QCD Data Parallel (Expressive C++ API for Lattice Field Theory) for GPUs

Frank Winter

Jefferson Lab

GPU Technology Conference 2013
March 18-21, San Jose, California
Outline

1. Introduction QDP++
2. QDP-JIT
3. Benchmark comparison JIT-C vs. JIT-PTX
4. HMC on Bluewaters
5. Conclusion and Outlook
Outline

1 Introduction QDP++

2 QDP-JIT

3 Benchmark comparison JIT-C vs. JIT-PTX

4 HMC on Bluewaters

5 Conclusion and Outlook
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
- targets parallel clusters and supercomputers with CPU architectures

- basis of the popular LQCD software suite Chroma (421718 C++ code lines, 287 cites)
- open source software
- main software architects: B. Joó, R. Edwards (Jefferson Lab)
Multiple levels of index spaces implemented with C++ templates

C++ operator overloading on each level

Lattice site index:

- partitioned accross MPI nodes
- overloading implemented with PETE (POOMA framework)

Spin, Color, Reality index as C aggregates

```cpp
typedef OLattice< PSpinVector< PColorVector< RComplex< float >, Nc>, 4 >, Ns > 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;
```
Outline

1. Introduction QDP++

2. QDP-JIT

3. Benchmark comparison JIT-C vs. JIT-PTX

4. HMC on Bluewaters

5. Conclusion and Outlook
QDP-JIT Motivation

- 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.
- JIT (Just-in-Time) approach is necessary because QDP++ is based on Expression Templates
  (functions are assembled at compile time)
- 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 single expressions to accelerators
- Dynamic code generation: CUDA C++, or PTX (work in progress)
- Just-In-Time (JIT) compilation
  - NVIDIA NVCC, or
  - NVIDIA compute compile driver
- Automatic memory transfers via a ‘software cache’
- Automatic tuning of CUDA kernels
Expression parsing on the outer index is implemented using PETE

Generates a syntax tree of the expression with

- Operations as inner tree nodes (like `add`, `map`, etc.)
- Data fields as leaf nodes (like `LatticeFermion`)

PETE tree traversal unparses the syntax tree

Interfacing to a CUDA C++ code generator

Reuse the inner levels of QDP++ (spin, color, complex) on the device
Expression parsing on site index level implemented with PETE

Unparsing the syntax tree interfaces to PTX code generator

Jitted on all index levels (spin, color, complex) down to the register level

GNU *libjit*-like interface
  - jit values, instructions, labels, branches
  - support for PTX types, *predicates*

Virtual registers of PTX are used to eliminate temporaries in stack/memory

```
.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 10,[param0];
.ld.param.s32 11,[param1];
mov.u16 h0,%ntid.x;
mov.u16 h1,%ctaid.x;
mov.u16 h2,h1;
mov.u16 h3,h0;
mul.wide.u16 u0,h0,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;
....
```
Automatic Memory Management

- Software implementation of a cache
- operates on a memory pool (CUDA `cuMalloc` not specified as being “fast”)
- support for multiple CUDA streams
  - concurrent data transfers and kernel execution
- provides cache & lock functionality
- least recently used (LRU) spilling strategy
- frees the user from annotating code or any explicit management
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
## Key Facs JIT-C and JIT-PTX

<table>
<thead>
<tr>
<th></th>
<th>JIT-C</th>
<th>JIT-PTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA API</td>
<td>Runtime</td>
<td>Driver</td>
</tr>
<tr>
<td>Kernel code</td>
<td>CUDA C++</td>
<td>PTX 1.4/2.3</td>
</tr>
<tr>
<td>min. compute capability</td>
<td>2.0</td>
<td>1.3</td>
</tr>
<tr>
<td>JIT compile time (per kernel)</td>
<td>$\approx 2 - 5\text{secs}$</td>
<td>$\approx 10 - 500\text{ms}$</td>
</tr>
<tr>
<td>Kernel backup</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Kernel autotuning</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Memory access</td>
<td>strided</td>
<td>coalesced</td>
</tr>
</tbody>
</table>
• `mathlib` functions only as hardware fast-math versions (approximations)

• Single or double precision versions with IEEE compliant rounding not available

• Workaround:

  Using PTX code produced by NVCC and defining separate math functions

```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, 0f?f8000000; // ((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.s32.s32 %r1, %r1;
  neg.f32 %f7, %f7;
  mov.f32 %f9, %f8;
  mov.f32 %f10, 0f3fc9000000; // 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, 0f39fd80000; // 0.000483513
  ...
}
```
Benchmark Measurements (JIT-C and JIT-PTX)

- **$U$ LatticeColorMatrix**
  - $3 \times 3$ complex
  
- **$\psi$ LatticeFermion**
  - $4 \otimes 3$ complex

- **$\Gamma$ LatticeSpinMatrix**
  - $4 \times 4$ complex

**Expression** | flop/byte (SP) | flop/byte (DP)
--- | --- | ---
$U_1 = U_2 \ast U_3$ | 0.917 | 0.458
$\psi_1 = U_1 \ast \psi_2$ | 1.0 | 0.5
$\Gamma_1 = \Gamma_2 \ast \Gamma_3$ | 1.25 | 0.62
$\psi_0 = U_1 \ast \psi_1 + U_2 \ast \psi_2$ | 1.28 | 0.64

- Benchmarking hardware: Kepler GK110 (K20), Fermi GF104 (GTX480)
Kepler K20 (single precision)

- JIT-PTX (upper set of curves), JIT-C (lower set)
- Coalesced memory access achieves a good speedup
large number of registers (per thread) on Kepler architecture is beneficial
On Fermi the larger expressions (3 and 4) generate higher register pressure
1 Introduction QDP++

2 QDP-JIT

3 Benchmark comparison JIT-C vs. JIT-PTX

4 HMC on Bluewaters

5 Conclusion and Outlook
Full HMC on Blue Waters

- Full HMC is the most complex application of the QDP-JIT technology
- QDP-JIT enables LQCD gauge generation on GPU-enabled supercomputers:
  - pseudofermion refresh, momenta update, quark smearing, force terms, projection of fields, leapfrog integration, reunitarization (SU3), random number generator, exponentials of color matrices
- \(32^3 \times 96\) isotropic clover, double precision
- QUDA (Mike Clark, Ron Babich, et al.) for matrix inversion
- Data generated by Balint Joo

![Bar chart showing HMC Time for Trajectory with non-QUDA and QUDA](chart.png)
Outline

1. Introduction QDP++
2. QDP-JIT
3. Benchmark comparison JIT-C vs. JIT-PTX
4. HMC on Bluewaters
5. Conclusion and Outlook
QDP-JIT provides QDP++ API with support for NVIDIA GPUs
- dynamic generation of CUDA C++/PTX kernels
- auto-tuning of kernels
- memory management completely automated
- high-level user code runs unaltered on GPUs
- PTX work in progress
- interfacing to LLVM would avoid pain of maths functions
- 'GPU Direct' for off-node communications
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.