VeGen: A Vectorizer Generator for SIMD and Beyond

Yishen Chen
MIT CSAIL
USA
ychen306@mit.edu

Charith Mendis
UIUC
USA
charithm@illinois.edu

Michael Carbin
MIT CSAIL
USA
mcarbin@csail.mit.edu

Saman Amarasinghe
MIT CSAIL
USA
saman@csail.mit.edu

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.

In this paper, we introduce Lane Level Parallelism (LLP), which captures the model of parallelism implemented by both SIMD and non-SIMD vector instructions. We present VeGen, a vectorizer generator that automatically generates a vectorization pass to uncover target-architecture-specific LLP in programs while using only instruction semantics as input. VeGen decouples, yet coordinates automatically generated target-specific vectorization utilities with its target-independent vectorization algorithm. This design enables us to systematically target non-SIMD vector instructions that until now require ad hoc coordination between different compiler stages. We show that VeGen can use non-SIMD vector instructions effectively, for example, getting speedup 3× (compared to LLVM's vectorizer) on x265's idct4 kernel.

CCS CONCEPTS

• **Software and its engineering** → **Translator writing systems and compiler generators**; **Retargetable compilers**; *Specification languages*; Automatic programming; • **Computer systems organization** → **Single instruction, multiple data**.

KEYWORDS

optimization, auto-vectorization, non-SIMD

ACM Reference Format:
Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446692

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446692

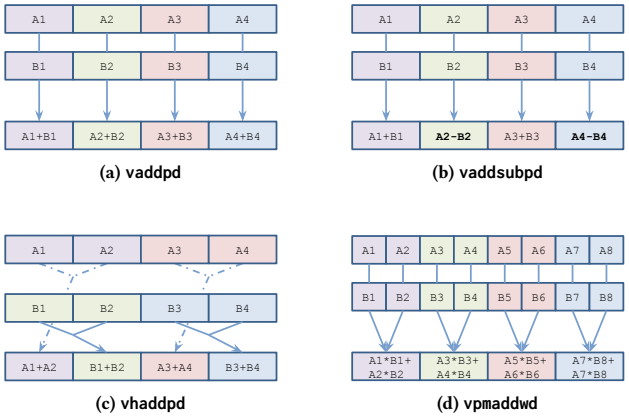


Figure 1: Examples of SIMD and non-SIMD instruction in the AVX2 instruction set. We use different colors to indicate how the input values flow to different output lanes.

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 [14]). In contrast with the SIMD instruction shown in Figure 1(a), Figures 1(b)–1(d) show three examples of the non-SIMD instructions from the AVX2 instruction set. Figure 1(b) shows a single instruction, multiple operations, multiple data (SIMOMD) instruction [2] that performs additions and subtractions on alternating lanes (`vaddsubpd`); Figure 1(c) shows a horizontal addition with lane interleaving (`vhaddpd`); and Figure 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 [1, 20, 21] and superword level parallelism (SLP) based vectorization [15, 17, 24]. Both techniques make two fundamental assumptions about vector instructions: a SIMD instruction performs isomorphic operations across all lanes, and the instruction applies the operations element-wise (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.

(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
...
```

(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
```

	(b) ICC	(c) GCC	(d) LLVM	(e) VeGen
Number of Instructions	273	106	61	4
Speedup Relative to ICC	1.0×	1.5×	2.2×	11.0×
Vector Extensions Used	Not Vectorized	SSE4	SSE4 & AVX-512	AVX512-VNNI

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

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 [16] for the addsub instruction family (Figure 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.

Our Approach: VeGen. In this paper, we 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) [15] 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 paper:

- 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.

2 MOTIVATIONAL EXAMPLE

In Figure 2, we compare VeGen with three production compilers on a kernel used by TVM's [8] 2D convolutional layers. Figure 2(a) shows the naive scalar implementation of this kernel. Figures 2(b)–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, 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 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.

3 LANE LEVEL PARALLELISM

Lane Level Parallelism (LLP) is our relaxation of superword level parallelism (SLP) [15], 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(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(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.

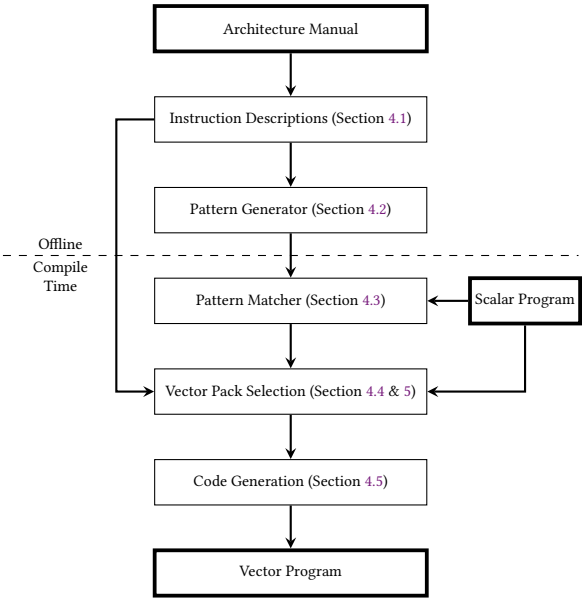


Figure 3: VeGen's workflow. Bolded boxes represent artifacts such as manuals and programs.

4 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 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 format such as Intel's Intrinsics Guide [9], this process can be automated. In Section 6, we describe how VeGen automatically translates semantics from the Intrinsics Guide.

```
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
```

(a) Intel's pseudocode documentation of `pmaddwd`

$$\text{op}_{\text{madd}} = (x_1 : 16, x_2 : 16, x_3 : 16, x_4 : 16) \mapsto \text{add}(\text{mul}(\text{sext32}(x_1), \text{sext32}(x_2)), \text{mul}(\text{sext32}(x_3), \text{sext32}(x_4)))$$

$$\text{pmaddwd} = (a : 4 \times 16, b : 4 \times 16) \mapsto [\text{op}_{\text{madd}}(a[0], b[0], a[1], b[1]), \text{op}_{\text{madd}}(a[2], b[2], a[3], b[3])]$$

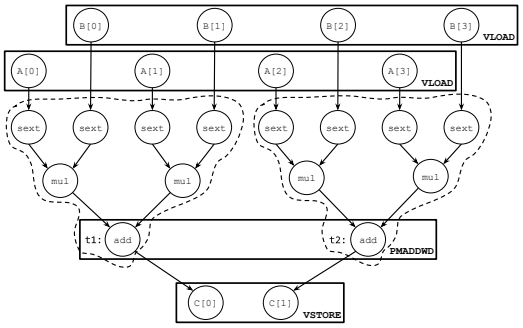
(b) Semantics of `pmaddwd` formalized in VeGen's vector instruction description language

```
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;
}
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] };
}
```

(c) Two examples of the vectorization utilities automatically generated from semantics: a pattern matcher and a function that describes how the input lanes of the first operand bind to the matched operations.

```
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];
}
```

(d) An example scalar program to vectorize.



(e) 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
```

(f) Generated vector code

Figure 4: How VeGen uses the instruction `pmaddwd`. First, VeGen translates the pseudocode semantics of `pmaddwd` (Figure 4(a)) into its vector instruction description language (Figure 4(b)). Next, VeGen generates the vectorization utility functions (Figure 4(c)) used by its vectorizer at compile time. Figure 4(d) shows an example scalar program before vectorization. At compile time, VeGen's vectorizer combines the matched operations into vector packs (Figure 4(e)), which are later lowered into vector assembly code (Figure 4(f)).

Figure 4 shows an end-to-end example of VeGen optimizing an integer dot-product kernel. In the rest of this section, we will use it as a running example.

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.

4.1 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 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 separately. 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 4(b) 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.

4.2 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 4(c) 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 Section 6 we discuss how to generate pattern matchers that are more robust.

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

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

4.3 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 4(e)), 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 (Section 4.4) to efficiently enumerate the set of candidate vector instructions that can produce a given vector (Algorithm 1).

4.4 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 4(e), 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,

$$
\begin{aligned}
values(p_{ex}) &= [t_1, t_2] \\
opcode(p_{ex}) &= pmaddwd
\end{aligned}
$$

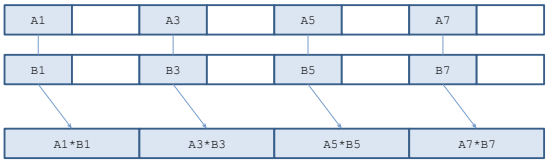


Figure 6: Semantics of `vpmuldq` (sign-extended integer multiplication). White cells represent lanes unused by the instruction.

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):

$$
\begin{aligned}
operand_1(p_{ex}) &= [A[0], A[1], A[2], A[3]] \\
operand_2(p_{ex}) &= [B[0], B[1], B[2], B[3]]
\end{aligned}
$$

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 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 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 Section 5.

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.

```
if there are dependent values in x then
    return {}
end
producers ← {}
for vinst ∈ I do
    matches ← [ ]
    for i ← 1 to number of lanes of vinst do
        f ← the i'th operation of vinst
        m ← M[⟨x_i, f⟩]
        if x_i is don't-care or m is not null then
            append m to matches
        end
    end
    if |matches| = number of lanes of vinst then
        producers ← producers ∪ pack(vinst, matches)
    end
end
return producers
```

4.5 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 $\textit{operand}_i(.)$ 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 $\textit{values}(p)$ 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 $\textit{operand}_i(p)$ 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.

$$cost_{\text{SLP}}(v) = \min \begin{cases} \min\limits_{p \in producers(v)} cost_{op}(opcode(p)) \\ \qquad\qquad + \sum_i cost_{\text{SLP}}(operand_i(p)) \\ C_{insert} \cdot |v| + cost_{scalar}(v) \end{cases}$$

Figure 7: The SLP heuristic uses this recurrence to decide whether to produce a vector operand v directly via a vector pack or by vector insertions. $cost_{scalar}(v)$ is the total cost of producing values in v and their dependencies using only scalar instructions.

5 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 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 (Section 4.1). 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.

Pack Selection Heuristics. Pack selection is NP-hard because the more restricted version of pack selection for SIMD instructions is NP-hard [19]. In the rest of this section, we discuss two heuristics for pack selection. We first present a heuristic based on the bottom-up SLP algorithm [15, 29] (we will refer to this heuristic as the *SLP heuristic*). While the SLP heuristic is compile-time efficient, it has various drawbacks that we will discuss and address with a tractable search algorithm based on the SLP heuristic.

5.1 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 7 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.

Enumerating Seed Packs. In addition to store packs, VeGen enumerates a limited set of non-store seed packs, in two steps. First, it computes a pairwise affinity score for each pair of IR instructions according to the equation in Figure 8. Second, if a non-memory IR instruction i is used by some store instruction, then for all target vector length VL (i.e., 2, 4, 8, etc.), VeGen enumerates the top k packs—according to the affinity score—that is VL-wide and whose first lane is i. VeGen only enumerates instructions that feed into stores to limit the total number of seeds.

Limitations. The SLP heuristic assumes that each vector pack is the sole user of its operands. 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.

5.2 Improving the SLP Heuristic with Search

To address the SLP heuristic's limitations in handling shared values in the instruction DAG, we apply a limited form of lookahead search on top of the SLP heuristic. We first introduce a recurrence (Figure 9) for optimally solving the pack selection problem. We don't intend to optimally solve the recurrence, which contains exponentially many subproblems. VeGen instead uses beam search to navigate a limited subset of the search space, using $cost_{SLP}(.)$ (Figure 7) as a state evaluation function.

Optimal Pack Selection. Figure 9 shows the recurrence for computing the optimal cost of vectorizing a given basic block, $cost(V, S, F)$, in which we solve for the optimal set of packs on the instruction DAG bottom-up (uses before definitions), tracking the set of vectors (V) and scalar (S) operands we need to produce and the set of free instructions (F) we have yet to decide whether to vectorize. We decide how to produce the set of unresolved vector (and scalar) operands jointly in order to correctly determine the amortized cost of producing vector values—whether with packs or using swizzle instructions—with multiple uses.

The full cost of a basic block B is $cost(\{\}, \textit{live-outs}(B), I)$, where I is the set of instructions in B. In other words, we need to produce the live outputs of the original basic blocks as scalars (VeGen does not vectorize across basic blocks). VeGen treats stores as special cases. Stores are live at the end of a basic block, but, unlike other live outputs, vectorized stores do not incur extraction costs.

There are two ways to produce a value: as part of some vector pack or with a scalar instruction. Using a vector pack p recursively adds its operands to V and removes its results from V and S. To avoid circular dependencies in the final pack set, we only consider a pack (or a scalar instruction) once all of its users have been decided (i.e., not in F).

Finally, the packing problem is solved once the sets of vector and scalar operands become empty. Note that F need not be empty for a subproblem to be solved because some machine operations (e.g., multiply-accumulate and dot-product) replace multiple IR instructions and turn the intermediate instructions into dead code.

Beam Search. VeGen selects vector packs using beam search and guided by the SLP heuristic. Beam search is a form of greedy tree search, where the search algorithm considers a limited number of promising search candidates (instead of only the most promising one). In the case of pack selection, keeping track of this set of candidates allows the vectorizer to consider some vector packs that are costly according to the SLP heuristic but actually profitable.

When using beam search, VeGen's pack selection heuristic (implicitly) builds a search tree whose nodes correspond to the sub-problems in Figure 9 (where each sub-problem is represented by the tuple $\langle V, S, F\rangle$), and whose edges correspond to either adding a vector pack or fixing an instruction as a scalar. Each tree edge additionally has a transition cost taken to be the non-recursive terms in Figure 9. For instance, if an edge corresponds to adding a pack p, then the cost is

$$cost_{op}(opcode(p)) + cost_{extract}(p, S) + cost_{shuffle}(p, V)$$

To cut down the branching factor, VeGen only considers two types of packs: (1) the producer packs of V and (2) the set of seed packs it enumerates before the main search loop.

At each iteration of the search, VeGen tracks a set of k candidate tree nodes, expands the candidate nodes and aggregates their children, sorts the children in increasing order of the estimated cost, and takes the top k nodes to be candidates of the next iteration. The special case of the beam search with $k = 1$ is equivalent to the SLP heuristic.

Ideally, we would like to order (and prune) the set of candidate tree nodes based on the true optimal cost of following a tree node: $g + cost(V, S, F)$, where g is the aggregate cost leading to a given tree node and $cost(V, S, F)$ is the cost of optimally solving the tree node's sub-problem. However, computing the optimal cost is intractable, and we instead order the candidate tree nodes using the following formula:

$$g + cost(V, S, F) \approx g + \sum_{v \in V} cost_{SLP}(v) + \sum_{s \in S} cost_{scalar}(s)$$

$$affinity(v,w) = \begin{cases} -\alpha_{broadcast} & \text{if } v = w \\ -\alpha_{constant} & \text{if } v \text{ and } w \text{ are both constants} \\ -\alpha_{mismatch} & \text{if } v \text{ and } w \text{ not packable} \\ -\alpha_{mismatch} & \text{if } v \text{ and } w \text{ are loads separated by an unknown offset} \\ -\alpha_{jumbled} \cdot offset(v,w) & \text{if } v \text{ and } w \text{ are loads separated by a constant offset} \\ \alpha_{match} & \text{if } v \text{ and } w \text{ are contiguous loads} \\ \alpha_{match} + \sum_i affinity(operand_i(v), operand_i(w)) & \text{otherwise} \end{cases}$$

Figure 8: Recurrence for estimating the affinity score between two IR values v and w. α_* are positive parameters. Before pack selection, VeGen uses this function to enumerate a limited set of (non-store) seed vector packs so that the sums of affinities of adjacent lanes are maximized.

$$cost(V,S,F) = \min \begin{cases} cost(V_p,S_p,F_p) + cost_{op}(opcode(p)) + cost_{extract}(p,S) + cost_{shuffle}(p,V) & \text{if } |F \cap \bigcup_{i \in values(p)} (users(i))| = 0 \\ cost(V_i,S_i,F_i) + cost_{op}(opcode(i)) + cost_{insert}(i,V) & \text{if } i \text{ a scalar} \wedge |F \cap users(i)| = 0 \\ 0 & \text{if } |V| = 0 \wedge |S| = 0 \end{cases}$$

where

$$F_p = F \setminus values(p) \quad \text{- Free instructions left if use pack } p$$
$$V_p = \{v \in V \mid v \cap F_p \neq \emptyset\} \cup \bigcup_i operand_i(p) \quad \text{- Vectors to produce if use pack } p$$
$$S_p = S \cap F_p \quad \text{- Scalars to produce if use pack } p$$
$$F_i = F \setminus \{i\} \quad \text{- Free instructions left if fix } i \text{ as scalar}$$
$$V_i = \{v \in V \mid v \cap F_i \neq \emptyset\} \quad \text{- Vectors to produce if fix } i \text{ as scalar}$$
$$S_i = S \setminus F_i \cup \bigcup_j operand_j(i) \quad \text{- Scalars to produce if fix } i \text{ as scalar}$$
$$cost_{extract}(p,S) = C_{extract} \cdot |values(p) \cap S| \quad \text{- Cost of extracting elements of } p \text{ (if } p \text{ is not a store pack)}$$
$$cost_{shuffle}(p,V) = C_{shuffle} \cdot |\{v \in V \mid v \neq p \wedge |v \cap values(p)| > 0\}| \quad \text{- Cost of shuffling elements out of } p$$
$$cost_{insert}(i,V) = C_{insert} \cdot \sum_{v \in V} \langle \#\text{ of times } i \text{ occur in } v \rangle \quad \text{- Cost of inserting } i \text{ into vectors in } V$$

Figure 9: Optimal vector pack selection within a basic block. V and S are the sets of vector and scalar values we need to produce. F is the set of (free) IR instructions we have yet to decide whether (or how) to vectorize. C_* are the cost model parameters.

6 IMPLEMENTATION

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`).

[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

6.1 Target Instruction Specification

VeGen generates SMT formulas from the XML file that Intel uses to render the Intrinsics Guide [9], 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).

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 [10] 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.

6.2 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.[2] 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.

7 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 evaluated the two pack selection heuristics discussed in Section 5—the SLP heuristic and beam search—separately. 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. We use LLVM 10.0.0. In all cases, we invoke clang with `-O3 -ffast-math -march=native`.

7.1 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.

[2] https://software.intel.com/sites/landingpage/IntrinsicsGuide/files/perf2.js

(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 10: 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.

Figure 10 shows the test results. Both the SLP heuristic and beam search generate the same code, so we report one set of numbers. 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.

7.2 Optimizing Image and Signal Processing Kernels

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 x265We 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 additionally evaluated the effect of pattern canonicalization (Section 6). We ported the idct4 and idct8 kernels from x265's reference implementation. The rest are from FFmpeg.

We evaluated beam search (Section 5.1) with three beam-widths: 1, 64, and 128. Recall that a beam-width of one is effectively the SLP heuristic. To evaluate the effect of canonicalizing our generated patterns, we additionally evaluated a version of VeGen with pattern canonicalization disabled (and with a beam-width of 128).

Figure 11 shows the benchmarking results. Both the SLP heuristic and beam search outperform LLVM in all cases—except for the SLP heuristic ($k = 1$) on the idct4 benchmark; in the best case, beam search with $k = 128$ gets a speedup of 3× on idct4. Beam search improves on the SLP heuristic on fft4, idct4, sbc, and chroma.

Using a larger beam-width does not always lead to better results, as shown by the performance degradation of idct8 on AVX512 with a beam-width of 64. We traced the search process and discovered that the larger beam-width caused the search to include some costly search states that are ignored when $k = 1$. Their successor states—the number of which is larger than the beam-width—are misestimated by $cost_{SLP}$ (recall our discussion in Section 5.1 regarding $cost_{SLP}$'s limitations) to be more profitable than other candidate states and ultimately misdirected the overall search effort.

Canonicalizing the generated patterns using LLVM's own canonicalizer pays off on idct4, idct8, and chroma, all of which use saturation arithmetic. Running LLVM's canonicalizer on the generated patterns effectively synchronizes VeGen's pattern matchers with the canonicalization pipeline that LLVM runs before invoking VeGen's vectorizer.

Vectorizing idct4. We highlight some instructions that VeGen generated for the idct4 kernel (targeting AVX512-VNNI). Figure 12 shows the generated code, which is 3× faster than LLVM's code. VeGen uses the instructions `vphadd` (integer horizontal reduction), `vpmaddwd` (the motivating dot-product instruction), and `vpackssdw` (saturate 32-bit to 16-bit integers). Of note are the `vpunpackhdq` and `vpunpackldq` 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`. VeGen discovers this code sequence with beam search (i.e., $k \in \{64, 128\}$) but not with the SLP heuristic ($k = 1$).

7.3 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 13 shows the benchmarking results. VeGen found non-trivial vectorization schemes for three of the four kernels. The SLP heuristic and beam search generate identical code, so we only report a single set of numbers. 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* × 8 kernel (Figure 14), 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.

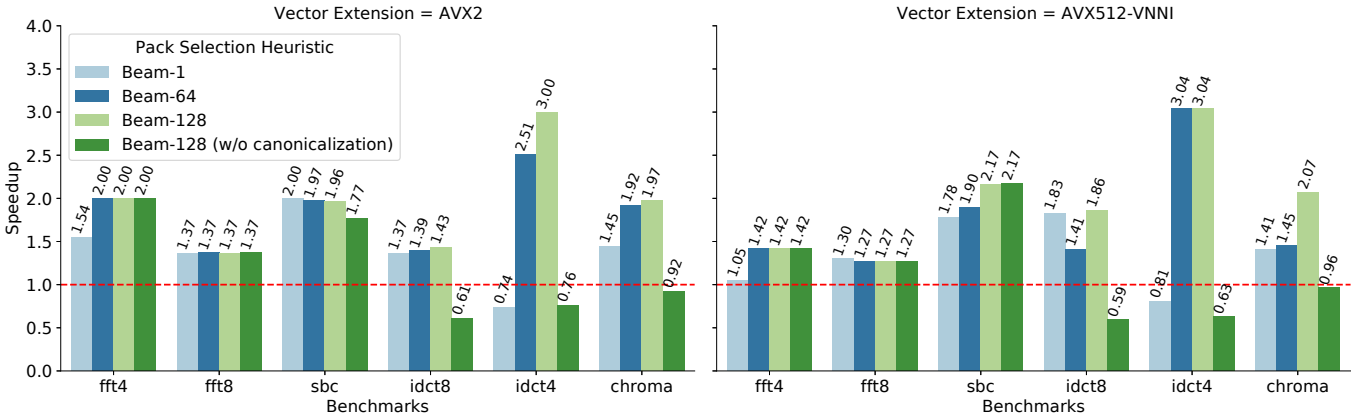


Figure 11: Speedup (over LLVM, higher is better) on kernels we selected from x265 (idct4 and idct8) and FFmpeg

```
...
vpermi2d     xmm3,     xmm7,  xmm5
vphaddd      xmm0,     xmm0,  xmm3
vpmaddwd     xmm1,     xmm2,  xmm1
...
vpackssdw    xmm1,     xmm1,  xmm2
vpunpckldq   xmm2,     xmm0,  xmm1
vmovdqu      [rsi],    xmm2
vpunpckhdq   xmm0,     xmm0,  xmm1
vmovdqu      [rsi+16], xmm0
```

Figure 12: Snippets of vector code generated by VeGen (using a beam-width of 128) for the idct4 kernel.

```
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 14: 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.

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

```
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 15: Complex multiplication kernel, generated by VeGen (Figure 15(a)) and LLVM (Figure 15(b)). VeGen's version is 1.27× faster.

The reference (naive) implementation of the *int32* × 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 6). The multiplications of the odd elements therefore map naturally to `vpmuldq`.

7.4 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 for complex arithmetic [2].

Figure 15 shows the complex multiplication kernel compiled by VeGen (both the SLP heuristic and beam search generated the same code) 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.

8 RELATED WORK

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 [20] proposed a technique for vectorizing interleaved memory accesses within a loop-based vectorizer. Eichenberger et al. proposed a technique for vectorizing misaligned memory accesses [11], and FlexVec [3] 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 [15]. 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 [17], Super-node SLP [26], TSLP [23], PSLP [24], VW-SLP [25], and ILP solver-aided goSLP [19].

There are domain-specific vectorizers that exploit architecture-specific vector instructions as well as application-specific patterns. The SPIRAL project [27] proposes several auto-vectorization schemes specific to DSP algorithms. More specifically, they propose a target-independent search-based vectorizing compiler targeting DSP algorithms [12] and show how to use the vector swizzle instructions supported by the AVX and Larrabee ISAs to implement the matrix transpositions found in FFTs [18]. Compared to SPIRAL and its extensions, VeGen is a general-purpose vectorizer and not designed to target any specific vector instruction sets.

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. [13] presented a survey on retargetable code generation. Cattell [7] investigated automatically generating code generators from machine descriptions. Ramsey and Fernández [28] proposed a specification language for describing instruction encoding. Buchwald et al. [6] synthesized instruction selection rules for 32-bit x86 integer instructions from their bit-vector specification.

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 [4] constructed a peephole superoptimizer by exhaustively enumerating short sequences of x86 instructions. Schkufza et al. [31] proposed a stochastic superoptimizer that trades completeness for scalability via a Markov Chain Monte Carlo sampler. Barthe et al. [5] 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. [22] build on previous work on enumerative [5], stochastic [31], and solver-based synthesis to scale up superoptimization. Sasnauskas et al. [30] described a superoptimizer for straight-line scalar LLVM IR.

9 CONCLUSIONS

We 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.

ACKNOWLEDGMENTS

We thank Jesse Michel, Ajay Brahmakshatriya, Teodoro Fields Collin, Logan Weber, and Alex Renda for reading early drafts and offering insightful feedback. We also thank our shepherd Guy Steele and the anonymous reviewers for guidance and valuable suggestions. Our work is supported by the Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; the Toyota Research Institute; the Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008923 and DESC0018121; the National Science Foundation under Grant No. CCF-1533753; and DARPA under Awards HR0011-18-3-0007 and HR0011-20-9-0017. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

REFERENCES

[1] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems* (1987).

[2] 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. 2004. A high-performance SIMD floating point unit for BlueGene/L: Architecture, compilation, and algorithm design. In *International Conference on Parallel Architecture and Compilation Techniques*.

[3] Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. 2016. FlexVec: Auto-vectorization for Irregular Loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[4] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[5] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *Symposium on Principles and Practice of Parallel Programming*.

[6] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an Instruction Selection Rule Library from Semantic Specifications. In *International Symposium on Code Generation and Optimization*.

[7] R. G. Cattell. 1980. Automatic Derivation of Code Generators from Machine Descriptions. *ACM Transaction on Programming Languages and Systems* (1980).

[8] 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. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Symposium on Operating Systems Design and Implementation*.

[9] Intel Corporation. 2012. *Intel Intrinsics Guide*. https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[11] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD Architectures with Alignment Constraints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[12] Franz Franchetti and Markus Püschel. 2002. A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms. In *International Parallel and Distributed Processing Symposium*.

[13] Mahadevan Ganapathi, Charles N. Fischer, and John L. Hennessy. 1982. Retargetable Compiler Code Generation. *Comput. Surveys* (1982).

[14] ARM Holdings. 2011. *Arm Architecture Reference Manual Armv8*. https://developer.arm.com/documentation/ddi0487/latest/

[15] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*.

[17] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. 2012. A Compiler Framework for Extracting Superword Level Parallelism. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[18] Daniel McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In *International Conference on Supercomputing*.

[19] Charith Mendis and Saman Amarasinghe. 2018. goSLP: Globally Optimized Superword Level Parallelism Framework. *Proceedings of the ACM on Programming Languages* (2018).

[20] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of Interleaved Data for SIMD. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[21] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *International Conference on Parallel Architectures and Compilation Techniques*.

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

[23] Vasileios Porpodas and Timothy M. Jones. 2015. Throttling Automatic Vectorization: When Less is More. In *Conference on Parallel Architecture and Compilation*.

[24] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP Automatic Vectorization. In *International Symposium on Code Generation and Optimization*.

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

[26] Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. 2019. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements. In *International Symposium on Code Generation and Optimization*.

[27] 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. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* (2005).

[28] Norman Ramsey and Mary F. Fernández. 1997. Specifying Representations of Machine Instructions. *ACM Transaction on Programming Languages and Systems* (1997).

[29] Ira Rosen, Dorit Nuzman, and Ayal Zaks. 2007. Loop-aware SLP in GCC. In *GCC Developers Summit*.

[30] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *arXiv preprint arXiv:1711.04422* (2017).

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