Design and Evaluation of Register Allocation on GPUs

A Thesis Presented
by

Charu Kalra

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

November 2015
Abstract

The rapid adoption of Graphics Processing Unit (GPU) computing has led to the development of workloads which can leverage the massive parallelism offered by GPUs. The use of GPUs is no longer limited to graphics, but has been extended to support compute-intensive applications from various scientific and commercial domains. Significant effort and expertise are required to optimize applications targeted for a specific GPU architecture. One of the keys to improving the performance of the applications is to utilize the available hardware resources efficiently. The architectural complexity of the GPU makes it challenging to optimize performance at both the source level and the compiler level.

In this thesis, we develop a transformation pass (i.e., a register allocator) that optimizes both vector and scalar register usage on GPUs. We introduce an open-source compiler framework, Multi2C, which converts OpenCL kernel into AMD Southern Islands binaries. The register allocator is integrated as a part of the Multi2C optimization framework. Register allocation for each class of registers is run as a separate pass in the compiler. We also highlight the challenges faced when performing register allocation targeting a GPU, including issues with thread divergence and proper handling of contiguous register series. We evaluate the performance of our allocator for applications taken from the AMD APP SDK. We compare our results with a baseline design which performs register assignment without any backtracking. The proposed allocator reduces register usage by 22.5% for VGPRs and by 44.2% for SGPRs, as compared to the baseline. By reducing the number of registers used, average wavefront occupancy increases by 73%, which leads to an average speed up of 5.4% over the baseline.
Acknowledgements

I first want to thank my wonderful family and friends for their encouragement and support during the course of this work.

I want to express my gratitude to Chris Reeve, Norm Rubin, Xiaoqing Wu, and Mike Bedy for their guidance and expertise on register allocation and compilers in general, which significantly improved the quality of this work.

I also thank Xiang Gong, without his help none of the results in this thesis would have been possible, and Yash Ukidave, for his valuable feedback regarding this work.

Finally, I want to thank my advisor, Professor David Kaeli, for his time and efforts.
Contents

List of Figures vi

List of Tables vii

1 Introduction 1
  1.1 Challenges for Compiler Optimization on GPUs 2
  1.2 Optimizing Register Usage 3
  1.3 Contributions 3
  1.4 Organization of this Thesis 4

2 Background 5
  2.1 Overview of a Compiler 5
  2.2 Intermediate Representation 6
    2.2.1 Types of IR 6
    2.2.2 Role of IR 7
    2.2.3 LLVM IR 7
  2.3 AMD Radeon 7970 Architecture 8
  2.4 OpenCL programming model 8
    2.4.1 Platform Model 9
    2.4.2 Memory Model 9
  2.5 Southern Islands ISA 10
    2.5.1 Vector-Scalar architecture 10
    2.5.2 Control Flow and Branch Divergence 10

3 Related Work 12
  3.1 Optimizations for GPU applications 12
  3.2 Compiler-based GPU Optimizations 13
  3.3 Cross-Compilation 13
  3.4 Intermediate Representations 14
  3.5 Optimizing Register Usage 15

4 Multi2C: A GPU Kernel Compiler 16
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The OpenCL platform model</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Structure of a three-phase compiler</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Compilation support from N source languages to M targets</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>The Radeon 7970 architecture</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>The OpenCL platform and memory model</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Structure of Multi2C compiler</td>
<td>16</td>
</tr>
<tr>
<td>5.1</td>
<td>Thread Divergence: (a) Sample pseudo code (b) Execution path when t0 and t1 are</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>vector temporaries (b) Execution path when t0 and t1 are scalar temporaries</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>A weighted interference graph</td>
<td>24</td>
</tr>
<tr>
<td>7.1</td>
<td>The change in register usage. Register usage is lower is better metric</td>
<td>29</td>
</tr>
<tr>
<td>7.2</td>
<td>Figure showing change in SIMD occupancy</td>
<td>30</td>
</tr>
<tr>
<td>7.3</td>
<td>Speed up, as a result of the change in occupancy. Speed up is higher is better metric</td>
<td>31</td>
</tr>
<tr>
<td>8.1</td>
<td>(a) A CFG in non-SSA form (b) CFG converted to SSA form by inserting phi instruc</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>tions</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

6.1 The device configuration of the Southern Islands GPU ........................................ 27
6.2 Benchmarks from AMD APP SDK ................................................................. 28
Chapter 1

Introduction

GPUs have become the most widely used accelerators available in the market today. The use of GPUs is no longer limited to graphics, but has been extended to support a broad range of compute-based applications. Effective exploitation of the massively parallel resources of a GPU can result in large speedups for applications in a number of computational domains including scientific computing, biomedical imaging, global positioning system, and signal processing applications [14, 20, 32].

The popularity of GPUs has led to extensive research in developing workloads for GPUs and optimizing them to enhance their performance. CUDA and OpenCL are commonly used parallel programming frameworks to create applications that can execute on GPUs [17, 27]. CUDA is currently supported on NVIDIA GPUs and multicore CPUs [38], whereas OpenCL is supported on different heterogeneous devices including GPUs, multi-core CPUs and FPGAs. Many techniques have been developed to exploit the potential parallelism in applications by understanding their interaction with the GPU architecture [26, 35, 36]. Several compiler frameworks have also been developed to optimize applications without programmer intervention [7, 24, 40].

The OpenCL model is based on a Compute Device that consists of Compute Units (CUs) with Processing Elements (PEs), as shown in Figure 1.1. A host program launches kernels with workitems over an index space. Workitems execute on a processing core of a compute unit on the device. These workitems are further grouped into workgroups. A set of workitems within a single workgroup, which executes a single instruction in SIMD manner on the compute unit is known as a wavefront. We describe the OpenCL platform model in further detail in Chapter 2.
1.1 Challenges for Compiler Optimization on GPUs

- **Architectural Complexity of GPUs:** In the past, GPUs were primarily used to render graphics, but now their capabilities are being harnessed to accelerate computational workloads in various scientific and commercial domains. Their popularity has led to the development of more sophisticated compilers to leverage their massive GPU resources more efficiently.

Architecturally, a GPU is composed of hundreds of cores that can handle thousands of concurrent threads. In contrast, the CPU can run many threads, but the number of cores (less than 16) limit the degree of concurrency. Therefore, GPUs are better at Thread Level Parallelism (TLP), whereas CPUs are better at extracting Instruction Level Parallelism (ILP).

Unlike GPUs, each thread in a CPU can follow its own path. Furthermore, branches can be efficiently executed because CPUs have sophisticated control flow prediction. Programs that lack inherent parallelism may actually execute faster on CPUs. The compilers targeted for GPUs incorporate many sophisticated optimizations to expose as much parallelism as can be extracted from an application. Some standard techniques that are used in CPU compilers are closely related to GPU optimizations, while some are specific to the GPU architecture. Presented with a massive number of processing elements, a unique memory hierarchy, and a rudimentary control flow mechanism, code optimization in a GPU compiler can be challenging.

- **Lack of open source tools:** Compilation on a GPU differs greatly from the environment on a CPU. For applications to run on GPUs, hardware vendors provide users with the necessary
tools, libraries and extensions to facilitate the compilation process specific to their GPUs. The frontend compiles the source kernel into an IR, while the backend is usually embedded in the device driver, which makes it challenging to develop optimizations. Unfortunately, many of the resources provided by the vendors are proprietary, and not open source, therefore cannot be modified. This restricts the developers to either optimize applications at the source level or create source-to-source compilers that produce optimized versions of their source code. This problem highlights the need for a new class of open-source tools that allow researchers to develop new compiler optimizations, working much closer to the GPU architecture. We therefore introduce Multi2C, an open-source compiler framework that translates OpenCL kernels into AMD Southern Islands binaries. The structure of Multi2C is described in more detail in Chapter 4.

1.2 Optimizing Register Usage

Registers are almost always a scarce resource on a processor. In Reduced Instruction Set Computer (RISC) systems, all operations other than data movement operate on register contents. Register-to-register operations are significantly faster than memory operations. Therefore, it is crucial to utilize the registers available on the GPU efficiently.

The benefits of producing lower register usage include:

- generating more work-groups per CU, which increases the occupancy of the GPU,
- producing larger work-groups, since each work-item will consume fewer registers, and
- generating fewer register spills which are very costly due to the limited band-width to off-chip memory.

1.3 Contributions

The contribution of this thesis are as follows:

- We design and implement algorithms for register allocation in the Multi2C compilation framework.
CHAPTER 1. INTRODUCTION

- We perform register allocation for both vector and scalar registers in non-SSA form, highlighting the challenges faced when performing register allocation, including issues such as thread divergence and handling contiguous register series.

- We evaluate the execution performance and register usage of nine applications from the AMD APP SDK, and consider the impact of register usage on SIMD occupancy.

1.4 Organization of this Thesis

The following chapters present the topics covered in this thesis. In Chapter 2, we cover compiler basics, the role of an Intermediate Representation (IR), and the targeted GPU architecture. Chapter 3 describes related work on compiler optimization on GPUs. Chapter 4 describes the Multi2C compilation framework in detail. Chapter 5 discusses the implementation details of our transformation pass. We describe the evaluation platform and applications used for our analyses in Chapter 6. Chapter 7 presents results from our study. Chapter 8 discusses register allocation in SSA form on GPUs and Chapter 9 provides conclusions and directions for future work.
Chapter 2

Background

In this chapter, we will provide an overview on compiler basics and LLVM intermediate code representation. The target processor of our compiler is the AMD Radeon 7970, which belongs to the Southern Islands family of GPU. We will provide an overview of its architecture, and mapping of the OpenCL model onto a Southern Islands GPU.

2.1 Overview of a Compiler

A compiler is a software program that translates source code written in a programming language (the source language) into another computer language (usually the machine code). A program that translates between high-level languages is usually called a source-to-source compiler. A compiler may be comprised of two or more phases, referred to as the frontend, the middle-end, and the backend. In some simpler compilers, the middle-end may be absent. A two phase compiler is comprised of only a frontend and a backend. The frontend of the compiler does lexical analysis, parsing, semantic analysis, and translation to the intermediate representation or IR. The middle-end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors. The backend performs optimizations on the IR and code generation for a particular processor. The structure of a three-phase compiler is shown in Figure 2.1.
CHAPTER 2. BACKGROUND

Figure 2.1: Structure of a three-phase compiler.

2.2 Intermediate Representation

Intermediate representations (IRs) are low level, machine independent, languages that compilers use to analyze and optimize code in a retargetable way. IRs allow compilers that can target multiple ISAs to maintain a single set of optimizations, rather than multiple optimizations duplicated for each supported ISA. As an example, the intermediate representation of the LLVM compiler infrastructure is robust from both the language perspective in the frontend of the compiler, and the ISA perspective in the backend of the compiler \[22\]. Consequently, LLVM can support many languages (C, Objective C, C++, OpenCL, Python, Ruby, Haskell, etc.) and ISAs (x86, ARM, MIPS, PowerPC, SPARC, etc.) with a mature set of analyses and optimizations developed in terms of the IR.

2.2.1 Types of IR

IRs broadly fall into three structural categories \[12\]:

- **Graphical IRs** encode the compiler’s knowledge in the form of a graph. All algorithms are expressed in terms of nodes, edges, trees, or lists. An example of graphical IR is an Abstract Syntax Tree (AST).

- **Linear IRs** resemble assembly code for some abstract machine. The algorithms iterate over simple, linear, sequences of instructions. LLVM IR is an example of a linear IR.

- **Hybrid IRs** are a combination of both linear and graphical IRs. A common hybrid representation uses a linear IR to represent blocks of straight code and a graph to represent the control flow among those blocks. A Control Flow Graph (CFG) is an example of a hybrid IR.
CHAPTER 2. BACKGROUND

Figure 2.2: Compilation support from N source languages to M targets.

2.2.2 Role of IR

IRs are used by the compilers for two purposes:

- **Portability**: Without an IR, compiling N different source languages to M different machines would require N x M compilers, as shown in Figure 2.2a. A portable compiler translates the source language into an IR, that then translates the IR into machine language, as shown in Figure 2.2b. This way only N + M compilers have to be built [15].

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

2.2.3 LLVM IR

LLVM IR is a RISC-like instruction set, but without any notion of the source language constructs, such as classes, inheritance and other semantics [22]. It has no knowledge about machine specific details such as processor pipeline, the number of registers, etc. LLVM is a load/store architecture: programs transfer values between registers and memory solely via load and store operations using typed pointers. The entire LLVM instruction set consists of only 31 opcodes. Most opcodes in LLVM are overloaded (for example, the add instruction can operate on integer or floating
CHAPTER 2. BACKGROUND

Figure 2.3: The Radeon 7970 architecture.

point operand types). Most instructions, including all arithmetic and logical operations, are in one of three address forms: they take one or two operands and produce a single result. LLVM uses SSA form as its primary code representation, i.e., each virtual register is written in only a single instruction, and each use of a register is dominated by its definition \[13\]. Memory locations in LLVM are not in SSA form. A detailed description of SSA form is provided in Section \[8.1\].

LLVM has received attention for many years. A variety of compiler frontends and backends have been developed to support a diverse range of high level languages and modern architectures using the LLVM code representation and transformations. The advanced and robust features offered by LLVM led us to utilize its intermediate code representation in our compiler infrastructure.

2.3 AMD Radeon 7970 Architecture

The AMD Radeon 7970 belongs to the Southern Islands family of GPUs. Compute units are the fundamental units of computation on the AMD GPU architectures. All compute units have access to a common global memory. Each compute unit contains a set of 64 vector stream cores. The compute units are organized into 4 SIMD units of 16 stream cores each. All stream cores within the compute unit have access to a common 64KB local data share (LDS). Stream cores also are given access to files of general-purpose registers. The GPU architecture is shown in Figure \[6.4\].

2.4 OpenCL programming model

OpenCL is a popular programming framework for heterogeneous computing, maintained by Khronos [17]. OpenCL terminology refers to a GPU as the device, and a CPU as the host. Next,
CHAPTER 2. BACKGROUND

2.4.1 Platform Model

The platform model for OpenCL consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more Compute Units (CUs) which are further divided into one or more Processing Elements (PEs). Computations on a device occur within the processing elements.

The programs which execute on a compute device are known as kernels. An instance of a kernel is referred to as a workitem, which executes on a processing core of a compute unit on the device. Workitems are expressed in terms of independent units of execution known as workgroups. Workgroups execute on the compute units of the device and are organized as an n-dimensional index space known as the NDRange. A set of workitems within a single workgroup, which executes a single instruction in SIMD manner on the compute unit, is known as a wavefront. Software abstractions used for passing commands from the host to the device in an OpenCL computation are known as Command Queues.

2.4.2 Memory Model

Workitem(s) executing a kernel have access to four distinct memory regions. This is highlighted in Figure 2.4: The OpenCL platform and memory model.
CHAPTER 2. BACKGROUND

- **Global Memory**: This memory region permits read/write access to all workitems in all workgroups. Workitems can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.

- **Constant Memory**: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

- **Local Memory**: A memory region local to a workgroup. This memory region can be used to allocate variables that are shared by all workitems in that workgroup. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

- **Private Memory**: A region of memory private to a workitem. Variables defined in one workitems private memory are not visible to another workitem.

2.5 Southern Islands ISA

2.5.1 Vector-Scalar architecture

The Southern Islands (SI) family has a distinctive vector-scalar design. The SI ISA contains two category of instructions: vector instructions, which function in a SIMD-manner, and scalar instructions, which only execute once per wavefront. Vector instructions are preceded by a $v$., whereas scalar instructions are preceded by a $s$. Register files are also separated into vector registers ($v_0, v_1, \ldots, v_n$) and scalar registers ($s_0, s_1, \ldots, s_n$). Vector registers are associated with the SIMD units, and hold different values for each workitem. Scalar registers are associated with the scalar unit and hold one value per wavefront. There are 256 vector general purpose registers and 104 scalar general purpose registers on AMD Radeon 7970. All registers are 32-bits, and all 64-bit instructions use two consecutive registers to store 64-bit values. Apart from the general purpose registers, there are some special scalar registers. One of the most common special register is the Vector Condition Code (VCC). It is composed of two scalar registers and is used as an Active Mask. The VCC contains one bit per workitem and is set by vector compare instructions.

2.5.2 Control Flow and Branch Divergence

Workitems are bundled together in workgroups and execute the same instruction at the same time in a SIMD manner. When a conditional branch instruction is resolved differently for
some of the workitems in a wavefront, it leads to \textit{branch divergence}. Divergence can cause a big loss in parallel execution efficiency. The SIMD engine executes machine instructions in lock-step, and branch divergence is handled by executing both paths of the branch and masking off results from inactive processing units as necessary.

The Southern Islands ISA utilizes a series of execution masks to address branch divergence. The execution mask is a 64-bit map, where each bit represents the active status of an individual workitem in the wavefront. If a workitem is labeled as inactive, the result of any arithmetic computation performed in its associated stream core is ignored, preventing it from changing the kernel state. In Southern Islands, the execution mask is a set of two consecutive special registers named EXEC. The execution mask is handled directly by software, and nested execution masks must be stored in scalar general purpose registers.
Chapter 3

Related Work

In this chapter, we describe prior relevant work that has been done to optimize GPU applications at source level. We also provide insight into some of the source-to-source compiler frameworks that have been developed to improve the execution performance and resource utilization of GPU applications.

3.1 Optimizations for GPU applications

Optimizing GPU applications is a highly challenging task. A lot of effort and expertise is required to maximize an application’s execution performance. OpenCL provides a simple, portable, programming model for application developers [25]. However, it is critical to utilize the hardware resources efficiently in order to achieve high performance. Several studies have been conducted to optimize the performance of GPU applications. An optimization space pruning technique has been proposed to find the optimal configuration of GPU applications using a Pareto Optimal Curve [36]. This work shows how to prune the configuration space by inspecting the way optimizations interact with the architecture, and by defining performance metrics to judge the performance of an optimization configuration.

The impact of classical loop unrolling optimization has been studied to identify the optimal unroll factors for GPU applications [26]. In this paper, the authors develop a static, semi-automatic, compile-time technique to select optimal unroll factors for suitable loops in CUDA programs. This technique is based on the analysis of the compiled CUDA code and estimation of relative performance of various unroll options. In addition, they design techniques for pruning the
CHAPTER 3. RELATED WORK

search space of unroll factors. Their findings report a 70% improvement in the performance of optimized versions of applications.

An experimental study on general optimization strategies for programs on a CUDA-supported GPU has also been presented [35]. This work focuses on general principles, namely producing efficient code, utilizing many threads to hide latency, and using local memory to reduce pressure on global memory bandwidth, in order to optimize applications targeting an NVIDIA GeForce 8800 GTX GPU. They evaluate speedup (versus a CPU implementation) obtained for a suite of applications, applying their proposed optimization principles.

3.2 Compiler-based GPU Optimizations

To relieve developers from the task of optimization, several efforts have been made to automate the optimizations using GPU compilers. Baskaran et al. propose a compiler framework that optimizes affine loop nests using a polyhedral model [7]. The polyhedral model is used to empirically search for the best set of loop transformation parameters, including the loop tiling sizes and unrolling factors. Although the transformed code generated is quite complex, their compiler achieves similar performance to cuBLAS 1.0 for matrix multiplication, and better results for other kernels.

Yang et al. propose an optimizing compiler framework to allow effective utilization of the GPU memory hierarchy and judicious management of parallelism [40]. Their source-to-source compiler produces an optimized CUDA code, which can be compiled using CUDA compiler. Their work focuses on optimizing global memory usage by improving memory coalescing and performing loop tiling. It also achieves effective data sharing by the use of thread-block/thread-merge techniques.

3.3 Cross-Compilation

The increasing popularity of GPUs has prompted researchers to develop source-to-source compilers that allow applications written in other languages to execute on GPUs. Lee et al. propose an automated compiler framework for translation of OpenMP applications into CUDA [24]. Their proposed translation converts the loop-level parallelism of the OpenMP programming model into the data parallelism of the CUDA programming model. They also identify several key transformations such as parallel-loop swap and matrix transpose for regular applications, and loop collapsing for irregular structures. Alternatively, MCUDA maps CUDA onto a conventional CPU architecture [38].
CHAPTER 3. RELATED WORK

The authors describe an algorithm to wrap kernel statements inside thread loops and perform loop fission to preserve the semantics of synchronization barriers.

The Swan source-to-source compilation tool can be used to compile CUDA C programs to run on CPU and GPU backends [12]. Internally, Swan compiles CUDA C programs into OpenCL using regular expression substitution and makes use of the OpenCL library to run on multiple platforms. An attempt to translate annotated LLVM IR to OpenCL kernel has also been made before [11].

3.4 Intermediate Representations

Compilers use intermediate representations (IR) as a stepping stone to map the user program into machine language. Designing a good IR has been studied for many years. One of the most important characteristics of a good IR is the flexibility that it can provide to the compiler to implement optimizations and perform data analysis. Several program representations have been proposed in this direction.

NVIDIA introduced its Parallel Thread Execution (PTX) to allow programs written in CUDA to be compiled to this virtual instruction set [28]. The PTX kernels can later be compiled at runtime by the driver, depending on the specific GPU card being used. This 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 as a load-store instruction set, i.e., operands have to loaded from and stored to memory before operating on them using specific memory instructions. PTX resembles an assembly-like language with additional features such as 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 AMD Intermediate Language (AMDIL) to compile different types of shaders for graphics (vertex and pixel shaders) and computing (compute shaders) [6]. AMDIL was originally designed for graphics and presents several instructions that are graphics-only related. The control flow is handled using higher-level constructs including if-then-else and while-loops. AMDIL is characterized as being typeless (a register holds raw 32-bit values), and has a number of swizzles and permutations to access the register file.
CHAPTER 3. RELATED WORK

In 2012, the HSA (Heterogeneous System Architecture) Foundation was formed with the objective of making programming heterogeneous devices dramatically easier. HSA proposed royalty-free hardware specifications and open source software [33]. Recently, the foundation put forth a specification for HSAIL (the HSA Intermediate Language), which is positioned as the ISA of an HSAIL virtual machine for any computing device that plans to adhere to the standard [34]. HSAIL is quite low level, and similar to the assembly language of a RISC machine. HSAIL assumes a specific program and memory model that caters to heterogeneous platforms where multiple ISAs exist, with one specified as the host. It also specifies a model of parallel processing as part of the virtual machine.

3.5 Optimizing Register Usage

CPU-based compilers have been extensively studied over many decades. Some noteworthy research has been done in this domain. Compiler back-ends incorporate many sophisticated optimizations that require in-depth understanding of both hardware and software features. One of the important compiler optimizations is Register Allocation. Register Allocation is the process of assigning a large number of program temporaries to a small number of hardware registers. Since there are only a few hardware registers available on CPUs/GPUs, it is critical to utilize them efficiently. Therefore, register allocation plays a crucial role in this case.

One of the earliest works on register allocation was done by Chaitin et al. in 1981. In this work, register allocation was considered to be a graph coloring problem [10]. They later improved upon this work by including spilling as a part of the graph coloring process [9]. P. Briggs further improved Chaitin’s work by incorporating other optimizations such as live-range splitting, value re-numbering etc., along with graph coloring [8]. Unfortunately, the graph coloring algorithm is an NP-complete problem, and is not the most efficient way of doing register allocation [31].

To address this problem, register allocation was done on programs in Static Single Assignment (SSA) form [18]. SSA form requires that each variable is assigned exactly once, and every variable is defined before it is used. SSA improves the results of a variety of compiler optimizations by simplifying the properties of variables [5]. The IR in SSA has properties of a chordal graph which can be colored in polynomial time [30].
Chapter 4

Multi2C: A GPU Kernel Compiler

In this chapter, we describe the Multi2C compiler framework. The register allocator transformation pass is present as a part of the Multi2C framework, which is an open source GPU kernel compiler. As shown in Figure 4.1, Multi2C is a modular framework that converts OpenCL to SI GPU ISA. It comprises three stages: a frontend, a backend, and an assembler. The frontend translates OpenCL C to LLVM 3.4 bitcode representation using the flex and bison tools [16, 29]. The backend translates the LLVM bit code to the GPU ISA. The target architecture is AMD Southern Islands (SI). The assembler reads the plain text assembly code and converts it to a kernel binary, which can execute on the Multi2Sim simulation framework [39]. The Multi2C backend also supports the Clang-based OpenCL frontend [21].

Figure 4.1: Structure of Multi2C compiler

**Frontend:** We illustrate the behavior of the frontend using the example of a Vector Addition OpenCL kernel. The following is the OpenCL C version of a Vector Addition kernel:
The frontend translates the above OpenCL kernel into an LLVM bitcode, which can be disassembled using the LLVM disassembler tool to produce the LLVM IR, as shown below.

```
; Function Attrs: nounwind
define void @vectorAdd(float addrspace(1)* %output, 
float addrspace(1)* %inputA, 
float addrspace(1)* %inputB) #0 {
  %tmp_4 = call i32 @__get_global_id_u32(i32 0)
  %tmp_7 = getelementptr inbounds float addrspace(1)* %inputA, i32 %tmp_4
  %tmp_8 = load float addrspace(1)* %tmp_7, align 4
  %tmp_11 = getelementptr inbounds float addrspace(1)* %inputB, i32 %tmp_4
  %tmp_12 = load float addrspace(1)* %tmp_11, align 4
  %tmp_13 = fadd float %tmp_8, %tmp_12
  %tmp_16 = getelementptr inbounds float addrspace(1)* %output, i32 %tmp_4
  store float %tmp_13, float addrspace(1)* %tmp_16, align 4
  ret void
}
declare i32 @__get_global_id_u32(i32) #1
```

The above LLVM IR code contains only one basic block. Basic blocks are a sequence of instructions without any control flow or branch-free code. These basic blocks form the nodes of the Control Flow Graph (CFG). A CFG is a directed graph that models the control flow between basic blocks in the program. As mentioned in Section 2.2, LLVM is a linear IR which is converted to a CFG, a hybrid IR.

**Backend:** The backend translates the LLVM instructions to Southern Islands (SI) ISA instructions. Each LLVM instruction may map to zero or more SI instructions. For example, an integer add instruction in LLVM has a one-on-one mapping to SI instructions, a floating point division instruction in LLVM translates to multiple SI instructions, and a phi instruction in LLVM does not have a corresponding instruction in SI. The code generated by the backend from the Vector Add LLVM IR is shown and described below.
CHAPTER 4. MULTI2C: A GPU KERNEL COMPILER

```
.global vectorAdd

.args
  float* output 0 uav10
  float* inputA 16 uav11
  float* inputB 32 uav12

.text

header:
  s_buffer_load_dword s13, s[2:5], 0x0
  s_buffer_load_dword s14, s[2:5], 0x1
  s_buffer_load_dword s15, s[2:5], 0x2
  s_buffer_load_dword s16, s[2:5], 0x4
  s_buffer_load_dword s17, s[2:5], 0x5
  s_buffer_load_dword s18, s[2:5], 0x6
  s_buffer_load_dword s19, s[2:5], 0x18
  s_buffer_load_dword s20, s[2:5], 0x19
  s_buffer_load_dword s21, s[2:5], 0x1a
  v_mov_b32 v3, s16
  v_mul_i32_i24 v3, s10, v3
  v_add_i32 v3, vcc, v3, v0
  v_add_i32 v3, vcc, s19, v3
  v_mov_b32 v4, s17
  v_mul_i32_i24 v4, s11, v4
  v_add_i32 v4, vcc, v4, v1
  v_add_i32 v4, vcc, s20, v4
  v_mov_b32 v5, s18
  v_mul_i32_i24 v5, s12, v5
  v_add_i32 v5, vcc, v5, v2
  v_add_i32 v5, vcc, s21, v5
  s_mov_b32 m0, 0x8000

uavs:
  s_load_dwordx4 s[24:27], s[0:1], 0x50
  s_load_dwordx4 s[32:35], s[0:1], 0x58
  s_load_dwordx4 s[40:43], s[0:1], 0x60

args:
  s_buffer_load_dword s22, s[6:9], 0x0
  s_waitcnt lgkmcnt(0)
  s_buffer_load_dword s28, s[6:9], 0x4
  s_waitcnt lgkmcnt(0)
  s_buffer_load_dword s36, s[6:9], 0x8
  s_waitcnt lgkmcnt(0)
```
CHAPTER 4. MULTI2C: A GPU KERNEL COMPILER

```assembly
eentry:
  v_mul_i32_i24 v10, 0x4, v3
  v_add_i32 v9, vcc, s28, v10
  tbuffer_load_format_x v11, v9, s[32:35],
    0x0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
  s_waitcnt vmcnt(0)
  v_mul_i32_i24 v13, 0x4, v3
  v_add_i32 v12, vcc, s36, v13
  tbuffer_load_format_x v14, v12, s[40:43],
    0x0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
  s_waitcnt vmcnt(0)
  v_add_f32 v15, v11, v14
  v_mul_i32_i24 v17, 0x4, v3
  v_add_i32 v16, vcc, s22, v17
  tbuffer_store_format_x v15, v16, s[24:27],
    0x0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
  s_waitcnt expcnt(0)
  s_endpgm

.metadata
  userElements[0] = PTR_UAV_TABLE, 0, s[0:1]
  userElements[1] = IMM_CONST_BUFFER, 0, s[2:5]
  userElements[2] = IMM_CONST_BUFFER, 1, s[6:9]

  FloatMode = 192
  IeeeMode = 0

  COMPUTE_PGM_RSRC2:USER_SGPR = 10
  COMPUTE_PGM_RSRC2:TGID_X_EN = 1
  COMPUTE_PGM_RSRC2:TGID_Y_EN = 1
  COMPUTE_PGM_RSRC2:TGID_Z_EN = 1
```

The various sections produced in the code are briefly explained below.

- The `.global` keyword is followed by the name of the kernel. This section has no content.
- The `.args` section declares the kernel arguments. A kernel argument can be a pointer, a vector of pointers, a value, or a vector of values. All arguments are stored in constant buffer 1 at a specific offset from the beginning of the buffer. These offsets have to maintain a 16-byte alignment.
- The `.text` section contains the assembly instructions for the kernel, following the format specified in the Southern Islands ISA documentation [3].
The .metadata section is composed entirely of assembler directives specifying information needed for the final binary creation. More details about this section can be found in the Multi2Sim Reference Manual [4].

The SI instructions generated after translation are not optimized to efficiently utilize the register file. We describe the various steps involved in optimizing register usage in the next chapter.
Chapter 5

Register Allocation

In this chapter, we describe the implementation details of the register allocator. As mentioned in Chapter 2, the current generation of the AMD Southern Islands GPU supports two classes of registers: SGPR (scalar registers) and VGPR (vector registers). Vector registers are associated with the SIMD units, and hold different values for each workitem. Scalar registers are associated with the scalar unit and hold one value per wavefront. Register allocation for each class is run separately - first for scalar registers and then for vector registers. This will allow scalar registers to spill into vector registers when we support spilling in future.

5.1 Liveness Analysis

The first step in register allocation is to perform liveness analysis on the program CFG. Liveness Analysis is a data-flow analysis technique used by compilers to calculate the live range of every variable. A live range is defined as a write to a register followed by all the uses of that register until the next write. A variable is live at a certain point if it holds a value that is needed in the future. One of the major issues with liveness analysis is thread divergence.

As discussed in Section 2.5, thread divergence causes some workitems to take a different execution path when a conditional branch is resolved. A 64-bit execution mask is used to represent the active status of an individual workitem in the wavefront. If a workitem is labeled as inactive, the result of any arithmetic computation performed in its associated stream core is ignored, preventing it from changing the kernel state. The scalar instructions are executed for all workitems regardless of whether a workitem is active or not. Therefore, the liveness analysis of scalar registers must take into account the state of the registers before the execution mask flips.
CHAPTER 5. REGISTER ALLOCATION

Figure 5.1: Thread Divergence: (a) Sample pseudo code (b) Execution path when t0 and t1 are vector temporaries (b) Execution path when t0 and t1 are scalar temporaries.

Figure 5.1a shows sample code containing a conditional branch causing thread divergence. If t0 and t1 are vector temporaries/values, then the state of vector registers for workitems taking the \textit{then} path is independent of the state of vector registers for workitems taking the \textit{else} path. This is because vector registers are associated with every SIMD and hold different values for every workitem. Hence, as shown in Figure 5.1b, a per-thread walk for vector registers would either be $1 \rightarrow 2 \rightarrow 4$, or $1 \rightarrow 3 \rightarrow 4$, depending on the state of the execution mask. In contrast, since there is only one value held by the scalar registers per wavefront, there is a dependency between the state of the scalar registers used in the \textit{then} path and the state of scalar registers used in the \textit{else} path. If t0 and t1 are scalar values, they cannot share the same physical register. In the scalar CFG (Figure 5.1b), this dependency is missed if we do not add a scalar edge. The scalar edge ensures that the liveness information of scalar edge correctly flows from \textit{then} path to \textit{else} path and are not inadvertently overwritten after the execution mask flips. Therefore, the scalar engine sees an execution order of $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

The algorithm for creating the scalar edge is shown in Algorithm 5. Our implementation uses a technique called \textit{Structural Analysis} to identify the regions of the control flow graph (CFG) \cite{57}. 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.
CHAPTER 5. REGISTER ALLOCATION

node of the control tree. The translator can then iterate through the control tree starting from the root node and recursively processes each node. When if-then-else regions are identified, a scalar edge is created to connect the then and else blocks. Connecting blocks by a new edge requires modification to their predecessor and successor basic block list. The new list must be considered while doing scalar liveness analysis.

Algorithm 1 Implementation of a scalar edge

```plaintext
function TDCONNECT(else_node)
    result ← Structural Analysis (program_tree)
    if result == if_then_else_block then
        then_td_succ_node ← else_node  ▷ then node has else node as TD successor
        elseTd_pred_node ← then_node   ▷ else node has then node as TD predecessor
    end if
end function
```

This information is used in Liveness Analysis to generate correct liveness information for every basic block as shown in Algorithm 2. The terms are defined as:

- `in[b]`: variables live at the start of the basic block
- `out[b]`: variables live out at the end of the basic block
- `use[b]`: variables that are used in `b` before any assignment.
- `def[b]`: variables assigned a value in `b` before any use.

Liveness analysis is a backward analysis pass and consists of two phases. In the first phase, the `use[b]` and the `def[b]` for every basic block are calculated. The `use[b]` and `def[b]` are consumed by the subsequent pass that iteratively calculates the `in[b]` and `out[b]`.

Algorithm 2 Liveness Analysis

```plaintext
function LIVENESSananalysis
    foreach basic block b : in[b] ← ∅; out[b] ← ∅  ▷ Initialize the sets
    repeat foreach b : in'[b] ← in[b]; out'[b] ← out[b]  ▷ Save current results
        out[b] ← ∪
            s∈succ[b]
            in[s]
        in[b] ← use[b] ∪ (out[b] - def[b])  ▷ Solve data-flow equations
    until in' = in ∧ out' = out  ▷ Iterate till the In and Out of all basic blocks converge
end function
```
5.2 Creating an Interference graph

The register allocation is algorithmically similar to the mathematical problem of graph coloring. When formulated as a graph coloring problem, each node in the graph represents the live range of a particular value. An edge between two nodes indicates that those two live ranges interfere with each other because they are simultaneously active at some point in time. Therefore, interfering live ranges must be assigned to different registers. The resulting graph formed is called an interference graph.

One of the challenges in the interference graph on a SI GPU is that scalar and vector memory operations allow program instructions to fetch multiple dwords at a time. For example, Scalar Memory Read (SMRD) instructions allow a shader program to load 1-16 doublewords at a time into SGPRs. Instructions S_LOAD_DWORD and S_BUFFER_LOAD_DWORD fall under the category of a SMRD instruction [3]. To accommodate scalar and vector register series, we created weighted nodes in the graph; each node having a weight/size attribute associated with it as shown in Figure 5.2. The weight/size corresponds to the number of colors or registers needed to color each node. For the purpose of simplicity, nodes that require multiple registers for coloring (i.e., size > 1) are referred to as big nodes. And individual registers that require only one color (i.e., size = 1) are referred to as small nodes. Having a weighted graph simplifies the representation by removing interference.

To further improve the allocator’s efficiency, we build both a lower-diagonal bit matrix and a set of adjacency lists to represent the interference graph, I. The bit matrix allows a constant-time test for interference, while the adjacency list allows efficient iteration over a node’s neighbors.
CHAPTER 5. REGISTER ALLOCATION

We use the algorithm provided in [12] to create an interference graph. Two separate interference graphs are generated for scalar and vector registers.

5.3 Graph Coloring

In the last stage, we assign physical registers to the nodes of the interference graph using graph coloring technique. The number of contiguous registers to be assigned to every node is equal to the size of the node. The problem arises during color selection of a big node. If the allocator cannot find a free color/register for any single value in the series/big node, it must restart the coloring of the big node until it finds a contiguous block of free registers equal to the size of the node. For the allocator to reconsider colors requires back tracking, which can require exponential time. Therefore, we color all the big nodes first, followed by the smaller nodes. Pseudocode for our implementation is shown in Algorithm 3. The algorithm shows the coloring process for a node, which is repeated until all nodes in the graph are colored.
CHAPTER 5. REGISTER ALLOCATION

Algorithm 3 Graph Coloring

1: function GRAPHCOLORING
2:     Input: series_reg  ▷ Could be small or big node of a graph
3:     Input: adj_list  ▷ Adjacency list of a node
4:     r_iter = 0  ▷ Iterator for registers
5:     while r_iter < Max_regs do  ▷ Max registers available on the GPU
6:         found = false;  ▷ variable to compare neighbors of a node and register number.
7:         for j in adj_list do
8:             if r_iter == j then  ▷ if neighbor color equals register number, the register cannot be used
9:                 found = true
10:                breakFor;
11:         end if
12:     end for
13:     if found == true then  ▷ if found = true, skip the while loop and move to next register
14:         r_iter + +;
15:         ContinueWhile;
16:     else  ▷ if match is not found
17:         if series_reg.size == 0 then  ▷ If series empty, add register to series
18:             series_reg ← r_iter
19:             r_iter + +;
20:            ContinueWhile;
21:         else if series_reg.size < node_size then  ▷ full series is not found yet
22:             if r_iter == series_reg.last + 1 then  ▷ Check for contiguous arrangement of registers
23:                 series_reg ← r_iter
24:                 r_iter + +;
25:                ContinueWhile;
26:         else
27:             series_reg.clear  ▷ Clear series if not contiguous
28:             r_iter + +;
29:            ContinueWhile;
30:         end if
31:     else if series_reg.size == node_size then  ▷ Full range is found
32:         r_iter + +;
33:        breakWhile;
34:     end if
35:     end if
36: end while  ▷ Repeat for all nodes
37: end function
Chapter 6

Evaluation Methodology

6.1 Platform for evaluation

We evaluate our register allocation mechanism using the Multi2Sim simulation framework [39]. Multi2Sim provides a detailed model of GPU compute units, cache hierarchy and interconnect network. We use the AMD Southern Islands GPU model for our evaluation. The configuration details of the AMD GPU are shown in Table 6.1.

<table>
<thead>
<tr>
<th>Compute Unit Config</th>
<th>Device Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td># of CU’s</td>
<td>32</td>
</tr>
<tr>
<td># of Wavefront Pools / CU</td>
<td>4</td>
</tr>
<tr>
<td># of SIMD Units / CU</td>
<td>4</td>
</tr>
<tr>
<td># of lanes / SIMD</td>
<td>16</td>
</tr>
<tr>
<td># of vector reg / CU</td>
<td>64K</td>
</tr>
<tr>
<td># of scalar reg / CU</td>
<td>2K</td>
</tr>
<tr>
<td>Frequency</td>
<td>1GHz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Architecture</th>
<th>Device Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 (1 /CU)</td>
<td>16KB</td>
</tr>
<tr>
<td># of shared L2</td>
<td>6</td>
</tr>
<tr>
<td>L2 Size</td>
<td>128KB</td>
</tr>
<tr>
<td>Global Memory</td>
<td>1GB</td>
</tr>
<tr>
<td>Local Memory / CU</td>
<td>64K</td>
</tr>
</tbody>
</table>

Table 6.1: The device configuration of the Southern Islands GPU.
CHAPTER 6. EVALUATION METHODOLOGY

6.2 Evaluated Benchmarks

Table 6.2 provides a list of benchmarks from AMD APP SDK that we used for performance evaluation [1].

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Working Set</th>
<th>Nbr of Work-Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Bitonic Sort: Sorting network of ( O(n\log^2 n) ) comparators. Performs best when sorting a small number of elements.</td>
<td>1024 elements</td>
<td>110</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform: Sum of cosine functions oscillating at different frequencies</td>
<td>( 2^6 ) elements</td>
<td>256</td>
</tr>
<tr>
<td>DWT</td>
<td>Discrete Wavelet Transform: 1-dimensional discrete wavelet transform for Haar signals.</td>
<td>( 2^{12} ) elements</td>
<td>5</td>
</tr>
<tr>
<td>FWT</td>
<td>Fast Walsh Transform: Naturally-ordered Fourier transformation.</td>
<td>( 2^8 ) elements</td>
<td>9</td>
</tr>
<tr>
<td>FYW</td>
<td>Floyd Warshall: Finds shortest path in a weighted graph with no negative cycles</td>
<td>64 nodes</td>
<td>65536</td>
</tr>
<tr>
<td>PRE</td>
<td>Prefix Sum: Sum of prefixes of the input sequence</td>
<td>1024 elements</td>
<td>1</td>
</tr>
<tr>
<td>RED</td>
<td>Reduction: Parallel sum reduction</td>
<td>( 2^{15} ) elements</td>
<td>16</td>
</tr>
<tr>
<td>TRN</td>
<td>Matrix Transpose: Matrix transpose optimized to coalesce accesses to shared memory and avoid bank conflicts.</td>
<td>512 X 512 elements</td>
<td>64</td>
</tr>
<tr>
<td>VCP</td>
<td>Vector Addition: Element-by-element vector add</td>
<td>8192 integer elements</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 6.2: Benchmarks from AMD APP SDK
Chapter 7

Results

This chapter compares the performance of our register allocator with the baseline. The baseline is a greedy algorithm that keeps assigning registers without reconsidering its choices or checking whether the previously assigned registers are available for reuse. The results are measured on the AMD Southern Islands GPU modeled on the Multi2sim simulator.

![Change in register usage](image)

Figure 7.1: The change in register usage. Register usage is *lower is better* metric.
CHAPTER 7. RESULTS

7.1 Change in Register usage

We evaluate the effect of register allocation on the Vector GPR usage and Scalar GPR usage. As shown in Figure 7.2, we see an average reduction of 22.5% and 44.2% in vector and scalar register usage, respectively. This is mainly attributable to the fact that our algorithm reuses registers more efficiently than the baseline algorithm. The liveness analysis algorithm in our allocator calculates the lifetime of different variables and so no interference is added between non-overlapping live ranges. These non-overlapping liveranges can subsequently be assigned to the same physical register, ensuring better register reuse.

None of the applications that we tested produced register spilling. Although we currently do not support register spilling in our compiler backend, our decision to color register series (big nodes) before individual registers (small nodes) is primarily to avoid spilling register series to off chip memory, which can be very expensive in terms of performance and power.
CHAPTER 7. RESULTS

7.2 Impact of register usage on kernel occupancy.

Kernel occupancy is used to measure the utilization of the resources of a compute unit on a GPU. Occupancy is the number of in-flight wavefronts for a given kernel, relative to the number of wavefronts that could be launched given the ideal kernel dispatch configuration [2].

Mathematically, occupancy (O) is given by:

\[ \text{Occupancy}, O = \frac{N^A}{N^T} \]

where, \( N^A \) is the actual number of in-flight wavefronts on a compute unit and \( N^T \) is the theoretical number of wavefronts that a compute unit can execute concurrently.

The number of wavefronts that can be scheduled on the compute units is constrained by the number of general purpose registers (GPRS) required by every kernel, the amount of shared memory (LDS for local data store) used by each work-group, and the configuration of the work-group (the work-group size). In Figure 7.3, we see an average improvement of 73% in kernel occupancy.
CHAPTER 7. RESULTS

occupancy for all applications. BIT, FYW, FWT, and VCP are bound by their GPR usage and experienced a significant improvement in occupancy when the register usage improved. In contrast, PRE, RED, and DWT did not show any improvement in occupancy because they were constrained by local memory usage. TRN has high GPR usage and uses a significant amount of local memory, so it did not see any benefit in terms of occupancy. DCT saw an improvement in register usage, but did not show any improvement in occupancy because it is constrained by the number of wavefronts executing on the compute unit.

7.3 Effect of kernel occupancy on performance

Figure 7.3 shows an overall improvement of 5.4% for all applications. Since we have been able to improve occupancy, BIT, FYW, FWT and VCP showed improvement in performance. This is because when the occupancy increases, more wavefronts can execute concurrently on the GPU, thereby reducing the overall execution time of the application. Also, with higher occupancy, at least one wavefront can execute while others are stalled on long-latency operations, such as memory loads. DWT, PRE, RED, and TRN did not show any change in performance as their occupancy remained unchanged as mentioned in section 7.2.
Chapter 8

Discussion

8.1 Register Allocation in SSA form using Virtual phi Instruction

As mentioned in Section 2.2.3, LLVM uses SSA form as its primary code representation. SSA form requires that each variable is assigned exactly once, and every variable is defined before it is used, as shown in Figure 8.1. SSA improves the results of a variety of compiler optimizations by simplifying the properties of variables. SSA form is achieved by inserting phi nodes, which are represented by using phi instructions. The syntax of phi instruction is as follows [23]:

\[
<\text{result}> = \phi <\text{ty}> [ <\text{val0}>, <\text{label0}>], \ldots
\]

The type field determines the type of the incoming values. The phi instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. Only
values of the first class type may be used as the value arguments to the phi node, and labels can only be used as the label arguments. The phi instructions must always be first in a basic block.

Since the SI ISA does not contain a phi instruction, we created a virtual phi instruction in the SI ISA to facilitate register allocation in SSA form. The pseudocode for generating virtual phi instructions is provided in Algorithm 4. Currently, this algorithm works only with 32-bit integer types, <ty>. The algorithm first maps the LLVM destination variable, <result> to a SI vector register. Then, for every incoming basic block, we check that its label <label 0>, <label 1>, .. and incoming value, <val 0>, <val 1>, ... associated with the basic block. The incoming value is assigned to a vector register and the phi arguments (label and value) are pushed into an argument list.

Algorithm 4 Implementation of virtual phi instructions.

1: function VIRTUAL PHI(llvm_instruction)
2: if llvmType ≠ i32 then
3: throw error
4: end if
5: dst_vector_register ← llvm_destination_argument ▷ Assign a vector register to the destination
6: for i = 0 → NumberOfIncomingValues() do
7: label ← IncomingBlock(i)
8: source_vector_register ← IncomingValue(i)
9: argument_list ← phi Arguments ▷ phi Args consist of label and source value
10: end for
11: Create a new SI phi instruction
12: Add phi Arguments to phi instruction
13: end function

This repeats for every incoming basic block, and the argument list is then attached to the phi instruction.
Chapter 9

Conclusion

In this thesis, we developed algorithms that performed liveness analysis, constructed interference graphs and applied graph coloring to guide vector and scalar register allocation on AMD GPUs. We compared our algorithm with a naive greedy baseline and observed an average overall savings of 22.5% and 44.2% in vector and scalar register usage. Reduction in vector and scalar register usage improved the average occupancy by 73% for applications that were not bound by local memory usage and the number of wavefronts. For all applications, we saw an average performance improvement of 5.4% over the baseline. Higher occupancy allows more wavefronts to execute concurrently reducing the overall execution time.

The practical importance of this work is that it provides researchers with an in-depth understanding of the challenges faced while designing a register allocator for GPUs. Our analysis shows that change in SIMD occupancy may or may not result in an improvement in performance. We currently do not support register spilling in our compiler, but the effect of register allocation is expected to be more pronounced when we support spilling in future. It will be an interesting study to understand the interaction of various compiler optimizations with register allocation on a real GPU. We also plan to implement and compare chordal graph allocator with our allocator.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


