An optimization toolkit for the Blue Gene and subsequent generations of SciDAC computers

A V Pochinsky
Laboratory for Nuclear Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, USA
E-mail: avp@mit.edu

Abstract. A tool for writing platform-independent optimized computational routines, QA0, is described. Application to optimizing SciDAC Lattice QCD codes is presented.

1. Introduction
Large scale computation relies heavily on floating point operations. Most of SciDAC applications are numerical codes that run on the largest computers in the world. Making efficient use of computer power is of the utmost importance.

The Top 500 List [1] contains four different architectures within the first one hundred fastest computers. While some architectures will move out of the list, the situation will likely remain fundamentally the same as new players enter the field. This observation makes it attractive to be able to run an application on may different architectures. However, there is a catch. While the most portable C code could be run unmodified on a new machine, in practice one can not expect to achieve good performance without an extra effort. Code tuning for a new machine is expected, sometimes at a considerable cost. Code tuning is time consuming, breaks portability, and is prone to subtle bugs. If, however, the benefits of higher performance outweigh these problems, one has to choose between several extremes. Writing critical parts of the code in a machine language is one extreme. Another approach is to use compiler-specific pragmas and language extensions to improve code generation. In addition to breaking portability, this method exposes dark corners of the compiler implementations where bugs often lurk.

In this work we explore another approach. An abstraction layer below the high level language and machine code is designed and implemented that allows the programmer to express a limited set of computation in a portable way that however lends itself to compilation into highly optimized code.

2. Application domains
A large and important class of applications could be expressed in terms of a massively data parallel model, e.g. highly successful USQCD application suite makes the data parallel model central to its application domain libraries [2]. Operations on arrays are performed element-wise with well localized regular communications in form of shifts and reductions. On modern parallel systems one usually does not write the data parallel code explicitly. Instead, the program is converted by hand into an MPI program assuming fat nodes and reliable communication
between the nodes. In addition, physical problems often exhibit data locality: to make progress
computation only needs to access a limited set of neighbors of a given element. In addition,
presently lattice quantum chromodynamics (LQCD) algorithms experience fast evolution, thus
making the area a good testbed for tool development.

2.1. Quantum chromodynamics
The QCD code spends more than 90% of its time a computation of the following kind:

$$\forall x : \phi^{a\alpha}[x] = \sum_{\mu} \left( (U_{\mu}[x])^{\alpha}_{\beta} (\delta^{\alpha}_{\beta} + \gamma^{\alpha}_{\beta}[\mu]) \phi^{b\beta}[x + \hat{\mu}] + (U_{\mu}^{\dagger}[x - \hat{\mu}])^{\alpha}_{\beta} (\delta^{\alpha}_{\beta} - \gamma^{\alpha}_{\beta}[\mu]) \phi^{b\beta}[x - \hat{\mu}] \right).$$

Computations like this one lend themselves very well to parallel execution with plenty of
opportunities to overlap computation and communication. Here we concentrate on the
computation part, since communication is to the nearest neighbors and standard tricks of
overlapping computation and communication are simple to apply by hand.

When matrix operations are unrolled on a site, the resulting basic blocks become long enough
to benefit from instruction scheduling. There is no need to guess which loops should be unrolled.
In addition, LQCD algorithms are insensitive to the details of floating point operations: for
code transformation purposes one does not need to worry about round off details: floating
point operations may be presumed exact and associative. Overall, the LQCD code presents
more optimization opportunities than the standard semantics of a programming language would
allow.

3. QA$_0$ abstraction layer
Instead of trying to twist the compiler into doing optimization we want, or making a programmer
into unrolling the loops and making code transformations by hand, we have designed an
abstraction layer, QA$_0$ that does exactly that. It takes as an input a high level language
procedure and slavishly does a fixed set of loop unrolling and constant foldings specific to the
application domain. The resulting register transfer representation of the procedure is converted
into a form suitable for a particular target machine.

The QA$_0$ input language has following features:

- 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 procedure 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.

The language also includes a hygenic macro system that was found invaluable in writing long
procedures.

3.1. Target application domains
Currently QA$_0$ supports only LQCD as an application domain. Other areas can be supported
in a straightforward way. One only needs to define application specific types, operations and a
set of transformation rules to reduce the operations on objects of these types into a simple form.
3.2. Supported targets
Currently QA0 supports the following targets:

**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.

Target **Cee–32** is used extensively for development and testing and serves as a reference for verification of other back ends.

**C99–32** and **C99–64** are comparison targets to cross check performance of other back ends. At present code generated for complex arithmetics in C compilers is rather uneven: small changes in a procedure source result in drastic changes in performance numbers.

For Blue Gene computers all complex arithmetic is expanded into XLC’s Double Hummer intrinsics. This way we benefit from instruction scheduling and code generation while avoiding hand coding. XLC also take care of the C API, thus simplifying the QA0 back end.

3.3. QA0 Example
As an example of the QA0 procedure, coding of

$$\forall x : \psi^{a\mu}[x] \leftarrow s^{a\mu}[x] + U^{a_b}[x] \phi^{b\mu}[x]$$

looks as follows:

```scheme
(procedure sample ([stem "sam"] count-flops)
  ([psi pointer "psi" "struct Fermion *"]
   [s double "s" "double"]
   [zeta pointer "zeta" "const struct Fermion *"]
   [U pointer "U" "const struct Gauge *"]
   [phi pointer "phi" "const struct Fermion *"]
   [len int "len" "int"])
(loop () ([i (const 0) (reg len)])
  (load qcd-fermion () r-zeta ([reg zeta]))
  (load qcd-fermion () r-phi ([reg phi]))
  (load qcd-qcd-su-n () r-U ([reg U]))
  (op qcd-mulf () (r-U*phi) ([reg r-U] [reg r-phi]))
  (op qcd-madd-fermion () (r-psi) ([reg s] [reg r-zeta] [reg r-U*phi]))
  (store qcd-fermion () ([reg psi]) [reg r-psi])
  (op pointer-add () (psi) ([reg psi] [const (size-of Fermion)]))
  (op pointer-add () (zeta) ([reg zeta] [const (size-of Fermion)]))
  (op pointer-add () (phi) ([reg phi] [const (size-of Fermion)]))
  (op pointer-add () (U) ([reg U] [const (size-of SU-n)])))
```

The S-expression surface syntax is presently used because the tool is still in flux; once semantics requirements to the system are better understood a more traditional C-like syntax will be provided as well.
Table 1. BG/P strong scaling behavior.

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

Table 2. SiCortex strong scaling behavior.

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

4. Results
We compare performance of one of the QCD inverters on different parallel systems. In the case of Blue Gene /P [3], IBM’s XLC compiler was used as a back-end for QA0. 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 [4] 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.

4.1. Strong scaling
For a given problem size strong scaling shows how far a job could be stretched across multiple processors. We used our production lattice size, \(32^3 \times 64 \times 16\), in this study. Tables 1 and 2 show per processing element performance on BG/P and SiCortex computers respectively.

4.2. Weak scaling
For weak scaling studies we keep the local problem 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 performance.

The BG/P performance is shown on figure 1. One can see that communications–computation separation model really works: for all local sizes performance scales linearly with the machine size. (One node data point serves as a reference, four nodes case is peculiar to the BG/P—it does not involve the torus fabric at all.)

On SiCortex, performance is also remarkably flat, see figure 2. Relatively low share of peak achieved is caused by a lack of hardware prefetch capabilities in the processor.

5. Conclusions
QA0 is a valuable tool for writing performance critical 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.

Overall design of $QA_0$ makes is suitable for other application domains. This direction is being exploited as well.

The tool is available under an open source license at the SciDAC Outreach LQCD page [5].

Acknowledgments
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/software.html
[3] http://www.ibm.com/
[4] http://www.sicortex.com/
[5] https://outreach.scidac.gov/frs/download.php/125/qa0-1.4.0.tar.bz2