EFFICIENT PERFORMANCE EVALUATION FOR HIGHLY MULTI-THREADED GRAPHICS PROCESSORS

BY

SARA SADEGHI BAGHSORKHI

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei W. Hwu, Chair and Director of Research
Professor William D. Gropp
Associate Professor Nacho Navarro
Professor David A. Padua
Associate Professor Sanjay J. Patel
ABSTRACT

With the emergence of highly multithreaded architectures, an effective performance monitoring system must reflect the interaction between a large number of concurrent events, and associate the overall effect of individual events and inefficiencies to the operations in the application source code. The state-of-the-art performance counters in highly multithreaded graphic processors currently do not provide this level of precision. Although fine-grained sampling of performance counters after each source-level operation could potentially achieve the desired precision, the high frequency of sampling required will likely cause too much distortion to the actual application behavior and make the sampled counter values inaccurate.

In this thesis, I present a novel software-based approach for monitoring the memory hierarchy performance in highly multithreaded general-purpose graphics processors. The proposed analysis is based on memory traces collected for small snapshots of application execution. A trace-based memory hierarchy model with a Monte-Carlo experimental methodology generates statistical bounds of performance measures in the presence of non-uniform thread interleaving and data sharing in a highly multithreaded execution environment. The statistical approach overcomes the classical problem of disturbed execution timing due to instrumentation. The approach scales well as I deploy a minimal sampling technique to reduce the trace generation overhead and model simulation time.

The proposed scheme also keeps track of individual memory operations in the
source code and can quantify the amount of their contribution to detrimental effects on memory system performance. A cross-validation of the model results shows close agreement with the values read from the hardware performance counters on an NVIDIA Tesla C2050.

I later use the predicted memory hierarchy performance statistics in an analytical model to identify performance characteristics of a kernel and its expected execution time. To account for the systematic error present in the predictions, I approximate the error function and express a range of potential true execution times for each predicted value.
To my parents and my grandmother
I would first like to express my deepest gratitude to my adviser Professor Wenmei Hwu. Over the past six years, he provided me with tremendous support and enlightened my perspective on compilers, computer architecture and applicable research. This work would have not been possible without his inimitable expertise and generous patience and support.

I would like to thank my doctoral dissertation committee for their constructive comments: Professor William Gropp and Professor Sanjay Patel provided me with valuable feedback during early stages of this work, which helped me start off in the right direction. Professor David Padua was a great asset during the course of my graduate studies; I learned a lot from him when I took my first courses on compiler design and parallel programming. His expertise and technical insights into my thesis work were extremely helpful. I thank Professor Nacho Navarro who has always provided me with insightful feedback and tremendous encouragement.

I would like to thank Matthieu Delehaye for his contribution to the infrastructure that this work is built upon, specially the Clang front-end for parsing CUDA applications. I specially thank my friend and colleague, Dr. Isaac Gelado for all the stimulating discussions.

I thank Professor Marc Snir, Dr. Yuri Dotsenko and Dr. Naga Govindaraju, from whom I have learned a lot during various stages of my doctoral studies.

Many thanks go to Ms. Marie-Pierre Lassiva-Moulin and Ms. Laurie Talkington for their valuable administrative help.
I am grateful to my friend Behzad for his companionship and support that carried me through difficult times.

I would like to dedicate this dissertation to my parents, Maryam and Abdolreza, and my grandmother, Saltanat, to whom I am forever indebted for their perpetual love and support.
# TABLE OF CONTENTS

LIST OF TABLES ......................................................... ix

LIST OF FIGURES ..................................................... x

LIST OF ABBREVIATIONS ........................................... xii

CHAPTER 1 INTRODUCTION ........................................... 1
  1.1 Limited Performance Visibility ............................... 2
  1.2 Shortcomings of Sampling-based Schemes ................... 3
  1.3 Limitations of Full System Simulation ....................... 4
  1.4 The Significance of Source Level Feedback ................. 5
  1.5 Sources of Uncertainty ........................................ 6
  1.6 Contributions and Organization ............................. 9

CHAPTER 2 BACKGROUND AND RELATED WORK .................... 11
  2.1 The Monte Carlo Method ..................................... 11
  2.2 The Bayesian Method ......................................... 14
  2.3 The Graphics Processing Unit ............................... 16
  2.4 Related Work .................................................. 21

CHAPTER 3 A HIGHLY MULTI-THREADED ARCHITECTURE MODEL ........................................... 26
  3.1 Throughput-oriented Computing ............................. 26
  3.2 Hardware Thread Scheduling ................................ 27
  3.3 Bulk Instruction Scheduling ................................. 27
  3.4 Throughput-oriented Memory System ........................ 29
  3.5 The Complete Model Overview and Use-case Realization .... 31
  3.6 Microbenchmarks .............................................. 32

CHAPTER 4 THE STOCHASTIC MEMORY HIERARCHY MODEL ........ 39
  4.1 Spatial and Temporal Locality ............................... 39
  4.2 The Monte Carlo Approach ................................... 41

CHAPTER 5 SOFTWARE FRAMEWORK ................................. 48
  5.1 Instrumenting the Kernel .................................... 48
  5.2 Recording Traces .............................................. 52
LIST OF TABLES

2.1 NVIDIA performance counters used to cross validate the predicted L1 and L2 caches hit ratios  ................................ 19
7.1 Dynamic load count per static load in SpMV kernel  .................. 79
8.1 Summary of predicted performance information for theStencil kernels ......................................................... 97
8.2 Comparison of the probabilistic relative error of the stencil kernels – reported probability for each pair of $X$ and $Y$ correspond to $P(X \leq Y)$  ........................................ 98
LIST OF FIGURES

2.1 Block diagram of NVIDIA’s G80 graphics processor . . . . . . . 17
3.1 A Highly Multithreaded GPU Model . . . . . . . . . . . . . . . 30
3.2 Array of linked-lists used in microbenchmarks . . . . . . . . . 32
3.3 Average memory access time for different stride values with a
fixed list size and number of memory access . . . . . . . . . . . . 34
3.4 Average memory access time for different list size values with
a fixed stride and number of memory access . . . . . . . . . . . . 36
4.1 Random points mimic schedule deviation for simultaneously
scheduled thread blocks . . . . . . . . . . . . . . . . . . . . . . . 43
4.2 The stochastic memory hierarchy model . . . . . . . . . . . . . 44
4.3 Probability distribution for main memory latency in LBM . . . . 46
4.4 Probability distribution for main memory latency in SpMV . . . 46
5.1 Organization of the sampling buffer . . . . . . . . . . . . . . . 49
6.1 Benchmarks suite used for experiments . . . . . . . . . . . . . 57
6.2 Probability distribution for the L1 and L2 hit ratios - dense
matrix multiply . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.3 Probability distribution for the L1 and L2 hit ratios – FFT . . . 60
6.4 Probability distribution for the L1 and L2 hit ratios – Black
Scholes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
6.5 Probability distribution for the L1 and L2 hit ratios – sparse
matrix vector multiply . . . . . . . . . . . . . . . . . . . . . . . . 62
6.6 Probability distribution for the L1 and L2 hit ratios – merge sort . 63
6.7 Probability distribution for the L1 and L2 hit ratios – lattice-
Boltzman method . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.8 Probability distribution for the L1 and L2 hit ratios – stencil . . 65
6.9 Probability distribution for the L1 and L2 hit ratios – merge
sort (global) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.10 Sampling error for L2 read hit ratio – sparse matrix vector
multiply . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.11 Prior probability density function of the relative error . . . . . . 70
6.12 Probability distribution of systematic error for predicted L1 and L2 hit ratios ........................................... 73

7.1 Breakdown of memory latency for static loads in LBM ............... 75
7.2 Breakdown of memory latency for static loads in Stencil ............. 76
7.3 Breakdown of memory latency for static loads in SpMV ............... 78

8.1 Scheduling of memory and compute cycles ................................. 85
8.2 Serialization of concurrent memory accesses ............................. 87
8.3 Scatter plot of predicted versus measured execution times .......... 90
8.4 Posterior probability density function of the relative error .......... 91

A.1 Expression simplification for Example 1. (a) Initial expression tree. (b) Distributing integer division. (c) Distribution is carried on along the path that contains free variables; constraint \( N = k \times 2^J \) is bound to free variable \( N \). (d) Multiplication is distributed over the add operator. (e) Multiplication and integer division cancel each other out. (f) The simplified expression tree. .................................................. 110
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>MLP</td>
<td>Memory Level Parallelism</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread Level Parallelism</td>
</tr>
<tr>
<td>WLP</td>
<td>Warp Level Parallelism</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction Multiple Thread</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single Program Multiple Data</td>
</tr>
<tr>
<td>PDF</td>
<td>Probability Distribution Function</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>IRM</td>
<td>Independent Reference Model</td>
</tr>
<tr>
<td>LLN</td>
<td>Law of Large Numbers</td>
</tr>
<tr>
<td>CLT</td>
<td>Central Limit Theorem</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correction Code</td>
</tr>
<tr>
<td>PTX</td>
<td>Parallel Thread Execution</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Graphics processing units (GPUs) traditionally had highly specialized programming models and interfaces that limited the ability of developers to easily map general-purpose applications to these platforms. With the emergence of programmable GPUs along with Compute Unified Device Architecture (CUDA) [1] and Open Computing Language (OpenCL) [2] programming interfaces, GPUs made their entry into the general purpose computing community. Developers started to have the standard programming interfaces and architectural features to quickly port programs to a platform with a massively parallel, GPU-based coprocessor.

Introduction of the newer generations of GPUs such as Fermi [3] created another inflection point in use of GPUs in general purpose and scientific computing. Graphics processors are traditionally optimized for throughput oriented workloads, which allowed early GPUs to not include data caches. Newer GPU generations took a step forward by introducing a new memory architecture that supports cache organizations to boost the performance of applications which have irregular and indirect accesses to data; small per-core L1 caches capture inter-thread reuse, and larger unified L2 caches exploit inter-core sharing. This is a significant step forward to better support a more diverse set of workloads and reduce some of the performance discrepancies that previously existed. Furthermore, supporting Error Correction Code (ECC) through the whole memory hierarchy and accelerating double-precision performance have enhanced the GPU programming ecosystem.
1.1 Limited Performance Visibility

The above improvements have made GPU computing users to take a more serious look at the significant computational power provided by GPUs. Yet, efficiently programming many-core GPUs remains intellectually challenging and requires a significant amount of effort. The major challenge in programming GPUs is usually in keeping the high-bandwidth computational processing elements well utilized by providing them with the required data in a timely manner. To achieve this, it is important for the GPU applications to make effective use of the GPU memory hierarchy. Therefore, information about interaction of an application with the memory system is necessary to optimized memory accesses and its performance.

Currently developers need to manually examine the GPU kernel source code and identify the sources of performance degradation such as long latency loads on the critical path of a tight loop. Much of this is done through guesswork and trial-and-error.

Furthermore, multiple levels of concurrency and the large number of concurrent events such as the in flight memory operations, reduce the developer’s or even the optimizing compiler’s performance visibility. A highly multithreaded GPU acts like a complex, nonlinear system with tightly coupled dynamics. A microarchitectural event such as a memory load should be evaluated in a meaningful context with regard to other events from the co-existing threads.

Therefore, the performance evaluation process is very time-consuming for experienced application developers and much more difficult and tedious for less-informed programmers. With millions of units currently in use, graphics processors have become the largest deployment of massively parallel systems. As more users experiment with graphic processors, the larger market is going to keep their cost down. The snowballing phenomenon will in turn expand the GPU computing
community and necessitate development of techniques that systematically analyze the source code in the context of a highly multithreaded execution environment to replace the manual investigation of the code.

1.2 Shortcomings of Sampling-based Schemes

Current GPUs’ level of support for application developers, compiler designers and system architects to investigate the behavior of source code segments that cause performance bottlenecks has been limited. While they provide a set of performance counters that collect raw statistics for microarchitecture events, e.g., the overall number of misses for the L1 and the L2 caches, the counter values cannot be monitored or sampled during application execution.

Likewise, support for time-based sampling is provided through application programming interfaces or profiler triggers [1] that can be inserted in the code at user level. For example, a call to the NVIDIA’s profiler API \_prof_trigger will increment the value of one out of the eight hardware counters enabled. While a time-based sampling built on top of these interfaces can help identify the hotspots in the code, a fine-grain sampling approach is not feasible because its high sampling rate will likely affect the accuracy of the profile data. On the other hand, infrequent sampling will increase the number of blind spots and reduce the precision of the measurement.

Profiling the performance of highly multithreaded applications is not confined to recently introduced high performance accelerators. Salapura et al. [4] discuss the significance of the feedback derived from the performance statistics in high performance computing systems such as Blue Gene and propose a new performance counter architecture that scales better with the number of concurrent events in a highly multithreaded system.
In this thesis, I propose an alternative software-level support to capture the performance statistics with respect to the highly parallel interleaved memory system in the presence of large number of concurrent events. The proposed approach does not require the instrumented application to have the same execution timing as the uninstrumented version. As a result, it does not suffer from the same limitations as the traditional profiling techniques.

Most highly multithreaded applications are memory bound and their performance is strongly dependent on efficient use of the memory subsystem. Therefore, the focus this thesis is on capturing the existing inefficiencies in a GPU kernel with respect to the graphics processors memory hierarchy.

1.3 Limitations of Full System Simulation

An alternative to hardware support performance monitoring is the full system simulation. Simulators work by emulating the full processor(s) architectural features. Depending on the accuracy of the simulation, they can be often excessively slow.

Cycle accurate simulators [5, 6, 7] reconstruct highly detailed models of system components and accurate timing information. The downside is that they can only emulate a few cycles of the actual processor execution per second. The slowdown factor is even more blatant when simulating a highly multithreaded processor with high number of in flight threads and states to keep track of. Therefore, the high execution cost of cycle accurate simulations, particularly for fine grain highly multithreaded systems such as graphics processors, precludes them as practical options to collect performance monitoring information.

Functional simulators [8], on the other hand, can simulate thousands of instructions per second at the cost of losing cycle accuracy and timing information. The direct utilization of these simulators for performance tuning of memory operations
in GPU application kernels will result in inaccurate performance profiles.

The idea presented in this thesis is based on use of a subset of instruction traces including the memory addresses. But unlike functional simulators, arrival ordering of memory operations is reconstructed based on a stochastic execution model of threads in the GPU. The proposed memory hierarchy simulation model runs significantly faster than traditional simulators and provides a good approximation to actual arrival ordering of memory operations at different levels of GPU memory hierarchy. Therefore, the proposed scheme provides capabilities for fine grain memory efficiency analysis of the GPU application by estimating the latency for accessing the memory and modeling all cache levels in the memory hierarchy in a cost-effective manner.

1.4 The Significance of Source Level Feedback

As mentioned earlier, knowledge of an applications interaction with the memory systems is crucial for tuning the performance. This information is most useful to the application developer or the optimizing compiler when it is presented in terms of specific memory accesses (locations) at source code level.

Limited hardware support in current graphics processors makes it difficult to provide this level of feedback to application developers and makes it difficult to determine which instructions cause accesses that poorly utilize the memory hierarchy.

Some processors such as Intel’s Itanium [9] have more support for performance monitoring. For example, hardware counters report the last memory address that has triggered a cache miss. The Itanium processor also provides means to limit counting of cache misses to a user-determined region of memory. In a highly multithreaded GPU, interleaved execution of a large number of threads makes it
difficult to determine – via hardware performance counters – which instruction has caused the miss. Therefore, it is not feasible to provide a programmer with an accurate idea of which program memory operations exhibit the worst memory efficiency behavior.

With the software-level approach for performance monitoring that I propose, source code level feedback is provided to a programmer about memory operations and data structures that exhibit a poor memory efficiency behavior. Source code locations of memory operations (that trigger microarchitecture events) are propagated through the memory hierarchy simulation model. Therefore, the proposed performance modeling framework is capable of producing precise and high resolution profile statistics.

1.5 Sources of Uncertainty

When analyzing complex systems or phenomena, complete agreement between the model assumptions and the properties of the system being analyzed is almost infeasible. This deviation between a model and the corresponding real system – a GPU in this work – can originate from the following factors:

1. Limited knowledge exists about the system or the phenomena being studied.

2. The model assumes a simplified representation of a system. The purpose of this deliberate simplification that is enforced when constructing the model is to reduce evaluation cost, e.g., the time of analysis.

In the course of this study, the first factor becomes relevant when specific design parameters of the GPU hardware and software cannot be reliably determined. For example, the exact scheduling scheme for thread blocks of a kernel is not disclosed by the GPU vendors. The amount of perturbation in terms of the actual
instructions being executed by the GPU hardware – after modification and optimization performed by the vendor’s compiler – is another source of discrepancy.

Unlike cycle accurate simulation approaches [5] discussed earlier, the proposed modeling framework does not account for all microarchitectural details of GPU. The choice of this source of uncertainty that belongs to the second category, was deliberate to make the analysis time tractable; execution cost of traditional simulation methods does not scale well in a highly multithreaded environment.

In this work, I devised two approaches to reduce the detrimental effect of these uncertainties on the quality of the results generated by the performance modeling framework:

1. I use a Monte Carlo experimental methodology – a classic statistical method – to account for lack of knowledge and randomness in a GPU execution environment, e.g., the scheduling and the order of memory requests arriving at data caches.

   The GPU execution model under this approach includes both deterministic and stochastic components. The deterministic component accounts for known events and properties of the GPU, while the stochastic part reflects the uncertainty or lack of knowledge related to the unknown events or behaviors. For example, this statistical approach allows the proposed modeling framework to overcome the problem of lack of timing information, e.g., exact ordering of memory events.

2. I use a Bayesian approach to account for the uncertainty factors that the discussed Monte Carlo method cannot compensate for. The Bayesian analysis assumes that all uncertainties are a result of lack of knowledge. It then focuses on the observable quantities or a set of outputs from the model. It then based on a set of subjective probabilities corresponding to each of the
observed outputs, expresses uncertainties in a system. Following the same approach, I estimate the systematic error function for the predicted performance statistics.

As a result of the above efforts, in spite of allowing for some level of inaccuracy, the model serves its purpose to provide meaningful and statistically sound results.

Using statistical models and analysis to compensate for lack of information in modeling computer systems and memory hierarchy is an established approach, specially when dealing with large amount of input data, e.g., large traces. A widely studied example is the Independent Reference Model (IRM), which is a memoryless model for analyzing the behavior of the I/O page references [10]. In the absence of accurate and detailed traces for the storage system, IRM assumes that at each discrete point in time, exactly one page is referenced with a probability, $p_i$, that is derived based on the number of times that the page is observed in a stream of sparse and less detailed traces. IRM is very simple with respect to the fact that it does not account for the exact ordering of the page reference pattern, but at the same time it retains the identity of a page via the probability $p_i$.

I follow a similar approach with respect to analyzing the behavior of memory operations in a GPU system. In the absence of exact information regarding the ordering and scheduling of instructions, I reconstruct enough statistically correct information to capture the overall behavior of memory operations with respect to the memory system.
1.6 Contributions and Organization

The contributions of this work are:

1. I propose a stochastic model of memory hierarchy for highly multithreaded execution environments: While major configuration parameters for the underlying hardware (graphics processor) are extracted through microbenchmarking as discussed in Chapter 3, I use a Monte Carlo approach (randomized trials) to capture the sources of non-determinism in the simulation model, e.g. the relative arrival ordering of the memory requests at each level of the memory hierarchy. This stochastic approach is presented in Chapter 4.

The results that are presented in Chapter 6 confirm that the proposed approach yields fast convergence with respect to randomized trials with a 95% confidence interval of 0.02 and high accuracy when compared against values read from the hardware performance counters.

2. I devise an adaptive trace collection mechanism that gathers enough memory addresses to capture the intra-thread, inter-thread and inter-core interactions within memory accesses issued from a set of independent but concurrently running threads (thread blocks).

The generated traces are later combined together through the stochastic memory hierarchy modeling framework to predict miss rates for the L1 and L2 caches and measure the expected latency of the main memory.

The trace collection is implemented as a source to source compiler transformation module that seamlessly inserts probes inside the GPU kernel (device code) and the required routines to handle trace buffers in the surrounding code (host code) without any user intervention. The trace collection
mechanism incurs modest execution overhead and results in minimal user intervention. This is discussed in Chapter 5.

3. The proposed approach can identify and isolate the behavior of specific memory operations in an application with respect to the memory hierarchy and in the context of other contemporary memory operations in the system. This information can later be used to target optimizations such as tiling for locality, data layout and thread coarsening for specific code segments. In Chapter 7, I show how based on the predicted statistics for the memory hierarchy efficiency, the performance of individual memory loads (expected latency) can be identified.

Based on this information, in Chapter 8, I devise a tractable analytical model and identify the critical path with respect to the execution time of a kernel. The latency of the critical path determines the expected execution time for a kernel. Based on the results presented in Chapter 8, the model predicts the execution times for the studied GPU kernels with an average relative error of 16%.

I also propose a method to estimate the systematic error of the performance predictions. This is discussed in Section 6.4. Based on the estimated error function, I compare and evaluate the probabilistic merit of a kernel with respect to other kernels in Section 8.2.
2.1 The Monte Carlo Method

The Monte Carlo method [11] is a technique to approximate solutions to a complex, nonlinear system that involves more than a few uncertain input parameters. For example, in mathematical physics, problems that involved smaller number of particles were solved through systems of ordinary differential equations which are apparatus to study physical phenomena in classical mechanics.

On the other hand, to find solutions to complex systems with a large number of particles, statistical mechanics uses a completely different approach, similar to randomized trials, that is not concerned about individual particles but rather studies the properties of a set of particles.

Through statistical sampling, the Monte Carlo method iteratively evaluates a deterministic model using sets of random numbers as inputs. As a result of using random inputs, the deterministic model of the complex system is transformed into a stochastic model.

The modern Monte Carlo technique – not the statistical sampling – was invented by Stan Ulam and later independently rediscovered by Enrico Fermi. The name Monte Carlo was used in World War II to refer to the sampling-based methods for neutron scattering computations.

Ulam automated the statistical sampling by developing algorithms for the newly invented electronic computer ENIAC [12] and transformed non-random problems
into random forms to facilitate the computation of their solution through statistical sampling. He describes how the idea took shape in his mind [13] as:

"The first thoughts and attempts I made to practice [the Monte Carlo Method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than "abstract thinking" might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations. Later [in 1946, I] described the idea to John von Neumann, and we began to plan actual calculations."

2.1.1 The Mathematical Foundation

The Monte Carlo method is designed to analyze and propagate uncertainty by determining how random variation and lack of knowledge affects the sensitivity of the solutions of the system that is being modeled. The inputs are randomly generated based on their probability distributions to represent samples collected from an actual population. The outputs of the simulation are also represented by probability distributions.

The expected value of a random variable $X$, defined over the interval $[a, b]$, with
a probability distribution of $f(x)$ is described by:

$$E(X) = \int_a^b xf(x)dx$$  \hspace{1cm} (2.1)

The expected value, $E(X)$, can become hard to compute via the above integral. The Monte Carlo method states that $E(X)$ can be estimated by collecting $n$ sample points $x_1, x_2, x_3, ..., x_n$ from $X$ based on $f(x)$ and then calculating the average values of $x_i$s:

$$E(X) \approx \hat{E}_n(X) = \frac{1}{n} \sum_{i=1}^{n} x_i$$  \hspace{1cm} (2.2)

The Monte Carlo method is validated by two simple laws that are foundations to statistical analysis:

1. The Law of Large Numbers (LLN) states that the mean (average) of a sequence of random variables with a common distribution converges to the (true) mean of the population, as the size of the sequence approaches infinity. The LLN is important because it guarantees the convergence of the Monte Carlo method via stable results for random events as the number of observations (sample size) increases.

2. The Central Limit Theorem (CLT) states that the sum of a large number of independent observations, in general, exhibit a normal distribution (the bell curve) as the number of samples increases. In the context of the Monte-Carlo method, the CLT is the basis to evaluate the quality of solutions of the simulated system.

For Equation 2.2, the CLT states that $E - \hat{E}_n = \sigma Z$, where $Z \sim N(0, 1)$
and \( \sigma \), the standard deviation of \( \hat{E}_n \), is estimated by the following equation:

\[
\hat{\sigma}_n = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{E}_n)^2}
\]  

(2.3)

The result of the Monte Carlo are reported in the form \( E = \hat{E}_n \pm \hat{\sigma}_n \), which gives one standard deviation error bar of \( [\hat{E}_n - \hat{\sigma}_n, \hat{E}_n + \hat{\sigma}_n] \). Based on the CLT, one standard deviation error bar indicates that the probability \( E \) being within the above interval over large number of trials is 66%. Similarly, based on the CLT, chances of \( E \) being in confidence interval \( [\hat{E}_n - 2\hat{\sigma}_n, \hat{E}_n + 2\hat{\sigma}_n] \) is 95%.

### 2.2 The Bayesian Method

In probability theory, *Bayesian theorem* states that the probability of a particular inference being true – the *posterior probability* – given some observed evidence, is a combination of the *prior probability* of the inference and the compatibility of the observed evidence with the inference.

The prior distribution is the probability distribution before observing data. After observing data, the new probability distribution, the posterior distribution, is computed using Bayes’ inference theorem [14]. The Bayes’ theorem for discrete probability distributions is based on the conditional probability and expressed as:

\[
P(A|B) = \frac{P(B|A)P(A)}{P(B)}
\]  

(2.4)

In Equation 2.4, \( P(A|B) \), refers to the probability of event \( A \), given the occurrence of some other event \( B \). In other words, in a random experiment when \( B \) has occurred, the probability of the occurrence of \( A \) is adjusted from the unconditional
probability into the conditional probability under $B$. In the context of the Bayes’ theorem:

- $P(A)$ is the prior probability and does not contain any information about occurrence of $B$.

- $P(A|B)$ is the posterior probability – the adjusted probability of $A$ upon observance of B.

- $P(B|A)$ is called the likelihood of event $B$ happening in the presence of prior probability distribution of $P(A)$.

- $P(B)$ is the prior probability of $B$ and is used to normalize the probabilities so that they sum up to one.

Bayes’ theorem in this form gives a mathematical representation of how the posterior probability distribution preserves knowledge contained in the prior distribution and the adjustment made according to the new observed data.

In this work, I use the Bayesian method to estimate the probability distribution of the systematic error function, which will be discussed in Section 6.4. I assume that the prior probability distribution is uniform over the interval $[0, 100)$ and decays exponentially afterwards.

The uniform distribution is used when background information about the prior distribution is limited, which indicates that before any observations, all outcomes are equally likely. The exponentially decaying tail represent the fact that the for a reasonably good performing model, chances of reporting a prediction with a large error magnitude tends to be very small.
2.3 The Graphics Processing Unit

A little over a decade ago, NVIDIA built the first graphics processor to accelerate the real-time graphics. Gradually, high arithmetic and memory bandwidth made GPUs ideal high performance computing platforms for data-parallel applications.

The first GPU programming interfaces were limited to shader languages such as DirectX [15], OpenGL [16] and Cg [17]. As a result, programmers had to express the computation of their applications in terms of graphics pipeline operations. Later, Brook [18] and Sh [19] added stream extensions to the C language to abstract away the graphics hardware details. Accelerator [20] moved one step further from stream programming by providing data-parallel arrays with aggregate element-wise operations.

2.3.1 The G80 Architecture

While the above research libraries made noticeable effort to overcome the drawbacks of the shader languages, the inflection point in GPU computing happened when NVIDIA introduced the G80 family of unified graphics and compute processors. The G80 GPUs were accompanied by a new combined software and hardware architecture called the Compute Unified Device Architecture (CUDA), which significantly improved programability of the new graphics processors and established the new GPU computing model.

As the G80 architecture combined the vertex and pixel pipelines, the new unified processor was capable of executing both graphics (vertex, pixel, etc.) and general purpose computing programs. This design established a ground base for the new generation of programmable graphics processors.

Figure 2.1 summarizes major components of a graphics processor – important during the course of my discussion in this thesis – based on a simplified diagram.
Figure 2.1: Block diagram of NVIDIA’s G80 graphics processor

of the NVIDIA’s GTX 8800 GPU.

At a high level, the GPU programming model is based on the Single Program Multiple Data (SPMD) model; a kernel function defines the program that is executed by each of the thousands or millions of fine-grain threads that compose a GPU application. Host code initializes the kernel parameters and invokes it through CUDA APIs.

From an architectural perspective, a graphics processor is formed by group of multiprocessors each consisting of a number of simple inorder cores and a
massively parallel memory architecture. The programmer decomposes the problem domain into a number of blocks. For each block, a small program is executed which uses a number of independent threads to perform its computation. The blocks are scheduled on multiprocessors which operate on them in parallel. Groups of Independent threads within in block operate based on the single-instruction multiple-thread (SIMT) execution model; threads within a group run in lock-step executing the same instruction.

The first programmable GPUs provided limited memory hierarchy and synchronization support: A small per core software-managed cache (shared memory) was provided to share data within a thread-block. A barrier synchronization intrinsic was supported for intra-thread-block communication.

2.3.2 The Fermi Architecture

Major successors to the G80 design are based on the Fermi architecture, which was first released in 2010. The Fermi architecture has taken a significant leap forward in GPU architecture design by providing a two-level cache hierarchy to better support the applications that are not able to use the GPU’s shared memory efficiently. Among other major key improvements are the improved double precision performance, faster atomic operations to reduce the cost of inter-thread-block communication, and the Error Correction Code (ECC) support.

I use the Fermi architecture (NVIDIA Tesla C2050) as the target platform for this work. In what follows, I briefly highlight hardware and software features that either significantly influenced the most important concepts in my work, or were used in the course of developing the infrastructure that this work is based upon. A more detailed description of these can be found in [3].
<table>
<thead>
<tr>
<th>Counter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1_global_load_hit</td>
<td>Number of global load hits in L1 cache</td>
</tr>
<tr>
<td>l1_global_load_miss</td>
<td>Number of global load misses in L1 cache</td>
</tr>
<tr>
<td>l2_subp0_write_sector_misses</td>
<td>Accumulated write sector misses from L2 cache for slice 0 for all the L2 cache units</td>
</tr>
<tr>
<td>l2_subp1_write_sector_misses</td>
<td>Accumulated write sector misses from L2 cache for slice 1 for all the L2 cache units</td>
</tr>
<tr>
<td>l2_subp0_read_sector_misses</td>
<td>Accumulated read sector misses from L2 cache for slice 0 for all the L2 cache units</td>
</tr>
<tr>
<td>l2_subp1_read_sector_misses</td>
<td>Accumulated read sector misses from L2 cache for slice 1 for all the L2 cache units</td>
</tr>
<tr>
<td>l2_subp0_write_sector_queries</td>
<td>Accumulated write sector queries from L1 to L2 cache for slice 0 of all the L2 cache units</td>
</tr>
<tr>
<td>l2_subp1_write_sector_queries</td>
<td>Accumulated write sector queries from L1 to L2 cache for slice 1 of all the L2 cache units</td>
</tr>
<tr>
<td>l2_subp0_read_sector_queries</td>
<td>Accumulated read sector queries from L1 to L2 cache for slice 0 of all the L2 cache units</td>
</tr>
<tr>
<td>l2_subp1_read_sector_queries</td>
<td>Accumulated read sector queries from L1 to L2 cache for slice 1 of all the L2 cache units</td>
</tr>
</tbody>
</table>

Table 2.1: NVIDIA performance counters used to cross validate the predicted L1 and L2 caches hit ratios

**Streaming Processors**

A streaming multiprocessor in Fermi has 32 in-order streaming cores. Each core can execute a single floating point or integer instruction per clock.

**Thread Scheduling**

Thread blocks are allocated to the streaming multiprocessors by a global thread scheduling engine. Within a streaming multiprocessor, a local thread scheduler picks up warps that are ready to run and executes them on the streaming processors.

**Memory Hierarchy**

The Fermi architecture memory hierarchy design takes a large step towards a more generic general purpose setting with two levels of data caches; a
private L1 cache shared within streaming cores of a multiprocessor and a unified L2 shared among multiprocessors.

Atomic Operations

Atomic operations are operations which are performed in a mutually exclusive manner (without any interruption from other threads) to prevent race conditions among concurrently running threads of a GPU application. They work for both shared memory (intra-thread-block mutual exclusion) and global memory (inter-thread-block mutual exclusion). The unified second level cache in Fermi, greatly enhanced the atomic operation performance for global memory.

In Chapter 5, I describe how atomic operations are used to collect trace information when an instrumented GPU kernel is being executed.

Parallel Thread eXecution Instruction Set Architecture

Parallel Thread Execution (PTX) is NVIDIA’s three address intermediate representation in Static Single Assignment (SSA) format for CUDA and OpenCL programming interfaces. The PTX assembly is translated by the shader compiler into the cubin form native to the target GPU architecture.

The CUDA programming interface has recently added support to allow PTX assembly being inlined in the kernel code. This feature is exploited in the design of the dynamic trace collection module, described in Chapter 5, to implement a robust and safe communication mechanism between GPU and CPU.

Performance Counters

Performance counters are enabled by setting a few environment variables to initialize the set of hardware counters to be sampled.
In this work, performance statistics provided by the hardware counters are used to cross validate the predictions of the memory hierarchy model, e.g., the overall hit ratios for the L1 and L2 caches. Table 2.1 lists the hardware counters that were used for the purpose of this validation.

2.4 Related Work

2.4.1 Functional Simulation

Functional simulators such as Barra [8], a GPU simulator that implement the PTX instruction set, simulate instructions one by one and are used to check whether the instruction set implementation is correct. They cannot provide accurate performance statistics as they do not account for the correct timing of events.

Nevertheless, a functional simulator can be used to generate instruction traces for a full system trace-driven simulation. Instruction traces can also be used in cache predictor simulations, relatively accurate in predicting hit and miss ratios for a single stream of instructions.

In this work, I use an approach similar to a trace-driven cache predictor, with the exception that in a highly multithreaded execution environment, there are more than a single stream of instructions – thousands of inflight threads issue memory operations all together. As a result, the proposed memory hierarchy simulation approach needs to combine the individual instruction streams (memory accesses) into a single one. It then, predicts the efficiency of the GPU memory hierarchy based on the synthesized relative ordering generated.
2.4.2 Full System Simulation

While full cycle-accurate simulators such as GPUSim [5] perform a detailed microarchitecture simulation of a program to study the performance of proposed architecture designs, the high overhead of full system simulation has lead to alternative simulation techniques at least for the CPU simulators. These simulation approaches that target reducing the simulation time can be grouped into the following categories:

- Reducing input set: Among the first studied approaches to reduce the simulation time is to modify the input set and have the simulator execute fewer number of instructions [21]. This way the benchmark would have gone through all different stages of computation and the reported performance metrics have potentially similar characteristics compared to the full input set. However, more comprehensive studies [22] showed that the reported cache miss rates for the reduced input set can be very different from the full input set simulation results. Therefore, the predicted Instruction per Cycle (IPC) statistics, computed under a reduced input set, will not be necessarily accurate.

- Truncated execution: The assumption behind this simulation technique is that an arbitrary interval of program execution is a reasonably good representative of the application behavior. Therefore, only a fixed number of instructions are simulated in detailed. This snapshot can start from the beginning of the application, or the initial phases of application computation can be skipped by functionally simulating those and later switching to a detailed simulation for the snapshot of interest. The functional simulation helps in avoiding the cold error due to uninitialized states of the memory and processor.
Sampling: Sampling approaches fully simulate only a subset (sampling units) of a benchmark’s execution. Instructions between the sampling units are fast forwarded through functional simulation, used primarily to maintain the up to date microarchitectural state (caches and branch predictors). Sampling can be performed under one the following schemes:

Representative sampling: This approach extracts a subset of dynamic instruction stream such that it matches the overall behavior of the benchmark when running the full input set. Representative subsets are chosen through profiling and offline analysis of the program basic blocks [23]. After a particular sequence of instruction streams (basic blocks) have been simulated once, the collected statistics is weighted appropriately to reflect the overall presence of each basic block when executing the full input set.

Periodic sampling: Periodic sampling [24] simulates selected subsets (sampling units) of the dynamic instruction stream sampled at fixed intervals. The sampling frequency is determined based on the desired level of accuracy.

Random sampling: In random sampling, the simulation results from a large number of randomly sampled intervals are combined together to produce the overall simulation statistics [25].

2.4.3 Analytical Modeling

Analytical modeling, on the other extreme, provides valuable insight by approximating the microarchitecture behavior. It specifically works well when the underlying design is relatively simple, e.g., in-order cores as opposed to superscalar processors. In this work, I combine an analytical model with a novel trace-driven
memory hierarchy simulation approach to evaluate the performance of applications running on a highly multithreaded graphics processor.

Previous studies on performance estimation [26, 27, 28] and tuning [29] for general purpose computing on graphics processing units (GPGPUs) were constrained by the programming environment and the necessity of mapping algorithms to existing GPU features.

The first body of research on GPU computing (on programable graphics processors) started with the work of Ryoo et al. [31] who used Pareto-optimal curves to prune the optimization space of general-purpose applications on GPUs. They introduce efficiency (a flat instruction count) and utilization (a measure of how many pure compute cycles are available from other executing warps) as single number metrics. They did not model memory latency and assumed that none of the studied GPU kernels were memory bound.

Later Baghsorkhi et al. [32] and Hong et al. [33] proposed analytical models to evaluate the performance of the GPU applications. In a previous work [32], I explained how static analysis can be efficiently used to account for control flow divergence, global memory coalescing and shared memory bank conflicts when the control flow and array index expressions are not data-dependent. This topic is discussed in more details in Appendix A. I also showed that GPU kernels with data-dependent control flow conditions and memory accesses can be efficiently instrumented to collected the data required for the analytical model.

Neither of the approaches discussed in [33, 31] handle the diverging control flow or the data-dependent GPU applications properly. Hong et al. [33], conservatively account for both control flow paths even if all threads (of a warp) go through the same path. Properly handling control flow is important for a performance model, specially when memory loads are protected by conditionals; a conservative approach can result in more than twice the number memory opera-
tions that a kernel executes, which will result in large deviation between predicted and real performance statistics.

None of the previous work in the area of performance tuning and modeling of graphics processors [31, 32, 33, 34, 35], to this point, have accounted for cache behavior with inter-core or inter-thread data sharing and neither did model a GPU with a multi-level memory hierarchy.

This work is also different from GPU simulators such as GPUSim [5] as they emulate each instruction in detail and account for all GPU micro-architecture states (in the case of GPUSim). This requires much more simulation overhead compared to the approach proposed in this thesis; I simulate a small portion of the memory traces. Furthermore, my approach provides better targeted (source code level) performance feedbacks to application developers and systems engineers.
In this chapter, I will define a machine model for a highly parallel graphics processor targeted toward general purpose and compute oriented workloads. This machine model must be simple enough to facilitate study and efficient evaluation of the graphics processors. On the other hand, it need to enclose enough details so that it can reflect a realistic behavior of the hardware. In what follows, I highlight a few fundamental differences reflected in this model that separates graphics processors from their peer multi-core processors.

3.1 Throughput-oriented Computing

GPUs emphasize high throughput and single instruction multiple thread (SIMT) performance rather than single thread (instruction) execution latency. As a result, the number of inflight threads and memory accesses is much higher compared to conventional multi-core CPUs.

The SIMT is enabled by having streaming processors running neighboring threads in lock-step, executing the same instruction (on different data). This microarchitectural grouping of threads which can affect both control flow and memory access efficiency introduces the concept of warp – a group of threads that compose a hardware vector unit.
3.2 Hardware Thread Scheduling

Preemptive thread scheduling by the operating system cannot efficiently support a throughput oriented environment. Therefore, GPUs implement thread scheduling through a dedicated hardware.

A global thread block scheduler assigns thread blocks to streaming multiprocessors – arrays of streaming processors that implement the SIMT execution model. In this work, I will consider a fine-grain round robin scheduling paradigm with respect to the relative order of thread blocks assigned to each streaming multiprocessor. For example, with 4 streaming multiprocessors available, each running 6 thread blocks, the model assumes the following mapping of logical thread blocks to multiprocessors:

<table>
<thead>
<tr>
<th>Multiprocessor 0</th>
<th>Multiprocessor 1</th>
<th>Multiprocessor 2</th>
<th>Multiprocessor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>56, 60, 64</td>
<td>57, 61, 65</td>
<td>58, 62, 66</td>
<td>59, 63, 67</td>
</tr>
<tr>
<td>68, 72, 76</td>
<td>69, 73, 77</td>
<td>70, 74, 78</td>
<td>71, 75, 79</td>
</tr>
</tbody>
</table>

This mapping is the closest model to the dynamic scheduling scheme that the hardware implements.

Within a multiprocessor, a warp scheduler alternate between different warps following a weighted round robin scheme.

3.3 Bulk Instruction Scheduling

To make the model tractable and avoid fine-grain instruction scheduling of a kernel instructions, which would result in a full system simulation model, I propose a bulk scheduling scheme. The computation within a kernel is divided into a sequence of memory loads followed by a sequence of compute or store instructions.

When static memory loads are repeated within a dynamic instruction stream, in the presence of enclosing loop constructs, they end up in separate sequences.
that are sequentially executed. Therefore, a back-edge in the control flow will terminate a sequence.

Another delimitating factor is the existence of data-dependence between static memory loads. For example, if the index expression of a memory load is computed based on the value that is being read by a previous memory load, the second load will start new sequences for both memory loads and compute instructions.

Explicit synchronization instructions will also mark the end of the current sequences.

The above simplifications preserve enough information about the instruction stream so that the model can reflect a reasonable representation of the kernel computation. Meanwhile, it is general enough that can approximate different instruction scheduling schemes adapted by different GPU vendors. For example, AMD’s GPUs are designed based on a Very Long Instruction Word (VLIW) Instruction Set Architecture (ISA); multiples of instructions that can execute in parallel are packed into statically scheduled VLIW bundles. These bundles then compose relatively short clauses; clauses that are initiated by control flow instructions, may only contain a single type of instructions, e.g., memory loads or ALU instructions. In contrast, NVIDIA’s GPUs require more sophisticated scheduling logic for fine-grain scoreboard and resolving dependencies among dynamic stream of instructions. Through the proposed bulk instruction scheduling scheme these level of details can be abstracted away.

The bulk instruction scheduling scheme proposed above accounts for the major dependencies among instructions and differentiate between types of instructions that consume different hardware resources (memory loads and compute instructions). At the same time, it consciously ignores the low level details of dependencies and scheduling order to make the analysis tractable.
3.4 Throughput-oriented Memory System

GPUs conserve the memory bandwidth by grouping together aligned loads and stores. Each coalesced memory access uses a single memory request, while moving multiple elements of data.

Traditionally, highly multithreaded architectures such as HEP [36], M-Machine [37], Tera MTA [38] and more recently multithreaded GPU processors employ hardware multi-threading for fast context switches to tolerate memory latency. Following this approach to tolerate memory latency, the early multithreaded architectures did not include data caches. More recent GPUs such as NVIDIA’s Fermi architecture [39] added caches as means to conserve memory bandwidth. Private L1 caches exploit inter-thread input data sharing to reduce the memory bandwidth consumption by each core. A shared unified L2 further reduces off-chip bandwidth requirements by exploiting data re-use between cores.

The throughput oriented environment also influences the design and functionality of the memory hierarchy. For example, data caches – if exist at all – are no longer optimized for exploiting long term locality as their effective size shrinks with high number of concurrent memory request. These caches are designed to exploit short-term spatial and temporal data locality across the concurrently executing threads rather than long-term temporal data reuse.

To lock data that is reused frequently but with a reuse distance larger than the effective cache capacity, the on-chip software-managed cache or shared memory can be used.¹

In NVIDIA GPUs, shared memory and the L1 data cache use the same underlying physical structure and cannot be accessed simultaneously. Therefore, shared memory can be conceptually considered as an extension to the L1 cache, with the

¹Reuse distance is defined as the number of distinct memory accesses between two occurrences of a memory reference.
difference that the data that is mapped to the shared memory address space will always hit in the L1. This assumption simplifies modeling of the memory level parallelism (MLP) that is discussed in Section 8.1.2.

![Figure 3.1: A Highly Multithreaded GPU Model](image-url)

Figure 3.1: A Highly Multithreaded GPU Model
```c
#include <cuda_runtime_api.h>

int queryDevice()
{
    // total number of devices
    int devCount;
    cudaGetDeviceCount(&devCount);

    for (int i = 0; i < devCount; ++i)
    {
        printf("Device %d:\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);

        printf("Name: %s\n", devProp.name);
        printf("Total global memory: %u\n", devProp.totalGlobalMem);
        printf("Max shared memory per block: %u\n", devProp.sharedMemPerBlock);
        printf("Max registers per block: %d\n", devProp.regsPerBlock);
        printf("Warp size: %d\n", devProp.warpSize);
        printf("Max threads per block: %d\n", devProp.maxThreadsPerBlock);
        for (int i = 0; i < 3; ++i)
            printf("Max block dimension: %d\n", devProp.maxThreadsDim[i]);
        for (int i = 0; i < 3; ++i)
            printf("Max grid dimension: %d\n", devProp.maxGridSize[i]);
        printf("Clock rate: %d\n", devProp.clockRate);
        printf("Num of multiprocessors: %d\n", devProp.multiProcessorCount);
    }

    return 0;
}
```

Listing 3.1: Query device function

### 3.5 The Complete Model Overview and Use-case Realization

Figure 3.1 shows a diagram of a GPU architecture with all the components that I discussed in previous sections. For the rest of this thesis, I will use this model of highly multithreaded graphics processors and customize it for NVIDIA’s Fermi architecture to demonstrate the design and effectiveness of proposed performance evaluation approach.

A Fermi-based GPU is composed of a number of streaming multiprocessors each consisting of a group of simple in-order processor pipelines. Each multiprocessor can have multiple in-flight warps from the same or different thread-blocks whose executions overlap. A warp scheduler switches among warps in
a semi-round-robin fashion. When a warp is stalled on a memory access, the scheduler picks another warp to execute. When the data becomes ready the warp is returned to the list of warps eligible for execution. Warps running on a node share an L1 data cache. A unified L2 cache sits between individual multiprocessors and the main memory. The L1 and L2 are set associative caches.

Configuration parameters for the streaming multiprocessors can be determined through a call to NVIDIA’s API function `cudaGetDeviceProperties` as shown in Listing 3.1. NVIDIA’s Tesla C2050, which is the Fermi based GPU that I use for experiments during the course of this work, contains 32 streaming processor per each of the 14 multiprocessors and works based on a warp size of 32.

To determine the specific memory hierarchy configurations and their management policies, I use a micro-benchmarking approach that will be discussed in the following section.

### 3.6 Microbenchmarks

I extract the configuration parameters of the memory hierarchy through micro-benchmarking. The information is later used by the model to adapt itself to the
profile of the specific target platform for executing the application.

The micro-benchmark suite is built around a GPU kernel that traverses elements of an array of linked-list in a circular (wrap-around) fashion. Consecutive elements of a linked-list are located within a certain stride from each other, while elements of different linked-lists are contiguous in the memory as illustrated by Figure 3.2. I invoke the kernel with a single GPU thread unless stated otherwise.

To determine different configuration parameters and arrangements of the memory hierarchy, I customize the micro-benchmark by varying the size of linked-lists, the number of linked-lists (stride), and the number of memory accesses inside the micro-benchmark kernel.

### 3.6.1 Cache Block Size

The first parameter that I measure is the cache block size. I run the micro-benchmark and vary the strides for a fixed linked-list size. The first access to a block in the cache is always a miss. Successive accesses to the same block will hit in the cache. As the stride increases, fewer accesses hit the same cache block and eventually when the stride is equal to the cache block every access will be a miss. This experiment initially implies a cache block size of 128 bytes (the solid line in Figure 3.3). This leads one to the conclusion that either there is no higher level caches or the cache block size of the higher level caches are at most 128 bytes. Later, the micro-benchmark results in Section 3.6.2 capture two levels of cache memory, which leads to the conclusion that L2 cache block size is at most 128 bytes. The Fermi L1 cache can be deactivated though a compiler switch. With that option available, I further investigate the cache block size of the second level cache (the dashed-line in Figure 3.3). The new experiment yields an L2 cache block size of 32 bytes. Based on this outcome, I assume that on an L1 miss four
Figure 3.3: Average memory access time for different stride values with a fixed list size and number of memory access consecutive L2 cache blocks are accessed.

3.6.2 Cache Capacity

Next, I determine the size of the first and second level caches. I run the micro-benchmark using a fixed stride of 128 bytes (the L1 cache block size) for a single linked-list. At each invocation, the kernel sequentially accesses elements of the list and wraps around a certain number of times. When the list size is smaller than the cache capacity, subsequent accesses to the same element will hit in the cache, and the average memory access time is low. When the list size exceeds the capacity of the cache, accesses to the same element will always miss in the cache and the average access time increases.

The solid line in Figure 3.4, which shows the results from running this bench-
mark on the Tesla C2050, suggests a 16KB first-level cache and a 768KB second-level cache. I repeat the same experiment with a fixed access stride of 32 bytes (the L2 block size) and the L1 cache disabled. Results shown in Figure 3.4 (the dashed line) re-confirm the 768KB capacity for the L2.

3.6.3 Cache Set Associativity

To measure the first and the second level cache associativities, I access elements of a linked-list with fixed strides that are equal to the cache size, which guarantees that all the accesses will end up in the same cache set. The linked list is traversed a large number of times in an experiment. I repeat the experiment multiple times and each time increase the linked-list size by one, which increases the number of distinct cache blocks accessed during linked list traversal by one. Eventually the cache set overflows. At this point, each access causes a (conflict) miss which will be reflected in a significantly longer average latency of the corresponding load. The results indicate that both L1 and L2 caches are 64-way set-associative.

The 64-way set associative L1 may at first glance look unreasonable, but when I measure the L1 access latency in the following section, it indicates a latency of about 52 cycles. This long latency access time justifies a 64-way set associative cache.

3.6.4 Latencies

To measure the access latencies for each level of the memory hierarchy, I run the micro-benchmark for a linked-list that completely fits into the memory in that level but exceeds the memory capacity of the lower levels. Using a stride equal to the cache block size, I access the linked-elements many times to factor out the effect of cold misses. I measure an average L1 access time of 90 nanoseconds,
3.6.5 Inclusive, Exclusive or Victim

I also need to determine whether the second-level cache is inclusive or exclusive. Therefore, I execute the micro-benchmark using a list size equal to the L1 cache capacity (i.e. 16KB) for as many thread blocks as the number of available streaming multiprocessors plus one, i.e. 15 thread blocks in the case of Tesla C2050. To ensure that different thread blocks are not scheduled at the same time on the same core (streaming multiprocessor), I enforce each thread block to have 1024 threads or 32 warps, which is the maximum number of in flight warps that can be simulta-
neously scheduled on each core. However, only one thread is performing memory accesses in each thread block. Each of the first 14 thread blocks access elements of 14 completely distinct linked-lists. The last thread block, which is roughly executed after the first 14 finished their execution, randomly chooses one of the previously accessed linked-lists to start its memory accesses. Memory accesses from the first 14 thread blocks will miss in both L1 and L2 caches. However, memory accesses of the last thread block will hit in L2, if the cache is inclusive and in L1 if the last thread block is scheduled on the same core as a thread block that has previously accessed the list. I run this benchmark 1024 times, and discard the results that indicate the accesses from the last thread block hit in the L1 cache.

Next, I run a second set of experiments: this time all 15 thread blocks access distinct linked-lists. I compare the average memory latency for these two set of experiments. Results show a lower memory latency for the case that the last thread block reads from a previously accessed linked-list. This leads one to the conclusion that the second-level cache is inclusive.

3.6.6 Write Policy

Since the L1 caches are relatively small considering the number of inflight threads for each core, it makes sense to forward stores directly to the shared L2 cache. To verify this scenario, I modify the micro-benchmark to first initialize the values of a linked list by writing the pre-calculated indices for the next items in the list. After this initialization phase, I use the internal clock register to measure the execution time of loading successive elements from the linked-list. The time measured indicates that loads do not hit in the L1 cache. In other words, write requests bypass the first-level cache. Through further micro-benchmarking and by closely investigating the numbers reported by the hardware counters, I verified that stores hit
in the L2 only if the corresponding cache block has been activated via a write request, and that the L2 follows a write back, write allocate policy for stores. So, for the memory hierarchy model I assume that when a store request reaches the L2, if the cache block is valid it writes to the cache while setting the dirty bit for that block. If the block is not valid, it updates the memory and brings the cache block to the cache and set its valid bit.
In this chapter I describe how a stochastic model of the GPU memory hierarchy is built on top of the deterministic memory model discussed earlier in Chapter 3; the order of memory request being issued is partially determined by random variables. Initial memory requests are generated by instrumenting the GPU kernel at source level to record the addresses accessed during the kernel execution.

The instrumentation framework, which is discussed in more detail in Chapter 5, is designed as a source-to-source transformation module. Static probes are inserted within the GPU kernel source code to record the memory addresses read from and written to by active threads for a subset of thread blocks as the GPU kernel is executed. This subset of thread blocks represents a snapshot of the kernel execution whose dimensions are determined through a heuristic based on the number of concurrently executing thread blocks, the (average) number of dynamic memory operations per warp, and the size of the last level cache.

4.1 Spatial and Temporal Locality

To understand the performance of the GPU kernel with respect to the memory hierarchy, it is necessary to collect enough traces to warm up the L1 and L2 caches. The information must also be detailed enough to capture:
• the intra-thread locality and cache interactions
• the inter-thread and inter-thread-block locality and cache interactions
• the behavior of the kernel during different phases of the computation

The memory references in a trace are distributed both temporally and spatially. To capture the spatial locality, which in the context of a highly multithreaded graphics processor is translated to inter-thread, inter-thread-block and inter-core locality and memory hierarchy interactions, the execution snapshot of traces is extended horizontally. The proposed technique collects traces for thread blocks that are potentially scheduled close together on different cores (streaming multi-processors), i.e. thread blocks with consecutive logical IDs.

To account for the temporal locality the execution snapshot is extended such that enough traces are collected within a single thread. The threshold is determined by the capacity of the last level cache and the total number of inflight threads for all the cores. If required, the execution snapshot is extended across the boundaries of multiple thread-blocks that are scheduled back-to-back on a single core.

With this approach, a subset of the memory traces is collected, but the subset is detailed enough to reflect locality within different granularities discussed above. If required, traces can be collected for multiple snapshots of the kernel execution. However, the initial experiments confirm that applying the model to the traces collected from a single snapshot produces precise and accurate enough estimations. This is quite expected as the computation in a typical GPU kernel is structured around a Single Program Multiple Data (SPMD) programming model. Nevertheless, the demand-driven sampling of the traces provides the flexibility to trade-off between the overhead of trace collection and the accuracy of the results generated based on the collected traces.
The above approach, is to some extent similar to the time sampling technique introduced by Laha et al. [40]. They used contiguous segments of memory accesses over certain time intervals for trace driven simulation of single-thread workloads. For large caches and in a single-thread execution model, the cold-start error caused due to unknown status of the cache at the start of each trace simulation is a major source of inaccuracy. In a highly multithreaded execution model, the effect of the cold-start error is fairly insignificant; the execution environment setup for throughput oriented caches limits their ability in exploiting long-term temporal locality. I expect that the same limitation will hold in future GPUs as parallelism scales with at least the same rate as the size of the caches increases.

4.2 The Monte Carlo Approach

Collected traces exhibit precise intra-warp ordering of the memory references. But they do not maintain any information on the relative order of memory references issued from different warps or thread blocks. Note that the proposed approach does not rely on the execution order of memory loads and stores when collecting traces. The rationale for not relying on the ordering during the execution of the instrumented kernel is that adding the static probes to the kernel source code:

- changes the kernel resource usage (number of registers) which may consequently alter the streaming multiprocessors occupancy, i.e., the number of concurrently active thread blocks.
- increases the number of inflight memory operations which will distort the state of caches and the level of congestion in the memory hierarchy.
- changes the instruction mix of the GPU kernel and introduces spurious synchronization or stall points.
As a result, instrumenting the kernel will induce considerable timing distortions in the execution order of the memory references. To account for this effect, the order of memory requests arriving at each level of the memory hierarchy is reconstructed via a Monte Carlo method which is an efficient sampling approach for systems with dynamics that are highly coupled together. Traces are then driven into a memory hierarchy simulator following the randomly sampled ordering for each run.

Thus, given a pool of memory requests waiting to be serviced at each level of the memory hierarchy, the following steps are performed:

1. Generate a valid random ordering from the pool of available memory requests.
2. Use the deterministic memory hierarchy simulator discussed in Section 3 to obtain an output for the ordering derived in step 1.
3. Repeat steps 1 and 2, N times.
4. Determine the probability distribution of results using histograms and summarize the confidence of the predictions.

The output of the model is a probabilistic performance behavior of the memory system such as hit ratios for the first and second level caches.

4.2.1 Schedule Deviation

As independent thread blocks are scheduled to run on streaming multiprocessors, their executions start to fall out of synch with each other due to non-uniform memory access latencies, variations in the number of memory operations and computation load as individual threads or warps may follow different control flow paths, etc. To account for this deviation, when scheduling loads and stores from inflight
Figure 4.1: Random points mimic schedule deviation for simultaneously scheduled thread blocks

warps, random start points are chosen for thread blocks scheduled simultaneously across different streaming multiprocessors. The horizontal dashed lines in Figure 4.1 highlight the execution snapshots devised based on random start points for two multiprocessors. In this example, snapshots are expanded across two back-to-back scheduled thread blocks – each composed of four warps – and each multiprocessor executes two thread blocks simultaneously. Tiny bars in front of dynamic memory operations $L_0, \ldots, L_6$ represent individual memory transactions being issued for the corresponding memory vector instruction. The entry is empty if none of the threads within the warp have executed the matching load or store.
4.2.2 Scheduling Traces

The scheduling scheme that is used preserves the vector nature of the memory references issued by threads within a warp. Memory references are coalesced based on the size of the data being accessed and the cache block size. Coalesced accesses (memory transactions) for each warp are bound together to ensure that they are all scheduled at the same time. Each warp of a thread block has a queue of ready-to-issue memory transactions. Attached to each memory transaction is

Figure 4.2: The stochastic memory hierarchy model
a unique ID that corresponds to the source code location of the memory load or store that has triggered the transaction.

Figure 4.2 illustrates the layout of the memory hierarchy simulator. The intra-core trace schedulers pick traces from the warp queues following a weighted round robin fashion. Traces are sent to the L1 caches in the order that they have been picked. Loads update the status and counters of the L1 caches. All stores and the loads that miss in the L1 are forwarded to the L2 cache.

When scheduling traces from a warp queue, the scheduler continues picking transactions from the same warp if the corresponding cache lines are triggered by static memory operations within the same scheduling sequence, provided that no data-dependence has been recorded between back to back scheduled accesses (read after writes). This scheduling policy follows the concept of bulk instruction scheduling discussed earlier in Section 3.3.

The inter-core trace scheduler picks traces from L1 queues based on a weighted random scheduling algorithm. Loads and Stores that miss in L2 along with store evictions are placed into the memory request queue as shown in Figure 4.2.

4.2.3 Main Memory

In the context of a highly multithreaded environment, memory latency observed from a single-thread execution is not necessarily a meaningful measure of performance. Congestions in the memory system may add to the memory latency. Congestion pattern is characterized based on how efficiently a shared resource is utilized by the memory requests that arrive close in time. The main memory, one of the most significant shared resources in a GPU, is organized into a number of interleaved channels. Depending on how memory accesses are spread across these channels the effective memory bandwidth and latency will change. To esti-
Figure 4.3: Probability distribution for main memory latency in LBM

Figure 4.4: Probability distribution for main memory latency in SpMV
mate the effective memory latency observed by memory operations issued close to each other, memory accesses that reach the main memory are grouped together following a uniform random distribution.

To form groups of simultaneously issued memory loads or stores, random number of memory requests are picked from the queue of misses arrived from the L2 cache. Each group can contain up to 32 memory requests which is the maximum number of distinct memory transactions a core can issue simultaneously, i.e. the SIMD memory vector size. Then up to 14 groups are picked, i.e. one group per multiprocessor core, to form a set of memory requests that reach the main memory relatively at the same time.

The addresses in this aggregated group of memory transactions are then normalized and translate into indices of an array whose elements are the size of a single main memory transaction (32 bytes). The indices are then loaded into a synthesized micro-benchmark and the main memory access latency for each batch of concurrently scheduled accesses is measured via the internal clock register. Figures 4.3 and 4.4 show the probability density function (PDF) of the observed latencies collected by the above approach for the lattice-Boltzmann method and sparse matrix vector multiplication benchmarks.

The general rule is that more frequent random accesses and higher memory bank conflict rates will degrade the main memory efficiency and increase the expected latency. If a dominant measured latency exists – which is not sensitive to the exact grouping or ordering of the memory accesses arriving closely at the main memory – that latency is a proper statistical representative for the memory accesses that miss in the L2 cache. For example, the PDFs shown in Figures 4.3 and 4.4 are far from a uniform distribution and both indicate a dominant expected access latency for the corresponding benchmarks.
CHAPTER 5
SOFTWARE FRAMEWORK

This chapter discusses the infrastructure and framework that collects: dynamic memory traces for the stochastic memory hierarchy simulator discussed in Chapter 4 and an approximation of the number of dynamic instructions executed, which is used by the analytical model discussed in Chapter 8.

5.1 Instrumenting the Kernel

To collect and analyze the memory addresses accessed and instructions executed by a given kernel, additional instructions are inserted inside the kernel and the host codes. This is performed automatically by a source-to-source transformation module that is run before the actual compilation of the program. This module and the actual compiler are wrapped together into a new compiler driver that can replace the original compiler in a project build system. This minimal level of intervention simplifies the use of the framework significantly.

The source-to-source transformation module performs the following modifications:

- Inserts memory address probes into the kernel source code at locations where the kernel is performing a read or a write operation. A probe is a call to a function that will be in charge of storing the memory address accessed as well as a unique ID that identifies the source code location of the memory operation.
• Stores for each memory address probe, its exact source code location and the size of the element being accessed into a static array.

• Inserts control flow probes into the kernel source code before and after each control structure, and before and after the main body of the control structure. A control structure is a block of programming that evaluates a condition and chooses a control flow path to take based on the result of the evaluation, e.g., a "for loop", an "if then else" clause, etc.

• Generates a mapping to record the number of instructions in the abstract syntax tree (AST) between each potentially consecutive pair of control flow probes. This mapping is used during the post processing of traces to produce an approximate dynamic count of instructions.

• Adds allocation and deallocation routines for the sampling buffer before and after the launch of the kernel. A pointer to the buffer is added as an extra argument to the kernel invocation code. The organization of the sampling buffer is explained later in Section 5.2.
• Adds an additional parameter to the kernel that communicates to the probes the location within the buffer that they need to store the corresponding sampled information.

• Adds routines to compute the original kernel occupancy as described in listing 3.1.

• Adds routines that collect traced memory addresses and post-process them for the memory hierarchy simulation model.

To initiate trace collection for a subset of simultaneously executing thread-blocks, one need to know the thread-block occupancy for a streaming multiprocessor and the number of available streaming multiprocessors; the latter is obtained via a call to the NVIDIA’s programming APIs. The former is computed as follows.

The occupancy of a GPU kernel is the ratio of the number of active warps to the maximum number warps supported on a streaming multiprocessor [41]. The occupancy is determined by the amount of resources that the kernel consumes. Resources can be allocated either statically or at the kernel invocation time. Occupancy is also dependent on the device that the kernel will be executed on.

Before starting the source-to-source transformation, the driver calls NVIDIA’s \texttt{nvcc} compiler to collect the kernel-specific information required to compute the occupancy. This includes the number of registers and the size of the shared memory used. The instrumentation module inserts calls to NVIDIA’s programming interfaces to identify the GPU device that the kernel will be executed on – similar to routines shown in Listing 3.1. It then compute the occupancy of the kernel just before its invocation; meanwhile, it collects the kernel invocation parameters whose values are required for the occupancy computation, i.e., the thread block dimensions and the amount of shared memory dynamically allocated.

Redundant loads and stores that are likely to be eliminated by the back-end
compiler are not instrumented. To achieve this, a global value numbering analysis is implemented to identify the redundant memory accesses and a basic pointer alias analysis is used to determine if a memory operation will be removed through redundant load elimination.

The source to source transformation module is build on top of Clang [30], the C language family frontend for LLVM [42]. Clang has been modified so that it can correctly parse a subset of the CUDA [1] source code and build the abstract syntax tree of both the host and the device code. Each node of the tree precisely records its exact location within the source code, even when the node is the result of one or more macro instantiations. This property is essential for the transformation module to provide accurate source level feedback.

The output of the transformation cannot be generated directly from a modified abstract syntax tree: To build the abstract syntax tree, a header containing the declaration of CUDA types and runtime functions must be preincluded. The serialization of the abstract syntax tree would contain these additional declarations. But *nvcc* preincludes a header file as well. So, if *nvcc* is run to compile the source code generated from the abstract syntax tree, it will result in duplicated declaration errors. Instead, the output of the transformation is a patch file that is applied to the user source code. The modifications that need to be applied to the original source code are recorded as a list of source code insertion operations that are later used to generate the patch file. This mechanism leverages the Rewriter API that Clang provides.

The instrumentation of the source code must be done with an unmodified abstract syntax tree to ensure that the source locations are correct. However, the abstract syntax tree must be modified to increase the quality of the global value numbering analysis: Expression trees are canonicalized to increase the chance that two arithmetically equivalent expressions have the same tree representation.
This means that the source code must be instrumented before the analysis is performed. As a result, the source code is over-instrumented. After the analysis is performed, the results are forwarded to the patch file generation module to skip the unnecessary instrumented memory operations.

A significant challenge in instrumenting the memory operations is identifying the correct type of the address space being accessed by a pointer, i.e. global memory, shared memory, etc. The type of a pointer cannot be used to infer which address space it points to as it can change during the lifetime of the pointer. A dataflow-based algorithm is used that disambiguates the address space that a pointer refers to for each segment of the source code. When incapable of resolving the actual pointer type, it conservatively associates the pointer to the global memory, similar to the \texttt{nvcc} compiler.

5.2 Recording Traces

The memory address probes inserted within the kernel store the addresses accessed by each thread into a sampling buffer. The control flow probes record the number of instructions observed between two control flow checkpoints. The implementation of these probes and the design of the buffer are inherent to the execution model within the GPU: The threads within a warp are executed in lockstep and the warps within a thread block are executed concurrently. Therefore, the probes are designed to store per-warp rather than per-thread memory accesses or instruction counts. In addition, the buffer is designed so that its size does not limit the acquisition of an unbounded amount of instructions or memory addresses.

A buffer is composed of a number of block buffers. Each block buffer itself contains a set of warp buffers – one for each warp in the thread block – as illustrated in Figure 5.1. The number of block buffers is equal to the number of
Listing 5.1: Instrumented SpMV – device code
Listing 5.2: Sampling functions – GPU-CPU communication

simultaneously active thread blocks in the GPU.

I use a per-block-buffer locking mechanism, described in Listing 5.3, to prevent concurrent accesses to the same buffer from multiple thread blocks. Each block buffer has a counter that holds the number of active threads using the buffer. When zero, the buffer is free to be acquired. A thread block acquires exclusive rights to the buffer when it successfully exchanges, via an atomic operation, the value zero in the counter with the number of threads that it contains. Each thread will atomically decrement this counter just before terminating.

A warp buffer is a ring buffer with fixed-sized lines that are wide enough to hold for each thread within a warp, an address value and a static ID associated with the source code location of the corresponding memory operation. Instruction counts are stored following the same format with the exception that the static ID associated with an instruction count entry is -1.

When an active thread executes a probe, the next available line in the warp buffer is updated and a line counter is incremented. The buffer is originally initialized with invalid addresses (odd values). So when an inactive thread does not store any address it can be easily captured. Probes will spin on the warp buffer counter when the buffer is full.

During the kernel execution, the host program constantly snoops at individual
Listing 5.3: Sampling functions – inter-thread (thread-block) communication
warp-buffer counters. When a warp buffer is full, the host program reads the warp buffer content (traces) and reset the warp buffer counter. Now, threads that were spinning on the counter will resume to record the trace information. When enough traces is gathered, the host program set the counters to a predefined value that disables probes inside the kernel. The minimum number of indices to be sampled and the start window are set through a pair of environment variables whose values are read and applied to the initialization routines on the host side.

To allow both the host and the device code access the buffer concurrently, the buffer is allocated using *pinned-memory*. Pinned-memory is the host memory that is removed from the virtual memory, so it is not paged out by the operating system. Since the atomic operations operating on a pinned-memory are not atomic from the point of view of the host [1], the communication between host and device is implemented via un-cached load and store operations by inlining PTX assembly as shown in Listing 5.2. The `ld.cv` command considers the cached memory lines are stale and `st.wt` performs as a write-through store bypassing the GPU L2 cache.

The exact layout of the buffer is computed based on thread block dimensions, number of concurrently active thread blocks and the overall size of the buffer itself.
CHAPTER 6

EXPERIMENTAL EVALUATION:
MEMORY HIERARCHY

In this chapter, I present the result of application of the stochastic memory hierarchy model discussed in Chapter 4 to several GPU kernel applications. The results, which are presented in terms of the L1 hit ratios for loads and the L2 hit ratios for loads and stores are cross validated against the hit ratios reported by the hardware performance counters.

6.1 Experiments Setup

<table>
<thead>
<tr>
<th>Description</th>
<th>Data-dependent accesses</th>
<th>Data-dependent control flow</th>
<th>Diverging control flow</th>
<th>Load imbalance within thread blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMM – Dense matrix multiplication</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SpMV – Sparse matrix vector multiplication</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FFT – fast Fourier transform</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Black Scholes – stochastic differential equation model for option pricing</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>LBM – 3D lattice-Boltzmann method</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Low</td>
</tr>
<tr>
<td>Merge sort</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Low</td>
</tr>
<tr>
<td>Stencil – 3D heat equation</td>
<td>No</td>
<td>No</td>
<td>Low</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 6.1: Benchmarks suite used for experiments
The proposed memory hierarchy modeling approach is validated on an NVIDIA Tesla C2050 general-purpose graphics processor.

The benchmarks used for the experiments consist of CUDA implementations of GPU kernels listed in Table 6.1. The benchmarks suite chosen covers GPU kernels commonly used in scientific computing and signal processing applications: dense matrix multiplication (DMM), sparse matrix vector multiplication (SpMV), fast Fourier transform (FFT), Black Scholes, the 3D lattice-Boltzmann method (LBM), merge sort, and a 3D stencil computation. These benchmark also exhibit diverse behavior with respect to memory access patterns, data-dependent control flow and memory references, mix of load and store operations, and the degree of locality. Table 6.1 summarize the functionality of each benchmark and program behaviors that it exhibits.

The overall probability distributions for the L1 and the L2 cache hit rates are generated by the proposed stochastic model for the discussed GPU benchmarks and results are displayed in Figures 6.2 to 6.8. Each histogram shows the probability of a specific predicted hit ratio on the y-axis based on results obtained from 1024 simulations. The dashed vertical red line marks the hit ratios reported by the hardware counters.
Figure 6.2: Probability distribution for the L1 and L2 hit ratios - dense matrix multiply
Figure 6.3: Probability distribution for the L1 and L2 hit ratios – FFT
Figure 6.4: Probability distribution for the L1 and L2 hit ratios – Black Scholes.
Figure 6.5: Probability distribution for the L1 and L2 hit ratios – sparse matrix vector multiply
Figure 6.6: Probability distribution for the L1 and L2 hit ratios – merge sort
Figure 6.7: Probability distribution for the L1 and L2 hit ratios – lattice-Boltzman method
Figure 6.8: Probability distribution for the L1 and L2 hit ratios – stencil
Figure 6.9: Probability distribution for the L1 and L2 hit ratios – merge sort (global)
6.2 Error Quantification: Sampling Error – Precision

I report the average and the standard deviation for each hit ratio probability distribution in Figures 6.2 to 6.8. The standard deviation describes the spread of the predicted hit ratios and determines how random variations affects the sensitivity and reliability of the proposed memory model. A small standard deviation – 0.01 on average for the outputs of the model – indicates that the predictions from the model are robust with respect to the randomly sampled ordering of memory requests.

![Histogram of Hit Ratios](image)

Figure 6.10: Sampling error for L2 read hit ratio – sparse matrix vector multiply.

Based on the Law of Large Numbers (LLN) and the Central Limit Theorem (CLT) – discussed in Section 2.1.1 – the probability of the expected predicted hit ratios being within the the confidence interval $[\hat{E}_n - 2\hat{\sigma}_n, \hat{E}_n + 2\hat{\sigma}_n]$ is 95%. For example, L2 hit ratio predictions for the SpMV kernel shown in Figure 6.10 exhibit an average of 0.22 with a standard deviation of 0.02. Over large number
of trials, the true average is expected to fall within interval $[0.18, 0.26]$ with a probability of 95%.

### 6.3 Error Quantification: Systematic Error – Accuracy

I define the absolute error function $E_{\text{abs}}(h, \hat{h}) = |h - \hat{h}|$ for measured ($h$) versus estimated ($\hat{h}$) hit ratios, where $h, \hat{h} \in [0, 1]$. Relative error, $E_{\text{rel}}(h, \hat{h}) = \frac{|h - \hat{h}|}{h}$, is a more meaningful measure of accuracy than the absolute error when the predicted values are large and unbounded. For the case presented here, relative error does not relate closely to the performance associated with the difference in the predicted cache hit ratios. For example, using the relative error one gets the same accuracy for $E_{\text{rel}}(0.02, 0.04)$, $E_{\text{rel}}(0.2, 0.4)$ and $E_{\text{rel}}(0.5, 1.0)$. The fact that the above errors would be judged to have equal importance is not correct from microarchitecture point of view. Furthermore, $E_{\text{rel}}$ would result in irrational high errors for very small values of $h$ that are close to zero. Therefore, I use absolute error as a measure of accuracy.

I cross-validated the results predicted by the memory hierarchy model with those read from the hardware performance counters provided by the NVIDIA. The performance counters – listed in Table 2.1 – cannot be read or sampled during the kernel execution. Before running the kernel, an environment variable is set to activate the counters. After the kernel execution is finished, the values of the counters that reflect the overall performance statistics with respect to all memory references are written into a log file. The vertical dashed lines in Figures 6.2 to 6.8 highlight the hit ratios computed based on values read from the hardware counters. When compared against hardware counters, the proposed approach have an average absolute error of 3.4% for the L1 read hit ratios, 1.9% for the L2 read hit ratios and 1.0% for the L2 write hit ratios.
6.4 Systematic Error Function Estimation

Experimental measurements are subject to inaccuracy and errors. When the error is large enough such that its impact is noticeable on estimations, it is appropriate to report and account for the uncertainty of the results.

Each predicted estimation for the kernel execution time contains an additive error component. In this case, since it is infeasible to calculate the error function in closed form, I will provide a statistical formulation of the error function based on Bayesian analysis, introduced in Section 2.2. In what follows, the error function is represented as a random variable. In general, to describe a random variable, one needs to have access to its corresponding probability density function (PDF), which expresses the relative likelihood of the function taking a given value.

To estimate the PDF of the error, I use a Bayesian curve fitting approach. The Bayesian method updates a prior PDF (of the error function) with a set of new observations to form a new posterior PDF. The resulting posterior distribution will optimally formulate all the observed “knowledge.”

Since the error function is initially unknown, it is necessary to come up with a prior PDF, which is constructed based on assumptions that are theoretically compatible with a reasonable error model. In what follows, I summarize the assumptions made in design of the error PDF:

1. I consider each of the observed errors (for the predictions) as a ”data point. These data points are expressed in the percentage values (which can be higher than 100%). Let \( T \) be the first integer value greater than the maximum value among such data points.

The interval \([0, T]\) is divided into \( T \) subintervals, each of length 1. For example, with a maximum observed error of 19.7\%, \([0, 20]\) is divided into subintervals \([0, 1), [1, 2), \cdots, [19, 20)\).
2. The likelihood of a data point (error) greater than $T$ drops exponentially.

3. In the absence of additional prior knowledge, I assume a uniform prior distribution for the error (points) in the interval $[0, T)$. However, note that the computed Bayesian posterior will not be uniformly distributed.

A reasonable prior PDF for the error that is consistent with the assumptions stated in items 1-3 above is the following:

$$f_{\text{prior}, e}(x) = \begin{cases} 
\beta \mathcal{I}_{[x]} & \text{if } 0 \leq x < T \\
e^{-\alpha(x-T)} & \text{if } x \geq T
\end{cases}$$

(6.1)

where, the indicator function $\mathcal{I}_{[x]}$ is the probability density of error occurring in the subinterval $[\lfloor x \rfloor, \lfloor x + 1 \rfloor)$. The real numbers $\alpha$ and $\beta$ are chosen appropriately as described in the following.

For $f_{\text{err, prior}}(x)$ to be a proper PDF [43], its integral over $(-\infty, +\infty)$ should
sum up to 1. The area under the exponentially decaying function $e^{-\alpha(x-T)}$ is equal to:

$$\int_{T}^{+\infty} e^{-\alpha(x-T)} \, dx = \frac{1}{\alpha}$$

(6.2)

For the prior PDF, I assume that the chances of error occurring within interval $[0, T)$ is $T$ times that of $[T, +\infty)$. This implies that $\beta = \frac{1}{\alpha}$. Therefore, the area under $f_{err,prior}(x)$ will be equal to:

$$\lim_{x \to +\infty} F_{err,prior}(x) = \int_{0}^{+\infty} f_{err,prior}(x) \, dx = \frac{T}{\alpha} + \frac{1}{\alpha} = 1 \implies \alpha = T + 1$$

(6.3)

A reasonable prior PDF – shown in Figure 6.11 – can be defined by initializing $T$ to 100 percent.

Next, I compute the error for the predicted hit ratios for each of the discussed benchmarks. The obtained errors serve as observations to update the prior PDF. To this end, the value of $T$ is updated according to the maximum error observed. For example, this upper limit value for the L1 hit ratios in the studied benchmark suite is equal to 8%.

Based on the Bayes rule, the probability (likelihood) of the observed error (data points) is proportional to the number of times that they have occurred (among the observations). To avoid overfitting the posterior PDF to the observed data [43], all the $T$ subinterval defined over $[0, T)$ are given a small and equal probability. Therefore, given $S$ sample points, the area under $f_{err,post}(x)$, the posterior PDF, is defined as:
\[ \lim_{x \to +\infty} F_{\text{err,post}}(x) = \int_{0}^{+\infty} f_{\text{err,post}}(x) \, dx = \frac{T}{\alpha} + \frac{1}{\alpha} + \frac{T \times S}{\alpha} = 1 \quad (6.4) \]

\[ \Rightarrow \alpha = T + 1 + T \times S \]

Here, each length-1 subinterval in \([0, T)\) and all the subintervals in \([T, +\infty)\) have a \(1/\alpha\) chance of accommodating the error. In addition, each observed data point effectively increases the probability of its corresponding subinterval by a factor of \(T\). Based on the observed error data and parameters derived from Equation 6.4, probabilities of \(I_x\) for \(x \in \{0, 1, \cdots, T - 1\}\) are updated accordingly. Note that the error function defined here is one realization of the general form described earlier in Definition 6.1. The relative weight of the area under each segment of the curve can be adjusted to tune the sensitivity of the posterior function with respect to the observed data points.

Figure 6.12 shows the PDF of posterior error function computed for the L1 read hit ratios, L2 read hit ratios and L2 write hit ratios respectively. With PDF of the systematic error computed as described above – for a new predicted value, e.g., \(\hat{h}\) in case of cache hit ratios – one can generate a range of true values that the new predicted value is likely to occupy. Therefore, the prediction result can be expressed as PDF of the true value. Application of PDF of the prediction error is covered in more detail in Section 8.2 when I discuss the probabilistic merit of different application kernel configurations.
Figure 6.12: Probability distribution of systematic error for predicted L1 and L2 hit ratios
CHAPTER 7

HIGH RESOLUTION PERFORMANCE STATISTICS

In this chapter, I discuss how high resolution performance statistics are generated by coupling source code level information and the results of the memory hierarchy simulation model. This comprehensive view of the level of memory efficiency exploited by individual data structures or memory operations within a particular code segment is a crucial step for targeted memory optimizations.

I later illustrate though a simple example how this low level information can highlight inefficient memory accesses and play up the data structures that are ideal candidates for optimization.

7.1 The Average Latency per each Static Load

In Section 4.2.3, I presented a micro-benchmarking approach to measuring the main memory access time, $T_m$, for each application. In Section 3.6.4, I discussed my approach to measuring the L1 and the L2 cache latencies, denoted by $t_1$ and $t_2$, with a simple micro-benchmark. I later summarized the overall probabilistic predicted hit ratios by the proposed memory hierarchy model for the L1 and L2 caches, denoted by $H_1$ and $H_2$, in Figures 6.2 to 6.8.

In this section, I combine all these results using the following simple access latency equation:

$$X(H_1, H_2, T_m) = H_1 \cdot t_1 + (1 - H_1) \cdot [H_2 \cdot t_2 + (1 - H_2) \cdot T_m] \quad (7.1)$$
In the above equation, $X$, $T_m$, $H_1$, $H_2$ are random variables; their probability density function (PDF) can vary from one application to the other or, from one static memory operation in the source code to another.

Since I propagate down the source code location of each memory operation, I can obtain the PDF for individual static loads in the code. Based on that information, for example, I can estimate the expected access latency, $E(X)$, for each static load in the kernel source code. Figures 7.1, 7.3 and 7.2 show the breakdown of the expected memory latency with respect to each static load for LBM, SpMV, and Stencil benchmarks. This information can be used to target memory optimizations such as tiling, thread coarsening and data layout transformation for specific memory operations or data structures in the program.
7.2 Case Study: Sparse Matrix Vector Multiplication Kernel

In this section I show through an example how high resolution performance information can be used to apply targeted optimizations. Initial examination of the sparse matrix vector multiplication kernel source code shown in Listing 7.1 implies that loads from lines 11 and 13 each exhibit a high level of intra-thread temporal locality as they are executed within a tight loop using consecutive indices. Therefore, the latency for these loads is expected to be low. On the other hand, memory references from the load in line 12 are not analyzable statically and expected to follow an irregular pattern resulting in poor data locality.

Table 7.1 summarizes information with respect to memory loads for this kernel

Figure 7.2: Breakdown of memory latency for static loads in Stencil
float t = 0;
unsigned int myi = bid * BLOCKSIZE + tid;
if ( myi < numRows ){
  unsigned int lb = rowIndices[myi];
  unsigned int ub = rowIndices[myi+1];
  for (j = lb; j < ub; j++) {
    unsigned int ind = indices[j];
    yval = y[ind];
    t += val[j] * yval;
  }
  x[myi] = t;
}

Listing 7.1: Sparse matrix vector kernel

float t = 0;
unsigned int myi = bid * BLOCKSIZE + tid;
if ( myi < numRows ){
  unsigned int lb = rowIndices[myi];
  unsigned int ub = rowIndices[myi+1];
  // prologue code — alignment adjustment
  for (j = newlb; j < newub; j+=4) {
    uint4 ind = indices[j/4];
    float4 value = val[j/4];
    float yval = y[ind.x];
    t += value.x * yval;
    yval = y[ind.y];
    t += value.y * yval;
    yval = y[ind.z];
    t += value.z * yval;
    yval = y[ind.w];
    t += value.w * yval;
  }
  // epilogue code — alignment adjustment
  x[myi] = t;
}

Listing 7.2: Vectorized sparse matrix vector kernel
Figure 7.3: Breakdown of memory latency for static loads in SpMV

and Figure 7.3 shows the predicted load latency for each of the static loads of SpMV kernel – generated by the stochastic memory hierarchy model – considering the interactive effects of all concurrent memory operations. Results reported in Figure 7.3 are different from the speculation derived based on manual source code examination. Based on this new information when considering the interactions among the concurrently running threads, loading the vector value (line 12) is more efficient compared to the other two static loads that initially exhibit a high reuse pattern. This observation makes these two loads the most promising candidates with respect to memory optimization.

Listing 7.2 shows an optimized version of the SpMV kernel with intra-thread vectorization applied to loads initially from lines 11 and 13 to exploit the available
<table>
<thead>
<tr>
<th>Static Load ID</th>
<th>Corresponding line in Listing 7.1</th>
<th>Avg. Dynamic count per thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>51</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 7.1: Dynamic load count per static load in SpMV kernel

data reuse that cannot be efficiently captured by the available data caches. The optimized kernel uses just enough extra registers to maintain the initial occupancy level of streaming multiprocessors, which combined with more efficient memory accesses results in a speedup of 2.9 compared to the initial kernel configuration.
In the previous chapter I discussed how the expected memory latency for a static load in the GPU kernel is computed based on the estimated L1 and L2 hit ratios associated with it and the expected main memory latency for the kernel. Being able to identify the least performing set of memory operations in the code is a valuable piece of information for an application developer or a compiler. In this chapter, I will take that information one step further to determine the dominating detrimental factor with respect to the performance of a kernel. Particularly, the goal is to determine to what degree a GPU kernel is memory bound, if at all.

8.1 Compute and Memory Time

I decompose the execution time of a GPU kernel into the *compute time* and the *memory time*. The former is the time to complete the computation assuming an ideal memory system, while the latter is the time needed to flow all data required by computational cores from memory.

As mentioned in Section 5, the instrumentation framework collects an estimate of the number of dynamic instructions executed during the kernel runtime by inserting probes at control flow entry and exit points. The number of instruction that the kernel executes between two control flow probes is estimated by the number of the abstract syntax tree nodes (representing actual computation) traversed when control flow proceeds from one probe to the other. It should be noted that
this instruction count may not reflect the following properties of the computational instructions:

- The actual number of instructions executed by the hardware – For example, a multiplication followed by an add may be converted into a multiply-add instruction by the compiler.

- Different types of instructions – A double precision multiply and a jump instruction are each counted as one instruction, though the streaming cores’ execution bandwidth with respect to each is different.

- Data dependencies – Memory or back-to-back register dependencies are not reflected by the instruction count.

The above limitations does not allow for a detailed instruction scheduling approach, but it will provide a relatively reasonable estimate to capture the first order performance degradation factors as indicated by the results discussed in Section 8.2.

The compute time of a kernel is estimated by examining the dynamic instruction count from the collected traces for a snapshot of the kernel execution. An average dynamic instruction count for warps of a thread block is computed based on the above information. The GPU hot clock frequency, which is the execution speed of individual streaming cores, is identified through a call to the CUDA programming APIs. Occupancy and the number of available streaming multiprocessors are similarly computed as describe earlier in Listing 3.1.

8.1.1 Memory Level Parallelism

Memory-Level Parallelism (MLP) is the ability to simultaneously service multiple memory transactions. The analytical model treats each streaming multiprocessor
as a number of in order cores that exploit memory level parallelism (MLP) through concurrent memory accesses issued from a set of coexisting threads. Coexisting threads are congregated from:

1. threads tightly coupled together through a SIMD memory vector instruction
2. threads executed tightly close together as a result of interleaving the execution of coexisting warps

To compute the memory time in the presence of the MLP, the analytical model makes the following assumptions:

1. A warp with outstanding memory transaction(s) initiated by a previous memory load cannot issue any new memory loads until all previous outstanding transactions are resolved.
2. The computation within a kernel is divided into a sequence of memory loads followed by a sequence of other type of instructions. A dynamic instruction stream, in the presence of enclosing loop constructs, generates separate sequences of memory and compute regions that are executed sequentially.
3. Graphics processors based on the Fermi architecture, support up to 48 concurrent warps on each streaming multiprocessor. And each streaming multiprocessor contains 32 in order cores each capable of issuing an independent memory transaction. Therefore, it is reasonable to assume a total of 32, 48, or 64 outstanding loads ($L_{os}$) per multiprocessor. Based on an error minimization process, I assume a maximum of 48 outstanding loads for the experimental results presented in this section.
4. As thread blocks enforce implicit synchronization points at their entry and exit points, any measurement of critical path or resources should be conducted at a thread block level.
Assumption 1 is a reasonable simplification when the amount of inter-thread MLP is large enough to consume the corresponding hardware resources, i.e., the number of outstanding loads. Thus, the model overlaps the latency of loads from different threads and serializes the intra-thread load latencies as shown in Figure 8.1. This is a reasonable assumption for most memory bound GPU applications as they are composed of a large number of concurrently running threads that can easily saturate the hardware limits. In cases that the inter-thread MLP – computed later in Equation 8.1 – is smaller than $L_{os}$, i.e. $\frac{L_{os}}{MLP} > 1$, up to $\lceil \frac{L_{os}}{MLP} \rceil$ memory loads – if they exist – within a memory region can be issued together, which results in a shorter memory time critical path. Accordingly, the initial inter-thread MLP computed in Equation 8.1 is also updated to reflect the exposed intra-thread MLP.

With the above assumptions, the degree of inter-thread MLP presented in a kernel or in other words, the number of inflight memory transactions, is determined by:

**Coalescing Factor:** Coalescing Factor (CF) represents the number of distinct memory transactions issued on average for a memory load operation in a kernel. For example, initially a coalescing factor of one means that memory locations referenced by all threads in a warp are mapped to a single cache block. The coalescing factor can vary from one memory load to another within a kernel. To capture the overall behavior, an average CF per warps of a thread block is computed over the total number of dynamic loads collected for the snapshot of the kernel execution and is then multiplied by the number of warps per thread block.

**Warp Level parallelism:** Warp Level Parallelism (WLP), analogous to the notion of Thread Level Parallelism (TLP), refers to the number of warps that
are being executed concurrently on a GPU. The execution of concurrent
warps is heavily interleaved by the warp scheduler. Therefore, it is reason-
able to assume that memory operations from concurrently executing warps
are issued tightly close together. This is specially true if the kernel is mem-
ory bound.

Based on the described assumptions and definitions, the available inter-thread
MLP within a kernel is estimated by:

\[
MLP = \frac{CF \times WLP}{TB_{warps}}
\]  

(8.1)

where, \( TB_{warps} \) is the number of warps per one thread block.

Based on dynamic memory traces, the expected L1 and L2 hit ratios, and the
main memory access latency, the critical path with respect to memory cycles
(time) is identified for each thread block. Different warps within a thread block
may have different dynamic number of memory loads, which results in load im-
balance with respect to the number of memory operations. Threads within a thread
block synchronize once at the beginning of the thread block execution and once
when the thread block execution ends. Therefore, the critical path analysis is con-
ducted in the granularity of a thread block.

8.1.2 Utilization of the Available Memory Level Parallelism

According to the execution model that I described earlier, GPUs exploit inter-
thread MLP to overlap the memory latency of loads issued from coexisting threads.
Figure 8.1 illustrates the case when the MLP available within a kernel can be
fully exploited by overlapping the latency of all memory operations (of different
threads) ready to be issued.
Figure 8.1: Scheduling of memory and compute cycles

For the sake of simplicity, assume there are a total of four concurrent thread blocks. The kernel in this example has 3 static loads – denoted by L_0, L_1, and L_2. The expected memory latency varies from one static load to the other and is proportional to the length of the vertical blocks in Figure 8.1. Static loads are repeated according to the dynamic control flow pattern present within (the warps of) a thread block. As a result the computation within a thread block is divided into multiple segments or regions. Each region starts with a set of memory loads that read the required data. When the data is ready, the region starts its computation. A kernel consists of one or more memory regions each followed by a compute
region. A sequence of memory-compute regions can occur when there is a loop within the body of a kernel.

To estimate the critical memory path (time), multiple (equal to the thread-block occupancy of a kernel) sample thread blocks are chosen. The length of the critical paths for memory access time of the sampled thread blocks are computed by scheduling their dynamic memory loads back-to-back and then adding up the estimated memory access latency of each memory load (based on its static ID). The critical paths are then averaged to obtain the expected total memory access latency for a typical thread block, $T_{mem\,path}$. Next, memory time for a set of concurrently running thread blocks (threads) is estimated by the following equation:

$$T_{mem} = T_{mem\,path} \times \max(1, \frac{MLP}{L_{os}})$$

(8.2)

The inter-thread MLP is estimated by Equation 8.1. Equation 8.2 states that memory access time for concurrently executing threads can be overlapped as long as enough hardware resources (outstanding load entries) exist to accommodate the inter-thread MLP available within a kernel. Otherwise, the memory level parallelism is repressed, which results in serialization of a portion of memory accesses. Figure 8.2 illustrates this serialization effect with an example similar to the one shown in Figure 8.1. The difference between the two cases is that in Figure 8.2 CF is increased such that the available inter-thread MLP is larger (almost twice in this example) than $L_{os}$. Therefore, memory accesses from different thread blocks (threads) are serialized. Equation 8.2 accounts for this serialization effect via the term $\frac{MLP}{L_{os}}$.

Figure 8.1 also shows how execution of compute cycles – represented by darker blocks – is interleaved with the latency of memory loads in the absence of ordering constraints enforced via synchronizations points. If the kernel is memory bound,
compute time is fully overlapped with the memory time, except for the compute time of the last region of a thread block.

As mentioned before, compute time is computed based on an estimate of the
number of instructions nodes in the abstract syntax tree. These instructions and consequently the count associated to them can be different from the actual stream of instructions being executed by the hardware. Therefore, it is more appropriate to assume a bulk scheduling model for their execution. Following this assumption and the concepts of memory and compute regions introduced before, the total count of instructions – which corresponds to the compute time $T_{comp}$ – is amortized over the total number of regions as follows:

$$T_{region} = \frac{T_{comp}}{N_{regions}}$$

(8.3)

Note that $T_{mem}$ corresponds to the memory time for all the concurrently running thread blocks (threads), while $T_{comp}$ account for compute time of only on thread block. Therefore, for a set of concurrently running thread blocks on a streaming multiprocessor, total compute time is equal to:

$$T_{comp, total} = \frac{T_{comp} \times WLP}{TB_{warps}}$$

(8.4)

1. If $T_{comp, total}$ for a kernel is less than the corresponding $T_{mem}$, the kernel is memory bound and the execution time for a single set of concurrently executing thread blocks – one execution window – can be estimated by:

$$T_{window} = T_{mem} + T_{region} \times \frac{WLP}{TB_{warps}}$$

(8.5)

According to the above equation, the overall execution time critical path is composed of memory time – latency of concurrently issued memory loads from simultaneous threads – and the time required to execute the last compute block of all the thread blocks scheduled on the same streaming multiprocessor.
2. For compute bound kernels, compute time, $T_{\text{comp, total}}$, dominates the execution time of a kernel. Consequently, $T_{\text{window}}$ is equal to $T_{\text{comp, total}}$.

3. In the presence of synchronization points, the critical path for the execution time of a kernel is constructed based on the assumption that memory and compute times cannot be interleaved. This assumption is accurate with respect to the warps within a thread-block as synchronization points are inserted between memory regions and compute regions. Therefore the latency of compute instructions cannot overlap with the latency of memory loads from the same thread block. With respect to different thread blocks, the assumption is still reasonable since now both compute time and memory time are part of the critical path. For example, if compute time of the current thread block is overlapped with the memory time of another thread block, part of the memory time of the current thread block that was supposed to be overlapped with the memory time of the second thread block is now exposed.

Therefore, $T_{\text{window}}$ is computed as:

$$T_{\text{window}} = T_{\text{mem}} + T_{\text{comp, total}}$$  \hspace{1cm} (8.6)

To obtain the total execution time of a kernel, $T_{\text{window}}$ computed by the above equations is multiplied by the total number of execution windows for a kernel, denoted by $N_{\text{windows}}$, which is equal to:

$$N_{\text{windows}} = \frac{N_{TBs}}{N_{SMs}} \times \frac{WLP}{N_{warps}}$$  \hspace{1cm} (8.7)

The model discussed in this section treats loads from the shared memory the same as global memory accesses. Shared memory accesses are even coalesced
to account for bank conflicts and accesses that result in multiple shared memory loads. The only difference is that references to shared memory have the fixed memory access latency that is equal to the L1 cache hit access time.

Figure 8.3 shows the scatter plot for the predicted versus measured execution time for the benchmark suite discussed in Section 6. The results indicates an average relative error of 16%, with the highest prediction error of 27.1%.

![Figure 8.3: Scatter plot of predicted versus measured execution times](image)

8.2 Probabilistic Merit of a Kernel

In Section 6.4, I discussed a curve fitting approach to estimate the PDF of the systematic error of a model. Following a similar discussion a posterior PDF is computed for the relative error of the predicted execution time for each of the
discussed benchmarks; the obtained relative errors serve as observations to update the prior PDF introduced in Section 6.4. Figure 8.4 shows the PDF of the computed posterior error function, $f_{err,post}(x)$.

Now that the error function, $f_{err,post}(x)$ is defined, for a predicted execution time, $\hat{t}$, one is interested in the range of true values that the execution time is likely to occupy. The PDF for the true value of the execution time, $t(E)$, where $E$ is the random variable representing the relative error with PDF of $f_{err,post}(x)$, is defined as follows:

$$\frac{|t(E) - \hat{t}|}{t} = E \quad \Rightarrow \quad t(e) = \begin{cases} \frac{i}{1+E} & E \geq 1 \\ \frac{i}{1+E} - \frac{i}{1-E} & E < 1 \end{cases}$$ (8.8)

The PDF $t(E)$ expresses the range representing the probable value of the true execution time. Therefore, when comparing the relative merit of two predicted
values \( \hat{t}_1 \) and \( \hat{t}_2 \), instead of asking the question, "Is \( \hat{t}_1 \leq \hat{t}_2 \)?", the question is rephrased to: "What is the probability of \( \hat{t}_1 \leq \hat{t}_2 \)?" In other words, the goal is to evaluate \( \mathbb{P}(\hat{t}_1 \leq \hat{t}_2) \) and not to determine whether \( \hat{t}_1 \leq \hat{t}_2 \).

To compute the probability of \( \mathbb{P}(\hat{t}_1 \leq \hat{t}_2) \), \( f_{\text{err,post}}(x) \) is sampled and based on each collected data point for error, the potential value(s) for each \( t(E) \) are computed. The number of the data points collected determine the resolution of the sampling. In this work, I choose 500 sample data points, which results in a total of 1000 sampled values for each pair of \( t_1(E) \) and \( t_2(E) \). The number of samples chosen from each subinterval \([\lceil e \rceil, \lceil e + 1 \rceil]\) is proportional to the likelihood of random variable \( E \) occurring within that interval.

Next, the \( n \) sampled values for \( t_1(E) \) and \( t_2(E) \) are sorted in ascending order as \( S_{t_1} = \{t_{1;0}, t_{1;1}, \ldots, t_{1;n-1}\} \) and \( S_{t_2} = \{t_{2;0}, t_{2;1}, \ldots, t_{2;n-1}\} \) respectively such that \( t_{k;i} \leq t_{k;i+1} \).

To compute the probability \( \mathbb{P}(\hat{t}_1 \leq \hat{t}_2) \) for each sample point \( t_{1;i} \) from \( S_{t_1} \), the following steps are performed:

1. Find the first sample point \( t_{2;j} \) in \( S_{t_2} \) such that \( t_{1;i} < t_{2;j} \). If no such \( t_{2;j} \) exists set \( j = n \).

2. Let \( p_i = \frac{1}{n} \times \frac{n-j}{n} \). The term \( p_i \) represents the probability of choosing the sample point \( t_{1;i} \) in \( S_{t_1} \) and the value of \( t_{1;i} \) being less than or equal to a sample point chosen in \( S_{t_2} \).

Consequently, \( \mathbb{P}(\hat{t}_1 \leq \hat{t}_2) \) is the sum of \( p_i \) terms computed above:

\[
\mathbb{P}(\hat{t}_1 \leq \hat{t}_2) = \sum_{i=0}^{n-1} p_i
\]  

\[(8.9)\]

\(^1\) \( P(X) \) is defined as the probability of event \( X \).
8.3 Case Study: 7-Point Stencil

In this section, following the previously discussed approach for evaluating the merit of a kernel, I will investigate three optimized version of the stencil code, which represent important class of scientific computation such as partial differential equation solvers that work based on structured grids.

In a general stencil code, data is arranged in a regular two or tree dimensional grid. The physical location of the grid can be directly derived from the indices of the array elements. Owing to the regular access patterns that can be statically determined and a low computational intensity, a number of effective optimization strategies can be applied to the stencil code.

Here, I look into a well studied stencil code [44] representing a three dimensional heat equation initially expressed as:

\[
\]

During each operation a point in the grid is updated by a weighted linear combination of a subset of its neighbors.

8.3.1 Exploiting Locality through the L1 Cache

The first version of the stencil benchmark, shown in Figure 8.1, is an improved version of the naive stencil code that uses one thread to calculate one output point.

In this improved version, as shown in Figure 8.1, each thread computes a one-element column across the z-dimension of the grid. As a result of this tiling optimization, threads reuse the data along the z-dimension. Threads within a warp
(and a thread block) compute a rectangular column along the z-dimension. Therefore, although threads load their input values independently from the global memory, a good portion of the memory accesses get coalesced.

The estimated memory time for this kernel is almost three times of the estimated compute time, indicating a memory bound kernel.

According to the core computation discussed above, each internal point contributes to calculation of seven output values – itself, 4 planar neighbors, and the top and bottom grid points. Corner points are used for fewer output values. With this memory access pattern, the only justifiable reason for a kernel to be memory bound is that the working set for the concurrently running thread blocks exceeds the effective capacity of the L1 cache and the existing temporal locality is not fully exploited.

An effective optimization commonly used for such a scenario is tiling. Therefore, the next natural choice of the optimized kernel is one that is tiled across the xy plane.

8.3.2 Two Dimensional Tiling in Shared Memory

To keep the working set of the concurrently running threads close to the streaming cores, the new version of the kernel uses shared memory to store grid points of the three active planes (top, current, and bottom) across x and y dimensions.

In the kernel shown in Listing 8.2, after each phase of computation along the z-dimension, threads update the content of the shared memory with the elements of the next three relevant slice along the z-dimension – currently hold locally in registers.

Since corner points are used for fewer output values and considering the limited capacity of the shared memory, only the the middle elements of a plane are stored
-_global_- void block2D_tiling(float c0, float cl, float *A0,
    float *Anext, int nx, int ny, int nz)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i > 0 && j > 0 && (i < nx - 1) && (j < ny - 1)) {
        float bottom = A0[i + nx * (j + ny * 0)];
        float current = A0[i + nx * (j + ny * 1)];
        for (int k = 1; k < nz - 1; k++) {
            current = top = A0[i + nx * (j + ny * (k + 1))];
            Anext[i + nx * (j + ny * k)] =
                (top + bottom +
                 A0[i + nx * (j + 1 + ny * k)] +
                 A0[i + nx * (j - 1 + ny * k)] +
                 A0[i + 1 + nx * (j + ny * k)] +
                 A0[i - 1 + nx * (j + ny * k)]) * cl
                - current * c0;
            bottom = current;
        }
        current = top;
    }
}

Listing 8.1: Stencil code loading data from the L1 and L2 caches

in the shared memory. The rest of the elements are loaded directly from the main
memory.

The estimated memory time for this version drops almost 70% compared to the
previous version. But the compute time increases by 40%. Since threads synchro-
nize frequently (after each step of computation), in this version both memory time
and compute time contribute to the total execution time. As a result, the over all
execution time reduction is only 4% compared to the previous version.

8.3.3 Thread Coarsening

Based on the data collected from the model, previous kernel with 2-dimentional
tiling in shared memory, observed an increase in the compute time portion of the
execution time.

A commonly performed optimization to reduce redundant calculations or mem-
```c
__global__ void block2D_hybrid(float c0, float c1, float *A0, float *Anext,
                               int nx, int ny, int nz)
{
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    const int j = blockIdx.y * blockDim.y + threadIdx.y;
    const int sh_id = threadIdx.x + threadIdx.y * blockDim.x;

    // shared memory
    extern __shared__ float sh_A0[];
    sh_A0[sh_id] = 0.0f;
    __syncthreads();

    // compute regions for load and store
    const bool region = i > 0 && j > 0 && (i < nx - 1) && (j < ny - 1);
    const bool x_l_bound = (threadIdx.x == 0);
    const bool x_h_bound = (threadIdx.x == (blockDim.x - 1));
    const bool y_l_bound = (threadIdx.y == 0);
    const bool y_h_bound = (threadIdx.y == (blockDim.y - 1));

    // register for bottom and top planes
    float bottom = 0.0f, top = 0.0f;

    // load data for the bottom and current planes
    if (i < nx) {
        bottom = A0[Index3D(nx, ny, i, j, 0)];
        sh_A0[sh_id] = A0[Index3D(nx, ny, i, j, 1)];
    }
    __syncthreads();

    for (int k = 1; k < nz - 1; k++) {
        float a_left, right, a_right, left, a_top, a_down;
        // load required data on xy planes
        // if it is in the shared memory, load from shared memory
        // otherwise, load from the global memory
        if ((i < nx) && (j < ny))
            top = A0[Index3D(nx, ny, i, j, k + 1)];
        if (region) {
            a_left = x_l_bound ? A0[Index3D(nx, ny, i - 1, j, k)] : sh_A0[sh_id - 1];
            a_right = x_h_bound ? A0[Index3D(nx, ny, i + 1, j, k)] : sh_A0[sh_id + 1];
            a_down = y_l_bound ? A0[Index3D(nx, ny, i, j - 1, k)] : sh_A0[sh_id - blockDim.x];
            a_top = y_h_bound ? A0[Index3D(nx, ny, i, j + 1, k)] : sh_A0[sh_id + blockDim.x];
            Anext[Index3D(nx, ny, i, j, k)] = (a_left + a_right + a_top +
                                              a_down + top + bottom) * c1 - sh_A0[sh_id] * c0;
        }
    }

    // swap the data
    __syncthreads();
    bottom = sh_A0[sh_id];
    sh_A0[sh_id] = top;
    __syncthreads();
}
```

Listing 8.2: Stencil code using shared memory for elements in the middle, and L1 and L2 for the hollows
<table>
<thead>
<tr>
<th>Version</th>
<th>Memory time %</th>
<th>Predicted bottleneck</th>
<th>Predicted execution time improvement %</th>
<th>Measured execution time improvement %</th>
</tr>
</thead>
<tbody>
<tr>
<td>No shared memory</td>
<td>100%</td>
<td>memory time</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Shared memory 2D tiled</td>
<td>46%</td>
<td>compute time</td>
<td>4%</td>
<td>≈ 0</td>
</tr>
<tr>
<td>Coarsened shared memory 2D tiled</td>
<td>52%</td>
<td>memory time</td>
<td>11%</td>
<td>21%</td>
</tr>
</tbody>
</table>

Table 8.1: Summary of predicted performance information for the Stencil kernels

ory accesses is thread coarsening. Thread coarsening consists of packing the computation of multiple fine grain threads into a more coarse grained one to avoid redundant work. The downside is that in order to perform redundant computation once and save the results, more registers are required. Increase in register usage may decrease the occupancy and utilization of the streaming cores.

To investigate this potential optimization, the previous shared memory tiled version of the stencil kernel is coarsened (tiled) across the x-dimension. Now each thread performs twice the amount of work it used to do. The coarsened version is run through the performance modeling framework. The result of the analysis indicates that occupancy is not affected by the increase in the number of registers used and that the compute time improves by 16% compared to the shared memory tiled version. The overall execution time is reduced by 6% and 11% compared to the shared memory tiled and the initial version respectively.

Table 8.1 summarize the performance results collected from the model and the GPU hardware for the three discussed kernels: The second column reports the contribution of the predicted memory time to the total predicted execution time for each kernel, based on which, the major time consuming component is identified in column 3. Columns 4 and 5 measure the amount of performance improvement for predicted (model) and measured (hardware) execution times with respect to

97
Table 8.2: Comparison of the probabilistic relative error of the stencil kernels – reported probability for each pair of $X$ and $Y$ correspond to $P(X \leq Y)$.

As discussed in Section 1.5, expressing the predicted performance or execution time through to a single number without accounting for the potential deviation of the output of the model from the real execution time is not appropriate. Therefore, based on the approach discussed in Section 8.2 the relative probabilistic merits of the three stencil kernels are computed and summarized in Table 8.2.

Probabilities reported in Table 8.2 put the raw prediction numbers into perspective with respected to the previously observed error in the model. For example, according to the data presented in Table 8.2, chances of the coarsened stencil kernel outperforming the initial version is 66%. The raw execution time reported by the model are 25.95\(ms\) and 29.10\(ms\) respectively. Actual measurement on the GPU results in corresponding execution times of 22.60\(ms\) and 29.00\(ms\), which follows the same trend predicted by the model. But one needs to account for the potential additive error reported along with the prediction times. After perturbing the raw predictions with the potential error, a more realist comparison states that likelihood of version 3 performing better than the first kernel is 66%.

In general, the larger the magnitude of the difference between two predicted execution times is, given a PDF for the error, the higher is the chance of circumventing the uncertainty of predictions caused by the systematic error present in the model.
CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

This work presents a novel solution to the problem of providing meaningful performance feedback to developers in highly multithreaded graphics processors.

The stochastic modeling approach presented allows for using a simple and efficient tracing scheme without user intervention and without concerns for distorted execution timing. It further provides confidence level information in the presence of scheduling uncertainties. Its sampling strategy also allows one to minimize the cost of tracing and simulation. The close match between the generated distribution mean and the sampled hardware performance counter values provides good validation for the proposed stochastic memory hierarchy model.

The high resolution performance statistic generated through coupling source code level instrumentation and the memory hierarchy simulation model provides a comprehensible view of the level of memory efficiency being exploited by individual data structures in the program. This information, though still far from targeted optimization hints to a developer or compiler, is a crucial and fundamental step forward towards that goal.

The analytical model built on top of the stochastic memory hierarchy model, on the other hand, provides a big picture overview of the overall performance and the first order bottlenecks in a GPU kernel.
9.2 Future Work Direction

There are several avenues for future work. While the model is demonstrated and validated based on CUDA applications and the Tesla C2050 hardware, it can potentially be applied to other highly multithreaded processors too. As a direction for future work, it would be useful to extend the system to OpenCL [2] applications and apply it to the AMD GPUs for further evaluation across a wider range of GPU architectures.

Another avenue for this work is to augment the current framework with an intelligent component to mine the performance statistics and retrieve more insightful information. For example, it would be interesting to collect extra raw performance data from the memory hierarchy model and later analyze it with respect to the interaction of memory accesses to different data structures in the program, or even accesses to the same data structure that are initiated from different locations in the source code.

To evaluate these interactions, it may be beneficial to feed partial traces – memory addresses from specific static memory locations in the program – to the memory hierarchy simulator. Then, one can evaluate the performance of the memory system in the absence of a subset of memory accesses and potentially set guidelines based on which one can optimally map the working set of an application into the proper address spaces available on a GPU. For example, a set of memory references may have detrimental effects – by causing conflict misses – on the existing locality within a data structure that is initially mapped to the global memory. If the subset is isolated and loaded into the shared memory, the detrimental effect of conflict misses may fade away.

Another interesting direction would be investigating the potential speed up and performance improvement of a data parallel computation – not necessarily ex-
pressed as a GPU kernel – on a target GPU computing platform. A systematic approach to solve this problem, in addition to performance analysis techniques presented in this work, requires an efficient mapping of data structures into the available address spaces on the a GPU, e.g., global memory, shared memory, etc.
APPENDIX A

SYMBOLIC EVALUATION

In this chapter, I propose a tractable approach to symbolic evaluation of conditions and array access expressions. The proposed symbolic evaluation system has two components:

1. A symbolic execution engine for evaluating effects of expressions that are simple enough.

2. A set of simplification rules for simplifying more complex expressions.

If the expression to be evaluated is too complicated for the symbolic evaluation engine, the expression is conditionally simplified such that the evaluation of the simplified expression implies results equal to that of the original expression. The simplification process may be applied recursively until tractable expressions are obtained or the simplification engine decides that no further simplification is applicable. In the latter case, the expression is interpreted conservatively, which results in a lower bound for the performance.

During the simplification process certain constraints on free variables may be imposed, e.g., input parameters. The validity of the results is contingent on the truth of the conjectures made about free variables by the simplification system. Therefore, the results of each evaluation step are accompanied by a set of constraints that specify the domain of applicability of each evaluation step.
A.0.1 Structural Conditions

A predicate denotes either a structural condition or a data-dependent condition. A structural conditional expression is a function of thread coordinates, enclosing loop induction variables and symbolic constants of the kernel. In this section I will show how to analyze a large class of structural conditions that are defined by Definition (A.1), through symbolic evaluation. Analyzing data-dependent conditions requires statistical knowledge of the data and is discussed in Section 5.

\[ O(At + B, I(i)), \quad I(i) = \alpha i + \beta \quad \text{or} \quad I(i) = 2^{\alpha i + \beta} \quad (A.1) \]

\[ i \in \{0, 1, \ldots, I\}, \quad t \in \{0, 1, \ldots, T - 1\} \quad \text{and} \quad A, B, \alpha, \beta \in \mathbb{Z} \]

\[ O(x, y) \in \{x = y, x \neq y, x \leq y, x \geq y, x \pmod{y} = 0\} \]

For brevity, I consider one-dimensional thread-blocks of the size T, where T is well-bounded due to hardware limitations. Variable t stands for the thread ID in the affine expression At + B. In Definition (A.1), i is the loop induction variable with the upper bound of I, which may not be known at compile time. Operator O can be either a comparison or a modulo operator. Without loss of generality, I restrict the discussions to \( \leq \) for comparison operators, A, B, \( \alpha, \beta \in \mathbb{N} \), and \( I(i) = \alpha i + \beta \). Conclusions on other cases can be derived similarly. The following two cases apply, based on the type of operator O.

1. \( O \) is a comparison operator: \( At + B \leq \alpha i + \beta \)

   We solve \( At + B \) for each \( t \in \{0, 1, \ldots, T - 1\} \). Let \( At' + B \leq \alpha i + \beta \) for \( 0 \leq t' < T \); we have \( \frac{At' + B - \beta}{\alpha} \leq i \leq I \). Consequently, the total number of steps for symbolic evaluation is given by:
\[
\sum_{t=0}^{T-1} I - \left\lfloor \frac{At + B - \beta}{\alpha} \right\rfloor + 1 \approx \frac{2T(\alpha I - B + \beta + \alpha) - AT(T - 1)}{2\alpha} \quad (A.2)
\]

Notice that for \( I_0 \leq i \), where \( I_0 = \frac{A(T-1)+B-\beta}{\alpha} \), all threads satisfy the inequality. In other words, control flow divergence does not occur for computation steps greater than \( I_0 \). If I replace \( I \) with \( I_0 \) in Eq. (A.2) the upper-bound for the total number of evaluation steps becomes:

\[
\frac{AT^2 - AT + 2\alpha T}{2\alpha} \quad (A.3)
\]

Notice that I have iterated over the thread IDs. To measure the thread ID dependent effects correctly, the active threads for each computation step \( i \) are replayed, then the control flow divergence, coalescing and bank conflict effects are measured. As a result, the total cost is twice the value of Expression (A.3). Notice that if \( I_0 \leq T \), we directly iterate over \( i \). The cost in Expression (A.3) is formulated in terms of program constants or known hardware limits and is proportional to cost of symbolically evaluating the condition. The cost is computed in advance. If the computed cost is less than a predefined threshold, the condition is symbolically evaluated. Otherwise, the condition is assumed to be true for all threads.

Now, let \( w_0 \) be the computed execution weight for the region under the control of the above predicate. To make up for the divergence-free part (the \( I - I_0 \) iterations), the overall weight is computed as:

\[
\frac{\max(I - I_0, 0) + w_0I_0}{\max(I_0, I)} \quad (A.4)
\]

If \( I \) is known at compile time, Expression (A.4) is reduced to a single value, otherwise it remains parametric on \( I \) and consequently the total execution
cycles become a function of $I$. Note that in cases where $I$ is not known statically and its actual value is less than $I_0$, I have approximated the corresponding execution weight by $w_0$.

2. $\mathcal{O}$ is the modulo operator: $(At + B) \equiv 0 \pmod{\alpha i + \beta}$

We solve expression $At + B$ for each $t \in \{0, 1, ..., T - 1\}$. Let $(At' + B) \equiv 0 \pmod{\alpha i + \beta}$ which implies that $\alpha i + \beta$ is some combination of prime factors of $At' + B$. If $AT + B$ is not a very large number we can store combinations of prime factors of all numbers less that $AT + B$ in a lookup table in order to retrieve them efficiently. Let $P$ be a combination of prime factors of $At' + B$. Factor $P$ is equal to $\alpha i + \beta$ if $\frac{P - \beta}{\alpha}$ is an integer and $0 \leq \frac{P - \beta}{\alpha} \leq I$; the converse is also true. Therefore, $t'$ is a candidate thread that satisfies the condition. The total number of steps to test all values of $t$ is bounded by:

$$\sum_{t=0}^{T-1} \left\lfloor \frac{At + B}{2} \right\rfloor \approx \frac{AT(T - 1)}{4} + \frac{BT}{2}$$

Similar to case 1, this method requires two passes to collect the performance information.

Since the largest prime factor that is examined is equal to $A(T - 1) + B$, the condition evaluates to true for all threads if $I_0 \leq i$, where $I_0 = \frac{A(T-1)+B-\beta}{\alpha}$. Similar to the discussion given for the previous case, the overall execution weight is computed as stated in Expression (A.4).

A.0.2 Structural Memory Accesses

In this section I first describe the class of memory access expressions that is accepted by our expression simplification engine. Later I explain how a candidate
expression is simplified through an example. Finally, I discuss my approach to efficiently capture qualitative properties of these expressions from the point of view of GPU architecture performance.

The subscripting function of an array reference, $F$, is a multivariate expression. Function $F$ can be defined recursively as the sum of possibly complex terms according to Definitions (A.5) and (A.6), where $v$ is a vector of integer variables of the program (thread coordinates, input parameters, enclosing loops induction variables, and an element representing integer number 1) all of which are involved in the evaluation of the subscripting expression, and $v[\ell]$ is the $\ell$th element of the variable vector $v$.

$$F^{(0)}(v) = v[\ell] \text{ for some } \ell \in \{1, 2, \cdots, |v|\} \quad (A.5)$$

$$F^{(n)}(v) = \sum_{i=1}^{N(n)} \prod_{j=1}^{M_i(n)} P_{i,j}$$

$$P_{i,j} \in \{O_k \cdot (F^{(n-1)}_{i,j})(v)\} \text{ for } k = 1, 2, 3, 4$$

For some $\alpha, \beta \in \mathbb{Z}$ and some $\ell \in \{1, 2, \cdots, |v|\}$, I define the following four operators, namely $O_1, \cdots, O_4$, on the subscripting function $F$. Note that negative exponent expressions are excluded by definition.

$$O_1 \cdot (F^{(n-1)}_{i,j})(v) = F^{(n-1)}_{i,j}(v) \mod 2^{\alpha v[\ell]+\beta} \quad (A.6)$$

$$O_2 \cdot (F^{(n-1)}_{i,j})(v) = \left\lfloor \frac{F^{(n-1)}_{i,j}(v)}{2^{\alpha v[\ell]+\beta}} \right\rfloor$$

$$O_3 \cdot (F^{(n-1)}_{i,j})(v) = 2^{\alpha v[\ell]+\beta} \times F^{(n-1)}_{i,j}(v)$$

$$O_4 \cdot (F^{(n-1)}_{i,j})(v) = \alpha \times F^{(n-1)}_{i,j}(v)$$

106
The operators defined above are commonly used to express complex access patterns, e.g., wrap-around and strided accesses.

**Example 1.** Expression (A.7) shows a relatively complex index expression from the FFT kernel. Variables involved in this expression include induction variables of two enclosing loops (i and 2^j), input parameter (N), and thread coordinates (b as thread-block ID and t as thread ID).

\[
\left\lfloor \frac{b \times N + t}{2^j} \right\rfloor \times 2^{j+1} + (b \times N + t) \mod 2^j + i \times 2^j \quad (A.7)
\]

The following derivation shows how the first term in Expression (A.7) satisfies the definition of \( F \), where the vector of involved variables is initialized as \( v = [b, t, i, j, N, 1] \):

\[
F^{(2)}(v) = \left[ \frac{F_{1,1}^{(1)}(v)}{2^j} \right] \times 2^{j+1} \times F_{1,2}^{(1)}(v) \\
= \left[ \frac{(F_{1,1})_{1,1}^{(0)}(v) \times (F_{1,1})_{1,2}^{(0)}(v) + (F_{1,1})_{2,1}^{(0)}(v)}{2^j} \right] \\
\times 2^{j+1} \times (F_{1,2})_{1,1}^{(0)}(v) \\
= \left[ \frac{b \times N + t}{2^j} \right] \times 2^{j+1} \times 1
\]

In step two of the derivation above, it is understood that for example, the term \((F_{1,1})_{1,2}\) refers to the second factor of the first term in \( F_{1,1}\).

Expression (A.7) is examined to determine whether the qualitative behavior of residing threads in a memory vector instruction (a half-warp in the case of NVIDIA GPUs) can be confined to their local thread coordinates. Figure A.1(a) shows the expression tree for Expression (A.7). We refer to the multivariate expressions that are composed of thread ID and a combination of other thread co-
ordinates (thread-block ID) with free variables as complex expressions. Through a bottom-up process I label the complex sub-expressions. Labels are propagated upward in the expression tree until a non-distributive operator such as modulo or integer division is reached (referred to as a tainted operator). At this point, the simplification engine attempts to recursively disentangle the local thread ID part from the rest of the terms by imposing constraints on free variables.

We explain this simplification process for the first term of Expression (A.7) which is highlighted in the expression tree of Fig. A.1(a) by dashed arcs. The first non-distributive tainted operator node that is visited during the bottom-up process for this sub-tree is the \texttt{div} operator (\(\lfloor \frac{2}{j} \rfloor\)). Operator \texttt{div} can be distributed if at least one of the terms in its enumerator is a multiple of the denominator \(2^j\). Tainted operator \texttt{div} is distributed speculatively over \texttt{+} operator. From the two resulting sub-expression trees in Fig. A.1(b), only the one on the left contains a free variable. Therefore, speculation continues down the left sub-tree and along the path that leads to the free variable \(N\), as shown in Fig. A.1(c). At this point, the constraint \(N = k \times 2^j\) (for some \(k \in \mathbb{Z}\)) is added to the simplified expression tree of Fig. A.1(c), where \(J\) is the upper bound for the induction variable \(j\). The simplification steps are valid only if the constraint on \(N\) is proved to be true. The proof can be either derived at compile time or checked at runtime. Figure A.1(d) shows the routine distribution of multiplication over addition. In Fig. A.1(e), based on the constraint put on \(N\) earlier, multiplication and integer division have equaled out each other. As the result of these simplifications, the original expression tree is transformed to the one shown in Fig. A.1(f). The local thread coordinate \(t\) is now separated from other thread coordinates such as \(b\). The two highlighted sub-trees in Fig. A.1(f) are now simple enough to be symbolically evaluated.
A.0.3 Memory Bank Conflict and Coalescing

In general, deriving the exact memory access pattern is infeasible, especially with the presence of relatively complex subscripting functions (Section A.0.2) and unknown upper bounds for the induction variables. However, we are interested in certain qualitative properties of the array subscript expressions. For example, for efficient utilization of memory bandwidth, successive words are assigned to successive memory banks. Let $\text{NUM}_{\text{bank}}$ be the number of memory banks, which is also equal to the number of threads that can simultaneously access memory. The problem of determining the pattern of memory bank conflicts can be transformed to the problem of computing $x \mod \text{NUM}_{\text{bank}}$, where $x$ belongs to a finite set of integers. Size of this set is proportional to the number of variables present in the subscripting expression. Let $v = [v_1, v_2, \cdots, v_n]$ be all $n$ participating variables in evaluation of expression $\mathcal{E}(v)$. Each $v_i$ can be formulated as $\alpha_i \text{NUM}_{\text{bank}} + \rho_i$, where $0 \leq \rho_i < \text{NUM}_{\text{bank}}$. Evaluating $\mathcal{E}(v)$ to capture memory bank conflict patterns is equivalent to evaluation of $\mathcal{E}(\rho)$, where $\rho = [\rho_1, \rho_2, \cdots, \rho_n]$. The difference is that members of variable vector $\rho$ are well-bounded.

The simplification engine is more stringent on subscripting expressions that are evaluated for memory coalescing. Not only should the terms involving local coordinates (thread ID) be separable from the terms that involve other thread coordinates and free variables, but each thread ID dependent term should be expressible as a sum of terms such as $At + B$, $\left\lfloor \frac{At + B}{2^{\alpha i + \beta}} \right\rfloor$ or $(At + B) \mod 2^{\alpha i + \beta}$, where $t$ is the local thread coordinate, $A, B \in \mathbb{Z}$ and $\alpha, \beta \in \mathbb{Z}$. As a result, the symbolic evaluation engine can identify through a limited number of steps, how many different memory segments are accessed by threads within a single memory vector instruction.
Figure A.1: Expression simplification for Example 1. (a) Initial expression tree. (b) Distributing integer division. (c) Distribution is carried on along the path that contains free variables; constraint $N = k \times 2^j$ is bound to free variable $N$. (d) Multiplication is distributed over the add operator. (e) Multiplication and integer division cancel each other out. (f) The simplified expression tree.
REFERENCES


