PROCEEDINGS OF SCIENCE

PoS(LATTICE 2013)042

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

F. T. Winter[*], **R. G. Edwards**
Thomas Jefferson National Accelerator Facility, 23606 Newport News, VA
E-mail: fwinter@jlab.org

These proceedings describe briefly the QDP-JIT/PTX framework for lattice field theory calculations on the CUDA architecture. The framework generates compute kernels in the PTX assembler language which can be compiled to efficient GPU machine code by the NVIDIA JIT compiler. A comprehensive memory management was added to the framework so that applications, e.g. Chroma, can run unaltered on GPU clusters and supercomputers.

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

[*]Speaker.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence. http://pos.sissa.it/

PoS(LATTICE 2013)042

1. Introduction

Precision lattice QCD calculations are performed on massively parallel (super)computers and there is justified desire for codes to be efficient. Compiled architecture-independent C++ code virtually never achieves a high performance [1]. A good example is the QDP++ library which provides data types and operations suitable for lattice field theory [2]. The library provides excellent and extensible functionality and makes formidable use of the generic C++ language features. However, typically it achieves a high performance only through specializations which partially removes its generic features.

QDP-JIT/PTX is a reimplementation of the QDP++ library with an identical high-level interface, but which provides on the code generating end an assembler language implementation which achieves a high performance on the CUDA architecture.

2. The QDP++ Library

The QCD Data Parallel (QDP++) library provides the data structures and implements the operations for high-level applications like Chroma [2]. The data type system provides “building blocks” consisting of up to rank-2 tensors which are combined in “levels” to build data types. The number of levels which combine the building blocks is in principle unbound, in the standard QDP++ implementation however, two levels, named after the QCD index spaces “spin” and “color”, are implemented where the number of spin and color components N_s and N_c are configurable. The elements of field variables are ascribed to the sites of a hyper-cubic lattice where the number of space-time dimensions N_d is configurable. Field variables ascribed to the links of the lattices are typically implemented as an 1-dimensional array of fields with N_d elements.

Operations on QDP++ data types are provided in operator infix-form (C++ operator overloading) on all levels of the type hierarchy. The operations are implicitly data-parallel, thus the programmatic loops over index spaces are implicitly generated by the framework. Stencil-like operations are provided which can be used as building blocks to construct, e.g., gauge invariant quantities. The lattice is implemented with periodic boundary conditions; other boundary conditions can be implemented as available, e.g., in Chroma.

Documentation and user information about the QDP++ library is available [3].

3. CUDA Architecture

The CUDA architecture is a parallel computing platform and programming model created by NVIDIA for use with their GPUs. CUDA C/C++ provides a small set of extensions to standard programming languages, like C/C++, that enables developers to implement parallel algorithms for GPUs. The Parallel Thread Execution (PTX) [4] is an intermediate assembler language for a virtual GPU architecture created by NVIDIA. This virtual architecture is entirely defined by the capabilities, or features, which it provides, i.e. its “compute capability”. For example, the compute capability defines whether the fused multiply-add or the division operation with IEEE 754 compliant rounding is available. The instruction set operates on an infinite register model which eliminates the need for any explicit stack management. NVIDIA made the documentation and interfaces to kernel development in the PTX language publicly available.

PoS(LATTICE 2013)042

The compute compiler driver translates a PTX program to machine code for a GPU architecture. It thus provides the possibility to take a PTX program which is either embedded in the executable or dynamically generated and "Just-In-Time" (JIT) compile it to the GPU architecture installed in the system.

An equivalent of the C math library implemented in PTX is not provided by NVIDIA. The developer working on the PTX assembler level is required to provide implementations of mathematical functions as needed.

4. QDP-JIT/PTX

The QDP-JIT/PTX library is a reimplementation of the QDP++ library for the CUDA architecture. It supports the same high-level interface of QDP++ (data types and operations) which enables applications such as Chroma to run unaltered on GPU clusters and supercomputers. Special emphasis was put on performance and we applied several optimization techniques as we shall demonstrate later on. In the following we will give a brief overview of its implementation for GPUs.

The Kepler architecture, in particular the K20Xm GPU, comprises a total number of 2688 CUDA cores which share 1.5 MB of L2 cache. Clearly, latency to global memory (400-800 cycles[5]) must be overcome by other means than using the cache hierarchy. Execution of an instruction with operands that are not loaded yet does not stall the Streaming Multiprocessor (SM) but triggers switching the thread context. Thus, on GPUs latency to global memory is typically overcome by a large number of resident threads which the scheduler can select for execution. CUDA thread parallelization was implemented on the level of lattice sites which provides a high occupancy if the local volume is large enough.

Kepler's ISA encoding allows for 255 GPU registers per CUDA thread. The compute compiler driver issues spilling code for registers which cannot be allocated in the register file. Spillage goes to local memory, a thread-private area of (high latency) GPU memory. Most of our generated kernels use less than 100 GPU registers on the Kepler architecture, thus register spilling does not occur. Exceptions are the Wilson DSlash operator and the trace of the plaquette $\mathrm{ReTr}(P_{\mu\nu}(x))$ which require in double precision 249 resp. 158 GPU registers thus still do not trigger spilling.

Another object competing for local memory is the threads' stack space. A consequence of (global) C++ operator overloading, as used in QDP++, is the allocation of return values on the program stack. In principle optimizing compilers should be able to eliminate these stack allocations. However, our experience with previous implementation of QDP++ for CUDA, namely QDP-JIT/C [6, 7], and our experience with the original QDP++ library is that C++ compilers are not able to eliminate stack allocations completely. QDP-JIT/PTX generates CUDA kernels in the PTX assembler language where stack allocations are eliminated by unrolling the operations to the register level.

We implemented a templated PTX code generator leveraging the Portable Expression Template Engine (PETE) [8] parametrized on the expression. Instantiation of the code generator yields C++ code which, when triggered, generates PTX code for the expression in question [9]. Code is generated for all levels of the type hierarchy which effectively eliminates the temporaries otherwise created as intermediate results. Subsequently the compute compiler driver translates the PTX

PoS(LATTICE 2013)042

code to executable GPU code. This procedure was found to execute smoothly. Neither the code generation, nor the subsequent compilation by the compute compiler generate a significant amount of overhead.

The QDP-JIT/PTX library includes comprehensive CUDA memory management. We leverage PETE to ensure that all referenced field variables within an expression are available in GPU memory prior to launching a kernel. Due to the dynamic nature of the memory access pattern (I/O routines, etc.) we implemented a "cache" for field variables which can spill least recently used fields to CPU memory in order to allocate new fields on the GPU. As a result CUDA memory management is completely automated and altering or annotating the application source code with memory copy routines is not needed.

PoS(LATTICE 2013)042

5. Optimizations

An optimization technique which we applied is changing the data layout for field variables to suit the GPU architecture. The original QDP++ library uses arrays of structures (AoS) for its data types which also determines the data layout. This layout is suitable for a scalar CPU architecture where relatively few threads can be served with a few cache lines. The GPU differs fundamentally here. The load and store units are activated simultaneously at thread warp granularity and the highest memory bandwidth is achieved when the memory accesses are coalesced. In QDP-JIT/PTX we use a data layout that results in coalesced memory accesses:

$$I(i_V, i_S, i_C, i_R) = ((i_R * I_C + i_C) * I_S + i_S) * I_V + i_V$$

where I_V,I_S,I_C,I_R are the index domains for, and i_V,i_S,i_C,i_R are the indices within these domains for the space-time, spin, color and complex component respectively. The coalescing condition can be read off when assuming thread-parallelisation takes place on the space-time index domain, i.e. adjacent threads (thread number i_V) access adjacent memory locations.

The generated kernels are not exclusively pure streaming kernels, i.e. kernels including non-local quantities like $U_\mu(x)\psi(x+\hat{\mu})$ incur non-coalesced memory accesses. The performance of these kernels is a non-trivial function of the launch geometry. For these compute kernels we use a low-overhead auto-tuning step. This auto-tuning determines experimentally a thread block size which maximizes the kernel performance.

On distributed memory systems evaluating non-local quantities introduce data dependencies on lattice sites resident to neighboring compute nodes. This provides the opportunity to overlap off-node (MPI) communication with kernel computation. Since the node-local lattice is logically shaped as a N_d-dimensional hypercubic grid the face sites, i.e. the sites to send off-node, can be easily determined for such quantities. The node-local lattice is decomposed into "inner sites" and "face sites". While off-node data is in transition the inner sites are evaluated first and after data has been received the remaining sites are evaluated. Our overlapping implementation is limited to nearest neighbor operations only. Quantities involving next-to-nearest neighboring site communications, like $\psi(x+\hat{\mu}+\hat{\nu})$, execute the right-most offsets in a non-overlapping fashion.

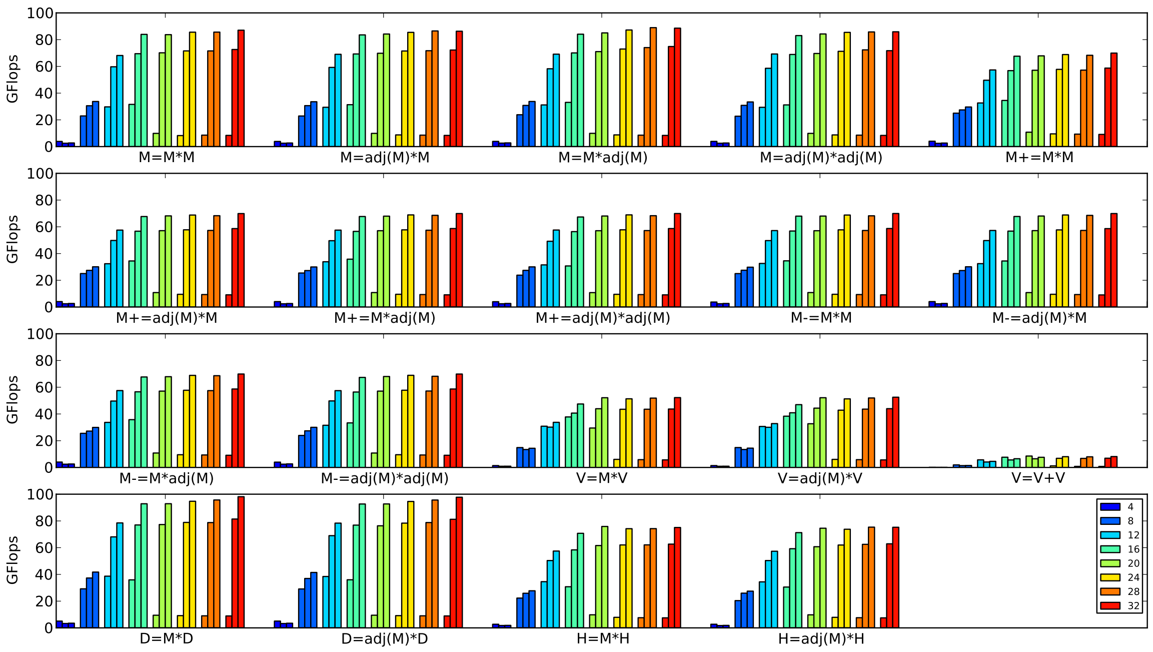

Figure 1: Performance comparison of QDP++ on Intel Sandybridge CPUs (left bar) and QDP-JIT/PTX on single NVIDIA K20Xm (middle bar) and K40m (right bar) in double precision. For each benchmarking function the performance for several volumes was measured. We use the following abbreviations: M for SU(3) field $U_\mu(x)_{a,b}$, V for "color vector" $V(x)_a$, D for Dirac fermion $\psi(x)_{\alpha,a}$, H for Dirac half fermion $\psi(x)_{\alpha',a}$. The number of lattice sites is given by $N_{\mathrm{vol}} = L^4$ with L given in the legend.

6. Performance Measurement

Our experimental results were acquired on the Jefferson Lab 12k cluster. Nodes of this cluster contain dual-socket Xeon E5-2650 CPUs (total of 16 cores) running at 2.0 GHz. Each processor comprises 20 MB L3 cache. The nodes run CentOS 6.2. We used two nodes, one node with four NVIDIA K20Xm GPUs and another node with four K40m GPUs (GK110(B) architecture) installed. The K20Xm (K40m) has as a peak performance of 1.3 (1.4) TFlops (DP) and a peak memory bandwidth of 250 (288) GB/sec (ECC disabled). We used the CUDA toolkit version 5.5 and the NVIDIA UNIX x86-64 kernel module version 319.37.

We use the `t_linalg` program which is included as one of the QDP++ example applications. This program measures the performance of several linear algebra kernels. We compare the performance achieved by QDP-JIT/PTX utilizing a single GPU of the installed GPUs on this compute node with that of standard QDP++ utilizing both CPUs. We use an SSE-optimized version of the "parscalar" architecture in double precision and use 32 OpenMP threads.

Figure 1 shows the performance comparison of QDP++ on the Sandybridge CPUs and QDP-JIT/PTX on a single NVIDIA K20Xm and K40m in double precision. The general observation is that for "sufficiently large" problem sizes the GPU performs much better than the CPU. The CPU falls out of cache for volumes larger than 16^4 sites. Note that this benchmark has an "artificial character" since it exercises the same evaluation repeatedly. The CG algorithm with an even-

PoS(LATTICE 2013)042

odd preconditioned pure Wilson operator in single precision falls on this system already for local volumes larger than 12^4 out of cache. The GPU performs better nearly throughout starting from 8^4 sites. In the regime where the CPU is still within cache the K20Xm resp. K40m provide speedup factors of ≈ 2.2 resp. ≈ 2.7 for a single GPU. We carried out weak scaling tests to the full compute node configuration and observed perfect scaling, thus another factor 4 is achieved when considering the whole compute node. The regime of larger local volumes is where the GPUs provide huge speedup factors. For the 32^4 volume a single K40m achieves a speedup factor of ≈ 10 over the CPUs.

We found that on the NVIDIA Kepler architecture the generated compute kernels sustain 79% of the peak memory bandwidth, which is the appropriate metric of efficiency, since the kernels in LQCD computations are memory bandwidth bound on this architecture.

7. Conclusion

The QDP-JIT/PTX framework pursues a quite sophisticated approach in order to provide an efficient implementation of the QDP++ library targeting the CUDA architecture. The PETE library is leveraged to build code generators for the PTX assembler language. A comprehensive CUDA memory management was added to the library so that applications can run unaltered on GPU clusters and supercomputers.

8. Outlook

QDP-JIT/PTX is an evolving framework which will integrate beneficial features and optimizations as they become available. This includes for example new CUDA features like “unified memory”.

We are looking into “expression fusion” in cases where there’s still head room in the register file to hold a subsequent evaluation operating partially on the same fields as the previous one.

Further optimizations can be applied by exploiting the algebraic properties of the SU(3) group and represent an element with fewer floating-point numbers. The QUDA library [10] applies this successfully.

Work is underway to interface our code generator to the compiler framework LLVM. This will allow us to target other architectures like x86, IBM Blue Gene/Q and Intel Xeon Phi.

9. Code Availability and Compatibility

The latest development version of the QDP-JIT/PTX library is always available in a publicly-accessible source code repository [11]. Applications using the QDP++ library can be build on top of QDP-JIT/PTX instead without any modifications to the source code. NVIDIA GPUs with compute capability 2.0 or higher are supported.

10. Acknowledgment

We acknowledge the use of computing facilities of the US National Computational Facility for Lattice Gauge Theory, in particular the GPU accelerated clusters at Jefferson Lab.

PoS(LATTICE 2013)042

Partial support for this work was provided through the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Offices of Advanced Scientific Computing Research, Nuclear Physics and High Energy Physics.

This research was in part supported by the Research Executive Agency (REA) of the European Union under Grant Agreement number PITN-GA-2009-238353 (ITN STRONGnet).

FW acknowledges support from the PRACE-2IP project under grant number EC-RI-283493.

References

[1] P. A. Boyle, The BAGEL assembler generation library , Comp. Phys. Comm. 180 (2009) 2739.

[2] R. G. Edwards, B. Joó, The Chroma software system for lattice QCD, Nucl.Phys.Proc.Suppl. 140 (2005) 832. arXiv:hep-lat/0409003, doi:10.1016/j.nuclphysbps.2004.11.254.

[3] QDP++ Documentation and User Information, [Online; accessed 10/18/2013].
URL `http://usqcd.jlab.org/usqcd-docs/qdp++`

[4] NVIDIA, Parallel Thread Execution ISA Version 3.1.
URL `http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf`

[5] Fundamental optimizations, [Online; accessed 10/18/2013].
URL `http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Fundamental_Optimizations.pdf`

[6] F. Winter, Accelerating QDP++/Chroma on GPUs, PoS LATTICE2011 (2011) 50. arXiv:1111.5596.

[7] F. Winter, Gauge Field Generation on Large-Scale GPU-Enabled Systems, PoS LATTICE2012 (2012) 185. arXiv:1212.0785.

[8] S. Haney, J. Crotinger, S. Karmesin, S. Smith, Easy expression templates using PETE, the Portable Expression Template Engine, Technical Report LA-UR-99 (1999) 777.

[9] T. L. Veldhuizen, C++ Templates as Partial Evaluation, CoRR cs.PL/9810010.

[10] [Online; accessed 10/27/2013]. [link].
URL `https://github.com/lattice/quda`

[11] [Online; accessed 10/18/2013]. [link].
URL `https://github.com/fwinter/qdp-jit`

PoS(LATTICE 2013)042