Writing Efficient QCD Code Made Simpler: QA_0

Andrew Pochinsky[*]
Massachusett Institute of Technology
E-mail: `avp@mit.edu`

A new tool for writing platform-independent optimized QCD code, QA_0, is described. Performance of a Möbius Domain Wall Fermion inverter written with qa0 on several platforms is presented.

The XXVI International Symposium on Lattice Field Theory
July 14-19 2008
Williamsburg, Virginia, USA

[*]Speaker.

1. Introduction

Presently there are many different computer architectures available for Lattice QCD caclulations. The June 2008 Top 500 list[1] contains four different architectures amongst a hundred fastest computers in the world. It is likely that while some architectures will move out of the list, the situation will likely remain fundamentally the same as new players enter the field, especially as the race to petaflops scale processes.

The above observation makes it attractive to be able to run LQCD code on many different architectures. However, there is a catch. While the most general C code could be ran unmodified on a new machine, in practice its performance will be not very impressive; to harvest as many compute cycles as possible, one has to tweak the code, sometimes to a considerable degree, for a new machine.

Code tuning is expensive in human time and is notoriously prone to errors. At the one extreme one would have to write the core algorithms in the machine lanugage at the price of staying locked to a particular architecture. On the other hand, modern compilers are getting better over time, but they require help from an application programmer to generate efficient code. Unfortunately, this approach does not completely solves the problem. One ends with a set of application codes that should solve the same physics problem on different architectures. Code maintenance quickly becomes a major challenge.

This work addresses the issue of writing high performance easily portable code for LQCD for modern computer architectures.

2. QA_0, a QCD register transfer language

2.1 The Problem

At the heart of every QCD code is computing a gauge-fermion product

$$\phi[x] \leftarrow (1+\gamma_\mu)U_\mu[x]\psi[x+\hat{\mu}]$$

One would expect that this is small enough piece of code that could not not possibly create a portability problem. However, in any QCD application variations on the above theme are required. The variations quickly lead to a combinatorial explosion: e.g., the QLA library [2] has more than 2500 routines for each value of N_c and precision.

Another issue that needs to be addressed is a mismatch between a data-parallel structure of QCD applications and the nature of modern computer designs. The problem is that modern processors do not work very well with the data-parallel paradigm, because speed of memory access relative to computational speed keeps decreasing. Instead, one should try exploit data caches by reusing recently access data as much as possible. For example, instead of writing a sequence of QDP/C calls, a better performing code would invert the loops—do all operations in the sequence on one lattice site before going to the next. This approach will win big if it could be work for routines like a Dirac operator and a preconditioned CG matrix. Unfortunatelly this is not possible to do within the QDP library framework.

2.2 A solution

QA_0 is a tool that empowers a QCD programmer to quickly write code that is “missing” in QDP/C. E.g., the following snippet of QDP

```
a0 = QDP_create_D();
QDP_D_veq_spproj_M_times_D(a0, U0, s0, 1, +1);
QDP_D_vpeq_spproj_Ma_times_D(a1, a0, U1, s1, 2, -1);
QDP_destroy(a0);
```

could be written in *QA_0* as follows

```
loop (s = 0 ..< site) {
  tmp = $pointer.copy(a1);
  loop (d = 0 ..< Ls) {
    (tmp2,s0) = $qcd.mul.spproj.1.plus(tmp, U0, s0);
  }
  loop (d = 0 ..< Ls) {
    (a1,s1) = $qcd.madd.spproj.2.minus(a1, U1, s1);
  }
}
```

The *QA_0* compiler produces efficient code from the same input on all supported architectures.

2.3 *QA_0* language

The goal of *QA_o* is to provide an abstraction layer between HLL and machine code. It is achieved by defining a QCD RISC-like abstract machine as follows:

- All memory accesses are explicit. There are only two operations on memory: `load` and `store`. All other operations take inputs from registers and place results into registers.
- Non-memory operations have no effects other than producing results.
- Control flow is explicit. There are loops, forward branches and returns in the present version. Only leaf procedures are supported.
- In addition to integer, pointer and floating point registers, there are “registers” for common QCD data: gauge matrices, Dirac fermions, staggered fermions, projected fermions and complex numbers.
- Both single and double precision is supported.
- Instruction set is extensible with user level macros to simplify coding.
- Like HLL, the same input language is used to generate code for different backends.

In fact, all QCD operations are defined as macros which are transformed into the underlying machine instructions by the *QA_o* compiler. Presently the following backends are implemented:

C99–32 Standard C with built-in complex types on 32-bit machines.

C99–64 Standard C with built-in complex types on 64-bit machines.

Cee–32 Traditional C (complex operations expanded into floats) on 32-bit machines.

Cee–64 Traditional C (complex operations expanded into floats) on 64-bit machines.

xlc–BG/X BG/L and BG/P with the Double Hummer support using IBM XLC as a backend.

3. Application Example: MDWF

The QA_0 approach to code generation was tested on Möbius Domain Wall Fermion code. Results are encouraging: all platforms run quite well in the initial C-backend implementation of QA_0 while there is no platform-dependent code outside the QA_0 compiler. The code is organized as a Level III library for the SciDAC LQCD infrastructure.

The MDWF code follows previous Domain Wall Fermion code in using SciDAC LQCD libraries [2] for communication. The code itself is organized into a library of operators and inverters (a Dirac CG solver, preconditioned CG, and shifted CG are provided), as well as helper routines to do linear algebra operations on fermion fields. For optimization purposes the code uses its own data layout, therefore, the interface provides import and export routines.

Both single and double precision versions are implemented and could be used in the same application. On platforms where there is a significant difference in performance this allows one to get an approximate solution of the Dirac equation in single precision and then quicky "polish" the result into double.

The code consists of two parts:

- Top level functions provide the user's interface. All routines are written in C. Code consists of 5000 lines of portable C.
- Low level functions implements all floating point operations. This code consists of 2500 lines of QA_0.

Currently all supported platforms share QA_0 code. Same QA_0 code is used to generate both double and single precision variants of the MDWF.

4. Performance comparison

We compare performance of the MDWF inverter on different parallel systems. In case of Blue Gene (both /P and /L) [4], IBM's XLC compiler was used as a back-end for QA_0. Though it does not produce the most optimal code, it is sufficiently good for our purposes. In all cases QMP over MPI was used for communication. For the SiCortex machine [5] we also used the vendor's compiler. In both cases it is possible to generate a better code for QCD than the current compilers are capable of, however, the required effort may be justified if significant time is available on a respective machine.

We show both weak scaling behavior and strong scaling on $32^3 \times 64 \times 16$ lattice.

4.1 Strong Scaling

For a given problem size strong scaling shows how far a job could be stratched accross multiple processors. We used our production lattice size, $32^3 \times 64 \times 16$, in this study.

BlueGene/L is used in the virtual node mode. The machine peak is 2800MFlops/cpu. One can clearly see that double precision performance is limited by the memory bandwidth.

#cpu	float, MFlops/cpu	double, MFlops/cpu
256	678	463
1024	734	505
2048	657	429

BlueGene/P is also used in virtual node mode. The machine peak is 3400MFlops/cpu. It uses a different version of XLC; the architecture is sufficiently different from BG/L to merit a careful study.

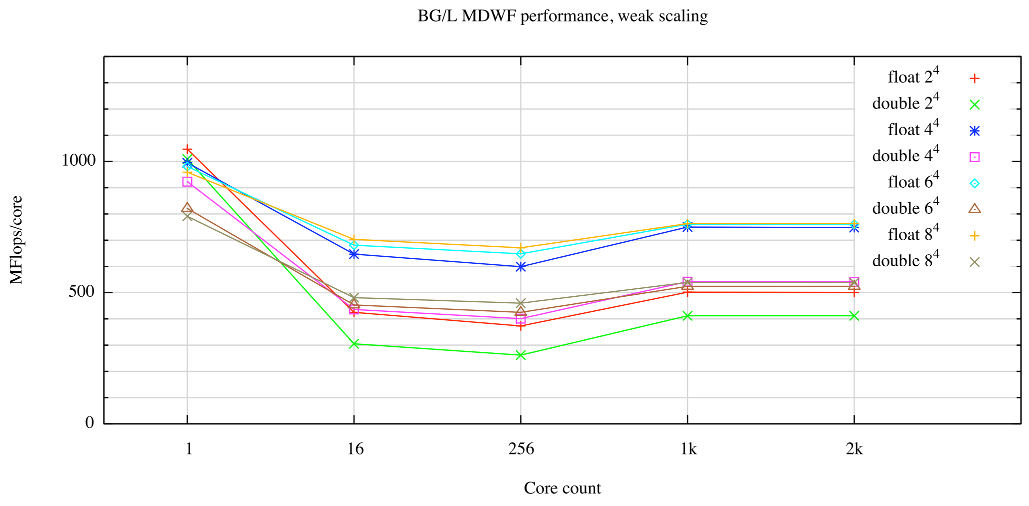
#cpu	float, MFlops/cpu	double, MFlops/cpu
256	814	626
512	808	620
1024	755	561
2048	802	609
4096	813	622

Initial tests were run on the SiCortex series of machines. The machine peak is 1000MFlops/cpu. The vendor's C compiler was used as well as the original MPI implementation.

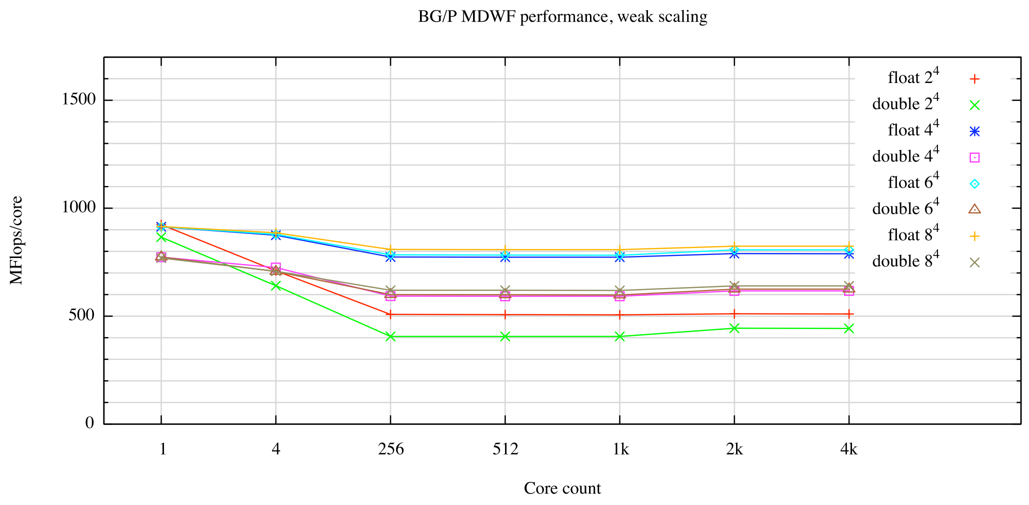
#cpu	float, MFlops/cpu	double, MFlops/cpu
128	195	181
256	193	176
512	191	173

4.2 Weak Scaling

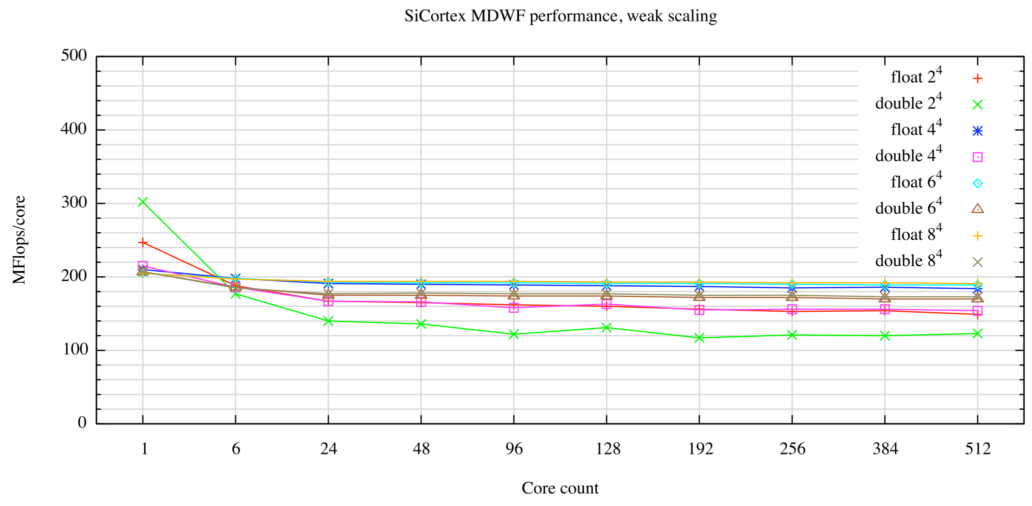
For weak scaling studies we keep the local lattice size fixed and change the machine size. A single node performance shows how well the processor is utilized, while various machine sizes show effects of the network on the perfomance.



On the BG/L one sees a drop in performance for 16 and 256 cores due to lack of torus in the network for these block sizes. Memory bandwidth limitations result in performance difference between single and double precision.



On BG/P small blocks do not show perculiar behavior, likely because of changes in the network hardware. It is worth noticing that the share of the peak is somewhat lower than on BG/L.



On Sicortex, performance is also remarkably flat. Relative low share of peak achived is caused by a lack of hardware prefetch capabilities in the processor.

5. Conclusions

QA$_0$ is a valuable tool for writing Level III LQCD routines. By combining platform independence and exposing a low level processor abstraction to the application programmer, *QA*$_0$ makes it possible to achieve performance of hand crafted code without writing in the assembly language.

The system is being extended to include target architectures of interest to the lattice QCD community, including the SSE and the CBE.

Overall design of *QA*$_0$ makes is suitable for other application domains. This direction is being exploited as well.

6. Acknowledgements

This work was supported in part by the DOE Office of Nuclear Physics under grant DEFG02-94ER40818. This research used resources under the INCITE and ESP programs of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357, resources provided by the DOE through its support of the MIT Blue Gene/L. IBM has provided early access to BG/P. Sicortex Inc. has graciously provided access to its hardware for scaling studies.

References

[1] `http://www.top500.org/`

[2] `http://www.usqcd.org/`

[3] `http://www.mit.edu/~avp/mdwf/`

[4] `http://www.ibm.com/`

[5] `http://www.sicortex.com/`