DYNAMIC TRANSLATION OF RUNTIME
ENVIRONMENTS FOR HETEROGENEOUS
COMPUTING

A Thesis Presented
by

Rodrigo Domínguez

to
The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in the field of
Computer Engineering

Northeastern University
Boston, Massachusetts

May 2013
Abstract

The recent move toward heterogeneous computer architectures calls for a global rethinking of current software and hardware paradigms. Researchers are exploring new parallel programming models, advanced compiler designs, and novel resource management techniques to exploit the features of many-core processor architectures. Graphics Processing Units (GPUs) have become the platform of choice in this area for accelerating a large range of data-parallel and task-parallel applications. The rapid adoption of GPU computing has been greatly aided by the introduction of high-level programming environments such as CUDA C and OpenCL. However, each vendor implements these programming models differently and we must analyze the internals in order to get a better understanding of the performance results.

This dissertation presents the design of Caracal, a dynamic compiler that can translate between runtime environments used in heterogeneous computing. A major challenge of developing such a translation system is the inherent difference in both the underlying instruction set architecture, as well as the runtime system. One of the more challenging questions across different runtime environments is the handling of program control flow by the compiler and the hardware. Some implementations can support unstructured control flow based on branches and labels, while others are based on structured control flow relying solely on if-then and while constructs.

Caracal can translate applications based on unstructured control flow so they can
run on hardware that requires structured programs. In order to accomplish this, Caracal builds the control tree of the program and creates single-entry, single-exit regions called hammock graphs. We use Caracal to analyze the performance differences between NVIDIA’s implementation of CUDA C and AMD’s implementation of OpenCL. Our results show that the requirement for structured control flow can impact register pressure, and can degrade performance by as much as 2.5X.

We also explore the differences between heterogeneous parallel processors architectures that will impact the design of a translator when attempting to optimize code. We evaluate this issue for two specific compiler optimizations that are highly sensitive to the underlying architecture. We use Caracal to evaluate 1) vectorization and 2) loop unrolling. Our translation system needs to capture rich semantics to be able to effectively translate code to different systems that will exploit these two optimization techniques differently. Our experiments show that by tuning these two optimizations in our translation system, we can reduce execution time by up to 90%.
Acknowledgements

First, I would like to thank my wife and my son for their support throughout this years. I also want to thank my parents who have always been a role model to me. To my advisor, Dr. David Kaeli, thank you for challenging me to be better and stirring up the ideas that led to this thesis. Finally, there is a long list of family, friends, and colleagues who were part of this process and who I would like to thank.
# Contents

Abstract iii

Acknowledgements v

1 Introduction 1

1.1 The Evolution of High Performance Computing 1
1.2 Programming Languages for Heterogeneous Systems 3
1.3 Runtime System 5
1.4 Binary Translation 7
1.5 The Role of Intermediate Representations 9
1.6 Contributions of the Work 10
1.7 Organization of this Thesis 11

2 Related Work 13

2.1 Binary Translation 13
2.2 Cross-compilation 16
2.3 Virtual Machines 17
2.4 Intermediate Representations 18
2.5 Control Flow Analysis 20
3 The Dynamic Translation Framework 23

3.1 Programming Models 24
3.2 Software Stack 25
3.3 Dynamic Translation 28
3.4 Portability 33
3.5 Experimental Results 35
3.6 Translating to Other Architectures 39
3.7 Summary 42

4 Unstructured Control Flow 43

4.1 Hammock Graphs 44
4.2 Cut Transformation 46
4.3 Backward Transformation 46
4.4 Forward Transformation 48
4.5 Experimental Results 48
4.6 Summary 50

5 Compiler Optimizations 52

5.1 Register Rematerialization 54
5.2 Instruction Scheduling 55
5.3 Vectorization and Loop Unrolling 59
5.4 Experimental Results 61
5.5 Summary 63

6 Contributions and Future Work 64
List of Figures

1.1 Compiler from N source languages to M targets. ...................... 8
3.1 Hardware organization of a GPU. ................................. 24
3.2 CUDA C and OpenCL software stacks. ......................... 26
3.3 Example of a vector addition kernel ......................... 27
3.4 CUDA C and OpenCL software stacks including Caracal. ... 29
3.5 Structural Analysis. ........................................ 32
3.6 Unstructured Code. ........................................ 32
3.7 Histogram256 example. ..................................... 34
3.8 Scheduling for the Histogram256 example. .................. 35
3.9 Translation vs Execution time. .................................. 36
3.10 Performance of Caracal versus OpenCL. ...................... 39
4.1 Two different approaches to handling control flow .............. 44
4.2 Hammock Graph ........................................... 45
4.3 Cut transformation algorithm. .................................. 46
4.4 Cut transformation graph. .................................... 47
4.5 Backward transformation algorithm. .......................... 47
4.6 Backward transformation graph. .............................. 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7</td>
<td>Forward transformation algorithm</td>
<td>49</td>
</tr>
<tr>
<td>4.8</td>
<td>Forward transformation graph</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Register Rematerialization</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Instruction scheduling algorithm</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>Register pressure</td>
<td>57</td>
</tr>
<tr>
<td>5.4</td>
<td>Register rematerialization speedup</td>
<td>58</td>
</tr>
<tr>
<td>5.5</td>
<td>Vectorization Pass</td>
<td>61</td>
</tr>
</tbody>
</table>
List of Tables

3.1 CUDA and CAL address spaces. ........................................ 28
3.2 Benchmarks from the CUDA SDK 3.2 ............................... 37
3.3 Register Usage ......................................................... 40
4.1 Unstructured Control Flow Results ................................... 50
5.1 Vectorization - Execution time ....................................... 62
Chapter 1

Introduction

1.1 The Evolution of High Performance Computing

For the past few decades, the semiconductor industry had been able to offer customer value based mainly on the benefits of Moore’s Law [1] which states that the number of transistors in a chip would double every 18 months. However, even though this rate of scaling is still going on, there are several limitations that are challenging the future of high performance computing including power consumption, clock speed scaling, instruction-level parallelism, and performance of traditional cache hierarchies.

Symmetric Multi-Processor (SMP) and multi-core chips attempt to address these challenges by including multiple cores in the same system or package. This presents a good short-term solution and leverages the benefits of thread-level parallelism present in many applications. However, in the long term, SMPs and multi-core chips are still limited by the amount of performance per watt that they can attain. Also, new applications are characterized by unstructured parallelism and media-rich inputs,
something SMP systems are not particularly good at.

Some of these challenges have already been addressed by the embedded world where the prevailing answer has been application specific cores (ASICs) integrated with general-purpose units. The advantage of this approach is the improved performance per watt and the flexibility of programming. For example, a programmable cellular accelerator can support multiple signal processing algorithms. Similarly, a programmable video engine can execute the many codecs in use today. For graphics, the programmable shaders can handle a wide range of special effects and at the same time they can carry on general-purpose computations.

With this in mind, the industry is directing their attention to heterogeneous designs that include a CPU and an accelerated unit. The new goal is to create an architecture that has good throughput computing (i.e., it can execute more tasks in a certain amount of time) giving less relevance to single-thread performance. The key design criteria to build such a machine will be: (a) to have a powerful general-purpose processor that can execute heavily control-dependent applications; (b) incorporate a large number of small and optimized cores for accelerations; (c) include an optimized memory system that can handle a lot of transactions efficiently; and (d) provide a programming model that enables programmer productivity.

Such a heterogeneous architecture is already being explored by researchers from the graphics community. Graphics Processing Units (GPU) are architectures that have evolved into more programmable processors compared to what they were in the last decade. Most of the evolution has come from the demands of the graphics and gaming sector which have created the necessity of programmable shader cores.

GPUs have become the accelerator of choice in a number of key high performance applications. GPUs can obtain from 10x to 100x speedups in terms of execution
performance compared to CPUs. In 2006, NVIDIA introduced the CUDA C language \cite{CUDA} which lowered the threshold of entry to reap the benefits of GPU computing. CUDA C provides a C-like programming environment with simple extensions and a runtime environment to allow developers to execute non-graphics applications, called kernels, on the GPU. These accelerators are designed to exploit the data-level and task-level parallelism present in a wide range of applications.

1.2 Programming Languages for Heterogeneous Systems

The wide variety of programming languages for many-core computing can be classified into three categories: shared memory programming languages, distributed memory programming languages, and languages for heterogeneous systems.

1.2.1 Shared Memory Programming Languages

Shared memory programming became popular with the introduction of Symmetric Multi-Processor (SMP) systems. The most popular implementation is POSIX Threads, or Pthreads. Pthreads is a low-level API supported directly by the operating system that developers can use to spawn new schedulable units. Pthreads in SMP systems make use of primitives like locks and mutual exclusion to communicate with each other. The biggest drawback of using Pthreads is that developers need to rewrite their code when the number of nodes scales in the system.

As an answer to this limitation, OpenMP proposes a compiler-supported API based on pragma-annotated source code that can be easily scaled to a greater number of cores. Also, OpenMP parallel annotations can be turned off by the compiler to
target single core architectures.

Other alternatives are based on higher-level languages and runtime support. This is the case of Intel’s *Thread Building Blocks* (TBB). TBB is a C++ template library that abstracts most of the low-level details in APIs like Pthreads. These libraries abstract the operations as tasks that can be allocated to individual cores at runtime. Other parallel programming languages, such as *Cilk* are implemented as extensions to C/C++, adding special keywords to express parallelism directly in the source code.

### 1.2.2 Distributed Memory Programming Languages

For architectures with distributed memory, libraries usually implement communication using messages containing the data to be exchanged. MPI is a library specification that supports message passing between programs running on distributed memory architectures (i.e., clusters). MPI provides functions that manage the communication of data from the memory space of one process to the memory space of another process. The specification is usually implemented on top of the network layer (TCP/IP) and provides bindings for C/C++. MPI also suffers from the scalability problem as Pthreads and developers have to be cautious about the delays for clusters where the communication time dominates the execution.

### 1.2.3 Heterogeneous Programming Languages

Heterogeneous programming environments are usually composed of two parts: a runtime environment that handles all the memory management and execution, and a compiler. In the case of the Cell processor, for example, the programming language is traditional C/C++ for the PPU and SPU processors, and adds a runtime environment provided as a library. At the same time, heterogeneous systems lack a formal
operating system and all its provided services. This implies that context switches are generally too expensive given the amount of state that has to be preserved in this systems.

Accelerator-based programming languages are usually based on explicit data transfers between the CPU and the accelerator. The most recent developments are CUDA C [2] and OpenCL [3]. These languages follow the *bulk-synchronous programming* (BSP) [4] model where a group of threads execute in parallel and synchronization is specified by the programmer. Under these models, thousands of fine-grain data-parallel computations are carried on by the accelerators in an efficient way.

Most of the programming models for heterogeneous systems involve memory with multiple hierarchies that must be explicitly managed by the software. This makes heterogeneous computing complex and hard to port across platforms. Moreover, applications can have different types of parallelism at the instruction, data, or task level. The programmer has the arduous task of stating the problem at hand according to these levels of parallelism and mapping them to the most appropriate core.

### 1.3 Runtime System

The runtime system is a layer of software that allows programs written in some computer language to be executed on a particular hardware. Runtime systems provide services to the applications such as thread management, memory allocation, synchronization, and communication. By leveraging an rich runtime system, application developers can abstract some of the more time-consuming and hardware-specific details making their code efficient and portable.

In the case of heterogeneous systems, the job of the runtime system is to make it
easy for application developers to access any of the cores (CPU and accelerators) and harness all the power effectively. Current runtime systems are usually not efficient nor general enough, while application developers are most concerned with programmability. In order to provide solutions to these competing goals, runtime systems will have to support new heterogeneous architectures.

Some examples of runtime systems designed for multithreaded architectures are Cilk++ [5] and Intel TBB [6]. The Cilk++ programming environment includes a compiler and the runtime system. Cilk++’s runtime system implements three basic extensions to C++ for thread spawning, thread synchronization, and divide-and-conquer parallel loops. The Cilk++ runtime system employs a work-stealing scheduler to dynamically and automatically exploit an arbitrary number of available processor cores. Intel Thread Building Blocks (TBB) is a threading abstraction runtime that provides a portable implementation of parallel algorithms in C++. The runtime automatically manages a thread pool and does dynamic load balancing. Intel TBB allows the developer to specify tasks that can be executed concurrently and the runtime maps these tasks onto threads in an efficient way.

Runtime systems for heterogeneous architectures have also emerged over the past few years. One type of such runtime systems is stream programming. In stream programming the inputs and outputs are streams of data, similar to the model found in multimedia applications. Realizations of this programming model can be found in StreamIt [7] and Brook [8] runtime environments. In both cases, the runtime takes advantage of the characteristics of the inputs to map the computation to parallel architectures using SIMD instruction sets. However, stream programming can run into limitations when compiling applications that are more general and exhibit more complex control flow.
Another type of runtime systems for heterogeneous architectures that has become popular is bulk-synchronous programming (BSP) [4]. Under this programming model, applications are divided into multiple threads that can execute in parallel independently, and only communicate at predetermined synchronization points specified by the developer. Two of the most representative runtime systems implementing the BSP model today are CUDA C [2] and OpenCL [3] for graphic processors. In CUDA C and OpenCL threads are organized in a hierarchical fashion, with different memory consistency models presented at each level. The programmer can guarantee consistency among the threads by using synchronization intrinsics in the language.

1.4 Binary Translation

Binary translation is a technique used to compile applications between two different instruction sets [9]. The purpose is to emulate the target instruction set and provide portability for applications that have already been compiled into a binary format. There are two main types of binary translation: static and dynamic.

In static binary translation, the idea is to perform the translation offline without running the application [10]. In this case, most static translators take a conservative approach and translate all the binary code including parts that may never get executed. Therefore, static binary translation tends to slowdown the application’s execution. Also, static binary translation only has a partial view of the characteristics of the program.

In dynamic binary translation, the translator performs the compilation by running the application and operating on a finer granularity – usually at the basic block level. A software code cache is utilized to hold the translation of basic blocks already
executed. This allows the translator to run faster compared to the native execution. At the same time, regions of code that tend to be executed more frequently can be recompiled for higher performance allowing the translator to apply optimization based on profiling [11, 12, 13]. Compared to static translation, dynamic binary translation can make use of profiling information to optimize only those regions of code that are executed more frequently and disambiguate data accesses.

The largest obstacle for adopting dynamic binary translation is the additional overhead of the runtime optimizer. However, as code caches and other techniques have been introduced, dynamic translators have become faster by compiling the code only once. Today, dynamic compilers can perform as well or better that their static counterparts. Finally, dynamic translation has been crucial to application portability. In most cases, the main purpose of using dynamic translation has been to retarget precompiled applications onto new architectures.
1.5 The Role of Intermediate Representations

Intermediate representations (IR) are abstract languages that can be used by compilers to describe machine operations without being tied to an specific instruction set target. The front-end part of the compiler does lexical analysis, parsing, semantic analysis, and translation to the intermediate representation. The back-end part of the compiler does optimizations on the intermediate representation and code generation to a machine language. IRs are used by compilers for two purposes:

- **Portability**: Without an IR, compiling N different source languages to M different machines would require $N \times M$ compilers, as shown in Figure 1.1a. A portable compiler translates the source language into an IR and then translates the IR into machine language, as shown in Figure 1.1b. This way only $N + M$ compilers have to be built.

- **Modularity**: When developing a compiler, an IR can keep the front-end of the compiler independent of machine-specific details. At the same time, an IR can help the back-end compiler to be independent of the peculiarities of the source language, therefore making the development of the compiler more modularized.

There are several factors involved in the design of an intermediate representation. The IR should be appropriate for the language(s) to be compiled and the target architecture(s). Also, some type of optimizations may be easier and faster on certain IRs. It is not uncommon to find that some compilers use more than one IR at different points in the translation process depending on the optimization being performed. Other factors that are important in the design of an IR are how machine-dependent it is, its structure, and its expressiveness.
In the case of intermediate representations used by heterogeneous systems with accelerators, it is important to design an IR that can express both the graphics and the general-purpose applications being compiled. Since most of the IRs in this realm originate from compilers for graphic applications with SIMD characteristics, they should be adapted to more general computations which have more complex control flows and a more typical MIMD nature.

Another reason for using intermediate representations is to achieve platform independence. Such is the case of IRs like the Java bytecode [14]. Java programs are compiled into bytecode that is translated by the virtual machine onto multiple platforms. Another example is the Parallel Thread eXecution (PTX) [15] virtual instruction set that is used by NVIDIA to represent programs to be translated just-in-time to several of their target GPUs.

1.6 Contributions of the Work

The key contribution of this thesis is the development of an open-source dynamic translation framework called Caracal. We designed an IR that can be used to represent any compute kernel for GPUs. This IR takes into consideration differences in both instruction sets and runtime systems. Caracal uses this IR as a tool to translate CUDA C programs so they can run on alternative GPU platforms addressing important issues of portability in heterogeneous computing from a compiler perspective. Once CUDA C programs are represented using this IR they can be dynamically recompiled to AMD’s Compute Abstraction Layer (CAL) platform. This creates a framework that we use to evaluate automatic optimizations. The benefits of this approach to portability and translation are numerous and are detailed in
Chapter 3.

Other contributions of this thesis include:

- A comparison between two heterogeneous architectures and runtime environments that can be used in the analysis of GPU performance and programability.

- A framework that can be used by researchers to explore compiler optimizations and runtime techniques that map a range of programs to different CPU/GPU architectures.

- A study of the challenges in dynamic translation and code generation when cross-compiling between these two massively parallel platforms.

- A description of the characteristics of the intermediate representations used by the compilations toolkits that can influence future designs.

- A study of cross-compiling and optimizations for heterogeneous computing.

1.7 Organization of this Thesis

Chapter 2 provides a summary of existing GPU compiler environments including compiling tools and runtime environments. It also presents a summary of techniques for binary translation and cross-compilation. Chapter 3 introduces the theory and concepts behind Caracal. We describe in detail the method to cross-compile CUDA C programs including the runtime environment and the intermediate representation. Chapter 4 presents the algorithm for turning an unstructured control flowgraph into an structured control flowgraph by building regions of code called hammock graphs.
We evaluate the effect of this algorithm on code size and register pressure. Chapter 5 describes the optimizations implemented in Caracal targeted to AMD’s GPU hardware and evaluates the performance. Finally, Chapter 6 discusses opportunities for future work.
Chapter 2

Related Work

Our work implements a dynamic translation framework that allows applications written for a specific execution environment to execute on a different second environment. We used this framework to evaluate a cross-platform environment for heterogeneous computing. In this thesis we use binary translation and compiler optimizations to arrive at these goals. In this chapter we first present prior work on binary translation for different platforms followed by research on cross-compilation techniques in high performance computing. Finally, we discuss some of the related work on virtual machines that pertains to our research.

2.1 Binary Translation

One of the main motivations for binary translation [9] systems has been the portability of compiled applications across different ISAs. The importance of portability is twofold: (a) for hardware designers, introducing radical changes to the ISAs is a risky proposition specially because of the customer deployed software base; (b) for software developers, porting an application to a new instruction set always represents
recompilation and, therefore, the potential introduction of new bugs. Another reason for implementing a binary translator is the gains in performance by executing a just-in-time compiler that can benefit from profiling.

There has been numerous projects in this area. In [16], Lee et al. describe Latte, a virtual machine (VM) that can perform lightweight dynamic optimizations to Java applications. This framework uses a two-pass algorithm to perform a fast register allocation: a backward pass to construct preferred mappings to registers, and a forward pass with the actual mappings. Additionally, Latte uses a lightweight monitor for thread synchronization, an on-demand translator of exception handlers, and a fast mark-and-sweep garbage collector. The performance reported by the authors is comparable to Sun’s Hotspot and JDK 1.2.

Dynamo [13] is a dynamic compiler developed at Hewlett-Packard. It uses profiling techniques to identify hot traces of execution at runtime. It monitors the target of backward-taken branches until they become hot. At that point, it performs a lightweight optimization of the target trace and stores them in a software code cache. These traces may belong to multiple procedures or libraries, allowing Dynamo’s optimizations to have a bigger scope compared to a static compiler. As the software code cache becomes stable, the performance of Dynamo improves.

Another important framework was developed by Digital Equipment Corporation and is called Digital FX!32 [11]. Digital FX!32 uses both emulation and binary translation to run x86 applications on DEC’s Alpha processors. The first time an x86 application is run, it works in the emulation mode and at the same time gather profiling information. Subsequently, Digital FX!32 feeds the profiling information to its binary translation engine to detect hot regions of code. During the binary translation, it performs both traditional compiler optimizations as well as special
optimizations for the Alpha architecture.

Similarly, the DAISY project [17] implements a translator that allows PowerPC applications to run on VLIW architectures. DAISY performs binary translation with precise exceptions. The first time a piece of code is executed, it is rapidly translated to VLIW code and saved in memory. Subsequent executions of the same code can reuse previous translations. The translation is done under the control of a Virtual Machine Monitor (VMM). In a related project, Transmeta implemented the *Code Morphing Software* [18] (CMS) used in its Crusoe microprocessors to translate x86 binaries to run on a VLIW architecture. Given that Crusoe and CMS are production-quality tools, they need to consider special cases not covered by other binary translation systems such as exceptions and interrupts, I/O, DMA, and self-modifying code.

Dynamite [19] is a dynamic binary translator created by Transitive Technologies. It provides several back-ends and front-ends and has been licensed by companies like IBM and Apple. The translator does most of the runtime optimizations on an intermediate representation of the code. The overhead is quite low making it suitable for embedded systems. Another project that involves the definition of an intermediate representation is Dixie [12]. Dixie is a simulation tool for computer architects to analyze multiple processors including x86, PowerPC, and Alpha. Dixie’s approach to support such a big number of binaries is to define a virtual ISA and use a compiler to translate between the host ISA and the virtual ISA. Dixie operates in two modes: an emulation mode where the virtual machine interprets the virtual ISA; and a binary translation mode where the virtual ISA is precompiled to the native instruction set so the simulation runs faster.
2.2 Cross-compilation

Programming languages for parallel systems are varied. It is common to find developers advocate to use of a particular language based on its virtues. Some of the virtues that developers seek include ease of transition from serial to parallel versions and good scalability across number of cores. This is the main reason for the development of cross-compilers that can adapt familiar programming languages and leverage on the huge software code base already written by others. Some examples for parallel computing are presented here.

There have been several proposals to implement programming models for SMP systems on different architectures. Probably the most notable attempt was the development of High Performance Fortran (HPF), an extension of Fortran 90 for parallel computing on distributed memory machines. HPF included compiler directives for the distribution of data across the nodes. Another example of cross-compilation of shared-memory programming models to clusters is the work by Basumallik [20] et al. where they designed a source-to-source compiler from OpenMP to MPI using an approach called partial replication where shared data is allocated on all nodes but no shadow copies or management structures need to be allocated.

Previous research has also focused on source-to-source cross-compilation for heterogeneous languages. For example, Harvey et al. implemented a source-to-source compiler tool called Swan [21] that can be used to compile CUDA C programs to run on CPU and GPU backends. Internally, Swan compiles CUDA C into OpenCL using regular expression substitution and makes use of the OpenCL library to run on multiple platforms. Another example of cross-compilation is the work developed by Lee [22] et al. compiling OpenMP programs for shared-memory parallel computers.
into CUDA C for GPUs. In this prior work, the authors use the Cetus compiler infrastructure [23] developed a system to interpret OpenMP semantics and translate them to the CUDA programming model, identify kernel regions, and apply optimizations such as loop collapsing to minimize the amount of communication between the CPU and the GPU. In MCUDA [24], Stratton et al. use source-to-source transformations to map CUDA C to multicore CPUs. They describe an algorithm to wrap kernel statements inside thread loops and perform loop fission to preserve the semantics of synchronization barriers.

O’Brien [25] et al. use compiler transformations based on the IBM XL compiler in collaboration with a runtime library to support OpenMP on the Cell processor. The systems makes use of the heterogeneity in the Cell processor and efficiently synchronizes the threads running on the PPE and SPEs. Their work focuses on issues of thread synchronization, code generation for the multiple-ISA processors, and mapping of the OpenMP memory model.

2.3 Virtual Machines

Virtual machines are software tools used mainly to solve the problem of portability in common applications.

An example of a mainstream virtual machine is QEMU[26]. QEMU supports full system emulation of several CPUs (x86, PowerPC, SPARC, ARM) on various hosts (x86, PowerPC, SPARC, ARM, Alpha and MIPS). At the same time, QEMU can also run in user mode emulation where it can execute an unmodified Linux process compiled for a target CPU on a different host CPU. QEMU consists of an efficient dynamic translator that can compile the most frequently executed code and saves it
into a code cache. By using a code cache, QEMU avoids recompiling the code multiple times. QEMU splits each target CPU instruction into fewer simpler micro operations so that the number of combinations of instructions is smaller and the translation can be done faster.

### 2.4 Intermediate Representations

Compilers use intermediate representations (IR) as the stepping stone to map the user’s program into the machine language. Designing a good IR has been a topic of research for a long time. One of the most important characteristics of a good IR is the flexibility that it can give the compiler to implement optimizations and perform data analysis. Several program representations have been proposed in this scope.

NVIDIA introduced its Parallel Thread Execution (PTX) [15] to allow programs written in CUDA to be compiled to this virtual instruction set. This way, PTX kernels could be compiled at runtime by the driver depending on the specific GPU card used at the moment. This also allows NVIDIA to update its microarchitecture and machine instructions without losing backward compatibility as long as the driver can compile the PTX code. PTX is characterized by being a load-store instruction set, i.e. operands have to loaded and stored from memory before operating on them using specific memory instructions. It resembles a assembly-like language with additional features like scoping and type casting. Another important characteristic is the handling of control flow based on jumps and conditional branches to user-specified labels. In general, a CUDA program is compiled into a PTX kernel that can implicitly express parallelism using intrinsics to differentiate among the data points in the grid. Initially, NVIDIA used the Open64 compiler to generate PTX. As of CUDA
4.0, NVIDIA transitioned to using LLVM to compile to PTX.

Similarly, AMD has been using an intermediate representation called Intermediate Language [27] to compile different types of shaders for graphics (vertex and pixel shaders) and computing (compute shaders). Given that IL was originally design for graphics, it presents several instructions that are graphics-only related. The handling of control flow is expressed using higher-level constructs including if-then-else and while-loops. IL is characterized for being typeless (register hold raw 32-bit values) and having a number of swizzles and permutations when accessing the register file.

In 2011, AMD announced it was working on a new intermediate representation called FSAIL [28] which highlights the importance of adapting the current IRs for GPGPU. The IR is used for its new Fusion chips that integrate both CPU and GPU on the same substrate. In order to accommodate this new architecture, FSAIL is agnostic to both the CPU and the GPU and includes the management of the multiple address spaces included in the graphics card plus the system memory. For the applications where a unified address space is more convenient and memory aliasing is not an issue, FSAIL includes appropriate new instructions. FSAIL will be the target of parallel programs and the input to AMD’s just-in-time compiler, also called Finalizer. More importantly, FSAIL includes instructions for exception handling, virtual functions, and other high level language features.

With GPGPU becoming more and more popular, we see a growing number of related studies in the area of intermediate representations. Companies, as well as academic research, have introduced new programming models for GPGPU that also include new IRs[29].
2.5 Control Flow Analysis

After the compiler has parsed the input program and built the syntax tree, it performs optimizations to the instruction stream. However, at this point the syntax tree representation of the program is insufficient for most of the analysis that it needs to perform in order to produce more optimized code. Therefore, the compiler must build a different representation of the program including information regarding the flow of control within each procedure. This information is necessary so the compiler can make sure that whatever optimizations it performs, the functional behavior of the program is maintained and all the semantics are preserved. This representation is often referred to as the Control Flow Graph (CFG).

There are two main approaches to control flow analysis. The first one builds on the concept of dominators and post dominators to discover loops. The second one, called interval analysis, performs an analysis of the overall structure of the procedure and attempts to decompose it into nested regions called intervals.

It is the second approach the one that is more relevant to this thesis. The nesting structure of the intervals forms the Control Tree. The more sophisticated form of interval analysis, called structural analysis [30], performs a postorder traversal of the syntax tree, identifying regions that correspond to the structured constructs and reducing them into abstract nodes. The original method for structural analysis was introduced by Sharir [31] and it is the one we use in Caracal. This method was later extended by Zhang et al.[32].

In work related to both control flow analysis and data flow analysis, Ferrante et al. present a Program Dependence Graph in [32] (PDG). The idea behind the PDG is to provide a representation that can be targeted by compilers to represent both data and control dependencies. By analyzing the PDG, several compiler optimizations can
operate more efficiently. The PDG allows transformations that requires information on both types of dependencies, such as loop vectorization, to be handled more easily.

Frequently, compilers use transformations like the single static assignment (SSA) form to improve the performance and reduce the compilation time. In [33], Cytron et al. propose an algorithm that efficiently computes the Control Dependence Graph and the SSA.

### 2.6 Summary

The benefits of binary translation have been proven in the history of hardware design. Every time we have reached an inflection point in hardware design, binary translation has been an interesting alternative to the problem of backwards compatibility. As we discussed in this section, that was the case with the introduction of new instruction sets coming from DEC’s Alpha processors or HP’s VLIW architecture. This gave raise to works like Dynamo [13] and Digital FX!32 [11]. This time, we believe we are at yet another inflection point: the advent of heterogeneous processors. Therefore, we believe our work involving binary translation of run-time systems for heterogeneous computing is relevant to the community and can leverage from the research cited in this section.

At the same time, the question of programming language portability has been addressed by various scientist in our field. The success of programming languages like OpenMP and MPI is based on the assumption that applications written in these languages can be executed on many platforms. We cited a few projects were applications were cross-compiled between languages or platforms. In our research, we contribute to this area by analyzing the implications of executing real applications across two
related, yet significantly different, ISAs and runtimes.
Chapter 3

The Dynamic Translation Framework

One of the main challenges that hinders heterogeneous computing to become a mainstream approach is the question of portability across different hardware vendors and architectures. Programming languages such as CUDA C and OpenCL are still in early stages and require the programmer to tune their code that heavily depends on the targeted platform. This leads to software that is vendor-specific, expensive to develop/maintain, and difficult to migrate.

With this in mind, we developed a framework that can be used to map explicitly parallel execution languages onto heterogeneous architectures. The framework, which we call Caracal, dynamically translates binaries targeted for a first architecture/runtime to a second architecture/runtime. The key is to provide an intermediate representation that is rich enough to span both architectural and runtime anomalies.

To demonstrate the utility of Caracal, we build an implementation that dynamically translates the CUDA C programming model so it can run on AMD’s software
stack. Caracal allows us to investigate on the issues of hardware and software design that are fundamentally different across GPUs. Also, Caracal can be used to evaluate automatic compiler optimizations that can take some of the burden off of the programmer.

This chapter presents the two programming models considered in this thesis, their associated software stacks, and various implementation aspects of Caracal.

### 3.1 Programming Models

Figure 3.1 shows the hardware organization of a GPU. It consists of a scalable array of SIMD cores with several memory address spaces: (a) an off-chip memory connected via a high-bandwidth network (called *global memory*), (b) an on-chip software-managed scratch pad (called *shared memory*), and (c) a *register file* for local storage that is shared among the threads executing in the core.
CHAPTER 3. THE DYNAMIC TRANSLATION FRAMEWORK

The threads running on the GPU are organized in a hierarchical fashion, with different memory consistency models presented at each level. The smallest schedulable unit is called a warp (wavefront in AMD’s terminology), which is a group of threads that share the program counter and, therefore, execute the same instruction in a lockstep fashion. The memory model at this level is sequential consistency.

At the next level, a group of warps forms a block. Threads belonging to the same block are guaranteed to execute on the same Single Instruction Multiple Data (SIMD) core and can communicate through the on-chip scratch pad memory for fast memory accesses. At this level, the memory model changes to a relaxed consistency model, requiring explicit synchronization barriers.

A collection of blocks constitutes a grid whose threads are executed across the entire array of SIMD cores in no specific order and without synchronization. Threads from different blocks can only communicate through the off-chip memory using atomic instructions and memory fences. These hierarchical levels and consistency models give hardware designers more flexibility and allows the architecture to scale easily. Note that GPUs are basically designed to perform graphics rendering, a highly data-parallel workload.

3.2 Software Stack

Figure 3.2 shows the software stacks for CUDA C and OpenCL. NVIDIA introduced CUDA C[2] to allow users to easily write code that runs on their GPUs. CUDA C consists of a simple set of extensions to C and a runtime library. Using CUDA C, developers can launch functions (referred to as kernels) to be executed on the GPU.

The programmer must partition the problem as a grid of blocks of threads when
launching a kernel. Each thread can use intrinsics to identify its thread number as well as other launch configuration parameters. Figure 3.3a shows a kernel that adds two arrays $A$ and $B$ of $N$ floats and writes the result to a third array $C$.

Developers use the NVIDIA compiler $\texttt{nvcc}$\[34\] to create an executable. During compilation, the CUDA kernel code is translated to an IR format. The executable code calls the CUDA runtime (which in turn calls the driver) to compile the kernel IR into machine code and launch the application.

NVIDIA has defined an IR called PTX\[15\] that acts as a bytecode and allows the hardware to evolve without having to recompile legacy applications. This IR is just-in-time compiled by the driver, so only the driver needs to worry about the actual architecture in the system. Similarly, AMD defines an IR called Intermediate Language (IL)\[27\]. Figure 3.3b and Figure 3.3c show the PTX and IL corresponding to the vector addition example from Figure 3.3a.

Both PTX and IL allow for an unlimited number of virtual registers and defer register allocation to the driver. Following these steps, most of the kernels are compiled directly into a Static Single Assignment (SSA) form, making it easier for the driver to perform back-end optimizations.
(a) CUDA C

```c
__global__ void vecAdd(
    float* A,
    float* B,
    float* C,
    int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

(b) PTX

```
.entry vecAdd (
    .param float A,
    .param float B,
    .param float C,
    .param int N)
{
    move u16 rh1, ctAdd.x;
    move u16 rh2, ntId.x;
    mul wide u16 r1, rh1, rh2;
    cvt u32 u16 r2, tId.x;
    add u32 r3, r2, r1;
    ld.param s32 r4, [N];
    setp le.u32 p1, r4, r3;
    @p1 bra Label_1;
    cvt u64 s32 rd1, r3;
    mul le.u64 rd2, rd1, 4;
    id.param u64 rd3, [A];
    add u64 rd4, rd3, rd2;
    ld.global.f32 r1, [p4+d+0];
    ld.param u64 rd5, [B];
    add u64 rd6, rd5, rd2;
    ld.global.f32 r2, [r6+d+0];
    add r32 r3, r3, r2;
    ld.global.f32 rd7, [C];
    add u64 rd8, rd7, rd2;
    st.global.f32 [r8+0].f3;
    Label_1:
        exit;
}
```

(c) IL

```
.cls_2_0
.dcl_raw_uav_id(0)
.dcl_ch cbo[2]
.dcl_ch cbo[4]
.dcl_literal 10, 4, 4, 4, 4
mov r0, vThreadGrpid.x
mov r1, cbo[0].x
imul r2, r0, r1
mov r3, vTidInGrp.x
iadd r4, r3, r2
mov r5, cbo[4]
ige r6, r4, r5
if_logical r6
mov r7, r4
imul r8, r7, 10
mov r9, cbo[0]
iadd r10, r9, r8
uav_raw_load_id(0) r11, r10
mov r12, cbo[1]
iadd r13, r12, r8
uav_raw_load_id(0) r14, r13
add r15, r11, r14
mov r16, cbo[2]
iadd r17, r16, r8
uav_raw_store_id(0) r17, r15
endif
end
```

Figure 3.3: Example of a vector addition kernel. (a) CUDA C source code; (b) NVIDIA’s PTX representation; and (c) AMD’s IL representation.
Caracal is based on the Ocelot translation framework for heterogeneous systems [35, 36, 37]. Ocelot allows CUDA C programs to run on NVIDIA GPUs and x86 CPUs. Ocelot implements the CUDA Runtime API [38] as a library that CUDA C programs can be linked to without any source modifications. It includes an internal representation for PTX, a PTX parser, a set of PTX to PTX transformation passes, a PTX emulator, and a dynamic compiler to many-core CPUs and NVIDIA GPUs. Table 3.1 shows the address spaces exposed by CUDA and CAL. We omit the local and texture address spaces since they are outside of the scope of this thesis.

In our work, we leverage the Ocelot front-end and use this prior work to develop our new IR. In this thesis we focus our IR development to target AMD GPUs. Figure 3.4 shows the software stack, including the Caracal framework that described in the next section. We discuss difficulties that arise due to differences in the memory system, the IR formats, and the architectural details.

### 3.3 Dynamic Translation

#### 3.3.1 Global Memory

To implement global memory, we allocate a global buffer in CAL that uses as the global memory space and access it within the kernel using an Unordered Access View
(UAV) [39]. We declare the UAV typeless (raw) in which case it is byte-addressable and memory operations must be double word-aligned. Additionally, we declare an Arena UAV for byte and short memory accesses and we bind it to the same buffer.

Caracal handles dynamic global memory allocations (i.e. \texttt{cudaMalloc}) by managing the memory in the global buffer. We use a straightforward memory management algorithm. UAVs support atomic operations similar to the way they are handled in PTX.

### 3.3.2 Shared Memory

Shared memory is byte-addressable in both environments. However, PTX allows for shared memory accesses of different sizes: from one byte (e.g., \texttt{ld.shared.u8}) up to four double words (e.g., \texttt{ld.shared.v4.u32}). In IL, shared memory accesses must be aligned to a four-byte boundary and the size must always be one double word.

Therefore, when translating a load operation from shared memory whose size is smaller than a double word, we need to extract the correct value from the result. This issue also arises for store operations to shared memory, where we have to merge the value with the contents in memory using the appropriate mask. This incurs an
overhead of 7 IL instructions for loads and 13 IL instructions for stores.

PTX also provides variable declarations in shared memory (e.g. `shared varname`). These variables can be used later as pointers (e.g. `mov r1, varname`). In IL, variables are not declared at all. In this case, the translator has to manage the layout of variables targeted for IL’s shared memory space.

### 3.3.3 Constant Memory

Besides global and shared memory, PTX also defines a constant memory region with an on-chip cache. CAL exposes a similar concept called Constant Buffers (CB) [39]. However, CBs are accessed in a non-linear way with each address consisting of 4 components (x, y, z, and w). This difference in memory architectures makes translation of constant memory accesses non-trivial. Given the scope of this thesis, we map constant memory to global memory (UAV). The downside of this approach is that we lose out on the benefits of caching constant memory in Caracal. This remains an area for future work in our framework.

### 3.3.4 IR Translation

In PTX, variables can be scalars or vectors of two or four components. However, only some instructions (`mov`, `ld`, `st`, and `tex`) allow vectors as operands. For all other instructions, vectors have to be unpacked into scalar variables.

On the AMD side, IL treats all variables as vectors of four components and all the instructions support vector operands. IL instructions have a rich set of modifiers to their operands including swizzles (e.g. `mov r0, r1.xxyy`).

Since we are translating from an environment that does not directly support vectors (versus AMD that supports vector operands for all instructions), this is not a
problem. First, the translator implements a straightforward solution mapping each PTX scalar variable to one of the components of an IL variable. Later, we explore an optimization pass that applies vectorization techniques when translating the PTX scalar variables. We describe and evaluate this optimization in Chapter 5. Note that since both PTX and IL allow for an unlimited number of virtual registers, our translation framework does not have to worry about register allocation nor spilling code.

One of the more challenging differences between PTX and IL is the representation and handling of control flow instructions. PTX is based on branch instructions and labels (e.g., @pi bra label), while IL is based on structured constructs without labels (e.g., iflogicalz-endif). In order to perform correct translation and preserve proper program control flow, we need to identify the regions of the control flow graph (CFG) that match the constructs available in IL. Our implementation uses a technique called Structural Analysis [30, 40] which has been used before for data-flow analysis in optimizing compilers.

The goal of Structural Analysis is to build the control tree of the program – a tree-shaped representation where the leaf nodes are the individual basic blocks of the CFG and the rest of the nodes represent the constructs recognized by the target language. The algorithm works by doing a postorder traversal of the depth-first spanning tree, identifying regions that correspond to the structured constructs and reducing them into abstract nodes to build the control tree in the process. The algorithm stops when the abstract flowgraph is reduced to one node and this becomes the root node of the control tree. The translator can then iterate through the control tree starting from the root node and recursively process each one of the nodes. Figure 3.5a shows a Control Flow Graph and Figure 3.5b shows the corresponding Control Tree using
This algorithm.

It is possible that the algorithm will not match any regions in the graph to the associated IL constructs. In some cases, we can perform transformations, as described in the next Chapter, to turn the CFG into a reducible graph. The case of short-circuit evaluation of if statements in C/C++, as illustrated in Figure 3.6, is a good example. We have developed techniques based on the algorithm described in [41] to solve this kind of problem. We present the results in the next Chapter.

In other cases, the graph could turn out to be irreducible. In other words, there
is no way to represent the CFG using the constructs available in IL. In this case we are not able to translate the program without modifications to the source code.

### 3.4 Portability

Some applications are written to work with an specific warp size in mind. The warp size is used explicitly in the code to determine, for example, the dimensions of a data structure that is shared among the threads in the warp. We consider the implications of translating between two architectures with different warp sizes.

A good example to illustrate the potential impact of these differences is Histogram256 from the NVIDIA SDK. This program provides an efficient implementation of a 256-bin histogram. This application takes a set of values between 0 and 255 and finds the frequency of occurrence of each data element. In order to achieve good performance, the CUDA application divides the histogram into sub-histograms that are stored in shared memory and later merged into one.

Histogram256 must resolve memory collisions from multiple threads updating the same sub-histogram bin. However, this application was implemented to run on GPUs that did not have hardware support for shared memory atomics. Consequently, the benchmark uses the code shown in Figure 3.7 to implement a software version of an atomic operation in shared memory:

The application allocates a different sub-histogram to each warp and tags the bin counters according to the last thread that wrote to them. The parameters are a pointer to the per-warp sub-histogram, the value from the input set, and a tag that is unique to each thread in the warp.

A thread starts by reading the value of the counter and masking off the tag of
the previous thread. The counter is incremented and the tag is replaced with the one from the current thread. Then each thread writes back the new counter to the sub-histogram in shared memory.

In case of a collision between multiple threads from the same warp, the hardware commits only one of the writes and rejects the others. After the write, each thread reads from the same shared memory location. The threads that are able to write the new counter exit the loop. The threads involved in a collision whose writes are rejected execute the loop again. The warp continues execution when all the threads exit the loop.

It is important to allocate a different sub-histogram to each warp so there are no race conditions. Consider the schedule shown in Figure 3.8 involving two threads from different warps (indicated by the numbers on the left):

If the threads were sharing the same sub-histogram and they were updating the same counter, this would lead to a race condition (thread X would increment the counter twice). Hence, it is necessary that threads from different warps do not share

```c
__device__ void addData256(
  volatile unsigned int *hist,
  unsigned int val,
  unsigned int tag)
{
  unsigned int c;
  do {
    c = hist[val] & 0x07FFFFFFFU;
    c = tag | (c + 1);
    hist[val] = c;
  } while (hist[val] != c)
}
```

Figure 3.7: Histogram256 example.
When starting with NVIDIA CUDA, we are translating from an architecture with a warp size of 32 to the AMD target with a warp size of 64. Fortunately, since 64 is a multiple of 32, we end up with two sub-histograms per warp, making this mapping natural. However, it is worth noting that if we were translating from a larger to a smaller warp size, this would have led to two warps sharing the same sub-histogram and resulted in a race condition.

### 3.5 Experimental Results

Next, we present our evaluation of the Caracal dynamic compiler. Our test platform is a 64-bit Linux (Ubuntu 12.10) system running on an Intel Core i7-920 CPU and an ATI Radeon HD5870 GPU. We ran the experiments using Caracal (rev. 2016), AMD’s APP SDK 2.8, and the ATI Catalyst 10.9 driver.

The benchmarks we used for the performance evaluation are the applications selected from the CUDA SDK shown in Table 3.2. We ported the CUDA benchmarks to OpenCL performing a one-to-one mapping of the language keywords and built-in functions and preserving the same optimizations on both versions. We verified 100%
functional correctness of our framework compared with CPU execution.

The first set of experiments evaluate the performance of Caracal, and include transformations necessary to translate PTX kernels into IL. There are 3 major steps in launching a kernel in Caracal: the IR translation, the compilation to machine code (performed by the driver), and the actual execution of the application. Figure 3.9 compares the translation time against the execution time. The results show that, on average, the translation time is less than 1/2 ms while the execution time is around 1 ms. Further analysis showed that most of the translation time is spent building the control tree and executing the structural analysis.

The algorithm used to build the control tree in Caracal operates on a branch by branch basis and it is dominated by the structural analysis presented in Chapter 4. This algorithm has $O(n^2)$ complexity. Other researchers have introduced linear time algorithms that build the control tree in an incremental and iterative way. In [42], Hauser et al., proposed an incremental approach to the transformation of the control flow that significantly reduced the overhead of the translation. Similarly, Johnson et
## Table 3.2: Benchmarks from the CUDA SDK 3.2

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Working Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Bitonic Sort</td>
<td>Sorting network of $O(n \log^2 n)$ comparators. Performs best when sorting a small number of elements.</td>
</tr>
<tr>
<td>CNV</td>
<td>Convolution</td>
<td>Separable convolution filter between a 2D image and a Gaussian blur filter.</td>
</tr>
<tr>
<td>HST</td>
<td>Histogram256</td>
<td>256-bin histogram calculation of an arbitrary-sized 8-bit data array.</td>
</tr>
<tr>
<td>SCN</td>
<td>Scan</td>
<td>Scan algorithm with $O(\log n)$ steps and $O(n)$ adds. Uses a balanced tree type algorithm (scan_best).</td>
</tr>
<tr>
<td>TRN</td>
<td>Transpose</td>
<td>Matrix transpose optimized to coalesce accesses to shared memory and avoid bank conflicts.</td>
</tr>
<tr>
<td>SCP</td>
<td>Scalar Product</td>
<td>Scalar product of multiple pairs of vectors.</td>
</tr>
<tr>
<td>SRG</td>
<td>Sobol QRNG</td>
<td>Sobol’s method for generating low-discrepancy sequences.</td>
</tr>
<tr>
<td>VEC</td>
<td>Vector Addition</td>
<td>Element-by-element vector addition.</td>
</tr>
<tr>
<td>SNT</td>
<td>Sorting Networks</td>
<td>Bitonic sort with $\theta(n \log(n)^2)$ comparators.</td>
</tr>
<tr>
<td>DWT</td>
<td>Haar Discrete Wavelet Transform</td>
<td>1-dimensional discrete wavelet transform for Haar signals.</td>
</tr>
<tr>
<td>BSC</td>
<td>BlackScholes</td>
<td>Fair call and put prices for a set of European options using Black-Scholes formula.</td>
</tr>
<tr>
<td>FWT</td>
<td>Fast Walsh Transform</td>
<td>Naturally-ordered Fourier transformation.</td>
</tr>
<tr>
<td>MAT</td>
<td>Matrix Multiplication</td>
<td>Traditional matrix multiplication using shared memory.</td>
</tr>
<tr>
<td>RED</td>
<td>Reduction</td>
<td>Parallel sum reduction using warp-synchronous programming.</td>
</tr>
</tbody>
</table>
al. [40], provided a linear time algorithm for finding hammock graphs and analyzing each one independently. Exploring a more efficient algorithm to build the control tree is part of the future work of this thesis.

We found that a good number of benchmarks (e.g., parallel sum reduction, Haar discrete wavelet transform, sorting networks) are inherently recursive. These applications take a divide-and-conquer approach to solving the problem at hand and invoke the same kernel on different input sets taken from the whole range. Also, problems such as a parallel sum reduction, zoom into the answer by reducing the size of the input set on each iteration of the recursion. For such cases, implementing a code cache to store the translation of the kernel the first time it is called and reusing it in subsequent invocations could be a good idea. This would greatly reduce the amount of overhead introduced due to building the control tree, especially for benchmarks with unstructured control flow. We leave the evaluation of the code cache technique as future work.

The next set of experiments, presented in Figure 3.10, show a comparison of the execution time of a kernel in Caracal versus native OpenCL. The time measurements were taken using the gettimeofday system call in Linux around the kernel launch API (calCtxRunProgramGrid in Caracal and clEnqueueNDRangeKernel in OpenCL). Caracal performs comparably to OpenCL with a small overhead of 2.1x.

The largest overhead is found in Convolution (4.5x). This benchmark implements a separable convolution between a 2D image and a Gaussian filter. The Gaussian filter is declared in constant memory and is accessed by each thread inside a loop over the width of the filter. We attribute the overhead to the fact that Caracal maps constant memory to a UAV (instead of a Constant Buffer) and, therefore, the filter does not get cached.
As another metric of identifying code quality, Table 3.3 shows the per-thread register usage for both the native OpenCL kernels and the translated CUDA kernels. We collected the register usage from the AMD Stream Profile 1.4. All the kernels translated by Caracal have a register usage similar to that of the OpenCL library.

Convolution is the benchmark with the highest register usage. Given the level of register pressure, Convolution is limited by the number of thread blocks that can run on each SIMD core. This makes the application especially sensitive to memory delays since it lacks enough threads to hide the latency. This follows the performance overhead results presented in Figure 3.10.

3.6 Translating to Other Architectures

We have created a methodology for translating between runtime environments used in heterogeneous computing. The following is a list of the main features from CUDA C to take into consideration when translating the CUDA runtime environment to other
Table 3.3: Register Usage

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Caracal</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>CNV</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>HST</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>SCN</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>TRN</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>SCP</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>SRG</td>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>VEC</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>SNT</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>DWT</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>BSC</td>
<td>14</td>
<td>31</td>
</tr>
<tr>
<td>FWT</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>MAT</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td>RED</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

architectures:

- **Control Flow**: As we discussed in this section, support for control flow is probably one of the main differences to take into consideration. It is important to determine if the target architecture supports branches and labels and if it can describe unstructured control flow. Also, newer versions of PTX (version 3.0 and later) include indirect branches. In order to translate indirect branches it may be necessary to collect traces and profile the application.

- **Address Spaces**: CUDA presents 5 different address spaces to the application developer: global, shared, constant, parameter, and the local state space. Each one of these address spaces is byte-addressable and has different initialization requirements (for example, constant memory can be initialized while parameter memory can not be initialized). It is important to identify similar address spaces exposed by the target architecture.
Synchronization Mechanisms: Threads communicate and synchronize through barriers, fences, and atomic instructions. The target architecture should provide the same synchronization mechanisms or a way to emulate them in software.

Memory Model: In CUDA, threads are organized hierarchically across the data domain: 1) threads from the same warp execute the same instruction at the same time in lock-step fashion; 2) threads from the same group execute in relaxed consistency and use barriers to synchronize, and 3) threads from the same grid execute the code in relaxed consistency as well but they can only communicate using fences. Preserving these memory models in the target architecture will become important. Otherwise, the target would end up supporting only a limited subset of benchmarks and performance would be impacted.

Register Types: Some intermediate representations use vector register types based on four channels (red, green, blue, alpha). This is due to the graphic nature of the applications that still need to be supported by the hardware. PTX was designed for general-purpose applications and its registers are scalar. Packing and unpacking passes may be necessary to achieve better efficiency.

Transcendental Operations: Different intermediate representations will most probably expose different operations. This is especially true for transcendental operations. The dynamic translation framework should make sure that the semantics of the PTX instruction and the target instruction are the same. When this is not the case, the translation process should try to emulate such PTX instructions using a combination of target instructions.
3.7 Summary

So far we have presented the design of the Caracal translation framework, and discussed various portability and compatibility issues between the NVIDIA CUDA and AMD CAL runtime environments. The experiments show that, although both programming models are similar and target GPUs, there are also some subtle differences that needed to be accounted for in our IR. In particular, source-code optimizations that make assumptions on the warp size are hard to port between platforms and can lead to data races that are very hard to detect by the programmer. We identified many of these issues by translating full applications from the NVIDIA SDK. In this process, we observed that a key difference between NVIDIA’s PTX and AMD’s IL is how they handle control divergence. In the case of PTX, control is implemented using branch instructions and named labels. For AMD’s IL, control flow is handled using well-structured constructs typical of higher-level languages (like C or C++). The implications of this differences are explained in detail in Chapter 4.

In this chapter we also reported that Caracal performs comparably to other IL compilers such as AMD’s OpenCL. The only exception we found was when applications make extensive use of constant memory. We attribute this difference to the fact that we currently do not use constant buffers available in the AMD Evergreen architecture. After profiling the Caracal framework, we observed that most of the time is spent building the control tree, which makes us think that implementing an offline translator together with a code cache could improve the performance.

The analysis of structured versus unstructured control flow and an algorithm to transform an unstructured program are presented in the next chapter.
Chapter 4

Unstructured Control Flow

One of the main differences in the implementation of the SPMD programming model is the handling of control flow: the threads executing the program can diverge upon encountering a control flow instruction and the GPU must execute the instructions on each path serially up to some point where the threads should reconverge and resume parallel execution. In 1982, graphic processors such as the CHAP [43] addressed this problem by implementing an instruction set consisting of if-then and while constructs that produced a structured control flow with implicit points of reconvergence. Today, some of the most modern GPUs support unstructured control flow based on branches and labels and leave to the compiler the generation of explicit reconverge instructions (i.e., join) [44].

Figure 4.1 shows an example of these two approaches. In this example, structured control flow results in both static and dynamic code duplication. In other cases, structured control flow can introduce higher register pressure. Alternatively, unstructured control flow avoids static code duplication and can potentially prevent dynamic code duplication.
4.1 Hammock Graphs

Caracal uses a control flow analysis algorithm based on hammock graphs [41] to transform unstructured programs into structured programs. Hammock graphs, as shown the example in Figure 4.2, are single-entry, single-exit regions in a control flow graph, and can be defined as follows:

**Definition 1** Let $CFG = (N, E, n_0, n_e)$ be a control flow graph and consider a subgraph $H = (N', E', u, v)$ with a distinguished node $u$ in $H$ and a distinguished node $v$ not in $H$ such that 1) all edges from $(CFG - H)$ to $H$ go to $u$, 2) all edges from $H$ to $(CFG - H)$ go to $v$. $H$ is called a **hammock graph**. Node $u$ is called the entry node and $v$ is called the exit node.

Figure 4.2 shows a hammock graph with node $BB2$ designated as the entry node and node $BB6$ as the exit node.

CUDA C has the characteristics of most imperative languages where control flow is enforced by block-oriented constructs, such as *if-then-else* and *while* loops. Also, the use of *goto* statements is restricted in CUDA C by the compiler. Therefore, current
CHAPTER 4. UNSTRUCTURED CONTROL FLOW

Figure 4.2: Hammock Graph $HG$ with distinguished node $BB_2$ as the entry node and distinguished node $BB_6$ as the exit node.

CUDA C kernels usually yield well-structured programs. A well-structured program is formally defined as:

**Definition 2** If all control graphs of a control flow graph are structured, i.e., are hammocks, then the control flow graph is called complete hammock and the program is well-structured.

However, there are some kernels that can lead to unstructured programs, such as those with compound if statements and early function returns. In the case where the kernel produces an unstructured program, Caracal uses a series of transformations with limited code expansion and a few additional local variables. The algorithm is well suited for GPU applications since it limits the amount of additional register pressure required and consists of three transformations: 1) a cut transformation, 2) a backward transformation, and 3) a forward transformation.
for $l$ = the innermost to the outermost loop do
    $n$ = the number of exit branches in loop $l$

Before loop $l$, insert:
1. $fp_i$ = false ($1 \leq i \leq n$)
   for all outgoing branch $i$ with guard condition $B_i(1 \leq i \leq n)$ do
       replace branch $i$ to target $t_i$ by:
       2. if ($B_i$) then $fp_i$ = true; exit;
   end for

After loop $l$ insert:
3. if ($fp_i$) then goto $t_i (1 \leq i \leq n)$
end for

Figure 4.3: Cut transformation algorithm.

4.2 Cut Transformation

The cut transformation is used in loops with outgoing branches, such as those found in kernels with early return statements. It replaces the outgoing branch with two branches: the first branch uses a new register to flag the exit condition and jumps to the end of the loop; the second branch comes right after the loop and jumps to the original target depending on the value of the flag. Figure 4.4 shows the cut transformation. The algorithm implemented in Caracal is presented in Figure 4.3.

4.3 Backward Transformation

The backward transformation is used in loops with incoming branches. This transformation is infrequently applied in CUDA C kernels where goto statements are not allowed, though can appear due to compiler optimizations. This transformation unrolls the loop body once and transforms the repeat loop into a while loop. Then, it redirects all the incoming branches to the unrolled block, as shown in Figure 4.6. Caracal implements the backward transformation using the algorithm in Figure 4.5.
for all backward branches \( b \) do
\[ MHG_b = \text{minimal hammock graph of } b \]
\[ \text{loop} = \text{cyclic region controlled by } b \]
\[ B_b = \text{guard condition of branch } b \]
\[ \text{if} \text{ there exist incoming branches into } \text{loop} \text{ then} \]
\[ CG_b \text{ is replaced by:} \]
\[ 1. \text{loop body} \]
\[ 2. \text{while } (B_b) \{\text{loop body}\} \text{ enddo} \]
end if
end for
4.4 Forward Transformation

After all the backward branches have been replaced by structured constructs using the cut and backward transformations, there can still exist unstructured forward branches. The forward transformation replaces these forward branches using tail duplication of the Minimal Hammock Graph, as shown in Figure 4.8. The algorithm implemented in Caracal is shown in Figure 4.7.

4.5 Experimental Results

Next, we analyze the difference between structured and unstructured control flow in terms of register file pressure and number of instructions cloned. The experiments evaluate the overhead of building the control tree and transforming an unstructured program using hammock graphs. The results are shown in Table 4.1. We identified the benchmarks from the CUDA SDK that require Caracal to apply the Cut, Backward,
while there exists an initial forward branch $i_f$ do
  $MHG_{i_f} =$ minimal hammock graph of $i_f$ with end node $i_e$
  $J =$ target of the branch $i_f$
  $B =$ guard condition of branch $i_f$
  if $CG_{i_f}$ interacts with other forward branches then
    $true\_part =$ the shared statements $= (MHG_{i_f} - CG_{i_f} - \{i_e\}) \cup J$
    $false\_part = MHG_{i_f} - \{i_f, i_e\}$
    $MHG_{i_f}$ is replaced by:
    1. if ($B$) then $true\_part$ else $false\_part$ endif
  end if
end while

Figure 4.7: Forward transformation algorithm.

Figure 4.8: Forward transformation graph.
Table 4.1: Unstructured Control Flow Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Ins</th>
<th>ICloned</th>
<th>Blocks</th>
<th>BCloned</th>
<th>Registers</th>
<th>Cut</th>
<th>Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitonic merge</td>
<td>512</td>
<td>32</td>
<td>64</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>particles</td>
<td>93</td>
<td>17</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>2040</td>
<td>96</td>
<td>183</td>
<td>72</td>
<td>24</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>mandelbrot2</td>
<td>2404</td>
<td>148</td>
<td>269</td>
<td>57</td>
<td>11</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>eigen values</td>
<td>1853</td>
<td>76</td>
<td>323</td>
<td>27</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>mri-fhd</td>
<td>1946</td>
<td>8</td>
<td>268</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>tpacf</td>
<td>482</td>
<td>26</td>
<td>68</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hotspot</td>
<td>237</td>
<td>4</td>
<td>28</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>nummergpu</td>
<td>237</td>
<td>4</td>
<td>28</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>leukocyte</td>
<td>637</td>
<td>4</td>
<td>61</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

or Forward transformations described in the previous Section. For these benchmarks, we measure the number of instructions that we need to clone, the number of basic blocks that are duplicated and the extra registers required.

As shown in Table 4.1, some of the benchmarks (i.e., Mandelbrot) can have pathological situations where additional registers would need to be allocated. Further investigation showed that this happens especially when we encounter nested loops with early exits (i.e., a return statement inside a triply-nested loop). Although the extra flags needed in these cases could be packed in fewer vector registers, we would still need to incur the penalty of packing and unpacking flags according to the nesting level inside the loops.

### 4.6 Summary

In this chapter we discussed the differences between unstructured and structured control flow. We also discussed the algorithm that Caracal implements to translate a PTX program with unstructured control flow into IL using only structured constructs. There are three transformations that Caracal implements: the cut transformation, the
backward transformation, and the forward transformation.

We evaluated the implications of running these transformations on some applications in terms of code duplication and register pressure. We found that for some applications (e.g., Mandelbrot) with nested loops, the cut transformation increased the register pressure by 24 additional registers. We found that although the backward transformation is seldom found in applications, in the future, as the programming language evolves and the compilers implement new optimizations, it is possible that it becomes important.

In the next Chapter, we explore the differences between parallel processors architectures that impacted the design of Caracal when we were optimizing code. We evaluate this issue for two compiler optimizations: vectorization and loop unrolling.
Chapter 5

Compiler Optimizations

To effectively utilize the large number of cores provided on any GPU, the compiler must be able to efficiently utilize the available registers on these devices. Most of these architectures are characterized by a register file shared among multiple threads which makes the job of the compiler even harder. We explored how to reduce register pressure in Caracal. In CUDA, the Open64 compiler generates PTX code, which is a virtual instruction set used to interface to the NVIDIA GPU device driver. Caracal optimizes the original PTX instructions with the goal of reducing the register pressure and evaluating the impact the front-end compiler can have on performance.

PTX has unlimited virtual registers, and CUDA programs rely on the driver to assign registers appropriately. However, the register allocator is limited by compile-time constraints, and sometimes uses more registers than is optimal. The compiler already performs a range of low-level optimizations such as rematerialization and vector coalescing, which are more appropriate to apply in the first compilation pass due to the lack of compile time restrictions.

In this portion of the dissertation, we investigate how the IR is impacted when
properly implementing optimizations. Our first focus is to reduce register pressure in
the PTX instructions. We attempt to reduce the maximum number of registers live
at any particular point in the instruction stream as a way to reduce register pressure
in the final binary.

The benefits of producing lower register pressure include:

• generating more thread blocks per core, which increases the thread occupancy
  of the GPU,

• producing larger thread blocks, since each thread will consume fewer registers, and

• generating fewer register spills which are very costly due to the limited band-
  width to off-chip memory.

First, we present the results from two optimizations that we explore during the
preliminary phases of our research: register rematerialization and instruction schedul-
ing. Even though the performance improvements we achieved by implementing these
two optimization were marginal, they still demonstrate the importance of choosing
the right set of optimizations according to the microarchitecture of the target GPU.

Second, we explored another couple of optimizations with the architecture of the
AMD GPU in mind: vectorization and loop unrolling. The results are presented at
the end of this Chapter. All of these optimizations are aimed at reducing register
pressure and, therefore, increase hardware occupancy.
5.1 Register Rematerialization

The goal of register rematerialization is to avoid registers spills to memory by recomputing the value stored in the register. Another way to use register rematerialization is to do *register live splitting* [45]. We used register rematerialization as the mechanism to reduce register pressure in the program. In this case, Caracal identifies registers whose value come from shared memory and inserts an additional load instruction to recreate the value and split the lifetime. We also perform this optimization for load instructions from the parameter and special register state spaces (i.e., kernel arguments and CUDA-specific kernel variables such as threadId and blockIdx). In Figure 5.1, for example, Caracal will identify register $r1$ to be a good candidate for rematerialization and will insert a load instruction to recreate its value.
5.2 Instruction Scheduling

We implemented a backward list scheduler [46] in Caracal that reorders instructions within a basic block in order to reduce the number of live registers. We start by doing a liveness analysis and building a data dependence graph. The scheduler iterates through the ready list, evaluating a cost function for each instruction. The cost function considers the number of live registers and the \textit{defs/uses} associated with each instruction. Once the cost function completes, the scheduler selects the instruction with the lowest cost, adds it to the schedule, updates the ready list, and repeats this process. Figure 5.2 illustrates this algorithm.

We evaluated our proposed set of optimizations on an ATI Radeon HD5870 GPU and a 64-bit Intel Core2 CPU running Linux (Ubuntu 12.10). Our target applications are taken from the CUDA SDK 3.2, CUBLAS [47], and PhysX [48] benchmark suites. The results were collected using the AMD Kernel Analyzer.

5.2.1 NVIDIA SDK 3.2

We compiled the applications included in the NVIDIA SDK 3.2. This program suite includes 52 CUDA applications containing a total of 748 kernels. We evaluated the ability to produce lower register pressure in the final binary by making changes at the PTX level. We found that the correlation coefficient between PTX register pressure and the binary register pressure is 0.94. In our experiments, 294 (39\%) kernels showed lower register pressure, 364 (49\%) experienced no change, and only 90 (12\%) had higher register pressure due to our optimizations. Figure 5.3 shows the register pressure before and after our optimizations for the kernels that experienced the largest improvements.
1: Compute_Liveness();
2: for (bb = First_BB() ; bb != NULL ; bb = Next_BB()) do
3:   Compute_Dep_Graph(bb);
4:   Bb_Linfo info = Liveness_Info(bb);
5:   \( \vec{sched} = NULL; \)
6:   VECTOR ready = NULL;
7:   TN_SET live = info.live_out;
8: for (op = First_OP(bb) ; op != NULL ; op = Next_OP(bb)) do
9:   if op.succs == NULL then
10:      ready += op;
11: end if
12: end for
13: while ready != NULL do
14:   int mincost = MAXINT;
15:   for each \( op \in ready \) do
16:      int cost = 0;
17:      if op.uses \notin live then
18:         cost++;
19:     end if
20:     if op.defs \in live then
21:         cost–;
22:     end if
23:     if cost < mincost then
24:        best_op = op;
25:        mincost = cost;
26:     end if
27:   end for
28:   sched = best_op + sched
29:   live += best_op.uses;
30:   live -= best_op.defs;
31: for each \( op \in best_op.pred \) do
32:   if op.succs == NULL then
33:      ready += op;
34:   end if
35: end for
36: end while
37: end for

Figure 5.2: Instruction scheduling algorithm.
For the one-dimensional Haar Wavelet Decomposition (dwtHaar1D), our optimizations reduce the binary register pressure from 11 to 9 registers, allowing one extra block to be allocated per Streaming Multiprocessor (SM). We ran the optimized version of this benchmark with a 16M input signal and the performance improved by 4%, as shown in Figure 5.4.

5.2.2 DTRSM

Double-precision TRiangular Solve Multiple (DTRSM) is part of the CUBLAS library and it is used to solve a triangular system of equations. DTRSM is helped by the rematerializations of parameters. The optimized code uses 2 fewer registers for all of the inputs sets, providing a 5% improvement in overall execution time as shown in Figure 5.4.
5.2.3 SGEMM

We tested a variant of the Single-precision General Matrix Multiplication (SGEMM) application that calculates \( A \times B^T + C \), where \( A \) was 416x256, \( B \) was 256x416, and \( C \) was 416x416. The improvement in execution time, as shown in Figure 5.4, was 21%. The reason for the large improvement in this application was due to the reduction of register spills, eliminating 2.9M dynamic loads and 1.5M dynamic stores.

5.2.4 PhysX

We applied our optimizations to a physics simulation package we are using in a surgery simulation project. This code is used to compute the dynamics of collisions, similar to the PhysX code that is used in games. Our results varied depending on the input used. For one input in particular, there were 3 less spill locations, while for another input we saw an increase of 12 more spill locations. The overall difference in execution time was in the noise range (0.4%).

In general, we found that the effect of doing rematerialization of parameters was
better than doing rematerialization of special registers. We attribute this to the fact that special register rematerialization adds an extra instruction (a *convert* instruction), whereas parameter rematerialization can be folded into an instruction, thus not costing anything. This is due to parameters being actually allocated in shared memory and the NVIDIA instruction set having the ability to directly access shared memory as one of the operands, in place of a register. This is not completely orthogonal, in that not all register operands can be shared memory operands, but the most common instructions can do this. So the *ld.param* instruction usually gets folded into the *use* instruction and, therefore, it becomes free.

### 5.3 Vectorization and Loop Unrolling

Our experiments with register rematerialization and instruction scheduling showed that in order to obtain larger performance improvements we would need to employ optimizations more targeted to AMD’s GPU architecture. Other researchers have shown that VLIW architectures can benefit from vectorization passes that attempt to increase the occupancy of the pipeline. The goal of this optimization is to make better use of the vector registers in the processor. It is important to account for alignment and coalescing when implementing this optimization for GPUs.

With this in mind, we implemented two optimizations in Caracal targeted to AMD’s GPUs, namely vectorization and loop unrolling. We chose to implement loop unrolling as well because it works well with vectorization by creating bigger basic blocks with more opportunities for vector instructions. Our results confirmed this theory.

Our vectorization pass is implemented in Caracal by iterating through the Control
Tree and identifying basic blocks with potential for improvements. These basic blocks turned out to be mainly loops in applications that expose data-parallelism. The vectorization pass works as presented in Figure 5.5 by implementing the following steps:

1. Data dependence analysis: We build the data-dependence graph of the kernel including information regarding the live-in and live-out registers. We use this information to preserve the data dependencies after running the vectorization pass.

2. Instruction scheduling: We try to group together instructions of the same type within each basic block to create opportunities for the vectorization pass to reduce them into a single instruction.

3. Packing/Unpacking: We modify load and store operations to pack and unpack values coming from memory into a single register. This also has the advantage of creating more opportunities for the memory controller to coalesce instructions into a single transaction. We take into account alignment restrictions.

4. Vectorization: Finally, we translate multiple scalar operations into a single vector operation that uses more than one channel.

We also evaluate the effectiveness of the vectorization pass when combined with loop unrolling. We perform loop unrolling right before vectorization and evaluate the results. The steps to perform loop unrolling can be described as:

1. Data dependence analysis: We take into account loop-carried dependencies that could prevent us from unrolling the loop. Also, we target loop whose trip count
Figure 5.5: Vectorization Pass

<table>
<thead>
<tr>
<th>a) original code</th>
<th>b) instruction scheduling</th>
<th>c) vectorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>uav_raw_load r11.x, r5</td>
<td>uav_raw_load r11.x, r5</td>
<td>uav_raw_load r11.x, r5</td>
</tr>
<tr>
<td>uav_raw_load r12.x, r7</td>
<td>uav_raw_load r12.x, r7</td>
<td>uav_raw_load r12.x, r7</td>
</tr>
<tr>
<td>add r13.x, r11.x, r12.x</td>
<td>uav_raw_load r14.x, r55</td>
<td>add r13.x, r11.x, r12.x</td>
</tr>
<tr>
<td>uav_raw_store r9, r13.x</td>
<td>uav_raw_load r15.x, r56</td>
<td>add r16.x, r14.x, r15.x</td>
</tr>
<tr>
<td></td>
<td>uav_raw_load r16.x, r15.x</td>
<td>uav_raw_store r9, r13.x</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>uav_raw_load r14.x, r55.x</td>
<td>uav_raw_load r14.x, r55</td>
<td>uav_raw_load r14.x, r55.x</td>
</tr>
<tr>
<td>uav_raw_load r15.x, r56.x</td>
<td>uav_raw_load r15.x, r56</td>
<td>uav_raw_load r15.x, r56.x</td>
</tr>
<tr>
<td>add r16.x, r14.x, r15.x</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>uav_raw_store r9, r13.x</td>
<td>uav_raw_store r9, r13.x</td>
<td>uav_raw_store r9, r13.x</td>
</tr>
<tr>
<td>uav_raw_store r57, r16.x</td>
<td>uav_raw_store r57, r16.x</td>
<td>uav_raw_store r57, r16.x</td>
</tr>
</tbody>
</table>

is known at compile time and can be expressed as a multiple of four (the number of channels in AMD’s GPU architecture).

2. Loop unrolling: We unroll the loop by a factor of 4 to create bigger basic blocks and, therefore, more opportunities for vectorization.

3. Register renaming: We perform register renaming right after unrolling the loop to break output dependencies (write-after-write) created by the previous step.

4. Vectorization: We run a vectorization pass again to evaluate the effectiveness of this optimization when used together with loop unrolling.

### 5.4 Experimental Results

We evaluated the vectorization pass by itself and also with loop unrolling on an ATI Radeon HD5870 GPU. The results were collected using the AMD Kernel Analyzer. We evaluated these optimizations on a series of microbenchmarks, each one using a
different functional unit in the GPU core. The microbenchmark consist of a GPU kernel code together with timing code. Each kernel instruction is executed 10 thousand times. Each kernel is invoked 100 times to avoid the effects of cold caches. The timing measurements are taken by invoking the Linux kernel system call `gettimeofday`. The results are presented in Table 5.1.

The results show that loop unrolling by itself can have a negative impact (-60%) in performance for most of the benchmarks. This is due to the increase in register pressure when we unroll the loop. In this case, we are unrolling the loop by a factor of 4 which means the register pressure can increase by four times. We observe this for 3 of the benchmarks in Table 5.1 (i.e. iadd, add, and mad). However, for one of the benchmarks (i.e. sqrt), loop unrolling shows an improvement. This is due to `sqrt` only having one operand and, therefore, less register pressure. For `sqrt`, the backend compiler can find enough hardware registers and avoid spilling into memory.

The biggest improvement for all of the benchmarks (up to 90%) comes from applying loop unrolling followed by the vectorization pass. Loop unrolling creates more opportunities for vectorization and, therefore, we can reduce the number of instruction and improve hardware occupancy. Moreover, by applying vectorization successfully we can also coalesce memory accesses and use less bandwidth.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No Opt</th>
<th>Vectorization</th>
<th>Unrolling</th>
<th>Vector + Unroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT (iadd)</td>
<td>29 ms</td>
<td>27 ms</td>
<td>5.6%</td>
<td>73 ms</td>
</tr>
<tr>
<td>SP (add)</td>
<td>29 ms</td>
<td>27 ms</td>
<td>5.6%</td>
<td>73 ms</td>
</tr>
<tr>
<td>SP (mad)</td>
<td>31 ms</td>
<td>26 ms</td>
<td>21%</td>
<td>78 ms</td>
</tr>
<tr>
<td>SFU (sqrt)</td>
<td>21 ms</td>
<td>21 ms</td>
<td>0%</td>
<td>15 ms</td>
</tr>
</tbody>
</table>

Table 5.1: Vectorization - Execution time
5.5 Summary

In this Chapter, we explored compiler optimizations aimed primarily at reducing register pressure.

First, we presented two traditional optimizations implemented by compilers for both CPUs and GPUs: register rematerialization and instruction scheduling. We found that the performance improvements were minimal due to the complexities of the GPU architecture.

Second, we implemented and evaluated two optimizations especially tuned to the AMD GPU architecture: vectorization and loop unrolling. The results showed that the performance improvements were much better in this case with speedups up to 90%.
The use of accelerators to improve the performance of scientific applications has been a research topic for many years. Recently, Graphics Processing Units (GPUs) have become the accelerator of choice in a number of applications mainly because of their availability in current systems and the new programming models introduced like CUDA C and OpenCL. However, in order to achieve levels of performance in the order of exaFLOPs, we are going to need new architectures and compiler techniques.

One of the main challenges for new compilers is the issue of portability. Developers expect their applications to run fast across different hardware vendors without having to spend too much time tuning them. This is not an easy challenge considering the differences among architectures and the intricacies of each instruction set.

This thesis contributes to this goal by designing an IR that is amenable to different instructions sets and runtime systems. We designed this IR taking into consideration the differences in the handling of control flow. Some implementations can support unstructured control flow based on branches and labels while others are based on structured control flow relying solely on if-then and while constructs. We built an
open-source framework called Caracal based on this IR and we demonstrate its capabilities by using it to translate CUDA C programs so they can run on AMD’s Compute Abstraction Layer (CAL) platform.

This thesis enhances the current state-of-the-art and makes key contributions to the following fields:

- **Runtimes for Heterogeneous Computing**: We address the issues of designing an IR and a runtime system that can be used in GPU computing for different platforms.

- **Dynamic Translation**: We create an open-source cross-platform translation framework targeted to new accelerators used in high performance computing.

- **GPU Compiler Optimizations** We explore compiler optimizations targeted to the AMD GPU architecture and evaluate their effectiveness at reducing register pressure and improving execution time.

Caracal implements mechanisms used before for disassembling low-level instructions in a novel way. It uses techniques and methods to abstract a program using higher-level representations to build an structured control flow graph using IL instructions. These techniques are based on hammock graphs: single-entry, single-exit regions of code that can be reduced by the compiler. We developed three different transformations to be used with different control flow:

- **Cut Transformation**: It works on strongly connected components in the control flow where there are jumps out of a loop. It replaces the unstructured jump with a if-then construct and allocates an additional register to use as a flag.
• **Backward Transformation:** This transformation is used on strongly connected components with jumps into a loop. We showed in our experiments that this transformation is rarely found in CUDA applications due to the lack of `goto` statements.

• **Forward Transformation:** We apply this transformation to interacting forward branches. Caracal duplicates the basic blocks in the interacting path and modifies the control flow to avoid the unstructured jumps.

To the best of our knowledge, unstructured control flow reduction in general-purpose GPU computing is an area that had not been explored before. We also look at the implications in terms of performance, register pressure, and other metrics by drawing a comparison between the IRs used by NVIDIA and AMD.

This thesis explicitly defines a methodology for translating runtime environments used in heterogeneous computing. This methodology is available to anyone interested in addressing the important challenge of portability and investigating other hardware targets. We identified the most important features in the CUDA runtime environment to take into consideration: 1) control flow reduction; 2) address space translation; 3) synchronization mechanisms; 4) memory model compatibility; 5) register types; and 6) transcendental operations. This features where discussed in detail in Chapter 3.

We implemented Caracal as a drop-in replacement library that is part of the Ocelot framework [36]. This open-source framework implements each one of the entry points of the CUDA runtime. Therefore, it can linked against any CUDA application. We strove to make the process as simple as possible for future researchers interested in enhancing our work. We extensively tested Caracal for functional correctness by examining the translated kernels and by comparing the final results with those performed by the CPU.
Different from other systems, Caracal can be used with pre-compiled binaries. This is especially important for developers who do not have access to source code, as is the case with certain libraries. Caracal requires no source code modifications and the device selection (GPU or CPU) can be done at run-time and can be different between different kernel invocations.

Our open-source dynamic translation framework for GPUs creates multiple avenues for future research in the area of heterogeneous computing. The runtime translation could be enhanced to make better used of the hardware units in AMD GPUs like constant caches and texture memory. New features of PTX, like unified global memory space, could be incorporated into the IR translation. We believe that by implementing some of these improvements, Caracal could perform comparably to native OpenCL applications.

The abstraction of kernels using a control tree, including the algorithm to reduce an unstructured control flow graph using an IR with only structured constructs, can be used with similar GPUs that require the programs to be expressed with this restriction. This is an interesting area of research since it brings up challenging questions of portability – one of the main concerns with emerging programming models.

Caracal builds the control tree of the program by following a post-order traversal of the depth-first spanning tree, identifying regions that correspond to the structured constructs and reducing them into abstract nodes. At the same time, it tries to find single-entry, single-exit regions of code called hammock graphs. The algorithm that Caracal implements has \( O(n^2) \) complexity. This is one of the main overheads that we found when comparing Caracal to OpenCL. An area of improvement in Caracal would be the implementation of more efficient algorithms to reduce unstructured control flow graphs used with GPUs.
Another front for research is the exploration of different programming models for heterogeneous systems. Since Caracal works by translating applications at a very low-level, it is suitable to be used with other programming models that include compilers that can emit PTX code. Caracal could be easily be expanded to support new instructions proposed by new programming models (e.g., context-aware barriers).

In the area of optimizations, Caracal can be used to explore target-dependent optimizations for AMD GPUs as well as other accelerators. By creating an abstract representation of the kernel, we have created a tool that can be easily used for evaluating other compiler optimizations based on IR analysis. Alternatively, Caracal would be an excellent tool to implement profile-guided optimizations based on dynamic information collected at runtime. It would be of great contribution to have a tool that could collect information dynamically and used it to tune optimization parameters like the loop unrolling factor and the vector width.

We investigated four different compiler optimizations with the common goal of reducing register pressure. We evaluated these optimization on a set of micro benchmarks. According to our results, architecture-aware optimizations like vectorization and loop unrolling showed promising results. As future work, it would be interesting to the research community to evaluate these optimizations on benchmarks that showed high register pressure (e.g., Mandelbrot) and combine them with the techniques we develop for control flow reduction (especially the cut transformation since it increases register pressure).
Bibliography


[15] PTX: Parallel Thread Execution ISA.


the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP ’09, pages 101–110, New York, NY, USA, 2009. ACM.


[34] The CUDA Compiler Driver NVCC.


[52] Rodrigo Domínguez, David Kaeli, John Cavazos, and Mike Murphy. Improving the open64 backend for gpus, October 2009. NVIDIA GPU Technology Conference.


[66] Intel array building blocks.