RICE UNIVERSITY
Performance Measurement, Analysis, and Optimization of GPU-accelerated Applications

By

Keren Zhou

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

John Mellor-Crummey
Professor of Computer Science and of Electrical and Computer Engineering

Xu Liu
Associate Professor of Computer Science
North Carolina State University

Alan Cox
Professor of Computer Science and of Electrical and Computer Engineering

Joe Cavallaro
Professor of Electrical and Computer Engineering

HOUSTON, TEXAS
April 2022
ABSTRACT

Performance Measurement, Analysis, and Optimization of GPU-accelerated Applications

by

Keren Zhou

With the end of Moore’s law, computing platforms are increasingly exploring heterogeneous processors for acceleration. Graphics Processing Units (GPUs) have emerged as a key component for accelerating applications in various domains, including deep learning, data analytics, and scientific simulations. While GPUs provide superior compute power and higher memory bandwidth than CPUs, writing efficient GPU code to achieve maximum possible performance is challenging because of the sophisticated programming models and architectural features. GPU performance tools are designed to pinpoint performance bottlenecks in GPU-accelerated applications and provide performance insights for users. However, existing performance tools are insufficient to identify hotspots and provide insights for complex applications.

This thesis describes novel GPU performance tools that measure and analyze GPU-accelerated applications to address these challenges. First, I describe a GPU profiler that uses API interception, instruction sampling, and binary instrumentation to collect GPU performance metrics. To lower the overhead caused by the profiler, I designed novel wait-free queues for communication between multiple threads, a GPU-accelerated method to process measurement data, and metrics derivation method that derives multiple essential GPU performance metrics without replaying GPU operations. Then, I present a framework that attributes measurement data collected at runtime to call paths with low overhead. Offline, I developed a binary analyzer that
reconstructs approximate GPU calling contexts by analyzing instruction samples and GPU binaries. Also, the analyzer analyzes def-use relations among GPU instructions to attribute instruction stalls to their root causes and identify the value type of memory instructions. Using performance metrics, program contexts, and instruction characteristics, I developed context-sensitive, instruction stall, and value redundancy analyzers to generate insightful performance reports. The context-sensitive analyzer focuses users’ attention on hotspots with sophisticated program contexts. The instruction stall analyzer matches performance bottlenecks with potential optimizations, estimates speedups for each optimization, and outputs the optimization suggestions with the highest estimated speedups. The value redundancy analyzer identifies GPU operations involving significantly redundant values and constructs a value flow graph to visualize value changes across GPU operations. To demonstrate the effectiveness of our performance tools, I have studied many machine learning and HPC applications. Guided by the insightful performance reports generated by our tools, I have identified performance hotspots and proposed effective optimizations that ameliorate underlying causes for inefficiency.
Acknowledgments

I would like to thank many people who have been consistently helping me during my PhD study. Without them, I couldn’t have published papers and finished this thesis.

The time when I first came to Rice flashed into my eyes. I was not sure what challenges I would be facing in the next few years, but after having a long and inspiring talk with Prof. John Mellor-Crummey, I decided to start my research to push the boundaries of performance tools. It was John’s passionate, kind, and diligent quality that impacted me a lot. Through five years of study with him, I improved my skills in programming, presentation, and collaboration. In this thesis, many ideas come from the discussion between me and Prof. Mellor-Crummey, and the writing was fine-tuned based on his feedback. I would like to thank Prof. Xu Liu who has collaborated with me throughout my PhD study. We together proposed many innovative ideas about value profiling on the GPU. Also, I would like to thank Prof. Joseph Cavallaro and Prof. Alan Cox who agreed to be my committee members and provided helpful feedback.

This work has been supported by members from the HPCToolkit team, I would like to thank the following people in the alphabetical order:

- Laksono Adhianto, who is the primary contributor to HPCToolkit’s GUI. He made HPCToolkit’s trace and profile views fit for displaying GPU performance metrics.
- Jonathon Anderson, who refactored HPCToolkit’s dynamic analyzer to make it scalable for large-scale applications.
- Aaron Cherian, who collaborated with me on profiling OpenCL activities.
- Dejan Grubisic, who collaborated with me on tracing GPU activities.
- Vladimir Indjic, who integrated OMPT into HPCToolkit.
• Mark Krentel, who corrected a vast number of runtime and binary analysis problems.

• Yumeng Liu, who implemented HPCToolkit’s sparse metrics representation.

• Xiaozhu Meng, who has collaborated with me on extending Dyninst to analyze GPU binaries, and discussing novel research ideas.

Additionally, I would like to thank my collaborators at other universities, companies, and national labs.

My collaborators at other universities have been working with me on extending my research in other projects, including Mao Lin and Prof. Pengfei Su at University of California Merced, Yueming Hao at North Carolina State University, and Prof. Jiajia Li at The College of William & Mary,

My collaborators at NVIDIA have solved many issues in their software based on our feedback, including Aurelien Chartier, who leads the Sanitizer API project, Mayank Jain, Sanjiv Satooor, Rafael Campana, Rahul Dhoot, and Barton Fiske, who work on the CUPTI project.

My collaborators at national labs have provided applications to evaluate our tools, including Dr. Xiaoye Li, Dr. Yang Liu, Dr. Helen He, Christopher Daley, and Ronnie Chatterjee at the Lawrence Berkeley National Laboratory, and Dr. Ye Luo at the Argonne National Laboratory.

In addition to research and engineering collaboration, I would like to sincerely thank my family members and my friends. Without them, I am not able to go through the PhD study. To my parents, who brought me up and gave me everything you could; and to my wife Binqian, my pets Lucas and Doudou, who wanted to see this day: I love them dearly and dedicate this thesis to them.

Last, I would like to thank Prof. Seiichi Matsuda and Bridgitt Dickey at the Department of Graduate and Postdoctoral Studies. Without their kind approval, I could not submit my thesis on time.

This thesis was supported by a Ken Kennedy Institute ExxonMobil Graduate Fellowship, a Ken Kennedy Institute Andrew Ladd Fellowship, a Ken Kennedy Institute CS&E Fellowship, an ACM-IEEE-CS George Michael Memorial HPC Fellowship, and
the Exascale Computing Project (17-SC-20-SC) — a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This thesis used computation on a system at Rice university, resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, and resources of the National Energy Research Scientific Computing Center at the Lawrence Berkeley National Laboratory.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations x
List of Tables xv

1 Introduction 1
1.1 GPU Performance Tools ........................................ 4
1.2 A Motivating Example .......................................... 5
1.3 The Framework ................................................ 7
1.4 Contributions .................................................. 10

2 Collecting GPU Performance Metrics 12
2.1 Background and Related Work ................................. 13
  2.1.1 GPU Architectures ........................................ 13
  2.1.2 GPU Performance Metrics ................................. 16
  2.1.3 Hardware Counters ....................................... 17
  2.1.4 Instrumentation ............................................ 18
2.2 Instruction Sampling on GPUs ................................ 20
  2.2.1 Interpreting Instruction Samples ........................ 20
  2.2.2 Deriving Metrics Using Instruction Samples .......... 22
  2.2.3 Optimizations Guided by Instruction Samples ....... 25
2.3 Instrumenting GPU Binaries .................................. 28
  2.3.1 Overview .................................................. 28
  2.3.2 Implementation Details .................................. 31
2.4 Using GPUs to Process Measurement Data .................. 33
  2.4.1 Overview .................................................. 34
2.4.2 Parallel Interval Merge ........................................... 35
2.5 Overhead ................................................................. 39
2.6 Summary ................................................................. 41

3 Attribution of Measurement Data ................................... 43
  3.1 Background and Related Work ...................................... 44
    3.1.1 Call Path Profiling ............................................ 44
    3.1.2 GPU Programming Models ...................................... 47
  3.2 Overview ............................................................... 51
  3.3 Wait-free Communication ............................................ 53
  3.4 Fast Call Path Unwinding ............................................ 58
    3.4.1 Improving Existing Unwinders ................................. 58
    3.4.2 Memoization-based Unwinding ................................. 59
  3.5 Range-based Attribution and Aggregation ....................... 65
    3.5.1 Fixed Range Profiling ......................................... 66
    3.5.2 Adaptive Range Profiling and Sampling .................... 68
    3.5.3 Range Aggregation ............................................. 69
  3.6 Domain-specific Profiling Interface .............................. 73
    3.6.1 OpenMP Tool Interface ....................................... 74
    3.6.2 Deep Learning Profiling Interface ......................... 75
  3.7 Overhead and Accuracy ............................................. 88
    3.7.1 Coarse-grained Profiling .................................... 90
    3.7.2 Fine-grained Profiling ...................................... 93
  3.8 Summary ............................................................... 97

4 GPU Binary Analysis .................................................. 100
  4.1 Background and Related Work ...................................... 100
    4.1.1 Binary Format ................................................ 100
    4.1.2 Instruction Format ................................ .......... 101
  4.2 Program Structure Recovery ...................................... 103
  4.3 GPU Calling Context Trees ....................................... 105
  4.4 GPU Instruction Analysis ......................................... 112
Illustrations

1.1 Training compute (FLOPs) of milestone Machine Learning systems over time. [1] ................................................................. 1
1.2 The GPU utilization rate of 19 deep learning models implemented in PyTorch. [2] ................................................................. 3
1.3 The hotspot of Quicksilver’s CycleTracking kernel with a flat GPU profile view in Nsight Compute. ................................. 5
1.4 The hotspot of Quicksilver’s CycleTracking kernel with reconstruction of approximate GPU calling context. ................. 6
1.5 The conceptual overview of the components in our tools and their products ................................................................. 8

2.1 NVIDIA GA100 architecture. Dashed arrows denote instruction and data paths. ............................................................ 14
2.2 A mental model of PC sampling on an SM of NVIDIA’s V100 GPU. Samples are taken every $N$ cycles. Samples at $N$, $4N$, and $6N$ are latency samples, and others are active samples. Samples at $N$, $3N$, $4N$, $5N$, and $6N$ are stall samples. ....................... 21
2.3 Nekbone optimization steps ......................................................... 26
2.4 Roofline model of Poisson operator with Double precision. *Arithmetic Intensity* denotes Poisson operator’s Double precision operations over global memory traffic .................................................. 28
2.5 A binary instrumentation example .................................................. 29
2.6 Our tool’s instrumentation workflow ................................................. 30
2.7 A pipeline for offloading data processing to the GPU ....................... 35
2.8 An example of merging intervals in parallel. ...................................... 36
2.9 An example of compacting intervals in parallel. 38
2.10 Slowdown with our tool’s hybrid mode and sampling mode. 39
2.11 Our tool’s overhead in collecting memory metrics 40

3.1 A calling context tree and its metrics storage. 45
3.2 GPU Programming Models 48
3.3 Multi-layer contexts imposed by GPU programming models and vendor tool APIs. 50
3.4 Our tool’s infrastructure which coordinates application threads, a monitor thread, and tracing threads for metrics attribution. 51
3.5 Interactions between the GPU monitor thread and application threads. pc0, pc1, and pc2 denote PC samples 57
3.6 The memoization mechanism to query, get, and update CPU call paths. 60
3.7 An example program and the stack of $K_1$ and $K_2$ when they are invoked. We use letters to represent GPU kernels and subscripts to represent different call paths. $\text{rsp}$ denotes the current location in stack. 61
3.8 The GPU operation traces of each operation in Figure 3.7 when $i = 0$. 64
3.9 Serialized profiling 65
3.10 Continuous profiling 65
3.11 Fixed range profiling 66
3.12 Attributing instruction samples to the heterogeneous calling context tree. 67
3.13 Adaptive range profiling 68
3.14 An example of recording GPU kernels using a call path trie. The $R_i : s/n$ annotation on the side of each node indicates $s$ out of $n$ kernels are sampled in $R_i$. The red $R_i$s are the first range of each node. The green boxes enclose kernels that have been profiled; the orange boxes enclose skipped kernels. We track the output range_id after applying Algorithm 3 for each kernel. 72
3.15 The hierarchy of GPU performance tools. 75
3.16 DLPT provides application level information to performance tools. 77
3.17 Major components of DLPT .................................................. 79
3.18 Profile View of the MNIST model using HPCToolkit ................. 85
3.19 Modifications on call paths to yield insightful contexts ............. 86
3.20 A profile view using DLPT .................................................. 87
3.21 The nested call frames for a convolution computation in PyTorch .. 88
3.22 Overhead comparison between tools. Our tool uses unwinding probability $U = 1/4$. .................................................. 90
3.23 Accuracy comparison using different unwinding probability ......... 92
3.24 Adaptive range profiling’s (high frequency) overhead w/ and w/o call path memoization. .................................................. 95
3.25 CPU memory consumption comparison between different profiling methods .................................................. 96

4.1 An example of splitting a call graph into a calling context tree. A-D denote functions. Each subscript denotes the number of samples associated with the function. Each edge is annotated with its number of call samples. .................................................. 109
4.2 Steps to reconstruct GPU calling context trees. Shapes with background are GPU instructions: circles denote call instructions, and rectangles denote other instructions. ................................. 110
4.3 An example of a barrier register dependency. ......................... 112
4.4 An example of blaming stalls on the IADD instruction in the sampling mode and the hybrid mode .................................................. 114
4.5 Memory dependency stall classification ................................. 118
4.6 Execution dependency stall classification ............................... 119
4.7 An example of access kind analysis. The search starts at STG_64 with Backward order until reaching LDS. The search proceeds in Forward order and terminates at DADD. .................................................. 122
4.8 Access kind coverage of different benchmarks ....................... 123

5.1 An unnecessary synchronization calling context in PeleC ............ 131
5.2 Fine-grained measurements of PeleC’s TG benchmark collected with binary instrumentation on an Intel Gen9 GPU. PeleC is implemented using DPC++ and executing atop Intel’s OpenCL runtime. 133

5.3 A bottom up view of memory importance in Laghos 135

5.4 Nyx’ trace view running across 22 nodes using 640 GPU streams and 128 processes. 137

5.5 A mental model for latency hiding optimizers. Green code indicates active samples, and red stalls represent latency samples; latency hiding optimizers consider the effect of moving the code in the dashed box to fill the stall slots. 143

5.6 The highest ranked optimization suggestion in our tool’s performance report for the ExaTensor benchmark and the corresponding hotspot in our tool’s GUI. 148

5.7 Code pattern that matches asynchronous memory copy optimization. 149

5.8 Data structures for detecting temporal value redundancy. The Thread ID column contains a pair \( (blockId, threadId) \) that uniquely identifies each GPU thread. 156

5.9 An example of construction and analysis of a value flow graph. Rectangles are GPU memory allocations, circles are GPU memory APIs, and ovals are GPU kernels. The wider the edge, the more bytes accessed. The red color indicates high redundancy, and the green color indicates low redundancy. To facilitate the presentation, we use the line number as the ID for each vertex. Fined-grained value patterns and calling contexts (not shown) are associated with each vertex. 162
5.10 Following the operations in sequence, the figure shows how memory snapshots are allocated over time. After allocating $a$, $b$, $c$, the analysis thread is woken and tries to analyze kernel 4. It first looks up data objects in snapshot 3, which is the closest to kernel 4. Then it erases all the memory snapshots before 3. Likewise, after 7: $Free(b)$ the analysis thread analyzes kernel 5. After analysis, it erases snapshot 3 but keeps snapshots 5 and 7. ........................................ 166

5.11 Three memory copy strategies. ........................................ 167

5.12 The profile view of our tool shows the calling context of the `fill` kernel invocation with high spatial store value redundancy. The top pane shows the source code; the bottom left pane shows the full calling contexts; and the bottom right pane displays the redundancy count (SPATIAL_WRITE_RED) and the redundancy ratio (SPATIAL_WRITE_RED_RATIO) of each context (e.g., source lines and kernels). ........................................ 168

5.13 A part of the value flow graph for Darknet generated by our tool. When a user hovers the cursor over a vertex, a text box appears to show details such as the vertex’s calling context to help the user locate inefficient code. ........................................ 170

5.14 Two hot spatially redundant stores in LAMMPS due to function `Kokkos::resize`. `Kokkos::resize` is invoked from every call to `LAMMPS_NS::MemoryKokkos::grow_kokkos`. Note: We collapse long function names generated by Kokkos. ........................................ 174
Tables

2.1 Several important GPU performance metrics .......................... 16

3.1 Call path statistics of applications ................................. 91

3.2 Comparisons of fine-grained profiling overhead between our tool
against HPCToolkit. HPCToolkit’s overhead is used as the baseline
to compare with our methods. ........................................ 93

3.3 Our tool’s fine-grained profiling accuracy. We do not compare
HPCToolkit’s accuracy because it uses the serialized mode to collect
instruction samples, yielding an exact correlation between instruction
samples and call paths. We use bold fonts to denote the highest
accuracy in each application. ........................................ 94

4.1 Dissection of the fields of “@P0 LDG.32 R0, [R2]” instruction. .... 102

5.1 A description of GPU optimizers in our tool. ......................... 140

5.2 Achieved speedups averaged among five runs. We improved each code
according to the suggestion provided by GPA. Estimate error is
computed by \( \frac{|\text{Estimated Speedup} - \text{Achieved Speedup}|}{\text{Achieved Speedup}} \times 100\% \). .......... 146

5.3 Various value patterns exist in GPU applications and benchmarks. . 158

5.4 Speedups of LAMMPS on varying numbers of GPUs. ............... 173
Chapter 1

Introduction

The computing landscape is undergoing rapid evolution to meet the demand in data-intensive applications and grand challenging scientific problems. Figure 1.1 illustrates the growing demand for compute power to train large-scale deep learning models. Starting from early 2010s, the training compute demand doubles about every six months; since late 2015, many large-scale models [3, 4] have been proposed with even 100 times larger requirements.

Figure 1.1: Training compute (FLOPs) of milestone Machine Learning systems over time. [1]

To address the challenges, the focus of chip design has shifted from increasing
transistor density to tailoring micro-architectures for applications that require large amounts of computational power. Consequently, heterogeneous architectures have revolutionized in the past decade, bringing opportunities and challenges for algorithms, computation models, programming models, and tools. Graphics Processing Units (GPUs) have seen the widest adoption among the emerging architectures and are key components of accelerated computing platforms. Today, two out of five top supercomputers employ GPUs to deliver superior performance; data centers use GPUs to speed up artificial intelligence workloads; autonomous vehicles rely on GPUs to balance performance and energy consumption. Moreover, forthcoming exascale systems being developed by the US Department of Energy (DOE) will all employ GPU-accelerated compute nodes.

While GPUs deliver massive parallelism and high bandwidth memory, writing fast and efficient GPU-accelerated code is challenging because of the sophisticated programming models and architectural features. Without careful design, GPU-accelerated applications may underutilize GPU resources due to many issues, such as poor data locality, insufficient parallelism, or unused compute units. As an example, Figure 1.2 measures the GPU utilization rate of 19 popular deep neural networks. Although these models are accelerated by adopting high performance compute routines (e.g., cuBLAS [5]), half of the models achieved less than 50% utilization. Moreover, the ShuffleNet model [6] only utilizes 20% GPU compute power.

To start optimizing GPU code, one would identify performance problems and associate the problems with corresponding optimization strategies. Pinpointing performance problems in GPU code can be difficult as it often requires a detailed analysis that combines measurements of a program’s execution, information passed to a GPU code in the calling context where it was invoked, characteristics of compiler-generated
Figure 1.2: The GPU utilization rate of 19 deep learning models implemented in PyTorch. [2]

GPU code, and GPU hardware features. While the use of higher-level programming models such as PyTorch [7], Tensorflow [8], Triton [9], TVM [10], RAJA [11], Kokkos [12], and OpenMP [13] can simplify the development of accelerated machine learning and high performance computing applications, they can increase the difficulty of tuning kernels for high performance by separating developers from many key details, such as what GPU code is generated and how it will be executed. To address the challenges, we propose innovative tool designs to measure, analyze, and optimize performance of GPU-accelerated applications. In the rest of this chapter, Section 1.1 introduces GPU performance tools and their shortcomings. Section 1.2 provides a motivating example that illustrates problems in existing tools. Section 1.3 outlines the framework of our tools. Section 1.4 describes contributions in this thesis.
1.1 GPU Performance Tools

GPU performance tools are used to bridge the performance gap between applications and the underlying architecture by providing measurement, analysis, and tuning functionalities. Two kinds of reports are typically provided by tools to assess performance and assist performance tuning. A trace view shows a series of GPU operations that happen over time on each process, thread, and GPU stream. A profile view collapses out the time dimension and aggregates performance metrics of GPU operations together.

Unfortunately, existing GPU performance tools, including nvprof [14], Nsight Systems [15], Nsight Compute [16], Score-P [17], Extrae/Paraver [18], TAU [19], VTune [20], ROCTracer [21], and ROCProfiler [22], fall short of providing performance insights to pinpoint inefficient code and provide optimization guidance in their trace and profile views. Foremost, many existing tools lack correlation between GPU operations and program contexts where they are initiated on the CPU, making it hard to locate hotspots in complex applications. Second, existing tools lack comprehensive contexts within GPU kernels (i.e., code executed on the GPU), which is important to understand the execution of sophisticated GPU code. Further, none of the existing tools analyze the root cause of inefficiencies of both coarse-grained GPU operations and fine-grained GPU instructions. Without automated root cause analysis, users take significant manual effort to inspect code and investigate effective optimizations. Last, many tools can cause significant runtime overhead. As a result, they may generate inaccurate performance reports by distorting the original execution.
1.2 A Motivating Example

Figure 1.3 shows a profiling result for the Quicksilver [23] proxy application from Lawrence Livermore National Laboratory using Nsight Compute [16]. Quicksilver has a single GPU kernel with thousands of lines that invokes many GPU device functions. Nsight Compute only associates instruction samples with individual lines to help users locate hot code but provides little performance insight regarding where inefficiencies come from and what optimizations can be applied to improve the performance. In Figure 1.3, we can observe that function `getReactionCrossSection` is time-consuming but; however, to understand the performance issues, we need a more comprehensive calling context to show where function `getReactionCrossSection` is invoked. More than that, Nsight Compute does not suggest possible performance optimizations for this hotspot. Without the tool’s help, one may have to analyze the assembly code to understand performance issues or try tens of candidate optimizations to find the best one without any guidance.

Compared with Nsight Compute, our performance tool provides functionalities to locate performance inefficiencies in deep call paths, automatically analyze the causes of inefficiencies, and suggest effective optimizations. As shown in Figure 1.4, our tools
provide a heterogeneous calling context view consisting of frames on both the CPU and the GPU. Using this view, we can locate where the `CycleTrackingKernel` is launched on the CPU and identify that the hot function `getReactionCrossSection` is called from Line 73 in function `macroscopicSection` enclosed in a loop on the GPU. Further, our value redundancy analysis analyzes the values loaded and stored by the instructions at the epilogue and prologue of function `macroscopicSection` and indicates that these values are temporally redundant. Finally, our stall analysis module suggests that applying the function inlining optimization could improve the performance of this kernel by 1.10×. With the help of the above anal-
yses, we derived that the major inefficiency of the `CycleTrackingKernel` function is caused by repeatedly storing and loading loop invariant values in a device function `getReactionCrossSection`. Inlining this function to its call site in Function `macroscopicSection` achieves a $1.13 \times$ speedup, which is close to the $1.10 \times$ speedup as our advisor estimates.

### 1.3 The Framework

Our goal is to design GPU performance tools that provide actionable optimization insights for performance tuning. We aim to have tools automate the measurement, analysis, and optimization (if possible) of GPU-accelerated applications. By studying many GPU-accelerated programs, we found that the two most common causes of inefficiencies are (1) GPUs are idle, and (2) GPU resources are not efficiently utilized. The first kind of inefficiency can be identified through low overhead and scalable tracing and profiling of GPU activities. The second kind of inefficiency requires a thorough understanding of what causes inefficiencies, if the inefficient operations are necessary, and where the inefficiencies occur. Unfortunately, existing tools neither incur low overhead to identify the first problem nor provide enough insights for the second problem.

Figure 1.5 shows important substrates in our tools as well as the workflow among them. To measure performance metrics on the GPU, we developed a low overhead profiler. In the profiler, we implemented a data collector to measure both coarse- and fine-grained metrics. The data collector is tuned for complex GPU-accelerated applications using innovative sampling and GPU-accelerated processing mechanisms. At runtime, the profiler attributes metrics collected to the CPU calling contexts where corresponding GPU operations are initialized using a low overhead call path unwinder.
Beyond associating metrics with general program contexts, we also designed a *domain specific profiling interface* for deep learning frameworks to refine metrics and program context association. We describe the design and the implementation of the profiler in Chapter 2 and Chapter 3.

The profiler records GPU binaries loaded at runtime. To enrich program contexts, we implemented a *static analyzer* to recover static program context information about lines, loops, and inlined functions. In addition, the static analyzer parses
GPU binaries to derive instruction dependencies, memory instruction access types, and architectural features. Chapter 4 describes the details of static analysis methods.

The static information and profiles are ingested into a dynamic analyzer that consists of a collection of analysis modules. The output of analysis modules are insightful performance reports that guide optimizations. The context sensitive analysis module pinpoints performance hotspots at levels of lines, loops, and functions. It generates a profile view that associates performance metrics at levels of lines, loops, and functions, and a trace view that presents GPU and CPU activities in each GPU stream and CPU thread across the timeline accordingly. The value redundancy analysis module analyzes redundant values read or write by GPU operations. A unique value flow graph is generated to visualize value changes of GPU data objects (e.g., arrays) across all GPU operations invoked during the execution. The instruction stall analysis module matches instruction stalls with optimization strategies and estimates each strategy’s potential speedup. The most effective optimizations ranked by their estimated speedups are output to a performance advice report.

To demonstrate the effectiveness of our tools, we studied a wide range of GPU-accelerated applications, including deep learning applications such as PyTorch [7] and Darknet [24], HPC applications such as LAMMPS [25], BerkeleyGW [26], Castro [27], and Nekbone [28], as well as well-known GPU benchmarks such as Rodinia [29], Quicksilver [23], and Laghos [30]. Our tools can quickly pinpoint the performance issues in these applications and provide insights for improving performance. Guided by our tools’ performance reports, we optimized the code and verified their effects. Further, we confirmed the performance issues with application developers and upstreamed commits to their repositories. We describe the dynamic analyzer and optimization insights provided by our tools in Chapter 5.
1.4 Contributions

We state the following thesis:

Guided by redundancy and stall analyses on performance metrics collected using low overhead measurement and context-sensitive attribution, we can pinpoint, quantify, and eliminate inefficiencies in a wide range of GPU-accelerated applications.

This dissertation describes four major contributions to the state of the art:

- novel sampling, instrumentation, metrics derivation, and GPU-accelerated data processing approaches [31, 32, 33, 34] for collecting GPU performance metrics with low overhead,

- a calling context sensitive attribution mechanism [31, 35] that constructs both CPU and GPU calling contexts with low overhead and high accuracy using an innovative call path memoization method, an adaptive range profiling method, and an offline GPU calling context reconstruction algorithm,

- an advisor tool [36, 37] that relieves users of the burden of interpreting performance counters and analyzing bottlenecks by employing data flow analysis to attribute instruction stalls to their root causes and using information about program structure and GPU architectural features to match inefficiency patterns with optimization suggestions, and

- a redundancy analysis tool [38, 34] for calculating value redundancy metrics and categorizing microscopic value patterns in GPU kernels and global value flows
across GPU operations to identify value-related redundancy issues in GPU-accelerated applications.
Chapter 2

Collecting GPU Performance Metrics

Performance tools collect and analyze performance metrics to provide insights for optimizations. Measuring GPU-accelerated applications imposes unique challenges in coordinating threads and processing measurement data. GPU-accelerated applications typically employ tens of thousands of threads for acceleration. Also, due to limited performance counters on GPUs, not all metrics cannot be collected in a single pass; multiple passes have to be applied to collect all necessary metrics.

This chapter describes a novel data collector, which measures GPU performance metrics using hardware counters and instrumentation. This chapter introduces essential GPU performance metrics and typical methods for collecting them. Because existing GPU profilers can cause prohibitive overhead in collecting metrics due to their problematic design, this chapter presents novel techniques to address the problems. First, to reduce the amount of measurement data, we developed efficient instrumentation routines to compact measurement data and a hierarchical sampling method to selectively monitor threads on the GPU. Further, instead of processing measurement on the CPU, we implemented GPU kernels that accelerate the process by utilizing high parallelism on the GPU. Finally, we designed a method to derive many essential metrics from the measurement in a single run.
2.1 Background and Related Work

This section describes the essential background to understand GPU performance metrics and methods for collecting metrics. We first present a brief introduction to general GPU architectures in Section 2.1.1. Next, we describe representative GPU performance metrics in Section 2.1.2. Finally, we describe two methods for collecting performance metrics in Section 2.1.3 and Section 2.1.4, correspondingly.

2.1.1 GPU Architectures

This section reviews the architectures of NVIDIA, Intel, and AMD GPUs used in data centers and supercomputers. Hardware components for graphic rendering [39] are not covered in this thesis. Since GPUs from the three vendors share many common features, we mainly use the terminology of NVIDIA GPUs and briefly describe the differences between NVIDIA GPUs, Intel GPUs, and AMD GPUs. We illustrate the typical composition of GPU architectures in Figure 2.1 using the modern NVIDIA GA100 architecture.

A GPU consists of a set of processor clusters. In modern GPUs, such as NVIDIA’s Ampere series, the processor clusters can be composed of GPU Processor Clusters (GPCs) and Texture Processor Clusters (TPCs). For instance, a GA100 GPU contains 8 GPCs, where each GPC has 8 TPCs, and each TPC has 2 Streaming Multiprocessors (SMs). The L2 cache on a GPU is shared among all SMs. Each SM accommodates an L1 instruction cache and an L1 data cache, shared among multiple warp schedulers. Part of the L1 data cache can be configured as the shared memory. Unlike traditional caches, users can explicitly allocate, access data, and evict data on the shared memory. Recent GPUs have incorporated an L0 instruction cache, which is private to each warp scheduler.
A warp is the minimum scheduling unit on the GPU, representing a group of 16, 32, or 64 threads, depending on vendor-specific configurations. Warp schedulers are responsible for issuing instructions to corresponding function units. Each scheduler accesses its own set of register files and function units, including floating point units (FPUs), integer units (INTUs) inside, double precision units (DPUs), special function units (SFUs), tensor cores units (TCUs), and load store units (LDSTs). In previous generation GPUs, warp schedulers support the standard single instruction multiple threads (SIMT) model, in which threads that belong to the same warp execute the same instruction in each lockstep. Recent generations of GPUs adopt independent thread scheduling [40] for threads within a warp to enable finer-grain synchronization and cooperation. A group of warps are organized into a block. Multiple blocks can
be scheduled concurrently by a single SM, but a block cannot be scheduled across multiple SMs.

AMD GPUs share many similar features as NVIDIA GPUs. In the following, we briefly compare the GA100 architecture with AMD’s recent CDNA2 GPU architecture [41] as an example. The minimum scheduling unit on AMD GPUs is a Compute Unit (CU), which is similar to the concept of SM on NVIDIA GPUs. Similar to NVIDIA GPUs, all CUs on an AMD GPU share an L2 cache. Each CU is equipped with four matrix cores and four SIMD cores, which are analogous to TCUs and FPUs on NVIDIA GPUs. Unlike NVIDIA GPUs, AMD GPUs use a warp size of 64 and do not support independent thread scheduling [42].

Previously, Intel only provided integrated GPUs whose memory and L3 cache are shared with the CPUs. In Intel’s discrete $X^e$ GPUs [43], function units are organized into $X^e$ cores. Compared to SMs on NVIDIA GPUs or CUs on AMD GPUs, Intel GPUs have fewer $X^e$ cores but each $X^e$ core consists of more function units (i.e., Vector Engine) than an SM or a CU has.

Due to the complexity of GPU architectures, many factors have to be taken into account while designing GPU kernels to achieve maximum performance. At a high level, a kernel should use as many SMs as possible on the GPU. Within each SM, programmers may also encounter performance efficiency problems in memory access patterns considering the complex memory hierarchy, overlapping instructions on different function units, and making the warp scheduler as busy as possible. To measure if a GPU kernel has used the GPU well, we can rely on essential performance metrics described in the next section.
2.1.2 GPU Performance Metrics

<table>
<thead>
<tr>
<th>Metrics Class</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallelism</strong></td>
<td>SM Efficiency, Theoretical Occupancy,</td>
</tr>
<tr>
<td></td>
<td>Achieved Occupancy</td>
</tr>
<tr>
<td><strong>Compute Workload</strong></td>
<td>IPC, Warp Issue Rate,</td>
</tr>
<tr>
<td></td>
<td>Warp Eligible Rate, SM Busy Rate</td>
</tr>
<tr>
<td><strong>Source Counters</strong></td>
<td>PC Samples (with different stall reasons),</td>
</tr>
<tr>
<td></td>
<td>Instruction Counts</td>
</tr>
<tr>
<td><strong>Launch Statistics</strong></td>
<td>Grid Dimension, Block Dimension,</td>
</tr>
<tr>
<td></td>
<td>Shared Memory Usage, Register Usage</td>
</tr>
<tr>
<td><strong>Instruction Statistics</strong></td>
<td>Instruction Throughput, Memory Throughput,</td>
</tr>
<tr>
<td></td>
<td>Cache Misses</td>
</tr>
<tr>
<td><strong>Device Statistics</strong></td>
<td>Elapsed Cycles, Utilization Rate,</td>
</tr>
<tr>
<td></td>
<td>SM Frequency, Memory Frequency</td>
</tr>
</tbody>
</table>

GPU profilers [20, 22, 14, 15, 16] measure both coarse- and fine-grained performance metrics. Coarse-grained metrics, such as the number of invocations, the execution time, and the number of bytes, are often attributed to individual operations. Previous work [44, 45] intercepts GPU APIs or uses vendor provided measurement APIs [46, 47, 21] to measure coarse-grained metrics. Our tool collects coarse-grained metrics on NVIDIA GPU using NVIDIA’s CUPTI Activity API [46]. We focus on monitoring GPU memory copies, memory sets, memory allocation/free, and custom GPU kernels.

In contrast, fine-grained performance metrics, such as memory accesses, cache misses, and instruction stalls, provide sophisticated information about instructions executed and addresses touched, but collecting them involves heavy instrumenta-
tion [48, 49] or frequent readings of performance counters.

Table 2.1 lists representative coarse- and fine-grained performance metrics of different classes. Coarse-grained metrics, such as metrics in the Launch Statistics class, are obtained through CUPTI. Some fine-grained metrics, such as SM frequency and cache misses, can only be acquired by reading hardware counters; some others, such as diverged instructions, can only be acquired using instrumentation. Our tool can collect a variety of fine-grained GPU performance metrics using both instrumentation and hardware counters. We elaborate on the background for hardware counters and instrumentation in Section 2.1.3 and Section 2.1.4 correspondingly.

2.1.3 Hardware Counters

Hardware counters are special-purpose registers in processors to store the values of hardware-related events. Using hardware counters, we can collect many kinds of performance information related to function units, caches, and memory with low overhead and no source code modification. However, the number of metrics collected in a single pass is limited by the number of hardware counters in a processor. To overcome this limitation, Nsight Compute organizes metrics into multiple groups, where each group contains a number of metrics that can be collected in a single run. Then, it replays application execution multiple times to get all requested performance metrics. This method could introduce prohibitive overhead in our experiments, especially for complex applications that launch millions of GPU kernels per second. Traditional CPU profilers, such as VTune, can run an application only once and use multiplexing [50] to monitor metrics in a round-robin fashion. Multiplexing can increase the kind of metrics collected in a single run and reduce application replay overhead. By now, performance counter multiplexing is not available on NVIDIA and AMD GPUs.
Using hardware counters, we can also monitor information about each instruction in addition to hardware events. Instruction sampling [51, 52, 53] is a useful technique for fine-grained performance measurement and analysis on modern processors. It can identify performance issues and associate issues with program sources by linking hardware events with the instructions that caused them.

In 2015, NVIDIA added support for instruction-based sampling to their GPUs [54], which they call PC sampling. Each instruction sample has a stall reason associated with it if the sampled instruction is stalled. At the time of this writing, hardware support for instruction-level measurement of GPU performance is only available on NVIDIA GPUs. This will change. Intel is developing hardware support for instruction-level performance measurement that will become available in a future generation of its GPUs.* In addition, recent commits in AMD’s ROCM GitHub repository†‡ indicate emerging support for PC sampling in their GPU software stack.

2.1.4 Instrumentation

Instrumentation refers to methods that augment an application with additional instructions to diagnose errors, trace execution, or measure performance. There are two primary instrumentation approaches: source code instrumentation and binary instrumentation.

For GPU-accelerated applications, source code instrumentation tools [55, 56, 57] are used to identify CPU code to annotate important regions so that GPU profilers only measure GPU operations in the areas annotated. Using this method, we can

---

*Intel has approved public release of this information.
†https://github.com/ROCm-Developer-Tools/roctracer/blob/amd-master/inc/ext/prof_protocol.h#L75
‡https://github.com/ROCm-Developer-Tools/roctracer/blob/amd-master/test/tool/tracer_tool.cpp#L655
avoid collecting performance metrics other GPU operations to reduce the profiling
overhead. However, there are two major shortcomings of these source code instru-
m entation tools. First, they often cause significant code modification for complex
applications if many functions have to be monitored. Second, instrumentation can
bring large overhead if instrumented functions are invoked frequently. Our tool fo-
cuses on measuring whole program execution without any source code modification
but is compatible with source code instrumentation tools. To the best of our knowl-
edge, there is no source code instrumentation tool available to annotate code within
GPU kernels.

Unlike source code instrumentation, binary instrumentation is done without source
code modification and recompilation. Binary instrumentation tools inject user-defined
code at specific instructions to monitor runtime characteristics. On CPUs, there ex-
ist static instrumentation tools [58, 59], dynamic instrumentation tools [60, 61], and
hybrid tools [62]. Static tools instrument a binary before it is loaded into memory,
while dynamic tools take control of the program after it loads into the memory. For
dynamic instrumentation, a tool first accesses an application’s instruction stream by
using hooks provided by the host operating system to modify the memory space [63].
Then, the tool injects analysis routines into the application. During execution, appli-
cations switch between instrumented code and the original code. Since only executed
code needs to be instrumented, dynamic instrumentation’s code size and overhead
are usually smaller than that of static instrumentation.

There also exist several instrumentation facilities [48, 64, 49] for GPU binaries.
Because instructions are offloaded to the GPU memory for execution, while instru-
mentation happens on the CPU, it is challenging to modify the GPU instruction
stream to apply dynamic instrumentation. For this reason, all existing GPU binary
instrumentation facilities are static tools that exhaustively instrument all functions within a GPU binary. Our tool employs NVIDIA’s Sanitizer API [64] and Intel’s GT-Pin [65]. GPU binaries are JIT compiled in many applications, so we instrument GPU binaries at runtime and dump them to disk for postmortem analysis.

GPU vendors have developed instrumentation tools [48, 66, 49, 64] for fine-grained performance measurement and analysis. These tools, however, introduce unavoidable overhead for GPU kernels. There have also been efforts that use instrumentation methods to diagnose specific types of inefficiencies. Yeh et al. [67] instrument GPU code as it is generated by LLVM to identify redundant instructions. CUDAAdvisor [68] also instruments code as it is generated by LLVM to monitor GPU memory access and decide if bypassing could be used. These tools only identify particular inefficiencies and do not correlate causes with expensive code.

2.2 Instruction Sampling on GPUs

This section first describes the instruction sampling mechanism on NVIDIA GPUs. Then, we present a metrics derivation method that derives metrics in Table 2.1 using instruction samples. In the end, we show how PC samples and derived metrics can be used to guide performance optimization with a concrete example.

2.2.1 Interpreting Instruction Samples

NVIDIA GPUs implement PC sampling to collect instruction samples. One can use NVIDIA’s CUPTI API [46] to collect PC samples for GPU-accelerated applications. Each SM in an NVIDIA GPU collects samples individually. When a PC sampling buffer used to collect samples is full on an SM, CUPTI merges samples from all SMs and transfers the samples to the CPU. Each SM on an NVIDIA GPU has four warp
Consider Figure 2.2 as an example. Because there are three latency samples and three active samples, we estimate both the stall ratio and the active ratio of the SM as 3/6. Assuming all SMs on the GPU have a similar workload, we estimate both the stall ratio and the active ratio of the GPU kernel as 3/6. In our example, there are five samples with a stall reason. We call such samples *stall samples* or *stalls*. In the PC sampling buffer yielded by CUPTI, we can obtain each instruction sample’s program counter (PC), the reason why the sample is stalled, and the number of samples. Using stall reasons, we can attribute the delay of instruction samples to certain function
units or memory hierarchies. CUPTI’s documentation [54] lists all the available stall reasons. It is noteworthy that PC sampling imposes negligible overhead to monitoring GPU kernels.

2.2.2 Deriving Metrics Using Instruction Samples

By attributing instruction samples to GPU kernels, we can identify hot code regions within each kernel. To identify opportunities for tuning, we need some additional metrics, as shown in Table 2.1. Due to the limited number of performance monitoring units, CUPTI cannot collect instruction samples in the same pass as other fine-grained metrics. Instead, Nsight Compute runs nine passes to collect all of these metrics for a GPU kernel; this approach is costly for a large-scale execution that performs millions of kernel launches. Our tool collects or estimates essential metrics. We propose a method to approximate instruction throughput and related metrics based on measurements gathered using PC sampling in a single profiling pass.

In Table 2.1, Launch Statistics metrics can be obtained by analyzing the arguments passed to each GPU kernel, and we can get Device Statistics by querying device properties periodically using CUPTI. Then, all metrics in the Parallelism class, except for Achieved Occupancy, can be estimated using Launch Statistics and Device Statistics [69]. In short, the Theoretical Occupancy, which represents the number of active blocks per SM, is limited by shared memory, block size, number of registers per block. Then, we can use the total number of blocks, the total number of SMs, and the Theoretical Occupancy to estimate SM efficiency.

Metrics in the Source Counters class come from CUPTI directly. Based on the number of total samples ($S$) and latency samples ($S_L$), we can estimate metrics in the Compute Workload class. The Warp Issue Rate ($W_R$) of schedulers is estimated
using Equation 2.1. *IPC* (Equation 2.2) is Warp Issue Rate times the number of threads per warp (*T*), and times the number of used warp schedulers (*W*), which is the minimum of the number of warp schedulers and active warps. The total number of eligible instructions observed is the sum of issued samples (*S_I*) and not selected samples (*S_N*). So Warp Eligible Rate (*E*) can be calculated by Equation 2.3. If any of the warp schedulers in an SM is busy, the SM is considered as busy. Therefore SM Busy Rate (*B*) is estimated by Equation 2.4.

\[
\text{WR} = \frac{S - S_L}{S}
\]  

(2.1)

\[
\text{IPC} = \text{WR} \times T \times W
\]  

(2.2)

\[
E = \frac{S_I + S_N}{S}
\]  

(2.3)

\[
B = 1 - (1 - E)^W
\]  

(2.4)

Finally, we estimate metrics in the *Instruction Statistics* class. We collect GPU SM Frequency continuously over time and compute the average frequency \( \bar{C} \) in cycles per second. Suppose a kernel runs for *T* seconds on a GPU. With PC sampling period *P*, we can estimate the total number of samples if the GPU is active all the time (*E_S*) as shown in Equation 2.5. A GPU’s active rate \( \mathcal{A} \)—the fraction of cycles when a GPU is actively processing data versus the total elapsed cycles can be estimated by Equation 2.6. With \( \mathcal{A} \) and the estimated used number of SMs (*N_SM*), we estimate the Issued Instruction Counts of a kernel (*I*) with Equation 2.7.
\[ E_S = \frac{C}{P \times T} \]  \hspace{1cm} (2.5)

\[ A = \frac{S}{E_S} \]  \hspace{1cm} (2.6)

\[ I = A \times T \times \bar{C} \times N_{SM} \]  \hspace{1cm} (2.7)

Using \( I \), we can derive throughput metrics for different types of instructions. We map instructions to different classes by splitting opcodes and modifiers to identify operation types, instruction length, tensor operation, and memory hierarchy. For example, a \texttt{LDG.64} instruction is categorized as “memory.global.load.64”. Then, we count the number of a specific type of issued samples within a kernel as \( S_{It} \). In Equation 2.8, we estimate the throughput of an instruction type by multiplying \( I \) with the ratio of the instruction type among all the issued samples.

\[ I_t = I \times \frac{S_{It}}{S_I} \]  \hspace{1cm} (2.8)

Our estimates may differ from actual values for various reasons, with the most critical factor being predicated instructions. Since PC sampling collects issued instructions instead of executed instructions, our estimated instruction throughput might be higher than the actual throughput if predicates are often false. We can fix the error with another instrumentation pass to get the exact number of executed instructions.
// G, DT: global memory arrays
// wr, us, ut, ul: shared memory arrays
for (int it = 0; it < tileSize; it += blockDim.x) {
    transform(it, i, j, k);
    for (int n = 0; n < N; ++n) {
        wr = wr + DT[i][n] * ul[j][k][n];
        ws = ws + DT[j][n] * ul[j][n][i];
        wt = wt + DT[k][n] * ul[n][k][i];
    }
    double *g = &G[blockId.x][i][j][k];
    ur[i][j][k] = g[0] * wr + g[1] * ws + g[2] * wt;
    us[i][j][k] = g[1] * wr + g[3] * ws + g[4] * wt;
}

Listing 1: A hotspot in Nekbone’s Poisson operator

2.2.3 Optimizations Guided by Instruction Samples

To demonstrate the use of instruction samples, we studied Nekbone [28], a lightweight subset of Nek5000 that mimics the computational characteristics of Nek5000, a high-order Navier-Stokes solver based on the spectral element method. The Poisson operator is a core component of Nekbone and consists of two steps—first calculate partitioned results on each node and then use an MPI exchange to collect the results. We focus profiling and analysis of the calculation. We describe the most important loop in Listing 1. Using instruction sampling on an NVIDIA V100 GPU, we identified a sequence of optimization opportunities, as shown in Figure 2.3.

We first located a hotspot at Loop Lines 5–9, which takes 32% execution time, of which 43% of the latencies are memory throttling. This phenomenon indicates that too many global memory requests occur from accesses to DT, and there is no guarantee that all accesses will be satisfied by the L1 cache. Since DT is small, to reduce the number of global memory requests, we loaded DT into shared memory.
This \textit{+shared} optimization improved performance by 12%.

After \textit{+shared} optimization, we found Line 11 is problematic. It takes 14% execution time, of which 93% latencies are caused by memory dependencies. By observing that Line 12 and Line 13, which also read global memory, do not cause many memory dependency latencies, we deduced that global memory load instructions (\texttt{LDGs}) are not reordered properly to hide latencies. We confirmed our hypothesis by checking the assembly code, finding that \texttt{LDG} instructions in Line 11 are not reordered before Line 5. We manually reordered instructions by reading \texttt{g[0]-g[6]} before Line 5. This \textit{+reorder} optimization improved performance by 15%.

Next, Loop Lines 5–9 became a hotspot again with 33% execution time. 98% latencies of the hotspot are from execution dependencies and pipeline busyness. For the busy pipe problem, templatization does not help because the index calculation at Lines 6-8 leads to high integer instruction throughput. For the execution dependency problem, we noticed that Line 4 involves integer division operations. Our tool showed that the hotspot’s \texttt{MUFU} instruction usage is high, which indicates the use of Special Function Units (SFUs) that cause low throughput and long latency. A division be-
between two integers involves complicated steps on GPUs: (1) the dividend and divisor are converted to floating numbers using SFUs, (2) the reciprocal of the divisor is calculated using an SFU, (3) the dividend is multiplied with the reciprocal, (4) the result is converted to an integer using an SFU. To alleviate the problem, we pre-calculated $1/shape\_output[j]$ and passed them as an array to the kernel. In this way, step 1 only converts an integer, and step 2 is eliminated. This $+reciprocal$ optimization improved performance by 3%.

Figure 2.3 shows our tool’s estimated GFLOPS (PC SAMPLING) for all code versions. Except for baseline, our estimated results are within 6% of the hand calculation. For baseline, neither our tool nor Nsight Compute reported GFLOPS accurately. We found that counting issued instructions is misleading for this code because some predicated instructions were not executed; PC samples do not provide predicate information. Nsight Compute by default measures executed instructions at warp level and therefore also has an error in baseline measurement.

Using measurement data from our profiler, we used the roofline model [70] to analyze the performance upper bound. We plotted the global memory roofline of $+reciprocal$ in Figure 2.4. Our tool’s estimated arithmetic intensity is 8% higher than the actual arithmetic intensity. The theoretical peak performance of this kernel is 1980 GFLOPS, and we achieved 1663 GFLOPS, which is 84% of the peak. As the instruction mix implies, the gap between our implementation and the peak performance is caused by unfused multiply instructions and add instructions. If all multiply instructions and add instructions are fused, it could improve performance by 19%, yielding 99% of peak bandwidth-limited performance.
Theoretical Performance (1980 GFLOPS)
Achieved Performance (1663 GFLOPS)
Arithmetic Intensity = 2.2

Figure 2.4 : Roofline model of Poisson operator with Double precision. *Arithmetic Intensity* denotes Poisson operator’s Double precision operations over global memory traffic

### 2.3 Instrumenting GPU Binaries

This section presents our tool’s binary instrumentation substrate. Section 2.1.4 describes the instrumentation workflow, and Section 2.3.2 elaborates on the implementation of individual components.

#### 2.3.1 Overview

Figure 2.5 presents the mechanism for instrumenting GPU binaries. Our tool uses GT-Pin and Sanitizer API to perform instrumentation; both are static tools that exhaustively instrument every GPU function. Before instrumentation, users can specify what kind of instructions or instruction locations are of interest. For example, we only monitor the `LDG` instruction in Figure 2.5. At each candidate instruction, the instrumentation facility inserts *prolog*, *call*, and *epilog* code sections. The prolog is used to save live registers to memory; the epilog is used to restore live registers from
memory. The call instruction transfers the control flow to a user-defined callback that collects addresses and values accessed by the LDG instruction.

For Intel GPU programs, GT-Pin provides high level APIs, as it requires users to claim a fixed size buffer to store GPU metrics before the launch of each GPU kernel. In this way, GT-Pin limits the number of metrics to collect for GPU kernels. Users who are not experts in GPU programming languages can benefit from the ease of GT-Pin’s high level APIs. However, on the other hand, expert users do not have access to low level details, losing the chance to use parallel primitives and threads to optimize instrumentation routines. Our tool [33] uses GT-Pin to count the number of executed instructions, the number of SIMD lanes used by each instruction, and the latency of each basic block.

Likewise, CUPTI provides interfaces to instrument GPU binaries to collect performance metrics that cannot be measured using instruction sampling, such as predicated instructions, executed instructions, and memory accesses efficiencies. In contrast, other NVIDIA’s tools, including Sanitizer API [64], SASSI [66], and NVBit [49] provide more flexibility for users by letting users write CUDA code directly in callbacks.
Using Sanitizer and NVBIT, users have full control of the metrics buffer. By repeatedly copying data from the GPU to the CPU, all measurement data can be collected and analyzed at runtime. Users can also leverage the standard CUDA semantics to employ warp level primitives to accelerate instrumentation code in callbacks.

Figure 2.6 demonstrates the metrics collection and analysis workflow of our tool on NVIDIA GPUs. The *GPU callbacks* collect GPU measurement data during execution. The *Runtime* on the host CPU communicates GPU measurement data to the CPU, manages the collected measurement data, and gleans extra information for analysis. The *Analyzer* spawns a helper thread to perform analysis (i.e., identifying redundancies [38]) on the GPU measurement data and passes analysis results to the runtime. The analyzer communicates with the runtime through a lock-free queue. The runtime communicates with the GPU callbacks through a blocking queue on the GPU. Our tool’s architecture is modular, and one can easily extend it for a wealth of GPU program analyses.
2.3.2 Implementation Details

**GPU Callbacks** Our tool instruments callback functions at each kernel’s entrance and exit, as well as after each memory access instruction. At kernel entrance and exit, we record every thread’s id. In memory access callbacks, we record information associated with a memory access, such as program counter (PC), value, effective address, and access size. If a memory access instruction is a store, our tool directly obtains the value from the register. Otherwise, our tool obtains the effective address accessed by the instruction and then reads the value stored in the memory by dereferencing the effective address according to the size and type (i.e., global, local, or shared) of the instruction. Typically, references to local and shared memory use 32-bit addresses, while references to global memory use 64-bit addresses.

**GPU queue** Our tool creates a queue shared between a GPU and a CPU to hold and transfer records collected with the callbacks. To distinguish from other queues, we call it the GPU queue. The size of the GPU queue is configurable by users. By default, we set the queue with 72 MB, which is a good tradeoff between runtime and memory overhead from our experiments.

Operations on the GPU queue are atomic. To minimize contention, our tool only allows the first active lane in a warp to request an empty slot in the queue. Our tool then uses a warp shuffle operation to broadcast the slot location to every active lane in the warp. All active threads write their records to the slot concurrently. All lanes are synchronized when the writing is done. Our tool uses the first active lane to inform the GPU queue that a slot is filled. In addition to memory records, the GPU
queue also holds records about each thread’s entrance and exit.

To simplify CPU-GPU communication, our tool locks the current GPU context to prevent many GPU operations from launching at the same time. After a kernel is submitted to the GPU, our tool leverages an independent stream to check whether the GPU queue is full repeatedly. Though the insert operation on the GPU queue lock-free, once the GPU queue is full, all active threads that operate on the GPU queue are blocked. Once all records in the queue are transferred to the CPU, a signal is sent to the GPU to resume its execution. Meanwhile, our tool tracks the number of active threads at thread exits. Once all threads are inactive (i.e., the current kernel is finished), our tool releases the GPU context lock.

Hierarchical Sampling  Our tool’s runtime overhead is proportional to the number of records collected. Typically, the larger the number of records, the higher the overhead. One can reduce the overhead by reducing the amount of measurement data collected on GPUs. We observed that most HPC applications employ iterative and data-parallel programming models; behaviors across different GPU kernel instances and across different thread blocks are similar. As a result, our tool employs a new hierarchical sampling mechanism to reduce runtime overhead but minimize measurement accuracy loss significantly. The hierarchical sampling consists of Kernel Sampling, Block Sampling, and Kernel Filtering.

Kernel sampling monitors a subset of instances of the same GPU kernel. Our tool uses the launching context to uniquely identify a GPU kernel. For example, if the same kernel is launched in two different calling contexts, our tool treats them as two different “kernels” instead of one. The calling context determination is done in the runtime system on the host CPU, which is described in Chapter 3. Our tool ensures
that each kernel is sampled at least once. A command line interface is exposed for users to specify the sampling period.

After applying kernel sampling, a large number of thread blocks used in a kernel instance can still incur a large overhead. In such a case, block sampling is used in complement to the kernel sampling. Block sampling randomly monitors thread blocks within a kernel instance. In the GPU callbacks, our tool checks whether a block should be monitored according to its block ID $b$. Our tool makes the decision with two more parameters: A sampling threshold $P$ defined by users and a random number $r$ generated upon each kernel launch. Our tool monitors a block $b$ only if $(b + r) \% P == 0$.

In our experiments, we often observed that not all GPU kernels are worth thorough optimization because many of them are not hotspots. Hence, our tool also supports monitoring a subset of GPU kernels specified by users. One can easily use domain knowledge or another profiling pass to identify important GPU kernels for instrumentation-based measurement.

2.4 Using GPUs to Process Measurement Data

With hierarchical sampling, we can reduce the amount of data collected on the GPU to mitigate measurement overhead. However, in some cases, it is still necessary to exhaustively measure every function and instruction. For instance, to measure the total accessed memory bytes of each GPU kernel, sampling and filtering can yield inaccurate results if blocks access many different addresses, which is common in irregular GPU kernels. However, exhaustive instrumentation can bring significant overhead. Without any optimization, our tool slows down the streamcluster code in Rodinia [29] by $1200 \times$, and it does not even finish measurement of other complex applications,
such as LAMMPS [25] and PyTorch [7]. To this end, we propose optimizations in this section to reduce our tool’s overhead without sampling for instrumentation-based measurement.

2.4.1 Overview

We designed an innovative mechanism to offload heavy data processing from the CPU to the GPU. Previously, CURD [71] and iGUARD [72] employed such an approach to detect GPU data race using GPU-accelerated processing. Their approaches reduce the overhead of copying large chunks of measurement data from the GPU to the CPU. However, they perform data processing only at instrumentation callbacks on the GPU. At best, these approaches leverage parallelism at the warp level; applying block/grid level parallelism will be difficult due to independent scheduling of warps and blocks on the GPU. In our approach, our data processing code is encapsulated in GPU kernel that is launched before each application kernel and executed concurrently with the application kernel. Compared with existing approaches, our method employs highly-parallel data processing using many GPU threads, little CPU-GPU transfer time, and concurrent execution of data processing and application kernels. Based on the most room policy [73], we let the analysis kernel occupy all resources of some streaming multiprocessors so that it will not be slowed down by sharing resources with an application kernel.

Figure 2.7 presents our pipeline for concurrent data processing on the GPU. The application kernel keeps putting measurement data in a GPU queue. Once the GPU queue is full, the analysis kernel grabs data from the queue, notifies the application kernel, and lets it resume execution. The analysis kernel then applies a custom data processing algorithm while the application kernel is executing. When the application
kernel ends, the final analysis result will be copied to the CPU.

Our data processing pipeline is both efficient and productive. Our analysis kernel runs concurrently with the application kernel so that GPU is only stalled when the analysis kernel copies data from the application kernel. Users can implement any feasible parallel algorithm for measurement data, regardless of the programming constraints imposed at instrumentation callbacks, where not all threads are active and shared memory cannot be used. In Section 2.4.2, we demonstrate the effectiveness of the solution using a parallel interval merge example. By incorporating parallel algorithm building blocks, such as Thrust [74] and CUB [75], we can easily implement other parallel processes, such as calculating prefix sum, finding peak values, and removing duplicates.

2.4.2 Parallel Interval Merge

We define an interval \((\text{start}, \text{end})\) as the memory range accessed by each GPU memory instruction. For complex GPU kernels, a vast number of intervals can be generated. Our tool merges these intervals if they are adjacent or overlap and copies the final intervals to the CPU when a kernel is done. The merged intervals can be used to facilitate subsequent analyses, such as value flow graphs [34] and memory liveness analysis.
One could copy all intervals from the GPU to the CPU and perform a sequential interval merge, which has a $O(N \log N)$ complexity, where $N$ denotes the number of intervals. This algorithm, however, only works for small GPU kernels as $N$ can be large in many benchmarks and applications (e.g., $3.4 \times 10^7$ for each kernel in stream-cluster), triggering large CPU processing and GPU-CPU memory copy overhead. To merge intervals on the GPU, we developed the parallel interval merge algorithm shown in Figure 2.8.

We describe the major steps in this algorithm as follows: 1. We first sort all intervals’ start and end addresses based on $\langle address, is\_end \rangle$ pairs such that an end
address is after a start address when they are equal. Next, we initialize a markers array to denote interval starts with 1 and interval ends with -1. We apply a parallel prefix scan on the markers array. The merged intervals cover a number of original intervals such that the prefix sum of merged interval starts are 1 and the interval ends are 0. We create a start flags array. Each entry in the array is zero, unless the corresponding start/end prefix sum value is 1 and the entry represents an interval start. We apply another parallel prefix scan to get output indices of the merged interval starts. The output indices of the merged interval ends are obtained similarly through steps and . Finally, we place the merged interval starts and interval ends to the output buffer and . The complexity of this parallel interval merge algorithm is using parallel radix sort, compared to the complexity achieved by analyzing intervals on the CPU.

This algorithm can be further optimized to merge intervals accessed by threads within the same warp using efficient warp primitives (i.e., shl, bfe, bfind, and brev). We refer to this simplified version as interval compaction. Unlike interval merge, interval compaction handles intervals with a fixed size. Specifically, interval merge can merge arbitrary $(start, end)$ pairs, while interval compaction only merges pairs of which $end - start = mem\_size$, where $mem\_size$ is the number of bytes of a memory instruction accesses. Interval compaction can be used for merging memory intervals of each warp because active threads within the same warp must execute the same memory instruction at a time.

We describe the algorithm for interval compaction as follows: We sort all interval starts using a warp sort, which is implemented using a sequence of shl instructions. Since all intervals in a warp share the same $mem\_size$, interval ends are not used. Next, we use another shl instruction to get the address of each
thread’s prior thread immediate predecessor. If a thread’s address is contiguous with its predecessor’s address, we assign 1 to the thread’s contiguous marker; otherwise, we assign 0. Next, we apply a \texttt{ballot} instruction to gather contiguous markers from all threads within the same warp. For threads whose contiguous marker is 0, we find the location of the nearest 1 in the gathered marker to the right of it. This location indicates the thread that has the end address of the interval with the current thread’s address as the start. To implement this operation in an efficient way, we use \texttt{bfind} and \texttt{brev} instructions. In the end, we apply the final \texttt{shfl} instruction to grab the end address, construct merged intervals, and output merged intervals to an output buffer.

The arithmetic complexity of \textit{interval compaction} is the same as \textit{interval merge}. However, in practice, \textit{interval compaction} can be much faster than \textit{interval merge}. 

Figure 2.9: An example of compacting intervals in parallel.
for two reasons. First, it adopts a warp sort algorithm that does not access global memory for data exchange. Second, apart from sorting, each of the other steps only takes \(O(1)\) complexity, while interval merge takes \(O(\log N)\) complexity in each step.

### 2.5 Overhead

This section presents an evaluation of the overhead of collecting instruction metrics and memory metrics on NVIDIA GPUs. We studied a wide range of GPU-accelerated applications and benchmarks, including Rodinia [29], Darknet [24], QMCPACK [76], Castro [77], BarraCUDA [78], PyTorch [7], NAMD [79], ExaTensor [80], Quicksilver [23], PeleC [81], Minimod [82], and BerkeleyGW [26].

**Instruction Metrics** We measured our tool’s overhead for collecting instruction metrics using both instruction sampling and instrumentation on an NVIDIA V100 GPU. In the sampling mode, our tool only collects instruction samples. In the hybrid
Figure 2.11: Our tool’s overhead in collecting memory metrics

mode, our tool measures each instruction’s executed count, L2 efficiency, shared memory efficiency, and predicated ratios. Figure 2.10 shows the slowdown of the hybrid mode compared to the sampling mode at runtime. CUPTI’s PC sampling mechanism by default serializes GPU kernels, increasing measurement time. For most benchmarks, the sampling mode introduces an overhead of less than 10×. However, it introduces 54× overhead to BerkeleyGW on cori-gpu [83]. Our tool’s hybrid mode has 2–14× slowdown compared to its sampling mode. For two ExaScale applications, PeleC and BerkeleyGW, the hybrid mode introduces 92× and 296× overhead, correspondingly.

**Memory Metrics** In addition to instruction metrics, our tool also collects memory metrics, such as the accessed addresses and values on the addresses. Currently, collecting these memory metrics requires instrumentation. We adopted both hierarchical sampling and GPU-accelerated prepossess to accelerate measurement. Figure 2.11 shows that our tool incurs moderate overhead, with a median of 7.35× on NVIDIA RTX 2080 Ti and 7.81× on NVIDIA A100. A100 has a lower overhead for applications that involve significant memory accesses, including Rodinia/streamcluster, Castro, BarraCUDA, and LAMMPS, because of its higher bandwidth memory than
RTX 2080 Ti. PyTorch-deepwave suffers from the highest overhead on both GPUs. This program accesses millions of different memory addresses for each kernel and results in about 100 thousand non-adjacent intervals after merging.

2.6 Summary

In this chapter, we describe measurement substrates based on instrumentation and hardware counters. Our tool employs instruction sampling to profile applications only once and derives other metrics using instruction samples. Compared with instrumentation, instruction sampling does not distort instruction execution in GPU kernels, yielding accurate measurement of time-sensitive metrics such as instruction latency. At the time of this writing, hardware support for instruction-level measurement of GPU performance is only available on NVIDIA GPUs. Intel is developing hardware support for instruction-level performance measurement that will become available in a future generation of its GPUs. Since CUDA 11.3, a new continuous PC sampling mode has been introduced. Details about using this mode to reduce measurement overhead further is covered in Chapter 3. The growing complexity of GPU programs make it important to have instruction sampling components in GPU hardware. Also, it would be useful if additional information could be associated with each instruction, such as on which memory hierarchy (e.g., L1, L2, or page fault) a memory instruction is stalled.

Though instrumentation for GPU binaries can introduce significant overhead, it is a flexible method for collecting many custom metrics without special hardware support. Our tool uses instrumentation to complement metrics that cannot be collected using instruction sampling, such as the predicated ratio of instructions. Besides, our tool collects addresses and values associated with memory accesses to enable redun-
dancy analysis as described in Chapter 5. With hierarchical sampling and GPU-accelerated data processing, we ameliorate instrumentation overhead to a moderate level acceptable for monitoring complex GPU-accelerated applications and benchmarks. Our data-parallel analysis approach opens up new opportunities for using GPUs to accelerate data-intensive analysis.
Chapter 3

Attribution of Measurement Data

Traditional GPU performance tools, such as VTune [20], nvprof [14], and ROCProfiler [22], collect performance metrics and attribute them to individual GPU operations, but they do not associate performance metrics with each GPU operation’s corresponding calling context [84]. In complex GPU-accelerated applications, a GPU operation can be called at many places. This necessitates calling context-sensitive analysis for GPU programs as it provides critical information about where GPU operations are invoked and what GPU code is executed to identify interprocedural optimizations. The growing complexity of programming models, such as Kokkos [12], RAJA [11], AMReX [85], PyTorch [7], and TensorFlow [8], further exacerbates the problem by separating GPU operations from execution details with many template-generated frames atop them.

This chapter presents approaches for attributing measurement data to call paths where they are incurred with low overhead. We designed a thread coordination and metrics attribution framework that interacts with underlying measurement substrates and runtime systems; currently, it is used to attribute performance metrics collected on Intel, AMD, and NVIDIA GPUs. Our profiler employs a novel and fast wait-free data structure for inter-thread communication to correlate performance metrics for each asynchronous GPU kernel invocation with its CPU calling context. To quickly obtain CPU calling contexts, we devised a novel call path memoization approach that obtains the call path of each GPU operation without walking the whole stack. To
attribute instruction samples with low overhead, we developed several range-based profiling methods that collect instruction samples without synchronizing the GPU and the CPU for every GPU kernel launch. In addition, to obtain domain-specific semantics, we designed and implemented profiling interfaces to refine raw call paths obtained from OpenMP and PyTorch applications. We evaluated our methods on many HPC and Machine Learning applications. Experimental results show that our framework is more stable, accurate, and efficient than existing solutions.

We provide background for call contexts and GPU programming models in Section 3.1. An overview of the framework is described in Section 3.2. Then, we present algorithms for wait-free communication in Section 3.3. Our methods for fast call path unwinding are described in Section 3.4. Next, we introduce range-based profiling methods for collecting instruction samples in Section 3.5. Finally, we sketch domain-specific profiling interfaces in Section 3.6.

3.1 Background and Related Work

In this section, we first describe methods for efficient call path profiling. Next, we sketch popular GPU programming models that add complexities to call path profiling.

3.1.1 Call Path Profiling

Calling context consists of function calls to a specific program location. We use the terms calling context, call path, and call path interchangeably in the following paragraphs. Calling context is important for performance analysis, and it has been extensively used in CPU profiling tools such as perf [86] and VTune [20]. Recently, HPCToolkit [87] and Nsight Systems [15] began to use calling context to annotate GPU APIs, including GPU kernels, memory APIs, and synchronizations, providing
users with a much better understanding of program performance than flat GPU profiles.

Each call path unwound at runtime is inserted into a tree to form a Calling Context Tree (CCT) for each CPU thread or GPU stream. In a CCT, each node represents the address of a machine instruction in a CPU or GPU binary as a (load module, offset) pair. When a CCT node is allocated, it receives a companion metrics array to store associated performance metrics. Our tool can collect well over 100 metrics; some for CPUs and some for GPUs. When measuring the performance of GPU-accelerated programs, many CCT nodes have CPU metrics only; all of their GPU metrics are zero. Storing zero values for all unused metrics at a CCT node would waste considerable memory. To reduce storage during measurement, our tool partitions metrics into kinds, such as GPU kernel info kind, GPU instruction stall kind, and CPU time.
kind. Figure 3.1 illustrates an example CCT and metrics associated with each node. Each node is associated with a metric kind list, and each metric kind represents an array of one or more metrics. For example, when measuring an execution with PC sampling, the CCT node representing a GPU kernel has *GPU kernel info* kind and *GPU instruction sampling info* kind. The kernel kind includes kernel running time, register usage, and shared memory usage, among others.

To obtain a call path at runtime, there exist many unwinding utilities such as libunwind [88]. However, as the frequency of stack unwinding becomes high, its costs become significant. For instance, Nsight Systems yields a $1.3\times$ overhead when tracing Laghos without call path annotation, while $3.5\times$ overhead is incurred if call path association is enabled. It is worth noting that this overhead not only dilates the overall execution time, but also increases the gap between each GPU activity because invocations to GPU APIs are delayed by stack unwinding.

There exist many approaches for reducing the overhead of walking the stack to collect call paths profiles. Nsight Systems enables users to customize unwinding threshold $T$. If stack unwinding is performed at GPU API A, and GPU API B is executed less than $T$ seconds after A, Nsight Systems won’t unwind B’s call path. Hence, this approach sacrifices the quality of call paths for lower runtime overhead. Zhuang et al. [89] proposed a method to stochastically capture all returns and calls below a call path sample instead of walking the stack. Froyd et al. [90] exploited common call path prefixes between adjacent samples to reduce the call path unwinding cost. Unlike Froyd’s approach, Servat et al. [91] recorded suffixes of call paths and inferred their prefix based on adjacent samples offline. These approaches reduce the unwinding cost based on the commonality between temporally adjacent samples but can still cause high overhead if adjacent samples do not share long common calling
contexts. Instead, our tool adopts an approach that reduces overhead in all cases regardless of the commonality between adjacent samples.

In addition to stack walking-based approaches, other work encodes calling contexts using unique identifiers and recovers call paths offline. By instrumenting each call site with a small integer, we can compute the identifier for a context by computing a hash from all of the integers along a call path and no longer performing expensive stack walking. However, this approach cannot achieve low overhead even when the call path unwinding frequency is high. Offline, calling context decoding decodes integers to recover the calling context of the program execution. Bond et al. [92] instrumented integers at each function call/return to track and encode calling contexts. This approach does not guarantee that each encoding refers to a single call path. Sumner et al. [93] extended the Ball-Larus algorithm [94] to generate unique numbers among all potential call paths offline. Mytkowicz et al. [95] proposed a method that uses the height of the stack as an indicator of the current call path. Both Sumner and Mytkowicz’s approaches use a training phase to gather call paths before the real measurement runs. The resulting calling contexts might be inaccurate if any call path is missed in the training phase. In contrast to the above methods, our tool does not require any instrumentation or training phases.

3.1.2 GPU Programming Models

Since the rise of General Purpose GPU Programming (GPGPU), there have been many emerging programming models whose initiatives are addressing the problems of productivity, performance, and portability in GPU code.

CUDA [96], DPC++ [97], and HIP [98] are the three default programming models on NVIDIA, Intel, and AMD GPUs accordingly. Among these three models, CUDA
compiler only generates executable on each vendor’s GPU families exclusively. HIP can generate code running on both NVIDIA and AMD GPUs. DPC++ can generate SYCL [99] code that be lowered to assembly code on many platforms.

Beyond the above three models, there also exists many other programming models [11, 12, 85, 7, 8, 13, 9, 10, 100, 101, 102] that focus on the efficiency and programmability of specific applications. As shown in Figure 3.2, we categorize them into two groups: high level and low level programming models. High level programming models customize runtime APIs to support specific data structures and executions. For instance, PyTorch [7] and TensorFlow [8] operate on Tensor data structures which are multi-dimensional arrays with slicing and indexing. PyTorch and TensorFlow by default reserve a large chunk of GPU memory at the beginning of the execution, which is used as a memory pool to avoid frequently calling CUDA malloc and free. GPU kernels are wrapped as tensor operations with Python interface. In this way, users do not need to write GPU code to implement complex formulas. Similarly, AMReX provides many common mesh refinement functions for users to quickly build HPC
applications. To extend these high level programming models using highly efficient GPU code, users still need to write code following the default GPU programming models on each platform, even though models such as RAJA and Kokkos simplify GPU programming by encapsulating common device functions for multiple GPUs.

Because GPU operations provided by high level programming models are the most common ones, implementing custom operators by combining general operations often decreases the performance. A common solution to boost performance is using just-in-time (JIT) compilation that fuses multiple GPU kernels together automatically. Currently, tools such as XLA [103] and TorchScript [104] have basic support to fuse deep learning operators. Kernel fusion has many limitations, including the increase in the number of registers, requirements for static and dynamic analysis, and limitations to applying optimizations at all levels.

Compared with high level programming models and JIT, low level programming models customize thread executions on the GPU. For instance, MLIR [100] incorporates primitives for machine learning operations, whose operands are often a chunk of tensor. Designed for image processing, Halide [101] employs many vectorized primitives. To better support tensor cores, Triton [9] eliminates the concept of threads on the GPU, only allowing users to program on the GPU block level. These programming models try to simplify the multi-level parallelism on the GPU to make using low level models as simple as writing a single-threaded application on the CPU and at the same time yielding high performance. Most languages employ LLVM to translate high-level code to low-level assembly languages. OpenMP [13] customizes both high and low level models.

Using these programming models complicates the calling contexts our tool obtains. Often, we use vendor provided tool APIs, such as CUPTI [46], ROCTracer [21], and
Level Zero [47], to monitor GPU-accelerated applications. Vendor APIs are designed to monitor low level driver APIs but are unaware of the runtime and the application’s contexts. On one hand, vendor APIs can be generally applied to many applications. On the other hand, tools are not able to apply custom routines to refine program contexts. For instance, in OpenMP, a worker thread is not guaranteed to launch at the place where it is created. Thus, we have to concatenate the context of a worker thread with its launch context that happens on the master thread. To obtain the information, we need to understand the semantics of the OpenMP runtime to know where threads are launched. The OMPT interface [105] is designed for this purpose. Besides, by knowing who creates the contexts, we can eliminate driver and tool context, which users often cannot control.

Figure 3.3 demonstrates the complexity added by GPU programming models. Inspired by OMPT, we can design profiling interfaces for other programming models. We call this kind of design _domain specific profiling interface_ in the following sections.
Figure 3.4: Our tool’s infrastructure which coordinates application threads, a monitor thread, and tracing threads for metrics attribution.

3.2 Overview

This section presents an overview of our tool’s metrics attribution workflow, which is illustrated in Figure 3.4. When monitoring the execution of GPU-accelerated applications, there exist several types of CPU threads. Application threads offload computations to GPUs. Our tool employs a GPU monitor thread to asynchronously process measurement data from the GPUs. If tracing is enabled, our tool creates one or more tracing threads to record an activity trace for each GPU stream.

When an application thread performs an invocation $I$ of a GPU operation (e.g., a kernel or a data copy), our tool unwinds the application thread’s call path to determine the CPU calling context of $I$, inserts a placeholder $P$ representing the operation in that context, communicates $I$ and $P$ it to the monitor thread, and initiates the GPU operation after tagging it with $I$. When tracing is enabled, the monitor thread separates GPU activities by their associated stream and sends each stream of activities to a tracing thread. Each tracing thread records one or more GPU
streams of activities and their timestamps into trace files. For efficient inter-thread communication, our tool uses bidirectional channels, each consisting of a pair of wait-free single-producer and single-consumer channel, which is elaborated in Section 3.3.

The precise instantiation of our tool monitoring infrastructure is tailored to each GPU vendor’s software for monitoring GPU computations. For different measurement substrates, we employ different methods to coordinate the communication between application threads and the monitor thread. When using NVIDIA’s CUPTI [46] and AMD’s ROCTracer [21] libraries for monitoring GPU activities, a monitor thread created by these libraries receives measurements of GPU activities via a buffer completion callback. The buffer contains one or more GPU activities $A_1, \ldots, A_n$ for each GPU API invocation $I$. At each GPU API invocation $I$, our tool unwinds the CPU call path and creates a placeholder node $P$. Each application thread $T$ shares two channels with the GPU monitor thread, including an activity channel $C_A$, from which $T$ receives information about GPU activities associated with operations it invoked, and an operation channel $C_O$ on which $T$ enqueues GPU operation tuples of $(I, P, C_A)$, representing an invocation $I$, its associated placeholder $P$, and its activity channel $C_A$. Every time the GPU monitor thread receives a buffer completion callback, it drains its incident operation channels prior to processing a buffer full of GPU activities. The GPU monitor thread matches each GPU activity $A$, tagged with its invocation $I$, with its associated operation tuple $(I, P, C_A)$. The monitor thread enqueues a pair $(A, P)$ into activity channel $C_A$ to attribute the GPU activity back to $T$.

When using OpenCL [106] and Level Zero [47], depending upon the GPU operation invoked, either an application thread or a runtime thread will receive a completion callback providing measurement data. At each GPU API invocation $I$ by
an application thread $\mathcal{T}$, our tool provides a *user.data* parameter, which includes a placeholder node $\mathcal{P}$ for the invocation $I$) and $\mathcal{T}$'s activity channel $\mathcal{C}_A$. The OpenCL or Level Zero runtime will pass *user.data* to the completion callback associated with $I$. At each completion callback, some thread receives measurement data about a GPU activity $\mathcal{A}$. Using information from its *user.data* argument, the completion callback correlates $\mathcal{A}$ with placeholder $\mathcal{P}$ and then enqueues an operation of $(\mathcal{A}, \mathcal{P}, \mathcal{C}_A)$ for the monitor thread in its operation channel $\mathcal{C}_O$. The monitor thread enqueues an $(\mathcal{A}, \mathcal{P})$ pair in $\mathcal{T}$'s activity channel $\mathcal{C}_A$. If the thread receiving the callback enqueued $(\mathcal{A}, \mathcal{P})$ pairs directly into $\mathcal{T}$'s activity channel $\mathcal{C}_A$, $\mathcal{C}_A$ would need to be a multi-producer queue since more than one thread may receive completion callbacks for $\mathcal{T}$. Our design, which employs a GPU monitor thread created by our tool, replaces the need for a multi-producer queue with several wait-free single producer queues.

When tracing is enabled, the monitor thread checks the GPU stream id $\mathcal{S}$ of each GPU activity and enqueues the activity and its placeholder $\mathcal{P}$ into a trace channel for $\mathcal{S}$. One or more tracing threads handle the recording of traces. Each tracing thread handles a set of trace channels by polling each channel periodically and processing its activities. For each activity in a trace channel for stream $\mathcal{S}$, the tracing thread records its timestamp and placeholder in a trace file for $\mathcal{S}$. Depending on the number of application threads used, the number of tracing threads can be adjusted by users to balance tracing efficiency with tool resource utilization.

### 3.3 Wait-free Communication

When attributing the performance of asynchronous kernel executions back to the application threads that initiated them, communication between a GPU monitor thread and application threads becomes frequent. Delaying any thread with lock-
Algorithm 1 A wait-free producer/consumer channel.

1: struct Node
2:    Node *next; Record r
3: 4: struct Channel
5:    Node *consumer_in  ▷ a list of nodes for the consumer
6:    Node *producer_out ▷ a list of nodes from the producer
7: 8: procedure PRODUCER_PUSH(Channel c, Node n) ▷ races with consumer_STEAL
9:          repeat
10:             n.next ← atomic_load(&c.producer_out)
11:          until atomic_compare_exchange(&c.producer_out, &n.next, n)
12: 13: procedure CONSUMER_STEAL(Channel c) ▷ races with producer_PUSH
14:          c.consumer_in ← atomic_exchange(&c.producer_out, NULL)
15: 16: function CONSUMER_POP(Channel c) ▷ serial with consumer_STEAL
17:          first ← c.consumer_in
18:          if first is not NULL then c.consumer_in ← first.next
19:          return first

ing or unbounded attempts of lock-free operations could disrupt execution behavior. Moreover, for long-running applications, memory for data passed between threads must be reclaimed. We designed a channel data structure that supports wait-free communication of invocation and activity records between the GPU monitor thread and each application thread.

A Wait-free Communication Channel Algorithm 1 shows a point-to-point channel designed for producer/consumer communication. The channel consists of a shared linked list producer_out and a private linked list consumer_in. A producer_PUSH repeatedly tries to add a node at the head of producer_out, like a Treiber stack push [107]. A consumer_STEAL points consumer_in at a list of nodes atomically removed from producer_out. A consumer_POP extracts and re-
turns an element from the head of consumer_in. The communication channel is wait-free: consumer_POP and consumer_STEAL execute only a few instructions; the atomic_compare_exchange* in producer_PUSH may fail at most once when a concurrent consumer_STEAL sets producer_out to NULL. Once producer_out is NULL, any additional consumer_STEALs cannot change that. Thus, the next atomic_load of producer_out by producer_PUSH will yield NULL and the next atomic_compare_exchange in producer_PUSH will succeed. The linearization point of producer_PUSH is its successful atomic_compare_exchange. The linearization point of consumer_STEAL is its atomic_exchange. Note that the channel is not FIFO; one could make it FIFO by reversing the list stolen from producer_out prior to setting consumer_in in consumer_STEAL. This change would not compromise wait-freedom because the cost of reversing the list is bounded and is proportional to the list length.

A Wait-free Coordination Protocol A bidirectional channel for producer-consumer communication consists of a pair of wait-free channels. Records flow from a producer to a consumer. The consumer ingests records transmitted and then returns them to the producer for reuse. Each application thread communicates with the GPU monitor thread using a pair of bidirectional channels, as shown in Figure 3.5. Next, we describe the four operations in Figure 3.5.

- **Application Thread Produce.** When a GPU API is invoked, an application thread calls Algorithm 2’s PRODUCE to push a invocation record into a produced invocation channel. Before allocating an invocation record, Line 5

---

*The primitive atomic_compare_exchange(&mem, &oldval, newval) will return true if the value of mem can be atomically changed from oldval to newval. If mem no longer has the value oldval, the primitive will return false and oldval will be updated with the value of mem.*
Algorithm 2 Wait-free producer-consumer protocol

1: struct BidirectionalChannel
2:     Channel produced, consumed
3:
4: function PRODUCE(BidirectionalChannel b, Record r)
5:     if node ← consumer_POP(b.consumed) is NULL then
6:         consumer_STEAL(b.consumed)
7:         node ← consumer_POP(b.consumed)
8:     if node is NULL then node ← allocate()
9:     node.r ← r
10:     producer_PUSH(b.produced, node)
11:
12: function CONSUME(BidirectionalChannel b, CallBack fn)
13:     consumer_STEAL(b.produced)
14:     while node ← consumer_POP(b.produced) is not NULL do
15:         fn(node.r)
16:     producer_PUSH(b.consumed, node)

attempts to pop a node from the consumed invocation channel. If the consumer_in list in the consumed invocation channel is empty, Line 6 updates it by stealing nodes (if any) from the channel’s producer_out list. If the consumed channel is empty, a new node is allocated at Line 8 to communicate the invocation record. The record r is entered in the empty node at Line 9. Finally, Line 10 pushes the node into the produced invocation channel.

- **Monitor Thread Consume.** When the GPU monitor thread receives a buffer of records from GPUs, it calls CONSUME to steal all invocation records previously pushed by the application thread (Line 13) and processes them using a callback function (fn) at Line 15. Consumed records are pushed into the consumed invocation channel (Line 16) for reuse by Application Thread Produce.

- **Monitor Thread Produce.** After the GPU monitor thread finishes Monitor Thread Consume, it adds activity records to its produced channel by calling
Figure 3.5: Interactions between the GPU monitor thread and application threads. pc0, pc1, and pc2 denote PC samples

**PRODUCE.**

- **Application Thread Consume.** Before the next GPU API call, an application thread calls **CONSUME** to process activity records pushed by **Monitor Thread Produce**. A callback function attributes metrics to the corresponding calling context at Line 15. As above, consumed records are returned to their producer for reuse.

An invariant at the end of each **CONSUME** call is that `produced.consumer.in` is empty, so the next **CONSUME** call may overwrite it without losing records. Each bidirectional channel is *wait-free* because Line 10 fails at most once if Line 13 executes concurrently, and Line 16 fails at most once when Line 6 executes concurrently. Using wait-free channels to communicate activity and invocation records between application threads and the GPU monitor thread accounts for only 2% of the time of a
monitored execution.

3.4 Fast Call Path Unwinding

Sampling-based performance tools monitor program execution on CPUs using asynchronous sampling. Typically, call path unwinding is performed when an asynchronous profiling event (e.g., a HW counter overflow event or a timer interrupt), triggers a sample. However, for GPU-accelerated programs, our tool unwinds the call path at every GPU API call. While one could use sampling to collect information only about a subset of GPU operation invocations; however, this approach can miss costly GPU operations and is fundamentally different from lazily collecting asynchronous call path samples on the CPU. To monitor frequent GPU invocations and attribute their cost to calling contexts without significant overhead, unwinding must be fast. To address this challenge, we designed several novel mechanisms to accelerate unwinding.

3.4.1 Improving Existing Unwinders

Our tool’s unwinding substrate employs the hpcrun tool from HPCToolkit [87]. We identified inefficiencies in hpcrun and applied optimizations. First, we avoided unnecessary unwinding of stack frames. In programs that frequently offload operations to GPUs, temporally adjacent GPU operations often share a long common call path prefix. We identify the adjacent common call path prefix unwind by using trampolines [90] on x86_64 and mark bits [108] on Power. Once the call path unwinder reaches a stack frame in common with the previous unwind, it unwinds no further.

Second, we accelerated lookup of unwind recipes. As a program executes, our tool computes and memoizes unwind recipes in a concurrent skip list. Prior to our
modifications, a thread acquires a read lock and searches the skip list for each unwind step. To avoid lock contention and the expected $O(\log n)$ lookup cost in a skip list with $n$ entries, each thread now employs a 1000-entry hashtable to memoize pointers to recipes found in the skip list. This reduces the expected cost of finding a recipe to $O(1)$ as we search the concurrent skip list only when a hashtable lookup fails.

Third, we eliminated redundant unwinding associated with CUDA API calls. Any CUDA driver API called by a CUDA runtime API shares its application calling context with the runtime API; thus, unwinding of the driver API can be skipped. Our three enhancements reduced overall measurement overhead by $3.5 \times$ for frequent GPU API calls by the Laghos code, compared with the overhead of hpcrun.

### 3.4.2 Memoization-based Unwinding

After applying the above optimizations, we still observed up to $3 \times$ overhead when unwinding frequent GPU API calls at many different contexts in complex applications. To further improve unwinding, we designed a memoization mechanism to fast obtain the CPU call path of each GPU operation.

The intuition is that we can design an easy-to-compute identifier for the call path of a GPU operation and memoize the mapping between an identifier and a call path, and avoid expensive stack walking when encountering the same identifier. We use a stochastic backoff mechanism to update map entries to handle identifier collision, meaning different call paths share the same identifier. Figure 3.6 demonstrates the workflow of this mechanism. After entering a GPU API, we first generate a call path identifier $I$ for the current invocation. Then, we query a Call Path Map to check if $I$ already exists. If $I$ is in the map, we enter the fast path; otherwise, we enter the slow path. In the slow path, we unwind the program stack to construct the full call path for
Figure 3.6: The memoization mechanism to query, get, and update CPU call paths.

and update \( \mathcal{I} \)’s mapping. In the fast path, we get \( \mathcal{I} \)’s call path from the map, which is likely but not guaranteed to be the correct call path for the current invocation. Then, we stochastically enter the slow path to compare the call path we found in the map with the current we call path just unwound. If two call paths are different, we rectify the mapping between the identifier and the call path. The probability of entering the slow path is controlled by a backoff parameter \( B \). Using identifiers with less ambiguity, we can lower the backoff probability to reduce the chance of entering the slow path.
Figure 3.7: An example program and the stack of $K_1$ and $K_2$ when they are invoked. We use letters to represent GPU kernels and subscripts to represent different call paths. rsp denotes the current location in stack.

Call Path Identifier

Definition 3.1 A call path identification is a mapping $I \rightarrow P$, where $I$ is a set of identifiers, and $P$ is a set of call paths.

Definition 3.2 A $k$–approximate call path identification is a mapping $I \rightarrow P$, $\not\exists i \in I$, $i \rightarrow P'$, and $|P'| \geq k$.

Based on Definition 3.1 and its relaxed form Definition 3.2, the key issue of designing an identification method is to balance computing cost and limiting $k$ to be as small as possible to yield high accuracy.

Inspired by previous work [95], we first use the stack height as the identifier, which is the value of the current stack pointer subtracted by the stack base. For each function invocation, an “activation record” is pushed onto the stack, which consists
of local variables, return address, and function parameters. Upon the return of a
function, the activation record is popped from the stack. Thereby, the stack height
can be a good indicator to differentiate between call paths. For instance, in Figure 3.7,
assuming function A is the entry function of the program, the call paths of GPU kernel
\( K \) in function A (\( K_1 \)) and function B (\( K_2 \)) are unambiguous using their stack heights
as an indicator. This simple approach, however, does not apply to other cases such
as \( K_3 \) and \( G_1 \) because their stack heights are the same. To extend the approach, we
add the \( \langle \text{binary} \_\text{id}, \text{function} \_\text{id} \rangle \) pair obtained from the \texttt{CUfunction} argument [96]
of each kernel launch invocation to the identification method. A \text{binary} \_\text{id} is a unique
identifier for GPU binaries loaded at runtime. A \text{function} \_\text{id} is a GPU kernel’s symbol
id in the GPU binary. Hence, each GPU kernel has a unique \( \langle \text{binary} \_\text{id}, \text{function} \_\text{id} \rangle \) pair.

\textit{Definition 3.3} The basic identifier \( I_{\text{basic}} \) is a tuple:

\( \langle \text{stack} \_\text{height}, \text{binary} \_\text{id}, \text{function} \_\text{id} \rangle \)

Using the basic identifier proposed in Definition 3.3, we can differentiate between \( K_3 \)
and \( G_1 \) because

\[
I_{\text{basic}}(K_3) = \langle \text{binary} \_\text{id}(K), \text{function} \_\text{id}(K), \text{stack} \_\text{height}(K_3) \rangle
\]

\[
I_{\text{basic}}(G_1) = \langle \text{binary} \_\text{id}(G), \text{function} \_\text{id}(G), \text{stack} \_\text{height}(G_1) \rangle
\]

where \( \text{stack} \_\text{height}(G_1) = \text{stack} \_\text{height}(K_3) \) but

\( \langle \text{binary} \_\text{id}(G), \text{function} \_\text{id}(G) \rangle \neq \langle \text{binary} \_\text{id}(K), \text{function} \_\text{id}(K) \rangle. \)
GPU Operation Tracing

$I_{basic}$ can still be ambiguous in common cases if the same kernel is launched at different call sites in a function. For example, in Figure 3.6, $G_1$ and $G_2$ cannot be differentiated based on $I_{basic}$, assuming no temporary stack variable is allocated in between. Previous research [95, 93] instruments binaries to allocate extra space to adjust the stack depth of ambiguous program locations. This solution not only increases the memory usage but also is limited if variable size arrays are allocated on the stack. To solve this problem without explicitly adjusting the stack height, we introduce the **GPU Operation Trace** (Definition 3.4) as part of the identifier.

**Definition 3.4** The GPU operation trace $O_k$ of a GPU operation $o_i$ is a sequence of call paths of previous GPU operations $p_{i-1},...,p_{i-k}$, where $k$ denotes the size of the trace.

If we set $k$ as the total number of GPU operations in the execution, each operation will have a different API trace. In practice, we limit $k = 1$ to reduce the memory and time cost of recording and comparing the operation trace in the identifier. Figure 3.8 shows the full operation trace of each operation.

**Definition 3.5** The advanced identifier $I_{advanced}$ is a tuple:

$\langle stack\_height, binary\_id, function\_id, O_k \rangle$

By adding the API trace to form the advanced identification function (Definition 3.5), $G_1$ and $G_2$ are unambiguous using a trace size of one, because $O_1(G_1) = K_3$, while $O_1(G_2) = G_1$. 
Backoff Mechanism

We still have a chance to map an identifier to multiple call paths even with $I_{\text{advanced}}$. For example, in Figure 3.8, $I_{\text{advanced}}(G_3)$ is the same as $I_{\text{advanced}}(G_2)$. Though increasing the size of the operation trace can mitigate this problem, it also increases the overhead in storing and comparing the trace. In order to solve this issue, we devise a backoff mechanism to stochastically unwind the call path using the slow path to self-correct wrong identification. This mechanism involves the following steps:

1. Set up a unwind probability $U$, where $0 < U \leq 1$.

2. For each identification tuple generated by $I_{\text{advanced}}$, we maintain a backoff parameter $B$ whose initial value is 0.

3. After entering a GPU operation, generate a random number $n$ and compare it with $U^B$. If $n \leq U^B$, enter the slow path. Otherwise, go to step 5.
4. Compare the unwound call path with the call path obtained from the call path map. If they are different, set $B = 0$. Otherwise, set $B = B + 1$.

5. Return the call path.

Using the backoff mechanism, $G_2$ and $G_3$ can be differentiated because their unwound call path is different from their memoized call path so that they always enter the slow path to unwind the call path. Since $B$ is private to each identifier, other GPU operations still execute the fast path. By assigning the unwinding probability $U$ a large value, we can improve the accuracy of our method by increasing the chance of entering the slow path. In our experiments, we found $U = 1/4$ can achieve a good trade-off between accuracy and speed for most applications.

### 3.5 Range-based Attribution and Aggregation

![Figure 3.9: Serialized profiling](image)

![Figure 3.10: Continuous profiling](image)
NVIDIA’s CUPTI API provides two default methods for collecting instruction samples: the serialized method and the continuous method, as shown in Figure 3.9 and Figure 3.10 respectively. Using the serialized method, our tool synchronizes each GPU kernel, copies instruction samples from the GPU to the CPU, and associates instruction samples with the current kernel’s call path. While the serialized method is accurate for attributing instruction samples to the corresponding kernel, its synchronization and metrics attribution can cause significant overhead in large applications. In contrast, using the continuous method, one can synchronize the GPU and the CPU once at the end of the entire execution to reduce the synchronization and the attribution cost. However, with that approach, we lose accurate instruction sample attribution when a GPU kernel is called at different call paths. We now describe several novel range-based profiling methods to balance overhead and attribution accuracy.

### 3.5.1 Fixed Range Profiling

Figure 3.11 illustrates the execution of the fixed range profiling method for collecting instruction samples. Given a sequence of GPU kernels, we divide them into several ranges with equal size. At the end of each range, we synchronize the GPU and
the CPU, communicate GPU instruction samples to the CPU, and attribute samples to the CPU call paths where they are initiated. Unlike the serialized method, we dispatch samples to a background CPU thread for attribution after copying instruction samples. As a result, the execution of the application thread and the background thread can be overlapped to reduce the overhead.

At runtime, we maintain separate calling context trees for instruction samples and the application, as shown in Figure 3.12. To avoid a data race when accessing the heterogeneous calling context tree maintained by the application thread, the monitor thread attributes instruction samples to their corresponding ranges in a Range Calling Context tree. Postmortem, we attribute samples back to the heterogeneous calling context tree. In each range, a GPU kernel can be called from multiple call paths. For example, GPU kernel A in Figure 3.12 has two call paths $A_0$ and $A_1$ in range $R_0$. In such a case, we apportion samples between call sites $A_0$ and $A_1$ based on certain metrics (e.g., call counts and the number of threads) to each call path. By reducing

Figure 3.12: Attributing instruction samples to the heterogeneous calling context tree.
the size of the range, we can reduce the chance of having multiple call paths within a range but increase the synchronization and metrics attribution cost. With a range size of one, we fall back to the serialized mode.

3.5.2 Adaptive Range Profiling and Sampling

Instead of using a fixed range size, we designed another method—adaptive range profiling, in which the range sizes vary. In this method, we maintain a range call path set $S_R$ to store call paths of all launched GPU kernels in the current range. If the next GPU kernel being launched exists in $S_R$ but has a different call path than the one in $S_R$, we say this kernel conflicts with $S_R$. To handle the conflict (e.g., Kernel $A_2$ in Figure 3.13), we clear $S_R$ and stop profiling. Then, we flush instruction samples from the GPU to the CPU and dispatch instruction samples to the background thread. In the end, we restart profiling for the next range. The size of each range depends on the number of contiguous GPU kernels that do not conflict with each other. If many GPU kernels only have a single call path in the application, we could have a long range without losing sample attribution accuracy.

Both the adaptive and the fixed range profiling methods measure and attribute instruction samples of every range. In long-running applications, GPU kernels are...
often repeatedly launched with similar or patterned behaviors. Therefore, we can sample some kernels and scale the number of instruction samples based on the ratio between total number of kernels and the number of sampled kernels, reducing both running time and memory overhead. Previous work employs kernel sampling [49, 38] to monitor a subset of GPU kernels but requires synchronization before and after each kernel invocation. In contrast, we devised a sampling approach that randomly starts profiling from a GPU kernel and only stops if the next GPU kernel conflicts with GPU kernels in the current range to reduce the synchronization and attribution cost. Besides the range call path set $S_R$, we allocate a global call path set $S_G$ to store all call paths to GPU kernels during the execution. For each GPU kernel’s call path $p_i$, if $p_i$ is not in $S_G$, we start profiling a new range. Once a range terminates because the upcoming kernel’s call path $p_j$ conflicts with $S_R$, we choose to randomly start the next range with an enabling rate $E$, if $p_j$ is in $S_G$. In Figure 3.13, at the second time we encounter $A_1$, we may either start profiling or skip profiling.

### 3.5.3 Range Aggregation

The number of ranges increases with an application’s execution time. Therefore, the range calling context tree in Figure 3.12 can grow significantly in long-running applications, consuming a considerable amount of memory. To address this issue, we designed an algorithm that uses a trie [109] to identify repeated ranges of the same GPU kernel sequences and aggregate samples in those ranges to reduce memory consumption. We refer to the trie as the *call path trie* in the following paragraphs.

In a call path trie, each node is keyed with a GPU kernel’s call path $p$. Before adding a range to the trie, we assign it a *range id*, which starts from one and is incremented when we start a new range. Each node in the call path trie has a
first_range_id field to record the range_id when the node is added into the trie, used to identify repetitive ranges. Each node also maintains a map from a range_id to its sampled count and invocation count, which are used to scale instruction samples to account for not profiled ranges. If a node \( p_i \) is a child of a node \( p_j \), it means \( p_i \)’s GPU kernel is launched right after \( p_j \)’s GPU kernel. The trie internally maintains a current pointer to the node being operated.

A call path trie supports three operations: insert, is_child, and unwind. The insert operation inserts a call path \( p_i \) as a child of the current node and assign range_id to \( p_i \)’s first_range_id if it is empty. The is_child operation checks whether \( p_i \) is a child of the current node. The unwind operation takes a range_id as input and traverses upward from the current node pointer to the root. During the traversal, it increments each node’s invocation count under the given range_id. If instruction sampling is active, it also increments each node’s sampled count. For a profiled range \( p_0...p_n \), our algorithm collects instruction samples for the range and stores them to \( p_n.\)first_range_id in the range profile tree so that samples triggered by repeated execution can be aggregated to the same range during the execution. We use the postmortem analysis to attribute instruction samples to each kernel.

Algorithm 3 is executed before launching each GPU kernel to construct the call path trie. At Line 9, we first check if \( p_i \) is different from any call path in \( S_R \) with the same GPU function. If so, it means we have encountered a conflict kernel, and we need to end the current range. More specifically, we get the first_range_id of the end node, unwind the call path trie, and increment each node’s invocation counts under the first_range_id entry along with the traversal (Line 11). If instruction sampling is active, we stop it and attribute instruction samples under the first_range_id node in the range calling context tree. Finally, we start a new range at Line 14 and increment
Algorithm 3 An algorithm for aggregating profiling ranges

1: **Input:**
2: $trie$ : A call path trie
3: $S_R$ : A call path set of the current range
4: $p_i$ : The call path of the next GPU kernel
5: $prev\_range\_id$ : The previous range’s id
6: **Output:**
7: $range\_id$ : The current range’s id
8:
9: if $S_R.lookup(p_i)$ is **True** then
10: \hspace{1em} $first\_range\_id \leftarrow trie.current.first\_range\_id$
11: $trie.unwind(first\_range\_id)$
12: if $instruction\_sampling\_active()$ is **True** then
13: \hspace{1em} $instruction\_sampling\_stop(first\_range\_id)$
14: $S_R \leftarrow \emptyset$
15: $range\_id \leftarrow range\_id + 1$
16: $S_R \leftarrow S_R \cup \{p_i\}$
17: $repeated \leftarrow trie.is\_child(p_i)$
18: if $instruction\_sampling\_active()$ is **False** then
19: \hspace{1em} if $repeated$ is **False** then
20: \hspace{2em} $instruction\_sampling\_start()$
21: \hspace{1em} else if $range\_enable()$ is **True** then
22: \hspace{2em} $trie.unwind(trie.current.first\_range\_id)$
23: \hspace{2em} $S_R \leftarrow \{p_i\}$
24: \hspace{2em} $instruction\_sampling\_start()$
25: $trie.insert(p_i, range\_id)$
26: **return** $range\_id$

the $range\_id$ at Line 15.

At Line 16, we add $p_i$ to the current range. Then, we check if $p_i$ is already a child of the current node at Line 17. If so, it means the current kernel (sub-)sequence has been recorded in the trie such that the current range is repeated. Next, if instruction sampling is not active, we may start it in two cases. First, if repeated is False, it is the first time we see this (sub-)sequence; we start instruction sampling at Line 20. Second, instruction sampling can be enabled randomly at Line 21, and we unwind the trie and initialize a new range set with $p_i$. Finally, we insert $p_i$ into the trie at
Figure 3.14: An example of recording GPU kernels using a call path trie. The $R_i : s/n$ annotation on the side of each node indicates $s$ out of $n$ kernels are sampled in $R_i$. The red $R_i$s are the first range of each node. The green boxes enclose kernels that have been profiled; the orange boxes enclose skipped kernels. We track the output range_id after applying Algorithm 3 for each kernel.

Line 25. If $p_i$ is a new child of the current node, $p_i$’s first range_id is assigned as range_id.

Figure 3.14 demonstrates an example for constructing the trie. In Figure 3.14a, since $A_2$ conflicts $A_1$, we unwind the trie and insert $A_2$ as a child of the root. Then, when we encounter existing nodes in the future, we may skip profiling based on random numbers generated. In Figure 3.14b, since $A_1B_1C_2$ is a new sequence, we
assign $A_1$ and $B_1$ corresponding kernel counts under the new range $R_3$. Next, in Figure 3.14c, we skip $A_1$ again and start profiling from $B_1$ to record a new sequence. In Figure 3.14d, we encounter $A_1B_1C_2$ again and skip profiling for all kernels. Without any compression, the above operations need six ranges to store samples. With compression, we use only four ranges as we aggregate instruction samples to the first range of the end node at each repeated sequence.

### 3.6 Domain-specific Profiling Interface

To enable profiling for complex applications, we need to gather sophisticated information about program semantics, runtime states, and contexts. Though general performance tools can collect metrics about operations on CPUs and GPUs, they only attribute metrics to program contexts based on debugging information. If an application is composed of many sophisticated runtime libraries spawning threads to schedule execution, the resulting calling contexts cannot provide enough insight for optimization. One way to solve the challenge is to annotate source code. While source code instrumentation can collect high-level information, it is application-specific and lacks portability; tool developers have to implement profiling functionalities for every application to be monitored, even if they are within the same domain. To enable portable tools for performance analysis, we can define a programming interface for tools as a standard for exchanging measurement data between applications and tools. In the following sections, we first describe the background of OpenMP Tool interface (OMPT). We implemented a measurement substrate using OMPT for NVIDIA GPUs. Then, we propose an original design of a deep learning profiling interface and its draft implementation for monitoring deep learning applications.
3.6.1 OpenMP Tool Interface

OMPT [105] consists of APIs to query and exchange data between the OpenMP runtime and performance tools. Since it’s been integrated into the OpenMP standard [13], many compilers have implemented OMPT as a runtime measurement substrate, including Intel’s ICC [110], LLVM [111], and the NVIDIA’s nvc/nvc++ [112]. All OMPT functions have a C binding only as tools are often implemented in C and C++.

The design goal of OMPT is to expose OpenMP runtime status to tools to better monitor OpenMP applications. First, the OMPT API provides event interfaces to trace the OpenMP runtime, along with utility functions to identify call frames belonging to the OpenMP runtime. Developers can use event enter/exit callbacks to mark the beginning and the end of runtime regions. Our tool focuses on monitoring offloading (i.e., target) regions. Specifically, our tool implements OMPT callbacks of target, target_data_op, target_data_submit, and target_target_map regions, in the forms of pragma or function invocations. Unlike individual GPU operations, these target regions can involve one or more operations. For example, an OMPT target_data_op can operate on multiple data objects, while call frames within the OpenMP runtime are often of interest to users. Hence, our tool only unwinds a call path once at the entry point of a target region and associates all metrics underlying to the target region.

Under a target region, GPU operations are classified according to their operation types.

Second, OMPT supports an interface to identify the runtime status of OpenMP threads during execution, which will enable a performance tool to interpret program execution behavior. Representative states are work, idle, barrier, and wait. Guided by states, our tool can incorporate the blaming shifting technique [113] to transfer
blame for GPU waiting or CPU waiting from contexts where waiting is observed to code responsible for the waiting.

Last, OMPT imposes negligible overhead to the OpenMP runtime system in our use cases while enabling our tools to gather sufficient information about an OpenMP program to associate costs with the program and the runtime system. Currently, the main monitoring overhead in our tool is caused by CUPTI, which typically brings 20% overhead for applications with frequent GPU API invocations.

### 3.6.2 Deep Learning Profiling Interface

![Figure 3.15 : The hierarchy of GPU performance tools.](image)

Figure 3.15 demonstrates the hierarchy of GPU performance tools. General profiling tools employ low-level profiling APIs to collect performance metrics and at best associate them with calling contexts using line mapping information. Other critical information to understand the behavior of deep learning applications, such as the operator to which a GPU API belongs and the shapes of its input tensors, are not available through general profiling tools. PyTorch and TensorFlow’s built-in profilers provide more program context through Traceme [114] and Kineton [115] profiling APIs correspondingly. However, these APIs are tightly bound to frameworks. More than that, while built-in performance tools are capable of collecting more framework-
specific information, they do not collect fine-grained GPU performance metrics with hardware counters or binary instrumentation, which are well supported by general performance tools. Finally, neither Traceme nor Kineto can profile applications at a large scale with a low overhead compared to general GPU performance tools.

In contrast to the above two tools that are bound to specific frameworks, DL-Prof [116] is a portable deep learning performance tool that supports analysis of both PyTorch and TensorFlow. At runtime, DLProf employs NVTX [55] to mark interesting functions and their invocation relations. In addition, it stores each function’s parameters and call stacks as a JSON string and processes the information offline. DLProf relies on Nsight Systems to collect GPU performance metrics. As a result, DLProf records each GPU activity without runtime aggregation, leading to poor profiling scalability for applications running on GPU clusters.

Inspired by profiling interfaces such as OMPT [105] and OMPI [117], we designed a uniform profiling interface for deep learning frameworks. We implemented prototype support for this interface in PyTorch to collect essential information.

Objectives

We name our interface DLPT, which stands for Deep Learning Profiling Tool Interface. DLPT is a first-class interface for deep learning frameworks that enables performance tools to collect application level information and attribute measurement data to insightful program contexts.

Figure 3.16 shows that DLPT is a layer between applications and their runtimes. In practice, DLPT can be implemented using function interception, source code instrumentation, or application callbacks. Since DLPT is aware of operations and states of applications, it can provide unique information for tools not available through low-
Figure 3.16: DLPT provides application level information to performance tools.

level monitoring APIs. At the initialization stage, tools can use an interface to register
themselves and select what features to monitor. Then, at deep learning operator en-
tries/exits, a deep learning framework will invoke callbacks provided by the interface
to send program information to performance tools. Finally, a deep learning framework
should terminate tools through a termination function before the execution ends. The
following paragraphs list essential information that deep learning frameworks should
provide when integrating DLPT.

**Insightful calling contexts** Previous sections described techniques that recover
calling contexts through call stack unwinding and reconstruction. Though function
names and line information in traditional calling contexts are helpful for HPC appli-
cations, they do not indicate important context for deep learning applications. Deep
learning applications are often written in scripting and compiled languages to achieve
programming flexibility and high performance. When performing stack unwinding,
there may be many stack frames, and the frames of the scripting language do not
show meaningful invocation relations. Also, call paths generated through unwinding
do not provide domain-specific information. Users are interested in the execution phase (e.g., backward, forward, or data loading) at a high level. Next, users want to explore an iteration within the current phase, the deep learning operator in the current iteration, and functions called by the operator. The aforementioned information cannot be simply obtained by interpreting line information and function names. Our proposal specifies what entry functions are essential and provides callbacks to these functions so that tools can construct insight calling contexts.

**Association of fine-grained metrics and runtime** After collecting meaningful calling contexts, a profiler can associate it with performance metrics other than the execution time, using interface functions defined by DLPT. Previously, we mentioned that our profiler employs correlation IDs: when a GPU API is invoked, the profiler assigns a unique ID to the invocation and the call stack where this invocation occurs. We can extend this mechanism to the deep learning profiling tool interface.

In addition to maintaining an association between GPU metrics and program context, it is also crucial to associate asynchronous CPU events to their corresponding calling contexts. It is common to observe that bottlenecks are from the CPU rather than the GPU in production workloads. For instance, reading data from disk to memory can be expensive for many deep learning workloads. By attributing both CPU and GPU metrics to the same program context, our tool can identify whether GPU is idle by comparing the CPU and GPU times associated with a context.

**Tensor information** Deep learning frameworks organize memory as tensors. Each tensor has several attributes upon its allocation, including the shape, the data type, and the allocation context. Knowing when and where tensors are allocated and freed is important to optimize GPU memory usage. Because deep learning frameworks man-
age tensors using a custom GPU memory pool, general GPU profiling tools cannot capture tensor allocation and free events through a low-level profiling API. TensorFlow instruments each operator in its framework to closely monitor operations on each tensor. To standardize monitoring of tensors, the proposal should specify what operations should have callbacks to a performance tool and to inform the tool about tensor operations. In addition, a tensor can be reshaped during its lifetime; knowing the meaning of each dimension is also helpful. For instance, an NCHW format tensor can be reshaped to NHWC to improve GPU memory access efficiency. Currently, only DLProf can provide such information.

The Design

![Figure 3.17: Major components of DLPT](image)

As shown in Figure 3.17, DLPT provides performance tools with callback and state APIs. Performance tools register callbacks via DLPT to obtain operator information entries and exits, as well as memory information at memory allocations and frees. The callback API can be used for exhaustively tracing every function. Unlike callback API, performance tools can query states of deep learning models, call
paths, and worker threads at any time using the state API. Hence, the state API can be used together with asynchronous sampling to achieve low overhead. Since the design intends to define interfaces only for performance tool developers rather than deep learning application developers, all interface functions have a portable C binding only.

**Callback API**  
Listing 2 lists all domains that we have defined for deep learning frameworks. Users can selectively monitor domains or a combination of several. Selecting specific domains can lower the monitoring overhead in some situations. For instance, if a script only contains the inference phase, backward functions do not have to be monitored. User scope is a special domain that supports user-defined regions. Sometimes a user may want to monitor a region of code but not refactor it into a function. Our tool also supports monitoring tensor allocation/free, which grabs or releases memory to a dedicated memory pool that cannot be monitored through low-level APIs. Monitoring JIT information is also necessary. With operation fusion or strength reduction, the original operators must be replaced by other operators or eliminated from the script.

A deep learning framework’s runtime provides information about operators and memory objects to tools, as described in Listing 3. The information can be gleaned at operator entries or exits and memory allocation or free points, as described in
typedef struct dlpt_op_data {
    uint64_t forward_thread_id;
    int64_t sequence_number;
    // nested_level: 0->1->2
    // |
    // master
    uint32_t nesting_level;
    const char *name;
    void **ptrs;
} dlpt_op_data_t;

typedef struct dlpt_mem_data {
    void *ptr;
    const char *device;
    uint64_t size;
    uint64_t total_allocated;
    uint64_t total_reserved;
} dlpt_mem_data_t;

typedef struct dlpt_tensor_info {
    void *ptr;
    const char *name;
    const char *dtype;
    const char *layout;
} dlpt_tensor_info_t;

Listing 3: DLPT callback data structure

Listing 4. Each operator is associated with a forward_thread_id to identify which thread initiates the forward operator. Backward operators can be initiated from a different thread than the forward thread where its corresponding forward operator started. The sequence_number field records the launching sequence of the operator, the backward phase and the forward phase of the same operator share the same sequence_number. An operator can be implemented by other operators; in this case, some tools may want to maintain only the top frames but not nested frames to curtail the unwinding cost. We also pass the name of each operator because compiler-
Listing 4: DLPT monitoring events

generated names may not be as clear as the operator name provided by the framework.

The runtime should also provide information about when a piece of data is released or freed to the memory pool. Total allocated and reserved memory are provided to denote how much memory has been used and how much is currently retained by the memory pool. The `device` field is used to indicate on which device the data is allocated. The `dlpt_mem_data` struct has a `ptr` field, which corresponds to each value in the `ptrs` field in the `dlpt_op_data` struct. An `ptr` value represents the start address of a currently alive data object. `dlpt_op_data` records all data objects passed to the operation. If a data object represents a tensor, we can look up its related information about name, data type, and layout in a `dlpt_tensor_data` struct.

Listing 5: DLPT monitoring states

State API  Listing 5 specifies five states for CPU threads and the model being trained in deep learning frameworks. For CPU threads, a state object returned to the tool can be composed of multiple values. Forward, backward, and optimizer states
are persistent as a CPU thread is assigned one or more of them as long as it involves
the corresponding workload. Active and idle states are transient. Before entering an
operator, the runtime should mark the current thread’s state as active; after exiting
an operator, the runtime should mark the current thread’s state as idle. In most
cases, a model can only be assigned one of the states because forward, backward, and
optimizer phases usually occur in sequence.

```c
typedef struct dlpt_context {
    const char *file_name;
    const char *function_name;
    size_t function_lineno;
    size_t lineno;
} dlpt_context_t;
```

Listing 6: DLPT calling context data structure

For frameworks that use interpreted languages, it is necessary for the runtime to
refine call paths but not let the tool use unwinding APIs. For example, if Python is
used as the front-end language, its frames cannot be directly obtained by C unwinding
APIs. Even using Python’s frame unwinding may recover long call chains that provide
little insight. Thus, it will be more flexible to enable a runtime to curate a clean
call path for users instead of having tools employ special code for each framework.
Currently, we specify that each frame should return a sequence of function names, line
number of calls, first line of functions, and file names, included in the dlpt_context
struct.

**Implementation and Use Cases**

We implemented DLPT as a monitoring library for PyTorch using its C++ APIs. Our
code does not modify any PyTorch source code and is compatible with precompiled
PyTorch libraries. Callbacks are triggered when PyTorch calls specific operations. To monitor deep learning operators, we registered our monitoring function using the \texttt{aten::addGlobalCallback} facility, which is also used by PyTorch’s internal profilers. We are able to get all of the information needed to fill the \texttt{dlpt\_op\_data} struct at every operator’s enter and exit. To monitor data object allocation and free, we registered a debug object through \texttt{c10::ThreadLocalDebugInfo} to fill fields in the \texttt{dlpt\_mem\_data} struct.

Our implementation also supports tracking the state of a deep learning framework. We allow users to explicitly obtain a fine-tuned Python call path at each operator callback. First, we request a Python global interpreter lock (GIL) since the Python interpreter is not fully thread-safe. Then, we iteratively get the current thread’s frame through \texttt{PyEval\_GetFrame}. For every frame, we unpack its function name, line number, and file name, and put all fields into a \texttt{dlpt\_context} object.

In the following sections, we describe two use cases that demonstrate new insights and views provided by our tool after integrating DLPT.

**Calling Context Refinement** PyTorch is implemented using Python at a high level with underlying deep learning computations accelerated by C/C++. As mentioned before, without DLPT, a performance tool can at best obtain native call paths with lots of frames inside the Python interpreter. However, these frames do not provide users with helpful information to gain insights. We show an example profile view for the MNIST [118] convolution neural network in Figure 3.18 using our profiler without DLPT. There are three problems with this view. First, only low-level calling contexts are captured. One cannot readily determine what deep learning
operators were executed based on this view. Second, backward functions are presented under “thread root”, while forward functions are presented under “program root”. Without associating backward functions with their forward counterparts, we cannot understand where these backward functions are called in the user-level code. Third, the call path for each GPU operation is very long, consisting of more than 30 frames. Unwinding long call paths can cause a high overhead at runtime. Because DLPT provides the callback API to track each deep learning operator’s entry and exit and the state API to query each CPU thread’s call path, we propose the following modifications to raw call paths, shown in Figure 3.19.

The first modification associates a forward function’s call path with its corresponding backward function. For each deep learning operator, DLPT associates it with a unique sequence number shared by its backward and forward functions. In the forward phase, we record mappings between a sequence number and its forward
function’s call path. In the backward phase, we query a sequence number’s call path for each backward function. Then, we copy the call path and use it as the call path of the backward function, attaching a backward placeholder at the end of the call path to differentiate it from the forward function.

The second modification is logical unwinding [119], which maps low-level contexts to high-level contexts. A raw call path consists of different contexts, including tool, driver, runtime, and Python contexts. We eliminate tool and driver contexts from the raw call path because users usually do not have access to proprietary driver code, and tool contexts do not exist in applications. Runtime contexts include a chain of sophisticated function invocations. While it might be interesting to show these functions to framework developers, application developers care more about application-level code. As a result, we compress runtime contexts by replacing them with a deep learning operator name (e.g., `aten::conv2d`). Last, for each deep learning operator, we unwind at function callbacks when `nesting_level` is 0; subsequent functions belong to the same operator with `nesting_level > 0` use the same call path.
Figure 3.20 illustrates the calling context of MNINST after applying our call path modifications. In this profile view, we show a full Python call path from the root to where the runtime is entered. We collect this operator’s name from DLPT to replace runtime contexts. At a frame below, we identify the operator’s execution phases and associate each forward function call path with its corresponding backward function. We attach GPU kernels to the end of the call path.

**Measurement of Framework Overhead** In PyTorch, the real computation of each operator is invoked through a sequence of wrappers. For instance, Figure 3.21, four extra functions are invoked before calling the underlying convolution implementation. In DLPT, the first function invoked by each operator has nesting_level = 0, and subsequent functions have nesting_level > 0. We denote the time spent between each intermediate function’s entry points and each function’s exit points as “dispatch time”. In our study, we profiled the total amount of dispatch time spent
in the ResNet-50 [120] model on an NVIDIA A100 GPU. The original end-to-end execution time of the model was 3.9s. Using our profiler, the execution time was increased to 4.0s, indicating that the profiler caused only 3% overhead. We found that the total dispatch time in PyTorch is 0.51s, which means that 13% time is spent in preparing data for GPU offloading and finding the right GPU kernel. If a compiler can exhaustively inline dispatch calls, it might reduce the dispatch time and provide lower latency for some important workloads.

3.7 Overhead and Accuracy

This section describes the evaluation results of our tool's coarse- and fine-grained profiling attributions. In the coarse-grained profiling mode, we profile metrics associated with each GPU operation. In the fine-grained profiling mode, we collect instruction samples associated with each GPU kernel. In both modes, we attribute metrics to call paths where they are incurred on the CPU. Our evaluation platform is a single node Linux machine equipped with a 512 GB main memory, two AMD EPYC 7402 24 core processors, and an NVIDIA A100 GPU with a 40GB HBM. The host
system is running Linux kernel 4.18, CUDA driver 510.39, and CUDA Toolkit 11.6. We compared our tool’s accuracy and overhead with HPCToolkit [87] and Nsight Systems [15]—the two state-of-the-art context-sensitive GPU profilers on nine HPC and machine learning applications listed below:

- **Laghos** [30] is a proxy application that solves the time-dependent Euler equations. We used Laghos’ CUDA implementation with a representative input "-p 0 -dim 2 -rs 3 -tf 0.75 -pa".

- **NAMD** [79] is a molecular dynamics application accelerated by GPUs. We used NAMD’s CUDA backend with its default `apoa.namd` input.

- **LAMMPS** [25] is a parallel atomic/molecular Simulator. We used LAMMPS’ Kokkos backend and its `in.intel.lj` input.

- **PeleC** [81], **Nyx** [121], **AMR-Wind** [122], **WarpX** [123] are adaptive mesh refinement applications written in AMReX [85]. We studied PeleC’s PMF example, Nyx’s LyA example, AMR-Wind’s `abl_mol.i` input, and WarpX’s beam-driven electron acceleration input.

- **PyTorch** [7] is a popular deep learning library. We studied the ResNet-50 model [120] implemented using pyTorch.

- **PyTorch-Geometric (PyG)** [124] is a graph neural network library built upon PyTorch. We studied PyG’s heterogeneous graph transformer (HGT) example [125].

We compiled applications using GCC 8.3.1 and CUDA Toolkit 11.6 with compilation flags "-g -lineinfo -03 -arch sm_80" to generate optimized binaries with line map information.
Figure 3.22: Overhead comparison between tools. Our tool uses unwinding probability $U = 1/4$.

### 3.7.1 Coarse-grained Profiling

In the coarse-grained profiling mode, we compared our tool’s overhead with Nsight Systems and HPCToolkit. Our tool focuses on reducing the overhead caused by call path unwinding, which is significant for many applications. Other overhead, such as the overhead caused by CUPTI is out of scope in this study. For HPCToolkit, we instrumented its unwinding component to accumulate time spent on call path unwinding. For Nsight Systems, we first measured an application using `--cudabacktrace=true` to get the time $t_p$ with call path association, and we measured again to get another measurement time $t$ without call path association. Then, we used $t_p - t$ to represent the unwinding time for Nsight Systems. For each application, we used its internal timer as the execution time to skip the start and end time caused by profilers, if any.
Table 3.1: Call path statistics of applications

<table>
<thead>
<tr>
<th>Application</th>
<th>#GPU Operations/s</th>
<th>#Unique Call Paths</th>
<th>Average Call Path Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAMD</td>
<td>10105.66</td>
<td>156</td>
<td>9.88</td>
</tr>
<tr>
<td>Laghos</td>
<td>207854.50</td>
<td>321</td>
<td>11.75</td>
</tr>
<tr>
<td>PeleC</td>
<td>43158.52</td>
<td>611</td>
<td>13.84</td>
</tr>
<tr>
<td>Nyx</td>
<td>83486.61</td>
<td>734</td>
<td>15.85</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>8278.78</td>
<td>152</td>
<td>12.93</td>
</tr>
<tr>
<td>AMRWind</td>
<td>88165.29</td>
<td>1659</td>
<td>15.20</td>
</tr>
<tr>
<td>WarpX</td>
<td>152886.40</td>
<td>737</td>
<td>14.04</td>
</tr>
<tr>
<td>PyTorch</td>
<td>42275.17</td>
<td>32</td>
<td>57.00</td>
</tr>
<tr>
<td>PyG</td>
<td>48651.74</td>
<td>152</td>
<td>38.46</td>
</tr>
</tbody>
</table>

From Figure 3.22, we observe that our tool’s overhead is remarkably lower than the others in most applications. Our tool’s average overhead is $1.42 \times$, compared to that of $2.08 \times$ and $2.76 \times$ for HPCToolkit and Nsight Systems. The call path unwinding overhead for our tool is on average 1.81%, ranging from 0.20% to 4%. In contrast, the unwinding overhead for HPCToolkit and Nsight Systems are 65% and 135%, respectively. The results show that our memoization mechanism significantly reduces overhead for call path unwinding: the end-to-end overhead for coarse-grained profiling is no longer caused by call path unwinding. In our experiments, most of the remaining overhead is caused by CUPTI, which is beyond the control of third-party tool researchers.

Table 3.1 summarizes the total number of unique call paths, GPU operation frequency, and the average call path depth of each application. NAMD and LAMMPS’s unwinding overhead are low because of their low GPU operation frequency; we observed that while they utilize the GPU well, their GPU kernels tend to run a long time, causing lower GPU operation frequency. For other applications, high GPU operation frequency and deep call paths are two major contributors to unwinding overhead. For instance, Laghos invokes 207854 GPU operations per second in which
the average call paths depth is 12, resulting in more than two million frames to unwind per second. HPCToolkit and Nsight Systems cause 1.68× and 1.28× overhead for Laghos, correspondingly. It is also worth noting that even without unwinding, CUPTI can bring 1.5× overhead to Laghos. PyG and PyTorch’s unwinding overhead is caused by their very deep call paths consisting of Python and C++ frames. Our tool’s unwinding overhead is negligible even in those extreme cases.

Figure 3.23 shows the accuracy of the call path memoization mechanism using different unwinding probabilities. We calculate the accuracy using the ratio of correct call paths using memoization. The ground truth is obtained by doing a full unwind at every GPU operation. The memoization mechanism yields an average accuracy of 99.93% among all applications. Even for applications with many unique call paths, such as Nyx, AMR-Wind, and WarpX, the call path memoization mechanism’s accu-
racy is still 99.0%. From the evaluation results, we observe the accuracy is reduced with the decrease of unwinding probability. In practice, we can use $U = 1/4$ to achieve both high accuracy and low overhead.

### 3.7.2 Fine-grained Profiling

Table 3.2: Comparisons of fine-grained profiling overhead between our tool against HPCToolkit. HPCToolkit’s overhead is used as the baseline to compare with our methods.

<table>
<thead>
<tr>
<th>Application</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HPCToolkit</td>
</tr>
<tr>
<td>NAMD</td>
<td>10.24×</td>
</tr>
<tr>
<td>Laghos</td>
<td>53.53×</td>
</tr>
<tr>
<td>PeleC</td>
<td>25.41×</td>
</tr>
<tr>
<td>Nyx</td>
<td>35.98×</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>7.50×</td>
</tr>
<tr>
<td>AMRWind</td>
<td>24.29×</td>
</tr>
<tr>
<td>WarpX</td>
<td>49.31×</td>
</tr>
<tr>
<td>PyTorch</td>
<td>18.99×</td>
</tr>
<tr>
<td>Average</td>
<td>27.51×</td>
</tr>
</tbody>
</table>

In the fine-grained profiling mode, we set up instruction sampling’s period as 1024, meaning that a sample is taken from each streaming multiprocessor in a GPU every 1024 cycles. We measured the overhead and accuracy of each fine-grained profiling method and compared them with the results obtained from HPCToolkit Table 3.2 and Table 3.3, accordingly. The fixed range approach uses a range size of 16 and 128 for the small and the large range. The adaptive range approach uses a range sampling enabling rate of 6% and 0.8% for the high and low frequencies.

We devised Equation 3.1 to calculate the accuracy of instruction sampling of each
method. \( R_{p_i}^{method} \) denotes the percentage of instruction samples attributed at a call path \( p_i \) using a profiling method. \( P \) refers to a set that contains all call paths we collected at runtime. Therefore, this equation calculates the sum of the percentage differences between profiles collected using a specific method and the serialized method.

\[
\sum_{p_i \in P} |R_{p_i}^{method} - R_{p_i}^{serialized}|
\]  

(3.1)

Table 3.3: Our tool’s fine-grained profiling accuracy. We do not compare HPC-Toolkit’s accuracy because it uses the serialized mode to collect instruction samples, yielding an exact correlation between instruction samples and call paths. We use bold fonts to denote the highest accuracy in each application.

<table>
<thead>
<tr>
<th>Application</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fixed Range-</td>
</tr>
<tr>
<td></td>
<td>Small</td>
</tr>
<tr>
<td>NAMD</td>
<td>98.81</td>
</tr>
<tr>
<td>Laghos</td>
<td>97.43</td>
</tr>
<tr>
<td>PeleC</td>
<td>91.55</td>
</tr>
<tr>
<td>Nyx</td>
<td>95.69</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>99.05</td>
</tr>
<tr>
<td>AMRWind</td>
<td>98.72</td>
</tr>
<tr>
<td>WarpX</td>
<td>95.85</td>
</tr>
<tr>
<td>PyTorch</td>
<td>97.20</td>
</tr>
<tr>
<td>PyG</td>
<td>81.02</td>
</tr>
</tbody>
</table>

From Table 3.3, we noticed that the adaptive range profiling method with high frequency yields the highest accuracy in six out of nine cases. Besides, it reduces the average instruction profiling overhead from 27.51× to 5.31× compared to the serialized mode. Using a low enabling rate can further reduce the overhead to 2.84× but also lowers the accuracy to 92.04%. Note that the fixed sampling profiling approaches yield a low accuracy for the PyG example. After analyzing the profiling result for
PyG, we found several kernels whose instruction samples are not proportional to the kernel counts. For example, the `indexSelectLargeIndex` kernel incurs 9.5% instruction samples. This kernel has three different call paths, taking 0.8%, 4.3%, and 4.3% instruction samples, respectively. However, all three call paths have the same invocation counts; this causes the fixed range apportioning method to evenly attribute 3.2% samples to each call path, causing 4.6% error for the kernel.

We also evaluated the combined effects of the call path memoization mechanism and the adaptive range profiling method. Figure 3.24 shows the overhead with and without call path memoization. On average, call path memoization reduces the overhead from 5.31× to 4.61×. We didn’t observe notable accuracy loss by combining two optimizations. Thus, in total, our adaptive range profiling method is 5.96× faster than the serialized mode (i.e., HPCToolkit) with an accuracy of 96.16%.

Figure 3.25 compares the memory consumption between HPCToolkit, the fixed
Figure 3.25: CPU memory consumption comparison between different profiling methods

range profile method (small range), and the adaptive range profiling method (high frequency) with and without trie-based range aggregation. Since HPCToolkit uses the serialized profiling method, instruction samples are directly attributed to their initiated call paths without maintaining a range calling context tree. Range sampling used by adaptive range profiling methods can reduce the average memory consumption from 8.88GB to 5.05GB, compared to fixed range profiling. Range aggregation further reduces the average memory consumption to 3.14GB, which is close to the 2.84GB memory consumption of the serialized mode (i.e., HPCToolkit), resulting in a total reduction of $2.83 \times$. Applications running for a long time with a large number of GPU kernels, such as Nyx and AMRWind, benefit significantly from range sampling and aggregation.
Comparison with Nsight Compute  Nsight Compute measures instruction samples using the serialized mode. Unlike our tool and HPCToolkit, it performs performance analysis immediately after every kernel and does not aggregate metrics across GPU kernel invocations. For this reason, it introduces prohibitive overhead to profile instruction samples on large applications. With Nsight Compute, we collected the smsp_pcsamp_sample_count metric and turned off kernel replay, but it still slowed down the running time for Laghos by $14400 \times$. Thus, Nsight Compute is not comparable to our tool. Users have to specify the kernels to monitor and adjust the number of invocations before using Nsight Compute.

3.8 Summary

To provide rich insights for users, our tool attributes GPU metrics to full calling contexts. To the best of our knowledge, existing tools provide, at best, only partial call paths for GPU operations with high overhead. On one hand, attributing metrics to call paths enables actionable optimizations; on the other hand, it also imposes new challenges.

First, storing call paths and metrics for GPU operations can significantly increase the memory overhead. Based on our evaluation, Nsight Systems often dumps a database of tens of Gigabytes in just a few minutes. Thus, it is by no means scalable for long program executions on many nodes. Our tool overcomes these challenges by building a calling context tree that collapses identical call paths into a calling context tree. In addition, we designed a sparse representation [126] in which each node only stores non-zero metrics. Currently, our design can reduce the size of profiles significantly if only a profile view is needed. In the future, we can explore other approaches [127] to reduce the size of trace files.
Another challenge brought by calling context attribution is frequent communication and unwinding to attribute GPU metrics to CPU threads where they are initiated. We employed efficient wait-free data structures to communicate GPU activity records back to CPU calling contexts. In addition, we reduced the frequent call path unwinding overhead using unique call path identifiers. Experimental results demonstrated that our methods can achieve low overhead and high accuracy. Our framework assumes that there’s at most one GPU monitor thread for collecting GPU metrics. However, it is possible that the GPU measurement substrate, such as LLVM’s support for OMPT in libomptarget [128], spawns multiple GPU monitor threads at runtime. Our wait-free communication channels are suited to such cases as it is designed for single-consumer and single-producer communication only. We solved this problem by creating an additional operation thread that gathers invocation records and GPU activities and dispatches them to the corresponding threads. Because a single thread is responsible for handling data from multiple threads, our solution can induce a burden on the operation thread in certain cases. We can refactor the communication channel using wait-free queues [129] suitable for multiple consumers and producers, eliminating the operation thread.

This chapter describes the deep learning profiling interface implemented for PyTorch. We can extend the interface as appropriate to create an API standard and implement it in other deep learning frameworks. The profiling interface can benefit third-party tools by enabling them to obtain detailed information from deep learning frameworks. Different from Tensorboard’s graph-centric profiling, our profiling interface is rather code-centric, which is more straightforward for performance tuning. Our tool still hasn’t supported several advanced features yet, including monitoring of data loaders, distributed models, and attribution to intermediate representations.
Also, we haven’t looked into deep learning frameworks written in modern languages such as Julia [130] and JAX [131].
Chapter 4

GPU Binary Analysis

Our tool’s profiler attributes performance metrics to the CPU call path of each GPU operation at runtime. Instruction samples and other fine-grained metrics are attributed to each GPU kernel without a hierarchy of program contexts that include device functions, inlined functions, and loops. We implemented a static analyzer that analyzes GPU binaries to provide necessary information about program structure, instruction dependencies, and GPU architectural features to refine fine-grained attribution. This chapter elaborates on the program analysis for analyzing GPU binaries, which is very different from analyzing CPU binaries.

4.1 Background and Related Work

This section reviews the format of GPU binaries and GPU instructions to provide essential background to understand the challenges and innovations in our static analyzer.

4.1.1 Binary Format

Our tool analyzes ELF binaries on Linux systems. ELF stands for Executable and Linkable Format [132], which is the standard format for executables, object code, shared libraries, and core dumps. GPU binaries for AMD, Intel, and NVIDIA GPUs all follow the ELF format, though Intel and NVIDIA’s binaries do not use standard ELF. An ELF file consists of an ELF header and file data. Using the `readelf`
command, we can interpret the header and other data of an ELF file, including GPU binaries. ELF files typically consist of three parts beyond the header, including program headers, section headers, and data.

A *program header* section consists of data segments that describe how the Linux kernel creates memory for a process in the virtual address space. *eh_frame* is a data segment that follows DWARF format [133]. Using the Debugging Information Entries (DIEs) in DWARF, we can access information about what registers to restore for previous function frames. Therefore, *eh_frames* can be used to unwind the call stack by recovering frame-relevant registers including stack pointer and base pointer of caller and return address.

A *section header* defines sections in the file used for linking and relocation. For executable files, there are four main sections, including *text*, *data*, *rodata*, and *bss*. Instructions are in the text section.

Binaries of GPU-accelerated applications are rather unusual. GPU binaries can be pre-compiled and embedded in a host binary or JIT-compiled at runtime. Furthermore, NVIDIA and Intel GPU binaries are unlike any CPU binaries we have ever seen. While each of the GPU vendors relies in part on ELF representations for GPU code, their formats for GPU binaries are quite different. We describe how our tool handles these complexities of GPU binaries in Section 4.2.

### 4.1.2 Instruction Format

Our tool analyzes GPU instructions to obtain necessary information for subsequent performance analysis. We elaborate on the instruction encoding format for NVIDIA GPUs because many details can only be reverse engineered and are not documented well in other papers. AMD and Intel ISAs are well documented in their open source
A fixed length instruction encoding is used on NVIDIA’s GPUs. Pre-Volta GPUs use a 64-bit word for an instruction, but Volta and later architectures use a 128-bit word. Among the fields of a GPU instruction shown in Table 4.1, we focus on the following three key fields:

- **Wait Mask and Write/Read Barrier.** Every GPU instruction has a control code [136, 137] field that encodes information to guide the warp scheduler as it issues instructions, including stall cycles, yielding flag, and dependencies. For each fixed latency instruction (e.g., most arithmetic instructions), the assembler sets stall cycles for the instruction to indicate how long the scheduler should wait before issuing the instruction. For each variable latency instruction, the assembler associates write/read barrier indices with it and includes a wait mask for these indices in instructions that depend on them to create dependencies. In recent GPU architectures, barrier instructions (e.g., LDGDEPBAR) can be used to decouple barrier dependency from register dependency [138], yielding more flexible instruction scheduling.

- **Predicate.** If an instruction’s predicate field is set, the instruction is executed when the predicate evaluates as true. There are both true and false predicate conditions: $P_i$ is a true predicate condition, and $!P_i$ is a false predicate condition, where $0 \leq i \leq 6$. In Table, the LDG instruction is executed if $P_0$ is true.

<table>
<thead>
<tr>
<th>Wait Mask</th>
<th>Write Barrier</th>
<th>Predicate</th>
<th>Opcode</th>
<th>Modifiers</th>
<th>Destination Operands</th>
<th>Source Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>B1</td>
<td>P0</td>
<td>LDG</td>
<td>32</td>
<td>R0</td>
<td>R2, R3</td>
</tr>
</tbody>
</table>

Table 4.1: Dissection of the fields of “@P0 LDG.32 R0, [R2]” instruction.
• **Opcode, Modifiers, and Operands.** Each thread can use up to 255 32-bit regular registers ranging from \( R_0 \) to \( R_{254} \). In addition, threads have access to uniform data path registers that store the same values among threads within the same warp. Opcode and modifiers together determine the length of operands used. In Table 4.1, the 32 modifier indicates each thread reads a 32-bit value from memory. Moreover, because data are loaded from global memory, which has a 64-bit address space, the source operand is a 64-bit value comprised of two registers—\( R_2 \) and \( R_3 \).

### 4.2 Program Structure Recovery

Relating performance metrics associated with a node in a calling context tree to source code requires understanding the mapping from machine instructions to source lines, loops, functions, and inlined code. Our tool analyzes GPU binaries to determine this mapping. This strategy works for any statically-compiled language, including C, C++, and Fortran.

NVIDIA’s GPU binaries sometimes have device functions embedded inside a global GPU function. Each function is in a separate text segment, and all functions start with offset 0 in their symbol table. To aid measurement and analysis, we relocate each function to a unique address—its offset in the binary’s section table. We split any overlapping GPU functions into disjoint address ranges. Then we use NVIDIA’s `nvdisasm` binary tool [139] to analyze GPU machine code and dump a control flow graph for each function in a binary. Because `nvdisasm` renders merged basic blocks in control flow graphs, we split superblocks into basic blocks to fit analysis.

Intel’s GPU binaries provided in an archive-like format in which each member contains information about a single GPU kernel. At runtime, we dump such binaries
into the file system. During post-mortem analysis, our tool reads a fat binary, iterates through all kernels, and processes the ELF binary for each kernel separately. We use Intel’s IGA API [140] to decode each kernel binary to obtain its control flow graph.

AMD’s GPU binaries are also embedded in a “fat” host binary. The locations of AMD’s GPU binaries inside the fat binary are specified by Uniform Resource Identifiers (URIs) reported by ROCm debug API and ROCTracer. AMD’s GPU binary uses a conventional ELF representation. We use the Dyninst [58] binary analysis framework to decode AMD GPU instructions and construct control flow graphs. The Dyninst team has developed preliminary support for analyzing AMD GPU binaries, including support for decoding scalar instructions and resolving direct and indirect control flow.

Control flow graphs extracted from Intel and NVIDIA GPU binaries are converted to a uniform representation and injected into Dyninst, which analyzes loop nests they contain. We classify GPU instructions by their operation types, instruction length, and the kind of memory they access. As Dyninst can analyze AMD GPU binaries, the analyses are performed directly in Dyninst.

We also use Dyninst to read DWARF sections in a GPU binary to attribute instructions to source lines and recover inlined functions. Intel GPU binaries do not provide information about the call site of an inlined function, even when they are compiled with debug information.

If our profiler records many GPU binaries, our static analyzer will invoke multiple processes to analyze these GPU binaries in parallel. Moreover, if a GPU binary contains thousands of GPU functions, we will invoke fine-grained parsing utilities (e.g., `nvdisasm`) in parallel. Evaluated on Laghos, the multi-level parallelism strategy can reduce the analysis time from an hour to a few minutes using 32 CPU threads.
Algorithm 4 Algorithm for calling context construction

1: function ConstructCallingContext(symbols, samples)
2:   \( \mathcal{G} \leftarrow \text{ConstructCallGraph}(\text{symbols, samples}) \)
3:   \( \text{SCCs} \leftarrow \text{IdentifySCC}(\mathcal{G}) \)
4:   \( \mathcal{G} \leftarrow \text{ConstructSCCNodes}(\mathcal{G}, \text{SCCs}) \)
5:   \( \mathcal{G} \leftarrow \text{PropagateCallGraph}(\mathcal{G}) \)
6:   root \leftarrow \text{find } v \in \mathcal{G}.V \text{ with indgree}(v) = 0 \)
7: return SplitCallGraph(\mathcal{G}, \text{root, 1.0})

4.3 GPU Calling Context Trees

This section describes an approach that reconstructs a fine-grained GPU calling context trees using instruction samples, which is currently available for NVIDIA GPUs only. In addition, with little modification, this approach can be used to reconstruct calling context trees using instruction counts collected with binary instrumentation. NVIDIA’s CUPTI API returns flat instruction samples without any information about call paths on the GPU. With complex code generated from higher-level GPU programming models, we need calling contexts on GPUs to understand the code and its performance. Currently, no API is available for efficiently unwinding call paths on NVIDIA’s GPUs. We designed a method to reconstruct approximate GPU calling contexts offline to address this issue.

Like Gprof [141], we assume that every call to a function takes the same amount of time. While Gprof counts calls exactly, we calculate approximate call counts based on instruction samples of call instructions. Algorithm 4 presents a sketch of ConstructCallingContext method, which takes two input arguments–function symbols in GPU binaries and instruction samples of each GPU kernel calling context. At Line 2, we call ConstructCallGraph to construct a call graph \( \mathcal{G} = \{V, E\} \), where each vertex \( v \in V \) is a function, and each edge \( e \in E \) is a call relation. Algorithm 5 describes the steps to create a call graph in detail. For each function symbol (Line 3), we create a
Algorithm 5 Algorithm for call graph construction

1: function \texttt{ConstructCallGraph}(symbols, samples)
2: \( G \leftarrow \) create an empty Graph
3: for \( s \) in symbols do
4: \( v \leftarrow \) create a vertex with address range \([s\.start, s\.end]\)
5: add \( v \) to \( G\.V \)
6: for \( v \) in \( G\.V \) do
7: for instruction \( i \) in address range \([v\.start, v\.end]\) do
8: \( i\.weight \leftarrow 0 \)
9: for sample \( p \) in samples with \( p\.address = i\.address \) do
10: \( i\.weight \leftarrow i\.weight + p\.count \)
11: if \( i \) is \texttt{Call} then
12: \( e \leftarrow \) create an edge
13: \( e \leftarrow (.weight \leftarrow i\.weight, .source \leftarrow i\.v, .target \leftarrow i\.target) \)
14: add \( e \) to \( e\.source\.outgoing \)
15: add \( e \) to \( e\.target\.incoming \)
16: add \( e \) to \( G\.E \)
17: \( v\.weight \leftarrow v\.weight + i\.weight \)
18: return \( G \)

vertex with the symbol’s start and end addresses and add the vertex to \( G \). Then, for each vertex, we iterate over instructions within its address range (Line 7). If we find an instruction sample has the same address as an instruction (Line 9), we assign the sample’s count to the instruction and increase the function’s total weight (Line 17). If the instruction is a \texttt{Call} (Line 11), we add an edge with \texttt{source} vertex, \texttt{target} vertex, and the instruction’s \texttt{weight} to \( G \).

The \texttt{ConstructCallingContext} method also handles recursive calls. First, we identify strongly connected components (SCCs) using Tarjan’s algorithm [142] at Line 3. Next, we unlink edges between vertices within the SCC, and add an SCC vertex to enclose the set of vertices in each SCC at Line 4. Then, we treat an SCC vertex as a normal “function” in the call graph. In some cases, the leaf function of a long call path has samples, but no call instruction is sampled in the call path that reaches it. In Line 5, we propagate samples to all possible call sites and prune the call graph to include vertices only for functions that appear to have been called at runtime.
Algorithm 6 Propagate instruction samples on the call graph

1: function PropagateCallGraph($\mathcal{G}$)  
2:     $q$ ← create a FIFO queue with \{\(v \in \mathcal{G} | v.\text{weight} > 0\}\}  
3:     \textbf{while} $q$ is not \textbf{Empty} \textbf{do}  
4:         $v$ ← pop the first node from $q$  
5:         \textit{caller} ← \text{False}  
6:         \textbf{for} $e$ \textbf{in} $v$.\text{incoming} \textbf{with} $e.\text{weight} > 0$ \textbf{do}  
7:             \textit{caller} ← \text{True}  
8:         \textbf{if} \textit{caller} is \text{False} \textbf{then}  
9:             \textbf{for} $e$ \textbf{in} $v$.\text{incoming} \textbf{do}  
10:                 \textbf{if} $e.\text{source}.\text{weight} = 0$ \textbf{then}  
11:                     push $e.\text{source}$ to the end of $q$  
12:                     $e.\text{weight}$ ← 1  
13:                     $e.\text{source}.\text{weight}$ ← $e.\text{source}.\text{weight} + 1$  
14:         \textbf{for} $v$ \textbf{in} $\mathcal{G}.V$ \textbf{with} $v.\text{weight} = 0$ \textbf{do}  
15:             remove $v$ from $\mathcal{G}$  
16:     \textbf{return} $\mathcal{G}$

In Algorithm 6, \textit{PropagateCallGraph} iteratively updates pattern to update the call graph. Initially, vertices with positive weight are pushed into a queue (Line 2); one of them is popped at a time (Line 4). If we cannot find any incoming edge of $v$ has positive weight (Line 8), we equally apportion the costs of $v$ among its call sites by assigning each edge to $v$ a weight of one sample (Line 12). We repeat this process until we reach a fixed point where each vertex has at least one incoming edge with positive weight. The propagation ends in $O(|V| + |E|)$ time because a vertex is pushed into the queue at most once (Line 11), and each edge is accessed at most twice (Line 6 and Line 9). After the propagation terminates, we prune vertices with zero samples (Line 15).

In the last step, we build a calling context tree by splitting the call graph. As shown in Algorithm 7, Function \textit{splitGraph} uses a depth-first traversal on a call graph to construct a calling context tree. We begin traversing the call graph at its root which does not have any incoming edge (Line 6). In each invocation, we update the current root’s weight (Line 2) and its instructions’ weight (Line 4) by multiplying by the
Algorithm 7 Split a call graph into a calling context tree

1: function SplitCallGraph(G, root, factor)
2:     root.weight ← root.weight × factor
3:     for instruction i in address range [root.start, root.end] do
4:         i.weight ← i.weight × factor
5:     for e in root.outgoing do
6:         v ← clone e.target
7:         create a tree link between root and v
8:         sum ← sum e.target’s incoming edges’ weight
9:         f ← factor × e.weight / sum
10:        SplitCallGraph(G, v, f)
11:    return root

current apportion factor. For each outgoing edge of root, we clone a copy of the edge’s target vertex (Line 6) to v and create a tree link between root and v (Line 7). A new apportion factor f is created based both the current factor and the ratio of an edge’s weight to the sum of all incoming edges’ weight of a vertex (Line 9). In the end, splitGraph is applied for v (Line 10).

Reconstruction from instruction samples collecting using the serialized instruction sampling

As we mentioned in Chapter 3, instruction samples can be collected using either serialized or continuous instruction sampling. In the former mode, instruction samples are attributed to the exact CPU call paths where they are initiated.

Figure 4.1 shows the reconstruction process for a small synthetic example by applying our calling context reconstruction algorithm.

1. We first construct a draft call graph based on function symbols and call instructions. Each function is annotated with the number of sampled instructions within the its address range; each edge is annotated with the number of times its call instruction is sampled.
2. We assign the edge between A to B one call sample because B does not have a sampled call site. If executed instructions are collected, each edge representing a call that executed should have a non-zero weight.

3. We identify an SCC that contains D and E.

4. Finally, we apportion the number of samples of the SCC using ratios of weights of calls from each call site to the total weight of calls from all call sites.
Figure 4.2: Steps to reconstruct GPU calling context trees. Shapes with background are GPU instructions: circles denote call instructions, and rectangles denote other instructions.
Reconstruction from instruction samples collecting using the continuous instruction sampling In the continuous mode, our profiler attributes GPU instruction samples to call paths where they are initiated but does not generate the invocation relations between device functions within a GPU kernel. For complex kernels, it is necessary to construct GPU calling contexts to analyze device functions that can be called at multiple places. Prior research [32, 31, 143] addresses this issue by constructing a call graph and splitting it into calling context trees in the serialized mode. We extended the existing approach by addressing three new challenges in range-based profiling, including:

- Apportion instruction samples of a GPU kernel to multiple call paths within a range in the fixed range profiling.
- Apportion instruction samples of a GPU device function called from multiple GPU kernels within a range.
- Scale instruction samples of a GPU kernel that has been sampled in several ranges.

Figure 4.2 demonstrates steps to reconstruct GPU calling context trees. We first construct a static call graph by analyzing functions and call instructions in GPU binaries. Using the collected profile, we assign each edge in the graph the number of call instruction samples. For functions without any call counts that have instruction samples, we assign all of its incoming call edges one instruction sample. Unlike previous work, this call graph represents the invocation relationships for all kernels and device functions within each range (Figure 4.2a). For each range, we apportion the number of instruction samples in each function among its call sites using ratios of calls from each call site to the total number of calls from all call sites within the
range (Figure 4.2b). Next, for each range, we scale the number of samples within each kernel by multiplying the ratio of the total number of kernel invocations to the number of sampled kernel invocations (Figure 4.2c). Finally, for each GPU kernel, we aggregate its samples from all ranges in which it appears to calculate the kernel’s total samples during the execution.

4.4 GPU Instruction Analysis

Understanding instruction characteristics is essential for GPU performance analyses. Our instruction analysis substrate is available for Intel and NVIDIA GPUs. First, we use backward slicing to analyze every instruction’s def-use chain in the control flow graph. We attribute stalls on instruction samples to their root cause instructions using the instruction dependency graph. In addition, we derive the access kind of GPU memory instructions using available arithmetic instructions to facilitate value-related analysis.

4.4.1 Backward Slicing

![Instruction Example](image)

Figure 4.3: An example of a barrier register dependency.

We extended a traditional backward slicing algorithm for CPU instructions to analyze special GPU operands, barriers, and predicated instructions. We target intra function backward slicing [144] for GPU instructions because instructions in the same function cause most stalls. According to Table 4.1, several fields of a GPU instruction
impact instruction dependencies, including operands, barriers, and predicate. Backward slicing for operands of GPU instructions is like traditional backward slicing for CPU instructions, but barriers and predicates need special processing.

**Virtual barrier registers:** We represent six available barrier indices as six virtual barrier registers B0-B5. A write/read barrier index association can be represented as a write operation to one or more barrier registers. Likewise, we treat a wait mask association as a read of barrier registers. In this way, dependencies caused by barrier indices can be identified through def-use chains of the virtual barrier registers. It is worth noting that barriers can be set even if there is no dependency between regular registers. Take Figure 4.3 as an example, the LDG instruction loads a value to R0 and writes barrier B0, and the BRA instruction does not consume R0 but still reads B0. Observed memory dependency stalls on the BRA instruction should be attributed to the LDG instruction. With such a dependence, the assembler can terminate waiting for B0 early and reuse B0 for the next basic block in the control flow.

**Predicated instructions:** Immediate dependency sources are not only the first def instruction of each of its operands on the search path. Consider Figure 4.4 as an example, suppose we observe a stall at the IADD instruction, which does not have a predicate; because the LDG instruction is executed only if P0 is true, it is possible that the stall comes from the LDC instruction earlier in the path, which is executed only if P0 is false. Therefore, backward slicing search should proceed until the predicates of def instructions on a path cover all conditions.

Let $P$ be the union of def instructions’ predicates on the path. $P = \cup p$, where $p \in \{p_i\} \cup \{!p_i\} \cup \{\_\}$, and $\{p_i\} \cup \{!p_i\} = \{\_\}$, for $0 \leq i \leq 6$. $\_$ is a special predicate that covers both true and false predicates. An instruction without a predicate has the same semantic as $\_$. We say $P$ contains $p'$ iff $p' \in P$ or $\_ \in P$. Our backward
Figure 4.4: An example of blaming stalls on the IADD instruction in the sampling mode and the hybrid mode.
slicing search proceeds until the union of $def$ instructions’ predicates on the search path $P$ contains the predicate of the $use$ instruction $(p')$.

From the def-use chains of a GPU binary, we build a *static instruction dependency graph*, in which each node represents an instruction, and each edge represents a def-use relation associated with $R0$. In practice, GPU binaries for some applications contain a large number of functions and blocks, which can make backward slicing costly. For example, NAMD [79] contains 68 GPU binaries, of which the largest is 89 MB with 4492 functions. Using a single CPU thread, our tool takes 12 hours to analyze these binaries. To accelerate backward slicing, we enhanced the backward slicing routine to perform slicing for each basic block in parallel to reduce the slicing time for NAMD to 30 minutes using 32 CPU threads, achieving a $24 \times$ speedup.

### 4.4.2 Blaming Instruction Stalls

After computing static def-use chains for a GPU function’s instructions, we create edges between a stalled instruction and its dependency sources that have any issued instruction sample. We call the resulted graph *instruction blame graph* to differentiate with the *static instruction dependency graph*. The blaming substrate supports both sampling and hybrid modes. In sampling mode, the instruction blamer considers every issued instruction as executed when attributing stalls and does not calculate efficiency metrics. We assign every dependency source one sample if none of them has been sampled as an issued instruction. In hybrid mode, our profiler uses one pass to collect instruction samples and a few more passes to collect instrumentation-based performance metrics. In hybrid mode, the instruction blamer assesses which instructions are executed to improve stall attribution.
Sampling Mode

**Cold edge pruning**  The instruction dependency graph our tool computes has many “cold edges” that are unlikely to cause stalls. Therefore, we employ the following heuristics rules to prune cold edges from the dependency graph to form the instruction blame graph.

1. **Opcode based pruning.** Attribute memory dependency stalls only to memory instructions and memory barrier instructions. Attribute synchronization dependency stalls only to synchronization instructions.

2. **Dominator based pruning.** For an edge $e$ from node $i$ to node $j$, remove the edge from the dependency graph if there is a non-predicated instruction $k$ that uses the same operands that $i$ defines and $j$ uses, and $k$ appears in every path from $i$ to $j$ in the control flow graph, because we would have observed stalls at $k$ rather than $j$ if $i$ caused any stalls.

3. **Instruction latency based pruning.** For an edge $e$ from node $i$ to node $j$, prune it from the dependency graph if the number of instructions in every path from $i$ to $j$ in the control flow graph is greater than the latency of $i$.

**Opcode based pruning** removes the edge from IMAD to the IADD in the sampling mode of Figure 4.4 because IMAD is an arithmetic instruction that does not cause memory dependency stalls.

Our tool uses micro-benchmarks [136] to measure the latency of fixed latency instructions. For variable latency instructions, such as memory load/store, our tool uses their upper bound to perform conservative pruning.
Stall attribution  After pruning cold edges, some nodes in the dependency graph may still have multiple incoming edges. Our tool blames stalls on the source of incoming edges based on the following two rules:

1. With more issued instruction samples, blame more stalls on the dependency source.

2. With a longer path, blame less stalls on the dependency source. If an instruction $i$ has multiple paths to instruction $j$ in the control flow graph, our tool calculates the average length of all paths to represent the length of edge $e$ between node $i$ and node $j$ in the dependency graph.

Equation 4.1 describes how our tool apportions stalls of an observed instruction $(S_j)$ on each dependency source $(S_i)$, where $R_{issue}^i$ and $R_{path}^i$ are the ratios of each incoming node $i$ calculated by heuristics (1) and (2) accordingly.

$$S_i = \frac{R_{path}^i \times R_{issue}^i}{\sum_{k \in incoming(j)} R_{path}^k \times R_{issue}^k} \times S_j$$  \hspace{1cm} (4.1)

Hybrid Mode

Our profiler’s hybrid mode considers information about executed instructions to blame stalls on their dependency sources more precisely. In hybrid mode, our tool employs the same backward slicing strategy used for sampling mode. However, we build an edge in the instruction blaming graph from node $i$ to node $j$ only if executed instructions have been observed for $i$. In hybrid mode, our tool employs an additional rule for cold edge pruning: it removes a path if any branch in the path has 1.0 instruction efficiency (always jump) and its next block is a fall through block, or 0.0 instruction efficiency (always fall through) and its next block is a jump block. If there
is no path from $i$ to $j$ after pruning, our tool removes this edge from the dependency graph. Consider Figure 4.4’s hybrid mode as an example, our tool removes the edge from LDL to IADD because the BRA after LDL has 1.0 instruction efficiency, indicating that no path contains B0 to B1 as a sub-path.

Finally, in the *stall attribution* phase, our tool calculates the ratios using executed instructions instead of issued instructions to apportion stalls (Equation 4.2).

$$S_i = \frac{\mathcal{R}_{i}^{\text{path}} \times \mathcal{R}_{i}^{\text{exec}}}{\sum_{k \in \text{incoming}(j)} \mathcal{R}_{k}^{\text{path}} \times \mathcal{R}_{k}^{\text{exec}}} \times S_j \quad \text{(4.2)}$$

**Stall classification**

Using information about dependency sources, our static analyzer further classifies memory dependency and execution dependency stalls as shown in Figure 4.5 and Figure 4.6 accordingly. Our tool categorizes memory dependencies as global memory, local memory, or generic memory according to the source instruction’s opcode. Execution dependency is more sophisticated. Our tool first classifies an execution dependency as shared memory, arithmetic, or write-after-read (WAR) based on op-
code. WAR dependency stalls occur when a use instruction writes a register that is read by a variable latency def instruction. It is worth noting that some execution dependency stalls may not have any dependency source. If such stalls occur on a memory instruction with indirect memory access, our tool classifies the stalls as address calculation stalls. For GPU architectures after Volta, modifiers such as X4 and X8 are extensively used to calculate address at memory instructions to reduce register usage. In other cases, our tool attributes stalls to the observed instruction and classifies them as scheduler stalls.

### 4.4.3 Identifying Access Kind

The raw value obtained for each GPU memory access is a sequence of bits, with no type information. A memory instruction can interpret a raw value in different ways. For example, a STS.64 instruction always stores 64-bit data to shared memory; the 64-bit data can be two 32-bit values or a single 64-bit value to shared memory, and the data type can be either float or integer. To distinguish values in different types, we define *access kind* to characterize memory accesses. An access kind is a triple
Algorithm 8 Identify GPU Instruction Access Kind

1: function FINDACCESSKIND(v, visited, dir)
2:     access_kind ← AccessKind(data_type=UNKNOWN, unit_size=0, vec_size=0)
3:     if v in visited then return access_kind
4:     visited ← visited ∪ {v}
5:     access_kind.vec_size ← GetVecSize(v.inst)
6:     neighbors ← v.outgoing_nodes if dir is FORWARD else v.incoming_nodes
7:     for n in neighbors do
8:         data_type, unit_size ← UNKNOWN, 0
9:         if n.inst.op is MOVE then
10:             kind ← FindAccessKind(n, visited, dir)
11:             data_type, unit_size ← kind.data_type, kind.unit_size
12:         else if n.inst.op is MEMORY then
13:             if dir is FORWARD and find v.inst.dst in n.inst.addr_regs then
14:                 data_type, unit_size ← 32 if n.inst.op is SHARED or LOCAL else 64, INTEGER
15:             else
16:                 ndir ← BACKWARD if dir is FORWARD else FORWARD
17:                 n.access_kind ← FindAccessKind(n, visited, ndir)
18:                 data_type, unit_size ← n.access_kind.data_type, n.access_kind.unit_size
19:         else
20:             data_type, unit_size ← GetDataTypeUnitSize(n.inst)
21:         if access_kind.data_type is UNKNOWN then
22:             access_kind.data_type ← data_type
23:         if access_kind.unit_size is 0 then
24:             access_kind.unit_size ← unit_size
25:         return access_kind
26:     return G

27: function CREATEDEPGRAF(instructions)
28:     G ← Graph()
29:     for i in instructions do
30:         G.V ← G.V ∪ {Vertex(i)}
31:     for v in G.V do
32:         for w in G.V with w.inst in v.inst.dep_insts do
33:             G.E ← G.E ∪ {Edge(from=w.inst, to=v.inst)}
34:     for v in G.V do
35:         if v.access_kind is not NULL or v.inst.op is MEMORY then
36:             continue
37:     visited ← Set()
38:     dir ← FORWARD if v.inst.op is LOAD else BACKWARD
39:     v.access_kind ← FindAccessKind(v, visited, dir)
40:     return G
associated with one memory access: unit size, vec size, and data type, where unit size indicates the size of each value, vec size indicates the number of values accessed, and data type specifies the value type, either float or integer. Once our tool knows the access kind of each memory instruction, it can correctly interpret the bit sequence accessed as values.

The high-level idea is that our tool creates a dependency graph for each function based on def-use chains and searches along these chains until access kind triples can be identified. While it is similar to the techniques used for type inference on CPU executables [145], our analysis is designed to track arithmetic data types (i.e., float vs. integer) with special handling for GPU instructions. The def-use chains for each GPU function is created using the method described in Section 4.4.1. The access kind identification analysis consists of two components:

- Access kind defining instruction. Our tool derives access kinds from a subset of GPU instructions. On NVIDIA GPUs, \texttt{CONVERT} instructions are used to convert between integer and float, or the same data type of different sizes; operands in \texttt{FLOAT} and \texttt{INTEGER} instructions are float and integer data types, respectively. Our tool can obtain the access kind for a value by directly decoding these instructions.

- Bidirectional search. Algorithm 8 depicts a sketch of our identification algorithm. Our algorithm employs both \texttt{Forward} and \texttt{Backward} orders to traverse along def-use chains using depth-first search. For each instruction, we iterate over all its neighbors and stop once an access kind’s unit size and data type can be explicitly identified by some arithmetic instructions. The search reverses the order when it encounters a memory load or store instruction because we do not
track access kind through memory.

![Diagram](image)

Figure 4.7: An example of access kind analysis. The search starts at STG.64 with Backward order until reaching LDS. The search proceeds in Forward order and terminates at DADD.

Figure 4.7 shows an example of the search algorithm. As the access kind for the STG.64 is unknown, we check the def instruction—MOV instruction that defines R8. Since the MOV instruction cannot tell its access kind, we keep searching for the def instruction that defines R14, which is the LDS instruction. However, the LDS instruction is not labeled with its access kind either; it depends on how R14 is used. Our tool then reverses the search direction to check uses of R14. Finally, our tool encounters the DADD instruction, which uses a single float 64 value in its registers. Our tool propagates the access kind back to both the LDS.64 and STG.64 instructions.

**Access Kind Accuracy**

Our approach may not recover all access kinds. For example, if the target register of a load instruction is immediately stored to memory and has no further use, our tool
Figure 4.8: Access kind coverage of different benchmarks

is unable to identify the load instruction’s unit size and data type. In such a case, our tool assigns a default unit size and data type predefined by users.

To quantify the accuracy of our tool’s access kind inference, we evaluated Algorithm 8 on several Rodinia benchmarks [29] and several CUDA SDK samples [146]. We define access kind coverage as the ratio of the number of correctly identified memory instructions over the total number of memory access instructions in all GPU kernels for a given program; 100% coverage means our access kind inference is always correct. We obtained the ground truth access kind for an instruction by manually checking every memory instruction in a GPU binary. It is worth noting that we evaluated coverage only for benchmarks because obtaining the ground truth for large kernels in applications is tedious and error-prone.

Figure 4.8 shows that our bidirectional slicing method for access kind inference yields high access kind coverage. Accuracy losses arise mainly due to two causes. On the one hand, nvdisasm fails to decode machine instructions for some GPU functions (e.g., euler3d). In such a case, we assign each GPU memory instruction the same unit size and data type. On the other hand, Algorithm 8 fails on some load instructions.
whose values are immediately stored to memory with no further use (e.g., dxtc and dct8x8).

4.5 Summary

To help application developers understand the performance of optimized GPU kernels, tools need compilers to generate high-quality DWARF information that describes the provenance of every machine instruction in each GPU kernel. While line mappings generated by today’s compilers relate most machine instructions back to an associated source line, they often fail to relate machine instructions back to any inlined call chain that caused it to be included. Starting from CUDA 11.5, NVIDIA has added encoded information within the linemap present in GPU binaries. We expect to see AMD and Intel to provide more complete inlining information in GPU binaries in the future.

While GPU instruction analysis can provide abundant information necessary for low level analysis, it has some limitations. First, NVIDIA does not expose binary analysis APIs to outside users; parsing text files generated by the command line nvdisasm tool [139] is error-prone. Second, while adding parallelism to backward slicing improves its speed, it also increases its memory footprint. At present, Dyninst performs backward slicing by analyzing the effects of individual instructions; it would be more space and time-efficient to use basic-block summaries, where appropriate. Third, our tool does not support analysis of several new instruction features, including branch divergence, warp synchronization, and tensor core instructions. We can leverage methods developed by others [147] to enhance our tool so that it can diagnose such issues.
Chapter 5

Performance Analysis and Case Studies

The last stage of performance analysis is generating performance reports to provide optimization insights. This chapter covers performance analysis modules and byproducts generated by analyses. We first review existing performance analysis methods in Section 5.1. Then, Section 5.2 highlights the differences between our approaches and others. Section 5.3, Section 5.4, and Section 5.5 describes our context-sensitive analysis, instruction stall analysis, and value redundancy analysis modules respectively.

5.1 Background and Related Work

Early efforts in GPU performance tools [14, 21, 20] focus on developing a flat trace or profile view. Since these tools do not have context information in performance reports, it is difficult to focus users’ attention on hotspots in large applications. To this end, some tools employ automatic analysis of GPU and CPU performance metrics to pinpoint bottlenecks. Early work on HPCToolkit [45] describes using GPU events and hardware counters for kernel-level monitoring on NVIDIA GPUs to compute profiles that blame CPU code for associated GPU idleness. Schmitt et al. [148] extended the blame analysis on CUDA to hybrid MPI-CUDA programs to identify application bottlenecks. Welton and Miller [44, 149] investigated hidden performance issues that impact several HPC applications but are not reported by tool APIs.

In addition to a trace or profile view, several tools examine code quality and pro-
vide optimization suggestions, mainly on the CPU. PerfExpert [150] collects performance metrics using sampling, analyzes measurement data and system parameters, and estimates performance upper-bounds. AutoScope [151] extends PerfExpert to suggest optimization strategies based on the detected bottlenecks. Unlike these two tools, CQA [152] builds a static model by emulating processor pipelines to check symptoms (e.g., vectorization) on the loop level. VTune [153] uses structured guidance to characterize bottlenecks by interpreting performance counters. Profile-guided optimization automates the tuning process one step further. It takes measurement data as input to guide the compiler to perform code transformation. Practical Path Profiling (PPP) [154] collects edge profiles using instrumentation to help compilers make decisions about function inlining and loop unrolling. Instrumentation-based methods require using representative inputs to dump meaningful profiles. To avoid the overhead of instrumentation-based approaches, AutoFDO [155] uses hardware counter based sampling to collect profiles for production applications and uses the profiles to guide optimizations. While most profile-guided optimization tools attribute measurement data to source lines to provide feedback for compilers, BOLT [156] is a post link optimizer that attributes samples on machine instructions and uses this information to derive binary optimizations. Recently, there also has been research that incorporates machine learning to guide optimizations. Cavazos et al. [157] use profile data as input features to a regression model that predicts the best compiler flags. DeepFrame [158] incorporates deep learning methods to learn the most likely paths during execution and offload the regions to FPGAs. Though profiler-guided optimizations can automatically adjust code based on rules or models, they only cover a subset of all the available optimizations. Many optimizations on GPUs need manual effort, such as warp balance, memory coalescing, and adjustments to thread counts.
The aforementioned analyses identify performance bottlenecks and attribute causes to specific code. However, they do not analyze redundancies in hotspot code. If a program is generating the same output repeatedly, it is often an indicator that redundant computations are performed. There exist several approaches to exploring value redundancies in CPU codes. Hardware-based approaches \cite{159, 160, 161, 162, 163, 164, 165} introduce new hardware components to detect and eliminate redundant computations and memory operations. Software-based approaches, such as RedSpy \cite{166}, LoadSpy \cite{167}, and Witch \cite{168} perform value profiling by leveraging binary rewriters (e.g., Intel Pin \cite{61}) or performance monitoring units and debug registers available only in CPU architectures. On GPUs, prior work mostly explores value redundancies via hardware approaches. Xiang et al. \cite{169} designed a hardware instruction reuse buffer to skip instructions with uniform values. Kim et al. \cite{170} proposed microarchitectural mechanisms to handle instructions composed of either uniform or affine value structures. Wang and Lin \cite{171} introduce a method to identify affine compute instructions and decouple them from the regular SIMT instruction pipeline.

5.2 Overview

Our tool’s context-sensitive analysis module pinpoints hotspots with specific contexts. Then, one can use the instruction stall analysis module to obtain the most effective optimization strategies. Our tool is the first performance advisor tool that suggests and estimates fine-grained and comprehensive optimization suggestions to the best of our knowledge. CUDAAdvisor \cite{68} only analyzes limited performance characteristics and instruments LLVM bit code. In contrast, our tool analyzes a set of performance issues. And our tool depends only on line-mapping information and is
not tied to any specific compiler. While nvprof and Nsight Compute provide high level optimization suggestions, they neither provide optimization suggestions at functions and lines nor estimate the potential speedup of each optimization.

Unlike such approaches that utilize specialized hardware components to identify the value redundancies, our tool’s value redundancy analysis module does not require any hardware extensions. Our tool uses instrumentation-based measurement like RedSpy and LoadSpy. However, there are significant differences in the approaches to identifying value redundancies because our tool: (1) handles sophisticated thread coordination across a massive number of SIMT threads on GPUs, (2) partitions the framework into GPU data collection and CPU on-the-fly analysis to avoid excessive memory overhead on GPUs, and (3) understands the data type (e.g., float or integer) and unit size (e.g., 32 or 64 bits) of values loaded or stored by memory instructions using data flow analysis whereas CPU registers are typically associated with fixed types.

Our tool’s value flow analysis is a variant of data flow analysis used in GPU programming models and frameworks, such as OpenMP task dependency graph (TDG) [13, 172], CUDA Graph [173], and automatic differentiation systems in deep learning frameworks [174, 8, 7]. Tensorflow’s monitoring framework—TensorBoard [175] supports a data flow view and value analysis. A fundamental limitation of TensorBoard is that it works for deep learning frameworks only and is generally applicable to other GPU-accelerated applications. In addition, Our tool differs from TensorBoard in several ways: (1) TensorBoard does not guide optimizations for value-related inefficiencies; (2) Unlike Our tool, which uses binary instrumentation, TensorBoard instruments program source code to collect graph topology and inspects tensor values; (3) TensorBoard only analyzes the distribution of values in
the end of kernels/iterations but does not capture values within GPU kernels.

5.3 Context-Sensitive Analysis

We can associate the full calling context with CPU and GPU code based on the profiles generated by our profiler and the static files generated by our static analyzer. To facilitate analysis, we also compute metrics in program contexts. Section 5.3.1 describes how we compute guidance metrics. Section 5.3.2 demonstrates how we optimize GPU code using context-sensitive information.

5.3.1 Methodology

A performance tool should focus users’ attention on the most expensive program contexts. Here we describe how to compute some useful derived metrics to identify such hotspots. Let $C_i$ and $G_i$ be the inclusive time spent in calling context $i$ on CPUs and GPUs respectively. GPU computations are typically launched asynchronously, so $C_i$ and $G_i$ may overlap. Let $P$ represent the calling context for the whole program. $I_{C,i}$ is the importance of CPU time spent in calling context $i$.

$$I_{C,i} = \text{Max}\left(\frac{C_i - G_i}{C_i}, 0\right) \times \frac{C_i}{C_P}$$ (5.1)

If a calling context takes more time on GPUs than CPUs, we consider the context unimportant and set $I_{C,i}$ to zero. Otherwise, $\frac{C_i - G_i}{C_i}$ indicates the importance of CPUs for this calling context and we multiply it by $\frac{C_i}{C_P}$ to normalize it to the whole execution.

If we find the bottleneck of an application is not on CPUs, we compute $I_{G,i}$ as shown in Equation 5.2 to assess the importance of GPU time in calling context $i$ by computing the fraction of the total GPU computation time spent in context $i$:
\[ I_{G,i} = \frac{G_i}{G_P} \] (5.2)

Let \( G_{i,j} \) represent the GPU time spent in calling context \( i \) performing a particular GPU operation type \( j \), e.g. kernel execution. We refine Equation 5.2 to consider operation types in Equation 5.3.

\[ I_{G,i,j} = \frac{G_{i,j}}{G_P} \] (5.3)

The metrics described by equations 5.1–5.3 are considered in sequence to identify performance bottlenecks. If a program has low \( I_{C,P} \), we follow paths from the calling context tree root to GPU operations where \( I_{G,i} \) is high and then identify the problematic GPU operation type \( j \) at that spot. For example, if memory copies and synchronizations take a long time, we first inspect the most expensive memory copies and synchronizations instead of optimizing GPU kernels.

Our tool provides three different views to guide analysis. The \textit{top-down} view illustrates performance metrics associated with each node in the calling context tree. It is often used to understand the sophisticated behavior of an application. The \textit{bottom-up} view aggregates metrics of the same GPU operation from multiple call paths. It is used to locate the hotspot and find call paths that can be optimized. The \textit{trace} view shows when a GPU operation happened and how long it takes. It can be used to correlate CPU and GPU activities and apply causality analysis [176, 113, 177] to find the root cause of idleness.

5.3.2 Case Studies
Top-down Analysis

We studied PeleC’s [81] PMF problem on NVIDIA’s GPUs only because there are not yet stable HIP or DPC++ implementations. We ran PeleC’s TG problem on the NVIDIA, Intel, and AMD GPUs and compared the performance characteristics to understand relative the performance of different GPUs and how efficiently the software maps to each.

**NVIDIA GPU**  We profiled PeleC’s PMF example with its default input using instruction sampling on an NVIDIA V100 GPU. Using our tool’s heterogeneous calling context, we were able to identify the hot kernel quickly. We noted that the GPU utilization for kernel `pc_expl_reactions` was only 2.5% on average, indicating low SM utilization. By reducing the number of threads per block from 256 to 128, we increased the number of blocks for this kernel from 16 to 32 and improved its performance by `1.14x`.

![Figure 5.1: An unnecessary synchronization calling context in PeleC](image)

Next, we profiled PeleC’s TG example with its default input. The generated profile shows that the time spent on GPU synchronization is close to GPU kernel execution time. We found an unusual phenomenon: in some cases, the number of
GPU synchronization API invocations exceeds the number of GPU kernel launches. We computed a derived metric: \( \text{diff} = \text{sync\_count} - \text{kernel\_count} \) in hpcviewer to find where synchronizations are unnecessary. Figure 5.1 shows three calling contexts where \text{diff} is high. In the first calling context, no kernel launch occurs in the loop while there are synchronization invocations. We found that an \text{MFIter} object is created for the loop, and the synchronization is called in the object’s deconstructor to synchronize computations in the loop. We optimized the code by not invoking synchronization if no computation is performed in the loop. In the second calling context, synchronization is needed only when there are multiple MPI ranks. Thus, we can conditionally invoke synchronization by checking the number of MPI ranks. In the third calling context, the synchronization is redundant because a copy function immediately before always synchronizes all GPU streams. These three code changes reduced the number of synchronization invocations by 38% and improved end-to-end performance by 1.05\times.

**Intel and AMD GPUs** We compiled the DPC++ code of PeleC’s TG example and ran it on an Intel Gen9 GPU using Intel’s OpenCL backend. We captured four coarse-grained metrics, including kernel execution time, memory transfer time, memory transfer bytes, and memory allocation bytes. For fine-grained measurement, our tool collected GPU instruction counts using instrumentation added to GPU kernels using Intel’s GT-Pin library.

The profile view in Figure 5.2 enables one to quickly identify the most costly GPU kernels with respect to GPU instruction count: \text{pc\_compute\_hyp\_mol\_flux} and \text{pc\_compute\_diffusion\_flux}, which execute 37.6% and 29.6% of the GPU instructions respectively. Measurement and attribution of instruction counts within each
Figure 5.2: Fine-grained measurements of PeleC’s TG benchmark collected with binary instrumentation on an Intel Gen9 GPU. PeleC is implemented using DPC++ and executing atop Intel’s OpenCL runtime.
kernel revealed an interesting detail: roughly one third of the instructions in each kernel are for array index calculation. It is worth noting that the fraction of kernel execution time measured for this kernel while collecting instruction level measurements may not be accurate since kernel instrumentation inflates kernel execution time.

In addition, we noted $28.8 \times 10^6$ bytes of GPU memory was allocated, but no memory transfers were measured. In this case, the memory transfers are implicit; thus, wrapping public OpenCL APIs may miss observing internal memory transfers performed by a vendor’s implementation. To observe implicit memory transfers, we either need either vendors to use only public APIs for data transfers or a better tool API.

In experiments with PeleC on AMD GPUs, our tool can collect heterogeneous call stacks that attribute costs to GPU operations, including kernel launches and data copies. Without support for fine-grained measurements, we are unable to measure performance inside GPU kernels.

**Bottom-up Analysis**

Using our tool’s bottom-up view, we delved into the bottlenecks of the GPU execution of Laghos [30]. Laghos is implemented in both CUDA and RAJA. We studied each version separately.

**Laghos-CUDA**  We used $I_{G,i,j}$ metrics to check importance of each GPU operation. Overall, GPU memory copies take 11% of the GPU execution time. We show the top three GPU memory copies in their CPU calling contexts in Figure 5.3.

We used different methods to optimize each of the three memory copies in Figure 5.3. For the first case, we noted `mfem::CudaProlongationOperator::Mult` and
Figure 5.3: A bottom up view of memory importance in Laghos

mfem::CudaProlongationOperator::MulTranspose do nothing but copy memory in a single process environment. In mfem::CudaRAPOperator::Mult, we can use the original memory without copying. Thus, we move environment check conditions to mfem::CudaRAPOperator::Mult, eliminating the two memory copies when using a single process.

In the second case, we observed that an array distX is copied from input x and is modified only if ess_tdofs_count is not zero. Thus, we proposed an optimization that conditionally copies x to distX based on ess_tdofs_count.

For the third case, we examined the memory copy’s invocation times and transferred bytes, inferring that a small piece of data is moved frequently from the GPU to the CPU. Using pinned host memory avoids extra copies between pageable and pinned memory.

Applying the above optimizations to Laghos increased overall performance by 6%, and the code sections that contain GPU APIs increased by 25%. Note that
blindly applying optimizations without consulting metrics in the calling contexts can have hurt performance. For example, in the third case, if the amount of transferred memory is massive, using pinned memory reduces the amount of memory available to the system and thereby may slow down the performance as a whole.

**Laghos-RAJA**  We ran the RAJA version of Laghos with the same input and parameters as the CUDA version. We observed that the RAJA version is 22% slower than the CUDA version, but $I_{G,P}$ of the RAJA version is higher than the CUDA version. Unlike the CUDA version, the RAJA version has high $I_{G,P,sync}$ while the CUDA version does not induce any explicit synchronization. By examining the synchronization contexts, we found that a stream synchronization occurs every time a kernel is launched, because Laghos developers used the default RAJA kernel launch pattern. We fixed the problem by making the kernel launches asynchronous.

After applying the first optimization, we noticed the RAJA version is still slower than the CUDA version. The major difference lies in the `rMassMultAdd2D<3,4>` kernel, where the RAJA version takes 66% more time than the CUDA version. The RAJA code and CUDA code are the same except for the RAJA template wrapper. By comparing every latency reason of the two codes, we found that the RAJA code has $3\times$ the memory dependency latencies and $2\times$ the memory throttling latencies of the CUDA code, which means the RAJA code may cause more memory requests. Next, we checked the instruction mix of the two codes and located a specific line that caused the problem. With the RAJA template wrapper, NVIDIA’s `nvcc` compiler cannot infer that store operations in a loop access the same address on the global memory in each iteration so that it generates $4\times$ as many `STG` instructions as the CUDA code.
Figure 5.4: Nyx’ trace view running across 22 nodes using 640 GPU streams and 128 processes.

Trace Analysis

Figure 5.4 shows a trace view of Nyx executing on Summit using 640 streams across 128 GPUs. The trace view shows that this execution consists of five phases. By inspecting the call stacks in the trace view, we can determine what the phases do, e.g., initializing particle and dark matter data, performing hydrodynamic calculations, calculating gravity, etc.

For each phase, we employed blame analysis to understand why GPUs are idle. In the first phase, 58.01% of idleness is caused by a call to `cuCtxSynchronize`, which synchronizes all streams on the GPU. Because only a single stream is used in this phase and a synchronized memory copy always follows the call to `cuCtxSynchronize`, we can safely remove the synchronization call and reduce the running time by 0.6s.

In the second phase, we easily identified that the major cause of idleness is JIT compilation at runtime. By providing the specific GPU architecture flag to the compiler and recompiling the program, we reduced the running time by 0.2s with this
single optimization.

In the following three phases, we identified that idleness is caused by calls to MPI\_Waitall, which suggests that there may be opportunities for improving performance by optimizing communication.

The insights provided by blame shifting analysis reduced the GPU running time of Nyx from 10.6s to 9.8s, achieving a 1.08\times speedup with 640 GPU streams. While the problems and improvements that we describe here are small, the important part is that our tool provides developers insights to identify even small problems.

5.4 Instruction Stall Analysis

To automate performance analysis and tuning, we developed an instruction stall analysis substrate that suggests potential code optimizations at a hierarchy of levels, including individual lines, loops, and functions. This module uses instruction samples, binary instrumentation, and static analysis components mentioned in previous chapters to attribute stalls to their root cause instructions. Further, it calculates instruction efficiency and memory efficiency for each instruction. The analysis module consists of a couple of performance optimizers and estimators. First, we employ performance optimizers to match instruction stalls with potential optimizations based on program structure, instruction efficiencies, and memory efficiencies of hot program regions. Next, performance estimator components estimate each optimization’s speedup based on the matched stalls. Finally, our tool outputs top-recommended optimization suggestions to a raw report.

Our tool supports both sampling and hybrid modes. In sampling mode, we collect instruction samples in instruction profiles. As a result, optimizations that rely on efficiency metrics are not available in the sampling mode. In hybrid mode, our
profiler uses one pass to collect instruction samples and a few more passes to collect instrumentation-based performance metrics; all GPU optimizers are used to match stalls with optimization strategies. To prepare an application for study, a developer needs to specify appropriate flags so that the compiler will include line mapping information in the generated executable. Our tool requires line mapping information for meaningful performance reports. Our tool’s performance advice report for an application contains optimization suggestions ranked by their estimated speedup in text format. One can start with our tool’s sampling mode to profile and analyze an application with a representative input. If the output advice does not contain insightful suggestions to guide performance optimization, one can switch to hybrid mode, enabling more optimizers with higher profiling and analysis overhead. Currently, our tool’s GUI associates instruction stalls on each GPU kernel’s source lines, loops, and functions.

5.4.1 Performance Optimizers

Each performance optimizer in our tool encodes rules to match stalls that could be reduced by its optimization. Table 5.1 presents all performance optimizers available in our tool. Each optimizer maintains a workflow to match instruction samples. Optimizers annotated with \( \mathcal{H} \) are only available in hybrid mode as they rely on instruction efficiency and/or memory efficiency.

At a high level, optimizers either improve parallelism or improve code efficiency. Parallelism optimizers check if a GPU kernel lacks sufficient parallelism and recommends adjusting the number of blocks and threads. For example, if occupancy is bounded by the number of threads, the *Thread Increase* optimizer suggests increas-
Table 5.1: A description of GPU optimizers in our tool.

<table>
<thead>
<tr>
<th>Code Optimizers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall Elimination</td>
</tr>
<tr>
<td>Register Increase</td>
</tr>
<tr>
<td>Strength Reduction</td>
</tr>
<tr>
<td>Function Split</td>
</tr>
<tr>
<td>Fast Math</td>
</tr>
<tr>
<td>Warp Balance</td>
</tr>
<tr>
<td>Global Memory Transaction Reduction</td>
</tr>
<tr>
<td>Shared Memory Transaction Reduction</td>
</tr>
<tr>
<td>Indirect Memory Access Reduction</td>
</tr>
<tr>
<td>Global Memory Access Adjustment</td>
</tr>
<tr>
<td>Shared Memory Access Adjustment</td>
</tr>
<tr>
<td>Latency Hiding</td>
</tr>
<tr>
<td>Loop Unrolling</td>
</tr>
<tr>
<td>Code Reordering</td>
</tr>
<tr>
<td>Function Inlining</td>
</tr>
<tr>
<td>Asynchronous Memory Copy</td>
</tr>
<tr>
<td>Branch Elimination</td>
</tr>
<tr>
<td>Parallelism Optimizers</td>
</tr>
<tr>
<td>Block Increase</td>
</tr>
<tr>
<td>Thread Increase</td>
</tr>
</tbody>
</table>
ing the number of threads per block to hide latency. The Block Increase optimizer investigates the potential of increasing the number of blocks.

Code optimizers check if certain patterns of stalls exist in particular program regions. Based on optimization methods, we further categorize the code optimizers as stall elimination and latency hiding optimizers. Stall elimination optimizers provide suggestions to reduce stalls; latency hiding optimizers suggest rearranging issue orders to overlap stall latency. The *Loop Unrolling* optimizer, for example, iterates through all latency samples. It records a latency sample if it has either a memory dependency stall or an execution dependency stall, and the def and use instructions are within the same loop. The optimizer suggests using pragma unroll annotation or manual unrolling for loops where the compiler fails to unroll automatically. *Branch Elimination* and *Asynchronous Memory Copy* optimizers are available in the hybrid mode only, using instrumentation-based metrics. The branch elimination optimizer checks branches instructions that are determined (always executed/not executed) and aggregates all stalls from the source block to the destination block because it assumes instructions in the two blocks can be rearranged to reduce stalls. Asynchronous memory copy instructions (*LDGSTS*, *LDGMEMBAR*), which load values from global memory directly to the shared memory, are new to NVIDIA’s Ampere GPU. In earlier GPUs, one must employ a global memory read instruction followed by a shared memory store instruction to load values to shared memory; such a strategy causes large register overhead and exposes global memory latency. The asynchronous memory copy optimizer finds instruction dependency pairs from global memory reads to shared memory stores and sums their stalls. The optimizer suggests using asynchronous memory copy instructions to replace synchronous memory copy instructions to increase the distance between load and use. Moreover, the optimizer also checks the stalls attributed to
asynchronous memory copy instructions and suggests increasing the distance between memory transaction commit (LDGMEMBAR) and its barrier (DEPBAR) if latency is not hidden as expected.

5.4.2 Performance Estimators

Performance optimizers match optimization suggestions with stalls, but do not provide information about which methods have a better effect considering the given measurement data, program structure, and the GPU architecture. Our tool addresses this issue with performance estimators that estimate the speedup of each optimizer based on their matched stalls. Our tool employs both code optimization estimators and parallelism optimization estimators. Suggestions from optimizers with top estimated speedups are output to a performance advice report.

**Code Optimization Estimators** We first estimate the effect of stall elimination optimizers. Suppose a GPU kernel has a total of \( T \) instruction samples and the matched samples for an optimizer is \( M \). Stall elimination optimizers assume that all matched stalls, including matched active samples and latency samples, can be eliminated by optimizing the code. We derive Equation 5.4 to estimate the speedup of stall elimination optimizers \( S^e \).

\[
S^e = \frac{T}{T - M} \tag{5.4}
\]

Unlike stall elimination optimizers, latency hiding optimizers reduce latency samples. We use Equation 5.5 to estimate the speedup of latency hiding optimizers \( S^h \), where \( M^L \) is the number of matched latency samples.
Latency Hiding Example

• Reorder instructions to hide latencies

LDG R0, [R2] STALL STALL IMUL R5, R0, R5 IADD R8, R8, R8 IADD R9, R9, R9

Figure 5.5: A mental model for latency hiding optimizers. Green code indicates active samples, and red stalls represent latency samples; latency hiding optimizers consider the effect of moving the code in the dashed box to fill the stall slots.

\[ S^h = \frac{T}{T - M^L} \]  

Equation 5.5 supposes all latency samples can be eliminated by rearranging instructions. However, in practice, not all \( M^L \) can be eliminated. Figure 5.5 explains the mental model of latency hiding optimizations, where the upper bound of the optimizations is bounded by both the matched latency samples and active samples. We derive Equation 5.6 to refine the estimate of \( S^h \), where \( A \) denotes the total number of active samples.

\[ S^h = \frac{T}{T - \text{Min}(A, M^L)} \]  

Some optimizations such as loop unrolling only rearrange code for a specific scope. Therefore, only active samples within the scope can be used to reduce the scope’s latency samples. Based on this limitation, we refine the upper bound of latency hiding optimization with Equation 5.7 to consider optimization scopes representing loops and functions. \( S^h_l \) indicates the speedup for a specific scope \( l \), and \( M^L_l \) is the
matched latency samples for a scope $l$.

$$S_l^h = \frac{T - \text{Min}(\sum_{l' \in \text{nested}(l)} A_{l'}, M_{l'})}{A_l} \quad (5.7)$$

Suppose we have two loops loop1 and loop2, where loop1 is nested in loop2, the speedup of loop2 is bounded by the total number of active samples of loop2 and loop1 according to Equation 5.7.

We prove that the upper bound of $S_l^h$ is two. We use $L$ to denote the total number of latency samples, and $T = A + L$.

**Theorem 5.1**

The speedup upper bound of latency hiding optimizations is $2 \times$.

**Proof 5.1**

• If $\text{Min}(A, M^L) = A$. $\frac{T}{T-A} = \frac{L+A}{(L+A)-A} = 1 + \frac{A}{L}$.

Because $A \leq M^L \leq L$, $\frac{T}{T-\text{Min}(A,M^L)} \leq 2$.

• If $\text{Min}(A, M^L) = M^L$. $\frac{T}{T-M^L} = \frac{1}{1-\frac{M^L}{L}} = \frac{1}{1-\frac{1}{L}}$.

Because $L \geq M^L$ and $A \geq M^L$, $\frac{M^L}{A+L} \leq \frac{1}{2}$.

Then $\frac{T}{T-\text{Min}(A,M^L)} \leq 2$.

**Parallelism Optimization Estimator** Unlike code optimizers, parallelism optimizers adjust the number of blocks and threads to improve performance. To estimate the speedup of parallelism optimizers, we consider each warp scheduler’s change of active warps $C_w$ (Equation 5.8) and change of issue rate—$C_T$ (Equation 5.9).

For example, if the number of blocks is increased, each warp scheduler is assigned fewer active warps so that $C_w$ is less than one. At the same time, as the number of threads is reduced, the rate that a warp scheduler is issuing is reduced, and $C_T$ is less than one.
\[ C_W = \frac{W_{new}}{W} \] (5.8)

\[ C_I = \frac{I_{new}}{I} \] (5.9)

By assuming each warp scheduler has the same issue rate, we derive Equation 5.10 and Equation 5.11 to calculate issue rate \( I \) and \( I_{new} \) respectively, where \( R_I \) is the ratio of issued samples among all samples. A warp scheduler is issuing if at least one warp assigned to the scheduler is ready to issue an instruction.

\[ I = 1 - (1 - R_I)^W \] (5.10)

\[ I_{new} = 1 - (1 - R_I)^{W_{new}} \] (5.11)

\[ S^p = \frac{1}{C_W} \times C_I \times f \] (5.12)

We estimate the speedup of parallelism optimizations (\( S^p \)) based on \( C_W \) and \( C_I \) using Equation 5.12, where \( f \) is a factor that varies between optimizers. Some optimizers assume there are no pipeline, memory throttle, and select stalls if we reduce the number of active warps per block to a certain number (e.g., less than the number of schedulers per SM).

### 5.4.3 Case Studies

Table 5.2 compares the estimated speedups reported by our tool’s hybrid mode and sampling mode. Most benchmarks show similar estimated speedups in two modes.
Table 5.2: Achieved speedups averaged among five runs. We improved each code according to the suggestion provided by GPA. Estimate error is computed by \( \frac{|\text{Estimated Speedup} - \text{Achieved Speedup}|}{\text{Achieved Speedup}} \times 100\% \).

<table>
<thead>
<tr>
<th>Application</th>
<th>Optimization</th>
<th>Achieved Speedup</th>
<th>Estimated Speedup</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>rodinia/backprop</td>
<td>Warp Balance</td>
<td>1.15±0.03×</td>
<td>1.14×</td>
<td>1%</td>
</tr>
<tr>
<td>rodinia/backprop</td>
<td>Strength Reduction</td>
<td>1.21±0.01×</td>
<td>1.24×</td>
<td>2%</td>
</tr>
<tr>
<td>rodinia/bfs</td>
<td>Loop Unrolling</td>
<td>1.12±0.01×</td>
<td>1.59×</td>
<td>42%</td>
</tr>
<tr>
<td>rodinia/b+tree</td>
<td>Code Reorder</td>
<td>1.16±0.10×</td>
<td>1.28×</td>
<td>10%</td>
</tr>
<tr>
<td>rodinia/cfd</td>
<td>Code Reorder</td>
<td>1.60±0.02×</td>
<td>1.68×</td>
<td>5%</td>
</tr>
<tr>
<td>rodinia/gaussian</td>
<td>Thread Increase</td>
<td>3.58±0.15×</td>
<td>3.32×</td>
<td>7%</td>
</tr>
<tr>
<td>rodinia/heartwall</td>
<td>Loop Unrolling</td>
<td>1.17±0.03×</td>
<td>1.18×</td>
<td>1%</td>
</tr>
<tr>
<td>rodinia/hotspot</td>
<td>Strength Reduction</td>
<td>1.14±0.01×</td>
<td>1.09×</td>
<td>4%</td>
</tr>
<tr>
<td>rodinia/huffman</td>
<td>Warp Balance</td>
<td>1.07±0.00×</td>
<td>1.17×</td>
<td>9%</td>
</tr>
<tr>
<td>rodinia/kmeans</td>
<td>Loop Unrolling</td>
<td>1.11±0.01×</td>
<td>1.20×</td>
<td>8%</td>
</tr>
<tr>
<td>rodinia/lavaMD</td>
<td>Loop Unrolling</td>
<td>1.11±0.03×</td>
<td>1.12×</td>
<td>1%</td>
</tr>
<tr>
<td>rodinia/lud</td>
<td>Code Reorder</td>
<td>1.41±0.00×</td>
<td>1.48×</td>
<td>5%</td>
</tr>
<tr>
<td>rodinia/myocyte</td>
<td>Fast Math</td>
<td>1.22±0.03×</td>
<td>1.13×</td>
<td>7%</td>
</tr>
<tr>
<td>rodinia/myocyte</td>
<td>Function Splitting</td>
<td>1.02±0.01×</td>
<td>1.01×</td>
<td>1%</td>
</tr>
<tr>
<td>rodinia/nw</td>
<td>Warp Balance</td>
<td>1.07±0.01×</td>
<td>1.09×</td>
<td>2%</td>
</tr>
<tr>
<td>rodinia/particlefilter</td>
<td>Block Increase</td>
<td>1.75±0.01×</td>
<td>1.92×</td>
<td>10%</td>
</tr>
<tr>
<td>rodinia/streamcluster</td>
<td>Block Increase</td>
<td>1.52±0.03×</td>
<td>1.35×</td>
<td>11%</td>
</tr>
<tr>
<td>rodinia/sradv1</td>
<td>Warp Balance</td>
<td>1.02±0.01×</td>
<td>1.10×</td>
<td>8%</td>
</tr>
<tr>
<td>rodinia/pathfinder</td>
<td>Code Reorder</td>
<td>1.04±0.01×</td>
<td>1.31×</td>
<td>26%</td>
</tr>
<tr>
<td>ExaTENSOR</td>
<td>Strength Reduction</td>
<td>1.11±0.01×</td>
<td>1.06×</td>
<td>5%</td>
</tr>
<tr>
<td>ExaTENSOR</td>
<td>Memory Transaction Reduction</td>
<td>1.03±0.00×</td>
<td>1.05×</td>
<td>2%</td>
</tr>
<tr>
<td>PeleC</td>
<td>Block Increase</td>
<td>1.21±0.01×</td>
<td>1.23×</td>
<td>2%</td>
</tr>
<tr>
<td>Minimod</td>
<td>Fast Math</td>
<td>1.03±0.01×</td>
<td>1.09×</td>
<td>6%</td>
</tr>
<tr>
<td>Minimod</td>
<td>Code Reorder</td>
<td>1.04±0.01×</td>
<td>1.05×</td>
<td>1%</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>1.22×</td>
<td>1.26×</td>
<td>4.1%</td>
</tr>
</tbody>
</table>
We note that the hybrid mode outperforms the sampling mode from the following aspects. First, the hybrid mode offers optimization suggestions that are not available in the sampling mode. For example, based on the global memory access adjustment optimizer, we achieved a $1.22 \times$ additional speedup for myocyte by coalescing the global memory accesses to an array. Second, the hybrid mode employs extra pruning and attribution rules to attribute and apportion stalls, which increases the estimate accuracy.

We selectively presented a few cases from Table 5.2:

- ExaTensor [80] is a tensor algebra library implemented on NVIDIA GPUs. We studied its tensor transpose kernel with a six-dimensional tensor.

- Minimod [82] is a stencil benchmark for seismic modeling. We analyzed its performance with a grid size of $100^3$.

- NAMD [79] is a high performance parallel molecular dynamics code for simulating large biomolecular systems. We used its alanin input configuration running $9 \times 10^5$ steps.

- BerkeleyGW [26] is a GPU-accelerated code that uses the GW method to calculate the quasiparticle properties and the optical responses of a large variety of materials. We studied it’s matrix elements calculation ($mtxel$) kernel.

- Quicksilver [23] is a proxy application that solves a dynamic Monte Carlo particle transport problem. Quicksilver has a single large kernel that invokes many device functions consisting of thousands of lines of code. We studied Quicksilver with its default input.
PeleC [81] is an application for reacting flows using adaptive-mesh compressible hydrodynamics. We studied PeleC using its default input.

The case studies were performed on a system with an NVIDIA A100 GPU. When compiling these codes for the GPU, we used the options `-lineinfo -O3` to generate line mapping information helpful for our tool’s performance reports.

**ExaTensor**

Figure 5.6 shows the top performance suggestion from our tool’s performance report for the ExaTensor benchmark. In our tool’s performance reports, GPU kernels are ordered by their execution time. For each kernel, the report lists several performance optimization suggestions ranked by their estimated speedups. For each suggestion, our tool offers *hints* about code changes to improve performance and lists hotspots where those hints could be applied. For each hotspot, our tool supplies *program context, importance,* and *speedup* information. Using the hybrid mode, our tool also provides the memory efficiency and instruction efficiency about the stall source. Program context provides information about the locations of the stalls and their dependency sources. The importance metric indicates the percentage of stalls
this optimizer matches, and the speedup metric indicates the estimated speedup after applying suggested code changes.

In Figure 5.6, our tool estimates that applying asynchronous memory copy optimization may improve code performance by 1.35×. Our tool also provides the locations of the matched hotspot for this optimization. We show this hotspot’s assembly instructions in Figure 5.7, in which a load from global memory (LDG) instruction is immediately followed by a store to shared memory (STS) instruction. Following our tool’s hints, we used a LDGSTS instruction to replace this LDG and STS pair and overlap the calculation of transposed index for each element with data transfer. This single optimization achieves a 1.24× speedup, which is 10% lower than the estimated 1.35× speedup because the scope for code rearrangement is limited by synchronization. On NVIDIA GPU architectures prior to the A100, which lack support for asynchronous memory copy, it is difficult to optimize this code. As illustrated in Figure 5.7, if we increase the distance between LDG and STS, we have to keep register R2 alive. Since a large tile is loaded into shared memory, separating each pair of dependent instructions would require many live registers and significantly increase register pressure.

We used our tool again to analyze optimized code. This time our tool suggests eliminating high execution dependency stalls caused by indirect memory access at constant memory load instructions that read the dimensions of the input tensor. We can assign the number of dimensions as a template parameter for this kernel so that the compiler can treat it as a constant and eliminate indirect memory access. This
optimization achieves another 1.03× speedup, which is close to the estimated 1.06× speedup.

Minimod

We applied our tool to analyze the target_pml_3d kernel of Minimod. This kernel loads all values from a three dimensional tensor at once and performs a high-order stencil computation with a halo size of 4. Every thread with an index within a halo size of a tile boundary reads values from a halo area.

Our tool first suggests using the fast math functions to replace high precision match functions. We applied the --use_fast_math compiler flag to achieve a 1.03× speedup.

Next, our tool suggests the code reordering optimizations for the updated code. Adjusting the code to read subscripted values from global memory well in advance of their use hides more of the memory latency and yields an additional 1.04× speedup.

Then, by incorporating instrumentation-based metrics, our tool’s branch elimination optimizer suggests some branches are always true. Because each GPU block has $32 \times 4 \times 4$ threads, each thread’s Y and Z dimensions are always less than the halo size away from a tile boundary. To eliminate stalls caused by these branch conditions, we used the block size as a template parameter and let the compiler elide the branch conditions at compilation time. This optimization achieves a 1.07× speedup, which matches our tool’s 1.06× estimated speedup.

Finally, our tool’s top warp balance optimization highlights significant synchronization dependency stalls at a _syncthreads invocation in a loop and suggests eliminating unnecessary synchronizations. Two synchronizations are needed to prevent data race if a single tile is used. If the amount of shared memory use is small,
we can use two tiles to read and update shared memory separately to eliminate one synchronization in each iteration. We achieved a $1.26 \times$ speedup by employing this optimization, which is the same as our tool’s estimated speedup.

**NAMD**

We used our tool to analyze the most costly `nonbondedForceKernel` function in NAMD. Based on our tool’s report, we observed that NAMD is a highly optimized application. Our tool’s report only suggests two significant optimizations—code reorder and register increase. We first followed the hints of the code reorder optimizer but failed to obtain a non-trivial speedup with our tool since the data dependencies are intricate. Then we checked the register increase optimizer which identifies the locations of local memory loads and stores caused by register spills, and suggests a $1.05 \times$ speedup by eliminating these stalls. This kernel uses `__launch_bounds__` to imply the minimum number of concurrent blocks on each SM of a GPU and thus enforces a very low register limit per thread. However, for the input we studied, while the number of kernel invocations is large, the number of blocks used by each kernel is small. By using a special `__launch_bounds__` constraint to allow more registers when the number of blocks is small, we achieved a $1.09 \times$ speedup.

**BerkeleyGW**

Listing 7 shows a code snippet distilled from a hot loop in *BerkeleyGW* [26]—an exascale GW approximation code that uses OpenMP to offload computation to GPUs. GPU profilers such as Nsight-Compute report high execution dependency stalls in this loop but provide neither information about the cause of the stalls nor
candidate optimizations to reduce the stalls. GPA attributes stalls to their causes, matches patterns of inefficiency with optimization strategies, and estimates the potential speedup for each applicable optimization. For the code in Listing 7, GPA attributes stalls to a device function that performs slow complex number division, and associates the device function with its call site on Line 3. To avoid using the costly device function, GPA suggests replacing the division with a multiplication by its reciprocal* and estimates a 1.18× speedup, which achieves close to the 1.24× speedup the optimization delivers.

Our tool’s loop unrolling optimizer also suggests unrolling the core loop on Line 1 to improve performance. We confirmed that this loop is not automatically unrolled because of dependencies among iterations and notified the developers about such a potential optimization. Next, we profiled BerkeleyGW again using our tool’s hybrid mode. This time our tool’s global memory access adjustment optimizer indicates that the `aqsmtemp_local` array on Line 5 is accessed with low efficiency and estimates a 1.09× speedup by coalescing memory accesses. The estimated speedup matches the performance difference between this `mxtel` kernel’s CUDA version and OpenMP version, and the CUDA code loads `aqsmtemp_local` to shared memory using coalesced memory read to ameliorate this problem. Unfortunately, shared memory is declared

*One can simplify division by a complex number by multiplying the numerator and denominator by the complex conjugate of the denominator.

Listing 7: A hot loop in the *BerkeleyGW* code.
implicitly in OpenMP Target code and is not widely supported in existing compilers.

Quicksilver

We used our tool to analyze Quicksilver on a single GPU. Our tool reports the function inlining optimization may render the highest speedup. Applying the `always_inline` keyword for these functions fails to inline due to the size/register limitation enforced by the compiler. Therefore, we manually inlined two small functions by integrating the whole function bodies into their callers. By modifying the code in this way, we obtained a $1.14 \times$ speedup.

Next, our tool’s register reuse optimizer indicates local memory stalls in a loop and points out the potential cause of register spilling. Our tool suggests splitting the loop into two to save registers. Without our tool, the raw PC sampling report by other tools only shows global memory stalls without identifying register pressure. Applying the optimization yields a $1.01 \times$ speedup. Note that the estimated speedup ($1.05 \times$) is higher than the achieved speedup because splitting the loop had the result of increasing the total number of instructions executed.

PeleC

We studied the `react_state` kernel of PeleC. Our tool estimates the code reordering optimization may result in the highest speedup. However, because the top five hotspots only account for 4% all of the matched stalls, there are many hotspots distributed across lines so it is difficult to adjust the code manually. The second best optimizer suggests increasing the number of blocks. Since the kernel only occupies 16 blocks, our tool suggests reducing the number of threads per block while increasing the number of blocks to improve the parallelism. By increasing the number of blocks
to 32, we achieved a $1.21 \times$ speedup.

5.5 Value Redundancy Analysis

Unlike the previous analyses, value redundancy analysis aims to detect redundant memory accesses involving the same values, which is a problem that pervasively exists in GPU-accelerated applications. Since hardware counters do not capture values produced or used, values can only be captured using binary instrumentation. We developed a value redundancy analysis module to calculate redundancy metrics and identify inefficient value access patterns to address this issue.

To enable our tool for use on production HPC systems, we addressed several challenges listed below in collecting, maintaining, analyzing, and presenting a large volume of performance data:

- We employed an efficient data collection mechanism to parallelize analysis and execution on GPUs with a massive number of threads (Chapter 2).
- We developed a hierarchical sampling scheme to reduce measurement overhead to a reasonable level (Chapter 2).
- As GPU assembly does not provide access kind information for memory instructions (i.e., integer or float of different sizes), we devised a novel bidirectional slicing method to extract access kind information (Chapter 4).

5.5.1 Redundancy Metrics

Our redundancy analysis modules investigates the values produced and used in GPU kernels and identifies the following four value redundancies metrics.
Definition 5.1 (Temporal Load Redundancy) A memory load \( L_2 \) is redundant iff it loads a value \( v_2 \) from address \( A \), and the last memory load \( L_1 \) from \( A \) loads \( v_1 \), where \( v_1 = v_2 \).

Definition 5.2 (Temporal Store Redundancy) A memory store \( S_2 \) is redundant iff it stores a value \( v_2 \) to address \( A \), and the last memory store \( S_1 \) stores \( v_1 \) to \( A \), where \( v_1 = v_2 \).

Definition 5.3 (Spatial Load Redundancy) A memory load \( L_2 \) is redundant iff it loads a value \( v_2 \) from address \( A_2 \), and another memory load \( L_1 \) loads \( v_1 \) from address \( A_1 \), where \( v_1 = v_2 \), and \( A_2 \) and \( A_1 \) are in the memory range of a data object allocated by a GPU memory allocation.

Definition 5.4 (Spatial Store Redundancy) A memory store \( S_2 \) is redundant iff it stores a value \( v_2 \) to address \( A_2 \), and another memory store \( S_1 \) stores \( v_1 \) to address \( A_1 \), where \( v_1 = v_2 \), and \( A_2 \) and \( A_1 \) are in the memory range of a data object allocated by a GPU memory allocation.

**Temporal redundancy**

To identify temporal value redundancies, our tool reasons about the value generated by each memory instruction instance and compares the last value at the target memory location with the newly generated value. If the two values are the same, our tool records the program counters of two involved memory accesses \( (PC_{old}, PC_{new}) \) as a pair of redundancy and accumulates redundancy metrics with it.

Our tool creates two tables for each GPU kernel to detect temporal value redundancies, as shown in Figure 5.8. The *Last Seen Table* records each thread’s last access value at an address. The *Redundant Pairs Table* records PC pairs involved in the
redundancy. For each access record, our tool first checks whether its effective address is in the Last Seen Table. If not, our tool inserts a record into the table. Otherwise, our tool compares the current value with the last value and reports redundancies in the Redundant Pairs Table if the two values are the same. The address in the Last Seen Table is then updated to the current address.

One challenge arises due to the large number of threads used on GPUs. For instance, the total number of threads in NVIDIA Volta V100 is up to $2^{41}$; the Last Seen Table with all thread records can easily overflow GPU memory. We minimize the size of the Last Seen Table by leveraging the fact that the maximum number of active threads on a GPU is limited by the number of stream multi-processors (e.g., 80 in Volta V100). Thus, upon a BLOCK_EXIT record in the GPU data, our tool removes records of all inactive threads from the Last Seen Table as they are never used again.

Spatial redundancy

To identify spatial value redundancies, our tool intercepts data allocations, and associates memory accesses with data objects allocated in GPU memory. If a memory
access produces the same value as a prior access to the same data object, a spa-
tial redundancy occurs. Our tool records allocation contexts of the data object and
program counters of memory accesses \( \langle C_{\text{data}}, PC_{\text{access}} \rangle \) and accumulates associated
redundancy metrics.

Instead of checking values loaded from or stored to the same effective address,
our tool checks values within the same data object. Our tool leverages memory
snapshots (explained in Section 5.5.4) captured by the runtime system to create a
map from data allocation contexts to allocated object memory ranges. Our tool then
attributes memory accesses to the data objects that enclose the effective addresses of
the memory accesses. Our tool reports the spatial redundancies to data objects as
well as the related memory accesses.

5.5.2 Redundancy Patterns

To provide optimization insights, our tool further characterizes redundancy values
into several typical patterns. Unlike redundancy metrics, our tool associates values
with data objects (e.g., arrays or tensors) to identify various value patterns at both
goarse- and fine-grained levels. We characterize eight pervasive value patterns found
in GPU-accelerated applications and further categorize them into coarse- and fine-
grained patterns. Table 5.3 overviews value patterns residing in popular Rodinia
benchmarks [29] and many applications. We elaborate on each value pattern with
eamples.

Coarse-Grained Value Patterns

Coarse-grained value patterns describe value
characteristics after each GPU API invocation. We define two coarse-grained pat-
terns.
Table 5.3: Various value patterns exist in GPU applications and benchmarks.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Redundant Values</th>
<th>Duplicate Values</th>
<th>Frequent Values</th>
<th>Single Value</th>
<th>Single Zero</th>
<th>Heavy Type</th>
<th>Structured Values</th>
<th>Approximate Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rodinia/bfs [29]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rodinia/backprop [29]</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rodinia/sradv1 [29]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rodinia/pathfinder [29]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rodinia/huffman [29]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Rodinia/streamcluster [29]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Darknet [34]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>QMDPACK [76]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Castro [77]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BarraCUDA [78]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyTorch-Deepwave [178]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyTorch-Bert [179]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAMD [180]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAMMPS [181]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Definition 5.5 (Redundant Values)** A data object \( D \) matches the redundant values pattern at a GPU API \( A \) if \( D \) is written by \( A \) and some or all of \( D \)'s elements are not changed by \( A \).

Coarse-grained value patterns are common in GPU-accelerated applications. One common cause of the redundant values pattern is double initialization of data objects — a data object may be initialized twice with the same values. In such a case, one of the initialization operations is redundant. Section 5.5.5 illustrates an example of this pattern found in PyTorch.

**Definition 5.6 (Duplicate Values)** A data object \( D_1 \) matches the duplicate values pattern with another data object \( D_2 \) if \( D_1 \) and \( D_2 \) have the same values at any GPU API.

The duplicate values pattern occurs across GPU API invocations. For instance, Darknet initializes the weight arrays of each layer on the CPU and then copies them to the GPU via memory copy APIs. These APIs copy duplicate values. One can directly
invoke memory set APIs to initialize all weights on the GPU to avoid CPU-GPU memory traffic.

**Fine-Grained Value Patterns** Fine-grained value patterns are identified based on all accesses to a data object at individual GPU APIs. We define six value patterns in this category.

*Definition 5.7 (Frequent Values)* A data object $D$ matches the frequent values pattern at a GPU API $A$ if accesses to one or more particular values in $D$ exceeds a predefined percentage threshold $T$ of accesses to $D$.

*Definition 5.8 (Single Value)* A data object $D$ matches the single value pattern at a GPU API $A$ if all of $D$’s accessed values are the same.

*Definition 5.9 (Single Zero)* A data object $D$ matches the single zero pattern at a GPU API $A$ if all of $D$’s accessed values are zeros.

The frequent values pattern exposes redundant computation on identical values. One can optimize it with conditional computation, which bypasses redundant computation. One example is Rodinia/huffman, where we observe that most values written to the array histo are zeros. To avoid identity computation, we bypass the computation on this array when zeros are found. The single value and single zero patterns are special cases of the frequent values pattern. They expose additional optimization opportunities, such as contracting a vector to a scalar to reduce memory traffic or applying a sparse data structure or algorithm to reduce computation intensity.

*Definition 5.10 (Heavy Type)* A data object $D$ matches the heavy type pattern at a GPU API $A$ if $D$’s data type is more expressive than the values used in $D$. 
The heavy type pattern identifies opportunities for contracting the value type to reduce memory traffic. As an example, the values in the g\_cost array in Rodinia/bfs are always in the range of int8 according to its input. Thus, demoting int32 to int8 can significantly improve the performance.

**Definition 5.11 (Structured Values)** A data object $D$ matches the structured values pattern at a GPU API $A$ if the values accessed in $D$ and the memory addresses storing these values are linearly correlated.

The structured values pattern exposes the relationship between values and the memory addresses storing these values in a data object. In other words, if the structured values pattern exists, one can infer the values stored in an array using the indices. The linear correlation between values and indices is the most common. As an example in Rodinia/srad\_v1, four arrays $d\_iN$, $d\_iS$, $d\_jW$, and $d\_jE$ store the coordinates of their neighbors, showing the structured value pattern. A typical optimization for this pattern is to compute the values based on the memory addresses (or array indices) to replace more costly memory load or store operations.

**Definition 5.12 (Approximate Values)** A data object $D$ matches the approximate values pattern at a GPU API $A$ if the values accessed in $D$ are floating-point numbers and the values with a mantissa of $K$ bits correspond to some fine-grained patterns.

If approximate computing is allowed, relaxing the exact value patterns to approximate value patterns can expose more optimization opportunities. The hotspot3D code of Rodinia falls into such an example. By controlling the accuracy loss within 2% RMSE [180], one can observe the array tIn\_d with the single value pattern and apply optimizations accordingly.
5.5.3 Value Flow

Unlike existing tools that provide a profile or trace view to present performance metrics, our tool constructs a value flow graph, which visualizes the value changes across GPU APIs to provide performance insights for optimizations. A value flow tracks data objects in a global view: across CPUs and GPUs, as well as across GPU API invocations. In a value flow graph, each node represents a GPU operation, including object allocations, initializations, transfers, uses, and updates. More formally, a value flow graph is defined in Definition 5.13.

Definition 5.13 (Value Flow Graph) A value flow graph $G = (V, E, v_{host})$ is a directed graph, where $V$ is the set of vertices and $E$ is the set of edges, and $v_{host}$ represents any host memory operation.

- Each vertex $v \in V$ represents a GPU API invocation such as GPU memory allocation, memory copy, memory set, or kernel launch.
- An edge $e_{i,j,k} \in E$ exists from $v_i$ to $v_j$ if
  - $v_j$ writes $D_{v_k}$ or $v_j$ reads $D_{v_k}$, where $D_{v_k}$ is a data object allocated by $v_k$,
  - $v_i$ writes $D_{v_k}$,
  - no $v_u$ writes $D_{v_k}$ following the write by $v_i$ and before $v_j$, and
  - $e_{i,j,k}$ is labelled with read/write operations for vertex $v_j$.
- $e_{i,host,k}$ is a sink edge that represents the device to host memory transfer.
- $e_{host,i,k}$ is a source edge that represents the host to device memory transfer.

Figure 5.9 shows an example of mapping a GPU program to a value flow graph based on Definition 5.13. For convenience, we use the line number at which a GPU
cudaMalloc(&A_dev, N * sizeof(int));
cudaMalloc(&B_dev, N * sizeof(int));
cudaMemset(A_dev, 0, N * sizeof(int));
cudaMemset(B_dev, 0, N * sizeof(int));
set_zeros<<<1, N>>>(A_dev, N/4);
set_zeros<<<1, N>>>(B_dev, N/4);
cudaMemcpy(B_dev, A_dev, N * sizeof(int), cudaMemcpyDeviceToDevice);

(a) An example program

(b) A value flow graph

(c) A value flow graph with value patterns.

(d) A vertex slice graph focusing on node 6.

(e) An important graph focusing on node 6.

Figure 5.9: An example of construction and analysis of a value flow graph. Rectangles are GPU memory allocations, circles are GPU memory APIs, and ovals are GPU kernels. The wider the edge, the more bytes accessed. The red color indicates high redundancy, and the green color indicates low redundancy. To facilitate the presentation, we use the line number as the ID for each vertex. Fined-grained value patterns and calling contexts (not shown) are associated with each vertex.
API is called as its ID in the value flow graph. At Lines 1 and 2, we create two vertices representing two allocated data objects. Next, at Lines 3 and 4, we create two vertices for `cudaMemset` invocations. Because Lines 3 and 4 write zeros to `A_dev` and `B_dev` respectively, we create edges from 1 to 3, and 2 to 4. Then, GPU kernels are invoked at Line 5 and Line 6 to write zeros to data object `A_dev` and `B_dev` respectively, triggering two new write edges. Finally, a read edge is created to indicate Line 7 reads data object `A_dev` from Line 5, and a write edge is created to indicate Line 7 writes data object `B_dev` from Line 6.

Our tool associates value patterns with value flow graphs. As shown in Figure 5.9, our tool uses edge colors to represent redundancy and thicknesses to quantify accessed bytes. The size of each vertex is determined by an importance factor, which could be this API’s total amount of invocations or execution time. A value flow graph is context sensitive. At runtime, our tool records the call path of each GPU API invocation and assigns a unique ID to denote this call path. Postmortem, our tool annotates the program source information for every frame on the call path as well as inline frames. Vertices with the same call path are merged to simplify presentation.

When profiling large production applications such as LAMMPS [25], our tool can generate a huge value flow graph. To facilitate the analysis, we describe two features that can help one identify interesting subgraphs to explore.

**Definition 5.14 (Vertex Slice Graph)** A vertex slice graph $G_B(v_u) = (V', E', v_{host})$ is a subgraph of a value flow graph $G = (V, E, v_{host})$ where

- $e_{i,j,k} \in E'$ if $e_{i,j,k} \in E$ and
  - $v_u$ writes $D_{v_k}$ or $v_u$ reads $D_{v_k}$, and
  - $e_{i,j,k}$ is on a valid path that consists of edges that read or write $D_{v_k}$ and
reaches $v_u$ or $v_v$ reaches.

- $v \in V'$ if $v$ is on any edge $e \in E'$.

Figure 5.9d shows that applying vertex slice analysis according to Definition 5.14 on vertex 6 generates a vertex slice graph $G_B(v_6)$ that tracks vertex 6’s inputs and outputs. Vertices that do not affect vertex 6’s value patterns and vertices whose value patterns are not affected by vertex 6 are eliminated.

We use $I(x)$ to represent user-defined metrics that measure the importance of a vertex or an edge. $I_e$ is the threshold for keeping an edge in a graph, and $I_v$ is the threshold for keeping a vertex in a graph. We define important graph using Definition 5.15.

**Definition 5.15 (Important Graph)** An important graph $G_I = (V', E', v_{host})$ is a subgraph of $G = (V, E, v_{host})$ where

- $e_{i,j,k} \in E'$ if $e_{i,j,k} \in E$ and $I(e_{i,j,k}) \geq I_e$
- $v \in V'$ if $v$ is on any edge $e \in E'$ or $I(v) \geq I_v$

We let $I(e)$ be accessed bytes on each edge, and $I(v)$ be the number of invocations of the GPU API represented by each vertex. Let $N$ be the number of nodes in a graph. With $I_e = N/2$ and $I_v = 1$, we can prune the graph in Figure 5.9d and yield the graph in Figure 5.9e with important vertices and edges only. Applying the important graph analysis, our tool trims the original value flow graph of LAMMPS from 660 nodes and 1258 edges to 132 nodes and 97 edges.
5.5.4 Implementation Details

Tracking Data Objects

Monitoring data object allocation and free is necessary for analyzing spatial value redundancies. Our tool uses Sanitizer API to intercept standard GPU memory allocation and free operations to track all active GPU data objects, including their allocated memory ranges and allocation contexts. For custom allocators, we could extend our tool to leverage LLNL’s GOTCHA library [181] or LD_AUDIT [182] to intercept and track allocations. The runtime system passes the list of all active data objects to the analysis thread via the CPU queue. Because our tool’s analysis thread is running asynchronously, its runtime system needs to keep the list up to date for all kernels.

Our tool employs a novel approach: Memory Snapshots to track data objects across kernels efficiently. A memory snapshot contains all active GPU data objects. A new memory snapshot is created when a GPU memory allocation (e.g., cudaMalloc) or free (e.g., cudaFree) occurs; all the memory snapshots are stored in a map.† Our tool assigns a unique ID opid for each snapshot and each kernel launch; opid always increments. The creation of memory snapshots follows two rules.

- Data allocation. When a GPU data object is allocated, our tool forks a new memory snapshot from the latest memory snapshot and inserts the new data object.

- Data reclamation. When a GPU data object is freed, our tool forks a new memory snapshot from the latest memory snapshot and removes the deleted

†These maps could be implemented more efficiently with a partially persistent data structure [183].
Figure 5.10: Following the operations in sequence, the figure shows how memory snapshots are allocated over time. After allocating $a$, $b$, $c$, the analysis thread is woken and tries to analyze kernel 4. It first looks up data objects in snapshot 3, which is the closest to kernel 4. Then it erases all the memory snapshots before 3. Likewise, after 7: Free(b) the analysis thread analyzes kernel 5. After analysis, it erases snapshot 3 but keeps snapshots 5 and 7.

- Kernel Launch. When a GPU kernel instance launches, our tool assigns a new opid as well.

Figure 5.10 shows a detailed example of how our tool maintains memory snapshots.

When the analysis thread investigates the current memory snapshot, it first obtains $opid_k$ of the current kernel launch and then to get the memory snapshot with $opid_m$, where $opid_m$ has the greatest value but smaller than $opid_k$ in the map. After the analysis, our tool removes memory snapshots whose opids are smaller than $opid_k$, except the one with $opid_m$. A global lock is used on the map to avoid data races between application threads and the analysis thread. Since our tool shrinks the map
whenever the analysis is done, snapshot lookup is fast even for complex programs with tens of thousands of memory allocations.

Tracking memory allocations only works for data allocated in global memory. Our tool cannot identify individual data allocated in shared and local memory because there is no explicit allocation API to intercept. As a workaround, our tool treats uses the kernel instance where shared and local memory accesses occur to different shared and local data objects.

Minimizing CPU-GPU Data Transfers

After each GPU operation, our tool updates each involved data object’s value snapshot on CPU. On the one hand, if we copy the entire interval of each data object (i.e., direct copy in Figure 5.11a), we waste time copying values that are not accessed. On the other hand, if we only copy values on the accessed addresses (i.e., segment copy in Figure 5.11c), we need invoke to memory copy APIs many times. Our tool employs an adaptive copy mechanism to switch between different copy strategies. In addition to the aforementioned two strategies, our tool supports a third one—min-max copy (i.e., Figure 5.11b), which copies memory based on the minimum and maximum ad-
dresses across all intervals. Our tool employs the segment copy when the distribution of accessed intervals is sparse and the number of intervals is small, and switches to the min-max copy when the distribution is dense or the number of intervals is large.

5.5.5 Case Studies

In this section, we describe several case studies in detail using value redundancy analysis. We profiled and optimized applications on an NVIDIA A100 GPU (A100) and an NVIDIA V100 GPU (V100). It is worth noting that our optimizations do not introduce any accuracy loss.

Darknet

Figure 5.12: The profile view of our tool shows the calling context of the fill kernel invocation with high spatial store value redundancy. The top pane shows the source code; the bottom left pane shows the full calling contexts; and the bottom right pane displays the redundancy count (SPATIAL.Write.Red) and the redundancy ratio (SPATIAL.Write.Red.Ratio) of each context (e.g., source lines and kernels).

Darknet [24] implements convolution using the lowering method [184], which transforms each image in a mini-batch into an input matrix, utilizes cuBLAS to perform
matrix multiplication with a filter matrix. As shown in Figure 5.12, our tool reports 28.9% spatially redundant stores in a fill kernel with 100% redundancy ratio. The fill kernel is applied before matrix multiplication $A \times B$ to set the output matrix $C$ to zeros. The computation follows the format: $C \leftarrow A \times B + C$, which introduces many spatial redundancies on loading $C$. Thus, we removed the fill kernel invocation and computed $C \leftarrow A \times B$ to avoid redundant loads and stores. This optimization yields a 1.02× speedup for the entire program.

Our tool further pinpoints that 99% spatially redundant loads are in cuBLAS’s matrix multiplication kernel. To investigate the root cause, we obtained the redundancy ratio for each convolution layer individually and found that the first convolution layer incurs the most spatial redundancy. Our tool shows that 50% values loaded from shared memory in this kernel are zeros. This is because cuBLAS uses a general tiling approach for tall-and-thin matrices in the first convolution layer of yolov3-tiny, which causes many zero values in shared memory. For optimization, we employed a fast implementation for tall-and-thin matrix multiplication [185] to better tile matrices. This optimization reduced spatially redundant loads from 41% to 6%, yielding a 1.60× speedup for the matrix multiplication kernel.

Our tool also provides a user-friendly GUI to visualize value patterns and flows with rich information to guide optimization. Figure 5.13 shows an example presentation of our tool’s GUI. The GUI quantifies coarse-grained value patterns on each vertex and edge. For each vertex, one can use its ID to look up its fine-grained value patterns in the value pattern profile. Furthermore, the GUI enables users to explore the value changes of any data object along specific paths. One can use the GUI to inspect an important portion of the graph, which is especially useful for large profiles collected from real application execution. The value flow figure for Darknet has 70
Figure 5.13: A part of the value flow graph for Darknet generated by our tool. When a user hovers the cursor over a vertex, a text box appears to show details such as the vertex’s calling context to help the user locate inefficient code.
nodes and 114 edges. Figure 5.13 highlights two severe redundant value flows: 390 → 392 and 218 → 220 → 1506, which pinpoint unnecessary CPU-GPU communication in Darknet. Darknet initializes an array on CPUs to zeros via function then copies this array from the CPU to the GPU. This copy on zeros wastes memory bandwidth. For optimization, one can use function `cudaMemset` to directly initialize these arrays on the GPU rather than copying from the CPU, which saves 84.2% CPU-GPU memory traffic.

**Quicksilver**

```cpp
bool CollisionEvent(..) : 
  for (int isoIndex = 0; isoIndex < numIsos; isoIndex++) :
    for (int reactIndex = 0; reactIndex < numReacts; reactIndex++) :
      currentCrossSection -= macroscopicCrossSection(
        isoIndex, reactIndex, ...);

double macroscopicCrossSection(
  int isoIndex, int reactIndex, ...) :
  microscopicCrossSection = monteCarlo->_nuclearData->
    getReactionCrossSection(reactIndex, isoIndex, ...);

double NuclearData::getReactionCrossSection(
  int reactIndex, int isoIndex, int group) :
  qs_assert(isoIndex < _isotopes.size());
  qs_assert(reactIndex < _isotopes[isoIndex]._species[0]._reactions.size());
  return _isotopes[isoIndex]._species[0].
    _reactions[reactIndex].getCrossSection(group);
```

Listing 8: Temporal value redundancies in Quicksilver.

Our tool identifies both temporal and spatial value redundancies in Quicksilver.

**Optimizing temporal value redundancies** Listing 8 shows the code that our tool highlights for two temporal redundancies. The first one is the `qs_assert` on line
19 and 20, which account for 20.9% of total temporally redundant loads. Further investigation shows that the function `getReactionCrossSection` enclosing the two assertions is called in a loop nest on line 5. Because `isotope_index` and `_isotopes` are both loop invariant, we can either hoist the assertions out of the loop or remove the assertions in the released version. We removed the two assertions and obtained a $1.10 \times$ speedup for the kernel.

Our tool also pinpoints the epilogue of the function `getReactionCrossSection` and its caller `macroscopicCrossSection` (both shown in Listing 8), accounting for 30.2% of total temporally redundant loads. This is because these two functions are called in a loop nest, introducing redundant local memory store and load operations to spill and restore unchanged values, such as loop trip count (e.g., `numIsos`), to make registers available for the callee. We inlined these two functions into their caller to avoid redundant local memory accesses and obtained an extra $1.10 \times$ speedup to the kernel.

**Optimizing spatial value redundancies** Our tool identifies 61.5% spatial store redundancies in the class constructor for `MC_Distance_To_Facet` (not shown), in which values are initialized to zeros. Spatial store redundancy is high because an array of `MC_Distance_To_Facet` is constructed repeatedly in a loop (`loop1`). There are three data members in `MC_Distance_To_Facet`: `distance`, `facet`, and `subfacet`. However, in `loop1`, `facet` and `subfacet` are never used so we can compress the `MC_Distance_To_Facet` array to a `double` array for `distance` only to reduce spatial store redundancy.

After this optimization, our tool identifies the `distance` array accounting for 10.5% spatially redundant loads in another loop (`loop2`). Our tool reports that a large
Table 5.4: Speedups of LAMMPS on varying numbers of GPUs.

<table>
<thead>
<tr>
<th>GPUs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>lj.in</td>
<td>1.47×</td>
<td>1.31×</td>
<td>1.30×</td>
<td>1.23×</td>
<td>1.04×</td>
<td>1.05×</td>
<td>1.05×</td>
</tr>
<tr>
<td>in.intel.lj</td>
<td>1.37×</td>
<td>1.39×</td>
<td>1.37×</td>
<td>1.33×</td>
<td>1.20×</td>
<td>1.19×</td>
<td>1.08×</td>
</tr>
</tbody>
</table>

double value assigned in loop1 dominates values in the array. Further investigation shows that Quicksilver uses the distance array to save the nearest distance to each facet of a cell in loop1, and finds the minimum value of the array in loop2 and discards the distance array. We optimized the code by fusing the two loops and compressing the distance array to a single variable to hold the minimum value. The two spatial redundancy optimizations yield a 1.12× additional speedup.

**Optimization summary** To summarize, our optimizations guided by our tool on both temporal and spatial value redundancies yields a 1.35× speedup to Quicksilver’s kernel.

**LAMMPS**

Our tool reports 52.3% spatial value redundant stores are in Kokkos::resize kernel under two hot calling contexts, as shown in Figure 5.14. Kokkos::resize is called repeatedly in loop nests to increase the size of multi-dimensional arrays. Resizing an array requires allocating a new piece of memory and initializing it to zero, thus, resulting in spatial store redundancy. Since arrays grow dynamically, LAMMPS defines an array growth factor that indicates how much additional space will be added to a dynamic array each time it resizes. To avoid frequent allocation and initialization, we increased the default factor at the two hotspots from 0.01 to 0.1 and 0.8 respectively.

Table 5.4 shows the speedups of this optimization for two different inputs. With
### Hotspot 1

<table>
<thead>
<tr>
<th>Line</th>
<th>Function Call</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>794</td>
<td>loop at create_atoms.cpp</td>
<td>21.4%</td>
</tr>
<tr>
<td>795</td>
<td></td>
<td>21.4%</td>
</tr>
<tr>
<td>796</td>
<td></td>
<td>21.4%</td>
</tr>
<tr>
<td>797</td>
<td></td>
<td>21.4%</td>
</tr>
<tr>
<td>831</td>
<td>LAMMPS_NS::AtomVecAtomicKokkos::create_atom(int, double*)</td>
<td>21.4%</td>
</tr>
<tr>
<td>795</td>
<td>LAMMPS_NS::AtomVecAtomicKokkos::grow(int)</td>
<td>21.4%</td>
</tr>
<tr>
<td>75</td>
<td>LAMMPS_NS::MemoryKokkos::grow_kokkos(…)</td>
<td>6.0%</td>
</tr>
<tr>
<td>232</td>
<td>Kokkos::DualView(…)</td>
<td>6.0%</td>
</tr>
<tr>
<td>679</td>
<td>Kokkos::resize(…)</td>
<td>6.0%</td>
</tr>
</tbody>
</table>

### Hotspot 2

<table>
<thead>
<tr>
<th>Line</th>
<th>Function Call</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>398</td>
<td>loop at atom_vec_atomic_kokkos.cpp:</td>
<td>30.9%</td>
</tr>
<tr>
<td>398</td>
<td>LAMMPS_NS::AtomVecAtomicKokkos::grow(int)</td>
<td>30.9%</td>
</tr>
<tr>
<td>75</td>
<td>LAMMPS_NS::MemoryKokkos::grow_kokkos(…)</td>
<td>9.6%</td>
</tr>
<tr>
<td>74</td>
<td>LAMMPS_NS::MemoryKokkos::grow_kokkos(…)</td>
<td>7.4%</td>
</tr>
<tr>
<td>73</td>
<td>LAMMPS_NS::MemoryKokkos::grow_kokkos(…)</td>
<td>6.7%</td>
</tr>
<tr>
<td>69</td>
<td>LAMMPS_NS::MemoryKokkos::grow_kokkos(…)</td>
<td>2.1%</td>
</tr>
<tr>
<td>70</td>
<td>LAMMPS_NS::MemoryKokkos::grow_kokkos(…)</td>
<td>2.1%</td>
</tr>
<tr>
<td>71</td>
<td>LAMMPS_NS::MemoryKokkos::grow_kokkos(…)</td>
<td>1.8%</td>
</tr>
<tr>
<td>68</td>
<td>LAMMPS_NS::MemoryKokkos::grow_kokkos(…)</td>
<td>1.2%</td>
</tr>
</tbody>
</table>

Figure 5.14: Two hot spatially redundant stores in LAMMPS due to function Kokkos::resize. Kokkos::resize is invoked from every call to LAMMPS_NS::MemoryKokkos::grow_kokkos. Note: We collapse long function names generated by Kokkos.
void replication_pad3d_backward_out_cuda_template(...) {
    gradInput.resize_as_(input);
    gradInput.zero_();
    ...
}

Tensor replication_pad3d_backward_cuda(...) {
    auto gradInput = at::zeros_like(input, LEGACY_CONTIGUOUS_MEMORY_FORMAT);
    auto gradInput = at::empty_like(input, LEGACY_CONTIGUOUS_MEMORY_FORMAT);
    replication_pad3d_backward_out_cuda_template(gradInput, gradOutput, input, paddingSize);
    ...
}

Listing 9: The redundant values and single zero patterns in Deepwave [178].

PyTorch

We studied three neural networks written in PyTorch [7]: Deepwave, Resnet50, and Bert.

Deepwave  Deepwave [178] provides efficient wave propagation modules that perform seismic imaging/inversion. Our tool first reports 100% memory accesses in function replication_pad3d_backward_cuda matches the redundant values pattern.
void THNN_(SpatialConvolutionMM_updateOutput)(...) {
  if (bias) {
    if (ones->dim() != 2 || ones->size(0)*ones->size(1) < outputHeight*outputWidth) {
      THCTensor_(resize2d)(state, ones, outputHeight, outputWidth);
      THCTensor_(fill)(state, ones, ScalarConvert<int, scalar_t>::to(1));
    }
  }
}

Listing 10: The redundant values and single value patterns in Resnet50 [120]. The array `ones` is resized and initialized to zeros even it is not used later.

Listing 9 shows the problematic code. Our tool also highlights that the input tensor at Line 7 matches the single zero value pattern. `input` is allocated and initialized to zeros at Line 7 and reinitialized again at Line 3 without being accessed in between. The value flow figure has 38 nodes and 49 edges. To optimize the code, we replaced the `zeros_like` function with the `empty_like` function that allocates memory without initialization at Line 8. Our tool reports two other tensors in `replication_pad2d_backward_cuda` and `replication_pad1d_backward_cuda` suffering from the same problem. By optimizing all of them, we obtained 1.07× and 1.04× speedups in the backward phase of the ReplicationPad operator on NVIDIA RTX 2080 Ti and NVIDIA A100 GPUs, respectively. This optimization has been upstreamed to the PyTorch repository.

**Resnet50** Resnet50 [120] is a 50-layer convolutional neural network. We profiled its inference phase using our tool to generate both coarse- and fine-grained value pattern reports for the GPU kernels in Listing 10. The value flow figure has 75 nodes and 223 edges. Our tool reports 14.25MB memory bytes at Line 5 involve redundant values; moreover, our tool reports the single value pattern for the `ones` tensor. The forward phase of convolution computation can be viewed as `input × filter + bias`. 
However, Resnet’s convolution operators foregoes the +bias calculation because its batchnorm operators that follow each convolution operator have already considered bias. Since the ones tensor is only used for accumulating bias, we omitted its allocation and initialization if bias is ignored. Simply adding the two lines eliminates the redundancy and yields 1.02× and 1.03× speedups for convolution layers on RTX 2080 Ti and A100, respectively. This patch has been upstreamed to the PyTorch repository.

**Bert**  
Bert [179] is a transformer-based neural network for natural language processing. Our tool reports the out array in the embedding operator matches the redundant value pattern and 2.8KB bytes are involved. The value flow graph has 101 nodes and 217 edges. With the value flow analysis, our tool shows that padding of out is initialized to zeros in the reset.parameters function, while they are reinitialized in every call to the embedding.maskek.fill function in each iteration. Thus, our tool suggests removing the second initialization, which yields 1.57× and 1.59× speedups for the embedding operator running on RTX 2080 Ti and A100, respectively. This issue has been confirmed by PyTorch developers.

**Castro**

Castro [77] is an astrophysical radiation hydrodynamics simulation code based on the AMReX framework [85]. Castro is an important application supported by Department of Energy’s exascale computing project. We studied Castro’s Sedov example using the inputs.2d.cyl_in_cartcoords input. The value flow graph generated by our tool has 1092 nodes and 1666 edges. Our tool reports that the array slopes matches the redundant values pattern in the GPU kernel cellconslin.slopes.mmlim, which
AMREX_GPU_HOSTDEVICE AMREX_FORCE_INLINE void cellconslin_slopes_mmlim() {
    for (int n = 0; n < ncomp; ++n) {
        for (int j = lo.y; j <= hi.y; ++j) {
            for (int i = lo.x; i <= hi.x; ++i) {
                if (a != 1.0) {
                    slopes(i, j, 0, n) *= a;
                    slopes(i, j, 0, n + ncomp) *= a;
                }
            }
        }
    }
}

Listing 11: The redundant values pattern in Castro [77]. The variable \texttt{a} is mostly 1.0 in our experiment of input \texttt{inputs.2d.cyl_in_cartcoords}. By adding the condition check at Line 5, we can save memory loads and stores.

is a function provided by AMReX. Listing 11 shows this GPU kernel, which computes a multi-dimensional array \texttt{slopes}. We observed that the scalar \texttt{a} at Line 7 is often 1.0, resulting in identity computation and unchanged values in \texttt{slope}. Thus, we can conditionally bypass the computation when \texttt{a} is 1.0, which yields 1.27× and 1.24× speedups for this GPU kernel running on RTX 2080 Ti and A100, respectively. It is worth noting that this optimization improves a library function of AMReX and could benefit all its applications, not limited to Castro. This optimization has been confirmed by Castro developers.

**BarraCUDA**

BarraCUDA [78] is a fast sequence mapping software to map sequencing reads to a particular location on a reference genome. We studied BarraCUDA using a typical input\textsuperscript{¶}. The value flow graph generated by our tool has 30 nodes and 42 edges. Our tool reports the redundant values pattern on array \texttt{global_sequences_index} in

\textsuperscript{¶}Saccharomyces_cerevisiae.SGD1.01.50.dna_rm.toplevel.fa
function `copy_sequences_to_cuda_memory`. BarraCUDA invokes memory copy APIs to copy values from the CPU to the GPU for this array even when it is empty. By adding a size check, we can avoid copying empty arrays and other arrays that only need to be updated when array `global_sequences_index` is changed. Our tool also reports the frequent values pattern with 99.6% zeros in array `global_alns` in GPU kernel `cuda_inexact_match_caller`. This array is copied from a thread-local array on the GPU. We created a `hits` array to record positions that have been updated with non-zero values, and only copy these values. With these optimizations, we obtained a 1.06× kernel execution time speedup and a 1.13× memory time speedup on both RTX 2080 Ti and A100.

**Rodinia CFD and Backprop**

CFD and Backprop are two Rodinia benchmarks. We studied CFD using `fvcorr.domn.097K` input. Our tool reports that the kernel `cuda_compute_flux` has frequent values pattern on array `variables`. We observed that this array is initialized with values within a small range and is unchanged in the first three iterations. Thus, we compress this array to enforce memory accesses within a small range, which greatly increases the data locality. This optimization yields 8.28× and 6.05× speedups on RTX 2080 Ti and A100, respectively.

We studied Backprop with its built-in input. Our tool reports that the kernel `bpnn_adjust_weights_cuda` has single zeros pattern on arrays `w` and `oldw`. We conditionally bypassed floating point computations and writes to these two arrays when they are zeros. In this way, we obtained 8.18× and 1.67× speedups on RTX 2080 Ti and A100, respectively. RTX 2080 Ti achieves a much higher speedup because reducing FP64 can alleviate significant computation workload on this architecture.
with fewer FP64 units than A100.

**NAMD**

NAMD [79] is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. We used NAMD with alanin input. Our tool reports array vdwTypes matches the heavy type pattern. With further analysis, we can demote this array type from int32 to uint8. Our tool also reports redundant values pattern on kernel submitHalf which is repeatedly invoked. However, after investigation, we found that the redundancy is introduced on purpose. NAMD allocates some variables on the device to accumulate global sums and only transfer these values back to the host using the last block of the kernel. In order to avoid extra communication or memset cost, at the end of these kernels, NAMD resets these values to zeros to make sure the next time the buffers are clean. In such scenario, the redundant pattern can not be removed easily. Therefore, we only optimized the heavy type pattern, which yields 1.03× speedup for this kernel, and 1.01× speedup for the entire GPU execution on NVIDIA A100.

**Optimization Tradeoffs**

It is worth noting that our optimizations do not always yield speedups on GPU kernels. For example, we obtained a negligible speedup on GPU kernels of Rodinia/lavaMD. For this benchmark, our tool reports the heavy type pattern on array rA, whose elements are ten values from \{0.1, 0.2, ..., 1.0\}. Our optimization demotes the type from double to unit8_t and reverts it to double when the array is copied to the GPU. The optimization increases the GPU kernel execution time by 2% but reduces the CPU-GPU memory transfer time by 28%. For NAMD and QMCPACK,
our tool reports the redundant values pattern for both, and the heavy type pattern for NAMD. Our optimizations do not yield significant speedups on RTX 2080 Ti and A100 GPUs because the inefficiencies do not occur in costly functions for the given inputs. Since all inefficiencies of these two applications reside in a loop nest whose trip counts depend on input, our optimizations are going to benefit other inputs for which these loops are more important.

5.6 Summary

We demonstrated that our tool can provide rich insights into both individual kernels and whole applications through case studies. In our instruction stall analysis module, we proved that the performance estimate component is accurate in many scenarios. In the future, we can ingest compiler information into this module to perform a more accurate estimate. In addition, for large-scale applications with hundreds of tiny hotspots, we can ingest insights derived from our tool to guide compilers to code transformation.

Our studies have revealed that many GPU applications have value-related inefficiencies. We developed an value redundancy analysis module that identifies and categorizes inefficient value patterns with insightful value flow graphs to guide optimizations of valued-related inefficiencies. Currently, our tool collects program context using the line mapping section in binaries, which does not directly provide straightforward information for interpreted languages. We plan to add more semantic information to our tool’s performance reports to facilitate optimizations. For instance, we can employ the deep learning profiling interface described in Chapter 3 to incorporate the layer/operator annotations.
Chapter 6

Conclusions

The rise of GPU-accelerated systems and applications has created an urgent need to improve programming models, compilers, performance tools, and runtime systems to better support application development and tuning for such platforms. We addressed the challenges by developing innovative GPU performance tools that bridge the gap between discovering GPU performance bottlenecks and identifying effective optimizations.

Our profiler is the first tool that attributes GPU performance metrics to heterogeneous calling contexts that span both the CPU and the GPU. We proposed context-sensitive analysis methods to help users pinpoint performance bottlenecks quickly. Through a set of case studies using our tool’s profile and trace views, we demonstrated that the heterogeneous calling context information could offer helpful guidance to optimize complex GPU code. While context-sensitive information is useful, we observed significant overhead when collecting performance metrics and attributing metrics for complex GPU-accelerated applications, which often involve a large amount of measurement data, deep call paths, and frequent GPU operations. To this end, we devised innovative algorithms and data structures. Our GPU-accelerated analysis mechanism can preprocess measurement data on the GPU to enable a fine-grained analysis of large GPU kernels, which is not feasible in existing instrumentation-based tools. Our metrics derivation mechanism can derive many essential performance metrics using a single run, compared to existing tools that replay GPU kernels multiple times. Our
call path memoization mechanism uses call path identifiers to memoize call paths to reduce overhead of costly unwinding. Our wait-free channels yield low overhead to coordinate frequent communication between application threads and GPU monitor threads.

Context-sensitive analysis focuses users’ attention on costly calling contexts but does not directly associate inefficiencies with performance optimizations. To relieve users of the burden of interpreting performance metrics and analyzing bottlenecks, we developed an automated instruction stall analysis tool. Our tool can identify a range of performance problems using instruction stalls, program structure, and architectural features and suggest effective optimizations to ameliorate these problems. The core analysis of our tool is blaming instruction stalls to root cause instructions using data flow analysis, which is unique among all existing GPU performance tools. Then, we employ GPU binary analysis to recover source information about lines, loops, functions, and inlined functions. Next, our tool matches the inefficiency patterns with optimization suggestions and estimates potential benefits from optimizations using novel performance models. Evaluation results show that our tool could analyze complex GPU code and suggest optimizations that enable users to achieve non-trivial speedups close to speedups estimated by our tool. To analyze deep learning applications composed of both interpreted contexts and compiled contexts, we proposed the deep learning profiler interface (DLPT). We implemented DLPT as a runtime library for PyTorch and curated calling contexts using the information provided by DLPT. To date, we have limited experience applying DLPT to deep learning applications; understanding its utility will require using our tool to analyze a larger set of deep learning codes in

In addition to the instruction stall analysis tool that identifies computation inef-
iciencies, we developed a value redundancy analysis tool that examines the semantic content of computation inputs, outputs, and effects. This gives unique insights out of the scope of existing GPU performance tools that do not analyze values. The redundancy analysis tool first calculates redundancy metrics at each GPU memory instruction. Then, our tool characterizes redundancy values into several typical patterns and associates patterns with GPU data objects. Finally, our tool generates a value flow graph that visualizes the value changes across GPU APIs to provide performance insights for optimizations. We showed that the value flow can provide insights into value-related inefficiencies for many benchmarks and applications, including performance problems that cannot be easily reasoned using existing tools. For instance, we fixed two performance problems in well-optimized operators in PyTorch and committed our fixes to the PyTorch repository.

There are still limitations in our tools in some aspects. Some problems are caused by missing components in existing hardware to efficiently collect important performance metrics. Today, NVIDIA is the only vendor whose GPUs support fine-grained measurement using PC sampling. Without hardware support for fine-grained performance measurement, application developers must use the "guess and check" strategy for performance optimization, which significantly complicates the analysis and optimization of large-scale applications. We hope that future Intel and AMD GPUs will be equipped with comparable instruction sampling units as NVIDIA GPUs. Also, there is a lack of a hardware counter multiplexing mechanism to monitor many metrics in a single run. We worked around this issue by deriving software metrics, but some important metrics can only be measured, such as the achieved occupancy rate. Moreover, it would be useful if additional information could be associated with each instruction. For example, if an instruction is stalled because of memory dependency,
we want to identify on which memory hierarchy (e.g., L1, L2, or page fault) it is stalled.

There are also limitations caused by deficiencies in software. Some issues can be fixed in our tools. We listed a few issues below that could be addressed in the short term. On the measurement side, our tool’s profiler creates an operation thread that gathers invocation records and GPU activities and dispatches them to the corresponding threads. This solution can induce a burden on the operation thread if many GPU monitor threads and application threads exist. This issue could be addressed using wait-free queues [129] suitable for multiple consumers and producers. For binary analysis, our tool’s backward slicing component analyzes each instruction in parallel, causing a huge memory footprint for large functions. It would be more space and time-efficient to use basic-block summaries for backward slicing. For performance analysis, our tool does not support the analysis of several new instruction features, including branch divergence, warp synchronization, and tensor core instructions. We can enhance our tool to diagnose these features.

Other software issues need to be addressed in vendor-provided facilities. First, vendor-provided measurement APIs have high overhead for some complex GPU code. For example, when monitoring AMR-Wind [122], NVIDIA’s CUPTI can cause up to 50% overhead even if only lightweight performance metrics are collected. Also, when collecting performance metrics using vendor-provided instrumentation APIs, we can only execute the kernels that have been instrumented but not the original kernels. This limitation prevents us from switching between instrumented and original code to reduce overhead. Last, NVIDIA does not expose binary analysis APIs to outside users; parsing text files generated by the command line `nvdisasm` tool is error-prone and costly.
In the past, we have collaborated with hardware vendors on solving some issues to push the boundaries on GPU hardware and software. For instance, we found and reported that NVIDIA’s CUPTI incurs thousands of times overhead due to problematic memory management, runtime DWARF parsing, and excessive synchronizations. And we have offered many suggestions for NVIDIA’s Sanitizer API [64] since its release, proposing new functionality, including patches at GPU thread begin/end, information about memory accesses, and GPU memory/stream handles for instrumentation-based tools. In the future, we will keep collaborating with hardware vendors to fix these issues and benefit the community.

6.1 Future Work

Here, we sketch possible extensions to our performance tool.

Measurement and analysis of emerging accelerators With the approaching end of Moore’s law, we are seeing an explosion of novel computer architectures that advance the capability to solve challenging scientific and technical problems. Emerging accelerators also provide new opportunities and challenges for programming models, runtimes, and tools. Beyond GPU-accelerated platforms, we also plan to explore the challenges faced by developers for harnessing the power of other emerging architectures, such as FPGA and domain-specific processors (e.g., Google’s TPUs [186] and Intel’s Loihi [187]). These accelerators often combine hardware and software considerations tightly to effectively exploit new architectures. For example, computing platforms that are dedicatedly designed for AI applications focus on specific customization for AI, constructed with a large number of computing units organized to support common matrix-matrix or matrix-vector multiplications [188]. One potential
project can be investigating and proposing performance monitoring units on different architectures. Unlike CPUs and GPUs, there are only limited performance monitoring units in these specialized processors. For this reason, we can use simulators to investigate what information is important to performance and what hardware support can recover key insights for application developers.

Monitoring long-running applications Our tool profiles GPU-accelerated applications and attributes performance metrics to full calling contexts. At runtime, we maintain a calling context tree data structure to aggregate performance metrics that belong to the same call path. Also, we employed a sparse representation for metrics in each node to store non-zero metrics only. In this way, our tool significantly reduces the size of profiles compared to existing tools.

There exist new challenges regarding memory usage and scalability when profiling long-running applications with multiple execution phases and time-sensitive behaviors. As a result, merging metrics to a single calling context tree may not provide enough insights. We could periodically extend our measurement substrate to flush measurement data from memory to disk. In addition, though profile data can be aggregated, tracing data cannot. Beyond writing profile data periodically, we can extend existing mechanisms [189] to compress the tracing data or sample events in traces. Last, the growing size of profiles and traces imposes burdens for users to pinpoint performance bottlenecks. Recently, some research [190, 191] targets performance differences between function invocations and between different processes. We can investigate this problem for long-running GPU-accelerated applications on thousands of GPUs.
Offloading program analysis on GPUs  Our measurement substrate incorporates an innovative design that offloads memory interval analysis to the GPU to accelerate processing. Our design can be further extended to analyze other memory-related metrics. For instance, to count the access count of each memory address, which is important for fine-grained cache utilization analysis, we could design a slightly different parallel algorithm like the interval merge algorithm. Since each memory access could only be 1, 2, 4, 8, and 16 bytes long, we can dispatch access records of different sizes to dedicated buffers. For each buffer, we can apply the GPU run length encoding [192] algorithm using CUB [75]’s sorting and prefix sum utilities. Likewise, reuse distance analysis [65] and taint analysis [193] can also be offloaded on the GPU.

We can build a portable library that consists of common performance analysis substrates. This library can be composed of three components: configuration, run-time analysis, and offline analysis. During the configuration stage, users can specify what analysis is to apply for the upcoming GPU kernel, how many GPU blocks will be used, how many GPU buffers can be reserved, and whether the analysis should be on the fly or not after the kernel is done. Unlike CUB, which provides building blocks for parallel algorithms only, our library wraps up complex analyses such as cache efficiency, address counts, and merged intervals so that users do not need to write GPU code. Offline analysis refers to costly binary analysis algorithms such as register liveness analysis and backward slicing. Previously, these algorithms are only parallelized on the CPU [194], without utilizing massive threads and high bandwidth memory on the GPU. To refactor these complex algorithms, we need first to transform the original data structures to GPU-friendly data structures, offload the most compute-intensive part to the GPU, and restore analysis results on the CPU.
Feedback-directed Optimizations  We plan to develop an end-to-end framework that profiles and optimizes GPU-accelerated applications. Our previous research has built a solid basis for profiling general GPU code and deriving useful insights. The next step aims to gather more detailed information about program semantics and context and automatically transform binaries or program sources to gain performance speedups.

Today, emerging GPU-accelerated code can present different behaviors under different inputs. For example, in graphics workload, the performance can be impacted by conditions and environments. Previous work [195] instruments Vulkan programs to investigate such behaviors. There also exists research [196] that dynamically optimizes GPU binaries using metrics collected from instrumentation. Due to the widely existing dynamic features, we believe that it is a good opportunity to explore feedback-directed optimization tools for GPU-accelerated applications.

Previous domain-specific JIT compiling tools, such as TorchScript [104] for deep learning and NUMBA [197] for high-performance computing, have notable shortcomings. These tools rarely optimize operators based on time-consuming code and lack GPU-level optimizations. In contrast, our tool analyzes fine-grained information collected on GPUs, providing thorough characteristics of GPU programs being monitored. We target optimizing the performance information at both application and GPU kernel levels. Existing JIT compilation engines apply optimizations as possible and do not check if any optimization brings side effects. If we can compile multiple code versions, break down the computation graphs into different pieces, and compare the time of each part, it could make the optimization decision more accurate. Moreover, none of the JIT compilers apply a transformation to use GPU programming model-specific features such as asynchronous memory transfer or concurrent GPU
kernels. Without the transformation, a program can be bounded by CPU time and unlikely to utilize all GPU resources.

Likewise, we want to address issues in profile-guided optimization for GPU kernels. Our tools aim to optimize performance using hardware counter and instrumentation values. Feedback directed compilers map instruction metrics to source code to guide performance optimization. Our work has laid a solid foundation for analyzing GPU binaries, recovering information about lines, loop nests, and inline functions in GPU code. Our existing work also provides solutions to overcome the overhead of collecting measurement data. Existing fine-grained GPU tools replay the measurement of GPU kernels multiple times to collect performance metrics. As a result, the profiling overhead is high for applications with many kernels where it isn’t easy to extract single kernels. Our metrics derivation method can be used to reduce the number of passes; our hierarchical sampling methodology can be used to reduce the amount of data, and our GPU-accelerated data processing method can be applied.

One challenge remains that we need to investigate further: performance models on CPUs and GPUs are different. We have shown that GPU-accelerated applications can be optimized with various methods different from those used to optimize for CPUs. For instance, current profile-guided optimizations for CPU programs mainly consider intra-thread performance, while GPUs are equipped with thousands of threads for parallel processing and latency hiding. Other differences come from function units and memory hierarchy. Memory instructions for different hierarchies result in very different latency due to access patterns or the number of requests. On GPUs, instruction dependencies can be specified through software barriers (AMD and NVIDIA). Thus, adjusting instruction order can impact performance significantly. These performance-critical factors can be addressed through new compiler passes complementary to ex-
isting ones.

**Fine-grained GPU memory profiling**  Compared to CPU memory, the size of on-chip GPU memory is limited. Even NVIDIA TESLA GPUs, designed for data centers and supercomputers, only have up to 80GB memory [198], which is not enough to hold all data and parameters used in large applications. Some hardware and software techniques are proposed to solve this problem. The unified memory approach, for example, enables the GPU and the CPU to share the same address space without explicit data movement. However, GPU applications suffer from significant performance loss due to the page fault triggered in GPU kernels without any fine-tuning for unified memory applications. There also exist several approaches designed for specific workloads. Superneurons [199] and its follow-up research [200] incorporates runtime recomputation and memory pool management to reduce memory cost, which is tightly bound to applications.

Existing GPU profilers such as Nsight Systems can monitor GPU memory usage at a high level. Still, they do not correlate GPU memory usage with GPU APIs, providing little insight for developers. To the best of our knowledge, the only fully functional GPU profiler is Tensorboard [175], which tracks used and free memory across timelines and presents the specific GPU kernel and data shape at each timestamp. Tensorboard obtains all the memory information via source code instrumentation of Tensorflow. Therefore, this approach is intrusive and costly to extend to other applications.

We aim to build a trace view to show memory metrics across timelines. We are interested in the total allocated memory, the memories accessed in a specific GPU API/kernel, and the memories that are alive after each GPU API/kernel. In addition,
we also plan to create a profile view to show the memory accessed at each kernel and GPU instruction/line. Ideally, it provides both the maximum and the total amount of memory among all the different data objects. Our GPU-accelerated data processing method can accelerate the memory access measurement part. We can implement a problem identification module that analyzes the trace view and outputs the most significant performance issues for data objects to provide optimization insights.
Bibliography


[169] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, “Exploiting


