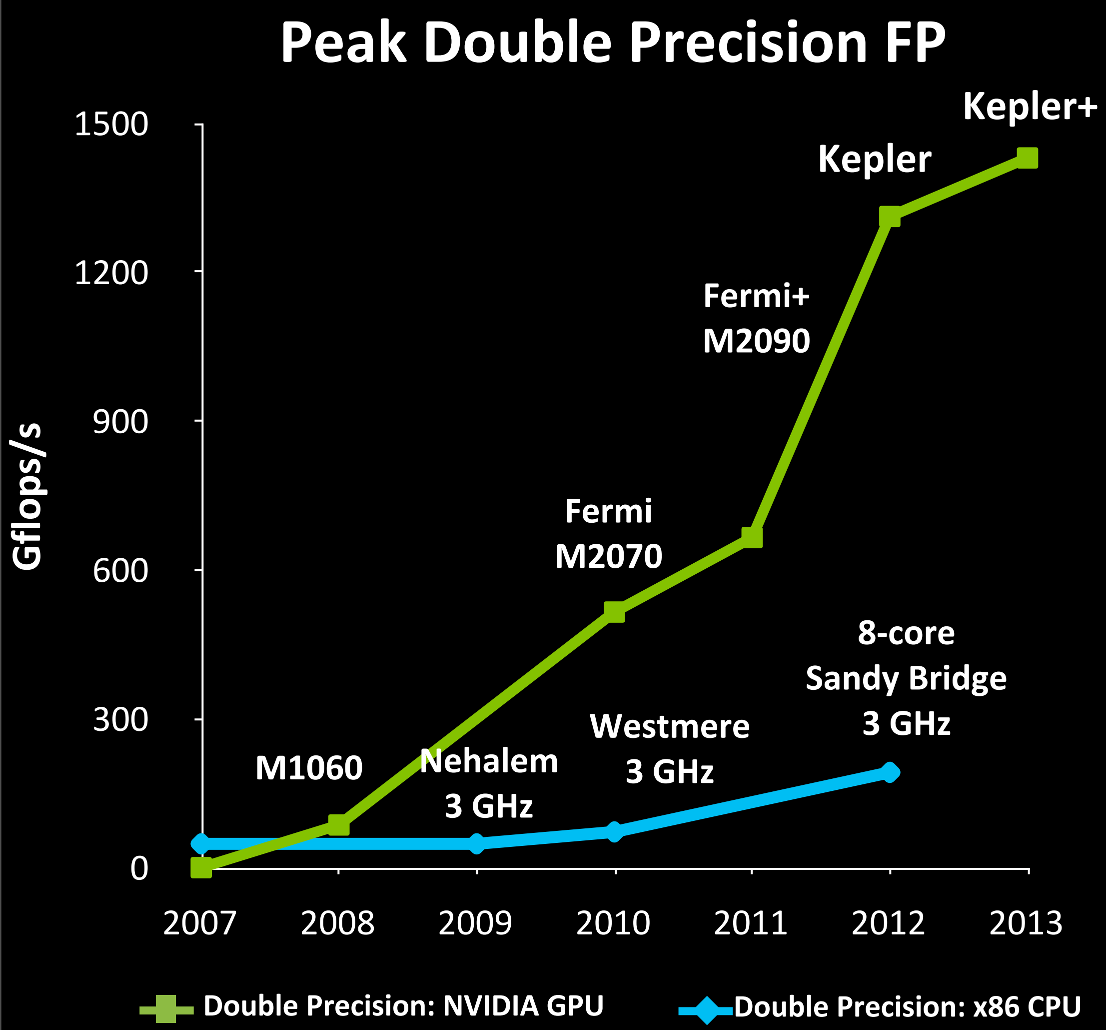
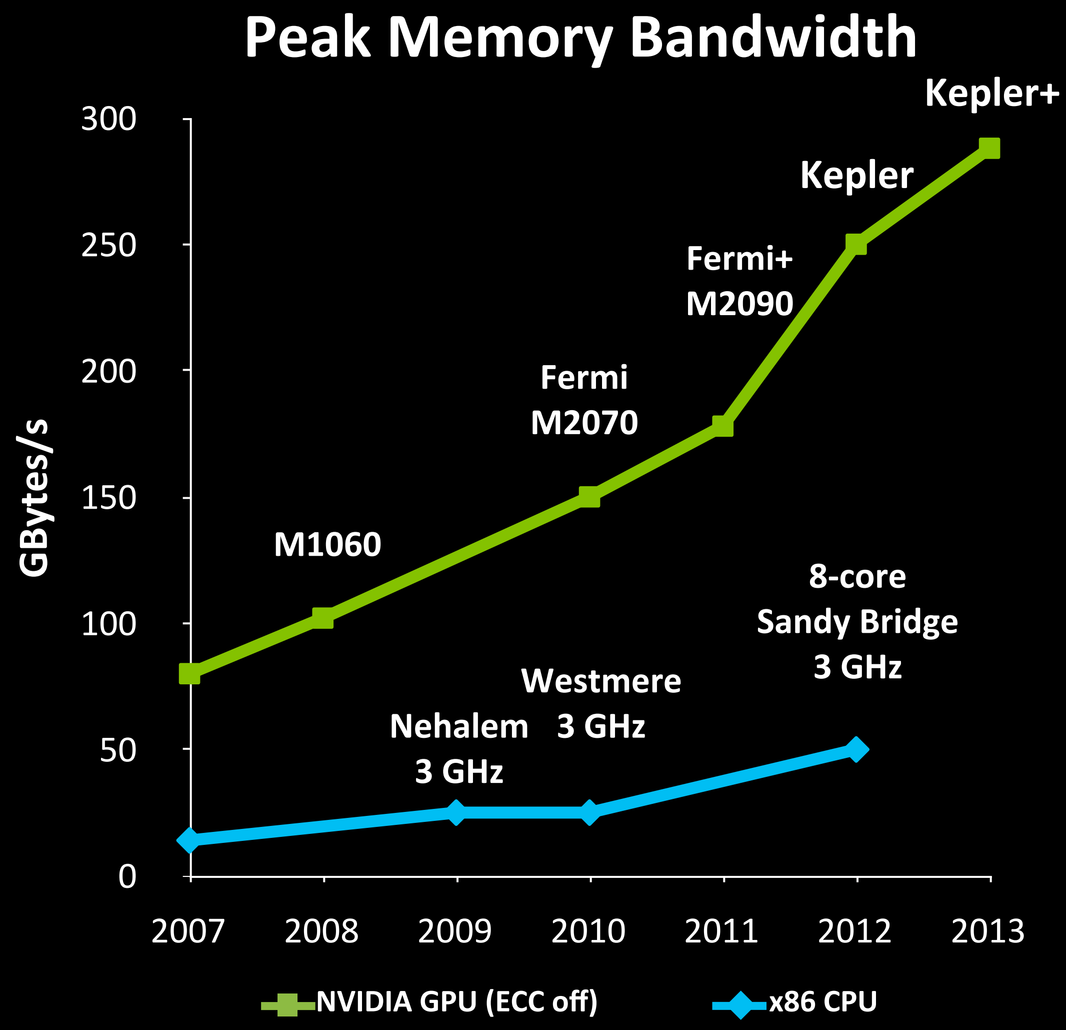
The QUDA Library

M Clark, NVIDIA
Developer Technology Group

The March of GPUs





QCD applications

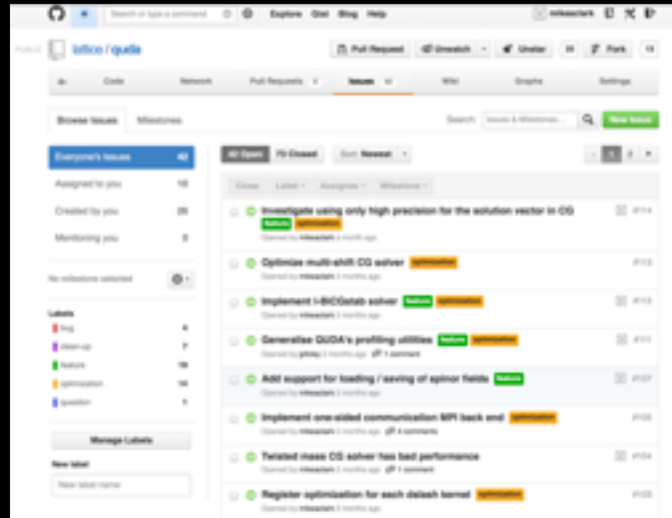
- Some examples
  - MILC (FNAL, Indiana, Arizona, Utah)
    - strict C, MPI only
  - CPS (Columbia, Brookhaven, Edinburgh)
    - C++ (but no templates), MPI and partially threaded
  - Chroma (Jefferson Laboratory, Edinburgh)
    - C++ expression-template programming, MPI and threads
  - BQCD (Berlin QCD)
    - F90, MPI and threads
- Each application consists of 100K-1M lines of code
- Porting each application not directly tractable
  - OpenACC possible for well-written code “Fortran-style” code (BQCD, maybe MILC)

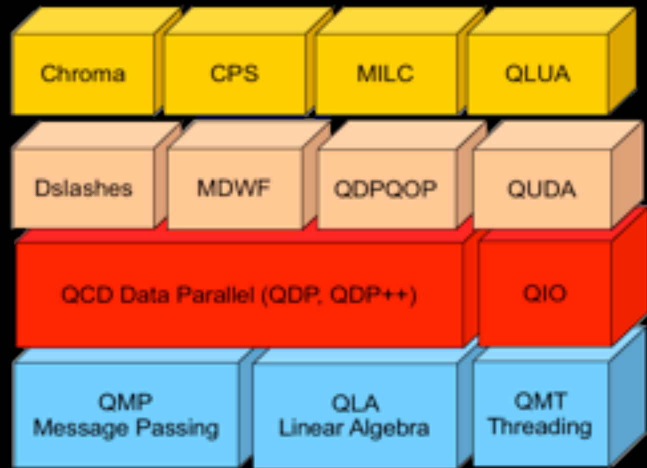
Enter QUDA

- "QCD on CUDA" – http://lattice.github.com/quda
  - Written in C / C++ / Python
- 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

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 (FNAL)
- Alejandro Vaquero (Cyprus Institute)
- Frank Winter (Jlab)
- Yibo Yang (IHEP)





QUDA Mission Statement

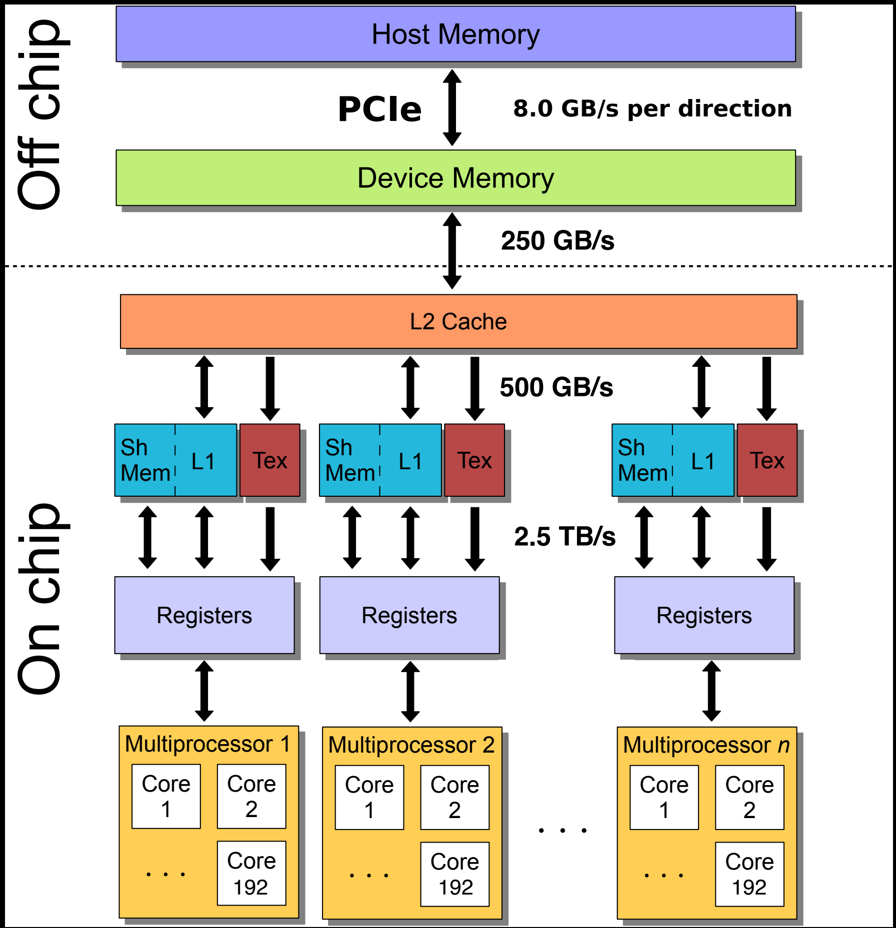
- QUDA is
  - a library enabling legacy applications to run on GPUs
  - open source so anyone can join the fun
  - evolving
    - more features
    - cleaner, easier to maintain
  - a research tool into how to reach the exascale
    - Lessons learned are mostly (platform) agnostic
    - Domain-specific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

QUDA High-Level Interface

- QUDA default interface provides a simple view for the outside world
  - C or Fortran
  - Host applications simply pass cpu-side or gpu-side pointers (new!)
  - QUDA takes care of all field reordering and data copying
  - No GPU code in user application
- Limitation
  - No control over memory management
  - No external opaque gpu objects
  - Considering different strawman

```
#include <quda.h>

int main() {

  // initialize the QUDA library
  initQuda(device);

  // load the gauge field
  loadGaugeQuda((void*)gauge, &gauge_param);

  // perform the linear solve
  invertQuda(spinorOut, spinorIn, &inv_param);

  // free the gauge field
  freeGaugeQuda();

  // finalize the QUDA library
  endQuda();

}
```

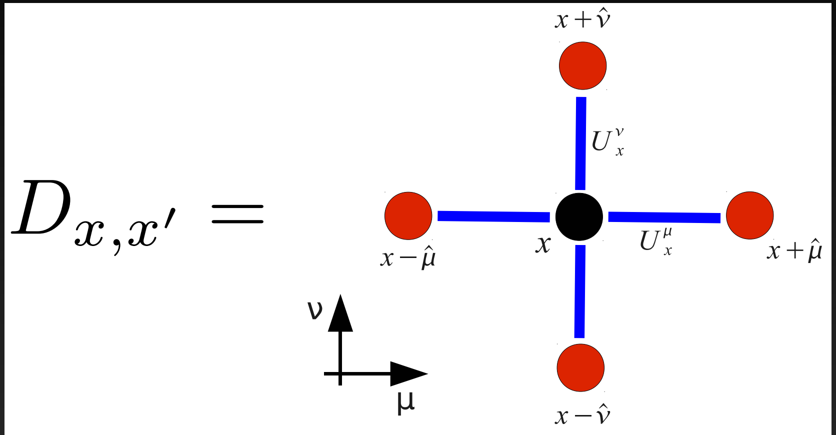
The Kepler Architecture



- 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

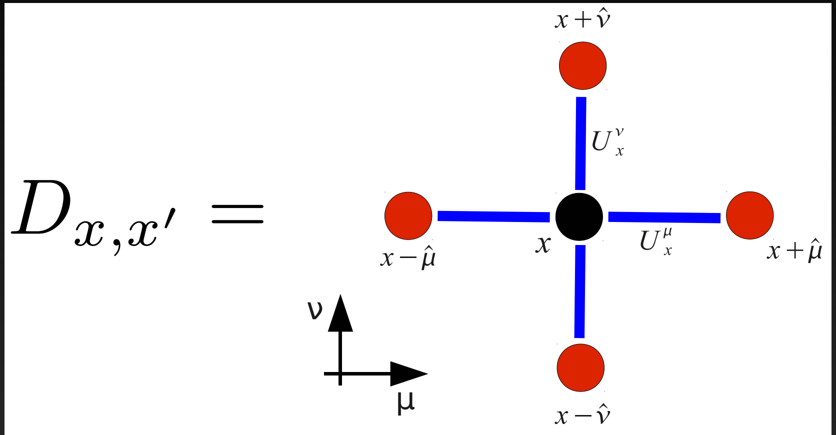
Mapping the Wilson Dslash to CUDA

- 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

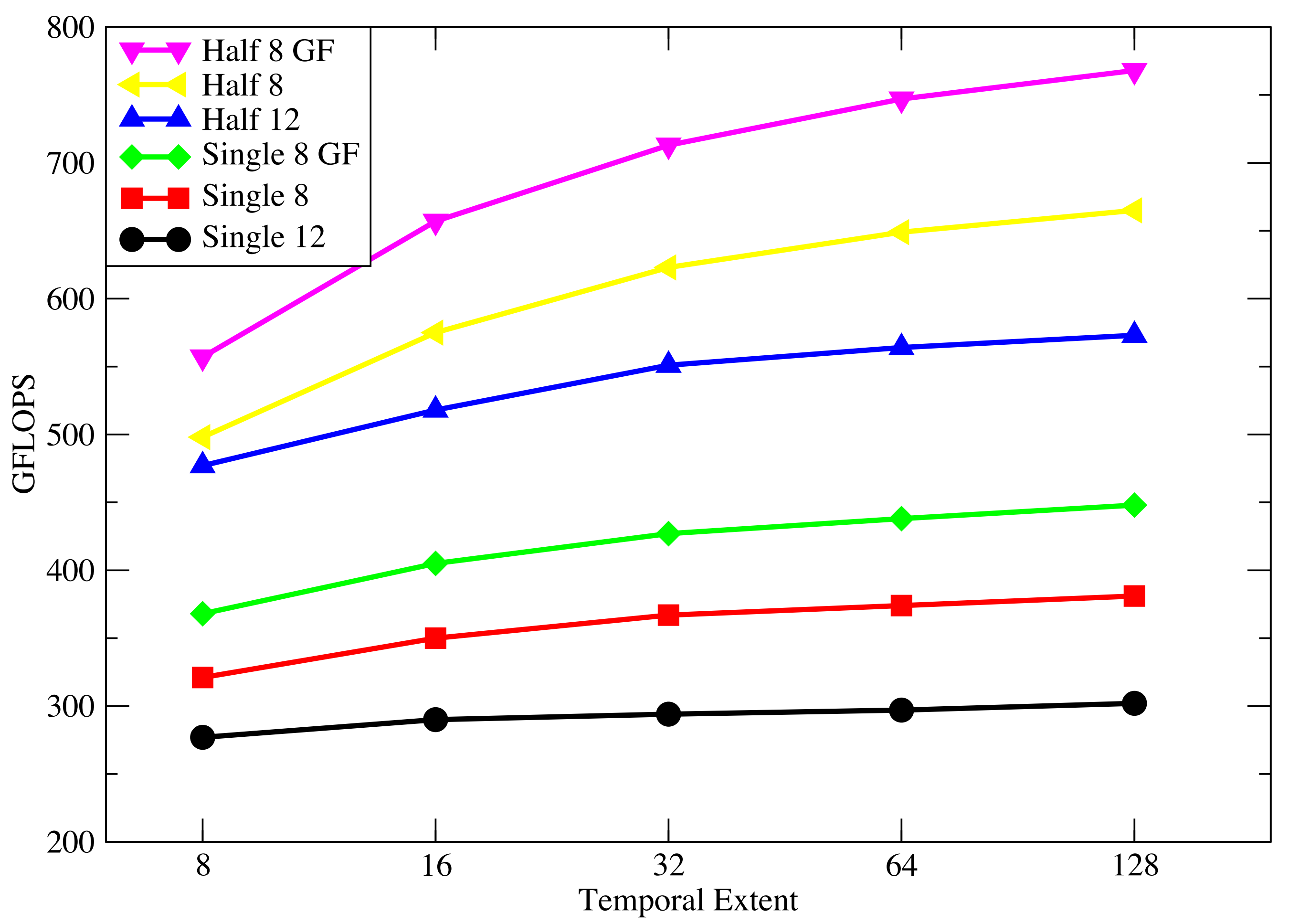
Reducing Memory Traffic

- SU(3) matrices are all unitary complex matrices with det = 1
  - 12-number parameterization: reconstruct full matrix on the fly in registers

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \rightarrow \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix} \quad \mathbf{c} = (\mathbf{a} \times \mathbf{b})^*$$

    - Additional 384 flops per site
  - Also have an 8-number parameterization (requires sin/cos and sqrt)
- Impose similarity transforms to increase sparsity
- Still memory bound - Can further reduce memory traffic by truncating the precision
  - 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



K20X Dslash performance
V = 24^3xT
Wilson-Clover is ±10%

GeForce GTX Titan
> 1 TFLOPS

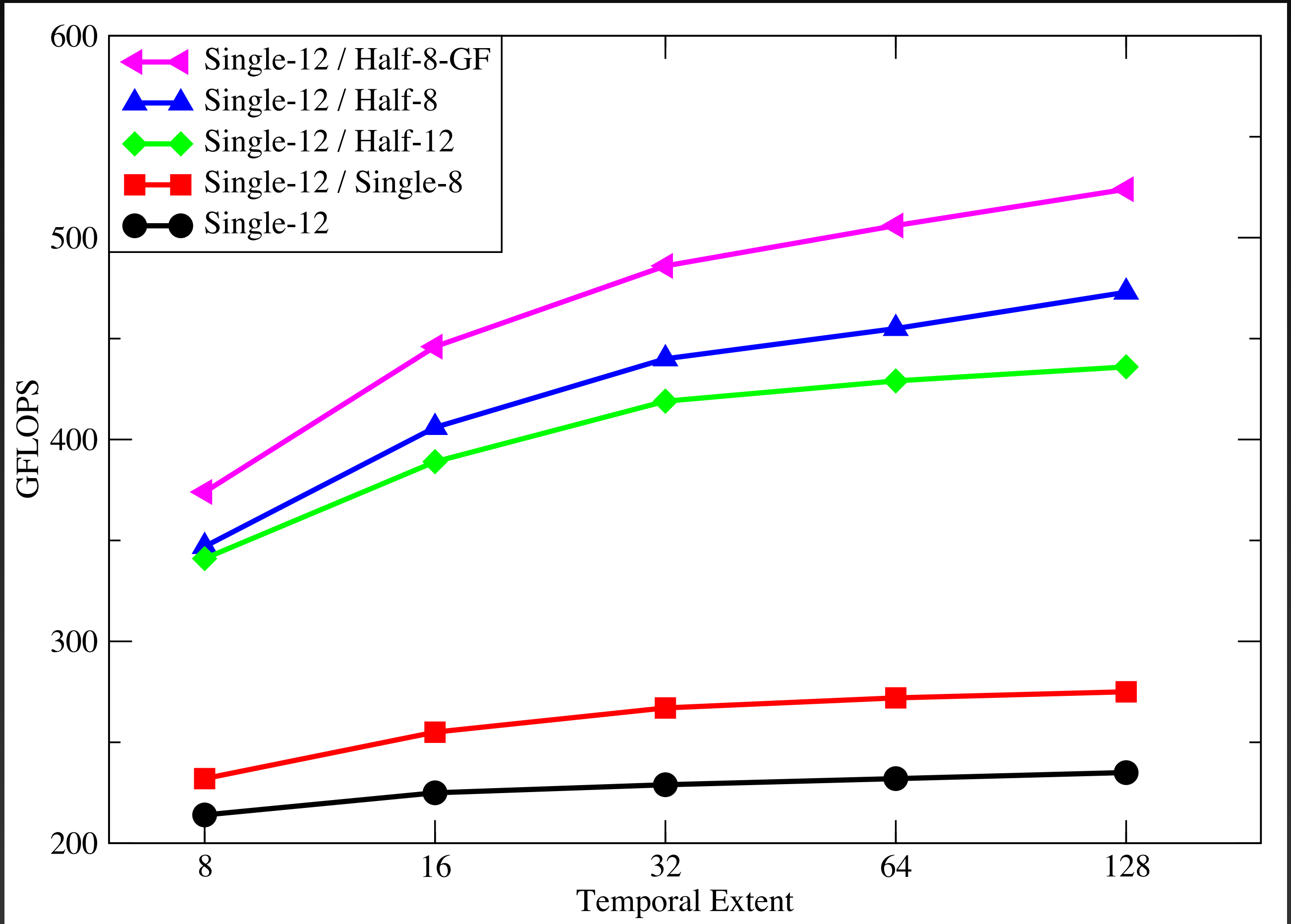
Krylov Solver Implementation

- 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

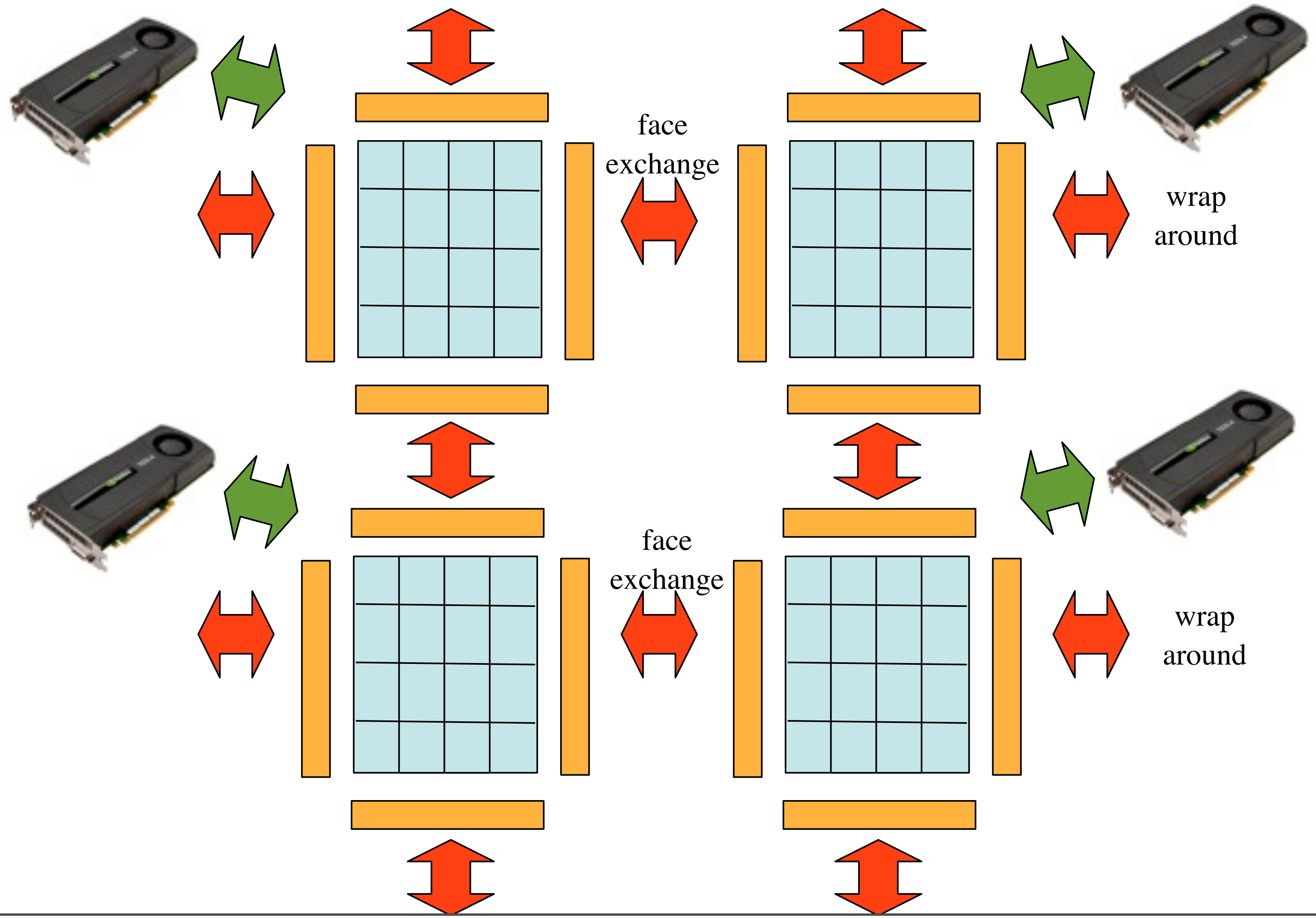
$$
\begin{aligned}
&\text{while } (|\mathbf{r}_k| > \varepsilon)\ \{\\
&\quad \beta_k = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{r}_{k-1},\mathbf{r}_{k-1})\\
&\quad \mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k\mathbf{p}_k\\
&\\
&\quad \alpha = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{p}_{k+1},A\mathbf{p}_{k+1})\\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

Kepler Wilson-Solver Performance



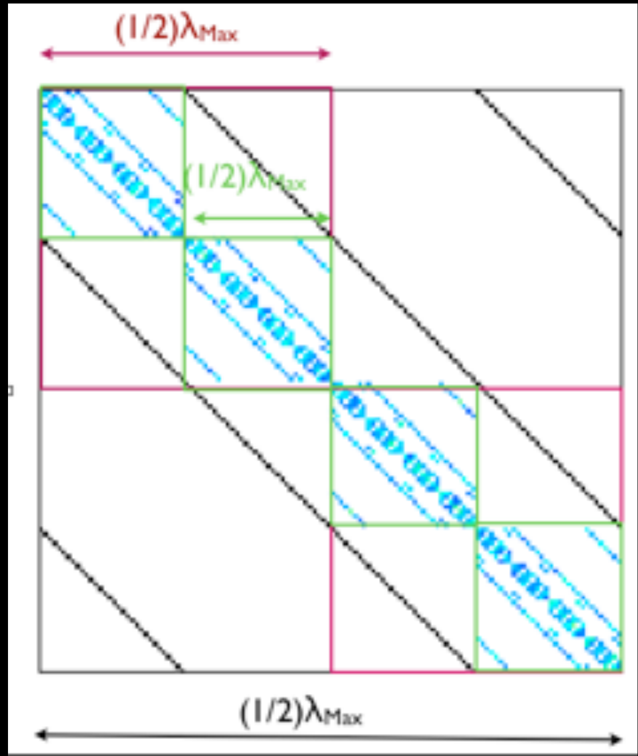
K20X performance
$V = 24^3 xT$
Wilson-Clover is ±10%
BiCGstab is -10%

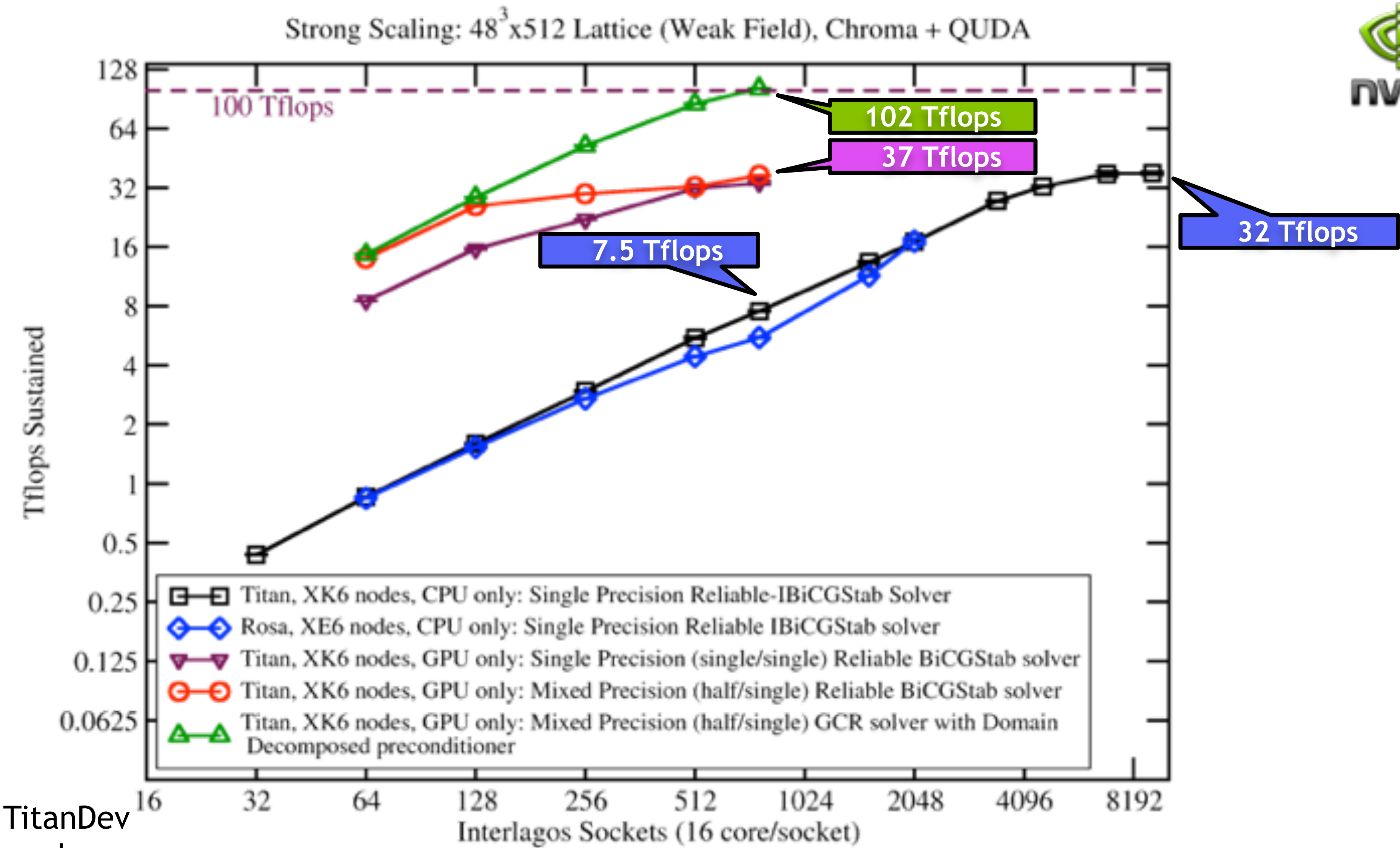
Multi-dimensional lattice decomposition



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 ~ $O(\Lambda_{QCD}^{-1})$, Λ_{QCD}^{-1} ~ 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





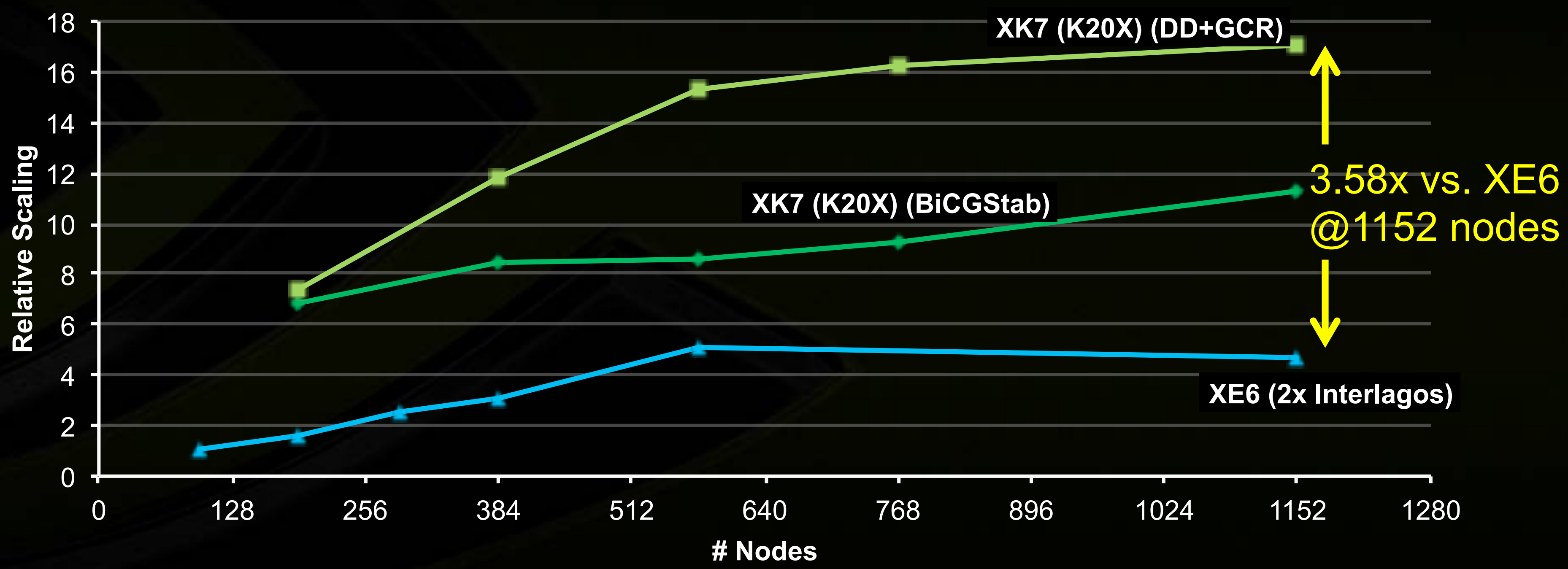
Results from TitanDev
- 48^3x512 aniso clover
- scaling up 768 GPUs

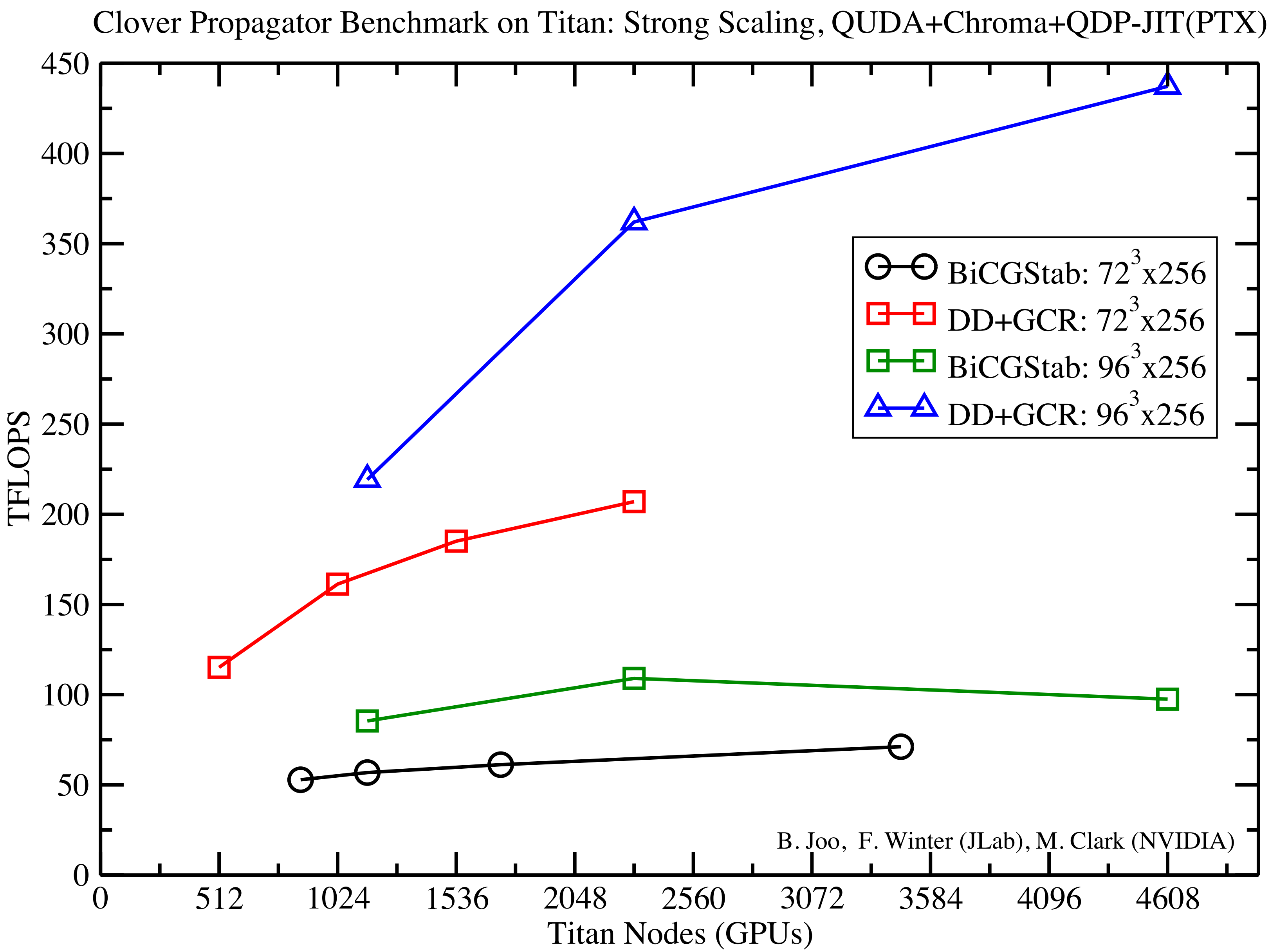
Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma

48^3x512 lattice
Relative Scaling (Application Time)

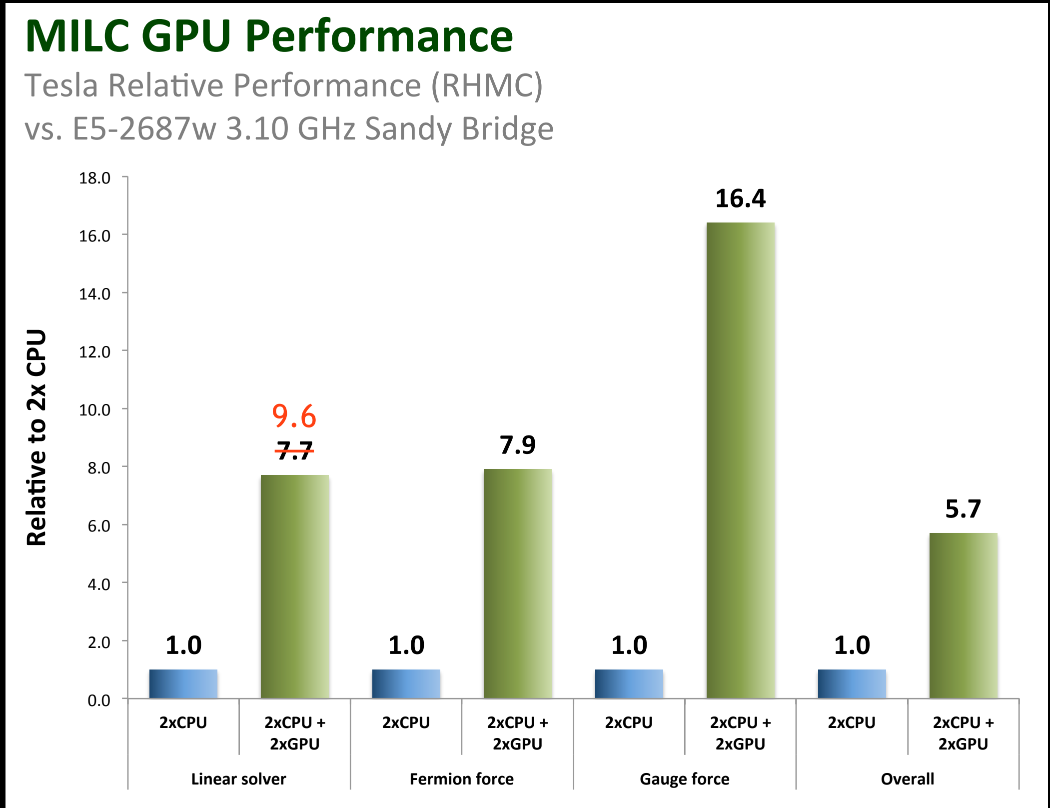
"XK7" node = XK7 (1x K20X + 1x Interlagos)
"XE6" node = XE6 (2x Interlagos)




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)
NVIDIA

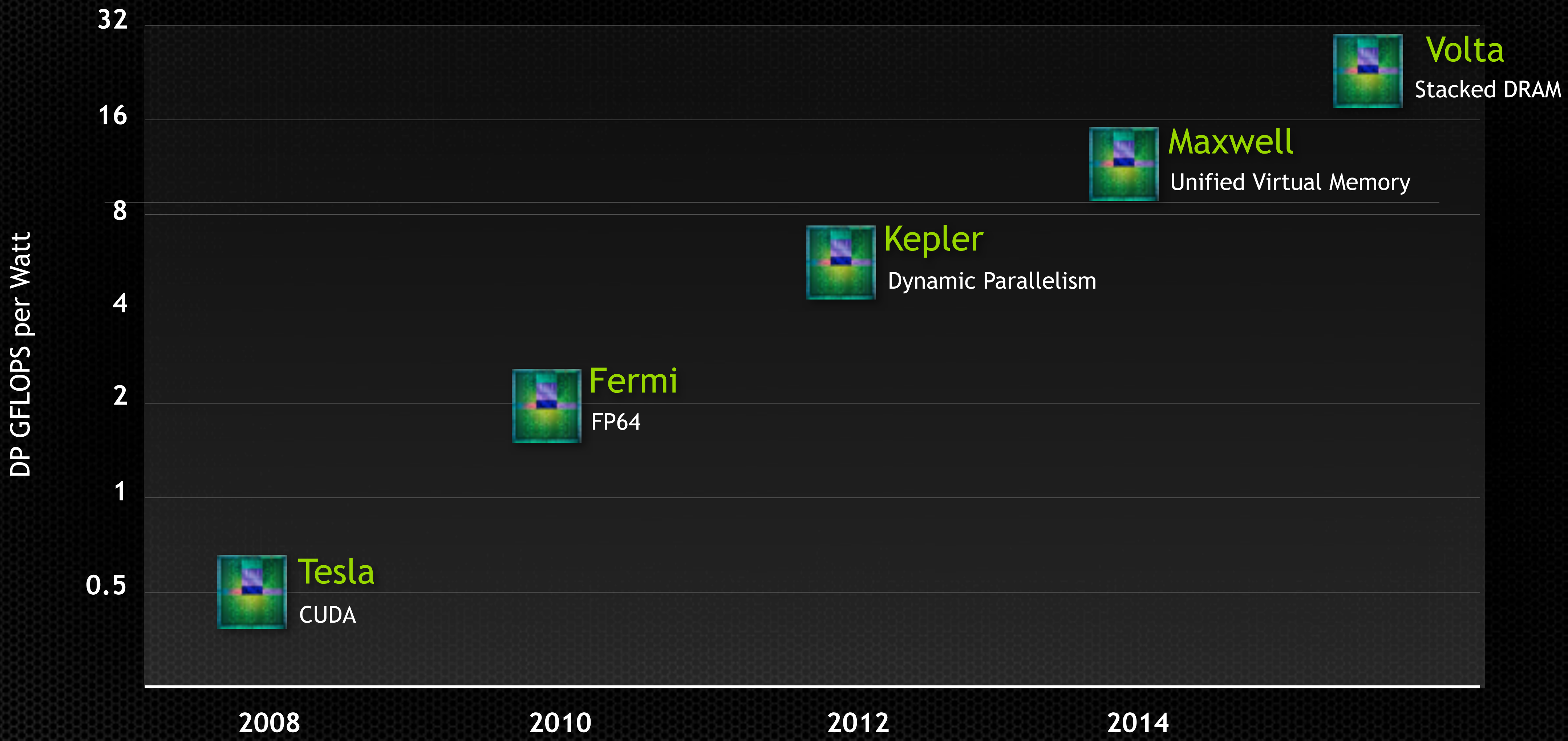
MILC on QUDA

- Gauge generation benchmark 256 BW nodes
  - Volume = 24^3x64
  - QUDA: solver, forces, fat link
  - MILC: long link, outer product
- MILC is multi-process only
  - 6x net gain in performance
  - But potential >8x gain in performance
  - Porting remaining functions (J. Foley)
    - Long link next week
    - Outer product after that

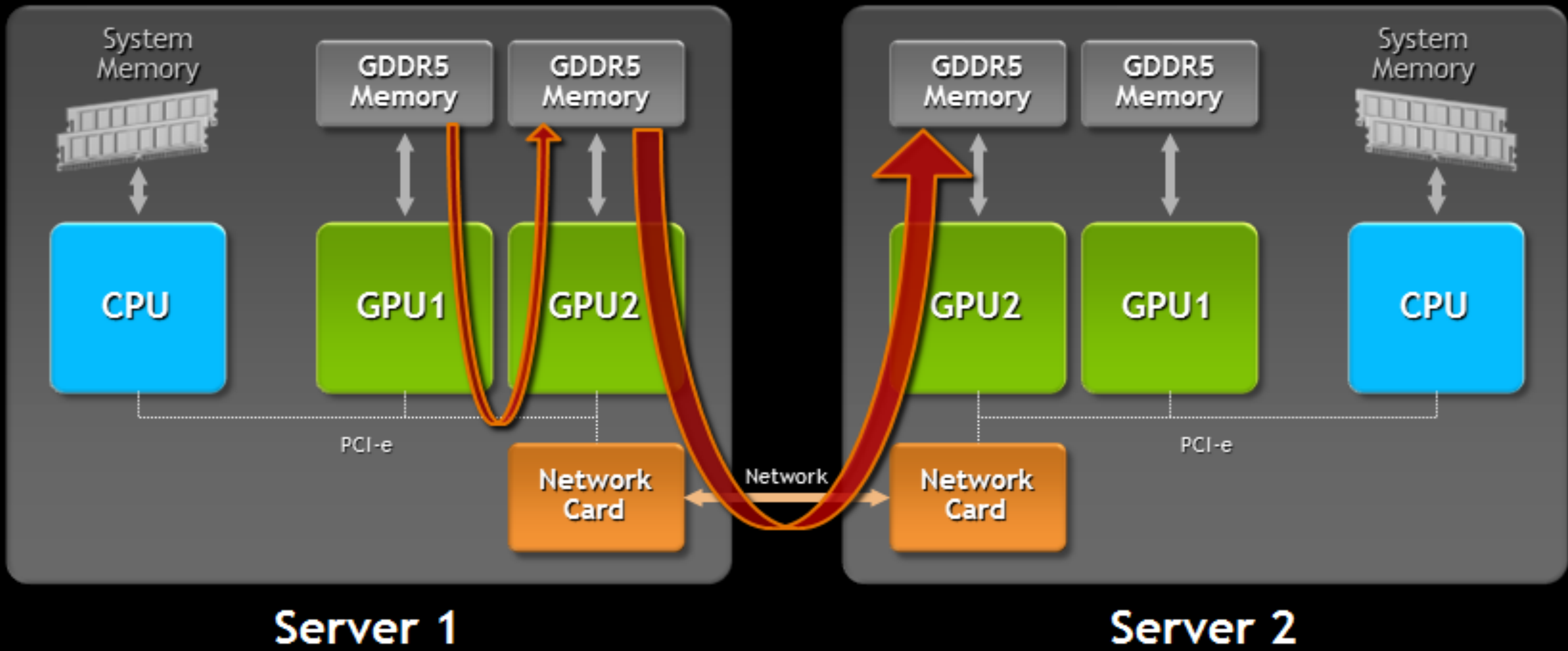


Future Directions

GPU Roadmap


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

GPUDirect



- GPUDirect RDMA will radically improve strong scaling
  - Coming in soon in QUDA

Future Directions

- LQCD coverage (avoiding Amdahl)
  - Remaining force terms needed for gauge generation
  - Contractions
  - Eigenvector solvers (EigCG probably first)
- Performance
  - Locality
  - Learning from today's lessons (software and hardware)
- Hierarchical Algorithm Toolbox
  - Adaptive Multigrid
  - Domain decomposition
  - Mixed-precision solvers
  - Provide an environment to experiment with optimal scalable solvers

Conclusions

mclark at nvidia dot com

- Introduction to QUDA
- Optimal performance required domain-specific knowledge
- Legacy Applications ready for accelerators
- Still lots of work to do
  - New developers welcome
- Lessons today are relevant for Exascale preparation

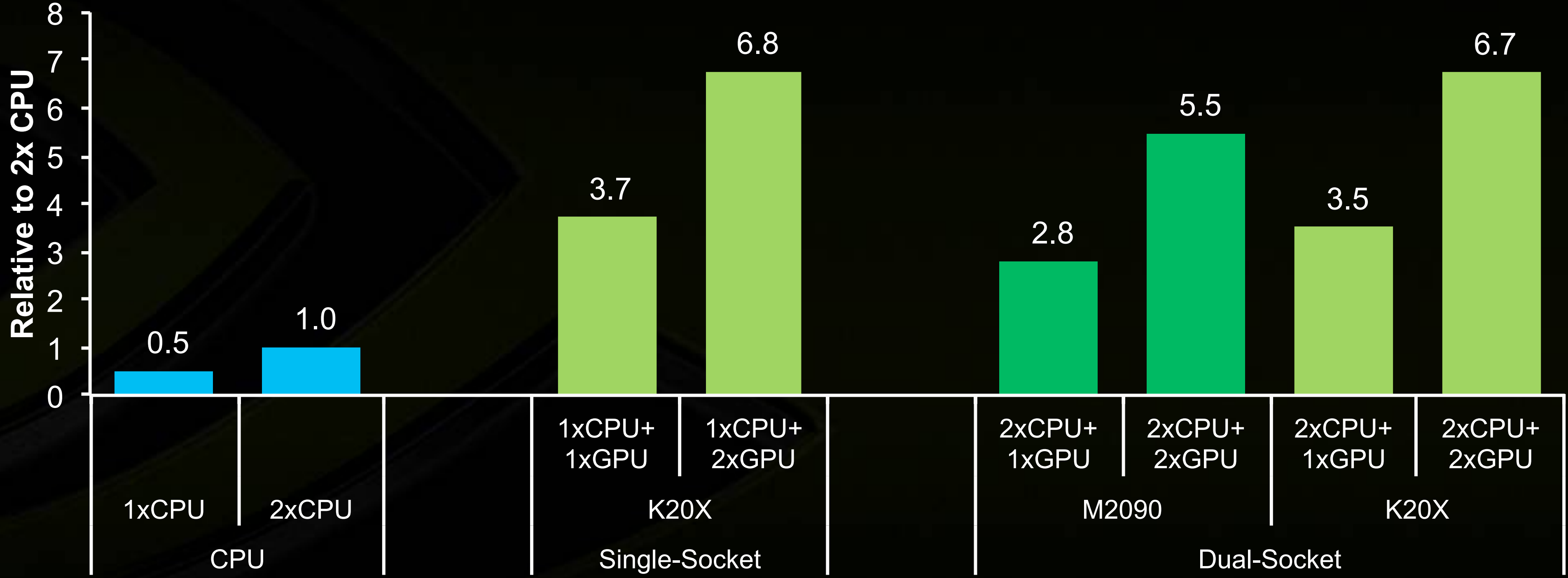
Backup slides

Chroma (Lattice QCD) – High Energy & Nuclear Physics

Chroma

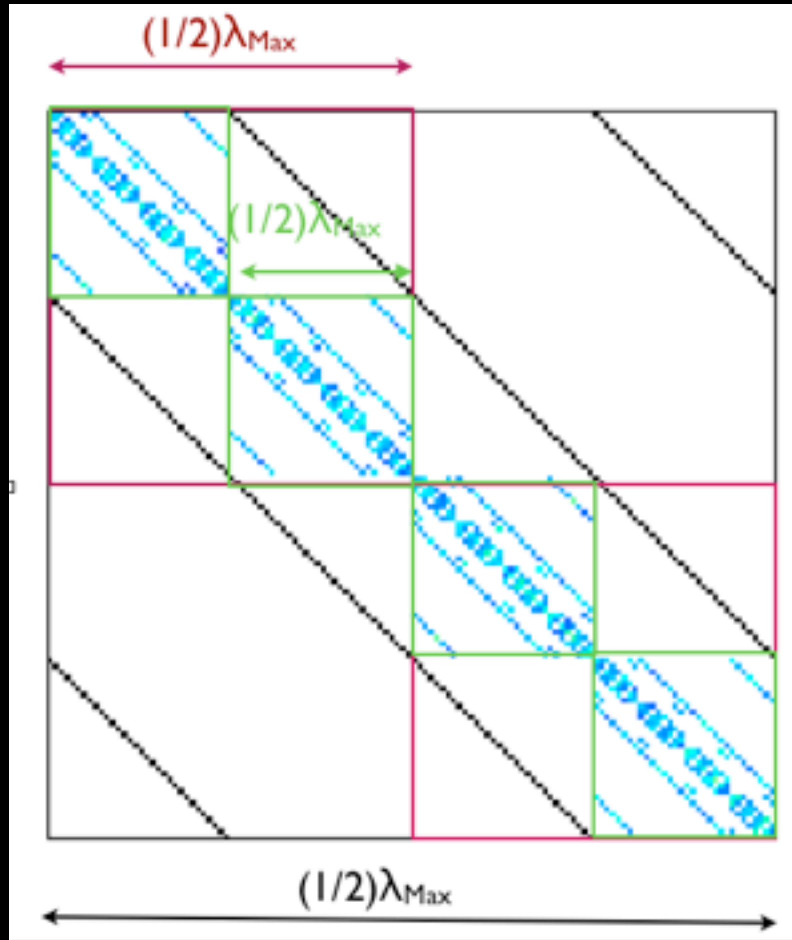
24^3x128 lattice

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



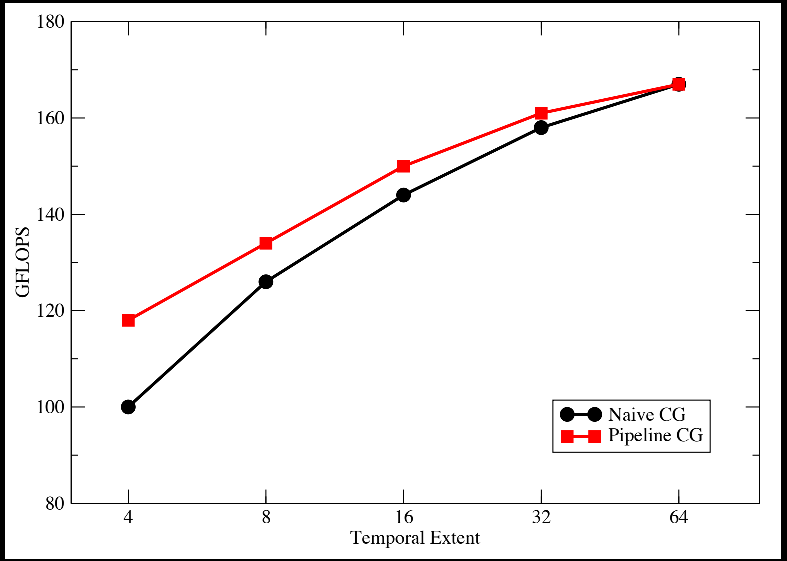
Future Directions - Communication

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

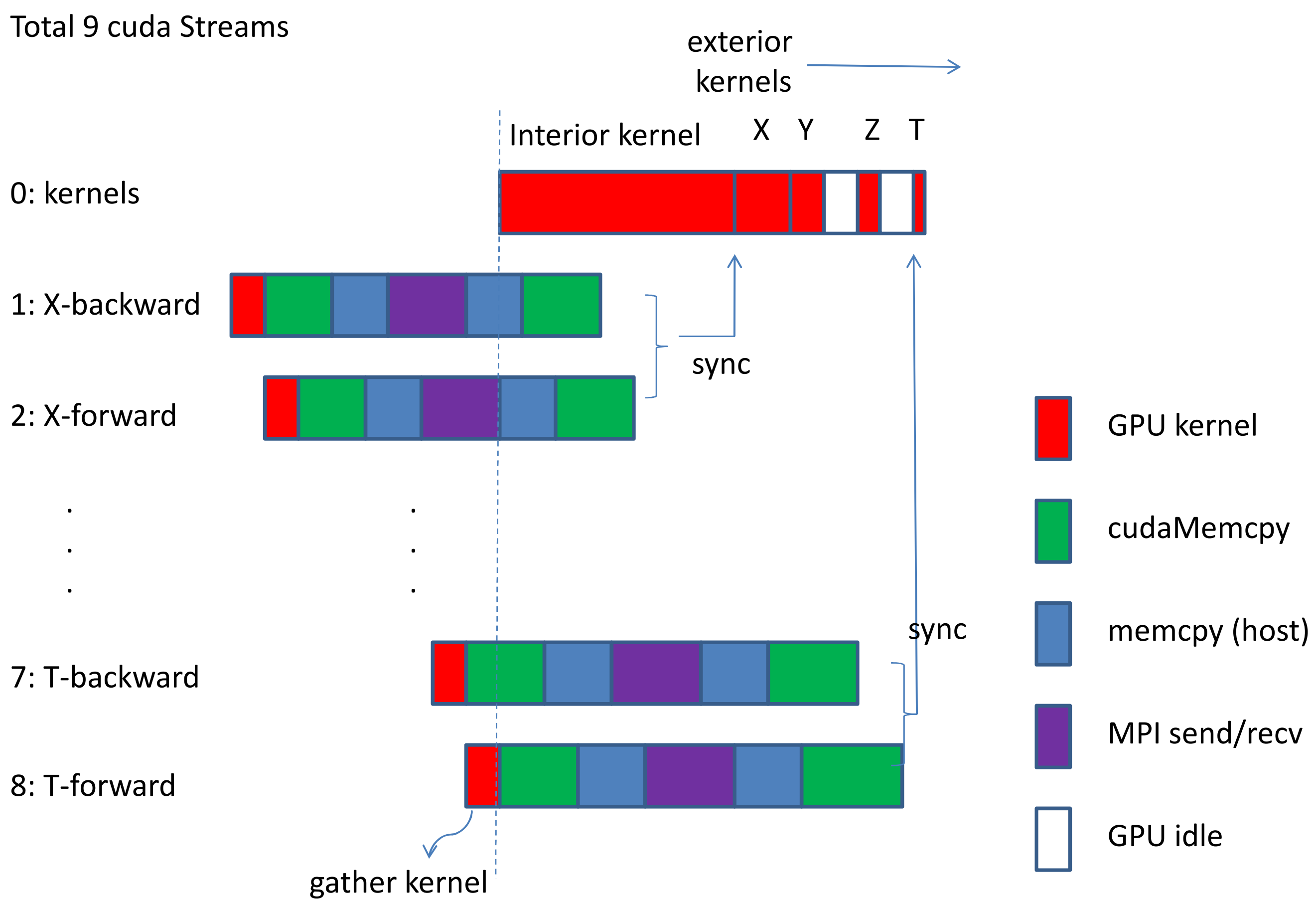


Future Directions - Latency

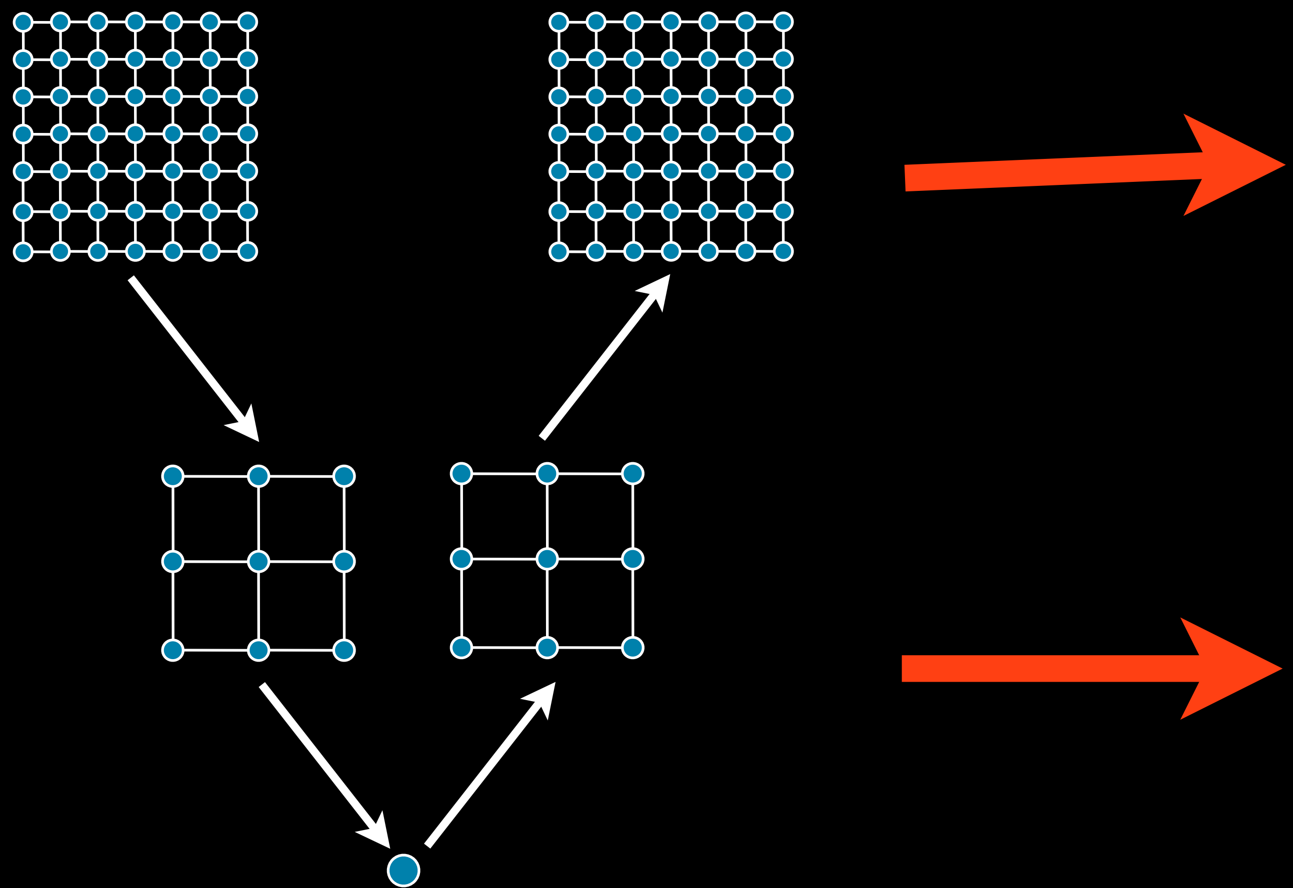
- Global sums are bad
  - Global synchronizations
  - Performance fluctuations
- New algorithms are required
  - S-step CG / BiCGstab, etc.
  - E.g., Pipeline CG vs. Naive
- One-sided communication
  - MPI-3 expands one-sided communications
  - Cray Gemini has hardware support
  - Asynchronous algorithms?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

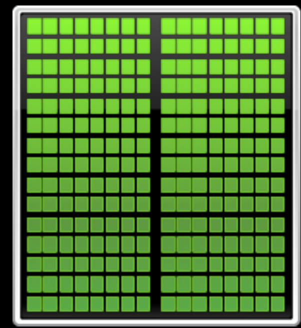


Multi-dimensional Communications Pipeline



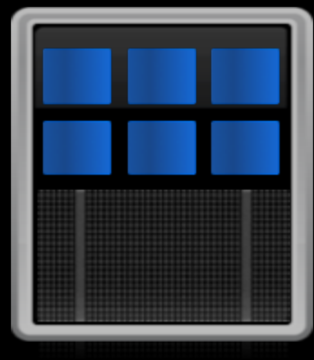
Hierarchical algorithms on heterogeneous architectures





GPU

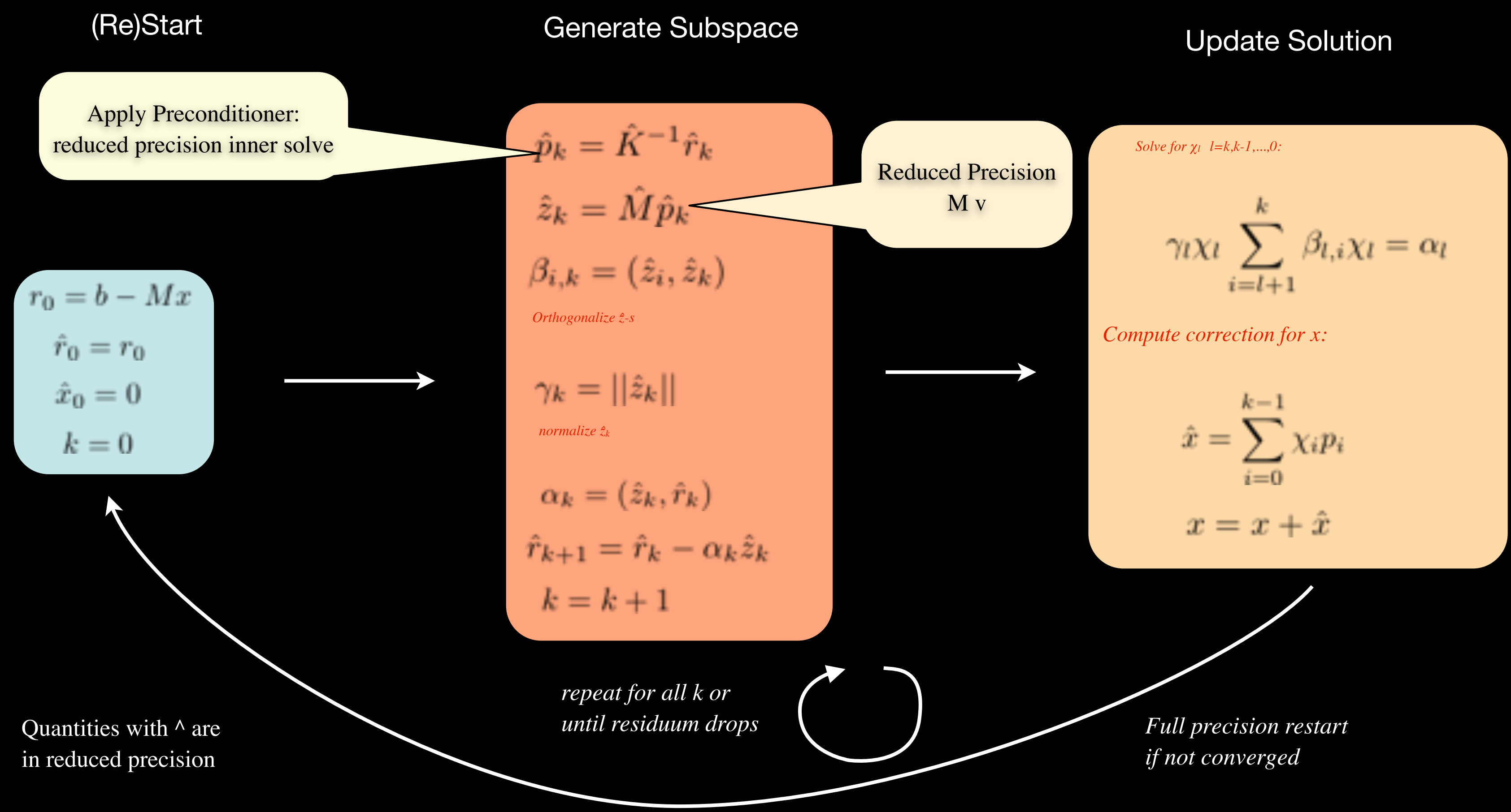
Thousands of cores
for parallel processing



CPU

Few Cores optimized
for serial work

Domain Decomposition



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
```

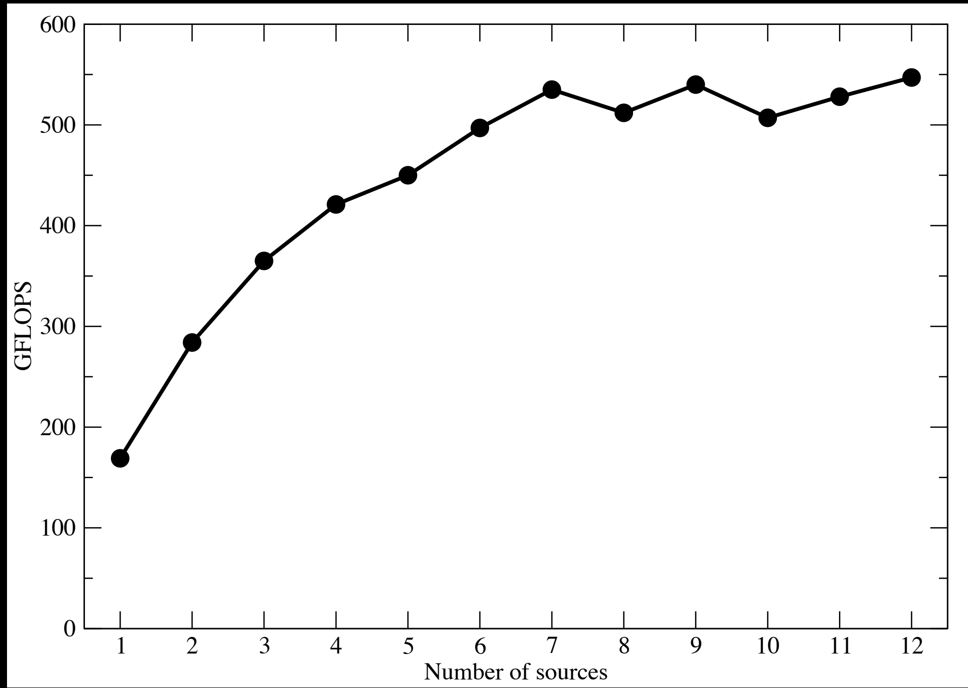
- Here MyKernelWrapper inherits from Tunable and optionally overloads various virtual member functions (next slide).
- Wrapping related kernels in a class hierarchy is often useful anyway, independent of tuning.

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.

Future Directions - Locality

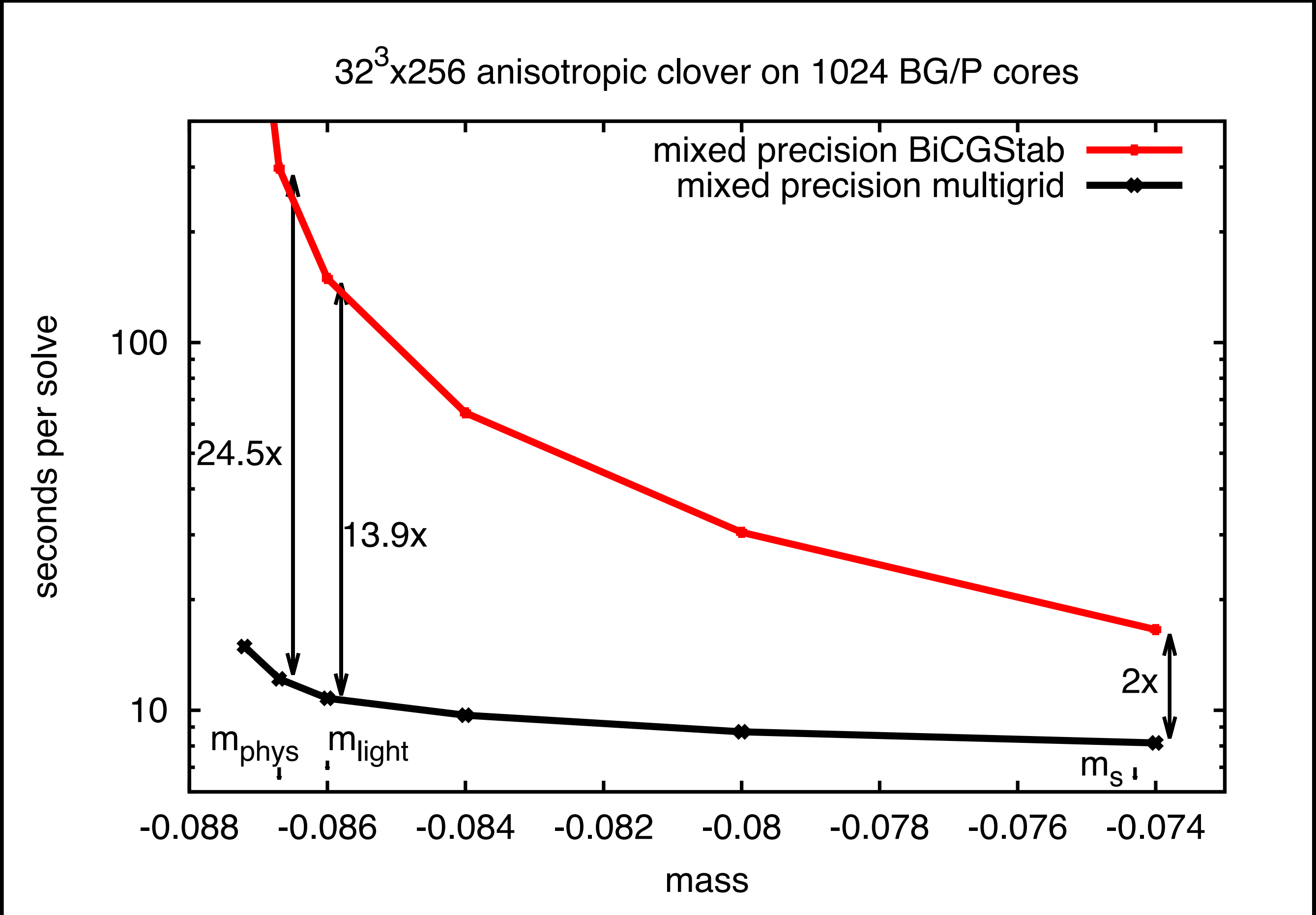
- Where locality does not exist, let's create it
  - E.g., Multi-source solvers
  - Staggered Dslash performance, K20X
  - Transform a memory-bound into a cache-bound problem
  - Entire solver will remain bandwidth bound



Future Directions - 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

Adaptive Multigrid



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

QUDA Low-Level Interface (in development)

- Possible strawman under consideration

```
lat = QUDA_new_lattice(dims, ndim, lat_param);
u = QUDA_new_link_field(lat, gauge_param);
source = QUDA_new_site_field(lat, spinor_param);
solution = QUDA_new_site_field(lat, spinor_param);
QUDA_load_link_field(u, host_u, gauge_order);
QUDA_load_site_field(source, host_source, spinor_order);
QUDA_solve(solution, source, u, solver);
QUDA_save_site_field(solution, host_solution, spinor_order);
QUDA_destroy_site_field(source);
etc...
```

- Here, src, sol, etc. are opaque objects that know about the GPU
- Allows the user to easily maintain data residency
- Users can easily provide their own kernels
- High-level interface becomes a compatibility layer built on top