PROCEEDINGS OF SCIENCE

# QCL: OpenCL meta programming for lattice QCD

**Massimo Di Pierro**
*School of Computing - DePaul University - Chicago*
*E-mail:* mdipierro@cs.depaul.edu

QCL is an experimental Pyhton library that generates and executes real time OpenCL code for lattice Quantum Field Theory computations. QCL allows the user to describe arbitrary gauge and fermionic actions in terms of path shapes (links, plauqettes, staples, etc.) and generates optimized code to perform matrix multiplications along the paths. The generated code is then utilized by the algorithms, including the heathbath and the $\not{D}$. It can generate code for any improved action that can be described as a sum of paths. QCL works for arbitrary $SU(N_c)$ gauge groups and number of dimensions. It includes the BLAS/LaPack libraries and plotting capabilities.

*31st International Symposium on Lattice Field Theory - LATTICE 2013*
*July 29 - August 3, 2013*
*Mainz, Germany*

 

PoS(LATTICE 2013)043

## 1. Introduction

In the last 10 years GPU programming has become increasingly important for High Performance Computing and Lattice QCD in particular, yet most of the focus has been on implementations of SU(3) gauge field theories in 4 dimensions and highly optimized CUDA kernels for specific QCD algorithms [1, 2, 3, 4, 5, 6]. Cardoso *et al.* [7] have tackled the problem for arbitrary $SU(N_n)$ but only in 4D and only for the Wilson Gauge Action. In this paper we present a different and more general approach.

QCL is a Python library that uses meta-programming techniques to generate and run OpenCL [8] code. QCL allows the developer to program using the Python language and exposes an Application Program Interface (API) to describe actions and algorithms. The algortihms, however, are not yet implemented but instead generated in real time based on the description of the action. Actions are described by a gauge group, a set of path shapes (closed paths for gauge terms, open paths for smearing operators and fermion terms), and their symmetries. The OpenCL code for the algorithms is then generated as needed in real time. QCL can generate code for improved actions as well, as long as they can be described as sums of paths. Some of the supported algorithms include the heathbath, the $\not{D}$, the Minimum Residual (MinRes), and the Stabilized Bi-Conjugate Inverter (BiCGStab). The heathbath supports arbitrary $SU(N_c)$ gauge groups and number of dimensions. $\not{D}$ also supports arbitrary $SU(N_c)$ gauge groups, Wilson and Staggered types of fermions.

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms including CPUs, GPUs, DSPs, and virtual machines (LLVM). Like CUDA, OpenCL implements a form of task-based parallelism and includes a programming language derived from C99 for writing computing kernels. One kernel and a unit of data constitutes a task. OpenCL programs consist of two parts: a host program and a collection of kernels. The host program loads data from RAM, assigns data to kernels, and queues the tasks for execution on the available devices. OpenCL includes a just-in-time compiler which allows the host program to generate and compile kernels at run-time. Unlike CUDA, which is mostly an initiative of a single vendor, OpenCL is an open standard maintained by the non-profit technology consortium Kronos Group, adopted and supported by Nvidia, AMD, Intel, ARM, Apple, Qualcomm, and others. Since kernels are written in OpenCL, QCL code presents no overhead compared to native OpenCL code.

In QCL the host program is written in Python using the pyOpenCL library [9], the `numpy` library (which provides efficient arrays and BLAS/LaPack algorithms), and the matplotlib library (for plotting).

## 2. Architecture Overview

QCL comprises of the following core classes (Fig 1):

- `Lattice`, which stores information about lattice size and topology.
- `Field`, which can store arbitrary data at the lattice sites.
- `ComplexScalarField`, which stores Complex numbers at lattice sites.
- `GaugeField`, a $SU(N_c)$ gauge field in $d$-dimensions.

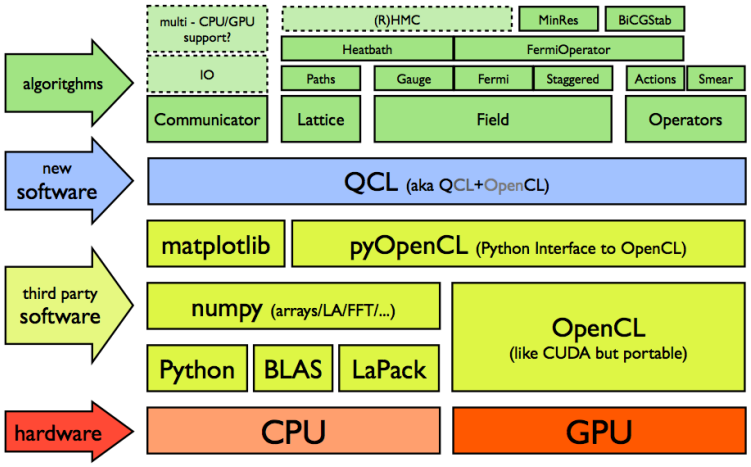


**Figure 1:** Architectural overview of QCL.

- `WilsonField`, which stores $N_c$ spinors at each lattice site.
- `StaggeredField`, which stores $N_c$ Complex numbers at each lattice site.
- `GaugeAction`, a collection of closed path shapes and their relative coefficients.
- `GaugeSmearOperator`, a collection of open path shapes, and their relative smearing coefficients.
- `FermiOperator`, a collection of open path shapes, and their relative coefficients and spin structure. A $\not{D}$ operator is a `FermiOperator`.

All QCL programs start by importing from the `qcl` module and defining a lattice instance. Note that one can define multiple lattices in the same program. Here is an example of a $6^4$ lattice:

```
1 from qcl import *
2 lattice = Q.Lattice([6,6,6,6])
```

Given a lattice one can define multiple fields on each lattice. Here is a $SU(2)$ gauge field:

```
1 U = lattice.GaugeField(2)
```

Fields live in RAM until an algorithm copies them onto a device (for example, a GPU). Gauge fields can be initialized using `set_cold`, `set_hot` (implemented in OpenCL) and can be accessed element by element:

```
1 U.set_cold()
2 print U[(0,0,0,0),0]
3 print U[(0,0,0,0),0,0,0]
```

While in RAM, fields are stored as `numpy` arrays.

In QCL we distinguish between a path (which has a shape and an origin) and a path shape (which lacks an origin and may be translated and symmetrized). Path shapes on the lattice are represented by Python tuples. Sets of paths are represented as lists of tuples. The `bc_symmetrize`

method can perform the hyper-octahedral symmetrization ($BC_n$) of any set of paths. The `derive_paths` method removes a link from a set of path shapes thus obtaining the corresponding staples:

```
plaquette = (1,2,-1,-2)
print bc_symmetrize(plaquette)
print remove_duplicates(derive_paths(bc_symmetrize(plaquette),2))
```

In the example below we define a scalar field $\rho$ and set it equal to the trace of the link product along a given path, for each lattice site:

```
rho = lattice.Field([1])
rho.set_link_product(U,[(1,2,3,4,-1,-2,-3,-4)])
print rho*rho
```

A gauge action can be constructed by adding terms (in the example below a single plaquette), which are automatically symmetrized. The `heathbath` of the `action` object generates OpenCL code. The `run` method runs the code. Here is an example for the simple Wilson gauge action:

```
action = GaugeAction(U).add_term((1,2,-1,-2))
code = action.heatbath(beta=4.0,n_iter=100)
code.run()
print U.average_plaquette()
```

The `add_term` method can be chained to construct more complex gauge actions or, similarly, `GaugeSmearOperator`s. QCL also understands named smearing algorithms including *fat3*, *fat5*, *fat7*, *lepage*, and *hisq*:

```
V = clone(U)
S = GaugeSmearOperator(U).add_term([1],1.0).add_term([2,1,-2],0.1)
V.set_smeared(S,reunitarize=4)
V.set_fat(U,'fat7+lepage')
V.set_hisq(U)
```

The simplest type of fermionic field is the FermiField which stores $N_{spin} \times N_c$ complex numbers per lattice site. In the example below we define one with 4 spin components, clone it, and set one of the components to 1:

```
phi = lattice.FermiField(4,U.nc)
psi = clone(phi)
phi[(0,0,3,3),0,0] = 1.0
```

Fermionic operators are built similarly to gauge actions using the `add_term` method but with an additional parameter, the Gamma structure of the term. QCL also understands named terms such as `add_clover4d_terms`:

```
kappa = 0.112
D = FermiOperator(U).add_diagonal_term(1.0)
for mu in (1,2,3,4):
    D.add_term(kappa*(I-G[mu]), [(mu,)])
    D.add_term(kappa*(I+G[mu]), [(-mu,)])
D.add_clover4d_terms(c_SW = 0.1)
```

Notice that `G[mu]` is a Gamma matrix and `kappa*(I-G[mu])` is computed only once at the Python level. `D.add_term(...)` then generates the OpenCL code and only considers non-zero terms. It does not generate code for terms that have a known zero coefficient. This is pictorially represented in Fig 2.

Here are examples of multiplication by $\not{D}$ and by its inverse:

$$\psi = \not{D}\phi \qquad \text{and} \qquad \phi = \not{D}^{-1}\psi \tag{2.1}$$

```
psi.set(D,phi)
phi.set(invert_minimum_residue,D,psi)
```

In the example below we make a meson propagator by looping over the 4 spin components and the `U.nc` color components for the source. For each source we use the `invert_minimum_residue` method to compute the sink and the `make_meson` operator to contract them. Notice that the first and last arguments of `make_meson` are fermions, while the second argument is an arbitrary spin structure:

```
prop = lattice.ComplexScalarField()
for spin in range(4):
    for color in range(U.nc):
        psi = lattice.FermiField(4,U.nc)
        psi[(0,0,0,0),spin,color] = 1.0
        phi.set(invert_minimum_residue,D,psi)
        prop += make_meson(phi,I,phi)
prop_fft = prop.fft()
prop_t = [(t,math.log(prop_fft[t,0,0,0].real)) for t in range(lattice.shape[0])]
Canvas().plot(prop_t).save('meson.prop.png')
```

QCL includes operators to contract abritrary mesons and baryons. In the above example we also used the `numpy fft` function to compute the propagator in momentum space, projected out the zero momentum component, and made a plot using the `Canvas` library, which is built into QCL.

Below we use `Canvas` to make a 2D slice of the (0,0) component of a `psi` field:

```
chi = psi.slice((0,0),(0,0))
Canvas().imshow(chi).save('fermi.propagator.png')
```

QCL also supports Staggered fields:

```
phi = lattice.StaggeredField(U.nc)
psi = clone(phi)
phi[(0,0,3,3),0] = 1.0
```

and they can be multiplied by `FermiOperator`s using the same syntax as the Wilson-like fields:

```
kappa = 0.112
D = FermiOperator(U).add_staggered_action(kappa = 0.112)
psi.set(D,phi)
psi.set(invert_minimum_residue,D,phi)
```

## 3. Limitations

At this time QCL has the following limitations, which we are working to overcome.

First, only one node and one device can be supported at a time. This means QCL cannot take advantage of multiple GPUs because it cannot split a lattice over multiple devices. The generated OpenCL is flexible enough to be able to access and loop over a subset of the lattice, but QCL lacks the code to synchronize memory buffers across devices. We are working to integrate QCL with MPI to overcome this limitation.

Second, QCL currently performs many unnecessary transfers between the host and the device. Most linear time operations, including scalar products, are performed on the host using `numpy`, while operations involving products of color matrices are performed on the device. This means that each application of $\not{D}$ involves transfers of data. This is unnecessary and can be eliminated by implementing all scalar products in OpenCL.

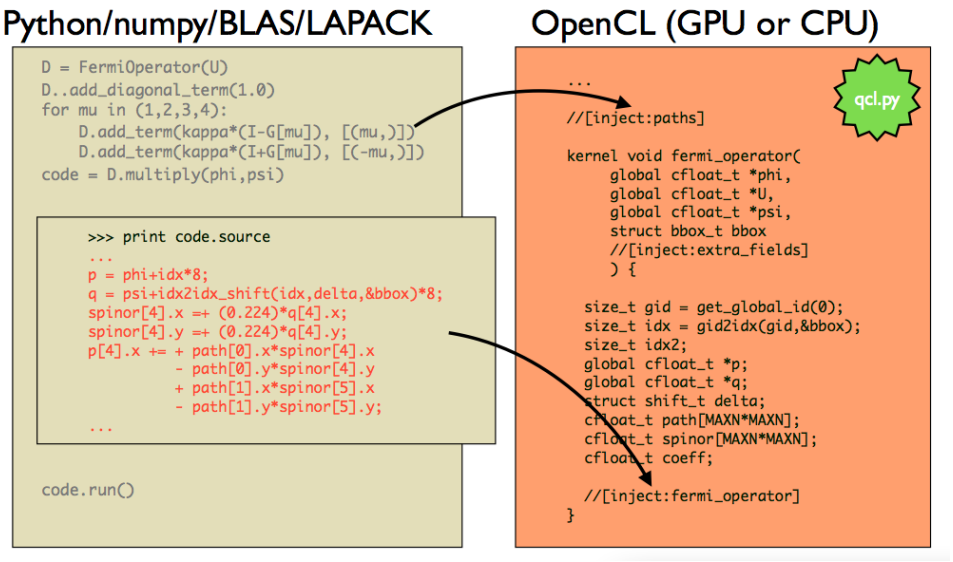


**Figure 2:** Example of code generation in QCL. The left pane shows Python code. The right pane shows an OpenCL template. The comments are replaced by generated code at runtime. The code is then compiled on demand.

QCL also lacks converters to and from existing file formats. In a future version of QCL we plan to support the FermiQCD, ILDG, and MILC formats.

QCL's fourth limitation is in its optimizations. Currently QCL performs the following optimizations when generating OpenCL code: 1) It groups together all paths that have the same spin structure. 2) It only generates code for those color/spin components that have a non-zero coefficient. 3) If two paths start with the same product of links, the product of links is cached and reused. Despite these optimizations, QCL is not smart enough to realize that, for example, a 5 staple includes a 3 staple and should therefore store the result for all 3 staples before computing 5 staples. The complexity here is twofold: determining all possible overlaps of paths is a problem of exponential complexity, and reusing sub-paths starting at a different origin requires additional memory and therefore a tradeoff of CPU usage vs memory. It is well known that GPUs are memory bound, so it is usually preferable to duplicate computation rather than cache more vectors. We plan to study this further.

Finally, QCL supports single precision only. We plan to support double and mixed precions in the future.

## 4. Conclusions and Outlook

In this paper we present an experimental approach to OpenCL metaprogramming for QFT and QCD. In our library QCL, we describe actions in terms of path shapes (links, plaquettes, staples, long staples, etc.) for arbirtrary $SU(N_x)$ gauge groups in D-dimensions. QCL then writes the corresponding OpenCL code. Thus far our main goal was correctness. We have tested our code against FermiQCD, and we believe it is working correctly. Some optimizations are implemented,

but several bottlenecks remain in the system, specifically for the fermionic sector. At this time we are working to eliminate them.

The code is hosted in GitHub [10].

## References


[1] Nvidia, C. U. D. A. "Compute unified device architecture programming guide." (2007).

[2] Babich, Ronald, Michael A. Clark, and Balint Jo. "Parallelizing the QUDA library for multi-GPU calculations in lattice quantum chromodynamics." High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for. IEEE, 2010.

[3] Bach, Matthias, et al. "Lattice QCD based on OpenCL." Computer Physics Communications (2013).

[4] Philipsen, Owe, et al. "Lattice QCD using OpenCL." XXIX International Symposium on Lattice Field Theory. 2011.

[5] Demchik, Vadim, and Natalia Kolomoyets. "QCDGPU: open-source package for Monte Carlo lattice simulations on OpenCL-compatible multi-GPU systems." arXiv preprint arXiv:1310.7087 (2013).

[6] Bonati, Claudio, et al. "QCD simulations with staggered fermions on GPUs." Computer Physics Communications 183.4 (2012): 853-863.

[7] Cardoso, Nuno, and Pedro Bicudo. "Generating SU (Nc) pure gauge lattice QCD configurations on GPUs with CUDA." Computer Physics Communications (2012).

[8] Munshi, Aaftab. "The opencl specification." Khronos OpenCL Working Group 1 (2009): 11-15.

[9] Klöckner, Andreas, et al. "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation." Parallel Computing 38.3 (2012): 157-174.

[10] `https://github.com/mdipierro/qcl/`


PoS(LATTICE 2013)043