Lattice QCD and GPU Computing

Bálint Joó, Jefferson Lab

Extreme Computing and Its Implications for the Nuclear Physics/Applied Mathematics/Computer Science Interface

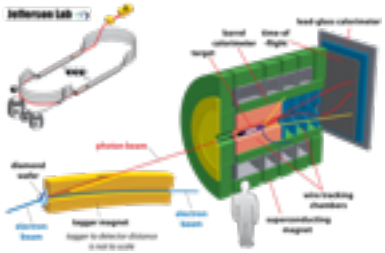
Seattle, July 2011

Contents

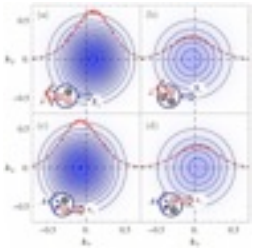
- Motivation: Lattice QCD Calculations
- The QUDA library for LQCD on GPUs
  - Capacity Use
  - Capability Use
    - Domain Decomposition
- LQCD Programming, Frameworks and the GPU
  - Re-engineering QDP++
- Musings on the Future
- Conclusions

QCD In Nuclear Physics

- Can QCD predict the spectrum of hadrons ?
  - what is the role of the gluons?
  - what about exotic matter?

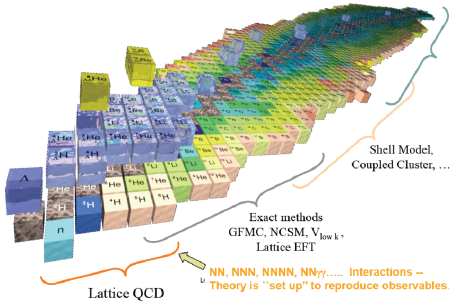


- How do quarks and gluons make nucleons?
  - what are the distribution of quarks, gluons, spin, etc ?

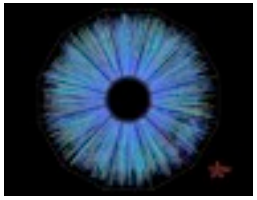


Hägler, Musch, Negele, Schäfer, EPL ***88*** *61001*

- QCD must explain nuclear interactions
  - ab initio calculations for simple systems
  - bridges to higher level effective theories

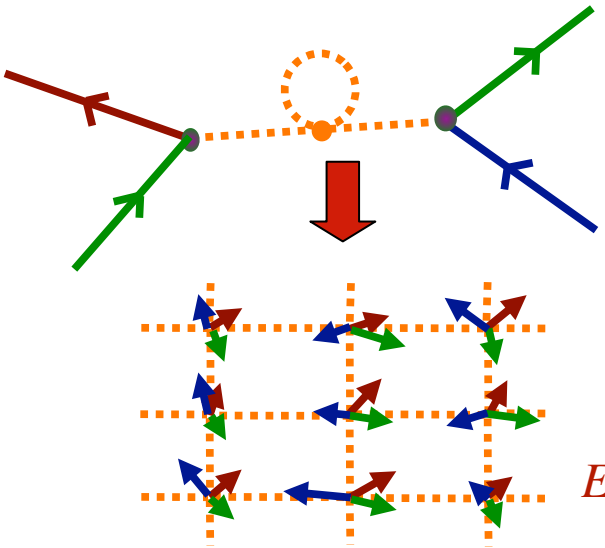


- QCD phase structure, equation of state
  - input to higher level models (e.g hydrodynamics)
  - experiments (e.g. RHIC), astrophysics (early universe)



Enter Lattice QCD...

- Lattice QCD is the only known model independent, non-perturbative technique for carrying out QCD calculations.
  - Move to Euclidean Space, Replace space-time with lattice
  - Move from Lie Algebra su(3) to group SU(3) for gluons
  - Gluons live on links (Wilson Lines) as SU(3) matrices
  - Quarks live on sites as 3-vectors.
  - Produce Lattice Versions of the Action

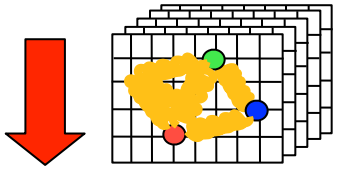


$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}A \; \mathcal{D}\bar{\psi} \; \mathcal{D}\psi \; \mathcal{O} \; e^{-S(A,\bar{\psi},\psi)}$$

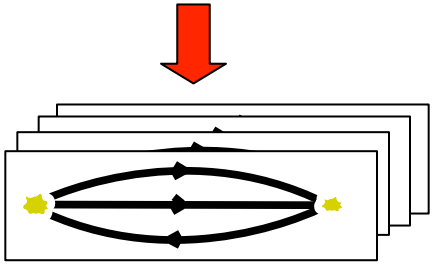
$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \prod_{\text{all links}} dU \prod_{\text{all sites}} d[\bar{\psi},\psi] \; \mathcal{O} \; e^{-S(U,\bar{\psi},\psi)}$$

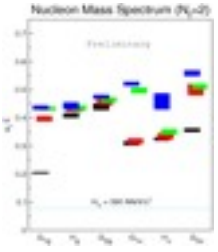
Evaluate Path Integral Using Markov Chain Monte Carlo Method

Large Scale LQCD Simulations Today



- Stage 1: Generate Configurations
  - snapshots of QCD vacuum
  - configurations generated in sequence
  - capability computing needed for large lattices and light quarks
- Stage 2a: Compute quark propagators
  - task parallelizable (per configuration)
  - capacity workload (but can also use capability h/w)
- Stage 2b: Contract propagators into Correlation Functions
  - determines the physics you'll see
  - complicated multi-index tensor contractions





- Stage 3: Extract Physics
  - on workstations, small cluster partitions

Solving the Dirac Equation

- Key component of Gauge Generation and Propagator Calculation

Props:

$$Mx = b$$

MD Force Terms:

$$M^\dagger M \; x = b$$

$$\left(M^\dagger M + \sigma_i I\right) x = b$$

- The Dirac Operator M describes interactions of quarks & gluons :
  - Features (Wilson-Clover formulation):
    - dim (M) = $N_c x N_s x V$, $V=32^3 x 256$, $N_c=3$, $N_s=4$ -> dim~100M
    - Complex, Wilson form is J-hermitian, ie: $JM=M^\dagger J^\dagger$ ($J=\gamma_5$)
    - NB: $\gamma_5 = \mathrm{diag}(1,-1)$ is maximally indefinite
    - Condition $\sim (1/m_q)(1/a)^5 \sim (1/m_\pi)^2(1/a)^6$
    - Local (nearest neighbor, or next-to-nearest neighbor)

Enter QUDA

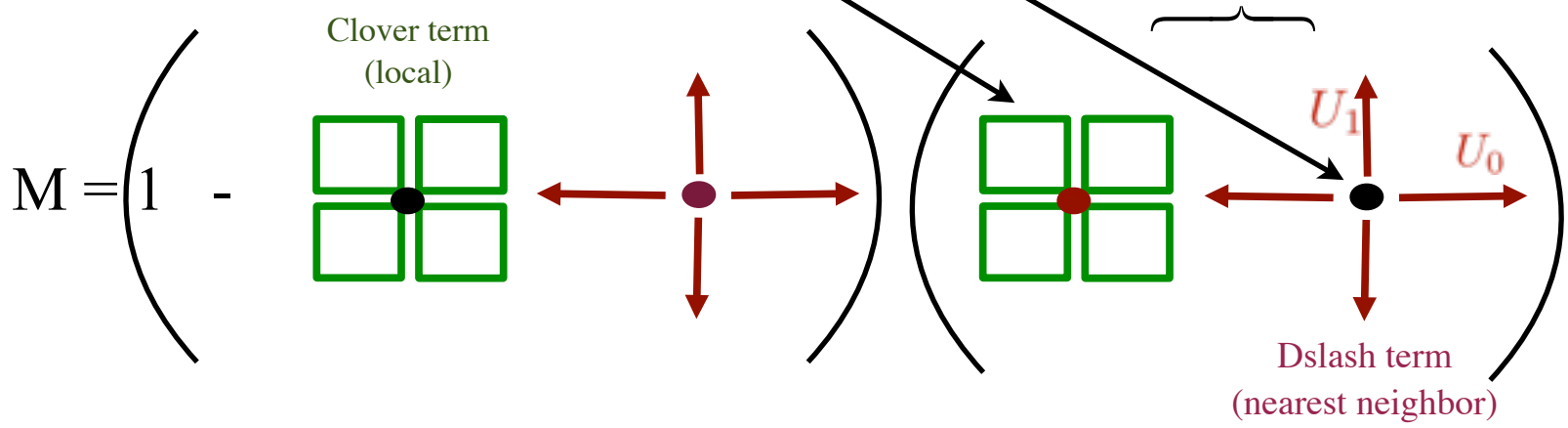
- QUDA is a library of solvers for lattice QCD on CUDA GPUs
  - *Clark, et. al., Comp. Phys. Commun. 181:1517-1528, 2010*
  - Supports: Wilson-Clover, Improved Staggered fermions
  - Domain Wall fermion support is 'in development'
  - 'Standard' Krylov Solvers for QCD: CG(NE), BiCGStab
- Key Optimizations
  - Memory Coalescing Friendly Data Layout
  - Memory Bandwidth reducing 'tricks'
    - Mixed Precision (16 bit, 32 bit, 64 bit) solvers
    - Field Compression
    - Dirac Basis (save loading half of t-neighbours)
    - Solve in Axial Gauge (save loading t-links)

The Wilson-Clover Fermion Matrix

After even-odd (red-black) preconditioning (Schur style):

$$M = 1 - A_{oo}^{-1} D_{oe} A_{ee}^{-1} D_{eo}$$

total: 1824 flops,
408 words in + 24 words out
FLOP/Byte: 1.06 (SP), 0.53 (DP)



SU(3) matrix

permutes spin components, flips signs

'get nearest neighbour' from forward μ direction

$$D_{x,y} = \frac{1}{2} \sum_{\mu=0}^{4} U_\mu(x) \otimes (1 - \gamma_\mu) \otimes \delta_{x+\hat{\mu},y} + U_\mu^\dagger(x - \hat{\mu}) \otimes (1 + \gamma_\mu) \otimes \delta_{x-\hat{\mu},y}$$

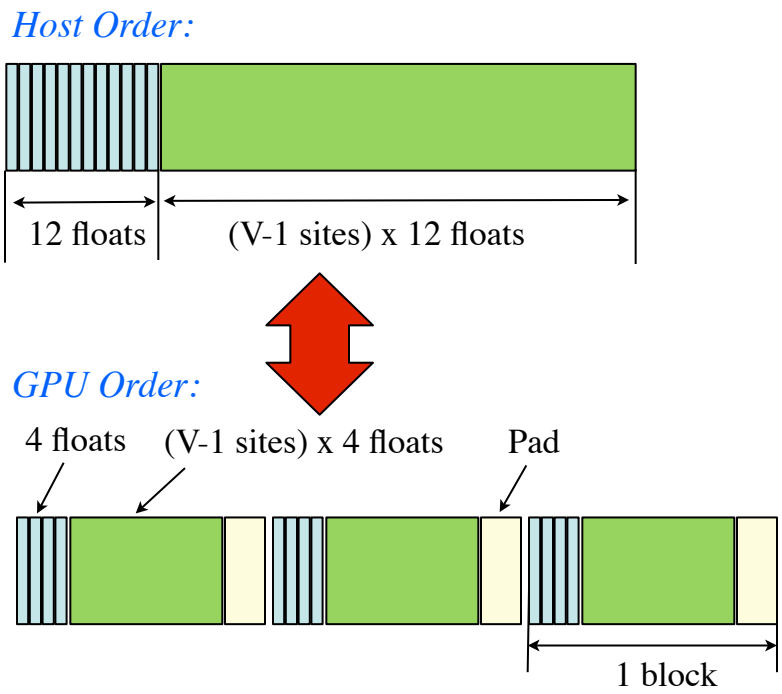
QUDA Tricks: Compression

- Bandwidth reduction through compression
  - Store 3x3 SU(3) matrix as 6 complex numbers, or 8 reals
  - spend 'free' flops to uncompress
    - For DP no compression is best - not enough free flops

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

QUDA Optimizations

- Data Layout tuned for Memory Coalescing
  - 1 thread / lattice site,
  - break up data for site data into chunks (e.g. float4 for SP)



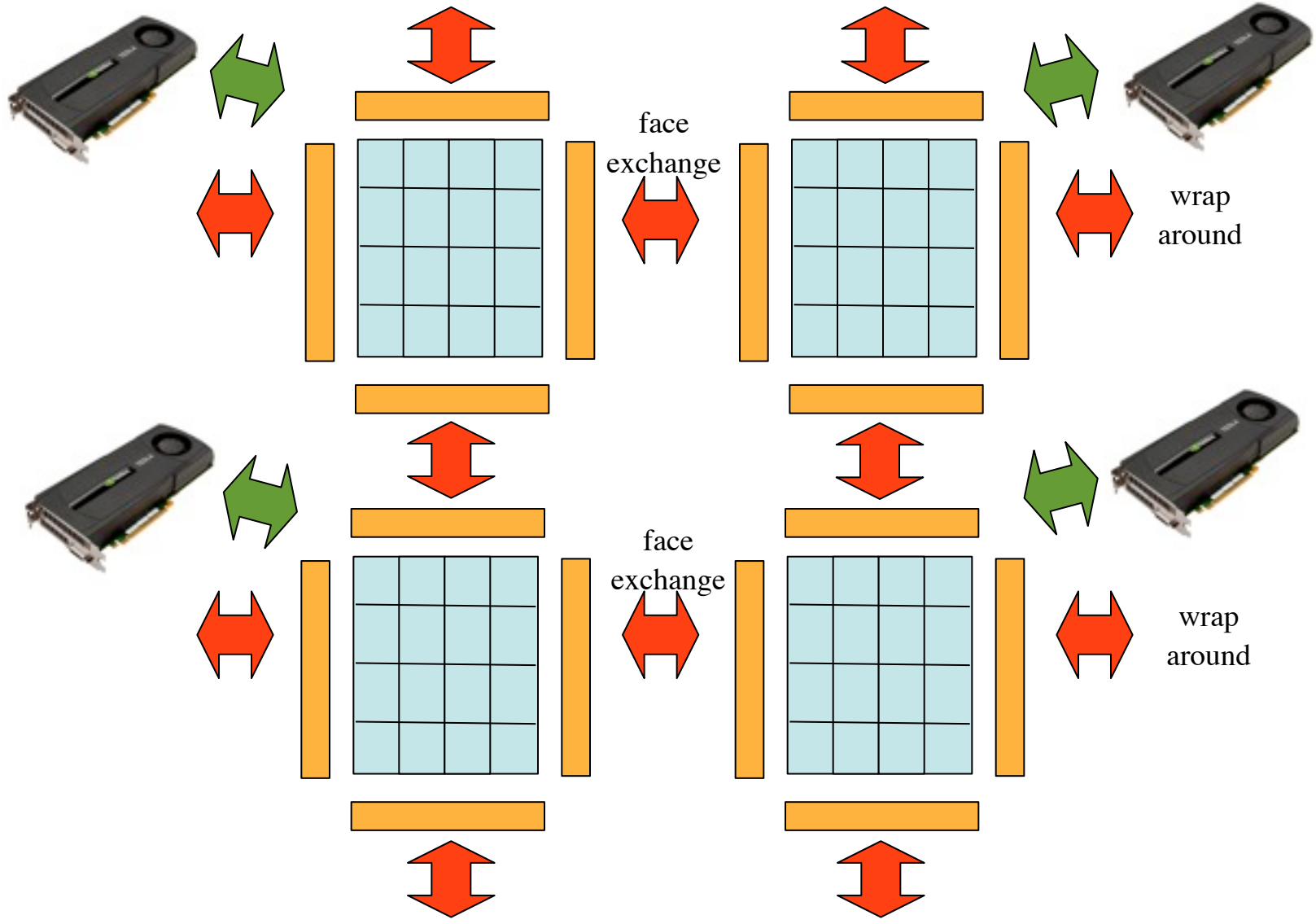
Single Precision Gauge Field Example

- V sites x 12 floats/site (2 row compressed)
- Break 12 into 3 chunks of 4 floats (float4-s)
- 1 block = V float4-s, 3 blocks for full field
- each thread reads a float4 at a time
  - coalesced reads
- Add Pad to avoid 'partition camping'
- Store ghost zones in Pad
- for spinors store ghost zones at end of data.
- similar for other types

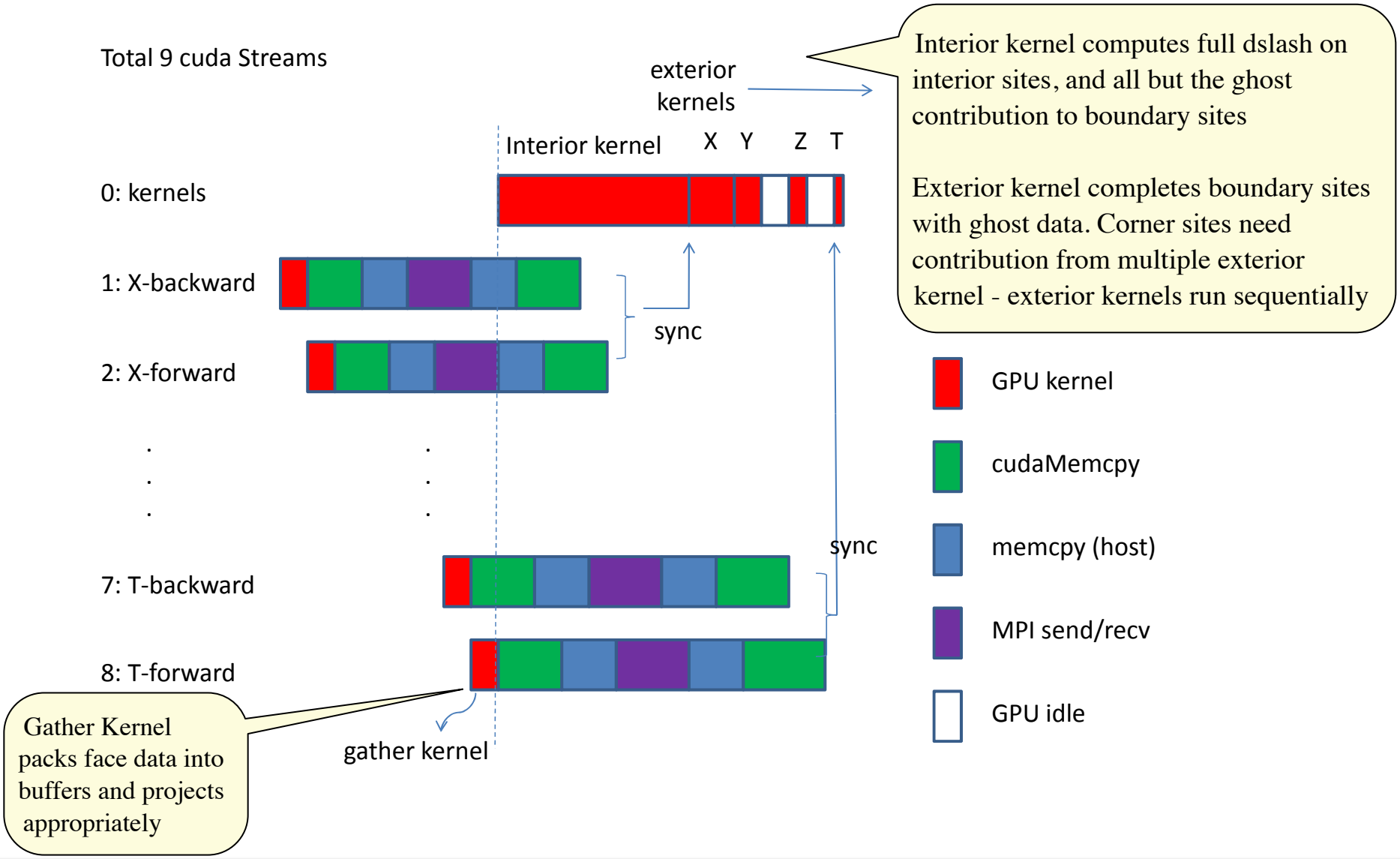
QUDA Community

- QUDA has unified separate development branches
  - Wilson, Clover, Twisted Mass, Staggered, DWF
- Integrated with Application Code - enlarge user base
  - Chroma & MILC
- A group of interested developers coalesced around QUDA
  - Mike Clark (Harvard), Ron Babich (BU) - QUDA leads
  - Bálint Joó (Jefferson Lab) - Chroma integration
  - Guochun Shi (NCSA) - Staggered Fermions, MILC integration
  - Will Detmold, Joel Giedt - previous contributors
  - Rich Brower, Steve Gottlieb
- Source Code Openly available from GitHub
  - http://github.com/lattice/quda

QUDA Parallelization

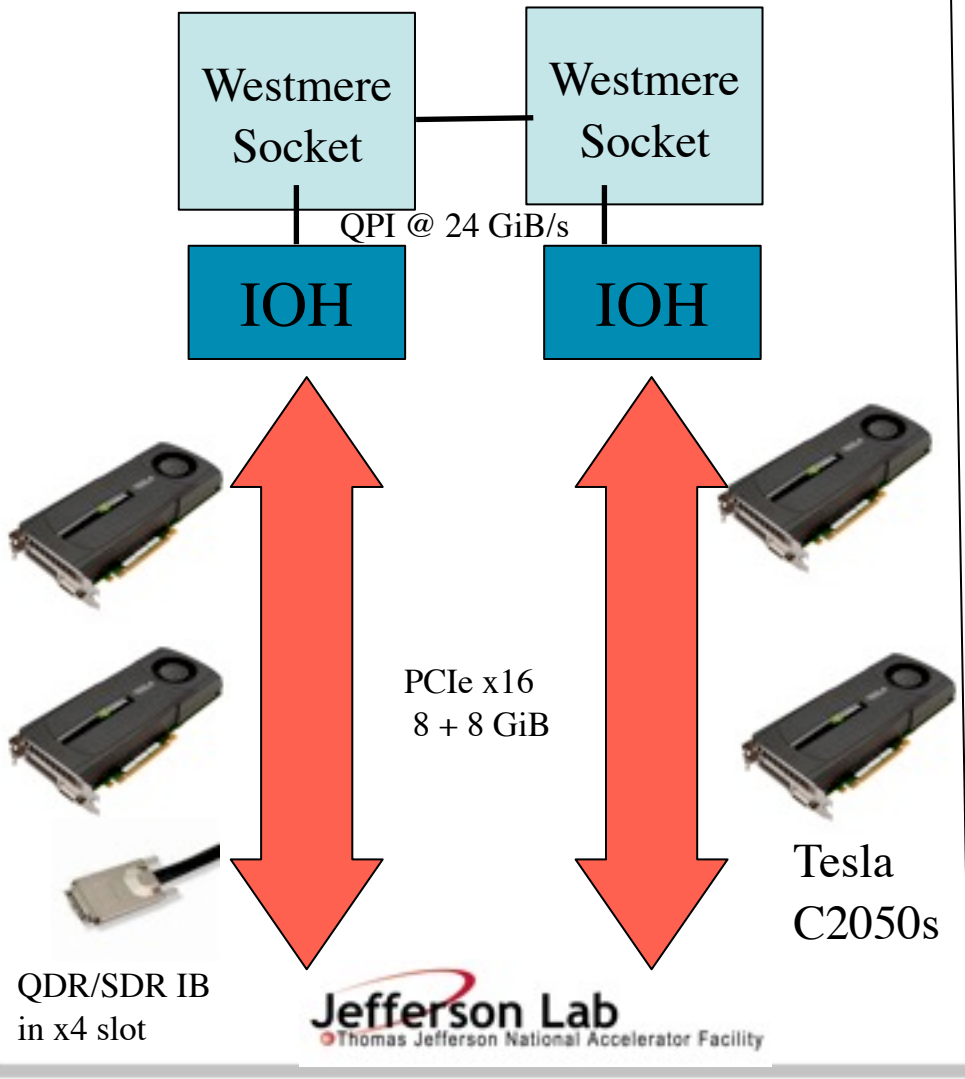


Face Exchange

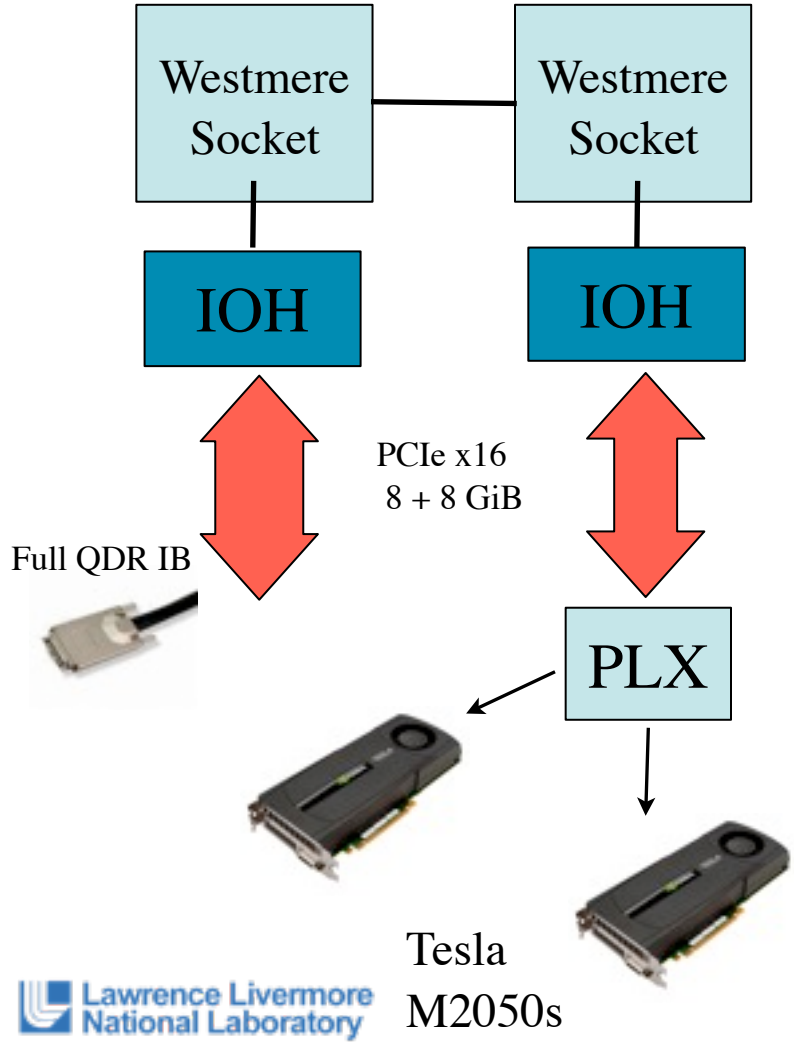


Test Clusters

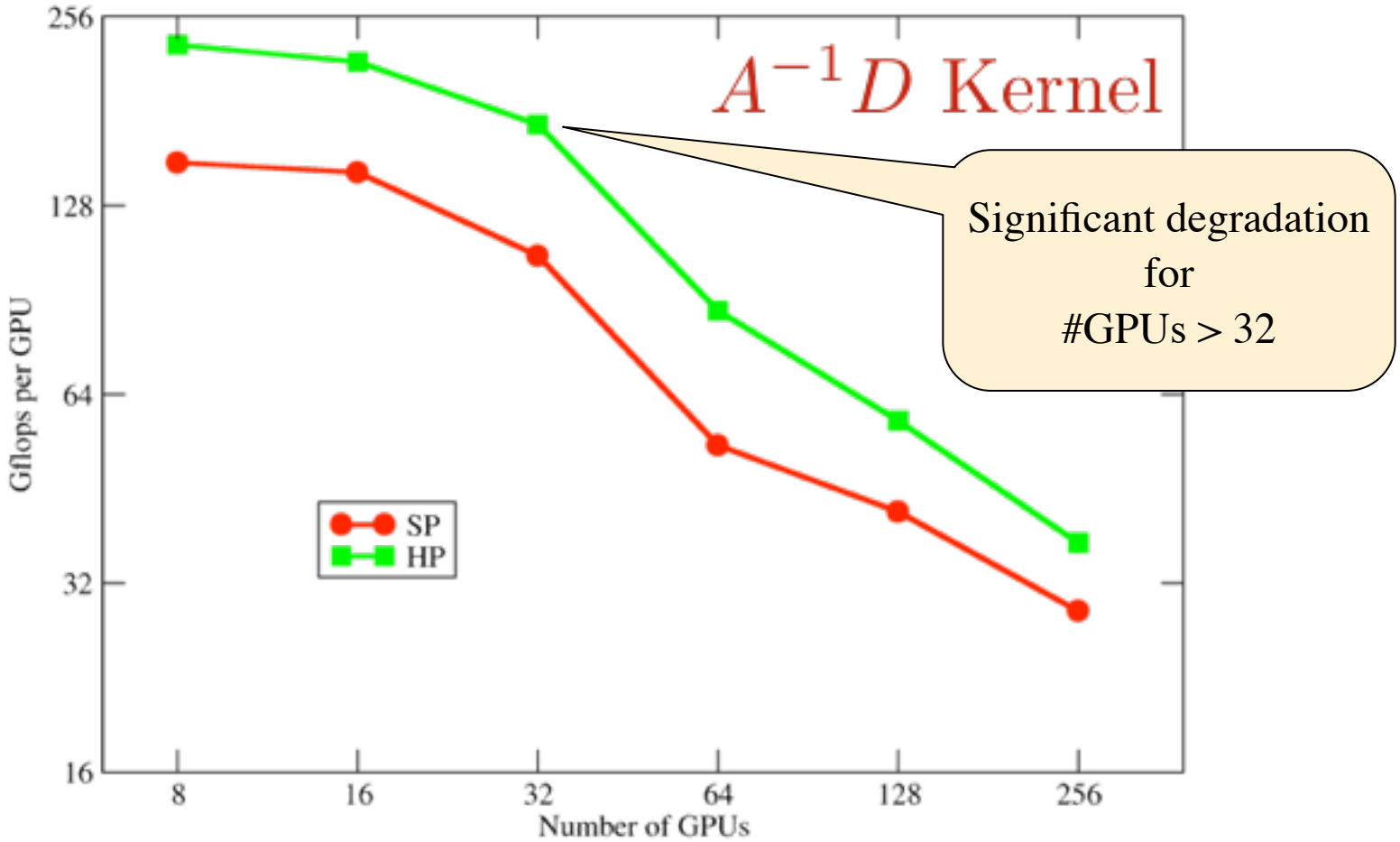
JLab Nodes (Up to 32 in partition)



Edge Nodes (Up to 392 in partition)

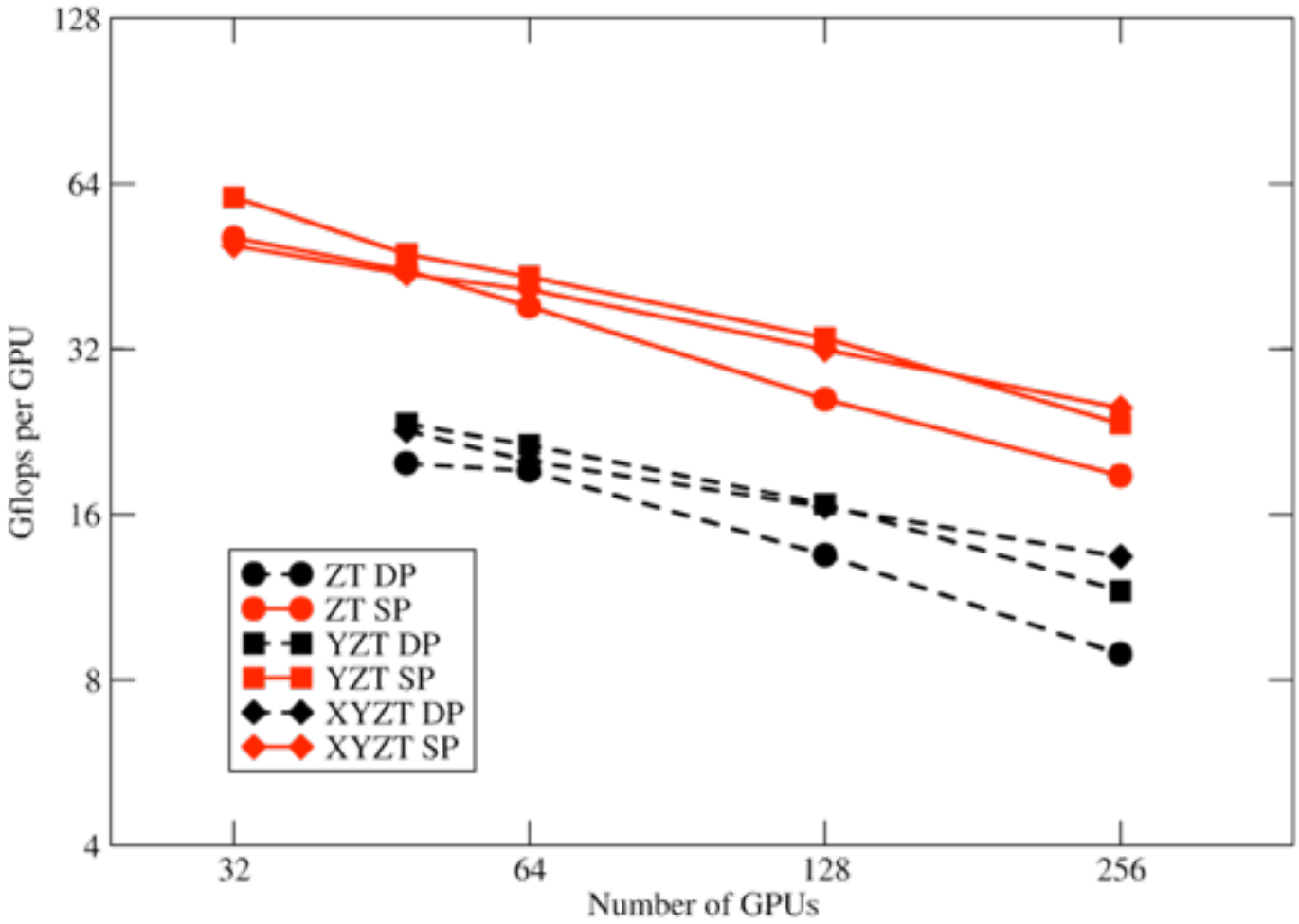


Basic Scaling: Clover Dslash on Edge



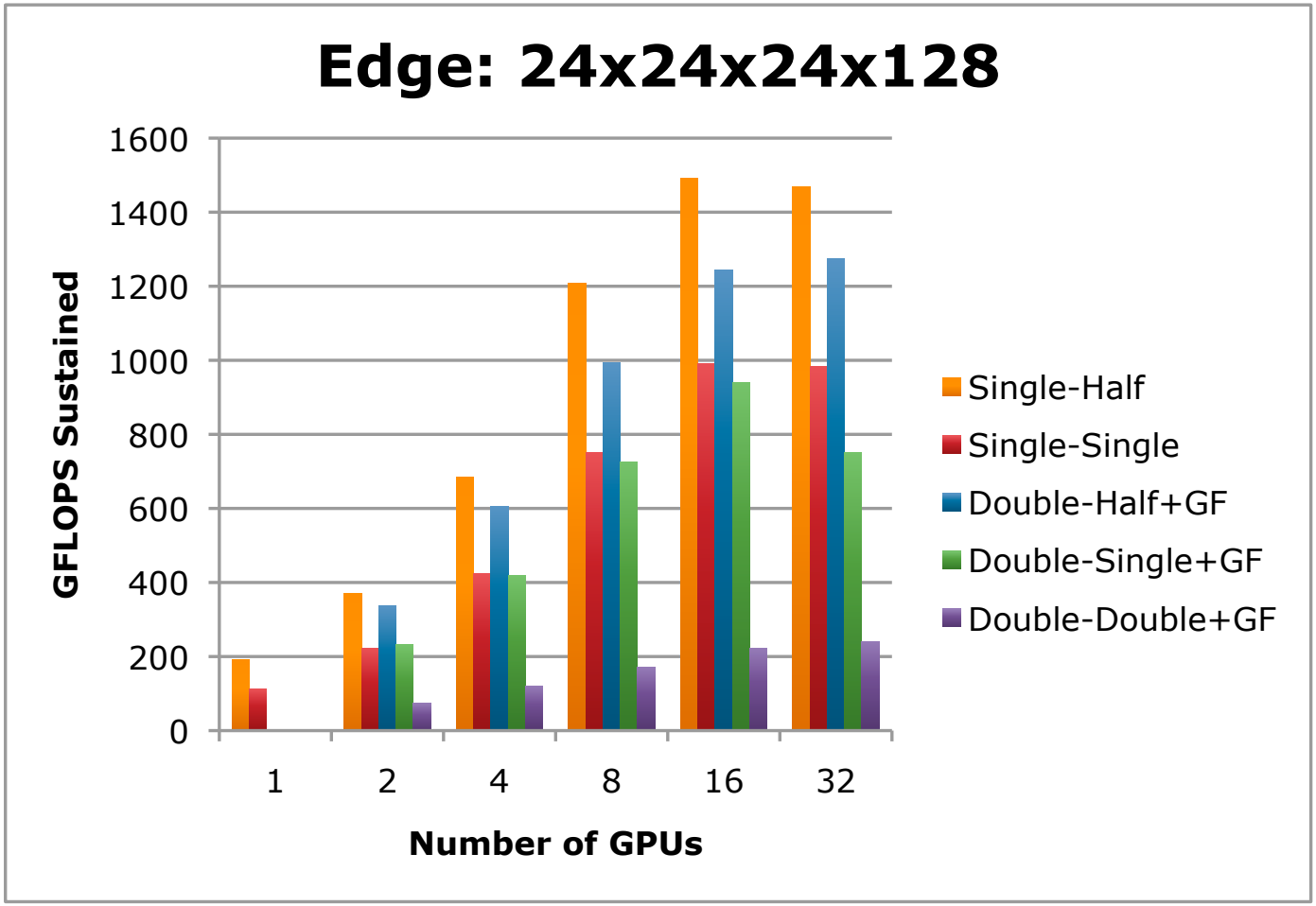
Tried 1, 2, 3 and 4D partitions, picked highest performance

Basic Scaling: AsqTAD on Edge

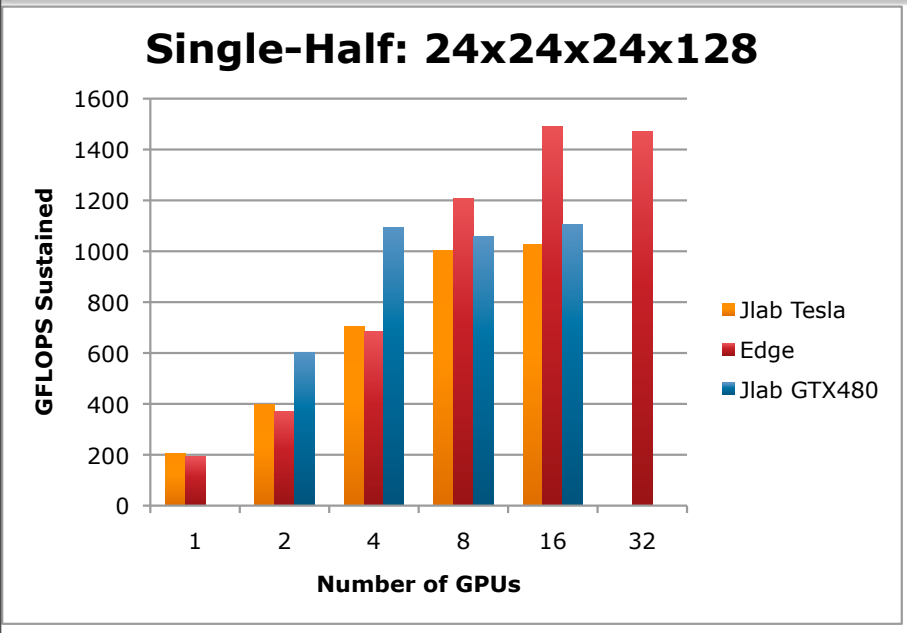


NB: Using Uncompressed Gauge Fields

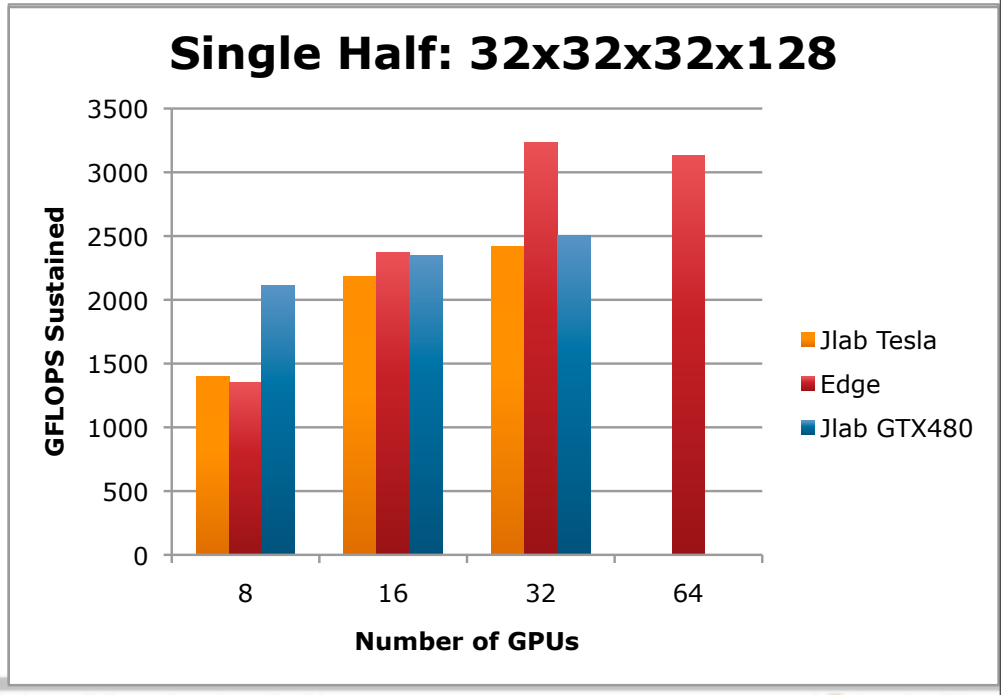
Scaling Of BiCGStab Solver



Scaling of BiCGStab Solver

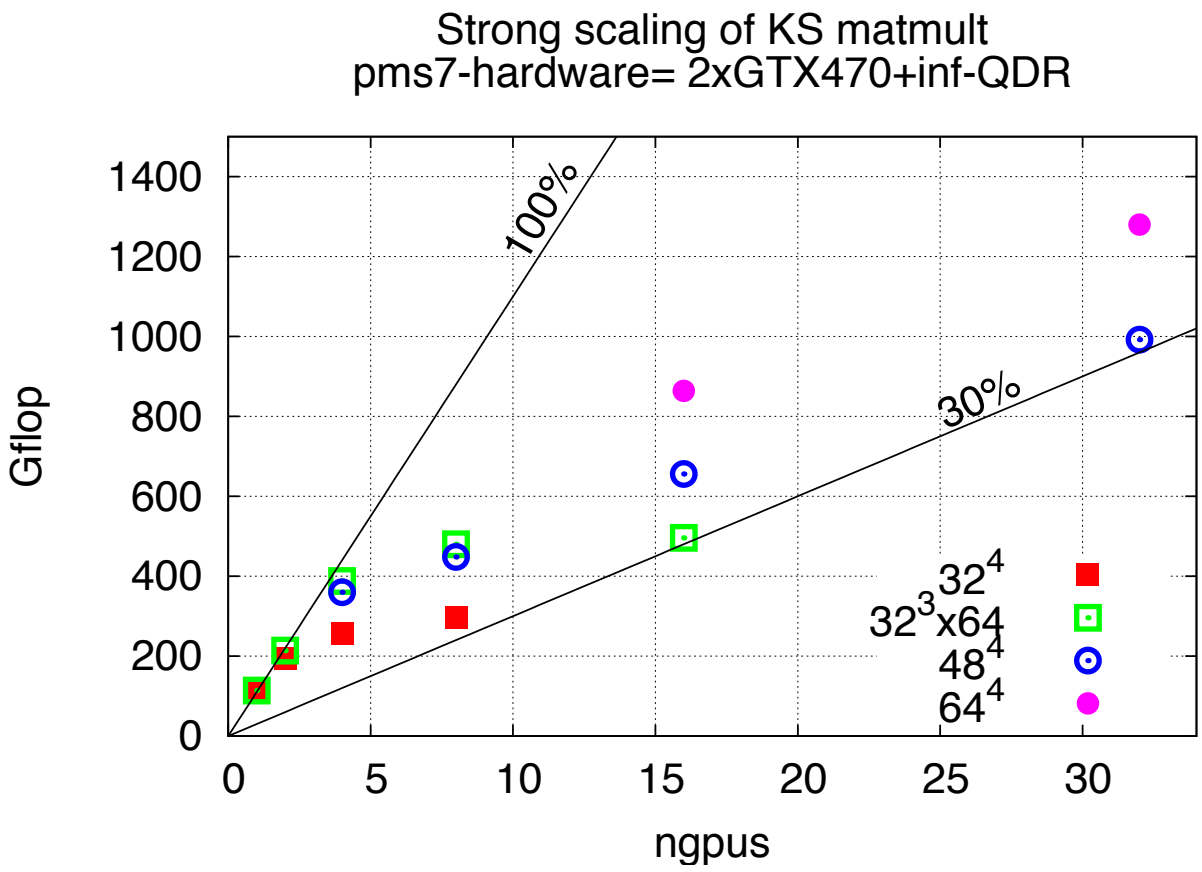


- JLab Tesla tops out at 8 GPUs
- JLab GTX480 tops out at 4 GPUs
  - GTX 480 cluster uses SDR
  - JLab Tesla: QDR in x4 slots



- JLab Tesla tops out at 16 GPUs
- GTX480 tops out at 8 GPUs
- Edge can almost make it to 32.

Others Fare Similarly

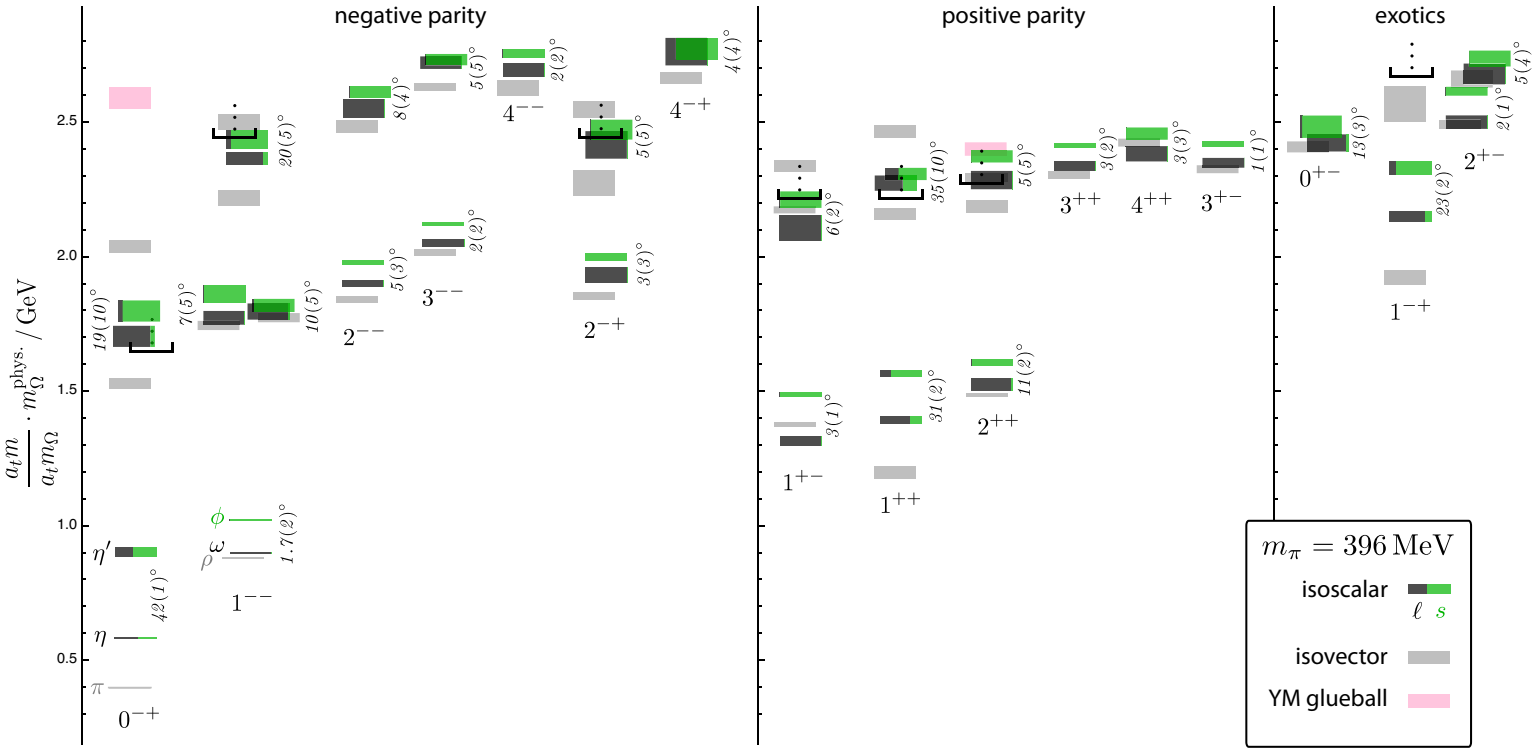


- Budapest-Wuppertal code. Courtesy of Kálmán Szabó
- 32^3 top out at 4-8 GPUs, 48^4 and 64^4 fare better (larger surf/vol)

Perfect for Capacity Work

- 4-8 GPUs can fit into a single host these days (8 in Tyan)
- Or can use 2 nodes with 4 GPU each + SDR connection
- Remaining Capacity Challenge: Amdahl's law
  - Solver is fast, but everything else is SLOW
    - Source (right hand side) creation
    - Link Smearing
    - Contractions
  - 8-12 cores for CPU work - used to be 128-256 cores.
- Solution:
  - Move more work to GPU - (come back to this later)
  - Rearrange Workflow

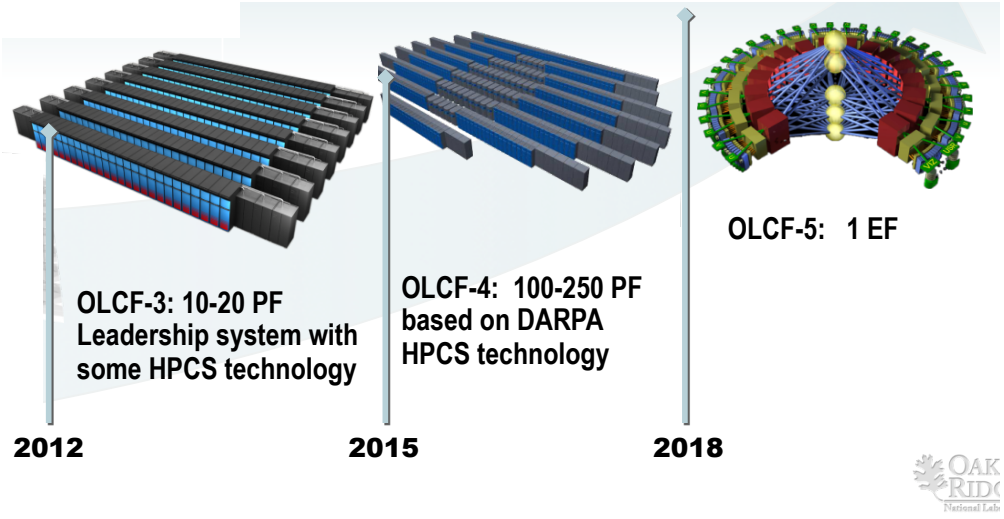
Isoscalar Meson Spectrum



- Dudek et. al. PRD, 83, 111502(R) (2011)
- 31 Million solves + large variational basis + anisotropic lattice
- All T to all T 'perambulators' using Distillation method
- Excited States, J^{PC} identfication, light/strange quark content
- Exotics within reach of JLab@12 GeV

What about Capability?

- Accelerated Capability Machines are on the Way
  - More Clusters like Edge
  - Cray XK6 System
  - Keeneland Phase 2
  - Titan@OLCF
  - Nvidia's Echelon?
- What about Capability GPU Capability?



What about Capability?

- Communication appears to be scaling bottleneck
- How to ameliorate this:
  - Wait for technology to improve
  - Change Algorithm, do less communication
- Domain Decomposed Preconditioner:
  - Divide lattice into domains, assign 1 domain to 1 GPU
  - No communication between domains (interior kernel only)
  - Apply preconditioner with 'inner solve'
  - Need a 'flexible' solver (variable preconditioner) e.g. GCR, Flexible GMRES etc.

Preconditioned GCR Algorithm

Apply Preconditioner: reduced precision inner solve

(Re)Start

$$r_0 = b - Mx$$
$$\hat{r}_0 = r_0$$
$$\hat{x}_0 = 0$$
$$k = 0$$

Generate Subspace

$$\hat{p}_k = \hat{K}^{-1}\hat{r}_k$$
$$\hat{z}_k = \hat{M}\hat{p}_k$$

Reduced Precision M v

$$\beta_{i,k} = (\hat{z}_i, \hat{z}_k)$$

Orthogonalize $\hat{z}$-s

$$\gamma_k = ||\hat{z}_k||$$

normalize $\hat{z}_k$

$$\alpha_k = (\hat{z}_k, \hat{r}_k)$$
$$\hat{r}_{k+1} = \hat{r}_k - \alpha_k \hat{z}_k$$
$$k = k + 1$$

repeat for all k or until residuum drops enough or convergence

Update Solution

Solve for χ_l l=k,k-1,...,0:

$$\gamma_l \chi_l \sum_{i=l+1}^{k} \beta_{l,i}\chi_l = \alpha_l$$

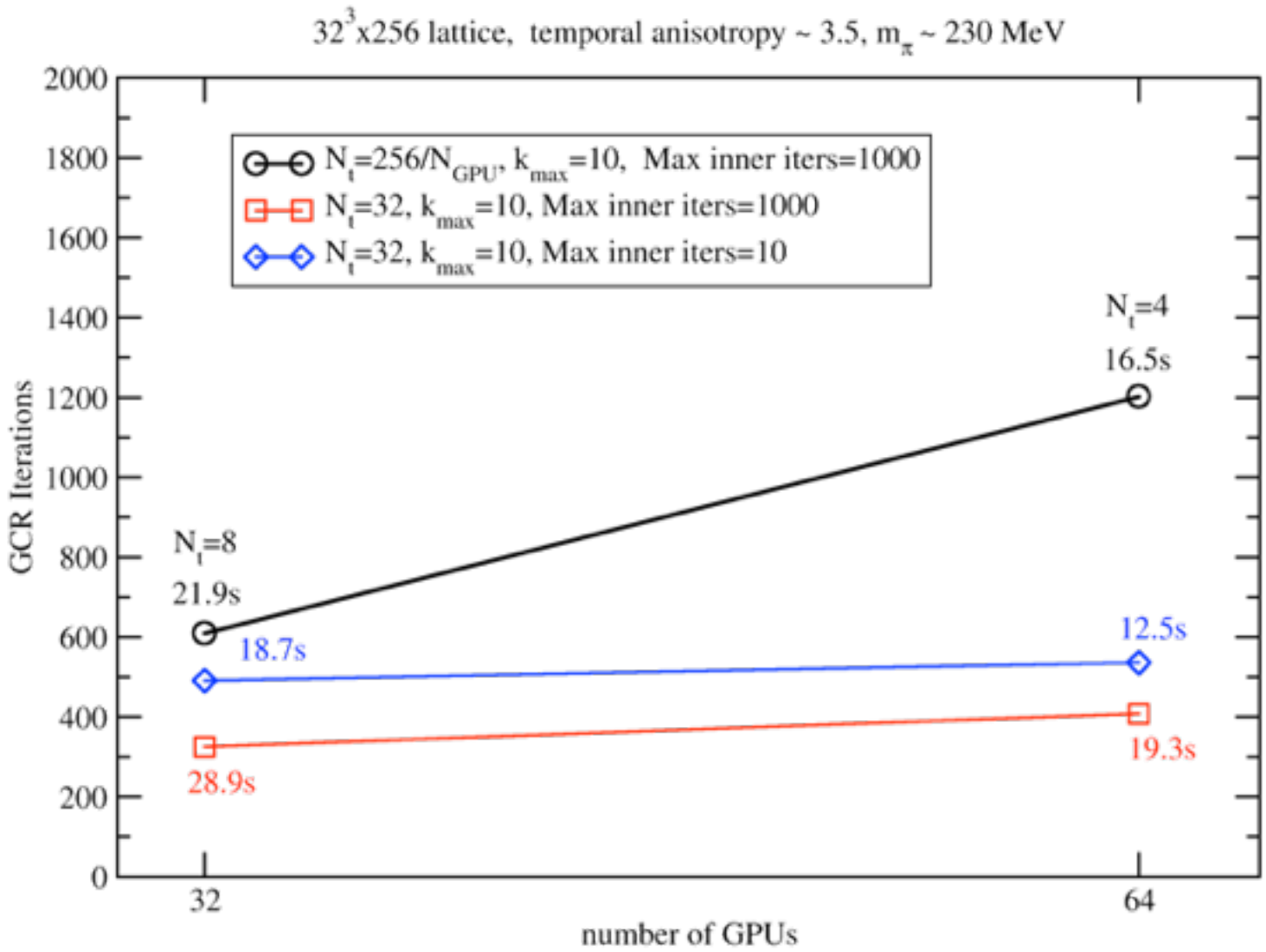
Compute correction for x:

$$\hat{x} = \sum_{i=0}^{k-1} \chi_i p_i$$
$$x = x + \hat{x}$$

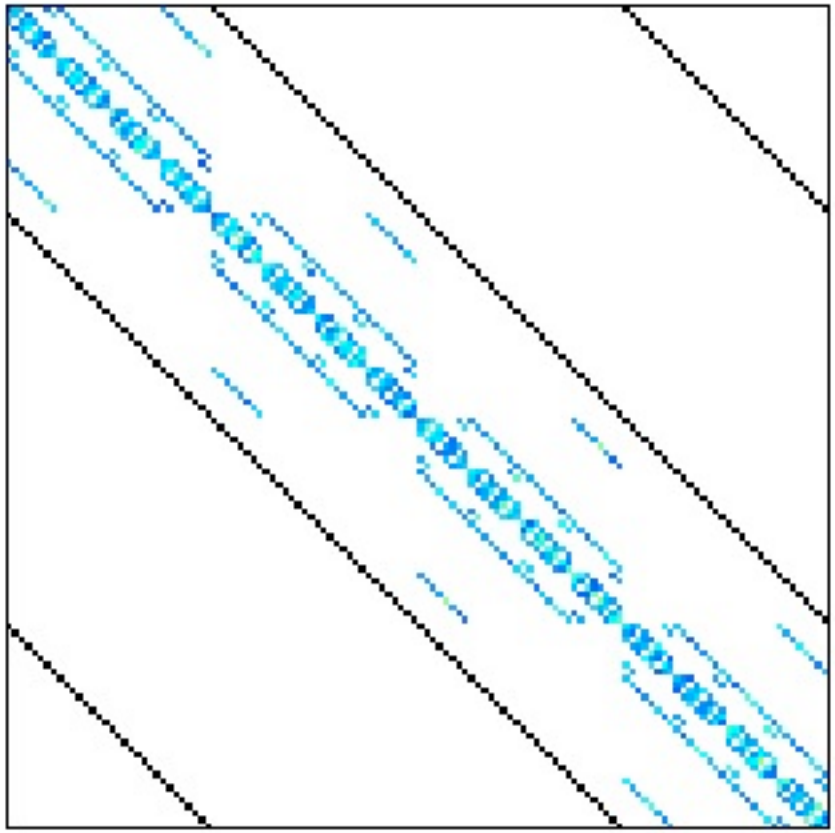
Full precision restart if not converged

Quantities with ^ are in reduced precision

Size Matters



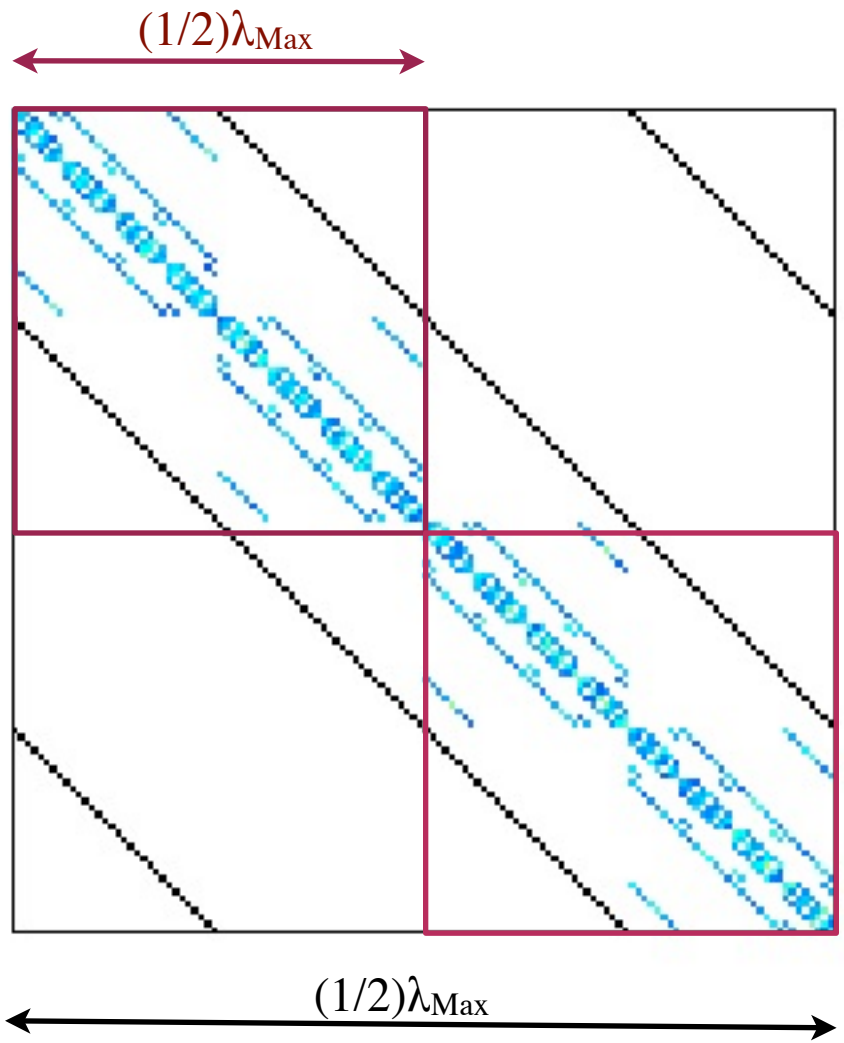
Size Matters



$(1/2)\lambda_{Max}$

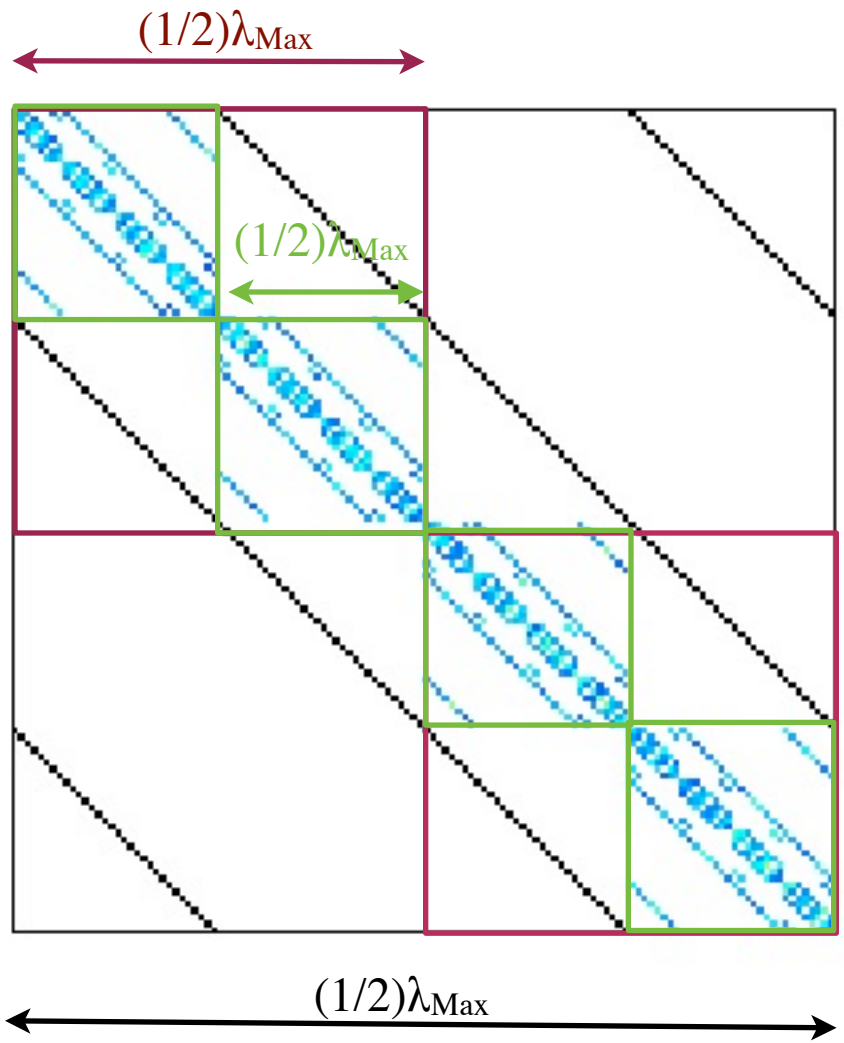
- No comms between domains
  - Block Diagonal Preconditioner
- Blocks impose λ cutoff
- Finer Blocks
  - lose structure in operator
  - lose long wavelength/low energy modes
- Heuristically (& from Lüscher)
  - keep wavelengths of ~ $O(\Lambda_{QCD}^{-1})$
  - $\Lambda_{QCD}^{-1} \sim 1$fm
  - Aniso: (a_s=0.125fm, a_t=0.035fm)
    - Our case: 8^3x32 blocks are ideal
  - Iso: 1fm ~ 8-10 sites (a=0.11fm)
  - Min. blocksize has scaling implications

Size Matters



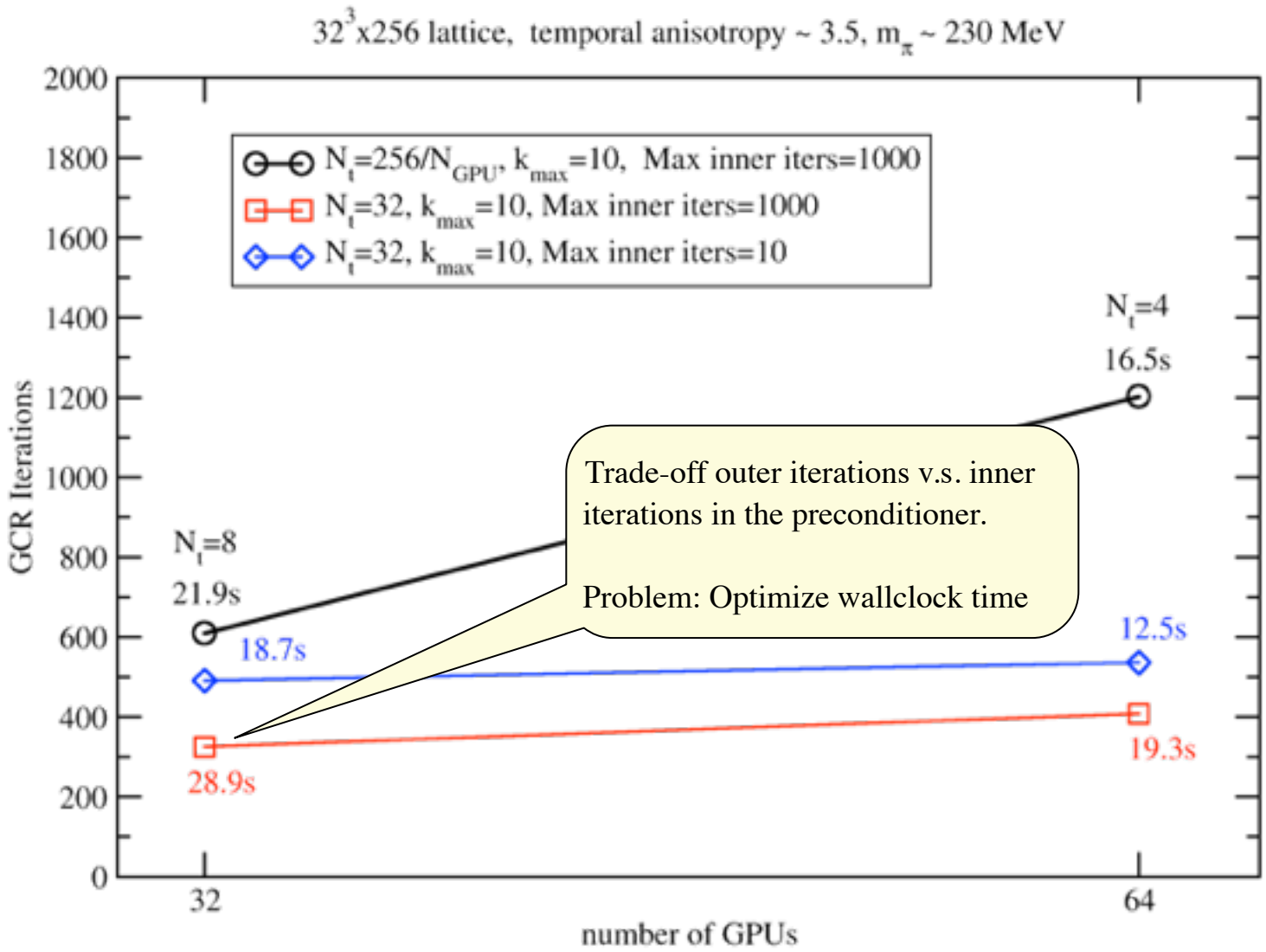
- No comms between domains
  - Block Diagonal Preconditioner
- Blocks impose λ cutoff
- Finer Blocks
  - lose structure in operator
  - lose long wavelength/low energy modes
- Heuristically (& from Lüscher)
  - keep wavelengths of ~ $O(\Lambda_{QCD}^{-1})$
  - $\Lambda_{QCD}^{-1} \sim 1$fm
  - Aniso: (a_s=0.125fm, a_t=0.035fm)
    - Our case: 8^3x32 blocks are ideal
  - Iso: 1fm ~ 8-10 sites (a=0.11fm)
  - Min. blocksize has scaling implications

Size Matters



- No comms between domains
  - Block Diagonal Preconditioner
- Blocks impose λ cutoff
- Finer Blocks
  - lose structure in operator
  - lose long wavelength/low energy modes
- Heuristically (& from Lüscher)
  - keep wavelengths of ~ $O(\Lambda_{QCD}^{-1})$
  - $\Lambda_{QCD}^{-1} \sim 1$fm
  - Aniso: (a_s=0.125fm, a_t=0.035fm)
    - Our case: 8^3x32 blocks are ideal
  - Iso: 1fm ~ 8-10 sites (a=0.11fm)
  - Min. blocksize has scaling implications

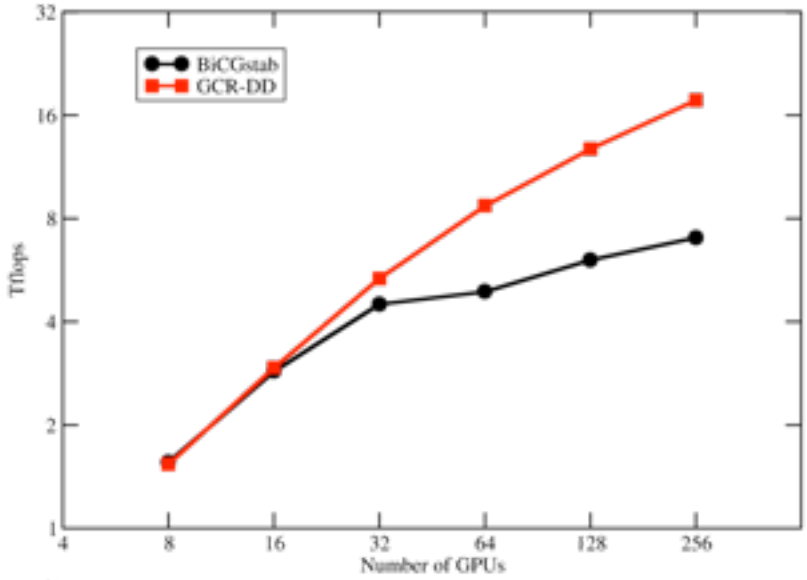
Size Matters

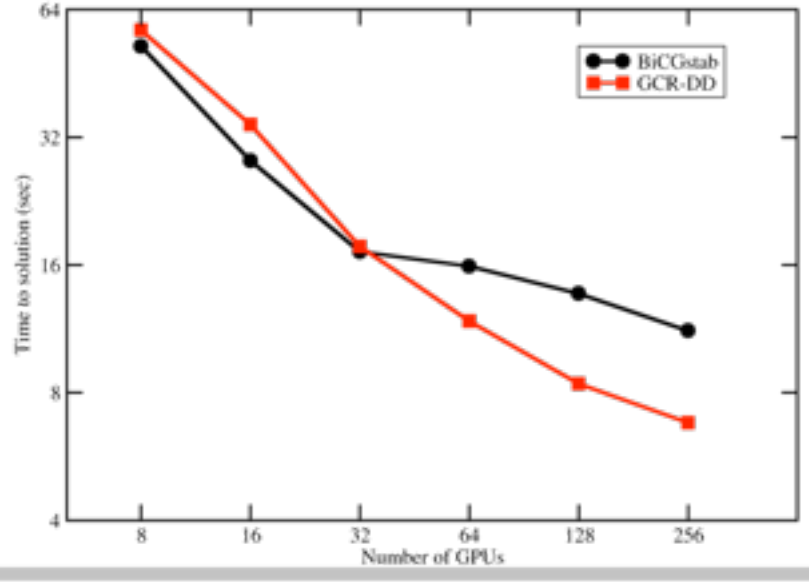


Trade-off outer iterations v.s. inner iterations in the preconditioner.

Problem: Optimize wallclock time

Strong Scaling of DD-GCR



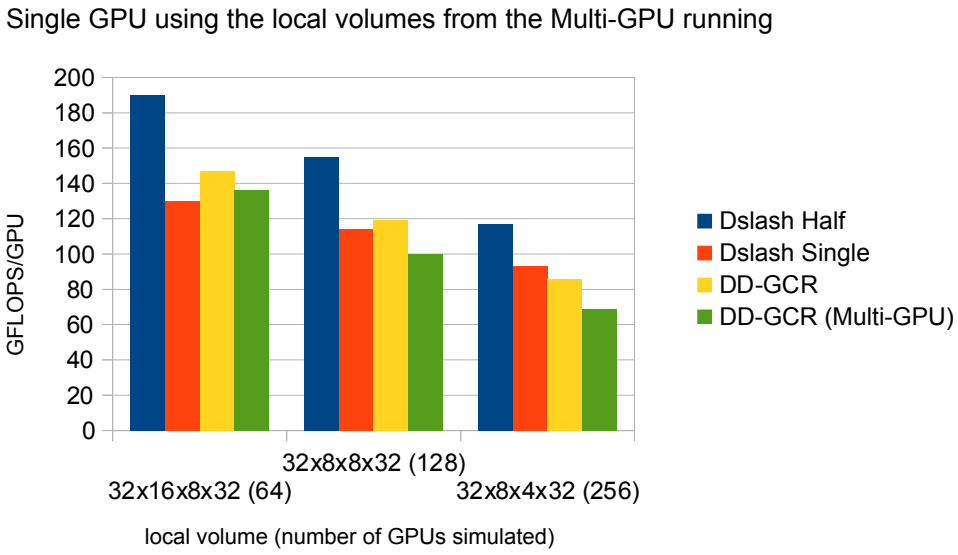


- With DD-GCR, we can scale up to 256 GPUs on 32^3x256 lattice.
  - 8^3x32 blocks: 512 GPUs max
- > 2x more FLOPS v.s. BiCGStab
  - but only 1.64x faster walltime
  - trade off fast inner/slow outer iterations
- Scaling drops off at 256 GPUs
  - outer solver + reductions (?)
- This is just the first step: need more research on 'architecture aware' algorithms

Babich, Clark, Joó, Shi, Brower, Gottlieb, accepted for SC'11

“Please sir, can we have some more?”

- 17.2 Tflop on 256 GPUs = 69 Gflops/GPU
  - using all the precision tricks
  - The local problem is small
    - Low occupancy?
    - Driver overheads?
    - Communications?
  - Single GPU runs
    - with ‘strong scaling’ local volumes
  - Communications seems not the worst bottleneck here.
- Current Multi-GPU sweet spot: 128 GPUs ~ 100Gflop/GPU.
- Ambition: 40^3x256 lattice, 1000 GPUs, 50-100 Tflops(?)
- Large algorithmic space to explore



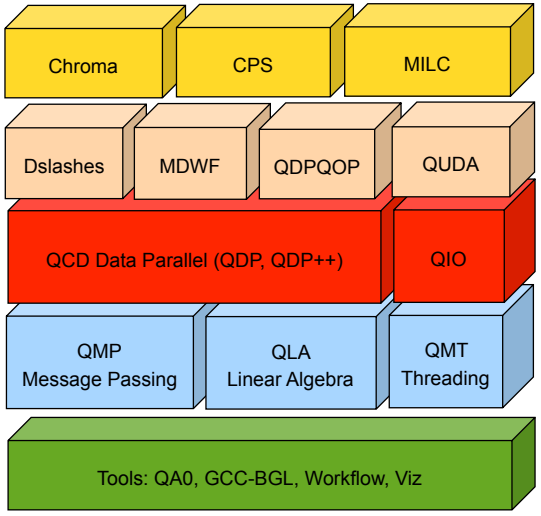
Related Algorithmic Work

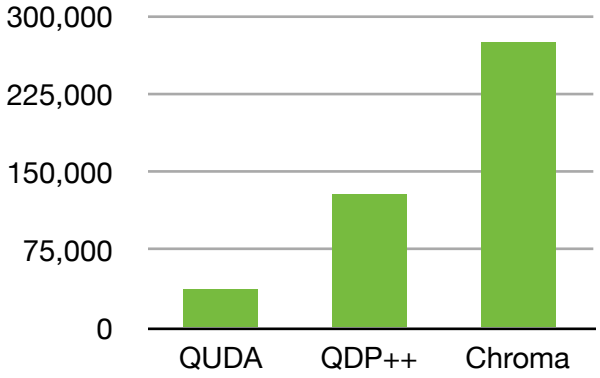
- Schwarz preconditioner
  - (SAP+GCR) Lüscher, Comput.Phys.Commun. 156(2004) 209-220
  - (RAS+ flex. BiCGStab) Osaki, Ishikawa, PoS(Lattice2010), 036
- Domain Decomposed HMC
  - Lüscher, JHEP 0305 (2003) 052
  - Lüscher, Comput.Phys.Commun.165:199-220,2005
- Multi-Grid:
  - Babich et. al., Phys.Rev.Lett.105:201602,2010
  - Osborn et. al., PoS Lattice2010:037,2010
- Deflation:
  - Lüscher, JHEP 0707:081,2007, JHEP 0712:011,2007
  - Stathopoulos & Orginos: SIAM J. Sci. Comput. 32, pp. 439-462

Challenge: Updating preconditioner/ deflation space in the Gauge evolution.

Programming GPUs, Frameworks

- GPU Programming today
  - CUDA, OpenCL, #pragma
  - low level, ‘general’
- Libraries: e.g. QUDA
  - Hide low level details
  - problem & architecture specific
- Domain Specific Frameworks
  - QDP++, QLUA, QDP/C
  - productivity enabling ‘glue’
- Application Suites: e.g. Chroma
  - large, prefer not to re-engineer
  - too large investment to throw away





Lines of C/C++ Code per package measured on May 11, 2011, using CLOC http://cloc.sourceforge.net/

QDP++

- QDP++ - a Data-Parallel Domain Specific framework for LQCD
  - Embedded in C++
  - provides LQCD types/operations
  - arithmetic ‘expressions’ on multi-tensor index objects
- productivity layer - the purpose is to be expressive
  - bedrock for Chroma code
- Implemented using
  - nested templates (for indices)
  - expression templates (ETs)
  - specialization (optimization)
- Parallel nature hidden from user
  - ETs hide OpenMP, QMT, QMP/MPI etc

parallel ‘forall’

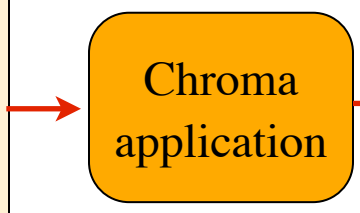
```
LatticeFermion x, y,z;
Real a = Real(5);
gaussian(x);
gaussian(y);
x += a*y;
z = shift(x,0,FORWARD);
Double zn = norm2(z);
```

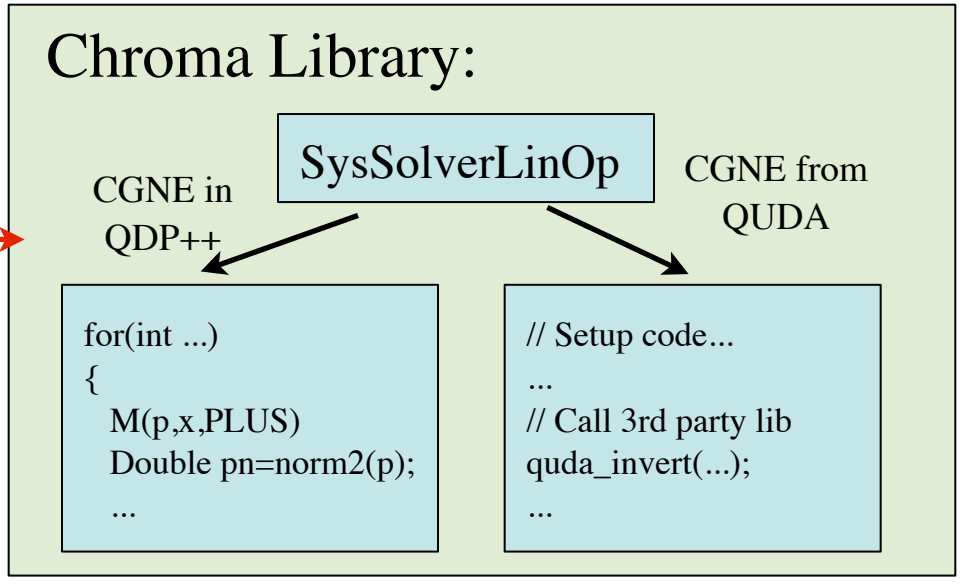
parallel reduce

Chroma

- Large library of LQCD components (solvers, gauge generation algs.)
  - e.g. CGNE, BiCGStab, HMC, Symplectic Integrators, physics...
  - implemented using QDP++ or wrapping 3rd party routines
- Key applications: chroma (analysis) and hmc (gauge generation)
- Applications driven by XML
- Can use as 'out of the box' application or as library to build on

```
<?xml version='1.0'?>
<chroma>
 ..
 <InvertParams>
 <invType>CG_INVERTER</invType>
 <RsdCG>1.0e-8</RsdCG>
 <MaxCG>1000</MaxCG>
 </InvertParams>
 ...
</chroma>
```





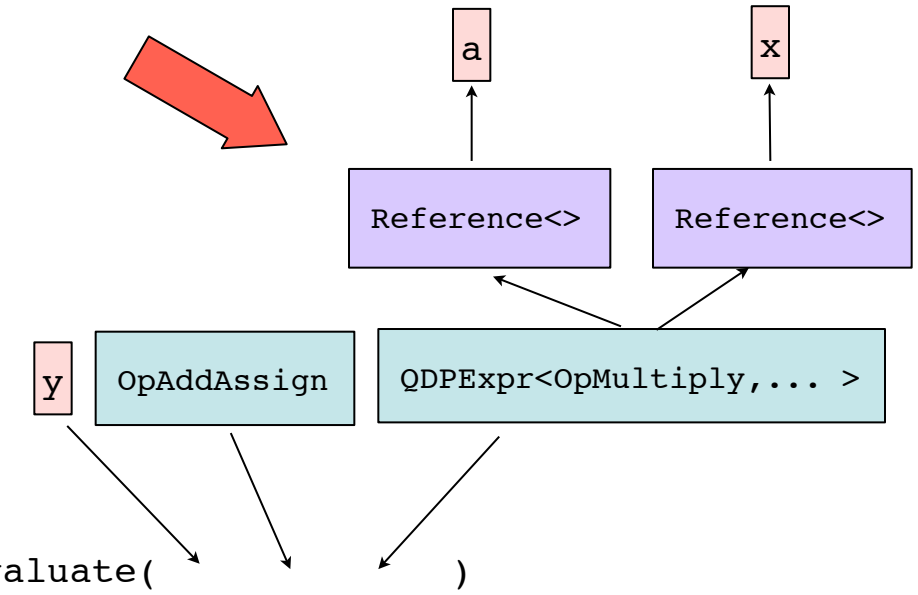
Re-engineering QDP++

- Move QDP++ to the GPU
  - Speed up all of Chroma that is not part of QUDA library
  - Needs to be 'just good enough'
    - there will always be super optimized libraries
    - but need to counter Amdahl's law for rest
- How to generate GPU Kernels for QDP++ expressions?
  - Compile time: e.g. source to source transformation
    - must deal with QDP++ types, expressions
    - but must retain full C++ compatibility
    - not easy, maybe doable with a framework like ROSE?
  - Alternative: Generate kernels 'just-in-time' (JIT)
    - The use of expression templates can help

JIT + Expression templates

- `QDPExpr` is a C++ Type
  - recursive
  - compile time type signature
  - run time parameter binding
- First instantiation:
  - Code Generation for signature
  - Just-In-Time Compilation
  - Dynamic Library of kernels
- Data movement
  - explicit v.s. automated

```
// QDP++ code
LatticeFermion x, y;
Real a=Real(1);
gaussian(x);
y += a * x;
```

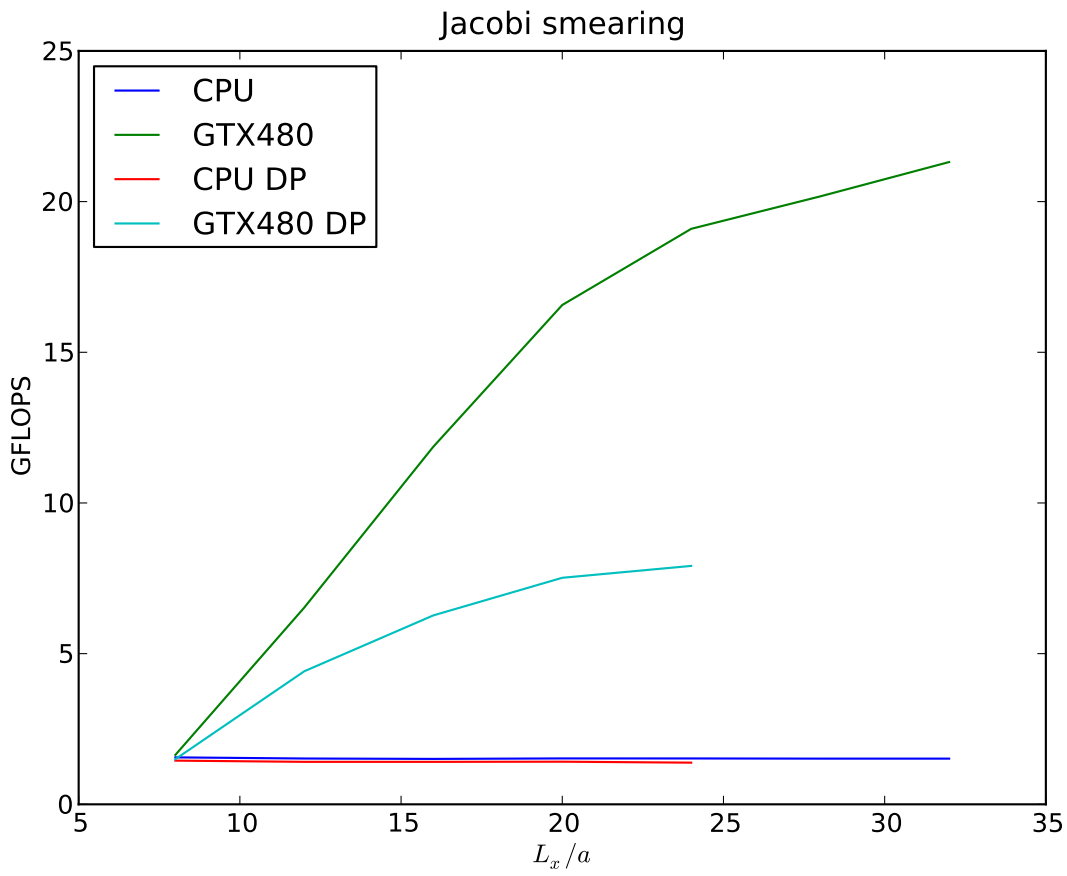



```
evaluate(                    )
{ // Lookup
  KFunc* kernel = lookupKernel(hash);
  // Generate if needed
  if( !kernel ) generate(kernel, hash);
  (*KFunc)(...); // Invoke
}
```

Current Progress

- Two independent efforts have sprung up
  - Frank Winter (U. of Edinburgh), Jie Chen (Jefferson Lab)
- Code Generation triggered by the QDP++ evaluate() functions
- Just In Time compilation: use 'system()' call to invoke nvcc
- Loading Resulting Kernels
  - generate .o file, use system dynamic loader interface or
  - generate PTX, load with CUDA driver API
- Data Movement:
  - push() pop() interface to push/pop data onto/off device
  - automatic management of data movement (sfw. cache)
- Beginning collaboration to join the two efforts

Re-Engineering QDP++



- Chroma Jacobi Smearing Interface accelerated. (F. Winter)

```
template<typename T>
void jacobiSmear(const multi1d<
    LatticeColorMatrix>& u, T& chi,
const Real& kappa, int iter, int
    no_smear_dir, const Real& _norm)
{
  T psi;
  Real norm;
  T s_0,h_smear;

  psi.pushToDevice();
  for(int mu = 0; mu < Nd; ++mu )
   u[mu].pushToDevice();
  chi.pushToDevice();
  h_smear.pushToDevice();
  s_0.pushToDevice();

  s_0 = chi;

  for(int n = 0; n < iter; ++n)
  {
    psi = chi;
```

http://github.com/fwinter/qdp

see Parallel Talk by F. Winter at Lattice'11

Re-engineering QDP++

- Our (ideal) wish list for the overall system
  - ‘syntax compatible’ with current QDP++, no change to Chroma
  - Multi-GPU / host, Multi-host
  - Generalize to also produce CPU code
    - same framework for CPU & GPU
  - Code transformation and auto-tuning of generated code
  - Configurable Data layout if possible.
  - Automated memory management (e.g. host/device traffic?)
  - Compilation via ‘system()’ is hacky
    - JIT via LLVM to PTX/binary?
    - or go back to compile time source transformation: ROSE?
- We’ll need help from Tools/Performance/DSL community.

Optimizaton Opportunities

QDP++ Code:

```
z = a*x + y;
zn = norm2(z);
y += b*z;
```

Optimizaton Opportunities

QDP++ Code:

```
z = a*x + y;
zn = norm2(z);
y += b*z;
```

JIT-ed (Pseudo) Code:

naive, untuned

```
// z= a*x + y
#pragma unroll,vectorize
forall(i=0;...) {
  z[i] = a*x[i] + y[i];
}

// zn = norm2(z)
#pragma unroll,vectorize
forall_reduce(zn=0,i=0;...) {
  zn += z[i]*z[i];
}

// y + = b*z
#pragma unroll,vectorize
forall(i=0; ...) {
  y[i] += b*z[i];
}
```

Optimizaton Opportunities

QDP++ Code:

```
z = a*x + y;
zn = norm2(z);
y += b*z;
```

JIT-ed (Pseudo) Code:

naive, untuned

```
// z= a*x + y
#pragma unroll,vectorize
forall(i=0;...) {
  z[i] = a*x[i] + y[i];
}

// zn = norm2(z)
#pragma unroll,vectorize
forall_reduce(zn=0,i=0;...) {
  zn += z[i]*z[i];
}

// y + = b*z
#pragma unroll,vectorize
forall(i=0; ...) {
  y[i] += b*z[i];
}
```

Autotuned (Pseudo) Code:

unrolled, vectorized

```
Vector vzn = bcast_vec(0);
Vector va = bcast_vec(a);
Vector vb = bcast_vec(b);

#pragma omp for reduction(+:vzn)
for(i=0;...;i+=veclen*UNROLL) {
  Vector vz;
  Vector vx = load_vec(&x[i]);
  Vector vy = load_vec(&y[i]);

  vz = vec_add(vy,
               vec_mul(va,vx));
  vzn = vec_add(vzn,
                vec_mul(vz,vz));
  vy = vec_add(vy,
               vec_mul(b,vz);

  vec_store(&z[i], vz);
  vec_store(&y[i], vy);
  ...
  // UNROLL times
}
zn = vec_sum(vzn)
```

Optimizaton Opportunities

QDP++ Code:

```
z = a*x + y;
zn = norm2(z);
y += b*z;
```

JIT-ed (Pseudo) Code:

naive, untuned

```
// z= a*x + y
#pragma unroll,vectorize
forall(i=0;...) {
  z[i] = a*x[i] + y[i];
}

// zn = norm2(z)
#pragma unroll,vectorize
forall_reduce(zn=0,i=0;...) {
  zn += z[i]*z[i];
}

// y + = b*z
#pragma unroll,vectorize
forall(i=0; ...) {
  y[i] += b*z[i];
}
```

Autotuned (Pseudo) Code:

unrolled, vectorized

```
Vector vzn = bcast_vec(0);
Vector va = bcast_vec(a);
Vector vb = bcast_vec(b);

#pragma omp for reduction(+:vzn)
for(i=0;...;i+=veclen*UNROLL) {
  Vector vz;
  Vector vx = load_vec(&x[i]);
  Vector vy = load_vec(&y[i]);

  vz = vec_add(vy,
               vec_mul(va,vx));
  vzn = vec_add(vzn,
                vec_mul(vz,vz));
  vy = vec_add(vy,
               vec_mul(b,vz);

  vec_store(&z[i], vz);
  vec_store(&y[i], vy);
  ...
  // UNROLL times
}
zn = vec_sum(vzn)
```

Similar, but more elaborate idea for GPUs

Future Hardware

- NVIDIA
  - next: Kepler GPU, rumored to be 3-4x Fermi FLOPS/Watt
  - after: Maxwell GPU
- Intel MIC architecture
  - Knights Corner announced at ISC'11: >50 cores
  - Current: Knights Ferry Software Development Platform
    - 7 Demos at ISC'11
  - x86 compatible cores, 512 bit vector unit
- AMD
  - Next gen. GPU architecture (GCN). More SIMD, less VLIW
  - AMD Fusion: GPU + CPU = APU (Accelerated Processing Unit)
  - Announced next generation Fusion System Architecture (FSA)

Remember CPUs?

- GPUs are great, but CPUs still exist... (and improve)
- New #1 on Top 500 is SPARC based K-computer (http://top500.org).
  - ~8.2 (HPL) PFlops, ~9.9 MW => ~1.2 kW/(HPL) TFlop
- CPU trends:
  - more cores
  - shared caches
  - Longer vectors (AVX: 256 bit= 8 SP / 4 DP)
  - More H/W threads (Intel Nehalem/Westmere: 2, Power7: 4)
- CPU Based Capability Systems are still with us (or coming soon)
  - Cray XT/XE,
  - BlueWaters,
  - BlueGene

Conclusions

- GPUs are extremely useful for LQCD Calculations
  - especially for capacity workloads
  - already producing useful physics (e.g. spectrum of hadrons)

- Successfully scaled DD+GCR solver to 256 GPUs (114,688 cores?)
  - Need more research on ‘architecture aware’ algorithms
    - RAS DD preconditioned GCR reduces communications
    - 17 Tflops on 256 GPUs is only the beginning
    - large algorithmic space to explore
  - Technology also improves
    - direct GPU to GPU transfers

Conclusions (cont'd)

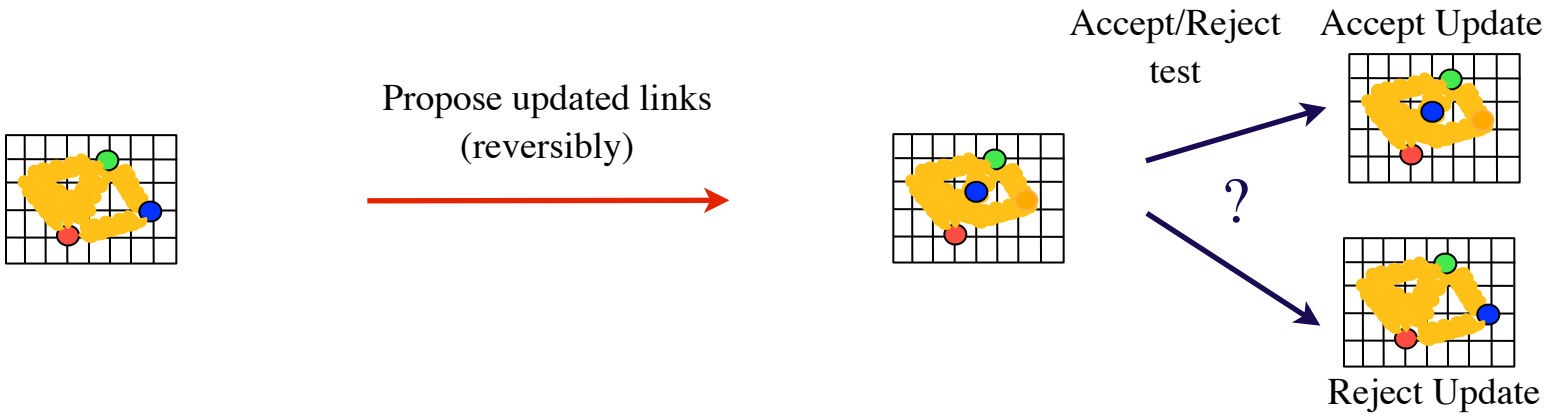
- Need to move more code to the accelerator
  - Counteract Amdahl's Law: in gauge generation AND analysis
  - Porting the framework level (QDP++) would be most useful
  - BUT want system to work on CPU as well (portable performance)
  - QDP++ Challenges
    - Expressions => Kernel Generation, Data Movement
    - First steps: efforts by Frank Winter, Jie Chen -> Collaboration
  - A lot of work: plenty more scope for collaboration
- Heterogeneity is now mainstream
  - many (sufficiently different) options (NVIDIA, AMD, soon Intel)
  - logical to expect CPU+GPU integration in future...
- CPUs, we still love you too!

Acknowledgements

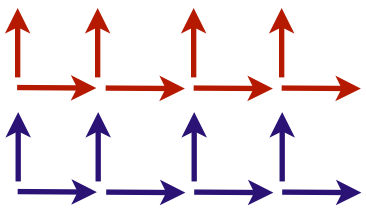
- QUDA Collaborators:
  - Mike Clark, Ron Babich, Guochun Shi, Steve Gottlieb, Rich Brower
- Thanks to Jefferson Lab and LLNL for cluster use.
- B. Joó acknowledges funding through US DOE grants
  - DE-FC02-06ER41440 and DE-FC02-06ER41449 (SciDAC)
  - DE-AC05-06OR23177 under which JSA LLC operates JLab.
- M. Clark acknowledges funding through NSF grant OCI-1060067
- G. Shi acknowledges funding through the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign.

Backup Slides

Hybrid Monte Carlo



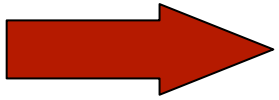
Canonical coordinates: U
Potential: S(U)



Canonical momenta: π
Kinetic energy: $(1/2)\,\pi^2$

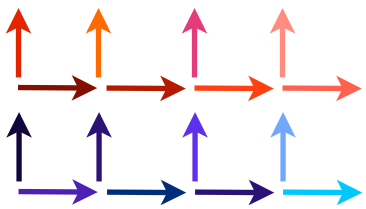
$$H = (1/2)\pi^2 + S(U)$$

Hamiltonian
Molecular Dynamics



Reversible
Symplectic Integrator

Updated links U'
Potential: S(U')



Updated momenta: π'
Kinetic energy: $(1/2)\,\pi'^2$

$$H' = (1/2)\pi'^2 + S(U')$$

Metropolis
Acceptance
Test:

$$P_{acc} = \min\left(1, e^{-\Delta H}\right)$$

$$\Delta H = H' - H$$

Capacity v.s. Capability

- Gauge Generation:
  - ~5000-10,000 MC Updates, use ~500-1000 configs for analysis
  - ~600-1000 solves per MC Update -> 3M - 10 M solves
  - MC Update process is sequential
  - Capability level computing is needed for timely progress
- Stage 1 Analysis:
  - Distillation Technique: current 'small' dataset 31M solves
  - Putative 32^3x256 dataset (300 cfgs, 192 ev/cfg): 118M solves
  - As much as 10x more solves than gauge generation
  - BUT
    - Task parallel, and batches of solves use the same config
    - worth computing costly preconditioner. or deflation space

Multi-Shift Solvers:

- Multi-Shift Solvers used to evaluate rational approximations in partial fraction form:

$$R(x)\ \phi = A \sum_i p_i \boxed{\left(M^\dagger M + q_i\right)^{-1} \phi}$$

- Multi-Shift Systems typically use:
  - Single Krylov Process for all Shifts
  - Initial guesses for all shifts must be parallel (usually 0)
  - This is a difficulty for Inner/Outer/Restarted Schemes
- Use Polynomial Approximation (don't use shifted solver)
- Use Single Mass Solver separately for each shift
  - All single mass accelerations + intelligent guesses for solutions of the shifted systems
  - Alexandru reports > 2x speedup on GPUs (arXiv:1103.5103)

What Else Do We Need?

- For Basic Gauge Generation one also needs
  - Gauge and Fermion Actions, MD Forces on the GPU
  - Link Smearing (e.g. Stout/HEX/etc) on the GPU
    - SU(3)xSU(3) matrix multiplication routines
    - Nearest and Next to Nearest Neighbor access

- Non-solver work can take between ~5-35% of runtime on CPU
  - Depending on your situation Amdahl's law may/may not bite.
- Progress from several groups:
  - Gauge Action + Link Fattening used by MILC in QUDA
  - BMW Group has full HMC implementation on GPU