Optimization of Vector Instructions in Divergent Control Flow on Long-Vector Architectures using Active-Lane Consolidation

by

Wyatt Praharenka

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

IF-conversion is a classic Control-Flow Graph (CFG) transformation that linearizes the basic blocks in a loop and assigns a predicate to them to enable or disable execution. IF-conversion is performed before vectorization where scalar statements are translated into vector statements to enable the vectorization of code with control flow constructs such as *if-then-else* statements. Divergent control flow contributes to the under-utilization of vector instructions where a low ratio of vector lanes produce a useful result. Traditional methods, such as the insertion of Branch on Super-Word Condition-Codes (BOSCCs), attempt to elide the execution of vector code with no active-lanes, however, with the increasing vector-lengths (VLs) appearing in vector Instruction-Set Architecture (ISA) extensions, BOSCCs lose their effectiveness.

This thesis presents Active-Lane Consolidation (ALC), a variable vector permutation that re-organizes the lanes between two vector registers to consolidate those that are active into a single vector, increasing utilization; details the implementation of ALC in a modern vector ISA using only the current permutations available; presents two loop-transformations for loops with divergent control flow to utilize ALC and presents case studies for four benchmarks from the SPEC 2017 benchmark suite. The case studies show that ALC can substantially increase vector utilization in certain loops with divergent control flow, especially where IF-conversion and BOSCCs struggle. The increase in vector utilization translates to a large reduction in the dynamic instruction count giving evidence to the profitability of using ALC.

Preface

Chapters 2, 3 and 4 have been submitted for publication as **W. Praharenka**, D. Pankratz, J. P. L. Carvalho, E. Amiri, J. N. Amaral, "Vectorizing Divergent Control Flow with Active-Lane Consolidation on Long-Vector Architectures." I was responsible for the conception, implementation and evaluation of the project. D. Pankratz contributed many fruitful discussions regarding the engineering aspects of the project in addition to providing insightful feedback on the organization of the manuscript. J. P. L. Carvalho provided recommendations on the workings of the loop transformations and helped guide the direction of the project. E. Amiri guided the project during the early formulation of the ideas and assisted in providing the required infrastructure for the experiments and evaluation. J. N. Amaral oversaw the research, from start to finish, ensuring a high standard of quality while contributing outstanding advice on all aspects involved. All authors collaborated in the writing and editing of the manuscript.

To my Mom

Your hard work in raising me and all your good teachings have resonated throughout my life. I love you.

To my Dad

Your lessons in work ethic and perseverance have inspired me when needed the most. I love you.

Acknowledgements

I give thanks to my supervisor, Dr. J. Nelson Amaral, for all the teachings, guidance, and opportunities he has given me. I asked him my first question, about the operation of arithmetic shifts, in an intro architecture course only a few short years ago. Since then, he has mentored me in my learning and has played an immense role in shaping my academic interests.

Thank you to those I have come across in the lab, especially João Carvalho, for all the discussions he took part in and feedback he has given, and Giancarlo Pernudi Segura, for his help in experiments.

I thank my manager at Huawei, Ehsan Amiri, for the numerous discussions and feedback he has given me since I began working with him. In addition, I thank those from the compiler team at Huawei that I have had an opportunity to work with. I have learned something from each one of you.

I thank all those that I have come across during my time at the University of Alberta, including Dr. Ioanis Nikolaidis, who introduced me to research and broadened my knowledge of computing science. It was a pleasure to have had the opportunity to work with you.

I would also like to thank David Pankratz, for his friendship and for being an incredible colleague. By now, our chat log used since first working together in our undergraduate probably exceeds the word count of this thesis and contains a great many conversations that reveal the depth of his creativity.

This research has been funded in part by the University of Alberta Huawei Joint Innovation Collaboration (UAHJIC) and Graduate Research Assistantship Fellowship (GRAF).

I thank all others who have supported me in one way or another throughout this endeavor:

Thank you, Joel Praharenka, for being there for me. You are outstanding as both a brother and a friend. I wish you the best.

I thank my friend, Steven Weikai Lu, for all the laughs and discussions we have shared throughout our university careers. And for keeping me at the computer lab until midnight.

I thank my close friend Brett Kueber, whom I had the pleasure of being roommates with for the better part of my academic career, for your steadfast fellowship. Your courtesy and humor always brought joy to my day, especially over the last two years. I also thank his girlfriend, Delane Litke, who shares the same kind-heartedness as Brett. It is always a delight to get together with the two of you.

Finally, I give thanks to Ezra Cyr, whom I have been close friends with since we were children. For all our experiences together and our regular conversations to this day. It is always exciting to discover the interesting things you are up to.

Contents

1 Introduction **1**

2 Background **5**
2.1 SIMD Execution and Vector Extensions 6
2.1.1 Scalable Vectors 7
2.2 Loop Vectorization 8
2.2.1 Gather and Scatter Memory Accesses 10
2.3 Control Divergence 11
2.3.1 IF-Conversion 11
2.3.2 Uniform and Divergent Vectors 12
2.3.3 Branch-on-Superword Condition Codes 12
2.4 Summary 14

3 Active-Lane Consolidation **15**
3.1 Optimizing Divergent Vectors with ALC 18
3.1.1 Active-lane Consolidation Permutation 18
3.1.2 Loop-Unrolling ALC 19
3.1.3 Tracking Lane Indices Through Permutation 22
3.1.4 Permuting Instruction Operands 23
3.2 Active-Lane Consolidation with SVE 24
3.2.1 SVE Permutations 24
3.2.2 Active-Lane Consolidation in SVE 28
3.2.3 Inter-Register Indexed Permutation in SVE 29
3.3 Proposal for native support in SVE 30
3.4 Iterative ALC 32
3.5 Summary 34

4 Evaluation **35**
4.1 Methodology 36
4.1.1 Case Studies 38
4.1.2 Loop Transformations 39
4.2 Results 45
4.2.1 Iterative ALC 46
4.2.2 Under-Utilization of Vector Instructions 47
4.2.3 Overhead of the ALC Permutation 48
4.3 Summary 50

5 Related Work **52**
5.1 Control-Flow Vectorization 52
5.2 Dynamic Uniformity Conditions 55
5.3 Vector Lane Re-organization 56
5.4 GPU Thread Re-organization 58
5.5 The ARM Scalable Vector Extension 58

6 Future Work **59**

7 Conclusion **62**

References **64**

List of Tables

4.1 Loops where ALC may be applicable discovered by the static analysis described in Section 4.1. 36
4.2 Static instruction counts for the control flow paths in each kernel. 48

List of Figures

1.1 Distribution of vector hardware utilization in the LBM program of the SPEC CPU 2017 benchmark suite when executing the function `StreamCollideTRT`. Darker shades indicate a higher vector utilization. This function accounts for 97% of execution time. **ifcvt:** naive predicated execution of vectorized code using flattened control-flow. **ifcvt+boscc:** the IF-converted code optimized using BOSCCs. BOSCCs detect uniform vectors to elide unnecessary execution of basic blocks. At a vector length of 2048-bits, BOSCC optimizations have nearly no effect. . . . 2

2.1 Visual breakdown of the components in vector and predicate registers. . . . 6
2.2 A simple data-parallel loop with control flow 8
2.3 SVE assembly of the vectorized loop in Figure 2.2 9
2.4 BOSCCs used to optimize the IF-converted code 13
2.5 Example vector execution of Figure 2.2 with an *any* BOSCC inserted to skip *B1*. Light coloured squares represent active lanes while dark inactive lanes. . . . 13

3.1 Simplified loop from NAB 16
3.2 The ALC permutation performed on two divergent vectors. Light coloured squares represent active lanes while dark coloured lanes inactive. Inactive lanes are moved into the remainder vector `vR`. . . . 18
3.3 Control flow graphs after: (a) Scalar, (b) If-conversion and vectorization of (a), (c) Unrolling of (b), (d) ALC on (c). . . . 20
3.4 Inter-vector permutation to create the merged operand by permuting the concatenated vectors of the unrolled iterations based on the consolidated index vector `idxM` 22
3.5 Example loop where the op variable must be permuted and contiguous loads converted to gather loads to correct the order to the consolidated iterations if ALC is applied. . . . 24
3.6 Example loop from Figure 3.5 with vectorization,unrolling and ALC applied. Operands are merged accordingly and contiguous loads inside the consolidate block are converted to gather loads. 25
3.7 Semantics of SVE permutation instructions and index/whilelt instructions used to implement the active-lane consolidation and multi-register permutation. . . . 26

3.8 Visual example of the ALC permutation implemented in SVE. Active lanes are shown as light coloured squares while inactive as dark coloured squares. For brevity, we use `cntp` as a function but requires another instruction that returns the count of active lanes. In addition, we use `~p0` to represent the logical `NOT` of predicate register `p0`. In 1, once compacted, there are additional zero values appended at the end of the vector which are empty lanes rather than a valid zero index. These are not problematic when used as the predicate will prevent any instruction from using the zeros at the end 27
3.9 Implementation of the ALC permutation in SVE described in Section 3.1.1. Example values show in this listing are consistent with the prior examples. 28
3.10 SVE-specific pseudo-code for the inter-register permute as introduced in Section 3.1.4. 30
3.11 Proposed instruction to support the active-lane consolidation permutation in SVE 31
3.12 Proposed extension of `tbl` to perform an multi-vector table lookup 31
3.13 Example of iterative ALC. Active lanes are shown as white squares and inactive as dark squares. 32

4.1 Results of a static analysis on the loops matching the basic criteria for ALC to be applicable in the SPEC CPU 2017 benchmark suite. 35
4.2 Control flow graphs of the body of the loops of the benchmarks. Percentages attached to edges indicate the runtime probability the branch is taken in the TRAIN workload. 38
4.3 Reduction in dynamic instruction count of loops optimized with BOSSCs (boscc) or the ALC transformations (alc_unroll, alc_iter) over the kernel with only vectorization and `if`-conversion (ifcvt) applied. Figure 4.3d shows the control-flow graphs of the kernels with the locations of the BOSSC branches that are indicated by a dashed line. 42
4.4 Percent divergence of vector instructions. Dotted black bars indicate 100% utilization while dotted white bars indicate 0% utilization. Shaded bars indicate partial utilization with darker shades representing higher utilization. 44

List of Acronyms

ALC Active-Lane Consolidation.

AVX Advanced Vector Extension.

BOSCC Branch on Super-Word Condition-Code.

BOSCCs Branch on Super-Word Condition-Codes.

CFG Control-Flow Graph.

DLP Data-Level Parallelism.

GPU Graphics Processing Unit.

IR Intermediate Representation.

ISA Instruction-Set Architecture.

PC Program Counter.

RISC Reduced Instruction Set Computer.

SIMD Single-Instruction Multiple-Data.

SSE Streaming SIMD Extension.

SVE Scalable Vector Extension.

VL Vector Length.

VLA Vector-Length Agnostic.

Chapter 1

Introduction

Loop vectorization is an important and popular optimization to extract performance from data-parallel loops. During loop vectorization, multiple independent loop iterations are mapped to the lanes of a single vector instruction that executes quicker than performing each scalar operation individually. Interest surrounding vector architectures is growing with the introduction of vector extensions such as ARM's Scalable Vector Extension (SVE) and RISC-V "V" that bring new capabilities to accelerate a diverse collection of workloads using vector hardware. For instance, ARM's SVE has been successfully deployed in the Fugaku supercomputer by Fujitsu, that has taken the lead in the TOP500 ranking of most powerful supercomputers [24].

The challenges to vectorization presented by control flow in loops — introduced by `if-then-else` and `goto` statements — are commonly addressed by linearizing the control flow through IF-conversion [1]. IF-conversion is supported through predicated vector instructions, a common feature in modern vector ISAs such as Intel's Advanced Vector Extension (AVX), ARM's SVE and RISC-V's "V" extension. However, predicated vector code generated after IF-conversion is often inefficient because many of the lanes in the vector may be inactive during execution and will not produce a useful result. In VL-time architectures, the presence of inactive lanes does not lower the latency of the vector instruction [7] and thus executing a vector instruction with a high ratio of active to inactive lanes is desirable. In contrast, density-time architectures [7] are designed such that the instruction latency depends on the

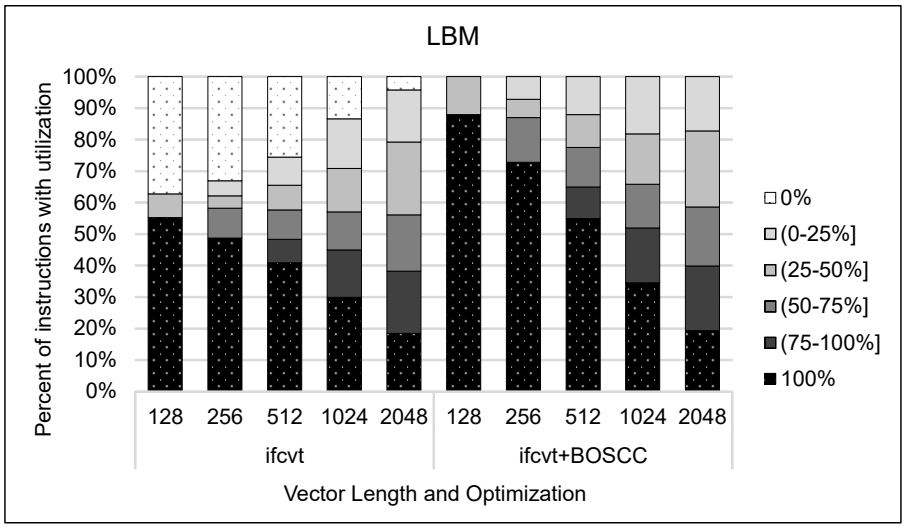


Figure 1.1: Distribution of vector hardware utilization in the LBM program of the SPEC CPU 2017 benchmark suite when executing the function `StreamCollideTRT`. Darker shades indicate a higher vector utilization. This function accounts for 97% of execution time. **ifcvt:** naive predicated execution of vectorized code using flattened control-flow. **ifcvt+boscc:** the IF-converted code optimized using BOSCCs. BOSCCs detect uniform vectors to elide unnecessary execution of basic blocks. At a vector length of 2048-bits, BOSCC optimizations have nearly no effect.

amount of active lanes so that executing an instruction with a single active lane takes a shorter amount of time than executing the same instruction with all lanes active. However, modern vector architectures are VL-time architectures.

Branch on Super-Word Condition-Code (BOSCC) instructions [30], [31] can be used to bypass execution of vector instructions containing only inactive lanes. BOSCC instructions are available in SVE and can be emulated in RISC-V "V". Figure 1.1 shows the vector utilization in a loop taken from the LBM benchmark, a SPEC CPU2017 benchmark, of IF-converted code before (ifcvt) and after the use of BOSCCs (ifcvt+boscc).[1] IF-conversion [1] is a compiler pass that enables vectorization of code containing control-flow by linearizing the basic blocks so that every block in the original CFG is executed and divergent control flow is handled by predicating the blocks with the corresponding condition. Each bar shows the vector utilization distribution of instructions measured at runtime. Darker bars represent higher utilization

[1]For details of the experiment refer to Chapter 4.1

of vector registers.[2] The results indicate that BOSCC instructions are effective on short vectors and much less effective on longer vectors that are becoming increasingly common. For instance, while executed on 128-bit long vectors, BOSCCs raise the percentage of uniformly active executions from 55% to 88%; for 2048-bit long vectors, *ifcvt* and *ifcvt+BOSCC* are indistinguishable because BOSCCs inserted in the code are unable to optimize the partially active vector instructions that dominate when executing with 2048-bit long vectors. The innovations presented in this thesis attempt to increase the efficiency of execution on long vector architectures in the presence of divergent control flow.

The effectiveness of BOSCC instructions decreases as vector-length grows because it is more likely that some lanes of the vector evaluate differently than the others for a given condition causing the vector to become divergent and the BOSCC instruction to fail. The penalty to the effectiveness of BOSCCs when moving to longer vectors is concerning given that the industry has been consistently increasing vector length. For instance, Intel first moved from 128-bit vectors in SSE to 256-bits with AVX and now to 512-bits in their most current vector extension, AVX-512. ARM has taken a slightly different approach, moving from 128-bit vectors in Advanced SIMD to SVE that now allows support for up to 2048-bit long scalable-vectors. This thesis addresses the issues with vector utilization by improving the handling of control-flow divergence especially in long-vector architectures.

This thesis proposes a new permutation, ALC, to improve vector utilization in the presence of divergent control flow where state-of-the-art optimizations fall short. The goal of the ALC permutation is to consolidate active lanes from two partially active vectors into a single vector register with the aim of forming uniform vectors that BOSCC instructions are able to optimize. We propose two loop-transformations that use the ALC permutation to consolidate vectors corresponding to a conditional block in the control-flow graph in attempt to increase vector utilization. The unrolling ALC transformation consolidates instances of vector instructions in a vectorized loop from two consecutive iterations exposed by loop-unrolling. The second transformation, iterative

[2]In this thesis vector registers are referred to simply as *vectors*.

ALC, broadens the range that lanes can be consolidated from to several iterations by decoupling the consolidated vector from the loop, allowing for active lanes in any iteration during the loop to be moved into the consolidated vector.

The contributions of this thesis are as follows:

1. Active-lane consolidation, a new permutation operation that consolidates active lanes to increase vector utilization. ALC includes a mechanism to also retain the inactive lanes, allowing ALC to be applied to arbitrary control flow.

2. An implementation of the ALC permutation to demonstrate that ALC readily maps to existing vector architectures.

3. Two loop transformations, Loop-Unrolling ALC and Iterative ALC, that marry ALC and BOSCCs to improve vector utilization leading to a reduction in the number of instructions executed.

4. A proposal for a new vector permutation instruction, motivated by ALC's efficacy, that would provide out-of-the-box lane consolidation.

5. Emulation-based case studies that show the applicability of the ALC permutation and the proposed loop-transformations to the SPEC CPU 2017 benchmark suite and the resulting benefits.

This remainder of this thesis is organized as follows: Chapter 2 introduces and explains the required background material. Active-lane consolidation and all accompanying techniques are presented throughout Chapter 3. Lastly, Chapter 4 presents an evaluation of ALC and the loop-transformations through four case studies of benchmarks selected from SPEC CPU2017.

Chapter 2

Background

Traditional scalar processors contain functional units that perform operations on either a single operand for unary operations or two operands for binary operations. Accordingly, each register is only able to hold a single data for a single operand. These scalar registers are used by scalar instructions in an ISA that define an operation on a set of input scalar operands to produce a single result. Certain situations can arise because of the behavior of the code where multiple similar scalar instructions, such as an add, are executed one after the other and do not depend on one another. To exploit performance in cases like this, where Data-Level Parallelism (DLP) exists, one could use a vector processor. Contrary to scalar processors, each vector register in a vector processor can hold multiple operands and each vector operation operates on all operands held in the vector register. A vector functional unit is the hardware component responsible for performing the operations on vector registers in a Single-Instruction Multiple-Data (SIMD) manner. Of course, taking advantage of vector execution requires the application to be written with vector execution in mind, so that at the assembly level, vector instructions are used rather than scalar instructions. There are a few different ways to accomplish this task, one way is for a programmer to manually write vector assembly but is a tedious and prohibitively time-consuming task. Another similar method is to take advantage of vector intrinsics in high-level languages where library calls are used to map high-level constructs to their vector assembly counterpart. The last method is to delegate the task to the compiler so that a compiler pass

automatically works to translate scalar code to the vector equivalent.

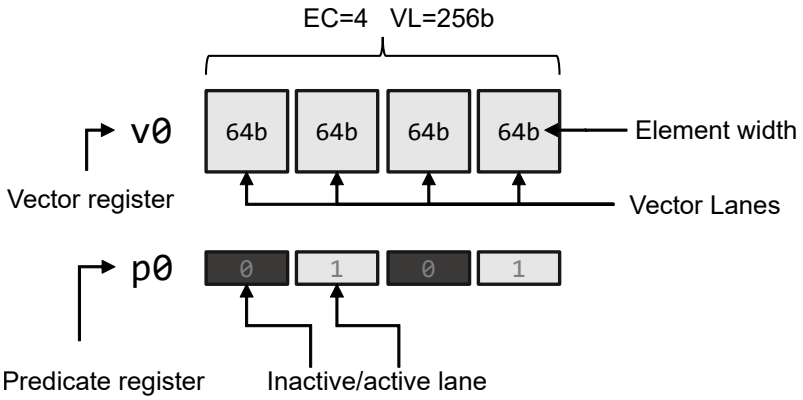


Figure 2.1: Visual breakdown of the components in vector and predicate registers.

2.1 SIMD Execution and Vector Extensions

Figure 2.1 shows a visual breakdown of the components in a vector. A vector register is organized into *vector lanes* with each lane able to hold a single vector element. CPUs support SIMD execution through vector ISA extensions — e.g. Intel AVX [12] and ARM's Advanced SIMD [11] — that define arithmetic and logic vector instructions for a number of data types. Vector operations may operate vertically on the elements in the same lane of the operand vectors, in the case of operations like *add* and *multiply*, or horizontally across lanes within a single vector as in *reduction* operations.

The size of the vector registers or *Vector Length (VL)* can vary greatly between architectures. For instance, Intel has three popular vector extensions: SSE (128 bits), AVX and AVX2 (256 bits) and AVX512 (512 bits) each with different vector lengths. Each vector register can be partitioned to enable storage of operands of different types with varying bit-widths. Thus, the number of lanes in a vector register with a set `VL` varies with the data type. For instance, two 64-bit elements can be held in a 128-bit vector register while four 32-bit elements can be held in the same register. In this thesis the number of lanes in a vector register is called the *vector element count* (`EC`).

2.1.1 Scalable Vectors

Early appearances of vectors in CPU ISAs were meant to introduce support for SIMD execution to accelerate graphics applications. To do so at a low cost, a small amount of additional hardware was added to allow for SIMD execution on an existing scalar register i.e. a 64-bit register could hold two 32-bit values or four 16-bit values and SIMD instructions could perform four 16-bit arithmetic operations at a time. No additional registers were introduced beyond the existing scalar register and thus vector extensions in commodity hardware were limited to small fixed-length vectors. Vector length increased as processors attempted to exploit more DLP using SIMD extensions.

Logic design complexity, power consumption, chip area, and the target application domains influence the choice of `VL` and thus the decision of vector length differs depending on the deployment of a vector system. For example, the CRAY-1 [27] supercomputer featured 4096-bit vector registers to exploit applications with a high amount of DLP while a typical low-power Intel processor enabled with Streaming SIMD Extension (SSE) has 128-bit vector registers. For years, the length of vectors has been coupled to the instruction encodings leading to unnecessary proliferation of instructions when architectures extend the vector length [25]. To prevent this proliferation while addressing the question of what vector length to use, ARM's Vector-Length Agnostic (VLA) SVE [21] decouples the VL from the vector instruction encoding and delegates the choice of `VL` to the hardware implementation. Another ISA favoring to follow the emerging `VLA` architecture is the RISC-V "V" vector extension [22].

In a VLA architecture, the ISA is cleverly designed to remove references of the VL inside instruction definitions. In this sense, VL is abstracted at the ISA level in VLA architectures. To accomplish this, instructions are introduced whose result depends on the VL of the system the instruction is executed on. For example, one common case the VL is needed in application code is to store data consecutively into an array. Such a situation arises when traversing memory in a loop. When vectorized, each store instruction results in multiple elements being stored into the array. In this case, the code must be informed of

the VL to update the address for the next store. In fixed-length architectures, the VL is known at compile time so that updating the pointer to the array is trivial. In comparison, VLA architectures introduce an instruction in which the VL is returned at runtime, such as SVEs `incd` instruction that increments a scalar register by the number of 64-bit elements that can fit inside a vector register defined by the hardware executing the program.

```
1 double * a, * b, * c, * d;
2
3 for (int i = 0; i < 1000; i++) {
4   if (a[i] < b[i])
5 B1:   c[i] = (a[i]*a[i] + b[i]*b[i]) / (d[i]*d[i]);
6   else
7 B2:   c[i] = (a[i] - b[i]) / d[i*2];
8 }
```

Figure 2.2: A simple data-parallel loop with control flow

2.2 Loop Vectorization

As mentioned, one of the methods to take advantage of SIMD execution in a vector ISA is by the programmer manually writing the application code in vector assembly. Often, only the hot loops in the program, uncovered through profiling, are rewritten in vector assembly to gain the most performance from re-writing a small section of code. Hand writing SIMD assembly fine-tunes performance but is a tedious and error-prone task that requires the programmer to have deep knowledge of the target architecture to achieve good performance gains. Vector intrinsics, exposed in a high-level programming language through a library, offer a compromise between productivity and performance. However, the most productive approach for the generation of vector code relies on a compiler that transforms sequential scalar code into vector code through a process called *automatic vectorization* or *SIMDization* [34]. Often, the majority of runtime of an application is spent in data-parallel loops, as such, loops are the primary target for vectorization. Modern compilers implement *loop-vectorizers* that *widen* each scalar operation appearing in a loop into a vector

```
// x0: i = 0 x1: d x2: c x3: b x4: a w3: 1000

whilelo p2.d, xzr, w3
.L1:
  ld1d z1.d, p2/z, [x3, x0, lsl #3]
  ld1d z0.d, p2/z, [x4, x0, lsl #3]
  fcmlt p0.d, p2/z, z0.d, z1.d
  bic p1.b, p3/z, p3.b, p0.b
  movprfx z2, z0
  fmul z2.d, p0/m, z2.d, z0.d
  fmla z2.d, p0/m, z1.d, z1.d
  index z4.d, x0, #2
  ld1d z3.d, p1/z, [x1, z4.d, lsl #3]
  fsub z0.d, p1/m, z0.d, z1.d
  ld1d z1.d, p0/z, [x1, x0, lsl #3]
  fdiv z0.d, p1/m, z0.d, z3.d
  fmul z1.d, p0/m, z1.d, z1.d
  movprfx z0.d, p0/m, z2.d
  fdiv z0.d, p0/m, z0.d, z1.d
  st1d z1.d, p2, [x2, z0, lsl #3]
  incd x0
  whilelo p2.d, w0, w3
  b.any .L1
```

Figure 2.3: SVE assembly of the vectorized loop in Figure 2.2

operation where consecutive iterations of the scalar loop map to lanes of a vector. To ensure correctness, an extensive list of legality checks is performed prior to loop vectorization. One important check uses memory dependence analysis [17], [39] to ensure that either there are no loop-carried dependencies between memory accesses in the loop or that the dependence distance is greater than the number of elements in each vector. Control structures, such as an `if-then-else` statement, introduce control dependencies that must also be addressed by the vectorizer as discussed in Section 2.3.

Figure 2.2 shows an example of a vectorizable loop written in C. The vectorized version of this example, in SVE assembly, is shown in Figure 2.3. Assuming that a `double` data type contains 64 bits and the architecture implements vector registers that are 512-bits wide, each vector register will have eight lanes (`EC = 8`). The comment on line 1 specifies the initial values in the corresponding scalar registers. After vectorization, lines 5 and 7 from Figure 2.2 are translated into various predicated vector arithmetic instructions. For example, the fused-multiply-add vector instruction on line 11 of Figure 2.3 will each perform eight multiply-add operations that correspond to iterations `i` to `i+7` of the scalar loop. The governing predicate `p` register in each instruction contains Boolean values to indicate which lanes are active. The values in the `p` registers are generated by the `fcmplt` and `bic` instructions that compute the predicate for the *if* and the *else* block in Figure 2.2.

2.2.1 Gather and Scatter Memory Accesses

While contiguous memory accesses are well-supported by most vector extensions and trivial to generate vectorized code for, non-contiguous accesses are not. For instance, the load `d[i*2]` on line 7 has a stride of two with consecutive lanes of this load accessing memory addresses: $\{d + 0,\ d + 2,\ ...,\ d + 2 * (VL - 1)\}$. More capable vector extensions feature *gather load* and *scatter store* instructions to move data between contiguous vector lanes and non-contiguous memory addresses. SVE is equipped with a gather load instruction, shown on line 13, that enables the vectorization of the load `d[i*2]`. In ISAs that do not support gather and scatter instructions, these non-contiguous accesses can be

accomplished by scalarizing the memory access but such a solution is much slower than a tailor made gather/scatter instruction.

2.3 Control Divergence

Control dependencies are introduced by conditional statements and must be handled properly when generating vectorized code. For example, in Figure 2.2 blocks B_1 and B_2 are control dependent on the block that computes the condition `a[i] < b[i]` (line 4). This control dependency is loop variant because the condition depends on the value of the loop induction variable `i`. A loop-variant dependency is a *divergent condition* — such conditions are also referred as *varying conditions [23]* — while a loop-invariant control-flow dependency, where the condition does not depend on the loop induction variable, is called a *uniform condition*. Uniform conditions can often be hoisted out of the loop by loop unswitching [2].

2.3.1 IF-Conversion

If-conversion [1] — also referred to as *control-flow linearization* [3], [23] — is a technique to enable the vectorization of loops containing control flow. *If-conversion* is applied to the control-flow graph of a loop prior to loop vectorization to convert control dependencies into data dependencies. If-conversion replaces a branch statement with the computation of predicates, one for each successor of the branch and assigns these predicate accordingly to each successor block in the if-converted code. As the branch in the original code is removed, all successor blocks will execute at runtime and in this aspect, the blocks are linearized. The predicated statements replace the behavior of the branch to control execution as only predicated statements whose predicate evaluates to true at runtime produce a result, thus the control dependence that once existed due to the presence of a branch now exists as a data dependence on the variable holding the predicate.

Predicated vector ISAs, like AVX-512 and ARM SVE, provide direct support for IF-conversion. In predicated ISAs, predicated vector instructions define

an additional operand, the predicate or mask register, that holds a Boolean value corresponding to each lane of a vector register. Lanes associated with a true predicate value produce a result in a predicated operation and are called *active lanes* while lanes that hold a false predicate do not and are referred to as *inactive lanes*.

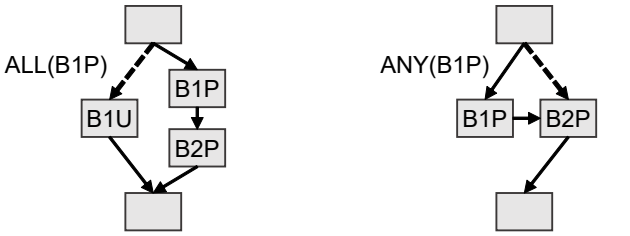
2.3.2 Uniform and Divergent Vectors

A vector is *uniform* either when all of the lanes are active or all of the lanes are false and is *divergent* when some of the lanes are active while others are not. The IF-converted and vectorized code for Figure 2.2 is shown in Figure 2.3 where all branches are removed and replaced by instructions to produce vector predicates. Line 7 shows the `fcmlt` instruction to generate the predicate for B_1 and the `bic` instruction on line 8 that generates the predicate for B_2. Operations inside blocks B_1 and B_2 are vectorized and predicated by these predicates.

In ISAs that do not support predicated execution the control dependencies can be handled via IF-selection [37] or scalarization. Vector `select` instructions perform an element permutation that retrieves values from the first input vector if the lane is active or from the second otherwise. Scalarization performs a scalar operations on each lane of the vector iteratively and may require expensive packing and unpacking of the lanes of the vector register into scalar registers.

2.3.3 Branch-on-Superword Condition Codes

Predicated vector instructions selectively enable and disable lanes, with disabled lanes producing no result. The ratio of active lanes to total lanes in a vector is called the *vector utilization*. Lower utilization leads vector instructions to consume CPU resources while producing few or no useful results. In addition, most modern vector architectures feature VL-time performance rather than density-time performance where the number of inactive lanes present in a vector does not affect the latency of the predicated instruction [7]. Inactive lanes in VL-time architectures are wasted opportunity and thus, it is desirable to have code with high vector utilization.



(a) All boscc (b) Any boscc

Figure 2.4: BOSCCs used to optimize the IF-converted code

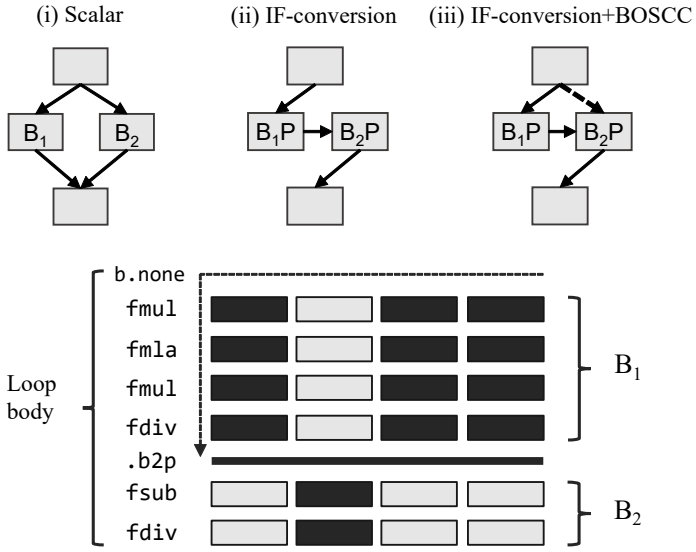


Figure 2.5: Example vector execution of Figure 2.2 with an *any* BOSCC inserted to skip *B1*. Light coloured squares represent active lanes while dark inactive lanes.

BOSCC [30], [31] instructions can be used to elide execution of vector instructions that have no active lanes. A BOSCC instruction is a branch instruction that branches based on whether a super-word (vector) satisfies a certain condition. For instance, SVEs condition codes `NZCV` are set as the result of vector compare instructions, e.g. `cmpeq`. Vector compares clear the flag `Z` if any lane of the resulting predicate becomes active. A `b.any` BOSCC in SVE, as shown in Figure 2.5, branches when the flag `Z` is not set, only taking the branch if at least one active element exists in the predicate.

Two common BOSCCs are the *any* BOSCC that branches when any lane of the vector is active and the *all* BOSCC that branches only if all lanes are active. Figure 2.4 shows how *any* and *all* BOSCCs can be used to optimize

vectorized code for Figure 2.2. A dashed line indicates the edge a BOSCC branch introduces into the control flow graph. Figure 2.4a inserts an *all* BOSCC in a manner described in WCCV [36] to branch to block B_1U if every lane is active for the B_1 block. Since B_1U is only executed if all lanes are active, this block can contain unpredicated vector code. Figure 2.4b uses an *any* BOSCC instruction to skip the execution of a predicated block when no lanes are active; this method of utilizing an *any* BOSCC is described in Partial Control Flow Linearization [23]. The *all* BOSCC instruction improves lane utilization in the presence of biased branches while *any* BOSCC instruction bypasses IF-converted code in blocks that rarely execute with active lanes [36].

Figure 2.5 shows one example of a possible execution of the control flow graph in Figure 2.4b once vectorized. This example execution illustrates how BOSCCs can break down in the presence of divergent vectors. In the example, the predicate for B_1P contains only a single active lane, as the BOSCC condition is true when *none* of the lanes are active, the branch to skip B_1P is not taken and executes the vector code with only a single active lane (25% utilization). Increasing vector length reduces the effectiveness of BOSCC instructions as divergent vectors may become more likely.

2.4 Summary

This chapter presented an overview of vectorization and its place in compilers as an automatic pass to optimize loops, modern vector ISAs, the qualitative impact of control-flow divergence on vector performance, and the use of BOSCC instructions to optimize divergent vector code. All of these topics are required to understand the rest of this thesis. Chapter 3 will present the ALC permutation and details two loops transformations that utilize ALC to optimize divergent vector code. Chapter 4 will introduce four case studies that analyze the efficacy and characteristics of applying the ALC transformations to loops suffering from control divergence from the SPEC CPU2017 benchmark suite.

Chapter 3

Active-Lane Consolidation

Figure 1.1 shows that using BOSCCs after vectorization and IF-conversion works well to optimize execution on short vectors as they are likely uniform because of the small amount of lanes they hold. The figure also indicates that the benefit of optimization with BOSCCs decreases as the vector size increases with the improvement becoming negligible at a vector length of 2048 bits. Thus, architectures that implement modern vector ISAs, which are supporting increasingly longer vectors, require new compilation approaches. In the effort towards this goal, this chapter introduces Active-Lane Consolidation (ALC), a vector permutation that increases vector uniformity to create more opportunities for profitable deployment of BOSCC instructions.

An application of ALC to a hot loop from the NAB (Nucleic Acid Builder) benchmark — a molecular-dynamics application that is part of the SPEC CPU 2017 suite — illustrates how this transformation can lead to performance improvements. Figure 3.1 shows a simplified excerpt of one of NAB's hot loops where, for each iteration of the loop, only one of the B_1-B_5 blocks in the `if-else-if` chain is executed. No loop-carried memory dependencies exist within this loop and thus it can be vectorized. Profiling the run-time branch behavior reveals that B_2 is executed in approximately 85% of the iterations. However, iterations that do not execute block B_2 are interspersed with iterations that execute B_2, leading to divergent vectors when this loop is vectorized. To improve the performance of this vectorized loop, ALC consolidates active lanes from divergent vectors in consecutive iterations into uniform vectors.

```
1  for (k = 0; k < lpears[i]+upears[i]; k++) {
2    ...
3    r2 = ...
4    diji1i = ...
5    dij = r2 * dij1i;
6    if (dij > rgbmax - sj) {       // C1
7      B1:
8      sumi -= ...
9    } else if (dij > 4.0 * sj) {   // C2
10     B2:
11     dij2i = dij1i*dij1i
12     ...
13     sumi -= ...
14   } else if (dij > ri + sj) {    // C3
15     B3:
16     sumi -= ...
17   } else if (dij > fabs(ri - sj)) { // C4
18     B4:
19     sumi -= ...
20   } else if (ri < sj) {          // C5
21     B5:
22     sumi -= ...
23   }
24 }
```

Figure 3.1: Simplified loop from NAB

The runtime bias toward condition C_2 implies that in many iterations of the vectorized loop, the linearized basic blocks B_1, B_3, B_4, B_5 in the if-converted code are unnecessarily executed. One solution is to unroll the vectorized loop and then to use ALC to consolidate the active lanes of divergent vectors into a single vector but has the downside of being limited to consolidating active lanes only from two consecutive iterations. The other solution, iterative ALC, decouples the consolidating vector from the loop iteration allowing active lanes from any iteration during the loop to be consolidated. In both schemes, the consolidated vector is stealing active lanes from future iterations of the vectorized loop in an attempt to form a uniform vector. If ALC is able to reorganize the lanes of two divergent vectors into a uniform vector, then BOSCCs can ensure that the vector loop only executes the necessary conditional block for this uniform vector. In the NAB example, ALC finds new opportunities to skip unnecessary executions of basic blocks B_1, B_3, B_4, and B_5 by consolidating the lanes in divergent vectors from subsequent iterations of the vectorized loop that execute block B_2.

Based on the predicates generated by IF-conversion, ALC moves as many of the active lanes of two vectors as possible into a single vector and fills a second vector with the remaining inactive lanes to handle the case when those inactive lanes follow a different path of control. Once the ALC permutation is applied on a pair of vectors, the order of the lanes will have changed and for correctness, all vectors that are used as an operand inside a consolidated block must be permuted in the same fashion. This chapter introduces the *indexed-based inter-register permutation* to solve this in light of the large number of instructions required by ALC in actual implementation. The ALC permutation is complex in comparison to existing vector permutations so that an implementation using only current instructions may be expensive depending on the capability of the chosen ISA. This chapter details a proposal for a new vector permutation using SVE as the base ISA that would solve some of the shortcomings of implementing ALC using only existing instructions.

The remainder of the chapter is organized as follows: Section 3.1 introduces the operation of the ALC permutation and the first ALC transformation: loop-

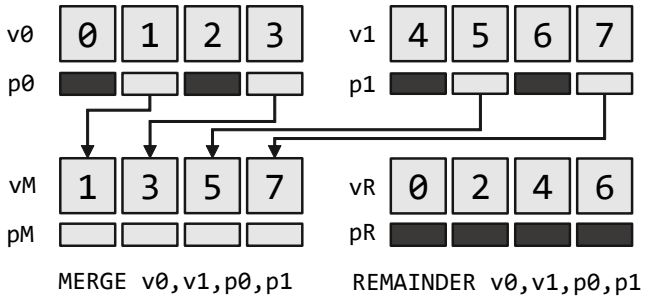


Figure 3.2: The ALC permutation performed on two divergent vectors. Light coloured squares represent active lanes while dark coloured lanes inactive. Inactive lanes are moved into the remainder vector `vR`.

unrolling ALC. In addition, this section details issues caused by the change in lane order and describes the necessary steps to resolve them. Section 3.2 describes the implementation of the ALC and index-based inter-register permutations in SVE after giving a brief overview of the standard SVE permutations required to implement ALC. Section 3.3 details an ISA extension to accommodate the ALC permutation. Finally, Section 3.4 introduces the alternate method of exposing candidate vector registers for consolidation: iterative ALC.

3.1 Optimizing Divergent Vectors with ALC

This section introduces the operation of the ALC permutation and its use in the loop-unrolling ALC transformation. First it provides a description of the lane-order consistency problem and then presents the *index-based inter-register* permutation as a solution.

3.1.1 Active-lane Consolidation Permutation

Figure 3.2 illustrates the application of ALC to two vector registers based on their predicate values. In this example the vectors `v0` and `v1` contain the indices of the loop (created with an `index` instruction) and the predicate vectors `p0` and `p1` indicate the lanes that are active (light gray) or inactive (dark gray). ALC attempts to fill the merged vector (`vM`) with active lanes spread between the two input vectors. The remaining lanes, active or not, are placed into the remainder vector (`vR`). Let `AL` be the number of active lanes. If `AL` $\geq$ `VL`, then

vM is uniform with all lanes active. If AL $\leq$ VL, then vR is uniform with all lanes inactive.

If vM is a uniformly active vector with respect to a condition C_2 after consolidation then BOSCCs will execute only B_2. Given that the conditions of the if-else-if statement are mutually exclusive, the consolidated vector will have an all-false predicate for conditions other than C_2 and thus BOSCCs can bypass execution of the blocks these conditions guard. In general, there are no guarantees about the uniformity of the remainder vector vR and it may have to be processed through the entire linearized graph. However, an additional BOSCC can be inserted to check if vR is all-false for C_2 to further optimize execution.

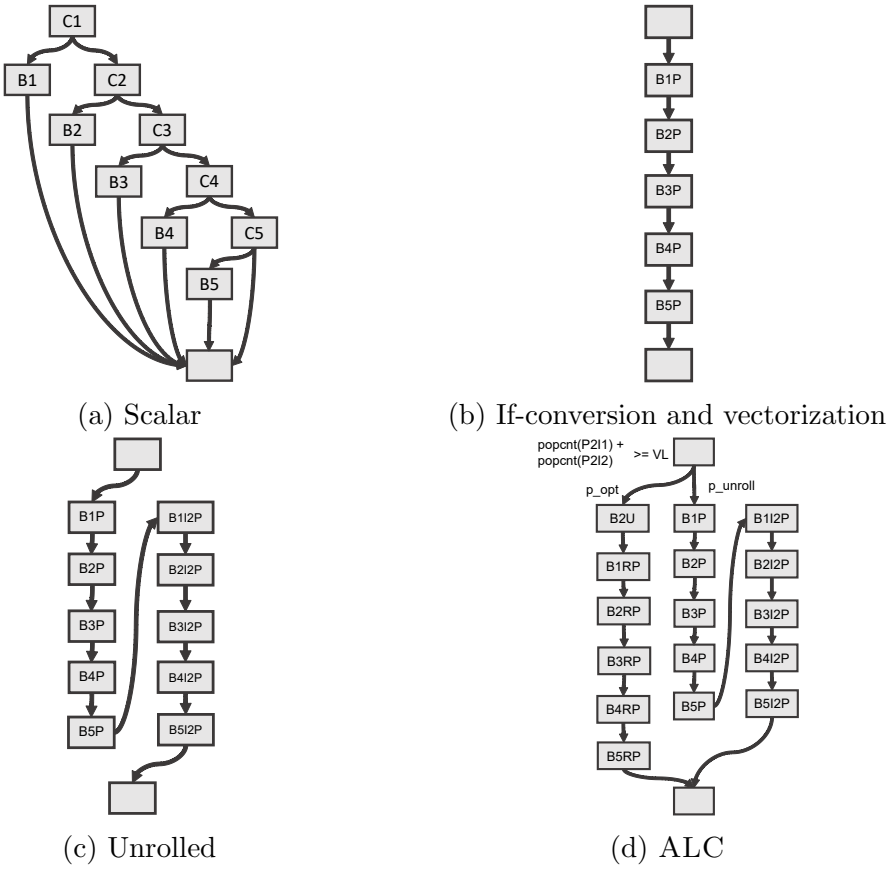
Figure 3.2 presents ALC as a permutation of the two vectors v0 and v1 to create two new vectors vM and vR. Abstractly, ALC can be viewed as a compaction on a vector created by concatenating v0 and v1. The AVX512 compress and the SVE compact instructions already perform the compaction but only on a single vector register and do not retain inactive lanes in the result.

Similar to the compact permutation, the ALC permutation retains the relative order of the active lanes. However, due to complexities encountered in the implementation for SVE, this is not the case for inactive lanes. The lack of relative order in the inactive lanes did not have an impact in our experiments but may become important later when new features are added to vector ISAs, for example, instructions to detect intra-vector dependencies [4].

3.1.2 Loop-Unrolling ALC

The goal of ALC is to consolidate lanes from multiple vectors to create a uniform vector. Thus, multiple vectors must be available for this consolidation. Unrolling the vectorized and IF-converted loop by a factor of two yields two vectors that can be consolidated by the ALC permutation into two new vectors, one containing mostly active lanes and the other containing mostly inactive lanes. This transformation is called "Unrolling ALC".

Figure 3.3a shows the control flow graph of the scalar loop from the NAB



(a) Scalar (b) If-conversion and vectorization

(c) Unrolled (d) ALC

Figure 3.3: Control flow graphs after: (a) Scalar, (b) If-conversion and vectorization of (a), (c) Unrolling of (b), (d) ALC on (c).

kernel in Figure 3.1. The conditions C_1 to C_5 control the execution of blocks B_1 to B_5. Two transformations are applied to the code from Figure 3.3a to Figure 3.3b: IF-conversion and vectorization. Vectorization widens each scalar instruction in Figure 3.3a so that each vector instruction in Figure 3.3b performs the operation for `EC` iterations of the scalar loop. The letter `P` after the block name in Figure 3.3b indicates that the block is predicated by a Boolean expression composed of the conditions that control each block's execution. For instance, the condition for the predicated block B_2P in Figure 3.3b is $C_2 \wedge \neg C_1$ because B_2 executes when C_2 is true (Figure 3.1, Line 9) and C_1 is false (Figure 3.1, Line 6). From Figure 3.3b to Figure 3.3c the linearized and vectorized loop is unrolled by two so that each iteration of the new loop contains two copies of the body of the loop in Figure 3.3b. Blocks to process the second iteration are annotated with I_2 in Figure 3.3c.

The most executed block in this loop, B_2, is the target for consolidation. In Figure 3.3d the active lanes of the vectors from the first and second iterations of the unrolled loop are consolidated by ALC based on the predicates for B_2P and the other in B_2I_2P. Let P_2I be the predicate of B_2P and P_2I_2 be the predicate of B_2I_2P, then, after ALC, `vM` is a uniform vector if and only if $popcnt(P_2I) + popcnt(P_2I_2) \geq VL$. This condition is checked by counting the number of active lanes using population-count instructions on the pair of predicate registers corresponding to the condition that is being consolidated. If this condition in Figure 3.3d is true, then the code can branch to the optimized path, labeled `p_opt`. Otherwise, the unoptimized path that executes the if-converted code `p_unroll` is taken. The `p_opt` path bypasses execution of B_1P, B_3P, B_4P and B_5P for the merged vector `vM` because it is uniform with respect to C_2. This is reflected in Figure 3.3d as B_2U is the only block that processes `vM`. The `p_opt` path must still conservatively process `vR` through the fully if-converted path represented by blocks B_1RP to B_5RP for correct execution because `vR` is not guaranteed to be uniform. In some instances, `vR` may have no lanes active for this predicate and thus, inserting an *any* BOSCC to skip the block B_2RP, may be worthwhile.

The idea of consolidating active lanes is applicable to more than a pair of

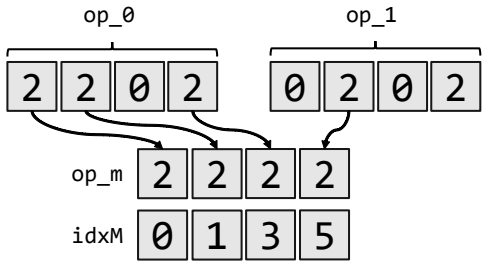


Figure 3.4: Inter-vector permutation to create the merged operand by permuting the concatenated vectors of the unrolled iterations based on the consolidated index vector `idxM`

vectors; ALC could unroll the loop by a factor greater than two to expose three, four, or more vectors for consolidation. Given the limitations of permutation instructions currently in SVE, this thesis only considers unrolling by a factor of two and consolidating the resulting pair of vectors. Unrolling the loop further and consolidating more than a pair of vectors would add additional complexity into the ALC permutation.

3.1.3 Tracking Lane Indices Through Permutation

Often, several vector operands within a single conditional block must be consolidated in the same fashion. This is necessary in order to ensure that the lanes of vectors defined outside of the block being consolidated are in the correct consolidated order when used within the consolidated block. Figure 3.5 illustrates this issue: after vectorization, the order of the lanes of variable `op` matches the unit-stride of the loop induction variable `i` with lane 0 holding the value for iteration `i`, lane 1 holding the value for iteration `i+1` and so forth. After the ALC permutation is performed, the variable `op` must also be permuted so the lanes match the order resulting from ALC.

In some vector extensions, such as in SVE, the implementation of the ALC permutation shown in Figure 3.2 is expensive and therefore a more efficient solution consists of performing ALC on a pair of vectors holding the indices 0 to VL-1 and VL to 2*VL-1 and using the result in an index-based inter-register permutation. For instance, in Figure 3.2 `EC` = 4 and the vectors `v0` and `v1` hold the indices in the original order that the induction variable is incremented.

After ALC, the consolidated index vectors are `vM` and `vR`, which are used to permute all other operands used in the conditional block. The indices in `vM` and `vR` are also used to compute the addresses of gather loads and scatter stores appearing inside the consolidated block.

3.1.4 Permuting Instruction Operands

Two issues deserve further examination:

1. a consolidated block may use an operand that was defined outside of the block; and
2. a load inside a consolidated block may depend on the loop induction variable.

For instance, Figure 3.5 shows a scalar loop while Figure 3.6 shows the loop after vectorization, unrolling, and ALC is applied on the `if` condition. The variable `op` is defined outside of the conditional block B_1 being consolidated; and the load `C[i]` depends on the loop induction variable `i`.

The notation `(pred) expr;` in lines 28-36 of Figure 3.6 represents predicated execution where `pred` is the predicate and `expr` is the expression executed if `pred` evaluates to true. The annotation `_0` or `_1` indicates to which of the two iterations exposed by unrolling each variable belongs. The original consecutive index vectors are created in lines 2 and 3. Line 19 performs ALC based on the predicates generated by the condition in line 4 of Figure 3.5 to create the new index vectors `idxM` and `idxR`. These indices are used to permute vectors `op_1` and `op_2`, in lines 22 and 26 before their use in the consolidated block. Figure 3.4 illustrates the index-based inter-vector permutation that appears in line 22 in Figure 3.6 within the consolidated block (B_1).

Load/store operations that are inside a consolidated block are converted to gather/scatter operations indexed by the consolidated induction vectors. For instance, ALC requires that the contiguous load `C[i]` in Figure 3.5 be converted to the gather loads indexed by `idxM` or by `idxR` in lines 23 and 28 of Figure 3.6. Gather and scatter accesses tend to be slower than contiguous

```
1 for (int i = 0; i < N; i++) {
2   op = A[i] * 2
3
4   if (op) {          // Consolidate
5     out += op * C[i] // B1
6   } else {
7     out += op / D[i] // B2
8   }
9 }
```

Figure 3.5: Example loop where the op variable must be permuted and contiguous loads converted to gather loads to correct the order to the consolidated iterations if ALC is applied.

accesses because they issue more micro-operations and may access a larger number of cache lines.

3.2 Active-Lane Consolidation with SVE

To demonstrate the efficacy of ALC in mitigating divergence we develop an implementation using ARM's SVE [21], a widely adopted extension with supporting tools for code generation and emulation. The permutation primitives and transformations presented in Section 3.1 are also applicable to other long-vector architectures such as RISCV "V". This section details how the ALC primitives map to SVE instructions and presents a brief overview of the instructions' semantics. An analogous mapping could be developed to apply ALC to other vector ISAs. This section concludes by outlining challenges discovered in mapping ALC to SVE and presenting additions to SVE that help mitigate these challenges.

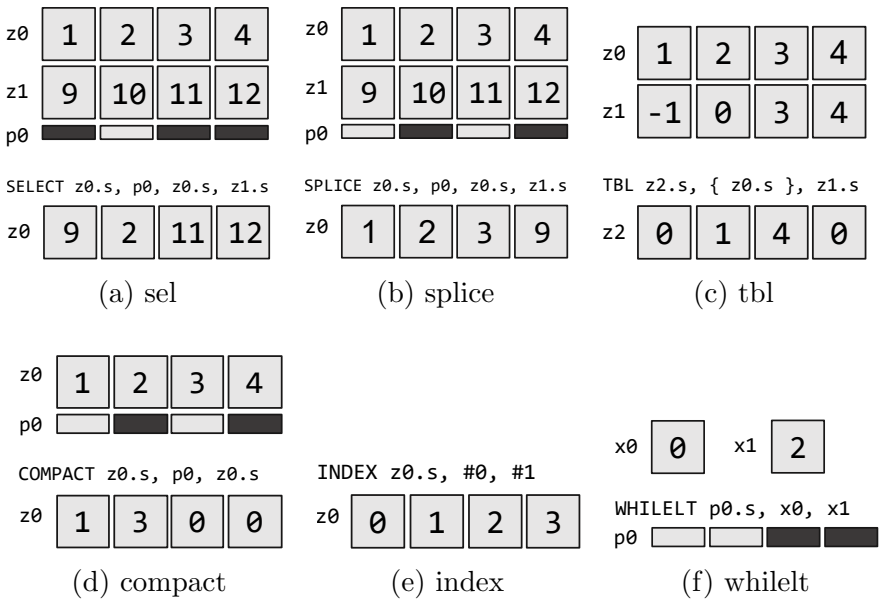
3.2.1 SVE Permutations

The following SVE instructions are illustrated with examples in Figure 3.7 where a light-grey predicate is true and a dark-grey predicate is false.

select: For each true predicate, take the elements from the first input vector register and for each false predicate take the elements from the second input

```
for (int i = 0; i < N; i += 2*VL) {
  idx_0 = index(i,i+VL,1) ;   // 0,1,2,3
  idx_1 = index(i+VL,i+2*VL,1); // 4,5,6,7

  a_0 = vld(A[i]) ;    // 1,1,0,1
  a_1 = vld(A[i+VL]);  // 0,1,0,1

  op_0 = a_0*2;        // 2,2,0,2
  op_1 = a_1*2;        // 0,2,0,2

  cond_0 = op_0 != 0;  // 1,1,0,1
  cond_1 = op_1 != 0;  // 0,1,0,1

  // Check whether ALC is beneficial, 3+2 >= 4
  if (popcnt(cond_0) + popcnt(cond_1) >= VL) {

    // ALC applied to index vectors
    // idxM = 0,1,3,5; idxR = 2,7,4,6
    idxM,idxR = alc(cond_0,cond_1,idx_0,idx_1,);

    // Process merged vector, no need to execute code for B2
    op_m = permute(op_0,op_1,idxM); // 2,2,2,2
    out += op_m * gather(C+i,idxM);

    // Process remainder vector, if-conversion
    op_r = permute(op_0,op_1,idxR); // 0,2,0,0
    cond_r = op_r != 0;      // 0,1,0,0
    (cond_r) out += op_r * gather(C+i,idxR);
    (!cond_r) out += op_r / gather(D+i,idxR);
  }
  else {
    // Fallback execution, if-conversion
    (cond_0) out += op_0 * C[i];
    (!cond_0) out += op_0 / D[i];
    (cond_1) out += op_1 * C[i+VL];
    (!cond_1) out += op_1 / D[i+VL];
  }
}
```

Figure 3.6: Example loop from Figure 3.5 with vectorization,unrolling and ALC applied. Operands are merged accordingly and contiguous loads inside the consolidate block are converted to gather loads.



(a) sel (b) splice (c) tbl

(d) compact (e) index (f) whilelt

Figure 3.7: Semantics of SVE permutation instructions and index/whilelt instructions used to implement the active-lane consolidation and multi-register permutation.

vector register.

splice: The predicate register defines a range from the first true predicate to the last true predicate. Elements within this range are taken from the first vector and placed at the start of the result register. The remainder of the result register is filled with elements from the second vector register starting from the element in the lowest position.

tbl: In this programmable table lookup illustrated in Figure 3.7c, `z0` is the data register and `z1` holds indices into `z0`. The result register receives elements from the data register based on the values in the index register. Lanes where the index register has a value outside of the interval $[0, \text{VLEN} - 1]$, where VLEN is the vector length, lead to the result register receiving the zero value. For instance, in Figure 3.7c the index 4 is outside of the [0,3] range.

compact: Active elements of the input vector register are placed compactly at the start of the output register. The remaining elements in the output register are zeroed.

index: Creates a vector of incrementing values. Two scalar values, supplied either in registers or as immediate values, specify the start value and the

incrementing step.

whilelt: Given two scalar registers `x0` and `x1`, generates a predicate register where the first `x1-x0` lanes are active and all other lanes are inactive.

The name `whilelt` highlights that the value of the predicate for each lane is set by testing the condition `x0` $<$ `x1` and then incrementing `x0` before testing for the next lane.

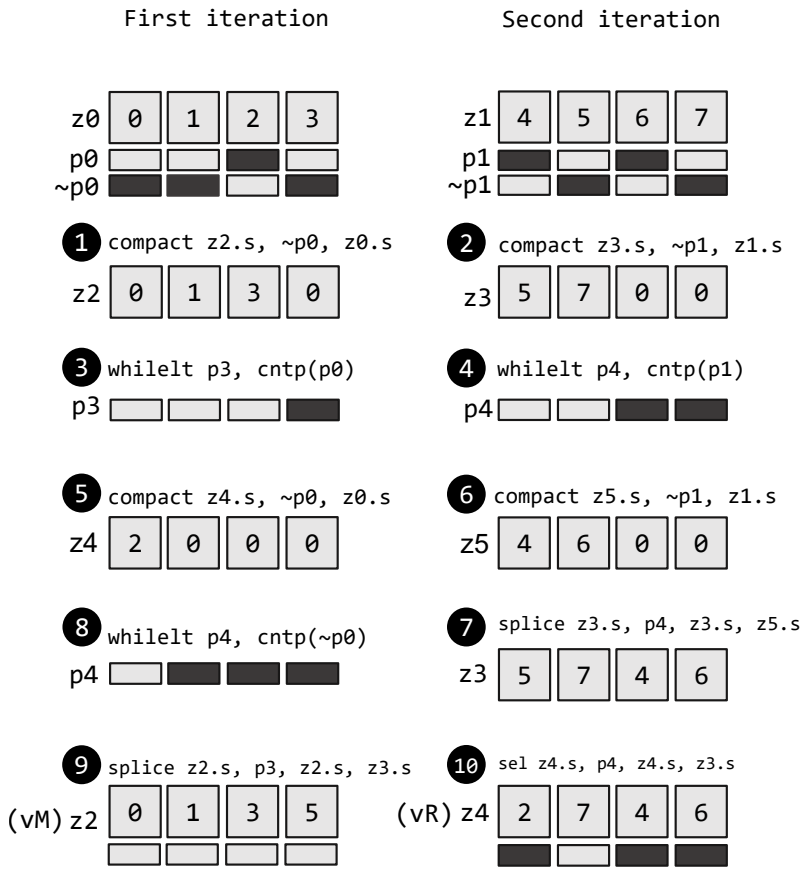


Figure 3.8: Visual example of the ALC permutation implemented in SVE. Active lanes are shown as light coloured squares while inactive as dark coloured squares. For brevity, we use `cntp` as a function but requires another instruction that returns the count of active lanes. In addition, we use `~p0` to represent the logical `NOT` of predicate register `p0`. In 1, once compacted, there are additional zero values appended at the end of the vector which are empty lanes rather than a valid zero index. These are not problematic when used as the predicate will prevent any instruction from using the zeros at the end

3.2.2 Active-Lane Consolidation in SVE

```
1 // EC=4, element count, i.e. VL/DTYPE_SIZE
2 z0: first vector containing indices // = 0 1 2 3
3 z1: second vector containing indices // = 4 5 6 7
4 p0: predicate for z0 // = 1 1 0 1
5 p1: predicate for z1 // = 0 1 0 1
6 p2: governing predicate
7
8 compact z2.s, p0, z0.s       // = 0 1 3 0
9
10 compact z3.s, p1, z1.s      // = 5 7 0 0
11
12 not p3.B, p2/z, p0.B        // = 0 0 1 0
13 compact z4.s, p3, z0.s      // = 2 0 0 0
14 cntp x2, p3, p3.s           // = 1
15
16 not p4.B, p2/z, p1.B        // = 1 0 1 0
17 compact z5.s, p4, z1.s      // = 4 6 0 0
18
19 cntp x0, p1, p1.s           // = 2
20 whilelt p4.s, xzr, x0       // = 1 1 0 0
21 splice z3.s, p4, z3.s, z5.s // = 5 7 4 6
22
23 cntp x1, p0, p0.s           // = 3
24 whilelt p3.s, xzr, x1       // = 1 1 1 0
25
26 whilelt p4.s, xzr, x2       // = 1 0 0 0
27
28 // Merge
29 splice z2.s, p3, z2.s, z4.s // = 0 1 3 5 - all true
30 // Remainder
31 sel z4.s, p4, z4.s, z3.s    // = 2 7 4 6
```

Figure 3.9: Implementation of the ALC permutation in SVE described in Section 3.1.1. Example values show in this listing are consistent with the prior examples.

Figure 3.9 shows the implementation of the ALC permutation in SVE. The variables in lines 2-6 are the input to the permutation: the original loop indices `z0` and `z1`; the predicate registers `p0` and `p1`; and the governing predicate `p2`. The governing predicate controls the execution of the loop-trip and is all true except for the tail-end of the loop when the trip count is not a multiple of `VL` or

if the loop has a data-dependent exit that is taken before the loop terminates by reaching the loop bound. The comments on each line provide an example value for the initialization of these variables and later show the value that results from the execution of each instruction. In this example, the value of the predicates `p0` and `p1` in lines 4 and 5 correspond to the values of `cond_0` and `cond_1` in lines 11 and 12 of Figure 3.5. The SVE population count instruction, `cntp` in shown on line 14 and counts the number of active lanes in a predicate. The `.s` in the instruction indicates the vector element width which is 32 bits in this example.

The SVE ALC permutation shown in Figure 3.9, and illustrated in Figure 3.8, is used to create the consolidated loop induction vectors that are used both for the index-based inter-register permutation of all the operands in the block and for the gather-load and scatter-store operations. After this sequence the merged vector, `z2.s`, contains up to `VLEN` elements from both `z0` and `z1` corresponding to the active lanes in `p0` and `p1` while the remainder vector, `z3`, contains all other elements in `z0` and `z1` that were not consolidated into `z2`. If the sum of the number of active lanes between `p0` and `p1` is greater than the vector length, then `z2` is guaranteed to be uniform with respect to the condition that created the predicates `p0` and `p1`.

3.2.3 Inter-Register Indexed Permutation in SVE

In SVE an index-based inter-register permutation can be implemented using a chain of `tbl` instructions as shown in Figure 3.10. The `tbl` on line 7 selects elements from the first vector data register that corresponds to the first iteration in the unrolled loop. The last lane receives a zero value because the index 5 is out of range for `VL = 4`. Line 10 normalizes the indices to the second vector data register by subtracting the element count from all lanes. Line 11 gathers the values from the second data vector with another `tbl` and the first three lanes take a zero value again because -4, -3 and -1 are outside the range of valid indices. Finally, adding the result of the two `tbl` instructions gives the final merged operand in the order created by performing the ALC permutation.

The inter-register indexed permute makes extensive use of the `tbl` in-

```
// EC=4, element count, i.e. VL/DTYPE_SIZE
z0 = first data vector         // = 2,2,0,2
z1 = second data vector        // = 0,2,0,2
z3 = merged indices            // = 0,1,3,5

// Data from first vector
tbl z4.s, { z0.s }, z3.s      // = 2 2 2 0

// Data from second vector with adjusted index
decw z3.s                     // = -4 -3 -1 1
tbl z5.s, { z1.s }, z3.s      // = 0 0 0 2

// Combine data from TBLs
ptrue p0.s
add z4.s, p0, z4.s, z5.s      // = 2 2 2 2
```

Figure 3.10: SVE-specific pseudo-code for the inter-register permute as introduced in Section 3.1.4.

struction. ARM's previous vector extension, Advanced SIMD, featured a `tbl` instruction that took a register-table as input i.e. two or three consecutive vector registers. However, the `tbl` in the SVE specification does not include this feature because the SVE designers considered them not naturally vector-length agnostic [21]. The proposed inter-register indexed permute presents a case for the inclusion of a multi-register-table permutation in SVE because indices into the vector table can be generated by other VLA instructions such as `index`.

3.3 Proposal for native support in SVE

There are two shortcomings in the SVE design that introduce inefficiencies when implementing ALC: there are no native instructions that directly map to the ALC permutation, and there is no support for indexed permutations on multi-register tables. These shortcomings make it difficult to re-order the lanes between a pair of vectors.

This section proposes two additional instructions to SVE that introduce the active-lane-consolidation permutation as an instruction and support for the multi-register-table permutation. Figure 3.11 shows the proposed syntax for this new instruction `consolidate` illustrating the consolidation of the adjacent

```
// Two-register group
consolidate {z0, z1}, {p0.t, p1.t}

// Three-register group
consolidate {z0, z1, z2}, {p0.t, p1.t, p2.t}

// Four-register group
consolidate {z0, z1, z2, z3}, {p0.t, p1.t, p2.t, p3.t}
```

Figure 3.11: Proposed instruction to support the active-lane consolidation permutation in SVE

```
// Two-register group
tbl z0.t, {z0, z1}, zN.t

// Three-register group
tbl z0.t, {z0, z1, z2}, zN.t

// Four-register group
tbl z0.t, {z0, z1, z2, z3}, zN.t
```

Figure 3.12: Proposed extension of `tbl` to perform an multi-vector table lookup

register group `{ zN, zN+1 ... zY }` based on the active lanes given by the predicate group `{ pN.t, pN+1.t .. pY.t }` with an element size of t. Similar to the fixed-length `tbl`, the input operands for the proposed consolidation instruction are provided as a list of consecutive vector registers to allow consolidation of more than two registers. Following SVE's destructive operation scheme, the input register group is also used as output.

Alternatively, a design could extend the existing `compact` instruction to operate on a register group because the semantics of `compact` are very similar to the semantics of `consolidate`. Such extension would need to change the semantics of `compact` so that the inactive lanes are retained instead of being zeroed as the `compact` instruction currently does.

The second shortcoming of SVE in the context of ALC is the lack of an indexed permutation on a scalable vector. In the current SVE design, `tbl` takes a single vector register as input while in Advanced SIMD the fixed-length version of `tbl` allows up to four consecutive registers as input. Figure 3.12

shows a proposed extension of `tbl` where the last vector operand is a vector of indices into the register group defined within the curly braces. The proposed `consolidate` operation could also be used to achieve the same result but requires a predicate group to be supplied to the instruction.

3.4 Iterative ALC

A limitation of relying on the unrolling of vectorized loops to apply ALC is that consolidation will only consider active lanes within the vectors exposed by unrolling. Practically, there may be many more opportunities to form a uniform vector if not limited to consolidating consecutive iterations. This section presents Iterative ALC, a code transformation that overcomes this limitation by consolidating active lanes across an arbitrary number of iterations of a loop. The insight that led to the design of this Iterative ALC transformation is that the merged vector can persist across loop iterations and consolidate active lanes until the merged vector becomes uniform, at which point it can be processed with 100% utilization.

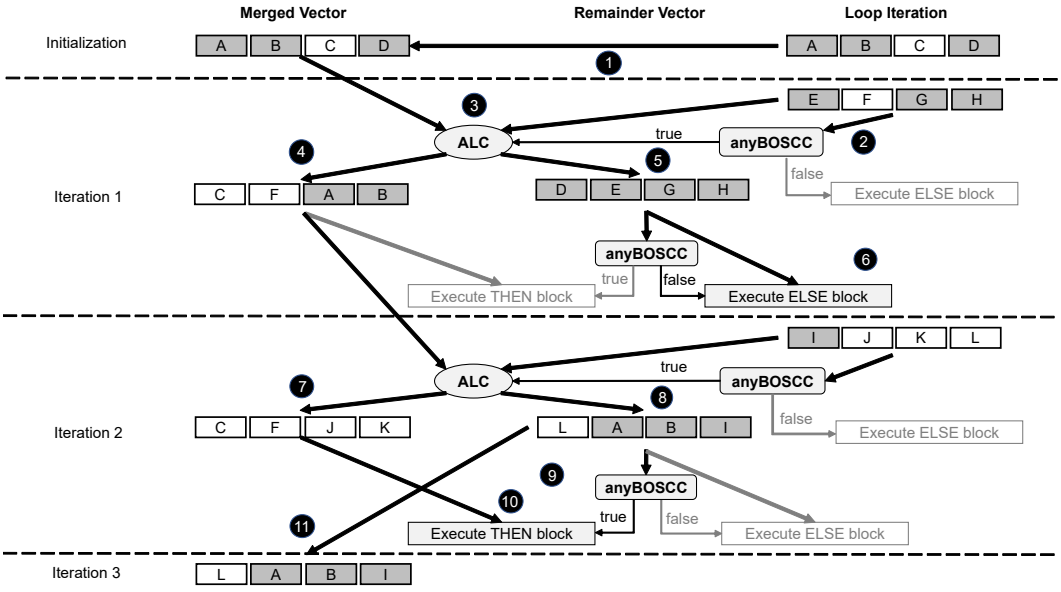


Figure 3.13: Example of iterative ALC. Active lanes are shown as white squares and inactive as dark squares.

Figure 3.13 shows an example of Iterative ALC applied to a loop with an `if-else` statement. The ALC permutation primitives and their applications

are detailed in Section 3.1. In the figure, a thin line indicates a control line and a thick line indicates that all the lanes of the vector are used in the operation. Greyed lines and blocks indicate code that is not executed in this example. Iterative ALC initializes the merged vector using the first iteration of the vectorized loop ❶. In each iteration, an *any* BOSCC checks for active lanes in the incoming iteration ❷ to determine if the ALC permutation should be applied ❸. If the incoming vector is already uniform then ALC is not applied as a BOSCC can already exploit this vector. The ALC permutation produces the updated merged vector ❹ that now contains any active lanes that were present in the incoming vector and also produces remainder vector ❺. In iteration 1, after ALC, the remainder vector ❺ is uniformly inactive so that the BOSCC check fails and the `else` block ❻ is executed. In iteration 2, the *any* BOSCC ❾ on the remainder vector ❽ succeeds, revealing that not all active lanes could be moved into the merged vector ❼ and thus the merged vector must now be uniform. The `then` block ❿ can then be executed with 100% utilization. Once the uniform merged block is executed, the remainder vector in that iteration is set as the merged vector for the next iteration ⓫ as illustrated in iteration 3.

Iterative ALC is most beneficial in loops where a single condition needs to be evaluated to determine the flow of control — loops that contain a simple `if-else` control flow or with a single `if` statement rather than loops containing long `if-else-if` statements — because in the presence of more complex control flow iterative ALC would have to save all lanes of operands appearing in each control path. In addition, if ALC is performed to consolidate active lanes, all saved vector operands must also be permuted to correct the order, further adding to the complexity of operand merging. In the case of a single `if` statement with no `else` block, execution can be completely bypassed. In addition, because there is no need to retain inactive lanes, the ALC permutation shown in Figure 3.9 can be simplified. This simplification is not shown in detail but involves removing instructions in Figure 3.9 dealing with retaining the inactive lanes of the vectors being consolidated.

In the first transformation to enable ALC, described in Section 3.1.2, the

number of active lanes available to be consolidated is limited by the unrolling factor. The unrolling transformation uses an unrolling factor of two because the combinatorial complexity of the ALC permutation increases with the unrolling factor. Larger unrolling factors are also limited by the number of vector registers available because unrolling increases register pressure. In contrast, iterative ALC is able to consolidate active lanes of vectors from any subsequent iteration of the vectorized loop, leading to better utilization at any vector length without additional register pressure.

3.5 Summary

This chapter presented ALC, a vector permutation that re-orders active lanes between two vector registers with the goal of creating a uniform vector. Consolidating active lanes into a uniform vector enables BOSCC branches to succeed during runtime to bypass unnecessary code. However, because inactive lanes may be needed through alternate control flow paths, they must be retained through permutation. In addition, variables defined outside a conditional block may be used inside the conditional block. Therefore, a mechanism — the index-based inter-register permutation presented in this chapter — is required to re-organize lanes to match the consolidated order. We implemented the ALC permutation using the ARM scalable vector extension, a modern vector ISA and showed that the proposed permutation is similar to the `compact` instruction already defined in SVE. Future applications of ALC may warrant the inclusion of a `consolidate` instruction that captures the semantics of the ALC permutation. The versatility of the ALC permutation is illustrated through the proposal of two loop transformations, unrolling ALC and iterative ALC, that increase vector utilization in the presence of divergent control flow. The next chapter presents four case studies that estimate the efficacy and performance impact of the ALC loop transformations.

Chapter 4

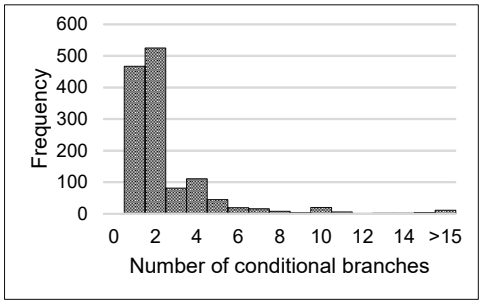
Evaluation

This benchmark-based study assesses the opportunities to apply, and potential benefit of, ALC and iterative ALC in future programs. Using the SPEC CPU 2017 benchmark suite, this study tries to answer the following questions:

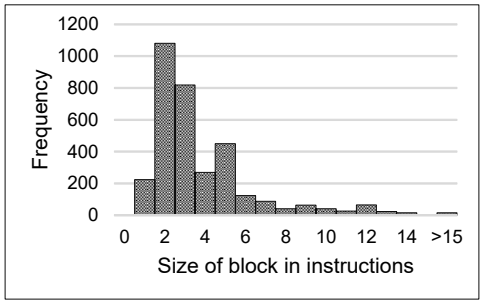
Q1: How many opportunities exist for applying ALC and iterative ALC in the SPEC CPU 2017 benchmark suite?

Q2: Can the ALC permutation, through loop transformations, be used to non-trivially increase vector utilization in loops that experience divergent control flow?

Q3: What is the overhead of performing the ALC permutation and index-based inter-register permutation?



(a) Number of conditional branches in each loop.



(b) Number of instructions in each block of the loop.

Figure 4.1: Results of a static analysis on the loops matching the basic criteria for ALC to be applicable in the SPEC CPU 2017 benchmark suite.

	Total	**Applicable**
500.perlbench	4667	11
502.gcc	17658	115
505.mcf	85	2
520.omnetpp	2170	1
523.xalan	10886	24
525.x264	1127	5
531.deepsjeng	148	8
541.leela	189	8
557.xz	247	0
508.namd	1449	0
510.parest	5486	35
511.povray	2450	35
519.lbm	2450	28
526.blender	19	4
538.imagick	36337	928
544.nab	2034	142
Sum	85230	1322

Table 4.1: Loops where ALC may be applicable discovered by the static analysis described in Section 4.1.

4.1 Methodology

In order for a loop to contain an opportunity to apply ALC it must:

1. Not have any loop-carried dependencies;
2. Contain a block that is terminated by a conditional branch;
3. Not contain function calls.

To answer **Q1** we designed a static analysis that detects loops with these properties. Table 4.1 shows the total number of loops found and the number of loops that meet the ALC conditions in each benchmark of the SPEC CPU 2017 benchmark suite compiled with vectorization, loop unrolling disabled and function inlining enabled. The same static analysis answers two additional questions of interest: how many conditional branches each loop contains; and how large are the executed blocks. Figure 4.1a shows a histogram of the number of conditional branches in ALC opportunity loops. Figure 4.1b shows a histogram of the number of LLVM IR operations in blocks inside the loop.

The counts used to construct the histograms in Figure 4.1 exclude the latch, exit, and header blocks whose predecessors are terminated by a conditional branch. This analysis reveals that a large portion of the loops to which ALC can be applied contain few conditional branches and the conditional blocks contain a small number of operations.

For a profitable application of ALC, a loop also must exhibit divergent control flow during its execution. Therefore the dynamic behavior of a loop also has to be examined. The remainder of this Section investigates four loops reported by the static analysis whose runtime branching behavior is divergent. These loops illustrate how ALC and iterative ALC affect the vector execution and performance. Divergency was confirmed by running the benchmark and inspecting the branch directions at runtime. The criteria for divergency was simply a non-uniform list of branch outcomes. When implemented inside a production compiler, the application of ALC must also check for other factors, such as non-vectorizable statements, that may prevent the application of these transformations.

As of writing, the only SVE-enabled processor available is the Fujitsu A64FX [16] that we are unable to access. Thus, this performance estimation uses the Arm Instruction Emulator (ARMIE) [5] running on the non-SVE-enabled AArch64 hardware. ARMIE uses DynamoRIO [8] to perform dynamic binary translation. ARMIE replaces SVE instructions with an equivalent sequence of scalar AArch64 instructions. Because the SVE instructions are being emulated, wall-clock runtime cannot be used as a result.

The goal of ALC is to increase uniformity in vectors to improve the efficacy of traditional compiler optimizations such as BOSCCs. BOSCCs elide the execution of redundant instructions by exploiting uniform vectors and thus reducing the dynamic instruction count. ARMIE collects these key measurements and enables an assessment of the efficacy of ALC. Dynamic runtime statistics collected through ARMIE include dynamic instruction count, control-flow-path frequency, and vector-instruction utilization. These statistics are used to compare ALC to the previous state of the art. Each version of the loop is compiled using Clang 10 with the `-fno-vectorize -fno-slp-vectorize`

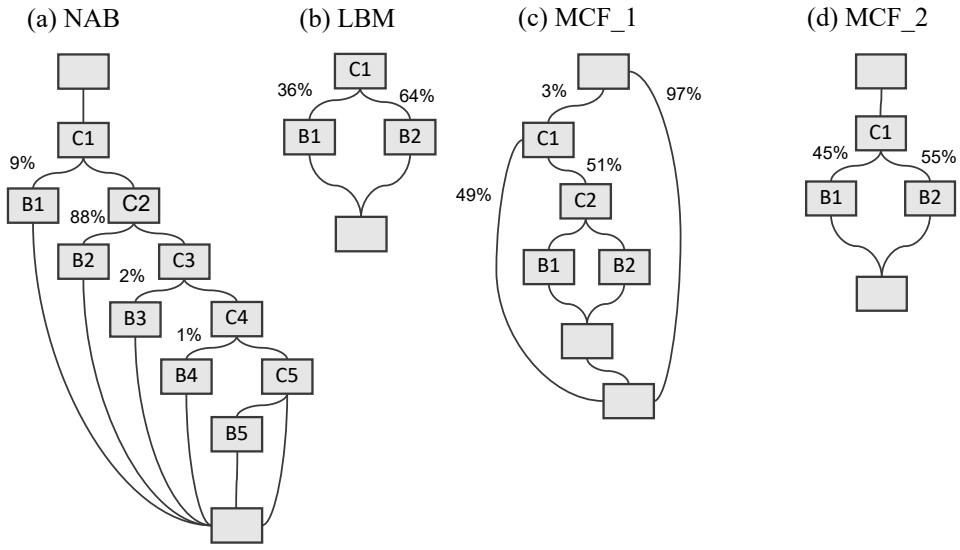


Figure 4.2: Control flow graphs of the body of the loops of the benchmarks. Percentages attached to edges indicate the runtime probability the branch is taken in the TRAIN workload.

`-fno-unroll-loops` compiler flags. Emulated executions take much longer to complete than native executions, therefore, the `TRAIN` workload of the SPEC CPU 2017 benchmark suite is used for all ARMIE-based measurements. A comparison of the frequency of execution of the branch instructions and of the branch outcomes with executions of the `REFRATE` workload revealed that they are very similar to the `TRAIN` workload.

4.1.1 Case Studies

The four test cases used in this study are loops taken from the following C/C++ SPEC CPU2017 benchmarks: NAB, LBM, and MCF. Open-source production compilers' implementations of SVE are in very early development. Until the requisite infrastructure necessary to write an automatic compiler pass — which requires effort from a team of developers — is available, the potential for applying ALC can be assessed by rewriting each loop by hand using the ARM C language extensions (ACLE) [4]. This laborious process limits the number of case studies reported.

LBM LBM is a fluid-dynamic benchmark that implements the Lattice-Boltzmann method to simulate incompressible fluids in 3D [33]. The case study

from LBM is extracted from the function `StreamCollideTRT`, which takes 97% of the runtime when LBM is run on the `TRAIN` workload. Figure 4.2 shows the two main paths of control in this loop, B_1 performs 19 loads and stores while B_2 performs 49 loads and 19 stores. The block B_2 also performs many multiply, add and subtract arithmetic operations. The step increment of the loop induction variable is not unitary. Therefore, after vectorization, load and store operations in B_1 and B_2 are translated into gather loads and scatter stores. The predicate computation for the condition C_1 in the vectorized version is simple because it depends only on a load. Also of importance, the statements inside the conditional blocks do not depend on values computed outside and therefore the ALC loop transformation does not need to permute operands.

NAB The NAB (Nucleic acid builder) is a molecular-dynamic program in the SPEC CPU2017 benchmark suite [33]. The case study from NAB is from the `egb` function and contains parallel hot loops with complex control flow. The control-flow excerpt for the first of the three loops is shown in Figure 4.2(a) and was also used in the study that evaluated the effect of simple BOSCCs on this loop [23].

MCF_1 The first case study from the MCF benchmark, `MCF_1`, is from the `flow_cost` function. The MCF benchmark solves a network-flow problem to compute schedules for public transportation [33]. The loop in this case study takes around 1% of the execution time of the benchmark.

MCF_2 The second case study from the MCF benchmark, `MCF_2`, is found in the `read_min` function. This loop calls the function `getArcPosition`, which contains an `if-else` statement and is inlined into the loop during compilation.

4.1.2 Loop Transformations

For these case studies, each loop is vectorized using ACLE intrinsics to insert BOSCC instructions and apply the unrolling ALC and iterative ALC transformations. The different versions of the loop implemented and evaluated are

as follows:

ifcvt: In this baseline the loop is vectorized with IF-conversion and no BOSCC instructions are inserted. This version is similar to how the LLVM and GCC vectorizers currently transform these loops.

boscc: After vectorization and IF-conversion, additional BOSCC instructions are inserted, where appropriate, to optimize execution.

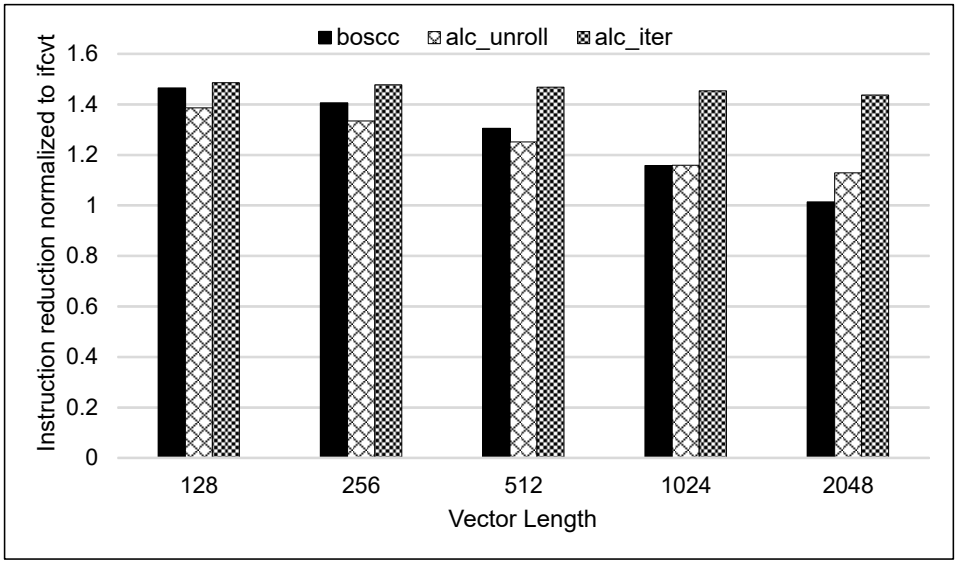
alc_unroll: The unrolling ALC transformation described in Chapter 3.1 is applied to the vectorized and IF-converted loop to consolidate active lanes of a conditional block.

alc_iter: In applicable benchmarks, the iterative ALC transformation described in Chapter 3.4 is applied to the loop. Iterative ALC was only applied to the LBM and MCF_1 loops because the long `if-else-if` statement in the NAB kernel and the small conditional blocks in MCF_2 prevented application on these kernels.

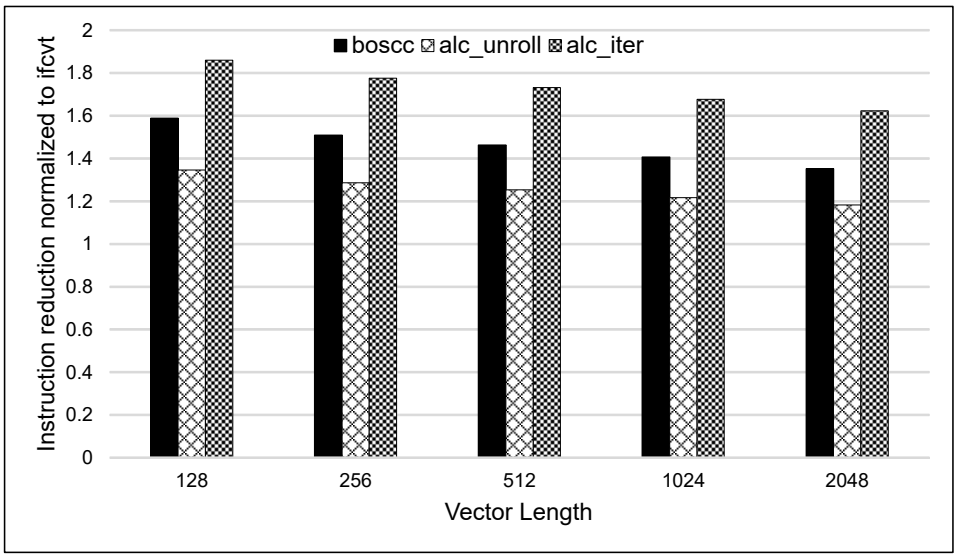
For the comparison with **boscc** one must establish the location in the execution path where BOSCC instructions should be placed and the type of instruction — `all`, `any` or `none` — to be placed. Both factors can greatly affect performance. At the time, there is no automated compiler analysis to determine a suitable placement of BOSCC instructions into a vectorized and if-converted control flow graph. Thus, in order to present a fair comparison with the ALC approaches for this study, we empirically vary the placement and type of BOSCC and use the best-performing kernel as a baseline for the comparison. For instance, the NAB loop contains an `if-else-if` statement that creates five conditional blocks in the IR. As shown in Figure 4.2a, block B_2 is by far the most frequently executed conditional block in the NAB kernel. Thus, the insertion of an *all* BOSCC on the condition computed in C_2 that predicates B_2 leads to the execution of B_2 only if all lanes are active and results in good performance. However, another performant solution is to insert an *any* BOSCC *after* B_2, before the execution of B_3 to elide execution of the remaining linearized blocks in the case where all lanes were active for either B_2 or B_1. In this study, we run experiments for both and report the best-performing kernel. The locations in the control flow graphs where BOSCC instructions

are inserted for each case study are shown as dashed edges in Figure 4.3(d).

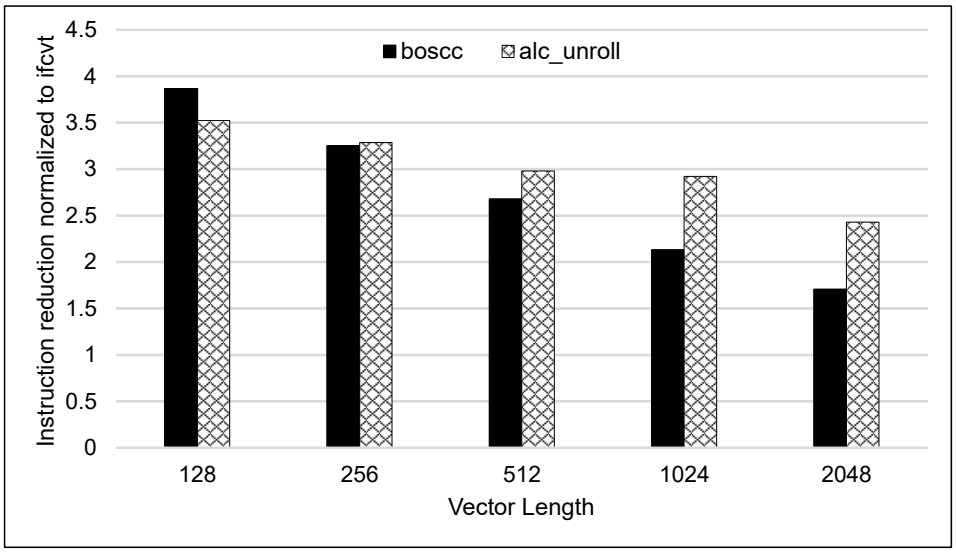
For **alc_unroll**, future compiler analysis will be needed to decide which conditional block should be consolidated. In most cases, consolidating on the most frequently executed block is a good heuristic. However there are cases where consolidating a less frequently taken block, such as B_1 in the LBM benchmark, is more beneficial. In this study, a similar empirical approach to the one used for **boscc** decides which block to consolidate for **alc_unroll**.



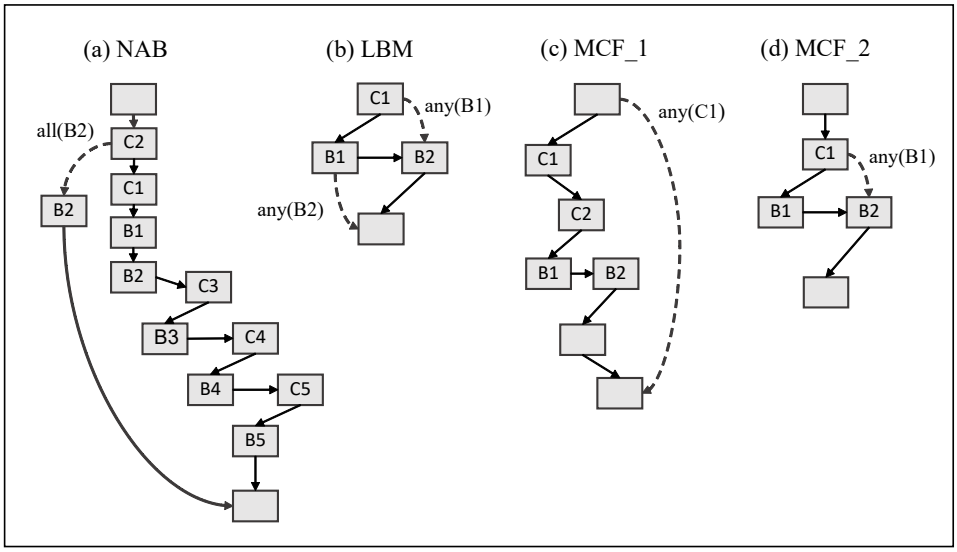
(a) LBM



(b) MCF_1

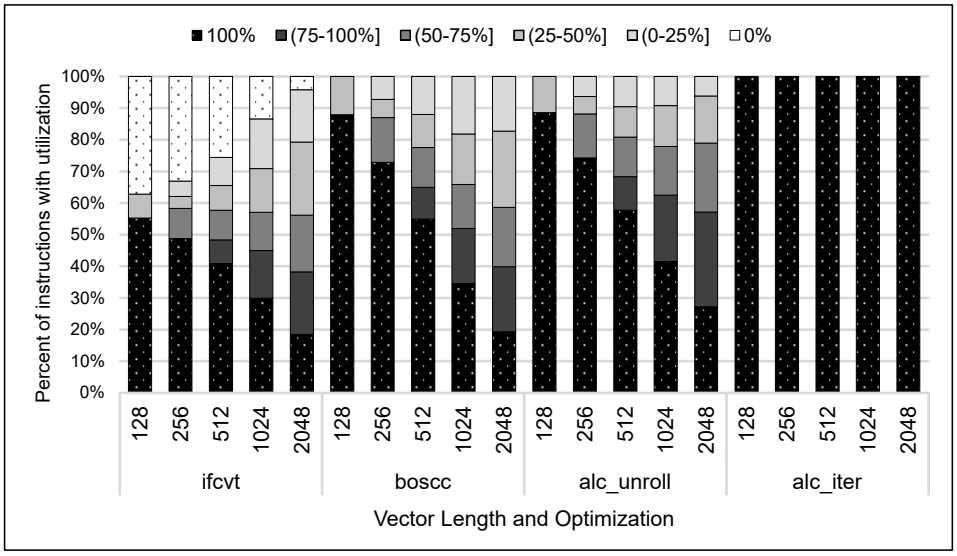


(c) NAB

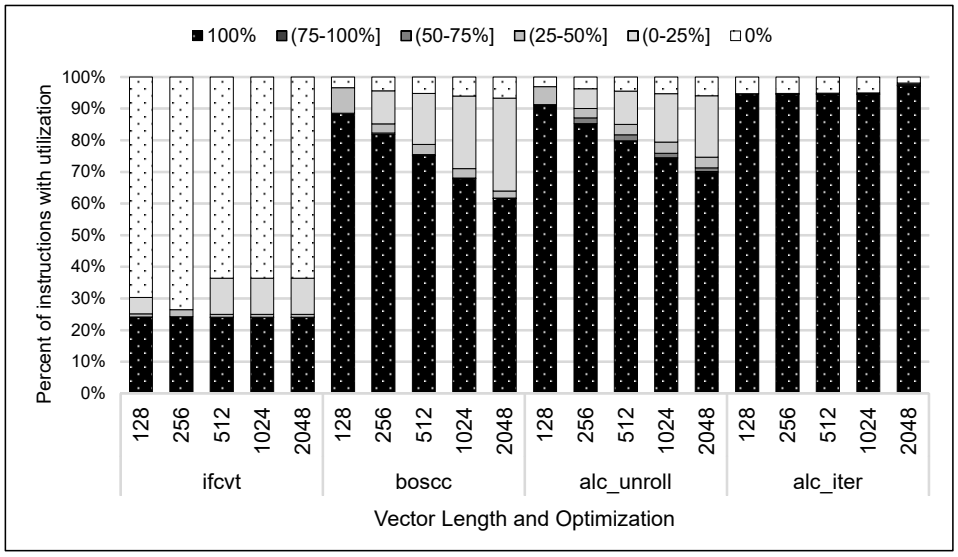


(d) If-converted CFGs with BOSCCs

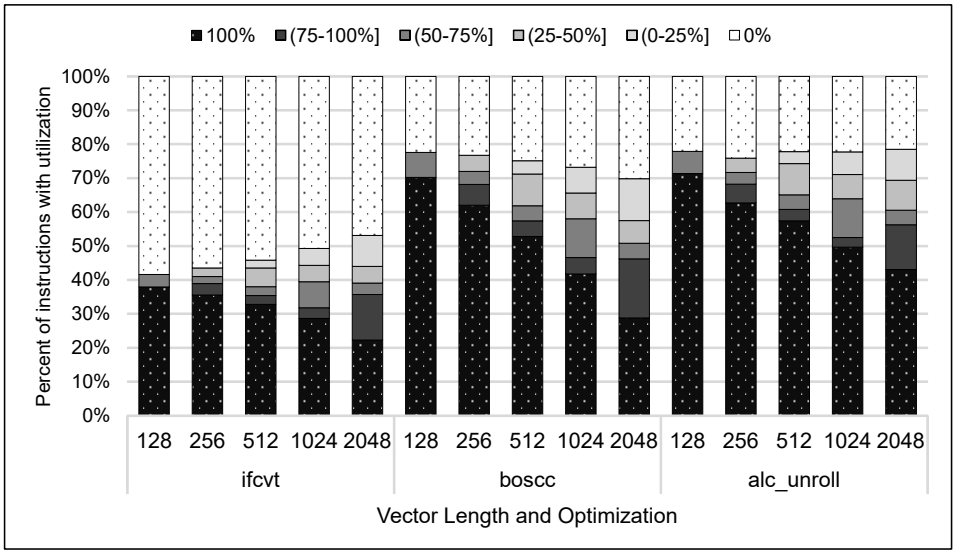
Figure 4.3: Reduction in dynamic instruction count of loops optimized with BOSSCs (boscc) or the ALC transformations (alc_unroll, alc_iter) over the kernel with only vectorization and `if`-conversion (ifcvt) applied. Figure 4.3d shows the control-flow graphs of the kernels with the locations of the BOSSC branches that are indicated by a dashed line.


100%
(75-100%]
(50-75%]
(25-50%]
(0-25%]
0%
Percent of instructions with utilization
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
128
256
512
1024
2048
ifcvt
boscc
alc_unroll
alc_iter
Vector Length and Optimization

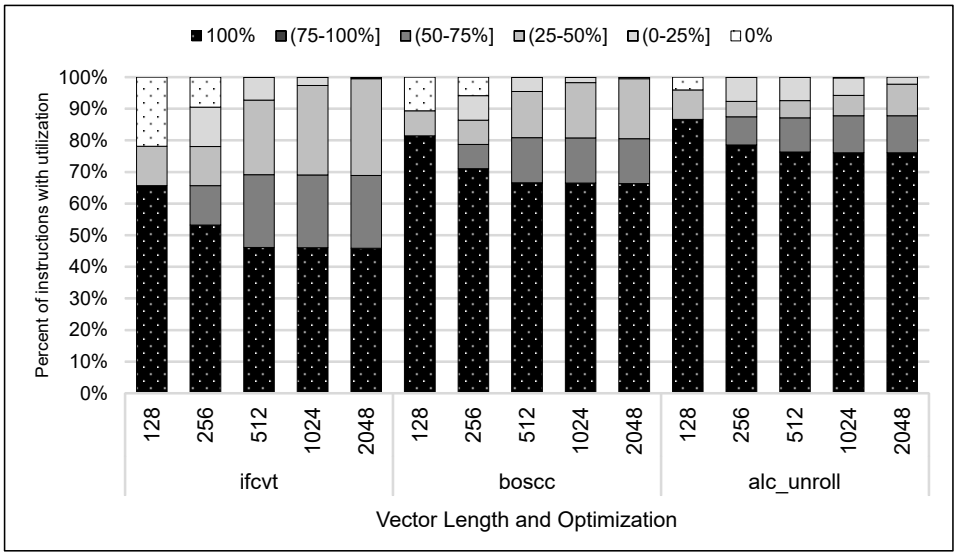
(a) LBM


100%
(75-100%]
(50-75%]
(25-50%]
(0-25%]
0%
Percent of instructions with utilization
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
128
256
512
1024
2048
ifcvt
boscc
alc_unroll
alc_iter
Vector Length and Optimization

(b) MCF_1



(c) NAB



(d) MCF_2

Figure 4.4: Percent divergence of vector instructions. Dotted black bars indicate 100% utilization while dotted white bars indicate 0% utilization. Shaded bars indicate partial utilization with darker shades representing higher utilization.

4.2 Results

Figure 4.3 reports the dynamic instruction-count reduction in the region-of-interest (ROI) for each benchmark for the three code transformations. The baseline is *ifcvt*. In comparison to the *boscc* kernels, the instruction reduction in *alc_unroll* is greater for longer vectors in the LBM and NAB case studies (12% and 39% at 2048 bits). For the MCF_2 case study, no results are reported because all three code transformations failed to reduce dynamic instruction count over baseline.

In the **alc_unroll** kernels, the control flow path which consolidates active lanes of vectors is followed only when there are enough active lanes to create a uniform vector and none of the input vectors are already uniform before the ALC permutation. To handle situations where uniform vectors are encountered before ALC, BOSCCs are inserted in the fall-back path that executes the regular if-converted code. BOSCCs inserted in the fall-back path can account for a large portion of the instruction reduction in the unrolling ALC versions. In fact, at lower vector lengths, such as 128-bits and 256-bits, most of the instruction reduction reported in Figure 4.3 is due to these BOSCCs.

A downward trend emerges in the graphs as VL increases because a larger portion of the vectors in the loop become divergent causing BOSCCs to lose effectiveness. In the **alc_unroll** kernels, the effect is lessened because the divergent vectors are consolidated into a uniform vector, leading ALC to outperform the **boscc** kernels in the LBM and NAB benchmarks after a certain vector length (1024 bits in LBM and 512 bits in NAB).

There is a clear correlation between lower dynamic instruction count and the loop kernels that execute more efficiently, i.e. higher vector utilization reported in Figure 4.4. In the LBM loop, the unrolling ALC transformation only begins to yield instruction reduction (Figure 4.3a) over the *boscc* kernel once there is a clear gap between utilization, as seen in vectors greater than 1024 bits on **alc_unroll**, compared with the the **boscc** kernel in Figure 4.4a. This trend is also observed in the NAB loop between **boscc** and **alc_unroll**.

alc_unroll finds many opportunities to consolidate vector lanes in MCF_1

(21% of iterations consolidate vectors at 2048 bits). However, it fails to out-perform **boscc** regardless of vector length because bypassing the vectorized code for the small basic blocks does not amortize the overhead introduced by performing the ALC permutation.

4.2.1 Iterative ALC

When **alc_unroll** encounters a vector that is uniform prior to the ALC permutation, the fall-back path is taken because consolidation does not benefit this case. This fall-back path contains IF-converted code with BOSCCs applied to exploit the existing uniform vector as done for **boscc**. When one of the two vectors exposed by unrolling is uniform and the other is divergent, the divergent vector must be processed by the inefficient if-converted code similar as is done for **ifcvt**. When both candidate vectors are divergent but do not contain enough active lanes to consolidate into a uniform vector, both vectors are processed by the inefficient if-converted code in the fall-back path. **alc_iter** addresses these inefficiencies by consolidating active lanes from several iterations until a uniform vector is formed instead of being limited to consolidating lanes only from the vectors exposed by unrolling.

Iterative ALC out-performs both **boscc** and **alc_unroll** kernels in the two loops where it was applied: LBM and MCF_1. We only apply iterative ALC to these two loops as the simple control flow structures present in the kernels make the implementation of iterative ALC straightforward. Iterative ALC was not applied on the NAB kernel as the nature of iterative ALC requires that all operands used within every conditional block be saved for later permutation. The five-long `if-else-if` cascade present in the NAB loop leads to a prohibitive state-saving space requirement. Furthermore, iterative ALC is less applicable to loops with long `if-else-if` cascades because an automatic compiler pass to perform the transformation would require a well-informed decision about which conditional block to consolidate compared to loops with simple single-`if` constructs.

Overall, fewer instructions are executed by **alc_iter** because vectors are utilized more efficiently. Where the **boscc** and **alc_unroll** would sub-optimally

process a vector with a partially active predicate, **alc_iter** trades inactive lanes in this vector with active lanes encountered in other iterations of the loop.

The MCF_1 case study contains a nested if-statement that sparsely takes the branch to block C_1 shown in Figure 4.2. BOSCC instructions are not effective for longer vectors in this case because the long vectors will often contain a few active lanes and cause the any-true BOSCC to succeed so that the branch to C_1 is taken. This gradual degradation of **boscc** due to it executing instructions with "near-empty" predicates as vector length increases is observed in Figure 4.4b. The sparsity of C_1 — taken 5% of the time — leads alc_unroll to struggle to find opportunities to merge vectors from consecutive iterations. In comparison, iterative ALC steals active lanes from several iterations to create a uniform merged vector with active lanes. The consolidation of active lanes into the merge vector leaves a uniform remainder vector with only false lanes whose execution can be completely bypassed because the control-flow structure of the kernel only contains a single `if` statement with no `else`. The result is that **alc_iter** outperforms both **boscc** and **alc_unroll** in both vector utilization (Figure 4.4b) and dynamic instruction reduction (Figure 4.3b).

4.2.2 Under-Utilization of Vector Instructions

Figure 4.4 shows the percentage of vector instructions that are executed with a certain predicate utilization. Darker shades indicate more lanes of the predicate are active while lighter shades indicate less. The **ifcvt** kernel shows the results from the version of the loop vectorized with naive IF-conversion and is similar to how LLVM and GCC would currently vectorize these loops. Figure 4.4 shows that a very large portion of vector instructions execute with a predicate that is either all-false or has low utilization. This illustrates the poor execution efficacy that traditional IF-conversion leads to especially when control divergence becomes more prevalent as the vector length increases.

The results in Figure 4.4 indicate that when properly placed, BOSCC instructions can effectively avoid executing instructions with all-false predicates. For instance, an `all` BOSCC inserted on B_2 in NAB and an `any` BOSCC inserted on B_1 in LBM result in the largest instruction reduction because B_2 is

the hottest block in NAB and B_1 is the least frequent block in LBM. However, a disadvantage of both **boscc** and **alc_unroll** is that they both depend on a yet-to-be-created compiler analysis. Such analysis has to determine the place and type of BOSCC instructions to insert for **boscc** and which block to consolidate for **alc_unroll**. These decisions depend on accurately predicting the branch probabilities at runtime. For some programs, such probabilities depend on the workload and thus vary from run to run.

In comparison, **alc_iter** fills the merged vector as it encounters active lanes in the loop and therefore it does not depend on accurate branch-outcome information. For instance, the block being consolidated in Figure 4.4a is the lesser executed block B_1. Even so, **alc_iter** achieves perfect utilization and significant instruction reduction over the best results for **boscc** and **alc_unroll**.

4.2.3 Overhead of the ALC Permutation

Kernel	**ifcvt**	**boscc**		**alc_unroll**		**alc_iter**	
		not-taken	taken	not-taken	taken	filled	continue
lbm	583	589	74	1212	114	96	546
nab	204	205	98	470	331		
mcf_1	25	32	15	74	66	48	21
mcf_2	45	46	40	90	130		

Table 4.2: Static instruction counts for the control flow paths in each kernel.

This section addresses question **Q3**. As discussed in Chapter 3.2 the ALC permutation requires at most fifteen instructions while the inter-register permutation requires three instructions per operand that needs permuting. To be profitable, the ALC loop transformations must amortize the cost of executing instructions to perform these permutations.

Table 4.2 shows the instruction count for the primary control flow paths in the loop kernels. As illustrated in Figure 4.3d BOSCC branches introduce control flow for **boscc**, **alc_unroll** and **alc_iter** to allow bypassing sections when all the lanes of a vector are inactive. If any of the lanes are active, the BOSCC branch is not taken and the entire sequence of partially-active predicated vector instructions must be executed. The number of instructions

for each of these paths are reported under the columns *boscc not-taken* and *boscc taken* in Table 4.2.

The slight increase in the number of instructions from *ifcvt* to *boscc not taken* is indicative of the low overhead introduced by the BOSCC instruction. The drastic instruction-count reduction from *ifcvt* to *boscc taken* for the LBM, MCF_1 and NAB kernels underscores the inefficiency of applying only IF-conversion to the loop. This inefficiency results from *ifcvt* executing predicated code to account for all paths, including some which may be predicated by an all-false predicate.

In the **alc_unroll** kernels, there is a taken and a not-taken path. In the taken path, ALC is performed and thus, it includes instructions to perform the ALC and index-based inter-register permutations. In the not-taken path, ALC is not performed either because candidate vectors for consolidation are already uniform or because there is an insufficient number of active lanes to form a uniform vector. Note that **alc_unroll** executes two iterations of the loop while *ifcvt* to *boscc* execute a single one — this difference accounts for static instruction counts for **alc_unroll** being close to twice the amount reported in the **ifcvt** column.

For **alc_unroll** in MCF_2 the taken path is longer than the not-taken path and thus consolidation will result in performance degradation because several operands must be permuted. The issue is compounded by the small size of the basic blocks, which lowers the benefit of a BOSCC branch. Larger blocks would result in more savings from bypassing execution and would amortize the operand permutation costs. For kernels exhibiting similar characteristics where many operands need to be permuted to correctly execute a small consolidated block, it is not beneficial to consolidate execution of vectors through the unrolling ALC transformations.

With the proposed ISA design changes described in Chapter 3.3 both the ALC permutation and the index-based inter-register permutation would require a single instruction each. Such change would significantly reduce the overhead of ALC. For instance, for MCF_2, the overhead could be reduced to seven instructions: one for ALC and six for index-based inter-register permutation

of operands for the consolidated block and for the remainder block. Significant overhead reduction, such as this one for MCF_2, will enable the use of ALC for loops with small block sizes. This is especially interesting given the findings presented in Figure 4.1 that show that small conditional blocks, containing only two to eight instructions are most common.

The **alc_iter** kernels contain two paths, *filled* and *continue*. The *filled* path processes the merged vector once it becomes full, this is illustrated in iteration 2 of Figure 3.13. The *continue* path represents the path that control follows when the ALC permutation is performed but the merged vector does not become full. In the case of a single `if` statement with no `else` block, such as in the MCF_1 benchmark, no additional code is executed. Furthermore, because there is no need to retain the inactive lanes, a simplified version of the ALC permutation, that requires fewer instructions, can be used. These characteristics make iterative ALC very appealing to optimize loops with lone `if` statements. In contrast, the LBM benchmark contains an `if-else` statement so that the *continue* path executes code to process the lanes that were inactive for the `if` block. In, MCF_1, 27 of the 48 instructions present in the *filled* path are related to the actual execution of the conditional block. The remaining 21 instructions execute the ALC permutation to consolidate active lanes. The difference between the 15 required instructions presented in Chapter 3.4 and the 21 listed here are due to imperfections in the compiler's code generation. Even with the large number of instructions required to perform the ALC permutation, the iterative ALC kernel outperforms the best *boscc* kernel because of the large increase in vector utilization provided by iterative ALC.

4.3 Summary

This chapter presented case studies of the application of ALC to four different loops found in the SPEC CPU2017 benchmark. These early results obtained through emulation of the vector ISA on a loop optimized by the ALC transformations indicate that the proposed permutation has significant potential

for improving the performance of vectorized code by increasing the number of lanes utilized by each instruction. Iterative ALC, in one setting in the LBM benchmark, demonstrated a 30.9% reduction in dynamic instruction count over the baseline IF-converted kernel optimized solely with BOSCCs. Such a result gives evidence to the sporadic branch behavior present in real code and workloads and indicates that there is vast room for improvement in vector code generation. The characterization of the vector utilization in each benchmark highlights the inefficiencies of unoptimized IF-converted code and the improvements that can be achieved using one of the ALC loop transformations. Furthermore, this case-based experimental study illustrates the breakdown of BOSCC instructions as vector lengths are extended and how ALC can be used to improve vector performance in these cases. This chapter showed that, in certain cases, complex control flow graphs may prevent the application of iterative ALC. In these cases, unrolling ALC is still applicable and shown to lead to a non-trivial instruction reduction over the baseline.

Chapter 5

Related Work

Vectorization is a broad subject and has many active areas of interest that are involved in this work. This chapter presents related work organized by topic.

5.1 Control-Flow Vectorization

Control flow is an integral programming construct that has presented an obstacle for SIMD execution for decades since early supercomputers like the CRAY-1 [27]. Today, supporting SIMD execution in the presence of control flow exists an important problem more than ever with the exploitation of DLP remaining as one of the most prominent directions of research to facilitate performance improvements.

The traditional IF-conversion algorithm [1] presents a method to convert control dependencies into data dependencies to enable compiler automatic vectorization to take place. IF-conversion removes the complexities of accounting for data dependence and control-dependence in compiler analysis, simplifying the vectorization transformation. Predicate information can easily be inferred from the data dependencies converted from control-dependencies in the IF-converted code and directly map to execution conditions needed in predicated/masked ISAs. ALC is supplemental to IF-conversion as it attempts to address some of the inefficiencies in execution caused by linearized code.

Predicated or masked vector instructions are only one method to support the vectorization of control flow. The various other methods such as using register compress/expands or memory gather/scatter instructions have been studied

extensively in comparison to masked instructions with the latter proving itself as the most beneficial method, especially in density-time architectures [32]. In addition, most modern long vector ISAs like Intel's AVX-512 or ARM's SVE support control flow vectorization through instruction masking and predication. Other non-masking methods are still an important topic as some popular vector ISAs like ARM's NEON or Intel's AVX do not provide predication. Recent research is still uncovering novel techniques, like IF-selection [37] to efficiently vectorize control flow in these architectures [26]. One emerging vector ISA of interest, the RISC-V "V" (vector) extension provides both masked instructions and a prefix-sum instruction (`IOTA`) that can be used for register compression operations [22]. In this work, we consider predicated vector ISAs and present a permutation able to consolidate the active from two under-utilized divergent vectors into one uniform vector that can execute with 100% utilization.

More modern IF-conversion algorithms have been proposed that try to address shortcomings in the original method. Partial control flow linearization [23] presents one such algorithm that takes into account the presence of uniform branches that will not cause vector divergence. Uniform branches are branches whose condition does not change throughout execution and can be discovered through divergence analysis [13]. Rather than linearizing the control-flow graph that includes such uniform branches as traditional IF-conversion would, the algorithm transforms the graph such that the branch is retained even in the final vectorized code. This elides execution of a large portion of vectorized code that would otherwise execute with no active lanes and considerably decrease vector utilization. In contrast, ALC addresses non-uniform branches that may cause divergence in vectors at runtime. These branches are particularly troublesome as compiler analysis is currently unable to discover much useful information them and thus, applying optimizations presents the risk of degrading performance unless a runtime profile on a specific workload is available for guidance. In this work, we present a loop transformation, unrolling ALC, that incorporates traditional IF-conversion as a way to retreat from committing an otherwise unprofitable optimization at runtime.

Warp-coherent-condition vectorization (WCCV) attempts to address the compiler inadequacies in analyzing non-uniform branch conditions [36]. In particular, WCCV presents two types of conditions and proposes techniques for their detection: high-probability conditions and Boolean-step conditions. Boolean-step conditions are non-uniform conditions whose outcome only changes once during execution as the thread ID or loop induction variable crosses a boundary, for example, the condition `i > 256` in a loop with a trip count of `512`. WCCV uses Affine-Analysis to analyze and prove the condition is Boolean. High-probability conditions are branch conditions that are statically provable to be biased at runtime. WCCV proposes detection of these conditions through auto-tuning the parameters given by LLVMs branch-probability estimation and branch cost also given by LLVMs costing framework. Even though the conditions are still considered non-uniform, by examining the equation associated with the condition or by taking into account the branches' parameters, it becomes clear the behavior of the branch will result in mostly uniform vectors that will not be optimized by partial control flow linearization.

In comparison, ALC attempts to provide a method to optimize conditions not meeting either the statically-uniform or runtime-uniform requirements focused on by partial control flow linearization and WCCV. Instead, ALC provides a solution to optimize runtime-divergent conditions by consolidating the active lanes resulting from these types of conditions.

The trend in increasing vector-length brings into question the possibility of their continued growth. Vector length has a profound impact on the performance of SIMD and vector code with control divergence being one factor. The impact of vector-length on SIMD performance has been studied in [29]. In this study, 76 benchmarks from various application domains were studied the conclusions show that a large majority of applications do not suffer from a heavy amount of control divergence and thus are likely to scale well with increasing vector length. However, the study also showed that a large portion of benchmarks suffered from a heavy amount of control divergence. In addition, one notable conclusion of the study was that out of the examined kernels, the vast majority had only four or less distinct control flow paths. In this work,

four kernels are studied from a popular benchmark suite, SPEC 2017, that suffer from control divergence and show the potential for ALC to alleviate SIMD performance degradation.

5.2 Dynamic Uniformity Conditions

Shin et al. first propose a compiler analysis and code generation technique to accelerate vectorized multimedia applications using the Altivec G4 BOSCC instructions [30]. They later improve on their approach by nesting BOSCCs to allow BOSCC instructions to bypass other BOSCCs improving their performance margins. However, they do not address the problem of automatic insertion as the compiler analysis used to estimate the profitability of inserting a BOSCC relies on a separate profiling phase to gather metrics concerning branch behavior.

Warp-coherent-condition vectorization [36], in addition to the contributions made to formalize divergent-conditions, also propose a a simple control flow transformation where an *all* BOSCC is inserted to guard the target block of the boolean-step or high-probability condition. This translates to an efficient execution where only the most frequently visited block is executed.

In comparison, Partial Control Flow Linearization, in addition to the contributions made to improve IF-conversion, presents a use case of the partial-linearization algorithm to showcase support for BOSCC constructs. They introduce a control flow construct called a BOSCC-gadget that refers to the semantics of a certain BOSCC instruction such as an *any* BOSCC. The partial control-flow linearization algorithm retains these BOSCC gadgets as the algorithm detects the branch as uniform and is later lowered to the respective BOSCC instruction supported in the ISA. In their experiments, the control flow transformation applied inserted *any* BOSCCs in such a manner so that the least frequently visited blocks were bypassed. This is in contrast to the transformation in WCCV where an *all* BOSCC was inserted on the most frequently visited block.

Works related to BOSCC instructions are foundational for ALC and the

proposed ALC loop transformations. In our experiments, the unrolling ALC transformation uses control flow transformations similar to both those found in Partial-Control-Flow-Linearization and WCCV. However, both works attempt to optimize only naturally occurring uniform instructions with BOSCCs, in contrast, ALC is able to dynamically re-organize the vector lanes such that a uniform vector can be formed out of multiple divergent vectors so that more opportunities for BOSCCs to bypass code are exposed. In this aspect, the goal of ALC is slightly different and complementary to Partial-Control-Flow-Linearization and WCCV.

5.3 Vector Lane Re-organization

Section 3.1.1 introduces the ALC permutation that dynamically re-organizes the active lanes between two vector registers. The concept of permuting lanes between registers with the goal of increasing coherency is not new and has been proposed in the context of vectorized database queries [19]. Lang et al. present a register-to-register and memory-to-register algorithm for refilling partially utilized vectors in AVX-512. The register-to-register algorithm makes extensive use of `compress` and `expand` instructions in AVX-512 while the memory-to-register algorithm uses compress loads and expand stores. In contrast to this work, the two ALC transformations operate on loops with arbitrary control flow rather than code to process database queries. Consequently, the refilling algorithms do not involve the retention of inactive lanes. Because the ALC transformations need to consider the case where inactive lanes are processed by another condition present in the control flow graph of a loop, ALC is written to segregate the inactive lanes into the remainder vector while moving active lanes into the merged vector. Section 3.4 presents a special case where ALC can be altered to not retain inactive lanes, saving on the cost of performing ALC.

Compaction/Restoration [7] proposes a hardware mechanism that detects compactable instructions to merge them into a single dense instruction which is issued to the VPU for execution. The restoration unit is one component of

CR that is responsible for the restoration of lanes compacted into the dense instruction back into their respective registers and correct lane for use in subsequent instructions that may depend on the result. The compaction unit is the component that performs the actual compaction of under-utilized vector instructions into a dense instruction. CR finds candidate vector instructions for compaction as they enter the reorder buffer and creates a new dense instruction when a candidate is found at a PC that is not already considered for compaction. A timeout policy controls the length of time a dense instruction may sit before being issued for execution if it did not already become full during that time. Because dense instructions may delay future vector instructions in out-of-order processors, the timeout policy can greatly affect the performance of CR. A timeout that is too short may pre-maturely issue dense instructions that could have been issued with greater utilization while too long of time may cause stalls due to contention in processor resources. Because of CRs sensitivity to the timeout policy, the performance of CR is impacted in loops with a large number of instructions. Their experimental results show that the performance significantly degrades when moving from a loop with 20 instructions to 60 instructions.

The ALC and index-based inter-register permutations are essentially a software version of the compaction/restoration unit in CR. While the actual vector consolidation/compaction mechanism performs faster in hardware compared to the ALC permutation's required 15 instructions, the sensitivity to loop length may limit the application where the ALC loop transformations are not. In the limited case studies presented in this work, we show two loops that would be completely un-optimizable by CR, LBM, and NAB with 583 and 204 instructions in the IF-converted kernel respectively; and two loops in the MCF benchmark that fall within the instruction limitation imposed by CR. The iterative ALC loop transformation presented in this work can consolidate active lanes from any iteration throughout the trip of the loop and was successfully applied to the LBM benchmark that is 583 instructions long once compiled. Issues concerning the complexity of the ALC permutation could be solved by introducing instructions similar to the one proposed in Section 3.3 that.

One clear benefit of CR over ALC, stemming from its ability to dynamically compact instructions at the ROB level, is that partially utilized vectors that are not expected at compile-time, i.e. instructions from two blocks that are not considered for consolidation by ALC, can be compacted to increase utilization whereas ALC requires information before the final output binary is built.

5.4 GPU Thread Re-organization

The concepts behind the techniques referenced in this work, specifically BOSCC instructions to bypass unnecessary blocks of code and lane re-organization to increase utilization are applicable to any SIMD accelerator that suffers from control flow divergence. The equivalent of lane re-organization in vectors has been studied in the context of graphics-processing units (GPUs) which typically group many parallel threads into a warp that is then executed in a SIMD manner. These works primarily focus on micro-architectural techniques to dynamically compact threads from different warps into a single warp that executes with greater efficiency [14], [15], [38]. Software approaches [18] have limited applicability as they must emulate moving lanes by copying data to shared memory.

5.5 The ARM Scalable Vector Extension

The ARM Scalable vector extension was introduced to address the issues with past vector extensions such as ISA disorganization and scalability [25] and as a solution to drive performance in numerous application domains in the face of demanding power requirements [28], [35]. The popularity of SVE is increasing and mainstream compilers like GCC and LLVM are actively working to implement support [20]. Research so far has focused on using SVE to accelerate applications in specific workloads like stencil codes [6] and image processing pipelines [10] and evaluates the resulting performance. ARM SVE is not the only ISA to re-adopt the VLA architecture with RISC-V [22] also introducing scalable vectors into their ISA for which production hardware has already been produced [9].

Chapter 6

Future Work

ALC is designed to optimize vector execution in the presence of divergent control flow where past techniques like BOSCC, Partial Control Flow Linearization, and WCCV fall short. Each work, including ALC, shares the concept of inserting a BOSCC branch to bypass unnecessary instructions and is well studied. However, the step before insertion, namely the identification of profitable locations to insert BOSCCs, and the cost modelling to prevent unprofitable insertions is underdeveloped in research. For transformations that use BOSCCs to work well in an automatic compiler method, better static techniques that can answer the question of when and where to place the BOSCCs are required. WCCV [36] begins to touch on these ideas as they formulate a way to automatically detect two types of conditions that are known to respond well to BOSCC branches. Similar ideas could be applied for ALC to allow the compiler to automatically perform the ALC transformations presented in this thesis. In another line of thinking, the problem of deciding when and where to apply the above transformations could be left up to the judgment of the developer as they may have insight into the control-flow behavior and have knowledge of the size of blocks in the target loop. In this case, ALC could be applied manually by the programmer by using vector intrinsics. A static analysis to find profitable opportunities could be used to only make the developer aware of a possible opportunity to apply the transformation, leaving the actual application up to the developer.

The ARM Scalable vector extension introduces radical changes in com-

parison to the traditional fixed-length vector extension designs. ARM's SVE expansion of the capabilities of code generation techniques and a clean instruction set made it the primary candidate for the implementation of ALC and for the case studies used to anticipate how ALC may affect performance. However, the ALC permutation and loop transformations presented in this thesis are applicable to any vector architecture provided the ISA is advanced enough to enable the implementation of the ALC permutation. One other interesting ISA, the RISC-V "V"" extension has similar capabilities to that of SVE and has the benefit of being open source. Future work could evaluate the ALC permutations and loop transformations on other vector ISAs to uncover some of the performance tradeoffs of using one ISA over another.

Infrastructure to support emerging vector ISAs like SVE and RISC-V "V"" is currently lacking. Auto-vectorization for these ISAs in mainstream compilers such as LLVM is still under development and will undergo many changes in the near future. The lack of mature infrastructure is a big limitation to experimentation with ALC, and to evaluation of ALC. The evaluation presented in this thesis uses a functional simulator and collects as much information about the experiment as possible. Such a functional simulation does not capture all the factors affecting performance that are present in a production processor. Future work could include a re-evaluation of the techniques proposed in this thesis when more processors implementing either SVE or RISC-V "V"" become available. In addition, the limitation listed above prevented the breadth of evaluation that would be desired. We present only four case studies because of the time required to hand-write the assembly code for each case study and the long simulation time. Future work could make progress by evaluating ALC on a wider set of kernels and workloads to uncover limitations or opportunities that were missed during this research.

One of the loop transformations presented in this thesis, unrolling ALC, harnesses loop unrolling to expose candidate lanes for consolidation. In this work, the unrolling factor is capped at two due to the complexity of implementation. The opportunity to use an unrolling factor greater than two, which would allow consolidation from a larger set of lanes leading to the formation of

a larger number of uniform vectors, is still to be investigated.

The ALC permutation and loop-transformations introduced in this thesis involve re-organizing possibly contiguous lanes of vectors to increase control-flow uniformity. In some cases, contiguous vector loads and stores may need to be translated into gather and scatter memory accesses. Because gather and scatter memory accesses are more expensive, due to the number of micro-operations issued and the potential decrease in memory access locality, ALC may degrade the performance of a vectorized loop. A study to characterize the potential impact of the increased memory-divergence due to ALC would provide a clearer answer to whether this is a problem for ALC. Such a study would be best performed on a production processor to be able to measure wall-clock runtime rather than emulation or simulation. One line of intuition regarding the results of this kind of study says that most uniform vectors formed by ALC will be formed within two-to-three iterations of the vectorized loop and thus, the locations accessed by a gather-load or scatter-store will most commonly access two cachelines. This increase will not prohibitively impact the performance beyond the performance savings gained when using ALC compared to leaving the vector memory accesses contiguous.

Chapter 7

Conclusion

Divergent control flow is a source for inefficiency in vectorized loops. While prior methods have proven useful to battle against inefficiencies there remains plenty of room for improvement. The control flow divergence problem is exacerbated by the industry trend to increase vector length that causes an uptick in the number of divergent predicates encountered. This thesis contributes a significant effort to counteract these effects. Prior methods with a similar goal that utilize BOSCC instructions focused only on the case when vectors are uniform without additional intervention, leaving the case when a divergent vector is encountered unable to be optimized.

Chapter 3 presents a novel vector permutation, Active-Lane Consolidation, that consolidates the active lanes between two divergent vectors to facilitate the formation of a uniform vector. Traditional BOSCC branches are then able to exploit this opportunity by bypassing the execution of unnecessary predicated vector code leading to an increase in vector utilization. The chapter describes an implementation of the ALC permutation on a modern vector ISA, the ARM Scalable Vector Extension, and proposes two loop transformations to illustrate the use of ALC: unrolling ALC and iterative ALC.

Chapter 4 presents case studies of the ALC transformations on a set of four kernels found in a popular benchmark, SPEC 2017, and show that using ALC in a loop transformation has significant potential to increase vector utilization and decrease dynamic instruction count, in one case up to 30.9%. The chapter highlights the current inefficiencies of IF-conversion, the pitfalls of BOSCC

instructions and the effect of ALC as a means to combat these increasingly frequent disturbances.

References

[1] Allen, John R and Kennedy, Ken and Porterfield, Carrie and Warren, Joe, "Conversion of Control Dependence to Data Dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1983, pp. 177–189. 1, 2, 11, 52

[2] F. Allen and J. Cocke, *A catalogue of optimizing transformations*. Prentice-Hall., 1971. 11

[3] J. Anantpur and G. R., "Taming control divergence in gpus through control flow linearization," in *Compiler Construction*, A. Cohen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 133–153, ISBN: 978-3-642-54807-9. 11

[4] ARM, *The Arm C Language Extensions*, https://developer.arm.com/architectures/system-architectures/software-standards/acle, Jan. 2020. 19, 38

[5] ——, *The ARM Instruction Emulator*, https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator, Jan. 2020. 37

[6] A. Armejach, H. Caminal, J. M. Cebrian, R. Langarita, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Using Arm's Scalable Vector Extension on Stencil Codes," *The Journal of Supercomputing*, vol. 76, no. 3, pp. 2039–2062, 2020. 58

[7] A. Barredo, J. M. Cebrian, M. Moretó, M. Casas, and M. Valero, "Improving predication efficiency through compaction/restoration of simd instructions," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 717–728. DOI: 10.1109/HPCA47549.2020.00064. 1, 12, 56

[8] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering ..., 2004. 37

[9] Chen, Chen and Xiang, Xiaoyan and Liu, Chang and Shang, Yunhai and Guo, Ren and Liu, Dongqi and Lu, Yimin and Hao, Ziyi and Luo, Jiahui and Chen, Zhijian and others, "Xuantie-910: A Commercial Multi-core 12-stage Pipeline Out-of-order 64-bit High Performance RISC-V Processor with Vector Extension: Industrial Product," in *2020 ACM/IEEE 47th*

Annual International Symposium on Computer Architecture (ISCA), IEEE, 2020, pp. 52–64. 58

[10] Cococcioni, Marco and Rossi, Federico and Ruffaldi, Emanuele and Saponara, Sergio, "Fast Deep Neural Networks for Image Processing Using Posits and Arm Scalable Vector Extension," *Journal of Real-Time Image Processing*, vol. 17, pp. 759–771, 2020. 58

[11] A. Corporation, *ARM Advanced SIMD*, https://developer.arm.com/architectures/instruction-sets/simd-isas/neon, Jan. 2021. 6

[12] I. Corporation, *Intel AVX-512*, `https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html`, Jan. 2021. 6

[13] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr, "Divergence analysis and optimizations," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2011, pp. 320–329. 53

[14] W. W. Fung and T. M. Aamodt, "Thread block compaction for efficient simt control flow," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, IEEE, 2011, pp. 25–36. 58

[15] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, IEEE, 2007, pp. 407–420. 58

[16] *A64FX® Microarchitecture Manual. Version 1.4*, Fujitsu Limited, Mar. 2021. 37

[17] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2001. 10

[18] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48, Waikiki, Hawaii, 2015, pp. 204–215. DOI: `10.1145/2830772.2830796`. 58

[19] Lang, Harald and Passing, Linnea and Kipf, Andreas and Boncz, Peter and Neumann, Thomas and Kemper, Alfons, "Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines," *The VLDB Journal*, vol. 29, no. 2, pp. 757–774, 2020. 56

[20] Lovett, *SVE in LLVM*, https://hps.vi4io.org/_media/events/2020/llvm-cth20_lovett.pdf, Jun. 2021. 58

[21] A. Ltd., *Arm® Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for Armv8-A*, ARM Ltd., Jan. 2021. 7, 24, 30

[22] R.-V. I. Members, *The RISC-V "V" Vector Extension. Version 0.10 (Visited on April 26, 2021)*, https://github.com/riscv/riscv-v-spec/releases/download/v0.10/riscv-v-spec-0.10.pdf, Jan. 2021. 7, 53, 58

[23] Moll, Simon and Hack, Sebastian, "Partial Control-flow Linearization," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 543–556, 2018. 11, 14, 39, 53

[24] D. Monroe, "Fugaku takes the lead," *Commun. ACM*, vol. 64, no. 1, pp. 16–18, Dec. 2020, ISSN: 0001-0782. DOI: 10.1145/3433954. [Online]. Available: https://doi.org/10.1145/3433954. 1

[25] D. Patterson, *SIMD Instructions Considered Harmful*, https://www.sigarch.org/simd-instructions-considered-harmful, Sep. 2017. 7, 58

[26] A. Pohl, B. Cosenza, and B. Juurlink, "Control flow vectorization for arm neon," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, 2018, pp. 66–75. 53

[27] Russell, Richard M, "The CRAY-1 Computer System," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978. 7, 52

[28] Sato, Mitsuhisa and Ishikawa, Yutaka and Tomita, Hirofumi and Kodama, Yuetsu and Odajima, Tetsuya and Tsuji, Miwako and Yashiro, Hisashi and Aoki, Masaki and Shida, Naoyuki and Miyoshi, Ikuo and others, "Co-Design for A64FX Manycore Processor and "Fugaku"," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2020, pp. 1–15. 58

[29] Schaub, Thomas and Moll, Simon and Karrenberg, Ralf and Hack, Sebastian, "The Impact of the SIMD Width on Control-flow and Memory Divergence," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 1–25, 2015. 54

[30] Shin, Jaewook and Hall, Mary W and Chame, Jacqueline, "Evaluating compiler technology for control-flow optimizations for multimedia extension architectures," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 235–243, 2009. 2, 13, 55

[31] J. Shin, "Introducing control flow into vectorized code," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, IEEE, 2007, pp. 280–291. 2, 13

[32] J. E. Smith, G. Faanes, and R. Sugumar, "Vector instruction set support for conditional operations," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 260–269, 2000. 53

[33] SPEC, *SPEC2017 Benchmark overview.* https://www.spec.org/cpu2017/Docs/overview.html, Jan. 2021. 38, 39

[34] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 363–400, 2000. 8

[35] Stephens, Nigel and Biles, Stuart and Boettcher, Matthias and Eapen, Jacob and Eyole, Mbou and Gabrielli, Giacomo and Horsnell, Matt and Magklis, Grigorios and Martinez, Alejandro and Premillieu, Nathanael and others, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. 58

[36] Sun, Huihui and Fey, Florian and Zhao, Jie and Gorlatch, Sergei, "WCCV: Improving the Vectorization of IF-statements with Warp-coherent Conditions," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 319–329. 14, 54, 55, 59

[37] H. Sun, S. Gorlatch, and R. Zhao, "Refactoring loops with nested ifs for simd extensions without masked instructions," in *European Conference on Parallel Processing*, Springer, 2018, pp. 769–781. 12, 53

[38] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi, "Simd divergence optimization through intra-warp compaction," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 368–379. 58

[39] Wolfe, Michael Joseph, *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995. 10