Introduction to the Quantum EXpressions (QEX) framework

James C. Osborn & Xiao-Yong Jin
Argonne Leadership Computing Facility

July 28
Lattice 2016
Southampton, UK
Evolution of USQCD SciDAC “C” software

- Shared base (in C): QMP, QIO
- C/C++ data parallel: QDP+QLA, QDP++
- QOPQDP: solvers, forces, etc. built on QDP
- Lua application scripting layers on QDP/QOPQDP: QLUA, FUEL
- Lua scripting provides
  - Ease of use
  - Rapid development & testing
  - Speed of C underneath

<table>
<thead>
<tr>
<th>Chroma</th>
<th>CPS</th>
<th>FUEL</th>
<th>MILC</th>
<th>QLUUA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverter</td>
<td>MDWF</td>
<td>QOPQDP</td>
<td>QUDA</td>
<td></td>
</tr>
<tr>
<td>QDP++</td>
<td>QDP</td>
<td>QIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QLA</td>
<td>QMP</td>
<td>QMT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- QLA/QDP
  - Array of structures
  - Originally no threading (now has OpenMP)
  - Needs modern update
Evolution of USQCD SciDAC C/Lua software

- Started new framework to experiment with threading and vectorization (QLL)
- Hand written + Lua generated C code
- Well tuned staggered + Naik CG gets 23% of peak on BG/Q
- Started looking for high-level language
  - Transform natural expressions into well optimized code
  - Have ability to perform optimizations across multiple expressions (i.e. loop fusion)
- Discovered (nearly*) perfect language for the job: Nim

* “not perfect yet”
Nim (nim-lang.org)

- Modern language started in 2008
- Designed to be “efficient, expressive, and elegant”
- Borrows heavily from: Modula 3, Delphi, Ada, C++, Python, Lisp, Oberon
- Statically typed, but has extensive type-inference, so feels like dynamically-typed scripting language
- Efficient garbage collection (optional)
- Extensive meta-programming support (nearly full language available at compile time)
- Still young for language
  - Current version 0.14.2
  - Strong desire to work towards 1.0 (backward stability)
  - Small, but growing community (users and developers)
Nim

- Nim compiles to C/C++ (also JS, PHP): “one level up” from C/C++
  
  \[ \text{C++} \rightarrow (\text{clang}) \rightarrow \text{IR} \rightarrow (\text{LLVM}) \rightarrow \text{asm} \rightarrow (\text{as}) \rightarrow \text{obj} \rightarrow (\text{ld}) \rightarrow \text{binary} \]

- C/C++ backend provides
  - Portability
  - Easy integration with C/C++ libraries, intrinsics (simd), pragmas (OpenMP, OpenACC), OpenCL, CUDA(?)

- Integrated build system tracks dependencies, compiles and links:
  - No Makefile necessary: copy main program, modify, compile
    
    \[
    \text{nim c myProject1.nim} \\
    \text{nim c myProject2.nim} \\
    \ldots
    \]


## Generic and meta-programming features

<table>
<thead>
<tr>
<th>C++</th>
<th>Nim</th>
</tr>
</thead>
<tbody>
<tr>
<td>preprocessor</td>
<td>templates:</td>
</tr>
<tr>
<td>macros</td>
<td>inline code substitutions</td>
</tr>
<tr>
<td></td>
<td>also allows overloading, completely hygenic (if desired)</td>
</tr>
<tr>
<td>templates</td>
<td>generics:</td>
</tr>
<tr>
<td></td>
<td>applies to type definitions, procedures, templates and macros</td>
</tr>
<tr>
<td></td>
<td>also allows typeclasses, concepts</td>
</tr>
<tr>
<td>???</td>
<td>macros:</td>
</tr>
<tr>
<td></td>
<td>similar to lisp: syntax tree of arguments passed to macro at</td>
</tr>
<tr>
<td></td>
<td>compile time to allow arbitrary manipulation</td>
</tr>
</tbody>
</table>
Simple macro example

- Transform loops at compile time
- Standard for loop:
  ```
  for i in 0..2:
    foo(i)
  ```
- macro:
  ```
  macro forStatic(index: untyped; slice: Slice[int]; body: untyped): stmt = ...
  ```
  ```
  forStatic i, 0..2:
    foo(i)
  ```
  →
  ```
  foo(0)
  foo(1)
  foo(2)
  ```
Macros for low level optimization

- optimize:
  ```
  var t: array[3, tuple[re: vector4double, im: vector4double]]
  ...
  t[0].re = ...
  t[0].im = ...
  ...
  
  →
  
  var t0re: vector4double
  var t0im: vector4double
  ...
  foo(t0re)
  foo(t0im)
  ...
  ```
Tensor operations (Xiao-Yong Jin)

- General tensor support in development:

  ```python
tensorOps:
v2 = 0
v2 += v1 + 0.1
v3 += m1 * v2

→

for j in 0..2:
v2[j] = 0
v2[j] += v1[j] + 0.1
for k in 0..2:
v3[k] += m1[k,j] * v2[j]
```

- Can also use Einstein notation (autosummation):

  ```python
v1[a] = p[μ,μ,a,b] * v2[b]
```
New lattice framework in Nim: QEX (Quantum EXpressions)

- Using layout/communications framework from QLL (will eventually convert to Nim, not urgent: Nim works great with C)
- Working example of staggered solver (plain & Naik) & simple meson analysis
- Plan to work on link smearings + HMC next
- Linear algebra undergoing reorganization
  - Optimizations and tensor support
- Once more code is running, will shift focus to improving high-level interface
- Code available on github
  https://github.com/jcosborn/qex
QEX: QCD (or Quantum) Expressions

```python
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:
    m1 := 1
    v1 := 2
    v2 := m1 * v1
    shift(v1, dir=3, len=1, v2) # len=+1: from forward
single:
    if myRank==0:
        echo v2[0][0] # vector "site" 0, color 0
qexFinalize()
```
QEX/Nim examples

- threads: implementation

```nim
template threads*(body:untyped):untyped =
let tidOld = tid
let nidOld = nid
proc tproc =
    {.emit:"
#pragma omp parallel"
    }
    block:
        setupForeignThreadGc()
        tid = ompGetThreadNum()
        nid = ompGetNumThreads()
        body
        tproc()
        tid = tidOld
        nid = nidOld
```
Benchmarks

- Single node KNL Developer Platform
- Intel Xeon Phi CPU 7210
  - 64 cores, 4 hardware threads/core
  - 16 GB high bandwidth memory
- Benchmark staggered CG (with and without Naik term)
- Volumes $L^3 \times T$
  - $L$ in $\{8, 12, 16, 24, 32\}$
  - $T$ in $\{8, 12, 16, 24, 32, 48, 64\}$
  - with 64, 128 and 256 threads
- Compiled with gcc 6.1
- Plot solver Gflops versus $(volume)^{(1/4)}$
Plain (one-link) staggered CG, single precision
Naik (one-link + three-link) staggered CG, single precision
Plain (one-link) staggered CG, double precision
Naik (one-link + three-link) staggered CG, double precision
Summary

- Nim offers extremely useful set of features
  - Extensive metaprogramming support
  - Integrated build system (modules)
  - Simple, high-level “script-like” syntax
  - Seamless integration with C/C++ code, intrinsics, pragmas, etc.

- New QEX framework written in Nim
  - Staggered CG running with good performance on x86 (BG/Q in progress)
  - Working on general optimization framework goal: performance portability across compilers & architectures
  - Find more ways to exploit metaprogramming to create easy to use input “languages” for specific operations: smearing, operator contraction, ...