National Chiao Tung University

EECS International Graduate Program

Thesis

記憶體追蹤方式在單指令多執行緒架構中資料分享程度之分析研究

A Memory Trace-Based Analysis for Data Sharing Degree in SIMT Architectures

Student: Luis Angel Garrido Platero
Advisor: Prof. Bo-Cheng Lai

June, 2013

以記憶體追蹤方式在單指令多執行緒架構中資料分享程度之分析研究

A Memory Trace-Based Analysis for Data Sharing Degree in SIMT Architectures

研　究　生：盧以斯　　Student: Luis Angel Garrido Platero

指導教授：賴伯承　　Advisor: Bo-Cheng Lai

國　立　交　通　大　學

EECS International Graduate Program

碩　士　論　文

A Thesis

Submitted to the EECS International Graduate Program

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

EECS

June 2013

Hsinchu, Taiwan, Republic of China

中華民國一Ｏ二年六月

以記憶體追蹤方式在單指令多執行緒架構中資料分享程度之分析研究

Student: Luis Angel Garrido Platero Advisors: Dr. Bo-Cheng Lai

EECS International Graduate Program
National Chiao Tung University

CHINESE ABSTRACT

在本論文中，我們透過量化單一指令多執行續程式中記憶體存取的資料分享程度進而分析應用程式的區域特性。此外，我們也提供了不同執行環境下之資料分享的視覺化方法。為了量化資料分享程度，本論文使用含有執行階段記憶體位置的記憶體追蹤以完成記憶體存取的資料分享程度分析。在此分析中，我們重新定義重複使用距離的概念以提供分析以及不同執行環境的需要。

A Memory Trace-based Analysis for Data Sharing Degree of SIMT Architectures

Student: Luis Angel Garrido Platero Advisors: Dr. Bo-Cheng Lai

EECS International Graduate Program
National Chiao Tung University

ENGLISH ABSTRACT

In this work, we address the problem of quantifying the data sharing degree of the memory access behavior within specific SIMT applications in order to quantify the locality characteristics of the application's workload. In addition, we also offer way to visualize the way the sharing patterns of the applications and the way they change under different models of runtime scenarios. For the purposes of quantifying the data sharing degree a memory trace is generated that contains information of the addresses accessed at a specific point of execution. Then, the information contained in the traces is used to perform the data sharing degree analysis of memory accesses. In this analysis, we have redefined the reuse distance concept in order to make it suitable to our analytical requirements, at the same time considering the particulars of the execution model previously mentioned.

TABLE OF CONTENTS

Chinese Abstract……………………………………………………………………………………..i

English Abstract..ii

Table of Contents………………………………………………………………………………iii

List of Tables…………………………………………………………………….....................iv

List of Figures………………………………………………………………………………....v

Symbols……………………………………………………………………………………...xvi

I. INTRODUCTION 1

II. OVERVIEW OF SIMT PROCESSORS 6

2.1 Hardware of GPU Architectures.. 6

2.2 Programming and execution abstractions of GPU 7

2.3 Memory Hierarchy of GPUs 8

III. LOCALITY ANALYSES IN CMP AND UNIPROCESSOR SYSTEMS 9

IV. DATA REUSE CHARACTERIZATION 12

4.1 Definition of the Data Reuse Degree.. 12

4.1 Definition of the Reuse Distance... 14

4.1.1 Traditional reuse distance analyses 15

4.1.2 Reuse Distance for SIMT Processors 18

V. ANALYSIS METHODOLOGY FOR
DATA REUSE CHARACTERIZATION 21

VI. SCENARIOS FOR DATA REUSE CHARACTERIZATION 26

5.1 Infinite resources, thread blocks are modeled as executing sequentially 27

5.2 Infinite resources, analysis within each thread block 27

5.3 Infinite resources, all thread are modeled as executing in parallel...................... 28

5.4 Infinite resources, a number ‘*K*’ of thread blocks modeled as
executing in parallel 29

5.5 Limited resources, '*K*' block modeled as executing in parallel, within core cluster analysis 31

5.6 Limited resources, '*K*' blocks modeled as executing in parallel, inter-core cluster 33

VII. EXPERIMENTATION FRAMEWORK 35

6.1 Trace Generation Stage 35

6.2 Reference Stream Analysis Stage 36

6.2.1 Model for Thread Blocks 36

6.2.2 Scheduling Policies 39

6.2.3 Core cluster modeling 43

6.2.4 Merging of reference streams 44

6.2.5 Adjusting the position index of Mis 48

6.2.6 Locality Analyzer Architecture: Putting it all together 50

VIII. APPLICATION OPTIMIZATION 52

8.1 Thread Mapping Methodology 52

IX. EXPERIMENTAL RESULTS 55

9.1 Data Reuse Characteristic with serialized blocks and on a per block basis 55

9.2 Data Reuse Characteristic with varying parallelism capabilities 64

9.3 Data Reuse Characteristic with limitations of SIMT Architectures 73

9.4 Data Reuse Characteristic when applying code optimizations 86

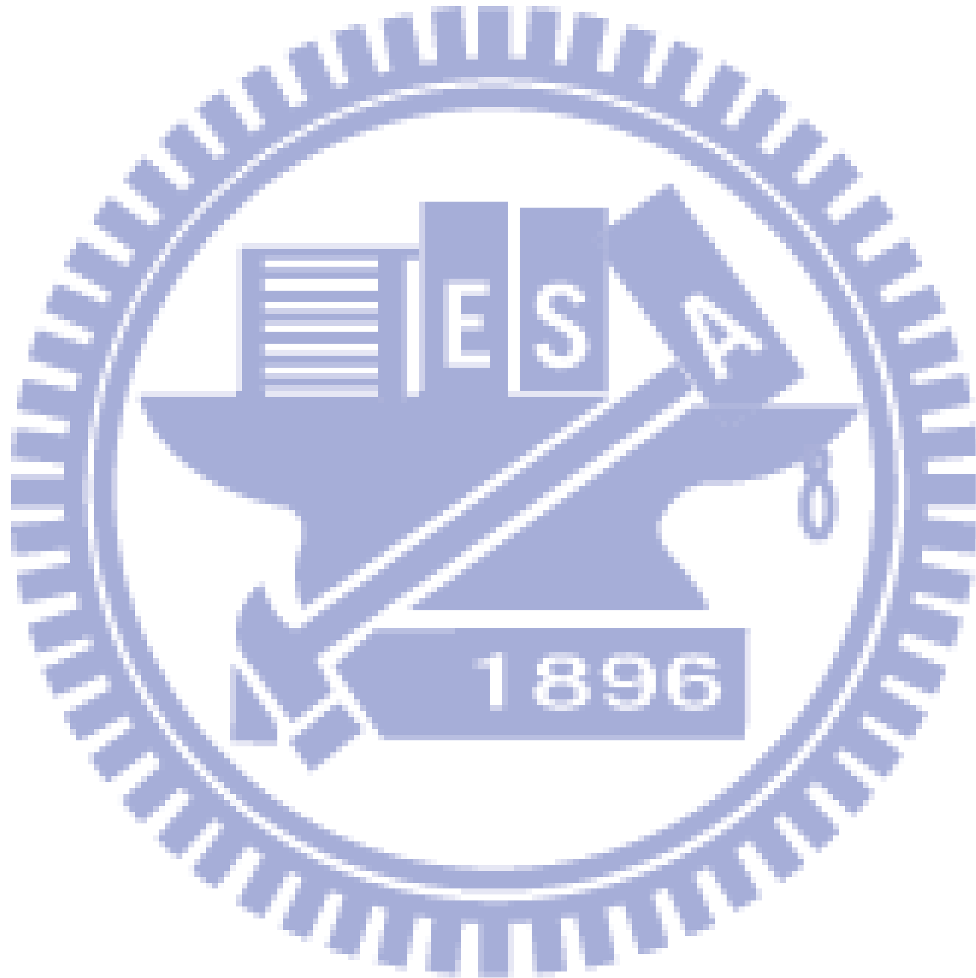
X. RELATED WORK 98

XI. CONCLUSIONS 101

XII. REFERENCES 103

LIST OF TABLES

Table 1: APPLICATIONS USED FOR EXPERIMENTATION



LIST OF FIGURES

Figure 1: Benefits obtained when taking advantage of the data reuse in SIMT applications. (a) DRAM memory transactions with and without coalescing. The first two cases from the top illustrate the case for coalescing. The last case shows the case when coalescing is not possible. (b) Illustration of the contention effect in a CMP. 3

Figure 2: Diagram of a core cluster based on NVIDIA's Kepler GeForce GTX 680 GPU. (a) The core cluster with its internal hardware modules. (b) Illustration of the thread hierarchy in the SIMT programming model. 6

Figure 3: Reuse distance concept and memory instruction reuse behavior in SIMT programs. (a) Sample measurement of reuse distance in traditional multiprocessors. (b) Reuse distance behavior in SIMT architectures. 10

Figure 4: A sample stream of memory instructions. The instruction array appears in the left column, the address array is presented in the middle column and the Reuse Degree for memory instructions '*i*'→'*i+2*' and '*i+2*'→'*i+4*' appears in the right column. 13

Figure 5: Reuse distance analysis as applied in uniprocessor systems. (a) Sample memory trace. (b) Changes of the state of the stack as memory instructions are issued. 15

Figure 6: Reuse distance analysis in CMP systems with private memory subsystem. (a) Sample reference stream. (b) Stacks for the private memory subsystem. 16

Figure 7: Reuse distance analysis in CMPs with shared memory subsystem. (a) Sample reference stream. (b) Stacks for each memory subsystem. 17

Figure 8: Data reuse degree in a sample reference stream of a SIMT processor with '*N*' processing cores. (a) Sample reference stream of a SIMT processor. (b) Data Reuse Degree for memory instructions '*i*'→'*i+k*' and '*i+2*'→'*i+2+k*'. 19

Figure 9: Limitations when performing the baseline methodology for reuse distance analysis on SIMT processors. (a) Subset of the reference stream as it appears in Figure 7. (b) Possible ways to arrange the accessed addresses on the stacks. 22

Figure 10: Flow chart of the data reuse characterization methodology. This flow chart shows all the steps when performing the analysis over a reference stream. 23

Figure 11: Data reuse characterization. (a) Scenario 1. (b) Scenario 2. In these two scenarios, the SM is assumed ideal, as represented by the infinite signs in the figures above. 27

Figure 12: Scenario 3 of data reuse characterization assuming ideal core clusters. (a) Illustrates the way blocks are intended to be executed. (b) Illustrates the Aggregate Block that results from merging the streams of all CTAs... 28

Figure 13: Scenario 4 of the reuse degree characterization. Only '*K*' blocks are modeled as executing concurrently. This is equivalent as having 'K' ideal core clusters, each one executing one block at the time.. 29

Figure 14: (a) An ideal core cluster executing the reference streams of '*K*' parallel blocks. (b) The streams of parallel blocks merged into a series of aggregate blocks, executed in sequence. 30

Figure 15: Scenario 5 of the data reuse characterization analysis. Each core cluster has now a finite number of load/store instructions. The analysis is performed within each core cluster. 32

Figure 16: Scenario 6 of the data reuse characterization. The analysis is performed over the aggregate of the reference streams in every core cluster. The number of load/store units is the total sum across the core clusters. 33

Figure 17: Formation of Block objects within the Locality Analyzer. The block objects contain their individual reference stream which consists on a series of ordered MIs. Each MI has access information of its own... 37

Figure 18: Block scheduling module and block scheduling flow (a) Block scheduling module assigning blocks to a system with one core cluster. (b) Block scheduling module assigning blocks to a system with multiple core clusters 40

Figure 19: After assigning blocks to the core clusters. The blocks are queued in each cluster, and are also inserted into the '*currBlocks*' structure, which represents the arrays of concurrent slots.. 42

Figure 20: Merging of reference streams from multiple blocks in the arrays of concurrent slots. Scenarios 3, 4, and 6 are the only ones that employ this procedure....................................... 45

Figure 21: The resulting reference stream of the aggregate blocks. It is possible to compare this with the streams shown in Figure 19. Notice how the 'Addresses' and 'Threads' fields are augmented.. 48

Figure 22: Modifications of the position reference index within the reference stream. The value of the index of the MIs in streams other than the first will depend on the stream length of the streams before the current one. 49

Figure 23: Block diagram of the architecture of the Locality Analyzer. This is a very general and simplified version of our framework 50

Figure 24: Data reuse characteristic when modeling block execution sequentially for *sta*. (a) Full reuse characteristic. (b) Showing the reuse characteristic for the range *RD={1, 400}*. (c) Showing the reuse characteristic for the range *RD={1, 100}*. Notice the particular patterns. . 56

Figure 25: Data reuse characteristic when modeling block execution sequentially for *gsim*. (a) Full reuse characteristic. (b) Showing the reuse characteristic for the range *RD={1, 400}*. (c) Showing the reuse characteristic for the range *RD={1, 100}*.. 56

Figure 26: Data reuse characteristic when modeling block execution sequentially for *bfs*. (a) Full reuse characteristic. (b) Showing the reuse characteristic for the range *RD={1, 400}*. (c) Showing the reuse characteristic for the range *RD={1, 100}*.. 57

Figure 27: Data reuse characteristic when modeling block execution sequentially for *vectoradd*. (a) Full reuse characteristic. (b) Showing the reuse characteristic for the range *RD={1, 400}*. (c) Showing the reuse characteristic for the range *RD={1, 100}*. 57

Figure 28: Source code for the kernels of *bfs* (a) and *sta* (b). 57

Figure 29: Data reuse characteristic on a per block basis for *sta*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1. 58

Figure 30: Data reuse characteristic on a per block basis for *gsim*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1. 58

Figure 31: Data reuse characteristic on a per block basis for *bfs*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1. 59

Figure 32: Data reuse characteristic on a per block basis for *vectoradd*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1. 59

Figure 33: Data reuse characteristic on a per block basis for *nbf*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1. 59

Figure 34: Data reuse characteristic on a per block basis for *moldyn*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1. 60

Figure 35: Data reuse characteristic on a per block basis for *irreg*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1. 60

Figure 36: Data reuse characteristic on a per block basis for *euler*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1. 60

Figure 37: Data reuse characteristic when modeling block execution when all blocks run in parallel for *sta*. 64

Figure 38: Data reuse characteristic when modeling block execution when all blocks run in parallel for *gsim*. 65

Figure 39: Data reuse characteristic when modeling block execution when all blocks run in parallel for *bfs*. 65

Figure 40: Data reuse characteristic when all blocks of *vectoradd* are modeled as executing in parallel. 65

Figure 41: Data reuse characteristic when all blocks of *nbf* are modeled as executing in parallel. 66

Figure 42: Data reuse characteristic when all blocks of *moldyn* are modeled as executing in parallel. 66

Figure 43: Data reuse characteristic when all blocks of *irreg* are modeled as executing in parallel. 66

Figure 44: Data reuse characteristic when all blocks of *euler* are modeled as executing in parallel. 67

Figure 45: Data reuse characteristic when only '*K*' blocks of *sta* are modeled as executing in parallel. (a) Data reuse characteristic for *K=2*. (b) Data reuse characteristic for *K=4*. (c) Data reuse characteristic for *K=8*. (d) Data reuse characteristic for *K=16*. 68

Figure 46: Data reuse characteristic when only '*K*' blocks of *gsim* are modeled as executing in parallel. (a) Data reuse characteristic for *K=2*. (b) Data reuse characteristic for *K=4*. (c) Data reuse characteristic for *K=8*. (d) Data reuse characteristic for *K=16*. 68

Figure 47: Data reuse characteristic when only '*K*' blocks of *bfs* are modeled as executing in parallel. (a) Data reuse characteristic for *K=2*. (b) Data reuse characteristic for *K=4*. (c) Data reuse characteristic for *K=8*. (d) Data reuse characteristic for *K=16*. 69

Figure 48: Data reuse characteristic when only '*K*' blocks of *vectoradd* are modeled as executing in parallel. (a) Data reuse characteristic for *K=2*. (b) Data reuse characteristic for *K=4*. (c) Data reuse characteristic for *K=8*. (d) Data reuse characteristic for *K=16*. 69

Figure 49: Data reuse characteristic when only 'K' blocks of *nbf* are modeled as executing in parallel. (a) Data reuse characteristic for K=2. (b) Data reuse characteristic for K=4. (c) Data reuse characteristic for K=8. (d) Data reuse characteristic for *K=16*...................................... 70

Figure 50: Data reuse characteristic when only 'K' blocks of *moldyn* are modeled as executing in parallel. (a) Data reuse characteristic for K=2. The reuse domain for this case is actually RD={1,124030}. The tool used to make the graphs could not display it properly. (b) Data reuse characteristic for K=4. (c) Data reuse characteristic for K=8. (d) Data reuse characteristic for *K=16*. 70

Figure 51: Data reuse characteristic when only 'K' blocks of *irreg* are modeled as executing in parallel. (a) Data reuse characteristic for K=2. (b) Data reuse characteristic for K=4. (c) Data reuse characteristic for K=8. (d) Data reuse characteristic.. 71

Figure 52: Data reuse characteristic when only 'K' blocks of *euler* are modeled as executing in parallel. (a) Data reuse characteristic for K=2. The reuse domain for this case is actually RD={1,66623}. The tool used to make the graphs could not display it properly. (b) Data reuse characteristic for K=4. (c) Data reuse characteristic for K=8. (d) Data reuse characteristic ... 71

Figure 53: Data reuse characteristic resulting when only *K=2* blocks of *sta* are modeled as executing in parallel. (a) Data reuse characteristic presented for reuse distance range *RD={1, 20}*. (b) Data reuse characteristic presented for reuse distance range *RD={21, 40}*. (c) Data reuse characteristic presented for reuse distance range *RD={41, 60}*.................................... 73

Figure 54: Data reuse characteristic resulting when only *K=16* blocks of *sta* are modeled as executing in parallel. (a) Data reuse characteristic presented for reuse distance range *RD={1, 20}*. (b) Data reuse characteristic presented for reuse distance range *RD={21, 40}*. (c) Data reuse characteristic presented for reuse distance range *RD={41, 60}*.................................... 73

Figure 55: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *sta*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster. 74

Figure 56: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *gsim*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster. 74

Figure 57: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *bfs*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster. 74

Figure 58: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *vectoradd*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster. 75

Figure 59: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *nbf*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster. 75

Figure 60: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *irreg*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster. 76

Figure 61: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *euler*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster. 76

Figure 62: Data reuse characteristic in reuse distance range *RD={0, 100}* of the aggregate reference stream of all core clusters with varying number of load/store units for *sta*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster. 77

Figure 63: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *sta*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units... 79

Figure 64: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *gsim*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units... 80

Figure 65: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *bfs*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units... 81

Figure 66: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *vectoradd*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units... 82

Figure 67: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *nbf*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units... 83

Figure 68: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *moldyn*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units... 84

Figure 69: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *irreg*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units. 85

Figure 70: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *euler*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units. 86

Figure 71: Data reuse characteristic for block 0 of *sta* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 17. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 802. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 802. 87

Figure 72: Data reuse characteristic for block 0 of *gsim* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 400. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 795. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 794. 87

Figure 73: Data reuse characteristic for block 0 of *bfs* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 17. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 802 (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 802. 88

Figure 74: Data reuse characteristic for block 0 of *nbf* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 44458. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 60346 (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 60576. 88

Figure 75: Data reuse characteristic for block 0 of *moldyn* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 352556. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 446146. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 447102. 89

Figure 76: Data reuse characteristic for block 0 of *irreg* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 47309. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 53448. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 43212. 90

Figure 77: Data reuse characteristic for block 0 of *euler* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 59694. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 81755. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 82204. 90

Figure 78: Data reuse characteristic for all blocks running in parallel of *sta* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is -10216. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 14229. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 11940. ... 92

Figure 79: Data reuse characteristic for all blocks running in parallel of *gsim* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 8236. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 21363. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 17129. ... 93

Figure 80: Data reuse characteristic for all blocks running in parallel of *bfs* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp

clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is -10812. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 13361. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 11324.... 93

Figure 81: Data reuse characteristic for all blocks running in parallel of *nbf* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 6053. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 7400. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 9138...... 94

Figure 82: Data reuse characteristic for all blocks running in parallel of *moldyn* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is -3373236. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is -5170268. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is -5791542.. 95

Figure 83: Data reuse characteristic for all blocks running in parallel of *irreg* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is -4445. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is -4402. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is -4647. ... 95

Figure 84: Data reuse characteristic for all blocks running in parallel of *euler* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 8904. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 7834. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 9172...... 96

SYMBOLS

DS: Data Reuse Degree, Data Sharing Degree.

RD: Reuse Distance.

R: short for Reuse Distance in Figures.

MI: memory instruction

M: multiplicity of an address in a memory instruction

$\mathbf{X}_{i \to j}$: common address array between memory instructions 'i' and 'j'.

$\mathbf{X}_{i}$: address array of memory instruction 'i'.

I. INTRODUCTION

The last decade has seen an increase in the processing demand in the different computing markets [1]. This has made necessary the introduction of novel computer architectures to satisfy the exponentially increasing processing needs of the end users. As a consequence, heterogeneous computing systems [2] have risen as commercially available solutions. These systems rely on one or more processing accelerators that are able to perform certain tasks within the users' applications faster and more efficiently. SIMT architectures are one of the most common many-core/multi-threaded processing accelerators. SIMT stands for *Single Instruction – Multiple Threads*. These processors are able to handle a relatively large amount of execution contexts simultaneously. Within this scope, GPUs are the most popular and widely used.

The current trend is to utilize these heterogeneous computing systems for a wider range of scientific computing applications and other general purpose tasks. To do this, it is necessary to understand the particularities of the processing accelerator. Thus, programmers are required to consider key architecture details at the software design stage. In addition, a thorough understanding of the application's characteristics and its interaction with the architecture is necessary to fully exploit the processing power of the accelerators. This is particularly delicate for SIMT processors.

The performance of an application executing on SIMT architectures, such as GPUs, is significantly dependent on its locality characteristic, resource utilization, control flow behavior, among other things [3]. The locality characteristic is dependent on the memory access patterns of the application. Considering these patterns and the details of the underlying memory sub-system is critical to boost performance. This is because the memory sub-system is the principal performance bottleneck [3]. Applications for SIMT architectures are extremely sensitive to memory utilization resources.

Many efforts already exist that have characterized the applications running on SIMT architectures [4, 5, 6]. Most of these works define a set of metrics (percentage of branch divergence, branch predictability, dynamic instructions, memory intensity, etc.), and observe the values of the metrics produced by each workload after conducting a series of simulations over real GPUs or simulators [7]. There have also been efforts to characterize the locality of

applications [8]. These works carefully explore the relationship between the execution model of the architecture and the data sharing of the application [9]. Such works are able to leverage the data sharing of the thread at different levels of the thread hierarchy in the SIMT architecture, and provide guidelines based on this information to improve performance. In this work, we use the terms data sharing between the threads and data reuse between the threads interchangeably.

The data reuse behavior of applications deserves particular attention. As Figure 1(a) shows, one of the benefits of taking advantage of the data reuse is the increase in memory coalescing. When threads request data from the off-chip memory, their accesses are said to be coalesced when many memory requests can be served in one single off-chip memory transaction. This happens when accesses are to contiguous or identical addresses. Memory coalescing is not possible when the memory accesses are too scattered. This makes either necessary additional off-chip memory transactions or increases the latency of transactions if caching is present. Thus, performance is reduced.

Another benefit is the avoidance of contention, illustrated in Figure 1(b). Contention occurs when data is evicted between two sub-sequent *requests* to the same data. In Figure 1(b), an example is presented for a CMP. First, processor *P0* requests a data from memory and uses it. Then, processor *P1* requests data of its own that causes the eviction of the previous data requested by *P0*. If *P0* requests that data again, *P0* will be stalled fetching the same data to the off-chip memory a second time. If these series of events repeat frequently during the application’s execution, then it is said that contention is present. Contention harms performance significantly, since the latency required to fetch data to off-chip memory is an order of magnitude higher than fetching data from on-chip caches.

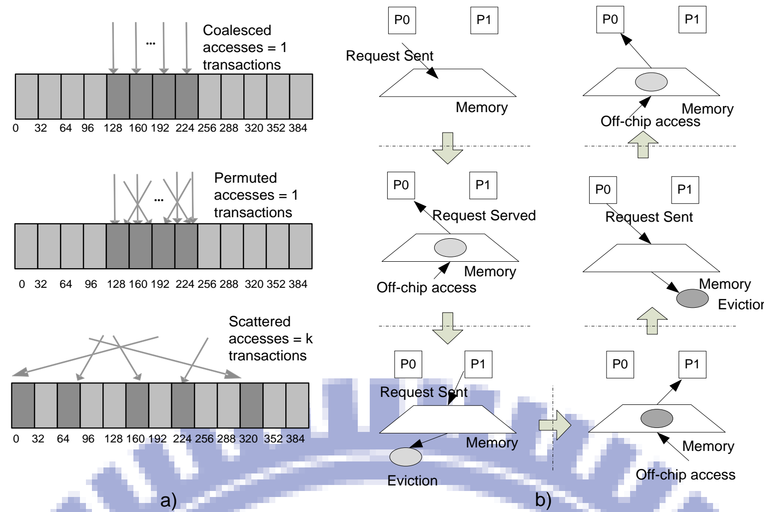


Figure 1: Benefits obtained when taking advantage of the data reuse in SIMT applications. (a) DRAM memory transactions with and without coalescing. The first two cases from the top illustrate the case for coalescing. The last case shows the case when coalescing is not possible. (b) Illustration of the contention effect in a CMP.

The impact of the data reuse over performance is significant [8, 9, 10]. For the case of SIMT processors, there's a need for *architecture-agnostic* analyses to assess qualitatively and quantitatively the locality characteristics of applications, in particular the data reuse behavior. Modeling the inherent large amount of parallelism in SIMT applications and its impact on the data reuse behavior of the applications is the main motivation behind performing such analyses. The existing methodologies to perform locality analyses used for applications running on CMP systems, such as the reuse distance analysis, are not appropriate for SIMT applications. The main reason for this limitation is the difference in the execution model.

Reuse distance analyses on CMP systems consider implementation details of the architecture in order to maintain accuracy [11]. In these analyses, locality is measured from the perspective of the memory subsystem, keeping track of the addresses accessed. These analyses model the effects of thread interference and amount of processor cores, which defines the total amount of threads running simultaneously. However, the locality measurements obtained with this methodology are heavily dependent on the configuration of the on-chip memory subsystem, and are affected by factors such as the type of task scheduling and allocation. The architectural agnosticism is sacrificed, but these analyses are still very valuable for memory subsystem design, to predict cache miss rates and estimate performance

When applying the previously described methodologies, the locality measurements are not solely of the application, but are of the application interacting with a memory subsystem that has specific characteristics. This methodology becomes inappropriate for SIMT processors, since it does not consider the particular execution model of the latter and does not consider its inherent large parallelism. Also, the memory subsystem in SIMT processors has different characteristics than their CMP counterparts, which imposes the need to develop better suited analysis methodologies.

In order to quantify the locality characteristic of SIMT applications in an integral way, it is necessary to abstract the analytical model from the implementation details and practical limitations of SIMT processors, and perform the analysis as closer to the application itself as possible. Analyses performed under such conditions would show the locality characteristic particular to an application in a self-contained, abstract and truly architecture-agnostic way. This would allow us to measure, as isolated as possible from implementation details, the changes of the locality characteristic under different runtime scenarios and optimizaitons. Once this has been quantified, the locality can then be measured in relation to other factors of the SIMT execution model (scheduling, allocation, pipeline length, etc.) and the limitations of commercial architectures.

In this work, we develop a methodology to analyze and quantify, while offering a graphical representation, of the data reuse behavior of SIMT applications under different execution conditions. For the characterization of the data reuse, we define a new metric: the data reuse degree, and also, we redefine the reuse distance concept in order to employ it in our analyses. We measure the reuse degree in the reuse distance domain of an application's kernel, assessing how significant the data reuse is at different segments of the application. We also obtain the data reuse characteristic for different kernels when modeling different abstractions of parallelism, which gives a clear idea on the manageable locality as processing resources are constraint.

The contributions of this work are as follows: 1) we provide a new analytical model for the analysis, quantification and to graphically represent the data reuse behavior of SIMT applications that is solely application dependent and architecture-agnostic, 2) provide a methodology that captures the data reuse behavior of SIMT applications under different types of parallelism constraints, from an ideal case where parallelism capabilities are infinite down

to more realistic scenarios, 3) we provide a new way to identify an application's access patterns, embodied in its data reuse characteristic, 4) we show the changes on the data reuse characteristic when coding optimizations are performed, 5) develop a flexible framework that enables to analyze the effects of that certain implementation details of SIMT architectures (scheduling, allocation, number of core clusters) have over the reuse characteristic.

This thesis is organized as follows. Chapter 2 gives an overview of SIMT processors. It explains the abstractions of the programming and execution models, and describes very briefly the architecture of a commercial SIMT processor. Chapter 3 explains current state-of-the-art locality analyses. Their limitations are explained when trying to use them as such when analyzing applications SIMT processors. Chapter 4 develops our new model for characterizing the data reuse, and formally defines the data reuse degree and the reuse distance. Chapter 5 explains the methodology used to perform the analyses. Chapter 6 details the different conditions under which the data reuse characteristic is obtained. We vary the amount of available parallelism, and a different reuse characteristic is obtained for each case. Chapter 7 explains with luxury of detail the framework developed to perform the analysis. Mostly programmed in C++, we show the algorithms it has and the elements that were modeled. Chapter 8 describes the coding optimization techniques performed over the benchmarks we use for our experiments. These optimization techniques are taken from [9], and are used in our experiments to observe the change on the reuse characteristic after applying them. Chapter 9 shows our experimental results. In Chapter 10, the related work is presented. Chapter 11 concludes this work.

II. OVERVIEW OF SIMT PROCESSORS

This section presents general background on SIMT processors. We take as our main reference current state-of-the-art GPU architectures. Thus, we provide general information on their hardware specifications and programming abstractions.

2.1 Hardware of GPU Architectures

Figure 2(a) presents a diagram of the architecture of a core cluster or, as NVIDIA calls it, a Streaming Multiprocessor (SM). The diagram presented is based on NVIDIA's Kepler GeForce GTX 680 GPU [12]. In this figure, the elements with the subscript "Core" represent the CUDA cores, which are the basic processing units inside a GPU. Core cluster are groups of these small cores.

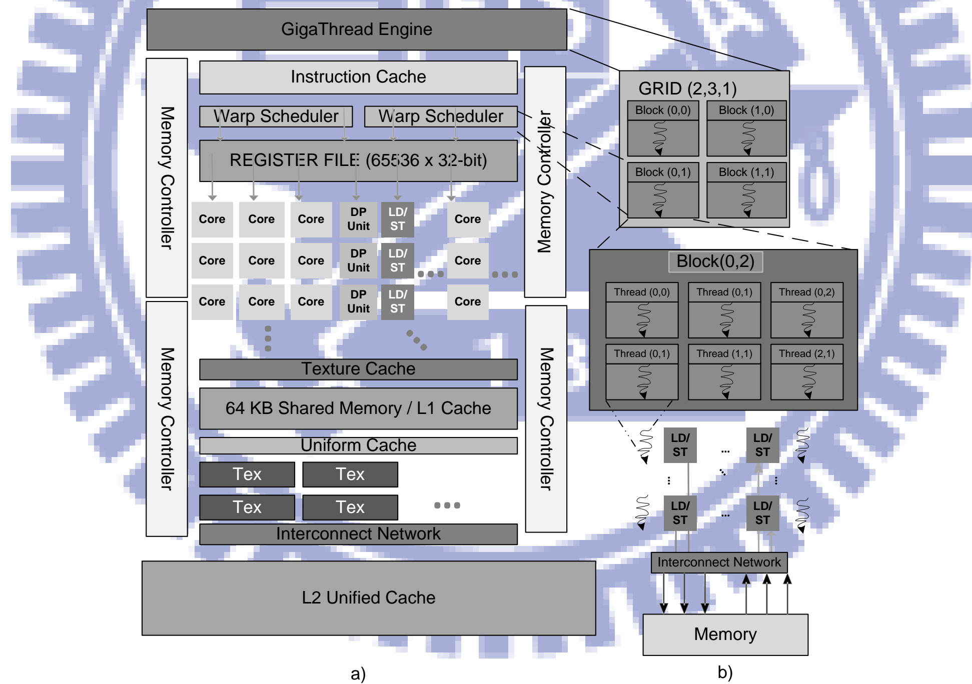


Figure 2: Diagram of a core cluster based on NVIDIA's Kepler GeForce GTX 680 GPU. (a) The core cluster with its internal hardware modules. (b) Illustration of the thread hierarchy in the SIMT programming model.

The number of cores in each cluster varies depending on the family of the GPU, but they are usually grouped by numbers of powers of 2. In the case of the GTX 680, there are 8 core clusters, arranged in groups of two, forming 4 separate groups. The task allocation to each

core cluster is handled by a thread block scheduler, which appears as the GigaThread Engine in Figure 2. This module issues a group of threads to each cluster based on a task allocation policy.

Each cluster has private caches that only the threads executing within it can access. Figure 2(a) also shows an L2 unified cache. This L2 cache is shared by all the threads running in all core clusters present in the GPU. Four memory controllers handle the access to the off-chip memory, which perform memory scheduling and coalescing techniques.

Every GPU has a PCI Express interface which is the bus that connects the GPU device to its CPU host. It is the CPU that launches the execution of applications in the GPU and transfers all the data to the GPU memory. Recent generations of GPUs are able to initiate tasks created autonomously [12]. The CPU offloads work into the GPU in order to accelerate the execution of highly parallel portions of applications, leveraging the latter's processing power

In Figure 2(a), there's also an array of texture units, a texture cache, a configurable shared cache and L1 cache, a uniform cache (for constant variables) and an interconnection network. The latter provides an interface for the core clusters to move data to and fro the L2 unified cache and the off-chip memory. It is important to stress the fact that there are no coherence or consistency models implemented in the programming model of the GPUs [13].

2.2 Programming and execution abstractions of GPU

In GPUs, threads are the smallest unit that can be executed. These are grouped obeying a hierarchical scheme that facilitates the task allocation from core clusters down to each individual core. Tasks issued to a GPU for execution are represented as a conglomerate of threads grouped into grids consisting on thread blocks, which are further divided into smaller groups of threads called warps [13]. This outlines the thread hierarchy inherent to the runtime model of the GPU. Figure 2(b) presents the thread hierarchy as previously described.

Each warp inside a block can have up to 32 threads in current state-of-the-art NVIDIA GPUs. The number 32 is chosen because it facilitates the management of the memory accesses by the memory subsystem [13]. Each warp of 32 threads executes in lockstep, which means that they execute the same instruction over different portion of data. The instructions they execute are the ones conforming the kernel code. Each thread executes the kernel code, but each thread

works over totally or partially mutually exclusive subsets of the data. Because of this fact, it is said that GPUs apply an SIMT execution model.

The threads can be arranged in multidimensional arrays, and so they are grouped into warps, which conform the blocks, as mentioned before. Each warp has a warp ID. Inside these warps, each thread also possesses a unique ID, which becomes useful to associate it to the data portion that it uses.

2.3 Memory Hierarchy of GPUs

The GPUs memory hierarchy is very particular, and it is somewhat suited to fit the needs of the programming model just described in the previous section. The memory hierarchy of the GPUs has 6 different memory spaces: register, local, shared, global, constant and texture.

The different spaces serve different purposes. The constant memory space is read-only memory used to store constants, parameters and data types declared as un-modifiable by the CUDA programming model. The texture cache is used to store texture and surface [13] data in a non-inclusive way: texture data is EXCLUSIVELY stored in the texture cache. The registers are assigned to each thread so these can store operands and perform calculations. The local memory space is a portion of the memory assigned to each individual thread, to which it can write or read information as the computation progresses. Also, it can use this space to spill registers when exceeding the register quote. The lifetime of this memory space lasts as long as the thread is active. The shared memory space can be accessed by all threads within a block and it is managed explicitly by the programmer. This space expires from the memory as soon as the block finishes execution in the SM. The global, constant and texture memory spaces remain in place even after the kernel has finished execution, or other kernels are launched into the GPU.

Understanding the details of the memory hierarchy of these processors is fundamental to comprehend the complexity of the locality characteristics. However, as it will be explained, the locality behavior of applications depends on multiple factors, starting from the resource and parallelism availability. This is the central point of the analytical models proposed in this work.

III. LOCALITY ANALYSES IN CMP AND UNIPROCESSOR SYSTEMS

SIMT architectures can execute a large amount of threads concurrently when compared to more conventional processor systems (CMPs, uniprocessors), and memory accesses are also managed in a different way. The threads in SIMT processors are highly symmetrical performing the same, or nearly the same, operations over different portions of data. The locality behavior in SIMT processors is closely related to how threads are grouped, allocated and identified at runtime [8]. The relationship between the threads and the data used by them is intrinsic to the programming model of these systems, and it is the most significant consideration at the software design stage. Additionally, there are different on-chip memory spaces in SIMT architectures that are used consciously by the programmer to store specific data types and data structures. This allows for a better administration of the memory resources depending on the particular requirements of an application.

In more conventional processor systems, the case is dramatically different. In these processors, the threads that enter execution do not necessarily present such similarities in the instructions they execute and their corresponding data sets. The amount of threads that can execute simultaneously is much smaller when compared to SIMT processors. The main reason for this is the significant difference in the amount of resources available for computation in both architectures, which are significantly higher in SIMT processors. Moreover, due to the asymmetry frequently common in threads running on conventional processor systems, control flow behavior becomes more complex. This limits the amount of parallelism available that can be leveraged to boost performance.

These conventional processors do not offer to programmers the same flexibility to manage on-chip memory resources that SIMT processors do. This is so because in the former, there is a fairly uniform and general purpose memory space, with relatively large capacity. In this case, memory allocation, replacement and fetching are managed by the memory hardware. This is in stark contrast with SIMT processors, where the memory spaces are more diverse, tailored for specific uses. Programmers can instruct the hardware which data to cache or not, or to allocate it in specific *memory spaces* depending on the characteristics of the data. Thus, the configuration of the memory subsystem and its utilization is significantly more complex in SIMT processors.

When data is specifically allocated by the programmer, it is done depending on the specifics of the applications access patterns, and the capabilities of the specific architecture. When accesses are too scattered, for example, caching harms performance [8], since a lot of data loaded to the on-chip cache is not used. Therefore, programmers need this flexibility to tune their applications to the capabilities of a specific SIMT processor.

All the factors previously described make the locality behavior of applications more complex for the case of SIMT architecture. The differences in the amount of parallelism and the characteristics of the memory subsystem impose the need to develop analytical models and methodologies of analyses to properly quantify and visualize the locality behavior of these applications.

We seek to capture the data reuse characteristic of applications. To do this, we need to have a notion of “time” in order to properly track the memory instructions in the instruction stream. It is for this reason that we adopt the reuse distance concept already used to analyze locality in more conventional processors. The existing methods to perform the reuse distance analysis are not appropriate to capture the multidimensionality of the data utilization behavior of applications running in SIMT processors.

The analysis methodologies developed for conventional processors apply the concept of stack distance. This concept is illustrated in Figure 3(a). Here, the data reuse distance ‘*RD*’ is the number of distinct memory references between two successive references to the same data item [14].

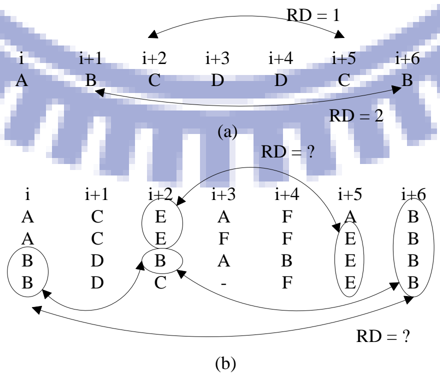
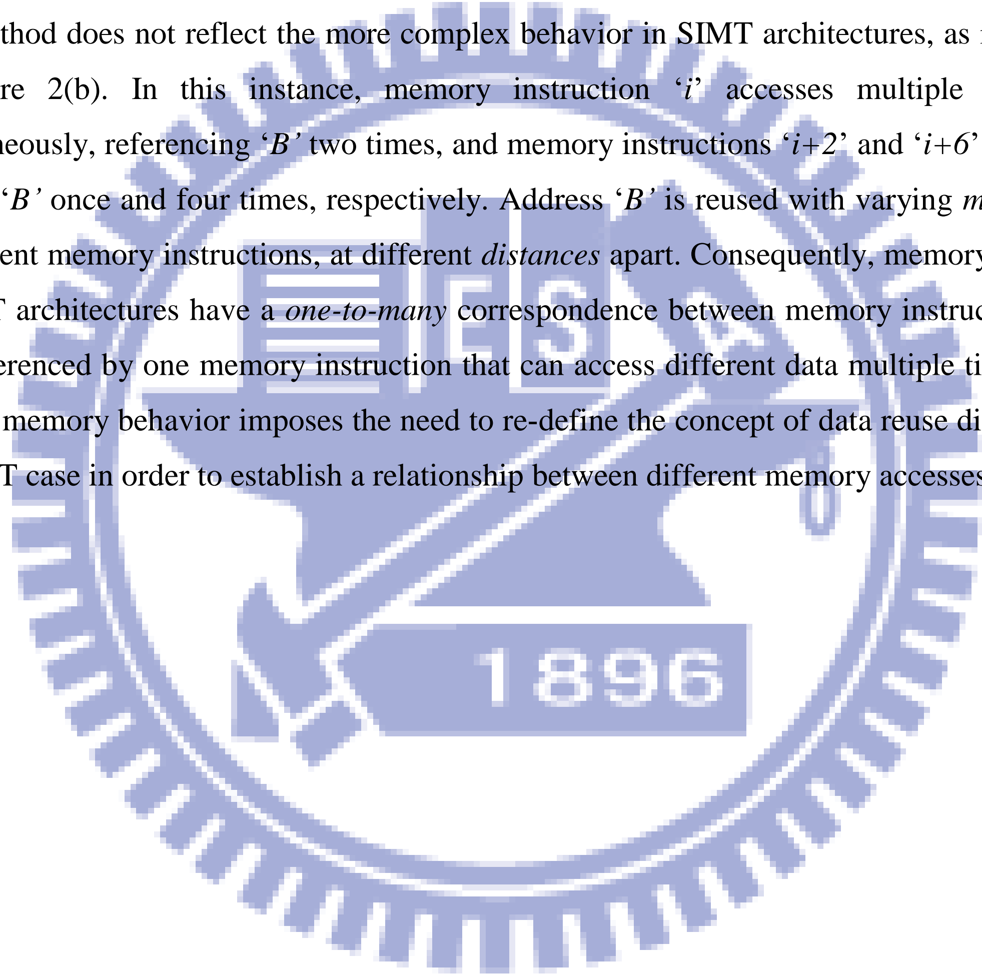


Figure 3: Reuse distance concept and memory instruction reuse behavior in SIMT programs. (a) Sample measurement of reuse distance in traditional multiprocessors. (b) Reuse distance behavior in SIMT architectures.

In Figure 3(a), the term '$i+k$', $k=1,2,3 \ldots n$ represents different memory instructions and the letters below represent addresses of the memory elements accessed. According to the traditional definition of reuse distance, in this example the reuse distance for address '*B'* is $RD = 2$ since there are four references to two different addresses between the two consecutive accesses to '*B'*. Consequently, the reuse distance of address '*C*' is $RD = 1$. This definition of reuse distance is unable to represent the locality characteristics of SIMT applications because it assumes a *one-to-one* correspondence between memory instruction and datum referenced. This method does not reflect the more complex behavior in SIMT architectures, as illustrated in Figure 2(b). In this instance, memory instruction 'i' accesses multiple addresses simultaneously, referencing '*B'* two times, and memory instructions '$i+2$' and '$i+6$' reference address '*B'* once and four times, respectively. Address '*B'* is reused with varying *multiplicity* in different memory instructions, at different *distances* apart. Consequently, memory accesses in SIMT architectures have a *one-to-many* correspondence between memory instructions and data referenced by one memory instruction that can access different data multiple times. This specific memory behavior imposes the need to re-define the concept of data reuse distance for the SIMT case in order to establish a relationship between different memory accesses.



IV. DATA REUSE CHARACTERIZATION

In order to properly analyze locality on SIMT processors, we have to consider the particulars of the SIMT execution model. In SIMT processors, the applications/kernels enter execution in the form of a grid of thread blocks. These are a conglomerate of blocks or, in NVIDIA's terminology, Cooperative Thread Arrays (CTAs) [13]. Each block has a determined amount of threads. The limit in the amount of threads depends on the specific processor architecture. Each block is scheduled for execution to a cluster of cores (SMs, or SMXs). The blocks are further broken down into smaller groups of threads called warps. There's also a warp scheduling mechanism in the each core cluster that issues instructions into the execution pipeline on a per warp basis. The number of threads in each warp is a fixed size for a specific processor. Each thread inside the warps accesses data and executes instructions independently. However, current SIMT processors are limited in the amount of parallelism that *they can exploit* from a given application because of the limits in the amount of processing resources, flow control capabilities and memory subsystem limitations. Thus, only a given number of threads can issue instructions simultaneously.

In order to capture the data reuse characteristic of threads, it is first necessary to examine the relationship between the memory instructions in the threads and the addresses accessed. Second, it is necessary to establish a relationship between different memory instructions that appear in the reference stream as execution progresses. For the former, we define a new metric called "data reuse degree" which quantifies the amount of addresses reused from one memory instruction to the next. For the latter, we employ a re-definition of the reuse distance concept tailored to capture the reuse behavior as execution progresses.

4.1 Definition of the Data Reuse Degree

To explain the concept of data reuse degree, we explore in more detail the properties of memory accesses in SIMT architectures. As mentioned previously, there's a one-to-many correspondence between the memory instructions (MIs) and the addresses they reference. This means that every memory access has an array of addresses **X** associated to it. Figure 4 illustrates a group of MIs '$i+k$', $k,i=1,2\ldots n$ that access a series of addresses. Every MI

therefore can be represented by two entities: a position within the reference stream, given by the index '*i+k*', *k,i=1,2…n* and an array of addresses $\mathbf{X}_{i+k}$.

MI.	i	i+1	i+2	i+3	i+4
	X_0	X_1	X_2	X_3	X_4
	…	…	…	…	…
Address Arrays	A	D	N	I	M
	A	E	A	J	N
	B	F	A	P	H
	C	G	H	L	H
	…	…	…	…	…
	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}
	X_{15}	X_{16}	X_{17}	X_{18}	X_{19}
	X_{20}	X_{21}	X_{22}	X_{23}	X_{24}
Reuse Degree (RD)	-	0	2	0	2+1 = 3

Figure 4: A sample stream of memory instructions. The instruction array appears in the left column, the address array is presented in the middle column and the Reuse Degree for memory instructions '*i*'→'*i+2*' and '*i+2*'→'*i+4*' appears in the right column.

As shown in Figure 4, the addresses in the array may be repeated. This is possible because in SIMT processors more than one thread can request data from the same address, which enables for the same memory request to be serviced by a single memory instruction. As a result, the addresses within an array may appear a certain number of times. When an address '*A*' appears repeated a number of times '*M*', we say that address '*A*' has a multiplicity of '*M*' in memory instruction '*i*'. It can be defined formally as follows:

Definition 1. *The multiplicity* $M_i(A)$ *of address 'A' in an MI 'i' is the number of times that address 'A' appears in the address array* $\mathbf{X}_i$ *of MI 'i'.*

Once the concept of multiplicity has been defined, it is possible then to define the data reuse degree.

Definition 2. *The data reuse degree (DS)* $\mathrm{D}_{i\to j}$ *between two MIs 'i' and 'j', where j>i, in an instruction stream is the sum of the multiplicities* $M_j(\mathbf{X}_{i\to j}(t))$ *in MI 'j' for the array of addresses* $\mathbf{X}_{i\to j}$ *common to 'i' and 'j', given by:*

$$D_{i \to j} = \sum_{t=0}^{T-1} M_j(\mathbf{X}_{i \to j}(t)) \qquad \text{Eq. 1}$$

where $0 \le i < j < S$*, S is the size of the reference stream,* $T = |\mathbf{X}_{i \to j}|$*,* $\mathbf{X}_{i \to j}$ *= {*X_0*,* X_1*,* X_2*, … ,* X_T*} and* X_k *are memory addresses.*

It is clear from Eq.1 that in order to determine $D_{i \to j}$ it is first necessary to determine the common address array $\mathbf{X}_{i \to j}$, which holds the subset of addresses common to MI '*j*' common to '*i*'.

It is necessary to explain the previous definitions and metrics with concrete cases. In the example presented in Figure 4, it is possible to determine the reuse degree *DS* for some of the memory instructions in the sample instruction stream. In MI '*i*', address '*A*' appears twice, and it is used again in MI '*i+2*'. In the latter, $M_{i+2}(A) = 2$. Since '*A*' is the only address in the common address array, then $\mathbf{X}_{i \to j} = \{A\}$ and $D_{i \to i+2} = 2$. When analyzing the *DS* between '*i+2*' and '*i+4*', we can see that the common address array is $\mathbf{X}_{i \to j}$={H, N}. The multiplicities for addresses '*H*' and '*N*' in MI '*i+4*' are $M_{i+4}(H)=2$ and $M_{i+4}(N)=1$, respectively. Thus, the reuse degree $D_{i+2 \to i+4} = \sum_{t=0}^{1} M_{i+4}(\mathbf{X}_{i \to j}(t)) = M_{i+4}(H) + M_{i+4}(N) = 2+1 = 3$. The result appears in the '*Reuse Degree*' column in Figure 4. Notice that the *DS* between MIs '*i*' to '*i+1*', '*i+3*' and '*i+4*' is *zero* (not undetermined). This is so because there are no common addresses between '*i*' and the other MIs different to '*i+2*'. Similar conclusions can be drawn from the rest of MIs.

Once the reuse degree and associated metrics have been defined, it is then necessary to establish a formal relationship between the MIs in the instruction stream in order to obtain the data reuse characteristic.

4.1 Definition of the Reuse Distance

Proposing a definition of the reuse distance concept for SIMT machines becomes necessary in order to model the reuse characteristic of applications. We first examine the concepts behind the reuse distance as it is applied in more conventional processors, and subsequently formulate a definition for SIMT processors.

4.1.1 *Traditional reuse distance analyses*

When applying the reuse distance analysis for applications in uniprocessor and CMP systems, the concept of reuse distance is equivalent to the concept of stack distance using LRU (Least Recently Used) replacement policy as defined by Mattson [15]. In uniprocessor systems, only one data structure (stack, splay tree, among others) is used [16] to store the addresses that the program accesses during execution. Figure 5 illustrates the details of this methodology.

In Figure 5(a), we can see a sample reference stream in a uniprocessor system. As previously mentioned, there is a one-to-one correspondence between the MI and the addresses referenced. Figure 5(b) shows a data structure, a stack in this case, changing state as data is requested. Whenever a memory access is issued by the processor, the stack is traversed to assess whether if the current address being accessed has been previously accessed. If so, as in MI '*i+2*' the RD for this address will be recorded as the number of different addresses i.e. number of entries, between the address being accessed and its previous entry. The previous entry is erased and the new entry is placed at the top of the stack, with an associated distance value. In case no previous entry for that address is found, the RD is recorded as infinity, as in MI '*i*'.

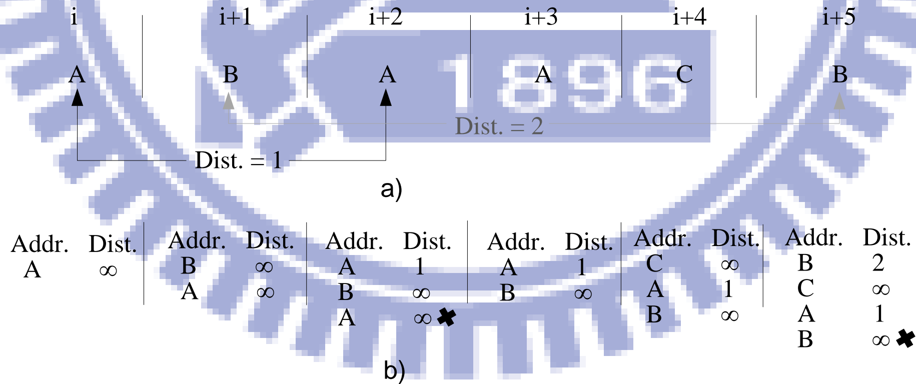


Figure 5: Reuse distance analysis as applied in uniprocessor systems. (a) Sample memory trace. (b) Changes of the state of the stack as memory instructions are issued.

Additional considerations become necessary when applying the reuse distance analysis for CMP systems. Figure 6(a) presents a sample instruction stream for a CMP with separate memory subsystems. There are two processors, *P0* and *P1*, that can request data simultaneously to their respective memories. We alternate the accesses by *P0* and *P1* for

simplicity, but it is not a necessary condition for this case. The one-to-one correspondence between the MI and addresses referenced is maintained from the stack's perspective. Additional details of the CMP system implementation are taken into consideration in order to model locality accurately [11]. Factors such as the presence of private memory subsystems and the details of the coherence mechanism become a part of analytical model for the reuse distance in CMPs.

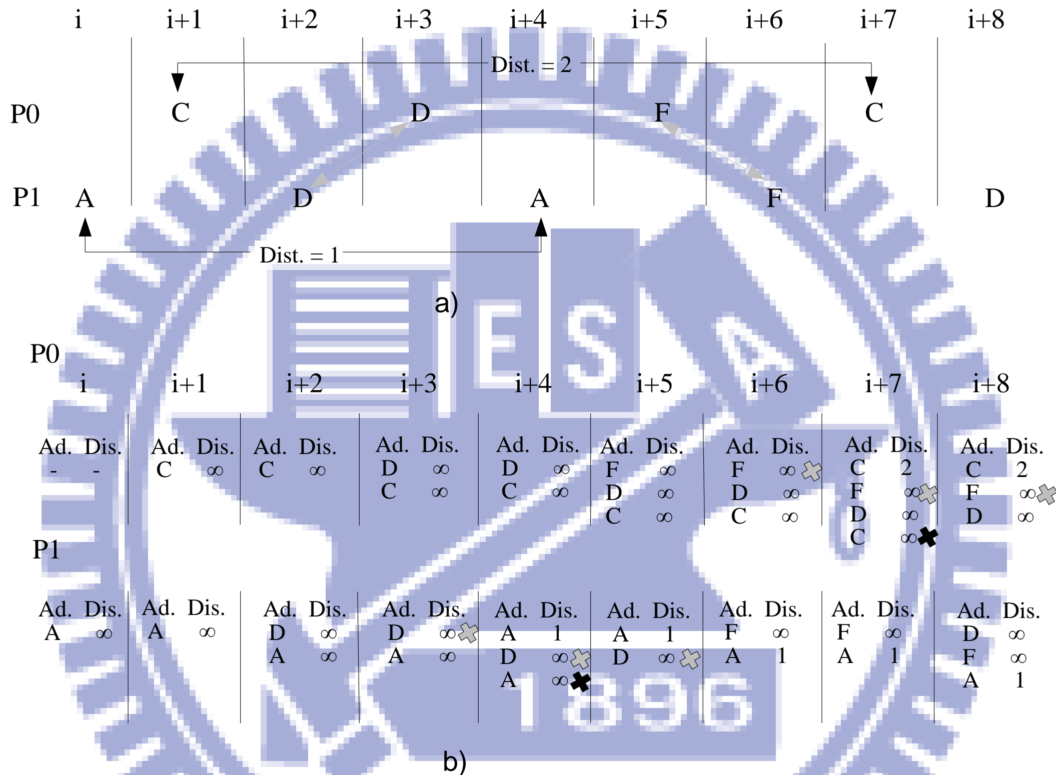


Figure 6: Reuse distance analysis in CMP systems with private memory subsystem. (a) Sample reference stream. (b) Stacks for the private memory subsystem.

In Figure 6(b), we show the corresponding stacks for each memory subsystem in a system with two private memories, assuming that all references are store operations for illustrative purposes. In MI '*i*', processor *P1* accesses address '*A*'. This address is stored as the first entry in the stack of *P1*. In MI '*i+3*', processor *P0* references address '*D*'. This causes the invalidation of address '*D*' in the stack of *P1*, as illustrated by the gray cross at one side of the entry for '*D*' in '*i+3*'. Despite being invalidated, it is maintained until it propagates to the bottom of the stack, as shown in MIs '*i+3*', '*i+4*' and '*i+5*'. The entry is kept so not to alter the RD values that result from referencing another memory element that was referenced prior

to the invalidation of '*D*', as it occurs with *P1* when referencing address '*A*' in MI '*i+4*', resulting in *RD = 1*. However, when the same processor references a previously invalidated element under such circumstances in the stack, it is treated as if it was the first reference to that element. This occurs in MI '*i+8*' for address '*D*', for which *RD* = ∞. Similar events occur for address '*F*' in MI '*i+6*', and for address '*C*' in MI '*i+7*'.

The invalidations due to references only occur for store memory operations. In the case where the instructions are loads, then the memory elements are allowed to have entries in more than one stack simultaneously. The invalidation mechanism implemented in the stacks model the implementation of the coherence mechanism, which is a major design factor in CMPs. The way stacks handle this situation is dependent on coherent mechanism itself, for which there can be many variations.

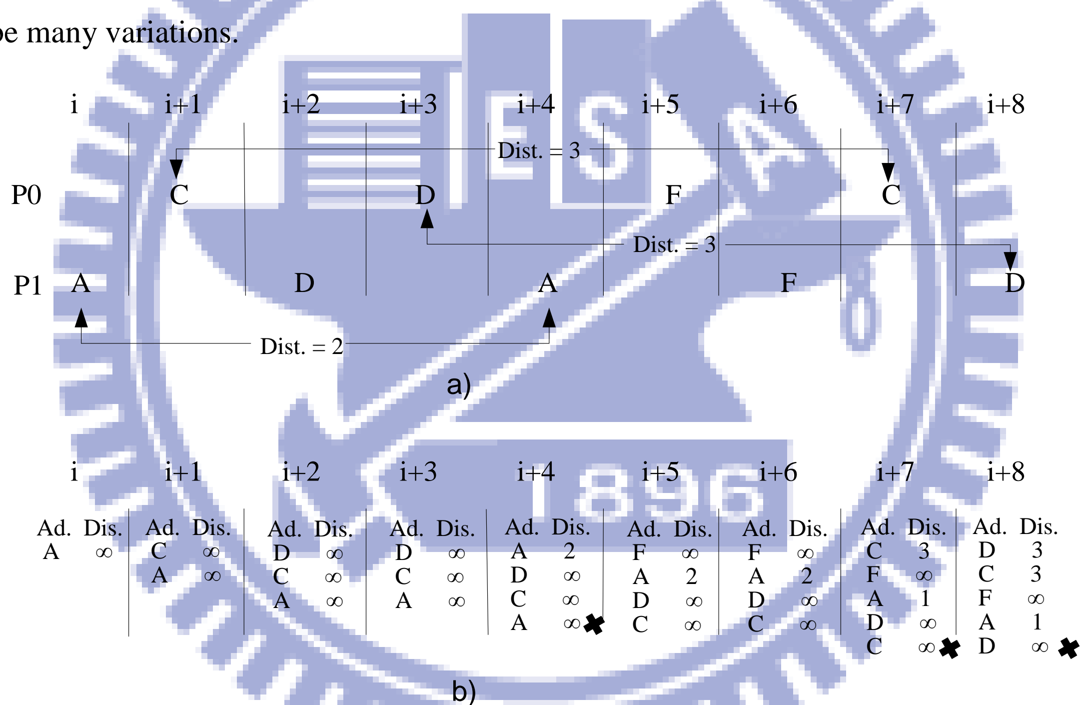


Figure 7: Reuse distance analysis in CMPs with shared memory subsystem. (a) Sample reference stream. (b) Stacks for each memory subsystem.

In Figure 7(a), the same reference stream as in Figure 6(a) is reproduced, assuming also that all reference are store memory instructions. The difference is that, in this instance, processors *P0* and *P1* utilize the same memory subsystem. There is only one stack, and the constraints for it to change state are relaxed. No inter-thread/inter-processor interference occurs as in the previous case. This simplifies the implementation for the distance stacks. In this case, in MI '*i+3*', *P1* references '*D*', and no invalidation occurs despite '*D*' also being accessed

(assuming store operations) by *P0* in '*i+2*'. Also, the distances change for addresses '*A*', '*C*' and '*D*' change. The two main reasons behind this change in behavior are: 1) all addresses referenced occupy one entry in a unified stack, and 2) the absence of stack invalidations. In Figure 6(b), the stacks of each processor will be smaller because these only hold the addresses referenced by the given processor. In Figure 7(b), all addresses have an entry in the same stack, creating a condition where distances vary. On the other hand, for address '*D*', we have that *RD* = 3. This is due to the absence of invalidations, since the previous accesses of any address will be tracked along the whole reference stream, regardless of a different processor accessing it.

4.1.2 Reuse Distance for SIMT Processors

The reuse distance analysis as applied for CMPs cannot be directly applied to SIMT processors. First, the analysis for the former is tightly coupled with the details of the implementation of the memory subsystem and the associated coherence mechanism in a particular CMP. This makes necessary a more intricate mechanism for the management of the entries in the distance stacks. Coherence mechanisms are non-existent in commercial SIMT architectures. Second, as mentioned before, there is not a one-to-one correspondence between the MIs and the addresses referenced. It is not adequate to assign a one dimensional distance value on a per address basis, since this does not model the multi-dimensional locality characteristic that becomes visible when each MI can access multiple addresses. Third, since the stack is traversed for each address accessed in on MI, the number of traverse operations per each MI could be very high, depending on how many threads can execute simultaneously in the architecture. Given that in each MI, the multiplicity for the addresses could vary, and traversing the stack for each address, regardless whether they have multiplicity or not, makes the traditional methodology inefficient for the purpose of characterizing the data reuse behavior. In view of all these limitations, a reuse distance model for SIMT processors is proposed.

Figure 8(a) presents a sample reference stream for a SIMT processor, analogous to the ones in Figures 6(a) and 7(a). In this example, we assume there are '*N*' processors. It is important to note that current state-of-the-art SIMT processors have significantly more cores than general

purpose CMPs. Figure 8(a) shows that each individual core can only request one data simultaneously. But the one-to-many correspondence between the MIs and the addresses accessed is possible because SIMT architectures can serve more than one memory access from more than one thread simultaneously. The actual number of simultaneous accesses that can be served depends on the number of load/store units in a core cluster, the number of core clusters present in the whole system, the width of memory bus interface, among other architectural factors.

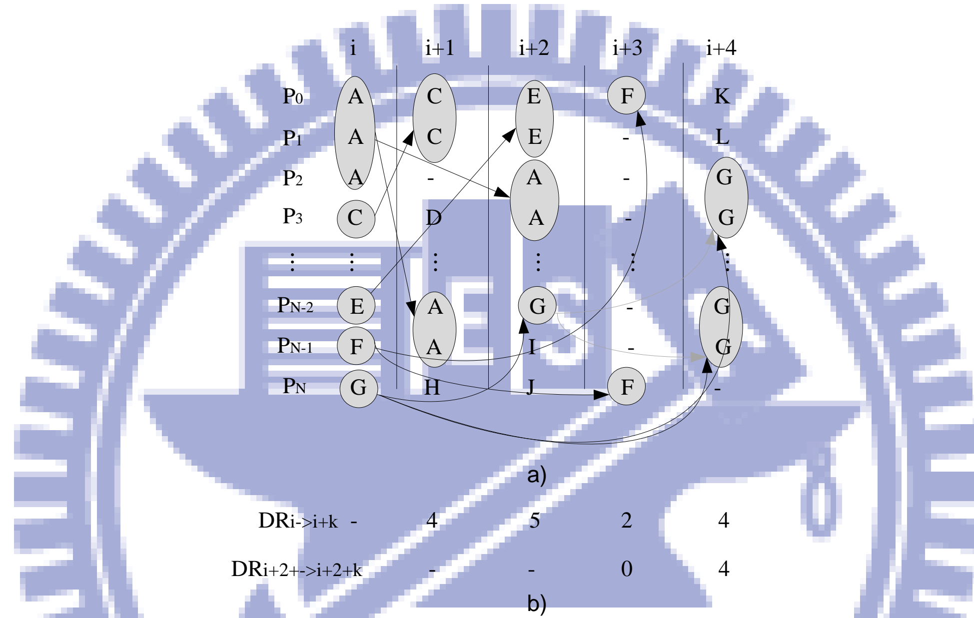


Figure 8: Data reuse degree in a sample reference stream of a SIMT processor with '*N*' processing cores. (a) Sample reference stream of a SIMT processor. (b) Data Reuse Degree for memory instructions '*i*'→'*i+k*' and '*i+2*'→'*i+2+k*'.

Figure 8(b) shows the data reuse degree with respect to different distances apart, calculated as explained in Section 3.1. There's a different value of DS for each MI with respect to a previous one. The values vary depending on the multiplicity of the common addresses between the two memory instructions. If each memory instruction in the reference stream has a different reuse degree with every other MI, then it is safe to say that between two MIs at a given *distance* from each other in the stream there is a specific degree of data reuse. Therefore, we define the reuse distance as follows:

Definition 3. *The reuse distance (RD) between two memory instructions 'i' and 'j', where i*< j, *in the reference stream is defined as the number of memory instructions in between the 'i' and 'j' plus one: RD = i – j.*

At first glance, this might seem as a very simplistic definition for the reuse distance, especially when compared to the original definition. But it is adequate to understand the way an SIMT application reuses data at different stages of execution. The position within the reference stream of the MIs represent the relative time in which they are executed. The data reuse degree varies between two memory instructions, and coupling this with the reuse distance concept previously defined, it is possible to quantify and graphically represent the *magnitude* of the data reuse between memory instructions at different distances apart. Then it becomes feasible to model the data reuse characteristic of SIMT applications with a temporal dimension (RD) with associated magnitude of data reuse.

As seen in Figure 8(b), there can be cases where two MIs with $RD = k$ have no addresses in common, as with '$i+2$' and '$i+3$'. Therefore, the data reuse degree between the two of them is zero. Then, for distance $RD = (i+3)-(i+2) = 1$, $DS = 0$. For simple streams, this can be calculated feasibly, but it is important to also consider the rest of the MIs in the streams in order to properly build a reuse distance histogram in which the value of each entry represents its total amount of DS for that specific distance. The data reuse characteristic is precisely this reuse distance histogram with the corresponding date reuse magnitudes.

V. ANALYSIS METHODOLOGY FOR DATA REUSE CHARACTERIZATION

In Section 3, the data reuse degree was defined and associated with a more appropriate definition of reuse distance for SIMT processors given in Section 4. This combination would allow us to obtain a data reuse characteristic for SIMT applications as each thread executes its correspondent instructions. But in order to build the complete data reuse characteristic it is necessary to develop a step by step description of the methodology, and obtain a physical meaning behind it.

In more conventional processors, the date utilization behavior is uni-dimensional (one address, one memory instruction) in principle. As explained in the previous section, a data structure per memory subsystem (either private or shared) holds an entry for each address accessed. As we have explained before, every time a new memory access is sent to memory, the data structure is traversed in order to assess whether a previous access to the address has been previously issued. Such methodology of analysis is also inappropriate for SIMT processors for the following reasons.

First, since many addresses are requested simultaneously, how to decide which of the addresses occupies which position within the stack? Figure 9(b) illustrates this issue when the MIs '*i*' and '*i+1*' are executed, shown in Figure 9(a). The Option 1 in Figure 9(b) shows a possible way in which the addresses are ordered in a stack when MI '*i*' executes. With this ordering, we can see the values of the RD for addresses '*A*' and '*C*' when MI '*i+1*' executes, which are both equal to 1. On the other hand, if the order in which the addresses accessed by MI '*i*' is altered when populating the stack, as shown in Option 2 of Figure 9(b), the values of the RD for addresses '*A*' and '*C*' change from 1 to 4. The issue is visible because of the complexity in the locality behavior due to the parallelism exploited by SIMT architectures that this methodology is not able to model appropriately. Notice also the stack growth from one MI to the next as compared to the cases in Figures 5, 6 and 7.

The second reason is that the growth of reuse stacks is expected to be much faster in SIMT machines. This is due to the large amount of parallelism present in the architecture. There might be the case that threads have largely independent data sets. Then, traversing a potentially large stack for thousands of different addresses referenced in one single MI is too computationally intensive. The third reason lies in the fact that the locality characteristic

captured by the traditional methodology assumes a sequencing of the memory addresses accessed imposed by the execution and programming model of the conventional processors. This assumption, which is inherent to the methodology used to analyze applications in those architectures, is only valid when the data utilization behavior is essentially uni-dimensional.

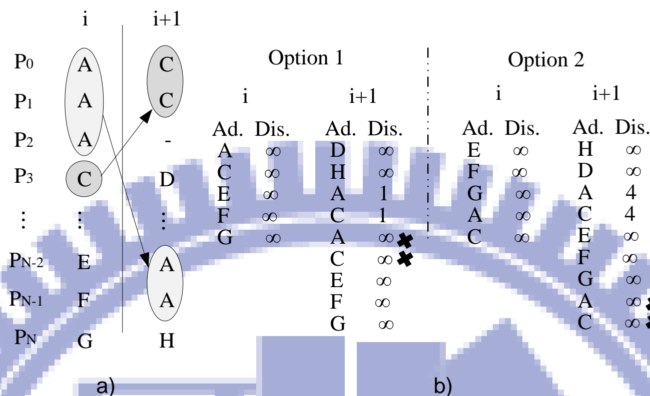


Figure 9: Limitations when performing the baseline methodology for reuse distance analysis on SIMT processors. (a) Subset of the reference stream as it appears in Figure 7. (b) Possible ways to arrange the accessed addresses on the stacks.

Keeping track of each individual address in SIMT applications is a challenging task, to say the least. In addition, the optimization procedure for the kernels these architectures execute is not based on individual addresses, but thinking on the simultaneous multithreading capabilities of the architecture [8, 9]. Code tuning optimizations are therefore implemented from a broader scope, by carefully analyzing the memory access patterns of the applications.

Consequently, we provide a better suited methodology to obtain the data reuse characteristic of SIMT applications based on the data reuse degree and the previously defined reuse distance. The detailed methodology is visible in the flow chart in Figure 10 and synthesized in **Algorithm 1**. We describe it as follows:

1. Select a memory instruction 'i' for analysis.

2. Scan through the addresses accessed by MI 'i' and build the address array '$\mathbf{X_i}$'. Store this information in a data structure. In such data structure, each entry is keyed with the address, and the value 'v' of the entry is $v = M_i(X_i)$, i.e. the multiplicity 'M' for the addresses in the address array '$\mathbf{X_i}$' of MI 'i'.

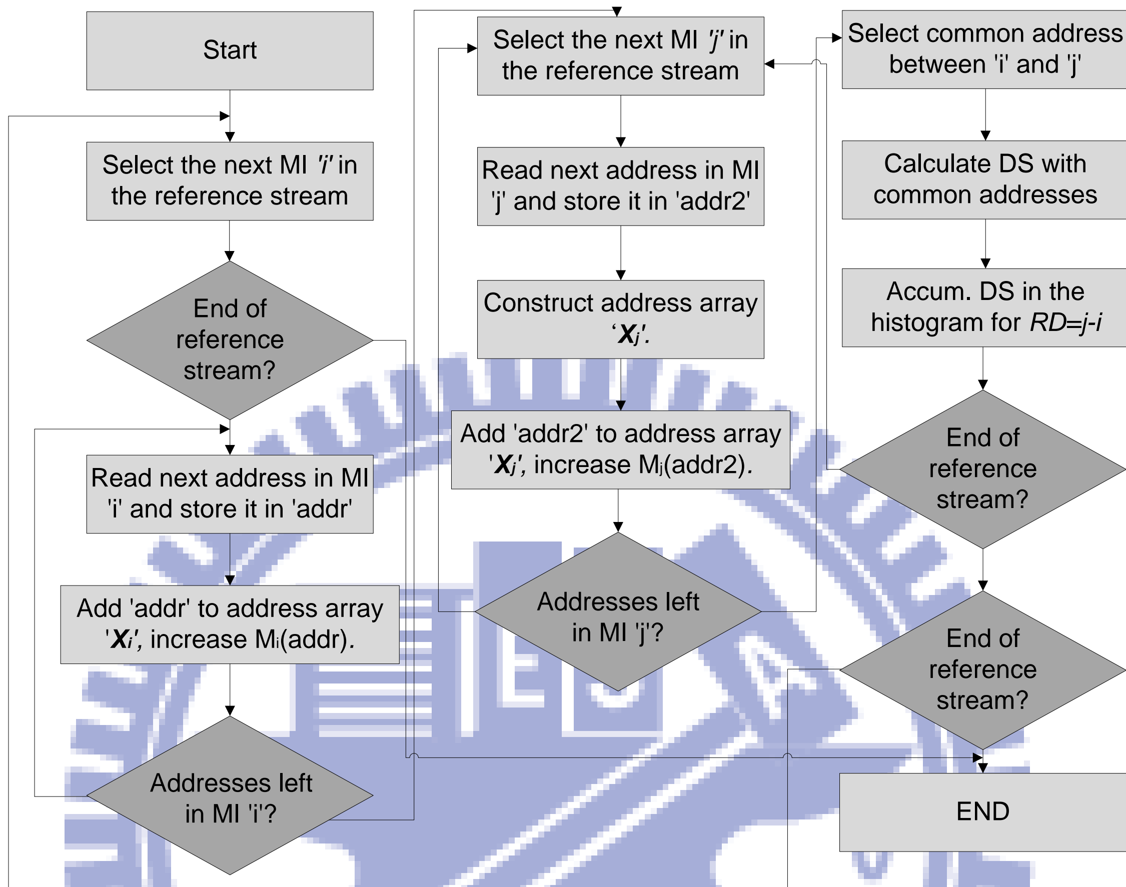


Figure 10: Flow chart of the data reuse characterization methodology. This flow chart shows all the steps when performing the analysis over a reference stream.

3. Select the subsequent MIs '*j*', where $j > i$. We can define $j = i+k$, where $k \geq 1$, imposing the condition $j < (S - 1)$, where '*S*' is the number of MIs in the reference stream of the application.

4. Create a different data structure for MI '*j*' similar to the one created for MI '*i*', and create address array '$\mathbf{X_j}$'. As with step 2, the keys are the addresses in the address array of '*j*' and the value of each entry is the corresponding multiplicity of the addresses in the array.

5. Search for the common addresses in both data structures, and determine the total data reuse degree (DS) between MI '*i*' and '*j*'.

6. Once the DS has been calculated, the result is accumulated in the reuse distance histogram in the entry corresponding to $RD = i - j$.

7. Steps 2~6 are repeated for all subsequent MIs that appear later than '*j*' in the reference stream. All the data reuse degrees for all subsequent cases will be accumulated in the corresponding entry of the reuse distance histogram

8. Steps 1~7 are repeated for all subsequent MIs that appear later than '*i*' in the reference stream. The data reuse degrees are stored in the corresponding reuse distance histogram entries. Thus, the data reuse characteristic is generated.

Algorithm 1. DR_Analysis( )

```
// RF is the reference stream
1.   S = RF.size;
2.   for i=0 to S-1;                              // Step1
3.     addressStruct(mI,RF(i));                   // Step 2
4.     for j = i + 1 to S-1                       // Step 3
5.       addressStruct(mJ, RF(j));                // Step 4
6.       for a = 0 to mJ.size                     // Step 5
7.         if ( mI.find(mJ(a)) )
8.           mIJ.put(mJ(a));
9.           DR = DR + mIJ(a).v;
10.    RD = i - j;                                // Step 6a
11.    hist(RD) = hist(RD) + DR;                  // Step 6b
12.
13. Function addressStruct( m, MI )               // scans addresses,
14.    for k=0 to MI.simAccesses;
15.      is = m.find( MI(k).address );
16.      if ( is )
17.        m[MI(k).address]++;    // calculates multiplicity
18.      else
19.        m.insert(MI(k).address);
20.        m[MI(k).address] = 1;
```

This model in particular uses reduction operations that concentrate all the reuse degree values from the reference stream in the respective histogram's entries. This gives insight on how *frequently* does the application reuses data and at which *frequency* does the application reuses data the most. Not only that, it also gives insight what are the total varying degrees of reuse at these different *frequencies*.

One particular property of our methodology is that, unlike the case for CMP systems, it is not dependent on the details of the SIMT architecture's memory subsystem, or on any other implementation details. The data structures are built based on the MIs of the reference stream, and the histogram is accumulative. The MIs are analyzed in the order in which they are expected to be issued from the application's perspective. In a real architecture, the ordering of the MIs and the addresses they access are heavily dependent on the practical limitations of the architecture itself: the number of simultaneous transactions carried out by the memory subsystem, whether there are bank-conflicts, the bandwidth utilization, etc. Thus, there is no explicit trade-off between accuracy and generality embedded in our methodology, since it is mostly determined by the code structure of the kernel.

As mentioned, the methodology allows analyzing the reference stream independently of any particular architecture. This enables to model locality under different conditions that would yield a data reuse degree characteristic *abstracted* from the limitations of current architectures. For example, given that we have a proper instrumentation tool, we would be able to obtain the reuse distance histogram assuming no limitations in the architecture's resources: infinite SMs, infinite load/store units, infinite thread capacity allocation, number of registers and issuing capabilities, bandwidth limitations, etc. In this way, we model the application's locality under a very controlled environment, solely dependent on the application's structure and the SIMT programming model, a feature particularly useful given the fast pace at which SIMT architecture are currently evolving.

VI. SCENARIOS FOR DATA REUSE CHARACTERIZATION

So far, we have detailed the methodology of the analysis in Section 5 using the metrics defined in Sections 4 and 5. The objective is to generate a data reuse characteristic of the application. In this section, we describe the different scenarios in which the data reuse characteristic is obtained, and certain particular properties of SIMT processors that are considered in the analysis. In this work, we focus specifically on GPU architectures, since are the most widely used, although our methodology can be applied to other types of SIMT processors.

In order to perform the analysis, we developed an experimental framework which we called Locality Analyzer for SIMT applications, programmed in C++. The details of our framework are explained in Section VII. The framework allows the modification of certain parameters and to choose the desired scenario. The six scenarios of the analysis implemented so far are:

Scenario 1. Analysis with infinite resources, thread blocks serialized: All threads are modeled as executing in parallel, while blocks are modeled as executing one at a time.

Scenario 2. Analysis with infinite parallel resources, within block analysis: All threads in the block are modeled as executing in parallel. The execution model of thread blocks is irrelevant for this scenario.

Scenario 3. Analysis with infinite resources, all blocks are modeled executing in parallel: all threads are modeled as executing in parallel, no resource limitations of any kind.

Scenario 4. Analysis with infinite resources, '*K*' blocks are modeled as executing in parallel: all threads in a block are modeled as executing in parallel, models limitations on the blocks that can execute concurrently. The '*K*' concurrent blocks are chosen according to a scheduling policy.

Scenario 5. Analysis with limited resources, '*K*' blocks in parallel: core clusters are modeled, and blocks are assigned to them according to a specific scheduling policy. The analysis focuses on the local reference streams of each core cluster.

Scenario 6. Analysis with limited resources, '*K*' blocks in parallel: core clusters are modeled, and blocks are assigned to them using a scheduling policy. The analysis focuses on the resulting reference stream of the core cluster collective.

6.1 Infinite resources, thread blocks are modeled as executing sequentially

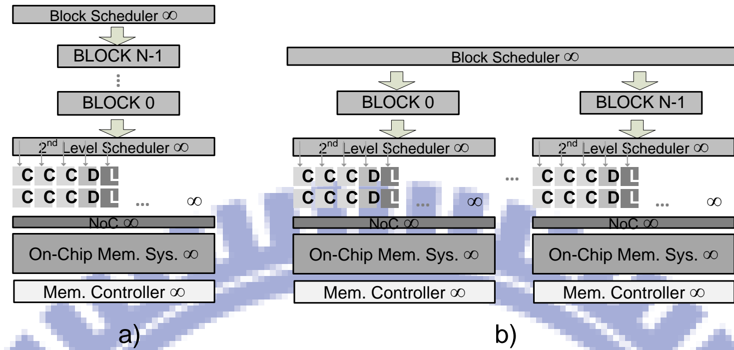


Figure 11: Data reuse characterization. (a) Scenario 1. (b) Scenario 2. In these two scenarios, the SM is assumed ideal, as represented by the infinite signs in the figures above.

In Scenario 1, the analysis of the trace is performed assuming that in fact the blocks are modeled as executing sequentially. These result in a reference stream in which instructions are repeated with nearly uniform frequency. However, branch divergence and other runtime dynamics will modify the reference streams of different blocks. This type of analysis does not comply with the execution behavior presented by current SIMT processors. In a real architecture, many blocks can execute in parallel in different core clusters. However, by performing the analysis assuming block sequencing, we will be able to observe some important characteristic of the cross-block data reuse behavior, if any. Figure 11(a) illustrates this case.

6.2 Infinite resources, analysis within each thread block

Scenario 2 executes under identical conditions than the initial scenario, except for the fact that the analysis is performed within the thread block. That is, the data reuse characterization is performed only within the reference stream of the block occupying the ideal core cluster. A core cluster with such characteristics allows all threads to execute simultaneously. Within the reference stream context, this means that all memory requests by the threads are served

concurrently, with no latency or scheduling that can affect the ordering of the memory instructions. The analyses in scenarios 1 and 2 are performed assuming this particular ideal core cluster. Figure 10(b) illustrates this case.

6.3 Infinite resources, all thread are modeled as executing in parallel

As with the case with the first two scenarios, scenario 3 of the data reuse characterization is also performed assuming an ideal core cluster. In this case, however, the highest theoretical amount of parallelism becomes available by allowing all blocks to execute simultaneously. In order to model this behavior properly, the reference streams of each block are merged into one single reference stream. The resulting reference stream is called the “Aggregate Block”, a concept which we will defined briefly in a formal way. In this case, the maximum number of threads that are able to issue memory requests is equal to the number of MIs executing memory instructions in a specific position of the reference stream. Figure 12 illustrates this scenario.

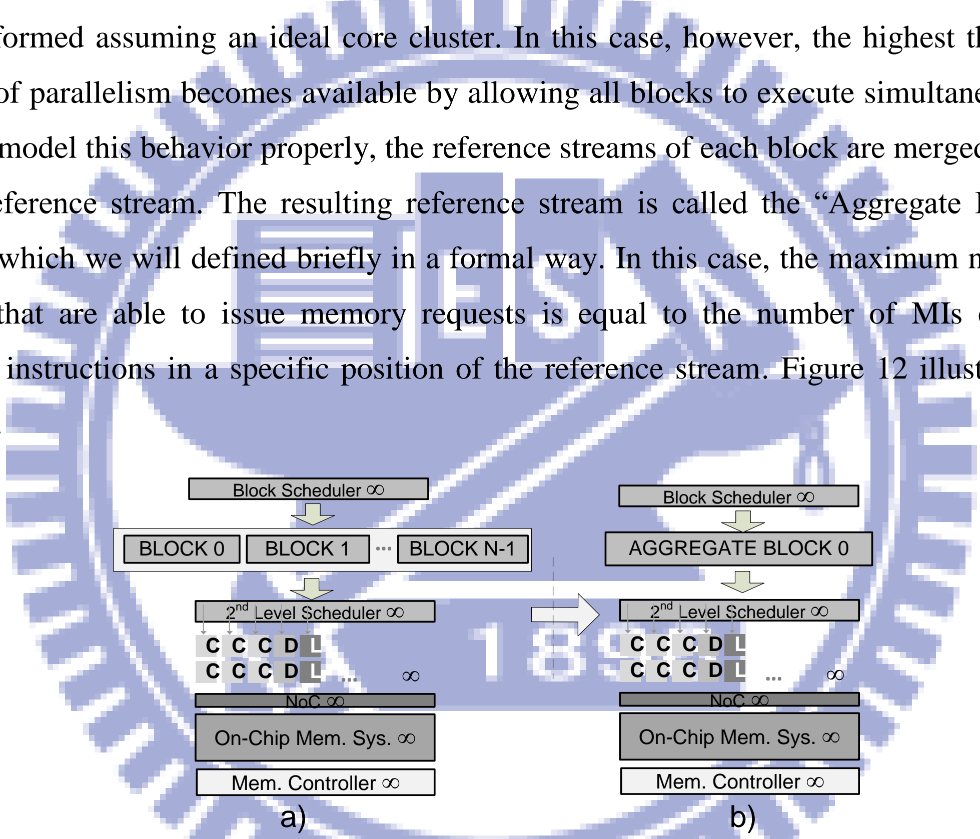


Figure 12: Scenario 3 of data reuse characterization assuming ideal core clusters. (a) Illustrates the way blocks are intended to be executed. (b) Illustrates the Aggregate Block that results from merging the streams of all CTAs.

The analysis as performed by Scenario 3 offers significant insight on the applications data reuse behavior. The model illustrated in Figure 12 maintains the constraints of the execution model inherent to SIMT processors, without all the practical parallelism and memory subsystem limitations of a specific architecture. Performing the analysis under such conditions will provide a very particular data reuse characteristic of the application given its code

structure and input. Scenario 3 allows us to obtain a reference data reuse characteristic that will be used to compare how different coding optimization techniques impact the data reuse characteristic, and how close they get to reproducing the ideal characterization. For example, a code can have a relative small number of blocks that can be allocated to one core cluster. If there are enough core clusters, and each cluster has enough resources, the ideal data reuse characteristic will be reproduced. Having a reference data reuse characteristic will enable to quantify how optimization procedures alter the reuse behavior.

6.4 Infinite resources, a number '*K*' of thread blocks modeled as executing in parallel

Scenario 4 is the first one that models limitations on the amount of blocks that can execute concurrently. It has the particularity that it allows to include different block scheduling policies.

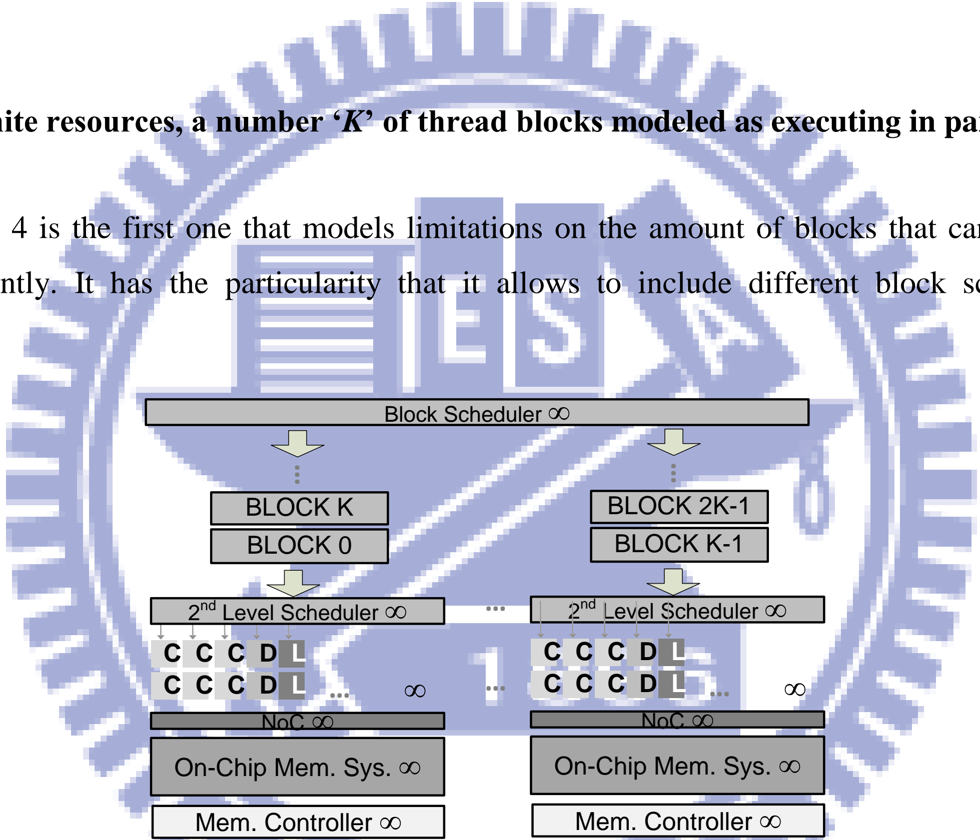


Figure 13: Scenario 4 of the reuse degree characterization. Only '*K*' blocks are modeled as executing concurrently. This is equivalent as having 'K' ideal core clusters, each one executing one block at the time.

Figure 13 illustrates this situation. There are '*K*' ideal core clusters available, each one running one single block. The scheduling policy implemented in any particular case can select the following block to execute in each SM, resulting in varying reference streams per core cluster.

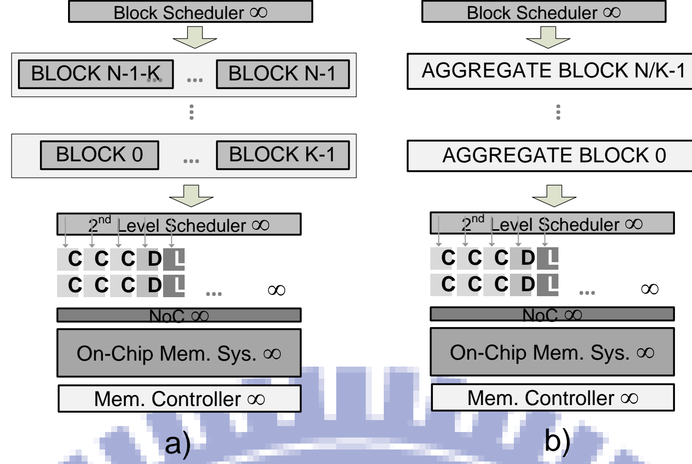


Figure 14: (a) An ideal core cluster executing the reference streams of '*K*' parallel blocks. (b) The streams of parallel blocks merged into a series of aggregate blocks, executed in sequence.

Since the core clusters considered in Scenario 4 are also considered to be ideal, the resulting effect of having blocks executing in parallel can be modeled as in Figure 14. This ideal core clusters represent an abstraction of parallelism resources, which we call array of concurrent slots, which we define as follows.

Definition 3. *An array of concurrent slots is an abstraction of a collective of parallel resources capable of executing the instruction/reference stream of a determined number of block(s) simultaneously.*

In the Locality Analyzer, concurrent slots appear only in arrays of more than one element. The blocks within each array of concurrent slots execute in parallel, but arrays are serialized with respect to each other. The blocks running in parallel in one ideal core cluster i.e. array of concurrent slots, as in Figure 14(a), can be merged together to create a series of $\left\lceil \frac{N}{K} - 1 \right\rceil$ aggregate blocks. We define an aggregate block as follows:

Definition 4. *An aggregate block is the reference stream that results from merging the reference streams of the blocks in the corresponding array of concurrent slots.*

These are then serialized as show in Figure 14(b). The analysis is therefore performed over serialized aggregate reference streams over ideal core clusters. It is important to mention that the blocks in concurrent slots are not always merged to create a resulting aggregate block. The merging process will take place depending on the analysis performed, and the parallelism resources to be modeled by a specific analysis.

6.5 Limited resources, '*K*' block modeled as executing in parallel, within core cluster analysis

Scenarios 5 and 6 of the analysis obtain the data reuse characteristic under conditions in which architectural limitations of the core clusters are modeled. The scheduling of threads is performed on a per-warp basis on NVIDIA GPUs. In NVIDIA GPUs and the number of threads in a warp is 32. The warp size harmonizes with other design characteristics of NVIDIA's GPUs: memory bus sizes, cores and functional units. The latter play a major role in the number of cycles needed for a warp to fully execute one instruction.

For the case of memory instructions, the number of cycles per instruction per warp, assuming an ideal memory subsystem, will be dependent on the number of load/store units available to each warp in a given cycle. By taking these into consideration, only a specific amount of threads will be able to issue memory accesses. In certain commercial GPUs, the amount of load/store units available for a warp in one cycle is usually 16, the size of a half-warp. As a consequence of this, the cycles necessary to complete a memory instruction increase. Scenarios 5 and 6 try to analyze the effect on the data reuse characteristic of an application under these conditions.

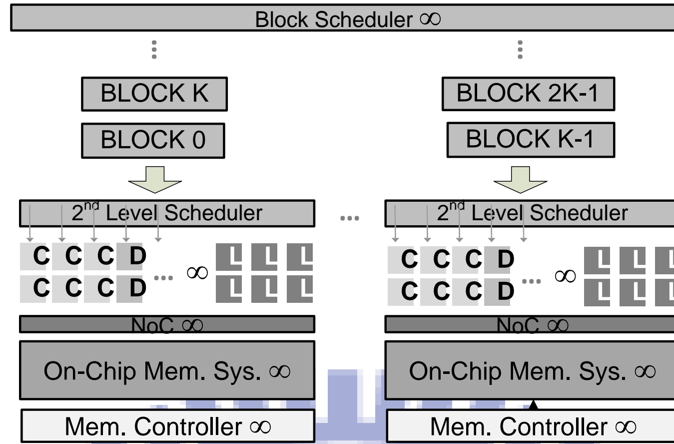


Figure 15: Scenario 5 of the data reuse characterization analysis. Each core cluster has now a finite number of load/store instructions. The analysis is performed within each core cluster.

Figure 15 illustrates the case for Scenario 5. In this case, there is a finite number of load/store units per core cluster. The warp scheduler inside the core cluster can only issue a number of memory instructions that the load/store units can give service to. In our framework, the number of load/store units can be decided at runtime by the user. This scenario models an additional resource constraint that reduces the total amount of parallelism that the SIMT processor can exploit. In Figure 15, the details of the memory subsystem are modeled ideally, so not to make the analysis depend on the architecture.

The data reuse characterization of Scenario 5 is done on a per core cluster basis. Each core cluster is assigned a series of thread blocks, as shown in Figure 15. As mentioned before, each core cluster is are a more physical representation of the array of concurrent slots. In this case, aggregate blocks are not used despite assigning blocks to each array of concurrent slots. The merging process does not take place even though parallelism can still be exploited. However, Scenario 6 does perform the block merging, as we shall see, and characterizes the data reuse behavior from a different perspective.

Scenario 5 analyzes the reference stream resulting from the serialized blocks assigned to each core cluster. This analysis captures the data reuse behavior that could be taken advantage of by an ideal shared memory subsystem within a specific cluster. The scheduling policy and the number of core clusters in the architecture will definitely have an impact on the reuse characteristic under such circumstances.

Since the number of load/store units is limited, all threads are unable to request accesses simultaneously. Therefore, more memory requests will be issued, which will increase the size of the reference stream of each block, and of all the overall blocks assigned to a core cluster. This will have a significant impact on the data reuse characteristic and the length of the histogram itself. In general terms, it will modify the way the application reuses data.

6.6 Limited resources, '*K*' blocks modeled as executing in parallel, inter-core cluster

The sixth and final scenario of the data reuse characteristic analysis is identical to Scenario 5 except for one fundamental difference. In this case, the blocks executing in parallel are merged into aggregate blocks. Figure 16 illustrates this case. The execution is modeled as if the series of resulting aggregate blocks where executing in a core cluster in which the total number of load/store units is the aggregate amount of load/store units present over all clusters in the system. This analysis captures the data reuse behavior that could be taken advantage of by an ideal shared memory between the overall threads of all clusters.

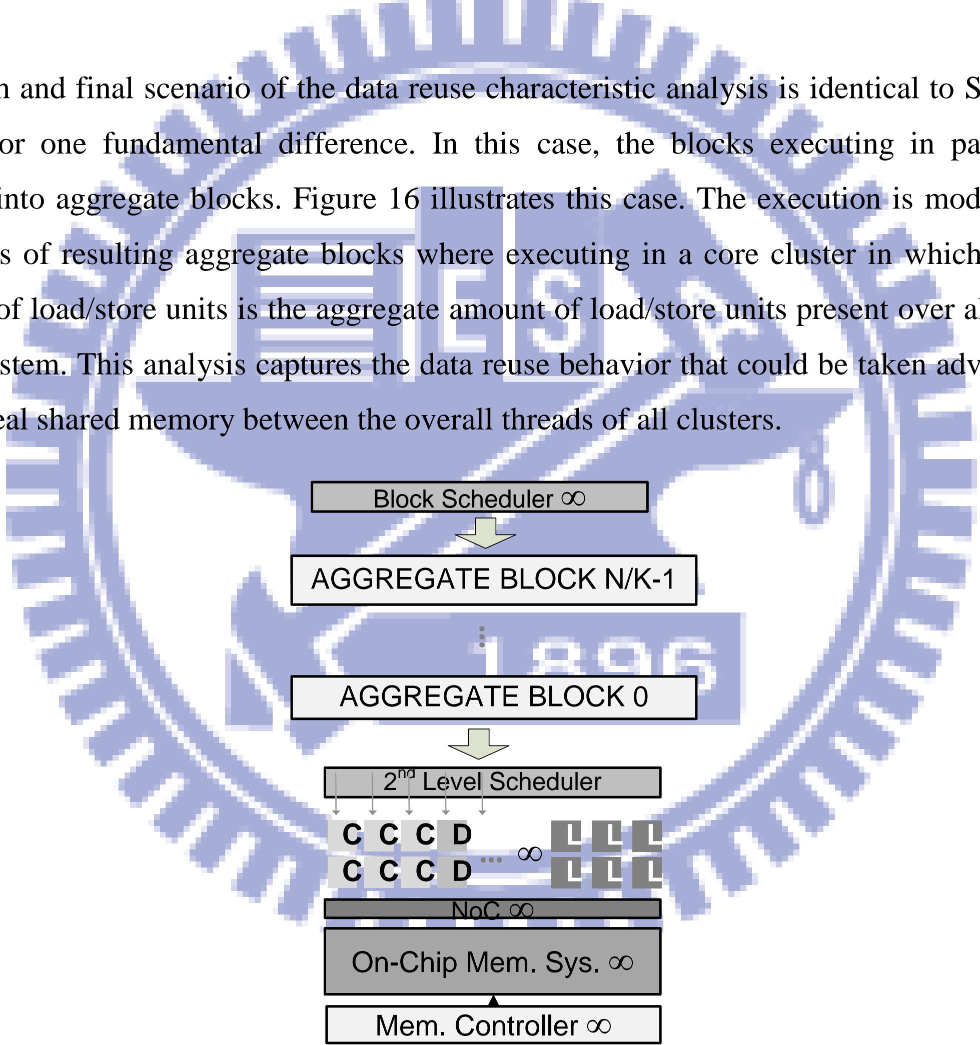


Figure 16: Scenario 6 of the data reuse characterization. The analysis is performed over the aggregate of the reference streams in every core cluster. The number of load/store units is the total sum across the core clusters.

The purpose of all these analyses is to get a quantified representation of the reuse characteristic under different parallelism constraints. The amount of parallelism that SIMT

processors can exploit is what fundamentally differentiates them from more conventional processors. This is coupled with a specific programming model. When the amount of parallelism that the processor can handle changes, it will interact in a different way with the application: less or more threads can execute concurrently, occupancy varies, coalescing will vary and locality characteristics will change as well as the real resource limitation. All this causes changes in overall running time, and memory performance. The resulting performance is multivariable, and it is difficult to build a model of an application's performance just by observing the way it varies.

By providing a way to characterize the data reuse behavior of an SIMT application, it is possible to get further insight on the resulting performance and ways to predict it. Scenario 3 provides a particular data reuse characteristic given the kernel code. This same characteristic will be reproduced if the SIMT processor has enough resources to exploit all the parallelism needed by the kernel. However, as kernels utilize bigger data sets and have larger reference streams, reaching this condition might not be a practical goal. But the ideal data reuse characteristic will be adequate to assess the positive or negative impact that different parallelism constraints will have in its reuse characteristic. This is achieved in a totally isolated way from other factors that affect performance, such as the capabilities for coalescing, or the memory subsystem.

When code tuning is performed, or architectural enhancements are added to the processor, developers proceed in view of the architecture and the programing model. A code tuning technique to improve data coalescing, for example, can also have an impact on the bank-conflict avoidance, contention avoidance, and on the way the schedulers issue instructions, which in turn will have other effects in different parts of the architecture. Detailing this cascade effects is particularly difficult given the cross-relation between them.

Now that the analyses have been detailed, the next section explains the details of the implementations of the experimental framework.

VII. EXPERIMENTATION FRAMEWORK

In order to perform the analyses, we developed an experimental framework which we called Locality Analyzer for SIMT applications, programmed in C++. In order to perform all the analyses described in Section VI, there are certain steps that need to be taken to ensure accuracy of the results, functional correctness and fidelity.

In this work, all the analyses for the data reuse characterization are trace-based. This means that the application is executed either on a PTX instruction emulator, and data relevant to its runtime is captured in trace files. Therefore, there are two stages that need to be considered to perform any of the analyses detailed so far: trace generation stage and reference stream analysis stage.

7.1 Trace Generation Stage

Our experimental framework takes as input a memory trace generated by a PTX instruction emulator. For our experiments, we used the framework provided by the GPUOcelot [17]. The GPUOcelot provides a trace generation tool that can capture a series of performance metrics from GPU kernels, as well as runtime information of the threads. To perform our experiments, we modified the trace generation code of the GPUOcelot, and modified it in order to output a trace with a format better suited to perform the analyses explained in Section VI. Any proper instruction emulator can be used to generate the trace, as long as it satisfies the trace formatting.

The trace information contains the memory instructions executed by the kernel. The tracer executes each block individually i.e. blocks are serialized, and reports the threads executing each memory instruction. Therefore, the trace file contains the MIs executed in each block, the threads that execute each MI, the number of simultaneous accesses and the addresses of the requested memory elements. Our analysis tool can model infinite parallel execution resources, as we shall see in the next section, but the GPUOCelot imposes a maximum of 1024 threads per block. Therefore, the results presented here present an idealized version similar to having infinite resource, when the block size is below 1024. When the size of the thread blocks exceeds this value, then the constraints imposed by the runtime of the GPUOcelot become a

limiting factor. The capacity to run threads of one thread block concurrently in one core clusters, to issue and execute instructions at this pace by far exceeds the capabilities of current commercial core clusters.

The traces are obtained in a per kernel basis. When applications are executed on GPU architectures, they can contain multiple kernels, call the same kernels multiple times and/or both. Since the blocks belonging to a specific kernel are executed in a serialized way, the traces contain an inherent order of the reference streams on a thread block basis. This is a useful property, since it provides a lot of flexibility to model different execution conditions of SIMT processors.

7.2 Reference Stream Analysis Stage

In this section, we explain the specifics of the implementation of the Locality Analyzer for SIMT applications. The purpose is to show and demonstrate the utility provided by our framework, the capabilities it has and the way the previous described analyses were implemented.

7.2.1 Model for Thread Blocks

The first task to perform any of the analyses detailed in the previous Section 6 is to read the trace file using the Locality Analyzer and build the reference stream of each block individually. Since the GPUOcelot emulates the blocks in a sequential way, the trace file captures the reference stream of each block in sequence. The trace format contains the position of the MIs in the reference stream, which we call the position index, the threads that specific MI and the addresses accessed. The trace is scanned and the information of reference streams is stored in an array of "block" objects. This array, which can be thought of as a block container object, would be equivalent to the grid, and the blocks are inserted in the order they are executed by the GPUOcelot. This procedure is graphically explained in Figure 17. The position index is the MI count. That is, the number of MIs that have been executed prior to a specific MI plus one.

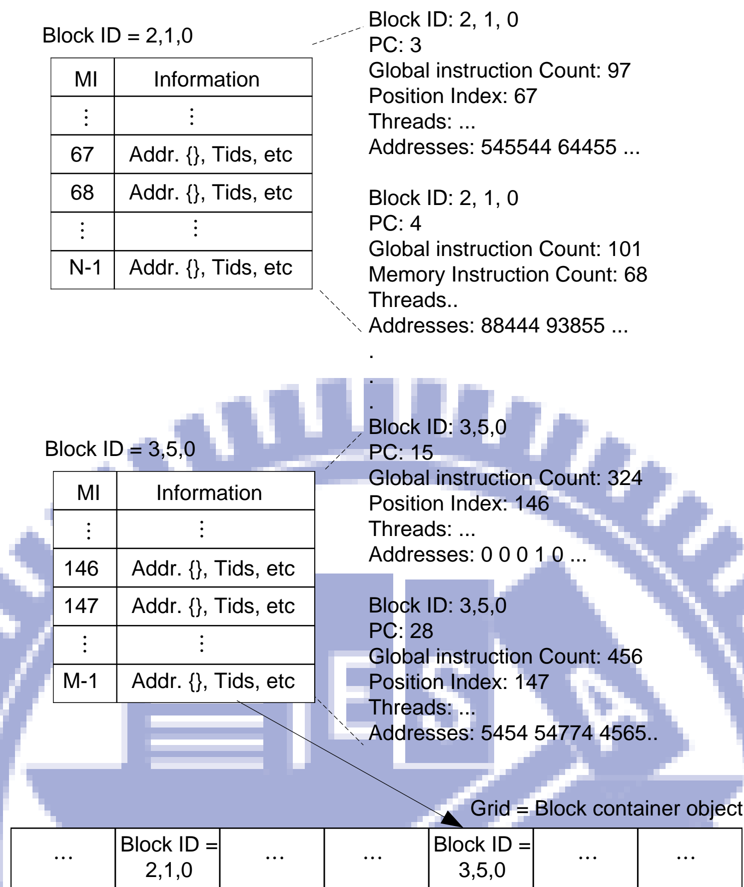


Figure 17: Formation of Block objects within the Locality Analyzer. The block objects contain their individual reference stream which consists on a series of ordered MIs. Each MI has access information of its own.

Each MI executed has an entry in the trace file. Each entry contains information to which thread block does the instruction belong to, its relative position in the stream of that block (position index), the global instruction count, the threads that issue this instruction, and the addresses accessed. The latter is the address array explained in Section 4. **Algorithm 2** shows the pseudo-code for the reconstruction of the thread block objects within the framework of the Locality Analyzer.

The instructions within the *while* loop declared in line 2 execute as long as the trace file has not been completely read. Line 3 creates a '*currBlock'* object, which is an instance of the '*blockObj'* class. This class contains all the information necessary to identify the block, contains its reference streams, the addresses each MI accesses, and functions to manage this information. Line 4 first reads the next block ID to read from the trace file, and stores it in

both variables '*currBlock.blockID*' and '*nextBlock*'. The *while* loop in line 6 checks whether or not these variables hold the same ID. In case they are not equal, the block so far built within the while loop in line 6 is inserted in the grid object, passed as an argument to the *GridFormation*() function.

In case the condition *currBlock.blockID == nextBlock* is true, all the instructions from line 7 to 21 are executed. Line 7 creates an '*inst*' instance of an '*instInfo*' class. This class holds the address array of the specific MI, together with other information of the MI such as the PC, the threads executing this instruction (*Threads*), the position index within the reference stream ('*posInRefStream*'), the global instruction index ('*genPos*'), analogous to the former but considers all instructions in general. All these information is read from the trace file in lines 8 to 11.

Line 12 reads the threads issuing the current MI. This is the number of addresses being accessed. The *for* loop declared in line 12 reads the addresses accessed by the threads in the current MI one by one, storing them in the variable '*Addr*', in line 13. In line 14, the algorithm searches an entry for the address '*Addr*' in the address array. In case it finds an entry with the value '*Addr*', line 16 will increment the multiplicity of that address. In case not, a new entry is created in the address array, and its multiplicity is initialized to 1, as shown in lines 18 and 19, respectively.

Once all addresses of the current MI have been read, the '*inst*' object is inserted in the stream field '*currBlock*' object, which is its corresponding reference stream. This task is carried out in line 20. In line 21 the next byte to be read in the trace file is stored, and right after that the following block ID is read. In case this ID is not equal to the one of the current block, then the next thread block in the trace has been reached. Line 23 checks for this condition, and moves to the memory position prior to the read operation in line 24 if necessary. Line 25 inserts the block object into the grid. When the trace has been completely read, the algorithm terminates.

Algorithm 2. GridFormation(GridObj)

```
1.  Tpos = 0;
2.  while (Tpos < TF.size-1)    // TF is the trace file
3.    blockObj currBlock;
4.    currBlock.blockID << TF;
```

```
5.      nextBlock = currBlock.blockID
6.      While nextBlock == BlockID
7.          instInfo inst;
8.          inst PC << TF;
9.          inst posInRefStream << TF;
10.         inst genPos << TF;
11.         inst Threads << TF;
12.         for k = 0 to Threads
13.             Addr << TF;
14.             Bool is = inst.addrArray.isPresent(Addr);
15.             if (is)
16.               addrArray[Addr]++;
17.             else
18.               addrArray.insert(Addr);
19.               addrArray[Addr] = 1;
20.         currBlock.stream(inst);
21.         currByte = TF.seek(TF.curr);
22.         nextBlock << TF;
23.         if (nextBlock != currBlock.blockID)
24.             TF.seek(currByte – nextBlock);
25.   GridObj.insert(currBlock);
```

After the block objects have been formed and stored in the grid container, the next step is to select which analysis to perform and pass the necessary parameters to the program when necessary. Among the parameters that the program can take are: size of the memory element in bytes, number of core clusters, number of load/store units, type of block scheduling policy, and number of concurrent parallel blocks.

The Locality Analyzer allows the implementation of different block scheduling policies and modeling of certain architectural characteristics of the core clusters, such as warp scheduling policies and load/store units.

7.2.2 Block Scheduling Policies

The Locality Analyzer enables can analyze the impact that different block scheduling policies have over the data reuse characteristic. This feature is enabled by allowing the developer to program its own block scheduling policy, select blocks according to the desired policy from the grid object, and assign it to a concurrent parallel slot. Figure 18 illustrates two cases for the scheduling procedure.

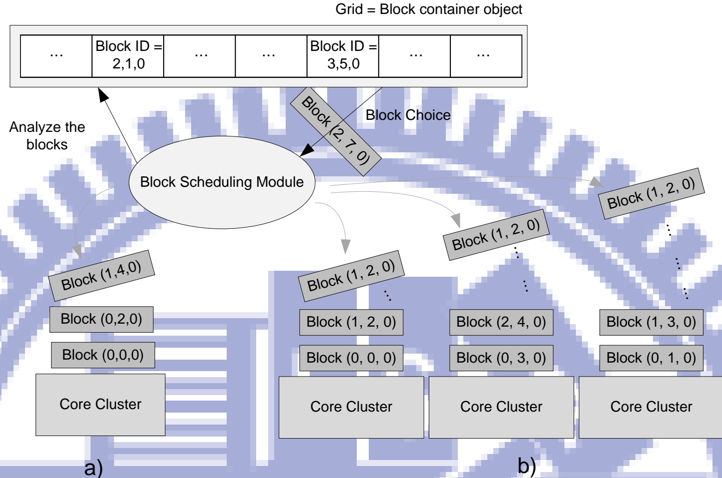


Figure 18: Block scheduling module and block scheduling flow (a) Block scheduling module assigning blocks to a system with one core cluster. (b) Block scheduling module assigning blocks to a system with multiple core clusters

Figure 18(a) shows the situation for the case in which there is only one core cluster, which we represent within the framework as abstraction that we call concurrency slot, defined shortly. In this case, the block fetched from the grid object enters at the tail of the queue, modeling a sequential execution. In Figure 17(b), a similar case is illustrated for when multiple blocks are able to execute in parallel, either by allowing many core clusters or by allowing one single core cluster to handle multiple blocks in parallel i.e. when the number of elements in the array of concurrency slots is more than one. In this case, the blocks can be allocated to each core cluster in such a way to prevent workload imbalance, or other type of optimization. This scheduling feature becomes useful for the Scenarios 3~6 of the data reuse characterization.

The block scheduling module can implement any particular policy desired. This module is able to perform its required analysis over the grid object, and the blocks within it. The purpose of such analysis is to determine which block is the best candidate to queue for execution to improve over a certain metric, possibly data reuse or any other. Once the block is chosen it is

read from the grid object, and allocated to a specific concurrency slot. Since the blocks in the concurrency arrays are not being merged, each slot maintains the blocks serialized. Every slot represents a block FIFO queue that models the sequencing of the scheduled blocks.

Algorithm 3 shows the pseudo-code for the block scheduling module. Line 1 declares a two-dimensional array labeled '*currBlocks*' that holds arrays of '*blockArray*' objects. The two cases illustrated in Figure 18 are considered by the scheduler. The analyses that will model execution over a realistic number of cores are Scenarios 5 and 6. But Scenarios 4 and 3 do not require core clusters nor core cluster modeling, since they only employ the array of concurrency slots and the resulting aggregate blocks. These are the four scenarios that are able to parallelize blocks. Scenarios 1 and 2 do not possess this feature, making the scheduling module trivial in these cases.

Algorithm 3. BlockSched(CCNum, CC, policy, parBlocks)

```
// GO: grid object, CCNum: core cluster number,
// CC: array of core clusters, parBlocks: number of
// parallel blocks, policy: scheduling policy
1.   blockArray currBlocks[][];
2.   j=0;
3.   while (GO.size > 0 AND (var = 5 OR var = 6) )
4.      for i=0 to CCNum-1
5.          tempBlock = BlockExtract(GO, policy);
6.          CC[i].blocks.insert[tempBlock];
7.          currBlocks[j].insert(tempBlock);
8.      j++;
9.
10.  while (GO.size > 0 AND (var = 4 OR var = 3) )
11.     for k = 0 to parBlock-1s;
12.         tempBlock = BlockExtract(GO, policy);
13.         currBlocks[j].insert(tempBlock);
14.     j++;
15.
16.   Function BlockExtract(grid, policy)
17.  blockObject candidate;
18.  candidate = scheAnalysis(grid, policy);
19.  grid.erase(candidate);
```

```
20. return candidate;
```

The code in lines 3~7 schedule blocks to each of the '*CCNum*' core clusters. The *while* loop in lines 3 first checks that the scenarios are the correct ones. Line 4 has a *for* loop that iterates over the core clusters. The function in line 5 extracts a block from the grid object ('*GO*') according to the block scheduling policy specified by '*policy*'. The extracted block is assigned to an instance of the block class labeled '*tempBlock*'. Once the block has been extracted from the grid, line 6 inserts it in the block FIFO queue belonging to the specific core cluster. In line 7, '*tempBlock*' is also inserted in the entry '*j*' of '*currBlocks*'. This object is the array of concurrent slots. The reason that both of these structures are populated in this stage is because Scenario 6 will require the array of concurrent slots to be converted into aggregate blocks.

The code in lines 10~14 schedule blocks for Scenarios 4 and 3. These analyses do not employ the models of core clusters. For this case, it is necessary to set the size of the arrays of concurrency slots, and populate the arrays with blocks. Line 10 makes sure that the Scenarios are the appropriate ones, as was the case for line 3. The *for* loop in line 11 will execute the instructions in line 12 and line 13, which select the block from the grid object and introduces it in '*currBlocks*'. The resulting state after executing of **Algorithm 3** is shown in Figure 19.

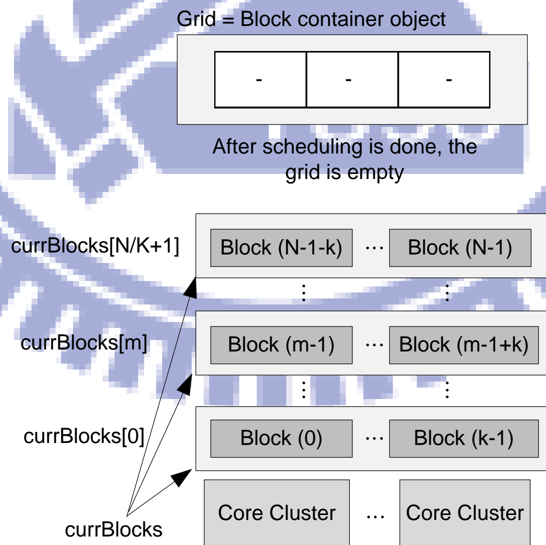


Figure 19: After assigning blocks to the core clusters. The blocks are queued in each cluster, and are also inserted into the '*currBlocks*' structure, which represents the arrays of concurrent slots.

The function *BlockExtract()* chooses a block from the grid and passes it back to the calling function. Line 17 declares an instance of the block class called '*candidate*'. The *scheAnalysis*() function called in line 18 is responsible to iterate over the grid object, and choose the best block according to the policy. Once the grid is analyzed and the block is chosen, it is assigned to '*candidate*'. Immediately after that, the entry belonging to the chosen block is erased from the grid, and '*candidate*' is returned to the calling function. Within the *scheAnalysis*() function, it is possible to implement many blocknscheduling policies, and analyze its effects on the data reuse characteristic.

In this work, however, we implement a simple sequential scheduling policy. That is, the blocks are chosen in the order in which the trace generator tool executed them. A thorough analysis of the impact of different block scheduling policies over the data reuse characteristic is left for future work.

7.2.3 Core cluster modeling

Modeling certain architectural characteristic of the core clusters becomes important for Scenarios 5~6. For this case, a thorough cycle-accurate modeling of the core cluster behavior is not necessary, since our focus is on the memory access behavior of the application itself, abstracted from timing details.

In order to comprehend why such thorough modeling is *not* necessary, we describe a simple example. When a core cluster executes a *sqrt* math instruction, it is after fetching the necessary data from global memory. Then, the functional unit within the core cluster reads the operands from the registers, and consumes a certain number of clock cycles performing the relevant mathematical operations. When a new memory instruction is issued, its ordering within the instruction stream will not be affected by the number of cycles consumed by the *sqrt* instruction, only by the previous memory access. Likewise, the memory latency taken by the MIs will not have an impact in the sequencing of MIs.

In this work, we are only evaluating the impact of the parallelism limitation over the data reuse characteristic of the applications. Therefore, we model the memory subsystem as ideal, and also considering unlimited functional units, only focusing on the data utilization behavior.

However, there are two particular resources within a core cluster that will have significant effect on the sequence of the memory accesses: the load/store units and the warp scheduler. The number of load/store units available to each warp in a given cycle plays a significant role on the ordering of MIs within the reference stream. This will create a dis-adjustment on the data reuse characteristic when compared to the ideal case. The effect can be illustrated by a simple example.

Let's assume there are 'x' number of load/store units and 'y' threads requesting data from memory simultaneously. If $x > y$, then all the memory requests can be serviced simultaneously. However, when $x < y$, the memory requests will not be issued all at once. This makes necessary for a subset of the threads in a warp to be issued, instead of all the requesting threads. The result is a larger number of MIs in the reference stream. If to this we couple the effects of the warp scheduler, which is the one responsible of issuing warps for execution, the impact on the reuse characteristic is two-fold. If there are multiple warps available for execution in a given clock cycle, then choosing warps that present significant data reuse among each other will definitely help to improve performance. Choosing warps based on the way the make use of the data could improve coalescing, avoid bank conflicts and improve overall runtime performance.

The Locality Analyzer enables the inclusion of more novel block scheduling policies at the warp scheduling level. However, in this work, we have only implemented a simple round-robin policy to ensure fairness among active threads. The scheduling policy at this level becomes relevant for Scenarios 5 and 6. Warps are modeled in the a similar way as blocks. Every warp object within the framework has information on their number of threads, the address array and to which block they belong to. A thorough analysis of the impact of warp scheduling techniques over the data reuse characteristic is out of the scope of this work.

7.2.4 Merging of reference streams

When multiple blocks can be executed in parallel, it's necessary to merge the reference streams into an aggregate stream in order to perform analyses efficiently. Scenarios 3, 4 and 6 of the data reuse characterization requires for this task to be performed in advance.

The block scheduling policy chooses which blocks will execute concurrently. As the blocks are popped from the grid object, they are merged in an empty reference stream i.e. an aggregate reference stream. This is done by creating an additional structure that absorbs and collapses the reference stream of the concurrency slots. Initially, this data structure is empty. When the first block is chosen, its reference stream is analyzed. The aggregate reference stream is filled with the MIs of the first block, creating an identical copy of the first block in the new data structure.

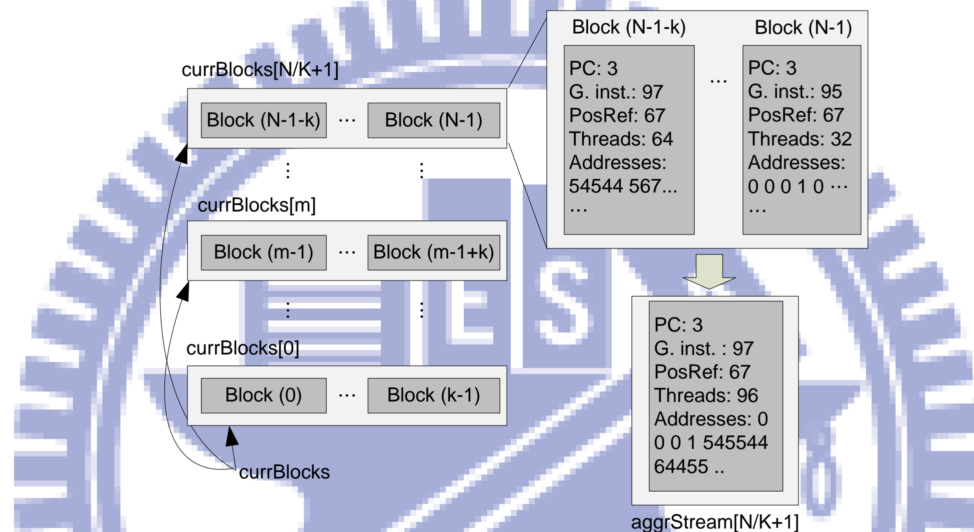


Figure 20: Merging of reference streams from multiple blocks in the arrays of concurrent slots. Scenarios 3, 4, and 6 are the only ones that employ this procedure.

Figure 20 illustrates the merging of reference streams from multiple blocks. When the sub-sequent blocks are chosen, a similar procedure occurs, but with some modifications. The MIs within each block need to be included in the aggregate stream, but new entries cannot re-write the entries introduced by the previous blocks. However, if a new block has an MI that falls into the relative position of an MI introduced by a previous block, additional procedures occur.

Algorithm 4. StreamMerge(currBlocks)

```
// GO: grid object
1.   aggrStream[].clear;
2.   for j=0 to currBlocks.size-1
3.       for m=0 to currBlocks[j].size-1
4.          for k=0 to currBlocks[j].block[m].streamSize
```

```
5.      instInfo inst;
6.      inst = currBlocks[j].block[m].instStream[k];
7.      is = aggrStream[j].isThere(inst.posRef);
8.      if (is)
9.          aggrStream[j].threads += inst.threads;
10.         for i = 0 to inst.addrArray.size
11.           bool addrPres =
              aggrStream[j].addrArray.isThere(inst.addrArray[i];
12.           if (addrPres)
13.             aggrStream[j].addrArray[ inst.addrArray[i] ] =+
                inst.addrArray[i].mult;
14.           else
15.             aggrStream[j].addrArray.insert( inst.addrArray[i],
                inst.addrArray[i].mult );
16.     else
17.         aggrStream[j].insert(inst.posRef, inst)
```

Algorithm 4 shows the pseudo-code for this procedure. Line 1 clears the '*aggrStream'* structure array. Each entry in this array will hold the different aggregate streams corresponding to each array of concurrent slots. The *for* loop in line 2 will iterate over the '*currBlocks'* object. Recall that '*currBlocks'* is an array of concurrency slots, populated in **Algorithm 3**. Every entry in this array holds the blocks that issued instructions concurrently. Line 3 will iterate over the concurrency slots in the queue, and line 4 will iterate over the streams of each block in the respective '*currBlocks*' entry. Line 5 creates an instance called '*inst*' of the '*instInfo*' class that holds all the information of that specific MI. Line 6 reads the instruction '*k*' of the stream belonging to the block analyzed in the mth iteration. Then, line 7 checks if there's an instruction in the aggregate stream '*aggrStream*', corresponding to the '*j*' aggregate block, with the same position index than the '*inst'* instruction. If no previous entry exists in that position, then line 17 executes, inserting the instruction in the slot corresponding to its position index '*inst.posInRefStream'*. In case a previous entry exists with the same position index, then an additional procedure needs to be performed.

For the case where *is == TRUE* i.e. a previous entry exists, it is necessary to combine the information of '*inst'* with the data already present in the corresponding entry of '*aggrStream*'. Line 9 will first increment the number of threads that issued that instruction. The *for* loop in line 10 will then iterate over the address array of '*inst*', in order to merge it with the array

already present in the corresponding entry of '*aggrStream*'. Line 11 will first look if the addresses are already present in the address array of '*aggrStream*'. In case there is not a previous entry, the instruction in the *else* clause of line 14 are executed. In this case, '*inst'* is inserted directly into '*aggrStream*'. When an MI of one block seeks a specific position within the aggregate stream, then the new MI is allocated. The information stored with each entry in the aggregate reference stream is identical to the one stored in the entry of the MI in its original reference stream, with the exception of the block ID, which is not necessary. Line 17 executes this task.

When a new MI requires a slot occupied by an MI previously entered, and both have addresses in common, the multiplicity of the common addresses will increase. The final value is the multiplicity accumulated by the MI in the aggregate stream of the specific address plus the multiplicity of that same address in the new MI. This can be expressed mathematically as follows:

$$M_{Agg}(A) = M_{Agg}(A) + M_{ID-i}(A) \qquad Eq.2$$

where $M_{Agg}(\cdot)$ is the aggregate multiplicity of a specific address in the aggregate reference stream, and $M_{ID-i}(\cdot)$ is the multiplicity of a specific address in the new MI '*i*' that belongs to the block identified with '*ID*'. This value is accumulated in line 13 of **Algorithm 4**.

The number of aggregate streams created will depend on the amount of blocks that application generates and the number of blocks that are executed in parallel. Expressed in a mathematic way:

$$\#\text{Agg. Streams} = \left\lceil \frac{N}{k} - 1 \right\rceil \qquad \text{Eq. 3}$$

where '*N*' is the number of blocks of the application, and '*k*' is the number of blocks running in parallel. Notice the $\lceil \cdot \rceil$ bracket, which rounds up the quotient to the highest integer in case of a non-exact division.

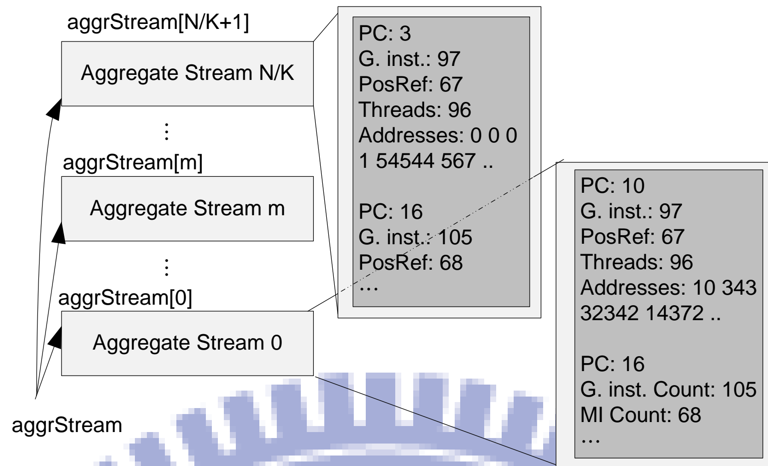


Figure 21: The resulting reference stream of the aggregate blocks. It is possible to compare this with the streams shown in Figure 19. Notice how the 'Addresses' and 'Threads' fields are augmented

Figure 21 illustrates the resulting state after executing **Algorithm 4.**

7.2.5 Adjusting the position index of Mis

Once the aggregate streams have been created, they are serialized with respect to each other in the order in which they are issued according to the previous procedure. A new data structure was created in **Algorithm 4** that holds entries for each aggregate stream. After this, an additional procedure needs to take place. Each aggregate stream has an ordering of the MIs valid only within itself. It is necessary to modify the values of the position index within the stream to reorder the MIs with respect to their homologous in the rest of the aggregate streams, since these are serialized. This procedure is shown in **Algorithm 5**.

Algorithm 5. posRefAdjust(aggrStream)

```
// GO: grid object
1.  posMI = 0;
2.  for i = 0 to aggrStream;
3.     for j=0 to aggrStream[i].streamSize
4.         aggrStream[i].inst[j]++;
5.         posMI++;
```

Algorithm 5 takes as input the aggregate reference stream. It iterates over all of its entries, as can be seen by the *for* loop in lines 2. The nested loop in line 3 will iterate over the streams of each entry in '*aggrStream*'. The sequencing of the aggregate streams is taken into consideration. The data structure that holds the streams is traversed with an incrementing variable that modifies the position index of each MI inside the stream. For the first aggregate stream in the sequence, the position index of the MIs remains identical. However, for the rest of the streams, the position index is assigned the value of the incrementing variable, which is basically a global count of MIs. The result after adjusting the position index is shown in Figure 22.

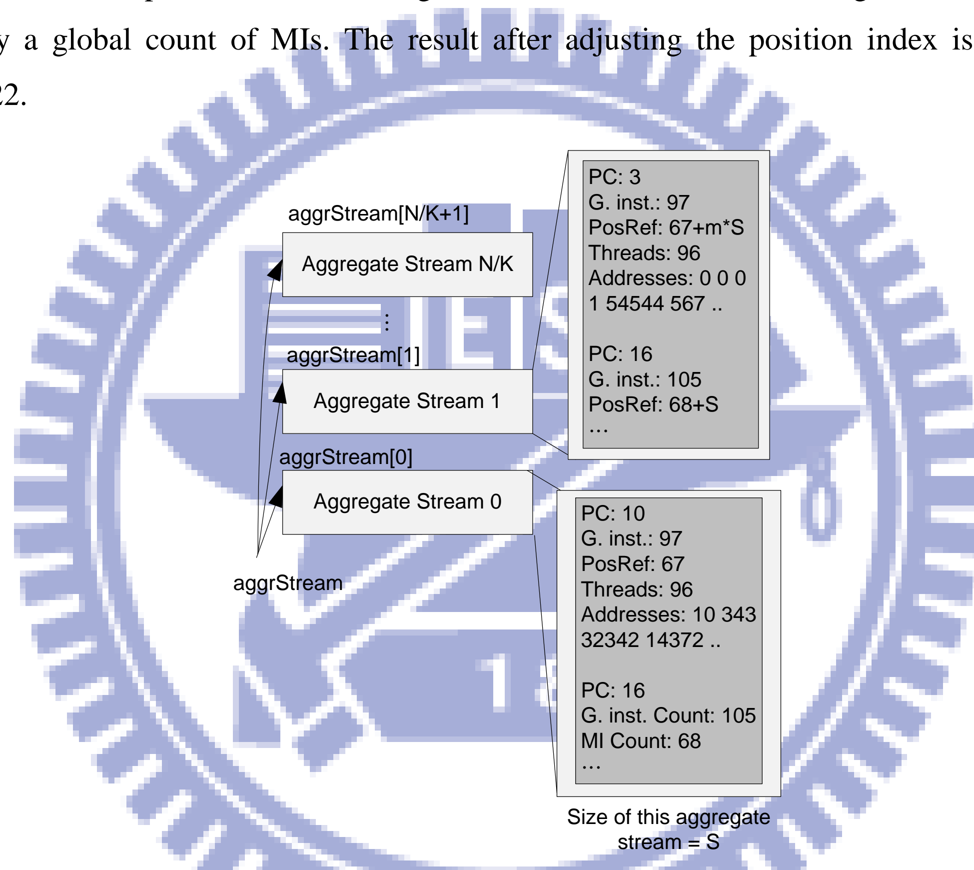


Figure 22: Modifications of the position reference index within the reference stream. The value of the index of the MIs in streams other than the first will depend on the stream length of the streams before the current one.

Notice in Figure 22 that the position index of the MIs in the Aggregate Stream 1 are now offset by '*S*', which is the stream size, or the number of MIs, of the previous stream. The resulting offset for the rest of the MIs in the a specific aggregate streams will depend on the stream sizes the previous aggregate streams

7.2.6 Locality Analyzer Architecture: Putting it all together

Figure 23 shows a flow chart that represents the architecture of the Locality Analyzer. As this figure shows, all of the modules described so far are included. The associated algorithms have been developed in the previous subsection. The methodology is clearly illustrated.

The most significant properties of this architecture are its flexibility, scalability and the model the kernel's runtime. The focus of this methodology is to understand the locality behavior of the applications dissociated from the details of particulars SIMT processors. The modeling of the programming abstractions within the Locality Analyzer allows developers to manipulate them in any way they see fit. In this way, the need for long cycle-accurate simulations is reduced, extracting efficiently locality information.

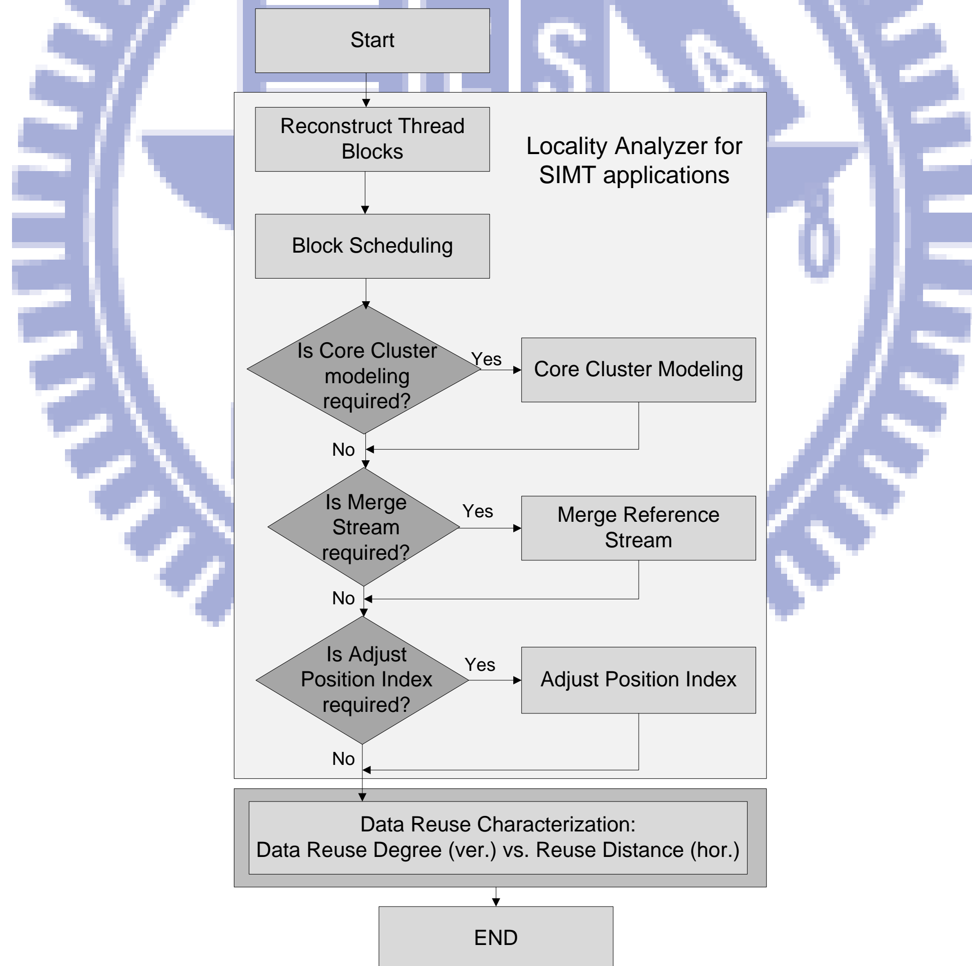


Figure 23: Block diagram of the architecture of the Locality Analyzer. This is a very general and simplified version of our framework

Keep in mind that while the SIMT processors evolve and scale very quickly, the structures of the applications for which they are used do not change as rapidly. When abstracting the applications' analyses from the particulars of each processor, the validity of the analysis is maintained across different families and types of SIMT processors, as long as the programming and runtime abstractions do not change. The analyses therefore have an important degree of generality and usability across different generations of architectures.

The architecture of the Locality Analyzer allows for many aspects related to control flow to be thoroughly analyzed, as well as the modeling of the certain architectural components (schedulers, core clusters, load/store units). Other types of analyses can be easily coded due to the modularity characteristic the architecture possesses. However, in this work, we focus on the resource limitations, coding optimizations and their impact on the data reuse characteristic of different applications. This will allow us to get further insight on the way the data reuse characteristic changes when coding optimizations, coupled with the details of the execution model, are applied to applications. This would be an important step to predict the performance improvement that could be gained from applying optimization techniques when running on architectures with different characteristics.

VIII. APPLICATION OPTIMIZATION

This section gives a brief explanation on the coding optimizations applied to the applications in our experiments. By analyzing the changes on the data reuse characteristic after optimization, assuming invariable the resources to exploit parallelism, it is possible to compare the effectiveness of the optimization procedures. These coding optimizations follow the thread mapping methodology for multi-level shared caches proposed in [9]. This methodology performs different optimization procedures over the application: thread clustering, warp clustering and block scheduling.

8.1 Thread Mapping Methodology

The coding optimizations are implemented as a stand-alone library. The procedure is performed by the CPU host. The core of the procedure consists on manipulating the threads within one kernel, modifying the baseline order of execution. To do this, it is necessary to first analyze the data accesses of an application. For this purpose, in [9] the compressed sparse row (CSR) format is used, since it facilitates the manipulation of the data accesses.

The thread mapping on the GPU side is accomplished by applying similar techniques to those of data and computation reordering [9]. Since GPUs do not allow programmers to explicitly schedule threads or thread blocks for execution, then it is necessary to implement an indirect way to mimic this behavior. Thus, the methodology makes use of appropriate data layout techniques coupled with the re-mapping of thread indexes [9]. The way to do this is basically to re-arrange the data contained in the data structures and associated arrays (array of structures) for threads to then access the data in a different sequence as initially expected [9]. A new thread mapping array is constructed that maps the baseline thread index of each thread to a different value, equivalent to a function that can be expressed as:

$$\mathrm{TIdx} = \mathrm{f(threadIdx)} \qquad Eq.4$$

where $'TIdx'$ represents the new thread index, '$threadIdx$' represents the baseline thread index assigned by the GPU, and *f*() is the function that performs the mapping.

By making use of these two ordering techniques, it is possible to manipulate the formation of warps and blocks, and mimic alterations in the order in which they are issued for execution. The resulting behavior is to coordinate better the accesses in order to exploit the benefits of data reuse [9].

The methodology proposed by [9] consists on the following steps:

1. Generate information about the data utilization and architectural parameters of the SIMT processor: to obtain information about the data utilization, a data sharing and volume graph is generated. The task of gathering architectural parameters of the SIMT processor is trivial for our purposes.
2. Thread and Warp Clustering: threads are grouped into warps such that the resulting warps issue the minimum number of memory transactions.
3. Thread Block Scheduling: tries to schedule in adjacent issue slots the thread blocks that present significant data sharing.
4. Resource utilization throttling stage: the depth of multithreading is throttled to use the shared cache and avoid contention.

The data sharing and volume between the threads both are modeled as a hypergraph called Data Sharing and Volume Graph (DSVG), define as [9]:

$$\mathrm{DSVG} = \mathrm{HG}(\mathrm{V}, \mathrm{E}, w_v, w_e) \qquad Eq. 5$$

in which 'V' is the set of vertices, 'E' is the set of hyperedges, 'w_v' is the vertex weight, and 'w_e' is the weight of the hyperedge. In this model, threads are represented as vertices within the DSVG, and the associated weight represents the amount of private data of the thread. A hyperedge represents threads that share data, and the weight associated with the hyperedge indicates the amount of shared data. The data sharing within a set of vertices i.e. a set of threads, is represented by the group of hyperedges incident to a vertex. Likewise, the data sharing involving other sets correspond to external hyperedges relative to a given set.

Thread clustering forms warps of threads so to minimize memory transactions. The warp clustering step builds blocks based on these newly created warps into thread blocks with an

increased amount of data sharing. The Thread Clustering and Warp Clustering techniques can be reduced to the hypergraph partitioning problem, a well-known NP-hard problem [9].

Thread Block Scheduling arranges the issuing order of the blocks with the objective to reuse data through the L2 cache. Using the projection of the result generated by the Thread and Warp Clustering, a DSVG is obtained that enables a formulation of the Thread Block Scheduling Problem. In this case, each vertex in the new $DSVG = HG(V, E, w_v, w_e)$ represents a thread block, and a new function is defined that maps each vertex to an integer, in a one-to-one relationship. The integers represent the scheduling sequence.

As previously explained, the reuse distance is the number of distinct memory accesses between references to the same shared data. The Thread Block Scheduling uses this definition to generate a new metric: the total reuse distance. This metric is the sum of all the reuse distances that appear as a result of a specific scheduling function applied over the vertices of the hypergraph. Therefore, a mapping function needs to be selected to minimize the value of that sum. The Thread Block Scheduling problem is actually a general version of the vertex ordering problem [9].

The fourth stage of the methodology, Resource Throttling, finds the best way to utilize the last level shared cache [9], currently the L2 cache in SIMT processors. This last step is not included in our experiments because we use model the memory subsystem as ideal. As previously explained, this work only models the aspects of the core cluster that have impact on the ordering of the MIs with an ideal memory subsystem.

IX. EXPERIMENTAL RESULTS

In this section we present the data reuse characteristic of a series of applications from different domains. The applications chosen are considered irregular because their data sharing behavior is not totally similar between threads in a block and depends substantially on the input data set size. We use the Electronic Design Applications (EDA) used in [18]. The input to these applications is taken from ITC'99 circuit suite [19]. We use only one input for the applications in this set in order to make a better comparison of the reuse characteristics under different parallelism capabilities and coding optimizations. Also, we use one application from the NVIDIA SDK suite. Irregular applications from the Chaos group [20] and the COSMIC project [21] are also used. Table 1 summarizes the set of applications used in this work.

TABLE I

APPLICATIONS USED FOR EXPERIMENTATION

Apps.	Source	Description	Inputs	Kernel Num.
sta	EDA	Static timing analysis (STA)	32160 nodes, 63497 edges (*b17* from ITC' 99)	2
gsim	EDA	Gate level logic simulation		2
bfs	EDA	Breadth First Search		2
vectoradd	NVIDIA SDK	Vector addition	Internal	1
nbf	COSMIC	Molecular dynamics	144,649 nodes, 1,074,393 edges (*foil* from COSMIC)	2
moldyn	COSMIC	Molecular dynamics		2
irreg	COSMIC	Partial differential equations		2
euler	CHAOS	Finite-difference estimations on Eulerian Mesh		1

9.1 Data Reuse Characteristic with serialized blocks and on a per block basis

These are the results corresponding to Scenario 1 detailed in Section 6.1. Figures 24~27 shows the data reuse characteristic for *sta*, *gsim*, *bfs* and *vectoradd*, respectively. The rest of

the applications (*nbf, moldyn, irreg, euler*) were not analyzed for the case of serialized blocks because the running time for such analysis was prohibitively long given the input set size.. There is striking similarity between *bfs* and *sta*, both from the EDA benchmark suite that becomes visible when observing Figures 24(c) and 26(c). When looking at the source codes, presented in Figure 28, it is possible to appreciate that the code of the kernels is almost identical, with identical access patterns, even though the operations executed differ. Thus, it is expected for the reuse characteristic to be almost identical. This observation demonstrates the consistency of our model and methodology, since they can capture the same behavior consistently across applications with similar data utilization patterns.

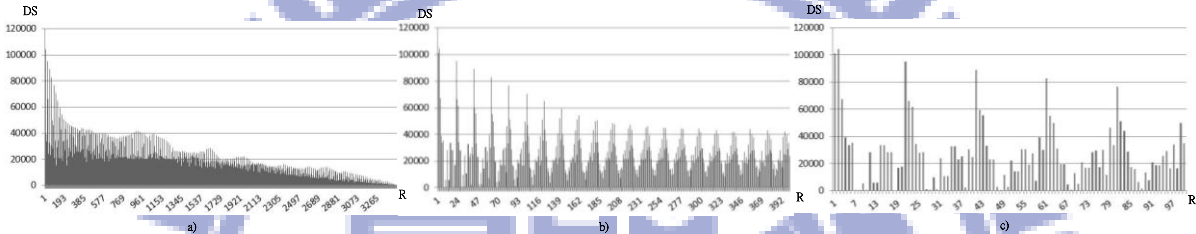


Figure 24: Data reuse characteristic when modeling block execution sequentially for *sta*. (a) Full reuse characteristic. (b) Showing the reuse characteristic for the range *RD={1, 400}*. (c) Showing the reuse characteristic for the range *RD={1, 100}*. Notice the particular patterns.

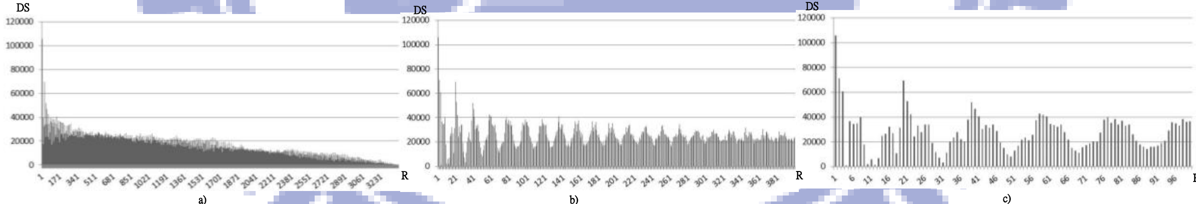


Figure 25: Data reuse characteristic when modeling block execution sequentially for *gsim*. (a) Full reuse characteristic. (b) Showing the reuse characteristic for the range *RD={1, 400}*. (c) Showing the reuse characteristic for the range *RD={1, 100}*.

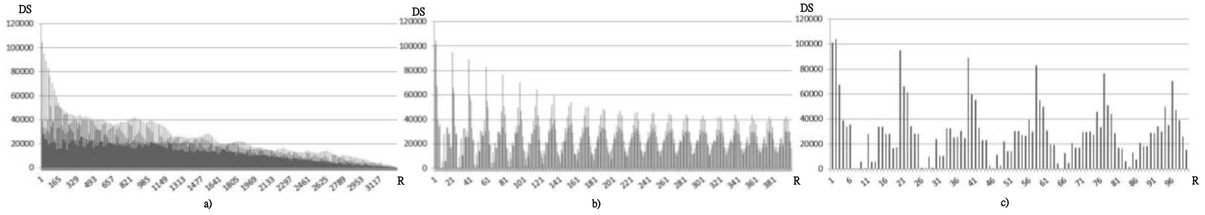


Figure 26: Data reuse characteristic when modeling block execution sequentially for *bfs*. (a) Full reuse characteristic. (b) Showing the reuse characteristic for the range *RD={1, 400}*. (c) Showing the reuse characteristic for the range *RD={1, 100}*.

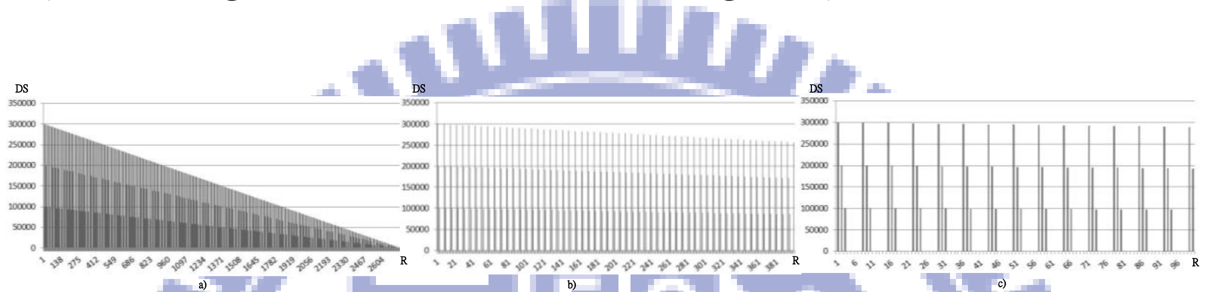


Figure 27: Data reuse characteristic when modeling block execution sequentially for *vectoradd*. (a) Full reuse characteristic. (b) Showing the reuse characteristic for the range *RD={1, 400}*. (c) Showing the reuse characteristic for the range *RD={1, 100}*.

```
unsigned gid = blockIdx.x * blockDim.x + threadIdx.x;
if (gid >= ngates)
    return;

Id adjb = ddata[DataIdx];
Id adje = ddata[DataIdx];

Data arrival;

for ( unsigned aid = adj;  aid < adje;  aid++ )
{
        adj = dadjs[aid];
        arrival = arrival || ddata[DataIdx];
}
```

a)

```
unsigned gid = blockIdx.x * blockDim.x + threadIdx.x;
if (gid >= ngates)
    return;

Id adjb = ddata[DataIdx];
Id adje = ddata[DataIdx];

Data arrival;

for ( unsigned aid = adj;  aid < adje;  aid++ )
{
        adj = dadjs[aid];
        arrival = max(arrival, ddata[DataIdx]);
}
```

b)

Figure 28: Source code for the kernels of *bfs* (a) and *sta* (b).

Also, notice that the applications presented above have a reuse characteristic with periodical behavior in the reuse distance domain. In *sta* (Figure 24) and *vectoradd* (Figure 27) it is possible to observe multiple super-imposed patterns that yield a very particular reuse characteristic for the application. Recall that, in these first set of charts, the thread blocks are treated as running in sequence.

In order to provide more insight into the reuse characteristics of Figure 24 ~ 27, Figures 29~36 show the reuse characteristic obtained on a per block basis corresponding to Scenario 2 of our analysis. We also show the results for *nbf*, *moldyn*, *irreg* and *euler*, which we will reference in subsequent sections. Notice that for *sta*, *gsim*, *bfs* and *vectoradd*, there is a significant similarity between the reuse characteristic of each block, and the reuse characteristic obtained when serializing the blocks. These figures show the reuse characteristic for two blocks out of more than a hundred that their corresponding kernels have. Notice that, as expected, the reuse characteristic of blocks from *sta* and *bfs* are almost identical to each other. They differ on the magnitudes of their reuse degree, but the contour (or shape) of the reuse characteristic is maintained across most of the blocks.

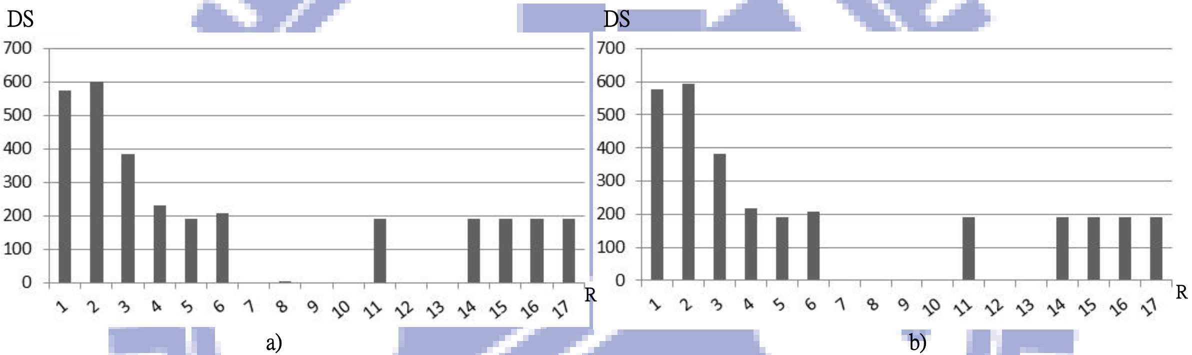


Figure 29: Data reuse characteristic on a per block basis for *sta*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1.

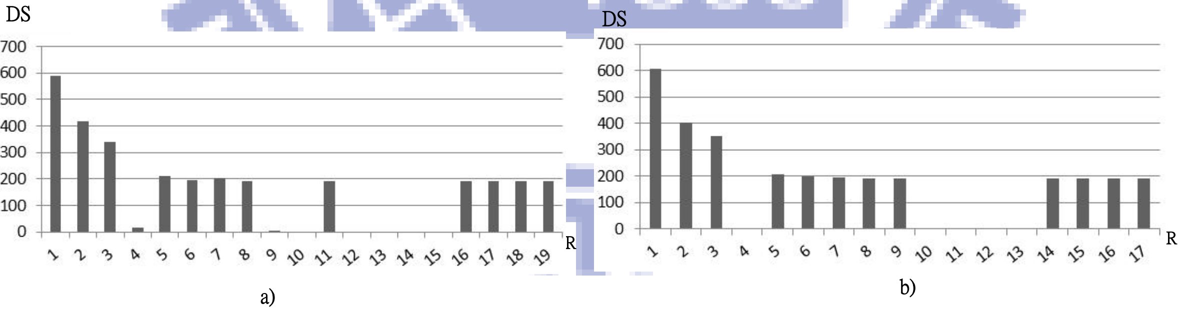


Figure 30: Data reuse characteristic on a per block basis for *gsim*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1.

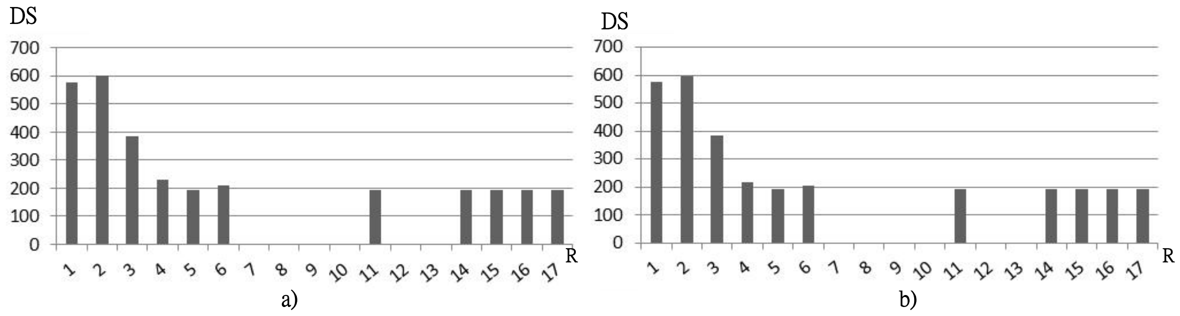


Figure 31: Data reuse characteristic on a per block basis for *bfs*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1.

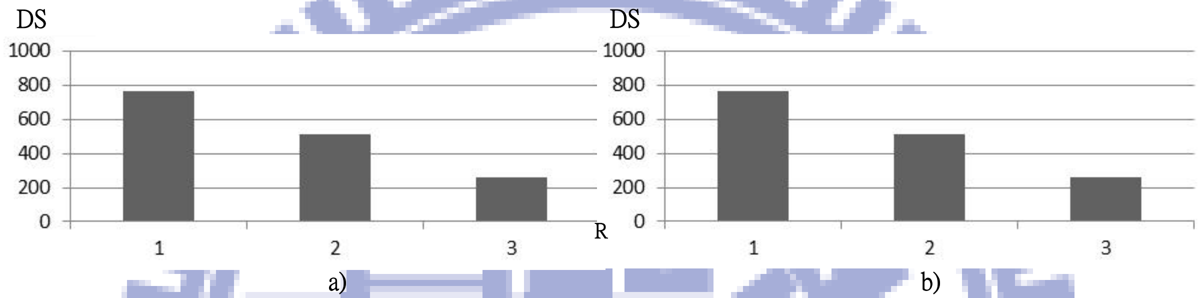


Figure 32: Data reuse characteristic on a per block basis for *vectoradd*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1.

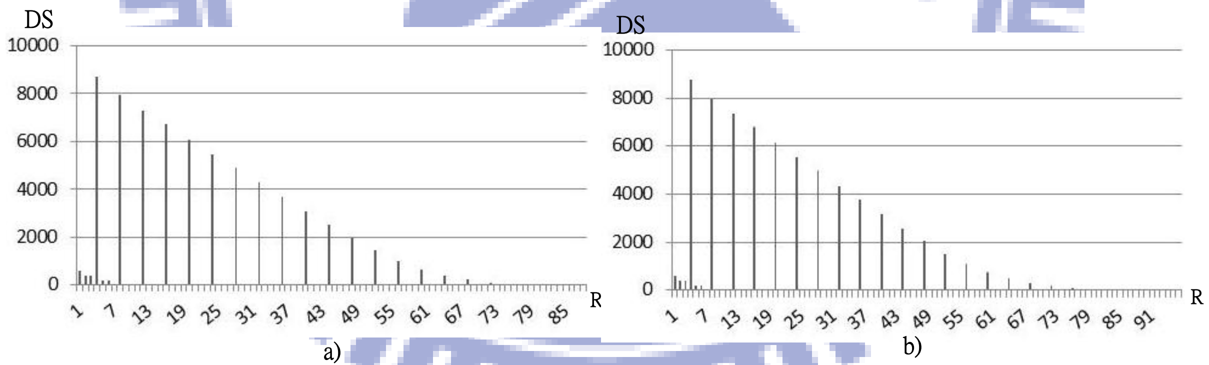


Figure 33: Data reuse characteristic on a per block basis for *nbf*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1.

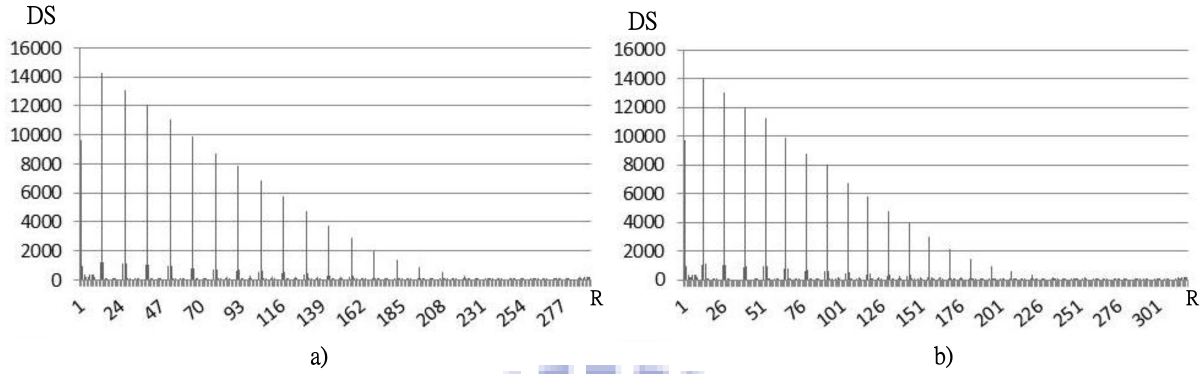


Figure 34: Data reuse characteristic on a per block basis for *moldyn*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1.

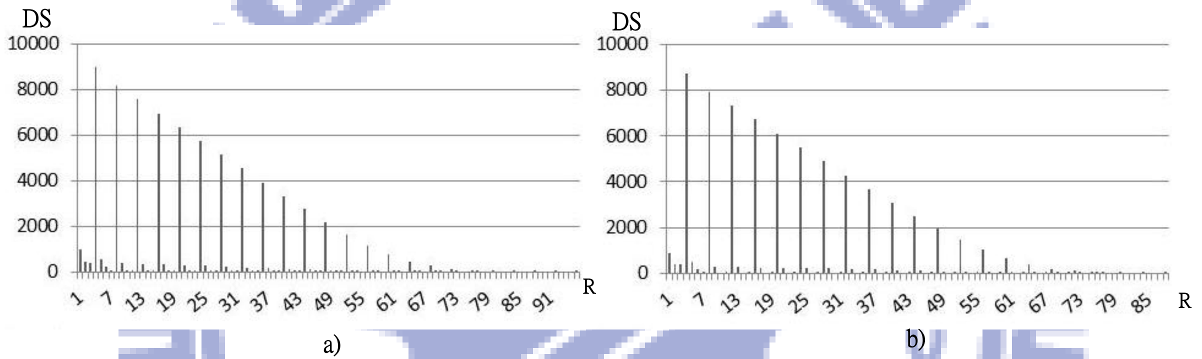


Figure 35: Data reuse characteristic on a per block basis for *irreg*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1.

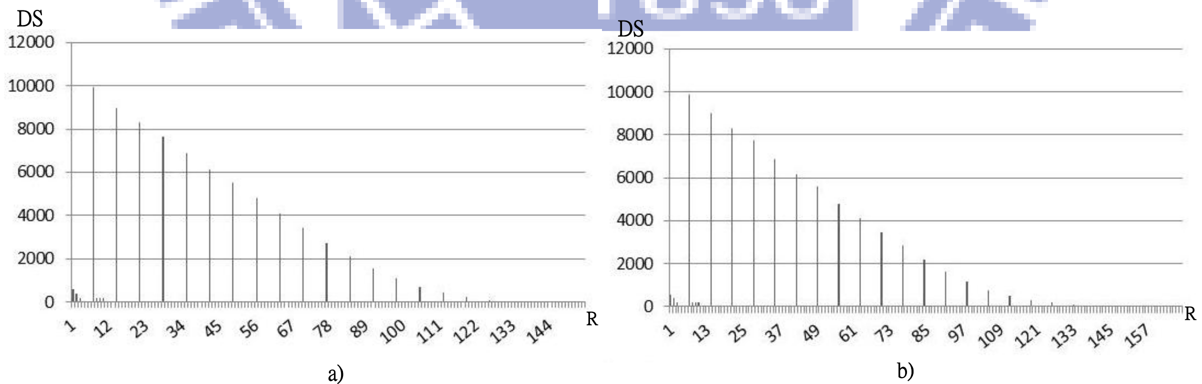


Figure 36: Data reuse characteristic on a per block basis for *euler*. (a) Data reuse characteristic for thread block 0. (b) Full reuse characteristic for thread block 1.

An illustrative case is *vectoradd*, shown in Figure 31. Here, the reuse characteristic for two of its thread blocks is presented. Notice that each block only reuses data up to a distance *RD = 3*, and presents a monotonically decreasing profile. In Figure 27, the maximum distance up to which data is reused for this benchmark is increased by more than a 800 times. To be exact, the maximum reuse distance for the case when the execution of the blocks is modeled in a serialized way is *RD = 2733*. This comparison proofs that there is a significant, not negligible, amount of data reuse between blocks of the kernel of *vectoradd*. When observing the rest of the reuse characteristics for the blocks in Figures 29~32, and comparing them to their homologous in the charts of Scenario 1, we can see that most of these applications do present significant data sharing among blocks. This is one of the most important observations that can be extracted when analyzing the results from Scenarios 1 and 2.

Note that there are some kernels in which the blocks present a monotonically decreasing behavior (MB) in data reuse characteristic such as *vectoradd* in Figure 32. Other kernels present an approximate monotonically decreasing (AMB) profile. For these kernels, we can differentiate two specific groups. The first group of kernels present a peak when RD = 2, and from there on, the data reuse degree at subsequent distances decrease in a fairly uniform way. In this group, we can include *sta* and *bfs*, in Figures 29 and 31, respectively. The second group of kernels that we differentiate present a peak at *RD = 1*, with a decrementing behavior similar to that of the first group. In this group, it is possible to include *gsim*, in Figure 30. Both of these cases do not strictly decrement monotonically given that the monotonicity is broken for certain reuse distances, with *DS = 0* or with a value smaller than those of subsequent distances. But the nearly uniform decreasing behavior is significant when comparing the data reuse characteristic with other applications that present totally different behavior, as we shall see.

When a data reuse degree peaks at *RD = 1*, means that the highest reuse degree occurs between adjacent MIs. But this does not mean that every two adjacent MIs present data reuse. Rather it means that, when considering all of the MIs (not two particular adjacent MIs) in the reference stream, most of the reuse degree occurs between contiguous MIs. In contrast, when the peak is at *RD = 2*, means that the highest degree of reuse, when considering all MIs in the reference stream, occur every other MI.

There are also kernels in which the blocks do not present any monotonic (NMB) behavior in their domain, neither a periodic behavior. To this group belong the majority of the kernels in

the applications analyzed in this work: *nbf*, *moldyn*, *irreg*, and *euler,* presented in Figures 33~36. These kernels present an oscillatory behavior in their reuse characteristic. These kernels present significant variability among its blocks and an oscillatory behavior in their reuse characteristic. It reuses data at longer distances than *sta*, *gsim* or *bfs*. Moreover, the peak magnitudes of the data reuse degree is at distances larger than 1 by an order of magnitude.

Even though the scope of this paper is not to find the correlation between the data reuse characteristic and the performance seen in real architectures, we still would like to make some comments regarding the results seen. When a block reuses data after so many memory instructions, the impact on performance can be very significant when running on a real SIMT architecture with cache memory, but it will depend on the way application reuses data. There are two possible cases that can occur: 1) the capacity of the cache memory is not completely used and the data fetched is used with a varying degree at different distances as execution progresses, or 2) the data allocated in the cache memory by a specific MI is eviction by a different MI before it is reused again by another instruction with a specific distance apart. Contention is present if the eviction of the data reused is very frequent in the reference stream.

In the first case, performance is increased when the cache is present as long as the whole data for a specific portion of the execution is present, and not reused again for other instructions in the stream before being evicted. The second case is more complex. Given that if the same data is needed again by one instruction in the block many memory instructions later (for example, more than 250 MIs as in the case for *moldyn*), it is possible that the data will have already been evicted from the cache. To determine whether or not this will occur, it is necessary to analyze the total amount of memory allocated before such memory access occurs from the MI that fetched the data the first time and on the analysis of the application's access patterns data. Kernels may in fact reuse data at such long distances, and this fact justifies the realization of sub-sequent analyses to establish the relationship with performance and the effectiveness of various optimization techniques over the data reuse characteristic.

An ideal reuse characteristic should be monotonically decreasing, implying that its maximum peak has to be at *RD=1*, and should also decrease its data sharing at high rate as the reuse distance grows. Therefore, it is desirable that kernels present a very small reuse distance domain i.e range of reuse distances. This is so because, in the best case, adjacent MIs will share the highest amount of data, with the sharing degree reducing its magnitude as distances

increase i.e. as execution progresses. This will reduce the possibility that data is fetched again in case a prior eviction occurs.

However, just the fact of having reuse degree at far away distances can also have an impact on performance irrespective of the magnitude. If for example there's only a single datum, reused at a far distance apart, that has been evicted by an intermediate MI, on a real architecture, the data will have to be fetched only to be used by a single thread, affecting the total throughput of system. Therefore, even though these reuse degree magnitudes at far distances might seem negligible when compared to the corresponding reuse degree peaks within the block, it is also desirable to reduce the maximum distance at which the blocks reuse distance, not just the magnitude of the reuse degree at a specific distance. Taking this into consideration is particularly important when comparing the way coding optimizations affect the data reuse characteristic.

It is also important to make some comments on the particular data reuse characteristic of the kernels. As mentioned before, Figures 29~36 present data reuse characteristics assuming enough parallelism to run all threads concurrently, very close to the ideal behavior of the application. In addition, we also have seen that blocks of one kernel present a data reuse characteristic with very similar profiles, only differentiating themselves with different magnitudes at reuse distances. It seems intuitive that the per block data reuse characteristic should be the way kernels of different applications from one another, since these are able to represent the data reuse patterns very effectively. However, depending on the characteristics of the kernel code, significant variations can occur, as can be seen by *nbf*, *moldyn*, *irreg* and *euler*. In the Figures 33~36 it is feasible to notice the way the reuse distance domains of these kernels vary in block 1 with respect to block 0. There are also variations in the reuse degree magnitudes.

When a kernel has a significant amount of branches with state only known at runtime or presents runtime variability of various kinds, and coupled with this, many instructions that access memory are within the branch paths, then the reuse characteristic can vary significantly from one block to the next. For some applications, the reuse distance domain is drastically reduced for many of blocks in the kernel. The data reuse characteristic of kernel blocks that were obtained with our methodology were not tested in their entirety. There are some kernels for which such variations are almost non-existent, as with the case for *sta*, *gsim*, *bfs* and

vectoradd. On the other hand, kernels such as *nbf*, *moldyn*, *irreg* and *euler* do present important variations. In addition, we also ran preliminary tests with applications from the Rodinia [22] benchmark suite prior to writing this document, and we were able to see that there was significant variability in the reuse characteristic of different blocks. The analyses of the variability in the data reuse characteristic presented by the thread blocks due to runtime dynamics is left for future work.

9.2 Data Reuse Characteristic with varying parallelism capabilities

In this sub-section, we present the results corresponding to Scenarios 3 and 4. Figures 37~44 show the data reuse characteristic of the applications obtained when all thread blocks of the kernel are executed in parallel. There's a huge similarity in the contour of the charts between these figures and their corresponding counterparts in Figure 29~36. Consider *bfs* in Figure 39, and consider the first three blocks of *bfs* shown in Figure 31. When all blocks are modeled as running in parallel, the resulting data reuse characteristic has the same contour but with varying magnitudes of the data reuse degree at the same distances. There's still some variation with respect to the particular blocks, but the contour similarities between both charts is very strong. This same behavior is presented between all of the applications and their associated thread blocks.

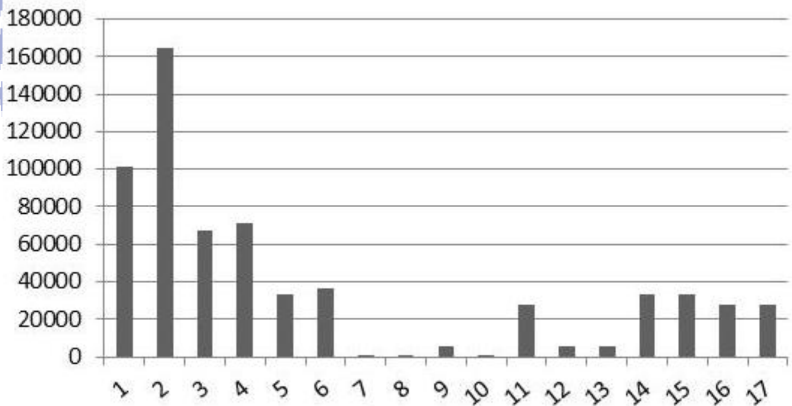


Figure 37: Data reuse characteristic when modeling block execution when all blocks run in parallel for *sta*.

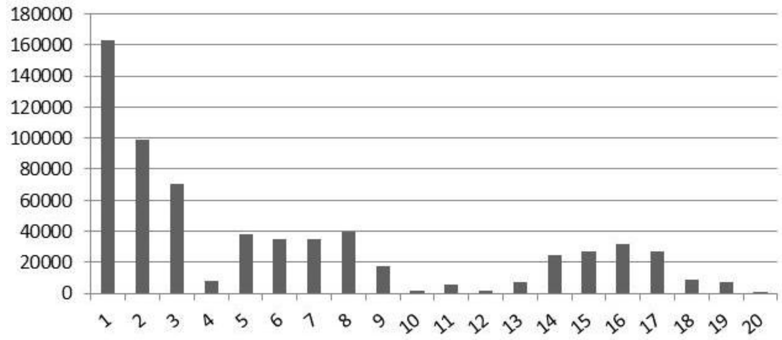


Figure 38: Data reuse characteristic when modeling block execution when all blocks run in parallel for *gsim*.

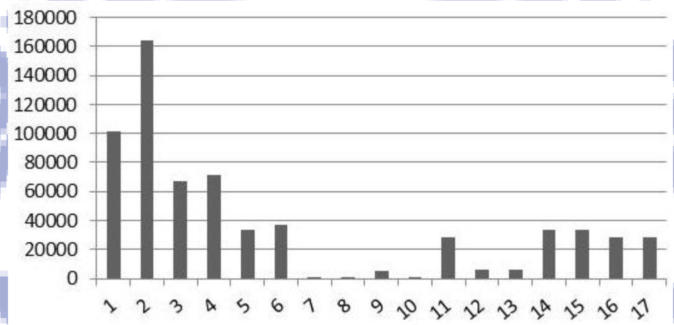


Figure 39: Data reuse characteristic when modeling block execution when all blocks run in parallel for *bfs*.

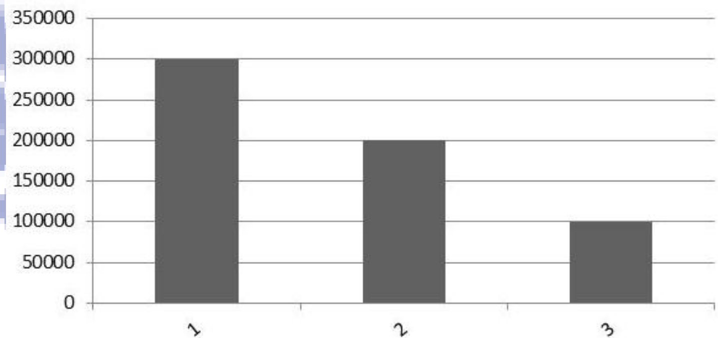


Figure 40: Data reuse characteristic when all blocks of *vectoradd* are modeled as executing in parallel.

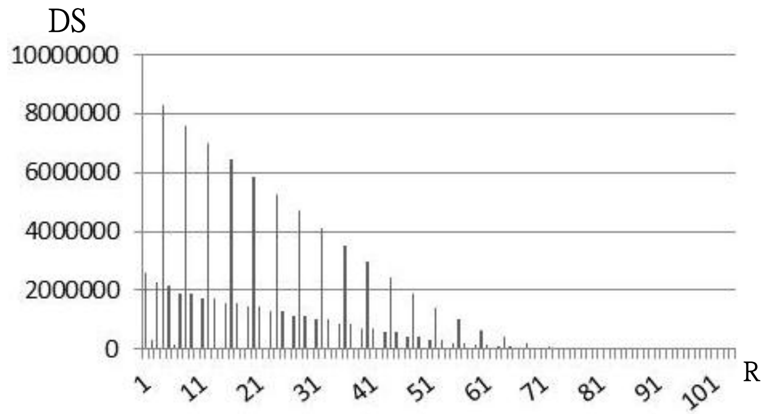


Figure 41: Data reuse characteristic when all blocks of *nbf* are modeled as executing in parallel.

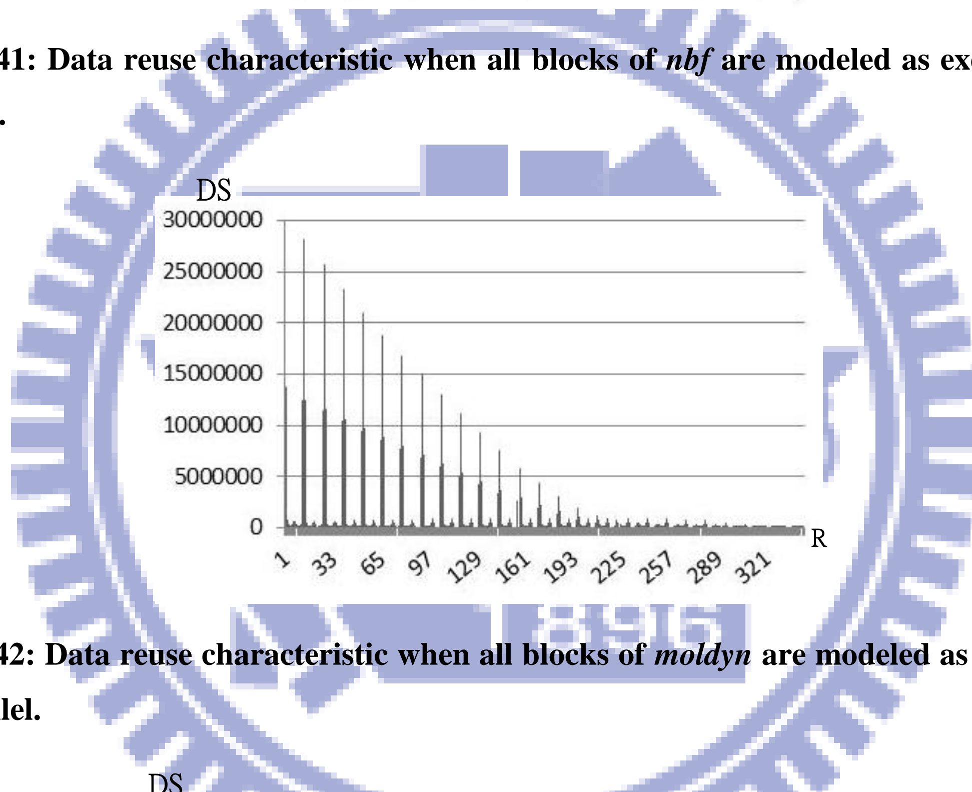


Figure 42: Data reuse characteristic when all blocks of *moldyn* are modeled as executing in parallel.

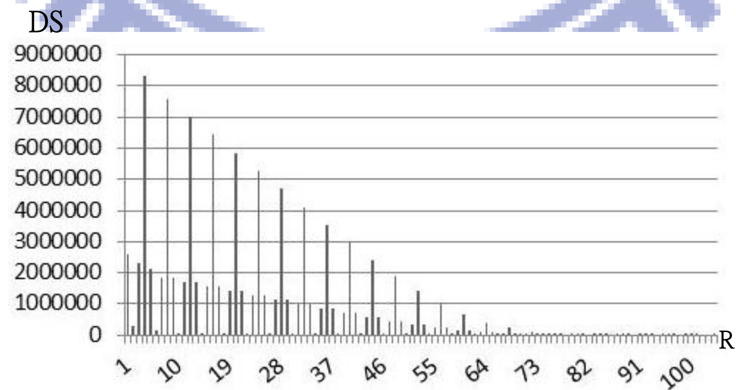


Figure 43: Data reuse characteristic when all blocks of *irreg* are modeled as executing in parallel.

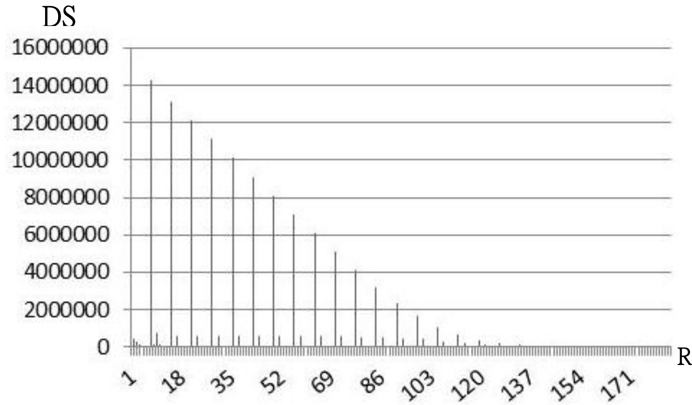


Figure 44: Data reuse characteristic when all blocks of *euler* are modeled as executing in parallel.

This similarity observed is particularly important. The runtime variations that alter the data reuse characteristic of each individual block, briefly explained in Section 9.1, get "absorbed" by the characterization methodology. That is, the resulting characteristic of all blocks running in parallel becomes insensitive to the changes in the reuse characteristic that certain blocks suffer due to runtime dynamics. This is because the analysis process is a reductionist since it accumulates all the data reuse degrees associated to a specific value of reuse distance. The result is that we obtain a reuse characteristic that is ideal for which block scheduling or task allocation policies are trivial. Such reuse characteristic is not defined by the details of the memory subsystem and not by parallelism limitations. Therefore, it becomes a less variable and more reliable representation of the data reuse characteristic of the kernel. Tentatively, we can define this as the signature reuse characteristic for the applications used in this work.

Figures 45~52 present the data reuse characteristic of the applications when a different number of blocks '*K*' are able to run concurrently. For every application, we used *K=2, K=4, K=8, K=16*. The blocks that run simultaneously are chosen by the block scheduling module and the policy it implements. The way the blocks are scheduled in the concurrent slots modifies the resulting reference stream, which will result in variations of the data reuse characteristic. Notice that all of the applications present a similar contour with peaks of different magnitude at different reuse distances. Also, they all present the same behavior as the number of blocks able to run in parallel increases.

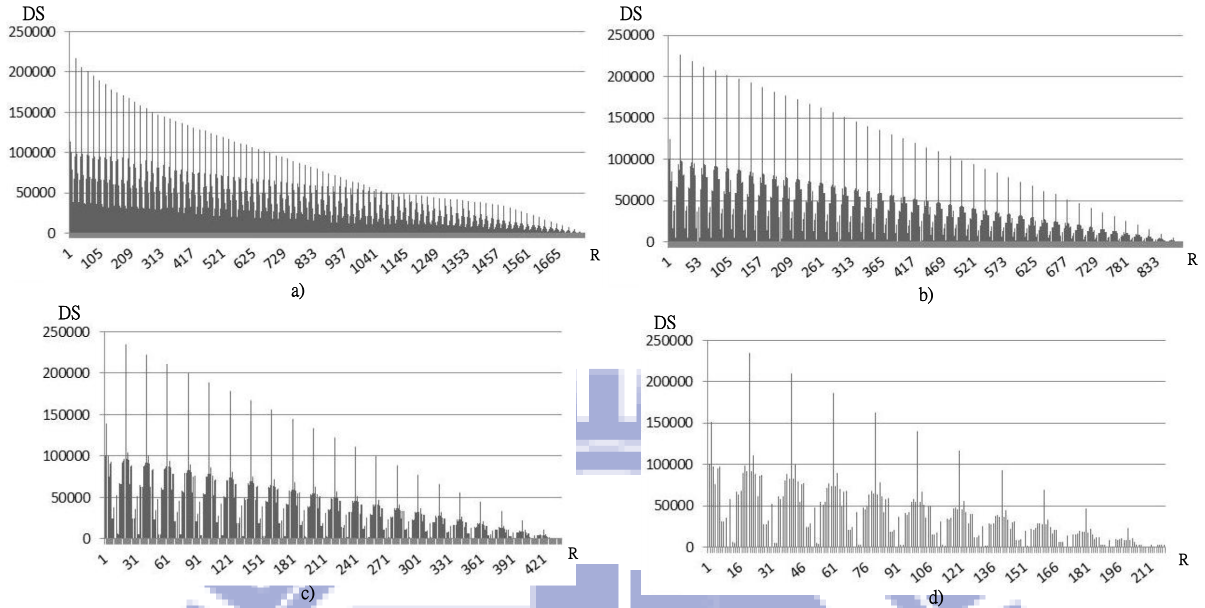


Figure 45: Data reuse characteristic when only '*K*' blocks of *sta* are modeled as executing in parallel. (a) Data reuse characteristic for *K=2*. (b) Data reuse characteristic for *K=4*. (c) Data reuse characteristic for *K=8*. (d) Data reuse characteristic for *K=16*.

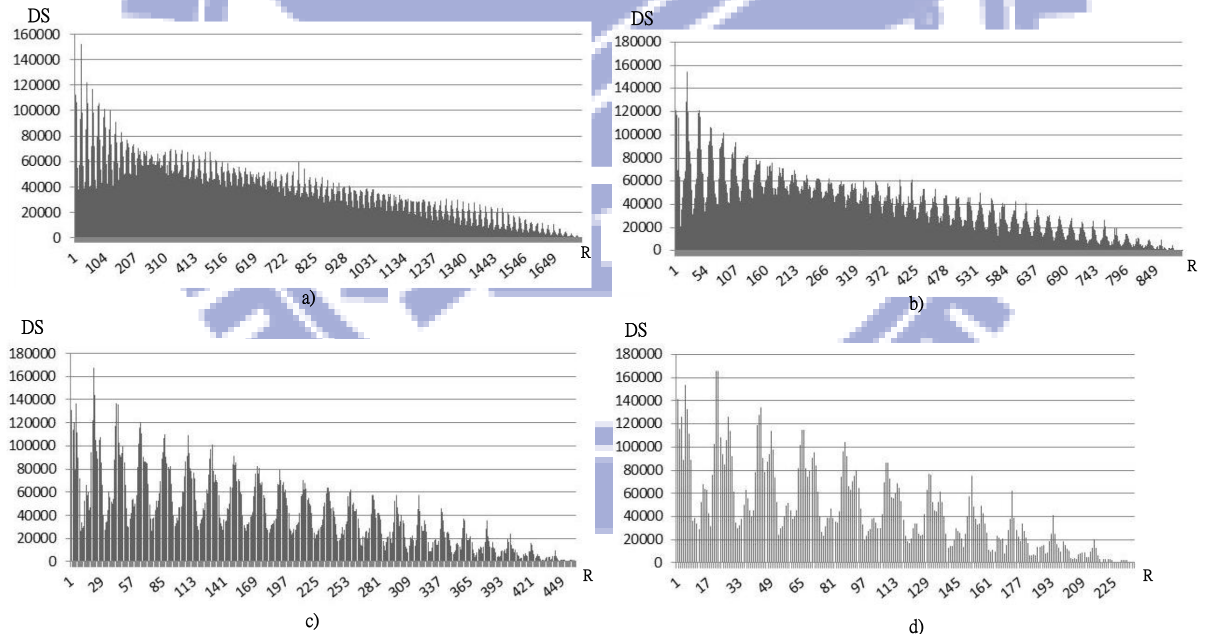


Figure 46: Data reuse characteristic when only '*K*' blocks of *gsim* are modeled as executing in parallel. (a) Data reuse characteristic for *K=2*. (b) Data reuse characteristic for *K=4*. (c) Data reuse characteristic for *K=8*. (d) Data reuse characteristic for *K=16*.

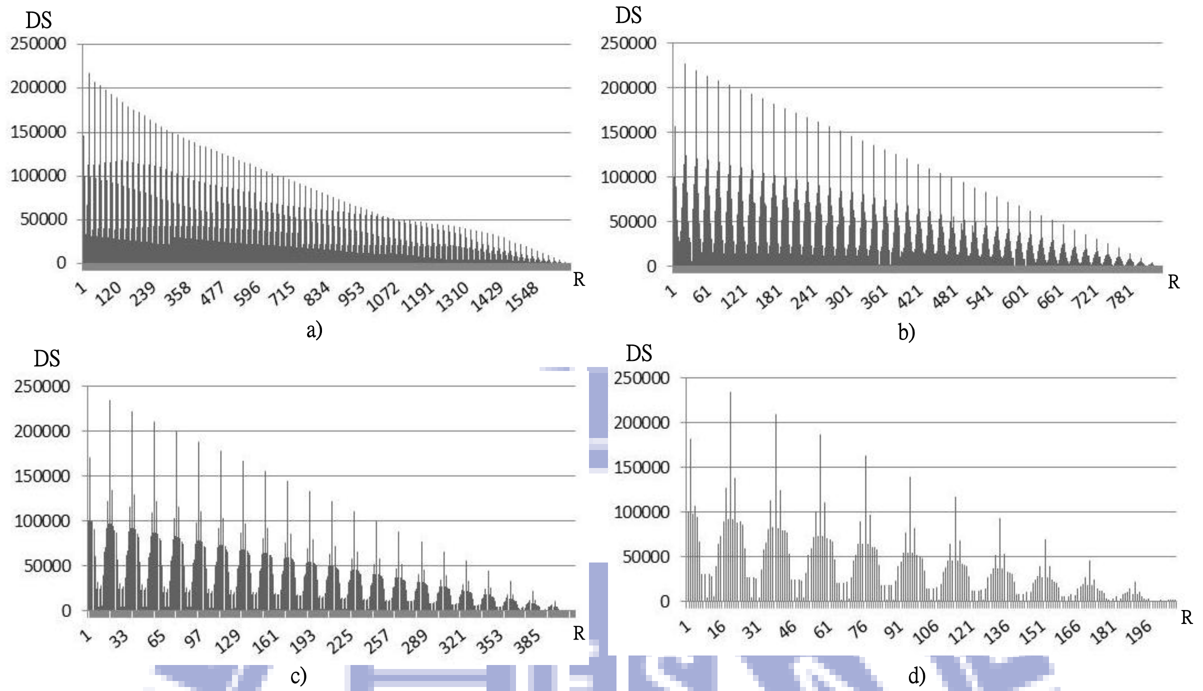


Figure 47: Data reuse characteristic when only '*K*' blocks of *bfs* are modeled as executing in parallel. (a) Data reuse characteristic for *K=2*. (b) Data reuse characteristic for *K=4*. (c) Data reuse characteristic for *K=8*. (d) Data reuse characteristic for *K=16*.

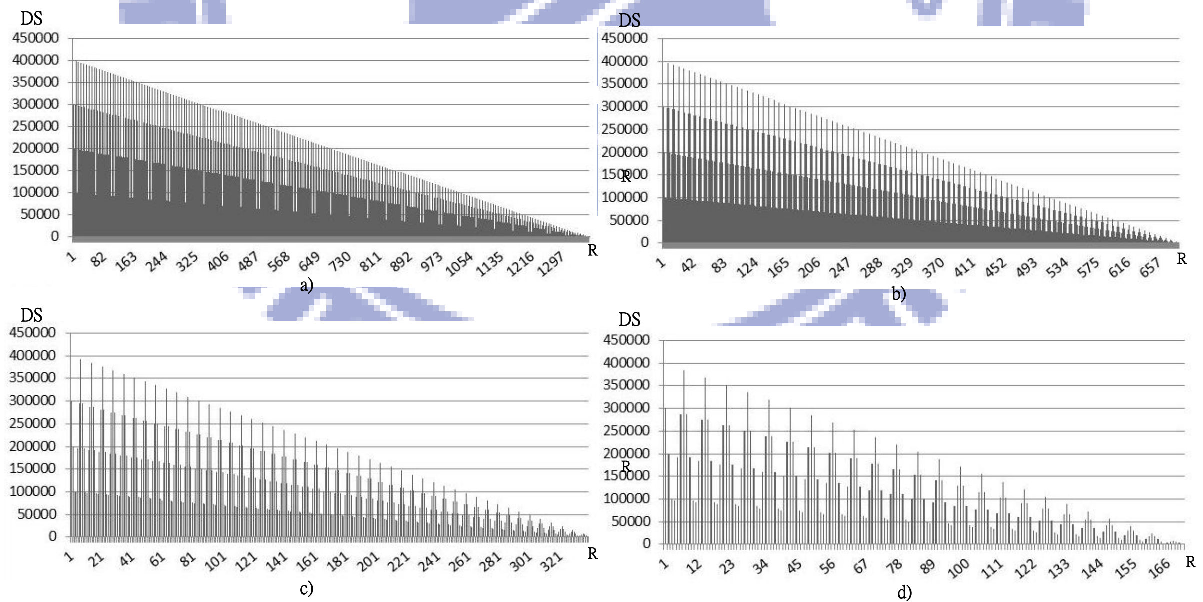


Figure 48: Data reuse characteristic when only '*K*' blocks of *vectoradd* are modeled as executing in parallel. (a) Data reuse characteristic for *K=2*. (b) Data reuse characteristic for *K=4*. (c) Data reuse characteristic for *K=8*. (d) Data reuse characteristic for *K=16*.

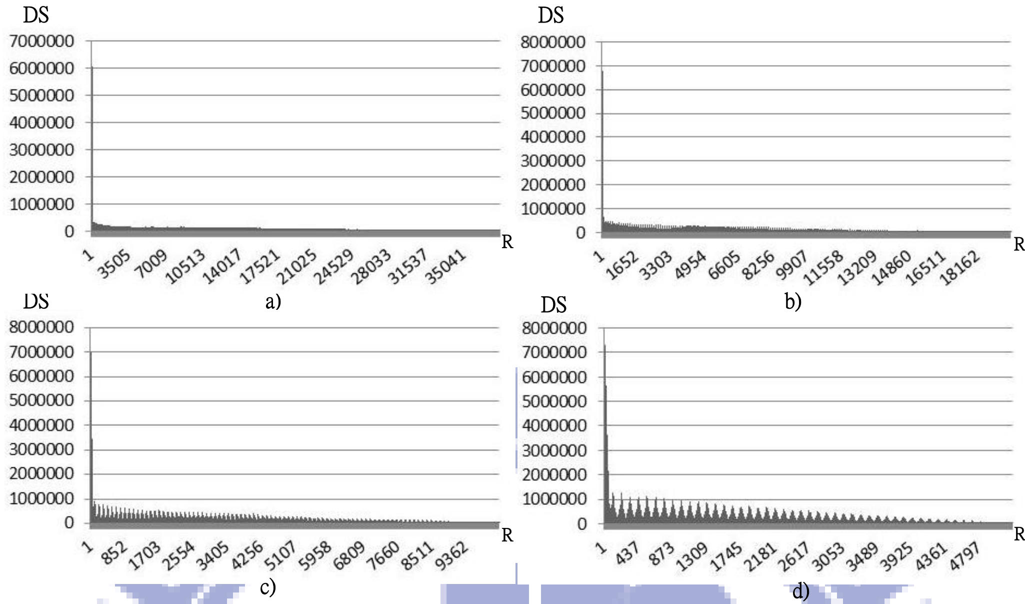


Figure 49: Data reuse characteristic when only 'K' blocks of *nbf* are modeled as executing in parallel. (a) Data reuse characteristic for K=2. (b) Data reuse characteristic for K=4. (c) Data reuse characteristic for K=8. (d) Data reuse characteristic for *K=16*.

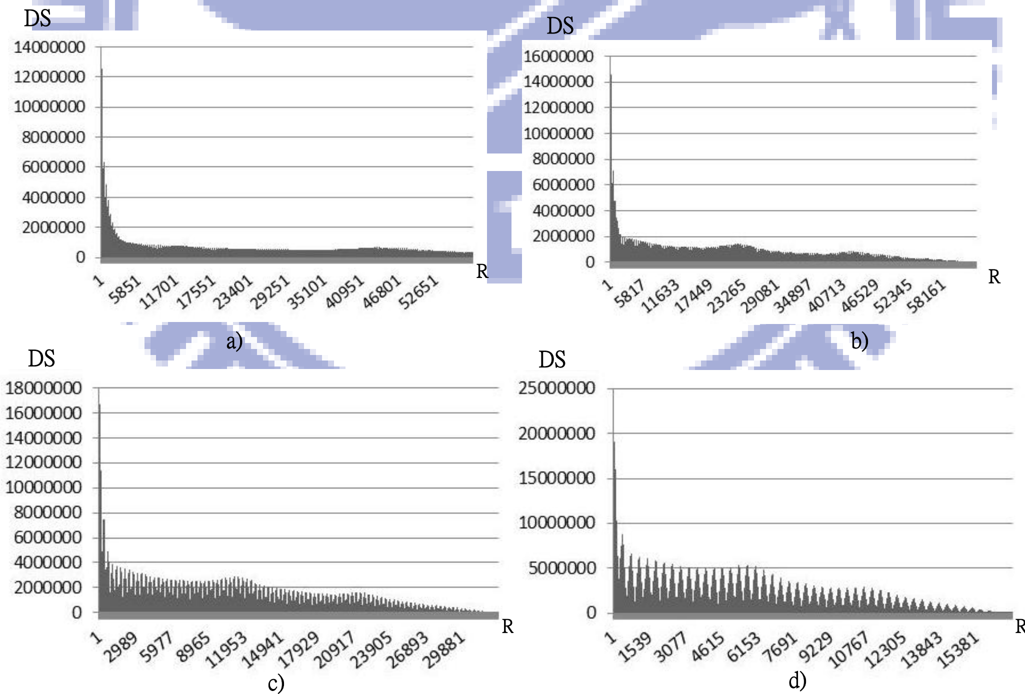


Figure 50: Data reuse characteristic when only 'K' blocks of *moldyn* are modeled as executing in parallel. (a) Data reuse characteristic for K=2. The reuse domain for this case is actually RD={1,124030}. The tool used to make the graphs could not display it

properly. (b) Data reuse characteristic for K=4. (c) Data reuse characteristic for K=8. (d) Data reuse characteristic for *K=16*.

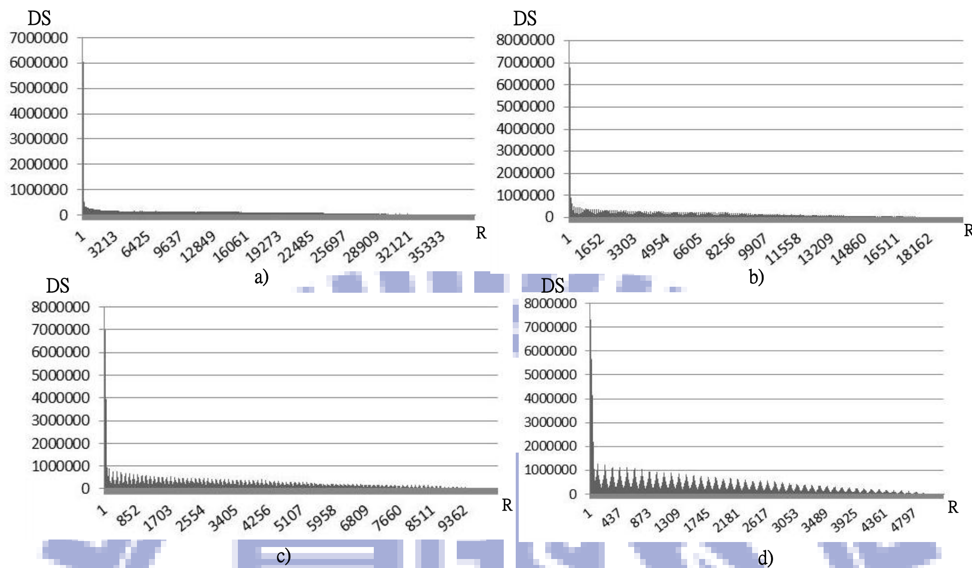


Figure 51: Data reuse characteristic when only 'K' blocks of *irreg* are modeled as executing in parallel. (a) Data reuse characteristic for K=2. (b) Data reuse characteristic for K=4. (c) Data reuse characteristic for K=8. (d) Data reuse characteristic

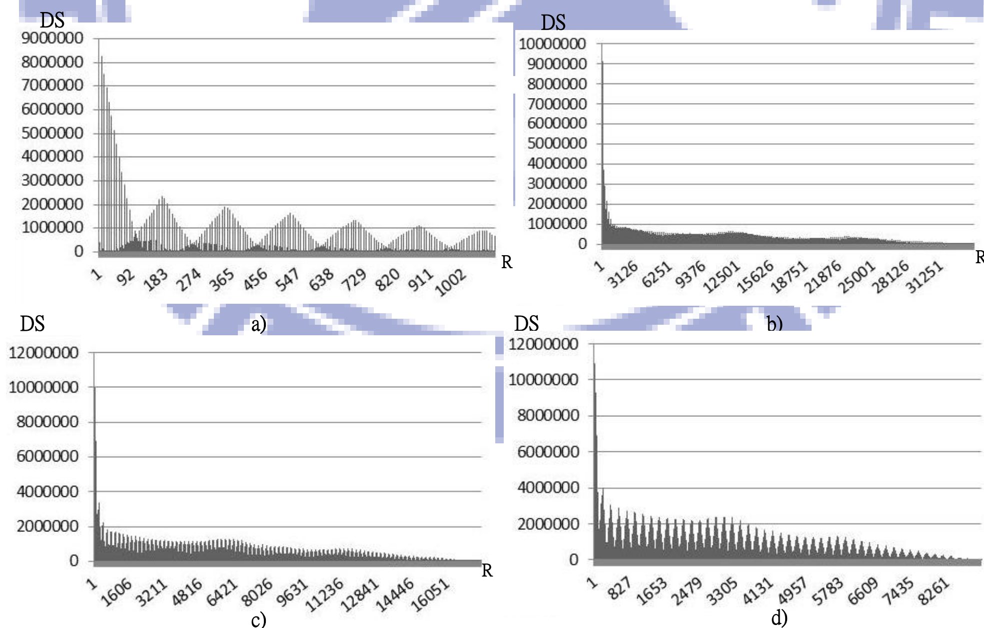


Figure 52: Data reuse characteristic when only 'K' blocks of *euler* are modeled as executing in parallel. (a) Data reuse characteristic for K=2. The reuse domain for this case is actually RD={1,66623}. The tool used to make the graphs could not display it

properly. (b) Data reuse characteristic for K=4. (c) Data reuse characteristic for K=8. (d) Data reuse characteristic

For every value of '*K*', all the kernels find their reuse distance domain (spectrum) increased when compared to the case where all blocks are running in parallel. This happens because of the amount of data reuse across blocks that these applications present, analogous to what occurs when all blocks are serialized. The increase in the total reuse distance domain is hard to determine a priori because of the variations that blocks may incur during runtime, no matter how slightly they are. Also, and for this same reason, the number of reuse degree peaks increase across the RD domain.

As '*K*' increases from 2 to 4, the RD domain is reduced in a near-linear away. For every application, the reuse distance domain when *K=4* is nearly half of the reuse distance domain when *K=2*. The same occurs between *K=8* and *K=4*, and between *K=16 and K=8*. This demonstrates, in a quantified way, the impact that the availability of resources has over the reuse characteristic of a kernel. If the amount of blocks that are able to run in parallel is very low, and assuming that a kernel has a fairly large amount of blocks when compared to the resources in a real architecture, it is very likely that data will be reused at very far away distances i.e. after a large number of MIs. For example, let's take a look at *sta*. When running all blocks in parallel, we observe *DS= 1256* at the largest reuse distance in the domain *RD = 37*. On the other hand, when only 2 blocks are able to run in parallel, we have that at the largest reuse distance *RD = 4103* with *DS=2*, very low reuse degree and very far. This fact can have a significant impact on the application's performance, as explained in Section 9.1. Notice that the RD domain has increased by more than 10 times.

Even though the reuse characteristics shown in Figures 45~52 present a similar contour as '*K*' varies, the actual reuse degree vs. reuse distance relationship is still very similar to the one of their corresponding blocks i.e present strong similarity, even after applying a round robin scheduling policy. In Figures 53 and 54 we present the data reuse characteristic of *sta* for *K=2* and *K=16*, respectively, with zoom in three different ranges. We see that in the range from *RD=1* to *RD=20* in Figure 53(a) for *K=2*, there's a strong similarity with the characteristic obtained when all blocks are running in parallel. Even the magnitudes are very similar. In the next range from *RD=21* to *RD=40* in Figure 53(b), the variation with respect to the previous range is trivial, both are almost identical. And the last range, from *RD=41* to *RD=60* in Figure

53(c), we see that it is also strikingly similar to the previous range. The same behavior is observed for *K=16*, as presented in Figure 54. Therefore, different ranges of the overall data reuse characteristic are heavily related to one another. This happens because, even though the runtime dynamics affect the reuse characteristic of the blocks, the data reuse pattern is still very similar across the majority of the blocks of the kernels used in this work.

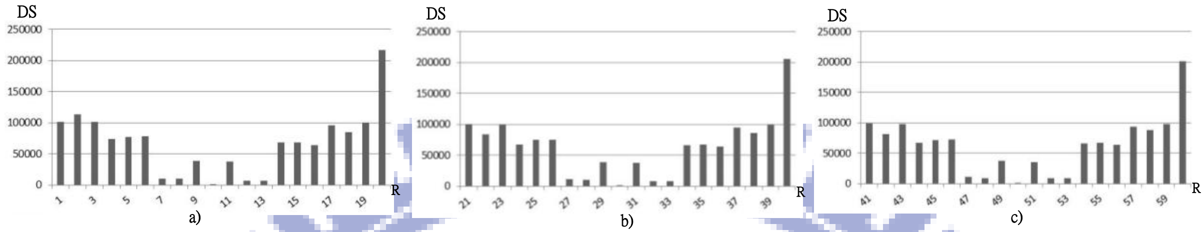


Figure 53: Data reuse characteristic resulting when only *K=2* blocks of *sta* are modeled as executing in parallel. (a) Data reuse characteristic presented for reuse distance range *RD={1, 20}*. (b) Data reuse characteristic presented for reuse distance range *RD={21, 40}*. (c) Data reuse characteristic presented for reuse distance range *RD={41, 60}*.

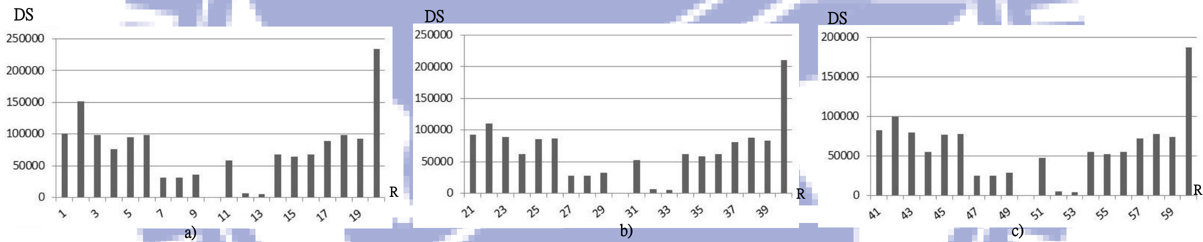


Figure 54: Data reuse characteristic resulting when only *K=16* blocks of *sta* are modeled as executing in parallel. (a) Data reuse characteristic presented for reuse distance range *RD={1, 20}*. (b) Data reuse characteristic presented for reuse distance range *RD={21, 40}*. (c) Data reuse characteristic presented for reuse distance range *RD={41, 60}*.

9.3 Data Reuse Characteristic with limitations of SIMT Architectures

The results presented in this subsection correspond to Scenarios 5 and 6 of the data reuse characterization. These Scenarios capture the data characteristic when modeling the limitations of SIMT architectures when issuing memory instructions. As explained in Sections 6.5 and 6.6, we include a finite number of load/store units associated with each core cluster in the system, which we modeled in the Locality Analyzer. We experimented with a constant number of core clusters and varying number of load/store units. Figures 55~61 present the

reuse characteristic obtained when modeling all core clusters in the systems, setting the number of core clusters to 16 and with 16, 32 and 64 load/store units per core cluster.

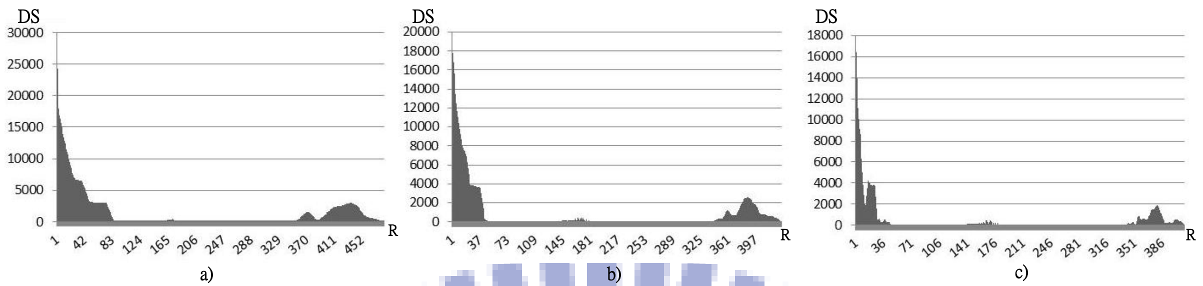


Figure 55: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *sta*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster.

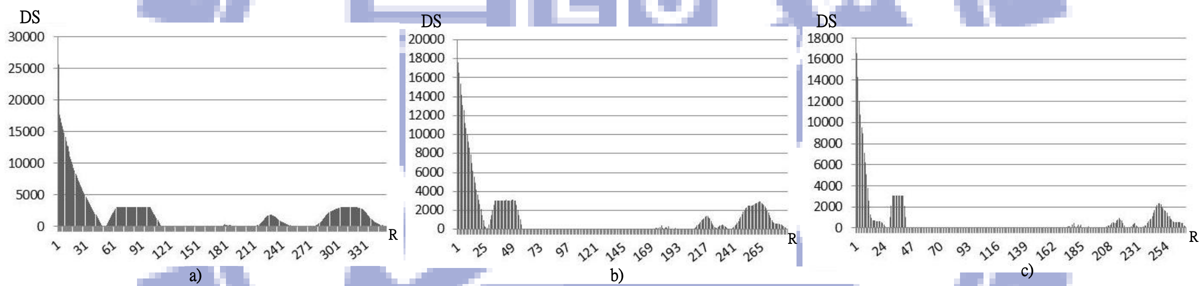


Figure 56: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *gsim*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster.

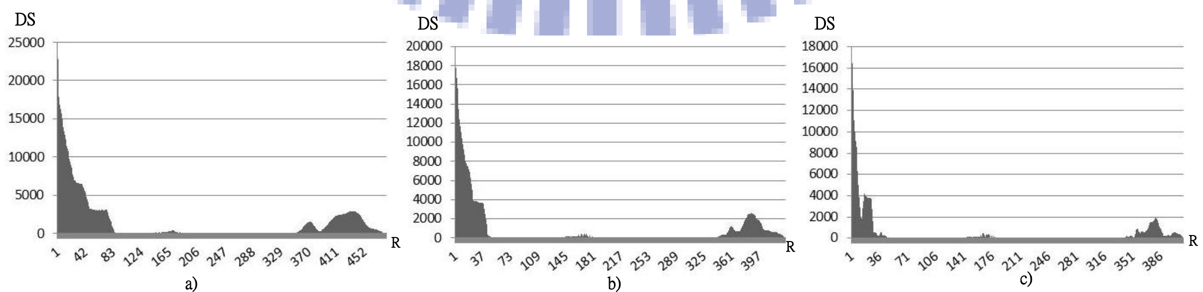


Figure 57: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *bfs*. (a) Data

reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster.

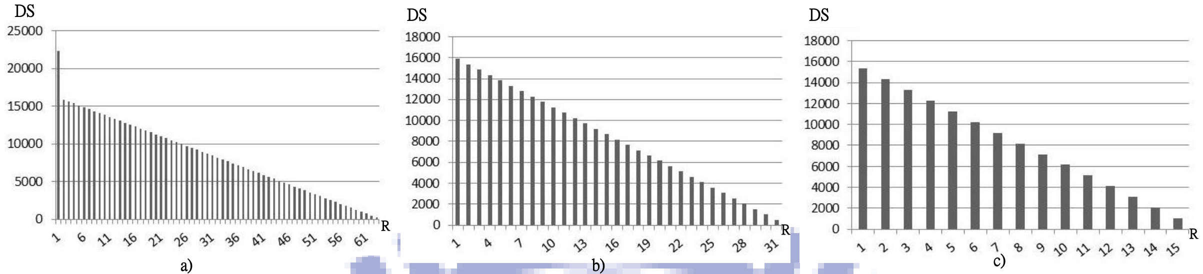


Figure 58: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *vectoradd*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster.

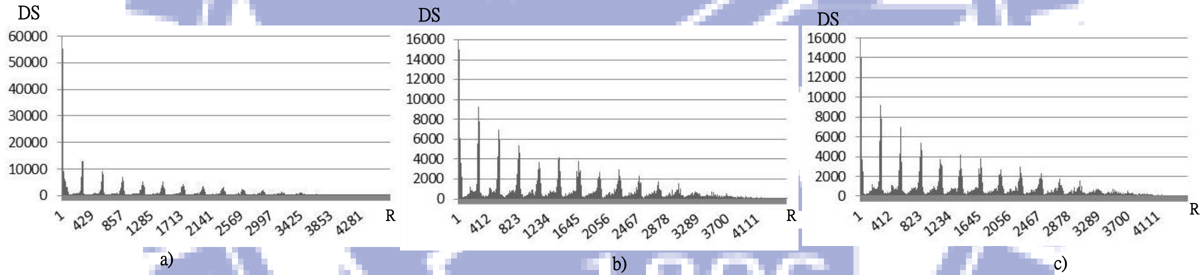


Figure 59: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *nbf*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster.

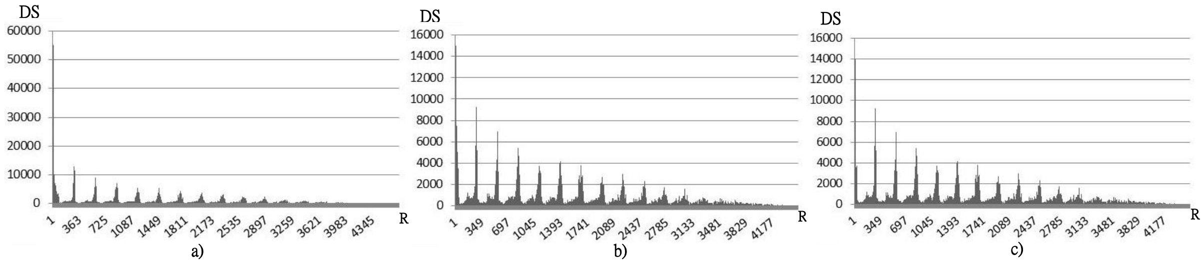


Figure 60: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *irreg*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster.

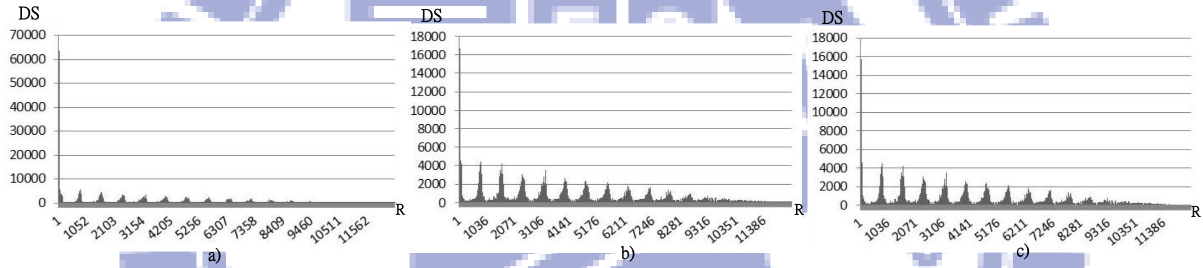


Figure 61: Data reuse characteristic from the aggregate reference stream of all core clusters with varying number of load/store units in each core cluster for *euler*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster.

In Figure 55, the data reuse characteristic for *sta* is shown when modeling certain SIMT architectural limitations. Notice that for the reuse degree peaks that appeared in the results of previous sections do not appear in this figure. It appears that the characteristic is flattened as the reuse distance increases. The reuse degree magnitudes at short distances are very high, but decrease at a higher rate as the reuse distance increases. This same behavior, as Figures 55~58 show, is common to *sta*, *gsim*, *bfs* and *vectoradd*.

These four applications present a very particular case. When the number of load/store units becomes limited, the amount of data reuse that one single MI can exploit with relationship to any other MI is significantly reduced. The amount of threads that execute one specific

memory instruction is usually much larger than the number of load/store units in a single core cluster. Consequently, the total number of simultaneous accesses that a single core cluster can handle is dependent on the amount of load/store units it has. Therefore, to service the requests of multiple threads, more execution cycles are necessary, which in turns means more MIs increasing the position index of every MI in the reference stream. This has the effect of distributing the total reuse degree across different memory accesses, which basically flattens the reuse characteristic, distributing the magnitude of degree peaks previously seen among a wider number of distances.

This observation is particularly important. Keep in mind that we have only modeled one single resource limitation of SIMT architecture for our experiments. Figure 62 shows a zoomed version of *sta* for 16 load/store units. Notice the way the characteristic is smoother than previous cases, and notice the 'kneels' of the chart it now presents at very specific distances. This case does not present any similarity whatsoever with the data reuse characteristic observed in individual blocks of *sta* nor when different amount of blocks run in parallel, as show in Section 9.2. Intuitively, it is more appropriate to conclude that the reuse characteristic obtained from *sta* when modeling SIMT limitations is actually the data reuse that the model of our SIMT architecture can in fact exploit from *sta*. Thus, it is not solely dependent on the application, but in the interaction between the reuse pattern of the *sta* kernel and the SIMT load/store units.

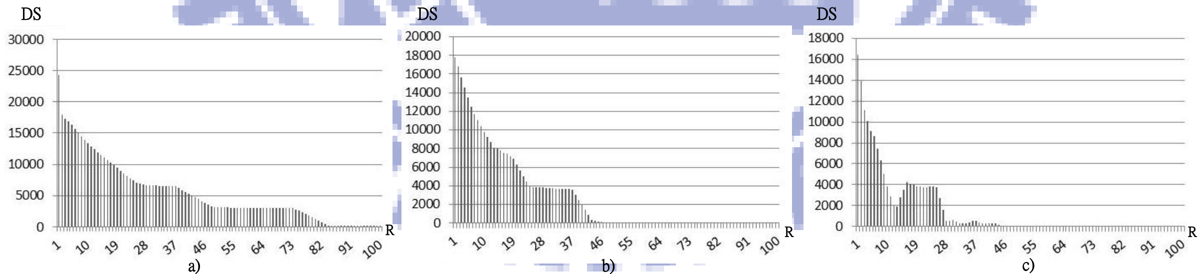


Figure 62: Data reuse characteristic in reuse distance range *RD={0, 100}* of the aggregate reference stream of all core clusters with varying number of load/store units for *sta*. (a) Data reuse characteristic for 16 load/store units per core cluster. (b) Data reuse characteristic for 32 load/store units per core cluster. (c) Data reuse characteristic for 64 load/store units per core cluster.

For the rest of the kernels tested, as shown in Figures 59~61 for *nbf*, *irreg* and *euler*, significant peaks are still present, but with at least 2 orders of magnitude smaller when compared to their respective counterpart for the case when all blocks are modeled as executing in parallel. The reason for why this reuse degree peaks appear is explained by the reuse characteristic itself of each block. These kernels reuse data at very long distances apart, making partially ineffective the smoothing effect due the presence of finite load/store units. It is partially ineffective because the smoothing of the data reuse characteristic curve is not total, as the case with *sta*, but very significant due to the reduction in magnitude. As explained before, when a kernel uses data at such far away distances, there could a huge impact on performance.

Another important observation can be extracted from Figures 55~61. It is not possible to obtain the characteristic of the reuse patterns of the kernels when modeling its runtime behavior under the limitations imposed by real SIMT processors considered in our analyses. When the analyses is performed on real architectures, the characteristic will deviate even further from what is seen under more controlled scenarios. When capturing the data reuse characteristic of the kernels becomes relevant, then it is necessary a methodology like the one proposed with the associated implementation as described in Sections 6 and 7. To accomplish this, it is necessary to abstract the analysis from the details of the architectures, and focus only on the programming and runtime models of the particular processor.

Even though we focused more on the case of *sta*, all of the applications used in this work present a similar behavior, as the figures show. The data reuse characteristics of the applications lose their reuse degree peaks, get flattened or partially flattened i.e. the reuse degree magnitude of the peaks in previous instances is distributed among a larger number of distances. There are cases, as *vectoradd* shown in Figure 58, the decrease rate of the reuse degree might be smaller, but the decrease rate maintains uniformity and relatively smooth. Even with the fact that the general behavior is similar in the kernels presented, there are still differences that need to be explored. A thorough exploration of these differences is left for future work.

Figure 63~70 present the reuse characteristic in two of the individual core clusters. In both Variations 5 (Figures 55~61) and Scenarios 6 (Figures 63~70) the resulting reference stream is different than in the case for Scenario 3 (all blocks running in parallel). This is due to the

effects of the scheduling policy, as in Scenario 4, and this also coupled with the presence of a finite number of core clusters in each core, which makes it necessary to serialize a significant portion of the thread blocks. This limits drastically the amount of MIs that can be executed at the same time. The behavior is very similar to the ones seen in Figures 55~61, corresponding to Scenario 5. There's a striking similarity between these two cases, but the magnitudes in Scenario 6 are comparatively much smaller.

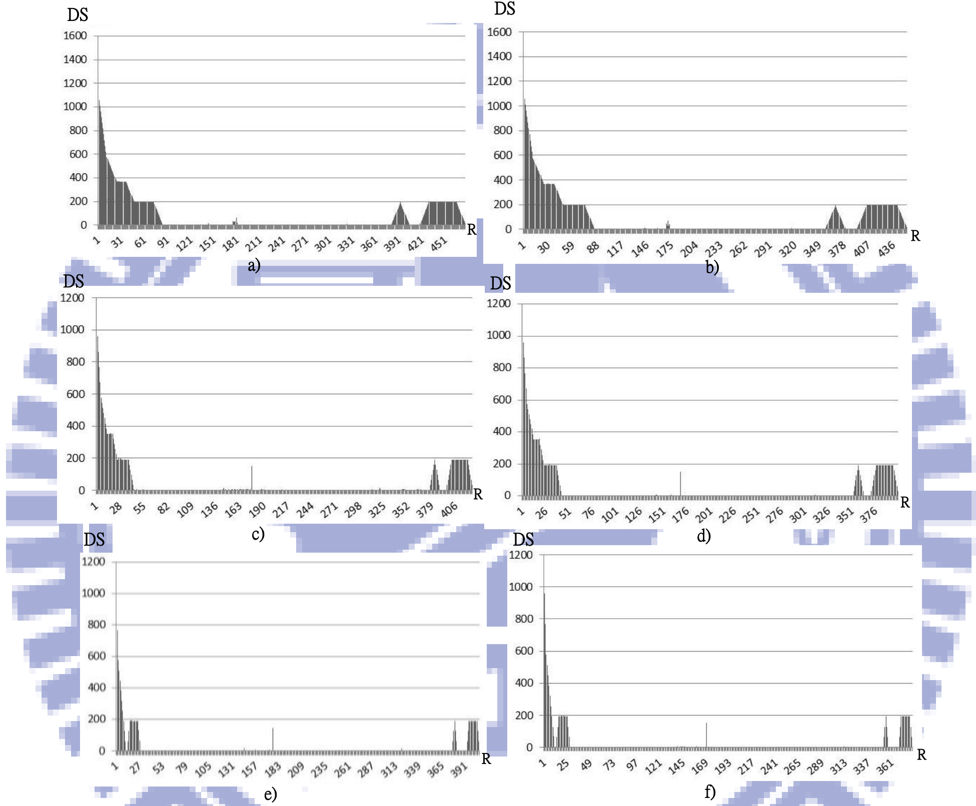


Figure 63: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *sta*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units.

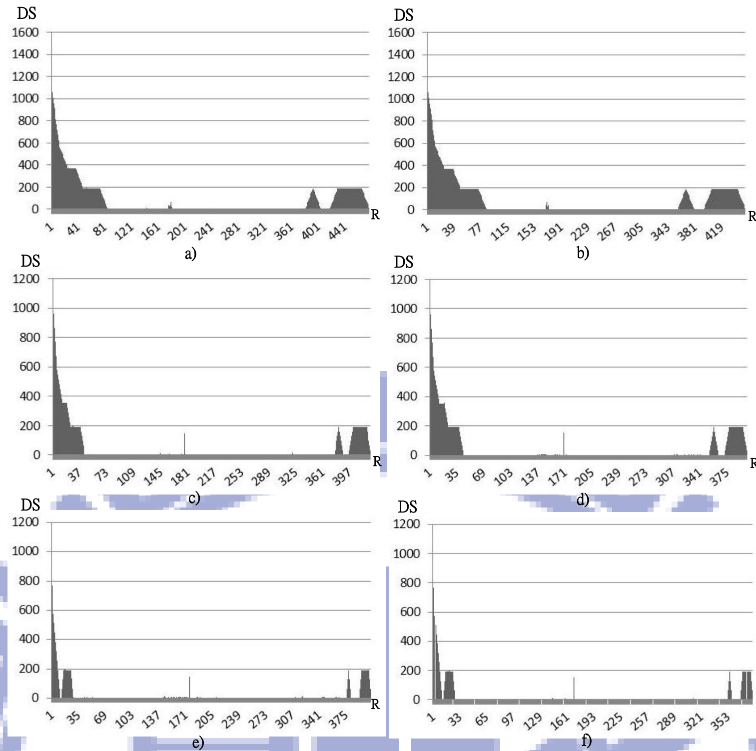


Figure 64: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *gsim*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units.

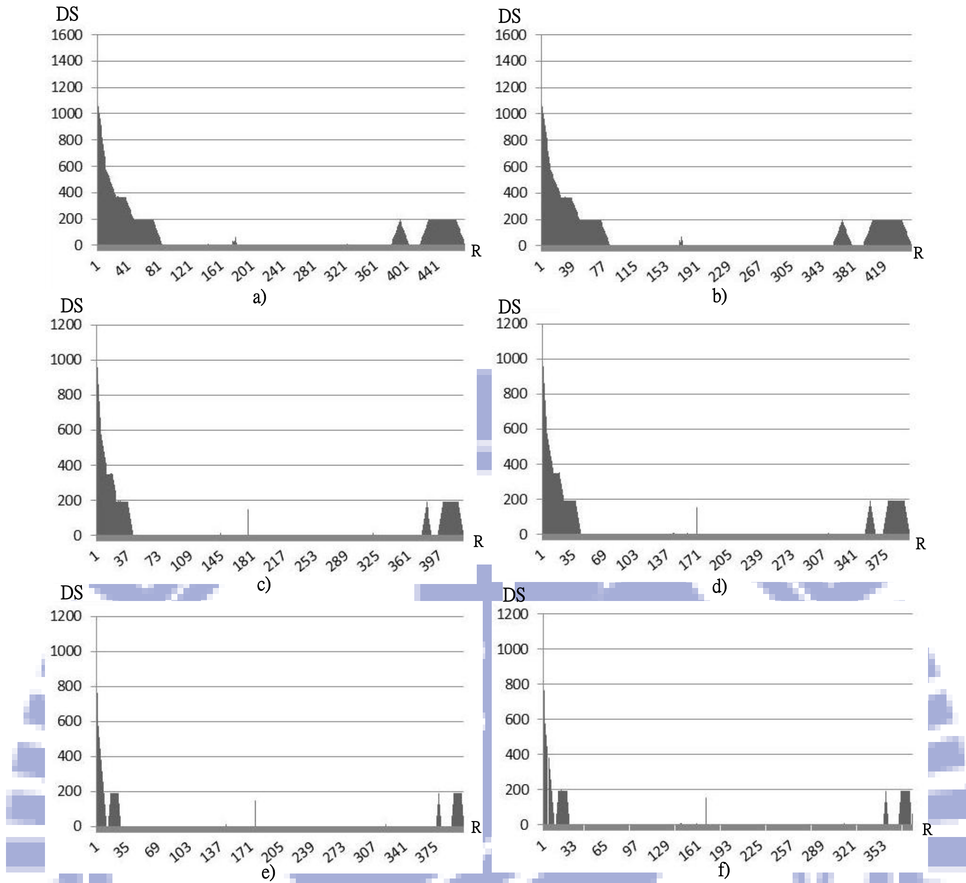


Figure 65: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *bfs*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units.

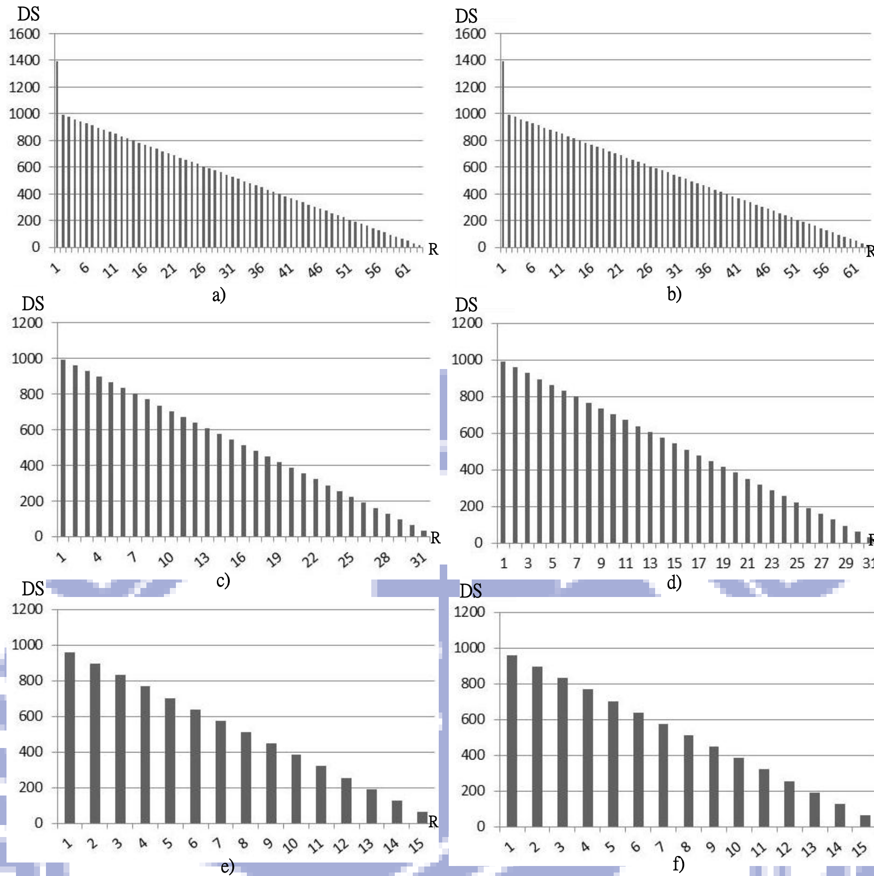


Figure 66: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *vectoradd*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units.

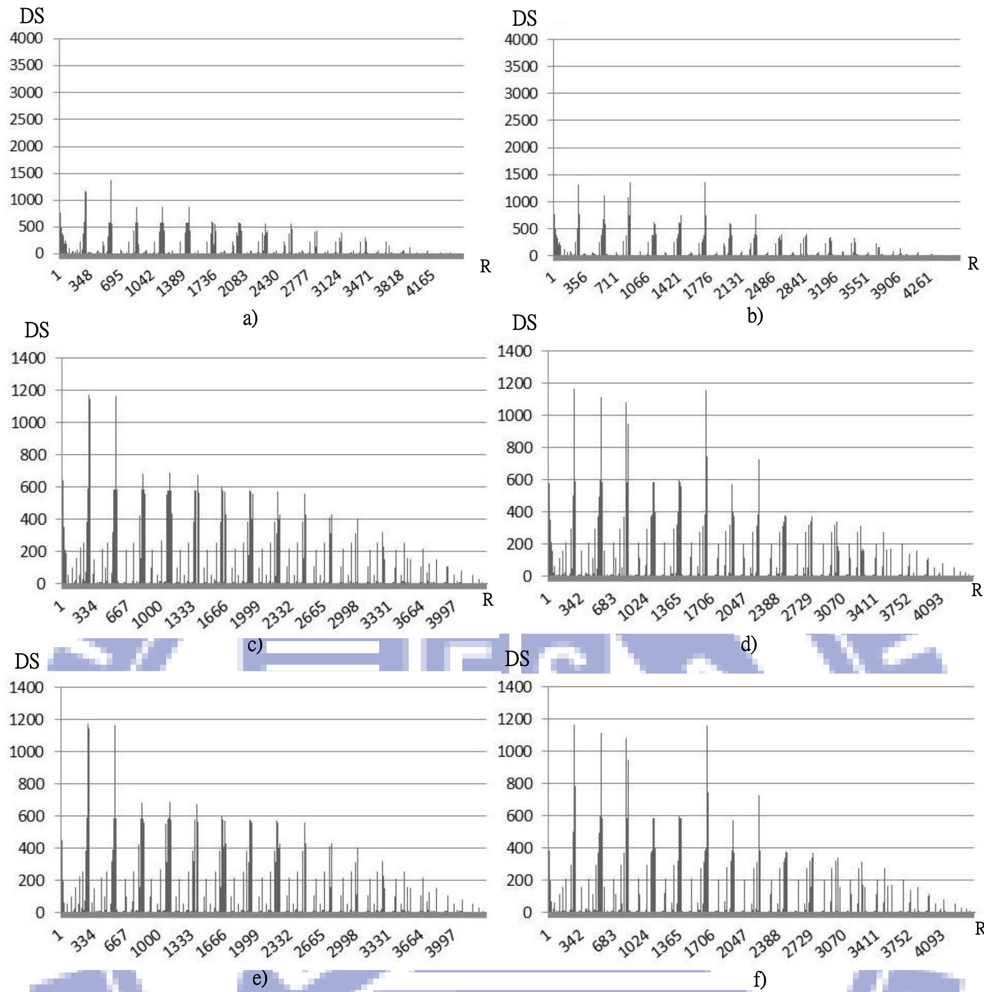


Figure 67: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *nbf*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units.

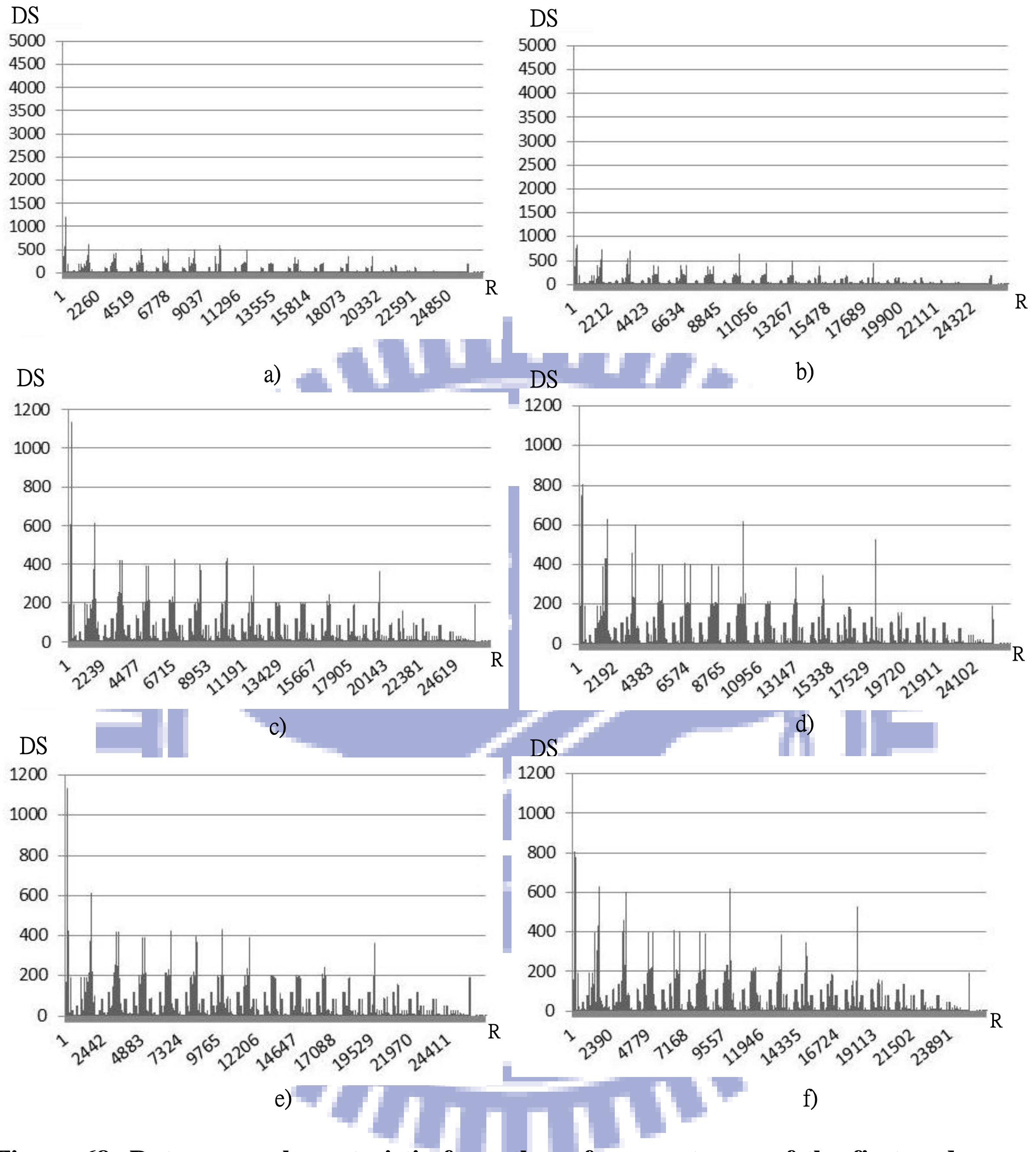


Figure 68: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *moldyn*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units.

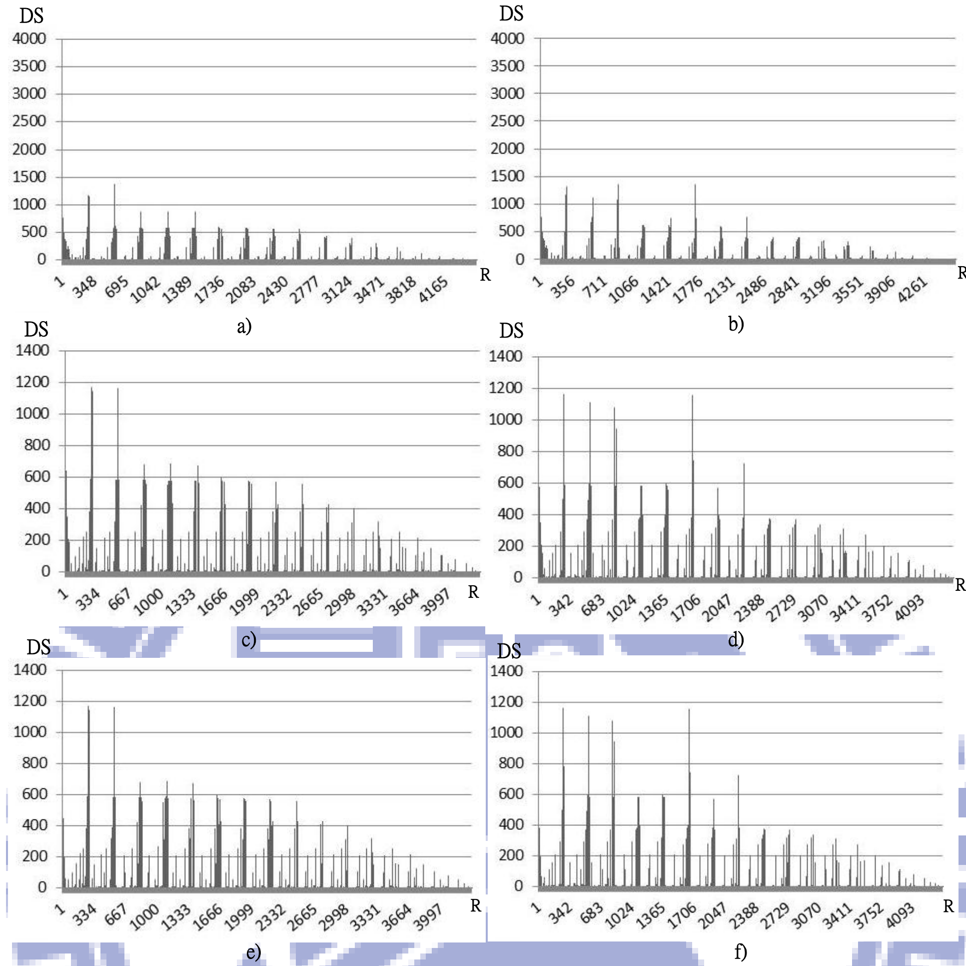


Figure 69: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *irreg*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units.

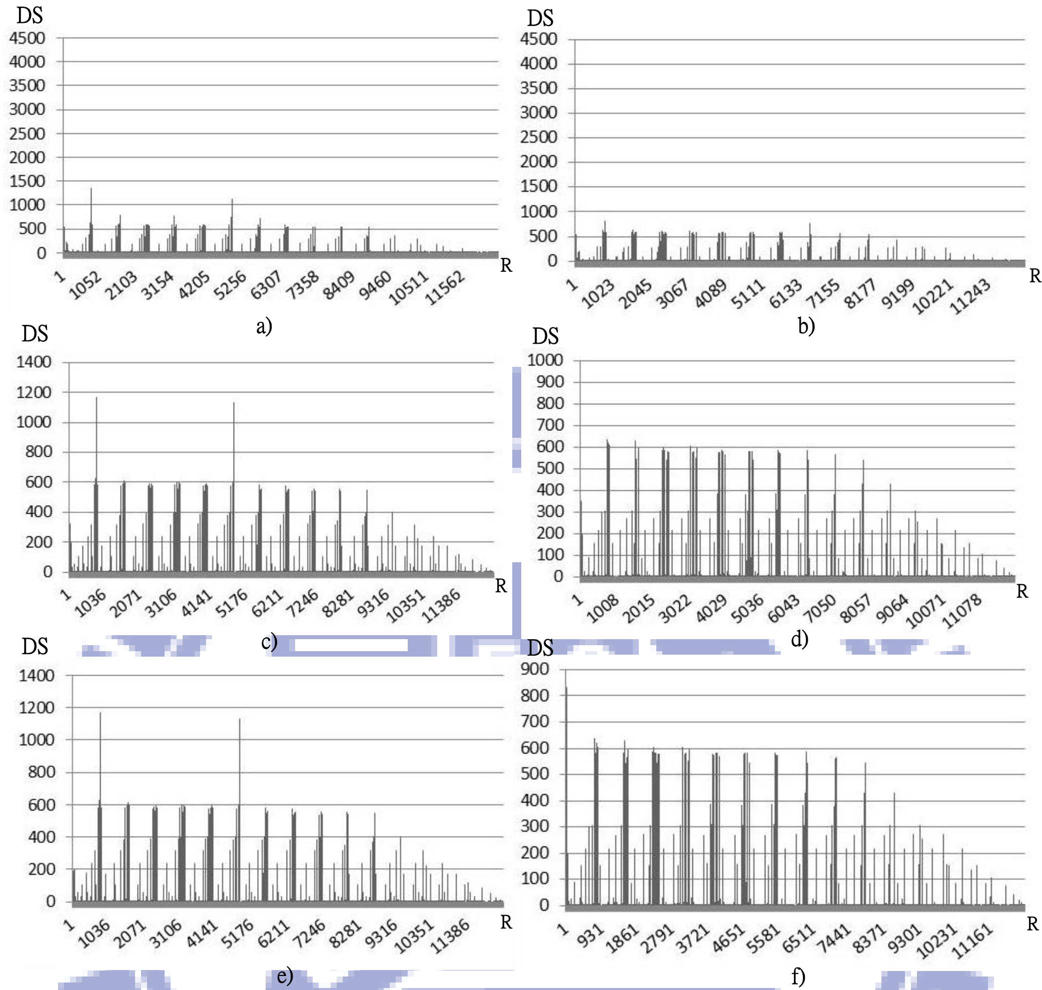


Figure 70: Data reuse characteristic from the reference stream of the first and second core clusters with varying number of load/store units for *euler*. (a) First core cluster with 16 load/store units. (b) Second core cluster with 16 load/store units. (c) First core cluster with 32 load/store units. (d) Second core cluster with 32 load/store units. (e) First core cluster with 64 load/store units. (f) Second core cluster with 64 load/store units.

9.4 Data Reuse Characteristic when applying code optimizations

In this section, the data reuse characteristics of selected applications are obtained after the coding optimizations explained in Section 8. Initially, we will focus on the changes that the data reuse characteristic presents under such optimization when performing Scenario 2 (per thread block) of the data reuse characterization. Figures 71~77 show the charts for *sta*, *gsim*,

bfs, *nbf*, *moldyn*, *irreg* and *euler* respectively. Only the changes of the first block in the kernels are presented. Each of these kernels has over a 100 blocks and to analyze them all, with all of the variations and explain them, will require an extensive analysis that is beyond the current scope.

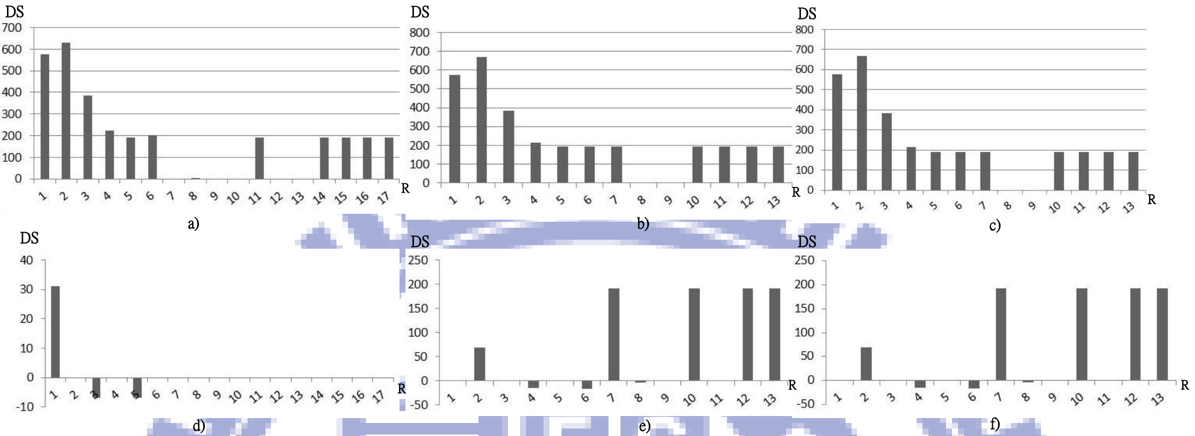


Figure 71: Data reuse characteristic for block 0 of *sta* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 17. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 802. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 802.

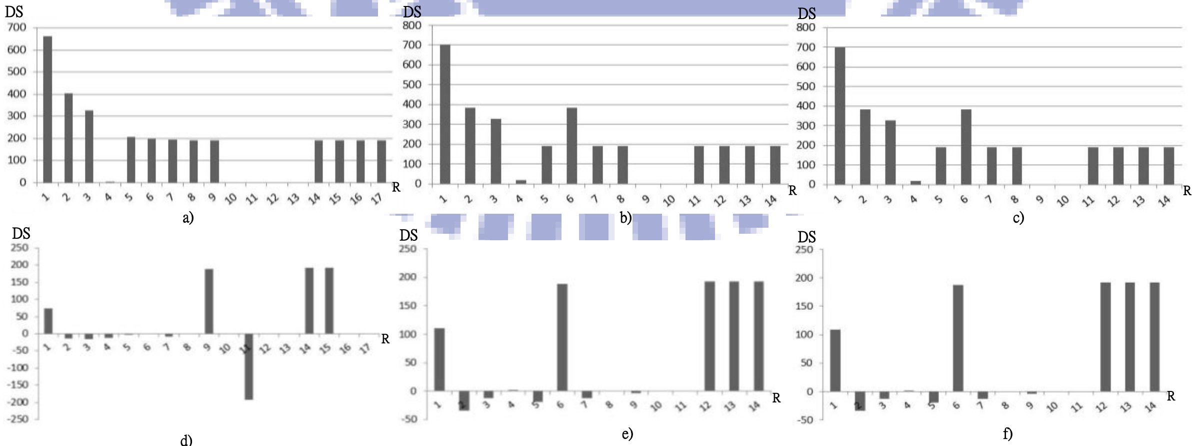


Figure 72: Data reuse characteristic for block 0 of *gsim* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison

prior to optimizations and after thread clustering. Difference in the reuse degree is 400. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 795. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 794.

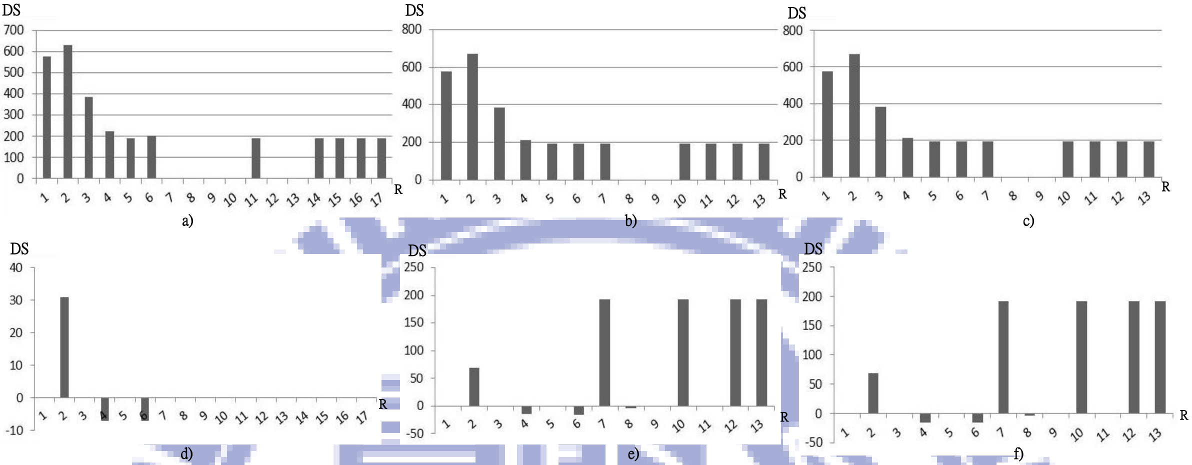


Figure 73: Data reuse characteristic for block 0 of *bfs* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 17. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 802 (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 802.

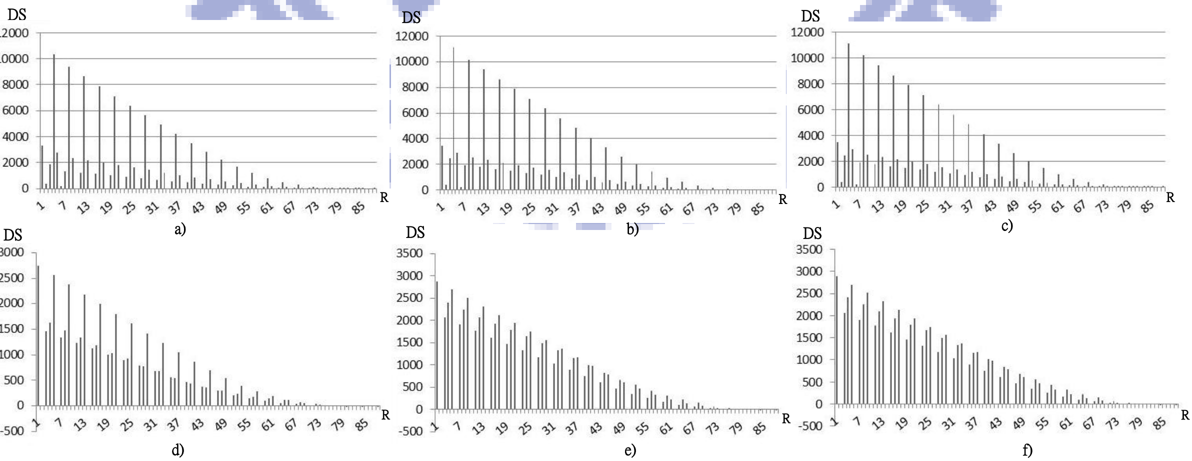


Figure 74: Data reuse characteristic for block 0 of *nbf* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c)

After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 44458. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 60346 (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 60576.

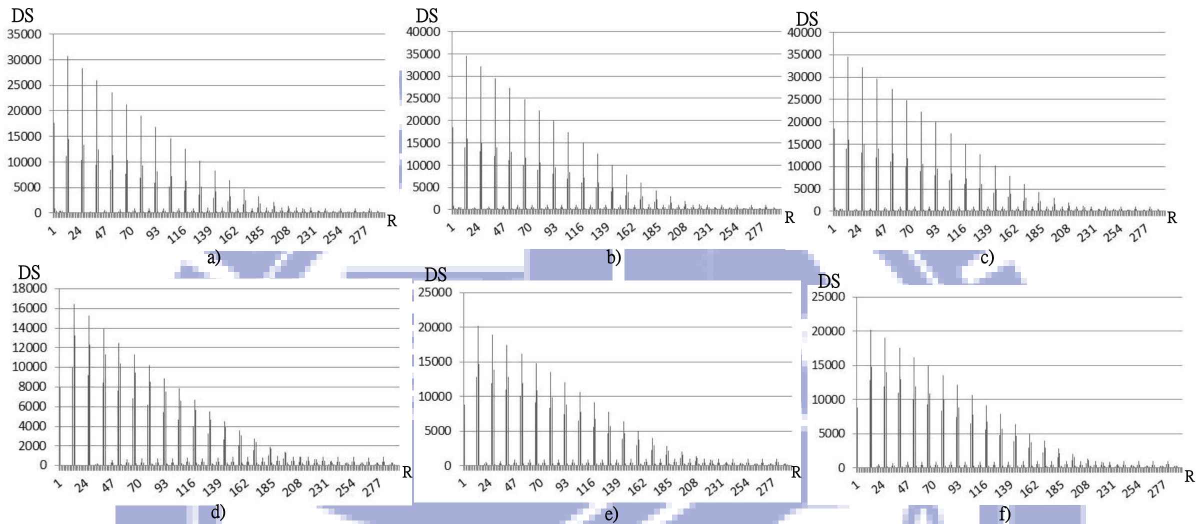


Figure 75: Data reuse characteristic for block 0 of *moldyn* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 352556. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 446146. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 447102.

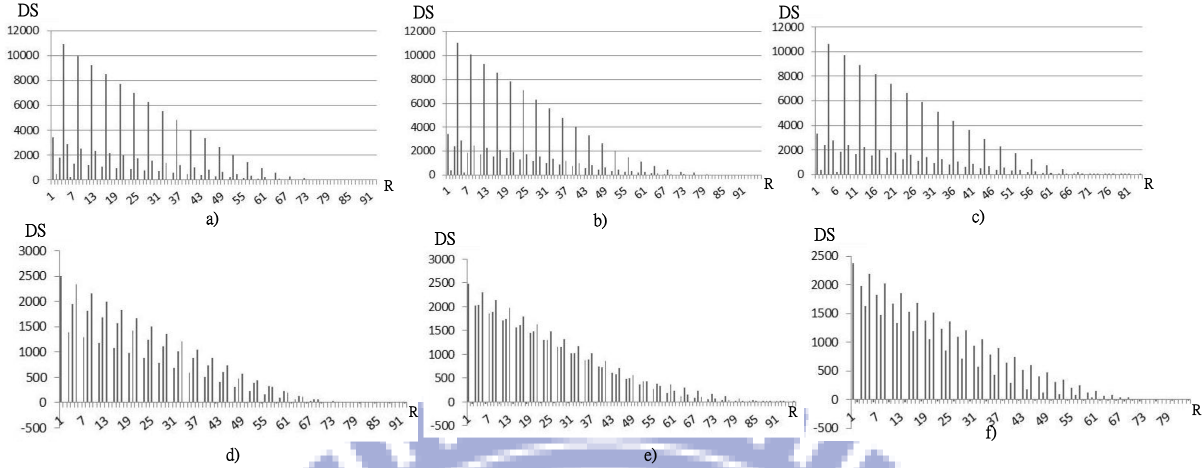


Figure 76: Data reuse characteristic for block 0 of *irreg* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 47309. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 53448. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 43212.

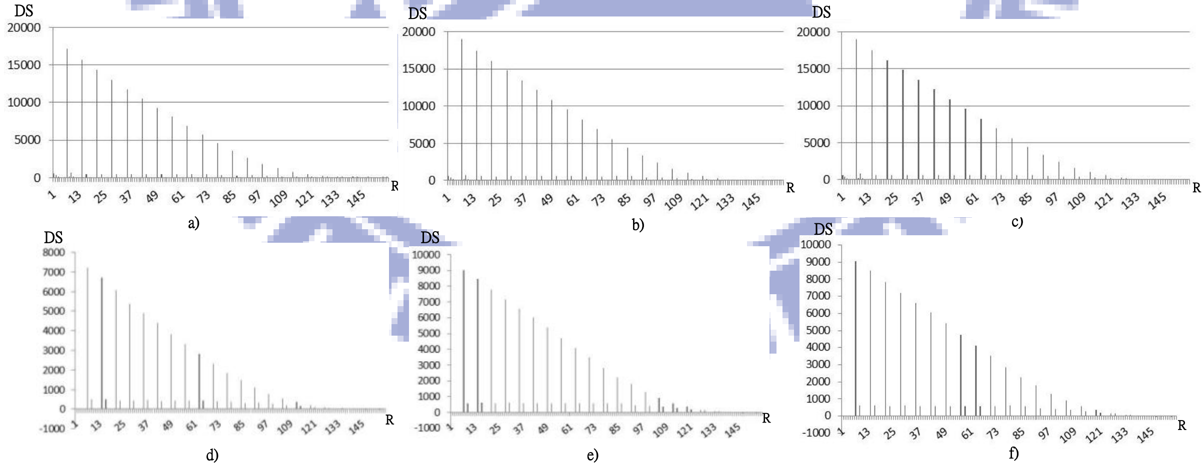


Figure 77: Data reuse characteristic for block 0 of *euler* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 59694. (e) Comparison prior to optimizations and after thread and warp clustering.

Difference in the reuse degree is 81755. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 82204.

Let's analyze first *sta*. In Figure 71(a), we see the resulting data reuse characteristic when applying the thread clustering technique. The contour is fairly the same as it for the data reuse characteristic prior to the optimization, presented in Section 9.1. However, a closer observation in fact reveals a very interesting variation in the magnitude for the reuse degree at very specific distances. In order to facilitate the comparisons, Figure 71(d) shows the difference between the data reuse degree in Figure 71(a) and the corresponding characteristic in Figure 29(a). This chart is obtained by subtracting the reuse degree at the corresponding reuse distance in the characteristic in Figure 29(a) from Figure 71(a). The sum of all the values in Figure 71(d) is shown at the bottom part of the figure. We can see that there's an increase in the total reuse degree of 17 units when compared to the case with no optimization. This sum, even though not a proper way to compare the changes in the reuse characteristic between the two cases, still provides a very intuitive way of understanding the improvement on the reuse behavior that the optimizations have over the data reuse patterns. However, as we shall see briefly, there are cases in which this magnitude can change negatively, requiring a different interpretation.

Notice also that in Figure 71(d), the portion of the reuse degree magnitudes at *RD=3, 5* have decreased, while the reuse degree for distances *RD=1* have increased by much more. Both quantities do not seem directly related. As this observation shows, the reuse degree for reuse distances has therefore increased for the short distances significantly, while there has been a decrement in the longer distances, but not that substantial.

A very particular behavior is observed for the case when thread clustering optimization technique is coupled with warp clustering, as described in Section 8. The resulting data reuse characteristic is presented in Figure 71(b). The contour is still the same, but the reuse distance domain has been reduced by 23.5% (from 17 down to 13). Figure 71(e) presents the comparison chart between Figure 71(b) and Figure 29(a). The total reuse degree increases even more up to 802, even though the reuse distance domain is also reduced, signaling a significant improvement. The fact that the reuse domain has been reduced means that there is less probability for data being evicted before being requested by a different MI at this specific

distance apart. In [18], Kuo et. al. report a running time performance improvement of around 60% for *sta* when applying thread clustering and warp clustering. The performance gain can be explained in part by the changes seen in the data reuse characteristic.

The data reuse characteristic for the case where the three optimization techniques (thread clustering, warp clustering and block scheduling) are applied is presented in Figure 71(c). For the thread block 0 of *sta* there is not much performance improvement when compared to the case where only thread clustering and warp clustering are applied. However, a performance improvement of more than 80% is reported in [18]. This is because the charts in Figures 71~77 only capture the reuse characteristic within the block itself, therefore any coding optimization that mimics a variation in the way blocks are scheduled will not be entirely visible within the block. However, for *nbf*, *moldyn*, *irreg* and *euler*, the block scheduling optimization does have an effect on the code. Recall that these optimizations are coding optimizations mimicking a scheduling approach. An analysis of the effects of this coding optimizations technique over the groups of applications used mentioned is left for further research.

Figures 78~84 presents the data reuse characteristic for the case where all blocks within the application are run in parallel when coding optimizations are applied.

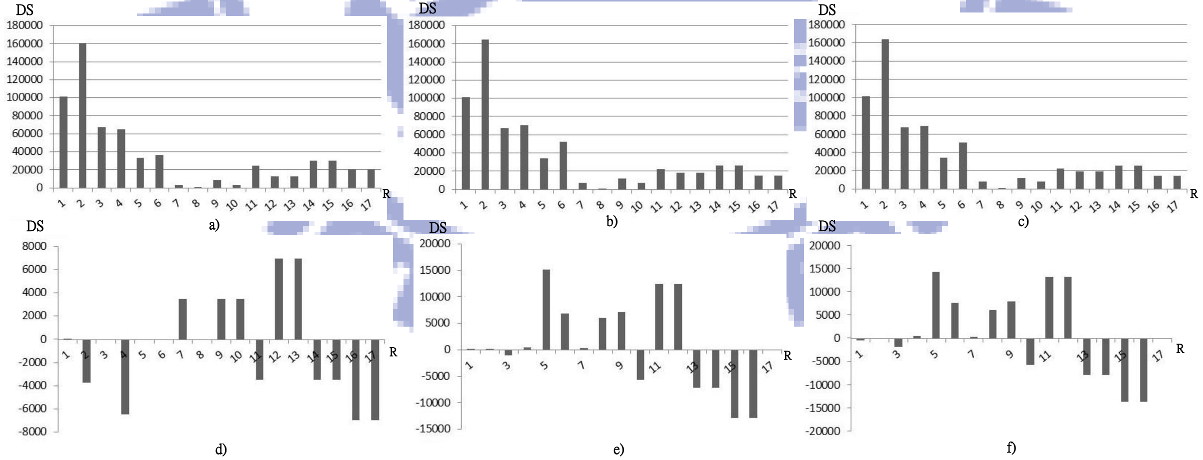


Figure 78: Data reuse characteristic for all blocks running in parallel of *sta* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is -10216. (e) Comparison prior to optimizations and after thread and warp

clustering. Difference in the reuse degree is 14229. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 11940.

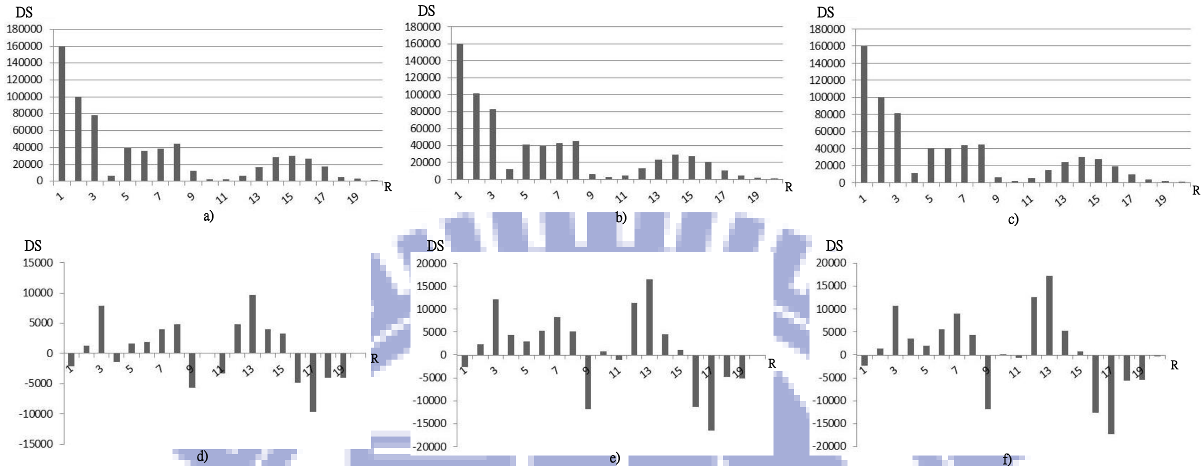


Figure 79: Data reuse characteristic for all blocks running in parallel of *gsim* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 8236. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 21363. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 17129.

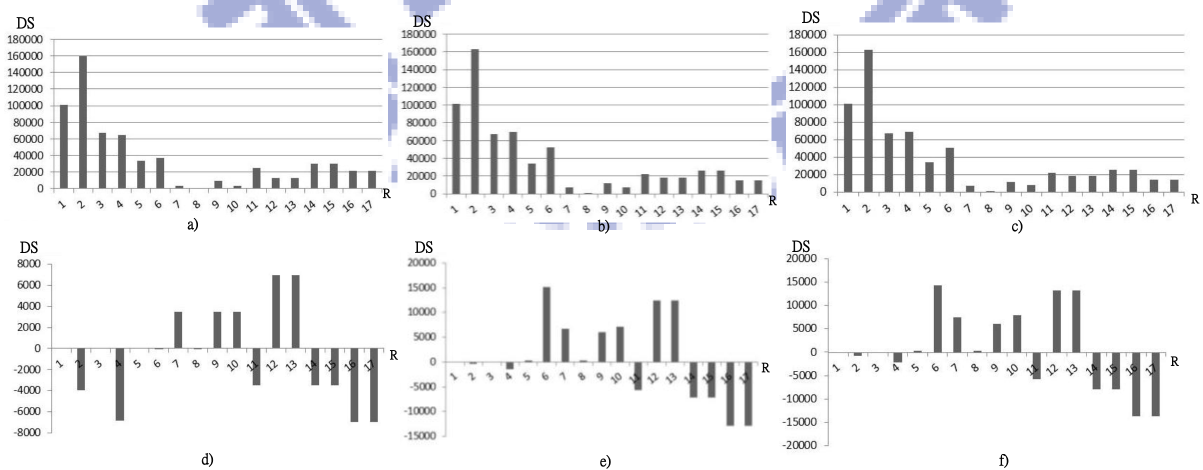


Figure 80: Data reuse characteristic for all blocks running in parallel of *bfs* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp

clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is -10812. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 13361. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 11324.

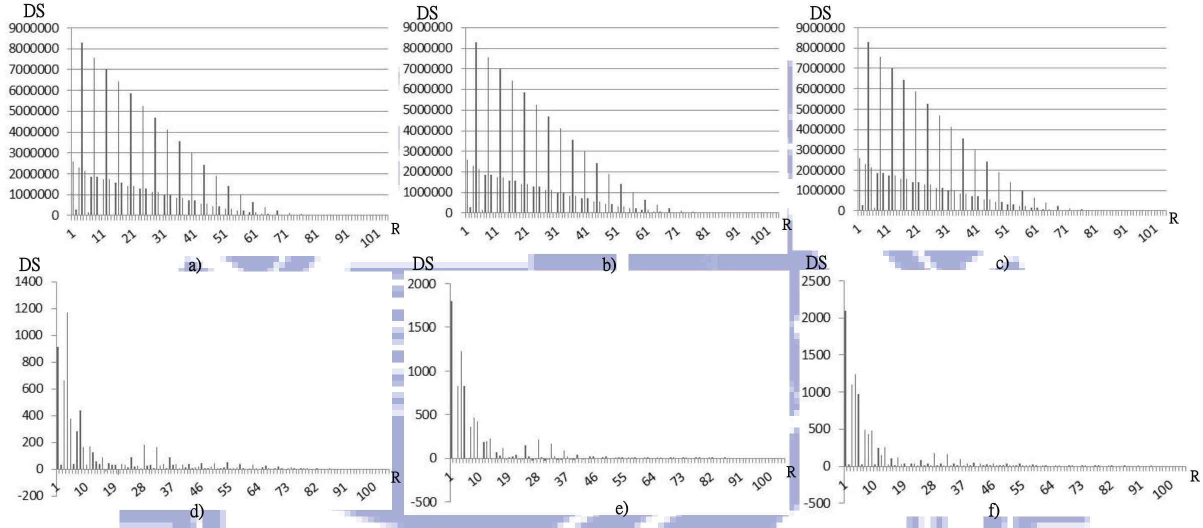


Figure 81: Data reuse characteristic for all blocks running in parallel of *nbf* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 6053. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 7400. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 9138.

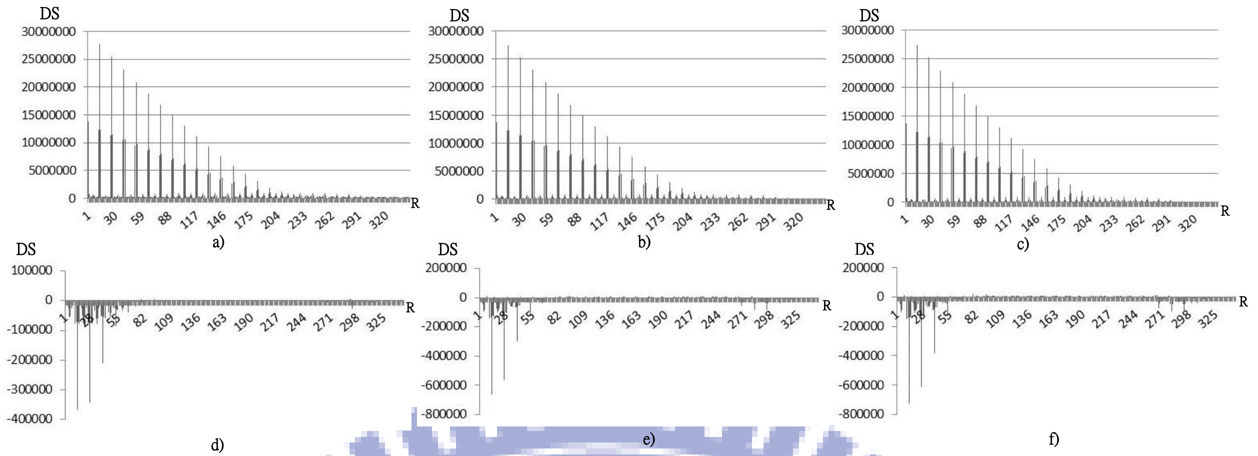


Figure 82: Data reuse characteristic for all blocks running in parallel of *moldyn* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is -3373236. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is -5170268. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is -5791542.

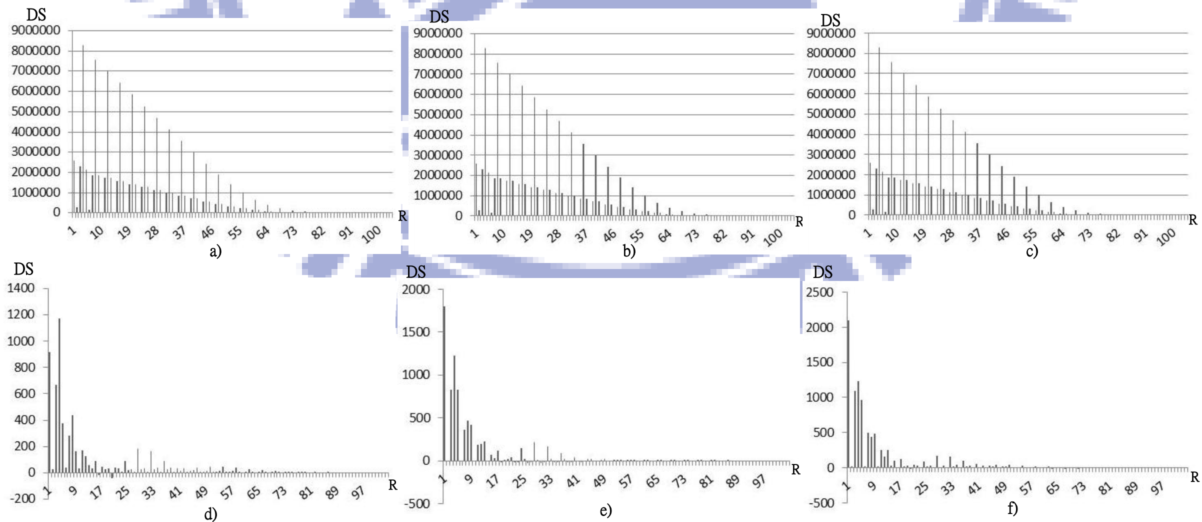


Figure 83: Data reuse characteristic for all blocks running in parallel of *irreg* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering.

Difference in the reuse degree is -4445. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is -4402. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is -4647.

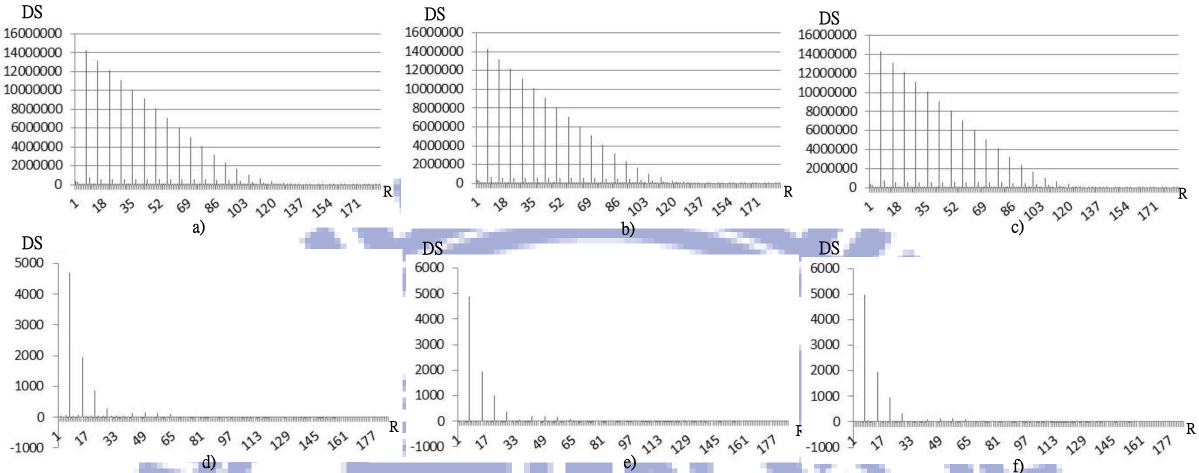


Figure 84: Data reuse characteristic for all blocks running in parallel of *euler* after coding optimizations. (a) After applying thread clustering. (b) After applying thread and warp clustering. (c) After applying thread clustering, warp clustering and block scheduling. (d) Comparison prior to optimizations and after thread clustering. Difference in the reuse degree is 8904. (e) Comparison prior to optimizations and after thread and warp clustering. Difference in the reuse degree is 7834. (f) Comparison prior to optimizations and after thread clustering, warp clustering and block scheduling. Difference in the reuse degree is 9172.

Notice once again the case for *sta* in Figure 78. The contour is the same, but the reuse degree magnitudes vary when compared to Figure 71. Figure 78(e) presents the comparison charts for the cases where the three coding optimizations are applied. In this case, the total reuse degree over the distance domain is drastically reduced after applying the optimizations. The explanation behind this is that when all blocks execute in parallel and are optimized, a situation occurs where the position of the MIs within the reference stream is reduced and/or the addresses accessed by one given MI are now accessed by a different MI earlier in the reference stream. This causes that one MI accesses data simultaneously for an MI in the original reference that would have accessed those addresses are a later position. When this occurs, the reuse degree for that distance will reduce, signaling some improvement under

these circumstances, even though the reuse characteristic does not present an increase in the overall reuse degree magnitudes.

Analyzing thoroughly why does in fact the reuse characteristic for the case when all blocks are running in parallel changes in such a way for when optimizations are applied, contrasting this with the behavior observed by block 0 in Figures 29~36, will require to analyze each of the blocks of the kernels. However, as we have explained, the huge amount of parallelism available in our idealized architecture model might be the cause behind this.

A reasonable conclusion to explain the performance improvement after optimizing is the increase of memory coalescing that rescheduling the blocks in a more efficient way cause. For now, neither the current methodology nor the analytical model provided here can quantify memory coalescing within a given MI. Therefore, the performance improvement cannot be explained only considering the data reuse characteristic as analyzed so far. The scope of this paper is not obtain a relationship between the reuse characteristic and the performance improvements, but the behavior just observed makes it necessary to make such analysis in future work.

A similar behavior is presented for all the applications shown. When the optimizations are applied, the total data reuse degree increases and, in some cases, the distance reuse domain is reduced. Notice how all the applications maintain without any significant changes the contour of their reuse characteristic, even when the reuse domain is shrunk.

X. RELATED WORK

In [8, 10, 23, 24, 25, 26] it was demonstrated the importance of data reuse in CMPs (multi/many-core) and SIMT processors as a means to improve performance for different benchmarks and application domains. As noted in [27], understanding data reuse becomes important in many-core systems that are limited by memory bandwidth, as the case for SIMT processors [28]. Exploiting data reuse saves memory bandwidth, because less accesses are required [12, 13, 29]. The capacity limitaiton is significant in SIMT processors, due to the relative small caches for the amount of threads [12, 29]. In [9], Kuo et. al. explain that the capacity constraint can cause contention and destructive sharing in certain cases. Understanding whether these phenomena are due to application behavior or to limitations of the subsystem is necessary. Also, understanding data reuse is key to improve the management of the memory resources of SIMT applications, which are relatively scarce, but are critical in boosting performance [3].

In [8], Jia et. al. propose a taxonomy of the data reuse behavior based on the abstractions of the execution model and proposed compiler-based techniques to analyze the reuse behavior. For this, they use the intrinsic relationship of the thread identification mechanism and their portion of the total data set. This approach is limited in that it cannot analyze memory accesses whose addresses are unknown until runtime. Also, they assume that applications running on a GPU present negligible cross-block data reuse. This assumption is valid for a specific set of applications. However, as we have seen in the results section of our work, there are applications that do present considerable cross-block data reuse.

In [9], Kuo et. al. developed a standalone library that builds a hypergraph that represents the data sharing between different run time abstractions of GPUs. Based on this analysis, coding optimizations are performed to CUDA kernels that mimic scheduling mechanisms. The gain in performance is signifcant.

Most of the efforts in data reuse characterization and analysis, such as [8] and [9], are based on static analyses. These works don't provide a way to quantify the data reuse behavior, or to characterizae any locality dimensions: whole-program, in program code, in program data, over time (program phases), interaction between programs, as explained in [10]. Arguably, some of

these dimensions might not be totally applicable to SIMT environment. For example, analyzing the different phasesof computation might not be efficient or relevant for certaub kernels in SIMT processors, since these kernels are relatively short-lived (when compared to threads in a CMP environment). However, analogous concepts to whole-program locality bare significant relevance, in our view, because of the necessity to analyze memory access patterns and to exploit data reuse of GPU kernels. Our model attempts to create a locality signature of the program totally isolated from particular architectural limitations. In this way, we quantify and visualize the specific reuse behavior of the application. This is useful in assessing the improvement due to code optimizations over the data reuse behavior (temporal/spatial locality) and performance since, as exposed in [30], these two, sometimes, do not relate to each other in a straightforward way.

In contrast to [8] and [9], for our work we preffered not to employ static analyses, using memory traces instead. Our main reason is because such static methods cannot account for the indeterministism of certain portions of the kernel. Memory traces can offer profound insight of the applications, allowing for more refined analyses to capture runtime variations, model them and increase predictability [10]. However, the methodology we propose is relatively straightforward, and has not been extended to provide prediction of any kind. In this work, we attempt to provide the locality signature of the SIMT applications using a new metric: the data reuse degree, and a variation of the reuse distance concept.

Locality characterization of applications using reuse distance profiles, concept introduced by [15] as LRU stack distance, has been widely used to predict cache performance and measure different dimensions of locality. It has been used for systems with serialized memory behavior (corresponding to uniprocessor systems), as is the case in [30, 31, 32] and also for systems in which concurrency in memory access is allowed, such as in CMPs [11, 33, 34]. The reuse distance profiles so obtained can be used to predict cache miss rates, under assumptions of other cache parameters (LRU policy, constant associativity, etc), and analyze different dimensions of locality [10, 14]. Additional complications arise when analyzing applications running on CMP systems, as explained in [11], but prediction is still feasible.

The methodology used to capture program locality based on the data reuse behavior used for CMP systems detailed in [11] is not totally adequate to model locality for SIMT processors.

The reason is that private caches in CMP systems are in fact exclusive to each core, whereas the privacy of the caches in SIMT processor is not. In this case, all threads within one core cluster utilize the same caches with a particular portion assigned to them when necessary. Trying to even perform the analysis on the portion assigned to each thread becomes impractical. This is because threads in SIMT threads are smaller in stream size and are relatively short-lived when compared to their counter-parts in most applications running on CMPs. Our work proposes a new methodology totally different than the use in CMP systems, and proposes a novel approach to analyzing locality in SIMT processors.

In [35], Tang et. al. offer an analytical model based on stack distance to predict cache miss rates on GPUs. Tang et. al. acknowledge the impact that the programming model of SIMT processors has on analyzing locality. Also, they consider very specific constraints and characteristics of the memory susbsytem (Effect Point) coupled with program behavior (Access Point) to perform the stack distance profiling. Their work is focused on predicting miss rates and the ocurrence of contention. The histograms obtained by this methodology are highly dependent on the cache parameterers assumed (associativiy, replacement policy). Therefore, it is not inherent to the kernel itself. Also, they assume that cross-block data reuse is negligible. As mentioned before, the validity of the assumption that there is no cross-block reuse depends on the particular kernel.

We attempt to improve over the approach proposed in [35] by quantifying the data reuse on per memory instruction basis, and building a histogram that captures the data reuse between two memory instructions at varying distances in an efficient way. In contrast to [35], we don't build stacks and traverse them in every access to build the histogram, since this makes problematic the creation of the histograms. In Section 2, we explained that this approach can alter the actual reuse distance depending on the order in which the addresses traverse the stack (or any other data structure for that matter). This problem is not addressed at all in [35]. Another improvement when compared to [35], is that we perform our analysis independent on particulars of the cache parameter, considering the programming model and the amount of available parallelism given the code structure of the kernel.

XI. CONCLUSIONS

In this work, we have a shown a novel methodology to analyze locality in SIMT applications. This methodology is totally architecture agnostic, dependent only on the programming and runtime models, and it allows to model the execution of the kernels under varying degrees of parallelism. In addition, we also define new metrics and a new reuse distance model which we use to obtain the data reuse characteristic, a term we coined to refer to a specific locality property of SIMT applications: data sharing or data reuse. Coupled with this, we also a developed a framework, the Locality Analyzer, that implements our methodology with the analyses detailed throughout the document. Our framework is very flexible and scalable, which will allow performing different types of analyses.

We were also capable of observing very interesting properties of the kernels we tested when showing their quantified data reuse characteristic in a visual way. We demonstrated that the kernels we tested do in fact present a significant amount of data reuse across different blocks in the kernel. We also showed the way different parallelism constraints alter the reuse behavior of the kernel, and the way SIMT architectural limitations basically damp out the reuse characteristic, making it smoother and distributing the memory requests across a larger number of memory instructions.

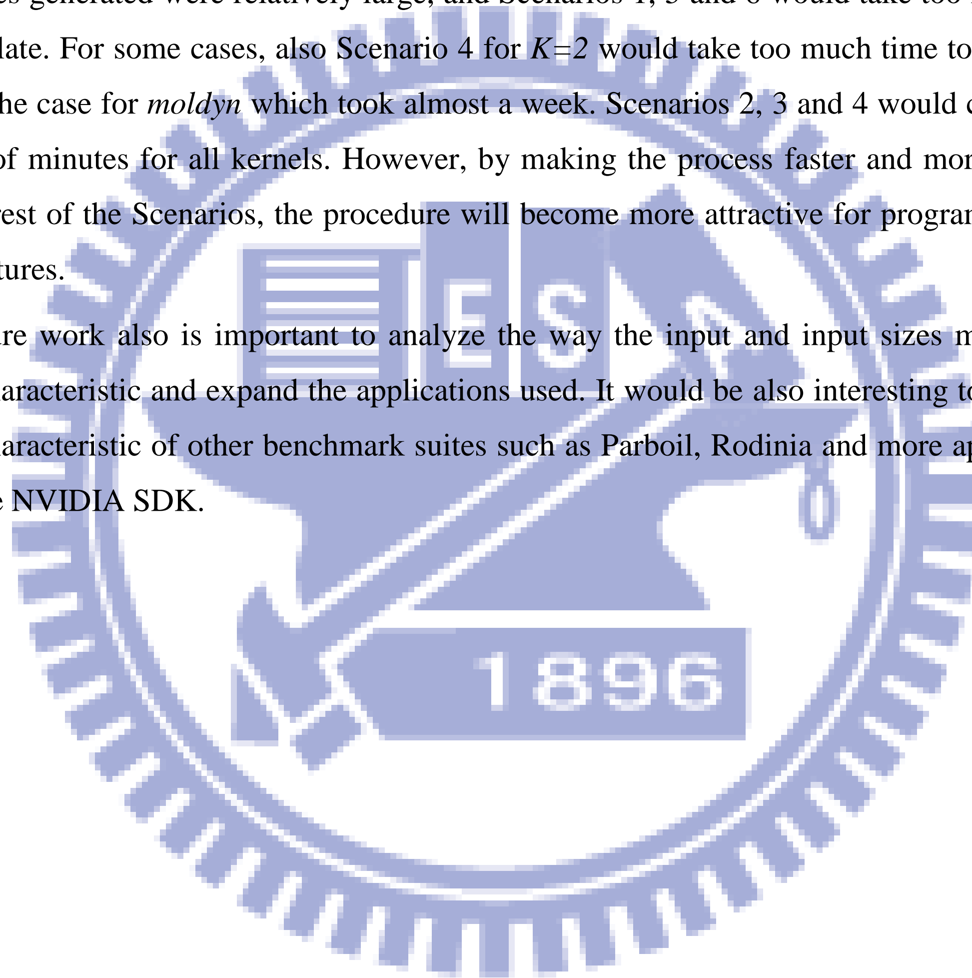
We also show the way certain coding optimization techniques modify the reuse characteristic of the applications. This accounts for part of the performance improvement that the kernels incur when running on a real GPU system, but cannot explain all of it, as we could see when analyzing the reuse characteristic under specific constraints.

When observing the results, we realized that further work is required in order to define properly the signature reuse characteristic of the kernels and correlate the behavior observed with performance in real architectures. First, we need make a more thorough analysis of the variability presented by the blocks of one kernel within the perspective of the data reuse characteristic. This will provide a different way to categorize applications based on a truly architecture agnostic metrics. Also, it is also necessary to test different scheduling policies in order to analyze their effect on the data reuse characteristic, and demonstrate if it is whether or not beneficial to control the scheduling policies of real architectures. Moreover, it is necessary

to extend the analyses to quantify coalescing, if possible, and to estimate the occurrence of contention and miss rates, since such analysis will allow establishing the relationship between data reuse characteristic and the performance observed. This will enable to explain better the performance improvements when running applications on real architectures.

In addition, it will also be necessary to optimize the algorithms used in the analytical framework and to calculate the reuse degree. For some of the kernels used in this work, the trace files generated were relatively large, and Scenarios 1, 5 and 6 would take too much time to calculate. For some cases, also Scenario 4 for $K=2$ would take too much time to complete, as was the case for *moldyn* which took almost a week. Scenarios 2, 3 and 4 would complete a couple of minutes for all kernels. However, by making the process faster and more efficient for the rest of the Scenarios, the procedure will become more attractive for programmers and architectures.

For future work also is important to analyze the way the input and input sizes modifiy the reuse characteristic and expand the applications used. It would be also interesting to show the reuse characteristic of other benchmark suites such as Parboil, Rodinia and more applications from the NVIDIA SDK.



XII. REFERENCES

[1] J. Hennessy; D. Patterson, 5th ed., Computer Architecture. Elsevier, 2012.

[2] Brodtkorb, A.R.; Dyken, C.; Hagen, T. R.; Hjelmervik, J.M.; Storaasli, O.O. "State-of-the-art in Heterogeneous Computing". In Journal of Scientific Programming, 2010.

[3] Lashgar, A.; Baniasadi, A. "Performance in GPU Architectures: Potentials and Distances". In Workshop on Duplicating, Deconstructing and Debunking (WDDD) in conjunction with ISCA, 2011.

[4] A. Kerr; G. Diamos; S. Yalamanchili. "A Characterization and Analysis of PTX Kernels", in IEEE International Symposium on Workload Characterization, October 2009.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron. Rodinia: A Benchmark Suite For Heterogenous Computing. In *Proc. of IEEE International Symposium of Workload Characterization*. 2009.

[6] Goswami, N., Shankar, R., Joshi, M., Li, T. "Exploring GPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation implications". In IEEE International Symposium on Workload Characterization, December 2010.

[7] Yuan, G.L.; Fung, W.W.L.; Wong, H.; Aamodt, T.M. "Analyzing CUDA workloads using a detailed GPU Simulator". International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2009.

[8] Jia, W.; Shaw, K.; Martonosi, M. “”Characterizing and Improving the Use of Demand-Fetched Caches in GPU". In International Conference on Supercomputing, 2012.

[9] Kuo, H.K.; Lai, B.C.C.; Jou, J.Y. "A Cache Hierarchy Aware Thread Mapping Methodology for GPGPUs".

[10] Zhang, E Z.; Jiang, Y.; Shen, X. "The Significance of CMP Cache Sharing on Contemporary Multithreaded Applications". In Annual Symposium on Principles and Practice of Parallel Programming. 2010.

[11] Schuff, D.L.; Parsons, B.S.; Pai, V.S. "Multicore-Aware Reuse Distance Analysis". In Parallel And Distributed Processing, Workshops and PhD Forum (IPDPSW), 2010.

[12] NVIDIA Corporation. NVIDIA GeForce GTX 680. 2012.

[13] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. Version 4.2. 2012.

[14] Ding, C.; Zhong, Y. "Predicting Whole-Program Locality Through Reuse Distance Analysis". In Conference of Programming Languages and Implementation, 2003.

[15] Mattson, R.L.; Gecsei, J.; Slutz, D.R.; Traiger, I.L. "Evaluation Techniques for storage hierarchies". In IBM Systems Journal, 1970.

[16] Niu, Q.; Dinan, J.; Lu, Q.; Sadayappan, P. "PARDA: A Fast Parallel Reuse Distance Analysis Algorithm". In IEEE International Symposium on Parallel and Distributed Processing Symposium, May 2012.

[17] G. Diamos, A. Kerr, S. Yalamanchili. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *Proc. of PACT*, pages 331-342. 2010.

[18] H.-K. Kuo, K.-T. Chen, B.-C. C. Lai and J.-Y. Jou, "Thread Affinity Mapping for Irregular Data Access on Shared Cache GPGPU," In Proc. Asia and South Pacific Design Automation Conf., pp. 659-664, 2012.

[19] "ITC'99 Benchmarks", http://www.cad.polito.it/downloads/tools/benchmarks.html

[20] Das, R.; Uysal, M.; Saltz, J.; Hwang, Y.-S. "Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures". In Journal of Parallel and Distributed Computing, vol. 22, no. 3, pp. 462-478, 1994.

[21] Han, H.; Tseng, C.-W. "Exploiting Locality for Irregular Scientific Codes". In IEEE Transactions on Parallel and Distributed Systems, vol. 17, no.7, pp. 606-618, 2006.

[22] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J.W.; Skadron, K. "Rodinia: A Benchmark Suite For Heterogenous Computing". In *Proc. of IEEE International Symposium of Workload Characterization*. 2009.

[23] Krishna, A; Samih, A.; Solihin, Y. "Data Sharing in Multi-Threaded Applications and Its Impact on Chip Design". In Proc. of IEEE International Symposium on Performance Analysis of System and Software, 2012

[24] NVIDIA Corporation. NVIDIA CUDA C Programming Guide.

[25] Zhang, Y.; Kandemir, M.; Yemliha, T. "Studying Inter-Core Data Reuse in Multicores". In SIGMETRICS, 2011.

[26] Wu, C.C.; Wei, K.C.; Lin, T.H. "Optimizing Dynamic Programming on Graphics Processing Units via Data Reuse and Data Prefetch with Inter-Block Barrier Synchronization". In International Conference on Parallel and Distributed Systems.

[27] Cong, J.; Zhang, P.; Zou, Yi. "Combined Loop Transformation and Hierarchy Allocation for Data Reuse Optimization". In International Conference on Computer-Aided Design (ICCAD), 2011.

[28] Gou, C.Y.; Gaydadjiev, G.N. "Addressing GPU On-Chip Shared Memory Bank Conflicts Using Elastic Pipeline". In International Journal of Parallel Programming, 2012.

[29] NVIDIA Corporation. NVIDIA's Next Generation CUDA Computer Architecture: Fermi. 2009.

[30] Pyo, C.; Lee, G. "Reference Distance as a Metric for Data Locality". In High Performance Computing on the Information Superhighway, 1997.

[31] Beyls, K.; D'Hollander, Erik H. "Reuse Distance as a Metric for Cache Behavior". In Proc. of the IASTED Conference on Parallel and Distributed Computing and Systems, 2001.

[32] Fang, C.; Carr, S.; Önder, S.; Wang, Z. "Reuse-distance-based Miss-rate Prediction on a Per Instruction Basis". In Workshop on Memory System Performance, 2004.

[33] Petoumenos, P.; Keramidas, G.; Zeffer, H.; Kaxiras, S.; Hagersten, E. "Modeling Cache Sharing on Chip Multiprocessor Architectures". In International Symposium on Workload Characterizations, 2006.

[34] Xu, C.; Chen, X.; Dick, R.P.; Mao, Z.M. "Cache Contention and Application Performance Prediction for Multi-Core Systems". In International Symposium on Performance Analysis of Systems and Software (ISPASS), 2010.

[35] Tang, T.; Yang, X.; Lin, Y. "Cache Miss Analysis for GPU Programs Based on Stack Distance Profile". In International conference on Distributed Systems, 2011.