PROCEEDINGS OF SCIENCE

Introduction to the Quantum Expressions (QEX) framework

James C. Osborn*
Leadership Computing Facility
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439, USA
E-mail: osborn@alcf.anl.gov

Xiao-Yong Jin
Leadership Computing Facility
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439, USA
E-mail: xjin@anl.gov

We present a new lattice field theory software framework designed with ease of use, flexibility, efficiency and portability in mind. The framework is written using the Nim programming language which offers many of the features one would find in a high-level scripting language, while in fact being a strongly-typed language with full control over low-level optimizations. This allows us to present a simple expression-based language to the end user that can be transformed into highly optimized code for a particular architecture. We will discuss the features of the QEX framework, performance results and development plans.

34th annual International Symposium on Lattice Field Theory
24-30 July 2016
University of Southampton, UK

*Speaker.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

PoS(LATTICE2016)271

Over the past 15 years, the USQCD collaboration has been developing a comprehensive set of libraries and application codes designed to provide a convenient abstraction over the common operations performed in lattice field theory. These codes have been ported and optimized over a large range of architectures. The design strategy called for a set of software "levels" with the higher levels building on top of the lower ones.

The main interface intended for algorithm development is the Data Parallel layer. This layer presents an abstraction over the lattice fields and handles their distribution across the machine along with common math, communications and I/O routines. The data parallel layer was developed as two separate libraries, one in C (QDP [1]) and one in C++ (QDP++ [2]). The C++ library uses expression template (ET) techniques to allow data parallel operations based on mathematical expressions to avoid lattice-wide temporary fields when possible. The use of ET was still relatively new at the time and would often encounter compiler bugs in heavily templated code, making their use difficult until compilers improved. To mitigate this risk, it was decided to also pursue a pure C option (QDP along with the companion linear algebra library QLA) which contains a large set of field operations that are generated by Perl scripts making generation of all the variants easier.

On top of the QDP C libraries, a library (QOPQDP) of common routines such as solvers, link smearings and force terms was developed. This library also has a multigrid solver for Wilson clover fermions [3]. In order to speed up application development, a set of applications were developed that provided access to the USQCD C libraries through a scripting language. These applications used the Lua scripting language [4] due to its ease of embedding into applications and simple yet powerful features. Qlua [5] provided a Lua wrapper on the QDP library, while FUEL [6] provided access to the QOPQDP library along with parts of QDP.

The Lua scripting layer provides a quick and easy way to develop code and is especially convenient for things like algorithm exploration or sophisticated analysis workflows. It provides a simpler environment for testing code since the scripts can be run without any further compilation necessary (once the main application is compiled), and adding new files doesn't require modifying any build infrastructure, such as makefiles or header files. For operations that have optimized C routines available, there is no performance disadvantage, however, for routines that aren't already available, constructing them from within the scripting layer may not be as efficient as if they were constructed directly at the C level.

While the USQCD C/Lua frameworks have been very successful in providing rapid development of efficient algorithm and application code, the basic design is not as suitable for future architectures. The QDP and QLA layers were built on an Array of Structures layout that makes vectorization difficult. QLA has gained threading through OpenMP, but this requires starting a parallel region at every site loop. Furthermore it is desirable to find a way to provide a high-level, easy to use script-like interface that can also have full control over low-level optimizations, so as to avoid a need for separate high-level and low-level interfaces. It was therefore desirable to do a full redesign of the software frameworks.

The initial work on the redesign began with experiments in the design of a new low-level library with full threading and vectorization in mind. The end result of this was the proof-of-concept QLL library [7] which contains new layout and communications routines along with some hand tuned C code. The performance of the Naik staggered CG solver available here is up to 23% of peak on BG/Q. With the base framework achieving good performance, work shifted to finding a

PoS(LATTICE2016)271

way to provide a new high-level interface.

The main design goal for the high-level interface was to have the ability to transform natural expressions into well optimized code. We also wanted to have the ability to perform optimizations across multiple expressions (e.g. loop fusion). Additionally, the simplicity of working with a scripting language (i.e. the simple build environment, and avoid duplication such as with header files and code files), was a highly desirable feature. After an extensive search of language and code-generation options, we settled on the Nim [8] language as the best option.

1. Nim

Nim is a modern programming language that first appeared in 2008. The original goal was to develop a safer alternative to C/C++, but since then it has continued to evolve an impressive list of features and has attracted a growing interest from users and developers. The design goal is to be "efficient, expressive, and elegant". It borrows many of its features from a large list of languages including Modula-3, Delphi, Ada, C++, Python, Lisp and Oberon. It is a statically typed language, but has extensive type-inference, so it retains the look and feel of a dynamically-typed scripting language. Some of its key features are an (optional) efficient garbage collector, extensive metaprogramming support (nearly full language available at compile time), module based imports (with no separation of headers and code), integrated build system and it compiles to C or C++ code.

The fact that Nim compiles to C or C++ provides excellent portability. All one needs to compile Nim code is a working C99 compiler. In addition the C/C++ code generation makes it easy to integrate Nim code with other libraries. One can easily declare C/C++ types and functions in Nim, then they can be used in Nim code and it will generate the appropriate C/C++ code. This also allows one to use all the language extensions available to the C/C++ compiler such as intrinsics (e.g. SIMD) and pragmas (e.g. OpenMP, OpenACC). In addition it is also possible to influence the C code generator so as to generate proper CUDA kernels to run on a GPU [9].

The integrated build system makes compiling applications very easy. It will automatically track module dependencies, compile all the Nim modules to C/C++ code, compile those to object files (using the specified C/C++ compiler and flags), then link them together into an executable. There are no separate build files that need to be maintained to specify what sources and steps are needed to make the executable. For example if you want to create a new project from an existing one, you can just copy the main program file, make some changes, then compile each program with the command "`nim c myProject.nim`".

Another of the advantages of using Nim is its extensive metaprogramming support. Figure 1 lists Nim's generic and metaprogramming features. Like C++ it has built-in support for generic programming (similar to C++ templates). It also features inline code substitutions somewhat like C++ preprocessor macros. However Nim's version is an integral part of the language and designed to be safer. They also allow overloading on argument types, which C++ doesn't provide. Lastly Nim has an extremely powerful macro system which is modeled after lisp. The body of the macro executes at compile time and gives one access to the syntax tree of the arguments to the macro. One can then inspect and manipulate the tree and return a new syntax tree that will be reinterpreted as if it were originally part of the input code. This ability has no counterpart in C++, and allows for a wide range of code transformations for optimization and the creation of easy to use domain specific languages (DSLs).

PoS(LATTICE2016)271

C++	Nim
preprocessor macros	templates: - inline code substitutions - also allows overloading, completely hygienic (if desired)
templates	generics: - applies to types, procedures, templates and macros - also allows typeclasses, concepts
N/A	macros: - similar to lisp: syntax tree of arguments passed to macro at compile time to allow arbitrary inspection and manipulation

Figure 1: Generic and meta-programming features of Nim along with C++ counterparts.

One example of macro use is transforming loops at compile time. Consider the Nim loop

```
for i in 0..2:
  foo(i)
```

which generates code to iterate `i` from 0 to 2 at runtime and call the function `foo` on `i`. We can define a macro `forStatic` to unroll the loop at compile time. Using this macro could look like

```
forStatic i, 0, 2:
  foo(i)
```

This could be expanded to

```
block:
  foo(0)
block:
  foo(1)
block:
  foo(2)
```

The **`block:`** statements are inserted to create a new scope in case the body of the loop were to create new local variables. Without the new scope it would result in a redefinition of the local variables. The macro that performs the unrolling can be written as

```
macro forStatic(index: untyped; a,b: static[int];
                body: untyped): untyped =
  result = newStmtList()
  for x in a..b:
    result.add newBlockStmt(body.replace(index, newLit(x)))
```

The index variable and the loop body are passed as untyped parameters, which means the macro gets the raw expressions without any attempt to determine the types of the expressions passed in. The loop bounds (`a` and `b`) are passed as static int's which means that their values are known at compile time. The `result` variable is an implicit variable available to all routines (functions, templates and macros) that return a value. The value of this variable is returned from the routine. We first initialize it with an empty statement list, then iterate over the loop bounds and add statements to it. Here we use another function `replace` (which we have written, but do not list here) to substitute all occurrences of the index variable (`index`, which in this case was passed the identifier `i`) in the loop body (`body`) with an int literal equal to the value of `x`.

PoS(LATTICE2016)271

Another example of using macros for optimization is in flattening complex data types. This optimization was developed for the XLC compiler on BG/Q since it didn't optimize temporary objects containing SIMD vectors (vector4double) as well as it did when those objects are split into individual SIMD variables. We designed a macro, `optimize`, which will take a code block like

```
optimize:
  var t: array[3, tuple[re: vector4double, im: vector4double]]
  ...
  t[0].re = ...
  t[0].im = ...
  ...
```

and turn it in to something like the following

```
var t0re: vector4double
var t0im: vector4double
...
t0re = ...
t0im = ...
...
```

Here the temporary variable `t`, which was an array of 3 complex numbers (here implemented as a tuple) of SIMD vectors, is transformed into 6 individual SIMD vectors, and the instances where the original variable was used are now modified to use the new variables. For this transformation to work, we must have already unrolled all loops containing the temporary variables. Transformations like this can give over a $2\times$ performance gain on the BG/Q.

One last example of macro transformations is for the creation of a simple DSL for dealing with tensor contractions [10]. A comprehensive tensor programming environment is being created to make contractions, such as those appearing in analysis routines, both easy to write and very efficient. A simple example of operations dealing with color vectors (`v1` and `v2`) and a color matrix (`m`) is

```
tpl:
  v2 = 0
  v2 += v1 + 0.1
  v3 += m * v2
```

This code will get transformed into something like

```
for j in 0..2:
  v2[j] = 0
  v2[j] += v1[j] + 0.1
  for k in 0..2:
    v3[k] += m[k,j] * v2[j]
```

Note that the `j` loop for `v2` only appears once. The macro expansion recognized the redundant loop and fused it across the three expressions. One can also use explicit indexing along with Einstein notation (auto-summation) like

PoS(LATTICE2016)271

```
v1[a] = p[mu,mu,a,b] * v2[b]
```

where the variables a, b and mu are implicitly summed.

2. New lattice framework in Nim: QEX (Quantum Expressions)

We have begun development of a new lattice field theory framework called QEX (Quantum Expressions) [11] using the Nim language. The framework is using the same layout and communications support code from the QLL library. This code will eventually be converted from C to Nim, but since Nim can easily work with C code, this hasn't been a priority. We are currently using the USQCD QMP [12] and QIO [13] libraries for message passing and I/O. A simple example of a QEX program is below.

```
import qex
import qcdTypes
qexInit()
var lat = [4,4,4,4]
var lo = newLayout(lat)
var v1 = lo.ColorVector()
var v2 = lo.ColorVector()
var m1 = lo.ColorMatrix()
threads: # start thread block (currently OpenMP parallel)
  m1 := 1
  v1 := 2
  v2 := m1 * v1
  shift(v1, dir=3, len=1, v2) # len=+1: from forward
  single: # execute only on single thread
    if myRank==0:
      echo v2[0][0] # vector "site" 0, color 0
qexFinalize()
```

We currently have a staggered solver (plain and Naik) and simple staggered meson analysis running in QEX, and have been working on optimizing the framework for BG/Q and KNL platforms. The performance of the staggered solver on a single node Intel Xeon Phi 7210 (KNL) CPU is shown in Figure 2. The performance of the full CG solver in GFlop/s is plotted versus the fourth root of the lattice volume for a large range of lattice sizes, using 64, 128 or 256 OpenMP threads. In all cases we can see a clear effect from the lattice size fitting in L2 cache for smaller lattice sizes. For larger volumes the code is running mostly from MCDRAM.

3. Summary

We have started developing a new high-level framework for lattice field theory calculations. For this we are using the Nim programming language to transform high-level expressions into optimized C code. Based on our exploration of the language, we find that Nim offers extremely useful set of features for developing optimized high-level frameworks for scientific computing. We have a staggered CG solver running with very good efficiency using this new framework, and are working on adding more physics capabilities along with improving optimization and exploring performance portability across CPU and GPU architectures.

PoS(LATTICE2016)271

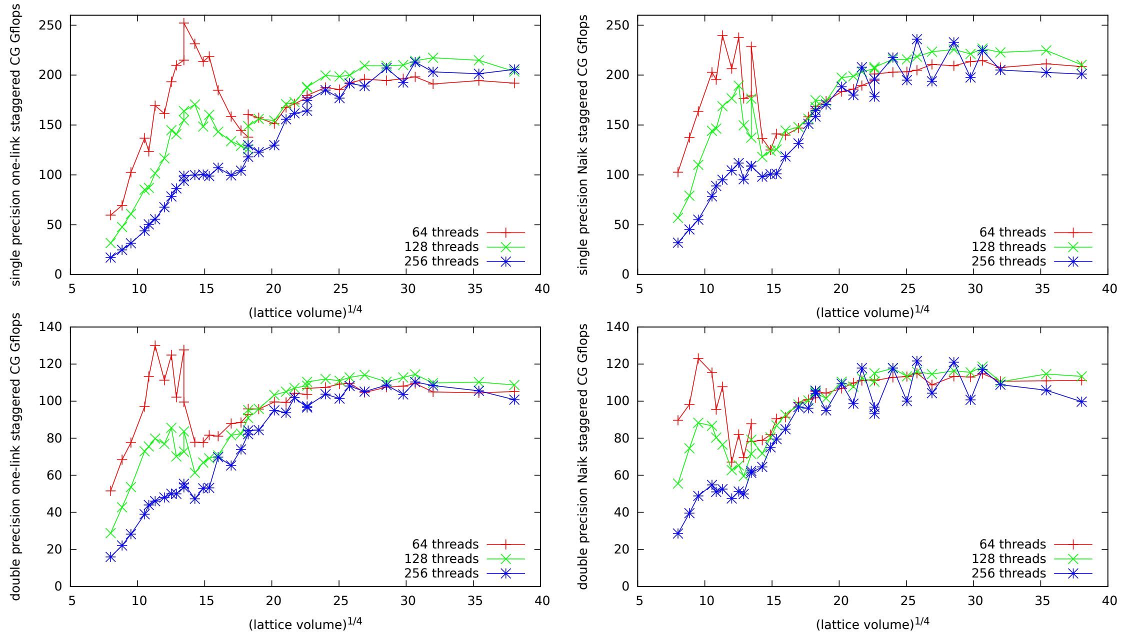

Figure 2: CG solver performance on KNL for single (top) and double (bottom) precision. Left plots are for the plain staggered (one-link) Dirac operator and right plots are Naik staggered (one-link + three-link).

Acknowledgments

This work was supported in part by and used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. X.-Y. Jin was also supported in part by the DOE SciDAC program.

References

[1] USQCD, "SciDAC QFT Data Parallel library," http://usqcd-software.github.io/qdp.

[2] R. G. Edwards *et al.* [SciDAC and LHPC and UKQCD Collaborations], Nucl. Phys. Proc. Suppl. **140**, 832 (2005) [hep-lat/0409003].

[3] J.C. Osborn, R. Babich, J. Brannick, R.C. Brower, M.A. Clark, S.D. Cohen and C. Rebbi, PoS LATTICE **2010** (2010) 037.

[4] R. Ierusalimschy, L. H. de Figueiredo and W. C. Filho, "Lua–An Extensible Extension Language," Softw: Pract. Exper., 26: 635-652 (1996).

[5] A. Pochinsky and Contributors, "Qlua—integration and optimization framework for lattice QCD," https://usqcd.lns.mit.edu/redmine/projects/qlua.

[6] J. C. Osborn, *The FUEL code project*, *PoS* **LATTICE2014** (2014) 028.

[7] J. C. Osborn, https://github.com/jcosborn/qll.

[8] A. Rumpf and Contributors, "Nim Programming Language," http://nim-lang.org.

[9] J. C. Osborn, https://github.com/jcosborn/cudanim.

[10] X.-Y. Jin and Contributors, "Tensor Programming Library in Nim for QEX," https://github.com/jxy/tpl.

[11] J. C. Osborn and Contributors, "Quantum Expressions lattice field theory framework," https://github.com/jcosborn/qex.

[12] USQCD, "QMP Message Passing Library," http://usqcd-software.github.io/qmp/.

[13] USQCD, "QIO Parallel IO Library," http://usqcd-software.github.io/qio/.

PoS(LATTICE2016)271