An Embedded Language for Vector Operations in OpenCL

Brian Brennan

Texas Tech University

June 5, 2012

Goal

Generate FEM code to run on various architectures:

Goal

Generate FEM code to run on various architectures:

$$\int_{\Omega} \nabla u \cdot \nabla v dx = \int_{\Omega} f v dx$$

Goal

Generate FEM code to run on various architectures:

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx$$

Architectures

Architectures

- Multicore CPU

Architectures

- Multicore CPU
- NVIDIA GPU

Architectures

- Multicore CPU
- NVIDIA GPU
- AMD GPU

Architectures

- Multicore CPU
- NVIDIA GPU
- AMD GPU
- Intel MIC

Motivation

Why are we interested in GPU computing?

Motivation

Why are we interested in GPU computing?

Pros:

- Very Fast
- Inexpensive
- Many Simple Cores

Motivation

Why are we interested in GPU computing?

Pros:

- Very Fast
- Inexpensive
- Many Simple Cores

Cons:

- Difficult to Program
- Architecture Specific
- Communication with CPU is Costly

Where to Begin?

Where to Begin?

Back to the basics!

Where to Begin?

Back to the basics!

```
y = a * (b - c) - d**2
```

PyOpenCL

What does PyOpenCL have to offer?

PyOpenCL

What does PyOpenCL have to offer?

PyOpenCL = Python + OpenCL

PyOpenCL

What does PyOpenCL have to offer?

PyOpenCL = Python + OpenCL

Python

- Easy to program
- Essential packages such as NumPy, SciPy, PyTrilinos, and many more

PyOpenCL

What does PyOpenCL have to offer?

PyOpenCL = Python + OpenCL

Python

- Easy to program
- Essential packages such as NumPy, SciPy, PyTrilinos, and many more

OpenCL

- General purpose parallel programming
- Vendor-neutral

PyOpenCL Example

```python
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
a_dev = cl_array.to_device(queue, a)
b_dev = cl_array.to_device(queue, b)
dest_dev = cl_array.empty_like(a_dev)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, None, a_dev.data, b_dev.data, dest_dev.data)
```

arraysum.py

Code Generation

We want all the ease of NumPy with the efficiency of PyOpenCL

Code Generation

We want all the ease of NumPy with the efficiency of PyOpenCL

```
y = a + b - 5.0*c - d**2
```

```
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global const float *c
    __global const float *d, __global float *y)
    {
      int gid = get_global_id(0);
      y[gid] = a[gid] + b[gid] - 5.0*c[gid] - d[gid]*d[gid];
    }
    """).build()
```

kernel.py

Main

```
if __name__ == "__main__":
    ctx = cl.create_some_context()
    queue = cl.CommandQueue(ctx)

    # declare random numpy arrays here #

    y_dev = cl_array.to_device(queue, a)
    a_dev = cl_array.to_device(queue, a)
    b_dev = cl_array.to_device(queue, b)
    c_dev = cl_array.to_device(queue, c)
    D_dev = cl_array.to_device(queue, c)
    Y = Vec( "y_dev" )
    A = Vec( "a_dev" )
    B = Vec( "b_dev" )
    C = Vec( "c_dev" )
    D = Vec( "d_dev" )

    kernel = assignVector(Y, A + B - 5.0*C - D*D)

    prg = cl.Program( ctx, kernel ).build()
    prg.op(queue, a.shape, None, y_dev.data, a_dev.data, b_dev.data,
        c_dev.data, d_dev.data)
```

mymain.py

Loop Kernel Output

```
__kernel void op( __global float *y_dev,
    __global const float *a_dev, __global const float *b_dev,
    __global const float *c_dev, __global const float *d_dev )
    {
      int gid = get_global_id(0);
      y[gid] = a[gid] + b[gid] - 5.0*c[gid] - d[gid]*d[gid];
    }
```

mykernel.py

GPU vs. CPU Results

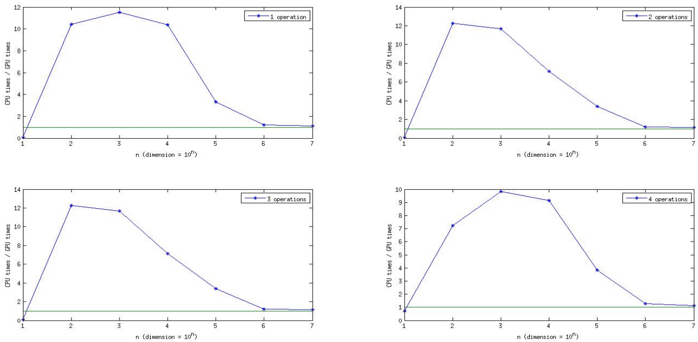

Figure: Ratio of CPU run times / GPU run times for 1, 2, 3, 4 basic vector operations.


What's
Next?

Loopy

$$\int_{\Omega} \nabla u \cdot \nabla v dx = \int_{\Omega} f v dx$$

```
for c in range(num_cells):
  for i in range(num_bf):
    for j in range(num_bf):
      K_loc[c,i,j] = 0.0
      for k in range(num_qp):
        K_loc[c,i,j] += jacs_det[c] * qwts[k] \
          * ( (jacs_inv[c,0,0] * bgrads[i,k,0] + jacs_inv[c,1,0] * bgrads[i,k,1] ) \
          * (jacs_inv[c,0,0] * bgrads[j,k,0] + jacs_inv[c,1,0] * bgrads[j,k,1] ) \
          + (jacs_inv[c,0,1] * bgrads[i,k,0] + jacs_inv[c,1,1] * bgrads[i,k,1] ) \
          * (jacs_inv[c,0,1] * bgrads[j,k,0] + jacs_inv[c,1,1] * bgrads[j,k,1] ) )
```

poissonloop.py

```
for c in range(num_cells):
  for i in range(num_bf):
    for j in range(num_bf):
      K_loc[c,i,j] = 0.0
      for k in range(num_qp):
        K_loc[c,i,j] += jacs_det[c] * qwts[k] \
          * ( (jacs_inv[c,0,0] * bgrads[i,k,0] + jacs_inv[c,1,0] * bgrads[i,k,1] ) \
          * (jacs_inv[c,0,0] * bgrads[j,k,0] + jacs_inv[c,1,0] * bgrads[j,k,1] ) \
          + (jacs_inv[c,0,1] * bgrads[i,k,0] + jacs_inv[c,1,1] * bgrads[i,k,1] ) \
          * (jacs_inv[c,0,1] * bgrads[j,k,0] + jacs_inv[c,1,1] * bgrads[j,k,1] ) )
```

poissonloop.py

In a Galaxy Far Far Away

Expand to large scale software packages:

In a Galaxy Far Far Away

Expand to large scale software packages:

In a Galaxy Far Far Away

Expand to large scale software packages:

In a Galaxy Far Far Away

Expand to large scale software packages:

Trilinos: Sundance Example

```
// create symbolic objects for test and unknown functions
Expr v = new TestFunction(new Lagrange(2));
Expr u = new UnknownFunction(new Lagrange(2));

// create symbolic differential operators
Expr dx = new Derivative(0,1);
Expr dy = new Derivative(1,1);
Expr grad = List(dx, dy);

// Write symbolic weak equation and Neumann and Robin BCs
Expr poisson = Integral(-(grad*v)*(grad*u)-f*v, new GaussQuadrature(2))
    + Integral(top, v/3.0) + Integral(right, v*(rightBCExpr - u));


// Write essential BCs:
// Bottom: u=x^2
EssentialBC bc = EssentialBC(bottom, v*(u - 0.5*x*x), new
    GaussQuadrature(4));

// Assemble everything into a problem object, with a specification that
// Petra be used as the low-level linear algebra representation
StaticLinearProblem prob(mesh, poisson, bc, v, u, petra);
```

heat2d.cpp

Questions

Thank you!

- Dr. Andreas Klöeckner

Thank you!

- Dr. Andreas Klöeckner
- Dr. Robert Kirby

Thank you!

- Dr. Andreas Klöeckner
- Dr. Robert Kirby
- Simula