A FRAMEWORK FOR GROUP LOCALITY AWARE MULTITHREADING

by

Sunil Shrestha

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Fall 2015

© 2015 Sunil Shrestha
All Rights Reserved
A FRAMEWORK FOR GROUP LOCALITY AWARE
MULTITHREADING

by

Sunil Shrestha

Approved: ________________________________________________________________
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: ________________________________________________________________
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: ________________________________________________________________
Ann L. Ardis, Ph.D.
Interim Vice Provost for Graduate and Professional Education
I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed:

Guang R. Gao, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed:

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed:

Stephan Bohacek, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed:

Andres Marquez, Ph.D.
Member of dissertation committee
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Guang R. Gao for his guidance throughout my time at the University. When I passed my Ph.D. Qualifier Exam in 2009, he told me “Now you are under my wings”, and I always felt I was. I enjoyed every bit of research discussions, motivational speeches and interesting history lessons.

I learned a lot from my friends and colleagues at CAPSL and PNNL. I would like to thank you all for sharing your knowledge and expertise with me. Special thanks to Andres Marquez and Joseph Manzano for all the fun meetings, discussions, support and guidance.

I would also like to thank ECE administration staffs Kathleen Forwood, Gwen Looby, Wendy Scott and Debbie Nelson for being patient and helping me with all the administration work.

Additionally, my awesome friends that were always there for me: Pabindra Rai, Tony Tran, Joseph Manzano, Juergen Ributzka, Eunjung Park, Ivanka Zhuo Li, Stephane Zuckerman, Ryan Taylor, Chen Chen and all the Mario Kart participants. There are many good memories that have led to who I am today. Thank you all for helping me keep my sanity throughout my Ph.D. years. Lastly, I would like to thank my family in United States, Georgette and Joe Metro, and my family and loved ones back home without whom this journey would not be possible.
To my loving parents
TABLE OF CONTENTS

LIST OF TABLES .................................................. x
LIST OF FIGURES .................................................. xi
ABSTRACT .......................................................... xvi

Chapter

1 INTRODUCTION .................................................. 1
  1.1 Early Years .................................................. 1
    1.1.1 Memory Hierarchy ...................................... 2
      1.1.1.1 Registers ........................................... 3
      1.1.1.2 Caches ............................................. 3
      1.1.1.3 DRAM ............................................... 7
    1.1.2 Further Improving Cache Locality ..................... 8
      1.1.2.1 Prefetchers ......................................... 8
      1.1.2.2 Loop Transformation ................................ 9
      1.1.2.3 Data Transformation ................................ 10
    1.1.3 Paradigm Shift to Multicore and the Impact .......... 11
  1.2 Multicore Era ............................................... 12
    1.2.1 On Chip Network Topology ............................... 13
      1.2.1.1 Shared Memory ..................................... 14
    1.2.2 Multicore Caches ....................................... 15
    1.2.3 Continuing Architectural Trends and Issues .......... 17
  1.3 Problem Statement .......................................... 19

vi
1.4 Contributions .................................. 21

2 BACKGROUND .................................. 23

2.1 Basics of Polyhedral Theory ....................... 23

2.1.1 Affine Function .................................. 23
2.1.2 Affine Space .................................... 24
2.1.3 Affine Subspace .................................. 25
2.1.4 Affine Hyperplane ................................. 26
2.1.5 Iteration Space Polytope .......................... 26
2.1.6 Affine Transformation ............................. 28
2.1.7 Classical Tiling .................................. 29
2.1.8 Legality for Tiling ................................. 29
2.1.9 Scheduling Tiled Code ............................ 31
2.1.10 Tiling for Concurrent Start ..................... 32
2.1.11 Condition for Concurrent Start .................. 33

2.2 Memory Layout and Access Function ................. 34

3 GROUP LOCALITY: THE FRAMEWORK ................. 37

3.1 Notion of Groups .................................. 38
3.2 The Framework .................................... 38

3.2.1 Code Generation ................................. 39
3.2.2 Hyperplane Communication Analysis ............... 41

3.2.2.1 Extracting Intra-tile Parallelism ............... 42
3.2.2.2 Scheduling Hierarchical Tiles .................. 45

3.2.3 Fine-Grain Execution ............................. 46
3.2.4 Data Restructuring and Reuse ..................... 48

3.2.4.1 The Approach .................................. 50

3.3 Conclusion ........................................ 54

4 JAGGED TILING FOR PARALLELISM ................. 55

4.1 Overview ......................................... 55
4.2 Jagged Tiling for Pipeline Start Applications ........................................... 57
  4.2.1 Our Approach .......................................................... 58
4.3 Jagged Tiling for Concurrent Start Applications ................................. 62
  4.3.1 Tile Shapes and Sizes .................................................. 64
  4.3.2 Our Approach .......................................................... 66
4.4 Fine-Grain Execution ............................................................................. 67
4.5 Convexity Analysis ................................................................................ 69
4.6 Conclusion ............................................................................................... 70

5 GREGARIOUS DATA RESTRUCTURING ................................................. 71
  5.1 Overview ......................................................................................... 71
  5.2 Reuse Analysis ................................................................................. 75
  5.3 Memory Storage Requirement ......................................................... 76
  5.4 Data Transformation ................................................................. 76
    5.4.1 Exploiting Parallelism ........................................................ 80
  5.5 Conclusion ......................................................................................... 80

6 EXPERIMENTS AND RESULTS .............................................................. 82
  6.1 Jagged Tiling Experimental Setup: ................................................. 82
    6.1.1 Pipeline Start Applications .................................................. 82
    6.1.2 Discussion ............................................................................. 86
    6.1.3 Parallel Start Applications ................................................... 90
    6.1.4 The Heat Kernels Case Study .............................................. 92
    6.1.5 DOE Proxy Mini-Apps ....................................................... 98
      6.1.5.1 Tealeaf Jacobi-2D ....................................................... 98
      6.1.5.2 MiniAMR ................................................................. 100
  6.2 Data Restructing Experimental Setup: ........................................... 104
    6.2.1 Discussion ............................................................................. 107
  6.3 Conclusion ........................................................................................... 114
7 RELATED WORKS .................................................. 115
  7.1 Compiler Techniques ........................................... 115
  7.2 Stencils and Tile Shapes ....................................... 117
  7.3 Data Movement and Transformation ............................. 119
  7.4 Other Techniques ............................................... 121

8 FUTURE WORK AND CONCLUSION .............................. 123

BIBLIOGRAPHY ...................................................... 126

Appendix

A RIGHTS AND PERMISSIONS ..................................... 136
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Performance Counters collected for analysis</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>Applications and their sizes</td>
<td>90</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance for PLT and FG execution with 240 threads. Note: Heat 3d peaks at 180 threads.</td>
<td>91</td>
</tr>
<tr>
<td>6.4</td>
<td>DOE Proxy Applications and Sizes</td>
<td>98</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance Counters collected in TileGx36</td>
<td>106</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance Counters collected in TileGx36 for Matrix Multiplication. Bold green and blue values represent the best values for Fine grain and OpenMP experiments</td>
<td>108</td>
</tr>
<tr>
<td>6.7</td>
<td>Performance Counters collected in TileGx36 for LU Decomposition. Bold green and blue values represent the best values for Fine grain and OpenMP experiments</td>
<td>109</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

1.1 Processor Memory Gap showing DRAM and Processor performance progression with 1980 performance as baseline [64]. .............................. 1

1.2 Memory Hierarchy ........................................................................ 3

1.3 SRAM Cache .............................................................................. 4

1.4 DRAM ......................................................................................... 7

1.5 Copy Transpose of matrix A to matrix AA ................................. 11

1.6 Different Network Topologies ...................................................... 14

1.7 Different Cache Layout ............................................................... 16

1.8 MESI states. Data comes from memory in ReadMiss(1) and from other caches in ReadMiss(2) [11] ...................................................... 17

1.9 Frequency, transistor count, power and performance trends [18] . 18

2.1 Affine and Convex Example .......................................................... 24

2.2 Affine Space (Derived from [57]) ................................................ 25

2.3 An affine hyperplane .................................................................. 26

2.4 Iteration Space Polytope ................................................................. 27

2.5 Scanning with new lexicographical ordering .................................... 29

2.6 Legal and Illegal Tiling ................................................................. 30

2.7 Two different tile schedules .......................................................... 32

2.8 Example with concurrent start ....................................................... 33
4.9 Jagged Tiling for Heat-1d. Tiling hyperplanes (1,-1) and (1,1) (denoted L1 Hyperplane) in original domain and (1,1) and (1,0) (denoted L2 Hyperplane) in the tiled domain

4.10 Fine Grain Execution Example

4.11 Dependencies Example for Jagged Tiling. Green arrows show satisfied dependencies

4.12 Dependency bits representation example. Top binary number representing dependencies status and lower decimal number representing bit index. Dependency1 (0x111F) and Dependency2 (0xF)

5.1 Matrix Multiplication

5.2 LU Decomposition

5.3 LU Decomposition inner block calculation showing row and column access of matrix A to calculate the result matrix as shown in Figure 5.4(b).

5.4 Iterative Block LU

5.5 Transformation Condition 1: Strided access (shown by downward arrows) and row arrangement of parallel tiles (A-E) transformed to have contiguous access using calculated offset (shown in red).

5.6 Transformation example calculating the offset and the new index. First the new index at tile level is calculated relative to the original index K and J, which then is translated to the element index by multiplying them with tile sizes.

5.7 Transformation Condition 2: Strided access (shown by downward arrows) and column arrangement of parallel tiles (A-E) transformed to have contiguous access using calculated offset (shown in red).

5.8 Matrix Multiplication restructuring data movement

5.9 LU Decomposition restructuring data movement

6.1 Intel Xeon Phi
6.2 Stencil 2K by 2K: Parameter sweep with different number of thread
groups and threads per group ........................................ 83

6.3 Stencil Example Execution times for PLUTO generated code (PLT)
and our framework (FG) ................................................. 84

6.4 Seidel 1D Execution times for PLUTO generated code (PLT) and our
framework (FG) ......................................................... 84

6.5 Seidel 2D Execution times for PLUTO generated code (PLT) and our
framework (FG) ......................................................... 84

6.6 Seidel 2D Remote Cache misses for PLUTO generated code (PLT)
and our framework (FG) ............................................... 85

6.7 Seidel 2D Memory-served misses for PLUTO generated code (PLT)
and our framework (FG) ............................................... 85

6.8 Seidel 2D write serviced by memory for PLUTO generated code
(PLT) our framework plus overhead and our framework without
overhead. Note that the Y-axis is in logarithmic scale. ............ 86

6.9 Heat 1D Memory behavior for PLUTO generated code (PLT) and our
framework (FG) ......................................................... 93

6.10 Heat 2D Memory behavior for PLUTO generated code (PLT) and our
framework (FG) ......................................................... 94

6.11 Heat 3D Memory behavior for PLUTO generated code (PLT) and our
framework (FG) ......................................................... 95


6.13 TeaLeaf Memory Behavior for PLUTO generated code (PLT) ... 100

6.14 TeaLeaf Memory Behavior for Fine-grain (FG) ..................... 101

6.15 miniAMR[2] ........................................................ 102

6.16 miniAMR Memory Behavior for PLUTO generated code (PLT) .. 103

6.17 miniAMR Memory Behavior for PLUTO generated code (PLT) .. 105

xiv
When powerful computational cores are matched against limited resources such as bandwidth and memory, competition across computational units can result in serious performance degradation due to contention at different levels of the memory hierarchy. In such a case, it becomes very important to maximize reuse at low latency memory space. The high performance community has been actively working on finding techniques that can improve data locality and reuse, as such endeavors have a direct and positive impact on both performance and power. In order to better utilize low latency memory such as caches and scratch-pad SRAMs, software techniques such as hierarchical tiling have proven very effective. However these techniques still operate under the paradigm that memory agnostic coarse grain parallelism is the norm. Therefore, they function in a coarse-grain fashion where inner tiles run serially. Such behavior can result in an under-utilization of processing and network resources. Even when the inner tiles are assigned to fine grain tasks, their memory agnostic placement will still incur heavy penalties when resources are contended. Finding a balance between parallelism and locality in such cases is an arduous task and it is essential to seek a common ground where both processing and memory resources can collaborate to yield a better overall utilization.

This thesis explores the concept of locality aware multithreading called “Group Locality”. Multiple group of threads work together in close proximity in time and space as a unit, taking advantage of each other’s data movement and collaborating at a very fine-grain level. When an execution pattern is known \textit{a priori} through static analysis and data is shared among different thread groups, the concept of group locality extends further to rearrange data to provide better access pattern for other thread groups. This thesis presents an end-to-end framework that takes advantage of information gained by
the compiler and utilizes it for careful orchestration of memory access, data movement and data restructuring. The contributions of this thesis include:

- An efficient tiling strategy to exploit intratile parallelism. Using an existing polyhedral framework and compiler toolchain, we created a multi-hierarchical tiling with highly parallel tiles that map to the lowest level cache and communicate with minimal overhead. Sets of these inner tiles form outer level tiles that run in parallel.

- A highly optimized fine grain runtime where threads execute in a data-flow fashion and synchronize mainly using atomic operations.

- A data movement and restructuring strategy based on access pattern that allows access to have minimal interference. Based on reuse and dependency vector information, threads move data using a low overhead restructuring strategy where multiple threads perform collaborative data restructuring and reuse.

Our current implementation of the framework exploring intra-tile parallelism and fine-grain execution show significant performance improvement over instances produced by state-of-the-art compiler framework for selected stencil applications on an Intel Xeon Phi board. Similarly, our results for our restructuring strategy on a many core architecture, Tilera TileGx, beats optimized OpenMP code by a significant margin. Hardware counters collected during our experiments shows improvement in reuse and reduction in conflicts at different levels of the memory hierarchy.
Chapter 1

INTRODUCTION

1.1 Early Years

In the early 80s, when silicon industry was in its infancy, economic difficulty set apart processor and DRAM industries [113]. Since then, processor industries controlled by the United States went for high performance to meet the existing demand and worked on designs to make faster processors. On the other hand, the DRAM industries controlled by the rest of the world emphasized on maximizing the bit capacity instead of producing faster memory. This led to a huge disparity in processor and memory performance and the trend that started in the early 80s has continued for over a couple of decades as shown in Figure 1.1. This memory processor gap is known as the “memory wall” [25].

![Figure 1.1: Processor Memory Gap showing DRAM and Processor performance progression with 1980 performance as baseline [64].](image-url)
With processors running at much higher speed than memory, improving bandwidth utilization and reducing memory access latency became crucial for achieving higher performance. However, increasing network bandwidth and replacing slower memory with faster memory was not a cost effective solution; one of the main reasons being the limited real estate on a chip. To remedy this situation, the concept of memory hierarchy [62] was introduced to take advantage of the locality of reference.

The Principle of Locality states that programs tend to reuse data and instructions that have been recently used [63]. The majority of a program execution time (roughly 80-90%) is spent in 10% of the code on average. These numbers roughly also applies to data. This implies that it is very likely that the once used data reference or its surrounding references are used within close proximity in time and space. Such locality can be divided into two types:

- Temporal Locality: Recently used references are likely to be accessed within close proximity in time.
- Spatial Locality: References within close proximity in space tend to be accessed close together in time.

This means that code and data that are used most often can be kept in a smaller but faster memory. It is a cost effective solution for memory architecture design. Since exploiting temporal and spatial locality has a huge impact on performance, architectures are designed with multiple levels of memory with different access latencies. Smaller and faster memories are kept closer to processors and larger slower memories are kept farther. Such design allows percolation of data through different levels of the hierarchy with a goal of keeping average memory access latency minimal.

1.1.1 Memory Hierarchy

Figure 1.2 shows a basic memory hierarchy structure with registers, caches, DRAM and hard disk. Registers and caches have very small memory access latency
but are costly in terms of manufacturing cost. In contrast, DRAM and hard disk are slower and cheaper to produce.

1.1.1.1 Registers

Processor registers are storage devices designed for faster access using multiple ports SRAM. This makes such design extremely efficient as read/write can be done in parallel. Although every architecture exposes many registers, only few known as General Purpose Registers (GPR) are made available to the programmers. In contrast, Special Function Registers (SFR) are designed to control various functions of processor (e.g. stack pointer, program counter, conditional codes) are not directly writable.

Nowadays, registers are normally 32 or 64 bits (one word) in length. However, high performance architectures also offer registers known as vector registers, which are designed to take advantage of parallelism using vectorization. In Intel and AMD architectures, such registers can be up to 512 bits in length allowing 8 words for data manipulation and arithmetic operations [1].

1.1.1.2 Caches

Caches are designed to allow faster access to data and are implemented using single port SRAM that can hold data when there is a power supply. Such design
requires six transistors as shown in Figure 1.3(a). This makes them costly in terms of silicon real estate and production values. Caches connect to network with wires and switches using different incore topologies as mentioned in Section 1.2.1. Figure 1.3(b) shows a single core with basic two levels caches connected to a DRAM with a bus.

Figure 1.3: SRAM Cache

Caches keep recently used instructions and data so they can be reused, reducing the need to go to the expensive DRAM. When a memory location is accessed, instead of bringing just a requested word, an entire cache line (4 or 8 words) is brought into caches. The reason for this extra data movement is inspired by the locality principle. The idea is to bring data into caches such that the processor can access any element within the cache line at lower cost. Next time the processor request data and if it is available in the cache, it is called a cache hit. On the other hand, if the data does not reside in the cache and the access has to go to the main memory, it is called a cache miss. Cache hits for the same memory address provides temporal locality whereas cache hits for addresses within the cache line provides spatial locality.

Caches have multiple levels. Lower level (a.k.a L1) caches are the closest to processors and are the fastest whereas higher level caches (a.k.a L2 and L3) are larger with slightly higher latencies. In such cases, data can reside at different levels and percolate in and out of the memory hierarchy. Caches employ different policies that dictate where the copies of data reside. In inclusive caches, copies of data in L1 also
reside in L2 and L3 (e.g. Intel Nehalem). On the other hand, in exclusive caches (AMD Athalon Thunderbird), data reside only in one cache level and not in both. Although exclusive caches allow more data to fit in limited cache space, when access latency between caches vary by a big margin, inclusive caches are preferable. Such design allows circulation of data within different levels of the cache hierarchy reducing the average memory access latency. Besides such, architectures like TileGx allow users to have full control of data movement and its placement. In all cases, when there is a miss in lower level caches, higher level caches are checked before sending requests to the memory.

Addresses in cache have three portions. A tag to check all the blocks (lines) in the set, an index to select the set and a block offset to retrieve data from the address. To check the validity of the data, a validity bit is added to the tag. Since caches are smaller than DRAM, there are multiple schemes that can be used to map data in a cache. The most widely used of all schemes is the set associative. In this scheme, a set is formed by a group of blocks. Thus, a set is chosen by using the modulus of the block address with the number of sets in cache. During data access, a block is first mapped to a set where it can be placed anywhere within the set.

Factors that determine the conflicts and hence the lifetime in caches are the cache size, block size and the associativity. When a cache has one block per set, i.e., the given address is mapped to only one address in the cache it is called directly mapped. However such design can be very inefficient due to possible conflicts. On the other side of the spectrum is the fully associative cache that has only one set, i.e., it allows data to go to any address in the cache. Although the conflict reduces with full associativity, it also reduces the effectiveness of caches due to the overhead of checking through the entire cache space. The well accepted solution is the n-way associative cache that divides a cache into sets with n blocks and an address can be mapped to any of them. Most of the newer Intel architectures such as Haswell, Sandy Bridge, Knights Corner use 8-way associative caches. Some architecture like TileGx-64 uses 8-way associativity for L2 cache but only 2-way associativity for L1.
When data is initially accessed, caches are cold, i.e., data reside only in memory and not in caches. This results in *compulsory misses* during cache access. Once data start moving into caches, all space in caches are eventually taken. New data replaces older data every time there is a conflict. When data eviction happens in n-way or fully associative caches because of collision and has to be brought in later, such misses are called *conflict misses*. Similarly, when an eviction occurs because a data set needed during execution is too big to fit in caches, such misses are called *capacity misses*.

When and how the updated data percolates within different levels of the memory hierarchy depends on its write policy. If some modified data is updated both in the lower level caches and its backup storage at the same time, it is called *write through* policy. On the other hand, if the modified data is updated only in the lower level caches and is updated in its backup storage (e.g. higher level caches and memory) only when replaced, such design policy is called *write back* policy. Although, *write through* policy is easier to design, *write back* policy is normally preferable because of bandwidth savings.

To make caches more efficient, some architectures use write buffers that hold outstanding memory writes [10]. Such buffers hold values temporarily allowing caches to perform other operations. In addition, if the data being written to the memory have contiguous addresses, such buffers allow aggregated write to the memory. Interested readers can find details about cache implementation in [62].

During the single core era, the benefits of caches were unprecedented. A single core used data brought into caches and this allowed full exploitation in terms of locality and reuse. When there is only one thread of execution, the thread has full control of what is brought in and when it is reused. However, in multicore architectures, caches are normally shared among multiple threads. This means, shared data can be updated by parallel thread units resulting in the need for keeping states about where the most

---

1 sequence of instructions within a process. It consist of program counter, register sets and stack space.
recent value resides. Such issues are resolved using a *Cache Coherency Protocol* as discussed briefly in section 1.2.2.

### 1.1.1.3 DRAM

DRAM stands for *Dynamic Random Access Memory*. It uses one transistor and a capacitor to store a bit in its design. Hence, it can store more data in a smaller space than SRAM. However, in order to prevent loss of data due to leakage, such design needs periodic refreshing to keep the capacitor charged. The core architecture of DRAM is constructed using array of rows (memory page) and columns.

Accessing a memory address requires two decoding steps. First, a row is selected and is followed by a column address [42, 109]. When the row is open, multiple column addresses from that row can be accessed without paying heavy penalty. When a read operation is completed, the row has to be written back to the same memory address. This is because DRAM accesses are destructive and hence need to be written back to the memory. Such operation is called precharge and is known as closing of a memory page.

![DRAM Schematic](image1)  ![DRAM Structure](image2)

Figure 1.4: DRAM

Figure 1.4(a) and 1.4(b) shows the picture of DRAM schematic and DRAM architecture respectively. Typically, only one thread at a time is allowed an access
to a rank which is a group of memory pages in a DIMM. Hence, achieving higher utilization of memory access requires reducing interference in ranks and rows in the physical memory device.

Also, DRAM runs at relatively lower frequency as compared to the processor. In order to keep up with faster processors and increasing demand for higher performance, manufacturers use burst read and write to the memory. For example, DDR3 uses 8 bit burst read/write, which means that the memory can transmit 8 bits of data to I/O buffers every cycle. As a result, DDR3 can keep up with I/O running at a frequency 8 times the DDR3. Obviously, the negative consequence of this is the extra data movement, which might not have any usage. Hence, there is a tradeoff between improving memory utilization and keeping available bandwidth occupied with useful traffic. For that reason, DDR4 still uses 8-bit burst read/write instead of 16 as it fits very well with the current trend of 64 bytes cache line that helps exploiting spatial locality without overwhelming the bandwidth.

1.1.2 Further Improving Cache Locality

Improving cache locality directly translates to higher performance as it reduces strain on the memory subsystem. Although in many cases, cache performance is application specific or access pattern dependent, there are hardware and software techniques to improve locality.

1.1.2.1 Prefetchers

Prefetching is a technique that overlaps data movement with computation such that data is brought into nearby caches a priori to the actual access. Such prefetching of data can be done both in hardware and software and there have been studies evaluating the two [31].

Hardware prefetching uses sophisticated pattern matching and detection algorithms to predict upcoming accesses and perform timely data movement. Such designs

---

2 Dual Inline Memory Module, module containing one or more memory modules
vary with architectures as they employ different prefetching policies. Some prefetchers are activated only when there is a cache miss whereas some monitor the access pattern, predict upcoming accesses and issue prefetch a few blocks ahead even when there is a cache hit. Such techniques only work well with regular applications where access patterns are predictable. The disadvantage of hardware prefetcher in some cases is the unnecessary data movement consuming limited bandwidth and performance impeding cache thrashing at different levels of the memory hierarchy. To limit such unwanted behavior, some architectures allow users to set parameters that set the behavior of the prefetcher [35].

Software prefetching, on the other hand, is controlled by the programmer. Within a loop nest, one can identify when a data is needed and use special prefetch instructions. With this technique, a data request is made a few clock cycles distance ahead of time, known as a prefetch distance. Since the average execution time can vary during runtime, such prefetch distance on one hand has to be large enough to hide latencies and on the other hand has to be small enough to not allow unnecessary eviction from caches.

Prefetchers and caches work together very well when the accesses to the memory are contiguous or have regular access patterns. However in many scientific applications there are loop constructs that are heavily compute intensive with complicated dependencies. Since most of the runtime is spent executing these kernels, optimizing them can have significant improvement on performance. This has been a well studied problem with many classical loop/data transformation and optimization techniques [69, 116, 115, 105, 33, 41, 94, 79, 73].

1.1.2.2 Loop Transformation

A loop transformation is a compiler technique that reduces the associated loop overhead without violating its loop carried dependencies. Transformations like loop skewing, loop permutation, loop reversal, loop interchange, loop fusion, loop fission, etc., can improve reuse in caches and also extract more parallelism. However, with
such transformations, it is very difficult to know how they interact with each other and in what order such optimizations have to be performed. One of the most important and widely used optimizations that attempts to maximize reuse in low latency caches is tiling \cite{78, 69, 116, 115, 105, 33, 41, 94, 95, 66, 65, 6, 7}. This approach sub-partitions iteration space into smaller subspaces called “tile” bringing different dimensions of iteration space together. As the reuse is exploited across multiple dimensions of an iteration space, the number of trips required to the memory is significantly reduced. Hence, this approach is very effective as it maximizes reuse at different levels of the memory hierarchy.

1.1.2.3 Data Transformation

The efficiency of memory access depends on memory layout and the way they are accessed during program execution. For example, when array elements with row major storage are accessed along a row, it exploits spatial locality. However, when elements are accessed by column, spatial locality is non-existent in general. This is because of the way memory subsystem is designed as data is read in burst in DRAM and brought into caches at cache line granularity. Memory access patterns however are entirely application dependent. When an application access data in a strided fashion, plenty of data moved to caches get evicted before being used. This puts an unnecessary burden on the memory subsystem. With data transformation techniques, strided accesses can be transformed to contiguous accesses resulting in better locality. Figure 1.5 shows an example of a copy transpose where accesses to matrix $A$ with a stride of n is changed to stride of 1 in $AA$.

Data transformation is normally done using an invertible transformation matrix where addresses in one space are mapped to another \cite{79, 73, 70}. Such technique is independent of loop carried dependencies, which provides a lot of flexibility in transformations. However, since data transformations affect the behavior of the entire program, it can have detrimental effects when multiple access patterns or multiple loop nests are involved. There always exists a chance that the optimal layout for one loop might
//Copy Transpose
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
        AA[j][i] = A[i][j];
    }
}
for (int t=0; t<N; t++) {
    for (int i=1; i<n; i++) {
        for (int j=1; j<n; j++) {
            //AA[i][j] has a stride of 1
            AA[i][j] = 0.2*(AA[i][j-1] + AA[i][j] + B[i][j]);
        }
    }
}

Figure 1.5: Copy Transpose of matrix A to matrix AA

have an undesired effect in other loops that access the same data structure. Also, data
transformation comes with an overhead of extra memory space and the need for data
movement. Such overhead, however, can be amortized when enough reuse is available.
Such transformations can also be unified with control transformation in cases when
data transformation alone is not able to exploit maximum locality [32].

1.1.3 Paradigm Shift to Multicore and the Impact

Until the early 2000s, higher performance was mainly achieved by increasing
the frequency of processors [5]. Higher frequency directly translated to more compute
power, enabling processors to execute instructions in a shorter time frame. This ap-
proach, although well accepted by many companies, came at a cost of sophisticated
hardware technology such as deeper pipelines, out-of-order engines, branch predic-
tion engines, and more. Also, architectures were designed to exploit instruction level
parallelism with VLIW and Superscalar technologies. Such ability to execute mul-
tiple instructions in parallel helped improve cycles per instruction leading to higher
performance. However, higher frequency led to more power consumption and higher
heat density, testing the physical limit of the silicon device. As a result, there was a
paradigm shift from single core to multicore. Many simple cores replaced complicated
single core and parallel execution became more visible.

1.2 Multicore Era

The concept of parallel processing existed long before the rise of multicore. Architectures like Cray XMP and IBM 308X computer series were using multiple processors in the mid 80s. Also, parallel processing with cluster using off-the-shelf computers started in the late 80s. Even in single processor, the use of *Simultaneous Multithreading (SMT)* to exploit parallelism at both the instruction and the thread level, have been around for multiple decades and is used to improve efficiency of superscalar processors [44]. Intel’s version of SMT known as hyperthreading [49] was employed to make efficient use of resources such as dedicated floating point and integer units. However, the performance improvement was limited by the availability of shared resources such as the pipeline. In contrast, multicore systems are equipped with multiple physical cores with their own pipeline, ALU units and are highly efficient.

In the mid 2000s, multicore became mainstream and spread from embedded systems, desktops, and servers to the mobile devices. This trend revolutionized the way of thinking in computer design, as such chips were able to deliver very high performance at lower clock frequency and supply voltage. In general, multicores are much simpler architecturally and hence more power efficient [56] but are still consistent with Moore’s [100] prediction without pushing the constraints of the physical device. Since every core within the system contributes to the overall performance, it delivers higher computational power. Such architectures can be homogeneous or heterogeneous,

- Homogeneous architectures are composed of cores with similar functionality and performance. They can run same binaries and are mostly indistinguishable from user point of view. Also, such architectures are easier to program.
- Heterogeneous architectures are composed of different types of cores designed for
specialized processing capabilities. The may have different functionality, performance and instruction sets. In terms of power and performance, such architectures are more efficient but are normally difficult to program (e.g. IBM Cell).

Both homogeneous and heterogeneous architectures have found success in recent era. With a goal of maximizing performance at low power budget, many innovative technologies, multicores and high performance computers have emerged over the years [20]. Supercomputers like Cray XMT [48], Cyclops-64 [67], RoadRunner the first hybrid supercomputer [77], Tianhe-2 [81] and many more, all boast very high, up to peta-scale performance while Exa-scale studies are being conducted at a rapid pace [76, 101].

Despite having abundant processing resources, it is very challenging to reach the theoretical peak offered by these new multicore architectures. Limited shared resources such as memory and network bandwidth grow very slowly and are very difficult to utilize efficiently. This translates to higher latencies while performing memory operations. In addition, complicated network topologies add to core communication and coherency required for multicore caches.

1.2.1 On Chip Network Topology

Topology determines how cores are interconnected within a network. The most widely used of all interconnection has been a bus network (e.g. Intel Atom, ARM Cortex). Bus is easy to implement and all cores have uniform latency to the memory. However, such design works only for few number of cores and has low bisection bandwidth.

Ring has higher bisection bandwidth than bus and can support more processors. However, it suffers from non-uniform latencies (e.g. IBM Cell [91], Intel Larrabbe).

Point-to-point interconnect provides higher bandwidth, lower latencies but with a loss of global view (e.g., Intel Sandy Bridge).

3 capacity across the wires of the narrowest bisector
The crossbar switch is arranged in a matrix configuration such that the input and output lines intersect each other. Each intersection point contains a switch that can be closed to establish a connection. Advantages of a crossbar switch include high bandwidth, support for large number of cores and equal latency to all connecting counterparts (e.g. Cyclops-64 [39], Sun Niagara, Microsoft Xenon).

Similarly, Cray-XMT [48] uses 3D-torus and Tile-64 [34] uses grid, and both of these design suffer from non-uniform latencies. Figure 1.6 shows a pictorial view of Crossbar Switch, Torus and Grid.

The selection of right topology depends on the goal of the architecture design and the applications it is targeting. For example, architecture designs that target irregular application such as the Cray XMT, does not have much negative impact because of non-uniform latencies. However, this is not the case for architectures with shared memory designed for regular applications.

1.2.1.1 Shared Memory

Multicore machines with shared memory allow parallel actors to run independently while sharing common memory resources. When a memory location is updated...
by one thread, it is visible to all. Such design takes the burden of explicit communication away from the programmer. However, ensuring locality in such cases become more difficult. Shared memory can be

- Uniform Memory Access (UMA): Machines that have equal latency access to the main memory from all processors. Such machines are sometimes represented by Symmetric Multiprocessors (SMP).

- Non Uniform Memory Access (NUMA): Machines that have unequal latency access to the main memory from different processors. When a processor access local memory, latency is minimal, whereas accessing shared memory depends on physical wire distance and the network hops it has to travel. For example, in Cray XMT, such latency can vary from 30 cycles to 1000 cycles.

Some multicore designs have shared caches where different cache hierarchy is shared by either hardware threads or physical cores. In such cases, extracting higher performance requires careful orchestration of data movement.

### 1.2.2 Multicore Caches

Where multicore designs have provided enormous amount of processing resources, they have also exacerbated the data locality issues. It is no longer true that the data moved by one thread is used only by the thread performing the data movement. In multicores, when caches are shared, possible conflicts can happen among cores sharing the space. Figure 1.7(a) shows a shared L3 cache with private L1 and L2 caches (e.g., Intel Nehalem, Sandy Bridge). Similarly, Figure 1.7(b) shows cache layout with private L1 and L2 caches, where L2 also acts as a distributed cache giving an impression of L3 cache (e.g., Intel Xeon Phi). Layout like these provides an opportunity for data sharing among different threads. However, such design is also prone to having unnecessary conflicts and misses.

Also, when caches are shared, multiple reads and writes can happen in parallel to the same address space. To ensure correctness, memory consistency models [88]
provide rules that are followed during memory operations. The other aspect of shared memory model is the coherency [106]. When multiple threads have the copies of the same data and one of them is a write, all other threads need to be updated about the changes so it does not use the stale value. Such updates are done using a cache coherency protocol such that the caches are invisible from programmer’s point of view and one does not need to know where the value comes from.

For example, one of such protocol is MESI. Each cache line can be in one of the following four states:

- Modified(M): $M$ states that the cache has a valid copy of the data block that has just been modified.
- Exclusive(E): $E$ states that the cache block is valid and exclusive suggesting no other cache have a valid copy of the data.
- Shared(S): $S$ states that the cache has a valid copy of the data that are shared and hence other caches may have a read copy as well.
- Invalid(I): $I$ states that the block is invalid i.e., potentially stale.

Figure 1.8 shows the state diagram for MESI protocol and the conditions for state transitions. On a local read miss, if the state is $(I)$ and a block is not present in
other core caches, it is loaded from the memory and the state is changed to E. However, if a block is in state S, M or E and the data is loaded from other core caches, the state is changed to S for both local and remote copy. On a write miss, the block is loaded and the state is changed to M with invalidation of all other cache’s copies. On a write hit at state S, the block state is changed to M and all other copies are invalidated. On a write hit at state E, the state is changed to M whereas on a write hit at state M, the state does not change. The advantage of MESI protocol is when a block is at state M and E, write hit traffic does not need to be posted in a bus.

**Figure 1.8:** MESI states. Data comes from memory in ReadMiss(1) and from other caches in ReadMiss(2) [11]

Maintaining coherency in multicore architectures is very expensive as it consumes both bandwidth and cycles. One solution, many architectures use is replace caches with scratchpads. Scratchpads are very similar to caches except that they are software controlled and allow explicit data movement. While it provides full control of data, it also places data movement burden to the programmer. Despite such, scratchpads have become common in architectures designed for high performance computing.

### 1.2.3 Continuing Architectural Trends and Issues

Figure 1.9 shows the current trend for transistor count, performance, frequency, power and the number of cores for Intel architectures since the middle of 70s. Since the
arrival of multicore, the trend shows that increasing transistor count is not returning performance as it did in single core days.

One major reason is because most architectures today are still built with Von Neumann model where programs and data are stored in memory that is separated from the processor. Using tightly coupled processor and memory a.k.a. Processor-in-memory (PIM) to take advantage of high bandwidth within DRAM is one approach to reduce memory access latency. Such design also claim to be efficient in terms of power and silicon area [24]. However, such technology has not been widely adopted.

Also, issues arise because of the segregation of architectural and software design. Although performance capability of new architectures is improving very rapidly, system software and tool chains like runtime systems and compilers are lagging behind [83]. While multicore design provides an opportunity for higher performance, it also forces us to think beyond traditional ways. Programming models such as OpenMP [86] while providing an efficient way of performing coarse grain execution, are not enough for highly multithreaded system where fine-grain is the only way to utilize all available resources. In order to improve system performance and utilization, execution models like data flow promise efficient execution, exploiting parallelism on a large scale [40].
With data driven execution in mind, data flow is designed to execute once its operands are ready. Such execution schemes do not have a concept of control flow but it can be simulated with special actors. It can allow many instructions to execute concurrently exploiting parallelism to maximum. Although, there have been few recent work on data driven execution targeting next milestone in performance \cite{110, 53, 68, 107}, it has yet to take off on a large scale.

Additionally, exploiting parallelism in the software does not necessarily result in parallel execution in the hardware. A thread accessing different pages in the same bank or multiple threads accessing the same memory bank result in sequential execution in the hardware. The realization that increasing cores and parallelism just from the control perspective is not enough to overcome the degradation caused by the memory subsystem is starting to sink in. Memory access latency still stands as a major bottleneck for performance and a significant contributor to the power consumption problem. Optimization for higher performance hence needs a global view of both control and memory such that they can be optimized as a single unit.

1.3 Problem Statement

As we strive for higher performance and lower power consumption, memory access latency continues to haunt us with long latency clock cycles and excessive power consumption. The problem seems to exacerbate as the trend continues to show increase in computational power at much higher pace than shared bandwidth and memory. In such a case, limited shared resources are contended across different computational units causing serious performance degradation. Such contention can happen in shared bandwidth, interconnect networks, memory controller and memory banks. Also, the interference at multiple levels of memory hierarchy can cause unnecessary eviction of data causing more contention and worsening the situation further. To solve this problem, high performance community has been actively investigating to come up with better optimization techniques. Taking advantage of the locality principle, multiple efforts \cite{50} have been made to minimize the access time to memory in an integrated
framework, such as by storing recently used data in a cache, efficient reuse of cached
data through tiling, data percolation using communication-avoiding algorithms [37],
code transformations and pre-fetching. Although these approaches work very well in
pushing the “Memory wall” farther, data movements end up being performed in most
cases for the benefit of a single thread or computational unit. This scenario can create a
myriad of problems when multiple hardware resources are shared between a significant
number of threads.

Hierarchical tiling techniques still operate under the paradigm that memory
agnostic coarse grain parallelism is the norm. As a result they run in a coarse grain
fashion, running inner tiles serially. Such behavior can result in underutilization of
processing resources. One of the main reasons that this pathology has persisted is a first
order limited memory bandwidth assumption. However, this assumption ignores the
possibility of memory reuse across processing units, especially when they share different
levels of the memory hierarchy. Secondly, when each thread performs data movements
for itself without a priori knowledge of concurrent thread execution and data movement
pattern, it can interfere with other threads causing performance deterioration (e.g. false
sharing of cache lines). This could take the form of increased cache misses and a higher
strain on the entire memory subsystem, as well as an increase in contention over the
system’s shared resources. Thirdly, when sets of tiles are reused multiple times from
the memory space in a state where accesses are not fully contiguous and they are spread
over multiple memory pages and memory banks, opportunities to speed up memory
accesses of reused tiles with well known access patterns are lost. All these observations
have a severe impact on both the performance of an application, as well as its energy
profile.

As the trend continue to show increasing number of processing element in a
chip, many fine-grain execution techniques have emerged to take advantage of these
overwhelming processing resources. However, their memory agnostic placement still
incurs heavy penalties when resources are contended. First of all, when reuse in the
highest level memory hierarchy is not fully utilized, trips to the memory increase causing contention in the limited memory bandwidth. Secondly, when concurrent threads access the same memory banks in DRAM, accesses get serialized. Also, when concurrent threads attempt to access addresses from different memory pages residing on same memory banks, opening and closing of pages can incur tremendous overhead. Studies have shown that accessing open pages is 3-5 times faster than accessing closed pages. However, keeping one page open would mean more overhead for accessing other pages in the same bank. In order to take advantage of additional processing resources, current and new software stacks need to provide concurrent units the ability to (re)act based on other threads’ execution and the application data movement pattern. In other words, these stacks need to update their machine models to reflect modern platforms. Starting with the axiom that the base optimization technique will be tiling, memory reuse needs to be considered in both the inter, as well as, the intra tile so that the strain or under-utilization of the system’s memory bandwidth can be alleviated.

1.4 Contributions

This thesis explores the concept of locality aware multithreading called “Group Locality”. The concept of group is based on threads working together in close proximity in time and space. Such threads take advantage of each others access pattern and hence maximize reuse not just within self but also among set of other concurrent threads. With a priori knowledge of each other’s execution pattern, set of threads collaborate at a very fine granularity exploiting both locality and parallelism. When an execution pattern is known a priori and data is shared among different thread groups, the concept of group locality extends further to rearrange data to provide better access pattern for other thread groups. Such an effort to improve locality is hence extended beyond the highest level of memory hierarchy. In order to achieve this, this thesis presents an end-to-end framework that takes advantage of information gained by the compiler and utilizes them for careful orchestration of memory access, data movement and data restructuring. The contributions of this framework includes:
• An efficient tiling strategy to exploit intratile parallelism. Using an existing compiler toolchain – in our case, the polyhedral framework PLUTO [22] and the code generator CLooG [15], we created a multi-hierarchical tiling with highly parallel tiles that map to the lowest level cache and communicate with minimal overhead. Sets of these inner tiles form outer level tiles that run in parallel [103, 102].

• A highly optimized fine grain runtime where threads execute in a data-flow fashion in which threads synchronize mainly using atomic operations [103].

• A data movement and restructuring strategy based on access pattern that allows access to have minimal interference. Based on reuse and dependency vector information, threads move data using a low overhead restructuring strategy where multiple threads perform collaborative data restructuring and reuse [104].

The rest of this dissertation is organized as follows. Chapter 2 presents notations and background on polyhedral theory needed to understand the content of this thesis. Chapter 3 describes the Group Locality multithreading framework and its components. Chapter 4 provides details about a novel jagged tiling technique to exploit parallelism and locality. Chapter 5 explores collaborative low overhead restructuring strategy. Chapter 6 describes the selected hardware / software testbeds and shows the experimental data for selected test cases. Chapter 7 presents prior related works. Finally, chapter 8 provides the future work and conclusions.
In recent years, there has been significant amount of work on loop/data transformation, optimization and code generation using polyhedral abstraction [84, 41, 73, 70]. Our work leverages some of the previous works such as PLUTO [22] and CLooG [15] to explore locality in multithreaded systems. In this chapter, we present the basics of the polyhedral model and notations required to understand the framework developed in this thesis. Since the framework explores different tile shapes and exploits additional parallelism, we also cover some theoretical background behind such techniques.

2.1 Basics of Polyhedral Theory

Polyhedral model provides a geometric and an algebraic way to capture the execution of a given program [60, 16, 23, 93, 92, 46, 47]. Using loop boundaries and dependencies between iterations, it captures the compact representation of a program. Such representation can be used to characterize inter and intra statement dependencies and reason about transformations. Such approaches are applicable to loop nest where loop bounds and data access functions are affine functions of parameters and variables [118].

2.1.1 Affine Function

A function $f$ is affine if there exist a matrix $A^{m \times n}$ and a vector $\vec{v} \in \mathbb{R}^n$ such that $\forall \vec{x} \in \mathbb{R}^m$, $f(\vec{x}) = A\vec{x} + \vec{v}$.

For example, a function $f(x) = 5x + 2$ is an affine function that forms a straight line in two dimensional space.
2.1.2 Affine Space

Given a matrix $A^{m \times n}$ and a vector $\vec{v} \in \mathbb{R}^n$, a set of solutions given by $S = \{\vec{x} \in \mathbb{R}^n | A\vec{x} \leq \vec{b}\}$ is an affine space of the system $A\vec{x} \leq \vec{b}$. Such space is closed under affine combination.

An affine combination of points $x_1, x_2, \ldots, x_n$ in a set $S$ is of the form,

$$x = \sum_{i=1}^{n} x_i \lambda_i \mid \sum_{i=1}^{n} \lambda_i = 1, \lambda_i \in \mathbb{R}$$

For example, given the points $x, y \in S$, a space is affine if and only if it includes the entire infinite line containing points $x$ and $y$. It means that such set includes every possible affine combination of its point. Figure 2.1(a) shows a pictorial view of such affine set for an equation $y = 1$.

An important concept regarding an affine space is a property of convexity. A convex combination of points $x_1, x_2, \ldots, x_n$ in a set $S$ is of the form,

$$x = \sum_{i=1}^{n} x_i \lambda_i \mid \sum_{i=1}^{n} \lambda_i = 1, \lambda_i \geq 0, \lambda_i \in \mathbb{R}$$

![Figure 2.1: Affine and Convex Example](image)

A set $S$ is convex if it is closed under convex combination. It can be viewed as a special case of the affine combination where all coefficients are non-negative. Figure 2.1(b) shows a pictorial view of a convex set for an equation $y = 1 : -1 \leq x \leq 1$. It only includes points between $-1$ and $1$. 

24
Obviously, above mentioned examples only show few properties of affine space. Affine space always impiles convexity but not the contrary. Such affine space can be defined further as: Given a set of points $a_1, a_2...a_n \in S$ there exists set of vectors $v_1, v_2...v_m \in \vec{V}$ and an action $+: S \times \vec{V} \to S$, satisfying the conditions below.

1. $a_1 + 0 = a_1$, for all $a \in S$

2. $(a_1 + v_1) + v_2 = a_1 + (v_1 + v_2)$, for all $a \in S$ and $v \in V$

3. For any $a_1, a_2 \in S$, there exist $v_1 \in \vec{V}$ such that $a_1 + v_1 = a_2$

Figure 2.2 shows a pictorial view of how these conditions apply to an affine space when $\vec{V}$ acts on $S$.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure2_2.png}
\caption{Affine Space (Derived from [57])}
\end{figure}

### 2.1.3 Affine Subspace

For any given affine space $S$ with points $a_1, a_2...a_n$, a subset $S' \subseteq S$ including points $a_1, a_2...a_k$ is also an affine space if it is closed under affine combination, i.e.,

\[ x = \sum_{i=1}^{k} x_i \lambda_i \mid \sum_{i=1}^{k} \lambda_i = 1, \lambda_i \in \mathbb{R} \]

For example, in 2D space a line is a subspace and similarly in 3D space a plane and a line both are subspaces.
2.1.4 Affine Hyperplane

A hyperplane is a $n - 1$ affine subspace in $n$ dimensional space. A hyperplane divides a space into two halves known as half-spaces, each of which can be represented by an affine inequality. A hyperplane $\phi(\vec{i})$ with iteration vector $\vec{i}$, dimensionality $m$ and normal $(c_1 c_2 ... c_m)$ can be represented in equation form as,

$$\phi(\vec{v}) = (c_1 c_2 ... c_m).\vec{i} + c_0$$

(2.1)

Figure 2.3: An affine hyperplane

Figure 2.3 shows a hyperplane $x + y = b$ that divides affine space in two affine halfspaces. Such hyperplane can be either represented as an equation $x + y = b$ or as a normal vector (1,1) as shown in the figure. In 3D space, a 2D plane is a hyperplane and in 2D space, a 1D line is a hyperplane.

2.1.5 Iteration Space Polytope

Under polyhedral terminology, loops are characterized by their iteration space, which can be represented as a system of inequalities. The intersection of finite halfspace or finite number of inequalities $A\vec{x} \leq \vec{b}$ such that $\vec{x} \in \mathbb{R} | A\vec{x} \leq \vec{b}$ forms a polyhedron. A bounded polyhedron is a polytope. Figure 2.4(a) shows a polytope representation (shown as shaded gray area) of a loop iteration in Figure 2.4(b). Such an
iteration space polytope, also known as domain, encompasses information about the bounds of a given loop iteration.

(a) Iteration Space

(b) Loop

Figure 2.4: Iteration Space Polytope

In a loop iteration in Figure 2.4(b), for every \( x, y \in \mathbb{N} \) within the polytope, the intersection is represented by a dotted circle in Figure 2.4(a). These intersections are different points in the iteration space. Since, affine space can be expressed as a solution to a set of linear equations, the iteration space becomes affine.

The iteration space polytope or domain \( D_s : i, j | i, j \in \mathbb{N}, 1 \leq i \leq n \land 1 \leq j \leq n \) of a statement \( S \) can be defined by a system of affine inequalities in the matrix form as,

\[
D_s . \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq 0
\]

Further expanding the matrix, the bounds for the iteration space can be represented as,
\[
\begin{align*}
&i \geq 1 \\
&i \leq n \\
&j \geq 1 \\
&j \leq n \end{align*}
\]
\[
\begin{bmatrix}
1 & 0 & 0 & -1 \\
-1 & 0 & 1 & 0 \\
0 & 1 & 0 & -1 \\
0 & -1 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
n \\
1
\end{bmatrix} \geq 0
\]

2.1.6 Affine Transformation

An affine transformation is an invertible mapping for all points in one affine space to another such that all affine properties are respected. Such transformation preserves points and the relative distance between them in a straight line. Such mappings can be used in a loop iteration to change the execution order of statements.

For a given statement \( S \) with iterators \( \vec{i}_S \) and the global parameters \( \vec{n} \), a one dimensional affine transformation can be represented as,

\[
\phi_S(\vec{i}_S) = T_S \cdot \begin{bmatrix} \vec{i}_S \\ n \\ 1 \end{bmatrix}
\]

where \( T_S \) is a row vector. In the case of d-dimensional affine mapping, such transformation \( T_S \) is represented as a matrix, such that each row represents a affine transform of the form \( \phi^k_S \), where \( 1 \leq k \leq d \).

For example, for a loop iteration with iterators \( i \) and \( j \), using the transformation matrix \( \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \) gives a new schedule as follows,

\[
i' = i + j \\
j' = j
\]

\(^1\) Symbols that are constant within a program representation
Figure 2.5: Scanning with new lexicographical ordering

Figure 2.5(a) shows how using a scanning along hyperplane normal (1,1) changes the scheduling and give an impression of a skewed loop nest. The number in ‘red’ shows the order in which nodes within the iteration space is scanned. Figure 2.5(b) shows how skewed loop nest looks pictorially.

2.1.7 Classical Tiling

Tiling is one of the loop transformation techniques used to exploit reuse at different levels of the memory hierarchy. It brings statements from different iterations together and improve locality. Such technique still has to respect all original dependencies in order to form a legal tile.

2.1.8 Legality for Tiling

For given $k$ statements, in order for a statement-wise hyperplane $(\phi_{s_1}, \phi_{s_2}\ldots\phi_{s_k})$ to be a legal tiling hyperplane, the distance between source $s$ and target $t$ along every dependence edge from $S_i \rightarrow S_j$ has to be strictly positive such that,

$$\text{dist}_{t,s} = \phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0$$  \hspace{1cm} (2.2)
When a combination of $m^2$ hyperplanes, represented by $\phi^1, \phi^2, ..., \phi^m$, form tiles, they are self-contained i.e. dependencies for statements within tiles are either satisfied or can be satisfied inside the tiles.

Figure 2.6 shows an iteration space with two different tiling schemes using different hyperplanes (shown as a dotted line). Figure 2.6(a) shows an example of a legal tiling constructed using hyperplane $(1,0)$ and $(0,1)$. In the figure, we can see that $dist_{t,s} \geq 0$ (condition required for tiling). On the other hand, tiling shown in Figure 2.6(b) uses hyperplane $(1,-1)$ and $(0,1)$. As a result, dependence distance between hyperplanes in some cases is negative. For example $\phi_{i4}$ and $\phi_{i5}$ depends on each other creating circular dependencies.

![Figure 2.6: Legal and Illegal Tiling](image)

For the iteration space polytope $D_s : i, j | i, j \in \mathbb{N}, 1 \leq i \leq n \land 1 \leq j \leq n$ of a statement $S$, when tiled with tile size of 32 using hyperplane $(1,0)$ and $(0,1)$, the resulting tiled iteration space polytope $D_{st} : (I, J | i \geq 32I, i \leq 32I + 31, j \geq 32J, j \leq 32J + 31)$ can be represented as,

\[^2\text{where } m \text{ is less or equal to the number of dimensions of the iteration space.}\]
where  and  are iterators in the tiled domain and  and  are iterators in the original domain.

This can be further expanded in a matrix form as,

\[
\begin{pmatrix}
0 & 0 & 1 & 0 & 0 & -1 \\
0 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & -1 & 1 & 0 \\
-32 & 0 & 1 & 0 & 0 & 0 \\
32 & 0 & -1 & 0 & 0 & 31 \\
0 & -32 & 0 & 1 & 0 & 0 \\
0 & 32 & 0 & -1 & 0 & 31
\end{pmatrix}
\begin{pmatrix}
I \\
J \\
i \\
j \\
n \\
1
\end{pmatrix}
\geq 0
\]

2.1.9 Scheduling Tiled Code

Tiled code scheduling is done in multiple steps using affine transformations in the tiled domain and the original domain. First the tiles are scheduled and then the element nodes within the tiles are scheduled. Let  and  be the iterators in a tiled domain and a original domain respectively. Using a transformation matrix  , the scheduling function  representing the new iterators in the transformed space can be represented as,
\[ \theta_S = T_S \cdot \begin{pmatrix} \hat{I}_{S_l} \\ \hat{i}_S \\ n \\ 1 \end{pmatrix} \]

Figure 2.7 shows two different schedules for a tiled iteration space. In Figure 2.7(a), a tile is scheduled with transformation matrix \( \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \), which implies that the tiled iterators \( I' \) and \( J' \) are,

\[ I' = I \text{ and } J' = J \]

Figure 2.7(b) shows a different schedule where tiles are scheduled with transformation matrix \( \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \). Then, the tiled iterators \( I' \) and \( J' \) in this case are,

\[ I' = I + J \text{ and } J' = J \]

\[ (2.3) \]

Figure 2.7: Two different tile schedules

Similarly, an additional hierarchy of tiles can be added and scheduled for multi hierarchical tiling.

2.1.10 Tiling for Concurrent Start

Not all valid tiling hyperplanes lead to the best scheduling strategy. For some class of applications such as heat equation and Jacobi kernel, when the iteration space
is represented as a polytope, there exist a face such that every element in that face can execute in parallel. For example, in Figure 2.8, all elements along \( i \) can be executed concurrently at different time steps \( t \) in the original iteration space. However, in order to have a concurrent start in the tiled domain, there must exist a face that can be executed concurrently such that its hyperplane normal \( \vec{f} \) carries all dependencies. Valid tiling hyperplanes for this example to create a diamond shaped tile are \((-1,1)\) and \((1,1)\) as shown in Figure 4.6.

```c
for (int i=0; i<N; i++){
  for (int j=1; j<=N; j++){
  }
}
```

**Figure 2.8:** Example with concurrent start

### 2.1.11 Condition for Concurrent Start

Given the set of vectors \( \vec{x}_1, \vec{x}_2, \vec{x}_3 ... \vec{x}_k \), a conical combination is a vector of the form

\[
\sum_{i=1}^{k} \lambda_i \vec{x}_i
\]  

(2.4)

When \( \lambda_i \) is strictly positive, such combination becomes a strict conical combination. By using such a strict conical combination of all hyperplanes \( \phi^1, \phi^2 ... \phi^k \), a hyperplane \( \phi \) representing a face with normal \( \vec{f} \) can be found such that,

\[
\phi = \theta \vec{f} = \sum_{i=1}^{k} \lambda_i \phi^i, \theta, \lambda_i \in \mathbb{N}
\]  

(2.5)

Although, concurrent start can be used to exploit \( n - 1 \) degrees of concurrency, such codes in practice are very complex and don’t provide much performance. Exploiting just one degree of concurrent start using the first two hyperplanes provides partial
concurrent start along one face of the iteration space. Thus, partial concurrent start can be exposed using a simplified version of Equation 2.5:

$$\phi = \lambda_1 \phi^1 + \lambda_2 \phi^2$$  \hspace{1cm} (2.6)

Interested readers are strongly recommended to read reference [12] to get more information about hyperplane selection for diamond tiling.

### 2.2 Memory Layout and Access Function

When the memory layout matches the access pattern, it allows contiguous accesses, resulting in better spatial locality. In contrast, when memory layout conflicts with the access pattern, it results in unnecessary conflicts and evictions in caches resulting in lower performance. For example, in Figure 2.9, assuming row major mapping, read accesses to $A$ are contiguous in memory whereas accesses to $B$ can jump between different memory banks and pages.

```c
for (int i=0; i<N; i++){
    for (int j=1; j<=N; j++){
        A[i][j] = B[j][i] + ...;
    }
}
```

Figure 2.9: Strided Access for B matrix

In cases like this, a loop transformation is not enough to improve locality since only one of the arrays ($A$ or $B$) can have a contiguous access in memory. To optimize such cases, the memory layout can be altered for one of the data structures. Data restructuring maps one iteration space to the other using an invertible transformation matrix. Such a mapping avoids conflicts and ensures that multiple indices are not mapped to the same space.
Accesses to arrays in affine space can be represented using matrices [119, 14]. Using the access matrix $A_m$ in the iteration space $\vec{i}_s$, the access function $A_f$ takes the form,

$$A_f(\vec{i}_s) = A_m \vec{i}_s$$

For example, the access function of array $B$ in Figure 2.9 can be represented as

$$A_{fB}(i) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

Using a transformation matrix $T$, a new access matrix and hence a new access function can be calculated as

$$A'_{f}(\vec{i}_s) = TA_m \vec{i}_s$$

The restructuring space can then be accessed using the newly calculated access function.

In our example in Figure 2.9 where B matrix has a stride of $n$, using a data transpose can convert access stride to 1 in this case. Such transformation can be represented as

$$A'_{fB}(i_s) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

Hence, then new indices for array $B$ becomes $B[i][j]$.

Data restructuring however comes with an overhead of memory space and data movement from one space to the other. Using unimodular transformation, one can ensure one-to-one mapping such that the new space poses no empty slots. However, allocating restructuring space the same size as the original space can be costly both in terms of space and the stress it can incur on the memory subsystem. Partitioning the iteration space into different disjoint sets such that they can be mapped to the same space allows restructuring space to be smaller and more efficient.

In our example in Figure 2.9, each iteration of $j$ is independent of $i$. Hence, for every $i$ iteration, data restructuring can be done for array $B$ where they are all
mapped to the same space of size $N$. In our framework, we use similar approach to reduce restructuring space.

The effectiveness of data transformation relies heavily on the amount of data reuse that can be extracted from the loop iteration. The order of reuse can be calculated using the rank ($R$) of the access matrix and the dimensionality ($D$) of the iteration space. When $D > R$, it implies that there exist a reuse [119, 14].

For example, in Figure 2.10, the iteration space has the dimensionality of 2 and the rank of access for data structure $C$ is 1. Hence each value in array $C$ has $N$ reuse.

```c
for (int i=0; i<N; i++){
    for (int j=0; j<=N; j++){
        A[i][j] = C[i]+...;
    }
}
```

**Figure 2.10:** Access to $C[i]$ has $N$ reuse

Our framework uses hierarchical tiling with multilevel parallelism where threads collaborate to move data and restructure to improve locality and reuse. In the next chapter, we explain our framework in detail.
Chapter 3

GROUP LOCALITY: THE FRAMEWORK

In this chapter, we describe our Group Locality Framework designed to take advantage of temporal and spatial locality for group of threads working in close proximity in time and space. It allows threads to be aware of each other’s execution pattern and improve performance with collaborative data movement and sharing.

Classically, threads are self-centric; Threads move data with single threaded execution in mind. Such approach worked great when caches were not shared, however in multicore architectures such strategies are prone to creating conflicts compromising locality and optimization attempts performed by other threads at different levels of the memory hierarchy. This is mainly because caches nowadays are shared either by physical cores or by hardware threads. Cache line evictions, false sharing and bank conflicts are few examples of such negative impacts. On the other hand, when threads work in close proximity there exists an opportunity for data sharing and reuse across them. To maximize performance, threads need to work together as a unit and help each other orchestrating data movement and performing optimizations to improve the utilization of the entire system. This notion of threads collaboration to maximize locality for group of threads is called Group Locality.

Classically, hierarchical tiles are designed to improve locality by exploiting parallelism at coarser granularity where inner tiles run sequentially. However, in our Group Locality framework, the extraction of collaborative data movement and sharing hinges on hierarchical tiles where parallelism exists at multiple levels. Unlike designs where placement of data and threads are locality agnostic, such design allows locality aware parallel execution. Threads working in close proximity maximize reuse of data among all parallel units before shipping the data back to the memory.
3.1 Notion of Groups

Cores that share the same socket or threads belonging to the same core have an advantage during data sharing because of the physical affinity introduced by the architectural design. Also, memory addresses belonging to the same page or the same bank are cheaper to access since it allows accesses from open banks and pages [99]. Group Locality framework is designed to take advantage of such affinity along with the locality/reuse made available by the application. In our framework, groups are of two types,

- **Thread Groups**: A group of threads \( ThGrp \) formed either by hardware threads within a core or group of cores with physical affinity such that they can collaborate in a shared space (e.g., the last level cache) to improve locality and parallelism as a unit. As a group, they get a larger chunk of task (e.g., L2 tile) where each thread work on set of tiles (e.g., L1 tiles), share data among group members and synchronize mainly using atomic operations. Such design allows locality aware data movement and execution among parallel units.

- **Restructuring Groups**: Group of \( ThGrps \) form \( ResGrp \) and collaborate by creating a shared restructuring space when there exists a strided access pattern with a high order of reuse. All participating \( ThGrps \) perform data layout transformations in parallel that are beneficial for all \( ResGrp \) members. Such a restructuring space is designed to convert strided access patterns to cache and memory friendly contiguous accesses.

3.2 The Framework

The overview of the framework is shown in Figure 3.1. The framework is designed to maximize the use of parallel processing resources while keeping runtime overhead low. It leverages the existing polyhedral tools PLUTO [22] and CLooG [15] for highly parallel tiled code generation as explained in section 3.2.1. Such tiled code is
post processed and modified to a PTHREAD implementation which then uses a data-flow like execution where threads collaborate within groups with minimal overhead as explained in section 3.2.3.

Additionally, the framework provides an opportunity to exploit locality by transforming strided accesses to more cache friendly accesses. Such restructuring is done for data sets with high order of reuse to justify and amortize the cost of such transformation. Section 3.2.4 provides the detail of such approach.

3.2.1 Code Generation

Our framework leverages the existing works for generating highly parallel code: PLUTO, a polyhedral source-to-source parallelizing compiler and CLooG, a code generation tool.

PLUTO was developed by Bondhugula [22]; it takes a C code as an input and transforms it into a coarse-grain parallel OpenMP tiled code that is optimized for data locality. Using its affine transformation framework, it finds transformations efficient for tiling. The legality of such tiling is given by the equation 2.2.
For any source \( s \) and target \( t \) along all dependence edges, PLUTO attempts to minimize the affine function \( \delta \), where

\[
\delta = \phi_S(t) - \phi_S(s)
\] 

(3.1)

Such an affine function gives the number of hyperplanes traveled along any given dependence edge between \( s \) and \( t \) if it were to execute sequentially in time along the hyperplane normal. Hence, it gives the reuse distance between two hyperplanes. Such a hyperplane when used as a space dimension for mapping processors in parallel execution, it gives a rough measure of the communication volume. Minimizing the upper bound of such a cost function, PLUTO calculates legal tiling hyperplanes, which it determines to be communication-minimal.

Once found, the same hyperplanes are then used for multiple levels of tiling targeting different levels of the memory hierarchy. The tool is designed to exploit coarse grain parallelism and hence assumes sequential execution of inner tiles. It uses PipLib [45] as its ILP solver and CLooG for code generation.

The reason for choosing PLUTO in our framework is mainly because it is a well-designed tool capable of automatically generating hierarchical tiled code optimized for communication. Since our framework uses fine-grain execution of inner tiles as shown in section 3.2.3, such minimal communication design is very important. We take advantage of the automatic polyhedral based optimization and add inner parallelism, fine-grain execution and data restructuring for better data locality.

**CLooG:** is a code generation tool that stands for *Chunky Loop Generator.* It was developed by Bastoul [15] for locality improvement. It takes *domain* and *scattering functions* as inputs. Such representations are made statement-wise where the domain specifies the affine constraints of the iteration space in both tiled and original spaces. Scattering functions specify how the iteration space is scanned for code generation in the given domain. It can be viewed as a time stamp for executing each statement. Such functions are chosen with the goal of exploiting parallelism and inducing loop optimizations.
Scanning of the polyhedron is done in a global lexicographic order. Since the code generator is oblivious to any information about the dependencies, the user has to guarantee the validity of the specified schedule to produce correct code. In our framework such validity is ensured by equation 3.2 defined in section 3.2.2.1 which is an additional layer of transformation on top of PLUTO transformation system. In the absence of a scattering function, CLooG scans the polyhedron in lexicographic order as specified by the original iterator.

In summary, PLUTO and CLooG provide an efficient way of generating code automatically and make complex transformations feasible. Interested readers are strongly encouraged to read references [22] and [15] for more information about PLUTO and CLooG respectively.

3.2.2 Hyperplane Communication Analysis

The selection of tiling hyperplanes affects the amount of communication involved during execution in the transformed space. Generally, the goal is to maximize the degree of parallelism and minimize the amount of synchronization between hyperplanes. However, exposing more parallelism often leads to an increase in communication, which can counteract possible gains.

Figure 3.2(a) shows a pictorial view of tiling using hyperplanes (0,1) and (1,0). The same iteration space can also be tiled using tiling hyperplanes (0,1) and (1,1) as shown in Figure 3.2(b). In both case, these tiles can be executed in a pipelined parallel fashion along hyperplane normal (1,1). Despite both being legal tilings, the one in Figure 3.2(b) comes with an increase in communication overhead (volume and reuse distance). This is shown with increase in boldface arrow for one of the tiles in both figures. Similarly, when executed along (1,1) as time step, there is an increase in reuse distance. In contrast with Figure 3.2(a), where orange tiles depend on blue tiles and blue tiles depend on yellow tiles, in Figure 3.2(b), orange tiles depend on both blue tiles and yellow tiles. In his work, Bondhunga did a similar analysis regarding
communicating hyperplanes which is reflected in PLUTO hyperplane selection as it tries to minimize such communication [21].

![Figure 3.2: Tiling with two different legal tiling hyperplanes](image)

An increase in communication volume creates more traffic and could lead to congestion. Also, an increase in communicating hyperplanes and reuse distance implies higher chances of data eviction from caches before it could be reused and more cache conflicts, leaving caches potentially polluted. Our framework is designed to improve inner parallelism where threads can form a group and work as a unit. This can sometime require a selection of hyperplanes that provide parallelism but not minimal communication. In order to balance the two, we use communication-minimal hyperplanes for inner tiles and parallelism exposing hyperplanes for outer tiles. We will discuss this approach further in chapter 4.

3.2.2.1 Extracting Intra-tile Parallelism

Under the PLUTO framework, a scheduling function is specified by the transformation system to maintain the legality of the execution and find the tiling hyperplanes with minimal communication. Our framework takes advantage of the existing transformation framework, parallelization and locality optimization that PLUTO uses for code generation. It uses the tiling hyperplanes generated by PLUTO to create tiles for
the lowest level of the memory hierarchy. It is because the first level tiling hyperplanes are already optimized for communication. Once a first level tiling is done, the second level tiles are created such that at least one face of the outer tile has a concurrent start for inner tiles.

In order to expose such multilevel parallelism, our framework uses at least one hyperplane along which tiles have satisfied dependencies. For clarity, we represent all original hyperplanes by \( \phi \), L1 hyperplanes by \( \varphi \) and L2 hyperplanes by \( \Phi \). Given \( m \) tiling hyperplanes, the framework first creates the L1 tiles. Then, in the tiled domain (at the L1 level), the framework finds at least one hyperplane \( \varphi \) that includes tiles that can be executed in parallel. The condition for such tiling requires that the distance between source \( s \) and target \( t \) along every dependence edge from \( S_i \rightarrow S_j \) for at least one of the tiling hyperplanes has to be strictly positive such that,

\[
\varphi_{S_i}(\vec{t}) - \varphi_{S_j}(\vec{s}) \geq 1 \tag{3.2}
\]

It is true that for some highly parallel applications where communication between legal tiling hyperplanes is non-existent, the above condition is easily satisfied. For example, in case of simple matrix addition example, as shown in Figure 3.3, the legal tiling hyperplanes (1,0) and (0,1) can be used for tiling for multiple hierarchies. Each tile level can still run in parallel because it has non-communicating hyperplanes.

```cpp
for (int i=1; i<=n; i++){
    for (int j=1; j<=n; j++){
        C[i][j] = A[i][j] + B[i][j];
    }
}
```

**Figure 3.3:** Matrix Addition Example

In contrast, many scientific applications have intricate dependencies that normally limit the amount of parallelism that can be extracted at different tile levels. In such cases, the required hyperplane is found by using a conic combination of the first level tiling hyperplanes. Equation 2.2 ensures that the dependencies are non-negative.
along all tiled iterators. The idea is to combine the non-negative hyperplanes to find a hyperplane in the resulting cone that satisfies all dependencies as described in section 2.1.11.

It is theoretically possible to extract multiple degree of parallelism using this technique. However, in practice it produces complex code with many function calls to determine loop bounds. In our framework, we use partial concurrent start as shown with equation 2.6 to extract one degree of parallelism for each level of tiling. Figure 3.4 shows two pictorial examples of how using the conic combination of two positive vectors results in a new positive vector that falls in the cone. In other words, executing along the resultant vector (anywhere in the green zone) satisfies dependencies for all participating vectors. In Figure 3.4(a), execution along vector (1,1) satisfies dependencies along direction (0,1) and (1,0). Similarly in Figure 3.4(b), execution along vector (0,1) satisfies dependencies along vector (-1,1) and (1,1).

**Figure 3.4:** Conic Combination of non-negative hyperplanes to extract parallelism. The green zone shows region along which all dependencies are non-negative.

Such a hyperplane is found using the conic combination of two original tiling hyperplanes, which is used together with the remaining hyperplanes to create L2 tiles. In the case of pipelined parallel applications, such an approach is enough to exploit multilevel parallelism. In contrast, applications with concurrent start require selecting
the right tile shapes and sizes to exploit inner parallelism without compromising the
outer parallelism. We will discuss this further in chapter 4.

3.2.2.2 Scheduling Hierarchical Tiles

The scheduling of tiled code has to be done for each level of tiling. We refer to
L1 tiles as L1 supernodes and L2 tiles as L2 supernodes for the purpose of scheduling
them in their respective domains. First the L2 supernodes are scheduled, followed by
the L1 supernodes. For clarity, we represent scheduling hyperplanes by $\theta$, $\vartheta$ and $\Theta$ in
the transformed space at base, L1 and L2 levels.

Since tiling hyperplanes satisfy equation 2.2, stepping through supernodes along
such tiling hyperplanes satisfies all dependencies. In addition, since our framework
exploits parallelism at multiple levels, it creates outer tiles satisfying equation 3.2.
The scheduling hyperplanes $\theta$, $\vartheta$ and $\Theta$ are representation of tiling hyperplanes in the
transformed space and hence holds properties presented by the original hyperplanes.
Thus, the scheduling is done such that the following conditions are satisfied for at least
one of the hyperplanes at each tiled level:

\[
\vartheta_{S_i}(\vec{t}) - \vartheta_{S_j}(\vec{s}) \geq 1 \quad (3.3)
\]

\[
\Theta_{S_i}(\vec{t}) - \Theta_{S_j}(\vec{s}) \geq 1 \quad (3.4)
\]

Scheduling tiles with the above conditions exposes multilevel parallelism. This
ensures that threads working together in the same group of L2 tiles only need to
communicate among themselves. Such localized communication of inner tiles during
execution is done using a fine-grain framework described in section 3.2.3.

Alternative Approach: One of the advantages of fine-grain execution, however, is its ability to run L2 tiles with partially satisfied dependencies by scheduling
outer tiles with the following condition:

\[
\Theta_{S_i}(\vec{t}) - \Theta_{S_j}(\vec{s}) \geq 0 \quad (3.5)
\]
Figure 3.5: Inter tile communication (L2) due to partially satisfied dependencies. Scheduling using Equation 3.5 (alternative approach).

This schedules L2 supernodes that have only partial dependencies satisfied and hence depends on each other during execution. Such approach requires dependence updates across L2 tiles as shown in Figure 3.5. On one hand, it allows L1 tiles to execute as soon as dependencies are satisfied, whereas on the other hand it can lead to increase in communication volumes. In this thesis, we focus mainly on tiles with fully satisfied dependencies and leave the latter for future work.

3.2.3 Fine-Grain Execution

Once parallelism is exposed at multiple levels, one of the major challenges is to execute such design with minimal overhead. Using synchronization mechanism like barriers can be very costly when such a construct has to be placed at multiple levels. It leads to a communication overhead that can have a negative impact on bandwidth and performance. Programming models like OpenMP place implicit barriers when intratile parallelism is exposed which in most cases hinders the overall performance.
Our framework takes the OpenMP code generated by PLUTO and converts them into a pthread implementation to reduce the synchronization overhead. The framework assigns *thread ID and Group ID* to each thread. Based on the given *Group ID*, each group of threads fetch a L2 task ready for execution. Inside each L2 tile, the framework uses a data-flow inspired low overhead dependency and task update scheme based on bit masking that enables multiple threads to work synchronously within a tile. The outer tiles synchronize either with barriers or with task level updates.

![Fine-Grain Execution Diagram](image)

**Figure 3.6:** Fine-Grain Execution

Dependencies among the lowest level tiles are represented by different sets of bits, which are collectively updated by a group of threads working together inside the highest-level tile. Each thread performs bitwise operations on the dependency masks and creates a task mask that represents all the tasks ready to execute. If the task mask is non-zero, the thread finds the task, executes, updates dependencies and updates the task as shown in Algorithm 1. This happens in a highly parallel fashion.
such that every thread is aware of the status of the tasks within the assigned tile. The implementation of this approach of synchronization between threads is done solely using atomic operations to keep the overhead low. Figure 3.6 shows the overview of our fine grain execution approach.

Algorithm 1 Fine-Grain Execution of L1 tasks

1: procedure EXECUTE_L2_TASK(groupID, threadID, L2taskIndex ...)
2:     Get initial dependencies
3:     Get current taskmask \(\Rightarrow\) done by bit-ANDing dependencies
4:     Count number of ready tasks \(\Rightarrow\) counts high bits
5:     while number of task \(\neq 0\) do \(\Rightarrow\) we have L1 tasks to execute
6:         L1index \(\leftarrow\) Get L1 index within the L2 tile
7:         if L1Index then
8:             EXECUTE_L1_TASK()
9:             Update Dependencies
10:        end if
11:     Get current taskmask
12:     Count number of ready tasks
13: end while
14: end procedure

Based on the availability of the processing resources multiple groups of threads participate in fine grain execution concurrently. This exploits both locality and parallelism. Additionally, multiple groups of threads come together to take advantage of locality further by restructuring data to improve cache residence.

3.2.4 Data Restructuring and Reuse

Our restructuring framework is designed to change access patterns for data sets that have strided accesses and a high order of reuse. A statement is said to have a high order of reuse when a rank of a access matrix, which is given by the access dimensionality, is less than the depth, i.e., the iteration space dimensionality where the statement is accessed. When \(\text{depth} > \text{rank}\), it implies that there exists a reuse in the iteration space.

For example, in Figure 2.10, access matrix of \(A\) is \(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\) and access matrix of \(C\) is \(\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\). Hence, the rank of the matrices \(A\) and \(B\) are 2 and 1 respectively. They are
accessed in a two-dimensional iteration space and hence have a depth of 2. Thus, \( A \)
has no reuse whereas \( C \) has a total of \( N^2 - N \) reuse in the given iteration space.

Our framework creates a restructuring space that is shared by different \( ThGrps \). Based on the memory access pattern and the reuse information, data is moved to the preallocated restructuring space by all participating parallel threads where it undergoes restructuring to improve its access patterns. Once restructured, when the same data set is accessed again, threads take advantage of the newly found data alignment, improving its access pattern efficiency. The data transformation is done:

- To change strided access patterns into contiguous accesses.
- To bring all elements within a tile to fit together in a contiguous memory space.

Such a design allows data to reside in caches longer as cache conflicts are reduced. However, restructuring by nature comes with an overhead of data movement and extra memory space. In order to reduce such an overhead, the restructuring space is made much smaller than the original space where disjoint data sets can be moved at different points in time. Also, restructuring space is allocated as a contiguous block of memory to allow data to fit better in caches. Figure 3.7 shows two possible ways a restructuring space can look like and how data can be placed in it. In our framework, we use the latter (restructuring 2) to present our restructuring strategy and calculate the new indices.

One of the techniques that is often used to map a larger space to a smaller one is by subtracting the lower bounds of the partition such that the new indices match the bounds of the restructuring space [14]. Figure 3.8 shows an example of such a remapping along with a localized transpose (tile granularity) during data movement. In the figure, \( i \) and \( j \) are the original indices; \( I', J', I'' \) and \( J'' \) are the indices after data transformation. The tiles of interest undergoing transformations are \( A, B \) and \( C \). The lower bounds of the space in the tiled domain is represented by \( LB_k \) in \( k \) and \( LB_j \) in \( j \) dimensions.
The figure shows that in the first step, lower-bounds are subtracted to calculate the transformed indices. In the second step, elements within each L2 tiles are transposed to place all elements of the tile within close proximity in the physical space with a contiguous access pattern. Such an approach reduces the chances of conflicts in caches, pages, banks, and also allows the prefetcher to perform better. As seen in this example, such a design normally requires expensive modulus operations.

Our framework thus introduces an effective restructuring technique that translates such mapping to a offset calculation relative to the original indices. Mapping data for restructuring is done using a sequence of transformations, which are based on initial access patterns, memory layout, and the size of the allocated restructuring space. The transformation is done first in a tiled domain, which is translated to a one step transformation by calculating the offset.

### 3.2.4.1 The Approach

Let $\vec{i}_S$ and $\vec{I}_S$ be the iterators in the original and the tiled domain. Also, let $LB$ be the lower bounds of the partitions in the tiled domain along $\vec{I}_S$, where $LB = (lb_1, lb_2, ... lb_n)^T$ for an n-dimensional space.

Subtracting the lower bounds, the index $\vec{I}'$ in the transformed space takes the
Figure 3.8: Subtracting lower bounds to map data elements to the restructuring space
form,

\[ \vec{I}' = \vec{I} - LB \]  \hspace{1cm} (3.6)

Using further transformation \( T_{TD} \) in the tiled domain (if needed, shown with examples in Chapter 5), the new index \( \vec{I}'' \) becomes,

\[ \vec{I}'' = T_{TD} \vec{I}' \]  \hspace{1cm} (3.7)

In order to compute the new indices, relative to the original indices, we add \( \vec{I} - \vec{I}' \) such that,

\[ \vec{I}'' = \vec{I} + T_{TD} \vec{I}' - \vec{I} \]  \hspace{1cm} (3.8)

Relative to the original indices, the term \( T_{TD} \vec{I}' - \vec{I} \) becomes the offset in the tiled domain. It can be translated to calculate the offset to the original indices by multiplying the tile level offset with the tile size \( ts \) where \( ts \) is a matrix of size \( n \) of the form:

\[
\begin{pmatrix}
ts_1 & 0 & 0 & 0 \\
0 & ts_1 & 0 & 0 \\
.. & .. & .. & .. \\
0 & 0 & 0 & ts_n
\end{pmatrix}
\]

such that,

\[ \vec{i}' = \vec{i} + ts(T_{TD} \vec{I}' - \vec{I}) \]  \hspace{1cm} (3.9)

Finally, the transformation \( T_{elem} \) is done in transformed space at element granularity to bring data to contiguous memory space. The entire transformation can then be represented as,

\[ \vec{i}'' = T_{elem}(\vec{i} + ts(T_{TD} \vec{I}' - \vec{I})) \]  \hspace{1cm} (3.10)

Figure 3.9 shows an example of such offset calculation. In the figure, three tiles \( A, B \) and \( C \) are mapped to the smaller space of size 9*3 with same results as Figure 3.8. The steps can be summarized as,
Subtracting Lower Bounds + Tile transpose

\[ J' = K - LB_k \]
\[ K' = J - LB_j \]

equivalent to

\[ J + K - LB_j - K \]
\[ J + K - LB_k - J \]

Translating tile level offset to element level by multiplying by tile size

\[ j'' = j + (K - LB_k - J)^*ts_k \]
\[ k'' = k + (J - LB_j - K)^*ts_k \]

Figure 3.9: Restructuring using offset relative to the original index
• Subtract the lower bounds $LB_k$ and $LB_j$ and transpose at the tile level.

• Calculate the tile indices in the transformed space relative to the original tile indices.

• Multiply the offset by tile sizes.

• Transpose in the transformed space at the element level.

The overall transformation and the calculation of new indices $i'$ and $j'$ in the transformed space in this example can be shown as,

$$\begin{pmatrix} j'' \\ k'' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} j \\ k \end{pmatrix} + ts_j \begin{pmatrix} 0 & 1 \\ 0 & ts_k \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} J \\ K \end{pmatrix} - \begin{pmatrix} J \\ K \end{pmatrix}$$

At the current state, our framework uses such technique only for read-only data. Chapter 5 provides the details about transformation sequences and the conditions. Our goal is to exploit the parallelism to the full extent that is available for a given architecture. With such approach, threads can work in a collaborative fashion reducing contention and improving overall performance.

### 3.3 Conclusion

In this chapter we have provided a general overview of the Group Locality framework. Such design allows threads to take full advantage of available resources without compromising parallelism. Chapter 4 and 5 provide the detail of our approach.
Chapter 4

JAGGED TILING FOR PARALLELISM

In this chapter, we discuss a novel methodology that takes into consideration multithreaded many-core designs to better utilize memory/processing resources and improve memory residence on tileable applications. It takes advantage of polyhedral analysis and transformation in the form of PLUTO [22], combined with a highly optimized fine grain tile runtime to exploit parallelism at all levels. This chapter includes the introduction of a multi-hierarchical tiling technique that increases intra tile parallelism. We present two different variations of such technique; one for pipelined parallel applications and the other for parallel start applications. We demonstrate our technique on selected representative stencil kernels.

4.1 Overview

As shared resources such as memory bandwidth and cache sizes increase at much slower pace than processing resources such as cores and threads, maximizing the utilization of these resources is very important to achieve higher performance. Compiler optimization techniques such as loop transformations have proven to be very effective. Such techniques improve reuse of data in multiple cache hierarchies and help to hide memory latency. In addition, loop transformations can also be performed to maximize parallelism and load balance. However, when there exist complicated loop carried dependencies, such transformations still operate under the paradigm that coarse-grain parallelism is the norm for many scientific applications.

Stencil computations are a computational intensive class of kernels that are used in many scientific and engineering codes. For some of these kernels, the computations are iterated over time until a solution is found. Since stencil computations involve
calculating values using neighboring elements, there are plenty of opportunities for data reuse. However, when stencil applications are tiled, the dependencies flow across neighboring tiles. In lower dimensionality stencils, the amount of communication required is low. Thus, the interactions between the tiles do not greatly affect the reuse of the application, i.e., the stencil elements needed for calculation are a short stride away. However, in higher dimensionality stencils, the interactions between the elements involved are farther and farther away. This greatly affects reuse since their accesses affect the memory hierarchy behavior.

With the axiom that the base optimization technique used would be tiling, memory reuse needs to be considered in both the inter, as well as, the intra tile so that the strain or underuse of resources can be alleviated. Classical tiling techniques encourage coarse grain parallelism with sequential execution of the inner tiles when tiled hierarchically. However, given the enormous amount of processing power, can we do better and utilize available resources while keeping communication overhead relatively low? During parallel execution, multiple threads work in close proximity in time and space and hence provide opportunities for reuse not only for themselves but also for neighboring threads. In this chapter, we attempt to explore this with a novel tiling technique that enables threads to work in a collaborative fashion for two classes of applications:

- Applications with *Pipelined Start* where execution starts with one node and progresses in a wavefront fashion, e.g., Gauss-Seidel Iteration.

- Applications with *Concurrent Start* where execution can start in parallel in at least one of the dimensions, e.g., Jacobi Iteration.

Such a technique maximizes parallelism and improve locality for threads working together as a group in a highly synchronous fashion.
4.2 Jagged Tiling for Pipeline Start Applications

There is a class of stencils where execution starts at a single node and spreads as a wavefront creating a pipeline parallelism. Because of the nature of these applications, it is very difficult to exploit inner parallelism when such applications are hierarchically tiled. In architectures that offer many parallel units where balancing parallelism and locality becomes more important, it creates a unique challenge. For example, for the simple for-loop in Figure 4.1, a classical approach produces tiling as shown in Figure 4.2. This is a very efficient way of tiling as it is designed for minimal communication and coarse-grained parallelism. When tiled for one level of memory hierarchy as shown in Figure 4.2(a), it maximizes parallelism but does not fully take advantage of locality offered by multi-level caches. Similarly, when tiled hierarchically as shown in Figure 4.2(b), it does not fully exploit parallelism of the inner tiles.

```c
for (int i=1; i<=n; i++){
    for (int j=1; j<=n; j++){
    }
}
```

**Figure 4.1:** A Stencil Example

![One Level Tiling](image1.png) ![Two Level Tiling](image2.png)

**Figure 4.2:** One Level and Hierarchical (Two-Level) Tiling
4.2.1 Our Approach

As discussed in Chapter 3, our framework uses the tiling hyperplanes generated by PLUTO for the lowest level memory hierarchy, i.e., L1 cache. These tiling hyperplanes are designed to be communication minimal, using the cost function shown in equation 3.1, which reduces the communication distance and volume. Under the PLUTO framework, these hyperplanes are used for tiling multiple level of memory hierarchy. However, using the same tiling hyperplanes for both levels come at the cost of sequential or pipeline parallel execution of inner L1 tiles as shown in Figure 4.3(a).

In our framework, we solve this by constructing outer tiles in which at least one face of the polytope has a concurrent start. Given \(m\) tiling hyperplanes, first we create L1 tiles using the given tile sizes. Then we create at least one parallel hyperplane in the L1 tiled domain and use it together with the rest of the tiling hyperplanes to create the L2 tiles. Again for clarity, we represent all original hyperplanes by \(\phi\), L1 hyperplanes by \(\varphi\) and L2 hyperplanes by \(\Phi\). For \(m\) tiling hyperplanes \((\varphi^1, \varphi^2...\varphi^m)\), the condition for such tiling is shown in equations 4.1 and 4.2.

\[
\varphi^1_S(\tilde{t}) - \varphi^1_S(\tilde{s}) \geq 1 \quad \text{for one hyperplane} \quad (4.1)
\]
\[ \varphi^l_{S_i}(\vec{t}) - \varphi^l_{S_j}(\vec{s}) \geq 0 \quad \text{where} \quad 1 < l \leq m \] (4.2)

In order to satisfy Equation 4.1, we use the conic combination of first two hyperplanes, i.e., \( \varphi^1 = \varphi^1 + \varphi^2 \) to create a parallel hyperplane. Such hyperplane \( \varphi^1 \) is then used with rest of the hyperplanes, i.e., \( \varphi^2 \ldots \varphi^m \) to create the outer L2 tiles. The iteration space matrix representing the Domain is updated to reflect the new L2 tile domain. Also the scattering functions are updated to mark outer and inner parallel tiles as shown in Algorithm 2. The code generation is then done using CLooG, the code generation tool discussed in chapter 3. Figure 4.3(b) shows the pictorial view of such jagged tiles.

For example, for our stencil example in Figure 4.1, the communication minimal tiling hyperplanes generated by PLUTO are (1,0) and (0,1). Using these hyperplanes, we create L1 tiles along iterators \( i \) and \( j \) with the given tile size of 32. This creates supernode iterators \( I \) and \( J \) since tiles are viewed as supernodes in the tiled domain. Then using the conic combination of hyperplanes (1,0) and (0,1), a hyperplane (1,1)\(^1\) is created. Such hyperplane is then used with hyperplane (0,1) to create L2 tiles using the tile size of 8 (which gives L2 size \( 8 \times 32 \)). The resulting scattering functions for L2 tiles \( (c_{1L2}, c_{2L2}) \), L1 tiles \( (c_{1L1}, c_{2L1}) \) and the original domain \( (c_1, c_2) \) are updated and used to create parallel code. The Domain and Scattering functions produced by this process are shown below\(^2\).

\(^1\) The parallel hyperplane

\(^2\) Where \( n \) is the size of a dimension in the iteration space. For our example, both dimensions are the same
Domain Scattering

\[1 \leq i \leq n - 1\]
\[c_{L2} = I_{L2} + J_{L2}\]
\[1 \leq j \leq n - 1\]
\[c_{L2} = J_{L2}\]
\[32I_{L1} \leq i \leq 32I_{L1} + 31\]
\[c_{L1} = I_{L1} + J_{L1}\]
\[32J_{L1} \leq j \leq 32J_{L1} + 31\]
\[c_{L1} = J_{L1}\]
\[8I_{L2} \leq I_{L1} + J_{L1} \leq 8I_{L2} + 7\]
\[c_{L2} = i\]
\[8J_{L2} \leq J_{L1} \leq 8J_{L2} + 7\]
\[c_{L2} = j\]

Algorithm 2 Generating Jagged Tiles for Pipeline Start

Input: Given:
(a) Original Domain \(D_{S}\)
(b) L1 tile sizes \(tL_{11}, tL_{12}...tL_{1m}\)
1: Get communication minimal hyperplanes \(\varphi_{S}^{1}, \varphi_{S}^{2}...\varphi_{S}^{m}\) using PLUTO
2: Tile for L1 using \(\varphi, D_{S}, TS_{1}\)
\[\therefore\] At this point, all \(\varphi_{S}\) are created
3: Update Domain constraint to get hyperplane \(\varphi_{S}^{1} \rightarrow \varphi_{S}^{1} + \varphi_{S}^{2}\) such that \(\varphi_{S(i)}^{1} - \varphi_{S(i)}^{1} \geq 1\) leaving other hyperplanes as is
4: Tile for L2 using \(\varphi, D_{S}, TS_{2}\)
\[\therefore\] At this point, all \(\Phi_{S}\) are created
5: Perform Unimodular transformation on L1 scattering supernode: \(\vartheta_{S}^{1} \rightarrow \vartheta_{S}^{1} + \vartheta_{S}^{2}\) to extract inner parallelism (Equation 2.6).
6: Perform Unimodular transformation on L2 scattering supernodes: \(\Theta_{S}^{1} \rightarrow \Theta_{S}^{1} + \Theta_{S}^{2}\) to extract outer parallelism (Equation 2.6).

Output: Updated domain and scattering function

In the case of higher dimensional iteration space, we use the conic combination of the first two hyperplanes as shown in Algorithm 2. This is because we only need one degree of parallelism at every tile level. For example, for Gauss-Seidel-2D example in Figure 4.4 where three dimensions are involved (including time), the domain and the scattering function for tile sizes \(32 \times 32\) (L1) and \(256 \times 256\) (L2) takes the form,
\begin{align*}
\text{Domain} & \quad \text{Scattering} \\
1 \leq t \leq n-1 & \quad c_{1L2} = T_{L2} + I_{L2} \\
1 \leq i \leq n-1 & \quad c_{2L2} = I_{L2} \\
1 \leq j \leq n-1 & \quad c_{3L2} = J_{L2} \\
32T_{L1} \leq t \leq 32T_{L1} + 31 & \quad c_{1L1} = T_{L1} + I_{L1} \\
32I_{L1} \leq i \leq 32I_{L1} + 31 & \quad c_{2L1} = I_{L1} \\
32J_{L1} \leq j \leq 32J_{L1} + 31 & \quad c_{3L1} = J_{L1} \\
8T_{L2} \leq T_{L1} + I_{L1} \leq 8T_{L2} + 7 & \quad c_{1} = t \\
8I_{L2} \leq I_{L1} \leq 8I_{L2} + 7 & \quad c_{2} = i \\
8J_{L2} \leq J_{L1} \leq 8J_{L2} + 7 & \quad c_{3} = j
\end{align*}

\textbf{Figure 4.4:} Gauss-Seidel-2d Example

This is a straightforward approach as it involves creating outer tiles with the collection of parallel first level tiles. However, there exist many stencils with different computational intensities, and many of them feature an execution in which an entire dimension of the iteration space can be executed concurrently. In such cases, exploiting multi-level parallelism requires careful consideration in selection of tile shapes and sizes. We discuss our approach for such applications in section 4.3.
4.3 Jagged Tiling for Concurrent Start Applications

Stencils with concurrent start have been heavily studied to maximize parallelism and load balance such that processing resources are fully utilized. Techniques like diamond tiling \[90, 12\], split and overlapped tiling \[78\], were created to take advantage of such feature. However, so far such techniques have not been able to fully exploit locality at multiple levels of the memory hierarchy.

```c
for (int t=0; t<T; t++){
    for (int i=1; i<=N; i++){
    }
}
```

**Figure 4.5:** Heat-1d Example

Figure 4.5 shows a for loop for one dimensional heat equation with the time dimension added in the array structure. We will use this simplified version of the heat-1d loop as a running example to explain our technique. Figure 4.6 shows a very efficient diamond tiling technique that maximizes parallelism. Although very efficient, such technique does not incorporate the concept of multiple levels or groupings of threads that can take advantage of each other’s data movement. The reason for this is two fold. Firstly, when tiled hierarchically, such tiling technique does not exploit intra-tile parallelism as shown in Figure 4.7(a). Secondly, when thread assignment is oblivious to the possibility of locality and reuse, threads are scheduled far enough in memory to miss out on locality opportunities across threads.

Also, using a straightforward approach of creating outer tiles from parallel diamond tiles does not give multi-level parallelism. Figure 4.7(b) shows that such approach leads to circular dependencies where set of outer tiles depend on each other hindering parallelism.

In order to solve such issues, we introduce a novel jagged tiling technique for
Figure 4.6: Diamond Tiling for Heat-1d

(a) Hierarchical diamond tiling

(b) Outer tiles created with parallel diamond tiles

Figure 4.7: Two different approaches to creating two level tiles
applications with concurrent start without compromising parallelism. Figure 4.9 provides a pictorial view of the jagged tiling for the kernel shown in Figure 4.5. It uses a hierarchical approach of tiling in which outer tiles are designed to take advantage of the highest level\(^3\) of the memory hierarchy, i.e., L2 cache. Threads working within L2 tiles thus have an opportunity to share data with each other and maximize data reuse. However, unlike the jagged tiles shown in Figure 4.3(b), in this case, the tile sizes differ across two given dimensions giving it a rectangular structure. We explain why such shapes are need in section 4.3.1.

![Figure 4.8: Tile Sizes for Jagged Tiling](image)

4.3.1 Tile Shapes and Sizes

Standard diamond tiling uses symmetric diamond shaped tiles that enable concurrent execution along at least one face of the iteration space. Although very effective for load balance and parallelism, such tiles are not able to provide intra-tile concurrent start when tiled hierarchically as shown in Figure 4.7(a). Since jagged tiling is designed to provide a locality aware concurrent start with intra-tile parallelism to further improve data locality, we provide a simple solution to solve this problem.

In Figure 4.8 (which shows two L2 tiles extracted from Figure 4.9), let the tiled iteration space be \( t \) and \( i \) and size of a L1 tile be \( x \) and \( y \) such that \( y = nx \). This implies

\(^3\) Farthest from the core, however we refer it mostly as L2 cache since we use two level tiling in our examples
Figure 4.9: Jagged Tiling for Heat-1d. Tiling hyperplanes (1,-1) and (1,1) (denoted L1 Hyperplane) in original domain and (1,1) and (1,0) (denoted L2 Hyperplane) in the tiled domain.

that it creates a rectangular polygon in which if one side has height $h$ (towards x), the opposite side has height $n \times h$. If we create jagged L2 tiles with $j_t$ and $j_i$ number of L1 tiles in $t$ and $i$ direction using Algorithm 3, two consecutive jagged tiles must stand at the same level, or for this example at the same height, to be able to provide concurrent start. Let $H1$ and $H2$ be the level at which two consecutive tiles start. Then,

$$H1 = j_t \times h$$

$$H2 = j_i \times (n \times h - h) = j_i \times h(n - 1)$$

For concurrent start $H1$ has to be equal to $H2$, hence

$$j_t \times h = j_i \times h(n - 1)$$

$$j_t = j_i(n - 1), \ h \neq 0 \tag{4.3}$$

When $n = 2$, $j_t = j_i$. This means that when $y = 2 \times x$, the number of L1 tiles required to form L2 tiles has to be equal in both direction (e.g, 3x3 L1 tiles in Figure 4.9).
Although, other sizes can be extracted using Equation 4.3, we use L1 tiles in which one side is double the size of the other.

4.3.2 Our Approach

Jagged tiles are generated using an enhancement of PLUTO created to generate code for load balance, e.g., diamond shaped tiles for concurrent start [12]. We create L1 tiles using the approach used in PLUTO for diamond tiling. It uses an iterative scheme to find hyperplanes [12] that satisfy Equation 2.5. We leverage their method while using a different tile shape, i.e., \( y = 2x \) that satisfies equation 4.3. Once L1 tiles are created, given \( m \) L1 hyperplanes, we use the condition given by Equation 2.6 to expose at least one hyperplane \( \varphi^1 = \varphi^1 + \varphi^2 \) with concurrent start such that equation 4.1 is satisfied. We use such hyperplane \( \varphi^1 \) with rest of the tiling hyperplanes \( \varphi^2...\varphi^m \) to create L2 tiles. The algorithm to generate such jagged tiles is shown in Algorithm 3.

**Algorithm 3** Generating Jagged Tiles for Concurrent Start

**Input:** Given:
(a) Original Domain \( D_S \)
(b) L1 tile sizes \( tL_{11}, tL_{12}...tL_{1m} \) (satisfying tile size condition from Equation 4.3 for \( tL_{21}, tL_{22} \))

1: Get **diamond tiling** hyperplanes \( \phi^1_S, \phi^2_S...\phi^m_S \) using PLUTO
2: Tile for L1 using \( \phi, D_S, TS_1 \) \n   \n   \( \triangleright \) At this point, all \( \varphi_S \) are created
3: Update Domain constraint to get hyperplane \( \varphi^1_S \to \varphi^1_S + \varphi^2_S \) such that \( \varphi^1_{S(i)} - \varphi^1_{S(a)} \geq 1 \) leaving other hyperplanes as is
4: Tile for L2 using \( \varphi, D_S, TS_2 \) \n   \n   \( \triangleright \) At this point, all \( \Phi_S \) are created
5: Perform Unimodular transformation on L1 scattering supernode: \( \vartheta^1_S \to \vartheta^1_S + \vartheta^2_S \) to extract inner parallelism (Equation 2.6).
6: Perform Unimodular transformation on L2 scattering supernodes: \( \Theta^1_S \to \Theta^1_S + \Theta^2_S \) to extract outer parallelism (Equation 2.6).

**Output:** Updated domain and scattering function

For example, let us assume that we want to create a jagged tile for Figure 4.5 with tile sizes 1024 x 2048 for L1 and 4x4 (L1 tiles) for L2. The tiling hyperplanes found by PLUTO are (1,1) and (1,-1). We use these hyperplanes to create inner L1
tiles with tile size 1024 and 2048 along iterators $t$ and $i$. These tiles can be viewed as supernodes for next level tiling and can be used to create outer L2 tiles using the supernode iterators $T$ and $I$. The domain with these supernode iterators becomes the tiled domain, where using hyperplanes (0,1) and (1,0) as tiling hyperplanes in tiled domain results in the same hyperplanes as the ones used to create Level 1 tiles. Instead, we use hyperplanes (1,1) (generated with conic combination, which satisfies the partial concurrent start condition) and (1,0) with a tile size of 4. Once the tiles are created and scattering functions for L2 tiles ($c_{1L2}, c_{2L2}$), L1 tiles ($c_{1L1}, c_{2L1}$) and the original domain ($c_1, c_2$) are updated, the parallel code is generated using CLooG. The resulting Domain and Scattering function for heat-1d is shown below.

\[
\begin{align*}
\text{Domain} & \quad \text{Scattering} \\
0 \leq t \leq T - 1 & \quad c_{1L2} = T_{L2} + I_{L2} \\
1 \leq i \leq N - 1 & \quad c_{2L2} = T_{L2} \\
1024T_{L1} \leq t - i \leq 1024T_{L1} + 1023 & \quad c_{1L1} = T_{L1} + I_{L1} \\
2048I_{L1} \leq t + i \leq 2048I_{L1} + 2047 & \quad c_{2L1} = T_{L1} \\
4T_{L2} \leq T_{L1} + I_{L1} \leq 4T_{L2} + 3 & \quad c_1 = t \\
4I_{L2} \leq I_{L1} \leq 4I_{L2} + 3 & \quad c_2 = t + i
\end{align*}
\]

### 4.4 Fine-Grain Execution

In this section, we provide a brief description of a fine-grain execution for the examples we used in this chapter. With the goal of taking advantage of locality across threads we combine threads into different groups. Each group of threads work together within the L2 tile taking advantage of intratile parallelism, L2 locality and sharing data brought into caches by threads within the group as explained in section 3.2.3.

Stencil computations have very regular dependencies and are repeatable across all L2 tiles. In other words, for any given L2 tile ready to execute, the same face
Figure 4.10: Fine Grain Execution Example

Figure 4.11: Dependencies Example for Jagged Tiling. Green arrows show satisfied dependencies
Figure 4.12: Dependency bits representation example. Top binary number representing dependencies status and lower decimal number representing bit index. Dependency 1 (0x111F) and Dependency 2 (0xF) (represented by L1 tiles) in its iteration space are ready to execute. To simplify the execution, we represent tiled domain dependencies by a set of bits which are collectively updated by a group of threads working together within the L2 tile. For example, in the case of our example stencil in Figure 4.1, at the beginning of each L2 tile (4x4 L1) execution, Dependency 1 is represented by bitmask 0x111F and Dependency 2 is represented by bitmask 0xF. Figures 4.11 and 4.12 show how such representation looks pictorially for 4x4 tiles. Each thread performs atomic bit-wise operations to create a task mask (initially 0xF in this case). Every nonzero bit in the task mask represents a L1 task ready to execute. Such execution happens in a highly parallel fashion and all required updates are done using atomic operations to minimize synchronization overhead. Figure 4.10 shows a pictorial view of such execution for jagged tiling with 3x3 tiles.

4.5 Convexity Analysis

One of the peculiarities of jagged tiling is the appearance of concaveness in the outer tile. However, this is because hierarchical tiling in done in two steps. First the inner tiles are created, which then becomes supernodes for the tiled domain. By view of iteration space from one level up such that tiles become individual node
contrast with classical hierarchical tiling where the same hyperplanes are used for both level tiles, jagged tiling uses different hyperplanes and thus gives an impression of loss of convexity. However, in the tiled domain, it still respects all properties of convexity and affine space needed for transformation.

4.6 Conclusion

In this chapter, we have provided a novel tiling technique that exploits multiple levels of parallelism for two classes of stencil applications that are very important to the scientific community. Such technique allows threads to take advantage of the data movement done by other threads working in close proximity and improves data reuse at different levels of the memory hierarchy. In the next chapter, we explore how restructuring data and making accesses contiguous can exploit data locality further.
In multicore systems, shared resources such as memory and bandwidth are limited. Improving utilization of such resources is crucial to improve performance of the overall system. This requires maximizing data reuse in caches and reducing memory and bandwidth contentions. Because of this reason, in parallel programs, the focus is placed mainly on maximizing reuse on single processor and reducing false sharing across parallel units. This is a very good approach to achieve good performance as it avoids unnecessary conflicts that can have negative impacts on performance.

In multithreaded environment, when data is shared across many parallel units, there exists opportunities to improve performance further by considering data reuse and their access pattern across threads. Where contiguous access pattern can improve performance of all participating parallel units, strided access pattern do the reverse. In this chapter, we discuss our restructuring framework designed to improve locality and reuse across parallel threads by restructuring the data to allow contiguous access. So far in previous chapters we have shown that grouping of threads improve locality and reuse for multiple threads working in close proximity. In this chapter, we extend that approach to data movement and restructuring to show how it can add more to group benefits.

5.1 Overview

The physical proximity of processing cores to the memory and the access pattern of an application significantly affects the performance of its memory subsystem. When access pattern is contiguous, data accesses mostly fall into the same cache line, memory page and the memory bank. The end effects are lower page misses, lower bank conflicts
and better cache utilization. This also leads to higher spatial locality resulting in higher bandwidth and less network congestion.

Similarly, predictable access patterns allow hardware prefetchers to move required data a priori to their actual access. Such behavior allows higher utilization of low latency caches. However, when access patterns are unpredictable or have higher access strides, they tax the underlying hardware and quite often lead to access conflicts and misses at various levels of the memory hierarchy. These conflicts lead to premature cache evictions and consequently, high access penalties at every eviction and reuse point.

As an example, consider a classical matrix multiplication as shown in Figure 5.1. Assuming row major storage, matrix A has contiguous access in memory and hence has better spatial locality as compared to matrix B access that has a stride of n. Since this example is known to have plenty of access reuse – $n^3$ times on $n^2$ data – maximizing reuse in low latency caches to improve performance is of importance. The classical solution to maximize reuse involves tiling for the one-threaded case. In general, though, access penalties arise in loop parallelized code when multiple threads contend over a single tile within the limited confines of the cache. In our example, the parallelized, tiled loop across the i dimension shares column tiles in B where the penalty of strided access is paid again and again for each and every access.

```c
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    for (k=1; k<=n; k++)
      C[i][j] += A[i][k] * B[k][j];
```

**Figure 5.1:** Matrix Multiplication

In a different scenario, an application can have different access patterns at different points in space and time with multiple references to the same data structure. For example, let us look at an interesting access pattern that can be observed in the LU decomposition code shown in Figure 5.2. This code factorizes a matrix into lower
triangular and upper triangular matrices (and, depending on the variant used, a third "Permutation" matrix) through a series of ever smaller steps. During this process, there are \((n-1)\) rank-one updates of the data matrix, reduced by one in every sweep. The last phase of the update for LU is a matrix multiply operation as seen in the code excerpt in Figure 5.2. With a blocked version of the LU decomposition code, the execution can be described pictorially by Figure 5.4. Moreover, every third phase in the figure is a matrix multiply operation on the block. This exposes the same type of issues that plagued the previous matrix multiplication algorithm. This example has the additional property of multiple access patterns. The same data structure is accessed along row and column at the same time in two different locations in \(A[i][k]\) and \(A[j][i]\) respectively (visible better in blocked version shown in Figure 5.3). That means there does not exist one mapping that allows array \(A\) to be contiguous along all iterations.

```cpp
for (i=1; i<n; i++){
    for (j=i+1; j<n; j++)
        S1: \(A[j][i] = A[j][i]/A[i][i]\);
    for (j=i+1; j<n; j++)
        for (k=i+1; k<n; k++)
}
```

**Figure 5.2:** LU Decomposition

A potential solution is to compute a result tile as an outer product and to use an element of \(A\) (in the left column sub-block in step 2 of Figure 5.2) and multiply it with the row of \(A\) (in the top row sub-block in step 2) to produce a row of \(A\) (in sub-block in step 3). This approach requires writing to the result matrix \(n\) times. The more efficient solution would be to use the inner product and calculate the result matrix in one swoop. Figure 5.4(b) shows a pictorial view of a blocking code where the intermediate block calculation requires the multiplication of row of \(A\) (in the left sub block of step 2) and row of \(A\) (in the top sub block of step 2). However, the inner product requires strided access for one operand as shown in Figure 5.3 (inner
block calculation in tiled code). Thus, improving locality requires careful analysis of dependencies, access patterns and orchestration of data movement.

\[
\text{for (jj = 16*I; jj <= min(N-1,16*I+15); jj ++)}\{
    \text{for (kk = 16*K; kk <= min(N-1,16*K+15); kk ++)}\{
        \text{range = min(jj-1,min(N-1,16*I+15))};
        \text{for (ii = 16*I; ii <= range; ii ++)}\{
        \}
    \}
\}
\]

**Figure 5.3:** LU Decomposition inner block calculation showing row and column access of matrix A to calculate the result matrix as shown in Figure 5.4(b).

![LU Decomposition iterative procedure](image)

**Figure 5.4:** Iterative Block LU

In an effort to ameliorate strided access bandwidth and latency deterioration, we introduce our restructuring framework as described in section 3.2.4 that takes into account dependencies, access pattern and multiple processing elements working together. Based on the amount of reuse for groups of threads working in close proximity and sharing the same data space, data is reshaped to adapt its access pattern to underlying hardware requirements. For the rest of this chapter, we will assume that data is stored
in row major mapping without a loss of generality. We further limit our presentation to
two-dimensional array structures to simplify the discussion, however our technique can
also be applied to higher dimensional arrays. We show how our framework restructures
data to improve reuse at different levels of the memory hierarchy.

5.2 Reuse Analysis

The number of independent equations, yielding the access dimensionality in our
case, gives a rank of a matrix for any statement. Similarly, the number of nested loops
within which the statement resides gives the depth of a statement. When \( \text{depth} > \text{rank} \)
\(^1\), it implies that there is \( n^{\text{depth}} - n^{\text{rank}} \) reuse in the iteration space. Revisiting matrix
multiplication in Figure 5.1, \( n^3 \) accesses in \( n^2 \) data implies that the code has \( n^2(n-1) \)
reuse.

Continuing the example, the reuse for matrix \( A \), \( B \) and \( C \) exists along iteration-
space vectors (0,1,0), (1,0,0) and (0,0,1) respectively. In case of tiled parallelization
at the outermost loop \( i \), the reuse vector (1,0,0) is shared by all parallel threads. The
reuse exists not only within threads but also across threads. Higher access strides with
this kind of shared data (stride \( n \) in \( B[k][j] \)) can potentially slow down all participating
treads.

In order to clarify the concept of data sharing between threads, we use the term
self-reuse whenever there exists a reuse for a set of data that is consumed by the same
thread. Similarly, when there exists a reuse for a set of data that is shared by a group of
threads running concurrently, we identify a group-reuse. Our goal is to reduce memory
latency overhead by reducing the access stride for such data so it fits better in shared
caches with minimal conflicts and eviction.

Group-reuse also exists in our LU Decomposition example. In Figure 5.3, during
the final phase of each iterative execution where a tile is computed using matrix mul-
tiplication, accesses are strided for one of the input matrix operands. Since this data

\(^1\) Both rank and depth have to be in the same domain. For example, a tiled iterator
cannot contribute to the depth of a statement.
has reuse throughout the computation of the tiled matrices, latency and bandwidth penalties for strided access have to be paid throughout the execution every time data is evicted and has to be brought into caches.

5.3 Memory Storage Requirement

We use a straightforward methodology to calculate the memory storage requirement for restructuring. We take the dimension of the iteration-space that is shared and identify strided access reuse across a group of threads. The size of the restructuring space is given by the number of tiles multiplied by the size of a tile in shared dimension.

\[
Restructuring\ Space(R) = S \times n
\]  

(5.1)

where \(S\) is the size of the tile and \(n\) is the number of tiles in the shared dimension.

For example, in Figure 5.2, the size of the restructuring space for a input matrix of size \(N \times N\) that is tiled with a tile size \((t * t)\) is given by \(t * t * N/t = Nt\).

5.4 Data Transformation

Based on how the data is accessed, some layouts are better than others. One layout may not provide the best access pattern throughout the execution. In some cases, the same data set can be accessed both by rows and columns within the same iteration space. In other cases, accesses themselves can still be disjoint in time such that at any given time having one mapping is more beneficial. We use information based on the access matrix and its memory mapping and perform data transformations to partial data sets to have access pattern that are more access and cache friendly within a given time frame \(\delta t\). Such transformation is done in a tiled code such that,

- All elements of outer L2 tiles (i.e., the inner L1 tiles in a tiled hierarchy) stay together in memory. The mapping allows memory accesses to take advantage of open memory pages and also results in high residence at various cache levels.
- Elements of the innermost tiles are accessed in a contiguous fashion. This mapping allows better cache locality and reduces unnecessary conflicts.
In order to achieve this, we perform a transformation that allows parallel threads to reuse the allocated restructuring memory space (i.e., increasing the “group reuse” in the application). At different points in time, different disjoint data sets are mapped to the same restructuring space such that there is no conflict. Since our restructuring strategy involves reshaping data to place all elements within a tile together, such transformation requires transformations both at the tile domain and the original domain. One straightforward solution is to find the new indices by traversing the sequence of transformations and compute the modulus for each. This is an expensive proposition. Instead, we compute the displacement (offset used interchangeably) required to map addresses to the restructuring space. This is done in five steps sketched below,

**Algorithm 4 Data Transformation and Movement**

**Input:** Array access matrix $A$ (accessing towards $(0,1)$)
Restructuring space $R$
Lower-bounds for tile indices $LB = lb_1, lb_2...lb_n$

1: **for** each shared data space partition **do**  
2: Subtract the lower bounds in tiled domain (equation 3.6)  
3: **if** tile arrangement and accesses are in the same direction (condition 2) **then**  
4: Transpose in the tiled domain (equation 3.7)  
5: **end if**  
6: Calculate offset in the tiled domain (equation 3.8)  
7: Translate displacement to reflect the original iteration index (equation 3.9).  
8: Transpose in original space (equation 3.10)  
9: **end for**

**Output:** Indices mapped to Restructuring Space

Algorithm 4 currently is tuned to operate under the following two conditions explained below that account for how the tiles are placed in the original space and how the accesses are aligned.

- **Condition 1:** If the shared space (set of outer tiles) is arranged by rows (towards $(1,0)$) and inner elements are accessed by column (towards $(0,1)$), then do the transformation with the calculated displacement (Lines 2, 4 and 5 in algorithm 4) and perform a copy transpose (Line 6) at the element level as shown in Figure 5.5. Initially, $K'$ and $J'$ are calculated by subtracting lower bounds $K$ and
0 respectively. The respective lower bounds become the offset (since they are already relative to the original index $K$ and $J$) and are translated to the values corresponding to the original domain by multiplying with the tile size. It is then followed by a transpose at element granularity to change access towards rows.

\[
\begin{align*}
\text{new index} &= \text{original index} - \text{offset} \\
K' &= K - K_t s_k \\
J' &= J
\end{align*}
\]

**Figure 5.5:** Transformation Condition 1: Strided access (shown by downward arrows) and row arrangement of parallel tiles (A-E) transformed to have contiguous access using calculated offset (shown in red).

For example, Figure 5.6 shows how this transformation can be done with our displacement technique. Consider an original 15x15 space and a restructure space of size 3x15 (shown in a red box) where the index (7,4) is mapped to the index (1,4): We calculate the displacement required in all dimensions of the iteration space. In this example, the lower bound in the ‘J’ and ‘K’ dimension in the tiled domain is 0 and 2. This can be translated to the original space by multiplying with the number of elements in a tile. Hence the displacements become 0 and 2x3 in the original space in the ‘j’ and ‘k’ dimension.

- **Condition 2:** If the outer tiles are arranged by column and the inner elements are accessed by column (towards (0,1)), then do the transformation with the calculated offset (Lines 2,3,4 and 5). This is followed by transpose at element level as shown in Figure 5.7. Here first $K'$ and $J'$ are calculated by subtracting the lower bounds and transposing the tile index. The resultant value of $K'$ and $J'$, i.e., 0 and K are then used to calculate the offset relative to the original index. In this case, relative to the original index $K$, $K'$ becomes $K-K$ and relative to $J$,
Figure 5.6: Transformation example calculating the offset and the new index. First the new index at tile level is calculated relative to the original index K and J, which then is translated to the element index by multiplying them with tile sizes.

\[ J' = J + K - J \]

This gives the offset required to calculate the new indices. It is then followed by a transpose at element granularity to change access towards row.

Figure 5.7: Transformation Condition 2: Strided access (shown by downward arrows) and column arrangement of parallel tiles (A-E) transformed to have contiguous access using calculated offset (shown in red).

In both cases, the amount of displacement required relative to the original index is calculated to perform targeted transformations. This approach allows calculation of the new indices in a single step without the use of modulus.
5.4.1 Exploiting Parallelism

Parallel thread units grab different data set according to their group identification and perform restructuring in a concurrent fashion. Figure 5.8 and 5.9 show the code snippet of such displacement done for matrix multiplication (applying condition 2) and LU decomposition (applying condition 1) respectively for tile size 64x64. In both examples, J is an iteration towards row and K towards column. Parallel units get different iterations of K in matrix multiplication and does restructuring in parallel. Similarly, in LU Decomposition parallelism is exposed along J iteration.

\[
\begin{align*}
\text{offset}_k &= K \times \text{tile size} \\
\text{offset}_j &= K \times \text{tile size} - J \times \text{tile size} \\
\text{for} (kk = 64*K; kk <= \min(N-1, 64*K+63); kk++)
& \quad \text{for} (jj = 64*J; jj <= \min(N-1, 64*J+63); jj++)
& \quad B_{\text{RES}}[jj+\text{offset}_j][kk-\text{offset}_k] = B[kk][jj];
\end{align*}
\]

**Figure 5.8:** Matrix Multiplication restructuring data movement

\[
\begin{align*}
\text{offset} &= K \times \text{tile size} \\
\text{for} (kk = 64*K; kk <= \min(N-1, 64*K+63); kk++)
& \quad \text{for} (jj = 64*J; jj <= \min(N-1, 64*J+63); jj++)
& \quad A_{\text{RES}}[jj][kk-\text{offset}] = a[kk][jj];
\end{align*}
\]

**Figure 5.9:** LU Decomposition restructuring data movement

5.5 Conclusion

In this chapter, we presented conditions and techniques to restructure data and make access contiguous. In order to keep the overhead of restructuring low, we showed that our technique uses series of transformations to calculate the final offsets relative to the original indices to map them to the transformed space. Such transformations when done for datasets with high order reuse, overhead associated with restructuring
can be amortized to benefit the system as a unit. We discuss our experimental results in the next chapter.
Chapter 6

EXPERIMENTS AND RESULTS

In this chapter we present the experimental results for our jagged tiling and restructuring techniques. We present the performance numbers along with the hardware counters showing cache misses at different levels of the memory hierarchy to make our case for the Group Locality framework.

6.1 Jagged Tiling Experimental Setup:

The experiments for jagged tiling were done on Intel Xeon Phi 7110P coprocessor. Each coprocessor is equipped with 61 cores running at 1.1 GHz connected with FDR infiniband interconnect. Each core can support up to 4 hyper-threads, totaling up to 244 threads. Each core has 32KB L1 cache per thread and 512KB L2 cache shared by 4 threads. In addition to the private L2, cores in this system also have access to L2s of all other cores via a ring topology. Only when there is a private L2 cache miss as well as ring L2 caches miss (shared L2s), the request is served by the memory.

6.1.1 Pipeline Start Applications

On the software side, for applications with pipeline start, we divided our experiments into three different sections. In the first section, we selected an example stencil in Figure 4.1 to explore the behavior of parallelism at fine granularity. First we ran the example stencil from 2 to 16 threads per group and varied group numbers from 2 to 30. This experiment was done mainly to analyze the behavior of an application regarding the tradeoff between locality and parallelism. We also ran the example stencil with sweeps from 4 to 128 threads with 4 threads per group with two workloads: 8k by 8k and 16k by 16k elements, as shown in Figure 6.3.
Figure 6.1: Intel Xeon Phi

Figure 6.2: Stencil 2K by 2K: Parameter sweep with different number of thread groups and threads per group
Figure 6.3: Stencil Example Execution times for PLUTO generated code (PLT) and our framework (FG)

Figure 6.4: Seidel 1D Execution times for PLUTO generated code (PLT) and our framework (FG)

Figure 6.5: Seidel 2D Execution times for PLUTO generated code (PLT) and our framework (FG)
In addition, we implemented two versions (1D and 2D) of the Seidel solver loop, a well-known scientific algorithm that computes the solution of a set of linear equations. We selected these examples to showcase our framework against PLUTO generated code using OpenMP as their parallel target. We selected arrays of 4 million elements and 8 millions elements running 4k and 16k times respectively for the one dimensional Seidel. The execution time for these runs are presented in Figure 6.4. For our 2D Seidel, our selected workloads are a 4k by 4k array of elements ran over 2K times and a 10k by 10K element array ran over 6k times. The results of these runs are shown in Figure 6.5

![Seidel 2D Remote Cache misses](image1)

(a) Seidel 2D Remote Cache misses for Reads

![Seidel 2D Remote Cache misses](image2)

(b) Seidel 2D Remote Cache misses for Writes

**Figure 6.6:** Seidel 2D Remote Cache misses for PLUTO generated code (PLT) and our framework (FG)

![Seidel 2D Memory-served misses](image3)

(a) Seidel 2D Memory-served misses for Reads

![Seidel 2D Memory-served misses](image4)

(b) Seidel 2D Memory-served misses for Writes

**Figure 6.7:** Seidel 2D Memory-served misses for PLUTO generated code (PLT) and our framework (FG)
Figure 6.8: Seidel 2D write serviced by memory for PLUTO generated code (PLT) our framework plus overhead and our framework without overhead. Note that the Y-axis is in logarithmic scale.

In the final section, we chose one Seidel example, the biggest 2D case, to characterize the memory and remote cache misses. These results are shown in Figure 6.6 for caches and in Figure 6.7 for memory.

All our experiments were designed to utilize all 4 hyperthreads provided by our target architecture. In order to do so, we did compact pinning such that hyperthreads form a thread group in our fine-grain execution. Similarly, we set KMP_AFFINITY to compact for OpenMP code. Our experiments show better runtime for fine grain execution compared to PLUTO generated parallel code using OpenMP as a baseline – denoted as PLT in figures. In addition, our results show that when threads collaborate, we gain in performance. All our codes were compiled using Intel’s icc version 13.1.1 and use the Linux Perf tool [3] to collect the memory and cache related performance counter information. The performance counters collected from our experiments are presented in Table 6.3.

6.1.2 Discussion

One of the major challenges in parallel programming is finding a balance between locality and parallelism. Allowing threads to run off with available parallel task exposes parallelism to maximum but does not always exploit locality. Figure 6.2 shows the progression of execution time (z-axis) as number of thread groups (x-axis) and number of threads per group (y-axis) vary for a simple stencil application. We can see that as
we increase intratile parallelism (y-axis), performance initially increases and slowly goes down. Similarly, as number of groups increases, we see similar behavior in performance numbers. This is because; initially intratile parallelism allows more locality along with parallelism. Since, the architecture provides 4 hyperthreads, the trend continues until the overhead of synchronization start to impact performance resulting in negative return. Also, with too many groups, performance decline because of starvation as some threads are forced to stay idle. The concave progression of execution time in 3d curve shows that careful consideration of both architecture features and application characteristics are needed to get higher performance.

The benefit of execution with locality consideration comes from the amount of reuse an application offers. The stencil in example in Figure 4.1 does not have much reuse. Thus, when smaller number of threads are used, the generated PLUTO code (represented as PLT in the charts) can outperform fine-grain execution with jagged tiling (represented as FG in the charts) since only limited amount of data is reused in few cores L2 caches and it is not enough to overcome the slow start introduced by coarser outer L2 tiles. However, as number of threads is incremented, FG shows better execution time. This trend is visible in Figure 6.3.
Figures 6.4 and 6.5, on the other hand, have much better reuse since the Seidel loops (for both 1D and 2D) have time dimensions in which the entire arrays are reused. When L2 locality is not considered as is the case with the PLUTO generated code, eviction rates increases and this is reflected in the performance of this approach. However with FG, threads working as a group execute different tiles within L2 in a synchronous fashion exploiting both parallelism and reuse offered within outer tiles. In all these approaches, the performance gains are clearly visible up to a number of threads and afterward the performance stays constant. This plateau is reached when the available parallelism is exhausted and the rest of the threads do not have any useful work. In Figures 6.4(a) and 6.5(a), the plateau is reached at 64 and 32 threads respectively. When increasing the workload sizes (as in Figures 6.4(b) and 6.5(b)), the plateau is pushed further (to 240 threads for the first case and 128 for the second).

In Intel Xeon Phi, when a thread misses a read or a write access to local L2 caches, it first checks if data is available in neighboring L2 caches and goes to the memory only if there is both local and remote L2 misses. These events include both demand fills as well as prefetches and are hence a close approximation for demands hits and misses in local L2 caches. In the Intel Xeon Phi, the remote cache accesses might be as expensive as an access to memory; thus, having a large number of remote cache access might greatly affect performance [27]. Figures 6.6(a) and 6.6(b) show amount of reads and write shared among caches within the L2 ring for the largest Seidel loop example (c.f. Figure 6.5(b)). Similarly, Figures 6.7(a) and 6.7(b) show amount of data brought from memory for the same test case.

The information extracted from Figures 6.6(a) and 6.7(a) show that the PLUTO generated code has a higher number of remote cache accesses and memory serviced reads compared with the Fine-grained approach. This tells us that for the same work-load, fine-grained data is reused more often than its PLUTO counterpart (i.e. low memory misses and low remote cache misses for the same data set).

In the case of writes, Figures 6.6(b), 6.8 and 6.7(b), show interesting results. The remote cache misses for the writes shows an inflection point around 64 threads.
After 64 threads, the FG approach shows improvements over the PLUTO generated code. However, before the inflection point, our approach incurs in more share writes. The reason is that when using more cores more core caches can participate into the computation.

Figure 6.8 show the original overhead for memory writes for our framework. The Y-axis in this figure is in log-scale to better show the differences between the approaches. The original framework suffers from a large number of writes to memory. This effect is due to a synchronization variable inside the framework. However, the number of writes in this application is an order magnitude lower than reads. Thus, their effect in performance is small. However, to obtain the application’s memory-serviced writes, we modified the framework to bypass the synchronization variable. This data is presented in Figure 6.7(b). In this case, we have an increase on writes, but as expressed before, the sheer number of more reads than writes, means that reads have a larger influence in the performance of the application for this application. However, we are investigating how to reduce this write pressure on the main memory.

These charts show that for application with plenty of reuse, threads can take advantage of accesses by threads within group and hence collaborate to maximize memory residence in nearby memory with minimal interference.

The balance between granularity of tasks and the amount of parallelism is very crucial to maximize performance. Coarse grain execution can lead to underutilization of resources whereas fine grain can lead to more conflicts and contentions at different levels of the memory hierarchy. Most of current architectures offer vast amount of computing resources, some with hyperthreads sharing caches and some with non-uniform memory access where accessing some address ranges are cheaper than others. In such cases, taking advantage of shared resources such as caches or address range in close proximity by threads working together as a unit can have significant effect on performance.
6.1.3 Parallel Start Applications

We picked five stencil kernels that exhibit concurrent start. In addition to our running example (Heat-1d) we used in Chapter 4, we also use Heat-2d, Heat-3d, Jacobi-2d, 7 point-3d stencils to show the efficiency of our techniques experimentally. All these kernels have a face along the iteration space where they can be executed concurrently. We selected a range of tile sizes based on full tile sizes, number of available cores, caches sizes, memory capacity, etc. From this range, we selected the best results as our experimental sizes. Table 6.2 shows the kernels and the sizes we used for our experiments.

<table>
<thead>
<tr>
<th>Application</th>
<th>Size(N)</th>
<th>Size(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat-1d</td>
<td>10000000</td>
<td>10K</td>
</tr>
<tr>
<td>Heat-2d</td>
<td>11504x11504</td>
<td>2K</td>
</tr>
<tr>
<td>Heat-3d</td>
<td>480x480x480</td>
<td>100</td>
</tr>
<tr>
<td>Jacobi-2d</td>
<td>11504x11504</td>
<td>2K</td>
</tr>
<tr>
<td>7point-3d</td>
<td>480x480x480</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6.2: Applications and their sizes

We compare our fine-grain (FG) jagged tiling approach with the diamond tiling generated by PLUTO (PLT). To test the effect of thread scheduling on the given techniques, we use different thread mappings supported by the architecture. For PLUTO generated OpenMP code we used BALANCED and COMPACT mode. In BALANCED mode, threads are distributed equally among cores to reduce load imbalance. This approach has the advantage of using physical cores first in which resources are not contended, in contrast to using hyperthreads. In COMPACT mode, all hyperthreads within a core are assigned before moving to the next one. For our fine-grain framework, we use mappings equivalent to SCATTER and COMPACT (with pthread_setaffinity_np) mode. In our framework’s scatter mode, the scheduler assigns threads to different cores and hence threads from different cores form a group. On the other hand in compact mode, hyperthreads within the core form a group. Such
approach has the advantage of data reuse among threads when they work in close proximity in time and space.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>PLT GFLOPS</th>
<th>Tile Size</th>
<th>Hierarchical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat-1d</td>
<td>106.59</td>
<td>107.05</td>
<td>4Kx4K</td>
</tr>
<tr>
<td>Heat-2d</td>
<td>122.42</td>
<td>122.78</td>
<td>16x16x256</td>
</tr>
<tr>
<td>Heat-3d</td>
<td>59.375</td>
<td>57.22</td>
<td>3x3x2x480</td>
</tr>
<tr>
<td>Jacobi-2d</td>
<td>68.26</td>
<td>68.40</td>
<td>16x16x256</td>
</tr>
<tr>
<td>7point-3d</td>
<td>31.48</td>
<td>31.82</td>
<td>2x2x4x480</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel</th>
<th>FG GFLOPS</th>
<th>Tile Size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat-1d</td>
<td>115.95</td>
<td>111.77 / 1Kx2K / 4x4</td>
<td>8.31%</td>
</tr>
<tr>
<td>Heat-2d</td>
<td>131.47</td>
<td>134.95 / 16x32x256 / 4x4x2</td>
<td>9.91%</td>
</tr>
<tr>
<td>Heat-3d</td>
<td>61.73</td>
<td>74.168 / 1x2x4x480/4x4x2x1</td>
<td>24.91%</td>
</tr>
<tr>
<td>Jacobi-2d</td>
<td>67.94</td>
<td>72.22 / 16x32x256 / 4x4x2</td>
<td>5.58%</td>
</tr>
<tr>
<td>7point-3d</td>
<td>33.46</td>
<td>41.74 / 1x2x4x480/4x4x2x1</td>
<td>31.17%</td>
</tr>
</tbody>
</table>

Table 6.3: Performance for PLT and FG execution with 240 threads. Note: Heat 3d peaks at 180 threads.

Table 6.3 shows the performance for each of the selected kernels in GFLOPS for both PLT and FG along with the selected tile sizes. For FG, a second set of tile sizes represent number of Level 1 (a.k.a L1) tiles in different dimensions. For example, tile size 16x32x256 / 4x4x2 represents a tile with Level 1 tile size 16x32x256 and Level 2 (a.k.a L2) tile size 16*4x32*4x256*2. As shown in this table, the best performance overall occurs when using the jagged tiling framework in compact mode, except for the case of heat 1d. In addition, the best performing cases were seen when 240 threads were used, except for heat-3d (PLT Balanced mode). The improvement over the PLUTO generated code ranges from 5.58% (in the case of Jacobi-2d) to 31.17% (in the case of the 7 point stencil kernel) for the highest performing codes. The last column of Table 6.3 shows the performance for two level tiling for the heat based kernels. The reduced performance is because of the additional control overhead and idle resources.
created due to lack of inner parallelism. However, further investigation shows improved performance for Heat-2D when using smaller tiles for both PLUTO and our framework, 127.24 GFLOPS and 140.01 GFLOPS respectively.

The three heat kernels are used as a case study to show how our framework performs with respect to the architecture’s memory hierarchy. The kernel’s access patterns illustrate the advantage and disadvantages of our approaches. The memory behavior for the three kernels are reported in Figures 6.9, 6.10 and 6.11. These figures showcase the most expensive operations (i.e. misses and remote hits) in each level of the memory hierarchy. An interesting feature of the study on the heat 3d kernel is that the best cases from our initial runs were selected for two tile sizes, named PLT_2 (2x2x4x480) and PLT_3 (3x3x2x480) in the graphs, to compare and contrast the effect of the size of the diamond tile for PLUTO generated code. The PLT_2 cases have better load balance whereas the PLT_3 cases take advantage of locality better, however cannot utilize all available threads. As the stride increases (in higher dimensionality stencils), the jagged tiling framework effectively reduces the accesses to main memory. The presented case studies use the compact schedule as their default. Section 6.1.4 discusses the graphs in more detail.

6.1.4 The Heat Kernels Case Study

The heat 1d kernel presents the lowest improvement of all the kernels. This behavior can be connected to an increase in L1 cache misses (reflected in the collected counters shown below) presented in our framework. The worst degradation of our framework versus the PLUTO generated code is 10% when using 60 threads and compact scheduling. The highest improvement occurs when we use the balanced and scatter schedules which is around 9%. The difference between the balance and compact schedules can be attributed to the access pattern exhibited by the 1D case. The Intel Phi hardware can take advantage of the contiguous access pattern of this kernel. Our framework behavior might interfere with the prefetcher hardware, especially to the
Figure 6.9: Heat 1D Memory behavior for PLUTO generated code (PLT) and our framework (FG)

highest memory hierarchy levels. A short discussion of the memory hierarchy counters is presented below.

Figure 6.9(d) shows the number of cache misses that were served by the memory subsystem. This chart shows that there is a considerable reduction of memory served cache misses (around 93% on average of total accesses are reduced) over all the presented cases. In case of using the entire machine, we have a 96% reduction of memory accesses.

Since the remote cache hits cycle can be as high as access to the memory, a higher number of remote cache hits may result in performance degradation. Figure 6.9(c) shows also a great reduction of these types of hits when comparing our framework and the PLUTO generated code. This reduction averages around 50% in favor of our
Figure 6.10: Heat 2D Memory behavior for PLUTO generated code (PLT) and our framework (FG)

framework and a 66% when using the full machine.

However, when examining Figure 6.9(b), we see that the Level 1 misses are disproportional high (around 2 times the PLUTO number). This rapidly decreases the performance advantages FG has over PLT from the other accesses. However, the reduction on the other level still provides the performance improvement shown in Figure 6.9(a).

The reason for this behavior is because 1D stencils do not have a locality problem to begin with. The accesses are fairly contiguous and tiles are big. Due to this, the hardware prefetcher can take advantage by bringing data to the level 1 cache. Our framework’s thread group might interfere with this behavior resulting in the higher miss rates for level 1. Figure 6.9(a) shows the evolution of performance for different
Figure 6.11: Heat 3D Memory behavior for PLUTO generated code (PLT) and our framework (FG)

number of threads. FG does better than PLT only when 240 threads are used. Also, it is interesting to note that for heat-1d, balanced mode does better than compact mode. This is because when locality is not much of an issue, having threads from different cores have the advantage, as resources are not contended. This behavior however changes for higher order stencils.

The Level 1 miss rates for the other heat kernels (Figures 6.10(b) and 6.11(b)) show a slightly different story with some improvements (around 7% for heat 3d) on the Level 1 misses. As the dimensionality of stencil increases, locality optimization becomes more important. Because of the strided access, PLT is not able to take full advantage of L1 locality as the effectiveness of hardware prefetcher reduces. However with FG, we start to see advantage of grouping threads and accessing Level 2 tiles to
improve locality.

In the case of heat 2d, the running times for both our framework and the PLUTO codes are very similar when not using the entire machine. However when using the entire machine, the gains are more visible (around 10% in compact scheduling) as shown in Figure 6.10(a). These gains are represented in the total reduction of memory accesses showcased by the counters presented below.

Figure 6.10(d) shows the cache misses that were served by the main memory. The chart shows a general trend in reduction of cache misses for all the thread sizes. On average, the reduction is around 33% over the PLUTO generated code with a maximum of 40% when using 60 threads. Although when the number of threads is increased, the gap reduces to up to 27% reduction. This showcases a small interference between the running threads that might be attributed to our framework. Nevertheless, this general trend shows us that more data is kept inside the internal caches of the machine (as was the case with heat 1d).

In Figure 6.10(c), we showcase the remote level 2 cache hits. These numbers show an increase (instead of a reduction) of the number of remote hits. As stated before, this might be a performance issue due to their latencies. However, since the other two sets of misses (L1 misses, L2 misses served by memory) are orders of magnitude higher than these hits, their effect are ameliorated. The increase, in average, is 50% and as high as twice when using the entire machine.

This set of charts shows us that the evolution of the application over the different thread sizes is similar for both level 1 misses and memory served misses. However, the Level 2 remote hits slightly increase with size. This is due to write miss increase resulting from our framework overhead. We are working to reduce this overhead. Figure 6.10(a) shows the performance evolution for different number of threads. FG, in this case, can take advantage of locality and reuse among thread group and hence shows better performance for all cases.

Figure 6.11(a) shows the gains in GFLOPS of heat 3d over one of the PLUTO generated cases. Over all cases, our framework has improvements over the PLUTO
generated cases with around 30% using the full 240 threads and it peaks at 70% improvement when using 180 threads. The memory characterization is discussed below and shows a great reduction on the memory accesses, remote hits and a slight reduction of level 1 cache misses.

For heat 3d, we have two special cases. The reason is, for PLT, using smaller tile size (2x2x4x480) creates perfect load balance, however, cannot take advantage of locality much. On the other hand tile sizes (3x3x2x480) takes advantage of locality better but is not able to use all processing resources. In the first case, we have a smaller diamond tile (shown as PLT_2). When increasing the number of threads, Figure 6.11(c) shows that the number of remote cache hits reduces as more threads are added to the mix. Our framework starts (running on 60 threads) with an increase of 60% over the PLUTO generated code. However, as more threads are added, the remote hits reduce to up to 90% over the PLUTO generated code. The reduction of memory served cache misses is not as impressive but still considerable (around 34% on average).

For the second tile size (named PLT_3) showcases a larger diamond tile. The story about the memory hierarchy behavior is very similar to the smaller diamond tile. However, the gaps are narrowed with up to 62% reduction for remote level 2 hits and 12% reduction in memory served cache misses. Here the bigger diamond tile takes advantage of the available locality. However, as mentioned earlier, doing so creates load imbalance resulting on idle resources. Because of this, the performance for PLT_3 peaks at 180 (in balanced mode, so all cores can be used) as shown in figure 6.11(a). Our framework would take advantage of the locality with greater tile sizes as well, but our test platform does not have enough memory to run this experiment.

The charts support the overall premise that there is a reduction of memory accesses over PLUTO generated diamond tiling. Thus, this supports the idea that our framework reuses data better by a hierarchical tiling strategy that manages data movement collectively across threads. This notion is reflected by the performance and supported by the counters. The behavior of heat 2d and heat 3d are representative of the memory behavior of the Jacobi kernel and the 7 Point Stencil, respectively.
They present similar behaviors in all the characterizations. For example, for the 7 Point Stencil, there are similar reductions across the board for all the memory related counters (4% reduction in Level 1 caches, 90% reduction in remote hits and 18% in memory accesses when using 240 threads) and an improvement of around 30% in GFLOPS (as shown in Table 6.3).

6.1.5 DOE Proxy Mini-Apps

The current DOE applications consist of large computational pipelines with large amount of glue code, transport and computational kernels. The idea of analyzing critical compute kernels gave rise to the idea of proxy and mini applications. They represent “hot” computational kernels from large computational pipelines dedicated on solving science problems, like material deformation under stress, explosion propagation, nuclear fusion, combustion, quantum chemistry, atmospheric simulation and forecasting. Thus, these proxy / mini applications become the target of many of the co-design efforts led by DOE, among them XSTACK [4]. Here we show and discuss results for Tealeaf-Jacobi-2D and miniAMR-3D for sizes shown in table 6.4. For these two examples, we did a tile size sweep for both PLUTO generated code and fine-grained jagged tile version. We used 240 threads and ran experiments under COMPACT affinity mode.

<table>
<thead>
<tr>
<th>Application</th>
<th>Size(N)</th>
<th>Size(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tealeaf Jacobi-2d</td>
<td>6Kx6K</td>
<td>1K</td>
</tr>
<tr>
<td>MiniAMR-3d</td>
<td>480x480x480</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6.4: DOE Proxy Applications and Sizes

6.1.5.1 Tealeaf Jacobi-2D

Tealeaf solves a heat conduction equation and uses a 5-point stencil solver. Since the implementation provided with Mantevo suite uses jacobi method, it has a face in iteration space where all dependencies are satisfied and hence it allows concurrent start.
Figure 6.12 shows a snippet of a tealeaf code. We use our jagged tiling technique along with our fine grain execution scheme to exploit both parallelism and locality.

```
DO n=1,max_iters
  !$OMP PARALLEL
  !$OMP DO
    DO k=y_min-1,y_max+1
    DO j=x_min-1,x_max+1
      un(j,k) = u1(j,k)
    ENDDO
    ENDDO
  !$OMP END DO
  !$OMP END PARALLEL

  !$OMP DO REDUCTION(+:error)
  DO k=y_min,y_max
    DO j=x_min,x_max
      u1(j,k) = (u0(j,k) + rx*(Kx(j+1,k)*un(j+1,k) + Kx(j ,k)*un(j-1,k)) &
                 + ry*(Ky(j ,k+1)*un(j   ,k+1) + Ky(j   ,k)*un(j   ,k-1))) &
                 / (1.0_8 &
                   + rx*(Kx(j,k)+Kx(j+1,k)) &
                   + ry*(Ky(j,k)+Ky(j+1,k))))
    error = error + ABS(u1(j,k)-un(j,k))
  ENDDO
  ENDDO
  !$OMP END DO
  !$OMP END PARALLEL
```

Figure 6.12: Tealeaf[2]

Figures 6.13 and 6.14 show the performance, L1 cache misses, L2 remote hits and the cache misses served by memory for PLUTO OMP (PLT) version and fine-grain (FG) version of a TeaLeaf code respectively. In the figure the x-axis shows the tile sizes in different dimensions for all levels of the memory hierarchy.

The best performance is shown when tile size is 4 x 4 x 256 for the PLUTO generated OMP code. The main reason is due to the very large third dimension that provides the prefetcher sufficient opportunities to intervene constructively. However increasing the other two dimensions slows the performance down as this topology limits parallelism forcing some of the 240 threads to idle.

The performance increases significantly with jagged tiled code. However, the trend stays the same as smaller tile sizes perform better than larger tile sizes. Read and write L1 misses increase with tile sizes as the resulting tile do not fit in the limited 32KB cache space. The number of remote L2 accesses – especially the reads – is lower with jagged tiled code, which plays to our advantage resulting in higher performance.
Figure 6.13: TeaLeaf Memory Behavior for PLUTO generated code (PLT)

Accessing remote L2 in Intel Xeon Phi can be as expensive as accessing a memory. When reuse is maximized within local caches, such accesses can be reduced.

The number of memory accesses is also lower with jagged tiled fine grain code. This is again due the improvement in local reuse. As tile size is increased, the cache misses increases which result in more memory accesses that lead to reduced performance as shown in the performance charts.

6.1.5.2 MiniAMR

This miniAMR application does the 3D calculation with Adaptive Mesh Refinement. Such technique allows adapting the precision based on the sensitivity of the region in cases where uniform precision is not required. It has applications in the field of hydrodynamic and astrophysics. The code provided in Mantevo suite uses a 3D
Figure 6.14: TeaLeaf Memory Behavior for Fine-grain (FG)
solver that iterates over a number of time steps providing a concurrent start. Figure 6.15 shows a snippet of a miniAMR kernel.

```c
for (ts = 1; ts <= num_tsteps; ts++) {
    for (i = 1; i <= x_block_size; i++)
        for (j = 1; j <= y_block_size; j++)
            for (k = 1; k <= z_block_size; k++) {
                sb = bp->array[i-1][j-1][k-1] + bp->array[i-1][j-1][k] + bp->array[i-1][j-1][k+1] +
                    bp->array[i-1][j][k-1] + bp->array[i-1][j][k] + bp->array[i-1][j][k+1] +
                    bp->array[i-1][j+1][k-1] + bp->array[i-1][j+1][k] + bp->array[i-1][j+1][k+1];
                sm = bp->array[i ][j-1][k-1] + bp->array[i ][j-1][k] + bp->array[i ][j-1][k+1] +
                    bp->array[i ][j ][k-1] + bp->array[i ][j ][k ] + bp->array[i ][j ][k+1] +
                    bp->array[i ][j+1][k-1] + bp->array[i ][j+1][k ] + bp->array[i ][j+1][k+1];
                sf = bp->array[i+1][j-1][k-1] + bp->array[i+1][j-1][k] + bp->array[i+1][j-1][k+1] +
                    bp->array[i+1][j ][k-1] + bp->array[i+1][j ][k ] + bp->array[i+1][j ][k+1] +
                    bp->array[i+1][j+1][k-1] + bp->array[i+1][j+1][k ] + bp->array[i+1][j+1][k+1];
                work[i][j][k] = (sb + sm + sf)/27.0;
            }
    for (i = 1; i <= x_block_size; i++)
        for (j = 1; j <= y_block_size; j++)
            for (k = 1; k <= z_block_size; k++)
                bp->array[i][j][k] = work[i][j][k];
}
```

**Figure 6.15:** miniAMR[2]

Figures 6.16 and 6.17 show the performance, L1 cache misses, L2 remote hits and the cache misses served by memory for PLUTO OMP (PLT) version and fine-grain (FG) version of a MiniAMR code. In the figure the x-axis shows the tile sizes in different dimensions for all levels of the memory hierarchy.

The miniAMR is a very compute intensive kernel. This means that the ratio between computation and communication is skewed towards computation. Although the effects of memory are still impactful, other core centric effects become visible as well such as contention in the pipeline, ALU or the amount of stalls. Despite the limitation of hitting the processors computational ceiling, our technique still performs comparable or slightly better than PLUTO generated OpenMP code.

The amount of read misses for L1 reduces for jagged tiling whereas the write increases. However in total, the number of read/write misses on L1 is very similar for the both best performing cases (i.e. for PLUTO 2x2x4x480 and for FG is 1x2x8x480).
Figure 6.16: miniAMR Memory Behavior for PLUTO generated code (PLT)
The remote L2 accesses, which can be as expensive as going to the memory, are less for jagged tiling as compared to PLUTO/OpenMP-generated code for both best cases, which plays to our advantage. The figure shows that for bigger tile sizes such accesses are very small for PLUTO/OpenMP generated code. This is because in such cases, data is fetched from the memory directly instead of any L2 cache since the tiles will not fit in the shared level 2 caches. Thus, this will result in a higher number of memory accesses.

The charts show a very interesting scenario. For best performing cases, access to memory is higher for jagged tiling as compared to PLUTO/OpenMP generated code. Although overall memory accesses are lower for the fine grain runtime. However, this increase in memory accesses does not correlate with the performance (for which both best cases are very similar). This can be explained by taking into consideration all the other compute related aspects that are taken place (i.e. vectorization, pipeline stalls, FP performance, etc).

6.2 Data Restructuring Experimental Setup:

Figure 6.18 shows the pictorial view of our hardware testbed: the Tile-Gx36 architecture. It has 36 processor cores, each equipped with 32KB local 2-way L1 cache and 256KB 8-way L2 cache. All caches are inclusive. Each core also has access to the other core’s L2 cache in the grid, giving an impression of a virtual L3 cache. Accesses to L3 caches are much cheaper than accessing the memory (anywhere from 2x to 3x faster). Our framework uses intratile parallelism to exploit reuse within the grid and performs restructuring for better access strides.

On the software side, we use our running examples from Chapter 5, matrix multiplication and LU decomposition, to display the effectiveness of our technique. We also present performance hardware counters to show how our techniques help reduce cache misses. Table 6.5 presents the counters we used from TileGX. These counters cover local and remote read and write misses for different cache levels.
Figure 6.17: miniAMR Memory Behavior for PLUTO generated code (PLT)
**Figure 6.18:** Tile-Gx36 Block Diagram

<table>
<thead>
<tr>
<th>Performance Counter</th>
<th>Description</th>
<th>Short-hand</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_MISS</td>
<td>Level 1 Cache misses for reads</td>
<td>L1RM</td>
</tr>
<tr>
<td>WRITE_MISS</td>
<td>Level 1 Cache misses for writes</td>
<td>L1WM</td>
</tr>
<tr>
<td>LOCAL_DATA_READ_MISS</td>
<td>Local Level 2/3 Cache misses for reads</td>
<td>L2RM</td>
</tr>
<tr>
<td>LOCAL_WRITE_MISS</td>
<td>Local Level 2/3 Cache misses for writes</td>
<td>L2WM</td>
</tr>
<tr>
<td>REMOTE_DATA_READ_MISS</td>
<td>Remote Level 2/3 Cache misses for reads</td>
<td>RRM</td>
</tr>
<tr>
<td>REMOTE_WRITE_MISS</td>
<td>Remote Level 2/3 Cache misses for writes</td>
<td>RWM</td>
</tr>
</tbody>
</table>

**Table 6.5:** Performance Counters collected in TileGx36
6.2.1 Discussion

Figure 6.19 shows performance for matrix multiplication over different problem sizes. Our result shows an improvement of up to 26.50% over best case OMP code (OMP_INTRA as in OMP intratile). Additionally, it shows up to 4.5% improvement when compared against our fine-grain grouping techniques briefly discussed in the background section that do not apply restructuring. Figures 6.20(a), 6.20(b) and 6.20(c) show hardware counters for different level cache misses. Our results show reduction in cache misses when compared to the best fine-grain and OMP cases. Compared to the OMP intratile parallel version, fine-grain with restructuring shows reduction in cache misses for all read and write misses at cache levels L1, L2 and L3 ranging from 23.53% to 89.81% except for the remote write misses (aka L3 write misses) where it deteriorates by 138%. Compared to our original work of fine-grain without restructuring, our restructuring policies lead to reduction of local L1 and L2 cache misses by up to 19.1%. However, the remote read and write misses (L3 misses) increase by 12.9% and 15.4% respectively. We believe that increases in remote misses are due to the synchronization overhead required by our framework grouping the physical cores.

Figure 6.22 shows performance for our second example, LU decomposition, at different problem sizes for fine-grain with/without restructuring and OMP code. For most cases, we use hierarchical and one level tiled OMP code instead of intra-tile OMP parallel code as reference as its performance better. Our technique with restructuring has up to 31.4% advantage over OMP one level tiling and 15.9% advantage over fine-grain without restructuring.

Similarly, Figures 6.23(a), 6.23(b) and 6.23(c) show hardware counters for cache misses. Compare to OMP one level tiling, our technique has an advantage ranging from 17.8% to 72.69% for reduction of all cache misses except for remote L2 writes misses where it increases by 267.23%.

A special case is apparent when looking at the 6912x6912 hierarchical OpenMP case. In this case, our framework performs better than any of the OpenMP cases but it has a perceived disadvantage with almost all the memory hierarchy counters. Our
<table>
<thead>
<tr>
<th>Size</th>
<th>GFlops</th>
<th>RRM</th>
<th>RWM</th>
<th>L1RM</th>
<th>L1WM</th>
<th>L2RM</th>
<th>L2WM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FineGrain+Restructuring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2304</td>
<td>4.9119</td>
<td>9.20E+07</td>
<td>1.18E+08</td>
<td>1.67E+10</td>
<td>1.34E+08</td>
<td>3.62E+06</td>
<td>5.29E+05</td>
</tr>
<tr>
<td>4608</td>
<td>4.9051</td>
<td>7.17E+08</td>
<td>2.84E+08</td>
<td>1.39E+11</td>
<td>3.37E+08</td>
<td>2.91E+07</td>
<td>3.87E+06</td>
</tr>
<tr>
<td>6912</td>
<td>4.9029</td>
<td>2.37E+09</td>
<td>6.33E+08</td>
<td>4.68E+11</td>
<td>8.12E+08</td>
<td>1.01E+08</td>
<td>1.25E+07</td>
</tr>
<tr>
<td>FineGrain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2304</td>
<td>4.873</td>
<td>7.86E+07</td>
<td>1.09E+08</td>
<td>1.95E+10</td>
<td>1.30E+08</td>
<td>3.88E+06</td>
<td>4.68E+05</td>
</tr>
<tr>
<td>4608</td>
<td>4.7514</td>
<td>6.51E+08</td>
<td>2.45E+08</td>
<td>1.63E+11</td>
<td>4.26E+08</td>
<td>3.80E+07</td>
<td>3.91E+06</td>
</tr>
<tr>
<td>6912</td>
<td>4.6898</td>
<td>2.12E+09</td>
<td>5.15E+08</td>
<td>5.52E+11</td>
<td>1.33E+09</td>
<td>1.48E+08</td>
<td>1.35E+07</td>
</tr>
<tr>
<td>OMP_Hierarchical</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2304</td>
<td>2.6142</td>
<td>1.42E+08</td>
<td>1.27E+10</td>
<td>1.85E+10</td>
<td>1.38E+10</td>
<td>4.11E+06</td>
<td>4.05E+05</td>
</tr>
<tr>
<td>4608</td>
<td>2.1063</td>
<td>2.67E+09</td>
<td>1.01E+11</td>
<td>2.32E+11</td>
<td>1.10E+11</td>
<td>7.42E+07</td>
<td>3.40E+06</td>
</tr>
<tr>
<td>6912</td>
<td>2.5894</td>
<td>4.76E+09</td>
<td>3.41E+11</td>
<td>5.06E+11</td>
<td>3.73E+11</td>
<td>1.71E+08</td>
<td>1.42E+07</td>
</tr>
<tr>
<td>OMP_with_IntratileParallelism</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2304</td>
<td>4.0357</td>
<td>1.34E+08</td>
<td>3.33E+07</td>
<td>2.17E+10</td>
<td>6.46E+08</td>
<td>2.22E+07</td>
<td>9.56E+05</td>
</tr>
<tr>
<td>4608</td>
<td>3.8775</td>
<td>1.94E+09</td>
<td>1.50E+08</td>
<td>2.19E+11</td>
<td>6.37E+09</td>
<td>1.86E+08</td>
<td>7.62E+06</td>
</tr>
<tr>
<td>6912</td>
<td>4.0186</td>
<td>3.70E+09</td>
<td>3.69E+08</td>
<td>5.25E+11</td>
<td>1.77E+10</td>
<td>6.08E+08</td>
<td>2.53E+07</td>
</tr>
</tbody>
</table>

**Table 6.6:** Performance Counters collected in TileGx36 for Matrix Multiplication. Bold green and blue values represent the best values for Fine grain and OpenMP experiments.

**Figure 6.19:** Matrix Multiplication Performance in GFlops
<table>
<thead>
<tr>
<th>Size</th>
<th>GFlops</th>
<th>RRM</th>
<th>RWM</th>
<th>L1RM</th>
<th>L1WM</th>
<th>L2RM</th>
<th>L2WM</th>
</tr>
</thead>
<tbody>
<tr>
<td>FineGrain+Restructuring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2304</td>
<td><strong>4.1616</strong></td>
<td>1.13E+08</td>
<td>5.83E+07</td>
<td>6.02E+09</td>
<td>8.29E+07</td>
<td>1.93E+06</td>
<td>1.99E+05</td>
</tr>
<tr>
<td>4608</td>
<td><strong>4.4568</strong></td>
<td>9.52E+08</td>
<td>3.55E+08</td>
<td>4.88E+10</td>
<td>5.01E+08</td>
<td>1.56E+07</td>
<td>1.18E+06</td>
</tr>
<tr>
<td>6912</td>
<td><strong>4.5659</strong></td>
<td>2.70E+09</td>
<td>1.06E+09</td>
<td>1.67E+11</td>
<td>1.65E+09</td>
<td>5.07E+07</td>
<td>3.40E+06</td>
</tr>
<tr>
<td>FineGrain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2304</td>
<td>4.1562</td>
<td>1.33E+08</td>
<td>5.29E+07</td>
<td>5.72E+09</td>
<td>2.76E+08</td>
<td>2.08E+06</td>
<td>2.22E+05</td>
</tr>
<tr>
<td>4608</td>
<td>4.297</td>
<td>1.27E+09</td>
<td>3.33E+08</td>
<td>7.51E+10</td>
<td>2.46E+09</td>
<td>2.39E+07</td>
<td>2.65E+06</td>
</tr>
<tr>
<td>6912</td>
<td>3.9364</td>
<td>3.22E+09</td>
<td>1.03E+09</td>
<td>1.77E+11</td>
<td>7.70E+09</td>
<td>5.82E+07</td>
<td>4.18E+06</td>
</tr>
<tr>
<td>OMP_Hierarchical</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2304</td>
<td>2.8563</td>
<td>5.06E+07</td>
<td>1.52E+07</td>
<td>6.20E+09</td>
<td>2.37E+08</td>
<td>1.69E+06</td>
<td>1.64E+05</td>
</tr>
<tr>
<td>4608</td>
<td>3.1133</td>
<td>5.77E+08</td>
<td>6.59E+07</td>
<td>8.09E+10</td>
<td>2.11E+09</td>
<td>1.77E+07</td>
<td>8.95E+05</td>
</tr>
<tr>
<td>6912</td>
<td>3.6915</td>
<td>1.36E+09</td>
<td>1.41E+08</td>
<td>1.65E+11</td>
<td>6.40E+09</td>
<td>4.54E+07</td>
<td>3.62E+06</td>
</tr>
<tr>
<td>OMP_One_Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2304</td>
<td><strong>3.9019</strong></td>
<td>6.67E+07</td>
<td>1.67E+07</td>
<td>7.73E+09</td>
<td>2.29E+08</td>
<td>1.77E+06</td>
<td>2.14E+05</td>
</tr>
<tr>
<td>4608</td>
<td><strong>4.1863</strong></td>
<td>5.75E+08</td>
<td>6.51E+07</td>
<td>9.14E+10</td>
<td>2.11E+09</td>
<td>1.37E+07</td>
<td>1.40E+06</td>
</tr>
<tr>
<td>6912</td>
<td><strong>3.9029</strong></td>
<td>1.80E+09</td>
<td>1.41E+08</td>
<td>2.10E+11</td>
<td>5.86E+09</td>
<td>4.75E+07</td>
<td>4.86E+06</td>
</tr>
</tbody>
</table>

**Table 6.7:** Performance Counters collected in TileGx36 for LU Decomposition. Bold green and blue values represent the best values for Fine grain and OpenMP experiments.
Figure 6.20: Matrix Multiplication performance and hardware counters normalized to the maximum value for that counter
**Figure 6.21:** LU Decomposition Performance counters normalized to the highest value against OpenMP Hierarchical Implementation
technique has considerable advantage for L1 writes misses and a very small degradation for L1 read misses. These counters are order of magnitude higher than other miss counters and hence contribute to our advantage leading to higher performance (as showcased in Figure 6.21).

Thus, by doing the restructuring, we are shifting work from the higher levels of the memory hierarchy toward the lower levels and increasing the locality of the levels closest to the processor, thus, improving overall performance. However, we have a trade-off between the framework overhead that comes from the interplay between the framework, the actual buffer and certain caches features (e.g. the inclusiveness) as showcased in the performance counters. Even then, the performance improvements are still substantial.

![Figure 6.22: LU Decomposition Performance in GFlops](image)

Figure 6.22: LU Decomposition Performance in GFlops

In summary, by judicious data allocation and data restructuring guided by compiler driven data reuse analysis, we are able to shift the burden between the different
Figure 6.23: LU performance and hardware counters normalized to the maximum value for that counter
cache level hierarchies, yielding better performance outcomes for our two very relevant test cases.

6.3 Conclusion

In this chapter we showed that our jagged tiling and data restructuring techniques are highly effective in improving data locality and reuse in multicore systems. Although, there have been many works targeting multicore architectures, most of the prior works have focused on a single thread of execution regarding locality and reuse optimization. We discuss some of them in the next chapter.
In this chapter, we present some of the existing works that are closely aligned to the work presented in this thesis.

7.1 Compiler Techniques

With a goal of reducing memory access latency, there have been many optimization techniques that have evolved to improve data locality and parallelism. Some of these techniques use program transformations such that instructions are reordered to bring closely connected elements together to improve data locality and reuse. Loop transformation techniques such as loop reversal, permutation, skewing, unroll and jam are few of its examples. While such techniques are very effective, the order at which such optimizations are applied can significantly affect the quality of the generated code.

One of the most successful techniques used for locality optimization is the Iteration Space Tiling proposed by Michael Wolfe [117, 116]. This technique, also known as blocking, has been widely used to aggregate different dimensions of loop iterations to improve data locality and reuse at multiple levels of the memory hierarchy [69, 114, 115, 32, 84, 41, 94, 95, 66, 65, 6, 7]. In this technique, communication takes place only before and after the tile is computed and hence it reduces the amount of communication during execution. Such technique is applicable to many applications with permutable loops and is useful for both single-core and multicore architectures. In our framework, we take advantage of hierarchical tiling technique and exploit locality without compromising parallelism such that we can take full advantage of processing resources in multicore systems.
Wolf and Lam [115] introduced the concept of group-temporal and group-spatial where reuse is calculated based on accesses across multiple references both in time and space. Their technique uses reuse analysis to find a subspace where more reuse can be exploited and attempts to convert it as the innermost loop iteration. Bastoul[17] used a chunking function to partition codes into different chunks. Such chunks exploit both group temporal and group spatial locality for references using an access function, which allows reuse for both uniformly and non-uniformly generated references. It uses simple cache model to evaluate memory traffic and perform transformations geared towards improving locality. McKinley[84] used grouping of references accessing the same cache line to reduce the number of access going to the memory. Their technique uses a detailed cost model using loop permutation, fusion and distribution that reasons about locality and reuse for having any particular loop as the innermost.

The placement of tiles across different processing units and its execution affects the amount of communication. Griebl et al. [58] introduced a method to reduce the amount of communication by mapping computation to different processing cores at different time units known as space-time mapping such that parallelism and locality are exposed. Such method requires finding a legal transformation without changing the semantics or violating the dependencies. Although sometime finding the best schedule can require multiple transformations, when such transformations are unimodular, they can be unified to allow any number of transformations into a single transformation steps as proposed by Banerjee [13]. Wolf and Lam [115] used unimodular transformations and applied techniques like skewing and reversal to create permutable loop such that tiling can be applied for data locality improvement. Also McKinley et al. [84] created permutable loops using fusion and distribution.

Recently, with the advancement made by the linear algebra community, there have been many compiler optimization techniques and tools (both optimization and code generation) using the polyhedral framework [46, 47] have emerged. Loopo[59], PLUTO[22], CLooG[15] are examples of such tools. They have the ability to analyze, to apply multiple optimizations and to generate code automatically. The main idea
is to have efficient space and time mapping [60] while extracting multiple degree of parallelism that can be tiled with cost efficient hyperplanes. As a result, a tool like PLUTO [22] has provided a convenient way to generate tiled code that is optimized for locality and communication. However, such generated codes still are designed for coarse-grain execution. The reason behind this is two fold. Firstly, most of the scientific/stencil like applications have very complicated loop carried dependencies, which makes intra-tile parallelism difficult. Secondly, when tiled for one-level to exploit fine-grained execution, it is locality agnostic in relation to other threads working in parallel. Our framework, on one hand leverages the existing tiling techniques, whereas on the other hand uses fine-grained communication among closely associated tiles.

Irigoin [69] introduced hyperplane partitioning to create supernodes with inner parallelism. Such technique allowed emerging vector architectures to take advantage of two level of parallelism and exploit locality in two level caches. Although similar in terms of exploiting multiple levels parallelism, our granularity of parallelism is at tile level as we target multithreaded architectures. Some recent works have attempted to exploit intratile parallelism to improve resource utilization of multicores as well. Meng [85] proposed symbiotic affinity scheduling for improving inter-thread cache sharing. The algorithm does first partition of tasks based on number of cores and the second partitioning for threads sharing the lowest level cache. It is designed to reduce contention when threads share common data set and access patterns are predictable. Although, the idea of data sharing and improving locality is similar to ours, symbiotic tiling is done based on number of data caches and sharing between multiple threads happens within the tile.

7.2 Stencils and Tile Shapes

There is significant body of works that focus on specific kind of applications that are of interest to the scientific community. Since stencils are at the heart of many scientific computation and are very compute intensive, there has been a considerable body of research on how to improve the performance of these codes [72, 51]. Some
optimizations are focused mainly on stencil computation targeting higher order stencils where locality optimization is more important [98, 38]. Using a domain specific language that is translated to parallel Cilk code, Pochoir [108] provides efficient cache oblivious algorithm for stencils.

One of the difficulty of tiling is to find right shape that can map well to the iteration space and is also cost efficient [66]. Apart from classical rectangular or cubical shaped tiles, other shaped tiling shapes (e.g. hexagonal) has also been studied [66, 9, 41]. These tiling shapes are selected based on the access patterns of the loop nest such that communication becomes minimal and/or allows parallel start for certain types of applications [61][12].

As many stencil codes have the property in which one face of the iteration space can run concurrently, optimizing stencils for load balance has gained popularity. Krishnamoothry [78] proposed split tiling and overlapped tiling for stencil computations that are able to provide concurrent start. Similarly, Orozco [90] used diamond tiling and showed the effectiveness of such technique on a FDTD application using a many core architecture. Bandishti [12] automated a diamond tiling technique using PLUTO as a base tool to optimize for parallelism and load balance. The technique is driven by data dependence and hence creates diamond shaped tiles, which are very effective for applications with concurrent start. Although these techniques are interesting, they do not look into the possibility of exploiting locality at multiple levels of the memory hierarchy, as we do.

Besides the shape, another important tiling parameter is the tile size. Optimal tile sizes vary based on architecture features and application properties. Hence, some techniques use parametric measures that allow running program for different tile sizes [96]. In our work, we consider tile shapes that allow parallel execution of tiles at multiple levels of the hierarchy and our tile selection is based on parametric sweep.
7.3 Data Movement and Transformation

Due to its importance, data movement and locality is the target of substantial compiler and runtime research. Data centric optimizations in multithreaded environments have become more visible. There is a growing trend towards giving programmers control of data management in order to improve performance. The common approach uses information about access pattern, strides, data sharing and reuse to reduce memory access latency.

Overlapping data movement with computation and data prefetching based on access pattern are typical data movement orchestration strategies. In order to do so, memory access pattern needs to be known \textit{a priori} to the actual use of data. However, prefetching has to be timed such that data stay in cache and do not get evicted before the use happens. Techniques such as loop splitting / loop peeling \cite{89} to divide the loop in multiple phases where one can be used to prefetch data has been attempted before. However, loop peeling is not obvious for all nested loops. Prefetch when done unnecessarily, becomes just an extra overhead. To alleviate these shortcomings, systems like APACS \cite{80} provide an integrated solution for cache and prefetching with adaptive partitioning, prefetch pipelining and prefetch buffer management techniques. Although such solutions are promising, they require substantial hardware support. Moreover, techniques like Intel helper threads \cite{74} and Gan’s data percolation \cite{52} allows “special” threads to improve locality. In our case, we take advantage of data moved by threads in close proximity by grouping closely connected tiles and threads together for improved locality.

Garcia \cite{54} used percolation along with dynamic scheduling to improve the locality of dense matrix multiplications on the Cyclops-64 manycore architecture. He used helper threads to perform explicit data movement to a double-buffered SRAM. Baskaran \cite{14} used explicit data movement and index transformations to improve locality and reuse on scratchpads. He developed an automatic polyhedral based framework to utilize the available processing resources and manage data movement to hide memory access latency. This technique is very efficient on exploiting parallelism and managing...
data movement on GPUs where accesses happen in SIMD fashion. Our restructuring framework uses technique similar to his to map disjoint partitions to smaller space by subtracting the lower bounds. However, we do this only in tiled domain and perform further restructuring to improve access pattern.

Prior works have showed that array restructuring can be used to improve spatial locality in cases when loop transformation is not enough. Leung [79] with his restructuring technique allowed transformation of data to provide the better access pattern. Such technique is independent of loop carried dependencies, providing flexibility in the application of this transformation. However, careful consideration is needed to make sure this transformation is not detrimental when the same memory structure is accessed in multiple ways. Non-canonical data layouts [8, 29, 28, 30] have also been used to reduce memory access latency. Indeed, our approach also uses non-canonical layouts, however we do so to improve residence for the caches closest to the processor – sometimes at the expense of the farther remote caches – when reuse exists across parallel threads.

Data transformation work has also been done to improve vectorization by Jang [70]. Based on the access pattern, different data transformation rules are used to have contiguous access, enabling vectorization. Jeremiassen [71] showed that false sharing misses can be reduced by performing data transformations. It uses a transpose when different processors access adjacent elements, uses indirection when data restructuring is not possible and does padding for shared data writes when falsely shared. Also, Rivera [97] showed how cache conflicts can be avoided using padding techniques. Lu [82] used data layout transformations using the polyhedral model to reduce hot spots and bank contention.

In some cases, both, loop and data transformations, are not able to produce efficient code independently. Cierniak and Li [32] proposed a technique to unify these two techniques using mapping and stride vectors. Within the loop nest, different array structures can use different mapping based on access pattern, allowing contiguous access in memory space. Although their techniques are difficult in cases of multiple
references, single reference access transformations work well. Also, Kandemir [73] proposed an integrated framework that uses loop transformation for temporal locality and data transformation for spatial locality. Our framework uses both loop and data transformation for locality optimization. However we leave the integration of two for future work.

One of the issues applying data transformations is the extra memory space requirement. Darte [36] proposed a mathematical framework that maps indexes to a limited memory space. The strategy allows effective use of shared space, like e.g. scratchpad, and ensures conflict free mapping by avoiding conflicting elements with simultaneous live. Also, such a technique can be used to hold intermediate values and maximize reuse. We believe that our framework can take advantage of such technique for optimizing restructuring space.

7.4 Other Techniques

Kodukula [75] used data blocking based on its flow through the memory hierarchy to optimize memory access. His approach selects a sequence of blocks that is touched by a processor and executes statements associated with those blocks. Nevertheless, instances with complex program structures and dependencies make this transformation approach unwieldy.

There also exist frameworks that allow users to express parallelism and locality in an organized fashion. Bikshandi [19] introduced hierarchical tiled array (HTAs) that manipulates tiles using array operations to represent parallel computation and communication. Also, it allows multiple levels of parallelism to exploit locality and parallelism at multiple levels of the memory hierarchy. Such parallelism is very well structured and is designed to work on both single and distributed machines. This is a very elegant way of expressing parallelism and exploiting locality, thus can be a nice fit as a programming paradigm to test ideas of our group locality framework.

Current fine grain runtime concentrate on the interaction of many threads
and had dealt with memory bandwidth and contention problems in many interesting ways. The Efficient Architecture for Running THreads (EARTH) [110] uses the un-interruptable fine grain threads, called fibers, which shares their activation frames. The SWARM framework developed by ET International [68] has similar concepts as EARTH, but added support for direct access shared memory structures, placement information and different “codelet” pattern operations that allow more efficient use of data. Also, the most recent development is Rice’s Open Community Runtime (OCR) [26] system. It is designed as a fertile ground to experiment with new techniques and systems. The dire need for fine grain execution technique to reach the next milestone in performance has also led to a dataflow language and compiler for multicore architecture [112].

Most of the original compiler work presented here has a single thread focus or starkly favors coarse grain parallelism. Improving on this, a fine-grain collaborative view on code generation for multiple threads can yield significant improvements. This thread collaboration can extend beyond scheduling techniques and encompass data restructuring as well; a topic that was not considered in the fine-grain multithreaded work presented above. The approach presented in [43] aims to solve these issues with a hierarchical tiling framework for multicore clusters. They exploit inter and intra (with threading) node parallelism using tiling. Moreover, they exploit NUMA aware allocation and take advantage of vector register blocking to further increase performance. Stencil codes are used to showcase benefits. Although their approach is similar to our framework, key differences in our favor are the creation of more intra-tile parallelism (similar to their intra-node parallelism) with our jagged tiling technique.
Chapter 8

FUTURE WORK AND CONCLUSION

The framework presented in this thesis opens doors for many future research. The current implementation uses static information gained by the compiler for exploiting parallelism and locality at multiple levels of the memory hierarchy. Based on the architectural features, current grouping of threads is statically determined and it stays consistent throughout the execution. Future work can investigate the dynamic behavior of such threads where based on architectural and algorithmic state, they can switch between doing computation and helping as memory threads for data movement. As the program behavior can change at different point in execution, giving threads the flexibility to switch between groups dynamically and allowing group size to grow or shrink can add another dimension to optimization.

As it stands to be true that memory subsystems have not been able to keep up with the accelerating pace of computing resources, there has been a dire need to reduce memory access overhead to gain higher performance. Many optimization techniques over the years have provided significant boost in performance by reducing memory access cycles. These techniques, although very effective, have often overlooked the collaboration opportunities that massively parallel systems offer. When resources are in abundance and are working in parallel, where there is a chance of conflicts there also exist opportunities for collaboration. For the most part, existing techniques either use coarse grain execution as a norm or are locality agnostic when fine grain execution is used. Such strategies can lead to resource underutilization and cause missed opportunities to maximize reuse among threads working in close proximity in time and space.
In this thesis, we presented a novel framework that is geared towards improving parallelism and locality in multithreaded environment. Our novel tiling technique allows multilevel parallelism such that threads can form a group based on their physical affinity and take advantage of locality without compromising parallelism. Similarly, our restructuring technique allows groups of threads to improve access patterns with minimal overhead. Our experimental results show that our technique is highly effective to improve locality and performance. For Jagged tiling, our experiments on an Intel Xeon Phi board shows up to 32.25% improvement against instances produced by a state-of-the-art compiler. Similarly, our restructuring technique on Tilera TileGX shows advantage up to 31% over optimized OpenMP kernel and up to 16% against our fine-grain runtime systems for selected kernels.

Our current implementation takes advantage of a powerful polyhedral framework. Such scheme has its limitations when it comes to non-affine spaces where application’s data is sparse, accesses are indirect and indices calculations are not monotonous. However, there are techniques for handling non-affine spaces. One of such technique is the use of inspector/executor paradigm where information gained during runtime is used for optimizations [87]. Additionally, prior works have also shown code generation using non-affine mapping with uninterpreted function of indices in the enclosed loop [111]. Such ideas can be integrated with our framework to handle non-affine applications.

Our framework leverages the existing PLUTO framework that is designed to produce hyperplanes optimized for communication. With such hyperplanes as a first level of our building block, second level hyperplanes are created for exploiting a multi-level parallelism. This approach is based on the notion that locality and communication are the major bottlenecks to performance in multithreaded systems. Although multi-objective cost functions with weighted sum can be used for finding an efficient solution, such weights are difficult to determine and the search requires attempting all positive values of such weights [55]. Adding additional parameters like tile size can create search space explosion making the scheme impractical. Also, dynamic behavior of the
network and the bandwidth affected by the selection of tile sizes can be very imprecise at compile time. Instead, our approach uses hyperplanes generated with communication minimization cost function and build upon it for exposing parallelism.

Our framework integrates a polyhedral code generation with a data flow scheme of execution. Such scheme gives the code generation framework more flexibility as the first level tile dependencies are handled during runtime. Thus, it gathers compile time information and passes it to the runtime for efficient balance between locality and parallelism. We believe that as we tread towards massively parallel architectures, such integration is necessary to provide practical solutions for high performance systems.

The idea presented in this thesis however is not limited to any specific framework and is a first stepping stone in understanding the complex interplay between computational components and their shared resources. In the past, this interplay was ameliorated because of hardware and software design but as many core architectures become bigger, the effects of unrestrictive interplay can be damaging to the performance of the systems. Thus, analyzing and creating techniques to optimize computational pattern from a resource perspective is one of the most paramount endeavors of the next generation of research.
BIBLIOGRAPHY


132


Appendix A

RIGHTS AND PERMISSIONS

**SPRINGER LICENSE**
**TERMS AND CONDITIONS**

Sep 29, 2015

This is a License Agreement between Sunil Shrestha ("You") and Springer ("Springer") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Springer, and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

<table>
<thead>
<tr>
<th>License Number</th>
<th>3719460166019</th>
</tr>
</thead>
<tbody>
<tr>
<td>License date</td>
<td>Sep 29, 2015</td>
</tr>
<tr>
<td>Licensed content publisher</td>
<td>Springer</td>
</tr>
<tr>
<td>Licensed content publication</td>
<td>Springer eBook</td>
</tr>
<tr>
<td>Licensed content title</td>
<td>Jagged Tiling for Intra-tile Parallelism and Fine-Grain Multithreading</td>
</tr>
<tr>
<td>Licensed content author</td>
<td>Sunil Shrestha</td>
</tr>
<tr>
<td>Licensed content date</td>
<td>Jan 1, 2015</td>
</tr>
<tr>
<td>Type of Use</td>
<td>Thesis/Dissertation</td>
</tr>
<tr>
<td>Portion</td>
<td>Excerpts</td>
</tr>
<tr>
<td>Number of figures/tables/illustrations</td>
<td>None</td>
</tr>
<tr>
<td>Author of this Springer article</td>
<td>Yes and you are a contributor of the new work</td>
</tr>
<tr>
<td>Order reference number</td>
<td>None</td>
</tr>
<tr>
<td>Title of your thesis / dissertation</td>
<td>A Framework for Group Locality Aware Multithreading</td>
</tr>
<tr>
<td>Expected completion date</td>
<td>Dec 2015</td>
</tr>
<tr>
<td>Estimated size(pages)</td>
<td>135</td>
</tr>
<tr>
<td>Total</td>
<td>0.00 USD</td>
</tr>
</tbody>
</table>

Figure A.1: Springer License