SciDAC Software
JLab AHM May 6, 2011

Possible Topics for Discussion

New Machines:

BG/Q & CPS (Mawhinney)
Blue Waters (Gottlieb)
GPU & Chroma (Joo)
Strong Scaling GPU (QUDA group)

New Software/Algorithms

QDPOP Multi-grid (Osborn)
QULA (Pochinsky)
QUDA slides (Babich at Software Workshop)

Mawhinney

CPS Status - Clusters

- Can call Pochinsky's cgDWF and mDWF from CPS (Jung, Yin)
- mDWF uses compiler for vectorized SSE instructions, unlike cgDWF
- Compare compilers on D_s nodes for mDWF (thanks to Don Holmgren for getting INTEL compiler very quickly)
  * With gcc: 0.738 sec/cg iteration for $L_s = 12$
  * With INTEL: 0.508 sec/cg iteration for $L_s = 12$ (Yin)
  * Speed-up of 1.45
- Single/double precision cgDWF running in CPS now on D_s (Izubuchi)
- Pochinsky's code uses 4d preconditioning, and see iteration count is ~20% higher, for a given accuracy, than the iteration count for 5d preconditioning used in native CPS codes
- This means 20% faster cgDWF/mDWF codes do not change time to solve

CPS Status - BG/Q

- Basic programming model: 1 MPI process per node, then threading to access the 64 threads/node that are available
- DWF inverter (Boyle)
  * Hand and Bagel optimized code running on prototype.
  * Code used in chip development and debugging
  * Respectable current performance, with improvement expected
  * Doing Kl3 physics on prototype with DWF+ID ensembles
- RHMC (Jung)
  * Using OpenMP to thread most critical parts of force calculations in RHMC
  * Basically ready to run evolutions on prototype
  * DWF thermo evolutions in very near future
- Access for other groups to prototype in process (Christ)

MADWF Inverter

Mobius Accelerated DWF Invertor (R. Mawhinney and H. Yin)

- Example: solve for DWF propagators on $L_s = 32$ lattice by doing intermediate solves with properly tuned Mobius fermions on $L_s = 12$ lattice

CG From zero init guess	CG via Mobius		
11290	16		
	121	4447	106
	101	4581	106
	102	4775	106
	517		
	equivalent:6351		
2.68e3 seconds	1.14e3 seconds		

- Maps DWF to 4d overlap version of DWF. Approximate this by 4d overlap of Mobius with smaller L_s
- Do much of solve in single precision Mobius using mDWF (Pochinsky)
- Requires multiple DWF <-> Mobius translations, intermediate DWF solves with $m_f = 1$.
- Operation count cut by 2x. Time to solve speed-up: BG/L, 2.7x; BG/P, 2.3x, D_s 1.5x (may improve)

Gottlieb

Blue Waters

- USQCD has a PRAC grant from NSF to support travel to NCSA and interaction with NCSA staff in preparation for Blue Waters, NSF's sustained petascale computer
- Public schedule: early science in late 2011, full service mid-2012

- Greg Bauer is our primary point of contact and has been a source of great help.
  - He has access to prototype hardware and has been running both Chroma and MILC codes.
  - Some of us have access to Blue Drop (Power 7 system at NCSA).
- Performance information is still NDA, but we can list some of the activities.

Chroma

- Code has compiled and been run.
- Variation on performance depending on local volume
  - data padding needed?
- NUMA issues are a concern
- VSX/VMX routines (see MILC) should be easy to integrate.

CPS

- Peter Boyle has NDA access through Edinburgh.
- He plans to port BAGEL to Power 7.

MILC

- SG spent 2009-10 sabbatical at NCSA.
- VSX/VMX routines written by Brad Elkins (IBM) tested by Greg Bauer.
- SMT tested. SMT=2 provides most of the gain. (SMT=4 is maximum.)
- Derived datatypes reduce copying of data to and from MPI buffers.
  - EuroMPI paper by Hoefler and Gottlieb
  - Now tested on BW prototype

- MILC code (su3_rmd) has been run on up to 256 cores of prototype hardware by Greg Bauer.
  - various options tested
- Performance model by Gottlieb and Hoefler
  - model is public; BW parameters are not
- Independent model by Hoisie et al.
  - It's late and have not seen a write-up.

- With so many cores/node, a hybrid OpenMP and MPI program may get better performance.
- This is expected for BlueGene/Q
- Doug Toussaint has been trying this approach on Hopper (NERSC Cray XE6)
- Greg Bauer is trying this on BW prototype.

Joo

Status: Chroma+QUDA

- Chroma wraps the QUDA Clover solvers
  - QUDA Propagator, two flavor & shifted solvers
- Chroma HMC trajectories possible with solver work on GPU
- Multi-Dimensional parallelization of QUDA has now happened
  - Wilson/Clover/AsqTAD (so far) -- (clover only in Chroma)
  - PCIe still choke point
    - Additive Schwarz Domain Decomposed Solver helped Clover inverter (usefully) scale to 256 GPUs
  - Some cleanup needed in some of the wrappers
    - Multi-Dim work changed QUDA interfaces...
  - Lots of work out there to do (volunteers?)

Current & Future Efforts at JLab

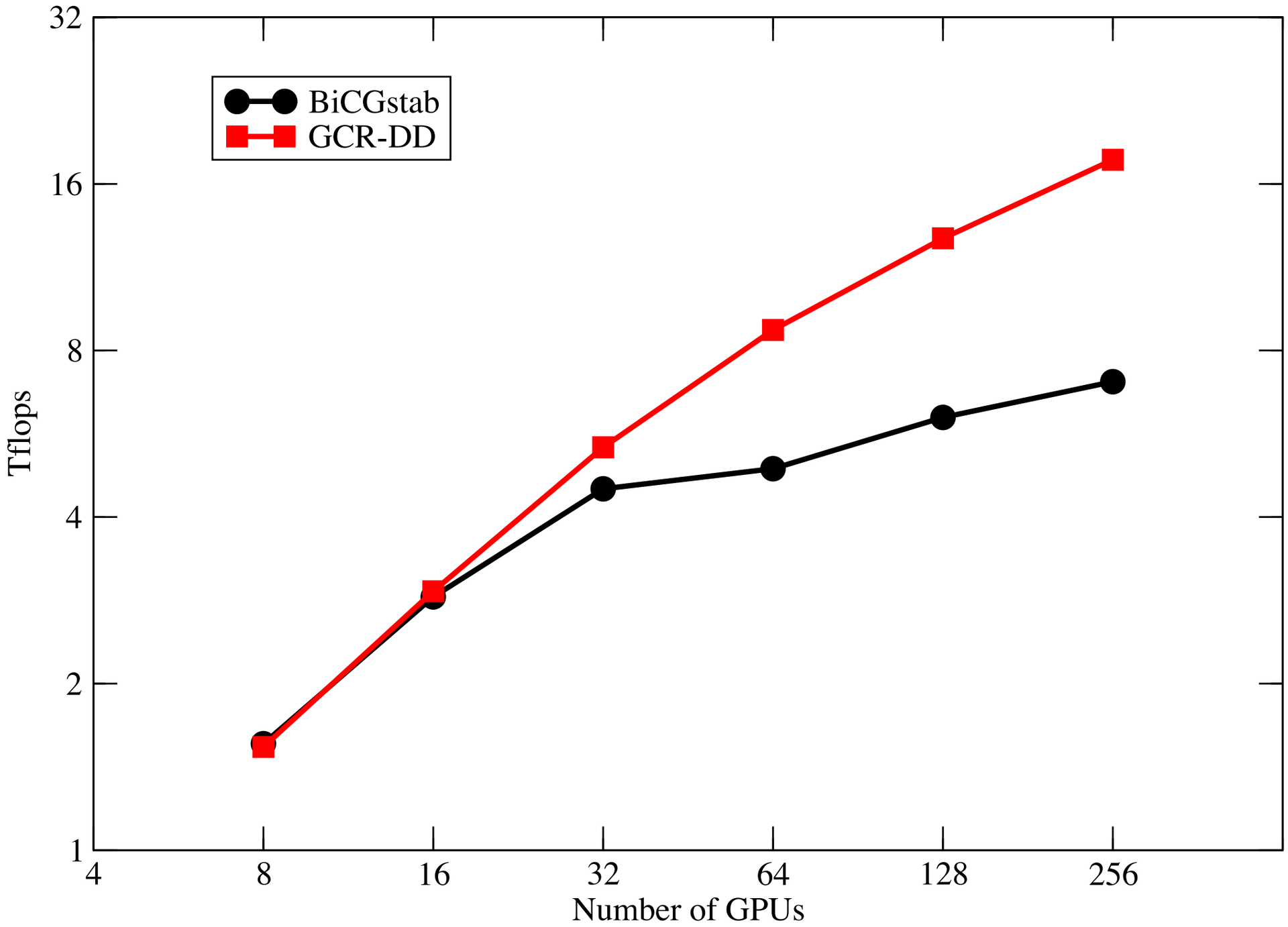
- 'General Computing' R&D
  - Porting/Optimization for Emerging Systems
    - BlueWaters, BG/Q
  - Direct QDP++ support for GPU/Heterogeneous systems
  - A lot of work in the invisible plumbing (beneath Chroma)
    - e.g. Optimized Dslash-es, Clover Terms, solvers etc.
  - Architectural Exploration Work
    - E.g. CUDA-4.0, PCIe networks, Intel Knights series, future NVIDA GPUs, Intel AVX etc.
  - Algorithmic Work
    - Scalable Solvers (e.g. Domain Decomposed, Mixed Precision, Multi-level etc)

Current & Future Efforts at JLab

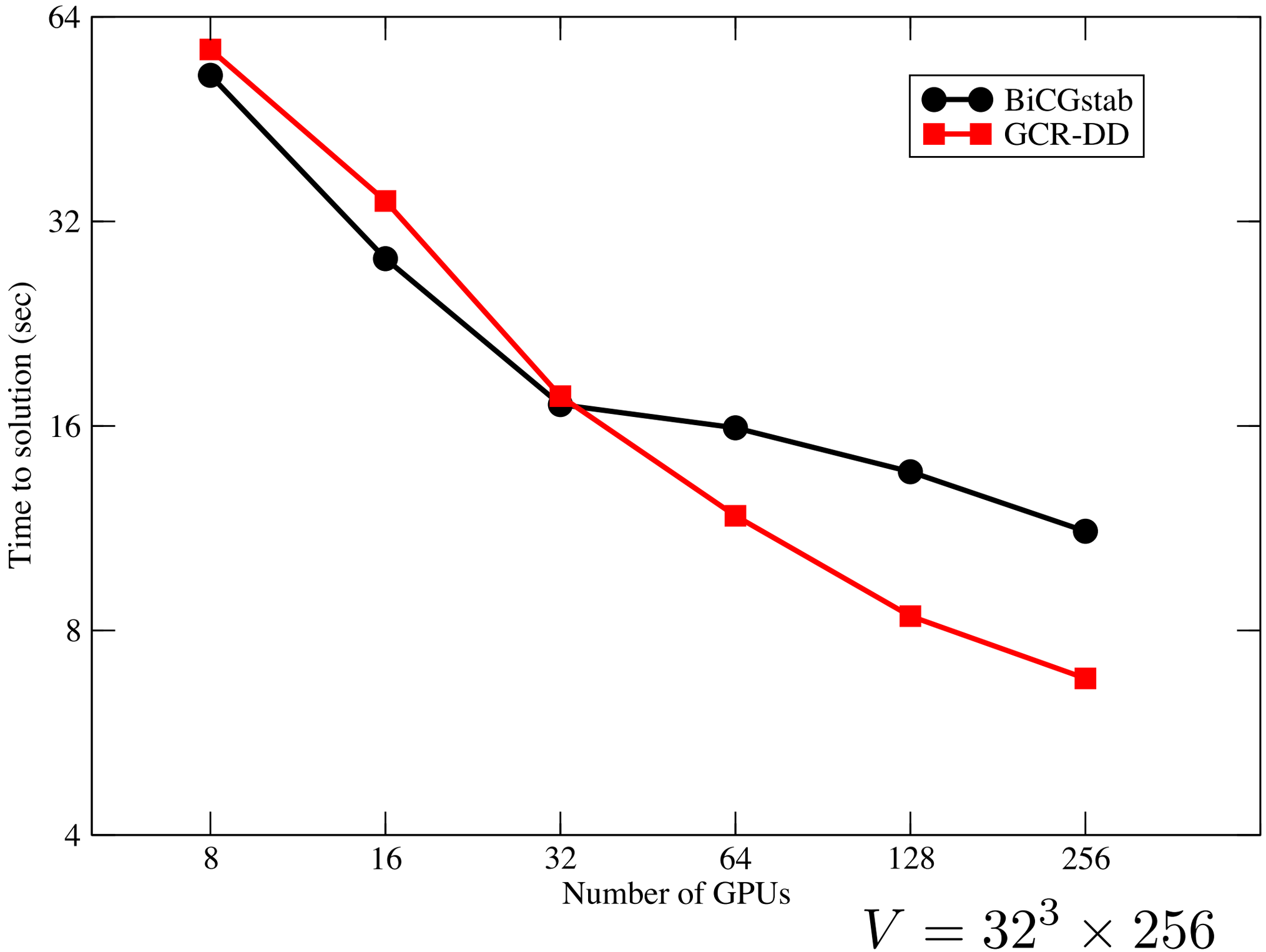
- Analysis Methods R & D:
  - Improvements on Distillation
    - Better Smearing Techniques
    - Hybrid Distillation/Stochastic (Noisy Methods)
    - Very large Q^2 form-factors
    - Changes to our workflow: Will need infrastructure support: tape libraries, parallel I/O, etc
  - Software
    - Három -- A 3D code for contractions
    - Redstar -- Compute 2pt-3pt correlation functions using the output of Három

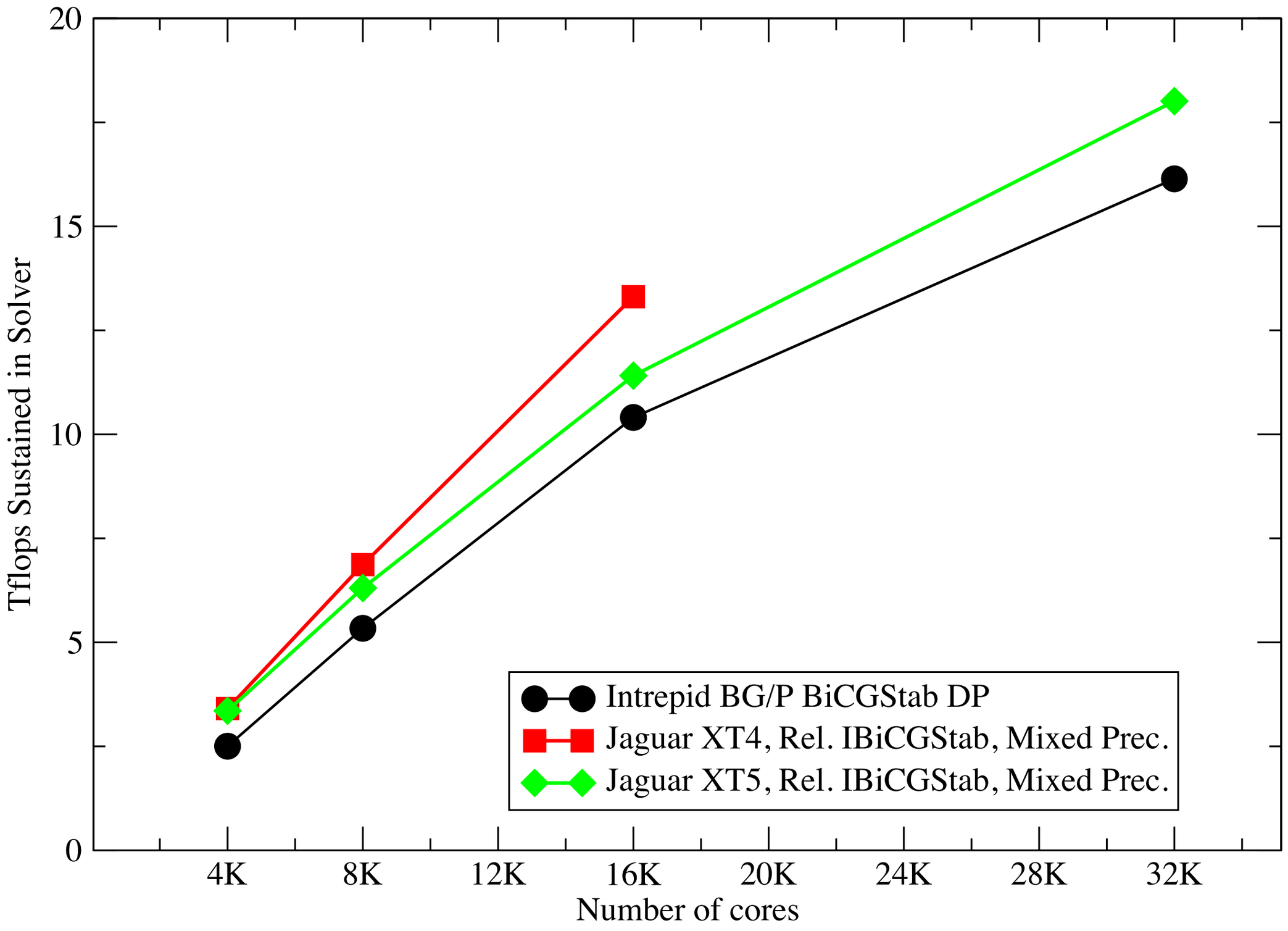
GPU Strong Scaling

(Babich, Clark, Joo, Shi, Brower &Gottlieb)

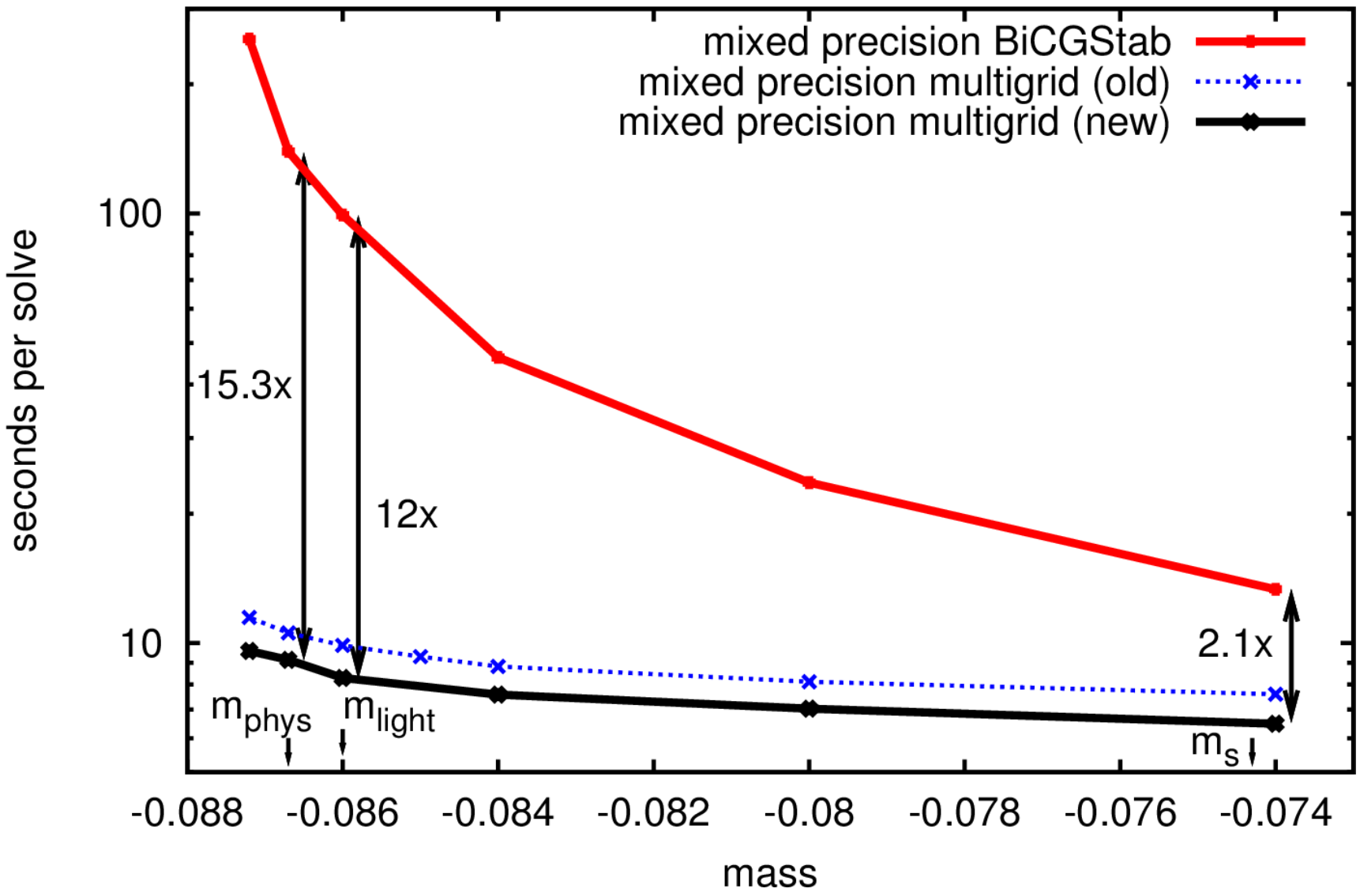


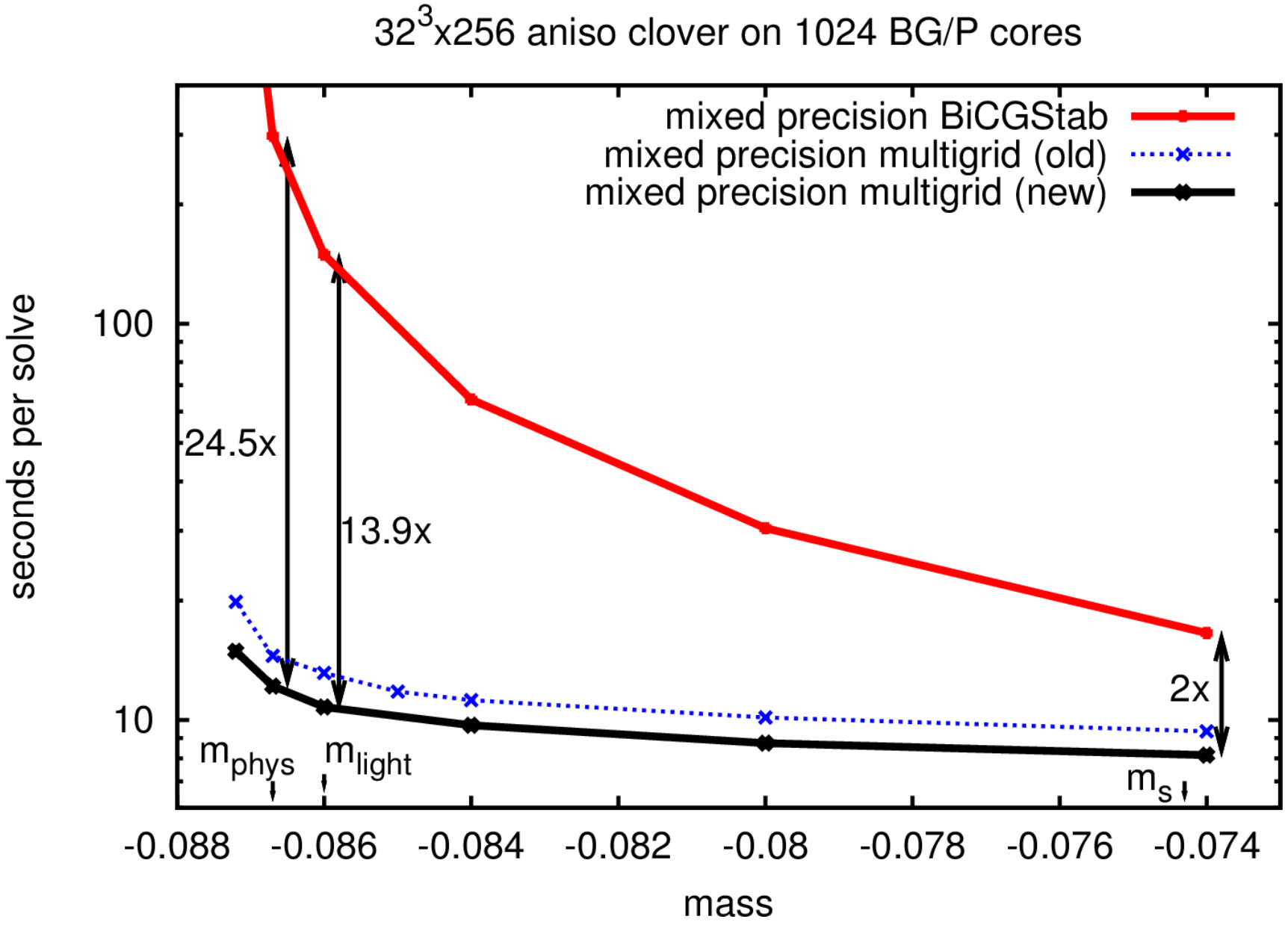
$$V = 32^3 \times 256$$

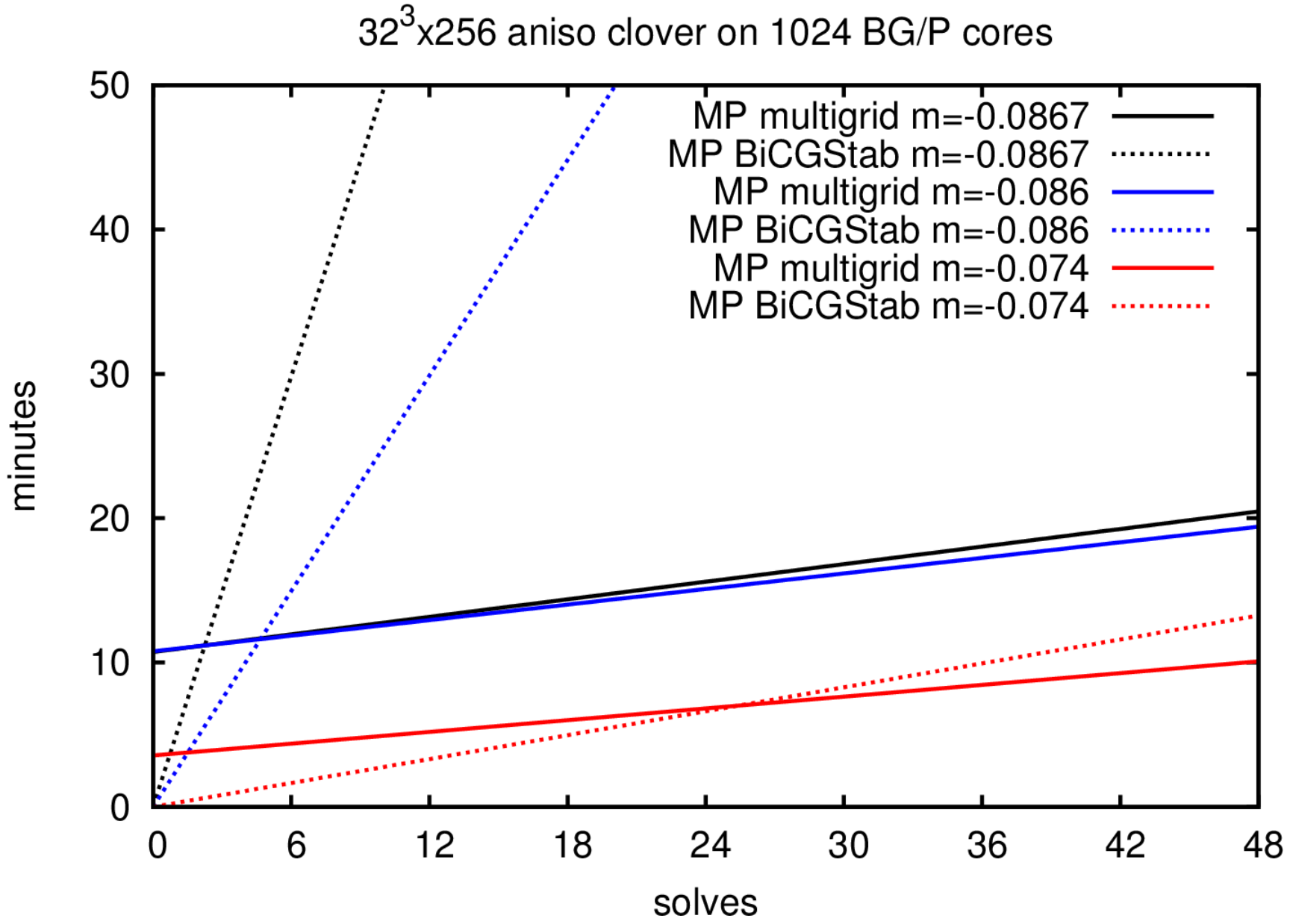

64
32
16
8
4
Time to solution (sec)
8
16
32
64
128
256
Number of GPUs
BiCGstab
GCR-DD
$V = 32^3 \times 256$


Tflops Sustained in Solver
20
15
10
5
0
4K
8K
12K
16K
20K
24K
28K
32K
Number of cores
Intrepid BG/P BiCGStab DP
Jaguar XT4, Rel. IBiCGStab, Mixed Prec.
Jaguar XT5, Rel. IBiCGStab, Mixed Prec.

Osborn


24^3x128 aniso clover on 256 BG/P cores
mixed precision BiCGStab
mixed precision multigrid (old)
mixed precision multigrid (new)
seconds per solve
100
10
15.3x
12x
2.1x
m_{phys}
m_{light}
m_s
-0.088
-0.086
-0.084
-0.082
-0.08
-0.078
-0.076
-0.074
mass


32^3x256 aniso clover on 1024 BG/P cores
mixed precision BiCGStab
mixed precision multigrid (old)
mixed precision multigrid (new)
seconds per solve
100
10
24.5x
13.9x
2x
m_{phys}
m_{light}
m_s
-0.088
-0.086
-0.084
-0.082
-0.08
-0.078
-0.076
-0.074
mass


32^3x256 aniso clover on 1024 BG/P cores
MP multigrid m=-0.0867
MP BiCGStab m=-0.0867
MP multigrid m=-0.086
MP BiCGStab m=-0.086
MP multigrid m=-0.074
MP BiCGStab m=-0.074
minutes
solves
0
10
20
30
40
50
0
6
12
18
24
30
36
42
48

Argonne NATIONAL LABORATORY

Q* "C" libraries

James C. Osborn

USQCD software meeting

FNAL, Feb. 2011

Outline

- QMP
- QIO
- QLA
- QDP
- QOPQDP
- (Q)MG
- QLUA

QMP

- Latest version 2.4.0-a3 (source in Jlab git, devel branch)
  - 'a' is for alpha, but should be stable
  - Could still be changes to new features
- New features:
  - Major reorganization, allows easier plug in of specialized code
  - Added communicator support
  - Indexed memory type
  - Job geometry (Carleton)
  - Change address (Chulwoo)
  - declare_send_recv_pairs?

QIO

- Latest version 2.3.8 (since 3/4/09) (in Jlab git and cvs, switch to git?)
- Lattice volume formats:
  - SINGLEFILE (serial or parallel)
  - MULTIFILE
  - PARTFILE
  - PARTFILE_DIR (volNNNN/<filename>) (Sergey Syritsyn, private version)
- Possible improvements:
  - Singlefile parallel with variable number of I/O nodes (also do I/O contiguously)
  - Layout.{node_number,node_index,get_coords} should take user supplied arg:

  add void *layout.arg which calls without arg if it is NULL?
  - MPI-IO? (problem is that lime opens filehandle, MPI-IO probably belongs in QMP)

QLA

- Current version 1.7.0-a6 (in cvs, want to move to git)
- New features
  - Better arbitrary Nc support, using C99 variable length arrays requires working C99 compiler (some older gcc's have problems)
  - New functions including inverse, det, exp, sqrt, log of ColorMatrix
  - Full support for OpenMP (--enable-openmp)
- Plans
  - BG/Q (have some code pieces from IBM)
  - Blue Waters
  - Use BLAS/LAPACK?

QDP

- Current version 1.9.0-a7 (also in cvs → git)
  - Multi-lattice support (with user supplied layouts)
  - Siteloop macro

```
QDP_loop_sites(i, s, {
  QLA_ColorMatrix *mi = QDP_site_ptr_readwrite_M(m, i);
  QLA_DiracFermion *di = QDP_site_ptr_readonly_D(d, i);
  QLA_c_eq_r_times_c(QLA_elem_M(*mi,0,0), 2., QLA_elem_D(*di,1,2));
});
```

- Possible improvements
  - Complete multilattice shifts (duplications and reductions)
  - OpenMP support (instead of in QLA)
  - Thread "groups" – can have independent groups of threads working on different tasks

QOPQDP

- Current version 0.16.4 (also in cvs → git)
  - Miscellaneous fixes and improvements
  - HISQ support in progress (Alexei & Carleton)
- Possible changes
  - Needs major update
    - Handle mixed precision gracefully (restructure headers)
    - Multiple lattices (currently initialized with fixed layout)
    - Other Nc (2 and arbitrary)
  - Add multigrid
  - Add eigensolvers

(Q)MG

- Working clover multigrid (10-20x)
- No particular home for code
  - Restriction and prolongation belongs in QDP
  - Rest probably belongs in QOPQDP
- Code works and in production, but still many opportunities for improvement
- Large number of parameters to tune
- Domain wall (Saul) and Staggered (James) in progress

QLUA

- Branch used for testing and development
- Now has
  - QOPQDP solvers (asqtad, HISQ, clover, DW)
  - Clover MG
  - Eigensolvers (asqtad, HISQ, DW)
- MG & eigensolvers are temporary hacks until QOPQDP supports them

Pochinsky

QLUA: LQCD SCRIPTING

- ☐ based on the Lua programming language
- ☐ integrated with QDP & QIO
- ☐ seamless Level III extensions
- ☐ platform-independent
- ☐ in production use

QLUA FEATURES

- [ ] Full programming language (variables, routines, &c)
- [ ] automatic memory management (including QCD data)
- [ ] Latent types
- [ ] Available Level III packages
  - [ ] Möbius Domain Wall Fermions
  - [ ] Clover Wilson (including EigCG)
  - [ ] QDPOP

QLUA POINTERS

- http://www.lua.org/
- https://lattice.lns.mit.edu/trac/downloads
- https://usqcd.lns.mit.edu/wiki/QLUA_tutorials

Babich

State of the QUDA Library

LQCD Software Workshop
February 25, 2011

Ron Babich
Boston University

on behalf of the QUDA developers & contributors:

Rich Brower, Mike Clark, Joel Giedt, Steve Gottlieb, Bálint Joó,
Claudio Rebbi, Guochun Shi, and Alexei Strelchenko

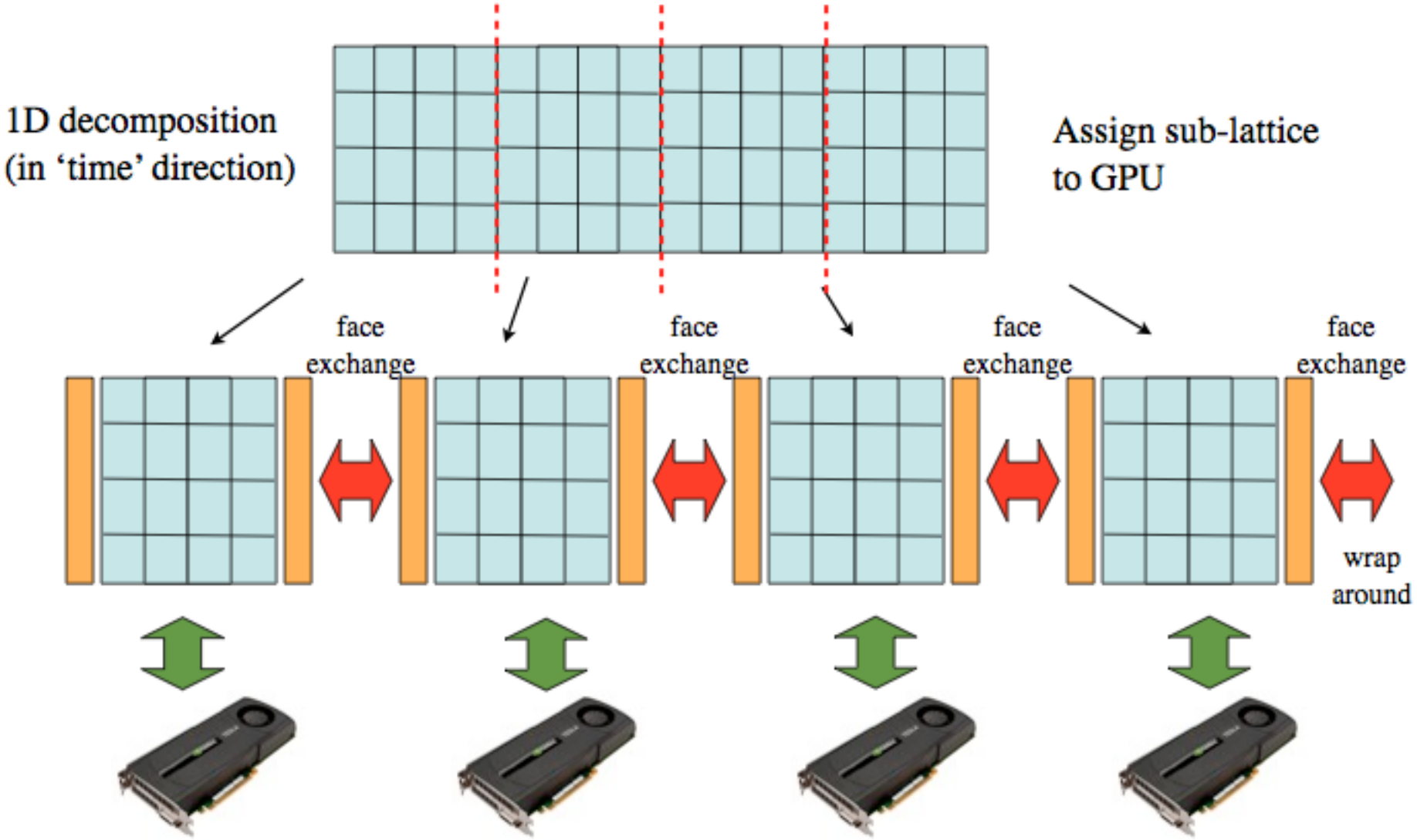
Current features

- Latest released version is 0.3.2.
- Includes single-GPU solvers for:
    - Wilson and Wilson-clover
    - Twisted mass
    - Improved staggered (asqtad/HISQ), but not yet naïve
    - Domain wall
- Both CG and BiCGstab are supported, including mixed precision with “reliable updates.”
- Staggered code does more besides (see update from Steve).

Unreleased features

- Available in public repository but not thoroughly tested:
    - Multi-GPU Wilson & Wilson-clover using QMP
    - Multi-GPU staggered using MPI
    - Dslash auto-tuning at runtime (in addition to existing compile-time “BLAS” auto-tuning)
    - Multi-shift solver generalized beyond staggered
- Available very soon (weeks):
    - Multi-GPU Wilson, Wilson-clover, staggered, and twisted mass to support both MPI and QMP, including reference host implementations.
    - Domain wall optimizations (and restructuring to facilitate multi-GPU)

Aside: Current multi-GPU implementation



Slide: Bálint Joó

Roadmap (continued)

- Available soonish (1-3 months):
    - Multi-GPU parallelization in more than one dimension, developed simultaneously for Wilson-like and staggered fermions.
    - Multi-GPU domain wall.
    - Interface overhaul? (See below.)
- Longer-term:
    - Additional routines: forces, quark smearing, eigensolvers, etc.
    - Multigrid for Wilson and Wilson-clover
    - Continued exploration of algorithms (domain decomposition, etc.)

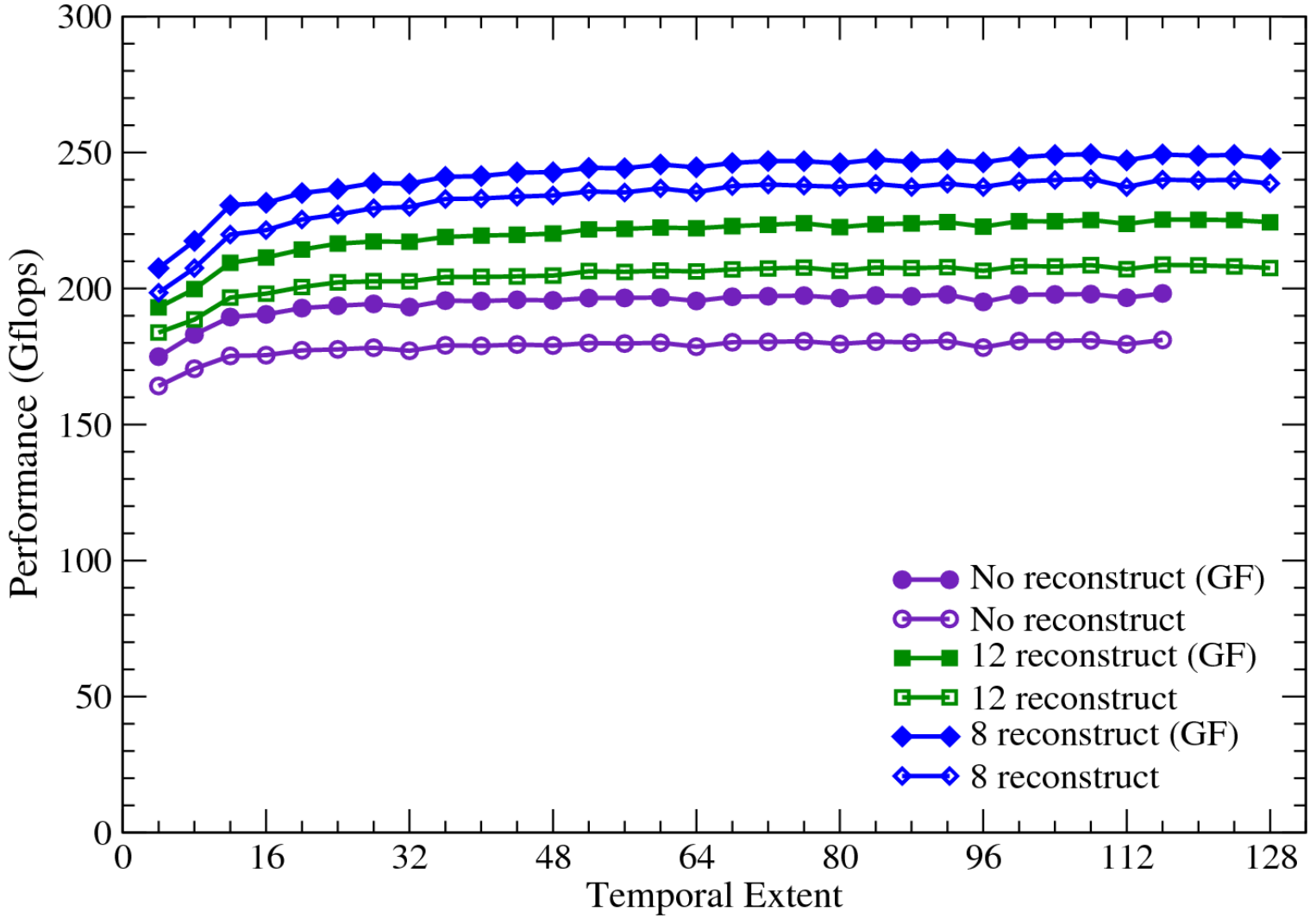
Performance update

- Results are for the even/odd preconditioned clover-improved Wilson matrix-vector product ("Dslash"):

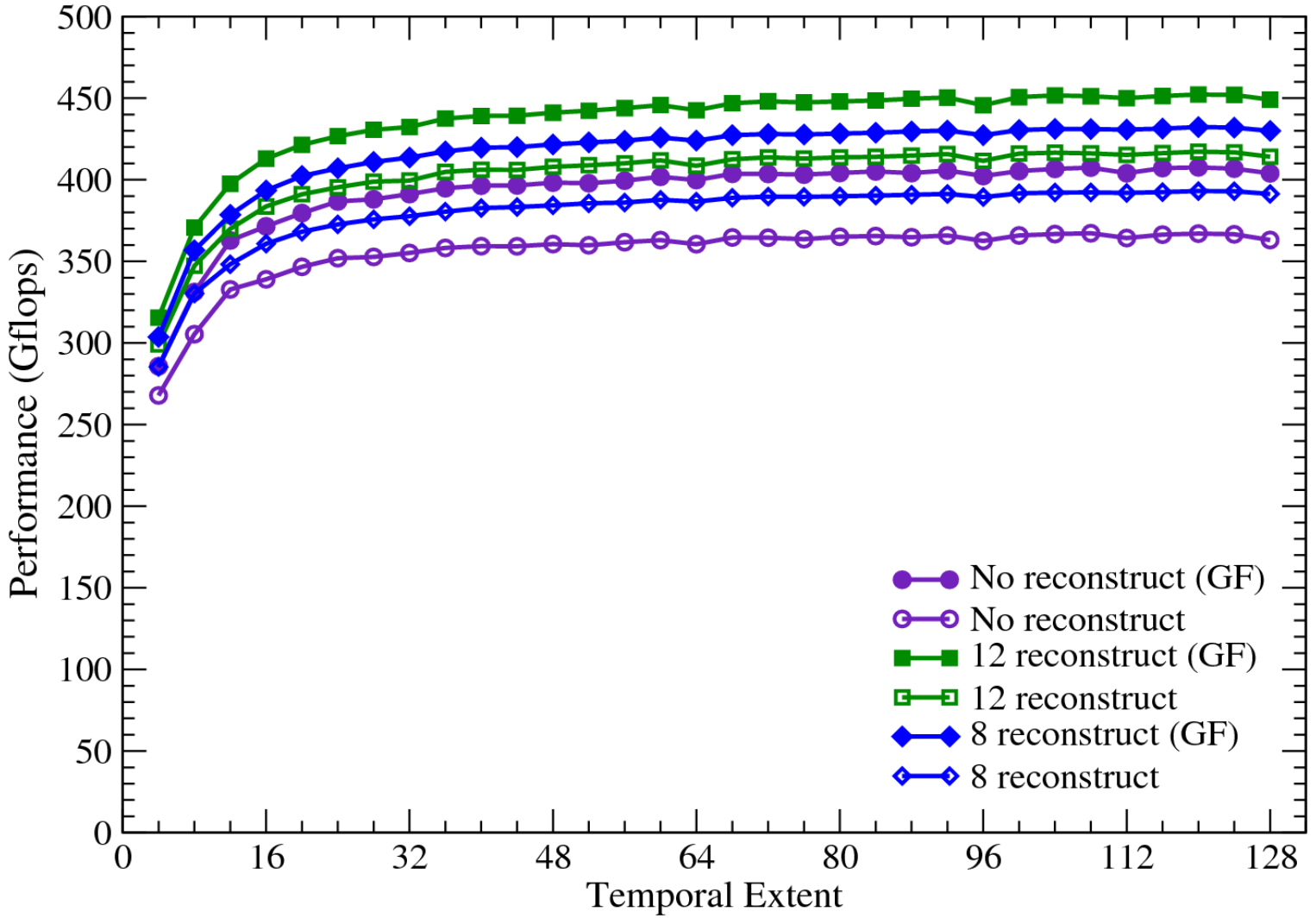
$$M = (1 - A_{ee}^{-1} D_{eo} A_{oo}^{-1} D_{oe})$$

- Runs were done on a GeForce GTX 480 (consumer "Fermi" card) with slightly out-of-date QUDA.
- Latest version includes Dslash auto-tuning and other enhancements, giving perhaps a 10% bump.
- For reference, a standard dual-socket node with recent (Westmere) quad-core Xeons would sustain around **20 Gflops** in single precision for a well-optimized Wilson-clover Dslash. The Ds cluster's quad-socket Opteron nodes do better than **40 Gflops.**
- Spatial volume is held fixed at 24^3.

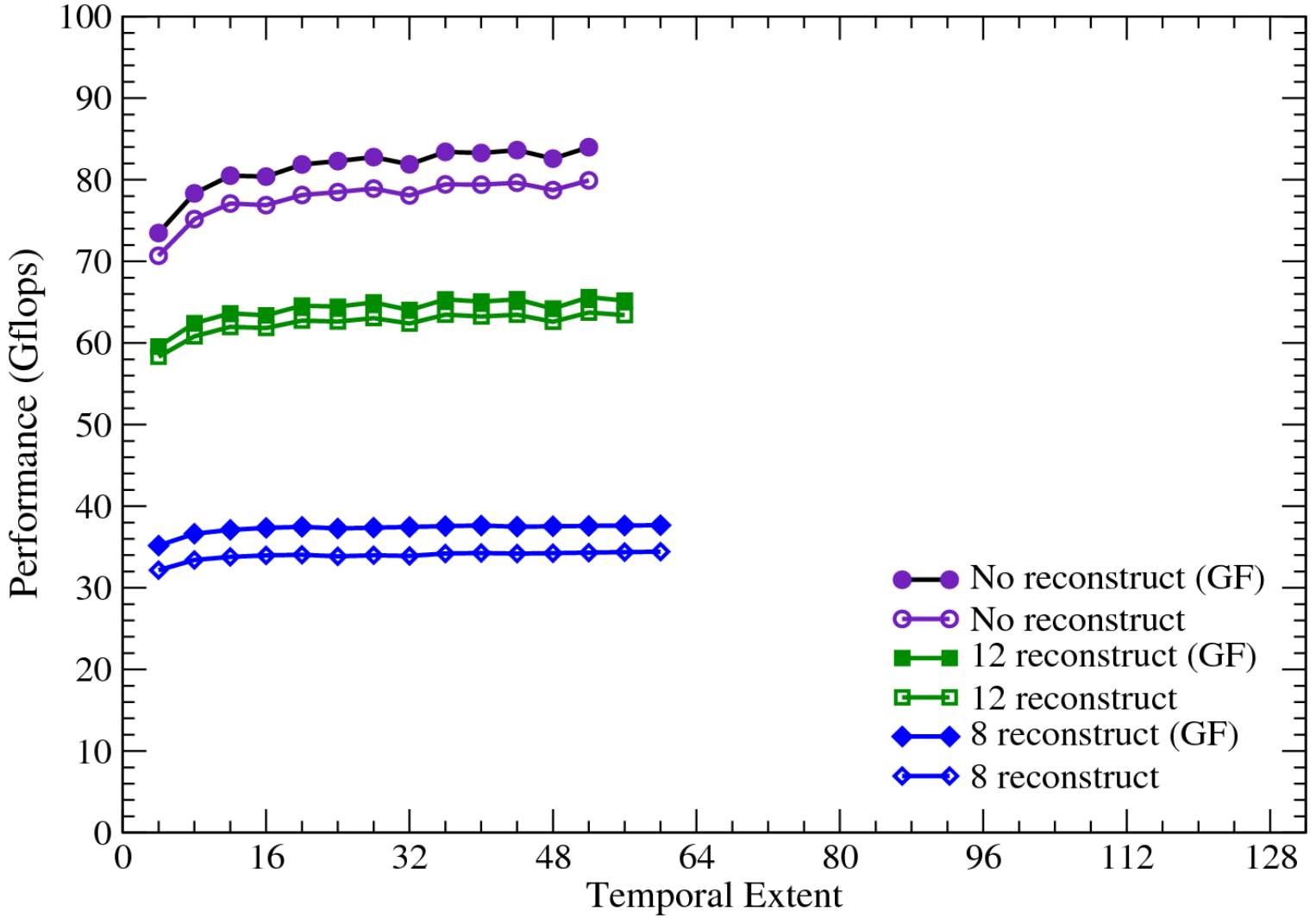
Clover performance (single precision)



Dslash performance (half precision)

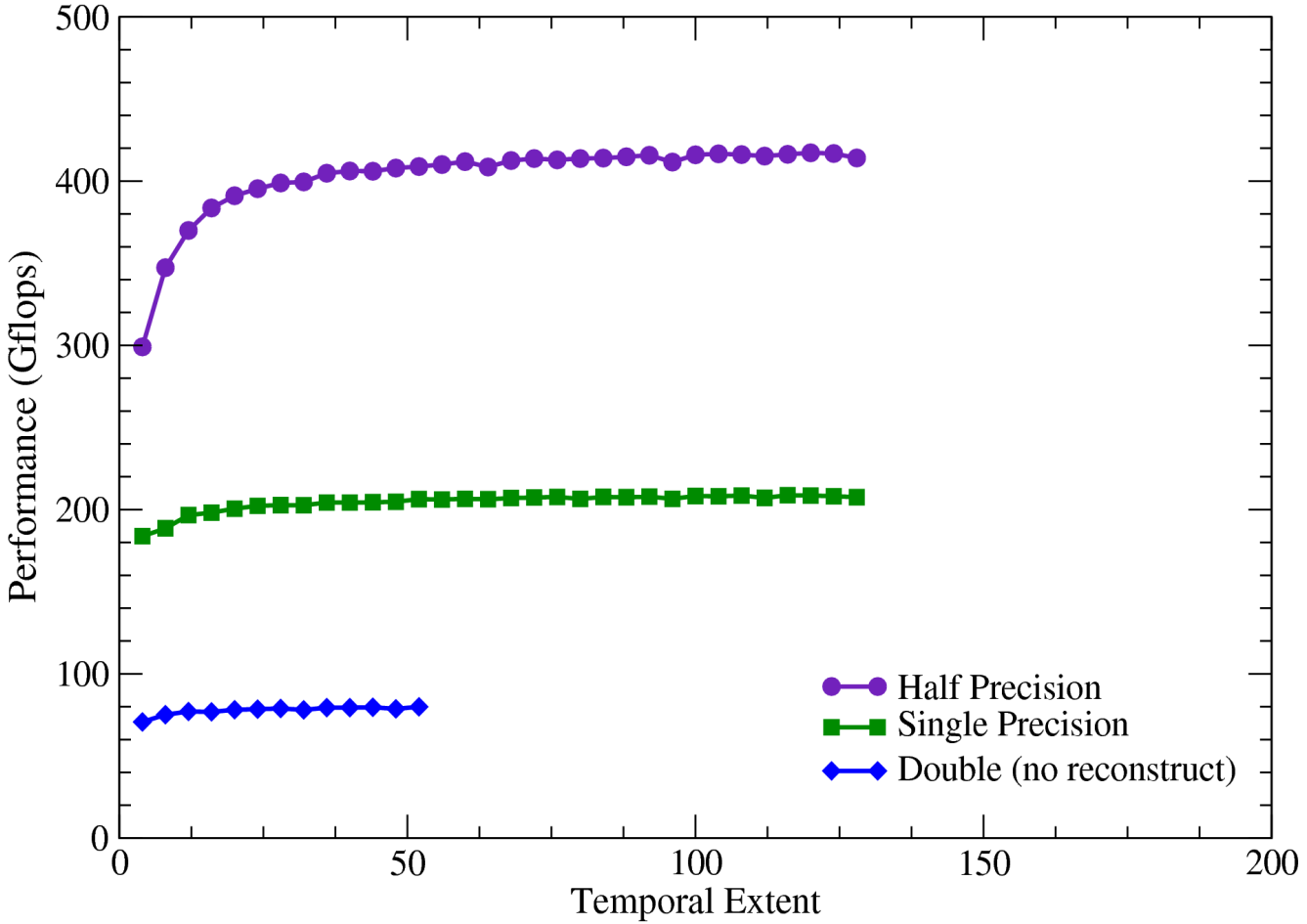


Dslash performance (double precision)



Dslash performance summary

- Summarized results are for a conservative case: 12-reconstruct (none for double) with no temporal gauge-fixing.



- Single and half performance are about 2.6x and 5.1x higher than double.

Logistics

- QUDA now lives at http://lattice.github.com/quda
- Future releases will be announced on the QUDA-announce mailing list. Join at the website or by emailing quda-announce+subscribe@googlegroups.com .
- Source code repository has been moved over to GitHub.
    - provides nice features such an issue tracker, source code browsing, commenting on commits, etc.
    - Git itself allows for more flexible collaboration and “parallel” code development.
    - Together these should make it easier for “outsiders” to contribute.

QUDA Interface Overhaul?

Current interface and limitations

- QUDA's interface is overdue for some changes.
- Currently, including `quda.h` gets you a handful of functions. Besides those for initializing and finalizing the library, the most commonly called are `loadGaugeQuda()` and `invertQuda()`.
- `loadGaugeQuda(gauge, gauge_param)` loads the gauge field onto the GPU.
- `invertQuda(solution, source, inv_param)` loads the source onto the GPU, performs the inversion, and returns the solution.
- Here `gauge`, `solution`, and `source` are all pointers to fields on the ***host***.
- There is no provision for manipulating fields on the GPU, without diving into the QUDA's internals.

Possible improvements

- **Disclaimer:** This is just a straw-man / personal wish list. Detailed mechanics, naming conventions, etc. have yet to be discussed by the QUDAnauts.
- We probably need something like the following:

```
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 source, solution, etc. are opaque “objects” that know about fields on the GPU.

Rationale

- This lets the user pipeline a series of operations on the GPU, without moving data back and forth. For example:

```
QUDA_load_site_field(source, host_source, spinor_order);
QUDA_smear(smeared_source, source, smear_param);
QUDA_solve(solution, smeared_source, u, solver);
.
.
.
QUDA_save_site_field(solution, host_solution, spinor_order);
```

- Supports multiple lattices and general “link fields” needed by multigrid
- More modular, making it possible to cooperate with user GPU code (next slide)

Interoperability

- QUDA completely insulates the user from GPU code. In principle, it should even be possible to support backends besides CUDA (e.g., OpenCL).
- One can imagine cases, however, where a savvy user would want to access GPU fields owned by QUDA. Something like the following might work:

```
#include <quda.h>
#include <cudart.h> // CUDA runtime library
#include <quda_expose_cuda.h>
. . .

float *prop[12]; // to hold pointers to GPU memory
QUDA_solve(solution, smeared_source, u, solver);
prop[0] = QUDA_expose_cuda_site_field(solution);
. . .
// user CUDA kernel
contract<<<nblocks, nthreads>>>(nucleon2pt, prop);
. . .
```

Challenges of multi-GPU strong scaling

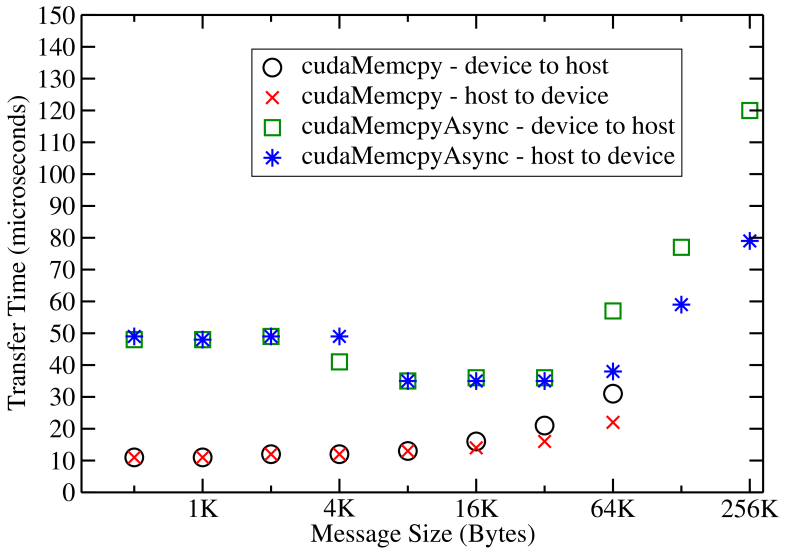
Multi-GPU motivation

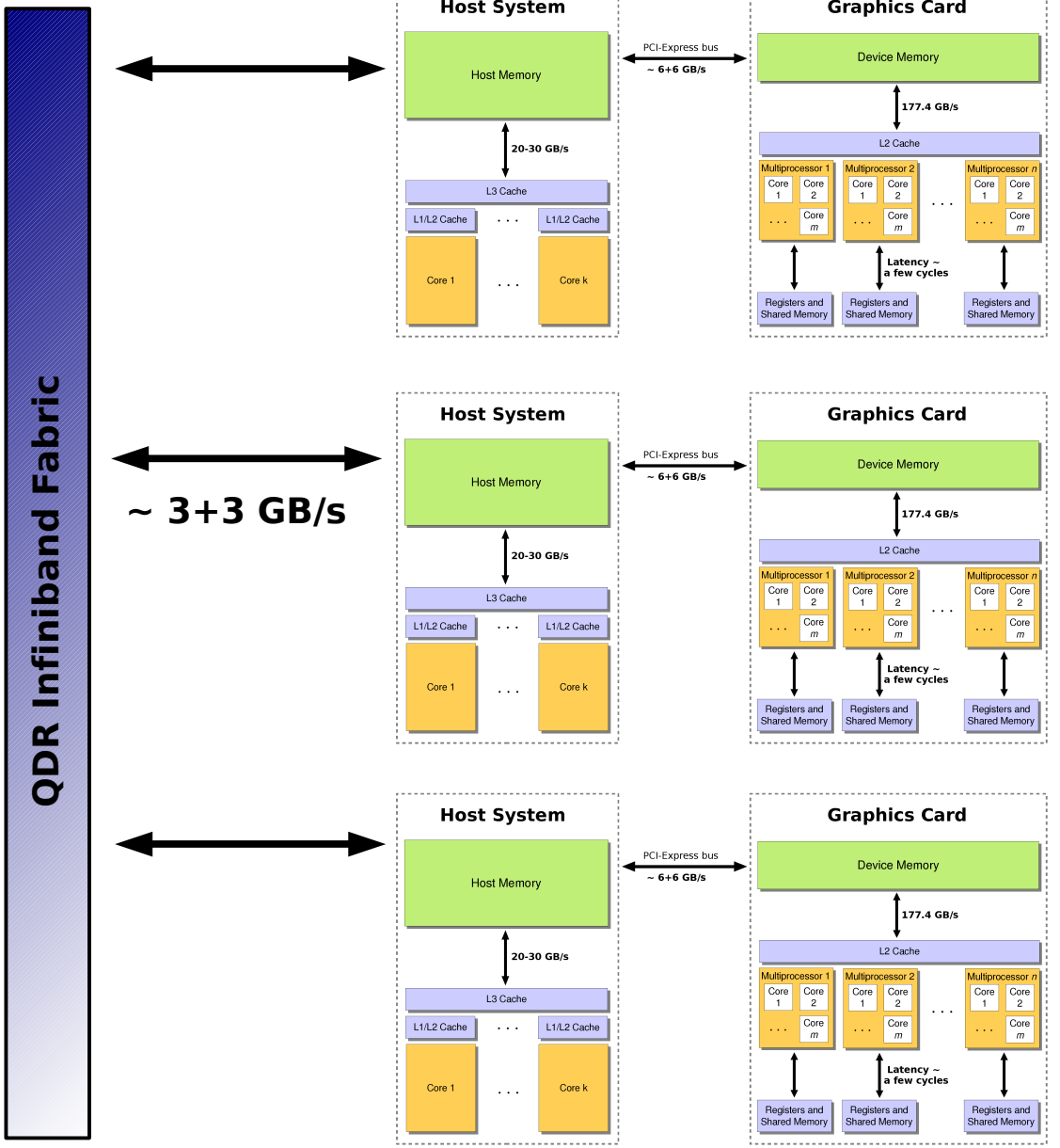
- **GPU memory:** For throughput jobs (e.g., computing propagators), it suffices to use the smallest number of GPUs that will fit the job, but often one GPU isn't enough.
- **Host memory:** It's generally most cost-effective to put more than one GPU in a node. These can be used in an embarrassingly parallel fashion (by running multiple separate jobs), but then host memory becomes a constraint.

- **Capability:** We'd like to broaden the range of problems to which GPUs are applicable (e.g., gauge generation). Here *strong scaling* is key.

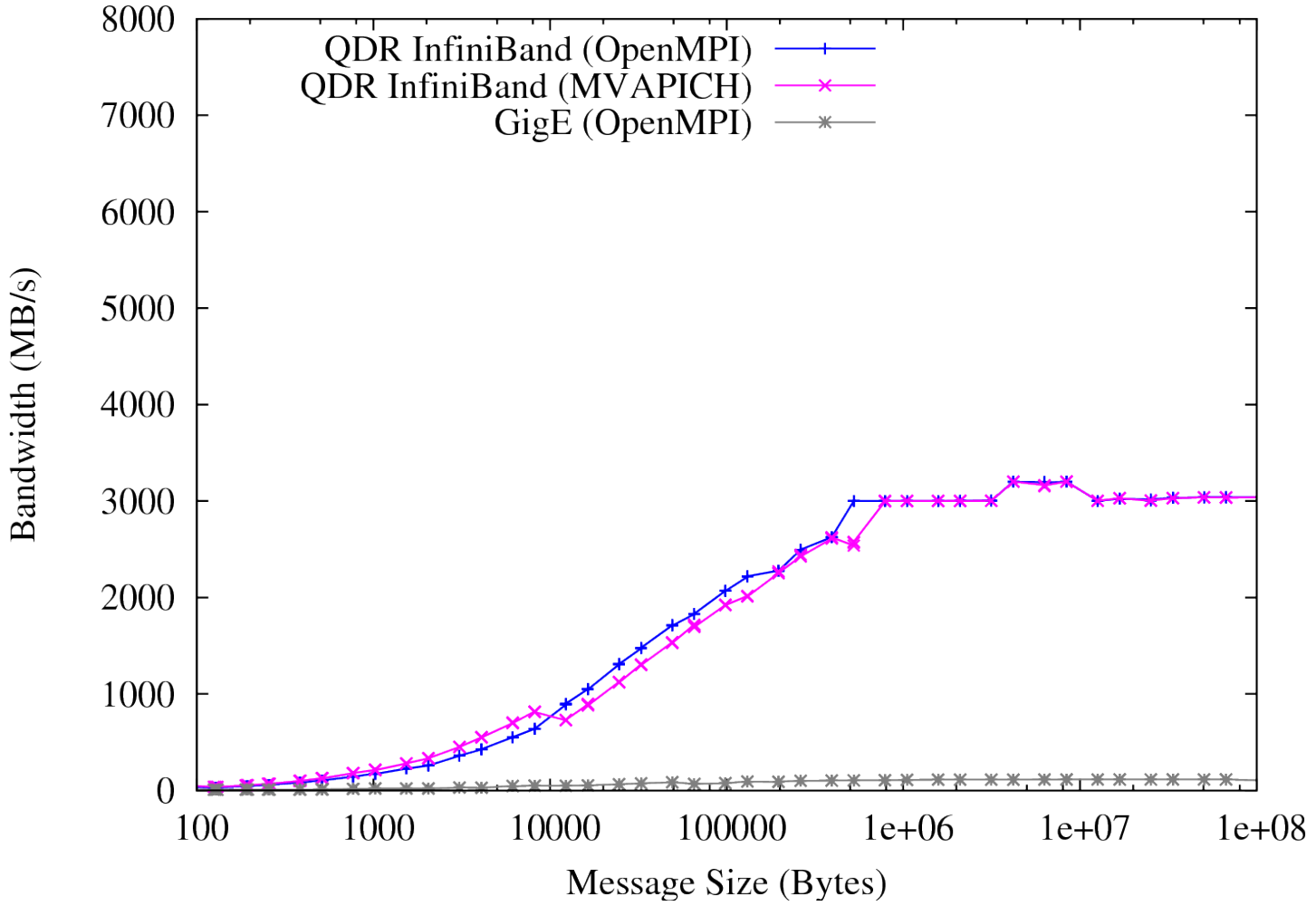
Challenges to scaling up

- GPU-to-host and inter-node **bandwidth**
- GPU-to-host and inter-node **latency**



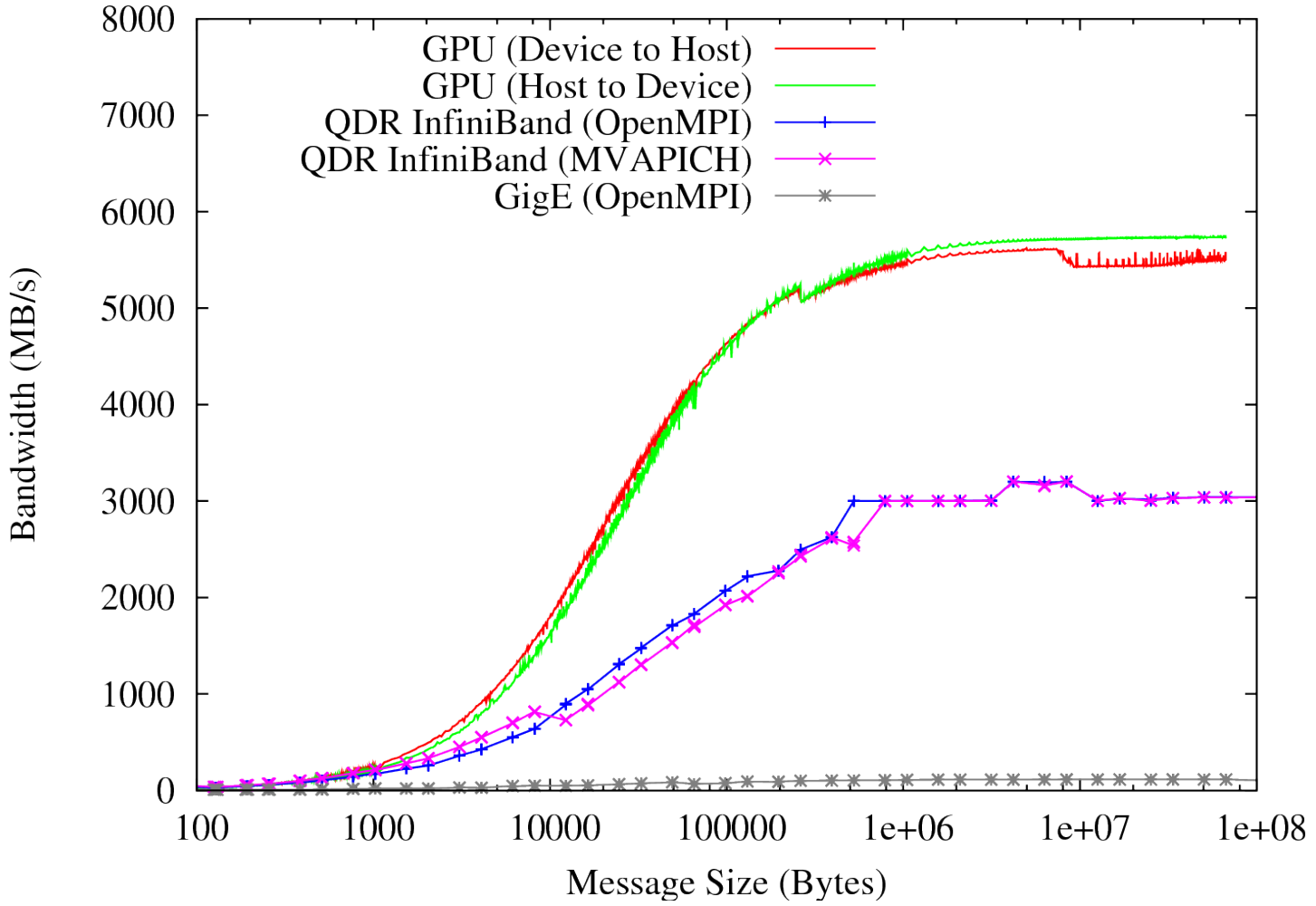


Interconnect bandwidth



(MPI bandwidth as measured by NetPIPE v3.7.1)

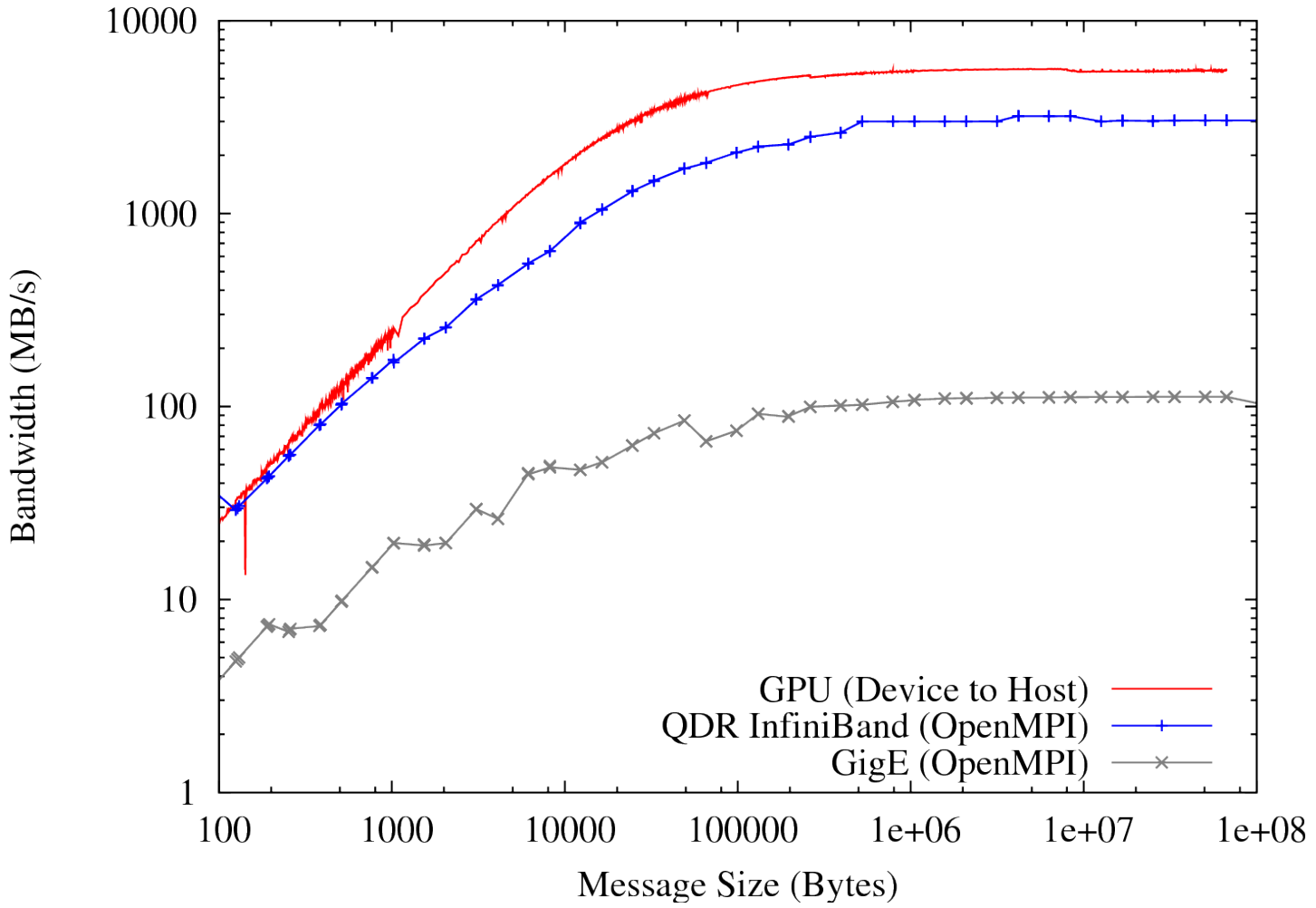
GPU bandwidth



(GPU bandwidth measured with CUDA SDK v2.3)

"`bandwidthTest --memory=pinned`"

Bandwidth (log scale)



Performance model

- For the Wilson matrix-vector product, we have:

$$(1320 \text{ flops/site}) \times (L^4/2 \text{ flops}) = 660L^4 \text{ flops}$$

$$(24/2 \times 4 \text{ bytes/boundary site}) \times (8L^3/2 \text{ sites}) = 192L^3 \text{ bytes}$$

$$\frac{660L^4}{\text{Perf}} = \frac{192L^3}{\text{Bandwidth}}$$

$$\text{Bandwidth [MB/s]} = \frac{0.29(\text{Perf [Mflop/s]})}{L}$$

$$\text{MessageSize [Bytes]} = 24L^3$$

Inspired by Gottlieb (2000), http://physics.indiana.edu/~sg/pcnets/

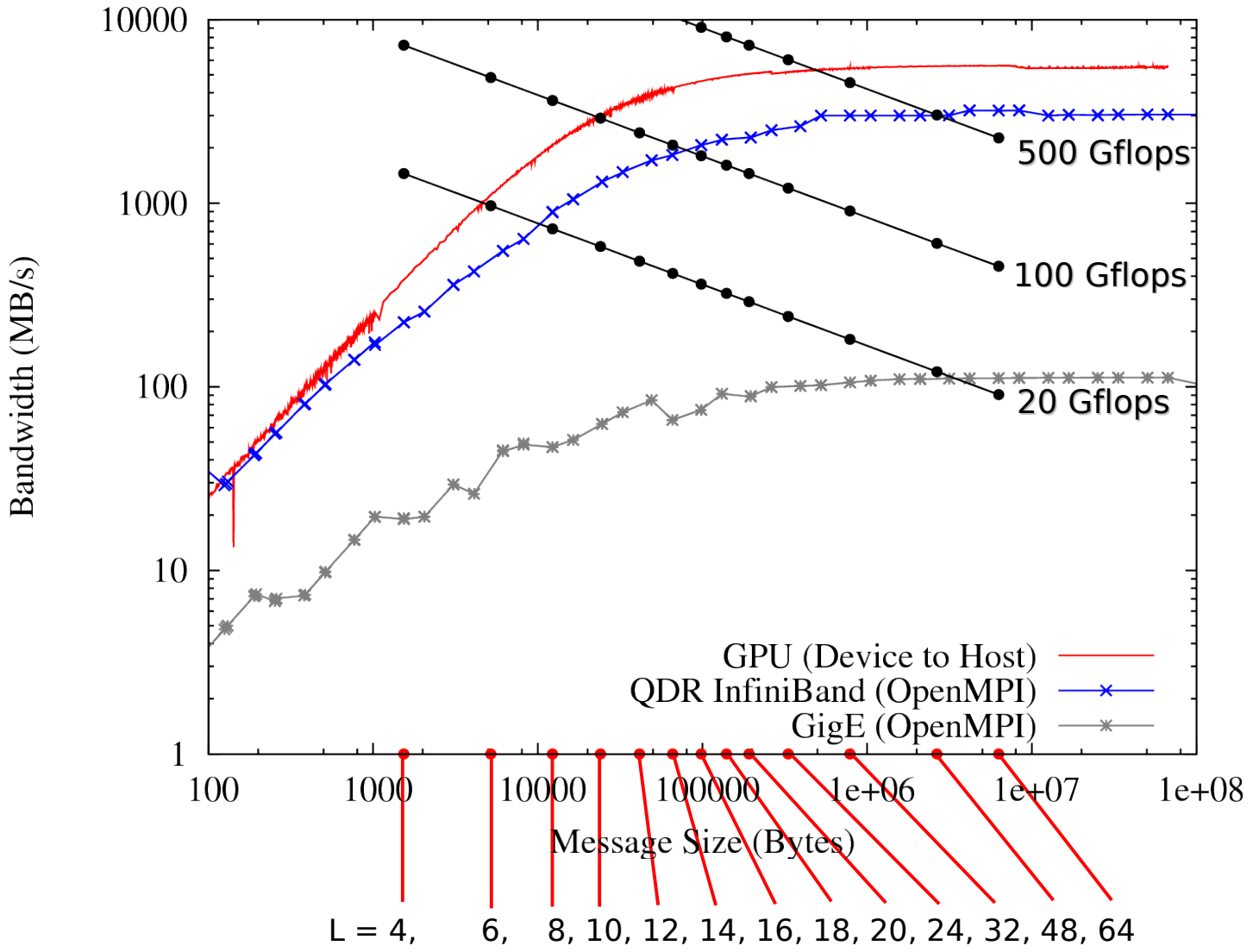
via Holmgren (2004), arXiv:hep-lat/0410049

Performance model

- This model is pessimistic in the sense that it assumes we are going to parallelize in all 4 dimensions. For small numbers of nodes, this is never optimal.
- It is optimistic in all other respects (assuming perfect overlapping of communication and computation, for example), telling us the best we can possibly do.
- For this exercise, we're interested in the *strong scaling* regime (smallest possible sub-volumes). How small can we go before the surface/volume ratio limits us?

- This is just a model, but I don't see any loopholes without resorting to new algorithms.
- We'll have actual experience very soon.

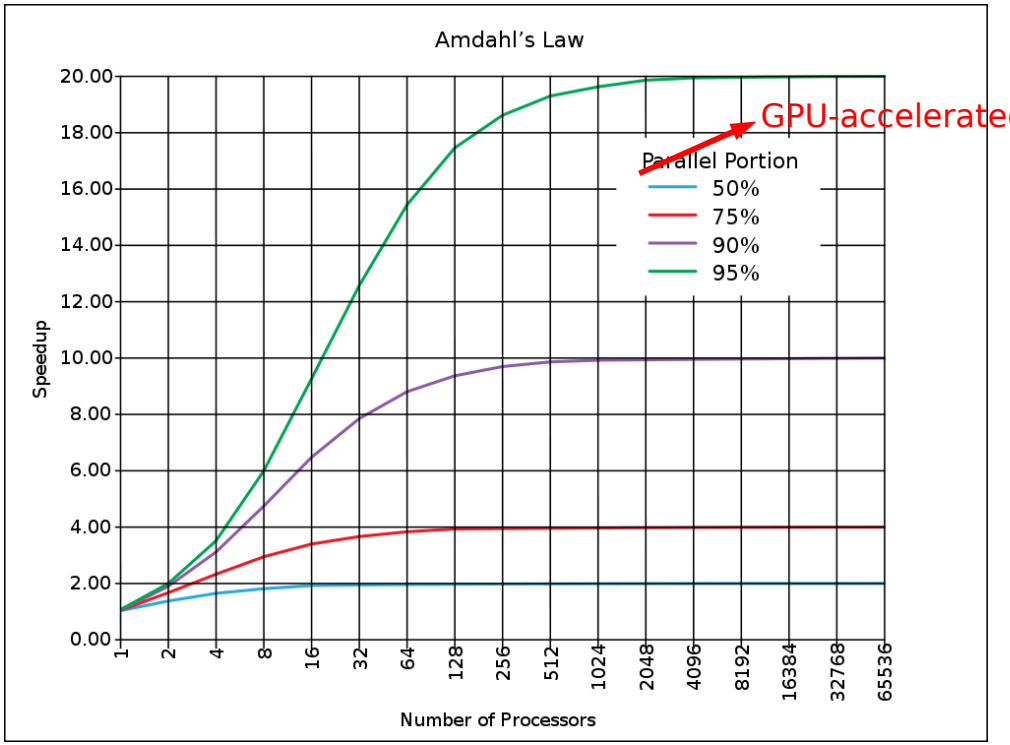
Required Bandwidth



A data parallel framework for QCD on GPUs?

Challenge: Amdahl's law

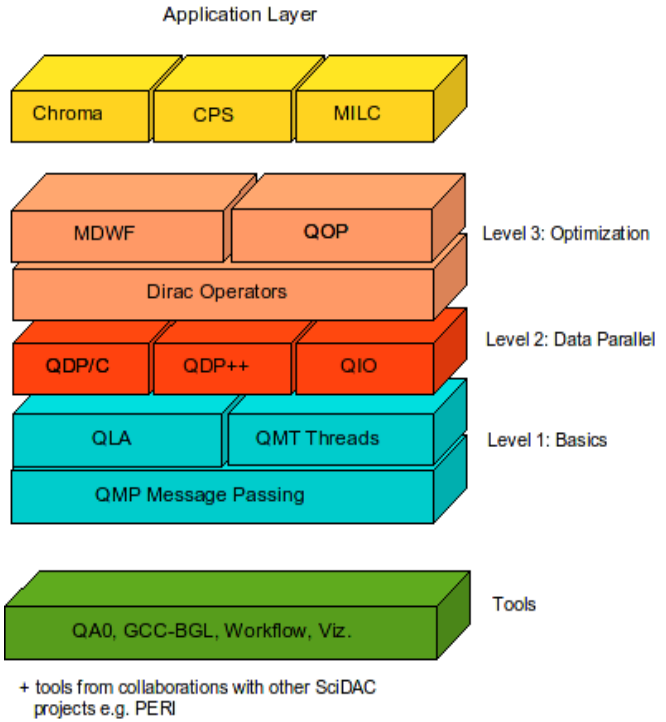
- It often isn't enough to speed up just the inverter.
- Other operations (e.g., propagator contractions, quark field smearing) will eventually become the bottleneck, requiring either more GPU code or a change in workflow (e.g., writing out propagators as an intermediate step).



Source: http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg

Philosophy

- Some operations deserve optimized routines of their own, to be added to QUDA.
- Others are problem-specific or consist of simple “glue” code.
- One could always write these pieces in CUDA, but can we make this process easier?
- Note that performance often isn't critical here. The important thing is to avoid moving data off the GPU.
- In the language of the USQCD software stack, QUDA is a “Level 3” package.
- As of yet, there is no Level 2 library for GPUs. (QUDA does contain components that might be useful for building such a thing.)



“QDP” for GPUs?

- I only mean to pose the question, not answer it.
- Since I'm most familiar with QDP/C, the following thought experiment seems natural: What sort of layer would we need to run our QDP/C code on GPUs with minimal changes?
- Simply porting QDP to GPUs probably isn't possible, but can we come close?
- QDP/C basically provides:

    1. Site-local operations performed in parallel: vector addition, SU(3) multiplication, spin projection, etc. This is doable.
    2. Support for arbitrary subsets (doable, with a performance penalty).
    3. Operations involving shifts. Supporting general shifts would be hard, but a subset might be enough.

Challenge 1: The non-QDP bits

- In practice, real code consists of more than just QDP calls. We often “break out” of QDP and perform site-local operations using QLA.
- Porting QLA in all its generality would be *hard,* and QLA code would perform terribly unless operations are fused together.
- Reasonable alternative might be to re-write these portions in CUDA directly, perhaps using a collection of QLA-like `__device__` functions or a convenience library such as *Thrust*.

Challenge 2: Heterogeneity

- I've so far assumed that *all* the work is done on the GPU.
- This won't be practical in many cases, due to limited GPU memory if nothing else.
- We would need both a host-side and GPU-side QDP, with functions for transferring data between them (often reordering arrays in the process).
- Or perhaps we can just treat the host as a storage resource: “[Host] memory is the new disk.” This wastes the CPU, however, and CPUs are undeniably better at some things.
- Either way, the programmer will be forced to make choices, many of which depend on the target machine.

- Our simple programming model is beginning to break down. Perhaps QDP is not the right place to start . . .