http://github.com/mdipierro/QCL

by Massimo Di Pierro

DePaul University

What is it?

- You program in Python (2.7)
- You describe Actions/Operators as paths
- It generates OpenCL code
- It runs the code on a GPU or CPU
- Process with numpy (BLAS/LaPack) & matplotlib

Idea

- Any Action is a list of paths
- Paths are ordered products of links
- Closed/traced paths are pure gauge
- Open paths represent fermi-gauge interactions
- All fermionic operators can be written as paths
- Inverters are (mostly) action-agnostic

Example: Lattice

Example: Lattice

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

```
>>> U = lattice.GaugeField(2)
>>> U.set_cold()
```

```
>>> print U[(0,0,0,0),0]
[[ 1.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j]]
>>> print U[(0,0,0,0),0,0,0]
1.+0.j
```

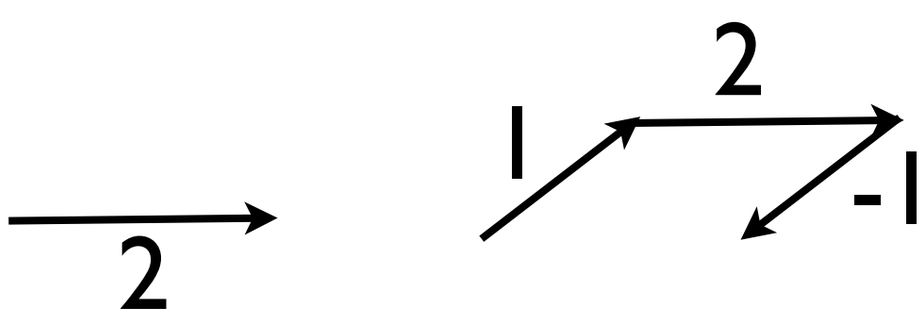
Example: Paths

Example: Paths

```
>>> plaquette = (1,2,-1,-2)
```

```
>>> print bc_symmetrize(plaquette)
[(2, 1, -2, -1), (3, 1, -3, -1), (4, 1, -4, -1), (-2, 1, 2, -1), (-3, 1, 3, -1), (-4, 1, 4, -1), (1, 2,
-1, -2), (3, 2, -3, -2), (4, 2, -4, -2), (-1, 2, 1, -2), (-3, 2, 3, -2), (-4, 2, 4, -2), (1, 3, -1, -3),
(2, 3, -2, -3), (4, 3, -4, -3), (-1, 3, 1, -3), (-2, 3, 2, -3), (-4, 3, 4, -3), (1, 4, -1, -4), (2, 4,
-2, -4), (3, 4, -3, -4), (-1, 4, 1, -4), (-2, 4, 2, -4), (-3, 4, 3, -4), (2, -1, -2, 1), (3, -1, -3, 1),
(4, -1, -4, 1), (-2, -1, 2, 1), (-3, -1, 3, 1), (-4, -1, 4, 1), (1, -2, -1, 2), (3, -2, -3, 2), (4, -2,
-4, 2), (-1, -2, 1, 2), (-3, -2, 3, 2), (-4, -2, 4, 2), (1, -3, -1, 3), (2, -3, -2, 3), (4, -3, -4, 3),
(-1, -3, 1, 3), (-2, -3, 2, 3), (-4, -3, 4, 3), (1, -4, -1, 4), (2, -4, -2, 4), (3, -4, -3, 4), (-1, -4,
1, 4), (-2, -4, 2, 4), (-3, -4, 3, 4)]
```

```
>>> print remove_duplicates(derive_paths(bc_symmetrize(plaquette),2))
[(3, -2, -3), (1, -2, -1), (-3, -2, 3), (-1, -2, 1), (4, -2, -4), (-4, -2, 4)]
```



Example: Links

Example: Links

```
>>> rho = lattice.Field([1])
```

```
>>> rho.set_link_product(U,[(1,2,3,4,-1,-2,-3,-4)])
```

Example: Links

```
>>> rho = lattice.Field([1])
```

```
>>> rho.set_link_product(U,[(1,2,3,4,-1,-2,-3,-4)])
```

writes

```
...
// load U((0, 1, 0, 0), (2,)) -> m01
shift.s[0] = 0;
shift.s[1] = 1;
shift.s[2] = 0;
shift.s[3] = 0;
ixmu = (idx2idx_shift(idx0,shift,bbox)*4+2)*4;
m01_0x0 = U[ixmu+0];
m01_0x1 = U[ixmu+1];
m01_1x0 = U[ixmu+2];
m01_1x1 = U[ixmu+3];
// compute m00*m01 -> m02
m02_0x0.x =
+m00_0x0.x*m01_0x0.x-m00_0x0.y*m01_0x0.y
+m00_0x1.x*m01_1x0.x-m00_0x1.y*m01_1x0.y;
m02_0x0.y =
+m00_0x0.x*m01_0x0.y+m00_0x0.y*m01_0x0.x
+m00_0x1.x*m01_1x0.y+m00_0x1.y*m01_1x0.x;
...
```

```
>>> print rho*rho
(5184+0j)
```

Example: Gauge Action

Example: Gauge Action

```
>>> action = GaugeAction(U).add_term((1,2,-1,-2))
>>> code = action.heatbath(beta=4.0)
>>> code.run()
```

```
>>> print U.average_plaquette()
(0.829631726928-1.71014946146e-10j)
```

Anisotropic Gauge

$$S_{II}[U] = \beta \left\{ \frac{5}{3\xi u_s^4} W_{\rm sp} + \frac{4\xi}{3u_s^2 u_t^2} W_{\rm tp} - \frac{1}{12\xi u_s^6} W_{\rm sr} - \frac{\xi}{12\xi^2 u_s^4 u_t^2} W_{\rm str} \right\}.$$

$$\begin{aligned}
W_{\rm sp} &= \frac{1}{3} \sum_x \sum_{i \neq j} \Re e \, {\rm Tr} \left[1 - U_i(x) U_j(x+\hat{i}) U_i^\dagger(x+\hat{j}) U_j^\dagger(x) \right], \\
W_{\rm tp} &= \frac{1}{3} \sum_x \sum_i \Re e \, {\rm Tr} \left[1 - U_t(x) U_i(x+\hat{t}) U_t^\dagger(x+\hat{i}) U_i^\dagger(x) \right], \\
W_{\rm sr} &= \frac{1}{3} \sum_x \sum_{i \neq j} \Re e \, {\rm Tr} \left[1 - U_i(x) U_i(x+\hat{i}) U_j(x+2\hat{i}) U_i^\dagger(x+\hat{i}+\hat{j}) U_i^\dagger(x+\hat{j}) U_j^\dagger(x) \right] \\
W_{\rm str} &= \frac{1}{3} \sum_x \sum_i \Re e \, {\rm Tr} \left[1 - U_i(x) U_i(x+\hat{i}) U_t(x+2\hat{i}) U_i^\dagger(x+\hat{i}+\hat{t}) U_i^\dagger(x+\hat{t}) U_t^\dagger(x) \right].
\end{aligned}$$

Anisotropic Gauge

$$S_{II}[U] = \beta \left\{ \frac{5}{3\xi u_s^4} W_{\rm sp} + \frac{4\xi}{3u_s^2 u_t^2} W_{\rm tp} - \frac{1}{12\xi u_s^6} W_{\rm sr} - \frac{\xi}{12\xi^2 u_s^4 u_t^2} W_{\rm str} \right\}.$$

$$\begin{aligned}
W_{\rm sp} &= \frac{1}{3} \sum_x \sum_{i \neq j} \Re e \, {\rm Tr} \left[1 - U_i(x) U_j(x+\hat{i}) U_i^\dagger(x+\hat{j}) U_j^\dagger(x) \right], \\
W_{\rm tp} &= \frac{1}{3} \sum_x \sum_{i} \Re e \, {\rm Tr} \left[1 - U_t(x) U_i(x+\hat{t}) U_t^\dagger(x+\hat{i}) U_i^\dagger(x) \right], \\
W_{\rm sr} &= \frac{1}{3} \sum_x \sum_{i \neq j} \Re e \, {\rm Tr} \left[1 - U_i(x) U_i(x+\hat{i}) U_j(x+2\hat{i}) U_i^\dagger(x+\hat{i}+\hat{j}) U_i^\dagger(x+\hat{j}) U_j^\dagger(x) \right] \\
W_{\rm str} &= \frac{1}{3} \sum_x \sum_{i} \Re e \, {\rm Tr} \left[1 - U_i(x) U_i(x+\hat{i}) U_t(x+2\hat{i}) U_i^\dagger(x+\hat{i}+\hat{t}) U_i^\dagger(x+\hat{t}) U_t^\dagger(x) \right].
\end{aligned}$$

```
action = GaugeAction(U)
action.add_term([(i,j,-i,-j) for i in (1,2,3) for j in (1,2,3) if i!=j],        ...) # Wsp
action.add_term([(i,4,-i,-4) for i in (1,2,3)],                                 ...) # Wtp
action.add_term([(i,i,j,-i,-i,-j) for i in (1,2,3) for j in (1,2,3) if i!=j],   ...) # Wsr
action.add_term([(i,i,4,-i,-i,-4) for i in s (1,2,3),                           ...) # Wstr
action.heatbath(beta=...)
```

Writes

```
...
// OpenCL Code
...
```

Example: Fermions

Example: Fermions

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

Example: Fermions

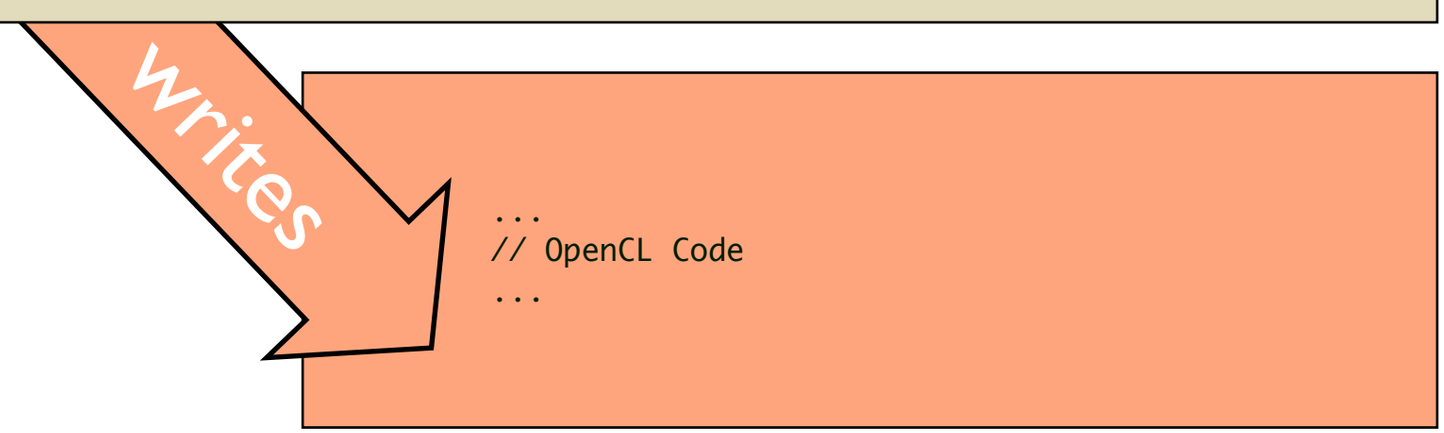
```
>>> phi = lattice.FermiField(4,U.nc)
>>> psi = clone(phi)
>>> phi[(0,0,3,3),0,0] = 1.0
```

```
>>> kappa = 0.112
>>> D = FermiOperator(U)
>>> D.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,)])
```

Example: Fermions

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

```
>>> kappa = 0.112
>>> D = FermiOperator(U)
>>> D.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,)])
```

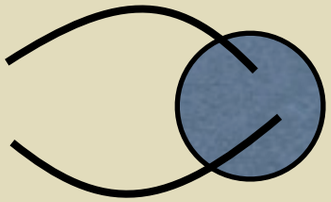



```
>>> psi.set(D,phi)
>>> psi.set(invert_minimum_residue,D,phi)
```

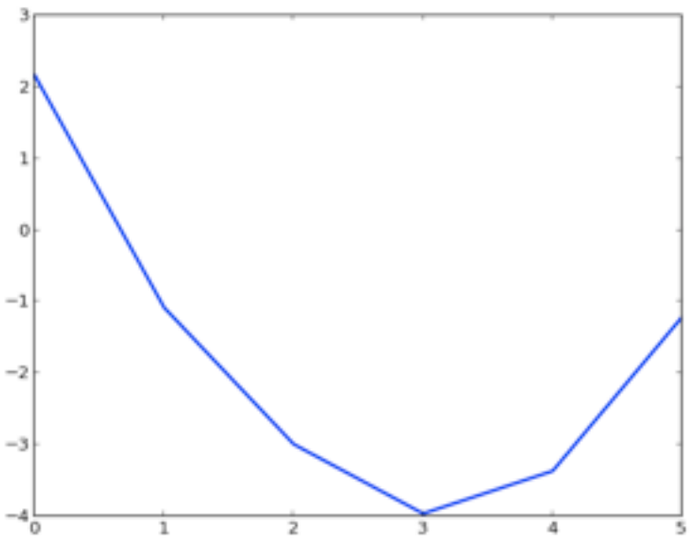
Example: Mesons

Example: Mesons

```
>>> 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)
```



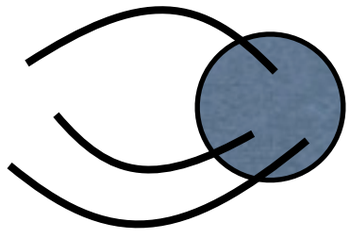
```
>>> prop_fft = prop.fft()
>>> prop_t = [(t,math.log(prop_fft[t,0,0,0].real))
              for t in range(lattice.shape[0])]
>>>> Canvas().plot(prop_t).save('meson.prop.png')
```



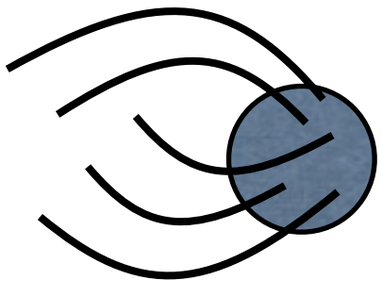
Example: Baryons

Example: Baryons

```
>>> for ...:
        prop += make_baryon3(quark1,quark2,gamma,quark3)
```



```
>>> for ...:
        prop += make_hadron(contractions, (quark1,quark2,quark3,...))
```



Example: Staggered

Example: Staggered

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

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

HISQ Smearing

HISQ Smearing

```
U = lattice.GaugeField(nc=5)
V = clone(U)
V.set_hisq(U)
```

HISQ Smearing

```
U = lattice.GaugeField(nc=5)
V = clone(U)
V.set_hisq(U)
```

```
W = clone(U)
W.set_fat(U,'fat5',reunitarize=10)
V = clone(U)
V.set_fat(W,'fat7+lepage')
```

qcl.py

HISQ Smearing

```
U = lattice.GaugeField(nc=5)
V = clone(U)
V.set_hisq(U)
```

```
W = clone(U)
W.set_fat(U,'fat5',reunitarize=10)
V = clone(U)
V.set_fat(W,'fat7+lepage')
```

qcl.py

```
def set_fat(self, U, name):
    S = GaugeSmearOperator(U)
    if name == 'fat7+lepage': c = [...]
    S.add_term([1],c[0])
    S.add_term([2,1,-2],c[1])
    S.add_term([3,2,1,-2,3],c[2])
    S.add_term([4,3,2,1,-2,-3,-4],c[3])
    S.add_term([2,2,1,-2,-2],c[4])
    self.set_smeared(S)
```

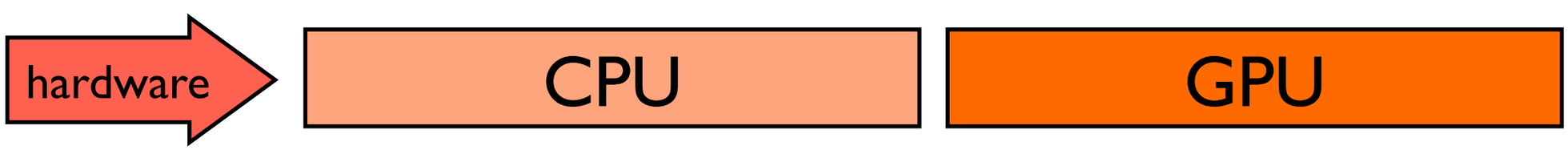
qcl.py

Supported Algorithms

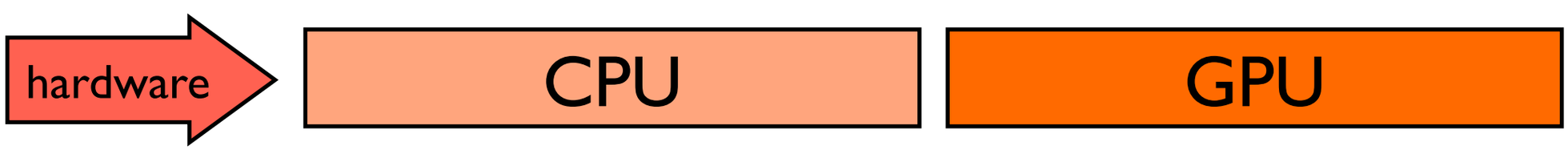
- Any number of dimensions
- Any SU(N)
- Heatbath: all actions written in terms of loops
- Dslash/Operators: all written in terms of spin matrices and paths
- Examples: naive, Wilson, Clover, Staggered, Asqtad, Fermilab, HISQ
- Isotropic and un-isotropic
- Inverters: MinRes & BiCGStab for any operator
- Smearing for gauge and fermions

Architecture

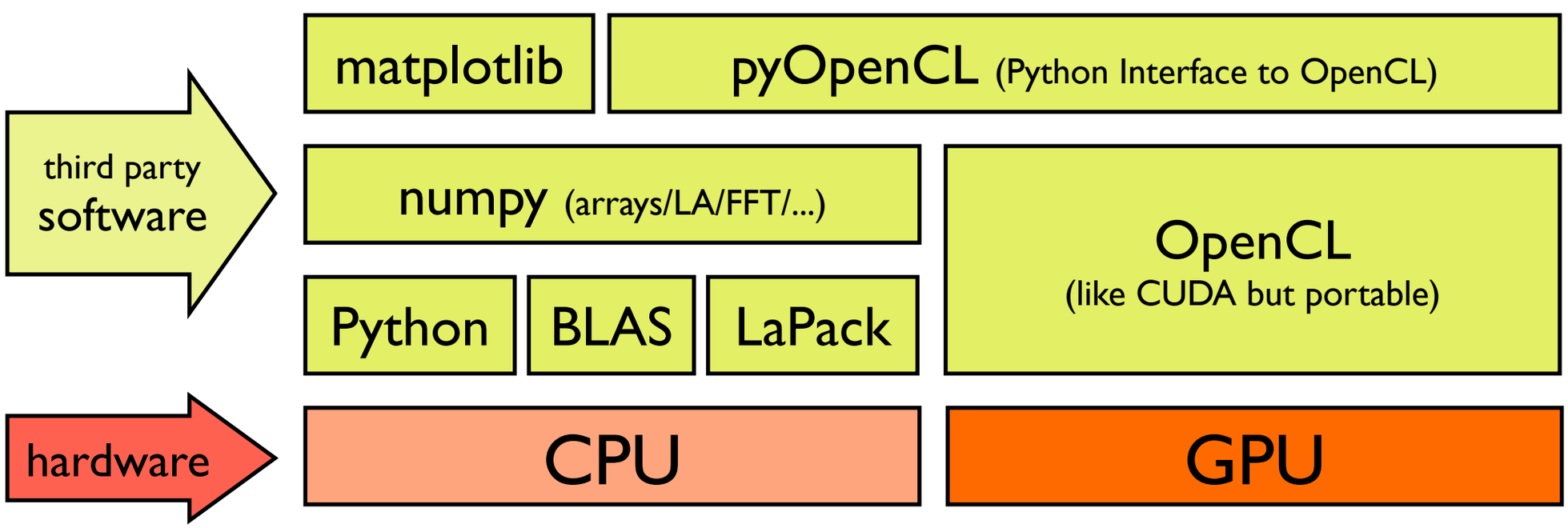
Architecture



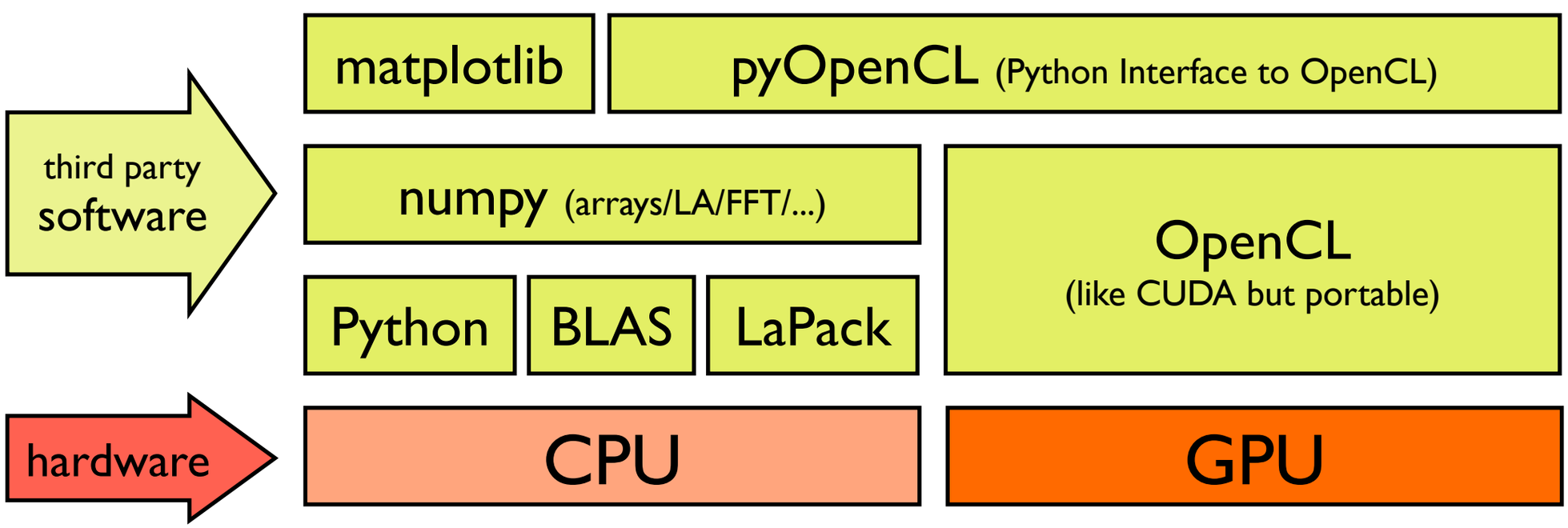
Architecture



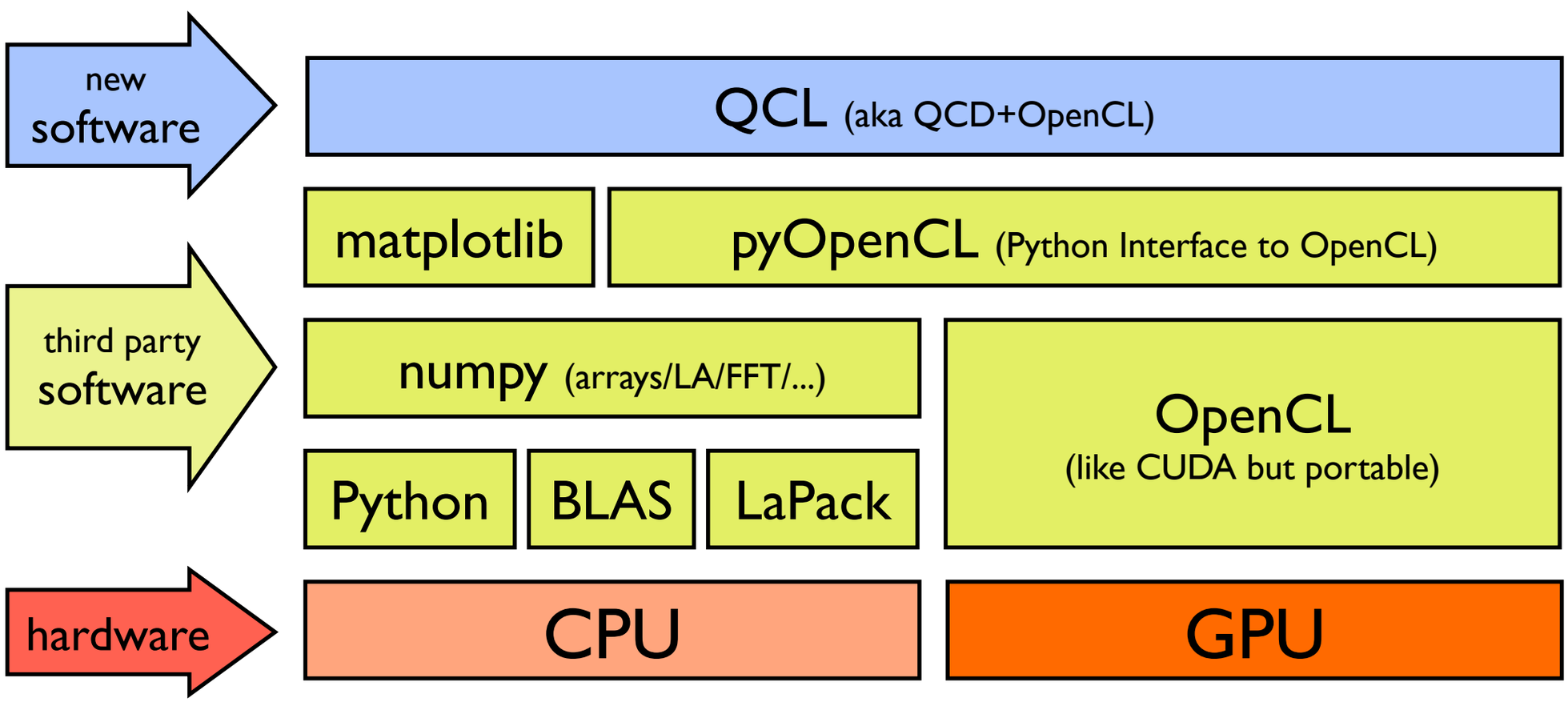
Architecture



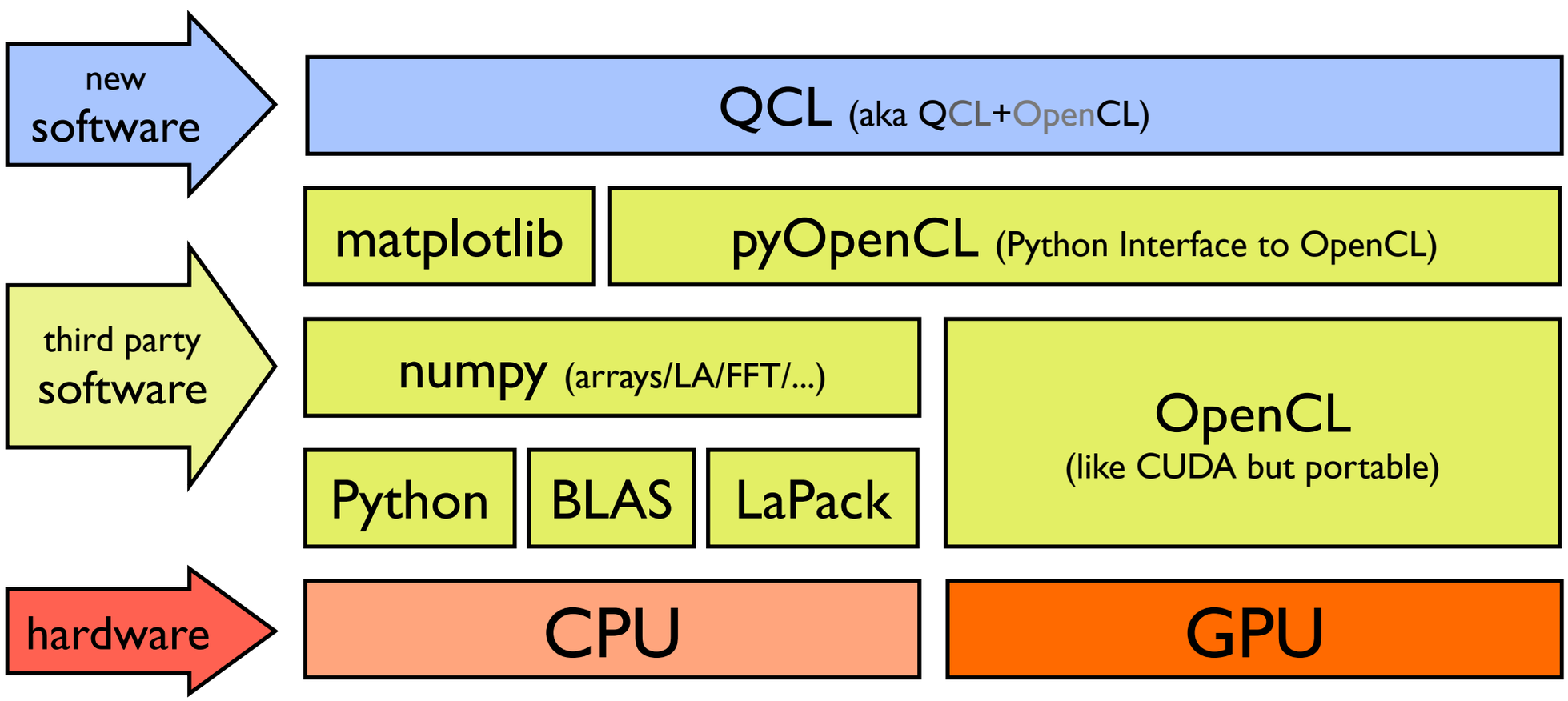
Architecture



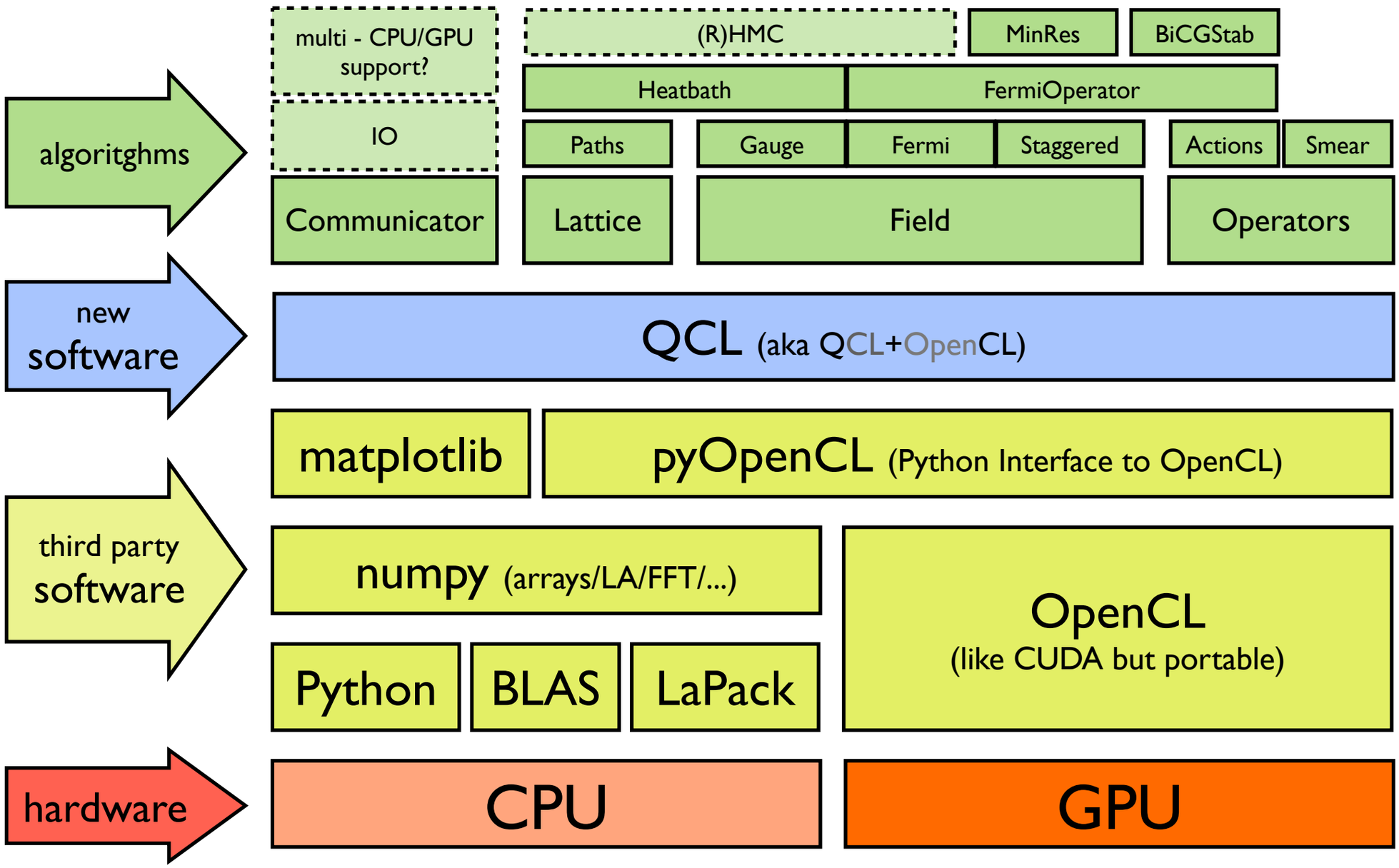
Architecture



Architecture



Architecture



How does it work?

Python/numpy/BLAS/LAPACK

OpenCL (GPU or CPU)

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D.add_wilson4d_terms(kappa=0.11)
phi.set(D,psi)
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D.add_wilson4d_terms(kappa=0.11)
phi.set(D,psi)
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D.add_wilson4d_terms(kappa=0.11)
phi.set(D,psi)
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D.add_wilson4d_terms(kappa=0.11)
phi.set(D,psi)
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D.add_wilson4d_terms(kappa=0.11)
phi.set(D,psi)
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D..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,)])
code = D.multiply(phi,psi)
code.run()
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D..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,)])
code = D.multiply(phi,psi)
code.run()
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D..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,)])
code = D.multiply(phi,psi)
code.run()
```

```
>>> print I-G[mu]
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  2.  0.]
 [ 0.  0.  0.  2.]]
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D..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,)])
code = D.multiply(phi,psi)



code.run()
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D..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,)])
code = D.multiply(phi,psi)
```

```
>>> print code.source
...
p = phi+idx*8;
q = psi+idx2idx_shift(idx,delta,&bbox)*8;
spinor[4].x =+ (0.224)*q[4].x;
spinor[4].y =+ (0.224)*q[4].y;
p[4].x += + path[0].x*spinor[4].x
          - path[0].y*spinor[4].y
          + path[1].x*spinor[5].x
          - path[1].y*spinor[5].y;
...
```

```
code.run()
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D..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,)])
code = D.multiply(phi,psi)
code.run()
```

OpenCL (GPU or CPU)

qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

How does it work?

Python/numpy/BLAS/LAPACK

```
D = FermiOperator(U)
D..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,)])
code = D.multiply(phi,psi)
code.run()
```

OpenCL (GPU or CPU)

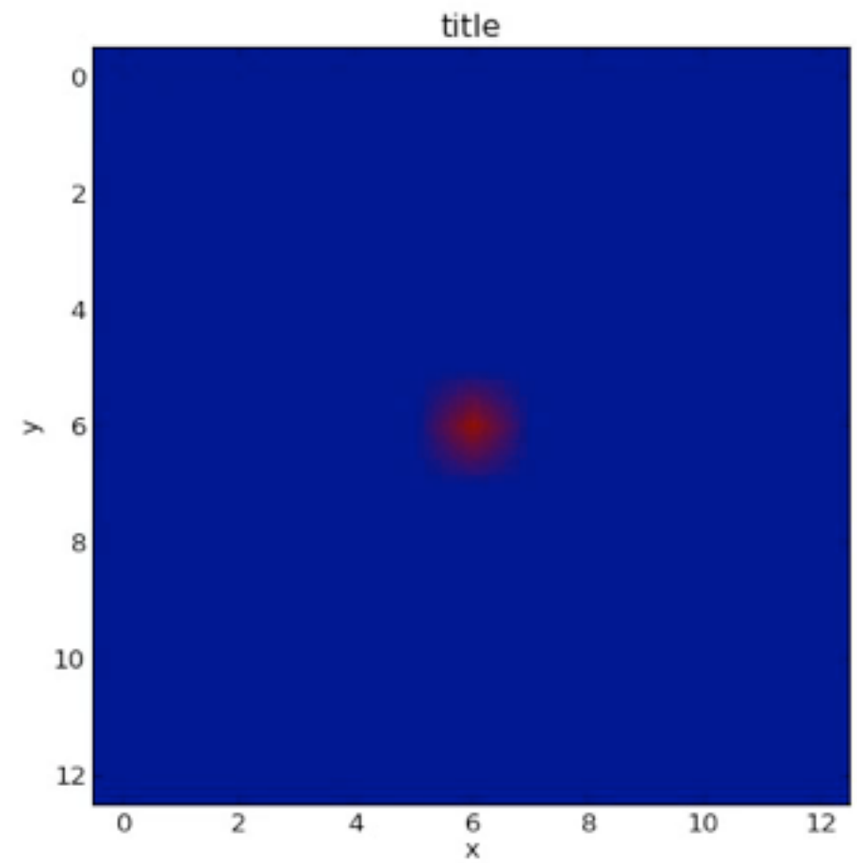
qcl.py

```
...

//[inject:paths]

kernel void fermi_operator(
     global cfloat_t *phi,
     global cfloat_t *U,
     global cfloat_t *psi,
     struct bbox_t bbox
     //[inject:extra_fields]
     ) {

  size_t gid = get_global_id(0);
  size_t idx = gid2idx(gid,&bbox);
  size_t idx2;
  global cfloat_t *p;
  global cfloat_t *q;
  struct shift_t delta;
  cfloat_t path[MAXN*MAXN];
  cfloat_t spinor[MAXN*MAXN];
  cfloat_t coeff;

  //[inject:fermi_operator]
}
```

Smearing and Plotting

Complete Code

```
from qcl import *
lattice = Q.Lattice([13,13,13,13])
U = lattice.GaugeField(2)
U.set_cold()
phi = lattice.FermiField(4,U.nc)
psi = clone(phi)
phi[(0,0,6,6),0,0] = 1.0
S = FermiOperator(U).add_diagonal_term(1.0)
for mu in (1,2,3,4):
    S.add_term(0.1*I,[(-mu,)]).add_term(0.1*I,[(mu,)])
for k in range(100):
    psi.set(S,phi)
    phi,psi = psi,phi
    chi = psi.slice((0,0),(0,0))
    Canvas().imshow(chi).save('smear.%.2i.png' % k)
```



Inverter Implementation

qcl.py

```
def invert_minimum_residue(y,f,x,ap=1e-4,rp=1e-4,ns=1000):
    q = clone(x)
    r = clone(x)
    copy_elements(y, x)
    copy_elements(r, x)
    f(q,x)
    r -= q
    for k in xrange(ns):
        f(q,r)
        alpha = vdot(q,r)/vdot(q,q)
        y += alpha*r
        r -= alpha*q
        residue = math.sqrt(vdot(r,r).real/r.size)
        norm = math.sqrt(vdot(y,y).real)
        if k>10 and residue<max(ap,norm*rp):
            return y
    raise ArithmeticError('no convergence')
```

OpenCL vs Cuda

	Cuda	OpenCL
Vendor	Nvidia	Nvidia, Intel, AMD, ARM
Target	Nvidia GPU, x86, LLVM	GPUs, CPUs, LLVM
Syntax	C(++)	C99
Runtime-Compiler	No	Yes
Speed	Andreas Kloeckner: "If you're addressing the same hardware, both frameworks should be able to achieve the same speeds"	

Todo

- Tests (there are built-in tests but need more)
- Benchmarks
- Add more algorithms
- Add double precision support
- Reduce Ram - GPU Input/Output (major problem)
- Move some logic from numpy to Opencl?
- Support for Multi-CPU & Multi-GPU (MPI?)

List of Classes

```
# matrices: Gamma, Lambda, I, etc.

class Communicator(object): ...
class Lattice(object): ...
class Site(object): ...
class Field(object): ...
class ComplexScalarField(Field): ...
class GaugeField(Field): ...
class FermiField(Field): ...
class StaggeredField(Field): ...
class GaugeAction(object): ...
class GaugeSmearOperator(GaugeAction): ...
class FermiOperator(object): ..

def make_hadron
def make_meson
def make_baryon3
```