Accelerating Lattice QCD Calculations on GPU-Enabled Systems

F. T. Winter

in collaboration with

M. A. Clark (NVIDIA), R. G. Edwards, B. Joó

Thomas Jefferson National Accelerator Facility

Numerical Algorithms for Extreme Computing Architectures

Boston, USA

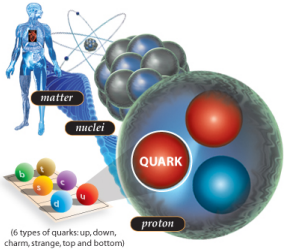
November 4-5, 2013 at Boston University

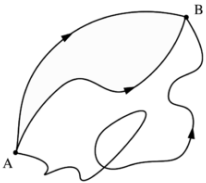
Outline

- QCD is the dynamical theory of quarks and gluons

$$\mathcal{L}_{\mathrm{QCD}} = \frac{1}{2}\mathrm{Tr}(F_{\mu\nu}F_{\mu\nu}) + \sum_{i=1}^{N_f} \overline{\psi}^{(i)} \left(\not{D} + m^{(i)} \right) \psi^{(i)} + i\theta \frac{1}{32\pi^2} \epsilon_{\mu\nu\rho\sigma} \mathrm{Tr}(F_{\mu\nu}F_{\rho\sigma})$$

- Quantum Field Theory, Non-Abelian gauge group
- Elementary degrees of freedom are quarks and gluons transforming in the fundamental representation of SU(3)





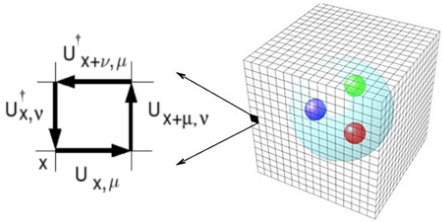
- Feynman's path integral approach

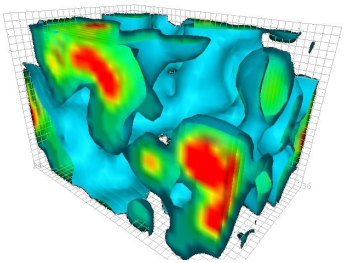
$$\langle \mathcal{O} \rangle = \frac{1}{Z} \int \mathcal{D}[\Phi] \mathcal{O} e^{-S(\Phi)}$$

- QCD must be regularized both in the UV and IR
- The formulation on the lattice which provides regularization for $a > 0$ and $V = L^4 < \infty$

- Discretization of space-time on a 4-dimensional hypercubic lattice
- Ascribe fermionic degrees of freedom to lattice sites, Gauge bosons to links
- Controlled discretization errors, typically $\mathcal{O}(a^2)$
- Continuum limit approached as

$$a \to 0 \qquad L \to \infty$$





- Infinite dimensional path integral $\to$ high dimensional sum
- Analytically integrate fermionic degrees, importance sampling (fermion determinant)
- Hybrid Monte Carlo (HMC) and variants update the quark and gluon fields

$$\langle \mathcal{O} \rangle = \frac{1}{Z} \int \prod_{\text{all links}} dU_i \mathcal{O} e^{-S(U)} \to \overline{\mathcal{O}} = \frac{1}{Z} \prod_{\text{configurations}} \mathcal{O}(U)\, P(U)$$

- Gauge field generation (Hybrid Monte Carlo methods)
- Evaluating the Path Integral:
  There are $4V$ links. $V = 40^3 \times 256 \rightarrow 4V = 65M$ links
- Typically requires “capability machine”
  (can solve one large problem in the shortest possible time)

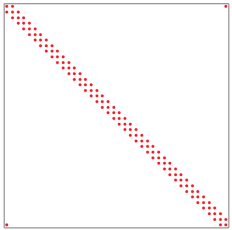
12k GPU cluster at JLab

- Analysis of gauge field configurations
- Propagator calculations, Contractions
- Parallelization on independent gauge field configurations
- Typically a “capacity machine” suffices
  (can solve many smaller problems simultaneously)

- Main consumer of computational power in Lattice QCD calculations:
- Solution to a large linear system of equations (dimension 10^9 and larger)

$$Dx = b$$

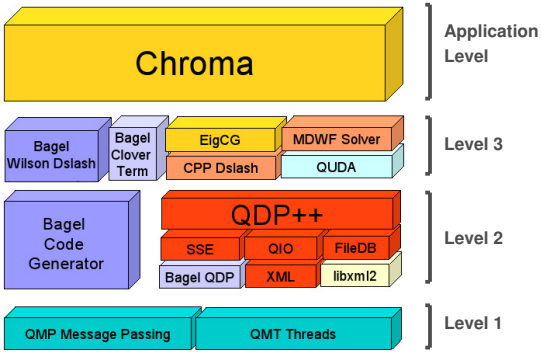
  where D is the discretized Dirac operator, a large sparse matrix
- Typically done by conjugate gradient algorithm (or similar)
- Requires efficient: Dx multiplication
  $y = ax + b$ linear algebra and $r = x \cdot y$ scalar product, global sums



Wilson $\not{D}$ is $12V \times 12V$ complex sparse matrix

- The QUDA library provides CUDA-accelerated linear solvers
- Highly-optimized operations, mixed precision, domain-decomposition methods
- QUDA has been successfully applied to the post-Monte Carlo analysis phase

- Software architecture for Lattice Field Theory
- Layered software architecture:
- Domain science (application) level, implements algorithms in terms of high-level domain-specific interfaces
- Level 3 (optimization): Plug-in functionality, e.g. linear solvers
- Data-parallel level, provides data types and operations for Lattice Field Theory
- Threading and Message Passing level: Provides abstractions for multi-core/node



Data types

- C++ template library with data-parallel expressions for lattice field theory
- QDP++ captures the tensor index structure of lattice QCD types

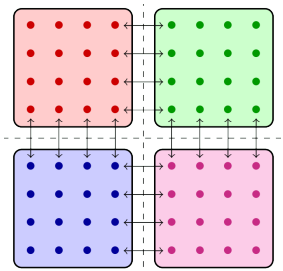
	Lattice	Spin	Color	Reality	built-in
`Real`	Scalar	Scalar	Scalar	Scalar	REAL
`LatticeColorMatrix`	Lattice	Scalar	Matrix	Complex	REAL
`LatticePropagator`	Lattice	Matrix	Matrix	Complex	REAL
`LatticeFermion`	Lattice	Vector	Vector	Complex	REAL

- These data types are implemented with C++ templates

```
typedef
OLattice< PSpinVector< PColorVector< RComplex<REAL>,Nc>,Ns»
LatticeFermion;
```

Parallelization

- `OLattice` ascribes its elements to an N_d-dimensional grid
- The grid is parallelized across multiple MPI ranks



MPI parallelization of the data-parallel container

Operations

- QDP++ provides an "embedded domain-specific, data-parallel language", like HPF
- Whole-array elemental operations, e.g. `z = a * x + y` where a, b and c are arrays.
- Array selection, the ability to refer to subsets of an array, e.g. `z[rb[0]] = a * x + y`
- `Shift` operations as building blocks for stencil operations, e.g. `shift( U[mu] * psi , FORWARD , mu )`
- Configure time parameters: Number of space-time N_d, spin N_s and color N_c indices.
- I/O routines, RNG, etc.

QDP++ (summary)

- It's a virtual machine for Lattice QCD calculations which abstracts architectural details with a high-level interface

- Just adding Level-3 libraries opens a serious Amdahl's law issue:

  On Blue Waters, adding QUDA to Chroma, for our HMC production we see a factor ~ 2 improvement

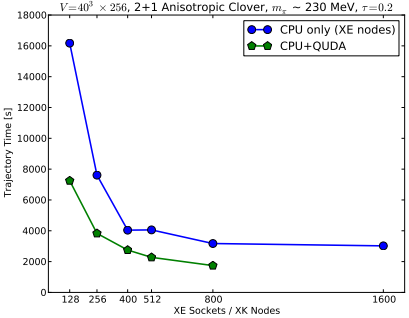
  Many calculations are not accelerated, we are constantly moving data between CPU and GPU memories

- this motivates to move *all* of Chroma to the GPUs
- QDP++ is a framework worth targeting because

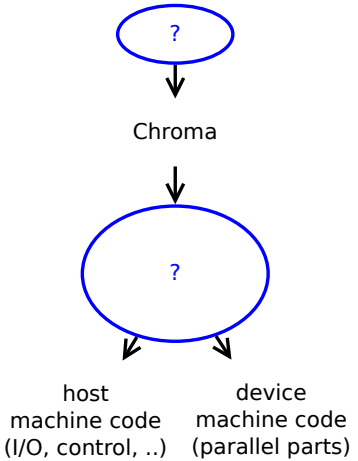
  It implements virtually all functionality of Chroma,

  it is relatively compact,

  and getting it onto GPUs can have an immediate, large impact.



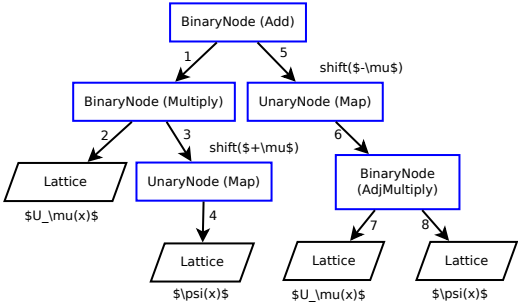
- To date no GPU programming framework can off-load “expressions” to accelerators
  - QDP++ uses *expression templates* to implement operations on the data-parallel container
  - Although CUDA supports C++ templates, refactoring of the Chroma code into “kernels” would be needed
- GPU programming frameworks expose “explicitly managed memory domains”
  - Program control flow determines the order of data accesses, i.e. host or device
  - It’s impractical to add memory copy instructions to the Chroma source code (over 400K C++ code lines)
- Data layout optimization, CUDA kernel tuning, etc.



- QDP-JIT implements QDP++ API with support for NVIDIA GPUs
- Automatic off-loading of expressions to the accelerators
- Multi-GPU support
- Dynamic PTX code generation
- Additional Just-In-Time (JIT) compilation step
- Data layout is optimized for coalesced memory accesses
- Automatic memory transfers via a 'software cache'
- Automatic tuning of CUDA kernels
- Only a general-purpose C++ compiler is required to build the application
  On the compute nodes: No additional compiler is needed, only the CUDA kernel driver

- We use the "structural information" in expression templates to build code generators
- In analogy to static compilers, unparsing the program AST interfaces to a code generator
- However, our approach implements unparsing the AST with the C++ visitor pattern
- While traversing the AST we issue calls to a code generator (PTX)
- NVIDIA JIT compiler generates GPU code

$$U_\mu(x)\psi(x+\hat{\mu}) + U^\dagger_\mu(x-\hat{\mu})\psi(x-\hat{\mu})$$

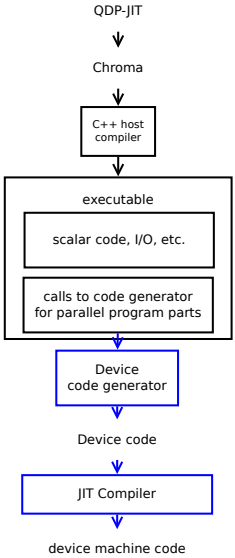


Syntax tree for a simple 1-dimensional nearest-neighbor derivative operator

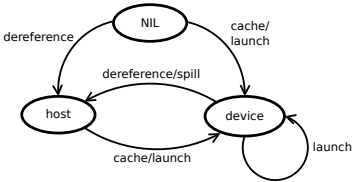
- Standard CUDA code generation toolchain:

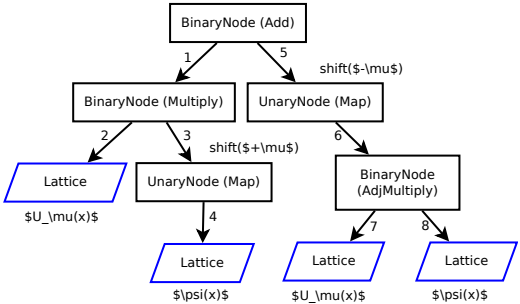
$$\text{CUDA C/C++} \xrightarrow{\text{NVCC}} \text{PTX} \xrightarrow{\text{Linux driver}} \text{GPU code}$$

- PTX (Parallel Thread eXectution), virtual, low-level, machine and instruction set architecture
- The Linux driver JIT-compiles PTX to machine code
- We skip the compilation with NVCC by directly generating our kernels in the PTX language



- We use the “dynamic information” in expression templates to identify data fields
- Prior to a kernel launch we make all data fields available in GPU memory
- The dynamic nature of memory accesses → Use a “cache” for GPU memory
- The cache dynamically spills “least recently used” data fields to CPU memory





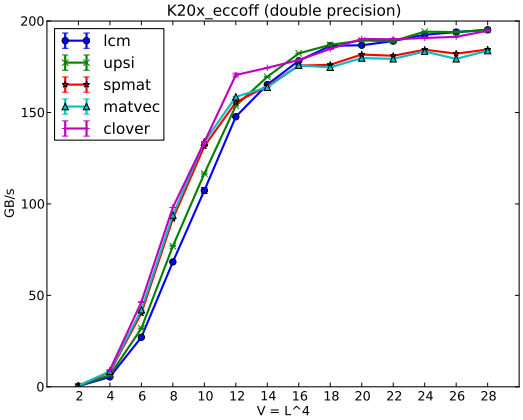
The leaf nodes include references to data fields

- We identified a set of relevant kernels

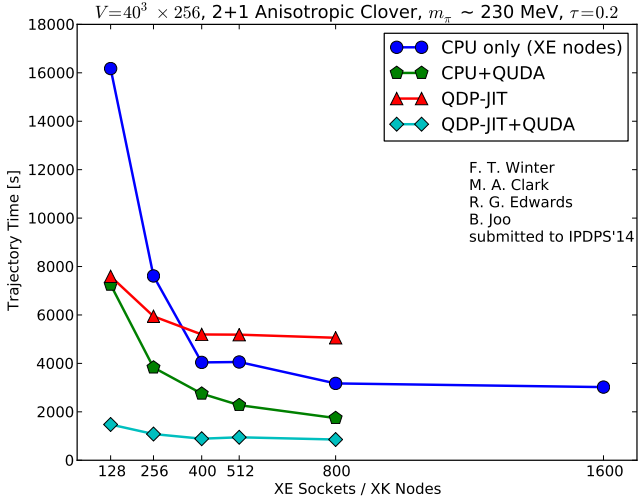
Test	Expression	flop/byte (DP)
`lcm`	$U_1 = U_2 * U_3$	0.458
`upsi`	$\psi_1 = U_1 * \psi_2$	0.5
`spmat`	$\Gamma_1 = \Gamma_2 * \Gamma_3$	0.62
`matvec`	$\psi_0 = U_1 * \psi_1 + U_1 * \psi_2$	0.64
`clover`	$\psi_0 = A * \psi_1$	0.525

- NVIDIA K20x GPU (GK110)
- Peak performance: 1.3 TFlops (DP)
- Peak memory bandwidth: 250 GB/sec (ECC disabled)

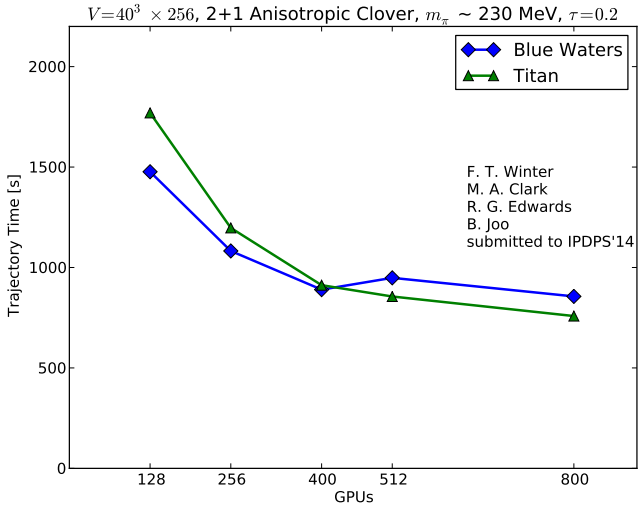
- "Performance shoulder" at around $V = 12^4$ due to SM occupancy
- Our generated kernel sustain up to 79% of peak memory bandwidth



- QDP-JIT allows us to deploy the full Chroma gauge-generation program in production on large-scale GPU-based machines such as Titan (OLCF)
- QUDA device interface understands QDP-JIT's datalayout
- Speedup factor of $\sim$ 11.0 on 128 GPUs/CPU sockets
- Speedup factor of $\sim$ 3.7 on 800 GPUs/CPU sockets



- The Blue Waters and Titan results are hardly distinguishable
- Our benchmark timings on these systems typically show some amount of fluctuation



Conclusions

- QDP-JIT provides a QDP++ API with support for the CUDA architecture
- QDP++ implements an embedded data-parallel language
- The expression templates are of enabling character for this work
- A JIT compiler from a low-level assembler to accelerator code is required
- This work reduced the computational cost for our production simulations on Titan by a factor of ~ 5

Acknowledgement

- Computer time:
  Oak Ridge Leadership Computing Facility (OLCF) NSF Blue Waters Facility at NCSA
  US National Computational Facility for LQCD (JLab)
- This research was in part supported by the Research Executive Agency (REA) of the European Union under Grant Agreement number PITN-GA-2009-238353.
- Funding through U.S. DOE Office of Science, Offices of Nuclear Physics, High Energy Physics and Advanced Scientific Computing Research through the SciDAC Program