ABSTRACT – Graphics Processing Units are a new type of hardware device that have over 112 floating-point units accessible from CUDA or OpenCL using the SPMD abstraction ~~from CUDA or OpenCL~~. These can be used in many high performance applications such as medical imaging, computational biology and video image processing. The Rootbeer GPU Compiler is a system that enables these processors to be used from the Java Programming Language. The system is the only system including support for Java Programming Language features such as methods, fields and objects. It includes a high performance throughput API that offers faster performance than the competitor Aparapi and contains a complete API to use CUDA primitives from Rootbeer including atomic integers, shared memory, threadfence and syncthreads. To obtain maximum performance while programming a GPU, the developer should experiment with thread count, block count, shared memory size and register count. These aspects of programming are all configurable while using Rootbeer and we show in examples that tuning these leads to improved performance. This dissertation explains GPU programming and describes the internal structure of the Rootbeer GPU Compiler along with additions to the Soot Java Optimization Framework and several examples demonstrating performance speedups. We show ~~in~~ an example from computational biology of using Hidden Markov Model learning and likelihood calculation which~~and~~ Rootbeer can accelerate 102.7x with an NVIDIA Tesla C2050 device. In the final concluding remarks we give recommendations for supporting Java on GPUs to NVIDIA and Oracle for an industrial setting.

THE ROOTBEER GPU COMPILER

By

Phil Pratt-Szeliga
B.S. Rensselaer Polytechnic Institute, 2005
M.S. Syracuse University, 2010

DISSERTATION

Submitted in partial fulfillment of the requirements for the
degree of Doctor of Philosophy in Computer Information Science and Engineering
in the Graduate School of Syracuse University

December 2015

Approved ____________________________
Professor James Fawcett

Date ____________________________

Table of Contents

List of Illustrative Materials

Chapter 1

Research Statement
GPU Programming
Contributions
Literature Review
Summary of Results

1.1. Research Statement

In this research the first compiler that allows methods, objects and fields from the Java Programming Language to run on the Graphics Processing Unit (GPU) was created. The work is more advanced than all other competing compilers because it supports serialization of complex objects directly to GPU memory. All other compilers require the developer to first convert their complex objects into simple arrays before using the GPU. Due to the support for objects, it is also possible to support method calls targeting any object. This is contrast to other compliers that only allow method calls within the same object. In addition, Rootbeer is the most advanced system for an expert user and certain parallel GPU algorithms and data structures are only possible with Rootbeer.

The first Rootbeer paper was published at HPCC-2012 [25]. Shortly after, in August 2012, a story about Rootbeer was published on Slashdot [1]. Slashdot is a popular news website for science and technology topics. After the story was released, the star count on the Rootbeer github page [2] began rapidly increasing and soon became a trending repository with over 570 stars. Twitter was active with the story and thousands of tweets were posted during that time. Currently, searching for "Rootbeer GPU Compiler" gives about 45,400 results on google.com. Now the Rootbeer github page has over 960 stars. After this in March 2013, a talk was presented at the NVIDIA GTC 2013 Conference [3] and another talk was presented on Rootbeer at SOAP 2013 [4] during the following summer.

Rootbeer has 19 stars from major technology companies (Appendix A), and has incoming links from the OpenJDK domain [5] and was also listed as "state of the art" in a DARPA STTR Solicitation [6]. Over 350 emails have been exchanged with people using Rootbeer and a former Master's student has extended Rootbeer in his thesis to support GPUs on clouds of multiple GPU enabled computers using Apache Hama.

These developments began with work on GPUs seven years ago and, at the time, the research was interested in accelerating C/C++ code. It was found that using these specialized processors could be quite a challenge for an inexperienced CUDA developer. The next section is devoted to GPU Programming and describes the process we went through to develop the application. The research indicated it was clear that assistance with serialization and CUDA code generation was key to improving programmer productivity. Rootbeer was then designed to provide this assistance so that one can focus on high-level changes needed in algorithms and data structures that will result in faster parallel programs using GPUs.

The up-coming discussion on GPU Programming should demonstrate to the reader why Rootbeer's support is important and needed. After the discussion, the remainder of the chapter describes the contributions, a summary of the performance numbers and a short literature review.

Chapter 2 will discuss technologies used in Rootbeer, which will give the reader a foundation for understanding how Rootbeer works and motivate the importance of our~~the~~ additions to the Soot Java Optimization Framework [7]. Chapter 3 will show how to program with Rootbeer and in Chapters 4, 5 and 6 we will show the algorithms and data structures used by Rootbeer and it~~'~~s internal structure. Chapter 7 will show performance of a variety of algorithms and discuss more advanced ways to optimize GPU performance on a single specific device. Chapter 8 ~~will~~ concludes the dissertation and provides recommendations for projects aiming to support GPUs from Java, such as the Sumatra Project from Oracle.

1.2. GPU Programming

Often a developer who wants to program the GPU will start out writing an~~the~~ application in serial using C/C++. Or maybe their task is to port an existing application to the GPU.

We started programming GPUs by attempting to accelerate a tomographic reconstruction algorithm for medical imaging. The code was written in a serial manner, and in this code, there were many loops ranging over a rotation and a 2-dimensional plane.

Inside of the loops there was code to compute the inverse radon transform, which also had some levels of looping.

We started looking for places where we could launch threads in parallel. We will call each of these places a “cut”. The code given to us had huge loops with methods and globals inside them. In the methods there were more loops. In the first attempt, we chose a fairly low level to make a cut. At this cut level, we needed to 1) allocate GPU memory, 2) copy state to GPU memory, 3) write the GPU code in CUDA, 4) write code to launch the task while choosing how many blocks and threads to use, 5) copy state back from GPU memory and 6) free GPU memory.

This first cut took a lot of developer time to organize transferring our state into and out of arrays of primitive types. Adding to our difficulty, the existing code had many global arrays. We had to manually find sizes of these arrays so we could copy them to the GPU using a specialized memory copy. Our data also included a two dimensional array. This required first a GPU memory allocation of the outer array, and then GPU memory allocations of all inner arrays and after special memory copies. This is common practice in C/C++, but with Java/Rootbeer a developer does not have to do this, saving programming time for an experienced developer and opening new possibilities for multi-disciplinary researchers coming from other fields such as computational biology.

Once through the issues of creating our first GPU program with a single cut, we ran the program and debugged the errors. This required approximately a month’s time. But to our disappointment, the code running on the GPU ran slower than our reference implementation on the CPU.

We studied the GPU documentation and learned that we needed to launch thousands of threads in parallel, not just a few. In order to get this many threads, we needed to increase the looping level. We decided to move our cut level to nearly the outer most loop level.

We re-wrote the code for 1) memory allocation, 2) serialization, 3) task size selection, 4) kernel code, 5) deserialization and 6) memory release. Now re-writing our GPU application to use a different "cut" level, we found that our tasks would not launch due to insufficient GPU resources. At this high-up level in the application, the program could not fit within the GPU registers available on our device, but we did have enough threads now.

It was learned, that we had to make a cut at the proper level. If the level is too low, the developer may not get a speedup. A lower cut does less computation per launch and cannot hide the latency associated with GPU computation. If the level is too high, the developer may run out of GPU resources. The higher level cut often requires more device memory, registers and threads.

Looking back, we saw that experimenting with a new cut level can be expensive in terms of developer time. This is primarily because of the needed manual development of interfacing the CPU memory with the GPU and writing the proper kernel code. In a full application, many cuts are most likely needed to handle regions of serial code mixed with regions of highly parallel GPU code. Also, in larger programs, the data may be more complicated; in general, the data is an arbitrary graph of composite objects.

Rootbeer's major feature is that it can serialize and deserialize arbitrary graphs of composite objects to GPU memory and generate the matching CUDA code. This enables the developer to experiment with many cut levels and changes in algorithms, data structures and GPU configuration to obtain maximum performance.

We will now turn to talk about the contributions of this research and continue on with an in-depth discussion of CUDA programming and GPU optimization later.

1.3 Contributions

We are making the following contributions in~~of~~ this dissertation.

1) Rootbeer is the most advanced system for expert GPU usage. With our system the developer can configure ~~the~~ shared memory, thread / block count and register count, which are essential for the correct operation of certain parallel GPU applications. Also Rootbeer contains support for CUDA built-ins such as syncthreads, syncthreadsCount, threadfence, atomicAdd, atomicExch, atomicCAS and more, which are required for advanced GPU algorithms and data structures. No other Java/GPU system can configure ~~the~~ expert features to the degree that our system can~~is able to~~.
2) Rootbeer is the most advanced system with regard to executing methods, fields and objects on the GPU.
3) Rootbeer has proven to work in practical systems and can be downloaded by anyone in the world, open-source. In 2013, Rootbeer ranked 2077th most popular by star count out of 10 million repositories in github, placing it in the top 0.02 percentile [8, 9] .
4) We have improved the Soot Java Optimization Framework by supplying a new class loader called the RTAClassLoader. This class loader is 10x faster than the existing

Soot class-loader and uses 8x less memory. These improvements can make infeasible analyses feasible on normal laptops or desktops. Soot is currently widely used for the analysis of Android malware and security of Java websites.

5) We ~~will~~ show a case study ~~of~~ using Rootbeer with an application from the Welch Lab and demonstrate a 102.7x speedup using Rootbeer over a single core using regular Java. This example uses a global synch primitive that deadlocks on all other competing systems.
6) We ~~will~~ give suggestions and insight to new projects such as OpenJDK Sumatra that wish to support Java on the GPU in industrial settings.

Rootbeer was developed using test-driven design (TDD) and contains a set of 103 high level test cases that can be run as built-in test. 77 of these test cases currently pass on Windows, Linux and Mac with Java 1.7. These test cases cover every aspect of the Java programming language, except the following aspects:

1) native methods
2) invokedynamic
3) reflection
4) garbage collection

Specifically, the features of the Java 1.7 language that are supported are listed below:

1) Using any composite object, method or field
2) Arrays of every primitive type of arbitrary dimension *
3) Arrays of any reference type of arbitrary dimension *
4) Instance and Static constructors, methods and fields

5) Efficient String manipulations using a custom CUDA StringBuilder class.
6) Exceptions that can be thrown and caught on the GPU. If an exception is thrown beyond the GPU boundary, it can be caught on the CPU by the developer
7) Classes with super and outer classes
8) Virtual methods
9) Dynamic memory allocation without garbage collection
10) The synchronized keyword on methods and objects
11) Printing to the screen for all primitive types and Strings from the GPU

* including the array length field and clone method

All of these features work together as a cohesive whole and arbitrarily complex combinations of them have been proven to work during our case study with the Welch Lab and in our performance examples. In addition, Martin Illecker from the University of Innsbruck has used Rootbeer in his Master's thesis to bring GPU computing to the cloud with Apache Hama and many people are using Rootbeer over the Internet.

1.3.1 Disclaimer

With all the great things about Rootbeer that we have just spoken about, we want to mention that it does have some faults. The major fault at this time is getting the code to compile due to class loading issues. Simple programs work well, but very complicated programs run into problems. In the Welch Lab example, we solved this by created a separate JAR file for all of the GPU code and compiled it separately from the CPU code. This JAR also requires a main method that makes calls to~~the~~ every Java method inside the GPU

JAR. After this was done, we have seen very few bugs with the actual serialization and CUDA code generation. We do not guarantee that Rootbeer works perfectly in industrial settings and cannot be held accountable for any defects in the software.

1.4 Welch Lab Case Study

In a case study with the Welch Lab, our~~the~~ research focused on accelerating Hidden Markov Model (HMM) computations using Rootbeer. The original requirements were to run a training phase for HMMs 308 times with each item requiring 39 minutes of computation time. Computing out the time, this takes approximately 8.3 core-days. We have accelerated this computation 102.7x and now we require 22 seconds each, giving an accelerated time of 1.992 hours total time. This computation is using 2240 HMM states with a signal length of 8000 doubles. The full signal is 62500 doubles and will benefit from this proven speedup. In addition, an analysis using wavelet transforms of raw image frames will more likely be possible using the HMM GPU code and a 16 GPU system. More details on this program are contained in Section 7.2.

1.5 Literature Review

Graphics Processing Units and the associated programming frameworks have a history starting around 1982. The very first GPUs supported acceleration of drawing lines and other simple graphics [3, 34, 40]. Using this hardware, the CPU could specify ~~the~~ endpoints of a line or ~~the~~ coordinates of a rectangle and the GPU would compute the pixels to fill and then fill them. While this was executing the CPU could complete other work.

GPU hardware continued to advance and in 1990 [10] it was shown that motion of a polygon shaped robot moving through a 2D space could be completed on the GPU using flood-fill while the CPU was running. This was one of the first times that the GPU hardware was used for computations other that drawing pixels directly to the display.

Between 1987 and 2006, gaming and CAD programs gave GPU manufactures a market to continually improve their architecture.

Around 1992, Wolfenstein 3D [11] was one of the original first-person shooter games created by John Carmack [12] and ID software. In the game, the players moved through a 3D world on the computer. This first version used ray casting to make a renderer that could effectively draw the world on an Intel 286 computer [13]. At about the same time, OpenGL 1.0 was first released for 3D gaming and CAD applications. OpenGL initiated an open standard for programmatically drawing 3D images on a frame buffer using a GPU.

The next game released by ID software was Doom in 1993 [14] and this game had more complicated graphics that could be drawn because of more powerful hardware. The heights of rooms could change, walls did not need to be rectangular, surfaces were texture mapped and complex lighting was used. Later, in 1996, Quake was released. The graphics were again improved and one year later, GLQuake was released. The first release of GLQuake by John Carmack would only run on a 3dfx Voodoo Graphics Card. Combining hardware accelerated OpenGL with transparent water and other lighting effects made the game quite popular. 3dfx went bankrupt in 2002 [15] and most of its assets were bought by NVIDIA. The source code to GLQuake was released in 1999 [16] and is available in the Ubuntu Repositories.

After the demonstration of transparent water using OpenGL in 1996, a researcher demonstrated that Voronoi diagrams could be computed using graphics hardware [17] in 1999. A similar idea to use the GPU for general purpose computation was published in 2002 with physics equations running on coupled map lattice cellular automata [18]. These several examples listed in this document show that early drawing primitives provided by graphics cards can do general-purpose computation. For more information regarding early uses of general-purpose programing on GPUs and a longer literature review, see the work by Dr. John Owens [19].

During the time between 2003 and 2007, improvements were made to OpenGL to support more advanced programmable shading.

In 2006, general-purpose computation on GPUs was enabled with a C programming language from NVIDIA called CUDA [20]. This was the first general-purpose language for GPGPU programming. Shortly before CUDA was released, AMD launched a GPGPU programming language, except that it was difficult to use because it was an assembly-based language. Later in 2009, a new programming language called OpenCL [21] was introduced that aimed to be a vendor-neutral programming language for GPUs and accelerators. Both AMD and NVIDIA devices now support the OpenCL programming language. However, Rootbeer only works with NVIDIA CUDA because some required features for Rootbeer are not supported in OpenCL.

In 2007, our work with tomography and CUDA began. We started by testing small applications on the GPU card and made sure that the computations were running on the hardware. Then we started to work on real applications. At this time we were programming in CUDA.

In 2009, JCuda [22] was released and allowed a developer to program GPUs from inside Java using provided JNI mappings to the NVIDIA driver. However, the developer still needed to convert complex Java objects into arrays of primitive types and write CUDA kernel code saved in a string. Later in 2009, JOCL [23] was released by the same author as JCuda with a similar API, except OpenCL was supported instead of CUDA. In 2011, JavaCL [24] was released and is similar to JCuda and JOCL except that is uses JNA rather than JNI generated using JNAerator.

Work began appearing in 2008 that attempts to execute Java on the GPU. In 2008, the Master's Thesis titled "Automatic Parallelization for Graphics Processing Units in JikesRVM" [25] focused on converting the outer loops of a method into a kernel using automatic parallelization. Here, known compiler algorithms where implemented and combined with original algorithms to make Java run on the GPU in parallel. The algorithms included are "Classification of Loops", "Identifying Loop Types" and more. Special treatment for Java was done for "Arrays", "Inter-array Aliasing", "Intra-array Aliasing", "Bound Checks", and "Recovering Control Flow". Much of the effort was spent on automatic parallelization and the support for advanced GPU programming features needed to support our HMM Learning example are not present. The code is not available for download and the author mentions that the code quality is prototype.

After the work with JikesRVM, in June 2010, Peter Calvert published his dissertation titled "Parallelization of Java for Graphics Cards" [26]. Again, the work focused on converting the outer loops of a method into a kernel using automatic parallelization. The user can annotate loops and the system can extract simple loops with simple increment variables. Not all loops can be extracted with this system. The author of this system implemented known compiler algorithms for “Code Graph”, “Type Inference”, “Dataflow Analysis”, “Increment Analysis”, “May-Alias”, “Loop Detection”, “Loop Trivialization” and “Dependency Analysis”. The work is available for download, but the Matrix Multiply test does not pass our correctness test, so the speed cannot be compared to Rootbeer. The authors of this work claims that the “developer does not need to consider the specification of their specific graphics card, or have knowledge of the threading model”. As in the JikesRVM work, our system takes the opposite approach and requires the developer to be aware of the programming model and hardware details to enable advanced GPU applications such as the HMM Learning example.

Later in July 2011, Aparapi [27] was released and is similar in programming ideology as to Rootbeer. However, Aparapi only allows the developer to configure global thread count and not both thread count and block count. This points to a fundamental misunderstanding of shared memory functionality and our shared memory test example failed to compile. We modified their compiler output for our shared memory test and made it pass. Aparapi matrix multiple using shared memory was slower than Aparapi without using shared memory and both were about 23x slower than Rootbeer matrix multiply using shared memory. Aparapi supports syncthreads and atomicAdd, however atomicAdd was implemented with Java synchronization, not CUDA atomic functions and is most likely slower because it is farther from the hardware. The CUDA atomicAdd, which Rootbeer uses, utilizes hardware in the GPU. Furthermore, Rootbeer provides support to configure register count along with supporting almost all of the atomic and synchronization functions from CUDA. In addition, Rootbeer has much better method, field and object support than Aparapi.

Rootbeer was released in 2012 [25] and is the only compiler in the related work that supports arbitrary execution on Java objects, methods and fields on GPUs. In addition, it is the most complete system for expert usage scenarios due to the support of shared memory, block / thread count and register count. Rootbeer supports using multiple GPUs within the same system using a context API and the cache configuration can be chosen as either PREFER_SHARED or PREFER_L1. Rootbeer contains support to use the CUDA atomic and synchronization functions listed in figure 1.5.1 below.

CUDA Atomic Function	**Description**
syncthreads	Synchronize threads within a block
syncthreadsCount	Synchronize threads within a block and return the number of threads whose predicate is non-zero
threadfenceBlock	All writes to shared and global memory before the instruction are seen before writes after the instruction for the current block
threadfence	All writes to shared and global memory before the instruction are seen before writes after the instruction for the current device
threadfenceSystem	All writes to shared and global memory before the instruction are seen before writes after the instruction for the current system
atomicAdd atomicSub	Reads an element from a Java array, adds or subtracts a given number, stores the result back into the Java array and returns the old value from the Java array.
atomicExch	Atomically exchanges two values in a Java array and returns the old value from the Java array
atomicMin atomicMax	Reads and element from a Java array, computes the min or max of the element and a given number and stores the result into the Java array.
atomicCAS	Executes an atomic compare-and-set operation using a Java array element, a compare value and a set value.
atomicAnd atomicOr atomicXor	Reads an element from a Java array, executes bitwise or, xor or and given a value and stores the result into the Java array. Returns the old value in the Java array.

Figure 1.5.1 Atomic Functions from CUDA that Rootbeer Supports

Rootbeer also supports removing exceptions and array bounds checks for speed once the program has been debugged. Primitive System.out.println is supported in Rootbeer for debugging purposes. Without the advanced features of configuring shared memory, thread / block count, register count and the atomic syncthreadsCount, our HMM Learning example that uses a global synchronization primitive will deadlock. Rootbeer is different from all other systems in that we aim to support advanced GPU algorithms and data structures. Programming with Rootbeer is as close as possible to programming with CUDA, except the developer has the additional support of Java object execution and serialization support. We find that by making the programming model as close to CUDA as possible, the most number of advanced GPU algorithms and data structures run with Rootbeer. Rootbeer's is categorized as production-beta, has extensive testing, and is freely available for download.

Shortly after Rootbeer was released, Oracle announced project Sumatra [21] aiming to support GPUs from within the JVM. They were first working to compile Java Bytecode into PTX assembly and then were working on strategies for selection of which computations to offload to the GPU. Recently, however, Oracle and the OpenJDK community has stopped work on this project [28].

In August 2015, a paper was published titled "Boosting Java Performance Using GPGPUs" [29] that describes a framework called "Java Acceleration System" or JACC. This system is not available for download and the code is categorized as experimental in the paper. This system uses *tasks* and *task graphs* that are specified by the developer to assist code generation and has comparable performance to Rootbeer. However, the advanced GPU features such as shared memory, block / thread count, register usage and atomic functions are not supported, so examples like our HMM Learning example will most likely deadlock.

In September 2015 a paper was published titled “HJ-OpenCL: Reducing the Gap Between the JVM and Accelerators” [30] that focuses on automatic generation of OpenCL kernels and JNI glue code from a parallel for-all construct available from Habanero-Java (HJ). In addition, the work uses HJ’s array_view language construct to efficiently support rectangular, multi-dimensional arrays on OpenCL devices. In Peter Calvert’s Java-GPU, the problem of multi-dimensional arrays in low level Java Bytecode required custom algorithms to handle aliasing. Here the authors are requiring a language construct to be added around rectangular, multi-dimensional arrays if they are to be integrated with the “parallel for-all” type automatic parallelization. Rootbeer does not do automatic parallelization, so finding aliases of multi-dimensional arrays is not a problem and both rectangular and jagged multi-dimensional arrays are easily supported. The system uses the OpenCL code generation from Aparapi, therefore only two types of atomic and synchronization functions are supported (syncthreads and atomicAdd). Again, this system does not address the advanced GPU features such as shared memory, block / thread count, register usage and many atomic functions and examples like our HMM Learning example will most likely deadlock.

There are a number of libraries to use GPUs including ArrayFire, MATLAB, cuDNN, cuFFT, NPP, cuBLAS and NVBIO. These are summarized in Figure 1.5.2 below.

System / Library	**Description**
ArrayFire	Open source library with many built in functions for math, signal and image processing. The company is also a vendor for GPU consultancy services.
MATLAB	MATLAB includes GPU accelerated versions of built in functions and Mathworks also distributes a number of toolboxes that contain custom kernels that accelerate common computations for many industries. MATLAB also allows a developer to run custom CUDA code from within a MATLAB script.

cuDNN	NVIDIA GPU Deep Neural Network Library
cuFFT	NVIDIA GPU Fast Fourier Transform Library
NPP	NVIDIA Performance Primitives for Image and Signal Processing
cuBLAS	NVIDIA GPU BLAS Library
NVBIO	NVIDIA Accelerated C++ Framework for High-Throughput Sequence Analysis

Figure 1.5.2 – Selected Libraries and Systems Supporting GPUs.

When CUDA was first released, developers found that they needed to think in parallel to obtain significant speedups using GPUs. Early work demonstrated algorithms such as parallel reduce to solve problems such as array summation. Since then, more complicated parallel GPU algorithms and data structures have been published that can accomplish more complicated processing. Examples of these are sorting, hashtables and GPU garbage collection using parallel prefix sum. Listed below is a summary of the new parallel GPU algorithms and data structures adapted from three leading authors: Sean Baxter, Duane Merrill and John Owens.

Algorithm	**Description**
Reduce	Sum the elements of an array into a single value in parallel. [31]
Scan	Execute an operation on each element. Prefix Sum computes the sum of the current number plus the previous sum in parallel. [32], [33] and [34]
Histogram	Organize a sequence into bins such that the frequency of a range of values is counted
Bulk Remove	Remove elements in an array specified by indices [31]
Bulk Insert	Insert elements from one array into another array specified by indices [31]
Merge	Merge two arrays in parallel [31]
Mergesort	Sort an array in parallel O(nlgn) [31]
Radixsort	Sort an array in parallel O(kn) [35]
Vectorized Sorted Search	Run concurrent searches in parallel [31]
Interval Expand	Given counts and values, replicate each value count times and emit to the output [31]
Segmented Reduce	Parallel reduction over many irregular-length segments [31]

Graph BFS Traversal	Traverse a sparse graph using breadth first search [36]
GPU Hashtable	Store elements in a set or map on the GPU with fast lookup and insert [37]
GPU Garbage Collection	Use prefix sum or histogram to accelerate garbage collection on GPUs [38]

Figure 1.5.3 – Summary of some selected parallel GPU algorithms and data structures

These algorithms can be implemented using Rootbeer ~~given~~ provided primitives and fused with sections of single-threaded CPU code to solve problems in areas such as medical imaging, image processing and bioinformatics along with many other diverse fields. Without using specific parallel algorithms a developer will often see a performance speedup of around a range of 10x for problems with little inherent parallel work. The developer must use some advanced parallel algorithm or data structure to achieve 100x speedups for all but the simplest problems. Given 448 cores in a Tesla C2050 device, a developer can approximately estimate 100x best-case speedup as part of resource planning to determine the number of GPUs that are needed in a multi-GPU system assuming the problem can be written in terms of parallel algorithms. Obtaining a speedup on the order of 400x to 1000x is most likely infeasible with a Tesla C2050 based on the author's experience.

1.6. Summary of Results

To summarize the performance of Rootbeer, the time to execute matrix multiply on a two-dimensional array for a small set of systems is given in Figure 1.6.1 below. An in-depth discussion of the Rootbeer Shared example in the below study along with other performance results are listed in Chapter 7. The matrix multiply tested here was multiplication of two 2048x2048 matrices of type float. Figure 1.6.2 shows the test setup used. Each test case was run eight times and the best elapsed time was taken to allow the JVM to effectively optimize code. The JVM can optimize methods for increased performance, but it can only use the optimized code after returning from the method. Long running methods, like matrix multiply, need to return from the method a few times to obtain best performance.

Performance Test Case	**Elapsed Time**	**Relative Speed**
Java4	27129 milliseconds	1x
Java4 Transpose	3519 milliseconds	7.7x
Rootbeer	636 milliseconds	42.7x
Rootbeer Transpose	7379 milliseconds	3.8x
Rootbeer Shared	286 milliseconds	94.9x
CUDA	376 milliseconds	72.2x
CUDA Transpose	7341 milliseconds	3.7x
CUDA Shared	94 milliseconds	288.6x
Aparapi	2205 milliseconds	12.3x
Aparapi Transpose	8937 milliseconds	3.0x
Aparapi Shared	2956 milliseconds*	9.2x

Figure 1.6.1 – Summary Performance of Rootbeer and Competing Systems. (*) required manual editing of generated opencl code to pass.

Test System	Specifications
CPU	4 core Xeon E5405 @ 2.00GHz with 16GB ram (DDR2 @ 667 MHz)
GPU	448 core Tesla C2050 @ 1.147 GHz with 3GB (384-bit) ram over PCI-e x16 Gen2

Figure 1.6.2 – Test Setup used for Performance Comparisons

You can see in the above example that Rootbeer Shared is 94.9x faster than Java4 and CUDA Shared is 288.6x than Java4. Rootbeer is built on top of CUDA, so it makes sense at this point that it is a bit slower. Aparapi is built on top of OpenCL but does not show significant speedups over Java4 in our tests. Aparapi Shared is an example that attempted to use OpenCL shared memory, but we needed to manually edit the generated OpenCL and modify the Aparapi source code to make the test case pass. Aparapi Shared does not show significant speedup over Java4. A full description of each test case is listed with bullets below.

- **Java4 –** Using four Java threads on the CPU desktop system to calculate the result of multiplying two 2048 by 2048 float matrices
- **Java4 Transpose –** Similar to Java4, except that a transpose is done on the B matrix before multiplying to improve cache effects. The transpose time is not included in the timings.
- **Rootbeer –** Use a simple Rootbeer programming setup to multiply 2048 by 2048 float matrices. No optimizations have been done, but parallelism has been used.
- **Rootbeer Transpose –** Similar to Rootbeer, except that a transpose is done on the B matrix.
- **Rootbeer Shared –** A highly optimized version of matrix multiply using Rootbeer and the shared memory (software defined cache).
- **CUDA –** A matrix multiply written in CUDA to compare the results of raw CUDA to Rootbeer/CUDA.
- **CUDA Transpose –** Similar to CUDA, except that a transpose is done on the B matrix. Again, the time to transpose is not taken into account.
- **CUDA Shared –** A highly optimized version of matrix multiply using CUDA C and shared memory.
- **Aparapi –** Use Aparapi to multiply two 2048 by 2048 matrices with no special optimizations.

- **Aparapi Transpose –** Similar to Aparapi, except that a transpose is done on the B matrix before multiplying.
- **Aparapi Shared –** Using shared memory with Aparapi to compare Rootbeer and raw CUDA to Aparapi.

Chapter 2

Technologies

2.1 Introduction

In order to understand how Rootbeer works, you must be familiar with the technologies involved. Rootbeer uses the following technologies:

1) Java Programming Language
2) Java Bytecode
3) Soot Java Optimization Framework
4) Java Native Interface
5) GPU Architectures
6) CUDA Programming

The code for Rootbeer is written in the Java Programming Language (Java). It starts out by parsing Java Bytecode Class Files (class files) using the Soot Java Optimization Framework (Soot). To complete whole program analysis using less memory, class files are loaded using the custom Rootbeer Class Loader. Once the classes are loaded, Rootbeer uses the API of Soot along with Rootbeer code to generate serialization bytecode and CUDA code. Once everything is generated, it is packed into an output Java Archive (jar) including the Rootbeer Runtimes. When the program is run, Java Native Interface (JNI) is used to access the Rootbeer CUDA Runtime from within Java. All of these technologies will be discussed in the first part of this chapter and then an introductory view of GPU architectures and CUDA programming will be discussed.

During the introduction to CUDA, it will be shown how Rootbeer gives assistance over CUDA and OpenCL. In Chapter 3 Rootbeer programming will be studied in detail.

2.2 Java Programming Language

Rootbeer primarily is written in Java using Soot and the Rootbeer Class Loader. There is a small amount of JNI and CUDA code. The figure below shows the number of lines of code for each language.

Component	**Lines of Code**
Rootbeer Product (Java)	15225
Rootbeer Product (JNI)	626
Rootbeer Product (CUDA)	3137
RTAClassLoad (Java)	3502
Rootbeer Testharness (Java)	12214
Total	**34704**

Figure 2.2.1 – Source Lines of Code Count in Rootbeer using SLOCCount

At this point, those proficient with the Java Programming Language or those more interested in the GPU aspects and contributions of this dissertation may skip to Section 2.4. Rootbeer can still be understood without reading the intervening sections.

2.2.1 Java Programming Language Details

The Java Programming Language was released in 1995 from Sun Microsystems under the direction of James Gosling [39]. It has since become the second most popular programming language in the world according to Langpop [40]. It has syntax similar to C/C++. Instead of compiling directly to assembly, Java compiles to Java Bytecode, which is run in a Java Virtual Machine (JVM). The JVM and associated libraries abstract away the operating system and processor so that a developer can write the code once and run the same binaries on Windows, Linux and Mac. The Java Programming Language is also used in the Android Operating System and some embedded devices, but Rootbeer does not run in these environments.

The Java Programming Language is statically typed with all objects deriving from java.lang.Object. Generics are supported through type erasure inside the Java Compiler (javac). Type erasure in Java is in contrast to the way generics are supported in C#. In Java, the generic types are checked by the javac compiler but then erased and replaced with instances of java.lang.Object in the bytecode. In C#, the generic types are compiled into the bytecode. Sun introduced generics into Java after the language had been released with a large standard library. Type erasure allowed them to reuse the entire standard library. On the one hand type erasure can enable easier reflection code and reuse of a standard library while on the other hand compiled types as in C# can offer better static analysis.

Java cannot link directly with existing C/C++ code because the JVM only runs bytecode. It does this to keep the bytecode language simple and pure. Also, it allows the JVM to do more verification on the safety of the bytecode with respect to security and functionality. Such verifications include 1) a program cannot have a buffer overflow, 2) a program cannot make a pointer to an arbitrary, un-typed memory location without using unsafe code and 3) strong typing enforcement. We believe that due to the third aspect (strong typing) that Java is a good language for compiler research and prefer doing compiler work in Java over other languages like Python or JavaScript.

To link native code with managed Java code, a developer must use JNI or one of the libraries built on top of JNI. Rootbeer has it's own runtime that uses JNI and links to the CUDA GPU drivers to allocate GPU memory, copy memory and launch GPU tasks.

Below is a small Java program that is used to print the numbers zero through nine. We will compile this with javac and in the next section show the Java Bytecode for it.

```
public class PrintNumbers {
  public void print(){
    for(int i = 0; i < 10; ++i){
      System.out.println(i);
    }
  }
}
```

Figure 2.2.1.1 – Source Code for PrintNumbers.java

After Java Bytecode, the discussion will move to JNI and then proceed to the Soot Java Optimization Framework, the Rootbeer Class Loader and GPU architecture and programming.

2.3 Java Bytecode

Those familiar with Java Bytecode or have more interest in the contributions of this dissertation, please skip to Section 2.4. Rootbeer can still be understood without reading Section 2.3.1.

2.3.1 Java Bytecode Details

The Java Compiler converts Java Programs into Java Bytecode. Java Bytecode is a typed, stack based assembly language that is run inside of the JVM. The Hotspot Virtual Machine [41] starts running bytecode in an interpreter and when it notices that a method has executed many times or for a long duration, it compiles the method to assembly. This assembly code is then run directly on the CPU next time the method is needed.

A Java Bytecode file has three parts: 1) the header and class information, 2) the constant pool and 3) the methods. We will show short examples of all of these in this section's tutorial. Appendix B contains a full listing.

It is important to note that the bytecode files are binary formatted and we are viewing them in human readable form after using javap. The command to generate the text-based format is also listed in Appendix B.

You can see the full header and class information for our PrintNumbers.java file in Figure 2.3.1.1 below. The header includes the source file, the name of the class and the super class. It contains the major and minor version of the minimum Java Virtual Machine that this bytecode file was compiled for.

```
Compiled from "PrintNumbers.java"
public class PrintNumbers extends java.lang.Object
  SourceFile: "PrintNumbers.java"
  minor version: 0
  major version: 50
```

Figure 2.3.1.1 – Decompiled Java Bytecode Header and Class Information for PrintNumbers.java

After the header and class information, the bytecode file contains the constant pool. The constant pool for PrintNumbers.java is shown in Figure 2.3.1.2. It is an array of items that are all either a number, a string or possibly typed composite combinations of strings called methods, fields or classes. The numbers can be ints, longs, floats or doubles that would appear in the source code. There are no number constants in this constant pool.

Constant #1 (line 2) is a method identifier. When this method is called from the bytecode assembly, the file will contain an identifier specifying that the instruction is a method call with identifier equal to constant #1. In the binary format, the constant starts out with a type identifier for method and then contains two integers. One integer is to reference the declaring class of the method and the other is for referencing the method signature.

```
001    Constant pool:
002  const #1 = Method #5.#14; //java/lang/Object."<init>":()V
003  const #2 = Field  #15.#16;
         //java/lang/System.out:Ljava/io/PrintStream;
004  const #3 = Method #17.#18; //java/io/PrintStream.println:(I)V
005  const #4 = class  #19; //PrintNumbers
006  const #5 = class  #20; //java/lang/Object
007  const #6 = Asciz  <init>;
008  const #7 = Asciz  ()V;
009  const #8 = Asciz  Code;
010  const #9 = Asciz  LineNumberTable;
011  const #10 = Asciz print;
012  const #11 = Asciz StackMapTable;
013  const #12 = Asciz SourceFile;
014  const #13 = Asciz PrintNumbers.java;
015  const #14 = NameAndType #6:#7; //"<init>":()V
016  const #15 = class #21; //java/lang/System
017  const #16 = NameAndType #22:#23; //out:Ljava/io/PrintStream;
018  const #17 = class #24; //java/io/PrintStream
019  const #18 = NameAndType #25:#26; //println:(I)V
```

```
const #19 = Asciz PrintNumbers;
const #20 = Asciz java/lang/Object;
const #21 = Asciz java/lang/System;
const #22 = Asciz out;
const #23 = Asciz Ljava/io/PrintStream;;
const #24 = Asciz java/io/PrintStream;
const #25 = Asciz println;
const #26 = Asciz (I)V;
```

Figure 2.3.1.2 – Decompiled Java Bytecode Constant Pool for PrintNumbers.java

We can see from the constant pool listing that the method constant 1 has a bytecode signature of “java/lang/Object."<init>":()V”. Shown in Figure 2.3.1.3 below is the method signature pulled apart.

Constant #5 is a class with a name of constant #20, which is a string containing “java/lang/Object”. Therefore this method is a method that is part of the java.lang.Object class. The other part of the signature contains Constant #14, which is a NameAndType that links to #6 and #7. Constant #6 is a string “<init>” and Constant #7 is a string “()V”. The method name “<init>” is a special method name for a constructor. The string “()V” is a method type signature that specifies that the method takes no arguments “()” and returns void “V”.

Overall, constant #1 references the void constructor of java.lang.Object. It is worthwhile to note here that strings in the Java class file format first begin with an integer specifying the length of the string followed by the actual string. This is beneficial over using null terminated strings while parsing class files because a single buffer can be allocated once for the string and the string never needs to be copied from a larger buffer.

#1 = Method #5.#14 = “java/lang/Object."<init>":()V”	
#5 = class #20;	#14 = NameAndType #6:#7;

#20 = Asciz java/lang/Object;	#6 = Asciz <init>;	#7 = Asciz ()V;

Figure 2.3.1.3 – Java Bytecode Method Signature Chart

There are more types to be aware of than what have been included in this small example. We show the full listing from Oracle's website [42] in the table below. The types for java.lang.String and double array are examples. L and [are special first characters. After an L any class named can be listed, followed by a semicolon and after a [and number of ['s can be included followed by a regular type identifier naming either a primitive or reference type.

Signature	**Type**
Z	Boolean
B	Byte
C	Character
S	Short
I	Integer
J	Long
F	Float
D	Double
Ljava/lang/String;	java.lang.String
[D	One Dimensional Double Array
[[D	Two Dimensional Double Array

Figure 2.3.1.4 – Java Bytecode Types

Each class file is an independent entity that is produced by javac. The javac compiler verifies that the referenced strings exist in the class path and these strings are verified again when the JVM loads the code for execution. If they have been changed in an invalid way, the JVM will throw a specific exception that tells what the problem is. The Rootbeer Class Loader has the ability to remap calls to a class by changing the strings in the class files. This allows Rootbeer to substitute class files that contain native code to pure Java versions that can be transformed into CUDA code.

Now that we have seen the constant pool and have learned a little bit about the constant strings, let's look at some bytecode assembly. Below in Figure 2.3.1.5 is the constructor for PrintNumbers. Javac created this automatically for us. At line 3 the decompilation by javap shows that there is a max stack size of one, one local and one argument.

In the bytecode assembly, there are no arguments to methods. Arguments are passed into a method by the JVM through locals. Locals in bytecode are untyped and are global to a method. They are referenced through numbers and can have different types at different times, if a single local is reused.

The zeroth local in an instance method is the self-reference for the current class (this pointer). The zeroth argument to the method is passed into the first local, and the first argument is passed into the second local, etc.

On line 4 in the assembly below, there is a bytecode instruction "aload_0". This instruction loads the object from local zero onto the data stack. At this point, local zero is still the "this" pointer for the currently running class. Then on the next line the code specifies "invokespecial #1". This instruction will take the current object from the top of the stack (the "this" pointer) and call the method #1 (java.lang.Object constructor). Then return is executed. That is a simple constructor calling the super class constructor. Notice again that in the actual bytecode, the objects in the locals and stack are untyped. Rootbeer does not directly operate on Java Bytecode; instead it uses the Soot Jimple IR, which gives much more information. Jimple will abstract away the data stack and provide a typed, three-address, form. We will show this later in Section 2.4.

```
public PrintNumbers();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial     #1; //Method java/lang/Object."<init>":()V
   4: return
```

Figure 2.3.1.5 – Decompiled Java Bytecode for PrintNumbers Constructor

To get a better idea about how bytecode works, there is a longer example below. This example prints the numbers zero through nine to the console. We have commented the instructions in a chart in Figure 2.3.1.7. You can read Figures 2.3.1.6 and 2.3.1.7 carefully and notice that there are no loops, only ifs and jumps.

```
public void print();
  Code:
   Stack=2, Locals=2, Args_size=1
   0: iconst_0
   1: istore_1
   2: iload_1
   3: bipush    10
   5: if_icmpge 21
   8: getstatic #2; //java/lang/System.out:Ljava/io/PrintStream;
   11: iload_1
   12: invokevirtual #3; //java/io/PrintStream.println:(I)V
   15: iinc 1, 1
   18: goto 2
   21: return
```

Figure 2.3.1.6 – Decompiled Java Bytecode for PrintNumbers Print Method

Location	**Instruction**	**Explanation**
0:	iconst_0	Load an integer constant zero onto the data stack
1:	istore_1	Store the integer from the top of the data stack into local 1 (this is the index variable for the loop)
2:	iload_1	Load the integer from local 1 onto the top of the data stack
3:	bipush 10	Push a signed, 8-bit byte onto the stack. This is the 10 used to specify the end of the loop iteration
5:	if_icmpge 21	Pop two elements from the stack. If the first one is greater than or equal to the second one, jump to location 21.
8:	getstatic #2	Get the static field System.out and place it on the data stack

11:	iload_1	Load the integer from local 1 onto the top of the data stack
12:	invokevirtual #3	Pop the arguments and object from the stack. Call println on the System.out object passing in the argument.
15:	iinc 1, 1	Increment the integer in local one by one count
18:	goto 2	Jump to location 2
21:	return	Return from the method

Figure 2.3.1.7 – Explanation of Print Bytecode Assembly

We have seen in this section a short tutorial on Java Bytecode. The research mentioned that Rootbeer operates on a higher level IR called Jimple provided by Soot. The next section is devoted to describing Soot and that IR.

2.4 Soot Java Optimization Framework

If you are familiar with the Soot Java Optimization Framework or are more interested in the GPU aspect of this dissertation, please skip to Section 2.6. However, if you want to understand the RTAClassLoader that was added to Soot as part of this dissertation, it may help to read Section 2.4.1.

2.4.1 Soot Java Optimization Framework Details

The Soot Java Optimization framework [43] can read Java Bytecode and convert it into Jimple. Jimple is a typed, three-address, intermediate representation. To obtain Jimple, Soot abstracts away the stack, creates new locals representing stack locations and then flows types from known locations into unknown locations using a fixed point. We discovered this while fixing some bugs in Soot that were important to Rootbeer. A developer can then do analyses on the higher-level form that contains more type information. After analysis, the Jimple can be translated back to Java Bytecode and run in a Java Virtual Machine. Compared to two other Java Bytecode analysis frameworks (BCEL [44], ASM [45]), Soot is the only system that can provide a typed, three-address form that can be manipulated. Compared to WALA [46] and JChord [47], Soot has a more active community and a larger analysis library. Soot is widely used by security researchers studying Android bytecode and Java based websites. We have added a new class-loader to Soot that is 10x faster so our compilation of whole-programs can complete in a reasonable time. Please see Chapter 4 for details on the new class-loader, optimizations done and performance results. During this dissertation, this research contributed nine bug fixes that have been accepted into the Soot master branch.

Rootbeer uses Soot to inspect Jimple to find instances of the Kernel interface and cross-compiles instructions to CUDA C. It then searches for types used on the GPU and generates serialization Jimple code to move Java State to GPU memory. The Jimple is translated down to Java Bytecode and packed in an output Java Archive (jar).

Soot provides a programmatic API for manipulating Jimple from Java and also can emit a text-based form. Below is an example of PrintNumbers.jimple.

At line 1 you can see that the file declares a public class that derives from java.lang.Object. There are two methods, the constructor and “print”. Notice that in the constructor at line 5, the local has a name (r0) and a type (PrintNumbers). This is an enhancement from what is found in the Java Bytecode. In the bytecode there are no types for locals but they are added by Soot for the Jimple format.

```
001 public class PrintNumbers extends java.lang.Object
002 {
003   public void <init>()
004   {
005     PrintNumbers r0;
006
007     r0 := @this: PrintNumbers;
008     specialinvoke r0.<java.lang.Object: void <init>()>();
009     return;
010   }
011
012   public void print()
013   {
014     PrintNumbers r0;
015     int i0;
016     java.io.PrintStream $r1;
017
018     r0 := @this: PrintNumbers;
019     i0 = 0;
020
021   label0:
022     if i0 >= 10 goto label1;
023
024     $r1 = <java.lang.System: java.io.PrintStream out>;
025     virtualinvoke $r1.<java.io.PrintStream: void
026       println(int)>(i0);
027     i0 = i0 + 1;
028     goto label0;
029
030   label1:
031     return;
032   }
033 }
034
```

Figure 2.4.1.1 – Decompiled Jimple for PrintNumbers.java

You can see the following identity statement at line 7. Identity is different than assignment because of the way method arguments are handled in the bytecode. In Jimple, the first identity statement in a method must reference this and all arguments must be assigned to locals. This is to make lowering to Java Bytecode easier. (Remember there are no parameters in bytecode, everything is passed as locals, so Soot's requirement is due to this).

```
r0 := @this: PrintNumbers;
```

Figure 2.4.1.2 – Identity Statement in Jimple

The call to the super class constructor from line 8 is listed below. This time the stack has been abstracted away and you can clearly see that the specialinvoke is invoked on r0. The method signatures are also different than Java Bytecode signatures.

```
specialinvoke r0.<java.lang.Object: void <init>()>();
```

Figure 2.4.1.3 – Special Invoke in Jimple

The method signature has the 1) declaring class, 2) return type, 3) method name and 4) parameter types. The method signature is broken down in Figure 2.4.1.4 below.

`<java.lang.Object: void <init>()>`	
1) Declaring Class:	java.lang.Object
2) Return Type:	void
3) Method Name:	<init>
4) Parameter Types:	none

Figure 2.4.1.4 – Method Signature for Object Constructor in Jimple

There are a few more things to talk about for this Jimple file. First, the "if" statements explicitly show the operands and locations have been substituted with labels. Second, there is a field reference and assignment to local $r1 shown below.

```
$r1 = <java.lang.System: java.io.PrintStream out>;
```

Figure 2.4.1.5 – Field Reference in Jimple

The statement is an AssignStmt with left hand and right hand sides. The left hand side is a local named $r1 with '$' to note that the local was pulled from the data stack. The right hand side is a field reference. The field reference signature contains the 1) declaring class, 2) the field type and 3) the field name. This is shown in detail in the figure below. All of the information for a field is directly in Java Bytecode, no additional analysis besides formatting is done in Soot.

`<java.lang.System: java.io.PrintStream out>`	
1) Declaring Class:	java.lang.System
2) Field Type:	java.io.PrintStream
3) Field Name:	out

Figure 2.4.1.6 – Field Signature for System.out in Jimple

When writing out to Java Bytecode, any field or method signature that may have had its signature changed will automatically be updated in the new constant pool.

The abstraction of the stack into typed locals and methods with parameters makes the Jimple IR much easier to work with than Java Bytecode. This research started out creating Rootbeer with Apache Byte Code Engineering Library (BCEL) [44] in 2008. BCEL leaves the instructions as low-level bytecode assembly. It was quickly determined that we needed more information about static types and Soot has served this purpose very well.

While we are still on the topic of Java Bytecode, we will quickly talk about Java Native Interface (JNI). You will see that Java Bytecode signatures and types are identical to the API provided to access Java objects from C programs. After completing the introduction to JNI, then CUDA programming and the NVIDIA GPU architectures will be covered.

2.5 Java Native Interface

If you are familiar with the Java Native Interface or are more interested in the GPU aspect of this dissertation, please skip to Section 2.6. You can still understand Rootbeer without reading Section 2.5.1.

2.5.1 Java Native Interface Details

Java Native Interface (JNI) allows a Java Program to access custom C methods from within Java. Java Bytecode is completely pure: the only thing existing in the assembly language format beyond the needs for pure computation is a simple flag specifying that a method is native. There are no special assembly instructions for accessing the world outside of the Virtual Machine. Support for things like the operating system, file system and sockets are provided by handwritten native methods provided by the Java Runtime Classes.

Rootbeer uses JNI to access the CUDA API. With JNI you must compile a binary for each operating system and architecture. Rootbeer compiles binaries for windows/linux/mac supporting x86 and x86_64 architectures and includes them in the Rootbeer.jar distribution. Rootbeer selectively unpacks the appropriate binary at runtime and stores it in the hidden rootbeer folder in the user's home directory.

To start programming with JNI, a developer will write Java code that contains at least one method decorated with the native keyword. A simple Java program that prints a string using a native method is shown below. It is advisable to keep native methods private with a public wrapper in the case that the public interface changes.

```
public class PrintString {

  private native void print(String str);

  public static void main(String[] args){
    PrintString printer = new PrintString();
    printer.print(args[0]);
  }
}
```
Figure 2.5.1.1 – Print Strings using a Native Method in Java

The above example is compiled with javac to produce a Java Class File (PrintString.class). Then we run the class file through javah to produce the header file below (the full command is in Appendix C.)

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class PrintString */

#ifndef _Included_PrintString
#define _Included_PrintString
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     PrintString
 * Method:    print
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_PrintString_print
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```
Figure 2.5.1.2 – Java Native Interface Header File Generated with javah

After the header file is created, we create the source C file shown below. You can see at line 8 that we use GetStringLength provided by JNI to get the length of the Java String (typed as jstring). Then on the next line we use malloc from the standard C library. Once we have allocated an array of the appropriate size, we use GetStringUTFRegion (again from JNI) to transfer a region of the Java String to our native string. Then we print the string and finally free the native memory.

```
001 #include "PrintString.h"
002 #include <stdio.h>
003 #include <stdlib.h>
```

```

JNIEXPORT void JNICALL Java_PrintString_print(JNIEnv * env,
jobject thisref, jstring str)
{
  int len = (*env)->GetStringLength(env, str);
  char * nstr = (char *) malloc(sizeof(char) * len);
  (*env)->GetStringUTFRegion(env, str, 0, len, nstr);
  printf("%s\n", nstr);
  free(nstr);
}
```

Figure 2.5.1.3 – Java Native Interface Source Code to Print a String to the Console

After building a shared object, dylib or dll, (Appendix C) one more thing needs to happen before running JNI code from Java. Change the PrintString Java code to load the native library. This is shown in Figure 2.5.1.4 below. Notice at line 6 a new File object is made pointing to "print_string.dylib" and then it loads the file on the next line with System.load. An absolute path is required for System.load and the File object can return this from a relative path.

```
001 import java.io.File;
002
003 public class PrintString {
004
005   public PrintString(){
006     File library = new File("print_string.dylib");
007     System.load(library.getAbsolutePath());
008   }
009
010   private native void print(String str);
011
012   public static void main(String[] args){
013     PrintString printer = new PrintString();
014     printer.print(args[0]);
015   }
016 }
```

Figure 2.5.1.4 – Loading a Native Library in Java

Up to this point, a complete example has been shown that can print a Java String using native strings and libraries. Rootbeer also calls Java methods from JNI and an example of this without using method caching is shown below. The Rootbeer native driver caches methods by saving method descriptors for later use.

```
jclass mem_class;
jmethodID get_address_method;
jlong * cpu_object_mem;

mem_class = (*env)->FindClass(env,
  “org/trifort/rootbeer/runtime/FixedMemory");
get_address_method = (*env)->GetMethodID(env, mem_class,
  "getAddress", "()J");

cpu_object_mem = (void *) (*env)->CallLongMethod(env,
  object_mem, get_address_method);
```

Figure 2.5.1.5 – Java Native Interface Code to Call a Method

We also included in the table below “jobject”. This is the JNI type for a Java Object. Our code above uses FindClass to obtain the jclass. Once we have the jclass we use GetMethodID to obtain the jmethodID. Notice how the class descriptor passed into FindClass and the method name and signature passed into GetMethodID are the same protocol used inside the Java Bytecode Constant Pool discussed in Section 2.3.

Type	**Description**
jclass	Represents a Java Class. Once the class description object has been retrieved from inside the JVM, we can use it to get a jmethodID
jmethodID	Represents a Java Method. Once the method identifier object has been retrieved, we can use it to call a method.
jlong	We keep handles to native buffers inside long fields in Java. A 64bit long can be cast to any pointer type on both 32 and 64 bit systems.
jobject	Represents a Java Object accessible from JNI to call methods and retrieve fields.

Figure 2.5.1.6 – Java Native Interface Types used in Rootbeer

Included in the JNI API are methods like CallVoidMethod, CallObjectMethod and CallByteMethod to distinguish the return type of the Java Method. There is one method per Java type. Also there are methods to get and set fields, among other things.

This wraps up the discussion on Java. Two very small examples of the quality have been seen: strings are stored smartly and class files use the same signatures as JNI. While improving Soot, we looked at the code of the Hotspot JVM and it was clear that there are many other examples of quality in the JVM. The 500k lines of code program are very clean and our impression is that Hotspot has some of the most expert compiler engineers in the world.

Now the talk turns to GPU Architectures and CUDA programming. After seeing how to program GPUs with CUDA, the document shows the improvements that Rootbeer brings in Chapter 3.

2.6 GPU Architectures

When we first began evaluating architectures to accelerate our tomography application we studied GPUs, FPGAs and the Cell Broadband Engine. The problem had many floating-point calculations and existing FPGAs could only fit a handful of floating point units on a single device. Therefore it was concluded that FPGAs were not suitable for high performance floating point calculations. The Cell Broadband Engine was studied and it was found that a 6x speedup easily could be obtained with the possibility of more if the 128-bit vector instructions were used. The physicists we worked with were looking for a speedup of around 100x with floating point data so we decided to focus attention to GPU processors because they had more non-vectorized floating point units to attempt to support this.

The discussion will focus on NVIDIA GPU architectures and optimization because of the in-depth work we have accomplished using these devices. Specifically, we will be talking about an NVIDIA Tesla C2050 GPU, the device purchased. A block-level diagram of the C2050 is shown in Figure 2.6.1 below. The CPU acts as the host to the GPU, which acts as the device. Before launch, data is transferred across the PCI-express bus from the CPU RAM to the GPU RAM. Once the needed data is in the GPU RAM, the CPU host issues control signals to the GPU device driver that specifies the GPU entry point and how many blocks and threads to use for the current computation.

The Tesla GPU receives these commands, loads the instruction code from GPU RAM and starts the computation. The Tesla C2050 GPUs have 14 Streaming Multiprocessor (SM) units, each with 1 Instruction Fetch (IF) unit and 8 arithmetic/logic cores. The device fetches one instruction and executes it on 8 arithmetic/logic cores at the same time, 4 times. This gives the view that there are 32 threads running on a single SM at a time and, in reality, that is pretty close to the fact. Eight are running at once and over four clock cycles one instruction fetched is run 32 times. Multiplying the 32 threads by 14 SMs gives 448 cores as published in NVIDIA literature.

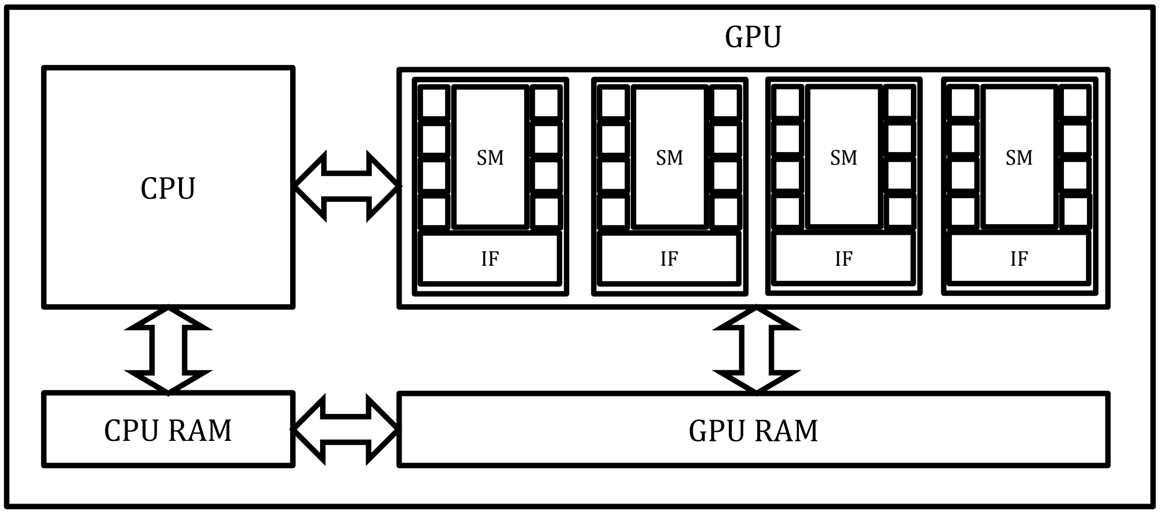

Figure 2.6.1 – High-Level Architecture of NVIDIA GPUs

The 32-thread warp size matches nicely with the number of memory banks. Global GPU RAM is banked into 32 banks that are each 4-bytes wide. This is designed so that each thread in a warp can fetch a 4-byte integer or float from a memory location proportionally related to its thread index. In this way, all threads within a warp can issue requests at the same time. Otherwise, if the threads in a warp are not fetching in an aligned way, bank conflicts cause the memory requests to become serialized at the bank controller.

The Tesla C2050 has 32768 registers (32-bit) per SM and 49152 bytes of shared memory. The software developer specifies the number of threads per block, which will be a group of threads sharing the shared memory. Once a block is placed on the SM for execution, it cannot be removed until the block is complete. There is no preemption by the scheduler and no yield keyword available in CUDA. In order for our global synchronization primitive to work in our HMM example (section 7.2), we needed to make sure that our 160 threads fit into the SM at the same time. If not enough registers or shared memory were available to fit all blocks on all devices, then the latter threads would never signal in the barrier sync and deadlock would occur. Preemption was not included in the GPU because it would require even more registers to hold threads that could be waiting to run and then less die area would be available for arithmetic/logic units.

Once the thread and block count has been optimized for register and shared memory space, additional optimizations are needed for memory reads and writes. A read from global ram requires 200-300 clock cycles and a read from L1 cache or shared memory requires 2-3 clock cycles. The developer can choose to give more of the 2-3-clock cycle memory to either L1 or shared. If the developer is not using shared memory algorithms, then they may want to favor the L1 cache and give less memory to the shared memory. Otherwise they can give more memory to the shared memory and program the software-defined cache. This was done because the developer may better know how to place items in the shared memory than the hardware control can place items in the L1 cache. If the developer can read items from the global memory and then place the items in shared memory and re-use the items in shared memory, this can save many 200-300 clock cycle fetches. In addition, if the developer reads global memory and properly takes into account the memory banks, the global fetch time will be closer to 200 clock cycles than to 300 clock cycles. All of the above settings are configurable from within Rootbeer.

Finally, the developer needs to make sure that there are many threads running in order to get a speedup because the memory clock rate is slower, the GPU clock rate is slower and the GPU has a simplified execution pipeline compared to the CPU.

Quickly comparing NVIDIA GPU devices with Intel MIC devices, it has been shown [48] while Intel MIC offers a promise of support for more branchy computations, the research has not developed to the point where it is feasible to obtain a speedup in applications such as HMM learning over GPUs.

2.7 CUDA Programming

To program with CUDA you need code for the GPU and code that controls the GPU from the CPU. The entry method to the GPU can be called the kernel function. We will talk about programming NVIDIA GPUs with CUDA. You can also program them with OpenCL, but for reasons discussed earlier we will not do that here.

The document will show a basic GPU kernel that sets each element in an array to its index. Don't expect a speedup from a computation as simple as this. Due to the architecture of the GPU and the fact that it connects via the PCIe bus means that 10 threads don't necessarily give a 10X speedup. In Chapter 7 a discussion of GPU performance optimization will be given.

Our basic GPU kernel written in CUDA shown is below. The function prototype starts out with a "__global__" intrinsic specifying that this method is an entry point to the GPU. The function basic_kernel returns void and accepts an array of integers called mem. At line 1, we obtain an index into the array using threadIdx.x. The index will be a different value for each thread in the computation. Then at the next line we set a single element of mem to equal its index. The programmer has to remember that each thread will have a distinct index so the outer loop of execution has been removed. For complicated applications, fully understanding concurrent behavior can require a developer who is very knowledgeable with concurrency. We refer the reader to "The Little Book of Semaphores" [49] for excellent coverage of parallel programming and semaphores.

```
000 __global__ void basic_kernel(int * mem) {
001   int index = threadIdx.x;
002   mem[index] = index;
003 }
```
Figure 2.7.1 – Basic GPU Kernel

We want to point out here that when programming with Rootbeer, the developer still needs to understand the concurrent model of CUDA. Conversion of serial to parallel has not been done, we simply provide a substrate to program GPUs from Java. Given that Java is the second most popular programming language in the world [12], we think that this is valuable. Our work also opens up the opportunity to build robust auto-parallelization systems on top of Rootbeer since it is freely available and open source.

Back to our CUDA example, we need to also create CPU code to control the GPU. This is shown in the figure below.

```
000 #include <stdio.h>
001
002 int main(int argc, char * argv[]) {
003   int num;
004   int size;
```

```
  int * d_arr;
  int * h_arr;

  num = 10;
  size = sizeof(int) * num;

  //initialize host array
  h_arr = (int *) calloc(num, sizeof(int));

  //allocate GPU array
  cudaMalloc((void **) &d_arr, size);

  //copy memory from CPU array to GPU array
  cudaMemcpy(d_arr, h_arr, size, cudaMemcpyHostToDevice);

  //launch the parallel tasks on the GPU
  basic_kernel<<<1, num>>>(d_arr);

  //block on kernel completion
  cudaThreadSynchronize();

  //copy memory from the GPU array to the CPU array
  cudaMemcpy(h_arr, d_arr, size, cudaMemcpyDeviceToHost);

  //free the GPU memory
  cudaFree(d_arr);

  //print the results using CPU memory
  for(int i = 0; i < num; ++i){
    printf("mem[%d]: %d\n", i, mem[i]);
  }

  return 0;
}
```

Figure 2.7.2 – Basic GPU Launch Code

The first thing we do is to use calloc to malloc an array with ten elements that are initialized to zero.

```
//initialize host array
h_arr = (int *) calloc(num, sizeof(int));
```

Figure 2.7.3 – Allocating Host Memory with calloc

Then we allocate the GPU array with cudaMalloc. Notice how we must pass in a pointer to a pointer (&d_array) and the data type is "void **". This is because the CUDA runtime must assign a pointer to d_array. If you haven't programmed in C for a while or if you are, say, a biologist trying to write a genomic sequencing application, this might take a little time to get right. And if your data types are arbitrary graphs of composite objects, you might have to work for a while to get everything right, and you will have to do this for each cut. This is where Rootbeer is useful, everything is done automatically for the developer.

```
//allocate GPU array
cudaMalloc((void **) &d_arr, size);
```

Figure 2.7.4 – Allocating Device Memory with cudaMalloc

After allocating one or more GPU arrays, we need to copy the CPU array to the GPU array. We need to pass in the device array, the host array, the size and cudaMemcpyHostToDevice to specify the direction. Again, if you are doing this for complicated objects, it can take time to write correct code to serialize the entire state into a set of closed arrays of primitive types. By closed we mean that no pointer points outside of our set of arrays. It can be complicated and time consuming to keep all allocations closed for collections such as a C++ map.

```
//copy memory from CPU array to GPU array
cudaMemcpy(d_arr, h_arr, size, cudaMemcpyHostToDevice);
```

Figure 2.7.5 – Copying CPU memory to GPU memory

Now to be fair, we should mention that OpenACC allows a developer to use directives from C and Fortran to avoid all of these issues. But OpenACC is still limited when it comes to complex types. A major problem is that, in C, a deep copy cannot always be determined via the compiler. But Rootbeer operates on Java Bytecode and a deep copy can be determined via the compiler.

After the cuda memory copy, we need to launch the kernel (named basic_kernel). We specify that there will be one block (more on this later) and num (or 10) threads. We are passing in the GPU array (d_arr) as the only parameter.

```
//launch the parallel tasks on the GPU
basic_kernel<<<1, num>>>(d_arr);
```

Figure 2.7.6 – Starting the basic_kernel on the GPU

Since the CUDA runtime allows a user to run multiple kernels on multiple GPUs, we also need to remember to wait for the current GPU to complete operation. When using multiple GPUs, you can use the Streams API to specify what GPU you are waiting for. Below we show how to wait for the GPU to complete.

```
//block on kernel completion
cudaThreadSynchronize();
```

Figure 2.7.7 – Synchronize Threads

After the GPU task is done, then copy the memory back from the GPU and use it on the CPU. Remember now to use cudaMemcpyDeviceToHost instead of cudaMemcpyHostToDevice to go in the opposite direction.

```
//copy memory from the GPU array to the CPU array
cudaMemcpy(h_arr, d_arr, size, cudaMemcpyDeviceToHost);
```

Figure 2.7.8 – Copying GPU memory to CPU memory

Finally we need to free the GPU memory. It is simple in our example, but with even a slightly more complicated data structure, such as a two-dimensional array, the developer needs to first free all the inner arrays in a loop and then free the outer array.

```
//free the GPU memory
cudaFree(d_arr);
```

Figure 2.7.9 – Releasing Device Memory with cudaFree

The previous text has shown the most basic example to program with CUDA. It may seem somewhat simple to use the special memory alloc functions, but when taking into account changing thread and block count, the entire CPU part of the code can change when configuring parameters. We find that the automatic serialization support from Rootbeer can greatly enhance productivity when tuning parameters.

Chapter 3

Rootbeer API
Illustrative Example

In this chapter we show how to program with Rootbeer. Then an example application will be given that will be used to demonstrate all aspects of Rootbeer compilation for later chapters.

3.1. Rootbeer Programming

Rootbeer supplies an interface called Kernel that developers implement to specify code that will possibly run on the GPU. After creating and compiling their application normally, developers also need to compile a packed Java Archive (jar) and then use the

Rootbeer Static Compiler to enable GPU usage. The packed jar must contain all classes reachable from the Kernel except Java Runtime Classes. The Kernel interface is shown below.

```java
package org.trifort.rootbeer.runtime;

public interface Kernel {
  void gpuMethod();
}
```

Figure 3.1.1 - Rootbeer Kernel Interface

The Kernel interface contains a single method called "gpuMethod". It takes zero arguments and returns void. A developer specifies what data will be copied to the GPU by placing it in a field reachable from the implemented gpuMethod. All pure Java data types are supported in these fields.

Now comes our application kernel code. We are going to write the same application from the CUDA programming example in Section 2.6, except we will be using Rootbeer and Java.

```java
import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.RootbeerGpu;

public class IndexArrayKernel implements Kernel {
  private int[] mem;

  public IndexArrayKernel(int[] mem){
    this.mem = mem;
  }

  public void gpuMethod(){
    int index = RootbeerGpu.getThreadIdxx();
    mem[index] = index;
  }
}
```

Figure 3.1.2 – Rootbeer Kernel Class Example #1

You can see in the figure above that we have a private field that is part of IndexArrayKernel. This will be the data to copy to the GPU. It is a single dimensional array of integers, but we could have used any other primitive or reference type inside the array.

We could use any number of dimensions or we could have chosen to not even use arrays at all. Finally, we could even use static data.

In “gpuMethod”, the GPU kernel gets threadIdx.x and assigns it to index. This is accomplished using a CUDA runtime packaged with Rootbeer. The CUDA runtime binds threadIdx.x to RootbeerGpu.getThreadIdxx. At line 13, we assign index to “mem[index]”. Under the covers, the first thing that happens is a reference is made to the mem field from the IndexArrayKernel class given the current “this pointer”. Then an integer array set operation is completed on the array object passing in the index and the value.

Now that we have our Kernel class written, we need to write the code to control the GPU from the CPU. This is shown in Figure 3.1.3 below.

```
import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.Rootbeer;

public class IndexArrayKernelLaunch {
  public static void main(String[] args){
    int size = 10;
    int[] mem = new int[size];
    List<Kernel> tasks = new ArrayList<Kernel>();
    for(int i = 0; i < size; ++i){
      tasks.add(new IndexArrayKernel(mem));
    }
    Rootbeer rootbeer = new Rootbeer();
    Rootbeer.run(tasks);

    for(int i = 0; i < mem.length; ++i){
      System.out.println(“mem[“+i+”]: “+mem[i]);
    }
  }
}
```

Figure 3.1.3 – Rootbeer Kernel Launch Example #1

Notice that we simply allocate a new integer array, create ten IndexArrayKernels stored in a list and call Rootbeer.run. There is no GPU memory allocation, no serialization code and no deserialization or free code. Rootbeer does all of these things automatically. In our simple example there are only a few lines of code difference, but in a complex application with multiple kernel launch points, this can save considerable programming and debugging time. This usage of Rootbeer is called the Kernel Object API. It has worse performance than the Kernel Template API discussed in the next section. This is because, in the Kernel Object API, an object and header are created for each Kernel Object and this takes considerable time to serialize and deserialize. However the Kernel Object API can be useful if you need very different data for each kernel or you are a beginner. For best performance you will usually want to use the Kernel Template API. After discussing the Kernel Template API in the next section, we will talk about using multiple GPUs from Rootbeer and how to use shared memory and atomics from within Rootbeer.

3.2. Kernel Templates API

The previous example showed the Kernel Objects API. That API is nice to get started when experimenting with Rootbeer, but it performs worse than the Kernel Templates API. In the Kernel Templates API, there is only one object and handle that needs to be transferred, improving copy time. In addition to that, the Kernel Templates API has been optimized for throughput processing.

```
public class IndexArrayKernelTemplateLaunch {
  public static void main(String[] args){

    Rootbeer rootbeer = new Rootbeer();
    List<GpuDevice> devices = rootbeer.getDevices();
    GpuDevice device0 = devices.get(0);
    Context context0 = device0.createContext(96000432);
    context0.setCacheConfig(CacheConfig.PREFER_SHARED);
    context0.setThreadConfig(size, blockSize, blockSize * size);
    context0.setKernel(new GPUHistKernel(input, resultGPU));
    context0.buildState();

    while(true){
      context0.run();
    }
  }
017  }
```

Figure 3.2.1 – Rootbeer Kernel Templates API in Throughput Mode

You can see in the above example that at line 6 we find the device we want to work with and in line 7 we create a context with the memory size initialized. The next three lines set the cache configuration, the thread configuration and the kernel. Then at line 11 we call build state, which initializes the driver to go into high throughput mode.

We have a while-true loop from lines 13 to 15 that runs the kernel over and over in high throughput mode. The context0.run() line serializes state reachable from the GPUHistKernel to the GPU, executes the tasks with the pre-built thread config on the GPU and then deserializes the state back. If needed, state can change before that line to give the GPU different data, and the results of the computation can be used after that line.

3.3. Multi-GPU Support

To use multiple GPUs within a single computer, you just need to query the Rootbeer API and run two contexts at once. The code listing below shows this.

```
public void multArray(int[] array0, int[] array1){

  Rootbeer rootbeer = new Rootbeer();
  List<GpuDevice> devices = rootbeer.getDevices();
  if(devices.size() >= 2){
    Kernel gpu_kernel0 = new IndexArrayKernel(array0);
    Kernel gpu_kernel1 = new IndexArrayKernel(array1);

    GpuDevice device0 = devices.get(0);
    GpuDevice device1 = devices.get(1);
    Context context0 = device0.createContext(4096);
    Context context1 = device1.createContext(4096);

    rootbeer.run(gpu_kernel0, context0, 1, 10);
    rootbeer.run(gpu_kernel1, context1, 1, 10);
  }
}
```

Figure 3.3.1 – Rootbeer Multi-GPU Example

A context is used as a container to hold memory buffers internally in Rootbeer. Some GPU applications can be optimized to leave some objects in GPU memory between task launches if they have not changed. Rootbeer does not support this optimization currently, but our architecture has been created while keeping that in mind.

The context must be created with a size. This is the size of the GPU heap that is allocated before serialization occurs. Originally in Rootbeer version 1.0, the runtime would allocate as much memory as possible. This causes issues on some systems due to differences in required reserve memory.

The new context API allows the developer to specify the size of the heap, or if not specified, will use the entire GPU memory. Once the program has run with the entire memory, you can query the Rootbeer API and determine exactly how much was used and then feed that back into the code for a manual allocation.

Now that it has been shown how to use multi-gpu support within a single machine, we will now show how to program against Rootbeer's API for Shared Memory.

3.4 Shared Memory Support

Shared memory in CUDA is like a software-defined cache. If the programmer can manage the cache better than hardware implementation, a speedup will occur. Rootbeer supports simple usage of the shared memory. A developer can place simple primitives or references to objects in the shared memory. A developer cannot currently place an entire object into the shared memory automatically. Future work could use static and dynamic analyses to automatically generate software-defined cache managing code or to allow objects to be placed into shared memory. Support is still primitive; the developer needs to know the size of each object placed into shared memory.

The example below multiplies each element of an array by the first two elements. To save memory transfer time, the first two elements of the array are stored in shared memory by the first two threads. Then a block level synchronization is done and all of the threads in the block can see the items properly updated in shared memory. After that, each thread can fetch the two elements from shared memory only using a few clock cycles. The computation shown in Figure 3.4.1 probably won't run faster, but the ideas on how to use shared memory from Rootbeer are correct.

```
import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.RootbeerGpu;

public class SharedMemoryKernel implement Kernel {
  private int mem;

  public void gpuMethod(){
    int index = RootbeerGpu.getThreadIdxx();
    if(index == 0 || index == 1){
      //fetch from global memory
      int element = mem[index];
      //store in shared memory
      RootbeerGpu.setSharedInteger(index * 4, element);
    }
    RootbeerGpu.syncthreads();
    int element0 = RootbeerGpu.getSharedInteger(0);
    int element1 = RootbeerGpu.getSharedInteger(4);
    int[] mem_ref = mem;
    int thread_item = mem_ref[index];
    thread_item = (thread_item * element0) +
                  (thread_item * element1);
    mem_ref[index] = thread_item;
  }
}
```
Figure 3.4.1 – Rootbeer Shared Memory Example

Let's take a moment to analyze what is happening in the memory model for the above example. First, it is important to realize that when you are using Rootbeer, almost everything starts out in global memory. In our example above, "mem" is a field of

SharedMemoryKernel. The serialization code will first write an object header to a CUDA memory buffer, and then all of the reference handles and primitive types. In the Rootbeer GPU memory model, “mem” is an integer handle that points to another object (an int[] object that was also serialized).

Accessing the mem field twice without saving it will require two reads of the handle from global memory. At line 18 you can see a small optimization that we have done. We are reading the mem field and storing it in an “int[]” called mem_ref. Now, the handle that was in the “mem” field in global memory is now in a register in CUDA. Then when we reuse mem_ref at line 22, we can save time. The rules are: 1) all fields are in global ram, 2) all local variables are represented directly or as handles in registers and 3) things in shared memory are explicitly in shared memory.

When programming Rootbeer to obtain the best results, you need to do your own optimizations of reading fields and array lengths. To understand why this is the case, you have to understand how javac compiles code and how that maps to our CUDA C. Jarvac completes zero optimizations on the Java Bytecode before writing it to the class file. This is because the JIT compiler needs to take into account externally linked code and it also uses runtime information to better optimize the code. It was determined by the Hotspot developers that it was best to simply not do any optimizations in Javac and do all optimizations in the JVM. Therefore our Java Bytecode that is compiled directly to CUDA C is unoptimized. This is why all of the redundant field references and array length reads are redundant reads to global memory.

3.5 Illustrative Example

An illustrative example that will be used in the remainder of this document will now be covered. The example covers every aspect of Java that will be displayed in the code generation section.

There are three classes in this example: 1) ArraySumKernel, 2) FloatArraySum and 3) IntArraySum. The example has two different algorithms to sum the contents of arrays in parallel on the GPU and uses both integers and floats for this. At the end it also prints out a string constant that was created on the GPU saying which algorithm it used.

ArraySumKernel is below. It shows a kernel with static reference type fields. It also contains a synchronized method that uses instanceof.

```
package arraysum;

import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.RootbeerGpu;

public class ArraySumKernel implements Kernel {

  private static Object array;
  private static FloatArraySum engine;

  public ArraySumKernel(Object array){
    this.array = array;
  }

  public void gpuMethod() {
    int thread_id = RootbeerGpu.getThreadIdxx();
    if(thread_id == 0){
      initEngine();
    }
    RootbeerGpu.threadfence();
    RootbeerGpu.syncthreads();
    engine.sum();
    RootbeerGpu.syncthreads();
    if(thread_id == 0){
      engine.printResult();
    }
  }

  private void initEngine(){
    if(array instanceof FloatArray){
      FloatArray floatArray = (FloatArray) array;
      engine = new FloatArraySum(floatArray.get());
    } else {
      IntArray intArray = (IntArray) array;
      engine = new IntArraySum(intArray.get());
    }
  }
}
```

Figure 3.5.1 - ArraySumKernel

FloatArraySum is a base class that sums the elements of the array using some parallel processing. It contains instance fields, primitive fields, array references, and shared memory usage. It also contains creating a new array of primitive type and assigning a string constant to a field.

```
package arraysum;

import org.trifort.rootbeer.runtime.RootbeerGpu;

public class FloatArraySum {

  protected String descriptor;
  protected float[] array;
  protected float[] newBuffer;
  protected float sum;

  public FloatArraySum(){
  }

  public FloatArraySum(Object array) {
    this.array = (float[]) array;
  }

  public void sum() {
    int thread_id = RootbeerGpu.getThreadIdxx();

    float value1 = array[thread_id * 2];
    float value2 = array[(thread_id * 2) + 1];
    RootbeerGpu.setSharedFloat(thread_id * 2 * 4, value1);
    RootbeerGpu.setSharedFloat((thread_id * 2 + 1) * 4, value2);
    RootbeerGpu.syncthreads();

    int numThreads = RootbeerGpu.getBlockDimx();
    for(int i = 0; i < 4; ++i){
      if(thread_id < numThreads){
        value1 = RootbeerGpu.getSharedFloat(thread_id * 2 * 4);
        value2 = RootbeerGpu.getSharedFloat(
          (thread_id * 2 + 1) * 4);
        float itemSum = value1 + value2;
        RootbeerGpu.setSharedFloat(thread_id * 4, itemSum);
      }
      RootbeerGpu.syncthreads();
```

```
        numThreads /= 2;
      }
      numThreads *= 2;

      if(thread_id == 0){
        newBuffer = new float[numThreads];
      }
      RootbeerGpu.threadfence();
      RootbeerGpu.syncthreads();

      if(thread_id < numThreads){
        newBuffer[thread_id] = RootbeerGpu.getSharedFloat
          (thread_id * 4);
      }
      RootbeerGpu.syncthreads();

      if(thread_id == 0){
        float sum = 0;
        for(int i = 0; i < numThreads; ++i){
          sum += newBuffer[i];
        }
        this.sum = sum;
        this.descriptor = "float-array-sum: ";
      }
    }

  private synchronized float doubleResult(float sum){
    return sum * 2;
  }

  public void printResult() {
    System.out.println(descriptor+doubleResult(sum));
  }
}
```

Figure 3.5.2 - FloatArraySum

IntArraySum is very similar to FloatArraySum, except that it operates on integers and not floats. It contains almost all of the elements from FloatArraySum, and it derives from FloatArraySum, so the field indexing will be interesting.

```
package arraysum;

import org.trifort.rootbeer.runtime.RootbeerGpu;

public class IntArraySum extends FloatArraySum {

  private int[] array;
  private int sum;

  public IntArraySum(Object array) {
    this.array = (int[]) array;
  }

  public void sum() {
    int thread_id = RootbeerGpu.getThreadIdxx();

    int value1 = array[thread_id * 2];
    int value2 = array[(thread_id * 2) + 1];
    RootbeerGpu.setSharedInteger(thread_id * 2 * 4, value1);
    RootbeerGpu.setSharedInteger((thread_id * 2 + 1) * 4, value2);
    RootbeerGpu.syncthreads();

    int numThreads = RootbeerGpu.getBlockDimx();
    for(int i = 0; i < 4; ++i){
      if(thread_id < numThreads){
        value1 = RootbeerGpu.getSharedInteger(thread_id * 2 * 4);
        value2 = RootbeerGpu.getSharedInteger(
          (thread_id * 2 + 1) * 4);
        int itemSum = value1 + value2;
        RootbeerGpu.setSharedInteger(thread_id * 4, itemSum);
      }
      RootbeerGpu.syncthreads();
      numThreads /= 2;
    }
    numThreads *= 2;

    if(thread_id == 0){
      int sum = 0;
      for(int i = 0; i < numThreads; ++i){
        sum += RootbeerGpu.getSharedInteger(i * 4);
      }
      this.sum = sum;
      this.descriptor = "int-array-sum: ";
    }
  }

  public void printResult(){
    String className = this.getClass().getName();
    System.out.println(descriptor+className+": "+sum);
  }
}
```

Figure 3.5.3 - IntArraySum

This concludes the Illustrative example and Chapter 3. We have seen in this chapter how to program using Rootbeer in a variety of ways. Next in Chapter 4 we will discuss the Rootbeer Class Loader.

Chapter 4

Rootbeer Class Loader

The Rootbeer Class loader is an improvement to the original class loader provided by Soot. It was designed to allow Rootbeer to complete whole-program analysis while using less memory and time. Using the original class-loading algorithm can easily require over 8GB of ram on simple programs. In our evaluation we show that the new class loader loaded a Scene that had 70% to 90% less classes and didn't crash on larger test cases as did the original class loader.

4.1 Soot Class Loading Algorithm

The original Soot class-loading algorithm loads all classes and any classes referenced from loaded classes. All methods in a class are loaded whether or not they exist in possible program traces. In contrast, our work only loads methods if they are reachable from the specified entry points on a depth-first search walk of the call graph.

The Soot class-loading algorithm uses a worklist and three resolving levels to load classes. The table below shows the resolving levels and the loaded information at each level.

Resolving Level	**Associated Information**
Dangling	Nothing is known about the class, except its fully qualified name.
Hierarchy	SootClass objects for the class, its superclass and interfaces are created. The class, its superclass and interfaces are resolved to Hierarchy.
Signatures	SootMethods for all methods in the current class are created. Types for the return value, parameters and exceptions are resolved to Hierarchy.
Bodies	JimpleBodies are created for all methods in the current class. Creating the Jimple format requires that all SootMethods that are invoked from the current body have been created properly beforehand. The original class-loader handles this problem by raising every type in the constant pool of the current class to signatures.

Figure 4.1 – Resolving Levels in Soot Class Loader

When a class is first referenced, it is added to the worklist at Hierarchy Level. At this level, the super class and interfaces are also added to the worklist at Hierarchy level.

Once another class needs to know the methods or fields that exist in a target class, the target class is loaded to Signatures Level. When a class is loaded to signatures, everything from Hierarchy Level is known, plus the method and field signatures. The original Soot class-loader raises each type found in the method and field signatures for the entire class to Hierarchy Level. At this point, while rising to Hierarchy Level, the type's super class and interfaces are also raised to Hierarchy Level.

To obtain the Java Bytecode for a method, a JimpleBody needs to be created from the method instructions. The JimpleBody creator needs all referenced methods to be raised to signatures or else it cannot handle interface invoke expressions properly. The original class-loader handles this problem by raising every type in the constant pool of the current class to signatures. Then as these are loaded to signatures, all the return, parameter and exception types are loaded to hierarchy. Doing this causes a type explosion - more types added cause yet even more types to be added. Due to the type explosion caused, loading a fairly large Scene can take 8GB to 15GB and some Scenes cannot be loaded in the available memory on many machines. All of this processing also takes a considerable amount of time.

4.2 Rootbeer Class Loading Algorithm

The Rootbeer Class Loader is an optimized implementation of Rapid Type Analysis (RTA) class loading [50] [51] for Soot. An overview of Rapid Type Analysis will be given and then we will focus on the optimizations that the Rootbeer Class Loader does specifically for Soot and Java Bytecode.

RTA Class-Loading starts with a depth first search walk from prescribed entry points. It marks methods found as reachable. When it encounters a method that is possibly virtual, it must find the call sites for this virtual method. It uses the class hierarchy combined with the knowledge of classes that have been invoked with new to find valid call sites. As the depth first search walk extends into the analyzed program, more possible classes have been created with new and now the call sites for previously analyzed methods must be updated. This is accomplished by running the whole class-loading algorithm in a fixed pointer over the number of classes created with new. An algorithm of this is listed in Figure 4.2.1 below

```
CallGraph cg = loadCallGraphUsingClassHierarchyAnalysis();
int prevSize = -1;
while(prevSize != newInvokes.size()){
    prevSize = newInvokes.size();
    for(EntryPoint entryPoint : entryPoints){
        Set<NewInvoke> dfsNewInvokes = forwardDFS(entryPoint, cg);
        newInvokes.addAll(dfsNewInvokes);
    }
}
```

Figure 4.2.1 – RTA Class Loading Algorithm

Many optimizations have been done in the Rootbeer Class Loader to make class-loading fast. We will discuss the various phases of the Rootbeer Class Loading algorithm in the following sections.

4.2.1 Load Hierarchy

The Rootbeer Class Loader starts out by loading the hierarchy from the input Java Archive (jar) files. A Java Archive is a zip file with compiled Java Bytecode stored in *.class files. The Rootbeer Class Loader starts out by reading the contents of every file ending in *.class from the submitted analysis classpath. There is special code at this point that can read a byte array representing the *.class file and return only the superclass string for the current class. The method for doing this came from reading Soot code and the internals of the JVM. After that is done, we have the class name string, the super class name string and the contents of the bytecode as a byte array. We have a treemap that can return the byte array contents given the class name for later use and we store the child/parent superclass information in two maps. One map is a superclass hierarchy map and another map is a subclasshierarchy map. In both of these maps, we map from RTAType to RTAType instead of using strings. RTAType is a flyweight [52] that can hold a typename. In the flyweight, the package name and class name is split into two parts. Then, in most of the Rootbeer Class Loader code, all class file strings are converted to a unique integer. This technique is called manual string pooling. The JVM contains automatic string pooling [53] code, but ahead of time constants for the string pool size must be tuned for best performance depending on what version you are using. Using manual string pooling in this case will give reasonable performance under a large range of input programs and JVMs. We split the class/package name into two strings because many classes contain repeated package names and this reduces the size of the string pool, thus enhancing performance. RTATypes are often kept in treemaps and treesets in the code that are comparable based on the string numbers and array dimensions of the type.

A prior version read every class file using an O(n) average time lookup into the Java Archive and this proved too slow for real-world compilation cases. Now, the entire byte arrays for all class files are saved into a map. The memory usage required for this is reasonable, around 35MB.

4.2.2 Find Entry Points

The next step of the processing is to find the entry points as specified by the user. The processing goes through all application classes (configurable) and searches all RTAMethods. At this point in the processing, the RTAMethods are read with special code that reads only the required entries from the constant pool, skipping all others for speed. Later, the full constant pool can be read if a method is loaded entirely to bodies.

While searching all application RTAMethods, the configured method signatures, method subsignatures and EntryMethodTesters are checked. If any return entry point matches, the RTAMethod is marked as an entry point. To configure this, the developer only uses simple strings passed in and doesn't need to worry about the complexity of the original Soot class loader associated with specifying entry points. The EntryMethodTester interface used for specifying entry points is shown below.

```
public interface EntryMethodTester {
  public boolean matches(RTAMethod method);
  public Set<String> getNewInvokes();
}
```

Figure 4.2.2.1 – EntryMethodTester Interface

The EntryMethodTester requires the developer to specify which classes were created with new at the entry point for the context sensitive class loader. To help fill in some details automatically, there is a MainEntryMethodTester class given that implements EntryMethodTester for regular Java main methods. In general, the RTAMethod object can determine a match based on the method signature, the staticness and publicness of a method and also from a custom string based instruction format. This string based instruction format is an entirely new low level instruction format that keeps all instructions as strings and retains type information that Rootbeer uses to detect the Rootbeer TestHarness. In general, it can be used by malware analyzers and others to work with very complex class loading.

4.2.3 Load Signatures Classes

After finding the entry points, specific classes can be marked to be loaded simply to signatures without requiring them on the DFS walk. Rootbeer needs this to generate Jimple code that uses the Rootbeer Runtime. If can be used by others anytime that code generation is done against an included runtime.

4.2.4 Call Graph Fixed Point

The next phase of the algorithm is to load the call graph using a fixed point over the number of new_invokes found. This can be done in a context sensitive or context insensitive manner. In the context insensitive manner, the new_invokes are seeded with the classes specified from the entry point and additional new_invokes are directly added to all points in the call graph. In the context sensitive manner, the context new_invokes are seeded with the classes specified from the entry point. For each method traversed on a DFS walk, the current new_invoke context is used to find possible call site targets for a virtual method. Classes created with new are added to the current context and propagated to the next methods. RTATypes are used to represent the new_invokes and it helps that they are implemented as a flyweight pattern to reduce memory usage of the contexts. If a class or method cannot be found in the input Java Archives, it is created on the fly and marked as phantom as in the existing Soot code. If a method is marked “don’t follow” the Rootbeer Class Loader will not walk into the method.

There is a special option in the Rootbeer Class Loaders to forward specific method invocations to other methods. It is used for situations like Thread.start(). When Thread.start() is called, the JVM internally calls Runnable.run() on the class passed into the Thread constructor. Using this special option in the Rootbeer Class Loader allows us to properly analyze code in a thread by passing the current context to Runnable.run(). This works nicely with Threads at this time but may need small improvements in the future to allow for more flexibility.

4.2.5 Load Scene

To load the scene, first the types are numbered according to a reverse topological numbering [54]. This helps when interfaces are involved as regular breadth first search numbering can work with non-interface classes. The reverse topological numbering has Object numbered at zero, interfaces directly below Object numbered starting at one, and interfaces extending other interfaces next and non-interfaces numbered last. A diagram of this is shown in Figure 4.2.5.1 below.

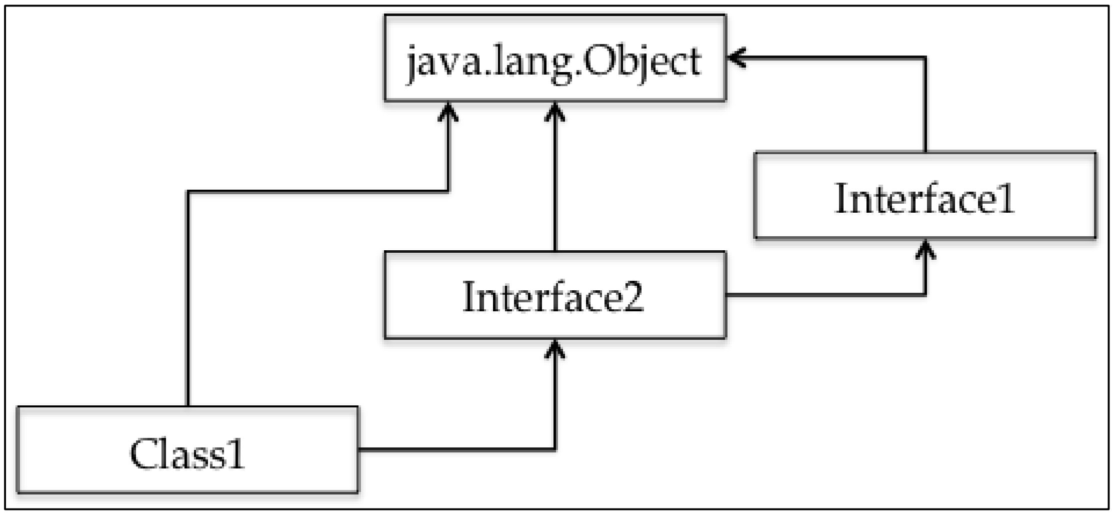

Figure 4.2.5.1 – Class Hierarchy Example for Reverse Topological Numbering

As can be seen in Figure 4.2.5.1, both Interface1 and Interface2 extend from Object, but Interface2 also implements Interface1. In this diagram, Object will be numbered 0, Interface1 will be numbered 1, Interface2 will be numbered 2 and Class1 will be numbered 3. This is according to reverse topological order. If the algorithm had incorrectly used breadth first search ordering, Interface2 could be numbered before Interface1 and this will cause problems because Interface1 must be loaded before Interface2 (since Interface2 derives from Interface1).

Once the classes are numbered, first the SootClass object is created for Object and added to the Scene. Since Object is the superclass of all other types, it will be correctly loaded into the Scene before a class below needs to refer to it as a superclass. Then the SootClass objects are created according to ascending number. The topological order guarantees that the superclass and all implementing interfaces are correctly loaded into the Scene before the current class, otherwise Soot would fail to load the Scene properly.

Once all of the SootClasses are created, all SootFields are created that were found on the depth first search walk of the call graph. Note that some declared fields from the class files would be missing in this case. In analysis this isn't that important, but for the case of code generation, that will become a problem. This problem is solved in the next section, 4.2.6.

Once all of the SootFields are created, empty SootMethods are created for every method reachable on the depth first search walk. Again, some declared methods will not be loaded and the solution for this in the case of code generation is listed in the next section.

Now that all of the empty SootMethods have been created, the existing code Soot can used to load the JimpleBodies. We guarantee that this point all required SootClasses, SootMethods and SootFields have been created properly and the existing complicated body loading code will work. Since we are not loading every type inside of the constant pool and every SootMethod inside every SootClass, by this time, many fewer SootClasses have been made, saving time and space.

4.2.6 Supporting Code Generation

Part of the Rootbeer Serialization strategy is to generate Java Bytecode that can serialize state in a high performance manner. In order for this to work, the compiler needs to attach generated code to existing code loaded with the Rootbeer Class Loader. There is a problem because the Rootbeer Class Loader does not load every single method and some absent methods will be needed for other aspects of the application. We could load the additional methods at this time using either class loading strategy, but this will cause more types to be loaded than are needed. The Rootbeer Class Loader notices that the entries in the constant pool are just strings and don't need to be converted into the types SootClass, SootMethod and SootField objects if the Jimple IR is not needed (which it isn't at this point). We have created custom code that can read the byte array representing the contents of a class file and convert it into a Jasmin text-based IR. Then we use the existing Jasmin text-based parser to parse the full class file text and merge it with the modified methods and fields from the previous class loader discussed. The existing Jasmin assembler is then used to convert the merged text back into a byte array representing a class file. (Jasmin is a project originally used by Soot to lower JimpleBodies to class files used by the JVM). Now we have the completed class files for placement into the Java Archive output.

This text-based extension is configurable with an API that is included with the Rootbeer Class Loader. The main functionality of this API is to present to the Rootbeer Class Loader what SootClasses have been modified so that it can return those SootClasses properly merged with the absent methods and fields from the original class file.

4.3 Evaluation

To evaluate the Rootbeer Class Loader, we have executed test cases on a MacBook Pro with 8 cores and 8GB of RAM, using Java Hotspot Server 1.7.0_51-b13.

Our test cases include a basic example (Appendix D), and the SPEC jvm 2008 benchmark [55]. We measured the execution time using available timers in Java, and used Java APIs to measure the total memory used in the JVM. We compare our implementation against the Soot class loader by loading programs and taking measurements in the Whole Jimple Transformation Pack (wjtp) phase in Table 4.3.1.

Test Cast	**Classes Loaded**	**Memory Used (MB)**	**Time (s)**
Basic Example (Soot)	2798	513	26
Basic Example (RTA)	673	135	11
SPEC jvm 2008 #1 (Soot)	7223	1325	105
SPEC jvm 2008 #1 (RTA)	671	149	11

Figure 4.3.1 – Evaluation Results for RTAClassLoader

We notice a significant decrease in the number of classes loaded and execution time over the original Soot class-loader. In addition, in some larger examples, the Rootbeer Class Loader will succeed while the original Soot class loader will crash with an error.

4.4 C# Class Loading

It is worthwhile to note that if Rootbeer was ported to C# [56], another very popular language by Microsoft, there may not be as many class loading issues. According to Robert Yates [57], C# does not keep methods virtual by default and this may lead to many less methods that need to be resolved in the Rootbeer Class Loader. This is in contrast where every method in Java is virtual by default.

4.5 Rootbeer Class Loader Conclusion

Data structures and algorithms have been shown that efficiently load a Soot Scene using the Rootber Class Loader. The class loader loaded a Scene in 10x less time and used 8x less memory. In large real-world problems this can turn infeasible analyses into feasible analyses in areas such as malware analysis, memory usage analysis and web-site security analysis. The Soot open source community is very active in Android malware analysis and the new class-loader created for Rootbeer could be incorporated into Soot once Android dex files are supported.

This concludes the chapter on the Rootbeer Class Loader. The Rootbeer Serialization discussion follows.

Chapter 5

Rootbeer Serialization

To find a sufficient way to serialize arbitrary state in Java, we studied several possible methods and then implemented our code using the most promising technique. We found that generating Java Bytecode was the best possible serialization technique given the other available options.

5.1 Serialization Study

The performance of the three fundamental ways to read a field in Java were studied. The three ways of reading a field that are built into Java are: 1) JNI, 2) Reflection and 3) Pure Java. We wrote code to read a byte from the same field 10,000,000 times using each approach. We did not cache any method or field descriptors to be fair to real-world cases. Reading the field from Pure Java was the fastest by a factor of 34x to 49x, which is seen in Figure 5.1.1 below.

Method	**Execution Time (ms)**
JNI	247
Reflection	173
Pure Java	5

Figure 5.1.1 – Performance of Accessing a Field in Java using Various Methods

Once it was known that Pure Java is the fastest way to read fields in Java, the question arises as to how to do that. In a compiler you cannot do this from within the program being compiled, because this will require Reflection. We accomplished this by generating Java Bytecode that can serialize and deserialize state for a certain root gpuMethod using Soot.

High performance serialization using bytecode generation can be used for purposes other than Rootbeer, such as removing live objects from the garbage collected heap onto a native, manually managed heap. This is usefull because some garbage collection algorithms start to introduce large pauses in Java execution when the heap contains many GB worth of live objects. With less memory on the live heap and mechanisms to convert quickly back from the native heap, we see less jitter in the latency profile of an application.

5.2 Serialization Capabilities

Rootbeer can serialize the types found in the table below. Many test cases have proven that the serialization works. In future work, we aim to re-write the serialization module to reduce JNI calls to improve performance. During work like this it helps to have a special memory buffer that can check that reads and writes to/from memory are aligned on the word size.

Primitive Types	Boolean, Byte, Char, Short, Integer, Float, Double
Reference Types	Strings, Objected and Boxed Primitive Types
Arrays	Single and Multi Dimensional Arrays of Primitive and Reference Types
Fields	Types referenced from instance and static fields
Created Objects	Reference types in the application or reference types with a void constructor from the Java Runtime Environment

Figure 5.2.1 – Rootbeer Serialization Capabilities

5.3 High-Level Memory Layout

From a high-level, the memory is organized with the static memory positioned at the top of the memory space and the instance objects and pointers are positioned below the static pointers. This is showing in Figure 5.3.1 below. The start of instance memory is aligned on a 16-byte boundary to ensure alignment of all types.

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 STATIC MEMORY  00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
-----------------------------------------------
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00INSTANCE MEMORY 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 5.3.1 – High-Level Memory Layout

The static memory is written first and, as it is being written, instance objects with static field handles are placeed in instance memory. The serialization code can switch between static and instance object writing.

5.4 Static Memory Layout

The static memory has static fields written at the top. The CUDA code knows the precomputed offset of static fields and accesses them that way. If a static field is an instance type, the handle is written into static memory and the instance is written in instance memory. After the fields and handles are a number of 32-bit lock locations initialized to negative, one for each of the classes reachable from the gpuMethod. After the locks come strings that represent the class names for each class that will exist on the GPU. This is needed to output the full name of an object during toString.

5.5 Instance Memory Layout

The instance memory is arranged as objects after the static memory. Handles into the instance memory come from a handle array passed into the CUDA kernel.

5.5.1 Reference Type Memory Layout

The objects first have an object header, followed by reference type handles, followed by primitive typs. The object header contains a reference type count, a 'constructor used' flag, a class identifier, an object size and a monitor field. A garbage collector can follow the reference type handles due to the location and count field, however no garbage collector has been written. Future work may include published research on parallel GPU garbage collection [38]. After the reference types, the primitive types are positioned according to decreasing size to ensure alignment on element-sized boundaries.

5.5.2 Single Dimensional Primitive Type Array Memory Layout

Single dimensional arrays of primitive types are stored using a header and a raw block. The header contains a reference type count, a 'constructor used' flag, a class identifier, object size (in bytes), array length (in elements) and a monitor field.

5.5.3 Multi-Dimensional Array Memory Layout

The higher dimensions of multi-dimensional arrays are real Java arrays, because they need to be in order to support array length and clone. They have an object header, just like the single dimensional arrays and the data block contains 32-bit compressed handles into other multi or single dimensional arrays.

5.5.4 String Type Array Memory Layout

Strings are treated specially in Rootbeer. In Java 1.6 strings are cached and, without special treatment, changes to a string on the GPU can cause incorrect when serializing back to the CPU. On some Java platforms a string has two fields (value and count) while on others a strings have other representations. To give uniformity to the CUDA runtime written for Rootbeer, we make strings contain an object header and a pointer to a character array in every case. The character array is a real Java array with a length field.

5.6 Serialization Algorithm

To actually execute the serialization, there are several important components. First, an IdentityHashMap is used to map from Object to long representing the memory location. This is used because, in an object graph, the same object can be referenced multiple times and without an IdentityHashMap it would cause the same object to be written twice, destroying object identity. The IdentityHashMap is checked first to see if the object has been written to the GPU heap, and if so the long reference is returned from the serialization method, otherwise the object is written and the long reference is saved in the IdentityHashMap.

In order to serialize a specific kernel, a Serializer object is obtained from the CompiledKernel. Serializer is an abstract base class that contains help on serializing and an abstract method to serialize a specific kernel. The concrete implementation for that abstract method is attached during Rootbeer complication. CompiledKernel is an interface that is then attached to the Kernel class that allows the Rootbeer runtime to obtain the concrete Serializer object, without writing the entire Rootbeer runtime using low-level Jimple instructions.

Once the concrete serializer is found, *doWriteToHeap* is called with the specifics for serializing a certain Kernel object. The diagram below displays the order of serialization

1.	Strings
2.	Multi-Dimensional RefType and PrimType Arrays
3.	Single-Dimensional RefType Arrays
4.	Single-Dimensional PrimType Arrays
5.	RefTypes

Figure 5.6.1 – Serialization Type Order

While serializing the RefTypes, the code is generated in order of most-derived class to most-base class. This is because we use instanceof to determine which serialization fragment should be used. If a base class instanceof catches a more-derived class, the system will have problems and won't serialize properly.

5.7 Deserialization Algorithm

In a way similar to the serialization algorithm, we have a Map that acts as a cache of objects that are being read from the GPU heap. However, in deserialization we use a TreeMap going from a long to an Object. TreeMaps are a good choice for keeping long integers in an ordered set and we do not need an IdentityHashMap because a long reference is always equal.

Just as in the serializer, we are using the Serializer object obtained from the CompiledKernel. If the item is not in the read cache, then we read a flag CTOR_USED from the GPU object to see if the object has been made inside the GPU. If it has, then a corresponding new invocation must happen inside the JVM on the CPU host (since we are not modifying the JVM). For all application classes, we have created sentinel constructors that ensure that we can create the object in this way with new. A sentinel constructor is a constructor that requires a single parameter: an object of type org.trifort.rootbeer.runtime.Sentinal. Since the sentinel package is from the rootbeer runtime, we can guarantee by convention that we are not overriding a constructor that another party has produced. If the GPU object was not an application class, we see if there is a void constructor that can be used instead. If there is no sentinel constructor and no void constructor, then we cannot deserialize the GPU object. Future work can incorporate modifying the JDK libraries to automatically include sentinel constructors, but this can enter into legal issues related to modifying the bootclasspath.

Once we have a handle to a created object, either passed in or created recently with new, we call the deserialization fragments that have been attached to the CompiledKernel object by the compiler.

5.8 Issues

Two primary issues with serialization occurred: 1) issues with private package classes and 2) issues with private and hidden fields. These issues will also occur in future general purpose high-performance serialization libraries.

5.8.1 Package Private Classes

Package private classes can be a problem because the serialization engine needs to access all classes including package private classes, but the serialization engine is most likely in the compiler package namespace. To solve this issue we are using the composite pattern [52] for serialization. We are generating code to form the pattern, so first we need to find the public roots of each package. Then we append serialization methods to classes, which eventually call the serialization method of the package private classes. The algorithm for package roots detection is in Section 5.8.1.1 below.

5.8.1.1 Package Roots Detection Algorithm

For the package roots detection algorithm, first it iterates through every reference type and checks to see if it is public. If so, a graph node for the public package is added to a list of roots. Then the remaining private packages are traversed and a public root is found from the previous list based on the namespace name. In this way we can keep package private classes and chain serialization starting from a package public class.

5.8.2 Private and Hidden Fields

A naïve approach to handling private fields in serialization is to make all fields public. However, when incorporating hidden fields, making all fields public will change the logic of the code. Our solution is to append serialization and deserialization methods to read private fields from within public methods of the object. This works well with the package-private classes solution.

Chapter 6

CUDA Code Generation

Rootbeer generates CUDA code and this is compiled to PTX by nvcc. This chapter shows and explains the technique to generate CUDA code that can support all of the Java Programming Language. This can also be used in other work to cross-compile Java Bytecode to C and run natively on a CPU.

6.1 CUDA Entry

To start a Rootbeer compiled program, the CUDA entry is called first. This code is included in the CUDA runtime for Rootbeer. The name and parameter types of the entry method are the same for every Rootbeer application. This is because the nvcc compiler uses name mangling to produce the PTX method name and we did not spend the time to predict how names are mangled in CUDA. Instead, we simply compiled the CUDA entry and inspect the PTX result file to obtain the function signature needed. An example entry signature is *"_Z5entryPiS_ii".* That string is pasted into the Rootbeer JNI runtime to load the correct function to run. The CUDA entry method is shown below.

```
__global__ void entry(int * handles, int * exceptions,
  int numThreads, int usingKernelTemplates){

  int totalThreadId = getThreadId();

  if(totalThreadId < numThreads){
    int exception = 0;
    int handle;
    if(usingKernelTemplates){
      handle = handles[0];
    } else {
      handle = handles[totalThreadId];
    }
    %%invoke_run%%(handle, &exception);
    if(%%using_exceptions%%){
      exceptions[totalThreadId] = exception;
    }
  }
}
```
Figure 6.1.1 - CUDA Entry Point for Rootbeer

The CUDA entry method contains four parameters. These are described in the table below. Thorsten Kiefer contributed the numThreads API inside the CUDAContext.

Parameter	**Description**
handles	Contains the top-level object handles to the Kernel objects. These object handles reference locations in the object_space.
exceptions	The place to write exceptions back to the CPU host. Contains the same number of spaces as the handles parameter. Each Kernel writes their exception into exceptions[loop_control] or leaves as zero for no exception (Out parameter)
numThreads	The number of threads to run on the GPU. The launched count is gridDim.x multiplied by blockDim.x (in the one dimensional case), but the application may not have a square thread count. That protects the GPU from running threads where data has not been initialized.

usingKernelTemplates	Marked as zero if not using kernel templates, otherwise marked as one. If we are using kernel templates, then the handle is located at element zero from handles; otherwise, the handle is located according to the loop_control.

Figure 6.1.2 - Parameters of the CUDA Entry Point

The CUDA entry starts out by finding the totalThreadId. Based on totalThreadId and numThreads, the code checks if the current thread should execute. If the current thread executes and it should not, the memory has not been setup and a segmentation fault will occur.

After that a handle is taken from handles[totalThreadId]. This contains the this-pointer for the top level Kernel object on that thread. After is a line:

```
%%invoke_run%%(handle, &exception);
```

Figure 6.1.3 - Specializing CUDA Entry Point for GPUMethod

The %%invoke_run%% is replaced using string replacement by the fully qualified name of the 'gpuMethod' method of the application Kernel class while compiling this hard-coded method. This is where the CUDA entry calls the users application code. After that, the exception returned is written to exceptions[totalThreadId] if exceptions have been enabled. Enabling exceptions is done with a string replace while 'usingKernelTemplates' is done with a parameter because enabling exceptions is done at compile time and kernel templates are configured at runtime. Using a string replace at compile time saves time transferring data at runtime.

6.2 Instance Methods

Instance methods are supported by writing a fully qualified function name that passes the thisref handle as the first parameter, followed by parameters (named parameter0, parameter1, etc.) and finally an exception as an out parameter. This function signature is shown in the figure below. Names for functions are mangled using a reduced set of type ids that only contain types of code being generated. Each argument is separated by _ and array types are preceded by n a's (where n is the number of dimensions in the array).

```
__device__
void arraysum_ArraySumKernel_gpuMethod0_(int thisref, int * exception)
```

Figure 6.2.1 – Instance Method Function Signature

After the function identifier, we write locals to the beginning of the function. Handles are represented as integers and initialized to null (-1). Since locals are global to methods in Java Bytecode, they are also global to functions in our CUDA C. This is also useful when compiling the code with a C99 compiler for use in the native emulator. The locals for this method are shown in the figure below. Primitive types from Java use the existing primitive types in CUDA.

```
  int r0 = -1;
  int i0 = 0;
  int $r1 = -1;
  int $r2 = -1;
```

Figure 6.2.2 – Instance Method Local Initialization

After the locals, come the instructions. They start out by assigning the thisref to r0, since the bytecode must do this from Jimple. Then i0 is initialized with the x thread index. The exception out parameter is checked and if it is non-zero, the method returns and the exception will either be caught in a higher function or transferred to the CPU. In this case, the method returns void, so return is simply used. In cases where the method returns a value, zero is returned, but will not be used in the higher method.

Then the thread id is checked against zero. If the thread id is zero, then initEngine is called, otherwise the rest of the threads wait after initEngine for thread zero. All threads execute a threadfence and syncthreads and then they obtain the initialized engine from a static field. Only one thread should initialize the engine so they all share the same instance. After, a virtually invoked method (FloatArraySum.sum) is called to sum the array contents. To make all theads converge after, a Syncthreads is called. Finally, thread executes printResult. Notice that the handle for the static engine was read twice from a field because the Java Bytecode came to the Rootbeer compiler unoptimized. The developer can remove this spurious read or future compiler work can optimize this away. After everything, execution returns from the method.

```
  r0  =  thisref ;
  i0  = org_trifort_rootbeer_runtime_RootbeerGpu_
    getThreadIdxx(exception);
  if(*exception != 0){
    return;
  }
  if(i0 != 0){
    goto label0;
  }
  arraysum_ArraySumKernel_initEngine0_(r0, exception);
  if(*exception != 0){
    return;
  }
label0:
  org_trifort_rootbeer_runtime_RootbeerGpu_threadfence(exception);
  if(*exception != 0){
    return;
  }
  org_trifort_rootbeer_runtime_RootbeerGpu_syncthreads(exception);
  if(*exception != 0){
    return;
  }
  $r1 = static_getter_arraysum_ArraySumKernel_engine(exception);
  invoke_arraysum_FloatArraySum_sum0_($r1, exception);
  if(*exception != 0){
    return;
  }
  org_trifort_rootbeer_runtime_RootbeerGpu_syncthreads(exception);
  if(*exception != 0){
    return;
  }
  if(i0 != 0){
    goto label1;
  }
  $r2  = static_getter_arraysum_ArraySumKernel_engine(exception);
  invoke_arraysum_FloatArraySum_printResult0_($r2, exception);
  if(*exception != 0) {
    return;
  }
label1:
  return;
```
Figure 6.2.3 – Instance Method Body

6.3 Static Methods

Static methods are supported by simply not passing in a thisref as the first parameter and compiling the method as normal. The CUDA code generator knows how to properly emit the function signatures without this parameter.

6.4 Virtual Method Calls

After class loading, if a method is deemed to have only one call target, it is directly called. Otherwise, an invoke_ method is called that reads the object type id from the object and calls the appropriate concrete method. This is shown in the figure below.

```
__device__
void invoke_arraysum_FloatArraySum_sum0_(int thisref, int *
exception){

  char * thisref_deref;
  GC_OBJ_TYPE_TYPE derived_type;
  if(thisref == -1){
    *exception = 25599;
    return;
  }
  thisref_deref = org_trifort_gc_deref(thisref);
  derived_type = org_trifort_gc_get_type(thisref_deref);
  if(0){
  } else if(derived_type == 3365){
```

```
015     arraysum_FloatArraySum_sum0_(thisref, exception);
016   } else if(derived_type == 14532){
017     arraysum_IntArraySum_sum0_(thisref, exception);
018   }
019   return;
020 }
```

Figure 6.4.1 - Resolving virtual method calls

At line 6, the object handle of the virtual method is checked against a null pointer. If it is null, a NullPointerException is thrown (exception type 25599 in this case). After that if the derived type is 3365, the FloatArraySum.sum is called, otherwise, if it is 14532, the IntArraySum.sum is called. These type numbers can be found in the ~/.rootbeer/types file.

6.5 Instance Fields

For each SootField that is found in a depth-first search walk from the gpuMethod entry point, CUDA instance and static getter and setter functions are generated. If they are not used, they are removed by dead code elimination before passing to nvcc (Section 6.21). The getter and setter for FloatArraySum.array is shown below.

```
__device__
int instance_getter_arraysum_FloatArraySum_array(int thisref, int
* exception){

  GC_OBJ_TYPE_TYPE derived_type;
  int offset;
  char * thisref_deref;
  if(thisref == -1){
    *exception = 25599;
    return 0;
  }
  thisref_deref = org_trifort_gc_deref(thisref);
  derived_type = org_trifort_gc_get_type(thisref_deref);
  offset = org_trifort_type_switch0(derived_type);
  return *(( int *) &thisref_deref[offset]);
}

__device__
void instance_setter_arraysum_FloatArraySum_array(int thisref,
int parameter0, int * exception){

  GC_OBJ_TYPE_TYPE derived_type;
  int offset;
  char * thisref_deref;
  if(thisref == -1){
    *exception = 25599;
    return;
  }
  thisref_deref = org_trifort_gc_deref(thisref);
  derived_type = org_trifort_gc_get_type(thisref_deref);
  offset = org_trifort_type_switch0(derived_type);
  *(( int *) &thisref_deref[offset]) = parameter0;
}
```
Figure 6.5.1 - Instance Field Getter / Setter Methods

In the figure above, org_trifort_type_switch0 is used. The SootFields in Soot have a declaring class that is bound to the handle. We want to use the object layout bound to the instance object, not the handle. The type switches translate from a derived type to an offset. They are compressed for bodies of type switches that are the same and they are placed in the same function. This was needed because the type switch code made the CUDA code too big to compile with nvcc. An example type switch function is shown below.

```
__device__
int org_trifort_type_switch0(int type){

  int offset;
  switch(type){
  case 3365:
    offset = 32;
    break;
  case 14532:
    offset = 40;
    break;
  default:
    offset = -1;
    break;
  }
  return offset;
}
```
Figure 6.5.2 – Field Offset Switch Function

You can see in the above type switch that if the type is 3365 (FloatArraySum) the offset into the object is 32 for the array field and if it is 14532 (IntArraySum) the offset is 40.

6.6 Static Fields

Static memory is at the beginning of the object_space. An example of a static field getter is in the figure below. At line 6, we deference memory location zero, and then use a precomputed offset for this static field at line 7. The precomputed offsets for generated serialization and CUDA code are the same.

```
001 __device__
002 int static_getter_arraysum_ArraySumKernel_engine(int * exception){
003   char * thisref_deref = org_trifort_gc_deref(0);
004   return *(( int *) &thisref_deref[28]);
005 }
```

Figure 6.6.1 – Static Field Getter Method

6.7 New Objects

To create new objects on the GPU, special init functions are created. The init functions call org_trifort_gc_malloc given a size that is computed from the Jimple data structures. If malloc returns -1, an OutOfMemoryError is thrown that can be caught from the application code, or transferred up to the CPU, where it will be thrown on the CPU. The init function name is appended with a specific hard coded UUID to prevent naming conflicts with application methods named init. (In Java Bytecode the name is a special name of <init> that cannot have conflicts).

```
__device__
int arraysum_FloatArraySum_
  initab850b60f96d11de8a390800200c9a660_19_(char parameter0, int *
  exception)
{
  int r0 = -1;
  int r1 = -1;
  int $r2 = -1;
  int thisref;
  char * thisref_deref;
  thisref = -1;
  org_trifort_gc_assign(&thisref, org_trifort_gc_malloc(64));
  if(thisref == -1){
    *exception = 25777;
    return -1;
  }
```

Figure 6.7.1 – Constructor Initialization

After the malloc, the header information is filled in (including the size of the object and the type id and monitor variable). Some reserved fields for a future garbage collector are also initialized, but they will most likely change. A garbage collector was attempted, but deadlocks occurred due to unknown GPU thread scheduling behavior at the time. After the heap is initialized, all Java fields are initialized with the default values that would be used from the CPU JVM.

```
  thisref_deref = org_trifort_gc_deref(thisref);
  org_trifort_gc_set_count(thisref_deref, 3);
  org_trifort_gc_set_color(thisref_deref, COLOR_GREY);
  org_trifort_gc_set_type(thisref_deref, 3365);
  org_trifort_gc_set_ctor_used(thisref_deref, 1);
  org_trifort_gc_set_size(thisref_deref, 64);
  org_trifort_gc_init_monitor(thisref_deref);
  instance_setter_arraysum_FloatArraySum_array(thisref, -1,
    exception);
  instance_setter_arraysum_FloatArraySum_descriptor(thisref, -1,
    exception);
  instance_setter_arraysum_FloatArraySum_newBuffer(thisref, -1,
    exception);
  instance_setter_arraysum_FloatArraySum_sum(thisref, 0,
    exception);
```

Figure 6.7.2 – Constructor Body

After the header and fields are initialized, the body of the constructor for that init method is emitted into the CUDA constructor. This is shown in figure 6.7.3 below.

```
  r0  =  thisref ;
  r1  =  parameter0 ;
  $r2  = (int)  r1 ;
  instance_setter_arraysum_FloatArraySum_array(r0, $r2,
    exception);
  if(*exception != 0){
    return 0;
  }
  return r0;
```

Figure 6.7.3 - Constructor Body #2

6.8 Instance Synchronized Methods

Synchronized methods are implemented by locking a re-entrant mutex in the object header of the method's current object. Care must be taken to make the nvcc static analyzer. First, instead of using "while(true)" we have a counter that always gets reset at a different part of the loop, causing the infinite loop that we want. Two avoid deadlock, we need different behavior on Windows and Unix. On Windows, we need to reset the counter before the critical section and on Linux and Mac we need to reset the counter after the critical section. This is because the nvcc compilers are slightly different between these two platform configurations and emit code differently. The Windows compiler depends on Visual Studio (cl.exe) while the Linux and Mac compilers depend on g++. An example synchronized method in CUDA is shown below.

```
__device__
float arraysum_FloatArraySum_doubleResult7_7_(int thisref, float
  parameter0, int * exception){

  int r0 = -1;
  float f0 = 0;
  float $f1 = 0;
  int id;
  char * mem;
  int count;
  int old;
  char * thisref_synch_deref;
  id = getThreadId();
  mem = org_trifort_gc_deref(thisref);
  mem += 16;
  count = 0;
  while(count < 100){
    old = atomicCAS((int *) mem, -1 , id);
    if(old == -1 || old == id){
      ////////////////////////////////////
      // the below three lines contain the
      // body of the original java method
      ////////////////////////////////////
      r0 = thisref;
      f0 = parameter0;
026       $f1 = f0 * 2.0F;
027       org_trifort_exitMonitorMem(mem, old);
028       return $f1;
029     } else {
030       count++;
031       if(count > 50){
032         count = 0;
033       }
034     }
035   }
036   return 0;
037 }
```
Figure 6.8.1 - Synchronized Method

To actually implement the monitor, we use atomic compare and set with a monitor location. If the location contains -1, we set the location equal to the global GPU thread id (blockDim.x * blockIdx.x + threadIdx.x). Then we check the old value from atomicCAS. If it is -1, we enter the critical section normally and if it is the same thread id, we enter the critical section in a re-entrant manner.

Before exiting the method in any way (including exceptions) we unlock the monitor. You can see we do this at line 30 before the only valid exit (returning the result). In the case of exceptions, we unlock the monitor before returing from those too.

6.9 Static Synchronized Methods

Static Synchronized methods are supported exactly as instance synchronized methods except for the monitor location. After the static fields in the static memory, a set of static monitors exist that are all initialized to -1. There is one static monitor per class and the CUDA code generation knows the index for a given class.

6.10 Synchronized Objects

Synchronized objects are supported much like synchronized methods, except the memory location is related to the object. Care is taken to ensure that the lock is released on all exit points including method return, locking-region end and exception throw.

6.11 Instanceof

For each instanceof instruction used in the application code, an instanceof check function is generated for CUDA. An example of calling "instanceof FloatArray" is shown below and after that the contents of the check function are shown.

```
  $z0 = org_trifort_rootbeer_instanceof_arraysum_FloatArray($r1,
    exception);
  if($z0 == 0) goto label0;
```

Figure 6.11.1 – Instanceof Method Call

You can see in the code below that we obtain the type id from the object and have a switch statement where true is returned for FloatArray and everything else returns false. In more complicated examples we take into account the full class hierarchy, but only FloatArray was created with new and matches in this case. (no types of Object have been created with new in our application)

```
__device__
char org_trifort_rootbeer_instanceof_arraysum_FloatArray(int
  thisref, int * exception){

  char * thisref_deref;
  GC_OBJ_TYPE_TYPE type;
  if(thisref == -1){
    return 0;
  }
  thisref_deref = org_trifort_gc_deref(thisref);
  type = org_trifort_gc_get_type(thisref_deref);
  switch(type){
    case 3363:
      return 1;
  }
  return 0;
}
```
Figure 6.11.2 – Instanceof Method

6.12 Exceptions

Handling exceptions requires that locks be released before returning from a method. Code is generated that does this, but only works for simple cases at the moment. There is a failing test case called the NestedMonitorTest that needs to be fixed to handle every type of nested monitor combination.

6.13 Console Printing

Console printing is achieved by mapping the PrintStream.println method to printf statements accessible from CUDA. This is incomplete because other PrintStream objects other than System.out will not work. However, it is still very useful in debugging and Aparapi will not run a Kernel that contains a System.out.println.

6.14 Strings

Strings are treated specially in Rootbeer. Differences exist in the internal representation for strings within different versions of Java. In some versions of the JVM, strings are cached and this disrupts our previous serialization algorithms. When we would use reflection to change the internal state of a String object, but it would change all instances of the String in the application.

Rootbeer uses String.toCharArray to properly access the internal char array buffer for a string to serialize. A String in Rootbeer is an object with an object header and one reference type field pointing to a character array object.

In our canonical example we assign a string constant to a field. The CUDA code generated for this purpose is displayed in the figure below.

```
int str_const = org_trifort_string_constant
  ((char *) "float-array-sum: ", exception);
instance_setter_arraysum_FloatArraySum_descriptor(r0, str_const,
  exception);
```

Figure 6.14.1 - String Constant Method Call

We have a built-in function in the supplied runtime called org_trifort_string_constant. This is used to convert the Java Bytecode string constants to String objects at runtime. The string constant built-in source is shown below. First we create a manage char[] from a native string constant (line 7). Then we take the managed char[] and pass it to the String(char[]) constructor.

```
001  __device__ int
002  org_trifort_string_constant(char * str_constant,
003    int * exception)
004  {
005    int characters;
006
007    characters = org_trifort_char_constant(str_constant,
008      exception);
009    return org_trifort_rootbeer_string_from_chars(
010      characters, exception);
011  }
```

Figure 6.14.2 - String Constant Method

String concatenation in Java is supported with StringBuilder operations inside Java Bytecode. Rootbeer has custom CUDA functions to execute StringBuilder append for all primitive types and Object, which uses Object.toString.

Another important aspect of String concatenation is converting primitive numbers to strings. Rootbeer supports converting integers, longs, floats and doubles to strings using custom CUDA code derived from open source standard c library code. This support was added by Martin Illecker as part of his Master's thesis titled "Scientific Computing in the Cloud with Apache Hadoop".

6.15 Primitive Arrays

Primitive arrays are supported with proper serialization and getter/setter function generation for each type of array used in the application. An example array getter is shown in the figure below.

```
001 __device__
002 float float__array_get(int thisref, int parameter0,
003   int * exception)
004 {
005   int offset;
006   int length;
007   char * thisref_deref;
008   offset = 32+(parameter0*4);
009 
010   if(thisref == -1){
011     *exception = 25599;
012     return 0;
013   }
014   thisref_deref = org_trifort_gc_deref(thisref);
015   length = org_trifort_getint(thisref_deref, 12);
016   if(parameter0 < 0 || parameter0 >= length){
017     *exception = org_trifort_rootbeer_ArrayOutOfBoundsEx(
018       parameter0, thisref, length, exception);
019     return 0;
020   }
021   return *(( float *) &thisref_deref[offset]);
022 }
```

Figure 6.15.1 - Float Array Getter

At line 8 in the code above, the offset into the array is calculated as header_size+(parameter0*4). Parameter zero is the index into the array and four is the size of each element. Rootbeer generates different constants for each primitive size (for instance doubles will be 8 bytes per double).

Then at line 10, a null pointer check is done. After, checks on the array bounds are done and if the array index is out of bounds, and ArrayIndexOutOfBoundsException is thrown. Finally, on the last line, the array element is referenced and returned. Once the developer gets their code right, both the null-pointer check and the out-of-bounds check can be removed by configuring Rootbeer complication. This can improve speed of executing code on the GPU.

6.16 Reference Type Arrays

Reference type arrays are supported in a similar manner as primitive type arrays, except the array elements contain integer handles. Every reference type array element is 4 bytes.

6.17 Higher-Dimensional Arrays

Arrays that have a dimension greater than one are supported as reference type arrays. Each handle is to an array of a lower dimension. Finally, at the lowest level, the primitive or reference data for the array element is stored.

6.18 New Array Instances

To handle creating new instances of arrays on the GPU, we generate code for each type. An example creator for float arrays is shown in the figure below. We start out at line 7 calculating the total size of the array object. It requires the size of the object header (32 bytes) plus the size of each element (4) times the number of elements (size). Then we align this size to the next multiple of 16 bytes (lines 8 to 11). An alignment of 16 bytes is required because the minimum object size is 32 bytes and all handles are compressed by shifting the value 4-bits down.

Then we make a call to GPU malloc (line 12) and throw an OutOfMemoryError (line 14) if we don't have any more space in the GPU memory. We initialize the header (lines 18 to 26) and initialize each element of the array to the Java or Rootbeer default.

```
001  __device__ int
002  float__array_new(int size, int *
003    exception)
004  {
005    //local variable declarations omitted
006
007    total_size = (size * 4)+ 32;
008    mod = total_size % 16;
009    if(mod != 0){
010      total_size += (16 - mod);
011    }
012    thisref = org_trifort_gc_malloc(total_size);
013    if(thisref == -1){
014      *exception = 23516;
015      return -1;
016    }
017    thisref_deref = org_trifort_gc_deref(thisref);
018    org_trifort_gc_set_count(thisref_deref, 0);
019    org_trifort_gc_set_color(thisref_deref, COLOR_GREY);
020    org_trifort_gc_set_type(thisref_deref, 5262);
021    org_trifort_gc_set_ctor_used(thisref_deref, 1);
022    org_trifort_gc_set_size(thisref_deref, total_size);
023    org_trifort_setint(thisref_deref, 12, size);
024    for(i = 0; i < size; ++i){
025      float__array_set(thisref, i, 0, exception);
026    }
027    return thisref;
028  }
```
Figure 6.18.1 - Float Array Creation

6.20 Class constants

Class constants are required to support Object.toString. They are supported in Rootbeer using special serialization of only the name inside the class constant to a special area of memory. A class constant is used in generated CUDA using a statement similar to the following. The class types are taken from the Java Bytecode and converted to a number. (Class constant zero is used here to represent IntArraySum.class).

```

$r13 = org_trifort_classConstant(0);

```

Figure 6.20.1 - Class Constant Method Call

The class constant object fetch code is shown below. We keep the pointer to the class constant space in shared memory (m_local[2]). Then once we have the constant space, we obtain the handle to the object_space of the class constant object.

```
__device__ int
org_trifort_classConstant(int type_num){
  int * temp = (int *) m_local[2];
  return temp[type_num];
}
```

Figure 6.20.2 - Class Constant Method

6.21 Shared Memory

Writing to shared memory is supported with a set of included methods for each primitive type. An example shared memory write function for integers is shown below. First we check the array bounds from lines 5 to 11 (if they are enabled). Then we write each byte of the integer into the shared memory.

```
001  __device__ void
002  org_trifort_rootbeer_runtime_RootbeerGpu_setSharedInteger
003    (int index, int value, int * exception)
004  {
005  #ifdef ARRAY_CHECKS
006    if(index < 0 || index + 4 >= %%shared_mem_size%%){
007      *exception = org_trifort_rootbeer_arrayOutOfBoundsEx(
008         index, 0, %%shared_mem_size%%, exception);
009      return;
010    }
011  #endif
012    m_shared[index] = (char) (value & 0xff);
013    m_shared[index + 1] = (char) ((value >> 8)  & 0xff);
```

```
  m_shared[index + 2] = (char) ((value >> 16) & 0xff);
  m_shared[index + 3] = (char) ((value >> 24) & 0xff);
}
```
Figure 6.21.1 - Shared memory writer

6.22 Dead Function Elimination

Simplistic dead function elimination is done on the generated CUDA code before sending to nvcc. If a function is not called from a method reachable from the CUDA entry, it is eliminated. We only support the subset of C that our CUDA code generator produces for this task, but it uses a well-tested, state-machine driven parse.

This allows our hand coded CUDA runtime to be combined with automatically generated code. The CUDA runtime code may call code that is not automatically generated, because that feature was not needed in the application. If we do not eliminate the unused code, parts will make calls into missing generated code and cause compiler errors in nvcc. This functionality is found in the org.trifort.rootbeer.deadmethods package.

6.23 CUDA Code Generation Conclusions

In conclusion, most of Java Bytecode can be represented using generated CUDA. While we are extremely happy with our results, there are a few areas that can be improved.

First, OpenCL cannot be used within Rootbeer due to the lack of recursive OpenCL functions. This precludes the use of AMD GPUs and the Intel MIC, which contains many weak Xeon Phi cores. However, a future work item could be to support OpenCL if no recursion was detected.

Next, exceptions are supported using an out parameter that is subsequently checked after each method. Migration to generating using the libNVVP library from NVIDIA for direct PTX code generation may help in this area regarding performance.

Chapter 7

Performance

In this chapter we will discuss 4 GPU programs created with Rootbeer. In examples where there is O(n^3) computation per O(n) elements of input, we obtain a speedup because the computation is larger than the serialization time. In examples with less computation, the CPU is faster because serialization becomes a bottleneck. An overview of the programs is below.

Program	Time Complexity	Speed
Matrix Multiply	O(n^3)	94.9x speedup
HMM Viterbi Path	O(n^2*t)	102.7x speedup
Scan	O(n)	7x slowdown
Histogram	O(n)	1.1x slowdown

Figure 7.1 – Overview of Performance Examples

For each example we have a separate section describing the example, listing the code and showing in-depth timings.

7.1 Matrix Multiply Example

Matrix Multiply is a classic example of using a GPU to obtain a speedup. There are O(n^3) operations per O(n) of data, so serialization time is easy to overcome. Chapter One has covered many types of Matrix Multiply examples; this section will focus on “Rootbeer Shared”, the example that obtained a 94.9x speedup over a 4-core Java CPU version. The kernel class for the shared matrix multiply is listed below.

```
public class MatrixKernel implements Kernel {

  private float[] a;
  private float[] b;
  private float[] c;

  private static final int SIZE_FLOAT = 4;
  private static final int TILE_SIZE = 16;
  private static final int SHARED_B_START =
    TILE_SIZE * TILE_SIZE;

  public MatrixKernel(float[] a, float[] b, float[] c){
    this.a = a;
    this.b = b;
    this.c = c;
  }

  @Override
  public void gpuMethod() {
    float[] registerA = a;
    float[] registerB = b;
    float[] registerC = c;

    int blockIdxx = RootbeerGpu.getBlockIdxx();
    int blockIdxy = RootbeerGpu.getBlockIdxy();

    int threadIdxx = RootbeerGpu.getThreadIdxx();
    int threadIdxy = RootbeerGpu.getThreadIdxy();

    int width = 2048;
    int aBegin = width * TILE_SIZE * blockIdxy;
    int aEnd = aBegin + width - 1;
    int aStep = TILE_SIZE;
    int bBegin = TILE_SIZE * blockIdxx;
    int bStep = TILE_SIZE * width;
    float sum = 0;
    int arrayIndex = width * threadIdxy + threadIdxx;

    int a = aBegin;
    int b = bBegin;

    while(a <= aEnd){
      float valueA = registerA[a + arrayIndex];
      float valueB = registerB[b + arrayIndex];

      int indexA = ((threadIdxy * TILE_SIZE) + threadIdxx);
      int indexB = SHARED_B_START +
        ((threadIdxy * TILE_SIZE) + threadIdxx);

      RootbeerGpu.setSharedFloat(indexA * SIZE_FLOAT, valueA);
      RootbeerGpu.setSharedFloat(indexB * SIZE_FLOAT, valueB);
      RootbeerGpu.syncthreads();

      for(int k = 0; k < TILE_SIZE; ++k){
        indexA = ((threadIdxy * TILE_SIZE) + k);
        indexB = SHARED_B_START + ((k * TILE_SIZE) + threadIdxx);

        valueA = RootbeerGpu.getSharedFloat(indexA * SIZE_FLOAT);
        valueB = RootbeerGpu.getSharedFloat(indexB * SIZE_FLOAT);
        sum += valueA * valueB;
      }

      RootbeerGpu.syncthreads();

      a += aStep;
      b += bStep;
    }

    int c = width * TILE_SIZE * blockIdxy +
      TILE_SIZE * blockIdxx;
    registerC[c + arrayIndex] = sum;
  }
}
```

Figure 7.1.1 – Shared Matrix Multiply Kernel

Below we have listed the individual timings of the GPU and CPU version. You can see that the serialization, deserialization and memory copy times are small compared to the total time that the CPU requires. Therefore Rootbeer can obtain a large speedup for this O(n^3) algorithm.

Java Serialization Time	35 milliseconds
JNI Driver Memcpy to Device	55 milliseconds
GPU Execution Time	136 milliseconds
JNI Driver Memcpy From Device	27 milliseconds

Java Deserialization Time	33 milliseconds
Total GPU Time	286 milliseconds
Total CPU Time	27129 milliseconds

Figure 7.1.2 – Shared Matrix Multiply Timings on NVIDIA Tesla C2050 GPU and Four Intel Xeon Cores

7.2 HMM Example

Hidden Markov Models are an interesting machine learning example that the author is using in the Welch Lab to complete a genomic clustering analysis for computational biology.

In "Fundamentals of Speech Recognition", Rabiner and Juang [58] describe the three major questions in HMMs. In our analysis, we are only concerned with two questions: 1) given a signal, what is the optimal HMM to represent it and 2) given an HMM what is the probability that a given, possibly different, signal produced the HMM.

Our analysis takes data from the genetic mutations of 308 *Myxococcus xanthus* bacteria and attempts to cluster this complex data. The analysis has three stages:

1) Train 308 HMMs with 308 signals produced from annotated time-lapse microcinematography image stacks.
2) Produce pairwise distances for 308 HMMs by 308 signals.
3) Use hierarchical agglomerate clustering given the pairwise distances to produce a final clustering image that can be analyzed by a biologist.

Training each HMM with 2240 states and a signal length of 8000 requires approximately 39 minutes each. Since there are 308 to train, the total training time is approximately 8.3 core days. The training phase can be accelerated using a GPU by focusing on the Viterbi path algorithm. We have obtained a 102.7x speedup using a single GPU, therefore the total time is now 1.992 hours. This proven speedup will be useful for the longer required signal of 62500 and also for an analysis using wavelet transformed raw image frames. In addition, a general purpose HMM library could be made with Rootbeer that can make real-time analysis feasible with HMMs in the areas of signal processing and medical imaging and voice recognition, to name a few.

The next step is to compute 308x308 pairwise distances. Each requires approximately 16 seconds, arriving at approximately 17 days of computation time. We have reduced the computation time for each pairwise distance by 16x using Rootbeer and each pair now requires 1 second each. With our improvements, the total time is now 1.097 days. With this speedup, an analysis based on raw image frames and the wavelet transform may now be possible using a 16 GPU system.

As a basis for our GPU HMM code, we are using Jahmm [59] by Jean-Marc François. We are using the k-means approach to initializing the HMM states from zero knowledge and a scaled forward-backward calculator to estimate the likelihood probability. We created a custom parallel GPU Viterbi algorithm and it is listed in the figure below.

```java
package be.ac.ulg.montefiore.run.jahmm.gpu.gpu;

import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.RootbeerGpu;

import be.ac.ulg.montefiore.run.jahmm.gpu.gpu.HmmGaussianReal;
import be.ac.ulg.montefiore.run.jahmm.gpu.gpu.OpdfGaussianGPU;

public class ViterbiKernel implements Kernel {

  private HmmGaussianReal hmm;
  private double[] observations;
  private double[][] delta;
  private int[][] psy;
  private int[] barrier_in;
  private int[] barrier_out;

  public ViterbiKernel(double[][] delta, int[][] psy,
      HmmGaussianReal hmm, double[] observations,
      int[] barrier_in, int[] barrier_out){

    this.delta = delta;
    this.psy = psy;
    this.hmm = hmm;
    this.observations = observations;
    this.barrier_in = barrier_in;
    this.barrier_out = barrier_out;
  }

  public void gpuMethod() {
    int threadId = RootbeerGpu.getThreadId();
    int obsSize = observations.length;

    for(int t = 1; t < obsSize; ++t){
      double observation = observations[t];

      computeStep(hmm, observation, t, threadId);
      RootbeerGpu.threadfenceSystem();
      globalSync(t);
    }
  }

  private void computeStep(HmmGaussianReal hmm, double o,
    int t, int j)
  {
    double minDelta = Double.MAX_VALUE;
    int min_psy = 0;
    int numStates = hmm.nbStates();
    double[][] a = hmm.getA();

    for (int i = 0; i < numStates; i++) {
      double delta_value = delta[t-1][i];
      double thisDelta = delta_value - StrictMath.log(a[i][j]);

      if (minDelta > thisDelta) {
        minDelta = thisDelta;
        min_psy = i;
      }
    }

    OpdfGaussianGPU opdf = hmm.getOpdf(j);
    delta[t][j] = minDelta - StrictMath.log(opdf.probability(o));
    psy[t][j] = min_psy;
    RootbeerGpu.threadfenceSystem();
  }
}
```

Figure 7.2.1 – Parallel HMM Viterbi Path with Rootbeer

The major parallel part of this code is the globalSync primitive. It is designed to synchronize threads across GPU blocks. This can be difficult to do on the Tesla C2050 GPU because blocks are never removed from the GPU until they are completed (see section 2.6). To make our globalSync primitive work, we needed to carefully tune the number of threads, blocks and register count. We were able to use 14 blocks and 160 threads without running out of GPU global ram. The block and thread count was specific to the Tesla C2050 and our code may simply deadlock if moved to another NVIDIA GPU model. The globalSync primitive is shown in the code below. We would like to thank Martin Illecker for bringing the globalSync [60] algorithm to our attention, however this is a modified version.

```
  private void globalSync(int goal_value){
    int[] local_barrier_in = barrier_in;
    int[] local_barrier_out = barrier_out;
    int count = 0;
    int iter = 0;
    int thread_value = 0;
    int needed_count = local_barrier_in.length;

    while(count < needed_count){
      if(RootbeerGpu.getThreadIdxx() == 0){
        local_barrier_in[RootbeerGpu.getBlockIdxx()] =
          goal_value;
      }
      RootbeerGpu.threadfenceSystem();
      if(RootbeerGpu.getThreadIdxx() < needed_count){
        thread_value =
          local_barrier_in[RootbeerGpu.getThreadIdxx()];
      } else {
        thread_value = -1;
      }
      int match;
      if(thread_value == goal_value){
        match = 1;
      } else {
        match = 0;
      }
      count = RootbeerGpu.syncthreadsCount(match);
    }
    count = 0;
    while(count < needed_count){
      if(RootbeerGpu.getThreadIdxx() == 0){
        local_barrier_out[RootbeerGpu.getBlockIdxx()] =
          goal_value;
      }
      RootbeerGpu.threadfenceSystem();
      if(RootbeerGpu.getThreadIdxx() < needed_count){
        thread_value =
          local_barrier_out[RootbeerGpu.getThreadIdxx()];
      } else {
        thread_value = -1;
      }
      int match;
      if(thread_value == goal_value){
        match = 1;
      } else {
        match = 0;
      }
      count = RootbeerGpu.syncthreadsCount(match);
    }
  }
```
Figure 7.2.2 – HMM Global Sync method in Java

Below we have listed the individual timings of the GPU and CPU version for the HMM Viterbi Path algorithm. You can see that the GPU is approximately 102.7x faster than a single core CPU. Since the computation kernel takes a long time, the serialization and memory copy times do not drastically affect the total GPU time and we obtain a good speedup with this O(n^2*t) algorithm.

Java Serialization Time	387 ms
JNI Driver Memcpy to Device	142 ms
GPU Execution Time	19568 ms
JNI Drive Memcpy From Device	141 ms
Java Deserialization Time	289 ms
Total GPU Time	22829 ms
Total CPU Time	2346225 ms

Figure 7.2.3 – HMM Viterbi Path on NVIDIA Tesla C2050 GPU and Single Intel Xeon Core

After the HMM training and likelihood calculations are complete we are using a hierarchical agglomerate clustering library from Lars Behnke on github [61]. This phase takes very little time compared to the previous two. After that is complete the clustering is displayed using iTOL [62]. A sample resultant output is shown below. You can see in this preliminary result that similar bacteria strains are colored the same and often close to each other in the chart. The biologist will then use this to try to make connections between the high-level behavior of the organism and the organization of the machinery inside the cell

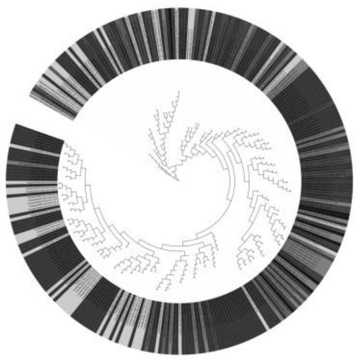

Figure 7.2.4 – Clustering of 308 mutated *Myxococcus xanthus* strains

7.3 Scan Example

Scan is a parallel primitive that can sum the array elements in parallel. We have ported a parallel scan example from the CUDA examples. We would like to note that this GPU algorithm is not original work, but it is important to measure the speed of other GPU algorithms and it shows that a fairly complicated NVIDIA example can be directly ported to use Rootbeer using the provided GPU primitives. There is not a speedup in this example because the serialization and memory copies to the device are a bottleneck.

```
package org.trifort.rootbeer.examples.scan;

import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.RootbeerGpu;

public class GPUScanKernel implements Kernel {

  private float[][] inputData;
  private float[][] outputData;
  private static final int INT_SIZE = 4;

  public GPUScanKernel(float[][] inputData,
    float[][] outputData){
    this.inputData = inputData;
```

```
      this.outputData = outputData;
    }

    public void gpuMethod(){
      int block_idxx = RootbeerGpu.getBlockIdxx();
      int thid = RootbeerGpu.getThreadIdxx();
      int offset = 1;

      float[] localInputData = inputData[block_idxx];
      int n = localInputData.length;
      RootbeerGpu.setSharedFloat((2*thid)*INT_SIZE,
        localInputData[2*thid]);
      RootbeerGpu.setSharedFloat((2*thid+1)*INT_SIZE,
        localInputData[2*thid+1]);

      for(int d = n >> 1; d > 0; d >>= 1){
        RootbeerGpu.syncthreads();
        if(thid < d){
          int ai = offset*(2*thid+1)-1;
          int bi = offset*(2*thid+2)-1;
          float temp_ai = RootbeerGpu.getSharedFloat(ai*INT_SIZE);
          float temp_bi = RootbeerGpu.getSharedFloat(bi*INT_SIZE);
          RootbeerGpu.setSharedFloat(bi*INT_SIZE,
            temp_ai + temp_bi);
        }
        offset *= 2;
      }

      if(thid == 0){
        RootbeerGpu.setSharedFloat((n-1)*INT_SIZE, 0);
      }

      for(int d = 1; d < n; d *= 2){
        offset >>= 1;
        RootbeerGpu.syncthreads();
        if(thid < d){
          int ai = offset*(2*thid+1)-1;
          int bi = offset*(2*thid+2)-1;

          float tA = RootbeerGpu.getSharedFloat(ai*INT_SIZE);
          float tB = RootbeerGpu.getSharedFloat(bi*INT_SIZE);

          RootbeerGpu.setSharedFloat(ai*INT_SIZE, tB);
          RootbeerGpu.setSharedFloat(bi*INT_SIZE, tA + tB);
        }
      }

      RootbeerGpu.syncthreads();
      outputData[block_idxx][2*thid] =
        RootbeerGpu.getSharedFloat((2*thid)*INT_SIZE);
      outputData[block_idxx][2*thid+1] =
        RootbeerGpu.getSharedFloat((2*thid+1)*INT_SIZE);
    }
  }
```

Figure 7.3.1 – Parallel GPU Scan with Rootbeer

Below we have listed the individual timings of the GPU and CPU version. We do not see a speedup using Rootbeer because the serialization and deserialization times are not fast enough. Improving the serialization speed is possible but has not been included in this work at this time.

Java Serialization Time	446 ms
JNI Driver Memcpy to Device	248 ms
GPU Execution Time	84 ms
JNI Drive Memcpy From Device	292 ms
Java Deserialization Time	411 ms
Total GPU Time	1484 ms
Total CPU Time	208 ms

Figure 7.3.2 - Scan Timings on NVIDIA Tesla C2050 GPU and Single Intel Xeon Core

7.4 Histogram Example

Parallel histogram is an algorithm that can be used in parallel garbage collection, as described in [38]. We have ported a parallel histogram example from the CUDA examples and, again, would like to note that the GPU algorithm is not original work. Again, this example shows that a complicated example from NVIDIA can be ported to Rootbeer. The code is shown below.

```
package org.trifort.rootbeer.examples.hist;

import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.RootbeerGpu;

public class GPUHistKernel implements Kernel {

  private int[] inputData;
  private int[] outputData;

  public static final int BIN_COUNT = 64;
  public static final int BYTE_SIZE = 1;
  public static final int SHORT_SIZE = 2;
  public static final int INT_SIZE = 4;
  public static final int THREAD_N = 192;
  public static final int BLOCK_MEMORY = THREAD_N * BIN_COUNT;
  public static final int BLOCK_DATA = THREAD_N * 63;
  public static final int DATA_N = 96000000 / INT_SIZE;
  public static final int MAX_BLOCK_N = 16384;

  public GPUHistKernel(int[] inputData, int[] outputData){
    this.inputData = inputData;
    this.outputData = outputData;
  }

  private int min(int left, int right){
    if(left < right){
      return left;
    } else {
      return right;
    }
  }

  private int mul24(int a, int b){
    return (a & 0xFFFFFF) * (b & 0xFFFFFF);
  }

  private void addData64(int threadPos, int data){
    int index = threadPos + mul24(data, THREAD_N);
    short value = RootbeerGpu.getSharedShort(index * SHORT_SIZE);
    ++value;
    RootbeerGpu.setSharedShort(index * SHORT_SIZE, value);
  }

  public void gpuMethod(){
    int block_dimx = RootbeerGpu.getBlockDimx();
    int block_idxx = RootbeerGpu.getBlockIdxx();
    int thread_idxx = RootbeerGpu.getThreadIdxx();

    //Global base index in input data for current block
    int baseIndex = mul24(BLOCK_DATA, block_idxx);

    //Current block size, clamp by array border
    int dataSize = min(DATA_N - baseIndex, BLOCK_DATA);

    //Encode thread index in order to avoid bank conflicts in
    //s_Hist[] access: each half-warp accesses consecutive shared
    //memory banks and the same bytes within the banks
    int threadPos =
      //[31 : 6] <== [31 : 6]
      ((thread_idxx & (~63)) >> 0) |
      //[5  : 2] <== [3  : 0]
      ((thread_idxx &    15) << 2) |
      //[1  : 0] <== [5  : 4]
      ((thread_idxx &    48) >> 4);

    int end_position = GPUHistConstants.BLOCK_MEMORY;
    for(int pos = thread_idxx; pos < end_position;
        pos += block_dimx){
      RootbeerGpu.setSharedInteger(pos * INT_SIZE, 0);
    }

    RootbeerGpu.syncthreads();

    ////////////////////////////////////////////////////////////
    // Cycle through current block, update per-thread histograms
    // Since only 64-bit histogram of 8-bit input data array is
    // calculated, only highest 6 bits of each 8-bit data element
    // are extracted, leaving out 2 lower bits.
    ////////////////////////////////////////////////////////////

    //read the handle from the field into a register because it
    //is used repeatedly in a loop
    int[] localInputData = inputData;
    int[] localOutputData = outputData;

    for(int pos = thread_idxx; pos < dataSize; pos += block_dimx){
      int item = localInputData[pos];
      addData64(threadPos, (item >>  2) & 0x3F);
      addData64(threadPos, (item >> 10) & 0x3F);
      addData64(threadPos, (item >> 18) & 0x3F);
      addData64(threadPos, (item >> 26) & 0x3F);
    }

    RootbeerGpu.syncthreads();

    ///////////////////////////////////////////////////////////
    // Merge per-thread histograms into per-block and write to
    // global memory. Start accumulation positions for half-warp
    // each thread are shifted in order to avoid bank conflicts.
    // See supplied whitepaper for detailed explanations.
    ///////////////////////////////////////////////////////////
    if(thread_idxx < BIN_COUNT){
      //64 threads here
      int sum = 0;
      int value = thread_idxx;

      int valueBase = mul24(value, THREAD_N);
      int startPos = mul24(thread_idxx & 15, 4);

      //Threads with non-zero start positions wrap around the
      //THREAD_N border
      int sharedIndex = 0;
      for(int i = 0, accumPos = startPos; i < THREAD_N; i++){
        int rawIndex = (valueBase + accumPos);
        int shortIndex = rawIndex * SHORT_SIZE;
        short shrdValue = RootbeerGpu.getSharedShort(shortIndex);
        sum += shrdValue;
        if(++accumPos == THREAD_N) {
          accumPos = 0;
        }
      }

      //value is index
      RootbeerGpu.atomicAddGlobal(localOutputData, value, sum);
    }
  }
}
```

Figure 7.4.1 – Parallel GPU Histogram with Rootbeer

Below we have listed the individual timings of the GPU and CPU version. Similar to the parallel scan, we are not seeing a speedup due to the serialization and memory copy pressures.

Java Serialization Time	45 ms
JNI Driver Memcpy to Device	45 ms
GPU Execution Time	10 ms
JNI Drive Memcpy From Device	53 ms
Java Deserialization Time	43 ms
Total GPU Time	194 ms
Total CPU Time	175 ms

Figure 7.4.2- Histogram Timings on NVIDIA Tesla C2050 GPU and Single Intel Xeon Core

7.5 Performance Conclusions

In conclusion, Rootbeer can speedup processing if there is approximatly O(n^3) work that needs to be done for O(n) elements of data that need to be serialized. In cases where there is approximately O(n) work per O(n) elements of data, we do not see any speedups.

Chapter 8

Conclusions
Recommendations
Future Work

In this chapter we will summarize this work, give recommendations on supporting Java on GPUs in an industrial setting and note areas of future research.

8.1 Conclusions

This research started as an attempt to ease programming on GPUs. With better serialization and CUDA code generation support, the developer can spend more time on choosing a “cut-point” to launch to the GPU and configure the thread/block/register count.

We have shown that Rootbeer has good wall-clock performance on our test cases demonstrating that it is feasible to program GPUs from Java. Therefore we conclude that Rootbeer makes programming GPUs easier. Rootbeer is well known in the GPU software community, with over 960 stars on github, the openjdk domain linking to Rootbeer and a statement of state-of-the-art in a DARPA grant solicitation.

8.2 Recommendations

Currently GPUs cannot be used from Java in an industrial setting where clients of Oracle and NVIDIA are expecting perfect operation for all features in Java. In order for this to happen, several problems must be solved.

The first problem is that NVIDIA GPUs do not support dynamically changing the instruction memory while the GPU is operating. NVIDIA must support this for an industrial version of Rootbeer because Java programs can dynamically load classes from byte arrays obtained through a network socket. With this problem, Oracle must decide if they will port the C1/C2 compilers to the GPU or leave them running solely on the CPU. If they port the C1/C2 compilers to run on the GPU they must re-write the code in CUDA and aggressively tune the machine instructions emitted to match performance of C1/C2 on the CPU. Otherwise the compilation from Java Bytecode to GPU assembly instructions will be too slow for JIT to be a viable strategy. Also, if the compilers are ported to the GPU, they must phrase compilation in terms of parallel processors or they will most likely see a performance degradation due to not using all 448 cores (Tesla C2050) since each core is clocked at a much lower clock rate (around 1.1 GHz) and the cores are not meant for branchy code. If a developer were to achieve compiling in parallel they will have to either schedule 448 methods to compile at once, facing divergence issues, or create a micro compilation engine that compiles pieces cooperatively with 448 threads, or otherwise create entirely new algorithms and data structures. This may be possible, but will most likely be a larger time and resource investment than the option listed below.

The other option is for Oracle to leave the C1/C2 compilers on the GPU. If this is chosen, then NVIDIA must provide first-class support for communication between the GPU and CPU while the GPU is running. With this feature from NVIDIA, the GPU can ask the existing CPU compiler to compile a method on the single-core host processor and return the results to the GPU when ready. The GPU will then take the binary fragment for the method and incorporate it into the existing GPU code by dynamically changing the instruction memory. There are many other uses for a general-purpose communication channel between the GPU and CPU, such as:

1) File I/O
2) Network I/O
3) Other device I/O such as medical imaging sensor

If there is no first-class support for CPU/GPU communication, Oracle can decide to use a static compilation step as Rootbeer did and they can use many of the ideas from Rootbeer. They can choose to precompile, and many things may be very similar to Rootbeer. If they decide to run complication right at the GPU launch point, then they will need to do static class loading for code found beyond that point. In this case it would help if there were pre-computed static call graphs for the JDK library classes. These could be added to the JAR executable package as a binary coded file with extension *.cg (for call-graph) that could be read by the JVM at GPU launch.

8.3 Future Work

At this point we would like to summarize all future work that can help to improve Rootbeer or other similar systems. Each section contains items that can be improved or enhanced.

8.3.1 Serialization Improvements

A major reason that the parallel scan example did not offer an improvement is that the serialization required too much time. In the existing serialization framework, a JNI call is made for each primitive type written to the GPU heap. If instead, each object had an allocated byte array and the primitives were written in pure Java, this would accelerate things significantly. It is well known that File-IO must be done using buffered access due to the overhead of JNI calls and this is a similar scenario.

If the serialization framework could change JDK library classes in addition to the application classes, then all items could be deserialized from the GPU heap, not just those items with sentinel and void constructors. However, there are legal terms with changing the bootclasspath, so we avoided this. Oracle, however, would be free to do this.

8.3.2 CUDA Code Generation Improvements

Two related aspects of CUDA code generation that could be improved are exceptions and stack traces. Passing in an additional out parameter to a function and checking for error conditions following execution is our technique that supports exceptions. In industrial compilers, the function simply jumps to an exception handler when the exception occurs. If Rootbeer emitted libNVVM IR that compiled directly to PTX and avoided using CUDA C, this would be better.

Regarding stack traces, it would be nice if the GPU could give a full stack trace of exceptions that occurred including line numbers from the original Java source code. This was attempted using line numbers from CUDA code, but the stack trace generation code was so slow that the computation could not return from the GPU displaying the error. The last time we checked, obtaining Java line numbers from the bytecode was buggy in Soot and this may also need additional work.

8.3.3 Rootbeer Class Loading Improvements

In order to get the Rootbeer Class Loader accepted into the Soot master branch, more work needs to be done to support another source format. The initial investment from the Rootbeer Class Loader was focused on Coffi and later the Soot community moved to another bytecode parser based on ASM. At the current time Rootbeer has not converted it’s subsystem to reading bytecode using ASM, but, if this was done, the class loader could be used by other researchers in malware analysis and website taint checking.

8.3.4 Miscellaneous Improvements

To improve ease of compilation a maven plugin could be created that would allow a user to compile their application using Rootbeer from within maven. In addition, a profiler and debugger could be added so that Rootbeer applications could be analyzed natively within the Rootbeer API. Please note that you can currently profile CUDA/Rootbeer applications using nvprof and debug using cuda-memcheck and cuda-gdb. This improvement would be better in that it would allow one to see profile and debugging information in terms of the original Java source code rather than CUDA code.

Appendix A

Listing of github stars from major technology companies.

1. Adobe
2. Alcatel-Lucent
3. Amazon
4. Apple
5. Google (2)
6. Intel
7. JBoss by Redhat
8. JetBrains
9. LinkedIn
10. Microsoft
11. MongoDB
12. NVIDIA
13. Oracle
14. Puppet Labs
15. Red Hat (3)
16. Twitter

Appendix B

This appendix contains the full Java Bytecode listing of PrintNumbers.java. To print this information, you can use the following two commands:

```
$ javac PrintNumbers.java
$ javap –v PrintNumbers
```

```
Classfile
/Users/pcpratts/Desktop/code/disser/code/PrintNumbers.class
  Last modified Oct 7, 2013; size 423 bytes
  MD5 checksum a27cc8bbfd33bb6e92ec12813e54d6ac
  Compiled from "PrintNumbers.java"
public class PrintNumbers
  SourceFile: "PrintNumbers.java"
  minor version: 0
  major version: 50
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #5.#14         //
java/lang/Object."<init>":()V
   #2 = Fieldref           #15.#16        //
java/lang/System.out:Ljava/io/PrintStream;
   #3 = Methodref          #17.#18        //
java/io/PrintStream.println:(I)V
   #4 = Class              #19            //  PrintNumbers
   #5 = Class              #20            //  java/lang/Object
   #6 = Utf8               <init>
   #7 = Utf8               ()V
   #8 = Utf8               Code
   #9 = Utf8               LineNumberTable
  #10 = Utf8               print
  #11 = Utf8               StackMapTable
  #12 = Utf8               SourceFile
  #13 = Utf8               PrintNumbers.java
  #14 = NameAndType        #6:#7          //  "<init>":()V
  #15 = Class              #21            //  java/lang/System
  #16 = NameAndType        #22:#23        //
out:Ljava/io/PrintStream;
  #17 = Class              #24            //  java/io/PrintStream
  #18 = NameAndType        #25:#26        //  println:(I)V
  #19 = Utf8               PrintNumbers
  #20 = Utf8               java/lang/Object
  #21 = Utf8               java/lang/System
  #22 = Utf8               out
  #23 = Utf8               Ljava/io/PrintStream;
  #24 = Utf8               java/io/PrintStream
  #25 = Utf8               println
  #26 = Utf8               (I)V
{
  public PrintNumbers();
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
```

```
         0: aload_0
         1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public void print();
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=2, args_size=1
         0: iconst_0
         1: istore_1
         2: iload_1
         3: bipush        10
         5: if_icmpge     21
         8: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
        11: iload_1
        12: invokevirtual #3                  // Method
java/io/PrintStream.println:(I)V
        15: iinc          1, 1
        18: goto          2
        21: return
      LineNumberTable:
        line 3: 0
        line 4: 8
        line 3: 15
        line 6: 21
      StackMapTable: number_of_entries = 2
           frame_type = 252 /* append */
             offset_delta = 2
        locals = [ int ]
           frame_type = 250 /* chop */
          offset_delta = 18

}
```
Figure B.1 – Complete Java Bytecode Listing for PrintNumbers.java

Appendix C

Examples of compiling JNI code and using javah

First you need a file with the native keyword:

```java
package org.trifort.jni_example;

public class JNIExample {

  //it is good to make public wrappers around
  //native methods
  public void printString(String str){
    printStringNative(str);
  }

  //the native keyword is present here
  private native void printStringNative(String str);
}
```

Figure C.1 – JNIExample source code

Once compiled with ant, the following directory structure is made:

- project_root
  - build
    - classes
      - org
        - trifort
          - jni_example
  - csrc (you need to make this directory)

Then you can make header files for JNI using javah and move it to the csrc folder

```
$ cd build/classes
$ javah org.trifort.jni_example.JNIExample
$ mv org_trifort_jni_example_JNIExample.h ../../csrc/
```

Figure C.2 – Shell commands to create JNI header file with javah

Appendix D

Code listing of Basic Example for RTAClass Loader Evaluation

A.java

```
public class A {
  public static void main(String[] args){
    B b = new B();
    b.foo();
  }
}
```

Figure D.1 – RTAClass Loader Evaluation A.java

B.java

```
public class B {
  public void foo(){
    C c = new C();
    c.abc();
  }

  public void bar(){
    D d = new D();
    d.abc();
  }
}
```

Figure D.2 – RTAClass Loader Evaluation B.java

C.java

```
public class C {
  public void abc(){
    System.out.println("In class C");
  }
}
```

Figure D.3 – RTAClass Loader Evaluation C.java

D.java

```
public class D {
  public void abc(){
    System.out.println("In class D");
  }
}
```

Figure D.4 – RTAClass Loader Evaluation D.java

Bibliography

1. Pratt-Szeliga, P.C. *Rootbeer GPU Compiler Lets Almost Any Java Code Run On the GPU*. 2012; Available from: http://developers.slashdot.org/story/12/08/12/0056228/rootbeer-gpu-compiler-lets-almost-any-java-code-run-on-the-gpu.
2. Pratt-Szeliga, P.C. *Rootbeer GPU Compiler*. 2012; Available from: https://github.com/pcpratts/rootbeer1.
3. Pratt-Szeliga, P.C., J.W. Fawcett, and R.D. Welch, *Rootbeer: Seamlessly Using GPUs from Java*, in *NVIDIA GTC*. 2013.
4. Pratt-Szeliga, P.C., et al., *Soot Class Loading in the Rootbeer GPU Compiler*, in *ACM PLDI SOAP*. 2013.
5. OpenJDK Community. *Project Sumatra*. 2012; Available from: http://openjdk.java.net/projects/sumatra/.
6. Defense Advanced Research Projects Agency, *Data-Parallel Analytics on Graphics Processing Units (GPUs)*. 2013.
7. Lam, P., E. Bodden, and O. Lhoták, *The Soot Framework for Java Program Analysis; A Retrospective*, in *Cetus Users and Compiler Infrastructure Workshop*. 2011.
8. Adereth, M. *Counting Stars on Github*. 2013; Available from: http://adereth.github.io/blog/2013/12/23/counting-stars-on-github/.
9. Doll, B. *10 Million Repositories*. 2013; Available from: https://github.com/blog/1724-10-million-repositories/.
10. Lengyel, J., et al., *Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*, in *ACM SIGGRAPH*. 1990.
11. Wikipedia. *Wolfenstein 3D*. 2015; Available from: http://en.wikipedia.org/wiki/Wolfenstein_3D.
12. Wikipedia. *John D. Carmack*. 2015; Available from: http://en.wikipedia.org/wiki/John_D._Carmack.
13. Game System Requirements. *Wolfenstein 3d System Requirements*. 2015; Available from: http://gamesystemrequirements.com/games.php?id=1728.
14. Wikipedia. *Doom (video Game)*. 2015; Available from: http://en.wikipedia.org/wiki/Doom_(video_game).
15. Wikipedia. *3dfx Interactive*. 2015; Available from: http://en.wikipedia.org/wiki/3dfx_Interactive.
16. Wikipedia. *Quake Engine*. 2015; Available from: http://en.wikipedia.org/wiki/Quake_engine.
17. Hoff, K.E., et al., *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*, in *ACM SIGGRAPH*. 1999.
18. Harris, M.J., et al., *Physically-based Visual Simulation on Graphics Hardware*, in *ACM SIGGRAPH / EUROGRAPHICS*. 2002.
19. Owens, J.D., et al., *A Survey of General-Purpose Computation on Graphics Hardware.* Computer Graphics Forum, 2007.
20. NVIDIA. *CUDA 1.0 Released*. 2007; Available from: https://devtalk.nvidia.com/default/topic/373147/cuda-1-0-released/.
21. The Khronos Group. *OpenCL*. 2009; Available from: https://www.khronos.org/opencl/.

22. Hutter, M. *Java Bindings for CUDA*. 2009; Available from: http://www.jcuda.org/.
23. Hutter, M. *Java Bindings for OpenCL*. 2009; Available from: http://www.jocl.org/.
24. Chafik, O. *JavaCL - OpenCL Bindings for Java*. 2011; Available from: https://code.google.com/p/javacl/.
25. Leung, A.C.-W., *Automatic Parallelization for Graphics Processing Units in JikesRVM*. 2008, University of Waterloo.
26. Calvert, P., *Parallelisation of Java for Graphics Processors*. 2010, University of Cambridge.
27. Frost, G. *Aparapi - API for data parallel Java. Allows suitable code to be executed on GPU via OpenCL*. 2011; Available from: http://code.google.com/p/aparapi/.
28. Deneau, T. *Sumatra Dev Mailing List Message*. 2015; Available from: http://mail.openjdk.java.net/pipermail/sumatra-dev/2015-May/000310.html.
29. Clarkson, J., et al., *Boosting Java Performance using GPGPUs.* arXiv, 2015.
30. Grossman, M., S. Imam, and V. Sarkar, *HJ-OpenCL: Reducing the Gap Between the JVM and Accelerators.* ACM PPPJ '15, 2015.
31. Baxter, S. *Modern GPU*. 2013; Available from: http://nvlabs.github.io/moderngpu/.
32. Sengupta, S., et al., *Efficient Parallel Scan Algorithms for Many-core GPUs*, in *Scientific Computing with Multicore and Accelerators*. 2010.
33. Harris, M., S. Shubhabrata, and J.D. Owens, *Parallel Prefix Sum (Scan) with CUDA*, in *GPU Gems 3*. 2007, Addison-Wesley: Boston, MA.
34. Merrill, D. and A. Grimshaw *Parallel Scan for Stream Architectures*. University of Virginia Technical Report, 2009.
35. Merrill, D. and A. Grimshaw, *High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing.* Parallel Processing Letters, 2011.
36. Merrill, D., M. Garland, and A. Grimshaw. *Scalable GPU Graph Traversal*. in *ACM SIGPLAN PPoPP*. 2012.
37. Alcantara, D.A., et al., *Building an Efficient Hash Table on the GPU*, in *GPU Computing Gems*, W.-m.W. Hwu, Editor. 2011, Morgan Kaufmann: San Francisco, CA. p. 39-53.
38. Maas, M., et al., *GPUs as an Opportunity for Offloading Garbage Collection*, in *ACM ISMM*. 2012.
39. Wikipedia. *Java (programming Language)*. 2015; Available from: http://en.wikipedia.org/wiki/Java_(programming_language).
40. Welton, D. *Programming Language Popularity*. 2013; Available from: http://www.langpop.com/.
41. Meloan, S. *The Java HotSpot performance engine: An in-depth look*. 1999.
42. Oracle, *JNI Types and Data Structures*, in *Java Native Interface Specification*. 2002.
43. Vallée-Rai, R., et al., *Soot - a Java Bytecode Optimization Framework*, in *IBM CASCON*. 1999.
44. Brosius, D., et al. *Apache Commons BCEL*. Available from: http://commons.apache.org/proper/commons-bcel/index.html.
45. Bruneton, E., R. Lenglet, and T. Coupaye, *ASM: A Code Manipulation Tool to Implement Adaptable Systems*, in *Adaptable and Extensible Component Systems*. 2002.
46. Dolby, J. and S.J. Fink. *WALA Wiki*. 2006; Available from: http://wala.sourceforge.net/wiki/index.php/Main_Page.

47. Naik, M. and A. Aiken. *JChord - A Static and Dynamic Program Analysis Playform for Java*. 2008; Available from: https://code.google.com/p/jchord/.
48. NVIDIA. *Intel Xeon Phi: Just the Facts*. 2013; Available from: http://www.nvidia.com/object/justthefacts.html.
49. Downey, A., *The Little Book of Semaphores*. 2008.
50. Bacon, D.F., *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*, in *Dissertation*. 1997, University of Berkeley: Berkely, California.
51. Bacon, D.F. and P.F. Sweeney, *Fast Static Analysis of C++ Virtual Function Calls*, in *OOPSLA '96*. 1996, ACM SIGPLAN. p. 324-341.
52. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-oriented Software*. 1995, Reading, MA: Addison-Wesley.
53. Vorontsov, M. *String.intern in Java 6, and 8 - String Pooling*. 2013; Available from: http://java-performance.info/string-intern-in-java-6-7-8/.
54. Cormen, T.H., et al., *Section 22.4: Topolocial sort*, in *Introduction to Algorithms*. 2001, MIT Press and McGraw-Hill.
55. Shiv, K., et al., *SPECjwm2008 Performance Characterization.* Springer, 2009.
56. Wikipedia. *C Sharp (programming Language)*. 2015; Available from: http://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29.
57. Yates, R. *Worlds Apart - C# and Java*. 2007; Available from: http://robubu.com/?p=28.
58. Rabiner, L. and B.-H. Juang, *Fundementals of Speech Recognition*. 1993: Prentice Hall.
59. François, J.-M. *Jahmm - An Implementation of Hidden Markov Models in Java*. 2009; Available from: https://code.google.com/p/jahmm/.
60. Xiao, S. and W.-c. Feng, *Inter-block GPU communication via fast barrier synchronization*, in *IEEE IPDPS*. 2010.
61. Behnke, L. *Hierarchical Clustering Java*. 2012; Available from: https://github.com/lbehnke/hierarchical-clustering-java.
62. Letunic, I. and P. Bork, *Interactive Tree of Life V2: Online Annotation and Display of Phylogenetic Trees Made Easy.* Nucleic Acids Research, 2011. **39**.

Biographical Data

Phil Pratt-Szeliga
Computer Information Science and Engineering

Phil Pratt-Szeliga earned his Bachelor of Science degree in Computer and Systems Engineering from Rensselaer Polytechnic Institute in 2005. He received his Master of Science degree in Computer Engineering in 2010 from Syracuse University. While earning his Master's degree he also joined the doctoral program in Computer and Information Science and Engineering at Syracuse University in 2008.

Phil Pratt-Szeliga has been honored at Syracuse University for his Master's degree performance with the All University Master of Science Prize.

While pursuing his degree, he worked as a Research Assistant in the Syracuse University Biology Department and also as an Instructor and Teaching Assistant in the Electrical Engineering and Computer Science Department.

Phil Pratt-Szeliga has presented his computer science research at international conference meetings and workshops including IEEE/HPCC, ACM/PLDI/SOAP and NVIDIA/GTC. His computation biology work has been published in Nature's Journal Scientific Reports which is ranked 5th among all multidisciplinary science primary research journals.

Pratt-Szeliga's open source compiler Rootbeer is a world leader in enabling Java on a Graphics Processing Units and has over 960 stars in github. Rootbeer's github page has incoming links from the OpenJDK and Aparapi projects and Rootbeer has been named state-of-the-art in a recent DARPA STTR solicitation.

Phil Pratt-Szeliga's dissertation, The Rootbeer GPU Compiler, was supervised by Dr. James W. Fawcett