Abstract:

Graphics Processing Units (GPUs) have the potential to achieve large performance improvements over CPUs for certain numerical computing applications. To write software that achieves these large performance improvements on real-world applications the developer must have considerable experience programming GPUs. We have developed a prototype software named JavaAutoCuda that enables programmers to write in a serial fashion using Java and automatically obtain often quite significant speedups with GPUs for certain numerical computing applications. JavaAutoCuda is unique in that it requires no work to use other than writing Java Source Code in a serial manner. In addition, no other related work translates Java Bytecode to use GPUs. Java Bytecode is a garbage collected intermediate form that over 40 programming languages use as their compiled form. Using this intermediate form opens up the possibility to support any language that compiles to Java Bytecode. In a simple test of writing a matrix multiplication program using our method the programmer development time was one third that of using the native GPU language CUDA C. The performance of the result using our method was up to 29X faster than a serial C version. However, using CUDA C the GPU application obtained a 70X speedup, signifying that eventually more advanced optimizations can be fruitfully applied.

AUTOMATICALLY UTILIZING GRAPHICS PROCESSING UNITS FROM JAVA BYTECODE

By

Philip C. Pratt-Szeliga
B.S. Rensselaer Polytechnic Institute, 2005

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering in the Graduate School of Syracuse University

May 2010

Approved_____
Professor James Fawcett

Date_____

TABLE OF CONTENTS

LIST OF ILLUSTRATIVE MATERIALS

TABLES

FIGURES

CHAPTER 1. INTRODUCTION

Researchers in applied sciences now commonly depend on computational analysis to obtain their results. For instance, a computational biologist may create a protein folding application to gain insight into the ways different proteins interact with one another. A protein folding application, along with many other applied science applications, takes a very long time to compute a result without using parallel processing. However, a researcher's first draft of an application will most likely be written without parallelism. This is because it takes additional skill and effort to manually program an application to use parallelism.

There are several ways to get parallelism from hardware today. There are multi-core CPUs, multi-core Graphics Processing Units (GPU) and multiple computers containing multi-core processors. The focus of this work is specifically using GPUs for parallelism. The GPU has a specialized architecture that has allowed many researchers to get phenomenal speedups at a low hardware cost. For instance, a radar signal processing application (space time adaptive processing) has achieved a 140X speedup over a serial CPU version [14]. This speedup is quite extraordinary, but there is a catch to using GPUs: The programmer skill and development time required to convert the original serial version of the application to use GPUs is typically too prohibitive for an applied scientist to work alone. Instead, the

applied scientist must collaborate with a computer scientist who will manually analyze the serial program and convert it a parallel program that uses the GPU.

The author has played the role of the computer scientist collaborating with an applied physicist to convert a serial medical imaging application to a parallel version that uses the GPU. The physicists hoped for a 50X speedup. With a 50X speedup the lab's improved medical imaging algorithms would undergo clinical trials. While the author was a fairly experienced programmer, he faced difficulty with this problem. Beginning the project he knew nothing about the use of GPUs and also nothing about the physics behind the medical imaging application. The author worked for five months trying to convert the 9000 lines of code into something that would achieve a 50X speedup using a GPU. The result was a 4.7X slowdown of the original serial application. The author temporarily gave up because he didn't have the tools needed to effectively work on this problem. Massive code restructuring was required to get a speedup and maintaining program correctness after the restructuring without tools was too complex and time consuming. Other researchers have had this problem as well. For instance, a computational biology researcher converted a serial application to a parallel GPU application and obtained a 60X slowdown [10]. After optimizing the parallel application a final result of a 7X slowdown was achieved.

The solution presented in this thesis is to automatically analyze serial code for places that would achieve a speedup on a GPU and automatically restructure the code to obtain that speedup. Other work has been done in this area. The significant difference from the other work is that with this work the application developer can program in the high level, Java

programming language in a serial fashion and obtain speedups possible with GPUs without any additional manual analysis. Most of the other related projects require the end user to program in older languages such as Fortran or C or languages that are not general purpose such as MATLAB. There is one other related project that requires the end user to program in Python, a high level general purpose programming language, but annotations of types are required to overcome the face that Python is an untyped language. The solution presented in this thesis resulted in a 29X speedup in a matrix multiplication test case. Future work will try to expand the number of user programs that can be parallelized.

Section 1.1 Overview of this Research

To gain insight into automatically transforming a serial Java program into a parallel Java program that uses the GPU, prototype software named JavaAutoCuda has been developed. Rather than analyzing Java Source Code, the intermediate representation named Java Bytecode is analyzed. This was done in hopes of supporting all of the programming languages that compile to Java Bytecode.

From a very high level, JavaAutoCuda does a static analysis to find possible loops that would obtain a speedup if they were run on the GPU. Then the static analyzer manipulates the input application so that, if at runtime it is determined there would be a speedup running on the GPU, execution is on the GPU. Otherwise, execution is on the CPU.

SECTION 1.2 SUMMARY OF RELATED WORK

There are seven other related systems that try to make it easier to use GPUs. These are Jacket, the PGI Accelerator Compiler, hiCUDA, HMPP, OpenMP to GPGPU, the unnamed Python project and CUDA-lite. What follows is a summary of these related works. A more extended review of the related work is located in Appendix A.

All of these works except CUDA-lite target making GPU programming easier by automatically generating the GPU program (called a kernel) from a CPU program and the required CPU code to get the data to and from the GPU. CUDA-lite targets transforming an existing GPU kernel into a new GPU kernel with optimized memory access patterns that will get performance enhancements with Nvidia brand GPUs. Below is a table specifying the input languages of each related project.

Related Project	**Input Language**
Jacket	MATLAB with Jacket library
PGI Accelerator Compiler	C or Fortran with custom annotations
hiCUDA	C with custom annotations
HMPP	C with custom annotations
OpenMP to GPGPU	C with OpenMP annotations
CUDA-lite	CUDA C with custom annotations
Python Project	Python with custom annotations and type annotations
This Research	Java

Table 1 – Input languages of related work and this research

It can be seen from table 1 above that most of the systems do not on a general purpose, garbage collected language such as C# or Java. This thesis describes the effort to support

the Java Programming Language in conjunction with GPUs. Additionally, each of these related works requires some sort of manual analysis beyond that of simply writing a serial application. JavaAutoCuda (this work) requires no additional manual analysis. The closest competitor, the Python project does operate on a general purpose language, but the user must compensate for the fact that Python is an untyped language by adding annotations. The degree of manual analysis that each related project requires is discussed in detail Appendix A and a summary table is listed below.

Related Project	**Relative Effort Required to Utilize GPU**
Jacket	Low
PGI Accelerator Compiler	Low
hiCUDA	High
HMPP	Medium
OpenMP to GPGPU	Medium
CUDA-lite	High
Python Project	Medium
This research	Very Low

Table 2 – Effort Required to Utilize GPU for each Related Project and this research

Several of the related projects apply optimizations that are more advanced than what the author of this work has currently done. The focus of this work was to support garbage collected languages so the largest contributions are in that area and not in optimizations. Below is a table that outlines the optimizations in each of the related projects.

Related Project	**Optimizations**
Jacket	- Proprietary project with unknown optimizations
PGI Accelerator Compiler	- Manual: Sharing data between GPU kernel launches - Automatic: Loop nesting level selection - Automatic: Array region analysis to reduce memory transfers

hiCUDA	- Manual: Any manual CUDA optimization can be applied - Automatic: Non-perfect distribution of iteration indexes supported
HMPP	- Manual: Sharing data between GPU kernel launches - Automatic: Compatible with MPI to support multiple computers
OpenMP to GPGPU	- Manual: Optimal caching requires manual assistance - Automatic: Parallel loop swap - Automatic: Loop collapsing - Automatic: Memory transfer reduction of results in shared memory
CUDA-lite	- Manual: Optimal coalescing requires manual assistance currently but may be possible to do automatically
Python Project	- Automatic: Loop unrolling - Automatic: Loop fusion (limited) - Automatic: Load coalescing (partial)
This research	- Automatic: High performance (de)serialization of Java objects to/from GPU memory - Automatic: Determination of what loops to run on the GPU that incorporates divergent execution

Table 3 – Optimizations of the Related Projects and this research

This research includes high performance (de)serialization of Java objects to/from GPU memory. It also pays attention to divergent execution (discussed in Section 3.1.2) when choosing loops to parallelize.

CHAPTER 2. BACKGROUND INFORMATION

In this chapter the architecture of the GPU is discussed along with GPU programming examples. Then Java Bytecode is explained and the Soot Java Optimization Framework is introduced. Finally, JCuda, the library used to access the CUDA runtime from Java is discussed.

The architecture of the GPU will be discussed specifically in terms of Nvidia brand GPUs. The other major vendor of GPUs is AMD. The Nvidia brand has been selected because they were the first to have a C-like programming language to program the GPU and our lab only has this brand GPU.

SECTION 2.1 ARCHITECTURE OF THE GPU

Nvidia GPUs are organized as sets of SIMD processing elements. Each processing element has 8 ALUs that all share instruction fetch hardware. Code is executed by grouping 32 threads together that all share instruction fetch hardware. This has implications where the user code must have groups of 32 threads executing in lock-step otherwise the threads will become serialized. The term for threads not executing in lock-step is divergent execution.

Example code with performance measurements demonstrating the penalties associated with divergent execution on a GPU are in Section 2.1.2. Nvidia GPUs also have a large global memory that is originally filled by the CPU before beginning the GPU calculation. The GPU global memory fetch is slow, so to achieve enhanced performance over a CPU the GPU must run thousands of threads concurrently to hide the latency associated with this global memory fetch. The memory is arranged in banks that favor coalesced memory access which is defined and discussed in Section 2.1.3.

The Nvidia GPUs also contain a small, fast 16KB shared memory that can be used to store frequently used global memory items. The memory is called shared memory because it is shared among all threads within a thread-block. An analysis of the performance improvement while using shared memory is found in Section 2.1.4. Along with the shared memory there is texture memory that is read-only global memory with a hardware cache.

The programming interface for Nvidia GPUs is named CUDA. In CUDA functions called kernels are written that are the entry point to the GPU calculation. When writing a kernel, and any function a kernel calls, a developer needs to keep in mind that the code will be executed many times in parallel. In the kernel function the user accesses specialized thread index registers that identify the current thread. The kernel function then uses the thread index to fetch data from global memory. This ensures each kernel invocation operates on a different input data set.

In CUDA there are barrier synchronization primitives that allow every thread on the device to become synchronized. This only works if every thread on the device executes the barrier synchronization instruction, otherwise there is a deadlock. In general, this work tries to avoid using barrier synchronization primitives because it requires more analysis to determine if deadlocks will occur. In addition to barriers the newer GPUs contain a set of atomic instructions to execute useful operations such as atomic exchange, add, subtract and compare and swap. Also, there is a "thread fence block" primitive that the author does use to make sure all threads within a block can see previous writes to shared memory before continuing execution.

At runtime the number of data elements to process is chosen by picking launch parameters. The launch parameters specify how many blocks and threads to run. The total number of data items is the number of blocks times the number of threads. Choosing the number of blocks and threads is a delicate matter. To achieve a performance enhancement the kernel launch must have thousands of iterations while at the same time not so many iterations that the GPU runs out of resources.

SECTION 2.1.1 GPU COMPUTATIONAL CAPABILITIES

The Nvidia Tesla C1060 GPU can reach 933 GFLOPS peak performance in single-precision floating point calculations. Comparatively, the fastest six-core CPU (Intel Core i7 980 X3) can reach 108 GFLOPS (single-precision) [17]. The C1060 is arranged as 240 cores with

groups of 8 sharing instruction fetch hardware. The actual performance is limited by divergent execution and memory bandwidth. The optimal actual memory bandwidth of 102GB/sec [12] is achieved when memory accesses are coalesced. If shared memory is used the GPU can perform at speeds beyond those restricted by memory bandwidth. These issues are discussed in detail in Sections 2.1.2 through 2.1.4.

SECTION 2.1.2 DIVERGENT EXECUTION

Divergent execution is when two GPU threads within the same thread-block need to fetch different instructions at a particular time. For instance, if there is an if statement whose condition is dependent on the thread id, one thread will execute the body of the if and another will not. If this happens execution has diverged. The related works in Appendix A do not focus on methods that specifically work to reduce the amount of divergent execution occurring on the GPU, but in the author's experience he has found it is a problem that needs to be addressed. To demonstrate the problems, two programs that have the same behavior are listed in figures 1 and 2. The program in figure 1 has more divergent execution than the program in figure 2 and runs at half the speed. The speed is cut in half because the program diverges into two execution paths. In a complex program there is the possibility for more divergent paths and a larger performance hit.

```
1  __global__ void large_diverge(int * x, int * y, int * z, int iters) {
2    int tid = threadIdx.x;      //the thread identifier
3    if(tid % 2 == 0){
4      z[tid] += 1;
5      for(int i = 0; i < num_iterations; i++){
6         z[tid] += x[i] + y[i];
```

```
7       }
8     } else if(tid % 2 == 1){
9       z[tid] += 2;
10      for(int i = 0; i < num_iterations; i++){
11        z[tid] += x[i] + y[i];
12      }
13    }
14    __syncthreads();
15  }
```

Figure 1 - Kernel With Lots of Divergent Execution

Figure 1 above contains two for loops where each is inside an if statement whose condition is dependent on the thread identifier. The only thing different between the two bodies of the "if" and "else if" is that a global variable is incremented by a different value. Since the for loops are the same, they can be pulled outside of the "if" and "else if" and divergent execution will be reduced. This modified program is shown in figure 2 below.

```
1   __global__ void small_diverge(int * x, int * y, int * z, int iters) {
2     int tid = threadIdx.x;            //the thread identifier
3     if(tid % 2 == 0){
4       z[tid] += 1;
5     } else if(tid % 2 == 1){
6       z[tid] += 2;
7     }
8     for(int i = 0; i < num_iterations; i++){
9       z[tid] += x[i] + y[i];
10    }
11    __syncthreads();
12  }
```

Figure 2 - Kernel With Less Divergent Execution

As shown below in table 4 the execution time of the kernel with large divergence is 2X that of the kernel with small divergence. The performance measurements were executed on a computer with a Tesla C1060 GPU. The number of iterations used when collecting the measurements was 4096.

Kernel	Execution Time
large_diverge	78 seconds
small_diverge	39 seconds

Table 4 - Execution Time Of Kernels With Different Divergent Execution

SECTION 2.1.3 MEMORY ACCESS COALESCING

Memory access coalescing occurs when the GPU can make one request to the global memory to fetch or set a number of memory locations. This occurs on the Nvidia GPUs when each thread accesses a memory location that is exactly n larger than the thread memory location access before it, where n is the size of the data value and limited to sizes of 4 and 8. An example of a kernel without coalesced reads and writes is in figure 4 and a coalesced version is found in figure 3. The kernels do not have the same exact behavior but they do have the same exact number of operations and memory accesses. Non-coalescing behavior for figure 4 was ensured by placing random numbers in the variable y.

```
1  __global__ void coalesced(int * x, int * y, int * z, int iters) {
2    int tid = threadIdx.x;      //the thread identifier
3    for(int i = 0; i < iters; ++i){
4      int y_tid = y[tid];
5      z[tid] += x[tid % blockDim.x] * y_tid + i;
6    }
7    __syncthreads();
8  }
```

Figure 3 – Kernel with Coalesced Reads and Writes

```
1  __global__ void not_coalesced(int * x, int * y, int * z, int iters) {
2    int tid = threadIdx.x;            //the thread identifier
3    for(int i = 0; i < iters; ++i){
4      int y_tid = y[tid];
5      z[tid] += x[y_tid % blockDim.x] * y_tid + i;
```

```
6     }
7     __syncthreads();
8   }
```

Figure 4 – Kernel without Coalesced Reads and Writes

The performance comparison of the two kernels is listed in table 5 below. With coalescing the kernel is 6X faster. The performance measurements where run on a computer with a Tesla C870 running 4096 iterations.

Kernel	Execution Time
coalesced	30 seconds
not_coalesced	194 seconds

Table 5 – Performance of Coalesced and Not Coalesced Kernels

SECTION 2.1.4 SHARED MEMORY PERFORMANCE

Storing commonly used data in the fast shared memory can boost performance of a GPU kernel past the global memory bandwidth barrier. Two versions of a matrix multiplication application are listed in Section 2.1.5. Figure 6 is a kernel without using shared memory and figure 7 is a kernel using shared memory. The use of shared memory dramatically increases performance in this case, as shown in table 6 below. Performance measurements were taken on a computer with a Tesla C870 GPU. The size variable was set to 4096.

Kernel	Execution Time
gpu_mult	189 seconds
gpu_mult2 (shared memory version)	1 second

Table 6 – Performance of Global Memory Only and Shared Memory Kernels

SECTION 2.1.5 PROGRAMMING GPUS WITH CUDA

A simple CUDA Matrix Multiplication example is shown below. The host code that runs on the CPU is shown in figure 5 and the device code that runs on the GPU is shown in figure 6.

In the host code from lines 3 though 5 memory is allocated on the GPU that will be used to store the CPU input data matrices x and y and the result z from the device. On lines 7 through 8 the CPU x and y are copied to the device d_x and d_y. On line 9 the gpu_mult kernel is called with 64 blocks and 64 threads and passing the device memory and size as arguments. On line 10 the result from the device is copied to the GPU and at line 11 the GPU memory is freed.

```
1   void mult(int * x, int * y, int * z, int size){
2     int * d_x; int * d_y; int * d_z;
3     cudaMalloc((void**)&d_x, sizeof(int)*size*size);
4     cudaMalloc((void**)&d_y, sizeof(int)*size*size);
5     cudaMalloc((void**)&d_z, sizeof(int)*size*size);
6
7     cudaMemcpy(d_x, x, sizeof(int)*size*size,cudaMemcpyHostToDevice);
8     cudaMemcpy(d_y, y, sizeof(int)*size*size,cudaMemcpyHostToDevice);
9     gpu_mult<<<64, 64>>>(d_x, d_y, d_z, size);
10    cudaMemcpy(z, d_z, sizeof(int)*size*size,cudaMemcpyDeviceToHost);
11    cudaFree(d_x); cudaFree(d_y);  cudaFree(d_z);
12  }
```

Figure 5 – Matrix Multiplication Host Code

In the gpu_mult kernel, below, the inner two loops of matrix multiplication can be clearly seen starting at lines 3 and 5. The outer loop is implicitly executed because there are 64 blocks and 64 threads running. At line 2 the index for the outer loop is obtained by

multiplying the block index (blockIdx.x) by the number of threads (blockDim.x) and adding the thread index (threadIdx.x).

```
1   __global__ void gpu_mult(int * x, int * y, int * z, int size){
2     int i = blockIdx.x * blockDim.x + threadIdx.x;
3     for(int j = 0; j < size; ++j){
4       int total = 0;
5       for(int k = 0; k < size; ++k){
6         total += x[k*size+j] * y[i*size+k];
7       }
8       z[i*size+j] = total;
9     }
10  }
```

Figure 6 – Matrix Multiplication Device Code

An optimization for the gpu_mult kernel is to use shared memory as a cache of array elements that will be shared between threads. This is shown in Figure 7 below.

```
1   __global__ void gpu_mult2(int * x, int * y, int * z, int size){
2     int i = blockIdx.x * blockDim.x + threadIdx.x;
3     for(int j = 0; j < size / BLOCK_SIZE; ++j){
4       __shared__ int Xs[BLOCK_SIZE*BLOCK_SIZE];
5       __shared__ int Ys[BLOCK_SIZE*BLOCK_SIZE];
6
7       Xs[i*size+j] = x[i*size+j];
8       Ys[i*size+j] = y[i*size+j];
9
10      __syncthreads();
11
12      int total = 0;
13      for(int k = 0; k < BLOCK_SIZE; ++k){
14        total += Xs[k*size+j] * Ys[i*size+k];
15      }
16
17      __syncthreads();
18      z[i*size+j] += total;  //assumes z has been initialized to zeros
19    }
20  }
```

Figure 7 – Matrix Multiplication Device Code Using Shared Memory

SECTION 2.1.6 OTHER EXISTING WAYS TO PROGRAM GPUS

There exists ways to program GPUs other than this work and the related work. There are low level methods that require the programmer to know at least two programming languages: one for the CPU and one for the GPU. These low level methods include OpenGL, ATI Stream and OpenCL. Then there are methods to access GPU programming languages and runtimes from high-level languages. These methods are JCuda, CUDA.NET, PyCuda, JavaCL and ScalaCL. JCuda is used from the JavaAutoCuda runtime to interact with the CUDA runtime.

Higher level numerical computing support includes the following optimized libraries for CUDA: CUBLAS, CUFFT and CUDPP. CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) that is GPU accelerated. CUFFT supports FFT (Fast Fourier Transform) and CUDPP is the CUDA Data Parallel Primitives Library. CUDPP supports parallel radix sorting, parallel compacting of arrays (removing zeros), parallel scan and parallel sparse matrix-vector multiply.

Lastly, there is Mars, a MapReduce framework for graphics processors [11]. MapReduce is a parallel programming methodology that requires the programmer to determine parallel tasks and phrase them as a map function and phrase the merging of parallel tasks at their completion with a reduce function.

SECTION 2.2 JAVA BYTECODE

Java Bytecode is a stack based intermediate code that gets interpreted or compiled down to assembly by the Java Virtual Machine. The three places data can exist are in the stack, in method locals and in object fields. There is a garbage collected memory model where all objects are accessed through references.

When choosing the language to analyze and manipulate the author wanted a general purpose, statically typed, garbage collected language. Also, platform independence is important so C# was eliminated from the list primarily because of the problems with Mono[1], notably the debugger. Java was chosen then and the choice was between analyzing Java Source Code and Java Bytecode. Java Bytecode was chosen because many languages compile to Bytecode and someone may not have access to the original source code for their application. From Wikipedia the most well-known languages that compile to Java Bytecode are: ColdFusion, Clojure, Groovy, JavaFX script, Jruby, Jython, Rhino, Scala and Java [19]. There are a total of 47 lesser known languages listed on that same page that compile to Java Bytecode. There is also a large number of Java Virtual Machines, 50 are listed on [18]. Note that a large number of languages also compile to CLI (C#'s intermediate form) but an open source, freely available, framework for manipulating CLI in Single Static Assignment form was not found when researching possible libraries to depend on. Therefore manipulation of Java Bytecode was chosen.

[1] Mono is an open source implementation of the C# runtime and compiler for Linux

SECTION 2.3 SOOT

Soot is a framework written in Java by the Sable research group from McGill University [16]. Soot allows a developer to analyze, generate and manipulate Java Bytecode. Soot provides four intermediate representations: Baf, Jimple, Shimple and Grimp. JavaAutoCuda takes as input Shimple code. Shimple is a typed 3-address intermediate representation with the SSA property. SSA stands for Single Static Assignment and makes some transformations easier. After JavaAutoCuda transforms the Shimple code, the code no longer has the SSA property so it is more like Jimple than Shimple. Jimple is a typed 3-address intermediate representation without the SSA property. Soot uses the algorithms from Ron Cytron's paper "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph" [6].

To give the reader a feel for Soot, some common operations with Soot are listed along with sample Java code. There are two categories of example operations: transformations outside of methods and transformations inside of methods.

SECTION 2.3.1 SOOT TRANSFORMATIONS OUTSIDE OF METHODS

The Soot transformations outside of methods that will be discussed are: 1) Creating a new class, 2) Creating a new method and 3) Making a private field public.

To create a new class that will be accessible to subsequent transformations the Java Source code in figure 8 could be used. Note that the version of Soot publicly available online will not allow you to do this while inside a Soot transformation pack due to the use of iterators in Soot. I desired this feature for my transformation so the feature was added to Soot. The new version of Soot can be downloaded from a location listed in Appendix B.

```
1    SootClass new_class = new SootClass(“className”, Modifier.PUBLIC);
2    SootClass parent = Scene.v().getSootClass(“java.lang.Object”);
3    new_class.setSuperClass(parent);
4    new_class.setApplicationClass();
5    Scene.v().addClass(new_class);
```

Figure 8 – Java Source Code to Create a Class

To create a new method that accepts an integer and returns an integer the Java Source Code in figure 9 could be used.

```
1    List<Type> input_types = new ArrayList<Type>();
2    input_types.add(IntType.v());
3    SootMethod method = new SootMethod(“methodName”, input_types,
       IntType.v(), Modifier.PUBLIC);
4    SootClass enclosingClass = Scene.v().getSootClass(“className”);
5    method.setDeclaringClass(enclosingClass);
6    Body body = Jimple.v().newBody(method);
7    method.setActiveBody(body);
8    enclosingClass.addMethod(method);
9    //now add code to body
```

Figure 9 – Java Source Code to Create a Method

Finally, it is useful have code that makes a private field public using Soot. My transformations destroy encapsulation that developers have created by making needed fields public. Java Source Code to make a field public is listed in figure 10 below.

```
1     void makePublic(SootField field){
2       int modifiers = field.getModifiers();
3       modifiers &= ~Modifier.PRIVATE;
4       modifiers &= ~Modifier.PROTECTED;
5       modifiers |= Modifier.PUBLIC;
6       field.setModifiers(modifiers);
7     }
```

Figure 10 – Java Source Code to Make a Field Public

SECTION 2.3.2 SOOT TRANSFORMATIONS INSIDE OF METHODS

The example Soot transformations occurring inside of methods are 1) calling another method, 2) changing the target of an If Statement, 3) getting a reference to this, 4) creating a new object and 5) reading a field. All of these transformations involve adding Units to a Jimple or Shimple Body.

The example Java Source Code found in figure 11 below can be used to call a method that expects a java.lang.Object as a parameter and returns an integer. The example works regardless if the method is a constructor, a virtual function or an interface function. Invoking static functions is slightly different because an instance is not required therefore static functions are not covered with invokeMethod. Currently the prototype translator JavaAutoCuda does not allow static method invocations to occur on the GPU because time did not permit the supporting CUDA C code to be written.

```
1   Local invokeMethod(Local instance, List<Value> args){
2     SootClass soot_class = Scene.v().getSootClass("className");
3     List<Type> input_types = new ArrayList<Type>();
4     input_types.add(RefType.v("java.lang.Object"));
5     Type ret_type = IntType.v();
```

```
6     SootMethod method = soot_class.getMethod(“methodName”,
7       input_types, ret_type);
8     Value invoke_expr;
9     if(method.getName().equals(“<init>”)){
10      invoke_expr = Jimple.v().newSpecialInvokeExpr(instance,
11        method.makeRef(), args);
12    } else if(method.isConcrete()) {
13      invoke_expr = Jimple.v().newVirtualInvokeExpr(instance,
14        method.makeRef(), args);
15    } else {
16      invoke_expr = Jimple.v(). newInterfaceInvokeExpr(instance,
17        method.makeRef(), args);
18    }
19    Local ret = Jimple.v().newLocal(“unique name”,
20      method.getReturnType());
21    Unit u = Jimple.newAssignStmt(ret, invoke_expr);
22    //add unit to Body
23    return ret;
```

Figure 11 – Java Source Code to Call Another Method

Getting a reference to this is a simple operation. The new Soot interface introduced in this example is the Value interface. Values and Units are the basic types that most Units contain. Important examples of where Values are used are Locals, the arguments to a binary expression, the condition of an If Statement and the left hand side of an assignment.

```
1   public Local refThis(){
2     String name = "this0";
3     SootClass curr_class = getCurrentSootClassOfTransformation();
4     RefType type = curr_class.getType();
5     Local thislocal = Jimple.v().newLocal(name, type);
6     Value thisref = Jimple.v().newThisRef(type);
7     Unit u = Jimple.v().newIdentityStmt(thislocal, thisref);
8     //add u to Body
9     return thislocal;
10  }
```

Figure 12 – Java Source Code to Reference This

Creating a new object involves creating a New Expression and calling the constructor of the class. This is shown in figure 13 below.

```
1   public Local newInstance(String className, List<Value> params) {
2     SootClass soot_class = Scene.v().getSootClass(className);
3     Local u1_lhs = Jimple.v().newLocal("localName",
4       soot_class.getType());
5     Value u1_rhs = Jimple.v().newNewExpr(soot_class.getType());
6     Unit u1 = Jimple.v().newAssignStmt(u1_lhs, u1_rhs);
7     //add unit to body
8     List<Type> arg_types = new ArrayList<Type>();
9     for(int i = 0; i < params.size(); ++i){
10      arg_types.add(params.get(i).getType());
11    }
12
13    //invoke ctor with arg_types and params
14    return u1_lhs;
15  }
```

Figure 13 – Java Source Code to Create a New Instance

Lastly, reading a field is shown in figure 14 below. Note that getFieldByName at line 4 has to search the class hierarchy to find the SootField in the declaring SootClass.

```
1   public Local refInstanceField(Local base, String field_name){
2     Type base_type = base.getType();
3     SootClass base_class= Scene.v().getSootClass(base_type.toString());
4     SootField field = getFieldByName(base_class, field_name);
5     Local ret = Jimple.v().newLocal(getLocalName(), field.getType());
6
7     Value rhs = Jimple.v().newInstanceFieldRef(base, field.makeRef());
8     Unit u = Jimple.v().newAssignStmt(ret, rhs);
9     //add unit to body
10    return ret;
11  }
```

Figure 14 – Java Source Code to Read a Field

SECTION 2.4 JCUDA

JCuda provides bindings to the CUDA runtime from within Java. The bindings are used in the JavaAutoCuda runtime to copy memory to/from the GPU and to launch GPU kernels. A

graphical representation of how the JavaAutoCuda runtime is built on top of JCuda is displayed below.

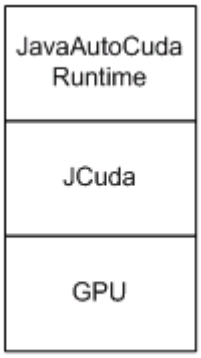

Figure 15 - JCuda's Relation to JavaAutoCuda

Originally, OpenCL C was the target language for GPU code generation. OpenCL provides functions in the runtime to compile OpenCL C code. CUDA and JCuda do not provide such functions so a process that executes Nvidia's compiler, nvcc, is created when JavaAutoCuda needs to compile CUDA code. An example of using the runtime is shown in figure 16 below.

```
1   CUcontext context = new CUcontext();
2   CUdevice device = new CUdevice();
3   JCudaDriver.cuDeviceGet(device, DEVICE_INDEX);
4   JCudaDriver.cuCtxCreate(context, 0, device);
5   CUmodule module = new CUmodule;
6   JCudaDriver.cuModuleLoad(module, "filename of compiled module");
7   CUfunction function = new CUfunction();
8   JCudaDriver.cuModuleGetFunction(function, module,
    "_Z5entryPcS_S_S_PiS0_iii"));
9
10  int size = 1024;
11  CUdeviceptr gpu_mem = new CUdeviceptr();
12  JCudaDriver.cuMemAlloc(gpu_mem, size);
13
14  byte[] host_mem = new byte[size];
15  //set values in host_mem
16
17  JCudaDriver.cuMemcpyHtoD(gpu_mem, Pointer.to(host_mem), Sizeof.INT);
18
19  JCudaDriver.cuParamSetv(kernel, 0, Pointer.to(gpu_mem),
```

```
20  Sizeof.POINTER);
21  JCudaDriver.cuParamSetSize(kernel, Sizeof.POINTER);
22
23  JCudaDriver.cuFuncSetBlockShape(function, num_threads, 1, 1));
24  JCudaDriver.cuLaunchGrid(function, num_blocks, 1));
25  JCudaDriver.cuCtxSynchronize();
26
27  JCudaDriver.cuMemcpyDtoH(Pointer.to(host_mem), gpu_mem, Sizeof.INT);
28
29  //free gpu resources
```

Figure 16 - Java Source Code to Run a GPU Job

JavaAutoCuda caches the CUmodules and CUfunctions in a Map where the key is the Concrete LoopBody Java class. This caching saves time when running the same loop twice in a program. Note that when running line 24 (executing the GPU job), one core of the CPU has 100% processor usage. This is detrimental when trying to combine multi-core CPU execution with the GPU execution. Future work will aim to fix this.

CHAPTER 3. ARCHITECTURE OF THE SOLUTION

JavaAutoCuda is implemented as a Java application that analyzes Java Bytecode. It depends on the Soot Java Optimization Framework to convert the stack based Java Bytecode representation of the end user program into a Single Static Assignment representation named Shimple. Then the contributed transformations are applied to Shimple code in memory. Soot then converts the transformed Shimple back into Java Bytecode. This is shown in Figure 17 below.

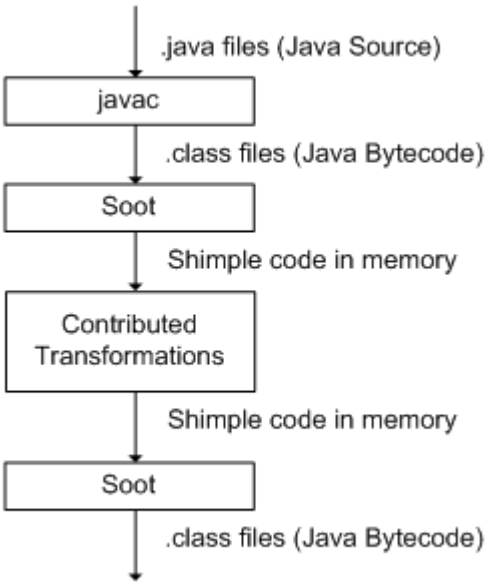

Figure 17 – High Level Transformation Flow of JavaAutoCuda

SECTION 3.1 DETAILED TRANSFORMATION FLOW

In the next figure our contributed transformations are shown. The diagram shows that first loops are detected, then the loops are analyzed to see if they are suitable for running on the GPU. For each loop that is suitable for the GPU CUDA C code is generated and a concrete GcObjectVisitor class is created to (de)serialize Java objects to/from the GPU. These are tied together with a Concrete LoopBody that can be enqueued to the JavaAutoCuda runtime for processing. After all of this is done the original loop is modified to use the Concrete LoopBody.

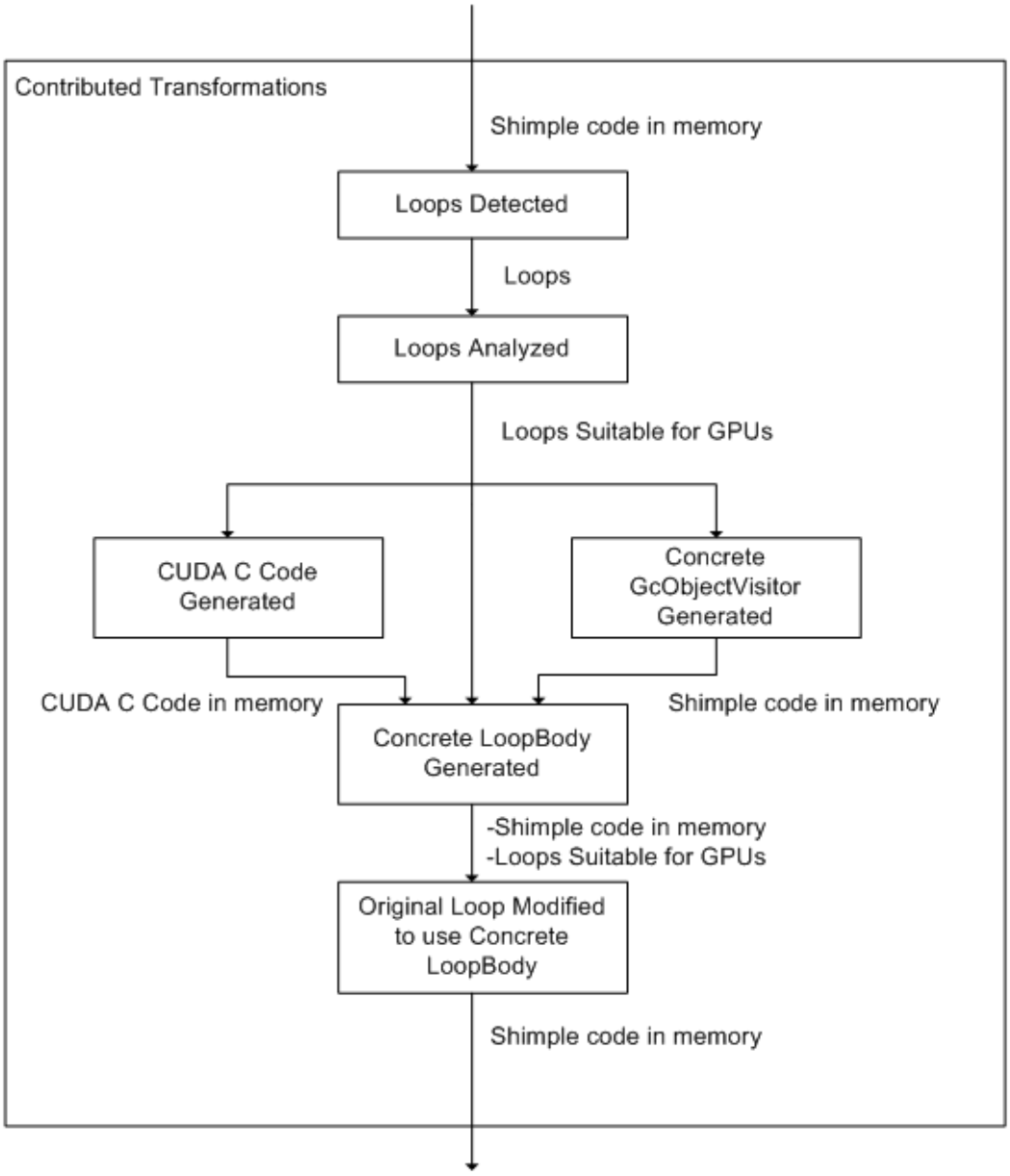

Figure 18 – Contributed Transformations

In figure 19 below the UML class diagram of the transformed class is displayed. This will help the reader in understanding the structure of the resulting code after the transformations in sections 3.1.1 through 3.1.6 are applied. The Concrete LoopBody and

Concrete GcObjectVisitor classes are generated entirely from scratch while the transformed_method is modified, possibly multiple times if multiple loops exist in series and match the policy. Everything else was developed ahead of time in Java and then combined with the results of the transformation.

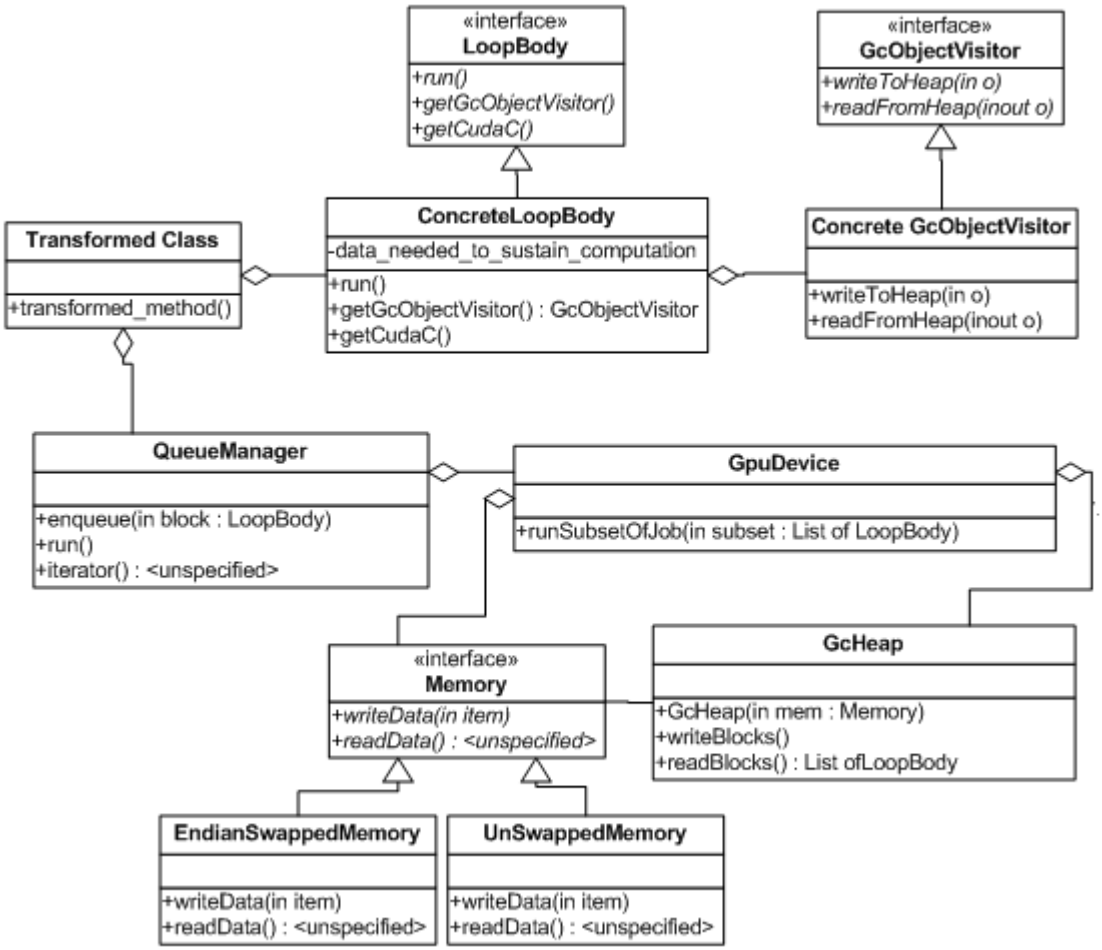

Figure 19 – UML Class Diagram of Transformed Class

The remainder of this chapter discusses each processing step from figure 18 in detail.

SECTION 3.1.1 LOOP DETECTION

The loop detection algorithm is in figure 20 below. This first step is very simple: if there is a statement in a method that points to a statement whose line number is above the original statement, a loop has been detected.

```
1  for each (Shimple Unit) {
2    if (the unit is a goto or an if) {
3      if(the pointed to unit has a line number above the goto) {
4        a loop is located between the two units
5      }
6    }
7  }
```

Figure 20 - Loop Detection Algorithm

To exemplify the steps to parallelize a Java program to use the GPU, a vector addition program will be used as input. This program is written in Java Source in figure 21 below. For simplicity it has been assumed that the lengths of the input arrays are the same. The end result of the translation is that when the translated program runs, the body of the loop (line 4) will be automatically executed on the GPU. Only the data needed to sustain that computation will be automatically copied to the GPU memory and only the data that has changed (the result ret) will be automatically copied from the GPU memory back to the CPU memory.

```
1  public int[] add(int[] x, int[] y){
2    int[] ret = new int[x.length];
3    for(int i = 0; i < x.length; ++i) {
4      ret[i] = x[i]+y[i];
5    }
6    return ret;
7  }
```

Figure 21 - Java Source Input

In figure 22 below the Shimple representation provided by Soot is displayed that corresponds to the Java Source code of figure 21 There are no fors or whiles to represent loops in Shimple, only ifs and gotos as is common in assembly like languages. Shimple is easier to analyze than assembly like programs because it contains the names of the types and methods involved in the computation. For instance, at line 5, a reference to this is made and stored in the local named r0. You can tell that r0 is of type VectorAddExample because at line 2 the local is declared as such and at line 5 the @this contains the type. Also note the Phi expression at line 12, this is used to specify that the local variable i0_1 gets its value from two different control paths (#0 (line 10) and #1 (line 20)).

```
1   public int[] add(int[], int[]) {
2     VectorAddExample r0;
3     int[] r1, r2, r3;
4     int i0, $i1, $i2, $i3, $i4, $i5, i0_1, i0_2;
5     r0 := @this: VectorAddExample;
6     r1 := @parameter0: int[];
7     r2 := @parameter1: int[];
8     $i1 = lengthof r1;
9     r3 = newarray (int)[$i1];
10    i0 = 0; (0)
11  label0:
12    i0_1 = Phi(i0 #0, i0_2 #1);
13    $i2 = lengthof r1;
14    if i0_1 >= $i2 goto label1;
15    $i3 = r1[i0_1];
16    $i4 = r2[i0_1];
17    $i5 = $i3 + $i4;
18    r3[i0_1] = $i5;
19    i0_2 = i0_1 + 1;
20    goto label0; (1)
21  label1:
22    return r3;
23  }
```

Figure 22 - Shimple Code Input

In the Shimple Code in figure 22 there is a loop from lines 12 to 20 that corresponds roughly to lines 3 to 5 of the Java Source Input. The next algorithm is Loop Analysis that will categorize that loop.

SECTION 3.1.2 LOOP ANALYSIS

The loop analysis algorithm takes every loop and determines if it is suitable for the GPU. The algorithm is shown in figure 23 below. A Unit is a Soot interface that represents a line of Shimple code.

```
1     for every Shimple Unit in each loop {
2       if the unit is an if or goto that points outside of the Loop {
3         the loop is not suitable for the GPU
4       }
5       if the unit is an assignment and depends on a unit below it {
6         if the value is not a loop control line {
7           the loop is not suitable for the GPU
8         }
9       }
10      if the unit calls a method in the Java class libraries {
11        the loop is not suitable for the GPU
12      }
13      if the unit uses an unsupported Java feature {
14        the loop is not suitable for the GPU
15      }
16      if the unit can possibly be called recursively {
17        the loop is not suitable for the GPU
18      }
19    }
```

Figure 23 – Loop Analysis Algorithm

The Shimple Code Input in figure 22 passes all of the requirements of being suitable for the GPU so the loop will be passed further along the processing chain. After loop analysis the next step is Concrete GcObjectVisitor Generation.

SECTION 3.1.3 CONCRETE GCOBJECTVISITOR GENERATION

Concrete GcObjectVisitor Generation enables high performance conversion of Java objects into GPU memory and back. The Gc in the name stands for Garbage Collected and the Visitor corresponds to the Visitor design pattern. The algorithm below is run separately on each loop. An example output of the GcObjectVisitor generation algorithm is located in Appendix D, figure 49.

```
1     create a ConcreteGcObjectVisitor class that is uniquely named
2     find every class involved with the loop
3     generate the start of a writeToHeap and a readFromHeap method
4     for every class involved with the loop
5       find the fields that are possibly used in the loop
6       determine the read/write properties of the field on the GPU
7       add the following code to the writeToHeap method:
8         "if the input object is not an instance of the current class
          goto label:next"
9            "cast the input object to the current class"
10           "write the object header"
11           "write all fields read on the GPU to the GPU memory"
12        "label:next"
13      add the following code to the readFromHeap method:
14        "if the input object is not an instance of the current class
          goto label:next"
15           "cast the input object to the current class"
16           "read all fields written on the GPU from the GPU memory"
17        "label:next"
```

Figure 24 – Concrete GcObjectVisitor Generation

For simplicity the code that ensures that data is aligned on proper byte boundaries has been omitted above. (On Nvidia GPUs integers have to be aligned on 4 byte boundaries, shorts on 2 byte boundaries and so on.). Also note that the AbstractGcObjectVisitor provides caching of previously visited objects so graphs of Java objects are supported. Lastly, to support the GcObjectVisitor a Memory interface was created that has two derived classes. One class has methods to write basic types to the GPU memory that does an endian

swap and the other does not do a swap. The correct class is chosen at runtime by the GpuRuntime depending on whether the GPU is big endian or little (Java is always big). This method optimizes out the if statements required to determine whether to swap or not that would have been needed otherwise.

SECTION 3.1.4 CUDA C CODE GENERATION

Cuda C code is generated by implementing a concrete Visitor[2] that can operate on each element of Shimple code. There are no fors and whiles in Shimple code but there are ifs and gotos. There are also ifs and gotos in CUDA C so loops that need to be represented on the GPU are written as ifs and gotos for simplicity. Field accesses are translated into getter and setter functions where the field address is computed from an object reference. Java references are kept as references in the GPU memory. An example of generated CUDA C code is located in Appendix D, figure 50.

SECTION 3.1.5 CONCRETE LOOPBODY GENERATION

A concrete LoopBody is generated for each loop that is suitable for the GPU. The three interface functions that are implemented through code generation are: run, getGcObjectVisitor and getCudaC. The runtime uses these functions to run jobs. A

[2] The Visitor Design Pattern is a design pattern that can ensure that every type of element in a tree is processed

constructor is also added that inputs the values of all data needed to sustain the loop body and saves them to fields that were also generated.

The run function is implemented by copying the body of the loop and setting up the inputs and outputs of the loop to correspond to generated concrete LoopBody fields. The getGcObjectVisitor function is implemented by returning an instance of the ConcreteGcObjectVisitor. The getCudaC function is implemented by returning a string that represents the compiled generated CUDA C code. An example of a generated Concrete LoopBody is found in Appendix D, figure 48.

SECTION 3.1.6 MODIFICATION OF ORIGINAL LOOP

Currently two forms of modification of the original loop have been implemented. The first method is the simplest method and just picks the outer-most for loop of series of nested loops and transforms that. The second method can use multiple nested loop levels to extract out more parallelism for the GPU than the simplest method. The third method, which has not been implemented yet, is to extend the second method to operate on loops that call methods that have loops inside. The result of transforming the input Java source in figure 25 is shown below in Java source. Note that it is shown in Java source to benefit the reader, the transformation result is actually Shimple code.

```
1  public int[] add(int[] x, int[] y){
2    int[] ret = new int[x.length];
3    QueueManager manager = QueueManager.instance();
4    for(int i = 0; i < x.length; ++i) {
```

```
5       LoopBody1 block = new LoopBody1(i, ret);
6       manager.enqueue(block);
7     }
8     manager.run();
9     Iterator<LoopBody> iter = manager.iterator();
10    while(iter.hasNext()){
11      LoopBody1 curr = (LoopBody1) iter.next();
12      ret = curr.ret;
13      //run loop body that is iter-loop dependent.Here there is no
14      //code
15    }
16    return ret;
17  }
```

Figure 25 – Simple Original Loop Transformation

The nice aspect of enqueueing blocks in this manner is that the number of iterations is determined when manager.run is called so Java iterators that may have side-effects are supported. When manager.run is called the runtime chooses to either run the LoopBody1 run function that was populated with the body of the loop or use the LoopBody1 getCudaC and getGcObjectVisitor functions to run the body of the loop on the GPU. Then, from lines 10 to 14, the results are iterated to support loops that have inter-loop dependencies at the bottom of the loop. Note that the original loop index is saved because if this were an iterator, it would have to be only been evaluated once so the application of side-effects would be correct.

CHAPTER 4. CONTRIBUTIONS

JavaAutoCuda is the first translator that the author is aware of that automatically enables GPU usage within Java programs. No additional manual analysis is required with JavaAutoCuda, unlike the related translators and compilers. The translator is also unique in that it operates on Java Bytecode, opening up the possibility to transform any programming language that compiles to Java Bytecode. High performance (de)serialization of Java objects to/from GPU memory is accomplished through the use of Java Bytecode code generation. To support the iterator design pattern commonly found in Java Bytecode loops, special attention was paid to the method used to detect the number of iterations a loop has. This method properly handles the cases where the iterator has side-effects by ensuring that the "hasNext" and "next" functions of the iterator interface are executed exactly the same number of times as in the original program. Since the number of iterations a loop has cannot always be determined during static analysis, a run time determination of the number has been outlined and tested. According to a policy variable, if the number of iterations is too low to obtain a speedup while using the GPU, the CPU is instead used. The remainder of this chapter explains each contribution discussed above in detail as well as supporting contributions that enable the ones listed above.

Section 4.1 Loop Detection and Loop Categorization of Java Bytecode

As discussed in Section 3.1.1, loops are detected in Java Bytecode. Once loops are detected they are categorized according to two aspects. These aspects are: 1) inter-loop data dependency and 2) divergent execution amount.

Inter-loop data dependency is detected by simply checking if an assigned value in the body of the loop depends on an assigned value from a unit later on in the loop. The data flow analysis required to implement this algorithm is very simple when operating on Single Static Assignment. Currently, the translations work on loops that have zero inter-loop dependent units. We are working to eliminate that constraint by splitting loops into a top and bottom body if there is an inter-loop independent top. The top bodies can be executed in parallel and the bottoms can then be executed serially. Additionally, the current state of JavaAutoCuda does not support detecting inter-loop dependencies while dealing with arrays. It simply can detect that a loop does not have inter-loop dependencies when only the indexes exactly equal to the control index of the loop is used. It is part of the future work to use Region Array SSA form [15] to solve this problem and support more numerical computing applications.

Divergent execution amount is represented as a score in the translator named DivergentScore. There is code in place that will filter out loops whose score is greater than a policy number named MaxDivergentScore. In future work, the MaxDivergentScore value

will be determined empirically by running many numerical computing loops both on the CPU and GPU with a known DivergentScore and measuring the time for completion of both. The DivergentScore is calculated by counting the number of units that are in the body of an If Statement whose condition is dependent on the loop control. Since divergent execution only serializes threads within a warp (a group of 32 threads) future work will calculate a DivergentScore that incorporates thread clustering.

Determining the loop control(s) has not yet been discussed and is required to calculate the DivergentScore. The loop control is determined by analyzing the variables in the condition of the If Statement of a loop. If a variable in the condition of an If Statement has its value changed during the body of the loop, it is marked as a control of the loop.

SECTION 4.2 A LOOP TRANSFORMATION THAT ENABLES DELEGATION OF WORK TO THE GPU

The first type of loop transformation was described in Section 3.1. This was the simplest case. The second type of loop transformation breaks up loops that may or may not be in methods. An original pseudo-code program is below and subsequent to that is the transformation of that program.

Original Program:

```
1  public class example {
2    public int entryFunction(){
3      int total = 0;
4      for(int i = 0; i < 4096; ++i){
5        total += functionWithPartialLoopDependency();
```

```
6       }
7       return total;
8     }
9
10    private int functionWithPartialLoopDependency(){
11      //loop without dependency
12      int[] totals = new int[4096*4096];
13      for(int j = 0; j < 4096; ++i){
14        for(int k = 0; k < 4096; ++k){
15          int curr_total = computation();
16          totals[j*4096+k] = curr_total;
17        }
18      }
19      int ret = 0;
20      for(int i = 0; i < totals.length; ++i){
21        ret += totals[i];
22      }
23      return ret;
24    }
25  }
```

Figure 26 – Original Complex Loop to be Modified

Transformed Program:

```
1   public class example {
2     public void entryFunction(){
3       int total = 0;
4       QueueManager manager = QueueManager.v();
5       for(int i = 0; i < 4096; ++i){
6         ConcLoopBody block = new ConcLoopBody(i);
7         block.setIterationFlag(0, true);
8         functionWithPartialLoopDependencyTop(manager, block);
9       }
10      manager.run();
11      PeekIterator<RuntimeBasicBloc> iter = manager.iterator();
12      while(iter.hasNext()){
13         ConcLoopBody block = (ConcLoopBody) iter.next();
14         if(iter.hasNextNext()){
15            ConcLoopBody next_next_block = (ConcLoopBody)
    iter.nextNext();
16            if(next_next_block.getIterationFlag(1)){
17              total += functionWithPartialLoopDependencyBottom(block);
18            }
19         }
20      }
21      return total;
22    }
23
24    private void functionWithPartialLoopDependencyTop(QueueManager
    manager, ConcLoopBody block){
25      int i = block.i;
26      //loop without dependency
27      int[] totals = new int[4096*4096];
```

```
28      for(int j = 0; j < 4096; ++i){
29        block.setIterationFlag(1, true);
30        for(int k = 0; k < 4096; ++k){
31          block.addj(j);
32          block.addk(k);
33          manager.enqueue(block);
34          block.setIterationFlag(1, false);
35        }
36        block.setItertionFlag(0, false);
37      }
38    }
39
40    private int functionWithPartialLoopDependencyBottom(ConcLoopBody
    block){
41      int[] totals = block.totals;
42      int ret = 0;
43      for(int i = 0; i < totals.length; ++i){
44        ret += totals[i];
45      }
46      return ret;
47    }
48  }
```

Figure 27 – Modified Complex Loop

With this transformation small sections of loop independent code are found and put in parallel and the dependent code is properly executed after the parallel execution. The author has not implemented this yet but he believes this method of loop finding is sufficient to accelerate real-world examples such as medical imaging.

SECTION 4.5 CHOOSING, AT RUNTIME, WHETHER TO USE THE CPU OR THE GPU FOR A LOOP EXECUTION

At runtime, during the iteration of a loop, the body of that loop is not executed directly. Instead, a Concrete LoopBody that represents the body and contains all the data needed to sustain the computation of that body is placed in a queue. Then, when the loop has

finished, all of the LoopBody objects are available in the queue. The QueueManager's run function is then called. If the QueueManager finds that there are a sufficient number of LoopBody objects to obtain a speedup, they are run all at the same time on the GPU. Otherwise, they are run on the CPU.

SECTION 4.6 TRANSLATING JAVA BYTECODE TO THE EQUIVALENT CUDA CODE

Translating Java Bytecode to the equivalent CUDA code requires determining each Java Class that can possibly be executed from the current loop. For each Java Class, each possibly executed method is found and from each method, each possibly accessed Java Field is found. Once all of this information is aggregated translation begins.

To support polymorphism, each reference type (including array types) that can exist on the GPU is assigned a unique number named the Type Identifier. The Type Identifier is stored in the object's memory (in GPU global memory). Now, for each virtual method that needs to be executed polymorphicly, a wrapper function is generated that accepts an instance of the object and the arguments to the method. Inside the wrapper function a switch statement is used to select the correct concrete function based on the stored Type Identifier. Note that using a Type Identifier was required because the CUDA specification disallows function pointers that could have been used to form a virtual function pointer table.

To support accessing fields, for each field that is found, accessor and mutator functions are generated. The functions accept a handle to the object. When JavaAutoCuda has determined that the current loop being processed requires a garbage collector, the handle is dereferenced using the handle map. Otherwise the handle corresponds directly to a location in global memory. Once the location of the object in global memory is determined a precomputed offset is applied to the location and proper casting is applied to return or mutate values of the proper type. Note that the computed offsets serve the same purpose as C structures, but computing offsets is more portable when using both Nvidia and AMD GPUs. Supporting AMD GPUs is a task of future work that needs to wait until all of the OpenCL compilers are production quality.

The body of each function is then generated by implementing Concrete StmtSwitch and JimpleValueSwitch classes. The StmtSwitch and JimpleValueSwitch are Soot interfaces that are the Visitors of the Visitor Design Pattern. A StmtSwitch visits Units in a JimpleBody and a JimpleValueSwitch visits Values in a JimpleBody. For each Element the Visitors cover, code is generated that has the same semantics in CUDA C. A special StmtSwitch is implemented to cover constructors because the JavaAutoCuda GPU Garbage Collected Object Header (discussed in Section 4.7) needs to be written to the object's memory.

Care is taken during the entire code generation to first output all the declarations of functions and next output all of the implementations. Additionally, existing CUDA C support code and a kernel entry point is combined with the generated code during code assembly.

SECTION 4.7 HIGH PERFORMANCE JAVA OBJECT (DE)SERIALIZATION TO AND FROM GPU MEMORY

To accomplish high performance Java object serialization and deserialization, first every Java Class that will be used from a loop is detected at static analysis time. Then it is determined what fields in each class are read from and what fields are written to. A concrete doWriteToHeap method is then generated that writes the garbage collected object header first and also writes each GPU read field. This is done for each type of object using the Visitor Design Pattern. Care is taken so that the memory locations of GPU write only fields are skipped during writing. Next a doReadFromHeap method is generated that reads only the fields that were modified on the GPU. If an object was created on the GPU (as determined by the CreatedOnGpu Garbaged Collected Object header Field) a constructor is called that accepts a sentinel value. Sentinel accepting constructors were generated at static code generation time for every class hierarchy that was found to be used on the GPU. The remainder of this section discusses Garbage Collected Object Header and the mechanics of writing objects to the GPU garbage collected heap.

The Garbage Collected Object Header is 8 bytes long and shown in table 7 below. All of these values are used by the garbage collector and discussed in Section 4.9.

Field	Size
GC_COUNT	1 byte
GC_COLOR	1 byte

GC_TYPE	1 byte
GC_CTOR_USED	1 byte
GC_SIZE	4 bytes

Table 7 – Garbage Collected Object Header Layout

To help write to the Garbage Collected Heap a Memory interface has been created that has a memory address stack. This is useful when writing reference types. Example code to write is shown below. Note that the garbage collector expects garbage collected reference fields to be at the beginning of the memory after the header and the non reference fields at the end.

```
1     int doWriteToHeap(Object o){
2       if(o instanceof LoopBody){
3         LoopBody b = (LoopBody) o;
4         int ret = heap.getHeapEndPtr();
5         mem.writeByte(gc_count);
6         mem.writeByte(0);
7         mem.writeByte(class_id);
8         mem.writeByte(0);
9         mem.writeInt(size);
10
11        heap.incrementHeapEndPtr(size);
12        mem.pushAddress();
13        mem.incrementAddress(size_minus_gc_info_size);
14        int field1_address = writeRefField(b.field1);
15        int top_address = mem.topAddress();
16        mem.popAddress();
17        mem.writeInt(field1_address);
18        //write non ref fields
19        mem.setAddress(top_address);
20      }
21    }
```

Figure 28 – Java Code that will be Generated to Write a LoopBody

Originally the swapped and unswapped Memory implementations allocated a very large byte array that was written to and then copied to the GPU in one transfer. This was not optimal because the GPU often has more available memory that can be allocated than the

CPUs and in the CPU memory this method effectively creates two copies of the objects involved. The problem was that even though the GPU had free memory, the CPU would run out of memory. The solution is to only allocate a small buffer and when that buffer gets full, copy the memory to the correct base address on the GPU. Implementing this without changing the doWriteToHeap code generation requires that a copy does not occur while there are addresses pushed on the address stack, because in this case the address will be reverted to a smaller value and the buffer should not be copied yet.

Section 4.8 Discussion of Failures

Several aspects of JavaAutoCuda failed on the first attempt or never successfully obtained a speedup. This section is devoted to explaining the problems that were encountered. The optimizations that never obtained a speedup are: 1) using shared memory to create a general purpose cache and 2) creating a GPU garbage collector. The following methods failed on the first attempt but were improved: 1) Java object (de)serialization and 2) Using OpenCL as the target GPU language. The Java object (de)serialization originally obtained correct results, but the performance was very slow.

Section 4.8.1 General Purpose GPU Cache Using Shared Memory

The first aspect that failed was a software defined cache using shared memory. The idea is to write software that uses the fast shared memory as a cache of the slow global memory. The first problem with this is that writable memory cannot be cached when there is no

global communication that allows updates to a dirty flag. To solve this only read only memory was cached. Note this is also the solution Nvidia came to with their cached read-only texture memory. The second problem with a software cache is that it doesn't combine with a moving garbage collector in a simple way. If the value at an address is constant memory, but gets overridden by a move during a garbage collect, it no longer appears constant. The solution to this is basing the cache address on references rather than absolute memory. The last problem with the software defined cache was that there is very little shared memory relative to the amount of concurrency involved in large speedups.

With all of the problems associated with the shared memory cache, there is a possibility that it can be used to obtain speedups on problems that don't exhibit massive parallelism. Because of this the performance results and algorithms of four types of caches are presented in the next sections.

SECTION 4.8.1.1 CACHE WITH READER'S WRITER'S LOCK

The cache with a reader's writer's lock allows multiple readers to read from the cache but only one writer to write to the cache at any instant. Since this is a cache, if a writer cannot enter the write section, it simply does not write to the cache. The cache requires five integers total for every two integers in the cache. The code for this cache is longer than the other caches so it is listed in Appendix C. The performance measurements for this cache

and the remaining caches in Section 4.8.1 can be found in table 8 and 9. This cache performed the worst.

SECTION 4.8.1.2 CACHE USING IDEAS FROM SOFTWARE TRANSACTIONAL MEMORY

Software transactional memory is an alternative to lock-based synchronization for controlling access to a shared memory [20]. Instead of requiring the reader to lock during a read, a version is kept for each variable. After the reader is done with all of the reads in a transaction, it checks to make sure all the versions are the same. If they are the same then the data is consistent. Otherwise the data needs to be re-read. In a cache design, rather than trying to re-read a variable in the event of an inconsistency, global memory is queried. The best STM-like cache algorithm created uses 4 integers of storage for every 4 or 8 bytes of data. There is a lock memory location for the writers, an address memory location that also acts as the version and two data locations. The algorithm written in CUDA C is below.

```
1   int cache_get_int_stm(int address){
2     int ptr = address % (CACHE_SIZE_INTS / (CACHE_ENTRY_SIZE));
3     ptr *= CACHE_ENTRY_SIZE;
4
5     int value;
6     int cache_addr = cache[ptr+ADDR_LOC];
7     if(cache_addr == address){
8       value = cache[ptr+DATA_LOC];
9       cache_addr = cache[ptr+ADDR_LOC];
10      if(cache_addr != address){
11        value = *((int *) &global_memory[address]);
12      }
13    } else {
14      value = *((int *) &global_memory[address]);
15
16      if(cache[ptr+LOCK_LOC] == 0){
17        int prev_value = atomicExch(&cache[ptr+LOCK_LOC], 1);
18        if(prev_value == 0){
```

```
19          cache[ptr+ADDR_LOC] = -1;
20          __threadfence_block();
21          cache[ptr+DATA_LOC] = value;
22          __threadfence_block();
23          cache[ptr+ADDR_LOC] = address;
24          cache[ptr+LOCK_LOC] = 0;
25        }
26      }
27    }
28    return value;
29  }
```

Figure 29 – STM-like cache CUDA C code

The performance of this cache was the best out of all the caches and the measurements can be found in tables 8 and 9.

SECTION 4.8.1.3 SEGMENTED MEMORY CACHE

The segmented memory cache organizes the shared memory so that each thread has its own small cache that is not shared with any other threads. This removes all locking requirements but also does not allow threads to share data. Two integers are required to store one integer in the cache with this method. The algorithm written in CUDA C is below.

```
1   int cache_get_int_seg(int address){
2     int ptr = address % CONSTANT_KEEPING_THREADS_DISTINCT;
3     ptr += (threadIdx.x * (SIZE_OF_CACHE_ENTRY *
                  CONSTANT_KEEPING_THREADS_DISTINCT));
4     if(cache[ptr+ADDR_LOC] == address){
5       return cache[ptr+DATA_LOC];
6     } else {
7       int value = *((int *) &global_memory[address]);
8       cache[ptr+ADDR_LOC] = address;
9       cache[ptr+DATA_LOC] = value;
10      return value;
11    }
12  }
```

Figure 30 – Segmented Memory Cache CUDA code

The performance measurements of the Segmented Memory Cache can be found in tables 8 and 9.

SECTION 4.8.1.4 CACHE WITH SIMPLE MUTEX

The cache with a simple mutex uses classical locks to ensure that only one thread is reading or writing at a single instant of time. This cache requires four integers for every two integers stored.

```
1   int cache_get_int_simple(int address){
2     int ptr = address % (CACHE_SIZE_INTS / (CACHE_ENTRY_SIZE));
3     ptr *= CACHE_ENTRY_SIZE;
4
5     int prev_lock = atomicExch(cache[ptr+LOCK_LOC], 1);
6     if(prev_lock == 1){
7       return *((int *) &global_memory[address]);
8     }
9
10    int value;
11    if(cache[ptr+ADDR_LOC] == address){
12      value = cache[ptr+DATA_LOC];
13    } else {
14      value = *((int *) &global_memory[address]);
15      cache[ptr+ADDR_LOC] = address;
16      cache[ptr+DATA_LOC] = value;
17    }
18
19    cache[ptr+0] = 0;
20    return value;
21  }
```

Figure 31 – Cache with Simple Mutex

See tables 8 and 9 for the performance results of the Cache with a Simple Mutex.

SECTION 4.8.1.5 CACHE PERFORMANCE RESULTS

The execution time of a 1024x1024 matrix multiplication has been measured for each cache and the results are in tables 8 and 9 below. Table 8 uses a 15360 byte cache while table 9 uses a 512 byte cache.

Cache Type	Execution Time
No cache	2.6 seconds
STM	7.2 seconds
Segmented Memory	14.3 seconds
Simple Mutex	54 seconds
Reader's Writer's Lock	146 seconds

Table 8 – Performance Results of Various Cache Algorithms with 15360 byte cache

Cache Type	Execution Time
No cache	2.6 seconds
STM	3.8 seconds
Segmented Memory	11 seconds
Simple Mutex	42.1 seconds
Reader's Writer's Lock	125 seconds

Table 9 – Performance Results of Various Cache Algorithms with 512 byte cache

Overall, it can be seen from tables 8 and 9 that using less shared memory for the cache improved performance. This is most likely due to the fact that when each thread requests more cache memory the GPU thread scheduler cannot schedule as many threads at once. The cache replacement algorithm is dependent on the ordering of thread execution, so it is also possible that the speedup is due to Belady's anomaly [3].

The STM cache has more possible locking that the Segmented Memory cache yet it achieves better performance. This is probably because the Segmented Memory has a smaller cache per thread and items in the cache cannot be shared between threads.

The Simple Mutex Cache requires a lock to read and write the data so it's performance is worse than the STM and the Reader's Writer's Lock Cache requires complex locking that has too much overhead.

While none of the caches obtained a speedup over not caching for the matrix multiplication program, it is useful to know that the STM cache was the fastest. In future work the performance of programs that have less parallelism than matrix multiplication will be studied while using an STM cache to see if using a large cache size can hide latency to global memory rather using than a large number of threads.

SECTION 4.8.2 GPU GARBAGE COLLECTOR

The second aspect that has currently failed is the GPU garbage collector. It has been started, but not yet completed due to the difficulty associated with debugging CUDA C code. The CUDA C code and the memory layout used on the GPU can readily be incorporated with a GPU garbage collector. The CUDA C code ensures that every reference assignment goes through a gc_assign function so the garbage collector can track it. Once the parallel garbage collector is finished, more possible code can be scheduled on the GPU.

Section 4.8.3 Java Object Serialization Using Reflection

The third aspect that failed was Java object to GPU memory conversion using reflection. This failed because reflection was heavily used in tight loops in a performance critical section of processing and Java reflection is slow compared to generated Java Bytecode. The solution was to generate custom Java Bytecode that could replace the reflection operations in the specific case of object serialization to GPU memory.

Section 4.8.4 Using OpenCL C as the Target GPU Language

The fourth aspect that failed was the generation of OpenCL C rather than CUDA C. OpenCL is a new open standard for GPU programming supported by several vendors. Usage of OpenCL C failed because the current OpenCL compilers are too buggy to implement a large computer generated GPU program. While trying to use the OpenCL compiler from Nvidia internal errors would occur often. At other times incorrect GPU machine code would be generated by the OpenCL C compiler. An attempt was made to automatically fix the machine code but it was soon deemed that such errors were out of the scope of this research.

CHAPTER 5. PERFORMANCE RESULTS

A matrix multiplication application was developed in Java, CUDA C and C. The Java version was translated with JavaAutoCuda into the Java + CUDA C version. The time to execute each version on two 4096x4096 matrices is shown in the Time to Execute column in table 10 below. Notice that the development time for the Java + CUDA C version is three times faster than the CUDA C version. The Java + CUDA C version is 2.5X slower than the CUDA C version but it is still 29X faster than the Java only version. With advanced optimizations that will be applied in the future work it is possible that the Java + CUDA C version will have performance closer to that of the CUDA C version.

Version	Development Time	Time to Execute	Speedup over C
Java Only	10 minutes	2893 seconds	1.25 X
Java + CUDA C	10 minutes	124 seconds	29.06 X
CUDA C	29 minutes	49 seconds	73.53 X
C	10 minutes	3603 seconds	1 X

Table 10 – Matrix Multiplication Development Time and Time to Execute

All performance measurements were run using a computer with an Intel Dual Core 2.40Ghz GPU and 2GB memory and a Tesla C1060 GPU. Debian GNU/Linux 5.0.4 was used as the operating system along with the OpenJDK Java Server Virtual Machine version 1.6. All CUDA kernels used 64 blocks and 64 threads.

The CUDA C version was written by the author, a fairly experienced CUDA developer, using a working copy of the matrix multiplication code in both C and Java for reference. 19 minutes of the total time was spent debugging the computation on the GPU to ensure the correct result. The application chosen was an easy problem to convert to something that uses a GPU. Real world problems take much longer to convert a serial version to a GPU parallel version.

Additionally the author has measured the execution time of different iteration sizes and different loop transformations. Detailed and summary tables of performance measurements versus iteration size for the loop transformation choosing the outer-most loop as the body are is in tables 11 and 12 below. Tables 13 and 14 give the same type of information but instead the translator is choosing an inner loop as the body.

Iterations	Time to Serialize	Execution Time	Time to De-Serialize
256x256	55 ms	133 ms	66 ms
512x512	43 ms	547 ms	46 ms
1024x1024	149 ms	2301 ms	141 ms
2048x2048	416 ms	10814 ms	334 ms
4096x4096	1523 ms	82781 ms	1443 ms

Table 11 – GPU Detailed Performance results of choosing the outer-most loop as the body

Iterations	GPU Total Time	CPU Total Time	Speedup
256x256	805 ms	138 ms	-6.2X
512x512	638 ms	1009 ms	1.6X
1024x1024	2595 ms	8058 ms	3.1X
2048x2048	11567 ms	183064 ms	15.8X
4096x4096	85753 ms	1619322 ms	18.9X

Table 12 – Summary GPU and CPU Performance results, outer-most loop

Note that the 4096x4096 iteration entry only has an 18X speedup while the same application with the same number of iterations had a 29X speedup in table 10 above. This is because the Java Virtual Machine optimizes execution of the Java code as it runs.

Iterations	Time to Serialize	Execution Time	Time to De-Serialize
256x256	179 ms	13 ms	125 ms
512x512	275 ms	97 ms	155 ms
1024x1024	1007 ms	785 ms	1192 ms
2048x2048	7247 ms	6388 ms	53757 ms
4096x4096	Out of Java Memory	Out of Java Memory	Out of Java Memory

Table 13 – GPU Detailed Performance results of choosing the inner-most loop as the body

Iterations	GPU Total Time	CPU Total Time	Speedup
256x256	762 ms	126 ms	-6.1X
512x512	570 ms	1014 ms	1.8X
1024x1024	3401 ms	6865 ms	2.0X
2048x2048	68755 ms	182200 ms	2.6X
4096x4096	Out of Java Memory	Out of Java Memory	Out of Java Memory

Table 14 – Summary GPU and CPU Performance results, inner loop

The Java Program that was run is shown in figure 32 below. Tables 11 and 12 refer to choosing the code between lines 3 and 9 to run on the GPU (named the outer loop). Tables 13 and 14 refer to choosing the code between lines 4 and 8 (named an inner loop). The execution time of choosing the inner loop is actually better, but more objects need to be (de)serialized so the total time including the (de)serialization is greater.

```
1   public void runOnGpu(){
2     for(int i = 0; i < mSize; ++i){
```

```
3       for(int j = 0; j < mSize; ++j){
4         int total = 0;
5         for(int n = 0; n < mSize; ++n){
6           total += x[n*mSize+j] * y[i*mSize+n];
7         }
8         z1[i*mSize+j] = total;
9       }
10    }
11  }
```

Figure 32 – Java Matrix Multiplication Program

CHAPTER 6. FUTURE WORK

While continuing work on a doctoral degree, this thesis work will be extended. First, the automatic usage of GPUs from the Java Programming Language will be extended. After that, a translator will be created that automatically distributes computations in Java Bytecode across multiple computers in a network. Each peer will automatically utilize any GPUs that are present. Additionally, each computer will automatically use threading to distribute work among local CPU cores. It will be a goal in the work to make setting up the network to have a low learning curve while at the same time maintaining security. This is essential to achieve wide adaptation by non-computer scientists. To ensure correctness, it will be formally verified that the transformations applied to the Java Bytecode maintain the original behavior. This verification will have to incorporate the different semantics of Java Bytecode for CPUs and CUDA for GPUs. In the next paragraph the optimizations that will be applied to the current work are described.

The first optimization that will be applied to the current work will be to detect more computation that can be automatically parallelized by doing more array access analysis. Currently only very simple and restrictive array access analysis is done. The algorithms from the paper “Region Array SSA” [15] will most likely be used.

The second optimization that will be applied is to take advantage of the fact that the Nvidia GPU architecture allows different groups of 32 threads (called warps) to do different things. Currently it is assumed that all threads in the kernel launch are doing the same thing.

The third optimization that will be applied is array region analysis to reduce the amount of data copied from large arrays. Currently if an array is used on the GPU, the entire array is copied. With array region analysis, only the portions of the array that are needed will be copied.

The fourth optimization that will be applied is optimization of the CUDA GPU code. Currently accesses to read only fields, such as the size of an array, are read every time from global GPU memory.

A fifth research activity will be to apply loop unrolling to loops that have inter-loop dependencies to see if the dependencies can be eliminated. This would allow more possible code to be executed on the GPU.

A sixth research activity will be to include all control constructs in the calculation of the DivergentScore. This will make the DivergentScore more accurate. JavaAutoCuda may be offered for use by the public through a website that users submit compiled Java Bytecode and the result of running JavaAutoCuda is returned. If this is done we will be able to easily run every detected loop and determine empirically a good MaxDivergentScore.

Finally, JavaAutoCuda will be improved to support more Java Programming Language features on the GPU such as: exceptions, static data and methods, anonymous methods and classes, multi-dimensional arrays, Java Collections and strings.

APPENDIX A: RELATED WORK

SECTION A.1 JACKET

Jacket is a patented commercial product from the company AccelerEyes [1]. With Jacket the programmer can develop in MATLAB and specifically tell Jacket which computations should be executed on the GPU. This is done in Jacket by prepending MATLAB functions with a g. For instance, gdouble will cast a CPU matrix to a double GPU matrix. Then once data structures are on the GPU any computation statements associated with those GPU data structures are executed on the GPU. Jacket's patent language specifically mentions working with matrices.

An example of a matrix multiplication using Jacket is available in the Jacket Getting Started Guide [2] and is listed below.

```
1  >>  X = gdouble( magic ( 3 ) );
2  >>  Y = gones( 3, 'double' );
3  >>  A = X * Y;
4
5  A  =
6        15    15    15
7        15    15    15
8        15    15    15
```

Figure 33 – Matrix Multiplication with Jacket and MATLAB

The Jacket Getting Started Guide recommends to first profile your existing serial application that is written in MATLAB and the convert key CPU matrices to GPU matrices.

Jacket also provides a parallel-for that is begun and ended using gfor and gend, respectively. Executing 200 parallel FFTs on a GPU is demonstrated in Figure 34 below. The Getting Started Guide mentions that the Jacket parallel-for is a preliminary feature that only supports a subset of the functionality that is provided with matrix operations in Jacket.

```
1   N = 128; % matrix size
2   M = 200; % number of tiled matrices
3
4   % Create Data
5   [Ac Bc] = deal(complex( gones(N,N,M, 'single'),0));
6
7   % Compute 200 (128x128) FFTs
8   gfor ii = 1:M
9     Ac(:,:,ii) = fft2(Bc(:,:,ii));
10  gend
11
12  % Bring the results back to CPU
13  Ac = single(Ac);
```

Figure 34 – Parallel FFT on GPU with Jacket and MATLAB [2]

The Jacket Getting Started Guide claims that the parallel FFT will achieve a 600% speedup over a serial CPU version. It also discusses a problem that Jacket has with loops. The simple loop listed below is compiled with Nvidia's nvcc compiler every loop iteration. This is because the n is a CPU variable and Jacket cannot effectively handle this case. This case is handled with the methods and algorithms provided in this thesis. The Getting Started Guide does offer a solution: make n a GPU variable.

```
1   A = geye( 3, 'double' );
2   for n = 1:10,
3     A * n
4   end
```

Figure 35 – MATLAB Loop that will be compiled every iteration with Jacket [2]

SECTION A.2 PGI ACCELERATOR COMPILER

The PGI Accelerator Compiler is described in the paper "Implementing the PGI Accelerator Mode" by Michael Wolfe [9]. With the PGI Accelerator Compiler the end user must specify regions that will be computed on the GPU. This is shown in the Fortran Programming Language below.

```
1   !$acc data region copy(a(1:n,1:m)) &
2   !$acc& local(b(2:n-1,2:m-1)) copyin(w(2:n-1)
3     do while(resid.gt.tol)
4       resid = 0.0
5   !$acc region
6       do i = 2, n-1
7         do j = 2, m-1
8           b(i,j) = 0.25*w(i)*(a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1))+
                     (1.0-w(i))*a(i,j)
9         enddo
10      enddo
11      do i = 2, n-1
12        do j = 2, m-1
13          resid = resid + (b(i,j)-a(i,j))**2
14          a(i,j) = b(i,j)
15        enddo
16      enddo
17  !$acc end region
18    enddo
19  !$acc end data region
```

Figure 36 – Fortran code with PGI compiler directives [9]

At line 5 there is a region start directive and at line 17 there is the region end directive. The paper says that the data region directives are optional (lines 1, 2 and 19). These data region directives can be used in cases where two or more consecutive loops can share memory transfers (each consecutive loop commonly has a separate GPU launch and separate memory transfers). Wolfe states that the data region directives are very important because memory transfer is relatively slow over the PCI-express data bus. With garbage collected languages there is an added hurdle which is converting a graph of object

references along with object data into a serial memory suitable for use on the GPU. We paid special attention to this issue while developing JavaAutoCuda.

The PGI Accelerator Compiler supports C and Fortran as input languages and does some advanced optimizations on the input code. To determine what data needs to be allocated on the GPU and copied from the CPU standard data-flow, alias and array region analysis is performed. JavaAutoCuda does a variant of data-flow and alias analysis by simply doing a data-flow analysis on code that has been converted to Single Static Assignment. No array region analysis is done in JavaAutoCuda but will be considered for future work.

Wolfe writes that it is desirable to have many tasks running on the GPU simultaneously so there is adequate use of the processing elements that individually have a lower clock rate than a current CPU. At the same time it is required that there is not a request to run a kernel with so many tasks that the GPU cannot schedule the job due to lack of resources. Nvidia GPU computational resources are grouped with the abstraction of blocks and threads. A kernel launch can have a maximum of 65535 blocks and 512 threads per block. This would equal 33553920 tasks. The PGI Accelerator Compiler tries to map loops directly to block and thread dimensions. This is because global memory accesses are fastest when a large group of tasks request memory locations that are consecutive. Wolfe also notes that loops can be reordered to reduce the number of global memory accesses.

To make an automatic determination of the best loop transformation to use an objective function is used that is attentive to parallelism, memory strides, redundancy across threads,

data cache usage and hardware limits. Their method is to generate many loop transformations and choose which one to run by using the following criteria:

●Choose the schedules with the least memory moves

●From the remaining schedules, choose the schedules with the least irregular memory moves

●From the remaining schedules, choose the schedules with the largest thread count

●From the remaining schedules, choose the schedules with the largest block count

●From the remaining schedules, choose the schedules with the least memory usage

●From the remaining schedules, choose which schedule will be easier for code generation

To evaluate the automatic determination of schedule to run, the sample program in figure 39 was run on 100x100, 1000x1000 and 1000x100 matrix sizes. The results are in table 15 below.

Matrix Size	Chosen Schedule	Fastest Schedule
100x100	25 usec	24 usec
1000x1000	394 usec	350 usec
1000x100	72 usec	54 usec

Table 15 – Results of automatic schedule determination

SECTION A.3 HICUDA

hiCUDA[7] allows the user to write the CPU and GPU code all in one file. hiCUDA is useful for someone who wants to still have control over the low level details of GPU programming but wants some assistance in writing the code that allocates global GPU memory and

transfers the host data to the global GPU memory. An example from [7] is shown below that demonstrates the programming model. In figure 37 the original matrix multiply program is shown and in figure 38 the hiCUDA matrix multiply program is shown.

```
1   float A[64][128];
2   float B[128][32];
3   float C[64][32];
4
5   //initialize A and B (not shown)
6
7   for(i = 0; i < 64; ++i){
8     for(j = 0; j < 32; ++j){
9       float sum = 0;
10      for(k = 0; k < 128; ++k){
11        sum += A[i][k] * B[k][j];
12      }
13      C[i][j] = sum;
14    }
15  }
```

Figure 37 – Matrix Multiplication From [7]

```
1   float A[64][128];
2   float B[128][32];
3   float C[64][32];
4
5   //initialize A and B (not shown)
6
7   #pragma hicuda global alloc A[*][*] copyin
8   #pragma hicuda global alloc B[*][*] copyin
9   $pragma hicuda global alloc C[*][*]
10
11  #pragma hicuda kernel matrixMul tblock(4,2) thread(16,16)
12  #pragma hicuda loop_partition over_tblock over_thread
13  for(i = 0; i < 64; ++i){
14  #pragma hicuda loop_partition over_tblock over_thread
15    for(j = 0; j < 32; ++j){
16      float sum = 0;
17      for(kk = 0; kk < 128; kk += 32){
18  #pragma hicuda shared alloc A[i][kk:kk+31] copyin
19  #pragma hicuda shared alloc B[kk:kk+31][j] copyin
20  #pragma hicuda barrier
21        for(k = 0; k < 32; ++k){
22          sum += A[i][kk+k] * B[kk+k][j];
23        }
24  #pragma hicuda barrier
25  #pragma hicuda shared remove A B
26      }
```

```
27      C[i][j] = sum;
28    }
29  }
30  #pragma hicuda kernel_end
31  #pragma hicuda global copyout C[*][*]
32  #pragma hicuda global free A B C
```

Figure 38 – hiCUDA Matrix Multiplication From [7]

The resulting hiCUDA Matrix Multiplication program exposes fairly low level aspects of using a GPU. The end user has to specify that memory needs to be allocated and copied to the device (lines 7 though 9), that a kernel has to be made of a certain size (line 11), that the kernel ends, memory is copied from the device and GPU memory is freed (lines 30 through 32). The writers of the paper also applied a strip-mining optimization to utilize the fast shared memory of the GPU manually (lines 18 though 20 and 24, 25).

hiCUDA also has a compiler directive that can turn on serial executing while on the GPU. Also it is noteworthy that hiCUDA supports non-perfect distribution of iterations over threads. This means that there may be gaps in iterations because the hiCUDA compiler directly maps loop indices to GPU tasks. To protect against this the compiler will generate guard code. JavaAutoCuda takes a different approach, if there is an loop index it is stored in global memory and fetched when needed. This method supports cases where there is no loop index (such as with Java iterators).

SECTION A.4 HMPP

HMPP [8] is a commercial software product that supports C code annotated with compiler directives that enables the use of accelerators, including GPUs. The directive types are: codelet, execution and data transfer. A codelet specifies that a function is "hardware assisted" (i.e. run on the GPU). An example of using the codelet directive is in figure 39 below. At line one the "output=outv" statement directs the compiler to copy outv back to the CPU after execution. Note that the end user has to supply and unsigned integer array specifying the dimensions of each pointer (for example N1 specifies the dimensions of inv and N3 specifies the dimensions of outv). Codelet functions must be pure functions in HMPP meaning that they always evaluate the same result given the same arguments, have no side effects and have no I/O.

```
1  #pragma hmpp trivial codelet, output=outv
2  void trivial(int n, float a, float *inv, unsigned int N1[1], float
   *outv, unsigned int N3[1]){
4    int i, j;
5    for(i = 0; i < n; i++){
6      outv[i] = a * inv[i];
7    }
8  }
```

Figure 39 – Codelet example from [8]

The next compiler directive is the remote code execution directive. It enables a function to call a codelet. An example is shown in figure 40 below. If the accelerator is busy the CPU will be used.

```
1  #pragma hmpp trivial callsite
2  trivial(n, 2.f, inc, N1, outv, N3);
```

Figure 40 – Remote code execution directive example from [8]

There are also data transfer directives that allow the end user to save copying the same data twice when it is shared amount codelets. HMPP also supports asynchronous launching of codelets so a CPU job can execute at the same time as an accelerated job. The paper [8] states that HMPP is compatible with OpenMP and MPI, two standards for parallel computations using shared memory and message passing. If there are MPI directives in the end user code they will be combined with the HMPP processing to enable multiple computers to assist the computation utilizing any accelerators supported with HMPP.

SECTION A.5 OPENMP TO GPGPU

OpenMP to GPGPU [13] is an effort to convert OpenMP programs into Nvidia GPU programs. The OpenMP to GPGPU reference makes the following high level observations regarding OpenMP and GPUs: 1) OpenMP is efficient at expressing loop-level parallelism 2) The OpenMP concept of a master thread and a pool of workers in the OpenMP fork-join model maps well to a master CPU and a pool of GPU threads and 3) The OpenMP feature of incremental parallelization of applications can be applied to GPU programming.

The paper contributes the first compiler framework for automatic conversion of OpenMP programs to CUDA based programs. It contributes to optimization of GPU computing by identifying and implementing compile-time transformation techniques that optimize GPU

global memory access. The transformations are parallel loop-swap, loop collapsing, caching and memory transfer reduction.

OpenMP to GPGPU translates OpenMP programs to GPU programs in two phases. The first phase is an OpenMP Stream Optimizer and the second phase is O2G Baseline Translation and CUDA optimization. The OpenMP Stream Optimizer is not described in detail other than that it performs parallel loop-swap and loop-collapsing optimizations. The baseline translator is described as executing the following steps: 1) identifying kernel regions for the GPU, 2) extracting kernel regions into functions and transforming them into CUDA kernel functions, and 3) analyzing data that will be accessed by the GPU and inserting necessary memory transfer code. To identify kernel regions for the GPU, the "parallel", "for" and "section" regions are detected. When global synchronization constructs are found (such as "barrier") a kernel region is split into two and in the resulting code the synchronization is done on the CPU.

The second phase is OpenMP to CUDA Baseline Translation. Notably in this stage, optimization is done regarding the sections of serial code within a parallel portion in the OpenMP code. The compiler includes the serial sections and redundantly executes them on all parallel threads to reduce costly memory transfers to and from the CPU.

The major high level aspects of OpenMP to GPGPU have been discussed. OpenMP to GPGPU also contains the following optimizations that are discussed below: parallel loop-swap, loop collapsing, caching global data, and memory transfer reduction.

Parallel loop-swap is an optimization that attempts to make accesses to global memory follow a pattern where each subsequent thread accesses a location one more than the previous thread. The algorithm is as follows: From all the possible loops within a set of nested loops, identify candidate arrays that have continuous memory access. Select a loop whose index variable will cause accesses to a candidate array to be monotonically incrementing. If all the loops between the selected loop and another inner loop from the selected loop can be parallelized, interchange the said loops.

Loop collapsing tries to collapse a nested loop into a single loop where accesses to an array are proved continuous. In OpenMP to GPGPU runtime checks are added to prove that array accesses are continuous for cases that can only be determine at runtime. It can be seen from figure 41 and 42 below that this optimization brings a multiplication that may possibly be shared among threads (figure 41, line 4) to a shared place (figure 42, line 2). At the same time control divergence is decreased because some of the calculation in a possibly divergent portion of code is moved outside of a loop.

```
1  #pragma omp parallel for
2  for (i = 0; i < NUM_ROWS; ++i) {
3    for (j = rowptr[i]; j < rowptr[i+1]; ++j)
4      w[i] += A[j]*p[col[j]];
5  }
```

Figure 41 – Original OpenMP code before loop collapsing [13]

```
1  if (tid1 < rowptr[NUM_ROWS]) {
2    l_w[tid1] = A[tid1]*p[col[tid1]];
3  }
4  if (tid2 < NUM_ROWS) {
5    for (j = rowptr[tid2]; j < rowptr[tid2+1; ++j)
6      w[tid] += l_w[j];
```

7	}

Figure 42 – Loop collapsed CUDA code [13]

The global memory caching optimization is accomplished in OpenMP to GPGPU by first running a static data flow analysis to identify temporal locality of global data. The variables declared in a fast memory space are declared and used as a cache for the most used elements. The paper states that there are limited hardware resources for the cache and there are performance issues that are difficult to control statically. As a result the compiler framework includes language extensions for a programmer to guide these optimizations.

The last optimization by the OpenMP to GPGPU compiler is memory transfer reduction. In this optimization a data flow analysis is done on shared memory to determine what data that is required on the CPU is in shared memory. The algorithms in this thesis include the same optimization, except operating on global memory.

SECTION A.6 CUDA-LITE

CUDA-lite [5] is not a whole compiler or translator as most of the other references are. CUDA-lite instead applies optimizations to programs already written in CUDA C by a programmer (or perhaps auto-generated by a compiler). CUDA-lite transforms programs in CUDA C that only use slow global memory to CUDA C programs that use a combination of shared memory and global memory. Shared memory can be thought of as useful to store data elements that will be used repeatedly. Additionally shared memory is useful as a

temporary storage when doing coalesced reads. CUDA-lite automatically transforms programs to do coalesced reads. Below in figure 43 is the original kernel that uses global memory only and in figure 44 the modified kernel that coalesces reads and writer.

```
1   #define ASIZE 3000
2   #define TPB 256
3
4   __global__ void kernel (float *a, float *b){
5     int thi = threadIdx.x;
6     int bki = blockIdx.x;
7     float t = (float) thi + bki;
8     int i;
9
10    if(bki * TPB + thi >= ASIZE)
11      return;
12    for(i = 0; i < ASIZE; ++i){
13      b[(bki*TPB+thi)*ASIZE+i] = a[(bki*TPB+thu)*ASIZE+i] * t;
14    }
15  }
```

Figure 43 – Original Kernel from [5]

```
1   #define ASIZE 3000
2   #define TPB 256
3
4   __global__ void kernel (float *a, float *b){
5     int thi = threadIdx.x;
6     int bki = blockIdx.x;
7     float t = (float) thi + bki;
8     int i;
9
10    int j, End, k;
11    __shared__ float a_shared[TPB][TPB];
12    __shared__ float b_shared[TPB][TPB];
13    //loop tiling
14    End = ASIZE % TPB == 0 ? ASIZE / TPB : (ASIZE/TPB) + 1;
15    for(j = 0; j < End; j++){
16      //coalesced loads
17      __syncthreads();
18      for(k = 0; k < TPB; ++k){
19        if((j*TPB + thi < ASIZE) &&
20           ((bki*TPB+k)*ASIZE + j*TPB + thi < ASIZE*ASIZE)){
21             a_shared[k][thi] = a[(bki*TPB + k)*ASIZE + j*TPB + thi];
22        }
23      }
24      __syncthreads();
25
26      //conditions: TPB && obey original end && !(early exit condition)
27      for(i = 0; (i < TPB) && (j*TPB+i < ASIZE) && !(bki*TPB + thi >=
```

```
    ASIZE); ++i){
28        b_shared[thi][i] = a_shared(thi][i] * t;
29      }
30
31      //coalesced stores
32      __syncthreads();
33      for(k = 0; k < TPB; ++k){
34        if((j*TPB + thi < ASIZE) &&
35           ((bki*TPB+k)*ASIZE + j*TPB + thi < ASIZE*ASIZE)){
36             b[(bki*TPB + k)*ASIZE + j*TPB + thi] = b_shared[k][thi];
37        }
38      }
39      __syncthreads();
40    }
41  }
```

Figure 44 – Modified Kernel from [5] that has coalesced reads and writes

Currently to enable CUDA-lite to correctly generate the coalesced version annotations must be added to the original CUDA code. It would be an interesting project to see if these annotations could be automatically generated from JavaAutoCuda (this work). The results from CUDA-lite show that coalescing reads and writes can improve performance by 2 to 17x.

SECTION A.7 PYTHON PROJECT

The unnamed Python project [4] converts the high level Python Programming Language to use GPUs. They assume that the program has been annotated with what loops are parallel. In addition, since Python is dynamically typed, a piece of Python code that will be on the GPU needs to have its types annotated to help the translator. A restriction is also made that

requires a variable to not change its type while on the GPU. The author of this thesis thinks that this is a heavy restriction on an end user project with many existing lines of code.

The Python project determines memory locations that need to be transferred rather than operating directly on objects. In this way they may send more data than is needed. The project uses a special iterator, prange to specify that a loop is parallel. A matrix multiplication program demonstrating this from the paper is listed below.

```
1   @unpython.gpu
2   @unpython.type (‘ndarray[double, 2]’, ‘ndarray[double, 2]’,
3                   ‘ndarray[double, 2]’, None)
    def matrix mult (a, b, c):
4     m = shape(a)[0]
5     n = shape(b)[1]
6     p = shape(a)[1]
7     for i in prange(m):
8       for j in prange(n):
9         sum = 0.0
10        for k in xrange(p):
11          sum += a[i, k] * b[k, j]
12        c[i, j] = sum
```

Figure 45 - Python GPU Matrix Multiplication

The project includes the following optimizations: loop unrolling, limited loop fusion, partial load coalescing and partial redundant-load elimination.

APPENDIX B: MODIFICATIONS TO OPEN SOURCE SOFTWARE

The author has modified the Soot Java Optimization Framework and JCuda. The modifications to Soot are larger than those to JCuda. Both are described in paragraphs below.

Soot was designed to have two modes. One mode is a local only mode that executes all user supplied transformations on classes one at a time. The other mode is a global mode that constructs graphs such as a call graph. The transformations I have designed require some global knowledge that is provided in the global mode, but the global mode is very slow for moderately sized input programs. To get the global knowledge I needed without the expense of creating the graphs I added functionality to the local mode to execute transformations one at a time on all classes. I also modified Soot to enable adding classes during a transformation phase. The modified version of Soot is publicly available at: http://github.com/pcpratts/SootPhil

For this project JCuda was modified in a very simple way. The modification is to allow the client code to load dynamic native libraries rather than the library to support multiple operating systems in JavaAutoCuda. The modified version of Jcuda is publicaly available at: http://github.com/pcpratts/JCudaJavaPhil

APPENDIX C. READER'S WRITER'S SHARED MEMORY CACHE

```
1   __device__ int
2   cache_get_int(edu_pcpratts_gc_info gc_info, int address){
3     int prev_lock;
4     int value;
5     int prev_num;
6
7     char * to_space = get_to_space_address(gc_info);
8     int ptr = address % (CACHE_SIZE_INTS / CACHE_ENTRY_SIZE);
9     ptr *= CACHE_ENTRY_SIZE;
10
11    atomicAdd(&mCache[ptr+0], 1);
12    prev_lock = atomicOr(&mCache[ptr+1], 2);
13    //reading or clear
14    if(prev_lock < 4){
15      if(mCache[ptr+2] == address){
16        value = mCache[ptr+3];
17        prev_num = atomicSub(&mCache[ptr+0], 1);
18        if(prev_num == 1){
19          atomicAnd(&mCache[ptr+1], ~2);
20        }
21        return value;
22      } else {
23        prev_num = atomicSub(&mCache[ptr+0], 1);
24        if(prev_num == 1){
25          atomicAnd(&mCache[ptr+1], ~2);
26        }
27        value = *((int *) &to_space[address]);
28      }
29    } else {
30      value = *((int *) &to_space[address]);
31      prev_num = atomicSub(&mCache[ptr+0], 1);
32      if(prev_num == 1){
33        atomicAnd(&mCache[ptr+1], ~2);
34      }
35    }
36    //replace entry in cache
37    prev_lock = atomicOr(&mCache[ptr+1], 4);
38    //if clear or empty
39    if(prev_lock == 0 || prev_lock == 1){
40      mCache[ptr+2] = address;
41      mCache[ptr+3] = value;
42      __threadfence_block();
43      mCache[ptr+1] = 0;
44    } else {
45      atomicAnd(&mCache[ptr+1], ~4);
46    }
47    return value;
48  }
```

Figure 46– Reader's Writer's Shared Memory Cache

APPENDIX D. COMPLETE JAVA+CUDA C VECTOR ADDITION CODE

This appendix lists all the generated and transformed code for a vector addition example. The Shimple code is listed in Java for simplicity. Figure 47 is the vector addition Java Source. Figure 48 lists the transformed vector addition Java Source. Figure 49 shows a Concrete LoopBody. Figure 50 shows a Concrete GcObjectVisitor. Finally, figure 51 displays generated CUDA C code.

```
1   package edu.syr.pcpratts.javaautogpu.runtime.test;
2
3   public class VectorAddExample {
4
5     int[] x;
6     int[] y;
7     int[] ret;
8
9     public void add(){
10      for(int i = 0; i < x.length; ++i) {
11         ret[i] = x[i]+y[i];
12      }
13    }
14  }
```

Figure 47 - Vector Addition Java Source

```
1   package edu.syr.pcpratts.javaautogpu.runtime.test;
2
3   public class VectorAddExample {
4
5     int[] x;
6     int[] y;
7     int[] ret;
8
9     public void add(){
10      QueueManager manager = QueueManager.v("0");
11      for(int i = 0; i < x.length; ++i){
12        LoopBody0 body = new LoopBody0(this, i);
13        manager.enqueue(body);
14      }
15      manager.run();
```

```
16      Iterator<LoopBody> iter = manager.iterator();
17      while(iter.hasNext()){
18        LoopBody0 curr_body = (LoopBody0) iter.next();
19        //any inter-loop dependent code below the parallel portion
20        //of the loop would be executed here. in this example there
21        //is none.
22      }
23    }
24  }
```

Figure 48- Transformed Vector Addition Java Source

```
1   package edu.syr.pcpratts.javaautogpu.generated;
2
3   import edu.syr.pcpratts.javaautogpu.runtime.LoopBody;
4   import edu.syr.pcpratts.javaautogpu.runtime.test;
5   import edu.syr.pcpratts.javaautpgpu.GcObjectVisitor;
6   import edu.syr.pcpratts.javaautogpu.runtime.Memory;
7   import edu.syr.pcpratts.javaautogpu.runtime.gpu.GcHeap;
8
9   public class LoopBody0 extends LoopBody {
10
11    public VectorAddExample r0;
12    public int i0_1;
13
14    public LoopBody0(VectorAddExample r0, int i0_1){
15      this.r0 = r0;
16      this.i0_1 = i0_1;
17    }
18
19    public void run(){
20      r0.ret = r0.x[i0_1] + r0.y[i0_1];
21    }
22
23    public String getCode(){
24      StringBuilder ret = new StringBuilder();
25      //note there is a maximum size of a String in a java class file
26      //so the string is split up
27      ret.append("<compiled CUDA C code part 1>\n");
28      //...
29      ret.append("<compiled CUDA C code part N>\n");
30      return ret.toString();
31    }
32
33    public GcObjectVisitor getVisitor(Memory mem, GcHeap heap){
34      return new LoopBody0GcObjectVisitor(mem, heap);
35    }
36  }
```

Figure 49- Generated LoopBody0 Java Source

```
1   package edu.syr.pcpratts.javaautogpu.generated;
2
3   import edu.syr.pcpratts.javaautpgpu.GcObjectVisitor;
4   import edu.syr.pcpratts.javaautogpu.runtime.Memory;
5   import edu.syr.pcpratts.javaautogpu.runtime.gpu.GcHeap;
6
7   public class LoopBody0GcObjectVisitor extends GcObjectVisitor {
8
9     public LoopBody0GcObjectVisitor(Memory mem, GcHeap heap){
10      super(mem, heap);
11    }
12
13    public int doWriteToHeap(Object o, boolean write_data){
14      if(o instanceof int[]){
15        return GcArrayMover.write((int[]) o, write_data);
16      }
17      if(o instanceof VectorAddExample){
18        VectorAddExample vec = (VectorAddExample) o;
19        int heap_end_ptr = mHeap.getHeapEndPtr();
20        mMem.writeByte(3);
21        mMem.writeByte(0);
22        mMem.writeByte(11);
23        mMem.writeByte(0);
24        mMem.writeInt(20);
25        mHeap.incrementHeapEndPtr(20);
26        mMem.pushAddress();
27        mMem.incrementAddress(12);
28        int ref_addr1 = writeToHeap(vec.ret, false);
29        int ref_addr2 = writeToHeap(vec.x, true);
30        int ref_addr3 = writeToHeap(vec.y, true);
31        int top_address = mMem.topAddress();
32        mMem.popAddress();
33        mMem.writeInt(ref_addr1);
34        mMem.writeInt(ref_addr2);
35        mMem.writeInt(ref_addr3);
36        mMem.setAddress(top_address);
37        return heap_end_ptr;
38      }
39      if(o instanceof LoopBody0){
40        LoopBody0 body = (LoopBody0) o;
41        int heap_end_ptr = mHeap.getHeapEndPtr();
42        mMem.writeByte(1);
43        mMem.writeByte(0);
44        mMem.writeByte(7);
45        mMem.writeByte(0);
46        mMem.writeInt(16);
47        mHeap.incrementHeapEndPtr(16);
48        mMem.pushAddress();
49        mMem.incrementAddress(8);
50        int ref_addr1 = writeToHeap(body.r0, true);
51        int top_address = mMem.topAddress();
52        mMem.popAddress();
53        mMem.writeInt(ref_addr1);
54        mMem.writeInt(body.i0_1);
```

```
55        mMem.setAddress(top_address);
56        return heap_end_ptr;
57      }
58      throw new RuntimeException("unknown type");
59    }
60
61    public Object doReadFromHeap(Object o, boolean read_data){
62      int type = readType();
63      if(type == 3){
64        return GcArrayMover.read((int[]) o);
65      }
66      if(type == 11){ //VectorAddExample
67        mMem.incrementAddress(3);
68        byte ctor_used_on_gpu = mMem.readByte();
69        if(ctor_used_on_gpu == 1){
70          o = new VectorAddExample(Sentinal.instance());
71        }
72        VectorAddExample vec = (VectorAddExample) o;
73        mMem.incrementAddress(4);
74        int address_of_ret = mMem.readInt();
75        mMem.pushAddress();
76        mMem.setAddress(address_of_ret);
77        vec.ret = readFromHeap(vec.ret);
78        mMem.popAddress();
79        mMem.incrementAddress(8);
80        return vec;
81      }
82      if(type == 7){ //LoopBody0
83        mMem.incrementAddress(3);
84        byte ctor_used_on_gpu = mMem.readByte();
85        if(ctor_used_on_gpu == 1){
86          o = new LoopBody0(Sentinal.instance());
87        }
88        LoopBody0 body = (LoopBody0) o;
89        mMem.incrementAddress(4);
90        int address_of_vec = mMem.readInt();
91        mMem.pushAddress();
92        mMem.setAddress(address_of_vec);
93        body.r0 = readFromHeap(body.r0);
94        mMem.popAddress();
95        mMem.incrementAddress(4);
96        return body;
97      }
98      throw new RuntimeException("unknown type");
99    }
100 }
```

Figure 50 - Generated LoopBody0GcObjectVisitor Java Source

```
1   __device__ int
2   edu_syr_pcpratts_gc_get_id(edu_pcpratts_gc_info gc_info){
3     return blockIdx.x * blockDim.x + threadIdx.x;
4   }
5
6   __device__ edu_pcpratts_gc_info
7   edu_syr_pcpratts_gc_init(char * gc_info_space, char * to_space,
      char * from_space, char * to_handle_map, char * from_handle_map,
      int to_space_free_ptr, int space_size, int cache_assoc){
8
9     mCacheAssoc = cache_assoc;
10    *((int *) (&gc_info_space[0])) = (int) to_space;
11    *((int *) (&gc_info_space[4])) = (int) from_space;
12    *((int *) (&gc_info_space[8])) = (int) to_handle_map;
13    *((int *) (&gc_info_space[12])) = (int) from_handle_map;
14    *((int *) (&gc_info_space[16])) = to_space_free_ptr;
15    *((int *) (&gc_info_space[20])) = space_size;
16
17    __syncthreads();
18
19    return gc_info_space;
20  }
21
22  __device__ char *
23  edu_syr_pcpratts_gc_deref(edu_pcpratts_gc_info gc_info, int handle){
24    char * to_space = edu_syr_pcpratts_gc_get_to_space_address(gc_info);
25    return &to_space[handle];
26  }
27
28  __device__ void
29  edu_syr_pcpratts_gc_assign(edu_pcpratts_gc_info gc_info,
      int * lhs_ptr, int rhs){
30    *lhs_ptr = rhs;
31  }
32
33  //no cache
34  __device__ int
35  edu_syr_pcpratts_cache_get_int(edu_pcpratts_gc_info gc_info,
      int address){
36    char * to_space = (char *)
37  edu_syr_pcpratts_gc_get_to_space_address(gc_info);
38    return *((int *) &to_space[address]);
39  }
40
41  __device__ void edu_syr_pcpratts_javaautogpu_generated_LoopBody0_run(
    edu_pcpratts_gc_info gc_info, int thisref){
42    int this0 = -1;
43    int r0 = -1;
44    int i0_1;
45    int $r2 = -1;
46    int $r3 = -1;
47    int $i2;
48    int $r4 = -1;
49    int $i3;
```

```
50    int $i4;
51    edu_syr_pcpratts_gc_assign(gc_info, & this0 ,  thisref );
52
53    r0  = instance_getter_edu_syr_pcpratts_javaautogpu_generated_
            LoopBody0_r0( gc_info, this0);
54
55    i0_1  = instance_getter_edu_syr_pcpratts_javaautogpu_generated_
              LoopBody_i0_1(gc_info, this0);
56
57    $r2  = instance_getter_edu_syr_pcpratts_javaautogpu_runtime
             _test_VectorAddExample_ret(gc_info, r0);
58
59    $r3  = instance_getter_edu_syr_pcpratts_javaautogpu_runtime_test
             _VectorAddExample_x(gc_info, r0);
60    $i2  = int__array_get_cached(gc_info, $r3, i0_1);;

61    $r4  = instance_getter_edu_syr_pcpratts_javaautogpu_runtime_test
62           _VectorAddExample_y(gc_info, r0);
63
      $i3  = int__array_get_cached(gc_info, $r4, i0_1);;
64
65    $i4  =  $i2   +   $i3  ;
66    int__array_set(gc_info, $r2, i0_1,  $i4 );
67
68    return;
69  }
70
71  __device__ void
72  edu_syr_pcpratts_javaautogpu_runtime_RuntimeBasicBlock_run(
      edu_pcpratts_gc_info gc_info, int thisref){
73
74  }
75
76  __device__ void
77  invoke_edu_syr_pcpratts_javaautogpu_generated_LoopBody0_run(
      edu_pcpratts_gc_info gc_info, int thisref){
78    char * thisref_deref = edu_syr_pcpratts_gc_deref(gc_info, thisref);
79    GC_OBJ_TYPE_TYPE derived_type =
        edu_syr_pcpratts_gc_get_type(thisref_deref);
80    if(0){}
81    else if(derived_type == 7){
82      edu_syr_pcpratts_javaautogpu_generated_LoopBody0_run(gc_info,
          thisref);
83    }
84    else if(derived_type == 8){
85      edu_syr_pcpratts_javaautogpu_runtime_LoopBody0_run(gc_info,
          thisref);
86    }
87  }
88
89  __device__ int
90  instance_getter_edu_syr_pcpratts_javaautogpu_generated_LoopBody0_r0(
      edu_pcpratts_gc_info gc_info, int thisref){
91
```

```
92    char * thisref_deref = edu_syr_pcpratts_gc_deref(gc_info, thisref);
93    return *((int *) &thisref_deref[8]);
94  }
95
96  __device__ int
97  instance_getter_edu_syr_pcpratts_javaautogpu_generated_LoopBody0_i0_1(
      edu_pcpratts_gc_info gc_info, int thisref){
98
99    return edu_syr_pcpratts_cache_get_int(gc_info, thisref+12);
100 }
101
102 __device__ int
103 instance_getter_edu_syr_pcpratts_javaautogpu_runtime_test_
      VectorAddExample_ret(edu_pcpratts_gc_info gc_info, int thisref){
104
105   char * thisref_deref = edu_syr_pcpratts_gc_deref(gc_info, thisref);
106   return *((int *) &thisref_deref[8]);
107 }
108
109 __device__ int
110 instance_getter_edu_syr_pcpratts_javaautogpu_runtime_test_
      VectorAddExample_x(edu_pcpratts_gc_info gc_info, int thisref){
111
112   return edu_syr_pcpratts_cache_get_int(gc_info, thisref+12);
113 }
114
115 __device__ int
    instance_getter_edu_syr_pcpratts_javaautogpu_runtime_test_
116   VectorAddExample_y(edu_pcpratts_gc_info gc_info, int thisref){
117
118   return edu_syr_pcpratts_cache_get_int(gc_info, thisref+16);
119 }
120 __device__ void
121 instance_setter_edu_syr_pcpratts_javaautogpu_runtime_test_
      VectorAddExample_y(edu_pcpratts_gc_info gc_info, int thisref,
      int parameter0){
122
123   char * thisref_deref = edu_syr_pcpratts_gc_deref(gc_info, thisref);
124   edu_syr_pcpratts_gc_assign(gc_info, (int *) &thisref_deref[16],
        parameter0);
125 }
126
127 __device__ int int__array_get(edu_pcpratts_gc_info gc_info,
      int thisref, int parameter0){
128   char * thisref_deref = edu_syr_pcpratts_gc_deref(gc_info, thisref);
129   return *((int *) &thisref_deref[12+(parameter0*4)]);
130 }
131
132 __device__ int int__array_get_cached(edu_pcpratts_gc_info gc_info,
      int thisref, int parameter0){
133   return edu_syr_pcpratts_cache_get_int(gc_info,
        thisref+12+(parameter0*4));
134 }
135
```

```
136 __device__ void int__array_set(edu_pcpratts_gc_info gc_info,
      int thisref, int parameter0, int parameter1){
137   char * thisref_deref = edu_syr_pcpratts_gc_deref(gc_info, thisref);
138   *((int *) &thisref_deref[12+(parameter0*4)]) = parameter1;
139 }
140
141 __global__ void entry(char * gc_info_space, char * to_space,
      char * from_space, char * to_handle_map,
      char * from_handle_map, int * handles, int * to_space_free_ptr,
      int space_size, int cache_assoc, int iters){
142
143   edu_pcpratts_gc_info gc_info =
      edu_syr_pcpratts_gc_init(gc_info_space,
        to_space, from_space, to_handle_map, from_handle_map,
        *to_space_free_ptr, space_size, cache_assoc);
144   int loop_control = edu_syr_pcpratts_gc_get_id(gc_info);
145   if(loop_control < iters){
146     int handle = handles[loop_control];
147     edu_syr_pcpratts_javaautogpu_generated_LoopBody0_run(gc_info,
        handle);
148   }
149   __syncthreads();
150   *to_space_free_ptr =
        edu_syr_pcpratts_gc_get_to_space_free_ptr(gc_info);
151   __syncthreads();
152 }
```

Figure 51 - Generated CUDA C code (abbreviated)

BIBLIOGRAPHY

[1] AccelerEyes Jacket: http://www.accelereyes.com/

[2] AccelerEyes Jacket, Getting Started Guide: http://www.accelereyes.com/content/doc/GettingStartedGuidev1.3.pdf

[3] An anomaly in space-time characteristics of certain programs running in a paging machine. L. A. Belady, R. A. Nelson, G. S. Shedler. Communications of the ACM. 1969.

[4] Compiling Python to a Hybrid Execution Environment. Rahul Garg, Jose Nelson Amaral. Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. 2010.

[5] CUDA-Lite: Reducing GPU Programming Complexity. Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, Wen-Mei W. Hwu. Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008.

[6] Efficiently computing static single assignment form and the control dependence graph. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. ACM Transactions on Programming Languages and Systems (TOPLAS) 1991.

[7] hiCUDA: a high-level directive-based language for GPU programming, T. D. Han and T. S. Abdelrahman, Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), 2009.

[8] HMPP: A hybrid multicore parallel programming environment, Dolbeau, R., Bihan, S. and Bodin.

[9] Implementing the PGI Accelerator model. Michael Wolfe, ACM International Conference Proceeding Series; Vol. 425 archive. Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics. 2010

[10] Infernal GPU Writeup: Adam Bazinet. http://serine.umiacs.umd.edu/files/infernal-gpu_writeup.pdf. 2008.

[11] Mars, a MapReduce Framework: http://www.cse.ust.hk/gpuqp/Mars.html

[12] Nvidia: http://www.nvidia.com/object/product_tesla_c1060_us.html

[13] OpenMP to GPGPU: a compiler framework for automatic translation and optimization. Seyong Lee, Seung-Jai Min, Rudolf Eigenmann. Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. 2009

[14] Radar Signal Processing with Graphics Processors (GPUs): http://www.hpcsweden.se/files/RadarSignalProcessingwithGraphicsProcessors.pdf

[15] Region array SSA. Silvius Rus, Guobin He, Christophe Alias, Lawrence Rauchwerger. Proceedings of the 15th international conference on Parallel architectures and compilation techniques. 2006

[16] Sable Research Group at McGill University: http://www.sable.mcgill.ca/soot/

[17] Wikipdeia, FLOPS: http://en.wikipedia.org/wiki/FLOPS

[18] Wikipedia, List of Java Virtual Machines: http://en.wikipedia.org/wiki/List_of_Java_virtual_machines

[19] Wikipedia, List of JVM Languages: http://en.wikipedia.org/wiki/List_of_JVM_languages

[20] Wikipedia, Software Transactional Memory: http://en.wikipedia.org/wiki/Software_transactional_memory

VITA

NAME OF AUTHOR: Philip Charles Pratt-Szeliga

PLACE OF BIRTH: Utica, NY, USA

DATE OF BIRTH: September 20, 1983

UNDERGRADUATE SCHOOL ATTENDED:

Rensselaer Polytechnic Institute, Troy, New York, USA

DEGREES AWARDED:

Bachelor of Science in Computer and Systems Engineering, 2005, Rensselaer Polytechnic Institute

AWARDS AND HONORS:

All University Masters of Science Prize, Syracuse University, 2010

PROFESSIONAL EXPERIENCE:

Teaching Assistant, Department of Electrical Engineering and Computer Science, Syracuse University, 2008-2010

Instructor, Department of Electrical Engineering and Computer Science, Syracuse University, 2009