APPLICATION-SPECIFIC MEMORY SUBSYSTEM BENCHMARKING

by

Mahesh Lakshminarasimhan

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2019

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Mahesh Lakshminarasimhan

Thesis Title: Application-Specific Memory Subsystem Benchmarking

Date of Final Oral Examination: 5th March 2019

The following individuals read and discussed the thesis submitted by student Mahesh Lakshminarasimhan, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Catherine Olschanowsky, Ph.D.	Chair, Supervisory Committee
Steven Cutchin, Ph.D.	Member, Supervisory Committee
Hoda Mehrpouyan, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Catherine Olschanowsky, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGMENTS

I am grateful to my graduate advisor Dr. Cathie Olschanowsky for her support and encouragement throughout the course of my Masters degree. She has taught me most of what I know about research and has helped me sharpen my technical writing and presentation skills. I am indebted to her for connecting me with great opportunities and providing invaluable guidance in shaping my career. I also thank her for funding me with graduate research assistantship supporting my graduate studies.

I am thankful to my supervisory committee members Dr. Steven Cutchin and Dr. Hoda Mehrpouyan for their constructive feedback in the successful completion of my thesis. The suggestions from Dr. Cutchin for data visualization have been very valuable.

The current and former members from the ADaPT Data Flow Optimizations lab have been very good friends and collaborators. I was fortunate to be supported by Travis Adsitt, the undergraduate researcher, in developing the late analysis tool and the benchmark framework. I greatly benefited from the productive discussions I have had with Eddie Davis.

I would like to thank the staff members of OIT Research Computing, specifically Kelly Bryne and Tyler Bevan for their support in addressing the technical issues that arose with the R2 HPC cluster. I appreciate the CS department staff for helping me with various administrative tasks. I also thank Ben Peterson for his assistance with the Onyx cluster and the workstations in the research neighborhood.

This research utilized the high performance computing support of the R2 compute cluster (DOI: 10.18122/B2S41H) provided by Boise State University's Research Computing Department. A part of thesis was funded by the U.S. National Science Foundation (NSF) award #1849463.

ABSTRACT

Application performance often depends on achieved memory bandwidth. Achieved memory bandwidth varies greatly given specific combinations of instruction mix and order, working set size, and access pattern. Achieving good application performance depends on optimizing these characteristics within the constraints of the given application. This task is complicated due to the lack of information about the impact of small changes on the performance. Some information is provided by benchmarks, but most memory benchmarks are confined to simple access patterns that are not representative of patterns found in real applications.

This thesis presents AdaptMemBench, a configurable benchmark framework designed to explore the performance capabilities of compute kernels extracted from applications. AdaptMemBench provides a framework to emulate application-specific memory access patterns. A set of templates manages standard timing and measurement tasks. The build system accommodates the polyhedral model, making the framework provides a convenient testbed for potential code optimizations.

AdaptMemBench supports reproducibility in experimental results and facilitates sharing results. Given that small changes in benchmarks have a large impact on performance a common framework isolates the measured portions of code. This eases the process of rerunning experiments and porting to new systems. The strengths of AdaptMemBench are demonstrated through a collection of case studies on common compute kernels including: streaming patterns, multidimensional stencils, and sparse matrix operations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HPC – High Performance Computing

SpMV – Sparse Matrix-Vector Multiplication

CPU – Central Processing Unit

DRAM – Dynamic Random Access Memory

NUMA – Non-Uniform Memory Access

ISL – Integer Set Library

COO – Coordinate

CSR – Compressed Sparse Row

BCSR – Block Compressed Sparse Row

ELL – Ellpack

DIA – Diagonal

PAPI – Performance Application Programming Interface

FLOP – Floating Point Operations

FMA – Fused Multiply Add

pmbw – Parallel Memory Bandwidth

ECM – Execution-Cache-Memory

LC – Layer Conditions

Chapter 1

INTRODUCTION

This thesis presents *AdaptMemBench*, a configurable memory benchmarking framework that measures achieved memory performance by emulating application-specific memory access patterns. This framework, directed at reliability and reproducibility of performance experiments in large scale projects, is designed to measure execution characteristics of isolated portions of code extracted from applications. *AdaptMemBench* provides a starting point to identify performance bottlenecks, evaluate potential optimizations, and explore the potential gains of those optimizations.

Many scientific applications are considered memory bound, meaning that interactions with memory are the limiting factor for performance improvement. In computer hardware, the memory performance has improved more slowly than floating point performance, the diverging exponential increase between the two has created a bottleneck referred to as the memory wall [68]. The CPU speed of the fastest available microprocessors is growing at approximately 80% [39] per year, while the speed of memory devices has been increasing only at the rate of 7% per year [22]. This ever-growing gap significantly impacts the performance of most scientific applications that constitute memory bound compute kernels.

The growing complexity of modern architectures makes it difficult to write memory-efficient software and achieve sustained application performance across architectures.

Modern memory architectures are hierarchical, hiding some of the impact of the memory wall. Smaller, faster, and more expensive cache memories are placed between the processor and main memory reducing latency. The depth and configuration of memory hierarchies is constantly changing. In recent high performance computing architectures, the system memory is physically fragmented and the memory hierarchies are getting deeper. The speed of processing units has been increasing but not at a rate that satisfies demand. Modern architectures have incorporated high levels of parallelism to improve performance.

Multi-core CPU architectures consist of multiple processing cores, each of which can concurrently execute its own sequence of instructions, increasing overall system performance. Existing multicore hardware configurations are more diverse and complex, complicating the development and optimization of application code. This is exacerbated with resources shared among cores depending on the residency of the data being accessed. Parallelism and cache coherency protocols increase the design complexity and change the impact of code optimizations.

Memory bound application codes spend a significant fraction of their processing time moving data across the memory subsystem, causing performance bottlenecks. Optimizing code to take the best advantage of the memory subsystem has been researched for many years. Reducing working set sizes is considered a good first step in optimization to take advantage of the caching capability of machines. However, optimizing is more complex than that, especially when dealing with shared memory parallelization. The application characteristics including memory access patterns, instruction mix and order, data sharing across caches, and vectorizability must all be considered in concert.

A combination of implementation challenges discourages effective code optimization. The hardware, and therefore performance characteristics, change significantly between hardware releases; testing and verifying optimizations within the context of a large application is error prone; and optimizations obfuscate the primary computation in the application. Automating optimizations in a compilation stage resolves the last two challenges, but the compiler still needs an indication of which optimizations will be successful. Given the difficulty around manipulating access patterns in situ when working with a large application, fewer optimization strategies are attempted and potential performance improvements are overlooked. Additionally, performance tools such as hardware counters, remain difficult to use in the context of a large application.

A framework that allows extracted code to be isolated and measured will benefit the optimization process for specific projects. During the exploration and experimentation phase, many different variants of the same code are produced. Tracking the differences between variants and maintaining correct execution becomes time consuming and challenging. A shared framework that supports experimentation and tracks code versions while outputting metadata with measurements will ease this challenge.

This thesis presents a benchmarking framework to explore the design landscape of a target architecture [32]. The *AdaptMemBench*[1] framework can be used to measure system performance and to guide application-specific optimization decisions. Expensive kernels extracted from larger applications can be manipulated in isolation to find the best optimization strategies. The framework reduces the amount of code that is transferred and provides mechanisms to experiment with data storage layout, execution order, and parallelization strategies.

AdaptMemBench provides several execution templates. The templates are com-

[1]https://github.com/BoiseState-AdaptLab/AdaptMemBench

bined with user-provided code segments. The templates provide a common command line interface, handle all timing and hardware counter code, and output metadata and measurements in a common format. The code segments provided by the user can be expressed as C code or by using the polyhedral model. The latter provides a convenient mechanism for optimization experiments. Combining benchmarking techniques with compiler transformation techniques enhances reproducibility of experimental results, with separate groups required to only exchange the core of the code in question.

Several benchmarks [38, 28, 50, 49, 53, 6, 37, 60, 18, 2] exist that measure machine performance, with the benchmarking results conveying essential information about the application performance on the memory hierarchy of the machine. Existing memory benchmarks [38, 50, 49] measure performance using a limited collection of streaming access patterns. However, benchmarking application-specific patterns that tend to be more complex remains a challenge. Current benchmarks [39, 28] are further constrained by the data sizes which can be executed, specifically in the higher levels of the memory subsystem.

AdaptMemBench differs from previous efforts by incorporating polyhedral code generation. This creates a configurable benchmarking framework that measures achieved memory bandwidth while mimicking application-specific memory access patterns. The existing set of benchmark templates support a variety of kernels which involve arbitrary random reads and writes with non-strided accesses. The Polyhedral model [64] simplifies writing the initial benchmark and provides a mechanism to automatically transform the code. Furthermore, our benchmark supports parallel applications and systems, and measures memory performance for data sizes across all levels of the memory hierarchy.

1.1 Problem Statement

Performance optimization is often abandoned because of its time-consuming nature, causing large scientific applications to run inefficiently on large supercomputers. The performance of memory-bound applications is determined by a combination of memory access patterns, instruction mix, and instruction order. Small changes to these characteristics impact achieved memory performance significantly. The changes are typically unpredictable, requiring a manual search of the optimization space. This is cumbersome and error-prone within large scientific applications.

1.2 The Proposed Solution

This thesis presents a configurable memory performance benchmarking framework that enables measuring application-specific access patterns. The framework characterizes the memory performance of compute kernels isolated from applications. It supports reproducibility in performance results that facilitates rerunning experiments and porting to new systems. Accommodation of the polyhedral model in the build system provides a convenient testbed for potential code optimizations. Access to hardware performance counters offers insights on the interaction of the application with the memory subsystem.

1.3 Contributions

The primary contribution of this thesis is the AdaptMemBench benchmarking framework along with the comprehensive case studies on common compute kernels using AdaptMemBench that demonstrates its strengths. The contributions of this thesis include:

- AdaptMemBench, a configurable benchmarking framework for application-specific memory performance characterization.
- A performance study on common compute kernels for the impact of implicit barriers, shared data spaces and false sharing. An interleaved execution schedule is proposed with demonstrated performance speedup for the triad pattern.
- An evaluation of existing spatial and temporal tiling strategies for multidimensional Jacobi patterns on modern multicore architectures, using polyhedral code generation.
- A comparison of the performance variation of Sparse Matrix-Vector (SpMV) multiplication implemented using different sparse matrix representations benchmarked with real-world sparse matrices.

Chapter 2

BACKGROUND

This chapter gives an overview on multicore architectures, the polyhedral model, and sparse matrix representations.

2.1 Modern Multicore Architectures

This section provides preliminaries on modern computer architectures emphasizing multicore hardware, shared memory systems, hierarchical memory structures, and caching in multicore systems.

2.1.1 Node Configurations

Over the past few decades, with the increasing demand for maximizing memory performance, computer hardware has evolved with larger and faster memories, along with faster and more complex processing units. Typically, every new processing generation increased clock frequency allowing for more operations to be performed per time unit. However, this trend had to be slowed down due to the *power wall*, with high power consumption and the ability to handle on-chip heat reaching its physical limit.

Modern compute nodes comprise multicore processors. A collection of simple compute cores work together in parallel to replace a single monolithic compute core.

The parallel cores are tightly coupled on a single chip. By running concurrently, they yield better overall performance than the equivalent single cores, leading to the shift from sequential computing to parallel computing. Most existing large-scale scientific applications thus have to be parallelized, tuned, and deployed on multicore processors.

Shared memory systems consist of multiple processors or processor cores that share main memory, and communication between them occurs through the shared memory itself. Shared memory systems are typically implemented with a physically distributed memory where each processor has its own local memory that can be accessed by the other processors. In order to achieve optimal parallel performance, it is non-trivial to understand the memory hierarchy of the system.

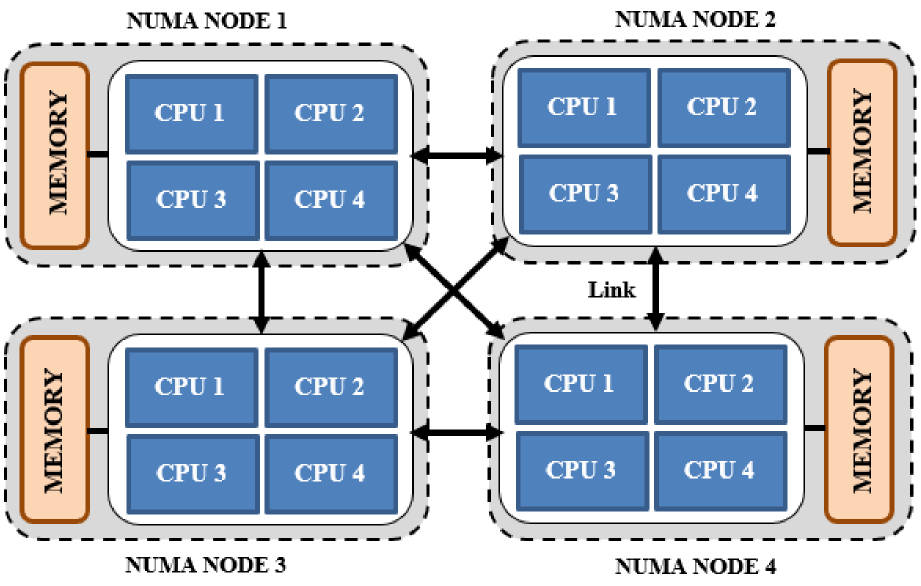


Figure 2.1: A modern NUMA system, with four nodes and four CPUs per node.

Modern multicore systems are based on Non-Uniform Memory Access (NUMA) architecture, where cores are grouped together in a set of nodes. Each core has its own local memory and each node is connected with one another by means of high speed interconnect links, as indicated in figure 2.1. This is in contrast to the

Uniform Memory Access (UMA) architecture, where a single shared bus connects all cores to main memory, thereby eliminating memory bandwidth bottlenecks. In NUMA system, the address space is global and remote memory accesses can introduce latencies, implying that high memory performance can be utilized only if memory accesses are scattered over all cores.

2.1.2 Hierarchical Memory Structures

The arithmetic units of current processors perform operations on registers, which typically have very limited storage capacity, forcing CPU to often load data to and flush data from registers. This requires processors to fetch data directly from Dynamic Random Access Memory (DRAM) frequently with very low access speed, causing latency. High performance processors overcome this limitation by inserting small low-latency cache buffers between the slow memory and high speed registers of the CPU.

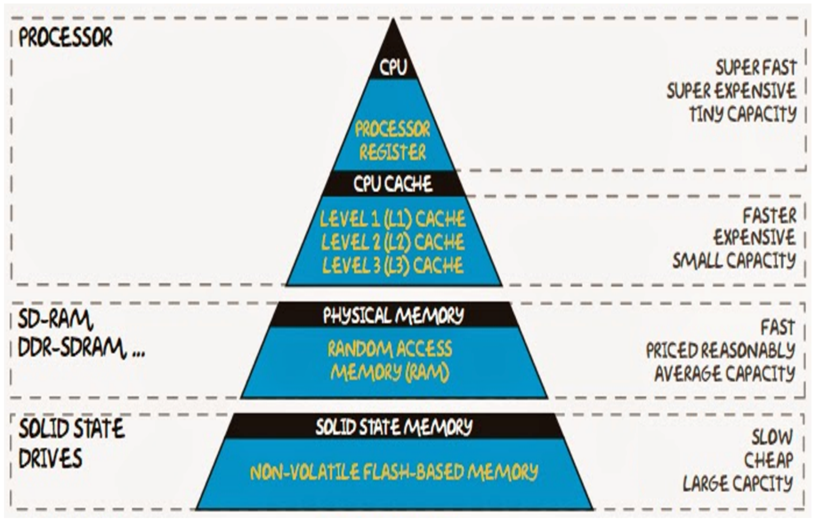


Figure 2.2: Memory hierarchy in modern computers.

Caches

Several layers of caches are present to uncouple fast registers from slow main memory. Typically, a combination of a small low-latency L1 memory is backed by a higher capacity, yet slower, L2 memory, followed by L3 memory with much larger memory and lower speed. In previous generation multicore hardware systems, the cache levels below L1 were shared among the processor cores. Most recently, L1 (both instruction and data caches) and L2 caches are dedicated to each core and the L3 is shared among cores that are on a single socket. A machine may have multiple sockets making the memory hierarchy deeper and more complex. Figure 2.3 shows the shared memory organization of R2 HPC cluster at Boise State University, which has two sockets with each of them consisting of 14 processor cores.

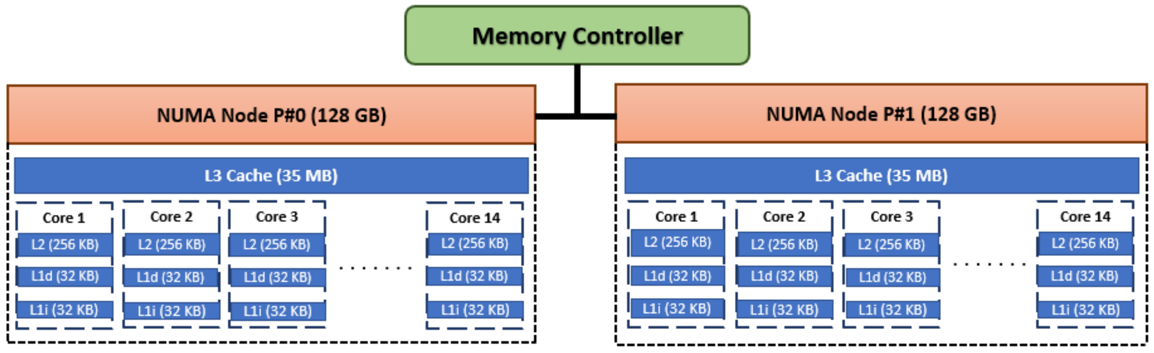


Figure 2.3: Memory organization of R2 HPC Cluster

False Sharing

When running a diverse range of applications, there may be substantial periods of execution during which performance degrades due to a mismatch between the memory system requirements of the application and the memory hierarchy implementation.

Thread synchronization, saturated memory bus, and false sharing are some of the circumstances that limit performance.

Caches read from, or write data to the DRAM in blocks of memory called cache lines. A cache line is typically 64 bytes or 128 bytes wide. When a multithreaded code reads data, caches contain a subset of information located in the DRAM and a cache can be private to its cores. Simultaneous updates of elements in the same cache line from different cores invalidate the entire cache lines, although these modifications are logically independent of each other. Other cores accessing a different element in the cache line find it marked invalid. They are compelled to load a more recent copy of the cache line from memory, though the accessed element is not modified. While a cache line is being updated, access to the elements in the line is restricted.

This circumstance is called *false sharing*. Though different threads are not sharing data, they are involuntarily sharing a cache line. This leads to an increase in memory traffic and causes overhead. False sharing occurs when multiple processors share data within the same cache line, or when shared data is modified by multiple processors. False sharing can be avoided by padding arrays to the end of a cache line to ensure that the array elements begin on a cache line boundary. Another solution is by using thread-local copies of data. Compiler's optimization features could also be used to eliminate false sharing.

Prefetching

Prefetching is a mechanism employed by processors to fetch data into the cache that may be accessed in the near future. This minimizes latencies of data loaded from memory to caches. Prefetching can either be performed at the hardware level or be initiated by the software. Hardware prefetchers are based on algorithms that monitor access patterns and predict future access. Hardware prefetching can effectively be utilized by streaming access of sequential elements.

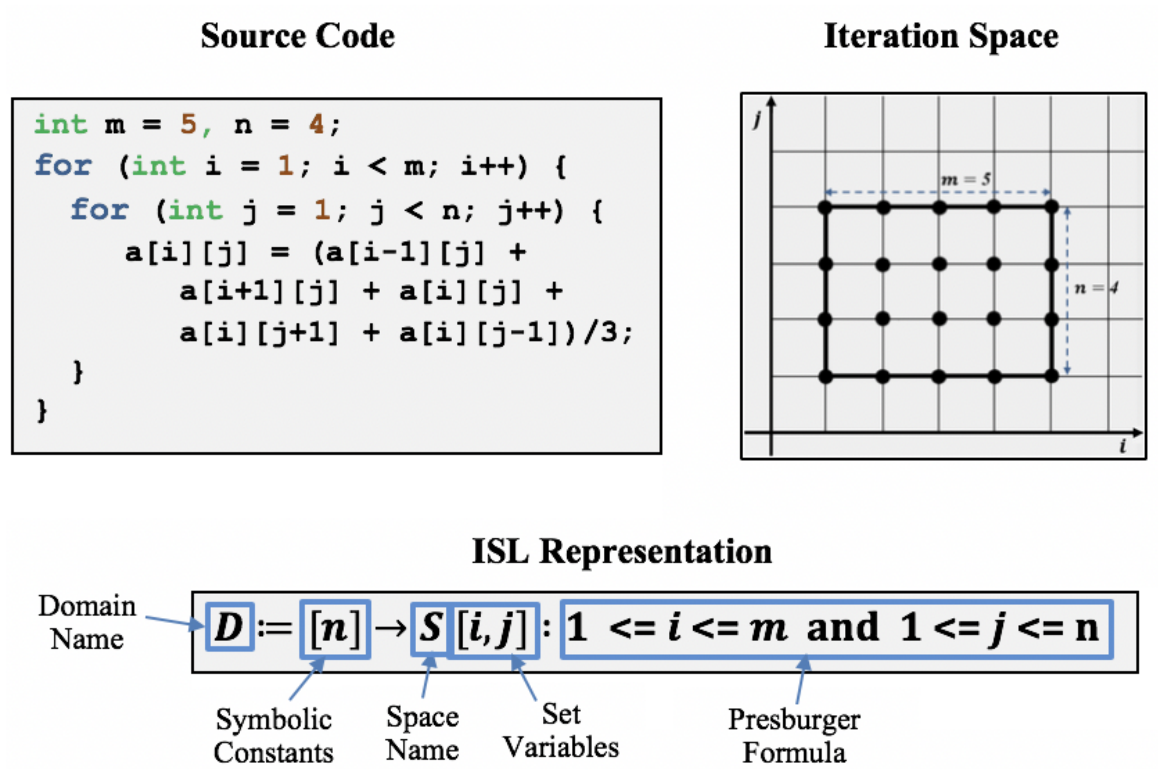
2.2 The Polyhedral Model

The polyhedral model expresses and manipulates loop constructs using a well-defined mathematical model. An iteration space that describes a loop nest is considered an affine space if the lower and upper bounds of each loop can be expressed as a linear function.

The execution schedule of a loop nest can be represented with the following components:

- *Iteration Space:* the set of statements in the block and the loop iterations where instances of the statement are executed.
- *Access Relations:* The set of reads, writes, and may-writes that relate statement instances in the iteration space to data locations.
- *Dependences:* The set of data dependences that impose restrictions on the execution order.
- *Schedule:* The overall execution order of each statement instance represented by a lexicographically ordered set of tuples in a multidimensional space.

Figure 2.4 shows a loop nest for solving the heat equation. The associated iteration space is shown graphically as a two-dimensional space (i, j). Each node in the graph represents an iteration. Polyhedral representations are expressed using Presburger arithmetic, and can be extracted from source code by analyzing loop bounds and array subscript expressions. The Presburger formula for this example is shown at the bottom of the figure.

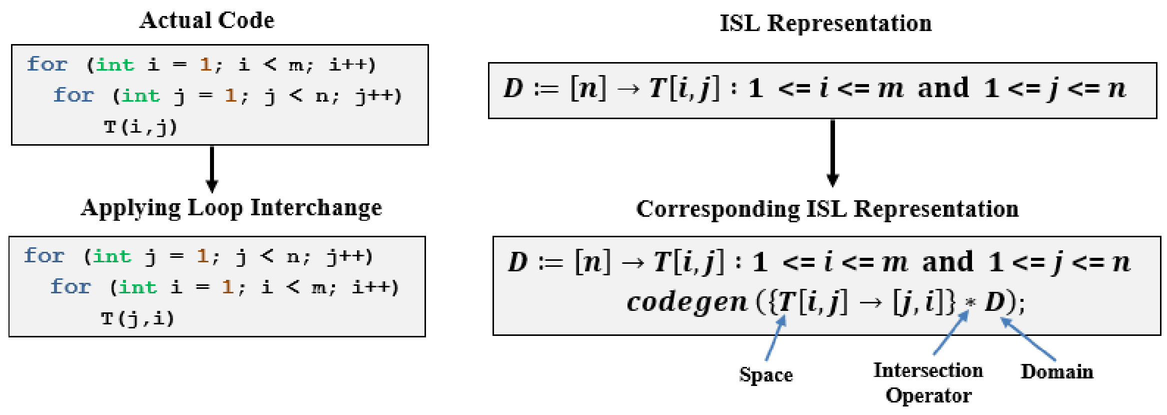


```
int m = 5, n = 4;
for (int i = 1; i < m; i++) {
  for (int j = 1; j < n; j++) {
     a[i][j] = (a[i-1][j] +
        a[i+1][j] + a[i][j] +
        a[i][j+1] + a[i][j-1])/3;
  }
}
```

Figure 2.4: An example of polyhedral code generation with ISCC/ISL.

Code generation is performed on sets through polyhedral scanning, the result is a control flow that produces the iterations in lexicographical ordering. As expressed in Figure 2.4 the original code would be produced. The polyhedral model provides a separation of concerns between the statement instances and the corresponding execution order. Polyhedral optimizations change the execution schedule without affecting the set of statements that are executed. Transformations on the code are realized through the application of relations (or functions).

Loop interchange is a loop transformation that switches the order of two loops. Figure 2.5 shows the relation used to apply loop interchange for the code in Figure 2.4. For the relation from `{i,j}` to `{j,i}`, we apply the transformation on the execution domain defined, using the intersection operator. Other transformations such as fission, fusion, reversal, skewing, and tiling can be performed with ease using the polyhedral model.



```
for (int i = 1; i < m; i++)
  for (int j = 1; j < n; j++)
      T(i,j)
```

$D := [n] \rightarrow T[i,j] : 1 <= i <= m \text{ and } 1 <= j <= n$

```
for (int j = 1; j < n; j++)
  for (int i = 1; i < m; i++)
      T(j,i)
```

$D := [n] \rightarrow T[i,j] : 1 <= i <= m \text{ and } 1 <= j <= n$
$codegen(\{T[i,j] \rightarrow [j,i]\} * D);$

Figure 2.5: An illustration of loop interchange using ISCC.

In this thesis, to automatically generate schedules for the application kernel initialization, execution, and validation, we use the ISCC [64] polyhedral code generation tool, which offers an interface to the functionality provided by Integer Set Library (ISL) [63] and Barvinok library [62]. This tool enables the end user to manipulate sets and relations and generate source code reflecting their input.

This tool, however, is restricted to iteration spaces that are affine. Affine loop bounds are considered constant at runtime time and allows for static analysis at compile time. A significant amount of work has been done to expand the iteration spaces and schedules that can be represented, including work that uses schedule trees for code generation within ISL [63].

The Omega+ code generation tool incorporates iteration bounds based on runtime information using uninterpreted functions [11]. Uninterpreted functions are mathematical symbols that act as placeholders, and consist of only a name and an n-ary form. The runtime realizations of uninterpreted functions can be implemented as arrays, functions, or macros. The input and output of the uninterpreted functions are unknown at compile time and cannot be statically analyzed, hence the term. These functions can be applied to represent non-affine loop bounds that are data-dependent or based on functions that are unknown at compile time.

Even with recent advances, the polyhedral model cannot express all C kernels. An example is conditional statements in an iterative loop that are not tied to the loop iterator. Recursive statements and while loops can also not be represented using the polyhedral model. Hence, polyhedral compilation can be bypassed in the AdaptMemBench specifications to represent such kernels in the benchmark.

2.3 Sparse Matrix Storage Formats

Matrices used in real-world applications are typically large and sparse, meaning most data elements are zero. A simple method to store a sparse matrix is using a two-dimensional dense array to store all the components of the matrix including the zero valued elements. It offers random access to the value at any coordinate in constant time. However, the dense representation is inefficient since a lot of memory space is consumed by storing zeros, degrading performance due to unnecessary computations. It can be impractical to utilize a dense array to store very large and hypersparse matrices due to lack of memory.

Several sparse matrix representations have been proposed that exploit the sparsity of its components to compactly use memory and to avoid unnecessary computations. In this thesis, we represent sparse matrices in the Coordinate (COO) format [5], Compressed Sparse Row (CSR) [46] , Block CSR (BCSR) [25], Diagonal (DIA) [30] and Ellpack (ELL) [46], each of which is described below.

Coordinate (COO) format

Coordinate format [5] keeps a list of non-zero values and two index arrays that map into dense indices. In figure 2.6, array A stores the non-zero values and the supplementary arrays `row` and `col` respectively are used to represent the row and column coordinates of each non-zero element. Many sparse matrix storage file formats like the Matrix Market exchange format (`.mtx` files) [10] used in this thesis mimic the COO format. This minimizes pre-processing cost as constructing the COO format requires only appending the values and the coordinates to the `val`, `row` and `col` arrays. This format could be used when the sparse matrix is used only once in the code, since it is more efficient to represent data with the COO format instead of converting existing data to another format. This format, however, does not provide efficient random access and when used more often, performance degrades.

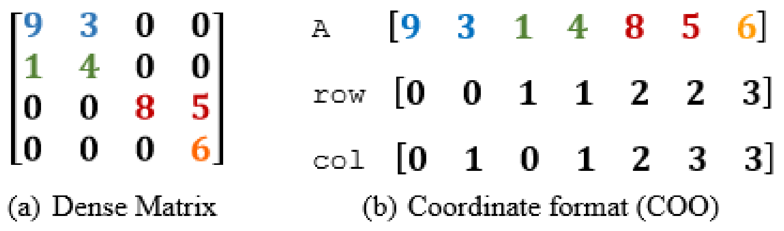

$\begin{bmatrix} 9 & 3 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 0 & 0 & 8 & 5 \\ 0 & 0 & 0 & 6 \end{bmatrix}$
(a) Dense Matrix
A [9 3 1 4 8 5 6]
row [0 0 1 1 2 2 3]
col [0 1 0 1 2 3 3]
(b) Coordinate format (COO)

Figure 2.6: The Coordinate (COO) format representation.

Compressed Sparse Row (CSR)

The CSR [46] format compresses out the redundantly stored row coordinates in the COO format. From the COO format, the `val` and `col` arrays are retained, but the `row` array only has one element per row, indicating the first element of that row (figure 2.7). Compression of the row coordinates increases the performance of memory intensive computations.

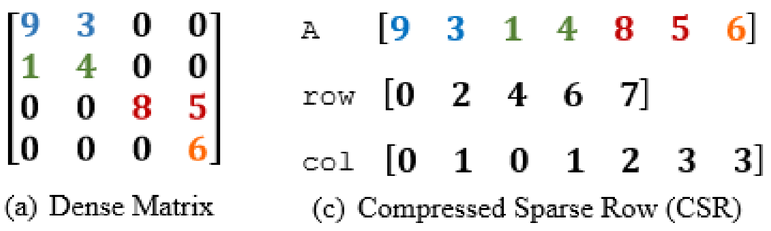


Figure 2.7: The Compressed Sparse Row (CSR) format representation.

Block Compressed Sparse Row (BCSR)

In the Block CSR (BCSR) format [25], the non-zero elements are clustered together in a collection of small dense blocks and the blocks where necessary are padded with zero values. The array `A_prime` consists of all such non-zero blocks (figure 2.8). The `block_col` array tracks the column of the upper left element of each non-zero block. The `block_col` array indicates the index of the first element of the rows in the array `A`. This reduces storage and the computation could be performed in a small dense array.

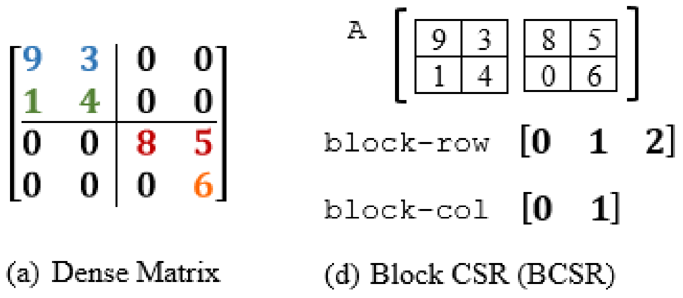


Figure 2.8: The Block CSR (BCSR) format representation.

Diagonal (DIA)

The Diagonal (DIA) format [30] is most effective for banded matrices, which have non-zero elements restricted to a few dense diagonals. Such matrices are common in grid and image applications. The `offset` supplementary array represents the offset from the main diagonal as shown in figure 2.9.

$$\begin{bmatrix} 9 & 3 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 0 & 0 & 8 & 5 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

(a) Dense Matrix

$$\texttt{A} \begin{bmatrix} 0 & 9 & 3 \\ 1 & 4 & 0 \\ 0 & 8 & 5 \\ 0 & 6 & 0 \end{bmatrix}$$
$$\texttt{Offsets} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

(e) Diagonal (DIA)

Figure 2.9: The Diagonal (DIA) format representation.

Ellpack (ELL)

$$\begin{bmatrix} 9 & 3 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 0 & 0 & 8 & 5 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

(a) Dense Matrix

$$\texttt{A} \begin{bmatrix} 9 & 3 \\ 1 & 4 \\ 8 & 5 \\ 6 & * \end{bmatrix} \quad \texttt{col} \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 2 & 3 \\ 3 & * \end{bmatrix}$$

(f) Ellpack (ELL)

Figure 2.10: The Ellpack (ELL) format representation.

The Ellpack (ELL) format [46] is suited for sparse matrices that contain a bounded number of non-zeros per row. This format uses a two-dimensional matrix with a fixed number of non-zeros per row and rows with fewer non-zeros are padded with zero values. The `col` matrix keeps track of the columns for the non-zeros, as in figure

2.10. ELL is more efficient when most rows have a similar number of non-zeros, because of a fixed number of iterations and no indirection in the loop bounds.

Chapter 3

THE *ADAPTMEMBENCH* FRAMEWORK

The AdaptMemBench framework is designed with flexibility and consistency to mimic memory access patterns from applications. This framework accommodates code extracted from applications and facilitates memory performance measurements by providing a space to experiment with the execution order, data storage layout, and parallelization strategies. AdaptMemBench provides a starting point to diagnose the performance bottlenecks, examine potential optimizations to overcome those bottlenecks, and explore the potential gains of those optimizations. The framework assists in keeping track of how small changes in application characteristics impact the achieved performance of different code variants that arise during the exploration and experimentation phase of a project.

AdaptMemBench provides a unified framework with the following capabilities:

- The framework benchmarks multithreaded kernels on shared memory systems using the OpenMP [41] parallel programming model.
- AdaptMemBench measures performance in terms of execution time, memory bandwidth and floating point operations per second. It is also capable of recording the execution time of each core under execution.
- It offers low overhead access to hardware performance counters using the low level Performance API (PAPI) [40].

- AdaptMemBench provides a flexible testbed to explore potential code optimizations using polyhedral code generation.
- The framework enables visualization of performance results using a late analysis tool that plots 2D and 3D graphs from the output metadata.

The AdaptMemBench framework separates the user interface, validation, and output of the benchmark from the code being measured. Each computational kernel of interest is coded in a pattern specification. If that pattern specification optionally involves the polyhedral model, it is passed through a polyhedral compiler. The resulting (or original) c code is compiled together with one of several potential templates. The templates provide a uniform interface and handle code to vary the working set size to cover each portion of the memory hierarchy, along with timing, PAPI data collection, and output formatting. The use of the polyhedral model adds a great deal of flexibility in terms of exploring optimizations.

3.1 Design of the Framework

AdaptMemBench accepts a set of pattern specifications and a benchmark driver template as input. The framework uses a collection of generic benchmark driver templates for all variations of the access patterns. These driver templates provide a standard command line interface and output machine parsable and human readable metadata. The pattern specifications consist of a header file and three code files that specify the execution schedule, initialization steps, and validation conditions. The input to the framework is marked green in the workflow diagram (figure 3.1).

AdaptMemBench supports any memory access pattern - patterns involving arbitrary random reads and writes, non-unit strides, indirect memory accesses, and so

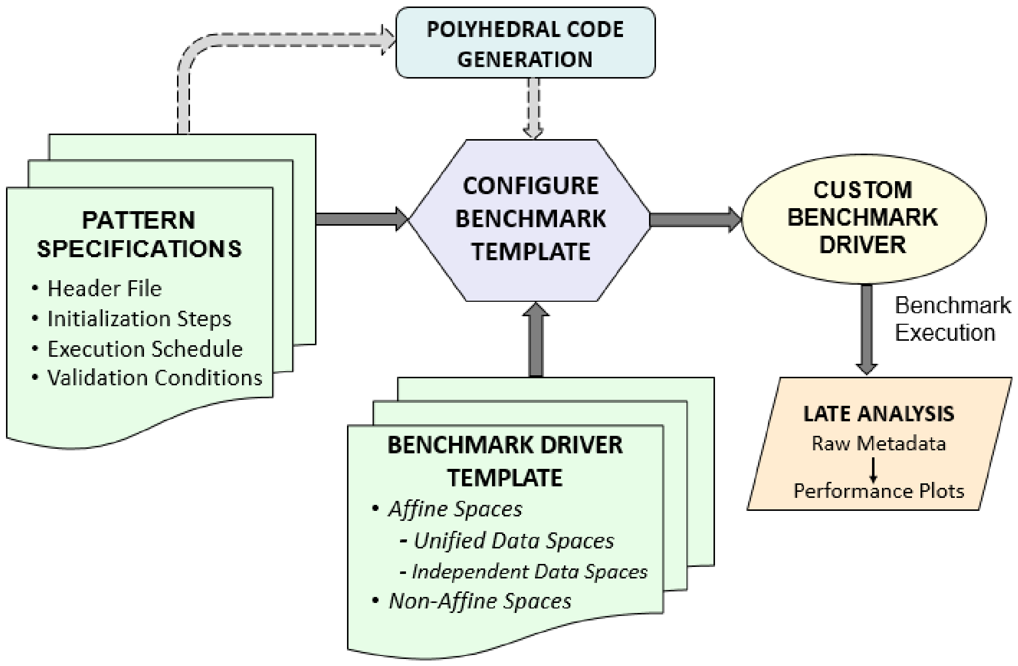


Figure 3.1: The overall workflow of the proposed framework.

on. The kernel to be measured is specified either using C or using the polyhedral model. When utilized, the polyhedral model offers the ability to compose sequences of transformations for the input access pattern. AdaptMemBench leverages the ISCC and the Omega+ code generation tools. The former is used for access patterns that can be represented with affine spaces, and the latter for non-affine constraints. The polyhedral code generation could be turned off for the access patterns that cannot be represented using the polyhedral model and/or for simplistic kernels.

The custom benchmark driver is created by configuring the user-chosen driver template with files from the pattern specifications. If polyhedral code generation is enabled, the code files specifying the execution schedule, initialization steps and the validation conditions are input as ISCC script (`.in`) files. These three script files are passed through the polyhedral compiler (either ISCC or Omega+, depending on the template chosen) and the resultant C code files generated are used to customize the template along with the header file (figure 3.2). If polyhedral code generation is disabled, the three code files are accepted as actual C code files bypassing the polyhedral compiler, and the customized benchmark driver is generated.

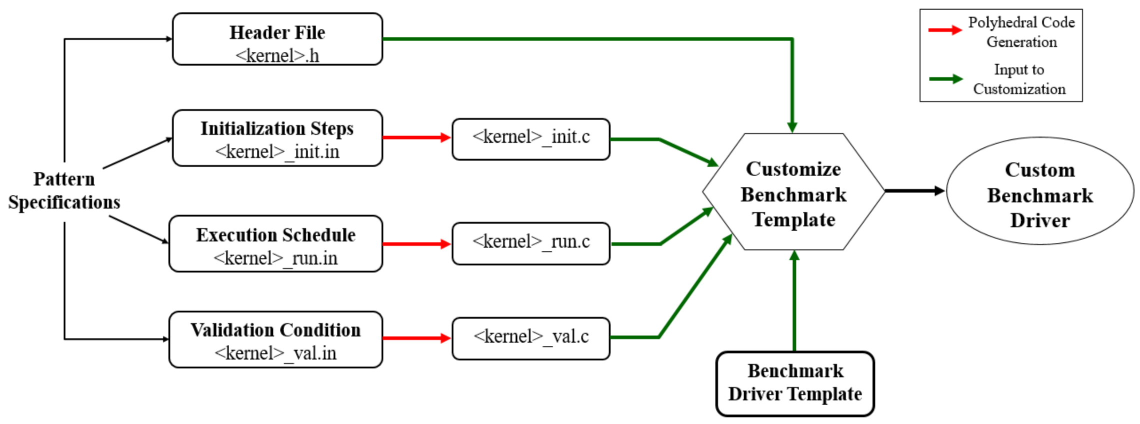


Figure 3.2: The process of generating the three C code files when polyhedral code generation is enabled.

The generated driver is then compiled and executed to benchmark the kernel specified in the pattern specification. The executable accepts runtime arguments such as the working set size, thread count, the number of repetitions, and other parameters depending on the access pattern benchmarked. The performance results are output in terms of execution time, cumulative memory bandwidth, and floating point operations per second (FLOP/s). The framework also offers an option to access the hardware counters using PAPI. In this case, the timers are replaced with the PAPI events in order to avoid the minimal overhead of running the timers while monitoring the hardware counters.

The output metadata can optionally be visualized graphically using the late analysis tool of AdaptMemBench. This tool transforms the raw data that consists of the working set size, execution time, memory bandwidth and other metrics into two-dimensional and three-dimensional graph plots. The late analysis tool leverages the *matplotlib* python library [24] to plot graphs.

3.2 Framework Components

This subsection elucidates the building blocks of the proposed framework - the benchmark driver templates and the pattern specifications.

3.2.1 Benchmark Driver Templates

The AdaptMemBench framework provides three benchmark driver templates. The unified and independent data space templates for affine spaces, and the non-affine spaces template suited for indirect memory access patterns. Each of the templates is described below with examples.

1. Templates for *Affine Spaces*:

   This set of templates is suited for access patterns that can be represented in affine space, i.e., for regular applications that consist of constant loop bounds or conditions at runtime. They are intended for static analysis that can be performed at compile time. The ISCC polyhedral code generation tool can be used to generate loop constructs for execution schedule, initialization steps, and the validation conditions to be customized in these templates. The polyhedral compilation could optionally be disabled to feed in the original C code files from the user into these templates.

   - *Unified Data Spaces* Template:

     This template is used for data structures that are shared among threads in shared memory applications. This form of representation adds overhead due to additional communication between threads. Listing 3.1 below shows the core components of this template.

```
...
//Include Header file
#include "<kernel>.h"
...

//Include Initialization Steps
#pragma omp parallel for
    #include "<kernel>_init.c"
...
/* start the timer */
//Include the execution schedule
#pragma omp parallel
{
    #pragma omp parallel for OMP_CLAUSE
    for (int k = 0; k < num_repetitions; k++) {
        #include "<kernel>_run.c"
    }
    <kernel>_postrun()
}
/* stop the timer */
...
// Validate results
if(!verify_code(ntimes, measurements, A, B)){
    return;
}
 /* Calculate execution time, memory bandwidth and GFLOPS/s */
...
```

Listing 3.1: The unified data spaces template.

- *Independent Data Spaces* Template:

  The data structures in this template are independently allocated for the threads. This implementation eliminates cross-thread communication and improves performance at the higher memory hierarchy levels. For this template, the constructs for OpenMP parallelization are reorganized from the unified data spaces template to accommodate distinct data structures for each thread under execution.

```
...
//Include Header file
#include "<kernel>.h"
...

//Include Initialization Steps
#pragma omp parallel
{
    #include "<kernel>_init.c"
}
...
/* start the timer */
//Include the execution schedule
#pragma omp parallel
{
    int t_id = omp_get_thread_num();
    for (int k = 0; k < num_repetitions; k++) {
        #include "<kernel>_run.c"
    }
    <kernel>_postrun()
}
/* stop the timer */
...
// Validate results
if(!verify_code(ntimes, measurements, A, B)){
    return;
}
 /* Calculate execution time, memory bandwidth and GFLOPS/s */
...
```

Listing 3.2: The independent data spaces template.

2. Template for *Non-affine Spaces*:

   This generic template supports non-affine constraints, i.e., the loop bounds that are data-dependent or based on the output of the functions that are unknown at compile time, as in sparse matrix index structures. The execution schedule for this template is generated using the uninterpreted functions supported by the Omega+ library [11]. The initialization steps and the validation conditions are accepted as original C code files for brevity. The execution schedule could also be input as original C code, bypassing the polyhedral code generator. The parallelization constructs are not part of the template and the input pattern specification files must include the appropriate OpenMP constructs.

```c
...
//Include Header file
#include "<kernel>.h"
...
//Include Initialization Steps
#include "<kernel>_init.c"
...
...
/* start the timer */
//Include the execution schedule
#include "<kernel>_run.c"
/* stop the timer */
...
...
// Validate results
if(!verify_code(ntimes, measurements, A, B)){
    return;
}
 /* Calculate execution time, memory bandwidth and GFLOPS/s */
...
```

Listing 3.3: The template for non-affine spaces.

3.2.2 Pattern Specifications

The input pattern specification describes the execution order and the statement instances separated into concerns. It provides information on memory allocation, data mapping, and validation conditions. The purpose and functionality of each component in the pattern specifications shown in figure 3.2 are described below:

1. **Header file** (`<kernel>.h`):

   This file contains the definitions of the memory mappings, statement macros, and the allocation code.

- *Memory Mapping*: Indicates how the statements should map into memory using iterators as input.
- *Statement Macros*: The definition of the statement macros substituted in each of the C code files generated from the `.in/.c` input files. Any data referred to within the statement should be referred to indirectly through the data mapping.
- *Allocation Code*: Specifies memory allocation of the data spaces used in the given application kernel.

  An example header file for a stencil 2D heat solving equation is shown below in listing 3.4.

```c
//Allocation Code
#define Stencil2D_alloc double* A = double *) malloc(sizeof(double) * n * n); \
                        double* B = double *) malloc(sizeof(double) * n * n); \

//Memory Mapping
#define A_map(i,j) A[i*n + j]
#define B_map(i,j) B[i*n + j]

//Initialization
#define Stencil2D_init(i) A_map(i) = i*1.12; B_map(i) = i*1.12;

//Statement Definition
#define Stencil2D_run(i,j) A_map(i,j) = (B_map(i-1,j) + B_map(i+1,j) + B_map(i,j) + B_map(i,j-1) + B_map(i,j+1)) * 0.2;

//Validation Condition
#define Stencil2D_val(j) flag_err = abs((A_map(i,j) - ((i-1)*n+j)*1.12 + ((i+1)*n+j)*1.12 + ((i*n+j)*1.12) + ((i*n+j-1)*1.12) + ((i*n+j+1)*1.12) ) *0.2 ) <= EPS && !flag_err ? 1 : flag_err;

//OpenMP clause
#define CLAUSE schedule(static)
```

Listing 3.4: Header file `<Stencil_2D.h>` for the 5-pt stencil 2D benchmark.

2. **Initialization steps** (`<kernel>_init.in`/`<kernel>_init.c`):

   This input file specifies the schedule for which the data domains allocated in the header file are initialized. In the C code file generated with ISCC, the associated statement macro specifying initialization steps is substituted when the benchmark is executed.

```
Domain_init := [n] -> {
    Stencil2D_init[j,i] :
   0<=i<n and 0<=j<n; };
codegen (Domain_init);
```

```
for (int c0=0; c0<n; c0+=1)
  for (int c1=0; c1<n; c1+=1)
    Stencil2D_init(c0, c1);
```

Listing 3.5: The files `Stencil2D_init.in` and `Stencil2D_init.c` are on the left and the right respectively.

3. **Execution Schedule** (`<kernel>_run.in`/`<kernel>_run.c`):

   An input file that defines the iteration space in which the access pattern is executed. The application kernel defined as a macro in the header file is replaced in the `.c` file generated. This code file consists of the for loop constructs associated with the execution domain which will be substituted in the driver when executed.

```
Domain_run := [n] -> {
    Stencil2D_run[j,i] : 1<=
   i<=n-2 and 1<=j<=n-2; };
codegen (Domain_run);
```

```
for (int c0=1; c0<=n-2; c0+=1)
  for (int c1=1; c1<=n-2; c1+=1)
    Stencil2D_run(c0, c1);
```

Listing 3.6: The code on the left and right are the files Stencil2D-run.in and Stencil2D-run.c respectively.

4. **Validation condition** (`<kernel>_val.in`/`<kernel>_val.c`):

This input file describes the schedule for which the results after executing the kernel is validated. The corresponding C code file generated is then called in the header file to validate the results.

```
Domain_val := [n] -> {
    Stencil2D_val[j,i] :
   0<=i<n and 0<=j<n; };
codegen (Domain_val);
```

```
for (int c0=0; c0<n; c0+=1)
  for (int c1=0; c1<n; c1+=1)
    Stencil2D_val(c0, c1);
```

Listing 3.7: The files `Stencil2D_init.in` and `Stencil2D_init.c` are on the left and the right respectively.

3.3 Timing the Benchmark

The proposed framework enables performance measurement across all the levels of the memory hierarchy. AdaptMemBench implements a tight loop (listing 3.8) that records a single cumulative execution time across all the time steps and then calculates the average run time for the kernel benchmarked. This makes computation of memory bandwidth possible for data sizes in the L1 and L2 caches. This approach of setting up the timers for the benchmarks overcomes the dependence on the system timer granularity.

Cold start cache miss is not an issue with our timing approach since the data spaces are initialized right before benchmarking the execution schedule. This means that the execution time of the first time iteration while benchmarking need not be discarded. This approach, however, does suffer from the drawback that we are unable to measure the execution time of each pass through the data and observe variations.

AdaptMemBench also measures the execution time for each thread under execution. The framework leverages the `omp_get_wtime()` method from the OpenMP library [41] to measure the start time and the end time of the execution. There are other tools such as Intel VTune [43] and Likwid [59] that could be used to measure each thread's execution time. The OpenMP timers are preferred for lower overhead and for accommodation in the driver templates.

```
....
//Start measuring execution time for num_reps iterations
init_time = get_time();
for ( int k = 0; k < num_reps; k++) {
    for (int i = 0; i < n; i++) {
        #include "<kernel>_run.c"
    }
}
//Stop time measurement
exec_time = get_time() - init_time;
//Calculate average execution time for a single time iteration
avg_exec_time = exec_time/num_reps;
...
```

Listing 3.8: Timing the benchmark in AdaptMemBench

3.4 Accessing Performance Counters

Hardware performance counters are special-purpose registers built into modern microprocessors that store critical information of hardware events. Hardware events are transpirations of particular signals akin to the function of a processor. Hardware counters provide information on events such as hit/miss rate at each memory level, cycle count, instruction count, branch prediction accuracy, TLB invalidations, and pipeline stalls. Performance experts and hardware engineers rely on these counters for hardware verification or debugging, and for low-level performance monitoring, analysis or tuning.

Information on accessing these counters is limited and it is quite challenging to record hardware related events. The Performance Application Programming Interface (PAPI) [56, 40] provides machine and operating system independent access to performance counters in most modern processors. It offers a consistent interface to interpret the relation between software performance and processor events. PAPI provides access to a collection of components that offers performance monitoring opportunities across the hardware and software stack.

PAPI offers two interfaces to access hardware events: the *high-level* and *low-level* interface. The high-level interface provides users simple and straightforward start, stop, and read routines which provides access to specified event information from hardware counters. This doesn't require code to be rewritten for different architectures and is useful for instrumenting end-user applications. The low-level interface provides an advanced interface for all applications and tools. It is fully programmable, thread-safe, and allows user to define their own event sets. In this thesis, we leverage the low-level interface due to its efficiency from lower overhead and its flexible functionality.

Other than PAPI, there exist other performance counter tools. The likwid-perfctr [61] supported by likwid Marker API [59], provides self-monitoring of an application. Compared to PAPI, it has to specify the events to be monitored on the command-line rather than in user's code. Furthermore, likwid-perfctr causes much higher overhead than PAPI since it requires context switch into the kernel each time it reads or writes to counters. The LiMiT tool [1] provides fast, userspace access to performance counters compared to PAPI. It is however quite complicated to use and is unstable due to frequent system crashes [59], and hence it is commonly not preferred. We thus incorporate AdaptMemBench with PAPI to monitor the performance counters instead the other existing tools.

Hardware counters are most commonly monitored by performance engineers to interpret the interaction of the software with the hardware and advise optimizations. Access to PAPI in the benchmarking framework gives opportunities even for novice software developers to understand the relation between application performance and processor events. This offers them an interface in guiding them with performance enhancement strategies at the node level.

In the AdaptMemBench framework, monitoring the hardware counters is optional. If enabled, the timers are replaced by the PAPI events to avoid the overhead added by the timing events while accessing the hardware counters. Listing 3.9 shows the core of the modified independent data spaces template with PAPI enabled.

```
...
//Include Header file
#include "<kernel>.h"
...

//Include Initialization Steps
#pragma omp parallel
{
    #include "<kernel>_init.c"
}
...
```

```

//PAPI events
int events[3] = {PAPI_L1_DCM, PAPI_CA_SHR, PAPI_CA_CLN };

//Start the PAPI events
 PAPI_start_counters(events, 3);

//Include the execution schedule
#pragma omp parallel
{
    int t_id = omp_get_thread_num();
    for (int k = 0; k < num_repetitions; k++) {
        #include "<kernel>_run.c"
    }
    <kernel>_postrun()
}
//Stop the PAPI events
PAPI_stop_counters(values, 3);
...
```

Listing 3.9: The modified independent data spaces template with PAPI enabled.

Chapter 4

CASE STUDIES

A series of case studies were performed to demonstrate the capabilities of AdaptMemBench. The case studies extend the capabilities of existing memory benchmark suites and include computer kernels that are commonly used in performance optimization studies. Existing benchmark suites were used as a baseline to establish the validity of the framework's results. Compute kernels are pieces of code that capture characteristics common in scientific applications. They are commonly used to showcase the impact of new optimization strategies that are difficult to reproduce. This chapter describes a set of case studies on three compute kernels that include:

1. Streaming Triad Kernel
    - An evaluation of the overhead associated with implicit barriers and shared data spaces.
    - A demonstration of interleaved scheduling that leverages prefetching to enhance the performance of triad.
2. Multidimensional Jacobi Patterns
    - An evaluation of the impact of false sharing on achieved performance.
    - A study on the efficacy of spatial and temporal tiling strategies for higher dimensional Jacobi kernels.

3. Sparse Matrix Computations

    - A comparison of different sparse matrix representations for Sparse Matrix-Vector (SpMV) multiplication with real-world matrices of varying sparsity structures.

Experimental Setup

Hardware: Experiments were run on one of the nodes in the R2 HPC cluster at Boise State University. R2 has dual 2.40GHz Intel Xeon E5-2680 v4 CPUs with Intel Turbo boost turned on. Each node consists of two NUMA domains each containing 14 cores. Each core has a dedicated 32K L1 data cache and 256K L2 cache. The 35 MB L3 cache is shared among all the cores in each NUMA domain. The size of each cache line in this architecture is 64 bytes.

Compilers: GNU's gcc (version 7.2) and Intel's icc (version 18.0.1) compilers were used to compare the memory performance results on executing the triad kernel. Since the execution on gcc compiler yielded higher and consistent results, all the other benchmarks are compiled with the gcc compiler.

Compilers Options: When building C++ benchmark drivers, `-fopenmp` and `-O3` optimization flags were set. The `-lpapi` flag was set for PAPI-enabled benchmark drivers. The gcc compiler option `-ftree-vectorizer-verbose=[n]` is used to obtain vectorization reports, with `n` ranging from 0 (no information reported) to 6 (all information reported).

Profiling Tool: The benchmark drivers are instrumented with the Performance API (PAPI) [40] library to access performance counters across the CPUs evaluated. PAPI is used to measure cache hits and the requests for exclusive access to cache lines.

Problem size: We executed the benchmarks with problem sizes across all levels of cache and those which exceeded the last-level cache and fit into the main memory. Table 4.1 shows the range of the working sets executed for the streaming and stencil benchmarks. Each benchmark is executed for 1000 time iterations. The number of repetitions is configurable.

Table 4.1: Range of working sets tested for all streaming and stencil benchmarks.

Memory Unit	Initial working set size	Final working set size	Stride
L1 Cache	5 KiB	900 KiB	10 KiB
L2 Cache	900 KiB	10 MiB	256 KiB
L3 Cache	10 MiB	80 MiB	512 MiB
Main memory	80 MiB	180 MiB	1 MiB

4.1 Streaming Triad Kernel

The simplest case of a loop kernel reads data sequentially from an array and writes updated values with no temporal locality. *Triad* is one such streaming kernel that involves three data spaces, as in listing 4.1, where array `a` is updated by multiplying a scalar `c` with array `d` and then adding the resultant to array `b`. It often achieves optimal performance since (i) this pattern accesses three prefetching lanes simultaneously and (ii) the operation utilizes the Fused Multiply Add (FMA or fmadd) unit.

This simplistic access pattern is quite common and its performance is well understood with the STREAM benchmark [38]. In this work, the benchmarked results of triad from AdaptMemBench is used for validation against STREAM. This basic pattern is used to explore the impact of implicit locks and shared data spaces.

```
for (int i = 0; i < n; i++){
   A[i] = B[i] + scalar * C[i];
}
```

Listing 4.1: Streaming Triad code.

4.1.1 Performance Validation with the STREAM benchmark

STREAM [38] is considered the de facto benchmark for streaming kernels that measures main memory bandwidth. However, the way the timers are set up in STREAM for benchmarking makes it impossible to measure performance at higher levels of the memory subsystem.

STREAM executes each kernel for `ntimes` time steps and records the execution time for each of the time iteration. The best bandwidth among `ntimes` is then computed with the minimum execution time across all time steps. Though this method yields reliable results for large working set sizes in the main memory, it is impossible to calculate the bandwidth for smaller data sizes in the higher cache levels, mainly due to dependence on the granularity of the system timer.

The timing approach in AdaptMemBench, as elucidated in chapter 3.3 enables performance measurement at all levels of the memory hierarchy. To ensure correctness of our timing approach, the memory bandwidth results from AdaptMemBench are validated by comparing them with the measurements STREAM. The streaming triad kernel is used for the comparison.

The triad benchmark driver is generated from AdaptMemBench using a combination of input C code files. Code from Listings 4.2 and 4.3 are customized with the *unified data spaces* template in Listing 3.1 to create a custom benchmark for the triad kernel, bypassing the polyhedral compiler for simplicity. Alternatively, the kernel could have been expressed as a set: $\{[j] | 0 <= j < n\}$ for code generation using

```
//Allocation Code
#define Triad_alloc double* A = double *) malloc(sizeof(double) *
    n); \
                        double* B = double *) malloc(sizeof(double) *
    n); \
                        double* C = double *) malloc(sizeof(double) *
    n);
//Memory Mapping
#define A_map(i) A[i]
#define B_map(i) B[i]
#define C_map(i) C[i]
//Initialization
#define Triad_init(i) A_map(i) = 1.0; B_map(i) = 3.0; C_map(i) =
    4.0;
//Statement Definition
#define Triad_run(i) A_map(i) = B_map(i) + scalar * C_map(i);
//OpenMP clause
#define CLAUSE schedule(static)
```

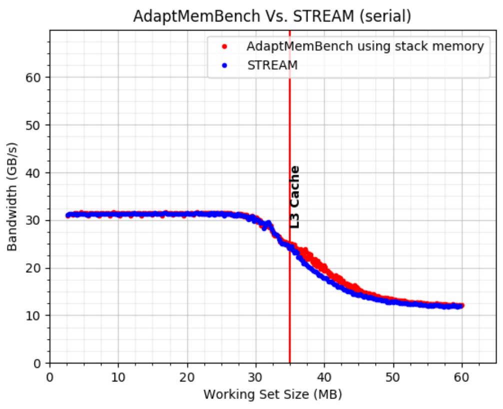
Listing 4.2: Header file `<triad.h>` for the triad benchmark.

```
for (int j = 0; j < n; j++){
        Triad_run(j);
}
```

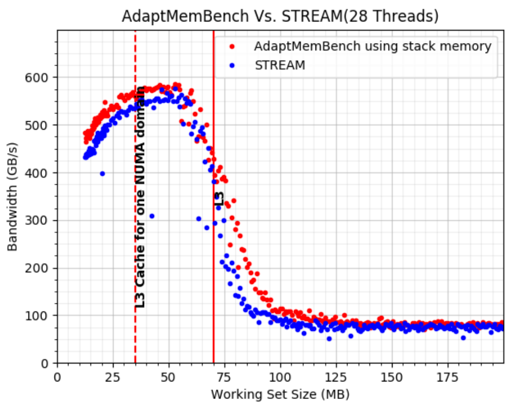
Listing 4.3: The execution schedule of the benchmark driver generated by combining the input file `<triad_run.c>` and the template.

the polyhedral model. The results are equivalent.

The AdaptMemBench triad driver and the STREAM benchmark are both compiled with g++ and gcc compilers respectively. In figure 4.1a, we observe a near-to-perfect match of bandwidth values between AdaptMemBench and STREAM for serial execution, with a minor deviation during the transition from L3 cache to main memory. However, in case of parallel execution with 28 threads (figure 4.1b), our benchmark achieves higher bandwidth than STREAM in most working set sizes. There are two primary differences between the two benchmarks that cause this. First, the use of the C++ compiler rather than the C compiler in AdaptMemBench, and second, the use of dynamic memory allocation in AdaptMemBench.



(a) Serial Execution



(b) Parallel Execution with 28 Threads

Figure 4.1: Validating AdaptMemBench with STREAM for serial and parallel execution.

4.1.2 Cost of Barriers in OpenMP

We use the triad benchmark generated by AdaptMemBench to evaluate the overhead associated with barriers in OpenMP. Triad is specifically chosen due to its simplicity and minimal dependencies associated with the pattern. A barrier is a synchronization mechanism in which, at a point of execution in a parallel program, all active threads wait for each other until all the threads in the team arrive at that point (figure 4.2). Many directive-based parallel programming models such as OpenMP impose implicit barriers to avoid race conditions. Implicit barriers when induced at a part of a program that involves threads that are independent of each other, causes unnecessary performance bottlenecks.

In OpenMP, barriers could be explicitly avoided by adding a `nowait` clause to the `pragma omp for` construct in the inner loop of the triad benchmark. With the AdaptMemBench framework, creation of such a benchmark driver is simplified by just modifying the definition of the macro `CLAUSE` in the header `triad.h` to be `nowait`.

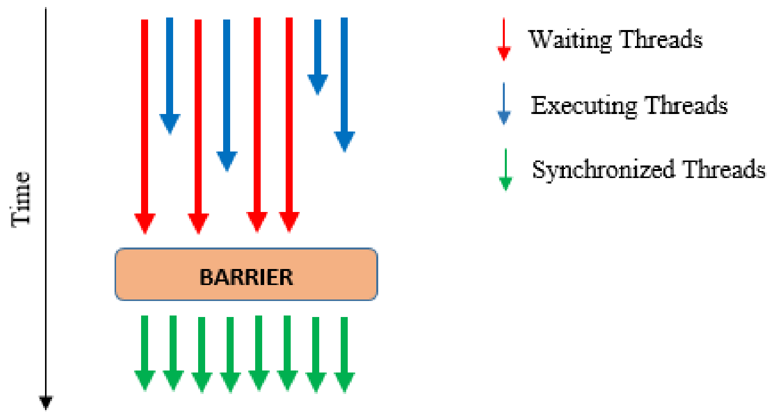


Figure 4.2: Illustration of barriers in multithreaded programming.

As memory bandwidth results in Figure 4.3 indicate, there is a significant overhead caused by the barrier, and by breaking the barrier using the `nowait` clause, we are able to achieve a reasonable speedup. Though this modification may not be possible for all computations, e.g. those that have loop carried dependencies, our intention is to demonstrate the performance degradation caused by implicit locks using the simplistic triad kernel.

4.1.3 Overhead of Shared Data Spaces

The shape of the curve in the performance results on the triad benchmark is disconcerting. Specifically, bandwidth in L1 is less than that in L2. There is a significant amount of overhead to utilize shared memory parallel applications. We explore the performance bottleneck caused by cross-thread communication with two variants of the triad benchmark: unified data spaces and independent data spaces.

The first variant is implemented with unified data spaces using OpenMP's work sharing constructs. Listing 4.4 is a part of the benchmark driver generated from the *unified data spaces* template with the macro `OMP_CLAUSE` in `triad.h` set to

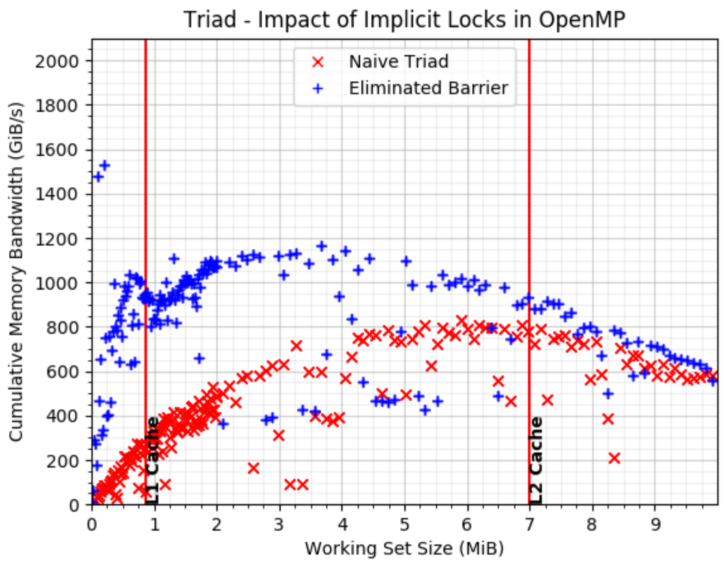


Figure 4.3: The impact of OpenMP barriers on achieved memory bandwidth.

`schedule(static, n/t) nowait`.

```
for(int k = 0; k < ntimes; k++) {
    #pragma omp parallel for schedule(static, n/t) nowait
    for (int i = 0; i < n; i++){
        A[i] = B[i] + scalar * C[i];
    }
}
```

Listing 4.4: Utilizing the OpenMP work sharing construct for data spaces of size `n` and `t` number of threads.

The second benchmark uses the *independent data spaces* template from Listing 3.2 implemented with distinct data spaces independent of the threads. The data mapping for the data structures in the header file `triad.h` are modified to be A[t_id], B[t_id] and C[t_id]. The listing 4.5 shows the resulting execution schedule after macro expansion in the generated driver.

Memory bandwidth results in Figure 4.4 clearly indicate the benefit of using distinct data spaces over the shared data spaces variant implemented using OpenMP

```
int N = n/t;
#pragma omp parallel
{
  int t_id = omp_get_thread_num();
  for(int k = 0; k < ntimes; k++) {
    for (int i = 0; i < N; i++){
      A[t_id][i] = B[t_id][i] + scalar * C[t_id][i];
    }
  }
}
```

Listing 4.5: The resultant triad benchmark using the *independent data spaces* driver template

work-sharing and scheduling constructs. Using independent data spaces separates data domains into separate memory regions, eliminating cross-thread communication. This in turn eliminates performance bottlenecks, for example, avoiding multiple threads accessing the same cache line. We observe an approximate two-fold performance boost in the L1 cache with this approach compared to unified data spaces using OpenMP work-sharing constructs, which is deemed to be efficient.

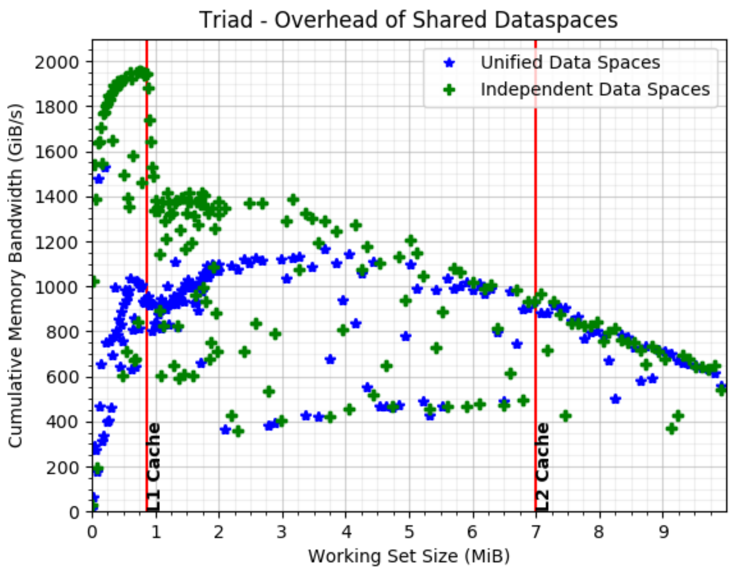


Figure 4.4: Illustrating the overhead associated with data shared among threads.

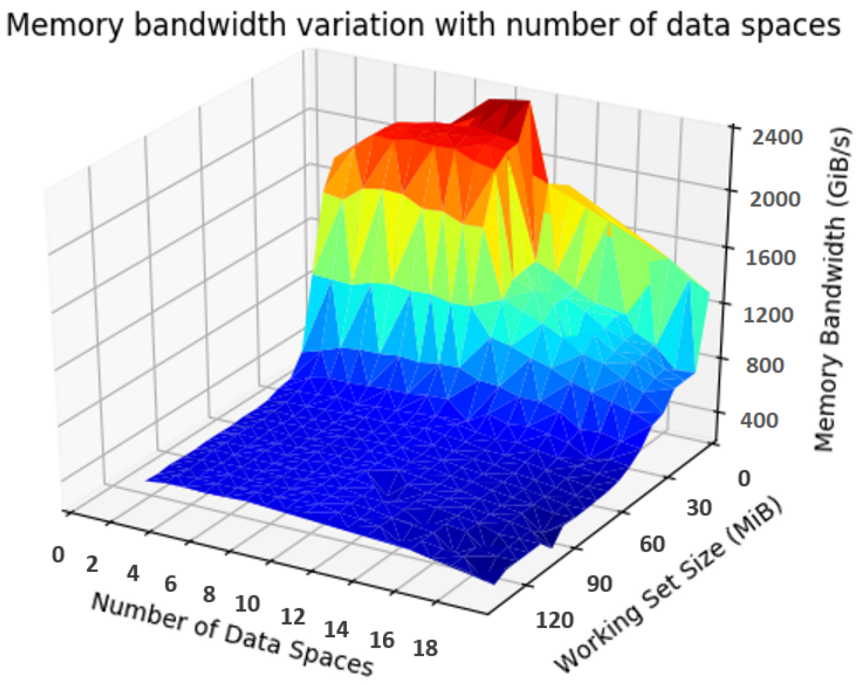


Figure 4.5: An experiment to identify the number of data streams fetching simultaneously that gives optimal performance on parallel execution with 28 threads.

```
for (int i = 0; i < n/2; i++){
    A[i] = B[i] + scalar * C[i];
    A[i+n/2] = B[i+n/2] + scalar * C[i+n/2];
}
```

Listing 4.6: Customized benchmark driver with unified spaces illustrating interleaved optimization for triad

4.1.4 Interleaved Scheduling

The triad pattern that comprises three data spaces is often considered to yield optimal performance in a given architecture. With the configurability offered by our benchmarking framework, we expand the number of data spaces evaluated from 3 (in triad) to 20 data streams that are simultaneously read in the body of the loop. This is

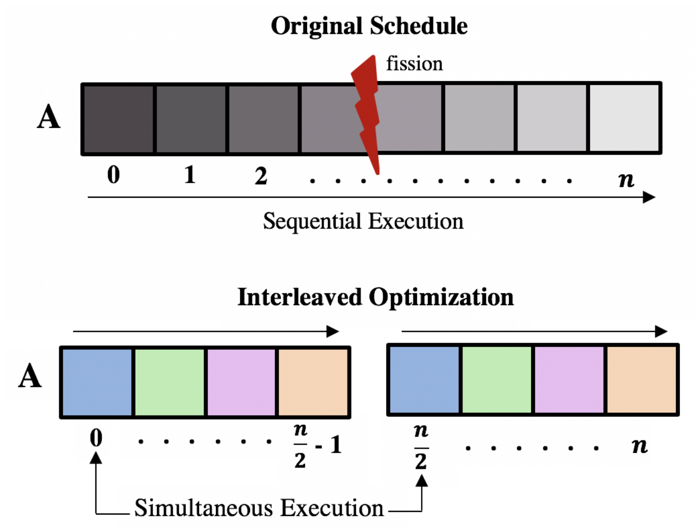


Figure 4.6: Illustration of interleaved optimization with a single data space of size n.

achieved by modifying the statement definition and memory allocation specifications in the header file. The framework generates each of the drivers by accepting the triad pattern and the number of data streams triad has to be expanded to, without requiring the user to manually manipulate pattern specifications for each driver.

Figure 4.5 shows the results of running this experiment in parallel with 28 threads. The memory bandwidth values are inconsistent for working set sizes that sit in L1 cache since small data sets are shared among a large number of threads. Considering working sets in L2 cache, where the performance is more consistent, we observe that the achieved memory bandwidth peaks for 11 data spaces, which is considerably higher when compared to triad that comprises 3 data streams. Not all access patterns may require access to such number of data spaces. We thus reschedule triad to represent the kernel to utilize more number of data streams and improve its overall performance by better exploiting the benefit of prefetching.

Listing 4.6 describes the interleaved execution schedule implemented for triad.

This schedule splits each data spaces of size n into two independent data blocks of size $\frac{n}{2}$ each. Each of these blocks are executed simultaneously within a single iteration by fusing their execution. This reduces the number of iterations to half of the total, which means, the data elements at $[\texttt{0}\ldots\ldots\frac{n}{2}-1]$ and $[\frac{n}{2}\ldots\ldots\texttt{n}]$ are accessed in $\frac{n}{2}$ iterations.

Figure 4.6 illustrates how a single data space is interleaved into two blocks and fused together to be accessed simultaneously within a single iteration. In the case of triad, this implies that the three data spaces access six prefetching lanes simultaneously. This behavior is equivalent to the *hexad* operation which comprises six data spaces that accesses six prefetching lanes.

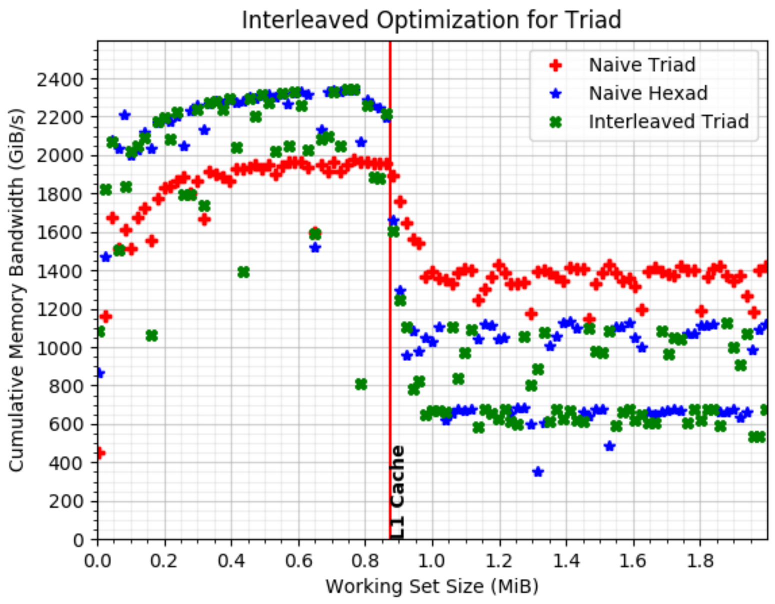


Figure 4.7: Interleaved optimization for triad is beneficial in L1 cache on parallel execution with 28 threads.

Performance results in Figure 4.7 illustrate the improvement in achieved bandwidth for triad in the L1 cache. A maximum speedup of 22% is observed from the naïve triad operation implemented with independent data spaces. For working set sizes outside of the L1 cache, this optimization is ineffective, and hence the working set size is a crucial factor when applying the interleaved schedule. The memory achieved memory bandwidth is quite similar for the interleaved triad (3 data spaces) and the naïve hexad (6 data spaces), both of which access 6 prefetching lanes. We attempted interleaving data spaces for triad with interleaving factors greater than 2, but the resulting execution schedules do not perform better.

4.2 One-dimensional Jacobi Kernel

This section provides background on stencil computations and then an evaluation on the impact of false sharing for the 3-pt Jacobi 1D kernel.

4.2.1 Stencil Computations

Stencil computations are algorithmic patterns at the core of a wide range of scientific applications. They are represented under the Structures Grid motif [4], one of the seven motifs of high performance computing. In a typical stencil computation, each element of a multidimensional grid is (iteratively) updated by performing a *stencil operation* to a neighborhood of its elements. This applies a *stencil function*, which describes the choice of neighboring elements accessed to perform the corresponding operation.

Stencil patterns with lower arithmetic intensity are mostly memory bandwidth-limited and data locality in caches impact achieved performance. Optimizing stencils to maximize performance has been researched for many years. The case study on stencil patterns using AdaptMemBench showcase an optimization space exploration.

Incorporation of the polyhedral model into the proposed framework offers a flexible proving ground for code transformations.

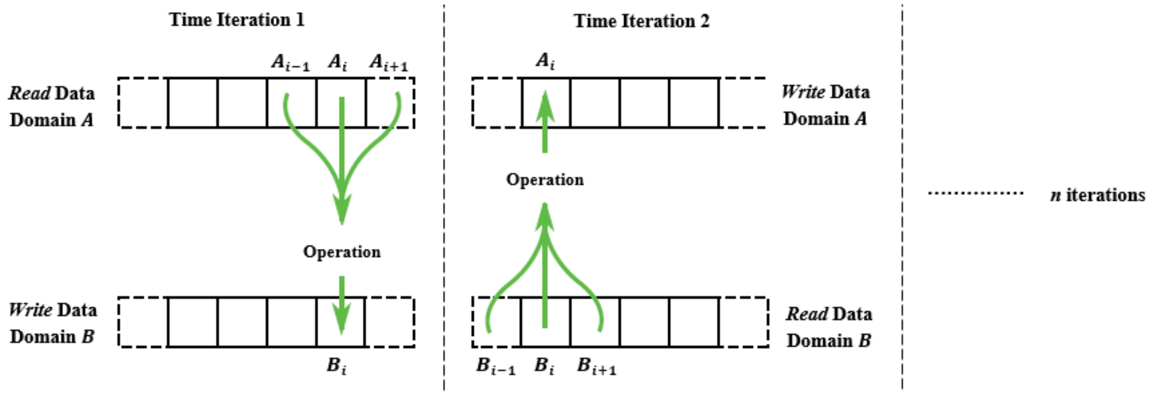
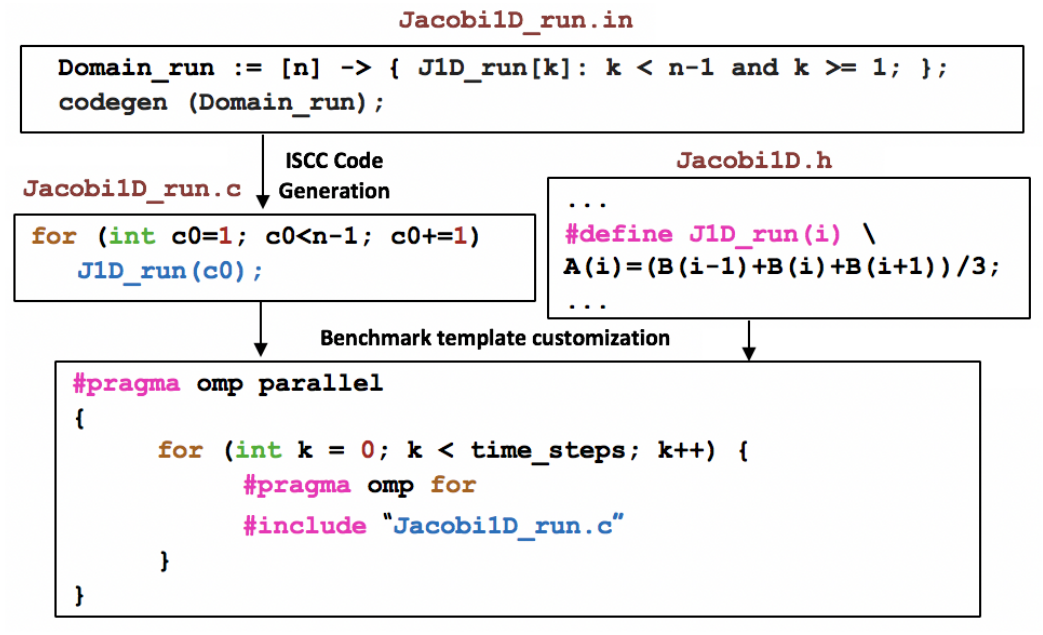


Figure 4.8: An illustration of 3-point 1D Jacobi iterative stencil.

In this thesis, we consider *Jacobi* iterative stencil loops, where the result of updating the grid is stored in a secondary grid instead of overwriting the values of the input grid. We implement 3-point 1D, 9-point 2D and 7-point 3D Jacobi operations in this work. For these patterns, we study the impact of false sharing and the efficacy of existing tiling optimizations.

4.2.2 Impact of False Sharing

Figure 4.9 demonstrates the process of custom benchmark generation for 3-pt Jacobi 1D using polyhedral code generation with the input pattern specifications using the *unified data spaces* benchmark template. Allocating independent spaces is advantageous for this pattern as well, as reflected by the memory bandwidth results in Figure 4.10. However, performance scaling in L1 is still an issue, due to false sharing.



```
Domain_run := [n] -> { J1D_run[k]: k < n-1 and k >= 1; };
codegen (Domain_run);
```

```
for (int c0=1; c0<n-1; c0+=1)
  J1D_run(c0);
```

```
...
#define J1D_run(i) \
A(i)=(B(i-1)+B(i)+B(i+1))/3;
...
```

```
#pragma omp parallel
{
      for (int k = 0; k < time_steps; k++) {
            #pragma omp for
            #include "Jacobi1D_run.c"
      }
}
```

Figure 4.9: Illustration of custom benchmark generation for 3-pt Jacobi 1D kernel with unified data spaces using the polyhedral model.

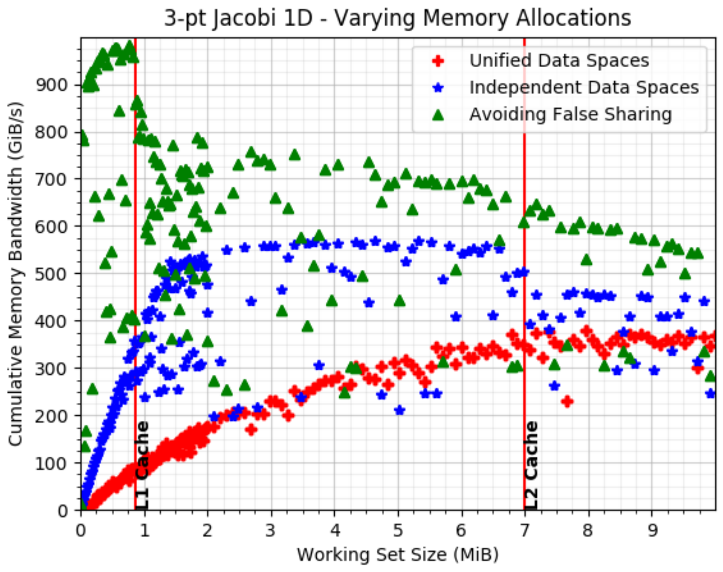


Figure 4.10: Demonstration of overhead associated with shared data spaces in SMP systems with Jacobi 1D.

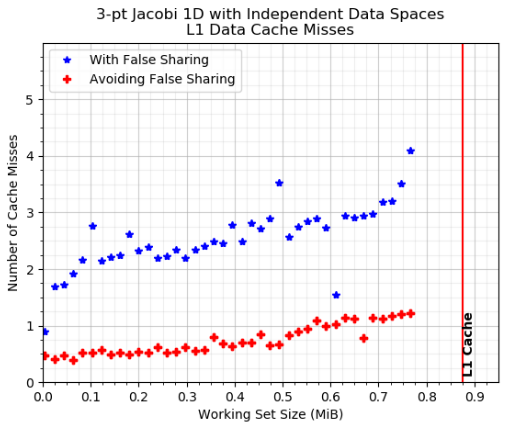
```
#pragma omp parallel
{
  int t_id = omp_get_thread_num();
  for(int k = 0; k < ntimes; k++) {
     for (int i = 1; i < n - 1; i++){
        A[t_id * 8][i] = (B[t_id * 8][i - 1] + B[t_id * 8][i] + B
    [t_id * 8][i + 1]) * 0.33;
     }
  }
}
```

Listing 4.7: The resultant independent data spaces benchmark driver reflecting array padding for Jacobi 1D

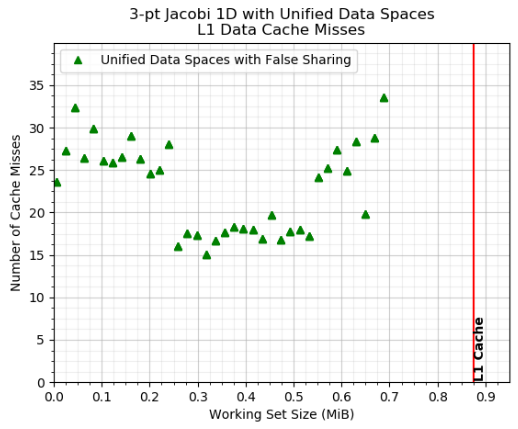
In symmetric multiprocessing systems, where each processor core has dedicated local cache(s), false sharing is a well-known performance issue. False sharing occurs when multiple threads involve in modifying independent variables sharing the same cache line, requiring unnecessary cache flushes and subsequent loads. The potential source of false sharing is multiple threads accessing dynamically allocated or global shared data structures simultaneously.

Padding arrays is a common solution to overcome false sharing. In the architecture evaluated, each cache line is of size 64 bytes. As shown in Listing 4.7, the data spaces of type double are padded with a factor 8 to allocate each element in different cache lines to avoid false sharing. With AdaptMemBench, this can be achieved just by modifying the memory mapping with the scaled padding factor for each data structure. Eliminating false sharing leads to a drastic performance speedup in the L1 cache, as the results in Figure 4.10 reflect.

The impact of false sharing is assessed by recording the performance counters using PAPI. We measure the data cache hits in L1 and the requests for exclusive access to shared cache lines in Figures 4.11 and 4.12. We observe that the shared data spaces get affected by cache misses nearly 10 times more than the independent



(a) Independent Data Spaces



(b) Number of requests to shared cache line

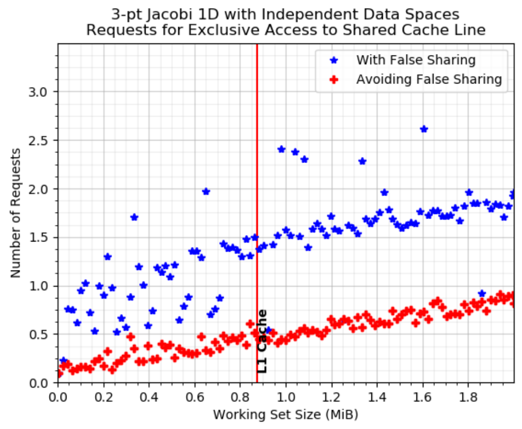
Figure 4.11: Number of L1 data cache misses for 3-pt Jacobi 1D.

data spaces (Figure 4.11). The cache misses recorded for independent data spaces is much lesser, but the variation in number of exclusive requests to clean cache line for the three cases in Figure 4.12 is much higher for L1 in the case that suffers from false sharing. The PAPI results were collected by running the same code configurations with a PAPI driver within the framework, and the memory bandwidth results are exclusive of the minimal overhead of accessing hardware counters.

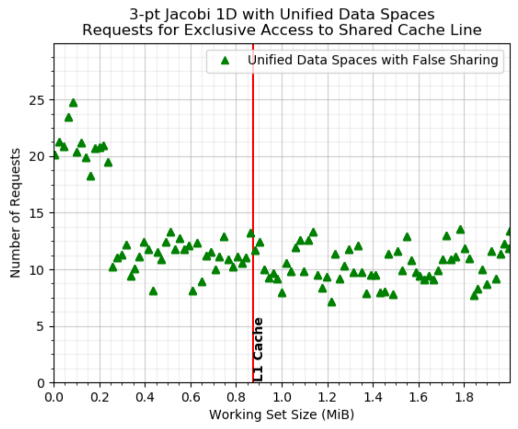
The source of false sharing needs to be identified and carefully avoided without hindering the effective use of the available caches. This experiment showcases the ability of AdaptMemBench to identify the performance bottlenecks in isolated hotspots of an application. This framework assists in analyzing the approaches of overcoming such bottlenecks, and evaluate the potential gains of that optimization.

4.3 Performance Comparison Across Compute Kernels

Application performance is a function of the memory footprint, instruction mix and order, memory access patterns, and achieved memory bandwidth. Forming



(a) Independent Data Spaces



(b) Number of requests to shared cache line

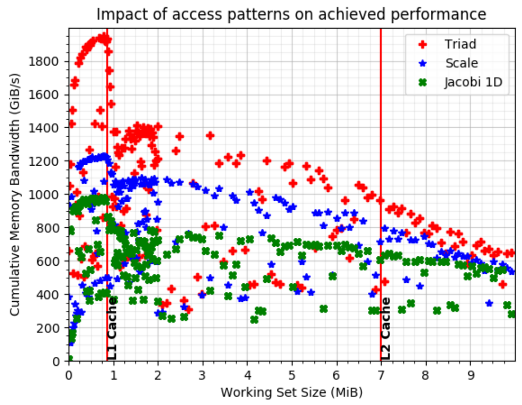
Figure 4.12: Number of requests for exclusive access to shared cache line for 3-pt Jacobi 1D.

reasonable expectations for application performance requires understanding these application characteristics, target resource behavior, and their interactions. In this section, we evaluate the impact on achieved performance due to small changes in the application characteristics.

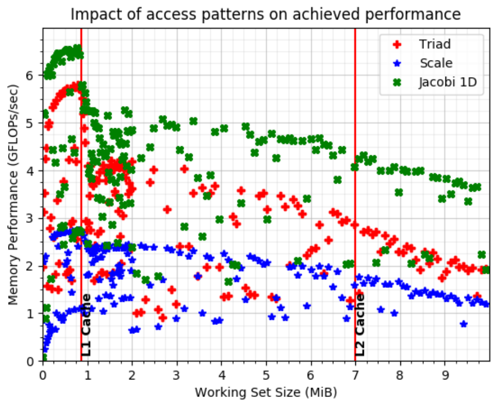
To understand this, we consider the following access patterns which are apparently similar and simple: triad, scale, and Jacobi 1D. Triad writes to a single data space in a streaming pattern while reading from two data spaces in the same pattern. It is commonly used for benchmarking systems. Scale and Jacobi 1D both write to a single dataspace in a streaming pattern and read from a single data space. The read pattern in Jacobi requires 3 neighboring values be in registers simultaneously, and performs two additions along with the multiply it has in common with scale.

- **Triad [38]:** `a[i] = b[i] + scalar * c[i]`
- **Scale [38]:** `a[i] = scalar * b[i]`

- **Jacobi 1D:** `a[i] = (b[i-1]+b[i]+b[i+1])*.33`



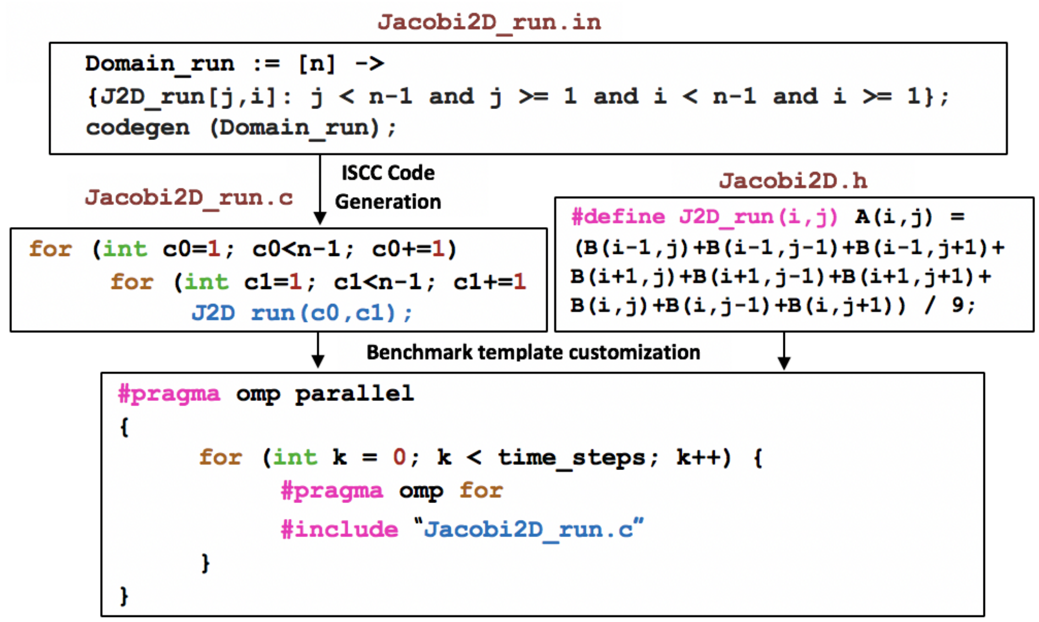
(a) Memory Performance in GiB/s



(b) Memory Performance in GFLOPS/s

Figure 4.13: Comparing the performance of Triad, Scale and 3-pt Jacobi 1D.

Figure 4.13 illustrates how widely their performance varies with small changes in the instruction mix and order, and the access pattern. Viewing performance as a function of bandwidth shows that triad is the more efficient pattern and viewing it as a function of FLOPs per second shows that Jacobi 1D performs best. The data presented in Figure 4.13 are benchmarked results from the drivers generated using the independent data spaces template avoiding false sharing. The benchmark driver for scale is generated similar to the driver created for triad in section 4.1.3, after modifying the pattern specifications accordingly for scale. demonstrate the impact of access patterns. The case studies on streaming kernels, stencil patterns and indirect access patterns using AdaptMemBench in this chapter further elaborate on the impact of application characterisitics on achieved performance for a given access pattern.



```
Domain_run := [n] ->
{J2D_run[j,i]: j < n-1 and j >= 1 and i < n-1 and i >= 1};
codegen (Domain_run);
```

```
for (int c0=1; c0<n-1; c0+=1)
     for (int c1=1; c1<n-1; c1+=1
          J2D run(c0,c1);
```

```
#define J2D_run(i,j) A(i,j) =
(B(i-1,j)+B(i-1,j-1)+B(i-1,j+1)+
B(i+1,j)+B(i+1,j-1)+B(i+1,j+1)+
B(i,j)+B(i,j-1)+B(i,j+1)) / 9;
```

```
#pragma omp parallel
{
      for (int k = 0; k < time_steps; k++) {
            #pragma omp for
            #include "Jacobi2D_run.c"
      }
}
```

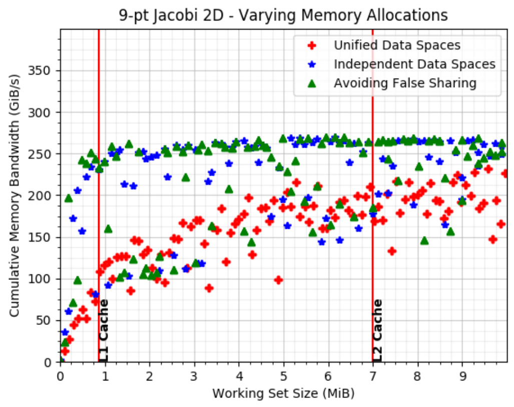
Figure 4.14: Illustration of custom benchmark generation for 9-pt Jacobi 2D kernel with unified data spaces using the polyhedral model.

4.4 Multidimensional Jacobi Kernels

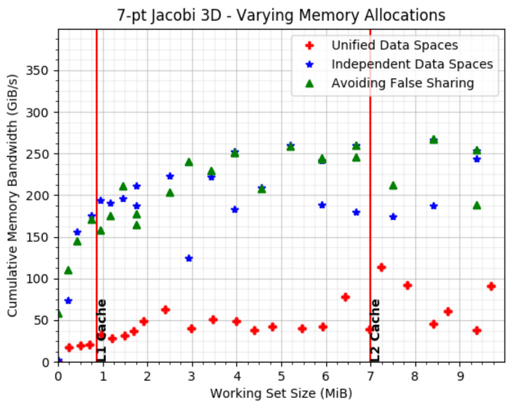
In this section, we evaluate the 9-pt Jacobi 2D and 7-pt Jacobi 3D kernels. The case study on tiling transformations showcases an optimization space exploration using AdaptMemBench.

The process of creating a custom benchmark driver for 9-pt Jacobi 2D using unified data spaces is illustrated in Figure 4.14. A 7-point Jacobi 3D benchmark driver can be similarly created with an added dimension to the code generation script and corresponding modifications to the pattern specification. The benchmark drivers for these two patterns can similarly be created using the independent data spaces template to understand the effect of shared data spaces and false sharing.

From Figures 4.15a and 4.15b, it can be observed that separating data spaces into different memory regions is beneficial for both Jacobi 2D and Jacobi 3D. However,



(a) 9-pt Jacobi 2D



(b) 7-pt Jacobi 3D

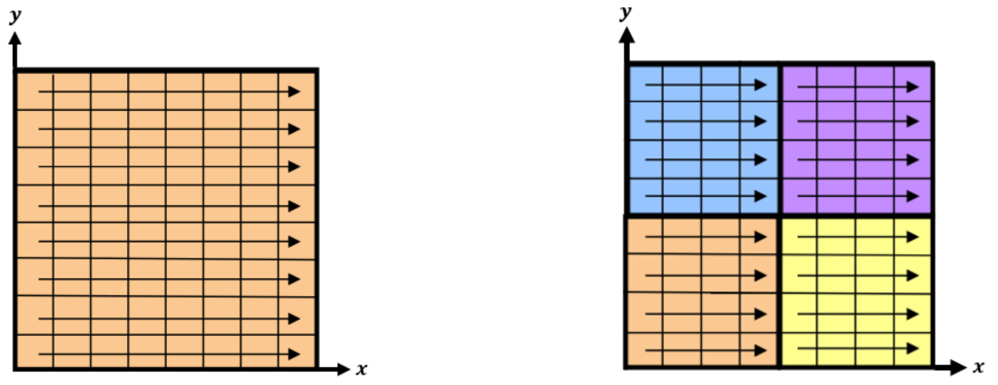
Figure 4.15: Impact of varying memory allocations for Jacobi 2D and Jacobi 3D.

false sharing doesn't affect performance and both the patterns struggle to scale in the L1 cache. For multidimensional stencils, factors beyond false sharing such as cache locality affect performance, which motivates us for tiling optimizations.

4.4.1 Tiling Optimizations

Though currently available caches are significantly large, stencil computations perform global sweeps through data spaces that are generally beyond the capacity of the data caches. The magnitude of data reuse is constrained to the typically small number of points in the stencil. This affects data locality in cache, which influences execution time within each computing node. Hence, these computations achieve only a small fraction of the theoretical peak performance.

Over the years, several optimization techniques [48, 45, 35, 31, 67, 8] have been proposed to improve the performance of multiple nested loops which are generally the most time-consuming parts of computationally intensive programs. Loop tiling optimizations [42, 66, 51, 20, 7] are one of those, which attempt to exploit data locality



(a) The original iteration space. (b) Modified iteration space on tiling

Figure 4.16: An illustration of the iteration space of two-dimensional stencil operation upon tiling.

in caches by operating computations on cache-sized chunks of data called tiles before moving on to the next tile. The size and shape of the tiles, and the scheduling strategy are the factors which impact locality, parallelism and communication cost of the code upon tiling.

Spatial Tiling Strategies

Rectangular space tiling [26] is one of the traditional optimization strategies for stencil computations. Rectangular tiling breaks a large iteration space into a set of smaller iteration spaces, which improves spatial and temporal locality. When iterating over a large two-dimensional data space applying a multipoint stencil, it is highly probable that one of the neighbors accessed would have fallen out of the cache while the iteration comes around to the same point again. Tiling iteration space eliminates such cache misses and improves data reuse. This optimization is explored, not to provide another data point on the impact of tiling, but to demonstrate the advantages of including polyhedral code representations in the framework.

Tiling 9-pt Jacobi 2D

We implement the rectangular tiling strategy on a 9-point 2D Jacobi iterative stencil. We initially decompose the equidimensional 2D iteration space into smaller square tiles of size ranging from 16 to 256 for a maximum grid size of 3000. With the results obtained on square tiling, we observed it is not beneficial for any data size, even on collapsing the loop nests using OpenMP. Listing 4.8 shows the input code to polyhedral compiler and the corresponding code generated.

```
Domain_run := [n] -> {
    J2D_Tiling_run[j,i] : i <= n and i >= 1 and j<=n and j >= 1;
};
Tiling := [n] -> {
    J2D_Tiling_run[j,i] -> J2D_Tiling_run[tj,ti,j,i]:exists rj,ri
   :
                and 0<=rj<64 and j=tj*64+rj
                and 0<=ri<32 and i=ti*32+ri;
};
codegen (Tiling * Domain_run);
```

```
for (int c0 = 0; c0 <= floord(n, 64); c0 += 1)
  for (int c1 = 0; c1 <= n / 32; c1 += 1)
    for (int c2 = max(1, 64 * c0); c2 <= min(n, 64 * c0 + (64 -
   1)); c2 += 1)
      for (int c3 = max(1, 32 * c1); c3 <= min(n, 32 * c1 + (32 -
    1)); c3 += 1)
        J2D_Tiling_run(c2, c3);
```

Listing 4.8: ISCC script `Jacobi2D_xy_tiled.in` and the generated C code file `Jacobi2D_xy_tiled.c`.

A rectangular tile sweep, i.e., an exhaustive search for the most performant tile size and shape was performed. The memory bandwidth results are plotted in the perpendicular y-axis of the 3D graph with horizontal z-axis denoting the area of the tiles executed for the working set sizes in x-axis. We observe no speedup with any tile area for any data size in figure 4.17, compared to the peak performance of each data

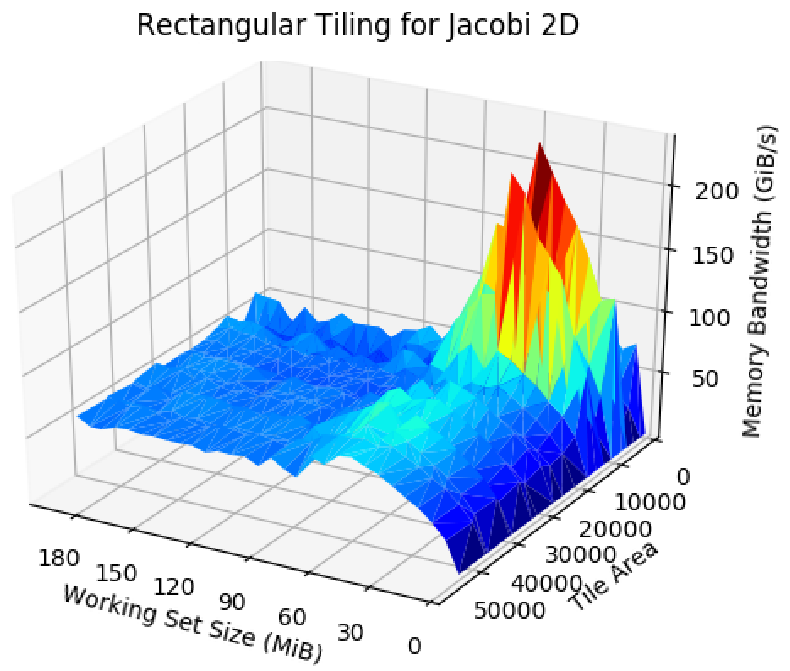


Figure 4.17: Achieved memory bandwidth with rectangular sweep for Jacobi 2D with a tile sweep for sizes ranging from 16 to 256 in both the directions. Higher the bandwidth, better is the memory performance.

size in figure 4.15a. The reason for this well-known tiling strategy being ineffective in practice is largely-sized on-chip data caches in modern microprocessors, causing the entire data space to fit in caches, limiting data reuse [28].

Tiling 7-pt Jacobi 3D

We now implement this spatial tiling strategy on the 7-point Jacobi 3D transformation. The initial approach is to tile in the 3D grid in all directions. We initially block the iteration space in all the three dimensions, for block sizes 16×16, 32×32 and 64×64. The experimental results show no performance gain with strategy caused by poor spatial locality due to frequent discontinuities in the memory stream since the array elements are read in a non-contiguous fashion (as described in [28]).

We implement the partial blocking strategy [45] in which blocking is done in two least significant dimensions alone. This results in a series of 2D slices that are stacked one over the other in the unblocked dimension. Listing 4.9 shows the ISCC input script and corresponding C code file generated for this benchmark. AdaptMemBench simplifies the implementation of this optimization with this input ISCC script as execution schedule file with the other pattern specifications remaining the same as for the naïve Jacobi 3D benchmark.

```
Domain_run := [n] -> {
    STM_3DS_run[k,j,i] : i <= n and i >= 1 and j<=n and j >= 1
    and k<=n and k >= 1;
};
Tiling := [n] -> {
    STM_3DS_run[k,j,i] -> STM_3DS_run[tk,tj,ti,k,j,i]:exists rk,
    rj,ri:
                    0<=rk<32 and k=tk*32+rk
                and 0<=rj<64 and j=tj*64+rj
};
codegen (Tiling * Domain_run);
```

```
for (int c0 = 0; c0 <= floord(n, 64); c0 += 1)
  for (int c1 = 0; c1 <= n / 32; c1 += 1)
    for (int c2 = max(1, 64 * c0); c2 <= min(n, 64 * c0 + 63); c2
    += 1)
      for (int c3 = max(1, 32 * c1); c3 <= min(n, 32 * c1 + 31);
    c3 += 1)
        for (int c4 = 1; c4 <= n; c4 += 1)
          STM_3DS_run(c2, c3, c4);
```

Listing 4.9: ISCC script `Jacobi3D_xyz_tiled.in` and the generated C code file `Jacobi3D_xyz_tiled.c`.

We tested the efficacy of this technique on grid sizes up to 256, with block sizes ranging from 16 to 64 in both directions. This approach too does not offer any speedup if we compare the peak bandwidth from Figure 4.15b with the most performant block area in figure 4.18. Large on-chip caches affect cache reuse and thus offer no performance gain with this blocking strategy. Increasing grid sizes would be

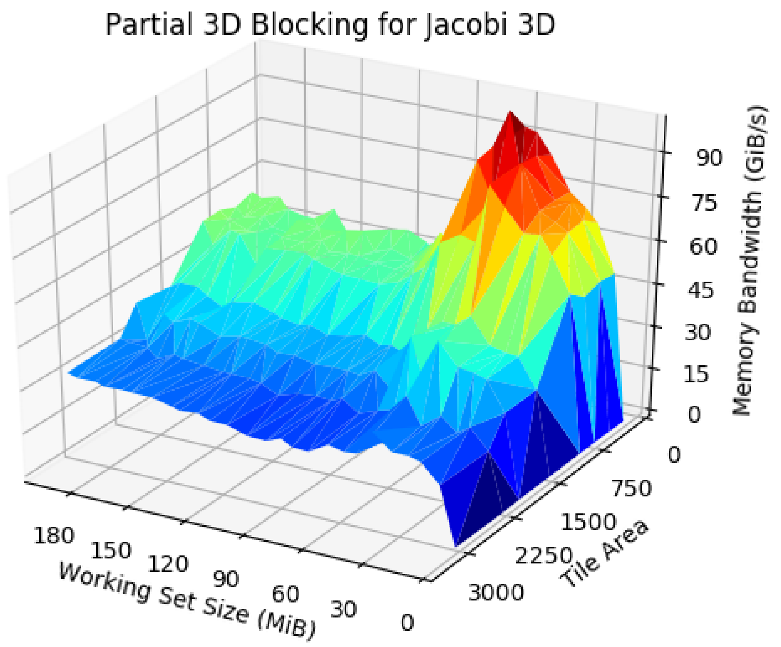


Figure 4.18: Achieved memory bandwidth with 2D Cache blocking for Jacobi 3D with a tile sweep for sizes ranging from 16 to 64 in both the tiled directions. Higher the bandwidth, the better is the memory performance.

impractical since many scientific applications, such as computation fluid dynamics, typically use a box size of 64^3 or less [3].

These results confirm conclusions from previous studies [28, 14, 27] on these tiling strategies performed for serial execution. We extend these studies to parallel applications and systems using with the flexibility of the polyhedral model offered by AdaptMemBench. This behavior of spatial tiling strategies motivates us to evaluate temporal tiling strategies such as the overlapped tiling [31].

Temporal Overlapped Tiling

Overlapped Tiling [31] is a space-time tiling strategy that eliminates inter-tile dependencies by duplicating points in the original iteration space, implying that same iteration point can be member of neighboring tiles, as in figure 4.19. This is achieved by adding a triangular region to the side of a standard tile overlapping with the iteration points in the right end of the neighboring tile. This removes dependency between adjacent tiles in the horizontal direction. This strategy improves data locality and eliminates the overhead of pipelined parallelism at the cost of slightly increased computational time.

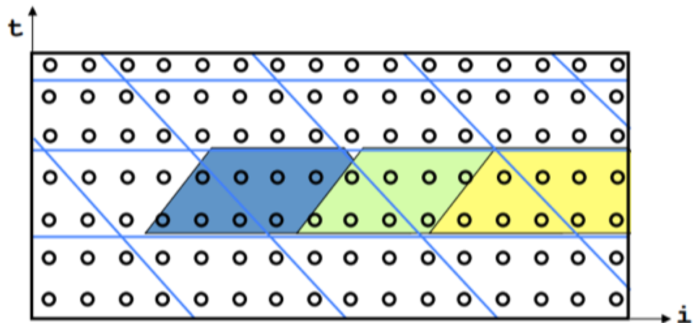


Figure 4.19: Overlapped tiling with 1D stencil and 1D time.

We implement the overlapped tiling technique on the 9-pt Jacobi 2D benchmark (figure 4.14). The listings 4.10 and 4.11 respectively show the ISCC input script and the corresponding tiling code generated. The first step is to unroll the outermost time iterator loop by 4. Each of the resulting iterations are tiled with overlapping iteration points across all the directions, the tile iterators are fused together for all the four unrolled iterations.

```
J2D_overlapped := [ntimes,n] -> {
    S1[t,i,j]->[t,ii,jj,0,i,j] : exists a,ri,rj: 1 <= t <= ntimes
    &&  t = 4a + 1 && 1 <= i <= n && 1 <= j <= n &&
                                    0-3<=ri<8+3 and i=ii*8+ri
                              and 0-3<=rj<8+3 and j=jj*8+rj;

S2[t,i,j]->[t,ii,jj,1,i,j] : exists a,ri,rj: 1 <= t <= ntimes &&
    t = 4a + 1 && 1 <= i <= n && 1 <= j <= n &&
                              0-2<=ri<8+2 and i=ii*8+ri
                         and 0-2<=rj<8+2 and j=jj*8+rj ;

 S3[t,i,j]->[t,ii,jj,2,i,j] : exists a,ri,rj: 1 <= t <= ntimes &&
    t = 4a + 1 && 1 <= i <= n && 1 <= j <= n &&
                              0-1<=ri<8+1 and i=ii*8+ri
                         and -1<=rj<8+1 and j=jj*8+rj ;

S4[t,i,j]->[t,ii,jj,3,i,j] : exists a,ri,rj: 1 <= t <= ntimes &&
    t = 4a + 1 && 1 <= i <= n && 1 <= j <= n &&
                              0<=ri<8 and i=ii*8+ri
                         and 0<=rj<8 and j=jj*8+rj ;};

 codegen(J2D_overlapped);
```

Listing 4.10: ISCC script `Jacobi2D_overlapped_tiled.in`

```
for (int c0 = 1; c0 <= ntimes; c0 += 4)
  for (int c1 = -1; c1 <= floord(n + 3, 8); c1 += 1)
    for (int c2 = -1; c2 <= floord(n + 3, 8); c2 += 1) {
      for (int c4 = max(1, 8 * c1 - 3); c4 <= min(n, 8 * c1 + 10)
   ; c4 += 1)
        for (int c5 = max(1, 8 * c2 - 3); c5 <= min(n, 8 * c2 +
   10); c5 += 1)
          S1(c0, c4, c5);
      for (int c4 = max(1, 8 * c1 - 2); c4 <= min(n, 8 * c1 + 9);
    c4 += 1)
        for (int c5 = max(1, 8 * c2 - 2); c5 <= min(n, 8 * c2 +
   9); c5 += 1)
```

```c
    S2(c0, c4, c5);
  for (int c4 = max(1, 8 * c1 - 1); c4 <= min(n, 8 * c1 + 8); c4 += 1)
   for (int c5 = max(1, 8 * c2 - 1); c5 <= min(n, 8 * c2 + 8); c5 += 1)
    S3(c0, c4, c5);
  for (int c4 = max(1, 8 * c1); c4 <= min(n, 8 * c1 + 7); c4 += 1)
   for (int c5 = max(1, 8 * c2); c5 <= min(n, 8 * c2 + 7); c5 += 1)
    S4(c0, c4, c5);
 }

```

Listing 4.11: The generated C code file `Jacobi2D_overlapped_tiled.c`

Overlapped tiling is evaluated with square grids of sizes 32, 64, 128, 256, 384, 512, 768, 1024, and 2048 in both the directions. Rectangular overlapping tile sizes of 8, 16, 32, 48, 64, and 128 in each direction were applied. For comparison, the same benchmarking constraints were applied for spatial rectangular tiling.

The achieved memory performance results comparing the original schedule, spatial rectangular tiling, and temporal tiling are shown in figure 4.20. The blue bar represents the achieved memory bandwidth with the actual schedule without tiling. The orange and green bars represent the most performant tile for rectangular tiling and overlapped tiling respectively.

As results indicate, the overlapped tiling consistently achieves higher performance compared to spatial tiling and the original schedule. Rectangular tiling is ineffective for grid sizes that fit into the caches. This tiling experiment clearly exhibits the

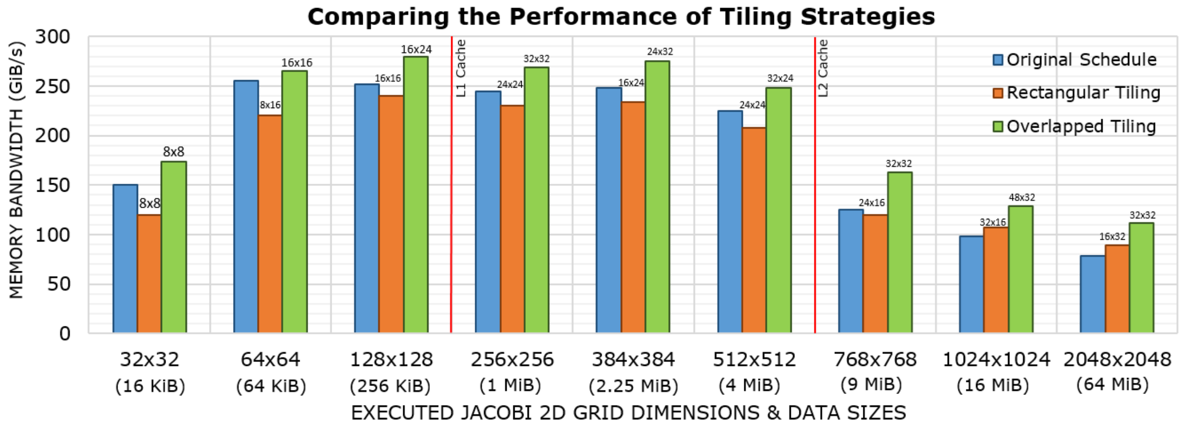


Figure 4.20: Comparing the achieved memory bandwidth between the original schedule, spatial rectangular tiling, and temporal overlapped tiling.

flexibility of the proposed framework for evaluating potential optimizations for given patterns. Other temporal tiling strategies such as the diamond tiling [7], time-skewing [67], cache oblivious blocking [20] can be implemented with AdaptMemBench, which is not covered in this work.

4.5 Sparse Matrix-Vector Multiplication

Sparse matrix computations are at the heart of various applications that involve iterative methods for solving large sparse linear systems, graph applications, and molecular dynamics. Sparse matrix representations store only the non-zero values of the matrix using supplementary arrays to record the row and column positions. These computations are memory bandwidth limited due to irregular accesses to these supplementary arrays and achieve only a fraction of theoretical peak performance.

In this thesis, we implement the Sparse Matrix-Vector (SpMV) multiplication kernel using AdaptMemBench to demonstrate its support for indirect memory access patterns. The SpMV kernel is commonly used in conjugate gradient solvers in sparse linear algebra. This kernel typically achieves 10% or less of the peak memory bandwidth [65], which motivates us to explore its performance with different representations using AdaptMemBench.

The SpMV kernel multiplies a sparse matrix with a dense vector and the resultant is a dense vector. Using index notations [44], it can be represented as:

$$y_i = A_{ij} \cdot x_j$$

where A is the input sparse matrix and x and y are the input and resultant dense matrices respectively.

In this work, we implement the SpMV kernel with Coordinate (COO), Compressed Sparse Row (CSR), Block CSR (BCSR), Diagonal (DIA), and Ellpack (ELL) formats (each of which is described in chapter 2.4). These SpMV representations are benchmarked with a collection of real-world sparse matrices with varying sparsity structures.

4.5.1 SpMV Implementation

Sparse matrix representations mostly store the non-zero data elements reducing computation and storage requirements. These storage representations involve indirection matrices to locate the non-zero elements in the corresponding dense matrix, and thus these memory access patterns are irregular and unpredictable at compile time. The block CSR, Ellpack and Diagonal representations exploit the structure of non-zero elements present in the sparse matrix by inserting a small number of zero-valued elements into sparse representations. This in effect makes the access patterns more sparse and regular, at the cost of increasing computations.

The irregular access patterns involve non-affine code constructs with indirect accesses like X[Y[i]]. In this thesis, we represent the polyhedral schedules for these constructs using uninterpreted functions supported by the Omega+ library [11]. Uninterpreted functions are mathematical symbols that act as placeholders and consist of a name and an n-ary form. The runtime realizations of uninterpreted functions can be implemented as arrays, functions, or macros. The input and output of the uninterpreted functions are unknown at compile time and cannot be statically analyzed, hence the term.

```
for (i = 0; i < N; i++)
  for(j = index[i]; j < index[i+1]; j++)
    y[i] += A[j]*x[col[j]];
```

Listing 4.12: SpMV code based on the CSR format

Listing 4.12 shows the implementation of SpMV based on the CSR format. The inner loop has non-affine loop bounds and the reference `x[col[j]]` has a non-affine array subscript. The iteration space for this SpMV code can be represented as:

$$I = \{[i, j] | 0 \leq i < N \wedge index(i) \leq j < index(i + 1)\}$$

The array expression in the loop bound of `j` is input as an uninterpreted function. The compiler encodes `index` as an uninterpreted function of the outer loop `i`. The argument to the uninterpreted function is encoded as a relation. The inputs to the relation are the outer loop variables and the output for the relation is the array subscript expression. The representation of non-affine loop bounds utilizing uninterpreted functions paves the way for transformations to manipulate these loop bounds, which are not evaluated in this work.

Similar to the CSR representation of SpMV, the SpMV kernel is implemented with the other storage formats using uninterpreted functions supported by Omega+. The generated code for COO, BCSR, ELL and DIA formats are shown in listings 4.13, 4.14, 4.15 and 4.16 respectively. The generated code is customized with the *non-affine spaces* driver template to create separate benchmark drivers for each of the storage representations.

```
for (i = 0; i < nnz; ++i)
    y[row[i]] += A[i] * x[col[i]];
```

Listing 4.13: SpMV code based on the COO format

```
for (ii = 0; ii < N/R; ii++) {
    for (jj = brow[ii]; jj < row[ii+1]; jj++) {
        kk = bcol[jj];
        for (ri = 0; ri < R; ri++) {
            for (ck = 0; ck < C; ck++) {
                i = ii * R + ri;
                k = kk * C + ck;
                m = offset3(jj, ri, ck, R, C);
                y[i] += a[m] * x[k];
            }
        }
    }
}
```

Listing 4.14: SpMV code based on the Block CSR format

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        k = cols[offset2(j,i,N)];
        y[i] += A[offset2(j,i,N)] * x[k];
    }
}
```

Listing 4.15: SpMV code based on the Ellpack format

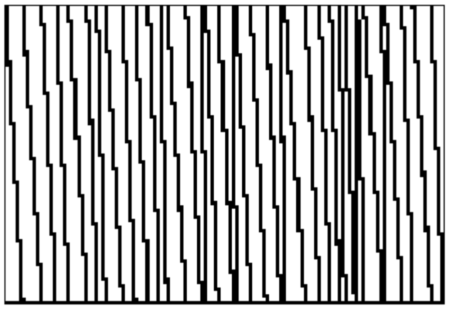
```
for (i = 0; i < N; i++) {
    for (d = 0; d < ND; d++) {
        k = ND * i + d;
        j = (i + offsets[d]) % N;
        y[i] += A[k] * x[j];
    }
}
```

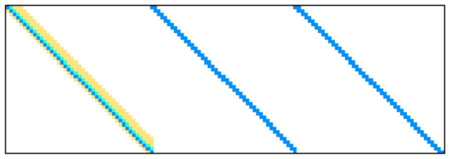
Listing 4.16: SpMV code based on the Diagonal format

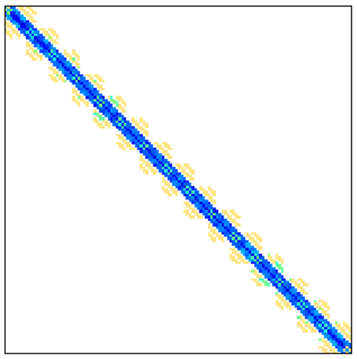
4.5.2 Evaluation

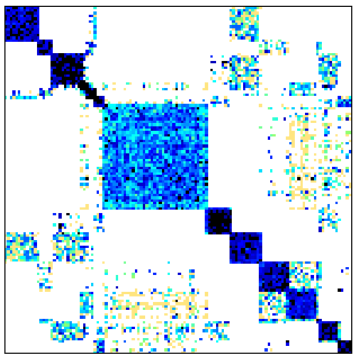
Table 4.2: Summary of the real-world sparse matrices from the SuiteSparse Matrix Collection [15] used in the experiment.

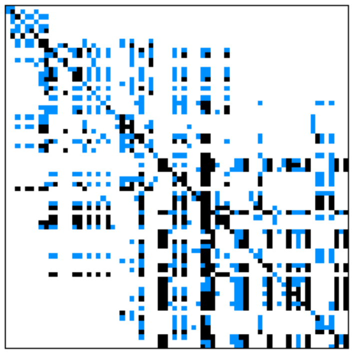
Matrix	**Domain**	**Dimensions**	**Non-zeros**	**Density**
`solver100K`	CFD	105127×154699	8018171	6×10^{-4}
`Facebook75K`	Social Media	75027×105436	2185005	2×10^{-4}
`microEco2`	Economics	195220×195220	3854576	4×10^{-4}
`spectral`	Machine Learning	250050×250050	6562418	1×10^{-3}
`lbnl01`	Networking	241628×241628	8564504	2×10^{-3}
`tweetsre1`	Social Media	95250×125740	7528526	2×10^{-3}
`pdb1M`	Protein Database	325245×325245	12854678	1×10^{-3}
`webbase`	Web connectivity	285475×285475	850379	9×10^{-5}
`pwtk`	Wind tunnel	157427×101525	4505420	2×10^{-4}
`scircuit`	Circuit	75452×156214	3505465	8×10^{-3}

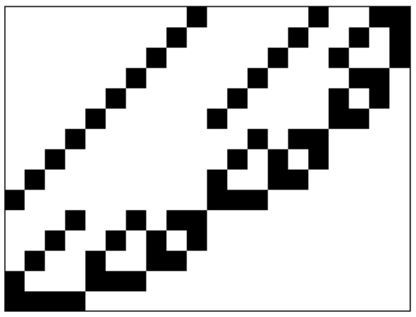

(a) `solver100K`

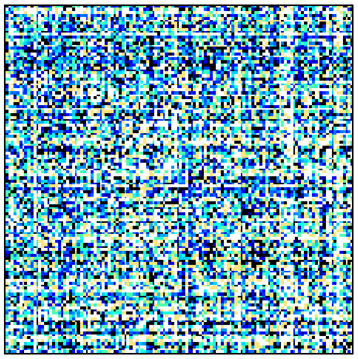

(b) `Facebook75K`

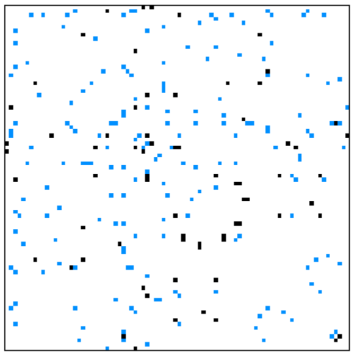

(c) `microEco2`

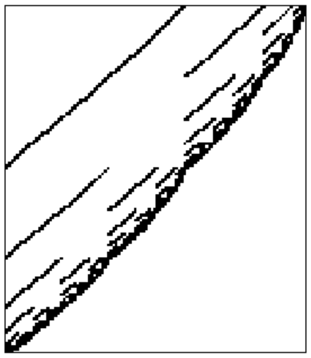

(d) `spectral`


(e) `lbnl01`


(f) `tweetsre1`


(g) `pdb1M`


(h) `webbase`


(i) `pwtk`

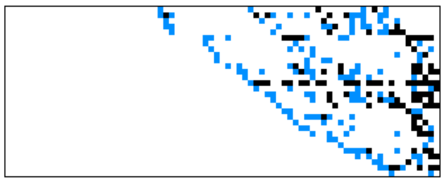

(j) `scircuit`

Figure 4.21: Sparse matrices from real-world applications with varying non-zero structures.

The different SpMV implementations are benchmarked with real-world matrices from the SuiteSparse Matrix Collection [15]. Table 4.2 reports some of the relevant statistics pertaining to these sparse matrices. Figure 4.21 shows pictures of each of the chosen matrices with varying density of non-zeros.

The sparse matrices were specifically chosen with an expectation to suit the different storage representations based on the structure of non-zeros present in the matrices. For example, we expect `tweetsrel` and `spectral` to perform well with

BCSR format, and `microEco2` and Facebook75K to be suitable for the Diagonal format.

Each of the SpMV benchmarks is executed with these ten sparse matrices. Each benchmark is executed 10 times and the average execution time is plotted in figure 4.22. The x-axis in the graph represents the 10 matrices benchmarked and the y-axis represents the average execution time in seconds.

The performance results are as expected. The following are the observations from these results:

- The sparse matrices `Facebook75K` and `solver100K` perform the best with the Ellpack format since both of these matrices consist a similar of number non-zeros in each row.
- Both DIA and BCSR representations run with lower execution times for `spectral` since it has non-zeros clustered at the diagonal in the form of blocks.
- `tweetsre1`, `lbnl01` and `spectral` perform the best with BCSR format since the non-zeros in these matrices are structured as blocks.
- The Diagonal format suits `microEco2` and `spectral` with the non-zero data elements in these matrices being dense at the diagonals.

As expected, we observe that no sparse matrix storage format is universally superior, each representation is well-suited for distinguishing conditions. The ideal format for sparse matrix computations depends on the structure and sparsity of the data. An application may require any, or several of these formats, and hence AdaptMemBench provides a unified flexible framework to examine the performance variation between different sparse representations for a given computation. This

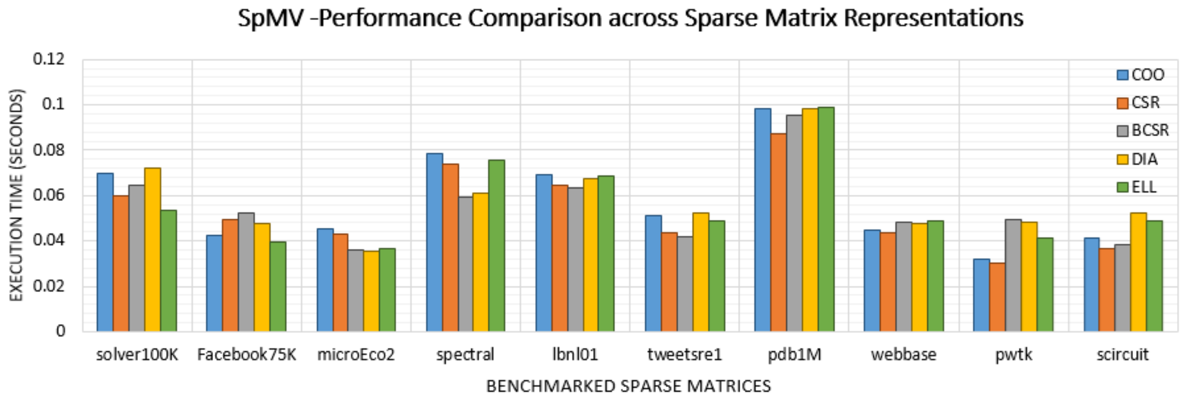


Figure 4.22: Comparing the execution time of the SpMV kernel implemented with different sparse matrix representations.

framework enhances reproducibility of performance results which could be shared between research groups and helps compare the results across different architectures.

4.6 Summary

The AdaptMemBench framework offers a convenient mechanism for manipulating isolated kernels from larger applications and measures execution characteristics. This framework assists in identifying the source of performance bottlenecks, testing potential optimizations of overcoming those bottlenecks, and interpreting the potential benefits of those optimizations. AdaptMemBench aids in experimenting with different data storage layouts, execution orders and parallelization strategies. This enhances reproducibility and sharing performance results, and porting to new systems. The case studies on diverse compute kernels in this chapter showcased the various capabilities of the proposed benchmarking framework, which are summarized as follows:

- The convenience to modify the memory mapping, execution order, and the statement instances, and the access to hardware counters facilitated evaluation of the impact of cross-thread communication and cache coherency on the memory performance of streaming and stencil patterns.

- The flexibility of the framework aided in manipulating the number of data spaces accessed in the triad. This experiment led to an interleaved execution schedule that achieved a speedup for triad in the L1 cache.
- The accommodation of the polyhedral model in the build system simplified the study on spatial and temporal tiling optimizations for multidimensional stencil operations, without requiring to rewrite the entire computation.
- AdaptMemBench's support for indirect memory access patterns enabled the performance comparison of sparse matrix operations implemented with different sparse matrix representations, benchmarked using real-world sparse matrices of varying sparsity structures.

Chapter 5

RELATED WORK

Several categories of memory benchmarks have been developed over the years. Most relevant to our work are the streaming bandwidth benchmarks, which use a predefined set of access patterns to measure achieved memory bandwidth, the stencil benchmarks and the cache bandwidth benchmarks. The following sections present representatives from each benchmarking category.

The proposed benchmarking framework adds capabilities beyond these benchmarks by offering configurability to explore the performance of scientific applications. It emulates application-specific memory access patterns and enhances reproducibility of performance results across different large-scale applications and architectures. It is a flexible and consistent proving ground for various code optimizations without needing to modify or port the entire application.

5.1 Streaming Bandwidth Benchmarks

STREAM [38] is a microbenchmark that measures sustainable memory bandwidth and the corresponding computation rates for the performance evaluation of high performance computing systems. STREAM measures the performance of four operations: COPY (`a[i] = b[i]`, measures data transfer without arithmetic), SCALE (`a[i] = q*b[i]`, with a simple arithmetic operation), SUM (`a[i] = b[i] + c[i]`, tests multiple load and store operations) and TRIAD (`a[i] = b[i] + q*c[i]`). The STREAM benchmark does not measure memory bandwidth for small data sizes in the higher levels of memory hierarchy, i.e., in level 1 cache and some portions of level 2 cache, depending on the target architecture. AdaptMemBench calculates the cumulative computation time for the overall execution of the kernel and enabling it to explore achieved performance in higher levels of cache.

MultiMAPS [50, 57] is a benchmark probe designed to measure platform-specific bandwidths, similar to STREAM, it accesses data arrays repeatedly. In MultiMAPS, the access pattern is varied in stride and array size varying spatial and temporal locality. It measures achieved memory bandwidth of different memory levels, different size working sets and a small set of access patterns. This benchmark is most closely related to ours. The primary difference is the ability of AdaptMemBench to include arbitrary memory access patterns, and test optimization strategies.

Stanza triad [28], a microbenchmark, is a derivative of STREAM, which measures the impact of prefetching on modern microprocessors. It works by comparing the bandwidth measurements by varying stanza length L and stride of access S for different data sizes and predicts performance. This being a serial benchmark, cannot be scaled to parallel applications, and cannot be configured for patterns other than triad.

5.2 Synthetic Memory Benchmarks

Apex-MAP [53] is a synthetic benchmark that characterizes application performance, implemented sequentially [54], and in parallel using MPI [55]. This benchmark *approximates* the memory access performance based on concurrent address streams considering regularity of access pattern, spatial locality, and temporal reuse. Using a set of characteristic performance factors, its execution profile is tuned such that these factors act as a proxy for the performance behavior of code with similar characteristics.

Stencil Probe [28] is a lightweight, flexible stencil application-specific benchmark that explores the behavior of grid-based computations. Stencil Probe mimics application kernels that use stencils on structured grids by modifying the operations in the inner loop of the benchmark. Similar to Stanza Triad, this benchmark is serial and cannot be extended to large-scale parallel applications and systems. Furthermore, this probe is not friendly for testing code optimizations and requires rewriting of the entire benchmark code for each transformation.

Bandwidth [49] is an artificial benchmark to measure memory bandwidth on x86 and x86_64 based architectures. This benchmark can be used to evaluate the performance of the memory subsystem, the bus architecture, the cache architecture, and the processor. Memory bandwidth is measured by performing sequential and random reads and writes of varying sizes across the levels of the memory hierarchy. However, this benchmark is neither application-specific nor customizable. It measures performance based on a predefined set of memory access patterns and cannot be configured specifically to a target application. Moreover, this benchmark executes serially and cannot be scaled to parallel systems and applications.

5.3 Application Benchmarks

Application Benchmarks are used as exemplars of application patterns. The NAS Parallel Benchmarks [6] comprises benchmarks developed to represent the major types of computations performed by highly parallel supercomputers and mimic the computation and data movement characteristics of scientific applications. It consists of five *parallel kernel* benchmarks (EP - an embarrassingly parallel kernel, MG - a simplified multigrid kernel, CG - a conjugate gradient method, FT - fast Fourier transforms and IS - a large integer sort) and three *simulated application* benchmarks (LU - lower and upper triangular system solution, SP - scalar pentadiagonal solver and BT - set of block tridiagonal equations).

The HPC Challenge benchmark suite [37] provides a set of benchmarks that define the performance boundaries of future Petascale computing systems. This hybrid benchmark suite examines the performance of HPC architectures as a function of memory access characteristics using different access patterns. It is composed of well known computational kernels such as STREAM, HPL [19], matrix multiply, parallel matrix transpose, FFT, RandomAccess and bandwidth/latency tests that span high and low spatial and temporal locality space.

The access patterns used in these application benchmarks is a predefined set and they do not effectively mimic the kernels found in diverse application code. This thus makes it intricate to refine their performance results specific to a given application in the architecture evaluated.

5.4 Cache Bandwidth Benchmarks

Likwid-Bench [60], part of Likwid-Tools [59] performance analysis suite, is designed to measure cache and main memory bandwidth of multithreaded assembly kernels. It is limited to one-dimensional streams and permits execution of a maximum of 38 streams, and it is impossible to benchmark multidimensional data structures. `pmbw` (Parallel Memory Bandwidth Benchmark) [2] is an effort that overcomes the limitations of likwid-bench. However, both of these tools are coded in assembly for serial execution with architecture-dependent intrinsics for vectorization, requiring to rewrite the entire benchmark for different instruction sets.

Deakin et. al. present a portable infrastructure [18] that sidesteps this shortcoming by using an extension of the BabelStream [16, 17] benchmark suite (described in 5.5) to determine cache bandwidth, coded using C and OpenMP that is autonomous of the architecture evaluated. This benchmark is however limited to the four kernels from the STREAM benchmark. AdaptMemBench is distinct from these works by allowing for configuration of the benchmarks and measurement of cache and DRAM bandwidths, and at the same time independent of the instruction sets.

5.5 Benchmarking Heterogeneous Systems

Heterogeneous computing systems are composed of a mix of compute devices such as commodity multicore processors, graphics processing units, reconfigurable processors, and others. SHOC [13] is a suite of benchmarks for scalable heterogeneous computing platforms, targeted at GPUs for the OpenCL and CUDA programming models. `clpeak` [9] is a benchmarking tool to measure peak performance capabilities of GPUs achieved using vector instructions, but is serial in nature and tests only a single device.

BabelStream [16, 17] is a tool for benchmarking GPUs and supports various programming models such as CUDA, OpenACC, OpenCL, Raja, and Kokkos. BabelStream doesn't include transfer time over the PCIe bus, unlike the other GPU benchmarks like SHOC. BabelStream is an enhancement of the STREAM benchmark to GPUs, and is thus limited to its four streaming kernels.

Though AdaptMemBench currently benchmarks only single-node performance with OpenMP, it can be very well extended to GPUs and support for different parallel programming models by accommodating different benchmark driver templates in the future.

5.6 Performance Models

While benchmarks *measure* achieved performance on a hardware platform, analytical and theoretical performance models *predict* the performance or runtime of the code in question. Performance models reduce complexity by simulating abstractions which provide a *close prediction* of the code to be executed, eliminating the need of running that code on different hardware architectures to understand potential performance bottlenecks. Benchmarks, on the other hand, provide a *perfect and accurate performance measure* of the executed code by actually running those kernels on the target platform. Thus, benchmarks and performance models deal with a balance between complexity and accuracy. Theoretical performance models most relevant to this thesis are the Roofline model and the Execution-Cache-Memory model which are described in the following subsections.

5.6.1 The Roofline Model

Sustainable memory bandwidth can be predicted using the Roofline Model [47]. This model is optimistic by its design, i.e., it yields an absolute lower execution time limit for a loop. The model gives the maximum achievable bandwidth for memory hierarchy on a machine by evaluating arithmetic intensity and a maximum floating point operations per second (FLOP/s) rate for the architecture. The Roofline model is based on the fundamental assumption that data transfer across the memory hierarchy overlaps perfectly with in-core execution (computational work).

The Roofline model for a given architecture can be plotted using the Empirical Roofline Toolkit (ERT) [36]. The software calculates the maximum FLOP/s rate and arithmetic intensity, from which it derives the plot. It repeatedly makes calls to either 1-FLOP or 2-FLOP kernel functions (i.e., `a = b + c`, `a = a*b + c` respectively) to execute the desired number of FLOPs. Figure 5.1 shows the Roofline model executing a 2-FLOP on a single node with 28 threads in the R2 cluster.

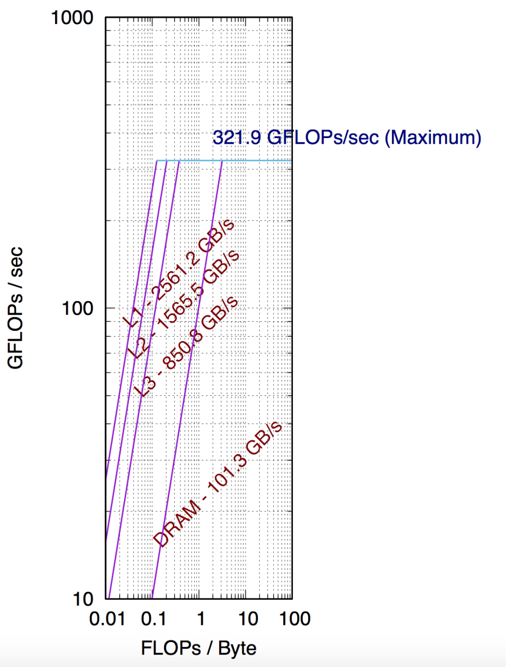


Figure 5.1: Roofline Model for R2 HPC Cluster at BSU.

This model computes the bounds of memory performance in a given architecture using simple synthetic application kernels. The Roofline model hence is not indicative of the performance capabilities of a given application and understand the outcome of different code transformations.

5.6.2 The Execution-Cache-Memory Model

The Execution-Cache-Memory (ECM) model [58, 23] is based on the same fundamental idea as the Roofline model that the data transfer or execution of instructions, whichever takes longer, determines the runtime of the loop. In contrast to the Roofline model, it drops the assumption of a single bottleneck- transfers of data through the memory hierarchy are serialized across the memory levels of a core and therefore contribute to the reduction of the total performance overlapping with one another. This model also presents a more precise metric than floating operations per second: cycles per cache line (cy/CL), the unit of work that gives more importance to the cache hierarchy.

The ECM model accommodates Layer Condition (LC) analysis [52, 21] that allows for predicting cache requirements for stencil codes. The basis of LC analysis is the least-recently-used (LRU) cache replacement policy, which is not perfectly implemented in large and real caches, but gives a good estimate on current architectures like the Intel and AMD CPUs. By taking the relative data access offsets and assuming sequential increments during the subsequent iterations, access hit/miss can be predicted depending on given cache sizes. The LC calculator can predict the optimal tile areas for spatial blocking of stencil codes. This, however, can neither predict the optimal tile size along each dimension, nor be extended to temporal blocking strategies. AdaptMemBench provides a testbed to interpret the optimal tile sizes invariant of the type of the tiling strategy evaluated.

Chapter 6

CONCLUSIONS

This thesis presents a configurable benchmarking framework to effectively emulate application-specific memory access patterns. This enables in characterizing the memory performance of compute kernels isolated from large applications. AdaptMemBench assists in identifying the influence of minor variations in the application characteristics on the overall performance. The use of the polyhedral model and associated code generation tools allows for quick development and experimentation with optimization strategies. This unified framework enhances reproducibility and sharing of performance results, and porting to new systems.

A collection of case studies exhibited various competences of the proposed benchmarking framework. The ability to manipulate the memory mapping, execution order, and the statement instances using AdaptMemBench enabled simplified analysis of the benefit of using distinct data spaces on threads and evaluate the overhead of false sharing on streaming and stencil patterns. A comparison of spatial and temporal tiling strategies was performed on multidimensional stencils with polyhedral code generation for flexible loop transformations. Sparse matrix-vector multiplication with different representations involving indirect access patterns was benchmarked with real-world sparse matrices of varying sparsity structures.

6.1 Future Directions

The work in this thesis can be extended in different directions which is summarized below:

- This framework currently supports shared memory parallelization. The support can be extended to benchmarking Graphics Processing Units (GPUs) and distributed memory systems by adding more benchmark driver templates.
- Automatic kernel extraction from large applications can be incorporated into or implemented in AdaptMemBench. KGen [29], Code Isolator [33] and Codelet Finder [34] are some of the existing tools that automatically extract kernels.
- Using the proposed framework, software-induced and hardware-induced performance variability across different architectures can be explored.
- Isolated portions of expensive compute kernels from real-world applications can be benchmarked using the proposed framework. The kernels studied in this thesis are proxies of realistic application patterns.
- Extend the case studies to a larger collection of application patterns, beyond steady-state loop kernels and indirect memory access patterns.

REFERENCES

[1] Limit-overview. `http://castl.cs.columbia.edu/limit/`. Accessed: 2019-01-10.

[2] pmbw — parallel memory bandwidth benchmark / measurement. `https://panthema.net/2013/pmbw/`. Accessed: 2018-10-30.

[3] M Adams, P O Schwartz, H Johansen, P Colella, T J Ligocki, D Martin, ND Keen, Dan Graves, D Modiano, Brian Van Straalen, et al. Chombo software package for amr applications-design document. Technical report, 2015.

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[5] Brett W Bader and Tamara G Kolda. Efficient matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[6] DH Bailey, E Barszcz, JT Barton, DS Browning, RL Carter, L Dagum, RA Fatoohi, Paul O Frederickson, Thomas A L, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[7] V Bandishti, I Pananilath, and U Bondhugula. Tiling stencil computations to maximize parallelism. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[8] Ian J Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L Chamberlain, David G Wonnacott, and Michelle Mills Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 197–206. ACM, 2015.

[9] K. Bhat. clpeak. `https://github.com/krrishnarraj/clpeak`. Accessed: 2018-07-20.

[10] Ronald F Boisvert, Roldan Pozo, Karin Remington, Richard F Barrett, and Jack J Dongarra. Matrix market: a web resource for test matrix collections. In *Quality of Numerical Software*, pages 125–137. Springer, 1997.

[11] Chun Chen. Polyhedra scanning revisited. *ACM SIGPLAN Notices*, 47(6):499–508, 2012.

[12] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[13] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[14] K Datta, S Kamil, S Williams, L Oliker, J Shalf, and K Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review*, 51(1):129–159, 2009.

[15] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[16] Tom Deakin and Simon McIntosh-Smith. Gpu-stream: Benchmarking the achievable memory bandwidth of graphics processing units. In *IEEE/ACM SuperComputing*, pages 3202–3216, 2015.

[17] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. Gpu-stream v2. 0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *International Conference on High Performance Computing*, pages 489–507. Springer, 2016.

[18] Tom Deakin, James Price, and Simon McIntosh-Smith. Portable methods for measuring cache hierarchy performance. *IEEE/ACM Super Computing*, 2017.

[19] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[20] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 361–366. ACM, 2005.

[21] Julian Hammer. Layer condition calculator. `https://rrze-hpc.github.io/layer-condition/#`. Accessed: 2019-01-03.

[22] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[23] Johannes Hofmann, Jan Eitzinger, and Dietmar Fey. Execution-cache-memory performance model: Introduction and validation. *arXiv preprint arXiv:1509.03118*, 2015.

[24] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[25] Eun-Jin Im and Katherine Yelick. Model-based memory hierarchy optimizations for sparse matrices. In *Workshop on Profile and Feedback-Directed Compilation*, volume 139, 1998.

[26] François Irigoin and Remi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, 1988.

[27] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60. ACM, 2006.

[28] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 workshop on Memory system performance*, pages 36–43. ACM, 2005.

[29] Youngsung Kim, John Dennis, Christopher Kerr, Raghu Raj Prasanna Kumar, Amogh Simha, Allison Baker, and Sheri Mickelson. Kgen: A python tool for automated fortran kernel generation and verification. *Procedia Computer Science*, 80:1450–1460, 2016.

[30] David R Kincaid, Thomas C Oppe, and David M Young. Itpackv 2d user's guide. Technical report, Texas Univ., Austin, TX (USA). Center for Numerical Analysis, 1989.

[31] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Effective automatic parallelization of stencil computations. In *ACM sigplan notices*, volume 42, pages 235–244. ACM, 2007.

[32] Mahesh Lakshminarasimhan and Catherine Olschanowsky. Adaptmembench: Application-specific memorysubsystem benchmarking, 2018.

[33] Yoon-Ju Lee and Mary Hall. A code isolator: Isolating code fragments from large programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 164–178. Springer, 2004.

[34] Chunhua Liao, Daniel J Quinlan, Richard Vuduc, and Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 308–322. Springer, 2009.

[35] Amy W Lim, Shih-Wei Liao, and Monica S Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. *Acm Sigplan Notices*, 36(7):103–112, 2001.

[36] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 129–148. Springer, 2014.

[37] P Luszczek, J J Dongarra, D Koester, R Rabenseifner, B Lucas, J Kepner, J McCalpin, D Bailey, and D Takahashi. Introduction to the hpc challenge benchmark suite. Technical report, Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), 2005.

[38] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[39] John D McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 19:25, 1995.

[40] P J Mucci, S Browne, C Deane, and G Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[41] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

[42] Jagannathan Ramanujam and Ponnuswamy Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 111–120. ACM, 1991.

[43] James Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.

[44] MMG Ricci and Tullio Levi-Civita. Méthodes de calcul différentiel absolu et leurs applications. *Mathematische Annalen*, 54(1-2):125–201, 1900.

[45] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3d scientific computations. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 32. IEEE Computer Society, 2000.

[46] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[47] David Patterson Sam Williams, Andrew Waterman. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM - A Direct Path to Dependable Software*, 52(4):65–76, April 2009.

[48] Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. *The International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[49] Zack Smith. Bandwidth: a memory bandwidth benchmark, 2008.

[50] A Snavely, L Carrington, N Wolter, J Labarta, R Badia, and A Purkayastha. A framework for performance modeling and prediction. In *Supercomputing 2002*, pages 21–21. IEEE, 2002.

[51] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices*, 34(5):215–228, 1999.

[52] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 207–216. ACM, 2015.

[53] E. Strohmaier and Hongzhang Shan. Apex-map: A global data access benchmark to analyze hpc systems and parallel programming paradigms. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 49–49, Nov 2005.

[54] Erich Strohmaier and Hongzhang Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 467–474. IEEE, 2004.

[55] Erich Strohmaier and Hongzhang Shan. Apex-map: A synthetic scalable benchmark probe to explore data access performance on highly parallel systems. In *European Conference on Parallel Processing*, pages 114–123. Springer, 2005.

[56] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[57] Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.

[58] Jan Treibig and Georg Hager. Introducing a performance model for bandwidth-limited loop kernels. In *International Conference on Parallel Processing and Applied Mathematics*, pages 615–624. Springer, 2009.

[59] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE, 2010.

[60] Jan Treibig, Georg Hager, and Gerhard Wellein. likwid-bench: An extensible microbenchmarking platform for x86 multicore compute nodes. In *Tools for High Performance Computing 2011*, pages 27–36. Springer, 2012.

[61] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: Lightweight performance tools. *Competence in High Performance Computing 2010*, pages 165–175, 2012.

[62] Sven Verdoolaege. barvinok: User guide. *Version 0.23), Electronically available at http://www. kotnet. org/skimo/barvinok*, 2007.

[63] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[64] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, pages 1–16, 2012.

[65] Richard Wilson Vuduc and James W Demmel. *Automatic performance tuning of sparse matrix kernels*, volume 1. University of California, Berkeley Berkeley, CA, 2003.

[66] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361. Society for Industrial and Applied Mathematics, 1987.

[67] David Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 171–180. IEEE, 2000.

[68] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

Appendix A

EXECUTING THE ADAPTMEMBENCH FRAMEWORK

This appendix explains the infrastructure and build instructions to reproduce the STREAM triad benchmark results with AdaptMemBench.

1. **Installation**

   For running this benchmarking framework, the following libraries are required to be installed:

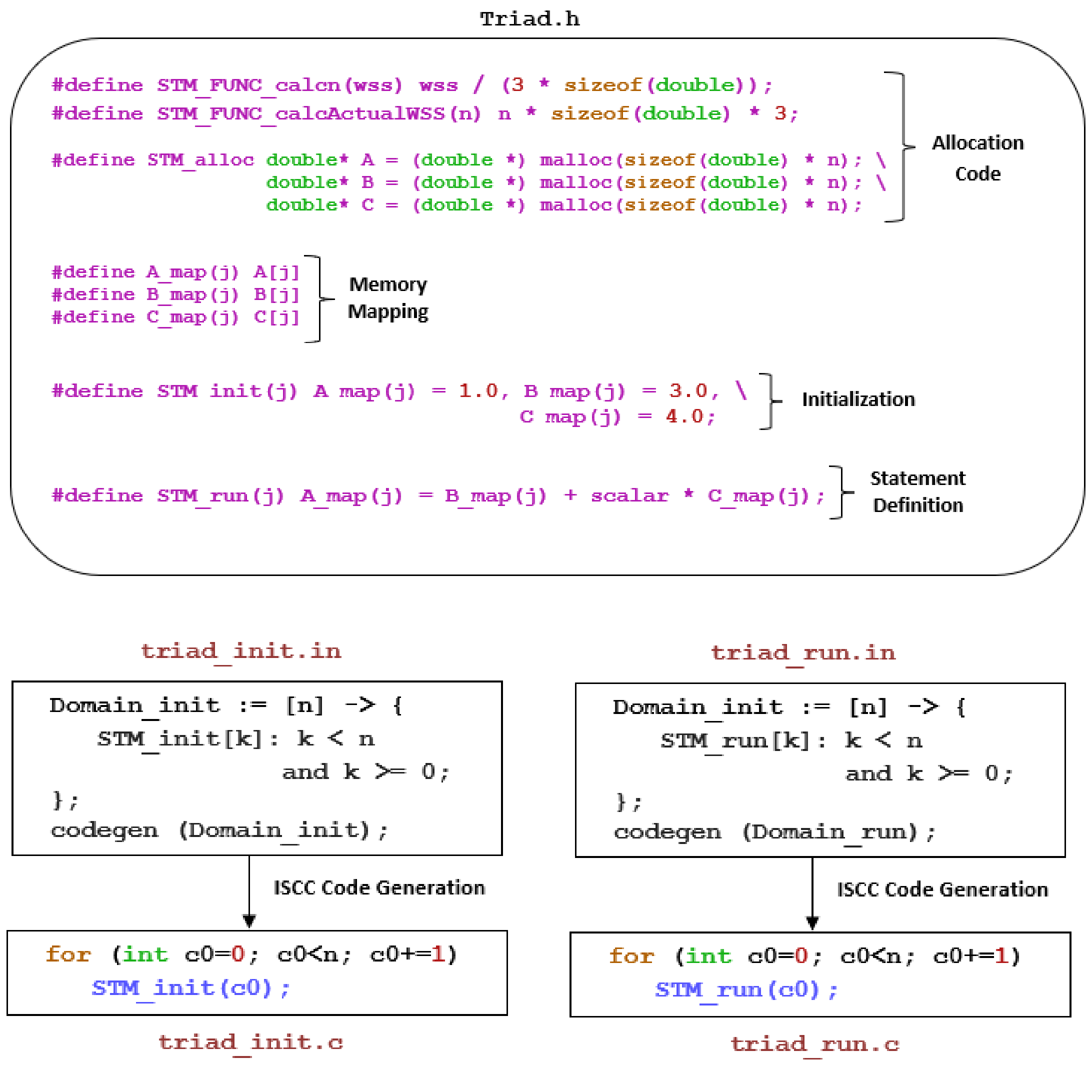
   (a) gcc/6.3.0
   (b) intel/compiler/64/2018/18.0.1
   (c) python/intel/3.5
   (d) papi/gcc/5.6.0 (The Performance API [40])
   (e) iscc/0.16.1 (The polyhedral code generation tool)
   (f) Omega+ Calculator (Part of the CHiLL library [12])

2. **Benchmark Customization**

   The entire process of customizing the driver template by substituting the pattern specifications and generating the benchmark driver code is fully automated by the means of a python script, which accepts the name of the kernel, driver template and benchmark mode as command line argument. For the STREAM triad kernel with timers, the benchmark driver using unified data spaces is generated as follows:

```
$ python adaptMemBench_driver_gen template=unified mode=unified
    kernel=triad
```

This creates the benchmark driver file named `adapt_membench_triad.cpp` customized for the STREAM triad kernel with the following user input pattern specifications.



Triad.h

```
#define STM_FUNC_calcn(wss) wss / (3 * sizeof(double));
#define STM_FUNC_calcActualWSS(n) n * sizeof(double) * 3;

#define STM_alloc double* A = (double *) malloc(sizeof(double) * n); \
                  double* B = (double *) malloc(sizeof(double) * n); \
                  double* C = (double *) malloc(sizeof(double) * n);

#define A_map(j) A[j]
#define B_map(j) B[j]
#define C_map(j) C[j]

#define STM_init(j) A_map(j) = 1.0, B_map(j) = 3.0, \
                    C_map(j) = 4.0;

#define STM_run(j) A_map(j) = B_map(j) + scalar * C_map(j);
```

triad_init.in

```
Domain_init := [n] -> {
    STM_init[k]: k < n
                 and k >= 0;
};
codegen (Domain_init);
```

triad_init.c

```
for (int c0=0; c0<n; c0+=1)
    STM_init(c0);
```

triad_run.in

```
Domain_init := [n] -> {
    STM_run[k]: k < n
                and k >= 0;
};
codegen (Domain_run);
```

triad_run.c

```
for (int c0=0; c0<n; c0+=1)
    STM_run(c0);
```

3. **Running the Benchmark**

The generated benchmark driver file `adapt_membench_triad.cpp` is compiled by running `make` command from the parent directory.

```
$ make adapt_membench_triad
```

The corresponding executable file named the same as the driver is run with the benchmark configuration parameters as command line arguments. For this kernel, the parameters are the working set size in bytes (–wss or -I), the number of threads (–num_threads or -T) and the total number of iterations (–ntimes or -N).

```
$ ./adapt_membench_triad --wss 1048576 --num_threads 4
   --ntimes 1000
```

4. **Performance Results**

   With the above input configuration parameters for the triad operation, the following performance results are obtained:

```
Configuration:adapt_membench_triad,
wss:1048576,num_threads:4,ntimes:1000

Cumulative Results:
num_elements_accessed:43690,
CumulativeMemoryBandwidth(GiB/s):122.008206,
TotalRunTime(s):0.008004

Performance Results with respect to individual threads:
Thread 0: 30.531278 GiB/s
Thread 1: 30.538607 GiB/s
Thread 2: 30.538566 GiB/s
Thread 3: 30.535006 GiB/s
```

Appendix B

COST OF RUNNING THE BENCHMARKS

Table B.1 shows the cost of running the benchmarks executed in terms of running time on a single R2 node. Execution times are put up for single, 14 and 28 thread configurations in 1000 time iterations.

Table B.1: Total running time of the benchmarks executed.

Kernel	**Total time taken for running the benchmark**								
	Serial Execution			14 Threads			28 Threads		
	L1/L2 (mins)	L3 (mins)	RAM (hours)	L1/L2 (mins)	L3 (mins)	RAM (hours)	L1/L2 (mins)	L3 (mins)	RAM (hours)
Triad	1	10	0.4	0.6	6	0.25	0.3	3.3	0.15
1D Stencil	5	20	0.8	3.5	11	0.45	2	7	0.3
2D Stencil	8	35	1.5	6	22	0.9	4	14	0.6
2D Stencil Rectangular Tile Sweep	30	120	108	18	75	88.5	11	65	68.2
3D Stencil	9	18	2.8	4.5	10	1.3	4	10	1.2
3D Stencil Partial blocking	20	80	25	9	47	15	7.5	39.5	13