Automatic Vectorization through Superword Level Parallelism with Associative Chain Reordering and Loop Shifting

by

Stephen Rogers

Thesis

Submitted to the School of Computer Science and Statistics in partial fulfilment of the requirements for the degree of

Master in Science

(Computer Science)

School of Computer Science and Statistics

TRINITY COLLEGE DUBLIN

February 2018

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Declaration

. . . .

Stephen Rogers

Dated: February 2018

Abstract

Single instruction, multiple data (SIMD) is a class of parallel computing that involves executing a single operation across multiple pieces of data. A common type of SIMD is vector processing which involves executing a single instruction across 1-dimensional arrays of data called vectors. A category of compiler optimization called automatic vectorization has been developed since the introduction of vector processing to allow "vectorizing compilers" to target such processor capabilities without direct intervention from application programmers.

Convolution is a fundamental concept in image processing. It involves the application of a matrix called a kernel to weight the sum of a pixel and its adjacent pixels, for all pixels in an image. This process is used to perform tasks like image blurring, edge detection and noise reduction.

In this thesis, we explore the challenges of automatic vectorization of image convolutions implemented in C and C++. We describe the fundamentals of vectorization and image convolutions and propose an approach for the effective vectorization of these convolutions. Our approach combines vectorization through Superword Level Parallelism with tentative loop unrolling, loop shifting, and the reordering of associative and commutative chains of instructions.

Most modern optimizing compilers are capable of vectorizing 3x3 image convolutions, but tend to fail at vectorizing larger sized convolutions, like 5x5. The vectorizer we describe in this thesis, with the aid of its combined optimizations, is designed to vectorize such larger convolutions.

Through this combination of optimizations, we have measured performance improvements for 5x5, 7x7, and 9x9 image convolutions. For convolutions operating on integer data types we measured performance improvements between 2.01x and 6.97x, and for floating-point types, between 2.19x and 5.34x.

Acknowledgements

I would like to start by thanking my family and friends for their encouragement during my time in Trinity College.

A big thank you is also due to my colleagues in Movidius for facilitating the time and support I needed to focus on my studies. In particular, I would like to thank Martin O'Riordan for his help and support in making this thesis possible.

Finally, it has been a great pleasure to work with my supervisor David Gregg over the last year. I would like to sincerely thank him for all his guidance and advice while working on this project.

Table of Contents

Chapter 1 Introduction

Single instruction, multiple data (SIMD) is a class of parallel computing described in Flynn’s Taxonomy [1]. It involves the execution of a single operation across multiple pieces of data at once. One common type of SIMD is vector processing which has been prevalent in computing since its introduction in the 1970s and 1980s in vector supercomputers. In modern processor architectures, this class of computing is typically made available through vector extensions.

Since their introduction, the programmability of these vector processors has been much debated. High-level programming languages were often extended to include vector support to prevent the need for direct assembly programming. In 1982, a method for automatically translating applications written in earlier versions of Fortran into a newer form of the language with vectorization support was developed [2]. Since then, programming languages like C and C++ have been extended by various third parties to include vector programming support [3].

The topic of automatic vectorization of applications by “vectorizing compilers” has also been discussed in detail. This kind of vectorization can come in the form of loop vectorization [4], Superword Level Parallelism [5], whole function vectorization [6], and more.

A convolution is a common mathematical operation which expresses the overlap between two functions, as one is shifted over the other [7]. It has applications in the likes of image processing, signal processing, and convolutional neural networks. In image processing, a convolution is applied on an image using a kernel to perform tasks like blurring, edge detection, and noise reduction.

In this thesis, we discuss each of the forms of vectorization mentioned above and how they pertain to the automatic vectorization of image convolutions. In particular, we describe the challenges involved in automatic vectorization of these convolutions. Most modern optimizing compilers are capable of vectorizing 3x3 image convolutions, but tend to fail at vectorizing larger sized convolutions, like 5x5, 7x7 and 9x9.

In Chapter 2, we provide a discussion of the background information and literature associated with this thesis. We begin by describing Very Long Instruction Word architectures and the Movidius SHAVE processor. We then move on to describe the various methods that can be used to perform vectorization, both manual and automatic. We provide a description of image convolutions and how each of the previously mentioned automatic vectorization approaches can be applied to them. We also provide a brief description of Clang, LLVM, and LLVM's intermediate representation, LLVM-IR.

In Chapter 3, we describe our approach to Superword Level Parallelism (SLP) which we have implemented as an automatic vectorizer in LLVM. This vectorizer serves as a base implementation on which further optimizations are built.

These optimizations are described in Chapter 4. In this chapter, we describe three additional optimizations:

- The first optimization is a tentative loop unroller which is designed to discover vectorization opportunities in loops for our SLP vectorizer.
- The second optimization reorders chains of associative and commutative operations to allow them to be vectorized.
- The third and final optimization is based on loop shifting and seeks to restructure loops and move memory load instructions to take advantage of data re-use between iterations of the loop.

In Chapter 5, we describe how these optimizations work in tandem to vectorize image convolutions, using a practical example.

In Chapter 6, we evaluate the performance results generated by each of the optimizations. We start by examining each optimization individually and then examine all optimizations together using a set of image convolutions. We examine convolution tests using integer and floating-point types of various bit-sizes.

We conclude this thesis in Chapter 7 by discussing some possible future work and presenting some final thoughts on our approach to vectorization.

Chapter 2 Background and Literature Survey

2.1. Very Long Instruction Word Architectures

Very Long Instruction Word (VLIW) architectures were first developed as a means of achieving significant performance improvements by exploiting instruction level parallelism in applications [8] [9]. VLIW architectures achieve these improvements by simplifying the hardware implementation, but at the cost of more complex compiler support [10]. VLIW architectures typically contain multiple, discrete functional units which are independently programmable. For example, consider the theoretical VLIW processor architecture shown in Figure 1. It contains two register files and twelve discretely programmable functional units, including:

- Four load/store units for accessing memory
- Two floating-point arithmetic logic units (ALUs) for performing floating point operations, operating on values in the floating-point register file
- Two integer ALUs for performing integer operations on values in the integer register file
- A copy/convert unit for copying values between registers in the same register file, and converting copies between the two register files
- A compare unit and predication unit which together are used to predicate the execution of instructions based on values stored in either register file
- A branch unit for performing jumps, procedure calls and returns.

Unlike many modern processors which leverage “out-of-order” instruction execution techniques [11], VLIW processors are statically scheduled [12]. This means that the compiler (or assembly programmer) is solely responsible for deciding which functional units will execute which operations on any given cycle. If there is no operation for any given functional unit in a VLIW instruction, then that functional unit will remain idle for that cycle (i.e. it will execute no operation or a NOP).

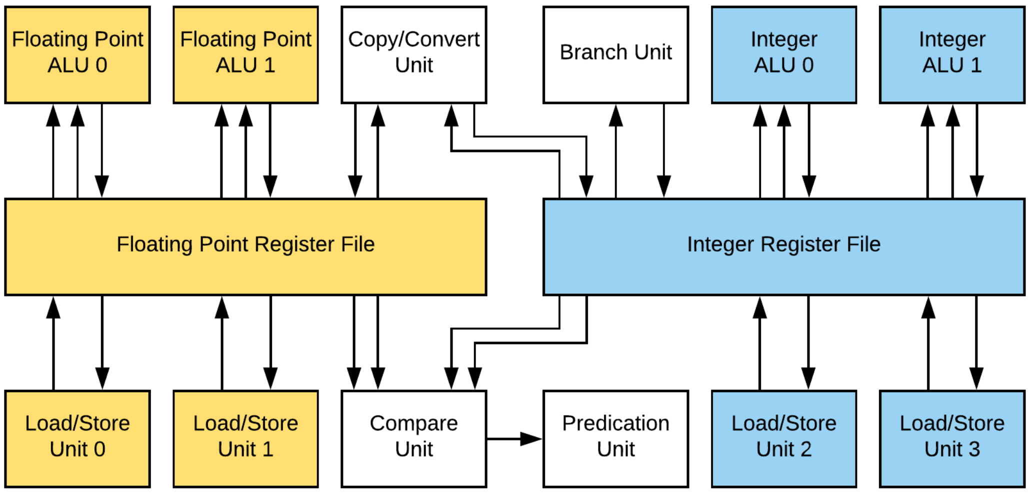


Figure 1 Theoretical VLIW processor architecture with twelve functional units

Each instruction encoded in an application contains a field for each functional unit in the processor. A functional unit will execute operations encoded in its own field in the instruction word. For our example architecture in Figure 1, each instruction contains twelve fields, one for each functional unit. This is shown in Figure 2. The field for each of the functional units contains the specific operation that that unit should execute for this instruction.

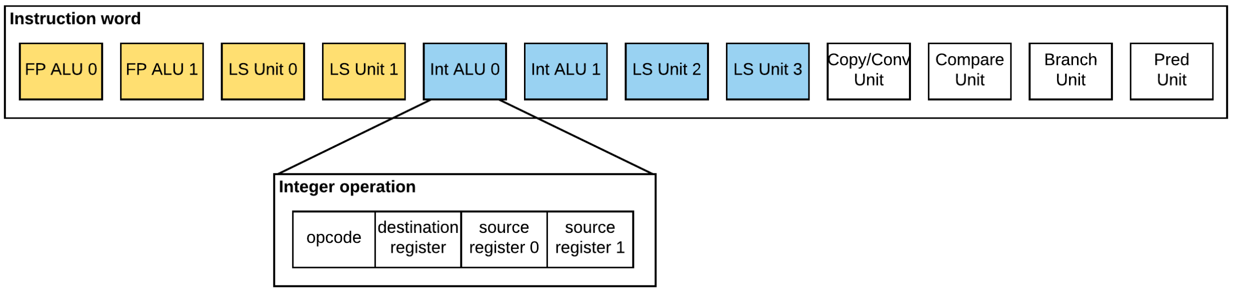


Figure 2 Instruction word for our theoretical VLIW architecture

An early example of a VLIW machine is the ELI-512 [9]. The ELI-512 was a VLIW architecture with 16 functional units and a 512-bit instruction word. The architecture of the ELI-512 was developed alongside the Bulldog compiler [13], which introduced the concept of trace scheduling. Trace scheduling is an instruction scheduling technique used in compilers which exploits instruction level parallelism between multiple basic blocks of a program. This technique enabled the use of the ELI-512, as previously it was not feasible to program such a VLIW machine [9].

Another compiler technique which can be used to efficiently schedule instructions for VLIW architectures is software pipelining [14] [15]. Software pipelining allows for multiple iterations of a loop to be at different stages of their execution simultaneously. Essentially, this means that multiple iterations of a loop partially overlap and are executed on different functional units via the same instruction word.[1]

Techniques like trace scheduling and software pipelining enable compilers to effectively target VLIW architectures.

2.2. Movidius SHAVE

The Movidius Myriad 2 Vision Processing Unit (VPU) incorporates a multitude of interconnected hardware components targeted at supporting computer vision and visual awareness in low power environments [16]. One of these components is the SHAVE processor. Myriad 2 incorporates twelve SHAVE cores [17]. A simplified overview of SHAVE is shown in Figure 3.

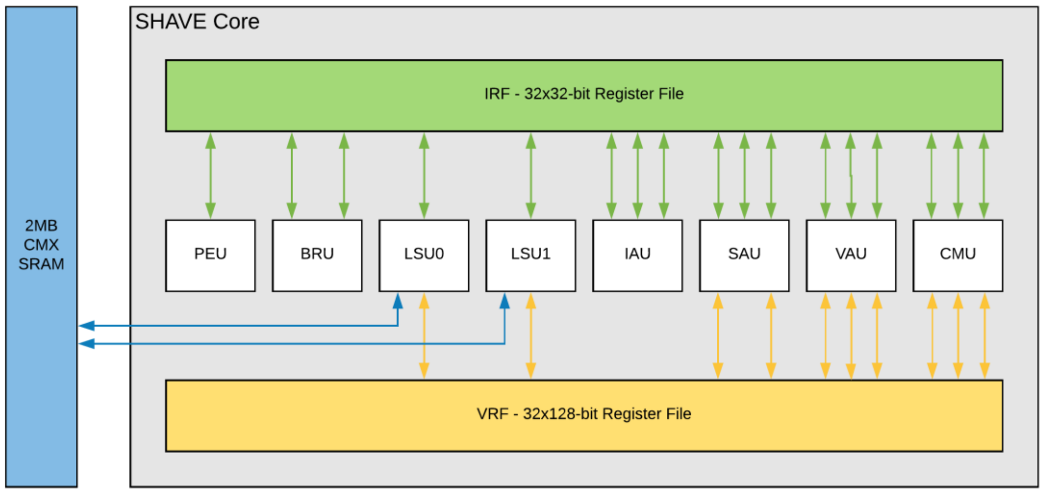


Figure 3 Simplified overview of the SHAVE architecture

[1] Software pipelining can also be used to exploit thread-level parallelism in multi-core processor architectures. Huang et al. [55] discuss one method of software pipelining as an "enabling transformation" for other loop parallelization techniques in multi-threaded platforms.

SHAVE is a VLIW processor with eight functional units and a variable length instruction word. One of the functional units on SHAVE is a 128-bit Vector Arithmetic Unit (VAU) with support for 8, 16, and 32-bit integer operations and 16, and 32-bit floating point operations. Through this functional unit (and the associated instructions in the two Load-Store Units (LSU) and Compare Move Unit (CMU)) SHAVE has single instruction, multiple data (SIMD) capabilities as well as the multiple instruction, multiple data (MIMD) capabilities inherent in a VLIW processor. SHAVE also contains a Predicated Execution Unit (PEU), a Branch and Repeat Unit (BRU), Integer Arithmetic Unit (IAU), and a Scalar Arithmetic Unit (SAU) which provides support for both integer and floating-point operations. The two LSUs shown in Figure 3 are used to access the 2MB of shared on-chip Connection Matrix (CMX) memory, as well as the larger stacked DRAM [16].

2.3. Manual Vectorization

With the advent of the mainstream success of vector processors in the 1970s and 1980s, there was a need for the introduction of vector operations in high-level programming languages [18]. Since then, several techniques have been developed to allow programmers to manually generate vector operations in their applications without the need for writing assembly code.

With Fortran 90, the language was introduced to the concept of array processing. Array processing allowed programmers to specify operations on entire arrays as a single operation, a kind of large vector processing [19]. This new feature was accompanied by several intrinsic functions that operate on array values, supplementing the ability of the programmer to manually vectorize their applications. As mentioned by Allen and Kennedy [18], features of this new Fortran standard came as a result of the standards committee desire to provide programmers a way to "explicitly specify vector and array operations".

While the C programming language does not provide any capabilities for manual vectorization of code, there are language extensions provided by compilers that do.

For example, GCC provides a method for creating vector types of constant size for any of the integer and floating point primitive types in C [3]. This is made available via the `vector_size` attribute that can be attached to type definitions. Vector types declared in this manner can be used with some standard C operators. GCC also provides some built-in functions for performing more complex vector operations, like vector element shuffling. Clang provides support for the GCC vector extensions, as well as the OpenCL, AltiVec and NEON vector extensions [20].

Although there have been significant advances in automatic vectorization in compilers in recent years, there are still many cases where manual vectorization provides far greater performance improvements [21]. However, these generally come at the cost of more complex source code and a more involved development process [22].

2.4. Automatic Vectorization

2.4.1. High-Level Programming Language Translators

Since the introduction of vector processing in mainstream processors, there have been many different approaches to automatic vectorization developed. An early approach to vectorization was to provide translators from older versions of a language without vector processing support into a newer form with vector support [2]. As mentioned in section 2.3, with Fortran 90 the concept of array processing was introduced to the language. In 1982, John R. Allen and Ken Kennedy developed a method for translating Fortran 66 or 77 to Fortran 90 (then called Fortran 8x) with minimal effort required by the programmer [2]. This approach allowed the programmer to further hand-tune the generated Fortran 8x for any opportunities the translator missed. The approach was further developed and in 1987 Randy Allen and Ken Kennedy published their approach to a new version of this translator [18].

2.4.2. Vectorizing Compilers

A more common approach to automatic vectorization in modern times is to develop vectorizing compilers. Vectorizing compilers are designed to take scalar source code and automatically transform it to vector form, without any additional input from the programmer. Approaches to automatic vectorization in compilers can be broadly separated into two groups: loop vectorization and basic block vectorization. There are however, other, less common approaches to vectorization like whole-function vectorization [6] which attempts to exploit vectorization across entire functions in data-parallel programming languages. While many automatic vectorization techniques like these were developed to target programming languages like Fortran, they have since been adapted and supplemented to target more recent languages like C [23].

In 2011, Maleki et al. performed an evaluation of vectorizing compilers available at the time [24]. They examined the capabilities of GCC, the Intel C Compiler (ICC) and IBM XLC. They found that each of these compilers could automatically vectorize between 45 and 71% of their synthetic benchmarks, although collectively they could vectorize around 83.05% of these benchmarks. However, they also found that these compilers could only vectorize between 18 and 30% "of the loops extracted from the PACT and Media Bench II codes" [24].

2.4.3. Loop Vectorization

Loop vectorization can come in one of two forms: inner-loop vectorization and outer-loop vectorization. As discussed earlier, many modern C and C++ compilers contain inner-loop vectorization support, including GCC, Clang/LLVM, Intel ICC, and IBM XLC [24]. This kind of loop vectorizer was first introduced to GCC in early 2004 [25], although they have existed in optimizing compilers for a much longer time [26].

Inner-loop vectorization typically centres around grouping instances of the same operations from multiple, consecutive iterations of a loop together to form equivalent vector operations. Early implementations often required memory accessing operations to access consecutive memory locations to be vectorized [25], however more advanced implementations can handle interleaved memory accesses through the use of scatter/gather vector instructions [27].

Outer-loop vectorization deals with vectorization of loops other than the inner-most loop of a group of nested loops. This approach is more beneficial in applications where an outer loop provides a greater opportunity for vectorization than the inner-most loop through a higher availability of data-level parallelism and data locality [28].

2.4.4. Superword Level Parallelism

Superword Level Parallelism (SLP) was a concept first introduced as a form of basic block vectorization by Larsen and Amarasinghe in 2000 [5]. SLP is designed to exploit vectorization opportunities in a single basic block by grouping sequences of the same operations in the same order together. Such sequences are often referred to as isomorphic instruction sequences. Such sequences can be vectorized through SLP by inserting equivalent operations from each chain into individual vector lanes. This transformation forms a new, equivalent vector sequence of operations in which each vector lane corresponds to one of the original scalar operation sequences. Despite targeting basic block parallelism specifically, SLP can be combined with loop unrolling techniques to perform inner-loop vectorization [29].

SLP has been expanded in subsequent years to create vectorization opportunities in the presence of control flow (i.e. SLP beyond a single basic block) [30]. This is achieved by leveraging instruction predication to flatten simple control flow patterns (e.g. an if-statement) into a single basic block. SLP can then be used to vectorize the predicated instruction sequences.

One major drawback of SLP was identified to be the absence of isomorphic instruction sequences in real-world applications [31]. Porpodas et al. proposed a solution whereby “padding” instructions could be inserted into sequences to force non-isomorphic sequences into isomorphic ones. This transformation is performed only when the cost of adding these padding instructions is outweighed by the benefits of SLP vectorization.

2.4.5. Restrictions on Automatic Vectorization

Regardless of the vectorization strategy employed, there are certain restrictions which can significantly complicate vectorization or prevent it outright.

Memory alignment has been identified as one such restriction [32]. Depending on the target architecture, unaligned vector memory accesses may be significantly slower than their aligned counterparts or may even be an outright error. In 2004, Eichenberger et al. proposed a solution whereby unaligned vector memory accesses are replaced by aligned memory accesses and a sequence of data reorganization operations to maintain the original semantics of the vectorized instruction sequences [32].

Non-contiguous and interleaved data also present a problem for automatic vectorization [33]. Generally, processors with vector capabilities require data to be packed together in vector registers to be processed. This presents a problem when the data to be processed is stored in non-contiguous memory locations. In 2006, Nuzman et al. proposed a technique which supported the automatic vectorization of applications with non-contiguous, power-of-2 constant strides. This was later generalized to include non-power-of-2 constant strides due to their prevalence in real-world use [27].

Control flow can also present problems for not only vectorization [34], but parallelism in general [35]. As mentioned in section 2.4.4, Shin et al. proposed a method for performing SLP vectorization in the presence of control flow by flattening certain patterns into a single basic block using instruction predication [30]. They do this, in-part, using a technique called if-conversion [36]. If-conversion transforms control dependencies into data dependencies by in-lining the body of an if-statement, predicating the execution of each operation individually. If-conversion can be used more generally to enable other forms of vectorization as well.

Pointer aliasing is another problem which can seriously impact automatic vectorization [37]. When vectorizing applications written in languages like C and C++, there can be limited information at compile-time pertaining to where in memory pointers may be pointing. More importantly, whether or not two or more pointers alias the same locations in memory is often ambiguous. This is a common problem which extends beyond the context of automatic vectorization. Many different approaches to alleviate this problem have been proposed over the years [38] [39].

2.5. Clang and LLVM

LLVM is a compiler infrastructure comprised of "a collection of modular and reusable compiler and toolchain technologies" [40] [41]. Throughout this thesis, we will use the name LLVM to refer to the LLVM Core libraries which includes the target-independent optimizer and the group of target architecture backends which provide support for assembly code generation.

All optimizations in the LLVM Core operate on a common intermediate representation called LLVM-IR [42]. An example of LLVM-IR is shown in Figure 4. LLVM-IR is a Static Single Assignment (SSA) representation. A representation is in SSA form when each variable in a function is the target of one, and only one assignment operator [43].

```
define void @multiply(i32* noalias nocapture %out,
                      i32* noalias nocapture readonly %inA,
                      i32* noalias nocapture readonly %inB,
                      i32 %size) local_unnamed_addr #0 {
entry:
  %cmp = icmp eq i32 %size, 0
  br i1 %cmp, label %for.cond.cleanup,
              label %for.body

for.body:
  %i = phi i32 [ %inc, %for.body ],
               [ 0, %entry ]
  %idx0 = getelementptr inbounds i32, i32* %inA, i32 %i
  %load0 = load i32, i32* %idx0, align 4, !tbaa !1
  %idx1 = getelementptr inbounds i32, i32* %inB, i32 %i
  %load1 = load i32, i32* %idx1, align 4, !tbaa !1
  %mul = mul nsw i32 %load1, %load0
  %idx2 = getelementptr inbounds i32, i32* %out, i32 %i
  store i32 %mul, i32* %idx2, align 4, !tbaa !1
  %inc = add nuw i32 %i, 1
  %exitcond = icmp eq i32 %inc, %size
  br i1 %exitcond, label %for.cond.cleanup,
                   label %for.body

for.cond.cleanup:
  ret void
}
```

Figure 4 LLVM-IR example function multiply

When translating a function to LLVM-IR (i.e. to SSA form), there is often a need to introduce PHI instructions at the beginning of basic blocks. PHI instructions are responsible for choosing between multiple values on entry to a block depending on which control flow path was taken into the block. For example, consider the loop iterator variable `%i` shown in Figure 4 which starts at zero. When first entering the block representing the body of the loop, the iterator value should be assigned zero. On subsequent iterations of the loop, when the back-edge of the loop is taken, the iterator value should be assigned the result of the iterator increment instruction from the previous iteration (i.e. the value `%inc`). A PHI instruction is responsible for choosing which of these values should be assigned to the iterator value based on which control flow edge is taken into the block. This is necessary to prevent the need for more than one assignment operation to the iterator value.

Functions like `multiply` shown in Figure 4 are organized into basic blocks in LLVM-IR. A basic block is a contiguous section of code with no branches out except at the end, and no branches in except at the start. LLVM requires that each basic block ends with a "terminator" instruction. Terminator instructions determine which basic block should be executed at runtime following the current block. Examples of terminators in LLVM-IR are `br`, LLVM's branch instruction, and `ret`, which returns control flow from this function to its caller [42]. Due to LLVM's requirement that all basic blocks must end in a terminator instruction, fall-through control flow is achieved using an unconditional branch instruction.

Each value in LLVM-IR is produced by a single instruction. Instructions in LLVM typically produce a single value (including "`void`" values), are defined by an opcode and use one or more values as operands. As an example, consider the `getelementptr` instruction used in the function `multiply`. This instruction is responsible for memory address computation in LLVM-IR [44]. The first operand to a `getelementptr` instruction is the array type of the second operand, which is always a pointer operand. The third, and subsequent operands are used to index the operands which came before. Depending on the complexity of the address computation and the target architecture, a `getelementptr` instruction may not map directly onto any specific instruction in the eventual generated machine code. It may be implicitly rolled into the attached load or store instruction (e.g. a load with a variable base pointer and constant index).

Clang is a frontend for LLVM which provides compilation support for C, C++, Objective-C, and Objective-C++ [45]. At the time of writing this thesis, Clang supports all published ISO C++ standards. Clang provides us with a mechanism for producing equivalent LLVM-IR for C and C++ programs.

All the example functions which we use throughout this thesis were written in either C11 or C++11. As of the C99 standard [46], `restrict` is a keyword in C which can be used as a qualifier on pointer declarations. An object in memory that is accessed through a restrict-qualified pointer is only ever accessed through that same pointer within the scope of that pointer. The `restrict` keyword provides a mechanism for the programmer to tell the compiler that certain pointers will never alias the same locations in memory [47]. Clang allows the use of the keyword `__restrict` with both C11 and C++11 in place of `restrict`. Because C++11 does not include support for the `restrict` keyword [48], we will use `__restrict` for all examples for the sake of simplicity. This keyword is translated to the `noalias` attribute on LLVM-IR function arguments, as shown in Figure 4.

2.6. Image Convolutions

In image processing, a two-dimensional convolution is the application of a square matrix called a kernel to produce a weighted sum of clusters of pixels in an image [49]. Consider the image shown in Figure 5. It shows the convolution of a section of an image using a 3x3 kernel.

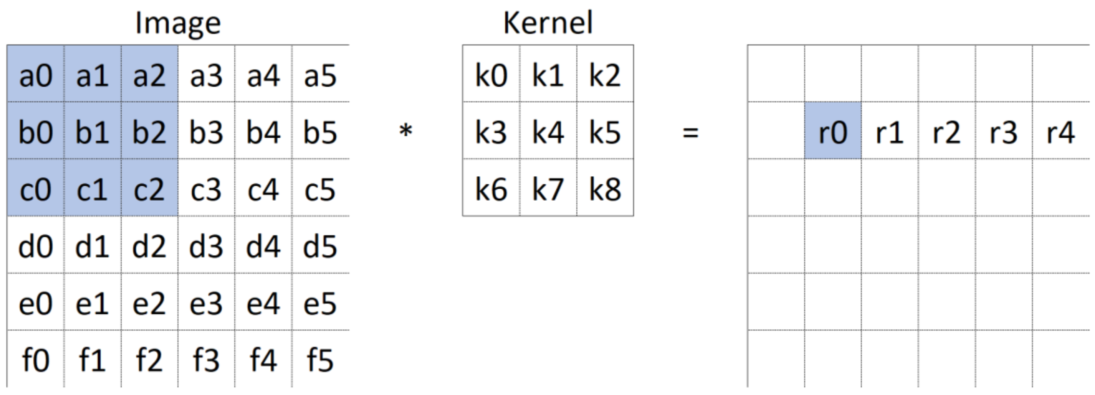


Figure 5 Application of a 3x3 kernel to a section of an image

The value $\mathrm{r0}$ is assigned its value by producing a weighted sum of all the pixels highlighted in blue. The value $\mathrm{r0}$ is calculated using the equation:

$$\begin{aligned} r0 = {} & (a0 * k0) + (a1 * k1) + (a2 * k2) + \\ & (b0 * k3) + (b1 * k4) + (b2 * k5) + \\ & (c0 * k6) + (c1 * k7) + (c2 * k8) \end{aligned}$$

This approach is taken to compute the convolution for all other pixels in the image. The convolution of a pixel is computed using each of its neighbouring pixels, where the current pixel always maps to position $\mathrm{k4}$ in the kernel.

Convolutions like this can be used to apply image processing techniques such as edge detection, image sharpening, image blurring and noise reduction [50].

The output image generated in Figure 5 is smaller than the input image on the left. This is because the pixels on the edge of the input image have no surrounding pixels on all sides. There are several methods that can be used to combat this and produce an output image of the same size as the input image. These methods usually involve artificially extending the size of the input image by inserting new values along the image's edge. Some common approaches to this include duplicating edge values, wrapping around to values on the far edge of the image, and padding with zeros [49].

Throughout this thesis, we assume that all input images to convolutions have been pre-extended so that the output image is the same size as the original input image.

2.7. Vectorization of Convolutions

Convolutions provide opportunities for several of the vectorization strategies discussed in section 2.4. The pseudo-code snippet shown in Figure 6 is for a 3x3 convolution with an input image called `input` and an output image called `output`.

This code snippet can be vectorized using various strategies, including the following three:

1. Inner-loop vectorization
2. Outer-loop vectorization
3. SLP vectorization

For each of these vectorization strategies, we will assume that the input and output images are represented using 32-bit integer values and that we are targeting an architecture with support for both 64-bit and 128-bit vector operations.

```
for i = 0; i < input_height; i += 1
  for j = 0; j < input_width; j += 1
    output[i][j] = input[i-1][j-1] * kernel[0][0] +
                   input[i-1][j]   * kernel[0][1] +
                   input[i-1][j+1] * kernel[0][2] +
                   input[i][j-1]   * kernel[1][0] +
                   input[i][j]     * kernel[1][1] +
                   input[i][j+1]   * kernel[1][2] +
                   input[i+1][j-1] * kernel[2][0] +
                   input[i+1][j]   * kernel[2][1] +
                   input[i+1][j+1] * kernel[2][2]
```

Figure 6 Pseudo-code implementation for a 3x3 convolution

2.7.1. Inner-loop Vectorization

Inner-loop vectorization involves vectorization across the back-edge of the inner-most for loop with iterator `j`. Consider the images shown in Figure 5 to be the `input` and `output` images in our example code snippet. For the first iteration of the outer-loop (`i=0`) and the first iteration of the vectorized inner-loop (`j=0`), there are four values being written to the `output` image. These values are `r0`, `r1`, `r2`, and `r3`.

```
for i = 0; i < input_height; i += 1
  for j = 0; j < input_width; j += 4
    output[i][j..j+3] =
                 input[i-1][j-1..j+2] * kernel[0][0] +
                 input[i-1][j..j+3]   * kernel[0][1] +
                 input[i-1][j+1..j+4] * kernel[0][2] +
                 input[i][j-1..j+2]   * kernel[1][0] +
                 input[i][j..j+3]     * kernel[1][1] +
                 input[i][j+1..j+4]   * kernel[1][2] +
                 input[i+1][j-1..j+2] * kernel[2][0] +
                 input[i+1][j..j+3]   * kernel[2][1] +
                 input[i+1][j+1..j+4] * kernel[2][2]
```

Figure 7 Pseudo-code implementation for a 3x3 convolution with inner-loop vectorization

To compute these values, each of the multiply sub-statements (shown one per line in Figure 6) are vectorized, as shown in Figure 7. For the first statement, the vector `{a0, a1, a2, a3}` is loaded from the `input` image, as indicated by the range `[j-1..j+2]` in Figure 7. For the second statement, the vector `{a1, a2, a3, a4}` is loaded. For the third statement, the vector `{a2, a3, a4, a5}` is loaded. For the fourth statement, the vector `{b0, b1, b2, b3}` is loaded, and so on.

The value loaded for the `kernel` values in each sub-statement is the same for each value of `j` in that sub-statement. This means that the scalar value is loaded from the `kernel`, and it is subsequently copied into every lane of a vector value (via a splat operation) for use in the vectorized loop. In fact, the values loaded from the `kernel` are the same for all iterations of both the inner and outer loops, meaning they are loop invariant for both loops. This makes them eligible for a compiler optimization called loop-invariant code motion (LICM) [51]. This is a common optimization that is present in many modern optimizing compilers like GCC and LLVM.

In the 3x3 convolution example, the use of LICM provides a significant speedup when used in conjunction with inner-loop vectorization since it hoists nine memory load instructions out of the loop body. Thereafter, each of the nine kernel values (`k0` through `k8` in Figure 5) is stored in its own vector register. Because of this transformation, there are nine vector registers live across the back-edge of both loops. This presents a problem for larger convolutions however. With a 5x5 convolution, the number of live vector registers across the loop back-edges is 25. With a 7x7 convolution, this jumps to 49. This presents a serious problem when targeting a processor like SHAVE which only has 32 vector registers [16]. The joint efforts of inner-loop vectorization and LICM force the register allocator into a position in which register spilling and reloading is unavoidable. Inner-loop vectorization becomes unprofitable for larger convolutions because of this register pressure problem.

2.7.2. Outer-loop Vectorization

Outer-loop vectorization is possible for image convolutions. However, convolutions suffer from poor data locality in the outer loop which iterates over each row of the image. When performing outer-loop vectorization, we produce four values together that are each assigned to the same column in consecutive rows of the `output` image, as shown in Figure 8. This means the generated vector values on each iteration of the loop are deconstructed and stored individually as scalars. Outer-loop vectorization also suffers from the same register pressure problem as inner-loop vectorization for larger convolution sizes.

```
for i = 0; i < input_height; i += 4
  for j = 0; j < input_width; j += 1
    output[i..i+3][j] =
                 input[i-1..i+2][j-1] * kernel[0][0] +
                 input[i-1..i+2][j]   * kernel[0][1] +
                 input[i-1..i+2][j+1] * kernel[0][2] +
                 input[i..i+3][j-1]   * kernel[1][0] +
                 input[i..i+3][j]     * kernel[1][1] +
                 input[i..i+3][j+1]   * kernel[1][2] +
                 input[i+1..i+4][j-1] * kernel[2][0] +
                 input[i+1..i+4][j]   * kernel[2][1] +
                 input[i+1..i+4][j+1] * kernel[2][2]
```

Figure 8 Pseudo-code implementation for a 3x3 convolution with outer-loop vectorization

2.7.3. SLP Vectorization

SLP vectorization involves vectorizing only the instructions for one iteration of the inner-for loop. In the case of the 3x3 convolution in Figure 6, there are only three groups of three sub-statements that can be vectorized. The first three multiply sub-statements can be vectorized together, the second group of three together, and the third three together as shown in Figure 9. Although there is a problem with this, since generally vector processors do not operate on vectors with an odd number of elements. As a result, traditional SLP vectorizers will only vectorize the first two sub-statements in each group together, leaving the third alone. The values produced by the generated vector operations also need to be deconstructed to scalars for the accumulation to the final scalar result. For the example in Figure 9, this means that the value `temp` would be generated as a vector of two elements instead of three. The third value for each line (i.e. the value generated by the multiply operation with `kernel[x][2]`) would be performed as a scalar. The vector part and the scalar part would need to be added together before storing to the `output` array. This approach can produce small performance improvements, but they are dwarfed by the inner-loop vectorizer for the 3x3 convolution.

However, SLP vectorization does not suffer from the register pressure problems created by LICM that are present with both inner and outer loop vectorization. For a 5x5 convolution, SLP produces five vector values for the `kernel` values, and five scalar values. This means that there are only five vector registers and five scalar register live across the loop back-edges when combined with LICM.

```
for i = 0; i < input_height; i += 1
  for j = 0; j < input_width; j += 1
    temp = input[i-1][j-1..j+1]  * kernel[0][0..2]
    temp += input[i][j-1..j+1]   * kernel[1][0..2]
    temp += input[i+1][j-1..j+1] * kernel[2][0..2]

    output[i][j] = temp[0] + temp[1] + temp[2]
```

Figure 9 Pseudo-code implementation for a 3x3 convolution with SLP vectorization

On a processor like SHAVE, there is an additional benefit to choosing SLP for vectorization of image convolutions. With SLP, the vector values produced by the grouped multiply instructions are deconstructed for the scalar accumulation sequence. However, on SHAVE there is an efficient vector horizontal sum instruction [52] which can mitigate the deconstruction cost for certain types and convolution sizes.

2.7.4. Proposed Approach

In this thesis, we propose an SLP vectorizer which leverages this lower register pressure to produce efficient vector code for larger sized convolutions. This vectorizer also includes some additional features designed to maximize the vectorization potential in these convolutions.

As mentioned previously in section 2.7.3, one of the reasons SLP vectorization is generally not profitable for convolutions is that the vectorized multiply results are deconstructed to scalars for the output accumulation. One of the features proposed by our vectorizer leverages the associativity and commutativity of the integer `add` operation to reorder operations in the accumulation chain to vectorize a significant portion of it. This is discussed in detail in section 4.2. In Chapter 6, we discuss how this approach compares to using the horizontal sum instructions on SHAVE to deconstruct the multiply results.

There is another aspect of convolutions that can be taken advantage of by our vectorizer. On each iteration of the inner-loop, there is re-use of data from the `input` image from the previous iteration of the loop. With a traditional SLP vectorizer, this data would simply be reloaded on each iteration. We propose a method that utilises a simple form of software pipelining [15] called loop shifting to completely re-use overlapping data between iterations and only load data that is new on each iteration. Loop shifting involves the movement of operations from the beginning of a loop to the end of the loop via the loop back-edge [53]. To maintain the original semantics of the loop, moved operations are also copied into the prologue or header of the loop. This optimization is described in greater detail in section 4.3.

Through these additional optimizations, our SLP vectorizer can produce significant performance improvements in 5x5, 7x7, and 9x9 convolutions without the drawbacks generated by LICM and inner-loop vectorization.

Chapter 3 Superword Level Parallelism

In this chapter, we describe the design and implementation of our Superword Level Parallelism (SLP) vectorizer. This vectorizer serves as a foundation on which the Tentative Loop Unrolling (section 4.1), Associative Chain Reordering (section 4.2) and Loop Shifting (section 4.3) described in the next chapter are built.

This chapter is split into two sections. The first section describes what transformation is being performed and the second part describes how the transformation is achieved.

3.1. Transformation

Superword Level Parallelism (SLP) is a type of vectorization which is performed at a basic block level. It involves grouping multiple scalar operations from a single basic block together to form equivalent vector operations. SLP is described in this section with the aid of two example functions: `add4` (Figure 10) and `maskAndAccumulate4` (Figure 13). A general description of SLP was provided in section 2.4.4. However in this section, we describe our own interpretation of SLP vectorization.

```
void add4 (int * __restrict a, int * __restrict b,
           int * __restrict c) {
   c[0] = a[0] + b[0];
   c[1] = a[1] + b[1];
   c[2] = a[2] + b[2];
   c[3] = a[3] + b[3];
}
```

Figure 10 add4 example function

Scalar operations are grouped together into a single vector instruction in two different ways. The first way involves scalar memory accesses (either load or store instructions, but never both together) which access contiguous addresses and share the same type (e.g. int, float). These can be grouped together into a vector instruction. In `add4`, we can see three examples of this. The memory loads from `a` with indexes zero through three are grouped together into a single vector load instruction as shown in Figure 11 and Figure 12. In both Figures, the loads from `a` are shown in purple, the loads from `b` in blue and stores to `c` in orange. The second way is by grouping instructions together based on other instructions already grouped together. For example, the `add` instructions shown in peach in Figure 11 are grouped together in Figure 12 due to the load and store groups.

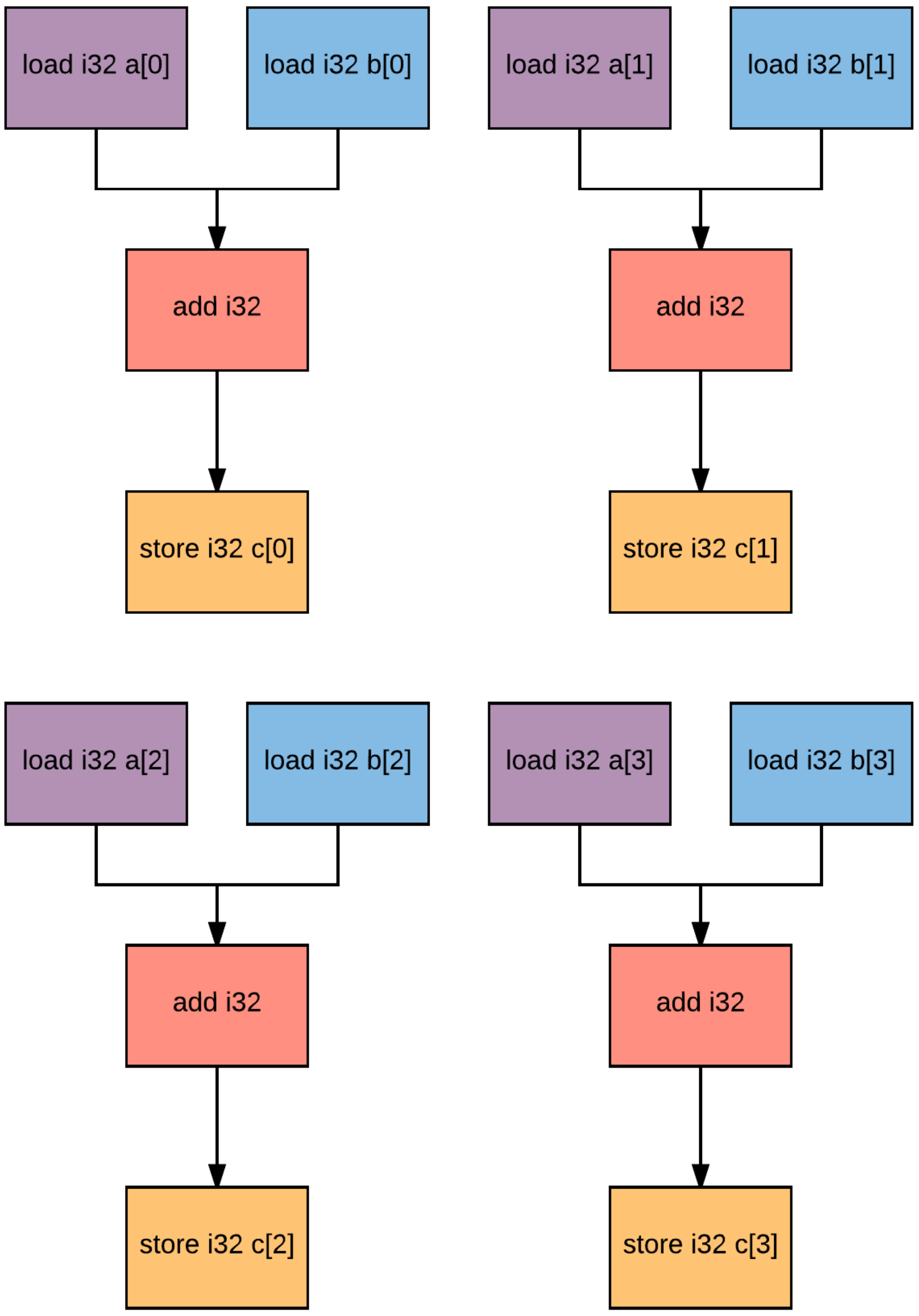


Figure 11 add4 original scalar instruction chains

Vector operations which are built by our vectorizer to replace a group of scalar operations use three different categories of values as input operands:

1. Values produced by vector instructions generated by the vectorizer by grouping other scalar instructions together.
2. Values that are manually built by the vectorizer using a sequence of vector element insert instructions from multiple independent scalar source values when those sources are not vectorizable by our design. These can also be generated using a vector splat instruction.
3. Constant vector values

Similarly, the output of vector instructions can be used in two different ways. Firstly, the output value of a vector instruction may be used by another vector instruction that has been generated by the vectorizer. Alternatively, a group of vector element extract instructions may be generated to deconstruct the output vector into scalar values.

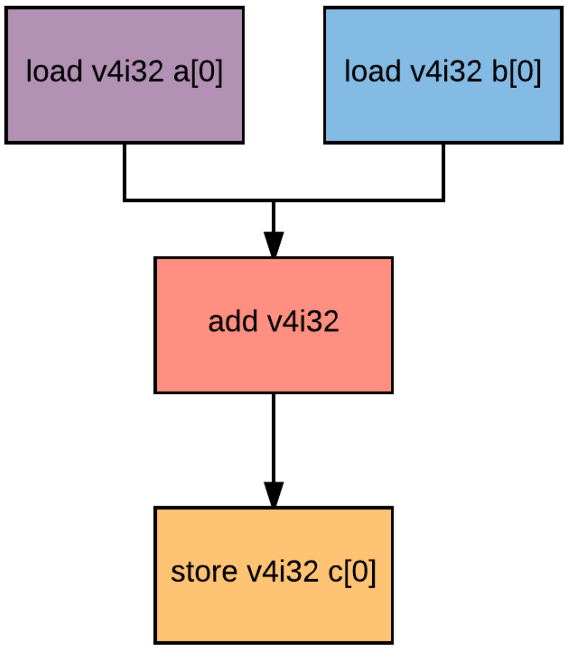


Figure 12 add4 vectorized instruction chain

Our design attempts to build as many vector instructions as possible and minimise the need for vector element insert and vector element extract instructions. For example, in `add4` the four `add` operations are grouped together into a single, equivalent vector `add` instruction. This new vector `add` instruction serves as an intermediary for the vector load and vector store instructions. The output from the two loads are used as the inputs to the `add` instruction, and the output from the `add` instruction is used as the input to the store instruction, as shown in Figure 12.

```
int maskAndAccumulate4 (int * a, int mask) {
  int result = 0;
  result += a[0] & mask;
  result += a[1] & mask;
  result += a[2] & mask;
  result += a[3] & mask;
  return result;
}
```

Figure 13 maskAndAccumulate4 example function

However, this is not always feasible for our design. There are times when constructing or deconstructing a vector value is the only available option. We can see an example of both in the function `maskAndAccumulate4`.

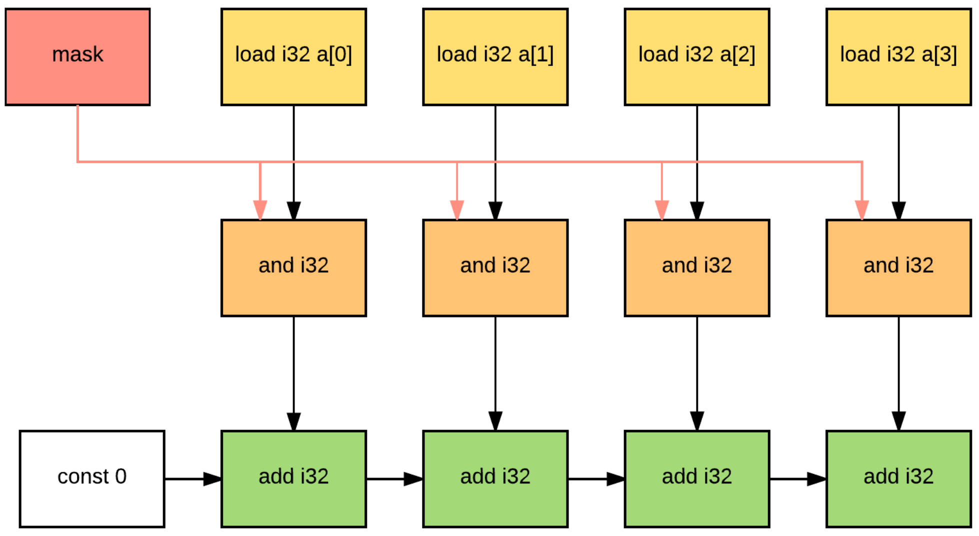


Figure 14 maskAndAccumulate4 original scalar instruction chain

Consider the value `mask` in the function `maskAndAccumulate4`, as shown in Figure 14. The value `mask` is an input for the four logical `and` instructions. The series of load instructions from `a` are usual candidates for vectorization, as are the logical `and` instructions as a result. However, the value `mask` is not since it is a single value with no matching equivalent operations. It is still used as the input to a vector operation as shown in Figure 15. The vectorized logical `and` instructions requires a second vector input. This means that a scalar to vector "splat" instruction is generated. This instruction creates a new vector value, with each element assigned the value of `mask`. This is a special case of the default behaviour which is to create a new vector and generate vector element insert instructions for every element of that vector (category 2 in the list outlined earlier).

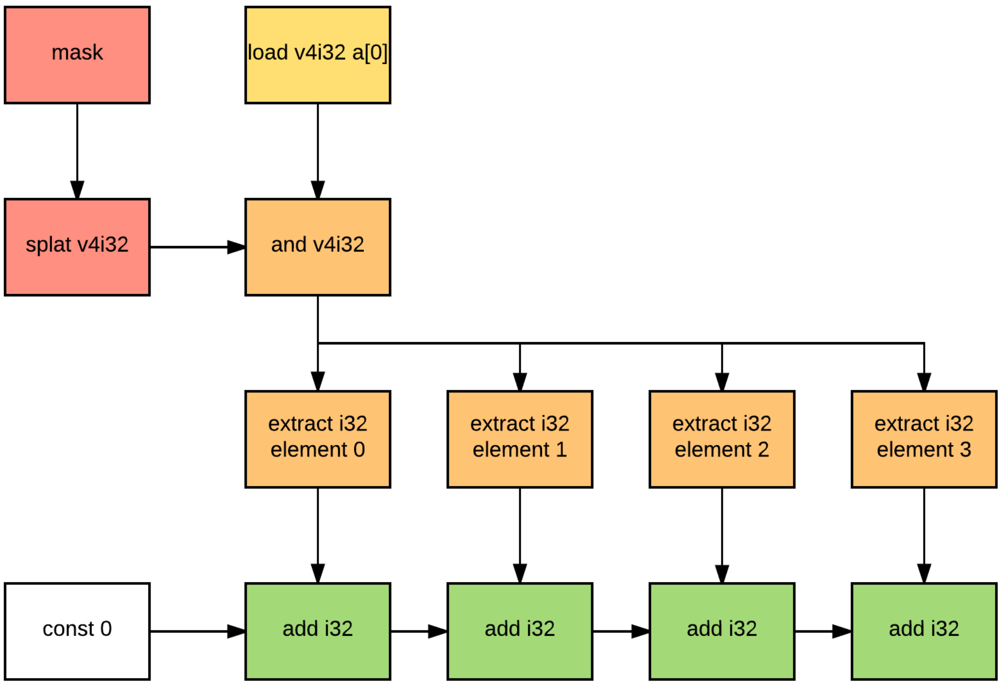


Figure 15 maskAndAccumulate4 vectorized instruction chain

Similarly, if the value of `mask` were constant and not a variable, then the vectorizer could create a new vector constant value, which would not require any additional instructions to be generated.

There is a similar problem present for the output value of the vectorized logical `and` instruction. We are unable to vectorize the series of `add` instructions as there are inter-dependencies between them, and as a result the vector output of the logical `and` instruction has scalar uses. The vectorizer can generate vector element extract instructions for such a case, as shown in Figure 15. This effectively deconstructs the vector to maintain the original scalar use chains of the function. This is not limited to vector instructions at the end of the vectorized chain. There can exist cases where individual elements of vector instructions in the middle of a chain may have additional scalar uses outside of the chain. In these cases, a vector element extract instruction is also generated.

3.2. Design and Implementation

There are five stages involved in our SLP vectorizer design. In order, these stages are:

1. Generate groups of available instructions
2. Generate starting groups
3. Generate vector chains
4. Check memory dependencies
5. Vectorize chains

Each of these stages is described in the five following subsections. Each stage of the vectorization process relies on the output of the previous stage.

```
void addRange (int * __restrict out,
               int * __restrict in,
               unsigned int size) {
  for (unsigned int i = 0, j = size;
       i < size;
       i+=4, ++j) {
    out[i]   = in[i]   + in[j];
    out[i+1] = in[i+1] + in[j];
    out[i+2] = in[i+2] + in[j];
    out[i+3] = in[i+3] + in[j];
  }
}
```

Figure 16 addRange example function

Our SLP vectorizer takes a single basic block as input. This block does not necessarily need to be the body of a loop; the vectorizer is run on every basic block in a function. Every basic block in a function is vectorized independently of all other blocks in that function. Throughout this section, we will use the function `addRange` shown in Figure 16 as an example to aid the description of each stage of the vectorizer.

```
define void @addRange(i32* noalias nocapture %out,
                      i32* noalias nocapture readonly %in,
                      i32 %size) local_unnamed_addr #0 {
; ...
for.body:   ; preds = %for.body.preheader, %for.body
  %j = phi i32 [ %incj, %for.body ],
               [ %size, %for.body.preheader ]
  %i = phi i32 [ %inci, %for.body ],
               [ 0, %for.body.preheader]
  %idxl0 = getelementptr inbounds i32, i32* %in, i32 %i
  %load0 = load i32, i32* %idxl0, align 4, !tbaa !1
  %idxl1 = getelementptr inbounds i32, i32* %in, i32 %j
  %load1 = load i32, i32* %idxl1, align 4, !tbaa !1
  %add0 = add nsw i32 %load1, %load0
  %idxs0 = getelementptr inbounds i32, i32* %out, i32 %i
  store i32 %add0, i32* %idxs0, align 4, !tbaa !1
  %or1 = or i32 %i, 1
  %idxl2 = getelementptr inbounds i32, i32* %in, i32 %or1
  %load2 = load i32, i32* %idxl2, align 4, !tbaa !1
  %add1 = add nsw i32 %load1, %load2
  %idxs1 = getelementptr inbounds i32, i32* %out, i32 %or1
  store i32 %add1, i32* %idxs1, align 4, !tbaa !1
  %or2 = or i32 %i, 2
  %idxl3 = getelementptr inbounds i32, i32* %in, i32 %or2
  %load3 = load i32, i32* %idxl3, align 4, !tbaa !1
  %add2 = add nsw i32 %load1, %load3
  %idxs2 = getelementptr inbounds i32, i32* %out, i32 %or2
  store i32 %add2, i32* %idxs2, align 4, !tbaa !1
  %or3 = or i32 %i, 3
  %idxl4 = getelementptr inbounds i32, i32* %in, i32 %or3
  %load4 = load i32, i32* %idxl4, align 4, !tbaa !1
  %add3 = add nsw i32 %load1, %load4
  %idxs3 = getelementptr inbounds i32, i32* %out, i32 %or3
  store i32 %add3, i32* %idxs3, align 4, !tbaa !1
  %inci = add i32 %i, 4
  %incj = add i32 %j, 1
  %cmp = icmp ult i32 %inci, %size
  br i1 %cmp, label %for.body,
              label %for.cond.cleanup.loopexit
}
```

Figure 17 addRange for loop body as LLVM-IR

The input array `in` is split logically into two sections. The first section contains `size` elements which are indexed using the iterator `i`. The second section contains `size/4` elements which are indexed using the iterator `j`. The output array `out` contains `size` elements and is also indexed using the iterator `i`.

The LLVM-IR for the main loop body is shown in Figure 17. The other basic blocks in the function are not relevant for this section as they are not vectorizable. The basic block with label `%for.body.preheader` is the entry block for the loop, and the basic block with label `%for.cond.cleanup.loopexit` is the exit block for the loop.

3.2.1. Generate Groups of Available Instructions

The first step taken by the vectorizer is to generate groups of instructions that exist in the basic block we are vectorizing. Instructions are grouped together based on their opcode. Each of these groups is a listing of all instructions with the corresponding opcode that are available for vectorization. When an instruction has been vectorized, it is removed from its corresponding list. We do this so that when we are generating the vectorizable instruction chains in section 3.2.3 we do not re-use instructions (i.e. to ensure each instruction is only vectorized once). Most instructions in the input basic block are placed into Groups of Available Instructions. We generate Groups of Available Instructions only for opcodes that we consider to be "vectorizable".

In no particular order, the list of vectorizable instructions is:

- memory load instructions
- memory store instructions
- "binary operator" instructions, e.g. `add`, `sub`, `mul`, `and`, `or`, `xor`
- type cast instructions
- select instructions
- compare instructions
- a subset of compiler intrinsics, e.g. `llvm.ctpop` which counts the number of bits set to one in a value

As the name suggests, only instructions which are a member of one of these Groups of Available Instructions are available for vectorization.

```
Group of Available add Instructions:
  %add0 = add nsw i32 %load1, %load0
  %add1 = add nsw i32 %load1, %load2
  %add2 = add nsw i32 %load1, %load3
  %add3 = add nsw i32 %load1, %load4
  %inci = add i32 %i, 4
  %incj = add i32 %j, 1

Group of Available or Instructions:
  %or1 = or i32 %i, 1
  %or2 = or i32 %i, 2
  %or3 = or i32 %i, 3

Group of Available load Instructions:
  %load0 = load i32, i32* %idxl0, align 4, !tbaa !1
  %load1 = load i32, i32* %idxl1, align 4, !tbaa !1
  %load2 = load i32, i32* %idxl2, align 4, !tbaa !1
  %load3 = load i32, i32* %idxl3, align 4, !tbaa !1
  %load4 = load i32, i32* %idxl4, align 4, !tbaa !1

Group of Available store Instructions:
  store i32 %add0, i32* %idxs0, align 4, !tbaa !1
  store i32 %add1, i32* %idxs1, align 4, !tbaa !1
  store i32 %add2, i32* %idxs2, align 4, !tbaa !1
  store i32 %add3, i32* %idxs3, align 4, !tbaa !1

Group of Available icmp Instructions:
  %cmp = icmp ult i32 %add21, %size
```

Figure 18 Groups of Available Instructions generated for the basic block %for.body in the function addRange

In our example basic block `%for.loop`, Groups of Available Instructions are generated for the opcodes `add`, `or`, `load`, `store`, and `icmp` as shown in Figure 18. There are some notable omissions from these groups.

The function arguments `%out` and `%in` are missing as they are not instructions, they are values.

None of the `getelementptr` instructions are included either. These instructions are used in LLVM to compute addresses for memory accessing instructions (i.e. loads and stores) as described in section 2.5. These instructions are a special case since address computation will be completely regenerated for memory accesses which are vectorized later. They are used in section 3.2.2 when grouping load and store instructions together. They are not vectorizable, but they enable the vectorization of these other instructions.

The SSA PHI instructions at the start of the basic block are also not included. These instructions provide a mechanism for assigning a different value to a variable depending on which basic block branched to this basic block at runtime. For example, the second `phi` instruction which produces the variable `%i` is assigned the value `%inci` when the loop back-edge is taken at runtime, and the constant value zero when the loop entry block `%for.body.preheader` branches into the loop. The value `%inci` is produced by the `add` instruction near the end of the loop body. This is the equivalent of the initialization and update statements for the iterator `i` in the original C function. Since these instructions use values from outside of the current basic block, they go beyond the scope of the single block vectorization performed by our base SLP vectorizer implementation. Therefore, they are excluded from the Groups of Available Instructions.

Finally, the `br` instruction is also missing. This is LLVM's branch instruction. Much like `ret` (LLVM's equivalent of the C keyword `return`), `br` cannot be vectorized as it is a basic block terminator instruction which affects the control flow of the function. It is not possible to vectorize this kind of instruction.

3.2.2. Generate Starting Groups

A subset of the Groups of Available Instructions is used as the starting point for building vectorizable chains. The Groups of Available Instructions used are the load and store groups. In our example, we will focus on the load group here.

For each instruction in the Group of Available Instructions (e.g. all the load instructions in the basic block), the vectorizer inserts them into a subgroup based on the base pointer for each memory access. This means that each subgroup contains instructions with either variable or constant offsets from the same base address pointer. In our example, all the load instructions use the same base pointer `%in` so they are all inserted into the same subgroup as shown in Figure 19.

```
Subgroup with base pointer in:
  %load0 = load i32, i32* %idxl0, align 4, !tbaa !1
  %load1 = load i32, i32* %idxl1, align 4, !tbaa !1
  %load2 = load i32, i32* %idxl2, align 4, !tbaa !1
  %load3 = load i32, i32* %idxl3, align 4, !tbaa !1
  %load4 = load i32, i32* %idxl4, align 4, !tbaa !1
```

Figure 19 Base pointer subgroups for the load instructions in %for.body

Instructions are split into these subgroups by inspecting the pointer operand to the load (or store) instruction. If the pointer operand is not a `getelementptr` instruction, then the value of the operand is used as the base pointer. If the pointer operand is a `getelementptr` instruction, then the first input operand of that instruction is used as the base pointer.

These subgroups are then split up again into smaller subgroups. Each of these smaller subgroups contain memory accessing instructions which have a common base pointer and a common variable base index. A common example of a common variable base index is a loop iterator. Instructions that have no variable base index are inserted into a subgroup of their own. In Figure 20, we can see there are now two subgroups for the load instructions in the basic block `%for.body`. Both subgroups share the base pointer `%in`, but one group is indexed using the iterator `%i`, and the other using the iterator `%j`.

```
Subgroup with base pointer in and variable index i:
  %load0 = load i32, i32* %idxl0, align 4, !tbaa !1
  %load2 = load i32, i32* %idxl2, align 4, !tbaa !1
  %load3 = load i32, i32* %idxl3, align 4, !tbaa !1
  %load4 = load i32, i32* %idxl4, align 4, !tbaa !1

Subgroup with base pointer in and variable index j:
  %load1 = load i32, i32* %idxl1, align 4, !tbaa !1
```

Figure 20 Base pointer and variable index subgroups for the load instructions in %for.body

These new subgroups are generated by inspecting the second input operand of the `getelementptr` instruction associated with each load instruction. If the value of the second input operand is produced by a logical `or` instruction or an `add` instruction with a constant second input operand, then the first input operand of that `or`/`add` is used as the base variable index. Otherwise, the second input operand of the `getelementptr` instruction is used as the base index. We only process logical `or` instructions in this way due to a transformation that is performed by another pass in LLVM. It replaces `add` instructions that have a constant second input operand with a logical `or` instruction if it can prove the least significant bits of the first operand are guaranteed to be zero. We describe this in more detail later in this section.

In our example, consider the subgroup with common variable base index `%i`. The `getelementptr` instruction for `%load0` has a first input operand of `%i`. However, the first input operands for `%load2`, `%load3` and `%load4` are all logical `or` instructions with constant second input operands. In each of these cases, the first input operand for the `or` is `%i`.

The elements of these subgroups are finally either removed or reordered based on their constant offset from the common base index. This is done by inspecting the second input operand of the `or`/`add` instruction associated with each `getelementptr` instruction. If a load instruction doesn't have a corresponding `getelementptr` instruction or if the second input operand of the `getelementptr` instruction is not an `or`/`add` instruction, then the constant offset is zero.

In our example, the `getelementptr` instruction associated with the load that produces the value `%load0` has a second input operand that is the variable base index. Therefore, the constant offset for `%load0` is zero. The second input operand for the logical `or` associated with `%load2` is one, therefore the constant offset is one. For `%load3`, it is two and for `%load4`, it is three.

We perform this process with logical `or` instructions only when it is provably safe to do so. In our example, the iterator `%i` is initialized by the `phi` instruction on first entry to the loop to zero. The increment instruction for the iterator, `%inci` adds constant four to the iterator on each iteration of the loop. Since the least significant bits of the iterator are initialized to `0000b` and the increment instruction only ever adds `0100b`, the two least significant bits are always `00b`. The values listed in Figure 21 are therefore guaranteed to be always consecutive since the constant values used in the logical `or` operations only modify these two least significant bits.

Value	Least significant bits
`%i`	xx00
`%or1 = or i32 %i, 1`	xx01
`%or2 = or i32 %i, 2`	xx10
`%or3 = or i32 %i, 3`	xx11

Figure 21 Least significant bits for the %i offset instructions

Once this has been completed, we are left with groups of memory accessing instructions which share a common type and access consecutive memory locations.

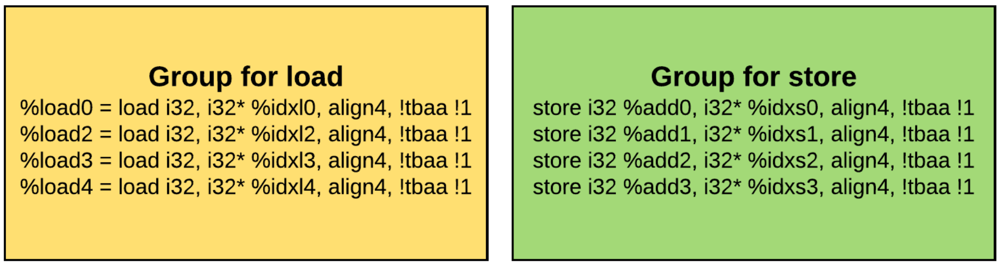
We require that these groups contain at least two elements each. If a group does not contain at least two elements, then it is discarded. In our example, the first group for iterator `%i` contains four instructions so it can be used as a starting point for vectorization. However, the group for iterator `%j` only contains one instruction, so it must be discarded.

If the size of a group exceeds the natural vectorization factor for the contained type, it may be split up into multiple groups. The natural vectorization factor for a type is the number of values of that type that need to be brought together to fill all lanes of a vector on the target architecture. For example, if a group of `i32` loads contains 16 instructions and the natural vectorization factor for `i32` on our target architecture is four, then this group will be split up into four separate groups each containing four instructions. In our example, we assume that the target architecture has a 128-bit vector unit, so this step is not necessary for the starting group which contains four 32-bit instructions. We do this to facilitate the Association Chain Reordering optimization described in section 4.2. This optimization requires multiple independent vector nodes as inputs to chains of associative instructions to vectorize them. It does not work with a single, large vector value. This is described in more detail in that section.

This process is also completed in our example for the Group of Available store Instructions. After this process is complete, there are two starting groups. One is the load group described above, and the other is a group containing the store instructions for the values `%add0`, `%add1`, `%add2`, `%add3`. These groups are used as the starting point for building chains of vectorizable instruction groups.

3.2.3. Generate Vector Chains

Vectorizable sets of instructions are grouped together and linked in definition-use chains. A chain is represented as a directed acyclic graph (DAG). Each vertex, or node in the graph represents a set of vectorizable instructions. Each edge in the graph represents a definition-use relationship between two nodes. If an edge exists in a graph going from node A to node B, then node A produces a value which is consumed or used by node B. This is the case for all instructions in both nodes (i.e. there is an all-to-all relationship between nodes connected by an edge). The instructions contained in a node are strictly ordered. The position of an instruction in a node corresponds to its lane position in the vector. Throughout this section, when we talk about nodes, edges and DAGs, we are referring to this specific implementation.


Group for load

```
%load0 = load i32, i32* %idxl0, align4, !tbaa !1
%load2 = load i32, i32* %idxl2, align4, !tbaa !1
%load3 = load i32, i32* %idxl3, align4, !tbaa !1
%load4 = load i32, i32* %idxl4, align4, !tbaa !1
```

Group for store

```
store i32 %add0, i32* %idxs0, align4, !tbaa !1
store i32 %add1, i32* %idxs1, align4, !tbaa !1
store i32 %add2, i32* %idxs2, align4, !tbaa !1
store i32 %add3, i32* %idxs3, align4, !tbaa !1
```

Figure 22 The starting nodes for the DAG of vectorizable nodes in %for.body

Each basic block can contain one or more DAGs of vectorizable nodes. The starting groups are the first nodes which are placed into a DAG of their own. Continuing with our example basic block `%for.body` from Figure 17 on page 32, Figure 22 shows the two starting points for the DAGs. Both nodes are placed into their own DAG to begin with and act as the root nodes for their respective DAGs. If a def-use chain exists between these nodes, an edge will not be added between the nodes at this point. This will be done later in the process. After all nodes have been built, a post-pass is performed over each DAG to create edges between nodes that should be linked but which were built independently of each other.

When building the DAGs in this section, the Groups of Available Instructions remaining for the current basic block are those which were not used to build the starting groups. For our example, of the groups shown in Figure 18 on page 34, the remaining Groups of Available Instructions are shown in Figure 23.

```
Group of Available add Instructions:
  %add0 = add nsw i32 %load1, %load0
  %add1 = add nsw i32 %load1, %load2
  %add2 = add nsw i32 %load1, %load3
  %add3 = add nsw i32 %load1, %load4
  %inci = add i32 %i, 4
  %incj = add i32 %j, 1

Group of Available or Instructions:
  %or1 = or i32 %i, 1
  %or2 = or i32 %i, 2
  %or3 = or i32 %i, 3

Group of Available icmp Instructions:
  %cmp = icmp ult i32 %add21, %size
```

Figure 23 Groups of Available Instructions for vectorization in %for.body

Each DAG is built node by node in two directions. First, we consider the users of the instructions in a node (i.e. other instructions in the basic block which use the value produced by a node instruction).

Starting with the first instruction in the node, we find another instruction in the basic block which uses its produced value. If this user instruction is in a Group of Available Instructions, then we can use it as a starting point for building a new node. In `%for.body`, we start by looking at the first instruction in the load group shown in Figure 22 in yellow. The value `%load0` is used by the instruction "`%add0 = add nsw i32 %load1, %load0`". This instruction is part of the Group of Available `add` Instructions, so it can be used as the starting point for building a new node.

We search the corresponding Group of Available Instructions for instructions which are users of each other instruction in the node. If we find a matching user instruction from the Group of Available Instructions for each node instruction, then we can build a new node with these user instructions. In this case, the Group of Available Instructions is the `add` group and we are looking for instructions which use the values `%load2`, `%load3`, and `%load4` from the load node. The following instructions all match this requirement:

```
%add1 = add nsw i32 %load1, %load2
%add2 = add nsw i32 %load1, %load3
%add3 = add nsw i32 %load1, %load4
```

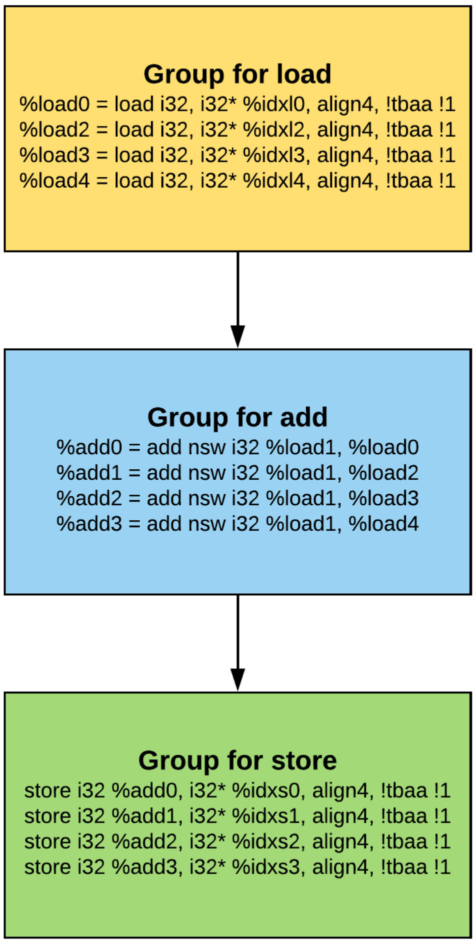
The value produced by each instruction in the original node must be the same operand number in their corresponding user in the new node (e.g. they must all be operand zero, one, etc.). In our example, each of the values produced by the original node instructions (shown in yellow in Figure 22) are used as the second input operand of their corresponding `add` instruction. We do this to ensure that all lanes for a single input operand are produced by the same, single vector instruction.

We use these instructions to construct a new node in the DAG. The new node is added as a successor to the original node, and the original node is added as a predecessor of the new node.

This approach is used to build the graph from all types of nodes, except for nodes that contain store instructions. These instructions do not produce a value as output, so they cannot be used by other instructions. Our vectorizer only considers register dependencies (value def-use dependencies) when building edges between nodes.

Next, we look at each of the operands to the first instruction in the node. Similar to above, we check whether the operand value is produced by an instruction that is in a Group of Available Instructions. If it is, then we look at the corresponding operand in each other instruction in the node for an instruction in the same Group of Available Instructions (e.g. all operands zero). If we find a matching use instruction for every instruction in the node, then we can build a new node using these instructions. The new node is added as a predecessor of the existing node, and the existing node is added as a successor of the new node.

This approach is used to build the graph from all types of nodes, except for nodes that contain load instructions. The values used by load instructions have already been examined during the starting group generation step in section 3.2.2.



```
%load0 = load i32, i32* %idxl0, align4, !tbaa !1
%load2 = load i32, i32* %idxl2, align4, !tbaa !1
%load3 = load i32, i32* %idxl3, align4, !tbaa !1
%load4 = load i32, i32* %idxl4, align4, !tbaa !1
```

```
%add0 = add nsw i32 %load1, %load0
%add1 = add nsw i32 %load1, %load2
%add2 = add nsw i32 %load1, %load3
%add3 = add nsw i32 %load1, %load4
```

```
store i32 %add0, i32* %idxs0, align4, !tbaa !1
store i32 %add1, i32* %idxs1, align4, !tbaa !1
store i32 %add2, i32* %idxs2, align4, !tbaa !1
store i32 %add3, i32* %idxs3, align4, !tbaa !1
```

Figure 24 Generated DAG for %for.body

In our example, there are no nodes which can take advantage of this approach. The node of `add` instructions which we built uses `%load1` as the other input operand in every instruction. This value is not in a Group of Available Instructions, so we cannot use it as a starting point for building a new node. Similarly, for the store node, the used values `%add0`, `%add1`, `%add2`, and `%add3` have already been added to a node for vectorization. This means they have been removed from their Group of Available Instructions and are no longer available.

When all instructions for a node have been gathered together, we perform a dependency check between the instructions before constructing the node. If any of the instructions we have gathered together depend on the output of another gathered instruction, then the node is not constructed, and the instructions are placed back into their Group of Available Instructions.

This process is completed for every node in the graph. Once it is complete, a final pass over every node is performed to add edges between nodes which were built independently of each other, but which match the criteria above for being related. The generated DAG for `%for.body` is shown in Figure 24. This final pass over every node creates the edge between the `add` node to the store node since the store node uses the values produced by the `add` node.

3.2.4. Check Memory Dependencies

Once all chains have been generated for the basic block, each node in the DAG must pass the memory check step before any chain can be vectorized. This step involves checking for memory alias dependencies between every memory access in a node and every other memory access in the basic block.

If a memory store instruction in a node may alias any other store or load instruction in the basic block, then vectorization is considered unsafe for every chain in this block. Similarly, if any load instruction in a node may alias any store instruction in the basic block, then vectorization is considered unsafe. Every DAG must be discarded since the vectorization of one DAG may cause instructions not in the DAG to be reordered. This could potentially lead to aliasing memory accesses to be reordered in relation to each other. This restriction does prevent code which reads from a memory location and writes another value back to it in the same statement from being vectorized. This is currently a limitation of our implementation that can be removed in future work.

In the DAG generated for the basic block labelled `%for.body` as shown in Figure 24, there are two nodes of memory accessing instructions. Starting with the load node, each instruction in the node is compared to every store instruction in the basic block. There are four store instructions in `%for.body`. Each store instruction uses `out` as a base pointer which was declared in the original C code with the `__restrict` keyword, which we described in section 2.5. Each load instruction uses `in` as a base pointer which was also declared with the `__restrict` keyword. Therefore, the load instructions in the node are guaranteed to not alias any of the store instructions in the basic block because they use discrete pointers each declared with `__restrict`. The same can be said for the store instructions in the store node and the five load instructions in the basic block.

Our vectorizer implementation uses LLVM's Memory Alias Analysis pass for this memory dependency checking step. This pass provides a function that takes two pointers as arguments and returns one of four values, `NoAlias`, `MayAlias`, `PartialAlias` or `MustAlias`. If the function returns anything other than `NoAlias` then to avoid performing an unsafe transformation, we conservatively assume that the pointers might alias.

If there are no memory aliasing problems, then each of the DAGs may be vectorized.

3.2.5. Vectorize Chains

The node generator described in section 3.2.3 does not enforce any rules on the size of the nodes it is generating beyond ensuring that all nodes in a chain are the same size. We use this restriction to ensure that all lanes of a vectorizable node have a corresponding input operand from one of the node's predecessors. Because of this restriction, the size of nodes will often not match the natural vectorization factor for the target architecture. The vectorizer must take this into account. Before generating any vector instructions, if necessary the vectorizer resizes all nodes down to the closest natural vector size for the contained type for the target. This means removing the last `N-VF` instructions from each node, where `N` is the node size and `VF` is the natural vectorization size chosen for the target. In our example, this is not necessary since each node contains four instructions each which is the natural vector size for our target. An alternative approach is to resize nodes upwards and ignore values in unused lanes. We use this approach in our loop shifting optimization described in section 4.3.

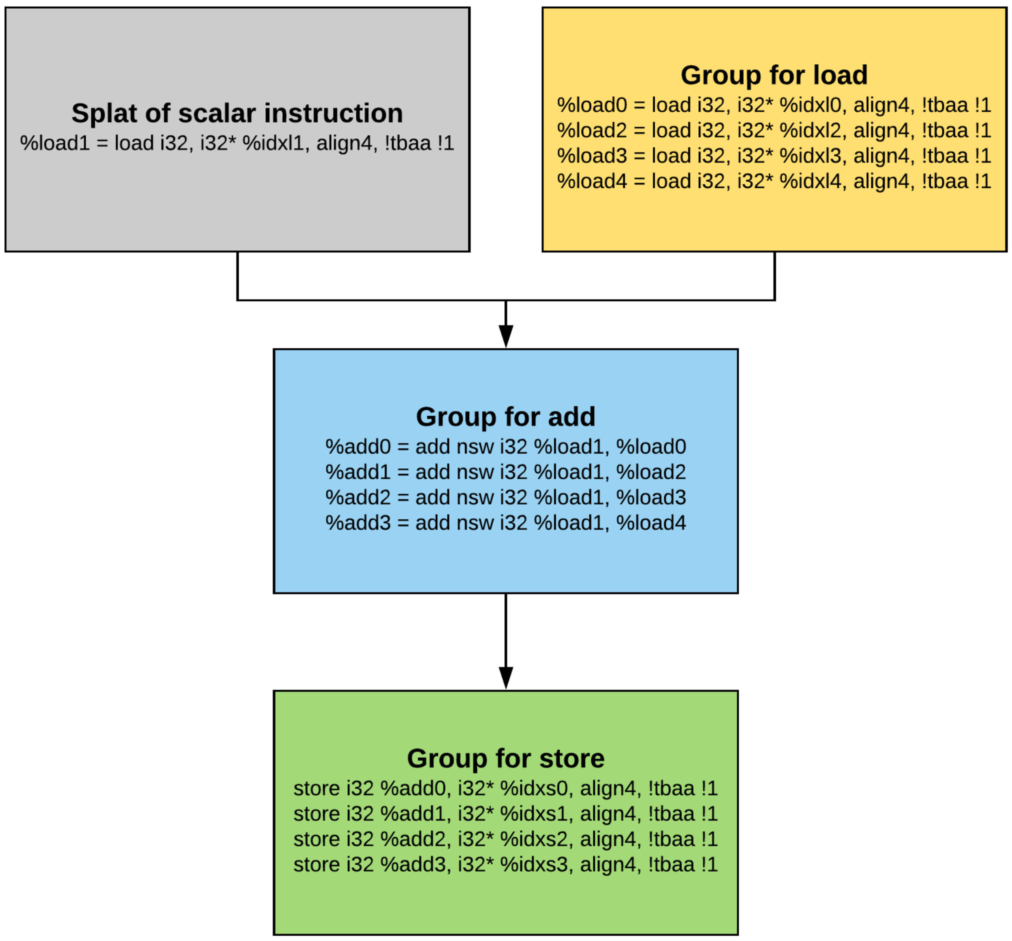
Each chain of instructions is vectorized independently, one after the other. The nodes in each chain are vectorized in a top-down breadth-first fashion, starting with the root nodes of the chain. A root node is any node which does not have any predecessor nodes in the chain. A node is only vectorized once all its predecessors have been vectorized. In `%for.body`, we start vectorizing with the load node since it has no predecessors. We then move on to the `add` node and finish off with the store node.

If any input operands for a node are not produced by a node in the chain, then a new vector node must be generated for that input operand through a legalization process. Depending on the type of input operands, the legalization can take one of three forms

1. If this operand for all instructions in the node are constant, then a new constant vector value is created and used as the corresponding operand in the vector instruction.
2. If this operand for all instructions in the node is the same non-constant value, then a vector splat instruction is created for the vector operand.
3. If there is no constant pattern or single variable operand, then a new vector value is created and the operand value for each instruction is inserted into the new vector one by one.

In `%for.body`, we must create a new input vector operand for the `add` node shown in blue in Figure 24. The second input operand value for each instruction is generated by the load node, however the first input operand has no corresponding vector value. Every first input operand in the node is the same value `%load1`. This means we need to use method 2 for generating the vector node. A vector splat node is generated as shown in Figure 25.

LLVM-IR does not have a vector splat instruction so we must construct the same functionality using a vector element insert instruction and vector shuffle as shown in Figure 26 by the instructions that generate the values `%splatinsert` and `%splat`. The value `%load1` is first inserted into element zero of an undefined vector value. A vector shuffle instruction then replicates the value in element zero into every other element of the vector. This effectively splats the value `%load1` into a new vector.



Splat of scalar instruction
%load1 = load i32, i32* %idxl1, align4, !tbaa !1
Group for load
%load0 = load i32, i32* %idxl0, align4, !tbaa !1
%load2 = load i32, i32* %idxl2, align4, !tbaa !1
%load3 = load i32, i32* %idxl3, align4, !tbaa !1
%load4 = load i32, i32* %idxl4, align4, !tbaa !1
Group for add
%add0 = add nsw i32 %load1, %load0
%add1 = add nsw i32 %load1, %load2
%add2 = add nsw i32 %load1, %load3
%add3 = add nsw i32 %load1, %load4
Group for store
store i32 %add0, i32* %idxs0, align4, !tbaa !1
store i32 %add1, i32* %idxs1, align4, !tbaa !1
store i32 %add2, i32* %idxs2, align4, !tbaa !1
store i32 %add3, i32* %idxs3, align4, !tbaa !1

Figure 25 Final DAG for %for.body

Once all the input operands have a corresponding node in the graph, the node is vectorized by creating a new vector instruction of the same opcode as the scalar instructions with all predecessor nodes as the input operands.

Once all nodes in the chain have been vectorized, we need to tidy up any references to values which were previously produced by scalar instructions which have now been vectorized. This involves creating vector element extract instructions for each reference and replacing the old value with this new extracted value. In `%for.body` from Figure 17 on page 32, there are no references to any of the vectorized values, so this step is not necessary.

Finally, all the vectorized scalar instructions are deleted and the vectorizer moves on to the next DAG.

```
define void @addRange(i32* noalias nocapture %out,
                      i32* noalias nocapture readonly %in,
                      i32 %size) local_unnamed_addr #0 {
; ...
for.body:    ; preds = %for.body.preheader, %for.body
  %j = phi i32 [ %incj, %for.body ],
               [ %size, %for.body.preheader ]
  %i = phi i32 [ %inci, %for.body ],
               [ 0, %for.body.preheader ]
  %idx0 = getelementptr inbounds i32, i32* %in, i32 %i
  %bitcast0 = bitcast i32* %idx0 to <4 x i32>*
  %loadv0 = load <4 x i32>, <4 x i32>* %bitcast0, align 4
  %idx1 = getelementptr inbounds i32, i32* %in, i32 %j
  %load1 = load i32, i32* %idx1, align 4, !tbaa !1
  %splatinsert = insertelement <4 x i32> undef,
                               i32 %load1, i32 0
  %splat = shufflevector <4 x i32> %splatinsert,
                         <4 x i32> undef,
                         <4 x i32> zeroinitializer
  %addv = add <4 x i32> %splat, %loadv0
  %idx2 = getelementptr inbounds i32, i32* %out, i32 %i
  %bitcast1 = bitcast i32* %idx2 to <4 x i32>*
  store <4 x i32> %addv, <4 x i32>* %bitcast1, align 4
```

Figure 26 Vectorized for loop body for addRange

In `%for.body`, the original instructions in the load node are replaced by the instruction sequence

```
%idx0 = getelementptr inbounds i32, i32* %in, i32 %i
%bitcast0 = bitcast i32* %idx0 to <4 x i32>*
%loadv0 = load <4 x i32>, <4 x i32>* %bitcast0, align 4
```

This sequence takes the pointer of the first load instruction in the node, bitcasts it to the required type (`v4i32 *`) and then performs the vector load using this pointer. This same approach is used for the store node as well.

The input basic block to our SLP vectorizer has now been successfully vectorized. In the following chapter we will describe a set of optimization techniques which build upon this SLP vectorizer.

Chapter 4 Additional Vectorizer Optimizations

In this chapter, we describe three additional optimizations which are built on top of the SLP vectorizer described in Chapter 3. These additional optimizations include Tentative Loop Unrolling (section 4.1), Associative Chain Reordering (section 4.2) and Loop Shifting (section 4.3).

Each section is split into two parts. The first part of each section describes the transformation that is being performed and the second part describes how the transformation is achieved.

4.1. Tentative Loop Unrolling

4.1.1. Transformation

Loop unrolling is a technique which can be used to create SLP vectorization opportunities where they otherwise do not exist. Consider the example function `multiply` that is shown in Figure 27. The for loop with iterator `i` and bound `size` contains only three memory accessing operations and one multiply operation. None of the memory accessing instructions are to consecutive regions in memory. This loop is not a candidate for optimization through SLP because of this. However, by unrolling the loop we can manufacture an opportunity for SLP vectorization.

```
void multiply(int * __restrict out, int * __restrict inA,
              int * __restrict inB, unsigned int size) {
  for (unsigned int i = 0; i < size; ++i)
    out[i] = inA[i] * inB[i];
}
```

Figure 27 multiply example function

The control flow graph (CFG) for this function is show in Figure 28. It contains only three basic blocks. The blocks shown in blue are the entry and exit blocks for the function. The block shown in green is the for loop which we are interested in. The edge from the entry block to the exit block is taken when the value of `size` is zero.

Loop unrolling involves duplicating the body of a loop N times and adjusting the iterator increment on each iteration of the loop. For our example function multiply, there is a manually unrolled version in Figure 29. In this case, N is four. The first for loop in this function is the unrolled version of the original loop. The loop iterator update statement has been changed from ++i to i+=4 to allow for the unrolling. The statement which read elements from inA and inB, multiplies the values together and stores the result to an element of out has been duplicated four times in the unrolled loop body. The original loop performed this action for all elements of inA, inB and out between indexes 0 and size. The unrolled loop body must preserve the semantics of the original loop. To allow this, the uses of i in the original statement are updated for each copy in the unrolled loop body. The first copy of the statement has uses of i replaced with i+1, the second copy with i+2, and the third copy with i+3. This ensures that when we increment i by four on each iteration of the unrolled loop, no elements of inA, inB or out are skipped.

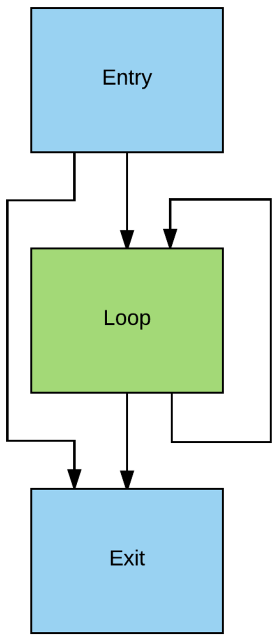


Figure 28 Control flow graph for the example function multiply

There is an additional concern when unrolling a loop like this. When the upper loop bound size is not a multiple of the unroll factor N, then we must include an additional loop to perform our loop action for the remaining elements of the arrays.

For example, if the value of `size` in the function `multiply` is 19 then we will perform four iterations of the unrolled loop. This only accounts for the first 16 elements of the arrays. We must perform the multiply and store for the remaining three elements for this transformation to be considered safe. This is performed by the additional for loop at the end of the function. The loop executes three times, for iterator values 16, 17 and 18. The unrolled version of the loop therefore completes the exact same actions as the original loop.

```
void multiply(int * __restrict out, int * __restrict inA,
              int * __restrict inB, unsigned int size) {
  unsigned int i = 0;
  if (size >= 4) {
    for (; i < (size & 0xFFFFFFFC); i+=4) {
      out[i+0] = inA[i+0] * inB[i+0];
      out[i+1] = inA[i+1] * inB[i+1];
      out[i+2] = inA[i+2] * inB[i+2];
      out[i+3] = inA[i+3] * inB[i+3];
    }
  }

  for (; i < size; ++i)
    out[i] = inA[i] * inB[i];
}
```

Figure 29 multiply example function manually unrolled by 4

In this example, the upper loop bound for the unrolled loop is "`size & 0xFFFFFFFC`". This can be read as the largest multiple of four that is less-than or equal to the value of `size`. However, the starting value of the iterator of the for loop may not always be zero and the loop iterator is not always incremented by one on each iteration. We must account for this by using the following formula to calculate the new upper bound for any loop.

$$\frac{size - startingI}{stride} \times stride$$

Where `stride` is the unroll factor `N` multiplied by the original loop stride, `startingI` is the starting value for the loop iterator and `size` is the upper loop bound of the original loop.

This equation uses these values to calculate the largest multiple of the new loop stride that is less-than or equal to the upper bound of the loop.

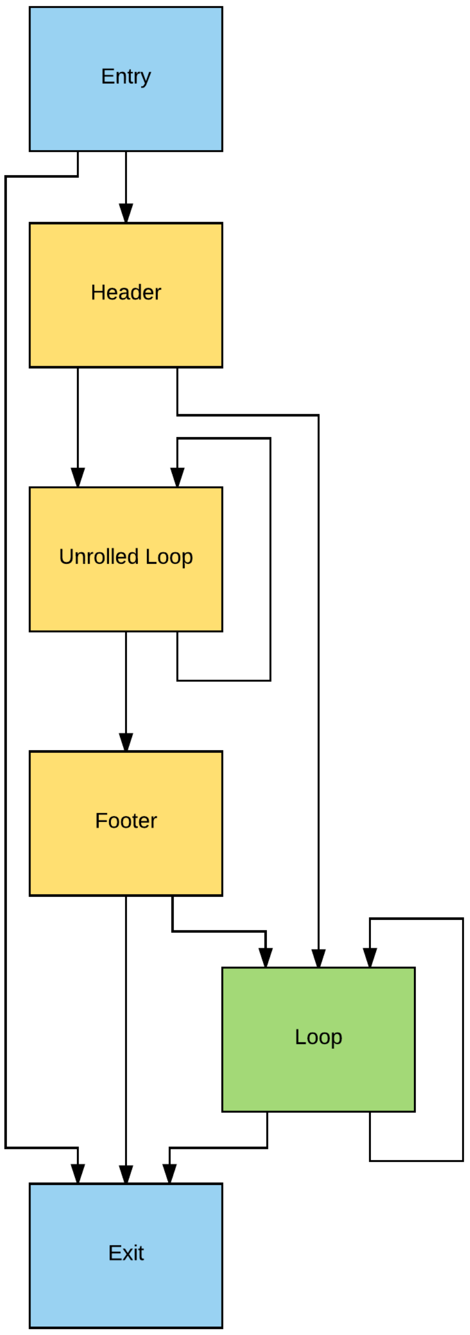


Figure 30 Control flow graph for the unrolled version of the example function multiply

The CFG for the unrolled version of the loop in the function `multiply` is shown in Figure 30. The entry and exit blocks for the function shown in blue remain the same in the unrolled version of the function. As seen in Figure 29, there are three new basic blocks inserted into the CFG which are shown in yellow. These blocks are a header block, the unrolled for loop and a footer block. An additional copy of the original loop at the end of the function is shown in green. This copy serves as a fall-back loop. When `size` is non-zero, the edge from the entry block to the header block is taken.

The header block is responsible for checking that the number of iterations to be performed is greater-than or equal to the unroll factor. In the example, the header block performs the check “`size >= 4`”. When this evaluates to true, the edge to the unrolled loop is taken and the unrolled loop is executed `iterations/N` times, where `iterations` is the number of times the original loop would have been executed before unrolling. When it evaluates to false the unrolled loop is skipped, and the original loop is executed `iterations` times. In this case, we do not need to perform the check "`i < size`” since we can prove that it always evaluates to true. In this case, `size` is guaranteed to be non-zero and the value of `i` is zero which means `i` is always less than `size`. This check is still performed at the end of each iteration of the loop.

The footer block is responsible for performing the check "`i < size`” when exiting the unrolled loop. If this check evaluates to true, there are additional iterations that must be performed so the edge to the fall-back loop is taken. When it evaluates to false, there are no additional iterations to perform so we can safely branch to the exit block.

The starting value of the iterator in the fall-back loop in the unrolled version of the function is dependent on which edge is taken on entry to the loop. When coming from the header block, the iterator starts at zero. When coming from the footer block, the iterator starts at the last iterator from the unrolled loop (as shown in Figure 29).

4.1.2. Design and Implementation

In this section, we will continue to use the function `multiply` as an example. The LLVM-IR for the main loop body is shown in Figure 31. The other basic blocks in the function are not relevant for this section as they are not vectorizable. The basic block

`%for.body.preheader` is the entry block for the loop, and the basic block `%for.cond.cleanup.loopexit` is the exit block for the loop.

There are two main steps involved in the tentative loop unroller. The first step is to determine if unrolling is feasible for the current basic block, and if it is, to provide a suitable unroll factor for the unroller. The second step duplicates the loop and unrolls that duplicate by the calculated unroll factor.

```
define void @multiply(i32* noalias nocapture %out,
             i32* noalias nocapture readonly %inA,
             i32* noalias nocapture readonly %inB,
             i32 %size) local_unnamed_addr #0 {
; ...
for.body:
  %i = phi i32 [ %inc, %for.body ],
               [ 0, %entry ]
  %idx0 = getelementptr inbounds i32, i32* %inA, i32 %i
  %load0 = load i32, i32* %idx0, align 4, !tbaa !1
  %idx1 = getelementptr inbounds i32, i32* %inB, i32 %i
  %load1 = load i32, i32* %idx1, align 4, !tbaa !1
  %mul = mul nsw i32 %load1, %load0
  %idx2 = getelementptr inbounds i32, i32* %out, i32 %i
  store i32 %mul, i32* %idx2, align 4, !tbaa !1
  %inc = add nuw i32 %i, 1
  %exitcond = icmp eq i32 %inc, %size
  br i1 %exitcond, label %for.cond.cleanup,
                   label %for.body
}
```

Figure 31 multiply for loop body as LLVM-IR

4.1.2.1. Calculate Unroll Factor

Before calculating the unroll factor for a given basic block, we must first ensure that it is a loop. If the current basic block is not a loop, it cannot be unrolled. Similarly, if the basic block already contains vector instructions then we assume that the loop has already been vectorized. It is possible that there are still vectorization opportunities in the loop, but we ignore them in our implementation due to potential problems with the existing vector instructions. We do this due to an implementation detail in LLVM's optimizer that causes the same optimization pass to be performed multiple times on the same source code function. As a result, our optimization could inadvertently unroll the same block multiple times, causing a cascading sequence of fall-back loops.

In section 4.3, we describe a technique called loop shifting which restructures the loop to facilitate the movement of instructions between iterations. We use this technique to attempt to re-use data that is shared between iterations of a vectorizable loop. If a loop already contains vectorizable instruction groups which re-use data between iterations, then loop unrolling is not performed. In these cases, unrolling a loop would only reduce the potential re-use of data between iterations.

Some loops may already be unrolled, either manually by the user or automatically by LLVM's own loop unrolling optimization pass. In these cases, if there are vectorizable groups of instructions with sizes that are equal to or greater than the natural vector size for the target architecture, loop unrolling is not performed here. Loop unrolling is still possible for these kinds of loops but we made the decision in our implementation to not attempt to unroll them.

When calculating the unroll factor for any given loop, the goal is to provide the vectorizer with the opportunity to utilize the full natural vector size of the target architecture. We use a heuristic in our implementation to achieve this. The equation that performs this calculation is

$$UF = 2^{\left\lceil \log_2\left(\frac{NVF}{VF}\right) \right\rceil}$$

Where UF is the unroll factor that is being calculated, NVF is the natural vectorization factor for the target architecture and VF is the existing vectorization factor of the loop. Once loop unrolling has been performed, the new vectorization factor for the loop becomes UF x VF.

To calculate the value of VF, the unroller performs the first two stages of the vectorizer as described in sections 3.2.1 and 3.2.2 with a few minor alterations. Instead of generating Groups of Available Instructions for all vectorizable opcodes, the unroller only generates load and store groups. These are the only two groups that are needed to perform the starting group generation step. When used by the vectorizer, the starting group generator imposes a minimum group size of two instructions on vectorizable groups. This minimum is reduced to one for the unroller. We do this so that we can gather as much information about the load and store instructions in the loop.

The number of generated groups for each vectorizable type is counted and the maximum group size for each type is also tracked. We do this to find the type that is most commonly used in the loop and what group size exists in the loop for that type. This is important as the number of elements in the natural vector size (NVF) for the target will vary based on type. We want to choose the value of NVF that suits the most used type in the loop. The value of VF is the maximum group size for this type.

For example, in the function `multiply` there are two load instructions and one store instruction in the basic block `%for.body`. The starting group generator will place each of these instructions into their own groups since they all use different base pointers. There are now three groups of one instruction each that have the type `i32`. This means `i32` is the most common type used in the loop and the maximum value for VF is one. Our target architecture has a natural vector size of 128-bits or 4x32-bit. When we plug these values into our equation we get

$$UF = 2^{\left\lceil \log_2\left(\frac{4}{1}\right)\right\rceil}$$

$$UF = 2^2$$

$$UF = 4$$

So, in the case of the function `multiply`, the loop `%for.body` has an unroll factor of four.

Consider another example function which has been manually partially unrolled by the user. This other loop operates on values of type `i16` and has been unrolled already by two. In this case, the starting group generator will generate groups of two instructions each. These groups all operator on values of type `i16`. Once again, our target has a natural vector size of 128-bits, which is also 8x16-bit. When we use these values in our equation we get

$$UF = 2^{\left\lceil \log_2\left(\frac{8}{2}\right)\right\rceil}$$

Which once again can be reduced to an unroll factor of 4.

4.1.2.2. *Perform Unrolling*

If the unroll factor that has been calculated is greater than one, then we can unroll the loop.

Before we start unrolling the loop, we first need to gather some information about the iterator that is used by the loop. We need to find

- the iterator update instruction
- the iterator's PHI instruction at the beginning of the loop
- the compare instruction attached to the loop's branch instruction
- the upper loop bound

There are several requirements attached to these pieces of information that must be met to allow safe unrolling:

1. The iterator must be an integer value (of any bitsize) and the compare instruction must be an integer compare with an integer equality predicate. If the predicate is any other (e.g. integer less-than) then there is no guarantee that the calculation of the new upper loop bound will return the correct value. This is because we do not know by how much the iterator may overrun the original loop bound at runtime.
2. The iterator update instruction must be either an integer add or subtract instruction in which the first input operand is produced by the iterator PHI instruction and the second input operand is an integer constant. If the loop stride is a variable, then we cannot safely unroll.

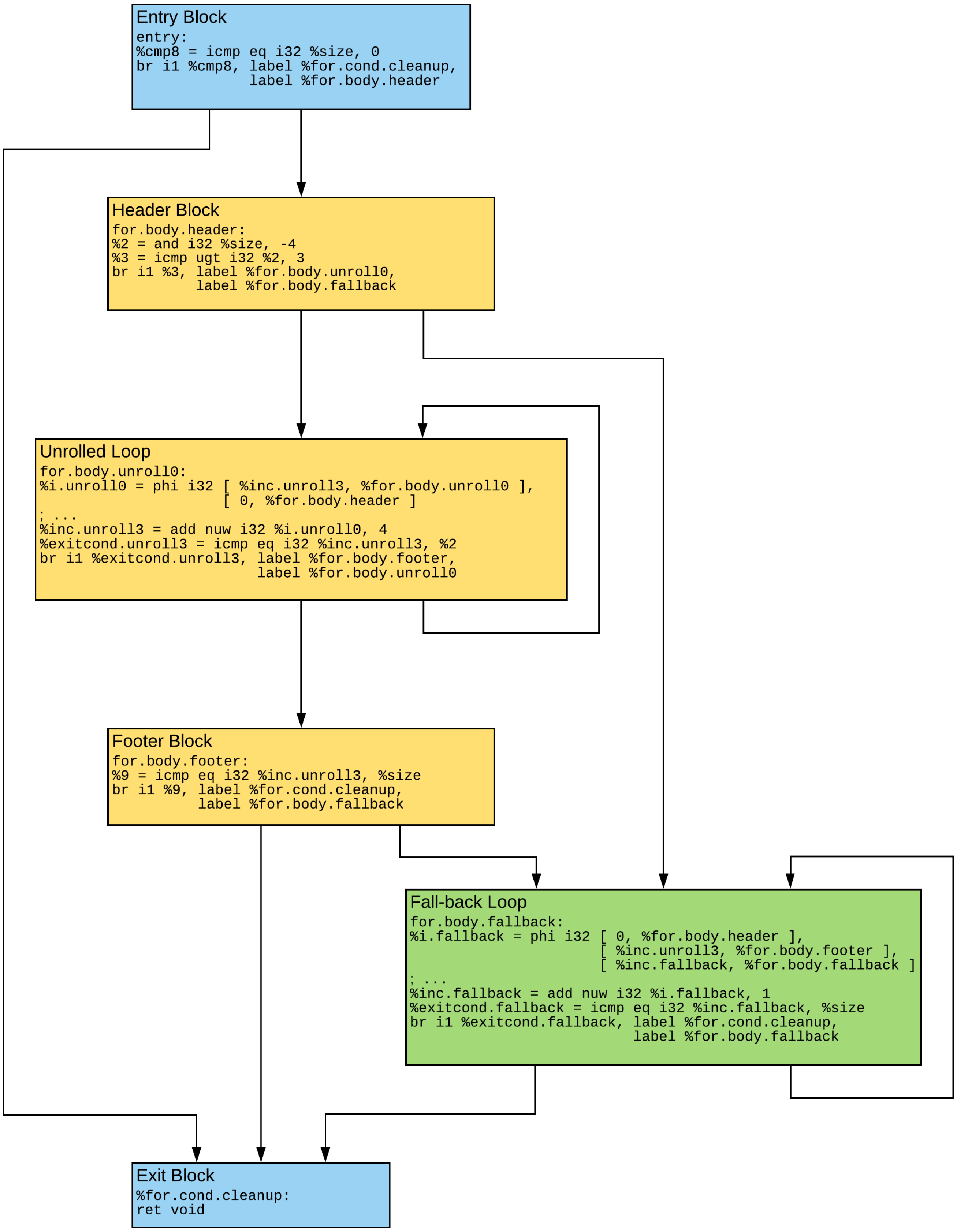
3. The iterator PHI instruction must take the iterator update instruction described in requirement 2 as its operand for the back-edge of the loop, creating a simple dependency loop of PHI to update to PHI across iterations.

Once all this information has been gathered and the conditions have been met, we can begin the loop unrolling process.

First, we create a clone of the original loop body. This clone will serve as the fall-back loop as shown in Figure 30 and in Figure 32. LLVM provides a function for cloning a basic block which we use here. This function duplicates all the instructions in a basic block in a new block but it does not replace references to the original instructions within the new block. This means we need to do a pass over all instructions in the new basic block and replace all references to instructions in the original block with references to their corresponding instruction in the new block. The cloning function provides a "Value to Value" map to facilitate this.

The next step is to create the unrolled loop body. We do this by creating UF clones of the original loop basic block. Just like the fall-back loop body, we need to tidy up references to instructions within each of the cloned basic blocks. This is done for all instructions except for uses of the iterator. Uses of the iterator must be replaced with a new set of instructions. Uses of the iterator in the first cloned basic block can be left as they are, since the iterator value is correct. Uses in subsequent blocks must be replaced with a new instruction for each block. In the first subsequent block, we create an instruction which generates what the iterator value would be during the corresponding iteration of the original loop. In our example function, we need to create an `add` instruction for this block that adds constant one to the loop iterator. In the next block, we need an instruction that adds constant two to the loop iterator, and so on.

The only exception to this is the loop iterator update instruction in the final block which uses the iterator PHI instruction from the very first block as its input. Similarly, we need to update this PHI instruction to take the iterator update instruction from the final block as its input for the loop back edge.



```
entry:
%cmp8 = icmp eq i32 %size, 0
br i1 %cmp8, label %for.cond.cleanup,
             label %for.body.header
```

```
for.body.header:
%2 = and i32 %size, -4
%3 = icmp ugt i32 %2, 3
br i1 %3, label %for.body.unroll0,
          label %for.body.fallback
```

```
for.body.unroll0:
%i.unroll0 = phi i32 [ %inc.unroll3, %for.body.unroll0 ],
                     [ 0, %for.body.header ]
; ...
%inc.unroll3 = add nuw i32 %i.unroll0, 4
%exitcond.unroll3 = icmp eq i32 %inc.unroll3, %2
br i1 %exitcond.unroll3, label %for.body.footer,
                         label %for.body.unroll0
```

```
for.body.footer:
%9 = icmp eq i32 %inc.unroll3, %size
br i1 %9, label %for.cond.cleanup,
          label %for.body.fallback
```

```
for.body.fallback:
%i.fallback = phi i32 [ 0, %for.body.header ],
                      [ %inc.unroll3, %for.body.footer ],
                      [ %inc.fallback, %for.body.fallback ]
; ...
%inc.fallback = add nuw i32 %i.fallback, 1
%exitcond.fallback = icmp eq i32 %inc.fallback, %size
br i1 %exitcond.fallback, label %for.cond.cleanup,
                          label %for.body.fallback
```

```
%for.cond.cleanup:
ret void
```

Figure 32 Control flow graph for the example function multiply after loop unrolling

Finally, we need to delete the compare and terminator instructions from all cloned basic blocks except for the last basic block in the chain. At this point all cloned blocks are merged together to form a single unrolled loop body. The terminator instruction (branch instruction) for this block must be updated to branch to the start of the block.

The next step is to create the header and footer blocks shown in yellow in Figure 32. Into the header instruction, we insert a few calculations. The first set of calculations are for the loop bound of the new unrolled loop. The calculation described in the previous section is used for this purpose. This new bound is inserted into the unrolled loop's compare instruction, in place of the original loop bound. This new bound is also used as a pre-check in this block to ensure we can safely enter the unrolled loop at runtime. If the loop iterator is incremented in the loop body, then we substract the starting iterator value from the new loop bound and compare that value with the new loop stride. If the new loop stride exceeds the value of the distance between the starting iterator and the new loop bound, then we cannot enter the unrolled loop without overflowing. In this case, the header branches to the fall-back loop. If the loop iterator is decremented in the loop body, then we subtract the new loop bound from the starting iterator to calculate the distance between them.

The PHI instructions at the beginning of the unrolled loop must be updated to reflect the new incoming edge from the header block.

Into the footer block we need to add compare and branch instructions to check if the unrolled loop completed all iterations of the original loop. This is done by comparing the last iterator value (after the update) to the original loop bound. If the iterator is less-than the bound, then we must branch into the fall-back loop to complete the final iterations. Otherwise we branch to exit block.

Once this has been completed, the unrolled loop body is handed over to the SLP vectorizer for vectorization. If the vectorizer fails to vectorize the unrolled loop, then the created basic blocks are all deleted, and the original loop body is maintained in the function. We unroll the loop only because it may be beneficial to vectorization, unrolling the loop for the potential benefits to the scalar loop is beyond the scope of this optimization. Therefore, if vectorization fails we revert all changes.

4.2. Associative Chain Reordering

4.2.1. Transformation

Associative Chain Reordering (ACR) is the name we have given to a technique which takes advantage of the associativity of certain operations to reorganize or reorder a sequence of such operations to make them suitable candidates for vectorization.

Throughout this section, we will consider ACR within the context of the example function `accumulate` as shown in Figure 33. This function accumulates `N` integer values from the input array `in` and stores the result to an element of the output array `out`. This is performed `size` times. For our purposes, we will assume that the inner for loop which accumulates into the value `sum` has been fully unrolled before vectorization begins.

```
template <unsigned int N>
void accumulate (int * __restrict in,
                 int * __restrict out,
                 unsigned int size) {
  for(unsigned int i = 0; i < size*N; i+=N) {
    int sum = 0;
    for(unsigned int j = 0; j < N; ++j)
      sum += in[i+j];
    out[i/N] = sum;
  }
}
```

Figure 33 accumulate<N> example function

Consider the function `accumulate<8>`. This version of the accumulate function accumulates eight values from the input array `in` and stores the result to the output array `out`. The original scalar instruction chains for this function are shown in Figure 34. The part of this instruction chain that we are particularly interested in for ACR is the `add` instruction chain shown in peach.

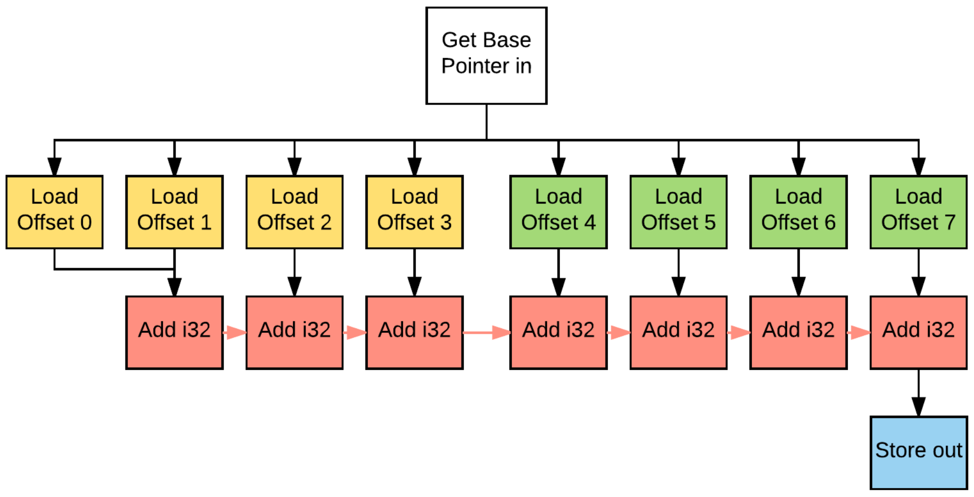


Figure 34 accumulate<8> original scalar instruction chain

For a chain of instructions to be considered a candidate for reordering there are several requirements that must be met. All instructions in the chain must contain the same opcode, and the chain must not be interrupted by any instruction of a different opcode. The opcode that is common throughout the chain must be both associative and commutative. Associativity enables us to rearrange the order of these operations only if the sequence of the operands is not changed. When this is coupled with the commutativity property, we can reorder both the operations themselves and the sequence of operands. Signed integer `add` instructions are both associative and commutative, so we can transform the chain of instructions in our example.

Conceptually, we can consider a sequence of instructions like those shown in Figure 34 as a single multiple input instruction with the same opcode as shown in Figure 35. This single multi-input instruction is a black-box which produces the sum of all its inputs. There is no defined ordering for the sequence of operations or the operands with this instruction.

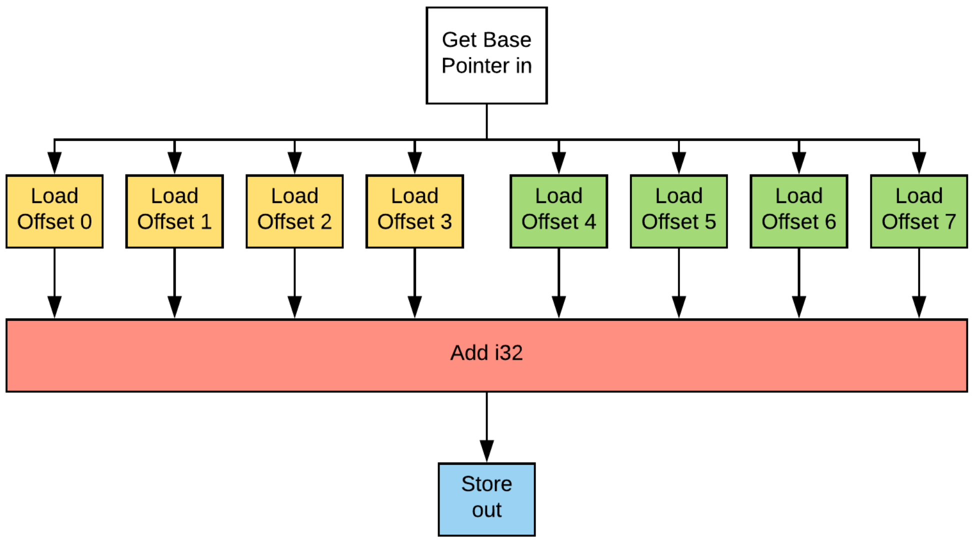


Figure 35 accumulate<8> instruction chain with multi-input add instruction

As described in section 3.1, scalar load instructions can be grouped together into vector instructions when they access consecutive locations in memory. Because of this, we can group the loads from `in` with offsets zero through three together (shown in yellow) and the loads from `in` with offsets four through seven together (shown in green). The vectorized chain for `accumulate<8>` is shown in Figure 36. These new vector instructions are now the only two input operands to the multi-input `add` instruction. This allows us to vectorize this `add` instruction as well. We start by creating a vector `add` instruction which takes the two generated vector instructions as inputs. This produces a single `v4i32` vector as a result. However, a single scalar value must be produced as the input to the store instruction to the array `out`. This scalar value is the sum of the four values in the result of the vector `add` instruction. To do this, we produce a horizontal vector `add` instruction which adds all the elements in a vector value together and produces a single scalar result.

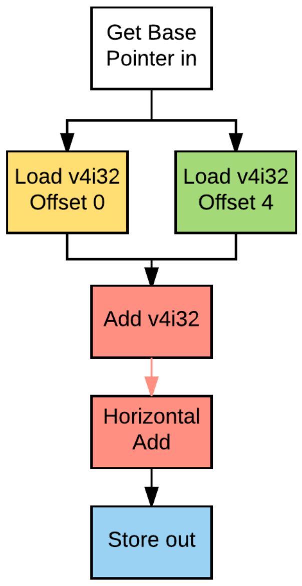


Figure 36 accumulate<8> vectorized instruction chain

There is one element of the transformation that is not evident in `accumulate<8>`. Not all input operands to the multi-input instruction will necessarily be vectorized. It is possible that after vectorization has been completed, there are input operands to the original scalar chain that are not present in the vector chain. We generate a new scalar chain of instructions which takes these remaining operands as their inputs. The result of this scalar chain is "joined" with the result of the horizontal operation through an additional scalar operation. The value produced by this final operation is the result of the chain.

4.2.2. Design and Implementation

While describing the design and implementation of the ACR transformation, we will use the example function `accumulate<17>`. This function gathers 17 values from the input array `in`, accumulates them and stores the result to the output array `out`. The diagram in Figure 37 shows the original scalar instruction chain for this function. Once again, we are interested primarily in the chain of `add` instructions shown in peach.

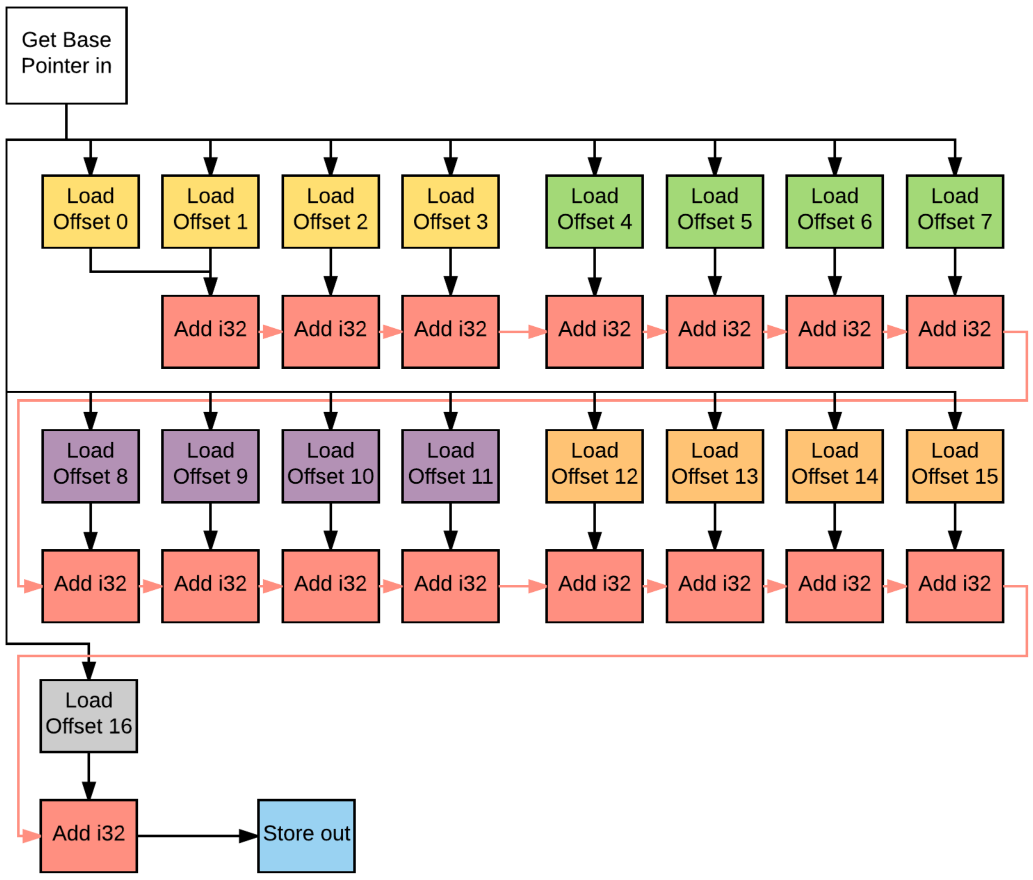


Figure 37 accumulate<17> original scalar instruction chain

In section 3.2 we described the design and implementation of the SLP vectorizer in the following five steps:

1. Generate groups of available instructions
2. Generate starting groups
3. Generate vector chains
4. Check memory dependencies
5. Vectorize chains

The ACR transformation requires modification to steps one, three, and five. We will describe the transformation design and implementation here in relation to these three steps independently in each of the three following subsections.

4.2.2.1. *Generate Groups of Available Instructions*

This step which is responsible for generating Groups of Available Instructions for the vectorizer is actually left unmodified for the ACR transformation. Instead, the information generated in this step is supplemented with additional information.

All instructions are placed into their respective groups as before, including all instructions that are part of instruction chains that are candidates for ACR. The instructions that are part of these chains are placed into additional groups as well. Each chain of instructions is placed into a group of its own. Each group for a chain contains all instructions in that chain, with no exceptions. Unlike the Groups of Available Instructions that are usually generated, there can be multiple associative groups for each opcode. Each instruction can only exist in one associative group.

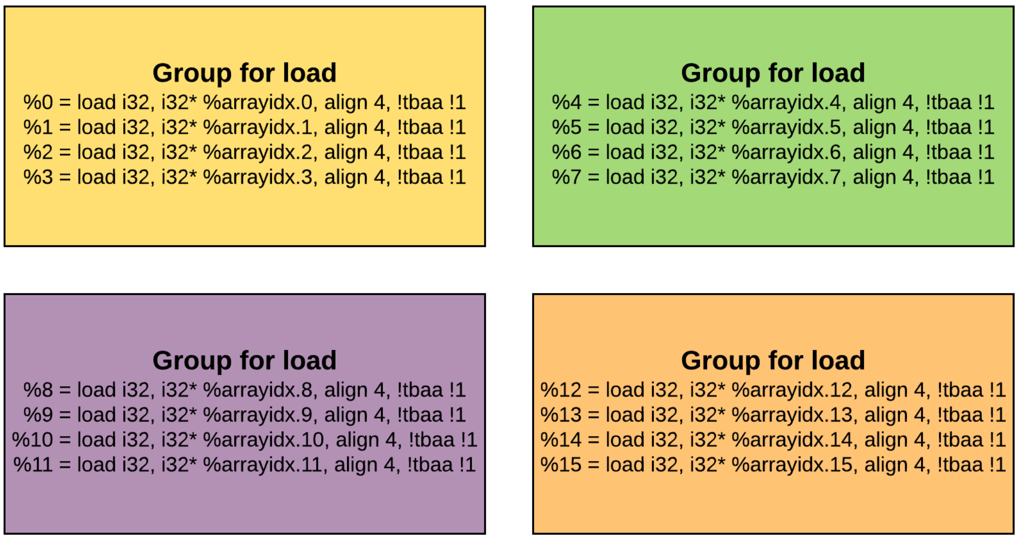
We place chains of instructions into individual groups like this to allow for a safe transformation when vectorizing. It is possible that not all input operands to an associative chain will be vectorized. In these cases, the vectorizer must be able to generate an additional scalar chain to include these remaining operands that were not vectorized. This is explained in greater detail later in section 4.2.2.3.

4.2.2.2. *Generate Vector Chains*

Before generating chains, we need a starting point. The starting group generator produces four starting groups for the loop in `accumulate<17>`. These groups are made up of four load instructions each. The groups of instructions generated are for the load instructions shown in yellow, green, purple and orange in Figure 37. These starting nodes for the DAG are shown in Figure 38.

Under normal circumstances, the chain generator will search for matching operands for one of these groups to grow the vectorizable graph. This is not possible for associative chains since they usually begin with a single operation with both operands coming from the same vectorized node as shown in Figure 37. The chain generator will also ensure that there are no inter-dependencies between instructions in the same node. In this context, dependencies refers to flow-dependencies. This means that any chain of instructions will never be vectorized since each instruction in the chain is dependent on the output of the previous instruction in the chain.

Because of this, we must handle associative chains independently of other instruction sequences. The vectorizer generates vectorizable nodes for associative chains only when all other vectorizable nodes have been generated. We wait until all other nodes have been generated so we can maximize the number of vector inputs to the associative chain. In `accumulate<17>` there are no other nodes to be generated, so we can jump straight into generating nodes for the associative chain.



```
%0 = load i32, i32* %arrayidx.0, align 4, !tbaa !1
%1 = load i32, i32* %arrayidx.1, align 4, !tbaa !1
%2 = load i32, i32* %arrayidx.2, align 4, !tbaa !1
%3 = load i32, i32* %arrayidx.3, align 4, !tbaa !1
```

```
%4 = load i32, i32* %arrayidx.4, align 4, !tbaa !1
%5 = load i32, i32* %arrayidx.5, align 4, !tbaa !1
%6 = load i32, i32* %arrayidx.6, align 4, !tbaa !1
%7 = load i32, i32* %arrayidx.7, align 4, !tbaa !1
```

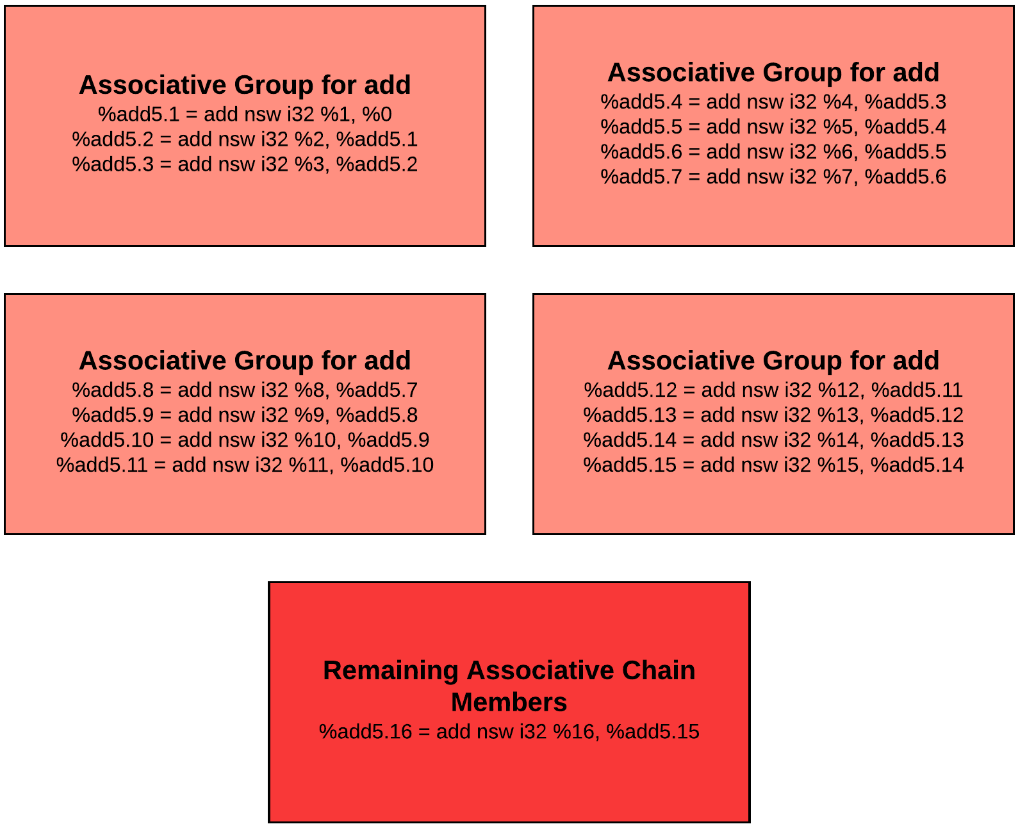
```
%8 = load i32, i32* %arrayidx.8, align 4, !tbaa !1
%9 = load i32, i32* %arrayidx.9, align 4, !tbaa !1
%10 = load i32, i32* %arrayidx.10, align 4, !tbaa !1
%11 = load i32, i32* %arrayidx.11, align 4, !tbaa !1
```

```
%12 = load i32, i32* %arrayidx.12, align 4, !tbaa !1
%13 = load i32, i32* %arrayidx.13, align 4, !tbaa !1
%14 = load i32, i32* %arrayidx.14, align 4, !tbaa !1
%15 = load i32, i32* %arrayidx.15, align 4, !tbaa !1
```

Figure 38 The starting nodes for the DAG of vectorizable nodes in accumulate<17>

All instructions in an associative chain are split into two separate groups, those with a vectorized input operand and those without. The group of instructions which have a vectorized input operand are further split up into subgroups of instructions which have a matching vector input node. As with the function `accumulate<17>`, the first instruction in a scalar associative chain can be a special case where both operands come from the same vector input. This is not always the case however. For example, if the accumulation happens across the outer loop bound then one of the operands to the first instruction in the chain will be a PHI node. Both of these possibilities must be handled when generating the vectorized chain.

Any instructions that can't be placed into a subgroup of instructions which have a matching vector input node are added to the group of instructions without a vectorized input operand. In the case of `accumulate<17>`, the last `add` instruction in the chain is the only member of this group. The subgroups generated for accumulate<17> are shown in Figure 39.


Associative Group for add
Associative Group for add
Associative Group for add
Associative Group for add
Remaining Associative Chain Members

```
%add5.1 = add nsw i32 %1, %0
%add5.2 = add nsw i32 %2, %add5.1
%add5.3 = add nsw i32 %3, %add5.2
```

```
%add5.4 = add nsw i32 %4, %add5.3
%add5.5 = add nsw i32 %5, %add5.4
%add5.6 = add nsw i32 %6, %add5.5
%add5.7 = add nsw i32 %7, %add5.6
```

```
%add5.8 = add nsw i32 %8, %add5.7
%add5.9 = add nsw i32 %9, %add5.8
%add5.10 = add nsw i32 %10, %add5.9
%add5.11 = add nsw i32 %11, %add5.10
```

```
%add5.12 = add nsw i32 %12, %add5.11
%add5.13 = add nsw i32 %13, %add5.12
%add5.14 = add nsw i32 %14, %add5.13
%add5.15 = add nsw i32 %15, %add5.14
```

```
%add5.16 = add nsw i32 %16, %add5.15
```

Figure 39 Associative subgroups generated for the add chain in accumulate<17>

These generated subgroups are now used to generate new nodes in the DAG. All groups that are generated are used to create a new chain of vector associative operations. This means that except for the first node in the new chain, each node in the chain must take another node in the chain as one of its inputs. The other input will come from an additional node in the DAG that is not part of the associative chain.

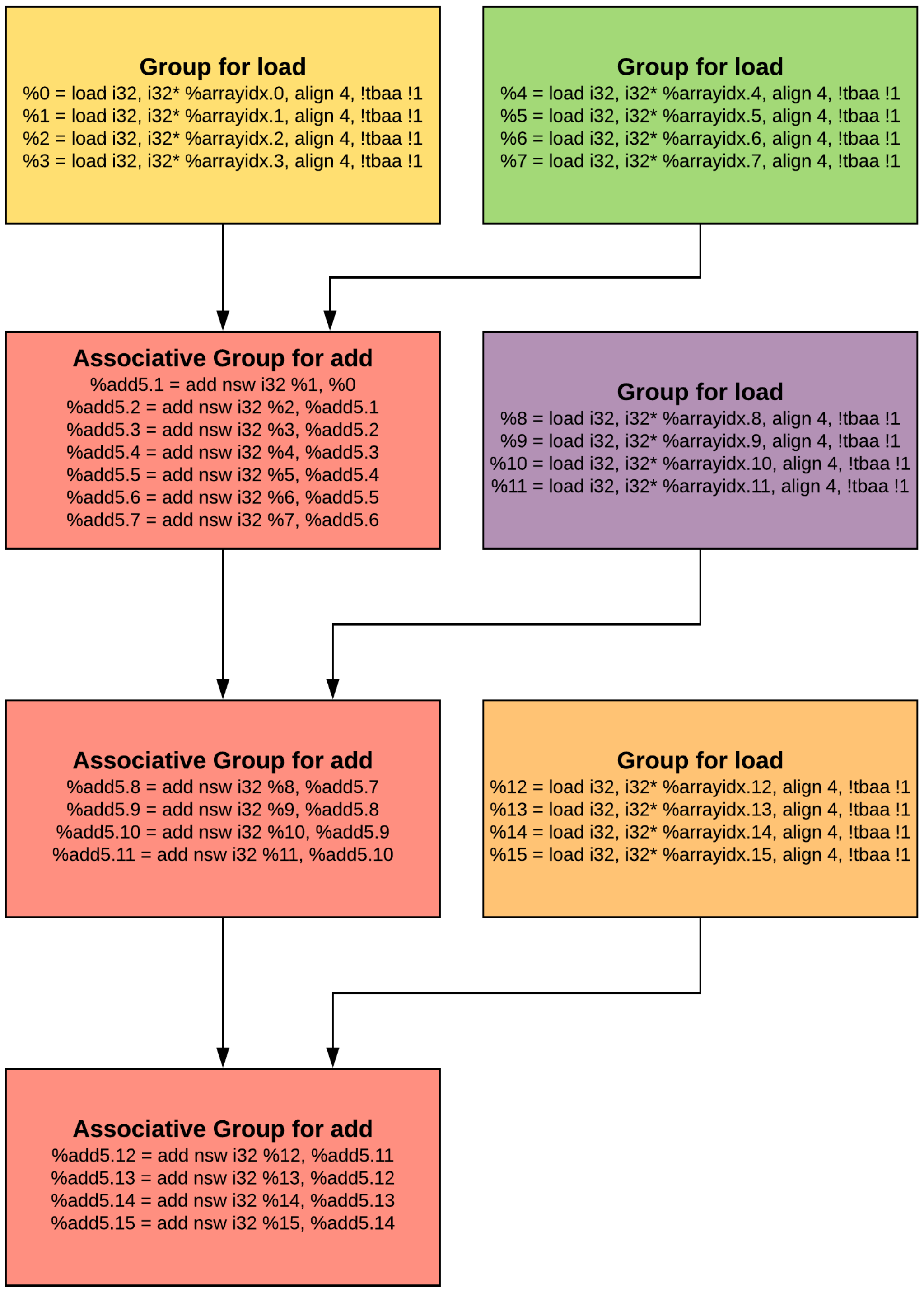

Group for load
%0 = load i32, i32* %arrayidx.0, align 4, !tbaa !1
%1 = load i32, i32* %arrayidx.1, align 4, !tbaa !1
%2 = load i32, i32* %arrayidx.2, align 4, !tbaa !1
%3 = load i32, i32* %arrayidx.3, align 4, !tbaa !1
Group for load
%4 = load i32, i32* %arrayidx.4, align 4, !tbaa !1
%5 = load i32, i32* %arrayidx.5, align 4, !tbaa !1
%6 = load i32, i32* %arrayidx.6, align 4, !tbaa !1
%7 = load i32, i32* %arrayidx.7, align 4, !tbaa !1
Associative Group for add
%add5.1 = add nsw i32 %1, %0
%add5.2 = add nsw i32 %2, %add5.1
%add5.3 = add nsw i32 %3, %add5.2
%add5.4 = add nsw i32 %4, %add5.3
%add5.5 = add nsw i32 %5, %add5.4
%add5.6 = add nsw i32 %6, %add5.5
%add5.7 = add nsw i32 %7, %add5.6
Group for load
%8 = load i32, i32* %arrayidx.8, align 4, !tbaa !1
%9 = load i32, i32* %arrayidx.9, align 4, !tbaa !1
%10 = load i32, i32* %arrayidx.10, align 4, !tbaa !1
%11 = load i32, i32* %arrayidx.11, align 4, !tbaa !1
Associative Group for add
%add5.8 = add nsw i32 %8, %add5.7
%add5.9 = add nsw i32 %9, %add5.8
%add5.10 = add nsw i32 %10, %add5.9
%add5.11 = add nsw i32 %11, %add5.10
Group for load
%12 = load i32, i32* %arrayidx.12, align 4, !tbaa !1
%13 = load i32, i32* %arrayidx.13, align 4, !tbaa !1
%14 = load i32, i32* %arrayidx.14, align 4, !tbaa !1
%15 = load i32, i32* %arrayidx.15, align 4, !tbaa !1
Associative Group for add
%add5.12 = add nsw i32 %12, %add5.11
%add5.13 = add nsw i32 %13, %add5.12
%add5.14 = add nsw i32 %14, %add5.13
%add5.15 = add nsw i32 %15, %add5.14

Figure 40 Generated DAG for the function accumulate17

For the first node in the chain, we merge two of the subgroups into a single node. As mentioned above, unlike every other node in the chain, the first node has two inputs that come from nodes in the DAG that are not in the chain. As a result, instead of `N` instructions being associated with the node, where `N` is the vectorization factor, there are either `N*2` or `(N*2)-1` instructions associated with the first node. In the case of `accumulate<17>`, the top two subgroups shown in Figure 39 are merged together. This means there are seven (`(N*2)-1`) instructions associated with the first node, instead of four (`N`) as shown in Figure 40. The number of instructions associated with the first node is `N*2` when the first instruction in the original chain of instructions has an input operand that is not a member of a vectorizable node in the DAG. As an example, as mentioned previously, this can happen when an associative chain is accumulating across the back-edge of a loop (i.e. the "starting value" of the accumulation in this iteration comes from the previous iteration of the loop).

The group of instructions that were not added to vectorizable nodes in the DAG (as shown in red in Figure 39) is maintained for use during vectorization of the DAG in the next section.

4.2.2.3. Vectorize Chains

For the most part, the vectorization of associative chain vector nodes is the same as every other type of node. However, there are some important differences which should be noted.

Vectorization of non-associative nodes requires strict enforcement of the number of instructions in each node. Specifically, each node must contain the same number of instructions as every other node in the DAG. However, the number of instructions in an associative chain node can be ignored. The size of the node will not always match the size of other nodes in the graph. For example, the first `add` node in the vectorized associative chain in Figure 40 has seven scalar instructions associated with it. These seven instructions are the first seven instructions in the associative chain in Figure 37. However, the next `add` in the vector chain only has four scalar instructions.

The node resizing step still applies to the associative chains. When resizing these nodes, we still remove instructions from the "end" of the group. However, we can't just leave the original scalars in place like we do for all other node types. We must keep track of these instructions by adding them to the group of non-vectorized instructions for the entire associative chain. This is the same group shown in red in Figure 39.

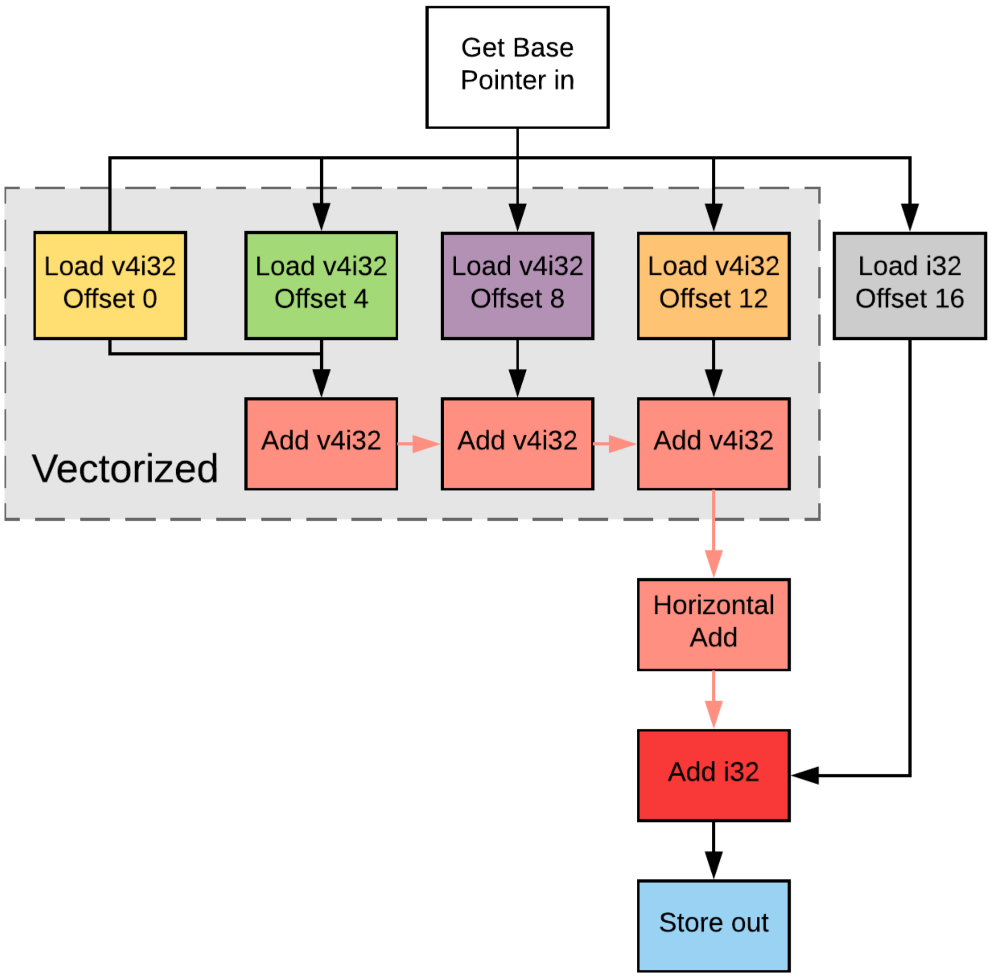


Figure 41 accumulate<17> vectorized instruction chain

The vector element insert step can be skipped for associative chain nodes. All inputs for these nodes are guaranteed to be already built vector nodes. When building the vectorizable nodes in section 4.2.2.2, each vectorizable node takes two full vector nodes as its input operands. Except in the case of the very first node in the chain, one of these operand nodes is another vectorized set of instructions from the original scalar chain and the other is some other node that was generated independently. Any instructions that don't have a corresponding vector input are not inserted into vectorizable nodes and are reserved for the additional scalar chain.

Once all nodes have been vectorized, the element extract step is performed to tidy up any remaining references to instructions in each of the nodes. It is during this step that the horizontal operation at the end of associated vector chain shown in Figure 41 is generated.

This is also the point at which we tidy up any remaining scalar instructions which are in the non-vectorized instruction groups for the associative chain. This is the red group of instructions we saw previously in Figure 39. We gather together all input operands from outside the chain for each of these instructions. These input operands are then used to create a new scalar associative chain of instructions. In the case of `accumulate<17>`, there is only one instruction in this set which has only one input operand that is generated outside of the chain. This operand is the remaining load operation from the input array `in`, shown in grey in Figure 41. We generate the new `add` instruction shown in red to tidy up this final reference from the original chain. The second operand to this new instruction is the result of the horizontal `add` instruction. The value produced by this `add` instruction is used to replace all references to the old value that was produced by the scalar accumulation chain.

4.3. Loop Shifting

4.3.1. Transformation

Loop shifting is an optimization technique that involves the movement of operations from the beginning of a loop to the end of that loop via the loop's back-edge. Operations which are moved in this way are duplicated in the loop's prologue to maintain the original semantics of the loop. In this section, we describe an optimization technique based on loop shifting that is designed to take advantage of data re-use across loop iterations, a sort of rotating partial loop invariance. We do this by shifting memory read instructions from the first iteration of a loop and restructuring the remaining loop body to minimise the number of memory accesses required on subsequent iterations.

Throughout this section, we will use the function `averages<N>` shown in Figure 42 as an example. This function computes a rolling average of `N` elements from the input array in and stores the result to the output array out. This action is performed size times. For our purposes here, we assume that the inner for loop with bounds N is fully unrolled so that it becomes part of the body of the outer for loop.

```
template <unsigned int N>
void averages (int * __restrict in, int * __restrict out,
               unsigned int size) {
  for (unsigned int i = 0; i < size; ++i) {
    int sum = 0;
    for (unsigned int j = 0; j < N; ++j)
      sum += in[i+j];
    out[i] = sum/N;
  }
}
```

Figure 42 averages<N> example function

The primary goal of our loop shifting optimization is to maximise the amount of data re-use between iterations and minimise the use of memory reading instructions. We do this to optimize important loops which load the same data from memory on consecutive iterations of the loop. To achieve this goal, we shift the load instructions for the first iteration into the loop prologue. At the end of each iteration, we load the extra data that is required for the next iteration of the loop only and re-use the rest of the data from the current iteration.

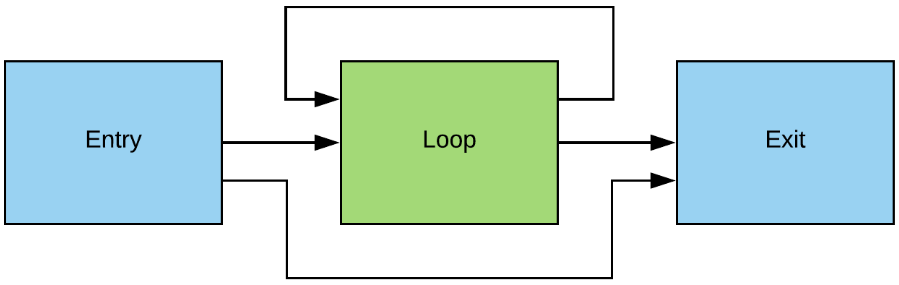


Figure 43 Control flow graph for the function averages<N>

Consider the control flow graph (CFG) for the function averages<N> shown in Figure 43. The basic block labelled Entry performs a value check for the argument size. If the value of size is zero, then the edge to the basic block labelled Exit is taken. Otherwise the edge into the loop body is taken. The main loop body shown in green is the basic block that we will be transforming. This CFG is the same for all `averages<N>` functions.

The function `averages<4>` provides an opportunity for shifting the load instructions from the first loop iteration to re-use data between iterations. The diagram in Figure 44 demonstrates the overlap present between each iteration of the loop. On the first iteration, the body of the loop accesses elements zero through three of the array `in`. On the next iteration, elements one through four are accessed. This iteration shares elements one, two and three with the previous iteration. Rather than reloading them on this iteration, we should use the data that was already loaded on the previous iteration.

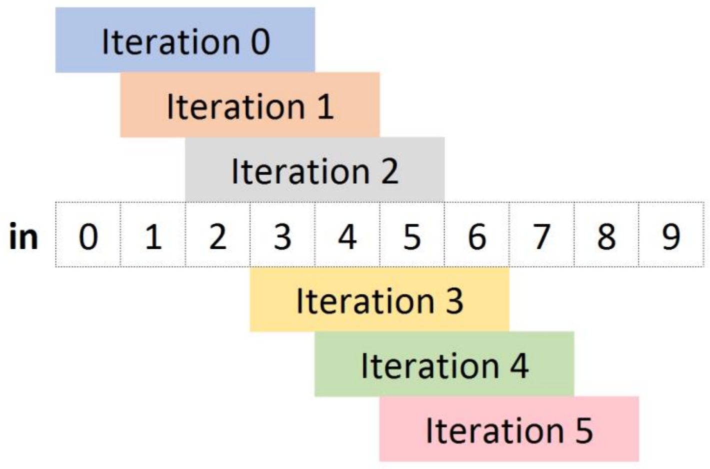


Figure 44 Overlapping regions of memory accessed on each iteration of averages<4>

To take advantage of this overlap in data use between iterations, we restructure the body of the loop. To shift instructions from the first loop iteration, we need to create two new basic blocks. The first of these blocks is the basic block labelled `Preload` shown in yellow in Figure 45. This is the block into which we will move load instructions from the first iteration of the loop. There are four scalar load operations which are to be moved into this block. These four scalar loads (shown in blue in Figure 44) are vectorized and inserted into the block labelled `Preload` as a single vector load of four elements. The `Preload` block serves as the loop's prologue.

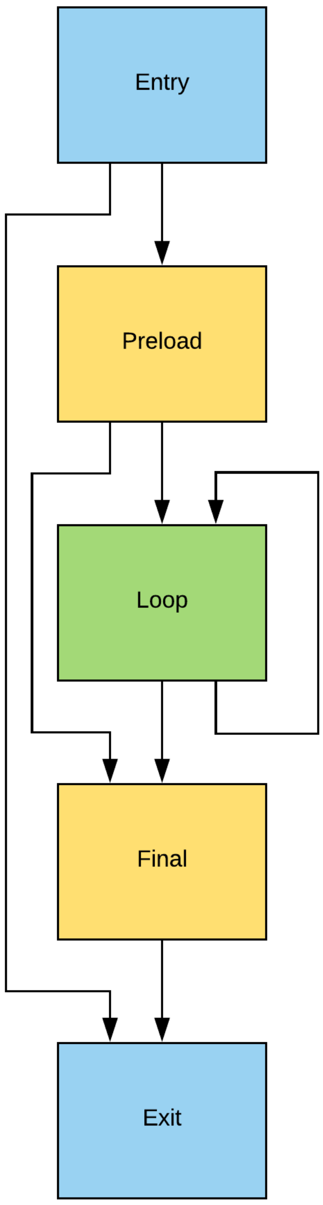


Figure 45 Control flow graph for the function averages<4> after loop shifting

A PHI instruction is inserted at the beginning of the `Loop` block (shown in green). This PHI instruction produces the value of the vector load instruction in the `Preload` block when the edge from that block is taken, or the modified vector value produced on the previous iteration when the back-edge of the loop is taken. This modified vector value contains the three overlapping elements from the previous iteration, plus the next value in the array that is “new” in this iteration.

```
for.preload:
  %newbound = add i32 %size, -1
  %preidx = bitcast i32* %in to <4 x i32>*
  %prevec = load <4 x i32>, <4 x i32>* %preidx, align 4
  %boundcheck = icmp eq i32 %size, 1
  br i1 %boundcheck, label %for.final, label %for.body

for.body:
  %vector = phi <4 x i32> [ %insert, %for.body ],
                          [ %prevec, %for.preload ]
  %i = phi i32 [ %add, %for.body ],
               [ 0, %for.preload ]
  ; ...
  %nextidx = getelementptr i32, i32* %arrayidx.3, i32 1
  %next = load i32, i32* %nextidx, align 4
  %shuf = shufflevector <4 x i32> %vector,
                        <4 x i32> undef,
          <4 x i32> <i32 1, i32 2, i32 3, i32 undef>
  %insert = insertelement <4 x i32> %shuf,
                          i32 %next, i64 3
  %add = add nuw i32 %i, 1
  %exitcond = icmp eq i32 %add, %newbound
  br i1 %exitcond, label %for.final, label %for.body

for.final:
  %finalvec = phi <4 x i32> [ %prevec, %for.preload ],
                            [ %insert, %for.body ]
  %i.final = phi i32 [ 0, %for.preload ],
                     [ %add, %for.body ]
  ; ...
  br label %exit
```

Figure 46 Stripped down version of averages<4> as LLVM-IR

At the end of the `Loop` block, we insert the instructions that produce this modified vector value. We do this using a sequence of three instructions as shown in Figure 46. The first instruction is a vector shuffle instruction (the instruction which produces the value `%shuf`) that shifts the elements of the vector left by one. This effectively replaces element zero with the value from element one, element one with the value from element two, and element two with the value from element three. The new value in element three is undefined. We replace this undefined value with the next value in the input array in. This new value is the value stored in the array at index `i+4` (i.e the value in the array immediately after the last value accessed in the current iteration). We do this by issuing a scalar load from the desired address (the instruction producing the value `%next`), followed by a vector element insert instruction into element three of the vector value that was produced by the shuffle. The value produced by this vector element insert instruction (`%insert`) is the incoming value for the PHI instruction at the beginning of the block for the back edge of the loop.

The basic block labelled `Final` in Figure 45 contains the instructions for the last iteration of the loop. All instructions, except for those moved into the `Preload` block are copied into the `Final` block. At the beginning of this block, we insert a PHI instruction. This PHI instruction produces the value of the vector load instruction in the `Preload` block when the edge from that block is taken. When the edge from the `Loop` block is taken, this PHI instruction produces the value of the vector element insert instruction described above. All uses of the values loaded from `in` in the `Final` block are replaced with the value that is produced by this PHI instruction. The `Final` block essentially serves as the loop's epilogue.

The edge between the `Preload` block and the `Final` block is taken when there is only one loop iteration to be performed (i.e. the value of `size` is one).

4.3.2. Aggressive Transformation

The function `averages<4>` is well-suited to this loop shifting optimization as it contains four consecutive scalar `i32` loads from memory and our target architecture has a 128-bit vector unit. However, consider the case of the function `averages<2>`. With this function, there are only two consecutive scalar `i32` loads from memory. To vectorize in cases like this, we could use the smaller vector size of 64-bits. Alternatively, we can use the full 128-bit vectors available on the target and only operate on the first two elements of the loaded values. This is the approach we will use for the aggressive form of the transformation.

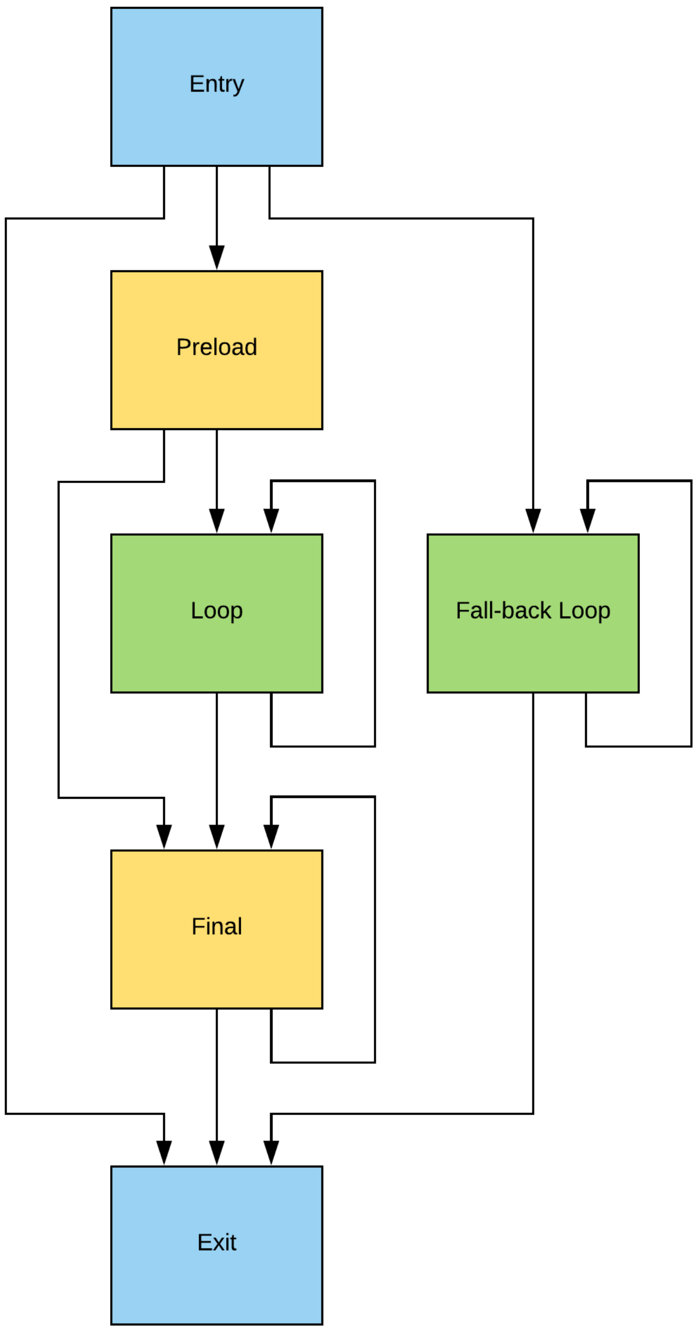


Figure 47 Control flow graph of the function averages<2> with aggressive pre-loading

As with the function `averages<4>`, in `averages<2>` we create two new basic blocks called `Preload` and `Final` to act as the loop's prologue and epilogue. However, we must take extra care as there are additional considerations when preloading more than one iteration like this.

We introduce a third additional block when performing this more aggressive preloading. Since we are shifting more than one iteration from the loop body, to maintain the original semantics of the loop and to prevent reading memory that is out of bounds, we check that there are enough iterations to be performed before entering the transformed loop. If there are not enough iterations, then we branch to the `Fall-back Loop` block shown in green in Figure 47. The following equation is used to determine how many iterations are being shifted from the loop body.

$$SI = (NVF - VF) + 1$$

Where SI is the number of iterations we are preloading data for, NVF is the natural vectorization factor for the type being loaded and VF is the existing vectorization factor in the block.

When the value of `size` is less than SI, the branch to the `Fall-back Loop` block is taken. Otherwise, the branch into the shifted loop via the `Preload` block is taken.

The `Preload` block is now responsible for loading data for more than one iteration of the loop. In the case of `averages<2>`, NVF is four and VF is two. Therefore, the number of iterations we are preloading for in the `Preload` block (i.e. SI) is three. At the end of the `Preload` block, the conditional logic for the branch instruction is different. If the value of `size` is SI, then the branch to the `Final` block is taken. Otherwise, the branch to the `Loop` block is taken.

The `Loop` block performs the same vector shuffle and insert operations as in the general case described previously. The only difference is that instead of loading the data that is new for the next iteration in the loop (i.e. iteration `i+1`), the value loaded is for iteration `i+SI`.

Since we are shifting more than one iteration from the loop body, the `Final` block is now a loop itself. The number of iterations of this loop is determined by how much extra data is loaded in the `Preload` block (i.e. the value of SI). In the case of `averages<2>`, it performs the final three iterations of the loop. Like the `Loop` block, we insert vector shuffle instructions at the end of the `Final` block to shift the elements to the left for the next iteration of the loop. However, since all the data for these final iterations has already been loaded there are no additional load instructions or vector element insert instructions necessary. With each shuffle, the "last" element in the vector becomes undefined. It does not matter what value is inserted into this element of the vector since it won't ever be used.

4.3.3. Design and Implementation

In this section, we will describe the design and implementation of the loop shifting optimization using the example function `maskedSum` shown in Figure 48. This function applies a mask to four elements of the input array `in` and stores the sum of these values to an element of the output array `out`. This is performed `size` times. The mask for each element is stored in the array `mask` and is the same on each iteration of the loop.

```
void maskedSum (int * __restrict in,
                int * __restrict mask,
                int * __restrict out, unsigned int size)
{
  for (unsigned int i = 0; i < size; ++i) {
    int sum = 0;
    for (unsigned int j = 0; j < 4; ++j)
      sum += in[i+j] & mask[j];
    out[i] = sum;
  }
}
```

Figure 48 maskedSum example function

The loop shifting optimization requires modifications to be made to two of the steps in the original SLP vectorizer. The following steps are modified:

- Generate starting groups
- Vectorize chains

4.3.3.1. Generate Starting Groups

The process involved in generating starting groups is unchanged for the loop shifting optimization. However, we do add an additional post-pass on the generated groups to annotate each group with information pertaining to the “iteration overlap” of that group.

```
for.preheader:
  %mask1 = getelementptr inbounds i32, i32* %mask, i32 1
  %mask2 = getelementptr inbounds i32, i32* %mask, i32 2
  %mask3 = getelementptr inbounds i32, i32* %mask, i32 3
  br label %for.body

for.body:
  %i = phi i32 [ %add1, %for.body ],
               [ 0, %for.preheader ]
  %arrayidx0 = getelementptr inbounds i32, i32* %in, i32 %i
  %load0 = load i32, i32* %arrayidx0, align 4, !tbaa !1
  %load1 = load i32, i32* %mask, align 4, !tbaa !1
  %and0 = and i32 %load1, %load0
  %add1 = add nuw i32 %i, 1
  %arrayidx1 = getelementptr inbounds i32, i32* %in, i32 %add1
  %load2 = load i32, i32* %arrayidx1, align 4, !tbaa !1
  %load3 = load i32, i32* %mask1, align 4, !tbaa !1
  %and1 = and i32 %load3, %load2
  %accumulate0 = add nsw i32 %and1, %and0
  %add2 = add i32 %i, 2
  %arrayidx2 = getelementptr inbounds i32, i32* %in, i32 %add2
  %load4 = load i32, i32* %arrayidx2, align 4, !tbaa !1
  %load5 = load i32, i32* %mask2, align 4, !tbaa !1
  %and2 = and i32 %load5, %load4
  %accumulate1 = add nsw i32 %and2, %accumulate0
  %add3 = add i32 %i, 3
  %arrayidx3 = getelementptr inbounds i32, i32* %in, i32 %add3
  %load6 = load i32, i32* %arrayidx3, align 4, !tbaa !1
  %load7 = load i32, i32* %mask3, align 4, !tbaa !1
  %and3 = and i32 %load7, %load6
  %accumulate2 = add nsw i32 %and3, %accumulate1
  %arrayidx7 = getelementptr inbounds i32, i32* %out, i32 %i
  store i32 %accumulate2, i32* %arrayidx7, align 4, !tbaa !1
  %exitcond = icmp eq i32 %add1, %size
  br i1 %exitcond, label %for.loopexit,
                   label %for.body
```

Figure 49 LLVM-IR for the for loop in the function maskedSum

Starting with the LLVM-IR shown in Figure 49, we generate two starting groups for the basic block labelled `%for.body`. These groups (with `VF` equal to four) are:

```
Load Group 0:
%load0 = load i32, i32* %arrayidx0, align 4, !tbaa !1
%load2 = load i32, i32* %arrayidx1, align 4, !tbaa !1
%load4 = load i32, i32* %arrayidx2, align 4, !tbaa !1
%load6 = load i32, i32* %arrayidx3, align 4, !tbaa !1

Load Group 1:
%load1 = load i32, i32* %mask, align 4, !tbaa !1
%load3 = load i32, i32* %mask1, align 4, !tbaa !1
%load5 = load i32, i32* %mask2, align 4, !tbaa !1
%load7 = load i32, i32* %mask3, align 4, !tbaa !1
```

`Load Group 0` contains the load instructions from the input array `in`. `Load Group 1` contains the load instructions from the input array `mask`. There are three methods that we can use to calculate the memory access overlap between iterations for each group of instructions. Each method involves inspecting the pointer operand to the first LLVM-IR load instruction in the group (e.g. inspecting the instructions which generate the value `%arrayidx0` for the first load group). We only consider the pointer operand of the first instruction since the remaining instructions in the group are part of the group only because they access consecutive regions of memory. Therefore, by inspecting the first pointer operand, we are implicitly inspecting the other pointer operands as well.

The first method is applied when the pointer operand is generated by a `getelementptr` instruction that is in the loop body, with a variable value for its index operand. This is the method that is used for the first load group in our example. Consider the instruction that generates the pointer value for the first load in this group.

```
%arrayidx0 = getelementptr inbounds i32, i32* %in, i32 %i
```

This instruction has a base pointer of `%in`, which is indexed using the value `%i`. For the index operand, we are looking for one of two things. We require that the index operand is either the value produced by the PHI instruction for the loop's iterator or the value produced by an integer `add` instruction with its first input operand being the PHI instruction for the loop's iterator. If the instruction that generates the index operand is an `add` instruction, then we require that it have a constant integer value as its second input operand. These conditions are a hard requirement because we want to ensure the indexing on each iteration of the loop is some constant value offset from the loop's iterator value. If the "base" index value is not one of the loop's iterators, then we are unable to calculate a constant overlap between iterations. If the index operand is an integer `add` instruction with a variable second input operand, once again we cannot calculate a constant overlap between iterations.

If these requirements are met, then the overlap between iterations for this group is the difference between the size of the group and the constant value by which the iterator is updated on each iteration. We can use this approach since we do not support groups of data accesses with a stride greater than one. In our example, the size of `Load Group 0` is four instructions, and the iterator `%i` is updated by one on each iteration of the loop by the instruction that generates the value `%add1`. Therefore, the iteration overlap for this group is three. This means that on each iteration of the loop, there are three elements in the group which were already loaded from memory on previous iterations.

The second method is applied when the pointer operand is generated by a `getelementptr` instruction with a constant value for its first index operand. If the pointer operand is not generated by an instruction, or if it is generated by an instruction that is not executed as part of the loop, then the overlap between iterations is the full size of the group. We know this as neither the pointer operand or the index operand will change value between iterations of the loop.

The third and final method is applied when the pointer operand of the first load instruction in the group is not generated by an instruction or is generated by an instruction that is not executed as part of the loop. This is the method we apply for the second load group in our example. The pointer operand for this load group is `%mask`. This value is a function argument that remains the same value throughout the lifetime of the entire function. Because of this, we can say that the iteration overlap is the full size of the group (i.e. the group is loop invariant).

This final method is also applied when the pointer operand of the first instruction in the group is generated by another group of instructions which are also loop invariant. This is worth noting as loading pointer values from consecutive memory locations can be vectorized, as well as the uses of those pointers. This is a common occurrence when vectorizing instructions that operate on 2D arrays in C and C++.

4.3.3.2. Vectorize Chains

Before we can begin vectorizing instruction chains with loop shifting, we need to create the `Preload` and `Final` basic blocks. As described in section 4.3.1, these blocks are responsible for preloading data for the first iteration of the loop and all other instructions in the last iteration of the loop respectively. The LLVM-IR for the vectorized versions of these blocks are shown in Figure 50 and Figure 51.

We create these blocks by duplicating the loop body in full. We use the same LLVM function here to clone the loop body as we did in section 4.1.2.2 for the loop unroller. This function duplicates all the instructions in a basic block into a new block but it does not replace references to the original instructions within the new block. Because of this we need to do a pass over all instructions in the new basic block and replace all references to instructions in the original block with references to their corresponding instruction in the new block. The cloning function provides a "Value to Value" map to facilitate this. We will use the maps for both the `Preload` and `Final` blocks extensively when vectorizing each node in the DAG.

When these blocks have been created we fix up the branch instructions at the end of each affected block in the function's CFG. The branch instruction at the end of the original loop predecessor block (e.g. `%for.preheader` in Figure 49) is redirected to the `Preload` block instead of the loop body. The `Preload` block is set to branch to either the `Final` block or the loop body, depending on the number of iterations there are to be performed. An integer compare instruction is generated to drive this conditional branch. When exiting the loop body, the new branch target is the `Final` block. At the end of the `Final` block, we insert an unconditional branch to the previous target block of the loop exit.

```
for.preload:
  %newbound = add i32 %size, -1
  %invpt = bitcast i32* %in to <4 x i32>*
  %inpreload = load <4 x i32>, <4 x i32>* %invpt, align 4
  %maskvpt = bitcast i32* %mask to <4 x i32>*
  %vectormask = load <4 x i32>, <4 x i32>* %maskvpt, align 4
  %5 = icmp eq i32 %size, 1
  br i1 %5, label %for.final,
            label %for.body

for.body:
  %inphi = phi <4 x i32> [ %next, %for.body ],
                         [ %inpreload, %for.preload ]
  %i = phi i32 [ %add1, %for.body ],
               [ 0, %for.preload ]
  %andvectorized = and <4 x i32> %vectormask, %inphi
  %extract0 = extractelement <4 x i32> %andvectorized, i64 0
  %extract1 = extractelement <4 x i32> %andvectorized, i64 1
  %extract2 = extractelement <4 x i32> %andvectorized, i64 2
  %extract3 = extractelement <4 x i32> %andvectorized, i64 3
  %add1 = add nuw i32 %i, 1
  %accumulate0 = add nsw i32 %extract1, %extract0
  %accumulate1 = add nsw i32 %extract2, %accumulate0
  %add3 = add i32 %i, 3
  %arrayidx3 = getelementptr inbounds i32, i32* %in,
                                          i32 %add3
  %accumulate2 = add nsw i32 %extract3, %accumulate1
  %arrayidx7 = getelementptr inbounds i32, i32* %out, i32 %i
  store i32 %accumulate2, i32* %arrayidx7, align 4, !tbaa !1
  %exitcond = icmp eq i32 %add1, %newbound
  %nextidx = getelementptr i32, i32* %arrayidx3, i32 1
  %load4 = load i32, i32* %nextidx, align 4
  %shuffle = shufflevector <4 x i32> %inphi, <4 x i32> undef,
                 <4 x i32> <i32 1, i32 2, i32 3, i32 undef>
  %next = insertelement <4 x i32> %shuffle, i32 %load4,
                                           i64 3
  br i1 %exitcond, label %for.final,
                   label %for.body
```

Figure 50 LLVM-IR for the vectorized maskedSum function (blocks preload and body)

At this stage, we also adjust the loop bound of the loop body. We do this by creating a `sub` instruction in the `Preload` block, which subtracts the stride of the loop iterator from the original loop bound. The value produced by this `sub` instruction becomes the new loop bound. In our example function `maskedSum`, the iterator stride is one, so the new loop bound is `size-1`.

We also take this opportunity to tidy up any references to basic blocks in PHI instructions that have been affected by the introduction of the two new blocks in the CFG.

```
for.final:
  %infinal = phi <4 x i32> [ %inpreload, %for.preload ],
                           [ %next, %for.body ]
  %ifinal = phi i32 [ 0, %for.preload ],
                    [ %add1, %for.body ]
  %andvectorizedfinal = and <4 x i32> %vectormask, %infinal
  %16 = extractelement <4 x i32> %andvectorizedfinal, i64 0
  %17 = extractelement <4 x i32> %andvectorizedfinal, i64 1
  %18 = extractelement <4 x i32> %andvectorizedfinal, i64 2
  %19 = extractelement <4 x i32> %andvectorizedfinal, i64 3
  %accumulate0final = add nsw i32 %17, %16
  %accumulate1final = add nsw i32 %18, %accumulate0final
  %accumulate2final = add nsw i32 %19, %accumulate1final
  %arrayidx7final = getelementptr inbounds i32, i32* %out,
                                                i32 %ifinal
  store i32 %accumulate2final, i32* %arrayidx7final,
                               align 4, !tbaa !1
  br label %for.loopexit
```

Figure 51 LLVM-IR for the vectorized maskedSum function (block final)

Once these changes to the CFG have been made, we are ready to begin vectorizing individual nodes in the vectorizable DAG. This is the DAG described in section 3.2 that contains nodes which represent groups of vectorizable LLVM-IR instructions. As before, the nodes in the DAG are vectorized one at a time, in a top-down breadth-first fashion, starting with the root nodes of the graph. We make a distinction here between nodes with a non-zero iteration overlap, and those with an iteration overlap of zero.

Nodes with a non-zero iteration overlap are first inserted into the `Preload` block. The “Value to Value” map that was generated during the block cloning process earlier is used to determine the appropriate insert point in the `Preload` block for the vectorized instruction. In our example, the original load groups fall into this category.

The loads from `%in` are vectorized and inserted into the `Preload` block as the following instruction in Figure 50:

```
%inpreload = load <4 x i32>, <4 x i32>* %invpt, align 4
```

Depending on the value of the iteration overlap, we take one of two actions:

1. If the iteration overlap is the same value as the number of instructions in the node (as is the case for the loads from `%mask`) then we only insert the vectorized instruction into the `Preload` block and use it for all uses of the original values in subsequent blocks (the loop body and the `Final` block).
2. If the iteration overlap is any other value (like with the loads from `%in`), we also insert additional instructions into the loop body to load the appropriate values and setup the vector value for the next iteration of the loop. In our example in Figure 50, this action is performed by the following sequence of instructions.

```
%nextidx = getelementptr i32, i32* %arrayidx3, i32 1
%load4 = load i32, i32* %nextidx, align 4
%shuffle = shufflevector <4 x i32> %inphi, <4 x i32> undef,
                  <4 x i32> <i32 1, i32 2, i32 3, i32 undef>
%next = insertelement <4 x i32> %shuffle, i32 %load4,
                                          i64 3
```

The first step in this sequence of instructions is to calculate the pointer for the next value that is required to be inserted into the vector for the next loop iteration. To do this, we use the pointer for the last load instruction in the group as a base pointer and index it with a constant index value for the next value in the array (in this case, the value is one). In our example, there is only one scalar value to be loaded for the next loop iteration, but depending on the iteration overlap value, this could be any value greater than zero. When performing the more aggressive loading optimization, scalar values for the `N`th iteration after this iteration are loaded, where `N` is the number of iterations shifted from the loop.

Next, we shuffle the vector value used in the current loop iteration to effectively shift the elements to the left by however many scalar values are being loaded. In our example, there is only one scalar, so we shift the elements left by one. This creates an undefined "hole" in the vector into which we can insert our new values. The values are inserted, and the produced vector value is used for the next iteration of the loop.

Finally, a PHI instruction at the start of the loop body is created for this node. When entering the loop block from the `Preload` block, the value produced by the vector instruction inserted into that block is used. When entering from the back-edge of the loop, the value produced by the final vector element insert instruction is used. Similarly, a PHI instruction is also inserted into the `Final` block for this node. When entering the block from the `Preload` block, the value produced by the vector instruction inserted into that block is used. When entering from the loop body, the value produced by the last vector element insert instruction is used.

For nodes with an iteration overlap of zero, the vectorized instructions are inserted into both the loop body and the `Final` blocks. The "Value to Value" map for the `Final` block is used to determine the appropriate position in the block to insert new vector instructions. In our example, there are a group of logical `and` instructions which are vectorizable with no iteration overlap as shown in Figure 49. The vector instruction in the loop body is created in the same fashion as before. When an input operand to the vectorized instruction is produced by a node with a non-zero iteration overlap (that is not loop invariant), the PHI instruction at the start of the loop body for that node is used as the value for that input operand. The same rules apply to the vector instruction inserted into the `Final` block, the PHI instruction at the start of the block for that node is used.

Once all nodes have been vectorized, we delete any unused scalar instructions in all three blocks and all scalar instructions that have been vectorized in all three blocks. Any scalar store instruction that have been inserted into the `Preload` block are also deleted at this point. They were duplicated into the `Preload` block during the cloning step and should not persist there.

This optimization, along with all other optimizations described in this chapter are demonstrated working together in tandem in the following chapter using the example of an image convolution.

Chapter 5 Vectorization of Image Convolutions

In this chapter, we bring together all aspects of the vectorizer and its additional optimizations described in Chapter 3 and Chapter 4. We do this through an example function. Our example function is a 5x5 image convolution that operates on 32-bit integer values. The code for this function is shown in Figure 52.

```
void convolution5x5 (int * __restrict * __restrict image,
                     int * __restrict * __restrict kernel,
                     int * __restrict * __restrict output,
                     unsigned int width,
                     unsigned int height ) {
  for (unsigned int i = 0; i < height; ++i) {
    for (unsigned int j = 0; j < width; ++j) {
      int sum = 0;
      #pragma unroll 5
      for (unsigned int x = 0; x < 5; ++x) {
        #pragma unroll 5
        for (unsigned int y = 0; y < 5; ++y) {
          sum += image[i+x-2][j+y-2] * kernel[x][y];
        }
      }
      output[i][j] = sum;
    }
  }
}
```

Figure 52 convolution5x5 example function

For the function `convolution5x5` we know that the input arrays `image` and `kernel`, and the output array `output` are all non-overlapping since they are declared with the `__restrict` keyword as described in section 2.5. This function assumes that the input array `image` has been pre-padded with appropriate values on all sides so that we can negatively index the array in the first iterations of both the inner and outer for loops. This is also so we can safely read over the end of the image for the final iterations of both loops. This is necessary since we read more values from the `image` array than are written to the `output` array, as discussed in section 2.6. Effectively, when the `output` array is $N \times M$, the `image` array is pre-padded to $(N + K - 1) \times (M + K - 1)$, where K is the size of the kernel.

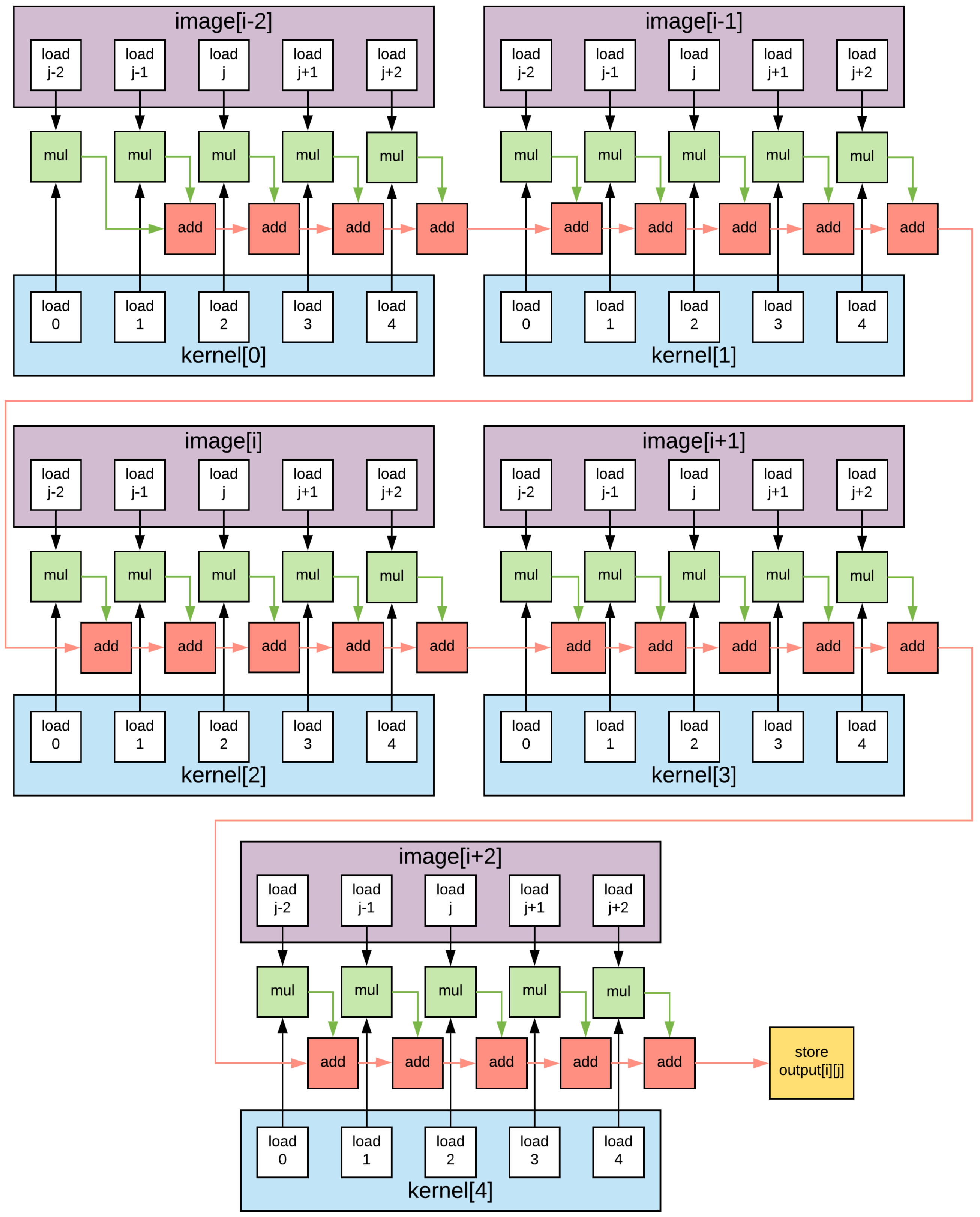


Figure 53 convolution5x5 original scalar instruction def-use chain

The original scalar instruction chain for the function is shown in Figure 53. The instructions in this function have been grouped into sections in this image. Load instructions accessing the `image` input array are highlighted in the purple groups. Load instructions from the `kernel` input array are highlighted in blue. Integer multiply instructions are shown in green and the accumulation chain (using the variable `sum` in Figure 52) is shown in peach.

When vectorizing this function, the SLP vectorizer, the Associative Chain Reordering and the loop shifting optimizations all work together in tandem.

We start by generating starting groups for the body of the inner-for loop. There are ten starting groups for this basic block. They are the five groups of load instructions from the `image` array shown in purple in Figure 53, and the five groups of load instructions from the `kernel` array shown in blue. Each of these groups has a non-zero iteration overlap, for the purposes of the loop shifting optimization. The index operands for the `kernel` accessing instructions are all constant integer values. This means that each of these five groups is loop invariant since neither the index values nor the pointer values change between iterations of the loop. Each of the `image` accessing groups have an iteration overlap of four. This is because each group in the 5x5 convolution accesses five consecutive locations of memory from their respective rows of the `image` array on each iteration of the loop using the iterator `j` as a base index value. The iterator `j` is updated on each iteration of the loop by constant one. Therefore, four of the memory locations accessed on iteration `j` were already accessed on iteration `j-1`, for all iterations `j>0`.

From these starting groups, we build the rest of the DAG for the basic block. The rest of the DAG is made up of two parts. The first part contains the integer multiply instructions shown in green in Figure 53. These instructions are brought together into five groups of five instructions each. The instructions are grouped based on their input operands from the `image` and `kernel` load groups using the process described in section 3.2.3. The second part of the DAG contains the multiple-input `add` instruction corresponding to the accumulation chain shown in peach in the graph. The output value of this mutliple-input `add` instruction is a scalar value which is used as the input operand to the final store instruction to the `output` array shown in yellow. We flatten this accumulation chain to a single multiple-input instruction to break the data-dependencies between the individual `add` instructions and allow them to be vectorized.

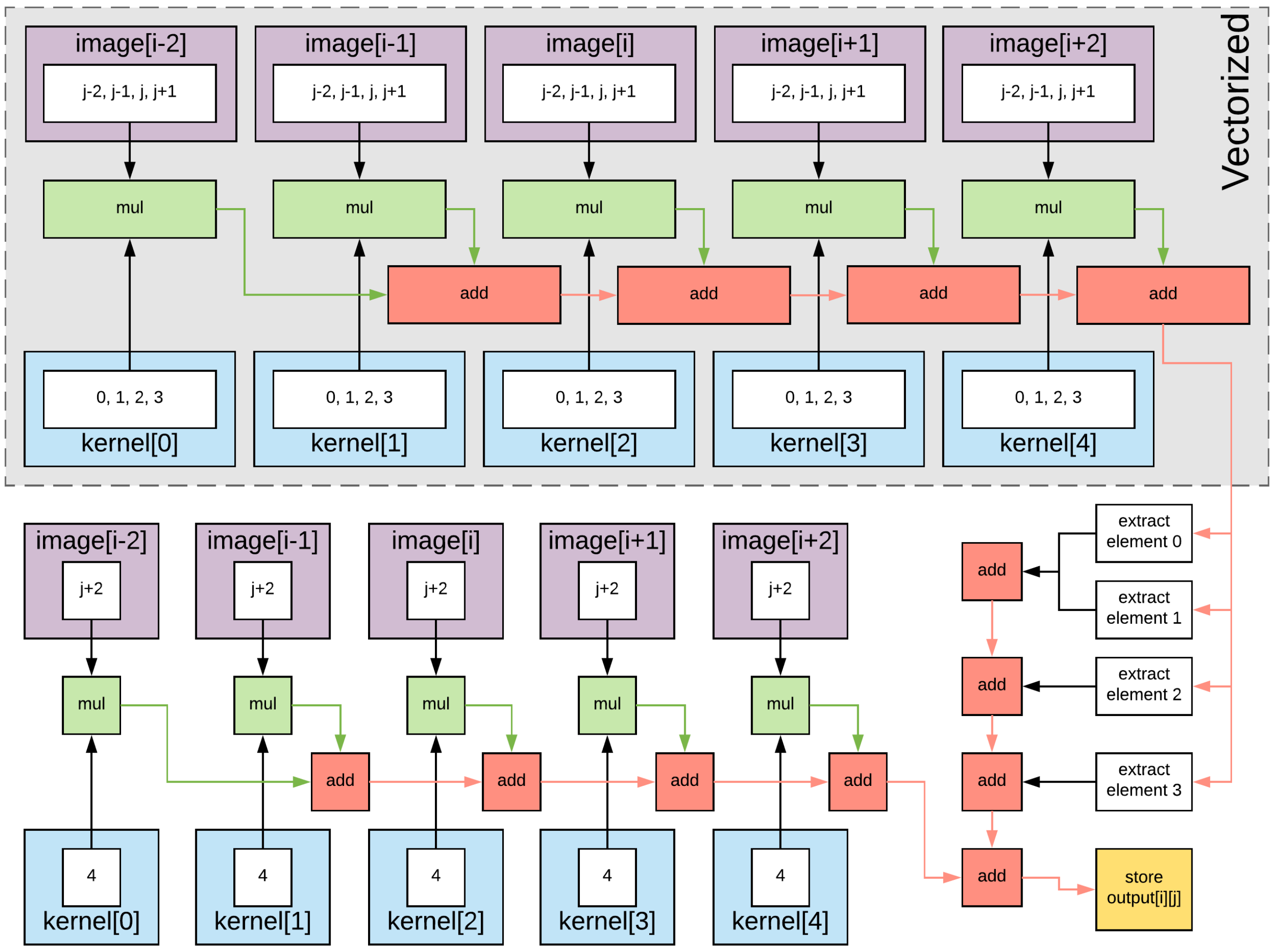


Figure 54 convolution5x5 vectorized instruction def-use chain

With this graph built, we can move on to vectorizing each individual group of instructions. The final vectorized instruction chain is shown in Figure 54. It should be noted that the image and kernel groups shown in this graph are not load instructions. They are only there to show which values are being used by which nodes in the graph. Where the load instructions are inserted is explained later using Figure 55.

There is one issue which we resolve before producing any vector instructions. The size of our groups of instructions is five. This means we would have `v5i32` vectors if we began vectorizing them immediately. Our target architecture does not support this vector type, but it does support `v4i32` vectors. Therefore, we remove the last instruction from each group and so that their original scalar instructions remain in the block and we vectorize only the first four as shown in Figure 54.

Reducing the size of the groups of instructions in this way presents a problem for the multiple-input `add` instructions which we generated for the chain of scalar `add` instructions in the original function. All input operands to this instruction no longer come from vectorized nodes. Because of this, we generate two separate chains of `add` instructions, one scalar and one vector, and join them together at the end. The vector part of the `add` chain contains four vector `add` instructions. The scalar part of the `add` chain contains four scalar `add` instructions. The number of instructions in both chains is determined by the convolution size. These two chains are joined together using an additional four (`VF`) scalar `add` instructions. First, we extract each of the elements from the value produced by the last instruction in the vector chain and produce three `add` instructions to accumulate them. These instructions may be lowered to a horizontal `add` instruction by the target backend. The result of this accumulation is added to the result of the scalar chain to produce the final, scalar output of the entire chain. This value is used as the input operand to the store instruction to the `output` array.

These chains of instructions do not exist in only one basic block. With the loop shifting optimization, parts of the chain are duplicated and split between the loop body, the `Preload` block, and the `Final` block. The structure of the final instruction chain as it exists across all three of these basic blocks is shown in Figure 55.

As we mentioned previously, the groups of load instructions from the `kernel` array are loop invariant. Because of this, we can hoist all of these loads into the `Preload` block and replace all uses of the original scalar loads with the values produced by these hoisted instructions. The uses of these values are shown by the blue edges in the graph in Figure 55.

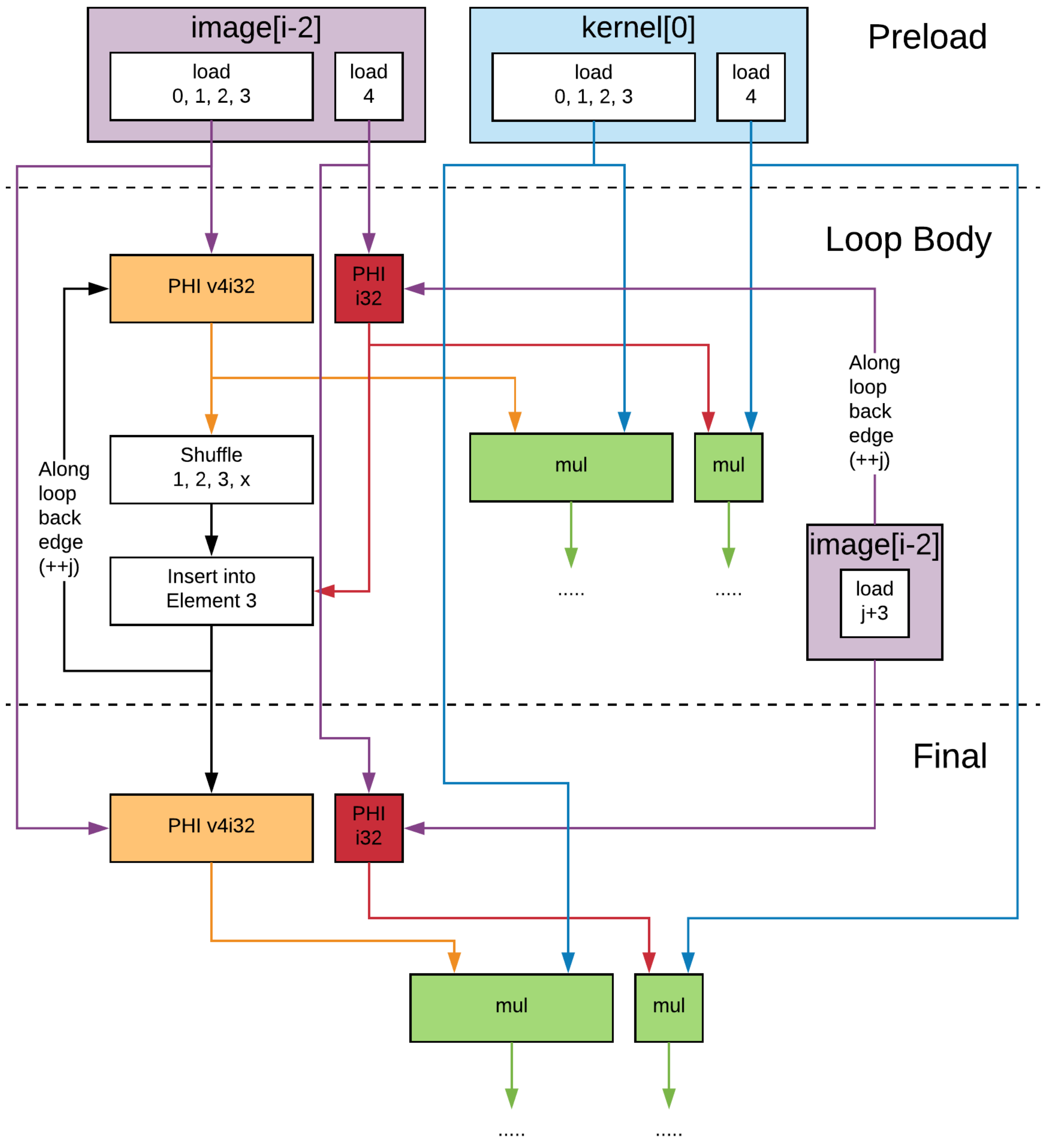


Figure 55 convolution5x5 instruction def-use chain snippet across the Preload, Loop Body, and Final blocks

The groups of load instructions from the `image` array also have data re-use between iterations. We start here by shifting the loads for the first iteration of the loop into the `Preload` block. Even though we are only vectorizing four of the instructions from the original group, we still hoist the last scalar instruction left over since it is also re-used between iterations. We insert two PHI instructions per group into the loop body, a vector PHI and a scalar PHI. These PHI instructions are responsible for choosing the correct value for this iteration of the loop (depending on which edge we took to enter the block). On the first iteration the value loaded in the `Preload` block is produced by the PHI, and on each subsequent iteration the value produced on the previous iteration is used.

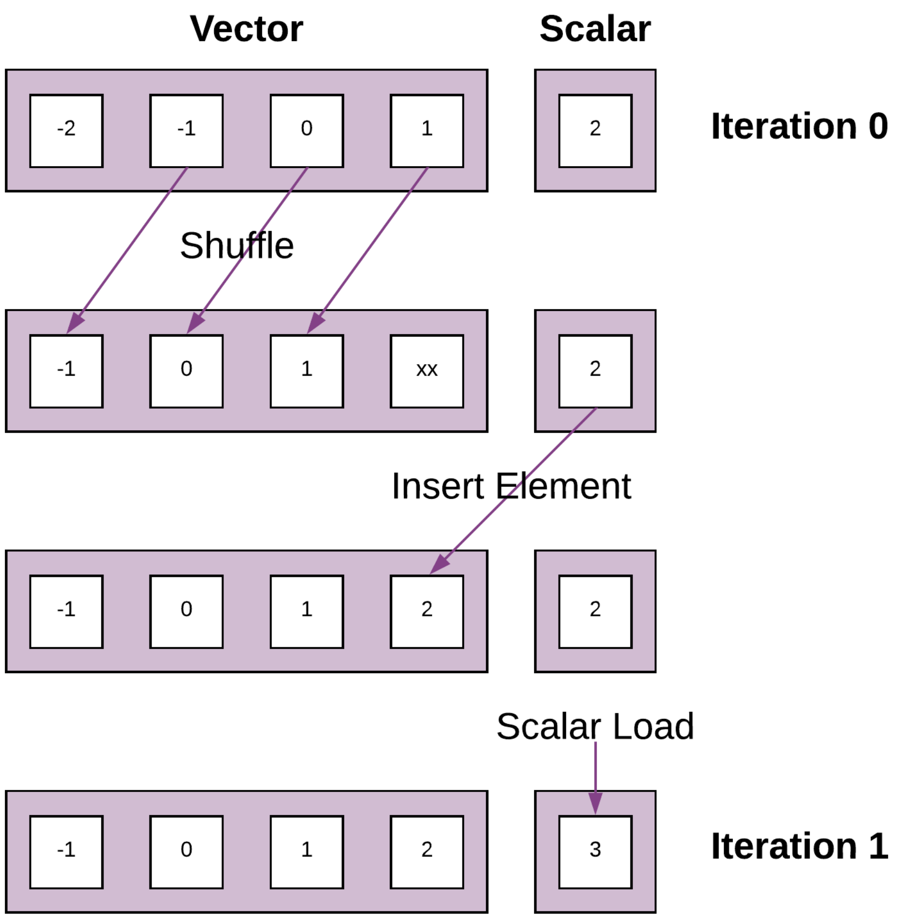


Figure 56 Instruction sequence that prepares the input data for the next loop iteration

On each iteration of the loop, there are four values already loaded from the `image` array that we can re-use on the next iteration of the loop. In order, to re-use the data we need to move it into the correct position in the vector register in which it is stored. As shown in Figure 56, this involves shuffling the vector value that we used on this iteration to shift the elements to the left by one. We then insert the scalar value we used into last element position of the shuffled vector. This is the vector value which we will use on the next iteration of the loop and is passed along the back-edge of the loop using the vector PHI instruction we inserted earlier. Now, we only need to load the extra piece of data for the next iteration that was not used in this iteration. This involves inserting an additional scalar load instruction at the end of the loop body that is passed along the back-edge of the loop using the scalar PHI instruction we inserted earlier.

For the final iteration of the loop which is executed in the `Final` basic block, in this example all data has already been loaded so there is no need to insert any additional load instructions. We only need to insert two PHI instructions per group to distinguish between the values used when entering the block from the `Preload` block and when entering from the loop body.

The multiply instructions and the accumulation instructions shown in Figure 54 are duplicated in both the loop body and the `Final` block.

With this process complete, we have effectively vectorized the inner-for loop of the function `convolution5x5`. Our technique leverages the associativity and commutativity of the integer `add` instruction to maximize the number of vector instructions that we can produce for the loop body. We also used a loop shifting technique to re-use data across loop iterations to prevent excessive reading from memory. In the following chapter, we will discuss the performance improvements that are achieved through these techniques in the case of image convolutions and several other example benchmarks.

Chapter 6 Evaluation

6.1. Evaluation Setup

Our vectorizer was implemented and integrated into the LLVM 4.0.0 version of the Movidius SHAVE C/C++ Compiler. This is the implementation we will use to evaluate the correctness and performance of our optimizations.

All functional and performance results which are presented here were generated by running each test on a single SHAVE core of a Movidius Myriad 2 MA2150 SoC, operating on data contained in the 2MB of CMX memory described in section 2.2. SHAVE has a 128-bit vector unit with support for 8, 16, and 32-bit integer operations and 16, and 32-bit floating point operations. We will consider the performance gains and losses generated by our vectorizer in relation to each of these types.

The implementation of our vectorizer was functionally verified using two groups of different test suites. First, a group of correctness-oriented tests were used: LLVM's own libc++ test suite and the GCC test suite. In total, approximately 11,200 tests from these test suites were used to verify correctness. These are the same tests that are among those used to ensure the correctness of every release Clang/LLVM and GCC.

The second group of tests consist of purpose-built image processing tests which we created for this thesis. These tests were used both for ensuring correctness of each individual optimization and measuring their performance. A subset of these purpose-built image processing tests is also used to gauge the changes in performance generated by our vectorizer in the next section. The tests used to verify the functionality of the implementation are shown in the table in Figure 57.

The image convolutions listed operate on all natively supported types on the target architecture. In total, there are 194 of these tests, accounting for all permutations of the input and output types. The box blur tests also operate on all natively supported types, with a 1:1 mapping of input and output types. There are 15 of these tests in total.

All the example functions used in the previous chapters of this thesis were also used to verify the correctness of the implementation.

Test	Input types	Output Types	Input size (elements)
5x5 image convolution	int8_t, int16_t, int32_t, uint8_t, uint16_t, uint32_t, half, float	int8_t, int16_t, int32_t, uint8_t, uint16_t, uint32_t, half, float	1080
7x7 image convolution	int8_t, int16_t, int32_t, uint8_t, uint16_t, uint32_t, half, float	int8_t, int16_t, int32_t, uint8_t, uint16_t, uint32_t, half, float	1080
9x9 image convolution	int8_t, int16_t, int32_t, uint8_t, uint16_t, uint32_t, half, float	int8_t, int16_t, int32_t, uint8_t, uint16_t, uint32_t, half, float	1080
5x5 image box blur	int8_t, int16_t, int32_t, half, float	int8_t, int16_t, int32_t, half, float	1080
7x7 image box blur	int8_t, int16_t, int32_t, half, float	int8_t, int16_t, int32_t, half, float	1080
9x9 image box blur	int8_t, int16_t, int32_t, half, float	int8_t, int16_t, int32_t, half, float	1080
Absolute difference	int8_t	int8_t	1024
Accumulate squared	uin8_t	uint8_t	1920
White balance	uint16_t	uint16_t	1920
Hamming distance	[uint8_t x 32], [uint8_t x 64], [uint8_t x 128]	uint32_t	512
Substring search	int8_t	[int8_t x 7], [int8_t x 11], [int8_t x 16], [int8_t x 21]	8192

Figure 57 table of purpose-built image processing tests

All tests in both groups of test suites were compiled using the same set of compiler options on the command-line:

```
-Wall -O3 {vectorizer enable/disable options}
```

The final set of options are those used to selectively enable and disable our vectorizer implementation.

6.2. Functional Results

For both the first group of 11,200 tests and the second group listed in Figure 57, there were performance changes detected in many of the tests, however there were no functional changes across all tests when the vectorizer was enabled compared to when it was disabled. This means that tests which passed with the vectorizer disabled also passed when the vectorizer was enabled, for all tests in each of the test suites. Passing this very large number of tests is a strong indication that the novel optimizations preserve the semantics of the compiled code.

6.3. Performance Results

In this section we will consider the performance gains and losses generated by our vectorizer implementation using a subset of the image processing tests mentioned in the previous section. We will evaluate the performance of each component of the vectorizer individually, and then in the final subsection consider the performance of all components working together as described in Chapter 5.

Throughout this section, we describe the performance changes in each test case in terms of “operations per cycle”. Within this context, operations refer to the operations directly involved in computing the results of the test. They do not include the likes of loop iterator updates, branch instructions, and any other “boiler-plate” instructions that are part of the test. We are only interested in the operations that directly perform the relevant work.

Since SHAVE is a VLIW architecture, the code generated by the compiler is a cycle by cycle listing of the instructions that will be executed at runtime, in the exact order they will be executed. There is no instruction re-ordering or pipeline changes made by the processor at runtime. Because of this, the number of cycles executed is an accurate and reliable measure of the efficacy of compiler optimizations. We use “operations per cycle” as a measure of performance in this thesis because of this.

In each subsection, we present the performance results and code-size changes for a set of tests chosen specifically for the optimization being discussed in that subsection.

This is followed by an analysis of these results. The tests were chosen based on how well they demonstrate the associated optimization. For example, the tests used to benchmark ACR do not have data re-use between iterations, so they are not used to benchmark the loop shifting optimization.

6.3.1. Superword Level Parallelism

To evaluate the performance of our Superword Level Parallelism (SLP) vectorizer (described in Chapter 3) on its own, we will consider the three following test cases:

- Absolute difference
- Accumulate squared
- White balance

The operations per cycle for each of these functions with and without the vectorizer enabled are shown in Figure 58. The code size changes generated by the vectorizer are shown in Figure 59.

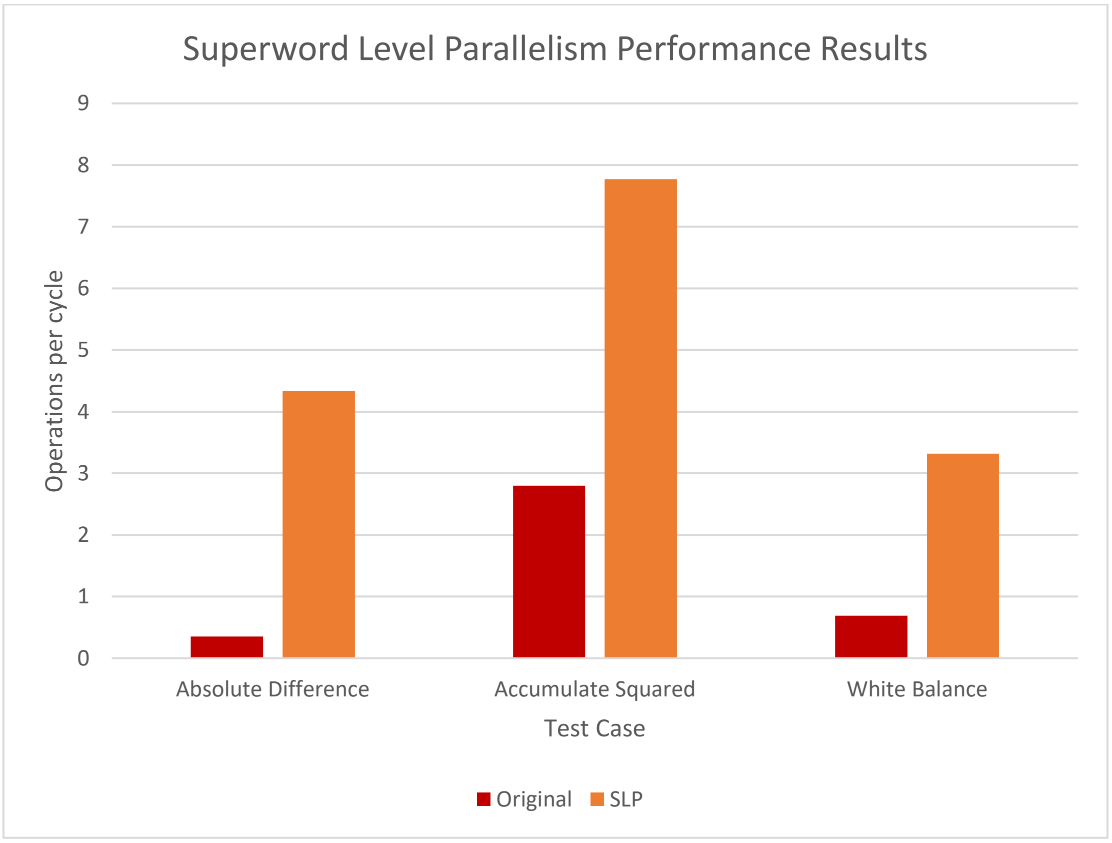


Figure 58 SLP vectorizer operations per cycle results with NVF of 16, 16, and 8 respectively

The absolute difference implementation computes the pixel-by-pixel absolute difference between two RGB images with an input and output type of `unsigned char`. With the vectorizer disabled, this implementation runs at rate of 0.355186 operations per cycle. When the vectorizer is enabled this jumps up to 4.331641 operations per cycle, a 12.1954x improvement. This comes at the cost of an increase in code size of 3.25x.

The accumulate squared implementation computes the square of each pixel in an input image and adds this value to the equivalent pixel of an output image. With the SLP vectorizer disabled, this implementation achieves an operations per cycle rate of 2.797619. With the vectorizer enabled, the implementation runs at 7.767258 operations per cycle, a 2.7763x improvement. This improvement is accompanied by an improvement in code size. The code size is nearly halved.

Finally, the white balance test case produces a white balanced version of an input RGB image. With the vectorizer enabled, there is a 4.8244x improvement in the operations per cycle, going from 0.687968 with the vectorizer disabled to 3.319053 with it enabled. This comes at the cost of a 1.0867x increase in code size.

While all three test cases examined here see an improvement in operations per cycle, the code size results are less consistent. The accumulate squared test can be vectorized without the need for loop unrolling. This is not the case for the absolute difference and white balance tests however. Both of these tests require loop unrolling for the SLP vectorizer to be able to vectorize them. The increase in code size is caused by the additional scalar loop that is generated for when the number of iterations of the loop is not a multiple of the unroll factor.

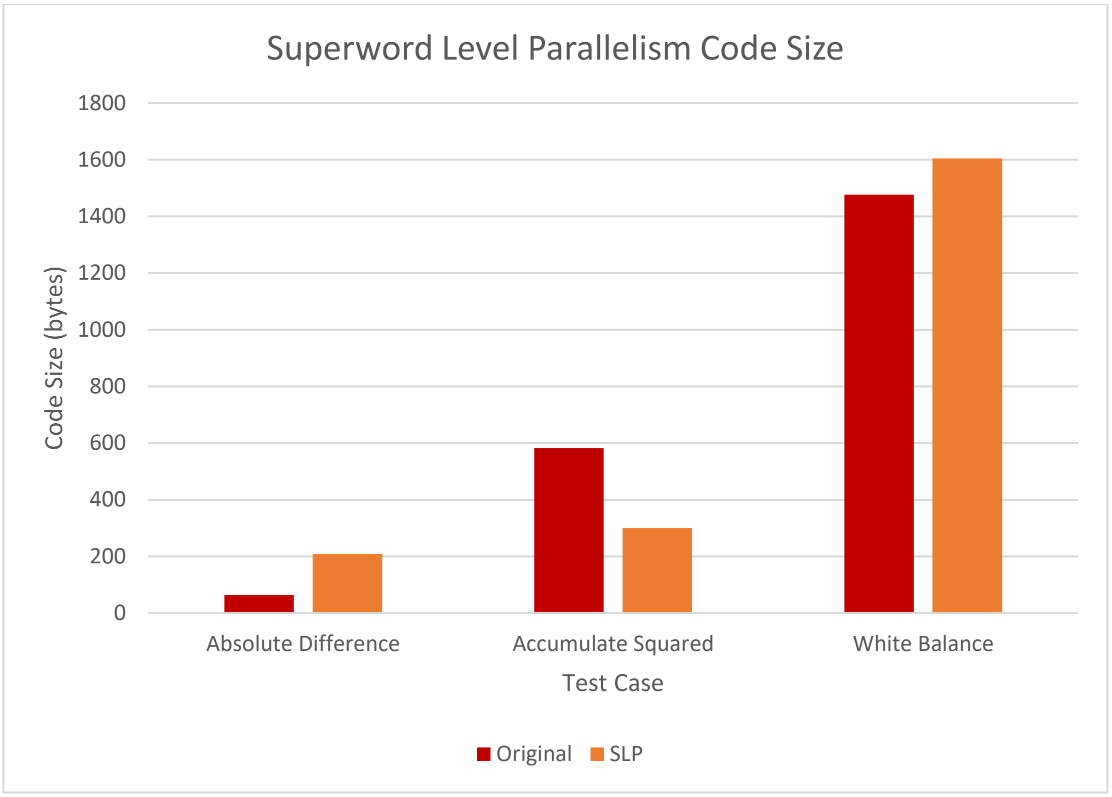


Figure 59 SLP vectorizer code size results

6.3.2. Associative Chain Reordering

When evaluating the performance of the SLP vectorizer with the Associative Chain Reordering (ACR) optimization enabled, we will consider a group of Hamming distance functions, operating on three different string sizes.

Hamming distance computes the number of points at which two strings of equal length have different bit values. Our Hamming distance implementation performs this operation for an array of byte strings, comparing each individual string in the array to a single reference string. Each string in the array and the reference string have the same length. The result for each string is stored to an output array of Hamming distance values.

We will consider the performance benefits of ACR in the context of 32, 64, and 128-byte string sizes. The operations per cycle results are shown in Figure 60, and the code size results are shown in Figure 61.

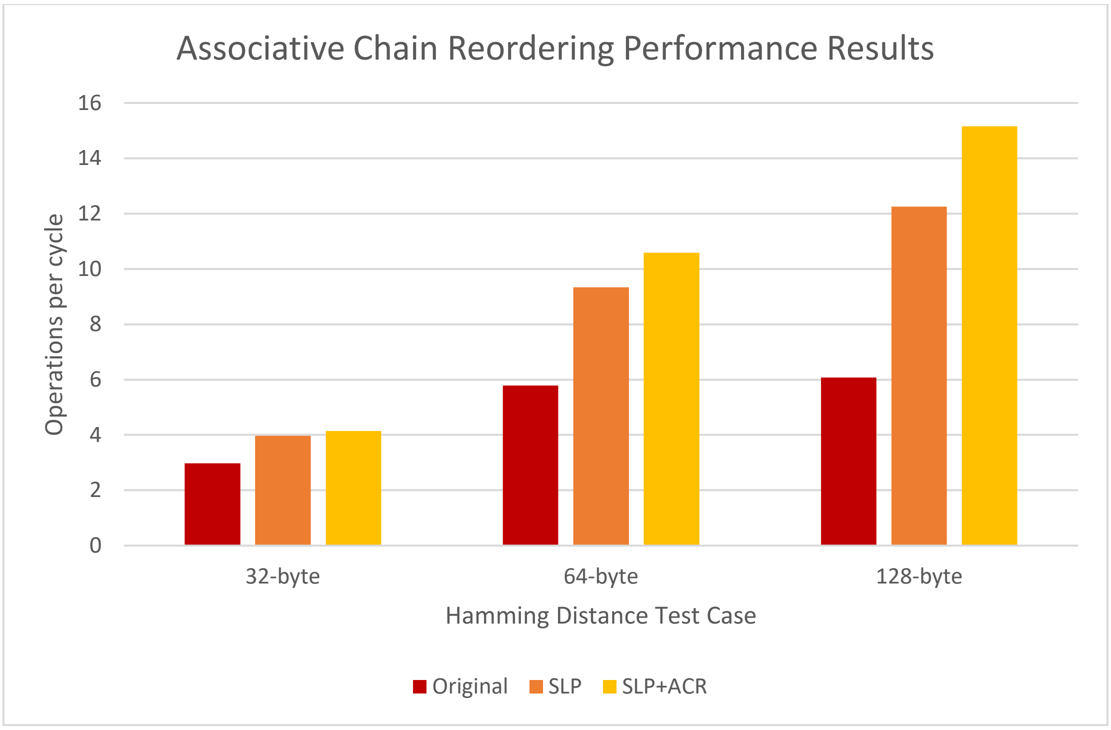


Figure 60 SLP vectorizer with ACR operations per cycle results, with NVF of 16

Starting with the 32-byte Hamming distance implementation, we see some decent performance improvements with both SLP and ACR enabled. With only SLP enabled, the operations per cycle jumps from 2.981439 to 3.969313, a 1.3313x performance improvement. With ACR enabled as well, there is a further increase to 4.139464 operations per cycle, a 1.3884x improvement on the original function. With SLP enabled, the code size remains the same for this test. However, with ACR enabled as well, there is a small reduction in code size from 208 bytes to 192 bytes.

Moving on to the 64-byte Hamming distance implementation, there is a more significant improvement in performance. With SLP enabled, the operations per cycle increases from 5.79247 to 9.342533, a 1.6128x improvement. With ACR enabled as well, this increases again to 10.58466 operations per cycle, a 1.8273x improvement. For this implementation, we also see a more significant reduction in code with SLP enabled on its own, and with both SLP and ACR enabled.

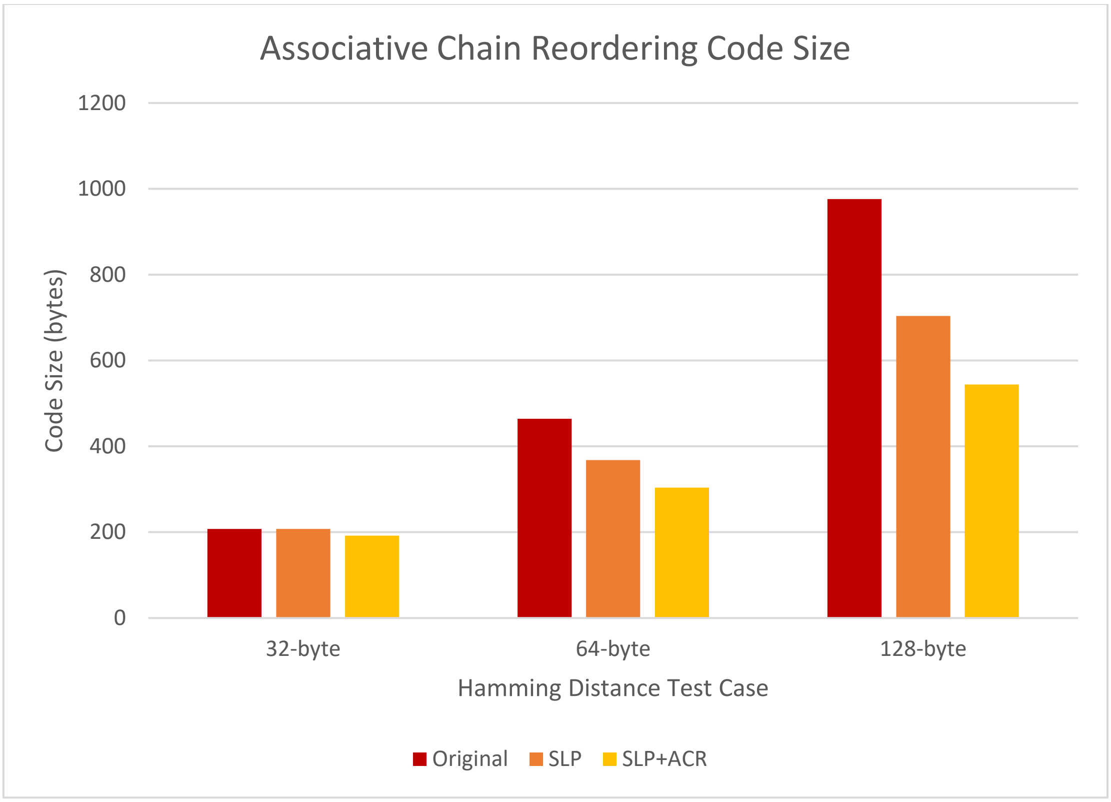


Figure 61 SLP vectorizer with ACR code size results

Finally, the 128-byte Hamming distance implementation sees a more significant improvement in performance again. With SLP enabled, the operations per cycle goes from 6.080309 to 12.24835, a 2.0144x improvement. This increases again with ACR enabled to 15.15423 operations per cycle. This signifies a 2.4923x improvement on the original function. Once again, there are more significant reductions in code size in this implementation than in the smaller Hamming distance implementations.

As the size of the strings which we are computing the Hamming distance of increase, so too do the performance improvements achieved by both SLP and ACR. The increased performance improvements through SLP are caused by more opportunities for instruction level parallelism in the assembly code generated by the compiler. When the size of the strings increases, so too do the number of independent instructions required to compute the Hamming distance. Many of these additional instructions are not data dependent on one another, so they can be executed in parallel (generating a higher instruction level parallelism). Because of this, the number of operations performed is doubled compared to the previous smaller test, but the number of cycles required to perform those operations increases by a smaller amount.

The performance improvements achieved through ACR also increase with string size. This is due to a longer accumulation chain which is used to sum all of the population count operations which are performed for each 4-byte section of the string (using LLVM's `llvm.ctpop.32` intrinsic). There are static costs associated with generating vector accumulation chains (e.g. retrieving the scalar result at the end). As the size of the chain increases, the relative cost of these to the rest of the chain decreases.

There is also a notable reduction in code size as the size of the strings increases. As we mentioned previously, as the size of the strings increase, so too does the availability of higher instruction level parallelism (ILP). On a VLIW architecture like SHAVE, a higher ILP can often equate to denser VLIW instructions (i.e. more operations encoded per instruction) and fewer "No Operation" or NOP instructions. This has the effect of reducing code size.

6.3.3. Data Re-use through Loop Shifting

When evaluating the performance of the SLP vectorizer with the Loop Shifting (LS) optimization enabled, we will use a set of substring search tests, each operating on different substring sizes. For each test, we will consider both the "basic" loop shifting optimization which performs at most one iteration of look-ahead and the more aggressive optimization which can perform more than one iteration of look-ahead.

There are four substring search tests in total, operating on seven, eleven, sixteen and twenty-one character strings respectively. Each of these tests finds the index of each instance of the specified substring in a single string of 8,192 characters. The generated performance results are shown in Figure 62, and the code size results are shown in Figure 63.

Each of these tests can be broadly split into three categories: undersize, native size and oversize. The 7-byte and 11-byte are undersize tests since they operate on substrings which have a length that is less than the native vector width of our target architecture. The 16-byte test is native size, and the 21-byte test is oversize since it operates on substrings larger than the native vector width.

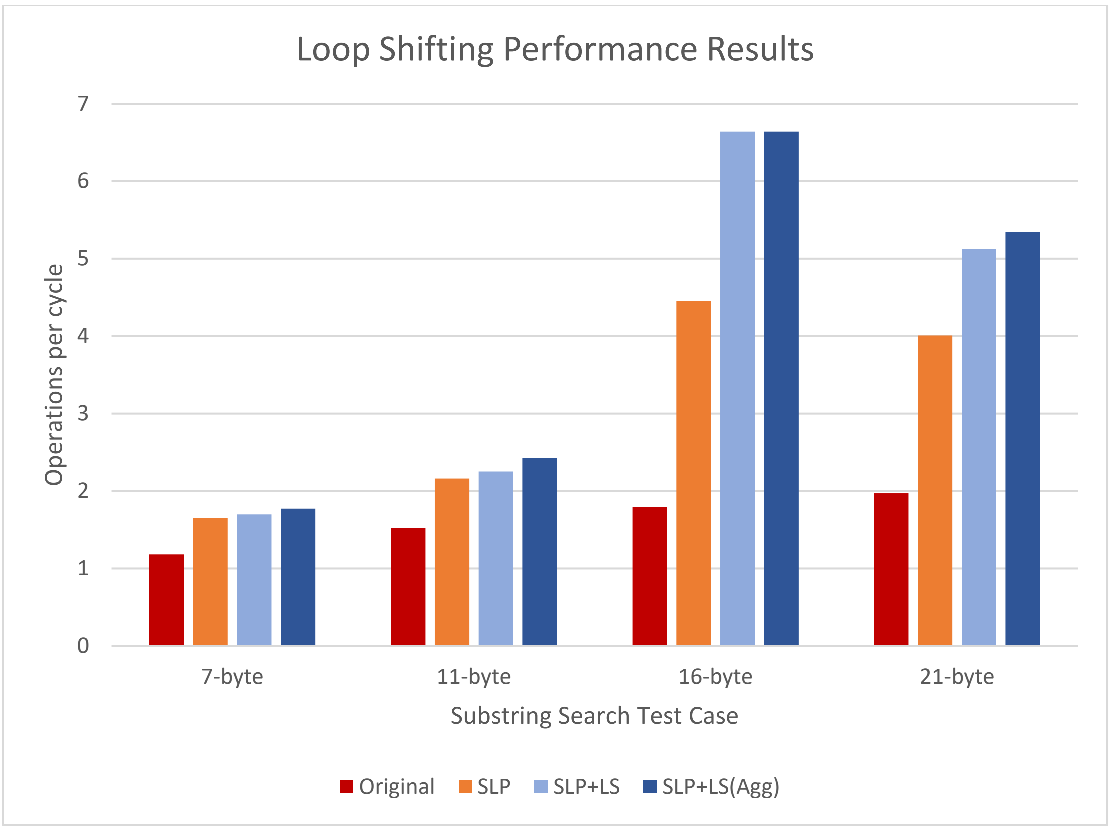


Figure 62 SLP vectorizer with LS operations per cycle results, with NVF of 16

Starting with the two undersize tests, we see some decent improvements when SLP is enabled. With it enabled, the 7-byte test improves in performance, going from 1.179427 operations per cycle to 1.652561 operations per cycle. This is a 1.4012x improvement. Similarly, the 11-byte test increases from 1.519055 operations per cycle to 2.16041 operations per cycle, a 1.4222x improvement.

With the data re-use optimization enabled, there are some modest improvements on the performance in each case. With basic loop shifting, the 7-byte test achieves a performance of 1.695184 operations per cycle, while the aggressive approach achieves 1.772855 operations per cycle. These represent a 1.4373x and a 1.5031x improvement respectively. The basic approach for the 11-byte test reaches 2.249568 operations per cycle, while the aggressive approach achieves 2.422804 operations per cycle. These represent a 1.4809x and a 1.5949x improvement respectively.

The performance improvement for loop shifting on these undersize tests may appear lower than one might expect. This optimization can be a double-edged sword for certain types of functions. While we do get the benefits of data re-use between iterations and the full native vector width for the target, we do not get the benefit of horizontal vector instructions to produce scalar results. This is because the horizontal vector instructions operate on all lanes of the specified vector value. In the case of the undersize tests, some of the lanes contain undefined values that will lead to undefined behaviour in the tests if we were to use them. Functions which have a high number of vertical vector instructions can mitigate the cost of expensive scalarizing epilogue code like this through the performance gained by those vertical instructions. However, as is the case in the substring search, there is a low number of vertical vector instructions generated. As a result, the absence of 7-byte and 11-byte horizontal vector instructions reduce the performance gains of data re-use and native vector sized memory accesses.

The absence of these horizontal vector instructions is also the reason why we see significant code size increases for these undersized tests. With SLP enabled on its own, there is a reduction in code size due to replacing multiple scalar instructions with single vector instructions. When LS is enabled, this is undone by the scalarizing vector element extract instructions at the end of the vectorized instruction chains. There is also an increase in code size caused by the duplication of instructions in the preload and final basic blocks that are generated by the optimization. The additional increase in code size in the aggressive LS optimization is caused by the introduction of the vector shuffle instructions inserted into the final block to shift the input vectors to the left, and into position for the next iteration.

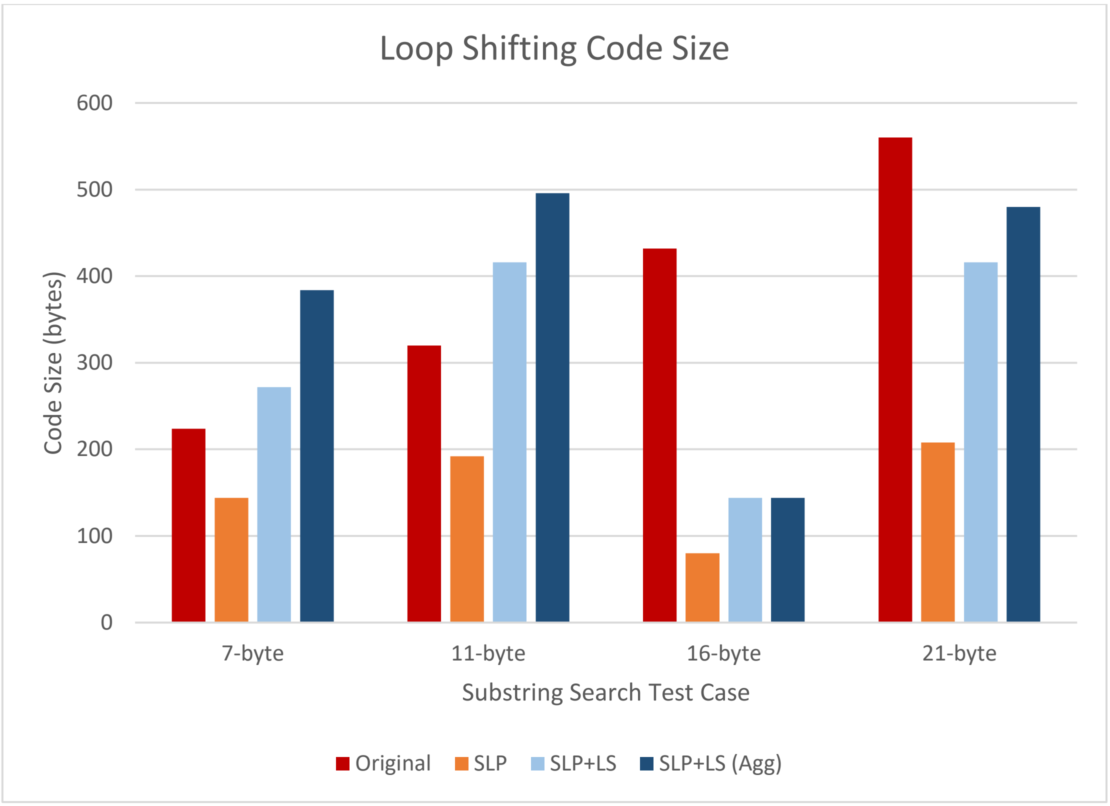


Figure 63 SLP vectorizer with LS code size results

This is not the case for the 16-byte test however. Since we are operating on 16-bytes of data at a time and the native vector size is 16-bytes, we get the benefits of data re-use between iterations, as well as the horizontal vector instructions. With just SLP enabled, the operations per cycle achieved by the test increase from 1.793695 to 4.454074, a 2.4832x improvement. With loop shifting enabled, this jumps to 6.640599 operations per cycle, signifying a 3.7022x improvement in performance. Since the function already operates on the native vector size of our target architecture, there are no additional benefits to performing the aggressive loop shifting since it produces the same code as the basic approach.

Another benefit of the horizontal vector instructions is the reduction in code size we see compared to the undersized tests. With LS enabled there is still an increase in code size over the SLP generated code due to the duplication of instructions, however it is significantly less of an increase than in the undersized tests.

For the 21-byte test, we see some more modest performance improvements from loop shifting. With SLP enabled, the performance improves from 1.970814 operations per cycle to 4.007509 operations per cycle, a 2.0334x improvement. Moving to loop shifting, the performance once again improves to 5.123593 operations per cycle for the basic approach, and 5.348280 operations per cycle for the aggressive approach, a 2.5997x and a 2.7137x improvement respectively.

The 21-byte test suffers from the same problem as the 7-byte and 11-byte tests. The 21-byte operations performed by this test are seen as `16+5` by the vectorizer. This means we get the full benefit of horizontal vector instructions for the first 16-bytes of the data, but the remaining 5-bytes do not, even though they are stored in a 16-byte vector as well (along with the data for the next eleven iterations of the loop).

Once again, the scalarizing epilogue code in place of the horizontal vector instructions is the reason why we see a greater increase in code size for this test, when compared to the 16-byte test. Similar to the undersized tests, the additional increase in code size with the aggressive LS optimization enabled is caused by the additional vector shuffle instructions that are inserted at the end of the final block to shift the input vector values to the left for the next iteration of the final loop.

6.3.4. SLP with Chain Reordering and Data Re-use

We use a set of 5x5, 7x7, and 9x9 image convolutions as our benchmarks for all our vectorization optimizations together. In this section, we will discuss the performance changes generated by each of the optimizations individually, as well as working together. We start by examining the performance results for the 5x5, 7x7, and 9x9 integer convolutions, followed by a brief examination of the results for the floating-point convolutions. We finish this section with an analysis of all results.

6.3.4.1. Integer Convolutions

The integer convolution performance results are shown in Figure 64. The code size results for these tests are shown in Figure 65.

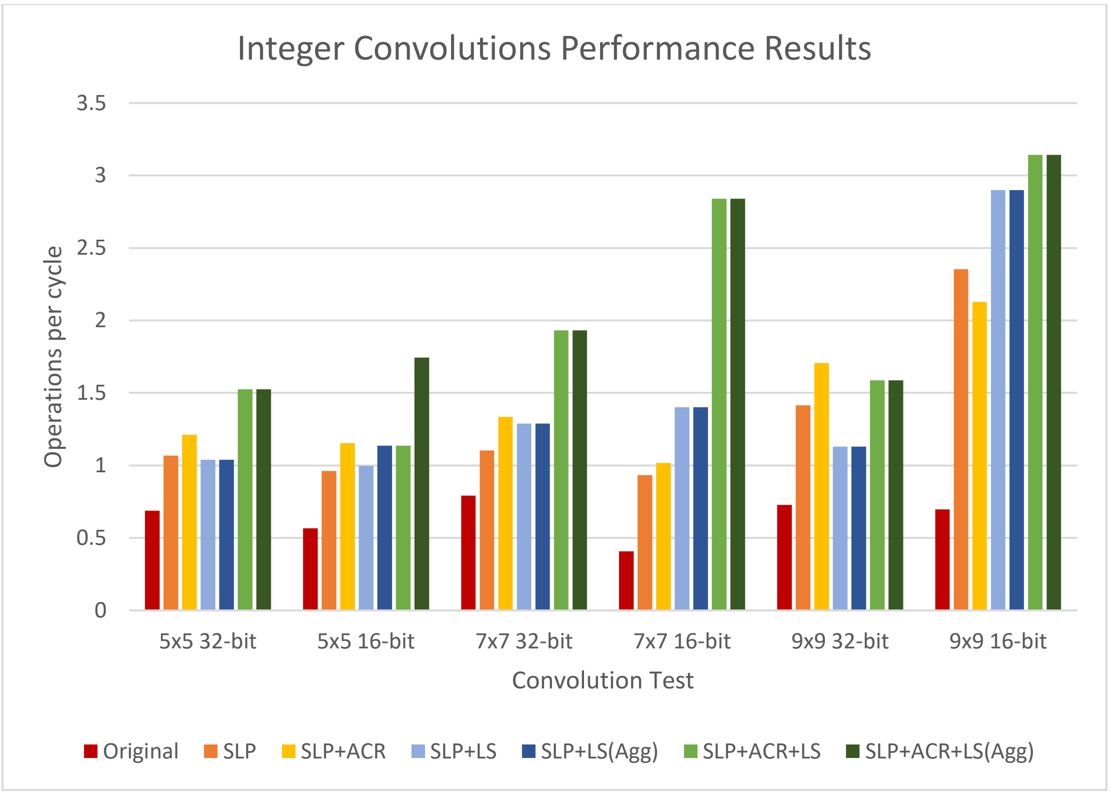


Figure 64 Operations per cycle results for integer image convolutions

With SLP enabled, the convolution tests see improvements of anywhere between 1.3941x and 3.774x when compared to the original scalar version of the function. The improvement fluctuates based on the convolution size and data type involved. The largest improvement is seen on the 9x9 16-bit test since the vectorizer can produce a `v8i16` vector and a single scalar for each operation. The compiler for our target architecture also supports 64-bit vector operations [54], which is why the 5x5 and 7x7 16-bit convolution achieve a significant performance improvement as well.

Every test receives an improvement in code size with SLP enabled. The reduction seen in each test depends on the number of vectorizable scalar operations in the original "unoptimized" version of the test and the vectorization factor that is used by the vectorizer. As the kernel size increases, so too does the improvement in code size.

With ACR enabled, the integer convolution tests see improvements between 1.6870x and 3.0561x when compared to the original scalar version of the function. When compared to the code generated by the SLP vectorizer on its own, the performance changes range from 0.9049x to 1.2102x.

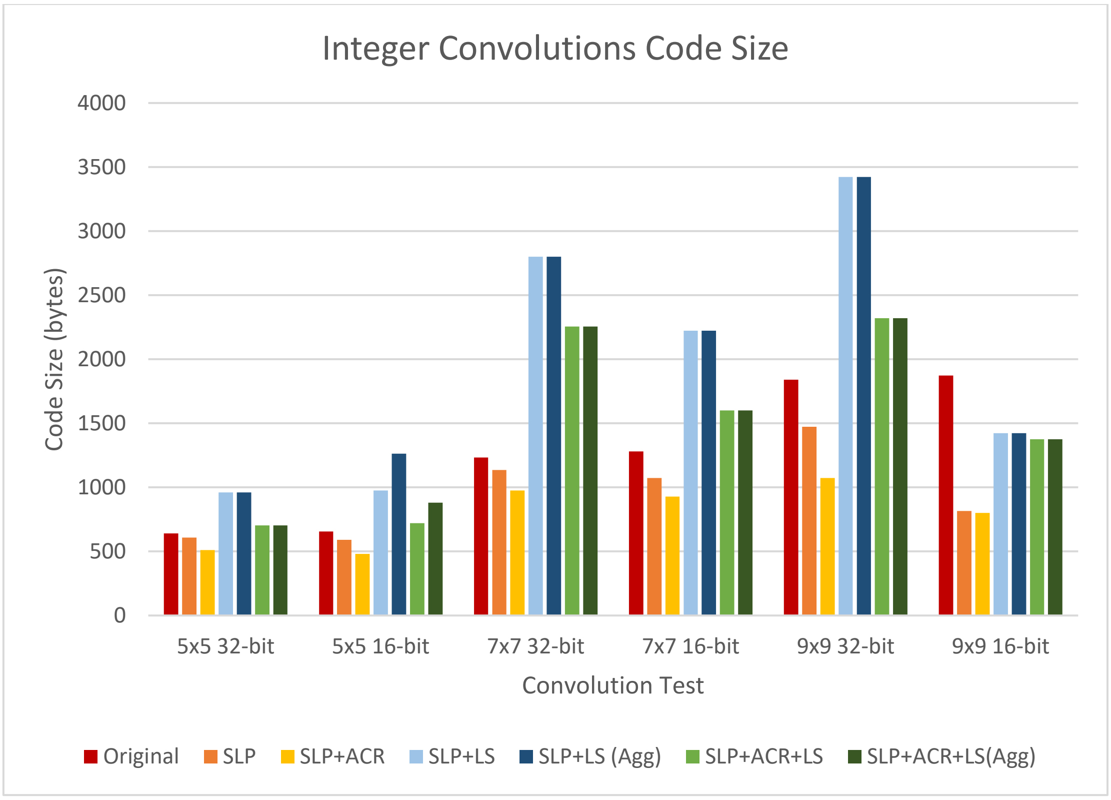


Figure 65 Code size results for integer image convolutions

There is one slowdown with ACR enabled over the SLP vectorizer on its own in the 9x9 16-bit convolution. In this case, the SLP vectorizer produces vector multiply instructions, the results of which are deconstructed into scalar values that are fed into the original scalar accumulation chain. Due to an optimization in the compiler backend, these are reduced to horizontal `add` instructions. This convolution is the only integer convolution in our test suite in which using horizontal `add` instructions in this way is faster than using the vector accumulation chain generated by ACR. It is faster in this case because the horizontal `add` instruction is performed on a different functional unit to the vertical vector multiplies. This means the horizontal `add` for one row of the convolution can be performed alongside the multiply instruction for another row, signifying an increase in instruction level parallelism.

The use of these horizontal `add` instructions in the SLP version of the 9x9 16-bit convolution is also the reason for the more significant reduction in code size compared to the other tests. While the performance in this test decreases with ACR enabled, there is a moderate improvement in code size. This is caused by a small reduction in the total number of instructions in the function.

For all other tests, there is a reduction in code size with ACR enabled that is caused by the removal of the scalarizing vector element extract instructions and the scalar accumulation chain. These instructions are all replaced by the shorter vector accumulation chain.

Moving on to the data re-use optimization (LS), we see improvements in the integer convolution tests in the range 1.5089x to 4.1612x over the original scalar version of the function. When compared to the version generated by the SLP vectorizer on its own, we see performance changes between 0.7984x and 1.5013x. These ranges are the same for both the basic and aggressive approaches to LS.

In all tests, the data re-use optimization causes an increase in code size. This is an expected cost in performing this optimization. The increase in code size is caused by the duplication of instructions that is inherent in shifting one or more iterations from the body of a loop. There are also additional boiler-plate instructions introduced that ensure the safety of the optimization at runtime (e.g. the loop bounds check in the preload basic block).

6.3.4.2. Floating-point Convolutions

The floating-point convolutions are implemented using the same template C++ function as their integer counterparts, only with floating-point types. All of the tests have been compiled with the Clang flag “`-ffast-math`” which tells Clang and LLVM that it may perform aggressive floating-point optimizations that may change the result of floating-point calculations. Our ACR implementation is one such optimization. The performance results for the 5x5, 7x7 and 9x9 floating-point convolutions are shown in Figure 66. The associated code size results are shown in Figure 67.

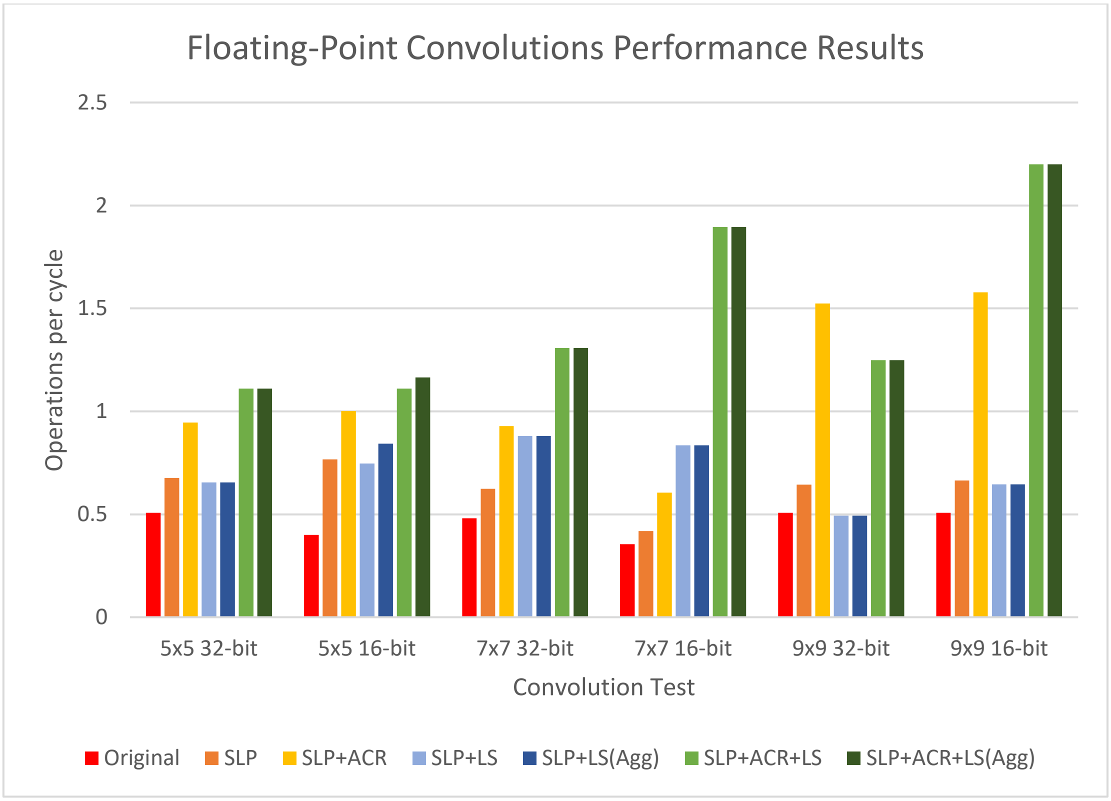


Figure 66 Operations per cycle results for floating-point image convolutions

When comparing the integer performance results in Figure 64 with the performance results for the equivalent floating-point tests in Figure 66, the results are largely the same. The values of the operations per cycle are different but the relationships between the results are very similar. One notable exception to this is the 9x9 16-bit convolution with the data re-use optimization enabled. As mentioned previously, the 9x9 16-bit integer convolution takes advantage of an optimization in the target backend that reduces the scalar accumulation chains to horizontal vector `add` instructions. This optimization is not performed on the equivalent 16-bit floating-point operations.

Because of this, the floating-point version of this convolution does not benefit as much from LS on its own. In fact, with LS enabled, the 9x9 16-bit floating-point convolution uses an excessive number of registers with the scalar accumulation chain.

This causes a more significant increase in code as shown in Figure 67, when compared with the integer version in Figure 65.

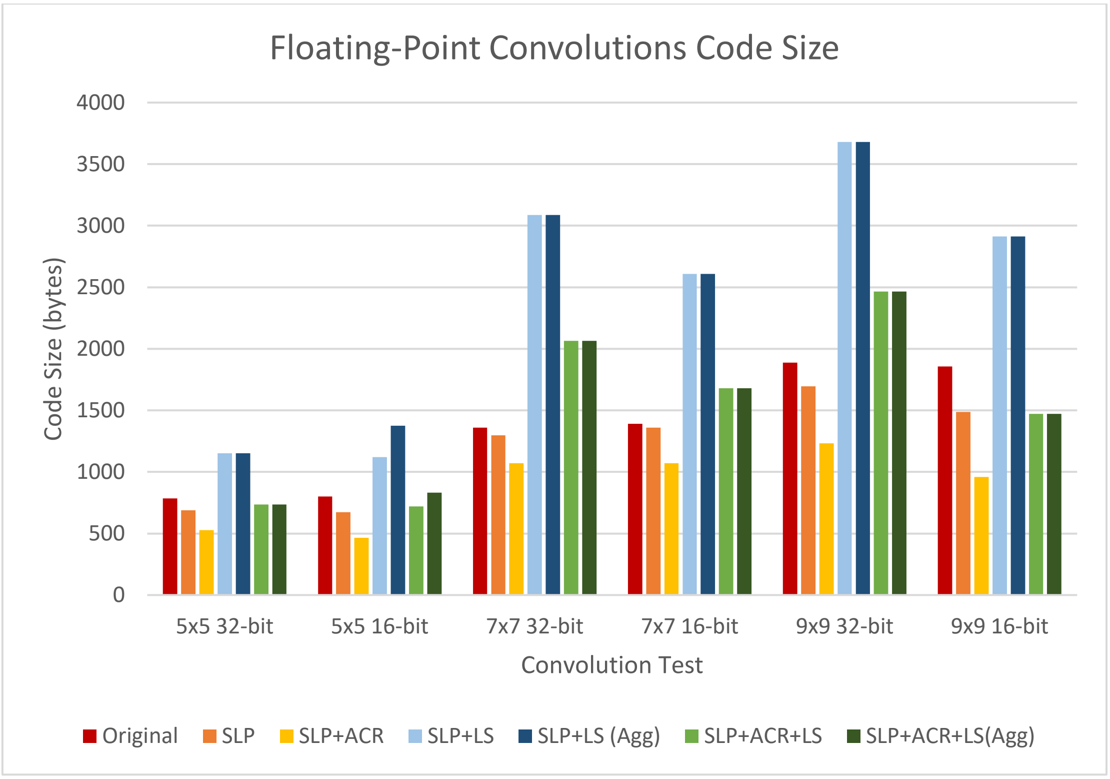


Figure 67 Code size results for floating-point image convolutions

Overall, the floating-point convolutions have a lower operations per cycle value than their integer counterparts. In general, floating-point operations are more computationally intensive than the equivalent integer operations which means they have a higher instruction latency. The operations per cycle results are lower because of this. The number of operations for each test is the same as the equivalent integer test, but the number of cycles required to execute them is higher.

6.3.4.3. Performance Analysis

With the data re-use optimization (LS) enabled, there are two slowdowns when compared to the SLP vectorizer on its own. These are in the 5x5 and 9x9 32-bit convolutions. When performing the LS optimization, the vectorizer hoists memory load instructions out of the body of the loop and replaces them with vector shuffle instructions and scalar loads for the values required for the next iteration of the loop. At a low level, this process involves hoisting instructions executed on the Load-Store

Unit (LSU) and replacing them with instructions executed on the Compare Move Unit (CMU). In the case of the 5x5 32-bit integer convolution, the CMU is already under heavy use by the vector element extract instructions required for extracting the multiply results for the scalar accumulate chain. As a result, we are moving many operations off an idle functional unit (the LSU) and creating new instructions on a busy functional unit. This means there is reduced instruction level parallelism available in the produced code, which increases the overall number of cycles required to execute the vectorized loop.

This is a difficult problem to solve in a code optimization like ours. Our vectorizer operates on LLVM-IR, the target independent intermediate representation used by LLVM. At this stage in compilation, there is no functional unit attached to any instructions in the function. This information is only available once LLVM-IR has been lowered to SHAVE assembly code in the backend. We cannot know what decisions the backend will make when lowering LLVM-IR to assembly code, multiple IR instructions may be squashed into a single assembly instruction, or a single IR instruction may be expanded to multiple assembly instructions.

The 9x9 32-bit integer convolution suffers from a different problem. When vectorizing this convolution, the vectorizer produces two `v4i32` vector operations and one scalar operation for each row of nine operations. As mentioned in Chapter 3, when vectorizing these convolutions, the LS optimization hoists all loads from the kernel into a preload block which is executed before entering the loop body. The image values for the first iteration of the loop are also hoisted into this block. In the case of the 9x9 32-bit integer convolution, this means hoisting 36 `v4i32` vector and 36 `i32` scalar loads out of the loop body and into the preload block. Our target architecture has a scalar register file which contains 32 32-bit registers and a vector register file which contains 32 128-bit registers. Since we have hoisted 36 values out of the loop body, there are 36 registers in both register files live across the back-edge of the loop. The LS optimization has therefore created a situation where register spilling to the stack is inevitable in the register allocator implementation. Naturally, the introduction of these register spills also causes a significant increase in code size due to the additional instructions required.

This problem may be solved by building an estimation of the number of registers that are live at any given point in the loop. This estimate could be used to limit the number of values that are hoisted out of the loop. However, this is a difficult process that would be prone to inaccuracies since (similarly to above) we do not know how LLVM-IR instructions will be lowered in the backend or how values will be mapped to physical registers.

Most of the slowdowns in performance we have listed here that are caused by either ACR or LS are alleviated when both optimizations are enabled together. With both enabled, the vectorizer achieves performance improvements between 2.0050x and 6.9656x over the original scalar versions. Compared to the SLP vectorizer on its own, we see performance improvements in the range 1.1215x to 3.0392x.

The only test that doesn't recover its performance slowdown is the 9x9 32-bit integer convolution. Unfortunately, the high register pressure exhibited in this test cannot be relieved by any current component of our vectorizer.

For the 9x9 16-bit integer convolution, the benefits of the horizontal `add` without ACR are lessened by the data re-use optimization. The number of cycles per loop iteration has been reduced to a point where using the regular vertical `add` instructions outweighs the benefits of the horizontal `add` instructions since there are less instruction cycles to alleviate their higher latency.

With LS enabled, the 5x5 32-bit convolution suffered a performance slowdown caused by creating new instructions on the already busy CMU functional unit. With ACR enabled as well, the pressure on the CMU is greatly reduced since we no longer need to extract every single scalar result from the vector multiply values. As a result, this convolution can utilize the full benefits of both ACR and LS and the performance increases to match this.

In all test cases, enabling ACR alongside LS caused a reduction in code size compared to just LS enabled. This is due to a reduction in the number of instructions that needed to be duplicated from the loop body into the final basic block.

This concludes our evaluation of the performance results generated from our benchmarks. In the next chapter, we provide details on some possible future work and conclude this thesis with some final thoughts.

Chapter 7 Conclusion

7.1. Future Work

There are many ways in which the vectorization strategy we described in this thesis can be expanded. In this section we will describe some possible directions for future work that could supplement and improve our approach to vectorization.

Add support for more complex memory access patterns: Currently, our approach to SLP vectorization uses a relatively simple technique for finding groups of instructions that access contiguous regions of memory. It is capable of grouping instructions which use a common base pointer, a common variable base index (or no base index) and contiguous constant offsets. This technique could be expanded to include more complex, or indirect memory access patterns like accessing C struct members, multi-dimensional arrays, or multiple levels of common variable base indexes.

Our approach is also only capable of vectorizing accesses to contiguous memory locations. Memory accessing strides greater than one are common in many real-world applications that are vectorizable [33]. Our SLP vectorizer, as well as the loop shifting optimization could be expanded to take advantage of these opportunities.

Tentative loop unrolling should be aware of associative chain reordering: The tentative loop unroller we have designed seeks to provide vectorization opportunities up to the natural vectorization factor for the target architecture. However, for loops that accumulate values across iterations, it may be beneficial to perform more aggressive unrolling to allow the associative chain reordering optimization to vectorize the accumulation. Our unroller could be expanded to consider the potential chain reordering in a loop when calculating the unrolling factor for that loop.

Operand shuffling when building vectorizable chains: The vectorizable chain builder in our approach to SLP vectorization requires a one-to-one match for all lanes in a vector in order to link two vectorizable nodes together. However, it is possible that two or more nodes could be linked (via a def-use chain) using a sequence of vector shuffle operations. Using vector shuffle operations to link existing nodes and build new nodes in an optimal fashion could be difficult because any number of independent values could be shuffled together in many different ways. An algorithm to find the “best” solution could be very complex.

Cost model: The optimizations described in this thesis are performed for all eligible loops, regardless of the effects they will have on performance. In some cases, like we saw in section 6.3.4.3, the cost of constructing and deconstructing vector values can outweigh the benefits of vectorization. For cases like these, a cost model could be built to selectively enable each individual optimization only when it is deemed beneficial to do so. Such a cost model would be heavily dependent on the target architecture, as the costs involved in constructing and deconstructing vector values can vary significantly from target to target.

7.2. Final Thoughts

Convolutions are a common and important operation in image processing. Despite their importance, modern optimizing compilers are often incapable of effectively vectorizing them. Existing approaches to vectorization fall victim to problems such as excessive register pressure and poor data locality when targeting convolutions.

In this thesis, we described an approach to vectorization that can effectively optimize convolutions and convolution-like functions. Our approach combines Super Level Parallelism (SLP) with some additional optimizations:

- Tentative loop unrolling is used to create vectorization opportunities for SLP in loops which previously had none
- Loop shifting provides a mechanism for re-using overlapping data across loop iterations
- Chain reordering is used to remove data dependencies between instructions by exploiting the associative and commutative properties of certain operations.

We evaluated our approach by implementing it in a version of LLVM targeting the Movidius SHAVE processor architecture. Through our evaluation, we found that the combination of all the optimizations outlined above was a good technique for automatic vectorization of image convolutions. When enabled individually, each optimization achieved speedups in some tests but slowdowns in others. However, when combined they provided a reduction in execution time in all convolution tests, with speedups ranging from 2.01x to 6.97x for convolutions operating on integer data types, and speedups ranging from 2.19x to 5.34x for the equivalent floating-point convolutions.

These improvements in performance are not limited to image convolutions. In Chapter 6, we demonstrated the efficacy of each of our techniques individually when used to optimize other code examples. These examples included common applications like image white balancing, Hamming distance and sub-string searching.

Through our evaluation and analysis of these examples, and the set of convolution tests, we have demonstrated that our approach to automatic vectorization is profitable for certain families of applications. We have proven our approach can produce significant gains in performance, but with room to improve through potential future work.

Bibliography

[1] M. J. Flynn, "Very High-speed Computing Systems," *Proceedings of the IEEE,* vol. 54, no. 12, pp. 1901-1909, 1966.

[2] J. R. Allen and K. Kennedy, "A Program to Convert Fortran to Parallel Form," Rice MASC TR82-6, Rice University, 1982.

[3] Richard M. Stallman and the GCC Developer Community, "Using the GNU Compiler Collection (GCC): Vector Extensions," 2018. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html. [Accessed 31 01 2018].

[4] M. E. Wolf and M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems,* vol. 2, no. 4, pp. 452-471, 1991.

[5] S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," in *PLDI '00 Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, Vancouver, British Columbia, Canada, 2000.

[6] R. Karrenberg and S. Hack, "Whole-function Vectorization," in *CGO '11 Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Chamonix, France, 2011.

[7] E. W. Weisstein, "Convolution," MathWorld--A Wolfram Web Resource, [Online]. Available: http://mathworld.wolfram.com/Convolution.html. [Accessed 24 02 2018].

[8] B. R. Rau and J. A. Fisher, “Instruction-level Parallel Processing: History, Overview, and Perspective,” *The Journal of Supercomputing,* vol. 7, no. 1-2, pp. 9-50, 1993.

[9] J. A. Fisher, “Very Long Instruction Word architectures and the ELI-512,” in *ISCA '83 Proceedings of the 10th annual international symposium on Computer architecture*, New York, NY, USA, 1983.

[10] Philips Semiconductors, “An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture,” Philips Semiconductors - Pub#9397-750-01759, 1997.

[11] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM Journal of Research and Development,* vol. 11, no. 1, pp. 25-33, 1967.

[12] J. A. Fisher and J. J. O'Donnell, “VLIW Machines: Multiprocessors We Can Actually Program,” Yale University - YALEU/DCS/RR-298, New Haven, CT, USA, 1984.

[13] J. R. Ellis, Bulldog: A Compiler for VLIW Architectures, Yale Univ., New Haven, CT, 1985.

[14] M. Lam, “Software Pipelining: An Effective Scheduling Technique for VLIW Machines,” in *PLDI '88 Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*, New York, NY, USA, 1988.

[15] V. H. Allan, R. B. Jones, R. M. Lee and S. J. Allan, “Software Pipelining,” *ACM Computing Surveys (CSUR),* vol. 27, no. 3, pp. 367-432, 1995.

[16] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan and V. Toma, “Always-on Vision Processing Unit for Mobile Applications,” *IEEE Micro,* vol. 35, no. 2, pp. 56-66, 2015.

[17] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick and D. Donohoe, “Myriad 2: Eye of the Computational Vision Storm,” in *Hot Chips 26 Symposium (HCS), 2014 IEEE*, Cupertino, CA, USA, 2014.

[18] R. Allen and K. Kennedy, “Automatic Translation of FORTRAN Programs to Vector Form,” *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 9, no. 4, pp. 491-542, 1987.

[19] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith and J. L. Wagener, Fortran 90 Handbook, New York, NY, USA: Intertext Publications McGraw-Hill Book Company, 1992.

[20] The Clang Team, “Clang Language Extensions,” 2018. [Online]. Available: https://clang.llvm.org/docs/LanguageExtensions.html. [Accessed 31 01 2018].

[21] G. Ren, P. Wu and D. Padua, “An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions,” in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, Denver, CO, USA, 2005.

[22] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi and B. Juurlink, “An Evaluation of Current SIMD Programming Models for C++,” in *WPMVP '16 Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, Barcelona, Spain, 2016.

[23] R. Allen and S. Johnson, “Compiling C for Vectorization, Parallelization, and Inline Expansion,” in *PLDI '88 Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*, Atlanta, GA, USA, 1988.

[24] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong and D. A. Padua, “An Evaluation of Vectorizing Compilers,” in *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Galveston, TX, USA, 2011.

[25] D. Nuzman and R. Henderson, “Multi-platform Auto-vectorization,” in *CGO '06 Proceedings of the International Symposium on Code Generation and Optimization*, New York, NY, USA, 2006.

[26] D. F. Bacon, S. L. Graham and O. J. Sharp, “Compiler Transformations for High-Performance Computing,” *ACM Computing Surveys (CSUR),* vol. 26, no. 4, pp. 345-420, 1994.

[27] A. Anderson, A. Malik and D. Gregg, “Automatic Vectorization of Interleaved Data Revisited,” *ACM Transactions on Architecture and Code Optimization (TACO),* vol. 12, no. 4, p. 50, 2016.

[28] D. Nuzman and A. Zaks, “Outer-loop Vectorization - Revisited for Short SIMD Architectures,” in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Toronto, ON, Canada, 2008.

[29] I. Rosen, D. Nuzman and A. Zaks, “Loop-Aware SLP in GCC,” in *2007*, Ottawa, ON, Canada, Proceedings of the GCC Developers’ Summit.

[30] J. Shin, M. Hall and J. Chame, “Superword-Level Parallelism in the Presence of Control Flow,” in *CGO '05 Proceedings of the international symposium on Code generation and optimization* , San Jose, CA, USA, 2005.

[31] V. Porpodas, A. Magni and T. M. Jones, “PSLP: Padded SLP Automatic Vectorization,” in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, USA, 2015.

[32] A. E. Eichenberger, P. Wu and K. O'Brien, “Vectorization for SIMD Architectures with Alignment Constraints,” in *PLDI '04 Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, Washington DC, USA, 2004.

[33] D. Nuzman, I. Rosen and A. Zaks, “Auto-Vectorization of Interleaved Data for SIMD,” in *PLDI '06 Proceedings of the 27th ACM SIGPLAN Conference on*

Programming Language Design and Implementation, Ottawa, Ontario, Canada, 2006.

[34] J. R. Allen, K. Kennedy, C. Porterfield and J. Warren, “Conversion of Control Dependence to Data Dependence,” in *POPL '83 Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, Austin, TX, USA, 1983.

[35] M. Lam and R. Wilson, “Limits of Control Flow on Parallelism,” in *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, Gold Coast, Australia, 1992.

[36] J. C. H. Park and M. Schlansker, “On Predicated Execution,” Hewlett-Packard Laboratories HPL-91-58, Palo Alto, CA, USA, 1991.

[37] G. Ren, P. Wu and D. Padua, “A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions,” *Languages and Compilers for Parallel Computing,* pp. 420-435, 2004.

[38] A. S. Huang, G. Slavenburg and J. P. Shen, “Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation,” in *ISCA '94 Proceedings of the 21st annual international symposium on Computer architecture* , Chicago, IL, USA, 1994.

[39] D. Bernstein, D. Cohen and D. E. Maydan, “Dynamic Memory Disambiguation for Array References,” in *MICRO 27 Proceedings of the 27th annual international symposium on Microarchitecture*, San Jose, CA, USA, 1994.

[40] The LLVM Team, “The LLVM Compiler Infrastructure Project,” 2018. [Online]. Available: http://llvm.org/. [Accessed 01 02 2018].

[41] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *CGO '04 Proceedings of the international*

symposium on Code generation and optimization: feedback-directed and runtime optimization, Palo Alto, CA, USA, 2004.

[42] The LLVM Team, “LLVM Language Reference Manual,” 2018. [Online]. Available: http://llvm.org/docs/LangRef.html. [Accessed 01 02 2018].

[43] S. S. Muchnick, “Static Single-Assignment (SSA) Form,” in *Advanced Compiler Design and Implementation*, Academic Press, 1997, pp. 252-258.

[44] The LLVM Team, “The Often Misunderstood GEP Instruction,” 08 02 2018. [Online]. Available: https://llvm.org/docs/GetElementPtr.html. [Accessed 09 02 2018].

[45] The Clang Team, “Clang: a C language family frontend for LLVM,” 2018. [Online]. Available: https://clang.llvm.org/. [Accessed 07 02 2018].

[46] International Organization for Standardization, “ISO C Standard 1999, ISO/IEC 9899:1999,” International Organization for Standardization, Geneva, Switzerland, 1999.

[47] M. Acton, “Demystifying The Restrict Keyword,” 29 05 2006. [Online]. Available: http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html. [Accessed 07 02 2018].

[48] International Organization for Standardization, “ISO C++ Standard 2011, ISO/IEC 14882:2011,” International Organization for Standardization, Geneva, Switzerland, 2011.

[49] U. Karn, “An Intuitive Explanation of Convolutional Neural Networks,” 11 08 2016. [Online]. Available: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/. [Accessed 02 02 2018].

[50] S. Kim and R. Casper, “Applications of Convolution in Image Processing with MATLAB,” University of Washington, 2013.

[51] S. S. Muchnick, “Loop-Invariant Code Motion,” in *Advanced Compiler Design and Implementation*, Academic Press, 1997, pp. 397-407.

[52] M. H. Ionica and D. Gregg, “The Movidius Myriad Architecture's Potential for Scientific Computing,” *IEEE Micro,* vol. 35, no. 1, pp. 6-14, 2015.

[53] S. Gupta, N. Dutt, R. Gupta and A. Nicolau, “Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow,” in *DATE '04 Proceedings of the conference on Design, automation and test in Europe - Volume 1* , Washington, DC, USA, 2004.

[54] E. Diken, M. J. O'Riordan, R. Jordans, L. Jozwiak, H. Corporaal and D. Moloney, “Mixed-length SIMD code generation for VLIW architectures with multiple native vector-widths,” in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, Toronto, ON, Canada, 2015.

[55] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung and D. I. August, “Decoupled Software Pipelining Creates Parallelization Opportunities,” in *CGO '10 Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* , Toronto, ON, Canada, 2010.