VeGen: A Vectorizer Generator for SIMD and Beyond

by

Yishen Chen

B.S., University of Illinois (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author .....
Department of Electrical Engineering and Computer Science
August 28, 2021

Certified by .....
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

VeGen: A Vectorizer Generator for SIMD and Beyond

by

Yishen Chen

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Vector instructions are ubiquitous in modern processors. Traditional compiler auto-vectorization techniques have focused on targeting single instruction multiple data (SIMD) instructions. However, these auto-vectorization techniques are not sufficiently powerful to model non-SIMD vector instructions, which can accelerate applications in domains such as image processing, digital signal processing, and machine learning. To target non-SIMD instruction, compiler developers have resorted to complicated, ad hoc peephole optimizations, expending significant development time while still coming up short. As vector instruction sets continue to rapidly evolve, compilers cannot keep up with these new hardware capabilities.

To facilitate the adaption of complex non-SIMD vector instructions, I propose a new model of vector parallelism that captures the semantics of these instructions and a new framework extracting this new model of vector parallelism automatically based on the formal semantics of the non-SIMD instructions.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor Professor Saman Amarasinghe for his guidance, giving me the freedom to work on problems with nothing to show (which I am getting good at), and most importantly, showing more optimism to my research than I could muster. Doing research is hard. Doing research in isolation, during a pandemic is hard hard. This research would not have been possible without Saman's constant encouragement.

I would like to thank Professor Michael Carbin for the early discussions and framing of VeGen and for showing me the ropes to thinking about and managing a research project.

I would like to thank Charith Mendis for the initial discussions that led to VeGen and his feedback during the project.

I would like to thank Ajay Brahmakshatriya for the long walks, discussions, and support.

Finally, I would like to thank my parents for giving me unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Vector instructions are ubiquitous in modern processors. Previous work on auto-vectorization has focused on single instruction multiple data (SIMD) instructions, but there is little research on systematically targeting non-SIMD vector instructions, which has applications in domains such as digital signal processing, image processing, and machine learning (e.g., Intel's VNNI extension and the dot-product instructions in ARMv8 [Holdings, 2011]). In contrast with the SIMD instruction shown in Figure 1-1(a), Figures 1-1(b)–1-1(d) show three examples of the non-SIMD instructions from the AVX2 instruction set. Figure 1-1(b) shows a single instruction, multiple operations, multiple data (SIMOMD) instruction [Bachega et al., 2004] that performs additions and subtractions on alternating lanes (`vaddsubpd`); Figure 1-1(c) shows a horizontal addition with lane interleaving (`vhaddpd`); and Figure 1-1(d) shows an instruction computing dot-products (`vpmaddwd`). To date, there is no unified model of parallelism that captures the capabilities of these instructions.

Automatic Vectorization. There are two mainstream techniques for extracting SIMD parallelism: loop vectorization [Allen and Kennedy, 1987, Nuzman et al., 2006, Nuzman and Zaks, 2008] and superword level parallelism (SLP) based vectorization [Larsen and Amarasinghe, 2000, Liu et al., 2012, Porpodas et al., 2015]. Both techniques make two fundamental assumptions about vector instructions: a SIMD instruction performs isomorphic operations across all lanes, and the instruction ap-

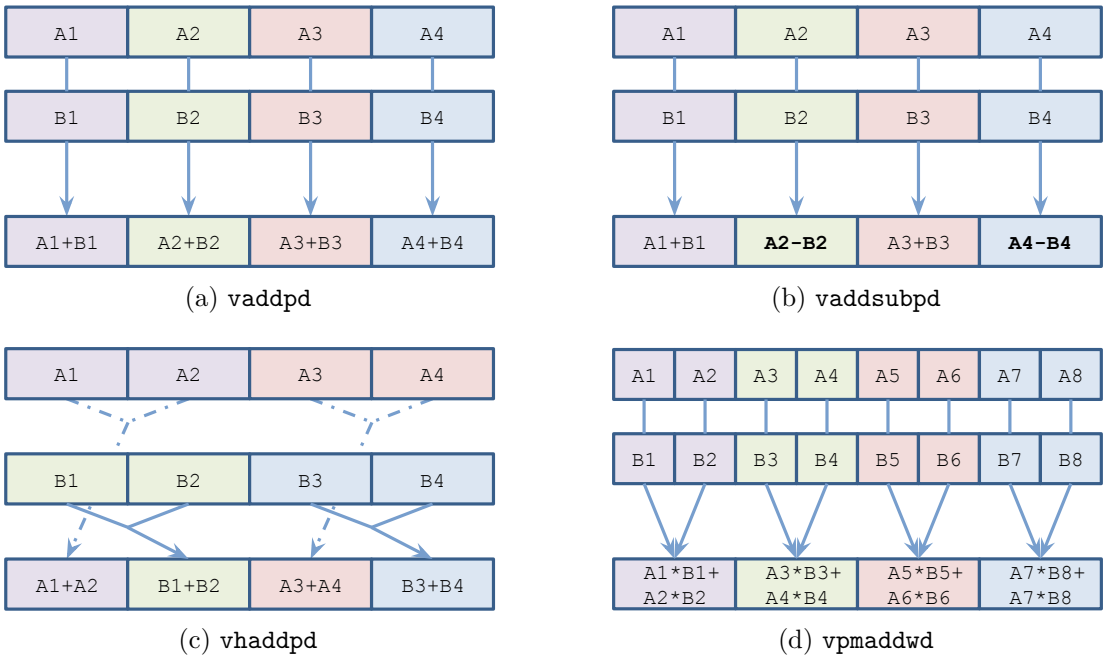

Figure 1-1: Examples of SIMD and non-SIMD instruction in the AVX2 instruction set. We use different colors to indicate how the input values flow to different output lanes.

plies the operations elementwise (i.e., there is no cross-lane operation). Relying on these two assumptions, these algorithms enable compiler developers to support SIMD instructions across a variety of architectures with relatively little incremental effort.

Existing Support for Non-SIMD Instructions. Because non-SIMD instructions violate the two fundamental assumptions of existing vectorization algorithms, compiler developers support non-SIMD instructions using ad hoc approaches that are cumbersome and often ineffective. For most non-SIMD instructions, compiler developers support them with backend peephole rewrites. However, because these peephole rewrites do not generate vector instructions by themselves—they fuse sequences of SIMD instructions and vector shuffles into more non-SIMD instructions—relying on peephole rewrites alone is ineffective. A relatively more effective but more labor-intensive strategy involves coordinating with the compiler's vectorizers to generate SIMD vector patterns that are tailored for those rewrite rules. For instance, the initial support in LLVM [Lattner and Adve, 2004] for the `addsub` instruction family (Figure 1-1(b)) required three coordinated changes to LLVM: refactoring LLVM's

SLP vectorizer to support alternating opcodes, changing LLVM's cost model to recognize a special case of vector shuffle (blending odd and even lanes), and modifying LLVM's backend lowering logic to detect the special patterns generated by the SLP vectorizer. As processor vendors continue to add more complex non-SIMD instructions, this methodology is not sustainable. Compilers are falling behind in identifying the complex code sequences that can be mapped to these instructions, and these multibillion-dollar investments by the processor vendors in enhancing the vector instruction sets go underutilized without expert developers manually writing assembly or compiler intrinsics.

VeGen. In this thesis, I describe an extensible framework for systematically targeting non-SIMD vector instructions. We define a new model of vector parallelism more general than SIMD parallelism, and we present a vectorizer generator that can effectively extract this new model of parallelism using non-SIMD instructions.

To broaden the parallelism modeled by existing vectorizers, we introduce *Lane Level Parallelism* (LLP), which generalizes superword level parallelism (SLP) [Larsen and Amarasinghe, 2000] beyond SIMD in two ways: (1) An instruction can execute multiple non-isomorphic operations, and (2) the operation on each output lane can use values from arbitrary input lanes. These two properties of LLP depend on the semantics of a given target vector instruction. Consequently, our framework encapsulates the two LLP properties (i.e., which operation executes on a given lane and which values the operation uses) in a couple of target-dependent vectorization utility functions. By interfacing with these utilities, the core vectorization algorithm in our framework remains target-independent, as traditional vectorization algorithms do.

We realize this framework with VeGen, a system that automatically generates target-architecture-aware vectorizers to uncover LLP in straight-line code sequences while using only instruction semantics as input. From these instruction semantics, VeGen automatically generates the implementation of the aforementioned vectorization utilities as a compiler library to describe the specific kind of LLP supported by the target architecture. With this automatically generated target-description library, VeGen's vectorizer can automatically use non-SIMD vector instructions. We

added support for newer classes of non-SIMD vector instructions (e.g., those found in AVX512-VNNI, which are not fully supported by LLVM) by providing only their semantics.

We make the following contributions in this thesis:

- We introduce Lane Level Parallelism, which captures the type of parallelism implemented by both SIMD and non-SIMD vector instructions.
- We describe a code-generation framework that jointly performs vectorization and vector instruction selection while maintaining the modularity of traditional target-independent vectorizers designed for SIMD instructions.
- We present VeGen, a vectorizer generator that automatically uses complex non-SIMD instructions using only their documented semantics as input.
- We integrated VeGen into LLVM. VeGen can use non-SIMD vector instructions effectively, e.g., getting speedup 3× (compared to Clang's vectorizer) on x265's idct4 kernel.

1.1 Teaser: Using a neural network instruction

In Figure 1-2, we compare VeGen with three production compilers on a kernel used by TVM's [Chen et al., 2018] 2D convolutional layers. Figure 1-2(a) shows the naive scalar implementation of this kernel. Figures 1-2(b)–1-2(e) show the assembly output of ICC 19.0.1, GCC 10.2, LLVM 10.0, and the VeGen-generated vectorizer, respectively. All code generators were configured to target AVX512-VNNI.

VeGen's vectorizer generates by far the shortest assembly code sequence, 15.25× shorter than the next shortest code generator, LLVM, and the generated code runs 5× faster than LLVM's. VeGen's vectorizer uses a new AVX512-VNNI instruction (`vpdpbusd`); GCC uses some of the integer vector instructions introduced in SSE4 (`vpaddd` and `vpmullw`); LLVM uses a mix of SSE and AVX512 instructions (`vpaddd` and `vpmulld` operating on the 512-bit `zmm` registers); and ICC, Intel's own compiler,

(a) Reference Implementation

```
void
dot_16x1x16_uint8_int8_int32(
 uint8_t data[restrict 4],
 int8_t kernel[restrict 16][4],
 int32_t output[restrict 16]) {
  for (int i = 0; i < 16; i++)
    for (int k = 0; k < 4; k++)
      output[i] +=
        data[k] * kernel[i][k];
}
```

(b) ICC

```
movzx r11d, [rdi]
movsx eax,  [rsi]
imul  r11d, eax
...
add   r11d, r10d
add   r11d, ecx
mov   [rdx], r11d
```

(c) GCC

```
vmovdqa      xmm0, [rip]
vmovdqu      xmm1, [rsi]
...
vpmovsxbw    xmm7, xmm6
vpbroadcastw xmm5, xmm5
vpmullw      xmm7, xmm7, xmm9
vpsrldq      xmm2, xmm6, 8
...
```

	(b) ICC	(c) GCC
Number of Instructions	273	106
Speedup Relative to ICC	1.0×	1.5×
Vector Extensions Used	Not Vectorized	SSE4

(d) LLVM

```
vmovdqu   xmm6, [rsi + 32]
vmovdqu   xmm7, [rsi + 48]
...
vpmulld   zmm1, zmm11, zmm1
vpaddd    zmm1, zmm1, [rdx]
vpmovsxbd zmm3, xmm3
vpmulld   zmm3, zmm10, zmm3
...
```

(e) VeGen

```
vmovdqu64     zmm0, [rdx]
vpbroadcastd  zmm1, [rdi]
vpdpbusd      zmm0, zmm0, [rsi]
vmovdqu64     [rdx], zmm0
```

	(d) LLVM	(e) VeGen
Number of Instructions	61	4
Speedup Relative to ICC	2.2×	11.0×
Vector Extensions Used	SSE4 & AVX-512	AVX512-VNNI

Figure 1-2: One of the dot-product kernels used by TVM's 2D convolution layer (Figure 1-2(a)). Compiler generated assembly and statistics for Intel's compiler ICC (Figure 1-2(b)), GCC (Figure 1-2(c)), LLVM (Figure 1-2(d)), and VeGen (Figure 1-2(e))

does not vectorize the code. This is in spite of many man-hours spent on these compilers to support Intel's multibillion-dollar investment in these vector extensions. In contrast to these manual engineering efforts to target new vector extensions, the target-specific components of VeGen are automatically generated from semantics.

In this example, VeGen's vectorizer uses a new dot-product instruction (`vpdpbusd`) introduced in the AVX512-VNNI instruction set. No other evaluated compilers were able to use this instruction. It is important to note that VeGen's output (Figure 1-2(e)) cannot be generated simply by pattern matching because of the extra data movement using the instruction `vbroadcastw`, which reorders the inputs of `vpdpbusd`.

VeGen allows compilers to target new vector instructions with less development effort. Thus, we believe this new capability will enable the creation of more robust vectorizers in production compilers.

Chapter 2

Lane Level Parallelism

Lane Level Parallelism (LLP) is our relaxation of superword level parallelism (SLP) [Larsen and Amarasinghe, 2000], which models short-vector parallelism (in which an instruction executes multiple scalar *operations* in parallel) with the following restrictions:

- The operations execute in lock-step.
- The inputs and outputs of the operations reside in packed storage (usually implemented as vector registers). We refer to an element of such packed storage as a *lane*.
- The operations are isomorphic.
- The operations are applied elementwise (i.e., there is no cross-lane communication).

LLP relaxes SLP by removing the last two restrictions: (1) The operations can be non-isomorphic, and (2) an operation executing on one lane can use values from another input lane.

Non-isomorphism. LLP allows different operations to execute in parallel, whereas SLP applies only one operation across all vector lanes. An example of an instruction that uses such a parallel pattern is the x86 instruction `vaddsubpd` (Figure 1-1(b)), which does addition on the odd lanes and subtraction on the even lanes.

Cross-lane communication. LLP allows an operation executing on one lane to access values from another input lane (as long as the lane is selected statically). In contrast, SLP restricts an operation to use values from its own input lane. This flexibility is useful for computations that require communication between lanes (e.g., parallel reduction). For example, `vhaddpd` horizontally combines pairs of lanes using addition and then interleaves the results (Figure 1-1(c)).

These properties of LLP depend on the semantics of individual instructions. Different instructions can use different combinations of operations or apply different cross-lane communication patterns.

Chapter 3

VeGen's Workflow

The key idea of VeGen is to encapsulate the details of the two LLP properties (non-isomorphism and cross-lane communication) behind two interfaces. VeGen views a given vector instruction as a list of operations, each of which associated with a pattern matcher (interface 1). Each vector instruction has a lane-binding function that tells VeGen how the input lanes bind to the operations (interface 2). VeGen generates the implementations of these two interfaces offline. At compile time, VeGen's target-independent vectorization algorithm works by first using the pattern matcher to find independent IR fragments that can be packed into the available vector instructions, then using the lane binding rule to identify the vector operands used by the packed vector instructions, and then recursively finding other IR fragments that can be packed to produce those vector operands.

Figure 3-1 shows the workflow of VeGen. VeGen targets non-SIMD (and SIMD) vector instructions in two phases. In the offline phase, VeGen takes instruction semantics (encoded in its vector instruction description language) as input and generates the target-dependent utility functions, such as the pattern matchers. At compile time, VeGen's target-independent heuristic uses the generated utility functions to combine independent streams of scalar instructions into vector instructions.

To target a new vector instruction set, VeGen only requires the compiler writers to describe the semantics of each instruction in VeGen's vector instruction description language. If the vendor has provided instruction semantics in a machine-readable

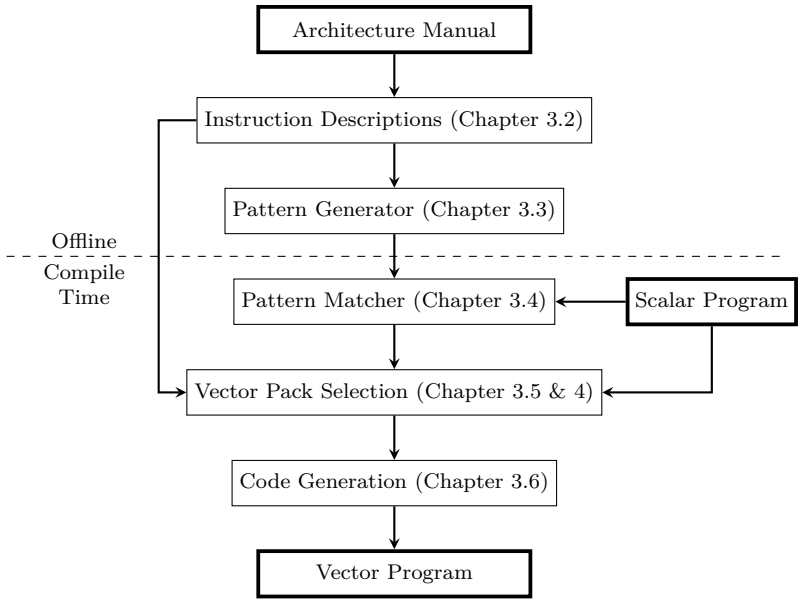

Figure 3-1: VeGen's workflow. Bolded boxes represent artifacts such as manuals and programs.

format such as Intel's Intrinsics Guide [Corporation, 2012], this process can be automated. In Chapter 5, we describe how VeGen automatically translates semantics from the Intrinsics Guide.

Terminology & Notation. We use two related but distinct terms: *instructions* and *operations*. *Instructions* can refer to either IR instructions such as LLVM IR or target instructions such as x86 instructions. *Operations* refer to (side-effect free) bit-vector functions that can be implemented both by IR and target instructions.

For brevity, we overload common set operations for vectors. While doing so, we implicitly convert a vector to a set before applying the set operator. For example, let x be a vector and i a scalar; when we say $i \in x$ we mean that x contains i.

3.1 An end-to-end example of VeGen using a non-SIMD instruction

Before I discuss in more details the different components of VeGen, I will give a short walkthrough of how VeGen uses `pmaddwd`, a complex non-SIMD instruction

$$\begin{aligned}\mathsf{op}_{\mathsf{madd}} &= (x_1 : 16, x_2 : 16, x_3 : 16, x_4 : 16) \mapsto \\ &\quad \mathsf{add}(\mathsf{mul}(\mathsf{sext32}(x_1), \mathsf{sext32}(x_2)), \mathsf{mul}(\mathsf{sext32}(x_3), \mathsf{sext32}(x_4))) \\ \mathsf{pmaddwd} &= (a : 4 \times 16, b : 4 \times 16) \mapsto \\ &\quad [\mathsf{op}_{\mathsf{madd}}(a[0], b[0], a[1], b[1]), \mathsf{op}_{\mathsf{madd}}(a[2], b[2], a[3], b[3])]\end{aligned}$$

Figure 3-2: Semantics of `pmaddwd` formalized in VeGen's vector instruction description language

supported by x86. This walkthrough should illustrate on a high level what these components of VeGen do and how they fit together.

We first start with the semantic description of the instructions. The following is Intel's pseudocode documentation of `pmaddwd`:

```
FOR j := 0 to 3
  i := j*32
  dst[i+31:i] :=
    SignExtend32(a[i+31:i+16]*b[i+31:i+16]) +
    SignExtend32(a[i+15:i]*b[i+15:i])
ENDFOR
```

In its offline phase, VeGen translates this semantic description into the following representation: We call this internal representation *Vector Instruction Description Language (VIDL)*. The main purpose of VIDL is to abstract away the syntactic details of instruction documentation that is irrelevant for code generation.

From this formal representation, VeGen then generates two utility functions that VeGen's vectorization heuristic will eventually use at compile time. The following is the generated pattern-matching function that recognizes the type of operations that `pmaddwd` applies for each of its output vector elements.

```
bool match_MADD_Op(llvm::Value *V, Match &M) {
    llvm::Value *t0, *t1, *t2, *t3;
    if (m_c_Add(m_c_Mul(m_SExt(t0), m_SExt(t1)),
                m_c_Mul(m_SExt(t2), m_SExt(t3))).match(V)) {
      M.LiveIns = { t0, t1, t2, t3 };
      return true;
    }
    return false;
}
```

VeGen also generates a lane-binding function. The lane-binding function informs VeGen's vectorization heuristic that, if it combines multiple such operations (as recognized by the above pattern-matching function) into a single instruction using `pmaddwd`, how the live-ins of matched operations should bind to the input vector lanes.

```
std::vector<llvm::Value *>
operand_1_pmaddwd(const std::vector<Match> &Matches) {
    return { Matches[0].LiveIns[0], Matches[0].LiveIns[2],
             Matches[1].LiveIns[0], Matches[1].LiveIns[2] };
}
```

At compile time, VeGen then uses these automatically generated utilities to vectorize its input program. Suppose we have the following input scalar program.

```
int16_t A[4], B[4];
int32_t C[2];
void dot_prod() {
    C[0] = A[0] * B[0] + A[1] * B[1];
    C[1] = A[2] * B[2] + A[3] * B[3];
}
```

At compile time, VeGen vectorizes its input program in three steps:

1. Apply the auto-generated pattern matchers for the target-specific operations.

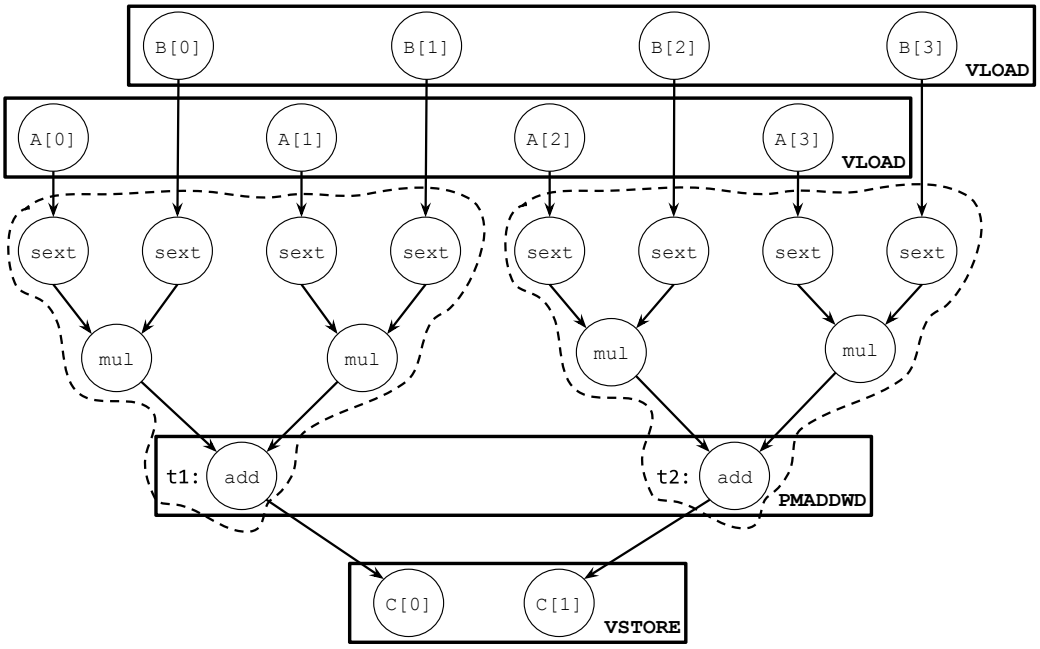

Figure 3-3: The instruction DAG corresponding to the example scalar program. The regions enclosed by the dotted curves represent matched integer multiply-add operations. The rectangles represent vector packs.

```
vmovd xmm0, [A]
vmovd xmm1, [B]
pmaddwd xmm0, xmm1, xmm0
vmovd [C], xmm0
```

Figure 3-4: Generated vector code

2. Select a profitable subset of the matched scalar instructions that can be combined into (potentially non-SIMD) vector instructions.

3. Combine the selected scalar instructions into vector instructions.

Figure 3-3 shows VeGen matching (step 1) and combining (step 2) the operations of `pmaddwd` that occurs in the input program. Figure 3-3 shows the result of code generation (step 3).

$$
\begin{aligned}
x &\in \textbf{variables} \quad i \in \textbf{integers} \\
sz &\in \textbf{bit-widths} \quad vl \in \textbf{vector-lengths} \\
lane &::= x[i] \\
expr &::= x \mid lane \mid binop(expr_1, expr_2) \mid \\
&\quad unop(expr) \mid \textbf{select}(expr_1, expr_2, expr_3) \\
opn &::= (x_1 : sz_1, \ldots, x_n : sz_n) \mapsto expr \\
res &::= opn(lane_1, \ldots, lane_n) \\
inst &::= (x_1 : vl_1 \times sz_1, \ldots, x_n : vl_n \times sz_n) \mapsto [res_1, \ldots, res_m]
\end{aligned}
$$

Figure 3-5: Syntax of the Vector Instruction Description Language (VIDL). $\mapsto$ denotes function abstraction.

3.2 Vector Instruction Description Language

VeGen uses its vector instruction description language (VIDL) to model the semantics of each target vector instruction as a list of scalar operations, with lane-binding rules indicating how the input lanes bind to the operations. Figure 3-5 shows the syntax of VIDL. VIDL assumes that target instructions read and write to registers but have no other side-effects. VeGen models memory instructions such as vector load as a separate, special class of instructions. VIDL only allows selecting the input lanes using constant indices: This restriction allows VeGen to statically determine the vector operands used by each vector instruction.

Figure 3-2 shows the semantics of the SSE instruction `pmaddwd` specified in VIDL. The instruction `pmaddwd` takes two vector registers as input, sign-extends the values from 16-bit to 32-bit temporaries, multiplies the sign-extended values element-wise, and finally adds together every adjacent pair of the multiplication results.

3.3 Generating Pattern Matchers

In the offline phase, VeGen collects the set of operations used by the target vector instructions, and for each operation, VeGen generates pattern matching rules to recognize IR sequences that implement the operation. Figure **??** shows an example of the pattern matching code generated by VeGen.

We designed VIDL to mirror the scalar IR that its vectorizer takes as input.

Thus, generating pattern matching code from VIDL is generally straightforward. In Chapter 5 we discuss how to generate pattern matchers that are more robust.

3.4 Pattern Matching

At compile time, VeGen applies the generated pattern matchers on the input scalar program. We call the result of pattern matching a *match*, an IR instruction DAG with (possibly) multiple live-ins and a single live-out. VeGen represents each match as a tuple consisting of its live-ins, live-out, and operation. In the running example (Figure 3-3), the integer multiply-add operation has two matches (the sub-graphs enclosed in dotted curves): one rooted at the instruction t_1, and another rooted at t_2.

Unlike other common applications of pattern matching such as term rewriting, VeGen does not directly use the result of pattern matching to rewrite the program. Instead, VeGen records the matched patterns in a match table, which records the mapping $\langle \textit{live-out}(m), \textit{operation}(m) \rangle \mapsto m$, for each match m. The match table allows VeGen's target-independent vectorization algorithm (Chapter 3.5) to efficiently enumerate the set of candidate vector instructions that can produce a given vector (Algorithm 1).

3.5 Vectorization

After running the generated pattern matchers (at compile time), VeGen (1) uses a target-independent heuristic to find profitable groups of matched IR instructions that can be packed into (possibly non-SIMD) vector instructions—we call such a group of instructions a *vector pack*—and then (2) lowers the vector packs into target vector instructions.

Vector Pack. A *pack* is a tuple $\langle v, [m_1, \ldots, m_k] \rangle$, where v is a vector instruction with k output lanes, and $m_1, \ldots, m_k$ are a list of matches whose live-outs are independent. For example, let m_1 and m_2 be the two matched integer multiply-add operations rooted at the instructions t_1 and t_2 in Figure 3-3, we can use the instruction `pmaddwd`

to combine them into a single vector pack:

$$p_{ex} = \langle pmaddwd, [m_1, m_2] \rangle$$

VeGen models vector loads and stores as two special kinds of packs, whose memory addresses must be contiguous.

We define two notations for vector packs. Let $p = \langle v, [m_1, \ldots, m_k] \rangle$ be a vector pack, then then $values(p)$ is the list of IR values produced by pack p (i.e., $values(p)_i = \textit{live-out}(m_i)$) and $opcode(p) = v$. In the running example,

$$values(p_{ex}) = [t_1, t_2]$$
$$opcode(p_{ex}) = pmaddwd$$

Vector Operand. Vector packs have vector operands, represented as lists of IR values. In the running example, p_{ex} has two vector operands (We overload the [.] operator here; e.g., $A[0]$ denotes a load of the first element of A):

$$operand_1(p_{ex}) = [A[0], A[1], A[2], A[3]]$$
$$operand_2(p_{ex}) = [B[0], B[1], B[2], B[3]]$$

More specifically, let $p = \langle v, [m_1, \ldots, m_k] \rangle$ be a vector pack, then $operand_i(p) = [x_1, \ldots, x_n]$; where $x_j \in \bigcup_k \textit{live-ins}(m_k)$ is one of the live-ins of the matches that should bind to the j'th lane of the i'th operand of the vector instruction v. VeGen generates the implementation of $operand_i(.)$ automatically from instruction semantics; $operand_i(.)$ is known statically because the VIDL only allows selecting input vector lanes using constant indices.

Don't-Care Lanes. Some instructions don't use all of their input lanes. For example, the SSE4 instruction `vpmuldq` (Figure 3-6) sign-extends and multiplies *only* the odd input lanes. To handle a case, we introduce a special *don't-care* value. Each element of a vector operand (i.e., $operand_i(.)$) therefore takes the value of either a

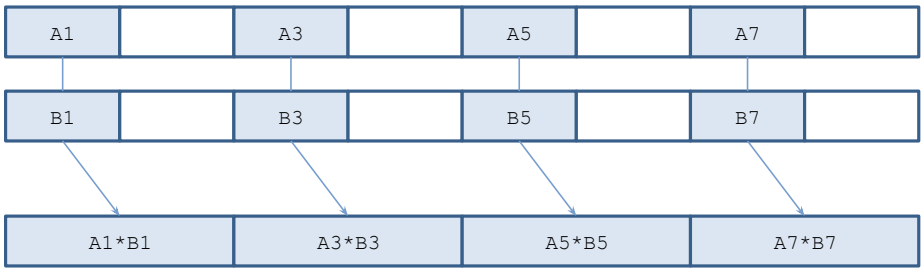

Figure 3-6: Semantics of `vpmuldq` (sign-extended integer multiplication). White cells represent lanes unused by the instruction.

scalar IR value (from the input program) or *don't-care*.

Producing a Vector Operand. A pack p produces a vector operand x if they have the same size (i.e., $|values(p)| = |x|$) and, for every lane i, x_i is either $values(p)_i$ or *don't-care*. Algorithm 1 shows the algorithm for finding the set of feasible producer packs for a given vector operand x. VeGen uses a separate routine to enumerate producer packs that are vector loads, which can be done efficiently because only contiguous loads can be packed together.

Dependence and Legality. A pack p_1 depends on another pack p_2 if there exists an instruction $i \in values(p_1)$ that depends on another instruction $j \in values(p_2)$. We define the dependencies among scalar IR instructions and vector packs similarly. A set of packs are legal when there are no cycles in the dependence graph.

Vector Pack Selection. Because lowering a given set of vector packs to target vector instructions is relatively straightforward, vectorization reduces to finding a subset of the matches and combining them into *legal* vector packs. The choice of packs determines the performance of the generated code by affecting the level of parallelism and the level of data-movement overhead (e.g., if a vector operand is not produced directly, VeGen needs to use vector shuffles to gather the elements of the operand). Given a scalar program, VeGen selects a set of profitable vector packs using two alternative heuristics that we will discuss in Chapter 4.

Algorithm 1: Find the set of (non-load) packs that produce a given vector operand x. Load packs are found separately by enumeration.

Input :
x: The vector operand that we need to produce
M: The match table, which contains the mapping $\langle \textit{live-out}(m), \textit{operation}(m) \rangle \mapsto m$ for each match m.
I: A list of instruction descriptions.

Output : A (potentially empty) set of producer packs of x.

```
1  if there are dependent values in x then
2  |  return {}
3  end
4  producers ← {}
5  for vinst ∈ I do
6  |  matches ← [ ]
7  |  for i ← 1 to number of lanes of vinst do
8  |  |  f ← the i'th operation of vinst
9  |  |  m ← M[⟨x_i, f⟩]
10 |  |  if x_i is don't-care or m is not null then
11 |  |  |  append m to matches
12 |  |  end
13 |  end
14 |  if |matches| = number of lanes of vinst then
15 |  |  producers ← producers ∪ pack(vinst, matches)
16 |  end
17 end
18 return producers
```

3.6 Code Generation

Given a set of vector packs (and the input program), VeGen's code generator emits a vector program as a combination of (1) the scalar instructions not covered by the packs, (2) the *compute* vector instructions corresponding to the packs, and (3) the *data-movement* vector instructions that follow from the dependence among the packs and scalars.

Given a pack set P, we generate vector code as follows. The code generation algorithm uses the target-specific functions $operand_i(.)$ generated from instruction semantics.

Scheduling. The code generator first schedules the *scalar* instructions (regardless of whether an instruction is replaced by vector instructions) according to their dependencies and the following constraint: For any pack $p \in P$, all instructions in $values(p)$ are grouped together in the final schedule. Such a schedule exists when the set of packs are legal.

Lowering. After scheduling, the code generator lowers the packs in P in topological order. The previous scheduling step ensures that all of the values in $operand_i(p)$ are ready by the time we lower any $p \in P$. The code generator also emits any required swizzle instructions to gather a vector operand if the operand is not produced directly by another pack and to extract an element of a vector pack if the pack has a scalar user.

Chapter 4

Vector Pack Selection

VeGen uses a target-independent heuristic to select a set of profitable vector packs. The goal of the heuristic is to select a set of packs to maximize the total saving from vectorization while minimizing the overhead of explicit data-movement that is necessary when an instruction (whether vector or scalar) operand is not produced exactly by any other instruction—such as when a scalar instruction uses a vector element and therefore requires a vector extraction.

Optimization Objective and Cost Model. Let P be the set of selected vector packs, and let us focus on one of the packs $p \in P$. If the results of p are used by some scalar instructions, we need to extract those values and pay the following cost:

$$C_{extract} \cdot |values(p) \cap scalarUses|$$

Let v be a vector operand of p. When a subset of v is produced by some other pack $p' \neq p$, we need to use vector shuffles to move those values into v and pay the following cost:

$$C_{shuffle} \cdot |\{p' \in P \setminus \{p\} \mid v \cap values(p') \neq \emptyset\}|$$

When some elements of v are produced by scalar instructions, we need to use vector

$$cost_{\text{SLP}}(v) = \min \begin{cases} \min\limits_{p \in producers(v)} cost_{op}(opcode(p)) & \text{Produce a vector directly.} \\ \qquad + \sum_i cost_{\text{SLP}}(operand_i(p)) & \\ C_{broadcast} + cost_{scalar}(v_0) & \text{If } \forall i. v_i = v_0, \text{ try broadcast} \\ C_{insert} \cdot |v| + \sum_{x \in v} cost_{scalar}(x) & \text{Produce each element as scalar} \end{cases}$$

Figure 4-1: The SLP heuristic uses this recurrence to decide whether to produce a vector operand v directly via a vector pack or by vector insertions. $cost_{scalar}(x)$ is the cost of producing the (scalar) value x and its dependences using only scalar instructions.

insertions to insert those values into v and pay the following cost:

$$C_{insert} \cdot |v \setminus [\bigcup_{p' \in P} values(p')]|$$

$C_{extract}$, C, and C_{insert} are cost-model parameters.

Recall that VIDL doesn't model vector shuffles (Chapter 3.2). VeGen's code generator therefore emits a mix of target vector instructions and virtual (target-independent) vector shuffles and relies on LLVM's backend to lower the shuffles.

4.1 Pack Selection Using the SLP heuristic

The SLP heuristic builds a set of vector packs by traversing the instruction DAG bottom-up (uses before definitions). Initially, the set of packs are seeded with *seed* packs such as chains of contiguous stores. The heuristic then recursively introduces vector packs to produce the vector operands—VeGen uses Algorithm 1 to find such producers—in the current set of packs.

There are often multiple vector packs that can produce a given operand. For a given operand, VeGen uses the dynamic programming algorithm shown in Figure 4-1 to choose a producer. This is the main modification we added to the original SLP algorithm—in SLP-based vectorization, there is at most one pack that can produce any given operand.

Limitations. A major limitation of the SLP heuristic is that it only considers a

vector pack p if p directly produces the operands of another pack. And there are real-world cases where it's more profitable to include a vector pack even when it's result is not directly used by any other vector packs. Consider the following code that swaps the first two elements of an array.

```
b[0] = a[1];
b[1] = a[0];
```

Notice that we can vectorize this program (manually) by first loading the a elements with two-wide vector load, swap the first and second elements with a vector shuffle, and finally store the shuffled two-wide vector to b. However, we could never generate this code sequence with the SLP heuristic because the vector load of a (which produces the vector $[a_0, a_1]$) doesn't produce the vector operand used by the vector store to a (which requires the vector $[a_1, a_0]$).

Another limitation of the SLP heuristic it assumes that each program value only has one user. Consequently, it is optimistic when there are external scalar users of a vector pack and fails to account for the vector extraction cost. On the other hand, the SLP heuristic is also pessimistic when there are multiple uses of non-vectorizable vector operands and fails to recognize that the multiple uses lower the cost of vector shuffle/insertion (by amortization).

Consider the following code snippet, where there are two seed packs: the two pairs of stores to the arrays `a` and `b`.

```
a[0] = x[0] + t1; a[1] = x[1] + t2;
b[0] = y[0] + t1; b[1] = y[0] + t2;
```

Suppose the temporaries `t1` and `t2` are not vectorizable. To vectorize the rest of the code snippet, the vectorizer would need to emit extra vector insertion instructions to create the vector $[t_1, t_2]$. On a machine where vector insertions are expensive, it is plausible that this code is profitable to vectorize *only* when the instruction (sub-)DAG rooted at both seed packs are vectorized to amortize the cost of creating $[t_1, t_2]$. Unfortunately, because the SLP heuristic processes each seed pack separately,

it would (correctly) conclude that none of the seed packs are individually profitable and (incorrectly) decide that the whole basic block is not worth vectorizing.

Improving the SLP heuristic. Let's rephrase the problem—that the SLP heuristic only consdier vector packs taht are direct producers of existing vector operands—in another way helps with understanding our new heuristic. Let v be some vector operand that we need to produce. We want to find some profitable vector pack p so that p produces v partially ($\mathit{values}(p) \cap v \neq \emptyset$). Note that in the original SLP heuristic, we insist that $\mathit{values}(p) = v$, which is a stricter condition. For such vector pack p, we can break it down to three cases that we will address separately.

Here are the three possible (disjoint) cases that can happen when we have a vector pack p that produces a subset of v's elements.

1. p produces a proper subset of v.

2. p produces exactly the v's elements but with the elements permuted.

3. p produces some values that are not in v. One might wonder why is such pack p desirable: why would anyone want to produce more than that's required? The answer is that sometimes it's more efficient produce multiple vectors jointly as a single vector and then use vector shuffles to extract the sub-vectors out.

The first case requires us to decompose v into smaller vectors and find a producer packs for those vectors separately. Because there are combinatorially many ways to decompose a given vector, we instead use a limited family of vector decomposition operator that we call *deinterleaving*.

The second case requires us to find a way to permute v and then a producer pack for the permuted vector. Similarly, because there are factorially many ways to permute any given vector, we instead use a limited set of permutations that we call *transposition*.

For the third case, we apply a heuristic that we call *combination*, in which we look at all possible pairs of vector operands in an existing set of packs and check whether it's more profitable to produce the elements of the two vectors simultaneously with a

single instruction (note that we still have much freedom in choosing how to combine those vectors, and different *combination* methods can give different performance).

Deinterleaving and *transposition* are non-intrusive in the sense that they still follow the high-level idea of the original SLP heuristic; that is, at each step, we inspect the vector operand x of some existing vector pack p, and we try to find another vector pack p' to produce x. This allows us to reuse the general struture of the dynamic programming algorithm shown in Figure 4-1 with some modifications.

The *combination* heuristic is more intrusive and requires a separate, more expensive optimization algorithm.

4.2 Improve the SLP heuristic with Deinterleaving and Transposition

The idea of *deinterleaving* is that for a given vector x, we decompose x into multiple sub-vectors by a constant stride, and then we attempt to find a producer pack p' that produces one of the sub-vectors exactly. For example, for a four-wide vector x, we can deinterleave its even and odd elements as follows.

$$x' = [x_0, x_2, x_4, x_6]$$
$$x'' = [x_1, x_3, x_5, x_7]$$

And then we recursively find vector packs that can produce x' and x''. In general, we define $\mathit{deinterleave}(m, n, x)$ as a function that takes integers m (the stride) and n (the start position and $1 \leq n \leq m$), an input x x, and returns another vector y whose individual elements are $y_i = x_{m+i \cdot n}$.

With *transposition*, we first transform the an input vector x by interpreting it as an $m \times \frac{|x|}{m}$ matrix (assuming $|x|$ is divisible by m), transposing that matrix, and re-interpreting it as the transformed vector x'. We then attempt to find a producer pack p' that produces x'.

Figure 4-2 shows the improved version of the SLP heuristic once we apply both

$$
cost_{\text{SLP}}(v) = \min \begin{cases}
\min\limits_{p \in producers(v)} cost_{op}(opcode(p)) & \text{Produce a vector directly.} \\
\qquad + \sum_i cost_{\text{SLP}}(operand_i(p)) & \\
\min\limits_{m \in \{2,4,8,\ldots\}} \min\limits_{p \in producers(transpose_m(v))} cost_{op}(opcode(p)) & \text{Transposition} \\
\qquad + \sum_i cost_{\text{SLP}}(operand_i(p)) & \\
\qquad + C_{shuffle} & \\
\min\limits_{m \in \{2,4,8,\ldots\}} \sum_{n \le m} cost_{\text{SLP}}(deinterleave(m, n, v)) & \text{Deinterleaving} \\
\qquad + (m-1) \cdot C_{shuffle} & \\
C_{broadcast} + cost_{scalar}(v_0) & \text{If } \forall i. v_i = v_0, \text{ try broadcast} \\
C_{insert} \cdot |v| + \sum_{x \in v} cost_{scalar}(x) & \text{Scalar insertions}
\end{cases}
$$

Figure 4-2: Improved recurrence for choosing a vector producer for a given vector operand. This is essentially of the algorithm in Figure 4-1 with extra rules for deinterleaving and transposing vectors before looking for the vectors' producers. Note that deinterleaving and transposing are not free. For detinerleaving, we need to use $m-1$ vector shuffles to combine the deinterleaved vectors to produce the final vector. For transposition, we need to use a vector shuffle to permute the transposed vector to produce the final vector.

deinterleaving and *transposition*.

4.3 Fine-tuning with Vector Combination

Combination is an improvement heuristic that we can apply to improve an existing set of vector packs (but doesn't directly give us a set of vector packs directly in the first case). It can give better performance in some cases (by trading off more compile time). The idea is to look at all of the vector operands used in an existing set of vector packs, and find some pair of vector operands that are more profitable to be produced jointly. This leaves two problem: 1) Which pair of vector operands? 2) And how to produce them jointly?

For the first problem, the *combination* heuristic simply checks all pairs of vector

operands by following steps iteratively until it reaches an fixed-point:

1. Enumerate all possible pairs of vector operands (v, v') s.t. elements of v and v' are independent and $|v| = |v'|$, in some arbitrary order

2. Combine each pair of vectors v and v' into a larger vector $combine(v, v')$ (I will discuss what the *combine* function is).

3. Apply the SLP heuristic to find a new set of vector packs that produces the vector $combine(v, v')$ and its transitive dependences; if any of the new vector packs overlap with an existing one, discard the existing one.

4. If the new set of vector packs are more profitable, accept and go back to step 1.

We need some way to comebine two vectors into a larger vectors. For this, we use two alternative combination functions—*concat* and *interleave*—and pick whichever that gives better performance. The function $concat(.,.)$ concatenates its two input vectors. The function $interleave(.,.)$ combines into two input vectors so that the odd elements of the final vector come from the first vector and the even elements come from the second.

Chapter 5

Implementation

5.1 Supporting x86 Vector Instructions

We implemented the offline part of VeGen (the part involved with semantics and pattern generation) in Python. We implemented the rest of VeGen, the part that performs compile time vectorization, as an LLVM pass in C++. The LLVM pass takes scalar LLVM IR as input and emits a mix of scalar IR and target-specific intrinsics[1] that in most cases, gets lowered to their corresponding instructions (e.g., the LLVM intrinsic `@llvm.x86.sse2.pmadd.wd` maps to the instruction `pmaddwd`).

5.2 Target Instruction Specification

VeGen generates SMT formulas from the XML file that Intel uses to render the Intrinsics Guide [Corporation, 2012], which contains pseudocode documentations of the intrinsics. VeGen then lifts the SMT formulas to VIDL (vector instruction description language). Lifting the SMT formulas to VIDL is straightforward because we designed VIDL to closely match the semantics of SMT bit-vector operations (which are also closely related to LLVM's integer instructions).

[1]There is a straightforward mapping from Intel intrinsics to small sequences of LLVM intrinsics. We find out the mapping from Intel intrinsics to the equivalent LLVM intrinsics by wrapping an intel intrinsic in a standalone function whose signature matches that of the intrinsic. We run Clang on this function, and record the instructions produced by Clang

Translating Semantics from the Intrinsic Guide. To document instruction semantics, Intel uses an imperative language that operates on fixed-length bit-vectors. All values in the language are bit-vectors and have one of four types: signed integer, unsigned integer, float, and double. There are no implicit integer overflows in this language; instead, if an operation can overflow its result (such addition and multiplication), the operation first converts its input bit-vectors to a wider width—using zero- or sign-extensions, depending on the signedness—before execution.

We implemented a symbolic evaluator for the language using z3 [De Moura and Bjørner, 2008] and translated Intel's pseudocode documentation into formal SMT formulas. We chose z3 mostly for its expression simplifier. The evaluator maps expression-level constructs such as ALU operators and bit-vector slicing to their SMT equivalents; for instance, additions become SMT bit-vector additions. We treat the following high-level program constructs specially:

- *Assignment.* We model each assignment to (sub-)bit-vector as a pure expression that takes the original bit-vector value and outputs the post-update value. The output of the expression is a concatenation of the unaffected sub-vector(s) and the updated sub-vector.

  Consider, for example, the statement `x[7:0] = 0`, which zeros the lower eight bits of a 32-bit variable `x`, we emit the following formula:

  ```
  Concat(Extract(31,8,x), 0b00000000)
  ```

- *Function calls.* We inline all function calls.

- *Loops.* We unroll all for-loop (All for-loops have constant trip-counts in the documentation language).

- *If-statements.* We apply if-conversion to the sub-vector being mutated—bit-vector assignment is the only construct with side-effects. In the if-converted expression, we set the predicate to the condition of the original if-statement, the true-branch to the right-hand side of the assignment, and the false-branch to the original value of the sub-vector.

For example, for the following statement, which conditionally zeros the lower eight bits of a 32-bit variable `x`,

```
IF ctrl[1:0]
    x[7:0] = 0
FI
```

we emit the following formula:

```
Concat(Extract(31,8,x),
       If(Extract(0,0,ctrl) == 1,
          Extract(7,0,x),
          0b00000000))
```

Our symbolic evaluator returns SMT formulas that are unnecessarily complicated in some cases because of the naive implementation of partial bit-vector updates and predicated updates. We use z3's simplifier to reduce the formula complexity. For most instructions, z3's simplifier simplifies their symbolic results into representations that reflect the high-level intent of the original documentation.

We validated the SMT formulas by random testing. Testing revealed incorrect semantics resulting from ambiguous or simply incorrect documentation. For instance, the signedness of saturation arithmetic is particularly ambiguously documented for instructions from the `psubus` family (subtract packed unsigned integers with saturation). It turns out the result of an *unsigned* subtraction should be saturated as a *signed* integer.

Pattern Generation. We use LLVM's pattern-matching library to implement VeGen's pattern matching logic. VeGen canonicalizes the patterns before emitting the pattern matchers. The canonicalizer takes a pattern and generates an LLVM function that has the same signature as the operation. We then run LLVM's instcombine pass on this function and generate pattern matching code according to the final canonicalized IR sequences. This canonicalization biases the patterns toward patterns that LLVM prefers. The most notable rewrite is canonicalizing all comparisons to strict inequalities (such as rewriting $x \leq 1$ to $x < 2$) and is crucial for recognizing integer

saturations. Additionally, for (sub-)patterns of the form $\textbf{select}(\textbf{cmp}(a, b), x, y)$, we generate additional code to also match the inverted case of the comparison.

5.3 Cost Model

For C_{insert} and $C_{extract}$, we use LLVM's cost model. We set $C_{shuffle} = 2$. VeGen additionally detects several special-case vector shuffle and insertion patterns, such as vector broadcast and permutation, and overrides the default cost model.

To estimate the cost of vector instructions, we use the instruction throughput statistics from Intrinsics Guide. To remain compatible with the rest of LLVM's cost model, we set the cost of each intrinsic to be its inverse throughput scaled by a factor of two.

One of VeGen's goals is reducing developer effort. It is important for us to account for the engineering effort as well as which LLVM components VeGen depends on. We implemented the semantics components of VeGen — parsing, symbolic execution, pattern generation, etc. — in 5.5k lines of Python. We implemented the pack selection heuristic with 489 lines of C++. We implemented the rest of the infrastructure with 2.6k lines of C++.

For reference, the closest analog of our vectorizer is LLVM's SLP vectorizer, which takes 7.5k lines of C++. The hand-written backend peephole rule of the SSE4 instruction `pmaddwd` alone takes 129 lines of C++.

VeGen uses the LLVM's target-specific selection DAG builder to lower virtual vector shuffles into hardware instructions, the backend to lower vector intrinsics into hardware instructions, and its cost model to determine the cost of scalar arithmetic instructions and vector shuffles.

x86 Specific Engineering. Most of the aforementioned implementation is target-independent. The only target-specific component is the semantics translator from Intel's documentation to our description language. This translator automatically translated the semantics of 1164 vector instrinsics and is written in 1.1k lines of Python. To target a new vector instruction set, one would only need to supply the

Table 5.1: Summary of the non-SIMD Neon instructions that VeGen supports

Category	Semantics Summary	Number of Instructions
Pairwise addition	Adding every pair of lanes	41
Folding min/max	Take the maximum of every pair of lanes	14
Dot product	-	4

semantic description of the instruction set. The main dependence is LLVM's shuffle lowering machinery, since (in general) there's a straightforward mapping from vector intrinsic to hardware instructions.[2]

5.4 Supporting Arm Neon Instructions

VeGen also has support for a limited subset of Arm's Neon instructions. Rather than using an automatic translator of Arm's instructions (i.e., following what we did for x86), we translated the semantics of these Neon instructions with a semi-automatic approach. Most Neon instructions come from a small family of instructions (e.g., vector add) but parametrized with different vector-length and bit-width (e.g., 4-wide 32-bit vector add), we implemented a semantics generator for various families of vector instructions and instantiating them with the vector-lengths and bit-widths that Arm supports.

We emit SMT formulas with this semantics generator. We then reused the same code-generation pipeline as we do for x86 vector instructions to process the generated SMT formulas.

All told, we added support for 152 Neon instructions. Table 5.1 lists the non-SIMD instructions in this subset of supported instructions. There are more Neon instructions whose semantics we could generate but not included here because we could not find available Arm machines that can execute those instructions (e.g., matrix multiplications).

Because Arm doesn't have a publicly available, machine-readable file detailing the

[2]Our implementation actually emits target-independent virtual instructions for common arithmetic operations such as addition, which do not have dedicated LLVM intrinsic. For instance, Clang directly emits virtual vector addition for the compiler intrinsic `_mm_add_epi32`.

throughput and latencies of their instructions (unlike Intel), VeGen assigns a uniform cost of two for all the Neon instructions that it supports.

Chapter 6

Experimental Results

We evaluated VeGen on a subset of LLVM's vector instruction selection tests, some reference DSP kernels chosen from FFmpeg and x265, and fixed-size dot-product kernels from OpenCV. We also evaluated VeGen's generated Arm Neon backend on the aforementioned DSP kernels. We show that in most cases, VeGen outperforms LLVM's vectorizer, and we explain how VeGen fails to vectorize in the other cases. Additionally, we present a case study of VeGen vectorizing the scalar complex-multiplication kernel.

Experimental Platforms. For experiments requiring only AVX2, we run the benchmarks on a server with the Intel®Xeon®CPU E5-2680 v3 CPU and 128 GB of memory. For experiments requiring AVX512-VNNI, we use a server with the Intel®Xeon®Platinum 8275CL CPU and 4 GB of memory. For experiments requiring Arm Neon instructions, we used an AWS Graviton2 processor (based on the Arm Neoverse N1 microarchitecture). We use LLVM 12.0.0. In all cases, we invoke clang with `-O3 -ffast-math -march=native`.

Note on cost model. During development, we discovered that LLVM's backend code-generator for LLVM `vectorshuffle` (LLVM's target-independent construct to represent arbitrary vector shuffles) are incomplete and crashes lowering the vector shuffles emitted by VeGen while targeting Icelake (the microarchitecture of our AVX512 machine). We got around the bug by configuring LLVM to target Haswell

(a) Tests LLVM able to vectorize

Test	Speedup
max_pd	1.0
min_pd	1.0
max_ps	1.0
min_ps	1.0
mul_addsub_pd	1.0
mul_addsub_ps	1.0
abs_pd	0.8
abs_ps	0.4
abs_i8	1.0
abs_i16	1.0
abs_i32	1.0

(b) Tests LLVM unable to vectorize

Test	Speedup
hadd_pd	1.4
hadd_ps	1.2
hsub_pd	1.4
hsub_ps	1.2
hadd_i16	2.9
hsub_i16	4.9
hadd_i32	1.3
hsub_i32	1.3
pmaddubs	16.8
pmaddwd	4.2

Figure 6-1: Speedup (over LLVM, higher is better) on instruction selection tests ported from LLVM's x86 backend. These tests were originally written to exercise the pass that lowers LLVM's vector IR into their desired target instructions. We ported the tests by manually transforming them into their scalar equivalents.

instead but with extra flags informing LLVM that the (purely hypothetical) Haswell is also equipped with AVX512 features. This allows us to get emit AVX512 instructions but has unfortunately also caused LLVM to use its Haswell cost model even while targeting the Icelake machine. As we analyze the experimental results, we can see evidence that this has an adverse performance effect.

6.1 Synthetic Benchmarks

For our first set of experiments, we ported some of LLVM's backend instruction selection tests for non-SIMD instructions and SIMD instructions with complex semantics (e.g., `min`). These tests were originally written to exercise the pass that lowers LLVM vector IR into target vector instructions. Because LLVM's vector IR only models isomorphic vector instructions, the tests for non-SIMD instructions (e.g., `haddpd`) are written as combinations of LLVM vector instructions and vector shuffles. We translated the test cases (written in LLVM IR) to their equivalent scalar version by expanding IR vector instructions into multiple scalar instructions and by converting vector function arguments to non-aliased pointer arguments. Clang/LLVM is not able to vectorize 10

out of 21 of the tests. Aside from the tests `abs_ps` and `abs_pd`, our system performs equally or better to Clang.

Figure 6-1 shows the test results. VeGen vectorizes 19 out of 21 of the tests. LLVM fails to vectorize 10 out of 21 of the tests, all of which are non-SIMD instructions and are vectorized by VeGen. Interestingly, the only non-SIMD tests that LLVM can vectorize are `mul_addsub_pd` and `mul_addsub_ps`, for which LLVM does have special-case support.

Both of the two tests that VeGen failed to vectorize compute floating-point absolute values, and for which LLVM uses the fact that the absolute value of a floating-point can be computed by masking-off the sign-bit (i.e., the most significant bit) to vectorize; VeGen does not have this knowledge and does not vectorize in these two cases.

6.2 Optimizing Image and Signal Processing Kernels

Table 6.1 shows our benchmarks. To demonstrate that VeGen can effectively use non-SIMD instructions on real-world kernels, we evaluated VeGen's pack selection heuristic on six kernels from x265 We chose these kernels because DSP and image processing are the motivating domains for non-SIMD instructions such as `pmaddwd`. These benchmarks are challenging to vectorize because they require intermediate shuffles and partial reductions. We ported the idct4 and idct8 kernels from x265's reference implementation. The rest are from FFmpeg.

Figure 6-2 shows the benchmarking results on AVX2, AVX512, and (Arm) Neon. VeGen outperforms LLVM in all cases.

6.2.1 Discussion

When considering the best-performing heuristics used by VeGen, VeGen outperforms Clang in all cases; in the best case, VeGen is able to reach more than 5× speedup.

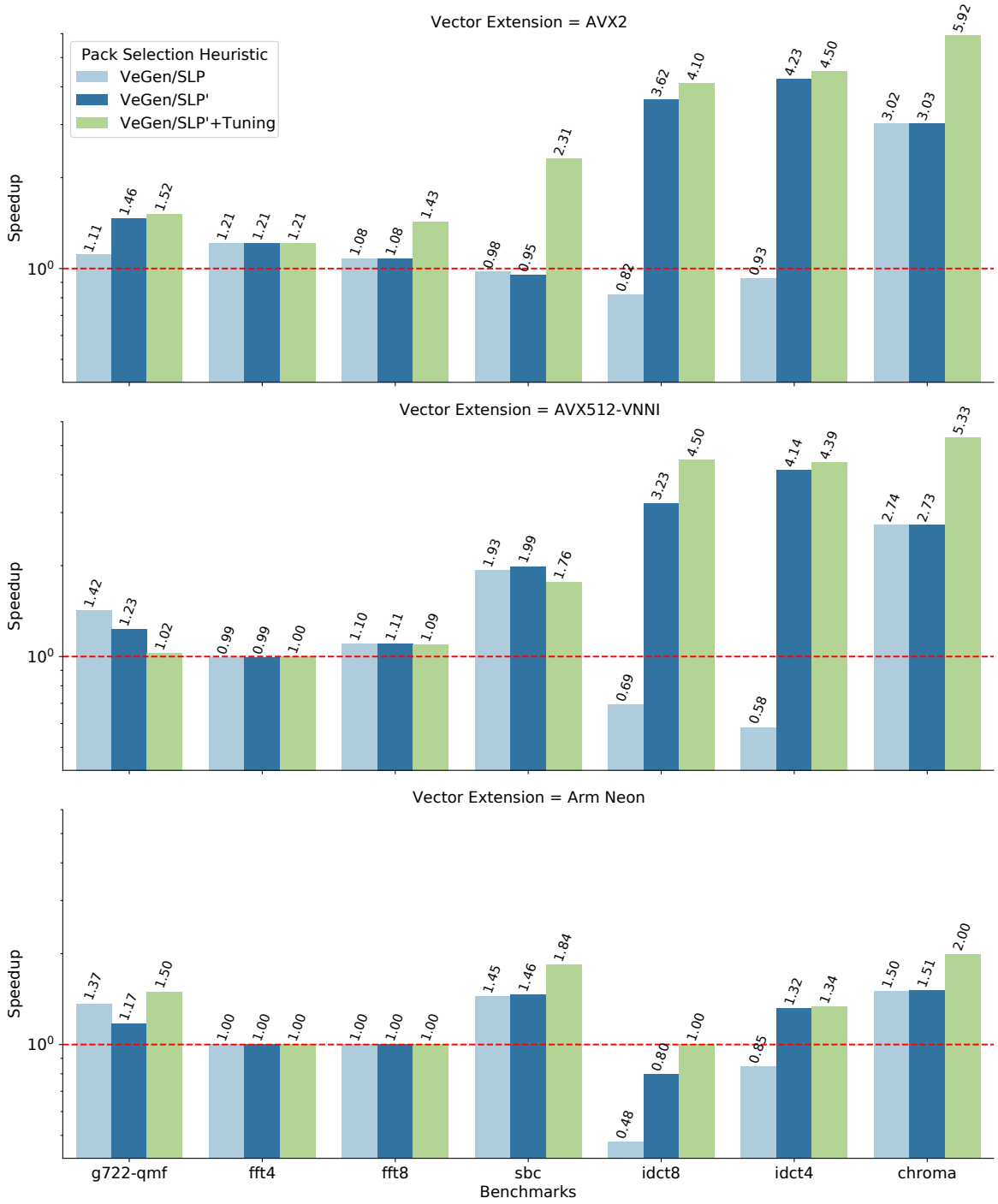

Figure 6-2: Speedup (over LLVM, higher is better) on kernels we selected from x265 (idct4 and idct8) and FFmpeg. `VeGen/SLP` is the baseline SLP heuristic used by VeGen. `VeGen/SLP'` is the SLP heuristic augmented with the *deinterleaving* and *transposition* heuristics. `VeGen/SLP'+Tuning` refers to running the augmented SLP heuristic and then fine-tuning with the *combination* heuristic, which iteratively finds pairs of vector operands that are more profitable to be produced jointly.

Table 6.1: Evaluation Benchmarks

Benchmark	Description
g722-qmf	Quadrature Mirror Filters for G.722 codec
fft4	4-point Fast Fourier Transform
fft8	8-point Fast Fourier Transform
sbc	Audio subband codec
chroma	Chroma subsampling
idct4	4-point Inverse Discrete Cosine Transform
idct8	8-point Inverse Discrete Cosine Transform
imdct36	36-point Modified Discrete Cosine Trasform

```
vmovdqu ymm0, ymmword ptr [rdi]
vpermq  ymm1, ymm0, 119
vmovdqa ymm2, ymmword ptr [rip + LCPI7_0]
vpshufb ymm1, ymm1, ymm2
vpmaddwd    ymm1, ymm1, ymmword ptr [rip + LCPI7_1]
vpermq  ymm0, ymm0, 34
vpshufb ymm0, ymm0, ymm2
vpmaddwd    ymm0, ymm0, ymmword ptr [rip + LCPI7_2]
vpaddd  ymm2, ymm0, ymm1
vpbroadcastd    ymm3, dword ptr [rip + LCPI7_3]
vpaddd  ymm2, ymm2, ymm3
vpsrad  ymm2, ymm2, 7
vpshufd ymm2, ymm2, 78
vpsubd  ymm0, ymm0, ymm1
vpaddd  ymm0, ymm0, ymm3
vpsrad  ymm0, ymm0, 7
vpackssdw   ymm0, ymm2, ymm0
vpshufb ymm0, ymm0, ymmword ptr [rip + LCPI7_4]
vpermq  ymm0, ymm0, 216
vmovdqu ymmword ptr [rsi], ymm0
```

Figure 6-3: idct4 kernels vectorized by VeGen. VeGen fully vectorizes this kernel with 20 vector instructions. In contrast, Clang partially vectorizes this kernel using 84 instructions (not shown here).

The baseline SLP heuristic is in general inferior to the SLP heuristic augmented with the *deinterleaving* and *transposition* heuristic, incurring slowdowns on idct4 and idct8 on all three (micro)architectures.

Aside from the g722-qmf benchmark, the SLP heuristic augmented with *deinterleaving* and *transposition* out-performs the baseline SLP heuristic. For the g722-qmf benchmark, the augmented SLP heuristic and finetuning found vectorization strategies that are actually more profitable according to LLVM's cost model, but the actual generated to runs slower, indicating inaccuracy in the cost model. The main benefit *deinterleaving* and *transposition* show up in the idct4 and idct8 benchmarks, both of which require shuffling the memory inputs before vector computations and require shuffling the computed value before storing back to the memory.

Fine-tuning with the *combination* heuristic (`VeGen/SLP'+Tuning` in Figure **??**) improves the initial vectorization strategy discovered by the augmented SLP heuristic on idct4, idct8, and the chroma benchmarks It improves augmented heuristic by more than 20%.

Vectorizing idct4. We highlight some instructions that VeGen generated for the idct4 kernel (targeting AVX512-VNNI). Figure 6-3 shows the generated code, which is more than 4× faster than LLVM's code. VeGen uses the non-IMSD instructions `vpmaddwd` (the motivating dot-product instruction) and `vpackssdw` (saturate two vectors of 32-bit integers to 16-bit integers and concatenate the result). Of note are the `vpshufb` and `permq` instructions preceding the vector stores. VeGen uses these shuffle instructions—without which it is not profitable to vectorize this kernel—to form vector operands that are not directly produced by *compute* instructions such as `vpmaddwd`.

6.3 Optimizing OpenCV's Dot-Product Kernels

For our next set of experiments, we evaluated VeGen on OpenCV's reference dot-product kernel implementations. OpenCV's reference implementation is a C++ template parameterized with different data types and kernel sizes. These kernels are

challenging to auto-vectorize because they have interleaved memory accesses as well as reduction.

Figure 6-4 shows the benchmarking results. VeGen found non-trivial vectorization schemes for three of the four kernels. VeGen vectorizes the first benchmark naively—essentially vectorizing across the unrolled iterations and paying the shuffle cost for the interleaved accesses—and only yielded a 10% speedup. We investigated the slowdown VeGen incurred on AVX512 (VNNI). It turned out that for the first kernel, VeGen actually emitted identical vector IR/intrinsics for both AVX2 and AVX-512. The performance difference comes down to how LLVM's backend lowered the shuffles emitted by VeGen. For the AVX2, LLVM emitted the `vpshufb` instruction, whose latency and inverse throughput are both one cycle. For the AVX-512, LLVM instead emitted the `vpmovdb` instruction, whose inverse throughput is two cycles (and latency four cycles) and slower than `vpshufb`.

Of note is the vector code VeGen generated for the *int32* $\times$ 8 kernel (Figure 6-5), which matches OpenCV's expert-optimized code. We inspected the machine code and confirmed that VeGen used the same high-level algorithm used by OpenCV's expert developer. The reference (naive) implementation of the *int32* $\times$ 8 kernel sign-extends the input elements from 32-bit to 64-bit, multiplies the two input arrays elementwise, and then reduces every adjacent pair of elements by addition. There is no single instruction that can implement this kernel by itself, and the high-level strategy of VeGen (and OpenCV) is to perform the odd multiplications separate from the even ones and finally add the odd and even entries together. To multiply the odd (and even) entries, VeGen uses the instruction `vpmuldq`, which is deceivingly complicated and performs sign-extended multiplications only on the *odd* input elements (Figure 3-6). The multiplications of the odd elements therefore map naturally to `vpmuldq`.

6.4 Optimizing Complex Multiplication

Complex arithmetic is a motivating application for SIMOMD instructions. In fact, (to the best knowledge of our knowledge) the first SIMOMD instructions were designed

(a) Results on AVX2

Kernel Size	Speedup
int8 × 32	1.1
uint8 × 32	2.0
int32 × 8	1.5
int16 × 16	1.6

(b) Results on AVX-512 (VNNI)

Kernel Size	Speedup
int8 × 32	0.7
uint8 × 32	2.2
int32 × 8	1.7
int16 × 16	2.5

Figure 6-4: OpenCV's dot-product kernels specialized for AVX2 and AVX-512 (VNNI) and different kernel sizes.

```
vmovdqu ymm0, [rdi]
vmovdqu ymm1, [rsi]
vpmuldq ymm2, ymm1, ymm0
vpshufd ymm0, ymm0, 245     ## ymm0 = ymm0[1,1,3,3,5,5,7,7]
vpshufd ymm1, ymm1, 245     ## ymm1 = ymm1[1,1,3,3,5,5,7,7]
vpmuldq ymm0, ymm1, ymm0
vpaddq  ymm0, ymm0, ymm2
vmovdqu [rdx], ymm0
```

Figure 6-5: Vector code that VeGen generated for the *int32* × 8 dot-product kernel in OpenCV. `vpmuldq` multiplies (with sign-extension) the *odd* elements of its two vector operands.

```
vmovupd         xmm0,  rsi
vpermilpd       xmm1,  xmm0, 1
vmovddup        xmm2,  [rdi+8]
vmulpd          xmm1,  xmm1, xmm2
vmovddup        xmm2,  [rdi]
vfmaddsub213pd  xmm2,  xmm0, xmm1
vmovupd         [rdx], xmm2
```

(a) Instructions generated by VeGen (`vfmaddsub213pd` does multiply-add on the odd lanes and multiply-sub on the even lanes)

```
vmovsd      xmm0,  [rdi]
vmovsd      xmm1,  [rdi + 8]
vmovsd      xmm2,  [rsi]
vmovsd      xmm3,  [rsi + 8]
vmulsd      xmm4,  xmm2, xmm1
vfmadd231sd xmm4,  xmm3, xmm0
vmulsd      xmm1,  xmm3, xmm1
vfmsub231sd xmm1,  xmm2, xmm0
vmovsd      [rdx], xmm1
vmovsd      [rdx + 8], xmm4
```

(b) Instructions generated by LLVM

Figure 6-6: Complex multiplication kernel, generated by VeGen (Figure 6-6(a)) and LLVM (Figure 6-6(b)). VeGen's version is 1.27× faster.

for complex arithmetic [Bachega et al., 2004].

Figure 6-6 shows the complex multiplication kernel compiled by VeGen and by LLVM. VeGen uses the instruction `vfmaddsub213pd` (which performs fused multiply-add on the odd lanes and multiply-sub on the even lanes). LLVM does not vectorize in this case, even though (as noted earlier) LLVM's SLP vectorizer has been specifically modified to support such a pattern. We stepped through the LLVM's optimization decisions and discovered that the root cause is an error in its cost-benefit analysis. Since LLVM's SLP vectorizer is target-independent, it models such an alternating pattern as two vector arithmetic instructions followed by a vector blending instruction that combines the results. The error occurs when the LLVM's vectorizer includes the cost of the blending instruction into its analysis and overestimates the total vectorization overhead. VeGen does not suffer from such issues because VeGen has direct knowledge of which target instructions are available.

Chapter 7

Related Work

VeGen is related two long lines of work on compiler research—automatic vectorization and automatic backend-generation from ISA semantics. VeGen is also more broadly related to recent work on program superoptimization and synthesis, both of which seek to automatically generate target programs while relying only on a declarative description of instruction semantics.

7.1 Auto-vectorization

Loop vectorization and SLP vectorization are the two dominant vectorization techniques used by modern compilers. Both types of vectorization techniques do not model non-SIMD vector instruction in principle, but their implementations in mainstream compilers such as LLVM have some special case non-SIMD support.

Nuzman and Zaks [Nuzman et al., 2006] proposed a technique for vectorizing interleaved memory accesses within a loop-based vectorizer. Eichenberger et al. proposed a technique for vectorizing misaligned memory accesses [Eichenberger et al., 2004], and FlexVec [Baghsorkhi et al., 2016] extends loop vectorizers to support vectorizing irregular programs with manually written rules. In contrast, VeGen systematically adds support to generate non-SIMD instructions automatically and is not limited to a particular class of non-SIMD instructions.

The vectorizer generated by VeGen is more similar to SLP vectorization introduced

by Larsen and Amarasinghe [Larsen and Amarasinghe, 2000]. However, VeGen supports a more general type of parallelism (LLP) and can therefore target non-SIMD instructions. Almost all published SLP vectorization techniques propose algorithmic improvements to capture more parallelism within the SLP framework. Some examples are Holistic SLP vectorization [Liu et al., 2012], Super-node SLP [Porpodas et al., 2019], TSLP [Porpodas and Jones, 2015], PSLP [Porpodas et al., 2015], VW-SLP [Porpodas et al., 2018], and ILP solver-aided goSLP [Mendis and Amarasinghe, 2018].

There are domain-specific vectorizers that exploit architecture-specific vector instructions as well as application-specific patterns. The SPIRAL project [Puschel et al., 2005] proposes several auto-vectorization schemes specific to DSP algorithms. More specifically, they propose a target-independent search-based vectorizing compiler targeting DSP algorithms [Franchetti and Püschel, 2002] and show how to use the vector swizzle instructions supported by the AVX and Larrabee ISAs to implement the matrix transpositions found in FFTs [McFarlin et al., 2011]. Compared to SPIRAL and its extensions, VeGen is a general-purpose vectorizer and not designed to target any specific vector instruction sets.

7.2 Instruction Selection

VeGen closely related to the research on building retargetable compilers. VeGen is different from this line of work in that it focuses on extracting fine-grained parallelism (as a vectorizer) while simultaneously being aware of the detailed operations supported by these target instructions (similar to an instruction selector). Instruction selection—regardless of the quality of the code generator—alone is insufficient for automatically targeting non-SIMD vector instructions because traditional instruction selectors only lowers IR vector instructions—thus requiring cooperation with the vectorizer.

Ganapathi et al. [1982] presented a survey on retargetable code generation. Cattell [1980] investigated automatically generating code generators from machine descriptions. Ramsey and Fernández [1997] proposed a specification language for describing instruction encoding. Buchwald et al. [2018] synthesized instruction selection rules for

32-bit x86 integer instructions from their bit-vector specification.

7.3 Superoptimization

VeGen is more broadly related to superoptimization, which uses search techniques to directly generate optimized programs based on instruction semantics. In principle, a superoptimizer can accomplish what VeGen does, but in practice, existing superoptimizers are orders of magnitude slower than auto-vectorizers such as VeGen.

Bansal and Aiken [2006] constructed a peephole superoptimizer by exhaustively enumerating short sequences of x86 instructions. Schkufza et al. [2013] proposed a stochastic superoptimizer that trades completeness for scalability via a Markov Chain Monte Carlo sampler. Barthe et al. [2013] proposed a synthesizing vectorizer that works by first unrolling the scalar code and then using an enumerative synthesizer to find more an efficient vector program that implements the unrolled loop body. Phothilimthana et al. [2016] build on previous work on enumerative [Barthe et al., 2013], stochastic [Schkufza et al., 2013], and solver-based synthesis to scale up superoptimization. Sasnauskas et al. [2017] described a superoptimizer for straight-line scalar LLVM IR.

Chapter 8

Conclusions

I have described a framework for building target-aware vectorizers that can use non-SIMD instructions. We introduce Lane Level Parallelism, a new model of short vector parallelism that captures the kind of parallelism implemented by non-SIMD instructions. We realize this framework with VeGen, a system that takes vector instruction semantics as input and generates a target-aware vectorizer that uncovers LLP found in straight-line code sequences. VeGen is flexible: to target a new vector instruction set, the developers only need to describe the semantics of the new vector instructions. VeGen allows compilers to target new vector instructions with less development effort and thus enable the creation of more robust vectorizers in future compilers.

Bibliography

Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 1987.

Leonardo Bachega, Siddhartha Chatterjee, Kenneth A. Dockser, John A. Gunnels, Manish Gupta, Fred G. Gustavson, Christopher A. Lapkowski, Gary K. Liu, Mark P. Mendell, Charles D. Wait, and T. J. Chris Ward. A high-performance SIMD floating point unit for BlueGene/L: Architecture, compilation, and algorithm design. In *International Conference on Parallel Architecture and Compilation Techniques*, 2004.

Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. Flexvec: Auto-vectorization for irregular loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. From relational verification to SIMD loop synthesis. In *Symposium on Principles and Practice of Parallel Programming*, 2013.

Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *International Symposium on Code Generation and Optimization*, 2018.

R. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transaction on Programming Languages and Systems*, 1980.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Symposium on Operating Systems Design and Implementation*, 2018.

Intel Corporation. Intel Intrinsics Guide, 2012. URL `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

Franz Franchetti and Markus Püschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *International Parallel and Distributed Processing Symposium*, 2002.

Mahadevan Ganapathi, Charles N. Fischer, and John L. Hennessy. Retargetable compiler code generation. *ACM Computing Surveys*, 1982.

ARM Holdings. Arm architecture reference manual Armv8, 2011. URL `https://developer.arm.com/documentation/ddi0487/latest/`.

Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. A compiler framework for extracting superword level parallelism. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

Daniel McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. Automatic SIMD vectorization of fast Fourier transforms for the Larrabee and AVX instruction sets. In *International Conference on Supercomputing*, 2011.

Charith Mendis and Saman Amarasinghe. goSLP: Globally optimized superword level parallelism framework. *Proceedings of the ACM on Programming Languages*, 2018.

Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: Revisited for short SIMD architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

Vasileios Porpodas and Timothy M. Jones. Throttling automatic vectorization: When less is more. In *Conference on Parallel Architecture and Compilation*, 2015.

Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. PSLP: Padded SLP automatic vectorization. In *International Symposium on Code Generation and Optimization*, 2015.

Vasileios Porpodas, Rodrigo CO Rocha, and Luís FW Góes. VW-SLP: auto-vectorization with adaptive vector width. In *International Conference on Parallel Architectures and Compilation Techniques*, 2018.

Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. Super-Node SLP: Optimized vectorization for code sequences containing operators and their inverse elements. In *International Symposium on Code Generation and Optimization*, 2019.

Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 2005.

Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transaction on Programming Languages and Systems*, 1997.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.