Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

School of Engineering and Sciences

# GPGPU Workload Characterization Using Memory Bottleneck Detection and Hierarchical Clustering Analysis

A thesis presented by

Luis Alberto Freire Bermudez

Submitted to the
School of Engineering and Sciences
in partial fulfillment of the requirements for the degree of

Master of Science

In

Electronics Engineering

Monterrey Nuevo León, December 3th, 2018

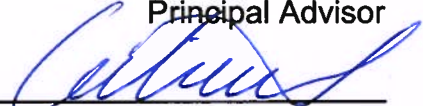
Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

School of Engineering and Sciences

The committee members, hereby, certify that have read the thesis presented by Luis Alberto Freire Bermudez and that it is fully adequate in scope and quality as a partial requirement for the degree of Master of Science in Electronics Engineering.

Dr. Alfonso Ávila
Tecnológico de Monterrey
School of Engineering and Sciences
Principal Advisor



Dr. Graciano Dieck Asaad
Tecnológico de Monterrey
Committee Member

Dr. Raúl Peña Ortega
Tecnológico de Monterrey
Committee Member

MSc. Luis Ricardo Salgado Garza
Tecnológico de Monterrey
Committee Member

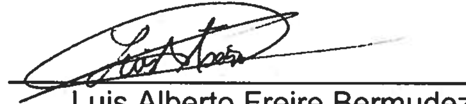
Dr. Rubén Morales Menéndez
Dean of Graduate Studies
School of Engineering and Sciences

Monterrey Nuevo León, December 3th, 2018

## Declaration of Authorship

I, Luis Alberto Freire Bermudez, declare that this thesis titled, “GPGPU Workload Characterization Using Memory Bottleneck Detection and Hierarchical Clustering Analysis” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.



Luis Alberto Freire Bermudez
Monterrey Nuevo León, December 3th, 2018

@2018 by Luis Alberto Freire Bermudez
All rights reserved

## Dedication

Este trabajo lo dedico primeramente a Dios, mis huellas en la arena son las suyas.

A mis padres, porque perdí la cuenta de las veces que me levantaron. Esto es de ustedes.

A mis hermanas, la vida es más bella cada vez que las recuerdo. Siempre las llevo en mi corazón.

A mi abuelita y a mis primos, que siempre están conmigo en la distancia.

A mis amigos, que siguen aquí a pesar del tiempo.

Finalmente, quiero dedicarle estas páginas de manera muy especial a Laura, Carlos, Leonardo, Alejandro, Colón, Sixter y Ana María; hasta que nos volvamos a ver.

## Acknowledgements

I express my gratitude to *Tecnológico de Monterrey* for awarding me with an important scholarship during my studies. Thanks to CONACyT for the financial support, which allowed me to continue my studies without any inopportuneness.

I would like to thank my advisor, Dr. Alfonso Ávila Ortega for his dedication towards this work and patience. I very much appreciate the hours of mentorship and caring, his guidance made this work possible.

In a very special way, I wish to express my deepest thanks to Professor Nikil Dutt at the *University of California Irvine*, for inviting me to be part of his research group during my stay at UCI and for his valuable contributions to this thesis. Also, I would like to thank Dr. Rosario Cammarota for his insightful comments to finish this work.

My deepest gratitude goes to the *Tecnológico de Monterrey* community, both the teachers and the students made of this an unforgettable experience.

**GPGPU Workload Characterization Using Memory Bottleneck Detection and Hierarchical Clustering Analysis**

by

Luis Alberto Freire Bermudez

**Abstract**

The use of Graphic Processing Units (GPU) for General Computing (GPGPU) has become increasingly common in recent years. In this type of processor, memory bottlenecks are a critical issue and the way data are commissioned to the partitions can cause several requests to get stalled behind each other, waiting for resources.

In this thesis, a methodology to characterize GPGPU kernels based on their likeability to create bottlenecks in the GPGPU memory hierarchy is presented. A GPGPU simulator is used to obtain unique fingerprints from more than 100 workloads and classify them using a Hierarchical Clustering Analysis.

The thesis also shows that that optimizations made to the kernels impact its run time memory bottleneck generation and that this behavior is successfully detected by the methodology. Two major groups of kernels were defined, naïve and optimized ones, and to characterize a set of exploration kernels within those groups with an effectiveness rate of over 75% for the two groups. A discussion is also held about how different levels of optimizations can be identified by our clustering engine and how those results could be use by subsequent approaches to predict bottleneck related issues in new kernels added to the cluster. Overall, a simple and transparent methodology to study bottleneck generation on GPGPU kernels is proposed which proves useful for future applications like static chararacterizer and statics predictor.

## List of Figures


Figure 2.1: Fermi Architecture Overview ..... 12
Figure 2.2: CUDA Hierarchy of threads, blocks and grids ..... 13
Figure 2.3: Euclidean Distance between to individuals ..... 15
Figure 2.4: Single-Linkage and Complete-Linkage distance measures ..... 16
Figure 2.5: Dendrogram in hierarchical Clustering ..... 17
Figure 3.1: Methodology Flowchart ..... 18
Figure 3.2: GPGPU-Sim Overall modeled architecture ..... 19
Figure 3.3: Memory Requests Stages as descripted by the simulator ..... 23
Figure 3.4: Data structure used for sampling stage ..... 24
Figure 3.5: Data acquisition Flowchart ..... 25
Figure 3.5: Data acquisition Flowchart (continued) ..... 26
Figure 3.6: Cycles spent in each stage by memory requests ..... 28
Figure 3.7: History of stalled requests as measured by one of the Queue Counters ..... 29
Figure 3.8: Total of memory requests commissioned to the partitions ..... 29
Figure 3.9: Matlab Queue Counter data processing for four cases ..... 31
Figure 3.10: Calculation of General Peakness Coefficients Vector ..... 32
Figure 3.11: Final workload signature ..... 33
Figure 4.1: First test dendrogram ..... 39
Figure 4.2: Radial dendrogram of Final Test ..... 41
Figure 4.3: Clustering for several values of k ..... 42
Figure 4.4: Elbow Method ..... 43
Figure 4.6: Gap Statistic Method ..... 44
Figure 4.5: Average Silhouette Method ..... 44
Figure 4.7: Final Radial Dendrogram with sub-clusters ..... 45
Figure G.1: Dendrogam using only SPC and mean ..... 96
Figure G.2: Dendrogam using only SPC and coefficient of variation ..... 97
Figure G.3: Dendrogam using SPC, coefficient of variation and standard deviation ..... 98
Figure G.4: Dendrogam using SPC mean and standard deviation ..... 99

## List of Tables

Table 2.1: Comparison between different approaches of Characterization and Modelling .........10
Table 2.1: Comparison between different approaches of Characterization and Modeling (Continued) .........11
Table 3.1: Configuration Options for simulated GTX480 Architecture .........20
Table 4.1: Platform Settings .........36
Table 4.2: Details of the exploration kernels .........37
Table 4.2: Details of the exploration kernels (Continued) .........38
Table 4.3: Distribution of the kernels on the clusters .........38
Table 4.3: Distribution of the kernels on the clusters .........46
Table A.1: Acronyms Definitions .........55
Table B.1: Variables Definitions .........56
Table C.1: Extended configuration options for simulated GTX480 Architecture .........57
Table C.1: Extended configuration options for simulated GTX480 Architecture(Continued) .........58
Table D.1: Extract of partition csv file produced .........59
Table D.1: Extract of partition csv file produced (Continued) .........60
Table D.2 Final File for Signature engine .........61
Table D.2 Final File for Signature engine (Continued) .........62

# Contents

## Contenido

Chapter 1 ........ 1
Introduction ........ 1
1.1 Motivation ........ 2
1.2 Research Questions ........ 3
1.3 Solution Overview ........ 3
1.4 Main Contributions ........ 4
1.5 Thesis Organization ........ 4
Chapter 2 ........ 5
Background ........ 5
2.1 State of the art ........ 5
A. Performance Models ........ 5
B. GPU simulators, emulators and compilers ........ 7
C. Workload Characterizers and Learning guided techniques ........ 7
2.2 Analysis of Areas of Opportunity ........ 8
2.2 Theoretical Background ........ 12
A. GPU ........ 12
B. Hierarchical Clustering Analysis ........ 14
Chapter 3 ........ 18
Methodology ........ 18
3.1 Obtaining Memory Queues Behavior History ........ 19
A. GPGPU-Sim Overview ........ 19
B. Extending GPGPU-Sim ........ 20
3.2 Obtaining Kernel signatures ........ 29
A. Overview ........ 29
B. Single Queue Peakness Coefficient ........ 30
C. General Stage-wise Peakness Vector ........ 32
D. The coefficient of Variation and Mean factors ........ 32
E. Final Kernel Signature ........ 33
3.3 Hierarchical Clustering Analysis ........ 33
Chapter 4 ........ 35

Results ..... 35
4.1 Introduction ..... 35
4.2 First Study ..... 35
A. Exploration Kernels ..... 36
C. Dendrogram ..... 37
4.3 Final Study ..... 40
A. Dendrogram ..... 40
B. Number of clusters ..... 40
C. Optimal Number of Clusters ..... 43
D. Final Radial Dendrogram ..... 45
4.4 Detecting Exploration kernels ..... 45
4.5 Discussion ..... 46
Chapter 5 ..... 48
Conclusion ..... 48
5.1 Summary ..... 48
5.2 Contributions ..... 48
5.3 Strengths and Limitations ..... 49
5.4 Future work ..... 50
Bibliography ..... 51
Appendix A ..... 55
Acronyms and Abbreviations Definitions ..... 55
Appendix B ..... 56
Variables and symbols ..... 56
Apendix C: Configuration Options Fermi Architecture ..... 57
Appendix D: CSV files produced ..... 59
Appendix E: Added and modified GPGPU-Sim code ..... 63
Appendix F: Matlab Signature Calculation Code ..... 94
Appendix G: Different Variable Selection ..... 96

# Chapter 1

# Introduction

Development of Complementary Metal Oxide Semiconductor (CMOS) technology has been the milestone for the success of information technology age. While inventiveness and imagination continue to drive society forward, processors have been struggling to keep up in the past decade. The difficulty of power dissipation has prevented RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) processors from reaching practical operating frequencies beyond the 5 GHZ. Miniaturization also presents problems as Moore correctly stated several years ago. Since applications such as deep learning, high accuracy simulations and medical research demand up to the task computing power, developers have turned to Graphics Processing Units (GPU) in order to handle the expected performance of a state-of-the-art application and prepare for future demands. General computing is has been taking advantage of parallel devices, this trend has come to rename Graphic Processors as GPGPUs (General Purpose Graphic Processing Units) when they are used for general applications [1].

Availability of massive Parallel Devices have proven useful in handling computations, but because it is a relatively novel technology, research in optimization is still scratching the surface of GPUs potential. On the other hand, differences between serial and parallel programming models alienates the technology from traditional

developers, efforts to overcome this includes environments like CUDA [2] and OpenCL [3] that aim to make parallel programing accessible and understandable.

While efforts for optimizing serial processing have proven to be useful and effective [4], [5], Parallel Processing still presents great challenges and opportunities mainly due to their complexity. In almost every system, research in throughput enhancements have demonstrated to be more effective for the processing units that for the memory, so the difference between the time it takes to extract data from storage and processing it remains one of the most problematic bottlenecks for developing applications [6], [7], [8]. This is still applicable to parallel computing, and even tough an operation can take advantage of a multicore setting, the way data is requested to the memory can make several requests to get stalled waiting for a certain resource in the hierarchy, causing a type of bottleneck that affects performance.

While both programmers and architects have clear opinions about the causes of this performance gap, finding and quantifying the real problems remains a topic for performance modeling tools [9]. To study these type of bottlenecks statistical and mathematical approaches have been proposed. Current techniques despite being useful and accurate, tend to depend either on complex modeling or intensive training. Characterization of this kind of performance degradation in GPGPU applications is scarce, which leaves room for improvement and proposition of new ideas.

## 1.1 Motivation

The frequency in which memory bottleneck caused by stalled requests occur, make them a critical issue when dealing with GPGPU applications optimization. Developers would take advantage of a tool that provides guidance for writing applications in a way that minimizes this type of bottleneck, so it keeps execution time as low as possible.

Although their consequences are evident after the execution, the nature of these performance pitfalls makes their study particularly hard. Memory bottleneck previously described occur at run-time and therefore virtually undetectable before and during execution. Current approaches consist of making sense of several performance counters and correlate them with know architecture characteristics in order to study the bottleneck effects and causes. Nowadays existing technologies allow us to simulate with accuracy the execution of GPGPU applications. These make possible a proper run time study of the memory behavior during the program execution without having to rely on offline metrics. These simulators can be used to precisely characterize specific bottleneck related issues and correlate them with performance pitfalls.This kind of methodology would make it easy to construct a fairly transparent and relatively simple GPGPU static performance predictors that take into consideration memory bottleneck prediction and correction. One part of the method should be able to detect and keep a record on how the program behaved during the entire execution and identify patterns related to know performance pitfalls. It also must be able to select appropriate places in the hierarchy to study and look for unwanted behavior. The methodology must be able to distinguish the patterns in one application among several other, without needing to run an excessive amount of training subjects. It is a goal of this work to construct not just an effective

methodology, but a transparent one that can be easily understood, applied and, if necessary, modified in the future.

In summary, the goal is to construct a methodology that successfully characterizes GPGPU applications based on their likelihood to cause a memory bottleneck at run-time. The methodology must work in a way that it can serve as a basis for constructing a static bottleneck predictor using the information about run-time pattern identified in the future.

## 1.2 Research Questions

Based on a brief *State-of-the-Art* review, the following hypothesis are proposed:

I) Run-Time memory bottlenecks in GPGPU applications can be detected using a GPGPU simulator

Ii) Geometric approximations are valid bottleneck detection methods

Iii) GPGPU applications can be characterized using a reduced application signature without losing accuracy

IV) Hierarchical Clustering Analysis can be used to classify applications and detect bottleneck related issues among different kernels

## 1.3 Solution Overview

Characterization of run-time metrics has been largely used to study and compare software behavior. In terms of serial processing, Cammarota et al. in 2011 gathered run-time metrics to prune a set of candidate systems for software, eliminating the ones that slowed down the computation [10]. They defined an application signature as a vector of Pearson correlation coefficients between resource stall counts and cpi. In this work, a minimum spanning tree was selected as the algorithm to classify kernels.

Another way to stablish similarities is to make a Hierarchical Clustering Analysis for characterizing GPU benchmarks, as demonstrated by Che et al. in 2010 [11]. Extracting some ideas from their work, and measuring run-time data like in [10], Gonzales et al. applied them to GPU computing [12]. They showed that the analysis generally distinguished between levels of optimization, but they didn't distinguish these kernels among other applications.

Benchmark characteristics can also be used to predict performance. Che and Skadron studied in 2013 the approach of gathering benchmark characteristics to predict the performance on an application of interest by mapping it to the workload space using nearest neighbors' techniques [13]. It is possible to observe the advantages of characterization and similarity techniques to study the performance of GPU applications. In this work, GPGPU applications will be characterized based on their likelihood to create memory bottlenecks and the classified using a Hierarchical Clustering Analysis to distinguish levels of optimization among other workloads.

## 1.4 Main Contributions

A method for extracting memory request behavior history from a kernel running on a GPGPU simulator using queues is proposed. Thi research effort is based upon previous research using a different kind of kernel signature that is still valid for prepresenting the memory bottleneck generation rate of the workload. Hierarchical clustering analysis is used to classify more than 120 kernels extracted from 49 GPGPU applications.
Exploration kernels, whose performance is generally known are used to test the characterization. The methodology shows that there is a tendency to identify different levels of optimization in different sub clusters of the hierarchy.

## 1.5 Thesis Organization

**Chapter 2** constitutes the background of our work. Firstly, the state of the art is presented including performance models and machine learning techniques for characterization. Additionally, theoretical notions like GPU Architecture and Clustering algorithms are visited so the unfamiliar reader could get used to these concepts. **Chapter 3** provides a detailed description of the three stages of our proposed methodology, which is the main contribution of this work. **Chapter 4** shows the results obtained by the proposed methodology. In addition, a discussion is presented looking for a validation of the methods. **Chapter 5** presents the conclusions, highlights the obtained contributions and introduces future work

# Chapter 2

# Background

## 2.1 State of the art

The study of performance and application characterization presented in this Chapter includes novel approaches in performance models for understating execution and predicting performance, Architecture simulators, compilers, and learning techniques.

### A. Performance Models

Performance has always been a concern when dealing with CPU or GPU applications since architectures are now more complex, their behavior is difficult to predict, but necessary in order to avoid testing applications by trial and error [1]. In terms of GPU, accelerated applications performance models still have several research areas such as complexity. In this section, approaches to propose more accurate or simple performance models for GPUs are described. One of the main factors preventing developers from writing optimal parallel applications is the enormous size of the design space and several approaches have been proposed to simplify design tasks using models. Mirsoleimani and Khunjusj proposed in 2015 a two-tier algorithm which builds a multiple linear regression model from a small set of simulated data instead of using the whole set of design point for their targeted architectures [14]. They were able to construct a GPU performance predictor which could predict the performance of any point in the design space with an average prediction error between 1% and 5% for different benchmark applications. Following that trend Tran and Lee proposed in 2015 a model for tuning the number of

Blocks and number of threads per block for a GPU kernel to a size that could reach optimal performance [15]. This approach has the potential to reduce design space and to prevent the programmers from performing this challenging task.

Bottlenecks in the hierarchy remain, in many cases, the principal reason for performance drawbacks, several efforts have been made to build models that represent and identify bottleneck behavior. In 2009, Georgia Institute of technology proposed a simple analytical model to estimate the number of parallel memory requests by considering the number of running threads and memory bandwidth and by doing so, estimating the execution time of massively parallel programs. Their goal was to provide insight into performance bottleneck of GPU applications to reduce design space for programmers [16]. Madougou et al. developed Black Forest in 2016, a statistical tool for predicting performance and analyzing bottlenecks in GPU applications based on random forest modeling and using hardware performance counters data [17]. Their method was tested using three different kernels as study cases and it is shown that it can predict execution time in a fairly accurate way and that bottlenecks are identified in each case.

With memory still being a critical design point for GPU applications, research in modeling has also leaned towards understanding the impact of memory usage when developing parallel accelerated programs. With the intention of providing insights into the performance of memory-intensive kernels, Hu, Liu and Hu proposed in 2014 a pipelined global memory that incorporates uncoalesced memory access pattern, as the author's state is the most critical global memory performance related factor, and provide a basis for predicting the performance of memory-intensive kernels [18]. Another model was proposed in 2015 by Konstantinis and Cotronis, which provides performance estimation of CUDA kernels on GPU hardware in an automated manner by looking at compute-memory bound to characterize kernels [19]. Using a set of kernels and device features they were able to get an absolute error in predictions of 10.1% in the average case. In terms of frequency, the way it impacts the performance was studied by Gupta et al., who presented a light-weight adaptive runtime performance model that predicts the frame processing time [20]. Their model was adaptative to different workloads, which made it suitable for online performance study. They later used this model to estimate the frame time sensitivity to the GPU frequency on mobile platforms. Benchmark characteristics can also be used to predict performance, like in [13] were the approach of gathering benchmark characteristics to predict the performance on an application of interest by mapping it to the workload space was studied. This work obtained a mean of 21.9% of prediction error.

Jiao et al. Presented in 2010 a power and performance characterization of GPU kernels revealing that energy saving mechanisms behave differently on GPU than on CPU and that this behavior can be detected by characterization [21]. Abe et al. in 2014 characterized and also modeled GPU-accelerated Systems across multiple generations of architectures [22]. Their approach took in consideration the impact of voltage and frequency in each platform and used characteristics from previous executions to provide a statistical model to predict performance in different generations of architectures. Nagasaka et al. presented in 2010 a statistical modeling approach for estimating power consumption using information from performance counters [23]. Since it is not always true that an operation could take advantage of a parallel architecture, Wen Wang and O'Boyle presented an efficient OpenCL task Scheduling scheme for determining at run-time which

kernels are more likely to take advantage of a parallel device [24]. In this research, static code structure metrics are combined with runtime features to build a predictive model.

### B. GPU simulators, emulators and compilers

In the past years, several simulators hardware models and emulators have been developed to obtain a testing platform for novel processing methods that could increase performance. These frameworks have helped to propose new methodologies that used collected data that would be nearly impossible to obtain from real hardware. In 2009, the University of British Columbia Characterized several non-graphics applications written in NVIDIA's CUDA programming model by running them on GPGPU-Sim, a novel detailed microarchitecture performance simulator that runs NVIDIA's parallel thread execution (PTX) virtual instruction set. [25]. Performance of different benchmarks was studied using GPGPU-Sim presenting performance analysis and bottleneck observations.

Kerr, Diamos and Yalamanchili used a full PTX (Parallel Thread eXecution architecture) emulator inside Ocelot: an open source infrastructure developed for the purpose of understanding data parallel applications, which can execute compiled kernels from the CUDA compiler [26]. The authors used this framework to characterize several kernels and applications. In 2010, Collange et al. presented Barra: A Parallel functional Simulator for GPGPU that generates detailed statistics on executions in GPU a simulated architecture [27]. Barra was developed inside the UNISIM environment and used cubin files as input. On the side of the compiler, efforts are focused in developing compilers capable to address the two major challenges of developing high performance GPGPU programs: effective utilization of GPU memory hierarchy and judicious management of parallelism. In the same year, Yang et al. developed a compiler that analyzes the code, identifies its memory access patterns and generates and optimized kernel obtaining promising results while maintaining the understandability of the resulting kernel [28].

### C. Workload Characterizers and Learning guided techniques

Another approach for understanding application characteristics or predict performance is Machine Learning, even though it may require intensive training they have the advantage of not requiring manual intervention to learn and detect patterns. This section explores previous research works that either characterize workloads or predicted performance using learning techniques.

Che et al. presented in 2010 a characterization of Rodinia Benchmark on an NVIDIA GeForce GTX480, comparing it to Parsec to search for similarities and differences using a Hierarchical Clustering Analysis [11]. Their metrics consisted of Instructions per Cycle (IPC), Memory Instruction mix, and warp divergence. According to the authors, important differences between benchmarks were observed. Effort from Goswami et al., at the University of Florida, focused on creating a characterization

methodology using microarchitecture agnostic characteristics to provide a meaningful way for architects to select a set of workloads appropriate to their research [29]. Their methodology was able to capture important behaviors such as coalescing and divergence characteristics.

Cammarota et al. Gathered run time metrics and used a similarity analysis to prune a set of target candidate systems from those that were more likely to reduce performance [10]. They defined an application signature as a vector of Pearson correlation coefficients between resource stall counts and cpi. To cluster the serial applications two algorithms based on minimum spanning tree were proposed in this research. Following this trend, research by Gonzales-Lugo et al. applied similar algorithms to GPU computing, selecting run time metrics and classifying kernels using hierarchical cluster analysis [12]. They showed that the analysis generally distinguished between levels of optimization, but they didn't distinguish these kernels among other applications. With the purpose of having a tool to estimate GPU performance for a piece of code before writing a GPU implementation, the University of Wisconsin-Madison proposed on 2015 Cross-Architecture Performance Prediction (XAPP) [30]. Their approach examined program properties and hardware characteristics to train a machine learning algorithm to recognize if the piece of code is worth being written for a GPU platform. Georgia Institute of Technology reported on 2010 an empirical evaluation of 25 CUDA applications on four GPUs and three CPUs [31]. A combination of different metrics for each application, gathered before kernel execution, was used to stablish relationships between program behavior and performance. A modeling framework is used then, in an attempt to predict the performance of similar applications on different processors.

Amaris et al. compared three different machine learning approaches: Linear regression, support vector machines and random forests with a BSP-based analytical model, to predict the execution time of GPU applications [32]. A Research group from The University of Texas at Austin in collaboration with AMD described a GPU performance and power estimation model that uses machine learning techniques [33]. This model was trained using measured performance and power data from several applications being run at different configurations. They combined hardware counters gathered from new kernels with the model to estimate performance and power with different GPU configurations. On the same way, IBM research department in 2014, presented an approach which applies supervised learning algorithms to predict the speedup obtained from porting a program to a GPU kernel. For GPU applications, the model can predict the best platform to run the kernel [34].

## 2.2 Analysis of research opportunity

GPU performance analysis remains a challenging task, and tool support is mandatory for the large adoption of systematic performance engineering [17]. Even tough great breakthroughs have been made in terms of mathematical models and machine learning algorithms, they both have several limitations like the extension of hardware knowledge

required or the lack of training data as was remarked by the work of the University of Wisconsin [30].

From the presented literature review, it can be inferred that a learning algorithm that performs a study of memory bottlenecks in GPGPU applications still presents areas of opportunity. One of the things that would benefit future approaches is the creation of a method for characterizing GPGPU programs that are transparent and therefore easy to understand. Looking towards prediction, the characterization must be able to work using a small training set and shall serve as a basis for identifying potential hazards at compiling time in the future. The review also presented hierarchical clustering analysis as a valid technique for characterizing applications, but unlike previous work, that identifies bottlenecks after execution based on offline performance metrics, our work focusses on dynamic characterization based of memory bottleneck generation, which can serve as a profiler to use in many future applications like bottleneck prediction. As stated by Nagasaka et al., characterization and modeling by performance counters measurements have limitations [23], for that reason, our approach combines Architecture simulators with learning techniques as an alternative to performance models. The use of a simulator allows us to gather metrics that would be virtually impossible to obtain from a real GPU, literature also backs up the decision of using Fermi as the target architecture since it demonstrates that it is still valid. Our approach is also the first one to propose a geometric approximation to characterize bottlenecks. The use of clustering allows us to stablish a hierarchy of similarity, rather than limiting to pointing out bottleneck issues.

Table 2.1: Comparison between different approaches of Characterization and Modelling

| Reference | Platform | Architecture | Study | Technique | Observation |
|---|---|---|---|---|---|
| Hong and Kim, 2009 [16] | GPU | Nvidia GTX-280 | Prediction | Analytical Model | Prediction based on the number of memory accesses |
| Bakhoda et al., 2009 [25] | GPGPU-Sim | Nvidia Geforce 8600GTS | Simulator | Does not apply | PTX instruction simulation |
| Kerr, Diamos and Yalamanchili [26] | GPU Ocelot | Nvidia GTX-280 | Characterization and emulation | Ocelot framework | PTX emulator |
| Che et al., 2010 [11] | GPGPU-Sim | Nvidia GTX-480 | Characterization | Hierarchical Clustering Analysis | Observe Benchmark Similarites |
| Yang et al., 2010 [28] | GPU | Nvidia GTX 8800 and Nvidia GTX280 | Optimization | Compiler development | Compiler Framework in Cetus |
| Kerr, Diamos and Yalamanchili, 2010 [31] | Ocelot | Nvidia GTX-280 | Prediction and Characterization | LLVM/Ocelot | Model for choosing between CPU and GPU |
| Jiao et al., 2010 [21] | GPU | Nvidia GTX-280 | Characterization | Profiling | Performance and power oriented |
| Goswami et al., 2010 [29] | GPGPU-Sim | AMD-ATI | Characterization | Hierarchical Clustering Analysis | Characterized on GPGPU-Sim and ran in ATI |
| Nagasaka et al., 2010 [23] | GPU | Nvidia GTX-285 | Prediction | Statistical modeling | Use of performance counters |
| Nagasaka et al., 2010 [27] | Barra | NVIDIA GeForce 9800 | Simulator | Does not apply | Nvidia Cubin File as input |
| Cammarota et al., 2011 [10] | CPU | Intel's Harpertown, Nehalem and Westmere | Prediction and Characterization | Minimum Spanning Tree | Correlation as the similarity metric |
| Che and Skadron, 2013 [13] | GPGPU-Sim | Tesla C2050 and a Kepler K20 | Prediction and Characterization | Nearest Neighbors | Characteristics obtained from GPGPU-Sim |
| Gonzalez-Lugo et al., 2013 [12] | GPU | Nvidia GT 335M | Characterization | Hierarchical Clustering Analysis | Minkowski distance as the similarity metric |
| Hu, Liu and Hu, 2014 [18] | GPU | NVIDIA Tesla M2050 | Prediction | Modeling | DRAM oriented |
| Abe et al., 2014 [22] | CPU/GPU | Several, including GTX-480. Intel Core i5 2400 | Prediction and Characterization | Characterization with statistical modeling | Performance prediction for different architecture generations |

Table 2.1: Comparison between different approaches of Characterization and Modeling (Continued)

| *Reference* | *Platform* | *Architecture* | *Study* | *Technique* | *Observation* |
|---|---|---|---|---|---|
| *Baldini, Fink and Altman, 2014* [34] | CPU/GPU | ATI FirePro v9800 Nvidia Tesla C2050. | Prediction | Binary Classification, K nearest neighbor | Predict GPU-porting speedup |
| *Wen, Wang and O'Boyle, 2014* [24] | CPU/GPU | Core i7/Nvidia GTX590 and Core i7/AMD Tahiti 7970 | Prediction and Characterization | Support Vector Machine Classifier | Dynamic Scheduling based on performance prediction |
| *Konstantinidis and Cotronis, 2015* [19] | GPU | Several, including GTX-480 | Prediction | Modeling | Kernel and device oriented. |
| *Mirsoleimani, Khunjush and Karami, 2015* [14] | GPU | Nvidia GTX-480 | Prediction | Two-tier design space exploration | Plackett–Burman and fractional Factorial designs are used |
| *Mirsoleimani, Khunjush and Karami, 2015* [30] | CPU/GPU | Maxwell and Kepler | Prediction | Cross-Arquitecture performance prediction | Program properties and hardware characteristics |
| *Wu et al., 2015* [33] | GPU | AMD Radeon HD 7970 | Prediction | K-means Clustering | Performance estimation using Machine Learning |
| *Amarís et al., 2016* [32] | GPU | Several | Prediction | linear regression, support vector machines and random forests | Machine Learning |
| *Gupta et al., 2016* [20] | Mobile GPU | Does not apply | Prediction and Characterization | Frame Time Characterization and Online Learning | Mobile platforms oriented |
| *Tran and Lee, 2016* [15] | GPU | Nvidia Tesla K20 and GTX285 | Tuning | Parameter Modeling | Selecting appropriate Block and Grid sizes |
| *S Madougou et al., 2016* [17] | GPU | Nvidia GTX-580 and K20m | Prediction | Black Forest | Use of performance counters |

## 2.2 Theoretical Background

### A. GPU

The background presented in this section is based on a review made upon NVIDIA Fermi architecture and NVIDIA Cuda Programming Model.

**Architecture Overview**

Fermi GPUs have 16 Streaming Processors(SM). Every streaming processor consists of computing units called CUDA cores (CC), this particular architecture has 32 cores per Streaming Processor for a total of 512 CUDA cores. The streaming processor is composed also by the instruction cache, warp schedulers, load and store units and a part Shared par L1 cache configurable Memory. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and Floating-Point Unit (FPU). Fermi GPU is also

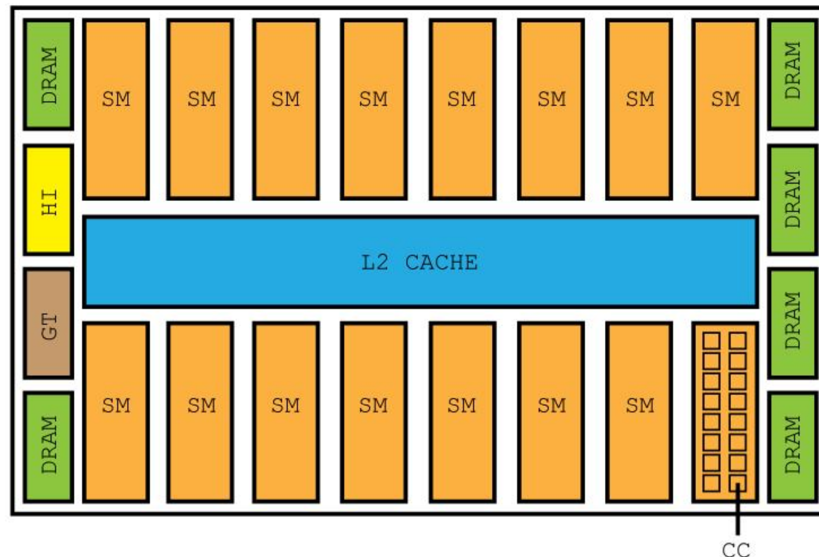



Figure 2.1: Fermi Architecture Overview [35], [36]

equipped with an L2 cache that covers GPU local DRAM as well as system memory. The architecture also features the GigaThread global scheduler, which distributes blocks to the streaming processors, the Host Interface and 6 different memory partitions that constitute memory level 3. Overview of Fermi architecture is presented by Figure 2.1

**The Programing Model**

CUDA is the hardware and software model that enables NVIDIA GPUs to execute programs written with C,C++,Fortran, OpenCL and other languages [35]. The computational elements of NVIDIA's CUDA software platform are known as *kernels,* that are pieces of code that run on a parallel device (in GPU context). An application may

consist of one or more kernels. The compiler transforms the kernels into many *threads* that execute the same operation. A thread can be seen as the iteration of a loop that works with, ideally, different data than the rest of them to help accelerate a computation. When an image needs processing, for example, the entire workload of pixels is decomposed so that each kernel can operate in only one pixel. Each thread has access to a local memory that is private for itself.

Several threads are grouped into *thread blocks* that can contain more than 1,500

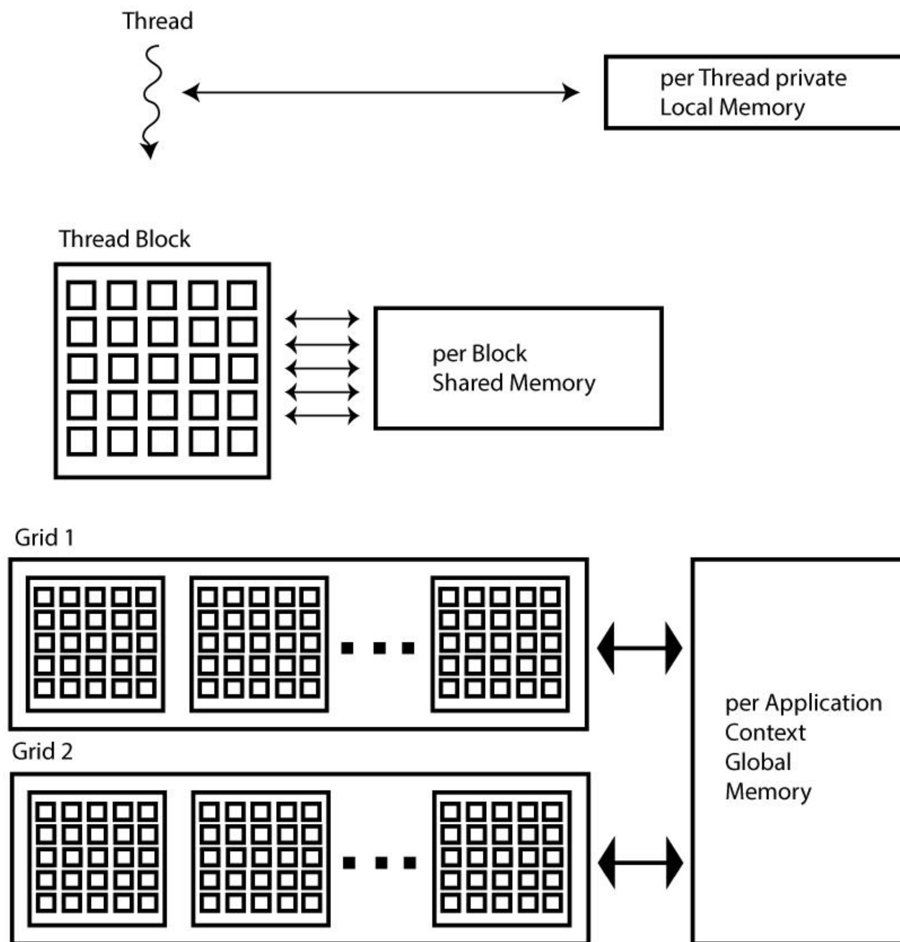



Figure 2.2: CUDA Hierarchy of threads, blocks and grids [35]

threads. Every thread in a block are run on a single *Streaming Multiprocessor* (SM), and within the SM all the thread can cooperate and share resources like the Shared Memory. The threads can execute in any order, in either concurrent or sequential fashion. Every thread has a thread id within its block. Thread blocks are divided into warps of 32 threads for execution. A warp is a fundamental unit of dispatch within a single SM [36]. Programmers can generally ignore warp execution since it is a hardware related aspect of the programming model, but the knowledge about its operation can improve performance.

Returning to the thread blocks, these blocks are grouped into *grids*, each of which executes a unique kernel, the grid can read inputs and write outputs to global memory. Thread blocks and grids each have identifiers that specify their position within the kernel

thread hierarchy. The GPU instantiates a kernel on a grid consisting of parallel thread blocks. Each thread within a thread block executes an instance of the kernel and has a thread ID within its block. Blocks have a Block ID within its grid. Figure 2.2 displays the hierarchy of threads, blocks and grids.

**The Modeled Architecture**

Overall GPU architecture modeled by GPGPU-Sim is showed in Figure 3.2. The GPU modeled by GPGPU-Sim is composed of Single Instruction Multiple Thread (SIMT) cores connected via an on-chip connection network to memory partitions that interface to graphics GDDR DRAM. SIMT cores are equivalent to Streaming Multiprocessors (SM), as referred to by NVIDIA or Compute Unit (CU) as referred by AMD. Several SIMT cores are grouped together under core clusters and share a common port to the interconnection network. Each SIMT core cluster has a single *response FIFO* that holds the packets delivered from the interconnecting network (ICNT) and distribute them to the different cores. Each core also simulated 4 different on-chip level 1 memories: shared memory and data cache, both with reading and writing capabilities, constant cache and texture cache with reading only capabilities. The interconnection network consists of a modified version of Bookism, a standalone network simulator as described in [37].

The memory system in GPGPU-Sim is modeled by a set of memory partitions. Each memory partition contains an L2 cache bank, a DRAM access scheduler and the off-chip DRAM channel. Various queues facilitate the flow of memory requests and responses between them. The memory request packets enter the Memory Partition from the interconnect via the ICNT->L2 queue. The L2 cache bank pops one request per L2 clock cycle from the ICNT->L2 queue for servicing. Any memory requests for the off-chip DRAM generated by the L2 are pushed into the L2->dram queue. If the L2 cache is disabled, packets are popped from the ICNT->L2 and pushed directly into the L2->DRAM queue, still at the L2 clock frequency. Fill requests returning from off-chip DRAM are popped from DRAM->L2 queue and consumed by the L2 cache bank. Read replies from the L2 to the SIMT core are pushed through the L2->ICNT queue. Requests exiting the L2->DRAM queue reside in the DRAM latency queue for a fixed number of SIMT core clock cycles. Each DRAM clock cycle, the DRAM channel can pop memory request from the DRAM latency queue to be serviced by off-chip DRAM and push one serviced memory request into the DRAM->L2 queue. More details about the simulator can be found in [38].

B. Hierarchical Clustering Analysis

Cluster Analysis is a group of techniques that intend to make groups of objects based on a set of characteristics selected by the user. The objective of the techniques is to form a cluster that exhibits high homogeneity within the clusters and high heterogeneity between the clusters. The primary objective of this type of analysis is to place the most similar individuals into the same groups. There are many ways to accomplish this task, and the configuration of procedures must be decided by the user. The two principal approaches that need to be selected are: a measure of similarity and a way to form

clusters. This section, based on [39], explores some of the approaches that are within the scope or were considered for this work.

### Similarity Measurement

The first step is to decide a way to determine similarity between to objects, this is an empirical measure of correspondence or resemblance. The two main types of similarity measures are Correlational Measures and Distance measures. Correlational Measures consists on calculating the correlation coefficient between the profiles of two objects, high correlations indicate similarity. These types of measures were not used in our approach.

Distance measures represent similarity as the proximity of observations to one another across the variables in the cluster, larger values denote lesser similarity. The most common similarity metrics used in cluster analysis are the following: Euclidean distance: Commonly referred to as straight-line distance is the length between two points in the space. If the space consists of two dimensions, the **Euclidean distance** between points $(x_1, y_1)$ and $(x_2, y_2)$ is the length of the hypothenuse of a right triangle, as stated in Figure 2.3. **Squared Euclidean distance**: Similar to the last metric, is the sum of the square differences without taking the square root. This similarity metrics speeds up computation. **Manhattan distance**: These metrics used the sum of the absolute differences of the variables as the way to determine similarities. The fastest metrics to compute but can lead to errors if there exists a high correlation between the variables. **Mahala Nobis distance**: A generalized distance measure that uses the correlation between variables giving equal weight to each of them. Although convenient in many cases, it is not yet available on many platforms.

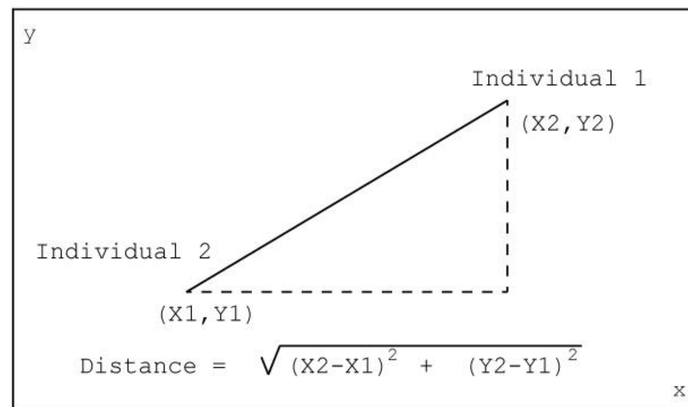



Figure 2.3: Euclidean Distance between to individuals [39]

### Clustering techniques

The two basic types of hierarchical clustering procedures are agglomerative methods and divisive method. Agglomerative methods, the ones used in our research, consists on considering every individual as a cluster and then it is successively joined to other clusters

until a single cluster is obtained. The clustering algorithm in a hierarchical procedure defines how similarity between to clusters will be determined. The five most popular agglomerative clustering algorithms are the following: **Single-Linkage**: Defines the similarity as the shortest distance from any object in one cluster to any other object in the other, as presented in. In clusters that are poorly delineated, single linkage can end up chaining clusters that are not similar because of the poor contrast between their outer elements. **Complete-Linkage**: Similar to Single-Linkage method, but it is based on the maximum distance instead. This approach, like the last one only represents one aspect of the data, and do not consider the rest of the elements of the cluster. Figure 2.4 presents Single and Complete linkage.

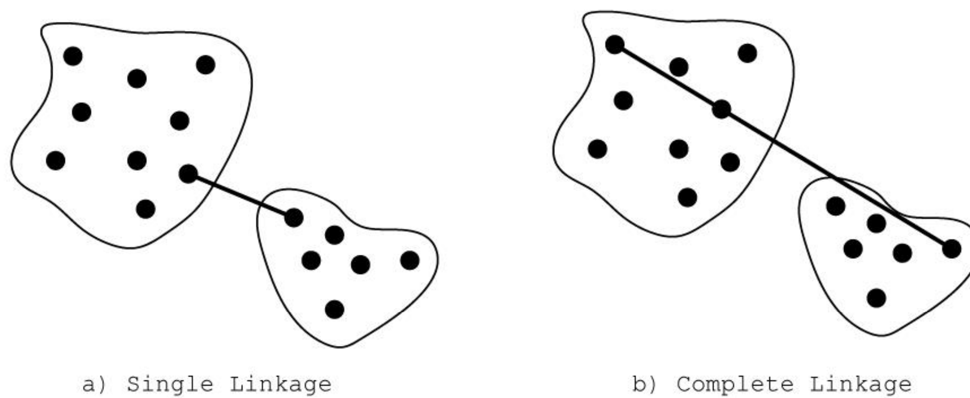



Figure 2.4: Single-Linkage and Complete-Linkage distance measures [39]

**Average Linkage**: In contrast to the last two methods, average linkage considers the similarity between any two clusters as the average similarity of all individuals in one cluster with all the individuals in another. This approach tends to generate clusters with small within-cluster variation. **Centroid Method**: In this method, the similarity metrics is the distance between two cluster centroids. Centroids are the mean between every individual in a cluster. Every time a new individual is added, the centroid is recalculated. This approach may produce confusing results. **Ward's Method**: In this procedure, the selection of which two clusters to combine is based on which combination of clusters minimizes the within-cluster sum of squares across the complete set of disjoint or separate clusters [39]. The clusters combined are the ones that minimize the increase in the total sum of squares across all clusters. Ward's method tends to produce clusters with the same number of observations.

After determining similarity between every cluster, a hierarchical representation where any individual can trace its membership to the rest of them through an unbroken path can be obtained, this representation is called dendrogram and is presented in Figure 2.5.

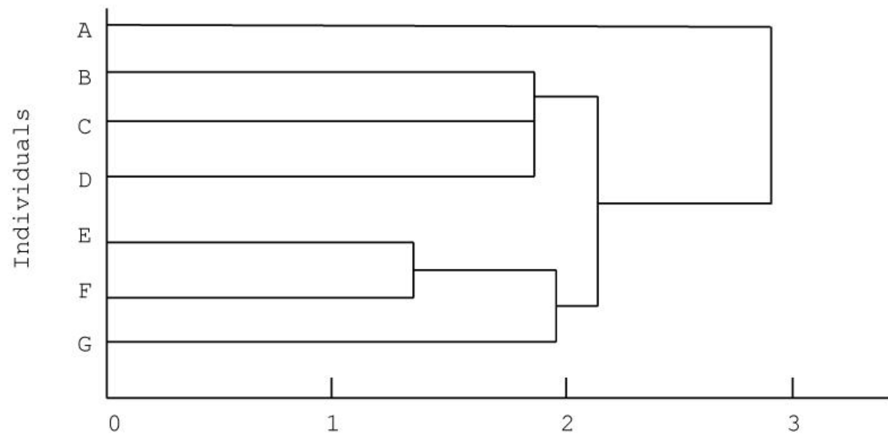



Figure 2.5: Dendrogram in hierarchical Clustering [39]

# Chapter 3

# Methodology

We build up on [12], proposing a novel method to characterize runtime bottleneck generation behavior in GPGPU applications using a GPGPU simulator. Our goal is to use this method to successfully identify kernels that share a similar level of performance degradation due to memory bottleneck generation among other kernels using a hierarchical clustering analysis.

Our approach consists of three main stages. (A) First, we obtain data for memory requests commissioned to the memory partitions throughout the CUDA kernel execution in a GPGPU simulator. (B) Then, we use that data to obtain a signature of the kernel. (C) Finally, we use the signatures obtained from all the kernels and use them to perform a hierarchical cluster analysis.

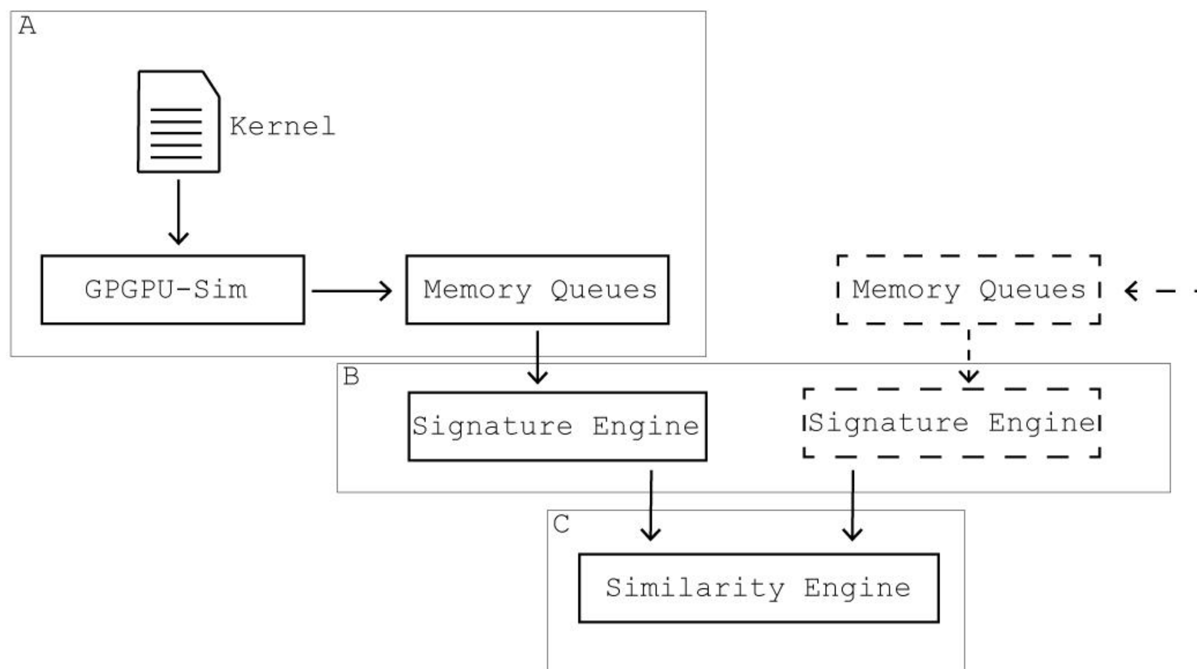



Figure 3.1: Methodology Flowchart

## 3.1 Obtaining Memory Queues Behavior History

A. GPGPU-Sim Overview

To obtain a history of how memory requests were stalled along the memory hierarchy during the kernel execution, a GPGPU simulator was used.

GPGPU-Sim was developed by Tor Aamodt along with his graduate students at the University of British Columbia. GPGPU-Sim provides a detailed simulation model of a contemporary GPU (such as NVIDIA's Fermi and GT200 architectures) running CUDA and/or OpenCL workloads. It Obtains IPC correlation of 98.3% (NVIDIA GT 200) and 97.3% (NVIDIA Fermi), [25]. The modeled architecture can be found in Figure 3.2 and was discussed in Chapter 2.2.

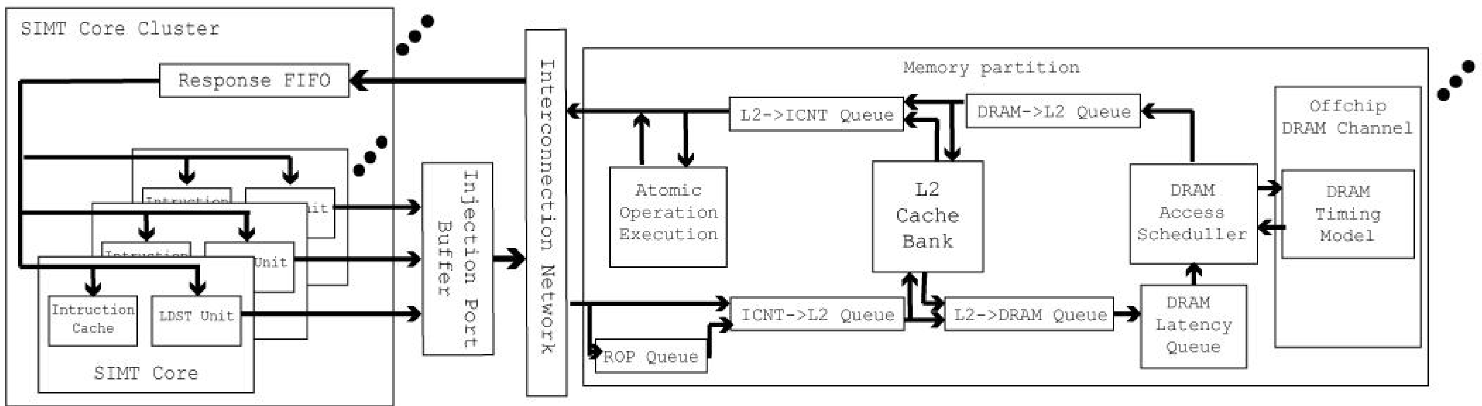



Figure 3.2: GPGPU-Sim Overall modeled architecture [38]

For this work, the selected architecture was GTX480 which has six memory partitions, simulated under GPGPU-Sim version 3.2.2. Some configuration options can be found in Table 3.1 while an extended list is shown in Apendix C. The simulator allow us to inpect and use memory hierarchy that is valid for different architectures, which would be usefull for future modifications and expantions. In terms of NVIDIA architectures, Fermi, Kepler and Maxwell differ in the constitution of internal caches, for example, but rely on similar levels of memory: Global, Shared and Local [40].

Table 3.1: Configuration Options for simulated GTX480 Architecture

| *Configuration Variable* | *Description* | *GTX480* |
|---|---|---|
| -gpgpu_n_mem <# memory controller> | Number of memory controllers (DRAM channels) in this configuration. | 6 |
| -gpgpu_n_clusters | Number of processing clusters | 15 |
| -gpgpu_n_cores_per_cluster | Number of SIMD cores per cluster | 1 |
| -gpgpu_num_sched_per_core | Number of warp schedulers per core | 2 |
| -gpgpu_shmem_size | Size of shared memory per SIMT core | 49152 |
| gpgpu_num_reg_banks | Number of register banks | 16 |

B. Extending GPGPU-Sim

Instrumentation was made using C++ to enhance GPGPU-Sim. Five Counters were placed in the marked stages of the memory hierarchy in each memory partition, (The simulated architecture divides the memory into 6 memory partitions, so there is a total of 30 counters). Figure 3.3.a denotes all the stages simulated by GPGPU-Sim for any memory requests, Table 3.2 provides details for the corresponding marks in the hierarchy. The flow of a request package is monitored using these different stages, so the simulator can keep track of its status. This feature was used to build a data structure that could detect changes in requests status and by doing so, maintain a history of the number of requests being stalled at every one of the stages throughout the entire kernel execution. Part of the instrumentation consisted in placing Queue Counters (QC) in each stage, that could keep a record of the number of instructions stalled in that point of the hierarchy.

To reduce the number of variables representing a particular partition, only selected Queue Counters were used in the final implementation. The final five QCs represent critical paths of the memory hierarchy [40], [41] and can be appreciated in Figure 3.3.b. The selected counters represent the partition requests behavior by capturing metrics like the use of the interconnection network, access to the L2 cache and access to Off-chip DRAM module.

Table 3.2 Stages names and ids, as presented by the simulator

| *id* | *Label* | *Stage* |
|---|---|---|
| 0 | a | IN_L1T_MISS_QUEUE |
| 1 | b | IN_ICNT_TO_MEM |
| 2 | c | IN_PARTITION_ROP_DELAY |
| 3 | d | IN_PARTITION_ICNT_TO_L2_QUEUE |
| 4 | - | IN_PARTITION_L2_TO_DRAM_QUEUE |
| 5 | e | IN_PARTITION_L2_MISS_QUEUE |
| 6 | f | IN_PARTITION_DRAM_LATENCY_QUEUE |
| 7 | g | IN_PARTITION_MC_INTERFACE_QUEUE |
| 8 | h | IN_PARTITION_MC_INPUT_QUEUE |
| 9 | i | IN_PARTITION_MC_BANK_ARB_QUEUE |
| 10 | j | IN_PARTITION_DRAM |
| 11 | k | IN_PARTITION_MC_RETURNQ |
| 12 | l | IN_PARTITION_DRAM_TO_L2_QUEUE |
| 13 | m | IN_PARTITION_L2_FILL_QUEUE |
| 14 | n | IN_PARTITION_L2_TO_ICNT_QUEUE |
| 15 | o | IN_ICNT_TO_SHADER |
| 16 | p | IN_CLUSTER_TO_SHADER_QUEUE |
| 17 | q | IN_SHADER_LDST_RESPONSE_FIFO |
| 18 | r | IN_SHADER_FETCHED |

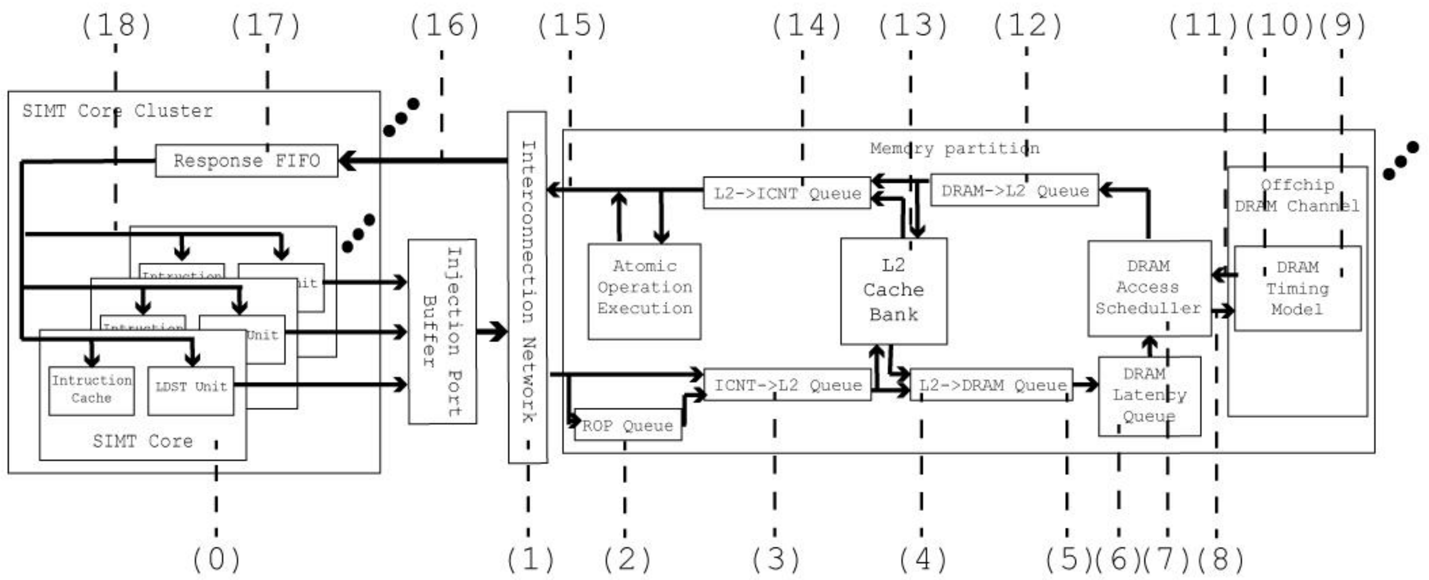
Selected final QCs monitor the memory partition during the execution of the application. They do so by keeping a record of how many requests are being stalled in a particular stage at every execution cycle. By the end of the execution, we obtain a history of occupancy at all the five selected stages in the hierarchy, which will help us to characterize the workload, details will be discussed in the following sections.

The simulator was instrumented to give three different performance metrics:

1. A track of the path each memory request follows through the memory partition, and the cycle in which each transition between stages happen.
2. A measure of how many instructions were stalled in each of the stages every cycle in which a request is moved.

3. A vector presenting how many requests were commissioned to each partition of the memory.

Instrumentation 1 was used for debugging purposes. Figure 3.4 and 3.5 present the data structure and workflow implemented in instrumentations 2 and 3. The data structure provided in Figure 3.4 display a configuration of linked lists used as a sampling tool for storing the status of many requests during the execution of the workload. One of the many challenges of instrumenting GPGPU-sim was the dynamic nature of the sampling, it was necessary for the engine to adjust the architecture characteristics like Number of Memory partitions or to different workload sizes. For that reason, linked list using dynamic memory were used instead of a fixed size array.



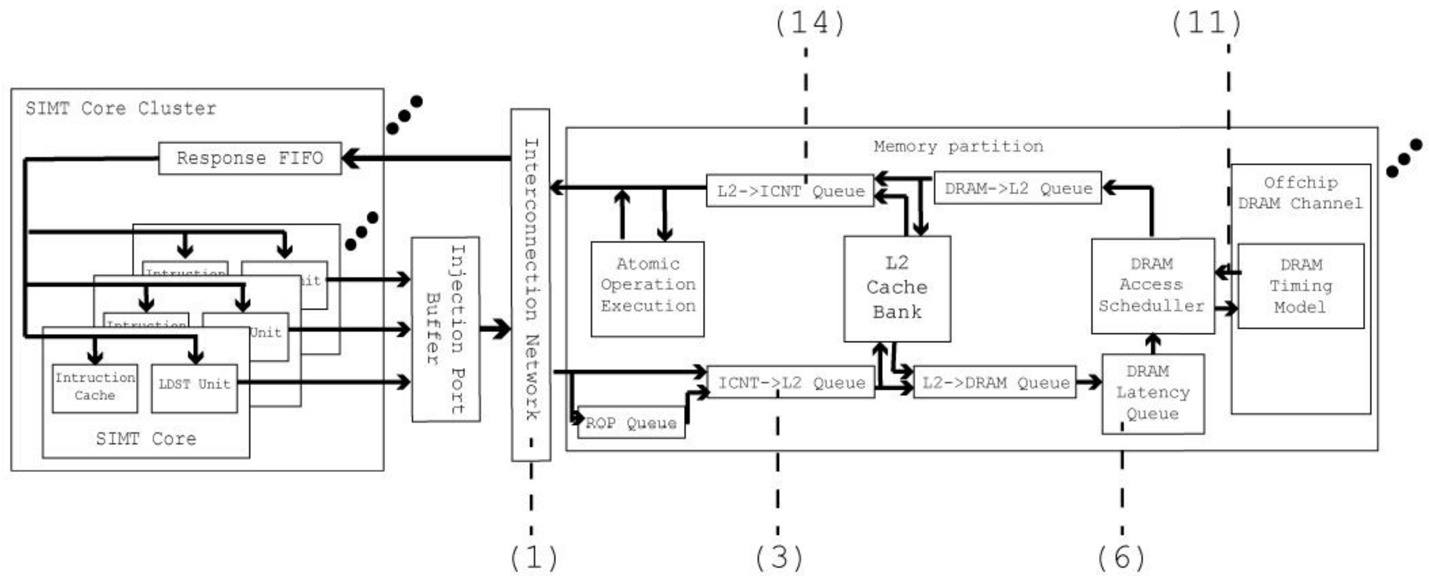




Figure 3.3: Memory Requests Stages as descripted by the simulator

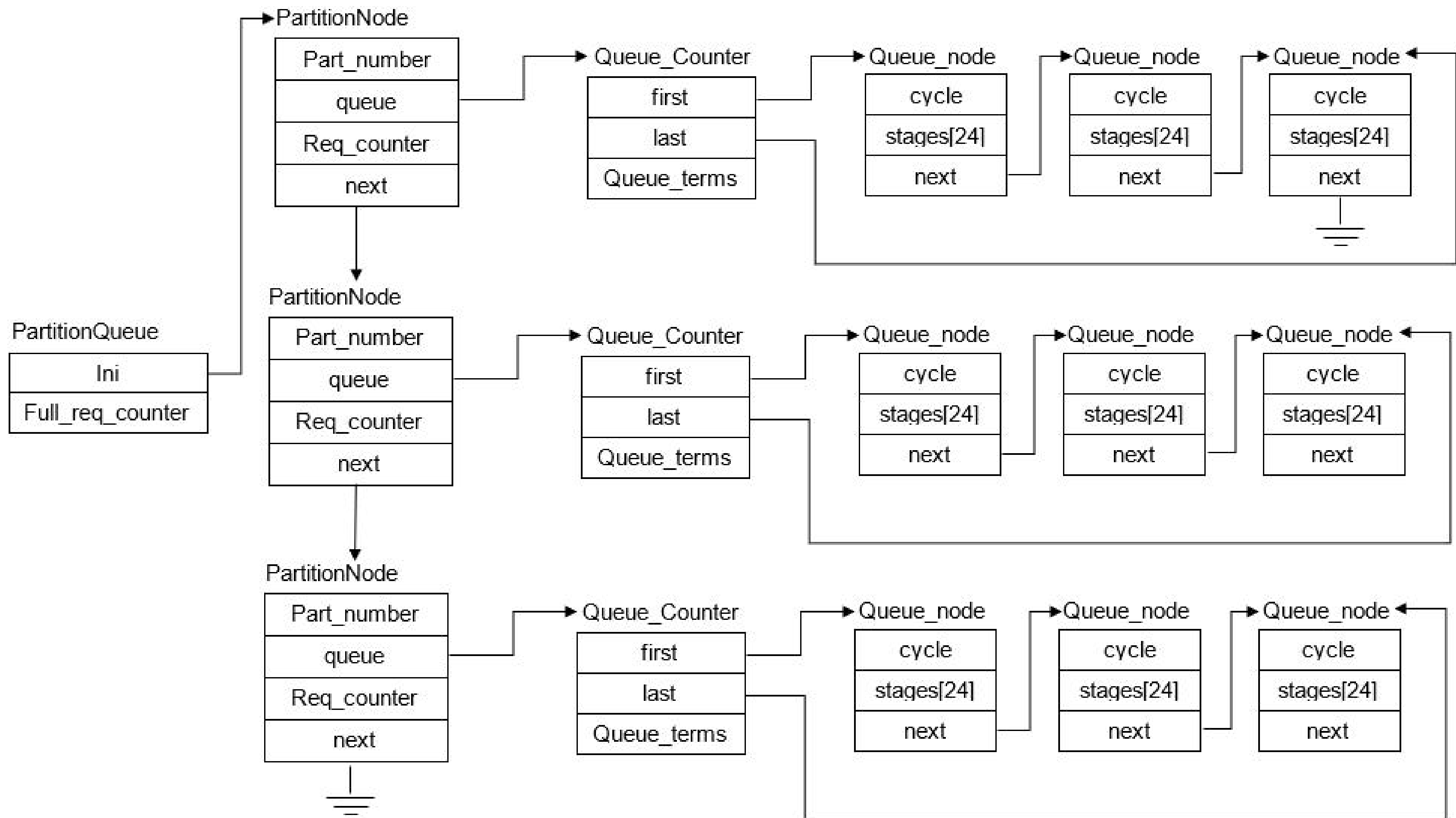



Figure 3.4: Data structure used for sampling stage

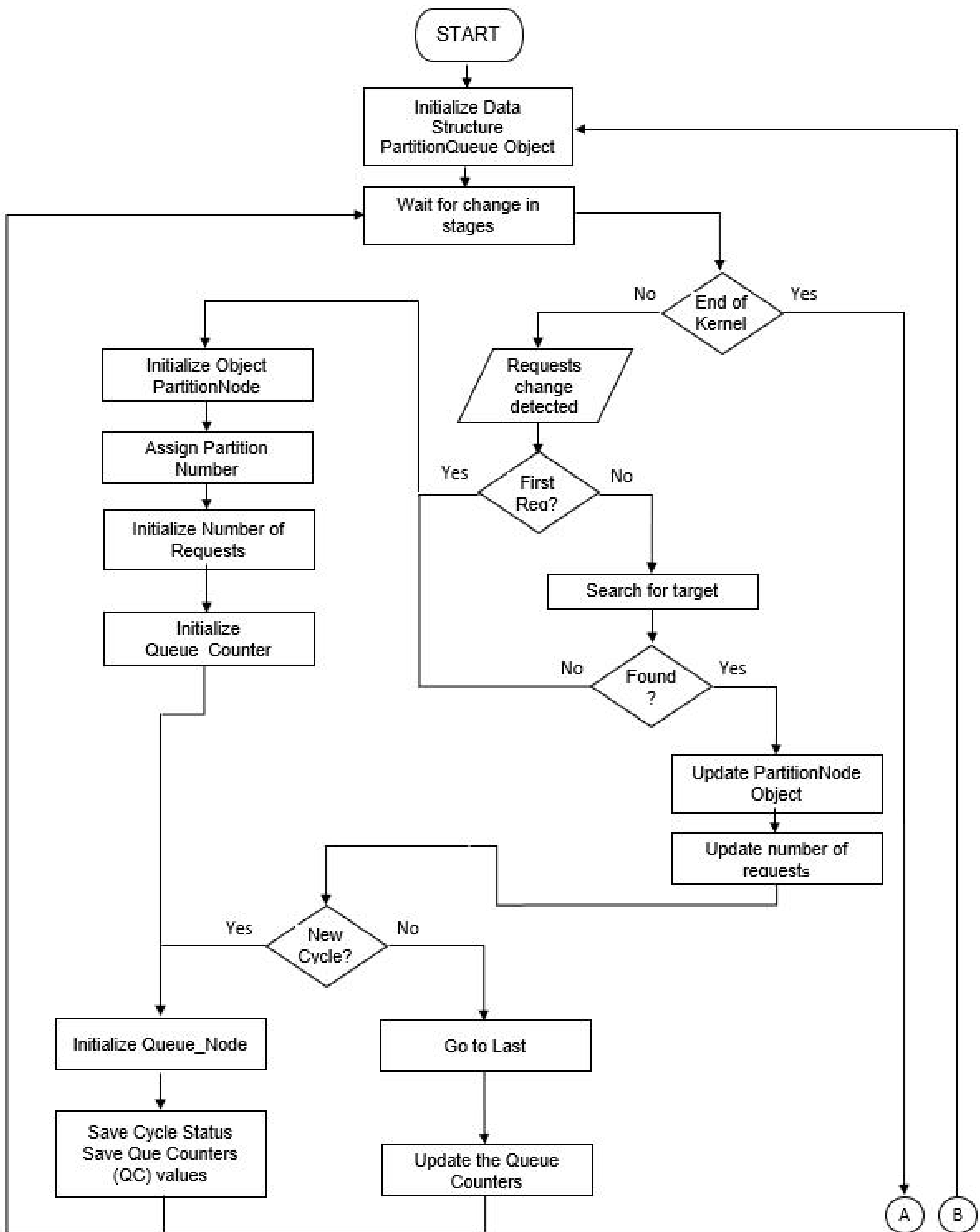



Figure 3.5: Data acquisition Flowchart

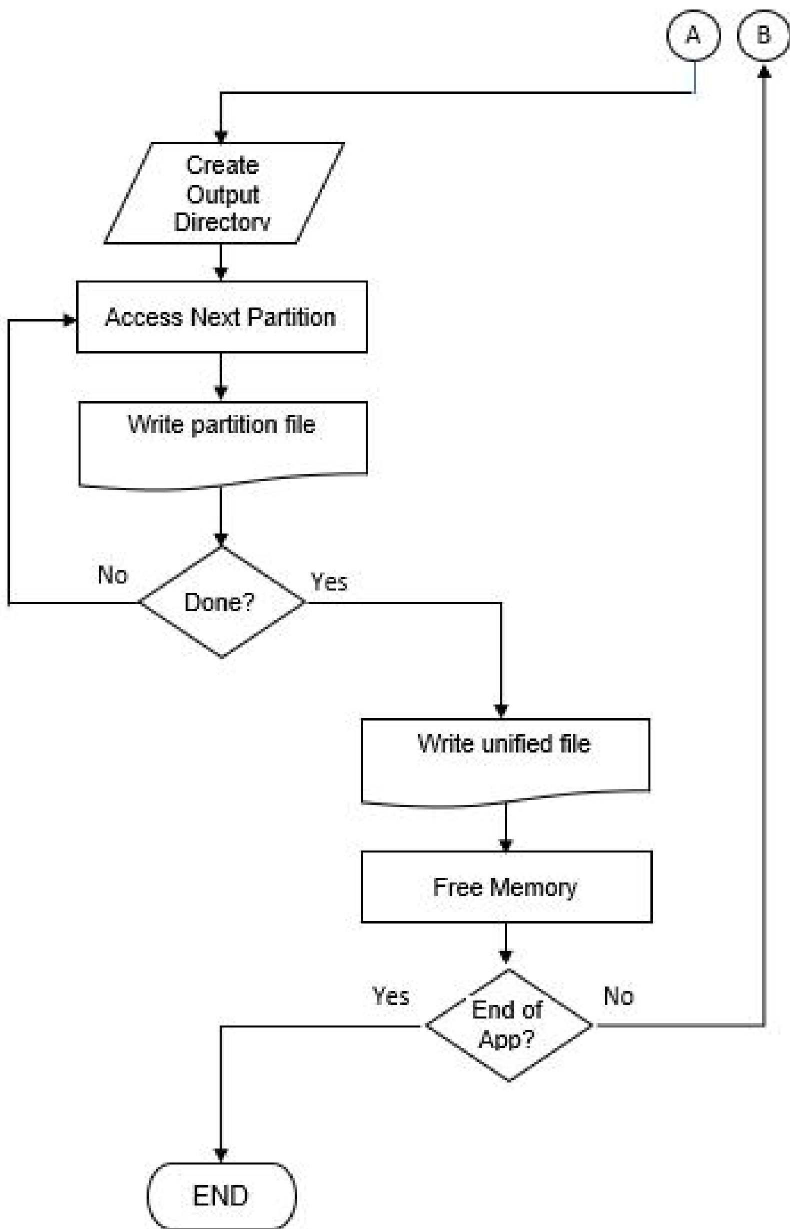



Figure 3.5: Data acquisition Flowchart (continued)

Class PartitionQueue holds the root of the structure, a pointer to the start of the list, and a variable that keep a record of the number of requests commissioned to each partition. An object of the class PartitionNode is created after receiving a request for a new memory partition, Figure 3.4 gives an example for 3 memory partitions. After a new

PartitionNode object is created it is appended to the last element of the list holding all the PartitionNode objects. Objects of this class also keep a variable indicating their corresponding partition number, a pointer to a Queue_Counter object (which is created separately to achieve abstraction), and the number of requests attended by the partition being held by it. The objects in this linked list are ordered in a FIFO fashion. Queue_Counter objects store pointers to the head and tail of a double lined list composed of Queue_node objects. Different queue metrics are also stored by objects of this class. Nodes in the double linked list are created upon arrival. When a new node is created, the last node on the list is accessed and their values modified according to the new information brought by the last requests at the time it stores the cycle in which the change occurred. In that way, the list keeps updating but maintains a history of behavior in previous cycles of execution.

This structure layout helps to maintain the possible number of partitions dynamic at the same time it keeps the capability of storing a large number of cycles updates, which was proved to be useful for large workloads. Figure 3.5 presents the sampling engine workflow. The previously discussed data structure is created at the beginning of the application execution, and the data collection is triggered by any change in a request stage detected by the simulator. After every request store, the engine will wait either for a new entry of for kernel termination. If the request, whose change was detected, is the first one or corresponds to a partition that has not received any request, yet the path of action is the following: a new PartitionNode object will be created and the partition number stored in the request packet will be mapped to this object. From that point on, that object will store information from any request commissioned to that partition number. Being the first request, a new object of the Class Queue_Counter will be created and after that, the information about the cycle and Queue Counters will be stored in a new Queue_node object under the Stages field. In previously created Queue_Counter object, first and last will point to the created first node in the list.

If the detected request is not the first one, the algorithm will then look for the PartitionNode object holding the linked list for the requested partition. If the PartitionNode object does not exist yet, the course of action is the same as in the previous case. In the case the request matches an existing PartitionNode holding their corresponding partition number linked list, an update course of action is followed. First, the Number of requests in PartitionNode object is updated, right after, the pointer to the last element of the list in Queue_Counter is extracted and there could be two scenarios: In the first one, the cycle in the requests packet matches the last node of the list, and in the other scenario it does not. When the cycles match, the last node is accessed and the values in its Queue Counters are updated so the values recorded in each cycle are only those that are final after that particular cycle of execution ended. On the other hand, when the cycles do no match a new Queue_node object is created, and it is appended to the tail of the list after extracting information from the previous tail. Using information from the previous list tail and combining it with the data brought by the requests packet, the new node calculates the last status and stores it in the Stages field with their corresponding cycle, becoming the new tail of the linked list.

Returning to the waiting step of the algorithm, if the kernel execution has stopped in the simulator the writing cycles of the workflow are initiated. In order to facilitate the manual tasks after sampling, all the kernel measured data in written into csv files in

comprised ways. Applications can run either one or several kernels, and for maintaining the order, data from different executed kernels must be kept separate from each other but related to their parent application. A new directory is created using the application name as a directory name, and inside it, a directory is created for storing the data of the specific kernel that was just executed. Using the data structure, PartitionNode list pointed by Ini are accessed in order and writing methods in the Class are executed beginning the writing of the csv file containing the history of Queue Counter values at every cycle of the execution, an example of this type of file can be found in Appendix D. At the same time the file is written, some information is extracted into variables that will later be written into a unified file, containing comprised information about all the memory partitions. The algorithms cycles through all the PartitionNode objects in the list writing their corresponding partition csv files. Right after this cycle, the comprised unified csv file, containing selected information from all partitions is also written in the directory, information about the number of requests commissioned to each partition is also included in this file. Finally, the memory used for storing data from the kernel execution is freed, keeping only the PartitionQueue object with an initialized Ini field. If the application has reached the end of the execution and there are no more kernels to be run, the algorithm finishes, otherwise the workflow just described begins again. The entire code for the instrumentations can be found in Appendix E

Figure 3.6 presents instrumentation 1. In this figure, the path taken by 3 different memory requests is presented. Bottlenecks in the memory hierarchy for the different instructions can be appreciated, as the graph states how many cycles were consumed in each stage. This instrumentation is useful for validating the performance of the sampling engine, but it is not used in the final version of the instrumented simulator. In Figure 3.7, the values on the X axis are the cycles of execution and the ones on the Y axis are requests stalled, this shows an example of the history of stalled requests in a stage QC of one of the memory partitions, being instrumentation 2. From this figure, the bottlenecks can be clearly appreciated, like the one between cycles 600 and 700. The simulator was modified to obtain in a concise fashion the number of requests that were commissioned to each one of the partitions (instrumentation 3), this will prove to be useful in the following section. Figure 3.8 compares measurements of total memory requests per partition obtained after 5 different kernel executions.

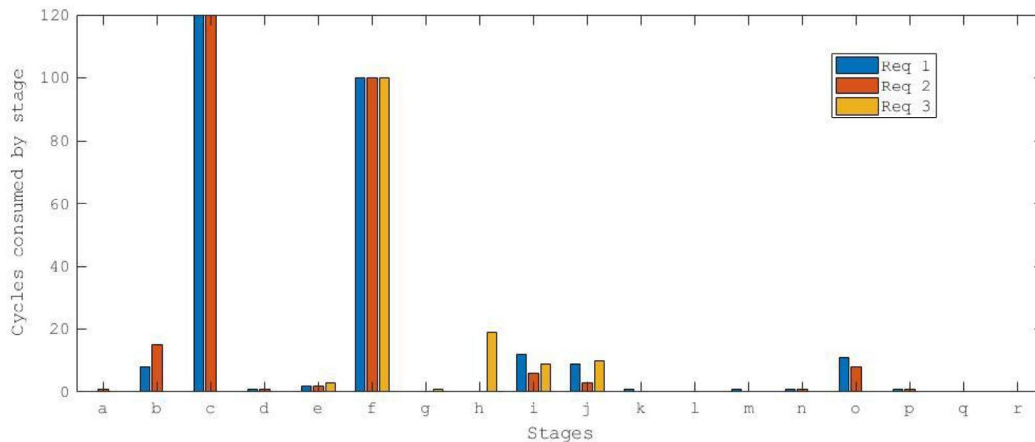



Figure 3.6: Cycles spent in each stage by memory requests

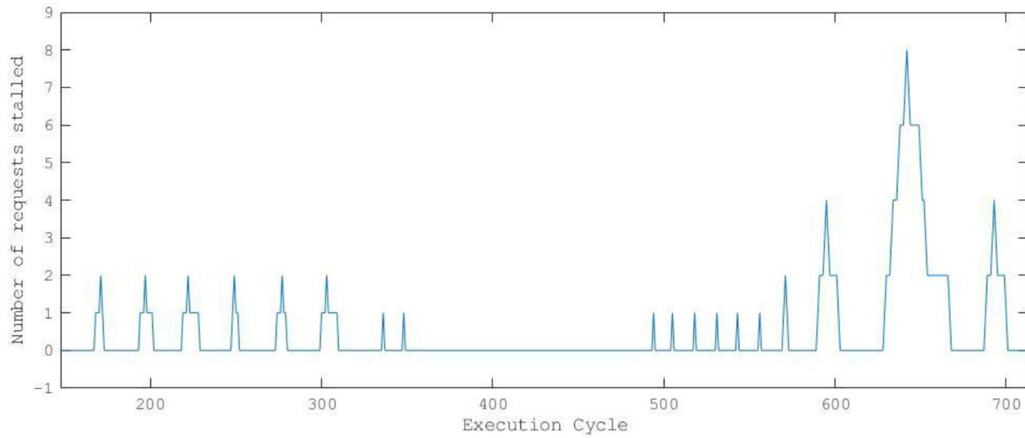



Figure 3.7: History of stalled requests as measured by one of the Queue Counters

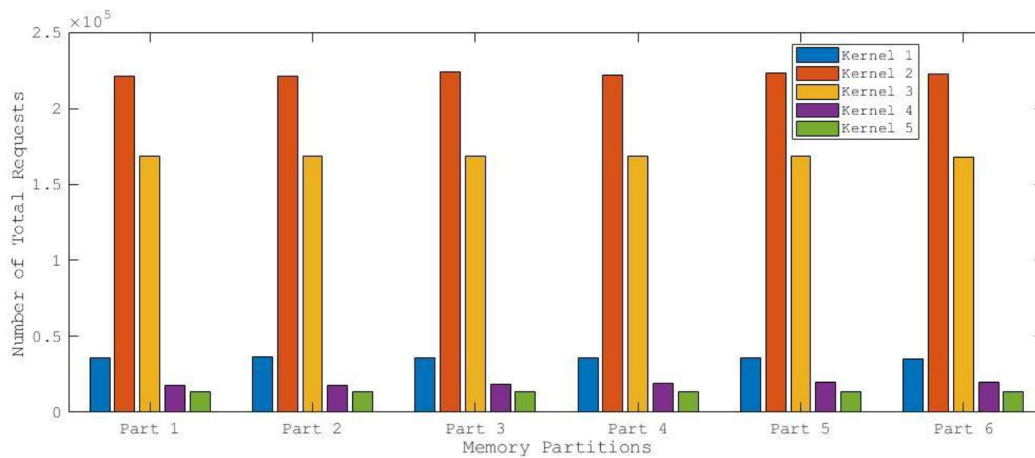



Figure 3.8: Total of memory requests commissioned to the partitions

## 3.2 Obtaining Kernel signatures

### A. Overview

A signature was implemented as in efforts by Cammarota et al, and Gonzales-Lugo et al. [10], [12], but unlike previous work, our signature consists on a vector of 7 elements that capture the behavior of the kernel in terms of bottleneck generation. This section explains the method for obtaining the signature.

B. Single Queue Peakness Coefficient

The first step translates the information obtained from Queue Counters, like the one in Figure 3.7, into a number that represents the “peakness” of the function. Algorithms like the ones proposed in [42] and [43] were considered but it was decided to come up with an approach of our own that would fit the demands of our research. Data coming from the unified csv file described earlier is loaded into Matlab R2018a version 9.4.0 function *findpeaks()* was used to identify peaks (pk) in the function, get their widths (w) and prominences (p), peak below a value of 2 is deprecated, Figure 3.9 presents 4 cases of these data processing. Using their widths and prominences, every identified peak was approximated to a triangle and its area calculated. Right after, every triangle area was escalated to the maximum peak measured among all kernels using a scale coefficient (sc). This had two advantages, first: We avoided taking two triangles of different shape but the same area as equals, and second: We were able to give more weight to greater bottlenecks at the time of making the hierarchical analysis. In (2) N stands for the total number of peaks identified.

$$sc = pk / pk_{\max} \tag{1}$$

$$spc = \sum_{i=0}^{N} sc_i \bullet \left[ \left( w_i \bullet p_i \right) / 2 \right] \tag{2}$$

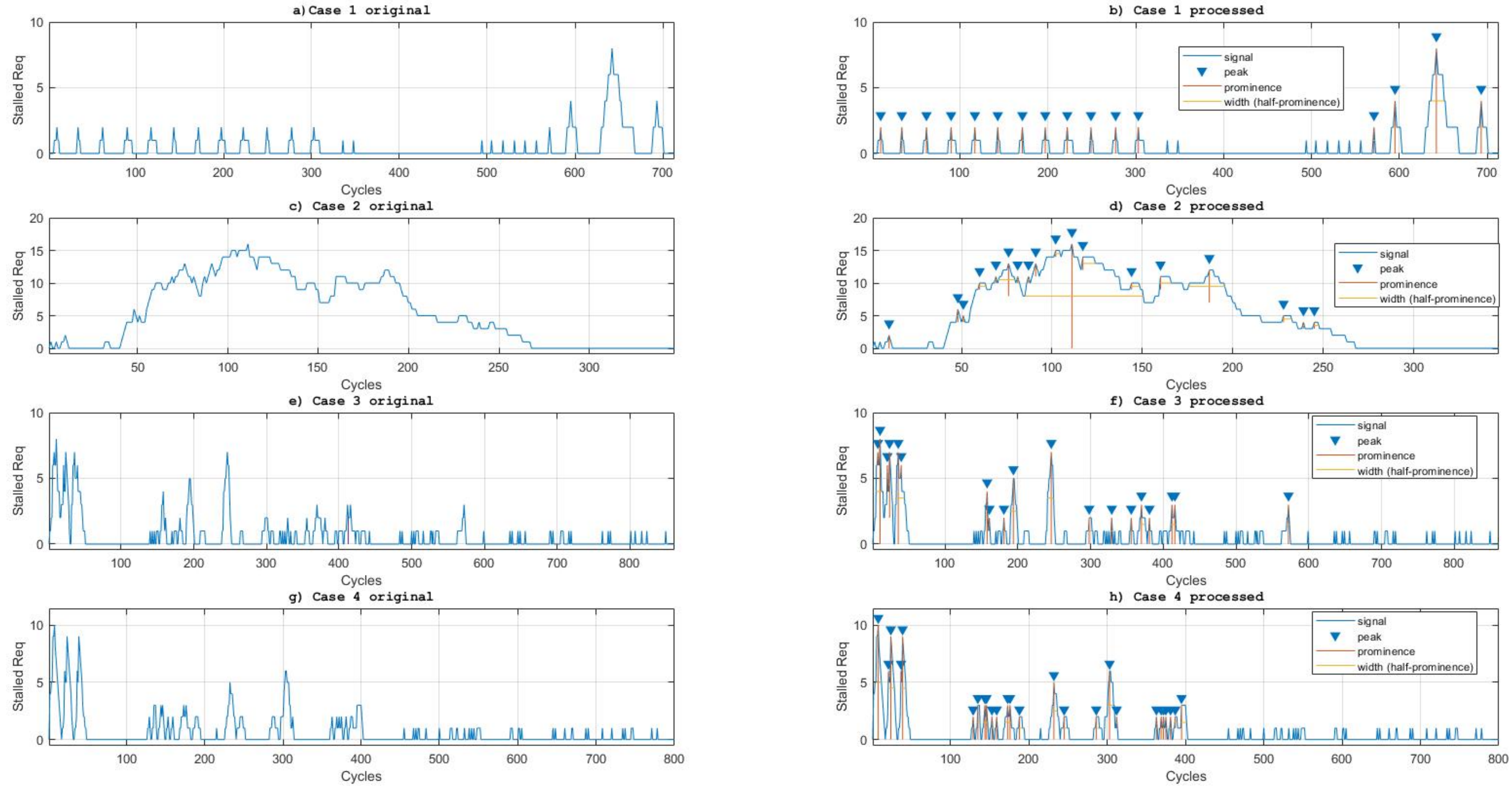



Figure 3.9: Matlab Queue Counter data processing for four cases

After every peak area has been approximated to a triangle and escalated to the maximum measured peak all the areas are summed, this is known as the Stage Peakness Coefficient (spc). Repeating this process for every one of the five Queue Counters in all six partitions gave us a total of 30 Stage Peakness Coefficients, as stated in Figure 3.10.

| | QC 1 | QC 2 | QC 3 | QC 4 | QC 5 |
|---|---|---|---|---|---|
| Partition 1 | $spc_{QC1}$ | $spc_{QC2}$ | $spc_{QC3}$ | $spc_{QC4}$ | $spc_{QC5}$ |
| Partition 2 | $spc_{QC1}$ | $spc_{QC2}$ | $spc_{QC3}$ | $spc_{QC4}$ | $spc_{QC5}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| Partition 6 | $spc_{QC1}$ | $spc_{QC2}$ | $spc_{QC3}$ | $spc_{QC4}$ | $spc_{QC5}$ |
| General Peakness Coefficients | $\sum_{i=1}^{6} spc^{i}_{QC1}$ | $\sum_{i=1}^{6} spc^{i}_{QC2}$ | $\sum_{i=1}^{6} spc^{i}_{QC3}$ | $\sum_{i=1}^{6} spc^{i}_{QC4}$ | $\sum_{i=1}^{6} spc^{i}_{QC5}$ |

Figure 3.10: Calculation of General Peakness Coefficients Vector

C. General Stage-wise Peakness Vector

Even though all 30 Coefficients are representative for the behavior of the kernel, section 3.3 will describe the use of Euclidean distance for measuring similarity between kernels, and previous research shows that it is counter-productive to use a great number of variables per object for this metric [44], [45], so we were forced to find a way to reduce dimensions. Being this the first version of our approach, we avoid the use of Principal Component Analysis (PCA) [46], in an attempt to maintain transparency. The Peakness Coefficients were summed together partition-wise, as shown in Figure 3.10, ending up with a vector of 5 General Peakness Coefficients that represent bottleneck related behavior in every stage and related to all the partitions.

D. The coefficient of Variation and Mean factors

To avoid losing a sense individuality between partitions by doing the previous step, we added a sixth variable to the signature. The coefficient of variation ($C_v$) is a standardized measure of dispersion defined as the ratio of the standard deviation (σ) to the mean (μ) [47]. This Coefficient of Variation measures how evenly distributed were the requests

commissioned to the partitions and becomes our sixth variable. The mean (μ) among the number of requests attended by the different partitions becomes our seventh dimension and measures the magnitude of requests made to the memory.

$$C_v = \sigma / \mu \tag{3}$$

E. Final Kernel Signature

Figure 3.11 details the overall constitution of the Final Workload Signature. Final signature consists of a vector of 7 elements, being the first 5 of them ,General Peakness Coefficients for stages one to five of the memory partitions, the sixth element being the coefficient of variation for the number memory requests commissioned to each one of the partitions, and finally the last element being the mean among the number of requests previously mentioned.

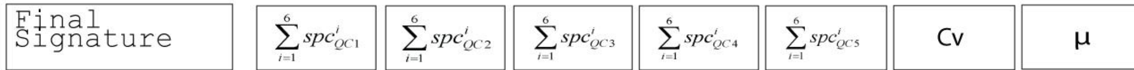



Figure 3.11: Final workload signature

Signatures are obtained from the unified csv file described in the previous section. Each vector is then written into another csv file that ends up containing all the signatures from the entire kernel pool executed by the end of the experiments. Code for obtaining the signature is displayed in Appendix F

## 3.3 Hierarchical Clustering Analysis

Hierarchical Clustering Analysis was made using R on Rstudio version 1.1.456. To perform cluster analysis in R, data from the signature file obtained in the previous section is loaded, rows are observations or individuals (kernels) and columns are variables (signature components). The data must be standardized (i.e., scaled) to make variables comparable. Recall that, standardization consists of transforming the variables such that they have mean zero and standard deviation one. We start by scaling the imported data using the R function *scale.* The next step is to perform the Agglomerative Hierarchical Clustering. First, we compute the dissimilarity values with function *dist* using Euclidean

distance, one on the distance metrics discussed in section 2.2, a similar distance metric was already used by [12] producing interesting results. After Euclidean distance matrix is calculated, the next step was to identify related clusters in a hierarchical fashion, these values are fed into *hclust* and Ward's method was used [48]. Resulting Hierarchy is presented in the next section using different sets of kernels and several visualization strategies for the resulting cluster including circular dendrograms.

# Chapter 4

# Results

## 4.1 Introduction

System specifications for experimentation are presented in Table 4.1. Kernel execution was simulated under Ubuntu Linux running over a virtual machine in VirtualBox 5.2.12.

This section presents the results obtained from clustering an initial set of 78 kernels on a dendrogram and then, the outcomes from applying the methodology to an extended final set of 128 kernels. From the final pool of kernels, several approaches and visualization techniques are studied.

The Final Hierarchical Cluster is presented in this section using a radial dendrogram, and the optimal number of sub-clusters in the hierarchy is determined by different methods.

The use of exploration kernels to validate the methodology is covered and details are provided on their source, nature, and expected behavior. Finally, the results obtained from the final cluster are discussed in an attempt to validate the accuracy of the methodology in identifying levels of optimization within a set of workloads.

## 4.2 First Study

The first study consisted of calculating signatures from 78 kernels obtained from 32 applications. These applications were taken from NVIDIA CUDA SDK version 4.0.17 [49] Ruetsch document [50] and Yang's et al. research [51]. For the first version of the methodology the complete signature was not considered since the signature engine was not fully developed at the time of this test, instead, a reduced signature consisting of only the General Peakness Coefficient vector was implemented. Euclidean distance was selected as the similarity metric and Ward's method used as the agglomeration policy.

A. Exploration Kernels

In order to test the methodology, selected kernels were introduced in the kernel pool. Table 4.2 presents the set of 24 kernels whose performance is generally known, we named them Exploration Kernels. Based on the description obtained from their applications we were able to classify these kernels in the categories relating to their level of optimization, note that Optimization 2 is certainly different from Optimization 1 but not necessarily better. Optimizations 1 and 2 kernels are expected to perform better than Naïve kernels. The location of these exploration kernels in the hierarchy help us test the effectiveness of our proposed methodology.

Table 4.1: Platform Settings

| *Virtual Platform* | |
|---|---|
| System | Oracle VirtualBox Graphical User interface version 5.2.12 r122591 |
| OS | Ubuntu 14.04 LTS |
| OS type | 64-bit |
| Processor | Intel® Core™ i7-6700HQ CPU @ 2.60GHz |
| Memory | 9.8 GiB |
| Graphics | Gallium 0.4 on llvmpipe (LLVM 3.8, 256 bits) |
| Disk | 32.5 GB |
| | |
| *Native Platform* | |
| System | ASUS-notebookSKU |
| OS | Microsoft Windows 10 Home 10.0.17134 Build 17134 |
| OS type | 64-bit |
| Processor | Intel® Core™ i7-6700HQ CPU @ 2.60GHz 4 Core(s) 8 Logical Processor(s) |
| Memory | 15.9 GB |
| Graphics | Intel(R) HD Graphics 530 128 MB of Display Memory |
| Graphics | NVIDIA GeForce GTX 960M 2010 MB of Display Memory |
| Disk | 881 GB |

## C. Dendrogram

Resulting dendrogram is presented by Figure 4.1, Y axis represents the height of the hierarchy which gives a sense of similarity between the individuals, X axis stands for the individuals (kernels). In this type of representation, as was explained in section 2.2, horizontal distance does not denote similarity, instead, we must look at the height of the branch connecting two kernels to investigate their relationship. For example, kernels 34 and 38 exhibits more similarity than 17 and 60, in the same way, those two sub-clusters are very dissimilar from one another. Taking another example, kernels in sub-cluster containing 66 and 73 are more similar to sub cluster containing 17 and 60 than to the one containing 34 and 38.

Table 4.2: Details of the exploration kernels

| *Optimization Level* | *Kernel Number* | *Operation* | *Application* | *Type of optimization* |
|---|---|---|---|---|
| Naive | 27 | Copy | Transpose 1 | None |
| Naïve | 29 | Transpose | Transpose 1 | None |
| Naïve | 60 | Demosaic | Demosaic | None |
| Naïve | 63 | Transpose Matrix Vector multiplication (Tmv) | Tmv | None |
| Naïve | 66 | Transpose | Transpose 2 | None |
| Naïve | 68 | Vector-vector mult (Vv) | Vv | None |
| Naïve | 71 | Copy | Transpose 3 | None |
| Naïve | 73 | Transpose | Transpose 3 | None |
| Optimized 1 | 28 | Copy | Transpose 1 | Shared Memory on 27 |
| Optimized 1 | 30 | Transpose | Transpose 1 | Coalesced Access on 29 |
| Optimized 1 | 31 | Transpose | Transpose 1 | Fewer Bank Conflicts on 29 |
| Optimized 1 | 61 | Demosaic | Demosaic | Coalesced Access on 60 |
| Optimized 1 | 64 | Transpose Matrix Vector multiplication (Tmv) | Tmv | Coalesced Access on 63 |
| Optimized 1 | 69 | Vector-vector mult (Vv) | Vv | Coalesced Access on 68 |
| Optimized 1 | 72 | Copy | Transpose 3 | Shared Memory on 71 |
| Optimized 1 | 75 | Transpose | Transpose 3 | Coalesced Access on 73 |
| Optimized 1 | 77 | Transpose Coarse Grain | Transpose 3 | Writing part of 78 |
| Optimized 1 | 78 | Transpose | Transpose 3 | Bank Conflicts free on 75 |

Table 4.2: Details of the exploration kernels (Continued)

| *Optimization Level* | *Kernel Number* | *Operation* | *Application* | *Type of optimization* |
|---|---|---|---|---|
| Optimized 2 | 62 | Demosaic | Demosaic | Prefetch, Partition Camping free 61 |
| Optimized 2 | 65 | Transpose Matrix Vector multiplication (Tmv) | Tmv | Prefetch, Partition Camping free 64 |
| Optimized 2 | 67 | Transpose | Tranpose 2 | Prefetch, Partition Camping free 66 |
| Optimized 2 | 70 | Vector-vector mult (Vv) | Vv | Prefetch, Partition Camping free 69 |
| Optimized 2 | 74 | Transpose | Transpose 3 | Diagonalization on 78 |
| Optimized 2 | 76 | Transpose Fine Grain | Transpose 3 | Computing part of 78 |

Preliminary results depict that the methodology shows a tendency to classify the majority of the exploration kernels into separate clusters, as intended. If we cut the dendrogram around half of the total height (proposing an arbitrary cutting height), we end up with three sub-clusters. The distribution of the exploration kernels between the different sub-clusters showed promising results, as presented in Table 4.3, and encouraged us to expand the experiments to continue the testing. The methodology struggles to differentiate between different types of optimizations. This could be a result of summing the Stage Peakness Coefficient vectors stage wise, losing the sense of how evenly distributed the requests were commissioned to the different partitions (Some optimizations tackle partition camping). For the next study case, the goals were to increment the size of the kernel pool, add the extra two dimensions to the signature and to implement partitioning methods to decide the optimal number of sub-clusters to extract from the hierarchy.

Table 4.3: Distribution of the kernels on the clusters

| *Cluster* | *Naive* | *Optimized* |
|---|---|---|
| 1 | 0(0%) | 0(0%) |
| 2 | 2(12.5%) | 14(87.5%) |
| 3 | 6(75%) | 2(25%) |

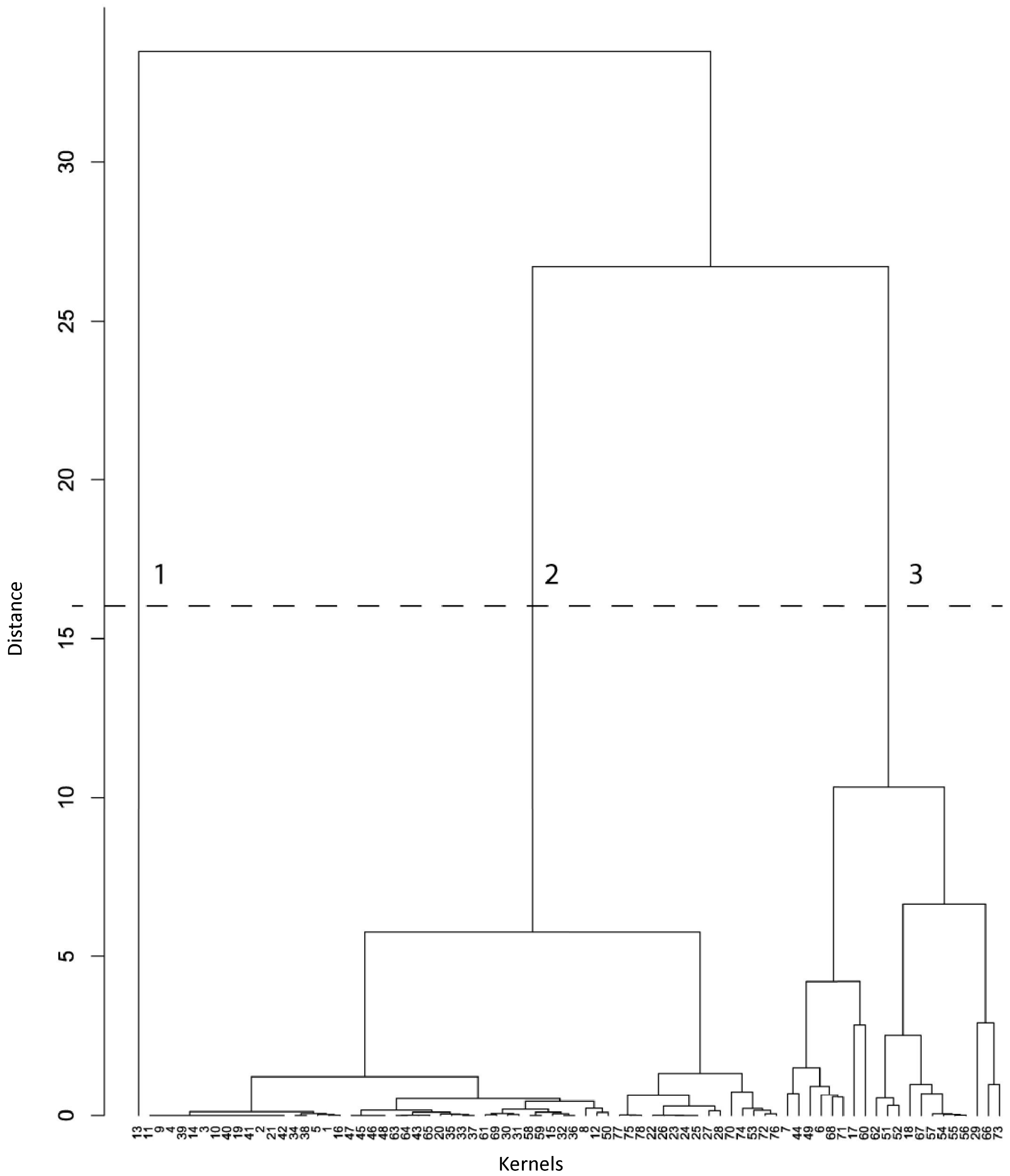



Figure 4.1: First test dendrogram

## 4.3 Final Study

To evaluate our proposed methodology the Hierarchical Clustering Analysis was made using 128 kernels obtained from 49 applications. These applications were taken from NVIDIA CUDA SDK version 4.0.17 [49], Rodinia 3.1 [52], Ruetsch document [50] and Yang's et al research [51].

For this last Clustering approach, the complete signature of seven elements is considered for each of the 128 kernels in an attempt to improve the accuracy and maintain at least reasonable congruency with the previous test despite having a greater number of kernels. Complete Kernel list with their corresponding signatures can be found in Appendix G. Euclidean distance was selected as the similarity metric and Ward's method used as the agglomeration policy. Different selection of kernel dimensions that were also tested and are listed in Appendix G.

### A. Dendrogram

For this second test, it was decided to use a radial representation of the Hierarchical Cluster due to the high number of individuals (kernels) that were being classified. Every one of the 128 kernels is represented by a number, the hierarchy is determined by the height of the branches connecting the kernels. To stablish the similarity between two kernels, we must investigate how close to the center we need to get to connect them, the closer to the center we need to get, the more dissimilar those kernels are. This type of analysis, like the first dendrogram, allows us to visualize hierarchical relationships and similarities, not evident before the experimentation, that were detected by the proposed methodology. The radial dendrogram is presented in Figure 4.2, where the 128 kernels are represented in a hierarchy that stablishes relationships based on the similarity between signatures obtained by the proposed methodology. Despite having a greater number of kernels than the previous test, this way of visualizing the dendrogram makes it easy to detect a relationship within the hierarchy

### B. Number of clusters

Dendrogram represented in figure 4.2 can be separated into several sub-clusters, these separations should be representative of relationships we are looking for in the hierarchy. As we separate the dendrogram into k different sub groups their start splitting after incrementing the k value. Figure 4.3 displays four selections of k value for cutting the dendrogram into a different number of sub-clusters. The higher the value of k, the stronger the relationship is among kernels of the same sub-cluster, but if the value is too high, the sense of hierarchy is lost and with it, the ability to obtain useful conclusions from the separation. A value must be chosen so it could represent the best selection of sub-cluster numbers, methods for acquiring this optimal number of k will be revised in the next section.

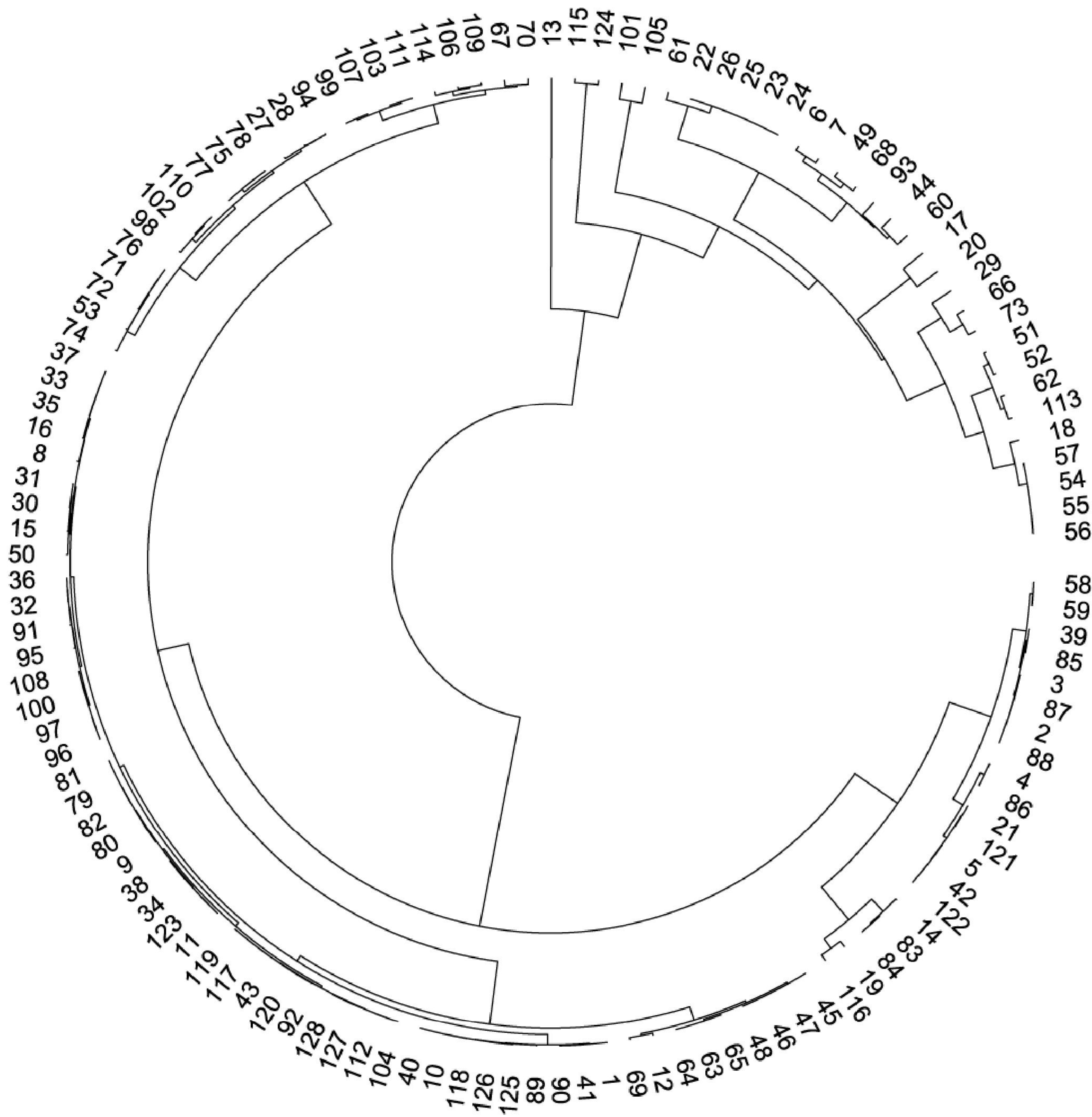
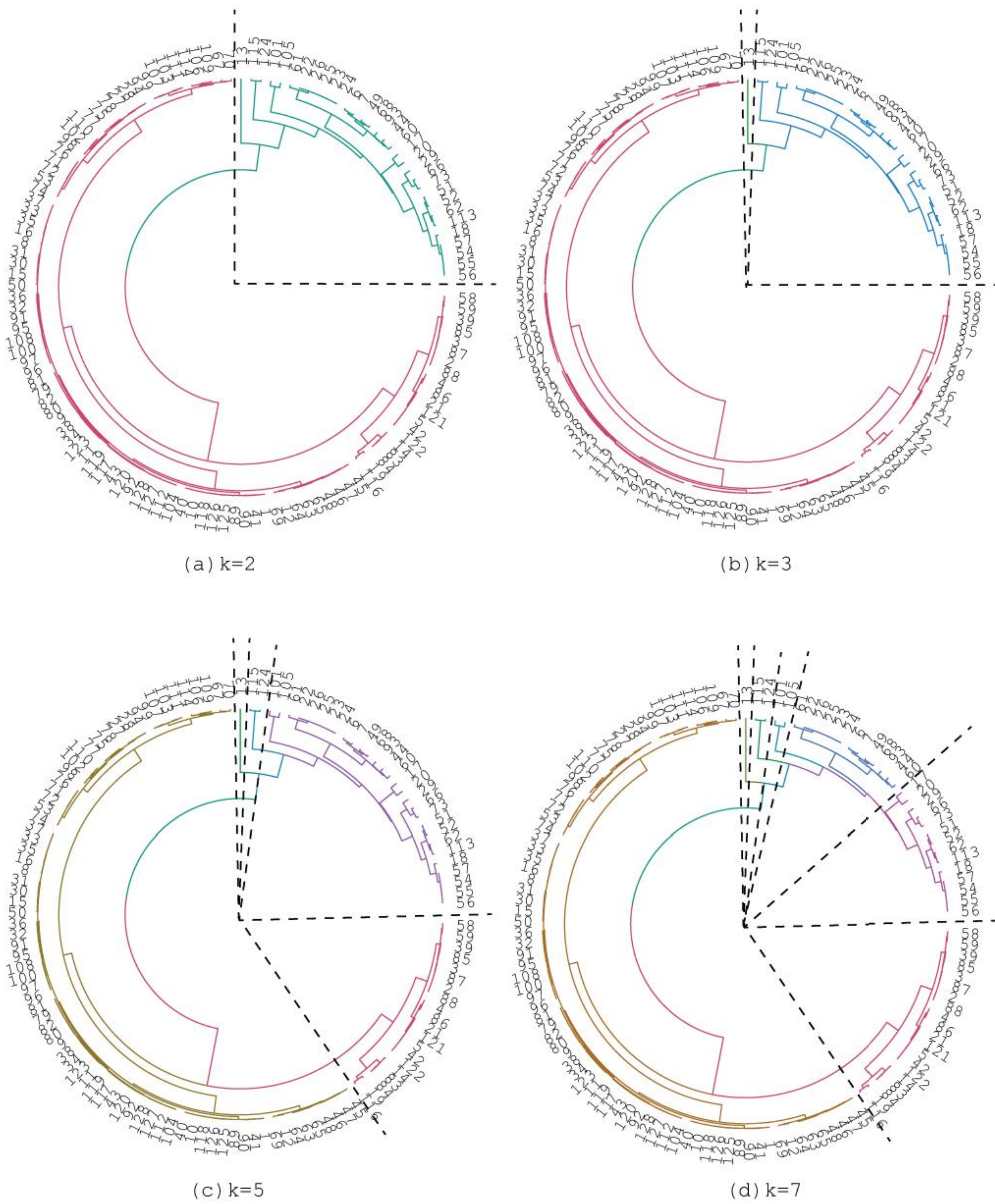


Figure 4.2: Radial dendrogram of Final Test



(a) k=2 (b) k=3

(c) k=5 (d) k=7

Figure 4.3: Clustering for several values of k

C. Optimal Number of Clusters

To avoid giving arbitrary values to k, three different methods for obtaining the optimal number of clusters were implemented in R. Figure 4.4 displays the outcome when

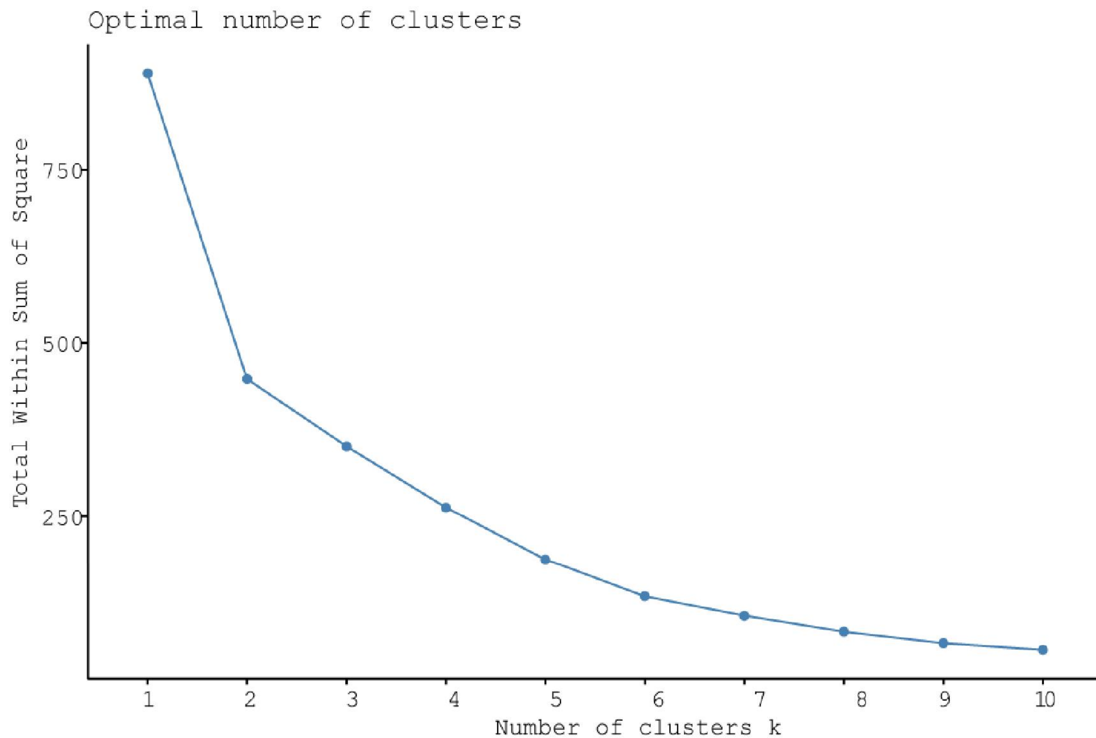



Figure 4.4: Optimal number of clusters selected by Elbow Method

applying the Elbow Method to the dissimilarity matrix. The function used for this step was *fviz_nbclust*. This method clearly indicates two as the optimal number of clusters since it is at this point where the inflection point can be appreciated. Figure 4.5 presents the result of applying the Average Silhouette Method to the dissimilarity Matrix. The function used for this step was *fviz_nbclust*. This method also marks two as the optimal number of clusters, as can be noted in the figure. Figure 4.6 applies the Gap Statistic Method to the matrix, obtaining a graph that indicates one as the optimal number of clusters. The function used for this step was fviz_gap_stat.

The majority of methods selected two as the optimal number of clusters, so the dendrogram displayed in Figure 4.3.a was the one designated as a final dendrogram, and the one we will use to discuss the results and get our conclusions from. This dendrogram is presented in the following section.

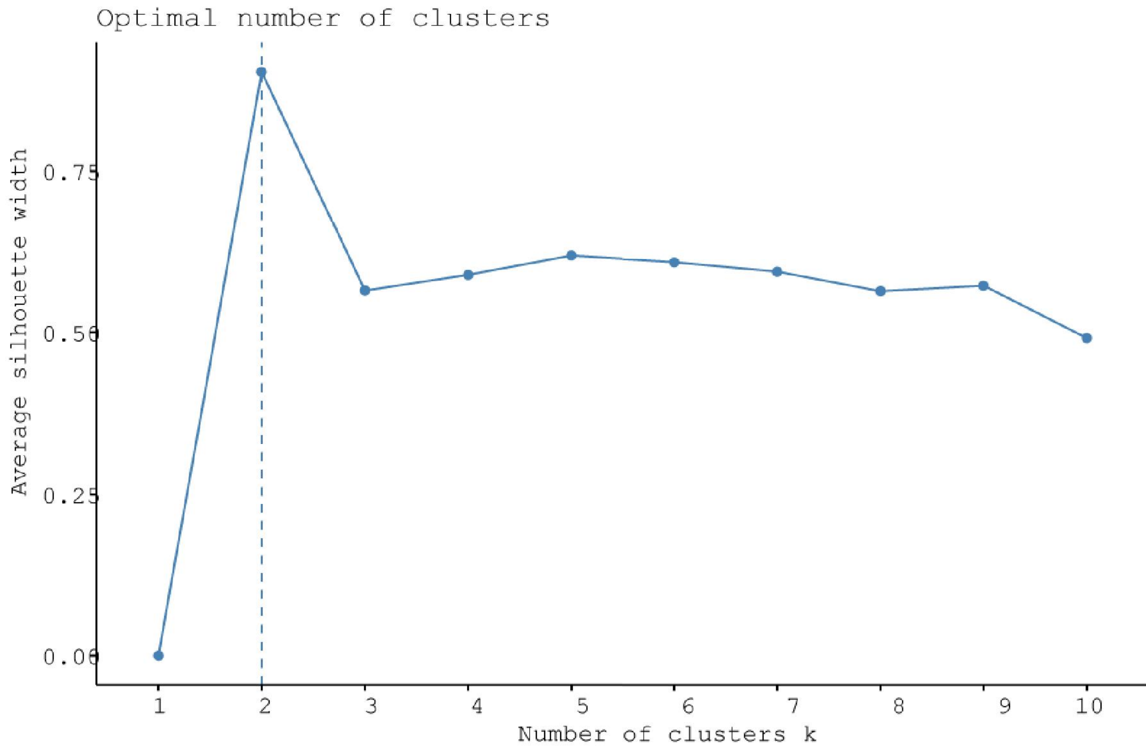



Figure 4.5: Optimal number of clusters selected by Average Silhouette Method

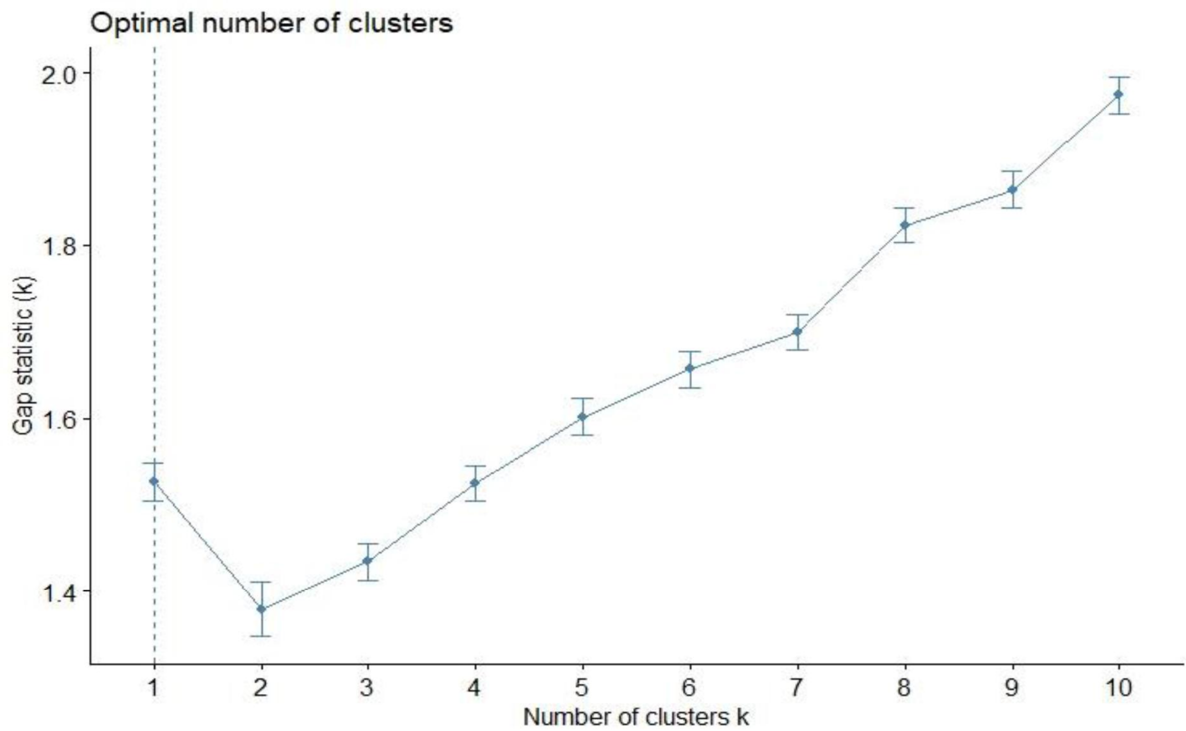



Figure 4.6: Optimal number of clusters selected by Gap Statistic Method

D. Final Radial Dendrogram

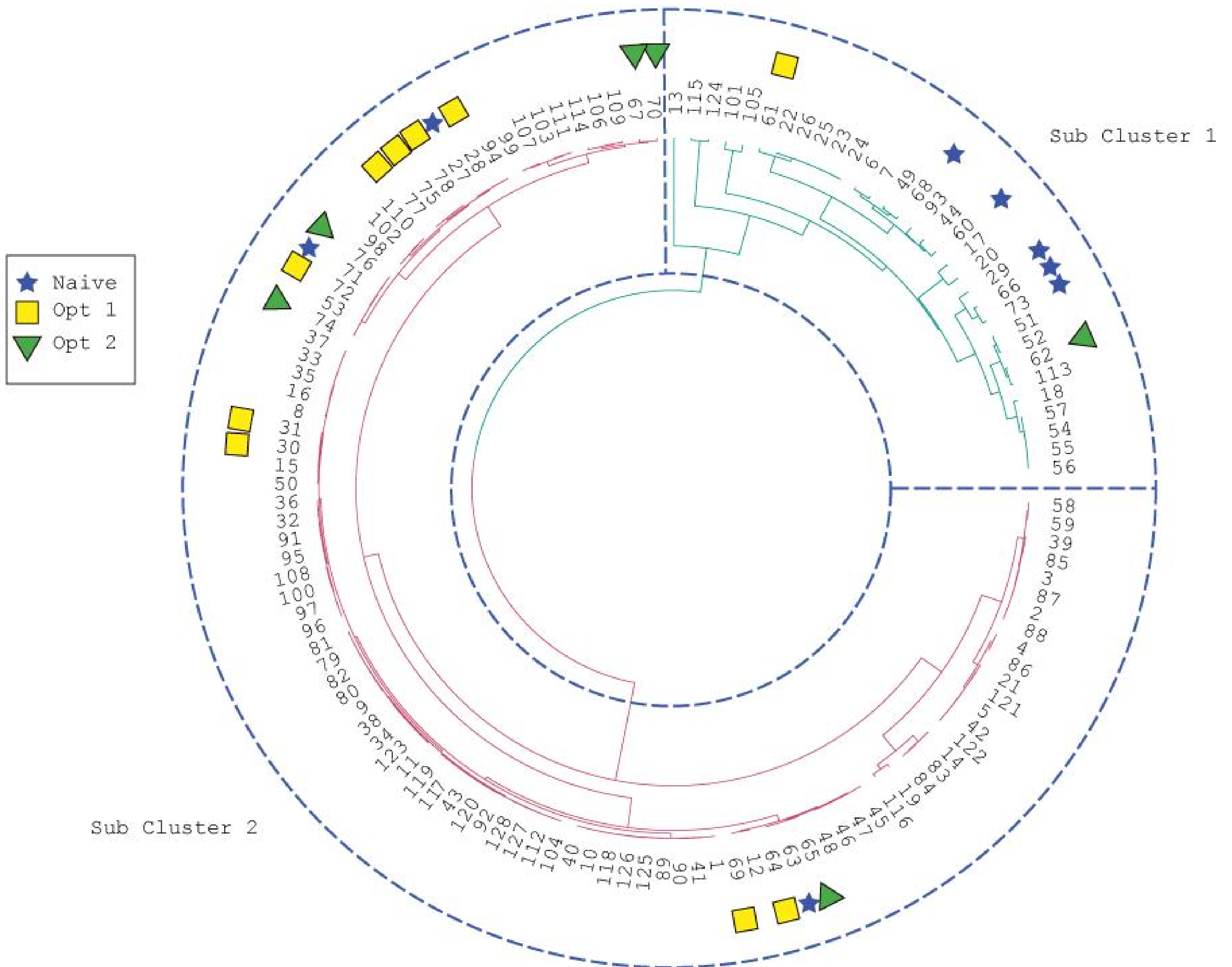



Figure 4.7: Final Radial Dendrogram with sub-clusters

## 4.4 Detecting Exploration kernels

To facilitate the identification of the Exploration Kernels, those that belong to naïve kernels, optimization 1 and optimization 2 in the dendrogram, were marked with a star, a square and a triangle, respectively. From Figure 4.7 it can be appreciated a clear tendency to separate the naïve versions from the rest of the kernels, 62 percent of naïve kernels ended up in Cluster 1, while Cluster 2 consists of 87 percent of the optimized

kernels. This distribution is also presented in Table 4.4. This could mean that the optimizations are having an impact in performance by reducing the kernels run time memory bottleneck generation and that this reduction is being successfully detected by the methodology.

Table 4.3: Distribution of the kernels on the clusters

| *Cluster* | *Naive* | *Optimized* |
|---|---|---|
| 1 | 5(62%) | 2(28%) |
| 2 | 3(13%) | 17(87%) |

Current distribution indicates that the spreading of the naïve and optimized kernels among the different sub clusters continues to be made in a relevant fashion despite having augmented the number of samples. There are still some apparently misclassified kernels, like naïve ones 27, 71 and 63 that ended up in Cluster 2 or 61 and 62, that ended up in Cluster one. Next section discussed this apparent error and attempts to make sense of the obtained results

## 4.5 Discussion

Taking a deeper dive into the resulting dendrogram there are some misclassified kernels like 61 and 62, that ended up in the naïve cluster, or 27, 63 and 71, that ended up leaning towards the optimized kernels. Kernels 60, 61 and 62 were extracted from [51], where an optimizing compiler for GPGPU applications was proposed. Being kernel 60 a naïve version of a kernel that executed a demosaic operation, 61 and 62 are optimized versions generated by that compiler, so the three of them are different versions of the same application. In [51], the optimizations made to 60 were also marked as the ones that produced less speedup which means that optimizations applied to this kernel were not as effective as the ones applied to the rest of the tests. The fact that kernels 61 and 62 ended up in the naïve cluster could be correlated with the relative (to the rest of the optimized kernels) lack of effectiveness of the optimizations applied to them in terms of memory bottleneck generation. Kernel 63 is next to 64 and 65, being the last two of them optimized versions of 63. This could be interpreted as a sign of little performance difference between these three kernels, it could also mean that kernel 63 does not exhibit many bottleneck related issues, despite being initially classified as a naïve kernel

Kernel 71 executes a matrix copy for benchmarking purposes, 72 is a version of the same kernel that uses shared memory and as stated in [50], no much difference in performance is obtained which explains the fact that the two of them are clustered together in the hierarchy (the same applies to 27 and 28, obtained from a different application). Authors make evident the performance improvement between transpose version 73 and 75, it also denotes that no significant improvement was produced by eliminating bank conflicts for 78, these observations are correctly identified. Kernel 74, which implements diagonal accesses on 78 and is the best version of 73 stays near to the previous versions but ends up closer to the copy benchmark kernels 71 and 72, this is coherent with results showed in the previously referenced work.

# Chapter 5

# Conclusion

## 5.1 Summary

The results indicate that the methodology can characterize kernels based upon their likelihood to present memory bottleneck related issues using a hierarchical clustering analysis. Additionally, the method is able to detect different levels of optimization in kernels among many other individuals and group in the hierarchy those that are known to have similar performance degradation. Therefore, the presented methodology may be used to study the nature of new workloads in terms of generation of bottlenecks, which should prove useful for developing new applications and to construct future prediction methodologies.

This section presents our principal contributions as the ones that have more impact or present novel propositions. Limitations to the method are also discussed in following sections and the future lines of research are proposed in order to build towards better characterizer and static predictors.

## 5.2 Contributions

The main contribution of this thesis resides in the method proposed, the characterization of kernels based on their likelihood to generate memory bottlenecks at run-time presents itself as a promising approach for future applications. We propose a method for extracting

memory request behavior history form a kernel running on a GPGPU simulator, using queues. We build up on previous research using a different kind of kernel signature, based on triangle-area approximation, that reflects its bottleneck-wise memory behavior. Hierarchical clustering analysis was used to classify more than 120 kernels extracted from 49 GPGPU applications. We utilize Euclidean distance and ward clustering method to construct a radial dendrogram. Exploration kernels, whose performance is generally known are used to test the characterization We show that there is a tendency to identify different levels of optimization in different sub clusters of the hierarchy.

## 5.3 Strengths and Limitations

The biggest strength of our methodology is that even tough is was implemented in Fermi architecture, it is not architecture dependent. Locating the Queue Counters has to do more with the memory hierarchy than with the structure of the components itself. This makes it adaptable to different architectures having alike memory hierarchy organization. Queues can be placed in the same way in any simulated GPU that possess similar organization with only a few changes, making the methodology adjustable to different and upcoming architectures.

Although the proposed methodology stands as a promising technique to characterize kernels bottleneck generation behavior, there are still opportunity areas that could benefit from more research in the future. Our approach exhibits a major drawback in the sense that the simulator, being one of our biggest strengths is also the general limiting factor for the proposed methodology. Despite the fact that the simulator allows us to obtain the bottleneck generation history of the memory partitions, simulating the kernels usually takes a considerable amount of time to finish execution. This feature slows down experimentation and makes it harder to add a larger number of kernels to the total of individuals that take part of the Hierarchical Clustering. The simulator also presents the limitation of not being able to execute kernels with graphical interaction, which decreases greatly the number of applications that can be selected for characterization purposes.

Another restraint is the number of architectures that can be simulated by GPGPU-Sim, we are limited only to the ones currently supported by the simulator. On the same way, the version of NVIDIA Cuda supported by it is far from recent, which also bounds the applications that can be selected for characterization. On the other hand, the technique currently used for obtaining que Peakness Coefficients based on triangle approximation also has its limitations. Approximating every peak area to the same geometric shape leaves out valuable information for the one that does not fit the selected shape. This technique could benefit from a polymorphic approach to obtaining a number that represents the peakness of the curve.

## 5.4 Future work

In terms of refining our current methodology, we plan to improve the accuracy of the Peakness Coefficient calculation by approximating the peaks not just to triangles but to trapezoids as well, since some bottlenecks exhibit this behavior. On the other hand, we plan to identify common static characteristics among kernels sharing the same sub clusters in the hierarchy. To do this would allow us to try to find a correlation between these characteristics and the bottleneck issues present in the specific sub cluster, building steps towards a static bottleneck predictor.

Following this plan of action, we broke down future work into three different stages:

*Stage one:*
Develop an engine able to identify common dynamic characteristics among the kernels in a sub-cluster, so valuable conclusion can be drawn from the hierarchy aside of the exploration kernels. This engine will be valuable to understand in an automated way relevant features of every cluster in the hierarchy.

*Stage two:*
Study the use of PCA as an alternative to reduce the number of dimensions. In this stage, we will also improve the peakness coefficient algorithm to make it polymorphic looking to enhance its accuracy.

*Stage three:*
Build an open source static characterizer for GPGPU applications. This characterizer will allow us to identify common static features of the kernels belonging to a certain sub-cluster. Once these characteristics are identified, correlations between them and specific bottleneck related issues identified in sub-clusters by *stage one* can be calculated. By completing this last step, a Predictive Methodology could be built based upon the Characterizing Methodology presented in this thesis.

# Bibliography


[1] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, “A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 272–281, Jan. 2015.

[2] “CUDA Zone | NVIDIA Developer.” [Online]. Available: https://developer.nvidia.com/cuda-zone. [Accessed: 07-Nov-2018].

[3] “OpenCL Overview - The Khronos Group Inc.” [Online]. Available: https://www.khronos.org/opencl/. [Accessed: 07-Nov-2018].

[4] G. Miao, N. Himayat, Y. (Geoffrey) Li, and A. Swami, “Cross-layer optimization for energy-efficient wireless communications: a survey,” *Wirel. Commun. Mob. Comput.*, vol. 9, no. 4, pp. 529–542, Apr. 2009.

[5] S. Mittal, “A Survey of Techniques For Improving Energy Efficiency in Embedded Computing Systems,” Jan. 2014.

[6] H. Vandierendonck and K. De Bosschere, “Many Benchmarks Stress the Same Bottlenecks,” *Work. Comput. Archit. Eval. Using Commer. Workload.*, no. June, pp. 57–64, 2004.

[7] F. Alted, “Why Modern CPUs Are Starving and What Can Be Done about It,” *Comput. Sci. Eng.*, vol. 12, no. 2, pp. 68–71, Mar. 2010.

[8] R. W. Hockney and I. J. Curington, “fl / 2 : A parameter to characterize memory and communication bottlenecks,” *Parallel Comput.*, vol. 10, pp. 277–286, 1989.

[9] S. Madougou, A. Varbanescu, C. De Laat, and R. Van Nieuwpoort, “The landscape of GPGPU performance modeling tools,” *Parallel Comput.*, vol. 56, pp. 18–33, 2016.

[10] R. Cammarota, A. Kejariwal, P. D’Alberto, S. Panigrahi, A. V. Veidenbaum, and A. Nicolau, “Pruning Hardware Evaluation Space via Correlation-driven Application Similarity Analysis,” p.

4:1–4:10, 2011.

[11] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron, “A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads,” in *IEEE International Symposium on Workload Characterization (IISWC’10)*, 2010, pp. 1–11.

[12] J. A. Gonzalez-Lugo, S. Rodriguez, A. Avila-Ortega, R. Cammarota, and N. Dutt, “Characterization of GPGPU workloads via correlation-driven kernel similarity analysis,” *Proc. - 2013 Int. Conf. Mechatronics, Electron. Automot. Eng. ICMEAE 2013*, pp. 199–204, 2013.

[13] S. Che and K. Skadron, “BenchFriend,” *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 238–250, Oct. 2013.

[14] S. A. Mirsoleimani, F. Khunjush, and A. Karami, “A two-tier design space exploration algorithm to construct {GPU} performance model,” *J. Syst. Archit.*, vol. 61, no. 10, pp. 576–583, 2015.

[15] N. P. Tran and M. Lee, “Parameter Tuning Model for Optimizing Application Performance on GPU,” *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. pp. 78–83, 2016.

[16] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 152, 2009.

[17] S. Madougou, A. L. Varbanescu, C. D. Laat, and R. V Nieuwpoort, “A Tool for Bottleneck Analysis and Performance Prediction for GPU-Accelerated Applications,” *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. pp. 641–652, 2016.

[18] Z. Hu, G. Liu, and Z. Hu, “A Performance Prediction Model for Memory-Intensive GPU Kernels,” *2014 IEEE Symposium on Computer Applications and Communications*. pp. 14–18, 2014.

[19] E. Konstantinidis and Y. Cotronis, “A Practical Performance Model for Compute and Memory Bound GPU Kernels,” *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. pp. 651–658, 2015.

[20] U. Gupta *et al.*, “Adaptive performance prediction for integrated GPUs,” *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. pp. 1–8, 2016.

[21] Y. Jiao, H. Lin, P. Balaji, and W. Feng, “Power and Performance Characterization of Computational Kernels on the GPU,” in *2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing*, 2010, pp. 221–228.

[22] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres, “Power and Performance Characterization and Modeling of GPU-Accelerated Systems,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 113–122.

[23] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, “Statistical power modeling of GPU kernels using performance counters,” in *International Conference on Green Computing*, 2010, pp. 115–122.

[24] Y. Wen, Z. Wang, and M. F. P. O’Boyle, “Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms,” in *2014 21st International Conference on High Performance Computing (HiPC)*, 2014, pp. 1–10.

[25] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing {CUDA} Workloads

Using a Detailed {GPU} Simulator," *Ispass*, pp. 163–174, 2009.

[26] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 3–12.

[27] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: a Parallel Functional Simulator for GPGPU," 2010.

[28] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," *Proc. 31st ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, p. 86, 2010.

[29] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications," *IEEE Int. Symp. Workload Charact. IISWC'10*, 2010.

[30] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. pp. 725–737, 2015.

[31] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU Workloads and Systems," *Comput. Eng.*, pp. 31–42, 2010.

[32] M. Amarís, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram, "A comparison of GPU execution time prediction using machine learning and analytical modeling," *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. pp. 326–333, 2016.

[33] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," *2015 IEEE 21st Int. Symp. High Perform. Comput. Archit.*, pp. 564–576, 2015.

[34] I. Baldini, S. J. Fink, and E. Altman, "Predicting GPU Performance from CPU Runs Using Machine Learning," *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. pp. 254–261, 2014.

[35] Nvidia, "Whitepaper NVIDIA's Next Generation CUDA Compute Architecture:" pp. 1–22, 2009.

[36] P. N. Glaskowsky, "NVIDIA's Fermi: The First Complete GPU Computing Architecture," 2009.

[37] N. Jiang, G. Michelogiannakis, D. Becker, B. Towles, and W. J. Dally, "Booksim 2.0 user�s guide," *Standford Univ.*, 2010.

[38] T. M. Aamodt, W. W. L. Fung, and T. H. Hetherington, "GPGPU-Sim Manual Main Page," 2017. [Online]. Available: http://gpgpu-sim.org/manual/index.php/Main_Page. [Accessed: 31-Oct-2018].

[39] J. Hair, W. Black, B. Babin, and R. Anderson, "Multivariate Data Analysis," 7th ed., Pearson Eaducation Limited, 2014, pp. 415–473.

[40] X. Mei, X. Chu, and S. Member, "Dissecting GPU Memory Hierarchy through Microbenchmarking," 2016.

[41] O. Maitre, "Understanding NVIDIA GPGPU hardware," *Natural Computing Series*. pp. 15–34, 2013.

[42] G. Palshikar, “Simple Algorithms for Peak Detection in Time-Series,” 2009.

[43] F. Scholkmann, J. Boss, and M. Wolf, “An Efficient Algorithm for Automatic Peak Detection in Noisy Periodic and Quasi-Periodic Signals,” *Algorithms*, vol. 5, no. 4, pp. 588–603, 2012.

[44] N. Singh, N. Garg, and J. Pant, “Article: A Comprehensive Study of Challenges and Approaches for Clustering High Dimensional Data,” *Int. J. Comput. Appl.*, vol. 92, no. 4, pp. 7–10, Apr. 2014.

[45] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, “On the Surprising Behavior of Distance Metrics in High Dimensional Space,” in *Database Theory --- ICDT 2001*, 2001, pp. 420–434.

[46] H. Abdi and L. J. Williams, “Principal component analysis,” *Wiley Interdiscip. Rev. Comput. Stat.*, vol. 2, no. 4, pp. 433–459, Jul. 2010.

[47] H. Abdi, “Coefficient of variation,” *Encycl. Res. Des.*, vol. 1, pp. 169–171, 2010.

[48] J. H. JR. Ward, “Hierarchical Grouping to Optimize an Objective Function,” *J. Am. Stat. Assoc.*, vol. 58, pp. 236–244, 1963.

[49] “CUDA Toolkit 4.0 | NVIDIA Developer.” [Online]. Available: https://developer.nvidia.com/cuda-toolkit-40. [Accessed: 29-Oct-2018].

[50] G. Ruetsch, P. Micikevicius, and T. Scudiero, “Optimizing Matrix Transpose in CUDA,” 2010.

[51] Y. Yang, P. Xiang, J. Kong, and H. Zhou, “A GPGPU Compiler for Memory Optimization and Parallelism Management,” pp. 86–97.

[52] S. Che *et al.*, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

# Appendix A

# Acronyms and Abbreviations Definitions

Table A.1: Acronyms Definitions

| *Acronyms or Abbreviation* | *Definition* |
|---|---|
| ALU | Arithmetic Logic Unit |
| AMD | Advanced Micro Devices |
| CC | CUDA core |
| CISC | Complex Instruction Set Computer |
| CMOS | Complementary Metal Oxide Semiconductor |
| CPU | Central Processing Unit |
| CU | Compute Unit |
| CUDA | Compute Unified Device Architecture |
| DRAM | Dynamic Random Access Memory |
| FIFO | First In First Out |
| FPU | Floating Point Unit |
| GPU | Graphic Processing Unit |
| GPGPU | General Purpose Graphic Processing Unit |
| IBM | International Business Machines |
| ICNT | InterConnection Network |
| ID | Identification |
| IPC | Instructions per cycle |
| OpenCL | Open Computing Language |
| PCA | Principal Component Analysis |
| PTX | Parallel Thread Execution |
| QC | Queue Counter |
| RISC | Reduced Instruction Set Computer |
| SM | Streaming Multiprocessor |
| SIMT | Single Instruction Multiple Thread |
| XAPP | Cross Architecture Performance Prediction |

# Appendix B

# Variables and symbols

Table B.1: Variables Definitions

| Symbol | Definition |
|---|---|
| pk | Peak |
| w | Widths |
| p | Prominences |
| sc | Scale coefficient |
| spc | Stage peakness coefficient |
| $C_v$ | Coefficient of variation |
| $\sigma$ | Standard deviation |
| $\mu$ | Mean |

# Appendix C: Configuration Options Fermi Architecture

Table C.1: Extended configuration options for simulated GTX480 Architecture

| *Configuration Variable* | *GTX480* |
|---|---|
| gpgpu_n_mem <# memory controller> | 6 |
| gpgpu_n_clusters | 15 |
| gpgpu_n_cores_per_cluster | 1 |
| gpgpu_num_sched_per_core | 2 |
| gpgpu_shmem_size | 49152 |
| gpgpu_num_reg_banks | 16 |
| gpgpu_n_sub_partition_per_mchannel | 2 |
| gpgpu_shader_registers | 32768 |
| gpgpu_shader_core_pipeline | 1536:32 |
| gpgpu_shader_cta | 8 |
| gpgpu_num_sp_units | 2 |
| gpgpu_num_sfu_units | 1 |
| gpgpu_cache:dl1 | 32:128:4,L:L:m:N:H,A:32:8,8 |
| gpgpu_shmem_size | 49152 |
| gpgpu_cache:dl2 | 64:128:8,L:B:m:W:L,A:32:4,4:0,32 |
| gpgpu_cache:dl2_texture_only | 0 |
| gpgpu_num_reg_banks | 16 |

Table C.1: Extended configuration options for simulated GTX480 Architecture(Continued)

| | |
|---|---|
| gpgpu_shmem_num_banks | 32 |
| gpgpu_max_insn_issue_per_warp | 1 |
| rop_latency | 120 |
| dram_latency | 100 |
| gpgpu_num_sched_per_core | 2 |
| gpgpu_frfcfs_dram_sched_queue_size | 16 |
| gpgpu_dram_return_queue_size | 116 |

# Appendix D: CSV files produced

Csv file produced by the instrumented simulator. This particular example is an extract of the output for memory partition 3. The file gives information about how many instructions were stalled in every QC at the registered cycles.

Table D.1: Extract of partition csv file produced

| Partition | Cycle | IN_ICNT_TO_MEM | IN_PARTITION_ICNT_TO_L2_QUEUE | IN_PARTITION_DRAM_LATENCY_QUEUE | IN_PARTITION_DRAM | IN_PARTITION_L2_TO_ICNT_QUEUE |
|---|---|---|---|---|---|---|
| 3 | 24287 | 0 | 0 | 1 | 0 | 1 |
| 3 | 24288 | 0 | 0 | 1 | 0 | 0 |
| 3 | 24289 | 0 | 1 | 1 | 0 | 0 |
| 3 | 24290 | 0 | 2 | 1 | 0 | 0 |
| 3 | 24291 | 0 | 1 | 1 | 0 | 2 |
| 3 | 24292 | 0 | 0 | 1 | 0 | 1 |
| 3 | 24293 | 0 | 2 | 1 | 0 | 0 |
| 3 | 24294 | 0 | 1 | 1 | 0 | 1 |
| 3 | 24295 | 0 | 2 | 1 | 0 | 0 |
| 3 | 24296 | 0 | 1 | 1 | 0 | 1 |
| 3 | 24297 | 0 | 2 | 1 | 0 | 0 |
| 3 | 24298 | 0 | 3 | 1 | 0 | 0 |
| 3 | 24299 | 0 | 3 | 1 | 0 | 1 |
| 3 | 24300 | 0 | 3 | 1 | 0 | 1 |
| 3 | 24301 | 0 | 3 | 1 | 0 | 1 |
| 3 | 24302 | 0 | 4 | 1 | 0 | 1 |
| 3 | 24303 | 0 | 3 | 1 | 0 | 1 |
| 3 | 24304 | 0 | 3 | 1 | 0 | 1 |
| 3 | 24305 | 0 | 2 | 2 | 0 | 2 |
| 3 | 24306 | 0 | 3 | 2 | 0 | 1 |
| 3 | 24307 | 0 | 3 | 2 | 0 | 2 |
| 3 | 24308 | 0 | 3 | 2 | 0 | 2 |
| 3 | 24309 | 0 | 4 | 2 | 0 | 2 |

Table D.1: Extract of partition csv file produced (Continued)

| 3 | 24310 | 0 | 5 | 2 | 0 | 2 |
|---|---|---|---|---|---|---|
| 3 | 24311 | 0 | 4 | 2 | 0 | 3 |
| 3 | 24312 | 0 | 4 | 2 | 0 | 3 |
| 3 | 24313 | 0 | 3 | 2 | 0 | 4 |
| 3 | 24315 | 0 | 2 | 2 | 0 | 5 |
| 3 | 24316 | 0 | 2 | 2 | 0 | 4 |
| 3 | 24317 | 0 | 1 | 2 | 0 | 5 |
| 3 | 24319 | 0 | 0 | 2 | 0 | 6 |
| 3 | 24325 | 0 | 1 | 2 | 0 | 6 |
| 3 | 24326 | 0 | 0 | 2 | 0 | 7 |
| 3 | 24327 | 0 | 1 | 2 | 0 | 7 |
| 3 | 24328 | 0 | 0 | 2 | 0 | 8 |
| 3 | 24332 | 0 | 0 | 2 | 0 | 7 |
| 3 | 24333 | 0 | 1 | 2 | 0 | 7 |
| 3 | 24334 | 0 | 0 | 2 | 0 | 8 |
| 3 | 24335 | 0 | 1 | 2 | 0 | 8 |
| 3 | 24336 | 0 | 1 | 2 | 0 | 8 |
| 3 | 24338 | 0 | 0 | 2 | 0 | 9 |
| 3 | 24339 | 1 | 1 | 2 | 0 | 9 |
| 3 | 24340 | 1 | 2 | 2 | 0 | 9 |
| 3 | 24341 | 1 | 2 | 2 | 0 | 10 |
| 3 | 24342 | 2 | 2 | 2 | 0 | 10 |
| 3 | 24343 | 3 | 1 | 2 | 0 | 9 |
| 3 | 24344 | 3 | 3 | 2 | 0 | 9 |
| 3 | 24345 | 3 | 2 | 3 | 0 | 9 |
| 3 | 24346 | 3 | 3 | 3 | 0 | 10 |
| 3 | 24347 | 4 | 2 | 4 | 0 | 10 |
| 3 | 24348 | 5 | 4 | 4 | 0 | 9 |
| 3 | 24349 | 5 | 4 | 4 | 0 | 8 |
| 3 | 24350 | 5 | 4 | 4 | 0 | 9 |
| 3 | 24351 | 5 | 2 | 4 | 0 | 11 |
| 3 | 24352 | 3 | 3 | 4 | 0 | 11 |
| 3 | 24355 | 3 | 2 | 4 | 0 | 12 |
| 3 | 24356 | 3 | 1 | 4 | 0 | 13 |
| 3 | 24357 | 3 | 2 | 5 | 0 | 12 |
| 3 | 24358 | 3 | 2 | 5 | 0 | 11 |

The following table is an extract from the final file, used for the signature engine written in Matlab. This file is final for a kernel execution, headers have been eliminated and partition file unified (every set of 5 columns represents a partition). The last column consists of the vector total memory requests. This file can be loaded directly into Matlab without any processing, speeding up the process.

Table D.2 Final File for Signature engine

| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 17629 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 17413 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 18558 |
| 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 19013 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 19920 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 19593 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 3 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Table D.2 Final File for Signature engine (Continued)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | 0 | 0 | 0 | 0 | 16 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 14 | 0 | 0 | 0 | 0 | 16 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | |
| 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 16 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 17 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | 18 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | |
| 0 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 19 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 11 | 0 | 0 | 0 | 0 | 19 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 18 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 19 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 3 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 20 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| 4 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 21 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| 5 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 22 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | |
| 7 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 22 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | |
| 6 | 0 | 0 | 0 | 0 | 14 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 22 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | |
| 7 | 0 | 0 | 0 | 0 | 16 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 21 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | |
| 9 | 0 | 0 | 0 | 0 | 18 | 0 | 0 | 0 | 0 | 11 | 0 | 0 | 0 | 0 | 21 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | |
| 8 | 0 | 0 | 0 | 0 | 19 | 0 | 0 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | 21 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | |

# Appendix E: Added and modified GPGPU-Sim code

Modified code is marked between the tags “//LAF”, files lista.cc and list.h were written from scratch

File name: gpgpusim_enrtypoint.cc

Directory:/gpgpu-sim_distribution/src

```
49    sem_t g_sim_signal_start;
50    sem_t g_sim_signal_finish;
51    sem_t g_sim_signal_exit;
52    time_t g_simulation_starttime;
53    pthread_t g_simulation_thread;
54
55    gpgpu_sim_config g_the_gpu_config;
56    gpgpu_sim *g_the_gpu;
57    stream_manager *g_stream_manager;
58
59    //LAF
60        PartitionQueue *part_q= new PartitionQueue();
61        int number_updates=0;
62        int last_one=0;
63        int kernel_number=1;
64    //LAF
65
66    static int sg_argc = 3;
67    static const char *sg_argv[] = {"", "-config","gpgpusim.config"};


237   void print_simulation_time()
238   {
239      time_t current_time, difference, d, h, m, s;
240      current_time = time((time_t *)NULL);
241      difference = MAX(current_time - g_simulation_starttime, 1);
242
243      d = difference/(3600*24);
244      h = difference/3600 - 24*d;
245      m = difference/60 - 60*(h + 24*d);
246      s = difference - 60*(m + 60*(h + 24*d));
247
248      fflush(stderr);
249      printf("\n\ngpgpu_simulation_time = %u days, %u hrs, %u min, %u sec (%u
250   sec)\n",
251               (unsigned)d, (unsigned)h, (unsigned)m, (unsigned)s,
252   (unsigned)difference );
```

```
253       printf("gpgpu_simulation_rate = %u (inst/sec)\n", (unsigned)(g_the_gpu-
254    >gpu_tot_sim_insn / difference) );
255       printf("gpgpu_simulation_rate = %u (cycle/sec)\n",
256    (unsigned)(gpu_tot_sim_cycle / difference) );
257
258        //LAF
259        printf("\nnumber of mem parts:%d\n",g_the_gpu->laf_mem_parts);
260        printf("Csv file done\n");
261        part_q->make_bneck_csv();
262        printf("bneck file done, number updates:%d\n",number_updates);
263        part_q->resetPartQueue(); //delete list after kernel is done executing
264        kernel_number++;
265        //LAF
266       fflush(stdout);
267    }
```

File name: gpu-sim.h

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
478      std::map<std::string, FuncCache> m_special_cache_config;
479
480      std::vector<std::string> m_executed_kernel_names; //< names of kernel for
481   stat printout
482      std::vector<unsigned> m_executed_kernel_uids; //< uids of kernel launches
483   for stat printout
484      std::string executed_kernel_info_string(); //< format the kernel
485   information into a string for stat printout
486      void clear_executed_kernel_info(); //< clear the kernel information after
487   stat printout
488
489   public:
490      unsigned long long  gpu_sim_insn;
491      unsigned long long  gpu_tot_sim_insn;
492       //LAF
493          unsigned laf_mem_parts;
494       //LAF
495      unsigned long long  gpu_sim_insn_last_update;
496      unsigned gpu_sim_insn_last_update_sid;
```

File name: l2cache.cc

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
151   bool memory_partition_unit::busy() const
152   {
153       bool busy = false;
154       for (unsigned p = 0; p < m_config->m_n_sub_partition_per_memory_channel;
155   p++) {
156           if (m_sub_partition[p]->busy()) {
157               busy = true;
158           }
159       }
160       return busy;
161   }
162   //LAF
163   //void memory_partition_unit::cache_cycle(unsigned cycle, double_llist
164   *list1)
165   //LAF
166   void memory_partition_unit::cache_cycle(unsigned cycle)
167   {
168       for (unsigned p = 0; p < m_config->m_n_sub_partition_per_memory_channel;
169   p++) {
170       //LAF
171           //m_sub_partition[p]->cache_cycle(cycle, list1);
172           //LAF
173       m_sub_partition[p]->cache_cycle(cycle);
174       }
175   }
```

```
200   void memory_partition_unit::dram_cycle()
201   {
202       // pop completed memory request from dram and push it to dram-to-L2 queue
203       // of the original sub partition
204       mem_fetch* mf_return = m_dram->return_queue_top();
205       if (mf_return) {
206           unsigned dest_global_spid = mf_return->get_sub_partition_id();
207           int dest_spid =
208   global_sub_partition_id_to_local_id(dest_global_spid);
209           assert(m_sub_partition[dest_spid]->get_id() == dest_global_spid);
210           if (!m_sub_partition[dest_spid]->dram_L2_queue_full()) {
211               if( mf_return->get_access_type() == L1_WRBK_ACC ) {
212                   m_sub_partition[dest_spid]->set_done(mf_return);
213                   delete mf_return;
214               } else {
215                   m_sub_partition[dest_spid]->dram_L2_queue_push(mf_return);
216                   mf_return-
217   >set_status(IN_PARTITION_DRAM_TO_L2_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
218               //LAF
219               mf_return->store();
220               //LAF
221                   m_arbitration_metadata.return_credit(dest_spid);
```

```
222                     MEMPART_DPRINTF("mem_fetch request %p return from dram to sub
223     partition %d\n", mf_return, dest_spid);
224                 }
225                 m_dram->return_queue_pop();
226             }
227         } else {
228             m_dram->return_queue_pop();
229         }
230
231         m_dram->cycle();
232         m_dram->dram_log(SAMPLELOG);
233
234         if( !m_dram->full() ) {
235             // L2->DRAM queue to DRAM latency queue
236             // Arbitrate among multiple L2 subpartitions
237             int last_issued_partition = m_arbitration_metadata.last_borrower();
238             for (unsigned p = 0; p < m_config-
239     >m_n_sub_partition_per_memory_channel; p++) {
240                 int spid = (p + last_issued_partition + 1) % m_config-
241     >m_n_sub_partition_per_memory_channel;
242                 if (!m_sub_partition[spid]->L2_dram_queue_empty() &&
243     can_issue_to_dram(spid)) {
244                     mem_fetch *mf = m_sub_partition[spid]->L2_dram_queue_top();
245                     m_sub_partition[spid]->L2_dram_queue_pop();
246                     MEMPART_DPRINTF("Issue mem_fetch request %p from sub
247     partition %d to dram\n", mf, spid);
248                     dram_delay_t d;
249                     d.req = mf;
250                     d.ready_cycle = gpu_sim_cycle+gpu_tot_sim_cycle + m_config-
251     >dram_latency;
252                     m_dram_latency_queue.push_back(d);
253                     mf-
254     >set_status(IN_PARTITION_DRAM_LATENCY_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
255             //LAF
256                 mf->store();
257             //LAF
258                     m_arbitration_metadata.borrow_credit(spid);
259                     break;  // the DRAM should only accept one request per cycle
260                 }
261             }
262         }
```

File name: gpu-sim.cc

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
558         m_shader_stats = new shader_core_stats(m_shader_config);
559         m_memory_stats = new
560     memory_stats_t(m_config.num_shader(),m_shader_config,m_memory_config);
561         average_pipeline_duty_cycle = (float *)malloc(sizeof(float));
562         active_sms=(float *)malloc(sizeof(float));
563         m_power_stats = new
564     power_stat_t(m_shader_config,average_pipeline_duty_cycle,active_sms,m_shader_
565     stats,m_memory_config,m_memory_stats);
566
567         gpu_sim_insn = 0;
568         gpu_tot_sim_insn = 0;
569         gpu_tot_issued_cta = 0;
570         gpu_deadlock = false;
571
572         //LAF
573         laf_mem_parts=m_memory_config->m_n_mem;
574         //LAF


1174

1175            // pop from memory controller to interconnect
1176            for (unsigned i=0;i<m_memory_config->m_n_mem_sub_partition;i++) {
1177                mem_fetch* mf = m_memory_sub_partition[i]->top();
1178                if (mf) {
1179                    unsigned response_size = mf->get_is_write()?mf-
1180    >get_ctrl_size():mf->size();
1181                    if ( ::icnt_has_buffer( m_shader_config->mem2device(i),
1182    response_size ) ) {
1183                        if (!mf->get_is_write())
1184
1185                           mf-
1186    >set_return_timestamp(gpu_sim_cycle+gpu_tot_sim_cycle);
1187                        mf-
1188    >set_status(IN_ICNT_TO_SHADER,gpu_sim_cycle+gpu_tot_sim_cycle);
1189                //LAF
1190                    mf->store();
1191                //LAF
1192                        ::icnt_push( m_shader_config->mem2device(i), mf-
1193    >get_tpc(), mf, response_size );
1194                        m_memory_sub_partition[i]->pop();
1195                    } else {
1196                        gpu_stall_icnt2sh++;
1197                    }
1198                } else {
1199                    m_memory_sub_partition[i]->pop();
1200                }
1201            }
1202        }
```

File name: mem_fetch.cc

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
109  //LAF
110  //Invoque the function store_req in "abstract_hardware_model.h", that stores
111  the path that every memory request follows trough the memory hierarchy
112  void mem_fetch::store()
113  {
114          mem_req_id=m_request_uid;
115          stages=m_status_change;
116          place=m_status;
117          store_m_sid=m_sid;
118          store_m_wid=m_wid;
119          store_m_raw_addr=m_raw_addr.chip;
120  }
121  //LAF
122  
123  //LAF //gpgpu-sim function that changes the status in memory of a request
124  void mem_fetch::set_status( enum mem_fetch_status status, unsigned long long
125  cycle )
126  {
127  
128      //LAF
129          store_m_raw_addr=m_raw_addr.chip;
130          if((m_status!=status)&&(cycle!=0))
131          {
132          //if the new and actual status are different, and the cycle isnt 0,
133  invoque bottle_store in "abstract_hardware_model.h", that stores the count of
134  queues on each cycle
135  
136  m_access.bottle_store(cycle,m_status,status,store_m_raw_addr,m_request_uid);
137          }
138      //LAF
139      m_status = status;
140      m_status_change = cycle;
141  }
```

File name: shader.cc

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
1449    void ldst_unit::fill( mem_fetch *mf )
1450    {
1451        mf-
1452    >set_status(IN_SHADER_LDST_RESPONSE_FIFO,gpu_sim_cycle+gpu_tot_sim_cycle);
1453        //LAF
1454            mf->store();
1455        //LAF
1456        m_response_fifo.push_back(mf);
1457    }
```

```
1818       if( !m_response_fifo.empty() ) {
1819           mem_fetch *mf = m_response_fifo.front();
1820           if (mf->istexture()) {
1821               if (m_L1T->fill_port_free()) {
1822                   m_L1T->fill(mf,gpu_sim_cycle+gpu_tot_sim_cycle);
1823                   m_response_fifo.pop_front();
1824               }
1825           } else if (mf->isconst())  {
1826               if (m_L1C->fill_port_free()) {
1827                   mf-
1828    >set_status(IN_SHADER_FETCHED,gpu_sim_cycle+gpu_tot_sim_cycle);
1829                //LAF
1830                    mf->store();
1831                //LAF
1832                   m_L1C->fill(mf,gpu_sim_cycle+gpu_tot_sim_cycle);
1833                   m_response_fifo.pop_front();
1834               }
```

```
1849                   if( bypassL1D ) {
1850                       if ( m_next_global == NULL ) {
1851                           mf-
1852    >set_status(IN_SHADER_FETCHED,gpu_sim_cycle+gpu_tot_sim_cycle);
1853                    //LAF
1854                    mf->store();
1855                    //LAF
1856                           m_response_fifo.pop_front();
1857                           m_next_global = mf;
1858                       }
```

```
2815    void shader_core_ctx::accept_fetch_response( mem_fetch *mf )
```

```
2816    {
2817         mf->set_status(IN_SHADER_FETCHED,gpu_sim_cycle+gpu_tot_sim_cycle);
2818             //LAF
2819             mf->store();
2820             //LAF
2821         m_L1I->fill(mf,gpu_sim_cycle+gpu_tot_sim_cycle);
2822    }
```

```
3323       if (!mf->get_is_write() && !mf->isatomic()) {
3324          packet_size = mf->get_ctrl_size();
3325       }
3326       m_stats->m_outgoing_traffic_stats->record_traffic(mf, packet_size);
3327       unsigned destination = mf->get_sub_partition_id();
3328       mf->set_status(IN_ICNT_TO_MEM,gpu_sim_cycle+gpu_tot_sim_cycle);
3329            //LAF
3330            mf->store();
3331            //LAF
3332       if (!mf->get_is_write() && !mf->isatomic())
3333          ::icnt_push(m_cluster_id, m_config->mem2device(destination), (void*)mf,
3334    mf->get_ctrl_size() );
3335       else
3336          ::icnt_push(m_cluster_id, m_config->mem2device(destination), (void*)mf,
3337    mf->size());
```

```
3369            m_stats->m_incoming_traffic_stats->record_traffic(mf, packet_size);
3370            mf-
3371    >set_status(IN_CLUSTER_TO_SHADER_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
3372                //LAF
3373                mf->store();
3374                //LAF
3375            //m_memory_stats->memlatstat_read_done(mf,m_shader_config-
3376    >max_warps_per_shader);
3377            m_response_fifo.push_back(mf);
```

File name: gpu-cache.cc

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
793  void baseline_cache::send_read_request(new_addr_type addr, new_addr_type
794  block_addr, unsigned cache_index, mem_fetch *mf,
795          unsigned time, bool &do_miss, bool &wb, cache_block_t &evicted,
796  std::list<cache_event> &events, bool read_only, bool wa){
797      bool mshr_hit = m_mshrs.probe(block_addr);
798      bool mshr_avail = !m_mshrs.full(block_addr);
799      if ( mshr_hit && mshr_avail ) {
800          if(read_only)
801              m_tag_array->access(block_addr,time,cache_index);
802          else
803              m_tag_array->access(block_addr,time,cache_index,wb,evicted);
804
805          m_mshrs.add(block_addr,mf);
806          do_miss = true;
807      } else if ( !mshr_hit && mshr_avail && (m_miss_queue.size() <
808  m_config.m_miss_queue_size) ) {
809          if(read_only)
810              m_tag_array->access(block_addr,time,cache_index);
811          else
812              m_tag_array->access(block_addr,time,cache_index,wb,evicted);
813
814          m_mshrs.add(block_addr,mf);
815          m_extra_mf_fields[mf] = extra_mf_fields(block_addr,cache_index, mf-
816  >get_data_size());
817          mf->set_data_size( m_config.get_line_sz() );
818          m_miss_queue.push_back(mf);
819          mf->set_status(m_miss_queue_status,time);
820          //LAF
821          mf->store();
822          //LAF
823          if(!wa)
824              events.push_back(READ_REQUEST_SENT);
825          do_miss = true;
826      }
827  }
828

829
830
831  /// Sends write request to lower level memory (write or writeback)
832  void data_cache::send_write_request(mem_fetch *mf, cache_event request,
833  unsigned time, std::list<cache_event> &events){
834      events.push_back(request);
835      m_miss_queue.push_back(mf);
836      mf->set_status(m_miss_queue_status,time);
837              //LAF
838              mf->store();
```

```
839                 //LAF
840     }



977         if( do_miss ){
978             // If evicted block is modified and not a write-through
979             // (already modified lower level)
980             if( wb && (m_config.m_write_policy != WRITE_THROUGH) ) {
981                 mem_fetch *wb = m_memfetch_creator->alloc(evicted.m_block_addr,
982                     m_wrbk_type,m_config.get_line_sz(),true);
983                 m_miss_queue.push_back(wb);
984                 wb->set_status(m_miss_queue_status,time);
985                 //LAF
986                 wb->store();
987                 //LAF
988             }
989             return MISS;
990         }
991
992         return RESERVATION_FAIL;



1378        assert( status != RESERVATION_FAIL );
1379        assert( status != HIT_RESERVED ); // as far as tags are concerned: HIT or
1380    MISS
1381        m_fragment_fifo.push( fragment_entry(mf,cache_index,status==MISS,mf-
1382    >get_data_size()) );
1383        if ( status == MISS ) {
1384            // we need to send a memory request...
1385            unsigned rob_index = m_rob.push( rob_entry(cache_index, mf,
1386    block_addr) );
1387            m_extra_mf_fields[mf] = extra_mf_fields(rob_index);
1388            mf->set_data_size(m_config.get_line_sz());
1389            m_tags.fill(cache_index,time); // mark block as valid
1390            m_request_fifo.push(mf);
1391            mf->set_status(m_request_queue_status,time);
1392                //LAF
1393                mf->store();
1394                //LAF
1395            events.push_back(READ_REQUEST_SENT);
1396            cache_status = MISS;
1397        } else {
1398            // the value *will* *be* in the cache already
1399            cache_status = HIT_RESERVED;
1400        }
1401        m_stats.inc_stats(mf->get_access_type(),
1402    m_stats.select_stats_status(status, cache_status));
1403        return cache_status;



1439    // Place returning cache block into reorder buffer
```

```
1440    void tex_cache::fill( mem_fetch *mf, unsigned time )
1441    {
1442        extra_mf_fields_lookup::iterator e = m_extra_mf_fields.find(mf);
1443        assert( e != m_extra_mf_fields.end() );
1444        assert( e->second.m_valid );
1445        assert( !m_rob.empty() );
1446        mf->set_status(m_rob_status,time);
1447                //LAF
1448                mf->store();
1449                //LAF
1450        unsigned rob_index = e->second.m_rob_index;
1451        rob_entry &r = m_rob.peek(rob_index);
1452        assert( !r.m_ready );
1453        r.m_ready = true;
1454        r.m_time = time;
1455        assert( r.m_block_addr == m_config.block_addr(mf->get_addr()) );
1456    }
```

File name: dram_sched.cc

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
145         req->data-
146     >set_status(IN_PARTITION_MC_INPUT_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
147           //LAF
148               req->data->store();
149           //LAF
150         sched->add_req(req);
151      }
152
153      dram_req_t *req;
154      unsigned i;
155      for ( i=0; i < m_config->nbk; i++ ) {
156         unsigned b = (i+prio)%m_config->nbk;
157         if ( !bk[b]->mrq ) {
158
159            req = sched->schedule(b, bk[b]->curr_row);
160
161            if ( req ) {
162               req->data-
163     >set_status(IN_PARTITION_MC_BANK_ARB_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
164           //LAF
165               req->data->store();
166           //LAF
167               prio = (prio+1)%m_config->nbk;
168               bk[b]->mrq = req;
169               if (m_config->gpgpu_memlatency_stat) {
170                  mrq_latency = gpu_sim_cycle + gpu_tot_sim_cycle - bk[b]->mrq-
171     >timestamp;
172                  bk[b]->mrq->timestamp = gpu_tot_sim_cycle + gpu_sim_cycle;
173                  m_stats->mrq_lat_table[LOGB2(mrq_latency)]++;
174                  if (mrq_latency > m_stats->max_mrq_latency) {
175                     m_stats->max_mrq_latency = mrq_latency;
176                  }
177               }
178
179               break;
180            }
181         }
182      }
```

File name: abstract_hardware_model.h

Directory:/gpgpu-sim_distribution/src

```
28  #ifndef ABSTRACT_HARDWARE_MODEL_INCLUDED
29  #define ABSTRACT_HARDWARE_MODEL_INCLUDED
30  //LAF
31  #include "gpgpu-sim/lista.h"
32  //LAF
33
34  // Forward declarations
35  class gpgpu_sim;
36  class kernel_info_t;
37
38  //Set a hard limit of 32 CTAs per shader [cuda only has 8]
39  #define MAX_CTA_PER_SHADER 32
40  #define MAX_BARRIERS_PER_CTA 16
41
42  //LAF
43  extern PartitionQueue *part_q;
44  //LAF
```

```
655     void print(FILE *fp) const
656     {
657         fprintf(fp,"addr=0x%llx, %s, size=%u, ", m_addr, m_write?"store":"load
658 ", m_req_size );
659         switch(m_type) {
660         case GLOBAL_ACC_R:  fprintf(fp,"GLOBAL_R"); break;
661         case LOCAL_ACC_R:   fprintf(fp,"LOCAL_R "); break;
662         case CONST_ACC_R:   fprintf(fp,"CONST   "); break;
663         case TEXTURE_ACC_R: fprintf(fp,"TEXTURE "); break;
664         case GLOBAL_ACC_W:  fprintf(fp,"GLOBAL_W"); break;
665         case LOCAL_ACC_W:   fprintf(fp,"LOCAL_W "); break;
666         case L2_WRBK_ACC:   fprintf(fp,"L2_WRBK "); break;
667         case INST_ACC_R:    fprintf(fp,"INST    "); break;
668         case L1_WRBK_ACC:   fprintf(fp,"L1_WRBK "); break;
669         //default:            fprintf(fp,"unknown "); break;
670      //LAF
671         case L1_WR_ALLOC_R:   fprintf(fp,"L1_ALLOC "); break;
672      case L2_WR_ALLOC_R:    fprintf(fp,"L2_ALLOC    "); break;
673         case NUM_MEM_ACCESS_TYPE:   fprintf(fp,"MUM_ACC_TYPE "); break;
674      //LAF
675      default:             fprintf(fp,"unknown %d",m_type); break;//LAF
676         }
677     }
678
679      //LAF
680      //function that stores changes in the table that that stores the path
681 that every memory request follows trough the memory hierarchy
```

```
682      void store_req(int mem_req_id, int stages, int place,int store_m_sid, int
683  store_m_wid, int store_m_raw_addr)
684      {
685          int store_type=m_type;
686          int store_addr=m_addr;
687          int store_size=m_req_size;
688          int store_write=m_write;
689
690  rec.addNode(mem_req_id,stages,place,store_m_sid,store_m_wid,store_m_raw_addr,
691  store_type,store_addr,store_size,store_write);
692
693      }
694
695      //function that invoques addPartitionQueue in lista.cc
696      void bottle_store(int _cycle,int place_prev, int place,int
697  store_m_raw_addr, unsigned m_request_uid)
698      {
699          part_q-
700  >addPartitionQueue(store_m_raw_addr,_cycle,place_prev,place,m_request_uid);
701
702      }
703      //LAF
```

File name: lista.h

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
1   #ifndef LISTA_H
2   #define LISTA_H
3
4   #include<iostream>
5   #include <fstream>
6   #include<cstdio>
7   #include<cstdlib>
8   #endif
9
10  //*********************************
11  //Queue_node Class
12  //*********************************
13  //Que_node Class holds a cycle number and a stages vector, this represents
14  the number of instructions are are being served in each stage at a given
15  cycle. Each cycle value is concatenated as a linked list
16  class Queue_node
17  {
18
19  friend class Queue_counter;
20  private:
21      int cycle;
22      int stages[24];
23      Queue_node *next;
24
25  public:
26      Queue_node(int &cycle_, int &stage_prev,int &stage_,Queue_node * prev,
27  int first);
28      ~Queue_node();
29      void Update_Queue_node(Queue_node * current, int &stage_prev,int stage_);
30
31  };
32
33  //*********************************
34  //Queue_counter Class
35  //*********************************
36  //Queue_counter class holds every Queue_node to contruct the bottleneck
37  linked list. Pointer "first" points to the first node and "last" to the last,
38  every node is concatenated. Variables por calculating the vector signature
39  are also defined in this class
40  class Queue_counter
41  {
42  friend class PartitionQueue;
43  private:
44      Queue_node *first;
45      Queue_node *last;
46      //correlation coeficients
47      //disabled on july 20
48      double signature[24];
49      float sum_x_square;
50      float sum_stage[24];
51      float sum_pro_x_stage[24];
```

```
52        float sum_stage_square[24];
53
54    public:
55        int total_number;
56        int sum_x;
57
58
59        Queue_counter();
60        ~Queue_counter();//Destructor
61        void addQueueNode(int&,int&,int&);
62        //void make_bneck_csv();
63        void make_part_csv(std::ofstream& file, int partition_number);
64        void make_short_part_csv(std::ofstream& file, int partition_number);
65        void signature_make_part_csv(std::ofstream& file1);
66
67    };
68
69    //**********************************
70    //PartitionNode Class
71    //**********************************
72    //Class that holds a bottleneck tables for a specific partition, this
73    partition is specified in part_number variable. Bottleneck table is stored in
74    object of type Queue_counter class, pointer "queue" points to it. Pointer
75    "next" points to the next PartitionNode
76
77    class PartitionNode
78    {
79    friend class PartitionQueue;
80
81    private:
82        int part_number;
83        unsigned pre_req;
84        int req_counter;
85        Queue_counter *queue;
86        PartitionNode *next;
87
88    public:
89        ~PartitionNode();
90        PartitionNode(int &part,int &cycle_, int &stage_prev,int &stage_,unsigned
91    m_request_uid);
92        void SetQueueNull();
93        void Update_Part(PartitionNode *actual, int &cycle_, int &stage_prev,int
94    &stage_,unsigned m_request_uid);
95
96    };
97
98    //**********************************
99    //PartitionQueue Class
100   //**********************************
101   //Class that holds a pointer to a linked list consisting of several nodes,
102   where everyone of them holds the partition table. The pointer to the first
103   node is called ini
104   class PartitionQueue
105   {
106       private:
107           PartitionNode *ini;
108           unsigned full_req_counter[6];
```

```
109
110         public:
111         PartitionQueue();
112         ~PartitionQueue(); //destructor
113         void resetPartQueue();
114         void addPartitionQueue(int&,int&,int&,int&,unsigned);
115         void make_bneck_csv();
116     };
```

File name: lista.cc

Directory:/gpgpu-sim_distribution/src/gpgpu-sim

```
1   #include <math.h>
2   #include <cerrno>
3   #include <cstring>
4   #include <sys/stat.h>
5   #include <stdlib.h>
6   #include "lista.h"
7
8   extern char * csvfilename;
9   extern char * kernelname;
10  extern int number_updates;
11  extern int kernel_number;
12
13  //*********************************
14  //Queue_node Class
15  //*********************************
16  //Queue_node Class holds a cycle number and a stages vector, this represents
17  the number of instructions are are being served in each stage at a given
18  cycle. Each cycle value is concatenated as a linked list
19
20
21  //function that creates a new node if a new cycle is added to the linked list
22  Queue_node::Queue_node(int &cycle_ , int &stage_prev,int &stage_,Queue_node *
23  prev, int primer)
24  {
25      cycle=cycle_;
26
27      for(int conta = 0; conta < 24; conta++)
28      {
29          if(primer==1)
30          {
31          stages[conta] = 0;
32          }
33          else
34          {
35              stages[conta] = prev->stages[conta];
36          }
37      }
38
39      switch(stage_){
40      case 0: break;
41      case 1: stages[1]++; break;
42      case 2: stages[1]++; break;
43      case 3: stages[1]++; break;
44      case 4: stages[1]++; break;
45      case 11: stages[9]++; break;
46      case 12: stages[8]++; break;
47      default: stages[stage_-3]++; break;
48      }
49
50      switch(stage_prev){
51      case 0: break;
```

```
52        case 1: if(stages[1]>0){stages[1]--;}; break;
53        case 2: if(stages[1]>0){stages[1]--;}; break;
54        case 3: if(stages[1]>0){stages[1]--;}; break;
55        case 4: if(stages[1]>0){stages[1]--;}; break;
56        case 11: if(stages[9]>0){stages[9]--;}; break;
57        case 10: if(stages[8]>0){stages[8]--;}; break;
58        default: if(stages[stage_prev-3]>0){stages[stage_prev-3]--;}; break;
59        }
60        next=NULL;
61
62    }
63
64    Queue_node::~Queue_node()
65    {
66    }
67
68    //function that updates the value of a node in the linked list
69    void Queue_node::Update_Queue_node(Queue_node * current, int &stage_prev,int
70    stage_)
71    {
72
73        switch(stage_){
74        case 0: break;
75        case 1: stages[1]++; break;
76        case 2: stages[1]++; break;
77        case 3: stages[1]++; break;
78        case 4: stages[1]++; break;
79        case 11: stages[9]++; break;
80        case 12: stages[8]++; break;
81        default: stages[stage_-3]++; break;
82        }
83
84        switch(stage_prev){
85        case 0: break;
86        case 1: if(stages[1]>0){stages[1]--;}; break;
87        case 2: if(stages[1]>0){stages[1]--;}; break;
88        case 3: if(stages[1]>0){stages[1]--;}; break;
89        case 4: if(stages[1]>0){stages[1]--;}; break;
90        case 11: if(stages[9]>0){stages[9]--;}; break;
91        case 10: if(stages[8]>0){stages[8]--;}; break;
92        default: if(stages[stage_prev-3]>0){stages[stage_prev-3]--;}; break;
93        }
94    }
95
96    //*********************************
97    //Queue_counter Class
98    //*********************************
99    //Queue_counter class holds every Queue_node to contruct the bottleneck
100   table. Pointer "first" points to the first node and "last" to the last, every
101   node is concatenated. Variables por calculating the vector signature are also
102   defined in this class
103
104   //Class constructor
105   Queue_counter::Queue_counter()
106   {
107       first = NULL;
108       total_number=1;
```

```
109         last = NULL;
110     }
111 
112     Queue_counter::~Queue_counter()
113     {
114         delete first;
115         delete last;
116     }
117 
118     //function that inserts a new node in the linked list
119     void Queue_counter::addQueueNode(int&cycle_,int&stage_prev,int&stage)
120     {
121 
122         if (first == NULL)//empty linked list
123         {
124             int b;
125             Queue_node *new_node = new
126     Queue_node(cycle_,stage_prev,stage,NULL,1);//create a new node
127             first = new_node;
128             last =new_node;
129             new_node->next=NULL;
130 
131             return;
132 
133         }
134 
135         else
136         {
137             if(last->cycle==cycle_)
138             {
139                 last->Update_Queue_node(last,stage_prev,stage); //if it is at the
140     end of the list, update the node
141             }
142             else
143             {
144 
145                 Queue_node *new_node = new
146     Queue_node(cycle_,stage_prev,stage,last,0);//create a new node if no match in
147     ids
148                 last->next = new_node;
149                 new_node->next=NULL;
150                 last=new_node;
151                 total_number=total_number+1;
152             }
153         }
154     }
155 
156     //Function that writes the corresponding bottlenecks table of a partition
157     void Queue_counter::make_part_csv(std::ofstream& file, int partition_number)
158     {
159         //NOTE: stages vector indexes are prunned so only the useful stages are
160     written
161         printf("writing partition table\n");
162         Queue_node *aux_node = first;
163         Queue_node *to_free;
164         while (aux_node != NULL)
165             {
```

```
166
167                 file <<std::to_string(static_cast<long
168 long>(partition_number))+","+std::to_string(static_cast<long long>(aux_node-
169 >cycle))+","+std::to_string(static_cast<long long>(aux_node-
170 >stages[4]))+","+std::to_string(static_cast<long long>(aux_node-
171 >stages[6]))+","+std::to_string(static_cast<long long>(aux_node-
172 >stages[9]))+","+std::to_string(static_cast<long long>(aux_node-
173 >stages[13]))+","+std::to_string(static_cast<long long>(aux_node-
174 >stages[17]))+"\n";
175
176             to_free=aux_node;
177             aux_node=aux_node->next;
178             delete to_free;
179             }
180             first=NULL;
181             last=NULL;
182
183     }
184
185     void Queue_counter::make_short_part_csv(std::ofstream& file, int
186     partition_number)
187     {
188         //NOTE: stages vector indexes are prunned so only the useful stages are
189     written
190         printf("writing partition table\n");
191         Queue_node *aux_node = first;
192         while (aux_node != NULL)
193             {
194
195                 file <<std::to_string(static_cast<long long>(aux_node-
196 >stages[4]))+","+std::to_string(static_cast<long long>(aux_node-
197 >stages[6]))+","+std::to_string(static_cast<long long>(aux_node-
198 >stages[9]))+","+std::to_string(static_cast<long long>(aux_node-
199 >stages[13]))+","+std::to_string(static_cast<long long>(aux_node-
200 >stages[17]))+"\n";
201
202             aux_node=aux_node->next;
203             }
204
205     }
206
207     //Function that writes the signature vector in a given file
208     void Queue_counter::signature_make_part_csv(std::ofstream& file1)
209     {
210         int b;
211         float x_avg;
212         double y_avg[24]={0.0};
213         double up_coef[24]={0.0};
214         double down_coef1[24]={0.0};
215         double down_coef2[24]={0.0};
216
217         Queue_node *aux;
218         Queue_node *to_free;
219
220         file1<<",";
221         printf("writing signature partition table\n");
222
```

```
223         for(b=0;b<=23;b++)
224         {
225             signature[b]=0.0;
226         }
227
228         //sum of the last added cycle
229         sum_x=sum_x+(last->cycle);
230         x_avg=((double)sum_x)/(double(total_number));
231         aux=first;
232         while(aux!=NULL)//iterate through all the nodes(cycles) while
233     constructing the correlation terms
234         {
235             for (int k=0; k<24;k++)//iterate through all the stage columns
236             {
237                 if((k!=0)&(k!=2)&(k!=3)&(k!=22)&(k!=23))//ignore stages that wont
238     be used
239                 {
240                     //sum of the last node of each stage
241                     sum_stage[k]=sum_stage[k]+(last->stages[k]);
242
243                     y_avg[k]=(sum_stage[k])/((double)total_number);
244
245
246                     up_coef[k]=up_coef[k]+(((aux->cycle)-(x_avg))*((aux-
247     >stages[k])-(y_avg[k])));
248                     down_coef1[k]=down_coef1[k]+(((aux->cycle)-(x_avg))*((aux-
249     >cycle)-(x_avg)));
250                     down_coef2[k]=down_coef2[k]+(((aux->stages[k])-
251     (y_avg[k]))*((aux->stages[k])-(y_avg[k])));
252
253
254                     if(aux->next==NULL)
255                     {
256
257     signature[k]=(up_coef[k])/((sqrt(down_coef1[k]))*(sqrt(down_coef2[k])));
258                         file1 <<std::to_string(static_cast<long
259     double>(signature[k]))+",";
260                     }
261                 }
262             }
263             //delete head of the list
264             to_free=aux;
265             aux=aux->next;
266             delete to_free;
267         }
268         file1<<"\n\n\n";
269
270     }
271
272     //*********************************
273     //PartitionNode Class
274     //*********************************
275     //Class that holds a bottleneck tables for a specific partition, this
276     partition is specified in part_number variable. Bottleneck table is stored in
277     object of type Queue_counter class, pointer "queue" points to it. Pointer
278     "next" points to the next PartitionNode
279
```

```
280
281    //Function that creates a new Queue_counter object, set the partition number
282    and then add a new node to the queue
283    PartitionNode::PartitionNode(int &part,int &cycle_, int &stage_prev,int
284    &stage_,unsigned m_request_uid)
285    {
286
287        queue= new Queue_counter();
288        part_number=part;
289
290        req_counter=1;
291        pre_req=m_request_uid;
292        queue->addQueueNode(cycle_,stage_prev,stage_);
293        next=NULL;
294
295    }
296
297    PartitionNode::~PartitionNode()
298    {
299        delete queue;
300    }
301
302    void PartitionNode::SetQueueNull()
303    {
304        queue=NULL;
305    }
306    //Function that adds a new node in a queue given the pointer to the
307    corresponding PartitionNode
308    void PartitionNode::Update_Part(PartitionNode *actual, int &cycle_, int
309    &stage_prev,int &stage_,unsigned m_request_uid)
310    {
311        if(pre_req!=m_request_uid)
312        {
313            pre_req=m_request_uid;
314            req_counter++;
315        }
316        actual->queue->addQueueNode(cycle_,stage_prev,stage_);
317    }
318
319    //*********************************
320    //PartitionQueue Class
321    //*********************************
322    //Class that holds a pointer to a linked list consisting of several nodes,
323    where everyone of them holds the partition table. The pointer to the first
324    node is called ini
325
326    //Constructor class
327    //First node initialized
328    PartitionQueue::PartitionQueue()
329    {
330
331        full_req_counter[4]={0};
332        ini = NULL;
333    }
334
335    PartitionQueue::~PartitionQueue()
336    {
```

```
337        delete ini;
338    }
339    //function that resets the partitions linked list
340    void PartitionQueue::resetPartQueue()
341    {
342
343        ini=NULL;
344    }
345
346    //function that gets called everytime there is a change in the queues, and
347    selects wich partition linked list it belongs to, and invoques the
348    corresponding function
349    void PartitionQueue::addPartitionQueue(int &part,int&cycle_,int
350    &stage_prev,int&stage,unsigned m_request_uid)
351    {
352        if (ini == NULL)//no partition linked list have been initialized
353        {
354            PartitionNode *new_part = new
355    PartitionNode(part,cycle_,stage_prev,stage,m_request_uid);//create a new node
356            ini = new_part;
357            return;
358        }
359
360        else
361        {
362            PartitionNode *temp_node = ini;//declares a PartitionNode data type
363    to start at ini node
364
365
366            while (temp_node->next != NULL)//searches trough the list for the
367    corresponding partition list
368            {
369                if(part==(temp_node->part_number))
370                {
371
372                    temp_node-
373    >Update_Part(temp_node,cycle_,stage_prev,stage,m_request_uid); //when
374    corresponding partition list is found, updates it with the new information
375                    return;
376                }
377                temp_node = temp_node->next;
378            }
379
380            if(part==(temp_node->part_number))//if the corresponding partition
381    list is at the end of the list
382            {
383                temp_node-
384    >Update_Part(temp_node,cycle_,stage_prev,stage,m_request_uid);
385                return;
386            }
387
388            PartitionNode *new_part = new
389    PartitionNode(part,cycle_,stage_prev,stage,m_request_uid);//create a new
390    partition list if the partition number is not found
391            temp_node->next = new_part;
392        }
393    }
```

```
394
395
396   //function that creates the bottlenecks and signature csv files
397   void PartitionQueue::make_bneck_csv()
398   {
399       int lp;
400       int done_vector[6]={-1,-1,-1,-1,-1,-1};
401       //writing the bottleneck file
402       std::string kernelString=kernelname;
403       std::string file_to_write=csvfilename;
404       std::string
405   dirname=std::string("./")+kernelString+std::to_string(static_cast<long
406   long>(kernel_number));
407
408       int first_delim_pos = file_to_write.find_last_of('/');
409
410       if(first_delim_pos!=-1)//if it is indeed a directory
411       {
412           file_to_write=file_to_write.substr(first_delim_pos+1);
413       }
414
415
416           printf("writing cycle starts\n");
417           PartitionNode *temp_node=ini;
418           PartitionNode *to_delete;
419
420           int dir_err = mkdir(dirname.c_str(),S_IRWXU | S_IRWXG | S_IRWXO);
421               if(dir_err==-1)
422               {
423                   printf("DIRECTORY ERROR\n");
424               }
425           while(temp_node!=NULL)
426           {
427               //short file writing cycle starts
428               std::string short_csv_name =
429   dirname+std::string("/")+std::string("file")+std::to_string(static_cast<long
430   long>(temp_node->part_number))+".csv";
431               printf("opening short file\n");
432               std::ofstream file_short;
433               file_short.open (short_csv_name);
434               temp_node->queue->make_short_part_csv(file_short,temp_node-
435   >part_number);
436               file_short.close();
437               printf("closing short file\n");
438               //short file writing cycle ends
439
440               std::string bottle_csv_name =
441   dirname+std::string("/")+kernelString+std::string("_number_")+std::to_string(
442   static_cast<long
443   long>(kernel_number))+std::string("_PARTITION_")+std::to_string(static_cast<l
444   ong long>(temp_node->part_number))+std::string("_")+file_to_write+".csv";
445               printf("opening bneck file\n");
446               std::ofstream file;
447               file.open (bottle_csv_name);
448
449                   if (file.fail())
450               {
```

```
451
452                     std::cerr << "Error while opening file: " <<
453     strerror(errno)<< '\n';
454                     break;
455
456                 }
457
458                 else
459                 {
460                     full_req_counter[temp_node->part_number]=temp_node-
461     >req_counter;
462
463     file<<"Partition,Cycle,IN_ICNT_TO_MEM,IN_PARTITION_ICNT_TO_L2_QUEUE,IN_PARTIT
464     ION_DRAM_LATENCY_QUEUE,IN_PARTITION_DRAM,IN_PARTITION_L2_TO_ICNT_QUEUE\n";
465                     temp_node->queue->make_part_csv(file,temp_node->part_number);
466
467                     //modify done_vector
468                     done_vector[temp_node->part_number]=1;
469                     delete temp_node->queue;
470                     temp_node->SetQueueNull();
471                     to_delete=temp_node;
472                     temp_node=temp_node->next;
473                     delete to_delete;
474                     file.close();
475                 }
476
477
478
479             }
480             printf("writing cycle ends, closing file:%s\n",csvfilename);
481
482
483             //Partition corrector cycle starts
484             printf("partition corrector:\n");
485             for(lp=0;lp<=5;lp++)
486             {
487                 if(done_vector[lp]==-1)
488                 {
489
490                     full_req_counter[lp]=0;
491
492                     std::string s_csv_name =
493     dirname+std::string("/")+std::string("file")+std::to_string(static_cast<long
494     long>(lp))+".csv";
495                     printf("opening short file\n");
496                     std::ofstream file_s;
497                     file_s.open (s_csv_name);
498                     file_s<<"-1,-1,-1,-1,-1\n";
499                     file_s.close();
500                     printf("closing short file\n");
501                 }
502
503
504             }
505             //partition corrector cycle ends
506
507             //partition file writing cycle starts
```

```
508         printf("writing cycle starts, writing request counter file:\n");
509         std::string req_count_csv =
510 dirname+std::string("/")+std::string("Requests.csv");
511         std::ofstream file1;
512         file1.open (req_count_csv);
513         file1<<std::to_string(static_cast<long
514 long>(full_req_counter[0]))+"\n"+std::to_string(static_cast<long
515 long>(full_req_counter[1]))+"\n"+std::to_string(static_cast<long
516 long>(full_req_counter[2]))+"\n"+std::to_string(static_cast<long
517 long>(full_req_counter[3]))+"\n"+std::to_string(static_cast<long
518 long>(full_req_counter[4]))+"\n"+std::to_string(static_cast<long
519 long>(full_req_counter[5]));
520
521         printf("writing cycle ends, closing request counter file:\n");
522
523         file1.close();
524         //partition file writing cycle starts
525
526         //system calls
527
528         std::string cmd0=std::string("paste -d,
529 ")+dirname+std::string("/")+std::string("file5.csv
530 ")+dirname+std::string("/")+std::string("Requests.csv | sed
531 's/^,/,,,,,/g'>")+dirname+std::string("/")+std::string("temp5.csv ");
532         std::string cmd1=std::string("paste -d,
533 ")+dirname+std::string("/")+std::string("file4.csv
534 ")+dirname+std::string("/")+std::string("temp5.csv | sed
535 's/^,/,,,,,/g'>")+dirname+std::string("/")+std::string("temp45.csv ");
536         std::string cmd2=std::string("paste -d,
537 ")+dirname+std::string("/")+std::string("file3.csv
538 ")+dirname+std::string("/")+std::string("temp45.csv | sed
539 's/^,/,,,,,/g'>")+dirname+std::string("/")+std::string("temp34.csv ");
540         std::string cmd3=std::string("paste -d,
541 ")+dirname+std::string("/")+std::string("file2.csv
542 ")+dirname+std::string("/")+std::string("temp34.csv | sed
543 's/^,/,,,,,/g'>")+dirname+std::string("/")+std::string("temp23.csv ");
544         std::string cmd4=std::string("paste -d,
545 ")+dirname+std::string("/")+std::string("file1.csv
546 ")+dirname+std::string("/")+std::string("temp23.csv | sed
547 's/^,/,,,,,/g'>")+dirname+std::string("/")+std::string("temp12.csv ");
548         std::string cmd5=std::string("paste -d,
549 ")+dirname+std::string("/")+std::string("file0.csv
550 ")+dirname+std::string("/")+std::string("temp12.csv | sed
551 's/^,/,,,,,/g'>")+dirname+std::string("/")+std::string("final_file.csv ");
552         std::string cmd6=std::string("rm
553 ")+dirname+std::string("/")+std::string("temp5.csv");
554         std::string cmd7=std::string("rm
555 ")+dirname+std::string("/")+std::string("temp45.csv");
556         std::string cmd8=std::string("rm
557 ")+dirname+std::string("/")+std::string("temp34.csv");
558         std::string cmd9=std::string("rm
559 ")+dirname+std::string("/")+std::string("temp23.csv");
560         std::string cmd10=std::string("rm
561 ")+dirname+std::string("/")+std::string("temp12.csv");
562
563         system((cmd0).c_str());
564         system((cmd1).c_str());
```

```
565             system((cmd2).c_str());
566             system((cmd3).c_str());
567             system((cmd4).c_str());
568             system((cmd5).c_str());
569
570
571             system((cmd6).c_str());
572             system((cmd7).c_str());
573             system((cmd8).c_str());
574             system((cmd9).c_str());
575             system((cmd10).c_str());
576             //end of system calls
577
578
579     }
```

File name: cuda_runtime_api.cc

Directory:/gpgpu-sim_distribution/libcuda

```
140    #ifdef __APPLE__
141    #include <mach-o/dyld.h>
142    #endif
143
144    //LAF
145    char * csvfilename="";
146    char * kernelname="";
147    //LAF

925        dim3 gridDim = config.grid_dim();
926        dim3 blockDim = config.block_dim();
927        printf("GPGPU-Sim PTX: pushing kernel \'%s\' to stream %u, gridDim=
928    (%u,%u,%u) blockDim = (%u,%u,%u) \n",
929                kname.c_str(), stream?stream->get_uid():0,
930    gridDim.x,gridDim.y,gridDim.z,blockDim.x,blockDim.y,blockDim.z );
931        //LAF
932            kernelname=const_cast<char*>(kname.c_str());
933        //LAF
934        stream_operation op(grid,g_ptx_sim_mode,stream);

1616            char * pfatbin = (char*) fatDeviceText->d;
1617            int offset = *((int*)(pfatbin+48));
1618            char * filename = (pfatbin+16+offset);
1619
1620            //LAF
1621            csvfilename =filename;
1622
1623            //LAF
```

# Appendix F: Matlab Signature Calculation Code

```
1   function [signature,max_vector] = calc_signature(InputMatrix)
2
3   signature=[0,0,0,0,0,0];
4   max_vector=[0,0,0,0,0,0];
5
6   peak = [0,0,0,0,0];
7   temp_peak=[0,0,0,0,0];
8
9   %%
10  %%%%  signature calc
11  for loop=1:30
12      actual=rem(loop,5);
13      if(actual==0)
14          actual=5;
15      end
16      aux=(InputMatrix(:,actual))';
17      aux = aux(~isnan(aux))';
18      [pks,locs,widths,proms] = findpeaks(aux,'MinPeakHeight',1);
19      for k=1:length(pks)
20          peak(actual)=peak(actual)+((pks(k))*((widths(k)*proms(k))/2));
21      end
22
23      if(actual==5)
24          temp_peak(1)=temp_peak(1)+peak(1);
25          temp_peak(2)=temp_peak(2)+peak(2);
26          temp_peak(3)=temp_peak(3)+peak(3);
27          temp_peak(4)=temp_peak(4)+peak(4);
28          temp_peak(5)=temp_peak(5)+peak(5);
29          peak = [0,0,0,0,0];
30      end
31  end
32
33  signature(1)=temp_peak(1);
34  signature(2)=temp_peak(2);
35  signature(3)=temp_peak(3);
36  signature(4)=temp_peak(4);
37  signature(5)=temp_peak(5);
38
39  %%variance coef calc
40  %%
41  requests=(InputMatrix(:,31))';
42  requests = requests(~isnan(requests))';
43
44  signature(6)=(std(requests)/mean(requests))*100;
45  %%
46  %%max calc
47
48  sub0= InputMatrix(:,1:5);
```

```
49    sub1= InputMatrix(:,6:10);
50    sub2= InputMatrix(:,11:15);
51    sub3= InputMatrix(:,16:20);
52    sub4= InputMatrix(:,21:25);
53    sub5= InputMatrix(:,26:30);
54
55    max_vector(1)=max(sub0(:));
56    max_vector(2)=max(sub1(:));
57    max_vector(3)=max(sub2(:));
58    max_vector(4)=max(sub3(:));
59    max_vector(5)=max(sub4(:));
60    max_vector(6)=max(sub5(:));
61    end
```

# Appendix G: Different Variable Selection

The final test was made using the 7 elements signature previously mentioned. This section presents other configuration of dimensions that were tested using different configuration of elements.

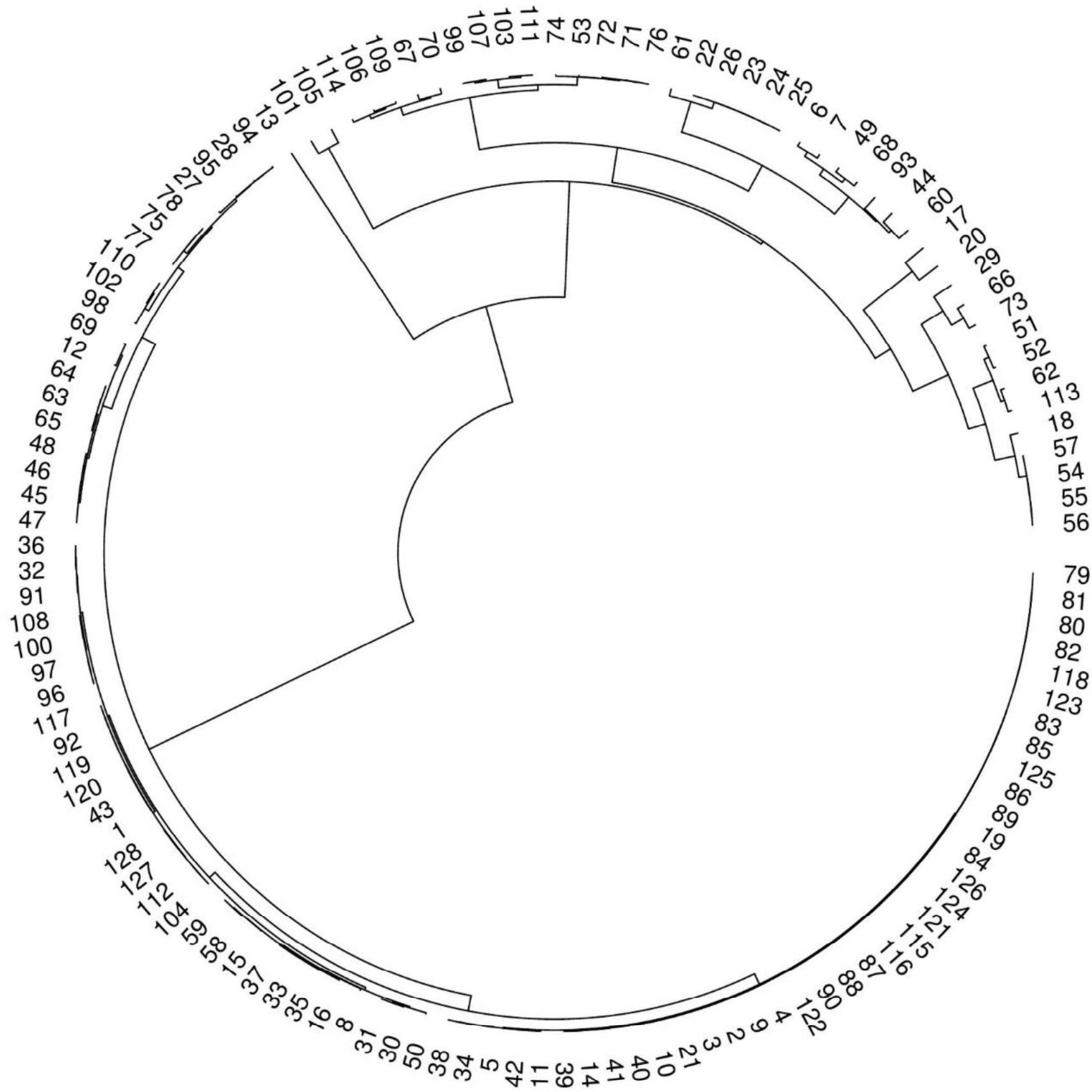



Figure G.1: Dendrogam using only  SPC and mean

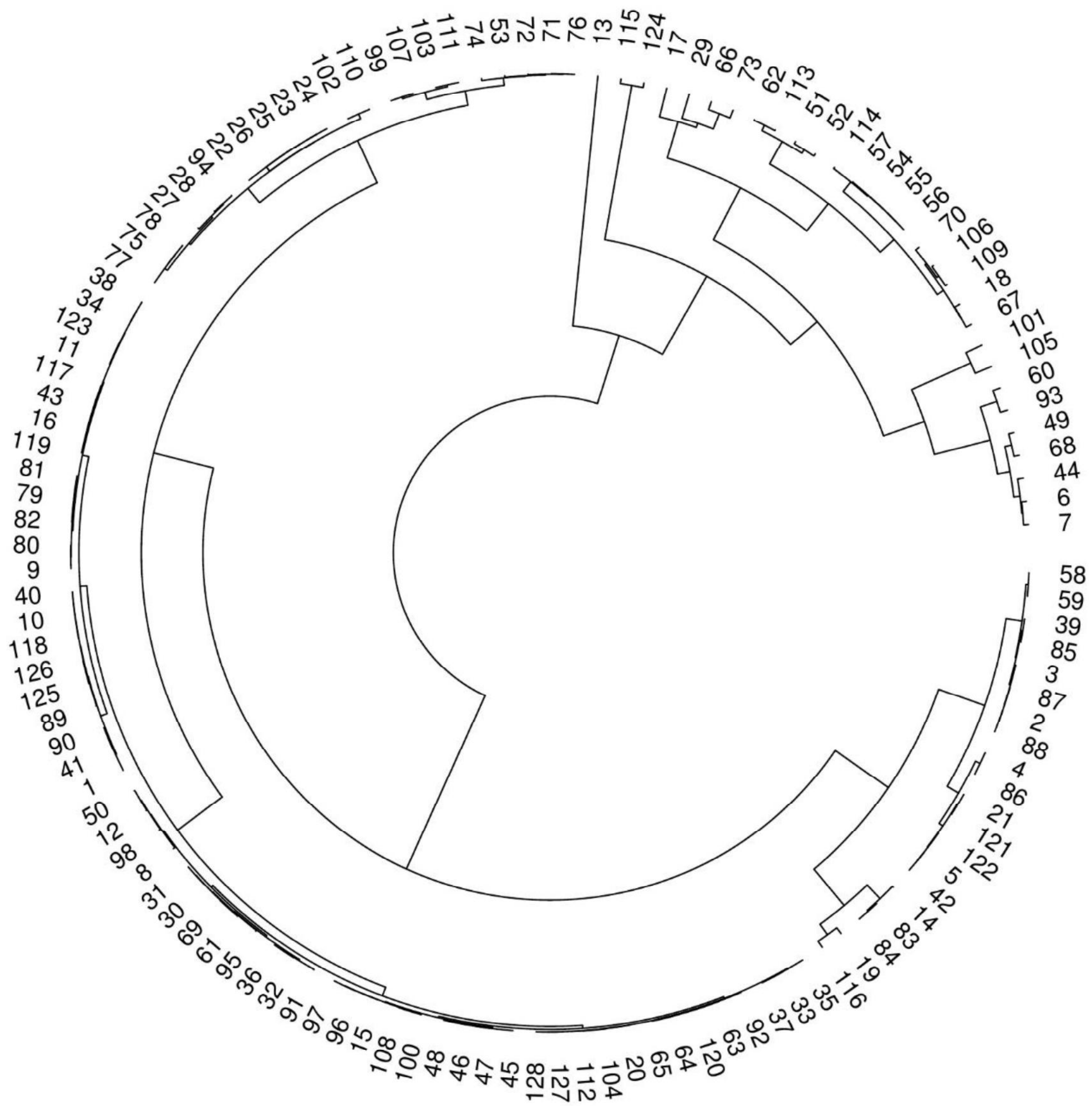



Figure G.2: Dendrogam using only SPC and coefficient of variation

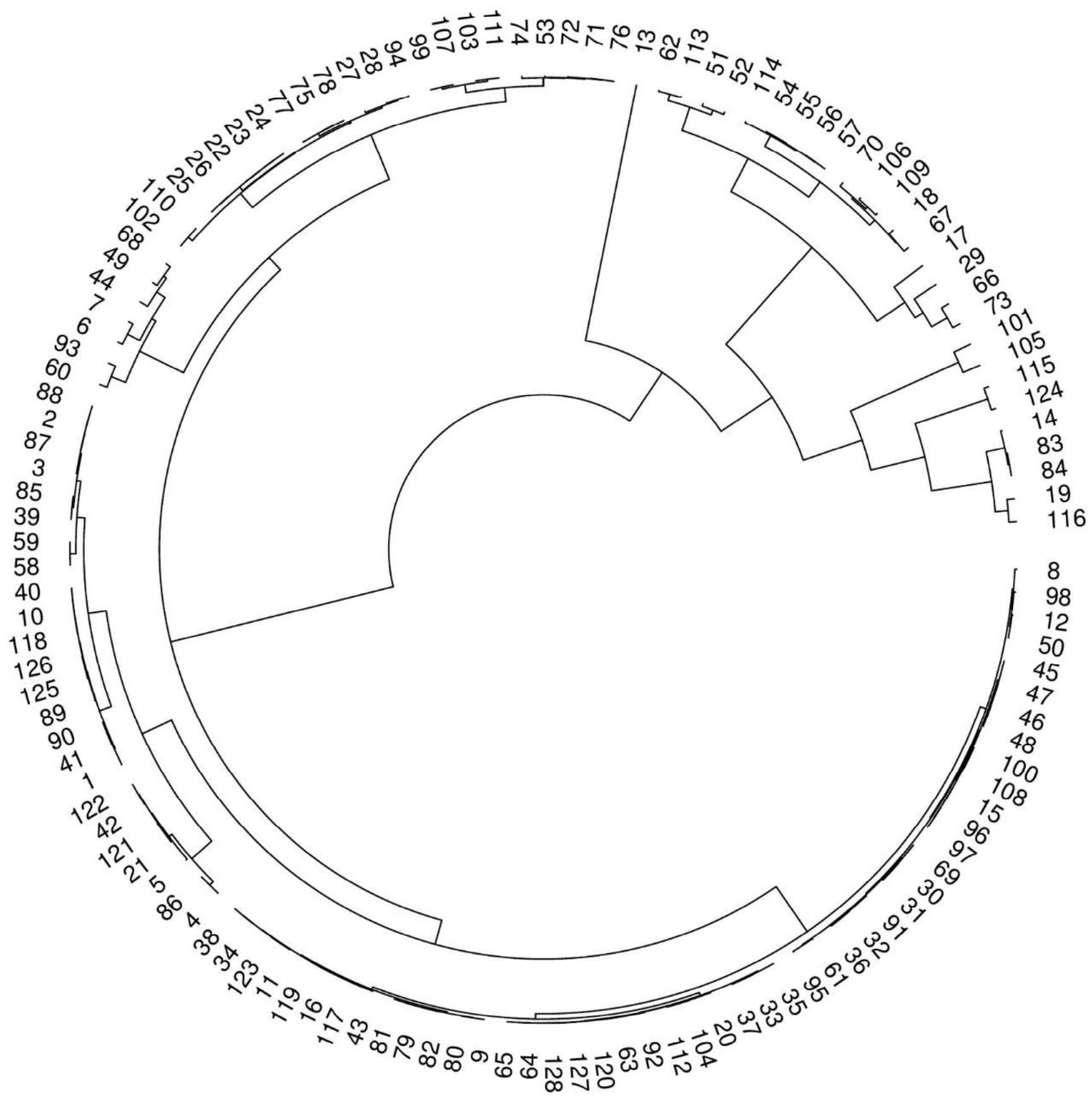



Figure G.3: Dendrogam using SPC, coefficient of variation and standard deviation

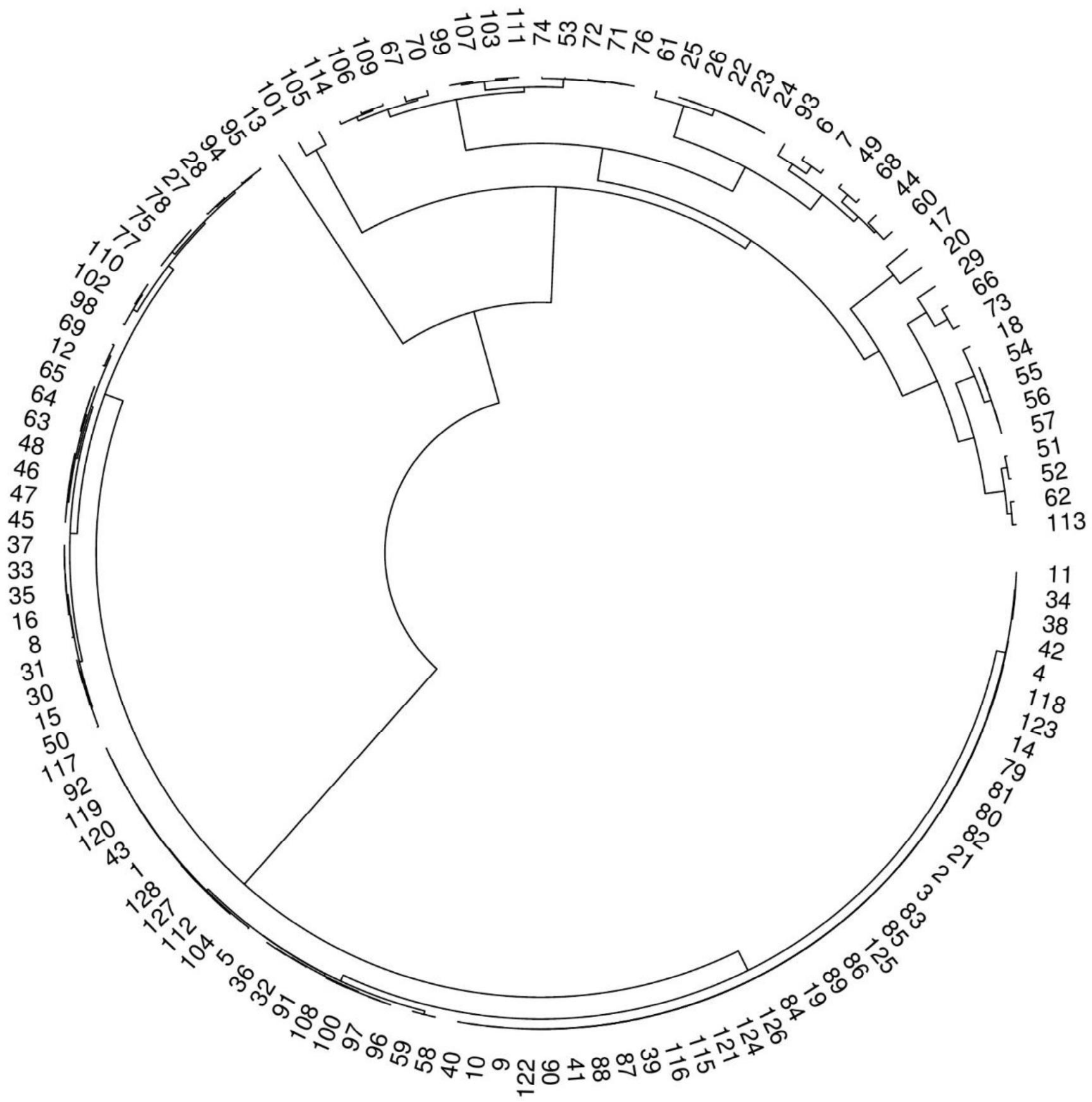



Figure G.4: Dendrogam using SPC mean and standard deviation

# Curriculum Vitae

*Luis Alberto Freire Bermudez* was born in Guayaquil, Ecuador on January 6th, 1993. He received the degree of Digital Systems Engineer from the Tecnológico de Monterrey Campus Toluca, Toluca México in December 2015. He attended a research visit at University of California, Irvine in 2018. He was accepted in the Master of Science in Electronic Engineering program in the Tecnológico de Monterrey, Escuela de Ingeniería y Ciencias, Campus Monterrey.

This document was typed in using Microsoft Word by *Luis Alberto Freire Bermudez*