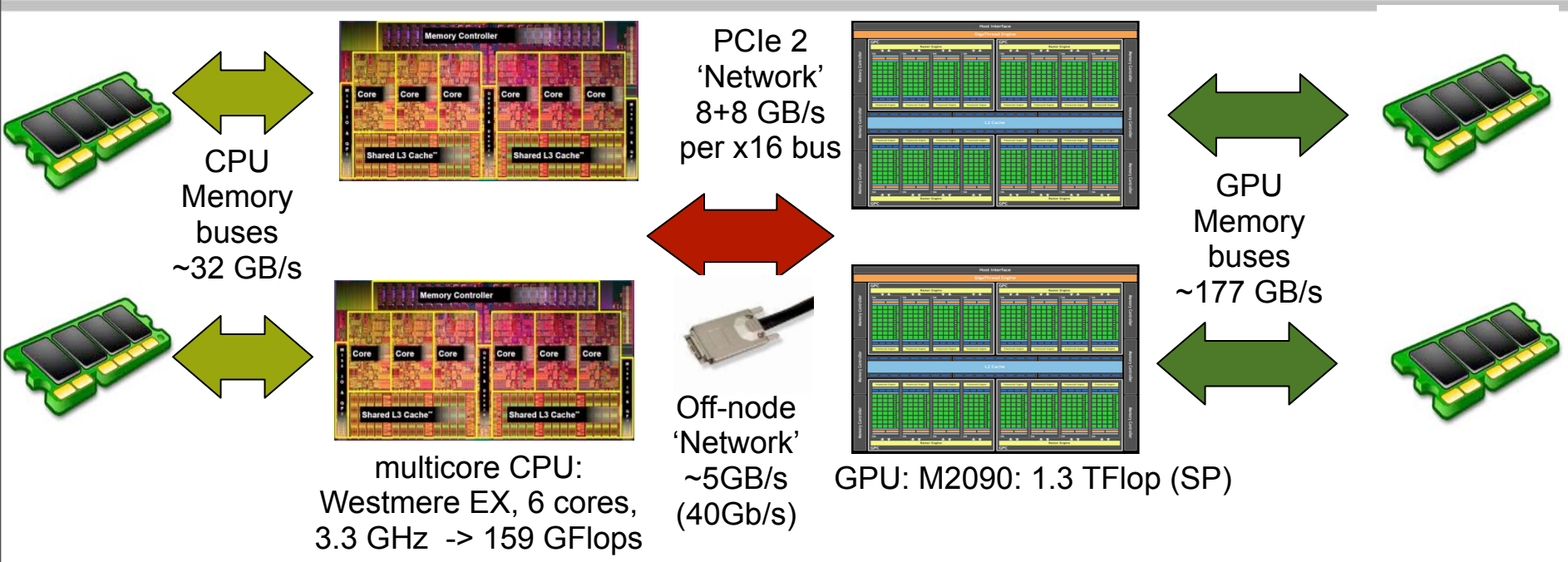
Chroma and GPUs

Bálint Joó,
Scientific Computing Group
Jefferson Lab
+
QDP-JIT, Frank Winter, University of Edinburgh

Chroma and GPUs

- GPU Computing promises a way to get lots of FLOPs
  - both in terms of FLOPS/$ (FLOPS/Euro)
  - and in terms of FLOPS/Watt
  - Mike's lectures cover how to program NVIDIA GPUs
- I'd like to touch on a couple points of GPU computing with Chroma
  - GPU Systems: Bottlenecks?
  - Integration with QUDA in terms of Solvers
  - Amdahl's Law : Yes! You get to hear it again :)
  - Working towards reducing Amdahl's law using QDP-JIT
    - This last bit should really be given by Frank :)

Typical Cluster Set Up



- GPU Mem. B/W / CPU Mem. B/W ~6.9x
- GPU Peak Flops (SP) / CPU Peak Flops(SP) ~ 8.4x
- PCIe Gen2 serious bottleneck for multi-GPU
- Changed slightly with SandyBridge/Interlagos, PCIe3
  - but not qualitatively

JLab 10G cluster

Improvements

- PCIe3: effectively doubles Bandwidth
- Recent processors: Faster CPUs
- GPU Direct: more efficient use of PCIe by GPUs
  - peer to peer amongst GPUs
  - GPU to Fabric without using hosts
    - Can argue: same latency / BW as if regular host was doing the communications
- Can the power of GPUs be leveraged from Chroma?

Chroma and QUDA

- Mike will have discussed QUDA in his lectures.
- A library for QCD Calculations using CUDA
  - provide solvers (for Wilson/Clover/Twisted Mass/Staggered etc)
    - and by extension, some linear operators
  - provide force terms (primarily for Improved Staggered for now)
  - provides a gauge action
  - DWF is work in progress.
  - Solvers are very fast (multi-precision and other techniques)
- Chroma integration:
  - Integrated Wilson and Clover Solvers
    - For propagator calculations (M x = b solves)
    - and for use in force calculations ($M^{+}M$ x = b solves)

How do I build it?

- Turnkey builds available in the package_xxx.tar.gz style:
  - package-quda.tar.gz
  - For now, contact me if you need these (bjoo AT jlab.org)
    - exist for general clusters
    - CrayXK systems
    - the variations are primarily in the env.sh file

How does it work?

- A new solver types. Their XML <invType>-s are:
  - QUDA_CLOVER_INVERTER
  - QUDA_WILSON_INVERTER
- These map as appropriate to SystemSolver-s in Chroma
- defined in /lib/actions/ferm/invert/quda_solvers/

The XML File

```
<InvertParam>
  <invType>QUDA_CLOVER_INVERTER</invType>
  <CloverParams>
   <Mass>0.0</Mass>
   <clovCoeffR>1</clovCoeffR>
   <clovCoeffT>1</clovCoeffT>
   <AnisoParam/>
  </CloverParams>

  <AntiPeriodicT>true</AntiPeriodicT>

  <RsdTarget>1.0e-7</RsdTarget>
  <Delta>1.0e-1</Delta>
  <MaxIter>10000</MaxIter>
  <SolverType>BICGSTAB</SolverType>
  <Verbose>true</Verbose>
  <AsymmetricLinop>false</AsymmetricLinop>
  <CudaReconstruct>RECONS_12</CudaReconstruct>
  <CudaSloppyPrecision>HALF</CudaSloppyPrecision>
  <CudaSloppyReconstruct>RECONS_12</CudaSloppyReconstruct>
  <AxialGaugeFix>false</AxialGaugeFix>
  <AutotuneDslash>true</AutotuneDslash>

  <GCRInnerParams/>
 </InvertParams>
```

Repeat these. Must be the same as in the preceding FermionAction

BC Info needs to be known (even if folded into the gauge field, but gets lost when using 2-row compression)

Select preconditioning style

true: $A_{oo} - D_{oe} A^{-1}_{ee} D_{eo}$

false: $1 - A^{-1}_{oo} D_{oe} A^{-1}_{ee} D_{eo}$

The XML File

```
<InvertParam>
  <invType>QUDA_CLOVER_INVERTER</invType>
  <CloverParams/>
  <AntiPeriodicT>true</AntiPeriodicT>

  <RsdTarget>1.0e-7</RsdTarget>
  <Delta>1.0e-1</Delta>
  <MaxIter>10000</MaxIter>
  <SolverType>BICGSTAB</SolverType>
  <Verbose>true</Verbose>
  <AsymmetricLinop>false</AsymmetricLinop>

  <CudaReconstruct>RECONS_12</CudaReconstruct>
  <CudaSloppyPrecision>HALF</CudaSloppyPrecision>
  <CudaSloppyReconstruct>RECONS_12</CudaSloppyReconstruct>
  <AxialGaugeFix>false</AxialGaugeFix>
  <AutotuneDslash>true</AutotuneDslash>

  <GCRInnerParams>
        <RsdSloppy>1.0e-1</RsdSloppy>
        <MaxIterSloppy>10</MaxIterSloppy>
        <NKrylov>10</NKrylov>
        <VerboseP>true</VerboseP>
        <InvTypeSloppy>MR</InvTypeSloppy>
  </GCRInnerParams>
</InvertParams>
```

Delta controls mixed precision:

Reduced Precision Solver should drop the residuum by this Delta factor

BICGSTAB, CG, GCR

QUDA Settings

Enable Autotuning

Inner solver params for, GCR
If not using GCR one should omit this group (Chroma should generate safe defaults)

Some Notes

- Typically in Chroma, for a solver we generate a linear operator, using the <FermionAction> XML
  - QUDA doesn't use our linear operator , but has its own
  - there is no way in Chroma of interrogating parameters from the instantiated linear operator
    - => We need to repeat the parameters for QUDA
- RECONS_12 (2 row reunitarization) cannot reconstruct the antiperiodic boundary (just a - sign on those links)
  - Only periodic/antiperiodic boundaries allowed in QUDA
- cudaPrecision (non-sloppy) is inferred from build precision
- setting <Verbose> to true will display the solver convergence history

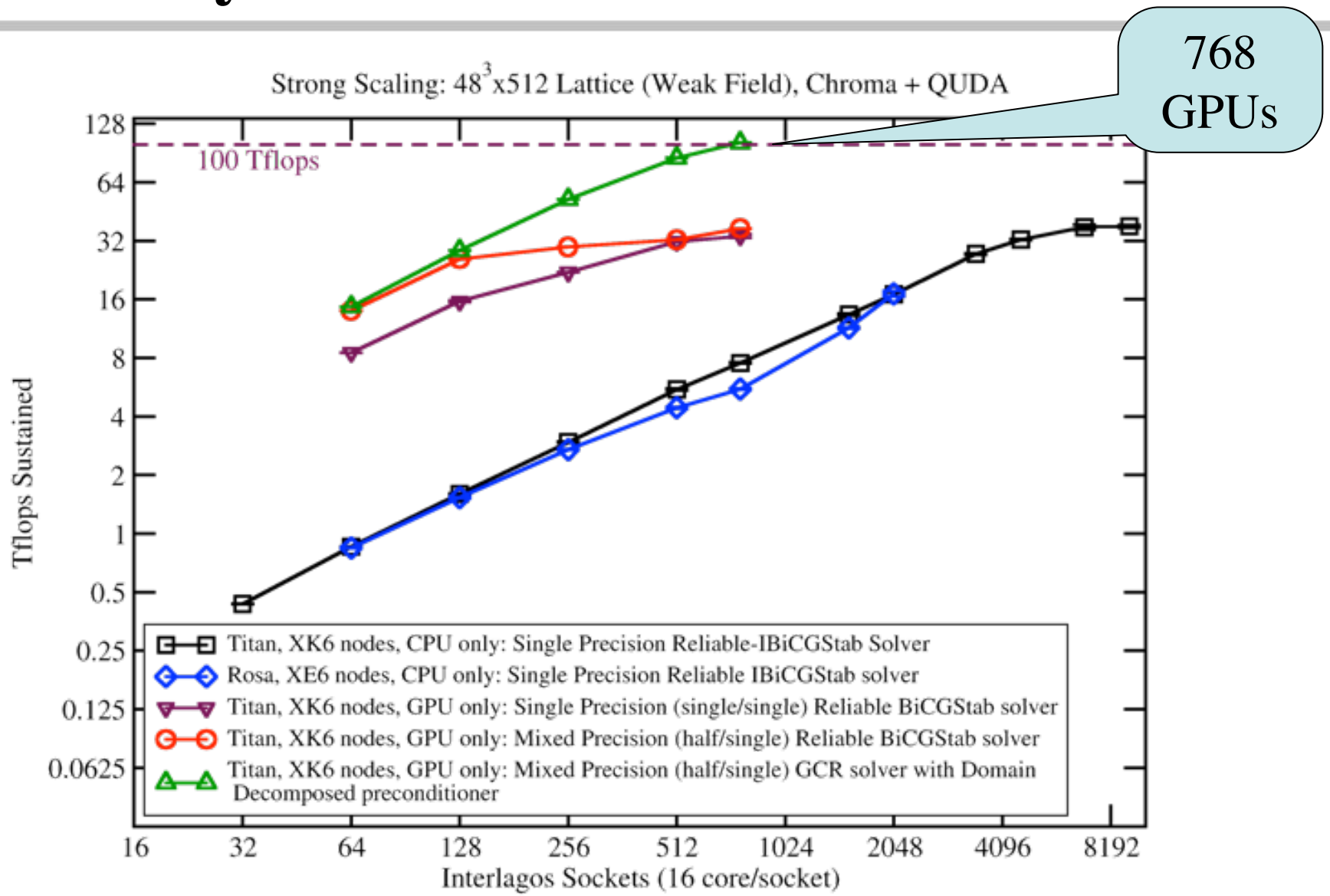
Autotuning

- QUDA will autotune the kernels used in your program, for optimal performance
  - it will try various block/grid size combinations and pick best one
  - but the first solve (when the tuning is done) will be slow.
  - QUDA will write out optimal parameters to a file on exit
  - this way you won't need to pay the tuning penalty more than once
  - specify file for autotuning params with env. var:
    - QUDA_RESOURCE_PATH
    - should point to a directory

Some gotcha's

- For CUDA version < CUDA 4.1 you must set env var
  - CUDA_NIC_INTEROP=1 (GPU direct)
  - versions 4.1 and higher don't need this
- For chroma you must always set the -geom command line argument -- to trip QMP into defining a logical topology
  - even if the grid is -geom 1 1 1 1
- You are allowed to run multiple host OpenMP threads
  - if chroma was built for mixed OpenMP/MPI operation

Very recent results from TitanDev



Can I use QUDA in HMC

- In principle, yes. The solvers are 'hooked up' also as MdagMSystemSolver (and MdagMSystemSolverMulti)
- In practice, the benefits from using just QUDA solvers will depend on:
  - your lattice size
    - If it is too small, QUDA may not get good throughput
    - If it is too big, the CPU can't keep up with the non solver stuff
    - Preliminary results with 2 flavor Wilson indicate cross-over point (for the particular lattice size and machine tried)
  - machine balance:
    - too many GPUs per CPU: Amdahl's law

Stopping Point

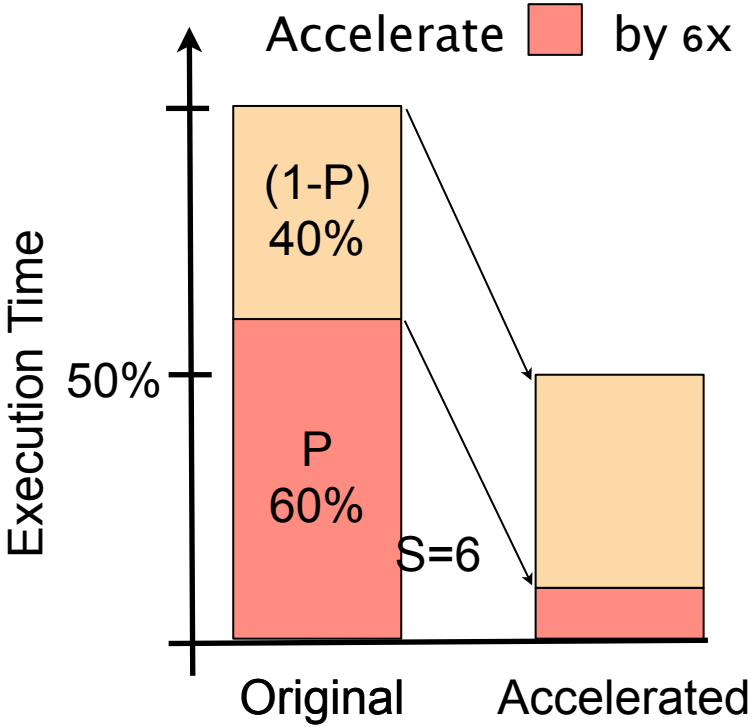
- Discussed the interfacing of Chroma + QUDA
- Went through the XML to run the QUDA solver from within Chroma
- Discussed naive implications for HMC

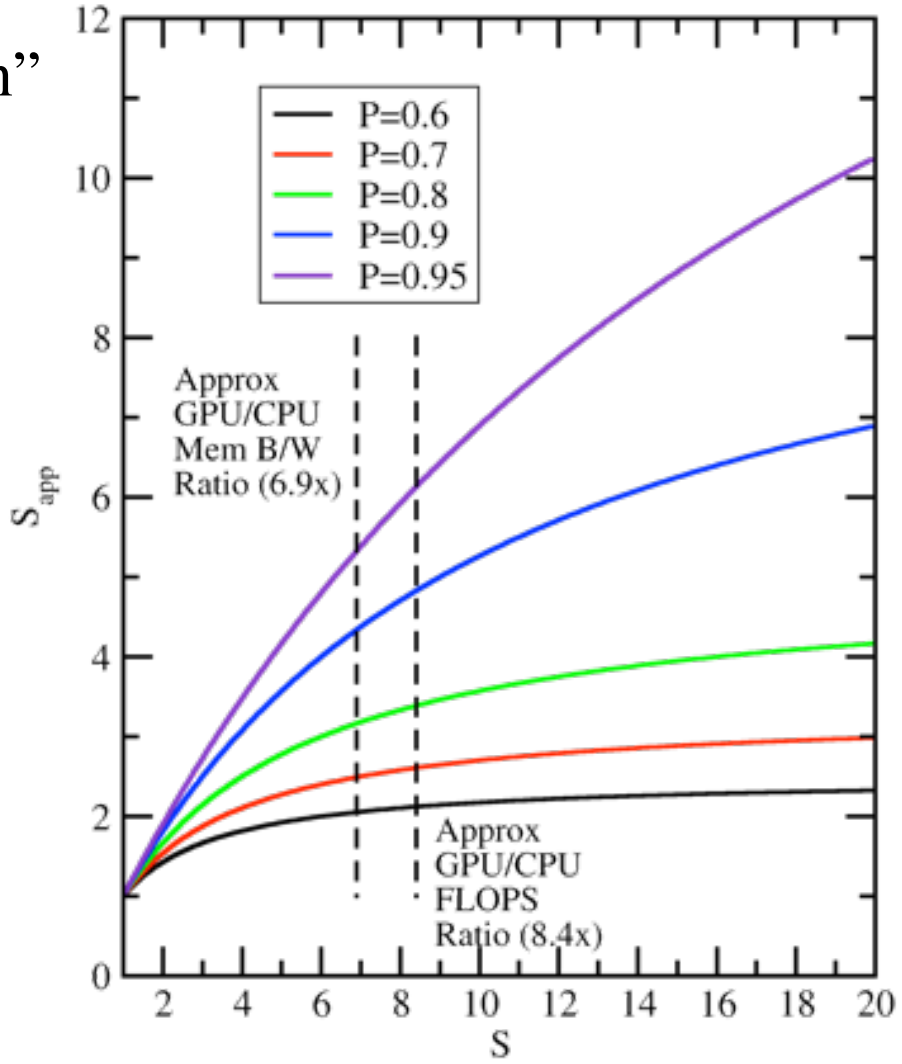
- Continuations:
  - Moving more of Chroma to the GPU: QDP-JIT

Amdahl's Law

"The speedup of the application is limited by the unaccelerated portion"

$$S_{\text{app}} = \frac{1}{(1-P) + \frac{P}{S}}$$





What does Amdahl's law mean for me?

- When using QUDA, you may find that time spent in the solver is longer the bottleneck
  - Gauge generation: Solver takes 50-70% of time: max S ~ 3x
  - Source Smearing, Sink Smearing in prop calculations
- But how do we get those parts onto the GPU?
- Solution: Move QDP++ to the GPU
  - expect greatest benefit in systems with high GPU/CPU ratio
  - since the CPU will essentially be ignored (wasted)

What are the issues

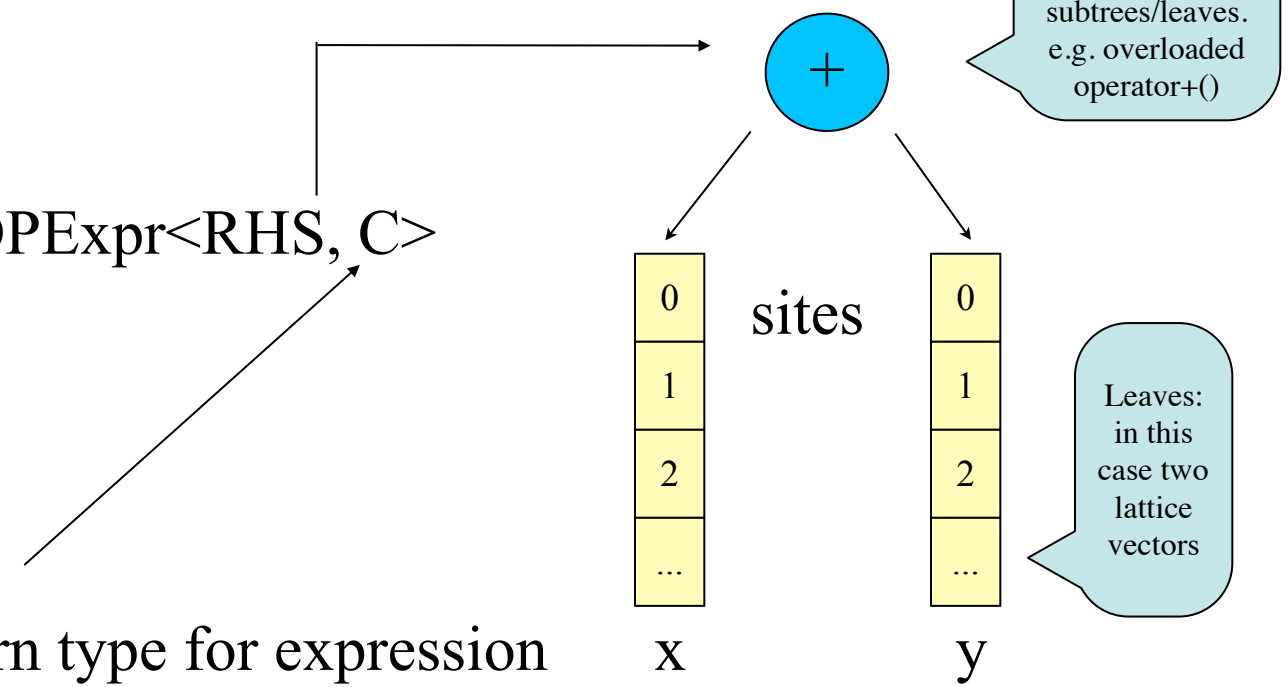
- The primary ones are:
  - getting your expressions onto the GPU
  - memory movement between host and device
  - memory layouts and coalescing

Reminder about Expression Templates

- Everything is 'controlled' from the host (accelerator model)
- operator+() (on host) creates expression.
- expression has references to the leaves
- expression has 'code' to evaluate on the host.

Overload operator+()

x + y ; ⟶ QDPExpr<RHS, C>



C= container for return type for expression

What must happen on evaluation

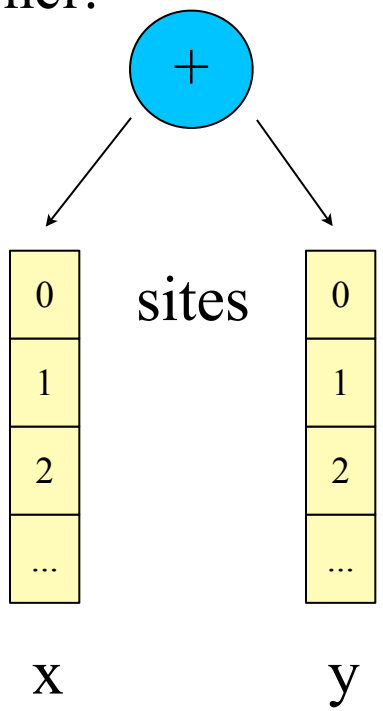
- evaluate() takes a reference to an expression: QDPExpr<RHS,T>& rhs
- The operations to be carried out are in the RHS type
- Regular C++ compiler generates code for the RHS
  - but knows nothing about CUDA?
  - how does the code for the node become a CUDA kernel?
- How does the data move to the device?

```
template<class T, class T1, class Op, class RHS>
void evaluate(OLattice<T>& dst, const Op& op,
              QDPExpr<RHS,OLattice<T1> >& rhs)
{
     forall sites i do:
        op( dst.elem(i),
            ForEach(rhs, EvalLeaf1(i), OpCombine()));
}
```

ForEach:
recursive tree traversal

EvalLeaf1 functor:
selects which site
to work with

OpCombine functor:
calls code in
node to evaluate its
subtrees



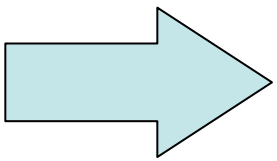
Dynamic (Just In Time) compilation

- NB: nvcc can deal with Expression Templates
  - but these need to live in the source for the kernels.
- So the problem can be refined:
  - for a QDPExpr<RHS,T>, which is 'lattice wide' on the host we must generate CUDA kernels which contain code for 'body' of the evaluate() site loop on the host
- Turns out, that judicious use of the ForEach() can **perform this transformation at run-time (when evaluate is called)**
- So, the call to evaluate() writes the kernel for us
- The technique of writing/building/linking code on the fly, is called dynamic compilation or Just In Time compilation (JIT)

A Sketch of the Idea

"Effective Host Code":
(after ET's do their magic)

```
template<>
void evaluate( ... )
{
   for( sites .. ) {
     SU3Mat a=arg_a[i];
     SU3Mat b=arg_b[i];
     res[i] = a * b;
   }
}
```



```
__device__
void kernel( ... )
{
     int i = ...
     SU3Mat a=arg_a[i];
     SU3Mat b=arg_b[i];
     res[i] = a * b;
}

template<>
void evaluate( ... )
{
   if( !generated ) {
     generateKernel();
   }
   if (!compiled) {
     compileKernel();
   }
   if (!loaded) {
     loadKernel();
   }
   ensureLeavesAreOnDevice();
   setupGrid();
   kernel<<< ... >>>()
}
```

Implementing QDP-JIT

- QDP-JIT : development lead by Frank Winter
  - Use PETE tree traversal to write the kernels and C++ callers
  - use ‘shell()’ to launch nvcc compiler, and build kernels
  - use ‘ldopen()’ library to load .o files
    - JIT only once, then keep kernels for successive runs
    - autotune CUDA block sizes (log tuned parameters)
  - Implement a special memory pool for allocations
    - track whether memory is on host, or GPU
    - ensure operands for kernels are on GPU when needed
    - cache management (spill data to CPU if necessary)
    - QUDA integration
      - QUDA should use QDP-JIT memory pool

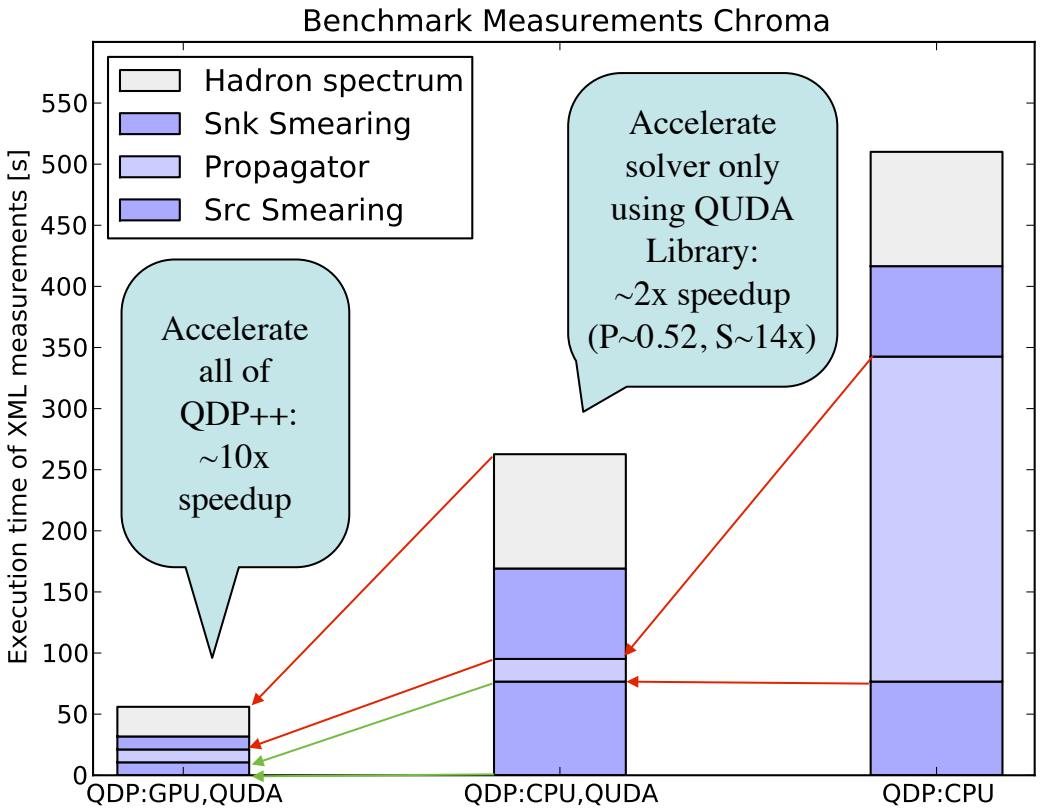
Compilation

- Get it from the GIT repo:
  - git clone --recursive git://git.jlab.org/pub/lattice/usqcd/qdp-jit.git
  - cd qdp-jit ; autoreconf
- GPU Specific options
  - --enable-gpu : turns on GPUs
  - --enable-gpuarch=sm20 : Fermi
  - --with-cuda=<cuda location>
  - --enable-cpuarch=x86_64
- Helps to turn off SSE etc for QDP++/Chroma
- Compile QUDA after QDP-JIT
  - --enable-qdp-jit : Tell QUDA to use QDP-JIT memory pool
  - --with-qdp=<location of QDP-JIT installation>

Running

- Run time environment variables:
  - QDP_TEMP=<directory>
    - directory for temporary files/JIT-ed kernels
  - QUDA_RESOURCE_PATH=<directory>
    - directory for autotuning databases (same as used by QUDA)
- Run time command line options
  - -poolsize <size>
    - Size of memory pool can e.g. “4.5g”
  - -maxpoolelement <size>
    - Size of biggest element to put in pool
  - -qudadynamic 0/1
    - Whether QUDA should use the pool (0) or not (1)

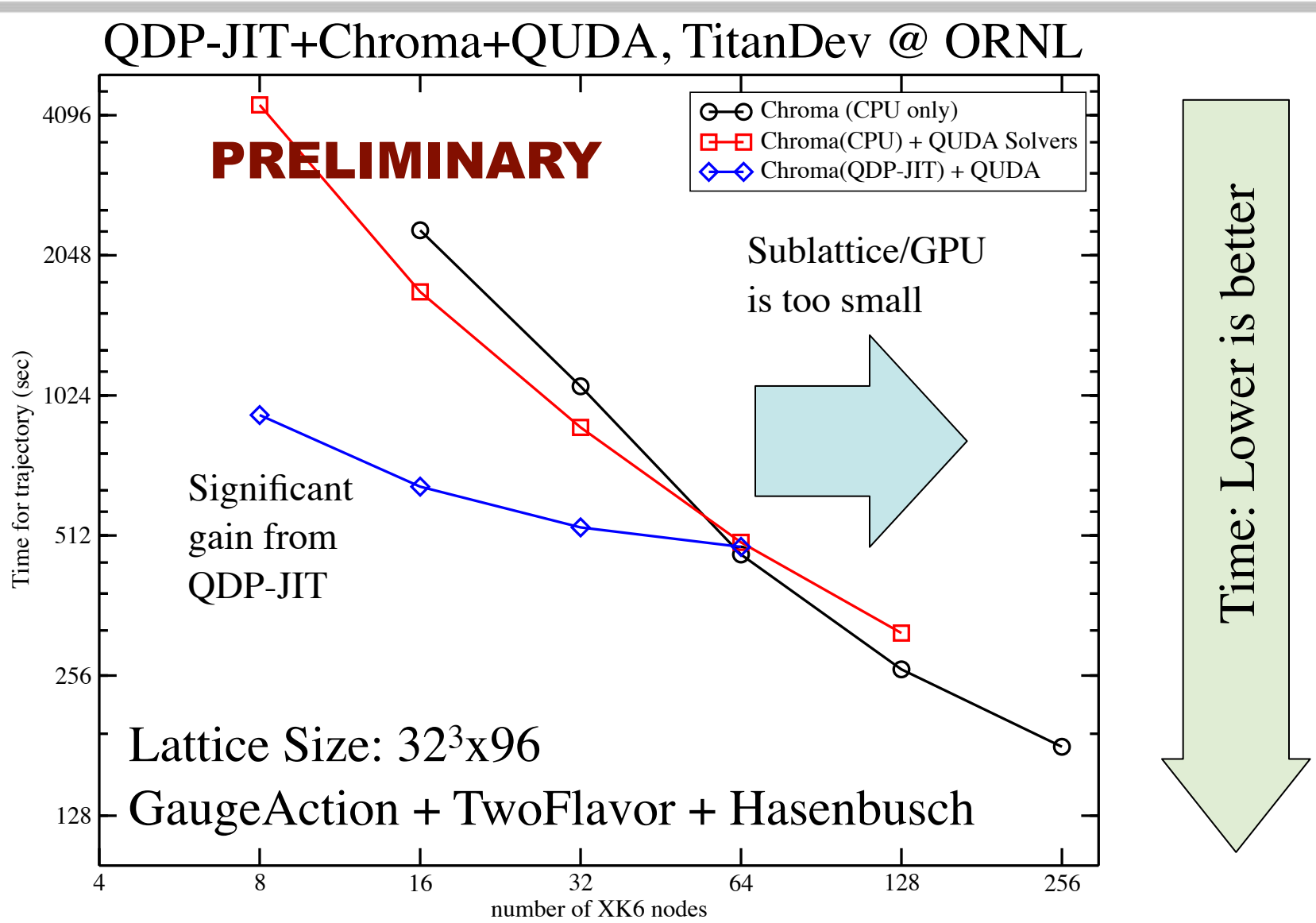
Beating Down Amdahl's Law



http://github.com/fwinter/qdp

- Results from Frank Winter's talk at Autumn StrongNET meeting (Trento, 2011)
- QUDA alone only gave ~2x speedup on full application
- QUDA + moving all of QDP++ to GPU resulted in ~10x speedup
- See also: F. Winter "Accelerating QDP++ using GPUs" arXiv:1105:2279[hep-lat]

2 Flavor Wilson HMC



Comments

- Chroma(CPU) + QUDA scales a lot better at the moment
  - more like straight Chroma(CPU)
  - cause: communications (e.g. in Gauge Forces)
    - CPU communicates directly to Gemini via HT
    - GPU has to go through PCIe2 first
- Will become much better with GPUDirect (and its Cray variant)
- Lattice Sizes very small in this case
  - GPU needs to be able to work in throughput mode
  - Use larger lattices (we want to do 48^3x512 anyway)
- QDP-JIT leaves CPU more or less idle (once JIT-ing is complete)
- For machines that can't JIT (e.g. Cray Back-End compute nodes)
  - transfer kernel .cu files from QDP_TEMP, and recompile

Stopping Point

- Discussed How to Interface QUDA + Chroma
  - how to set up XML files for QUDA Solvers
- Discussed Amdahl's law and its implications
- Discussed Implementing QDP++ on GPUs using JIT compilation
- Discussed Building/Running with QDP-JIT
- Discussed current status of running HMC on GPU based machines

- Possible continuations
  - Tutorial / Demonstration of QDP-JIT + Chroma + QUDA
  - General discussion
    - e.g. Templates, Design Patterns