Multi-Dimensional Auto-Vectorization of Stencil Codes

by

Shrey Sharma

Master's Thesis

Compiler Design Lab
Faculty of Mathematics and Computer Science
Department of Computer Science
Saarland University

Supervisor
Prof. Dr. Sebastian Hack

Advisor
Simon Moll, M.Sc

Reviewers
Prof. Dr. Sebastian Hack
Prof. Dr. Jan Reineke

August 1, 2019

Declaration of Authorship

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Datum/Date: _____

Unterschrift/Signature: _____

SAARLAND UNIVERSITY
Compiler Design Lab
Department of Computer Science

Abstract

Multi-Dimensional Auto-Vectorization of Stencil Codes

by Shrey Sharma

Vectorization is an important method for improving performance of data parallel computations on architectures with SIMD support. It is commonly used for accelerating scientific simulations, image processing, and multimedia applications. These applications often involve the computation of stencil kernels of single or multiple dimensions. Most vectorization techniques involve selecting a single optimal dimension and vectorizing along that dimension to improve throughput. Recent work has shown improvements in compute performance for 3D stencils when vectorizing in multiple dimensions. In this thesis we build a framework that performs multi-dimensional vectorization efficiently by reducing memory operations. In our experiments we observed average speedups of up to 1.37× in 2D stencil applications and up to 7.48× in the matrix transpose operation.

Acknowledgements

I would first like to thank my supervisor Professor Sebastian Hack for supervising this thesis and providing an excellent working environment. I would also like to thank Professor Jan Reineke for agreeing to review this thesis.

I am grateful to my advisor Simon Moll, for his guidance and support. I thank him for always having time for my questions and being patient when I would go off course during the project.

Further, I would like to extend my thanks to the students and members of the Compiler Design/Real-Time and Embedded Systems Lab. I would like to thank Sebastian Hahn, Fabian Ritter and Tina Jung for proofreading this thesis.

Finally I would like to thank my friends and family for their love and support.

Content

Chapter 1

Introduction

Most modern computer architectures support Single Instruction Multiple Data (SIMD) instruction sets, which operate on sets of data elements called vectors, as opposed to single elements. SIMD instructions use vector registers in the hardware to read operands and store results. Hence, they can simultaneously process elements up to the width of available vector registers. However, writing efficient SIMD code is complex, tedious and error prone. Optimizing compilers use the method of vectorization to convert scalar code, which uses scalar instructions, into vector code, that uses the available SIMD instructions.

Vectorization is commonly used for optimizing performance of stencil codes found in solvers for scientific computations, image processing applications, and computational fluid dynamics simulations. Stencil codes perform element-wise operations on arrays where the value of each element in the output array depends on a fixed pattern of neighboring elements in the input array. Generally, they involve multiple iterations over the input array where the output of each iteration is used as input for the next one. Since iterations over array elements are done using loop nests, loop vectorization is an important technique for improving throughput of such programs. For example, the loop nest in Figure 1.1(A) computes a single iteration of the Jacobi solver for a 9-point, two-dimensional Jacobi stencil. Each element of `B` depends on nine elements of `A` as shown in the stencil pattern in Figure 1.1(A).

Current state-of-the-art optimizing compilers vectorize along a single dimension of a loop nest. The vectorization dimension can be specified by the programmer in the source language or chosen automatically based on cost models in the compiler. The goal is to process the maximum number of elements simultaneously while also minimizing the cost of loading and storing data from and to memory. For a row-major, single-dimensional memory layout, this is achieved by vectorizing along the innermost loop dimension and using contiguous vector load and store instructions. In Figure 1.1(B),

we can see the Jacobi stencil from Figure 1.1(A) being vectorized along the `j` dimension with a vectorization width of 8.

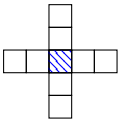
In stencil codes, we often have data overlap between neighboring iterations of the loops that leads to expensive data reloads. In our example, the element `A[2][1]` is loaded twice, once when computing `B[2][2]` and again when computing `B[2][3]`. The elements towards the center of the input matrix can be loaded up to nine times – once for each stencil point. Single-dimensional vectorization can reduce reloads by exploiting the overlap between operand vectors in the middle row. In Figure 1.1(B) we see how six memory loads and three shuffle operations give us the required operand vectors. When the input size is large, the cost of reloading elements in the outer dimension is far greater than that of reloading in the inner dimension. In order to reduce reload along the outer dimension, one may consider outer loop vectorization. However, outer loop vectorization would require expensive gather operations to load input elements as elements are not stored contiguously along the outer dimension.

Multi-dimensional vectorization improves performance by reordering computations to reduce reloads in multiple dimensions. We vectorize along the inner dimension with a width of four elements and unroll two iterations of the outer loop to jam together for a vectorization layout of 2×4 as shown in Figure 1.1(C). We load the elements using contiguous loads along the inner dimension and perform shuffles to construct the required operand vectors. Even though we do not reduce the number of memory loads in each iteration when using this technique, the reduction in number of iterations in the outer loop leads to fewer reloads overall. However, due to the added shuffles, we need more vector registers to perform each iteration, which may lead to register spilling.

SIMD vector lengths have been increasing over the years with the recent Intel AVX-512 standard having 64 vector registers that accommodate up to eight double precision elements. Further, specialized vector hardware like the NEC SX-Aurora have vector registers that can hold up to 256 double precision elements. For multi-dimensional vectorization, we need such long vector registers as shorter registers would limit our choices with respect to the vectorization layout.

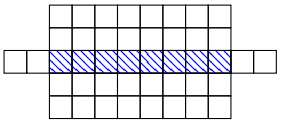
Finding an efficient multi-dimensional vectorization layout is complicated due to the trade-off between cost of memory loads and increased register pressure. Also, given a vectorization layout, writing multi-dimensional vector code is difficult as current compilers do not support multi-dimensional vectorization. However, for stencil codes that have a regular grid-like access pattern we show that the process can be automated. In this thesis we develop a framework that can analyze the scalar code for a stencil pattern and generate efficient multi-dimensional vector code. Our framework analyzes the memory access pattern in all dimensions of the input and forms groups of contiguous

```
for (i=2; i<rows-2; i++) {
  for (j=2; j<cols-2; j++) {
    B[i][j] = 0.2 * (A[i][j]
            + A[i][j-2] + A[i][j-1]
            + A[i][j+2] + A[i][j+1]
            + A[i-2][j] + A[i-1][j]
            + A[i+2][j] + A[i+1][j]);
  }
}
```



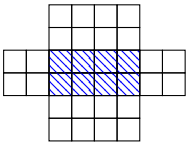
(A) Scalar code for 9 point 2D Jacobi stencil

```
for (i=2; i<rows-2; i++) {
  for (j=2; j<cols-10; j+=8) {
    u1 = *(double8*)(&A[i-1][j]);
    u2 = *(double8*)(&A[i-2][j]);
    d1 = *(double8*)(&A[i+1][j]);
    d2 = *(double8*)(&A[i+2][j]);
    ml = *(double8*)(&A[i][j-2]);
    mr = *(double8*)(&A[i][j+2]);
    c = __builtin_shufflevector(ml,mr,2,3,4,5,6,7,12,13);
    l1 = __builtin_shufflevector(ml,mr,1,2,3,4,5,6,7,12);
    r1 = __builtin_shufflevector(ml,mr,3,4,5,6,7,12,13,14);

    res = 0.2 * (u1 + u2 + d1 + d2 + ml + mr + c + l1 + r1);
    *(double8*)(&B[i][j]) = res;
  }
}
```



(B) Vectorized code along the `j` dimension with vector width 8

```
for (i=2; i<rows-4; i+=2) {
  for (j=2; j<cols-6; j+=4) {
    uh = *(double8*)(&A[i-2][j]);
    ul = *(double8*)(&A[i-1][j]);
    dh = *(double8*)(&A[i+2][j]);
    dl = *(double8*)(&A[i+3][j]);
    m1 = *(double8*)(&A[i][j-2]);
    m2 = *(double8*)(&A[i+1][j-2]);
    u2 = __builtin_shufflevector(uh,ul,0,1,2,3,8,9,10,11);
    u1 = __builtin_shufflevector(ul,m1,0,1,2,3,10,11,12,13);
    c  = __builtin_shufflevector(m1,m2,2,3,4,5,10,11,12,13);
    d2 = __builtin_shufflevector(dh,dl,0,1,2,3,8,9,10,11);
    d1 = __builtin_shufflevector(m2,dh,2,3,4,5,8,9,10,11);
    l1 = __builtin_shufflevector(m1,m2,1,2,3,4,9,10,11,12);
    l2 = __builtin_shufflevector(m1,m2,0,1,2,3,8,9,10,11);
    r1 = __builtin_shufflevector(m1,m2,3,4,5,6,11,12,13,14);
    r2 = __builtin_shufflevector(m1,m2,4,5,6,7,12,13,14,15);

    res = 0.2 * (u1 + u2 + d1 + d2 + l2 + l1 + c + r1 + r2);
    *(double4*)(&B[i][j]) = *(double4*)(&res);
    *(double4*)(&B[i+1][j]) = *((double4*)(&res) + 1);
  }
}
```



(C) Proposed multi-dimensional vectorized code

FIGURE 1.1: Loop nests for 9-point 2D Jacobi stencil: scalar (A), 1D vectorized (B), 2D vectorized (C)

memory accesses so that they can be accessed using SIMD memory instructions to speed up execution.

1.1 Contributions

The contributions of this thesis are as follows:

- We developed the tensor shape analysis as an extension of divergence analysis [1, 2] to multi-dimensional loop nests. The analysis computes tensor shapes [3] for all instructions in a loop nest.
- We show how memory accesses in multiple dimensions can be grouped together using analyses like Scalar Evolution [4] to generate efficient vector code for multi-dimensional stencil codes. Memory access grouping enables us to reduce memory traffic by replacing multiple scalar memory accesses by a single vector memory access.
- We implemented our approach as a fork of Region Vectorizer [5] called *TensorRV*. Our framework takes scalar LLVM IR code, a vectorization layout and a memory data layout as input and generates efficient multi-dimensional vector code. We evaluated *TensorRV* on some common 2D stencils with default memory data layout as well as transformed data layouts. We also evaluated our framework for the matrix transpose operation with the default memory layout. We observed average speedups of up to 1.37× for stencil benchmarks and 7.48× for matrix transpose in our experiments using AVX-512 as our target SIMD ISA.

Chapter 2

Background

In this chapter we discuss the technical concepts and background information associated with this thesis. We first discuss the LLVM framework and why it is suitable for our implementation. We then give an overview of the Region Vectorizer framework which we extend for our prototype. Finally, we discuss the Scalar Evolution [4] analysis and the SCEV representation that we use to model memory groups.

2.1 LLVM

LLVM [6] is an open-source compiler development framework. LLVM-based compilers have a three-phased design. The frontend is responsible for parsing, validating and converting source code to the LLVM intermediate representation (LLVM-IR). The optimization phase contains various analyses and transformations for the LLVM-IR code. Finally, the backend phase is used for optimizing the code for different target platforms and generating assembly code. This modular design makes it easy to support new languages and architectures. To create an LLVM-based compiler for a new language, we only need to build a frontend for it. Similarly, new target platforms can be added to the framework by creating backends for them.

The core of the framework is the LLVM-IR code representation. It is based on the Static Single Assignment (SSA) [7] form that ensures that all variables are defined only once and that the program points of their use are always dominated by their definition. Loops are represented using branches and labels. All non-branching code is grouped in a basic block and is executed unconditionally. Control flow is represented using conditional or unconditional branches at the end of basic blocks.

In our implementation, we vectorize scalar LLVM-IR and generate vectorized LLVM-IR. We rely on LLVM's backend to generate the assembly code for our target platform.

LLVM also features a Scalar Evolution analysis which is important for the memory access grouping phase of our framework.

2.2 Region Vectorizer

The Region Vectorizer (RV) [5] is a vectorization framework for LLVM. RV vectorizes code regions from loop nests to whole functions. It supports both inner and outer loop vectorization. RV performs a divergence analysis [1, 2] on the input scalar code region to figure out the divergent branches in its Control-Flow Graph. This information is then used for code generation. We extend RV's divergence analysis to multiple dimensions to build our tensor shape analysis. We also build upon its code generation phase to generate multi-dimensional vectorized code.

2.3 Scalar Evolution

The Scalar Evolution [4] analysis computes a closed form representation for the evolution of integer variables in a loop nest. It represents this information in the form of a *SCEV*. SCEVs are typed based on the kind of mathematical expressions they represent. The smallest building blocks are *SCEVConstant* for constant values and *SCEVUnknown* for variables with unknown values at compile time. Mathematical expressions like addition or multiplication are represented as *SCEVAddExpr* or *SCEVMulExpr*. These expressions are recursively defined using other SCEV expressions. There is support for arithmetic operations like addition, substraction, multiplication and unsigned division. The analysis also supports the normalization of SCEVs into a canonical form.

Scalar Evolution is commonly used for analyzing loop nests as it detects the iteration variables and provides a *SCEVAddRecExpr* for their change during loop execution. A SCEVAddRecExpr represents an add recurrence of the form

$$\{ < \text{base} >, +, < \text{stride} > \}$$

The *base* value represents the value at the first loop iteration and the *stride* represents the increase in every following loop iteration. Consider the loop nest in Figure 2.1. We see two memory accesses, the first is to a single-dimensional array `A` and the second is to a two-dimensional array `B`. Both arrays are of type `int`.

```
for(int i=1; i<rows; i++) {
  for(int j=0; j<cols; j++) {
    A[i] = i;
    B[i-1][j] = j;
  }
}
```

FIGURE 2.1: Loop nest with memory accesses

The SCEV for the memory access to array `A` is given by:

$$\{\underbrace{(4+\%\text{A})}_{base}, +, \underbrace{4}_{stride_i}\} < loop_i >$$

Here, %A represents the address of the pointer to the base of array `A`. The *stride* is of 4 bytes which is the size of the `int` type. The *base* is shifted by 4 bytes from the beginning of array `A` because loop dimension, i, starts iterating at index `1`. This SCEV tells us that for every iteration of the outer loop, the value of the address of the memory access to array `A` increments by a *stride* of `4` bytes starting with the *base* value of (`4` + %A).

For the two-dimensional array `B`, the SCEV for the memory access is given by:

$$\{\{\%\text{B}, +, \underbrace{(4 * \%cols)}_{stride_i}\} < loop_i >, +, \underbrace{4}_{stride_j}\} < loop_j >$$

There are two add recurrences involved in this expression. The inner add recurrence forms the *base* of the outer add recurrence. The inner add recurrence represents the increase, $(4 * \%cols)$ bytes, in the value of the pointer to array `B`, with every iteration of the outer loop dimension, i. The outer add recurrence represents the increase in the value of the pointer to array `B` for every iteration of the inner loop dimension, j.

The memory access grouper in our framework, uses SCEVs for grouping memory accesses. The constant offsets between addresses of memory accesses are computed using SCEV expressions.

Chapter 3

Multi-Dimensional Vectorization

In this chapter we discuss the motivation behind multi-dimensional vectorization and how it differs from single-dimensional vectorization when applied to stencil applications. We then discuss the existing techniques that perform multi-dimensional vectorization. Finally, we give an overview of the various factors that influence the performance of multi-dimensional vectorization.

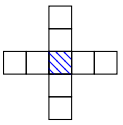
3.1 Overview

Multi-dimensional vectorization extends loop vectorization to multiple dimensions. It involves vectorizing along two or more loops in a loop nest and can improve performance of multi-dimensional stencil applications. In such applications, each element of the resultant matrix depends on elements from multiple dimensions of the input matrix. Since memory in modern computers is single-dimensional, elements in the outer dimensions of the input matrix are always non-contiguous and cannot be accessed using fast vector memory accesses. For larger input matrices, when the size is too large for the elements to fit into the caches, reloading elements from the outer dimensions becomes significantly expensive. Multi-dimensional vectorization reduces data reloads in multiple dimensions, thereby reducing the execution time of such applications.

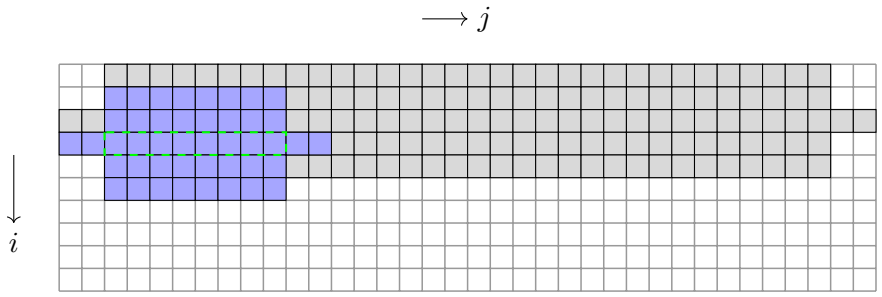
Consider the 9-point 2D Jacobi stencil in Figure 3.1(A). If we vectorize it in a single dimension with a vectorization width of 8, we get the iteration pattern in the input matrix as shown in Figure 3.1(B). If we do it without using shuffle operations to reuse loaded elements shown as in Figure1.1(B), we would need nine vector loads and one vector store to perform each iteration. We see that the elements of the second row

are loaded twice, first when computing the results for the third row and again when computing the results for the fourth row.

```
for(i = 2; i<rows - 2; i++){
  for(j = 2; j<cols - 2; j++) {
    B[i][j] = 0.2 * ( A[i][j]
            + A[i][j-2] + A[i][j-1]
            + A[i][j+2] + A[i][j+1]
            + A[i-2][j] + A[i-1][j]
            + A[i+2][j] + A[i+1][j] );
  }
}
```



(A)



(B)

FIGURE 3.1: Scalar code for 9-point 2D Jacobi stencil (A) and its iteration pattern when vectorized along inner loop with a vectorization width of 8 (B)

Using existing techniques, the above stencil can be vectorized in multiple dimensions by either modifying the code to process multiple rows of the matrix in the same iteration or by modifying the data layout to store elements from multiple rows contiguously. We discuss both approaches in the following sections.

3.1.1 Register Tiling

When using register tiling, we first vectorize along the inner dimensions like for single-dimensional vectorization. Next, we unroll some iterations of the outer loop and jam the resulting inner loops together. For our Jacobi stencil, if we unroll two iterations of the outer dimension we get the iteration pattern as shown in Figure 3.2. This would require 14 vector loads and two vector stores. In single-dimensional vectorization the same computation would have required 18 vector loads and two vector stores. Due to the contiguous data access pattern of the stencil, we can reuse four operand vectors from the data loaded for the first of the two outer loop iterations that we unroll. This reduces the memory accesses for the second unrolled iteration to five vector loads and one store. Also, the elements of the second row are loaded only once in this case, since the results for the third and fourth row are computed in the same iteration. Each

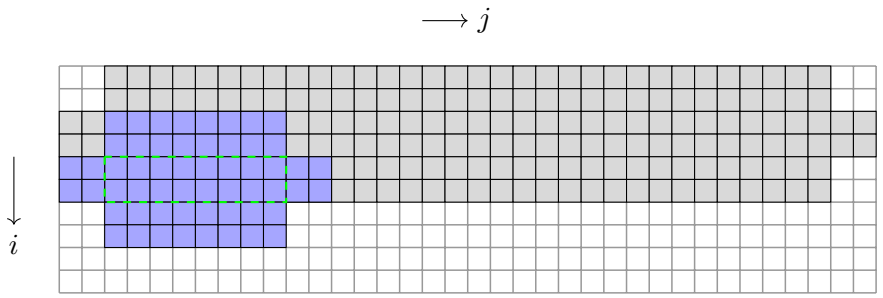


FIGURE 3.2: Iteration pattern for stencil from Figure 3.1(A) when vectorized in multiple dimensions using register tiling.

iteration now computes results for 16 elements of the output matrix as opposed to 8 in single-dimensional vectorization. By unrolling more iterations of the outer loop, we can process even more elements per iteration. However, by doing so, we would also increase the number of vector registers required per iteration. For larger stencils, the increase in register pressure may lead to register spilling. This technique is also not well suited for stencils that access data non-contiguously or with larger strides in each dimension as this would reduce the amount of reuse that we get when unrolling multiple outer loop iterations.

3.1.2 Vector Folding

Vector folding [8] involves storing blocks of elements from multiple dimensions contiguously as opposed to the standard single-dimensional data layout. The layout of these blocks or *vector folds* is determined by the multi-dimensional vectorization layout. The only requirement is that the size of each block must be equal to the vectorization width available on the hardware platform. For a 2D matrix, a vector fold of 2×4 would require a data layout transformation as shown in Figure 3.3.

Each vector block can be loaded by a single vector load operation. Operand vectors that contain elements from neighboring vector blocks are called unaligned operand vectors. For such memory accesses, we need to load both neighboring blocks and perform shuffle or permute operations to build operand vectors. The shuffle operations are faster than vector loads but require more vector registers.

For our example from Figure 3.1(A), vectorizing using vector folding would give us the iteration pattern as shown in Figure 3.4. We would need six vector loads, two stores per iteration with seven permute operations for the unaligned operand vectors and two for the unaligned result vector. Due to the 2×4 layout, the elements in the two rows on top and bottom of the input elements in every iteration are unaligned and we would

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	...
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	...
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	...
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	...

(A)

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	0,4	0,5	0,6	0,7	1,4	1,5	1,6	1,7	...

(B)

FIGURE 3.3: 2D input matrix with vector folds of size 2×4 (A), and its corresponding single-dimensional memory representation (B)

have to load two neighboring vector folds for each of them. We process eight elements per iteration as in the single-dimensional case but we use comparatively fewer memory operations for the same. Also, the second row would not be reloaded in this technique. Yount[8] has shown in his experiments that vector folding outperforms single-dimensional vectorization for 3D stencils when optimizing for instructions per cycle.

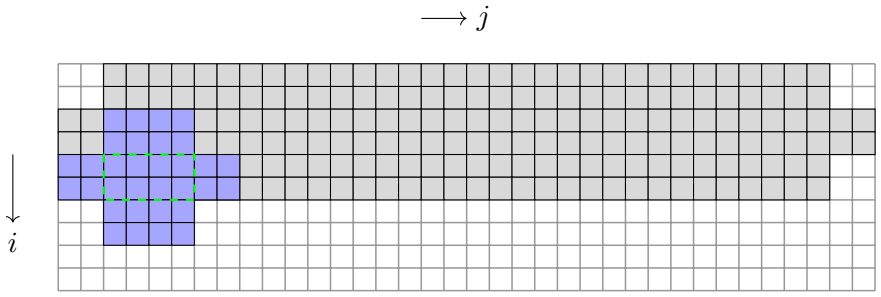


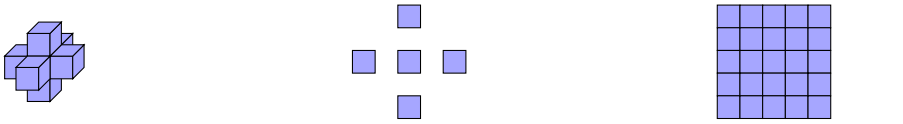
FIGURE 3.4: Iteration pattern for stencil from Figure 3.1(A) when vectorized in multiple dimensions using vector folding

However, this technique requires transforming the data layout before and after the computation. Hence, it is not preferable when we do not perform multiple time-step iterations over the input matrix. Also, similar to register tiling, it is not efficient for stencils that have non-contiguous and strided access patterns.

The techniques discussed above require computing an efficient vector fold or a vectorization layout that would lead to faster overall execution. This depends on the input stencil pattern and the hardware platform used. In the following sections we discuss the influence of these two factors.

3.2 Influence of the Stencil Pattern

Stencil code can be described by its properties like number of dimensions, order, number of points and data access pattern. These properties play an important role in determining an efficient vectorization layout. We discuss these properties and their influence on the vectorization layout in the following.



(A) 3D, 2nd-order, 7-point Jacobi stencil

(B) 2D, 4th-order, 5-point stencil

(C) 2D, 4th-order, 25-point Seidel stencil

FIGURE 3.5: Stencils with different dimensions, orders, data access patterns and number of points.

- **Dimensions:** The dimensions of a stencil are needed to select the dimensions to vectorize. Similar to tiling based approaches, the reordering of computations in multi-dimensional vectorization leads to faster execution than single-dimensional vectorization when there are data dependencies in multiple dimensions. If there are no dependencies in a given dimension of the input array, vectorizing along it cannot get us more reuse of loaded data.
- **Order:** The order of a stencil gives us an idea of the distance up to which we can exploit the reuse in each dimension. In Figure 3.5(C), we see that in both dimensions of the stencil, we have elements up to a distance of two from the central element. By vectorizing using a 2×4 layout, we can exploit reuse of two elements in the outer loop while benefiting from contiguous loads of four elements in the inner loop.
- **Data access pattern:** The data access pattern indicates how the points in a stencil are spread out in the input matrix. In Figure 3.5, we have two stencils with contiguous data access (Figure 3.5(A) and Figure 3.5(C)) and one stencil (Figure 3.5(B)) with a strided data access with a stride length of 2. For the stencil in Figure 3.5(B), vectorizing with a layout of 2×4 would not get us any reuse in the outer dimension because the stride length is 2. We need to vectorize further in that dimension to get an overlap of accesses between consecutive iterations.
- **Number of points:** The total number of points in a stencil is useful for estimating the number of vector registers that would be needed for each iteration. For stencils with more points, it is better to avoid techniques that require complex shuffle operations as this would further increase register pressure.

The dimensions of the stencil are also important for data layout transformations. For single-dimensional stencils, the default single-dimensional data layout is efficient while for stencils in two or more dimensions, we would need a data layout transformation to speed up memory operations.

3.3 Influence of the Hardware Platform

We need to consider the number of available vector registers, their width and available SIMD instructions in our target hardware platform. The width of vector registers limits the number of choices we have for vectorization. For example, with a width of 8 elements we can have layouts such as: 1×8, 2×4, $2 \times 2 \times 2$, 4×2, 8×1. A vectorization width of 4 elements gives us a multi-dimensional layout of 2×2 only. The number of vector registers helps to account for register pressure. In systems with more vector registers, we can use larger vectorization layouts. Conversely, if we do not have extra registers beyond those required for the stencil operands, we need to limit the number of shuffle operations or reduce the dimensions of our vectorization layout. In multicore execution, large vectorization layouts may lead to more misses in the shared cache. In systems that do not support efficient shuffling of values between registers, it might be more efficient to use the register-tiling-based approach that requires loading values directly from memory as opposed to shuffling.

In the next chapter we discuss the fundamental concepts of our framework and the analyses that it performs for efficient code generation. Our framework tries to combine ideas from the existing approaches to multi-dimensional vectorization. We mainly focus on reducing memory accesses and replacing multiple scalar memory operations by a single vector operation where possible. The different factors influencing the stencil computation help us to understand the results of our experiments later in the Evaluation chapter.

Chapter 4

Our Framework

In this chapter we discuss our approach to multi-dimensional vectorization and the various concepts associated with our framework. We first discuss our representation for a multi-dimensional vectorization layout followed by the analysis phases of our framework: vectorization analysis and memory access grouping. We also take a look at data layout transformations and how they affect the performance of stencil applications.

4.1 Vectorization Layout

In single-dimensional vectorization, the vectorization width often referred to as vectorization factor determines the number of elements that are processed simultaneously. For stencil codes without loop carried dependencies, this is often the maximum number of elements of the input datatype that can be fit into a vector register of the system. For multi-dimensional vectorization, we use the *Tensor Brush* [3] to represent the vectorization layout.

For a loop nest with d loops, the tensor brush is defined as,

$$\mathcal{B} = (m_0 \times \cdots \times m_{d-1})$$

Here, $m_i \in \mathbb{N}$ is the size of the brush in the i^{th} dimension of the loop nest. The dimensions are indexed starting from the outermost loop with index 0, moving inwards to the innermost loop that has the index $(d-1)$. The product of brush sizes from all dimensions gives us the effective vectorization width.

Each loop in a loop nest has a corresponding brush size. For loops that are not being vectorized, we set the brush size to 1. In a loop nest with three loops, if we vectorize along the inner two loops with a vectorization layout of 2×8, our tensor brush would be $1 \times 2 \times 8$.

4.2 Brush Projections

While the tensor brush is useful for representing multi-dimensional operand vectors and how they relate to loop nests, for code generation, we need to compute a mapping between our multi-dimensional tensor brush and single-dimensional hardware vector registers. We define a brush projection ($\mathcal{P}_{\mathcal{B}}$) as a unique mapping from tensor brush coordinates to single-dimensional vector lanes in a vector register. The projection is defined by a projection vector $\mathcal{P}_{(i_0,..,i_{d-1})}$ where $(i_0, ..., i_{d-1})$ are the dimension numbers in the tensor brush. Depending on the ordering of these dimension numbers, we can choose the order of expansion of the tensor.

$$\mathcal{P}_{(i_0,i_1,..,i_{d-1})}(c_0,..,c_{d-1}) = \left\{ \begin{array}{cc} c_{i_0} & \text{if } d = 1 \\ c_{i_0} + m_{i_0}\mathcal{P}_{(i_1,..,i_{d-1})}(c_0,..,c_{d-1}) & \text{if } d > 1 \end{array} \right\} \tag{4.1}$$

Equation 4.1 defines a projection from coordinates $(c_0, .., c_{d-1})$ to vector lanes where the brush is given by $\mathcal{B} = (m_0 \times \cdots \times m_{d-1})$ and $\mathcal{P}_{(i_0,i_1,..,i_{d-1})}$ is the projection vector.

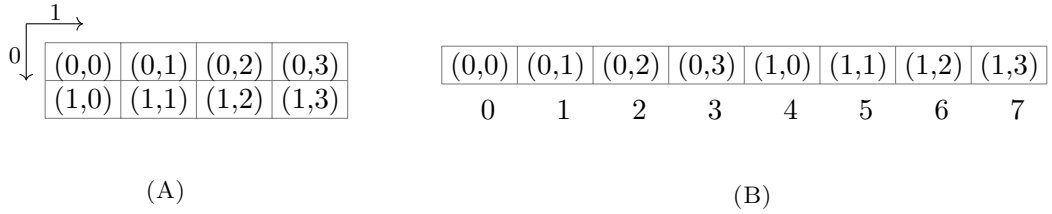


FIGURE 4.1: Brush layout for a vectorization brush of 2×4 (A) and its corresponding vector lane mapping with $\mathcal{P}_{(1,0)}$ (B)

For example, in Figure 4.1(A), we see a tensor brush of 2×4 with the coordinates of its elements. For a row-major single-dimensional layout we use a projection brush of $\mathcal{P}_{(1,0)}$ to get the projection as shown in Figure 4.1(B).

4.3 Outline of Our Framework

Our framework aims at reducing memory traffic to improve the efficiency of the generated multi-dimensional vector code. To this end, we group contiguous memory accesses together so that they can be replaced by vector memory accesses. We use analyses to compute the data access pattern in every vectorized iteration of the input loop nest.

We do not limit ourselves to vectorization layouts that are at least as wide as the size of vector registers in the innermost dimension like register tiling. If the vectorization

layout has a width in the innermost dimension equal to or larger than the size of the vector registers, we get a layout similar to the register-tiling-based approach. On the other hand, if the width of the innermost dimension in our vectorization layout is smaller than the width of the vector registers, we perform shuffle operations to pack elements from multiple vector memory accesses into a single vector operand register. For certain stencils, the default single-dimensional data layout is not the most efficient one. Our framework supports transforming the memory addresses from scalar code into the transformed data layout. However, we do not use data layout transformation as a means of performing multi-dimensional vectorization like vector folding [8].

For a given stencil application code and a vectorization layout, our framework generates efficient multi-dimensional vector code in three phases: First, we analyze the vector shape [9] along each dimension of the loop nest and assign vector shapes to each instruction. Next, for each assignment statement, we group all required memory accesses into contiguous groups. If the memory data layout is not the default single-dimensional layout, we compute the corresponding transformed memory addresses and store them into the respective memory groups. Finally, we generate vector code for all instructions in the loop body. In the following sections we discuss the first two phases and associated concepts. Code generation is discussed in the next chapter with the implementation details of our prototype *TensorRV*.

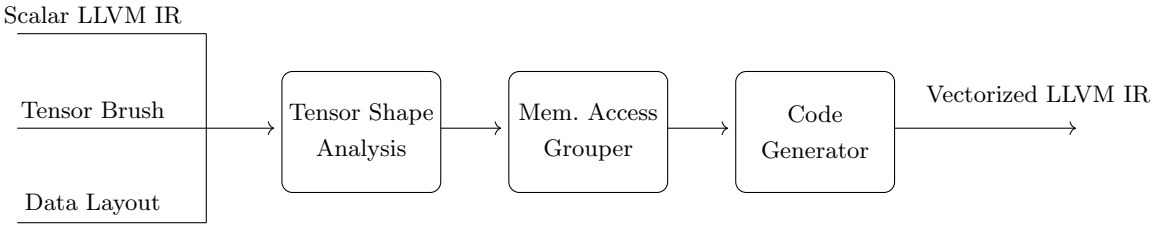


FIGURE 4.2: An outline of our framework

4.4 Tensor Shape Analysis

The tensor shape analysis as presented in our previous work [3] determines the relationship between instructions in the loop body and the iteration variables of its surrounding loops. For each loop dimension in a loop nest, the analysis computes how the values of instructions that involve its iteration variable evolve with each iteration of the particular

loop. For a single loop dimension, the analysis stores a vector shape of type,

$$\mathcal{T}_{\texttt{1D}} = \mathbb{Z} \cup \{\top\}$$

The set of d-dimensional tensor shapes $\mathcal{T}^d$, can be defined as the composition of d one dimensional vector shapes:

$$\mathcal{T}^d = (\mathcal{T}_{\texttt{1D}})^d$$

The vector shape, $s_i \in \mathcal{T}_{\texttt{1D}}$, of an instruction for a particular loop dimension, i, is a non-zero integer if its value increments by that integer value in every iteration of that loop dimension. The instruction is considered to be *strided* with a stride length of s_i along the i^{th} loop dimension. If $s_i = 0$, it indicates that the instruction is invariant to the change in the value of the iteration variable, it is *uniform* along the i^{th} loop dimension. If s_i is $\top$, the instruction is considered to be *varying* in the i^{th} dimension but the variation is unknown. An example of such an instruction is an access to some array location with the iteration variable as the array index. The variation in this case depends on the contents of the array at that index, which are usually unknown at compile time.

The iteration variables themselves are invariant in dimensions other than their own loop dimension. The vector shapes in each loop dimension constituting the tensor shape $\mathcal{T}^d$ are similar to partial derivatives of a function. The vector shapes together give us the change in instruction value based on all surrounding loops just as the partial derivatives of a multivariate function together give us its derivative.

For each iteration variable, the vector shape in its own dimension is given by the increment value of the loop. This initial value is then propagated to all instructions in the loop nests for computing their shapes. Affine combinations of iteration variables in instructions are represented by strides in the tensor shape. The analysis assumes that there are no loop-carried dependencies between the instructions.

Consider the loop nest in Figure 4.3(A) and its corresponding tensor shape values in Figure 4.3(B). The computed value of `a` is propagated to the statement computing the value of `b` which in turn is propagated to the access of array `C`.

Tensor shapes tell us how an instruction's value changes across iterations of the scalar loop nest. In stencil codes we also have to consider the data access pattern within the same iteration of the loop when grouping memory accesses. This is implicitly handled by the memory access grouper by computing relative offsets of all accessed elements of an input matrix from its base address.

```
for (i = 0; i < m; ++i) {// dim 0
  for (j = 0; j < n; j += 2) {// dim 1
    for (k = 0; k < w; k += 4) {// dim 2
      2 * i - 3 * j;
      a = j * n + k;
      b = A[a];
      C[b];
    }
  }
}
```

(A) 3D loop nest.

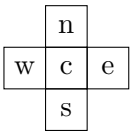
Statement	Tensor shape $\mathcal{T}^d$
`i`	$(1, 0, 0)$
`j`	$(0, 2, 0)$
`k`	$(0, 0, 4)$
`2 * i - 3 * j`	$(2, -6, 0)$
`a = j * n + k`	$(0, \top, 4)$
`b = A[a]`	$(0, \top, \top)$
`C[b]`	$(0, \top, \top)$

(B) Initial tensor shapes (upper half) and, Tensor shapes computed by the tensor shape analysis (lower half) .

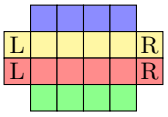
FIGURE 4.3: A 3D loop nest (A) and its corresponding tensor shapes (B).

4.5 Memory Access Grouping

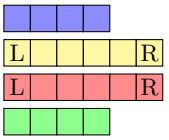
In stencil applications, expanding scalar memory accesses to their tensor layout, as per the tensor brush, gives us overlapping elements between neighboring operand vectors. Our framework groups the addresses of these elements into contiguous *memory groups* that can be accessed using fast vector memory instructions in the code generation phase.



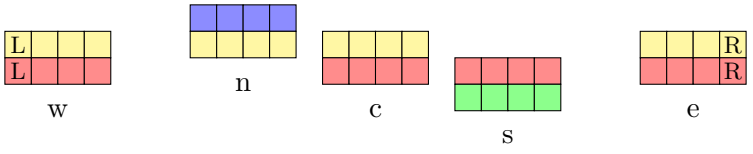
(A) 2D, 2nd-order, 5-point Jacobi stencil



(B) Vectorized using 2×4 layout.



(C) Computed Memory Groups



(D) Expanded brush for each operand from stencil in (A).

FIGURE 4.4: Overlapping accesses (D) in vectorized brush layout (B) of 5-point Jacobi stencil (A) being put into memory groups (C).

Consider the 5-point Jacobi stencil in Figure 4.4(A) and its expanded brush layout in Figure 4.4(B). In Figure 4.4(D) we show the individual expanded operand vectors for each point of the stencil. For the points `n` and `c` we see that the top four elements in the vector expansion of `c` are the same as the bottom four in the expansion of `n`. This is referred to as an overlap between the brush expansions of points `n` and `c`. Between the points `w` and `c` or `e` and `c` we get a larger overlap of six elements. These overlapping

elements are shuffled to form the different operand vectors after they are loaded into memory.

The memory access grouper fits elements whose addresses differ by a constant less than a pre-determined group size (s_g) into a single group. For our example, we get four memory groups as shown in Figure 4.4(C). Each vector memory access can process elements up to the width of vector registers. Hence, one would expect the group size to be equal to the width of a vector register (s_{vr}). However, this can lead to ambiguity when adding elements to memory groups. For elements that are at a distance greater than or equal to s_{vr} from the first elements of existing memory groups, we create a new group. For elements that are at a distance less than s_{vr}, we add them to the matching group. When the group size is equal to s_{vr}, we may end up with situations where an element can be added to more than one group. To avoid this ambiguity, we set the group size to twice the vector register size.

If there are gaps in the group or if the group has more elements than the vector register size, in the code generation phase, the group is broken down into chunks that can be loaded using vector memory instructions. For transformed data layouts, we add elements to groups after converting their addresses to the new layout.

4.6 Data Layout Transformation

The goal of data layout transformation is to reduce the number of shuffle operations and memory loads per iteration. We transform the layout of our matrices by storing elements from multiple dimensions contiguously as discussed in Section 3.1.2. Since this process requires shuffles to compute non-aligned operand vectors, it is unsuitable for systems that do not support efficient shuffle operations. For computing an efficient data layout, we need to consider both the stencil pattern and the vectorization layout.

In contrast to vector folding where the size of the vector folds must be equal to the width of vector registers, our framework supports arbitrary data layouts. We also do not require the data layout to be the same as the vectorization layout. The framework itself does not compute an efficient data layout or perform data layout transformation on the operand matrices. If the input data is stored in a different data layout than the default on the target platform, the framework transforms the addresses of the vectorized memory accesses into the new layout before adding them into the memory access grouper.

We represent data layouts using the *Data Brush*. It is similar to the *Tensor Brush* defined in Section 4.1. For example, in the 2D Jacobi example from Section 3.1.2, we used a data brush of 2×4 with a vectorization layout of the same size. We needed six

vector loads, two stores per iteration with seven shuffle operations for unaligned operand vectors and two for the unaligned result vector. If we use a data brush of 2×2 with a vectorization layout of 2×4 for the same stencil, we can load all the required elements in four vector loads and the required number of shuffles is also reduced to five. This is due to the fact that in a 2×2 layout, the two top and bottom rows are no longer unaligned and can be loaded directly from memory. The result vector can also be written to memory without shuffles as it is also not unaligned.

In general, for multi-dimensional vectorization, increasing the size of the memory blocks or vector folds would reduce the number of memory accesses needed per iteration when compared to the default single-dimensional memory layout. However, if the size of the data layout in a given dimension is larger than the order of the stencil or the size of the tensor brush in that dimension, we have to load unaligned memory blocks, which increases the required number of shuffles. For larger memory blocks, constructing operand vectors may need multiple shuffles per operand vector due to alignment issues.

In the next chapter we discuss the implementation of our prototype using the LLVM framework. The information collected by the tensor shape analysis described in this chapter is used for driving code generation of each instruction in the loop body. Depending on the shape of the instruction, we decide on which instructions to emit for it. The memory groups are the core of our framework's optimization technique. We reduce memory traffic by loading each memory group only once using fast vector instructions. We discuss the implementation of the memory access grouper and describe the code generation phase of our framework.

Chapter 5

TensorRV

In the last chapter we talked about our framework for multi-dimensional vectorization and the fundamental concepts associated with its different phases. In this chapter we describe the implementation of our framework in LLVM as a fork of the Region Vectorizer (RV) project. We discuss the design of the memory access grouper using Scalar Evolution and the techniques used to transform memory addresses from one memory data layout to another. We end the chapter with a discussion of the code generation phase of our framework.

5.1 Implementation of Tensor Shape Analysis

Our prototype, *TensorRV*, extends RV's analyses and code generation to support multi-dimensional vectorization. The prototype takes scalar LLVM IR code along with a vectorization layout and a memory data layout as input and emits vectorized LLVM IR that can be compiled to different target platforms by LLVM's backend phase.

We extend RV's divergence analysis to multiple loop dimensions in order to implement the Tensor Shape Analysis from Section 4.4. The analysis begins by initializing shapes for all the loop iteration variables from their increments. We then process each loop dimension one by one using a worklist based algorithm to process the instructions in the loop body. For each loop dimension, we initialize the worklist with all instructions in the loop body whose value depends on its iteration variable. For each instruction in the worklist, we compute its shape in that loop dimension and add all instructions that use its value to the worklist. The algorithm terminates when the worklist is empty.

In the following sections, we discuss the memory access grouper followed by the code generator of our framework.

5.2 Memory Access Grouper

The memory access grouper groups addresses of memory accesses that are required for each assignment statement. For simplicity, we limit our loop bodies to a single assignment statement in our prototype. Hence, in our implementation, it groups addresses of memory accesses in each iteration of our vectorized code.

The memory access grouper maintains two data structures. The first is the set of memory groups. Each group contains memory addresses that can be loaded contiguously. The second is a mapping between the SCEV of each memory address required in the vector code and its corresponding vector lane in the brush projection of the tensor brush. Both data structures depend on the computation of SCEVs as discussed below.

```
for (int j=1; j<rows-1; ++j) {
  for (int i=1; i<cols-1; ++i) {
    B(j,i) =                  + .2 * A(j-1, i) +
             .2 * A(  j,i-1) + .2 * A(  j, i) + .2 * A(  j,i+1) +
                              + .2 * A(j+1, i);
  }
}
```

FIGURE 5.1: Loop nest for 5-point, 2D Jacobi stencil

Consider the loop nest in Figure 5.1. It computes an iteration of a 5-point, 2D Jacobi stencil on matrices of type `double`. The SCEV of the central element in the stencil, `A(j,i)` is given by,

$$\{\{(8 + (8 \ * \ \%cols) + \%A), \ + , (8 \ * \ \%cols)\} < loop_j >, \ + , 8\} < loop_i >$$

We can deduce the stride in each dimension of the input matrix from the stride value for each loop dimension. For the inner dimension, i, it is 8 bytes and for the outer dimension, j, it is $(8 \ * \ \%cols)$ bytes.

Now, if we consider the SCEV expression in the base part of the inner add recurrence,

$$(8 \ + \ (8 \ * \ \%cols) \ + \ \%A)$$

we see that the stride lengths in both dimensions are added to the base address of array `A`. This is done because the first iteration of the loops starts at index `1`. Similarly, the SCEV for `A(j,i+1)` is given by,

$$\{\{(16 + (8 \ * \ \%cols) + \%A), \ + , (8 \ * \ \%cols)\} < loop_j >, \ + , 8\} < loop_i >$$

In this case, the SCEV expression in the base part of the inner add recurrence has increased by 8 bytes. Hence, we can generate SCEVs for addresses in the neighborhood of an element of array `A` by using the strides and the SCEV for the element.

Using the coordinates from the tensor brush, and the SCEV for each memory access in the scalar code, the memory grouper generates SCEVs, similar to the ones we saw earlier, for all elements required for the vectorized code. These SCEVs are used for building memory groups and are also stored in the mapping between SCEVs and vector lanes.

When adding a new address to the set of memory groups, we compute the difference between the new address and the first elements in the existing groups. If the difference is less than the group size for a particular group, the element gets added to it. Otherwise, we create a new group for it.

5.2.1 Addressing Transformed Data Layouts

During data layout transformation, the input matrix is divided into multiple blocks or vector folds which are stored contiguously. In Figure 5.2(A) we have a 2D matrix that is transformed using a data layout of 2×2. We represent the transformed matrix in 2D as shown in Figure 5.2(B).

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7

(A)

0,0	0,1	1,0	1,1	0,2	0,3	1,2	1,3	0,4	0,5	1,4	1,5	0,6	0,7	1,6	1,7
2,0	2,1	3,0	3,1	2,2	2,3	3,2	3,3	2,4	2,5	3,4	3,5	2,6	2,7	3,6	3,7

(B)

FIGURE 5.2: 2D matrix (A) and its 2×2 data layout transformed version (B)

In the transformed matrix, we divide the number of rows by two and multiply the number of columns by two since elements from two rows are stored in one row. The addressing works by first locating the block in the transformed matrix and then pointing to the element in the block. This simplifies the computation of the transformed address as explained below.

We compute the transformed address of an element by first computing the index of the block to which it belongs and then adding the offset of the element inside the block. The block index $(b_0, ..., b_{d-1})$ of an element having indices $(ind_0, ..., ind_{d-1})$ in a d-dimensional matrix having blocks of size $(s_{b_0}, ..., s_{b_{d-1}})$ is computed by,

$$\forall\ b_i \in (b_0, ..., b_{d-1}), \quad b_i = \frac{ind_i}{s_{b_i}}$$

The offset of the element inside the block is computed using brush projections as defined in Section 4.2. For example, for the element $(3, 3)$ in the matrix in Figure 5.2, the block offset is $(1, 1)$ and the offset of the element inside the block is 3. We use a projection brush of $\mathcal{P}_{(1,0)}$ for the 2×2 data brush.

The bounds of the iteration variables of our loop nest are modified for the transformed matrix as per the new representation. Hence, for dimensions other than the innermost one, the loop bound is divided by the block size in that dimension. The strides of the iteration variables are also adjusted to step between blocks. In the earlier example, this implies that the stride of the inner loop is adjusted to four times the size of one element. When generating the SCEV for the transformed address, we first generate the SCEV for the element's block. The offset inside the block is added to the generated SCEV address for the block.

The transformed address can be computed without changing the sizes of the dimensions of the original matrix by distributing the blocks in each row of the transformed matrix into two rows. However, this requires the size of the innermost dimension in the matrix for the computation. Since we do not have a constant value for it at compile time, Scalar Evolution generates a *SCEVUDivExpr* for it. Such expressions often give non-constant differences when adding them into memory groups. Since our memory grouper only works with constant differences we cannot use these SCEVs for our requirements.

5.3 Code Generation

The code generation phase uses tensor shapes and memory groups from the earlier phases to generate vector code. Before generating code for the instructions in the loop body, the loop increments and loop bounds for each loop dimension are adjusted based on the tensor brush size in that dimension. We assume that all loop bounds are such that the loop can be fully vectorized using the chosen vectorization layout and that there are no remainder iterations left over after vectorization.

For the instructions in the loop body, the code generator visits each instruction in a reverse post-order and generates code for them. We assume that all instructions other than memory accesses have no side-effects. For the next parts of this phase we distinguish between memory access instructions and all other instructions. We first discuss code generation for side-effects-free instructions and then memory access instructions.

5.3.1 Side-Effect-Free Instructions

The tensor shape plays an important role in the vectorization of these instructions. Depending on the tensor shape, such instructions are handled as follows:

- **All strided or strided and uniform:** For strided instructions that have no varying dimensions, we emit a scalar instruction to compute the value for the first vector lane. When the instruction is being used by a vector instruction and a full expansion is required, we first generate a stride vector of the same dimensions as the tensor brush. The values of this vector are the increments for each coordinate based on the stride value for that dimension in the tensor shape. We then generate a vector with all its lanes having the computed scalar value. Finally, the stride vector is added to this generated vector having the scalar value of the instruction in all its lanes. If the stride is zero in all dimensions, we can skip generating and adding the stride vector.
- **Varying:** If there are any varying dimensions in the tensor shape, we change the data type of the instruction to a vector data type. The vector data type is essentially a vector of the scalar type whose size is equal to the size of the projected tensor brush. For example, a `float` scalar type with a tensor brush of 2×4 would give us a vector of type `<8 x float>`. The strided or uniform operands of this instruction would also be expanded to vectors of the same size based on the brush projection vector.

Our framework allows for tensor brushes that when projected, form vector operands that are larger than the hardware vector registers. For example, a tensor brush of 2×16 is projected to a single-dimensional vector of 32 elements. If the input data type is `double`, on a AVX-512 system that supports up to eight elements of type `double`, our tensor brush is four times as wide as a vector register. In such cases we rely on LLVM's legalization phase for register tiling. A single vector instruction that computes a 32 elements wide vector, would be broken into four vector instructions, each computing on vectors that are eight elements wide.

5.3.2 Memory Access Instructions

The code for memory access instructions is also generated based on their tensor shape. For instructions having all uniform dimensions, we emit a scalar memory instruction. If the instruction is a load instruction and an expansion is needed, the code generator copies the loaded value to all vector lanes. For instructions having varying dimensions, we generate gather or scatter instructions based on whether they are load or store operations. For instructions having strided dimensions, the code generator uses the memory groups from the memory access grouper to reduce memory operations as explained in the following.

The code generator breaks down the memory groups into chunks that can be loaded contiguously. The memory groups from the memory access grouper can be longer than the width of vector registers and may have gaps. During the chunking phase, the memory groups that are contiguous but wider than the vector registers are broken down into parts that are within the vector register width. For memory groups that have gaps, the chunking phase creates chunks of contiguous accesses.

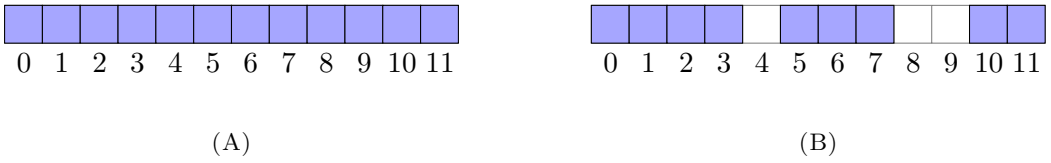


FIGURE 5.3: Memory groups: contiguous (A), with gaps (B)

In Figure 5.3, we see two memory groups – the first having 12 contiguous elements (Figure 5.3(A)) and the second having gaps (Figure 5.3(B)). If we assume the vector registers to be eight elements wide, the chunking phase would divide the first group into two chunks: $[0, 7]$ and $[8, 11]$. For the second group, the chunking algorithm computes three chunks: $[0, 3]$, $[5, 7]$ and $[10, 11]$.

When generating code for a strided memory instruction, the code generator fetches the SCEVs to vector lane mapping for the scalar memory address from the memory access grouper. Depending on whether the instruction is a load or store operation, the code generator proceeds as follows:

- **Load instruction:** For each vector lane, the code generator finds the chunk to be loaded using the corresponding SCEVs. If the chunk has not been loaded yet, it emits a vector load instruction for the chunk. Otherwise, it uses the existing load instruction to generate a shuffle instruction for moving the element from the register of the loaded chunk to the required operand register. Once all the elements

have been mapped, the shuffle generator emits the required shuffle instructions for the operand vector.

- **Store instruction:** For store instructions, the code generator generates shuffle instructions to transfer elements from the result registers to the registers used for the store operation. After the shuffle operation, it emits vector store instructions to complete the operation.

In contrast to side-effects-free instructions, we ensure that the width of generated memory instructions are within the limits of vector register size using the chunking phase.

In this chapter we discussed the memory access grouper and the code generation phase of our framework. The SCEV-based memory access grouping enables us to reduce memory operations in every iteration of the vectorized code. In the next chapter we evaluate our framework experimentally on different 2D stencils.

Chapter 6

Evaluation

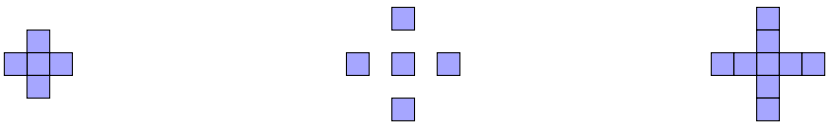
In this chapter, we evaluate our framework experimentally on some commonly used stencils from scientific applications. We compare the performance of different multi-dimensional vectorization layouts to that of single-dimensional vectorization for different data layouts.

6.1 Experimental Setup

We performed our experiments on an Intel i9-7900x CPU that supports AVX-512 vector instructions. The CPU had 10 physical cores and three layers of cache memory (L1 : 640 KiB, L2 : 10 MiB, L3 : 13.75 MiB) with the last layer being shared between the cores. For multi-core experiments, we deactivated hyperthreading and used OpenMP for parallelization.

Our benchmarks are described as follows:

1. Jacobi: This is a cross shaped stencil that averages over the Von Neumann neighborhood of an element. In Figure 6.1, we show the different versions of this stencil that were used in our experiments. We added a 2D, 4th-order, 5-point version (5-point, Sp-Jacobi) of this stencil to test our prototype on non-contiguous memory access patterns.



(A) 2D, 2nd-order, 5-point

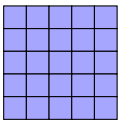
(B) 2D, 4th-order, 5-point

(C) 2D, 4th-order, 9-point

Figure 6.1: 2D and 3D Jacobi stencils that were used in our experiments

2. Seidel: Unlike the Jacobi, this stencil averages over the Moore neighborhood of an element. Figure 6.2 shows the different variations of the Seidel stencil used in our experiments.

(A) 2D, 2nd-order, 9-point



(B) 2D, 4th-order, 25-point

FIGURE 6.2: 2D and 3D Seidel stencils that were used in our experiments

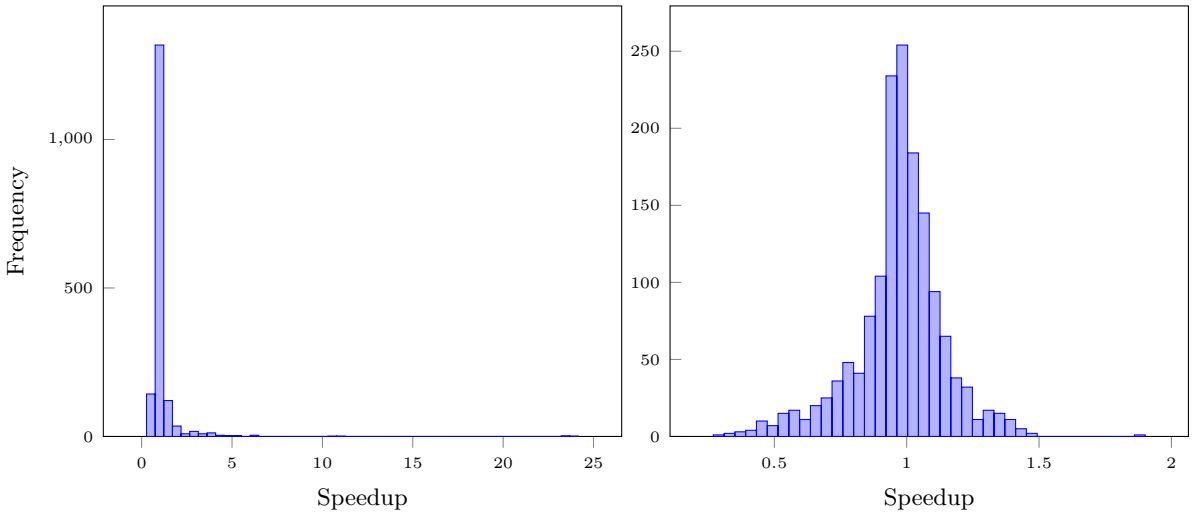
3. Matrix Transpose: The matrix transpose operation is a simple copy operation with a permutation of the indices of the elements. It is different from stencil benchmarks as it does not involve any computations.

We experimented with nine tensor brushes spread over vectorization widths of 8, 16 and 32. We used the default single-dimensional data layout and a data brush of 2×2 for the stencils. We used only the default single-dimensional data layout for matrix transpose as the operation itself is a layout transformation. Our input sizes for the single-core experiments ranged from 128 KiB to 18 MiB and for multi-core experiments, the range was between 2 MiB to 32 MiB.

6.2 Results

We measured the execution times of single-dimensional and multi-dimensional vectorized code for all our benchmarks. We collected about 1683 scores over all our parameter configurations and inputs for multi-dimensional vectorization. Each data point was recorded by taking the median of 51 runs of the experiment. We computed speedup scores for each experiment by taking the ratio between the execution times of single-dimensional and multi-dimensional vectorized code.

In Figure 6.3(A) we show the frequency distribution of the speedups over single-dimensional vectorization for all benchmarks. Most values lie within 0.0 to 2.0 with the rest being distributed between 2.0 to 24.0. We show the distribution without the matrix transpose experiment in Figure 6.3(B). We see that the performance of the stencils is distributed more between 0.3 to 1.0 than 1.0 and 2.0. This indicates that overall we have more slowdowns than speedups in the stencil experiments. We also note that the speedup values higher than 2.0 are from the matrix transpose experiments.

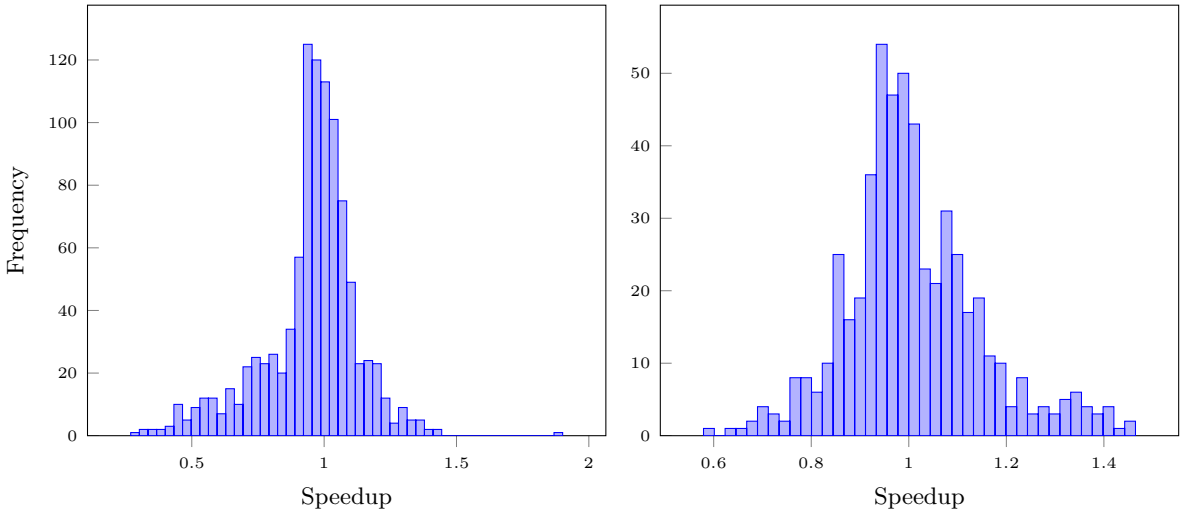


(A) Including Matrix Transpose (B) Excluding Matrix Transpose

FIGURE 6.3: Frequency distribution of speedups over single-dimensional vectorized code

We further divide our stencil experiments into two parts: single-core and multi-core experiments. In Figure 6.4, we see the frequency distribution for each of them.

We see that the single-core values are spread wider than the multi-core ones. Also, the distribution is more evenly distributed around 1.0 in multi-core than single-core. There are more slowdowns than speedups in the single-core experiments.



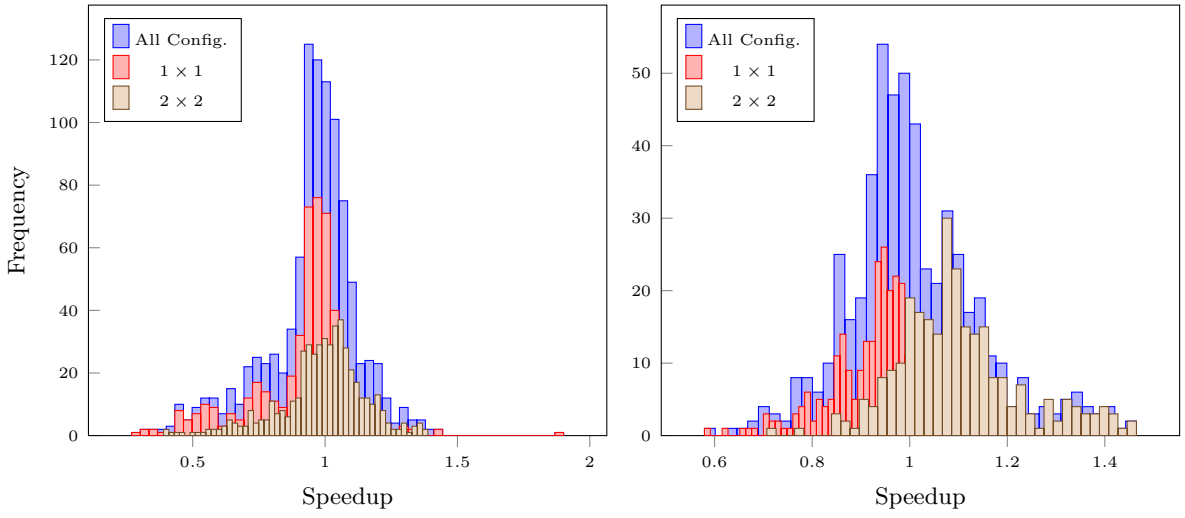
(A) Single-core Experiments (B) Multi-core Experiments

FIGURE 6.4: Frequency distribution of speedups over single-dimensional vectorized code for single-core and multi-core experiments on 2D stencils

In the following sections we discuss the effects of the different parameters that we used in our configurations.

6.2.1 Data Layout Transformations

We examine the influence of data layout transformations for single-core and multi-core experiments separately. We compare the frequency distribution of the individual data layouts with the distribution of all single-core and multi-core configurations. We show the frequency distribution for the different data layouts along with the frequency distribution for single-core and multi-core experiments in Figure 6.5(A) and 6.5(B).



(A) Single-core Experiments

(B) Multi-core Experiments

FIGURE 6.5: Frequency distribution of speedups over single-dimensional vectorized code for different data layouts

In the multi-core experiments graph, we see that the 2×2 data layout's distribution is larger in the zone between 1.0 and 1.5 whereas the 1×1 layout's distributions is more between 0.6 and 1.0. This indicates that for multi-core experiments, a data layout of 2×2 performs better than a data layout of 1×1.

In the case of single-core experiments, we notice that the 2×2 data layout's distribution is concentrated around 1.0 with a longer spread towards 0.3 than 1.5. This indicates that the performance is mostly comparable to that of single-dimensional vectorization but the slowdowns can go below 0.5 while the speedups are less than 1.5 times. The 1×1 layout has more values in the range between 0.5 to 1.0. This indicates that the performance is worse than single-dimensional vectorization in most single-core experiments.

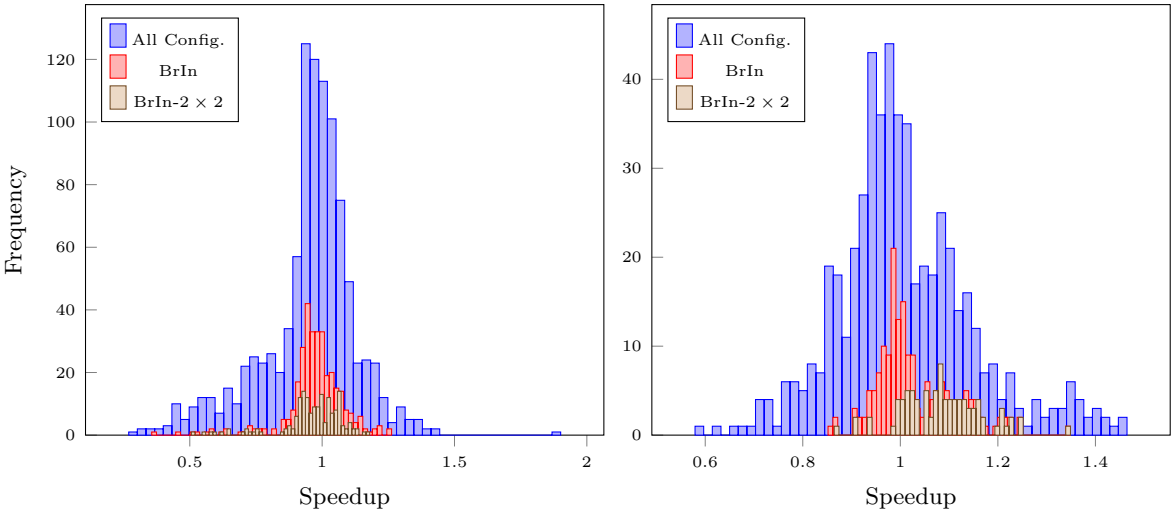
6.2.2 Tensor Brush

We split our tensor brushes into three parts based on how their sizes were distributed between the two loop dimensions. For each set, we plot the distribution for single-core and multi-core experiments and overlay them with the performance distribution for the

respective set. We also add a third layer for the configurations that include these tensor brush subsets with a data layout of 2×2.

- **Brush Inner (BrIn):** This set consists of brushes that have larger sizes in the innermost dimension: 2×4, 2×8, 2×16. In Figure 6.6, we see how the speedup values are distributed for them.

  For both single-core and multi-core experiments, we see that the speedup values of the experiments performed using these brushes lie close to 1.0 with the single-core experiments leaning more towards slowdowns. With a data layout of 2×2, we notice that most values lie above 1.0 in the multi-core experiments. The single-core experiments distribution remains centered around 1.0 even with a 2×2 data layout.



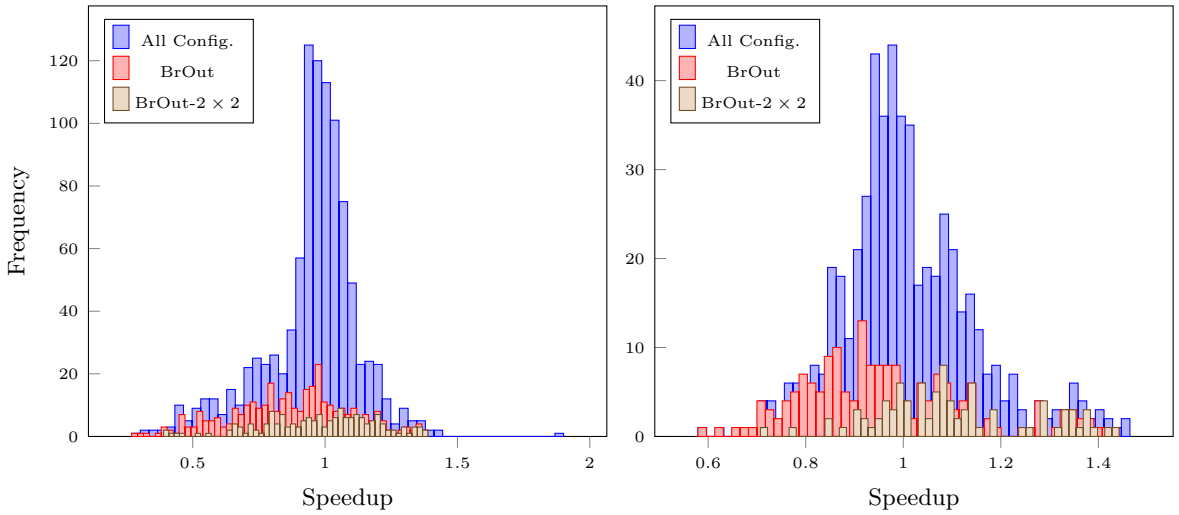
(A) Single-core Experiments (B) Multi-core Experiments

FIGURE 6.6: Frequency distribution of speedups over single-dimensional vectorized code for tensor brushes having larger sizes in the inner loop dimension

- **Brush Outer (BrOut):** This set consists of brushes that have larger sizes in the outermost dimension: 4×2, 8×2, 16×2. In Figure 6.7, we show the distribution of speedup values for this brush set.

  For this set, we see more speedup values in the region below 1.0 for both single-core and multicore experiments. This indicates that we have more slowdowns. However, for the experiments with a 2×2 data layout, we that the distribution shifts towards higher speedup values. It implies that most of the lower performance values come from the 1×1 data layout.

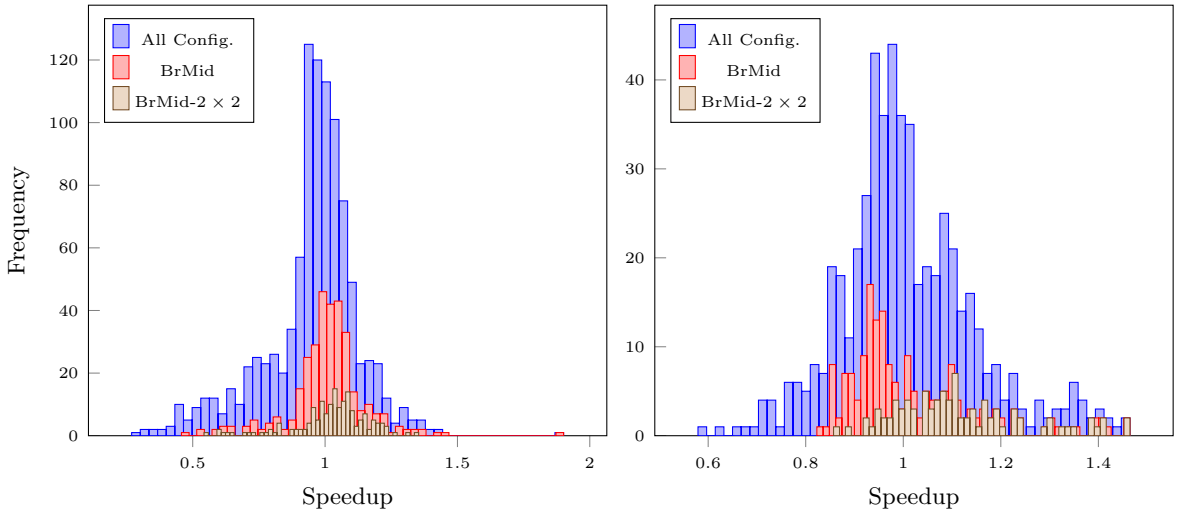
- **Brush Middle (BrMid):**This set consists of tensor brushes that have somewhat even size in both inner and outer dimensions: 4×4, 4×8, 8×4. We show the performance distribution for this set in Figure 6.8.



(A) Single-core Experiments (B) Multi-core Experiments

FIGURE 6.7: Frequency distribution of speedups over single-dimensional vectorized code for tensor brushes having larger sizes in the outer loop dimension

The speedup values of the single-core experiments are concentrated around 1.0 while those of multi-core experiments are mostly between 0.8 and 1.0. We can also see that the highest speedup value in single-core experiments comes from this brush set. The distribution shows that for multi-core experiments, these brushes lead to more slow downs than BrIn but less than BrOut.



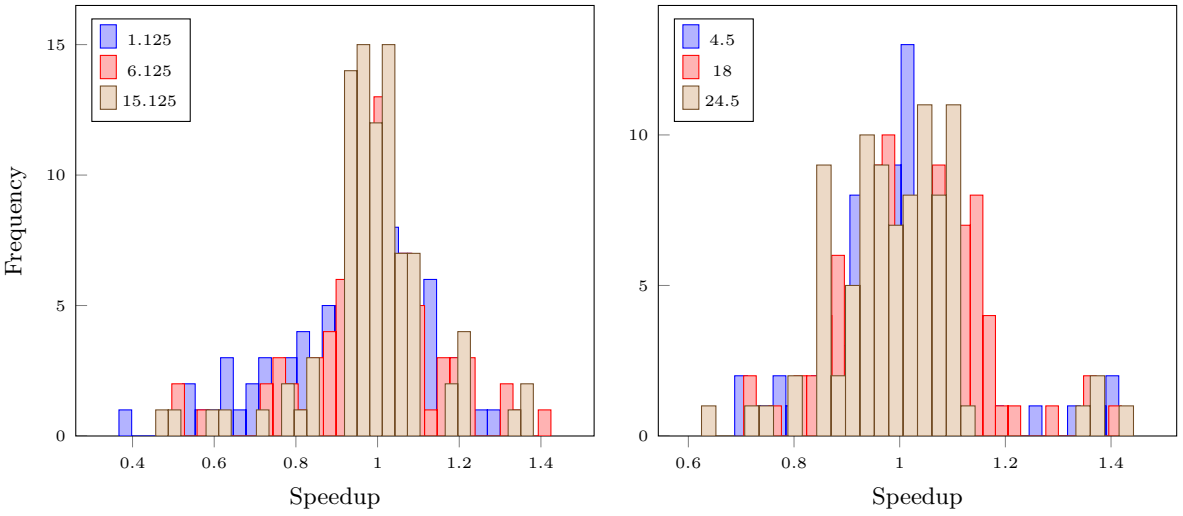
(A) Single-core Experiments (B) Multi-core Experiments

FIGURE 6.8: Frequency distribution of speedups over single-dimensional vectorized code for tensor brushes having even sizes in inner and outer dimensions

In the experiments with a 2×2 data layout, we see that the distribution shifts towards higher speedup values for both single-core and multi-core experiments. This shows that similar to the BrOut set, the 2×2 data layout improves performance for the configurations that use brushes from this set.

6.2.3 Input Size

We evaluate the effect of the size of the input matrix by selecting three different input sizes for both single-core and multi-core experiments and observing the distribution of speedup values. According to our hypothesis, with the increase in size of input, the variation in the speedup values must increase. For single-core experiments, we selected input sizes: 1.125 MiB, 6.125 MiB, 15.125 MiB. For multi-core we selected input sizes: 4.5 MiB, 18.0 MiB, 24.5 MiB.



(A) Single-core Experiments (B) Multi-core Experiments

FIGURE 6.9: Frequency distribution of speedups over single-dimensional vectorized code for different input sizes

In Figure 6.9, we see the distribution of speedup values over different input sizes for single-core and multi-core experiments. The variation in the speedup values does not change with the change in input sizes. We cannot distinguish between the three distributions. Therefore, our hypothesis is unfounded in this case.

6.2.4 Stencil Pattern

In Table 6.1, we show the best performing configurations for single-core and multi-core experiments for a 1×1 data layout. We computed the speedup values for the table by averaging over all input sizes.

We see that for single core experiments, all stencils perform best with a tensor brush layout of 4×8 or 4×4. This is because the increase of the tensor brush size in the outer dimension, increases the number of memory accesses needed per iteration. On the other hand, if we have a larger tensor brush size in the inner dimension, we perform more iterations of the outer loop. A balanced tensor brush performs best in such situations.

We also note that for the 5-point Jacobi and 5-point Sp-Jacobi stencils, the speedup values are below 1.0 which indicates that multi-dimensional vectorization of these stencils leads to slower performance. The speedup values are correlated with the number of points in the stencils. With the increase in the number of points we see an increase in the speedup values.

Stencil	Single Core		Multi Core	
	T. Brush	Speedup	T. Brush	Speedup
5-pt, Jacobi	4×8	0.97	2×4	0.99
9-pt, Seidel	4×4	1.02	2×16	0.99
5-pt, Sp-Jacobi	4×4	0.98	2×8	0.98
9-pt, Jacobi	4×8	1.04	2×4	0.99
25-pt, Seidel	4×8	1.37	2×16	1.01

TABLE 6.1: Best performing configurations for 2D stencils for single-core and multi-core experiments for a 1×1 data layout

In multi-core experiments, we see that all stencils have tensor brushes that have larger sizes in the inner dimension. We can also see that the speedup values are within 0.98 to 1.01. This indicates that multi-dimensional vectorization does not improve performance in our multi-core experiments with a 1×1 data layout.

In Table 6.2, we show the best performing configurations for single-core and multi-core experiments for a 2×2 data layout. Similar to the 1×1 data layout table, the speedup values were computed by averaging over all input sizes.

Stencil	Single Core		Multi Core	
	T. Brush	Speedup	T. Brush	Speedup
5-pt, Jacobi	16×2	1.14	16×2	1.30
9-pt, Seidel	4×8	1.00	16×2	1.20
5-pt, Sp-Jacobi	16×2	1.20	16×2	1.26
9-pt, Jacobi	8×4	1.16	16×2	1.24
25-pt, Seidel	4×8	1.16	4×8	1.10

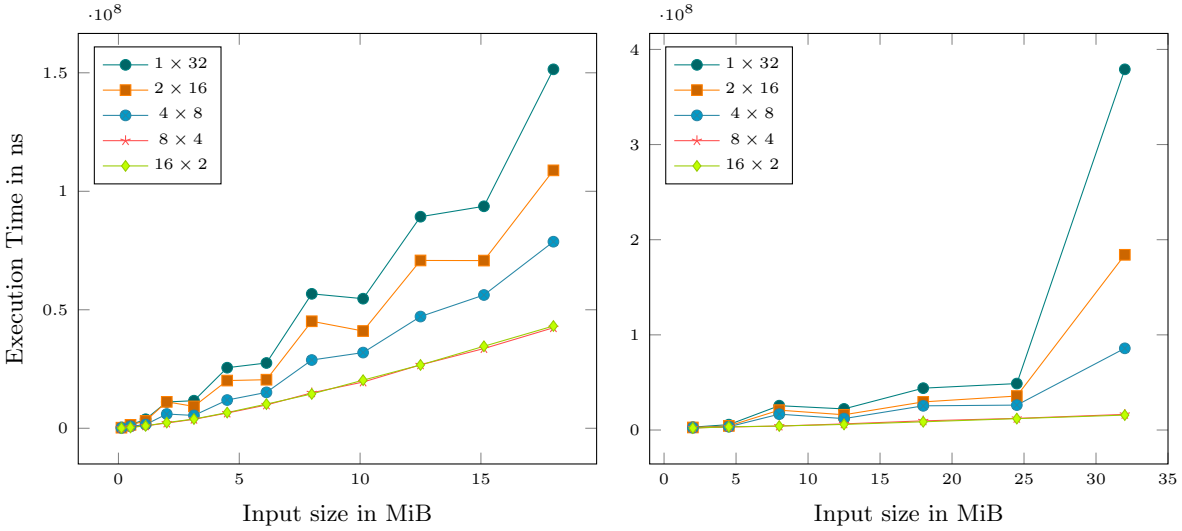
TABLE 6.2: Best performing configurations for 2D stencils for single-core and multi-core experiments for a 2×2 data layout

In single-core experiments, we see that the 5-point Jacobi and 5-point Sp-Jacobi stencils perform best with a 16×2 tensor brush. Compared to the 1×1 data layout, the speedup values are better for all the Jacobi stencils and worse for the two Seidel stencils.

In multi-core experiments, we see that except for the 25-point Seidel stencil, all other stencils perform best with a 16×2 tensor brush. We see an inverse correlation between the speedup values and the number of points in the stencil in this case. With the increase in the number of points in the stencil, we see a decrease in the speedup values.

6.2.5 Matrix Transpose

In Figure 6.10, we show the impact of input size on the execution time of the matrix transpose benchmark for different tensor brushes. The single-dimensional vectorization brush is labeled as 1×32.



(A) Single-core Experiments (B) Multi-core Experiments

FIGURE 6.10: Impact of input size on execution time of the matrix transpose benchmark

In single-core experiments, we see that with the increase of input size the execution time increases for all tensor brushes. However, as the brush layout moves from 1×32 to 16×2, the slope of the curve decreases. This indicates that the tensor brushes with larger sizes in the outer loop dimension are less sensitive to the increase in input size. The curves for brushes 8×4 and 16×2 overlap each other.

In multi-core experiments, we see that there is very little increase in execution time for the brushes 8×4 and 16×2. After the 25 MiB mark, we see a sharp increase in the slope of the curve for the single-dimensional vectorization layout. Once again, we can see that the slope of the curve decreases as we increase the size of the outer dimension and reduce the size of the inner dimension of our tensor brush.

6.3 Discussion

Based on our observations, we distinguish between the different choices of tensor brushes and data layouts based on whether we are executing the stencil on a single-core or multi-core system. In the following paragraphs we discuss both scenarios.

Stencil	Single Core			Multi Core		
	D. Brush	T. Brush	Speedup	D. Brush	T. Brush	Speedup
5-pt, Jacobi	2×2	16×2	1.14	2×2	16×2	1.30
9-pt, Seidel	1×1	4×4	1.02	2×2	16×2	1.20
5-pt, Sp-Jacobi	2×2	16×2	1.20	2×2	16×2	1.26
9-pt, Jacobi	2×2	8×4	1.16	2×2	16×2	1.24
25-pt, Seidel	1×1	4×8	1.37	2×2	4×8	1.10
Transpose	1×1	8×2	3.58	1×1	16×2	7.48

TABLE 6.3: Best performing configurations for 2D benchmarks for single-core and multi-core experiments

In single-core experiments, the stencils with more number of points – the Seidel stencils, performed better with a 1×1 layout. This is due to the higher overlap between the neighboring elements of these stencils. In contrast, the Jacobi stencils which have less points for the same order did not gain anything from multi-dimensional vectorization with a 1×1 layout. However, with a 2×2 layout that reduces memory accesses, we see an increase in speedups for Jacobi stencils. For the performance of the Seidel stencils, the reduction in the memory accesses from a 2×2 data layout is not influential. Due to the higher number of points in these stencils, their performance is influenced more by the number of computations per iteration than the number of memory accesses.

In general, we see that multi-core experiments on stencils perform better with a 2×2 data layout. This was expected because multi-core execution shares the last level of cache and it would benefit from a data layout of 2×2 which reduces memory operations.

We have also seen that tensor brushes that have larger sizes in the outer loop dimension, perform better with a 2×2 data layout. This is because these tensor brushes load only 2 values contiguously in a 1×1 layout. This leads to more memory accesses per iteration. In contrast, a data layout of 2×2 reduces the number of required memory accesses to half of the earlier amount as it doubles the number of elements loaded contiguously. The advantage of these brushes is that they reduce the number of outer loop iterations which are relatively more expensive. However, in a 1×1 layout, the cost of memory accesses per iteration slows down their performance. For stencils like Jacobi, that are influenced more by the memory accesses, these tensor brushes give the best performance with a 2×2 data layout.

In the matrix transpose, we see a similar gain in speedup when increasing the size of the tensor brush in the outer dimension. In Table 6.3 we see that the best performing tensor brush layouts for matrix transpose are 8×2 and 16×2.

Chapter 7

Related Work

The idea of multi-dimensional loop vectorization can be traced back to Allen and Kennedy's work on the Parallel Fortran Compiler [10]. As a source to source translator, their framework would analyze loop nests for data dependencies and vectorize independent inner loops to generate multi-dimensional *Fortran 8x* [11] code. In contrast, our framework deals with LLVM IR and can be used for different source languages.

Since most modern systems have single-dimensional memory, even contiguous accesses in higher dimensions lead to strided memory accesses that require gather and scatter instructions. Such instructions are not as efficient as contiguous vector memory access instructions and can be detrimental to performance in multi-dimensional vectorization. This was addressed in Vector Folding [8] by performing data layout transformations as a preprocessing step before multi-dimensional vectorization. Data layout transformations have also been used [12] to reduce shuffle operations induced by unaligned memory accesses. However, there is an associated cost of transforming the data layout both before and after the execution of the stencil kernel code. Depending on the stencil pattern, vectorization layout and hardware platform, data layout transformations can increase or decrease throughput. In our approach, we group memory accesses and use shuffles with contiguous loads to generate operand vectors [13, 14]. We do this with the default data layout as well as with transformed data layouts.

For loop nests with short trip counts, single-dimensional vectorization leads to under-utilization of available SIMD lanes. Rodrigues et al. [15] showed that multi-dimensional vectorization enables efficient vector register utilization in such cases. However, their compiler requires a high-level specification of the tensor operation algorithm and needs contiguous memory accesses without gaps. Our framework uses scalar code as input and can handle non-constant strides. There is limited support for multi-dimensional vectorization in the ISPC programming language [16] through the `foreach_tiled` statement. However, lacking a multi-dimensional analysis, ISPC will use scatter/gather to vectorize every memory access that is not fully uniform in that mode.

Our approach is orthogonal to spatio-temporal tiling approaches [17, 18] in the sense that the generated tiles may still be processed by multi-dimensional vector code.

Optimization of stencil codes on SIMD hardware is an active area of research and various manual [19] and automatic [15, 20] vectorization techniques have been developed to improve throughput. Such applications are well suited to optimization techniques that reorder computations to improve register reuse [21] and increase arithmetic intensity. Kong et al. [22] have shown that register tiling [23–25] is useful for exploiting multi-dimensional data reuse. For certain combinations of vectorization layout and hardware platform, our framework generates register tiled code. However, if the register pressure is high, our framework uses smaller vectorization layouts and shuffles elements between registers to build operand vectors.

The multi-dimensional vectorization analysis is an extension of divergence analysis [1, 2]. The strides in the vector shapes are comparable to affine constraints in some, one-dimensional, divergence analysis lattices [1, 9].

A part of this work has already been published as Tensorization [3] of loop nests. In that work, we limited our framework to the default data layout in memory. Our experiments were also restricted to single core execution of 2D stencils. In this work, we further extend our framework to handle transformed data layouts and conduct single core and multi core experiments on 2D stencils.

Chapter 8

Conclusion

In this thesis we have developed a framework that analyzes scalar code for stencil patterns and generates efficient vectorized code by reducing memory operations. The framework groups contiguous memory accesses together to substitute multiple scalar memory operations by fast contiguous vector memory operations. This requires additional shuffle operations to build operand vectors for the computations which increases register usage. A key limitation of this approach is that it depends on the regular grid-like memory access pattern in stencil codes to compute memory groups that can be loaded contiguously.

Our framework performs multi-dimensional loop vectorization in three phases. The tensor shape analysis phase of the framework computes the memory access pattern across loop iterations that are vectorized into a single vector loop iteration. The memory grouping phase uses LLVM's Scalar Evolution analysis to detect the memory access pattern within the same loop iteration. Further, it also generates SCEV representations for neighboring memory addresses and groups them together. Finally, the code generation phase generates vector code using the memory groups and tensor shapes.

Unlike existing techniques, our framework does not limit itself to processing vectorization layouts that are at least as wide as the vector registers in the innermost dimension. It also does not require the size of the memory blocks in the transformed data layouts to be equal to the width of vector registers. However, it assumes that the loop bounds are such that the loop can be fully vectorized. It requires a vectorization layout and a data layout as input from the programmer together with the scalar code.

Our experiments show that with the right parameter configurations, the generated vector code can improve performance by up to 90% for stencil applications. However, we also note that wrong parameter configurations can lead to significant slowdowns. We discussed the influence of different parameters on performance. We found that for multi-core experiments, a multi-dimensional data layout performs better than the default

single-dimensional layout. For single-core experiments we observed that the choice of a good tensor brush depends mainly on the number of points in the stencil and the data layout of the inputs.

8.1 Future Work

Our framework needs a cost model to be completely automatic. Currently, it relies on the programmer to provide an efficient vectorization and data layout for the input code. The framework analyzes the properties of the stencil pattern. This information can be combined with the heuristics from our experiments to build a cost model.

The code generator is designed for the AVX-512 instruction set. If possible, it replaces memory operations by register local shuffle operations because they are faster. For systems that do not support fast register shuffle operations this code generator would not generate efficient code. It could be extended to other hardware platforms like GPUs that have large memory latencies and would benefit from this approach.

List of Figures

List of Tables

Bibliography

[1] Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4):13:1–13:36, 2013. doi: 10.1145/2523815. URL https://doi.org/10.1145/2523815.

[2] Ralf Karrenberg and Sebastian Hack. Improving performance of opencl on cpus. In *Compiler Construction - 21st International Conference, CC 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 1–20, 2012. doi: 10.1007/978-3-642-28652-0_1. URL https://doi.org/10.1007/978-3-642-28652-0_1.

[3] Simon Moll, Shrey Sharma, Matthias Kurtenacker, and Sebastian Hack. Multi-dimensional vectorization in LLVM. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*, page 3. ACM, 2019.

[4] Sebastian Pop, Albert Cohen, and Georges-André Silber. Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and Compilers, First International Conference, HiPEAC 2005, Barcelona, Spain, November 17-18, 2005, Proceedings*, pages 218–232, 2005. doi: 10.1007/11587514_15. URL https://doi.org/10.1007/11587514_15.

[5] Simon Moll and Sebastian Hack. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 543–556, 2018. doi: 10.1145/3192366.3192413. URL http://doi.acm.org/10.1145/3192366.3192413.

[6] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004. doi: 10.1109/CGO.2004.1281665. URL https://doi.org/10.1109/CGO.2004.1281665.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. doi: 10.1145/115372.115320. URL https://doi.org/10.1145/115372.115320.

[8] Charles Yount. Vector folding: Improving stencil performance via multi-dimensional simd-vector representation. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*, pages 865–870, 2015. doi: 10.1109/HPCC-CSS-ICESS.2015.27. URL `https://doi.org/10.1109/HPCC-CSS-ICESS.2015.27`.

[9] Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. Pacxxv2 + RV: an LLVM-based portable high-performance programming model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2017, Denver, CO, USA, November 13, 2017*, pages 7:1–7:12, 2017. doi: 10.1145/3148173.3148185. URL `https://doi.org/10.1145/3148173.3148185`.

[10] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987. doi: 10.1145/29873.29875. URL `https://doi.org/10.1145/29873.29875`.

[11] T Lahey. The fortran 8x standard. In *ACM SIGPLAN Fortran Forum*, volume 6, pages 27–30. ACM, 1987.

[12] Thomas Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector SIMD architectures. In *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 225–245, 2011. doi: 10.1007/978-3-642-19861-8_13. URL `https://doi.org/10.1007/978-3-642-19861-8_13`.

[13] Diego Caballero, Sara Royuela, Roger Ferrer, Alejandro Duran, and Xavier Martorell. Optimizing overlapped memory accesses in user-directed vectorization. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 393–404, 2015. doi: 10.1145/2751205.2751224. URL `http://doi.acm.org/10.1145/2751205.2751224`.

[14] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 82–93, 2004. doi: 10.1145/996841.996853. URL `http://doi.acm.org/10.1145/996841.996853`.

[15] Christopher Rodrigues, Amarin Phaosawasdi, and Peng Wu. Simdization of small tensor multiplication kernels for wide SIMD vector processors. In *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*, pages 3:1–3:8, 2018. doi: 10.1145/3178433.3178436. URL `https://doi.org/10.1145/3178433.3178436`.

[16] M. Pharr and W. R. Mark. ispc: A spmd compiler for high-performance cpu programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13, May 2012. doi: 10.1109/InPar.2012.6339601.

[17] Tareq M. Malas, Georg Hager, Hatem Ltaief, and David E. Keyes. Multidimensional intratile parallelization for memory-starved stencil computations. *TOPC*, 4(3):12:1–12:32, 2018. doi: 10.1145/3155290. URL `http://doi.acm.org/10.1145/3155290`.

[18] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine A. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009. doi: 10.1137/070693199. URL `https://doi.org/10.1137/070693199`.

[19] Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne. Small SIMD matrices for CERN high throughput computing. In *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*, pages 1:1–1:8, 2018. doi: 10.1145/3178433.3178434. URL `https://doi.org/10.1145/3178433.3178434`.

[20] Kevin Stock, Thomas Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert J. Harrison. Model-driven SIMD code generation for a multi-resolution tensor kernel. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 1058–1067, 2011. doi: 10.1109/IPDPS.2011.101. URL `https://doi.org/10.1109/IPDPS.2011.101`.

[21] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 65–76, 2014. doi: 10.1145/2594291.2594342. URL `https://doi.org/10.1145/2594291.2594342`.

[22] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation.

In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 127–138, 2013. doi: 10.1145/2491956.2462187. URL https://doi.org/10.1145/2491956.2462187.

[23] Alejandro Berna, Marta Jiménez, and José María Llabería. Source code transformations for efficient simd code generation. 2012.

[24] Lakshminarayanan Renganarayanan, Uday Bondhugula, Salem Derisavi, Alexandre E. Eichenberger, and Kevin O'Brien. Compact multi-dimensional kernel extraction for register tiling. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*, 2009. doi: 10.1145/1654059.1654105. URL https://doi.org/10.1145/1654059.1654105.

[25] Larry Carter, Jeanne Ferrante, and Susan Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *Proceedings of IPPS '95, The 9th International Parallel Processing Symposium, April 25-28, 1995, Santa Barbara, California, USA*, pages 239–245, 1995. doi: 10.1109/IPPS.1995.395939. URL https://doi.org/10.1109/IPPS.1995.395939.