has plenty of grid-level parallelism
    - E.g., 16^4 = 65536 lattice sites
  - Coarse grid operator has diminishing grid-level parallelism
    - first coarse grid 4^4 = 256 lattice sites
    - second coarse grid 2^4 = 16 lattice sites
- Current GPUs have up to 3072 processing cores
- Need to consider finer-grained parallelization
  - Increase parallelism to use all GPU resources
  - Load balancing

Sources of Parallelism

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

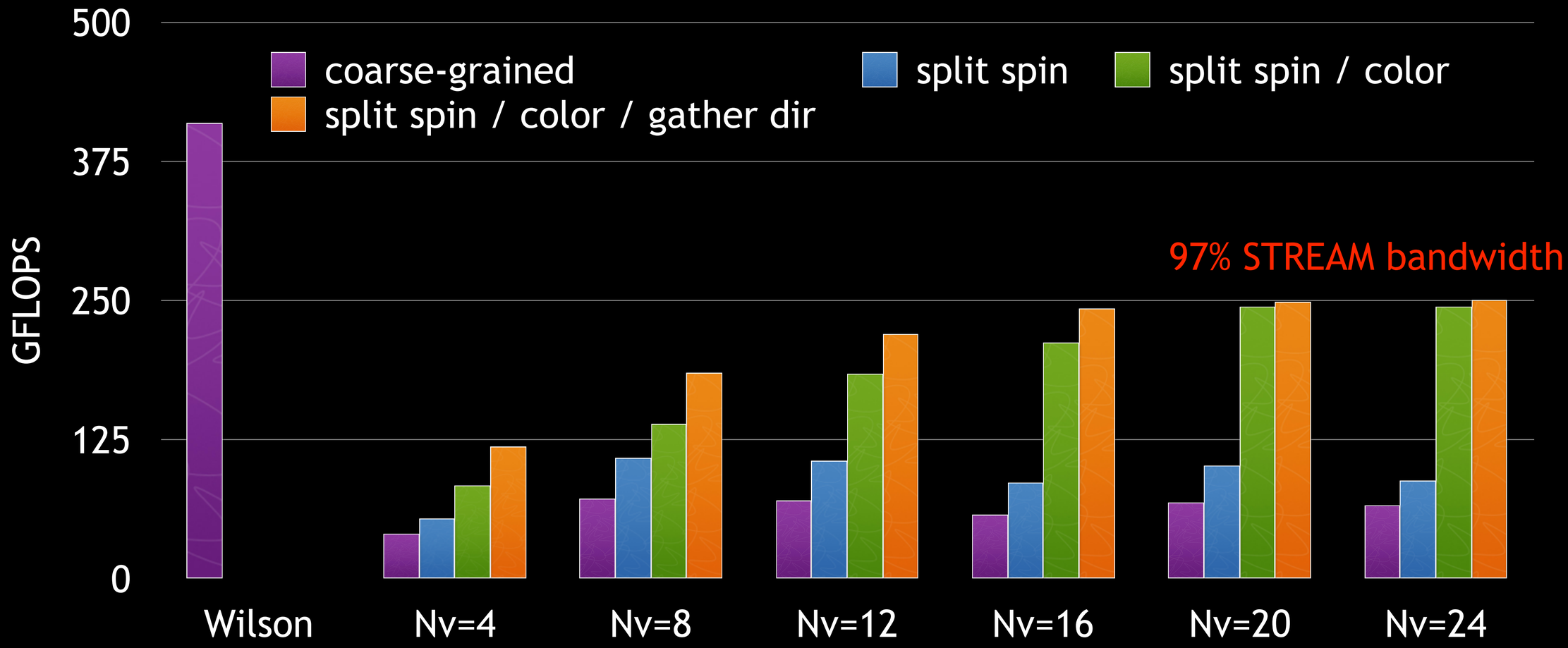
- Site-level parallelism (i index)
  - Trivially data parallel
- Spin and color output index (s and c indices)
  - Trivially data parallel
- Stencil direction (μ index)
  - Loosely coupled parallelism (gather)
- Spin and color input index (s' and c’ indices)
  - Tightly coupled parallelism (dot product)

2^4 => 24576-way parallel

N_{vec}=24

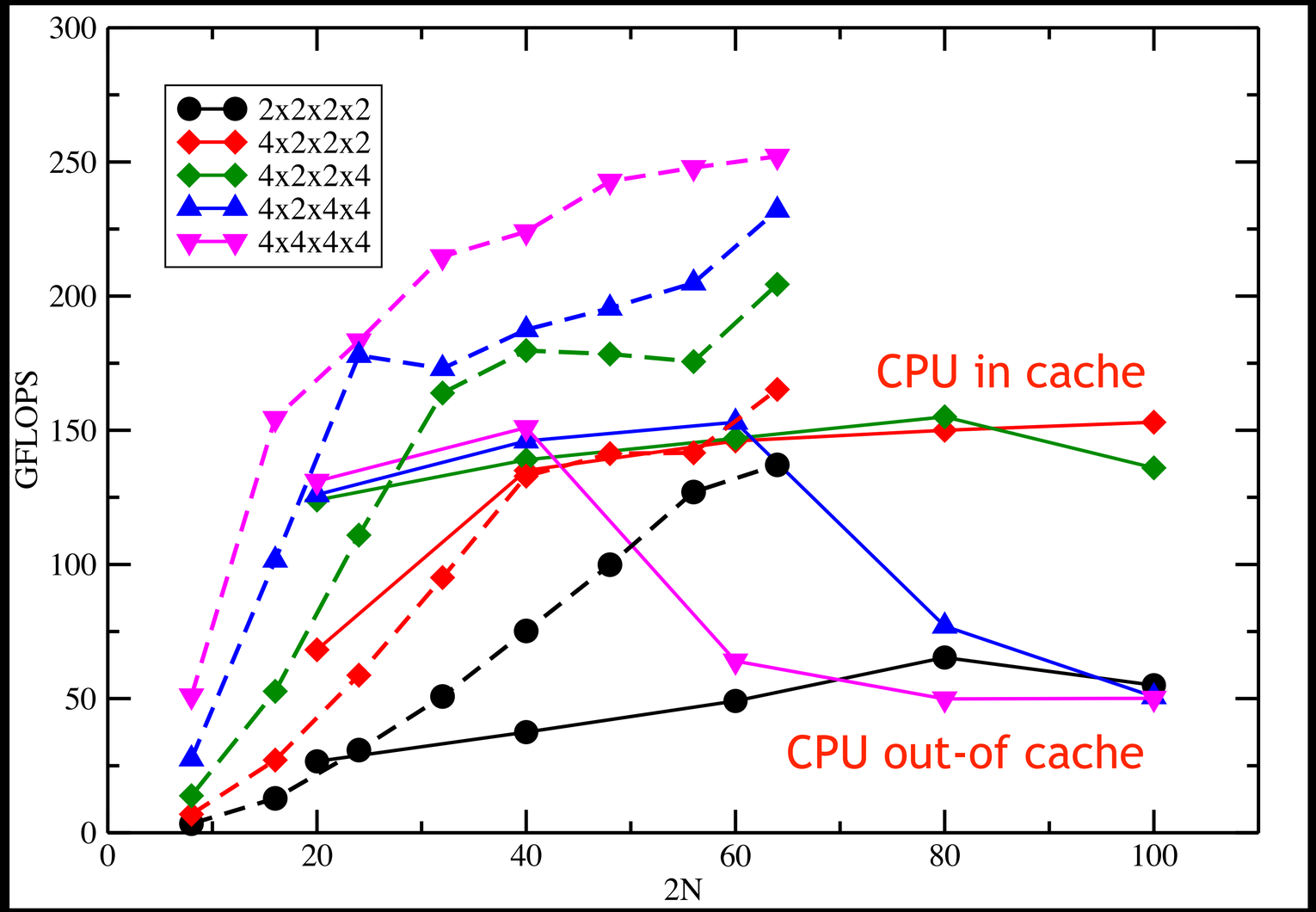
Coarse Grid Operator Performance

$V_{fine} = 16^3 \times 64$, $V_{coarse} = 4^3 \times 16$, FP32, no reconstruction, Quadro M6000



Coarse Dslash Performance

8-core Haswell 2.4 GHz (solid line) vs M6000 (dashed lined)



- Autotuner finds optimum degree of parallelization
  - Larger grids favor less fine grained
  - Coarse grids favor most fine grained
- GPU is nearly always faster than CPU
- Expect in future that coarse grids will favor CPUs
- For now, use GPU exclusively

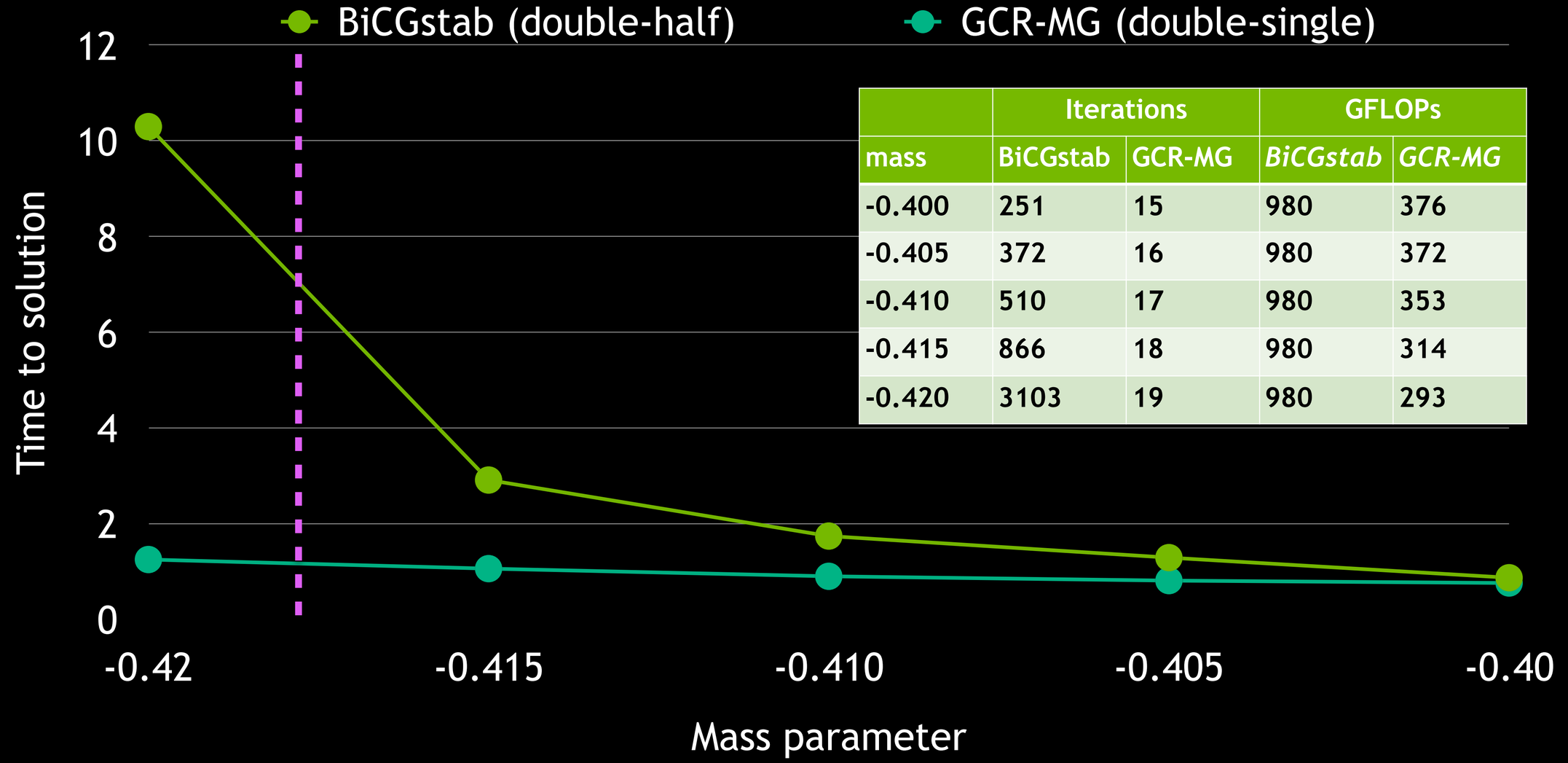
Results

- Compare MG against the best traditional clover Krylov solver
  - BiCGstab in double/half precision
  - 12/8 reconstruct
  - Even-odd preconditioning

- Adaptive Multigrid algorithm
  - GCR outer solver wraps 3-level MG preconditioner
  - GCR restarts done in double, everything else in single
  - 24 or 32 null-space vectors on fine grid
  - Minimum Residual smoother
  - Even-odd preconditioning

Multigrid versus BiCGstab

Anisotropic Wilson, V = 24^3x64, 3x Quadro M6000



	Iterations		GFLOPs	
mass	BiCGstab	GCR-MG	*BiCGstab*	*GCR-MG*
-0.400	251	15	980	376
-0.405	372	16	980	372
-0.410	510	17	980	353
-0.415	866	18	980	314
-0.420	3103	19	980	293

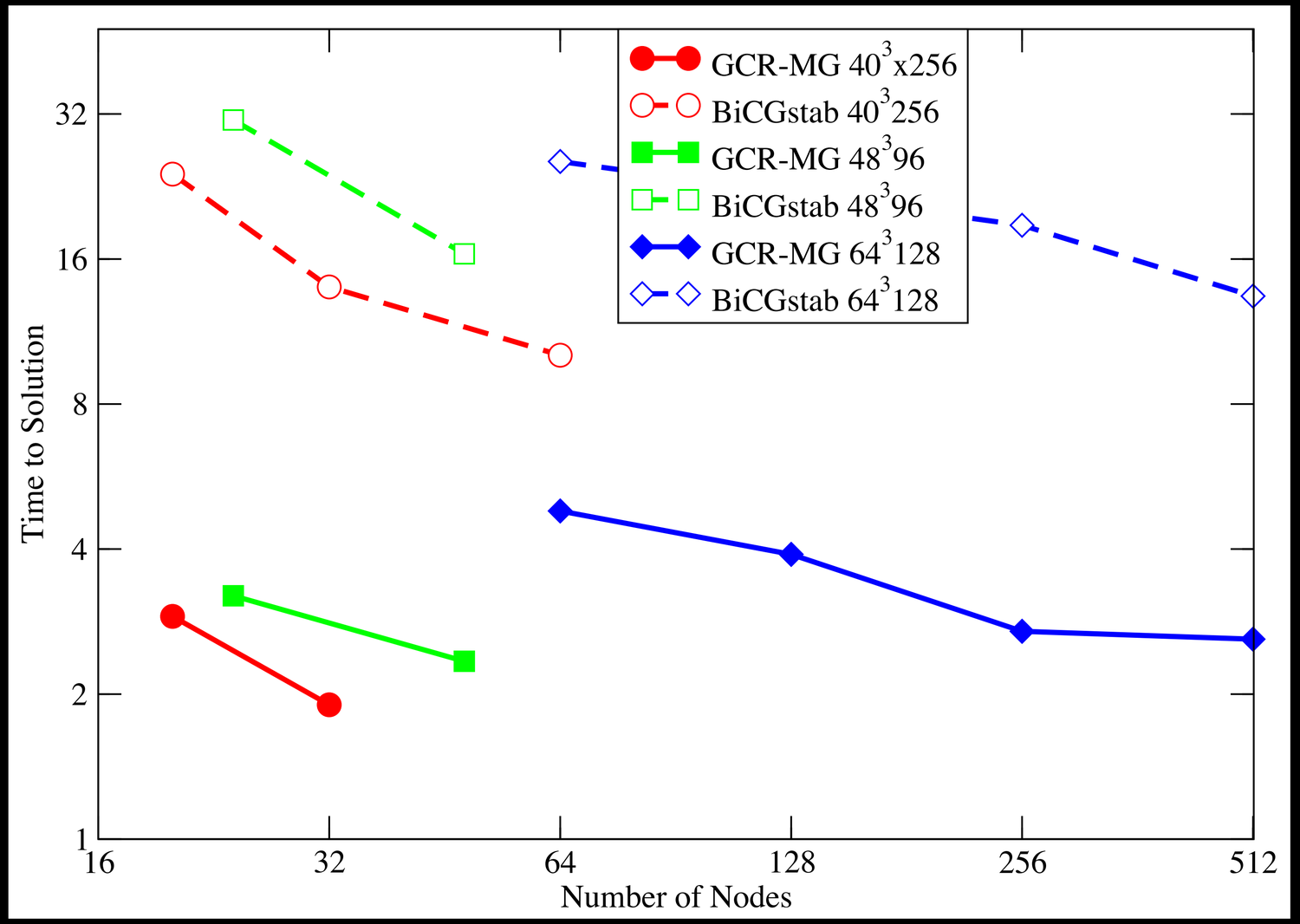
Multigrid vs BiCGstab

Anisotropic Clover, V = 40^3x256, m_π = 230 MeV, light quark, 32 nodes of Titan

QUDA MG		*QUDA BiCGStab*		*Speedup*
Iterations	*Time (sec)*	*Iterations*	*Time (sec)*	
18	*1.90*	*1811*	*17.98*	*9.46*
18	*1.89*	*1582*	*15.63*	*8.27*
18	*1.85*	*1613*	*16.02*	*8.66*
19	*2.02*	*1644*	*16.29*	*8.06*
19	*2.01*	*1788*	*17.67*	*8.79*
18	*1.90*	*1940*	*19.19*	*10.10*
18	*1.89*	*1568*	*15.58*	*8.24*
19	*2.00*	*1698*	*16.88*	*8.44*
19	*2.03*	*1914*	*18.98*	*9.35*
18	*1.85*	*1589*	*15.76*	*8.52*
18	*1.91*	*1735*	*17.16*	*8.98*
Average	*1.93*		*17.01*	*8.81*

Titan Scaling

Clover, 40^3x256 210 m_π, 48^3x96 / 64^3x128 192 MeV m_π



Multigrid Summary and Future Work

- Up to 10x speedups observed with multigrid
- Exploiting fine-grained parallelism was key
- Coarse solve dominates at large node count

- Lots more work to do
  - Strong scaling improvements
  - Absolute Performance tuning, e.g., half precision
  - Accelerate coarse grid solver: deflation or direct solve?
  - More flexible coarse grid distribution, e.g., redundant nodes

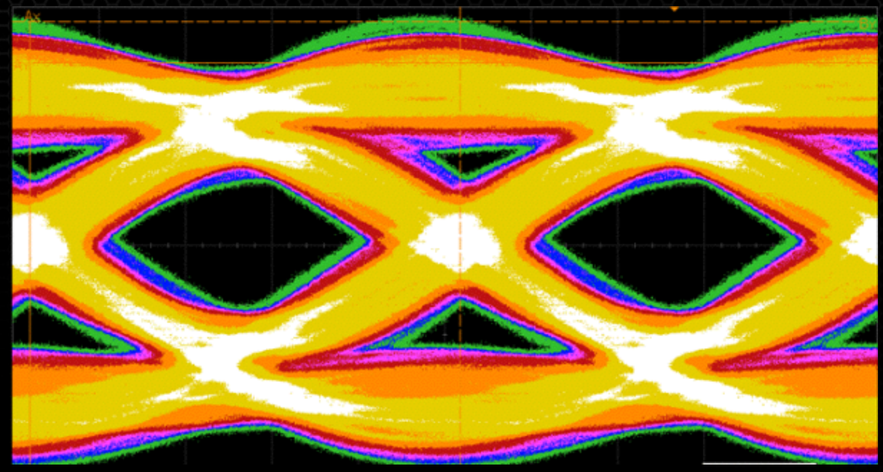
- Investigate off load coarse grids to the CPU
  - Use CPU and GPU simultaneously using additive MG
  - Likely optimal in the strong scaling limit

THE FUTURE

Introducing NVLINK and Stacked Memory

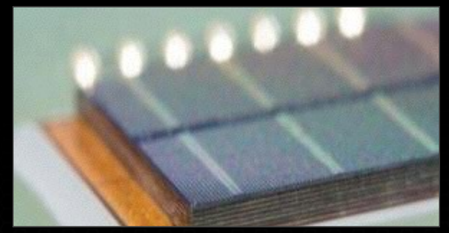
NVLINK

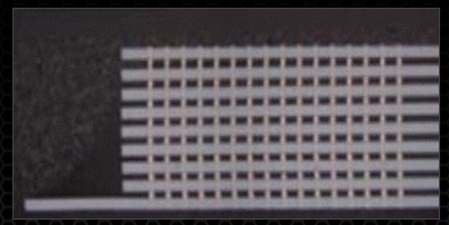
- GPU high speed interconnect
- 80-200 GB/s
- Planned support for POWER CPUs

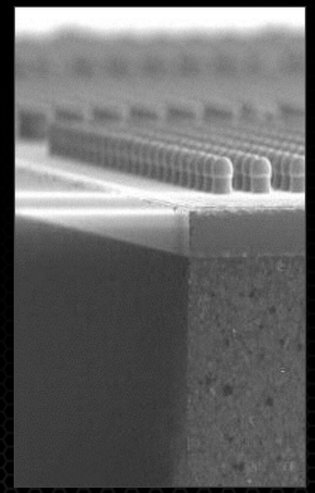


Stacked Memory

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

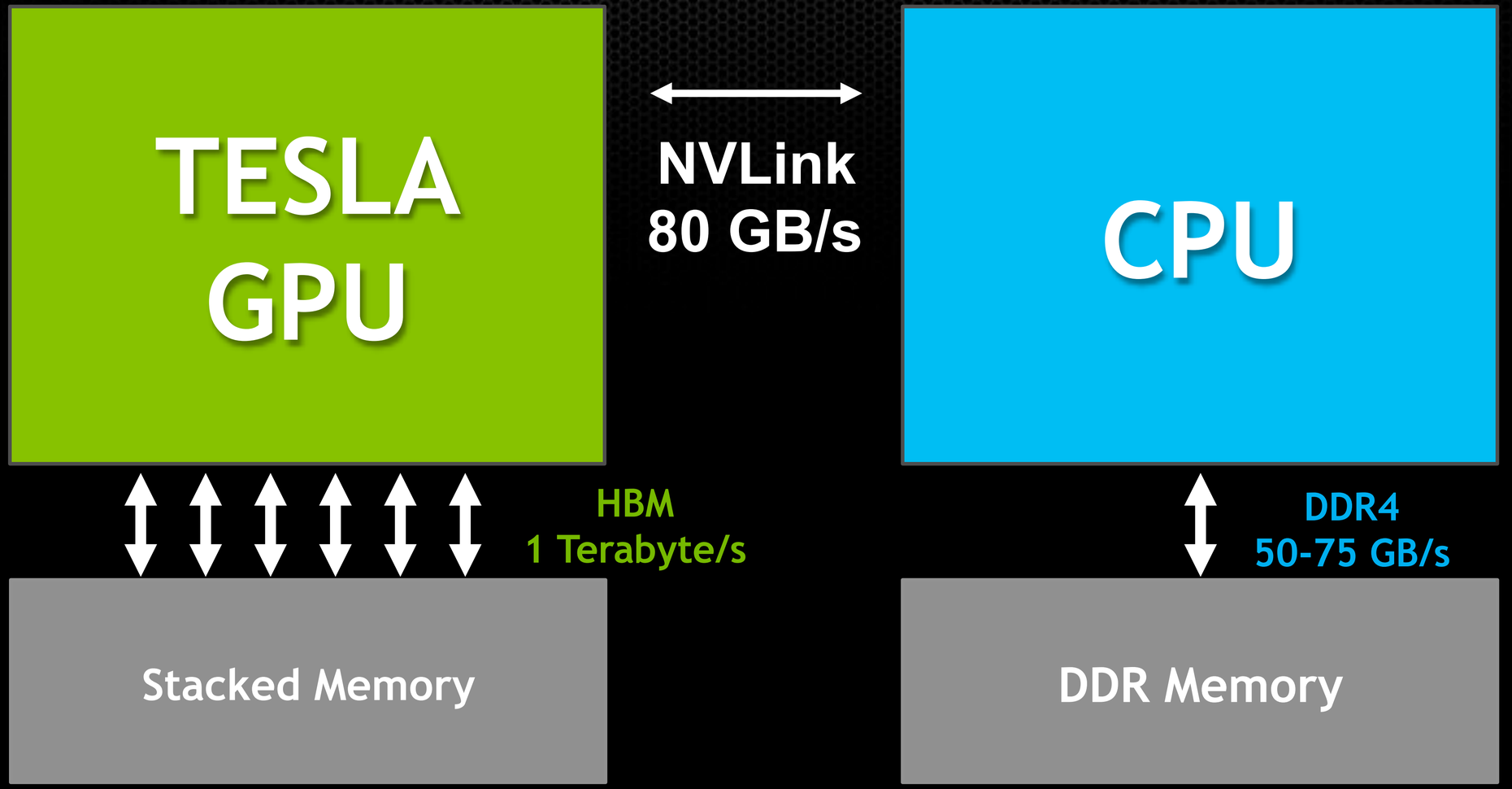






Introduced with Pascal in 2016

GPU Computing in 2016



US to Build Two Flagship Supercomputers

Powered by the Tesla Platform

OAK RIDGE National Laboratory

Lawrence Livermore National Laboratory

IBM

NVIDIA

100-300 PFLOPS Peak

10x in Scientific App Performance

IBM POWER9 CPU + NVIDIA Volta GPU

NVLink High Speed Interconnect

40 TFLOPS per Node, >3,400 Nodes

2017

Major Step Forward on the Path to Exascale

TOP500®
SUPERCOMPUTER SITES

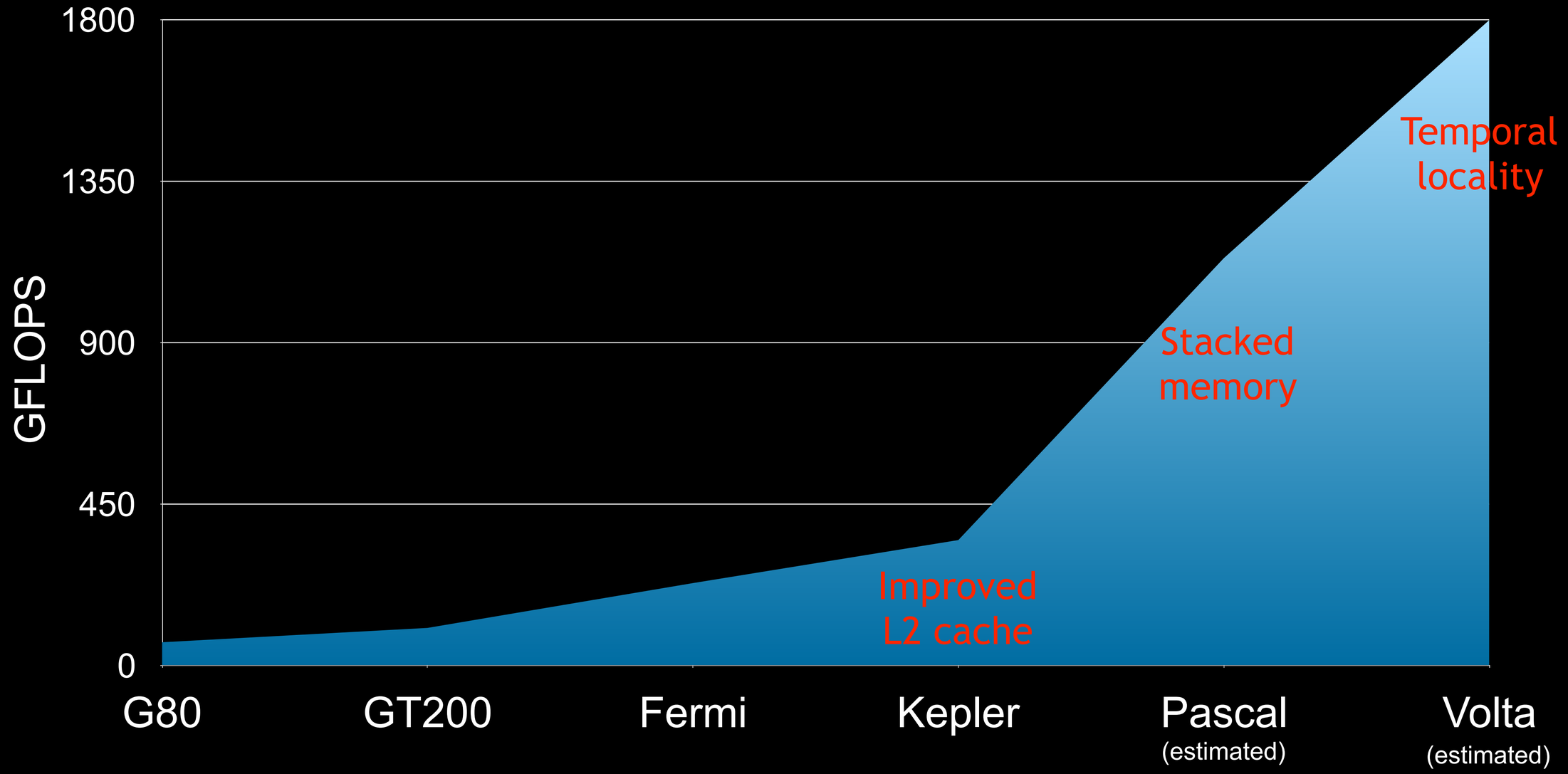
Just 4 nodes in Summit
would make the Top500 list of supercomputers today

Similar Power as Titan
5-10x Faster
1/5th the Size

150 PF = 3M Laptops
One laptop for Every Resident in State of Mississippi

LQCD Performance with GPU generation

Single Precision Wilson-Dslash performance, $V=24^4$



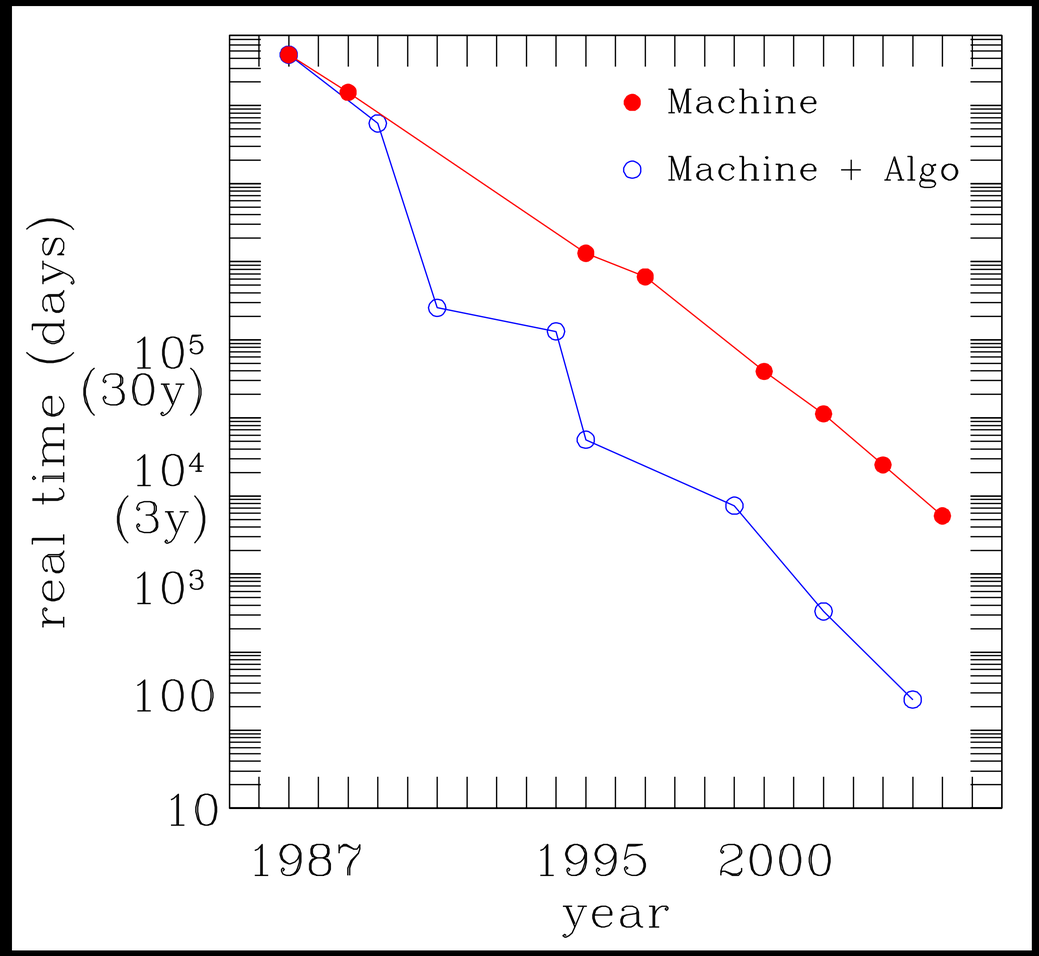
How to get to the Exascale (and beyond)?

- Four challenges to overcome
  - Communication
  - Latency
  - Parallelism
  - Locality
- What's the answer to all of the above?

How to get to the Exascale (and beyond)?

- Four challenges to overcome
  - Communication
  - Latency
  - Parallelism
  - Locality

- What's the answer to all of the above?



K Jansen

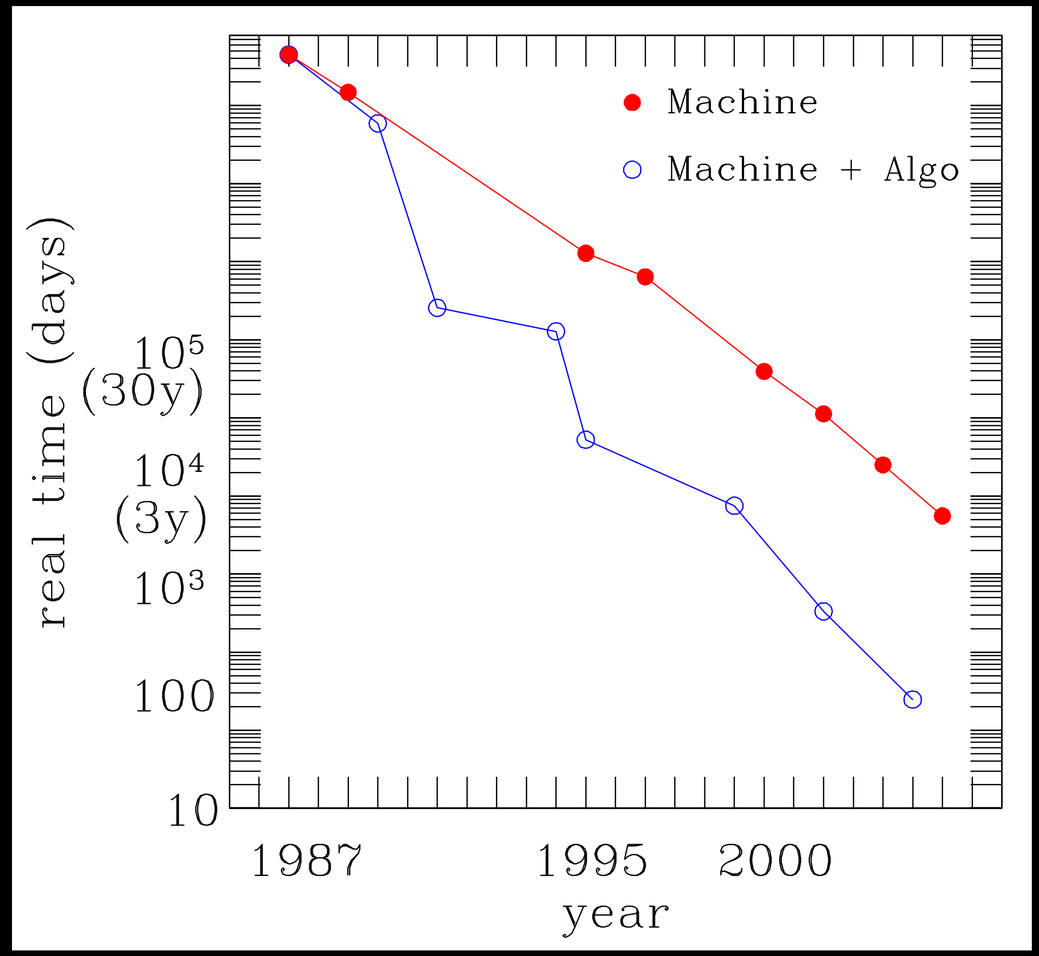
How to get to the Exascale (and beyond)?

- Four challenges to overcome
  - Communication
  - Latency
  - Parallelism
  - Locality

- What's the answer to all of the above?

Algorithms

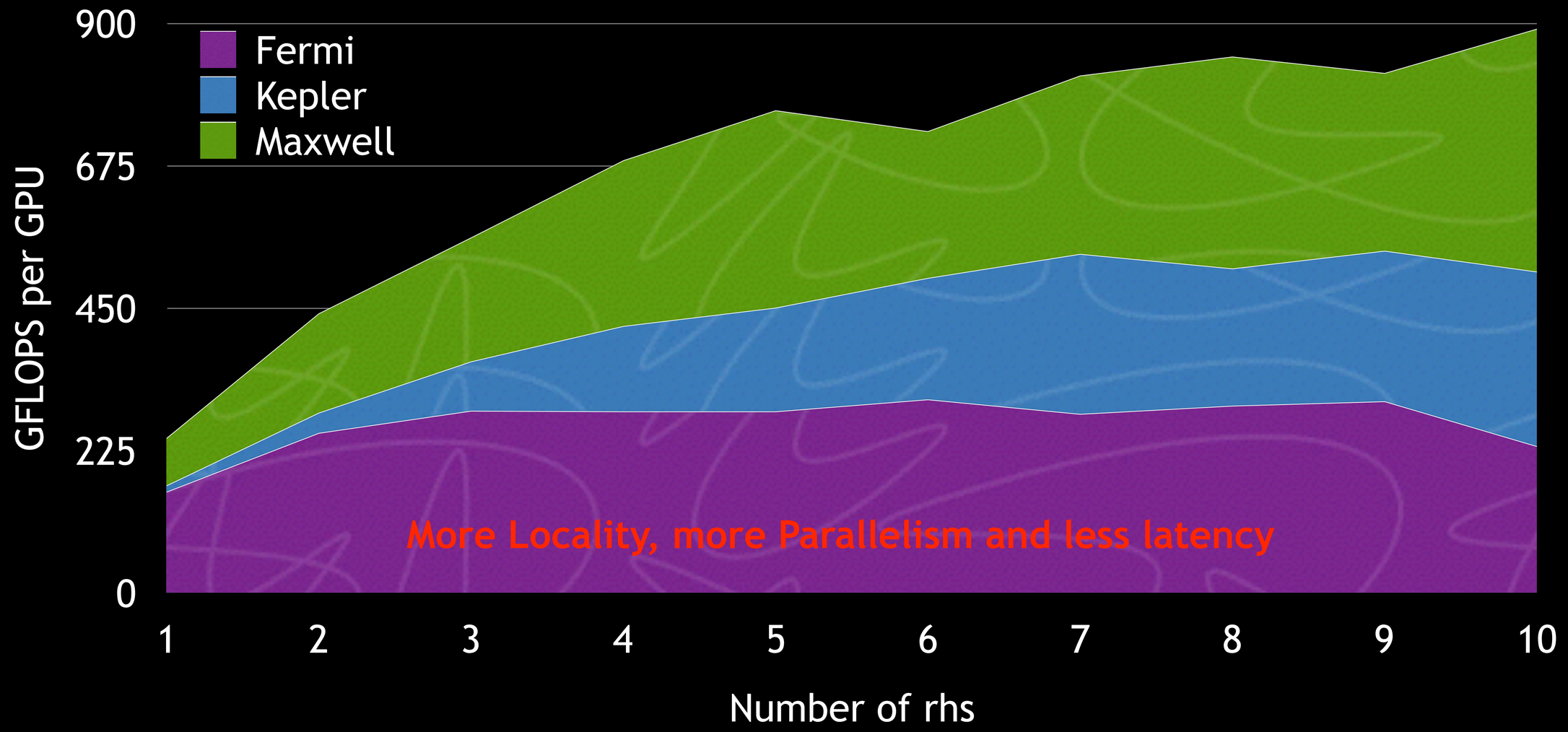
**and the ability to express and utilize those algorithms*



K Jansen

Example: Multi-right-hand-side Dslash

Improved staggered, volume = 24^4, single precision, no reconstruction



Software and Algorithms

- Solvers continue to innovate rapidly
  - Communication avoiding Krylov solvers (Demmel et al)
  - Cooperative Krylov methods (Bhaya et al)
  - Enlarged Krylov space methods (Grigori et al)

- Software can be the problem
  - Hierarchical grids breaks most LQCD frameworks
  - Used to calling solvers in a serial fashion
  - Precision is often baked in

Fine-grained Parallelism and the Implications for DSLs

- Traditional DSL approach is to abstract the grid parallelism

```
Matrix u;
Vector x, y;
y = u * x;
```

- Compiler / front end will then transform this expression into a data parallel operation using OpenMP / CUDA / C++ meta template magic, etc.

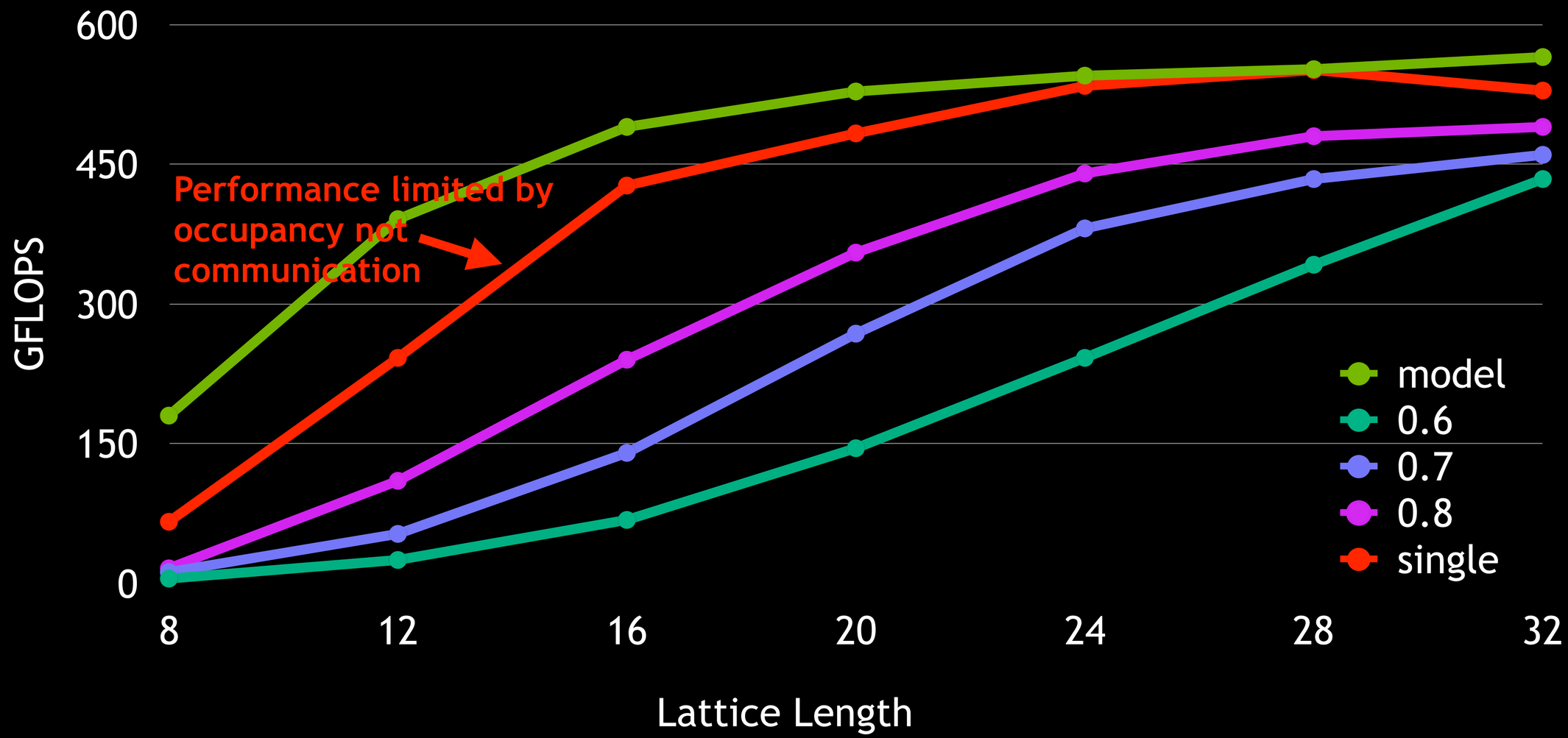
- This abstraction breaks with multigrid
  - Not enough grid parallelism

- Platform and algorithmic independent conjecture

“Fine-grained parallelization will becoming increasingly a requirement at the Exascale (and beyond)”

Dslash Strong Scaling

K40, wilson, half precision, 8-way communication, 12 reconstruct



Communication at the Exascale

- Traditional two-sided MPI will not scale
  - Overheads will dominate communication
- Expect machine-wide unified address space
  - Every CPU/GPU can read/write directly to each other
  - SHMEM / UPC / MPI-3 everywhere

- Utilize the GPU's fine-grained latency hiding through thread oversubscription to hide inter-node latency
  - LQCD is an ideal application for this programming model

- QUDA is one of the applications being investigated in the DOE's "DesignForward" Exascale programming investigation

Conclusions and Outlook

- GPUs provide a compelling platform for lattice computation

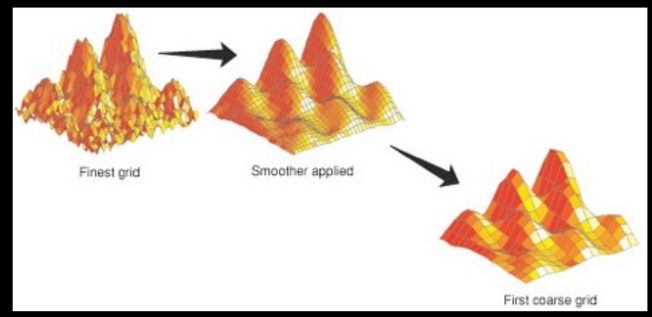
- Multigrid algorithms are running well on GPUs
  - Much more work to do
  - Fine-grained parallelization is key
  - Glimpse into the challenges of the exascale

- Importance of algorithms will only increase

- Exascale potentially challenging from a software point of view

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



Halo Region Updates (QUDA 0.7)

- Best way to reduce latency all round is kernel fusion
  - Reduces API calls
  - Reduces kernel launch overhead
  - Increases GPU occupancy
- Previous multi-GPU dslash had 6 kernels
  - pack (all faces), interior, halo_t, halo_z, halo_y, halo_x
- This puts a lower bound on the minimum time taken regardless of the speed of the GPU execution
- Fused multi-GPU dslash now has 3 kernels halving lower bound
  - pack (all faces), interior, halo (all faces)
- Scope for further fusing if we consider D_{eo} D_{oe} together
  - pack -> interior -> halo -> pack -> interior -> halo

Other improvements

- Double buffering of QMP/MPI receive buffers (QUDA 0.7)
  - Early pre-posting of MPI Receive
- Dslash has been rewritten using pthreads to parallelize between independent MPI and CUDA API calls (QUDA 0.8)
  - Parallelize between CUDA -> MPI and MPI -> CUDA dependent operations. E.g., waiting on MPI in t dimension while waiting on device -> host copy in z dimension
- Improvement to half-precision latency (QUDA 0.8)
  - Previously norm field was stored in separate halo region
  - Now store in same array as main quark field
    - Halves host -> device API calls
    - Increases message size for improved throughput

What is limiting strong scaling?

What is limiting strong scaling?

- GPU kernel / memcpy launch overhead (4-10 us)
  - Reduce number of kernels / memcpy
  - Use a single kernel for all halo regions (6->3 kernel calls) (0.7)

What is limiting strong scaling?

- GPU kernel / memcpy launch overhead (4-10 us)
  - Reduce number of kernels / memcpy
  - Use a single kernel for all halo regions (6->3 kernel calls) (0.7)
- Halo region kernels don’t saturate the GPU
  - Use a single kernel for all halo regions (4x threads) (0.7)

What is limiting strong scaling?

- GPU kernel / memcpy launch overhead (4-10 us)
  - Reduce number of kernels / memcpy
  - Use a single kernel for all halo regions (6->3 kernel calls) (0.7)
- Halo region kernels don’t saturate the GPU
  - Use a single kernel for all halo regions (4x threads) (0.7)
- MPI / CUDA can block each other from progressing
  - Use pthreads to parallelize CUDA and MPI calls (0.8)

What is limiting strong scaling?

- GPU kernel / memcpy launch overhead (4-10 us)
  - Reduce number of kernels / memcpy
  - Use a single kernel for all halo regions (6->3 kernel calls) (0.7)
- Halo region kernels don’t saturate the GPU
  - Use a single kernel for all halo regions (4x threads) (0.7)
- MPI / CUDA can block each other from progressing
  - Use pthreads to parallelize CUDA and MPI calls (0.8)
- PCIe bus contention on Multi-GPU nodes
  - Use CUDA peer-to-peer API for direct communication (0.9)

What is limiting strong scaling?

- GPU kernel / memcpy launch overhead (4-10 us)
  - Reduce number of kernels / memcpy
  - Use a single kernel for all halo regions (6->3 kernel calls) (0.7)
- Halo region kernels don’t saturate the GPU
  - Use a single kernel for all halo regions (4x threads) (0.7)
- MPI / CUDA can block each other from progressing
  - Use pthreads to parallelize CUDA and MPI calls (0.8)
- PCIe bus contention on Multi-GPU nodes
  - Use CUDA peer-to-peer API for direct communication (0.9)
- MPI / CUDA have to interact synchronously via CPU
  - GPU Direct Async coming with CUDA 8.0

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

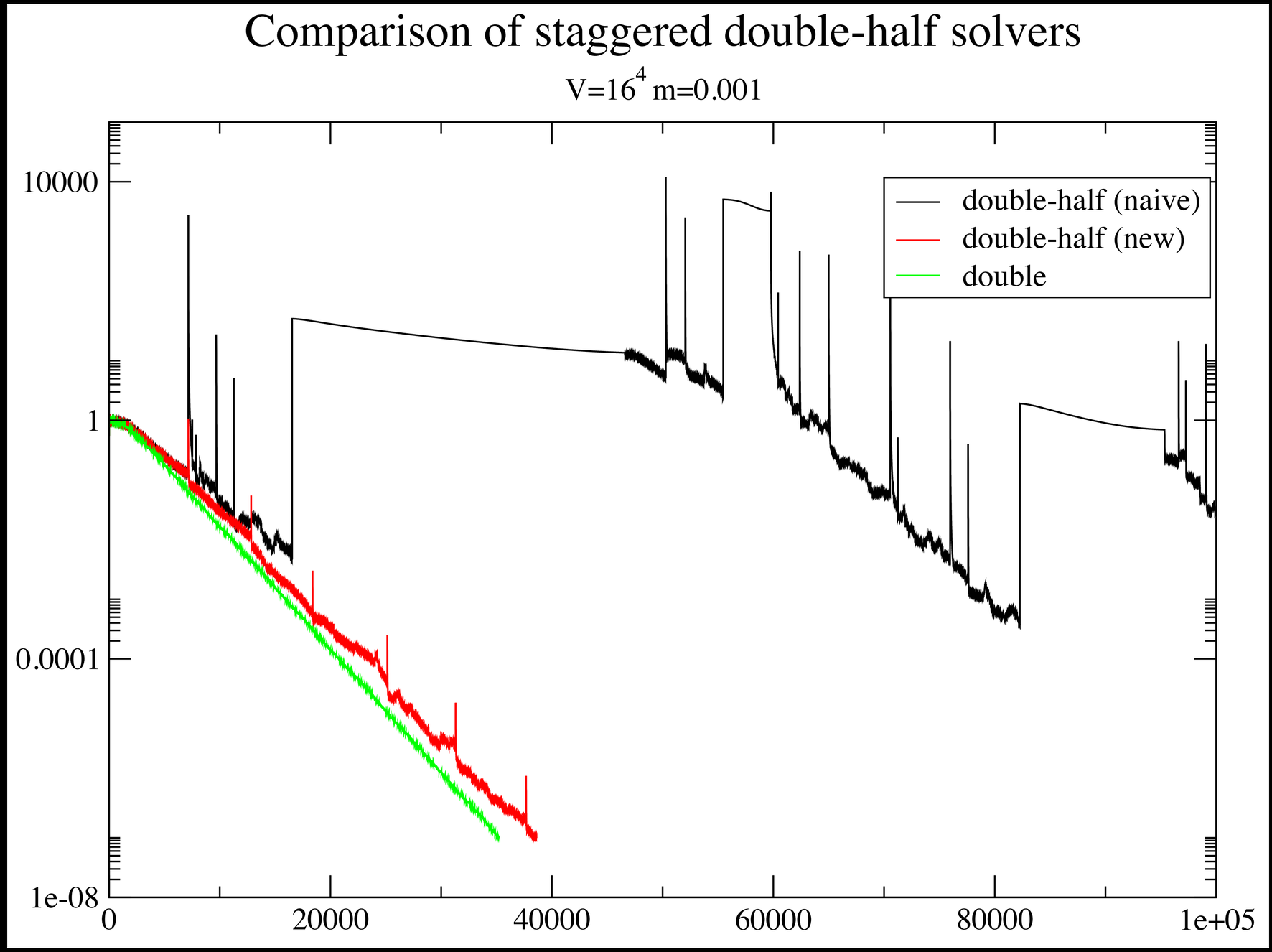
(Stable) Mixed-precision CG

- 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
}
```


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

Multigrid vs BiCGstab

Anisotropic Clover, V = 40^3x256, m_π =230 MeV, strange quark, 32 nodes of Titan

QUDA MG		*QUDA BiCGStab*		*Speedup*
Iterations	*Time (sec)*	*Iterations*	*Time (sec)*	
12	*1.74*	*178*	*2.19*	*1.25*
12	*1.74*	*167*	*2.04*	*1.18*
12	*1.74*	*186*	*2.24*	*1.29*
12	*1.77*	*163*	*2.01*	*1.13*
12	*1.74*	*171*	*2.04*	*1.18*
12	*1.74*	*184*	*2.26*	*1.29*
12	*1.75*	*173*	*2.09*	*1.19*
12	*1.73*	*161*	*1.94*	*1.12*
12	*1.74*	*179*	*2.20*	*1.26*
12	*1.73*	*208*	*2.53*	*1.46*
12	*1.73*	*163*	*1.97*	*1.14*
12	*1.74*	*169*	*2.06*	*1.19*
Average	*1.74*		*2.13*	*1.22*

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, Möbius
- Condition number inversely proportional to mass
  - Light (realistic) masses are highly singular
  - Naive Krylov solvers suffer from critical slowing down at decreasing mass

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

conjugate gradient

Coarse Grid Operator Performance