Trinity University
Digital Commons @ Trinity

Computer Science Honors Theses | Computer Science Department

4-20-2012

The Design and Implementation of a Bytecode for Optimization on Heterogeneous Systems

Kerry A. Seitz
Trinity University, kerry.a.seitz@gmail.com

Follow this and additional works at: http://digitalcommons.trinity.edu/compsci_honors

Part of the Computer Sciences Commons

Recommended Citation
Seitz, Kerry A., "The Design and Implementation of a Bytecode for Optimization on Heterogeneous Systems" (2012). *Computer Science Honors Theses*. 28.
http://digitalcommons.trinity.edu/compsci_honors/28

This Thesis open access is brought to you for free and open access by the Computer Science Department at Digital Commons @ Trinity. It has been accepted for inclusion in Computer Science Honors Theses by an authorized administrator of Digital Commons @ Trinity. For more information, please contact jcostanz@trinity.edu.

The Design and Implementation of a Bytecode for Optimization on Heterogeneous Systems

Kerry Allen Seitz, Jr.

Abstract

As hardware architectures shift towards more heterogeneous platforms with different varieties of multi- and many-core processors and graphics processing units (GPUs) by various manufacturers, programmers need a way to write simple and highly optimized code without worrying about the specifics of the underlying hardware. To meet this need, I have designed a virtual machine and bytecode around the goal of optimized execution on highly variable, heterogeneous hardware, instead of having goals such as small bytecodes as was the objective of the Java® Virtual Machine. The approach used here is to combine elements of the Dalvik® virtual machine with concepts from the OpenCL® heterogeneous computing platform, along with an annotation system so that the results of complex compile time analysis can be available to the Just-In-Time compiler. The annotation format is flexible so that the set of annotations can be expanded as the field of heterogeneous computing continues to grow. An initial implementation of this virtual machine was written in the Scala programming language and makes use of the Java bindings for OpenCL to execute code segments on a GPU. The implementation consists of an assembler that converts an assembly version of the bytecode into its binary representation and an interpreter that runs programs from the assembled binary. Because the bytecode contains valuable optimization information, decisions can be made at runtime to choose how best to execute code segments. To demonstrate this concept, the interpreter uses this information to produce OpenCL kernel code for specified bytecode blocks and then builds and executes these kernels to improve

performance. This hybrid interpreter/Just-In-Time compiler serves as an initial implementation of a virtual machine that provides optimized code tailored to the available hardware on which the application is running.

Acknowledgments

I would like to thank my advisor, Dr. Mark Lewis, for all of his help and support on this thesis. From developing the initial idea, to providing useful and interesting ideas and feedback throughout the process, this thesis would not have been possible without his help.

I would also like to thank Dr. Berna Massingill for serving on my thesis committee, for providing helpful feedback on this work, and for always being eager to help tackle the little technical challenges that often arise.

Thank you to Dr. Maurice Eggen for serving on my thesis committee, for providing helpful feedback, and for teaching my Computer Design course, which proved invaluable when working on this thesis.

I want to thank Dr. Kevin Livingstone for giving me my first research opportunity and Dr. Daniela Raicu, Dr. Jacob Furst, and Anne-Marie Giuca for their work and help in my second research opportunity. Without these experiences, I would not have nearly as strong of an interest in research as I do.

Funding for this project was provided by the Trinity University Mach Fellowship Program. This funding gave me the wonderful opportunity to present this work at a peer-reviewed conference.

Thank you to the Trinity University Computer Science Department for providing me with a great education and assistance with many things throughout my time at Trinity.

I would like to thank my friends both at Trinity and elsewhere for always being supportive and for making my undergraduate career fantastic.

Finally, thank you to my parents and my sister for their continuing love and support, no matter what endeavours I pursue in life.

The Design and Implementation of a Bytecode for Optimization on Heterogeneous Systems

Kerry Allen Seitz, Jr.

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation
with departmental honors.

April 20, 2012

______________________________ ______________________________

Thesis Advisor Department Chair

____

Associate Vice President
for
Academic Affairs

Student Copyright Declaration: the author has selected the following copyright provision:

[X] This thesis is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License, which allows some noncommercial copying and distribution of the thesis, given proper attribution. To view a copy of this license, visit http://creativecommons.org/licenses/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

[ ] This thesis is protected under the provisions of U.S. Code Title 17. Any copying of this work other than fair use (17 USC 107) is prohibited without the copyright holders permission.

[ ] Other:

Distribution options for digital thesis:

[X] Open Access (full-text discoverable via search engines)

[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via digitalcommons.trinity.edu)

The Design and Implementation of a Bytecode for Optimization on Heterogeneous Systems

Kerry Allen Seitz, Jr.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Design Plan 2

1.3 Implementation Plan 3

2 Related Work **5**

2.1 Background 5

2.1.1 Annotations 5

2.1.2 Concurrency 6

2.1.3 Heterogeneous 8

2.2 Dalvik 9

2.3 OpenCL 10

3 Bytecode Design **12**

3.1 Instructions 12

3.1.1 General Format 12

3.1.2 Variables and Types 13

3.1.3 Function Calls 15

3.1.4 Other Considerations 16
3.2 Annotations 17

4 Instruction Set **19**
4.1 Variable and Object Instructions 20
4.2 Mathematical Instructions 20
4.3 Array Instructions 21
4.4 Control Structure Instructions 21
4.5 Annotation Instruction 21
4.6 Binary Encoding 22

5 First-Cut Annotations **28**

6 Implementation **31**
6.1 The Class File Format 32
6.2 Scala Code 34
6.2.1 Assembler 34
6.2.2 Class and Object Representation 34
6.2.3 Interpreter 35
6.3 OpenCL 36
6.3.1 The Annotation File 37
6.3.2 Executing Annotations 38

7 Conclusion and Future Work **40**

A Implementation Code **47**
A.1 InstructionList 47

A.2 Assembler 59
A.3 MyClass 60
A.4 MyObject 62
A.5 Interpreter 62
A.6 Annotations 66

B Example Class Files **69**

B.1 CPU Only Execution 69
B.1.1 Class 1 69
B.1.2 Class 2 70
B.2 GPU Execution 70

List of Figures

3.1 Basic Bytecode Instruction Format 13

4.1 Variable and Object Instructions 23
4.2 Mathematic Instructions 24
4.3 Array Instructions 25
4.4 Control Structure Instructions 26
4.5 Annotation Instruction 27
4.6 Binary Encodings for Instructions 27

6.1 Example Class File Header 32
6.2 Example Annotation File 38
6.3 Example OpenCL Kernel 39

Chapter 1

Introduction

1.1 Motivation

Computing hardware is in the middle of a dramatic shift from the single-core, generally homogeneous model that dominated prior to roughly 2005 to a many-core, generally heterogeneous future. This shift began as a simple move from single-core to multi-core processors. More recently, many-core processors in the form of Graphics Processing Units (GPU) from NVIDIA® and AMD®, as well as Many Integrated Core (MIC) Architectures from Intel®, have moved more into the mainstream. This move is probably best illustrated by the line of Fusion® chips from AMD that include GPU circuitry on the same piece of silicon as the CPU. The ubiquitous nature of this development is illustrated by the fact that the first Fusion chips to be released were intended for laptops, not workstations or servers.

In many ways, hardware has gotten ahead of software. Most developers are still used to working in a single-threaded, homogeneous environment. Not only do we have the legacy of developers being trained for that world, but also most of the languages and tools that are in use were designed for that environment. Tools and languages are improving for multi-core

environments, but for the most part, this type of development is harder and more costly than single-threaded development.

If development is not yet prepared for homogeneous, multi-core environments, it is certainly far from ready to tackle the challenges of many-core, heterogeneous environments. There are standards for doing this type of programming such as OpenCL® and CUDA®. However, these standards force the programmer to interact with the machine at a low level, which tends to increase development cost and require much more effort from the programmers. To make these systems more accessible, we need higher-level tools that allow the developer to spend his/her time focusing on the logic of the application instead of the details of the platform on which it will be running. This need is likely to become increasingly important as the options for platforms grow. If making a portable program requires separate development for the different options from Intel (CPU, CPU/GPU, MIC), AMD (CPU, Fusion, GPU), NVIDIA (GPU, Tegra), and the host of other combinations that are likely to appear in the near future, then producing optimized code that can cover the various options will be prohibitively expensive.[1]

1.2 Design Plan

The productivity of developers on single-threaded, homogeneous environments was greatly assisted by the development of platforms that support higher-level programming in a manner that is generally machine independent. These types of platforms include the Java® Virtual Machine (JVM) and the Common Language Runtime (.NET®) environment in which bytecode is produced for a virtual machine and then optimized by a Just-In-Time (JIT) compiler that can produce code that is suited to the machine on which the program is

[1] It is worth noting that even today there are a combinatorial number of options as a machine might have an Intel Sandy Bridge CPU/GPU combo with a graphics card from either AMD or NVIDIA.

actually running. Unfortunately, these environments were made with design goals that did not include optimizability on a JIT or support for many-core, heterogeneous environments. Indeed, in the case of the JVM, the design goals were small bytecodes and fast emulation. As a result, it uses a stack based architecture which causes some problems for producing optimal performance even on single-threaded hardware [11, 23].

With all of these things in mind, I set out to develop a bytecode that has a primary design goal of being able to support optimization for performance on heterogeneous platforms. My approach was to borrow ideas from one of the newest register-based virtual machine platforms, Dalvik®, and combine those with aspects of the OpenCL standard for heterogeneous computing. In order to provide possible optimization suggestions to the virtual machine, the bytecode contains annotations. These annotations may pertain to individual instructions, blocks of code, or variables. Because programming environments are rapidly changing and predicting the next major advances is challenging at best, the annotation format is very flexible so that additional annotations can be implemented in the future and can be formatted to best represent the appropriate information.

1.3 Implementation Plan

As an initial implementation of a virtual machine to execute the bytecode, I wrote a hybrid interpreter/JIT compiler using the Scala programming language. I chose to use a high-level language for the interpreter/JIT to make the implementation work less time consuming in order to focus on bytecode design issues (especially those that arose during implementation) rather than tedious coding. More importantly, however, I wanted the implementation to be able to run code on a GPU, as the main design goal of this project is to create a better platform for heterogeneous computing. By using Scala, I could easily use the Java

bindings for OpenCL to execute blocks of code on a GPU (for information about OpenCL, see [16]). Also, because Scala compiles to bytecode for the JVM, the interpreter is platform independent. As long as the appropriate Java Runtime Environment (JRE) and OpenCL environments are installed, then this implementation also meets the design goal of being platform independent.

Chapter 2

Related Work

2.1 Background

Arguably the biggest goal of heterogeneous computing is increasing code efficiency, a topic explored in great depth. Almost all modern compilers perform optimizations to increase code performance, but these optimizations sometimes incur a performance cost. Dynamic compilation and optimization that occurs in a virtual machine (VM) must execute quickly; otherwise, the time spent optimizing might negate the speed-ups provided by the optimizations. Furthermore, users expect quick and responsive program execution, so a VM cannot perform time consuming optimizations and compilations at start-up.

2.1.1 Annotations

In order to aid the VM's optimization efforts, Krintz and Calder [9] developed an annotation framework for the JVM to reduce compilation overhead. By reducing this overhead, complex optimizations can more efficiently be performed in the VM, allowing for greater runtime performance. This framework integrates annotations into Java class files using the *attribute*

structure defined in the Java language specification. Because one of Java's design goals is small bytecode size, Krintz and Calder, when designing their framework, decided to minimize the increase in file size caused by adding annotations. However, because small bytecode size is not one of my goals in this design, the annotations here can use as much size as necessary to provide valuable runtime information.[2] Vallée-Rai et al. [24] also investigated an annotation framework for the JVM. They found that because class files can often be optimized statically, adding annotations to these files may reduce the number of optimizations that the JIT compiler must perform, thereby allowing the compiler to focus on more expensive optimizations. Furthermore, some runtime checks, such as certain array index bounds checks, can be performed at compile time. If these checks are performed at compile time, annotations can be used to forgo the checks at runtime.

2.1.2 Concurrency

The idea of providing more high-level concurrency support in VMs is not a novel concept. Marr et al. [12], recognizing the transition to many-core processors, argued that VMs need to abstract concurrency to take advantage of this shift. In addition, they surveyed 17 VMs to determine the current support for concurrency models and found only limited explicit concurrency support. While some current VMs support models of concurrency on multi-core processors, few provide abstractions for running code on GPUs. General purpose GPU (GPGPU) programming has come a long way in recent years. Many of the first GPGPU applications had to use graphics libraries such as OpenGL and Direct3D to access the capabilities of the GPU. Because these libraries were not designed for GPGPU programming, they limit the programmer's ability to access parts of the hardware that can

[2] As memory continues to increase in capacity while decreasing in price, the size of a compiled program will become less significant, but performance will remain an issue, especially in high performance computing applications.

accelerate performance for general purpose applications. Furthermore, using these libraries for GPGPU programming is, at best, a work-around that puts additional burdens on the programmer. Realizing the potential of GPGPU programming, Peercy et al. [20] developed a VM for GPUs focused on performance. The Data Parallel Virtual Machine (DPVM) that they developed provides a simplified method to write code for GPUs yet still reveals low-level functionality for when it is needed. Many applications that would benefit from acceleration using a GPU are indeed highly data parallelizable. However, having the option to easily program other types of parallel applications for execution on a GPU would be preferable, and the DPVM is limited in this regard.

Another example of a VM that utilizes the GPU is GViM (GPU-accelerated Virtual Machine). Because high performance computing systems are now taking advantage of the many-core GPUs, Gupta et al. [7] sought to virtualize this hardware to reduce the complexity when writing code for it. Their result was GViM, "a system designed for virtualizing and managing the resources of a general purpose system accelerated by graphics processors." Although GViM incurs a performance cost over non-virtualized solutions, the flexibility and reduction in programmer effort resulting from this VM model helps to justify the virtualization of hardware other than simple CPUs. Indeed, virtualized execution will likely always perform worse than code written and compiled for specific devices. If a programmer knows the intricacies of the hardware for which he/she is writing code, then he/she can write extremely efficient code. However, I suspect that very few programmers know these minute details, and likely even fewer are willing to spend the time and effort to write fine-tuned code for perfectly efficient execution. Doing so would substantially increase development costs, and for most applications, the effort is not worth the result. Moreover, even when the most efficient code is written, this code is not portable to other devices. By compiling source code to a bytecode and allowing a VM to optimize the code at runtime, we simul-

taneously get portability and efficiency, while allowing programmers to write in high-level languages without concern for low-level concepts.

2.1.3 Heterogeneous

While the examples provided above allow easier access to the GPU, they do not utilize code execution on both the CPU and the GPU. Ideally, we would like to take advantage of all of the hardware within a computer, which is a driving force in heterogeneous computing. The MapCG framework [8] provides a means to write code for both the GPU and the CPU using a high-level programming model. This framework allows programmers to write code in the high-level MapReduce framework [5]. The MapCG runtime then compiles the source code for execution on the CPU and the GPU. Along with providing access to both the CPU and the GPU in a high-level environment, this framework also allows for code portability to other CPU and GPU architectures.

In the recent past, most computing occurred on the CPU. Now, more attention is being paid to GPGPU computing as a way to increase program efficiency. New technology is constantly evolving, and predicting the next big advance in hardware development is challenging at best, if not impossible. Therefore, we need systems that can adapt to new technology, while maintaining backwards compatibility with older code. Devices such as field-programmable gate arrays (FPGA) provide a promising avenue for advancement in hardware efficiency [10], and, ideally, we would like to have old code utilize this hardware as effectively as possible without requiring rewrites or recompiles from source. In a world where computers are becoming more heterogeneous, we need a way to provide programmers with a programming model that is independent of the hardware, because each piece of hardware has its own underlying instruction set architecture. The key to this model is virtualization [1]. Therefore, I have developed a bytecode and VM for use in a heterogeneous environment

that preserves optimization information in annotations so that, at runtime, the code can be optimized and executed efficiently on whichever devices are available and will provide the best performance. Because technology is ever-changing, I present a flexible annotation format to allow additional annotations to be implemented as new technology and techniques arise.

2.2 Dalvik

The Dalvik Virtual Machine is an integral part of Google's Android® operating system. This VM runs all of the apps written for this platform. Code for the Dalvik VM is typically written in Java, but the Dalvik bytecode differs significantly from the Java bytecode. First and foremost, the Dalvik VM is register-based [2], rather than stack-based like the JVM [11]. A register-based VM and bytecode offer a few advantages over stack-based architectures. Because modern CPUs are register-based, this type of bytecode more closely mimics the hardware on which it is running. Therefore, if the number of registers implemented in the VM matches (or is close to) the number of registers in the actual hardware, then register allocation can be performed during the original compilation rather than in the JIT compilation stage. Even though register allocation can be quite efficient (as it must be for the runtime allocation that occurs in the JVM), performing allocation at compile time eliminates one more thing the VM must do, thus freeing time for more complex runtime optimization. Furthermore, Shi et al. [23] found that a register-based VM is faster than a stack-based VM when implemented as an interpreter. Because many JIT compilers, such as Oracle's Hotspot VM [13], initially use an interpreter to decrease start-up time, Shi's findings suggest that a register-based bytecode will improve performance in modern VMs.

Of course, a register-based bytecode also has some disadvantages. Because it does not

make any assumptions about the number of registers, a stack-based bytecode is simpler and possibly better suited for JIT compilation [23]. To mitigate these disadvantages, the bytecode presented in this work is neither register- nor stack-based. Instead, it is more high-level in that it preserves the concept of variables. The bytecode is based on the Dalvik bytecode, but instead of referencing registers, it references variables. Type information is also preserved for these variables. Use of type information in low-level languages has been explored in other contexts before (for example, see [14] and [15]).

2.3 OpenCL

The biggest motivation behind developing this new bytecode and virtual machine model is the increasing emphasis on heterogeneous computing. The OpenCL standard provides a way to utilize any piece of hardware in the computer, as long as the appropriate drivers are installed. A program written in OpenCL can query the machine to determine which devices are available and then select the device or devices best suited for running each section of code. Because this functionality is consistent with my goal of exploiting a heterogeneous environment, I chose to use OpenCL to implement GPU code execution in this virtual machine.

While the bytecode does not contain any instructions specifically for the OpenCL implementation, the OpenCL computation model influenced some of the information preserved in the bytecode from the original source code. This additional information is stored in annotations, and provides optimization hints to the JIT compiler. An OpenCL program consists of two distinct sections: the host and the kernels. The host serves only to set up kernels and queue them for execution, while the kernels are where the real computation happens. Part of the host setup includes querying for and deciding which devices the program will

use [16]. This execution model works well with annotations that specify on what type of device a particular block of code will execute most efficiently. For example, a code section that can be parallelized over a large set of data is well suited for execution on a GPU, so an annotation can accompany this bytecode block indicating as such. Furthermore, this annotation might aid the virtual machine in determining how to convert the bytecode into kernels when using the OpenCL implementation.

Another annotation that OpenCL helped to motivate is an annotation that states the variables used in a certain section of code. Because the OpenCL memory model requires that data be copied to each device where it is needed, this annotation allows the VM to optimize memory moves to different devices. Moreover, because memory move operations are computationally expensive, annotations such as whether an object is immutable or functionally immutable will tell the VM whether certain memory moves are actually necessary (for example, copying an immutable object back to the host is unnecessary and wastes computation time). A functionally immutable object is technically a mutable object (i.e., some state in that object changes), but the state change is not visible outside of that object. Therefore, a functionally immutable object can be considered immutable for certain optimization purposes (such as memory moves). Furthermore, even if an object is neither immutable nor functionally immutable, in certain programs the object may not actually mutate. If the mutable object does not mutate during program execution, then this object can be treated as immutable. This type of analysis can be performed at compile time in many situations, so an annotation can accompany that object stating its pseudo-immutability.

Chapter 3

Bytecode Design

3.1 Instructions

The bytecode contains two distinct types of information: the instructions and the annotations. The instructions are the actual operations that must be performed when executing the code. These instructions are very low-level, much like instructions in assembly and machine code. When compiled bytecode is generated, it is encoded in a binary format.

3.1.1 General Format

Each instruction is 128 bits long. A fixed-length instruction format was chosen so that the VM could make use of the many advantages of this format over variable-length instructions. By knowing exactly how long each instruction is, the VM can pipeline instructions by beginning to decode future instructions before the current one is finished executing. The VM can also more easily rearrange instructions to better suit the devices if it knows the length of each instruction from the start. Furthermore, when making use of annotations that apply to blocks of code, the VM needs to know how long the block is in order to

perform the appropriate optimization action. Variable-length instructions can reduce the size of the compiled bytecode by eliminating wasted space that comes from the unused bits of some instructions in the fixed-length design, but because bytecode file size is not a major concern of this bytecode and VM, the advantages of fixed-length instructions far outweigh those of variable-length instructions.

For each instruction, the first 16 bits denote which instruction to perform, and the last 16 bits are reserved for annotations associated with the instruction. The other bits are used to store references to variables, type information, or literal values, as determined by the individual instructions, using 16 bits for each variable or type and groups of 16 bits for literals. Figure 3.1 shows a pictoral representation of the instruction format.

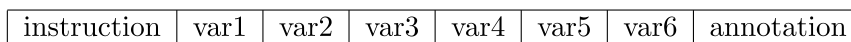

Figure 3.1: The basic format of the instructions in this bytecode. Each box represents 16 bits. Some fields may be unused in certain instructions. The var slots may also be replaced by types or literals, depending on the instruction. Slots can be combined to accommodate literals longer than 16 bits. See Chapter 4 for a list of the instructions in this bytecode.

3.1.2 Variables and Types

Instructions refer to variables by using a 16 bit integer reference. Each stack frame contains a new set of variables. Each variable also has a type associated with it, which is specified using a 16 bit reference when the variable is initialized. For this reason, all variables must be initialized explicitly in the bytecode in order to specify the type and to provide annotations associated with it. By using 16 bits to reference variables and types, a program can contain up to 65,536 (2^{16}) variables per stack frame and up to 65,536 types, which should be many more than any program should ever need. The first eight types are reserved for the primitives that are built into most modern hardware (boolean, char, byte, short, integer, long, float,

and double), and the ninth type is reserved to denote arrays. The rest of the types are defined by user and language/library defined types. The use of a 16 bit field to represent a type has already been established in Dalvik and the JVM [2, 11]; however, associating annotations with variables and types has only been explored in a limited context.

Along with type information for individual variables, the types used in generics will also be preserved in the bytecode. This information will be located in annotations to variables that take a generic parameter. Thus, unlike the JVM, this bytecode will not be limited by type erasure. Because object types can contain annotations, knowing as much type information as possible is beneficial for optimization; therefore, preventing type erasure both aids the programmer's efforts and allows for greater optimization potential.

Like Dalvik's Dex files and the JVM class files, this bytecode also has type files which contain methods and variables associated with objects. The headers for these type files are similar in form to the headers used in Dalvik and the JVM [2, 11]. However, the headers also contain annotation information that pertains to the type as a whole. In addition, annotations will accompany the fields and methods to provide specific details for each. For example, an object may or may not be immutable as a whole, so this information is stored in the header. In mutable objects, methods that do not change the object can be classified as non-mutating so that if only these methods are used within a program, the object can be considered immutable and take advantage of optimizations associated with immutable objects. By storing annotations both in the header and with individual methods and fields, the bytecode provides more information to the JIT compiler so that it can make intelligent optimization decisions based on the hardware that it has available.

3.1.3 Function Calls

When a function call instruction executes, a few key steps occur. First, the stack frame shifts so that the function will have a new set of variables. The first variable contains a pointer to the object on which the function was invoked (i.e., a "this" pointer). If the function is static, then the data in the first variable slot does not matter as the VM will not use it. The next n variables contain the arguments to the function, where n is the number of arguments to that function and $n \leq 4$. If a function requires more than four variables, the first three are passed using variable indices, and the remaining variables are stored in memory through instructions preceding the function call. The address to this memory location is then stored in a variable and passed to the function in the fourth argument slot. The function is responsible for extracting the variables from memory as needed, using bytecode instructions inserted by the compiler. The fourth argument slot also serves as the location to which the function should return a variable. Thus, the data stored in the variable referenced by this slot will be lost when the function returns unless the data is stored elsewhere. Unfortunately, this design decision requires extra variable copies in many situations (e.g., whenever four arguments are required and all of them must remain unchanged after the function executes); however, the process of storing variables in memory and referencing their location when the number of function arguments is greater than four is computationally expensive. If a dedicated return variable slot were built into this instruction (rather than the dual-use fourth argument/return variable slot), then the function call instruction would only have three variable slots, meaning that any method with four or more arguments would need to perform the memory storage operations. Therefore, I decided that having more arguments passed directly in the instruction is preferable and, thus, that using the fourth slot as both an argument slot and the return slot is the best

decision for this bytecode.

3.1.4 Other Considerations

Much attention was given to deciding how long to make each instruction. Much like the rationale for using 16 bits to represent variables and types, using 16 bits to identify instructions and annotations provides many more combinations than should be necessary, without making instructions unreasonably large. After the instruction field and the annotation field, 96 bits remain for variable references, type references, and literals. Considering the variable initialization instruction, the biggest primitive types (the long and the double) are 64 bits. Then, 32 bits remain in the instruction after the instruction field, annotation field, and 64 bit literal field. The two sets of 16 bits within the 32 remaining bits, then, are perfect for referencing a variable and a type. Thus, when using 16 bits each to reference instructions, annotations, variables, and types, 128 bits is the minimum instruction size needed to contain the variable initialization instructions for the long and the double. The 128 bit instruction length also works well for function calls by providing six 16 bit fields to reference the object on which the method is called, to encode the method number of the method to execute, and to hold four arguments to the method (providing four slots for arguments before resorting to storing additional arguments in memory seemed sufficient).

As the instruction format is rather long, many instructions will not use the full 128 bits available and will instead have some slots with unused data. For some instructions, the bytecode could be modified to contain variations that perform multiple operations on different sets of data. For example, an add instruction could take two sets of three variables (one destination variable and two source variables in each set), denoting two distinct add operations that can be executed in parallel. This possibility echoes the ideas of the Very Long Instruction Word (VLIW) architectures [6]. For simplicity, I chose not to include

these types of instruction in the current bytecode design. However, I felt that mentioning this aspect of flexibility in the bytecode is important in case new advantages of the VLIW architecture or something similar are discovered. Recent research in auto-vectorization may also help to motivate the addition of these types of instructions (see, for example, [17, 18, 21]). By performing compile time auto-vectorization and having special bytecode instructions for the results of this analysis, the virtual machine could make immediate use of Single Instruction Multiple Data (SIMD) instructions at runtime to improve performance.

3.2 Annotations

The annotations in this bytecode come in two forms. The first form is a set of annotations that are expressed in individual instructions, represented by a 16 bit integer in the last 16 bits of the instruction. The meaning of the annotation depends on the instruction with which it is associated (e.g., the same configuration of 16 bits may mean one thing when used with a variable instantiation instruction but have an entirely different meaning when used with an add instruction). For some instructions, each combination of 16 bits will represent a unique annotation. For others, however, the 16 bits will serve as bit flags so that multiple pieces of information can be associated with a single instruction. For example, knowing whether a variable is constant, escapes the current thread, and/or escapes the current stack frame can provide important optimization information, so the 16-bit annotation for the variable declaration instruction is implemented as a set of bit flags. Thus, a variable can be flagged with all three of these annotations, and more, within the declaration instruction itself.

The other annotation form in this bytecode is the annotation file. This file contains all of the annotations associated with the code other than the annotations specified in the last

field of the instructions themselves. A special annotation instruction is used to reference these annotations, and these instructions are inserted by the compiler at the appropriate locations to specify information for potential optimizations, such as before a block of code that is well suited for execution on a GPU. The annotation instruction uses the last 16 bits to describe the annotation type. The 64 bits after the instruction field encode an offset in the annotation file for the full annotation data. The location in the annotation file must contain the length of the annotation followed by the specific information for that annotation. The format of the information is defined by the annotation type that was in the bytecode instruction. Each type of annotation can have an entirely different format. This flexible annotation format allows for new annotations to be easily added in the future. The creators of those annotations can decide how best to arrange the information associated with them. With this flexibility comes the limitation that the annotation creator must also modify the VM to include information about how to process this new annotation. If a VM identifies an annotation that it is not aware of, it will simply ignore the annotation. Thus, programs compiled with new annotations are backwards compatible with older VMs, but some potentially useful optimization information will go unused.

Chapter 4

Instruction Set

This chapter contains a list of the instructions included in this bytecode. It is divided into sections based on the type of instruction, and descriptions are also included in these sections. In the figures, empty boxes indicate unused fields. Unless otherwise noted, all fields in the following figures are 16 bits. Asterisks are used to denote instructions that are not implemented in the current version of the hybrid interpreter/JIT compiler (discussed in Chapter 6). The current implementation works only with integer instructions.

Each instruction contains an annotation segment, which uses a full 16 bits to provide for future additions and to make sure that there are not constraints on future growth; program analysis is an area of significant current work, and new languages might have constructs that provide particularly useful information for optimization. In some cases, there are clear and obvious uses for some of the bits. Variable declaration and object creation are two of these cases. For variable declaration, a bit is reserved for specifying if the variable is constant after declaration. For object creation, two bits are reserved for the results of escape analysis to see if the object can escape the current stack frame or the current thread. This type of analysis can allow for stack allocation of objects or the removal of synchronization code [4].

4.1 Variable and Object Instructions

The first group of instructions is shown in Figure 4.1 and deals with variables and object creation, as well as type casting. As with many instructions, after the 16 bit opcode, the next 16 bit field stores a destination variable. The third 16 bit field encodes a type, except in the assignment instructions, where the type will be known from the initialization of the destination variable. The variable creation instructions then provide an initial value, either from a source variable or from a literal that can be up to 64 bits. The type casting instruction specifies a source variable and a destination variable for the cast, storing the value of the source variable in the destination variable of the new type rather than changing the type of the source variable.

The `init-var-lit`, `set-var-lit`, and `set-field-lit` instructions use 64 bits to encode the literal value. When variables that do not use 64 bits (int, short, etc.) are initialized and set with these instructions, only the n rightmost bits are used, where n is the number of bits used in that type. For example, when initializing an int with a literal value, only the rightmost 32 bits are extracted from the 64 bit literal and used to set the int with a value.

4.2 Mathematical Instructions

Figure 4.2 contains the instructions for numeric unary and binary operations. The binary operations come in forms where both operands are variables as well as forms where one operand is a literal. In the instruction name, *type* is used to indicate an appropriate primitive type (e.g., int, double, etc.).

4.3 Array Instructions

Figure 4.3 shows the different array instructions. While arrays can be modeled in languages as data types that sit at the library level [19], the bytecode needs to have a way to set aside and access larger chunks of continuous memory. The virtual machine implementation is given freedom in how it stores objects in an array. The Java model of storing references provides simplicity for languages that make extensive use of subtyping, but it has performance costs in the form of additional memory loads and potentially poor interactions with cache. For that reason, if the type is final and references to elements of the array do not escape local usage to outlive the array itself, the VM should have the freedom to place objects directly into the array block. This option will be particularly useful for multi-dimensional arrays and is a significant part of the reason the X10 language includes the concept of a Rail [3]. Through program analysis, this same type of optimization can be applied at the VM level in languages that do not explicitly provide constructs for it.

4.4 Control Structure Instructions

The last major group of instructions is shown in Figure 4.4, which contains the flow control instructions. These instructions include required and conditional branches, as well as method invocations. The branch to a specific instruction will be the most commonly used form, but branching to variable locations is also allowed to provide greater flexibility.

4.5 Annotation Instruction

The annotation instruction, shown in Figure 4.5, is used to specify all annotations that are not associated with any single instruction. For example, if an annotation applies to a group

of instructions (e.g., denoting that a code block is well suited for a GPU), then it should use the annotation instruction to identify the annotation. The annotation instruction contains a literal value that indicates an offset within the annotation file where the information for this annotation is located. This instruction also contains an annotation segment in the last 16 bits that is used to specify the type of the annotation. Because the format of the annotation file is not strictly defined, the VM must know how to process each annotation type in order to make use of the information within. The annotation type is provided so that the VM can either determine how to interpret the data in the annotation file or ignore the annotation if the type is not recognized. This design allows for backwards compatibility so that code compiled with new annotations can run on older VMs.

4.6 Binary Encoding

The instruction names listed in the previous sections are convenient when describing the bytecode and understanding the function of each instruction. However, the actual bytecode contains only binary values, not string representations of instruction names. Thus, each instruction in the bytecode has a corresponding binary representation. Figure 4.6 lists the binary encodings (as hexadecimal values) for each of the instructions implemented in the current version of the VM. The current implementation only works with integers, but the binary mappings are designed such that they can easily be extended to other types. For example, the binary representation for `add-int` is `0x1402`. In the VM, integers are denoted as type 4, so the second digit in the hexadecimal form of the binary encoding denotes the type of the instruction. Thus, to determine the binary encodings of the add instructions for the other primitives, simply change the second digit to the value that corresponds to that type.

Instantiates variable *destVar* with the value from *sourceVar*

init-var	*destVar*	*type*	*sourceVar*				annotation

Instantiates variable *destVar* with the value of *literal*

<table><tr><td>init-var-lit</td><td>destVar</td><td>type</td><td colspan="4">literal (64 bits)</td><td>annotation</td></tr></table>

Assigns variable *destVar* with the value from *sourceVar*

set-var	*destVar*	*sourceVar*					annotation

Assigns variable *destVar* with the value of *literal*

<table><tr><td>set-var-lit</td><td>destVar</td><td colspan="4">literal (64 bits)</td><td></td><td>annotation</td></tr></table>

Sets the field *fieldNum* in object *objVar* to the value in *sourceVar*

set-field	*objVar*	*fieldNum*	*sourceVar*				annotation

Sets the field *fieldNum* in object *objVar* to the value of *literal*

<table><tr><td>set-field-lit</td><td>objVar</td><td>fieldNum</td><td colspan="4">literal (64 bits)</td><td>annotation</td></tr></table>

Gets the value of field *fieldNum* in object *objVar* and stores it in *destVar*

get-field	*objVar*	*fieldNum*	*destVar*				annotation

Creates a new object of type *type*, storing a reference to it in *destVar*

obj-create	*destVar*	*type*					annotation

Typecast of *sourceVar* to *type*, storing the type-casted object in *destVar*

typecast*	*destVar*	*type*	*sourceVar*				annotation

Figure 4.1: The instructions related to variable and object creation, assignment, and the type cast. Empty boxes indicate unused fields. Unless otherwise noted, all fields are 16 bits. Asterisks are used to denote instructions that are not implemented in the current version of the hybrid interpreter/JIT compiler.

Unary ones-complement

not-*type*	*destVar*	*sourceVar*					annotation

Unary twos-complement

neg-*type*	*destVar*	*sourceVar*					annotation

Binary operators on two variables

op-type	*destVar*	*sourceVar1*	*sourceVar2*				annotation
add							
sub							
mul							
div							
rem							
and							
or							
xor							

Binary operation between a variable and a literal. The size of *literal* depends on the *type*

op-type-lit	*destVar*	*sourceVar*	*literal* (16-64 bits)	annotation
add				
sub				
mul				
div				
rem				
and				
or				
xor				

Figure 4.2: The basic math instructions that are part of the bytecode. In the instruction name, *type* can be replaced by any of the primitive types. Empty boxes indicate unused fields. Unless otherwise noted, all fields are 16 bits.

Creates a new array of size *sizeVar* and type *type*, storing a reference to it into *destVar*

new-array	*destVar*	*type*	*sizeVar*				annotation

Creates a new array of size *literal* and type *type*, storing a reference to it into *destVar*

new-array-lit	*destVar*	*type*	*literal* (32 bits)				annotation

Stores the length of the array referenced by *sourceVar* into *destVar*

array-len	*destVar*	*sourceVar*					annotation

Stores *sourceVar* in array *arrayVar* at position *locVar*

array-put	*sourceVar*	*arrayVar*	*locVar*				annotation

Stores value at position *locVar* in array *arrayVar* into *destVar*

array-get	*destVar*	*arrayVar*	*locVar*				annotation

Figure 4.3: The instructions that are intended to work with arrays in the VM. Empty boxes indicate unused fields. Unless otherwise noted, all fields are 16 bits.

Jumps to the bytecode instruction at location *dest*

goto-lit	*dest* (64 bits)						annotation

Jumps to the bytecode instruction at the location stored in *destVar*

goto-var	*destVar*						annotation

Branch to bytecode instruction at location *dest* if *sourceVar1* and *sourceVar2* compare as specified

if-*test*-lit if-eq-lit if-ne-lit if-lt-lit if-le-lit if-gt-lit if-ge-lit	*sourceVar1*	*sourceVar2*	*dest* (64 bits)				annotation

Branch to bytecode instruction stored in *destVar* if *sourceVar1* and *sourceVar2* compare as specified

if-*test*-var if-eq-var if-ne-var if-lt-var if-le-var if-gt-var if-ge-var	*sourceVar1*	*sourceVar2*	*destVar*				annotation

Invokes method *methodNo* on object *objVar* with arguments *argVar1* - *argVar4*

invoke	*objVar*	*methodNo*	*argVar1*	*argVar2*	*argVar3*	*argVar4*	annotation

Invokes static method *methodNo* on class *literal* with arguments *argVar1* - *argVar4*

invoke- static	*literal*	*methodNo*	*argVar1*	*argVar2*	*argVar3*	*argVar4*	annotation

Returns from a function, returning the value of type *type* from variable *sourceVar*

return	*sourceVar*	*type*					annotation

Returns from a function

return- void							annotation

Throws the exception stored in *sourceVar*

throw*	*sourceVar*						annotation

Figure 4.4: The bytecode instructions that support program flow control. Empty boxes indicate unused fields. Unless otherwise noted, all fields are 16 bits. Asterisks are used to denote instructions that are not implemented in the current version of the hybrid interpreter/JIT compiler.

Accesses information for the annotation type *annotation* at offset *literal* in the annotation file

annot	*literal* (64 bits)			annotation

Figure 4.5: The instruction used to reference annotations in the annotation file. Empty boxes indicate unused fields. Unless otherwise noted, all fields are 16 bits.

annot	0x0000	or-int	0x1408	if-lt-var	0x3003
init-var	0x0001	xor-int	0x1409	if-le-var	0x3004
set-var	0x0002	add-int-lit	0x1412	if-gt-var	0x3005
set-field	0x0003	sub-int-lit	0x1413	if-ge-var	0x3006
get-field	0x0004	mul-int-lit	0x1414	invoke	0x3007
obj-create	0x0005	div-int-lit	0x1415	invoke-static	0x3008
typecast	0x0006	rem-int-lit	0x1416	return	0x3009
init-var-lit	0x0011	and-int-lit	0x1417	return-void	0x300A
set-var-lit	0x0012	or-int-lit	0x1418	throw	0x300B
set-field-lit	0x0013	xor-int-lit	0x1419	goto-lit	0x3010
not-int	0x1400	new-array	0x2000	if-eq-lit	0x3011
neg-int	0x1401	new-array-lit	0x2010	if-ne-lit	0x3012
add-int	0x1402	array-len	0x2001	if-lt-lit	0x3013
sub-int	0x1403	array-put	0x2002	if-le-lit	0x3014
mul-int	0x1404	array-get	0x2003	if-gt-lit	0x3015
div-int	0x1405	goto-var	0x3000	if-ge-lit	0x3016
rem-int	0x1406	if-eq-var	0x3001		
and-int	0x1407	if-ne-var	0x3002		

Figure 4.6: The binary encodings (in hexadecimal form) for each bytecode instruction.

Chapter 5

First-Cut Annotations

The cornerstone of the proposed VM design is that compilers can do costly and complex analysis to discover information that is useful for optimization, but that the information will only be useful if it is retained in some form in the bytecode so that it is available to the JIT. As a rule, the JIT does not have the luxury of duplicating these analyses because it needs to start up quickly and consume minimal resources while the program is running. My solution is to provide ample options for the results of these analyses to be included in the bytecode. Some of this can be placed in the instructions themselves, as noted in the previous section. However, additional flexibility is provided through a separate annotation file and instructions that can reference into it. In this section, I consider some of these annotations.

At the class level, there are obvious advantages for annotations that specify if a given class is immutable or functionally immutable. An immutable class is one for which the values stored in instances cannot change in any way after instantiation. A functionally immutable class relaxes this restriction so that any changes in the object's state are not visible to outside code. An instance of a functionally immutable class might have a mutable

field that is not directly accessible from the outside and which is used only for optimization. Instances of such a class are not as thread safe as a truly immutable class, but they can be safely copied and distributed without needing to communicate changes back to the original. This fact is particularly beneficial in a heterogeneous environment where devices often have separate memory spaces.

The class level annotations can also include the results of analyses such as shape analysis [22]. This annotation can provide the VM with information that is significant for memory allocation or the use of preloading instructions to prevent pipeline stalls. The memory allocation aspect is particularly useful in heterogeneous systems because many devices do not support dynamic memory allocation or have cached memory hierarchies that are standard in CPUs.

Another use of annotations is the specification of full types for generics/parametric types. This information has to be stored in annotations because the types can be arbitrarily long, as it is possible for them to be recursively nested. The optimization benefits of having access to full types is well known from their use in C++ in techniques such as expression templates [25]. Putting the basic types that result from type erasure into the primary bytecode allows the JIT to get running quickly using the limited space in standard instructions. The complete types will be available in annotations for optimizations or for languages that allow run time matching on those types.

The true power of allowing arbitrary annotations in a separate file is realized when one looks at the impact on heterogeneous computing. This area is still in rapid development and will inevitably experience significant changes in the coming years. Annotations aimed at heterogeneous computing specify when blocks of code are particularly well suited for being run as kernels or when they would work particularly well on certain types of devices.

For example, if there is a loop that is data parallel, annotations could be added that

indicate that the range of commands for that loop would be well suited for a kernel and that the kernel would be well suited for execution on a data parallel device such as a GPU. This type of information is distinctly lacking in current platform-independent bytecodes, and it is too much to ask a JIT to do the analysis that would be required to identify them. Even worse, platforms like the JVM throw away a significant amount of information that is associated with the original code, making it even harder to discover what may or may not run well on a particular device. In the near future, systems in which most of the computing power resides in a many-core chip will likely be common. These types of systems will make platforms that cannot effectively use that power far less effective.

One of the other major benefits of JITs is that they can take runtime information into account. This feature is particularly significant in parallel and heterogeneous processing. Deciding whether a particular process should be split across threads or run as a kernel on a particular device can often only happen at runtime when the value of some size parameter is known. Compilers can add annotations estimating how large a parameter needs to be in order for these approaches to be beneficial, but the decision for whether any particular call should be handled in one way or another requires knowing the runtime values.

Chapter 6

Implementation

The current implementation of a virtual machine to run this bytecode is written using the Scala programming language and incorporates OpenCL code (through the JavaCL bindings) to execute code segments on a GPU. Thus, because Scala is a high-level programming language, this implementation is high-level and not intended to be used for real applications. Rather, this version of the VM serves three main purposes: 1) to show that this bytecode can be successfully implemented; 2) to aid in the bytecode and VM design process by bringing to light certain considerations and difficulties that arise when trying implement the design; and 3) to demonstrate that code can be translated at runtime to execute on various devices (in this case, the CPU and the GPU). Also, the current implementation only works with integer values, and not all bytecode instructions are implemented (the figures in Chapter 4 denotes which instructions are not implemented). The following sections describe the current implementation. Code for this implementation is included in Appendix A.

6.1 The Class File Format

All code that the VM runs is contained within a class file. Like in the JVM and Dalvik, a class can contain both static and non-static methods. The main method that the VM should initially run when launched must be static. All of the data contained in the class files are encoded in binary.

In order to provide relevant information to the VM, each class file contains a header. The header format used in this design is modeled largely after the headers in the Dalvik Dex files. However, the headers in this design contain less information than Dalvik's. The current version of the header design contains only the information needed for this implementation. Whereas, for example, Dalvik contains offsets for certain pieces of data, such as the method table, these offsets were not needed for the current implementation. Thus, this data was excluded to simplify the headers. A future implementation might need more information in the headers, so this format is subject to change, especially considering that the current header version does not store any annotation information. Figure 6.1 shows a text representation of an example class file header.

```
10
1
2
3
Class2
4 4
0 5 11
```

Figure 6.1: An example class file header. A header appears at the beginning of each class file to provide the virtual machine with relevant information. In the actual class files, the headers are encoded in binary.

The first item in the header is a two-byte unique number to identify the type of the

class. Because the first eight types (0-7) are reserved for primitives and the ninth type (8) is reserved to denote arrays, this piece of data must be an integer greater than eight. Each class must have a value unique from every other class used in the program. After the unique identifier, the header contains three additional two-byte values indicating the number of classes in the class table, the number of fields in the field table, and the number of methods in the method table, respectively. The class table follows these values and is formatted as a series of strings indicating the names of the other classes that the current class references. Each string is encoded in binary by character and is terminated by a null character ('\0'). After the class table is the field table which consists of a list of two-byte values, each representing one field in the class. Each value indicates the type of the given field. The values are in order by field number (so field number 0 is the first value in the field table and is of the type expressed by that value). Finally, the last item in the header is the method table. This table is represented as a series of four-byte values, each indicating the instruction number where a method starts. Like the field table, the order in the method table indicates the method number. If the class contains a main method, it must be the first method in the class. Also, if the class contains initialization code to be run during object creation, then the first method must begin after the initialization instructions. For example, if the object has three instructions to be executed upon object creation, then the first number in the method table should be 3 to indicate that the first method starts at instruction 3 and, thus, that the first three instructions in the class are initialization instructions.

After the header comes the actual instructions for the class. Each instruction is 16 bytes and follows the format described in Chapter 3. Each method, and the initialization instructions, must end with a return instruction so that the VM knows where each method terminates. Text representations of example class files can be seen in Appendix B. The

binary encoding (as a hexdump produced by the `xxd` UNIX/Linux command) of one of these examples is also included.

6.2 Scala Code

6.2.1 Assembler

To avoid having to write class files directly in binary, I wrote an assembler that takes the files as shown in Appendix B and encodes them in binary. The main method for the assembler is listed in Appendix A.2, but this method simply passes the filename to the `textToBinary` method in the object `MyClass` (Appendix A.3). This method contains the logic to read in the data from the text representation of a class and produce the binary encodings for all of the information. A very useful data structure for both encoding instruction names to binary and decoding the binary to the actual functions to execute is the `instrList` list in the `InstructionList.scala` file (Appendix A.1). This variable is a list of 3-tuples, each of which contains the name of a bytecode instruction, the binary encoding of that instruction, and the Scala function to execute for that instruction. Using the `nameToHex` method, which returns a map from instruction name to binary encoding, the assembler can easily find the binary encoding for each instruction. Similarly, the `hexToFunc` method returns a map from binary encoding to function so that the VM can easily determine how to execute each bytecode instruction.

6.2.2 Class and Object Representation

Because the bytecode instructions can be contained in multiple class files, each class needs to be represented in the VM. This task is accomplished by having a Scala class named `MyClass` (Appendix A.3), which contains the name, type number, class table, field table, method

table, and list of instructions associated with that class. Objects are represented using the Scala class named `MyObject` (Appendix A.4). This class contains a variable that stores a reference to the `MyClass` object to indicate the type of the object. The `MyObject` class also contains an array that is used to store the fields of that object. When the `obj-create` instruction is executed, a `MyObject` object is created, and if the first method in the method table does not begin at instruction zero (indicating that the class has initialization code), then a context shift occurs to run the initialization instructions.

6.2.3 Interpreter

The interpreter is used to run the programs written in this bytecode. The implementation is actually a hybrid interpreter and JIT compiler in that the bytecode instructions are generally interpreted one by one, but the VM also compiles certain sets of instructions to OpenCL kernels at runtime (see Section 6.3 for more information on the OpenCL part of the implementation). For simplicity, I will henceforth refer to the hybrid interpreter/JIT compiler as “the interpreter.”

Machine State

The state for the virtual machine is represented by the Scala class `MachineState` (Appendix A.5). The state keeps track of all of the classes used in the current program, the current class and object (possibly null for static methods) in context, the current instruction number, the current list of instructions, the current set of variables and their types, and the variable location to which to store the return value upon a method return. The state also has a call stack which stores all of the aforementioned data for previous stack frames (i.e., when a context shift occurs, the stack frame shifts to allocate a new set of variables, so the call stack stores the variables and other data from the previous frames so that they can be available

upon returning). The `MachineState` class also contains methods to handle method calls, method returns, and jumps to different instructions.

Bytecode Program Execution

The main method for the interpreter is in the Scala object `Interpreter` (Appendix A.5). When running the interpreter, the only command-line argument is the name of the compiled class file that contains the main bytecode method to execute. This class is loaded into the VM using the `fromBinary` method in the `MyClass` object (Appendix A.3). The other classes used in this code, as referenced in the class tables, are then loaded. The machine state is set up by providing it with the set of classes as well as identifying the class that contains the main method. Finally, the bytecode instructions are executed, branching and performing method calls and returns where appropriate, until the main method returns. At that time, the program execution is finished and the interpreter program terminates. The logic for each bytecode instruction is contained in the `InstructionList` object (Appendix A.1). The only exception to the one-by-one execution of instructions is when code is compiled to run on the GPU, which is covered in the next section.

6.3 OpenCL

In the current version of the interpreter, most of the bytecode instructions are executed sequentially on the CPU. However, parallel GPU execution is also implemented for a specific situation. Because the focus of this project was on the design of the bytecode and VM, and not to study complex program analysis techniques, I was limited in my ability to implement general purpose, heterogeneous program execution. In order to demonstrate that the bytecode can indeed be used to run heterogeneous code, the current VM is able

to perform operations involving two arrays and store the results into a third array. Data parallel operations are well suited for a GPU, so this demonstration of GPU execution is applicable to real situations in heterogeneous computing.

6.3.1 The Annotation File

GPU execution applies to a block of bytecode instructions, so this block must be preceded by an annotation instruction so that the VM knows about the potential optimization. The annotation instruction specifies the type of the annotation and provides the address in the annotation file where that annotation begins. In the current implementation, the address is simply the line number of the first line of data for that annotation in the annotation file.

Like the class file headers, the format of the GPU execution annotation is subject to change based on the information that future versions of the VM require. For the current implementation of the GPU execution annotation, the annotation file first contains the number of lines that this annotation uses in the annotation file. The next line is the number of bytecode instructions that are replaced by this annotation so the VM knows how to continue program execution after the GPU code. Next is the number of input arrays on which to perform the operations, followed by the length of these arrays (the arrays must be the same length). The next line contains a space separated list indicating the variables where the input arrays are stored as well as the variable in which to store the output array. Finally, any remaining lines are valid OpenCL code to perform, with the assumption that the output array will be named `out`, the input arrays will be named `a,b,c,...`, and the global ID of the process will be named `i`. For an example annotation file, see Figure 6.2.

One of the major assumptions in this project is that the compiler can perform complex code analysis to generate informative annotations. Thus, while it may at first seem strange

```
6
7
2
50
1 2 3
out[i] = a[i] + b[i];
```

Figure 6.2: An example annotation file that the virtual machine uses to obtain optimization information. This file is accessed using the annotation instruction, and the format of this file depends on the annotations used, as each annotation type has its own format.

to include OpenCL code within the annotation file itself, the decision makes more sense after considering that the compiler could theoretically generate this code by analysing the bytecode.

6.3.2 Executing Annotations

When the interpreter executes an annotation instruction, it determines the type of the annotation and calls the appropriate function in the `AnnotationDecoder` object (Appendix A.6). The function then extracts the annotation information from the annotation file and calls the correct function in the `AnnotationRunner` object (Appendix A.6). Since the heterogeneous aspect of the current implementation is primarily concerned with running code on the GPU, the `AnnotationRunner` object creates an OpenCL context that contains a GPU device. Then, when a method is called to actually run code on the GPU (such as the `dataParallelTwoInt` method), the appropriate OpenCL variables are initialized and filled with the correct data from the VM variables. The actual code for the OpenCL kernel is then written to a file. This kernel contains references to OpenCL variables, as well as other necessary code like identifying the global ID for the process. The OpenCL code from the annotation file is also inserted into this kernel. An example of an OpenCL kernel

generated by this process is shown in Figure 6.3. Finally, the OpenCL kernel is compiled and executed, and the resulting output array is stored in the appropriate variable as defined in the annotation file.

```
__kernel void my_kernel_0(__global const int* a, __global const int* b,
                          __global int* out, int n) {
    int i = get_global_id(0);
    if(i >= n)
        return;
    out[i] = a[i] + b[i];
}
```

Figure 6.3: An example OpenCL kernel generated by the virtual machine based on a byte-code annotation.

An example class file that executes code on the GPU can be seen in Appendix B.2. Important to note is the annotation instruction that comes right before a block of code that is looped over the arrays. This block of code is then skipped when the VM finishes the GPU execution.

Chapter 7

Conclusion and Future Work

In this thesis, I have presented a bytecode and virtual machine design that prioritizes optimization in a heterogeneous environment. By storing annotations with the bytecode, the JIT compiler can make runtime optimization decisions, taking into account the devices available in the computer. This design provides code portability to different architectures and different combinations of devices, and because many optimization decisions are made at runtime, the application can adapt to the hardware and provide optimizations specific for each platform. For example, consider a data parallel block of code that is well suited for a GPU. In a computer with a powerful GPU, this code will likely perform best if executed on that GPU; however, if a computer does not contain an efficient GPU, the code can be reoptimized to run on the CPU to provide the best performance for the given system.

The field of heterogeneous computing is in a state of constant, rapid evolution. In order to incorporate new advances in this field, the annotation format for this bytecode is extremely flexible and designed for annotations to be added over time. New annotations can be formatted in a way that best organizes the information within, and because most annotations only provide optimization information, code compiled with new annotations is

backwards compatible with older VMs. Preserving backwards compatibility is important for many applications, but this requirement should not limit future advances in optimizability.

I have also developed an initial implementation for a virtual machine to run programs that are compiled to this bytecode. The VM is written in Scala and uses the JavaCL bindings for OpenCL to run code on a GPU. The implementation is a hybrid interpreter/JIT compiler in that a program is mostly interpreted, but certain code segments will be compiled at runtime to OpenCL kernels for GPU execution. While very high-level and not intended for real world use, this VM implementation serves as a demonstration that this bytecode is implementable and that code can be recompiled at runtime to run on a GPU and, thus, other devices that might be present in a heterogeneous environment. Furthermore, because the implementation is written in Scala, which compiles to the JVM, this initial VM can be run on any system with Java and the appropriate OpenCL drivers installed, providing platform independence for this bytecode.

In the future, I plan to implement more annotations in the VM. I want to focus on ways to make more general code compile for GPU execution. Along with implementing more annotations, I will likely need to restructure the class file headers to include annotation information and also develop a way to associate annotation data with specific fields and methods within a class. Because operations that involve moving data to different devices are computationally expensive, I will also work on implementing the annotations that flag immutability and functional immutability. In some cases, immutability information is easily obtained, so including methods to detect immutability in an assembler or compiler may be a relatively easy task and a good place to begin working on the program analysis portion of the bytecode design. I also plan to design more annotations to add to the bytecode to improve optimization potential and determine what types of information should be stored with these annotations. Currently, none of the annotations have a strictly defined structure

for information, so formalizing the representation of annotation data is another important step.

Heterogeneous computing in quickly becoming increasingly important as many manufacturers develop more varieties of devices. As specialized hardware becomes more common in computers, we need a simple, highly adaptable, and platform independent way to program for a wide range of devices. The bytecode and virtual machine design presented in this thesis is a first step towards meeting these goals.

Bibliography

[1] David F. Bacon. Virtualization in the age of heterogeneous machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 1–2, New York, NY, USA, 2011. ACM.

[2] Dan Bornstein. Dalvik VM internals. In *Google I/O Developer Conference*, 2008.

[3] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[4] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34:1–19, October 1999.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[6] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pages 140–150, New York, NY, USA, 1983. ACM.

[7] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM.

[8] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: writing parallel program portable between CPU and GPU. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[9] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 156–167, New York, NY, USA, 2001. ACM.

[10] Mingjie Lin. The amorphous FPGA architecture. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 191–200, New York, NY, USA, 2008. ACM.

[11] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[12] Stefan Marr, Michael Haupt, and Theo D'Hondt. Intermediate language design of high-level language virtual machines: towards comprehensive concurrency support. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 3:1–3:2, New York, NY, USA, 2009. ACM.

[13] Steve Meloan. The Java HotSpot performance engine: An in-depth look. Technical report, Oracle's Sun Developer Network, 1999.

[14] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *J. Funct. Program.*, 12:43–88, January 2002.

[15] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21:527–568, May 1999.

[16] Aaftab Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group, 2011.

[17] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM.

[19] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Incorporation, USA, 2nd edition, 2011.

[20] Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[21] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for simd devices. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI '06, pages 118–131, New York, NY, USA, 2006. ACM.

[22] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, May 2002.

[23] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4:2:1–2:36, January 2008.

[24] Raja Vallée-Rai, Patrick Lam, Clark Verbrugge, Patrice Pominville, and Feng Qian. Soot (poster session): a Java bytecode optimization and annotation framework. In *Addendum to the 2000 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*, OOPSLA '00, pages 113–114, New York, NY, USA, 2000. ACM.

[25] Todd Veldhuizen. *Expression templates*, pages 475–487. SIGS Publications, Inc., New York, NY, USA, 1996.

Appendix A

Implementation Code

A.1 InstructionList

```
package interpreter

object InstructionList {
        val state = new MachineState()

        private val instrList =
                List(
                        //annotation instruction
                        ("annot",0x0000,annot _),

                        //variable instructions
                        ("init-var",0x0001,initVar _),
                        ("set-var",0x0002,setVar _),
                        ("set-field",0x0003,setField _),
                        ("get-field",0x0004,getField _),
                        ("obj-create",0x0005,objCreate _),
                        ("typecast",0x0006,typecast _),

                        ("init-var-lit",0x0011,initVarLit _),
                        ("set-var-lit",0x0012,setVarLit _),
                        ("set-field-lit",0x0013,setFieldLit _),

                        //unary and binary math operations
                        ("not-int",0x1400,notInt _),
```

```
("neg-int",0x1401,negInt _),
("add-int",0x1402,addInt _),
("sub-int",0x1403,subInt _),
("mul-int",0x1404,mulInt _),
("div-int",0x1405,divInt _),
("rem-int",0x1406,remInt _),
("and-int",0x1407,andInt _),
( "or-int",0x1408,orInt _),
("xor-int",0x1409,xorInt _),

("add-int-lit",0x1412,addIntLit _),
("sub-int-lit",0x1413,subIntLit _),
("mul-int-lit",0x1414,mulIntLit _),
("div-int-lit",0x1415,divIntLit _),
("rem-int-lit",0x1416,remIntLit _),
("and-int-lit",0x1417,andIntLit _),
( "or-int-lit",0x1418,orIntLit _),
("xor-int-lit",0x1419,xorIntLit _),

//Array instructions
("new-array",0x2000,arrayNew _),
("new-array-lit",0x2010, arrayNewLit _),
("array-len",0x2001,arrayLength _),
("array-put",0x2002,arrayPut _),
("array-get",0x2003,arrayGet _),

//Control structures
("goto-var",0x3000,gotoVar _),
("if-eq-var",0x3001,ifEqVar _),
("if-ne-var",0x3002,ifNeVar _),
("if-lt-var",0x3003,ifLtVar _),
("if-le-var",0x3004,ifLeVar _),
("if-gt-var",0x3005,ifGtVar _),
("if-ge-var",0x3006,ifGeVar _),
(  "invoke",0x3007,invoke _),
("invoke-static",0x3008,invokeStatic _),
("return",0x3009,myReturn _),
("return-void",0x300A,myReturnVoid _),
("throw",0x300B,myThrow _),

( "goto-lit",0x3010,gotoLit _),
("if-eq-lit",0x3011,ifEqLit _),
("if-ne-lit",0x3012,ifNeLit _),
("if-lt-lit",0x3013,ifLtLit _),
("if-le-lit",0x3014,ifLeLit _),
```

```
                ("if-gt-lit",0x3015,ifGtLit _),
                ("if-ge-lit",0x3016,ifGeLit _)

                )

def nameToHex = instrList.map( t => (t._1,t._2)).toMap
def hexToFunc = instrList.map( t => (t._2,t._3)).toMap
//for testing purposes
def hexToName = instrList.map( t => (t._2,t._1)).toMap

//annotation instruction
def annot(a: Instruction) = {
        val loc = Utilities.buildLong(a.field1,a.field2,
                        a.field3,a.field4).toInt
        val annotType = a.annot

        annotType match {
                case 0 => AnnotationDecoder.dataParallelInt(loc)
                case _ =>
        }
}

//variable instructions
def initVar(a: Instruction) = {
        val dest = a.field1.toInt
        val varType = a.field2.toShort
        Utilities.varsPadTo(dest+1, state)
        state.curTypes(dest) = varType
        val src = a.field3.toInt
        state.curVars(dest) = state.curVars(src)
}
def setVar(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        if(state.curTypes(dest) != state.curTypes(src))
                println("!!! ERROR - set-var type mismatch")
        state.curVars(dest) = state.curVars(src)
}
def setField(a: Instruction) = {
        val obj = state.curVars(a.field1.toInt).asInstanceOf[MyObject]
        val fieldNum = a.field2.toInt
        val src = a.field3.toInt
        obj.fields(fieldNum) = state.curVars(src)
}
def getField(a: Instruction) = {
```

```
        val obj = state.curVars(a.field1.toInt).asInstanceOf[MyObject]
        val fieldNum = a.field2.toInt
        val dest = a.field3.toInt
        state.curVars(dest) = obj.fields(fieldNum)
}
def objCreate(a: Instruction) = {
        val dest = a.field1.toInt
        val cls = state.classes(a.field2.toShort)
        val obj = new MyObject(cls)
        Utilities.varsPadTo(dest+1, state)
        state.curVars(dest) = obj
        state.curTypes(dest) = cls.myType
        if(cls.methodTable(0) > 0)
                state.methodCall(cls, obj, -1, 0, 0, 0, 0)
}
def typecast(a: Instruction) = {
        //TODO
}
def initVarLit(a: Instruction) = {
        val dest = a.field1.toInt
        val varType = a.field2.toShort
        Utilities.varsPadTo(dest+1, state)
        state.curTypes(dest) = varType
        val lit = Utilities.buildLong(a.field3, a.field4, a.field5,
                                a.field6)
        state.curVars(dest) = Utilities.litToType(varType,lit)
}
def setVarLit(a: Instruction) = {
        val dest = a.field1.toInt
        val lit = Utilities.buildLong(a.field2, a.field3, a.field4,
                                a.field5)
        state.curVars(dest) =
                Utilities.litToType(state.curTypes(dest),lit)
}
def setFieldLit(a: Instruction) = {
        val obj = state.curVars(a.field1.toInt).asInstanceOf[MyObject]
        val fieldNum = a.field2.toInt
        val lit = Utilities.buildLong(a.field3, a.field4, a.field5,
                                a.field6)
        val varType = obj.fieldTypes(fieldNum)
        obj.fields(fieldNum) = Utilities.litToType(varType,lit)
}

//math int
def notInt(a: Instruction) = {
```

```
        val dest = a.field1.toInt
        val src = a.field2.toInt
        state.curVars(dest) = ~state.curVars(src).asInstanceOf[Int]
}
def negInt(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        state.curVars(dest) = -state.curVars(src).asInstanceOf[Int]
}
def addInt(a: Instruction) = {
        val dest = a.field1.toInt
        val src1 = a.field2.toInt
        val src2 = a.field3.toInt
        state.curVars(dest) = state.curVars(src1).asInstanceOf[Int] +
                state.curVars(src2).asInstanceOf[Int]
}
def subInt(a: Instruction) = {
        val dest = a.field1.toInt
        val src1 = a.field2.toInt
        val src2 = a.field3.toInt
        state.curVars(dest) = state.curVars(src1).asInstanceOf[Int] -
                state.curVars(src2).asInstanceOf[Int]
}
def mulInt(a: Instruction) = {
        val dest = a.field1.toInt
        val src1 = a.field2.toInt
        val src2 = a.field3.toInt
        state.curVars(dest) = state.curVars(src1).asInstanceOf[Int] *
                state.curVars(src2).asInstanceOf[Int]
}
def divInt(a: Instruction) = {
        val dest = a.field1.toInt
        val src1 = a.field2.toInt
        val src2 = a.field3.toInt
        state.curVars(dest) = state.curVars(src1).asInstanceOf[Int] /
                state.curVars(src2).asInstanceOf[Int]
}
def remInt(a: Instruction) = {
        val dest = a.field1.toInt
        val src1 = a.field2.toInt
        val src2 = a.field3.toInt
        state.curVars(dest) = state.curVars(src1).asInstanceOf[Int] %
                state.curVars(src2).asInstanceOf[Int]
}
def andInt(a: Instruction) = {
```

```
        val dest = a.field1.toInt
        val src1 = a.field2.toInt
        val src2 = a.field3.toInt
        state.curVars(dest) = state.curVars(src1).asInstanceOf[Int] &
                state.curVars(src2).asInstanceOf[Int]
}
def orInt(a: Instruction) = {
        val dest = a.field1.toInt
        val src1 = a.field2.toInt
        val src2 = a.field3.toInt
        state.curVars(dest) = state.curVars(src1).asInstanceOf[Int] |
                state.curVars(src2).asInstanceOf[Int]
}
def xorInt(a: Instruction) = {
        val dest = a.field1.toInt
        val src1 = a.field2.toInt
        val src2 = a.field3.toInt
        state.curVars(dest) = state.curVars(src1).asInstanceOf[Int] ^
                state.curVars(src2).asInstanceOf[Int]
}

//math int literal
def addIntLit(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        val lit = Utilities.buildInt(a.field3, a.field4)
        state.curVars(dest) = state.curVars(src).asInstanceOf[Int] + lit
}
def subIntLit(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        val lit = Utilities.buildInt(a.field3, a.field4)
        state.curVars(dest) = state.curVars(src).asInstanceOf[Int] - lit
}
def mulIntLit(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        val lit = Utilities.buildInt(a.field3, a.field4)
        state.curVars(dest) = state.curVars(src).asInstanceOf[Int] * lit
}
def divIntLit(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        val lit = Utilities.buildInt(a.field3, a.field4)
        state.curVars(dest) = state.curVars(src).asInstanceOf[Int] / lit
```

```
}
def remIntLit(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        val lit = Utilities.buildInt(a.field3, a.field4)
        state.curVars(dest) = state.curVars(src).asInstanceOf[Int] % lit
}
def andIntLit(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        val lit = Utilities.buildInt(a.field3, a.field4)
        state.curVars(dest) = state.curVars(src).asInstanceOf[Int] & lit
}
def orIntLit(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        val lit = Utilities.buildInt(a.field3, a.field4)
        state.curVars(dest) = state.curVars(src).asInstanceOf[Int] | lit
}
def xorIntLit(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt
        val lit = Utilities.buildInt(a.field3, a.field4)
        state.curVars(dest) = state.curVars(src).asInstanceOf[Int] ^ lit
}

//array instructions
def arrayNew(a: Instruction) = {
        val dest = a.field1.toInt
        val arrType = a.field2.toShort
        val size = state.curVars(a.field3.toInt).asInstanceOf[Int]
        Utilities.varsPadTo(dest+1,state)

        state.curTypes(dest) = 8
        state.curVars(dest) = Utilities.makeArray(arrType, size)
}
def arrayNewLit(a: Instruction) = {
        val dest = a.field1.toInt
        val arrType = a.field2.toShort
        val size = Utilities.buildInt(a.field3, a.field4)
        Utilities.varsPadTo(dest+1,state)

        state.curTypes(dest) = 8
        state.curVars(dest) = Utilities.makeArray(arrType, size)
}
```

```
def arrayLength(a: Instruction) = {
        val dest = a.field1.toInt
        val src = a.field2.toInt

        state.curVars(dest) =
                state.curVars(src).asInstanceOf[Array[Any]].length
}
def arrayPut(a: Instruction) = {
        val srctmp = state.curVars(a.field1.toInt)
        val arrtmp = state.curVars(a.field2.toInt)
        val loc = state.curVars(a.field3.toInt).asInstanceOf[Int]


        if(arrtmp.isInstanceOf[Array[Boolean]])
                arrtmp.asInstanceOf[Array[Boolean]](loc) =
                        srctmp.asInstanceOf[Boolean]
        else if(arrtmp.isInstanceOf[Array[Byte]])
                arrtmp.asInstanceOf[Array[Byte]](loc) =
                        srctmp.asInstanceOf[Byte]
        else if(arrtmp.isInstanceOf[Array[Char]])
                arrtmp.asInstanceOf[Array[Char]](loc) =
                        srctmp.asInstanceOf[Char]
        else if(arrtmp.isInstanceOf[Array[Short]])
                arrtmp.asInstanceOf[Array[Short]](loc) =
                        srctmp.asInstanceOf[Short]
        else if(arrtmp.isInstanceOf[Array[Int]])
                arrtmp.asInstanceOf[Array[Int]](loc) =
                        srctmp.asInstanceOf[Int]
        else if(arrtmp.isInstanceOf[Array[Long]])
                arrtmp.asInstanceOf[Array[Long]](loc) =
                        srctmp.asInstanceOf[Long]
        else if(arrtmp.isInstanceOf[Array[Float]])
                arrtmp.asInstanceOf[Array[Float]](loc) =
                        srctmp.asInstanceOf[Float]
        else if(arrtmp.isInstanceOf[Array[Double]])
                arrtmp.asInstanceOf[Array[Double]](loc) =
                        srctmp.asInstanceOf[Double]
        else
                arrtmp.asInstanceOf[Array[MyObject]](loc) =
                        srctmp.asInstanceOf[MyObject]
}
def arrayGet(a: Instruction) = {
        val dest = a.field1.toInt
        val arrtmp = state.curVars(a.field2.toInt)
        val arr =
```

```
                if(arrtmp.isInstanceOf[Array[Boolean]])
                        arrtmp.asInstanceOf[Array[Boolean]]
                else if(arrtmp.isInstanceOf[Array[Byte]])
                        arrtmp.asInstanceOf[Array[Byte]]
                else if(arrtmp.isInstanceOf[Array[Char]])
                        arrtmp.asInstanceOf[Array[Char]]
                else if(arrtmp.isInstanceOf[Array[Short]])
                        arrtmp.asInstanceOf[Array[Short]]
                else if(arrtmp.isInstanceOf[Array[Int]])
                        arrtmp.asInstanceOf[Array[Int]]
                else if(arrtmp.isInstanceOf[Array[Long]])
                        arrtmp.asInstanceOf[Array[Long]]
                else if(arrtmp.isInstanceOf[Array[Float]])
                        arrtmp.asInstanceOf[Array[Float]]
                else if(arrtmp.isInstanceOf[Array[Double]])
                        arrtmp.asInstanceOf[Array[Double]]
                else
                        arrtmp.asInstanceOf[Array[MyObject]]

        val loc = state.curVars(a.field3.toInt).asInstanceOf[Int]

        state.curVars(dest) = arr(loc)
}

//control structures
def gotoVar(a: Instruction) = {
        val dest = state.curVars(a.field1.toInt)
        state.jump(dest.asInstanceOf[Int])
}
def ifEqVar(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = state.curVars(a.field3.toInt)
        val var1 = state.curVars(src1).asInstanceOf[Long]
        val var2 = state.curVars(src2).asInstanceOf[Long]
        if(var1 == var2) state.jump(dest.asInstanceOf[Int])
}
def ifNeVar(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = state.curVars(a.field3.toInt)
        val var1 = state.curVars(src1).asInstanceOf[Long]
        val var2 = state.curVars(src2).asInstanceOf[Long]
        if(var1 != var2) state.jump(dest.asInstanceOf[Int])
}
```

```
def ifLtVar(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = state.curVars(a.field3.toInt)
        val var1 = state.curVars(src1).asInstanceOf[Long]
        val var2 = state.curVars(src2).asInstanceOf[Long]
        if(var1 < var2) state.jump(dest.asInstanceOf[Int])
}
def ifLeVar(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = state.curVars(a.field3.toInt)
        val var1 = state.curVars(src1).asInstanceOf[Long]
        val var2 = state.curVars(src2).asInstanceOf[Long]
        if(var1 <= var2) state.jump(dest.asInstanceOf[Int])
}
def ifGtVar(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = state.curVars(a.field3.toInt)
        val var1 = state.curVars(src1).asInstanceOf[Long]
        val var2 = state.curVars(src2).asInstanceOf[Long]
        if(var1 > var2) state.jump(dest.asInstanceOf[Int])
}
def ifGeVar(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = state.curVars(a.field3.toInt)
        val var1 = state.curVars(src1).asInstanceOf[Long]
        val var2 = state.curVars(src2).asInstanceOf[Long]
        if(var1 >= var2) state.jump(dest.asInstanceOf[Int])
}
def invoke(a: Instruction) = {
        val obj = state.curVars(a.field1.toInt).asInstanceOf[MyObject]
        val cls = state.classes(state.curTypes(a.field1.toShort))
        val methodNo = a.field2.toInt
        state.methodCall(cls, obj, methodNo, a.field3.toInt,
                        a.field4.toInt, a.field5.toInt, a.field6.toInt)
}
def invokeStatic(a: Instruction) = {
        val obj = null
        val cls = state.classes(a.field1.toShort)
        val methodNo = a.field2.toInt
        state.methodCall(cls, obj, methodNo, a.field3.toInt,
                        a.field4.toInt, a.field5.toInt, a.field6.toInt)
```

```
}
def myReturn(a: Instruction) = {
        state.methodReturn(a.field1.toInt, a.field2.toShort)
}
def myReturnVoid(a: Instruction) = {
        state.methodReturn(-1,-1)
}
def myThrow(a: Instruction) = {
        //TODO
}

//control structures literal
def gotoLit(a: Instruction) = {
        val dest = Utilities.buildLong(a.field1, a.field2, a.field3,
                                a.field4)
        state.jump(dest.toInt)
}
def ifEqLit(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = Utilities.buildLong(a.field3, a.field4, a.field5,
                                a.field6)
        val var1 = state.curVars(src1)
        val var2 = state.curVars(src2)
        if(var1 == var2) state.jump(dest.toInt)
}
def ifNeLit(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = Utilities.buildLong(a.field3, a.field4, a.field5,
                                a.field6)
        val var1 = state.curVars(src1)
        val var2 = state.curVars(src2)
        if(var1 != var2) state.jump(dest.toInt)
}
def ifLtLit(a: Instruction) = {
        val src1 = a.field1.toInt
        val src2 = a.field2.toInt
        val dest = Utilities.buildLong(a.field3, a.field4, a.field5,
                                a.field6)
        val var1 = state.curVars(src1).asInstanceOf[Long]
        val var2 = state.curVars(src2).asInstanceOf[Long]
        if(var1 < var2) state.jump(dest.toInt)
}
def ifLeLit(a: Instruction) = {
```

```
                val src1 = a.field1.toInt
                val src2 = a.field2.toInt
                val dest = Utilities.buildLong(a.field3, a.field4, a.field5,
                                        a.field6)
                val var1 = state.curVars(src1).asInstanceOf[Long]
                val var2 = state.curVars(src2).asInstanceOf[Long]
                if(var1 <= var2) state.jump(dest.toInt)
        }
        def ifGtLit(a: Instruction) = {
                val src1 = a.field1.toInt
                val src2 = a.field2.toInt
                val dest = Utilities.buildLong(a.field3, a.field4, a.field5,
                                        a.field6)
                val var1 = state.curVars(src1).asInstanceOf[Long]
                val var2 = state.curVars(src2).asInstanceOf[Long]
                if(var1 > var2) state.jump(dest.toInt)
        }
        def ifGeLit(a: Instruction) = {
                val src1 = a.field1.toInt
                val src2 = a.field2.toInt
                val dest = Utilities.buildLong(a.field3, a.field4, a.field5,
                                        a.field6)
                val var1 = state.curVars(src1).asInstanceOf[Long]
                val var2 = state.curVars(src2).asInstanceOf[Long]
                if(var1 >= var2) state.jump(dest.toInt)
        }
}

object Utilities {
        def buildInt(a: Long, b: Long): Int =
                ((a << 2*8) | b).toInt
        def buildLong(a: Long, b: Long, c: Long, d: Long): Long =
                (a << 6*8) | (b << 4*8) | (c << 2*8) | d

        def litToType(varType: Short, lit: Long) =
                varType match {
                        case 0 => if(lit == 0) false else true
                        case 1 => lit.toByte
                        case 2 => lit.toChar
                        case 3 => lit.toShort
                        case 4 => lit.toInt
                        case 5 => lit.toLong
                        case 6 => lit.toFloat
                        case 7 => lit.toDouble
                        case _ => println("ERROR: type not recognized " +
```

```
                                                "for literal assignment")
                }

        def makeArray(arrType: Short, size: Int) =
                arrType match {
                        case 0 => new Array[Boolean](size)
                        case 1 => new Array[Byte](size)
                        case 2 => new Array[Char](size)
                        case 3 => new Array[Short](size)
                        case 4 => new Array[Int](size)
                        case 5 => new Array[Long](size)
                        case 6 => new Array[Float](size)
                        case 7 => new Array[Double](size)
                        case _ => new Array[MyObject](size)
                }

        def varsPadTo(amt: Int, s: MachineState) {
                if(amt > s.curVars.length) {
                        for(i <- 0 until amt - s.curVars.length) {
                                s.curVars += 0
                                s.curTypes += 0.toShort
                        }
                }
        }
}
```

A.2 Assembler

```
package interpreter

object Assembler {
        //filename must be name of class followed by .het
        //(e.g. for the class "HelloWorld",
        //the filename should be "HelloWorld.het")
        def main(args: Array[String]): Unit = {
                if(args.length != 1) {
                        println("Usage: Assembler infilename")
                        sys.exit(1)
                }
                MyClass.textToBinary(args(0))
        }
}
```

A.3 MyClass

```
package interpreter
import java.io.RandomAccessFile

object MyClass {
        //filename must be name of class followed by .bytecode
        //(e.g. for the class "HelloWorld",
        //the filename should be "HelloWorld.bytecode")
        def fromBinary(filename: String) = {
                val file = new RandomAccessFile(filename, "r")
                val n = filename.substring(0,filename.length-9)
                val t = file.readShort
                val c = new Array[String](file.readShort)
                val f = new Array[Short](file.readShort)
                val m = new Array[Int](file.readShort)

                var tmp = ""
                for(i <- 0 until c.length) {
                        var tmp = "" + file.readChar
                        while(tmp.charAt(tmp.length-1) != '\0') {
                                tmp += file.readChar
                        }
                        c(i) = tmp.substring(0,tmp.length-1)
                }
                for(i <- 0 until f.length) {
                        f(i) = file.readShort
                }
                for(i <- 0 until m.length) {
                        m(i) = file.readInt
                }

                var instrList = collection.mutable.Buffer[Instruction]()
                try {
                        while(true) {
                                instrList += new Instruction(
                                                file.readLong(),file.readLong())
                        }
                }
                catch {
                        case e: java.io.EOFException =>
                }

                new MyClass(n,t,c,f,m,instrList)
        }
```

```
//filename must be name of class followed by .het
//(e.g. for the class "HelloWorld",
//the filename should be "HelloWorld.het")
def textToBinary(filename: String) = {
        val file = scala.io.Source.fromFile(filename).getLines
        val n = filename.substring(0,filename.length-4)+".bytecode"
        val t = file.next.toShort
        val c = new Array[String](file.next.toInt)
        val f = new Array[Short](file.next.toInt)
        val m = new Array[Int](file.next.toInt)

        var temp = file.next
        var tmp = temp.split(" ")//file.next.split(" ")
        for(i <- 0 until c.length) {
                c(i) = tmp(i)
        }

        tmp = file.next.split(" ")
        for(i <- 0 until f.length) {
                f(i) = tmp(i).toShort
        }

        tmp = file.next.split(" ")
        for(i <- 0 until m.length) {
                m(i) = tmp(i).toInt
        }

        val outfile = new RandomAccessFile(n, "rw")
        outfile.writeShort(t) //type
        outfile.writeShort(c.length) //size of class list
        outfile.writeShort(f.length) //size of field table
        outfile.writeShort(m.length) //size of method table

        c.foreach{a =>
                a.foreach(outfile.writeChar(_))
                outfile.writeChar('\0')
        }
        f.foreach(outfile.writeShort(_))
        m.foreach(outfile.writeInt(_))

        val instrs = InstructionList.nameToHex

        while(file.hasNext) {
                val l = file.next.split(" ")
```

```
                        for(i <- 0 until l.length) {
                                if(i == 0) outfile.writeShort(instrs(l(i)))
                                else outfile.writeShort(l(i).toShort)
                        }
                }
                outfile.close
        }
}

class MyClass(val name: String, val myType: Short,
                val classTable: Array[String], val fieldTable: Array[Short],
                val methodTable: Array[Int],
                val instrs: scala.collection.mutable.Buffer[Instruction])
```

A.4 MyObject

```
package interpreter

class MyObject(val myType: MyClass) {
        val fields = new Array[Any](myType.fieldTable.length)
        val fieldTypes = myType.fieldTable
}
```

A.5 Interpreter

```
package interpreter

import scala.collection.mutable

object Interpreter {
        def main(args: Array[String]): Unit = {
                if(args.length != 1) {
                        println("Usage: Interpreter filename")
                        sys.exit(1)
                }
                val filename = args(0)
                val mainClass = MyClass.fromBinary(filename)
                val classes = mutable.HashMap((mainClass.myType,mainClass))
                loadClasses(mainClass, classes)
                InstructionList.state.setup(mainClass, classes)
                execute()

        }
```

```
	def loadClasses(c: MyClass, col: mutable.HashMap[Short,MyClass]) {
		val tmp =
			c.classTable.map(a => MyClass.fromBinary(a + ".bytecode"))
		tmp.foreach{a =>
			if(!col.contains(a.myType)) {
				col += ((a.myType,a))
				loadClasses(a,col)
			}
		}
	}

	//executes the set of instructions
	//prints machine state (first 10 vars only) after each
	//instruction finishes executing
	def execute() {
		println("Begin Program Exection\n")
		val instrs = InstructionList.hexToFunc
		val state = InstructionList.state
		while(state.keepRunning) {
			val num = state.curInstrNum
			val i = state.getCurInstr

			instrs(i.opcode)(i)
			state.nextInstr()
			printState(state, num)
		}
		println("End Program Exection")
	}

	def printState(state: MachineState, num: Int) {
		var numVars = 11
		println("After Instruction "+num+" in class "+state.curClass.name)
		println("VarNum\tValue\tType")
		val times = if(state.curVars.length > numVars) numVars
				    else state.curVars.length
		for(i <- 0 until times)
			println(i+"\t"+state.curVars(i)+"\t"+state.curTypes(i))
		println()
	}
}

class MachineState {
	var instrUpdated = false
	var keepRunning = true
```

```
var classes: mutable.HashMap[Short,MyClass] = null

var curClass: MyClass = null
var curObj: MyObject = null

var curInstrNum = 0
var curInstrList: mutable.Buffer[Instruction] = null
var curVars = mutable.Buffer[Any]()
var curTypes = mutable.Buffer[Short]()
var retLoc: Int = -1

//curClass, curObj, curInstr, curInstrList, curVars, curTypes, retLoc
val callStack =
        mutable.Stack[(MyClass,MyObject,Int,mutable.Buffer[Instruction],
                mutable.Buffer[Any],mutable.Buffer[Short],Int)]()

def getCurInstr = curInstrList(curInstrNum)

def setup(c: MyClass, cList: mutable.HashMap[Short,MyClass]) {
        classes = cList
        curClass = c
        curInstrNum = c.methodTable(0)
        curInstrList = c.instrs
}

def methodCall(c: MyClass, o: MyObject, methNum: Int,
                arg1: Int, arg2: Int, arg3: Int, arg4: Int) {

        Utilities.varsPadTo((arg1 max arg2 max arg3 max arg4)+1, this)
        val tmpVars = mutable.Buffer[Any](o,curVars(arg1),curVars(arg2),
                                        curVars(arg3),curVars(arg4))
        val tmpTypes =
                mutable.Buffer[Short](c.myType,curTypes(arg1),
                        curTypes(arg2),curTypes(arg3),curTypes(arg4))

        callStack.push((curClass,curObj,curInstrNum,curInstrList,
                        curVars,curTypes,retLoc))

        curClass = c
        curObj = o
        curInstrNum = if(methNum == -1) 0 else c.methodTable(methNum)
        instrUpdated = true
        curInstrList = c.instrs

        curVars = tmpVars
```

```
        curTypes = tmpTypes
        retLoc = arg4
}

def methodReturn(varLoc: Int, retType: Short) {
        if(callStack.isEmpty) {
                keepRunning = false
        }
        else if(varLoc == -1) {
                val popped = callStack.pop

                curClass = popped._1
                curObj = popped._2
                curInstrNum = popped._3
                curInstrList = popped._4
                curVars = popped._5
                curTypes = popped._6
                retLoc = popped._7
        }
        else {
                val tmp = curVars(varLoc)
                val tmpRetLoc = retLoc
                val popped = callStack.pop

                curClass = popped._1
                curObj = popped._2
                curInstrNum = popped._3
                curInstrList = popped._4
                curVars = popped._5
                curTypes = popped._6
                retLoc = popped._7

                Utilities.varsPadTo(tmpRetLoc+1, this)
                curVars(tmpRetLoc) = tmp
                curTypes(tmpRetLoc) = retType
        }
}

def nextInstr() {
        if(!instrUpdated) curInstrNum += 1
        instrUpdated = false
}

def jump(num: Int) {
        instrUpdated = true
```

```
                curInstrNum = num
        }
}

class Instruction(a: Long, b: Long) {
        def opcode = ((a >>> 6*8) & 0xffff).toInt
        def field1 = ((a >>> 4*8) & 0xffff)
        def field2 = ((a >>> 2*8) & 0xffff)
        def field3 = ((a >>> 0*8) & 0xffff)
        def field4 = ((b >>> 6*8) & 0xffff)
        def field5 = ((b >>> 4*8) & 0xffff)
        def field6 = ((b >>> 2*8) & 0xffff)
        def annot  = ((b >>> 0*8) & 0xffff)
}
```

A.6 Annotations

```
package interpreter

import com.nativelibs4java.opencl._
import com.nativelibs4java.opencl.util._
import com.nativelibs4java.util._
import org.bridj.Pointer
import org.bridj.Pointer._
import java.lang.Math._
import java.io.File
import java.io.PrintWriter

object AnnotationRunner {
        val context = JavaCL.createBestContext(CLPlatform.DeviceFeature.GPU)
        val queue = context.createDefaultQueue()

        val state = InstructionList.state
        var kernelNum = 0

        def dataParallelTwoInt(arr1: Array[Int], arr2: Array[Int],
                        code: String) = {
                println("Running Annotation on")
                context.getDevices().foreach(println _)
                val n = arr1.length
                val aPtr = allocateInts(n)
                val bPtr = allocateInts(n)

                for(i <- 0 until n) {
```

```
                aPtr.set(i, arr1(i))
                bPtr.set(i, arr2(i))
        }

        val a = context.createIntBuffer(CLMem.Usage.Input, aPtr, true)
        val b = context.createIntBuffer(CLMem.Usage.Input, bPtr, true)
        val out = context.createIntBuffer(CLMem.Usage.Output,n)

        val funcName = """my_kernel_"""+kernelNum
        val fileName = "myKernel"+kernelNum+".cl"
        kernelNum += 1
        val line = """__kernel void """+funcName+
        """(__global const int* a, __global const int* b,"""+
        """ __global int* out, int n) {
                int i = get_global_id(0);
                if(i >= n)
                        return;
                """ + code + "\n}"

        val pw = new PrintWriter(fileName)
        pw.println(line)
        pw.close

        val src = IOUtils.readText(new File(fileName))
        val program = context.createProgram(src)

        val myKernel = program.createKernel(funcName)
        myKernel.setArgs(a,b,out,n: java.lang.Integer)
        val addEvt = myKernel.enqueueNDRange(queue, Array.fill(1)(n))

        val outPtr = out.read(queue, addEvt)

        val ret = new Array[Int](n)
        for(i <- 0 until n)
                ret(i) = outPtr.get(i)

        ret
    }
}

object AnnotationDecoder {
    val file = scala.io.Source.fromFile("annotFile.txt").getLines.toArray
    val state = InstructionList.state

    def dataParallelInt(loc: Int) {
```

```
                println("Decoding Annotation")
                val numLines = file(loc).toInt
                val bytecodeLines = file(loc+1).toInt
                val numArrays = file(loc+2).toInt
                val arrLen = file(loc+3).toInt
                val arrVars = file(loc+4).split(" ").map(_.toInt)
                var code = ""
                for(i <- loc+5 until numLines)
                        code += file(i)

                if(numArrays == 2) {
                        val a = state.curVars(arrVars(0)).asInstanceOf[Array[Int]]
                        val b = state.curVars(arrVars(1)).asInstanceOf[Array[Int]]
                        val dest = arrVars(2)

                        state.curVars(dest) =
                                AnnotationRunner.dataParallelTwoInt(a, b, code)
                        state.jump(state.curInstrNum+bytecodeLines+1)
                }
        }
}
```

Appendix B

Example Class Files

B.1 CPU Only Execution

B.1.1 Class 1

```
10
1
2
3
Class2
4 4
0 5 11
invoke-static 10 1 0 0 0 1 0
invoke-static 10 2 0 0 0 3 0
obj-create 4 11 0 0 0 0 0
invoke 4 0 0 0 0 5 0
return-void 0 0 0 0 0 0 0
init-var-lit 0 4 0 0 0 0 0
init-var-lit 1 4 0 0 0 5 0
if-eq-lit 0 1 0 0 0 10 0
add-int-lit 0 0 0 1 0 0 0
goto-lit 0 0 0 7 0 0 0
return 0 4 0 0 0 0 0
init-var-lit 0 4 0 0 0 6 0
init-var-lit 1 4 0 0 0 1 0
if-eq-lit 0 1 0 0 0 16 0
sub-int-lit 0 0 0 1 0 0 0
```

```
goto-lit 0 0 0 13 0 0 0
return 0 4 0 0 0 0 0
```

B.1.2 Class 2

```
11
0
1
1

4
2
set-field-lit 0 0 0 0 0 8 0
return-void 0 0 0 0 0 0 0
init-var-lit 1 4 0 0 0 0 0
get-field 0 0 1 0 0 0 0
add-int-lit 1 1 0 2 0 0 0
return 1 4 0 0 0 0 0
```

Class 2 Binary (Hexdump)

```
0000000: 000b 0000 0001 0001 0004 0000 0002 0013  ................
0000010: 0000 0000 0000 0000 0000 0008 0000 300a  ..............0.
0000020: 0000 0000 0000 0000 0000 0000 0000 0011  ................
0000030: 0001 0004 0000 0000 0000 0000 0000 0004  ................
0000040: 0000 0000 0001 0000 0000 0000 0000 1412  ................
0000050: 0001 0001 0000 0002 0000 0000 0000 3009  ..............0.
0000060: 0001 0004 0000 0000 0000 0000 0000       ..............
```

B.2 GPU Execution

```
12
0
1
2

4
0 14
invoke-static 12 1 0 0 0 10 0
init-var-lit 1 4 0 0 0 0 0
init-var-lit 2 4 0 0 0 1 0
init-var-lit 3 4 0 0 0 2 0
init-var-lit 4 4 0 0 0 47 0
init-var-lit 5 4 0 0 0 48 0
init-var-lit 6 4 0 0 0 49 0
```

```
array-get 1 10 1 0 0 0 0
array-get 2 10 2 0 0 0 0
array-get 3 10 3 0 0 0 0
array-get 4 10 4 0 0 0 0
array-get 5 10 5 0 0 0 0
array-get 6 10 6 0 0 0 0
return-void 0 0 0 0 0 0 0
new-array-lit 1 4 0 50 0 0 0
new-array-lit 2 4 0 50 0 0 0
new-array-lit 3 4 0 50 0 0 0
init-var-lit 4 4 0 0 0 0 0
init-var-lit 5 4 0 0 0 50 0
init-var-lit 6 4 0 0 0 0 0
init-var-lit 7 4 0 0 0 0 0
if-eq-lit 4 5 0 0 0 27 0
array-put 4 1 4 0 0 0 0
add-int-lit 4 4 0 1 0 0 0
array-put 4 2 6 0 0 0 0
add-int-lit 6 6 0 1 0 0 0
goto-lit 0 0 0 21 0 0 0
set-var-lit 4 0 0 0 0 0 0
annot 0 0 0 0 0 0 0
if-eq-lit 4 5 0 0 0 36 0
array-get 6 1 4 0 0 0 0
array-get 7 2 4 0 0 0 0
add-int 6 6 7 0 0 0 0
array-put 6 3 4 0 0 0 0
add-int-lit 4 4 0 1 0 0 0
goto-lit 0 0 0 29 0 0 0
return 3 8 0 0 0 0 0
```