QDP-JIT: A QDP++ Implementation for CUDA-Enabled GPUs

Frank Winter, Balint Joo, Robert Edwards

Jefferson Lab

The 31th International Symposium on Lattice Field Theory
Mainz, Germany
July 29 - August 3, 2013

Outline

1. Introduction QDP++
2. QDP-JIT
3. Benchmark results
4. HMC on Titan
5. Conclusions

Outline

1. Introduction QDP++
2. QDP-JIT
3. Benchmark results
4. HMC on Titan
5. Conclusions

Brief Overview of QDP++

What is QDP++?

(QCD Data Parallel, C++ interface)

- provides data types and operations suitable for lattice field theory
- implicitly data-parallel framework
- hides architectural details from the user
- can be deployed on parallel systems with CPU architectures

- basis of the popular LQCD software suite Chroma (421718 C++ code lines, 295 cites)
- open source software
- main software architects: B. Joó, R. Edwards (Jefferson Lab)

QDP++ Type System

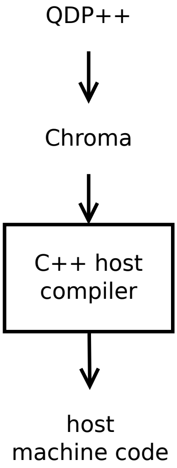
- Domain specific data types (C++ templates)
- Operations are implemented as expressions (PETE)
- ETs provide syntactic sugar, e.g. write `x[rb[0]]=a*y+z;` instead of `saxpy(x,a,y,z)`
- ETs evaluate in a cache-friendly manner
- However, high performance only achievable through specialization (not portable)
  (efforts underway to avoid this, e.g., parscalarvec architecture)

```
typedef OLattice< PSpinVector< PColorVector< RComplex< float >, Nc>, 4 > > LatticeDiracFermion
typedef OLattice< PScalar<     PColorMatrix< RComplex< float >, Nc>    > > LatticeColorMatrix;
typedef OLattice< PSpinVector< PColorVector< RComplex< float >, Nc>, Ns> > LatticeFermion;
typedef OLattice< PSpinMatrix< PColorMatrix< RComplex< float >, Nc>, Ns> > LatticePropagator;

typedef OScalar< PSpinMatrix< PColorMatrix< RComplex< double >, Nc>, Ns> > PropagatorD;
typedef OScalar< PSpinVector< PColorVector< RComplex< double >, Nc>, Ns> > FermionD;
```

Chroma

- Extensive range of Lattice QCD functionality
  - Gauge generation, Propagators, Contractions
  - Variety of actions (Gauge, Fermion, 4D and 5D)
- Clean design
  - Operations implemented using QDP++ API (some exceptions apply)
  - Design patterns, e.g. C++ factories
  - Dynamic binding of class implementations via XML
- Plug-in libraries (mostly solvers)
  - QUDA (highly-tuned solver package for NVIDIA CUDA GPUs)



Outline

1. Introduction QDP++
2. QDP-JIT
3. Benchmark results
4. HMC on Titan
5. Conclusions

Motivation

- Goal: Run Chroma application on GPU-enabled parallel systems
- just adding Level-3 libraries (QUDA) opens a serious Amdahl's law issue
- this motivates to move *all* of Chroma to the GPUs
- QDP++ is a framework worth targeting because

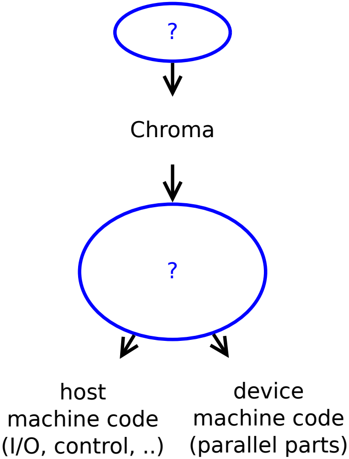
  it implements virtually all functionality of Chroma,

  it is relatively compact,

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

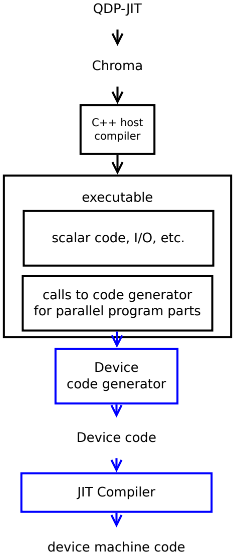
Problems

- Expressions: To date no compiler can off-load ET to accelerators
- Possible solutions (assumed no code changes to Chroma):
  - Domain specific compiler
  - ETs to build code generators
- Memory management: Current GPU programming frameworks expose low-level details to the user
  given the large code base of Chroma an automatic memory management is desired



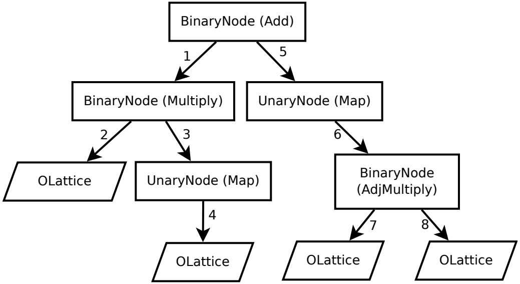
QDP-JIT: Overview

- QDP-JIT implements QDP++ API with support for NVIDIA GPUs
- Automatic off-loading of expressions to the accelerator
- 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



Dynamic PTX code generation in QDP-JIT

- Using ET to build code generators (T. Veldhuizen)
- In analogy to static compilers, unparsing the program AST interfaces to a code generator
- In contrast, our ASTs are ETs and are visited at static compile time
- Example: $U_\mu(x)\psi(x+\hat{\mu}) + U_\mu^\dagger(x-\hat{\mu})\psi(x-\hat{\mu})$
  - Inner tree nodes: Operations
  - Leaf tree nodes: Data fields
- Visiting the tree nodes (depth first, 1-8) interfacing a PTX code generator



NVIDIA Parallel Thread Execution (PTX)

- low-level, strongly-typed, machine-independent virtual machine and instruction set architecture
- three-argument assembly language style syntax
- defined on an infinite register model
- NVIDIA compute compiler driver:
  - optimizes the PTX code
  - JIT compilation to architecture-dependent code (SASS)
- Math functions only as hardware fast-math versions (approximations, small subset of `libm`)

```
.version 2.3
.target sm_20
.address_size 64
.entry function (.param .s32 param0,
.param .s32 param1,
.param .s32 param2,
.param .u64 param3,
.param .u64 param4,
.param .u64 param5,
.param .u64 param6,
.param .u64 param7,
.param .u64 param8)
{
.reg .f32 f<2052>;
.reg .u16 h<5>;
.reg .u32 u<5>;
.reg .u64 w<6>;
.reg .s32 i<19>;
.reg .s64 l<234>;
.reg .pred p<2>;
ld.param.s32 i0,[param0];
ld.param.s32 i1,[param1];
mov.u16 h0,%ntid.x;
mov.u16 h1,%ctaid.x;
mov.u16 h2,h1;
mov.u16 h3,h0;
mul.wide.u16 u0,h2,h3;
mov.u16 h4,%tid.x;
mov.u32 u2,u0;
cvt.u32.u16 u3,h4;
add.u32 u1,u2,u3;
cvt.s32.u32 i2,u1;
mov.s32 i3,i1;
....
```

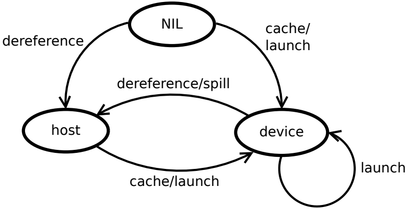
Maths functions hack

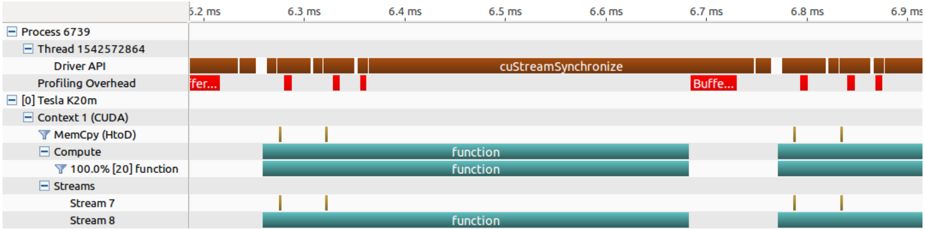
- QDP-JIT provides math functions versions with IEEE compliant rounding (SP/DP)
- Hack: Pre-generate functions with NVIDIA's high-level toolchain (NVCC)
- Not ideal:
    - Compiler issues function calls (no inlining) and might miss optimization opportunities
    - Ships in text form with QDP-JIT (specific version of PTX)

```
.func (.reg .f64 %res) _func_sin_ptx ( .reg .f64 %arg )
{
  .reg .u32 %r<77>;
  .reg .u64 %rd<10>;
  .reg .f32 %f<94>;
  .reg .f64 %fd<5>;
  .reg .pred %p<14>;
begin:
mov.f64         %fd1, %arg;
cvt.rn.f32.f64  %f1, %fd1;
abs.f64         %fd2, %fd1;
cvt.rn.f32.f64  %f2, %fd2;
mov.f32         %f3, 0f7f800000;        // ((1.0F)/(0.0F))
setp.eq.f32     %p1, %f2, %f3;
@!%p1 bra       $Lt_0_10242;
mov.f32         %f4, 0f00000000;        // 0
mul.rn.f32      %f1, %f1, %f4;
Lt:
mov.f32         %f5, 0f3f22f983;        // 0.63662
mul.f32         %f6, %f1, %f5;
cvt.rni.s32.f32         %r1, %f6;
mov.s32         %r2, %r1;
cvt.rn.f32.s32  %f7, %r1;
neg.f32         %f8, %f7;
mov.f32         %f9, %f8;
mov.f32         %f10, 0f3fc90000;       // 1.57031
mov.f32         %f11, %f10;
mov.f32         %f12, %f1;
mad.f32 %f13, %f9, %f11, %f12;
mov.f32         %f14, %f13;
mov.f32         %f15, %f8;
mov.f32         %f16, 0f39fd8000;       // 0.000483513
...
}
```

Automatic Memory Management

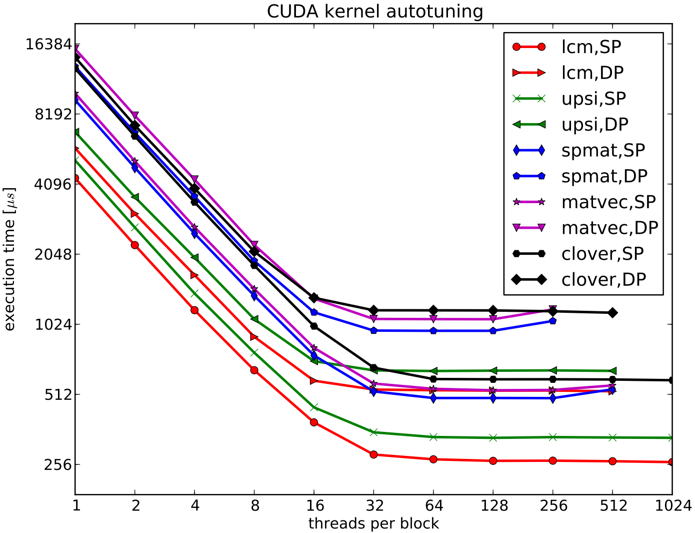
- QDP-JIT abstracts and completely hides CUDA memory management from the user
- No need for data placement annotations like e.g. in OpenACC
- Cache (LRU) implementation as a state machine
- Simultaneous PCIe data transfers and compute kernels





Auto-Tuning of CUDA Kernels

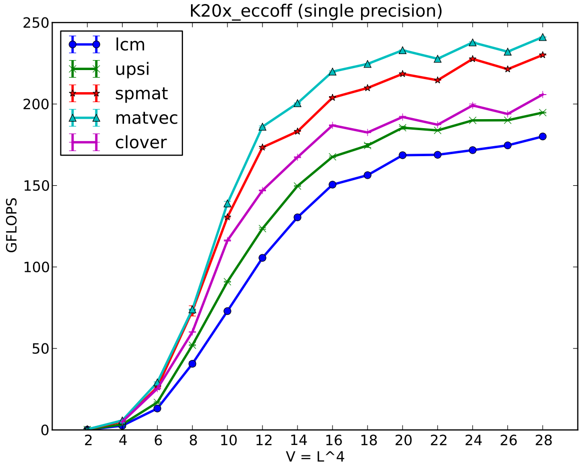
- Kernel performance depends on number of threads per block
- Auto-tuning of each kernel determines best configuration
- Carried out once for each local volume, kernel combination
- Subsequent launches use optimal settings

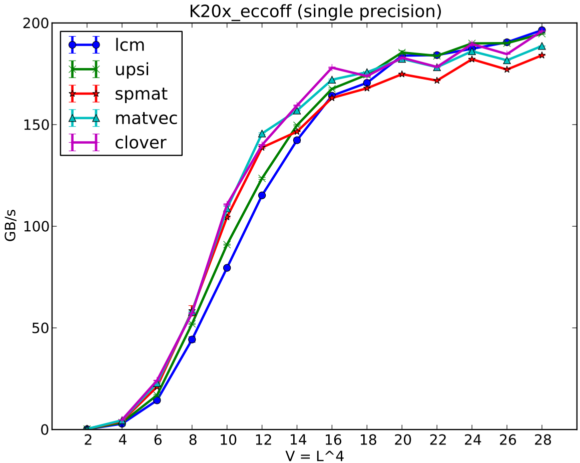


Outline

1. Introduction QDP++
2. QDP-JIT
3. **Benchmark results**
4. HMC on Titan
5. Conclusions

Kepler K20X (single precision)

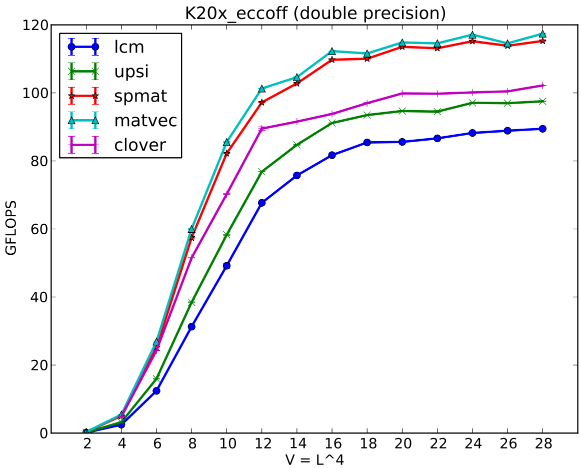


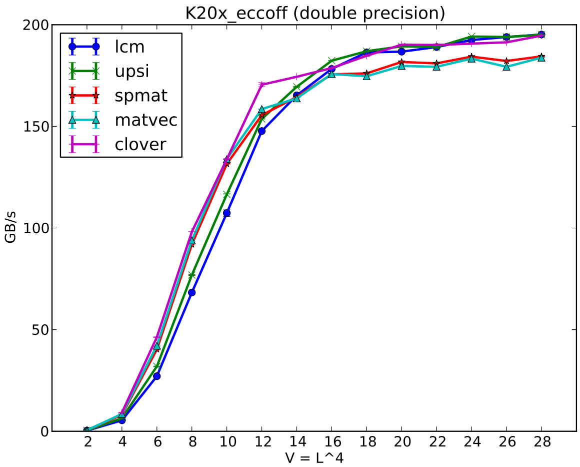


`lcm` $U_1 = U_2 * U_3$ flops/byte = 0.917 (SP), `upsi` $\psi_1 = U_1 * \psi_2$ (1.0), `spmat` $\Gamma_1 = \Gamma_2 * \Gamma_3$ (1.25), `matvec` $\psi_0 = U_1 * \psi_1 + U_2 * \psi_2$ (1.28)

- NVIDIA Kepler K20X peak bandwidth 250 GB/s. Kernels are memory bandwidth bound and achieve up to 79% of achievable performance.

Kepler K20X (double precision)



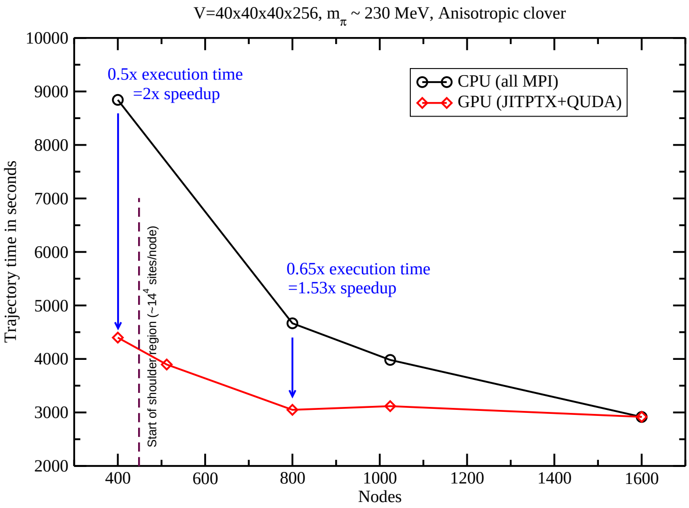


- For DP Kepler’s large register file helps to maintain the performance
- Identified performance “shoulder” at around 14^4 local volume

Outline

1. Introduction QDP++
2. QDP-JIT
3. Benchmark results
4. **HMC on Titan**
5. Conclusions

Full HMC on Blue Waters



- Still preliminary! (in this run QUDA interface via CPU memory)

Outline

1. Introduction QDP++
2. QDP-JIT
3. Benchmark results
4. HMC on Titan
5. **Conclusions**

Conclusions

- QDP-JIT provides QDP++ API with support for NVIDIA GPUs
- dynamic generation of PTX kernels
- coalesced memory accesses
- auto-tuning of kernels
- memory management completely automated
- high-level user code runs unaltered on GPUs

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