A GPU BINARY ANALYSIS FRAMEWORK FOR MEMORY PERFORMANCE AND SAFETY

By

ARI B. HAYES

A dissertation submitted to the School of Graduate Studies Rutgers, The State University of New Jersey

In partial fulfillment of the requirements For the degree of Doctor of Philosophy Graduate Program in Computer Science

Written under the direction of Zheng Zhang

And approved by

________________________________________

________________________________________

________________________________________

________________________________________

New Brunswick, New Jersey
May 2022
ABSTRACT OF THE DISSERTATION

A GPU Binary Analysis Framework for Memory Performance and Safety

by ARI B. HAYES

Dissertation Director: Zheng Zhang

General-Purpose Graphics Processing Units (GPUs) have attained popularity for their massive concurrency. But with their relative infancy, as well as a prevalence of closed-source and proprietary technology, GPU software has not undergone the degree of optimization that CPU software has. In this work, we focus on NVIDIA’s GPUs and the CUDA framework, but our techniques may be generalized to other GPU platforms.

We develop a compiler which targets the low-level SASS assembly, rather than the high-level source code or the intermediate-level PTX assembly. This allows for a degree of tuning and modification not possible at higher levels. We reconstruct program information such as the control-flow graph and call graph, thereby permitting data flow analysis. Thanks to extensive reverse-engineering, our compiler retains compatibility with numerous versions of the CUDA framework and several generations of NVIDIA GPUs.

We are able to improve memory performance with optimizations not available in the proprietary compiler. We perform memory-allocation across the multiple types of available on-chip memory, making full use of available resources including registers, scratchpad memory, and the L1 data cache. This further allows us to tune occupancy - the number of threads allowed to be simultaneously active. We perform static and dynamic occupancy tuning, in order to find an effective balance between concurrency and resource contention.

Our compiler can also be applied toward improvement of memory safety. We use it
to implement dynamic taint tracking, a technique previously used on CPUs to identify sensitive data as it spreads through memory. By analyzing and modifying the low-level assembly, we minimize tracking overhead, track memory resources not visible to the programmer, and erase sensitive data before it has opportunity to leak.

We evaluate our compiler across a number of benchmarks on NVIDIA devices of the Fermi, Kepler, Maxwell, and Pascal architectures. We demonstrate that our resource allocation and occupancy tuning provides significant improvement in both speed and energy usage. We additionally show that our GPU-specific optimizations for taint tracking can enormously reduce overhead.
ACKNOWLEDGMENTS

I offer my deepest gratitude to my advisor, Prof. Zheng Zhang, for her guidance and support throughout my studies, and for encouraging me to apply to the PhD program at Rutgers University in the first place.

I would also like to thank the members of my dissertation committee, Prof. Ulrich Kremer, Prof. Manish Parashar, and Prof. Chen Ding.

I would like to thank my lab-mates and other collaborators, without whom my work and my time at Rutgers would have been lesser.

Last, but never least, I would like to thank my family and friends for sticking by me through the years.

I acknowledge that my research presented in this dissertation has previously appeared in [25, 26, 27, 32].
TABLE OF CONTENTS

Abstract ........................................................................................................... ii

Acknowledgments ......................................................................................... iv

List of Tables ................................................................................................. ix

List of Figures ............................................................................................... x

Chapter 1: Introduction .................................................................................. 1
  1.1 Motivation .................................................................................................. 1
  1.2 Background .............................................................................................. 1
    1.2.1 GPU Threads and Occupancy ............................................................. 1
    1.2.2 Memory Security .............................................................................. 3
  1.3 Organization ............................................................................................. 4

Chapter 2: Decoding CUDA Binary .............................................................. 5
  2.1 Overview .................................................................................................. 5
  2.2 The Instruction Set Architecture ........................................................... 7
    2.2.1 Assembler Generating Framework .................................................... 8
    2.2.2 Unique CUDA Binary Instructions & Behavior ................................. 10
  2.3 Assembler Generation ............................................................................. 12
Chapter 4: Orion System Design

4.2 Orion System Design

4.2.1 Our Framework

4.2.2 Realizing Occupancy

4.2.3 Compile-Time Occupancy Tuning

4.2.4 Runtime Occupancy Adaptation

4.3 Evaluation

4.3.1 Performance

4.3.2 Resource & Energy Usage

4.4 Related Work

4.5 Conclusion

Chapter 5: GPU Taint Tracking

5.1 Overview

5.2 Efficient GPU Taint Tracking

5.2.1 Taint Reachability

5.2.2 Iterative Two-pass Taint Analysis

5.2.3 Register Taint Map in Registers

5.3 Tainting-Enabled Data Protection

5.3.1 Sensitive Data Removal

5.3.2 GPU Malware Countermeasure

5.4 Evaluation

5.4.1 Benchmarks

5.4.2 Taint Analysis & Optimizations
LIST OF TABLES

2.1 Common memory instructions on GPU. We refer to each general register as Rx or Ry, and each literal as 0xa for illustration purpose. ........................................ 10

2.2 Common instructions for Compute Capability 3.x, with operands defined in terms of Figure 2.8. Memory operands are denoted with square brackets. PC refers to the program counter, LOP refers to an arbitrary logic operation, and comp is short for composite. ................................................. 23

2.3 The most common special registers used on GPU. ........................................ 24

3.1 Benchmark Description. RegDemand is the number of registers the compiler tries to use. InstChange is the increase to the instruction count in the auto occupancy. CacheMissRates are the cache miss rates in the auto occupancy; the numerator is with use of shared memory after transformation, and the denominator is with purely global spills before transformation. ........ 52

4.1 Notations .......................................................... 68

4.2 Detailed benchmark information. Reg is the number of registers needed to avoid spilling. Func is the number of static function calls. Smem indicates whether there is user-allocated shared memory. ........................................... 78

4.3 Speedup with Small Cache (SC) vs Large Cache (LC) at ORION’s selected occupancy. In some cases, hardware constraints prevent the LC case from running. ................................................................. 82

5.1 Percentage of filtered-out instructions for various reasons. .......................... 105

5.2 Slowdown from memory erasure during kernel execution, measured as a fraction of the original kernel time. “Memory” column indicates which memory types need to be cleared (besides registers). ................................. 108
# LIST OF FIGURES

2.1 Our assembler generation framework. Our work is highlighted with black background, and the existing tools are in white background. .......................... 8

2.2 Decoded IADD instruction for computing capability 3.5. We show which bits correspond to which component in the IADD instruction. ................................. 9

2.3 An example of output from cuobjdump, compiled for NVIDIA’s Compute Capability 3.5 architecture. We have omitted certain details such as instruction address for illustration purpose. ........................................ 9

2.4 A simple example of thread-warp divergence. (a) is the source and assembly code, (b) is the control flow graph. .................................................. 11

2.5 Looking for the bits controlled by the first operand. ............................. 16

2.6 Structures used in our Algorithms’ pseudocode. ................................. 18

2.7 A simplified example of an assembler that has been automatically generated. 22

2.8 Common operands on different CUDA architectures. .......................... 26

2.9 An example of how we extract the scheduling information for each group of seven instructions on Kepler GPUs. These 8-bit values indicate dispatch behavior. .................................................... 28

2.10 An example of how we extract the scheduling information for each group of three instructions on Maxwell and Pascal GPUs. The first and second values indicates dispatch behavior. The third and fourth indicate which barriers to set. The fifth indicates which barriers to wait for. .......................... 29

2.11 Example of converting local memory instructions to shared memory instructions. ................................................................. 31

2.12 Example of instrumenting the code to clear some registers before exit. ... 33
3.1 Performance of compiled programs with various per-thread register usage. The x-axis represents the per-thread register usage for every compiled version. The y-axis shows the running time in seconds. We used an NVIDIA Kepler [66] GTX680. The range of register count is from 32 to the maximal register demand.

3.2 Unified On-Chip Memory Allocation

3.3 Reuse shared memory across procedure boundaries

3.4 Concurrency Level Sensitivity

3.5 Concurrency Level Search

3.6 Kepler Performance Results. The Auto bar shows speedup with automatically selected occupancy. The Best bar shows highest speedup among all occupancies by exhaustive search. The Worse bar shows worst speedup among all occupancies. For our baseline, we used the runtime when compiling each benchmark with default settings.

3.7 Fermi Performance Results. The Auto bar shows speedup with automatically selected occupancy. The Best bar shows highest speedup among all occupancies by exhaustive search. The Worse bar shows worst speedup among all occupancies. For our baseline, we used the runtime when compiling each benchmark with default settings.

4.1 Running time v.s. occupancy for the imageDenoising benchmark on GTX680. Occupancy is between 0.125 and 1.0.

4.2 Effect of occupancy on performance for matrixMul

4.3 The ORION compiler and runtime collaboration

4.4 Single procedure multi-class allocation alg.

4.5 Optimized v.s. unoptimized inter-procedure allocation. The benchmarks are from Rodinia benchmark suite and CUDA SDK.

4.6 The impact of on-chip memory slot layout on the number of data movements for a compressible stack. Every column $S_i$ represents a slot. The vertical blocks in the column represent the liveness of the variable assigned to the slot. Bold arrows represent direction of data movement for entering or leaving a sub-procedure.
4.7 Bipartite matching model for minimal-mov-assignment problem. $SS_i$ represents the i-th set of variables that can be mapped to one on-chip memory slot, $SLOT_i$ represents the i-th physical on-chip memory slot in the procedure of interest. .......................................................... 70

4.8 Occupancy update algorithm ..................................................... 72

4.9 Dynamic occupancy selection algorithm ..................................... 74

4.10 Normalized srad performance on Tesla C2075. ............................ 76

4.11 Normalized speedup over nvcc version. We show the best/worst performance at all occupancy levels. ORION-MAX is the best performance. ORION-MIN is the worst. ORION-SELECT is the performance with tuning. NVCC is the performance by nvcc generated code. ................................. 81

4.12 Results of downward occupancy tuning. ...................................... 82

4.13 Energy usage of selected kernel. .............................................. 83

4.14 Effects of occupancy on performance for C2075. ......................... 84

4.15 Effects of occupancy on performance for GTX680. ...................... 84

5.1 Neural network information leaking example. ............................... 87

5.2 Overview of our taint tracking system. ....................................... 93

5.3 Forward taint reachability analysis. .......................................... 94

5.4 Backward memory reachability analysis. .................................... 96

5.5 Overhead of tainting instrumentation. ........................................ 104
CHAPTER 1
INTRODUCTION

1.1 Motivation

Binary instrumentation tools such as Pin [55] for CPU programs have been around for a long time, whereas the very nature of binary code on the GPU has remained a comparative enigma. This is particularly true for the popular NVIDIA brand’s devices, which see use in such applications as machine learning, high-performance computing, and healthcare. It’s unclear why NVIDIA chose to close-source their work, but whether the goal is to prevent competitors from learning their secrets or to stop the end-user from having too much control, the result is the same.

How well refined are the compilation techniques for this proprietary platform? What really happens to your data once it enters the mysterious GPU memories? In this work we strive to answer these questions, as well as to address the problems these answers raise. All of our techniques are applied to real hardware, and our core binary analysis framework is open-source.

1.2 Background

1.2.1 GPU Threads and Occupancy

GPUs deliver high performance via massive multithreading through the single instruction multiple thread (SIMT) execution model. Every thread runs the same code on different input sets. GPU threads are organized into thread warps. A thread warp, the minimum execution unit, typically consists of 32 threads. Thread warps are further organized into thread blocks. A GPU is composed of multiple SMs. A thread block runs on at most one SM. The number of active threads on one SM is a multiple of the thread block size.
GPU on-chip memory includes registers and cache. *Shared memory* is the software-managed cache in NVIDIA terminology. We use NVIDIA terminology throughout this work. Access to shared memory is explicitly managed by software.

Every active thread gets an even partition of register file and shared memory. The amount of registers and shared memory used by every thread determines how many threads can be active at one time (the occupancy). Data from the off-chip local (thread-private) and global memories pass through the hardware-managed L1/L2 caches. Unlike registers and shared memory, hardware cache usage does not impose any constraints on the occupancy.

GPU *occupancy* [62] is defined as the ratio between the actual number of active thread warps and the maximum number of thread warps the hardware can schedule. The occupancy can be calculated using per-thread register usage, per-thread shared memory usage, and thread block size. At runtime, occupancy is set by the GPU driver, based on these parameters.

Assume in a program that every thread uses $V_{reg}$ register space and $V_{smem}$ shared memory space. The total register file size is $N_{reg}$ and the total shared memory size is $N_{smem}$. The maximum number of threads the hardware can schedule at one time is $S_{max}$. The formula below gives the occupancy.

\[
\text{Occupancy} = \min\left(\frac{N_{reg}}{V_{reg}}, \frac{N_{smem}}{V_{smem}}\right) / S_{max}.
\] (1.1)

Since the number of active threads needs to be rounded up to a multiple of thread block size, and the register partition needs to be aligned according to register bank size constraints, the occupancy may be smaller than above. We use the formula in NVIDIA occupancy calculator [63] to obtain the accurate occupancy.

A GPU program consists of both CPU code and GPU code. The code that runs on the GPU side is organized into GPU kernels. A *GPU kernel* is a function that runs on the GPU. Every GPU kernel embodies an implicit barrier since all threads that are launched by this kernel need to finish before the next kernel starts. We perform performance tuning for GPU
kernels only.

1.2.2 Memory Security

Both texture memory and constant memory are read-only during the GPU kernel execution, hence our focus on registers, shared memory, local memory, and global memory. Registers and local memory are thread-private, shared memory is shared by entire thread blocks, and global memory is shared by all GPU threads. In all four of these memory types, data persists after deallocation [69].

Global memory can be set and cleared through API functions, with overhead similar to that of running a GPU kernel. But local memory, shared memory, and registers are only accessible from within a kernel function, and allocated and deallocated by the driver. These three memory types can only be reliably cleared through binary instrumentation. Moreover, local memory and registers are managed by compilers and they can only be cleared by compile-time instrumentation.

Sensitive information can also propagate to different data storage locations on GPU: memory, software caches, and registers. An example is the advanced encryption standard (AES), in which the key and the plain text to be encrypted may reside in different types of memory [69]. They can be stored in global memory as allocated data objects and in registers as program execution operands.

Currently, there is less memory protection on GPUs as compared with CPUs. When two applications run simultaneously on the same GPU with the Multi-process Service (MPS), one application can peek into the memory of another application, documented in NVIDIA’s MPS manual at Section 2.3.3.1, “An out-of-range read in a CUDA Kernel can access CUDA-accessible memory modified by another process, and will not trigger an error, leading to undefined behavior.” When two applications do not run simultaneously, in which case every application will get a serially scheduled time-slice on the whole GPU, information leaking is still possible. The second running application can read data left by
the first running application if its allocated memory locations happen to overlap with those of the first one. This vulnerability has been detailed in several recent works [45, 69, 77].

Future hardware trends such as the fine-grained memory protection in AMD APUs suggest potentially better process isolation. Hardware-level memory protection may exhibit superior performance, but its realization must take into account the hardware implementation complexity. And more importantly, process memory protection does not distinguish sensitive data and its propagation within one program or process. Such protection would be critical for securing sensitive information flows between CPU, GPU and their memories.

1.3 Organization

The rest of this work is organized as follows.

- In Chapter 2 we detail the creation of our binary analysis & instrumentation framework for the GPU, offering unprecedented control of the GPU binary code. The techniques therein future-proof our work, easing the addition of compatibility for successive generations of hardware.

- In Chapter 3 we explore on-chip memory reallocation, in order to improve performance of GPU programs. We demonstrate the trade-off between resource usage and concurrency on GPU devices.

- In Chapter 4 we propose a tuning framework for GPU occupancy, taking into consideration the trade-offs explored in the previous chapter. We achieve valuable reductions in both run-time and energy usage.

- In Chapter 5 we demonstrate dynamic taint tracking analysis on the GPU. Having recognized severe vulnerabilities, our techniques provide a degree of data protection that would not otherwise be possible.

- Finally, Chapter 6 concludes and summarizes our work.
CHAPTER 2
DECODING CUDA BINARY

2.1 Overview

The CUDA parallel computing platform developed by NVIDIA has benefited greatly from the popularity of their powerful GPU hardware, seeing widespread use. But the proprietary, closed-source nature of this platform makes research more difficult and limited. Although NVIDIA provides documentation of the intermediate language, they provide very little information about the hardware-specific instruction sets.

Open-sourcing of software and hardware has two-fold benefits. The level of documentation that open-sourcing permits allows for better tuning of specific software. It also allows the community to spot errors and security flaws that can be corrected before they become more widespread.

Openness of an instruction set architecture (ISA), in particular, allows the development of more effective compilation techniques. The proprietary compiler is not guaranteed to be optimal when generating executable code. Indeed, it is usually the case that code can be further tuned to improve its efficiency. For example, by using known instruction encoding to tune allocation and scheduling, several works [43] [44] [22] [88] are able to achieve performance beyond that of what NVIDIA’s compiler, nvcc, can produce on its own. Furthermore, several other works [89] [32] [26] [47] [59] [27] demonstrate that modifications to the binary code can allow for useful techniques that are impossible for a programmer to fully implement in source or intermediate language.

An instruction set architecture (ISA) acts as the interface between software and hardware. The ISA allows independent development of software and hardware. The details the ISA describes, such as binary instruction encoding and data types, are vital to the devel-
opment of compilers. An open ISA helps research in GPU micro-architecture. There is a large number of GPU works that rely on GPU-Sim [4]. However, the cycle-level GPU simulator is unable to run the binary code used by actual NVIDIA devices, thus reducing its accuracy - as well as the accuracy of any works which make use of it. Further, an open ISA may help enhance GPU security: our work in [27] showed that understanding the ISA details is a necessity when tracking and protecting sensitive data across the whole system.

We develop a framework of techniques that enables easier interaction with the ISA of various NVIDIA architectures. Our work has found use in many different settings for program analysis and performance tuning of GPUs [32] [26] [27] [48].

In particular, we make the following contributions:

• We design a method which largely automates the creation of a GPU assembler. The method first analyzes binary code and then generates a translator for converting assembly code to binary code. This automation allows researchers to quickly add support for new ISAs.

• We provide the binary encoding for multiple existing GPU architectures, including how operands and opcodes are mapped to 0s and 1s, and how instruction dependence is handled. We have made our findings freely available on Zenodo, with opcodes at [30], and operands at [28].

• We provide the GPU ELF file format in Zenodo [29], which is necessary for complete editing of executable.

• Our techniques and framework have demonstrated compatibility with at least four consecutive major versions of NVIDIA devices: Compute Capability 2.x (Fermi), Compute Capability 3.x (Kepler), Compute Capability 5.x (Maxwell) and Compute Capability 6.x (Pascal).

• We develop an extensible framework that supports analysis and transformation of assembly code for use with the GPU assemblers, which has been used in multiple
works [32] [26] [27] [48]. Our framework is available on GitHub at https://github.com/decodecudabinary/Decoding-CUDA-Binary.

Prior studies have been trying to understand the hidden workings of NVIDIA’s devices. The work by Wong et al. [82] focuses on micro-architecture details such as cache design and memory latency. There have also been efforts at decoding binary instructions: Hou et al. [36] created an assembler for Compute Capability 2.x, Zhang et al. [88] decoded Compute Capability 3.x instructions used in an SGEMM implementation, Gray [22] developed an assembler for Compute Capability 5.x, and Jia et al. [41] decoded Compute Capability 7.x instructions. Each of these existing works is limited to a specific architectural generation, and in some cases a specific application. Our work is a comprehensive and cross-architecture framework that is useful for general-purpose and future-proof applications.

The rest of this chapter is organized as follows. In Section 2.2, we describe the ISA and our framework. We compare CPU ISA with GPU ISA in subsection 2.2.2. Section 2.3 describes our automatic decoding method. Section 2.4 summarizes our findings for different generations of NVIDIA GPUs. Section 2.5 outlines applications of our framework. We discuss related works in Section 2.6.

2.2 The Instruction Set Architecture

The assembly code used for NVIDIA devices is known as ”SASS”, which corresponds exactly to the actual binary code. Most details of SASS are kept secret. In addition to the closed-source compiler, *nvcc*, NVIDIA provides a closed-source disassembler called *cuobjdump*, which can translate the binary into SASS assembly code. However, NVIDIA offers no means of translating the SASS to binary, leaving researchers unable to make use of the assembly code. Our work fills this gap.
2.2.1 Assembler Generating Framework

We show our assembler generating framework in Figure 2.1. We first use `nvcc` to generate binary code from source programs, and then we use `cuobjdump` to retrieve the SASS assembly, as well as the mapping between the assembly and binary. Our ISA Analyzer will automatically generate variants of given \{(assembly, binary)\} pairs to enrich the data set for analysis.

Next, our ISA Analyzer starts analyzing the binary encoding of different components of each type of instructions, for instance, which bits correspond to opcode, operands, modifiers, etc. The ISA Analyzer yields a list of recorded operations, that is, the set of decoded assembly/binary instructions. An example of the recorded IADD instruction for computing capability 3.5 is shown in Figure 2.2.

Our ISA analyzer exploits the fact that there is one-to-one mapping between each SASS assembly instruction and each binary instruction in the listing generated by the disassembler `cuobjdump`. An example of cuobjdump’s output is shown in Figure 2.3: every assembly instruction corresponds to one 64-bit binary instruction (represented as a hex number). By retrieving enough \{(assembly, binary)\} pairs (subsection 2.3.2), we can decode the instruc-
Our Assembler Generator takes as input the binary encoding information generated by our ISA Analyzer, and outputs an Assembler. The generated assembler can convert assembly code to binary code with respect to the underlying GPU architecture. This corresponds to the conversion from assembly to binary in the compilation process shown in Figure 2.1.

Performing the decoding analysis manually is a slow and arduous process, especially considering the frequency with which the GPU ISA changes - usually once per GPU generation. The framework we developed is important as it can automatically decode the ISA and generate assemblers.

Furthermore, we incorporated our assemblers into a larger framework that permits analysis and transformation of assembly code for multiple GPU architectures, enabling several applications which we will discuss in Section 2.5.

Figure 2.3: An example of output from cuobjdump, compiled for NVIDIA’s Compute Capability 3.5 architecture. We have omitted certain details such as instruction address for illustration purpose.
2.2.2 Unique CUDA Binary Instructions & Behavior

In this section we describe the unique features of GPU ISA compared with CPU ISA. We summarize the differences from the following aspects: memory instruction, control flow, dependence handling, register shuffling, barrier setup, and compile-time instruction scheduling.

Table 2.1: Common memory instructions on GPU. We refer to each general register as Rx or Ry, and each literal as 0xa for illustration purpose.

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDG Ry, [Rx+0xa]</td>
<td>Load from global memory</td>
</tr>
<tr>
<td>STG [Rx+0xa], Ry</td>
<td>Store to global memory</td>
</tr>
<tr>
<td>LDL Ry, [Rx+0xa]</td>
<td>Load from local memory</td>
</tr>
<tr>
<td>STL [Rx+0xa], Ry</td>
<td>Store to local memory</td>
</tr>
<tr>
<td>LDS Ry, [Rx+0xa]</td>
<td>Load from shared memory</td>
</tr>
<tr>
<td>STS [Rx+0xa], Ry</td>
<td>Store to shared memory</td>
</tr>
<tr>
<td>LDC Ry, c[0xa][Rx+0xa]</td>
<td>Load from constant memory</td>
</tr>
<tr>
<td>TEX Ry, Rx, 0xa, 0xa, 0xa</td>
<td>Texture fetch</td>
</tr>
</tbody>
</table>

**Memory Instructions** There are different types of memory on the GPU, each associated with a type of assembly/binary instructions, as shown in Table 2.1. Global memory can be accessed by all threads. Local memory is a range of thread-private memory in GDDR memory. Shared memory is an on-chip memory that is private to each thread block. Constant memory is a type of read-only memory that is split into logical banks. Texture memory is another type of read-only memory, but is only accessed through texture instructions.

**Divergence and Re-convergence** GPU threads are organized into thread-warsps, threads within a warp run the same instruction at one time. When threads inside a warp need to diverge on the execution path, the program uses the SSY instruction to signal the hardware to prepare and specify the address for re-convergence. Threads that take different execution paths will be serialized instead of running in parallel. When every thread in the warp has reached a re-convergence command - either a .S modifier or a SYNC instruction,depending
Figure 2.4: A simple example of thread-warp divergence. (a) is the source and assembly code, (b) is the control flow graph.

on the architecture - it will wait until the thread warp reaches the instruction whose address is specified by the SSY instruction, and then return to running in lock-step.

We show an example of divergence and re-convergence handling in Figure 2.4. The source code and assembly code is on the left, organized into basic blocks. The control flow graph is on the right, showing the paths the thread-warp may be split across. The SSY instruction in basic block BB1 indicates that threads may take different paths (diverge), and the instruction with 0x238 (BB6) is where they will re-converge. The conditional SYNC instruction causes some threads to immediately prepare for re-convergence, but others in the same thread-warp execute basic block BB2 instead. In BB2, the already divergent threads hit the next SSY, preparing for another potential divergence; some take the conditional branch to BB4, whereas others execute BB3 instead. At the end of BB3 and BB4, these doubly-divergent threads re-converge in BB5. At the end of BB5, they finally re-converge in BB6 with the threads that only executed the first SYNC from BB1.

Warp-Register Shuffling  Thread-warps are allocated consecutive physical registers, aligned to hardware-specified boundaries. Compute Capability 3.0 introduced the SHFL instruction, which can read registers from another thread in the same thread-warp. This permits a
thread-warp to perform internal communication far faster than would otherwise be possible, thanks to the low latency of the register file as compared to other memory types.

**Barrier Instructions** The BAR instruction is be used for synchronization, halting the thread until every other live thread in the same thread-block has reached its position. In Compute Capability 3.0 the TEXDEPBAR instruction was introduced, which halts the thread until specified texture operations are completely resolved; texture dependences are thus handled by the compiler, rather than determined by the device.

**Compile-Time Scheduling** As of Compute Capability 3.0, instruction scheduling is handled by the compiler rather than by the hardware. On this architecture every 8-th instruction, rather than being a real instruction, is a set of scheduling codes inserted by the compiler. These scheduling codes dictate the minimum number of cycles that the thread must wait between every two consecutive instructions in the following seven instructions in order to satisfy dependence constraints.

Starting with Compute Capability 5.0, NVIDIA moved even more control logic away from the hardware, saving power and space. Thus instruction-level barrier has been added to the scheduling codes generated by the compiler. The scheduling codes on Compute Capabilities 5.x and 6.x occur in place of every fourth instruction. As of Compute Capability 7.0, they are embedded into each individual instruction, rather than controlling larger blocks of instructions.

### 2.3 Assembler Generation

The ISA for NVIDIA GPUs changes in almost every major version. Prior works [36] [88] [22] [41] have analyzed the ISA, but only for a particular architecture. To ensure forward compatibility as the ISA continues to change across generations, we develop a generic method for decoding the instruction set and create a system which can generate GPU assemblers for newer architectures in a partially automated manner. As a result, we
can quickly add compatibility for additional architectures to support open-source research work.

Our Assembler Generator makes use of our ISA Analyzer, shown in black background in Figure 2.1. The ISA Analyzer takes in a list of binary and SASS assembly instruction pairs from NVIDIA’s cuobjdump tool, and determines which bits in the binary instruction correspond to which components in the assembly instruction. To ensure we have enough input, we use bit flipping to prepare additional \{assembly, binary\} pairs for further analysis, as described below in subsection 2.3.2. Finally, the Assembler Generator component outputs C++ code capable of converting SASS assembly to binary.

2.3.1 Structure of an Instruction

When decoding an instruction, it is vital to gather and maintain information about every component of the instruction that affects the binary encoding. The first component is the opcode. We treat bits which indicate the expected operand types as part of the opcode. For example, if two instructions are both named IADD, but one of them adds two registers whereas the other adds a register to an integer literal, then we treat them as two distinct operations due to the different encoding.

The second component of the instruction is operands. CUDA operands can be one of the following types such as literal value, register, predicate register, special register, bit-field, memory, constant memory, texture shape, texture channel, and etc. Most of these are encoded with a single value, but memory operands may be represented by up to two values (register and offset), and constant memory by up to three (bank, register, and offset). Many instructions allow for one or more optional unary operations to be attached to their operands: arithmetic-negation, bitwise-complement, absolute value, and logical-negation, each of which is typically encoded as a single bit.

Hexadecimal (integer literal) operands need to be handled specially during analysis, because they are encoded differently depending on the type of instruction. For control
flow instructions the assembly will typically use absolute addresses whereas the binary encoding will use relative offsets. Furthermore, a negative value might be handled by directly encoding a negative value, or by simply enabling a unary arithmetic-negation bit.

Another important component of the instruction is its optional modifiers. Most modifiers are attached to the opcode in the assembly, though there are exceptions, in which one or more of them are instead attached to operands. Some modifiers act as simple boolean values that flip a single bit, but many instructions have more complicated modifiers with a range of possible values. For example, several instructions make use of a two-bit logic modifier, which can be “.AND” if the operation will perform a logical and, “.OR” if the operation will perform a logical or, or “.XOR” if the operation will perform a logical exclusive-or. The GPU does not have separate instructions dedicated to each of these logical operations, so an understanding of these modifiers is valuable.

Certain instructions expect multiple modifiers of the same type, in which case their order has important meaning. For example, the PSETP (predicate set-predicate) instruction uses two logic steps to reduce three predicate registers into a single value. PSETP.AND.OR will apply and and then or, whereas PSETP.OR.AND will do the opposite and has a different encoding. This is why it is necessary to know the type of these modifiers. Another example is the format modifiers which can appear twice in cast instructions, such as the F2F instruction, which casts a numeric value from one floating-point format to another floating-point format. Using our knowledge of the types of each modifier, our Assembler Generator is able to determine whether or not a modifier follows another of the same type, and stores information about the first instance separately from the second instance.

The fact is that every part of nearly every type of instructions uses predictable rules to enforce values on particular bits with respect to the assembly instruction.
2.3.2 Analyzing the ISA

The first task before analyzing the ISA is to gather enough instruction \( \{\text{assembly}, \text{binary}\} \) pairs as input. This is vital, since with little data, it is impossible to determine precisely which bits correspond to which parts of the instruction and which encoding rules are used. We start by gathering multiple executable from real source programs, extracting the assembly and binary mapping with cuobjdump, and giving them to our ISA Analyzer’s parser. We gathered executable from the CUDA SDK [64] and Rodinia benchmark suite [9]. After processing all of the input samples, we expand our data set via our bit flipper module.

Our bit flipper takes the binary instruction of every known operation as input, and outputs variants of each one, which we can inject into an executable in order to extract more assembly code. This is similar to a technique used in [88]. Each variant generated by the bit flipper is identical to the instruction it is based on, except that a single distinct bit has been flipped. These flipped instructions will permit the analyzer to infer how different parts of the instruction are affected by each precise bit. Depending on which bits are changed, a new operation might be generated instead; in this case, we resume bit flipping, repeating the process until the results converge.

An important consideration for the bit flipper is that the disassembler may crash without producing output upon encountering unexpected instructions. Our solution is to narrow the range of bits that are flipped - skipping over most of the opcode bits. An alternate solution would be to disassemble flipped instructions one at a time, so that the disassembler’s crashes do not prevent retrieval of any valid instructions. This alternate solution may lead to more complete data set, but in practice we find that this is unnecessary; our faster implementation is sufficient to prepare an assembler which can reproduce every program we have tried.

The first time the analyzer encounters an operation, it makes the broadest possible assumptions. For the opcode, each modifier, and each unary operation, it assumes that every bit in the binary instruction is important to the encoding (setting every value to True in-
side a boolean array), and records the current values. For each operand, it searches for
every subset bits of the instruction which matches each possible interpretation of each of
the operand’s values, recording the maximum possible matching bit-sequence(s). Upon
encountering another instance of a known operation, it is compared to the recorded data,
narrowing down which bits are actually important to each part of the instruction.

For example, suppose we encounter the two instances of FFMA in Figure 2.5, and are
trying to decode the first operand. During the first instruction the operand is R9 (register
9), so we look for the value 9 (1001 in binary format) in the binary. We identify three
matching bit-sequences of size 10, 5, and 4, respectively starting from bit 2, bit 19, and
bit 59. Therefore we set the maximum size at these locations to 10, 5, and 4, and set the
maximum size for other locations to 0. During the next FFMA, the same operand is R5,
so we look for value 5 (101 in binary format) in the opcode. The position (bit 2) which
we previously marked as having the size 10 (for the bit-sequence length) only has size 8
now, thus we reduce its value to 8. The other two non-zero positions no longer contain the
operand’s value, so we reduce them each to 0. At this point, we have correctly identified
that the first register in FFMA assembly instruction controls exactly the eight bits which
start at bit 2.

Figure 2.6 shows the structures used in this section’s pseudo-code. The key structs here
are ASSEM, which is used to hold a parsed assembly instruction, and OPERATION, which is
used to maintain analysis of an instruction. The ASSEM struct contains an identifier for the
opcode, a list of operands, and a list of modifier strings. The ASMOPERAND struct, which
Algorithm 1 Binary Instruction Decoding

Global: knownOps is list of known encodings
Input: a is a parsed assembly instruction
Input: binary is binary code corresponding to a
Result: knownOps is updated

1: procedure ANALYZEINST(ASSEM a, bool[] binary)
2: OPERATION op ← knownOps.lookup(a.opcode)
3: if op == null then
4: op ← knownOps.insert(a.opcode)
5: op.opcodeBinary ← binary
6: op.opcodeBits ← \{true, true, ..., true\}
7: for each OPERAND ‘oprd’ in op.operands do
8: oprd.comp[0:2].size ← \{64, 63, ..., 1\}
9: for each bit ‘b’ in binary do
10: if binary[b] != op.opcodeBinary[b] then
11: op.opcodeBits[b] ← false
12: for each string ‘m’ in a.mods do
13: if m not in op.mods then
14: op.mods.insertModifier(m)
15: op.mods[m].binary ← binary
16: op.mods[m].bits ← \{true, true, ..., true\}
17: for each bit ‘b’ in binary do
18: if binary[b] != op.mods[m].binary[b] then
19: op.mods[m].bits[b] ← false
20: for each OPERAND ‘oprd’ in op.operands do
21: ASMOPERAND asmOprd ← a.operands[oprd]
22: analyzeOperand(oprd, asmOprd, binary)
23: end procedure

represents the assembly for a single operand, contains the components of the operand’s value (since some memory operands have up to three discrete values) a list of unary operators (such as negation, absolute value), and a list of modifier strings. The OPERATION struct, in addition to lists analogous to those of the ASSEM struct, contains the ‘opcodeBinary’ array which holds the binary code from one instance of the associated instruction, and ‘opcodeBits’ which indicates which parts of opcodeBinary have remained consistent across different instances of the instruction. The Operand struct used by Operation is analogous to the ASMOPERAND struct. The COMPONENT struct used by OPERAND contains the ‘size’
Figure 2.6: Structures used in our Algorithms’ pseudocode.

**Algorithm 2** Binary Operand Decoding

**Input:** oprd is an operation’s operand  
**Input:** asmOprd is an operand’s assembly  
**Input:** binary is an instruction’s binary  
**Result:** oprd is updated

1: **procedure** ANALYZEOPERAND(OPERAAND oprd, ASMOPERAND asmOprd, bool[] binary)

2:   **for** each COMPONENT ‘comp’ in oprd.comps **do**

3:     **for** each bit ‘b’ in comp **do**

4:       **for** size = comp.size[b] to 0 **do**

5:         **if** binary[b:b+size] == comp.value

6:             comp.size[b] ⇐ size

7:             **break**

8:   **for** each unary ‘un’ in asmOprd.unary **do**

9:     **if** un not in oprd.unary

10:        oprd.unary.insert(m)

11:        oprd.unary[un].binary ⇐ binary

12:        oprd.unary[un].bits ⇐ {true, true, ..., true}

13:   **for** each bit ‘b’ in binary **do**

14:     **if** binary[b] != oprd.unary[un].binary[b]

15:        oprd.unary[un].bits[b] ⇐ false

16: **end procedure**
array, which holds the maximum valid size for its operand component at each possible bit position. The **UNARYFUNC** struct, representing a unary operator, holds the operator type, the binary code from a single instance of the instruction where the operator was present, and the ‘bits’ array which indicates which parts of the binary have been consistent across instances. Finally, the **MODIFIER** struct used by **OPERATION** and **OPERAND** represents a modifier; it includes the modifier’s name, its type, whether it’s the second instance of its type, the binary code associated with a single instance of the instruction in which this modifier was present, and the ‘bits’ array which indicates which parts of the binary have been consistent across instances.

Algorithm 1 and Algorithm 2 describe the ISA analyzer. At a high level, the goal of these functions is to analyze every individual component of the instruction’s assembly, by updating and comparing the instruction’s binary value when these individual component are present.

Algorithm 1 takes an instruction in its assembly format and its binary format. It identifies the corresponding operation in the list of known encodings (by looking up the opcode), or generates a new one with default values if the given assembly instruction is not yet a known operation. It then performs the analysis: examining the instruction’s opcode bits, the predicate guard, and its modifiers, updating the operation’s encoding record accordingly. The helper function in Algorithm 2 is used to analyze each of the instruction’s operands, identifying the bits associated with their modifiers, unary operators, and the components of their values (up to three). An example of the search for the value-components was shown in Figure 2.5, with register operands that have only a single component (the register ID).

By applying the above algorithm to inputs which include the extra instruction variants created by the bit flipper, we obtain the necessary information for the Assembler Generator to understand all of the instructions and values encountered.
2.3.3 Assembler

Once we have provided the assembler generator with sufficient input, we can use it to output an actual assembler in C++ language. Algorithm 3 contains pseudo-code for generating the assembler and making use of the list of encodings yielded by Algorithm 1. It loops through the complete list of decoded operations and creates separate conditional blocks for each operation. The operation list includes, for each component of the instruction, a list of its associated bits and their expected values. For each component of an assembly instruction, Algorithm 3 generates code to set its binary bits appropriately: the opcode bits, each modifier, and so on.

Figure 2.7 shows an example of generated assembler code. The first conditional block executes if the instruction is an IADD (integer addition) operation: it sets the opcode bits based on recorded values for IADD and each of the three operands’ bits correspondingly. Additional code blocks are generated for each type of operations. In the end it sets the conditional guard’s bits, and returns the final binary format.

Our actual implementation is more optimized and complicated than the example in Figure 2.7. We omit those details for illustration purpose. For example, to make the assembler more efficient, we distinguish between operands that are encoded the same for every operand-list and those which differ; the assembler generator only needs to print the shared operands once for the entire opcode, whereas the others need to be printed once per operand-list. Additionally, upon encountering anything unexpected, such as a modifier that the assembler generator does not recognize, our generated assemblers are designed to print an error message to the standard error stream.

2.4 Decoding Summary

In this section we describe the findings we have gathered with the help of our Assembler Generator. Although instructions are of fixed length, NVIDIA’s instruction sets lack
Algorithm 3 Assembler Generator

Input: operations is the list of binary ISA encodings
Result: output the source code of an assembler

1: procedure GENASSEMBLER(OPERATION[] operations)
2:   for each OPERATION ‘op’ in operations do
3:     print “if opcode == ” op.opcode
4:       for each OPERAND ‘oprd’ in op.operands do
5:         for each COMPONENT ‘c’ in oprd.comps do
6:           for each bit ‘b’ do
7:             integer s ⇐ c.size[b]
8:             if s > 0 then
9:               print “binary[b:b+”s “] = c.value”
10:          print “for each MODIFIER ‘m’”
11:         for each MODIFIER ‘m’ do
12:           print “if m.name == ” m.name
13:           print “if seenModType[m.type] == ” m.2nd
14:             for each bit ‘b’ in m.binary do
15:               if m.bits[b] then
16:                 print “binary[” b “] = ” m.binary[b]
17:             print “end if”
18:           print “seenModType[m.type] ⇐ true”
19:         print “end if”
20:       print “end for”
21:     print “else”
22:       print “throw error”
23:     print “end if”
24:   end procedure
procedure assembler (instruction inst)
    if inst.opcode == "IADD" then
        binary = opcode_Table3x[IADD];
        binary[2:9] = inst.operands[0].value;
    else if inst.opcode == "MOV" then
        ...
    end if
    //auto-generated code ends here
    if inst.guard then
        binary[18:21] = inst.guard;
    end if
    return binary;

Figure 2.7: A simplified example of an assembler that has been automatically generated.

the relative simplicity of a RISC architecture. It includes complicated instructions such as multiplication-and-addition, multi-function operation that performs trigonometric functions including sine and cosine, and so on. Although we can make generalizations about which bits are used for which components of the instruction, there are few consistent rules across different instructions.

Table 2.2 shows a set of common instructions used in GPU, and their effects. We use ‘regX’ to refer to 32-bit general registers, ‘pX’ to refer to 1-bit predicate registers, and ‘composite’ to refer to operands with multiple possible types. The mapping between the bit-sequence and the operand/opcode/modifier in the assembly instruction is shown in Figure 2.8, for instance, reg1 bits are 2 to 9 in computing capability 3.x. We have provided a complete table of opcodes for all decoded instructions in Zenodo [30].

2.4.1 Operands

There are several operand types we observe in the assembly code. First, 32-bit general registers, which we usually just call registers, are used in most instructions. Depending
Table 2.2: Common instructions for Compute Capability 3.x, with operands defined in terms of Figure 2.8. Memory operands are denoted with square brackets. PC refers to the program counter, LOP refers to an arbitrary logic operation, and comp is short for composite.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV reg1, comp</td>
<td>reg1 ⇐ comp</td>
</tr>
<tr>
<td>S2R reg1, special_reg</td>
<td>reg1 ⇐ special_reg</td>
</tr>
<tr>
<td>IADD reg1, reg2, comp</td>
<td>reg1 ⇐ reg2+comp</td>
</tr>
<tr>
<td>IMUL reg1, reg2, comp</td>
<td>reg1 ⇐ reg2×comp</td>
</tr>
<tr>
<td>IMAD reg1, reg2, comp, reg4</td>
<td>reg1 ⇐ reg2×comp+reg4</td>
</tr>
<tr>
<td>IMAD reg1, reg2, reg4, comp</td>
<td>reg1 ⇐ reg2×reg4+comp</td>
</tr>
<tr>
<td>PSETP p2, p1, p3, p4, p5</td>
<td>p2 ⇐ p3 LOP p4 LOP p5; p1 ⇐ !p2</td>
</tr>
<tr>
<td>BRA const/lit comp</td>
<td>PC ⇐ const/lit comp</td>
</tr>
<tr>
<td>CAL const/lit comp</td>
<td>callstack.push(PC); PC ⇐ const/lit comp</td>
</tr>
<tr>
<td>RET</td>
<td>PC ⇐ callstack.pop()</td>
</tr>
<tr>
<td>LD reg1, [reg2 + 32-bit lit]</td>
<td>reg1 ⇐ [reg2 + 32-bit lit]</td>
</tr>
<tr>
<td>ST [reg2 + 32-bit lit], reg1</td>
<td>[reg2 + 32-bit lit] ⇐ reg1</td>
</tr>
</tbody>
</table>

on the ISA, these registers’ index is encoded as either six bits or eight bits. The register with maximum ID (either 63 or 255) is the read-only zero register, which always holds a value of zero, and is written as RZ in assembly code. In operations which use 64-bit or larger values, the GPU will use a range of consecutive registers, starting with the register specified in the instruction.

Predicate registers hold boolean values, and are encoded as three bits. Additionally, predicates can be used as conditional guards for most instructions, allowing individual threads to selectively skip execution of instructions. The conditional guard’s binary is in a different location for each distinct ISA, but is always four bits: the lower three bits are the predicate register ID, and the highest bit is used for logical negation. The predicate register with ID 7 is the null predicate, which always holds a value of true, and is written as PT in the assembly code.

Special registers are only used in the S2R instruction, and are encoded with eight bits. In the assembly code they can be written as SRx for some value x, but are more common ly...
Table 2.3: The most common special registers used on GPU.

<table>
<thead>
<tr>
<th>Special Register</th>
<th>Encoding</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR_TID.X</td>
<td>33</td>
<td>Thread ID (x-dimension)</td>
</tr>
<tr>
<td>SR_TID.Y</td>
<td>34</td>
<td>Thread ID (y-dimension)</td>
</tr>
<tr>
<td>SR_TID.Z</td>
<td>35</td>
<td>Thread ID (z-dimension)</td>
</tr>
<tr>
<td>SR_CTAID.X</td>
<td>37</td>
<td>Thread-Block ID (x)</td>
</tr>
<tr>
<td>SR_CTAID.Y</td>
<td>38</td>
<td>Thread-Block ID (y)</td>
</tr>
<tr>
<td>SR_CTAID.Z</td>
<td>39</td>
<td>Thread-Block ID (z)</td>
</tr>
<tr>
<td>SR_CLOCK LO</td>
<td>80</td>
<td>Cycle Counter (32 bits)</td>
</tr>
</tbody>
</table>

used with the English name of the associated value. For example, the special register "SR_CLOCKLO" can be used to retrieve the lowest 32-bits of the clock value, and the special register "SR_TID.Y" can be used to retrieve the current thread’s ID along the thread block’s y-dimension. Table 2.3 shows common special registers and their encoded values. The encodings for most special registers remain the same across GPU generations, though additional special registers are added over time. We have provided a more complete list of special registers in Zenodo [31].

Memory operands have a register and a literal offset, regardless of whether they are used for global memory, local memory, or shared memory. The literal offset is usually either 24 bits or 32 bits, depending on the instruction.

Constant memory operands point to a type of read-only memory. There is a dedicated load-constant-memory (LDC) instruction, but constant memory operands are also usable in other instructions. They have up to three components: a memory bank, a register, and a literal offset. The bank and offset use a combined 19, 20, or 21 bits for their encoding, depending on the ISA and the instruction. When using 19 or 21 bits, the highest 5 are the bank, and the rest are the offset. When it uses 20 bits, the lowest 16 are the offset, and the highest four and lowest one are the bank. The register component, which is added to the offset, is only used in the LDC instruction and some control-flow instructions.

Texture operations have a texture operand, whose value is a shape such as 1D, 2D-Array, CUBE, etc, and is encoded as three bits. On some architectures, texture operations
also have a channel operand with some combination of "R", "G", "B", and "A", encoded with up to four bits.

Barrier operations may use an SB operand (written as SBx for some integer x) and a bitfield operand (written as a set of bit indices).

Finally, instructions may use literal, numeric values of various sizes. In assembly code these are usually written as hexadecimal values, though in floating-point operations they may be written in decimal instead. Notably, the encoding for floating-point literals does not quite match the IEEE Standard, due to lack of enough bits - for example, in some instructions 19 bits may be used to try to hold a 64-bit double-precision floating point value. These floating point values are handled by discarding the lowest bits until the remaining portion fits in the available space.

We use the term ‘composite’ to refer to a single operand that has multiple possible types. On Compute Capabilities 2.x and 3.0, the most common composite operand is 20 bits; it can either hold a 20-bit literal, a 6-bit register ID, or 20-bit constant memory location without a register component. We have also seen 16-bit composites that can either hold a literal or a register.

On Compute Capabilities 3.x through 6.x, the most common composite is 19 bits; it can either hold a 19-bit literal, an 8-bit register, or a 19-bit constant memory location without a register component. We have also seen 24-bit and 32-bit composites which can hold either a literal of that size or a 21-bit constant memory location without the register component.

In Figure 2.8, we show common locations and sizes for different operands in each of the architectures that we have studied. In Zenodo [28] we have provided the operands for specific instructions that have been decoded. In Table 2.2, we show a subset of this data, with several common instructions for version 3.x of the ISA. We refer to general registers as regX and predicate registers as pX with integer values x, and texture shapes as tex.
2.4.2 ISA Change Over Time

Compute Capability 2.x  The earliest hardware we consider is NVIDIA’s "Fermi" generation, which includes devices with Compute Capabilities of 2.0 and 2.1. Instructions can be either 4 bytes or 8 bytes, but we have only seen the compiler generate 8 byte instructions. We find that the executable uses little-endian format for instructions, but CUDA’s disassembler instead displays the hexadecimal values of each instruction starting with the most significant byte. This architecture handles instruction-level scheduling via the hardware.

Compute Capability 3.0  The earliest of NVIDIA’s "Kepler" generation has Compute Capability 3.0. For this architecture, the ISA is very similar to the previous one, with every pre-existing instruction having exactly the same binary encoding as before, though some additional instructions have been added.

The major difference between the ISA of this architecture and the previous is that some

Figure 2.8: Common operands on different CUDA architectures.
instruction scheduling has been offloaded from the hardware to the compiler. For Kepler devices, every eighth instruction (including the first one) is not a real instruction, but instead contains the instruction scheduling information for GPU. [36] developed an assembler for Compute Capability 2.x, but also explored this architecture due to its similar ISA. The aforementioned latency instructions have no official name of which we are aware, but Asfermi used the name SCHI to represent them, which we also use in this work.

We refer to the least significant bit as bit 0, and the most significant bit as bit 63. Bits 0-3 of the SCHI instruction contain the value 7. Bits 4-11 control the minimum number of cycles that the GPU must wait between dispatching the first following instruction and the second. Bits 12-19 control the minimum cycles between the second following instruction and the third. This continues for the next five groups of eight bits. Finally, bits 60-63 hold the value 2. The seven dispatch values can hold a value of 0x4 when the associated instruction is able to be dispatched in the same cycle as the next, or a value between 0x20 and 0x3f to mean the minimum number of cycles between them is the value minus 15.

Compute Capability 3.x The remainder of the Kepler generation comprises Compute Capabilities 3.2, 3.5, and 3.7. Unlike the earlier Kepler devices, these GPUs allow threads to use up to 256 registers instead of 64, so the six bits which the previous ISA’s instructions used to indicate the register are insufficient. Therefore, although the assembly code looks much like that of the previous generation, every instruction has a new encoding. A SCHI instruction now holds the value 0 in its two least significant bits and the value 2 in six most significant bits - the seven dispatch intervals have been shifted two bits, but are otherwise unchanged.

The manner in which the disassembler presents SCHI instructions on Kepler is not conducive to analysis or editing. It provides the 64-bit binary value of the entire SCHI, but offers no indication of its meaning. Furthermore, since the SCHI describes a group of consecutive instructions, addition, deletion, or reordering of instructions becomes more
Figure 2.9: An example of how we extract the scheduling information for each group of seven instructions on Kepler GPUs. These 8-bit values indicate dispatch behavior. The compiler has determined that these stalls are necessary according to the latency of the instructions involved.

Compute Capabilities 5.x and 6.x  The ”Maxwell” and ”Pascal” generations, which include devices with Compute Capabilities 5.0, 5.2, 5.3, 6.0, 6.1, and 6.2, use a different ISA than the previous generations of NVIDIA devices, with the opcode contained in bits 52-63.
or two rarely-used flags. But in Maxwell and Pascal, they have been extended to control barriers and the register cache. Every fourth instruction on these architectures is a SCHI, so they each affect the following three real instructions. Different from previous ISAs, the binary for a SCHI instruction no longer has any bits dedicated to its opcode, and is therefore identified purely by its position in the code.

As with the previous generation, NVIDIA’s dissembler outputs each instruction group’s SCHI data only as a single, combined hexadecimal value. As such, we again break up the SCHI data, in-lining values with individual instructions so that the code can be easily understood and edited. Figure 2.10 shows how we extract the separate values, and how they line up with the following three instructions. The 21 SCHI bits associated with each instruction are distributed as follows. Bits 0 through 3 are the minimum number of cycles to wait before dispatching the next instruction. According to [22], bit 4 is a ”yield hint flag”, which encourages the GPU to switch to another thread. Higher dispatch values in the lowest bits will fail without the yield hint. Bits 5 through 7 and 8 through 11 each indicate a barrier to set after the instruction, or have value 7 when no barrier needs to be set. Bits 12 through 17 indicate which combination of the six barriers the thread must wait for before dispatching the associated instruction. We skip the remaining four bits, which the disassembler already in-lines as cache ”reuse” modifiers attached to register operands.

Figure 2.10: An example of how we extract the scheduling information for each group of three instructions on Maxwell and Pascal GPUs. The first and second values indicates dispatch behavior. The third and fourth indicate which barriers to set. The fifth indicates which barriers to wait for.
Although this architecture allows instructions to set two barriers, trying to set an unexpected barrier can cause erroneous behavior. The first barrier slot is used to handle true dependencies, for variable-latency instructions that have a destination register (e.g. loads); the second barrier slot is used to handle anti-dependencies, for variable-latency instructions that have source registers (e.g. stores). If an instruction depends on a barrier, the thread will wait for every instruction that set that barrier.

In the case of Figure 2.10, the thread must wait at least 3 cycles after the first instruction. Upon dispatching the second instruction, it sets write barrier #1, and then must wait at least 13 cycles. Since the lowest two bits are set for the barrier wait bit-field, the thread waits for both barrier #0 and barrier #1 to be released before dispatching the third instruction, and after dispatch it waits at least 6 cycles.

**Compute Capability 7.x** The “Volta” generation, which includes Compute Capabilities 7.0 and 7.2, has another new ISA. In this generation, instructions have been expanded to 16 bytes in size, with scheduling information embedded into each individual instruction. We have not completely decoded this ISA yet, but it can be decoded with similar methods as the previous architectures.

### 2.5 Applications

There have been several works which interact with the binary-level GPU code, but due to the closed-source nature of NVIDIA’s ISAs, they have been forced to make use of third-party projects. Our work is designed to support as many of these ISAs as possible, and has already been used by authors of [26], [27], and [48].

Our framework uses Flex [68] and Bison [16] to parse the assembly code, generating an intermediate representation (IR) in which the precise version of the ISA is largely irrelevant. Applications can directly make use of our framework as a front-end and back-end, operating on this IR. For example, if an application needs to convert local memory operations to
shared memory operations, it can simply scan the IR for local memory instructions, change each one’s memory type, and change their memory addresses as necessary, as shown in Figure 2.11.

In Figure 2.11, (a) shows Compute Capability 3.x binary code in hexadecimal format, (b) shows the corresponding assembly code extracted with our front-end, (c) shows the result of modifying the memory instructions, and finally (d) shows the new binary generated by our assembler. Our framework already performs each of these steps except the transformation from (b) to (c).

Our framework can serve as an architecture-independent infrastructure for various applications described below. Unlike existing assemblers, it can provide them compatibil-
ity with multiple generations of GPUs. Our framework is available at https://github.com/decodecudabinary/Decoding-CUDA-Binary.

**Compilation** CUDA developers are currently reliant on NVIDIA’s closed-source software for compilation. With access to the instruction encoding, however, the development of an open-source GPU compiler is possible. This would allow researchers to easily explore various compiler optimizations, such as new algorithms for GPU register allocation.

**IR Generation** Our framework enables versatile control flow analysis by leveraging our intermediate representation. When we parse the assembly into its IR, we organize the instructions into basic blocks. We convert branch targets from literal offsets to pointers, and break up instruction-scheduling values into their associated instructions. This organization of the code results in human-readable assembly, allowing developers to better understand a program, and facilitates techniques such as binary instrumentation.

**Binary Instrumentation** Once we organize the assembly into basic blocks, code can easily be inserted or deleted, with scheduling data placed automatically. In [27] our framework was used to perform GPU binary instrumentation, in order to protect otherwise vulnerable regions of memory. In Figure 2.12, we show a simple example: (a) is a snippet of raw assembly code, (b) is human-readable assembly generated by our framework, and (c) is modified assembly that has been instrumented to clear some registers before exiting the kernel.

Binary instrumentation can be done even without access to a program’s source code. Furthermore, since our IR is not tied to a single version of the ISA, changes to the code can be compatible with many architectures, using our generated assemblers to target different devices as needed.

### 2.6 Related Work

There are several works which interact with the binary code, but are limited to specific architectures, and so could benefit from our framework in order to support additional de-
Figure 2.12: Example of instrumenting the code to clear some registers before exit.
vices. In GPES [89], the authors transform the binary to enable workload partitioning to be performed in a manner totally transparent to applications - though they state that more sophisticated binary analysis is still needed to support complex kernels. [47] proposes a synchronization scheme which uses binary-level instructions to achieve greater efficiency than other lock-based approaches. In [59], the KernelGen framework employs several instructions that exist only in the binary ISA, modifying the code and other parts of the ELF.

In [36], the Fermi generation (Compute Capability 2.x) was examined. In [88], the late Kepler generation (Compute Capability 3.x) was explored to optimize the SGEMM application. [22] reverse-engineered Maxas (Compute Capability 5.x), and [41] looked at Volta (Compute Capability 7.x). But to the best of our knowledge, ours is the first work that strives to provide an assembler generating framework in order to support as many NVIDIA architectures as possible.

2.7 Conclusion

The closed-source nature of NVIDIA’s ISA restricts the capabilities of developers and researchers. Sufficient knowledge of the ISA enables better optimization and security, as well as allowing various applications including register allocation and binary instrumentation. The assembler generation technique we developed permits rapid decoding of the machine code for various GPU generations. Furthermore, the framework we built makes these assemblers easy to use for architecture-independent applications. Several works [32] [26] [27] [48] have utilized our framework.
CHAPTER 3
UNIFIED ON-CHIP MEMORY ALLOCATION FOR SIMT ARCHITECTURE

3.1 Overview

Existing compilation techniques for on-chip memory resource allocation, including register allocation, mainly target single-thread performance. In the past several decades, efficient techniques have been studied and widely adopted in mainstream compilers. In the context of single instruction multiple threads (SIMT) architecture for GPUs, the whole program performance not only depends on single thread performance, but also the interaction between the group of threads that run concurrently – mainly the process to hide each other’s latency caused by control/data dependence. The number of concurrent threads depends on the physical on-chip memory constraint as well as the per-thread on-chip memory demand from a given program. The latter mainly depends on compile-time decision. The traditional register allocation technique for CPU program tends to gives the maximal number of physical registers to a single thread according to its register pressure. This is because there is no dynamic partitioning of registers among concurrent threads for CPUs.

The goal of traditional register allocation technique is then to minimize the number of register spills and maximize single thread performance. However, this strategy does not necessarily work well for programs running on SIMT architecture. Allocating registers according to a single thread’s register pressure may lead to resource contention among concurrently executing threads and lead to sub-optimal performance. We show this phenomenon using the results of a case study over a set of important GPU applications in physics simulation, numerical analysis, and image processing [9] [64]. We control per-thread register count at compile-time and we compile one program into different versions over a range of register usage from 20 or $32^{1}$ to the maximal register demand. We show the

---

$^{1}$We choose 20 for Fermi and 32 for Kepler because this leads to the maximal number of concurrent
Figure 3.1: Performance of compiled programs with various per-thread register usage. The $x$-axis represents the per-thread register usage for every compiled version. The $y$-axis shows the running time in seconds. We used an NVIDIA Kepler [66] GTX680. The range of register count is from 32 to the maximal register demand.

results of four benchmarks *heartwall*, *particles*, *dxtc* and *FDTD3d* in Figure 3.1. As can be seen in Figure 3.1, using as many registers as needed to avoid spilling does not necessarily yield optimal performance. Furthermore, using the smallest number of registers per-thread does not necessarily yield best performance either. For instance, the *dxtc* benchmark’s best per-thread register count is 40, which is too high to allow maximal occupancy, yet lower than the register demand.

A fundamental question arises: what is the best per-thread register usage for a given program on a given GPU architecture? An intuitive approach is to try all possible register count, compile the program and profile performance over different runs. However, exhaustive search is prohibitive when the range of possible register counts is large. It can be up to 255 for current GPUs[66]. Furthermore, depending on the input parameters, the number of profiling runs needed for a fully representative data set can be exponential.

To have a good answer to this fundamental question, we need to understand of the implications of compile-time on-chip memory allocation decisions on the efficiency of concurrent execution. Per-thread register usage, as well as other on-chip memory usage, is tightly correlated with the concurrency level a program can achieve. Using fewer registers per thread may lead to high concurrency, but more local memory loads and stores due to register spills. The local memory resides in DRAM, which has large access latency. How can threads. Having a smaller per-thread register count does not improve concurrency; it only degrades single thread performance, leading to worse performance overall.
we minimize per-thread register usage while maintaining good single-thread performance? And how can we strike a balance between the benefits brought by high concurrency and the overhead brought by extra off-chip memory operations?

In this chapter, we study the implications of concurrent execution on many-core GPUs and exploit these implications to develop efficient compile-time on-chip memory allocation strategies. We address the above challenges from two main aspects. We first propose a unified on-chip memory allocation framework that not only uses registers but also on-chip scratch-pad memory, to store thread context. The scratch-pad memory acts as a buffering layer between registers and off-chip memory, alleviates single-thread register pressure, and increases the concurrency level. We develop a novel inter-procedure scratch-pad memory allocation scheme that maximizes reuse across procedure boundaries, and implement a prototype on-chip memory allocator. To ensure compatibility, we only use scratch-pad memory not already allocated by the user. Secondly, we characterize the relationship between the program performance and the concurrency level. Our characterization model predicts the desired concurrency level for a given program, and guides the selection of per-thread register count and scratch-pad memory usage.

There is a large body of research work on register allocation for CPUs and embedded processors [7] [8] [70] [24] [67] [3] [11]. They have shown promising results for single-thread applications. Some studies investigated scratch-pad memory allocation techniques on embedded architectures [15] [57] [76], but not on GPUs equipped with a bulk synchronous parallel (BSP) execution model. Gebhart and others [20] proposed techniques to dynamically partition on-chip memory into cache, scratch-pad and register memory according to application’s register/scratch-pad memory demand with architecture support. However, it does not address the problems of how to reduce and how to determine register pressure for a given program. Overall, there is a lack of exploration in the implications of compile-time on-chip memory allocation on the concurrent execution efficiency of GPU applications.
In this chapter, we propose efficient on-chip memory allocation techniques to enable maximal utilization of many-core GPU processors. We summarize our contributions as follows:

- **On-Chip Memory Allocation** We build an unified on-chip memory allocation framework for GPU applications. We offload register pressure to scratch-pad memory when necessary and we determine the corresponding per-thread register and scratch-pad memory usage for maximal concurrency level. Under this framework, we develop a novel inter-procedure on-chip memory allocation strategy, which maximizes the reuse of on-chip memory across procedure boundaries.

- **Concurrency-oriented Program Analysis** We reveal that severe resource contention can be caused by static memory resource allocation for GPU programs. For the first time, we address the problem of mapping GPU program features to its achievable concurrency and its desirable concurrency level. We propose efficient characterization model to determine if increased concurrency level will always yield better whole program performance. Our model is a pure static model and yet it is effective.

- **Implemented Allocator for Real GPU Systems** We reverse engineered the NVIDIA hardware ISA and implemented our prototype on-chip memory allocator for programs that run on real GPUs. Our approach can be readily deployed and does not require any architecture level extension.

The rest of this chapter is outlined as follows: Section 3.2 presents our unified on-chip memory allocation framework. Section 3.3 describes program characterization and concurrency selection techniques. Section 3.4 and Section 3.5 respectively present evaluation results and related work.
3.2 Unified On-Chip Memory Allocation

3.2.1 Framework Overview

In this Section, we describe our transformation framework that uses shared memory to store local variables and to alleviate register pressure, which ultimately leads to better GPU concurrency and whole program performance. This framework uses shared memory to store live variables that cannot fit in registers, as if we are spilling registers into shared memory. Given fixed numbers of registers and shared memory slots, our transformation framework place live variables into on-chip memory storage. We perform register allocation first and select the variables that can stay in registers. For the rest of the variables, we perform shared memory allocation and choose a subset of them to be stored in shared memory.

The target of the unified on-chip memory allocation is to store as many local variables into a fixed number of registers and shared memory slots as possible. We treat both registers and shared memory as one type of memory – the on-chip memory. Then we perform on-chip memory allocation as if we are performing register allocation for traditional CPU programs. We illustrate this idea in Figure 3.2. Assume we have two registers, one shared memory slot, and one local memory slot. In Figure 3.2(a), we show the interference graph of five local variables \(v1, v2, v3, v4, v5\). Two variables interfere with each other if they are both live at one or more instructions. It implies the two variables cannot be assigned to the
same register or shared memory slot. If two variables interfere, there is an edge between the nodes representing them. In Figure 3.2(b), we show the result after register allocation. Variables $v1$ and $v2$ are assigned to registers $r1$ and $r2$ respectively. Now we have three variables that are not assigned and we have one shared memory slot. In Figure 3.2(c), we assign variables $v2$ and $v5$ to shared memory slot $s1$. By this step, we have completed assigning as many variables as we can to registers and shared memory. We then let the last variable $v4$ stay in local memory. The minimal number of variables to be stored in local memory is 1 in this case.

Register allocation techniques have been extensively studied in the past three decades [7] [8] [70] [24] [67] [3] [11]. However, they mainly focus on single procedure register allocation. A few of them [53] [12] have studied reuse of registers across procedures but mainly focus on minimizing register pressure penalty at procedure calls. Typically, most of the registers in the caller procedure are saved in local memory at procedure calls so that they can be reused in the callee procedure. Previous work [53] [12] avoid saving all the registers when procedure calls happen by determining if the registers will be used or not in the callee procedure. We leverage the register allocation algorithms for single procedure on-chip memory allocation and we develop an algorithm that maximizes reuse of on-chip memory across procedure boundaries. We describe this approach in subsection 3.2.2. We illustrate this approach by applying it to shared memory allocation. However, the same approach applies to registers as well.

In summary, with a given number of registers and shared memory, our unified on-chip memory allocation framework performs both register allocation and shared memory allocation. To separate the coupling effects from other phases of compilation, we build a experiment platform that takes binary as input. As in the binary file, the other phases like instruction scheduling have already completed, and we can simply replace the live variable accesses as shared memory access or off-chip memory accesses and add corresponding instructions. Therefore, the only effect we are testing is the placement of live variables.
We use the binary generated by *nvcc* with a fixed register count. Then we analyze the other variables that are spilled into local memory and transform them correspondingly given a fixed number of shared memory slots. The NVIDIA GPU instruction set architecture (ISA) and application binary interface (ABI) is proprietary. We reverse engineered part of the ISA and ABI for CUDA computing capability 3.0 with information from the open source project *asfermi* [36] on CUDA computing capability 2.0. We are able to decode the instructions, parse the assembly code and perform data flow analysis. In the following Section, we elaborate inter-procedure shared memory allocation in subsubsection 3.2.2.

### 3.2.2 Shared Memory Allocation

*Inter-procedure Shared Memory Reuse*

We start describing our technique on enhancing inter-procedure reuse of shared memory with an example. Note that CUDA does not allow objects with virtual functions to be parameters, and every CUDA kernel we have seen has a call graph which can be determined statically.

We first show that there are opportunities to reuse shared memory slots across procedure calls. In Figure 3.3, we show the stack status of a call sequence that involves three procedures: `proc_A`, `proc_B`, `proc_C`. `proc_A` calls `proc_B`, `proc_B` calls `proc_C`. Assume the stack memory space for every procedure can hold exactly four different variables. This means at any instruction in this procedure, at most four variables can be live at the same time. The variables that are live when `proc_A` calls `proc_B` are stored in locations `La1`, `La3`. In `proc_B`, when `proc_C` is called, variables are saved in locations `Lb2` and `Lb3`. In Figure 3.3, we first show what the call stack looks like with traditional CPU procedure local memory management – labeled as *traditional*. The cells with dark background represent that the variable in the corresponding location is live when another procedure is called. With the traditional approach, we can see that the local memory space of different procedures in the call sequence is stacked. Therefore, for these three procedures, the size of the
local memory space needed in the call context is 12 slots, assuming one slot can hold one variable. Even if some variables are not live when a procedure is called, the memory stack is incremented from its original maximal stack depth. This is because the size of local memory for a CPU procedure is trivial compared to the size of the off-chip memory.

In the second approach described in Figure 3.3, we show a CPU architecture that utilizes a register stack when an architecture has relatively large register space to hold callee-saved registers across multiple procedures [11]. This is only made possible with special architecture support [11]. This architecture utilizes the available register region at the end of the register stack when a procedure is called. In the subfigure marked as Register Stack Architecture in Figure 3.3, proc_B can reuse the last available slot in proc_A’s stack, and its stack pointer starts from the end of La3. Similarly, proc_C can reuse the last slot in the stack for proc_B. In this case, we use 10 slots in the stack. This approach is related to inter-procedure register usage, but it can be applied to shared memory allocation across
procedure boundaries. It saves 2 slots compared to the approach denoted as *Traditional* in Figure 3.3. However, there are still slots that are not used when proc\_B and proc\_C are called.

In the third approach described in Figure 3.3, we show our approach – *Moving Stack* approach, which minimizes unused stack space when there are nested procedure calls. Our resource allocator emits instructions to be inserted in the original binary, which shuffles variables in the stack so that the used variables will be stored in consecutive memory space. Then we emit code to shift the stack pointer before the callee procedure is invoked. For instance, in Figure 3.3 *Moving Stack* section, when proc\_B is called, the variable in La3 is shuffled to the second slot. Then we let proc\_B use the space from La3. Similarly, when proc\_B calls proc\_C, we move the variable in Lb3 to Lb1. Therefore proc\_C can use the third slot in proc\_B’s stack space, which is the 6th slot in the overall runtime stack space. In this case, we use 8 slots in total and no local memory slot is wasted. Compared to the original case that uses 12 slots, we save more than 30% stack space.

*Inter-procedure Shared Memory Assignment*

In the last section, we presented an approach to maximize the reusability of shared memory slots across procedures. If we have a large amount of shared memory to hold all local variables, then we can directly start assigning shared memory slots to individual variables. However, the shared memory is a scarce resource, as its size is the same or even smaller than the register file size. Therefore, we need to select a subset of local variables to reside in shared memory. Meanwhile, we need to determine how many shared memory slots every procedure gets assigned. Then we can perform shared memory assignment on a per-procedure basis.

In this Section, we describe our approach to map selected local memory variables to shared memory variables. We define *Live-on-exit* to be set of variables live at the exit of an instruction. *Max-live* is defined as the maximum number of simultaneously live variables at
the exit of an instruction. \textit{Max-live} of a procedure that does not call any other procedure is easy to acquire. We can traverse all instructions in the procedure and pick the largest \textit{Live-on-exit} set. For procedures that call other procedure calls, we propose a recursive approach built on the following idea. We obtain the number of live variables for an instruction that calls another procedure $P_{\text{callee}}$ as the sum of its local —\textit{Live-on-exit}— and $\text{Max-live}(P_{\text{callee}})$. If the \textit{Max-live} of the callee procedure is unknown, we recurse into the callee procedure to find its \textit{Max-live}.

Assume we have $N_{\text{smem}}$ available shared memory slots. If \textit{Max-live} of the main GPU kernel function is greater than $N_{\text{smem}}$, we need to prune at least $\text{Max-live} - N_{\text{smem}}$ variables from the \textit{Live-on-exit} sets and let these variables reside in local memory. Our heuristic approach ranks different variables based on a pre-defined priority function. We prune low priority variables until the updated \textit{Max-live} is less than or equal to $N_{\text{smem}}$. This approach is simple, yet effective.

We rank local variables from different procedures and give them a global ranking. We first define a \textit{composition instruction}. It is a list of 2-tuples used to specify a call sequence. If the instruction \textit{inst}_2 at \textit{func}_0 calls \textit{func}_1, and instruction \textit{inst}_3 at \textit{func}_1 calls \textit{func}_2, and the specific executing instruction in \textit{func}_2 is \textit{inst}_0, then the resulting composition is \{(\textit{func}_0, \textit{inst}_2), (\textit{func}_1, \textit{inst}_3), (\textit{func}_2, \textit{inst}_0)\}. The call context information exposed in a composition instruction helps keep track of caller instructions so that we can obtain the \textit{Live-on-exit} set easily from a union of live variables at all relevant instructions in this calling context. Then we can compare these variables from different procedures as if they are from the same procedure.

Our inter-procedure variable pruning algorithm takes the following steps:

- **Step 1:** We find the set of all composition instructions whose \textit{Live-on-exit} > $N_{\text{smem}}$. We call it the \textit{Over-smem-limit} set.

- **Step 2:** For all live variables in the union of \textit{live-on-exit var} sets of composition instructions in the \textit{Over-smem-limit} set, we compute their priority values based on
the priority function. We use the priority function of variable frequency in the union live variable set.

- **Step 3:** We eliminate one variable from the above set with lowest priority value and check whether $Max\text{-}live$ is less than or equal to $N_{smem}$ after this variable is eliminated from all $Live\text{-}on\text{-}exit$ sets. If it is, then we go to Step 4. Otherwise we go back to Step 3.

- **Step 4:** We have successfully pruned all the necessary variables. We return the set of variables that are candidates to be placed in shared memory.

**Individual Shared Memory Slot Assignment**  The eliminated variables are the ones that stay in local memory and the rest are mapped to shared memory. With this information, we can compute up-to-date $Max\text{-}live$ for every procedure again. This is used as the maximal number of shared memory slots assigned to every procedure. Then we perform shared memory slot assignment in a way similar to register allocation. We use a heuristic graph coloring approach that starts with the node of highest degree in the interference graph. We assign this node a shared memory slot that does not conflict with any of its neighbors that are already assigned. If there are multiple choices, then we choose the shared memory slot that was previously assigned to some other variable. We process every node. If a variable cannot be assigned to any shared memory slot without conflicting with its neighbors, we map it to local memory. If the interference graph has a chordal property, then we will not have any spills [67]. In most cases, we don’t need to spill any shared-memory mapped variable into local memory.

### 3.3 GPU Program Occupancy Characterization

Our transformation framework in Section 3.2 tackles the problem of minimizing local memory spills given a fixed amount of registers and shared memory. What would be the best amount of registers and shared memory to allocate for every running thread in any
given GPU program? Given a typically much larger number of registers than on CPUs, usually in the scale of tens of thousands, we have many possible combinations of register count and shared memory consumption per thread. In this large search space, exhaustive search is prohibitive. In this section, we address the problem of finding best per-thread register and shared memory usage.

The number of registers and the amount of shared memory used per-thread determine the number of concurrent threads on every streaming processor. The number of concurrent threads can be estimated using the formula below:

$$Active.Thread = \min\left(\frac{Total.Reg.Num}{PerThread.Reg.Num}, \frac{Total.Smem}{PerThread.Smem}\right).$$

Essentially, the specific questions on per-thread register and shared-memory usage all boil down to one fundamental question: what would be the most desirable concurrency level for any GPU program on a specific GPU architecture? If we know the best concurrency level, we can estimate per-thread register and shared memory usage by solving the above equation. The optimal concurrency level has the capability of overlapping different types of operations and minimizing computing unit idleness. The number of different operations and how much they can be overlapped depends on the characteristics of a program. The problem of desired concurrency level is thus closely related to the problem of GPU program characterization in a many-thread cooperation/contention context. We describe our characterization approach first and concurrency level determination algorithm secondly.

---

\(^2\)The total number of threads is also bound by the register bank alignment and the thread block sizes for CUDA programs. This formula illustrates the idea that per-thread register/shared-memory usage dominates the number of concurrent threads. We use the GPU occupancy calculator [63] to get the accurate number of active threads based on all other factors in our experiments.
3.3.1 Model Many-thread Running Process

We can start from the minimal concurrency level indicated by the maximal register/shared-memory request in the program, keep increasing the concurrency level by spilling live variables into shared memory and/or local memory, and keep increasing concurrency level until the point that the overhead of extra local memory spilling cannot be offset by the benefit brought by increased program concurrency. However, increasing concurrency level does not alway help; in fact it helps for most programs except one special case.

This exception is the case which we define as *computation intensive* case; it is when single thread instruction level parallelism (ILP) is inherently good in the program so that the latency is well hidden when a thread is running by itself. We illustrate it with an example in Figure 3.4 *Scenario B*. In Figure 3.4, assume every instruction takes three cycles, and the processor is able to dispatch one instruction in every cycle. The *x-axis* represents the cycle number. The *y-axis* represents the instruction number. In *Scenario B*, instruction 2 does not directly depend on instruction 1, and it can be dispatched immediately at the beginning of the second cycle. It is similar for instructions 2, 3, and 4. We only need one thread per-core in *Scenario B* to fully utilize the computation pipeline (one thread warp for one SM). In this cases, an increased concurrency level does not help improve performance, and they are not concurrency-bound cases. Next we show a concurrency-bound case in which increased concurrency helps improve performance. In Figure 3.4 *Scenario A*, assume we
have dependences between instruction 2 and instruction 1. Instruction 2 can’t start until instruction 1 finishes. If we have only one warp, we cannot dispatch instruction 2 until the beginning of the fourth cycle in the One Warp case of Scenario A. The processor pipeline is thus not fully utilized. However, if we have three warps, in the Three Warps case in Scenario A of Figure 3.4, at the beginning of the second cycle, we can schedule warp 2 to run instruction 1, and at the beginning of the third cycle, we can schedule warp 3 to run instruction 1. Therefore the processor pipeline is fully utilized. Overall, in Scenario A, we need three warps to fully utilize the computing units. Increasing the number of concurrent thread warps from one to three helps improve performance. These cases belong to the concurrency-bound category.

In summary, we want to optimize concurrency-bound programs with multi-level on-chip memory resource allocation strategies. We use a heuristic metric to distinguish between programs that can benefit from increased concurrency and the ones that do not necessarily benefit from increased concurrency. The metric is the average dispatch interval between every two adjacent instructions in the same thread specified statically in the GPU binary code. The average dispatch interval reflects the ratio between the idle cycles and the busy cycles in the pipeline. As illustrated in Figure 3.4 scenario A and scenario B, the dispatch intervals are 3 and 1 respectively. The minimal number of warps to fully utilize the processor pipeline happens to be 3 and 1 for these two cases respectively. The average dispatch interval\(^3\) can also be used as an initial estimate of the number of active warps for every SM. We elaborate our algorithm for concurrency selection in next section.

### 3.3.2 Concurrency Level Search

The main idea of our concurrency level search algorithm is to make the benefits of increased concurrency outweigh the overhead of local memory spilling. With limited registers and

\(^3\)NVIDIA kepler GPU architecture uses a static instruction scheduling approach instead of architecture based instruction scheduling. Dispatch interval is encoded in the instructions if we compile with nvcc for computing capability of 3.0 and above. We reverse-engineered the ISA and parsed instructions to get the dispatch interval for every two adjacent instructions.
shared memory, increasing the concurrency level may force live variables to be spilled into slow local memory. How many local memory spills can be allowed depends on the concurrency level we select. We start from an initially estimated number of active threads as the product of average dispatch interval and the number of cores per SM (every SM is the same so we discuss how to find concurrency level for every SM). Based on the initial estimate, we derive the number of registers and the amount of shared memory for every thread. We then perform shared memory allocation with the initial per-thread register number and shared memory amount. Then we check the number of local memory spills to see if we should decrease or increase the concurrency level. In this algorithm, we use a heuristic criteria to check whether a given concurrency level is good enough; we keep increasing concurrency level above the initial concurrency level if the criteria is met, or keep decreasing concurrency level below the initial concurrency level until the criteria is met. We set the criteria in a way that the local memory spilling overhead can be overlapped with the arithmetic and other non off-chip memory instructions. We use $\text{COMPUTE}_{\text{inst}_{\text{trsf}}}$ to denote the number of instructions that are not off-chip memory instructions after transformation and we use $\text{MEM}_{\text{inst}_{\text{trsf}}}$ to denote the number of off-chip memory instructions after transformation. We refer to this criteria as Computation Interleaving predicate – $CI_{\text{pred}}$ and we describe it as follows:

$$CI_{\text{pred}} : \frac{\text{COMPUTE}_{\text{inst}_{\text{trsf}}} \times AD_{\text{int}}}{\text{MEM}_{\text{inst}_{\text{trsf}}}} > MAX_{\text{cmratio}}$$ (3.1)

$AD_{\text{int}}$ denotes the average instruction dispatch interval. We obtain this by decoding NVIDIA Kepler’s binary ISA. $MAX_{\text{cmratio}}$ is correlated with the number of cycles for an off-chip memory instruction; it is the number of computation instructions with a specific dispatch interval that are needed to hide the latency of one off-chip memory instruction. Its value varies from architecture to architecture. We obtain this parameter value by measuring the cycles of computation and off-chip memory instructions for a specific architecture. If
the condition in Equation 3.1 is satisfied, we consider this concurrency level to be beneficial. In our concurrency level search algorithm, if the initially estimated concurrency level is beneficial, we keep increasing it step by step (and we use thread block size as the step since it is the minimal unit to run on a SM). We choose the largest concurrency level which is beneficial. Otherwise, we keep decreasing the concurrency level step by step until we find the first concurrency level that is beneficial.

We illustrate the major components of our concurrency level search algorithm in Figure 3.5. Note that when the algorithm traverses down, it stops when it hits a threshold $SmemFitTnum$. This is the number of concurrent threads which results in no spills into the off-chip memory, which means the live variables can completely fit into registers and shared memory. The variable $ActTnum$, denotes the number of active threads per SM, and the variable $ActThrd_{opt}$ denotes the final number of active threads per SM we selected.

### 3.4 Evaluation

In this section, we present our experiment results. We perform experiments on two different machine configurations. One is NVIDIA Kepler GTX680. It has 8 streaming multiprocessors (SM), with 192 cores on each of them and 1536 cores in total. It has CUDA computing capability 3.0. Every streaming multi-processor is equipped with 65536 registers and 48KB shared memory. The maximum number of concurrent threads that can run simultaneously on each streaming multi-processor is 2048. The second machine is configured with NVIDIA Fermi card - Tesla C2075. It has 448 cores in total, with 32 cores on each SM. It has CUDA computing capability 2.0. There are 32768 registers and 48KB shared memory per SM. The maximal number of concurrent threads that can run simultaneously is 1536. Notice that these two configurations impose different constraints on single-thread register count and single-thread shared memory with respect to maximal concurrency supported by hardware. We denote the Kepler card as *Kepler* and the Fermi card as *Fermi*. 
We measured computation and off-chip memory instructions latencies with the `clock()` function. Normal algebra instructions like addition and subtraction take 9 cycles and an off-chip memory instruction takes between 300 and 400 cycles. Since reads and writes happen in parallel, we set the average of off-chip memory latency to be between 150-200. Therefore, we choose the larger one 200 and set the parameter `MAX_cmratio` in subsection 3.3.2 to be 200. This parameter is used in our automatic occupancy level selection algorithm, and our experiments support this value as being effective.

To process a benchmark, we first extract the assembly code for the kernel function using NVIDIA binary listing tool `cuobjdump`. We decoded necessary parts of the binary instruction set for NVIDIA Kepler architecture, including scheduling instructions omitted.
Table 3.1: Benchmark Description. RegDemand is the number of registers the compiler tries to use. InstChange is the increase to the instruction count in the auto occupancy. CacheMissRates are the cache miss rates in the auto occupancy; the numerator is with use of shared memory after transformation, and the denominator is with purely global spills before transformation.

by `cuobjdump`, based on `asfermi` [36]. Our binary analysis and modification pass is implemented with the `libelf` library. We implemented our parser with `flex` and `bison`. Our shared memory allocator then performs program analysis, determines the best occupancy level, and transforms the code. We use one fixed register as a shared-memory stack pointer. If necessary, we use a second fixed register to shuffle shared-memory slots for Moving Stack algorithm. Note that this process is done quickly, and takes less than a second on most benchmarks.

We evaluate our methods with seven benchmarks selected from the Rodinia benchmark suite 2.2 [9] and CUDA SDK 5.0. We choose them because they have non-trivial register demand. Note that a lot of benchmarks from Rodinia [9] and CUDA Computing SDK have a low register demand of below 20, which happens to enable maximal hardware supported concurrency for previous and current NVIDIA GPU architectures. Decreasing register pressure for these benchmarks will not help improve concurrency or improve single-thread performance. Our algorithm will choose not to transform these programs, thus we do not include them in discussion. We describe the list of benchmarks used for this chapter in Table 3.1. RegDemand is the number of registers needed per-thread if no spilling to on-chip or off-chip memory happens. It is the default choice by nvcc and traditional CPU register
allocation approach. *InstChange* is the increase in size to the transformed kernel function at the auto occupancy. *CacheMissRate* lists cache miss rates for the auto occupancy before and after transformation.

We present both the results of automatically selected occupancy level through our approach, and the exhaustive search through all possible occupancy levels. When an occupancy level is selected, per-thread register and shared-memory limits are determined by the NVIDIA GPU occupancy calculator [63].

![Figure 3.6: Kepler Performance Results. The Auto bar shows speedup with automatically selected occupancy. The Best bar shows highest speedup among all occupancies by exhaustive search. The Worse bar shows worst speedup among all occupancies. For our baseline, we used the runtime when compiling each benchmark with default settings.](image.png)

We first present the overall performance results for Kepler in Figure 3.6. Each group of bars along the x-axis represents a benchmark. The y-axis represents a particular kernel’s speedup compared to its baseline. For our baselines, we compiled each benchmark using nvcc with the default settings, including no register limits. The first bar Auto represents the speedup at the concurrency level selected by our concurrency selector. The second bar Best represents the best speedup among all possible concurrency levels. The third bar Worst represents the speedup in the worst case among all different concurrency levels. The results demonstrate that the automatically transformed program is typically faster than the
original, and in most cases is close to the best speedup (from an exhaustive search through concurrency levels). \textit{FDTD3d} and \textit{imageDenoising} fail to reach their best speedup due to our conservative algorithm, which avoids the highest occupancies in this case due to the number of static memory operations. In our future work, we plan to incorporate dynamic analysis in order to allow more optimal selections. Overall, although no dynamic analysis is performed, we still have good performance improvement for these benchmarks. This demonstrates the importance of static on-chip memory resource allocation.

![Figure 3.7: Fermi Performance Results. The Auto bar shows speedup with automatically selected occupancy. The Best bar shows highest speedup among all occupancies by exhaustive search. The Worse bar shows worst speedup among all occupancies. For our baseline, we used the runtime when compiling each benchmark with default settings.](image-url)

Next we present the results for the Fermi GPU in Figure 3.7. For our baseline, we compiled the benchmark using nvcc with the default settings, including no register limits except where necessary for the benchmark to run (due to hardware limitations). The bars and axes have the same meaning as in Figure 3.6. Only the \textit{cfd} benchmark here is not improved; this is because \textit{cfd} has an unusually large number of memory instructions, even at the lowest occupancy, and so our conservative algorithm chooses not to increase the occupancy at all. Due to the many memory instructions, increasing the occupancy has less effect than in most benchmarks, regardless. The \textit{particles} and especially the \textit{FDTD3d} benchmarks also
have auto speedups below their best due to the conservative algorithm choosing a lower occupancy than is optimal in these cases. The recursiveGaussian benchmark sees much less improvement than on Kepler. This has to do with the differing limitations of the hardware. On Kepler, a kernel can use up to 63 registers regardless of its block size, but on Fermi, programs with higher block size have a lower register limit, due to the smaller register file. Having a high block size, recursiveGaussian must be compiled with less registers to run at all, increasing the initial occupancy, and therefore lessening the extent to which it can be improved.

3.5 Related Work

Many studies in the past few years have been proposed on register spilling between physical registers and off-chip global memory. A fundamental model is the graph coloring model [7]. In [2], the authors propose an integer linear program modeling of register allocation for CISC machines. In [80] and [52], the authors particularly tackled the problem of register spilling due to software pipelining in loops. Most of these previous studies are for sequential programs, instead of massively parallel architecture. In [1], the authors studied the register allocation schemes for vector machines. However, a vector processor is different from the SIMT processor on modern GPUs. Sampaio and others [73] proposed a divergence aware spilling strategy to save memory, but did not consider concurrency.

The previous studies on GPU also investigate the implication of interaction between concurrent threads on latency minimization. The authors of [74] point out that the ability for memory latency hiding among different vector thread groups is critical. The authors present a model for GPU programs that predicts the performance by calculating memory warp parallelism (MWP) and computation warp parallelism (CWP). While this work focused on modeling of concurrent execution, it did not discuss how to achieve the desired concurrency level. Other relevant GPU work includes topics such as GPU exception handling [58], where register states need to be restored for resuming execution after exception,
and energy saving [21], where the location of registers is critical to energy consumption because the distance between the registers and processors determines the amount of energy consumed during data movement, and hardware register space saving [86], which combines SRAM and DRAM to store more bits into the die area. In [85], a means of optimizing shared memory is explored in order to prevent user-allocated shared memory from reducing occupancy, whereas our approach makes use of non-user-allocated shared memory to lessen the cost of improving occupancy. In [56], an integer programming technique is used to allocate scalars and arrays in shared memory, in order to optimize the code at a higher level than we consider. Most of the aforementioned studies on GPU architecture extensions are implemented and evaluated in hardware simulators.

3.6 Conclusion

In this chapter, we propose a unified on-chip memory resource allocation framework for GPU programs. Our on-chip memory resource framework predicts near-optimal partition of on-chip memory resources and adjusts GPU program to the best concurrency level that matches program characteristics without any on-line or off-line profiling.
CHAPTER 4
ORION: A FRAMEWORK FOR GPU OCCUPANCY TUNING

4.1 Overview

GPU performance tuning and optimization is challenging because of the complexities in GPU architecture and the massive scale of threads in its simultaneous execution environment – a GPU typically runs 10,000s of active threads at one time.

Previous GPU tuning frameworks exploit different factors to improve performance. The PORPLE framework [10] helps programmers determine which type of memory to use with respect to data access patterns. Liu and others [51] leverage input-sensitivity to select the best program compilation and execution parameter. Many studies have focused on domain-specific performance tuning due to the complexities in tuning general-purpose applications. Anand and colleagues [78] explored the tuning of computation transformation parameters and data representation for sparse matrix code. The Halide [72] framework targets image processing applications and focus on tuning the locality and parallelism parameters.

In this chapter, we reintroduce occupancy tuning for GPU programs. Occupancy is an important performance tuning factor that is unique to GPU programs. Occupancy [62] is defined as the ratio between the number of threads active at one time and the maximum number of threads the GPU hardware can schedule. A program running at two different occupancy levels can have up to three or four times difference in performance. We show an example in Figure 4.1. We use the imageDenoising program from CUDA SDK [64] and show its performance at different occupancy levels. We normalize the performance with respect to the best occupancy running time (at 50% occupancy). It can be seen from Figure 4.1 that the difference in running time between the best and worst cases can be up to three times, indicating the importance of occupancy tuning.
Occupancy tuning is tightly coupled with resource allocation. Efficient occupancy tuning requires efficient resource allocation. As discussed in Chapter 3, the occupancy level is controlled by tuning the amount of on-chip memory assigned to every thread. High occupancy leads to high contention, high latency-hiding capability, and high resource allocation pressure (every thread gets less resources). Low occupancy results in low contention, low latency-hiding capability, and low resource allocation pressure (every thread gets more resources). Tuning occupancy and determining the best trade-off point is challenging, as it depends on multiple factors including compile-time resource allocation efficiency, dynamic program behavior, and execution configuration.

In this chapter, we design and implement the first occupancy tuning framework for GPU programs – the ORION framework. ORION determines the most desirable occupancy level and generates the occupancy-adaptive code for any GPU program by combining iterative static compilation and dynamic program adaptation. It has two components: a compiler that generates and selects binaries at several different occupancy levels, and a runtime adaptation component that chooses one of these binaries at execution time. The ORION compiler narrows down the search of best occupancy level to five or fewer possibilities, and the runtime component selects the occupancy level that adapts to dynamic program
behavior. These two stages work together to provide users with the code running the best occupancy level.

Previous work that models the relationship between GPU occupancy and performance does not address the problem of determining and achieving the best occupancy. The GPU performance model proposed by Hong and others [35] uses off-line profiled information to predict the performance of a GPU program. While the prediction method by Hong and others provides satisfactory accuracy, it requires fine-grained information based on an architecture simulator. And it does not provide pro-active occupancy tuning solution. Previous work for GPU resource allocation optimizes per-thread resource allocation. We have demonstrated in Chapter 3 that we can alleviate GPU per-thread register allocation pressure via static compile-time transformation. However, the static optimization does not adapt to runtime program behavior. Furthermore, while alleviating register pressure does indirectly increase occupancy, it is not necessarily true that higher occupancy is always better than lower occupancy [79]. Overall, efficient GPU program execution requires a systematic exploration of both single thread performance and concurrent thread dynamics.

In this chapter, we develop the occupancy-oriented tuning and on-chip resource allocation (ORION) framework. We are not aware of any prior work that systematically explores the influence of occupancy tuning for GPU programs. Our contributions are summarized as follows.

- ORION is the first occupancy tuning framework that taps into both single-thread resource allocation and concurrent thread interaction.

- It combines static and dynamic occupancy tuning, enabling a fast and accurate search for the best occupancy. (subsection 4.2.3 and subsection 4.2.4).

- ORION’s compiler provides an inter-procedure resource allocation model that is rigorously proved to be optimal in both memory space and movement cost (subsection 4.2.2).
• **ORION** not only improves performance – up to 1.61 times speedup – but also resource & energy efficiency, with up to 62.5% memory resource saving, and 6.7% energy reduction over the highly optimized code generated by *nvcc* (Section 4.3).

• **ORION** is immediately deployable on real systems and does not require hardware modification.

### 4.2 Orion System Design

The **ORION** system automatically tunes occupancy. The design of **ORION** addresses two important questions respectively. The first question is how to generate the code given any occupancy level. Since one occupancy level gives the number of registers (shared memory) every thread can use – see Equation 1.1 – it implies a register allocation and register allocation can only be done at compile time. How to perform register allocation efficiently at compile-time and adapt it to GPU program characteristics is critical.

![Normalized Runtime vs. Occupancy](figure4_2.png)

**Figure 4.2: Effect of occupancy on performance for matrixMul**

The second question is how to choose the best code version out of all the code versions corresponding to different occupancy levels. We use two-level selection: compile-time tuning and runtime tuning. The **ORION** compiler selects a set of candidate occupancy levels
that potentially correspond to the optimal occupancy level. The compiler generates multiple candidate versions because the dynamic factors (such as inputs, control divergence, irregular memory accesses) might affect the choice of best occupancy level in a way that no single occupancy is always the best. Therefore we let the runtime component chooses the best occupancy code to use and to adapt to complex runtime program behavior.

The design and implementation of the dynamic occupancy selection process is based on two principles. Note that we discovered that there are two patterns of performance variation with respect to occupancy changes. When concurrency increases, the performance increases, until it reaches a point that benefits from concurrency improvement cannot offset the performance overhead due to individual thread’s resource allocation pressure (recall that individual thread gets less resource when occupancy increases). From this point, there are two patterns: 1) the performance deteriorates, as illustrated in Figure 4.1, and 2) the performance plateaus, as illustrated in the matrix multiplication program in Figure 4.2, due to the fact that the program itself does not have much register pressure.

Therefore, the first principle is based on the fact that the occupancy optimization function is a function that has only one local minimum in running time. In other words, the local minimum is also the global minimum, which makes the optimization problem of occupancy tuning much simpler. We can start from any initial occupancy level, find the right tuning direction, and then move step by step, until we reach the optimal occupancy case.

The second principle is based on the observation for the matrix multiplication example in Figure 4.2, that when we reach best performance occupancy, we should keep tuning until we find the range of occupancy levels that yields performance similar to the best performance. In Figure 4.2, from occupancy 50% to 100%, the performance is similar. Therefore, we can potentially lower the occupancy, increase resource usage per thread for other purposes (e.g. caching or loop unrolling), and improve intra-thread performance significantly. If we know the range in which performance is stable, we know the safe occupancy reduction range without hurting performance. This coincides with the discovery made in
in 2010, which demonstrated that by lowering occupancy and applying optimizations made possible by the reduced concurrency, *matrixMul* can achieve superior performance. However, the work in [79] only looked at matrix multiplication and also it did not give an approach on how to tune the occupancy levels. In our design of occupancy tuning, we not only find the point where the performance is best, but also find the range of occupancies where the performance is best, in particular identifying the lowest occupancy that gives the best performance.

Next, we discuss the detailed design and implementation of the ORION compiler and runtime adaptation component.

![Figure 4.3: The ORION compiler and runtime collaboration](image)

4.2.1 Our Framework

The overall ORION framework works as follows (Figure 4.3). In the first step, the compiler sets an initial occupancy level, generate corresponding code and determines the occupancy tuning direction: increasing or decreasing based on the performance model in subsection 4.2.3. The initial occupancy is defined such that all live values fit into the minimal number of registers, or the maximum registers per thread limit (by hardware) has been reached. Then the compiler performs occupancy testing, if the occupancy needs to be increased or decreased, the ORION compiler performs on-chip memory allocation, assigning register and shared memory per-thread to achieve the updated occupancy. And we repeat the testing and updating process until a termination condition is met, defined by our
performance model in subsection 4.2.3. The compiler generates and selects \( \leq 5 \) different code versions for a GPU kernel function and in most cases \( \leq 3 \) versions (in evaluation), which helps the runtime quickly adapt to the best version. Runtime adaption is shown in Figure 4.3(b).

There are three major stages in the ORION framework.

- The \textit{realizing occupancy} stage (subsection 4.2.2) ensures that the code generated to achieve a certain occupancy level is efficient. For instance, it avoids excessive spilling from on-chip memory (register and shared memory) to off-chip memory.

- The \textit{compile-time occupancy tuning} stage (subsection 4.2.3) ensures that we select a small set of kernel binaries with good occupancy levels, ruling out the versions that are unlikely to perform well.

- The \textit{runtime occupancy adaptation} stage (subsection 4.2.4) ensures that we select the best kernel to execute at runtime, and also that we avoid aggressive optimization.

Every stage is important. The first two stages are performed at compile-time, and the last stage is performed at runtime.

### 4.2.2 Realizing Occupancy

To realize a certain occupancy, we bound the number of registers and the size of shared memory used per thread. We first represent a program in the Static Single Assignment (SSA) form, in which every variable is defined once and only once. Then we generate the pruned SSA form to eliminate \( \phi \) functions. Next we start assigning the pruned SSA variables, first placing them into registers with spills into local memory, and then reassigning a subset of local memory variables to shared memory. Since every thread executes the same binary code, allocating for one thread is equivalent to allocating for all the threads. We call the commensurate amount of space for a 4-byte register in on-chip memory (including
shared memory and cache) an on-chip memory slot. A variable can be placed into register, shared memory, or L1 cache (via local memory).

(a) Input parameters:
G: the interference graph; each node is a variable.
C: the number of colors (physical registers)

(b) Stack Order:
  S = empty stack
  while G is nonempty
     nextVar = null
     for each variable v in G
       if v.width + v.edges <= C
          nextVar.width > v.width
             nextVar = v
       if nextVar == null
          nextVar = first variable in G
     for each variable v in G
       if nextVar.width > v.width || (nextVar.width == v.width && nextVar.edges > v.edges)
          nextVar = v
     S.push(nextVar)
     G.remove(nextVar)

(c) Coloring pseudocode:
  s = S
  while s is nonempty
     v = s.pop()
     usedColors = {}
     for each colored variable u in v.edges
       usedColors = usedColors U u.color
     for c = 0 to C - v.width
       if {c, ..., c + v.width - 1} U usedColors == {} 
          v.color = {c, ..., c + v.width - 1}
          break
     if v was colored
        S.remove(v)
     else
        s.remove(v)
        spillList.add(v)
     s = S

Figure 4.4: Single procedure multi-class allocation alg.

Minimizing Space Requirement  The on-chip memory space to store the variables should be minimal since if it does not fit into on-chip memory space, there will be spilling into off-chip DRAM memory, which is significantly slower. Therefore the first thing we need to optimize is the on-chip memory space needed for a set of live variables.

Optimal register allocation for single procedure has been studied extensively in CPU literature, and the technique can be applied to on-chip memory allocation. We adopt the Chaitin-Briggs register allocator [6] and build a variant of it by taking into consideration the wide variables (64-bit, 96-bit, or 128-bit) that need consecutive and aligned registers. We use a stack to track the priority of variables to be allocated (colored). Our single-procedure allocation algorithm is detailed in Figure 4.4.

However, there is limited inter-procedure allocation research on GPUs. While functions can be in-lined, it is not practical to in-line every function. For instance, after aggressive inlining by the nvcc compiler, the cfd program still has 36 static function calls (please see
Section 4.3 Table 4.2). Moreover, although certain GPU programs exhibit no procedure calls in source code, there are still function calls in the binary. An example is the intrinsic division function, which is implemented as a function call for GPU architecture. This may appear frequently in scientific programs in Rodinia [9] benchmark suite that use the floating-point division function.

We propose an allocation algorithm for multi-procedure GPU kernels which is **optimal** in terms of both space and data movement requirements.

We describe our inter-procedure allocation algorithm as a *compressible stack*. The idea of the *compressible stack* is simple. With *compressible stack*, right before entering a sub-procedure, we compress the stack by coalescing the used on-chip slots such that the sub-procedure can use the maximum number of contiguous on-chip memory slots. Right after the sub-procedure returns, we restore the location of moved slots back to their original locations so that the program can continue execution correctly.

We show how the compressible stack works with an example in Figure 4.6. In Figure 4.6(a), we use the column $S_i$ to represent an on-chip memory slot. After single procedure allocation, one or more variables are mapped to one on-chip memory slot. We
show the live range of every variable using the vertical black bar in Figure 4.6(a). The liveness information indicates the used/unused status of an on-chip memory slot $S_i$ at different call points in the program. The left “Program” column represents the code of the program including the procedure calls. If no variable is mapped to an on-chip memory slot at an execution point, the slot is unused. For instance, S3 is unused when $foo2$ is called in Figure 4.6(a).

In Figure 4.6(a), before calling $foo1$, we move the slot that contains variable “var5” into the slot between the slots that contain variables “var3” and “var1” so that we have larger contiguous free space for procedure $foo1$. This is important since a sub-procedure uses contiguous stack space. Right after the sub-procedure returns, we restore the location of variable “var5” to resume execution.

We demonstrate that it is important to minimize space for inter-procedure allocation by showing the performance difference between the space minimize version and the space unoptimized version in Figure 4.5. The “no-space minimization” bar corresponds to the unoptimized version and the running time is normalized to the optimized one.

**Minimizing Data Movement** The above shows how to minimize on-chip memory space used across procedure calls. This comes at the cost of increased data movements. Therefore, minimizing data movement is also important.

In CPU single procedure allocation literature, there has been work that proposes to trade-off data movement for space. The chordal graph coloring model [67] is a well known model that minimizes register usage while increasing data movements for removing $\phi$-functions. Later work by Hack and other further optimize data movements caused by removal of $\phi$-functions [23]. However, there is no such work in minimizing data movements for inter-procedure allocation. As far as we know, our work is the first one that minimizes data movement for inter-procedure on-chip memory allocation.

We first show an example of how data movements can be reduced. Figure 4.6(b) is the
Figure 4.6: The impact of on-chip memory slot layout on the number of data movements for a compressible stack. Every column $S_i$ represents a slot. The vertical blocks in the column represent the liveness of the variable assigned to the slot. Bold arrows represent direction of data movement for entering or leaving a sub-procedure.
Table 4.1: Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SS_i$</td>
<td>set of variables mapped to the i-th stack slot</td>
</tr>
<tr>
<td>SLOT$_i$</td>
<td>i-th slot from the bottom of the stack</td>
</tr>
<tr>
<td>$X_{ij}$</td>
<td>mapping between $SS_i$ and j-th slot (0 or 1)</td>
</tr>
<tr>
<td>$L_{ik}$</td>
<td>liveness of $SS_i$ at k-th sub-procedure call</td>
</tr>
<tr>
<td>$B_k$</td>
<td>desired stack height at k-th sub-procedure call</td>
</tr>
<tr>
<td>$C_{ijk}$</td>
<td># of swaps incurred by placing $SS_i$ at j-th slot for k-th sub-procedure</td>
</tr>
<tr>
<td>$W_{ij}$</td>
<td># of swaps incurred by placing $SS_i$ at j-th slot</td>
</tr>
<tr>
<td>$N$</td>
<td># of sub-procedure calls</td>
</tr>
<tr>
<td>$M$</td>
<td>maximum # of simultaneously live variables in this procedure</td>
</tr>
<tr>
<td>$P_{mov}$</td>
<td># of swaps incurred because of k-th sub-procedure call</td>
</tr>
</tbody>
</table>

same as Figure 4.6(a) except that the addressing of the on-chip memory slots is different. For the original layout in Figure 4.6(a) at the point call(foo1), the sub-procedure foo1 needs to use three consecutive slots, thus var5 in S4 needs to be copied to S2. In Figure 4.6(a), altogether three data movements are necessary before entering three call points call(foo1), call(foo2) and call(foo3) as indicated by the three arc arrows for “before call”. However, if we place the variables of the original S2 slot to the location of original S4, S3 to S2, S4 to S3, as illustrated in Figure 4.6(b), the total number of data movements is reduced to 1 at call(foo2) point, while at call(foo1) and call(foo2), all available on-chip memory slots are contiguous.

Our compressible stack optimizes the layout by changing the address of physical on-chip memory slots. We provide a polynomial time algorithm that finds the optimal address mapping. We achieve this by modeling the problem as a maximum-weight bipartite matching problem [81].

Let $N$ be the number of static sub-procedures calls in the procedure of interest. $M$ represents the number of variable sets, $SS_i$, assigned during the single-procedure graph coloring (allocation) process such that $i = 0...M - 1$ and each $SS_i$ is mapped to one on-chip memory slot.

We denote the number of data movements incurred for entering/leaving the $k$-th procedure call as $P_{mov}^k$. The objective is to find the physical on-chip memory slot each variable
set \( SS_i \) for \( i = 0 \ldots M - 1 \) is mapped to such that the total number of data movements is minimal.

\[
\min T_{\text{mov}} = \sum_{k=0}^{N-1} P_{k}^{\text{mov}}
\]

We model this problem as a *maximum-weight bipartite matching* problem by making use of the following Theorem.

**Theorem 1.** With the notations defined in Table 4.1, in the minimal-mov-assignment (MMA) problem, the total number of data movements contributed by placing an arbitrary variable set \( SS_i \) at an arbitrary location \( j \)-th memory slot \( SLOT_j \) across all \( k \) immediate sub-procedures is a constant. We define this number as \( W_{ij} \). Assume at the \( k \)-th sub-procedure call we need to bound the compressed caller stack to at most \( B_k \) slots, we have:

\[
W_{ij} = \sum_{k=0}^{N-1} C_{ijk}
\]

while

\[
C_{ijk} = 1 \text{ if } (L_{ik} == 1 \&\& j \geq B_k), \text{ otherwise } C_{ijk} = 0.
\]

**Proof.** Since a movement will be invoked if and only if there is an available stack slot placed beyond the top of the after-compression stack, we can determine if a placement of the \( SS_i \) set will incur a movement by checking its location (address) in the stack.

If and only if \( SS_i \) is live during the lifetime of the \( k \)-th sub-procedure (liveness indicated as \( L_{ik} \) in Table 4.1), and placed at \( j \)-th slot (\( j \) starts from 0) counting from the current procedure’s stack bottom is greater than or equal to \( B_k \), \( j \geq B_k \), the number of data movements invoked by placing variable set \( SS_i \) at \( SLOT_j \) for \( k \)-th sub-procedure \( C_{ijk} \) is 1; otherwise, \( C_{ijk} \) is 0. Therefore, we get all the movement contributed by placing \( SS_i \) into \( SLOT_j \) by summing up \( C_{ijk} \) for \( k = 0 \ldots N - 1 \). The theorem is thus proved. 

We then transform the problem into a bipartite matching problem. A bipartite graph is a graph whose nodes can be decomposed into two disjoint sets such that no two nodes within the same set are adjacent as shown in Figure 4.7. A perfect matching in a bipartite
Figure 4.7: Bipartite matching model for minimal-mov-assignment problem. $SS_i$ represents the $i$-th set of variables that can be mapped to one on-chip memory slot, $SLOT_i$ represents the $i$-th physical on-chip memory slot in the procedure of interest.

The graph is a set of pairwise non-adjacent edges that covers every node in the graph. A maximum weighted bipartite matching is one where the sum of the weights of the edges in the matching is maximal as shown in Figure 4.7.

We let one of the two disjoint sets of nodes correspond to the sets of variables – $SS_i$ for $i = 0...M - 1$. The other set of nodes correspond to the memory slots $SLOT_0...SLOT_{M-1}$ from the bottom of the stack. One edge connects between a variable set and a stack location (address).

We set the weight of the edge between a set $SS_i$ and a $j$-th stack slot to negative $W_{ij}$ as defined in Theorem 1. This value is the total number of movements invoked by placing $SS_i$ at $SLOT_j$. Therefore, a maximum weighted matching will indicate a minimum number of movements as indicated in Figure 4.7.

We solve the maximum weighted bipartite matching problem using the modified Kuhn-Munkres algorithm [60], with $O(M^3)$ time complexity, with $M$ being the total number of
variable sets. Once a matching is found, we can infer where every variable set can be placed.

Our model works for the case where we need not compress the stack to the minimal size possible. For example, if the stack can be compressed to allow for four free slots, but we only need three, then it is sufficient to compress the stack to allow three slots. Therefore, we avoid extra overhead from pointless stack compression movements. Let the parameter $B_k$ be the size the stack needs to be compressed to, such that $B_k$ is greater than the minimal possible compressed stack size. Then the optimality and complexity results still hold.

We demonstrate the effectiveness of the data movement optimization Figure 4.5. The bars that correspond to the case of “unoptimized data movement minimization” are the case without data movement optimization. Note that without data movement optimization, the performance might be even worse than not doing stack compression again. Therefore, minimizing data movement is extremely critical for minimal space optimization to work well (which is critical for occupancy tuning).

4.2.3 Compile-Time Occupancy Tuning

During compile-time tuning, we perform test and update steps, as shown in Figure 4.3(a). The *occupancy testing* component checks the generated kernel binary and determines if the occupancy needs to be further increased and decreased. If the occupancy needs to be further updated, shown as the back loop in Figure 4.3(a), then the *realizing occupancy* component will be invoked again to generate a new kernel binary. If the occupancy does not need to be further updated, then the previously checked versions will be saved into the candidate set of kernel binaries for runtime adaptation. Our compiler determines and generates a set of kernel versions in which the optimal occupancy version is most likely to appear. We are able to narrow down the set of candidate kernel versions to within six, and in most cases less than three, making it easy for the runtime component to choose the right kernel version.
original: the kernel at original occupancy
canTune: true iff benchmark has enough iterations

```
if MAXLIVE >= 32
  direction = increasing
else
  direction = decreasing
if canTune
  kernels.add(original)
  if direction = increasing
    for each occupancy from conservative to max
      kernels.add(occupancy)
  else
    kernels.add(conservative)
else
  kernels.add(get_static_selection())
```

Figure 4.8: Occupancy update algorithm

We set the initial occupancy such that all variables fit into the minimal number of registers, or the maximum number of registers per thread (by hardware) is reached. We call it the original version since this is the version for which we decide the tuning direction (increasing or decreasing). The original version is not necessarily the best version, but it is a safe version. Once the initial occupancy is set, next we decide the direction of increasing or decreasing occupancy in the iterative selection process. Once the direction is determined, we keep increasing/decreasing the occupancy levels and testing the generated code. We stop increasing/decreasing at a certain point if a termination condition has been met according to our performance model below. We show the detailed occupancy test and update algorithm in Figure 4.8. In Figure 4.8, we define another version called the conservative version, which is the version where all variables fit into on-chip memory. The conservative version usually provides better occupancy than the original version since we fit more threads using all on-chip memory (register, smem, and cache) than only using registers.
To determine the direction of tuning, we rely on a metric determinable at compile-time, max-live, which indicates the total number of registers and memory slots necessary for the program. The analytical model [34] uses off-line profiled information, including memory throughput and dynamic instruction count, to estimate the performance of a GPU program. Our approach does not require off-line profiling and yet is lightweight in determining the amount of memory/computation parallelism.

In cases where the kernel function cannot be tuned (for example, if it only has a single iteration), the selection process will use the static selection algorithm described in [32] to generate the final kernel function.

**Max-live** We use a metric called max-live, which is equal to the number of registers necessary to hold all simultaneously live variables. When this value is low, the on-chip memory resource demand per-thread is low, and thus a high occupancy is reached (potentially hitting the maximum number of active threads that hardware can handle). For this type of application, we can tune only by decreasing the occupancy from the initial original occupancy. We set to max-live threshold to 32 in our experiments, which is the number registers needed to achieve the hardware maximum occupancy level for Kepler architecture. If a program’s max-live is less than the number of registers which allows the hardware maximum occupancy, then occupancy cannot be increased through allocation.

Finally, we provide a fail-safe option in case the direction predicted at compile-time does not work at runtime (although this has rarely happened in our evaluation). We generate kernel codes in the increasing occupancy direction (the conservative code and the next occupancy up) if the direction is predicted to be decreasing, and also generate code that enables decreasing occupancy (if the direction is predicted to be increasing). This way, in the rare event that our initial direction is wrong, we can try the other direction as a fail-safe. Note that we do not need to generate multiple versions of code to correspond to decreased
Final kernel has been set?

Yes → (Run final kernel.)

No →

First iteration?

Yes → (Run first kernel.)

No →

Tried every occ in tuning direction?

Yes → (Set final kernel and run it.)

No →

Direction of increasing occ?

Yes →

(Run next kernel.) Worse runtime?

Yes → (Set final kernel to previous kernel.)

No →

(Run next kernel.) Slowdown > .02?

Yes → (Set final kernel to previous kernel.)

No →

Figure 4.9: Dynamic occupancy selection algorithm
occupancy levels, since we can tune occupancy down by dynamically increasing shared memory usage per thread as shown in Figure 4.8.

4.2.4 Runtime Occupancy Adaptation

Given the candidate list of kernel binaries generated by the compiler, the ORION runtime monitors kernel performance and dynamically selects the best kernel versions. In a loop that calls the GPU kernel of interest, we run the original kernel in the first iteration. From the second iteration we start running the next version in the list and updating occupancy level in the predicted tuning direction by the ORION compiler, until we see performance degradation. If an iteration has no performance degradation, then in the next iteration we simply run the next occupancy in the current direction. This algorithm is shown in Figure 4.9.

In practice, we find that the tuner usually only needs three iterations to adapt to the best occupancy. As long as the kernel runs for many iterations, the overhead of tuning is low. The algorithm can be augmented to handle misprediction of the tuning direction, by switching direction if the original occupancy is selected as the final kernel. We find that this is not typically necessary.

Most GPU programs contain a loop around the GPU kernel of interest. If there is no loop but there are enough threads, then we perform kernel splitting [87]. We split one kernel invocation into multiple invocations, such that every invocation of the split kernel launches a subset of the threads and the total threads across invocations is the same as the original kernel invocation.

It is worth noticing that in certain cases, a decreased occupancy can yield the same performance while significantly reducing resource usage. We show the normalized running time of the srad program on NVIDIA Tesla C2075 in Figure 4.10. The performance in Figure 4.10 is normalized to the performance when there are maximal active threads on
each SM. In \textit{srad}, even reducing the occupancy by half yields nearly the same performance, and so reducing occupancy is suggested for this program.

### 4.3 Evaluation

Our framework consists of two components. One is the \textsc{Orion} compiler and the other is the runtime adaptation component. The \textsc{Orion} compiler’s front end, middle end, and back end take on different responsibilities. The front end is responsible for taking a GPU binary file as input, converting it into assembly code, and analyzing the assembly to extract a high level intermediate representation (IR). The middle end utilizes the IR generated in the front end and transforms the IR. The IR includes the control flow graph and the call graph. The middle end obtains a single static assignment (SSA) form of the code, extracts live ranges, performs resource allocation, updates the control flow graph, and writes back to the assembly code. The static multi-kernel selection and generation is in the middle end of the \textsc{Orion} compiler. The back end converts the transformed assembly code back to binary code.

Candidate kernels are generated at multiple different occupancy levels. For evaluation,
we let the ORION compiler generate code at all occupancy levels, allowing for identification of the best and worst cases. We compare these with the ORION selected occupancy and the default code generated by nvcc.

The runtime adaptation component performs the feedback-based tuning algorithm as described in subsection 4.2.4. The runtime component works with the compiler component, as it can only choose the kernel binaries that are generated from the multi-kernel binary generation stage in the ORION compiler.

We perform transformation directly on the binary code (SASS) rather than PTX so that the transformation effects can immediately be reflected in the final binary code. Using PTX requires a further compilation using ptxas from PTX to binary, and the changes made in PTX may be lost since ptxas may perform another level of register allocation.

We build the ORION framework upon the following tools. The front end of ORION compiler is built upon the parser generator tools flex and bison. To encode and decode binary, we use the binary instruction set architecture (ISA) of NVIDIA GPUs documented in the open-source project asfermi [36] for cuda computing capability 2.0, and we also extended the ISA support by reverse engineering the ISA using the same approaches in asfermi [36].

While the ORION framework currently only supports a subset NVIDIA GPU architectures, it can easily be extended to support additional GPU architectures if we add a new front-end and back-end to decode and encode the binary, since the middle-end and the transformation algorithms remain the same.

**Platform** We perform experiments on two different machine platforms. One is equipped with an NVIDIA GTX680 GPU. It has 8 streaming multi-processors (SMs) with 192 cores, for a total of 1536 cores. Every SM has 65536 registers, and 64KB of combined shared memory and L1 cache. The maximal number of active thread warps on one SM is 64, and the maximal number of active threads per SM is 2048.
Table 4.2: Detailed benchmark information. Reg is the number of registers needed to avoid spilling. Func is the number of static function calls. Smem indicates whether there is user-allocated shared memory.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Domain</th>
<th>Reg</th>
<th>Func</th>
<th>Smem</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfd [9]</td>
<td>Fluid dynam.</td>
<td>63</td>
<td>36</td>
<td>No</td>
</tr>
<tr>
<td>dxtc [64]</td>
<td>Image proc.</td>
<td>49</td>
<td>11</td>
<td>Yes</td>
</tr>
<tr>
<td>FDTD3d [64]</td>
<td>Numer. analysis</td>
<td>48</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>hotspot [9]</td>
<td>Temp. modeling</td>
<td>37</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>imageDenoising [64]</td>
<td>Image proc.</td>
<td>63</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>particles [64]</td>
<td>Simulation</td>
<td>52</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>recursiveGaussian [64]</td>
<td>Numer. analysis</td>
<td>42</td>
<td>21</td>
<td>No</td>
</tr>
<tr>
<td>backprop [9]</td>
<td>Machine learning</td>
<td>21</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>bfs [9]</td>
<td>Graph traversal</td>
<td>16</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>gaussian [9]</td>
<td>Numer. analysis</td>
<td>11</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>srad [9]</td>
<td>Imaging app</td>
<td>20</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>streamcluster [9]</td>
<td>Data mining</td>
<td>18</td>
<td>0</td>
<td>No</td>
</tr>
</tbody>
</table>

The second machine platform is configured with an NVIDIA Tesla C2075 GPU. It has 14 streaming multi-processors (SM) with 32 cores, for a total of 448 CUDA cores. Each SM has 32768 registers and 64KB of combined shared memory and L1 cache. The maximal number of active thread warps on every SM is 48 and the maximal number of active threads is 1536.

For both platforms, each register is 4 bytes. If there are wide variables (i.e., 64-bit, 96-bit, or 128-bit variables), then they must be placed in aligned, consecutive 32-bit registers. We refer to the first machine configuration as GTX680, and the second one as Tesla C2075.

**Benchmarks** We evaluate the effectiveness of the ORION framework on benchmarks shown in Table 4.2. These benchmarks are chosen to cover GPU programs from various domains with different characteristics: high register pressure v.s. low register pressure, with and without function calls, and with and without user-defined shared memory. Note that the number of function calls is counted after function inlining. In GPU program compilation, function calls are inlined as much as possible since there is a local stack for every thread, which needs to be minimized for a large number of threads running at the same
time. However, as shown in Table 4.2, there is still a non-trivial number of function calls that are not practical to be inlined. This confirms the necessity of efficient inter-procedure register allocation. These benchmarks come from the Rodinia [9] benchmark suite and the CUDA Computing SDK [64].

**Metrics** We evaluate the ORION occupancy tuning framework using three different metrics. The first metric is performance - in particular, the effectiveness of the ORION compiler at generating good code, as well as the effectiveness of the tuner to adapt to the best occupancy as compared with exhaustive search. The second metric is the resource usage efficiency of Orion: whether it uses minimal resources (registers) to achieve the best performance, where the best performance is defined as the best running time for all different occupancy levels. The third metric is energy usage. We discovered that there is also energy saving when there is resource usage saving. The reason is that when occupancy decreases (while maintaining the same performance), the power usage of the register file (and/or cache) is also reduced, thus effecting energy saving.

4.3.1 Performance

We show performance evaluation results first, using the seven benchmarks which the ORION compiler determined would benefit from increased occupancy. For comparison purpose, we let the ORION compiler generate code for every occupancy level. Figure 4.11 shows the worst performance across different occupancy levels (longest running time – the Orion–Min bar), the best performance across all occupancy levels (shortest running time – the Orion–Max bar), and the performance of the code generated by nvcc. For all of these benchmarks, the difference between the best performance and worst performance across occupancy levels is significant - in some cases, more than 75% - demonstrating the importance of occupancy selection. It is not clear how nvcc chooses the occupancy since the nvcc compiler backend is not open source. We can observe from Figure 4.11 that the version selected by nvcc typ-
ically is not the worst case scenario among all occupancy levels, but it certainly has missed performance optimization opportunities for most cases. The nvcc selected version is rarely the best occupancy, as illustrated in Figure 4.11, except in the case of recursiveGaussian.

We also show the occupancy selection result of our ORION framework. Figure 4.11 shows ORION performance – the Orion-Select bar. This bar includes the overhead of dynamic tuning. We can see that Orion-Select is close to the best performance obtained by exhaustively searching all occupancy levels. The performance of Orion-Select come from two aspects – static selection that narrows down the possible kernel versions and dynamic selection that chooses the best kernel version at runtime. The static selection ensures that there are no more than five different kernel versions selected at compile-time. During runtime selection, ORION required less than three iterations on average to tune each benchmark. We find that in most benchmarks, either there are sufficiently many iterations to perform dynamic tuning, or there are sufficiently many threads that we can split the kernel call into multiple, smaller invocations in order to create additional iterations. The particles benchmark, however, is an exception to this, and so ORION chooses the compiler-picked, statically-tuned kernel version as described in subsection 4.2.3, which still provides significant speedup over the default code generated by nvcc.

On average, ORION achieves 26.17% speedup on C2075 and 24.94% speedup on GTX680.

Another factor which affects the performance is cache configuration. On both Tesla C2075 and GTX680 devices, the shared memory and the L1 cache can be reconfigured to use a different size ratio. In Table 4.3, we show the comparative speedup of using a small cache configuration (16KB L1 cache and 48KB shared memory) versus using a large cache configuration (48KB L1 cache and 16KB shared memory) for the Orion-Select occupancy level. Note that the results presented in Figure 4.11 are all for small cache configuration.

With different cache configurations, we distribute variables differently. For the smaller shared memory configuration, we fit fewer variables into the shared memory but spill more variables into local memory, which can reside in on-chip memory using L1 cache. To
Figure 4.11: Normalized speedup over nvcc version. We show the best/worst performance at all occupancy levels. ORION-MAX is the best performance. ORION-MIN is the worst. ORION-SELECT is the performance with tuning. NVCC is the performance by nvcc generated code.

determine how many shared memory slots to use for a given occupancy, we use the formula described in subsection 1.2.1).

From Table 4.3 we can see that performance is often similar for both configurations. However, cases such as FDTD3d on Tesla C2075 show more degradation in the large cache performance. When we spill the variables to local memory, due to the non-deterministic feature of thread interleaving, it is difficult to guarantee that the L1 cache will behave as expected. For example, cache thrashing may happen when thread execution is interleaved at different times. Overall, it is safer to use shared memory to explicitly store live variables than to use hardware cache. Note that for programs that use a significant amount of user-defined shared memory per thread block, the large cache configuration cannot be used due to the occupancy requirement, and therefore three entries in Table 4.3 are empty.

4.3.2 Resource & Energy Usage

For five of our benchmarks, the ORION compiler predicts that dynamic tuning should lower occupancy, since these benchmarks have small register pressure (small max—live as described in subsection 4.2.3). They have already reached the maximum occupancy supported by hardware, therefore the only potential tuning direction is downwards. The
Table 4.3: Speedup with Small Cache (SC) vs Large Cache (LC) at ORION’s selected occupancy. In some cases, hardware constraints prevent the LC case from running.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>C2075 SC</th>
<th>C2075 LC</th>
<th>GTX680 SC</th>
<th>GTX680 LC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfd</td>
<td>1.3230</td>
<td>1.2939</td>
<td>1.1656</td>
<td>1.1588</td>
</tr>
<tr>
<td>dxtc</td>
<td>1.5409</td>
<td>-</td>
<td>1.3980</td>
<td>-</td>
</tr>
<tr>
<td>FDTD3d</td>
<td>1.5674</td>
<td>1.2811</td>
<td>1.2605</td>
<td>1.2552</td>
</tr>
<tr>
<td>hotspot</td>
<td>1.1834</td>
<td>1.1780</td>
<td>1.1175</td>
<td>-</td>
</tr>
<tr>
<td>imageD</td>
<td>1.1229</td>
<td>1.1225</td>
<td>1.0834</td>
<td>1.0850</td>
</tr>
<tr>
<td>particles</td>
<td>1.1608</td>
<td>1.1492</td>
<td>1.6045</td>
<td>1.6774</td>
</tr>
<tr>
<td>recursiveG</td>
<td>1.0000</td>
<td>1.0002</td>
<td>1.0008</td>
<td>1.0001</td>
</tr>
</tbody>
</table>

Figure 4.12: Results of downward occupancy tuning.

register file utilization and runtime for these five benchmarks are shown in Figure 4.12. These values are normalized to those of the program generated by nvcc. The normalized occupancy is the same as the register file utilization, and so we exclude it from the figure. Overall, we decrease occupancy and register usage by 19.17% on average, with little loss in performance.

We are unable to tune the backprop benchmark, due to the simplicity of its kernel function, which contains less than 100 binary instructions and has no loops or subroutines. Attempting to tune this benchmark’s kernel function would lead to significant overhead, as the runtime of the kernel function is similar in scale to the overhead of launching an empty kernel function, especially if kernel splitting is employed. In such cases, it makes more sense to simply default to the original version of the kernel. ORION is also unable to get
significant reduction of occupancy for bfs, despite lower occupancy being advantageous in this benchmark, because bfs does different amounts of work in each iteration, making it difficult to compare consecutive invocations. Even so, we do get some reduction for bfs on the GTX 680 architecture. In future work, we may be able to improve tuning for such cases by calculating the amount of work at each iteration and applying a multiplicative factor to the runtime.

Besides saving registers, lowering concurrency has the additional benefit of reducing power consumption due to the reduced utilization of the register file. We measured this using NVIDIA’s CUPTI API. GTX680 does not allow for power measurement in this manner, and so we show the energy savings only for Tesla C2075 in Figure 4.13. We include both the energy saving at the selected occupancy level, and the ideal energy saving determined via exhaustive search.

It is demonstrated in Figure 4.12 that we can sometimes attain speedup when lowering occupancy, due to the decreased resource contention that results from fewer active threads. We find that this occurs more easily on Tesla C2075, where the L1 cache is used for both global memory and local memory, than on GTX680 where the L1 cache is used exclusively.
for thread-private local memory. Overall, we get an average speedup of 3.24% for these five benchmarks.

![Graphs showing effects of occupancy on performance for C2075 and GTX680.](image)

**Figure 4.14**: Effects of occupancy on performance for C2075.

![Graphs showing effects of occupancy on performance for GTX680.](image)

**Figure 4.15**: Effects of occupancy on performance for GTX680.

Finally, in Figure 4.14 and Figure 4.15, we show in detail how performance varies as occupancy changes. We show two benchmarks for each architecture since the performance variation pattern is similar. In Figure 4.14(a), we show the benchmarks *gaussian*, which is insensitive to occupancy tuning. Its performance changes very little across occupancy levels, demonstrating significant potential for resource saving and energy saving. Figure 4.14(b) (*streamcluster*) and Figure 4.15(a) (*backprop*) show skewed bell curves, with performance being best at around 75% occupancy but not changing significantly above 50%. In Figure 4.15(b) (*bfs*), performance is best at highest occupancy, but again changes only a little when above 50%.
In all four of these cases, performance as a function of occupancy plateaus, demonstrating a range of occupancy values between which performance is very similar. This observation confirms the advantage of dynamic tuning: even when the performance is the best, we can still keep tuning and obtain potential saving for resources and energy usage. Further, in such programs, we can use this information for additional optimization. For example, loop unrolling is a common technique which reduces branch penalties, but may increase register pressure and therefore lower occupancy. By finding this range of similar occupancies, however, we can determine the amount of leeway available with which to perform such optimizations without experiencing slowdown.

4.4 Related Work

In this chapter, we propose an occupancy tuning framework for GPU programs. There have been GPU performance tuning frameworks that focus on different factors. The PORPLE framework [10] determines which type of memory to use according to different data access patterns. Liu and colleagues [51] utilize input-sensitivity to select the best program compilation and execution parameter for every input. Yang and others [84] have developed a GPU compiler that focus on static-time memory optimization and parallelism management. There are also domain-specific program tuning studies by compiler designers. Anand et al. [78] explores the dimension of data representation methods for sparse matrix code. The Halide [72] framework tunes the locality and parallelism parameters for image processing pipeline. However, none of the tuning frameworks exploited the impact of occupancy tuning on general-purpose GPU programs.

Since GPU program occupancy tuning is correlated with resource allocation, we also compare our work with previous resource allocation studies. Register allocation for CPU programs has been extensively studied [7] [8] [70] [2] [24] [67] [3] [11] [80] [54]. For GPU register allocation, Sampaio and others [73] identified the opportunities in saving registers for control flow statements in GPU programs. Our prior work [32] places the spilled reg-
ister variables by nvcc into available shared memory, however it does not perform register allocation and it does not handle wide variable registers. Further it is purely static compile-time approach and does not adaptively tune the occupancy (downwards or upwards) based on runtime program behavior. None of the above work thoroughly explores the relations between register (resource) allocation and occupancy tuning.

4.5 Conclusion

We propose the first framework for GPU occupancy tuning, ORION. The ORION framework performs two-level occupancy selection. The ORION compiler performs the first-level occupancy selection and generates a set of kernel binaries for dynamic tuning. The ORION runtime performs a second-level of occupancy selection which adapts to dynamic program behavior. It is able to find the best occupancy or an occupancy close to the best one within three iterations on average. The ORION compiler and runtime not only improve performance – achieving up to 1.61 times speedup – but also resource & energy efficiency, with up to 62.5% memory resource saving, and 6.7% energy reduction over the highly optimized code generated by the nvcc compiler.
CHAPTER 5
GPU TAINT TRACKING

5.1 Overview

GPUs have been widely used in many important application domains beyond scientific computing, including machine learning, graph processing, data encryption, computer vision, etc. Sensitive information propagates into GPUs and, while being processed, leaves traces in GPU memory. For example, in a face recognition application, besides the input photo itself, the features extracted at different levels of the deep learning neural networks may also contain part of sensitive or private information. Figure 5.1 shows the extracted features from the first level of neural networks in a face recognition program, where Figure 5.1(a) is the original picture and Figure 5.1(b) are features such as silhouette of a human face. Given a sensitive input user photo, features in deep learning applications may contain much of the sensitive data as well. Other sensitive data in today’s GPU applications include encryption keys, digits in personal checks, license plates, location information in virtual reality apps, etc. If not tracked or protected, sensitive information can be inadvertently leaked or stolen by malicious applications on GPUs.

![Image of a face and its extracted features.](image)

(a) Org. Photo  
(b) Extracted Features

Figure 5.1: Neural network information leaking example.

Taint analysis [13, 14, 18, 33, 61, 71, 83, 90] is a powerful tool for information flow tracking and security. It tracks where and how sensitive information flows during program
execution. Taint analysis is a form of data flow analysis, wherein an input set of sensitive data is marked as “tainted”, and this taint is tracked during runtime as it spreads into different locations in memory via move, arithmetic, and control operations. Taint analysis results can be used to protect data by clearing tainted variables at the end of its life range—for instance, the temporary key schedule at every round of AES algorithm—or by encrypting live but inactive tainted data [75]. Taint analysis can also help identify and counter abnormal behaviors of malicious malware. Existing dynamic taint analysis has primarily been applied to CPU programs though its functions are increasingly desirable for GPUs as well.

This chapter presents the first design and implementation of a GPU taint tracking system. Our approach is based on static binary instrumentation that enables dynamic taint tracking of a GPU program. In comparison to dynamic instrumentation that captures and modifies instructions on the fly, our approach does not require a dynamic instrumentation framework or virtual machine emulation that is not readily available on GPUs. We perform static instrumentation on GPU program binaries without source access so that it is easy to apply in practice. We instrument programs on a per-application basis and when the program runs, every thread can dynamically track information flow by itself.

The major challenge for efficient taint tracking is that tracking every dynamic binary instruction is expensive. Our solution exploits the fact that a large portion of a typical GPU program execution operates on un-taintable runtime parameters and constants. Examples include the logical thread indexes, thread block identifiers, dimension configurations, and pointer-type kernel parameters. We use a simple filtering policy that the runtime taint tracking only operates on instructions whose operands can be reached from potential global memory taint sources through dependencies and can reach potential global memory taint sinks. We present an iterative two-pass taint reachability analysis to implement such instruction filtering which significantly reduces runtime taint tracking costs.

Our taint tracking system also exploits the heterogeneous memory architecture on GPUs. A GPU has different types of memory, including either physically partitioned or logically
partitioned memory storage. For instance, local memory is private to every thread, shared memory is a software cache visible to a group of threads, and global memory is visible to all threads. Our taint system handles different types of memory storage separately and optimizes the tracking for different types of memory storage. Specifically, we allocate a portion of the register file to store part of the taint map, since GPU contains a much larger register file than CPU—e.g., every streaming multi-processor (SM) has 64K registers on most NVIDIA GPUs. Not all registers are needed [20, 49], and we demonstrated in Chapter 3 that the maximum occupancy is not necessary for best performance. Using fast access registers to maintain the taint map of frequently accessed data will improve the dynamic tainting performance.

GPU taint tracking enables data protection that clears sensitive (tainted) data objects at the end of their life range as well as detects leak of the sensitive data in the midst of program execution. We recognize that data in different GPU memory storage may have different life ranges. For instance, registers and local memory are thread private and can be cleared once a thread finishes its execution; shared memory is only used by a thread block and sensitive data in shared memory can be cleared by that thread block once it releases the SM. Global memory may be accessed at any time of a program run so we cannot clear it at the end of every kernel execution. However, we can detect when and where the sensitive information (in global memory) is sent out by instrumenting memory communication APIs since all communication between GPU, CPU and other network devices require explicit memory API calls. By checking if the sensitive information falls within the region of memory that is transferred, we can identify GPU malware (like Keylogger [42] and Jellyfish [39]) that uses GPU to snoop CPU activities while storing these activities in GPU memory. Such GPU-resident malware would escape detection by a CPU-only taint tracking mechanism.
5.2 Efficient GPU Taint Tracking

A typical information flow tracking system on CPUs monitors instructions and operands to maintain proper taint propagations. For example, in a binary operation \( v = \text{binop} \ v_1, v_2 \), assuming \( T(v_1), T(v_2), \) and \( T(v) \) represent the taint status for operands \( v, v_1, \) and \( v_2 \) respectively: true means tainted and false means untainted. The taint tracking rule for this instruction is \( T(v) = T(v_1) \ || \ T(v_2) \). Taint statuses for all data storage locations (program memory, registers, conditional flags, etc.) are maintained in a taint map in memory. A baseline GPU taint tracking system would operate in a similar way.

Dynamic taint tracking [13, 14, 18, 33, 61, 71, 83, 90] is known to incur high runtime costs. Fortunately, GPU executions exhibit some unique characteristics that enable optimization. We present an optimization that recognizes and identifies the large portion of GPU instructions that cannot be involved in taint propagation from sources to sinks. Furthermore, given the large register file on GPU and frequent register accesses, we maintain register taint map in registers to accelerate their taint tracking. These optimizations are performed through binary-level static analysis.

5.2.1 Taint Reachability

On GPUs, we discover that programs frequently operate on a set of critical runtime untaintable values, and that not all operands need to be tracked. We exploit this fact and only track the operands that potentially carry taints or may have an impact on the state transition of the un-taintable objects. In the earlier example, if \( v_1 \) does not carry any taint, the taint maintenance only needs to track \( v_2 \) and \( v \) such that \( T(v) = T(v_2) \). If neither \( v_1 \) or \( v_2 \) can be tainted, or if \( v \) does not propagate to memory, no taint maintenance is necessary for the variables \( v, v_1, \) and \( v_2 \).

A frequently used GPU runtime un-taintable is the logical thread index. A thread index is used to help identify the task that is assigned to every thread. It is a built-in variable, and does not come from global memory that is managed by a programmer, and thus the
instruction operand as a thread index or an expression of thread index can never be tainted. Similarly, other built-in thread identification variables, including thread block id and dimension configuration, are also untaintable.

Another frequently used GPU runtime untaintable value are the non-scalar pointer-type kernel parameters. A GPU kernel function does not allow call-by-reference. To reference a memory data object that can be modified at runtime, it can only use pointers. Moreover, these kernel parameters are kept in a memory region named as “constant memory” in GPUs and are read-only in kernel execution. The memory region pointed to by the kernel parameter must be tracked, but the pointer or the address expression computed using the pointer and thread index (or part of the expression) does not need to be tracked. Other examples include compile-time untaintable values, such as loop induction variables and stack framework pointers, programmer-specified constants, and combinations of GPU-specific runtime constants with these constants. We analyze and categorize these untaintable values in Section 5.4 and Table 5.1.

To avoid tracking untaintable values in GPU programs, we take the following approaches.

1. We classify an instruction operand into two types: taintable and untaintable. The taintable state indicates that the operand might be tainted at runtime—whether it will be really tainted depends on the exact dynamic analysis done by tracking instructions. The untaintable state indicates that the operand cannot be tainted at runtime. Any operand that cannot be reached from the taintable source is untaintable. The taintable sources are program inputs given by the users and reside in the global memory on GPUs. Examples include face recognition photos, a plain-text message, and encryption key.

2. A variable can be overwritten with taintable or untaintable values at different program execution points. We check for potential state transition of a variable: from untaintable to taintable, or from taintable to untaintable. The latter arises in a situ-
ation called *taint removal*—e.g., assigning a constant to a register who might be in a tainted state before the assignment but must now transition to the un-tainted state.

3. We statically check the memory reachability: whether an operand might reach memory (potential taint sinks). Even if an operand is taintable, as long as it does not flow into memory, it will not affect any taint sink. We do not need to add tracking instruction for this type of operands. Common examples include loop trip counters, predicate registers, and stack frame pointers.

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Tracking Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: block0:</td>
<td>1:</td>
</tr>
<tr>
<td>2: R0 = 0x1234;</td>
<td>2:</td>
</tr>
<tr>
<td>3: R0 = 0x0;</td>
<td>3: T(R0) = false;</td>
</tr>
<tr>
<td>4: if (some_condition)</td>
<td>4:</td>
</tr>
<tr>
<td>5: R0 = [R1];</td>
<td>5: T(R0) = T([R1])</td>
</tr>
<tr>
<td>6: [R2] = R0;</td>
<td>6: T([R2]) = T(R0);</td>
</tr>
<tr>
<td>7: some_condition = random();</td>
<td>7:</td>
</tr>
<tr>
<td>8: GOTO block0;</td>
<td>8:</td>
</tr>
</tbody>
</table>

We show an example in the code snippet above. The code describes a loop. Register R0 is overwritten with different types of values. Initially R0 is written with an un-taintable value (lines 2-3). Later in the *if* statement, it is written by a taintable value [R1]; note that here the [R1] notation indicates a memory operation and the address of the memory location is R1. We need a tracking instruction within the *if* statement since [R1] comes from global memory and every operand from memory needs to be tracked. We do not need a tracking instruction for line 2 since 0x1234 is a constant and the assignment target R0 at line 2 cannot reach memory. However, we do need a tracking instruction for line 3 since the assignment target may reach memory and taint removal applies here (R0 may be tainted from an earlier iteration of the loop and if so, taint must be removed here).
5.2.2 Iterative Two-pass Taint Analysis

To mark the taintability and reachability attributes for every operand and to detect potential taint state transition, we perform an iterative dataflow analysis.

There are two passes in our iterative dataflow analysis component. The forward pass marks the taintable operands and the un-taintable operands only at the program points where a potential taint state transition occurs. The backward pass marks an operand that potentially reaches memory (taint sinks). In the end, when adding code to track the original program, we only track the operands that are marked in both forward and backward passes.

Figure 5.2 provides an overview of our taint tracking system. First, we analyze the binary code to obtain the control flow graph and a list of basic blocks. A basic block is the maximum length single-entrance and single-exit code segment. We also mark the operands that are known to be un-taintable before the program starts. They include built-in thread identification variables, non-scalar pointer type kernel parameters, and other programmer-specified constants.

We perform the backward pass first to analyze each operand and set its memory reach-
Forward Filtering Pass:

Every operand in every instruction will be assigned a TaintTrack state: this is set to true if the operand might be tainted or its taint state might be changed.

1. forward_pass(program P)
2. L = all basic blocks of P:
3. while (worklist L not empty)
4. i = dequeue(L)
5. r = i.taintTrackBegin;
6. forward_prep(i, r);
7. for (each successor block j of i)
8. if (r != j.taintTrackBegin)
9. j.taintTrackBegin = j.taintTrackBegin | r;
10. enqueue(L, j);

Figure 5.3: Forward taint reachability analysis.

ability attribute. We name it the mightSpread attribute, indicating whether there exists an execution path through which the value of this operand might spread into memory.

We then perform the forward pass to mark all operands as taintable or un-taintable, and for every un-taintable operand, we also analyze if its last immediate state is taintable in one of the potential execution paths. If an operand is taintable or its last immediate state is taintable, we set the taintTrack attribute to be true. The taintTrack attribute indicates that the operand may be tainted at runtime. For an indirect memory operand, we also need an attribute on the taintability of the addressing register. We call this addrTrack attribute.

Finally, in the Tracking Filter component, we scan all instructions and review the taintability and reachability attributes each operand. For the destination operand, if its taintTrack and mightSpread attributes are both true, we add tracking code for this destination operand, otherwise we don’t. Similarly, for source operands, if both of its taintTrack and mightSpread attributes are true, we add tracking code for the source operand before the tracking code for the destination operand. For an indirect memory source operand, if its addrTrack and mightSpread attributes are both true, we add taint tracking code for the source operand addressing register.

We describe the detailed algorithms for forward and backward passes below.

Forward Taint Reachability Analysis The input is a control flow graph and a set of basic blocks for the GPU program. The output is the taintTrack property value for every operand in every instruction. We show the forward pass algorithm in Figure 5.3(a).
We adopt the fixed-point computation algorithm that is used in standard dataflow analysis (DFA) framework. Function \textit{forward\_pass} in Figure 5.3(a) scans the basic blocks one by one, sets the taintTrack attribute for every operand, and updates the \textit{taintTrackBeg} attribute for every basic block. Our forward analysis pass checks if one basic block’s taintability updates affect another basic block’s taintability results, and if so, adds the affected basic block to the worklist. Initially, all basic blocks are added to the list. The analysis pass finishes only when all basic block’s taintability results do not change.

A DFA problem is formulated using (a set of) dataflow equation(s). We describe the dataflow equation as follows. The \textit{taintTrackBeg} attribute describes the taint tracking state of every register at the beginning of a basic block, which is a bit array. Every bit in the bit array corresponds to one physical register. If a register’s taintTrack attribute is true at the beginning of the basic block of interest, this bit is set to 1, otherwise 0. Assume a basic block \( P \) and it has \( n \) predecessor basic blocks \( Q_i \), \( i = 1...n \), the dataflow relation is

\[
P.taintTrackBeg = \bigcup \text{forward\_prep}(Q_i, Q_i.taintTrackBeg).
\]

The \textit{forward\_prep} function in Figure 5.3(b) updates the taintability state for all instructions in a basic block based on the taintability state at the beginning of the basic block. It scans the first instruction to the last instruction.

Given an instruction, the \textit{forward\_prep} function checks its source operands first (lines 3–6 in Figure 5.3(b)). If a source operand is register and the taintTrack attribute is true, this source operand needs to be tracked. If a source operand is of memory type, it has to be tracked. Note that if the address register of an indirect memory operand is taintable, we need to track the register as well—setting addrTrack attribute at line 6 in Figure 5.3(b).

Next, the \textit{forward\_prep} function checks every destination operand. If any source operand needs to be tracked based on the above analysis, destination operand needs to be tracked as well. In the meantime, we update the register tracking state for the corresponding destination operand (line 10 in Figure 5.3(b)). If the destination operand is of memory type, it needs to be tracked. If the destination operand is un-taintable (lines 13-15 in Figure 5.3),
and its prior tracking state is taintable, and the destination operand might spread to memory, the destination operand needs to be tracked as well. Further we update the register tracking state for the corresponding destination operand.

<table>
<thead>
<tr>
<th>code</th>
<th>regTaintState</th>
<th>taintTrack</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 = R1 + R2;</td>
<td>[1,1,0,0];</td>
<td>taintTrack(R0, R1, R2) = {true, true, false};</td>
</tr>
<tr>
<td>R1 = 0x5000;</td>
<td>[1,0,0,0];</td>
<td>taintTrack(R1) = true;</td>
</tr>
<tr>
<td>R2 = [R1];</td>
<td>[1,0,1,0];</td>
<td>taintTrack(R2, R1) = {true, false};</td>
</tr>
<tr>
<td>R3 = R0 + 0x1;</td>
<td>[1,0,1,1];</td>
<td>taintTrack(R3, R0) = {true, true};</td>
</tr>
</tbody>
</table>

```
R0 = R1 + R2;
R1 = 0x5000;
R2 = [R1];
R3 = R0 + 0x1;
BRA block5;
```

taintTrack(R0, R1, R2) = {true, true, false};
taintTrack(R1) = true;
taintTrack(R2, R1) = {true, false};
taintTrack(R3, R0) = {true, true};

Figure 5.4: Backward memory reachability analysis.

We use the above example to illustrate the `forward_prep` step for updating the register tracking state. Let the initial regTaintState be [0, 1, 0, 0], meaning that only register R1 is found to be taintable on entry to this basic block. Since the first instruction has R1 as a source and R0 as a destination, we set the operand’s taintTrack flag and regTaintState[0] to true.

Since the second instruction writes an immediate value to R1, but since regTaintState[1] was previously true, we have to set the operand’s taintTrack flag to true, if its result can spread to memory. This instruction potentially changes the taint value of R1 at runtime from true to false, so if it can reach memory, then we need to instrument it, or else we will suffer from over-tainting as a result of incorrectly treating the data as still being tainted. We flip regTaintState[1] to false since at compile-time and at the second instruction, register R1 is untaintable.
The next instruction loads from memory into R2, so we set the operands’ taintTrack flags and regTaintState[2] to true, because memory is a possible taint source. The final instruction before the branch carries potential taint from R0 to R3; since regTaintState[0] is true, regTaintState[3] is set to true along with the operand’s taintTrack flag.

**Backward Memory Reachability Analysis** Similar to the forward pass, the backward pass uses the program as input. The output is the memory reachability property of every operand. The backward reachability analysis also uses a dataflow analysis framework, which solves the $mightSpreadBeg$ bit array for every individual basic block, representing the memory reachability state of the registers at the beginning of basic block. In this bit array, each bit corresponds to one physical register. A value of 1 for the bit at index $n$ of basic block $b$ means that the value of register $Rn$ at the beginning of basic block $b$ might reach memory.

The relationship between one basic block $P$ and its successor basic blocks $Q_i$, $i = 1..m$, where $m$ is the total number of immediate successor basic blocks, is described using the following equation:

$$P.mightSpreadBeg = \text{backward}_{\text{prep}}(\bigcup Q_i.mightSpreadBeg).$$

The initial $mightSpreadBeg$ bit array is set to 0 for every basic block. Our backward pass keeps updating the $mightSpreadBeg$ bit arrays until they do not change any further (Figure 5.4(a)). In the meantime, the attribute $mightSpread$ is updated for every operand, as described in Figure 5.4(b).

The $backward_{\text{prep}}$ function calculates $mightSpreadBeg$ for every individual basic block. In Figure 5.4(b), we scan the instructions in reverse order in a basic block. First, we check the destination operand, if it is register type and the register’s memory reachability state is true, the destination operand’s $mightSpread$ attribute is set to true. In the meantime, we update the register’s memory reachability state for the destination register to false since the value to spread into memory is defined at this point and for any instruction that happens before this instruction, they don’t see the same value as defined here. If it is memory type,
the mightSpread attribute is set to true and the address register’s reachability state is set to true (line 7 in Figure 5.4(b)). Next, we check the source operands. If any destination operand can spread into memory, then all source operands’ mightSpread property is set to true (line 10). Correspondingly, we will set the register reachability state to true (line 11).

\[ \text{regSpreadState is initially [0, 1, 0, 1].} \]

block4:

\[
\begin{align*}
\text{R0} &= \text{R1} + \text{R2}; & \text{regSpreadState} &= [1, 1, 0, 0]. \\
\text{R1} &= 0x6000; & \text{regSpreadState} &= [1, 1, 1, 0]. \\
[\text{R1}] &= \text{R2}; & \text{regSpreadState} &= [1, 0, 1, 0]. \\
\text{R3} &= \text{R0} + 0x1; & \text{regSpreadState} &= [0, 1, 1, 0]. \\
\text{BRA block6;} & \\
\end{align*}
\]

We use the above example to illustrate the process for updating the mightSpread bit arrays in the backward pass. The backward pass is mechanically similar to the forward pass, aside from the direction in which instructions are processed. In this example, we assume that registers R1 and R3 have been determined to spread to memory in later blocks, hence the initial regSpreadState value of [0, 1, 0, 1]. We skip over the branch instruction since it has no operands except for a jump offset.

The last instruction has data flow into R3 from R0, and regSpreadState[3] is true, so we mark the R0 operand’s mightSpread flag as true and set regSpreadState[0] to true. We also flip regSpreadState[3] to false since this instruction is overwriting R3.

The instruction before the last stores register R2 to memory, so we simply mark the R2 operand’s mightSpread flag as true and set regSpreadState[2] to true.

The third instruction counting from the last puts an immediate value into R1, so we set regSpreadState[1] to false. Finally, the fourth instruction counting from the last has data flow into R0 from R1 and R2, and regSpreadState[0] is true, so we mark both source operands’ mightSpread as true, set regSpreadState[1] and regSpreadState[2] to true, and set regSpreadState[0] to false since R0 has been overwritten.
5.2.3 Register Taint Map in Registers

A GPU contains a much larger register file than CPU does—e.g., every streaming multi-processor has 64K registers on most NVIDIA GPUs. Registers are naturally accessed frequently and maintaining their taint statuses require frequent reads and writes from/to their taint map locations. At the same time, the large GPU register file presents the opportunity to maintain a portion of the taint map in registers. These facts motivate us to place the register taint map in registers.

We use multiple 32-bit general purpose registers to store the taint map, in which one bit corresponds to one register that is tracked. Using register-stored taint map increases the number of registers used per-thread, and might decrease occupancy, determined as the number of active threads running at the same time. Fortunately in many GPU programs, not all the register file is needed [20, 49] nor the maximum occupancy is necessary [32] for the best program performance. Therefore the overall taint tracking cost is significantly reduced by our use of register-stored taint map, as demonstrated later in evaluation.

5.3 Tainting-Enabled Data Protection

Taint tracking results can be used to help protect sensitive data and prevent information leaking on GPUs. We describe two major use cases of taint tracking analysis and present our prototype implementation of tainting-enabled data protection.

5.3.1 Sensitive Data Removal

Lee et al. [45] and Pietro et al. [69] have recently demonstrated that information leaking from one program to another may occur in GPU local memory between GPU kernel executions, and in GPU global memory between program runs. Our taint tracking results may help a program understand the propagation of certain sensitive information and clear all taints before relevant points of vulnerability (e.g., clearing local memory taints at the end of each kernel and clearing global memory taints at the end of program run).
We make a prototype implementation of this use case. For registers, we let every thread clear its own tainted registers. It is possible that some threads exit earlier than others. However since register taint map is thread-private, we can insert the clearing code right before every *EXIT* point and thus early-exiting threads can also clear their tainted registers early. For local memory, since it is thread-private, we treat it the same way as registers. Note that registers and local memory cannot be cleared by programmers themselves (unlike shared memory and global memory) and thus a trustworthy binary instrumentation tool is necessary to prevent sensitive data from leaving taints on GPUs.

For shared memory, since shared memory is visible to all threads in the same basic block, we need to make sure the sensitive shared memory data is cleared *after* all threads in the same thread block finish their work. Therefore, our design is to create a control flow reconvergence point for all threads since different threads might take different execution paths. We then insert a thread block level barrier at the reconvergence point before clearing the tainted shared memory data.

Pietro et al. [69] proposed a register-spilling based attack, which makes use of compiler to force spilling the registers so that the encryption key (or reversibly transformed encryption key) in the AES encryption module in the *SSLShader* program can be moved from registers to local memory. Then a second running application can steal the leaked information in local memory. Our taint clearing approach prevents such attacks by clearing the registers, local memory, and shared memory right before every thread in the GPU application completes.

Experimental results in Section 5.4 will show that the data clearing cost is low—worst-case slowdown of 13% and in most cases no more than 5% slowdown.

5.3.2 GPU Malware Countermeasure

GPU taint analysis identifies where and when sensitive data is sent from GPU device to CPU or other network devices. This is especially important for integrated CPU-GPU whole system taint tracking. A dynamic taint tracking system that only monitors data dependences
during CPU execution may miss the influence propagation of untrusted inputs or execution results through GPU computation. For example, GPU malware Keylogger [42] and Jellyfish [39] exploited direct memory access (DMA) at mapped CPU memory to snoop the CPU system activities and steal host information. GPU may obtain the leaked CPU information, process it, and send it through a network or other output device while evading countermeasures that only monitor CPU executions.

Our GPU data protection system can not only clear sensitive data, but also capture possible attempts of stealing and emitting sensitive information. We prevent this type of attacks by dynamically monitoring the data transfer between CPU and GPU. If the GPU-mapped CPU memory contains sensitive information (i.e., keystroke buffer in the Keylogger attack [42]), the mapped data region is marked as taint sources. We track the dependency propagation of tainted data in GPU executions. Further we statically instrument memory transfer APIs so that before any data is sent from GPU through cudaMemcpy APIs in CUDA or clEnqueueReadBuffer APIs in OpenCL, the memory address range is checked. If the transmitted data falls within the sensitive tainted memory range, we either alert the system that tainted data is transmitted, or mark the corresponding CPU destination memory region (if data is transferred back to CPU) as tainted. Since all communication between GPU and other devices rely on explicit memory transfer API, we can check and protect information flow by instrumenting these memory transfer APIs.

Our taint tracking and data protection system helps protect applications that utilize both CPU and GPU, if combined with CPU taint tracking. Our work ensures full system taint tracking that is essential to whole system security. We are not aware of any other work that provides the same degree of protection. Besides the Keylogger case, other potential whole-system tracking examples include a web site that relies on taint tracking to prevent untrusted user inputs with malicious database queries (e.g., through SQL injections) or invoking dangerous system calls (e.g., through buffer overflow attacks).

Finally, when GPU tainting is securely applied to untrusted programs, it can also iden-
tify malicious programs that attempt to scan uninitialized data which may have been left by previous kernel and program runs from other users. We have not implemented this type of data protection. However, our tool can be readily extended to help detect uninitialized memory region containing sensitive data left by prior GPU program execution and clear/zero them if appropriate.

5.4 Evaluation

We perform evaluation on a machine configured with an NVIDIA GTX 745. This is a “Maxwell” generation GPU with compute capability 5.0. Since NVIDIA’s compiler and binary ISA are closed-source, we modify the GPU binaries using tools inspired by the asfermi [36] and MaxAs [22] projects, allowing for binary instructions be directly inserted into the executable.

5.4.1 Benchmarks

Our evaluation employs a variety of GPU kernels in deep learning, image processing, and data encryption. First, Caffe [40] is a deep learning framework in which a user writes a “prototxt” file describing the input and layers of the deep learning network (e.g., convolutional layers, inner-product layers, etc.), which can be fed into the Caffe executable to create, train, and test the network. Newer versions of Caffe allow various layers to be executed on the GPU via CUDA. A common use of Caffe is image classification. We use three Caffe kernels in our evaluation: im2col, ReLUForward, and MaxPoolForward. These three kernels consume the majority of the execution time for image classification.

We additionally use kernel functions from the CUDA SDK [64], the Rodinia benchmark suite [9], and SSLShader [37]. From the CUDA SDK we include BlackScholes, a program for financial analysis, and FDTD3d, a 3D Finite Difference Time Domain solver. As a numerical analysis program, FDTD3d is unlikely to have sensitive data to protect, but serves as an additional data point for testing our performance. From Rodinia, we in-
clude Needleman-Wunsch, a bioinformatics benchmark used for DNA sequencing. From SSLShader, we include an AES encryption program.

5.4.2 Taint Analysis & Optimizations

We evaluate the effectiveness of the two performance enhancing techniques in Section 5.2—taint reachability filtering and taint map in registers.

Since we modify the executable directly, we measure the cost of taint analysis in terms of both slowdown and code size. There are a few factors which exacerbate these costs. Whenever we insert an instruction to get or set a location’s taintedness in memory, we first have to calculate its address. Since addresses for global memory are 64-bit on this architecture, but registers and integer operations are 32-bit, this requires multiple instructions with immediate dependencies.

Additionally, each thread has access to only one carry flag, so if it is already in use where we need to get the taint address, extra instructions are needed to spill it into a register or to memory. Furthermore, the singular carry flag makes it difficult to interleave instructions effectively, since they may overwrite each other’s result. As the GPU is incapable of out-of-order execution within a thread, the latency for accessing the taint-map is costly.

Figure 5.5(a) illustrates the GPU tainting slowdown with each of our optimizations, compared to native execution. The ‘naive’ bar shows slowdown without any optimizations, the ‘reg-in-reg’ bar shows the results of placing part of the taint map into registers, the ‘forward-filter’ and ‘backward-filter’ bars show the results of each filter pass, the ‘two-pass-filter’ bar shows results when using both filter passes, and the ‘fully optimized’ bar shows results when using all of these optimizations. Figure 5.5(b) shows normalized code sizes (static instruction counts) for the same cases.

Figure 5.5 shows that both two-pass filtering and hot register taint map can reduce the tainting cost significantly. For the filter passes, there is a high correlation between relative slowdown and code size after instrumentation. Saving taint mapping into registers does
Figure 5.5: Overhead of tainting instrumentation.
not shrink as much of the code size as two-pass filtering, but it still improves the tracking performance significantly. The tracking cost saving comes more from the reduced memory latency than from reduced instruction count.

**Taint Map in Registers**  Even on its own, saving part of the taint map in registers reduces significant time during taint analysis. The main alternatives, local memory and global memory, are both off-chip memories that may take hundreds of cycles to access. Even the cache to which such memory is saved is off-chip, because the on-chip L1-cache is typically only used for read-only data on newer architectures [65]. Since most GPU programs have numerous threads running at once, some of this latency is hidden by some threads continuing to execute while others wait for memory accesses to complete, but even so, saving register taint information into registers reduces slowdown compared to naive taint tracking in our benchmarks by 78% on average.

**Filtering**  The forward pass filtering also saves significant time, though it has more variance across different benchmarks. Its effectiveness stems from the properties of GPU kernels. Most kernels make use of non-taintable, read-only data such as thread ID and grid size to perform many calculations. Additionally, function parameters are read-only in GPU functions, making it impossible for them to become tainted in most programs. On its own, the forward pass reduces slowdown in our benchmarks by an average of 53%.

<table>
<thead>
<tr>
<th>kernel</th>
<th>parameter</th>
<th>immediate</th>
<th>const mem.</th>
<th>thread block id</th>
</tr>
</thead>
<tbody>
<tr>
<td>im2col</td>
<td>85%</td>
<td>85%</td>
<td>29%</td>
<td>64%</td>
</tr>
<tr>
<td>ReLUForward</td>
<td>20%</td>
<td>40%</td>
<td>47%</td>
<td>57%</td>
</tr>
<tr>
<td>MaxPoolForward</td>
<td>70%</td>
<td>72%</td>
<td>57%</td>
<td>58%</td>
</tr>
<tr>
<td>FDTD3d</td>
<td>17%</td>
<td>17%</td>
<td>11%</td>
<td>12%</td>
</tr>
</tbody>
</table>

Table 5.1: Percentage of filtered-out instructions for various reasons.

We also analyze the reason why we are able to filter out a significant number of instructions for some applications in the forward pass. Table 5.1 shows the percentage of filtered out instructions under different categories. *Parameter* means one or more source
registers are from the (constant-memory) kernel parameters. **Immediate** means one or more source operands are immediate numbers. **Const memory** means at least one source is from constant memory. Finally, **thread/block id** means the influence is from the identifier of the current thread or thread block. The identifiers are stored in special registers private to each thread or constant memory depending on the architecture, but in either case they are known at static-time. While it might be surprising that the sum of percentages due to multiple reasons may exceed 100%, note that an instruction may be filtered out due to multiple reasons.

We discover that most instructions are filtered out because of these four categories. The reason is that GPU programs distribute workload among threads based on their ids. To get the assigned workload, each thread must perform a lot of computation using ids, immediate, and constant memory values (e.g., thread block & grid dimensions). The computation results, together with parameters (e.g., the start address of an array), are used to fetch assigned data. Then the real computation starts as well as the taint tracking. For most GPU programs, the real computation is short with several instructions, and the preprocessing including address calculation consumes most of the time. That is why we can filter out most instructions in our forward pass: most instructions do preparation work and are not related to the potentially tainted input data. For FDTD3d, the computation is more complex and fewer instructions are filtered out. It also explains why FDTD3d does not benefit from two-pass filtering as much as compared with other benchmarks, as shown in Figure 5.5(a).

The backward pass is usually less effective than the other optimizations. While a lot of the inputs to a kernel function are effectively constants, the only means of returning anything is through global memory. As such, we can expect that most operations will produce values which influence memory. Regardless, the backward pass does provide some benefit in most cases, and in the SSLShader benchmark it reduces slowdown compared to the naive approach by 22%.
**Combined Optimizations**  Compared to the forward pass, the two-pass filter reduces slowdown by 12% on average, and compared to the backward pass, it reduces slowdown by 50%. Full optimization reduces slowdown by 56% compared to the two-pass filter, and 42% compared to only keeping part of the taint map in registers. This demonstrates the merit of combining our different optimization techniques, which together reduce slowdown by an average of 87%.

With full optimizations, our benchmarks’ kernel functions experience an average normalized runtime of 3.0× after instrumentation. The FDTD3d benchmark suffers the worst slowdown at 5.7× runtime, due to frequent use of shared memory making the filter less effective. The Needleman-Wunsch benchmark, which also has shared memory usage, is the next slowest with a 3.6× runtime. Although the SSLShader benchmark also makes use of shared memory, it only uses shared memory to store compile-time constants for faster retrieval, allowing us to filter out all shared memory instructions for less runtime slowdown of 2.5×.

One special consideration when modifying GPU programs is occupancy—the number of threads that can be live at once. A high occupancy means that latency is less costly, as the GPU can switch to different groups of threads every cycle. Since our instrumentation results in additional use of registers, and the register file is evenly split among all live threads, there is potential for occupancy to be decreased, hurting performance more drastically. In such a case, it may be more beneficial not to store any part of the taint map into registers. However, in practice, we use few enough additional registers that reducing occupancy is unlikely, since for every 32 registers in the original program, we only need 1 extra register to store their taintedness. We find that GPU programs typically use less than 64 registers per-thread, and so none of our benchmarks require more than two extra registers per-thread for storing register taintedness.
<table>
<thead>
<tr>
<th>GPU kernel</th>
<th>Memory</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>im2col</td>
<td>N/A</td>
<td>0.26%</td>
</tr>
<tr>
<td>ReLUForward</td>
<td>N/A</td>
<td>0.33%</td>
</tr>
<tr>
<td>MaxPoolForward</td>
<td>N/A</td>
<td>0.59%</td>
</tr>
<tr>
<td>FDTD3d</td>
<td>Shared</td>
<td>5.10%</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>N/A</td>
<td>0.40%</td>
</tr>
<tr>
<td>SSLShader</td>
<td>Local</td>
<td>0.41%</td>
</tr>
<tr>
<td>needle</td>
<td>Shared</td>
<td>13.05%</td>
</tr>
</tbody>
</table>

Table 5.2: Slowdown from memory erasure during kernel execution, measured as a fraction of the original kernel time. "Memory" column indicates which memory types need to be cleared (besides registers).

5.4.3 Memory Protection

We next evaluate the incorporation of memory protection into our dynamic analysis framework. As discussed in Section 5.3, the GPU does not clear memory before deallocation. This includes all types of memory, both on-chip and off-chip. [69] demonstrates that data left behind even in local memory and shared memory can be stolen, such as the encryption key and plaintext in the SSLShader benchmark. We have found that this data can also be stolen directly from registers by preparing a kernel function with the same thread block size and occupancy as the victim kernel function—thereby ensuring the register file will be partitioned in the same, predictable manner—and then manually coding the eavesdropping kernel’s binary to read the desired registers.

Programmers can manually erase global memory before program exit, but registers and local memory are allocated by the compiler and cannot be as easily cleared. Sensitive data in registers, local memory, and also shared memory must be cleared before the kernel function exits, or else a malicious kernel function may be invoked and acquire these resources for itself. We leverage our instrumentation framework to clear sensitive data in these regions, via additional modification to the binary code. This can be used to prevent attacks such as the one in [69], which stole encryption key data through such resources. The results are summarized in Table 5.2.
Since registers and local memory are thread-private, they can be safely cleared by each thread prior to exit. We insert instructions to clear this data before the EXIT instruction, using the results of our forward-filter pass to avoid unnecessary work. But shared memory is shared by every thread in a thread block, and therefore may not be safe to erase until all of its threads finish execution. Before the EXIT instruction we insert a synchronization barrier, which causes threads to halt until all other threads in the block reach the same point, and then add a loop which has every thread zero out a separate portion of shared memory. In benchmarks with less regular control flow, where threads exit at different points in the code, we can instead have shared memory cleared by a subset of its threads.

We find that the cost to clear tainted registers is trivial, adding only a fraction of a percent to runtime. Each register takes only one cycle of amortized time to erase for every 32 threads, and the GPU is likely able to overlap most of these cycles with memory stalls from other threads. None of our benchmarks use local memory by default, since it is usually used for register spilling. In order to evaluate the slowdown of clearing local memory, we recompile SSLShader, which uses 40 registers, to instead use 20 registers. Clearing local memory and registers in this benchmark adds 0.41% time overhead.

Shared memory is slower to clear. In FDTD3d, clearing taints in shared memory adds 5.10% runtime compared to the original kernel function, and in Needleman-Wunsch it adds 13.05%. The increased slowdown compared to clearing local memory likely stems from the use of a loop, due to the GPU’s inability to perform speculative and out-of-order execution, forcing a thread to wait until each shared memory location is cleared until it can zero the next one. Local memory is simpler to handle, with every thread accessing the same logical addresses despite using different physical locations, allowing for the local memory clearing loop to be fully unrolled.

Using the taint information to erase only sensitive data can help significantly, compared to naively clearing these memories fully. For example, in the SSLShader benchmark the tainted registers and local memory are cleared in 47 mSecs, but this benchmark makes
use of shared memory which is never tainted. If its shared memory arrays are erased, in addition to clearing the small amount of registers and local memory in their entirety, then the overhead would jump to 407 mSecs.

5.5 Related Work

Dynamic taint analysis [13, 14, 18, 33, 61, 71, 83, 90] tracks data (and sometimes control) dependencies of information as a program or system runs. Its purpose is to identify the influence of taint sources on data storage locations (memory, registers, etc.) during execution. Taint tracking is useful for understanding data flows in complex systems, detecting security attacks, protecting sensitive data, and analyzing software bugs. Its implementation usually involves static code transformation, dynamic instrumentation, or instruction emulation using virtual machines to extend the program to maintain tainting metadata. While existing dynamic tainting systems track CPU execution, this chapter presents the first design and implementation of a GPU taint tracking system.

A large body of previous work presented techniques to improve the performance of CPU taint tracking. LIFT [71] checks whether unsafe data are involved before a code region is executed, and if not, no taint tracking code is executed for that code region to reduce overhead. Minemu [5] proposes a novel memory layout to reduce the number of taint tracking instructions. It also uses SSE registers for taint tracking to reduce performance overhead. TaintEraser [90] makes use of function summary to reduce the performance overhead of taint tracking. It summarizes taint propagation at the function level so that instruction level taint tracking is reduced. TaintDroid [18] is a taint analysis tool proposed for Android systems. By leveraging Android’s virtualized execution environment and coarse-grained taint propagation tracking, it can achieve nearly real time analysis with low performance overhead. Jee et al. [38] proposed to separate taint analysis code from the original program, and dynamic and static analysis was applied on the taint analysis code to optimize its per-
formance. In this chapter, we present new performance optimizations by exploiting unique GPU characteristics.

Security vulnerabilities on GPUs have been recognized recently. Dunn et al. [17] showed that sensitive data can be leaked into graphics device driver buffers. They proposed encryption to protect data in transit over the device driver but their approach does not protect data in GPU memory. Lee et al. [45] uncovered several vulnerabilities of leaking sensitive data in GPU memory—leaking global memory data after a program context finishes and releases memory without clearing; leaking local memory data across kernel switches on a CU. They did not present any solution to address these vulnerabilities. More recently, Pietro et al. [69] proposed memory zeroing to prevent information leaking in GPU. However, memory zeroing alone provides limited protection—it cannot track information flow in memory; nor can it counter GPU malware such as Keylogger [42] and Jellyfish [39]. Furthermore, GPU tainting is complementary to memory zeroing—tainting identifies a subset of sensitive memory for zeroing to reduce the costs.

GPU information flow analysis has been performed in the past. Leung et al. [46] and Li et al. [50] employed static taint analysis to reduce the overhead of GPU program analysis and verification. Static analysis requires memory aliasing analysis of memory accesses that are inherently imprecise. While they are suitable for testing and debugging purposes [46, 50], security data flow analysis in this chapter requires more precise dynamic tracking. Farooqui et al. [19] proposed static dependency analysis between thread index and control conditions to identify possible thread divergence in GPU executions (the result of which helps determine whether symbolic execution can be performed on given GPU basic blocks). Their static dependency analysis is narrowly targeted and it is unclear whether it applies to general taint tracking.
5.6 Conclusion

Recent discoveries of information leaking through GPU memory and GPU-resident malware call for systematic data protection in GPUs. In this chapter we present the first design and implementation of a dynamic taint tracking system for GPU programs. We exploit unique characteristics of GPU programs and architecture to optimize taint tracking performance. Specifically, we recognize that a large portion of instructions on GPU runtime parameters and constants can be safely eliminated from taint tracking to reduce tainting costs. We also utilize the large GPU register file for fast maintenance of the taint map for registers. These optimizations result in 5 to 20 times tainting speed improvement for a range of image processing, data encryption, and deep learning applications.
CHAPTER 6
CONCLUSION

This dissertation explores binary analysis and instrumentation on the GPU. We start from the ground up, demonstrating the methodical creation of the initial framework, and then using it to develop and evaluate multiple applications that rely on access to the binary code.

In Chapter 2, we learned how to take apart the executable program, inject modified binary GPU code, and rebuild the program. We studied the structure of the code, and demonstrated a means of automating most of the work of building an assembler to translate the low-level assembly code to binary. The framework we built with this technique is open source.

In Chapter 3, we leveraged our framework to improve performance of GPU programs, reallocating registers across the on-chip memories. In Chapter 4 we expanded on this, improving our allocation and producing a framework that utilized static and dynamic tuning to find a more effective trade-off between resource allocation and concurrency. In this manner, we are able to achieve superior results in terms of speed and energy usage.

In Chapter 5, we applied our binary analysis framework to the problem of dynamic taint tracking on the GPU. To current CPU taint tracking programs, the GPU is a black-box through which sensitive data can unpredictably spread, but with our binary instrumentation work it becomes possible to perform taint tracking across both CPU and GPU. This is especially important due to the relative lack of memory protection on GPU devices.

Altogether, we demonstrate the utility of binary analysis and instrumentation on the GPU. The techniques developed in this dissertation would not have been possible without access to the binary, and our evaluations show that these techniques are effective. Furthermore, the nature of our assembler generation technique enables us to add support for later generations of GPUs.
REFERENCES


[77] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioan-
nidis, “PixelVault: Using GPUs for securing cryptographic operations,” in *Proc. of
the 21st ACM Conf. on Computer and Communications Security (CCS)*, Scottsdale,

[78] Anand Venkat, Mary Hall, and Michelle Strout, “Loop and data transformations for
sparse matrix code,” in *Proceedings of the 36th ACM SIGPLAN Conference on Pro-
gramming Language Design and Implementation*, ser. PLDI 2015, Portland, OR,


[80] Jian Wang, Andreas Krall, M Anton Ertl, and Christine Eisenbeis, “Software pipelin-
ing with register allocation and spilling,” in *Proceedings of the 27th annual interna-
tional symposium on Microarchitecture*, ser. MICRO 27, San Jose, California, United


[82] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas

[83] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar, “Taint-enhanced policy en-
facement: A practical approach to defeat a wide range of attacks,” in *Proc. of the

[84] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou, “A GPGPU compiler for
memory optimization and parallelism management,” in *Proceedings of the 31st ACM
SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI
1806596.1806606.

[85] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou, “Shared mem-
ory multiplexing: A novel way to improve GPGPU throughput,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Tech-
niques*, ser. PACT ’12, Minneapolis, Minnesota, USA: ACM, 2012, pp. 283–292,
http://doi.acm.org/10.1145/2370816.2370858.


