ABSTRACT

A Parallel Implementation of the Galerkin Method for Solving Partial Differential Equations on a Triangular Mesh

Rachel N. Hess

Director: Robert C. Kirby, Ph.D.

Finite Element Methods are techniques for estimating solutions to boundary value problems for partial differential equations from an approximating subspace. These methods are based on weak or variational forms of the BVP that require less of the problem functions than what the original PDE would suggest in terms of order of differentiability and continuity. In the scope of this project, we focused on implementing the Galerkin Finite Element Method, which provides a best approximation to the true solution from a finite-dimensional subspace of piecewise polynomial functions defined on a triangular mesh. For this thesis, we developed a shared memory parallel implementation of the Galerkin Method that can be executed on a GPU to minimize runtime by means of multiple processors working simultaneously in unison on each calculation. For this purpose, we used the open-source libraries PyOpenCL and Loo.py. Thus we are able to explore how essential tasks in the solution process map onto shared memory platforms, such as the construction of the stiffness matrix from the connectivity data of the triangular mesh that may then be used to approximate the true solution with numerical methods.

APPROVED BY DIRECTOR OF HONORS THESIS:

____

Dr. Robert C. Kirby, Department of Mathematics

APPROVED BY THE HONORS PROGRAM:

____

Dr. Andrew Wisely, Director

DATE: ____________________

A PARALLEL IMPLEMENTATION OF THE GALERKIN METHOD FOR SOLVING PARTIAL DIFFERENTIAL EQUATIONS ON A TRIANGULAR MESH

A Thesis Submitted to the Faculty of

Baylor University

In Partial Fulfillment of the Requirements for the

Honors Program

By

Rachel N. Hess

Waco, Texas

May 2015

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER ONE

An Introduction to Parallel Shared Memory Organization

In this chapter, I will discuss the basic principles of parallel programming on a shared memory machine. To perform computations in a parallel structure requires specific hardware and interfaces that will support parallel algorithms within a multiprocessor system or across multiple systems. As I will be using the PyOpenCL interface for a GPU environment in my implementation of the Finite Element Method, I will thus explain the structure of the OpenCL interface from which PyOpenCL was derived and generally introduce the GPU as one of the environments on which OpenCL may be run. Upon introducing the PyOpenCL structure, I will then explore some of the parallel algorithms that will be used in my later implementation.

What is Parallel Programming?

To execute something in parallel means to divide a larger problem into smaller computations that can be performed simultaneously by multiple system components used to compute the final solution. There are two main memory organizations for parallel computing: through a distributed memory machine or through a shared memory machine. In the context of shared memory organization, the parallelism of the computation necessitates the use of synchronization between threads. Data of the application is stored as global memory that can be accessed by all of the processors or cores of the system hardware. This way, multiple processors may access the global memory simultaneously, or in parallel, to increase overall performance by reducing runtime. The threads share memory by writing variables that can then be subsequently read by another thread, thus making the synchronization of threads necessary to ensure that no shared data variables can be accessed by one thread while another is still writing to said variables. In my later implementation, I will be using programming models that will support this type of hardware. In comparison, a distributed memory machine is a computer system consisting of multiple processing units, or nodes, and a network with which data can be passed. With this type of hardware system, data and calculations are partitioned among the nodes, but in this case, each node contains its own private memory and information exchange is performed by passing data to and from the nodes via the interconnected network. [9]

Hardware and Interfaces

The utilization of parallelism requires implementation on hardware with parallel computing capabilities and the use of specialized programming interfaces. One such hardware example mentioned earlier is a multicore system. Other possible examples of hardware capable of parallel computation include the Intel Xeon Phi coprocessors or the Graphics Processing Unit. The GPU was created to handle the type of data typical of graphics applications. Such high-performance data handling capabilities can thus be adapted for shared memory computing to efficiently execute algorithms in parallel so as to reduce the runtime. Furthermore, the GPU is becoming an incredible resource for parallel computing as a fairly cheap and easily accessible piece of hardware available in multiple platforms across a wide community. [9]

Using the proper hardware, a computing language with parallel programming capabilities across multiple processors is required. OpenCL was the first cross-platform open standard for this type of computing. [5] Through this interface, the programmer uses a host application to communicate to the device, dictated by five basic structures with which an algorithm may run in parallel. These structures include a device, kernel, program, command queue, and context. Looking at each individual data structure, the overall format of OpenCL and its parallel nature can thus be observed. First, the host is the user's computer from which kernels are dispatched to connected devices. [10] The context is defined as a container that organizes the interaction between the host and the device, manages the available memory objects, and keeps track of all created kernels and projects. [2] In other words, the context is the means through which data and kernels are transferred with each command being sent to and stored into a command queue. For this queue, each command is thus stored in the order with which the host will use to request an action of a device. Further, a program is then a collection of kernels to be distributed. [2]

From here, we may then look into an additional computing interface that sits on top of OpenCL, known as PyOpenCL. PyOpenCL is an open-source package that enables access to OpenCL with Python, thus granting users the tools with which to more easily implement complex algorithms through the use of additional aids and templates that the interface provides. With this, PyOpenCL has certain advantages, including an easier object cleanup, all-accessibility, automatic error checking, speed, and open-source licensing. [4] I will be using PyOpenCL in the later chapters as I implement the Finite Element Method in parallel.

Basic Parallel Algorithms

Prefix sums are an important building block in shared memory algorithms. A prefix sum takes a binary associative operator $\oplus$ and an ordered set of n elements $[a_0, a_1, ..., a_{n-1}]$, before returning the ordered set $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$. [1] A binary associative operator is defined as an operator under associativity that takes two data members of a set and combines them to produce a new data member. This operation need not simply be addition or multiplication, though these meet the specifications. For example, prefix sums can be used to calculate a cumulative sum of integers. As the operation takes in two integers and combines them into one by summing them, this operation shows itself to be a binary operator. What is left to show is whether the operation in associative, which follows from properties of addition of integers. Recognizing certain common algorithms, PyOpenCL has created templates for the frequent use of such kernels in the implementation of an algorithm. Some of the kernel templates include Elementwise, Scan, Reduce, and RadixSort, all of which use prefix sums except for the Elementwise kernel. [4]

Elementwise Expression Evaluation

According to the PyOpenCL documentation, the kernel builder for an element-wise operation will evaluate expressions through one or more operations across each element in a single pass. By passing in the arguments on which the computation will be performed and the operation to be performed on each element of an argument, the kernel will perform the specified task in parallel across each element. This form of calculation is also known as a mapping from the arguments. Through the way the kernel is defined with its specified parameters, multiple processors are able to perform the computational operation on different elements simultaneously to enable the desired parallel structure.

```
class  pyopencl.elementwise.ElementwiseKernel(context, arguments,
       operation, name="kernel", preamble="", options=[])
```

Figure 1: ElementwiseKernel Signature

Such a kernel will have the signature shown in Figure 1. For the first three parameters, "context" is the context within which the code will be generated, "arguments" is the list of arguments on which to operate, and "operation" defines the operation to be performed. Then, "name" specifies the function name, "preamble" is a section of the kernel's source code outside the function's context, and "options" is passed directly to the build program.

```
krnl = ElementwiseKernel(ctx, "__global int *x, __global int *y,
                           <> temp",
                            operation="""if(x[i]>y[i]){
                                          temp = x[i];
                                          x[i] = y[i];
                                          y[i] = temp;}""")
```

Figure 2: ElementwiseKernel Example

One example of an ElementwiseKernel that I will be using in my later implementation is shown in Figure 2. Under this kernel, we see the parameters ctx as the context, global variables of integer array x, integer array y, and integer temp, and then the operation to be performed over the variables. In this example, the operation is checking to see whether the elements of each corresponding index of array x are greater than the elements of array y and if so, swaps them. This operation necessarily falls under an ElementwiseKernel because this operation must be performed for every index i across the two variable arguments assumed to be of equal size.

Reductions

The *reduce* operation is based on a prefix sum by taking a binary operator $\oplus$ with identity i, and an ordered set $[a_0, a_1, ..., a_{n-1}]$ of n elements to return $a_0 \oplus a_1 \oplus ... \oplus a_{n-1}$. [1] Because the binary operator is associative, the operand can be applied to any subset of the data set, allowing for the usage of parallelism to partition the data into tasks sent to multiple processors. Likewise, this reduction is clearly seen to be a binary associative operator applied across an entire data set, making it a type of prefix sum. In this case, a reduction will return the final entry of a cumulative sum.

```
krnl = ReductionKernel(ctx, numpy.float32, neutral="0",
        reduce_expr="a+b", map_expr="x[i]*y[i]",
        arguments="__global float *x, __global float *y")
```

Figure 3: ReductionKernel Signature

A common reduction kernel is to perform a dot product of two vectors, shown in Figure 3. A dot product takes the product of corresponding elements in two vectors of equal size across each element, as defined in the map_expr, and then returns a summation of each of these element-wise computations, as defined in the reduce_expr. In this way, a dot product is a reduction by multiplying corresponding elements between two vectors of equal size and summing the products. This is thus implemented in PyOpenCL through the usage of the ReductionKernel template by first mapping the vectors x and y with a multiplication operation then reducing this mapping by taking the sum across all of the elements. [7]

Scans

As defined by Blelloch, the scan operation is a vector all-prefix-sums operation. [1] Therefore, a scan is a cumulative sum, or a running operation in which the final result of performing the algorithm has taken account of all elements in the array in the solution. Thus, a scan is much like the simpler reduction operation in that it also performs summations across an array, except that it also stores the running result with each element whereas the reduction kernel only stores the last element of the scan operation. In PyOpenCL, this calculation can be performed with the Generic Scan Kernel, for which the operation can be specified as any binary associative operator. One such example of another type of scan is a prescan, which is the running sum of the elements of an array, not including the current element.

```
class pyopencl.scan.GenericScanKernel(context, dtype, arguments,
        input_expr, scan_expr, neutral, output_statement,
        is_segment_start_expr=None, input_fetch_exprs=[],
        index_dtype=<'numpy.int32'>, name = 'scan', options=[],
        preamble='''', devices=None)
```

Figure 4: GenericScanKernel Signature

This Generic Scan Kernel will have the signature shown in Figure 4 as defined in its documentation. In this case, the parameters for context, arguments, name, options, and preamble are the same as the corresponding parameters defined above in the Elementwise Kernel. “dtype” will specify the data type with which to perform the scan. “input_expr” defines an expression of the values on which the scan is applied with “scan_expr” defining the operation to carry out on the values. Then, “output_ statement” writes the output of the scan. Finally, “is_segment_start_expr” determines whether a new scan segment starts at index i with each segment a set of values on which the scan is executed, and “input_fetch_exprs” is a list of tuples.

Radix Sort

A radix sort is a type of sort that looks at the binary representation of all of the elements in a vector or list to reorder them sequentially. This sorting algorithm thus flags indices of the data according to each individual specific element. To put it technically, a radix sort is a sorting algorithm in which the elements are looped over sequentially while the algorithm checks over each of the elements' bits, starting with the lowest. All elements with a zero as the bit in the current iteration are packed to the bottom, leaving the elements with a one as that bit packed at the top of the list. This operation is known as a split operation, using the keys of zero or one in each iteration over the bits. This algorithm can thus be seen as a prefix sum by defining the split operation using scans. This Reduction Kernel is defined in PyOpenCL by the signature in Figure 5.

```
class pyopencl.algorithm.RadixSort(context, arguments, key_expr,
      sort_arg_names, bits_at_a_time=2, index_dtype=<‘numpy.int32’>,
      key_dtype=<‘numpy.uint32’>, options=[])
```

Figure 5: RadixSort Signature

Within this signature, “key_expr” is an expression to return the key on which the sort is performed and “sort_arg_names” is a list of argument names whose array arguments will be sorted according to the key. For the remaining parameters, their content is evident or has already been defined above.

CHAPTER TWO

The Galerkin Method

This chapter will provide a basic understanding of the Galerkin method by solving partial differential equations. The Galerkin method is a best approximation to the true solution from a finite-dimensional subspace, under the proper conditions derived from a projection theorem that I will outline later in this chapter. In order to set up the method, the partial differential equation must first be relaxed into its weak form so that the function requirements are eased, allowing for a broader range of computations to be performed, such as the Galerkin method. In this chapter, I will discuss the setup and use of the Galerkin method through calculations performed on a boundary value problem that transform it into its weak, variational form, allowing for the method's use. I will then begin to discuss the triangular mesh on which the boundary value problem will be solved. In the next chapter, I will then discuss the parallel implementation of connectivity on the triangular mesh in preparation for finite element methods.

Relaxing the PDE

In order to implement the Galerkin method to solve a partial differential equation, that boundary problem must first be changed into its weak form, or variational form, to loosen the restraints required. Some typical examples of problems in variational forms include:

Example 1:

$u \in H_0^1(\Omega), \int_\Omega K \bigtriangledown u \cdot \bigtriangledown v = \int_\Omega fv$ for all $v \in H_0^1(\Omega)$

is the variational form of the Dirichlet BVP:

$- \bigtriangledown \cdot (\kappa \bigtriangledown u) = f$ in Ω, $u = 0$ on $\partial\Omega$

Example 2:

$u \in H^1(\Omega), \int_\Omega K \bigtriangledown u \cdot \bigtriangledown v = \int_\Omega fv + \int_{d\Omega} vh$ for all $v \in H^1(\Omega)$

is the variational form of the BVP with inhomogeneous Neumann conditions:

$- \bigtriangledown \cdot (\kappa \bigtriangledown u) = f$ in Ω, $\kappa \frac{\partial u}{\partial n} = h$ on $\partial\Omega$, where h is a function defined on $\partial\Omega$

In comparison to the variational equations, the original boundary value problems included the use of the divergence operator, which signifies the calculation of the partial second derivatives of the variables. In order to perform this calculation, the variables within the equation must then be twice differentiable. Additionally, the right-hand side of the equation requires continuity. Although, when we take the variational form of the boundary value problem, we are integrating the functions on each side of the equality over their common domain. Therefore, the functions now only require differentiability instead of twice differentiability, and the right-hand side only requires integrability instead of the stricter condition of continuity. By loosening these constraints, the variational forms of boundary values problems allow for more various calculations and analysis to be performed on the equations that would have not been possible if not for the transformation between forms. [3]

In general, to transform the PDE into the variational form, the following steps must be taken. First, multiply both sides of the PDE by a function defined in the domain. Then, since the two sides of the PDE are equal, integrate both sides by parts. Finally, Green's Theorem can be applied to arrive at the variational form of the PDE.

All of the example variational problems listed above can be rewritten in the general form $u \in V, a(u,v) = l(v)$ for all $v \in V$. $l(v)$ is a linear functional that can be simply defined as a linear function from V to $\mathbf{R}$. Now because of its linearity, this linear functional has two implications on continuity. The first is that if l is continuous at any $u \in V$, it must be continuous at every $u \in V$. Secondly, l is continuous at u if and only if it is bounded. For l to be bounded, there must exist a nonnegative constant M such that $|l(u)| \leq M||u||$ for all $u \in V$. $a(\cdot,\cdot)$ is then the symmetric bilinear form that satisfies the three properties:

1.$a(u,v) = a(v,u)$ for all $u, v \in V$

2.$a(\alpha u + \beta v, w) = \alpha a(u,w) + \beta a(v,w)$ for all $u, v, w \in V$ and all $\alpha, \beta \in \mathbf{R}$

3.$a(u,u) \geq 0$, and $u = 0$ implies that $a(u,u) = 0$

In addition, if $a(\cdot,\cdot)$ has the property that there exists $\alpha > 0$ such that $a(u,u) \geq \alpha||u||^2$ for all $u \in V$, $a(\cdot,\cdot)$ is elliptic over V. Also, if $a(\cdot,\cdot)$ has the property that there exists $\beta > 0$ such that $a(u,u) \leq \beta||u||||v||$ for all $u, v \in V$, then $a(\cdot,\cdot)$ is bounded. [3]

Using the Galerkin Method

Projection Theorem

Providing the framework for the Galerkin method and other methods of best approximation is the theorem that given a finite-dimensional subspace W of inner product space V and vector $u \in V$, there exists a unique best approximation $w \in W$

that satisfies $||u-w|| < ||u-z||$ for all $z \in W$ not equal to w, where w is the projection of u onto W. This best approximation also satisfies the orthogonality condition that $w \in W$ is this best approximation if and only if the inner product of $u - w$ with z is equal to zero for all $z \in W$. In this theorem, we thus denote the inner product of two vectors in the inner product space by $(\cdot, \cdot)$. Now, orthogonality can be defined as a property associated to two vectors within a vector space whose inner product is equal to zero. In summary, based upon conditions of orthogonality, the projection theorem defines what the best approximation would be of a vector in an inner product space from a finite-dimensional space. [3]

Because W is finite-dimensional, it has a basis $\{w_1, ..., w_n\}$ that can be represented as $w = \sum_{j=1}^{n} \alpha_j w_j$. From the orthogonality condition, it is clear that using this basis definition of w that we may derive $\sum_{j=1}^{n}(w_j, w_i)\alpha_j = (u, w_i)$,where $i = 1, 2, ..., n$. Thus, from this conclusion, we may define $(w_j, w_i), i, j = 1, 2, \ldots, n$ as the Gram matrix G_{ij} and $(u, w_i), i = 1, 2, \ldots, n$ as vector b_i such that we have the system of linear equations $G\alpha = b$. Now, if it turns out that w defines an orthogonal basis, $G\alpha = b$ can be solved simply with $\alpha_i = \frac{(u,w_i)}{w_i,w_i}, i = 1, 2, ..., n$. Otherwise, $G\alpha = b$ can still be solved efficiently so long as G is nearly diagonal, or sparse, even when n is fairly large. [3]

Galerkin with Variational Problems

Since the true solution u is unknown, it cannot be computed directly from the finite-dimensional approximating subspace W of V. However, from here the Galerkin method can be used through the usage of the alternate inner product defined by the bilinear form $a(u, v)$ to compute the best approximation w of u on W. To use the Galerkin Method, the system of equations must be first redefined in terms of the bilinear form. So now we may define our system as $KU = F$ with stiffness matrix $K_{ij} = a(w_j, w_i), i, j, = 1, 2, ..., n$ and load vector $F_i = a(u, w_i) = l(w_i), i = 1, 2, ..., n$. Consequently, the load vector is thus calculable and known such that the only remaining unknown in our system of equations is the vector U, which defines our solution w as $w = \sum_{i=1}^{n} U_i w_i$. [3]

Therefore, the Galerkin method will produce a best approximation $w \in W$ to u from W that satisfies $||u - w||_E = min_{v \in W} ||u - v||_E$. Using Céa′s Theorem, we know that $||u - w|| \leq \frac{\beta}{\alpha} ||u - v||$ for all $v \in W$. From this theorem arises a central fact of the finite element method that $||u - w||$ cannot be too much larger than $||w - v||$ for all v in W. This is due to the condition that the Galerkin approximations $w_h \in W_h$ must improve at the same rate as the best approximations from W_h as $h \to 0$. [3]

The Galerkin Approximating Subspace

Mark S. Gockenbach, in his book Understanding and Implementing the Finite Element Method, defines the finite element method as Galerkin's method with a subspace of piecewise polynomial functions. When selecting an approximating a subspace, the method must be efficient in the computing of K and F and the solving of the system and also must approximate the true solution well by an element of the subspace. By selecting piecewise polynomials for our approximating subspace, we are able to meet the criteria effectively with the given properties of guaranteed differentiable and integrable functions due to the properties of polynomials. [3]

In order to construct such a subspace of polynomials, we must partition the domain into subdomains. Polynomials are then defined on each subdomain to create a subspace. Most commonly, the domain will be partitioned into triangular or rectangular subdomains on which the polynomials are defined, of which, we will use a triangular subdomain. Then, this collection of triangular subdomains is known as a triangular mesh, and must be constructed in such a way that any intersection of two triangles is either node to node or edge to edge. On this mesh, a piecewise polynomial must then be defined over all of the individual triangular subdomains with the piecewise polynomial reducing to a polynomial of desired order over each subdomain. For example, if linear polynomials were used, the piecewise polynomial over the entire mesh must reduce to a linear polynomial over each triangle. In this construction, the piecewise polynomial must maintain continuity. Therefore, the polynomials over each specific triangle must satisfy the conditions that whenever any two triangles meet at a node, their respective polynomials must be equal at that node, and whenever two triangles share an edge in the mesh, their respective polynomials must be equal at every point along that edge. In the next chapter, I will discuss the triangular mesh in more detail as well as begin my implementation of Galerkin's method, starting with the construction of the mesh and enumeration of mesh data. [3]

CHAPTER THREE

The Triangular Mesh

In this chapter, I will discuss the triangular mesh on which we will implement the Galerkin Method and how to set up all of the necessary data that will be needed for the calculations. In order to solve the problem, we will first need to generate the triangular mesh then organize the data from the mesh by assembling connectivity matrices to describe mesh entity interaction. These connectivity relations within the mesh serve as an effective representation of the triangular mesh as data structures most convenient to access when such data is needed for our main calculations, which I will describe in the next chapter.

Generating the Mesh

As described in the previous chapter, a triangular mesh is needed as the Galerkin Approximating Subspace to solve the given partial differential equation. This mesh can be obtained through mesh generating software that will create the mesh and provide files describing the details of the mesh. For this project, I used Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator created by Jonathan Richard Shewchuk to generate a two-dimensional triangular mesh with accompanying files. For connectivity purposes, I will be using the *.ele* and *.node* files to execute connectivity implementations. An example of a simple two-triangle mesh is shown in Figure 6.

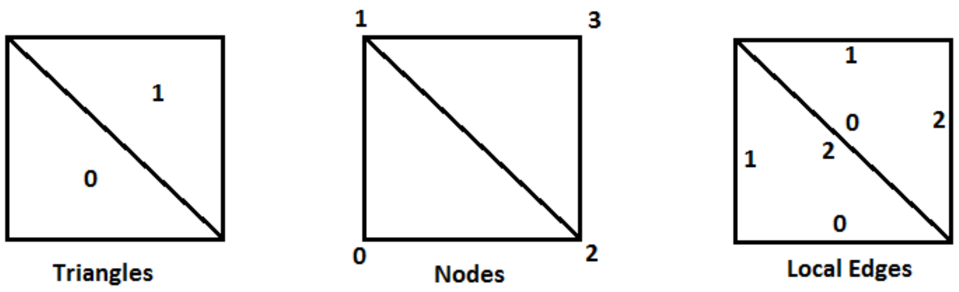


Figure 6: Triangular Mesh Example

The *.ele* file has the format of storing the number of triangles, number of nodes per triangle, and number of attributes on the first line. Each of the remaining lines then store a triangle number followed by list of nodes on that triangle with their attributes. The triangle attributes are typically floating-point values to represent specific qualities of the triangle such as color, mass, or conductivity. This file will be used as input while computing connectivity information. Following in the next chapter, I will be using the *.node* file to compute the basis functions in constructing the stiffness matrix and load vector. This file is formatted to store the number of vertices, dimension, number of attributes, and number of boundary markers for the mesh on the first line. The remaining lines each store a vertex number with the coordinates, attributes, and boundary marker of that vertex. Here, the boundary markers are a binary tag for whether each specific vertex is on the boundary of the mesh, with the tag set to 1 to signify that the vertex is on the boundary, 0 otherwise. From this file, I will be able to create a matrix to store the coordinates of each vertex for easy access throughout my calculations. [11] Example *.ele* and *.node* files for the mesh in Figure 6 can be referenced in Appendix A.

Connectivity

In computing the mesh connectivity before beginning any Finite Element Method computations towards solving partial differential equations, this will effectively increase efficiency and reduce runtime by allowing easy access to mesh data. [8] Through my implementation, I will compute this information into array-based data structures that will serve to provide instant access to mesh and connectivity information by simply calling the desired data structure at the proper index. Therefore, I will be using Loo.py in my implementation as it is a code generator for array-based data in PyOpenCL and will thus be able to generate the proper data structures for my representation. In this way, the connectivity of the mesh serves as a mesh representation more suited for a programming implementation of Finite Element Methods. [6]

In addition to serving as an alternative to a graphical representation of the triangular mesh, the connectivity information calculated in this chapter can be applied to various situations. For example, not all of the connectivity information from this chapter will be directly used in the scope of this project but can be applied elsewhere, such as in other Finite Element Methods extending into higher dimensions. Therefore, in computing the mesh connectivity, we will explore how such an exercise can be executed in parallel and expressed as prefix sums.

Enumerating Edges

In order to obtain edge information, the edges must first be constructed from the mesh data. This can be accomplished by reading in the data from the *.ele* file and characterizing each edge as the connection between two nodes. The file must be opened and the lists of triangle nodes read into a $3 \times N$ matrix with N being the number of triangles in the mesh as stated on the first line of the file. This way, each row of the matrix, corresponding to its respective triangle, will contain the three nodes on that triangle, with implementation shown in Figure 7. Using this data, the task of enumerating the edges falls to the use of a Loo.py kernel that will construct two corresponding arrays by looping through the matrix such that the first array will contain the first node to each edge and the second array will contain the second, with implementation shown in Figure 8. It is through this kernel that the edge location data is also obtained. By storing the edges in this manner, it is important to keep in mind that any alteration performed on one array must extend to the other as well or otherwise fail to accurately represent the edges.

```
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array

f = open('littlebox.ele', 'r')
nels = f.readline()
nels = nels.split()
cells = np.zeros((int(nels[0]), 3), dtype=np.int32)

for x in range(0, int(nels[0])):
    line = f.readline()
    line = line.split()
    cells[x, :] = map(int, line[1:])
```

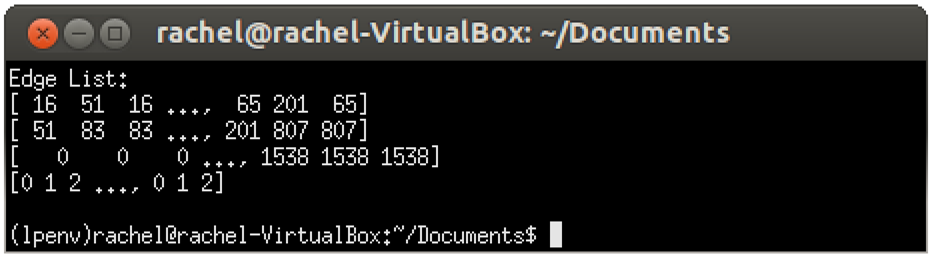
Figure 7: Assembling the Cell Matrix

```
import loopy as lp
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx,
                        properties=
                        cl.command_queue_properties.PROFILING_ENABLE)

loop_bounds = "{[c, i, j]: 0 <= c < nels and 0 <= i < 3 and i < j < 3}"

kernel_code = """
<> ed = 0 {inames=c, id=init_ed}
edges0[c, ed] = cells[c, i] {id=write_v0, dep=init_ed, inames=c:i:j}
edges1[c, ed] = cells[c, j] {id=write_v1, dep=init_ed, inames=c:i:j}
triangle_num[c, ed] = c {id=write_v2, dep=init_ed, inames=c:i:j}
edge_num[c, ed] = j+i-1 {id=write_v3, dep=init_ed, inames=c:i:j}
ed = ed + 1 {dep=write_v0:write_v1:write_v2:write_v3, inames=c:i:j}
"""

args = [lp.ValueArg("nels", np.int32),
        lp.GlobalArg("cells", np.int32, shape=("nels", 3)),
        lp.GlobalArg("edges0", np.int32, shape=("nels", 3)),
        lp.GlobalArg("edges1", np.int32, shape=("nels", 3)),
        lp.GlobalArg("triangle_num", np.int32, shape=("nels", 3)),
        lp.GlobalArg("edge_num", np.int32, shape=("nels", 3))]

knl = lp.make_kernel(loop_bounds,
                     kernel_code,
                     args,
                     assumptions="nels>=1")

cknl = lp.CompiledKernel(ctx, knl)
```

Figure 8: Enumerating the Edges

Now with the edge list, it is beneficial to organize the list so that the data may be accessed quickly and efficiently without requiring searching through the entire edge list for a single data point. This will be accomplished by implementing sorting kernels to order the list. First, an ElementwiseKernel must be used to perform a check on each edge by index in the list and ensure that the lesser enumerated node is listed in the first array with the greater at the corresponding index of the second array. This

```
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
from pyopencl.elementwise import ElementwiseKernel

sort1 = ElementwiseKernel(ctx, "__global int *x, __global int *y,
                          __global int temp",
                          operation="""if(x[i]>y[i]){
                                       temp = x[i];
                                       x[i] = y[i];
                                       y[i] = temp;}""")

temp = 0
sort1(edges0.ravel(), edges1.ravel(), temp)

print "Edge List:"
print edges0.ravel()
print edges1.ravel()
print triangle_num.ravel()
print edge_num.ravel()
print
```

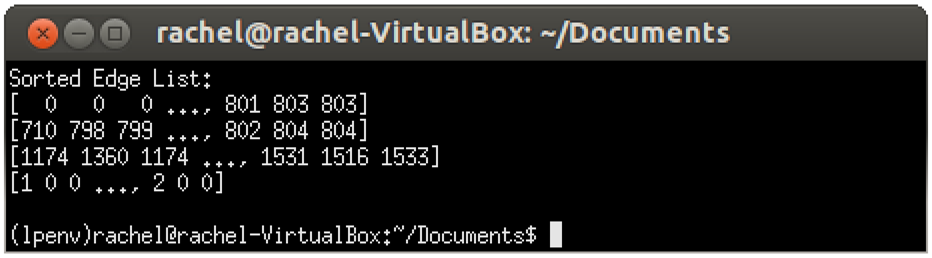


```
Edge List:
[ 16  51  16 ...,  65 201  65]
[ 51  83  83 ..., 201 807 807]
[   0    0    0 ..., 1538 1538 1538]
[0 1 2 ..., 0 1 2]

(lpenv)rachel@rachel-VirtualBox:~/Documents$
```

Figure 9: Edge Ordering

algorithm, shown in Figure 9, safeguards the integrity of the following calculations by representing all edges the same way. Otherwise, the edge say from node 1 to node 5 could be represented as $\left[\begin{smallmatrix}1\\5\end{smallmatrix}\right]$ or $\left[\begin{smallmatrix}5\\1\end{smallmatrix}\right]$. Now, a RadixSort kernel can be used to sort the edges in increasing order by their nodes, using a radix sort algorithm implemented in parallel through the kernel. By first ordering the edges by greater enumerated node and then the lesser, the edges will thus be sorted in ascending order primarily by the first node and then by the second node when multiple edges share the common lesser enumerated node. Additionally, this kernel will serve as a segmented scan that will append additional edge connectivity data to the index of each edge and maintain the edge information as the kernel is executed, as shown in Figure 10.

```
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
from pyopencl.algorithm import RadixSort

sort_edges = RadixSort(ctx,
                        """__global int *edge1, __global int *edge2,
                           __global int *tri, __global int *side""",
                        key_expr="edge1[i]",
                        sort_arg_names=["edge1","edge2","tri","side"])

(e1, e0, t, s), evt = sort_edges(edges1.ravel(), edges0.ravel(),
                                 triangle_num.ravel(), edge_num.ravel())

(e0_srted, e1_srted, tri, side), evt = sort_edges(e0, e1, t, s)

print "Sorted Edge List:"
print e0_srted
print e1_srted
print tri
print side
print
```



```
Sorted Edge List:
[   0    0    0 ..., 801 803 803]
[710 798 799 ..., 802 804 804]
[1174 1360 1174 ..., 1531 1516 1533]
[1 0 0 ..., 2 0 0]

(lpenv)rachel@rachel-VirtualBox:~/Documents$ 
```

Figure 10: Sorting the Edge List

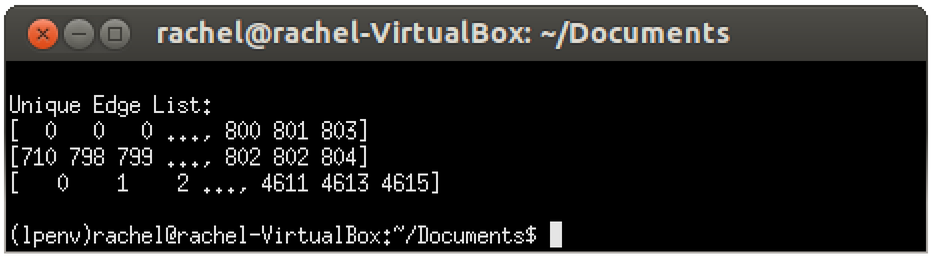
Edge Connectivity Data

Within the algorithms described above, connectivity of the edges to the triangles is obtained and carried through the sorting algorithms through the use of segmented scans. Further, the connectivity obtained is that from the edge to the triangle on which it lies and the enumeration of the edge on the triangle. While looping over the node information of each triangle in the Loo.py kernel, the scan simultaneously created arrays to record the triangle number to which each edge is associated and the local edge number of the edge on the triangle: either side 0, 1, or 2. In other words, the connectivity obtained here represents where the edge is located globally in the mesh by which triangle it lies on as well as where the edge is located locally on its triangle by enumerating the three sides of the triangle and recording which of the three sides of the triangle is represented by the edge.

Unique Edges

Once the edges have been sorted, it follows that to remove any duplicate edges in the list, one need only to check each edge against the edges listed immediately before and after it to ensure it is unique. These duplicate edges arise in the list because of the method in which the edges were enumerated; the Loo.py kernel traversed over each triangle for its edges, so if two triangles shared an edge, that edge will be duplicated in the list. With the implementation of a ReductionKernel, shown in Figure 11, this operation may thus be performed to create a unique edge list separate from the original enumerated list. Because of this, the kernel only uses the two arrays composing the edge list as its arguments to create the unique list, keeping in mind that the edge connectivity information would not apply in this context as many of the edges are not unique to a single triangle.

```
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
from pyopencl.reduction import ReductionKernel
from pyopencl.scan import GenericScanKernel

unique = ReductionKernel(ctx, np.int32, neutral="0",
         reduce_expr="a+b",
         map_expr="((i==0)||(x[i]!=x[i-1])||(y[i]!=y[i-1])) ? 1 : 0",
         arguments="""__global int *x, __global int *y""")

remove_duplicates = GenericScanKernel(ctx, np.int32,
                """__global int *x, __global int *y,
                   __global int *final0, __global int *final1,
                   __global int *index""",
                   input_expr="""
                   ((i==0)||(x[i]!=x[i-1])||(y[i]!=y[i-1])) ? 1 : 0""",
                   scan_expr="a+b", neutral="0",
                   output_statement="""
                                     if((prev_item!=item))
                                     {final0[item-1] = x[i];
                                      final1[item-1] = y[i];
                                      index[item-1] = i;}""")

keys = unique(e0_srted.ravel(), e1_srted.ravel())

num_edges = keys.get()
final0 = np.zeros(num_edges, dtype=np.int32)
final1 = np.zeros(num_edges, dtype=np.int32)
ind = np.zeros(num_edges, dtype=np.int32)
final_0 = cl_array.to_device(queue, final0)
final_1 = cl_array.to_device(queue, final1)
index = cl_array.to_device(queue, ind)

remove_duplicates(e0_srted.ravel(), e1_srted.ravel(), final_0,
                  final_1, index);

print "Unique Edge List:"
print final_0
print final_1
print index
print
```



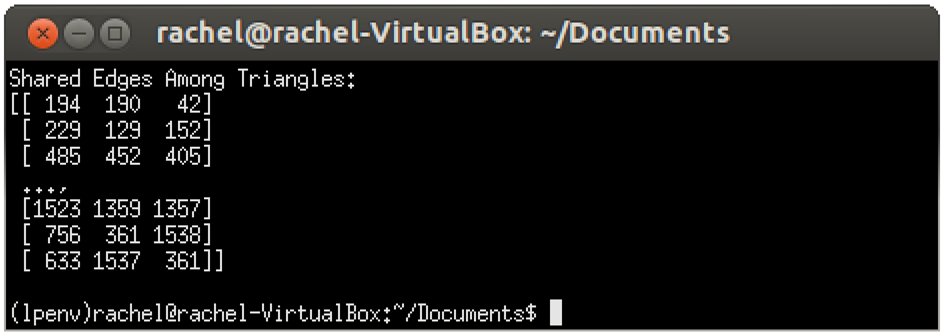
```
rachel@rachel-VirtualBox: ~/Documents

Unique Edge List:
[  0   0   0 ..., 800 801 803]
[710 798 799 ..., 802 802 804]
[   0    1    2 ..., 4611 4613 4615]

(lpenv)rachel@rachel-VirtualBox:~/Documents$
```

Figure 11: The Unique Edge List

Shared Edges

From the data of the enumerated edge list and edge connectivity data, a GenericScanKernel may then be performed with said data as its arguments to detail the connectivity between triangles that share an edge. This kernel will thus be able to scan over the enumerated edge list for the duplicate edges that indicate a shared edge between triangles as the edge is listed for each triangle it lies on. This information is then recorded into a matrix with each row representing the corresponding enumerated triangle, and each of the three columns representing the three edges of that triangle. If a triangle edge is not shared with another triangle, a -1 is stored for that edge in the matrix. Then, if the edge is a shared edge, the number of the triangle of which that edge is shared is stored at the corresponding matrix location. The implementation of this kernel is shown in Figure 12.

```
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
from pyopencl.scan import GenericScanKernel

triangle_edges = GenericScanKernel(ctx, np.int32,
                """__global int *x, __global int *y,
                   __global int *tri, __global int *side,
                   __global int *test""",
                   input_expr="""
                   ((i==0)||(x[i]!=x[i-1])||(y[i]!=y[i-1])) ? 1 : 0""",
                   scan_expr="a+b", neutral="0",
                   output_statement="""
                       if((prev_item==item))
                       {test[(3*tri[i-1])+side[i-1]] = tri[i];
                        test[(3*tri[i])+side[i]] = tri[i-1];}""")

test = np.ones([int(nels[0]), 3], dtype=np.int32)
test = test * (-1)
test1 = cl_array.to_device(queue, test)

triangle_edges(e0_srted.ravel(), e1_srted.ravel(), tri, side,
               test1.ravel());
np.reshape(test1, (int(nels[0]), 3))

print "Shared Edges Among Triangles:"
print test1
print
```



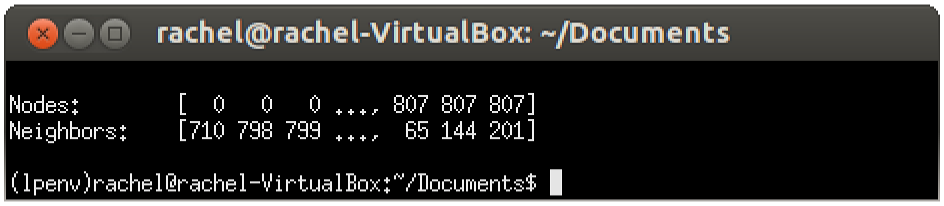
```
Shared Edges Among Triangles:
[[ 194  190   42]
 [ 229  129  152]
 [ 485  452  405]
 ...,
 [1523 1359 1357]
 [ 756  361 1538]
 [ 633 1537  361]]

(lpenv)rachel@rachel-VirtualBox:~/Documents$
```

Figure 12: Shared Edge Triangle Connectivity List

Node-to-Node Connectivity

The node-to-node connectivity can then be represented similar to the enumerated edge list by constructing two arrays with the nodes of the first array associated to the corresponding nodes of the second array as being connected by an edge. The unique edge list can therefore be seen as half of this connectivity by representing one node connectivity for each edge going from lower to higher enumerated node. Thus, the full connectivity list can be created by appending an inverted edge list from higher to lower node connectivity to the original edge list using an ElementwiseKernel shown in Figure 13. The final step would then be to sort the new list using the same RadixSort kernel used to originally sort the edge list.

```
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
from pyopencl.elementwise import ElementwiseKernel
from pyopencl.algorithm import RadixSort

double_edges = ElementwiseKernel(ctx, """__global int *ed0,
                           __global int *ed1, __global int *x,
                           __global int *y, __global int size""",
                           operation="""if(i<size){
                                            ed0[i]=x[i];
                                            ed1[i]=y[i];}
                                      else{ ed0[i]=y[i-size];
                                            ed1[i]=x[i-size];}""")

double0 = np.zeros(num_edges*2, dtype=np.int32)
double1 = np.zeros(num_edges*2, dtype=np.int32)
double_0 = cl_array.to_device(queue, double0)
double_1 = cl_array.to_device(queue, double1)
double_edges(double_0, double_1, final_0, final_1, num_edges)

(d1, d0, x, y), evt = sort_edges(double_1.ravel(), double_0.ravel(),
                             double_0.ravel(), double_0.ravel())

(d0_srted, d1_srted, trash_x, trash_y), evt = sort_edges(d0,d1,x,y)

print "Nodes:        ", d0_srted
print "Neighbors:    ", d1_srted
print
```



```
rachel@rachel-VirtualBox: ~/Documents

Nodes:        [  0   0   0 ..., 807 807 807]
Neighbors:    [710 798 799 ...,  65 144 201]

(lpenv)rachel@rachel-VirtualBox:~/Documents$
```

Figure 13: Node to Node Connectivity List

Node-to-Triangle Connectivity

In this last implementation, the node to triangle connectivity shall be constructed in the form of two arrays with the first containing the nodes and the second containing the corresponding triangle numbers on which the nodes lie. For nodes connected to multiple triangles, they shall appear in the list for every time they are found on a unique triangle. This implementation shown in Figure 14 is achieved by means of an ElementwiseKernel that will traverse the enumerated edge list to create an array displaying the three nodes of each triangle in order by triangle number. To this may then be associated the triangle number array from the edge location data as it will correctly associate each node to its corresponding triangle. From here the last step is then to sort the connectivity list first in order by node and then by triangle so that the final connectivity list will display the nodes in enumerated order with all the triangles they lie on.

```
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
from pyopencl.elementwise import ElementwiseKernel
from pyopencl.algorithm import RadixSort

surround = ElementwiseKernel(ctx, "__global int *x, __global int *y,
                              __global int *nhbr",
                              operation="""if(i%3==1){
                                          nhbr[i]=y[i];}
                                          else{nhbr[i]=x[i];}""")

nhbr = np.zeros(int(nels[0])*3, dtype=np.int32)
neighbor = cl_array.to_device(queue, nhbr)

surround(edges0.ravel(), edges1.ravel(), neighbor)

(tnum, n, x, y),  evt = sort_edges(triangle_num.ravel(), neighbor,
                                   neighbor, neighbor)

(e_nhbr, t_nhbr, trash_x, trash_y),  evt = sort_edges(n, tnum, x, y)

print "Surrounding Triangles: ", t_nhbr
print "Nodes:                 ", e_nhbr
print
```

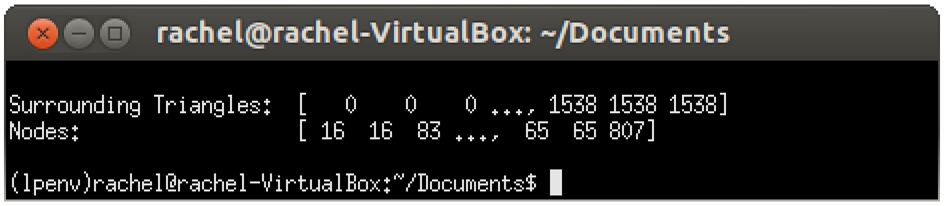

```
rachel@rachel-VirtualBox: ~/Documents

Surrounding Triangles:  [    0    0    0 ..., 1538 1538 1538]
Nodes:                  [ 16  16  83 ...,  65  65 807]

(lpenv)rachel@rachel-VirtualBox:~/Documents$
```

Figure 14: Node to Triangle Connectivity List

CHAPTER FOUR

Constructing the Stiffness Matrix

In this final chapter I shall explore constructing elementwise stiffness matrices for the Galerkin Method based upon the triangular mesh connectivity implemented in the previous chapter. This construction can be implemented efficiently by means of a Loo.py kernel that will traverse the mesh over each triangle so that the basis functions need only be calculated once on a reference triangle then transformed by a change of variables for each subsequent triangle. In this way, each triangle's contribution to the matrix may be calculated by transforming the reference data. This chapter will outline such implementation for the element stiffness matrix and discuss its application to shared memory platforms in which it may be used to numerically solve partial differential equations.

Element Stiffness Matrix

The construction of the stiffness matrix is invaluable to the Galerkin Method in solving partial differential equations, but there is more than one way in which this can be implemented. For example, the stiffness matrix can be constructed by looping over each element of the matrix to be calculated by $K_{ij} = \sum_{k=1}^{t} \int_{T_k} \kappa \bigtriangledown \phi_j \cdot \bigtriangledown \phi_i$, where each i, j pair represents an element of the matrix, ϕ_i and ϕ_j are basis functions for the mesh approximating subspace, and T_k is the triangle in the mesh to which the basis functions correspond in reference to the current element of stiffness matrix

K is being calculated. The downside to this method of calculation is that the basis functions for each triangle in the mesh must be calculated multiple times as they contribute to many elements of the stiffness matrix.

Instead of looping over the i, j pairs of the matrix for calculation, we thus chose to loop over each triangle in the mesh to compute the element stiffness matrices for each triangle in the mesh so that each triangle's basis functions need only to be calculated once without needing to store them. With each triangle having a basis function corresponding to each of its three vertices, each element stiffness matrix will thus be a 3×3 matrix to account for all combinations of the basis functions. Each element of the element stiffness matrix can then be calculated as $\int_{T_k} \kappa \bigtriangledown \phi_j \cdot \bigtriangledown \phi_i$ with basis functions $\{\phi_1, \phi_2, \phi_3\}$ over each triangle T_k in the mesh. [3]

Reference Triangle

The use of a reference triangle, shown in Figure 15, in the matrix calculations will greatly improve the efficiency of our implementation by only having to directly compute basis functions ϕ_1, ϕ_2, and ϕ_3 once for the reference triangle and then used on the triangles in the mesh by means of transformation. Each basis function can be represented in the form $\phi_i(x, y) = a_i + b_i x + c_i y$ for (x, y) coordinates in the respective triangle of the mesh. The three basis functions for the reference triangle can then be calculated by solving the matrix equation:

$$\begin{bmatrix} 1 & x_0 & y_0 \\ 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ c_0 & c_1 & c_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $(x_0, y_0) = (0, 0)$, $(x_1, y_1) = (1, 0)$, and $(x_2, y_2) = (0, 1)$. By solving the matrix equation, the three basis functions are solved to be:

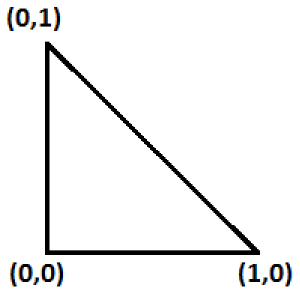


Figure 15: Reference Triangle

$$\phi_0(x,y) = 1 - x - y$$

$$\phi_1(x,y) = x$$

$$\phi_2(x,y) = y$$

After which, the gradients of the basis functions can be calculated to be the following:

$$\bigtriangledown\phi_0 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \qquad \bigtriangledown\phi_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \bigtriangledown\phi_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Now that we have the basis functions for the reference triangle calculated, we may begin our computations for the element stiffness matrices over the mesh triangles. [3]

Matrix Computation

Now each element stiffness matrix can be computed through the transformation of elements from the reference triangle to each triangle in the mesh. Therefore, instead of calculating $\int_{T_k} \kappa \bigtriangledown\phi_j \cdot \bigtriangledown\phi_i$, a change of variables must be performed on the integral, giving us $\int_{T_R} |det(J)|\kappa(J^{-T} \bigtriangledown \gamma_j) \cdot (J^{-T} \bigtriangledown \gamma_i)$, where γ is the transformation of basis ϕ and J is the Jacobian of the triangle. The formula for the change of variables of the basis functions defined in the previous section on the reference triangle is thus:

$$\gamma_i(s,t) = (a + bx_0 + cy_0) + (b(x_1 - x_0) + c(y_1 - y_0))s + (b(x_2 - x_0) + c(y_2 - y_0))t$$

Thus, the gradient of the transformed basis function can easily be defined by:

$$\triangledown \gamma_i = \begin{bmatrix} (b(x_1 - x_0) + c(y_1 - y_0)) \\ (b(x_2 - x_0) + c(y_2 - y_0)) \end{bmatrix}$$

Then, from Jacobian

$$J = \begin{bmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{bmatrix},$$

we can calculate the following:

$$|det(J)| = |(x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0)|$$

$$J^{-T} = \begin{bmatrix} y_2 - y_0 & y_0 - y_1 \\ x_0 - x_2 & x_1 - x_0 \end{bmatrix}$$

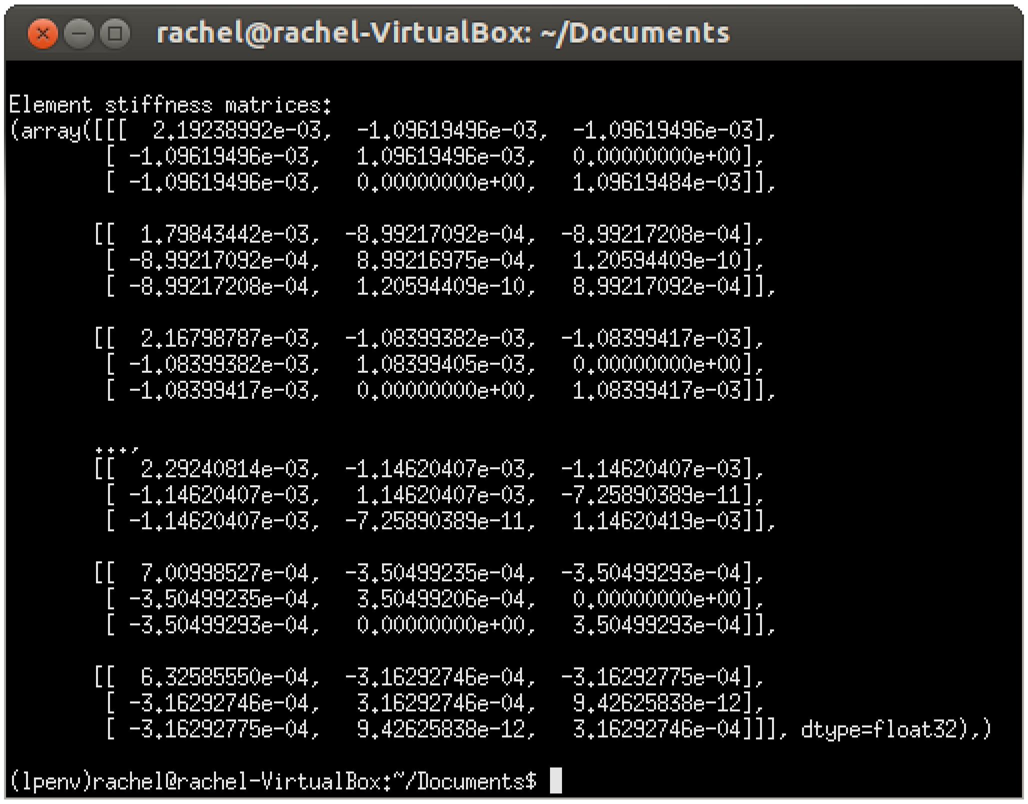
For this project, we have chosen to implement the simple case of linear polynomials on a triangular matrix as our approximating subspace. This means that the basis functions must also be linear so that the gradients of the basis functions are constants and can be pulled out from the integral so that we now have the equation $|det(J)|(J^{-T} \triangledown \gamma_j) \cdot (J^{-T} \triangledown \gamma_i) \int_{T_R} \kappa$ to calculate matrix elements. In the scope of this project under our current construction, $\kappa = 1$ so that each element of element stiffness matrix K can be calculated by $K_{ji} = |det(J)|(J^{-T} \triangledown \gamma_j) \cdot (J^{-T} \triangledown \gamma_i)$ with $j, i = 0, 1, 2$. [3]

Implementation

In order to traverse over each triangle in the mesh, generating a matrix for each triangle, I will thus be using a Loo.py kernel to perform these calculations given the proper bounds to loop over each triangle. Thus, the kernel will have three tiers in the loop structure with the outermost loop traversing over the triangles in the mesh and the two inner loops traversing through all possible i, j pairs to calculate the elements for each element stiffness matrix. First, in order to execute these calculations, I need access to the coordinates to the nodes in the mesh. As mentioned in Chapter Three, these coordinates are stored in the *.node* file and must be inputted into an accessible list that can then be sent as an argument to the Loo.py kernel. This implementation is shown in Figure 16.

```
import loopy as lp
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
import types as ty

loop_b = "{[c, i, j]: 0 <= c < tri and 0 <= i < 3 and 0 <= j < 3}"

kernel_c = """
<> d = fabs((inc[cells[c,1],0] - inc[cells[c,0],0]) * (inc[cells[c,2],1]
     - inc[cells[c,0],1]) - (inc[cells[c,1],1] - inc[cells[c,0],1]) *
   (inc[cells[c,2],0] - inc[cells[c,0],0])) {id=det, inames=c:i:j}
<> J_0 = (inc[cells[c,2],1] - inc[cells[c,0],1]) / d {id=ja, dep=det,
   inames=c:i:j}
<> J_1 = (inc[cells[c,0],1] - inc[cells[c,1],1]) / d {id=jb, dep=det,
   inames=c:i:j}
<> J_2 = (inc[cells[c,0],0] - inc[cells[c,2],0]) / d {id=jc, dep=det,
   inames=c:i:j}
<> J_3 = (inc[cells[c,1],0] - inc[cells[c,0],0]) / d {id=jd, dep=det,
   inames=c:i:j}
<> dqidx = b[i,0]*(inc[cells[c,1],0] - inc[cells[c,0],0]) + b[i,1]*
   (inc[cells[c,1],1] - inc[cells[c,0],1]) {id=qix,   inames=c:i:j}
<> dqidy = b[i,0]*(inc[cells[c,2],0] - inc[cells[c,0],0]) + b[i,1]*
   (inc[cells[c,2],1] - inc[cells[c,0],1]) {id=qiy,   inames=c:i:j}
<> dqjdx = b[j,0]*(inc[cells[c,1],0] - inc[cells[c,0],0]) + b[j,1]*
   (inc[cells[c,1],1] - inc[cells[c,0],1]) {id=qjx,   inames=c:i:j}
<> dqjdy = b[j,0]*(inc[cells[c,2],0] - inc[cells[c,0],0]) + b[j,1]*
   (inc[cells[c,2],1] - inc[cells[c,0],1]) {id=qjy,   inames=c:i:j}
K[c,i,j] = d * ((J_0*dqidx + J_1*dqidy)*(J_0*dqjdx + J_1*dqjdy) +
  (J_2*dqidx + J_3*dqidy)*(J_2*dqjdx + J_3*dqjdy))
  {id=mat, dep=det:ja:jb:jc:jd:qix:qiy:qjx:qjy, inames=c:i:j}
"""



argmt = [lp.ValueArg("tri", np.int32),
        lp.ValueArg("coord", np.int32),
        lp.GlobalArg("cells", np.int32, shape=("tri", 3)),
        lp.GlobalArg("b", np.float32, shape=(3, 2)),
        lp.GlobalArg("inc", np.float32, shape=("coord", 2)),
        lp.GlobalArg("K", np.float32, shape=("tri", 3, 3))]

kn = lp.make_kernel(loop_b, kernel_c, argmt,
                    assumptions="tri>=1")
```

```
construct = lp.CompiledKernel(ctx, kn)

g = open('littlebox.node', 'r')
ind = g.readline()
ind = ind.split()
indices = np.zeros((int(ind[0]), 2), dtype=np.float32)

for y in range(0, int(ind[0])):
    line = g.readline()
    line = line.split()
    indices[y, 0:2] = map(ty.FloatType, line[1:3])

K = np.zeros((int(nels[0]), 3), dtype=np.float32)
u = np.ones((3, 1), dtype=np.float32)
b = np.zeros((3, 2), dtype=np.float32)
b[0][0]=-1;
b[0][1]=-1;
b[1][0]=1;
b[2][1]=1;
q = np.zeros((3, 2), dtype=np.float32)
J = np.zeros((2, 2), dtype=np.float32)

evt, (stiffness_matrix) = construct(queue,
                                    b=b,
                                    cells=cells,
                                    inc=indices,
                                    tri=int(nels[0]),
                                    coord=int(ind[0]),
                                    out_host=False)
```



```
Element stiffness matrices:
(array([[[  2.19238992e-03,  -1.09619496e-03,  -1.09619496e-03],
        [ -1.09619496e-03,   1.09619496e-03,   0.00000000e+00],
        [ -1.09619496e-03,   0.00000000e+00,   1.09619484e-03]],

       [[  1.79843442e-03,  -8.99217092e-04,  -8.99217208e-04],
        [ -8.99217092e-04,   8.99216975e-04,   1.20594409e-10],
        [ -8.99217208e-04,   1.20594409e-10,   8.99217092e-04]],

       [[  2.16798787e-03,  -1.08399382e-03,  -1.08399417e-03],
        [ -1.08399382e-03,   1.08399405e-03,   0.00000000e+00],
        [ -1.08399417e-03,   0.00000000e+00,   1.08399417e-03]],


       ...,
       [[  2.29240814e-03,  -1.14620407e-03,  -1.14620407e-03],
        [ -1.14620407e-03,   1.14620407e-03,  -7.25890389e-11],
        [ -1.14620407e-03,  -7.25890389e-11,   1.14620419e-03]],

       [[  7.00998527e-04,  -3.50499235e-04,  -3.50499293e-04],
        [ -3.50499235e-04,   3.50499206e-04,   0.00000000e+00],
        [ -3.50499293e-04,   0.00000000e+00,   3.50499293e-04]],

       [[  6.32585550e-04,  -3.16292746e-04,  -3.16292775e-04],
        [ -3.16292746e-04,   3.16292746e-04,   9.42625838e-12],
        [ -3.16292775e-04,   9.42625838e-12,   3.16292746e-04]]], dtype=float32),)

(lpenv)rachel@rachel-VirtualBox:~/Documents$
```

Figure 16: Element Stiffness Matrix Implmentation

Application and Conclusion

In our implementation, we now have element stiffness matrices for solving partial differential equations with the Galerkin Finite Element Method over the approximating subspace of linear piecewise polynomials defined on a triangular mesh. This implementation was executed under shared memory parallel computing within the PyOpenCL package to reduce the runtime of execution and explore such computation applied to devices with parallel computing capabilities. With the final result of this project being element stiffness matrices, we have thus established the beginning framework for solving a finite element method in parallel.

From here, the stiffness matrix and load vector can be computed to solve for a best approximation to the true solution as one possible application. Alternatively, the stiffness matrix can be computed iteratively with approximations of the gradient of solution u to converge to the best approximation. From these results and the principles used to arrive at this implementation, parallelism can be similarly applied to various other finite element methods, including methods in higher dimensions. Therefore, with the foundational implementation complete, we are able to observe the principles of parallel computing in actions through our implementation and discover the application of this foundation that can be built upon and applied to various finite element methods.

APPENDICES

APPENDIX A

Example *.ele* and *.node* Files

Example *.ele* File

```
2 3 0

0 0 1 2

1 1 2 3
```

Example *.node* File

```
4 2 0 1

0 0 0 1

1 1 0 1

2 0 1 1

3 1 1 1
```

APPENDIX B

Implementation Source Code

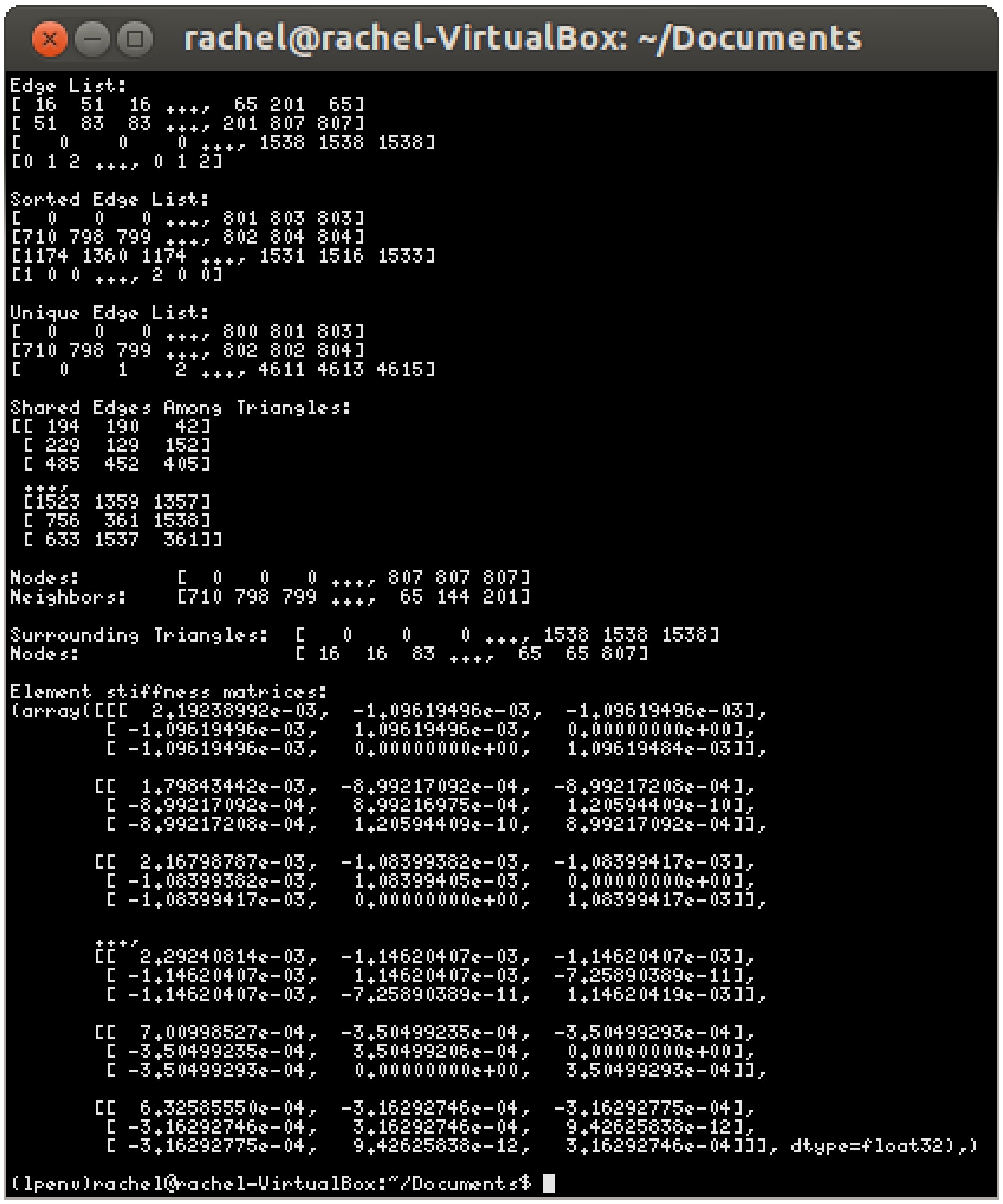
```
import loopy as lp
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
import types as ty
import cmath as mth
from pyopencl.algorithm import RadixSort
from pyopencl.scan import GenericScanKernel
from pyopencl.algorithm import copy_if
from pyopencl.reduction import ReductionKernel
from pyopencl.elementwise import ElementwiseKernel


ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx,
                        properties=
                        cl.command_queue_properties.PROFILING_ENABLE)


loop_bounds = "{[c, i, j]: 0 <= c < nels and 0 <= i < 3 and i < j < 3}"

kernel_code = """
<> ed = 0 {inames=c, id=init_ed}
edges0[c, ed] = cells[c, i] {id=write_v0, dep=init_ed, inames=c:i:j}
edges1[c, ed] = cells[c, j] {id=write_v1, dep=init_ed, inames=c:i:j}
triangle_num[c, ed] = c {id=write_v2, dep=init_ed, inames=c:i:j}
edge_num[c, ed] = j+i-1 {id=write_v3, dep=init_ed, inames=c:i:j}
ed = ed + 1 {dep=write_v0:write_v1:write_v2:write_v3, inames=c:i:j}
"""


args = [lp.ValueArg("nels", np.int32),
        lp.GlobalArg("cells", np.int32, shape=("nels", 3)),
        lp.GlobalArg("edges0", np.int32, shape=("nels", 3)),
        lp.GlobalArg("edges1", np.int32, shape=("nels", 3)),
        lp.GlobalArg("triangle_num", np.int32, shape=("nels", 3)),
        lp.GlobalArg("edge_num", np.int32, shape=("nels", 3))]


knl = lp.make_kernel(loop_bounds,
                     kernel_code,
                     args,
                     assumptions="nels>=1")


cknl = lp.CompiledKernel(ctx, knl)
```

```
loop_b = "{[c, i, j]: 0 <= c < tri and 0 <= i < 3 and 0 <= j < 3}"

kernel_c = """
<> d = fabs((inc[cells[c,1],0] - inc[cells[c,0],0]) * (inc[cells[c,2],1]
    - inc[cells[c,0],1]) - (inc[cells[c,1],1] - inc[cells[c,0],1]) *
   (inc[cells[c,2],0] - inc[cells[c,0],0])) {id=det, inames=c:i:j}
<> J_0 = (inc[cells[c,2],1] - inc[cells[c,0],1]) / d {id=ja, dep=det,
   inames=c:i:j}
<> J_1 = (inc[cells[c,0],1] - inc[cells[c,1],1]) / d {id=jb, dep=det,
   inames=c:i:j}
<> J_2 = (inc[cells[c,0],0] - inc[cells[c,2],0]) / d {id=jc, dep=det,
   inames=c:i:j}
<> J_3 = (inc[cells[c,1],0] - inc[cells[c,0],0]) / d {id=jd, dep=det,
   inames=c:i:j}
<> dqidx = b[i,0]*(inc[cells[c,1],0] - inc[cells[c,0],0]) + b[i,1]*
   (inc[cells[c,1],1] - inc[cells[c,0],1]) {id=qix,   inames=c:i:j}
<> dqidy = b[i,0]*(inc[cells[c,2],0] - inc[cells[c,0],0]) + b[i,1]*
   (inc[cells[c,2],1] - inc[cells[c,0],1]) {id=qiy,   inames=c:i:j}
<> dqjdx = b[j,0]*(inc[cells[c,1],0] - inc[cells[c,0],0]) + b[j,1]*
   (inc[cells[c,1],1] - inc[cells[c,0],1]) {id=qjx,   inames=c:i:j}
<> dqjdy = b[j,0]*(inc[cells[c,2],0] - inc[cells[c,0],0]) + b[j,1]*
   (inc[cells[c,2],1] - inc[cells[c,0],1]) {id=qjy,   inames=c:i:j}
K[c,i,j] = d * ((J_0*dqidx + J_1*dqidy)*(J_0*dqjdx + J_1*dqjdy) +
  (J_2*dqidx + J_3*dqidy)*(J_2*dqjdx + J_3*dqjdy))
  {id=mat, dep=det:ja:jb:jc:jd:qix:qiy:qjx:qjy, inames=c:i:j}
"""

argmt = [lp.ValueArg("tri", np.int32),
         lp.ValueArg("coord", np.int32),
         lp.GlobalArg("cells", np.int32, shape=("tri", 3)),
         lp.GlobalArg("b", np.float32, shape=(3, 2)),
         lp.GlobalArg("inc", np.float32, shape=("coord", 2)),
         lp.GlobalArg("K", np.float32, shape=("tri", 3, 3))]


kn = lp.make_kernel(loop_b,
                    kernel_c,
                    argmt,
                    assumptions="tri>=1")


construct = lp.CompiledKernel(ctx, kn)

sort1 = ElementwiseKernel(ctx, "__global int *x, __global int *y,
                          __global int temp",
                          operation="""if(x[i]>y[i]){
                                       temp = x[i];
                                       x[i] = y[i];
                                       y[i] = temp;}""")

double_edges = ElementwiseKernel(ctx,
                                 """__global int *ed0, __global int *ed1,
                                    __global int *x, __global int *y,
                                    __global int size""",
                                     operation="""if(i<size){
                                                    ed0[i]=x[i];
                                                    ed1[i]=y[i];}
                                                    else{ ed0[i]=y[i-size];
                                                    ed1[i]=x[i-size];}""")

sort_edges = RadixSort(ctx,
                        """__global int *edge1, __global int *edge2,
                           __global int *tri, __global int *side""",
                        key_expr="edge1[i]",
                        sort_arg_names=["edge1", "edge2", "tri", "side"])

unique = ReductionKernel(ctx, np.int32, neutral="0", reduce_expr="a+b",
             map_expr="((i==0)||(x[i]!=x[i-1])||(y[i]!=y[i-1])) ? 1 : 0",
             arguments="""__global int *x, __global int *y""")

remove_duplicates = GenericScanKernel(ctx, np.int32,
                                       """__global int *x, __global int *y,
                                          __global int *final0,
                                          __global int *final1,
                                          __global int *index""",
                                          input_expr="""
                                          ((i==0)||(x[i]!=x[i-1])||
                                           (y[i]!=y[i-1])) ? 1 : 0""",
                                          scan_expr="a+b",
                                          neutral="0",
                                          output_statement="""
                                              if((prev_item!=item))
                                              {final0[item-1] = x[i];
                                               final1[item-1] = y[i];
                                               index[item-1] = i;}""")

triangle_edges = GenericScanKernel(ctx, np.int32,
                                  """__global int *x, __global int *y,
                                     __global int *tri, __global int *side,
                                     __global int *test""",
                                     input_expr="""
                                     ((i==0)||(x[i]!=x[i-1])||
                                      (y[i]!=y[i-1])) ? 1 : 0""",
                                     scan_expr="a+b",
                                     neutral="0",
                                     output_statement="""
                                if((prev_item==item))
                                {test[(3*tri[i-1])+side[i-1]] = tri[i];
                                 test[(3*tri[i])+side[i]] = tri[i-1];}""")

surround = ElementwiseKernel(ctx, "__global int *x, __global int *y,
                                __global int *nhbr",
                                operation="""if(i%3==1){
                                             nhbr[i]=y[i];}
                                             else{nhbr[i]=x[i];}""")
```

```
f = open('littlebox.ele', 'r')
nels = f.readline()
nels = nels.split()
cells = np.zeros((int(nels[0]), 3), dtype=np.int32)

for x in range(0, int(nels[0])):
    line = f.readline()
    line = line.split()
    cells[x, :] = map(int, line[1:])

g = open('littlebox.node', 'r')
ind = g.readline()
ind = ind.split()
indices = np.zeros((int(ind[0]), 2), dtype=np.float32)

for y in range(0, int(ind[0])):
    line = g.readline()
    line = line.split()
    indices[y, 0:2] = map(ty.FloatType, line[1:3])

evt, (edges0, edges1, triangle_num, edge_num) = cknl(queue,
                              nels=len(cells),
                              cells=cells,
                              out_host=False)
temp = 0
sort1(edges0.ravel(), edges1.ravel(), temp)


(e1, e0, t, s),  evt = sort_edges(edges1.ravel(), edges0.ravel(),
                              triangle_num.ravel(), edge_num.ravel())


(e0_srted, e1_srted, tri, side),  evt = sort_edges(e0, e1, t, s)


keys = unique(e0_srted.ravel(), e1_srted.ravel())

num_edges = keys.get()
final0 = np.zeros(num_edges, dtype=np.int32)
final1 = np.zeros(num_edges, dtype=np.int32)
inc = np.zeros(num_edges, dtype=np.int32)
final_0 = cl_array.to_device(queue, final0)
final_1 = cl_array.to_device(queue, final1)
index = cl_array.to_device(queue, inc)

remove_duplicates(e0_srted.ravel(), e1_srted.ravel(), final_0, final_1,
   index);

print "Edge List:"
print edges0.ravel()
print edges1.ravel()
print triangle_num.ravel()
print edge_num.ravel()
print
```

```
print "Sorted Edge List:"
print e0_srted
print e1_srted
print tri
print side
print

print "Unique Edge List:"
print final_0
print final_1
print index
print

#Triangle Shared Edges
test = np.ones([int(nels[0]), 3], dtype=np.int32)
test = test * (-1)
test1 = cl_array.to_device(queue, test)

triangle_edges(e0_srted.ravel(), e1_srted.ravel(), tri, side,
               test1.ravel());
np.reshape(test1, (int(nels[0]), 3))
print "Shared Edges Among Triangles:"
print test1
print

double0 = np.zeros(num_edges*2, dtype=np.int32)
double1 = np.zeros(num_edges*2, dtype=np.int32)
double_0 = cl_array.to_device(queue, double0)
double_1 = cl_array.to_device(queue, double1)
double_edges(double_0, double_1, final_0, final_1, num_edges)

(d1, d0, x, y),  evt = sort_edges(double_1.ravel(), double_0.ravel(),
                              double_0.ravel(), double_0.ravel())

(d0_srted, d1_srted, trash_x, trash_y),  evt = sort_edges(d0, d1, x, y)

print "Nodes:       ", d0_srted
print "Neighbors:   ", d1_srted
print

nhbr = np.zeros(int(nels[0])*3, dtype=np.int32)
neighbor = cl_array.to_device(queue, nhbr)

surround(edges0.ravel(), edges1.ravel(), neighbor)

(n, tnum, x, y),  evt = sort_edges(neighbor, triangle_num.ravel(),
                                   neighbor, neighbor)

(t_nhbr, e_nhbr, trash_x, trash_y),  evt = sort_edges(tnum, n, x, y)

print "Surrounding Triangles: ", t_nhbr
print "Nodes:                 ", e_nhbr
print
```

```
K = np.zeros((int(nels[0]), 3), dtype=np.float32)
u = np.ones((3, 1), dtype=np.float32)
b = np.zeros((3, 2), dtype=np.float32)
b[0][0]=-1;
b[0][1]=-1;
b[1][0]=1;
b[2][1]=1;
q = np.zeros((3, 2), dtype=np.float32)
J = np.zeros((2, 2), dtype=np.float32)

evt, (stiffness_matrix) = construct(queue,
                             b=b,
                             cells=cells,
                             inc=indices,
                             tri=int(nels[0]),
                             coord=int(ind[0]),
                             out_host=False)

print "Element stiffness matrices:"
print stiffness_matrix
```


rachel@rachel-VirtualBox: ~/Documents

```
Edge List:
[ 16  51  16 ...,  65 201  65]
[ 51  83  83 ..., 201 807 807]
[   0    0    0 ..., 1538 1538 1538]
[0 1 2 ..., 0 1 2]

Sorted Edge List:
[  0   0   0 ..., 801 803 803]
[710 798 799 ..., 802 804 804]
[1174 1360 1174 ..., 1531 1516 1533]
[1 0 0 ..., 2 0 0]

Unique Edge List:
[  0   0   0 ..., 800 801 803]
[710 798 799 ..., 802 802 804]
[   0    1    2 ..., 4611 4613 4615]

Shared Edges Among Triangles:
[[ 194  190   42]
 [ 229  129  152]
 [ 485  452  405]
 ...,
 [1523 1359 1357]
 [ 756  361 1538]
 [ 633 1537  361]]

Nodes:         [  0   0   0 ..., 807 807 807]
Neighbors:     [710 798 799 ...,  65 144 201]

Surrounding Triangles:  [   0    0    0 ..., 1538 1538 1538]
Nodes:                  [ 16  16  83 ...,  65  65 807]

Element stiffness matrices:
(array([[[  2.19238992e-03,  -1.09619496e-03,  -1.09619496e-03],
        [ -1.09619496e-03,   1.09619496e-03,   0.00000000e+00],
        [ -1.09619496e-03,   0.00000000e+00,   1.09619484e-03]],

       [[  1.79843442e-03,  -8.99217092e-04,  -8.99217208e-04],
        [ -8.99217092e-04,   8.99216975e-04,   1.20594409e-10],
        [ -8.99217208e-04,   1.20594409e-10,   8.99217092e-04]],

       [[  2.16798787e-03,  -1.08399382e-03,  -1.08399417e-03],
        [ -1.08399382e-03,   1.08399405e-03,   0.00000000e+00],
        [ -1.08399417e-03,   0.00000000e+00,   1.08399417e-03]],

       ...,
       [[  2.29240814e-03,  -1.14620407e-03,  -1.14620407e-03],
        [ -1.14620407e-03,   1.14620407e-03,  -7.25890389e-11],
        [ -1.14620407e-03,  -7.25890389e-11,   1.14620419e-03]],

       [[  7.00998527e-04,  -3.50499235e-04,  -3.50499293e-04],
        [ -3.50499235e-04,   3.50499206e-04,   0.00000000e+00],
        [ -3.50499293e-04,   0.00000000e+00,   3.50499293e-04]],

       [[  6.32585550e-04,  -3.16292746e-04,  -3.16292775e-04],
        [ -3.16292746e-04,   3.16292746e-04,   9.42625838e-12],
        [ -3.16292775e-04,   9.42625838e-12,   3.16292746e-04]]], dtype=float32),)

(lpenv)rachel@rachel-VirtualBox:~/Documents$
```

BIBLIOGRAPHY

[1] G. E. Blelloch. Prefix sums and their applications.

[2] L. Howes Gaster, B. and D. R. Kaeli. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2012.

[3] M. S. Gockenbach. *Understanding and Implementing the Finite Element Method*. Society for Industrial and Applied Mathematics, 2006.

[4] G. Hillar. *Easy OpenCL with Python*. Dr. Dobb's, UBM Tech, October 2013.

[5] Khronos Group. *OpenCL Documentation*, 2.0 edition, 2014.

[6] A. Klöckner. Loo.py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Lan guages, and Compilers for Array Programming*, Edinburgh, Scotland., 2014. As sociation for Computing Machinery.

[7] A. Klöckner. *PyOpenCL Documentation*, 2013.2 edition, 2009.

[8] A. Logg. *Efficient Representation of Computational Meshes*, volume 4, pages 283–295. 4 edition, 2001.

[9] T. Rauber and G. Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer-Verlag, Berlin, 2 edition, 2010.

[10] M. Scarpino. *A Gentle Introduction to OpenCL*. Dr. Dobb's, UBM Tech, August 2011.

[11] J. R. Shewchuk. *Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*, 1.6 edition, July 2005.