Characterization of Data Locality Potential of CPU and GPU Applications through Dynamic Analysis

DISSERTATION

Presented in Partial Fulfillment of the Requirements
for the Degree Doctor of Philosophy
in the Graduate School of The Ohio State University

By

Naznin Fauzia,

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

P. Sadayappan, Advisor

Atanas Rountev

Gagan Agrawal

ABSTRACT

Emerging computer architectures will feature drastically decreased flops/byte (ratio of peak processing rate to memory bandwidth), as highlighted by recent studies on Exascale architectural trends. Further, flops are getting cheaper while the energy cost of data movement is increasingly dominant. The understanding and characterization of data locality properties of computations is critical in order to guide efforts to enhance data locality.

Reuse distance analysis of memory address traces is a valuable tool to perform data locality characterization of programs. A single reuse distance analysis can be used to estimate the number of cache misses in a fully associative LRU cache of any size, thereby providing estimates on the minimum bandwidth requirements at different levels of the memory hierarchy to avoid being bandwidth bound. However, such an analysis only holds for the particular execution order that produced the trace. It cannot estimate potential improvement in data locality through dependence preserving transformations that change the execution schedule of the operations in the computation.

In this dissertation, we present a novel dynamic analysis approach to characterize the inherent locality properties of a computation and thereby assess the potential for data locality enhancement via dependence preserving transformations. The execution trace of a code is analyzed to extract a computational directed acyclic graph (CDAG) of the data dependences. The CDAG is then partitioned into convex subsets, and the convex partitioning is used to reorder the operations in the execution trace to enhance data locality. The approach

enables us to go beyond reuse distance analysis of a single specific order of execution of the operations of a computation in characterization of its data locality properties. It can serve a valuable role in identifying promising code regions for manual transformation, as well as assessing the effectiveness of compiler transformations for data locality enhancement. We demonstrate the effectiveness of the approach using a number of benchmarks, including case studies where the potential shown by the analysis is exploited to achieve lower data movement costs and better performance.

Effective parallel programming for GPUs requires careful attention to several factors, including ensuring coalesced access of data from global memory. There is a need for tools that can provide feedback to users about statements in a GPU kernel where non-coalesced data access occurs, and assistance in fixing the problem. In this dissertation, we address both these needs. We develop a two step framework, where dynamic analysis is first used to detect uncoalesced accesses by instrumenting PTX code to generate traces. Transformations to optimize global memory access by introducing coalesced access whenever possible are achieved using feedback from the dynamic analysis or using a model-driven approach. Experimental results demonstrate the benefits of use of the tools on a number of benchmarks from the Rodinia and Polybench suites.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Sadayappan from the bottom of my heart. I would be nowhere near where I am today without his guidance and help as an advisor. Thank you for all your patience with me during my struggling times and giving me hopes. It was an honor to work with you and learn from you. I would also like to sincerely thank Dr. Louis-Noël Pouchet for mentoring me throughout this whole process. I have learnt a lot from you and I do appreciate all your help.

I am thankful to my dissertation committee, Dr. Gagan Agrawal and Dr. Atanas Nasko Rountev. Thank you Nasko for all those brainstorming meetings and helping me to find the path whenever I was looking for one. I would also like to thank my collaborators Dr. Fabrice Rastello and Dr. J Ramanujam for their contributions. Special thanks to my colleague and friend Venmugil Elango for his help and contributions for the TACO paper.

Thanks to my dear friends Tanima Dey and Ingy Youssef who were beside me during my ups and downs. Thanks Pai Wei Lai for proof reading my papers. Also thanks to all my labmates, specially - Arash, Sanket, Mahesh, Qingpeng, Kevin, Martin, Tom and Justin.

Finally, thanks to my loving husband Humayun Arafat for having faith in me. You were and always will be a true inspiration for me and I could not do this without you being on my side. Thanks to my parents for all their love and wishes for me. Thanks to the Almighty for everything He has given me. And thanks to my unborn angel for bringing light and laughter to my life.

VITA

June 3, 1984 . . . . . Born: Dhaka, Bangladesh

June 2007 . . . . . Bachelor of Science
Computer Science and Engineering
Bangladesh University of Engineering and Technology,
Dhaka, Bangladesh

2007-2008 . . . . . Lecturer,
Computer Science and Engineering
Bangladesh University of Engineering and Technology,
Dhaka, Bangladesh

September 2010—May2012 . . . . . Graduate Research Associate,
Computer Science and Engineering,
The Ohio State University,
Columbus, OH, USA

September 2012—Present . . . . . Graduate Teaching Associate,
Computer Science and Engineering,
The Ohio State University,
Columbus, OH, USA

Summer 2012 . . . . . Graduate Intern,
Intel Compiler Group,
Intel Corporation,
Santa Clara, CA, USA

Summer 2013 . . . . . Graduate Intern,
CCE Compiler Group,
Cray Inc.,
Saint Paul, MN, USA

PUBLICATIONS

Research Publications

Naznin Fauzia, Louis-Nol Pouchet and P. Sadayappan
Characterizing and Enhancing Global Memory Data Coalescing on GPUs.
In *The International Symposium on Code Generation and Optimization (CGO)*, 2015

Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J. Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Nol Pouchet and P. Sadayappan
Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential.
In *The ACM Transactions on Architecture and Code Optimization (TACO)*, 2014

Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J. Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Nol Pouchet and P. Sadayappan
Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential.
Technical Report OSU-CISRC-9/13-TR19, 2013

Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Nol Pouchet, J. Ramanujam, P. Sadayappan and Vivek Sarkar
Analytical Bounds for Optimal Tile Size Selection.
In *International Conference on Compiler Construction (CC)*, 2012

Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Nol Pouchet, Atanas Rountev and P. Sadayappan
Dynamic trace-based analysis of vectorization potential of applications.
In *Programming Language Design and Implementation (PLDI)*, 2012

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

High Performance Computing	Prof. P. Sadayappan
Software Engineering	Prof. Atanas Rountev
Computer Networks	Prof. Dong Xuan

TABLE OF CONTENTS

Page

Abstract ii

Acknowledgments iv

Vita v

List of Figures x

List of Tables xiii

List of Algorithms xiv

Chapters:

1. Introduction 1

2. Background 7

   2.1 Reuse Distance Analysis 7
   2.1.1 Example 9
   2.1.2 Limitations of reuse distance analysis 12
   2.2 Dynamic Analysis 14
   2.3 DDG Generation 16

3. Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential 17

   3.1 Introduction 17
   3.2 Background & Overview of Approach 20
   3.2.1 Benefits of the proposed dynamic analysis 21

3.2.2 Overview of Approach . . . 22
3.3 Convex Partitioning of CDAG . . . 26
3.3.1 Definitions . . . 27
3.3.2 Forming Convex Partitions . . . 29
3.3.3 CDAG Traversal: Breadth-first Versus Depth-first . . . 31
3.3.4 Multi-level Cache-oblivious Partitioning . . . 33
3.3.5 Complexity Analysis . . . 34
3.4 Experimental Results . . . 36
3.4.1 Experimental Setup . . . 37
3.4.2 Case Studies . . . 38
3.4.3 Dataset Sensitivity Experiments . . . 58
3.5 Related Work . . . 62
3.6 Discussion . . . 63
3.7 Conclusion . . . 66

4. Convex Partitioning using Loop Induction Variable Information . . . 68

4.1 Introduction . . . 68
4.2 Background . . . 70
4.2.1 Canonical Induction Variables . . . 70
4.2.2 Impact of Heuristic Parameters . . . 71
4.3 Overview of Approach . . . 75
4.3.1 Formatting the Induction Variables . . . 76
4.3.2 Preprocessing of Dynamic Dependency Graph . . . 77
4.3.3 Convex-partitioning heuristic . . . 79
4.4 Experimental Results . . . 83
4.4.1 Experimental Setup . . . 83
4.4.2 Results . . . 84
4.5 Conclusion . . . 88

5. Characterizing and Enhancing Global Memory Data Coalescing on GPUs . . . 90

5.1 Introduction . . . 90
5.2 Background and Overview . . . 92
5.2.1 GPU Architecture . . . 92
5.2.2 Global Memory Coalescing . . . 93
5.2.3 Overview of the Framework . . . 94
5.3 Dynamic Analysis of Uncoalesced Accesses . . . 96
5.3.1 Instrumentation and Execution . . . 96
5.3.2 Dynamic Analysis Algorithm . . . 97
5.4 Compiler Transforms for Data Coalescing . . . 99
5.4.1 Overview . . . 99

5.4.2 Computing a New Thread Block Geometry 100
5.4.3 Geometry and Thread Code Transformations 103
5.4.4 Other Static Transformations 109
5.5 Experimental Results 112
5.5.1 Experimental Protocol 112
5.5.2 Dynamic Analysis Results 112
5.5.3 Static Transformation Results 115
5.5.4 Discussions 119
5.6 Related Work 120
5.7 Conclusion 121

6. Conclusion 122

Bibliography 124

LIST OF FIGURES

Figure **Page**

2.1 Example data reference trace 7

2.2 Example: Single-sweep two-point Gauss-Seidel code, (a) Untiled and (b) Tiled 9

2.3 Reuse distance profile:cache hit rate 10

2.4 Reuse distance profile:cache miss rate 11

2.5 Reuse distance profile:memory bandwidth demand 11

3.1 Reuse distance analysis for Householder and Floyd-Warshall 22

3.2 CDAG for Gauss-Seidel code in Fig. 2.2. Input vertices are shown in black, other vertices represent operations performed. 24

3.3 Convex-partition of the CDAG for the code in Fig. 2.2 for N = 10. 25

3.4 Floyd-Warshall all-pairs shortest path 39

3.5 Floyd-Warshall: Analysis and performance improvements due to tiling . . 40

3.6 Tiled Floyd-Warshall implementation 42

3.7 Givens Rotation 43

3.8 Results with different heuristics for Givens Rotation 44

3.9 Modified Givens Rotation before tiling 45

3.10 Modified and automatically tiled Givens Rotation 46

3.11 Givens Rotation: performance improvements due to tiling 47

3.12 Householder computation 48

3.13 Results with different heuristics for Givens Rotation 49

3.14 Results with different heuristics for 470.lbm 51

3.15 Results with different heuristics for 410.bwaves 52

3.16 Results with different heuristics for 437.leslie3d 53

3.17 Odd-Even sort on linked list 54

3.18 Tiled odd-even sort 55

3.19 Odd-Even sort: Performance improvements due to tiling 56

3.20 LU Decomposition 58

3.21 Sensitivity analysis for odd-even sort 59

3.22 Sensitivity analysis for LUD 59

3.23 Performance for odd-even sort 61

3.24 Performance for LUD 61

4.1 Results with different heuristics for Jacobi-2D 72

4.2 Results with different heuristics for matrix multiplication 74

4.3 Convex Partitioning Approach for 2D Loop Iteration Space with $T = 3$ . . 80

4.4 Matmult 84

4.5 Jacobi 2D 85

4.6 Floyd-Warshall all-pairs shortest path 86

4.7 Givens Rotation 87

4.8 Householder 87

5.1 Overall Flow Chart of Our Approach 95

5.2 Partial Sum Method for Load Optimization 110

5.3 Effective bandwidth on Tesla K10. Y-axis is in logarithmic scale 116

5.4 Effective bandwidth on Tesla K20. Y-axis is in logarithmic scale 117

LIST OF TABLES

Table **Page**

5.1 Sample Output of Dynamic Analysis 112

5.2 Benchmarks with Uncoalesced Access in Rodinia and Polybench/GPU . . . 113

5.3 Execution times of applications on Tesla K20 114

5.4 Execution times using different thread block geometry on Rodinia 118

LIST OF ALGORITHMS

Algorithm **Page**

3.1 GenerateConvexComponents(*G, C, CF*) 29

3.2 UpdateListOfReadyNodes(*R, n*) 30

3.3 updateLiveSet(*p, n, C*) 31

3.4 selectBestNode(*R, cc, priority, n*) 32

3.5 MultiLevelPartitioning(*G, C, Priority, factor*) 34

3.6 updateLiveSet(*p, n, C*) 34

4.1 GenerateConvexPartitions(G,T) 81

5.1 Memory Trace Analysis Algorithm 98

5.2 Polyhedral optimization flow 105

5.3 Transform Consecutive Load Operations by Single Thread 111

CHAPTER 1

Introduction

Data locality optimization is an important problem in the modern high-performance era. Advances in technology over the last few decades have yielded significantly different rates of improvement in the computational performance of processors relative to the speed of memory access. Computation latency has been decreasing in a much higher rate than the memory access latency. Therefore, efficient memory access is required to optimize the overall program performance. The problem is equally important for both sequential CPU applications and highly parallel GPU applications. In this dissertation, we approach the problem of program characterization in terms of their memory access efficiency with a novel dynamic analysis strategy and show the effectiveness of our tools over existing ones.

Dynamic Analysis for Characterization of Data Locality Potential of Sequential Programs. The recent Intel Core i7 processor has an operation latency of 4ns and a memory latency of 37 ns, illustrating an order of magnitude shift in the ratio of operation latency to memory access latency. Because of the significant mismatch between computational latency and throughput when compared to main memory latency and bandwidth, the use of hierarchical memory systems and the exploitation of significant data reuse in the higher (i.e., faster) levels of the memory hierarchy is critical for high performance. Techniques

such as pre-fetching and overlap of computation with communication can be used to mitigate the impact of high memory access latency on performance, but the mismatch between maximum computational rate and peak memory bandwidth is much more fundamental; the only solution is to limit the volume of data movement to/from memory by enhancing data reuse in registers and higher levels of the cache.

With future systems, the cost of data movement through the memory hierarchy is expected to become even more dominant relative to the cost of performing arithmetic operations, both in terms of throughput and energy. Optimizing data access costs will become ever more critical in the coming years. Given the crucial importance of optimizing data access costs in systems with hierarchical memory, *it is of great interest to develop tools and techniques to assess the inherent data locality characteristics of different parts of a computation, and the potential for data locality enhancement via dependence preserving transformations*.

Although reuse distance analysis has found many uses in characterizing data locality in computations, it has a fundamental constraint: *The analysis is based on the memory address trace corresponding to* ***a particular execution order*** *of the operations constituting the computation.* Thus, it does not in any way account for the possibility of alternate valid execution orders for the computation that may exploit much better data locality. While reuse distance analysis provides a useful characterization of data locality for a given execution trace, it fails to provide any information on the potential for improvement in data locality that may be feasible through valid reordering of the operations in the execution trace.

In this dissertation, we present a novel dynamic analysis approach to provide insights beyond that possible from standard reuse distance analysis. The analysis seeks to characterize the *inherent data locality potential* of the implemented algorithm, instead of the

reuse distance profile of the address trace from a specific execution order of the constituent operations. We develop graph partitioning techniques that could be seen as a generalization of loop tiling, but considering arbitrary shapes for the tiles that enable atomic execution of tiles.

Instead of simply performing reuse distance analysis on the execution trace of a given sequential program, we first explicitly construct a computational directed acyclic graph (CDAG) to capture the statement instances and their inter-dependences, then perform convex partitioning of the CDAG to generate a modified dependence-preserving execution order with better expected data reuse, and finally perform reuse distance analysis for the address trace corresponding to the modified execution order. We apply the proposed approach on a number of benchmarks and demonstrate that it can be very effective.

Characterizing and Enhancing GPU Global Memory Access. While the previous tool targets sequential CPU applications, in another piece of work we target highly parallel GPU applications. Parallel programming is hard and programming GPUs is even harder. In order to achieve high performance, it is essential to address many aspects, such as avoidance/minimization of control divergence among threads, ensuring sufficiently high degrees of parallelism to effectively mask main memory latency, and achieving coalesced access to global memory. In contrast to shared-memory parallel programs for CPUs, where stride-1 access to memory by each thread is very efficient, for effective utilization of memory bandwidth on GPUs, adjacent threads must access adjacent data elements in global memory. Thus coalesced access generally implies that a single thread will not access contiguous

memory locations in adjacent iterations of a loop. Therefore many programs directly converted from OpenMP to CUDA without a fundamental change to the loop structure exhibit uncoalesced access to global memory.

While attempts have been made to develop tools to ease the development of GPU applications, many existing CUDA applications still suffer from uncoalesced accesses. Thus, there is a strong need for tools to assist application developers develop codes that exhibit a high fraction of coalesced accesses. Unless the programmer is able to detect the problem, other optimization tools depending on programmer input are of little help. If uncoalesced access is detected, the programmer can then seek to transform the code.

Existing static transformation approaches to enhance coalesced access are only applicable to restricted classes of programs. In this thesis, we propose a new method to overcome the limitations of purely static approaches by combining the benefits of dynamic analysis with static transformations. When dynamic analysis on traces generated from the program detects uncoalesced accesses, some recommendations are made depending on the overall memory access pattern. In many cases, static transformations can then be applied to convert the uncoalesced access to coalesced access. We provide a static transformation framework that transforms CUDA/PTX code to improve data coalescing.

This dissertation makes the following contributions:

- It is the first work, to the best of our knowledge, to develop a dynamic analysis approach that seeks to characterize the inherent data locality characteristics of algorithms.
- It develops effective algorithms to perform convex partitioning of CDAGs to enhance data locality. While convex partitioning of DAGs has previously been used

for estimating parallel speedup, to our knowledge this is the first effort to use it for characterizing data locality potential.

- It demonstrates the potential of the approach to identify opportunities for enhancement of data locality in existing implementations of computations. Thus, an analysis tool based on this approach to data locality characterization can be valuable to: (i) application developers, for comparing alternate algorithms and identifying parts of existing code that may have significant potential for data locality enhancement, and (ii) compiler writers, for assessing the effectiveness of a compiler's program optimization module in enhancing data locality.

- It demonstrates, through case studies, the use of the new dynamic analysis approach in identifying opportunities for data locality optimization that are beyond the scope of the current state-of-the-art optimizing compilers. Based on the insights provided by the tool, such as the existence of 3D tiles for Floyd-Warshall, we focused our design effort on high-potential codes, leading to significant reduction in data movement.

- It develops a dynamic analysis tool for analyzing arbitrary PTX codes for identifying loops with uncoalesced accesses in GPU kernels, and categorization of uncoalesced accesses to different groups, along with suggestions for potential improvement strategies.

- It constructs a static transformation framework that implements a remapping of work among threads to optimize CUDA/PTX codes exhibiting uncoalesced global memory access.

- It demonstrates the effectiveness of the tool by characterizing well-known GPU benchmark suites and transforming a number of them, including irregular applications and generating transformed versions ensuring coalesced access and improved performance.

The rest of the dissertation is organized as follows:

Chapter 2 provides pertinent background information on reuse distance analysis, dynamic analysis and GPU architecture.

Chapter 3 provides a detailed description of the tool for characterization of CPU applications.

Chapter 4 presents an enhancement of the characterization tool that uses loop induction variable information.

Chapter 5 presents the approach to characterize and enhance data coalescing on GPUs.

Chapter 6 concludes the dissertation.

CHAPTER 2

Background

2.1 Reuse Distance Analysis

Reuse distance analysis is a widely used metric that models data locality [29, 65]. The reuse distance of a reference in a memory address trace is defined as the number of distinct memory references between two successive references to the same location. Since its introduction in 1970 by Mattson et al. [65], reuse distance analysis has found numerous applications in performance analysis and optimization, such as cache miss rate prediction [46, 64, 111], program phase detection [87], data layout optimization [112], virtual memory management [19] and I/O performance optimization [45].

Time	0	1	2	3	4	5	6	7	8	9
Data Ref.	d	a	c	b	c	c	e	b	a	d
Reuse Dist.	∞	∞	∞	∞	1	0	∞	2	3	4

Figure 2.1: Example data reference trace

An example data reference trace of length $N = 10$ is shown in Fig. 2.1. It contains references to $M = 5$ distinct data addresses $\{a, b, c, d, e\}$. As shown in the figure, each

reference to an address in the trace is associated with a reuse distance. The first time an address is referenced, its reuse distance is ∞. For all later references to an address, the reuse distance is the number of distinct intervening addresses referenced. In the figure, address c is referenced three times. Since b is referenced in between the first and second references to c, the latter has a reuse distance of 1. Since the second and third references to c are consecutive, without any other distinct intervening references to any other addresses, the last access to c has a reuse distance of 0.

A significant advantage of reuse distance analysis (RDA) is that a single analysis of the address trace of a code's execution can be used to estimate data locality characteristics as a function of the cache size. In contrast, cache simulation to determine the number of hits/misses would have to be repeated for each cache size of interest. Although real caches have non-unit line size and finite associativity, the data transfer volume estimated from the cache miss count via reuse distance analysis can serve as a valuable estimate for any real cache. Efficient parallel algorithms [25, 68] have also been developed for reuse distance analysis.

Given an execution trace, a reuse distance histogram for that sequence is obtained as follows. For each memory reference, M in the trace, its reuse distance is the number of distinct addresses in the trace after the most recent access to M (the reuse distance is considered to be infinity if there was no previous reference in the trace to M). The number of references in the trace with reuse distance of 0, 1, 2, ..., are counted to form the reuse distance histogram. A cumulative reuse distance histogram plots, as a function of k, the number of references in the trace that have reuse distance less than or equal to k. The cumulative reuse distance histogram directly provides the number of cache hits for a fully associative cache of capacity C with a LRU (Least Recently Used) replacement policy,

since the data accessed by any reference with reuse distance less than or equal to C would result in a cache hit.

2.1.1 Example

We use a simple example to illustrate both the benefits as well as a significant limitation of standard RDA. Fig. 2.2 shows code for a simple Seidel-like spatial sweep, with a default implementation and a fully equivalent tiled variant, where the computation is executed in blocks.

```
1 for(i = 1; i < N-1; i++)
2   for(j = 1; j < N-1; j++)
3     A[i][j] = A[i-1][j] + A[i][j-1];
```

(a) Untiled

```
1 /* B -> Tile size */
2 for(it = 1; it < N-1; it += B)
3   for(jt = 1; jt < N-1; jt += B)
4     for(i = it; i < min(it+B, N-1); i++)
5       for(j = jt; j < min(jt+B, N-1); j++)
6         A[i][j] = A[i-1][j] + A[i][j-1];
```

(b) Tiled

Figure 2.2: Example: Single-sweep two-point Gauss-Seidel code, (a) Untiled and (b) Tiled

Fig. 2.3 displays the cumulative reuse distance histogram for both versions. As explained above, it can be interpreted as the number of data cache hits (y axis) as a function of the cache size (x axis). The same data is depicted in Fig. 2.4, showing the number of cache misses (by subtracting the number of hits from the total number of references). For

the untiled form of the code, a cache with capacity less than 400 words (3200 bytes, with 8 bytes per element) will be too small to effectively exploit reuse. The reuse distance profile for the tiled code is quite different, suggesting that effective exploitation of reuse is feasible with a smaller cache of capacity of 50 words (400 bytes). This example illustrates the benefits of RDA: i) For a given code, it provides insights into the impact of cache capacity on the expected effectiveness of data locality exploitation, and ii) Given two known alternative implementations for a computation, it enables a comparative assessment of the codes with respect to data locality.

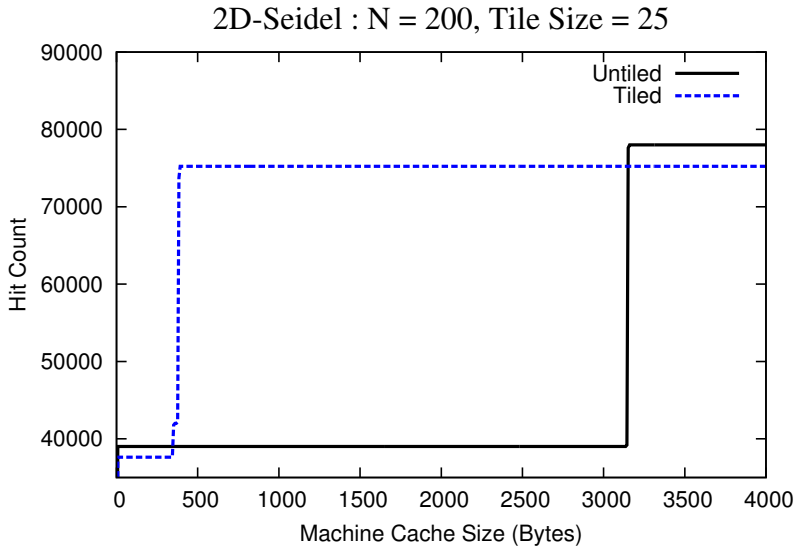


Figure 2.3: Reuse distance profile:cache hit rate

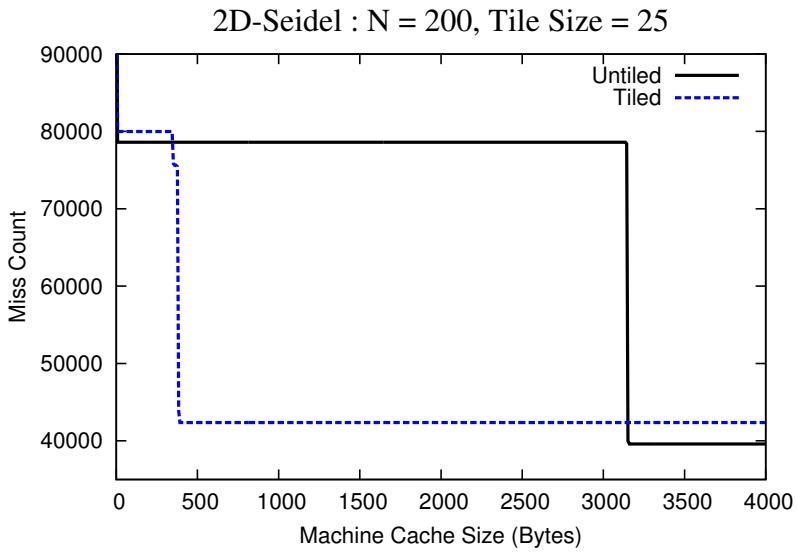


Figure 2.4: Reuse distance profile:cache miss rate

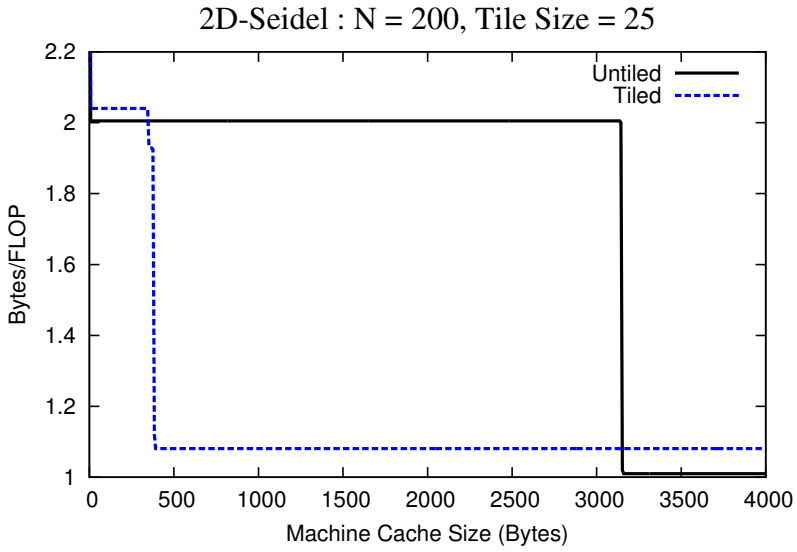


Figure 2.5: Reuse distance profile:memory bandwidth demand

2.1.2 Limitations of reuse distance analysis

The Seidel example also illustrates a fundamental shortcoming of RDA that we address through the work presented in this article: Given an execution trace for a code, RDA only provides a data locality characterization for one *particular execution order* of the constituent operations, and provides no insights on whether significant improvements may be possible via dependence preserving reordering of execution of the operations. The tiled and untiled variants in Fig. 2.2 represent equivalent computations, with the only difference being the relative order of execution of exactly the same set of primitive arithmetic operations on exactly the same sets of array operands. Although state-of-the-art static compiler transformation techniques (e.g., using polyhedral loop transformations) can transform the untiled code in Fig. 2.2(a) to a tiled form of Fig. 2.2(b), many codes exist (as illustrated through case studies later in this article), where data locality characteristics can be improved, but are beyond the scope of the most advanced compilers today. The main question that RDA does not answer is whether the poor reuse distance profile for the code due to a sub-optimal execution order of the operations (e.g., untiled code version of a tileable algorithm) or is it more fundamental property of the computation that remains relatively unchangeable through any transformations that change the execution order of the operations? This is the question our work seeks to assist in answering. By analyzing the execution trace of a given code, forming a dynamic data dependence graph, and reordering the operations by forming convex partitions, the potential for improving the reuse distance profile is evaluated. The change to the reuse distance profile after the dynamic analysis and reordering, rather than the shape of the initial reuse distance profile of a code, provides guidance on the potential for further improvement.

Fig. 2.5 presents the information in Fig. 2.4 in terms of memory bandwidth required per operation. It translates the cache miss count into the bandwidth demand on the memory system in bytes/second per floating-point operation. For this code, we have one floating point operation per two memory references. With a cache miss rate m, assuming double-precision (8 bytes per word), the demand on the main-memory bandwidth would be $16 * m$ bytes per Flop. If this ratio is significantly higher than the ratio of a system's main memory bandwidth (in Gbytes/sec) to its peak performance (in GFlops), the locality analysis indicates that achieving high performance will be critically dependent on effective data locality optimization. For example, on most current systems, the performance of this computation will be severely constrained by main memory bandwidth for problem sizes that are too large to fit in cache.

A point of note is that while the estimated cache hit rates/counts by using RDA can deviate quite significantly from actually measured cache hit rates/counts on real systems (due to a number of aspects of real caches, such as non-unit line size, finite associativity, pre-fetching, etc.), the bytes/flop metric serves as a good indicator of the bandwidth requirement for real caches. This is because pre-fetching and non-unit line sizes only affect the latency and number of main memory accesses and not the minimum volume of data that must be moved from memory to cache. Finite cache associativity could cause an increase in the number of misses compared to a fully associative cache, but not a decrease. All the experimental results presented later in the article depict estimates on the bytes/flop bandwidth demanded by a code. Thus, despite the fact that RDA essentially models an idealized fully associative cache, the data represents useful estimates on the bandwidth demand for any real cache.

A key observation is in the cache size required to achieve full reuse: for the untiled version, cache misses are lowest for a cache size of 400 elements or more. For the tiled version, a cache size of about 10 elements is enough to achieve the same number of misses, a reduction of 40x in cache size, *by considering an alternative execution order for the program operations*. Therefore, depending on whether the input application on which RDA is performed has been optimized or not, one may get totally different RDA profiles. Therefore RDA alone is not enough to characterize what could be achieved, in terms of data locality, through program/software modification.

Another feature expected from a data locality analysis tool is the ability to drive the programmer's implementation efforts towards code regions with high locality *potential*. If the reuse distance profile shows low reuse (for instance, a large cache size is required to achieve 80% or more cache hits), then efforts should be put to improve the implementation so as to try to reduce the reuse distance. This is clear for instance with the example of Fig. 2.2 where a simple tiled variant leads to a 40x reduction in the cache size needed.

2.2 Dynamic Analysis

Previous efforts that use dynamic analysis focus on finding potential parallelism. Kumar's [52] approach performs time-stamp based analysis of instrumented statement level execution of the sequential program, using shadow variables to maintain last modification times for each variable. Each run-time instance of a statement is associated with a timestamp that is one greater than the maximum among the last-modify times of all input operands of the statement. A histogram of the number of operations at each time value provides a fine-grained parallelism profile of the computation, and the maximal timestamp

represents the critical path length for the entire computation. Other prior efforts with a similar overall approach include [7, 52, 53, 63, 67, 78, 81, 88, 89, 100].

The analysis of potential parallelism from [52] computes a timestamp for each DDG node, representing the earliest time this node could be executed. The largest timestamp, compared to the number of nodes, provides a characterization of the inherent fine-grain parallelism in the program; this largest timestamp gives the length of the *critical path* in the dynamic dependence graph. In essence, the computed timestamps implicitly model the best parallel execution of all possible dependence-preserving reordering of the operations performed by the program. All nodes with the same timestamp are independent and can be executed in parallel.

In contrast to the above fine-grained approach, an alternate technique developed by Larus [55] performed analysis of loop-level parallelism at different levels of nested loops. Loop-level parallelism is measured by forcing a sequential order of execution of statements within each iteration of a loop being characterized, so that the only available concurrency is across different iterations of that loop.

A related technique is applied in the context of speculative parallelization of loops, where dynamic dependences across loop iterations are tracked [82]. A few recent approaches of similar nature include [18, 73, 90, 91, 105, 110].

Our approach uses the idea of shadow variables to create the dynamic dependence graph but does not focus on loop level parallelism. Instead, a dynamic dependence graph is generated with all necessary flow dependences, where each node carries its loop induction variables information. The graph is later processed as a whole, looking for a *valid convex partitioning*. The details of the method are described in later sections.

2.3 DDG Generation

Generating a dynamic data-dependence graph (DDG) requires an execution trace of the program (or a contiguous subtrace), containing run-time instances of static instructions, including any relevant run-time data such as memory addresses for loads/stores, procedure calls, etc. Our implementation uses LLVM [96] to instrument arbitrary C/C++/Fortran code. The Clang [94] front-end is used to compile C/C++ code into LLVM IR, and the DragonEgg [95] GCC plugin is used to compile Fortran 77/90 code into LLVMIR. The LLVMIR is instrumented to generate a run-time trace to disk, and the instrumented code is compiled to native code. Once an execution trace is available, the construction of the DDG creates a graph node for each dynamic instruction instance. Edges are created between pairs of dependent nodes (i.e., one instruction instance consumes a value produced by the other). In our implementation each graph node represents a dynamic instance of an LLVM IR instruction, and dependences are tracked through memory and LLVM virtual registers. To construct the graph edges, information is maintained for each memory/register about the graph node that performed the last write to each location. Note that the graph represents only flow dependences. Anti-dependences and output dependences are not considered, since they do not represent essential constraints of the computation, and could potentially be eliminated via transformations such as scalar/array expansion. Control dependences are also not considered, since our goal is to focus on the data flow and the optimization potential implied by it. It is straightforward to augment the DDG with additional categories of dependences, without having to modify in any way the subsequent graph analyses.

CHAPTER 3

Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential

3.1 Introduction

As data access time has become the bottleneck in program performance, we need to re-think program performance in terms of their data locality. To assess how well written an algorithm is, we need to be able to characterize them by their data reuse potential. A program that has better reuse potential is more likely to perform better in terms of memory access time compared to the one that has less data reuse potential. Sometimes we need to transform the program from its original form to measure its actual data locality potential. Therefore, we need a tool that can characterize applications for us in terms of their inherent data reuse potential, considering possible program transformation.

Advances in technology over the last few decades have yielded significantly different rates of improvement in the computational performance of processors relative to the speed of memory access. The Intel 80286 processor introduced in 1982 had an operation execution latency of 320 ns and a main memory access time of 225 ns [41]. The recent Intel Core i7 processor has an operation latency of 4ns and a memory latency of 37 ns, illustrating an order of magnitude shift in the ratio of operation latency to memory access latency.

Since processors use parallelism and pipelining in execution of operations and for memory access, it is instructive to also examine the trends in the peak execution throughput and memory bandwidth for these two processors: 2 MIPS and 13 MBytes/sec for the 80286 versus 50,000 MIPS and 16,000 MBytes/sec for the Core i7. The ratio of peak computational rate to peak memory access bandwidth has also changed by more than an order of magnitude.

Because of the significant mismatch between computational latency and throughput when compared to main memory latency and bandwidth, the use of hierarchical memory systems and the exploitation of significant data reuse in the higher (i.e., faster) levels of the memory hierarchy is critical for high performance. Techniques such as pre-fetching and overlap of computation with communication can be used to mitigate the impact of high memory access latency on performance, but the mismatch between maximum computational rate and peak memory bandwidth is much more fundamental; *the only solution is to limit the volume of data movement to/from memory by enhancing data reuse in registers and higher levels of the cache.*

A significant number of research efforts have focused on improving data locality, by developing new algorithms such as the so called *communication avoiding* algorithms [8, 9, 28] as well as automated compiler transformation techniques [16, 44, 48, 101]. With future systems, the cost of data movement through the memory hierarchy is expected to become even more dominant relative to the cost of performing arithmetic operations [14, 34, 86], both in terms of throughput and energy. Optimizing data access costs will become ever more critical in the coming years. Given the crucial importance of optimizing data access costs in systems with hierarchical memory, it is of great interest to develop tools and techniques to assess the inherent data locality characteristics of different parts of a

computation, and the potential for data locality enhancement via dependence preserving transformations.

While reuse distance analysis provides a useful characterization of data locality for a given execution trace, it fails to provide any information on the potential for improvement in data locality that may be feasible through valid reordering of the operations in the execution trace. In particular, given only the reuse distance profile for the address trace generated by execution of some code, it is not possible to determine whether the observed locality characteristics reveal fundamental inherent limitations of an algorithm, or are merely the consequence of a sub-optimal implementation choice.

In this chapter, we present a tool that characterizes application with a view to measure their data locality potential. We use dynamic analysis approach where we get profile information from an actual execution of the program. The profile/trace contains information about the actual data accessed in the program. We then build a dependence graph and apply a convex partitioning heuristic to obtain tiling of the computation space. The tiling is dependence preserving and therefore equivalent to a valid program transformation. We then compare the reuse distance profile of different algorithms or different versions of the same program to assess their inherent data locality potential. Given an execution trace from a sequential program, we seek in addition to estimate the inherent data locality characteristics of an algorithm and thereby determine if there exists potential for enhancement of data locality through execution reordering. It is the first work, to the best of our knowledge, to use a dynamic analysis approach to solve the problem of characterization of the inherent data locality characteristics of algorithms. We have proposed and developed two alternate convex partitioning heuristics of the dependence graph and demonstrate the potential of the approach to identify opportunities for enhancement of data locality in existing applications.

The analysis tool based on this approach to data locality characterization can be used by application developers, for comparing alternate algorithms and identifying parts of existing code that may have significant potential for data locality enhancement. Compiler writers can also test the effectiveness of a compiler's program optimization module in enhancing data locality. Based on the insights provided by our end-to-end tool, such as the existence of 3D tiles for Floyd-Warshall, we focused our design effort on high-potential codes, leading to significant reduction in data movements.

Previous techniques depending dynamic analysis looks for parallel loops [7, 18, 52, 53, 55, 63, 67, 73, 78, 81, 88–90, 100, 105, 110]. To the best of our knowledge, none attempted to apply the technique to solve the problem of loop tiling.

The rest of this chapter is organized as follows. Section 3.2 presents background on reuse distance analysis, and a high-level overview of the proposed approach for locality characterization. The algorithmic details of the approach to convex partitioning of CDAGs are provided in Section 3.3. Section 3.4 presents experimental results. Related work is discussed in Section 3.5, followed by concluding remarks in Sections 3.6 and 3.7.

3.2 Background & Overview of Approach

Reuse distance analysis (RDA) is particularly meant to characterize, as a function of the cache size, what is the quantity of memory references that can be cache hits. More precisely, for a *given implementation*, reuse distance analysis can drive the hardware design exploration (esp. for the cache size parameter) to best execute this particular computation. But a severe limitation of RDA is that it ignores any kind of software design exploration that could be used to improve the data locality of a computation. For instance, the program's

computations may be reordered so as to minimize the reuse distance, therefore leading to possibly vastly different RDA on this transformed program. It could lead to significantly smaller cache size requirement to exploit the program's data reuse.

3.2.1 Benefits of the proposed dynamic analysis

Using results from two case studies presented later in the chapter, we illustrate the benefits of the approach we develop. Fig. 3.1 shows the original reuse distance profiles as well as the profiles after dynamic analysis and convex partitioning, for two benchmarks: Householder transformation on the left, and Floyd-Warshall all-pairs shortest path on the right.

As seen in the upper plot in Fig. 3.1, with the Householder code, no appreciable change to the reuse distance profile results from the attempted reordering after dynamic analysis. In contrast, the lower plot in Fig. 3.1 shows a significantly improved reuse distance profile for the Floyd-Warshall code, after dynamic analysis and reordering of operations. This suggests potential for enhanced data locality via suitable code transformations. As explained later in the experimental results section, manual examination of the convex partitions provided insights into how the code could be transformed into an equivalent form that in turn could be tiled by an optimizing compiler. The reuse distance profile of that tiled version is shown as a third curve in the lower plot in Fig. 3.1, showing much better reuse than the original code. The actual performance of the modified code was also significantly higher than the original code. To the best of our knowledge, this is the first 3D tiled implementation of the Floyd Warshall algorithm (other blocked versions have been previously developed [75, 98], but have required domain-specific reasoning for semantic changes to form equivalent algorithms that generate different intermediate values but the same final

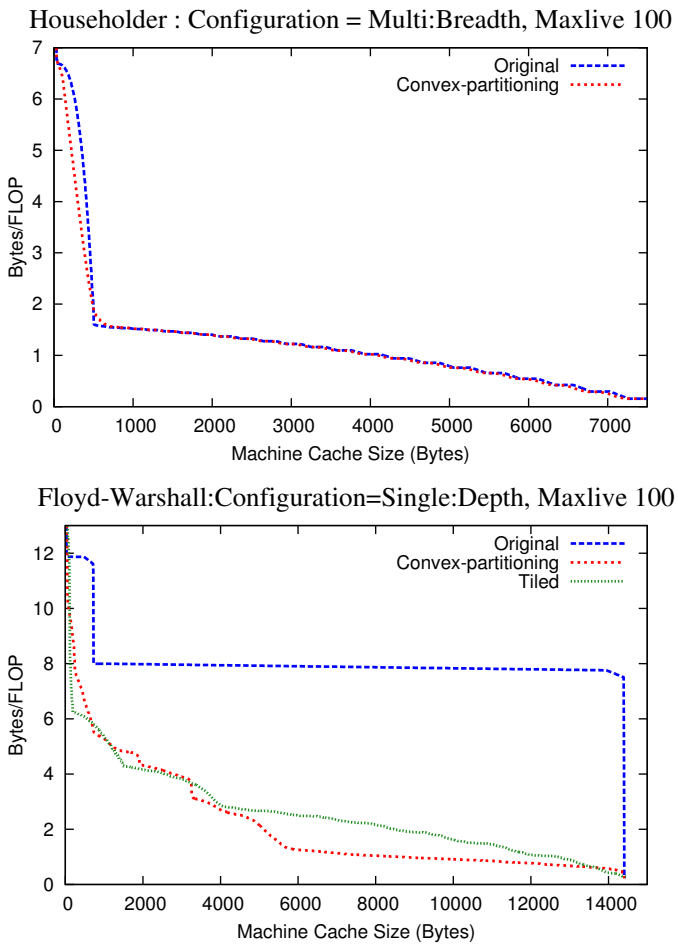


Figure 3.1: Reuse distance analysis for Householder and Floyd-Warshall

results as the standard algorithm).

3.2.2 Overview of Approach

The new dynamic analysis approach we propose attempts to characterize the inherent data locality properties of a given (sequential) computation, and to assess the potential for enhancing data locality via change of execution ordering. To achieve this goal, we

proceed in two stages. First, a new ordering of the program's operations is computed, by using graph algorithms (that is, convex partitioning techniques) operating on the expanded computation graph. Then, standard reuse distance analysis is performed on the reordered set of operations. We note that our analysis does not directly provide an optimized program. Implementing the (possibly very complex) schedule found through our graph analysis is impractical. Instead, our analysis highlights gaps between the reuse distance profile of a current implementation and existing data locality potential: the task of devising a better implementation is left to the user or compiler writer. In Sec. 3.4, we show the benefits of the approach on several benchmarks.

To implement our new dynamic analysis, we first analyze the data accesses and dependences between the primitive operations in a sequential execution trace of the program to extract a more abstract model of the computation: a computational directed acyclic graph (CDAG), where operations are represented as vertices and the flow of values between operations as edges. This is defined as follows.

Definition 1 (CDAG [15]) *A computation directed acyclic graph (CDAG) is a 4-tuple* $C = (I, V, E, O)$ *of finite sets such that: (1)* $I \cap V = \emptyset$*; (2)* $E \subseteq (I \cup V) \times V$ *is the set of arcs; (3)* $G = (I \cup V, E)$ *is a directed acyclic graph with no isolated vertices; (4)* I *is called the input set; (5)* V *is called the operation set and all its vertices have one or two incoming arcs; (6)* $O \subseteq (I \cup V)$ *is called the output set.*

Fig. 3.2 shows the CDAG corresponding to the code in Fig. 2.2 for N=6 — both versions have identical CDAGs since they perform exactly the same set of floating-point computations, with the same inter-instance data dependences, even though the total order of execution of the statement instances is different. The loop body performs only one addition and is executed a total of 16 times, so the CDAG has 16 computation nodes (white circles).

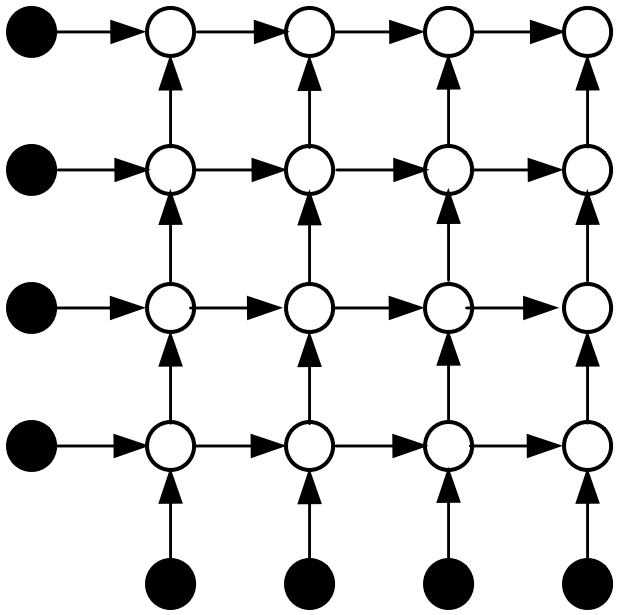


Figure 3.2: CDAG for Gauss-Seidel code in Fig. 2.2. Input vertices are shown in black, other vertices represent operations performed.

Although a CDAG is derived from analysis of dependences between instances of statements executed by a sequential program, it abstracts away that sequential schedule of operations and only imposes an essential partial order captured by the data dependences between the operation instances. Control dependences in the computation need not be represented since the goal is to capture the inherent data locality characteristics based on the set of operations that were actually execution in the program.

They key idea behind the work presented in this chapter is to perform analysis on the CDAG of a computation, attempting to find a different order of execution of the operations that can improve the reuse-distance profile compared to that of the given program's sequential execution trace. If this analysis reveals a significantly improved reuse distance profile,

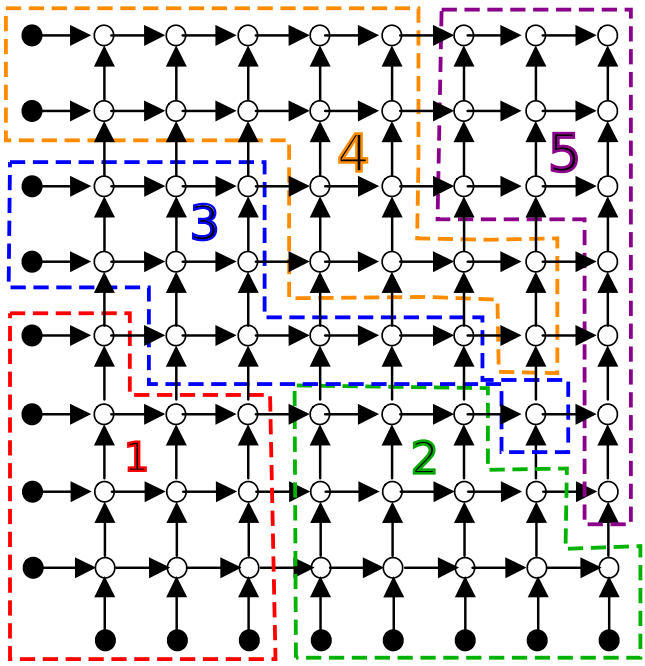


Figure 3.3: Convex-partition of the CDAG for the code in Fig. 2.2 for $N = 10$.

it suggests that suitable source code transformations have the potential to enhance data locality. On the other hand, if the analysis is unable to improve the reuse-distance profile of the code, it is likely that it is already as well optimized for data locality as possible. The dynamic analysis involves the following steps:

1. Generate a sequential execution trace of a program.

2. Run a reuse-distance analysis of the original trace.

3. Form a CDAG from the execution trace.

4. Perform a multi-level convex partitioning of the CDAG, which is then used to change the schedule of operations of the CDAG from the original order in the given input code. A convex partitioning of a CDAG is analogous to tiling the iteration space of

a regular nested loop computation. Multi-level convex partitioning is analogous to multi-level cache-oblivious blocking.

5. Perform standard reuse distance analysis of the reordered trace after multi-level convex partitioning.

Finally, Fig. 3.3 shows the convex partitioning of the CDAG corresponding to the code in Fig. 2.2.

After such a partitioning, the execution order of the vertices is reordered so that the convex partitions are executed in some valid order (corresponding to a topological sort of a coarse-grained inter-partition dependence graph), with the vertices within a partition being executed in the same relative order as the original sequential execution. Details are presented in the next section.

3.3 Convex Partitioning of CDAG

In this section, we provide details on our algorithm for convex partitioning of CDAGs, which is at the heart of our proposed dynamic analysis. In the case of loops, numerous efforts have attempted to optimize data locality by applying loop transformations, in particular involving loop tiling and loop fusion [16, 44, 48, 101]. Tiling for locality attempts to group points in an iteration space of a loop into smaller blocks (tiles) allowing reuse (thereby reducing reuse distance) in multiple directions when the block fits in a faster memory (registers, L1, or L2 cache). Forming a valid tiling for a loop requires that each tile can be executed atomically, i.e., each tile can start after performing required synchronizations for the data it needs, then execute all the iterations in the tile without requiring intervening synchronization. This means that there are no cyclic data dependencies between any two

tiles. *Our goal in this work is to extend this notion of "tiling" to arbitrary CDAGs that represent a computation: we form valid partitioning of CDAGs into components such that the components can be scheduled and executed, with all vertices in a component being executed "atomically," i.e., without being interleaved with vertices in any other components.* For this, we rely on the notion of *convex partitioning* of CDAGs, which is the generalization of loop tiling to graphs.

3.3.1 Definitions

We first define what is a convex component, that is a tile in a graph.

Definition 2 (Convex component) *Given a CDAG G, a convex component V_i in G is defined as a subset of the vertices of G such that, for any pair of vertices u and v in V_i, if there are paths between u and v in G, then every vertex on every path between u and v also belongs to V_i.*

A convex partition of a graph G is obtained by assigning each vertex of G to a single convex component. Since there are no cycles among convex components, the graph in which nodes are convex components and edges define dependences among them, is acyclic. Therefore, we can execute the convex components using any topologically sorted order. Executing all the convex components results in executing the full computation.

A *convex partition* of a graph $G = (V, E)$ is a collection of convex components $\{V_1, \ldots, V_k\}$ of G such that $\bigcup V_{i=1}^{k} = V$ and for any i, j s.t. $1 \leq i, j \leq k$ and $i \neq j$, $V_i \cap V_j = \emptyset$. We remark that tiling of iterations spaces of loops results in convex partitions.

The *component graph* $\mathcal{C} = (\mathcal{V}_\mathcal{C}, \mathcal{E}_\mathcal{C})$ is defined as a graph whose vertices $\mathcal{V}_\mathcal{C}$ represent the convex components, i.e., $\mathcal{V}_\mathcal{C} = \{V_1, \cdots, V_k\}$. Given two distinct components V_i and

V_j, there is an edge in $\mathcal{C}$ from V_i to V_j if and only if there is an edge (a, b) in the CDAG, where $a \in V_i$ and $b \in V_j$.

For a given schedule of execution of the vertices of convex component V_i, we define *maxlive* to be the maximum number of simultaneously live nodes for this schedule. A node can be in one of the following states throughout its life:

Initial state: at the beginning no node is live. They are in the initial state.

Birth: any node is considered live right after it is fired (executed). Whenever we pick a node and include it in a partition, then the node is born and live.

Resurrection: if not part of the convex component, it is also considered as live when used by another node of the component (predecessor of a fired node belonging to the component). During convex partitioning, a node might be already included in a partition but has unprocessed successors. Whenever one of its successor is born or picked from the list to be included in a partition, the predecessor node also becomes live and contributes to the maxlive computation. We denote this process as resurrection. A node may be resurrected whenever one of its successors is getting fired.

Live: a born or resurrected node stays alive until it dies, which happens if all its successor nodes have executed (are part of a partition).

Death: a node dies right after its last successor is fired.

Our goal is to form convex components along with a scheduling such that the *maxlive* of each component does not exceed the local memory capacity. We consider the nodes we add to the component (just fired and alive), and their predecessors (resurrected) in computing the *maxlive*.

3.3.2 Forming Convex Partitions

We show in Algorithm 3.1 our technique to build convex partitions from an arbitrary CDAG. It implements a convex-component growing heuristic that successively adds ready vertices into the component until a capacity constraint is exceeded. The key requirement in adding a new vertex to a convex component is that if any path to that vertex exists from a vertex in the component, then all vertices in that path must also be included. We avoid an expensive search for such paths by constraining the added vertices to be those that already have all their predecessors in the current (or previously formed) convex component.

Algorithm 3.1 GenerateConvexComponents(*G, C, CF*)

Input : G : CDAG; C : Cache Size;
CF: Cost function to decide next best node
InOut: P : Partition containing convex components

```
begin
    P ⟵ ∅
    R ⟵ getTheInitialReadyNodes(G)
    while R ≠ ∅ do
        n ⟵ selectReadyNode(R)
        cc ⟵ ∅
        while R ≠ ∅ ∧ updateLiveSet(cc, n, C) do
            cc ⟵ cc ∪ {n}
            R ⟵ R − {n}
            UpdateListOfReadyNodes(R, n)
            priority ⟵ CF()
            n ⟵ selectBestNode(R, cc, priority, n)
        P ⟵ P ∪ {cc}
```

The partitioning heuristic generates a valid schedule as it proceeds. At the beginning, all input vertices to the CDAG are placed in a ready list R. A vertex is said to be *ready* if all its predecessors (if any) have already executed, i.e., have been assigned to some

convex component. A new convex component cc is started by adding a ready vertex to it (the function `selectReadyNode`(R) simply picks up one element of R) and it grows by successively adding more ready nodes to it (`selectBestNode`(*R, cc, priority, n*) selects one of the ready nodes, as shown in Algorithm 3.4 – the criterion is described later). Suppose a vertex n is just added to a component cc. As a result, zero or more of the successors of n in G may become ready: a successor s of n becomes ready if the last predecessor needed to execute s is n. The addition of newly readied vertices to the ready list is done by the function `updateListOfReadyNodes`(*R, n*), as shown in Algorithm 3.2. In this function, the test that checks if s has unprocessed predecessors is implemented using a counter that is updated whenever a node is processed.

Algorithm 3.2 UpdateListOfReadyNodes(*R, n*)

Input : n: Latest processed node
InOut: R: List of ready nodes
begin
  for $s \in$ `successors(`n`)` **do**
    if *s has no more unprocessed predecessors* **then**
      $R \longleftarrow R \cup \{s\}$

Before adding a node to cc, the set *cc.liveset*, the liveout set of cc, is updated through the call to `updateLiveSet`*(p, n, C)*, as shown in Algorithm 3.3. `updateLiveSet` exactly implements our definition of liveness previously described: *(birth)* if n has some successors it is added to the liveset of cc; *(resurrect)* its predecessor nodes that still have unprocessed successors are added to the liveset (if not already in it); *(die)* predecessor nodes for which n is the last unprocessed successor are removed from the liveset.

Algorithm 3.3 updateLiveSet(*p, n, C*)

Input : n : New node added in the partition p
C : Cache size
InOut : $p.liveset$: Live set of p
Output: true if $|p.liveset| \leq C$, false otherwise
begin
  $lset \longleftarrow p.liveset$
  if n *has unprocessed successors* **then**
    $p.liveset \longleftarrow p.liveset \cup \{n\}$
  for $n' \in$ `predecessors(n)` **do**
    if n' *has unprocessed successors* **then**
      $p.liveset \longleftarrow p.liveset \cup \{n'\}$
    else if $n' \in p.liveset$ **then**
      $p.liveset \longleftarrow p.liveset - \{n'\}$
  if $|p.liveset| > C$ **then**
    $p.liveset \longleftarrow lset$ **return** false
  return true

3.3.3 CDAG Traversal: Breadth-first Versus Depth-first

The heuristic `selectBestNode` to select the next processed node within the ready list uses two affinity notions: a node is a ready-successor of cc (thus element of the *cc.readySuccessors* list) if it is a ready successor of some nodes of cc; a node is a ready-neighbor of cc (thus element of *cc.readyNeighbors* list) if it has a successor node that is also a successor of some node in cc. We note that those two lists can overlap. The growing strategy picks up ready nodes, using a first-in first-out policy, from one of those two lists. In practice, we observe that favoring nodes of the ready-successor list would favor growing depth-first in the CDAG, while favoring nodes that belongs to the ready-neighbor list would favor a breadth-first traversal.

The heuristic uses a combination of growing alternately in these two different directions till the $Maxlive$ of the component exceeds the capacity of the cache. The priority is

controlled by a number that represents the ratio of selected ready-neighbors over selected ready-successors. If the ratio is larger than 1, we refer to it as *breadth-priority*; if lower than 1, we refer to it as *depth-priority*, otherwise it is referred to as *equal-priority*. Function $neighbor(n)$ will return the list of all nodes (excluding n) that have a common successor with n.

Algorithm 3.4 selectBestNode(*R, cc, priority, n*)

Input : R: List of ready nodes
cc: current convex component
$priority$: Give priority to neighbor or successor
n: Latest node added in the partition

InOut : *cc.readyNeighbors* : Ready neighbors of current growing partition nodes
cc.readySuccessors : Ready successors of current growing partition nodes

Output: $next$: Next node to add in the partition

begin
  for $a \in \texttt{neighbors}(n) \cap R -$ *cc.readyNeighbors* **do**
    cc.readyNeighbors.`enqueue(`a`)`
  for $a \in \texttt{successors}(n) \cap R -$ *cc.readySuccessors* **do**
    cc.readySuccessors.`enqueue(`a`)`
  cc.readyNeighbors $\longleftarrow$ *cc.readyNeighbors* $- n$
  cc.readySuccessors $\longleftarrow$ *cc.readySuccessors* $- n$
  if *cc.takenNeighbors* $<$ *cc.takenSuccessors* $\times$ *priority*
  $\wedge$ *cc.readyNeighbors* $\neq \emptyset$ **then**
    next $\longleftarrow$ `dequeue(`*cc.readyNeighbors*`)`
    cc.takenNeighbors $\longleftarrow$ *cc.takenNeighbors* $+ 1$
  else if *cc.readySuccessors* $\neq \emptyset$ **then**
    next $\longleftarrow$ `dequeue(`*cc.readySuccessors*`)`
    cc.takenSuccessors $\longleftarrow$ *cc.takenSuccessors* $+ 1$
  else
    next $\longleftarrow$ `selectReadyNode(`R`)`
  return *next*

3.3.4 Multi-level Cache-oblivious Partitioning

Here we address the problem of finding a schedule that is cache-size oblivious, and in particular suitable for multi-level memory hierarchy. In order to address this problem, we construct a hierarchical partitioning using the multi-level component growing heuristic shown in Algorithm 3.5. This algorithm combines individual components formed for a cache of size C using the heuristic in Algorithm 3.1 to form components for a cache of size factor * C. In this approach, each component of the partition built at level l of the heuristic is seen as a single node at level $l + 1$. We call these nodes "macro-nodes" as they typically represent sets of nodes in the original CDAG.

This approach could be compared with multi-level tiling for multi-level cache hierarchy, a classical scheme to optimize data locality for multi-level caches. In our multilevel approach, each component is actually made of *Macro-nodes* which correspond to components created by the lower level heuristic. For the first level of this heuristic, a macro-node corresponds to a node in the original CDAG. The heuristic then proceeds with the next level, seeing each component of the partition at the previous level as a macro-node at the current level. The heuristic stops when only one component is generated at the current level, that is, all macro-nodes were successfully added to a single convex component without exceeding the input/output set size constraints.

The number of levels in the multi-level partitioning varies with each CDAG, and is not controlled by the user. When a component cc_0 formed at a lower level is added to the current component cc_1 being formed at a higher level, the *liveset* of cc_1 has to be updated, just as if all the nodes composing cc_0 have been added to cc_1. This leads to the modified version of `updateLiveSet`(*p,n,C*) reported in Algorithm 3.6, where the function `FirstLevelBaseNodes`(*np*) returns the actual CDAG nodes (that we call first level base nodes) the

macro-node *np* is built upon. At the first level `FirstLevelBaseNodes`(*np*) may be understood as returning just *np*.

Algorithm 3.5 MultiLevelPartitioning(*G, C, Priority, factor*)

Input : G : CDAG
C : initial cache Size
Priority : priority to Neighbor or Successor
factor : multiplication factor of Cache size for each level
InOut : $G.M$: memory footprint of the CDAG
Output: P : Final single partition
begin
  $P \longleftarrow$ `GenerateConvexComponents`(*G, C, Priority*)
  while $C < G.M$ **do**
    $G' \longleftarrow$ `formMacroNodeWithEachPartition`(P, G)
    $C \longleftarrow factor * C$
    $P' \longleftarrow$ `GenerateConvexComponents`(G', C, $Priority$)
    $P \longleftarrow P'$
  return P

Algorithm 3.6 updateLiveSet(*p, n, C*)

Input : n : New macro-node added in the partition p
InOut : $p.liveset$: Live set of p
Output: true if $|p.liveset| \leq C$, false otherwise
begin
  $b \longleftarrow$ true
  $plset \longleftarrow p.liveset$ **for** $nb \in$ `FirstLevelBaseNodes`(*n*) **do**
    $b \longleftarrow b \wedge$ `updateLiveSet`(*p, nb, C*)
  if $b =$ *false* **then**
    $p.liveset \longleftarrow plset$
    return false
  return true

3.3.5 Complexity Analysis

Finally, we analyze the computational complexity of the steps in the algorithm.

1. First, the code is instrumented to generate an execution trace of all operations and memory accesses.

2. Next, the trace is analyzed on-line [50] to generate the corresponding CDAG of the execution instance. Let the program size be $|P|$, the size of the trace be $|T|$, and the size of the programs footprint (used memory) be $|M|$. The time complexity (execution) is $O(|T|)$ and space complexity (shadow memory + CDAG) is $O(|M| + |T|)$.

3. Third, this CDAG for which the number of nodes and edges is $O(|T|)$ is used off-line as input to the CDAG partitioning heuristic.

    - Counters for each node are initialized. Time and space complexity are $O(|T|)$.
    - For each node, its set of neighbors is computed. Considering that each node has at most two predecessors (3-address code), the number of neighbors is twice the number of edges of the CDAG, so $O(|T|)$. Computing the set for each node is linear in the size of the result, i.e., $O(|T|)$.
    - When a node is processed, for each of its successors a counter is updated to determine if it is ready to fire. The amortized time complexity is the number of edges, i.e., $O(|T|)$.
    - When a node gets ready, it is pushed into the ready list. The space complexity of this list can be the total number of nodes, i.e., $O(|T|)$ in the worst case.
    - When a node gets ready, or respectively gets processed its set of neighbors is scanned to update the ready-neighbors list (to either push itself or respectively some of its neighbors in it). The amortized time complexity for this is thus the number of edges, i.e., $O(|T|)$.

- Whenever a node gets ready, or respectively gets processed its set of predecessors (respectively successors) is scanned to update the ready-successors list. Amortized time complexity for this is thus the number of edges i.e. $O(|T|)$.
- Prior to actually processing a node, the live-set of the convex component is updated. Each of its predecessors is scanned to add it or remove it from the component's live-set (counters are used to check if it should be added or removed). Amortized complexity is thus the number of edges plus the number of components (for the last node we do not commit, because maxlive may get too large), i.e., $O(|T|)$.

4. Fourth, given a partitioning evaluating the number of cache misses can be done either by over-approximating it adding live-in and live-out sets of each convex partition leading to an $O(|T|)$ (for updating the live-in set of each component during the growing process) or by simulating a LRU cache on the so obtained schedule. LRU cache can be simulated linearly using both a double-linked list and a map.

The overall complexity of the single-level version is thus linear with the size of the trace. The multi-level version is run $\log_{factor}\left(\frac{|M|}{|C|}\right)$ times `GenerateConvexComponents`, thus leading to an overall time complexity of $O(|T|\log(|M|))$.

3.4 Experimental Results

In the following chapter, we evaluate the impact of the various parameters of the convex partitioning heuristics in Sec. 4.2.2, using two well understood benchmarks: matrix multiplication and a 2D Jacobi stencil computation. With these benchmarks, it is well known how the data locality characteristics of the computations can be improved via loop tiling.

The goal therefore is to assess how the reuse distance profiles after dynamic analysis and operation reordering compares with the unoptimized untiled original code as well as the optimized tiled version of code.

In this chapter, the experimental results are organized as follows: Sec. 3.4.1 describes the experimental environment. In Sec. 3.4.2 we detail several case studies where we use dynamic analysis to characterize the locality potential of several benchmarks for which the state-of-the-art optimizing compilers are unable to automatically optimize data locality. We demonstrate the benefits of dynamic analysis in providing insights into the inherent data locality properties of these computations and the potential for data locality enhancement. Finally we discuss in Sec. 3.4.3 the sensitivity of our techniques to varying datasets.

3.4.1 Experimental Setup

The dynamic analysis we have implemented involves three steps. For the CDAG Generation, we use automated LLVM-based instrumentation to generate the sequential execution trace of a program, which is then processed to generate the CDAG. The trace generator was previously developed for performing dynamic analysis to assess vectorization potential in applications [43]. For the convex partitioning of the CDAG, we have implemented the algorithms explained in detail in the previous section. Finally for the reuse distance analysis of the reordered address trace after convex partitioning, it is done using a parallel reuse distance analyzer PARDA [69] that was previously developed.

The total time taken to perform the dynamic analysis is dependent on the input program trace size. In our experiments, computing the trace and performing the full dynamic analysis can range between seconds for benchmarks like Givens, Householder, odd-even sort or Floyd-Warshall to about one hour for SPEC benchmarks such as 420.LBM. For instance,

for Givens rotation (QR decomposition) the trace is built in 4 seconds, the analysis takes another 14 seconds, and computing the reuse distance analysis on the partitioned graph takes well below a second. We note that while CDAGs are often program-dependent, the shape of the CDAG and the associated dynamic analysis reasoning can usually be performed on a smaller problem size: the conclusion about the data locality potential is likely to hold for the same program running on larger datasets. A study of the impact of the sensitivity of our analysis to different datasets is provided in later Sec. 3.4.3. All performance experiments were done on an Intel Core i7 2700K, using a single core.

3.4.2 Case Studies

We next present experimental results from applying dynamic analysis to several benchmarks: the Floyd-Warshall algorithm to find all-pairs shortest path in a graph represented with an adjacency matrix, two QR decomposition methods: the Givens rotation and the Householder transformation, three SPEC benchmarks, a LU decomposition code from the LAPACK package, and an implementation of odd-even sorting using linked list. None of these benchmarks could be fully tiled for enhanced data locality by state-of-the art research compilers (e.g., Pluto [76]) or by production compilers (e.g., Gnu GCC, Intel ICC). For each benchmark, we study the reuse distance profile of the original code and the code after convex partitioning. Where significant potential for data locality enhancement was revealed by the dynamic analysis, we analyzed the code in greater detail. For two of the four benchmarks we manually optimized (namely, Floyd-Warshall and Givens rotation) have static control-flow. Therefore, the performance for these will be only a function of the dataset size and not the content of the dataset. For all optimized benchmarks, we report performance on various dataset sizes (and various datasets when relevant).

Floyd-Warshall

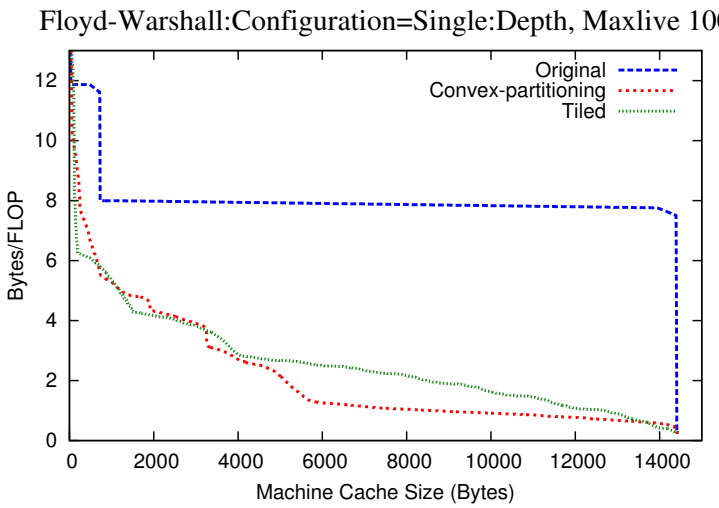
Original program We show in Fig. 3.4 the original input code that we used to implement the Floyd-Warshall algorithm. We refer to this code as "out-of-place" Floyd-Warshall because it uses a temporary array to implement the all-pairs shortest path computation.

```
1 for (k = 0; k < N; k++) {
2   for (i = 0; i < N; i++)
3     for (j = 0; j < N; j++)
4       temp[i][j] = MIN(A[i][j], (A[i][k] + A[k][j]));
5   k++;
6   for (i = 0; i < N; i++)
7     for (j = 0; j < N; j++)
8       A[i][j] = MIN(temp[i][j], (temp[i][k] + temp[k][j
            ]));
9 }
```

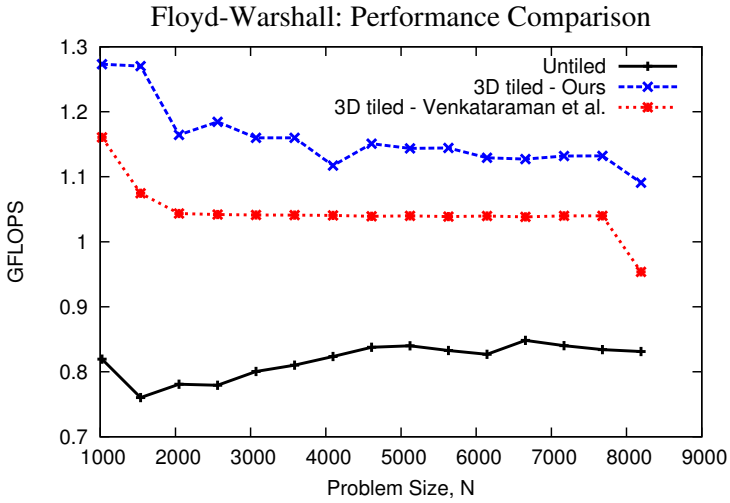
Figure 3.4: Floyd-Warshall all-pairs shortest path

Analysis Fig. 3.5-(a) shows the reuse distance profile of the original code, and the best convex-partitioning found by our heuristics for the Floyd-Warshall algorithm, for a matrix of size 30 by 30. Since the convex-partitioning heuristic shows significantly better reuse distance profile than the original code, there is potential for improvement of data locality through transformations for this program. Studies on the impact of the various heuristic parameters on the quality of the convex partitioning obtained can be found in [30].

Indeed, the Floyd-Warshall algorithm is immediately tilable, along the two loops `i` and `j`. Such tiling can for instance be achieved automatically with polyhedral model based compilers such as Pluto [16]. Since the three loops are not fully permutable, it has been believed that the Floyd-Warshall code cannot be 3D-tiled without transformations using semantic properties of the algorithm to create a modified algorithm (i.e., with a different



(a)



(b)

Figure 3.5: Floyd-Warshall: Analysis and performance improvements due to tiling

CDAG) that provably produces the same final result [75,98]. However, a careful inspection of the convex partitions revealed that *valid 3D tiles can be formed among the operations*

of the standard Floyd-Warshall algorithm. This non-intuitive result comes from the non-rectangular shape of the tiles needed, with varying tile size along the `k` dimension as a function of the value of `k`. This motivated us to look for possible transformations that could enable 3D tiling of the code without any semantic transformations.

Modified implementation The modified implementation we designed is shown in Figure 3.6. It has an identical CDAG, i.e., it is semantically equivalent, to the code in Listing 3.4. To create this version, we fist split the original iteration space into four distinct regions through manual index splitting, followed by tiling of each loop nest.

Performance comparison Fig. 3.5 compares the performance of the tiled version against the original code. From Fig. 3.5(a), we can observe that the tiled code is able to achieve better data locality, that is close to the potential uncovered by the convex partitioning heuristics. Fig. 3.5(b) shows the improvement in actual performance of our tiled code (3D tiled - Ours), due to reduced cache misses. A performance improvement of about 1.6× (sequential execution) is achieved, across a range of problem sizes. Further, to the best of our knowledge, this is the first development of a tiled version of the standard Floyd-Warshall code that preserves the original code's CDAG. We also compared with the performance achieved by the semantically modified 3D-tiled implementation from [98] and found it to have slightly lower performance.

Givens Rotation

Original program Fig. 3.7 shows the original input code for the Givens rotation method used for QR decomposition.

```
1  /*Region 4*/
2  for (k = 0; k < N; k+=B1)
3   for (i = k, iend = N; i < iend; i+=B2)
4    for (j = k, jend = N; j < jend; j+=B3)
5     for(kt = k, ktend = MIN(k+B1,N); kt < ktend; ++kt) {
6      for(it = MAX(i,kt), itend = MIN(i+B2,N); it < itend; ++it)
7       for(jt = MAX(j,kt), jtend = MIN(j+B3,N); jt < jtend; ++jt)
8        temp[it][jt] = MIN(A[it][jt],(A[it][kt] + A[kt][jt]));
9        ++kt;
10      for(it = MAX(i,kt), itend = MIN(i+B2,N); it < itend; ++it)
11       for(jt = MAX(j,kt), jtend = MIN(j+B3,N); jt < jtend; ++jt)
12        A[it][jt]=MIN(temp[it][jt],(temp[it][kt]+temp[kt][jt]));
13    }
14 /*Region 3*/
15 for (k = 0; k < N; k+=B1)
16  for (i = k, iend = N; i < iend; i+=B2)
17   for(j = 0, jend = k+B1; j<jend; j+=B3)
18    for(kt = k, ktend = MIN(k+B1,N); kt < ktend; ++kt) {
19     for(it = MAX(i,kt), itend = MIN(i+B2,N); it < itend; ++it)
20      for(jt = j, jtend = MIN(j+B3,kt); jt < jtend; ++jt)
21       temp[it][jt] = MIN(A[it][jt],(A[it][kt] + A[kt][jt]));
22       ++kt;
23      for(it = MAX(i,kt), itend = MIN(i+B2,N); it < itend; ++it)
24       for(jt = j, jtend = MIN(j+B3,kt); jt < jtend; ++jt)
25        A[it][jt]=MIN(temp[it][jt],(temp[it][kt]+temp[kt][jt]));
26    }
27 /*Region 2*/
28 for(k = 0; k < N; k+=B1)
29  for(i = 0, iend = k+B1; i<iend; i+=B2)
30   for(j = k, jend = N; j<jend; j+=B3)
31    for(kt = k, ktend = MIN(k+B1,N); kt < ktend; ++kt) {
32     for(it = i, itend = MIN(i+B2,kt); it < itend; ++it)
33      for(jt = MAX(j,kt), jtend = MIN(j+B3,N); jt < jtend; ++jt)
34       temp[it][jt] = MIN(A[it][jt], (A[it][kt] + A[kt][jt]));
35       ++kt;
36      for(it = i, itend = MIN(i+B2,kt); it < itend; ++it)
37       for(jt = MAX(j,kt), jtend = MIN(j+B3,N); jt < jtend; ++jt)
38        A[it][jt] = temp[it][jt] + (temp[it][kt] + temp[kt][jt]);
39    }
40 /*Region 1*/
41 for(k = 0; k < N; k+=B1)
42  for(i = 0, iend = k+B1; i<iend; i+=B2)
43   for(j = 0, jend = k+B1; j<jend; j+=B3)
44    for(kt = k, ktend = MIN(k+B1,N); kt < ktend; ++kt)  {
45     for(it = i, itend = MIN(i+B2,kt); it < itend; ++it)
46      for(jt = j, jtend = MIN(j+B3,kt); jt < jtend; ++jt)
47       temp[it][jt] = MIN(A[it][jt], (A[it][kt] + A[kt][jt]));
48       ++kt;
49      for(it = i, itend = MIN(i+B2,kt); it < itend; ++it)
50       for(jt = j, jtend = MIN(j+B3,kt); jt < jtend; ++jt)
51        A[it][jt] = MIN(temp[it][jt],(temp[it][kt]+temp[kt][jt]))
              ;
52    }
```

Figure 3.6: Tiled Floyd-Warshall implementation

Analysis Fig. 3.11 shows the reuse distance profile of the original code and after convex-partitioning for the Givens rotation algorithm, for an input matrix of size 30 by 30. Studies on the impact of the various heuristic parameters on the quality of the convex partitioning is shown in Fig. 3.8.

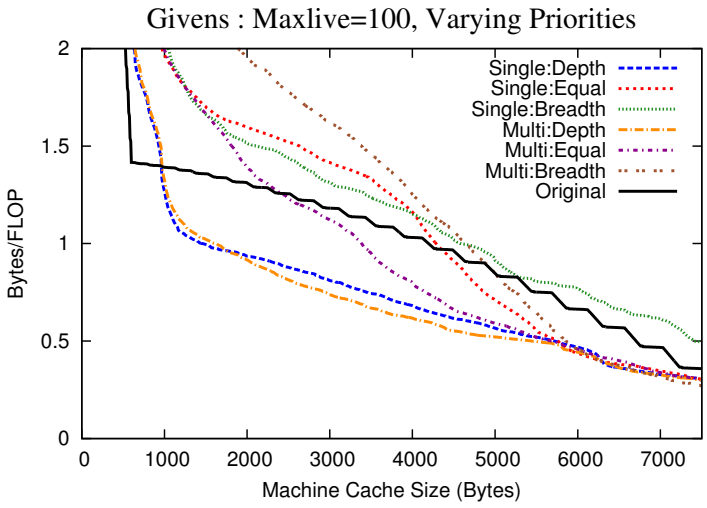
The convex partitioning analysis shows good potential for data locality improvement. Similarly to Floyd-Warshall, this code can be automatically tiled by a polyhedral-based compiler [16], after implementing simple loop normalization techniques. However, this is not sufficient to tile all dimensions. Subsequent transformations are needed, as shown below.

Modified implementation Based on the indicated potential for data locality enhancement, the code in Listing 3.7 was carefully analyzed and then manually modified to enhance the applicability of automated tiling techniques. Fig. 3.9 shows this modified version.

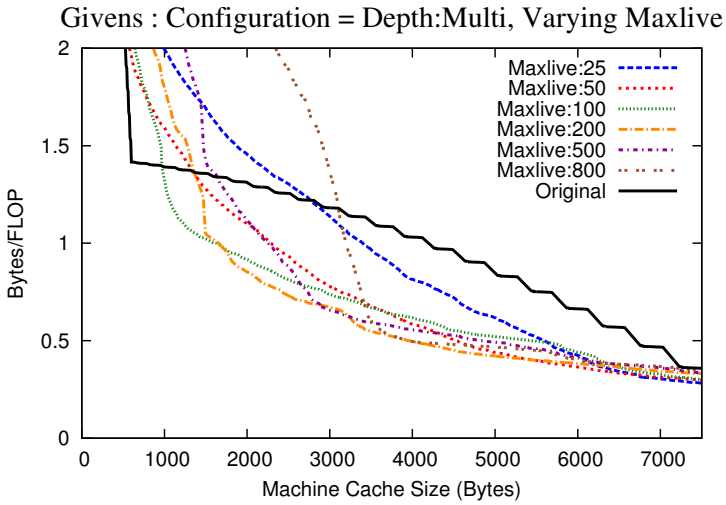
It was obtained by first applying loop normalization [47] which consists in making all loops iterate from 0 to some greater value while appropriately modifying the expressions involving the loop iterators within the loop body. Then, we applied scalar expansion on c

```
1  for (j = 0; j < N; j++) {
2    for (i = M-2; i >= j; i--) {
3      double c =  A[i][j] / sqrt(A[i][j]*A[i][j] + A[i+1][j]*A[i
           +1][j]);
4      double s = -A[i+1][j] / sqrt(A[i][j]*A[i][j] + A[i+1][j]*A[i
           +1][j]);
5      for (k = j; k < N; k++) {
6        double t1 = c * A[i][k] - s * A[i+1][k];
7        double t2 = s * A[i][k] + c * A[i+1][k];
8        A[i][k]   = t1;
9        A[i+1][k] = t2;
10     }
11   }
12 }
```

Figure 3.7: Givens Rotation



(a)



(b)

Figure 3.8: Results with different heuristics for Givens Rotation

and `s` [47] to remove dependences induced by those scalars which make loop permutation illegal. As a result, the modified code is an affine code with fewer dependences, enabling it to be automatically tiled by the Pluto compiler [76]. The final tiled code obtained, with default tile-sizes, is shown in Fig. 3.10.

```
1  for (j = 0; j < N; j++) {
2    for (i = 0; i <= M-2 - j; i++) {
3      c[i][j] =  A[(M-2)-(i)][j] / sqrt(A[(M-2)-(i)][j]*A[(M-2)-(i)][j]
4                          + A[(M-2) - (i)+1][j]*A[(M-2) - (i)+1][j]);
5      s[i][j] = -A[(M-2)-(i)+1][j] / sqrt(A[(M-2)-(i)][j]*A[(M-2)-(i)][j]
6                          + A[(M-2)-(i)+1][j]*A[(M-2)-(i)+1][j]);
7      for (k = j; k < N; k++) {
8        A[(M-2)-(i)][k] = c[i][j]*A[(M-2)-(i)][k] - s[i][j]*A[(M-2)-(i)+1][k];
9        A[(M-2)-(i)+1][k] = s[i][j]*A[(M-2)-(i)][k] + c[i][j]*A[(M-2)-(i)+1][k];
10     }
11   }
12 }
```

Figure 3.9: Modified Givens Rotation before tiling

Performance comparison Fig. 3.11(a) shows the improvements in the reuse distance profile using the convex partitioning heuristics and the improved reuse distance profile of the tiled code. A better profile is obtained for the tiled version than for convex partitioning. Similarly to Matmult, this is because our convex partitioning heuristic makes simplification for scalability and is therefore not guaranteed to provide the best achievable reuse profile.

Fig 3.11(b) shows a two-fold improvement in the performance of the transformed code for a matrix of size 4000.

Householder Transformation

Original program Fig. 3.12 shows the original input code for the Householder transform, another approach for QR decomposition.

Comparing this with Givens (a different approach to compute the QR decomposition) in terms of data locality potential is of interest: if one has better locality potential than the other, then it would be better suited for deployment on machines where the data movement cost is the bottleneck. It complements complexity analysis, which only characterizes the total number of arithmetic operations to be performed. Indeed, on hardware where the

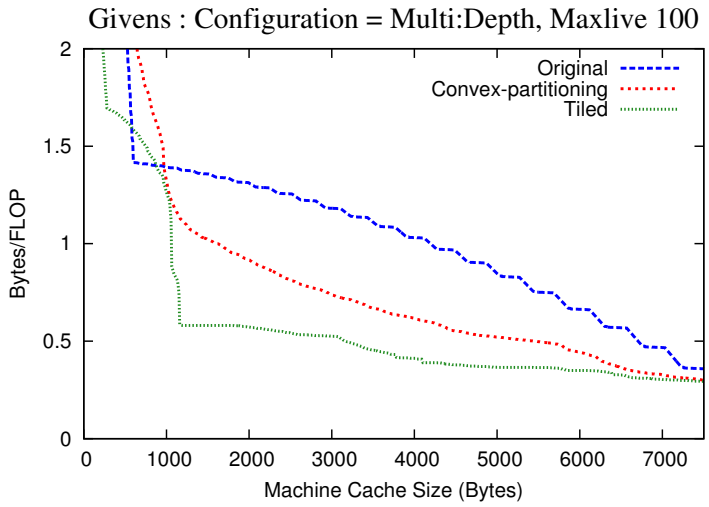
```
1  if (((M >= 2) && (N >= 1))) {
2   for (c0 = 0;c0 <= min(floord((M-2), 256),floord((N-1),256));c0++){
3    for (c1 = c0;c1 <= floord((M-2), 256); c1++) {
4     for (c2 = c0;c2 <= floord((N-1), 256); c2++) {
5      if ((c0 == c2)) {
6       for (c3 = (256*c0);c3 <= min(min((M-2),(N-1)),((256*c0)+255));c3++){
7        for (c4 = max((256*c1),c3);c4 <= min((M-2),((256*c1)+255));c4++){
8         c[((-1 * c3) + c4)][c3] = A[(M-2)-(((-1 * c3) + c4))][c3]
9                                 / sqrt(A[(M-2)-(((-1 * c3) + c4))][c3]
10                                 * A[(M-2)-(((-1 * c3) + c4))][c3]
11                                 + A[(M-2)-(((-1 * c3) + c4))+1][c3]
12                                 * A[(M-2)-(((-1 * c3) + c4))+1][c3] );
13        s[((-1 * c3) + c4)][c3] = -A[(M-2)-(((-1 * c3) + c4))+1][c3]
14                                / sqrt(A[(M-2)-(((-1 * c3) + c4))][c3]
15                                 * A[(M-2)-(((-1 * c3) + c4))][c3]
16                                  + A[(M-2)-(((-1 * c3) + c4))+1][c3]
17                                  * A[(M-2)-(((-1 * c3) + c4))+1][c3]);
18        for (c5 = c3; c5 <= min((N + -1), ((256 * c0) + 255)); c5++) {
19         A[(M-2)-(((-1 * c3)+c4))][c5] = c[((-1*c3)+c4)][c3]
20                                        * A[(M-2)-(((-1*c3) + c4))][c5]
21                                        - s[((-1*c3)+c4)][c3]
22                                        * A[(M-2)-(((-1*c3)+c4))+1][c5];
23         A[(M-2)-(((-1 * c3)+c4))+1][c5] = s[((-1*c3)+c4)][c3]
24                                          * A[(M-2)-(((-1*c3)+c4))][c5]
25                                          + c[((-1*c3)+c4)][c3]
26                                          * A[(M-2)-(((-1*c3)+c4))+1][c5];
27     } } } }
28     if (c0 <= (c2 + -1)) {
29      for (c3 = (256*c0);c3 <= min((M-2),((256*c0)+255));c3++){
30       for (c4 = max((256*c1),c3);c4 <= min((M-2),((256*c1)+255));c4++){
31        for (c5 = (256*c2);c5 <= min((N-1),((256*c2)+255));c5++) {
32         A[(M-2)-(((-1 * c3)+c4))][c5] = c[((-1 * c3) + c4)][c3]
33                                        * A[(M-2)-(((-1 * c3) + c4))][c5]
34                                        - s[((-1 * c3) + c4)][c3]
35                                        * A[(M-2)-(((-1*c3)+c4))+1][c5];
36         A[(M-2)-(((-1 * c3)+c4))+1][c5] = s[((-1 * c3) + c4)][c3]
37                                          * A[(M-2)-(((-1*c3)+c4))][c5]
38                                          + c[((-1 * c3) + c4)][c3]
39                                          * A[(M-2)-(((-1*c3)+c4))+1][c5];
40        }  }  } } }  } } }
```

Figure 3.10: Modified and automatically tiled Givens Rotation

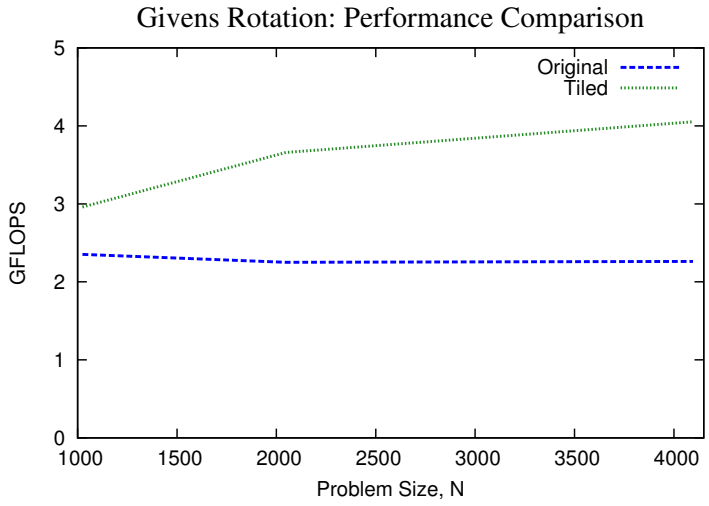
computation power has become increasingly cheaper relative to data access costs, standard complexity analysis alone is insufficient to capture the relative merits of alternative algorithms for a computation such as QR decomposition.

Analysis Fig. 3.13 shows the reuse distance profile of the original code and the convex-partitioning for the Householder algorithm, for an input matrix of size 30 by 30.

We observe a significant difference compared with Givens: the gap between the reuse distance profile of the original code and that found by our dynamic analysis is negligible.



(a)



(b)

Figure 3.11: Givens Rotation: performance improvements due to tiling

```
1  for (j = 0; j < N; j++) {
2    total = 0;
3    for (i = j+1; i < M; i++) {
4      total += A[i][j] * A[i][j];
5    }
6    norm_x = (A[j][j] * A[j][j] + total);
7    if (norm_x != 0) {
8      if (A[j][j] < 0)
9        norm_x = -norm_x;
10     v[j] = norm_x + A[j][j];
11     norm_v = (v[j] * v[j] + total);
12     v[j] /= norm_v;
13     for (i = j+1; i < M; i++) {
14       v[i] = A[i][j] / norm_v;
15     }
16     for (jj = j; jj < N; jj++) {
17       dot = 0.;
18       for (kk = j; kk < M; kk++) {
19         dot += v[kk] * A[kk][jj];
20       }
21       for (ii = j; ii < M; ii++) {
22         A[ii][jj] -= 2 * v[ii] * dot;
23       }
24     }
25   }
26 }
```

Figure 3.12: Householder computation

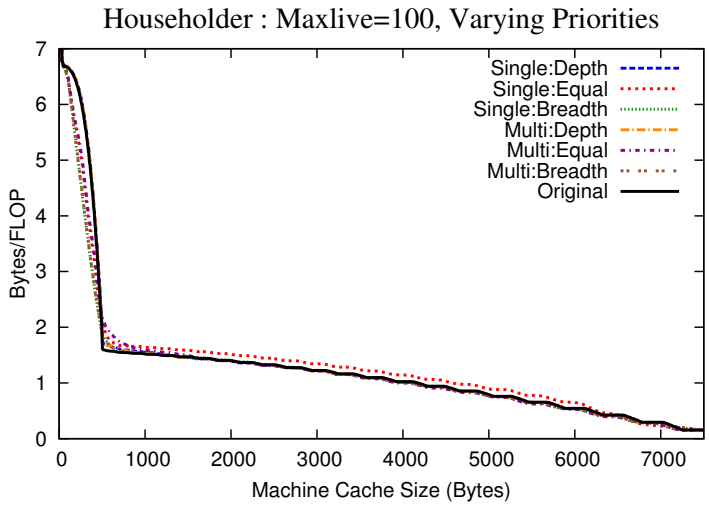
From this we conclude that the potential for data locality improvement of this algorithm is limited, and therefore we did not seek an optimized implementation for it.

Furthermore, comparing the bytes/flop required with the Givens graph in Fig. 3.11(a) shows that our tiled implementation of Givens achieves a significantly lower byte/flop ratio, especially for small cache sizes. We conclude that the Givens rotation algorithm may be better suited for deployment on future hardware, because of its lower bandwidth demand than Householder, especially for small cache sizes.

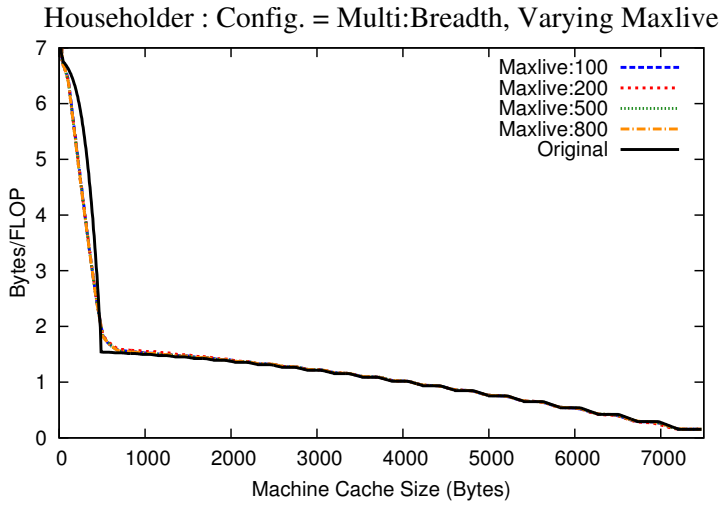
Lattice-Boltzmann Method

Original program *470.lbm*, a SPEC2006 [42, 77] benchmark, implements the Lattice-Boltzmann technique to simulate fluid flow in 3 dimensions, in the presence of obstacles.

The position and structure of the obstacles are known only at run-time, but do not change throughout the course of the computation.



(a)



(b)

Figure 3.13: Results with different heuristics for Givens Rotation

Analysis The convex-partition heuristics account for the data dependent behavior of the computation and are able to find valid operation reordering with enhanced data locality, as shown Fig. 3.14. The *test* input size provided by the SPEC benchmark suite was used for the analysis. To reduce the size of the generated trace the problem size was reduced by a factor of 4 along each dimension. The reduced problem still has the same behavior as the original problem.

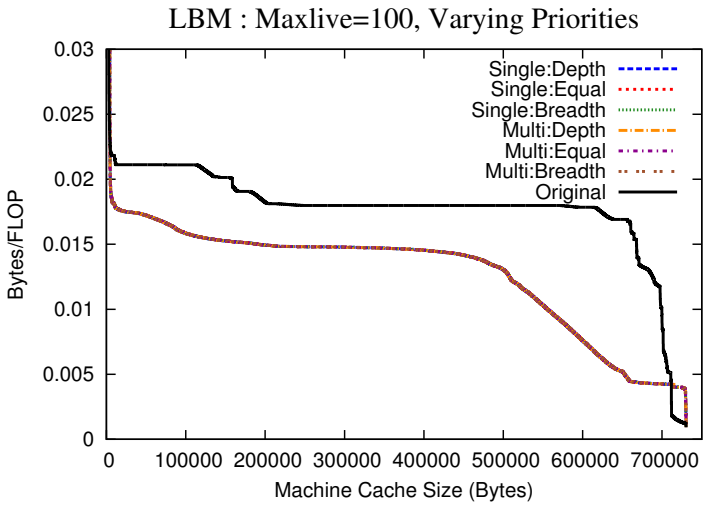
For a cache size of 60KB, the reordering after convex partitioning obtained by the heuristics show an improvement in the Bytes/FLOP ratio. For this benchmark all configurations of the heuristics yield essentially identical results. However, unlike the previous benchmarks, the absolute value of the bytes/flop is extremely low (Fig. 3.14), indicating that the computation is already compute-bound and a tiled version of the code would not be able to achieve significant improvements in performance over the untiled code. On an Intel Xeon E5640 with a clock speed of 2.53GHz, the untiled version already achieves a performance of 4GFLOPS. But since the current trend in hardware architecture suggests that the peak performance will continue to grow at a faster rate than the increase in main-memory bandwidth, it is reasonable to expect that optimizations like tiling that improve data locality will be critical in the future even for such computations that are currently compute-bound.

410.bwaves

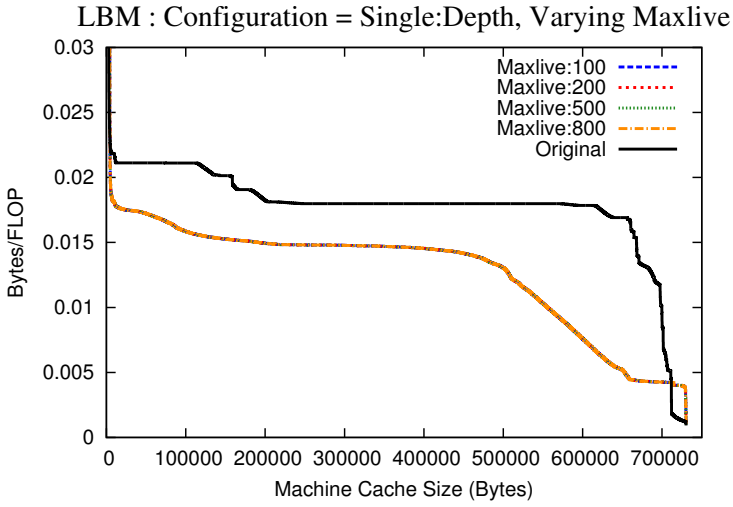
This benchmark is a computational fluid dynamic application from SPEC2006 [42]. We ran our analysis on the whole benchmark with *test* input size. For this benchmark too, the size of the problem was reduced by a factor of 4 along each dimension. The result of the analysis, shown in Fig. 3.15, indicate a limited potential for improving data locality.

Large-Eddy Simulations with Linear-Eddy Model in 3D

437.leslie3d is another computational fluid dynamic benchmark from SPEC2006 [42]. Here too, the analysis was done using the *test* dataset as given. As shown in Fig. 3.16,



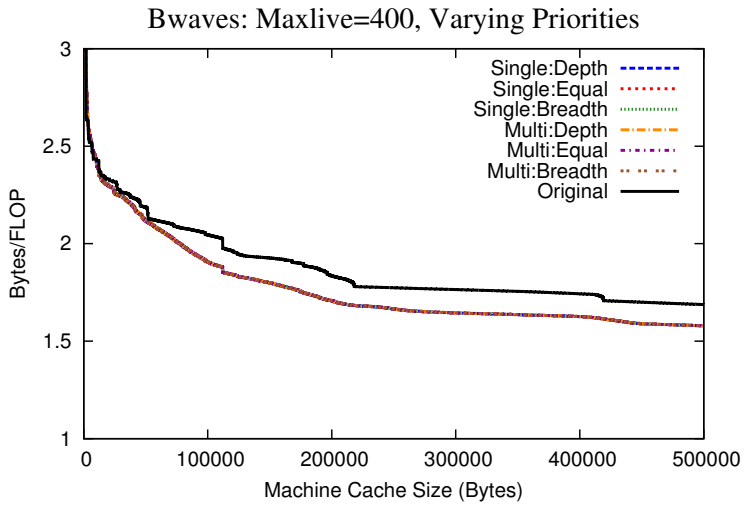
(a)



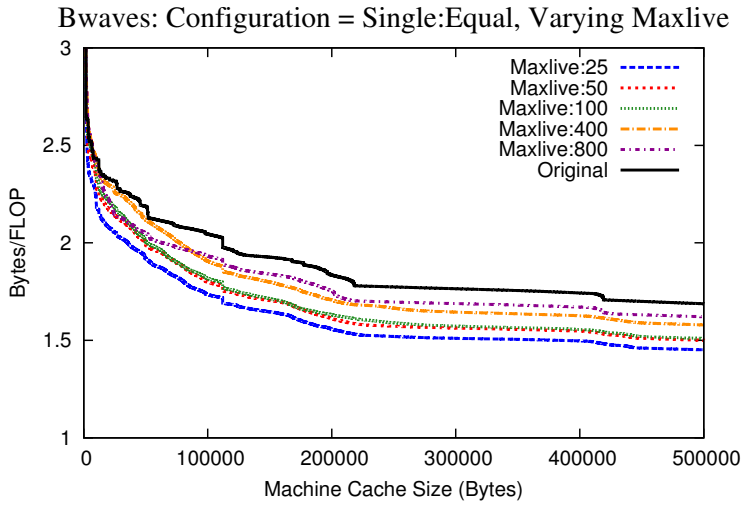
(b)

Figure 3.14: Results with different heuristics for 470.lbm

leslie3d achieves a lower bytes/FLOP ratio with the multi-level algorithm. The trend is not sensitive to varying Maxlive. Therefore, from the results, we conclude that this benchmark



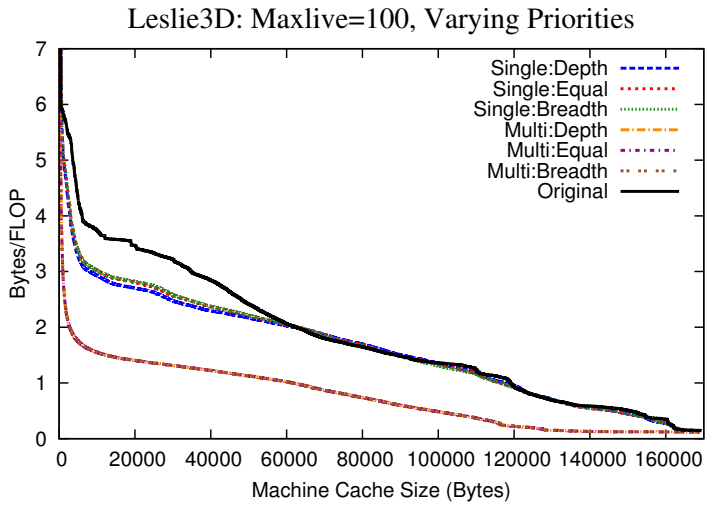
(a)



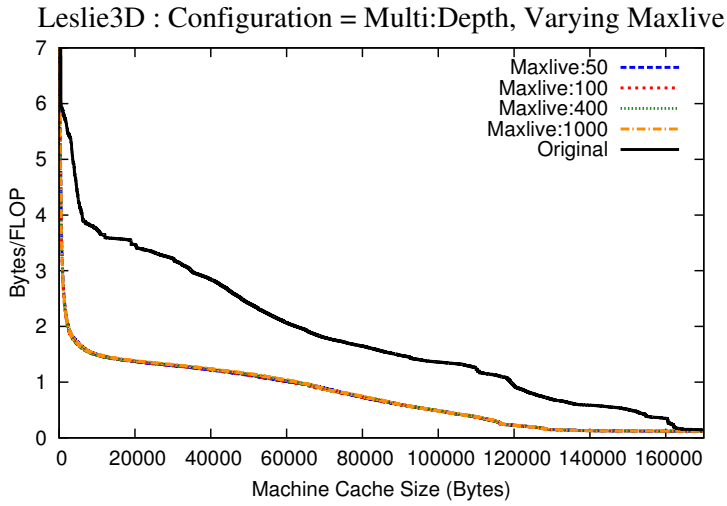
(b)

Figure 3.15: Results with different heuristics for 410.bwaves

has high potential for locality improvement. We however leave for future work the task of deriving an optimized implementation for leslie3d.



(a)



(b)

Figure 3.16: Results with different heuristics for 437.leslie3d

Odd-Even Sort

Original program Our dynamic analysis does not impose any requirement on the data layout of the program: arrays, pointers, structs etc. are seamlessly handled as the trace extraction tool focuses exclusively on the address used in memory read/write operations. To illustrate this we show in Fig. 3.17 the original code for an odd-even sorting algorithm, using a linked-list implementation. `CompareSwap` compares the data between two consecutive elements in the list, and swaps them if necessary. Each swap operation is represented by a node in the CDAG.

```
1 for(i=0; i<N/2; ++i) {
2   node *curr;
3   for(curr=head->nxt; curr->nxt; curr=curr->nxt->nxt) {
4     CompareSwap(curr, curr->nxt);
5   }
6   for(curr=head; curr; curr=curr->nxt->nxt) {
7     CompareSwap(curr, curr->nxt);
8   }
9 }
```

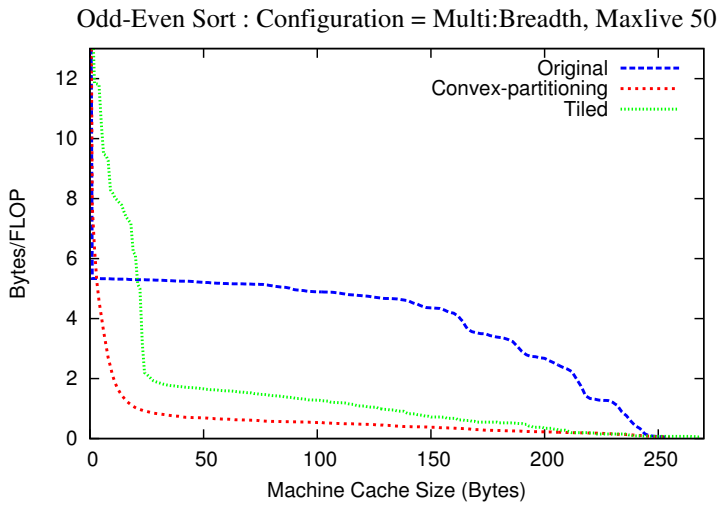
Figure 3.17: Odd-Even sort on linked list

Analysis We have performed our analysis on the original code, for a input list of size 256, with random values. The profile of the original, best convex partitioning (obtained with the best set of heuristic parameters for this program) and tiled (our modified) implementation are shown shown in the left plot in Fig. 3.19.

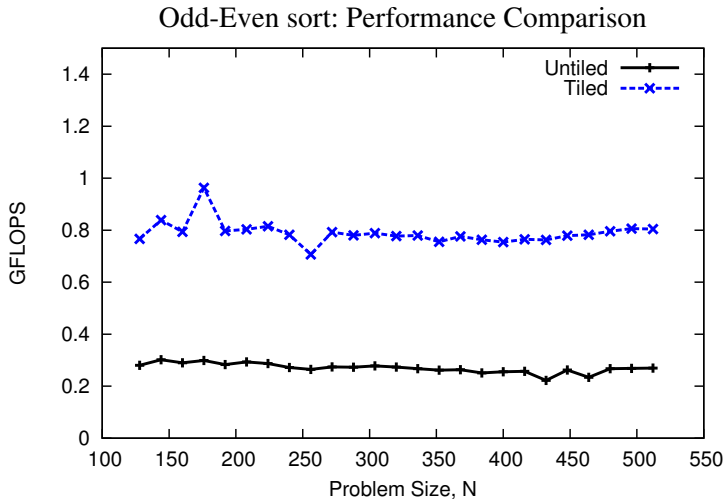
Modified implementation Based on careful analysis of the original code in Fig. 3.17, an equivalent register-tiled version with a tile size of 4 was manually developed. It is shown in Fig. 3.18.

```
1  register float Ra, Rb, Rc, Rd;
2  node *tail0, *tail1, *tail2, *tail3;
3  tail0 = tail;
4  tail1 = tail = tail->prev;
5  tail2 = tail = tail->prev;
6  tail3 = tail = tail->prev;
7
8  // Upper left part of the iteration space
9  node *curr = head;
10 for (I=0; I<N; I+=4) {
11   ptr0 = curr;  Ra = ptr0->data;
12   ptr1 = ptr0->nxt;  Rb = ptr1->data;
13   ptr2 = ptr1->nxt;  Rc = ptr2->data;
14   ptr3 = ptr2->nxt;  Rd = ptr3->data;
15   curr = ptr3->nxt;
16   // Full tiles
17   for (T=I; T>0; T-=4) {
18     CompareSwap(Ra, Rb); CompareSwap(Rc, Rd);
19     ptr3->data = Rd; ptr3 = ptr0->prev; Rd = ptr3->data;
20     CompareSwap(Rd, Ra); CompareSwap(Rb, Rc);
21     ptr2->data = Rc; ptr2 = ptr3->prev; Rc = ptr2->data;
22     CompareSwap(Rc, Rd); CompareSwap(Ra, Rb);
23     ptr1->data = Rb; ptr1 = ptr2->prev; Rb = ptr1->data;
24     CompareSwap(Rb, Rc); CompareSwap(Rd, Ra);
25     ptr0->data = Ra; ptr0 = ptr1->prev; Ra = ptr0->data;
26   }
27   // Half tile corresponding to T==0
28   CompareSwap(Ra, Rb); CompareSwap(Rc, Rd);
29   ptr3->data = Rd;
30   CompareSwap(Rb, Rc);
31   ptr2->data = Rc;
32   CompareSwap(Ra, Rb);
33   ptr1->data = Rb;
34   ptr0->data = Ra;
35 }
36 // Lower right part of the iteration space
37 for (I=4; I<=N; I+=4) {
38   // Half tile corresponding to T==N
39   ptr3 = tail0; Rd = tail0->data;
40   ptr2 = tail1; Rc = tail1->data;
41   CompareSwap(Rc, Rd);
42   ptr1 = tail2; Rb = tail2->data;
43   CompareSwap(Rb, Rc);
44   ptr0 = tail3; Ra = tail3->data;
45   // Full tiles
46   for (T=N-4; T>=I; T-=4) {
47     CompareSwap(Ra, Rb); CompareSwap(Rc, Rd);
48     ptr3->data = Rd; ptr3 = ptr0->prev; Rd = ptr3->data;
49     CompareSwap(Rd, Ra); CompareSwap(Rb, Rc);
50     ptr2->data = Rc; ptr2 = ptr3->prev; Rc = ptr2->data;
51     CompareSwap(Rc, Rd); CompareSwap(Ra, Rb);
52     ptr1->data = Rb; ptr1 = ptr2->prev; Rb = ptr1->data;
53     CompareSwap(Rb, Rc); CompareSwap(Rd, Ra);
54     ptr0->data = Ra; ptr0 = ptr1->prev; Ra = ptr0->data;
55   }
56   ptr0->data = Ra; ptr1->data = Rb; ptr2->data = Rc; ptr3->data
         = Rd;
57 }
```

Figure 3.18: Tiled odd-even sort



(a)



(b)

Figure 3.19: Odd-Even sort: Performance improvements due to tiling

Performance comparison The comparison of performance of the untiled and tiled versions of the code is shown in Fig. 3.19. Fig. 3.19(a) shows the improved data locality for the tiled code compared to the original code. The actual improvement in performance of

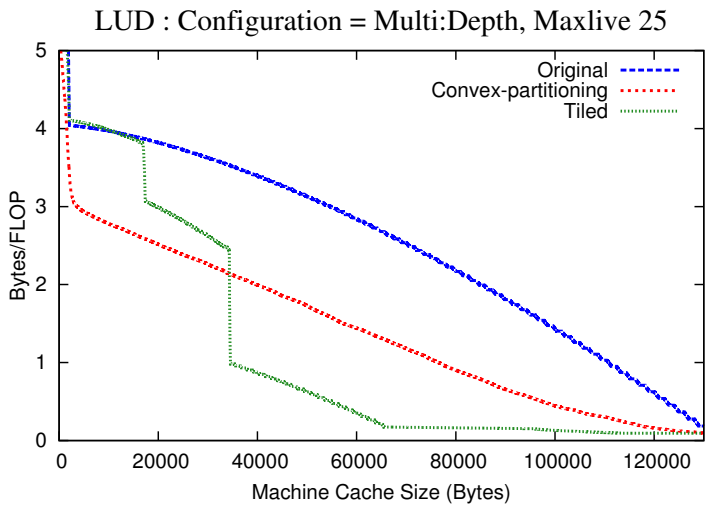
the tiled code is shown in Fig. 3.19(b) for a random input. Experiments about sensitivity to datasets reported in later Sec. 3.4.3 confirm that our optimized variant consistently outperform the original code.

LU Decomposition (LAPACK)

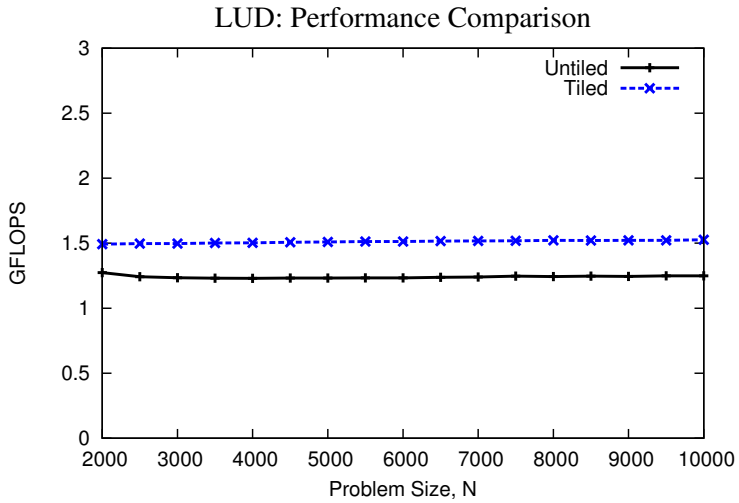
The last benchmark we analyze is an implementation of the LU decomposition for dense matrices, from the LAPACK package [6]. It uses pivoting (therefore the computation is input-dependent) and LAPACK provides both a base implementation meant for small problem sizes, and a block decomposition for large problem sizes [54]. Both code versions can be found in [6].

We have run our dynamic analysis on the non-blocked implementation of LU decomposition, for a single random matrix of size 128 by 128. The results of varying heuristics and maxlive can be found in [30].

A potential for locality improvement is established from the result of the analysis. On Fig. 3.20-left, we plot the reuse distance profile of the non-blocked implementation; the convex partitioning with the best heuristic parameters for this program; and tiled (i.e., blocked) implementation, in addition to the original (i.e., non-blocked) one. The blocked version shows a better bytes/Flop than the convex partitioning for cache sizes larger than 35kB. This is likely due to inter-block reuse achieved in the highly optimized LAPACK implementation, combined with a sub-optimal schedule found by our heuristic. Further, Fig. 3.20-right shows actual performance, in GFLOPS, for non-blocked and blocked versions of the code.



(a)



(b)

Figure 3.20: LU Decomposition

3.4.3 Dataset Sensitivity Experiments

We conclude our experimental analysis with a study of the sensitivity of our analysis to different datasets. In the set of benchmarks presented in the previous section, the majority of them have a CDAG that depends only on the input dataset size, and not on the input

values. Therefore for these codes our dynamic analysis results hold for any dataset of identical size.

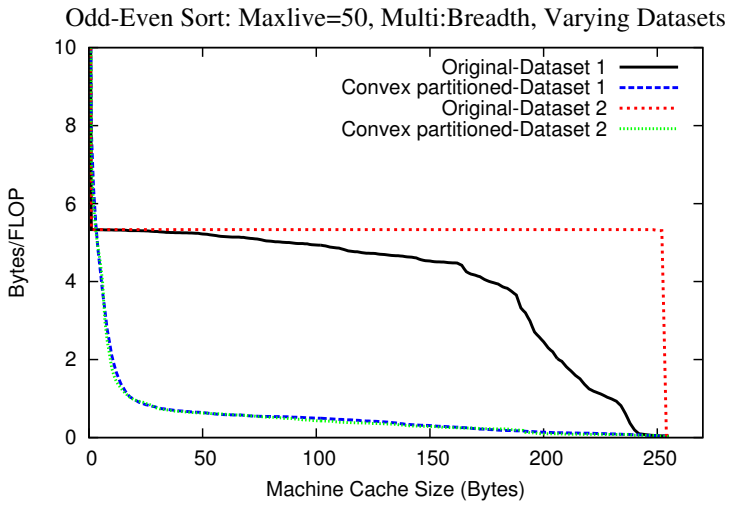


Figure 3.21: Sensitivity analysis for odd-even sort

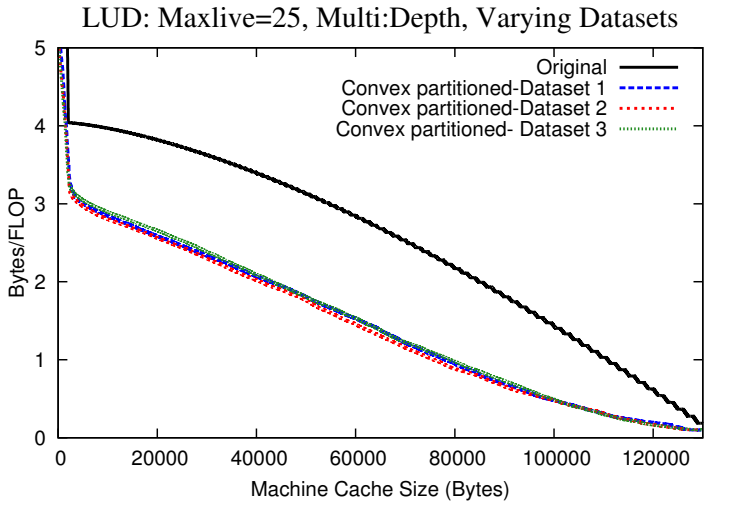


Figure 3.22: Sensitivity analysis for LUD

Odd-even sort and LUD are two benchmarks that are input-dependent. To determine the sensitivity of the convex-partitioning heuristics on the input, we ran the heuristics on different datasets, as shown in Fig. 3.21 and Fig. 3.22.

Fig. 3.21 shows the result for two such datasets - one with random input elements and the other with the reverse sorted input list, a worst-case dataset. We used multi-level heuristics with breadth-first priority, which corresponds to the parameters of the best result, which can be found in [30]. Fig. 3.21 shows that the potential for improvement exhibited by the heuristics remains consistent with varying inputs, i.e., the "Convex partitioned" reuse distance profile does not vary with the input value. We note that in the general case, some variations are expected for different datasets. Similar to complexity analysis of such algorithms, one needs to perform both worst-case analysis (i.e., reverse-sorted) and analysis on random/representative datasets for best results.

Fig. 3.22 exhibits a similar behavior where the three tested datasets have a similar analysis profile. The first dataset has is a random matrix, the second was created so that the pivot changes for about half the rows of the matrix, and for the third one the pivot changes for all rows of the matrix.

Finally, we complete our analysis by reporting in Fig. 3.23-3.24 the performance of our manually optimized programs for odd-even sorting and LU decomposition. For two programs (Original, and our modified implementation Tiled), we plot the performance for various datasets (different curves) and various sizes for those datasets (different points on the x axis). We observe very similar asymptotic performance for the various datasets.

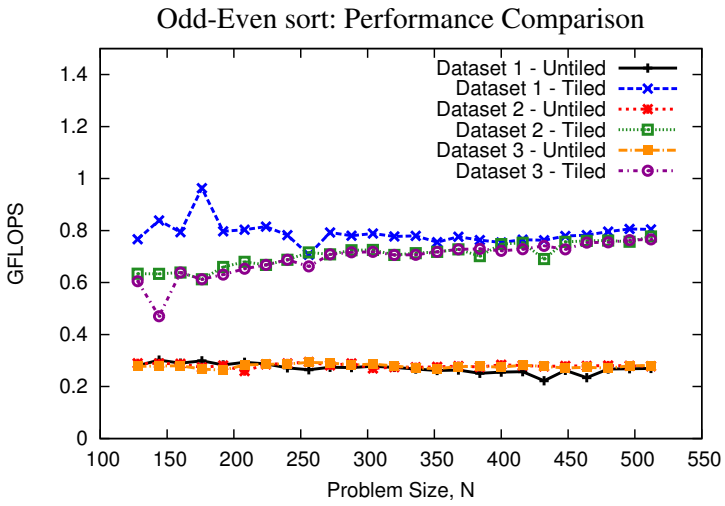


Figure 3.23: Performance for odd-even sort

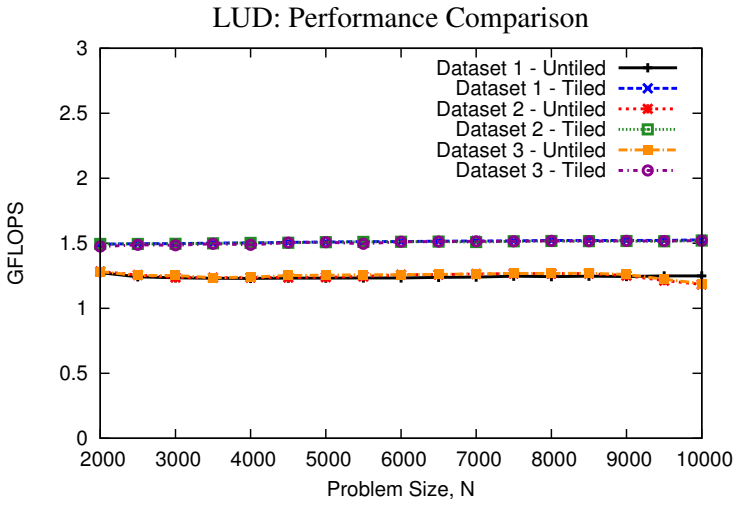


Figure 3.24: Performance for LUD

3.5 Related Work

Both algorithmic approaches (e.g., [8, 9, 28]) and compiler transformations (e.g., [16, 44, 48, 101]) have been employed to improve data locality. The applicability of these techniques to arbitrary computations is limited. For example, compiler optimizations typically require programs in which precise static characterization of the run-time behavior is possible; this is challenging in the presence of inter-procedural control flow, diverse data structures, aliasing, etc. Reuse distance analysis [29, 65], which considers the actual observed run-time behavior, is more generally applicable and has been used for cache miss rate prediction [46, 64, 111], program phase detection [87], data layout optimizations [112], virtual memory management [19], and I/O performance optimizations [45]. Even though reuse distance analysis provides insights into the data locality of software behavior, it has been limited to analyzing locality characteristics for a specific execution order of the operations, typically that generated by a sequential program. In contrast to all previous work on reuse distance analysis, to the best of our knowledge, our upper-bounding approach is the first to attempt a schedule-independent characterization of the inherent data locality characteristics of a CDAG.

The idea of considering valid re-orderings of a given execution trace has been applied successfully to characterize the potential parallelism of applications. Kumar's approach [52] computes a possible valid schedule using a timestamping analysis of an instrumented statement-level execution of the sequential program. Shadow variables are used to store the last modification times for each variable. Each run-time instance of a statement is associated with a timestamp that is one greater than the last-modify times of all its operands. A histogram of the number of operations at each time value provides a fine-grained parallelism profile of the computation, and the maximal timestamp represents the

critical path length for the entire computation. Other prior efforts with a similar overall approach include [7, 35, 52, 53, 63, 67, 78, 81, 88, 89, 100].

In contrast to the above fine-grained approach, an alternate technique developed by Larus [55] performed analysis of loop-level parallelism at different levels of a nested loop. Loop-level parallelism is measured by forcing a sequential order of execution of statements within each iteration of a loop being characterized, so that the only available concurrency is across different iterations of that loop. A related technique is applied in the context of speculative parallelization of loops, where dynamic dependences across loop iterations are tracked [82]. A few recent approaches of similar nature include [18, 73, 90, 91, 105, 110]. In order to estimate parallel speedup of DAGs, Sarkar and Hennessy [85] developed convex partitioning of DAGs. In previous work, we [43] used dynamic analysis of CDAGs to assess the vectorization potential of codes that are not effectively vectorized by current vectorizing compilers. However, we are not aware of any prior work on dynamic analysis of CDAGs with the goal of characterizing and/or enhancing data locality properties of computations.

3.6 Discussion

In this section, we discuss the potential and some of the current limitations of the dynamic analysis approach for data locality characterization/enhancement that we have developed in this article.

Dependence on Input Values As with any work that uses dynamic analysis of the actual execution trace of a program, any conclusions drawn are only strictly true for that particular execution. For programs where the execution trace is dependent on input data, the CDAG

will change for different runs. Due to space limitations, we only present RDA results for a single problem size for each benchmark. However, we have experimented with different problem sizes and the qualitative conclusions remain stable across problem size. Further, as demonstrated by the case studies of the Floyd-Warshall and Givens rotation codes, the modified codes based on insights from the dynamic analysis were demonstrated to exhibit consistent performance improvement for different problem sizes.

Overhead of Analysis The initial prototype implementation of the partitioning algorithm has not yet been optimized and currently has a fairly high overhead (about 4 orders of magnitude) compare to the execution time. As discussed earlier, the computational complexity of the partitioning algorithm is $O(|T|)$ for the single-level version, and $O(|T| \log(|M|))$ for the multi-level version and can therefore be made sufficiently efficient to be able to analyze large-scale applications in their entirety.

Trace Size Limitations A more significant challenge and limitation of the current system we implemented is the memory requirement. The CDAG size is usually a function of the problem size. For instance, for 470.lbm using the *test* dataset as is would generate a CDAG of size 120GB. The additional data structures used by the tool quickly exhaust the memory on our machines. For instance for 470.lbm we were able to easily create a smaller input dataset (e.g., 1/64th) leading to a 3GB CDAG, that our tool could handle. However, there are numerous benchmarks where such reduction of the input dataset is not possible, and/or does not affect the CDAG size. For those codes, where the CDAG was beyond a few GBs, our current implementation fails due to insufficient memory. The development of an "out-of-core" analyzer is the focus of ongoing follow-up work. Another solution is to compress the CDAG using a technique similar to trace compression [49] leading to a space

complexity of $O(|M| + |P|)$ in the most favorable scenario (in which all dependences turn out to be affine). More generally trace sampling techniques can be applied to tackle scalability issues of this approach.

Tightness of Estimation The primary goal of our CDAG partitioning algorithm is to find a more favorable valid reordering of the schedule of its operations so as to lower the volume of data movement between main memory and cache. From the perspective of data movement, the lowest possible amount, which corresponds to the best possible execution order for a given CDAG and a given cache size, can be considered the *inherent data access complexity* of that computation. Irrespective of how much lower the reordered schedule's data movement volume is compared to the original schedule's data movement volume, how do we determine how close we are to the best possible valid order of execution? A possible solution is to work from the opposite direction and develop lower bound techniques for the data access complexity of CDAGs. In a complementary work we are developing an approach to establishing lower bounds on the data movement costs for arbitrary CDAGs. One way of assessing the tightness of the upper bounds (this work) and lower bounds (complementary work in progress) is to see how close these two bounds are for a given CDAG.

Use of Analysis We envision several uses of a tool based on the dynamic analysis approach developed in this work. (1) *For Application Developers*: By running the dynamic analysis on different phases of an application, along with standard performance profiling, it is possible to identify which of the computationally dominant phases of the application may have the best potential for performance improvement through code changes that enhance data reuse. Dynamic analysis for data locality can also be used to compare and choose

between alternate equivalent algorithms with the same functionality – even if they have similar performance on current machines. If the reuse distance profiles of two algorithms after reordering based on dynamic analysis are very different, the algorithm with lower bandwidth demands would likely be better for the future. (2) *For Compiler Developers*: Compilers implement many transformations like fusion and tiling that enhance data reuse. The results from the convex partitioning of the CDAG can be used to gauge the impact of compiler transformations and potential for improvement. (3) *For Architecture Designers*: Running the dynamic analysis tool on a collection of representative applications can guide vendors of architectures in designing hardware that provides adequate bandwidth and/or sufficient capacity for the different levels of the memory hierarchy.

3.7 Conclusion

With future systems, the cost of data movement through the memory hierarchy is expected to become even more dominant relative to the cost of performing arithmetic operations [14, 34, 86], both in terms of throughput and energy. Therefore optimizing data locality will become ever more critical in the coming years. Given the crucial importance of optimizing data access costs in systems with hierarchical memory, it is of great interest to develop tools and techniques for characterization and enhancement of the data locality properties of an algorithm. Although reuse distance analysis [29,65] provides a useful characterization of data locality for a given execution trace, it fails to provide any information on the potential for improving the in data reuse through valid reordering of the operations in the execution trace.

In this piece of work, we have developed a dynamic analysis approach to provide insights beyond what is possible from standard reuse distance analysis. Given an execution trace from a sequential program, we seek to (i) characterize the data locality properties of an algorithm and (ii) determine if there exists potential for enhancement of data locality through execution reordering. Since we first explicitly construct a dynamic computational directed acyclic graph (CDAG) to capture the statement instances and their interdependences; perform convex partitioning of the CDAG to generate a modified, dependence-preserving, execution order with better expected data reuse; and then perform reuse distance analysis on the trace corresponding to the modified execution order, we expect to get a better characterization of the potential benefit of reordering. We have demonstrated the utility of the approach in characterizing/enhancing data locality for a number of benchmarks.

CHAPTER 4

Convex Partitioning using Loop Induction Variable Information

4.1 Introduction

For many loop optimization and parallelization transformations, loop induction variables play an important role. An induction variable is defined as a variable whose value is systematically incremented or decremented by a constant value in a loop [2, 33]. Techniques have been developed to detect and classify a full range of loop induction variables for optimization and parallelization purposes [37, 102]. Loop induction variables are useful in operator strength reduction [24], vectorization [103], array subscript range checking [51], array privatization [92], software pipelining [80, 83, 84], reducing register pressure etc. Most importantly, for data-dependence tests, the array subscripts should be known in terms of the loop induction variables [74].

For data locality optimization and program characterization, our major focus lies on the loops of the program. Our convex partitioning heuristic performs loop tiling where the tiles can be of any arbitrary shape but convex and the final schedule must preserve the program dependence structure. Loop induction variables may help us to achieve better convex partitions.

The convex partitioning heuristic described in the previous chapter depends on some parameters like priority selection, maxlive value, single level or multi level partitioning, etc. Programs vary in their dependence structure, which results in different kinds of CDAGs. Therefore they behave differently for different applications using the same combination of parameters. No single configuration achieves the best results for all. In this chapter, we propose a new approach to convex partitioning, where we use loop induction variables as a major attribute for the partition building procedure. The new heuristic no longer uses the other parameters and instead just relies on the induction variables and a tile diameter - based on which a more conventional tiling is attempted initially. Then the tiles are expanded appropriately to preserve dependences as well as the convexity property. Depending on the CDAG of the programs, we may or may not end up with conventional cube tiles. We test the new heuristic on a number of benchmarks and show that the new heuristics performs comparably or better to the previous one, and provides the same level of insight about the inherent data locality property of the program. The new algorithm is more general and applicable to all programs. It no longer depends on proper parameter choice for different applications.

The rest of this chapter is organized as follows. Section 4.2 presents background on LLVM canonicalized induction variables and the impact of parameters on the heuristic. Section 4.3 describes the new dynamic analysis approach in detail. Section 4.4 presents experimental results, followed by concluding remarks in Section 4.5.

4.2 Background

In this section we provide the necessary background on the canonical induction variable produced by LLVM and also discuss the impact of parameters on the previously described heuristic (Chapter 3) and how the new approach based on loop induction variables can provide reduction in the required number of parameters.

4.2.1 Canonical Induction Variables

Loops in general can have multiple induction variables. This is especially true in hand-strength-reduced loops where arrays are referenced via incremented pointers [61]. Compiler optimizations that uses loop induction variables generally prefer to handle only a single unit-stride induction variable in each loop. Allowing them to make this assumption enables them to handle more forms of loops without additional implementation effort. This simplification process for induction variables is called canonicalization. Loop induction variable canonicalization normalizes loops by transforming the primary induction variable so that it is incremented by 1 at the end of each iteration.

LLVM is able to generate canonical induction variable - an integer recurrence that starts at 0 and increments by one each time through the loop. The IndVarSimplify pass transforms loops to have a canonical induction variable, which in turn uses the LLVM pass for analyzing scalar evolution. Another analysis pass IndVars is then able to generate canonical induction variables. Our loop pass for program instrumentation uses the getCanonicalInductionVariable method which checks whether the loop has a canonical induction variable or not. If available, the instrumentation process includes the values in the trace file.

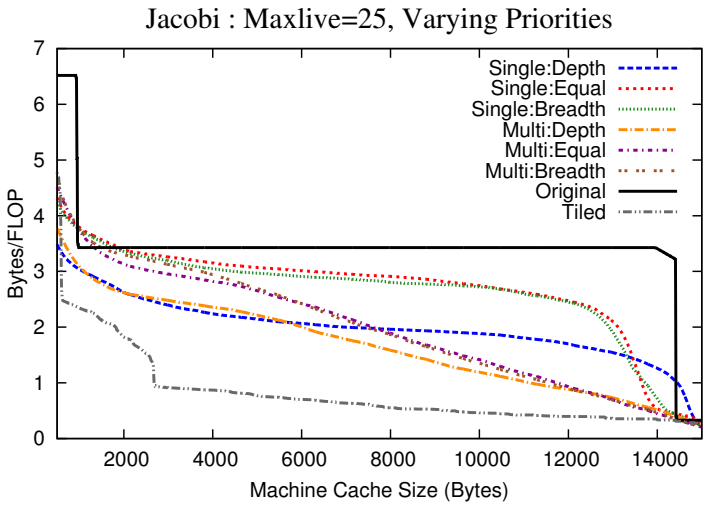
4.2.2 Impact of Heuristic Parameters

The convex partitioning heuristic described in Chapter 3 takes two parameters. First, we have the *Search Strategy*: this includes (a) prioritization in selecting a new vertex to include in a convex partition: depth-priority, breadth-priority, or alternation between depth and breadth priority (equal priority); and (b) single level partitioning versus multi-level partitioning. The second parameter is *Maxlive*: the parameter that sets a limit on the maximum number of live vertices allowed while forming a partition. In this section, we study the impact of choosing different values for these parameters on two different benchmarks - Jacobi 2D and Matmult.

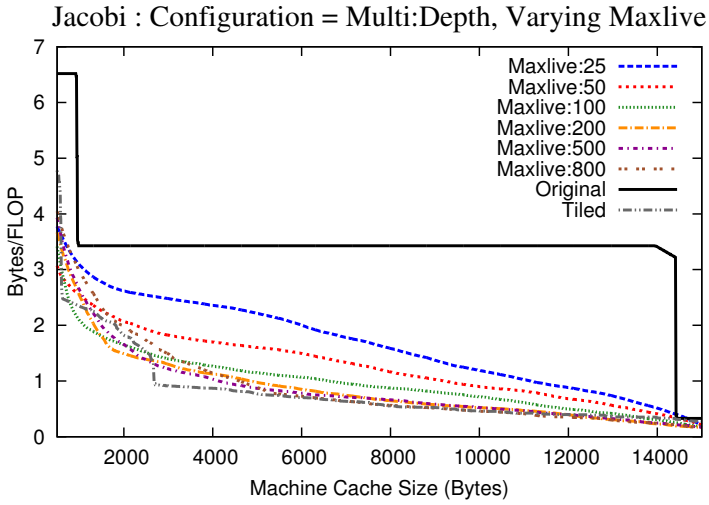
Jacobi 2D

Fig. 4.1 presents the results of applying the dynamic analysis on a Jacobi stencil on a regular 2-dimensional grid of size 32, and 30 time iterations.

Fig. 4.1(a) shows reuse distance profiles for a fixed value of $Maxlive$ and different configurations for single versus multi-level partitioning, and different priorities for next node selection. With single-level partitioning, depth-priority is seen to provide the best results. Using multi-level partitioning further improves the reuse distance profile. In order to judge the effectiveness of the convex partitioning in improving the reuse distance profile, we show both the reuse distance profiles for the original code and an optimally tiled version of the Jacobi code. It can be seen that there is significant improvement over the original code, but still quite some distance from the profile for the tiled code. Fig. 4.1(b) shows the effect of varying $Maxlive$ from 25 to 800, with multi-level depth-priority partitioning.



(a)



(b)

Figure 4.1: Results with different heuristics for Jacobi-2D

Here it can be seen that at large values of $Maxlive$, the profile is very close to that of the optimized tiled code. Thus, with a large value of $Maxlive$ and use of multi-level depth-priority partitioning, the convex partitioning heuristic is very effective for the Jacobi-2D benchmark.

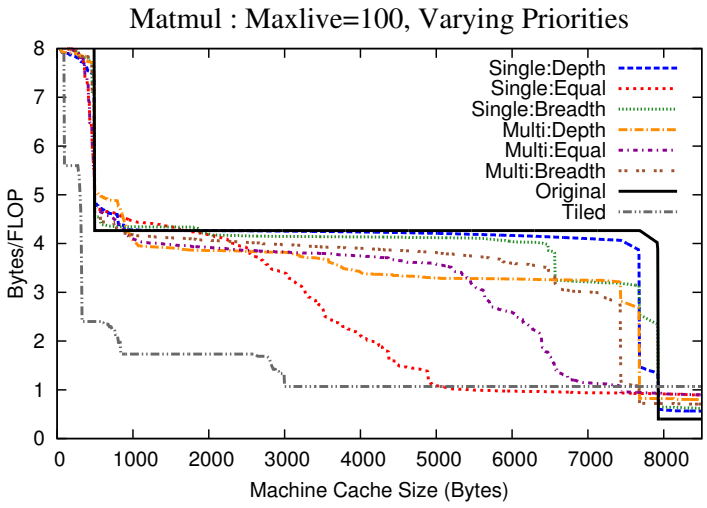
Matrix Multiplication

Fig. 4.2 shows experimental results for matrix multiplication, for matrices of size 30 by 30. In Fig. 4.2(a), the selection priority is varied, for single and multi-level partitioning. In contrast to the Jacobi benchmark, for Matmult, equal priority works better than breadth or depth priority. Further, single level partitioning provides better results than multi-level partitioning. In Fig. 4.2(b), we see performance variation as a function of $Maxlive$ for single-level equal-priority partitioning. Again the trends are quite different from those of Jacobi-2D: the best results are obtained with the lowest value of 25 for $Maxlive$.

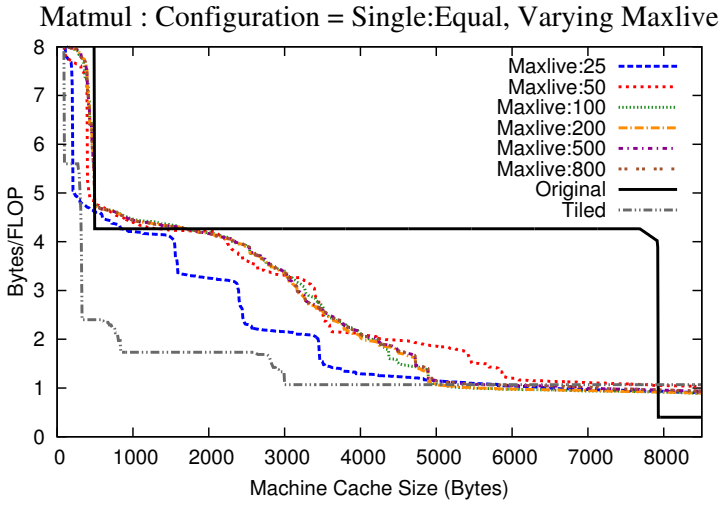
For Matmult, the tiled code has a better reuse distance profile than the best reported heuristic in Fig. 4.2. Our heuristics for building convex partitions use several simplifications to improve scalability, in particular in the choice of candidates to be inserted in a partition, and in scheduling the obtained partitions. In addition, we do not explore the Cartesian product of all possible parameters values (priority and maxlive values) but instead limit to a reasonable subset for scalability purposes. All these factors contribute to our analysis possibly under-estimating the data locality potential of the application.

These results suggest that no single setting of parameters for the convex partitioning heuristic is likely to be consistently effective across benchmarks. We conjecture that there may be a relationship between graph properties of the CDAGs (e.g., low fan-out vs. high fan-out) and the best parameters for the partitioning heuristic.

With the new approach, we use the loop induction variables for the convex partitioning algorithm instead of the neighbor-successor strategy. Therefore we no longer require the parameters used for the previous algorithm. We achieve similar or better reuse distance



(a)



(b)

Figure 4.2: Results with different heuristics for matrix multiplication

profile from the new approach but are able to avoid the significant dependence on the parameters.

4.3 Overview of Approach

The new dynamic analysis approach we propose attempts to characterize the inherent data locality properties of a given (sequential) computation, and to assess the potential for enhancing data locality via change of execution ordering. We perform analysis on the CDAG of a computation, attempting to find a different order of execution of the operations that can improve the reuse-distance profile compared to that of the given program's sequential execution trace. If this analysis reveals a significantly improved reuse distance profile, it suggests that suitable source code transformations have the potential to enhance data locality. On the other hand, if the analysis is unable to improve the reuse-distance profile of the code, it is likely that it is already as well optimized for data locality as possible. The dynamic analysis involves the following steps:

1. Generate a sequential execution trace of a program.

2. Run a reuse-distance analysis of the original trace.

3. Form a CDAG from the execution trace.

4. Perform a convex partitioning of the CDAG, which is then used to change the schedule of operations of the CDAG from the original order in the given input code.

5. Perform standard reuse distance analysis of the reordered trace after convex partitioning.

After such a partitioning, the execution order of the vertices is reordered so that the convex partitions are executed in some valid order (corresponding to a topological sort of a coarse-grained inter-partition dependence graph), with the vertices within a partition

being executed in the same relative order as the original sequential execution. Details are presented in the next section.

The key difference between the approach of the previous chapter and the approach presented in this chapter is the convex partitioning heuristic. We aim to reduce input parameter dependence of the heuristic and apply a more generally applicable algorithm that works well for any application. We use the loop induction variable generated from LLVM pass *IndVars*, process it more to make it appropriate for our purpose and use this information during the partitioning heuristic. While growing each partition we try to achieve tiles close to size T^d where d is the maximum loop depth and T is a tunable tile size. Section 4.3.1 describes the details of processing of the loop induction variables associated with the run-time program instances. Then we explain the preprocessing performed on the CDAG in section 4.3.2. Finally, section 4.3.3 covers the details of the new convex partitioning algorithm.

4.3.1 Formatting the Induction Variables

For our new heuristic, we need to know the loop iterator values/induction variables for the run time instances of program statements. The target code is instrumented in such way that the run time trace contains the LLVM generated canonical induction variable (Section 4.2.1) information for all statements. We further process the available loop induction variables to better serve our purpose.

To differentiate between basic blocks in a loop, we use $2d+1$ induction variables, where d is the number of surrounding loops of the statement instance. For example, if the target code has 3 nested loops L_i, L_j and L_k, the statement instances that is perfectly nested by all three loops will have a 7 element induction variables vector (i.e., $(0, i, 1, j, 1, k, 0)$, i =

```
1  for(i=0; i<N; i++){
2    S1;                                    [0, i, 0]
3    for(j=0; j<N ; j++){
4      S2;                              [0, i, 1, j, 0]
5    }
6    S3;                                    [0, i, 2]
7    for(k=0; k<N; k++) {
8      S4;                              [0, i, 3, k, 0]
9    }
10   S5;                                    [0, i, 4]
11 }
```

Listing 4.1: Induction Variables Before Pre-processing

L_i loop iterator value, j = L_j loop iterator value and so on. The other elements are the basic block numbers of the preceding loop to which this statement instance belongs to), while the statement instance that is nested by only the L_i loop will have a 3 element induction variables vector (i.e., $(0, i, 1)$). Listing 4.1 demonstrates an example code and the induction variables for each statement/basic block, before they are pre-processed (described later in Section 4.3.2) .

These values will later be used during the analysis phase to differentiate between nodes and tiles of different loop dimensions.

4.3.2 Preprocessing of Dynamic Dependency Graph

At the core of our approach lies the process of generating the dynamic dependence graph from the program execution. The target code segment is instrumented to generate the runtime information we need for the dependence analysis. Once a runtime execution trace is available for a program, we can start the construction of the dynamic data-dependence graph by creating a graph node for each dynamic instruction instance observed at runtime.

```
1  for(i=0; i<N; i++){
2    S1;                                    [0, i, 0, 0, 0]
3    for(j=0; j<N ; j++){
4      S2;                                  [0, i, 1, j, 0]
5    }
6    S3;                                    [0, i, 2, 0, 0]
7    for(k=0; k<N; k++) {
8      S4;                                  [0, i, 3, k, 0]
9    }
10   S5;                                    [0, i, 4, 0, 0]
11 }
```

Listing 4.2: Induction Variables After Pre-processing

Edges between every pair of nodes are then created whose instructions depend on each other. When a new node is encountered, we get the producer of the values read by this node to detect the read after write (raw) dependences. We also keep track of all the readers of a memory or register location. When a later node reads from or writes to such a location, a read after read (rar) or a write after read (war) dependence is detected, respectively. Edges are created for all these types of dependences. The result is a graph that encapsulates all data-dependences that actually occurred during execution. Since our implementation is based on LLVM, each graph node represents an LLVM IR instruction and dependences are tracked through memory and LLVM virtual registers).

After generating the DDG, we preprocess it for our implementation purpose. Right now, we are only concentrating on statements in a basic block level and so we do not differentiate between several statements in a single basic block. Therefore, different DDG nodes having the same induction variable (as they belong to the same basic block) are being merged.

Our goal is to handle both perfectly and imperfectly nested loops. In order to do that, we eliminate the difference between the instances of a perfectly nested statement and an

imperfectly nested statement. To consider all of them equal during the analysis phase described later, the induction variables are extended and filled with trailing 0 values for the imperfectly nested nodes to have a length of $2d+1$, where d is maximum loop depth. These nodes can be thought of as perfectly nested by all the loops, but with loops not surrounding them running from 0 to 0, implying no execution of that loop for this node/statement. Listing 4.2 shows the preprocessed version of the example shown earlier in Listing 4.1.

4.3.3 Convex-partitioning heuristic

Tiling for locality attempts to group points in an iteration space of a loop into smaller blocks (tiles) allowing reuse (thereby reducing reuse distance) in multiple directions when the block fits in a faster memory (registers, L1, or L2 cache). Forming a valid tiling for a loop requires that each tile can be executed atomically, i.e., each tile can start after performing required synchronizations for the data it needs, then execute all the iterations in the tile without requiring intervening synchronization. This means that there are no cyclic data dependences between any two tiles. *Our aim is to achieve T^d tiling without violating any data dependence*. The algorithm proceeds by repeating the following two steps unless all the nodes in the CDAG belong to some partition:

- Pick the next ready node from the ready list and try to build a tile of size T^d gathering all the required nodes.
- Add any additional nodes that need to be included to make this a valid convex partition.

Fig. 4.3 depicts this approach on a two dimensional iteration space ($d = 2$). If the value of T is 3, the first step will create a partition or tile that is covered by the blue rectangle in

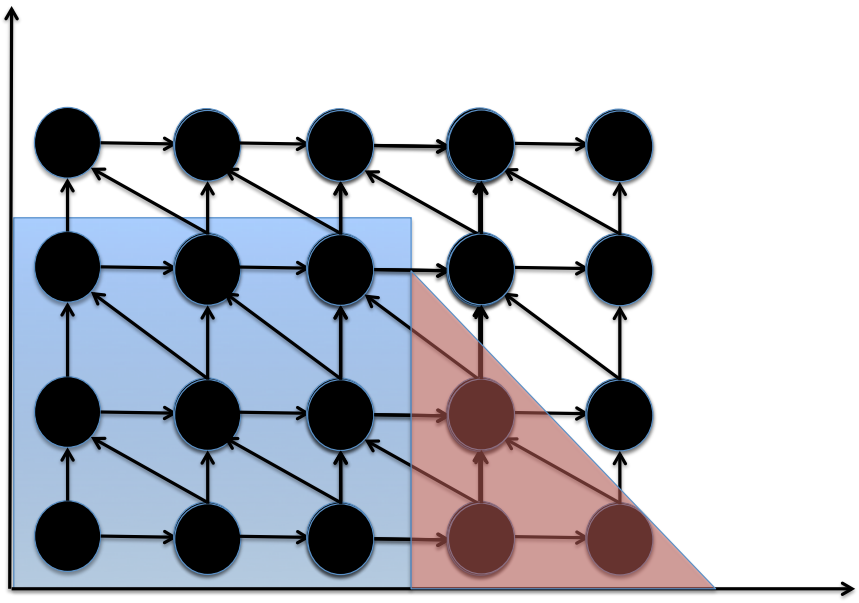


Figure 4.3: Convex Partitioning Approach for 2D Loop Iteration Space with $T = 3$

the figure. Then to preserve dependence and maintain the convexity property, we need to include all the nodes inside the pink triangle into the current growing partition. Therefore, the first partition will contain 12 nodes instead of just $3^2 = 9$ nodes.

We show in Algorithm 4.1 our technique to build convex partitions from an arbitrary CDAG. It implements a convex-partition growing heuristic that successively formulates a tile starting from a ready vertex. The key requirement in adding a new vertex to a convex partition is that if any path to that vertex exists from a vertex in the partition, then all vertices in that path must also be included. We first build the tile/partition and then add any other nodes necessary to hold the convexity property.

The inputs to the algorithm are the CDAG and the tile diameter T. The partitioning heuristic generates a valid schedule as it proceeds. At the beginning, all input vertices to the

Algorithm 4.1 GenerateConvexPartitions(G,T)

Input : G : CDAG; T: Tile diameter
InOut: P : Partition containing convex components
begin

```
P ⟵ ∅
R ⟵ getTheInitialReadyNodes(G)
while R ≠ ∅ do
    cp ⟵ ∅
    n ⟵ selectReadyNode(R)
    cp ⟵ cp ∪ {n}
    UpdateListOfReadyNodes(R, n)
    < i, j, k > ⟵ Induction variable vector for n
    for All vector IV in the range [< i, j, k >, < i + T − 1, j + T − 1, k + T − 1 >] do
        if IV is a valid induction variable then
            n ⟵ Node with induction variable IV
            if n is unprocessed then
                cp ⟵ cp ∪ {n}
                UpdateListOfReadyNodes(R, n)
    S ⟵ ∅
    for n ∈ cp do
        if Any predecessor p of n is unprocessed then
            S.push(p)
    while S ≠ ∅ do
        n ⟵ S.top()
        if n is unprocessed then
            cp ⟵ cp ∪ {n}
            S.pop()
            UpdateListOfReadyNodes(R, n)
            if Any predecessor p of n is unprocessed then
                S.push(p)
        else
            S.pop()
    P ⟵ P ∪ {cp}
```

CDAG are placed in a ready list R. A vertex is said to be *ready* if all its predecessors (if any) have already executed, i.e., have been assigned to some convex partition. A new convex partition cp is started by adding a ready vertex to it (the function `selectReadyNode`(R)

simply picks up one element of R). Now we want to create a partition that starts at this node and possibly is of size T^d. Therefore, we compute all the valid loop induction variable values that might belong to this new partition. For simplicity, we assumed $d = 3$ in the Algorithm 4.1 (The actual implementation can handle loops of any depth). The growing partition then adds any unprocessed node remaining that has loop induction variable value that falls within the computed range. Note that, the nodes may or may not be in the ready list R.

To ensure the convexity property, and produce a dependence preserving valid schedule at the end of the partitioning process, we need to make sure that there exists no such node p which does not belong to a partition yet (still unprocessed) but has a path to one or more of the nodes in the current growing partition. Therefore the partitioning algorithm starts from the current nodes in the partition and looks backwards as long as there is no unprocessed node remaining that is a direct or indirect predecessor to the nodes in the current tile. We use a stack S which is initially filled up with any unprocessed immediate predecessors of the current partition nodes. Then we loop through the stack one node at a time. Each loop iteration looks at the current stack top, includes it in the current tile if it is not already part of any tile and removes it from the stack. Any immediate predecessor of this node that is not part of any tile are pushed into S for future processing. The process ends when the stack becomes empty, which indicates we have no more unprocessed predecessor (immediate or transitive) or we have exhausted the predecessor lists. At this point of the algorithm, we have a new valid partition.

After the addition of any node to a partition, we need to update the ready list R. Suppose a vertex n is just added to a partition cp. As a result, zero or more of the successors of n in G may become ready: a successor s of n becomes ready if the last predecessor

needed to execute s is n. The addition of newly readied vertices to the ready list is done by the function `updateListOfReadyNodes`(*R, n*), as shown in Algorithm 3.2. In this function, the test that checks if s has unprocessed predecessors is implemented using a counter that is updated whenever a node is processed.

After such a partitioning, the execution order of the vertices is reordered so that the convex partitions are executed in some valid order (corresponding to a topological sort of a coarse-grained inter-partition dependence graph), with the vertices within a partition being executed in the same relative order as the original sequential execution. Details are presented in the next section.

4.4 Experimental Results

4.4.1 Experimental Setup

The dynamic analysis we have implemented involves three steps. For the CDAG Generation, we use automated LLVM-based instrumentation to generate the sequential execution trace of a program, which is then processed to generate the CDAG. The trace generator was previously developed for performing dynamic analysis to assess vectorization potential in applications [43]. For the convex partitioning of the CDAG, we have implemented the algorithms explained in detail in the previous section. Finally, the reuse distance analysis of the reordered address trace after convex partitioning is done using a parallel reuse distance analyzer PARDA [69] that was previously developed. All performance experiments were performed on an Intel Core i7 2700K, using a single core.

4.4.2 Results

In this section, we present experimental results of dynamic analysis on a number of benchmarks - Matrix-matrix multiplication, 2D Jacobi, the Floyd-Warshall algorithm to find all-pairs shortest paths in a graph, and two QR decomposition methods: the Givens rotation and the Householder transformation. None of these benchmarks could be fully tiled for enhanced data locality by state-of-the art research compilers (e.g., Pluto [76]) or by production compilers (e.g., Gnu GCC, Intel ICC). For each benchmark, we study the reuse distance profile of the original code and the code after convex partitioning. We compare the two convex-partitioning heuristics alongside the original code and also the tiled version whenever applicable. The figures 4.4- 4.8 plots the reuse distance profile in bytes/flop metric for different cache sizes. The lower the curve, the better the reuse profile and the lower I/O cost.

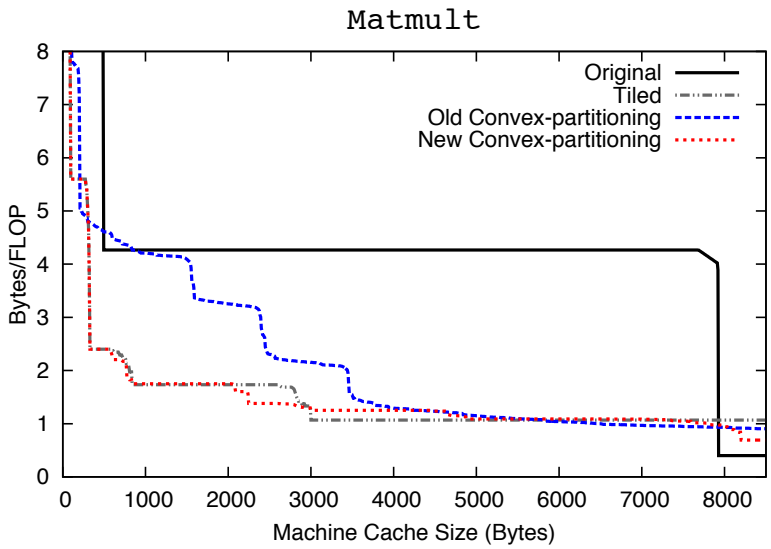


Figure 4.4: Matmult

Matmult Fig. 4.4 shows the experimental results for matrix-matrix multiplication for problem size 30×30. We can see that the new heuristic is better than the old one by a large margin. It is also almost overlapping with the reuse distance profile of the optimized tiled version. The new heuristic achieves more regular shaped tiles for Matmult and therefore the reuse distance profile is very similar to hand-tiled code of Matmult.

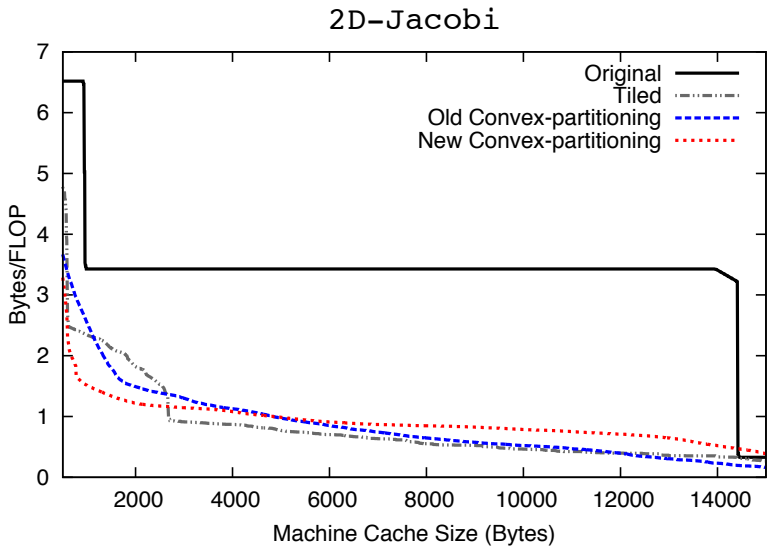


Figure 4.5: Jacobi 2D

Jacobi Fig. 4.5 shows the results on 2D Jacobi for problem size of 32×32 and for 30 iteration. The new heuristic performs well for lower cache sizes. After a certain cache size the performance is slightly worse but still remains comparable to the tiled code. Jacobi tiled code has tiles more of a parallelogram shape than rectangular shape. The new heuristic attempt close to rectangular shape tiles, which might be the reason behind the performance difference.

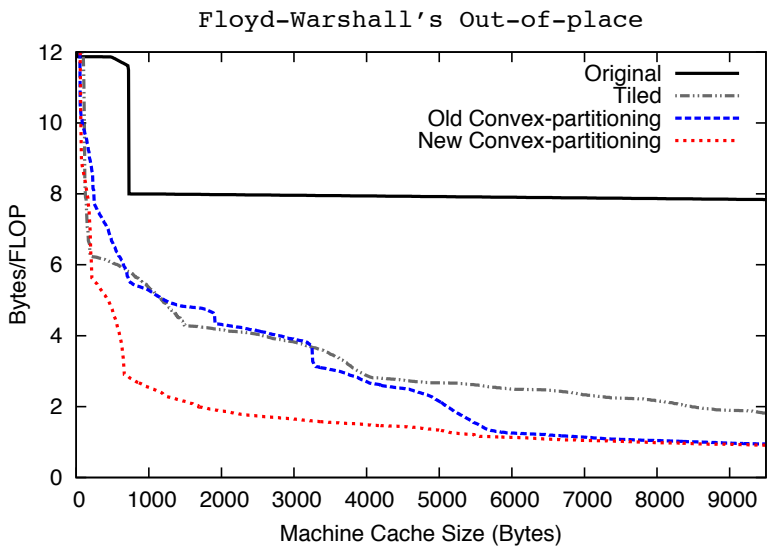


Figure 4.6: Floyd-Warshall all-pairs shortest path

Floyd-Warshall all-pairs shortest path Fig. 4.6 shows the results on Floyd-Warshall all-pairs shortest path for problem size of 30×30. We refer to this code as "out-of-place" Floyd-Warshall because it uses a temporary array to implement the all-pairs shortest path computation. The new heuristic shows potential for improvement as the previous heuristics. The performance for the new heuristic is better than the tiled version. On further analysis, we found that the tile shapes are still somewhat irregular with the new heuristic. Although the heuristic attempts to find regular shaped tiles, due to the complex dependence structure for this algorithm, the final dependence preserving partitions end up being irregularly shaped. The partitions we achieve have better reuse than the manually tiled version.

Givens Rotation and Householder For the two QR decomposition methods, we find similar results from both of the convex partitioning heuristics. Fig. 4.7 shows the results

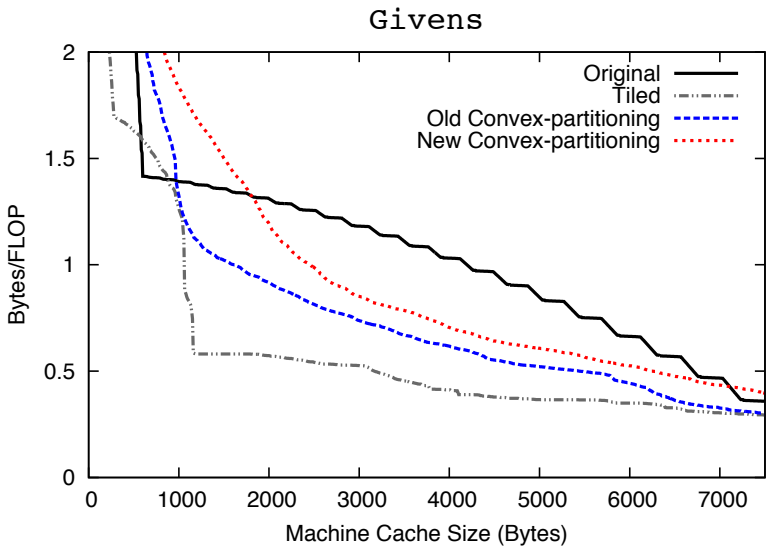


Figure 4.7: Givens Rotation

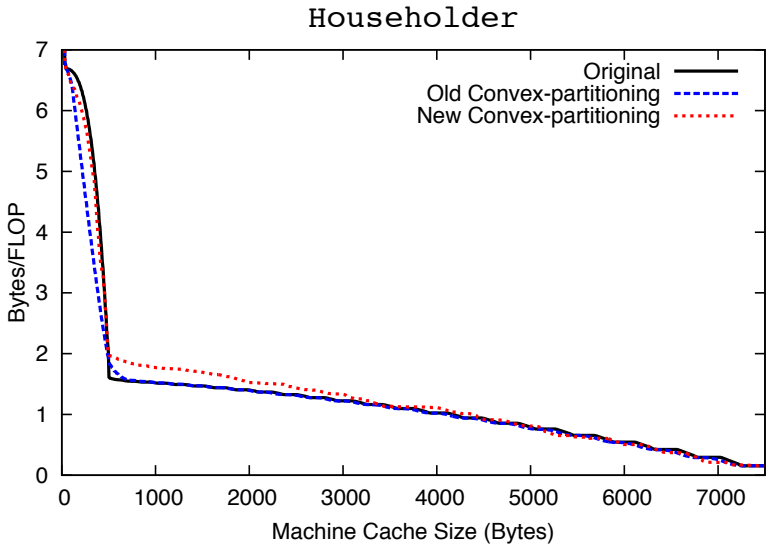


Figure 4.8: Householder

for Givens rotation. Although the new partitioning heuristic performs slightly worse than the previous one, it still shows scope of improvement from tiling, while householder shows no potential. Therefore, we can reach similar conclusions from both of the heuristics about the data locality potential for these two methods - Givens rotation algorithm may be better suited for deployment on future hardware, because of its lower bandwidth demand than Householder, especially for small cache sizes.

4.5 Conclusion

In this piece of work, we have developed a new and improved convex partitioning algorithm that uses loop induction variable information for our existing dynamic analysis tool for providing insights about the inherent data locality property of sequential programs. We aim to get similar results from the new heuristic that is independent of parameters that depend on program characteristics. We construct a dynamic computational directed acyclic graph (CDAG) to capture the statement instances and their inter-dependences, where each node has the loop induction variable associated with the statement instance; preprocess the CDAG to merge nodes that belongs to the same loop iteration; perform convex partitioning of the CDAG to generate a modified, dependence-preserving, execution order with better expected data reuse; and then perform reuse distance analysis on the trace corresponding to the modified execution order. The partitioning initially attempts to build regular shaped tiles of size T^d, where T is the tile diameter and d is the maximum loop depth, then increases the tile size by adding any necessary node which is required to be executed for preserving dependences and the convexity property, which might end up modifying the tile

shape. We demonstrated the effectiveness on a number of benchmarks, where we get similar or better results using the new heuristics. We conclude that the results obtained from this new more general convex partitioning heuristic are consistent with the previous heuristic.

CHAPTER 5

Characterizing and Enhancing Global Memory Data Coalescing on GPUs

5.1 Introduction

While the computing capability of GPU is far ahead than CPU nowadays, the memory bandwidth and latency still remains the bottleneck. To make the best out of heterogeneous systems, we need to understand and exploit the parallelism in the memory hierarchy. But writing parallel applications for heterogeneous environments that efficiently use the complex memory hierarchy is still not very intuitive to many programmers. Although the highly parallel environment of GPUs makes it possible to run applications much faster than CPUs, inefficient memory usage can significantly impact the overall performance gain. Due to its large capacity and programming convenience, the GPU global memory is used the most among all the memory levels available, but it is also the slowest among them all. With advancement in GPU technology, the off-chip global memory is now cached, but still slow. The slow dynamic random access memories (DRAMs) that are used to build them is the primary reason behind this performance bottleneck. Therefore, optimal usage of the global memory and taking advantage of the coalescing capability whenever possible is crucial.

There are existing tools that attempt to ease the development of GPU applications [10–12, 39, 56–58, 99]. But existing CUDA applications still suffer from uncoalesced accesses. Most often it is strenuous to manually find out the inefficiency in the complex computations, specially for long, arbitrary codes. Unless the programmer is able to detect the problem, other optimization tools depending on programmer input (e.g., [23,40,93]) are of little help. If uncoalesced access is detected automatically, the programmer can then seek to transform the code. Thus, there is a strong need for tools to assist application developers develop codes that exhibit a high fraction of coalesced accesses.

In this chapter, we present our combined approach of dynamic analysis and static transformations to overcome the above mentioned limitations. Dynamic analysis has the power to look beyond affine codes. As it relies on actual program execution, we can handle non-affine data dependent codes. When dynamic analysis on traces generated from the program detects uncoalesced accesses, some recommendations are made depending on the overall memory access pattern. In many cases, static transformations can then be applied to convert the uncoalesced access to coalesced access. Our dynamic analysis tool is designed for analyzing arbitrary CUDA/PTX codes for identifying loops with uncoalesced accesses, and categorization of uncoalesced accesses to different groups along with suggestions for potential improvement strategies. We also provide a static transformation framework that implements a remapping of work among threads to optimize CUDA/PTX codes exhibiting uncoalesced global memory access. To demonstrate the effectiveness of our tool, we have characterized GPU benchmark suites using the dynamic analysis and transformed a number of them, including irregular applications. Our transformed version ensures coalesced access and yields better performance.

The rest of the chapter is organized as follows. Section 5.2 contains background information on GPU global memory access and the PTX intermediate representation. Section 5.3 elaborates on the design and implementation of our dynamic analysis, and Section 5.4 presents our static transformation approach. Section 5.5 presents an experimental evaluation of our approach. Finally, Section 5.6 discusses related work, followed by the conclusion in Section 5.7.

5.2 Background and Overview

5.2.1 GPU Architecture

Computation GPUs are designed for high computational throughput. GPUs typically contain hundreds of cores (streaming processors) arranged in tightly coupled groups of 8-32 scalar processors per streaming multi-processor (SMs). Parallel threads are grouped into thread blocks that are scheduled on a SM and cannot migrate. Threads are spawned in 1-, 2-, or 3-dimensional rectangular groups of cooperative threads, called blocks (CUDA) or work-groups (OpenCL). A 1-, 2- or 3-dimensional grid of blocks is used to schedule the thread blocks. Both the size and number of thread blocks are fixed when launching a GPU kernel and cannot be changed after the threads have launched. We note $\vec{G} = (bdim_x, bdim_y, bdim_z, tdim_x, tdim_y, tdim_z)$ the geometry of the thread space: the sizes of a thread block in each dimension are denoted $tdim_x$, $tdim_y$ and $tdim_z$. In the case of a 2D or 1D geometry for a thread block, we simply set $tdim_z$ or $tdim_z$ and $tdim_y$ to 1. The grid of thread blocks also has a 3D geometry, the dimension in each dimension is typically computed from the total number of threads (e.g., problem size) divided by the thread block size in the dimension. A thread in the computation is uniquely identified by

$\vec{t} = (b_x, b_y, b_z, t_x, t_y, t_z)$ a vector of 6 integers, where each component can range between 0 and the size in $\vec{G}$ from the corresponding component.

Memory The GPU memory hierarchy consists of global memory (shared across thread blocks), shared memory (shared only among the threads in a single block), local memory and registers. Global memory is the largest in terms of size, but also the slowest. Shared memory is faster than global memory but limited in size. In modern GPUs, each Streaming Multiprocessor (SM) has 64KB of fast memory that can be partitioned between L1 cache for global memory and shared memory. The 64KB space can be either divided into two 32KB partitions, or 48KB and 16KB. Global memory coalescing (described in Sec. 5.2.2) leads to efficient usage of the available bandwidth between global memory and shared memory or L1 cache.

5.2.2 Global Memory Coalescing

When a kernel is launched on a GPU, it is executed by all the threads in parallel. A typical scenario is to have a global memory reference in the kernel that is executed by all threads, but requesting different memory addresses for each thread, as shown in Lst. 5.1.

```
1 __global__  void kernel(float* a) {
2   int tid = threadIdx.x;
3   a[tid] = 1.0;
4 }
```

Listing 5.1: CUDA code

These memory requests are grouped into a number of memory transactions by the GPU in the current scheduling unit for a thread block to maximize the bandwidth usage. That is, the memory transaction is computed based on the region of data requested by a set of active threads. When consecutive threads access consecutive global memory region (as in Lst. 5.1) then a single transaction may be implemented, and accesses are *coalesced*.

With modern GPUs (compute capability 1.2 or higher) consecutive threads are no longer required to get coalesced access, it is enough that the set of data accessed by the set of threads considered (e.g., threads having the same t_y, t_z but different t_x) is a contiguous chunk of memory [72].

When the data region accessed by threads is not contiguous (e.g., for an access `a[tid * N]` instead of `a[tid]` in Lst. 5.1, leading to a poor spatial data locality), then it is not possible anymore to pack the data request into a single, large transaction: the reference leads to *uncoalesced* accesses. Up to one transaction per thread will be needed, dramatically reducing the effective bandwidth. Accessing non-contiguous memory from the global memory incurs significant performance penalties [72]. One approach to possibly hide the effects of uncoalescing is to use the shared memory for caching the accesses (see Sec. 5.4.4), however this requires complex code restructuring. In this work, we first take the approach of changing the thread geometry (i.e., which threads will end up being grouped together at the time of issuing the memory transactions) to improve spatial data locality and global memory coalescing before resorting to shared memory usage.

5.2.3 Overview of the Framework

Fig. 5.1 depicts the overall steps of our approach, which consists of 4 stages: instrument, execute, analyze, and transformation.

We use Parallel Thread Execution (PTX), an intermediate language for kernels designed to run efficiently on NVIDIA GPUs [71]. High-level language compilers like nvcc [72] or clang [94] can generate PTX instructions from CUDA or OpenCL codes. First, the input PTX code is instrumented and executed on a GPU to generate its memory traces. We use Ocelot [36], a compiler framework for PTX code analysis on heterogeneous systems, to

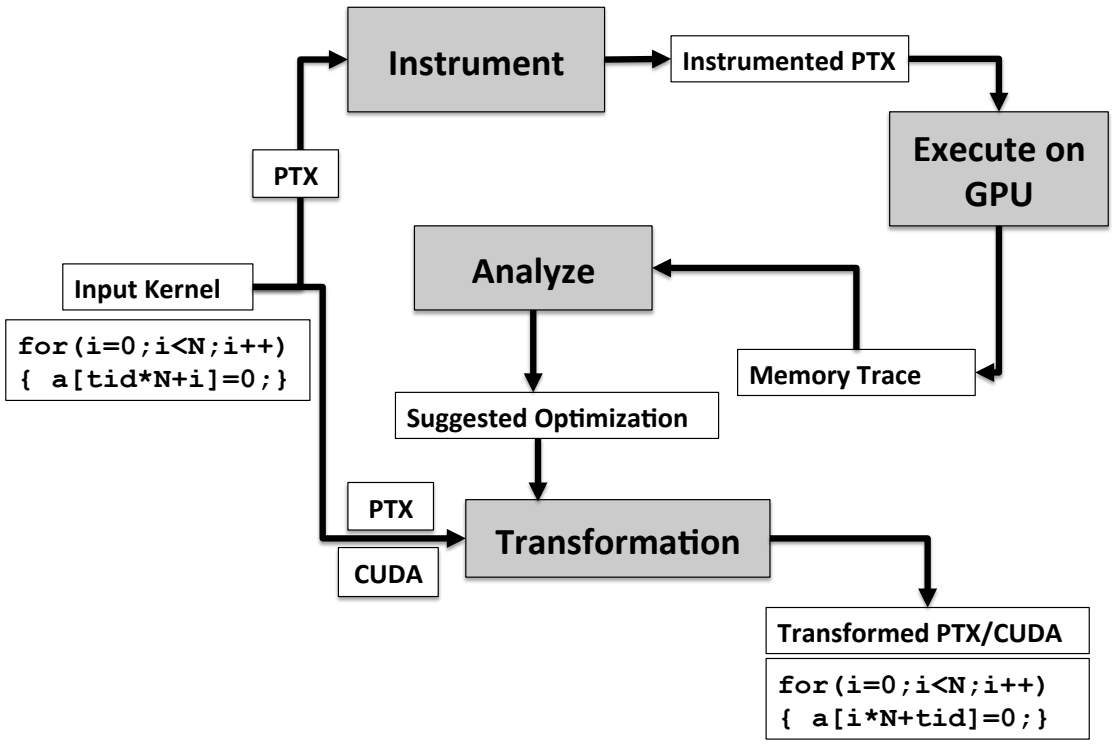


Figure 5.1: Overall Flow Chart of Our Approach

instrument PTX codes. Next, an analysis is performed on the memory traces to characterize coalesced and uncoalesced accessed, with recommendation of possible transformations based on the dynamic analysis results (Sec. 5.3). Finally, based on the recommendation, the code may be transformed. A small and always-applicable set of thread geometry permutations can be applied on arbitrary PTX programs (Sec. 5.4.2), and the dynamic analysis can be re-run on the resulting program to observe whether coalescing has improved on the working dataset. Alternatively, a more powerful geometry transformation framework but which operates only a subset of CUDA programs can be applied to maximize coalesced accesses (Sec. 5.4.3).

5.3 Dynamic Analysis of Uncoalesced Accesses

5.3.1 Instrumentation and Execution

The input to the dynamic analysis is the PTX code of a kernel we wish to investigate. First, the PTX kernel is instrumented by inserting a function call to a device function that stores all the necessary information about the memory access after each global memory access to compute and store a trace of the program. The memory trace of an instrumented PTX code is generated simply by executing it on the GPU. The trace can be dependent on the values of the input data, and so the results may vary with different input datasets. In such cases, it is recommended to perform the dynamic analysis with a variety of representative input datasets, determining such sets is out of the scope of this work. An example of a trace excerpt is shown in Listing 5.2:

```
1  #tx ty tz static_id Load(1)/Store(2) Address Dyn_id
2  0 0 0 31  1 30066082304 0
3  0 0 0 34  2 30066081792 0
4  0 0 0 31  1 30066082320 1
5  0 0 0 34  2 30066081796 1
6  0 0 0 31  1 30066082336 2
7  0 0 0 34  2 30066081800 2
8  0 1 0 31  1 30066082320 0
9  0 1 0 34  2 30066081796 0
10 0 1 0 31  1 30066082336 1
11 0 1 0 34  2 30066081800 1
12 0 1 0 31  1 30066082352 2
13 0 1 0 34  2 30066081804 2
```

Listing 5.2: Trace

where $(tid_x, tid_y, tid_z, static\_id, type, address, dynamic\_id)$ is the tuple forming a trace entry. $static\_id$ is the unique identifier of the memory instruction in the PTX code generating this trace entry. $dynamic\_id$ is the unique identifier of the *instance* of the static instruction, for a particular $(tid_x, tid_y, tid_z, static\_id)$ value. Dynamic ids correspond to different run-time instances of a single static instruction.

The tuple $(tid_x, tid_y, tid_z, static\_id, dynamic\_id)$ is necessarily unique in a trace. The actual memory address accessed by this instruction as well as the type of access (load or store)

is also captured. In this work, only trace entries with identical $(static\_id, dynamic\_id)$ values may execute in parallel and are candidate for coalescing analysis.

5.3.2 Dynamic Analysis Algorithm

Our analysis algorithm detailed in Algorithm 5.1 scans through the memory trace file to characterize memory access patterns and produce meaningful, per-instruction statistics about the access strides and the potential for coalescing. The algorithm takes as input W, the scheduled block size (e.g., $W = 16$ for a half warp for compute capability lower than 2.0).

The algorithm analyzes all memory traces from one single static instruction at a time. The memory traces corresponding to the same static id and dynamic id are stored in M to test for coalescing. The memory traces are sorted according to their thread ids lexicographic order and stored in V. The entries in V are then analyzed in groups of size W. These are the memory executions that happened during warp execution and need to be coalesced. We compute the maximum, minimum and average stride for the half warp entries (W consecutive entries from V). All the actual memory addresses accessed are also recorded in $AllAddrs$. This step is required to test that a consecutive portion of memory is actually accessed by the threads over time. Otherwise, no coalescing is possible. The algorithm then outputs the information about the instruction - memory access type, the strides computed, etc. If the maximum stride is less than the size of the data type (i.e., 4 for float, 8 for double etc.), then the access is coalesced. Otherwise, the access is uncoalesced and we report possible improvement strategies.

Algorithm 5.1 Memory Trace Analysis Algorithm

Input : T: Memory trace, viewed as a map T[$static\_id$][$dynamic\_id$][tid_y][tid_z][tid_x] = $addr$
W: Warp scheduling size
Output: Report on coalescing and possible optimization strategy
begin
  for *all unique* $static\_id$ *in* T **do**
    $(min\_stride, max\_stride, avg\_stride, allStr) \leftarrow (\infty, 0, 0, 0)$
    $AllAddrs \leftarrow$emptyVector()
    for *all unique* $dynamic\_id$ *in T[*$static\_id$*]* **do**
      // Take the trace entries corresponding to candidate
      // accesses for coalescing.
      $M \leftarrow$ T[$static\_id$][dyn_id][*] // Build the linearized list of memory addresses accessed.
      $i \leftarrow 0$
      for *all* (tid_y, tid_z, tid_x) *in* M *in lexicographic order* **do**
        $V[i] \leftarrow$ M[tid_y][tid_z][tid_x]
        $i \leftarrow i + 1$
      $c \leftarrow 0$
      while $c < i$ **do**
        // Sort W consecutive elements of V to compute the sorted
        // list of memory addresses accessed by W consecutive threads.
        $V[c..c + W] \leftarrow$ sortByIncreasingValue($V[c..c + W]$)
        for $j \in [0..W - 1]$ **do**
          $stride \leftarrow V[c + j + 1] - V[c + j]$
          $min\_stride \leftarrow min(min\_stride, stride)$
          $max\_stride \leftarrow max(max\_stride, stride)$
          $avg\_stride \leftarrow avg\_stride + stride$
        $avg\_stride \leftarrow avg\_stride/W$
        $c \leftarrow c + W$
      $AllAddrs \leftarrow concat(AllAddrs, V)$
    // Check if the entire memory space accessed is contiguous.
    $AllAddrs \leftarrow$ sortByIncreasingValue($AllAddrs$)
    for $i \in [0..AllAddrs.size - 1]$ **do**
      $allStr \leftarrow max(allStr, AllAddrs[i + 1] - AllAddrs[i])$
    // Produce report and suggestions.
    Print(static_id, load/store type)
    Print(min_stride,max_stride,avg_stride)
    if $max\_stride \leq sizeof(data\_type)$ **then**
      Print("coalesced")
    else
      Print("uncoalesced")
      if $allStr > sizeof(data\_type)$ **then**
        Print("accesses cannot be all coalesced")
      else
        Print("Suggest thread geometry transformations") **if** *instruction is a load* **then**
          Print("Suggest also shared memory usage")

5.4 Compiler Transforms for Data Coalescing

In this section we present a compiler framework to improve data coalescing. We first present an approach that uses our dynamic analysis to empirically drive a re-scheduling of the CUDA threads, that is a change of the thread block geometry in Sec. 5.4.2. This approach operates on arbitrary CUDA/PTX programs. We then present a purely compile-time approach that only requires very basic static analysis of the references in a CUDA/PTX program to compute a new thread block geometry aimed at reducing the number of uncoalesced accesses in Sec. 5.4.2. To address cases of uncoalesced accesses which require using loops from the thread code to formulate a new thread geometry, we focus on a subset of CUDA programs that can be handled with the polyhedral compilation framework [10] and discuss our method in Sec. 5.4.3. Finally, complementary transformations for load optimization is presented in Sec. 5.4.4.

5.4.1 Overview

Let us first denote by $\vec{G} = (b_x, b_y, b_z, t_x, t_y, t_z)$ the geometry of the thread space: the sizes of a thread block in the three dimensions are denoted t_x, t_y and t_z. In the case of a 2D or 1D geometry for a thread block, we simply set t_z or t_z and t_y to 1. The grid of thread blocks also has a 3D geometry, the size in each dimension is typically computed from the total number of threads (e.g., problem size) divided by the thread block size in the dimension. An uncoalesced access arises from non-consecutive data being accessed by two consecutive threads along the t_x dimension. A rescheduling of the threads is analogous to loop permutation: for instance to permute dimensions t_x and t_y, we (1) update the geometry to become $\vec{G} = (b_y, b_x, b_z, t_y, t_x, t_z)$; and (2) substitute each occurrence of `threadIdx.x` by `threadIdx.y` (and conversely) in the CUDA/PTX program.

Our dynamic analysis presented previously provides two key pieces of information to drive the profitability of a program transformation: which reference is uncoalesced (and its stride), and how often a reference is executed. To improve data coalescing, we seek a program transformation essentially based on finding a new geometry for the threads, such that accesses are consecutive in memory along the newly computed t_x dimension.

5.4.2 Computing a New Thread Block Geometry

This transformation stage takes two inputs: (1) the original geometry, and (2) the AST of the thread code. CUDA programs have the key property of allowing any bijective transformation of the thread geometry. That is, all inter-block and inter-thread dimensions are fully data-parallel and hence interchangeable without violating program semantics. We remark that at this stage we do not require any additional property of the code such as having affine control or data-flow: we simply exploit the parallelism readily available through the thread geometry, and seek an alternative geometry with improved coalescing of data accesses.

Empirical search using dynamic analysis

Given $\vec{G}$ the vector describing the geometry. An uncoalesced access on a reference R arises typically because for two consecutive values of t_x, the same reference accesses non-consecutive data. Our first approach seeks a permutation of thread block dimensions so that the fastest varying one does not incur non-unit stride accesses by R. An iterative approach to test all possibilities is effective and straightforward in this case: only 3 possibilities exist (one for each of the 3 original thread block dimensions when used as the t_x component of the geometry), they are all semantically correct, so one can implement the 3 alternatives and run the dynamic analysis on each of the 3 cases. The rest of the permutation, that is

which of the original thread block dimensions will be used as the new t_y and t_z dimensions, can be chosen arbitrarily.

Then, the dynamic analysis presented previously is run on each of the three cases, the result is inspected and the configuration providing the lowest number of uncoalesced accesses is retained. In our experiments, this simple approach successfully solved uncoalesced accesses for the benchmarks Gaussian Elimination and Cell. We remark that this approach implicitly assumes that the test data set used during dynamic analysis is representative of the typical control-flow for the program.

Model-driven geometry transformation

Another approach that does not rely on dynamic analysis is possible when the reference can be successfully characterized using standard static analysis. Contrary to the previous empirical approach, this model-driven framework requires a static analysis of all the references in the CUDA/PTX program to gather information for the cost model. The objective and constraints do not change: we are seeking a permutation of the geometry, and support arbitrary CUDA/PTX programs.

Given an array reference R `A[pos]`[1], where `pos` is the expression used to index the array, we first perform static analysis on `pos` (possibly inspecting the entire kernel code) to uncover key properties on the relationship between thread ids in each dimension and the value of `pos`. Precisely, we analyze each sub-expression involved in the computation of `pos`, so as to determine:

1. if `pos` is of the form `x + b`, where x is a thread id and b is invariant to x; otherwise `pos` is of the form `b`.

[1]We only discuss the case of linearized array references

2. If b above is of the form `y*c + d` where y is a thread id, and c is greater than 1.

3. The list of all thread ids used to compute `pos`.

In other words, we perform static analysis of the expression `pos` for the purpose of finding (1) which thread id, if any, occurs without any multiplier (it will therefore be suitable for coalesced accesses along this dimension); (2) which thread id, if any, occurs with a non-unit multiplying factor (it will not be suitable for coalesced accesses); and (3) which thread id is used to compute the reference (any thread id not involved will be suitable for stride-0 accesses). Standard dataflow analysis can be used, in particular computing reaching definitions for `pos` [3] and the read/write sets of these definitions.

We denote $\vec{c} = (c_x, c_y, c_z)$ a vector of 3 Booleans, such that if c_x is set to 1 then the thread dimension t_x matches x in the pattern $x + b$ above, 0 otherwise. Similarly for c_y when t_y matches x in the pattern $x + b$, etc. We denote $\vec{u} = (u_x, u_y, u_z)$ another vector of 3 Booleans, with the semantics that u_x is set to 1 if t_x matches y in the pattern $b = y * c + d$ above, and so on for the other coordinates. Finally we denote $\vec{z} = (z_x, z_y, z_z)$ the vector where z_x is set to 1 if t_x is not used to compute the value `pos`, and so on.

We are now equipped to formulate an Integer Linear Program whose optimization provides us with the thread dimension to use for t_x. To achieve this, we implement the solution in the form of a permutation matrix for the three thread block coordinates. We first introduce 9 Boolean variables $p_{i,j}$, one for each of the elements in a 3×3 permutation matrix which models the 3D thread geometry permutation we need to apply to maximize coalesced accesses. To ensure it is a true permutation matrix, we add the constraint $\sum_i p_{i,j} = 1$, one for each of the three values for j, and $\sum_j p_{i,j} = 1$, for each value of i, that is a total of six constraints. Then, for each reference R, we add constraints to capture the cost of large-stride accesses (that is, t_x would be true in $\vec{u}$), stride-1 accesses (t_x is true in $\vec{c}$) and stride-0

accesses (t_x is true in $\vec{z}$). We use equal cost of 1 for stride-0 and stride-1 accesses, and a fictitious large value N for the cost of large-stride accesses (N must be greater than the number of references). We get for a reference R:

$$C_R = \sum_{i=1}^{3} c(i).p_{1,i} + \sum_{i=1}^{3} z(i).p_{1,i} + \sum_{i=1}^{3} N.u(i).p_{1,i}$$

where only the cost associated to the dimension used as "inner-most" (the first row of the permutation matrix, capturing the output t_x dimension) is being modeled. The final optimization problem, where $p_{i,j}$ are Boolean unknowns, is then:

$$\begin{aligned} P &= \sum_{R} C_R \\ \text{minimize} \quad P &\quad \text{s.t.} \sum_{i} p_{i,j} = 1,\ \forall j \wedge \sum_{j} p_{i,j} = 1,\ \forall i \end{aligned}$$

Because of the constraints on P to output a permutation matrix, we are guaranteed to find a permutation that minimizes the number of large-stride references. If multiple solutions with identical cost exist, one may be selected randomly. Further refinement can be achieved by weighting each cost by the number of times the reference is accessed. The optimal solution is implemented as the geometry permutation encoded in the permutation matrix is applied to the CUDA/PTX program, altering both the thread block geometry and substituting thread ids in the code as necessary.

5.4.3 Geometry and Thread Code Transformations

Threads are grouped into the same warp according to their threadIdx.x. Therefore, these threads should read/write consecutive global memory locations as much as possible. Listing 5.3 shows a common example where each thread is assigned a complete row of a matrix, which leads to strided memory access of threads in the warp.

```
1 __global__
2 void kernel(float *A, float value){
3   int tx = threadIdx.x + blockIdx.x*blockDim.x;
4   for ( int i = 0; i < N; i++){
5 R:   A[tx*N + i] = value;
6   }
7 }
```

Listing 5.3: Inefficient Global Memory Store Example

Loop `i` is a parallel loop and therefore we can re-distribute the stores among the threads to ensure coalesced access. That is, we can use the `i` loop *in the thread code* as an additional source of parallel threads, and compute a new 2D geometry in place of the original 1D one, such that accesses will be coalesced for t_x: in this example this amounts to (1) making i the t_x dimension, updating b_x correspondingly to capture N threads; and (2) make the original t_x dimension t_y in the transformed code.

The example above highlights a key issue when transforming codes for coalescing: that *intra-thread loops may need to be analyzed and transformed into CUDA threads* to ensure proper coalescing without data layout transformations. Indeed, in our benchmark suite the Rodinia Myocyte and all PolyBench/GPU kernels are programmed using 1-dimensional thread block geometry, with parallel loops inside the kernel code. Such programs require deep static analysis and transformation of both the geometry and kernel codes to implement efficient data accesses.

To address this problem, we now present a polyhedral-based framework to transform CUDA programs by lifting loops inside the kernel code and producing a multi-dimensional thread geometry. Because of the need for precise analysis, contrary to the previous sections this framework applies only to a specific subset of CUDA kernels: those whose control- and data-flow can be exactly characterized at compile-time using the polyhedral model [10, 32], and which do not contain any synchronization primitives. We remark that this framework uses the CUDA kernel source code as input, in place of the PTX representation. Algorithm 5.2 outlines our polyhedral optimization flow, and is detailed in the following.

Algorithm 5.2 Polyhedral optimization flow

Input : AST: AST of the CUDA kernel code;
G: original thread geometry;
Output: AST, G: Output new CUDA code and geometry
begin
  $Poly \leftarrow$extractPolyhedralRepresentation(AST,G)
  $C \leftarrow$computeLegalityConstraints($Poly$)
  $C_p \leftarrow$computeParallelismConstraints($Poly$)
  $C \leftarrow C \cap C_p$
  $P \leftarrow$computeCostForAllRefs($Poly$)
  $sol \leftarrow$minimize P s.t. C
  $AST \leftarrow$polyhedralCodegen($Poly$,AST,sol)
  $AST, G \leftarrow$postProcessingAndTiling(AST)
  return AST, G

Program representation

To represent a CUDA program in the polyhedral model, we first resort to classical AST-to-polyhedron conversion (function `extractPolyhedralRepresentation`). First we compute, for each syntactic statement in the program, its iteration domain. This set captures all the statement run-time instances, with an integer set bounded by affine inequalities. We then integrate the thread geometry (both for thread block geometry in the grid and thread geometry within thread blocks) by viewing it as a loop nest with 6 loops, each iterating from 0 to b_x, etc. for the other 5. For instance for statement `R` above, its iteration domain $\mathcal{D}_S$ is:

$$\begin{aligned}\mathcal{D}_S \;=\; & \{(b_x, b_y, b_z, t_x, t_y, t_z, i) \in \mathbb{Z}^7 \\ & \mid 0 \le b_x < \text{N/blDim.x} \wedge 0 \le t_x < \text{blDim.x} \wedge \\ & \; 0 \le i < N \wedge b_y = b_z = t_y = t_z = 1\}\end{aligned}$$

Access functions describe the location of the data accessed by a statement instance. In static control parts, memory accesses are performed through array references (a scalar

variable being a particular zero dimensional case of an array). We restrict ourselves to subscripts that are affine expressions of surrounding loop counters and global parameters. For instance, the subscript function of a read reference `A[i][k]` surrounded by 3 loops i, j and k is simply $f_A(i, j, k) = (i, k)$.

The execution order of the dynamic instances of statements captured in the iteration sets is described using a scheduling function Θ^{S_i} for each statement S_i. A schedule is a function which associates a logical execution date (a timestamp) to each instance of a given statement. In the case of multidimensional schedules, this timestamp is a vector. In the target program, statement instances will be executed according to the increasing lexicographic order of their timestamp. To construct a full program description, we build a collection of schedules $\Theta = \{\Theta^{S1}, \ldots, \Theta^{Sn}\}$, that is a list of the statement scheduling function for each statement in the program.

In this work, we restrict the scheduling function to model only the original program order and any possible permutations of the loop dimensions, including intra-thread loops and geometry dimensions. In other words, we seek a permutation matrix which will consider all six geometry dimensions as well as any loop dimension in the code. For homogeneity if two statements are not surrounded by the same number of loops, artificial one-time loops are introduced in the representation before the statement's inner loop, so that all statement iteration domains and schedules have the same dimensionality for the program.

Computing the program transformation

Our objective to find a transformation of the program is extremely analogous to Section 5.4.2: we formulate an ILP integrating the cost of accesses for each reference, seeking a solution in the form of a permutation matrix. But we also need to now take into account

(1) legality conditions, as not all permutation of intra-thread loops may be semantically correct; and (2) parallelism conditions, as the intra-thread dimensions can be permuted with geometry dimensions only if they are parallel loops. Finally, to generate correct CUDA codes we must comply with the maximal thread block sizes as specified by the GPU and resort to a complementary tiling phase if needed.

The function `computeLegalityConstraints` computes the legality conditions that constrain the coefficients of the permutation matrix. The convex set of semantics-preserving schedules can be built in the traditional manner, linearizing the constraints provided by each dependence polyhedron using the Farkas Lemma [32, 79] to end up with affine inequalities bounding the schedule coefficients (i.e., the permutation matrix) so that each solution satisfying the inequalities necessarily preserves the program semantics. The Ponos tool automatically computes these constraints from the polyhedral representation [1, 79]. The sets of statement instances between which there is a data dependence relationship are modeled as equalities and inequalities describing a *dependence polyhedron* [31]. All dependence polyhedra can be automatically extracted from the program polyhedral representation, for instance using the Candl tool [1].

The function `computeParallelismConstraints` encodes a system of constraints on the schedule coefficients so that for two instances $\vec{x}_R$ and $\vec{x}_S$ in dependence, the constraint $\Theta_R(\vec{x}_R) = \Theta_S(\vec{x}_S)$ is enforced for all and only geometry dimensions. Indeed, for the rest of the program, only the semantics-preserving condition $\Theta_R(\vec{x}_R) \preceq \Theta_S(\vec{x}_S)$ is enforced, however for all dimensions/loop levels. This condition is the classical sync-free parallelism condition [10, 60] to be enforced at the scheduling level, and is also linearized using the Farkas Lemma [32]. The final set of constraints C is computed as the intersection of the legality and parallelism constraints.

The function `computeCostForAllRefs` first analyzes each reference to compute the various $\vec{c}$, $\vec{u}$ and $\vec{z}$ vectors based on the reference access functions. We remark that these vectors are extended to have one element per surrounding dimension (that is, the geometry dimensions and any surrounding loop, including one-time loops), instead of only 3. We then formulate an ILP to find the coefficients of a permutation matrix (the linear part of the scheduling functions), in a manner analogous to Sec. 5.4.2. The permutation matrix size is adapted to properly model all dimensions, and we encode the cost constraints not only for the t_x dimension (i.e., $p_{4,j}$ the coefficients of the permutation matrix corresponding to the output t_x dimension) but also for all intra-thread loops (i.e., $p_{6+k,j}$, for each $k \in [1..d]$ with d intra-thread loops). We then minimize this ILP to obtain sol, the coefficients of the permutation matrix subject to all constraints.

The function `polyhedralCodegen` embeds the permutation sol into a complete schedule Θ and implements the transformation on the polyhedral representation of the CUDA code, by generating a code scanning iteration domains following the order specified by the schedule [13]. As we have modeled the geometry dimensions as standard loops in the polyhedral representation, a final post-processing stage needs to be performed to translate the geometry loops into CUDA geometry syntax, the function `postProcessingAndTiling` performs this task. Finally, because of hardware GPU constraints, we may need to apply tiling along the thread block dimensions to ensure the new t_x, t_y and t_z do not exceed their maximal allowed size. Returning to Listing 5.3, $(b_x, b_y, b_z, t_x, t_y, t_z, i)$ is permuted to $(b_z, b_x, b_y, i, t_x, t_y, t_z)$, where a one-time loop t_z is used inside the thread code (in other words, after code generation there is no loop inside the thread code), and loop i becomes the new t_x dimension. Because i has N iterations, a strip-mining of i is

performed, updating b_z and i, so that the new dimension t_x has a size not exceeding the hardware geometry constraints, with $b_x \times t_x = N$.

5.4.4 Other Static Transformations

Listing 5.4 describes a common inefficient access pattern where a single thread is used to compute a reduction operation on a row of a matrix.

```
1 __global__
2 void kernel(float *A, float *x, float *tmp){
3  int i = blockIdx.x*blockDim.x+threadIdx.x;
4  if (i < N){
5   for(j=0; j < N; j++){
6    tmp[i] += A[i * N + j] * x[j];
7   }}}
```

Listing 5.4: Inefficient Global Memory Load Example

This straightforward implementation leads to poor performance, because of uncoalesced accesses. We propose an effective work re-distribution strategy to improve the performance of such kernels.

During the dynamic analysis phase, such inefficiency is detected when we find a thread performs two contiguous loads followed by a store to a single location, but only one of the load operations is uncoalesced. It is highly likely, but not guaranteed that the thread is performing a reduction operation. To ensure correctness, we find and analyze the loads and the store in the original unoptimized PTX during the static transformation phase. The reduction operation is detectable at this point as it appears as a sequence of ld.global (data load from global memory), ld.global, fma (fused multiplication and add) and st.global (data store in global memory) operations in the PTX code. The optimization pass then proceeds with the transformation.

The transformation converts the reduction operation into a series of intermediate partial result computations, followed by atomic operations. The overall idea is depicted in Fig. 5.2.

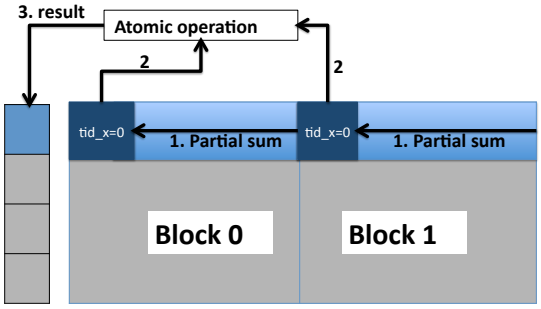


Figure 5.2: Partial Sum Method for Load Optimization

The matrix is logically divided into blocks, where each block is loaded in a coalesced fashion by a thread block into a shared memory buffer. The block size needs to be a perfect square (e.g., 256) so that T is an integer. Another run-time requirement is the use of N^2/T^2 blocks, so that each thread just loads one element from the NxN matrix. We allocate a buffer in the shared memory to hold data from global memory, where the size of the buffer is equal to the thread block size. The multiplication operation is performed on these data and the results are stored back in the shared buffer, replacing the the old values. After syncthreads, a subgroup of threads with ids $a * T + 0$, where $a = 0, 1, ...T - 1$ computes the partial sum of row a of the shared memory. Following another syncthreads, we need to perform an atomic addition across the blocks to get the final result from the partial results.

The transformation steps are shown in Algorithm 5.3. The declaration of the 2-D shared array is inserted in the first basic block. Lines 4-10 are the computations inserted and stored in registers. For simplicity, we skipped using PTX level computations in the algorithm. From the $inst_{ID}$, we find the array base addresses of the two ld.global and the st.global instructions that surround the fma. The original PTX instructions are moved out of the loop that contained them. We replace them with new instructions. Then the uncoalesced

Algorithm 5.3 Transform Consecutive Load Operations by Single Thread

Input : $inst_{ID}$: Static ID of the instruction to transform;
T: Shared Memory dimension TxT;
C: Input PTX code;
Output: C': Output PTX code
begin
$C' \longleftarrow C$
Allocate shared memory $S[TxT]$ in C'
$i \longleftarrow threadID/T$
$j \longleftarrow threadID\%T$
$t_1 \longleftarrow blockID * BLOCK\_SIZE$
$t_2 \longleftarrow (blockID * T)\%N$
$index1 \longleftarrow t_1 + t_2 + i * N + j$
$index2 \longleftarrow (blockID * T + j)\%N$
$index3 \longleftarrow (blockID * T * T)/N + i$
$load_inst \longleftarrow$PTX instruction for $inst_{ID}$
$A \longleftarrow$ Base address of $load\_inst$
$x \longleftarrow$ Base address of the next load instruction
$y \longleftarrow$ Base address of the store instruction
Find loop containing $load\_inst$
Move reduction instructions out of loop
`// Coalesced read from global memory`
Load $A[index1]$ into $S[i][j]$
$S[i][j] \longleftarrow S[i][j] * x[index2]$
Syncthreads
if $j = 0$ **then**
`// Compute partial sum for block`
for $k = 0$ *to* T **do**
$sum \longleftarrow sum + S[i][k]$
Syncthreads
Insert atomic addition of sum in C'
Store result of atomic add at $y[index3]$
return C'

load instruction is transformed into a coalesced load by using the new computed offset $index1$. The ld.global is now followed by a st.shared (store in shared location). The offset of the second load operation is also replaced by $index2$. The multiplication is inserted after the second load and the result is stored back in the shared buffer. Following a syncthreads

instruction, we add a code block that checks whether the thread has an ID that equals $a * T + 0$ for some a. Inside the true branch, we insert a loop to perform the partial sums. After the if block, we add another syncthreads to make sure that all the partial results are ready. Finally, the atomicadd instruction is inserted to add the partial sums and stores the results at the new offset $index3$.

5.5 Experimental Results

5.5.1 Experimental Protocol

We have used Ocelot v2.1 to build the instrumentation and the optimization passes for PTX codes. CUDA Toolkit v5.5 was used for the nvcc compiler and also to write the CUDA driver API for executing PTX codes. Experiments were done on machines with Tesla K20 and Tesla K10 - both had compute capability of 3.0. The benchmarks presented are from Rodinia v2.4 [20,22] and PolyBench/GPU v1.0 [38]. The sparse matrix-vector multiplication (SpMV) application is taken from SHOC [26]. The spmv_csr_kernel from SHOC has uncoalesced access of the column vector that stores indices of the actual elements in the matrix. This access is affine and we were able to apply the transformation. The execution time is measured by taking the average of 100 executions.

5.5.2 Dynamic Analysis Results

Type	C/U	Stride	Bandwidth
Load	Coalesced	0	1
Load	Uncoalesced	N	N
Store	Uncoalesced	N	N

Table 5.1. Sample Output of Dynamic Analysis

Rodinia			
Benchmark	Kernel	Total	Uncoalesced
Gaussian	Fan1	$2N + 1$	$2N$
	Fan2	$4N^2 + 4N$	$3N^2 + N$
Kmeans	invert_mapping	$2NK$	NK
MUMmerGPU	printKernel	$5N$	$5N$
Myocyte	solver	$7N^2$	$7N^2$
k-NN	euclid	$5N$	$4N$
Cell	evolve	$2N^3$	$2N^3$
StreamCluster	compute_cost	$6N$	$5N$
Polybench/GPU			
Benchmark	Kernel	Total	Uncoalesced
AtAx	atax_kernel1	$N^2 + 3N$	N^2
BiCG	bicg_kernel2	$N^2 + 3N$	N^2
Correlation	corr_kernel	$7N^2 + N$	$5N^2$
covariance	covar_kernel	$8N^2$	$5N^2$
GESUMMV	gsumv_kernel	$2N^2 + 8N$	$2N^2$
Gramschmidt	gram_kernel2	$2N + 1$	$2N$
mvt	mvt_kernel1	$N^2 + 3N$	N^2
SYRK	syrk_kernel	$6N^2$	N^2
SYR2K	syr2k_kernel	$8N^2$	$2N^2$

Table 5.2. Benchmarks with Uncoalesced Access in Rodinia and Polybench/GPU

To evaluate our dynamic analysis tool, we characterized the global memory coalescing properties of Rodinia and PolyBench/GPU benchmarks. The time for the dynamic analysis depends on the size of the memory traces. The total time for the instrumentation, trace generation and running the analysis of an application is generally in the order of hundreds of milliseconds. The profile sizes are approximately in the range of ten kilobytes.

We feed one kernel at at time to the analysis tool and produce useful information for each of the static global memory load/store operation. For each operation the tool is designed to report the following: type (Load/Store), Coalesced/Uncoalesced, stride (distance between memory accessed by consecutive thread in the thread space) and the bandwidth of the operation. For example, for the Fan1 kernel of the Gaussian Elimination benchmark, the tool produces output like Table 5.1. Table 5.2 depicts a summary of the results

Benchmark	Kernel Execution Time		Data Copy Times		GFLOPS		**Kernel Speedup**	App. Speedup
	Original	Transformed	Host to Device	Device to Host	Original	Transformed		
(R) Myocyte	79.81ms	2.25ms	43.81ms	29.25ms	1.8	63.5	35.5×	2.1×
(R) Gaussian	1.91s	0.43s	0.01s	0.01s	17.91	78.71	4.4×	4.3×
(R) Cell	6.73ms	1.74ms	1.37ms	1.55ms	n/a	n/a	3.9×	2.1×
(P) covariance	23.27s	4.61s	0.011s	0.01s	5.91	29.82	5.1 ×	5.1×
(P) GESUMMV	82.67ms	10.59ms	20.63ms	0.01ms	3.25	25.35	7.8×	3.3×
(P) AtAx	28.71ms	15.28 ms	40.87ms	0.02ms	9.35	17.56	1.9×	1.3×
(P) Correlation	23.27s	4.61s	0.02s	0.01s	5.91	29.82	5.1 ×	5.1×
(P) mvt	23.61ms	10.19ms	40.89ms	0.02ms	11.37	26.33	2.3×	1.3×
(P) BiCG	28.86ms	15.33ms	40.88ms	0.02ms	9.28	17.51	1.9×	1.2×
SpMV	27.49ms	8.01ms	10.44ms	0.01ms	5.22	17.23	3.4×	2.1×

Table 5.3. Execution times of applications on Tesla K20

of the analysis on Rodinia and Polybench/GPU. For space limitation, we only report the benchmark kernels that has uncoalesced access detected by the tool. Seven Rodinia benchmarks are identified to have uncoalesced accesses. Three out of these seven benchmarks (*MUMmerGPU*, *k-Nearest Neighbor* and *Stream Cluster*) use array of structures where each thread accesses all the members of a structure element. This results in a strided access for the threads in a warp. Using existing APIs [23], the array of structures can be converted to structure of arrays to achieve coalesced accesses. While our static transformation framework could be extended in the future to incorporate such an optimization, it currently does not. Hence, we do not report performance for these applications. *Myocyte*, on the other hand, assigns each thread to compute a full row of a matrix which is very inefficient. *Gaussian elimination* distributes a 2D matrix onto a 2D thread block but assigns the fastest growing dimension to threadIdy, leading into strided access. *Cell* has the same issue for 3D matrices and thread blocks. We optimize these three benchmarks using static transformation (results in the next section).

The PolyBench/GPU suite has nine applications with uncoalesced memory accesses. Among them, the *Gramschmidt* kernel2 requires re-writing of the whole application to achieve coalesced access - which is out of scope for this work. *Syrk* and *Syr2k* do not have

much scope of improvement. We transform the remaining six applications. We also analyzed SHOC sparse matrix-vector (SpMV) multiplication as an representative for irregular applications. The results are in the following section.

5.5.3 Static Transformation Results

From the dynamic analysis tool, the applications identified with improvement potential are then fed to a static transformation framework for optimization. We report the improvement of execution times for these applications for Tesla K20 in Table 5.3. Benchmarks taken from Rodinia are marked with R while benchmarks taken from Polybench/GPU are marked with P. For each kernel, we also measured the time for copying data between host and device. We compared the performance in GFLOPS. Cell only performs copy operation, therefore GFLOPS is not reported for this benchmark. The effective bandwidth comparison for K10 and K20 machines are reported in Fig. 5.3 and Fig. 5.4 respectively. Note that y-axis is in logarithmic scale for these figures.

We observe significant speed up for all of the applications. The speedup ranges from $2\times$ to $35\times$ on K20. and $6\times$ to $40\times$ on K10. Myocyte in its original form suffers from a high memory latency due to the high amount (refer to Table 5.2) of uncoalesced accesses. Our transformation successfully re-distributed the workload so that threads now perform the same number of reads and writes in a coalesced fashion, leading to a significant $35\times$ speedup. Similarly, simple geometric transformation of thread dimensions in *Gaussian Elimination* and *Cell* improves their performances by around $4\times$. Although it seems that Cell had more uncoalesced access, it uses a 3D thread block on O(N^3) data vs Gaussian Elimination using 2D thread block on O(N^2) data. Therefore the speedup is similar. The

Polybench/GPU benchmarks and SpMV achieved $2\times$ to $8\times$ speedup after the transformation, which fixed the uncoalesced load operations. As the speed up is related to the amount of uncoalesced access in the original code, GESUMMV has higher speedup compared to AtAx, BiCG, etc due to its $2\times$ more uncoalesced accesses.

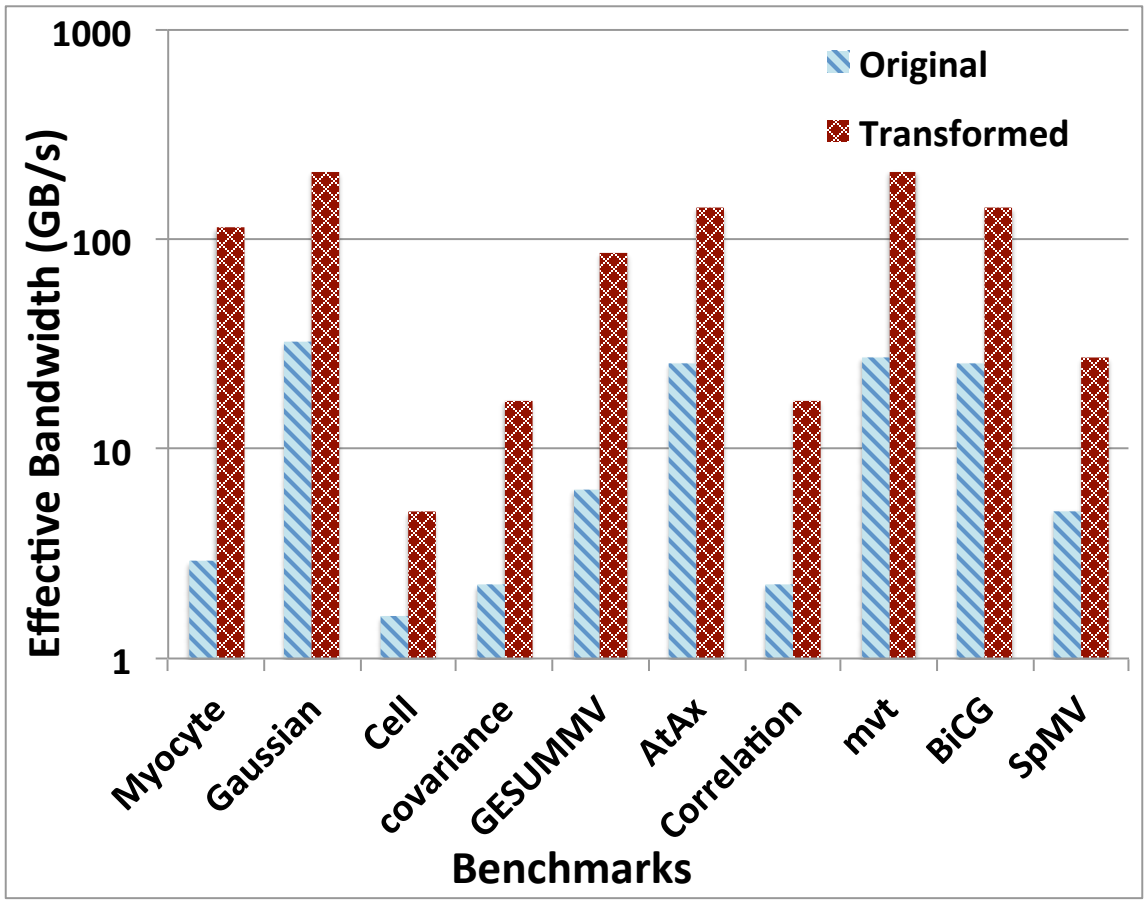


Figure 5.3: Effective bandwidth on Tesla K10. Y-axis is in logarithmic scale

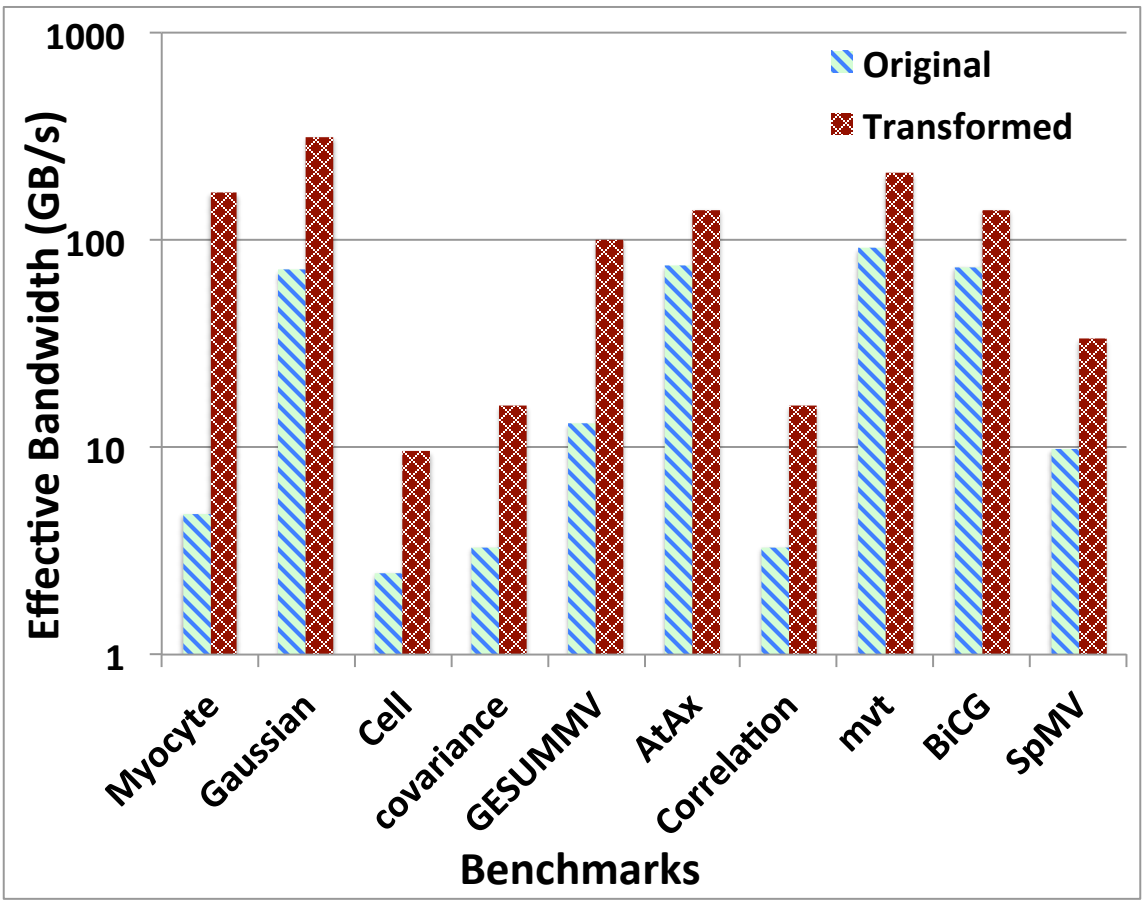


Figure 5.4: Effective bandwidth on Tesla K20. Y-axis is in logarithmic scale

Fig. 5.3 and 5.4 provides useful insights into the benefits of our transformation algorithms. Five out of the ten benchmarks reached over 100GB/s bandwidth after our optimization, and the high memory bandwidth directly contributes to overall application performance, because the time for fetching data from global memory is significantly reduced by coalesced accesses. Memory bounded kernels benefit more from our algorithms than compute bounded ones. K20 has a wider memory bus than K10, which results into lower

execution times and copying times. It is for the same reason we presume that the performance gain is more on K10. The uncoalesced kernels are penalized more on K10 compared to K20. The average speedup is $6\times$ for all of the applications tested.

2D Benchmarks	Thread Geometry	Kernel Execution	Program Execution
backprop	Y-X (X innermost)	0.63ms	0.081s
	X-Y	0.91ms	0.14s
hotspot	Y-X (X innermost)	0.38ms	0.39s
	X-Y	0.51ms	0.45s
lud	Y-X (X innermost)	57.6ms	68.6ms
	X-Y	188.8ms	199.7ms
srad	Y-X (X innermost)	8.28ms	0.25s
	X-Y	27.06ms	0.31s
gaussian	Y-X (X innermost)	3.97s	4.06s
	X-Y	0.83s	0.93s
3D Benchmark	Thread Geometry	Kernel Execution	Program Execution
Cell	Z-Y-X(X innermost)	3.35ms	0.47s
	X-Y-Z(Z innermost)	10.53ms	1.19s
	X-Z-Y(Y innermost)	4.33ms	0.57s

Table 5.4. Execution times using different thread block geometry on Rodinia

We tested the importance and effect of thread block geometry permutation on the Rodinia benchmark suite. We picked the Rodinia benchmarks that uses 2D or 3D thread block, chose all possible permutation of the thread block geometry and measured the execution times. The comparison is shown in Table 5.4. It is clear from the table that the best performance always comes from choosing threaIdx.x as the innermost dimension. Our static analysis method was able to detect the best permutation for all of these benchmarks. For Gaussian and Cell, the benchmarks had less than optimal permutation which the static analysis corrected by choosing the correct permutation and then performing the transformation

according to the findings. For the others, the analysis correctly detected that the existing permutation is the best and performed no transformation.

5.5.4 Discussions

Dynamic analysis, in general, can be inaccurate in some cases as the analysis might be dependent on input data sets. But the effectiveness of our specific characterizing tool does not depends on choosing appropriate input sets. To the best of our knowledge, we are not aware of any real benchmark on which different input datasets can make the same memory reference changing from being coalesced to uncoalesced, or vice-versa. The input dataset may only change the run-time uncoalesced access count/stride/bandwidth usage metric. But, ignoring corner/artificial cases, it will not change the fact whether a memory reference is coalesced or not. However, the final performance of the transformed code can be affected, of course, by the input dataset. In our test suite, for 6 out of 10 benchmarks the control flow is independent of the input, therefore the same transformation is always required whatever the input, and such transformation is automatically computed and implemented in our framework.

Our dynamic analysis for detecting uncoalesced access operates on arbitrary CUDA/PTX codes. Our transformation scheme based on geometry permutation can also handle any CUDA/PTX program. Therefore, we can analyze all Rodinia benchmarks and apply the geometric permutation on them. However our intra- and inter-thread optimization framework using the polyhedral model is limited to affine CUDA programs, and cannot handle all Rodinia benchmarks.

Coalesced access may increase the register pressure and result into higher amount of spill. But we observe significant speedup by achieving coalesced access (as shown in Table 5.3). Therefore, the effects of other factors, if any, must have been negligible.

5.6 Related Work

Many previous works focused on improving the programming productivity by automate transformation tools such as from C to CUDA [10, 11] and OpenMP to CUDA [56–58]. PPCG [99] uses polyhedral analysis and convert legacy affine sequential C codes to CUDA automatically. In contrast, our work can operate on arbitrary input CUDA/PTX codes and takes as input a CUDA program, and our intra-thread optimization focuses exclusively on data coalescing. Par4All [5] transforms C or Fortran code to CUDA or OpenCL code. Par4All uses a polyhedral analysis tool called PIPS but the tool itself is not entirely based on polyhedral analysis. Unlike PPCG, it does not use any shared memory. Optimization techniques such as loop collapsing or thread coarsening are used in [62, 106], they however differ from coalescing-centric approach to find a new thread block geometry. Few works have been proposed for specific algorithms such as [27, 59, 66, 70]. Inspector/executor based strategies [97, 104] have been proposed to support non-affine irregular codes. Another set of work provides directive-based CUDA code optimizations [40, 93] or API to transform CUDA codes [23] but rely on manual annotations from the programmer.

CUPL [4] uses polyhedral methods to detect possible uncoalesced accesses of affine CUDA codes. In contrast, our dynamic analysis method can detect coalesced or uncoalesced access pattern in any affine and irregular PTX codes. To the best of our knowledge, CUPL limits to detecting uncoalesced accesses, and does not automatically transform programs. Therefore CUPL cannot lead to any automatic improvement in performance,

in contrast to our approach which automatically transforms programs. GMRace [109], GRace [108] and GMProf [107] was developed to detect data races in shared memory, using a similar approach to ours that combines dynamic analysis and static transformation. Previous dynamic analysis techniques ([17, 21]) also focuses on program correctness and aims to detect race conditions and bank conflicts. To the best of our knowledge, we present the first dynamic analysis approach for improving the global memory access pattern on GPU. In addition, our framework targets at PTX code that can be derived from any heterogeneous programming language or directives.

5.7 Conclusion

In this work, we have combined dynamic analysis approach with static transformation with an aim to improve locality of global memory data during a thread warp execution. Given a PTX code of a program, we (1) characterize its global memory access operations and separate coalesced and uncoalesced access, (2) study access pattern of the uncoalesced accesses and recommend some improvement strategy if possible and (3) implement transformations of the input PTX (or CUDA) code to improve data coalescing. We have characterized GPU benchmark suites using the dynamic analysis and transformed a number of them, including irregular applications. Our transformed version ensures coalesced access and improves the kernel computation time by $2\times$ to $40\times$.

CHAPTER 6

Conclusion

In this dissertation, we have presented an approach to addressing a very important problem for both CPU and GPU architectures. Our tool for characterizing CPU applications looks beyond the limitation of reuse distance analysis and considers possible valid reordering of execution. We have developed a dynamic analysis approach to provide insights about the inherent data locality property of algorithm implementations. Given an execution trace from a sequential program, we seek to characterize the data locality properties of an algorithm and determine if there exists potential for enhancement of data locality through execution reordering. We first explicitly construct a dynamic computational directed acyclic graph (CDAG) to capture the statement instances and their inter-dependences. We then perform convex partitioning of the CDAG to generate a modified, dependence-preserving execution order with better expected data reuse. By performing reuse distance analysis on the trace corresponding to the modified execution order, we expect to get a better characterization of the potential benefit of reordering. We have demonstrated the utility of the approach in characterizing/enhancing data locality for a number of benchmarks. We also present another tool to characterize and enhance global memory coalescing in GPU applications. We have combined dynamic analysis with static transformation, with an aim to improve locality of global memory data during a thread warp execution. Given the PTX

code of a program, we characterize its global memory access operations and separate coalesced and uncoalesced access, study access pattern of the uncoalesced accesses and recommend an improvement strategy if possible and implement transformations of the input PTX (or CUDA) code to improve data coalescing. We have characterized GPU benchmark suites using the dynamic analysis and transformed a number of them, including irregular applications. Our transformed versions achieve significant improvement over the original versions.

BIBLIOGRAPHY

[1] the polyhedral compiler collection. *http://www.cs.ucla.edu/ pouchet/software/pocc/*.

[2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[4] AMILKANTHWAR, M., AND BALACHANDRAN, S. Cupl: A compile-time uncoalesced memory access pattern locator for cuda. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, ACM, pp. 459–460.

[5] AMINI, M., GOUBIER, O., GUELTON, S., MCMAHON, J. O., XAVIER PASQUIER, F., PAN, G., AND VILLALON, P. Par4all: From convex array regions to heterogeneous computing. http://www.par4all.org/.

[6] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[7] AUSTIN, T., AND SOHI, G. Dynamic dependency analysis of ordinary programs. In *ISCA* (1992), pp. 342–351.

[8] BALLARD, G., DEMMEL, J., HOLTZ, O., AND SCHWARTZ, O. Graph expansion and communication costs of fast matrix multiplication: regular submission. In *Proc. SPAA* (New York, NY, USA, 2011), ACM, pp. 1–12.

[9] BALLARD, G., DEMMEL, J., HOLTZ, O., AND SCHWARTZ, O. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications 32*, 3 (2011), 866–901.

[10] BASKARAN, M. M., BONDHUGULA, U., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPOPP* (2008), pp. 1–10.

[11] BASKARAN, M. M., BONDHUGULA, U., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS* (2008), pp. 225–234.

[12] BASKARAN, M. M., RAMANUJAM, J., AND SADAYAPPAN, P. Automatic c-to-cuda code generation for affine programs. In *CC* (2010), pp. 244–263.

[13] BASTOUL, C. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)* (Juan-les-Pins, France, Sept. 2004), pp. 7–16.

[14] BERGMAN, K., BORKAR, S., ET AL. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA IPTO, Tech. Rep* (2008).

[15] BILARDI, G., AND PESERICO, E. A characterization of temporal locality and its portability across memory hierarchies. *Proc. ICALP* (2001), 128–139.

[16] BONDHUGULA, U., HARTONO, A., RAMANUJAN, J., AND SADAYAPPAN, P. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. PLDI* (2008).

[17] BOYER, M., SKADRON, K., AND WEIMER, W. Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems* (2008).

[18] BRIDGES, M., VACHHARAJANI, N., ZHANG, Y., JABLIN, T., AND AUGUST, D. Revisiting the sequential programming model for multi-core. In *MICRO* (2007), pp. 69–84.

[19] CASCAVAL, C., DUESTERWALD, E., SWEENEY, P. F., AND WISNIEWSKI, R. W. Multiple page size modeling and optimization. In *Proc. PACT* (2005), IEEE Computer Society.

[20] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), pp. 44–54.

[21] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., AND SKADRON, K. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput. 68*, 10 (Oct. 2008), 1370–1380.

[22] CHE, S., SHEAFFER, J., BOYER, M., SZAFARYN, L., WANG, L., AND SKADRON, K. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on* (Dec 2010), pp. 1–11.

[23] CHE, S., SHEAFFER, J. W., AND SKADRON, K. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, pp. 13:1–13:11.

[24] COOPER, K. D., SIMPSON, L. T., AND VICK, C. A. Operator strength reduction. *ACM Trans. Program. Lang. Syst. 23*, 5 (Sept. 2001), 603–625.

[25] CUI, H., YI, Q., XUE, J., WANG, L., YANG, Y., AND FENG, X. A highly parallel reuse distance analysis algorithm on GPUs. In *2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)* (2012), IEEE, pp. 1080–1092.

[26] DANALIS, A., MARIN, G., MCCURDY, C., MEREDITH, J. S., ROTH, P. C., SPAFFORD, K., TIPPARAJU, V., AND VETTER, J. S. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (New York, NY, USA, 2010), GPGPU '10, pp. 63–74.

[27] DATTA, K., MURPHY, M., VOLKOV, V., WILLIAMS, S., CARTER, J., OLIKER, L., PATTERSON, D., SHALF, J., AND YELICK, K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for* (Nov 2008), pp. 1–12.

[28] DEMMEL, J., GRIGORI, L., HOEMMEN, M., AND LANGOU, J. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing 34*, 1 (2012), 206–239.

[29] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. In *PLDI* (2003), ACM, pp. 245–257.

[30] FAUZIA, N., ELANGO, V., RAVISHANKAR, M., POUCHET, L.-N., RAMANUJAM, J., RASTELLO, F., ROUNTEV, A., AND SADAYAPPAN, P. Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential. Tech. Rep. OSU-CISRC-9/13-TR19, Ohio State University, Sept. 2013.

[31] FEAUTRIER, P. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming 20*, 1 (Feb. 1991), 23–53.

[32] FEAUTRIER, P. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming 21*, 6 (Dec. 1992), 389–420.

[33] FISCHER, C. N., AND LEBLANC, JR., R. J. *Crafting a Compiler*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1988.

[34] FULLER, S. H., AND MILLETT, L. I. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011.

[35] GARCIA, S., JEON, D., LOUIE, C. M., AND TAYLOR, M. B. Kremlin: Rethinking and rebooting gprof for the multicore age. In *PLDI* (2011), pp. 458–469.

[36] GEORGIA INSTITUTE OF TECHNOLOGY. GPUOcelot. https://code.google.com/p/gpuocelot.

[37] GERLEK, M. P., STOLTZ, E., AND WOLFE, M. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17*, 1 (1995), 85–122.

[38] GRAUER-GRAY, S., XU, L., SEARLES, R., AYALASOMAYAJULA, S., AND CAVAZOS, J. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012* (May 2012), pp. 1–10.

[39] GROSSER, T., COHEN, A., KELLY, P. H. J., RAMANUJAM, J., SADAYAPPAN, P., AND VERDOOLAEGE, S. Split tiling for gpus: Automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (New York, NY, USA, 2013), GPGPU-6, ACM, pp. 24–31.

[40] HAN, T., AND ABDELRAHMAN, T. hicuda: High-level gpgpu programming. *Parallel and Distributed Systems, IEEE Transactions on 22*, 1 (Jan 2011), 78–90.

[41] HENNESSY, J., AND PATTERSON, D. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.

[42] HENNING, J. L. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News* (2006).

[43] HOLEWINSKI, J., RAMAMURTHI, R., RAVISHANKAR, M., FAUZIA, N., POUCHET, L.-N., ROUNTEV, A., AND SADAYAPPAN, P. Dynamic trace-based analysis of vectorization potential of applications. In *Proc. PLDI* (New York, NY, USA, 2012), ACM, pp. 371–382.

[44] IRIGOIN, F., AND TRIOLET, R. Supernode partitioning. In *Proc. POPL* (1988), pp. 319–329.

[45] JIANG, S., AND ZHANG, X. Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance. *IEEE Trans. Comput. 54* (August 2005), 939–952.

[46] JIANG, Y., ZHANG, E. Z., TIAN, K., AND SHEN, X. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proc. Comp. Const.* (2010), pp. 264–282.

[47] KENNEDY, K., AND ALLEN, J. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann, 2002.

[48] KENNEDY, K., AND MCKINLEY, K. S. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing* (1993), Springer-Verlag, pp. 301–320.

[49] KETTERLIN, A., AND CLAUSS, P. Prediction and trace compression of data access addresses through nested loop recognition. In *Proc. CGO* (2008), pp. 94–103.

[50] KETTERLIN, A., AND CLAUSS, P. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proc. MICRO* (2012).

[51] KOLTE, P., AND WOLFE, M. Elimination of redundant array subscript range checks. In *ACM SIGPLAN Notices* (1995), vol. 30, ACM, pp. 270–278.

[52] KUMAR, M. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers 37*, 9 (Sept. 1988), 1088–1098.

[53] LAM, M., AND WILSON, R. Limits of control flow on parallelism. In *ISCA* (1992), pp. 46–57.

[54] LAPACK. http://www.netlib.org/lapack.

[55] LARUS, J. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems 4*, 1 (July 1993), 812–826.

[56] LEE, S., AND EIGENMANN, R. Openmpc: Extended openmp programming and tuning for gpus. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for* (Nov 2010), pp. 1–11.

[57] LEE, S., AND EIGENMANN, R. Openmpc: Extended openmp for efficient programming and tuning on gpus. *Int. J. Computational Science and Engineering 7*, 1 (2012), 116.

[58] LEE, S., MIN, S.-J., AND EIGENMANN, R. Openmp to gpgpu: A compiler framework for automatic translation and optimization. *SIGPLAN Not. 44*, 4 (Feb. 2009), 101–110.

[59] LI, Y., DONGARRA, J., AND TOMOV, S. A note on auto-tuning gemm for gpus. In *Proceedings of the 9th International Conference on Computational Science: Part I* (2009), ICCS '09, pp. 884–892.

[60] LIM, A. W., AND LAM, M. S. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1997), ACM, pp. 201–214.

[61] LIU, S.-M., LO, R., AND CHOW, F. Loop induction variable canonicalization in parallelizing compilers. In *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on* (1996), IEEE, pp. 228–237.

[62] MAGNI, A., DUBACH, C., AND O'BOYLE, M. F. P. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 11:1–11:11.

[63] MAK, J., AND MYCROFT, A. Limits of parallelism using dynamic dependency graphs. In *WODA* (2009), pp. 42–48.

[64] MARIN, G., AND MELLOR-CRUMMEY, J. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS '04/Performance '04* (2004), ACM.

[65] MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78–117.

[66] NATH, R., TOMOV, S., DONG, T. T., AND DONGARRA, J. Optimizing symmetric dense matrix-vector multiplication on gpus. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), SC '11, pp. 6:1–6:10.

[67] NICOLAU, A., AND FISHER, J. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers 33*, 11 (1984), 968–976.

[68] NIU, Q., DINAN, J., LU, Q., AND SADAYAPPAN, P. Parda: A fast parallel reuse distance analysis algorithm. In *2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)* (2012), IEEE, pp. 1284–1294.

[69] NIU, Q., DINAN, J., LU, Q., AND SADAYAPPAN, P. Parda: A fast parallel reuse distance analysis algorithm. In *Proc. IPDPS* (may 2012), pp. 1284 –1294.

[70] NUKADA, A., AND MATSUOKA, S. Auto-tuning 3-d fft library for cuda gpus. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on* (Nov 2009), pp. 1–10.

[71] NVIDIA CORPORATION. *Parallel Thread Execution ISA*.

[72] NVIDIA CORPORATION. *NVIDIA CUDA C Programming Guide*, June 2011.

[73] OANCEA, C., AND MYCROFT, A. Set-congruence dynamic analysis for thread-level speculation (TLS). In *LCPC* (2008), pp. 156–171.

[74] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Communications of the ACM 29*, 12 (1986), 1184–1201.

[75] PARK, J.-S., PENNER, M., AND PRASANNA, V. K. Optimizing Graph Algorithms for Improved Cache Performance. *IEEE Transactions on Parallel Distributed Systems 15*, 9 (2004), 769–782.

[76] Pluto: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[77] POHL, T. 470.lbm. http://www.spec.org/cpu2006/Docs/470.lbm.html.

[78] POSTIFF, M., GREENE, D., TYSON, G., AND MUDGE, T. The limits of instruction level parallelism in SPEC95 applications. *SIGARCH Computer Architecture News 27*, 1 (1999), 31–34.

[79] POUCHET, L.-N., BONDHUGULA, U., BASTOUL, C., COHEN, A., RAMANUJAM, J., SADAYAPPAN, P., AND VASILACHE, N. Loop transformations: Convexity, pruning and optimization. In *POPL* (Austin, TX, Jan. 2011), pp. 549–562.

[80] RAU, B. R., AND FISHER, J. A. Instruction-level parallel processing: History, overview, and perspective. *J. Supercomput. 7*, 1-2 (May 1993), 9–50.

[81] RAUCHWERGER, L., DUBEY, P., AND NAIR, R. Measuring limits of parallelism and characterizing its vulnerability to resource constraints. In *MICRO* (1993), pp. 105–117.

[82] RAUCHWERGER, L., AND PADUA, D. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI* (1995), pp. 218–232.

[83] RUTTENBERG, J., GAO, G. R., STOUTCHININ, A., AND LICHTENSTEIN, W. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. *SIGPLAN Not. 31*, 5 (May 1996), 1–11.

[84] RUTTENBERG, J., GAO, G. R., STOUTCHININ, A., AND LICHTENSTEIN, W. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1996), PLDI '96, ACM, pp. 1–11.

[85] SARKAR, V., AND HENNESSY, J. L. Compile-time partitioning and scheduling of parallel programs. In *SIGPLAN Symposium on Compiler Construction* (1986), pp. 17–26.

[86] SHALF, J., DOSANJH, S., AND MORRISON, J. Exascale computing technology challenges. *High Performance Computing for Computational Science–VECPAR 2010* (2011), 1–25.

[87] SHEN, X., ZHONG, Y., AND DING, C. Locality phase prediction. In *Proc. ASPLOS* (2004), ACM.

[88] STEFANOVIĆ, D., AND MARTONOSI, M. Limits and graph structure of available instruction-level parallelism. In *Euro-Par* (2000), pp. 1018–1022.

[89] THEOBALD, K., GAO, G., AND HENDREN, L. On the limits of program parallelism and its smoothability. In *MICRO* (1992), pp. 10–19.

[90] TIAN, C., FENG, M., NAGARAJAN, V., AND GUPTA, R. Copy or discard execution model for speculative parallelization on multicores. In *MICRO* (2008), pp. 330–341.

[91] TOURNAVITIS, G., WANG, Z., ZHENG, FRANKE, B., AND O'BOYLE, M. Towards a holistic approach to auto-parallelization. In *PLDI* (2009), pp. 177–187.

[92] TU, P., AND PADUA, D. Automatic array privatization. In *Languages and Compilers for Parallel Computing*. Springer, 1994, pp. 500–521.

[93] UENG, S.-Z., LATHARA, M., BAGHSORKHI, S. S., AND HWU, W.-M. W. Languages and compilers for parallel computing. Berlin, Heidelberg, 2008, ch. CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15.

[94] UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN. Clang. http://clang.llvm.org.

[95] UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN. DragonEgg. http://dragonegg.llvm.org.

[96] UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN. Low-Level Virtual Machine. http://www.llvm.org.

[97] VENKAT, A., SHANTHARAM, M., HALL, M., AND STROUT, M. M. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2014), CGO '14, pp. 185:185–185:194.

[98] VENKATARAMAN, G., SAHNI, S., AND MUKHOPADHYAYA, S. A Blocked All-Pairs Shortest-Paths Algorithm. *Journal of Experimental Algorithmics 8* (Dec. 2003), 2.2.

[99] VERDOOLAEGE, S., CARLOS JUEGA, J., COHEN, A., IGNACIO GÓMEZ, J., TENLLADO, C., AND CATTHOOR, F. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim. 9*, 4 (Jan. 2013), 54:1–54:23.

[100] WALL, D. Limits of instruction-level parallelism. In *ASPLOS* (1991), pp. 176–188.

[101] WOLF, M. E., AND LAM, M. S. A data locality optimizing algorithm. In *PLDI '91: ACM SIGPLAN 1991 conference on Programming language design and implementation* (New York, NY, USA, 1991), ACM Press, pp. 30–44.

[102] WOLFE, M. Beyond induction variables. In *ACM SIGPLAN Notices* (1992), vol. 27, ACM, pp. 162–174.

[103] WOLFE, M. J. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[104] WU, B., ZHAO, Z., ZHANG, E. Z., JIANG, Y., AND SHEN, X. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013), PPoPP '13, pp. 57–68.

[105] WU, P., KEJARIWAL, A., AND CAŞCAVAL, C. Compiler-driven dependence profiling to guide program parallelization. In *LCPC* (2008), pp. 232–248.

[106] YANG, Y., XIANG, P., KONG, J., AND ZHOU, H. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, pp. 86–97.

[107] ZHENG, M., RAVI, V., MA, W., QIN, F., AND AGRAWAL, G. Gmprof: A low-overhead, fine-grained profiling approach for gpu programs. In *High Performance Computing (HiPC), 2012 19th International Conference on* (Dec 2012), pp. 1–10.

[108] ZHENG, M., RAVI, V. T., QIN, F., AND AGRAWAL, G. Grace: A low-overhead mechanism for detecting data races in gpu programs. *SIGPLAN Not. 46*, 8 (Feb. 2011), 135–146.

[109] ZHENG, M., RAVI, V. T., QIN, F., AND AGRAWAL, G. Gmrace: Detecting data races in GPU programs via a low-overhead scheme. *IEEE Trans. Parallel Distrib. Syst. 25*, 1 (2014), 104–115.

[110] ZHONG, H., MEHRARA, M., LIEBERMAN, S., AND MAHLKE, S. Uncovering hidden loop level parallelism in sequential applications. In *HPCA* (2008), pp. 290–301.

[111] ZHONG, Y., DROPSHO, S. G., AND DING, C. Miss rate prediction across all program inputs. In *Proc. PACT* (2003).

[112] ZHONG, Y., ORLOVICH, M., SHEN, X., AND DING, C. Array regrouping and structure splitting using whole-program reference affinity. In *Proc. PLDI* (2004), ACM.