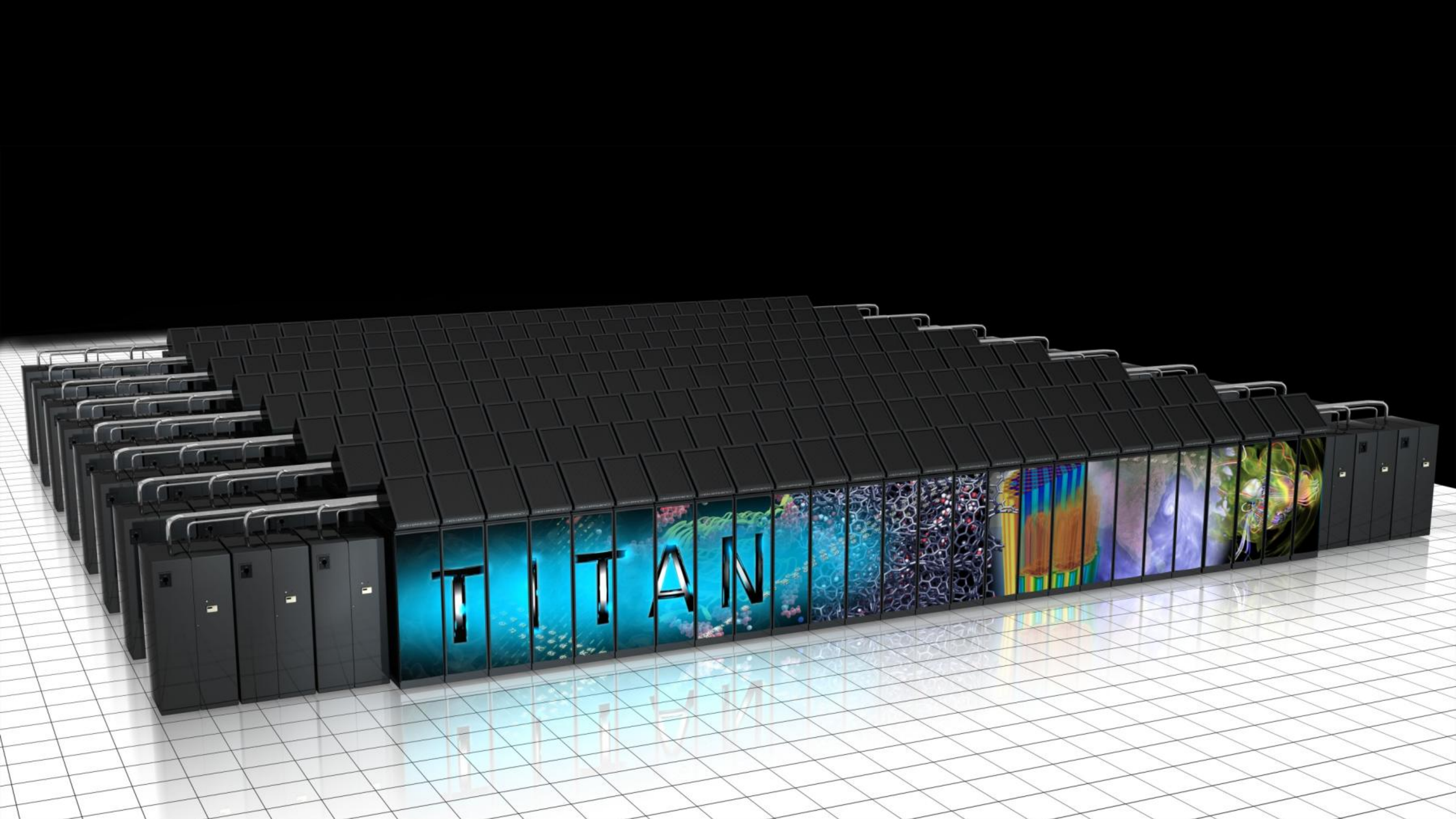
ACCELERATING LATTICE QCD MULTIGRID ON GPUS USING FINE-GRAINED PARALLELIZATION

Kate Clark, July 28th 2016

CONTENTS

QUDA

- “QCD on CUDA” – http://lattice.github.com/quda (open source, BSD license)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, 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
- 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)
  - Multi-source solvers
  - Multigrid solvers for optimal convergence
- A research tool for how to reach the exascale


TITAN

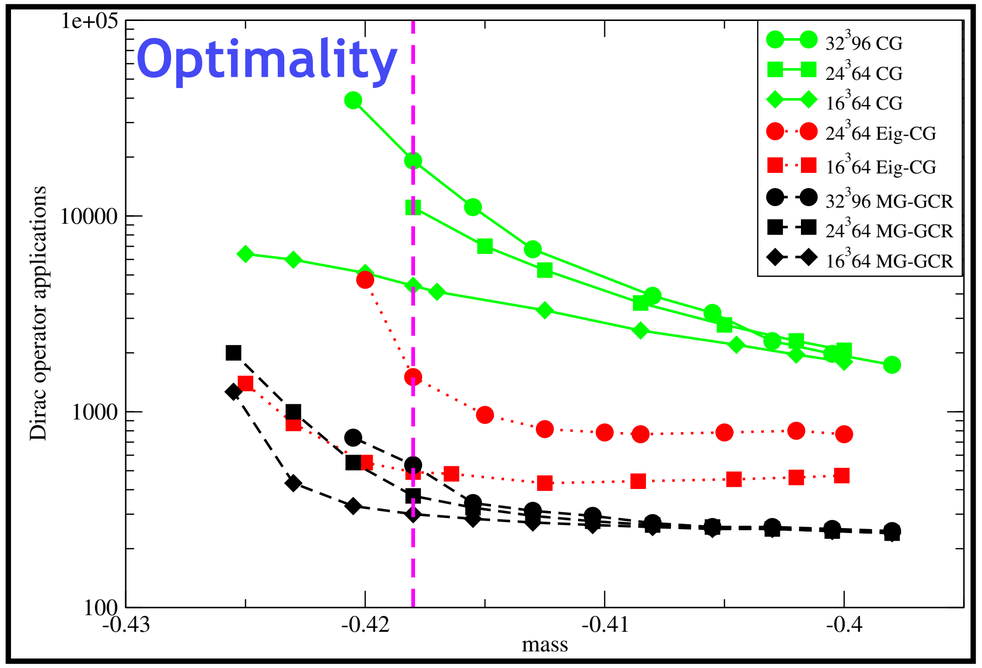
QUDA CONTRIBUTORS

(multigrid collaborators in green)

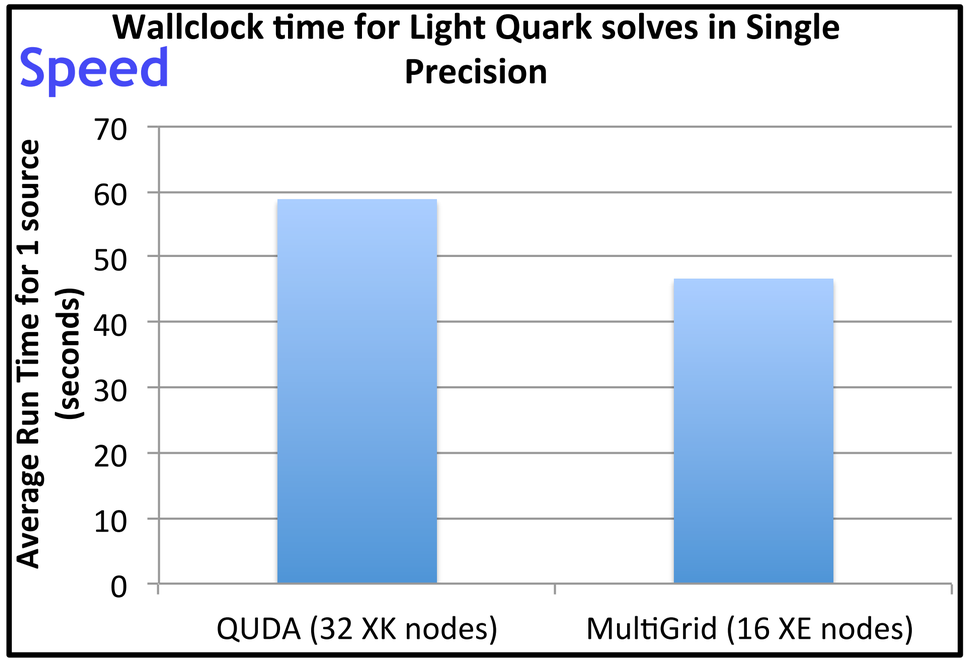
Ron Babich (NVIDIA)
Michael Baldhauf (Regensburg)
Kip Barros (LANL)
Rich Brower (Boston University)
Nuno Cardoso (NCSA)
Kate Clark (NVIDIA)
Michael Cheng (Boston University)
Carleton DeTar (Utah University)
Justin Foley (Utah -> NIH)
Joel Giedt (Rensselaer Polytechnic Institute)
Arjun Gambhir (William and Mary)
Steve Gottlieb (Indiana University)
Dean Howarth (Rensselaer Polytechnic Institute)
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)

MULTIGRID FOR QCD

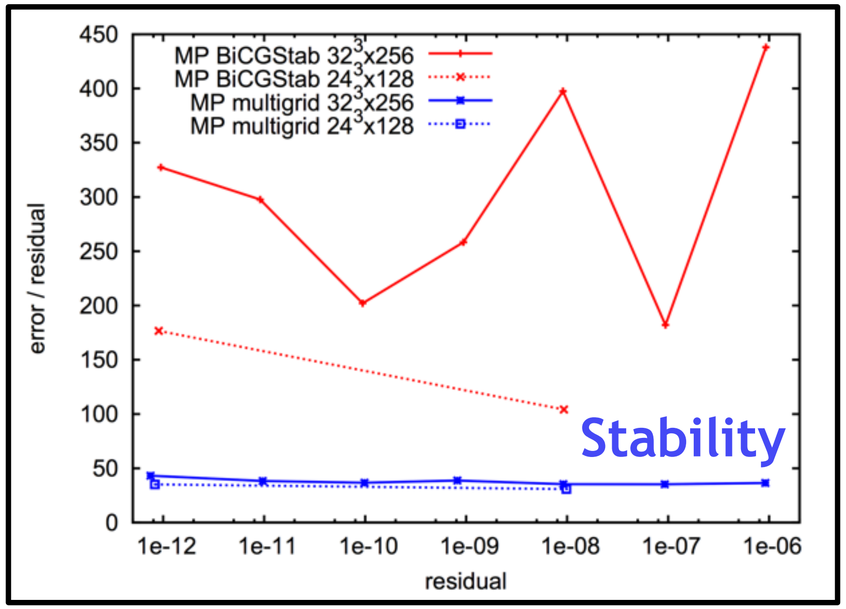
WHY MULTIGRID?



Babich *et al* 2010



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



Osborn *et al* 2010

NVIDIA

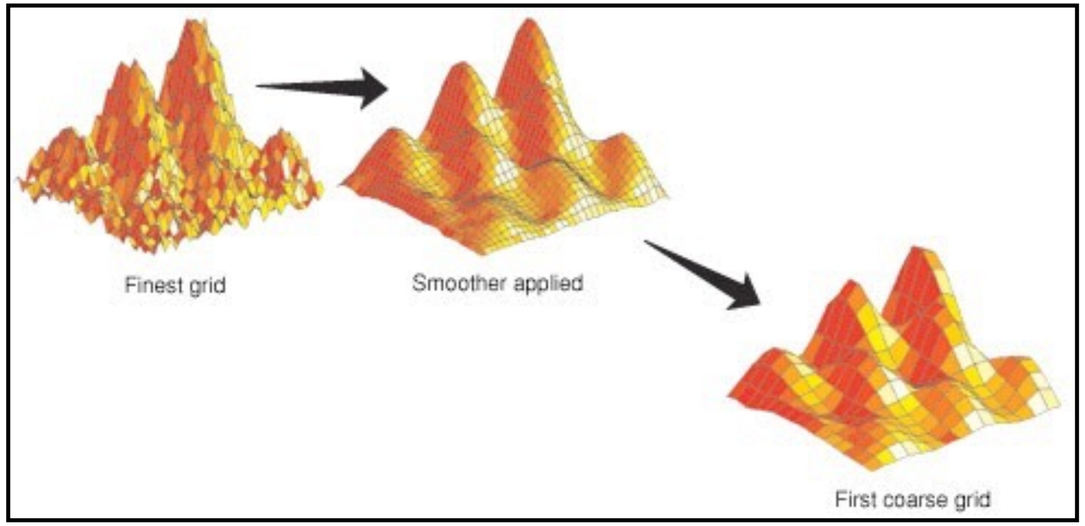
ADAPTIVE GEOMETRIC MULTIGRID

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

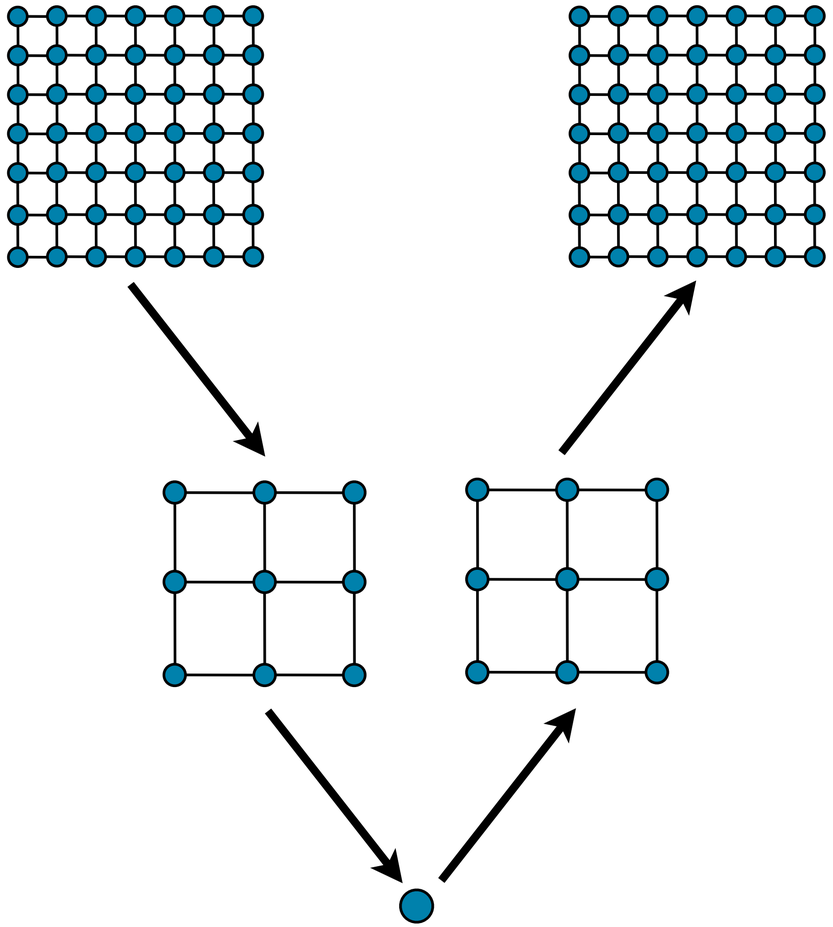


Falgout

Babich *et al* 2010

MULTIGRID ON GPUS

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

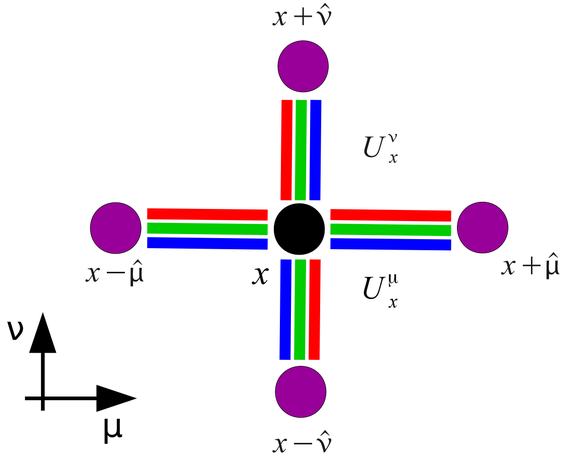
Coarse grids are worst possible scenario

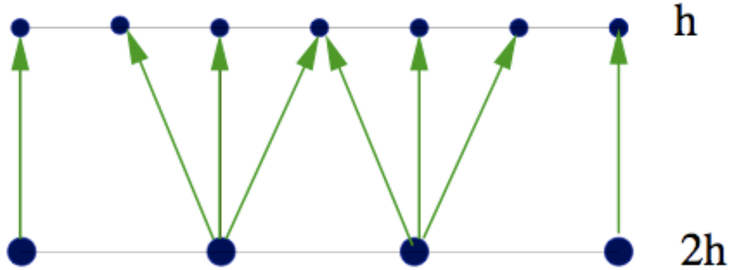
- More cores than degrees of freedom
- Increasingly serial and latency bound
- Little's law (bytes = bandwidth * latency)
- Amdahl's law limiter

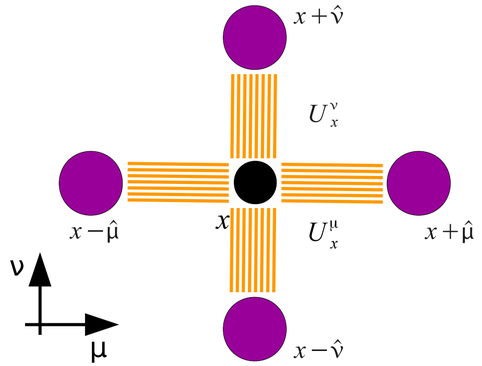
Multigrid exposes many of the problems expected at the Exascale

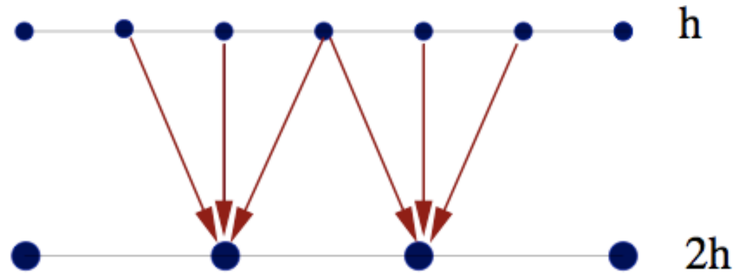
INGREDIENTS FOR PARALLEL ADAPTIVE MULTIGRID

- Multigrid setup
  - Block orthogonalization of null space vectors
  - Batched QR decomposition
- Smoothing (relaxation on a given grid)
  - Repurpose existing solvers
- 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
  - Need optimal coarse-grid operator

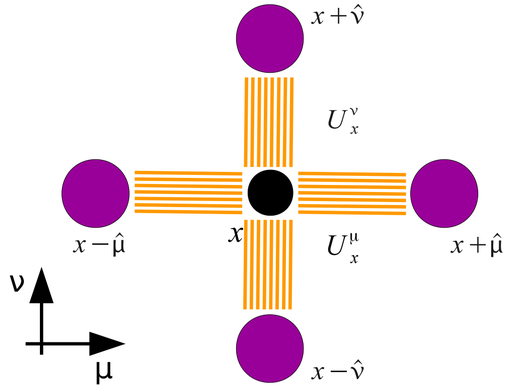








COARSE GRID OPERATOR

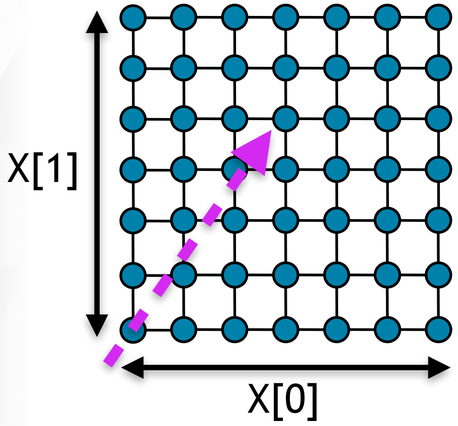


- Coarse operator looks like a Dirac operator (many more colors)
  - Link matrices have dimension $2N_v$ x $2N_v$ (e.g., 48 x 48)

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

- Fine vs. Coarse grid parallelization
  - Fine grid operator has plenty of grid-level parallelism
    - E.g., 16x16x16x16 = 65536 lattice sites
  - Coarse grid operator has diminishing grid-level parallelism
    - first coarse grid 4x4x4x4= 256 lattice sites
    - second coarse grid 2x2x2x2 = 16 lattice sites
- Current GPUs have up to 3840 processing cores
- Need to consider finer-grained parallelization
  - Increase parallelism to use all GPU resources
  - Load balancing

SOURCE OF PARALLELISM



1. Grid parallelism

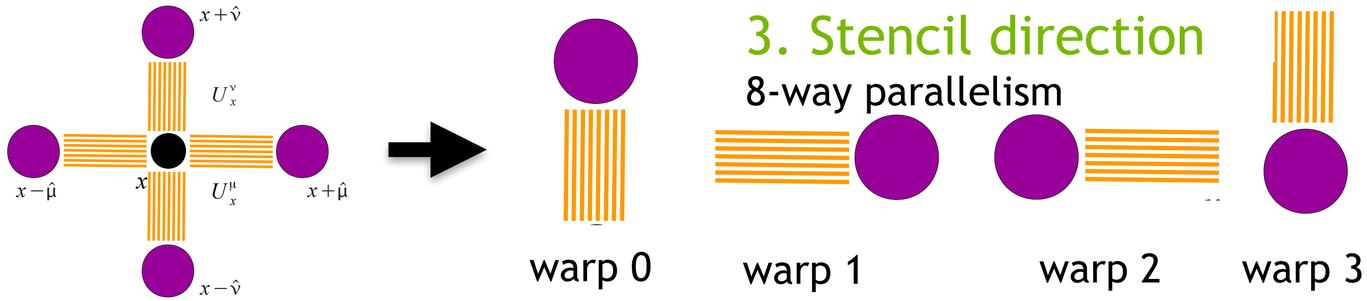
Volume of threads

thread y index

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} += \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

2. Link matrix-vector partitioning

2 N_{vec}-way parallelism (spin * color)



3. Stencil direction

8-way parallelism

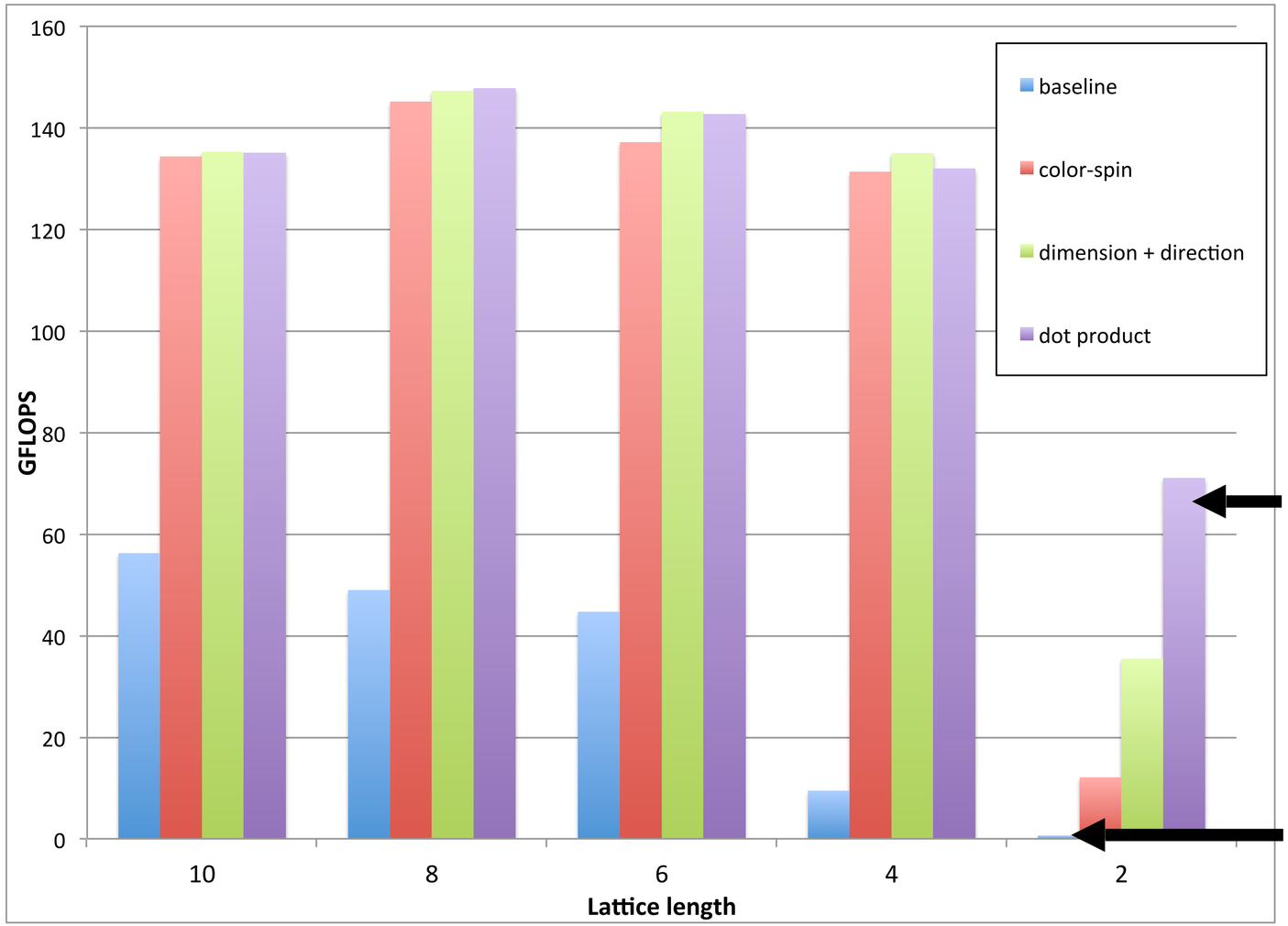
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \Rightarrow \begin{pmatrix} a_{00} & a_{01} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} + \begin{pmatrix} a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_2 \\ b_3 \end{pmatrix}$$

4. Dot-product partitioning

4-way parallelism

COARSE GRID OPERATOR PERFORMANCE

Tesla K20X (Titan), FP32, N_{vec} = 24



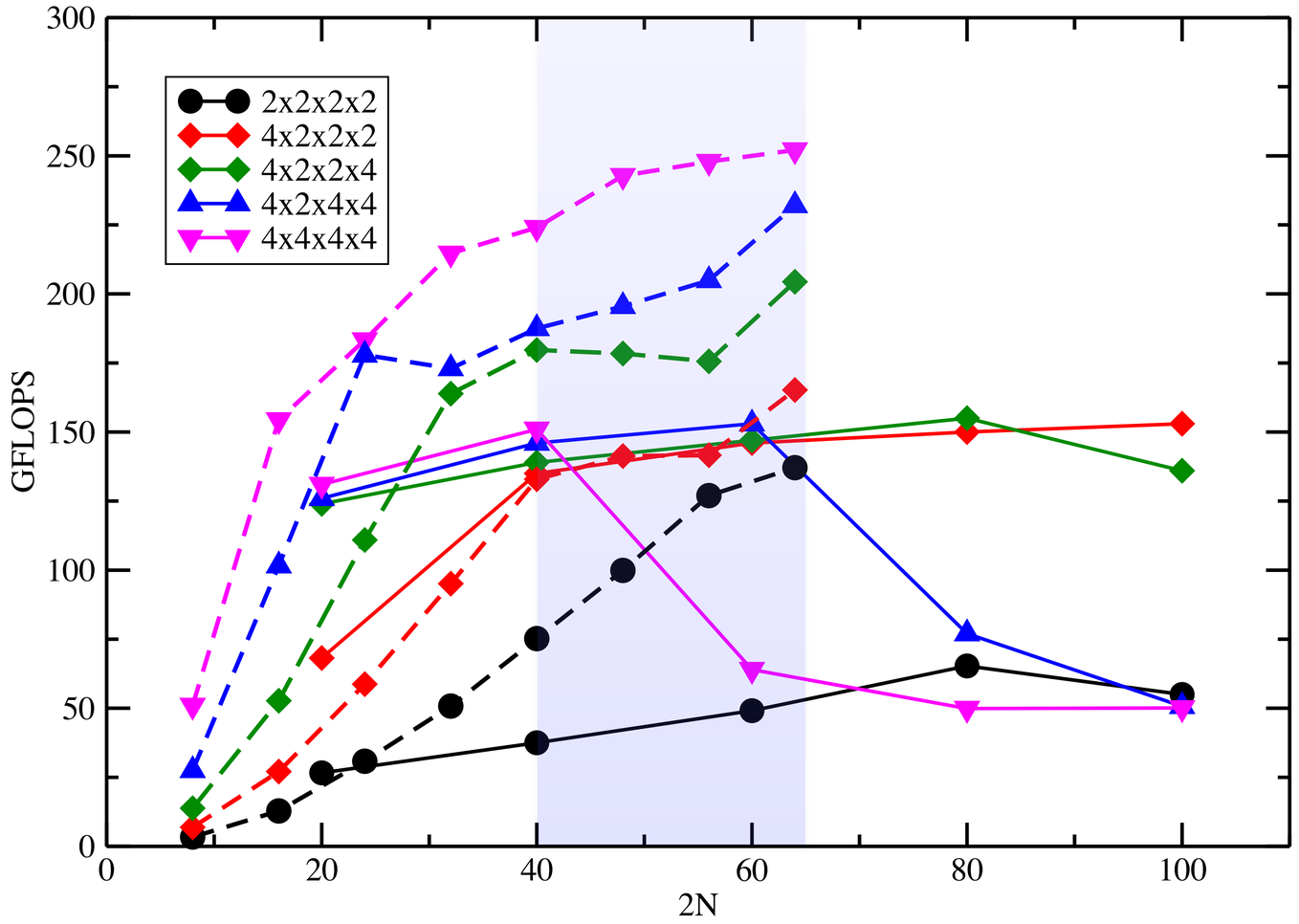
24,576-way parallel

16-way parallel

NVIDIA

COARSE GRID OPERATOR PERFORMANCE

8-core Haswell 2.4 GHz (solid line) vs M6000 (dashed lined), FP32



- 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

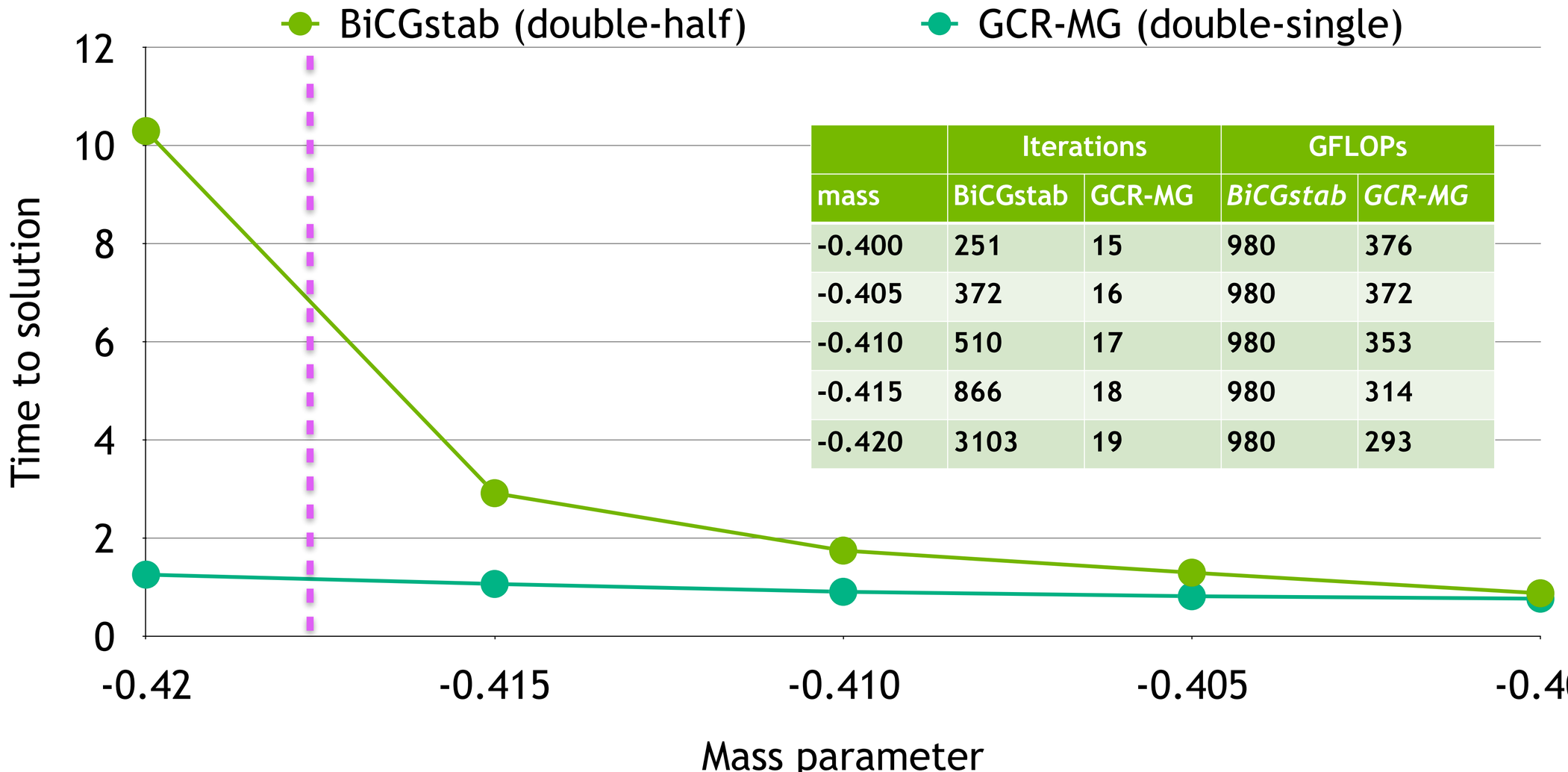
MULTIGRID VERSUS BICGSTAB

- Compare MG against the state-of-the-art traditional Krylov solver on GPU
  - BiCGstab in double/half precision with reliable updates
  - 12/8 reconstruct
  - Symmetric red-black preconditioning $\hat{M}_{ee} = 1_{ee} - \kappa^2 A_{ee}^{-1} D_{eo} A_{oo}^{-1} D_{oe}$

- Adaptive Multigrid algorithm
  - GCR outer solver wraps 3-level MG preconditioner
  - GCR restarts done in double, everything else in single
  - 24 null-space vectors on fine grid
  - Minimum Residual smoother
  - Symmetric red-black preconditioning on each level
  - GCR coarse-grid solver

MULTIGRID VERSUS BICGSTAB

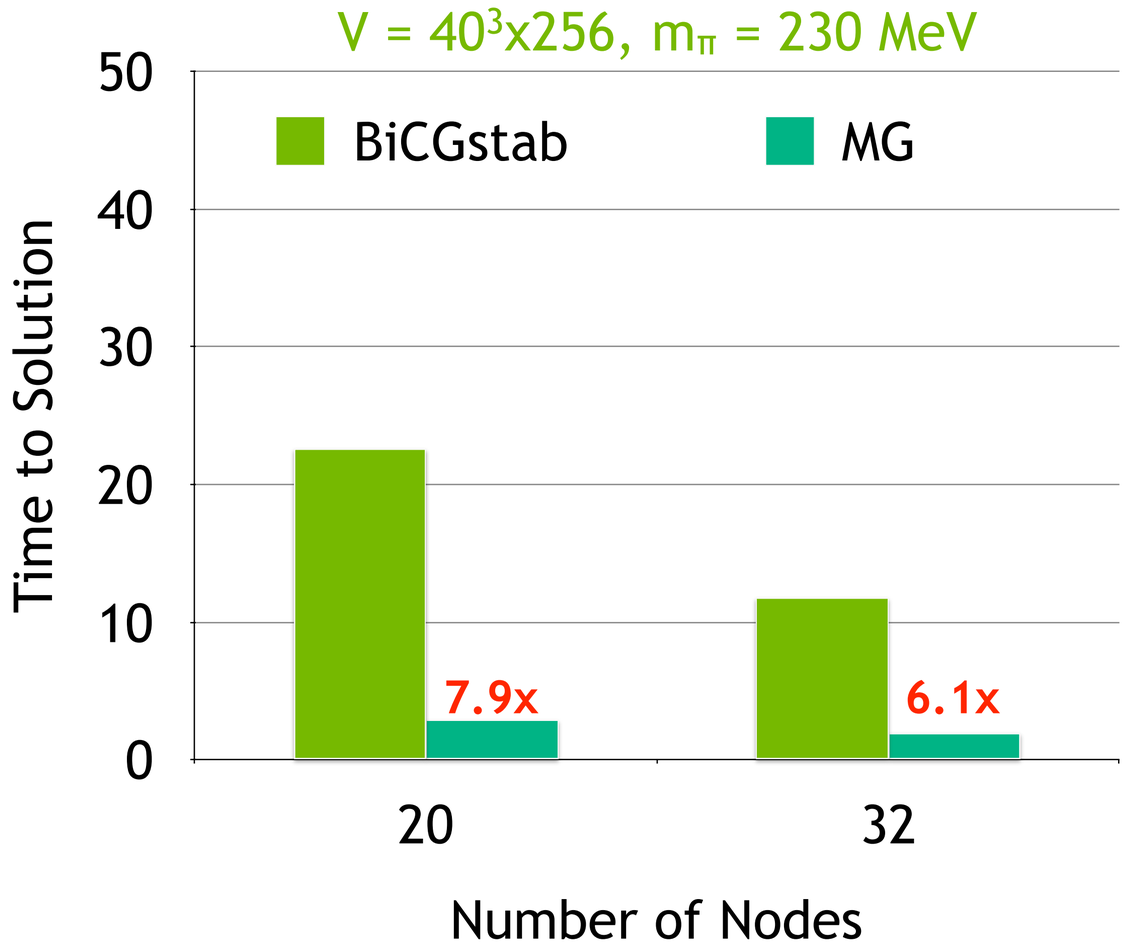
Wilson, V = 24^3x64, single workstation (3x M6000)

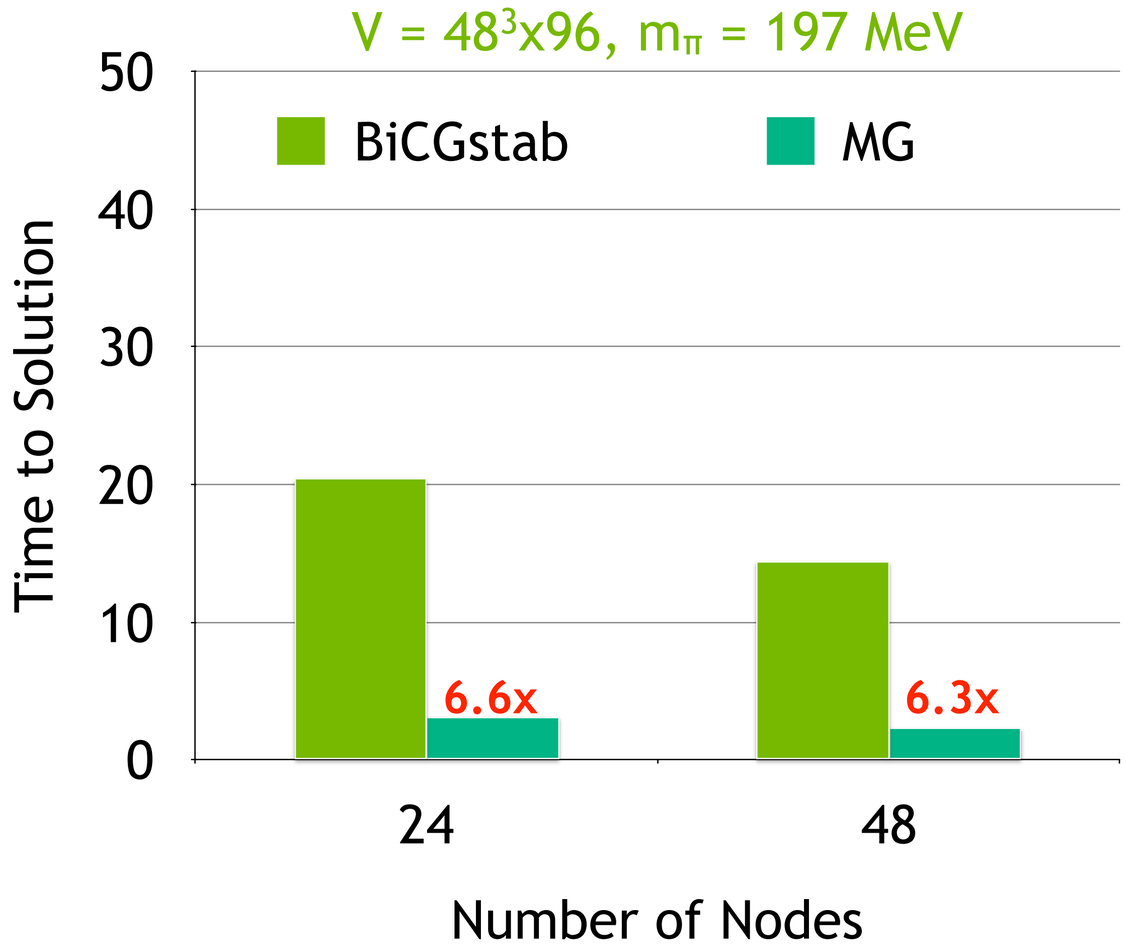


	Iterations		GFLOPs	
mass	BiCGstab	GCR-MG	*BiCGstab*	*GCR-MG*
-0.400	251	15	980	376
-0.405	372	16	980	372
-0.410	510	17	980	353
-0.415	866	18	980	314
-0.420	3103	19	980	293

MULTIGRID VERSUS BICGSTAB

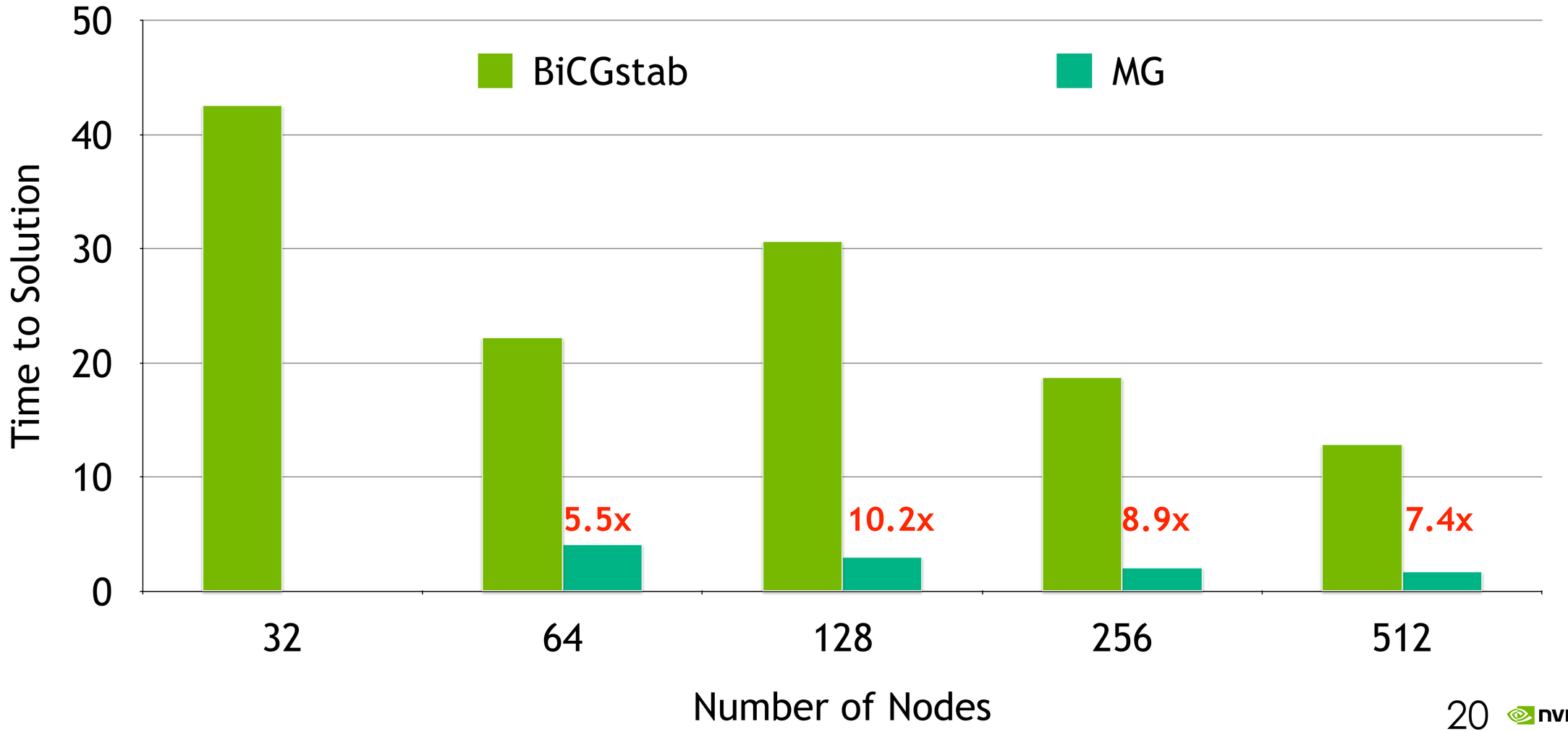
Wilson-clover, Strong scaling on Titan (K20X)





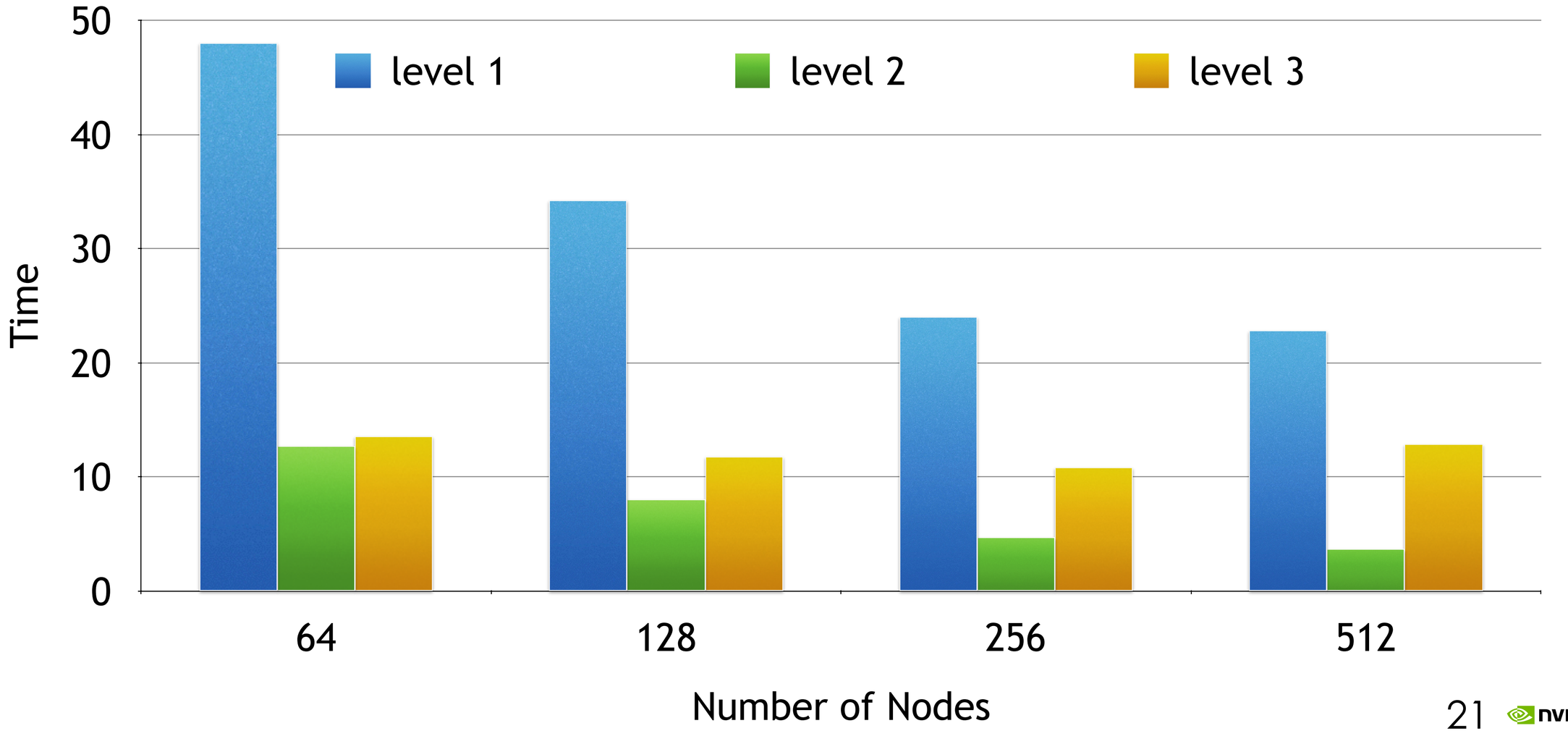
MULTIGRID VERSUS BICGSTAB

Wilson-clover, Strong scaling on Titan (K20X), V = 64^3x128, m_π = 197 MeV



MULTIGRID TIMING BREAKDOWN

Wilson-clover, Strong scaling on Titan (K20X), V = 64^3x128, 12 linear solves

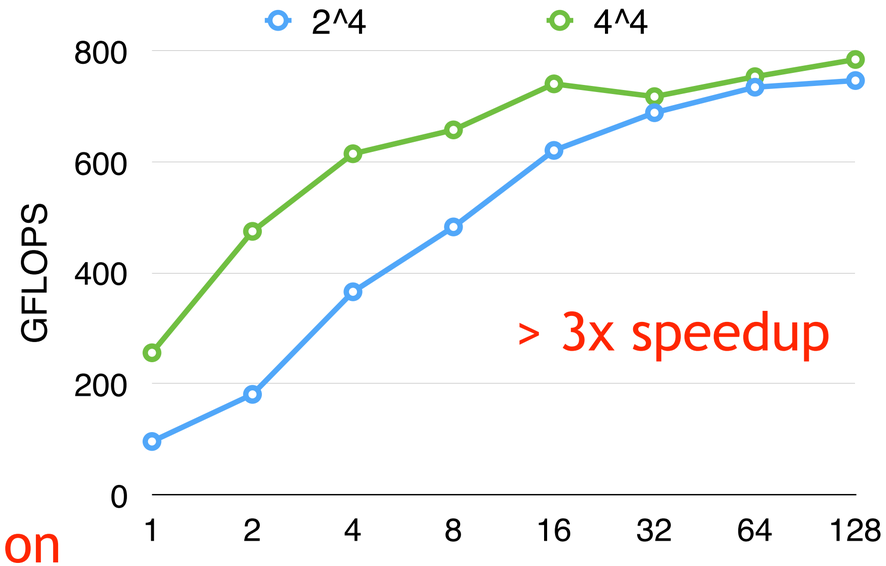


MULTIGRID FUTURE WORK

- Absolute Performance tuning, e.g., half precision on coarse grids
- Strong scaling improvements:
  - Combine with Schwarz preconditioner
  - Accelerate coarse grid solver: CA-GMRES instead of GCR, deflation
  - More flexible coarse grid distribution, e.g., redundant nodes
- Investigate off load of coarse grids to the CPU
  - Use CPU and GPU simultaneously using additive MG
- Full off load of setup phase to GPU - required for HMC

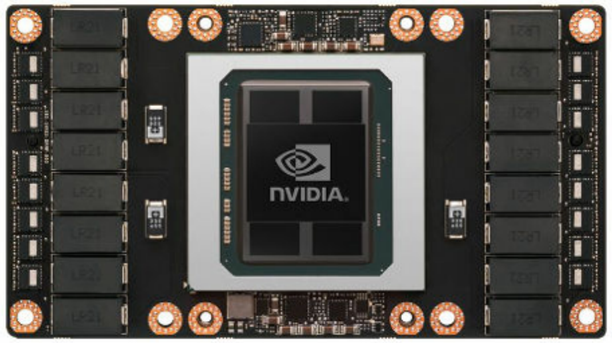
MULTI-SRC SOLVERS

- Multi-src solvers increase locality through link-field reuse
  - see previous talk by M. Wagner
- Multi-grid operators even more so since link matrices are 48x48
  - Coarse Dslash / Prolongator / Restrictor
- Coarsest grids also latency limited
  - Kernel level latency
  - Network latency
- Multi-src solvers are a solution
  - More parallelism
  - Bigger messages



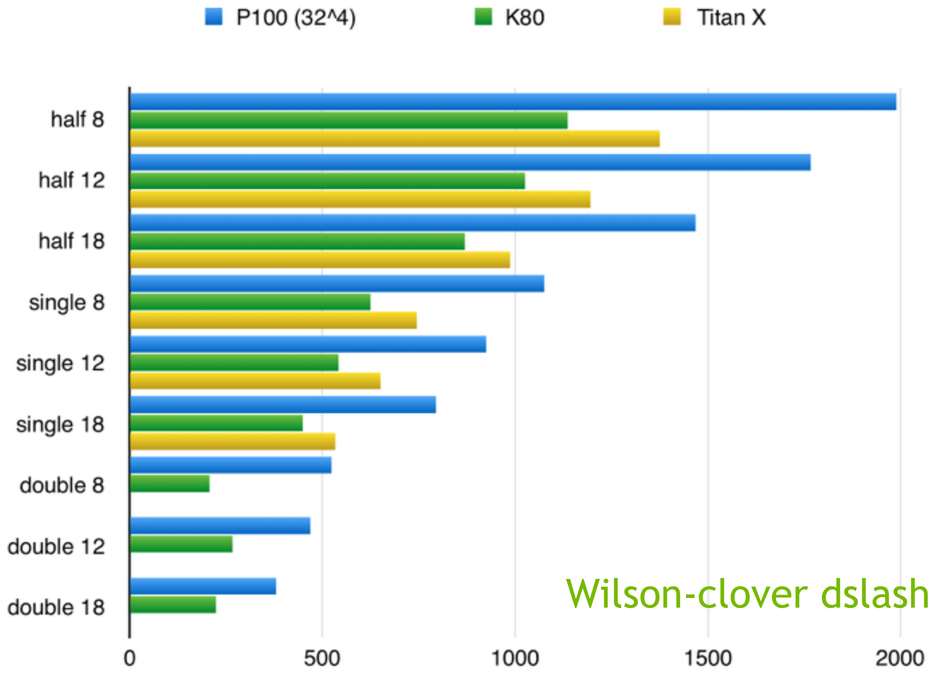
Coarse dslash on M6000 GPU vs #rhs

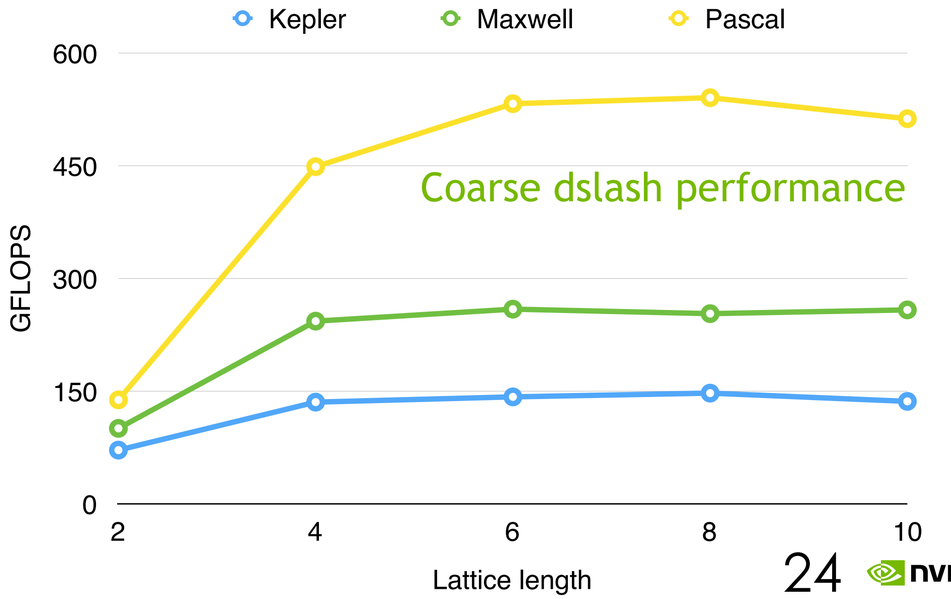
PASCAL



P100

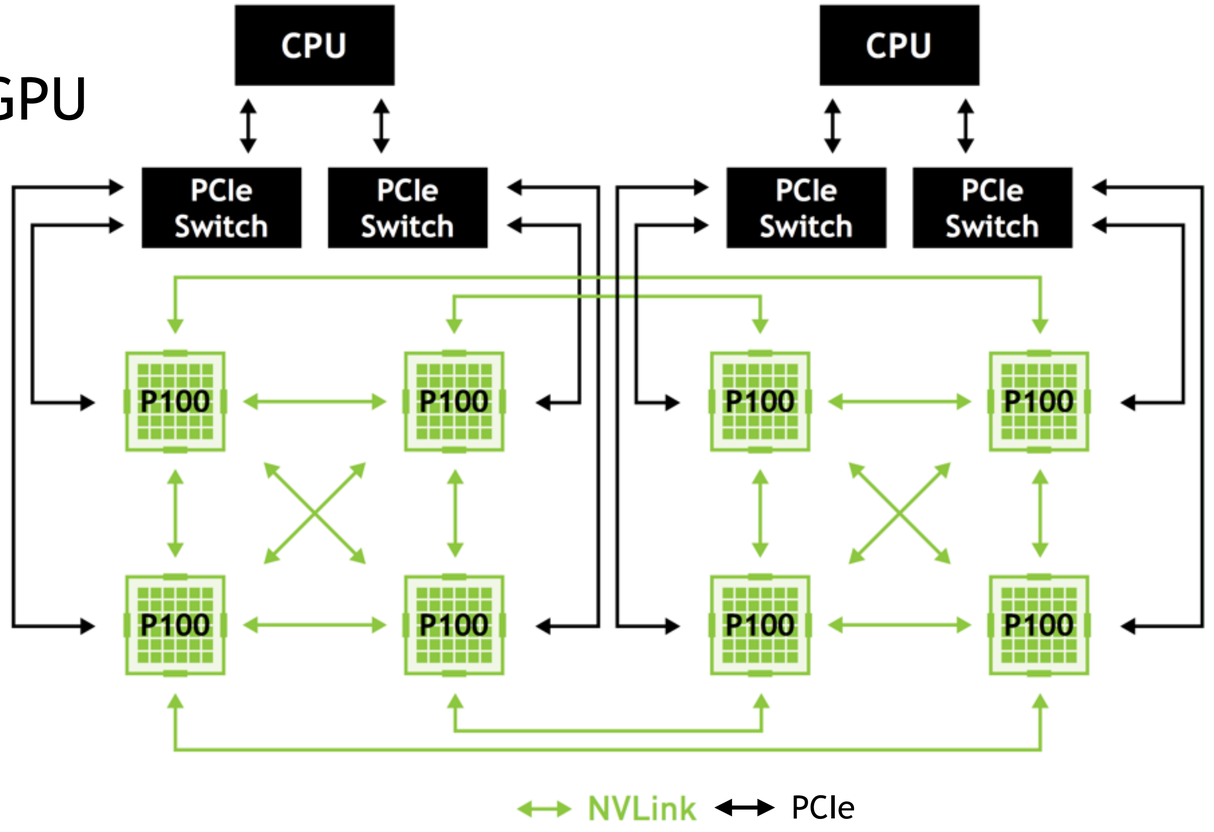
- Newly launched Pascal P100 GPU provides significant performance uplift through stacked HBM memory
- Wilson-clover dslash 3x speedup vs K40
  - double ~500 GFLOPS
  - single ~1000 GFLOPS
  - half ~2000 GFLOPS
- Same kernel from 2008 runs unchanged
  - 5 generations ago
- Similar gains for MG building blocks





DGX-1

- Pascal includes NVLink interconnect technology
  - 4x NVLink connections per GPU
  - 160 GB/s bi-directional bandwidth per GPU
- NVIDIA DGX-1
  - 8x P100 GPUs
  - 2^3 NVLink topology
  - 4x EDR IB (via PCIe switches)
  - 2x Intel Xeon hosts
- 1 DGX-1 equivalent inter-GPU bandwidth to 64 nodes of Titan



COMING SOON

- >10x speedup from MG
  - Lower bound since much more optimization to do
- 6x node-to-node speedup
  - DGX-1 (8x GP100) vs Pi0g (4x K40)
- 3x speedup from multi-src solvers
  - Increased temporal locality from links and increased parallelism for MG
- Expect >100x speedup for analysis workloads versus previous GPU workflow

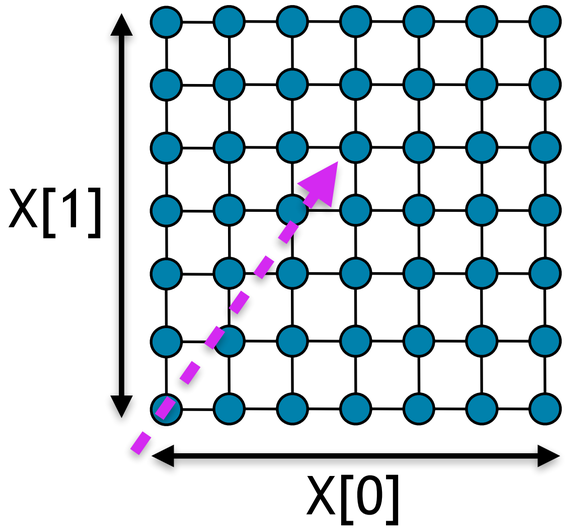
CONCLUSIONS AND OUTLOOK

- Multigrid algorithms LQCD are running well on GPUs
  - In production by multiple groups
  - Wilson, Wilson-clover, twisted-mass, twisted-clover all supported
  - Staggered in progress (see next talk by E. Weinberg)
  - 5-d operators when I get a chance…
- Up to 10x speedup
  - Fine-grained parallelization was key
  - To be presented / published at SC16
- Lots more to come
  - Combine with multi-right-hand side solver and Pascal architecture

GRID PARALLELISM

Thread x dimension maps to location on the grid

```
__device__ void grid_idx(int x[], const int X[])
{
  // X[] holds the local lattice dimension
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  int za = (idx / X[0]);
  int zb =  (za / X[1]);
  x[1] = za - zb * X[1];
  x[3] = (zb / X[2]);
  x[2] = zb - x[3] * X[2];
  x[0] = idx - za * X[0];
  // x[] now holds the thread coordinates
}
```



MATRIX-VECTOR PARALLELISM

Each stencil application is a sum of matrix-vector products

Parallelize over output vector indices (parallelization over color and spin)

Thread y dimension maps to vector indices

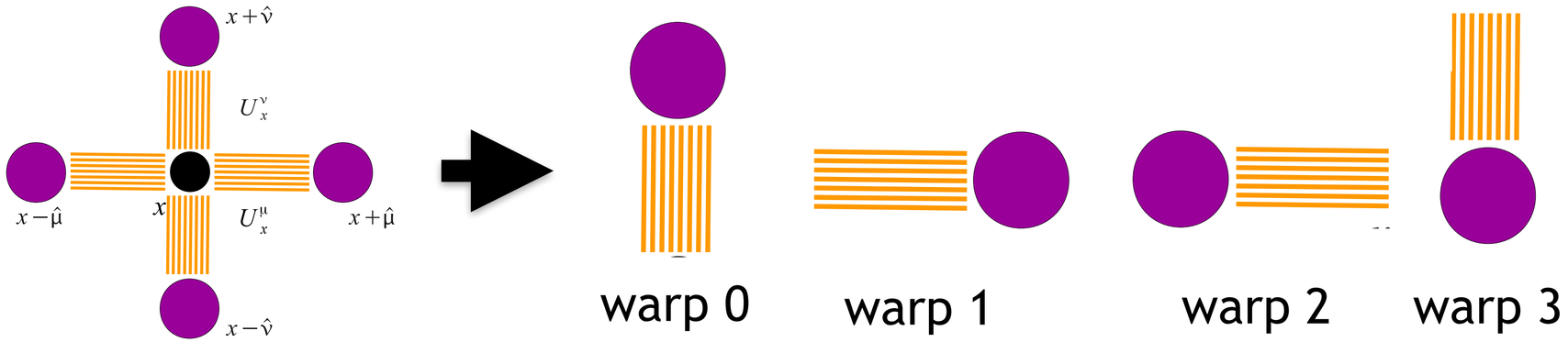
Up to 2 x N_v more parallelism

thread y index

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} += \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

```
template<int Nv>
__device__ void color_spin_idx(int &s, int &c)
{
  int yIdx = blockDim.y*blockIdx.y + threadIdx.y;
  int s = yIdx / Nv;
  int c = yIdx % Nv;
  // s is now spin index for this thread
  // c is now color index for this thread
}
```

STENCIL DIRECTION PARALLELSIM



Partition computation over stencil direction and dimension onto different threads

Write result to shared memory

Synchronize

dim=0/dir=0 threads combine and write out result

Introduces up to 8x more parallelism

```
__device__ void dim_dir_idx(int &dim, int &dir)
{
  int zIdx = blockDim.z*blockIdx.z + threadIdx.z;
  int dir = zIdx % 2;
  int dim = zIdx / 2;
  // dir is now the fwd/back direction for this thread
  // dim is now the dim for this thread
}
```

DOT PRODUCT PARALLELIZATION I

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \Rightarrow \begin{pmatrix} a_{00} & a_{01} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} + \begin{pmatrix} a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_2 \\ b_3 \end{pmatrix}$$

Partition dot product between threads in the same warp

Use warp shuffle for final result

Useful when not enough grid parallelism to fill a warp

```
const int warp_size = 32; // warp size
const int n_split = 4; // four-way warp split
const int grid_points = warp_size/n_split; // grid points per warp
complex<real> sum = 0.0;
for (int i=0; i<N; i+=n_split)
  sum += a[i] * b[i];

// cascading reduction
for (int offset = warp_size/2; offset >= grid_points; offset /= 2)
  sum += __shfl_down(sum, offset);
// first grid_points threads now hold desired result
```

DOT PRODUCT PARALLELIZATION II

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \Rightarrow \begin{pmatrix} a_{00} & a_{01} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} + \begin{pmatrix} a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_2 \\ b_3 \end{pmatrix}$$

Partition dot product computation within a thread

Hide dependent arithmetic latency within a thread

More important for Kepler then Maxwell / Pascal

```
const int n_ilp = 2; // two-way ILP
complex<real> sum[n_ilp] = { };
for (int i=0; i<N; i+=n_ilp)
  for (int j=0; j<n_ilp; j++)
    sum[j] += a[i+j] * b[i+j];

complex<real> total = static_cast<real>(0.0);
for (int j=0; j<n_ilp; j++) total += sum[j];
```

← Degree of ILP exposed

← Multiple computations with no dependencies

← Compute final result

ERROR REDUCTION AND VARIANCE

$V = 40^3 x 256$, $m_\pi = 230$ MeV