A Systematic Approach for Obtaining Performance on Matrix-Like Operations

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Richard Michael Veras

B.S., Computer Science & Mathematics, The University of Texas at Austin
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

August, 2017

Acknowledgments

This work would not have been possible without the encouragement and support of those around me. To the members of my thesis committee – Dr. Richard Vuduc, Dr. Keshav Pingali, Dr. James Hoe and my Advisor Dr. Franz Franchetti – thank you for taking the time to be part of this process. Your questions and suggestions helped shape this work. In addition, I am grateful for the advice I have received from the four of you throughout my PhD. Franz, thank you for being a supportive and understanding advisor. Your excitement is contagious and is easily the best part of our meetings, especially during the times when I was stuck on a difficult problem. You encouraged me to pursue project I enjoyed and guided me to be the researcher that I am today. I am incredibly grateful for your understanding and flexibility when it came to my long distance relationship and working remotely.

To the Spiral group past and present and A-Level, you are amazing group and community to be a part of. You are all wonderful friends, and you made Pittsburgh feel like a home. Thom and Tze Meng, whether we were talking shop or sci-fi over beers, our discussions have really shaped the way I think and approach ideas, and that will stick with me for a long time. I would also like to thank my undergraduate advisor Dr. Robert van de Geijn at the University of Texas. You saw my interest in research and did everything you could to get me to graduate school.

To my parents Elsa and Osvaldo Veras, my sister Sandra and brother-in-law Chris, thank you for always being there and encouraging me on my childhood dream of becoming a scientist. To my in-laws, Gail and Terry, you have always shown me kindness and support and for that I thank you. To my amazing and perfect wife, Dr. Lauren Elizabeth Grimley, your constant love and support got me through the toughest times. You really helped push me through, and I could not have done this without you. Thank you my princess.

The work in this dissertation was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-16-2-0033, FA8750-12-2-0291 and HR0011-13-2-0007, the United States Department of Defense (DoD) under Contract No. FA8702-15-D-0002 and the National Science Foundation (NSF) under Contract No. 1116802.

Abstract

Scientific Computation provides a critical role in the scientific process because it allows us ask complex queries and test predictions that would otherwise be unfeasible to perform experimentally. Because of its power, Scientific Computing has helped drive advances in many fields ranging from Engineering and Physics to Biology and Sociology to Economics and Drug Development and even to Machine Learning and Artificial Intelligence. Common among these domains is the desire for timely computational results, thus a considerable amount of human expert effort is spent towards obtaining performance for these scientific codes. However, this is no easy task because each of these domains present their own unique set of challenges to software developers, such as domain specific operations, structurally complex data and ever-growing datasets. Compounding these problems are the myriads of constantly changing, complex and unique hardware platforms that an expert must target. Unfortunately, an expert is typically forced to reproduce their effort across multiple problem domains and hardware platforms.

In this thesis, we demonstrate the automatic generation of expert level high-performance scientific codes for Dense Linear Algebra (DLA), Structured Mesh (Stencil), Sparse Linear Algebra and Graph Analytic. In particular, this thesis seeks to address the issue of obtaining performance on many complex platforms for a certain class of matrix-like operations that span across many scientific, engineering and social fields. We do this by automating a method used for obtaining high performance in DLA and extending it to structured, sparse and scale-free domains. We argue that it is through the use of the underlying structure found in the data from these domains that enables this process. Thus, obtaining performance for most operations does not occur in isolation of the data being operated on, but instead depends significantly on the structure of the data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past seven decades, Scientific Computation joined the ranks of theoretical and experimental science as part of the three pillars of the scientific process. In this process, we make observations of natural phenomena and use the first pillar, theoretical, to deduce the underlying process in order to develop a theory. Using the second pillar, experimental, we test the accuracy of our theory. We then iterate over these steps to refine our theory until we get closer to the truth. However, we are limited in the experiments we can conduct in that they may be too large, too dangerous, too unrealistic to carry out, they may require a time scale that is too short or too long to be practical, or require time to flow backwards. The third pillar, Scientific Computation, addresses these limitations. Scientific queries are posed as mathematical operations which we implement as code and compute in order to obtain our answer. Problem size, time scale, and even the direction of time is no longer dependent on real world limitations. Instead these features are dependent on the computational power of the hardware and the ability of the scientific code to extract performance from that power. Thus, for Scientific Computation, obtaining performance from the code is key. This has been the driving force behind High Performance Computing (HPC), which is the science and application of producing high performance software and hardware. While much work has resulted in the automation of how we obtain performance, a perpetual frontier exists because we can only automate what we understand. Therefore, a great deal of performance is dependent on expert programmers who hand code and tune these application for every new piece of hardware.

The development of Scientific Computation has lead to the widespread availability of HPC resources and expertise. This has lead to ability to experimentally measure networks outside the original applications of HPC, such as those that arise in social, biological, financial, epidemiological, energy and transportation settings. In the same way that physics and engineering drove much of the early innovation in HPC, we see a rise in the use of computational techniques to answer questions in the natural and social sciences. Unlike the datasets collected from physics and engineering problems, the structure of these collections are seemingly irregular, hyper sparse and extremely large. Once again performance is critical in order to answer computational complex queries over these datasets. Thus not only has computation in the social sciences inherited the problems of HPC, but the complexity of their data has precluded the use of many existing HPC solutions. Is there a common class of problems between traditional scientific computing and big data where we can use the same performance techniques? And under what conditions are these techniques applicable on social science problems.

In this dissertation, I argue that if we have a matrix-like operation over structured data, then we can use the knowledge of this structure to produce a high performance implementation. More precisely, if we start with an operation that we can decompose into an efficient access pattern and compute core, then the dataset's structure allows us to determine two things: First, how to efficiently access and store the data in memory. Second, the structure determines how to specialize the computation to this problem. Thus, knowing the structure allows us to couple an efficient compute core to an efficient access to a high performance implementation of the original operation.

1.1 A Motivating Example

Alone, an efficient compute kernel or memory access pattern is not sufficient for performance, but rather both pieces are needed in tandem. We illustrate this in Figure 1.1 using a double precision matrix-matrix multiply (Gemm) as our example. This operation computes $C = AB + \hat{C}$, where $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$ and $C, \hat{C} \in \mathbb{R}^{m\times k}$. More precisely, it computes the following:

$$c_{i,j} = \sum_{p}^{k} a_{i,p} b_{p,j} + \hat{c}_{i,j} \tag{1.1}$$

Where each element $c_{i,j}$ of C is the result of an inner-product between a row in A and a column in B.

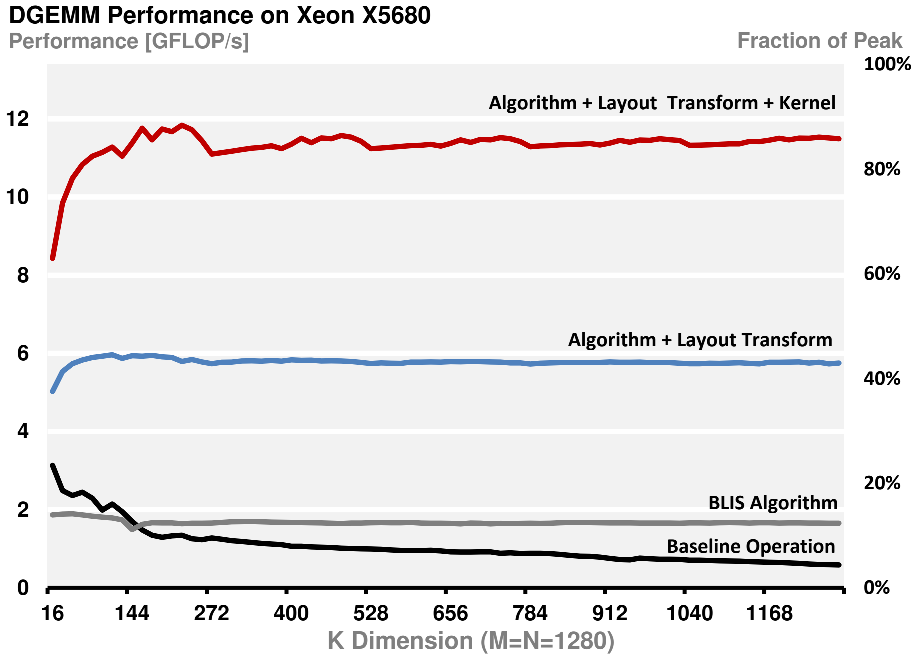


Figure 1.1: Performance starts with finding the correct algorithm and ends with combining data layout transformations with efficient kernels.

In order to motivate this two part approach to performance, we will break an implementation of Gemminto multiple pieces: a baseline implementation, an efficient algorithm, an algorithm turned to an efficient access pattern, and the access pattern coupled with a fast kernel. We show this in Figure 1.1, we compare the performance versus the problem size of four implementations. The first implementation, *baseline operation*, directly implements equation 1.1 as three loops in C without any additional algorithmic transformations. This implementation sustains 10% of the machine's peak performance. The second implementation, *BLIS Algorithm*, uses 8 loops to implement the Gemm algorithm in [1, 2]. The additional looping allows for sustained performance at 15% of the machine peak. It does this by insuring that the cache hierarchy is effectively used. However, the additional loops incur a substantial indexing overhead, in order to access the input matrices. These matrices are not stored in an order that is amenable to the access pattern determined by the algorithm. To correct this, the implementation marked *algorithm + layout transform*, uses the same algorithm as the previous implementation, but performs a data layout transformation on the A and B matrices such that the algorithm accesses each element in unit stride. The kernel in this implementation is naively implemented and relies entirely on the compiler to extract performance. Overall this implementation achieves near 45% of the machine's peak. The final implementation, *algorithm + layout transform + kernel* achieves near 90% of peak by replacing the naive kernel from the previous implementation with a generated and tuned kernel for that machine.

From Figure 1.1, both the data layout transformation and an efficient kernel are necessary for performance, but they both depend on finding an efficient algorithm for the target operation on a given dataset. This approach sketches how we handle the other three domains targeted in this thesis, where we will split the problems in two parts (access and kernel) and optimize each part to the problem.

1.2 My Contributions

The goal of this work is to uncover the necessary techniques and transformations for systematically producing high performance code for Linear Algebra and similar operations. Specifically, my contributions are as follows:

- I generalize a two part method for performance, used in dense linear algebra, to a broader set of domains.
- I develop a systematic approach to generating high performance dense matrix-matrix multiplication kernel that matches or exceeds the performance of expert written kernels.
- I provide a method for generating efficient vectorized time-tiled stencil kernels for finite difference and similar structed mesh applications that perform near their theoretical peak.
- I implement a high performance Sparse Matrix-Vector multiplication for scale-free data using a hierarchical sparse data structure. This implementation outperforms the state of the art Sparse Matrix-Vector library.
- I also implement a high performance graph analytic library for real-world scale-free data, which leverages the techniques from the three previous contributions. For a broad class of problems this library also outperforms the state of the art.

Chapter 2

A Method for High Performance

2.1 Introduction

In this chapter, we describe the method for producing high performance implementations of matrix-like operations in the domains of dense linear algebra, structured mesh, sparse linear algebra and graph analytics. This two part method – illustrated in Figure 2.1 – obtains performance by first yielding an efficient access pattern, which can provide a high rate of access through the memory hierarchy (Figure 2.2 left) to the second component of this process – a generated kernel that is tuned to the system's microarchitecture (Figure 2.2 right) to maintain this rate of computation. This method is a generalization of the Goto approach [2] for designing high performance implementations of dense matrix-matrix multiplication. In order to make this generalization from the dense to sparse we take into account the structure of the data, in particular the relationship between the elements in the dataset and their neighbors. By taking a structure-centric approach, we reshape the dataset and build algorithms on them in such a way that we are no longer computing on sparse data, but instead on dense clusters. We then use the same techniques for generating high performance dense matrix-matrix multiply kernels for these dense clusters. Thus, finding and capturing this structure is key for this method.

To elaborate on this method for high performance (Figure 2.1), we begin with the operation, which is a mathematical that maps the input of the

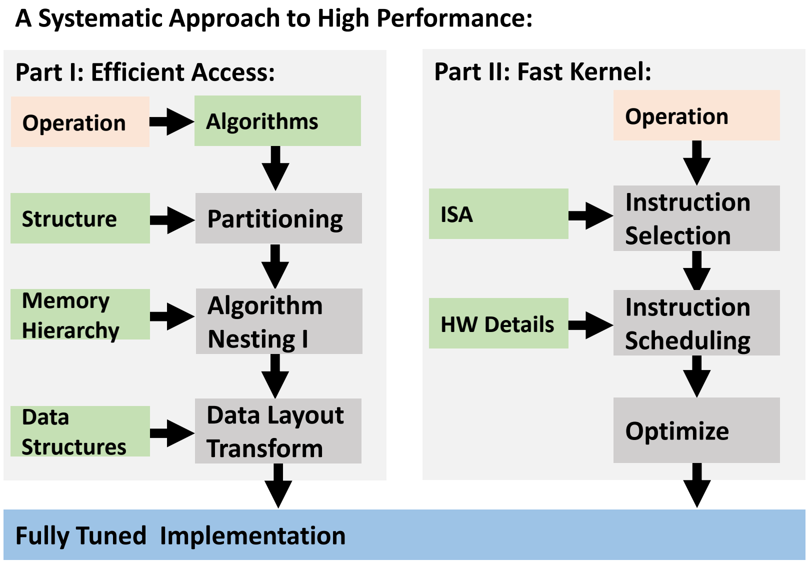


Figure 2.1: Throughout this thesis we use this two part process to produce high performance implementations of matrix-like operations. This method divides the implementation of an operation into an efficient access pattern that feeds an efficient kernel that performs the computation. This process is a generalization of the Goto technique [2] for high performance Matrix-Matrix Multiplication.

problem to the desired output. We restrict ourselves to operations that are similar to matrix-matrix, matrix-vector and iterative matrix-vector multiplication. This restriction allows us perform our computations using a class of algorithms, divide-and-conquer, which are amenable to machines with deep cache hierarchies.

How we layer these divide-and-conquer algorithms depends entirely on the input dataset is partitioned, which is why we focus on structured (dense and structured mesh) and semi-structured (scale-free networks) data. This partitioning is determined by the structure of the data because we want to hierarchically capture the dense clusters of elements in the data. With this hierarchical partitioning we determine the layering – or nesting – of divide-and-conquer algorithms that recursively compute on these partitions. We then pick a data structure that provides efficient access to the partitioned data. The dataset is then packed contiguously in this data structure, also prescribed by the algorithm nesting. These steps result in an efficient access pattern for a tuned computational. This kernel starts with the inner most nesting of algorithms from the previous step. We then select a mix of instruction to map this nesting of algorithms to the hardware. This mix is then statically scheduled and then optimized before being emitted as kernel code. We then combine the access pattern and kernel to produce the high performance implementation of the target operation.

We apply this approach in the remainder of thesis, where we target four domains using parts or the whole of this method. We selected these domains because they are representative of four of the computational dwarfs in [3] and span a large space of computational science. In Chapter 4, we automate this method by providing a systematic approach to generating expert level matrix-matrix multiply kernels. The data access portion of this process is provided by [2, 1]. In Chapter 5, we apply this method to structured mesh computations and we demonstrate how structure plays an important role in both the access pattern and in the kernel code. In Chapter 6, we demonstrate how to apply this entire method to sparse matrix operations on synthetic scale-free data which hinges on using the underlying structure in the data. We extend this to graph operations on real-world data in Chapter 7 to show that the necessary structure for this method does exist in real data. The remainder of this chapter is devoted to a detailed description of this method for high performance.

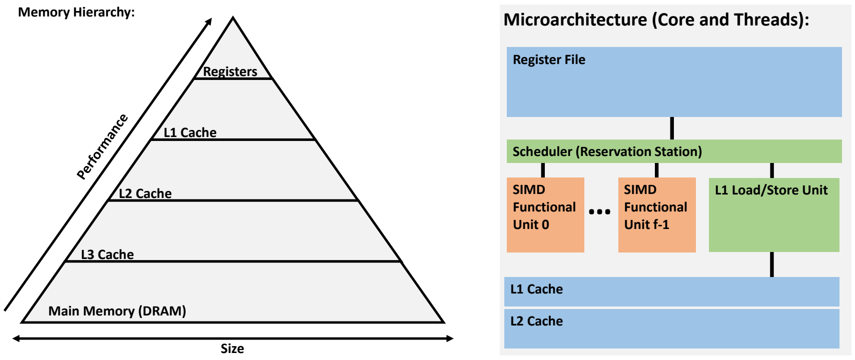


Figure 2.2: Our focus for performance is exclusively on the memory hierarchy and the microarchitecture. The method described in Figure 2.1 creates a high performance implementation by laying the dataset to efficiently feed a kernel tuned to the microarchitecture.

2.2 Preliminaries

In this section, we will provide a brief overview of the computational domains and the class of operations in these domains that we target. In particular, we will describe the linear algebra-like operations where this method for performance works. Lastly, this thesis focuses on obtaining high performance on modern computer architectures, therefore we will discuss the characteristics of these architectures.

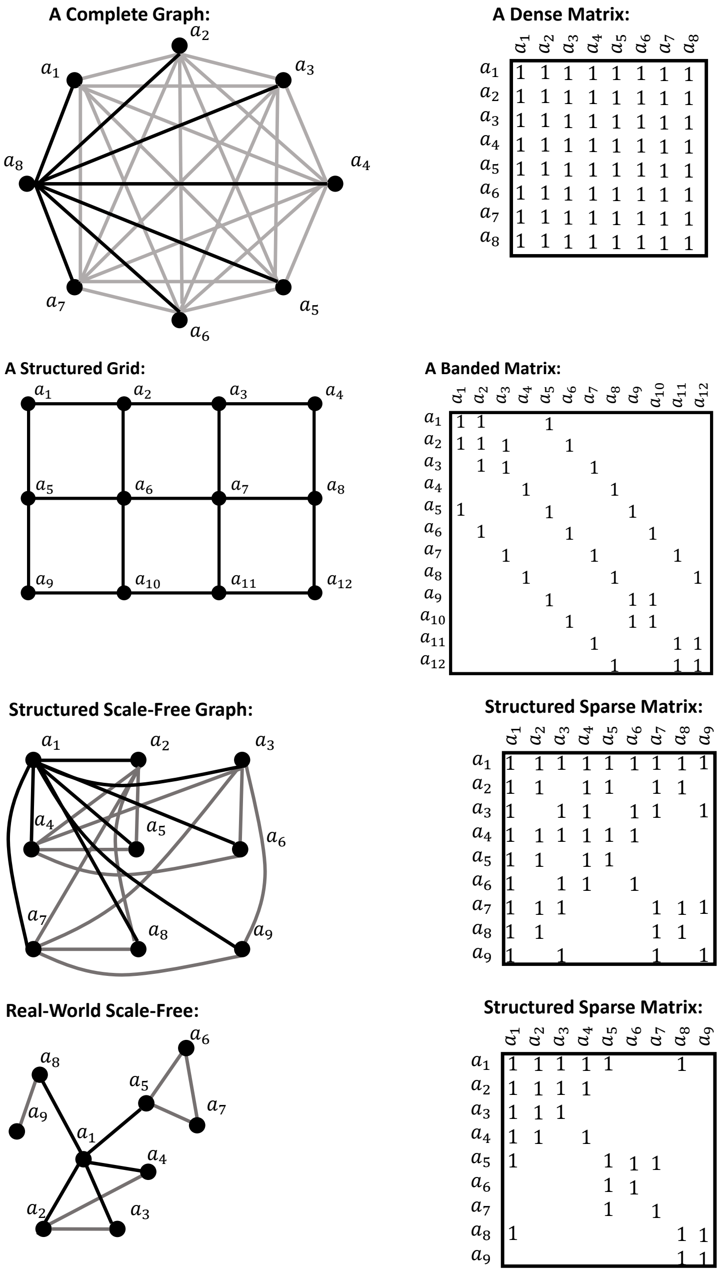
2.2.1 Domains

In Figure 2.3 we show a graphical representation along with its corresponding matrix representation of the data typically found in the four domains that we target. We picked these domains partly because they represent data typically seen in a large number of scientific applications. More importantly, we selected these domains because they present their own unique challenges in how data in these domains are structured.

Dense Linear Algebra. In this domain, datasets are dense because the value of an element is determined by a linear combination of all elements in the system. If we were to represent this graphically, this connectivity leads to a complete graph. This domain is at the heart of high performance computing to the extent that the performance of large machines are determined by the speed at which they can compute solutions to large dense systems of equations. Thus, modern computer architectures are typically optimized for these computations. We selected this domain because it provides the benchmark for our method.

Structured Mesh (Stencils). These computations typically arise in the computation of PDEs using finite-difference methods. The structure of this data is typically an n-dimensional mesh that corresponds to the relationship between an element and its neighbors. We chose this domain because exploiting this regular structure is critical for performance.

Sparse Linear Algebra. Data in this domains typically arises in the solution of PDEs using finite-element methods. The structure of this data is dependent on how the problem domain is divided into mesh elements. In



A Dense Matrix:

$$\begin{array}{c|cccccccc|}
 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \\
\hline
a_1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_6 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_7 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_8 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}$$

A Banded Matrix:

$$\begin{array}{c|cccccccccccc|}
 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} \\
\hline
a_1 & 1 & 1 & & & 1 & & & & & & & \\
a_2 & 1 & 1 & 1 & & & 1 & & & & & & \\
a_3 & & 1 & 1 & & & & 1 & & & & & \\
a_4 & & & & 1 & & & & 1 & & & & \\
a_5 & 1 & & & & 1 & & & & 1 & & & \\
a_6 & & 1 & & & & 1 & & & & 1 & & \\
a_7 & & & 1 & & & & 1 & & & & 1 & \\
a_8 & & & & 1 & & & & 1 & & & & 1 \\
a_9 & & & & & 1 & & & & 1 & 1 & & \\
a_{10} & & & & & & 1 & & & 1 & 1 & & \\
a_{11} & & & & & & & 1 & & & & 1 & 1 \\
a_{12} & & & & & & & & 1 & & & 1 & 1 \\
\hline
\end{array}$$

Structured Sparse Matrix:

$$\begin{array}{c|ccccccccc|}
 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\
\hline
a_1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_2 & 1 & 1 & & 1 & 1 & & 1 & 1 & \\
a_3 & 1 & & 1 & 1 & & 1 & 1 & & 1 \\
a_4 & 1 & 1 & 1 & 1 & 1 & 1 & & & \\
a_5 & 1 & 1 & & 1 & 1 & & & & \\
a_6 & 1 & & 1 & 1 & & 1 & & & \\
a_7 & 1 & 1 & 1 & & & & 1 & 1 & 1 \\
a_8 & 1 & 1 & & & & & 1 & 1 & \\
a_9 & 1 & & 1 & & & & 1 & & 1 \\
\hline
\end{array}$$

Structured Sparse Matrix:

$$\begin{array}{c|ccccccccc|}
 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\
\hline
a_1 & 1 & 1 & 1 & 1 & 1 & & & 1 & \\
a_2 & 1 & 1 & 1 & 1 & & & & & \\
a_3 & 1 & 1 & 1 & & & & & & \\
a_4 & 1 & 1 & & 1 & & & & & \\
a_5 & 1 & & & & 1 & 1 & 1 & & \\
a_6 & & & & & 1 & 1 & & & \\
a_7 & & & & & 1 & & 1 & & \\
a_8 & 1 & & & & & & & 1 & 1 \\
a_9 & & & & & & & & 1 & 1 \\
\hline
\end{array}$$

Figure 2.3: In this thesis we target four different domains: Dense Linear Algebra, Structured Mesh, Scale-Free and Real-World Networks. On the left we show a graphical representation of these domains with the matrix interpretation shown on the right. We picked these domains because they represent four of the original Seven Computational Dwarfs [3].

this thesis we actually restricted ourselves to sparse matrices generated from synthetic scale-free data. This data is fractal-like and hypersparse and fairly representative of real-world graph data. We chose this domain the show that the method for performance works on sparse scale-free data.

Graph Analytics. The data in this domain typical arises from biological and social networks. This data is typically scale-free, but random. A hallmark of this data is the hierarchical clustering of elements that naturally captures dense communities. This domain was selected to demonstrate that the method indeed works on real-world data.

2.2.2 Operations and Semirings

We restrict our method to matrix-like operations over the described domains. Specifically, these operations are matrix-matrix multiplication, matrix-vector multiplication and iterative matrix-vector multiplication which are described in Table 2.1. We say these are matrix-like operations because we define them over operation specific *semirings*. At a high level these semirings replace the addition and multiplication operator with a different function. More formally, a semiring is a set S with an additive operator $+$ (which is not necessarily the standard addition) and a multiplicative operator $\times$ (once again, not necessarily the standard multiplication), along with an additive identity 0 where $a + 0 = a$ and a multiplicative identity 1 where $a \times 1 = a$.

For example, if we want to compute Single Source Shortest Path (SSSP) as an iterative matrix-vector product, then we would formulate the semiring as follows: S is an integer value which corresponds to distance to the source, the additive operator $+$ is the minimum value between two elements in S, the multiplicative operator $\times$ is the addition of two elements from S, the additive identity is ∞, and the multiplicative identity is zero. In Table 2.2, we list the matrix operation and semiring for the operations discussed in this thesis. The author in [4] provides a listing of semiring-like operations for n-body problems and in [5] the authors describe graph operations in terms of semirings.

2.2.3 An Overview of Modern Computer Architecture

We target modern computer architecture which are characterized by two key features, deep memory hierarchies and complex microarchitectures. This is

Operation	**Matrix Form**	**Formalism**
Matrix-Matrix	$C = AB$	$c_{ij} = \sum_p^k a_{ip} b_{pj}$
Matrix-Vector	$y = Ax$	$y_i = \sum_j^n A_{ij} x_j$
Iterative Matrix-Vector	$y = A^t x$	$y = (\prod^p A)x$

Table 2.1: These are the matrix operations we target in this thesis. By defining these operations over arbitrary semirings we are able to target a large class of problems.

Operation	**Matrix Op.**	$+$	$\times$
Matrix-Matrix	$C = AB$	`fadd(a,b)`	`fmul(a,b)`
Matrix-Vector	$y = Ax$	`fadd(a,b)`	`fmul(a,b)`
Stencils	$y = A^t x$	`fadd(a,b)`	`fmul(a,b)`
SSSP	$y = A^t x$	`min(a,b)`	`add(a,b)`
PageRank	$y = A^t x$	`scaled_add(a,b)`	`dampen_add(a,b)`

Table 2.2: In this table we describe the operations treated in this thesis and how they are represented as matrix-like operations over arbitrary semirings.

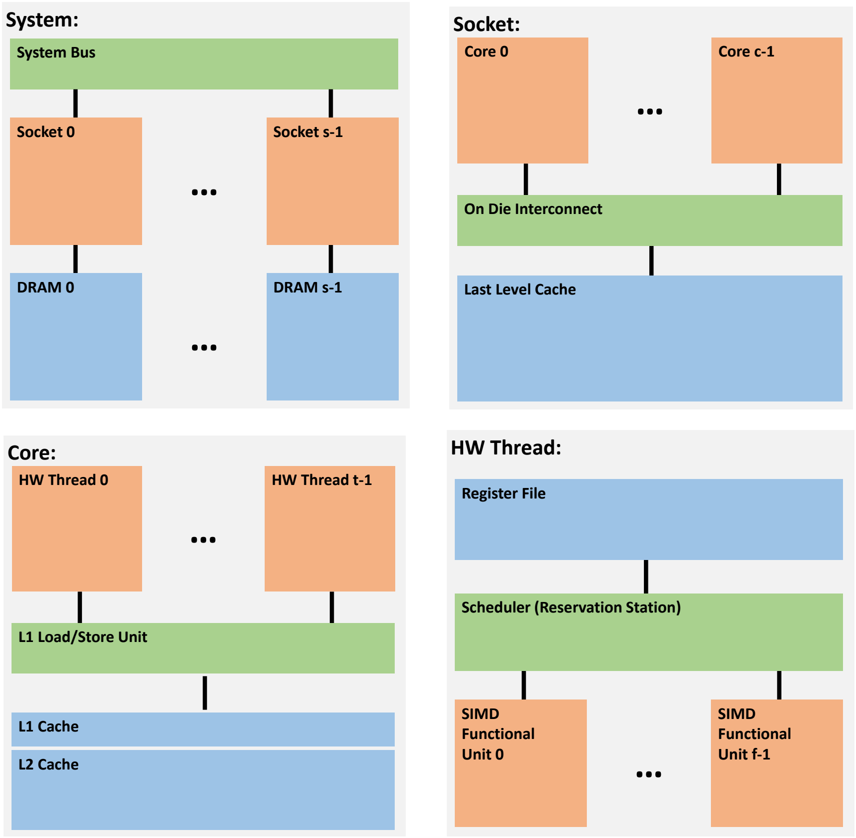


Figure 2.4: Here we provide a representative overview of a modern shared memory architecture. We color code each icon as follows: orange represents a computational unit, blue represents a storage unit and green corresponds to a communication unit. Additionally, we show how these units are connected to each other. For example, in the socket level view, each core must pass through a common interconnect in order to reach the shared last level cache.

why we split our method into two parts, where the first part focuses on the memory hierarchy and the second part focuses on generating a tuned kernel for the microarchitecture. In Figure 2.4, we illustrate a modern computer architecture at multiple levels of granularity from the system level all the way down to a functional unit view. Note, in each of these views, we distinguish between compute, memory and communication units and how they interconnect.

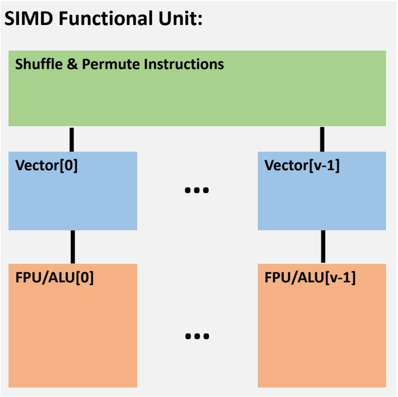


Figure 2.5: We can subdivide Figure 2.4 by exposing the view of a SIMD functional unit which compute short vector SIMD instructions. These functional units perform multiple independent operations simultaneously using a single instruction.

Continuing with the figure, we start at the system level which is composed of multiple compute sockets where each socket can directly access its own memory and can access the memory of its neighbors at the added cost of communicating over a shared bus. This configuration is called Non-Uniform Memory Access (NUMA). Second, we can decompose these sockets into multiple compute cores that communicate to each other over a shared On-Die Interconnect that connects to a shared Last Level Cache (LLC). This configuration, where each compute element shares access to the same memory at the same speed, is called Symmetric Multi Processor (SMP). Next, we can divide a core into Hardware Threads that share functional units that access the L1 and L2 cache. These hardware threads also share the underlying hardware resources on the core. We can view each thread as having its own private memory in the form of a register file, where computation on this register file is performed by a variety of Single Instruction Multiple Data (SIMD) functional units and communication between these units is governed by the instruction scheduler. These SIMD instructions compute short vector operations using a single instruction, and are necessary for performance on these architectures. We go even further in describing these SIMD functional units in Figure 2.5. Here each element in a SIMD functional unit corresponds to a location in a vector register. We have multiple scalar functional units which compute on specific locations in the SIMD vector and the communication of elements between these vectors is performed by the shuffle and permute engine. In Figure 2.2, we provide a simplified view of a modern architecture where we show the memory hierarchy and a combined view of the microarchitecture.

Given a computer architecture that fits this described model, performance is obtained by efficiently streaming the dataset through this hierarchy of memory and caches, and effectively using all of the computational units in parallel. In the next two sections, we will show how we target operations to this hierarchy and microarchitecture.

2.3 A Method for Efficient Data Access

We have a two part method for obtaining performance on modern computer architectures that splits the problem into efficient data access and kernel code generation. The goal of the first part of our method (labeled Part I in Figure 2.1) is to produce an efficient access pattern that brings the working dataset through the memory hierarchy as quickly as possible. This process entails dividing the original dataset in smaller working sets that can fit through the various levels of cache between DRAM and registers (left side of Figure 2.2). We achieve this by partitioning the dataset hierarchically in a manner that preserves tightly connected clusters. We then implement the target operation as layering of divide-and-conquer algorithms over the partitioned data which will move these partitions through the memory hierarchy in a cache efficient manner. Once we determine a layering of algorithms, we transform and repack the dataset to match how it is accessed. This insures that the kernel generated in the next section (labeled Part II in Figure 2.1) has contiguous access to the data. In the remainder of this section we will detail this process and in the next section we will connect this to the kernel generation.

2.3.1 Partitioning the Data

Partitioning the dataset is critical for performance because the operations we target essentially compute an element's value based on its neighbors (for example PageRank). Thus, maintaining the physical locality of the dataset in how it is partitioned will maximize cache utilization of neighboring elements. In our approach, we recursively partition our dataset in a manner

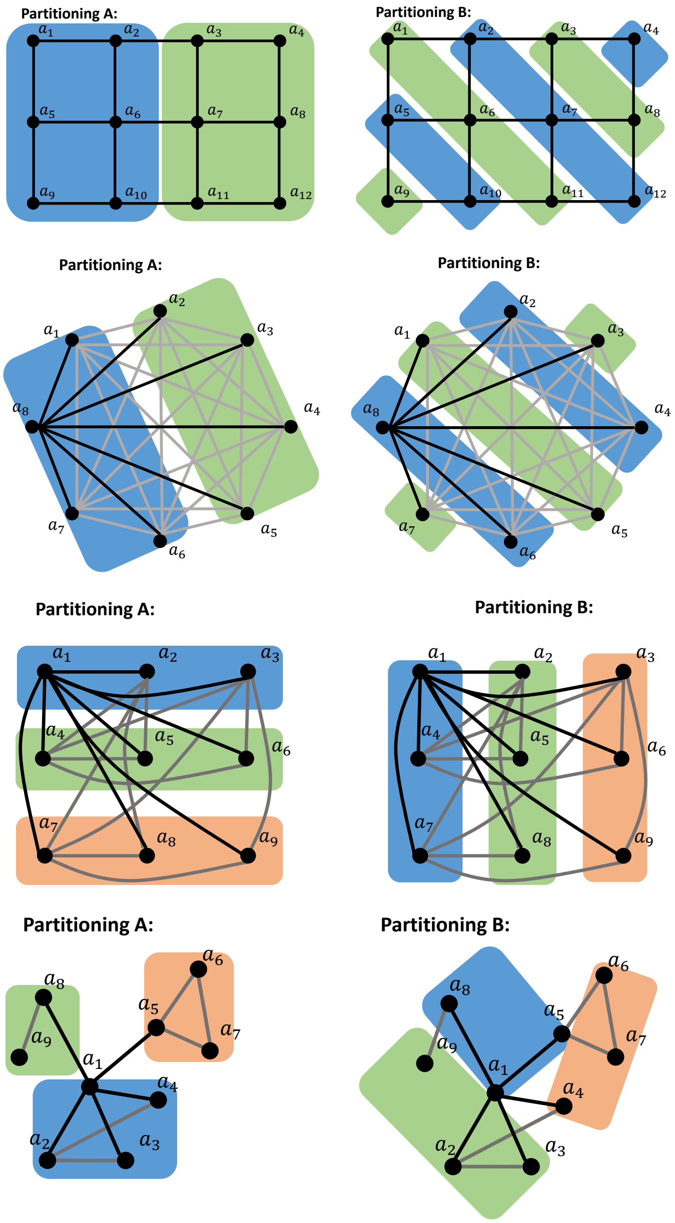

Partitioning A:
Partitioning B:
a_1 a_2 a_3 a_4
a_5 a_6 a_7 a_8
a_9 a_{10} a_{11} a_{12}
Partitioning A:
Partitioning B:
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
Partitioning A:
Partitioning B:
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9
Partitioning A:
Partitioning B:
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9

Figure 2.6: For each domain we show two different strategies for partitioning the data using only two partitions in the first two domains and three in the last two. How we partition our data will greatly affect the performance of the computations being performed on them. Ideally, we want partitions to capture clusters of neighboring data.

that hierarchically captures compute dense communities in the data. This allows us to efficiently map the dataset to memory using divide-and-conquer algorithms. These algorithms work by recursively traversing through this hierarchical partitioning until a small enough partitioned is reached an computed on. By computing an entire partition before proceeding to the next makes these algorithms amenable to the cache hierarchy. However, in order to effectively use the cache these partitions need to fit the cache which they are targeting and ideally these partitions contain elements which communicate frequently with each other.

In Figure 2.6, we show two different partitionings for our four dataset examples. In the case of our Complete Graph (representative of DLA) both partitionings are equally as good because each partition contains an equal number of elements and these elements have the same number of edges between them. In the Structured Grid example, we preference the partitioning which keeps neighboring grid elements together because these grid points exchange information to each other, thus keeping them in the same partition insures that this information is exchanged more frequently than if they were not in the same partition. This is more apparent in the Scale-Free and Real-World examples where one set of partitions captures the community clusters and the other does not.

More formally, we can state a hierarchical partitioning over graph $G = (V, E)$ as follows: First, let the top level partition be $P^{(0)} = V$, then we can partition the vertices such that all partitions for a given level i cover their parent set, $P^{(i)} = \bigcup_j P_j^{(i+1)}$, and these partitions are non-overlapping, $\emptyset = \bigcap_j P_j^{(i+1)}$. If we want each partition at a particular level i to fit in a given cache of size s, then we would select are partitions such that $\forall_j |P^{(i)}| \leq s$. We can express this constraint for each level of the memory hierarchy.

Now, it is critical that we select our partitions in a way that captures tightly connected clusters. The elements in these cluster communicate more with each other than elements outside of their community. To express this, we need a graphical view our partitions, let $G_{jk}^{(i)} = (V_{jk}^{(i)}, E_{jk}^{(i)})$ which is the subgraph that connects vertices of $P_j^{(i)}$ to vertices of $P_k^{(i)}$ where $V_{jk}^{(i)} = V \cap (P_j^{(i)} \cup P_k^{(i)})$ and $E_{jk}^{(i)} = E \cap (P_j^{(i)} \times P_k^{(i)}))$. To express our goal that the partitions capture dense communities we can use the following formulation: $\forall_j |E_{jj}^{(i)}| \geq \sum_{k \neq j} |E_{jk}^{(i)}| + |E_{kj}^{(i)}|$. This states that we want the number of edges within the graph of a partition to be greater than the number of edges leaving that partition. Capturing these communities is the objective of partitioning, however it is up to the expert to determine these partitions within these constraints. If they do satisfy these constraints then the following steps in this section will insure to production of a high performance data access pattern.

The expert can use a graph partitioner to automatically determine these partitions [6]. However, this approach is computationally expensive. Ideally, this partitioning step is performed by the domain expert prior to the computation. The expert is typically involved in the collection of the dataset therefore the expert can use her domain knowledge at this step to determine the location of these partitions.

For the purposes of this chapter, we will assume that these partitions are obtained and stored hierarchically. We will also assume that this partitioned data is stored in a way which we can access the partitions. In the following code listing, we illustrate how a partition G_{jk} from the graph G:

```
G_jk = get_col_part( get_row_part( G, j ), k )
```

In this code snippet, obtaining the incoming edges into subgraph corresponds to gathering the column partitions of an adjacency matrix, and obtaining the outgoing edges corresponds to gathering the row partitions. Combining these two functions allow us to extract subgraphs from a larger graph. We will express our divide-and-conquer algorithms in terms of these extractions. If our partitions capture physically locality, then these algorithms will preserve them throughout the cache hierarchy.

2.3.2 A Nesting of Algorithms

In this section we show how a nesting of divide-and-conquer algorithms efficiently computes on the partitions of the previous subsection. These algorithms recursively divide the dataset until a base case is reached and directly computed on. For our purposes, the size and contents of these divisions is determined by the partitioning of the dataset. Thus by hierarchically capturing clusters, we can insure that computations within a cluster occur before computations outside said cluster, and this computation occurs within the cache.

If we continuing with our running matrix-vector and use the dataset and partitioning in Figure 2.6, then we can decompose it into the following nesting of algorithms:

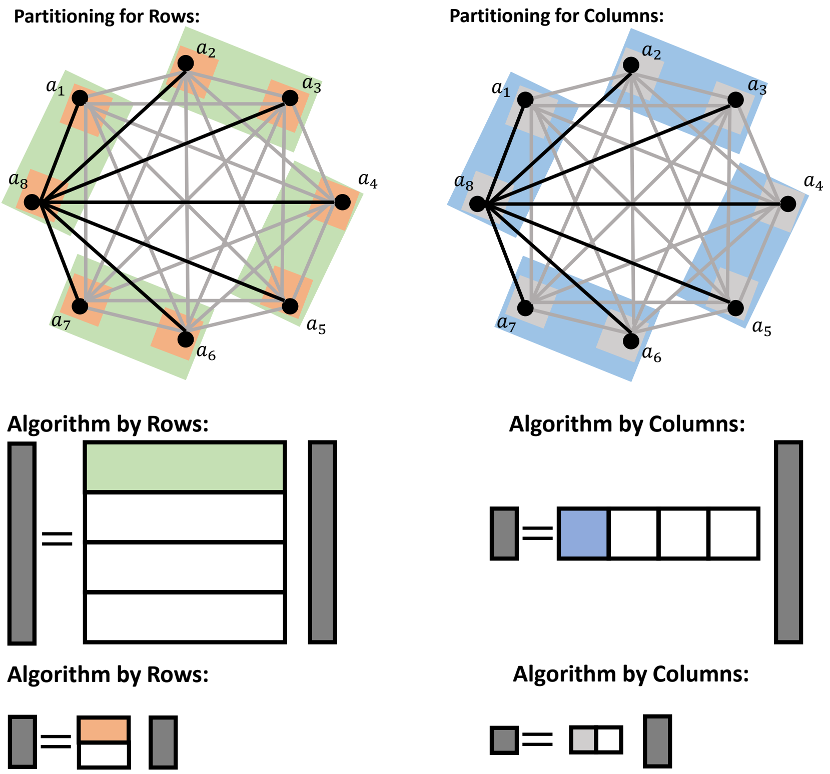


Figure 2.7: **Top:** we show the partitioning of the output vector y and the input vector x. **Bottom:** We show the application of divide-and-conquer algorithms to performance matrix-vector multiplication ($y = Ax$).

```
mv-row(y,A,x,
  mv-col(_,_,_,
    mv-row(_,_,_,
      mv-col(_,_,_,
          op(_,_,_ )))))
```

Here we perform the computation by alternating between an algorithm that computes by rows and one that computes by columns using the algorithms from Table 2.3. How the algorithms decompose the dataset is based on its partitioning (Figure 2.6). We illustrate this pictorially in Figure 2.7. Alternatively, we can express this nesting as the following series of C loops:

```
for(io=0 < 8; io+=2)      /* algo by rows */
  for(jo=0 < 8; jo+=2)    /* algo by cols */
    for(ii=0 < 2; ii++)   /* algo by rows */
      for(ji=0 < 2; ji++) /* algo by cols */
       y[idx_y(io,ii)] += A[idx_a(io,ii,jo,ji)] *
                          x[idx_x(jo,jj)];
```

We can clearly see that in the first two loops we divide the problem into sixteen 2 × 2 matrix-vector multiplies and the second set of loops computes those 2 × 2 multiplies. Additionally, note that indexing (idx_A, idx_y and idx_x) is described as a function of the loop variables. This is done to separate the logical indexing of the data from the physical indexing in memory. Ideally, this mapping is designed to insure contiguous memory access.

In Table 2.3, we show the algorithms we use for Matrix-Vector Multiplication, Iterative Matrix-Vector Multiplication and Matrix-Matrix Multiplication. We call the recursive application of these algorithms a *nesting* and this determines how the dataset is accessed during the computation. Ideally, we want the elements in the dataset stored in the order that they will be computed, so we will take this a step further in the next subsection use this nesting to rearrange the dataset in that desired order.

Coarse Grain Parallelism. As we indicated in Figure 2.4, modern systems provide various resources for coarse grain parallelism. We determine the division of computation across these resources through the selection of the same algorithms which have been decorated with parallel directives.

Algorithm	**Formula**	**Code**

`mv-row(y,A,x,op)`	$y_i = A_{i:}x$	`for(i=0;i<nrows;i++)`
		`  A_i = get_row_part(A,i)`
		`  y_i = get_row_part(y,i)`
		`  op(y_i,A_i,x)`

`mv-col(y,A,x,op)`	$y = A_{:j}x_j$	`for(j=0;j<ncols;j++)`
		`  A_j = get_col_part(A,j)`
		`  y_j = get_col_part(x,j)`
		`  op(y,A_j,x_j)`

`imv-it(y,A,x,T,op)`	$y = A^T x$	`r[0] = x`
		`for(t=0;t<T;t++)`
		`  op(r[t+1],A,r[t])`
		`y = r[T]`

`mm-row(C,A,B,op)`	$C_{i:} = A_{i:}B$	`for(i=0;i<nrows;i++)`
		`  C_i = get_row_part(C,i)`
		`  A_i = get_row_part(A,i)`
		`  op(C_i, A_i, B)`

`mm-col(C,A,B,op)`	$C_{:j} = AB_{:j}$	`for(j=0;i<ncols;j++)`
		`  C_j = get_col_part(C,j)`
		`  B_j = get_col_part(B,j)`
		`  op(C_j, A, B_j)`

`mm-acc(C,A,B,op)`	$C = \sum_p A_{:p}B_{p:}$	`for(p=0;p<ncom;p++)`
		`  A_p = get_col_part(A,p)`
		`  B_p = get_row_part(B,p)`
		`  op(C, A_p, B_p)`

Table 2.3: In this table we list the various divide-and-conquer algorithms used throughout this thesis.

The cost of communication and availability of the shared resources ultimately determine which algorithms get selected. For example, we preference partitioning by rows rather than columns because the latter requires accumulating to a shared resource (elements of vector y). This contention in turn requires locks which serialize access and potentially slow down computation if this access is in slower memory. Additionally, parallelism is not restricted to one level of the machine hierarchy. For example the authors in [7] provide a detailed treatment of nested parallelism.

Fine Grain Parallelism. In Figure 2.5 we illustrate a Single Instruction Multiple Data (SIMD) functional unit. These instructions allow fine grain parallelism at the microarchitectural level. In order to use them, we have to design our code to efficiently use these instructions. Thus, our inner-most algorithms are typically selected to permit the computation of multiple independent operations because communication is typically expensive within a SIMD vector. Using matrix-vector multiplication as our example, we preference the algorithm by rows over the columns for the inner most algorithm because it computes multiple independent operations that accumulate to different values of the output vector y.

2.3.3 Designing a Data Structure

The algorithm nesting in the previous subsection tells us how the dataset is accessed logically (as a mathematical object) but it does not prescribe how the dataset will be accessed physically (as elements in memory). However, how the dataset is stored physically and the format in which it is stored will greatly impact the overall performance of the implementation. This is because our divide-and-conquer algorithms make extensive use of partition access (i.e. `get_row_part` and `get_col_part`). If we have a hierarchically partitioned graph like Figure 2.6, then we want a data structure that insures low cost access to these partitions. To achieve this, we will select a hierarchical data structure the matches our algorithm nest and the format for each level of that data structure is determined by the structure of the original dataset. In Chapter 6 and Chapter 7, we will show how we select a data structure for scale-free and real-world datasets.

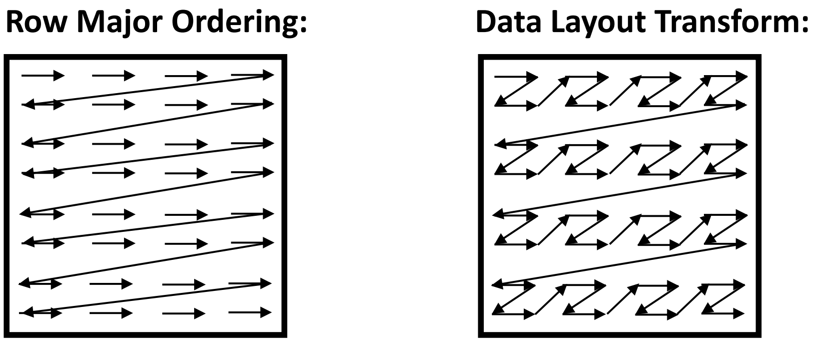


Figure 2.8: On the left we show a row major access pattern for a matrix and on the right we show the access pattern for our matrix-vector example. Ideally we want to arrange the matrix in memory in the access pattern on the right.

2.3.4 The Power of a Data Layout Transformation

Once the algorithm nesting is determined, we can go one step further by insuring that the dataset is access in unit stride. Doing this will maximize our cache utilization and reduce the number of TLB entries needed to address the working data set [2]. For example, a standard two dimensional C matrix is most efficiently accessed in row major ordering (Figure 2.8 left) however, our matrix-vector example uses a more complex access pattern (Figure 2.8 right). Continuing with matrix-vector example, we can repack the input dataset `A` into `A_dlt` where each element is stored in the other that it will be accessed:

```
p=0;
for(io=0 < 8; io+=2)
  for(jo=0 < 8; jo+=2)
    for(ii=0 < 2; ii++)
      for(ji=0 < 2; ji++)
        A_DLT[p++] =
          A[idx_a(io,ii,jo,ji)];
```

Once our data is repacked, we can implement our algorithm nesting using `A_dlt`:

```
#define contig(io,ii,jo,ji) (p++)
for(io=0 < 8; io+=2)      /* algo by rows */
  for(jo=0 < 8; jo+=2)    /* algo by cols */
    for(ii=0 < 2; ii++)   /* algo by rows */
      for(ji=0 < 2; ji++) /* algo by cols */
       y[idx_y(io,ii)] +=
        A_dlt[contig(io,ii,jo,ji)] * /* Contiguous */
         x[idx_x(jo,jj)];
```

We can see that each element of A_dlt is accessed in the contiguous order illustrated on the right side of Figure 2.8. By performing this repacking, we can insure that the kernel we generate will efficiently access the dataset.

2.4 A Method for Fast Kernels

In the previous section, we outlined the process for constructing on efficient access pattern for matrix-like operations. In this section we discuss the second part of our method – how to construct an efficient kernel for this access pattern. We do this by extract the inner most loops of our access pattern which we will call the kernel. Using our matrix-vector multiply example we would separate the outer two loops from the inner two loops to form the access pattern and kernel:

```
access_patern_mv(A,x,y)
  for(io=0 < 8; io+=2)
    for(jo=0 < 8; jo+=2)
      kernel(A,x,y,io,jo);

kernel(A,x,y)
  for(ii=0 < 2; ii++)
    for(ji=0 < 2; ji++)
      y[idx_y(io,ii)] +=
       A[contig(io,ii,jo,ji)] *
       x[idx_x(jo,jj)];
```

Then, we find an efficient mix of instructions that implements this kernel. After that, we statically schedule these instructions. Last, we optimize and emit these instructions as code. In the rest of this section we detail our generic process for generating kernels.

2.4.1 Selecting an Instruction Mapping

In Figure 2.5 we illustrate a SIMD functional unit and in order to produce a high performance implementation we need to efficiently utilize these units. Given a kernel we can find a myriad of instruction mixes – or combinations of SIMD instructions – that implement the kernel. Continuing with our 2×2 matrix-vector multiplication kernel, we can construct two different instructions mixes for the same operation. In this first example, we store the y vector in a SIMD register and broadcast each element of x into their own register. We will call this the broadcast-based instruction mix.

```
x0vec = vbroadcast(x[0])
x1vec = vbroadcast(x[1])
A0vec = vload(A[:][0])
A1vec = vload(A[:][1])
r0vec = vfma(A0vec, x0vec, 0)
r1vec = vfma(A1vec, x1vec, r0vec)
vstore(y,r1vec)
```

In the next example, we load rows of A into SIMD registers that are multiplied by a SIMD register of x elements, and the resulting vectors are accumulated within the register and stored into y. We will call this the accumulation-based instruction mix.

```
xvec  = vload(x)
A0vec = vload(A[0][:])
A1vec = vload(A[1][:])
r0vec = vmul( A0vec, xvec )
r1vec = vmul( A1vec, xvec )
y0sca = hadd( r0vec )
y1sca = hadd( r1vec )
sstore( y[0], y0sca )
sstore( y[1], y1sca )
```

Both of these instruction mixes will implement the desired matrix-vector kernel. However, we want to select the most efficient kernel. In particular we want to select a mix that sustains a high rate of computation because many of these kernel operations will be performed inside a nesting of loops. In this situation, the individual performance of a kernel is less important than the steady-state performance.

Selecting a High-Throughput Instruction Mix. Our goal is to find the instruction mix that, on average, takes the fewest number of cycles to pass through the microarchitecture. Because we are only interested in the instruction mix performance inside a much larger loop in steady-state, we can use Little's Law [8] to model the microarchitecture. Little's law tells us that the average waiting time of a queue in steady-state, W, is equal to the average number of elements in the system N divided by the average arrival rate of elements into the system, λ. We can express this as follows:

$$W = \frac{N}{\lambda} \tag{2.1}$$

Because the system is in steady-state, the arrival rate of elements into the system is equal to the average departure rate out of the system. We can use this formula to estimate the average amount of time it takes the microarchitecture to clear a given instruction mix in steady-state. We are not concerned if those instructions came from the same iteration, but rather we want to know if that particular mix of instructions has moved through the processor, this allows us to ignore instruction dependencies for this computation.

To illustrate this, let us assume that we only have one type of functional unit that computes at a rate of two instructions per cycle ($\lambda = 2$), and let us select the broadcast based matrix-vector multiply instruction mix from the beginning of this subsection which has $N = 7$ instructions. By modeling this system as a queue we can apply Little's Law, so the average waiting time of the instructions in this mix would then be 3.5 cycles.

However, if we want to model a more complex microarchitecture, then we can adjust our calculations. If we have multiple functional units, then we are concerned with the functional unit that is the bottleneck. We want to know which functional unit, queue, has the longest average waiting time. For example, let us assume that we have two functional units, a memory functional unit and a arithmetic functional unit. We assign them their own average waiting time $W_{\texttt{mem}}$ and $W_{\texttt{alu}}$, and their own average arrival rate $\lambda_{\texttt{mem}}$ and $\lambda_{\texttt{alu}}$. Let us also assume that our microarchitecture has two memory functional units and one arithmetic functional unit, that will give us $\lambda_{\texttt{mem}} = 2$ and $\lambda_{\texttt{alu}} = 1$. If we take the broadcast based matrix-vector kernel, we can divide its instructions into two queues with $N_{\texttt{mem}} = 5$ and $N_{\texttt{alu}} = 2$. This would result in the waiting times of $W_{\texttt{mem}} = 2.5$ and $W_{\texttt{alu}} = 2$. This means that on this microarchitecture the broadcast based implementation has a waiting time of 2.5 cycles. We can apply the exact same process to the second instruction mix, the accumulation-based mix, for the matrix-vector kernel, where we have $N_{\texttt{mem}} = 5$ and $N_{\texttt{alu}} = 4$. This mix will give us a maximum average waiting time of 4 cycles. Thus, for this hypothetical architecture, the first instruction-mix will sustain a higher throughput relative to the second one. This process only estimates which mix will have the lowest waiting time on average in the steady-state. Once we select this mix, we will need to schedule the instructions in order to sustain the estimated performance.

2.4.2 Instruction Scheduling is Important

Once an instruction mix is selected, we need to statically schedule the instructions in order to sustain the rate of computation enabled by the mix. Typically, instruction take more than a single cycle to compute, and if we do not overlap instructions then during those cycles the processor is stalled. Thus, we prevent these stall cycles by overlapping independent instructions during those stalled cycles. While most modern architectures are out-of-order, meaning they can rearrange instruction on the fly as resources become available, we show in [9] that at high throughput an out-of-order processor benefits from statically scheduled instructions. For our purposes, we use software pipeline scheduling [10] which is a systematic method for scheduling loops by interleaving instructions between different iterations in order to hide instruction latency.

2.4.3 Further Optimizations and Code Generation

The last step of kernel generation process involves performing compiler optimizations on our scheduled instruction mix. For example, we fully unrolling the loops in the kernel which eliminates branching and simplifies index computation. Additionally, we include transformations such as common subexpression and array scalarization where arrays are replaced with scalar registers. Lastly, in Chapter 6 we show how we can specialize kernels to specific patterns and shapes that appear in the dataset. Once the code is optimized we generate it using ANSI C with inline assembly intrinsics that preserve instruction order. We detail this in [9].

2.5 Chapter Summary

In this chapter, we outlined our two part method for producing high performance implementations of matrix-like operations. This process is a generalization of the Goto approach for matrix-matrix multiplication, and works by yielding an efficient access pattern that feeds a tuned kernel. We also detailed how we use problem structure to generalize this approach to domains outside of dense linear algebra, specifically to operations that are expressible as matrix-like operations over arbitrary semirings. In the following chapters we will highlight key features in this method and show that it is extensible to various domains such as structured mesh computations, sparse computations and graph analytics.

Chapter 3

Related Work

3.1 Introduction

In this chapter, we describe the overarching story of the development of high performance computational libraries for the domains targeted in this thesis. We will also show how these domains relate to one another and lay the groundwork for our approach. Additionally, we will discuss where this thesis fits in this story.

Before beginning, we will provide a brief overview of what follows. We will start with the development of high performance Dense Linear Algebra (DLA) libraries. The advances in this field proceeded in lock step with the development of computer hardware, responding to the divergence in processor and main memory speed, then the advancement of caches and eventually with the addition of short vector instructions. These refinements have led to the development of modern Linear Algebra libraries that cast the bulk of their work in terms of the Matrix-Matrix Multiplication operation, which in turn is built on divide and conquer algorithms that partition their inputs into cache size blocks that are eventually computed on hand tuned kernels. Thus the bulk of DLA performance rests on the shoulders of expert coders who hand tune Matrix-Matrix Multiply kernels. In this thesis we provide a systematic approach to generating these kernels.

While the machinery for DLA achieves high performance, it is not necessarily suited to problems over structured matrices such as those which arise from solving differential equations over discretized objects, i.e. when *Finite Difference Methods* are employed. In this domain, the very regular structure the arises from discretization of physical objects is exploited in the selection and implementation of algorithms for solving these problems. What results is a layering of loops – which partition the problem in both spacial dimensions and in the temporal dimension – over a kernel which acts as a *stencil* over these discretized points. Like Matrix-Matrix Multiply kernels we also provide a mechanical approach to generating high performance stencil kernels.

The previous two domains deal with extremely regular problems. However, half of this thesis targets graph analytics over seemingly unstructured real-world problems. To bridge the previous work to the graph domain, we discuss the work done by Kim et. al. in [11] where they show that high performance for *Finite Element Methods* (FEM) is achievable without graph partitioning. They do this by retaining the structural information of the input domain and using this to determine how elements are stored in a hierarchical data structure. In this thesis, we leverage this idea for real-world graphs with underlying structure.

The current state of the graph domain can be captured in two different approaches. The first is a dataflow approach which expresses graph operations as functions over edges and vertices that are applied when their inputs are modified an continue until a convergence criteria is met. Overhead is the limiting factor in this approach. The second approach, cast graph operations in terms of Linear Algebra over operation specific semirings. Much like DLA, in this approach performance is dependent on the underlying building blocks which we will address in this thesis.

3.2 Related Work

In this section, we elaborate on the historical development of Matrix-Matrix Multiplication, Stencil computations, Sparse Matrix computations and graph analytics.

Matrix-Matrix Multiplication. In the field of Dense Linear Algebra (DLA), Matrix Multiplication is the key operation on which most of the BLAS-3 [12] and LAPACK routines [13]) are built [14, 15].

As long as this operation is efficient on a given target architecture, then all other BLAS-3 and LAPACK operations that cast the bulk of their computation on it could leverage this performance. It was recognized that one could achieve high performance for matrix-matrix multiplication by taking advantage of its high operational intensity and exploiting algorithmic optimizations such as blocking to achieve a high level of performance.

In PhiPAC [16], the authors took this blocked approach and parameterized the various optimizations over the operations so search – or auto-tuning – could determine their values. This auto-tuning approach was extended by ATLAS [17] which decoupled the compute kernel from the bulk of the data access by packing the working data-set into buffers for a tuned kernel to compute on. The authors of [18] identified that many of the search parameters can be determined analytically without search.

The GotoBLAS [2] provided a key insight regarding blocking that if the blocking strategy can guarantee that the working data-set can be brought to the registers at the rate necessary for peak performance then given an efficient kernel peak performance can be practically achieved. The authors further identified that blocking for the L2 cache is sufficient on modern machines and that performing a data layout transformation simplifies the construction of a high performance kernel. The BLIS [19] project extended the GotoBLAS by decomposing via blocking into an even smaller micro-kernel. Once the outer blocking parameters are determined [20] tuning this kernel to a given architecture is all that is necessary for high performance. In [21], we automate this last leg of performance for Matrix-Multiplication by distilling the kernel down even further and automating its generation. In this thesis, we discuss how we generate these high performance kernels.

It is worth noting that a great deal of work has focused on the generation of cache resident small matrix kernels such as LGen [22, 23, 24] and the Built-to-Order BLAS [25]. For many scientific and engineering application there is a need for linear algebra operations on cache resident data which these kernel code generators solve. However our focus is on problems larger than the cache.

Stencils and Structured Mesh. Many applications, which can be expressed in terms of Linear Algebra, give rise to systems of equations with very specific structures. For example in solving Partial Differential Equations (PDE) using Finite Difference Methods (FDM) the resulting system of equations for a 3 dimensional object can be expressed using a tridiagonal matrix. This type of matrix is far from dense which makes DLA libraries inefficient for solving these systems of equations. These problems are extremely regular and can be expressed as the successive application of function over every point, where the function computes the value of each point from its neighbors.

This function is called a *Stencil* because it applies a repeated pattern over a collection of elements. Stencil Computations are not restricted to PDEs, in fact they appear in convolution codes in signal and image processing, in simulations for cellular automata, weather and earthquakes and in any domain where the computation can be described as a regular computation over neighboring elements. Because of their ubiquity and importance in scientific computing, the compiler community has focused a great deal of attention on optimizing them.

In the early 1980s compilers were designed specifically for compiling stencil kernels like the [26, 27, 28] . These early compilers split the stencil computation into two parts: A microcode compute kernel, which was fed to the sequencer of the Connection Machine, and a layering of outer loops that dealt with data movement which fed the appropriate elements to the kernel. These stencil compilers achieved performance by focus on the reuse of elements across multiple stencils (multistencils) and minimizing communication across nodes.

As caches became more common, new stencil designs focused on the explicit use of cache locality through tiling. The authors in [29] identified that explicit cache tiling is not necessary for one and two dimensional stencils because even a small cache is sufficient for reuse. However, as the number of dimensions increase, explicit cache tiling becomes necessary for performance. In their work the provide a strategy for applying tiling to three dimensional stencil codes, along with an analytical approach to determining tile sizes. In [30] the authors identify the limitations of spacial tiling alone and provide strategies for prefetching using a model based approach.

The issue is that only blocking in the spacial dimensions of a stencil ignores the amount of reuse that can be obtained by also blocking in the time dimension. In the paper [31] , the authors identified that loop skewing alone is not sufficient to improve locality in stencils, but when combined with loop interchange and data forwarding it becomes a more powerful transformation called *time skewing* or *time tiling*. The idea is that the stencil dimensions can be rearranged to allow for blocking in the time dimension. This is made more powerful by forwarding the result of one iteration to the next, without making a round trip to memory. This approach is automated in the Panorama compiler [32] for a broad class of imperfectly nested stencil problems. In [33] the authors extend time tiling to modern multiprocessors with deep memory hierarchies. In their treatment of time tiling they provide a systematic approach for determining tile sizes.

In [34] the authors put these techniques to practice and develop a temporal blocking stencil implementation for the Gauss-Seidel algorithm for multigrid solvers. The idea is that cache sized multistencils are applied iteratively over the same block before proceeding to the next block. This insures that the computation of the multistencil occurs entirely within cache after the first iteration, as opposed to being loaded from main memory before each iteration. This is significant development in stencil computations because now stencil performance is no longer dependent on the bandwidth between main memory and the processor, but on the size of the cache and the ability of an expert programmer to produce an efficient kernel.

Spacial and temporal blocking are at the core of high performance stencil computation, however their application on modern multiprocessors and heterogeneous systems is non-trivial. Thus, much of the current work has focused on systematically extending these optimizations in a mechanical fashion for a broad class of stencils. In [35] the authors show that these techniques work on a wide variety of computer architectures ranging from modern microarchitectures, to Very Large Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) processors. The authors then extend this work in [36] by creating an autotuning framework that implements stencils that are expressed in a FORTRAN-like descriptor language. Autotuning is employed to find the optimal spacial and temporal tiling for the target system. Similarly, the Parallel Autotuned Stencils (PATUS) framework [37] also automatic tunes and generates stencils from a specification, but in a C rather than FORTRAN. The Porchoir framework [38] produces stencil code from a C++ specification, but instead of using autotuning, it relies on cache oblivious algorithms for performance. Rather than using cache oblivious approach or autotuning, the PLuTo framework [39] casts the problem of optimal tiling for parallelization of stencils in terms of the polyhedral model then solves for those parameters. In [40] we address the issue of generating high performance SIMD stencil code by separating the problem into three separate concerns optimizing loop tiling for cache locality, optimizing tiling for parallelism and generating high performance SIMD stencil kernels. The work described in this thesis focuses on the third part. In [41] we take this a step further by incorporating a data layout transformation.

Sparse Matrix Multiplication. In the same way that stencils arise as a specialization of Linear Algebra operations over matrices with specific structures, many applications such as Finite Element Methods (FEM) result in system of equations that lead to unstructured sparse matrices. These matrices are then efficiently solved using sparse specific operations such as Sparse Matrix-Vector Multiplication (spMV), Sparse Triangular Solve (spTS) and Sparse Matrix-Matrix Multiplication (spMM). In [42] the author provides an excellent survey of the applications where sparse matrices arise, computations over sparse matrices and early storage formats formats for sparse matrices.

Following in the footsteps of DLA, many sparse APIs were built around a common core of routine over a variety of formats [43, 44]. Unlike DLA, sparse Linear Algebra libraries must deal with a variety of formats to deal with the different sparsity patterns that arise in real world problems. This exacerbates the problem of producing high performance libraries in this domain, as each format would require a tuned implementation for each architecture. This would be an enormous undertaking if done by hand, which is why a great deal of research has focused on the automatic generation of fast sparse libraries.

In [45, 46] the authors create a template based library for implementing sparse operations over an arbitrary datatype which they call the Woodenman API. They combine this approach with a restructuring compiler which converts the matrix code into an optimized interface called the Ironman API.

The SPARSITY framework [47] combined cache and register blocking along with autotuning in order to automatically generate tuned Sparse Matrix-Vector and Sparse Matrix-Matrix Operations. They demonstrated that these techniques work for a wide variety of architectures and reduce the amount of expert itme needed to produce fast kernels. The Optimized Sparse Kernel Interface (OSKI) [48] took SPARSITY a step further by exposing an autotuning API along with providing memory hierarchy aware kernels to match modern architectures. The benefit of this level of performance abstractions is that it allows library users to pass domain knowledge to these building blocks and reap the performance benefit while minimizing the amount of expensive autotuning. More recently, the authors in [49] take these techniques to manycore accelerators, showing that these transformations work even on the extreme end of the hardware spectrum.

Up until this point, the bulk of the research in sparse computations has focused on problems arising from engineering and the hard sciences. However, a growing body of research is focusing on sparse problems arising from the social sciences. These problems tend to be larger and more unstructured than their engineering counterparts. These real world networks tend to fall in the category of scale-free graphs [50], where their degree distribution follow an inverse exponential. This means that a few key elements in the graph carry the bulk of the edges, while the vast majority of nodes contain only a few. This structural departure means that many of the blocking and tiling techniques that worked for engineering-type networks no longer fit as naturally. As a result, we are seeing the development of sparse libraries for these type of real-world networks. For example, the Compressed Sparse Block (CSB) [51] assumes that little reuse will occur over these graphs and therefore minimizes data access by compressing the indices of the sparse elements and laying out the data efficiently in memory. In [52] the authors take a different approach by incorporating the problem of graph partitioning along with cache-oblivious storage. The authors recognize that locality is necessary in order to take advantage of cache blocking.

In [53], we also combine locality with storage by creating a hierarchical sparse data structure for storing scale-free graphs which we call Recursiv Matrix Vectors. The idea is that even in real world scale-free data there is a hierarchical cluster structure and if we can capture that in a tree like format then we can maximize cache reuse within these clusters. In this thesis, we show how to implement a high performance SPMV around our storage format. This format borrows ideas from FLASH [54], a hierarchical dense matrix format. We will show how to implement a high performance graph analytic library on this format.

Graph Computation. The graph computation story is rapidly unfolding and we can classify the emerging work into two distinct, but not necessarily incompatible approaches: the data flow approach and the linear algebra-like approach.

In the data flow approach, the graph frameworks decouple the computation from the data access pattern. Graph operations are implemented in terms of small atomic functions over vertices and edges, and these functions are executed when their inputs have been modified. This process either occurs as bulk computations or as the new data is made available, and this continues until no more vertices or edges are modified. Prime examples of this approach include Pregel [55], GraphLAB [56], Ligra [57]. There are a myriad of optimizations to this approach, for example GraphChi [58], X-

Stream [59] reshape the graph to take advantage of the memory hierarchy. A more aggressive optimization is seen in Galois [60] and its extension in Elixir [61] introduce the notion of speculation by requiring the atomic functions be invertible. This allows the framework to run-ahead on computation on the assumption that more work can be done on a particular path. If this is later deemed an incorrect assumption, then the computation can be rolled back.

For the linear algebra approach to graph analytics we have the Combinatorial BLAS [62] and Knowledge Discovery Toolbox represent graph operations as matrix operations over specialized semirings which capture the desired graph computation. These libraries leverage many of the approaches used for sparse matrix computations on distributed memory systems and reduce the burden on the programmer. This approach is made into an API in the GraphBLAS project [5] through the use of C++ templates. Algorithms in this interface are implemented as iterative sparse matrix vector products on user defined semirings. The goal of the GraphBLAS is to define an interface for common graph analytic routines that can be tuned to the target system. It is worth noting that most real-world networks are not static, but evolve over time. Therefore, the Spatio-Temporal Interaction Network and Graph Extensible Representation (STINGER) [63] is a graph library and data format for performing analytics on time varying graphs. They use a specialized data structure to accommodate frequent updates to the graph, and to facilitate parallel operations over the graph. Their interface allows the user to provide a mapping function between physical vertex ID in the graph and their logical ID in memory.

In this thesis we implement a library in the style of the GraphBLAS using the techniques and transformations from our Matrix-Matrix Multiplication, Stencil and SPMV implementations. The key idea that allows us to link the work in the three previous areas to graph analytics comes from [11]. In this paper the authors created a solver for hp-Adaptive FEM that takes advantage of structural information from the original application to keep neighboring data nearby in memory. They use a hierarchical storage – Unassembled Hyper-Matrices (UHM) – to store each refinement of a node as a child of the parent. In this way they use structural information to preserve locality which naturally allows for an efficient parallel implementation in [64]. We use this idea by developing a data structure that preserves graph locality in a hierarchical fashion. This allows us to take advantage of the cache hierarchy by using a similarly layering approach seen in Matrix-Matrix multiplication which in turn allows us to use high performance kernels.

3.3 Chapter Summary

In this chapter, we detailed the development of Matrix-Matrix Multiplication kernels, Stencil Computation, Sparse Matrix Multiplication and Graph Analytics. Dense Matrix-Matrix Multiplication kernels form the computational basis for high performance Dense Linear Algebra Operations (DLA). We can view stencil and sparse computations as specializations of DLA, which take advantage of structure and sparsity for performance. Further, we can cast graph operations in terms of linear algebra and take advantage of these specializations. The remainder of this thesis is devoted to the generation of efficient kernels in these fields, along with the implementation of high performance sparse matrix and graph libraries that take advantage of these types of kernels.

Part I

Dense and Structured Applications

Chapter 4

Matrix-Matrix Multiplication as Template for Performance

4.1 Introduction

Throughout this thesis, we discuss a systematic approach for producing high performance implementations for matrix-like operations. This approach divides the problem into an algorithmic portion that feeds an efficient kernel portion. Our approach is a generalization of the approach used to produce high performance matrix-matrix multiply code for dense problems, which is why we use it as a our template. We will discuss the prior work behind the algorithmic side of the problem which includes a provably high performance algorithmic nesting that in turn leads to an efficient access pattern and data layout transformation. We will also discuss our contribution to this process by automating the generation of efficient dense kernel.

Around the turn of the century, Kazushige Goto revolutionalized the way matrix-matrix multiplication is implemented with the GotoBLAS approach [2]. Specifically, Goto demonstrated that data movement and computation for computing gemm can be *systematically* orchestrated in a specific manner, as depicted in Figure 4.1, that does not change as we shift between architectures. In Goto's implementations, a small *macro-kernel* needs to be custom-implemented. The BLIS framework [1] refactored Goto's algorithm exposing additional loops so that only a much smaller micro-kernel needs to be customized for a new architecture, with all the other parts implemented in the C programming language.

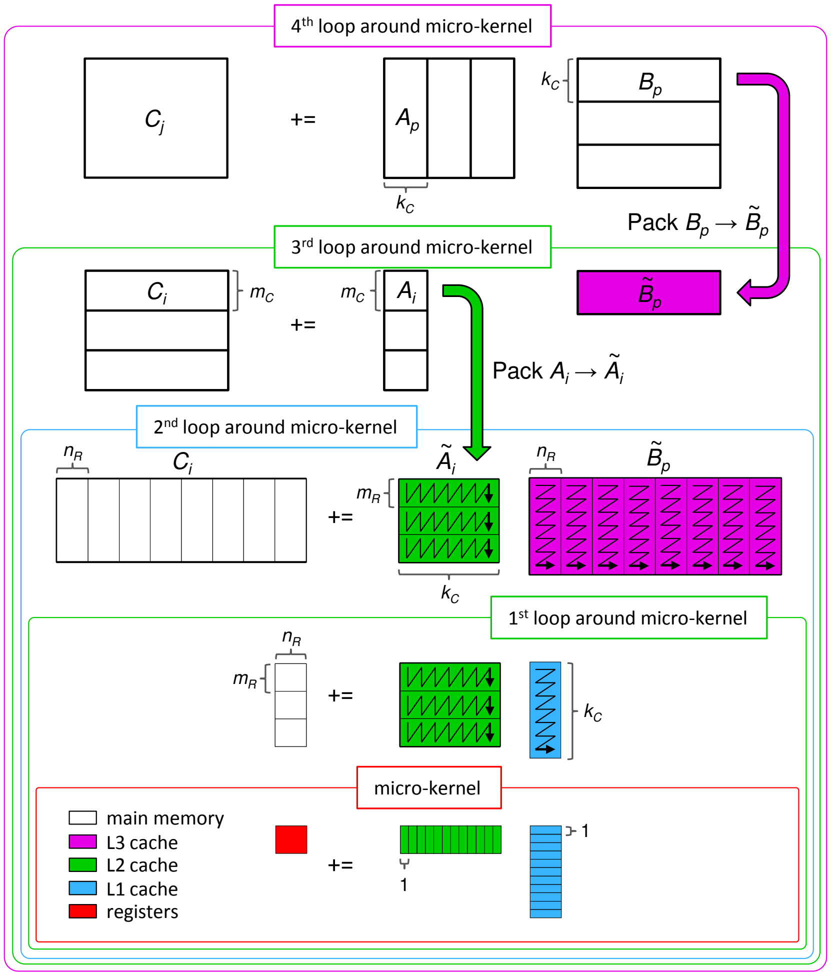


Figure 4.1: (Figure from [1]) The GotoBLAS approach for matrix multiplication as refactored in BLIS. Blocked arrows represent explicit data packing, and thin arrows represent the data layout in after packing.

In isolation this chapter provides a key contribution to the field of Dense Linear Algebra – the generation of a high performance gemm kernel. These efficient gemm kernels provide the foundation for high performance scientific codes which is why substantial effort is placed on improving the performance of these kernels on new hardware. Our contribution reduces this gemm kernel into an even smaller building block and provides a mechanical process for generating these blocks. As a part of this thesis, this chapter emphasizes the method for performance described in Chapter 2 and demonstrates how to make the kernel generation phase mechanical. In later chapters, this approach will allows us produce high performance implementations of operations for structured mesh, spares linear algebra and graph analytics.

4.2 Anatomy of a Gemm Micro-Kernel

BLIS is a framework for rapidly instantiating the BLAS using the GotoBLAS (now maintained as OpenBLAS[65]) approach [2], and it is one of the most efficient expert-tuned implementations of the BLAS. The GotoBLAS approach performs loop tiling [66] and packing of data for different layers [67] within the cache hierarchy in a specific manner to expose an inner kernel. The specific loop tiling strategy in GotoBLAS has been shown to work well on many modern CPU architectures. BLIS extends these ideas, and tiles the loops in the GotoBLAS inner kernel to expose an even smaller Gemm kernel, the *micro-kernel*, and showed that high performance is attained when this micro-kernel is optimized [68].

4.2.1 The Micro-Kernel

The micro-kernel is a small matrix multiplication that implements $C += AB$, where C is a $m_r \times n_r$ matrix, while A and B are micro-panels of size $m_r \times k_c$ and $k_c \times n_r$ respectively. In addition, because of the packing of A and B prior to the invocation of the micro-kernel, it can be assumed that A is stored in a contiguous block of memory in column-major order while B is contiguously stored in row-major order. Since the micro-kernel is a small gemm kernel, the micro-kernel can be described, using compiler terminology, as a gemm kernel computed using a triply-nested loop of the KIJ or KJI variant.

Within the BLIS framework, it can also be assumed that $m_r, n_r \gg k_c$. In addition, we assume that the bounds of the loops (i.e. k_c, m_r, and n_r) are determined analytically using the models from [20].

Computing the micro-kernel. Mathematically, the micro-kernel is computed by first partitioning A into columns and B into rows. The output C is then computed in the following manner:

$$\begin{aligned} C &\mathrel{+}= \left(a_0 \,\middle|\, \dots \,\middle|\, a_{k_c-1} \right) \left(\begin{array}{c} b_0^T \\ \hline \vdots \\ \hline b_{k_c-1}^T \end{array} \right) \\ &\mathrel{+}= \sum_{i=0}^{k_c-1} a_i b_i^T, \end{aligned}$$

where the fundamental computation is now

$$C \mathrel{+}= a_i b_i^T,$$

a single *outer-product*, and our task is to compute the outer-product multiple times, each time with a new column and row from A and B, in as efficient a manner as possible.

4.2.2 Decomposing the Outer-Product

Focusing on a single outer-product, $C+ = ab^T$, we can decompose the outer-product further by performing loop tiling of C. This, in turn, will require us to block the columns of A and rows of B into sub-columns and sub-rows of conformal lengths respectively, as follows:

$$C \rightarrow \left(\begin{array}{c|c|c} C^{0,0} & C^{0,1} & \dots \\ \hline C^{1,0} & C^{1,1} & \dots \\ \hline \vdots & \vdots & \ddots \end{array} \right) a \rightarrow \left(\begin{array}{c} a^0 \\ \hline a^1 \\ \hline \vdots \end{array} \right) b \rightarrow \left(\begin{array}{c} b^0 \\ \hline b^1 \\ \hline \vdots \end{array} \right)$$

In this case, the outer-product is decomposed into a smaller unit of computation, which we will term as *Unit Update*, which computes

$$C^{i,j} + = (a^i)^T b^j,$$

where $C^{i,j}$ is now a $m_v \times n_u$ matrix, and the subvectors of a^i and b^j are vectors of length m_v and n_u. We can view this operation in the following pseudo-code snippet:

```
unit_update( C, a, b )
  for( i = 0; i < m_v; ++i )
    for( j = 0; j < n_u; ++j )
      C[i][j] += a[i] * b[j];
```

Decomposing the outer-product into smaller unit updates allows us to determine how to compute the outer-product with the available instructions on the targeted architectures. In the case where $m_v = n_u = 1$, each unit update computes a single element in the matrix. This means that the outer-product, C, is computed element-wise, can be performed using the following code:

```
for( i = 0; i < m_r / m_v; ++i )
  for( j = 0; j < n_r / n_u; ++j )
    a_i  = sub_vector( a[i*m_v : (i+1)*m_v] );
    b_j  = sub_vector( b[j*n_u : (j+1)*n_u] );
    C_ij = sub_block( C[i*m_v:(i+1)*m_v][j*n_u:(j+1)*n_u] );
    unit_update( C_ij, a_i, b_j );
```

Where b^j is streamed from the L1 cache, and a^i is loaded into the registers from the L2 cache. Alternatively, interchanging the two loops yields,

```
for( j = 0; j < n_r / n_u; ++j )
  for( i = 0; i < m_r / m_v; ++i )
    a_i  = sub_vector( a[i*m_v : (i+1)*m_v] );
    b_j  = sub_vector( b[j*n_u : (j+1)*n_u] );
    C_ij = sub_block( C[i*m_v:(i+1)*m_v][j*n_u:(j+1)*n_u] );
    unit_update( C_ij, a_i, b_j );
```

Where each iteration of the inner-most loop requires new values of a^i to be loaded from the L2 cache. In either case, we can replace the micro-kernel block in Figure 4.1 with the new diagram in Figure 4.2.

The interesting case is when $m_v \neq 1$ and/or $n_u \neq 1$. In this case, each unit update is a smaller outer-product. By selecting appropriate values of m_v and n_u, we gain the flexibility of mapping the computation of the unit update ($C^{i,j}$) to the availability of vector / single-instruction-multiple data (SIMD)

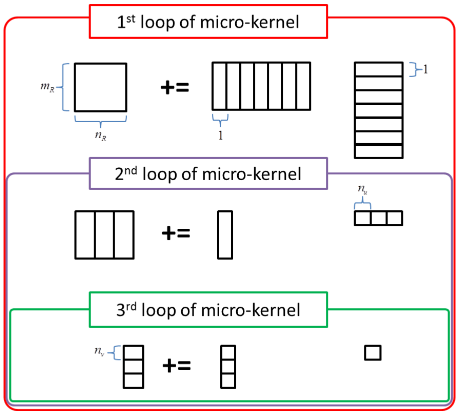


Figure 4.2: An additional three loops are introduced after decomposing the BLIS micro-kernel into smaller outer-product kernels of size $m_v \times n_u$. These set of loops would replace the micro-kernel shown in Figure 4.1.

instructions available on modern architecture. This flexibility of mapping also yields us a family of algorithms that compute the outer-product, which is the kernel within the micro-kernel we are trying to optimized.

4.3 Identifying Outer-Product Kernels

Recall that the micro-kernel is essentially a loop around multiple outer-products. In addition, each outer-product can be further decomposed into smaller unit updates of size $m_v \times n_u$. In this section, we discuss how the values of m_v and n_u are determined by the instructions available on the desired architecture.

4.3.1 The Building Blocks: SIMD Instructions

The key to high performance is to use Single Instruction Multiple Data (SIMD) vector instructions available on many of the modern processors. We assume that all computation involves double precision arithmetic and that each vector register can store v double precision floating point numbers, where v is a power of two. In addition, we assume that the following classes of vector instructions are available:

1. Vector **Stores.** Vector store instructions write all v elements of a vector register to memory.
2. Vector **Load.** Load instructions read u unique elements of data from memory, where $u \leq v$ and u is a power of two. An element is considered unique if it resides in a unique memory address. In cases where $u < v$, each of the u unique elements are duplicated v/u times.

   We assume that all elements loaded by a single Load instruction are within v memory addresses of each other[1]. Prefetches are considered Load instructions.
3. Vector **Shuffles.** Shuffle Instructions reorders and/or duplicates the elements in a vector register. We restrict ourselves to only instructions that be represented by a $n_v \times n_v$ matrix where each row contains exactly $n_v - 1$ *zeros* and a single *one*, but each column may contain multiple *ones*. In addition, we assume that the number of *ones* in each column is a power of two.
4. Vector **Computation.** It is assumed instructions that perform computation on vector registers are element-wise operations. This means that, given vector registers, `reg_a` and `reg_b`, the output is of the form:

   $$\texttt{reg\_a}\,\mathrm{op}\,\texttt{reg\_b} = \left(\begin{array}{ccc} \alpha_0 & op_0 & \beta_0 \\ \alpha_1 & op_1 & \beta_1 \\ & \vdots & \\ \alpha_{v-1} & op_{v-1} & \beta_{v-1} \end{array}\right),$$

   where op_i and $op_j, i \neq j$ may be different binary operators. The result of the computation may be stored in one of the input registers, or a third vector register.
5. **Composite Instructions.** Some instructions – we will call them Composite Instructions – can be viewed as a combination of some of the previous three types of instruction. For example, the instruction,

   ```
   vfmadd231pd, reg, reg, mem1to8
   ```

   instruction on the Xeon Phi can be expressed as a Load instruction, followed by a broadcast (Shuffle) instruction, followed by a fused multiply-add (Computation) instruction.

[1] This implies that our framework do not handle vector gather instructions. However, these Gather/Scatter instructions are not required in dense linear algebra kernels, as matrices are often packed for locality.

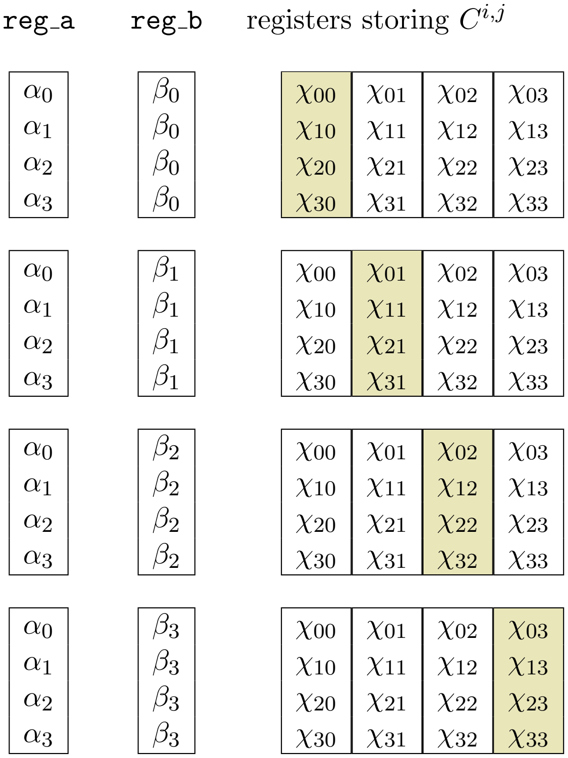

reg_a
reg_b
registers storing $C^{i,j}$
α_0 α_1 α_2 α_3
β_0 β_0 β_0 β_0
β_1 β_1 β_1 β_1
β_2 β_2 β_2 β_2
β_3 β_3 β_3 β_3
χ_{00} χ_{01} χ_{02} χ_{03}
χ_{10} χ_{11} χ_{12} χ_{13}
χ_{20} χ_{21} χ_{22} χ_{23}
χ_{30} χ_{31} χ_{32} χ_{33}

Figure 4.3: SIMD computation of a 4 × 4 outer-product using four unit updates of size 4 × 1. Shaded register represents the register that is being updated during the particular stage of computation.

4.3.2 Mapping Unit Updates to SIMD Instructions

Given the available SIMD instructions, one possible size of a single unit update is for $m_v = v$ and $n_u = 1$, where v is the size of a SIMD register. This means that v values from a, and a single value of b is loaded into two SIMD registers, `reg_a` and `reg_b`. In addition, we know that the loaded value of b is duplicated v times because $n_u < v$. Computing with `reg_a` and `reg_b` will yield a single unit update of size $v \times 1$. A single outer-product of $m_r \times n_r$ can then be computed through $m_r/m_v \times n_r/n_u$ multiple unit updates as shown in Figure 4.3.

Alternatively, a different algorithm emerges when $m_v = v$ and $n_u = 2$. The difference is that `reg_b` would contain two unique values, each duplicated $v/2$ times. After the first computation is performed, the values in `reg_b` has to be shuffled before the next computation can be performed. This computation-shuffle cycle has to be repeated at least $n_u - 1$ times in order to compute a single unit update. Pictorially, this is shown in Figure 4.4. The astute reader will recognized that we could have chosen to shuffle the values in `reg_a` without significantly changing the computation of the unit update.

4.3.3 A family of Outer-Product Algorithms

What we can observe from the two algorithms described previously are the following:

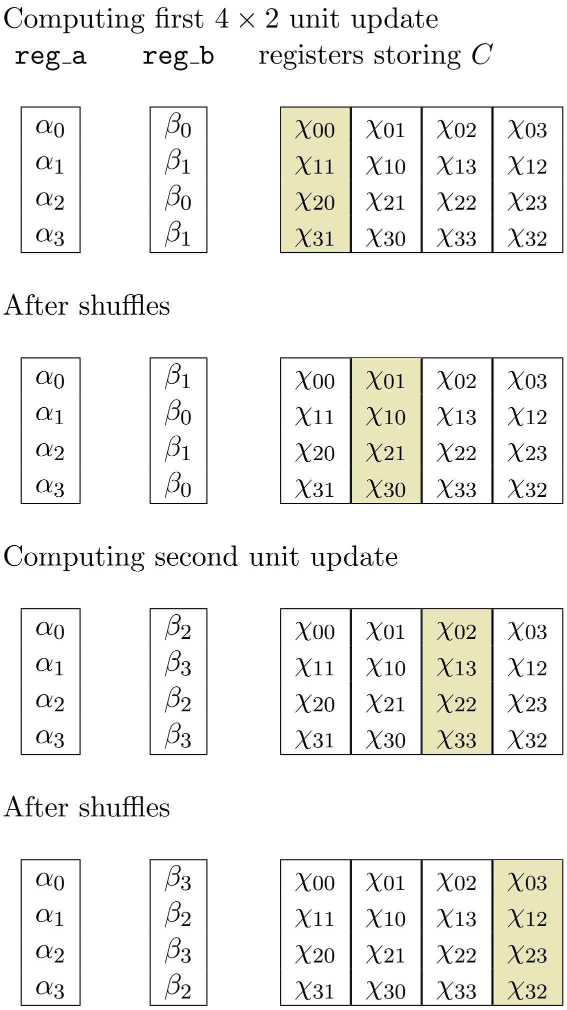
- The size of a single unit update can be determined by the number of unique values loaded into registers `reg_a` and `reg_b`.
- When there are more than one unique values in registers `reg_a` and `reg_b`, the number of computation-shuffles required is the minimum of m_v and n_u.
- Loading more unique values into `reg_b` reduces the number of Loads of b from the L1 cache, at the cost of increasing the number of shuffles required to compute the unit update.

Given that we chose not to shuffle `reg_a`, this means that there are $\log_2(v) + 1$ different ways of picking n_u, i.e. the number of unique elements loaded into `reg_b` (while still being a power of 2). For a given choice of n_u, the different ways in which the data in `reg_b` should be shuffled yields different implementations for the unit update. By accumulating the instructions for computing all $m_r/m_v \times n_r/n_u$ unit updates, different sets of instructions, or *instruction mix*, describing different implementations of the outer-product can be obtained.

Recall that the different number of loaded unique elements results in different number of loads and shuffle stages required. On different architectures, the cost (in term of latency) of loads and shuffles may differ, which suggests a need for a means to estimate the cost of computing with a set of instruction mix.

4.4 Selecting Outer-Product Kernels

Having derived a family of algorithms – or instruction mixes – to compute the outer-product, we need to select one of these algorithms to implement. We build a model of the architecture and then rely on queuing theory to select the kernel with the highest throughput. Specifically, we are selecting



Computing first 4 × 2 unit update

After shuffles

Computing second unit update

After shuffles

reg_a
reg_b
registers storing C
α_0 α_1 α_2 α_3
β_0 β_1 β_0 β_1
χ_{00} χ_{11} χ_{20} χ_{31}
χ_{01} χ_{10} χ_{21} χ_{30}
χ_{02} χ_{13} χ_{22} χ_{33}
χ_{03} χ_{12} χ_{23} χ_{32}
α_0 α_1 α_2 α_3
β_1 β_0 β_1 β_0
χ_{00} χ_{11} χ_{20} χ_{31}
χ_{01} χ_{10} χ_{21} χ_{30}
χ_{02} χ_{13} χ_{22} χ_{33}
χ_{03} χ_{12} χ_{23} χ_{32}
α_0 α_1 α_2 α_3
β_2 β_3 β_2 β_3
χ_{00} χ_{11} χ_{20} χ_{31}
χ_{01} χ_{10} χ_{21} χ_{30}
χ_{02} χ_{13} χ_{22} χ_{33}
χ_{03} χ_{12} χ_{23} χ_{32}
α_0 α_1 α_2 α_3
β_3 β_2 β_3 β_2
χ_{00} χ_{11} χ_{20} χ_{31}
χ_{01} χ_{10} χ_{21} χ_{30}
χ_{02} χ_{13} χ_{22} χ_{33}
χ_{03} χ_{12} χ_{23} χ_{32}

Figure 4.4: SIMD computation of a 4 × 4 outer-product using two unit updates of size 4 × 2. As there are multiple (2) unique values, vector shuffles must be performed to compute each unit update. Shaded registers denotes output register for the current stage of computation.

the instruction mix with the shortest average waiting time. This is because this instruction mix will fit inside a much larger looping structure and will be computed for many iterations. Thus, we are only concerned with the steady-state behavior, which is where most the compute time will be spent. The key assumption is that our architecture is a stable system in steady-state, where the arrival rate of instructions into the architecture is equal to the average departure rate of instructions out of the system. Additionally, we can ignore dependencies between instructions, because we are only interesting in how long the mix of instructions wait in the system. These instructions can come from different iterations of the instruction mix, which is sufficient because our kernels typically have no inter-loop dependencies other than accumulations. We will provide a brief overview of this process, but for a more detailed treatment of using queueing theory to estimate the performance of the outer-product instruction mixes, we refer the reader to our work in [21].

The first part of our queuing theory based approach to estimating the performance of instructions mixes starts with representing the microarchitecture as queues. Each functional unit in the microarchitecture is responsible for processing a subset of instructions from the Instruction Set Architecture (ISA), so we represent these functional units as servers and the instructions in the mix as jobs in a server's queue. We illustrate this in Figure 4.5.

Taking this server and queue view of the microarchitecture allows us to adapt Little's Law [8] to estimate the throughput of a given instruction mix. Formally stated as $L = \lambda W$, this law tells us that the expected number of jobs waiting for a server, L, is equal to the product of the average arrival rate of jobs into the system (λ) and the average waiting time of jobs in the system.

Since we are concerned with the throughput of an instruction mix – in particular we want to find the mix with the highest throughput – we can rewrite the formula as $\lambda = \frac{L}{W}$ to determine the throughput for any given functional unit. Because performance of an outer-product instruction mix is limited by the functional unit with the slowest throughput, we can estimate an outer-product mix as follows:

$$\lambda_{\text{outer product}} = \min\left(\frac{\lambda_i}{L_i}\right)$$

Where $\lambda_{\text{outer product}}$ is the throughput of a given outer-product instruction mix, λ_i is the throughput for functional unit i and L_i is the number of

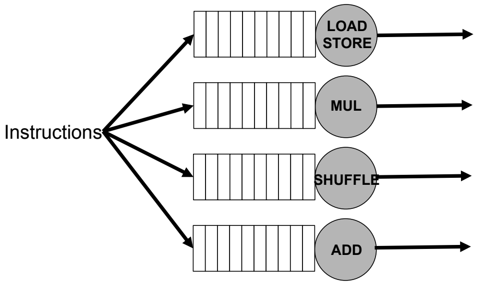


Figure 4.5: Model of a subset of the Intel Sandy Bridge architecture, showing only the floating point addition and multiplication units, the load/store units and the vector shuffle units. Instructions enter the pipelines, and when all instructions required for computing the outer-product leave their respectively pipeline, the outer-product is computed.

instructions in the given outer-product mix going to functional unit i. The throughput of the outer-product mix is the inverse of the time needed to clear the slowest functional unit. In the next subsection we will provide a concrete example.

4.4.1 Estimating Throughput

Consider the instruction mix required to compute the 4×4 outer-product shown in Figure 4.3 being executed on the model Sandy Bridge architecture described in Figure 4.5. The instruction mix contains a single Load of a vector of a, four Loads (with duplication) of elements from b, four multiplies and four adds (Computation) instructions. All instructions will be sent to their respective pipelines. In addition, another four job items are also sent to the pipeline connected to the shuffle functional unit. This is because the Load of element from b on the SandyBridge is a Composite instruction, that comprises of two instructions, a Load and a Shuffle.

Based on documentations from the hardware manufacturers [69], we know that the throughputs of the Shuffle and Computation instructions are 1 per cycle, and Loads have a throughput of 2 per cycle. This means that the

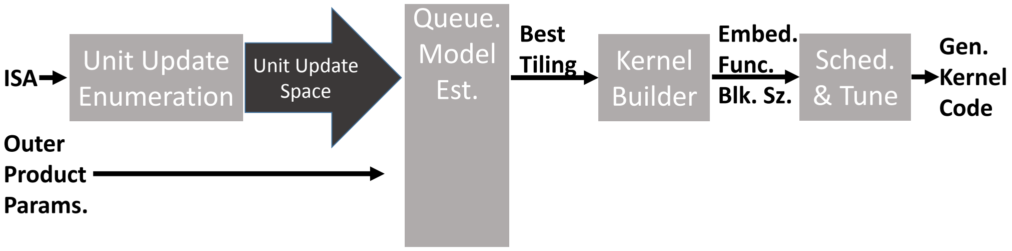


Figure 4.6: Our outer product kernel generation work flow produces expert level code by: enumerating a space of implementations, modeling their expected performance, selecting the top candidate and translating that into efficient code.

estimated throughput of the system is

$$\begin{aligned}\lambda_{\text{outer product}} &= \min\left(\frac{\lambda_{\text{load}}}{L_{\text{load}}}, \frac{\lambda_{\text{add}}}{L_{\text{add}}}, \frac{\lambda_{\text{mul}}}{L_{\text{mul}}}, \frac{\lambda_{\text{shuffle}}}{L_{\text{shuffle}}}\right) \\ &= \min\left(\frac{2}{5}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right) \\ &= 0.25 \text{ outer-product per cycle}\end{aligned}$$

However, these throughput values are based on the assumption that the instructions in all queues are fully pipelined, and independent. When the instructions in the pipelines are not independent, this implies that the latency of executing that instruction cannot be hidden. As such the throughput of the pipeline drops. This can happen when one instruction has to be computed before dependent instructions can be processed. What this means is that the pipeline has to stall, thus increasing the average waiting time of that pipeline.

4.5 Generating the Gemm Micro-Kernel

In the previous section, we described an analytical technique for determining the most efficient instructions mix for implementing the outer-product kernel on a given architecture. By using queuing theory to we can estimate the sustained throughput of a potential implementation without empirical measurements. The end goal of this step is to transform this highly parallel instruction-mix into an efficient kernel.

In this section we describe our code generator that performs this transformation on an instruction-mix. This generator takes as its input an outer-product instruction-mix and outputs a high performance kernel.

By using the parallel outer-product formulation over an inner(dot)-product formulation the resulting mix contains a large number of independent instructions. Using these independent instructions, or instruction level parallelism (ILP), the latency of these instruction are easily hidden through the application of static scheduling. This enables the kernel to achieve the performance predicted by the model. Thus, the main functions of our code generator is to capture the desired instruction-mix in a template, statically schedule the instruction-mix, and emit ANSI C code in a manner that preserves the static instruction schedule. In these three steps our generator produces a high performance kernel that approaches the theoretical performance of an instruction-mix as determined by our queueing theory model.

4.5.1 A Work Flow for Kernel Generation

Our complete work flow is captured in Figure 4.6, and it accomplishes the following: We start with the ISA for the target architecture, this includes the available instructions along with their latency and throughput. This information is passed to our *Unit Update Enumeration* stage which enumerates the space of all unit updates, for example given the ISA in Figure 4.7 the unit updates in Figure 4.8 are enumerated. After the unit update space is enumerated, all possible tilings of these unit updates that lead to a valid $m_r \times n_r$ outer-products are enumerated (Figure 4.9). The performance of these outer-products are then estimated using our queueing theory model described in the previous section. This process insures that the selected instruction-mix can sustain a high throughput, given that all other instruction overheads are minimize. The steps that follow, focus on minimizing said overhead through several optimizations including static instruction scheduling.

Continuing with the work flow, the best instruction-mix tiling is selected and passed to a *kernel builder* which blocks the outer-product into chunks of unit updates. This will reduce register pressure when we perform instruction scheduling (see Figure 4.11). The kernel builder then outputs a skeleton of the kernel, like the one in Figure 4.12, which captures the various blocking parameters of the kernel along with a set of *embedding functions* such as Figure 4.10. These embedding functions capture the selected instruction-mix as functions. At this point the embedding functions and skeleton represented

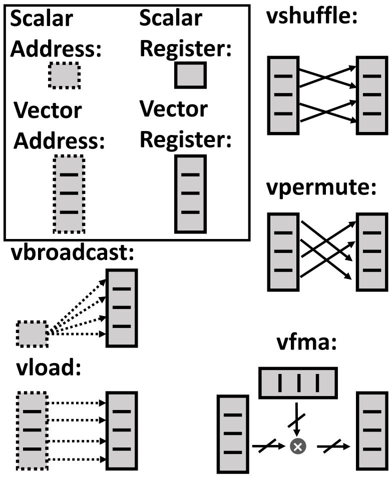


Figure 4.7: These cartoons illustrate the SIMD Vector instructions that are considered for outer-product kernel generation.

an untuned matrix multiply kernel that is implemented using the selected instruction-mix.

The embedding functions and the skeleton are then passed to a *Scheduling and Optimization* phase which hides the instruction latency by statically performing software pipelining scheduling [10], which allows the kernel to perform near the predicted performance. The resulting statically scheduled code is then emitted using inline assembly ANSI C intrinsics from [9], which produces C code that can be compiled with a fixed static schedule. This process yields outer-product kernel code (Figure 4.13) that achieves expert level performance. The role of the external compiler on this emitted C code is to provide register coloring, simplify memory indexing computation, and insure efficient instruction alignments for the fetch and decode stages of the processor.

4.5.2 Embedding Functions Capture the Instruction-Mix

The instruction-mix selected by our queuing model is not a complete kernel, instead it is a collection of instructions that describe the data movement and

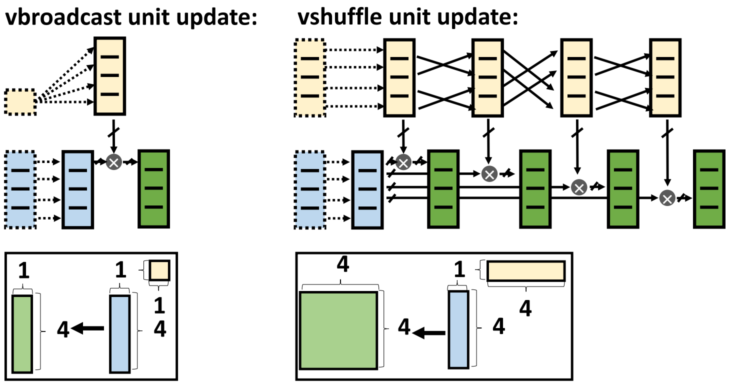


Figure 4.8: In this figure we show the smallest unit updates (or small outer products) that can be constructed from our base set of SIMD instructions.

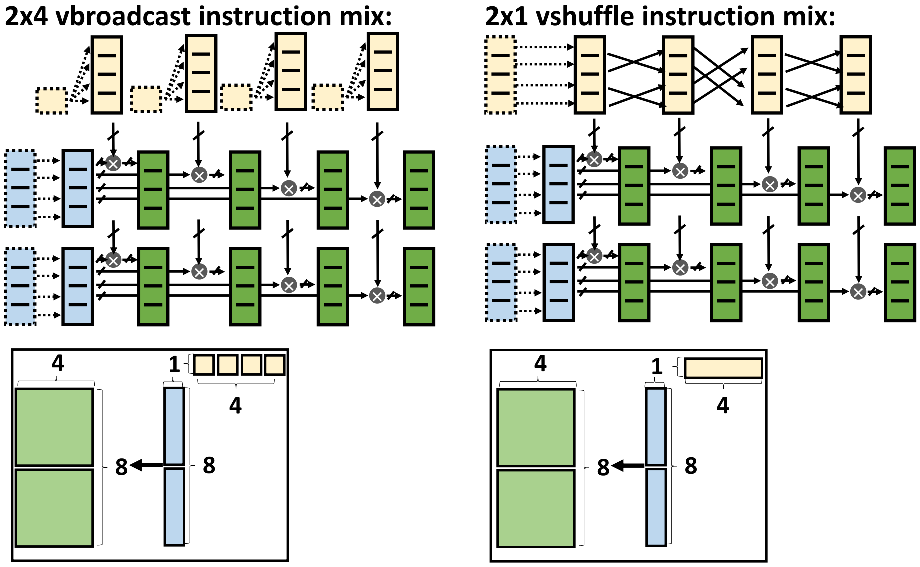

2x4 vbroadcast instruction mix:
2x1 vshuffle instruction mix:
4
1
4
8
8
4
1
4
8
8

Figure 4.9: Given the unit updates in Figure 4.8 we can enumerate two possible implementations of a $m_r \times n_r = 8 \times 4$ outer-product. On the left we have an instruction-mix composed entirely of `vbroadcast` unit updates and on the right we have an instruction-mix composed of `vshuffle` unit updates.

```
get_b_element( b_reg, ii, jj, pp )
 if( ii == 0 )
  switch( jj )
   case 0:
     b_reg[jj] = vload(&B[jj + pp*nr])
   case 1:
     b_reg[jj] = vshuffle(b_reg[jj-1])
   case 2:
     b_reg[jj] = vpermute(b_reg[jj-1])
   case 3:
     b_reg[jj] = vshuffle(b_reg[jj-1])

get_a_element( a_reg, ii, jj, pp )
 if( jj == 0 && ii mod v == 0)
  a_reg[ii] = vload( &A[ii + pp*mr] )

fma( a_reg, b_reg, c_reg, ii, jj, pp )
 if( ii mod v == 0 )
  c_reg[ii][jj] = vfma( a_reg[ii],
                        b_reg[jj],
                        c_reg[ii][jj] )
```

Figure 4.10: In order to pass the instruction-mix to the kernel code generator, it is encoded as a function, similar to listing. These functions dispatch to a specific instruction which depends on what element $C_{ii,jj}$ is being on in the outer product.

floating-point computation of data elements in a permuted outer-product. The dependencies, register allocation and memory address computation is implicit in this stage. Therefore, the first step in transforming this instruction-mix into a kernel is to make these characteristics explicit. This is done by expressing the three components of an outer-product instruction-mix (gathering of the elements A, gathering of the elements of B, performing the multiply and accumulate of C) are expressed as embedding functions: `get_a_element`, `get_b_element`, and `fma` (Figure 4.10). In these functions dependencies, register utilization, and memory address computation are made explicit.

These functions take the following inputs: arrays of registers that represent the elements of a,b and c, along with the indices for the m, n and k dimension of the current unit-update. The embedding functions maps the indices of the outer-product to the appropriate instructions from the instruction-mix. Because these functions capture the majority of the work a handful of optimizations are applied to these embedding functions. Namely, we optimize for the fetch and decode stage of the processor by minimizing bytes per instruction. This is critical because the maximum rate of floating point computations for a modern processor is very close to the maximum rate that it can fetch and decode instructions.

Optimizing for Bytes per Instruction. On most modern processors, the maximum throughput of the fetch and decode units is low enough to become a bottleneck. Thus, some of the optimizations performed by our generator minimize the instruction length and decode complexity in order to avoid this bottleneck. The following decisions ensure that shorter instructions are generated:

1. In some cases, we generate instructions that are meant to operate on single-precision data instead of instructions that operate on double-precision data. An example of this is the use of the `vmovaps` instruction to load `reg_a`, instead of `vmovapd`. This is because both instructions perform the identical operation but the single-precision instruction can be encoded in fewer bytes.
2. We hold the partially accumulated intermediate $m_r \times n_r$ matrix of C, which we will refer to as T, using high-ordered registers (i.e. register `xmm8` to `xmm16`). On most architecture we tested, high-ordered SIMD registers require more bytes to encode. Thus, by using the low-ordered registers to hold working values and high-ordered registers to store T, we ensure that each instruction has at most one register operand (i.e. the output operand) that is a high-ordered register.
3. For memory operations, address offsets that are beyond the range of -128 to 127 bytes require additional bytes to encode. Therefore, we restrict address offsets to fit in this range by subtracting 128 bytes from the base pointers into A and B.

4.5.3 From Embedding Functions to Generated Code

Once we have established the necessary peripheral instructions to the instruction-mix and established the dependencies between these instructions we can generate an implementation. This entails the application of instruction schedul-

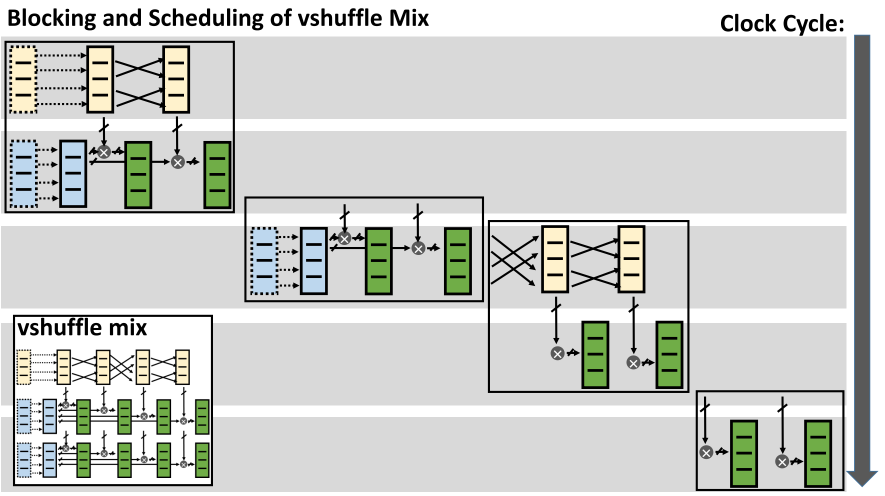


Figure 4.11: Once a candidate outer product tiling is selected (figure 4.9), we perform an additional layer of blocking (m_s, and n_s) to assist the code generator in minimizing register spills. Register blocking allows fewer registers to be live at a given cycle thus allowing the code generator to aggressively schedule the instructions.

ing in order to hide – or overlap – the latency associated with these instructions. Static instruction scheduling is key for this next step, however this optimization is limited by the number of available registers. Thus, the primary step in the *kernel builder* (Figure 4.6) is to determine a further layer of blocking for the outer product to insure a sufficient number of registers for scheduling.

4.5.4 Limits Imposed by Registers

Recall that in order to compute a unit update, n_u permutations of the elements in `reg_b` are required. However, a multiply and an add is performed with each permutation. This implies that each unit update will require two new registers ($R_R = 2$): One to store the permutation of `reg_b`, and another to hold the output of the multiplication. On architectures with a *fused-multiply-add* instruction, only one new register is required (i.e. $R_R = 1$).

As the register that holds the output of the accumulated result is reused over multiple outer-products, it means that the number of vector width computations that we can perform in a given unit-update is (N_{updates}) that can

```
/* initialize temp buffer */
for( i = 0; i <  m_r; i++ )
 for( j = 0; j <  n_r; j++ )
  c_reg[ii][jj] = 0;

/* computation */
#unroll(k_u)
#schedule_software_pipeline
for( pp = 0; pp < k_b; pp++ )
 /* perform the outer products */
 for( i = 0; i <  m_r; i+=m_s )
  for( j = 0; j <  n_r; j+=n_s )
   for( ii = i; ii < i+m_s; ii++ )
    get_a_elem(a_reg, ii,j );
    for( jj = j; jj < j+n_s; jj++ )
     get_b_elem(b_reg, ii,jj );
     fma( c_reg, a_reg, b_reg, ii,jj,pp );

/* accumulate temp to results */
for( i = 0; i <  m_r; i++ )
 for( j = 0; j <  n_r; j++ )
    C[(ii,jj)] += c_reg[ii][jj];
```

Figure 4.12: In this code skeleton we capture an outline of the generated kernel. We pass a similar outline to our code generated along with our instruction-mix. This mix is encoded as the functions `get_a_elem`, `get_b_elem` and `fma`. The code generator uses this information to generate the code, perform optimizations such as unrolling and code motion and schedule the resulting kernel code using software pipelining in way that targets the microarchitecture.

be performed without register spilling is constraint only by the number of registers given by the following:

$$N_{\text{updates}} = \left\lfloor \frac{R_{total} - \frac{m_r n_r}{m_v} - R_A}{R_R} \right\rfloor, \tag{4.1}$$

where R_{total} and R_A are the total number of registers and the registers required to hold the the column of A.

If we let $m_s n_s v = m_r n_u$ then these blocking dimensions m_s and n_s for tiling unit-updates are selected such that:

$$m_s n_s \leq N_{\text{updates}} v \tag{4.2}$$

In Figure 4.11, we show the blocking and scheduling of a `vshuffle` instruction-mix (Figure 4.9) for a $m_r \times n_r = 8 \times 4$ outer-product.

4.5.5 Scheduling and Tuning

Remember, the instruction-mix selected by the *Queueing Model Estimator* is translated by the *Kernel Builder* into several *embedding functions* (`get_a_elem`, `get_b_elem` and `fma`), like those in Figure 4.10. These function are embedded in a looping structure that matches the outer-product kernel (Figure 4.12). These loops iterate over the m_r, n_r, m_s and n_s dimensions and generate the dependencies between the instructions inside the embedding functions.

Once these dependencies are built, a few basic optimizations, such as common sub-expression elimination, are performed such that the only the original instruction-mix plus a few looping instructions exist in the final output code.

Next, the code generator performs software pipelining [10] over the entire looping structure of the outer-product. By statically scheduling the kernel, the risk of instruction stalls is minimized thus allowing the processor to compute the instruction-mix near the rate predicted by our model. Once the code is scheduled, the generator emits a mix of C code and inline assembly instruction macros which preserve the schedule [9]. This resulting code implements a high performance outer-product kernel with the desired dimensions and the selected instruction-mix. We provide an excerpt of a generated kernel in Figure 4.13.

```
for( pp = 0; pp < k_b; pp+=KUNR )
{
 /* STEADY STATE CODE */
 VLOAD_IA(GET_A_ADDR(0),GET_A_REG(0))
 VLOAD_IA(GET_A_ADDR(1),GET_A_REG(1))
 VLOAD_IA(GET_B_ADDR(0),GET_B_REG(0))
 VSHUFFLE_IA(0x05,GET_B_REG(0),GET_B_REG(1))
 VFMA(GET_A_REG(0),GET_B_REG(0),GET_C_REG(0,0))
 VFMA(GET_A_REG(0),GET_B_REG(1),GET_C_REG(0,1))
 VPERM2F128_IA(0x01,GET_B_REG(1),GET_B_REG(2))
 VSHUFFLE_IA(0x05,GET_B_REG(2),GET_B_REG(3))
 VFMA(GET_A_REG(1),GET_B_REG(0),GET_C_REG(0,0))
 VFMA(GET_A_REG(1),GET_B_REG(1),GET_C_REG(0,1))
 /* snip */
}
```

Figure 4.13: This is generated excerpt from our kernel generator. The resulting kernel code implements the instruction-mix identified by our queueing theory model and is statically scheduled to maintain the estimated performance of the mix.

4.6 Experimental Results

In this section, we test the effectiveness of our kernel generation system in automating the last-mile for high performance dense linear algebra. We evaluate both the queueing theory model, which finds an efficient outer-product instruction-mix, and the code generation system, which translates that mix into a high performance kernel.

We use a variety of machines listed in Table 4.5 that span a diverse range of double precision vector lengths ($v \in \{2, 4, 8\}$), number and partitioning of functional units, and instruction latencies. Because our kernel fits in the context of a larger Goto/BLIS-style Gemm context, the blocking parameters m_c, k_c, m_r and n_r are determined from the analytical models developed in [2] and [20] along with the cache and microarchitecture details listed in Table 4.1 and Table 4.2 respectively. The microarchitecture details in particular (Table 4.2) were used by the queueing theory model to select the highest throughput outer-product instruction-mix. Additionally, these details determined N_{updates} and the register sub-blocking dimensions m_s, n_s using the formula developed in the previous section.

Proc.	**uArch.**	**Freq.**	$S_{\mathbf{L1}}$ **(KiB)**	$W_{\mathbf{L1}}$	$N_{\mathbf{L1}}$	$S_{\mathbf{L2}}$ **(KiB)**	$W_{\mathbf{L2}}$	$N_{\mathbf{L2}}$	$S_{\mathbf{L3}}$ **(MiB)**	$W_{\mathbf{L3}}$	$N_{\mathbf{L3}}$
Core 2 X9650	Penryn	3	4×32	8	256	2×6000	24	16384	-	-	-
Xeon X5680	Nehalem	3.333	6×32	8	64	6×256	8	512	12	16	12288
Core i5-2500	Sandy Bridge	3.3	4×32	4	512	4×256	4	4096	6	12	32768
Core i7-4770K	Haswell	3.5	4×32	8	256	4×256	8	2048	8	16	32768
Xeon Phi 5110p	Knights Corner	1.053	60×32	8	256	60×512	8	4096	-	-	-

Table 4.1: Cache details of the processors used in our experiments. These cache details are needed for determining m_c, k_c, m_r and n_r according to [2] and [20]. The value S_l corresponds to the size of the l level of cache. W_l is the number of ways and N_l is the number of cache lines in each way.

uArch.	**Reg.**	$\ell_{\mathbf{fma}}$	$\ell_{\mathbf{L1}}$	$\ell_{\mathbf{L2}}$	$\ell_{\mathbf{shuf.}}$	$\ell_{\mathbf{prm.}}$	$\ell_{\mathbf{bcst.}}$	$R_{\mathbf{fma}}$	$R_{\mathbf{mem}}$	$R_{\mathbf{shuf.}}$	$R_{\mathbf{prm.}}$	$R_{\mathbf{bcst.}}$
Penryn	16	8	3	15	1	-	1	$p_0 \wedge p_1$	p_2	p_5	-	p_0
Nehalem	16	8	4	10	1	-	2	$p_0 \wedge p_1$	p_2	$p_0 \vee p_5$	-	p_5
Sandy Bridge	16	8	4	12	1	2	3	$p_0 \wedge p_1$	$p_2 \vee p_3$	p_5	p_5	$p_5 \wedge (p_2 \vee p_3)$
Haswell	16	5	4	12	1	3	5	$p_0 \vee p_1$	$p_2 \vee p_3$	p_5	p_5	$p_2 \vee p_3$
Knights Corner	32	4	1	11	4	6	4	p_0	p_{mem}	$p_0 \wedge p_{\text{mem}}$	p_0	$p_0 \wedge p_{\text{mem}}$

Table 4.2: Here we capture the pertinent microarchitecture parameters that are used for our queueing theory model. The column ℓ_u represents the latency in cycles of instruction u, where L1 and L2 represents instruction reads that hit in those caches. In the case of a system without fused-multiply-add (fma), the latency is represented as the sum of the multiply instruction and add instruction. The columns R_u represent the functional units that are required to compute instruction u. For some instructions multiple function units may be required (represented by $\wedge$) and some instruction may take multiple paths (represented by $\vee$).

# vperm. updates	# vbcast. updates	# Reads (L_{mem})	# Vect. (L_{p_0})	λ_{op}	$\frac{\textbf{flop}}{\textbf{cyc.}}$	$\frac{\textbf{GFLOP}}{s}$
0	30	1+0+30+4=35	31	0.02857	13.71	14.44
1	26	1+1+26+4=32	32	0.03125	15	15.80
2	22	1+2+22+4=29	33	0.03030	14.55	15.32
3	18	1+3+18+4=26	34	0.02941	14.12	14.87
4	14	1+4+14+4=23	35	0.02857	13.71	14.41
5	10	1+5+10+4=20	36	0.02778	13.33	14.04
6	6	1+6+6+4 =17	37	0.02703	12.97	13.66
7	2	1+7+2+4 =14	38	0.02632	12.63	13.30

Table 4.3: We estimate the number of cycles needed to compute our generated Xeon Phi outer-product kernels. The first column is the number of `vpermute` unit updates of size 4 × 8 used to implement the outer product. The remainder of the outer-product is computed using multiple 1 × 8 broadcast based unit updates. Note, $\lambda_{\text{op}} = \lambda_{\text{outer-product}} = \min(\frac{1}{L_{\text{mem}}}, \frac{1}{L_{p_0}})$ is the throughput of an outer-product.

$m_r \times n_r$	**#bcast. updt.**	**#shuf. updt.**	**#mem.** $L_{p2 \vee p3}$	**#fma** $L_{p0 \wedge p1}$	**#shuf.** L_{p5}	$\lambda_{\textbf{out.-prod.}}$	$\frac{\text{flop}}{\text{cyc.}}$	$\frac{\text{GFLOP}}{s}$
8×4	8	0	$2 + 8 = 10$	8	8	0.125	4	26.4
8×4	0	2	$2 + 1 = 3$	8	3	0.125	4	26.4
4×12	0	3	$1 + 3 = 4$	12	9	0.083	4	26.4
4×12	12	0	$1 + 12 = 13$	12	12	0.083	4	26.4

Table 4.4: We estimate the number of cycles needed to compute our generated Sandy Bridge outer-product kernels. We implement outer-products of size $m_r \times n_r \in \{8 \times 4, 4 \times 12\}$. Note that the model predicts similar performance across these implementation, however due to subtle microarchitecture details the experimental performance is different.

Processor(uArch.)	$m_c \times k_c$	$m_r \times n_r$	N_{updates}	$m_s \times n_s$
Core 2 X9650 (Penryn)	256×256	4×4	3	2×2
Xeon X5680 (Nehalem)	256×256	2×8	3	2×2
Core i5-2500 (Sandy Bridge)	96×256	8×4	3	4×2
Core i7-4770K (Haswell)	256×512	4×12	3	4×4
Xeon Phi 5110p (Xeon Phi)	120×240	30×8	1	8×1

Table 4.5: The cache blocking parameters m_c and k_c where determined using the results in [2] and the hardware parameters in Table 4.1 and Table 4.2. The register blocking parameters m_r and n_r were determined from [20] using the values in Table 4.1. Lastly, N_{updates} and subsequently the sub-blocking dimension m_s and n_s were determined using Equation 4.1. m_c, k_c, m_r, n_r, m_s and n_s correspond to the values used in the generated code (see Figure 4.12). Note $N_{\text{updates}} v \geq m_s n_s$

.

4.6.1 Analysis of Queueing Model

In order to demonstrate the effectiveness of our model, we compare the predicted performance against the actual performance estimated by our queueing theory model. For the Xeon Phi, we compare the performance of eight different instruction-mix implementations of an 8×30 outer-product. We selected a family of instruction-mixes where the work is partitioned between 8 permute unit updates and 8×1 broadcast based unit updates. In Table 4.3, we detail each outer-product implementation. Each row represents a specific implementation, where the first two columns represent the number of permute and broadcast unit updates in the implementation. In the next two columns we compute the number of instructions that need the memory port (p_{mem}) and vector port (p_0) respectively. For the Xeon Phi each permute component requires one load instruction, a permute instruction and four fma instructions. The broadcast based component requires one load and one fma instruction. Additionally, each implementation requires four prefetch instructions that occupy the memory ports. In the fifth column we use our queueing theory model to estimate the performance of the implementation.

We can estimate the performance in FLOP per cycle as:

$$\frac{\text{flop}}{\text{cyc.}} = \lambda_{\text{outer-product}}(m_r)(n_r)(2flop) \tag{4.3}$$

In the last column, we estimate the performance in GFLOP using the following formula:

$$\frac{\text{GFLOP}}{\text{s}} = f\frac{\text{flop}}{\text{cyc.}} \tag{4.4}$$

Each of these implementations has a different throughput predicted by our model. In our experiment (Figure 4.14), we compare the relative performance of these implementations. Assuming that the overheads are similar between all implementations, then if the model does not fit, we expect a significant difference between the relative ordering of the implementations and the predictions. However, for the Xeon Phi we see that the relative ordering of the implementations is preserved in the experimental results, with the exception of one of the implementations. We suspect that the overhead is slightly lower for the *0 permute, 30 broadcast* implementation.

It is worth noting that the Xeon Phi requires that four threads run concurrently in order to effectively utilize a core. This requires that we distribute the work across multiple threads. Therefore, we used the implementation in [7] with the following parameters: the number of threads used in each dimension (i_c and j_r) must satisfy $i_c j_r \leq 59(4)$, and ideally should be factors of $\frac{m}{m_c}$ and n respectively. By empirical selection, $i_c = 12$ and $j_r = 16$ satisfied both of those requirements and resulted in the largest number of cores that achieved efficient per core performance.

We repeat the same experiment with the Sandy Bridge processor. In Table 4.4 we estimate the performance for several implementations. Unlike the Xeon Phi experiment we chose two different kernel sizes ($m_r \times n_r$). According to [20], the 8×4 implementations is more efficient than the 4×12. In Figure 4.14 we plot the performance of the four implementations. What we see is despite that our model predicts identical performance, we see a significant difference between the kernels of different sizes. What this demonstrates is that even if we can produce an efficient kernel in isolation, our model operates within the constraints of the larger GotoBLAS/BLIS algorithm. There are also additional and subtle microarchitectural details that explain the difference between implementations of the same size on this system. For example, even though both ports p_2 and p_3 service memory operations, they are limited in the total number of bytes that can be read in a cycle. Therefore, the

Sandy Bridge retires less than 2 memory operations per cycle. In the case where this is not an issue (between the two 8 × 4 implementations, we attribute the performance difference to scheduling because the permute based approach has fewer dependencies than the broadcast implementation, giving the scheduler greater freedom to hide instruction latency.

This experiment demonstrates that for outer-product implementations of same size we can accurately estimate the performance of our generated implementations. However, our kernels operate within the constraints of a bigger GotoBLAS/BLIS Gemm algorithm, and our performance is ultimately limited by the parameters selected for the bigger algorithm. In the next subsections, we look at this interaction in the opposite direction, how decisions made in generating the kernel affect the overall GotoBLAS/BLIS algorithm.

4.6.2 Analysis of the Generated Kernels

We evaluate the effectiveness of our kernel generation approach by comparing the performance of our generated outer-product kernels against state-of-the-art Gemm implementations such as OpenBLAS [65] and ATLAS [17]. We selected OpenBLAS because it is the highest performance open source BLAS implementation on most architectures, including the systems used in this paper. ATLAS was also selected because it is a high performance code generation system. Unlike, our code generator, this framework relies on hand tuned assembly kernels and uses search to determine the blocking dimensions around these kernels. The systems used in this experiment represent the past four major microarchitecture designs from Intel Table 4.5. The parameters for the GotoBLAS/BLIS Gemmalgorithm were analytically selected to maximize performance. These values also match the ones used by the OpenBLAS.

In these experiments (Figure 4.15 and Figure 4.16), our generated code is within 2-5% of the expert tuned OpenBLAS. We suspect this difference is due to loop overhead because we rely on the compiler to optimize this which results in several extra instructions over the expert code. The older the processor generation, the more pronounced of an effect this has, which is why ATLAS outperforms our code on the Penryn. We believe we can resolve this difference by implementing the looping structure in inline assembly which should give us performance that is near identical to the expert written code.

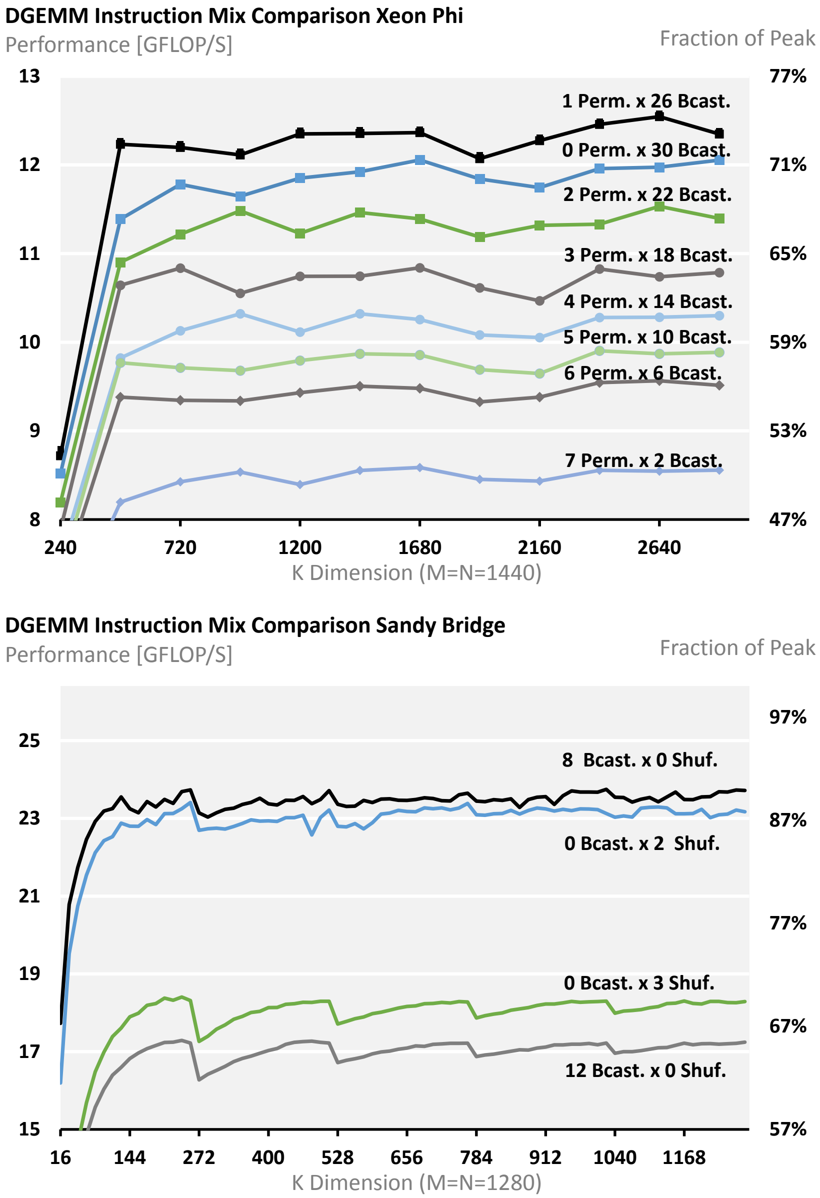


Figure 4.14: In both of these experiments we test the accuracy of our queueing theory model. **Top:** On the Xeon Phi we evaluate the performance of eight implementations of the same outer-product. **Bottom:** We do a similar experiment, but with two different outer-product sizes.

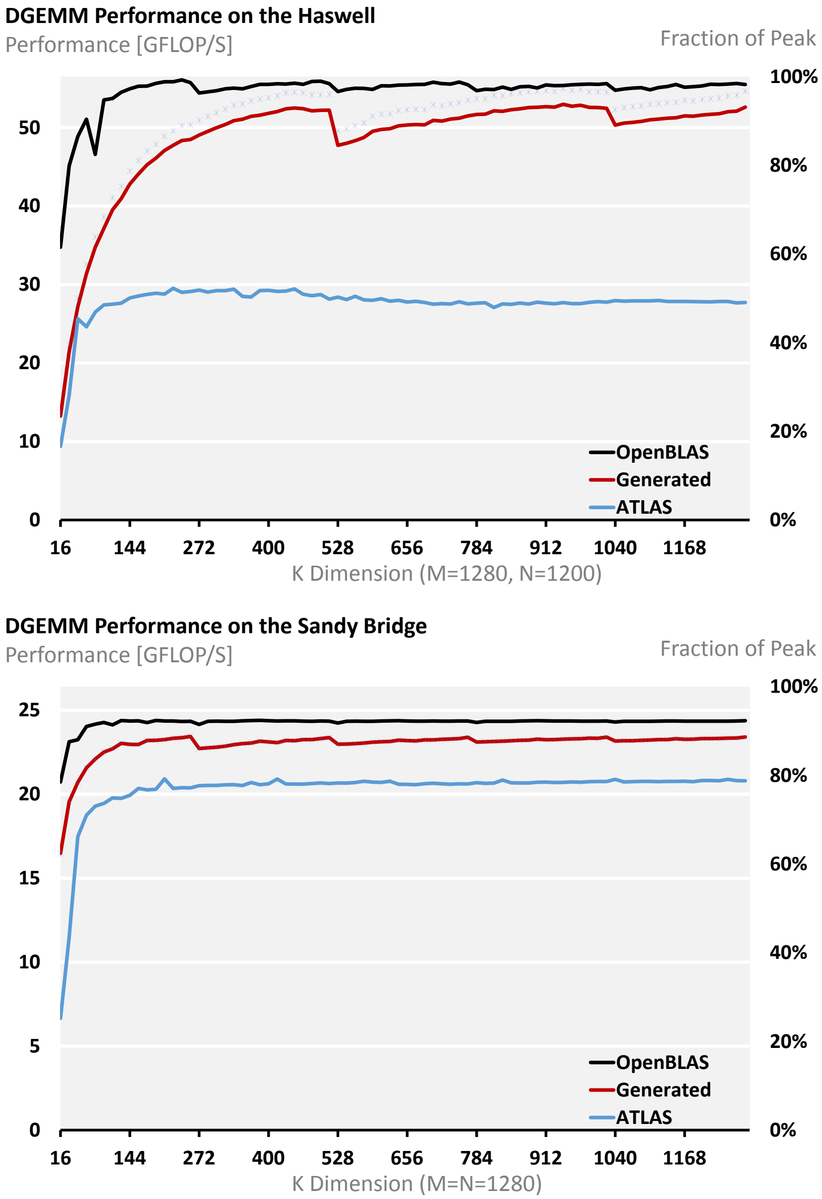


Figure 4.15: We compare the performance of our generated kernels against ATLAS and the OpenBLAS for various problem sizes to demonstrate that expert level performance can be automated. We see that our generated code approaches the performance of hand tuned expert code and for most architectures exceeds the performance of the generated ATLAS code.

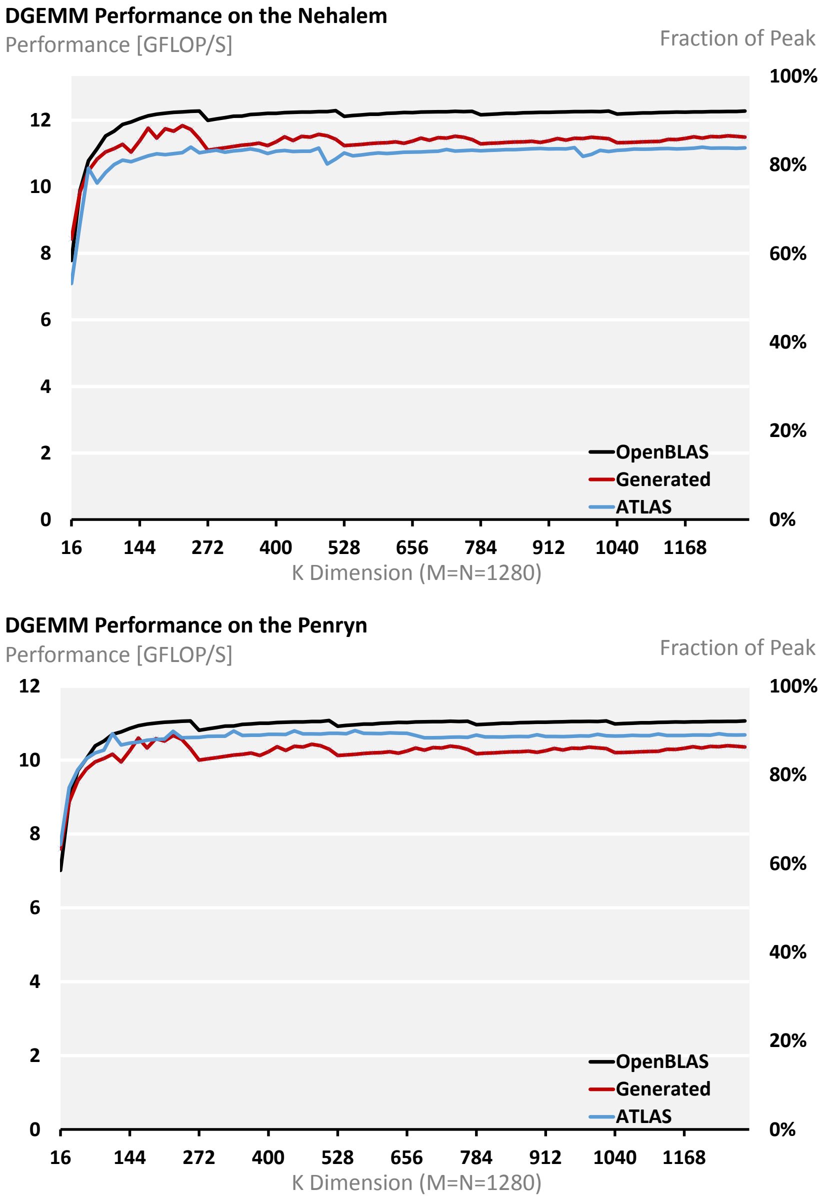


Figure 4.16: Like the graphs in Figure 4.15, we compare the performance of our generated kernels against the OpenBLAS and ATLAS.

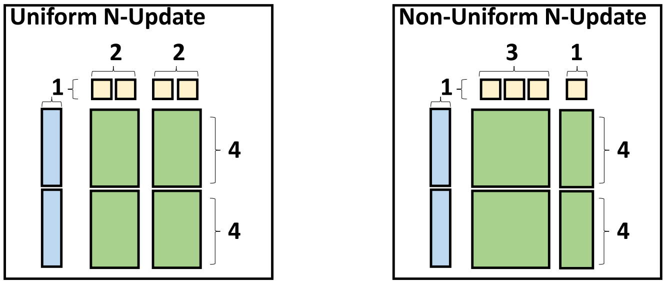


Figure 4.17: Given an outer-product instruction-mix of an $m_r \times n_r = 8 \times 4$, we can partition it into uniformly sized N-updates or non-uniform size N-updates. In the uniform case, each N-update is $m_s \times n_s = 4 \times 2$, in the non-uniform case each N-update is $m_s \times n_s \in \{4 \times 3, 4 \times 1\}$.

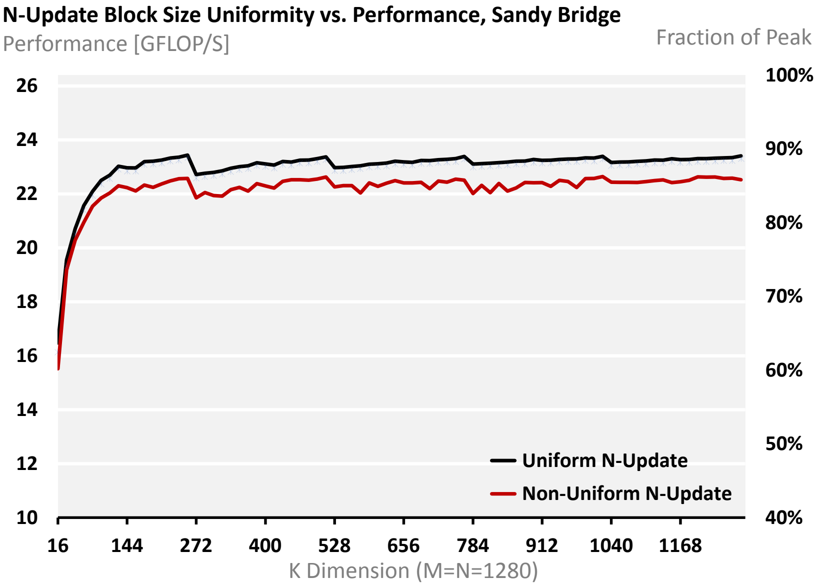


Figure 4.18: In this experiment, we compare the performance of two kernels implementing the same $m_r \times n_r = 8 \times 4$ outer-product using either uniform or non-uniform N-update sizes. The uniform N-update implementation performs better because it leads to few clusters of large (in Bytes) instructions which prevent the fetch and decode stages from becoming bottlenecks.

Port (R_U)	p_0	p_1	p_2 - ℓ_D	p_3 - ℓ_D	p_5
Cycles - Uniform	32.0	32.0	8.0 - 12.0	8.0 - 16.0	16.0
Cycles - Non-Uniform	32.0	32.0	8.0 - 12.0	8.0 - 16.0	16.0

Table 4.6: IACA results comparing instruction throughput between the uniform and non-uniform N-update implementations. Each port represents a functional unit that is used for our operation. ℓ_D represents data fetch latency. What this shows is that both the uniform and non-uniform shaped implementations of the same outer product look identical to the Out-of-Order engine, as simulated by the IACA tool. However, we will show the performance of the two implementations are significantly different.

4.6.3 Sensitivity to Parameters

In addition to using static scheduling and avoiding register spilling, we observe that even in the presence of an Out-Of-Order engine there is a benefit from maintaining uniformly-sized N-updates for creating the outer product.

Given two implementation of the micro-kernel, we vary the instruction tile sizes and compare the performance. Our reference implementation uses uniformly sized N-updates of size $m_s \times n_s = 4 \times 2$. We compare this to an implementation composed of two types of N-updates of size $m_s \times n_s = 4 \times 3$ and $m_s \times n_s = 4 \times 1$. We illustrate these two implementations in Figure 4.17, where each outer-product is partitioned according to a uniform or non-uniform scheme.

We ensure that both implementations are free of register spilling and are scheduled – not only to avoid stalls – but also to insure that the number of instructions between prefetch instructions and their subsequent loads are uniform. We ran both implementations through the Intel Code Architecture Analyzer (IACA), a software simulator for Intel microarchitectures, and determined that both implementations lack instruction stalls, spend an equal number of cycles on each functional unit, and have an identical throughput (Figure 4.6). However, the results in Figure 4.18 do not reflect the results we obtained from IACA because the non-uniform N-update implementation performs 4% worse than the uniform N-update.

The non-uniform N-update implementation leads to clusters of instructions with very long encodings. This would present a bottleneck for the

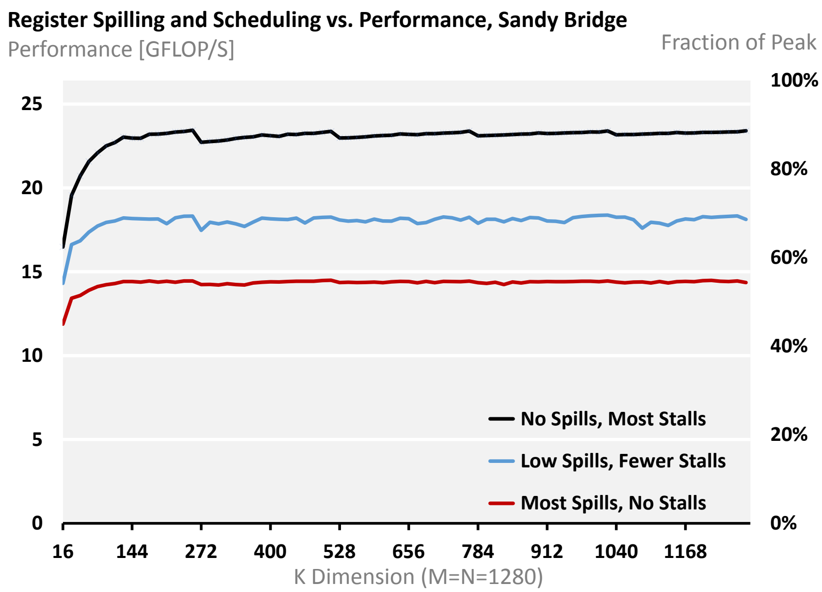


Figure 4.19: In this experiment we show that improving static scheduling by increasing the number of register spills degrades the overall performance of the Gemm operation. The GotoBLAS/BLIS algorithm attempts to minimize the number of TLB misses, however spilling into memory that would not have otherwise been used, increases TLB misses in this algorithm.

decoder and slow down the overall execution rate. Using uniform N-updates results in large instructions being evenly distributed throughout the code which prevents the decoder from becoming a bottleneck.

Registers. For generating our kernels, our aversion to register spilling goes beyond the performance penalty of the additional store to and load from memory. The reason is that these kernels fit in a much larger Gemm algorithm that achieves high performance by reducing Translation Look-aside Buffer (TLB) misses, and by spilling registers to memory this is disrupted and performance degrades significantly as result of these TLB misses. To demonstrate how large of an effect that register spilling in the kernel has on the number of TLB misses, we evaluate three kernels with varying degrees of register spilling (No Spilling, Moderate and Heavy). This is achieved by varying how much the N-updates overlap when we schedule them using software pipelining. The greater the overlap, the greater the register pressure

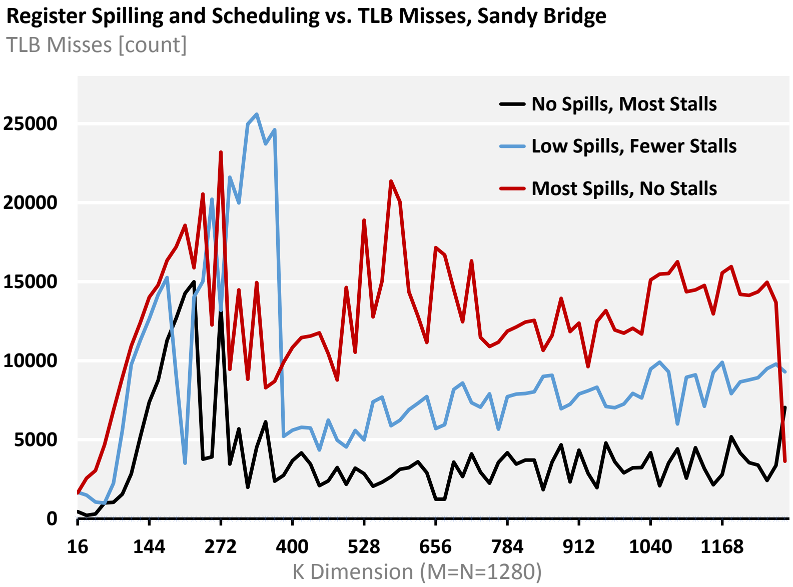


Figure 4.20: The overall algorithm that our kernels embedded in, the GotoBLAS/BLIS Gemm, maximizes performance by insuring that the kernel receives data at a sufficient rate while minimizing TLB misses. Spilling into memory requires that extra TLB entries be utilized to address memory that would not have been used otherwise. Thus even if spilling improves the kernel performance in isolation, it will degrade the overall performance of the Gemm operation.

and the larger the number of spills. In addition to measuring performance (FLOPs per cycle), we also measure TLB misses using PAPI [70]. The goal is to show that by increasing the amount of register spilling we will disrupt how the larger Gemm algorithm avoids TLB misses.

The performance per cycle results in Figure 4.6.3 demonstrate that as we increase the number of spills performance decreases – which is what we would expect. In Figure 4.6.3, We see that for large problem sizes the number of TLB misses is greater for the Heavy amount of spilling compared to the Moderate amount which is greater for the No Spilling case. If it were the case that the added latency incurred by the register spills were the only source of performance penalty, then we would not expect to see a change in the number of TLB misses between the three cases.

This shows that register spilling has performance implication beyond the additional round trip to cache because it disrupts the TLB miss avoiding characteristics of the Gemm algorithm described in [2]. For practical purposes this removes spilling as an option when the outer-product instruction-mix is translated into a kernel.

4.7 Chapter Summary

In this chapter, we address the last-mile problem of generating the architecture-specific micro-kernel for the general matrix-matrix multiplication routine that underlies most high performance linear algebra libraries. Specifically, we reveal the system behind generating high performance kernels that traditionally is implemented manually by a domain expert.

In following with the two part method of this thesis, we rely on an existing access pattern from the GotoBLAS and BLIS, and provide an approach to generating high performance tuned kernels. To validate that the performance of our generated micro-kernels are indeed high performance, we compared our generated results with those from OpenBLAS, which uses a similar approach to high performance matrix multiply. On many of the architectures, we demonstrated that the generated kernels are within 2-5% of the OpenBLAS performance. In addition, we also show that the generated kernels also scale in a similar fashion when parallelized on SMP system such as the Xeon Phi. While analytically generating the micro-kernels makes us competitive with expert-implemented kernels, empirical fine-tuning could then be employed to recover the missing performance without having to exhaustively search over a large space. This is something we will explore in the future. In the next chapter, we extend this mechanical approach of decoupling data access from the kernel computation and optimizing each separately, to structured mesh (stencil) computations.

Chapter 5

Structure Matters for Structured Mesh Computations

5.1 Introduction

Stencil, or structured mesh, computations are an important class of problems that arise from the simulation of a variety of problems. In this chapter, we approach stencil computations in a fashion similar to our matrix-multiplication approach. Given a stencil computation, we split the problem into two parts: the access pattern and the kernel code. The algorithm, and in turn, the access pattern is determined by the polyhedral compiler PolyOpt/C [71] and the kernel generation is performed by our automatic code generator. In this chapter we will focus on the code generator, and a more detailed explanation of the compilation processed can be found in [40].

The contribution of this chapter to the whole of the thesis is as follows:

- We will describe where the structure arises in these problems and how it is exploited.
- We show how the regular structure of a stencil computation can be exploited for performance in the generation of kernel code.
- We will show how to produce efficient SIMD code for structured computations.

In the following section, we provide a motivating example which shows step by step where stencil computations come from. After that, we will describe important optimizations for stencil operations and how the structure of the problem enables these transformations. Following that, we will discuss the details of our code generation system which was built on Spiral [72]. In the next section we evaluate the performance of our approach. Finally, we will summarize the chapter.

5.2 Background

A stencil computation is an extremely regular computation that is typically defined as a filter, or stencil, applied on each element in the data set. This stencil is a function which computes the value of a point based on its neighbors. In the following paragraphs we describe the transformation of a real world problem into a stencil computation.

5.2.1 A Motivating Example

Stencils commonly arise from the solution of Partial Differential Equation (PDE) using the Finite Difference Method. For example, suppose we want to solve the heat flow equation for a conductive rod. We would first describe the problem using a models and in this situation they are the Partial Differential Equation (PDE) for the heat transfer equation.

$$\frac{\partial u}{\partial t} = h^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{5.1}$$

This equation succinctly captures the flow of heat through an object where $u(x, y, z, t)$ is a function that represents the temperature at position (x, y, z) at time t, and h is the thermal diffusivity of the material. For the sake of simplicity we are going to ignore boundary conditions and heat sources. If we only focus on the one dimensional case we can drop the other two variables and use the following equation:

$$\frac{\partial u}{\partial t} = h^2 \frac{\partial^2 u}{\partial x^2} \tag{5.2}$$

If we want to solve this equation numerically then we need to discretize the function u on a grid. We show this in Figure 5.1, where the spacing s of the grid elements determines the desired resolution.

Additionally, because we are looking for numerical solution we will approximate the partial derivatives of u using difference equations. For example

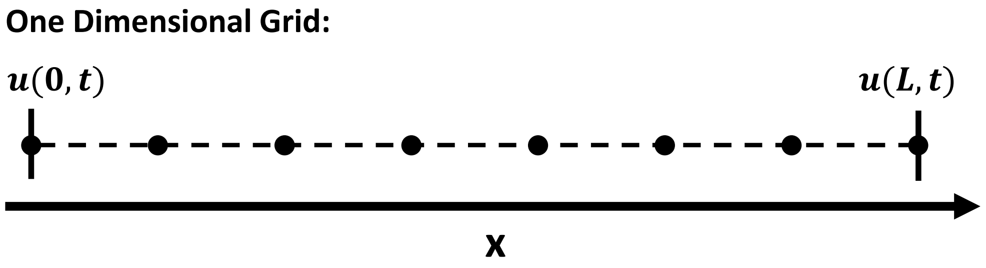


Figure 5.1: We discretize the metal rod on this one dimensional grid of length L with a grid spacing of s.

we can compute the first derivative $\frac{\partial u}{\partial x}$ using the following forward difference equation:

$$\frac{\partial}{\partial x}u(x) \approx \frac{u(x+1) - u(x)}{\Delta x} \tag{5.3}$$

Where $\Delta x = s$ is our step size and $x + 1$ is the grid point to right of of grid point x. We show these elements on the grid in Figure 5.2

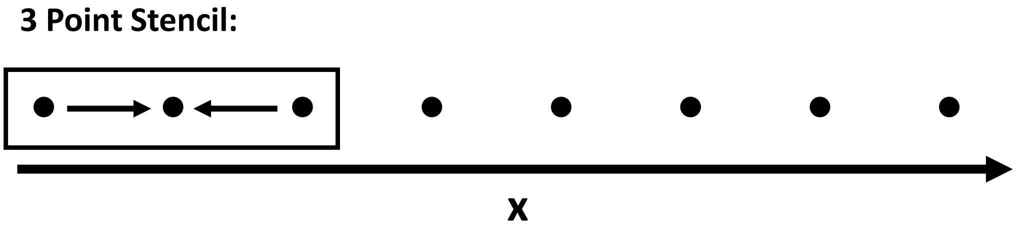


Figure 5.2: We can approximate the derivative of the center element using the values of the left and right neighbors.

In order to numerically solve the heat equation, we also need to compute the second derivative. In the following equation, we compute it using a central difference approximation:

$$\frac{\partial^2}{\partial x^2}u(x) \approx \frac{u(x+1) - 2u(x) + u(x-1)}{2(\Delta x)^2} \tag{5.4}$$

These equation (5.2.1 and 5.2.1) are by no means the only approximations and other expansions can be used depending on the desired error. Using these approximations, we can describe Equation 5.2.1 numerically.

$$\frac{u(x,t+1)-u(x,t)}{\Delta t} = h\frac{u(x+1,t)-2u(x,t)+u(x-1,t)}{2(\Delta x)^2} \tag{5.5}$$

To simplify this expression we are going to set $\Delta x = \Delta t = s$ and rewrite in terms of $u(x,t+1)$:

$$u(x,t+1) = h\frac{u(x+1,t)-2u(x,t)+u(x-1,t)}{s} + u(x,t) \tag{5.6}$$

If we were to describe the problem in terms of linear system of equations, then we will see that this admits a regular structure, namely a banded matrix. For the sake of simplicity we will set $s = 1$.

$$A = \begin{pmatrix} h & 2h & & & & \\ h & 2h+1 & h & & & \\ & h & 2h+1 & h & & \\ & & \ddots & \ddots & \ddots & \\ & & & h & 2h+1 & h \\ & & & & 2h & h \end{pmatrix} \tag{5.7}$$

We will immediately see that this structure will come in handy in reducing the amount of computation needed on operations over A. For example, if we want to compute a time step we can simply do a matrix vector product $x^{(t+1)} = Ax^{(t)}$. In the general case this would require $O(n^2)$ operations. However, our matrix is tridiagonal, so we can reduce the computation down to $O(n)$. We capture this in the inner most loop of the following listing:

```
for( t = 0; t < TSTEPS; ++t )
 for( i = 1; i< L; ++i )
  x[i][t+1] =      h * x[i-1][t] +
               (2h+1) * x[i][t]   +
                    h * x[i+1][t];
```

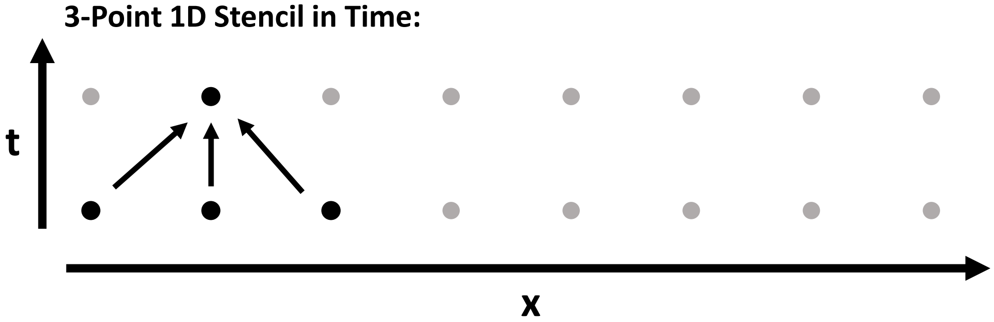


Figure 5.3: The calculation of an element in the subsequent time step depend on the neighbors in the previous time step.

In this listing, we added an extra loop for traversing through the time dimension. We can visualize each iteration as a stencil over the elements in Figure 5.3

We can imagine that the entire computation can be viewed as the application of this stencil on every data point. We will call this the 3-pt 1-D stencil. If we want, we can view the iteration space of multiple iterations applied through time. An important point to note is the computation of an element only depends on its neighbors in the previous iteration

However, if we have measured a set of temperatures and we want to find the starting values, then we must solve $b = Ax$ where b are our ending values and x are our unknowns. We can find this solution iteratively using Jacobi's method.

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \tag{5.8}$$

Where $A = D + R$, D is a diagonal matrix and R has 0 for each of its diagonal elements. This expression is computed iteratively until $\|x^{(k+1)} - x^{(k)}\| < \epsilon$. We can break this matrix equation into summations in terms of elements of A:

$$\chi_i^{(k+1)} = \frac{1}{\alpha_{ii}}(\beta_i - \sum_{i \neq j} \alpha_{ij}\chi_j^{(k)}) \tag{5.9}$$

Once again, we take advantage of the fact that A has a very specific structure in the context of our finite difference approximation of the heat equation. Using this fact we can rewrite Equation 5.9 as follows:

$$\chi_i^{(k+1)} = \frac{1}{2}(\beta_i - \chi_{i+1}^{(k)} - \chi_{i-1}^{(k)}) \tag{5.10}$$

Using this specialization significantly reduces the amount of computation needed by the Jacobi method. We sketch this code in the following listing:

```
/* time dimension */
for( t=0; t<NITER; ++t)
  /* space dimension  */
  for( i=1; i<L; ++i )
    x[i][t+1] =
      0.5 * (b[i]-x[i+1][t]-x[x-1][t]);
```

In summary, the process of translating a problem into a stencil begins the problem statement as a set of relations, in this case a PDE. We then discretize the problem space and express the approximation of our original relations in terms of elements on this grid. This allows us to express the approximation of problem as a system of equations. If we want to implement this as code we can use the structure of the stencil to minimize the amount of work needed to perform the operation. In the following section, we will discuss a variety of optimizations for improving the performance of stencil computations.

5.3 Theory

In this section, we will discuss the key optimizations necessary for achieving high performance on stencil computations. This includes spacial blocking, temporal blocking (time-tiling) and SIMD vectorization. For our examples, we will use a one dimensional 3-point stencil and a 9-point two dimensional stencil. In Figure 5.4, we show two options for two dimensional stencils. We will write these stencils with the coefficients in an array that we traverse in the inner-most loop.

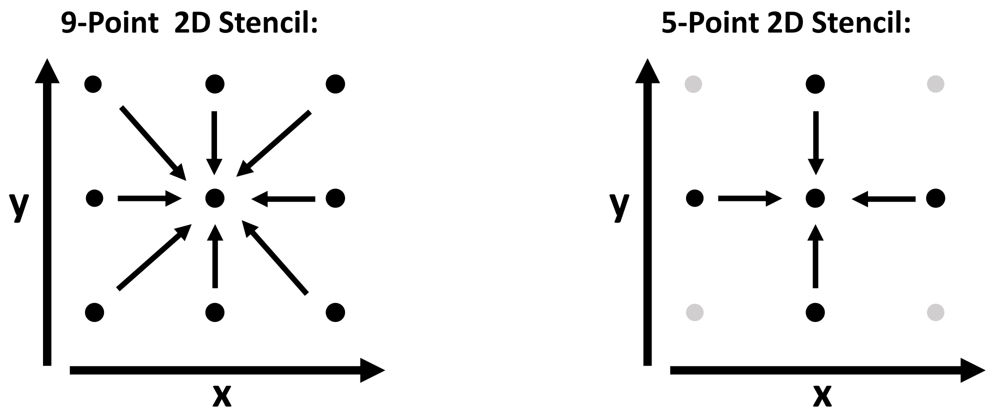


Figure 5.4: In these two 2-Dimensional stencils the value of the center element is dependent on the values of its neighbors.

```
coef[v] = {h,2h+1,h};
/* time dimension */
for( t = 0; t < TSTEPS; ++t )
  /* space dimension */
  for( i = 1; i< L; ++i )
    /* 3-point 1D stencil */
    for( p = 0; p < v; ++p )
      x[i][t+1] += coef[p] *
                   x[i+p-1][t]
```

5.3.1 Spacial Blocking

Modern computer architectures utilize deep memory hierarchies to compensate for the speed gap between the faster processor and the slower memory. In order to take full advantage of these caches, it is necessary to block – or tile – our code in the spacial dimension. Just like the matrix-matrix multiplication case, the operation is partitioned such that a small, cache-resident sub-problem is computed completely before moving on to this next block. To illustrate this transformation, it makes more sense to consider the two dimensional case of 9-point stencil.

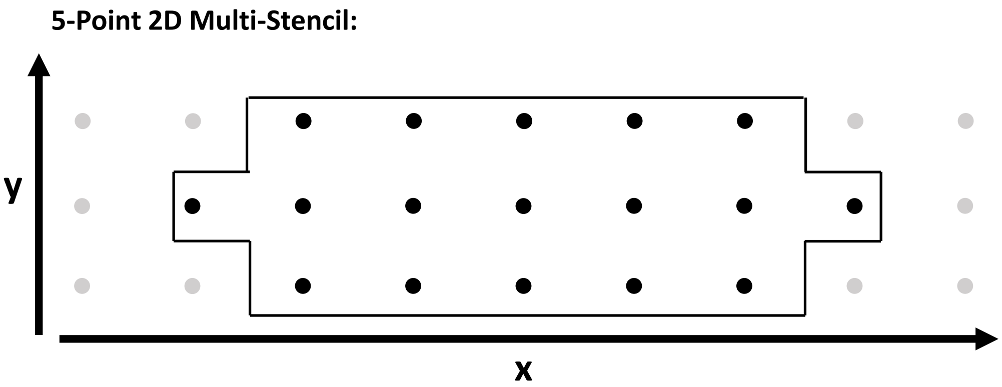


Figure 5.5: A multistencil is the application of a stencil on a block of elements.

```
/* time dimension */
for( t = 0; t < TSTEPS; ++t )
  /* space dimensions */
  for( i = 1; i< M; ++i )
    for( j = 1; j< N; ++j )
      /* 9-point 2D stencil */
      for( p = 0; p < v; ++p )
        for( q = 0; q < w; ++q )
          X[i][j][t+1] += coef[p][q] *
                            X[i+p-1][][t]
```

In this example, the coefficients of the stencil are stored in the two dimensional matrix labeled *coef*. The input is a matrix $X \in \mathbb{R}^{M \times N}$ and its spacial dimensions indexed by the variables i and j. We can add two additional loops to the spacial dimensions of the input matrix to create the following multistencils.

```
/* time dimension */
for( t = 0; t < TSTEPS; ++t )
  /* outer space dimensions */
  for( i = 1; i< M; i+=MB )
    for( j = 1; j< N; j+=NB )
      /* inner space dimensions (multistencil) */
      for( ii = 0; ii< LB; ++ii )
        for( jj = 0; jj< MB; ++jj )
          /* 9-point 2D stencil */
          for( p = 0; p < v; ++p )
            for( q = 0; q < w; ++q )
              X[i][j][t+1] += coef[p][q] *
                              X[i+p-1][j+q-1][t]
```

In this example we block over X using $m_b \times n_b$ tiles. These dimensions would be selected to fit within the targeted cache level. It is easy to imagine that as we target more levels of cache the number of loops will increase. As we will see in the next section, these transformations will results in complex and costly index computations.

An important detail to note is that this pair of inner-most loops combined with the computation is a stencil in and of its own. We can view this as a non-tiled – or unblocked – stencil, and we can view the outer loops over this unblocked stencil as the blocked version of the stencil. This is called a multistencil Figure 5.5.

5.3.2 Temporal Blocking

Alone, spacial blocking only provides a marginal benefit. This is because the amount of reuses of each element in the input is dependent on the size of the stencil. For example, in the case of a 5-point stencil, each input element is only reused seven times and for a 9-point stencil only nine times. If we include the temporal dimension then we would get much more reuse of the spacial elements over multiple time steps. This would be *time tiling* or temporal blocking. The idea is that the computation is reorder such that correctness is preserved, however a small cache sized spacial block is computed for multiple time steps which we illustrate in Figure 5.6. Once the block is computed, then the region that overlaps with the next block – ghost region – is shared

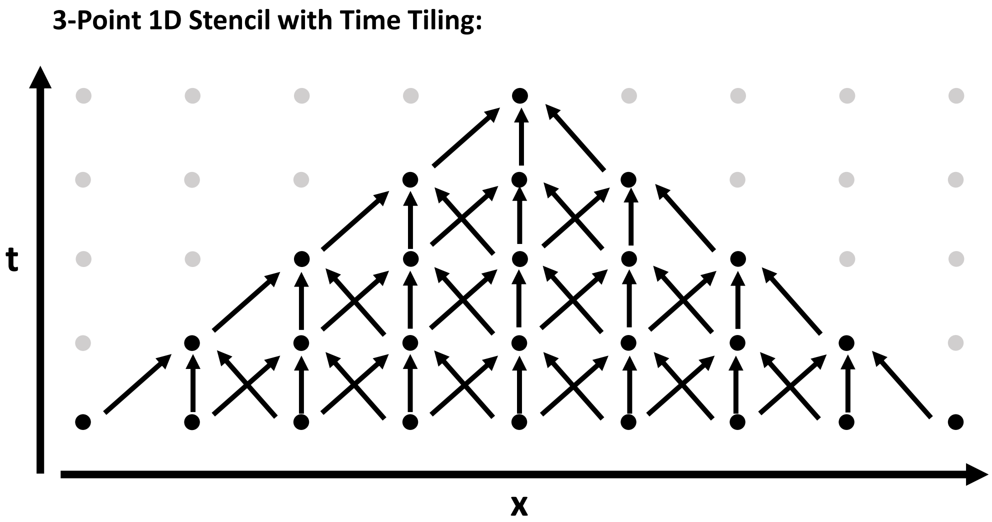


Figure 5.6: We can block a stencil in the time dimension –time tiling– provided that the dependencies are resolved in a correct order.

with the next block and it too is computed. We will demonstrate a time-tiled 1D stencil in Figure 5.7 in the following listing:

```
/* outer time dimension */
for( to = 0; to < TSTEPS; to+=d )
  for( xo = 0; xo < M; xo+=mb )
    /* start of an upward trapezoid */
    for( ti = 0; ti < d; ++ti )
      time    = to + ti;
     /* overlap between time tiles */
     start_x = xo + ti;
     end_x   = xo + mb + 2d - ti;
     for( xi = start_x; xi < end_x; xi++ )
       /* 3 point stencil */
       for( p = 0; p < 3; ++p )
         x[xi][time+1] += coef[p] *
                          x[xi+p-1][time]
```

In this listing, we block both the space dimension by m_b and the time

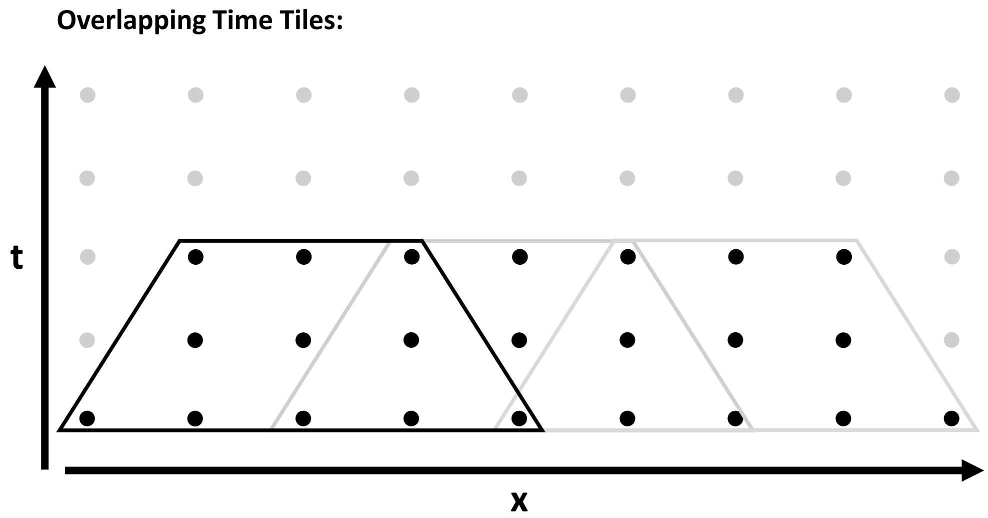


Figure 5.7: We can use multiple overlapping time tiles to block the operation in both the time and space dimension.

dimension by d, which adds loops over t_i and x_i. These inner loops create the time-tile. We can view this time tile as a whole stencil of its own that is marching along the outer time (t_o) and space dimension (x_o). If the blocking dimensions are small enough then this time-tile is computed exclusively in cache.

There are a wide variety of tile shapes meant to minimize communication between blocks or simplify computation (see Figure 5.8). In this case, we only showed the simplest strategy. Unfortunately, in this implementation of the upward facing trapezoid we must keep copies of the spacial elements over multiple time steps. If we are more deliberate with our application of time-tiles and allow the use of different shapes then we can minimize the overlap and thus the need to keep more than two copies in time of a given point in space. We illustrate this in Figure 5.9. In order to simplify how we express this in code, we will break the tiles into functions. We start with the one dimensional stencil:

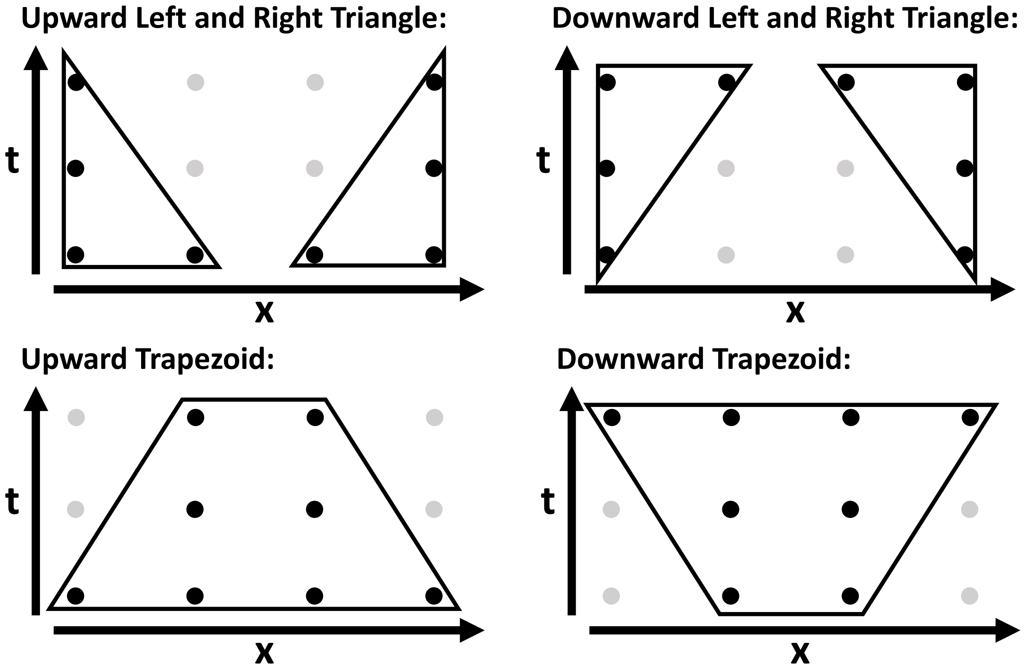

Upward Left and Right Triangle:
t
x
Downward Left and Right Triangle:
t
x
Upward Trapezoid:
t
x
Downward Trapezoid:
t
x

Figure 5.8: We can minimize redundant computation by mixing differently shaped time-tiled blocks.

```
inline unblocked_sten1d(x, time_cur, time_nxt, xi)
  /* 3 point stencil */
  for( p = 0; p < 3; ++p )
      x[xi][time_nxt] += coef[p] *
                         x[xi+p-1][time_cur]
```

As long as we minimize the overlap between tiles such that we are not recomputed values, then for this one dimensional stencil we only need two spacial dimensions. Next, we will describe downward trapezoids as the following function.

```
inline down_trap_sten1d(mb,db,*x,xo,to)
  /* start of a downward trapezoid */
  for( ti = 0; ti < db; ++ti )
   /* we only need two time copies */
   time_cur = (to + ti) % 2;
   time_nxt = (to + ti + 1) % 2;
   start_x = xo + db - 1 - ti;
   end_x   = xo + mb + db + 1 + ti;
   for( xi = start_x; xi < end_x; xi++ )
     unblocked_sten1d(x, time_cur, time_nxt, xi);
```

For completeness we will also include a upwards left sided triangle stencil (Figure 5.8) which will handle the edge case.

```
inline left_up_tri_sten1d(db,*x,xo,to)
  for( ti = 0; ti < db; ++ti )
   time_cur = (to + ti) % 2 ;
   time_nxt = (to + ti + 1) % 2;
   start_x = xo;
   end_x   = xo + d - ti;
   for( xi = start_x; xi < end_x; xi++ )
     unblocked_sten1d(x, time_cur, time_nxt, xi);
```

The original upward trapezoid and downward right sided triangle can also be implemented in this fashion. Additionally, let us assume that the boundaries are handled by separate stencils that use either forward or backward differences to compute their elements. Lastly, we are restricting By combining these time-tiled stencils blocks, we can re-implement our original

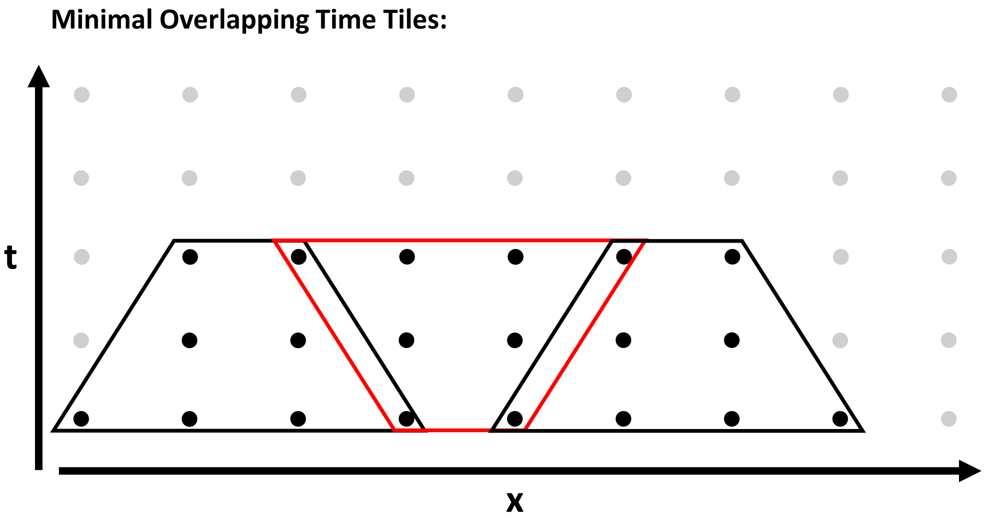


Figure 5.9: We can alternate between upward and downward trapezoidal tiles to minimize redundant computation.

unblocked stencil into a spacial and temporal blocked version:

```
void blocked_sten1d(x,M,TSTEPS,mb,db)
 for( to = 0; to < TSTEPS; to+=db )
  /* edge cases */
  left_up_tri_sten1d(db,x,0,to);

  /* steady-state */
  for( xo = db; xo < M-mb; xo+=(mb+2*db) )
    up_trap_sten1d(mb,db,x,xo+mb,to);
    for( xo = db+mb; xo < M-mb; xo+=(mb+2*db) )
      down_trap_sten1d(mb,db,x,xo,to);

  /* epilog*/
  right_dwn_tri_sten1d(db,x,M-mb,to);
```

In this blocked implementation, we have broken the time-tile into two parts: the edge cases that computes enough values of x for a steady-state, which in turn computes until there are no longer enough values to continue the pattern. If m_b and d_b are appropriately selected we can insure that much of the stencil computation operates within the cache and performs near the processor's peak performance. An important note is that we can recursively apply both spacial and temporal blocking on our stencil. This is necessary, if we wish to block for multiple levels of cache.

Time-tiling is not an optimization that works in isolation. Several additional components are needed in order to maximize its benefit: First, we need to incorporate spacial blocking in order to have enough elements to reuse over multiple time dimensions. Second, we need to reuses vectors for each time dimension in order to benefit from the cache. While this is not absolutely necessary, it increases our chances of cache hits and minimizes traffic to main memory. Third, index computation becomes complicated as we add additional spacial and temporal dimensions. Therefore, transformations such as unrolling and code motion help alleviate the indexing bottleneck. Last, we can maximize cache utilization and reduce cache misses if we perform a data layout transformation by packing our input vector in a manner that guarantees unit stride access.

5.3.3 SIMD Vectorization

In order to perform more useful work per instruction, most modern architectures provide short vector instructions or, Single Instruction Multiple Data (SIMD) units. These instructions are necessary in order to fully utilize the peak floating point capacity of the target system. However, efficiently utilizing them is not necessarily straight forward and typically involves additional transformation. For example, in the following code listing we demonstrate a naive SIMD implementation which illustrates the implementation in Figure 5.10.

```
for( t = 0; t < TSTEPS; ++t )
  for( xo = 0; xo< L; xo+=v )
    for( p = 0; p < w; ++p )
      #pragma vectorize(v)
      for( xv = 0; xi< v; xv++ )
        x[xo+xv][t+1] += coef[p] *
                         x[xo+xv+p-1][t]
```

While this approach works, it has a severe limitation, each element is loaded v times. Not only are these loads redundant, but they are also costly. A typical modern microarchitecture contains a limited number of memory

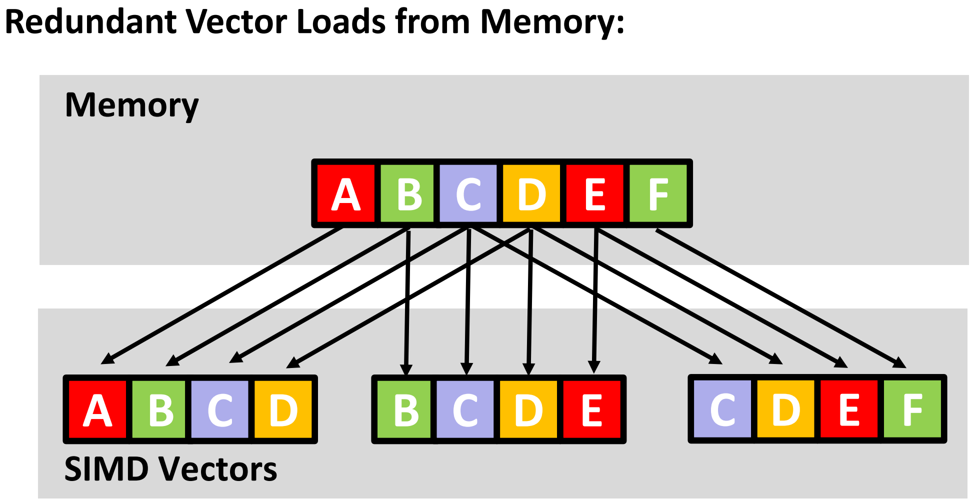


Figure 5.10: SIMD instructions allow us to maximize the number of floating point operations per cycle, however a naive implementation results in a redundant number of costly loads.

functional units, and in this scenario they will become the bottleneck. Ideally, an element should only be loaded once.

A DLT Inside the Vectors: To address these redundant loads, we introduced a method for performing data layout transformations within vectors. An element is loaded once, but reordered multiple times within SIMD vector registers. We illustrate this in Figure 5.12, 5.13, 5.14 and 5.15 and in the following listing:

```
for( t = 0; t < TSTEPS; ++t )
   vblock_left   = vload(&x[t][0]);
   vblock_cent   = vload(&x[t][v]);
   vblock_right  = vload(&x[t][2*v]);
   /* steady-state */
   for( xo = v; xo< L; xo+=v )
     /* transform layout in vectors */
     xform_left  = shift_left(vblock_left,vblock_cent,v);
     xform_cent  = block_cent;
     xform_right = shift_right(vblock_cent,vblock_right);

     /* perform stencil */
     vres = c[0]*xform_left + c[1]*xform_cent + c[2]*xform_right;
     vstore( x[t+1][xo], vres);

     /* get next blocks */
     vblock_left  = vblock_cent;
     vblock_cent  = vblock_right;
     vblock_right = vload(&x[t][2*v+xo])
```

At each iteration of the steady-state three transformed vectors are filled with shifted data from the vectors blocks `vblock_left`, `vblock_cent` and `vblock_right`. The shifting functions concatenate two vectors and returns a shifted sub-vector. For example, `shift_right` takes two vectors c and r and returns $c = [c_1, \ldots, c_{v-1}, r_0]$. Similarly, `shift_left` takes l and r and returns $c = [l_{v-1}, c_0, \ldots, c_{v-1}]$. These functions are implemented using vector shuffle and permutation instructions. Once the vectors blocks are shifted then the stencil can be computed. After the operation is performed then the next vector block is loaded. The net result is that we trade off expensive vector loads for less expensive vector permutation instructions.

While we focus on reshaping the data within the SIMD vectors, the authors in [41] provide an excellent example of how we can reshape the input data within memory to facilitate the use of aligned SIMD loads.

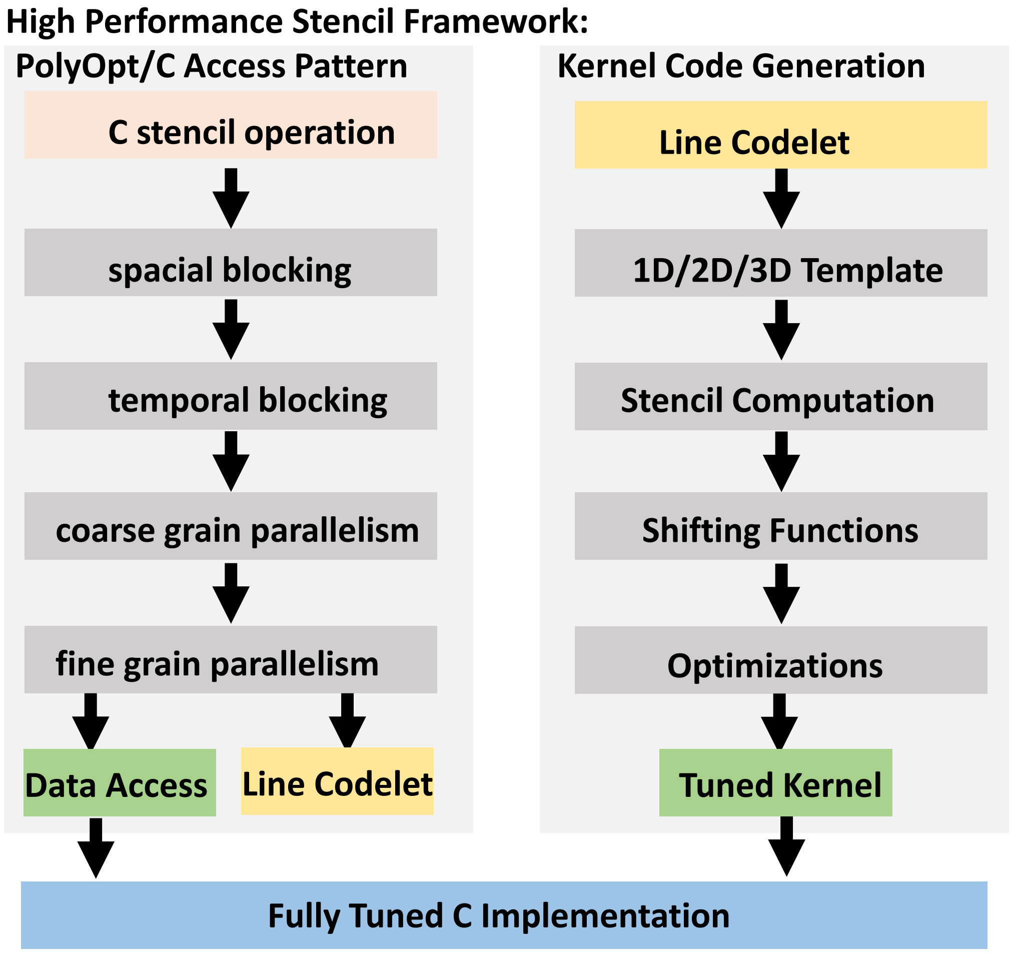


Figure 5.11: Our stencil code generator is composed of two parts: First, a compiler based framework which takes an input stencil as C code and uses polyhedral analysis to generate an spacial and temporally blocked access pattern that is amenable to fine and coarse grain parallelism. Second, a kernel code generator which generates highly tuned vectorized kernel code for the access pattern. The result of these two processes are combined in order to produce a high performance implementation.

5.4 Mechanism

In the previous section, we discussed the optimizations which are necessary for implementing a high performance stencil operation. In this section, we will describe our framework for generating arbitrary stencil operations using these transformations. We accomplish this by splitting the problem in two parts, data movement and kernel generation. In the first part, the operation is expressed in terms of simple C implementation. This implementation passed through the PolyOpt/C compiler which uses polyhedral analysis to perform spacial and temporal blocking, followed by transformations for coarse grain parallelism and fine grain SIMD parallelism. This described in more detail in [40] and [71]. The second part of the generator replaces the *line codelet*, or inner-most stencil, with an automatically generated kernel using the SPIRAL framework [72]. We detail this complete process in Figure 5.11.

5.4.1 Kernel Generation

The first part of our framework provides an access pattern that allows a kernel to efficient compute on aligned, cache resident data. Just like the matrix-matrix multiplication case, it is up to the kernel to achieve the performance that the access pattern allows. Our stencil kernel generator takes a template of the desired kernel along with architectural parameters and produces a high tuned implementation within the SPIRAL framework. We can break this process up into the three parts.

A Templated Kernel. The first part of kernel generation process starts with a template of the stencil's loop structure and data access pattern. This template takes as an input the unit stencil computation and an *architecture container*. Using these two pieces, we can create an untuned stencil kernel. In the following listing, we illustrate the template for the one dimensional kernel. For the higher dimensional kernels we add additional loops.

```
inline template_generic_1d_sten(x,L,T, STEN, arch)
  for( t = 0; t < T; ++t )
    vb_left = arch.VLOAD(&x[t][0]);
    vb_cent = arch.VLOAD(&x[t][arch.V]);
    vb_rght = arch.VLOAD(&x[t][2*arch.V]);
    /* steady-state */
    for( xo = v; xo< L; xo+=arch.V )
      /* shift elements in vectors */
      xf_left = arch.SHFT_LEFT(vb_left,vb_cent,vb_rght);
      xf_cent = arch.SHFT_CENT(vb_left,vb_cent,vb_rght);
      xf_rght = arch.SHFT_RGHT(vb_left,vb_cent,vb_rght);

      /* perform stencil */
      vres = STEN(xf_left, xf_cent, xf_rght,arch);
      arch.VSTORE(x[t+1][xo],vres);

      /* get next blocks */
      vb_left = vb_cent; vb_cent = vb_rght;
      vb_rght = arch.VLOAD(&x[t][2*v+xo])
```

This template captures our approach to SIMD vectorization, where elements are only gathered once. The template first gathers vector blocks, then performs the steady state loop where elements are shifted using an architecture specific shifting functions. Once the elements are shifted they are passed to the stencil computation. The following listing shows an implementation of the 1D Jacobi for use in the template.

```
inline generic_1d_jacobi(left,cent,rght,arch)
  sum = arch.ADD(left,cent,right);
  return arch.MUL(0.33f,sum);
```

This unit of computation is expressed in terms of the right, center and left element of the stencil along with the architecture specific parameters. This computation is portable across any platform as long as the architecture container expresses the requisite operation.

Parameterized SIMD Mapping. The bulk of the performance for our kernels depends on the implementation of the *shifting functions*. These in-

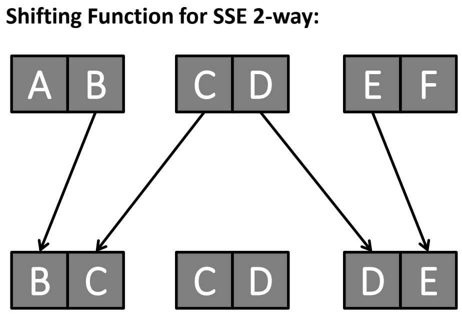


Figure 5.12: We can minimize the need for redundant vector loads through the use of shifting functions which load the data once and rearrange the elements within SIMD registers. This illustration shows the implementation of a shifting function for SSE 2-way SIMD instructions.

sure that we only load an from memory to register once. However, their implementation is performance critical because at least one shifting function occurs for each computation. Thus to insure that the shifting function does not become the bottleneck, it must sustain a throughput greater than or equal to the computation. In the next listing we show a simple example of the shifting functions for 2-way SSE instructions. This is the function shown in Figure 5.12.

```
SSE_2x64f.SHIFT_LEFT:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   vshuffle_2x64f(BLOCK_LEFT,BLOCK_CENT, [2,1]);
SSE_2x64f.SHIFT_CENT:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   vshuffle_2x64f(BLOCK_LEFT,BLOCK_CENT, [2,1]);
SSE_2x64f.SHIFT_RGHT:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   vshuffle_2x64f(BLOCK_CENT,BLOCK_RGHT, [2,1]);
```

In this listing, we provide a mapping between the shifting functions and architecture specific instructions. This architecture is the simplest case, as we can express each shift in terms of a `shuffle` instruction. This mapping is expressed using a LISP like intermediate language inside SPIRAL. We can extend this to 4-way SSE instructions as shown in Figure 5.13:

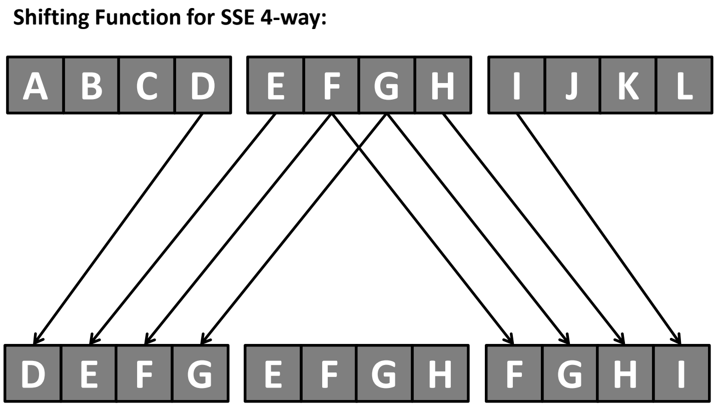


Figure 5.13: In the case of the SSE 4-way shifting function we can use the instruction `palignr` which allows us to concatenate two vectors and extract a contiguous sub-vector.

```
SHIFT_LEFT_SSE_4x32f:= (BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
     alignr_4x32f(BLOCK_LEFT,BLOCK_CENT, (3)*4) );
SHIFT_CENT_SSE_4x32f:= (BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
     BLOCK_CENT;
SHIFT_RGHT_SSE_4x32f:= (BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
     alignr_4x32f(BLOCK_CENT,BLOCK_RGHT, (1)*4) );
```

In the 4-way SSE case, we can use the **`alignr`** instruction, which concatenates two inputs and returns a shifted sub-vector of the concatenation. While this is a convenient instruction, it is not available for longer SIMD width. Next, we will show how the complexity of these shift functions increases as SIMD width increases. In the following listings, we show the AVX 4-way and 8-way case, which we illustrate in Figure 5.14 and Figure 5.15:

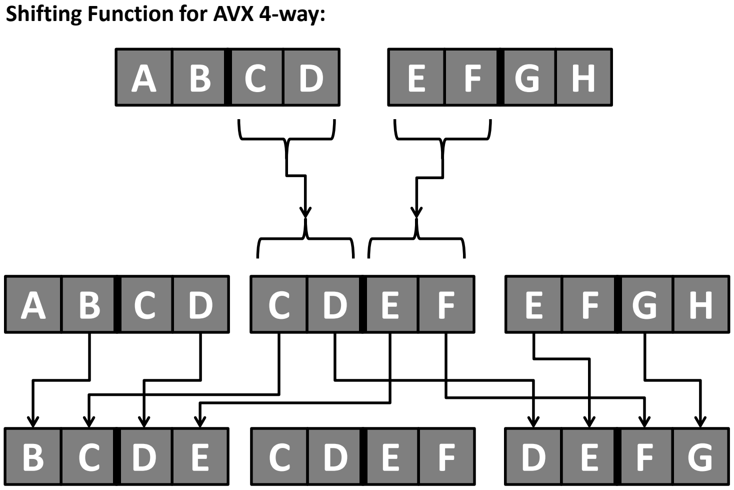


Figure 5.14: As we move from SSE to AVX we are confronted with the issue of moving elements between lanes (upper and lower half of the vector) and within lanes. Because no AVX instruction can move within and between lanes we first move blocks across lanes then within.

```
SHIFT_CENT_AVX_4x64f:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   vpermf128_4x64f(BLOCK_CENT,BLOCK_RGHT, [2,1]);
SHIFT_LEFT_AVX_4x64f:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   let(
   tmp:=SHIFT_CENT_AVX_4x64f(BLOCK_LEFT,
                           BLOCK_CENT,BLOCK_RGHT),
   vshuffle_4x64f(BLOCK_CENT,tmp,[2,1,2,1]));
SHIFT_RGHT_AVX_4x64f:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   let(
   tmp := SHIFT_CENT_AVX_4x64f(BLOCK_LEFT,
                             BLOCK_CENT,BLOCK_RGHT),
   vshuffle_4x64f(tmp,BLOCK_RGHT,[2,1,2,1]));
```

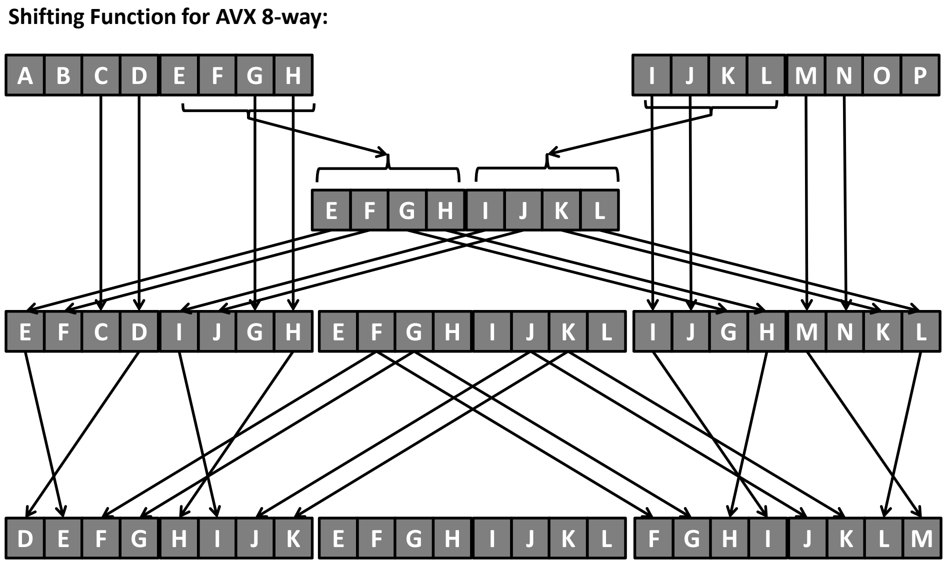


Figure 5.15: Here we show the 8-way AVX shifting function. Note that the longer the SIMD vector width, the more complicated these shifting functions become.

```
SHIFT_CENT_AVX_8x32f:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   vpermf128_8x32f(BLOCK_CENT,BLOCK_RGHT, [2,1]);
SHIFT_LEFT_AVX_8x32f:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   let(
   tmp1 := SHIFT_CENT_AVX_8x32f(BLOCK_LEFT,
                            BLOCK_CENT,BLOCK_RGHT),
   tmp2 := vshuffle_8x32f(tmp1, BLOCK_CENT,
                      [4,3,2,1,4,3,2,1]),
   vshuffle_8x32f(tmp2,tmp1,[3,2,1,4,3,2,1,4]));
SHIFT_RGHT_AVX_8x32f:=(BLOCK_LEFT,BLOCK_CENT,BLOCK_RGHT)->
   let(
   tmp1 := SHIFT_CENT_AVX_8x32f(BLOCK_LEFT,BLOCK_CENT,
                            BLOCK_RGHT),
   tmp2 := vshuffle_8x32f(BLOCK_RGHT,tmp1,
                       [4,3,2,1,4,3,2,1]),
   vshuffle_8x32f(tmp1,tmp2,[1,4,3,2,1,4,3,2]));
```

These AVX implementations are substantially more complex than their SSE counterparts. This is a result of how AVX registers are partitioned into lanes. Instructions are split between those that move elements within lanes and those that move elements across lanes. Thus, in order to shift elements within a SIMD register a combination of these instructions are necessary.

Further Optimizations. The previous two steps build an unoptimized stencil kernel with the desired SIMD instructions. The final step is to optimize these code such that the performance is only constrained by the computation and not by the overhead. The key transformation performed on this unoptimized code include loop unrolling, array scalarization and Common Subexpression Elimination (CSE). First, both the spacial and temporal loops are unrolled to minimize loop overhead and to simplify index computation. For these stencils, index computation can become a substantial source of overhead and unrolling mitigates this by replacing loop variables with constants. We can demonstrate this transformation below:

```
#pragma unroll
for( xi = 0; xi < 2; ++xi )
  y[xo+xi]=x[xo+xi-1]+x[xo+xi]+x[xo+xi+1];
```

In this previous listing each index computation requires the addition two variables and a constant. If we left this as a loop we would have ten index computations per iteration. However, if we unroll it we can reduce this computation significantly to three operations per iteration.

```
y_p = &y[0]; x_p = &x[0];
y_p[0]=x_p[-1]+x_p[0]+x_p[1];
y_p[1]=x_p[0] +x_p[1]+x_p[2];
```

By combining unrolling with simple indexing optimization we can simplify all memory accesses to a base address plus an offset. Now that indexing is no longer the bottleneck the next issue that emerges is memory access. Each iteration performing $y_i = x_{i-1} + x_i + x_{i+1}$ requires four memory operations. However, many of these elements are reused because of spacial blocking and temporal blocking. But in order to take advantage of this, we need to store these elements in registers. In our next transformation, local arrays – namely the intermediate values of the input and output arrays – are replaced with variables. This reduces memory traffic by storing these intermediate values in registers. We can visualize this transformation in the following listings:

```
/* read into temp buffers */
for( xi = -1; xi < 3; ++xi )
  in[xi+1] = x[xi];

/* steady-state */
for( xi = 0; xi < 2; ++xi )
  out[xi]=in[xi]+in[xi+1]+in[xi+2];

/* write out */
for( xi = 0; xi < 2; ++xi )
  y[xo+xi] = out[xi];
```

In this listing, the input elements are stored in a temporary array in, the intermediate result is computed and stored in in, and those values are eventually written out. This transformation only introduces many more memory accesses, but like many other optimizations this is is used in conjunction with other transformations. If we unroll all three loops and replace the temporary arrays with variables.

```
/* read into temp buffers */
in_0 = x[xo-1];
in_1 = x[xo+0];
in_2 = x[xo+1];
in_3 = x[xo+2];

/* steady-state */
out_0 =in_0+in_1+in_2;
out_1 =in_1+in_2+in_3;

/* write out */
y[xo+0] = out_0;
y[xo+1] = out_1;
```

Provided we have a sufficient number of registers, we can use these temporary variables without having to incur memory traffic. These two optimizations – unrolling and array scalarization – help insure that the performance is dependent only on the computation. Thus, the next transformation reduces the amount of computation needed for certain stencils. In the previous listing, we see that certain computations are repeated between stencils. By using Common Subexpression Elimination (CSE), we can eliminate redundant computation by computing an operation once and reusing its result multiple times. For example, we can eliminate an addition from the previous steady-state computation:

```
/* steady-state */
t_1p2 =in_1+in_2;
out_0 =in_0+t_1p2;
out_1 =t1p2+in_3;
```

The effectiveness of this transformation is dependent on the stencil being used. In the general case it may not provide a benefit. The stencil kernel generation and these optimizations are performed inside of the SPIRAL framework. In particular the optimizations are handled by a built in optimizing compiler. The result of the kernel generator is a highly optimized SIMD kernel implemented in C. This kernel is then placed inside the loop nesting that is provided by the PolyOpt/C compiler. In the next section we will evaluate the performance of the code resulting from this process.

5.5 Experiments

In this chapter, we have discussed where stencil computation arise, what transformations are necessary for their performance and how to systematically produce a high performance implementation. This section is devoted to the performance evaluation of this approach. We will provide an empirical and an analytical evaluation of our approach. In the empirical test we will show that for real world stencil computations we outperform the state of the art and in the analytical analysis we will show how close our kernels are to the peak theoretical performance.

5.5.1 Empirical Results

In this experiment, we will compare the performance of the stencil code generated by our framework – with and without the tuned kernels – against a baseline implementation and the PTile compiler [73]. Both our approach and PTile output an implementation in C code which is then compiled using

Stencil	t dim.	x dim.	y dim.	z dim.
Jacobi 2D	20	2000	2000	–
Laplacian 2D	20	2000	2000	–
Poisson 2D	20	2000	2000	–
Jacobi 3D	20	256	256	256
Laplacian 3D	20	256	256	256
Correlation	–	2000	2000	2000
Covariance	–	2000	2000	2000
Doitgen	–	256	256	256

Table 5.1: This table contains the stencils and their corresponding block sizes.

the Intel Compiler. These experiments were performed on an Intel Core i7-2600K (Sandy Bridge) which has four cores, each running at 3.4 GHz. The overall peak performance of this chip is 108.8 GFLOP/s for double precision and 217.6 GFLOP/s for single precision. We selected kernels from the PolyBench collection [71] which represent stencil computations from a variety of domains including physics (two and three dimension Laplacian and Poisson Solver), probability (covariance), signal processing (correlation) and linear algebra (doitgen). We list the problem sizes used for these experiments in Table 5.1. We break the results in four plots for each of the vector widths, 2-way SSE double precision, 4-way SSE single precision, 4-way AVX double precision and 8-way SSE single precision.

In Figure 5.16 and Figure 5.17, we show the results our stencils using SSE instructions. In each of these plots we show the baseline operation compiled both sequentially and in parallel. We also show the performance of our implementation with the data access alone along with the data access combined with our generated kernels. In the SSE case the addition of the kernels adds between a 1.5× to 2× improvement over the data access pattern alone. In Figure 5.18 and Figure 5.19 are the results for AVX instructions. In these two cases the addition of the generated kernels provide an even greater speed up over the access pattern than in the SSE case.

Overall, our generated implementations outperform those produced by PTile for most stencils on most SIMD data types. However, in the 3 dimensional cases our implementation does not perform as well as PTile. Our approach tends to do the best where arithmetic is greater than data. This is

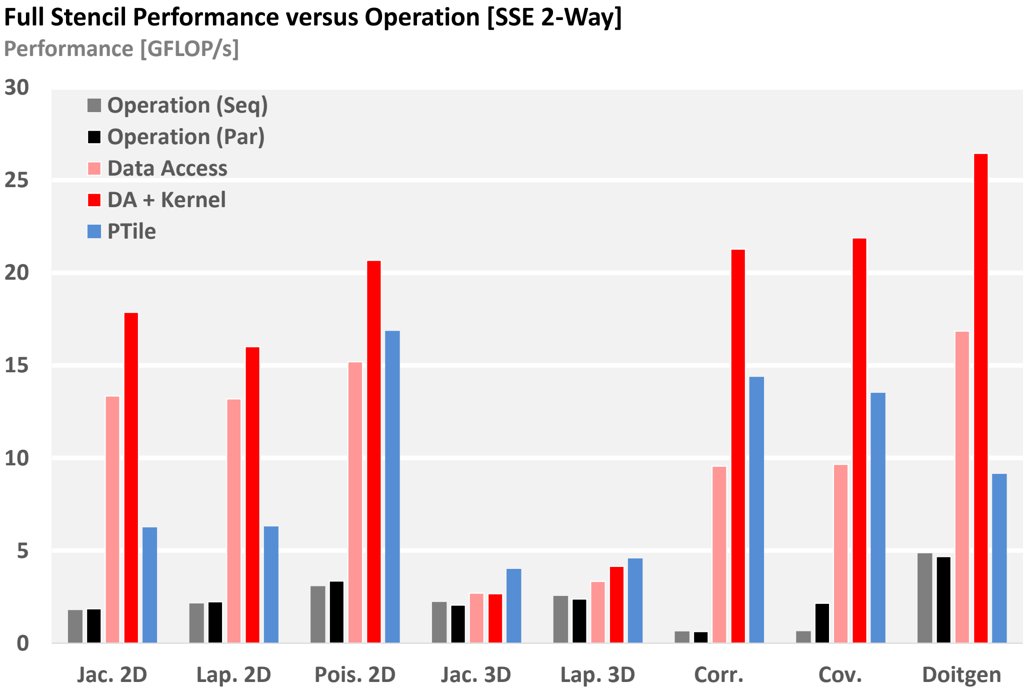


Figure 5.16: In this and Figures 5.17, 5.17 and 5.17 we compare the performance of our generated code against PTile for 8 different stencils. For each stencil we show several bar plots: The first – Operation (Seq.) – is the baseline stencil operation written in C and compiled with full sequential optimizations. The second bar – Operation (Par.) is the same baseline C implementation compiled with automatic parallelization. The third line – Data Access – is our generated implementation without a tuned kernel. This included spacial and temporal blocking and fine and coarse grain parallelism. The fourth line – DA+kernel – is our access pattern combined with a highly tuned kernel. The fifth line represents the baseline C code compiled with PTile. With the exception of the 3D kernels we outperform PTile for SSE 2-way instructions.

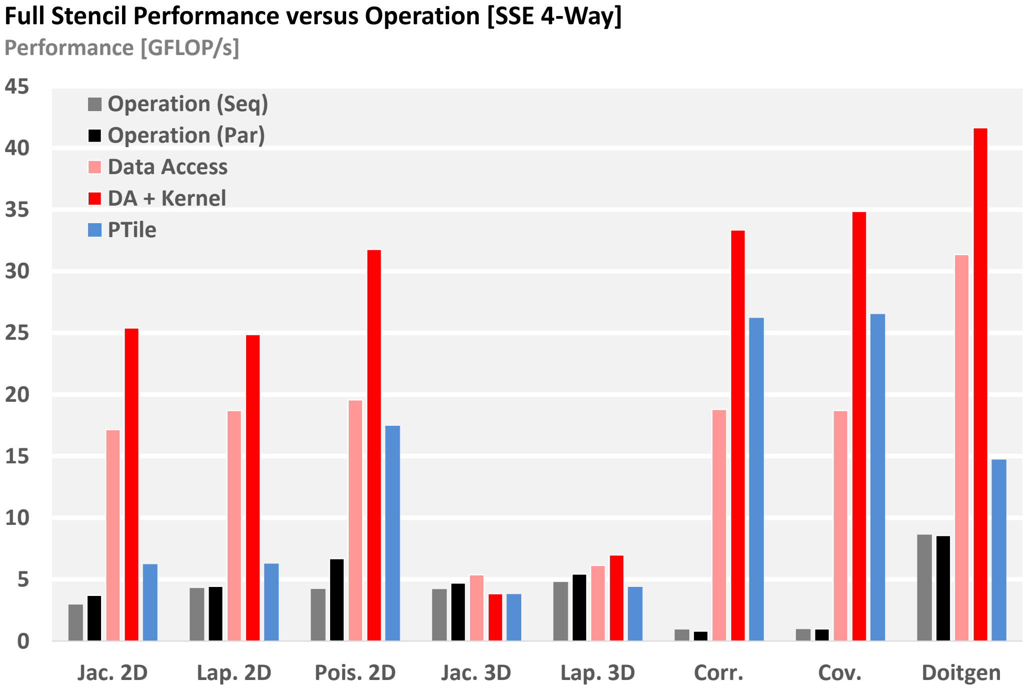


Figure 5.17: In the SSE 4-way case we outperform PTile for every stencil. However, our kernel reduces the performance for the Jacobi 3D stencil.

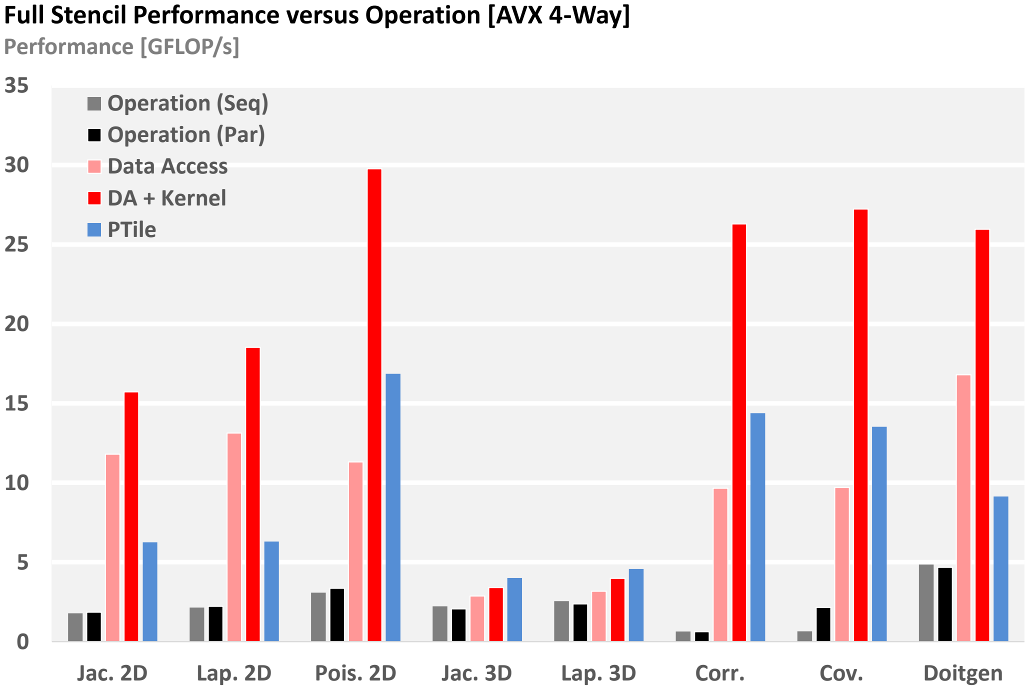


Figure 5.18: With the exception of the 3 dimensional stencils we outperform PTile.

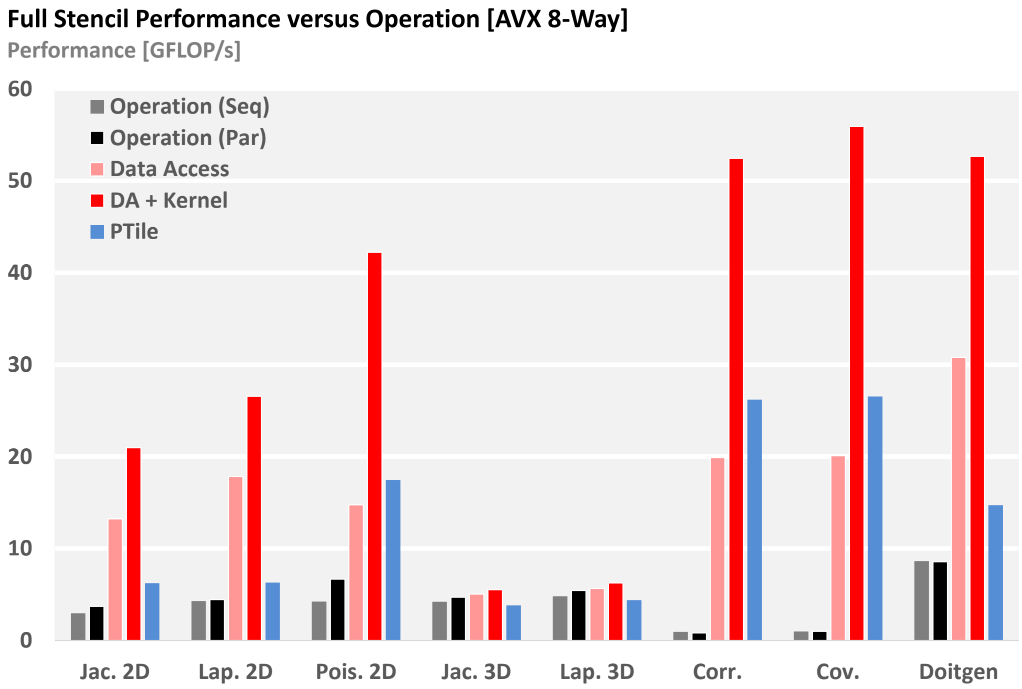


Figure 5.19: In the AVX 8-way case our performance is substantially better than PTile.

because modern architectures are constrained by memory access. As we add more dimensions we need more registers to hold reusable values or we are forced to spill values to memory. However, on these machines we are limited in the number of available to us.

5.5.2 Analytic Model

In addition to the empirical results, we provide an analytical analysis of the kernel portion of our stencil code. Here we determine how effective our kernels are at approaching the peak theoretical performance. For this analysis we use a very simple model to determine the maximum throughput based on bottlenecks in the processors functional units. The idea is we can characterize a kernel by the number of each type of SIMD instruction (n_i), where the type (i) is based on which functional units will compute that function. Given this characterization, we can estimate the bottleneck by finding the minimum quotient of the number of functional units of a particular type (f_i) and the number of instructions going to that unit (n_i). If we take that bottleneck and multiply it by the number of floating point operations in that particular stencil (k) then we get the estimated performance. More formally we have:

$$r = k \min_i (\frac{f_i}{n_i}) \tag{5.11}$$

If we were to take the 7-point 3-D Jacobi stencil using SSE instructions on the Intel Core i7-2600K our bottleneck would be the addition instructions ($n_{\texttt{add}} = 6$) because only one functional unit can service it ($f_{\texttt{add}} = 1$). The number of floating point operations for this stencil using SSE instructions would be $j = 14$ because we have 6 additions and 1 multiplications using 2-way vectors. This would give us a theoretical peak of 2.33 Floating Point Operations per cycle. We show these instruction counts for several stencils in Table 5.2. In Figure 5.20 we compare the theoretical peak of several stencils using our analytic model against an empirical benchmark of these kernels. We show this for two machines, the Intel Core i7-2600K (Sandy Bridge) and an Intel Xeon X5680 (Nehalem). It is worth noting that our model does not consider the Common Subexpression Elimination optimization where some redundant floating point instructions are removed. This leads to a lower theoretical estimate than what some of our kernels achieve. We chose not to include this in the model because it is highly dependent on the kernel being used and whether or not this transformation is applicable.

Stencil	# of Points	Adds	Muls.	Mem.	Shuffles
Jacobi 1D	3	2	1	2	1 – 5
Jacobi 2D	7	4	1	2	1 – 5
Jacobi 3D	9	6	1	2	1 – 5

Table 5.2: In this table we show the type and number of instructions needed per operation to compute the given stencils.

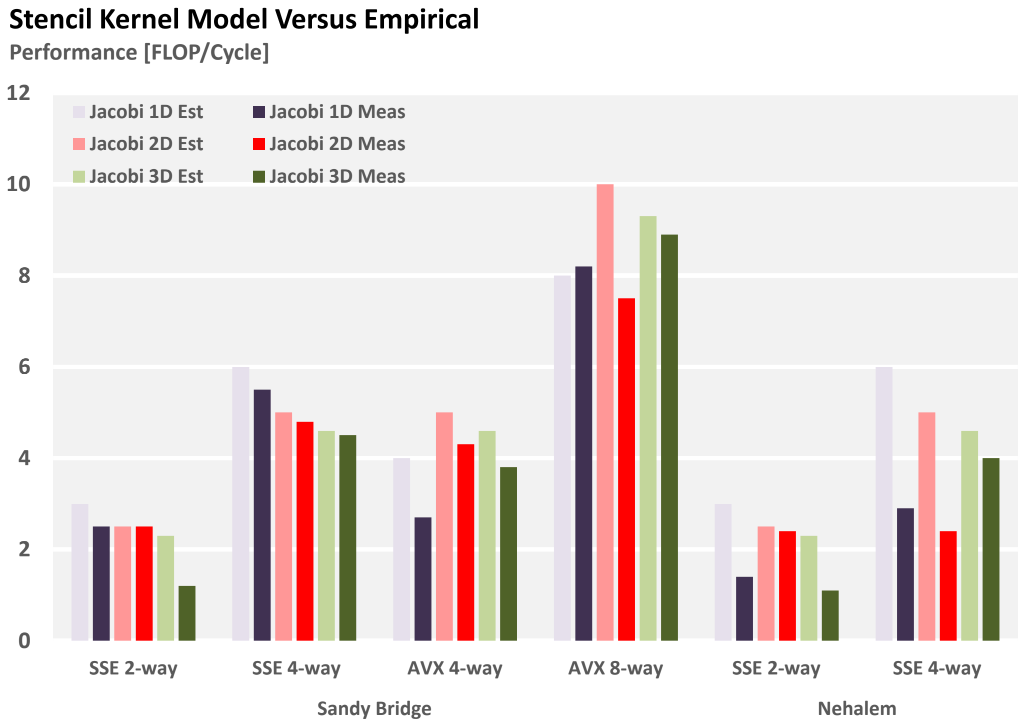


Figure 5.20: In this plot we compare our analytical performance model against empirical results of our generated kernels.

5.6 Chapter Summary

In going with the theme of this thesis, we demonstrated how to produce high performance implementations for stencil operation using the two part method. The regular structure of the stencil operation, coupled with the algorithm determined how we should lay out the data. In this class of operations the layout was performed on the SIMD registers as opposed to explicit blocks in memory, as was the case for matrix-matrix multiplication. The kernel generation allowed us to produce highly specialized stencil code that was tailored both to the problem and the computer architecture. What this shows is that for stencil computations we have a systematic approach for obtaining performance.

We focused on stencil computations because it captures a large class of problems in many domains where structure plays an important roll in the computation. Moreover, the particular stencils we focused on are representative of this fairly broad class, which is why will extend this framework to more operations. This means we must put more focus on representing this operations within our kernel generator and capturing the boundaries of their domain in a more formal approach. Additionally, the data layout transform used in this chapter was mostly focused within SIMD registers. For future work, we will look at in-memory data layout transformations, like those used in the high performance matrix-matrix multiplication operation. This would provide a major performance benefit by simplifying index computation and by preserving spacial locality of the working dataset. In the next chapter, we will show how this can be achieved for graph computations over regular, yet scale-free graphs.

Part II

Sparse and Unstructured Applications

Chapter 6

Hierarchical Data Structures Bridge the Dense-Sparse Performance Gap

6.1 Introduction

In the previous chapters, we discussed a systematic approach for producing high performance implementations of Dense Linear Algebra (DLA) and Structured Mesh operations. This entailed splitting the operation in two parts. The first is the algorithm, data access and data layout which feed the second part, an efficiently generated kernel. In the case of DLA, the first part was provided for us by the work of others. Similarly, in the Structured Mesh case we relied on a Polyhedral Compiler to provide us with that access pattern while we provided efficient generated kernels. In this chapter, we will tackle both halves of the problem, data access and kernel code, for sparse Matrix-Vector Multiplication (spMV) on synthetic scale-free data. Further, we will show the importance of a data structure that bridges these halves together.

We target scale-free graphs because many real-world networks share that topology [74, 75]. In these graphs, there are a small number of nodes with many edges and many nodes with only a few edges. The nodes with many connections are referred to as hub nodes and they connect many neighboring nodes within their cluster. If we zoom into one of these neighborhoods we would see this behavior repeat, as illustrated by Figure 6.1. In this chap-

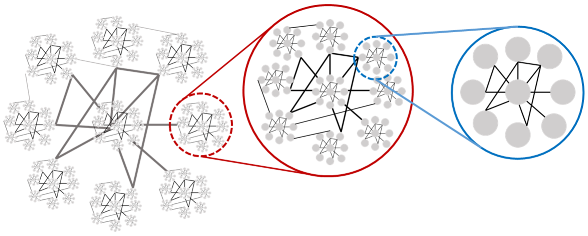


Figure 6.1: In this cartoon, we demonstrate the recursive nature of a scale-free graph. At the highest view, we see that a few key vertices (hubs) contain the majority of the edges. We can partition the graph based on these key vertices. If we descend into a sub-graph created by this partitioning, we see the same behavior.

ter, we leverage the characteristics of scale-free graphs in order to create an efficient data structure that provides fast access to clusters of nodes in a hierarchical fashion.

Our data structure uses a hierarchical sparse data format to efficiently map scale-free graphs to modern computer architectures. This format captures the structure of the graph at multiple levels of resolution. By varying the resolution at a given level, we can fit that sub-graphs to the caches of the target system. This allows us to maximize cache reuse and minimize bandwidth requirements for spMV like operations. Additionally, by using a sparse encoding at each level of the hierarchy, we can reduce the storage requirements of our format.

We recognize that the domain expert has extremely valuable knowledge on the structure of their graph and the details underlying the problem from which it came. In order to capture this structure, we rely on the domain expert to provide a mapping function between the structure of the graph and our data structure.

6.1.1 Our Contribution

This chapter can be broken into three key contributions.

- We provide a hierarchical sparse format called Recursive Matrix and Vector (RMV). This hierarchical format captures the structure of the graph using a tree of specialized containers.
- We provide an Application Programming Interface (API) that allows

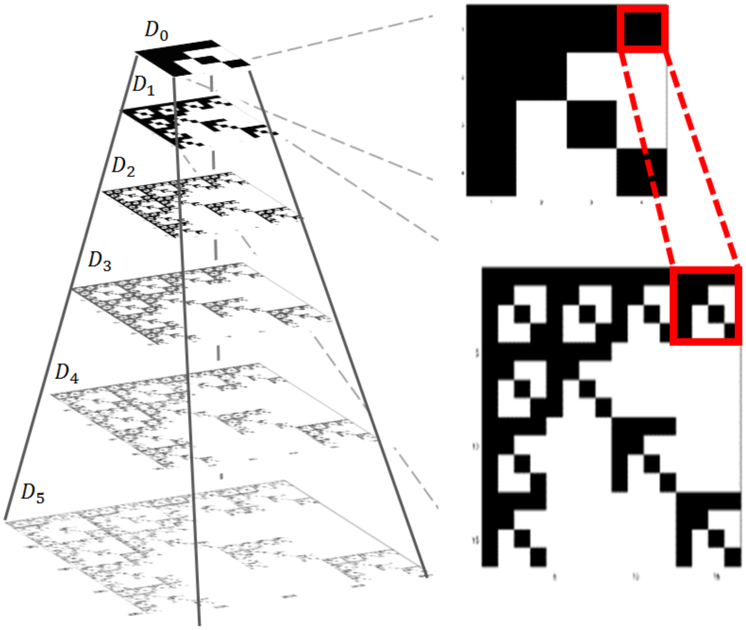


Figure 6.2: In this figure, we show how the adjacency matrix of the graph is stored hierarchically using our format. At the depth d_5, we have a view of what the original incidence matrix. We store the graph hierarchically, so as we move up this pyramid, the graph is coarsened. Each element in the levels above d_5 is a pointer to the elements in the level below it.

the domain expert to construct these RMV objects from domain knowledge. The expert can pass the structural information of the graph to our data structure.

- Lastly, we perform a performance analysis of our format using synthetic scale-free graphs.

This chapter will demonstrate that the mechanical process for obtaining performance is also applicable to sparse data, if we understand the underlying structure of the data. In the next chapter, we extend this work from synthetic scale-free data to real world scale-free data.

6.2 Proposed Mechanism

In this section, we describe our hierarchical sparse data format and its constructor function. If the user provides our framework with domain knowledge about the structure of the graph, then our framework can construct both a sparse matrix and dense vector in our format that preserves this structure in memory.

Data Format. The key component to our storage scheme is the hierachical Recursive Matrix or Vector (RMV) element. The graph is stored recursively by blocks inside this container. The graph is represented at multiple granularities (Figure 6.2). At the very top this block contains the entire graph below it and at the very bottom the blocks contain pointers to the weights of the edges or values of the vertices.

```
struct recursive_dense_vector{
 type; size;
 values; };

struct recursive_sparse_matrix{
 type; size;
 bitmask_matrix;
 values; };
```

In each Recursive Matrix or Vector container, there is a field that determines if its elements point to containers at a finer level of the graph, or if they point to the actual values. Next, it contains the number of rows and columns of blocks that it can access. Additionally, there is a bit matrix that describes the pattern of the non-zero elements. Lastly, there is an array that either containers pointers to the next level of blocks or values if we are the final level. In Figure 6.3 we show an instantiation of this object. We then show in Figure 6.4 a fully constructed Recursive Sparse Matrix of height 2. The top level provides a coarse view of the matrix, and each of its elements point to a Recursive Sparse Matrix on the bottom level.

Construction. The user provides the structural information of the graph to our framework. In order to enable the assembly of these RMV objects based on the user's knowledge, we provide a tree based descriptor. This tree

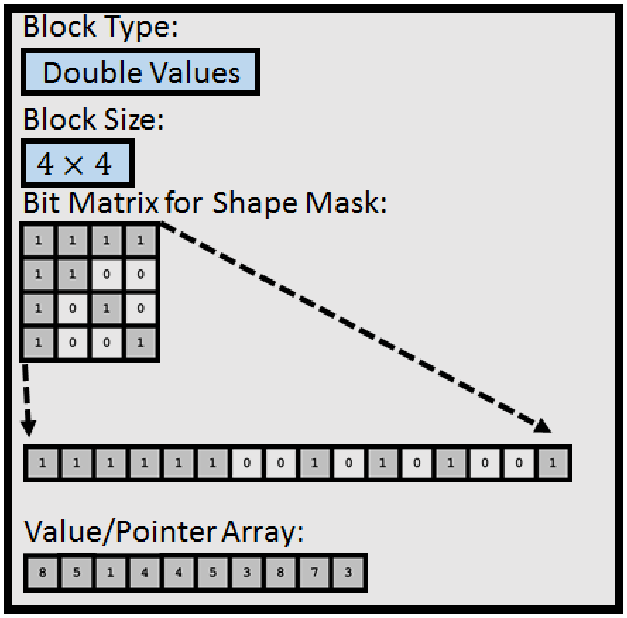


Figure 6.3: This figure represents an instantiated Recursive Sparse Matrix. It contains the shape of a sparse graph as a bit matrix and the corresponding values stored as a dense list. Note that the bit matrix can be treated as an integer, which can be used to select a kernel specialized to that specific shape.

captures the recursive partitioning of the vertices in the graph as a nesting of partitionings. The nodes of this tree are shape descriptors and they are generated by a user provided get_child function.

```
struct shape_desc{
 depth; nnz; rows; cols;
 void *user_data
 void *get_child; };
```

Each of these descriptors represents the shape of the graph at its assigned depth. The first four parameters give the coarse structure of the sub-graph being viewed. The user data field allows the user to pass bookkeeping information to the get_child function during the formation of this tree.

```
desc_child = get_child(i,j, desc_parent )
```

Using these descriptors and functions, we can assemble a RMV object

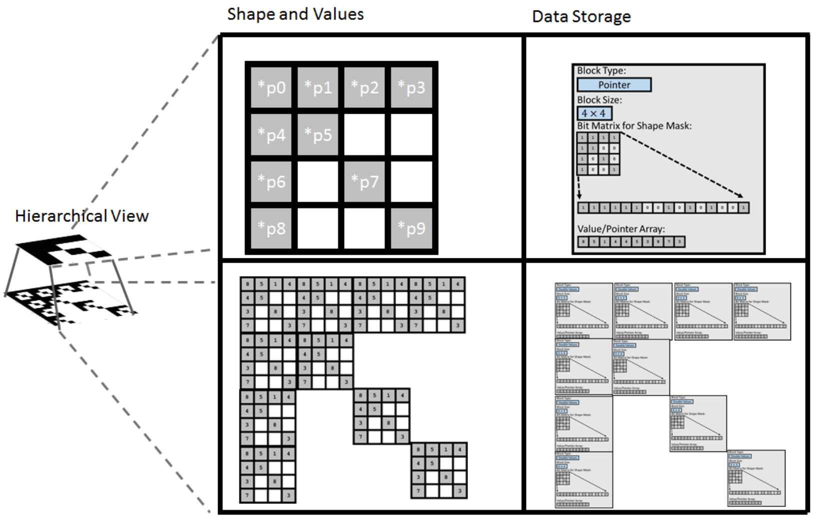


Figure 6.4: We have a two level Recursive Sparse Matrix object. The top level captures the shape of the bottom layer. Additionally, each of its elements points to the Recursive Sparse Matrix objects on the bottom level.

according to a user defined partitioning. We have used this tree based approach for assembling the Recursive Matrix object from synthetic data, as well as, from COO formatted sparse matrix data.

```
assemble_recursive_matrix(
      recursive_sparse_matrix *head,
      shape_desc              *parent )
{
 head->values = malloc( parent-> nnz )

 for i,j in parent->rows,cols
  if child = parent->get_child(i,j,parent)
     != NULL
   mark_bit_mask(i,j, head )
   assemble_recursive_matrix
       ( head->values[p++], child )
}
```

In the pseudo code snippet listed above, we capture the essence of how a Recursive Matrix is constructed recursively from the shape tree. In the

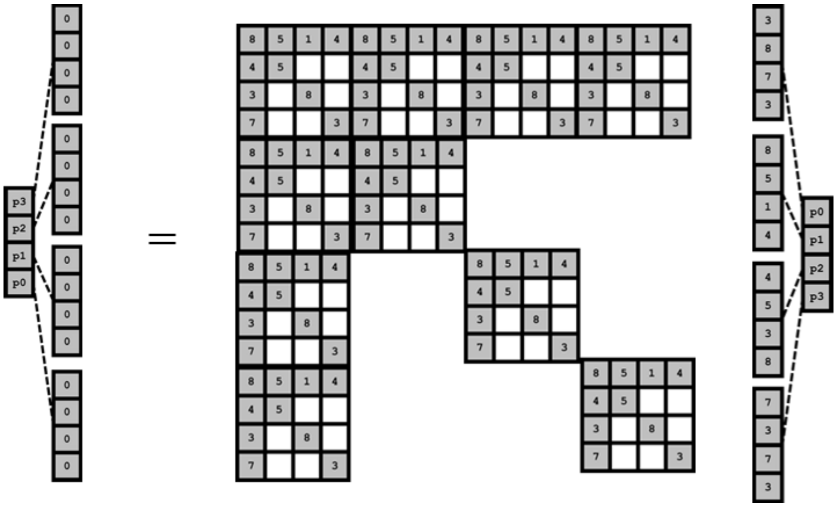


Figure 6.5: Here we illustrate a blocked Sparse Matrix-Vector Multiply (spMV). The output and input vector are stored as Recursive Dense Vectors with a depth of 2. The top level of the vectors are 1×4 vectors of pointers, that point to 1×4 sub-vectors of scalar elements. The matrix is stored as a Recursive Sparse Matrix, also with a depth of 2. The top level is a 4×4 sparse matrix, where each non-zero points to a 4×4 sparse matrix of scalars.

snippet we show that at each descent into the recursive matrix we simultaneously descend into the user provided shape tree that describes the graph. The construction of the Recursive Vector follows a similar approach based on these tree shape descriptors.

Computation. Computing a Sparse Matrix-Vector Multiply (spMV) using the RMV object is equivalent to performing a blocked Matrix-Vector Multiply. Functionally, it entails a recursively blocked spMV, that descends down the matrix and vector objects until it reaches the actual data values. This is illustrated in Figure 6.5. Note that the sub-vectors are reused across many elements in the matrix, so they are kept in the cache. The amount of reuse is dependent on how the user partitioned the vertices of the graph.

An additional feature of our structures is the ability to dispatch to specialized functions. In each container for the sparse matrices, there is a bit matrix which determines the shape at that level. This matrix can be treated as a single integral type and used to dispatch the children to a specialized version of that function. The specialized function can be optimized to include the indirect indexing directly in the code.

```
spmv_dispatch( y, mat, x){
  switch(mat->bit_matrix)
  ..
  case 65:
    spmv_65(y,mat->values, x)
}
```

The corresponding spmv_65 function can be fully unrolled and optimized to avoid indirect address computation. As long as the size of the instruction cache permits, we can specialize to all potential matrix shapes for a given block size.

```
spmv_65(y,vals,x){
 y_0 += vals[0] * x[0];
 y_0 += vals[1] * x[1];
 y_0 += vals[2] * x[2];
 y_0 += vals[3] * x[3];
 y_1 += vals[4] * x[0];
 y_1 += vals[5] * x[1];
 y_2 += vals[6] * x[0];
 y_2 += vals[7] * x[2];
 y_3 += vals[8] * x[0];
 y_3 += vals[9] * x[3];

 y[0] += y_0;
 y[1] += y_1;
 y[2] += y_2;
 y[3] += y_3;
 }
```

In the previous example, we demonstrated how to aggressively optimize the specialized spMV kernel for a specific shape using scalar instructions. We could extend this approach to SIMD instructions.

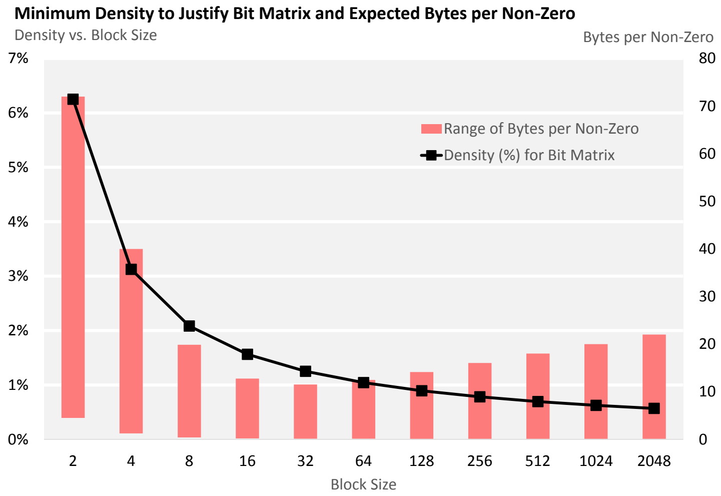


Figure 6.6: In this analysis, we determine the minimum block density needed to justify the use of a bit matrix over a COO style list for a range of block sizes. If the user can partition their graph into blocks that maintain these densities, then the graph will reap the benefits of our storage mechanism. On the secondary axis we show the possible range of overhead of the RMV structure, measured in Bytes per Non-Zero, for each given block size. The denser the block, the lower the overhead.

Data Structure Analysis.

$$\begin{aligned} s_{\text{rmv}}(n) \quad &= \quad s_{\text{type}}(n) + s_{\text{size}}(n) + s_{\text{nnz}}(n) + s_{\text{mask}}(n) + \\ &\quad\quad s_{\text{ptr\_mask}} + s_{\text{ptr\_vals}} \end{aligned} \tag{6.1}$$

Where s_{type} is the block type, s_{size} are the dimensions of the block, s_{nnz} is the number of non zeros in this block, s_{mask} is the actual bit mask matrix, and $s_{\text{ptr\_mask}}$ and $s_{\text{ptr\_vals}}$ are the bit mask and value pointers, respectively.

These blocks only store a small local sub-graph, so the size of the data types used can be minimized to the number of bits needed to encode a block of that size. We can represent these sizes s as functions of n, where n represents the size of an $n \times n$ block that we would like to store in our RMV Structure.

We can replace the size s terms with the following functions:

$$\begin{array}{lcl} s_{\text{type}}(n) & = & 8b \\ s_{\text{size}}(n) & = & 2log_2(n) \\ s_{\text{nnz}}(n) & = & 2log_2(n) \\ s_{\text{mask}}(n) & = & n^2 \\ s_{\text{ptr\_mask}}(n) & = & 64b \\ s_{\text{ptr\_vals}}(n) & = & 64b \end{array} \tag{6.2}$$

This overhead formula is only for a single block. If we assemble a graph in a hierarchical fashion using blocks of size b (fixed size blocks are not a requirement of our structure), then our overhead becomes:

$$s(n, b) = \frac{1 - \frac{1}{r^k}}{1 - \frac{1}{r}} s_{\text{rmv}}(b) \tag{6.3}$$

Where $k = \log_b n$, $r = db^2$ and d is the average density of the graph in each block.

If we make n arbitrarily large, then $s_{\text{mask}}(n)$ becomes the dominant factor. If we store a large graph as a single block in our storage scheme, then the overhead would be unnecessarily large compared to Coordinate Storage (COO). However, our scheme is designed to store a scale-free graph hierarchically such that the density of the leaves is greater than the entire graph. Thus, if the density of the blocks is sufficiently large, then this overhead is amortized over many elements. Otherwise, a COO list should be used for that block, a feature that our format allows. Thus, we need to balance the size of our blocks with the density of the sub-graph that they will store.

Achieving this desired density for larger block sizes may not be practical for real world scale-free graphs. Fortunately, we do not need a high density of the overall graph, only high densities in tightly clustered sub-graphs. Thus, the question becomes: given a sub-graph of size n, what minimum density is needed to overcome the overhead of our storage format, RMV, compared to COO? In Figure 6.6, we compare necessary density to break even in overhead for a given block size. This is computed using by solving for density d in $\text{mask}(n) = \text{coo}(n)\text{nnz}$, where $\text{nnz} = dn^2$. For a range of block sizes, this plot shows what density is needed to justify the use of the bit matrix over COO inside the RMV structure. In that plot, we also calculate the range of the overhead of our RMV mechanism for each block size.

6.3 Experimental Setup and Analysis

In this section, we evaluate the performance of our data format for synthetic scale-free graphs. We want to show that we can achieve reasonable spMV performance for a single threaded, scalar, and un-optimized implementation of our framework. Additionally, we show that we achieve competitive performance with the state of the art. We chose spMV as a proxy for graph operations for the following reasons: First, many graph operations can be represented in terms of iterative spMV-like operations. Second, this operation is typically expertly tuned, so it sets a high bar for performance.

Synthetic Dataset. For our datasets, we generate synthetic Discrete Kronecker Graphs [76] of various sizes. We chose these graphs because they approximate scale-free graphs. Using Kronecker graphs we can control the sparsity, number of non-zeros and graph size in a predictable fashion. The construction of the Kronecker Graphs used in our experiments is as follows:

We start with an initiator matrix B_i, which in our case is the arrowhead pattern.

$$B_1 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \tag{6.4}$$

We construct larger Kronecker Graphs B_i for $i > 1$ using the Kronecker Tensor $\otimes$ in this formula $B_i = B_1 \otimes B_{i-1}$. The term on the left-hand side describes the coarse structure of the matrix, where the term on the right-hand side describes the fine grain structure. For our arrowhead initiator B_1, we can visualize this matrix as:

$$B_{i+1} = \begin{bmatrix} B_i & B_i & B_i & B_i \\ B_i & B_i & 0 & 0 \\ B_i & 0 & B_i & 0 \\ B_i & 0 & 0 & B_i \end{bmatrix} \tag{6.5}$$

While our format does not require the graph to be a Kronecker Graph, using it allows us to quickly compute the number of non-zeroes (or edges), the number of vertices and the density. We can then relate these features to the performance of our implementation.

$$\begin{array}{lcl} \text{nnz}(B_i) & = & 10^i \\ \text{num\_verts}(B_i) & = & 4^i \\ \text{density}(B_i) & = & \frac{\text{nnz}(B_i)}{\text{num\_verts}(B_i)^2} \\ & = & (\frac{2}{5})^i \end{array} \tag{6.6}$$

We selected these graphs because they represent the ideal case for our data format, because we can base our partitioning on the mathematical representation of the Kronecker Graph. However, our data format is not limited only to Kronecker Graphs. It can store arbitrary sparse graphs, but we only expect to see performance benefits if the user can map the structure of their graph to our Recursive Matrix data structure.

Test Bench. Our target systems include: an Intel Core i5-5200U running at 2.20 GHz with a memory bandwidth of 25.6 GB/s and an Intel Xeon E5-2667 v3 running at 3.2 GHz with 68 GB/s of memory bandwidth. The theoretical peak spMVperformance on these machines are 6.4 GFLOP/s and 12.75 GFLOP/s respectively. This assumes that the vectors are resident in the cache and that for every 8B consumed, 2 FLOPs are performed.

6.3.1 Performance Analysis

In order to demonstrate the effectiveness of our data structure for matrix-vector like operations on scale-free graphs, we evaluated two variants of our framework: a scalar and Single Instruction Multiple Data (SIMD) implementation.

The goal of our scalar experiment (Figure 6.7) is to show the performance of a minimally tuned scalar spMV implementation using our RMV data structure over synthetic scale-free data. We do this to establish a baseline of what is achievable by using our data-structure and we relate this to the density of the graph and the overhead of our format. What we see in Figure 6.7 is that as the problem size grows larger, the density of the synthetic scale free graph decreases, but the fraction of overhead introduced by our structure converges to 25%. For comparison this value is 66% for COO. We also see that for our untuned scalar implementation, the RMV spMV sustains 10% and 40% peak floating point performance on the Xeon E5 and Core i5, respectively.

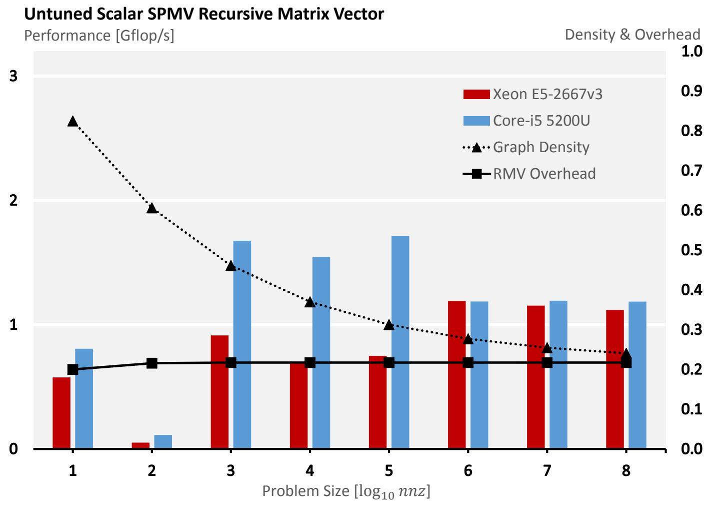


Figure 6.7: On the primary y-axis we measure the performance of an untuned scalar spMV on our RMV data structure for graph sizes ranging from 10 to 10^8 non-zero edges. For the problems larger than 10^6 the graph data exceeds the size of the caches on both systems, with the largest graph containing 800MB of graph data. The untuned scalar implementation achieves between 10% to 40% of the target systems' double precision floating point peak performance for synthetic data. On the secondary axis, we compare the graph's density relative to the overhead imposed by our data structure as the problem size grows. For comparison the overhead for COO is 66%.

In Figure 6.8 we compare an optimized version of our data-structure and spMV framework against state of the art implementations. We use SIMD short vector instructions to compute the inner-most 4×4 blocks of the spMV. These kernels specialize to the shape of sub-graph, which dispatch on the bit-matrix and only compute on the non-zero elements for that particular mask. Additionally, we chose block sizes that fit the sub-graphs into the various caches. When constructing the graph, we lay out the elements in a contiguous order that matches how they will be computed (this is done by passing a user defined `malloc` routine during the RMV construction). Lastly, we use prefetching to load the next sub-graph in its respective cache.

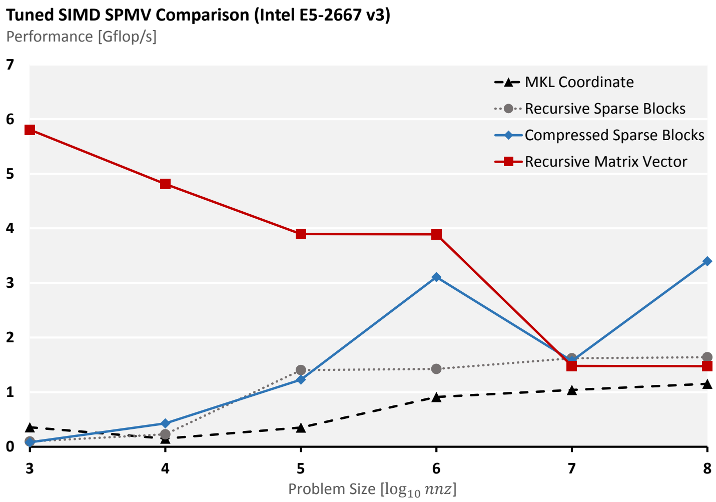


Figure 6.8: In this experiment, we compare the performance of a tuned SIMD spMV implementation on our RMV data structure against state of the art implementations on synthetic scale-free data. This data set exceeds the cache and is 800MB, excluding overhead. Our RMV implementation outperforms the other implementations until 10^7 non-zero elements. We suspect that by rearranging our data layout we can make more effective use of the large number of channels on this system.

The application of these optimizations parallel the implementation of a high performance dense linear algebra routine.

For a single thread, our implementation outperforms the state of the art for synthetic scale-free graphs up to the size of 10^7 non-zero edges. We suspect that for larger sizes our blocking dimensions are not optimal. Furthermore, we suspect that for those larger sizes our data layout does not efficiently use the memory subsystem. This could be addressed by using a layout that maximizes memory level parallelism. We leave these two adjustments for future work.

6.4 Chapter Summary

In this chapter, we demonstrate that our mechanical approach to performance is applicable to matrix operations over sparse data, where the structure of the data is known. Just as in the Matrix-Matrix Multiply and Structured Mesh case, we split the problem in a data access and kernel generation portion. The key to extending this approach was the use of a data structure which captured the structure of the graph and bridged the data access to the kernel. In the next chapter, we will extend this work to graph operations on real-world scale-free networks.

Chapter 7

Real-World Networks Provide Sufficient Structure for Performance

7.1 Introduction

In this chapter, we extend our work in the previous chapter on synthetic scale-free networks to Graph Operations over real-world scale-free networks. In keeping with the overall theme of this thesis, we identify hierarchical structures in real-world graphs that allow us to map the graph data to the memory hierarchy. Once the graph data is laid out efficiently in memory we can perform efficient iterative spMV-like graph operations.

The contribution of this chapter:

- We identify that the hierarchical cluster structure seen in real-world scale free graphs can be used to partition the graph hierarchically in memory.
- We demonstrate how we can develop a high performance libraries for graph algorithms implemented as matrix computations.
- We show how to utilize time-tiling for high performance graph operations.

This chapter is organized as follows: First, we examine the properties of real-world scale-free networks. Next, we discuss a matrix formulation of graph operations. From that we develop a time-tile approach for these graph operations. This mechanism allows us to leverage the structure of the graph to speed up the time needed to converge to a solution, and it allows us to make efficient use of the cache hierarchy. After we develop the theory behind our approach, we describe the implementation of graph framework. This is a linear algebra based framework which utilizes time-tiling. After this, we evaluate the performance of our framework on real-world data and analyze the results. Last, we summarize our graph framework.

7.2 Theory

In this chapter, we develop a framework for efficiently performing graph analytics over real-world scale-free graphs on modern computer architectures. Our solution leverages the structure in real-world graphs to guide the selection of the algorithms used and the mapping of the graph data to our target architecture. The structure in question is the hierarchical clustering property seen in many real-world graphs, and we use this to dictate how we recursively partition our graph for our graph algorithms, and how this partitioned graph is laid out in memory to maximize effective cache reuse in the hierarchical memory subsystem.

7.2.1 Real-World Graphs

In this work, we restrict ourselves to real-world graphs where the vast majority of vertices have only a few edges, and a very small number of vertices contain many edges. Examples of these kinds of graphs include:

- web networks: web pages are the vertices and the edges are the hyperlinks between those pages [77, 75, 78].
- email traffic: we can view as the email addresses representing vertices and the destination of the email represents the edge between two vertices. Alternatively, the emails can be viewed as the vertices with edges between two emails representing similar correspondence [79, 80].
- research paper relations: papers are the vertices and the edges can represent citations between articles or co-authorship [81, 82, 83, 84, 85]
- routing networks: vertices are routes and edges are either physical or virtual connections between routers [86, 87].
- social networks: individuals form the vertices and their relationships are captured as edges [88, 89, 90].

- biological networks: for example in protein-protein interaction networks, each protein represents a vertex and the edge connect two proteins represents that two proteins are involved in the same process. [91, 92]
- financial data: accounts are represented as vertices and transfers between accounts form the edges [93, 94, 95].
- transportation networks: vertices can represent intersections and the roads connecting intersections are captured by edges [96, 97].

In the follow sections, we will review generic, yet important properties observed in real-world graphs.

7.2.2 Scale-Free Networks

A scale-free network [50, 98] is a graph whose degree distribution, or the probability of a vertex having a certain number of edges, follows a power-law distribution:

$$P(k) = k^{-\gamma} \tag{7.1}$$

This equations captures the probability that a vertex has k edges is approximately $k^{-\gamma}$, where γ is a constant specific to the graph. This creates a long tail distribution where the vast majority of vertices in real-world graphs have very few edges and a small number of key vertices have a very large number of edges. We call these rare, highly connected vertices hubs or authorities [77]. In the following sections, we will show why these hubs are critical for propagating information throughout the graph. For now it is worth noting that these hubs form distinct clusters within the graph. In Figure 7.1 we show a cartoon of a small scale-free graph with the distinct communities marked using colored backgrounds. The hubs in this illustration are the nodes with the most edges.

Models for Real-world. Another key property of real networks is the small-world behavior [99]. This property states that neighboring vertices are more likely to share each other neighbors than to be connected to other vertices in the graph. This behavior explains the tight knit communities that arise in real-world networks. Additionally, this property gives rise to hub nodes, or vertices with many edges. Lastly, even those these networks

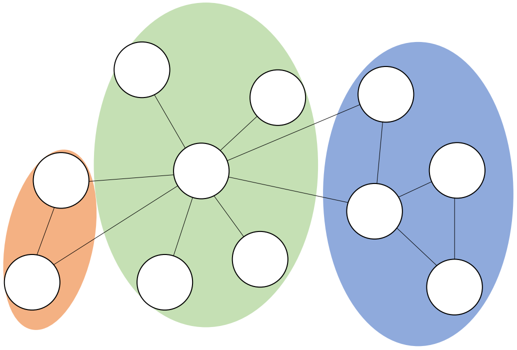


Figure 7.1: In this illustration we show a cartoon of a scale-free graph. We highlighted the clusters in this graph.

are sparse they have relative small diameters, thus the distance between any two vertices is small.

7.2.3 Hierarchical Clustering in Real-World Graphs

A key property that emerges in real-world networks is the hierarchical organization of clusters [100], [101], and [102]. Vertices in these graphs hierarchically organize to form a recursive cluster structure. We can visualize this with the hub and spoke model. The vast majority of vertices have only a few edges (spokes) that connect to vertices with many edges (hubs). If we were to collapse the hubs and their spokes into a single *super-node*, we would see the exact same hub and spoke pattern over these super-nodes. Thus, the pattern repeats until there is a single super-node. We capture this hierarchical structure in Figure 7.1.

As a consequence of this hierarchical topology, hubs are critical for information flow into their communities. This notion is reinforced by observations in real-world networks. For example, in [103] the authors observed that in infection models over scale-free graph, highly connects hubs contracted and spread the infection very quickly. Thus, providing preferential treatment to highly connected hubs decreased the overall infection rate. Similarly, the authors in [104] observed that for viruses spreading on computer networks, that topology, not the spreading rate of the virus, determines the overall rate in which machines are infected. In [90] the authors provided another perspective of information flow through scale-free topologies. They observed that information originating from a community is more important within the community than to vertices outside of the community. This is explained by the fact that vertices within a community have high connectivity to each other, but very little connectivity outside.

Communities in real-world scale-free networks are hierarchically clustered. Information flowing throughout the graph travels through highly connected hubs, but most information within a community is mostly propagated locally. We use these two key properties in the design and implementation of graph analytic framework. First, all graph data is stored in a manner that preserves this hierarchical structure and matches it to the cache hierarchy. Second, the algorithms we select in implementing our graph operations focus their computation locally, within the communities of the graph, before computing the operation globally on the graph.

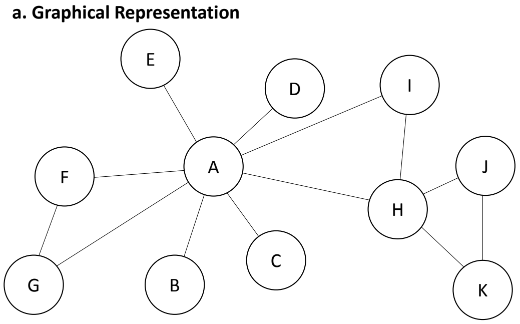
In the next sections, we describe the graph operations we target, the algorithms selected in their implementation, and the optimizations we perform. Whenever possible, we take advantage of the hierarchical to extract performance at each step of the way.

7.2.4 Operations Over Graphs

For our experiments in this chapter, we target Sparse Matrix Vector Product (spMV), the Single Source Shortest Path (SSSP) and PageRank operation. The spMV operation computes $y = Ax$ where y and x are dense vectors and A is a sparse matrix. This operation serves as a proxy for more complex operations and it is a well studied operations with highly tuned implementations on modern hardware. We will demonstrate how many graph algorithms can be built on spMV and why it is important that this operation is efficient.

The Single Source Shortest Path (SSSP) operations solves the problem of computing the shortest path between all vertices to a selected source. More precisely: given a graph $G = (V, E)$ and a source vertex $s \in V$ we compute the distance d_v from vertex s to vertex $v \in V$. We illustrate this process in Figure 7.3.

We selected this operation because it forms the basis of other operations such as betweeness centrality and All Pairs Shortest Path (APSP). Additionally, this operation serves as a proxy for other graph operations such as Traversal, Minimum Spanning Tree (MST), and Inference because with slight modification of the algorithm we use (GraphBLAS) [5] we can implement them.



b. Matrix Representation

$$\left(\begin{array}{cccccccccccc}
 & A & B & C & D & E & F & G & H & I & J & K \\
A & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & \\
B & 1 & 1 & & & & & & & & & \\
C & 1 & & 1 & & & & & & & & \\
D & 1 & & & 1 & & & & & & & \\
E & 1 & & & & 1 & & & & & & \\
F & 1 & & & & & 1 & 1 & & & & \\
G & 1 & & & & & 1 & 1 & & & & \\
H & 1 & & & & & & & 1 & 1 & 1 & 1 \\
I & 1 & & & & & & & 1 & 1 & & \\
J & & & & & & & & 1 & & 1 & 1 \\
K & & & & & & & & 1 & & 1 & 1
\end{array}\right)$$

Figure 7.2: We can represent our graph on the left as an adjacency matrix A where each entry a_{ij} represents an edge from vertex j to vertex i. Depending on how we label our graph we can preserve the locality of the graph in the matrix.

We also target the PageRank [105] operation, which determines the importance of website based on the probability that a random web surfer will reach that page. In this Graph BLAS style, we implement this as an iterative sparse Matrix-Vector product where for each vertex we compute the weighted sum of all incoming edges. Between every iteration the weights of each vertex are dampened and a fudge factor is added. This continues until the difference between two iterations is less than a given threshold.

7.2.5 Graph Operations as Matrix Operations

In this chapter we implement graph operations using the approach in the GraphBLAS/Combinatorial BLAS [5]. In this approach, graph operations are represented as transformations of a module defined over a particular semiring that is selected for the operation. This is done in two steps: First, the input graph $G = (V, E)$ is represented as an Adjacency Matrix A (Figure 7.2) where each entry α_{ij} represents the weight of an edge from vertex j to vertex i. If the entry is 0 then there is no edge. For scale-free networks this leads to a hypersparse [106] matrix. Second, the graph operation is implemented as either a Matrix-Vector, iterative Matrix-Vector, or Matrix Multiplication over a semiring S, selected for the operations. The semiring is an algebraic structure with an addition operator $+$, multiplication operator $\cdot$, an additive identity 0 and multiplicative identity 1. Additionally, the additive identity 0 is the multiplicative annilator $0 \cdot a = 0$.

For example, we want a transformation $B = A^k$ where each entry β_{ij} the computes the shortest path, of k or fewer hops, between vertex j and i. Then we define the underlying semiring such that $+$ is the min operator, $\cdot$ is normal addition and 0 and 1 are ∞ and 0 respectively. If we want to explicitly compute the shortest path of k or fewer hops from vertex j to i then using the standard basis vector e_x, we perform:

$$d_{ij} = e_i^T A^k e_j \tag{7.2}$$

If we want the distances between vertex s and all other vertices then we can compute

$$y = A^k e_s \tag{7.3}$$

Where $\gamma_i \in y$ is the shortest distance d_{si} between s and i using k or fewer hops. This operation is equivalent to the Single Source Shortest Path (SSSP) problem. In order to implement this operation we can either explicitly compute A^k where $k = |V| - 1$ (longest possible path) or iteratively compute the Matrix Vector Product, $y = (A \ldots (A(Ax)))$. The former approach is computationally prohibitive as it may require $O(|V|^{3(k-1)})$ operations whereas the later requires $O(|V|^3)$ operations.

To compute SSSP using the GraphBLAS approach, we perform an Iterative Matrix-Vector product $y(k) = A^k x$, where $x_s = 0$ and $x_{i \neq s} = \infty$. Additionally, k is largest number of hops between needed for all shortest paths between s and vertex i. The Iterative Matrix-Vector product is computed until the convergence condition, $y^{(k+1)} = y^k$ is met, because once the shortest path is found then no additional hops will make the path shorter. Stated differently, if the most number of hops in any shortest path is k then $A^k = A^h$ where $h > k$.

It is important to note that this approach to graph analytics hinges on the performance of the matrix-vector operation. However, for large graphs spMV is bandwidth bound. Therefore, without modification this approach is limited by the available bandwidth between the processor and memory.

7.2.6 Performance Enhancements for Graph Operations

We identify a major issues in this approach with regards to performance. We know from real-world scale-free networks that a significant proportion of information flows within clusters [104, 90]. Another way to look at this is

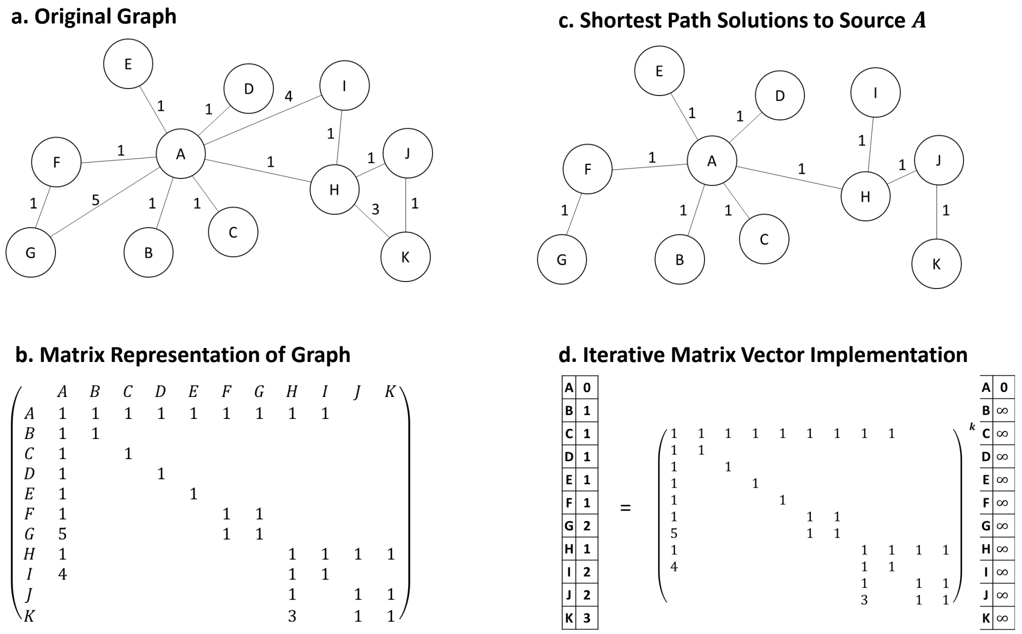
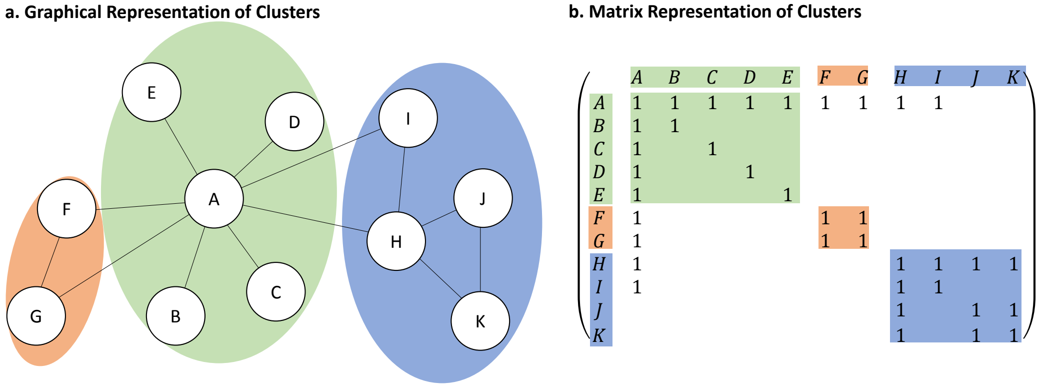


Figure 7.3: **a.** We show a graph $G = (V, E)$ with weights. **b.** Graph G is represented as an adjacency matrix A. **c.** We show the solution of the shortest paths problem (SSSP) to source vertex a **d.** We show the iterative matrix vector computation of SSSP, $y = A^k x$ where we use a semiring that replaces the addition and multiplication with min and $+$. The vector x is the starting values and y is the solution.



a. Graphical Representation of Clusters

b. Matrix Representation of Clusters

$$\left(\begin{array}{c|ccccc|cc|cccc}
 & A & B & C & D & E & F & G & H & I & J & K \\
A & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & \\
B & 1 & 1 & & & & & & & & & \\
C & 1 & & 1 & & & & & & & & \\
D & 1 & & & 1 & & & & & & & \\
E & 1 & & & & 1 & & & & & & \\
F & 1 & & & & & 1 & 1 & & & & \\
G & 1 & & & & & 1 & 1 & & & & \\
H & 1 & & & & & & & 1 & 1 & 1 & 1 \\
I & 1 & & & & & & & 1 & 1 & & \\
J & & & & & & & & 1 & & 1 & 1 \\
K & & & & & & & & 1 & & 1 & 1
\end{array}\right)$$

Figure 7.4: Depending on how the vertices are labeled we can preserve the locality of the graph in the adjacency matrix. This is ideal because a large amount of communication occurs within the community. Thus, preserving this locality in the graph insures that this communication occurs within blocks of cache.

the density of edges on the diagonal of a Kronecker graph approximation of a real-world graph is greater than the off-diagonals [76].

Therefore, it would be beneficial to focus computation on these clusters. Thus, multiple passes in these clusters are necessary and performing them in succession would benefit from temporal locality. However, in the iterative matrix-vector approach only a single pass is made to each edge between iterations.

7.2.7 Temporal Blocking for Graphs

In scale-free networks, the bulk of the communication occurs within the clusters, but the iterative spMV approach does not make use of this feature, and instead computes over all edges before repeating an edge computation. Ideally, we want to focus as much useful computation within a cluster before moving on. This would both reduce the amount of work until convergence and improve locality.

Thus, we propose an adjustment to the iterative matrix-vector product approach to graph analytics by adopting a similar technique used for the stencil methods, time-tiling [31]. Rather than iteratively computing the entire Matrix-Vector product on the graph, we perform multiple smaller local passes on subgraphs. Because these subgraphs are clusters, where information propagates the fastest, a local solution can be determined within cluster before distributing it outside of the cluster.

We can formulate our time-tiled approach as the following:

$$y = \begin{pmatrix} (A_{0,0})^k & A_{0,1} \\ A_{1,0} & (A_{1,1})^k \end{pmatrix}^n x \tag{7.4}$$

Where A_{ij} are sub-blocks of the shortest distance matrix A.

It is worth noting that if we look at the shortest path transformation matrix A^k from earlier, and $k < |V|$ is the maximum number of hops for the shortest path, then $A^h = A^k$ for any $h > k$. Additionally, we can say $e_i^T \alpha_{ij} e_j A^k = A^k$. An informal proof of this is that A^k already represents the shortest paths between any two vertices, therefore adding an additional vertex to any of those paths ($e_i^T \alpha_{ij} e_j$) will not make them shorter. A consequence of this result is that whatever optimization perform on the SSSP GraphBLAS operation, we will have a correct result as long as we eventually compute A^k. Therefore, for all $n > k$, the following true:

$$A^k = \begin{pmatrix} (A_{0,0})^k & A_{0,1} \\ A_{1,0} & (A_{1,1})^k \end{pmatrix}^n \tag{7.5}$$

While this may appear like additional work compared to the iterative matrix-vector product $y = A^k x$, it is worth noting that we iterate until convergence. Therefore, work stops once all shortest paths back to the source are found.

If tiled properly, computing local solutions from these time-tiled iterations can occur entirely within the cache by making effective use of temporal locality in the cache. While the local solutions may not be part of the global solution, it may be close because the solution of the spokes are more likely to be affected by the hubs. Ultimately, the identical solution will be reached, whether we do multiple global passes or time-tiled iterations, but by leveraging the structure of the graph and the behavior of scale-free networks the global solution may be reached faster.

7.2.8 Spacial Blocking for Graphs

Similar to the previous chapter we want to capture clusters hierarchically in blocks that fit the memory hierarchy. This builds on how information

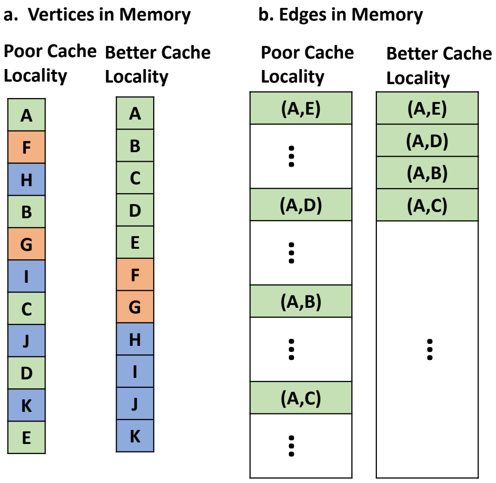


Figure 7.5: **left:** We show a non-ideal layout of vertex and edge data in memory. Elements with the same color come from the same cluster. **right:** We show an ideal layout where neighboring edges and vertices are laid out contiguously in memory.

flows within scale-free graphs [104, 90, 107, 103] and leverages the fact that nearby vertices are more likely to communicate with each other than distant neighbors. By capturing the proximity of this vertices in cache we can insure that communication between them is efficient. Thus, if two vertices are neighbors in the graph then they will be neighbors in memory and utilize spacial locality. In Figure 7.4 we capture this graph locality in the adjacency matrix, and in Figure 7.5 we show how this would look laid out in memory.

7.3 Implementation of a High-Performance Graph Library

In the previous section, we identified that real-world scale free networks have a hierarchical cluster structure that we can use to hierarchically partition a graph. We described a method for representing graph algorithms in terms of linear algebra-like operations. Lastly, we described how to use the hierarchical partitioning for space and time tiling on these linear algebra-like algorithms.

In this section, we describe how we go from this theory – hierarchical partitioning, linear algebra representation and space/time tiling – to a high performance implementation of a graph library.

We build our graph library around a generalized version of the Recursive Matrix Vector (RMV) data structure that we presented in the previous chapter. This structure allows us to capture the hierarchical recursive structure of the graph and traverse through that structure efficiently. We then develop recursive graph algorithms that operate on this structure. These recursive algorithms allow us to tile both spatially and temporally. These algorithms will ultimate divide the graph into very small sub-graphs that are computed on using extremely efficient, automatically generated kernels.

7.3.1 Generalized Hierarchical Sparse Framework

In the previous chapter, we developed a hierarchical sparse matrix framework for scale-free networks. This implementation was optimized for Kronecker Graphs [76], a class of scale-free networks. This data structure stores structured Kronecker Graphs hierarchically and efficiently in memory. While Kronecker Graphs can approximate real-world networks, more flexibility is needed for storing real-world networks. For this flexibility we generalize our RMV data structure to accommodate real-world data, while retaining the hierarchical sparse structure. We call this format Generic Recursive Matrix Vector Storage (GERMV).

Like the RMV data structure, GERMV is a tree-like hierarchical matrix storage. It stores a matrix as a hierarchical nesting of blocks. Further, each of these blocks is described by a node. This structure can be viewed as Hierarchically Tiled Array [108] or FLASH [54] if the base data type in each tile could be sparse instead of dense. The key difference between the RMV data structure of the previous chapter and the GERMV data structure is the generalization of indexing. In the original RMV structure indexing was restricted by a bit-matrix, but in the GERMV indexing can be done in COO, CSR, bit matrix, or a dense matrix format.

To accommodate generic indexing and generic data container in C we break up the GERMV container into two pieces, a base container that determines the payload and a payload that contains the indexing and data elements. In the following listing we show the base container.

```
typedef struct germv_base_ts
{
  enum  type;
  void *payload
}
```

The field *type* determines how the following field payload needs to be interpreted. Essentially, this is one method for implementing class-like structures in C.

```
typedef struct payload_dense_double
{
  double [][] data;
}

typedef struct payload_coo_uint16
{
  uint8_t row_idx;
  uint8_t col_idx;
  uint16_t   data;
}
```

In this code snippet, we demonstrate how to capture different formats as wrappers. If the base object's type is *dense double* then its payload is cast using the appropriate wrapper. This approach to classes provides us with a low overhead method of implementing an abstract hierarchical matrix type. In the next section, we demonstrate how we use GERMV object.

Abstract Matrix Types. The GERMV structure allows us to capture a graph as a hierarchical partitioning. This is done by creating a tree-like representation of the matrix where each level of the tree represents a partitioning of the parent node. To accommodate this, the GERMV allows for arbitrary data types. Typically, the leaf blocks will contain a value data type (i.e. float or int), whereas the interior nodes - or hierarchically blocks - have values with a pointer data type. These pointers, in turn, point to their child blocks. We can continue this recursively until we hit the leaf nodes. For example, in the following listing we describe a payload type that indexes using the COO format:

```
typedef struct payload_coo_germv_base
{
  uint8_t      row_idx;
  uint8_t      col_idx;
  germv_base_t data;
}
```

Given this pointer data type we can create a hierarchically partitioned graph with a single vertex and edge, one leaf node and one interior root node.

```
#define CONSTRUCT(type) ...
#define ADD_ELEM(package, i,j, data) ..

germv_base_t *interior =
  CONSTRUCT( payload_coo_germv_base );
germv_base_t *leaf     =
  CONSTRUCT( payload_coo_float );

ADD_ELEM(interior, 0,0, leaf);
ADD_ELEM(leaf,     0,0, 3.14f );
```

We create two GERMV objects, one is the root or interior node and the other is the leaf node. We use a pointer data type with COO indexing for payload type for the root node. For the leaf node we use COO indexing with a floating point indexing. Once the objects are constructed, we attach the leaf to the interior by adding it as an element and similarly we add an edge and value to the leaf using the same mechanism.

7.3.2 A High Performance spMV Implementation

Now that we have a hierarchical data structure we can build recursive algorithms for the operations we are interested in. We start with Sparse Matrix Vector Product (spMV) because it is an important operation on its own and because it will form the basis of the graph operations that we are ultimately interested in.

The spMV computes $y = Ax$ where $y \in \mathbb{R}^m$, $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$. This operations requires mn multiplications and additions and at the very least $2m + n + mn$ memory accesses. The latter figure assumes perfect reuse of

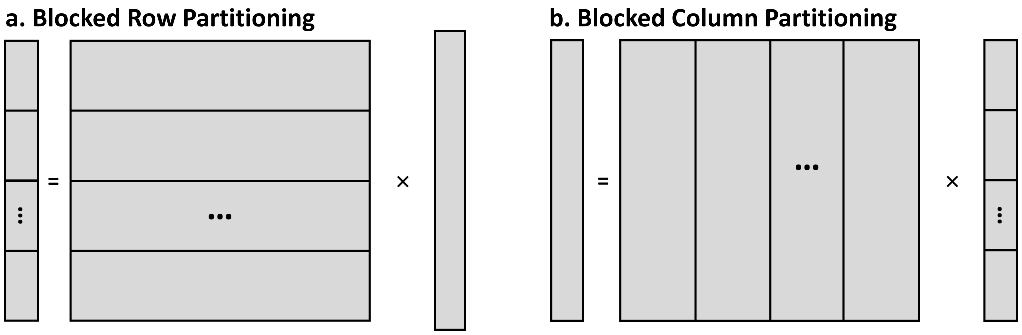


Figure 7.6: In our framework we implement the spMV operation by the recursive application of two blocked spMV algorithms. Left: we have a partitioning of rows where the output vector and the matrix are divided into blocks of rows and each output vector block is computed with the corresponding block from the matrix and the entire input vector. Right: we divide the matrix into column blocks and input into row blocks and we compute the output vector by accumulating the results of each matrix and input product.

the y and x vector. The cost of memory access is typically more expensive than computation, therefore we want to maximize reuse. In order to do this, we construct a recursive spMV over our GERMV and we match the block sizes to fit the cache hierarchy. In Figure 7.6, we show two recursive spMV algorithms, one that partitions the matrix by rows and the other that partitions by columns. When computing the operation, We divide the GERMV matrix into blocks then recursively apply these spMV algorithms until we reach the base elements of the matrix.

In Figure 7.7 and Figure 7.8 we show a matrix and graphical representation of the spMV blocked algorithms we use. We realize this process in the following code snippet:

```
spmv_dispatch(A,x,y)
{
  if( is_leaf(A) )
    spmv_kernel(A,x,y)
  else if( is_node(A) )
    spmv_block_row_and_col(A,x,y)
}
```

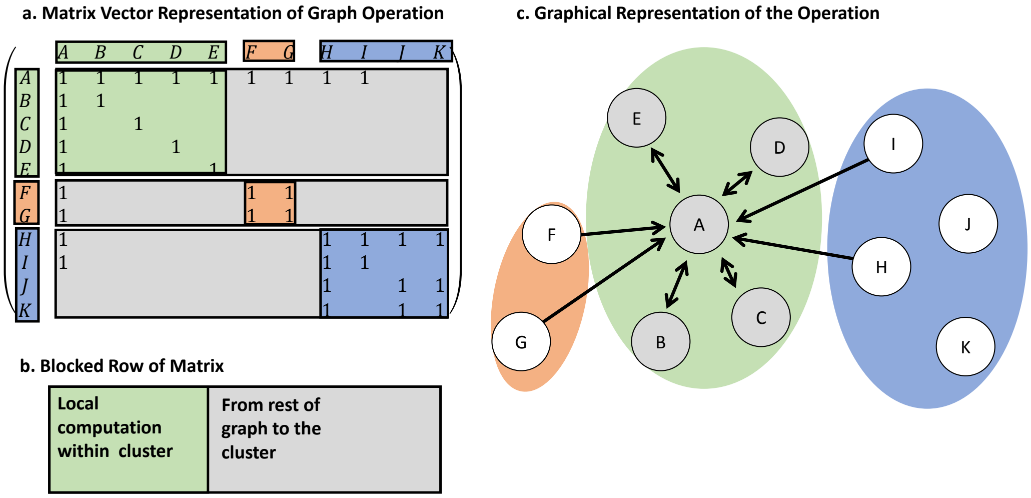
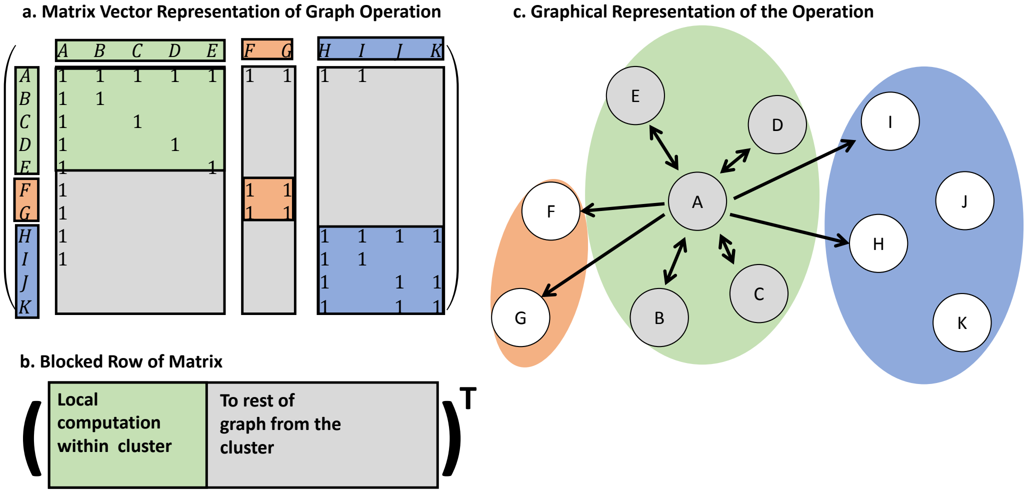

a. Matrix Vector Representation of Graph Operation
A B C D E F G H I J K
A 1 1 1 1 1 1 1 1 1
B 1 1
C 1 1
D 1 1
E 1 1
F 1 1 1
G 1 1 1
H 1 1 1 1 1
I 1 1 1
J 1 1 1 1
K 1 1 1 1
b. Blocked Row of Matrix
Local computation within cluster
From rest of graph to the cluster
c. Graphical Representation of the Operation
E
D
I
F
A
J
H
G
B
C
K

Figure 7.7: **Part a:** We show the adjacency matrix partitioned by rows. The colors correspond to the cluster where the input vector and output vector are accessing the same vertices on the graph. **Part b:** If we look at the first blocked row of the matrix we can divide it into two pieces, a block that contains the edges within a cluster, and the edges that gather data from the rest of the graph to the cluster. **Part c:** We can visualize this as a graph where information flows bidirectionally within the cluster and the remaining edges flow into the cluster.



a. Matrix Vector Representation of Graph Operation
b. Blocked Row of Matrix
Local computation within cluster
To rest of graph from the cluster
T
c. Graphical Representation of the Operation
E
D
I
A
F
J
H
G
B
C
K

$$\left(\begin{array}{c|ccccc|cc|cccc}
 & A & B & C & D & E & F & G & H & I & J & K \\
\hline
A & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & \\
B & 1 & 1 & & & & & & & & & \\
C & 1 & & 1 & & & & & & & & \\
D & 1 & & & 1 & & & & & & & \\
E & 1 & & & & 1 & & & & & & \\
F & 1 & & & & & 1 & 1 & & & & \\
G & 1 & & & & & 1 & 1 & & & & \\
H & 1 & & & & & & & 1 & 1 & 1 & 1 \\
I & 1 & & & & & & & 1 & 1 & & \\
J & & & & & & & & 1 & & 1 & 1 \\
K & & & & & & & & 1 & & 1 & 1
\end{array}\right)$$

Figure 7.8: **Part a:** In this figure we partition the adjacency matrix by columns. The colored diagonal blocks represents the clusters in the graph. **Part b:** We select a blocked column from the graph and we divide it into two pieces, the cluster where all computation occurs within the cluster and the rest of the block which scatters data from the cluster to the rest of the graph. **Part c:** We capture this behavior in the graph where the edge direction represents the flow of information.

The algorithm first determines if we are dealing with a leaf GERMV container (an object with real values) or a node GERMV container (an object with pointer values) and dispatches to the appropriate code. If the GERMV is a node, then spMV is computed recursively on the non-NULL pointer values by calling the *dispatch* function on the value. If the GERMV is a leaf, then it is computed.

```
spmv_block_row_and_col(A,x,y)
{
  for( i = 0 .. mb-1 )
    for( j = 0 .. nb-1 )
      Ab = get_elem( A, i,j );
      yb = get_elem( y, i );
      xb = get_elem( x, j );
      spmv_dispatch( Ab,xb,yb )
}
```

In this listing we show a blocked spMV algorithm that partitions both the rows and columns. The values m_b and n_b correspond to the number of blocked rows and columns, respectively. In the next listing, we show the base case for our spMV.

```
spmv_kernel(A,x,y)
{
  for( i = 0 .. mb-1 )
    for( j = 0 .. nb-1 )
      y[i] += A[i][j] * x[j]
}
```

To summarize our spMV implementation, we divide the operation by recursive applications of a blocked row and column spMV algorithm. The block sizes for each recursion match the size of the cache at each level of the hierarchy. Further, the partitioning of the spMV operation and the block sizes determine how the matrix is recursively stored in our GERMV object. Assuming that the vertices in the adjacency matrix are ordered in locality preserving manner, then our approach keeps graph cluster in cache and effectively uses that locality.

7.3.3 Parallelism

Our target systems are modern multi-core architectures. Thus it is important that our framework supports efficient parallel computation. In order to accommodate this we add two key features to our framework, parallel algorithms and efficient allocation of GERMV objects. First, we examine a parallel blocked spMV algorithm in the following listing:

```
spmv_block_row_and_col(A,x,y)
{
  #pragma parallel for
  for( i = 0 .. mb-1 )
    for( j = 0 .. nb-1 )
      Ab = get_elem( A, i,j );
      yb = get_elem( y, i );
      xb = get_elem( x, j );
      spmv_dispatch( Ab,xb,yb )
}
```

In this implementation, the outer loop is parallelized, which means that it performs m_b blocked row spMV operations. By dividing the work among the rows of the output vector and matrix we can avoid the need for multiple threads accumulating to the same location. Alternatively, we can view this approach as each graph cluster computing locally and gathering external information in parallel.

The second component of our parallel framework is a parallel-aware allocation of the GERMV matrix object. During the construction of the object, each row is allocated contiguously on the core that will ultimately compute it. This insures that on Non-Uniform Memory Architectures (NUMA) data will be located close to the core that will compute it. By combing this efficient allocation with a parallel blocked row algorithm we are able to efficiently compute spMV on parallel architectures.

7.3.4 Kernel Code Generation

We have discussed divide-and-conquer algorithms for computing spMV on our GERMV data structure which recurse until they reach a base case. This base case is implemented as a high performance kernel which computes on the residual cache resident sliver that the recursive algorithms provide. Thus, in order to achieve a high performance implementation we need efficient kernels. In order to make our framework as generic as possible we leverage kernel code generation to provide efficient spMV kernels for any arbitrary data type.

Our kernel code generator works as follows: Given a semi-ring with an addition and multiplication operator the kernel code generator produces a tuned spMV implementation over that semi-ring. For example if we want a standard spMV over floating point values, we would provide the semiring including a floating-point multiplication and a floating-point addition. Alternatively, if we wanted a kernel that traverses one hop through a graph then that semiring then we would use a binary *and* operator with a binary *or* operator over a binary adjacency matrix.

In this code snippet we illustrate at high level how we template the kernel.

```
#define spmv_dense_kern(_ADD,_MUL) \
  void spmv_semiring_kernel(A,x,y)    \
  {                                   \
    for( i = 0 .. mb-1 )              \
      for( j = 0 .. nb-1 )            \
        res  = _MUL(A[i][j],x[j]);    \
        y[i] = _ADD(y[i],res );       \
  }
```

In order to instantiate it we must define the addition and multiplication operator:

```
#define FLOAT_ADD(A,B) (A+B)
#define FLOAT_MUL(A,B) (A*B)

spmv_semiring_kernel(FLOAT_ADD,FLOAT_MUL)
```

Alternatively, if we want a single hop graph traversal over a binary adjacency matrix we could define the addition and multiplication as follows:

```
#define BIN_ADD(A,B) (A|B)
#define BIN_MUL(A,B) (A&B)

spmv_semiring_kernel(BIN_ADD,BIN_MUL)
```

This is the essence of our kernel generation system for spMV-like operations. By taking this semiring approach, not only do we define spMV operations of different data types, but also graph operations. We make this a high performance code generator by using the techniques that we developed for dense matrix-matrix multiplication and stencil kernels.

Performance Enhancements. The key optimizations that we perform on the generated kernels involve minimizing the number of loads, instruction scheduling and loop unrolling. Typically, these spMV kernels contain significantly more load operations than computations. Even if the processor has sufficient bandwidth to sustain these loads, typically there are not enough memory functional units to support the number of operations. To address this we combine load operations to maximize the memory functional unit utilization and minimize the number of load operations.

In this listing, we illustrate how we combine contiguous loads into a single operation. If we start with an spMV kernel for data stored in a COO style format:

```
void spmv_semiring_kernel(A,x,y)
{
  for( p = 0 .. nnz )
  // sequential
  a = A.data[p];
  i = A.row_idx[p];
  j = A.col_idx[p];

  // random access
  yd = y[i];
  xd = x[j];

  // computation
  res  = _MUL(a,xd);
  y[i] = _ADD(yd,res );
}
```

We can condense the sequential load operations into fewer loads with larger data widths. If we need a smaller data chunk then we can simply extract it with bitwise operations. If we assume that the data elements and indices are 1 Byte in width and that we can perform 8 Byte loads, then applying this transformation on the previous listing would look like:

```
#define EXTRACT(bulk,pos) (bulk >> pos*8) & 0xF

chunk_width = sizeof(uint8_t);
num_chunks  = sizeof(uint64_t)/chunk_width
for( q = 0: num_chunks: nnz )
{
  bulk_a    = A.data[q: q+num_chunks];
  bulk_ridx = A.row_idx[q: q+num_chunks];
  bulk_cidx = A.col_idx[q: q+num_chunks];
  for( p = q :q+num_chunks )
   {
   // sequential
   a = EXTRACT(bulk_a,p);
   i = EXTRACT(bulk_ridx,p);
   j = EXTRACT(bulk_cidx,p);
       .....
   }
}
```

The number of load operations is significantly reduced, which is critical for performance on processors with a low number of memory functional units. This does come at a trade off with the addition of more integer operations. Fortunately, most processors have a large number of integer units.

In addition to combining sequential load operations into bulk load operations, we employ static instruction scheduling and loop unrolling. In previous chapters, we demonstrated the importance of static instruction scheduling on modern out-of-order architecture. By performing this optimization on our generated kernels we can insure that the kernel sustains a high rate of computation while the inputs are in the cache. By including loop unrolling we minimize a significant amount of loop overhead. These combined transformations allow us to produce high-performance spMV kernels over arbitrary semirings. In the following sections, we will demonstrate how this generalization enables the implementation of high-performance graph operations.

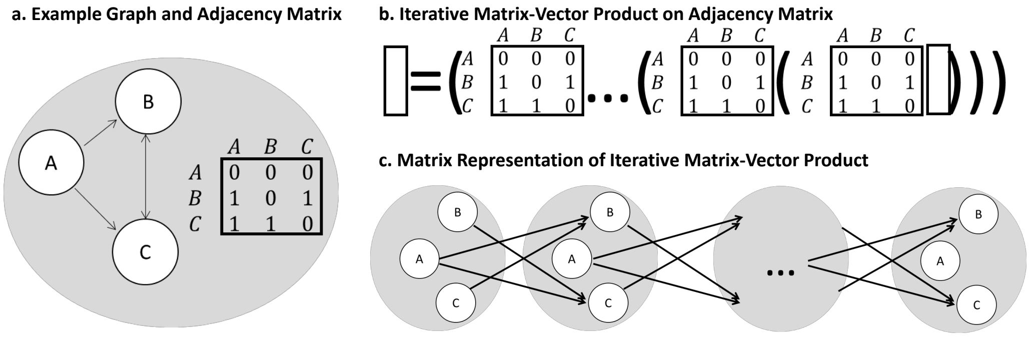

a. Example Graph and Adjacency Matrix
A
B
C
A B C
A 0 0 0
B 1 0 1
C 1 1 0
b. Iterative Matrix-Vector Product on Adjacency Matrix
A B C
A 0 0 0
B 1 0 1
C 1 1 0
c. Matrix Representation of Iterative Matrix-Vector Product
A
B
C
...

Figure 7.9: **a.** We show a small graph and its adjacency matrix. **b.** We represent an iterative sparse matrix-vector in terms of matrices and vectors. **c.** We represent the same operation in terms of data movement through a graph.

7.3.5 A High Performance Graph Library

Earlier, we described a method by which we can cast a graph algorithm in terms of an iterative matrix vector product (spMV). This achieved by replacing the underlying semiring with a graph computation. In this formulation, the performance of the graph operation is dependent on the underlying spMV. Therefore, we build our graph framework on an iterative spMV which in turn is based on the high-performance spMV described in the previous subsection. In this section, we will develop our baseline implementation of the iterative spMV graph framework. Then we will discuss our improvements over this approach using stencil-like time-tiling optimizations.

The iterative spMV successively applies a sparse matrix on the result of the previous matrix-vector operation until a stopping condition is met. In our case this halting condition is when the output of one iteration is the same as the previous. I.e. we are solving $x^{(k+1)} = Ax^k$ until $x^{(k+1)} = x^k$. In Figure 7.9 we capture this process on example graph in several different views. An important feature of this approach is that each iteration processes the entire graph before proceeding to the next iteration.

In the following pseudo code snippet, we express this iterative spMV operation. We can express this in pseudo code as follows:

```
iter_spmv(A,x,y)
{
  xk[0] = x;
  do
  {
    spmv(A,xk[k+1],xk[k])
  }while(xk[k] != xk[k+1])
  y = xk[k]
}
```

In the next section we will discuss cache improvements to this approach.

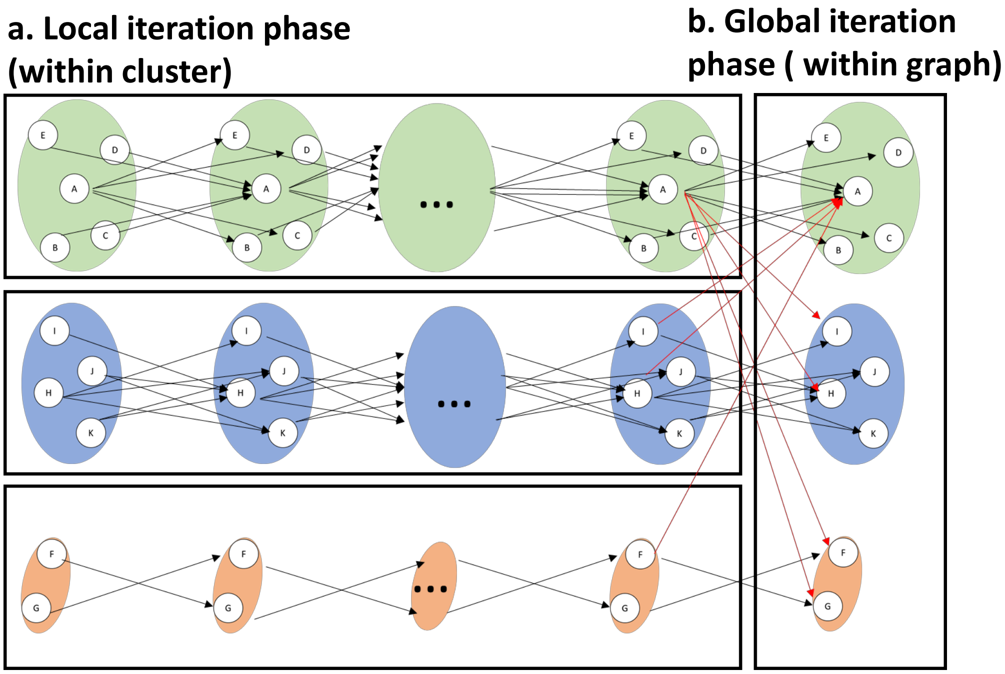
7.3.6 Time Tiling for Graph Algorithms

At each iteration the entire spMV is computed before proceeding to the next. This is noteworthy because if the vector $x^{(k)}$ exceeds the size of the cache then at each iteration everything there is no reuse.

To alleviate this, we borrow the time-tiling technique utilized in stencil computations. The general idea is to iterate over small a sub-graph until the convergence condition is reached. This continues with the next sub-graph is processed until the entire graph is processed. We illustrate this process in Figure 7.10. After all sub-graphs are computed in this fashion the process repeats for the entire graph until the stopping condition is met. In Figure 7.11 we show how time-tiling applies to a graph operation. The following listing we show how we implement time-tiling in our graph library.

```
spmv_rec_iter(A,x,y)
{
  for( i = 0 .. mb-1 )
    for( j = 0 .. nb-1 )
      Ab = A[i:i+mb][j:j+mb]
      yb = y[i:i+mb]
      xb = x[j:j+nb]
      iter_spmv_rec( Ab,xb,yb )
}
```

In the previous listing we sketched a blocked spMV-like operation that computes a time-tiled spMV on each block. This is described in the next


a. Local iteration phase (within cluster)

b. Global iteration phase (within graph)

Figure 7.10: Here we show an iterative matrix vector-like operation where the computation is computed within clusters before the computation occurs between clusters. The red lines represent global communication. This process insures that a local solution is computed before moving to a global solution.

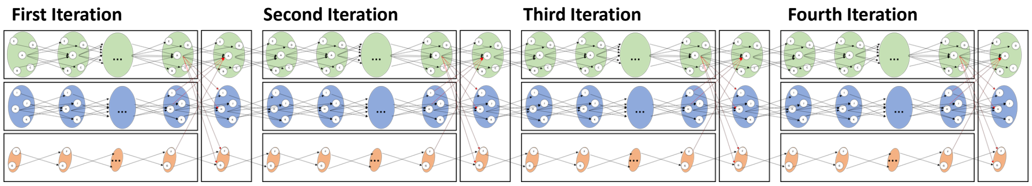


Figure 7.11: We show our time-tiled implementation in Figure 7.10 across multiple global iterations. Between each global iteration, the local iterations in each cluster are computed first. The rational is that the most communication occurs within a cluster, so focusing on the cluster first should lead to faster convergence.

listing:

```
iter_spmv_rec(A,x,y)
{
  z[0] = x;
  do
  {
    spmv_rec_iter(A,z[k+1],z[k])

  }while(z[k] != z[k+1])
  y = z[k]
}
```

In the basic iterative spMV graph algorithm implementation, our performance is limited by the performance of the spMV for the entire graph. However, we can achieve high-performance by apply time-tiling. This techniques makes effective of the cache by reusing sub-graphs before proceeding to the rest of the graph. Thus our performance depends mostly on the performance of the kernels which we have already provided an efficient solution.

7.3.7 Putting This All Together

We started this chapter with the goal of implementing a high performance graph library. By expressing our target graph operations in terms of linear algebra we are able to use the standard optimization of blocking and time-tiling. We implement these optimization using recursive spMV and iterative spMV-like operations over a data structure that is designed for scale-free graphs. This approach allows us to effectively store the working sub-graphs in cache. By taking this layered approach we can apply efficient generated computational graph kernels. This allows us to sustain a high rate of computation that is only limited by how quickly the next working sub-graph can be brought into cache. The combined approach of a data structure – that stores the graph in a data layout determined by the algorithm – and an efficient generated kernel gives us a high performance graph framework which will demonstrate in the next section.

7.4 Experimental Evaluation

The primary objective of our experiments is to demonstrate that the optimizations which require structure – such as blocking, time-tiling and data layout transformations – do enable high performance for graph operations over scale-free networks. We want show that scale-free networks have a structure that can be exploited for performance in the same way that we approach dense linear algebra and structured mesh operations. Our secondary objective for these experiments is to provide a high-performance multi-threaded implementation of an spMV and Graph Operation library.

The rest of this section is organized as follows: First, we explain our experimental setup. Next, we describe our target data set, why we selected it and how we store it in our GERMV Object. Next, we discuss our target systems and why we selected them. Last, we analyze the results of our spMV and graph experiments.

7.4.1 Experimental Setup

At the core of each of our experiments we are simply evaluating and comparing the performance of our implementation against the current state-of-the-art for a range of graphs. We run each operation on representative sub-graphs ranging in size from small cache resident sub-graphs to entire graphs that exceeds the size of the cache. This allows to characterize the performance of our implementations as a function of the problem size.

7.4.2 Target Data Set

For our experiments, we used a graph from the Stanford WebBase [109]. The graph is a web crawl of Berkeley and Stanford websites called web-BerkStan. Each vertex represents a page and an edge represents a hyperlink from one page to the other. The graph itself has $685,230$ vertices and $7,600,595$ edges and its adjacency matrix is shown in Figure 7.12. We selected this graph as a representative of real-world scale-free networks because it has been analyzed in many papers including [80].

In order to show a trend over a range of graph sizes with same overall graph behavior, we created ten graphs from the original web-BerkStan, $G = (V, E)$. Assuming we label all of the vertices $v_i \in V$ where $0 \leq i < |V|$ this is done by creating ten sub graphs $G_i = (V_i, E_i)$ where $V_i = v_j \in V | j \leq i$ and

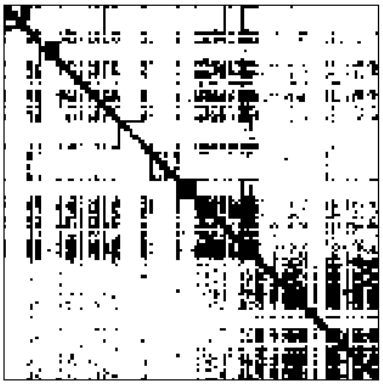


Figure 7.12: The graph used for our experiments is a web crawl of the Berkeley and Stanford networks. Each non-zero entry represents a hyperlink between two pages and the two distinct blocks correspond to the two networks. We chose this graph because it is representative of real-world scale-free networks.

$E_i = E \cap V_i \times V_i$. By using sub-graphs of a range of sizes we can examine the behavior of our library as the graph size changes, but the underlying structure stays the same.

7.4.3 Storing the Graph in the GERMV Object

For all of the experiments we pack the target graph in a GERMV object and perform the spMV and SSSP operations over this packed object. The partition size and number of partitions is dependent on the organization of the cache hierarchy and the size of each cache.

For example, in the graph G_{720896} we partition and store the graph hierarchically according to the blocking dimensions listed in Table 7.1. To illustrate how the graph is captured, we show views of the adjacency matrix at various granularity of the GERMV object in Table 7.2. At each depth of this table we provide spy-plot of the non-zero blocks at that depth, along with a histogram of the density of each of those blocks. Two key observations are that the distribution of block densities follows a power law distribution, and the non-zero blocks are fairly dense despite the graph being very sparse.

Depth	Row Block Size (m_b)	Col Block Size (n_b)
1	720896	720896
2	65536	65536
3	16384	16384
4	4096	4096
5	1024	1024
6	256	256

Table 7.1: These are the GERMV data structure blocking dimensions used in our experiments.

7.4.4 Target Systems

For our experiments we focused on two systems: a large Symmetric Multi Processor (SMP) machine which we will call the Xeon, and small desktop processor which we will call Kaby Lake. The large Xeon machine consists of four Intel Xeon E7-4850 v3 with 14 cores each with 2 threads running at 2.2GHz. The VMWare ESX hypervisor runs on top of this hardware layer and we run our experiments on a virtual machine with 22 cores. The smaller desktop Kaby Lake is an Intel Core i7-7700K with four cores, each running two hardware threads running at 4.2GHz.

We selected these two machines for the following reason: First, they represent two distinct microarchitectures which we have evaluated for other operations. Second, they span a wide range of memory hierarchies, from single processor to SMP. Last, one machine represents large memory and caches whereas the other is representative of small memories and caches. What we want to show is that our graph library is applicable at the ends of each of these spectra.

7.4.5 Analysis of spMV

The spMV implementation is at the core of our graph framework which is why in these series of experiments we evaluate the efficiency of our implementation. We expect that this implementation is efficient for three distinct reasons and we should see a corresponding behavior in the results for each reason when the dataset is small, medium and large. Firstly, we generated and tuned our spMV kernels to be efficient when the input vectors are cache resident. In the performance results, our implementation will be efficient for small sizes. Secondly – in order to efficiently use the cache – we pack the graph in a GERMV object in a blocked hierarchical fashion. Thus, we expect our spMV to perform the most efficiently when the vectors are small enough to fill the cache. Once the vectors leave the cache then the performance behavior is dependent on how much computation our implementation does with the given memory bandwidth. Lastly, our implementation aims to minimize the amount of indexing overhead needed and compactly packs the GERMV objects. This insures that we minimize the number of loads from memory needed to perform a computation. Therefore, for problems exceeding the size of the cache should perform near the peak that the memory bandwidth will allow.

Overall Performance. In the Figure 7.13 and Figure 7.14 we show the results for our overall multi-threaded performance experiment. In this experiment, we compare our performance against the CSB [51] and MKL's COO implementation of spMV. The MKL COO implementation is selected as reference implementation and the CSB implementation is the current state-of-the-art. On both machines we used the maximum number of threads available.

Our spMV GERMV efficiently uses the system cache, multiple threads and the memory bandwidth. In both results, our performance is significantly greater than CSB. In particular on the Kaby Lake when the vector is small enough to fit in the cache the performance is 30% greater than CSB. This we attribute to our efficient use of the data cache. It is worth noting that the CSB implementation was not designed for NUMA systems which is why on the Xeon our performance is substantially higher than CSB.

Parallel Scaling. In the experiments shown in Figure 7.15 and Figure 7.16 we compare the performance of our GERMV spMV implementation for various thread counts. The idea is show that our implementation scales as we add additional threads. On the Kaby Lake Figure 7.15 we can divide the results into two parts, in cache and out of cache. For the out of cache behavior our speedup is linear with respect to the number of threads. When the problem size in the cache we get super linear speedup because as we add

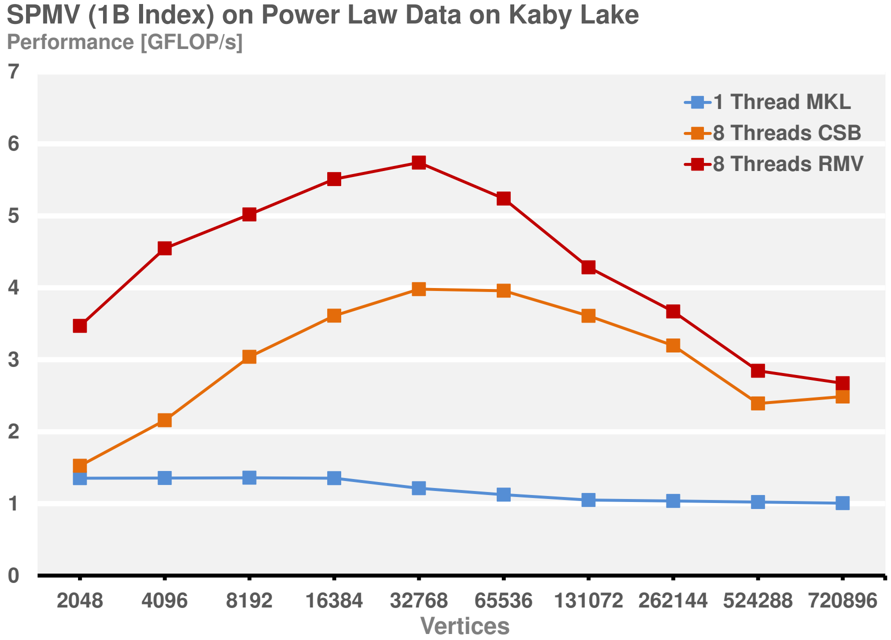


Figure 7.13: In this plot we compare the performance of our spMV implementation against the CSB implementation. In both cases we use all the available threads on the system and for a baseline we show MKL's COO implementation. As we can see our implementation takes advantage of the cache and peaks in performance when the problem size fits in the cache.

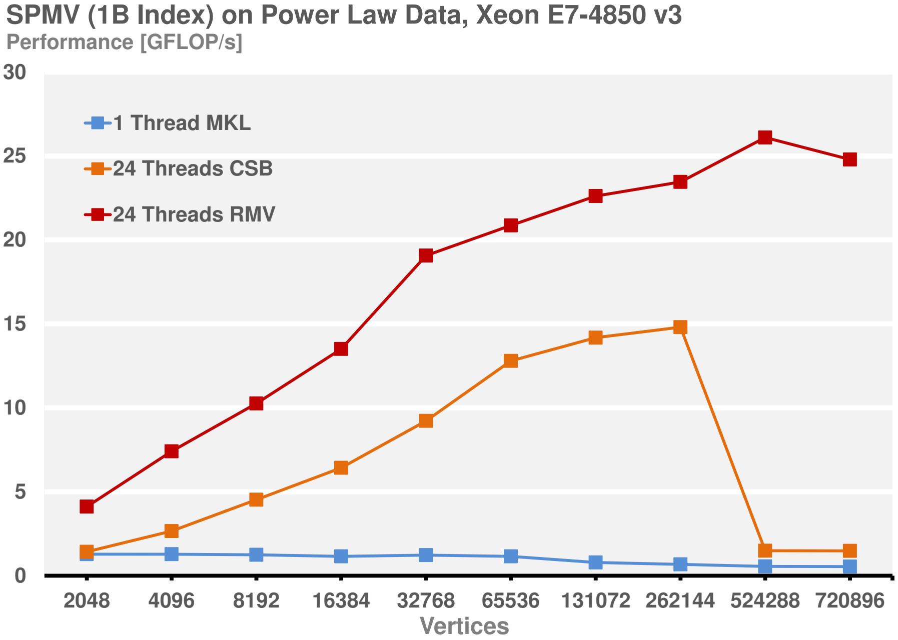


Figure 7.14: Again we compare against CSB but on a much larger system. Our implementation is optimized for SMP systems and therefore performs better.

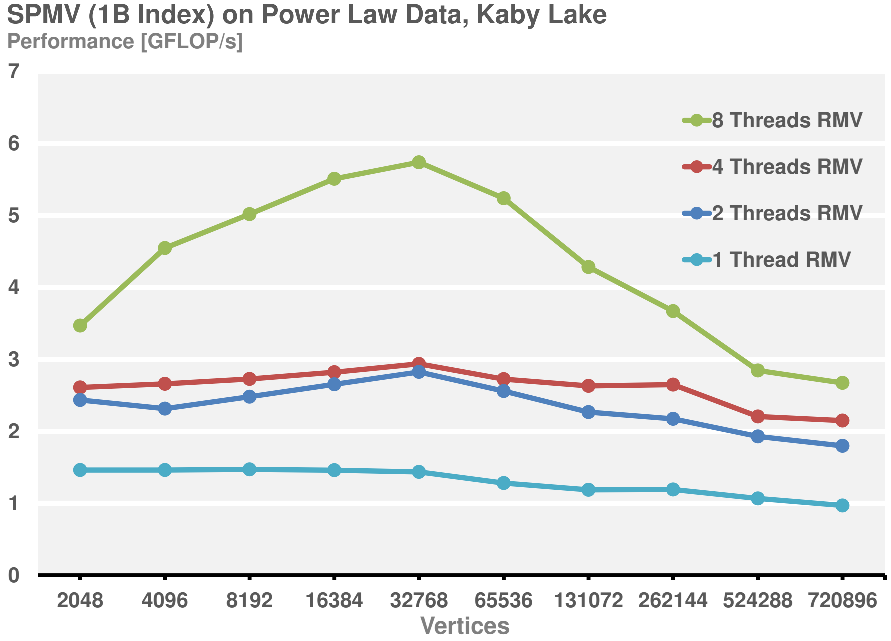


Figure 7.15: Here we compare the scaling performance of our implementation on the Kaby Lake system. When we are using all 8 threads our implementation is able to make the most effective use of the cache on that particular system.

threads our implementation has access to more cache and therefore can have greater reuse. On the Xeon Figure 7.16 we see a more linear speedup as we increase the thread count. The takeaway is that our implementation scales across multiple threads and multiple sockets.

Compact Indexing. Our GERMV implementation allows us to use arbitrary indexing format. This feature allows us to use an indexing format that minimizes the amount of overhead needed for indexing. Because the indices we use will always be smaller than the block sizes selected we only need to use enough bytes to encode those indices. In the experiment in Figure 7.17

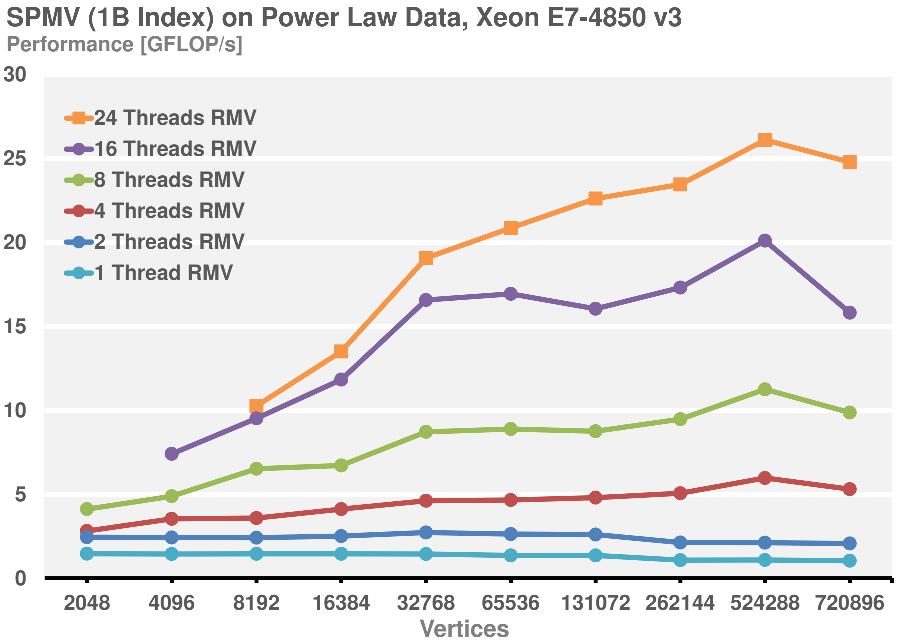


Figure 7.16: Even on larger SMP systems our spMV implementation scales well as we increase the number of threads.

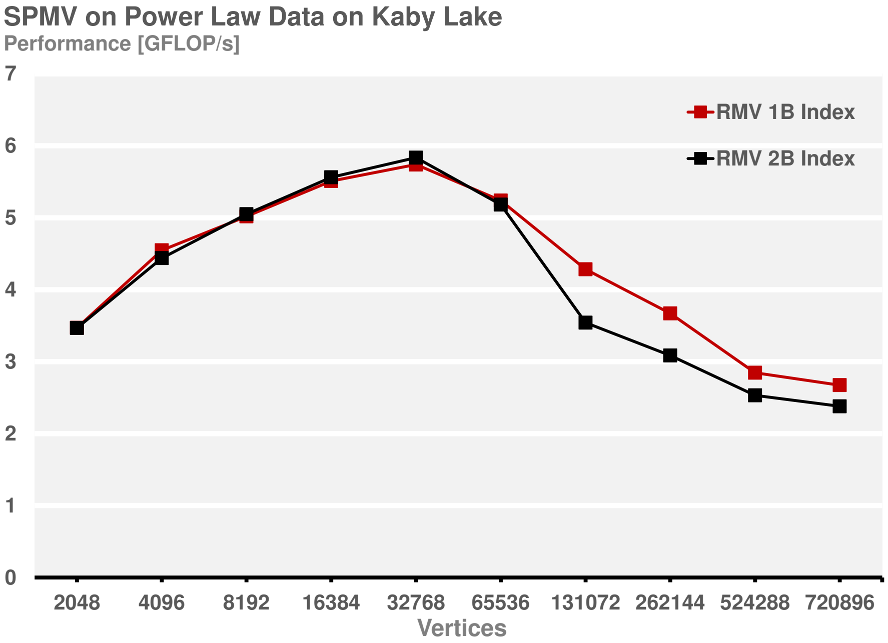


Figure 7.17: We compare the performance of our implementation using 1 Byte indices versus 2B indices which perform better as we leave cache.

we compare the use of 1 Byte indexing versus 2 Bytes for indexing. When the number of vertices fit in the cache there is no difference in performance. However, the moment our implementation needs to access main memory the extra overhead reduces performance.

7.4.6 Analysis of SSSP

In our first SSSP experiment (Figure 7.18) we compare the behavior our spMV implementation, $y = Ax$, versus our iterative matrix-vector approach $y = A^k x$. When the graph exceeds the size of the cache for both implementations the performance starts to decrease. In the spMV case there is no opportunity to reuse however in the iterative case we can reuse the graph data and gain benefits from cache locality.

In the next experiment in we test this idea of reusing the graph across multiple iterations through time-tiling. We compare our standard iterative matrix vector approach to a recursive iterative –time tiled – implementation of SSSP. In Figure 7.19, we compare the throughput of both implementations and we see that the time tiled version (recursive iteration) has a higher throughput than the standard iterative approach. In the Figure 7.20 we show the run-time and by time-tiling we are able to reduce the run-time of our implementation. This is because we are reusing the graph within the cache.

Our next experiments (Figure 7.21) we show that our recursive iterative approach scales as we increase the number of threads. As we increase the number of threads the performance improves proportionally. This empirically demonstrate that for SMP machines our time-tiled approach does not preclude the use parallelism.

Finally in Figure 7.22 we compare our implementation against Ligra for SSSP. When the graph is in the cache, we significantly outperform Ligra. However, as we leave cache they perform better by 6×. This is because between iterations only a small amount of data needs to be read and written. However the use of dense vectors for intermediate results does not confer with that need. If we were to use sparse vectors to reduce the bandwidth requirements for intermediate results then we should be able to reduce that overhead significantly.

7.4.7 Analysis of Pagerank

In our final experiment, we look at PageRank which would not benefit from the use of sparse vectors for intermediates. In Figure 7.23 we show a comparison between our implementation and that of Ligra's. For all problem sizes and thread counts we outperform their implementation using the standard iterative approach. We suspect that by adding time-tiling to our implementation we should see even greater gains in performance because performance peaks when the problem size fills the cache ($|V| = 32768$).

7.5 Chapter Summary

In this chapter, we used graph analytics of scale-free real-world graphs as an example to demonstrate our thesis that if there is a structure in the prob-

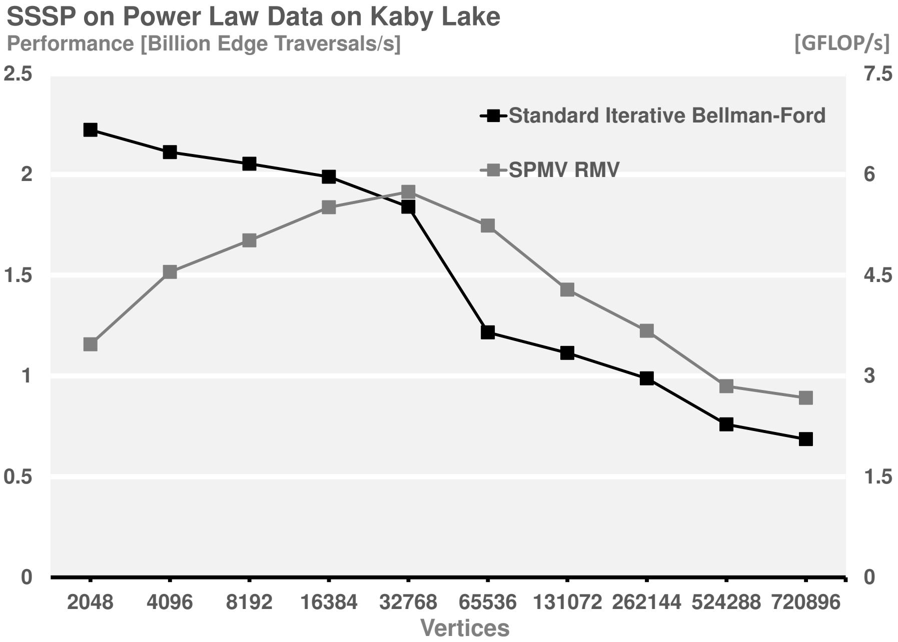


Figure 7.18: In this experiment we compare the performance behavior of our spMV implementation again our iterative matrix-vector SSSP implementation. As we can see the performance of this standard iterative approach decrease substantially once the problem is no longer in the cache.

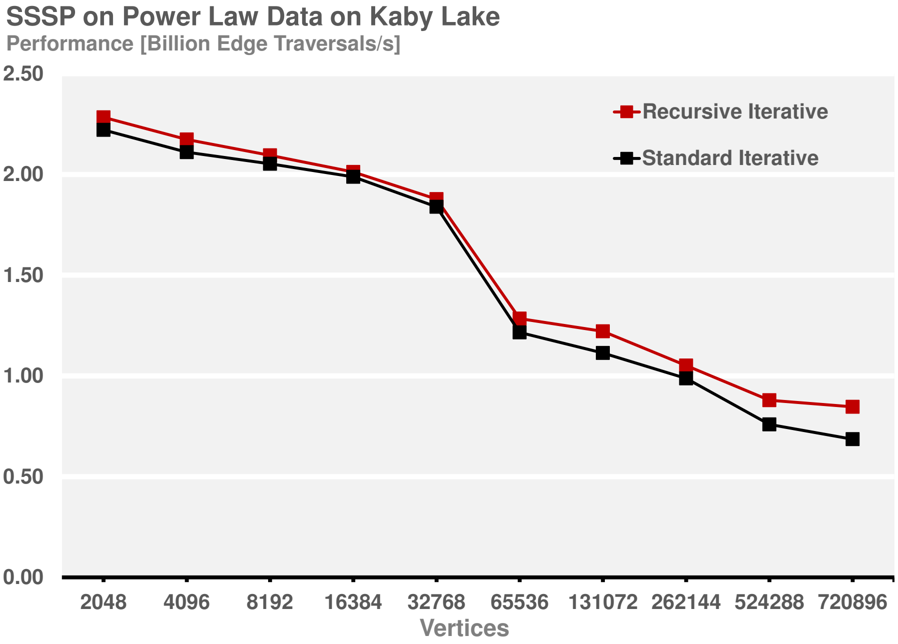


Figure 7.19: In this plot we compare the throughput of the standard iterative SSSP implementation against a time-tiled version.

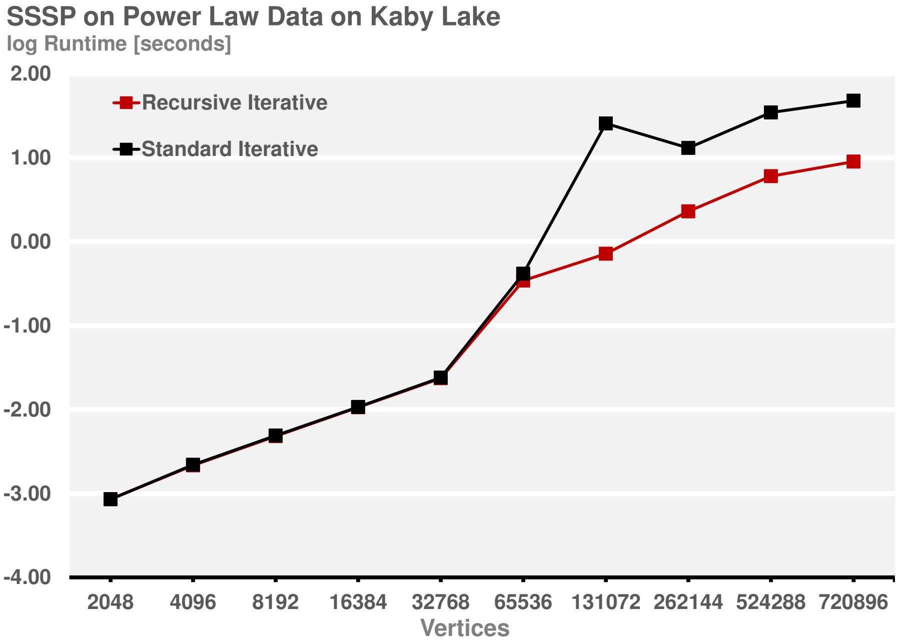


Figure 7.20: We compare the run-time of the standard iterative SSSP implementation against the recursive iterative –time-tiled– implementation.

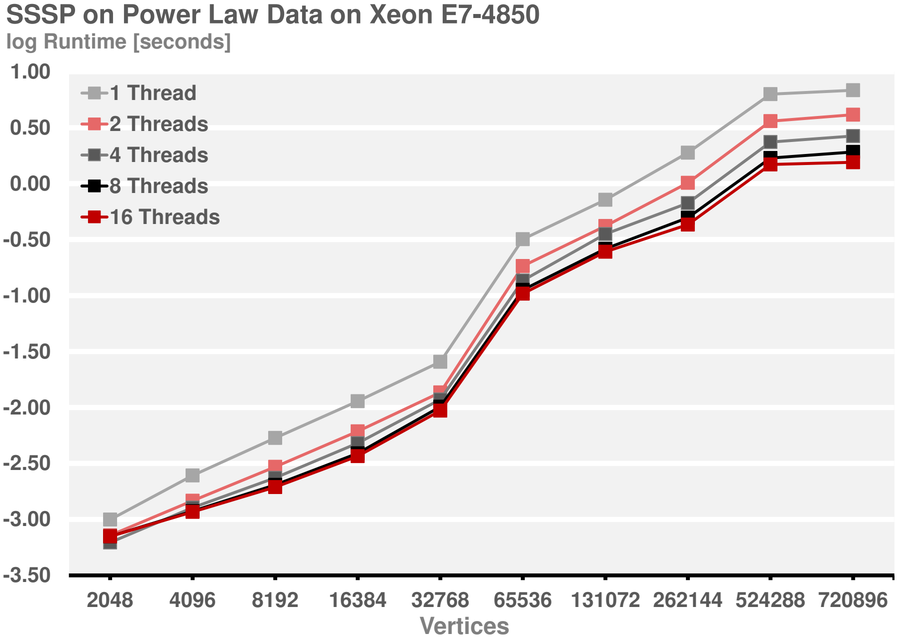


Figure 7.21: As we increase the number of threads the performance of our recursive iteration implementation improves proportionally.

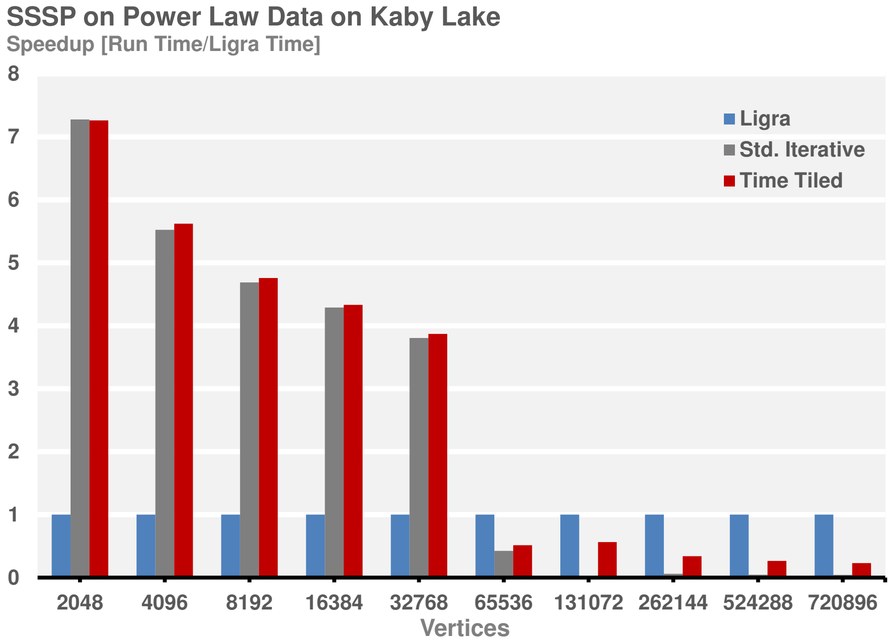


Figure 7.22: As the graph is in the cache our implementation outperforms Ligra for SSSP. However once we leave cache our advantage for this problem disappears. This is because we use dense vectors to store intermediate results which consumes a great deal of bandwidth. We could recover the advantage if we were to use sparse vectors to store intermediate results.

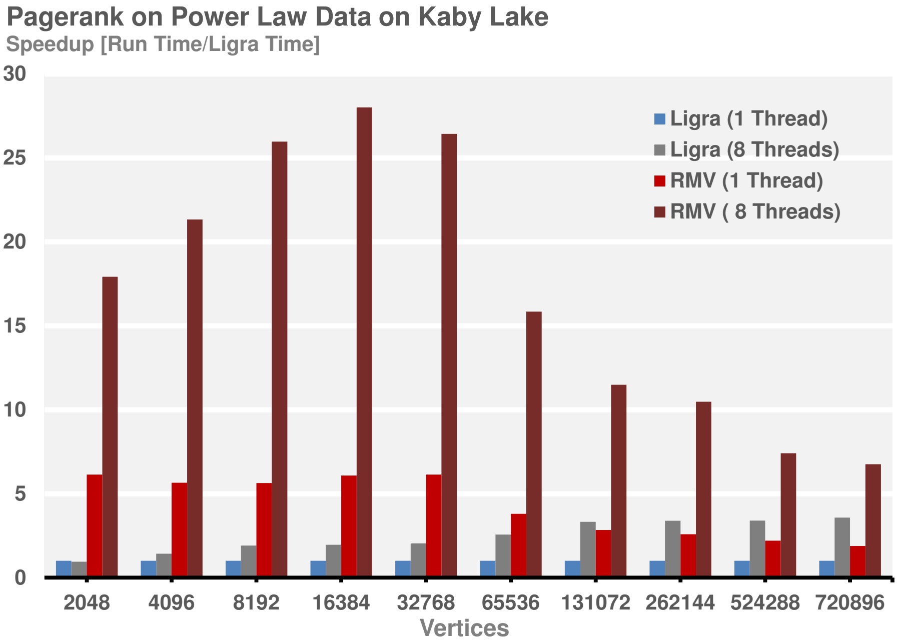


Figure 7.23: For PageRank we outperform Ligra for all problem sizes. This is because this problem is amenable to the use of dense vectors for intermediate results and fits naturally in our framework.

lem then we can develop a high performance operation using data layout transformations and kernel code generation. We found that we could use the hierarchical clustering behavior of real-world graphs to guide how we partitioned the graph which in turn determined how we transformed the data layout. This gave us the benefit that neighboring vertices, which communicated frequently with each other, would be cache adjacent. Because these small sub-graphs are cache resident we can use tuned kernels to process them. Thus we use the same techniques from our Matrix-Matrix Multiplication and Stencil examples to generate high performance graph kernels. By combining these two pieces, efficient layout and kernels, our spMV and PageRank outperform the state of the art.

Our implementation assumes the input graph is efficiently ordered such that the diagonal blocks represent clustered communities. What we found is that for most datasets where the elements are stored in the order they are collected that ordering is sufficient. We can improve the ordering using graph partitioning at the expense of an additional preprocessing step. Alternatively, we want to focus on how the data is collected and how we can modify this step to improve the ordering of the resulting data. The rationale is to match the traversal of the desired graph algorithm to the traversal. This essentially integrates the data layout transformation in the collection and insure that data arrives and is stored in the order that it will be computed.

We only focused on two graph operations in this chapter, SSSP and PageRank, because they are representative of many other graph algorithms that can be implemented in terms of an iterative spMV over an operation specific semiring. In future work we would like to push the frontier on these other graph operations such as other classical graph operations (Minimum Spanning Tree and All Pairs Shortest Path), centrality (Betweenness and Page Rank), clustering (Triangle counting) and inference (belief propagation). We believe that these operations can be cast in terms of linear algebra and that the techniques developed here will extend.

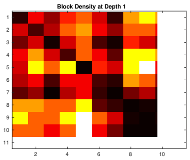
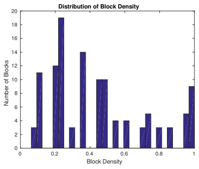
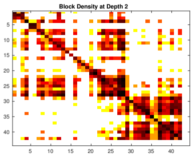
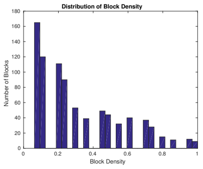
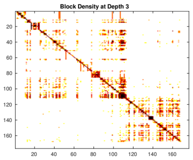
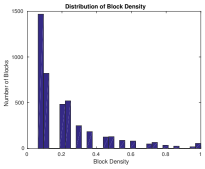
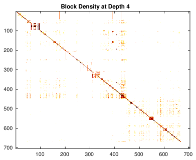
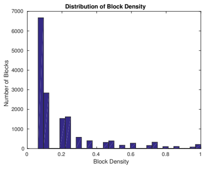
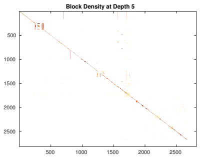
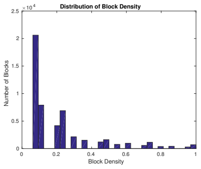
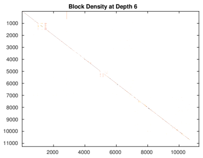
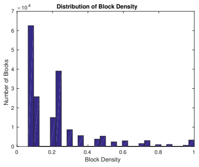
Density of Non-NULL Blocks	Density Distribution
	
	
	
	
	
	

Table 7.2: We store our target data set – web-BerkStan – hierarchically in our GERMV object. In this table we show the density of the data stored at each depth of this hierarchy. On the **left** we provide a heatmap that shows the density of each block, and on the **right** we have a histogram comparing the densities of these blocks.

Part III

Moving Forward

Chapter 8

Summary

In this thesis we discussed a systematic approach for producing high performance implementations for several classes of Linear Algebra-like operations. This process entails using the structure of the target dataset to determine both an efficient access pattern and an efficient kernel to couple with that access pattern. We illustrate this process in Figure 8.1 where we show that structure determines how we apply divide-and-conquer algorithms for our operation, which in turn determines how we arrange our data in memory. We can extract the kernel from the access pattern and generate a specialized kernel tuned to the structure of the problem. The result is a tuned computational kernel being fed data from an efficiently packed data container.

We started this thesis by showing that this premise holds for Matrix-Matrix Multiplication. Existing work gave us the requisite access pattern and packed data, we finished this approach by automating the generation of high performance kernels. For problems outside of Dense Linear Algebra (DLA) this performance hinges on the problem containing a usable structure and we show that indeed these structures fall out naturally in several large domains. We used stencils – structured mesh – computations to demonstrate how we can use this mechanical process on problems with extremely regular structure, namely problems that arise from the application of Finite Difference Methods. We took this a step further by examining the implementation of high performance Sparse Matrix-Vector Multiplication (spMV) over synthetic scale-free graphs. To achieve this we developed a hierarchical data storage format for packing these extremely sparse graphs in a locality preserving manner. This storage format allowed us to implement efficient algorithms and generate kernels tuned to the scale-free structure. We extended

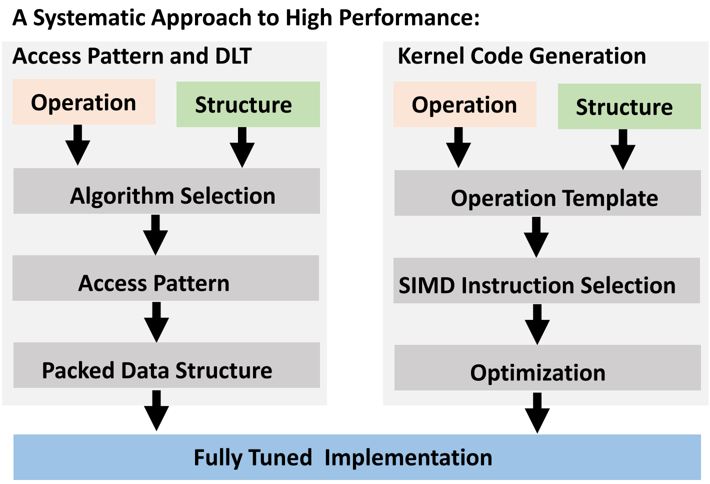


Figure 8.1: This is the mechanical process that we use to implement high performance operations. We split the problem in two halves, a data access and a kernel generation portion. Both halves are implemented and optimized separately and combined to produce a high performance piece of code.

our format to accommodate real-world scale-free data. Finally, we built a graph framework over this data structure to demonstrate that the same level of performance seen in DLA, stencil computation and spMV is achievable for graph analytics. In each of these four cases our implementations outperformed the state-of-the-art. It is in our understanding of the structure of the problem that enabled a systematic approach for obtaining performance.

In this section, we discussed where we are, in the following section we will discuss the various paths that this work is leading and in the final section we will discuss where we want to go.

8.1 Next Steps

Each of our four targeted domains open the door to many of their own questions. In this section we will look at a few of the key next steps towards answering those questions. In the following section, we will look at these domains as a whole and discuss the far reaching questions that we want to explore.

Matrix Multiply and Dense Linear Algebra. For dense Matrix-Matrix Multiply we devised an analytical approach for determining SIMD instructions and a corresponding code generator for implementing expert level kernels with these instructions. We focused on traditional out-of-order and in-order processors, however we can extend our techniques to more esoteric devices and accelerators. Additionally, the GEMM kernel we implement represents only one of six different algorithms which are optimal for different input shapes. While each of these algorithms will present their own unique challenges, we are confident that once we understand those challenges that we can generate their kernel code. Lastly, we want to extend our Matrix-Matrix Multiply kernel generation system to accommodate arbitrary semirings. This would easily allow us to generate mixed precision implementations, along with specialized kernels for graph analytics, computationally biology, statistic and machine learning. We would see this type of generic high performance kernel generator as the basis for all of kernels in the other domains that we have discussed.

Stencil Computations. For stencil computations we developed a mechanical process for generating efficient stencil code. This entailed the use of a polyhedral compiler to produce an efficient loop structure and coupling that with highly tuned generated SIMD kernels. Once again we would like to extend this approach to different target architectures as well as different stencil operations. Additionally, we want to create a hierarchical data structure for storing intermediate data during stencil computation. Such a structure should allow us to produce even more efficient code by simplifying index computation and insuring contiguous memory access. Most importantly, we want to change how we implement these operations. Currently, we take a compiler-like approach to a simple C implementation of the desired stencil. However, we would like to raise this level of abstraction. For example, if we take a library approach then notions such as spacial blocking and time-tiling would be viewed as algorithms and not loop transformations. We could imagine this as a BLAS or LAPACK like library for stencil computations. Ideally, we would like to raise this abstraction even further and focus on the differential operators as written in a Domain Specific Language (DSL). At this level we could decide which approximation scheme would work best with respect to both the problem and the hardware.

Graph Analytics and Sparse Matrix Computations. For spMV and Graph Analytics we provided a data structure for storing real-world and synthetic scale-free data in a manner that permitted the use of the same techniques used for Matrix-Matrix Multiplication and Stencil Computations. This depended on our understanding of our target data set had a hierarchical clustering structure which appears in scale-free data. Using our data structure along with the appropriate divide and conquer algorithms and efficient kernels we are able to produce efficient spMV and Graph Analytic code. One our challenges was in dealing with intermediate sparse data which appear in certain graph operations like Single Source Shortest Path (SSSP). An ideal solution will most likely involve a further understanding of how the operations evolves during computation and how to leverage that knowledge in our data format. Following this, we need to examine more real-world datasets to see how far our understanding takes us or if modifications are needed for our data structure. Currently, we use search to determine blocking sizes for our structure, but if we can approximate our target graph using a graph model then we could analytically determine our blocking parameters. Lastly, how datasets are collected and stored impact the performance of the operations performed on them. We would like to integrate our storage format with the collection process. Doing so would give us greater control over the performance in the later stages of the analytic pipeline without the overhead of partitioning and repacking. We suspect that operating at the beginning of the pipeline may give us a greater understanding of the data we are ultimately computing on.

8.2 Future

Implementations. One of the most immediate but far reaching extension of this thesis is a generalized kernel generator. The techniques used for producing kernels in the four domains we discussed were all similar. The differences resulted from the problem structure which opens up two questions. The first question, is if we can formally express this structure, which we will discuss later in the section. The second question, can we create a generalized kernel code generation framework that takes this structure and produces efficient code.

Algorithms. Earlier we touched on the idea that we can express spacial and temporal blocking at the algorithmic level instead of at the implementation level, which requires analysis. Therefore, if blocking for graphs is accessible at the algorithmic level, then can we express operations like PageRank or SSSP in terms of blocked – or divide-and-conquer – algorithms, as opposed to implementing the traditional scalar operations and performing loop transformations to obtain space and time blocking? We suspect that this would require us to defining these graph algorithms in terms of hierarchically partitioned graphs. Thus we may need to describe our input graphs in terms of hierarchical partitions.

Operations. When we briefly reviewed the stencil operations, we hinted at the idea of expressing these operations as a DSL of difference equations. We could imagine a system like Spiral where we can express our operation in mathematical terms and the system would enumerate and select the optimal nesting of algorithms. Stencil operations naturally lend themselves to mathematical expression, but what would the DSL for graph operations look like? We suspect there will be analogs to the differential operators which characterize the behavior of the graph. Regardless of what this language would look like we can say with certainty that it is at the intersection of the operation and the dataset where we decide the optimal nesting of algorithms. Once again this would require knowledge of the particular graph and its structure.

Structure. In this discussion of future work we stated the need for a language to express structure in graphs. Because we target machines with deep memory hierarchies which require divide-and-conquer algorithms, we would want this language to capture the hierarchical structure in the graph. Additionally, we want this language to capture the locality of neighboring elements because we want this locality realized in the cache. Lastly, graphs are domain specific and we expect that the scientist collecting these graphs are domain experts. We want this scientist to be able to easily convey her knowledge of this dataset in this structural language.

What we propose is a language for expressing the data in terms of a metric space and a set of hierarchical partitions which we illustrate in Figure 8.2. First, the domain expert would define a metric d such that $d(a, b) < d(a, c)$ if the elements a and b communicate important information to each other more often than a and c. This notion of communication would be context specific

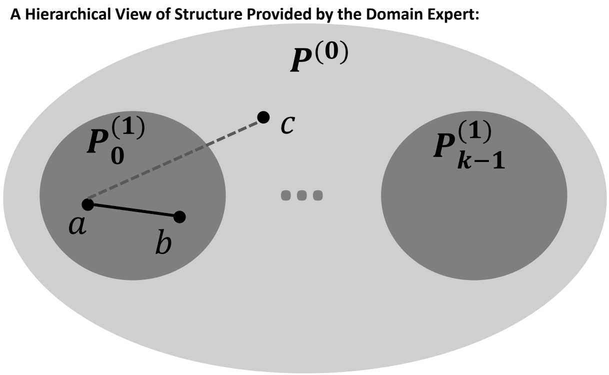

A Hierarchical View of Structure Provided by the Domain Expert:
$P^{(0)}$
$P_0^{(1)}$
c
a
b
$P_{k-1}^{(1)}$

Figure 8.2: We envision a metric space language for capturing the structure of graphs and similar datasets. The domain expert captures the importance of neighboring elements in terms of a distance metric $d(a, b)$ where proximity conveys significant communication. Additionally, the domain expert provides a hierarchical partitioning of the dataset where elements within a cluster are closer than elements outside. i.e. $d(a, b) < d(a, c)$ if $a, b \in P$ and $c \notin P$. This structural description is sufficient for us to apply our mechanical process for high performance.

and entirely determined by the domain expert. Second, the expert would define a hierarchical partitioning P on their dataset S such that $P^{(n-1)} = \bigcup_i P_i^{(n)}$, $\emptyset = \bigcup_i P_i^{(n)}$. Additionally, we would add the constraint that these partitions capture neighborhoods such that if and only if $a, b \in P^{(n)}$ and $c \notin P^{(n)}$ then $d(a, b) < d(a, c)$. The depth n would also be determined by the expert as the number of hierarchical cluster and $P^0 = S$. While this partitioning could be determined numerically once the distance metric is given, we suspect that the expert would have greater insight into the structure of their problem.

With these two details of the data we can apply the mechanical process described in this thesis for producing high performance code. The partitioning allows us to express and select algorithms in terms of said partitions. With this algorithm nest and the expert provided distance metric we can lay the graph data in memory such that neighboring elements – as determined by the expert – will be closer in memory than non-neighboring elements. Lastly, the distance metric would allow us to generate kernels which are tuned to the structure of the data. Therefore, from this structural information we could mechanically generate an efficient implementation.

8.3 Closing Thoughts

Today we observe big data, we rely on existing data and collection techniques and the current view on big data states that these datasets lack structure and organization. This assumption governs what operations we select, how we design our algorithms how we implement our code and even how we design our hardware. In this thesis we demonstrated that performance comes from the application of our understanding of the problem and its structure. We show that even for graph analytics that usable structure indeed exists and it enables the same optimizations necessary for high performance. In the future, our ability to conduct larger and more expensive computational social science experiments will continue to grow. Thus in order to order to efficiently conduct these computational experiments it will become increasingly important to characterizing the structure of the data we collect. Ultimately, our understanding of the underlying phenomena will drive this characterization and in turn our ability to achieve high performance for these experiments.

Bibliography

[1] F. G. Van Zee and R. A. van de Geijn, “Blis: A framework for rapid instantiation of blas functionality,” *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 14:1–14:33, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454 3, 7, 41, 42

[2] K. Goto and R. van de Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1356052.1356053 3, 5, 6, 7, 22, 31, 41, 43, 62, 63, 67, 77

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The landscape of parallel computing research: A view from berkeley,” Technical Report, UC Berkeley, Tech. Rep., 2006. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf 7, 9

[4] A. G. Gray, “Bringing tractability to generalized n-body problems in statistical and scientific computation,” Carnegie Mellon University, School of Computer Science, Tech. Rep. CMU-CS-04-189, April 2003. [Online]. Available: http://reports-archive.adm.cs.cmu.edu/anon/2003/CMU-CS-03-222.pdf 10

[5] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, “Mathematical foundations of the graphblas,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sept 2016, pp. 1–9. [Online]. Available: https://doi.org/10.1109/HPEC.2016.7761646 10, 36, 137, 138

[6] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, “Recent advances in graph partitioning,” *CoRR*, vol. abs/1311.3144, 2013.

[Online]. Available: http://arxiv.org/abs/1311.3144 17

[7] T. M. Smith, R. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1049–1059. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.110 21, 68

[8] J. D. C. Little, "A proof for the queuing formula: L = λ w," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961. [Online]. Available: http://dx.doi.org/10.1287/opre.9.3.383 25, 51

[9] R. Veras, D. T. Popovici, T. M. Low, and F. Franchetti, "Compilers, hands-off my hands-on optimizations," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '16. New York, NY, USA: ACM, 2016, pp. 4:1–4:8. [Online]. Available: http://doi.acm.org/10.1145/2870650.2870654 26, 27, 55, 61

[10] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 318–328. [Online]. Available: http://doi.acm.org/10.1145/53990.54022 26, 55, 61

[11] P. Bientinesi, V. Eijkhout, K. Kim, J. Kurtz, and R. van de Geijn, "Sparse direct factorizations through unassembled hyper-matrices," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, pp. 430–438, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045782509002333 30, 36

[12] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, March 1990. [Online]. Available: http://doi.acm.org/10.1145/77626.79170 30

[13] E. Anderson, Z. Bai, C. Bischof, L. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Society for Industrial and Applied Mathematics, 1999. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9780898719604 30

[14] K. Goto and R. van de Geijn, "High-performance implementation

of the level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, no. 1, pp. 4:1–4:14, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1377603.1377607 30

[15] B. Kågström, P. Ling, and C. van Loan, "Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, pp. 268–302, Sept. 1998. [Online]. Available: http://doi.acm.org/10.1145/292395.292412 30

[16] J. Bilmes, K. Asanović, C. whye Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 340–347. [Online]. Available: http://doi.acm.org/10.1145/263580.263662 31

[17] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509096 31, 69

[18] K. Yotov, X. Li, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance BLAS?" *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 358–386, Feb 2005. [Online]. Available: https://doi.org/10.1109/JPROC.2004.840444 31

[19] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapid instantiation of blas functionality," *ACM Trans. Math. Soft.*, 2013, in review. 31

[20] T. M. Low, F. D. Igual, T. Smith, and E. S. Quintana, "Analytical modeling is enough for high performance BLIS," *ACM Trans. Math. Soft.*, vol. 43, no. 2, pp. 12:1–12:18, August 2016. [Online]. Available: http://doi.acm.org/10.1145/2925987 31, 44, 62, 63, 67, 68

[21] R. M. Veras, T. M. Low, T. M. Smith, R. A. van de Geijn, and F. Franchetti, "Automating the last-mile for high performance dense linear algebra," *CoRR*, vol. abs/1611.08035, 2016. [Online]. Available: http://arxiv.org/abs/1611.08035 31, 51

[22] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 23:23–23:32. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544155 31

[23] N. Kyrtatas, D. G. Spampinato, and M. Püschel, "A basic linear algebra compiler for embedded processors," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1054–1059. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2757058 31

[24] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler for structured matrices," in *International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '16. New York, NY, USA: ACM, 2016, pp. 117–127. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854060 31

[25] J. G. Siek, I. Karlin, and E. R. Jessup, "Build to order linear algebra kernels," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IPDPS.2008.4536183 31

[26] M. Bromley, S. Heller, T. McNerney, and G. L. Steele, Jr., "Fortran at ten gigaflops: The connection machine convolution compiler," *SIGPLAN Not.*, vol. 26, no. 6, pp. 145–156, May 1991. [Online]. Available: http://doi.acm.org/10.1145/113446.113458 32

[27] R. G. Brickner, W. George, S. L. Johnsson, and A. Ruttenberg, "A stencil complier for the connection machine models cm-2/200," 1993. [Online]. Available: http://ftp.deas.harvard.edu/techreports/tr-01-93.ps.gz 32

[28] R. Brickner, K. Holian, B. Thiagarajan, and S. Johnsson, *Designing a stencil compiler for the Connection Machine model CM-5*, Dec 1994. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/10119048 32

[29] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. SC '00. Washington, DC, USA: IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=

370049.370403 32

[30] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, "Impact of modern memory subsystems on cache optimizations for stencil computations," in *Proceedings of the 2005 workshop on Memory system performance*, ser. MSP '05. New York, NY, USA: ACM, 2005, pp. 36–43. [Online]. Available: http://doi.acm.org/10.1145/1111583.1111589 32

[31] J. McCalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," Technical Report DCS-TR-379, Department of Computer Science, Rugers University, Tech. Rep., 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.9428 32, 141

[32] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 215–228, May 1999. [Online]. Available: http://doi.acm.org/10.1145/301631.301668 32

[33] D. Wonnacott, "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '00, IEEE. Washington, DC, USA: IEEE Computer Society, 2000, pp. 171–. [Online]. Available: http://dl.acm.org/citation.cfm?id=846234.849346 32

[34] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 115–133, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004041295 33

[35] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, ser. MSPC '06. New York, NY, USA: ACM, 2006, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/1178597.1178605 33

[36] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–12. [Online]. Available:

https://doi.org/10.1109/IPDPS.2010.5470421 33

[37] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 676–687. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.70 33

[38] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508 33

[39] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595 33

[40] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," *SIGPLAN Not.*, vol. 48, no. 6, pp. 127–138, June 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462187 33, 79, 97

[41] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2464996.2467268 33, 95

[42] I. S. Duff, "A survey of sparse matrix research," *Proceedings of the IEEE*, vol. 65, no. 4, pp. 500–535, 1977. [Online]. Available: https://doi.org/10.1109/PROC.1977.10514 34

[43] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 239–267, June 2002. [Online]. Available: http://doi.acm.org/10.1145/567806.567810

34

[44] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations," 1994. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.3853 34

[45] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar, "Next-generation generic programming and its application to sparse matrix computations," in *Proceedings of the 14th international conference on Supercomputing*, ser. ICS '00, ACM. New York, NY, USA: ACM, 2000, pp. 88–99. [Online]. Available: http://doi.acm.org/10.1145/335231.335240 34

[46] N. Ahmed, N. Mateev, and K. Pingali, "A framework for sparse matrix code synthesis from high-level specifications," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, Nov 2000, pp. 58–58. [Online]. Available: https://doi.org/10.1109/SC.2000.10033 34

[47] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004041296 34

[48] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.1940 34

[49] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465013 34

[50] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999. [Online]. Available: http://dx.doi.org/10.1126/science.286.5439.509 35, 135

[51] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-

transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 233–244. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584053 35, 162

[52] A.-J. N. Yzelman and R. H. Bisseling, "A cache-oblivious sparse matrix–vector multiplication scheme based on the hilbert curve," in *Progress in Industrial Mathematics at ECMI 2010*. Springer, 2012, pp. 627–633. [Online]. Available: https://doi.org/10.1007/978-3-642-25100-9_73 35

[53] R. Veras, T. M. Low, and F. Franchetti, "A scale-free structure for power-law graphs," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2016, pp. 1–7. [Online]. Available: https://doi.org/10.1109/HPEC.2016.7761608 35

[54] T. M. Low and R. van de Geijn, "An API for manipulating matrices stored by blocks," Department of Computer Sciences, The University of Texas at Austin, Tech. Rep. TR-2004-15, May 2004. 35, 144

[55] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184 35

[56] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, April 2012. [Online]. Available: https://doi.org/10.14778/2212351.2212354 35

[57] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. New York, NY, USA: ACM, Feb. 2013, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/2517327.2442530 35

[58] A. Kyrola, G. E. Blelloch, C. Guestrin *et al.*, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–

46. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387884 35

[59] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: edge-centric graph processing using streaming partitions,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13. New York, NY, USA: ACM, 2013, pp. 472–488. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522740 36

[60] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, “The tao of parallelism in algorithms,” in *ACM Sigplan Notices*, vol. 46, no. 6. New York, NY, USA: ACM, June 2011, pp. 12–25. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993501 36

[61] D. Prountzos, R. Manevich, and K. Pingali, “Elixir: A system for synthesizing concurrent graph programs,” *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 375–394, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2398857.2384644 36

[62] A. Buluç and J. R. Gilbert, “The combinatorial blas: Design, implementation, and applications,” *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342011403516 36

[63] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader, “STINGER: high performance data structure for streaming graphs,” in *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, 2012, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2012.6408680 36

[64] K. Kim and V. Eijkhout, “A parallel sparse direct solver via hierarchical dag scheduling,” *ACM Trans. Math. Softw.*, vol. 41, no. 1, pp. 3:1–3:27, Oct. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629641 36

[65] http://xianyi.github.com/OpenBLAS/, 2012. 43, 69

[66] D. A. Padua and M. J. Wolfe, “Advanced compiler optimizations for supercomputers,” *Commun. ACM*, vol. 29, no. 12, pp. 1184–1201, Dec. 1986. [Online]. Available: http://doi.acm.org/10.1145/7902.7904 43

[67] G. Henry, “BLAS based on block data structures,” Cornell University,

Theory Center Technical Report CTC92TR89, Feb. 1992. [Online]. Available: https://ecommons.cornell.edu/handle/1813/5471 43

[68] F. G. Van Zee, T. Smith, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, T. M. Low, B. Marker, L. Killough, and R. A. van de Geijn, "The BLIS framework: Experiments in portability," *ACM Transactions on Mathematical Software*, vol. 42, no. 2, pp. 12:1–12:19, June 2016. [Online]. Available: http://doi.acm.org/10.1145/2755561 43

[69] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Sept. 2015. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html 52

[70] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.6801 77

[71] L.-N. Pouchet, "Polyopt/c: A polyhedral optimizer for the rose compiler," 2012. [Online]. Available: http://hpcrl.cse.ohio-state.edu/wiki/index.php/polyopt/c 79, 97, 106

[72] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation". [Online]. Available: https://doi.org/10.1109/JPROC.2004.840306 80, 97

[73] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan, "Parameterized tiling revisited," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 200–209. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772983 105

[74] A.-L. Barabsi, R. Albert, and H. Jeong, "Mean-field theory for scale-free random networks," *Physica A: Statistical Mechanics and its Applications*, vol. 272, no. 12, pp. 173 – 187, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/

pii/S0378437199002915 117

[75] ——, "Scale-free characteristics of random networks: the topology of the world-wide web," *Physica A: Statistical Mechanics and its Applications*, vol. 281, no. 14, pp. 69 – 77, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378437100000182 117, 134

[76] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, March 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756039 127, 141, 144

[77] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, Sept. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324140 134, 135

[78] S. Dill, R. Kumar, K. S. Mccurley, S. Rajagopalan, D. Sivakumar, and A. Tomkins, "Self-similarity in the web," *ACM Trans. Internet Technol.*, vol. 2, no. 3, pp. 205–223, Aug. 2002. [Online]. Available: http://doi.acm.org/10.1145/572326.572328 134

[79] H. Ebel, L.-I. Mielsch, and S. Bornholdt, "Scale-free topology of e-mail networks," *Phys. Rev. E*, vol. 66, p. 035103, Sep 2002. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.66.035103 134

[80] R. Guimerà, L. Danon, A. Díaz-Guilera, F. Giralt, and A. Arenas, "Self-similar community structure in a network of human interactions," *Phys. Rev. E*, vol. 68, p. 065103, Dec 2003. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.68.065103 134, 159

[81] M. E. J. Newman, "Scientific collaboration networks. i. network construction and fundamental results," *Phys. Rev. E*, vol. 64, p. 016131, Jun 2001. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.64.016131 134

[82] ——, "Scientific collaboration networks.ii. shortest paths, weighted networks, and centrality," *Phys. Rev. E*, vol. 64, p. 016132, Jun 2001. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.64.016132 134

[83] C. Chen, "Visualising semantic spaces and author co-citation networks in digital libraries," *Information Processing and Management*,

vol. 35, no. 3, pp. 401 – 420, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306457398000685 134

[84] S. Bilke and C. Peterson, "Topological properties of citation and metabolic networks," *Phys. Rev. E*, vol. 64, p. 036106, Aug 2001. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.64.036106 134

[85] S. Lehmann, B. Lautrup, and A. D. Jackson, "Citation networks in high energy physics," *Phys. Rev. E*, vol. 68, p. 026113, Aug 2003. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.68.026113 134

[86] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 251–262, Aug. 1999. [Online]. Available: http://doi.acm.org/10.1145/316194.316229 134

[87] R. Albert, H. Jeong, and A.-L. Barabasi, "Error and attack tolerance of complex networks," *Nature*, vol. 406, no. 6794, pp. 378–382, Jul 2000. [Online]. Available: http://dx.doi.org/10.1038/35019019 134

[88] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman, "Search in power-law networks," *Phys. Rev. E*, vol. 64, p. 046135, Sep 2001. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.64.046135 134

[89] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298311 134

[90] F. Wu, B. A. Huberman, L. A. Adamic, and J. R. Tyler, "Information flow in social groups," *Physica A: Statistical Mechanics and its Applications*, vol. 337, no. 1, pp. 327–335, 2004. [Online]. Available: https://doi.org/10.1016/j.physa.2004.01.030 134, 136, 139, 143

[91] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos, "Using graph theory to analyze biological networks," *BioData Mining*, vol. 4, no. 1, p. 10, 2011. [Online]. Available: http://dx.doi.org/10.1186/1756-0381-4-10 135

[92] R. Albert, “Scale-free networks in cell biology,” *Journal of Cell Science*, vol. 118, no. 21, pp. 4947–4957, 2005. [Online]. Available: http://jcs.biologists.org/content/118/21/4947 135

[93] D. Garlaschelli, S. Battiston, M. Castri, V. D. Servedio, and G. Caldarelli, “The scale-free topology of market investments,” *Physica A: Statistical Mechanics and its Applications*, vol. 350, no. 2, pp. 491 – 499, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378437104014943 135

[94] D. Garlaschelli and M. I. Loffredo, “Structure and evolution of the world trade network,” *Physica A: Statistical Mechanics and its Applications*, vol. 355, no. 1, pp. 138 – 144, 2005, market Dynamics and Quantitative Economics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378437105002852 135

[95] W.-Q. Huang, X.-T. Zhuang, and S. Yao, “A network analysis of the chinese stock market,” *Physica A: Statistical Mechanics and its Applications*, vol. 388, no. 14, pp. 2956 – 2964, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378437109002519 135

[96] J. R. Banavar, A. Maritan, and A. Rinaldo, “Size and form in efficient transportation networks,” *Nature*, vol. 399, no. 6732, pp. 130–132, May 1999. [Online]. Available: http://dx.doi.org/10.1038/20144 135

[97] R. Guimer, S. Mossa, A. Turtschi, and L. A. N. Amaral, “The worldwide air transportation network: Anomalous centrality, community structure, and cities’ global roles,” *Proceedings of the National Academy of Sciences*, vol. 102, no. 22, pp. 7794–7799, 2005. [Online]. Available: http://www.pnas.org/content/102/22/7794.abstract 135

[98] A.-L. Barabási, “Scale-free networks: A decade and beyond,” *Science*, vol. 325, no. 5939, pp. 412–413, 2009. [Online]. Available: http://science.sciencemag.org/content/325/5939/412 135

[99] D. J. Watts and S. H. Strogatz, “Collective dynamics of /‘small-world/’ networks,” *Nature*, vol. 393, no. 6684, pp. 440–442, Jun 1998. [Online]. Available: http://dx.doi.org/10.1038/30918 135

[100] E. Ravasz and A.-L. Barabási, “Hierarchical organization in complex networks,” *Physical Review E*, vol. 67, no. 2, p. 026112, 2003. [Online].

Available: http://dx.doi.org/10.1103/PhysRevE.67.026112 136

[101] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann, "Link communities reveal multiscale complexity in networks," *Nature*, vol. 466, no. 7307, pp. 761–764, 2010. [Online]. Available: http://dx.doi.org/10.1038/nature09182 136

[102] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabási, "Hierarchical organization of modularity in metabolic networks," *science*, vol. 297, no. 5586, pp. 1551–1555, 2002. [Online]. Available: https://doi.org/10.1126/science.1073374 136

[103] M. Barthélemy, A. Barrat, R. Pastor-Satorras, and A. Vespignani, "Velocity and hierarchical spread of epidemic outbreaks in scale-free networks," *Physical Review Letters*, vol. 92, no. 17, p. 178701, 2004. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.92.178701 136, 143

[104] R. Pastor-Satorras and A. Vespignani, "Epidemic spreading in scale-free networks," *Physical review letters*, vol. 86, pp. 3200–3203, Apr 2001. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.86.3200 136, 139, 143

[105] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.5427 138

[106] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–11. [Online]. Available: https://doi.org/10.1109/IPDPS.2008.4536313 138

[107] D. Braha and Y. Bar-Yam, "Information flow structure in large-scale product development organizational networks," EconWPA, Industrial Organization, 2004. [Online]. Available: http://EconPapers.repec.org/RePEc:wpa:wuwpio:0407012 143

[108] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzaran, G. Almasi, J. Moreira, and D. Padua, "The hierarchically tiled arrays programming approach," in *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, ser. LCR '04. New York, NY, USA: ACM, 2004, pp. 1–12.

[Online]. Available: http://doi.acm.org/10.1145/1066650.1066657 144

[109] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan, and G. Wesley, “Stanford webbase components and applications,” *ACM Trans. Internet Technol.*, vol. 6, no. 2, pp. 153–186, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1149121.1149124 159