All You Need Is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP

Yishen Chen
Massachusetts Institute of Technology
Cambridge, MA, USA
ychen306@mit.edu

Charith Mendis
University of Illinois at Urbana-Chamaign
Urbana, IL, USA
charithm@illinois.edu

Saman Amarasinghe
Massachusetts Institute of Technology
Cambridge, MA, USA
saman@csail.mit.edu

Abstract

Superword-level parallelism (SLP) vectorization is a proven technique for vectorizing straight-line code. It works by replacing independent, isomorphic instructions with equivalent vector instructions. Larsen and Amarasinghe originally proposed using SLP vectorization (together with loop unrolling) as a simpler, more flexible alternative to traditional loop vectorization. However, this vision of replacing traditional loop vectorization has not been realized because SLP vectorization cannot directly reason with control flow.

In this work, we introduce SuperVectorization, a new vectorization framework that generalizes SLP vectorization to uncover parallelism that spans different basic blocks and loop nests. With the capability to systematically vectorize instructions across control-flow regions such as basic blocks and loops, our framework simultaneously subsumes the roles of inner-loop, outer-loop, and straight-line vectorizer while retaining the flexibility of SLP vectorization (e.g., partial vectorization).

Our evaluation shows that a single instance of our vectorizer is competitive with and, in many cases, significantly better than LLVM's vectorization pipeline, which includes both loop and SLP vectorizers. For example, on an unoptimized, sequential volume renderer from Pharr and Mark, our vectorizer gains a 3.28× speedup, whereas none of the production compilers that we tested vectorizes to its complex control-flow constructs.

CCS Concepts: • **Computing methodologies** → Vector / streaming algorithms; • **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Single instruction, multiple data**.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523701

ACM Reference Format:
Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All You Need Is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523701

1 Introduction

Allen and Kennedy [4] pioneered loop vectorization with their seminal work in the 1980s. The basic idea of loop vectorization is to widen instructions in the loop body from scalar to vector instructions. This technique was originally motivated by the long vector architectures that were common at the time (e.g., the Cray machines).

In the early 2000s, Larsen and Amarasinghe [14] developed superword-level parallelism (SLP) vectorization to target the multimedia extensions that were emerging. SLP vectorization works by searching for independent, isomorphic instructions in straight-line code and replacing them with vector instructions. Because SLP vectorization targets straight-line code, it requires much simpler dependence analyses (as opposed to the more complex loop-dependence analyses required by loop vectorization).

Larsen and Amarasinghe initially intended SLP vectorization as a simpler and more flexible alternative to traditional loop vectorization. Indeed, when combined with loop unrolling, SLP vectorization can also exploit loop-level parallelism [14, 29]. However, because it was not designed to directly reason with control flow, SLP vectorization cannot exploit parallelism that spans control-flow regions such as different basic blocks or loops. Meanwhile, multiple works have extended traditional loop vectorization to handle complicated control-flow constructs such as divergent branches and outer loops [4, 12, 20]. Today, with neither vectorization technique being superior to the other across all applications, production compilers such as GCC and LLVM implement both loop and SLP vectorizers [1, 2].

In this work, we introduce SuperVectorization, a new vectorization framework that generalizes SLP vectorization to pack independent instructions across different basic blocks and loop nests—such loops can be imperfectly nested and have different trip counts or multiple side exits. Because our

framework can pack instructions across control-flow regions such as basic blocks and loops, it simultaneously subsumes the roles of inner-loop, outer-loop, and straight-line vectorizers while retaining the flexibility of SLP vectorization (e.g., partial vectorization). For example, outer-loop vectorization in our framework involves unrolling an outer loop and then packing the instructions from the inner loops duplicated by the unroller. Our evaluation shows that a single instance of our vectorizer is competitive with and, in many cases, significantly better than LLVM's vectorization pipeline, which includes both loop and SLP vectorizers.

Central to our framework is *Predicated SSA*, a new intermediate representation (IR) that we developed to simplify the pervasive inter-basic block code motion required to target SLP that spans different basic blocks or loops. Rather than using a control-flow graph (CFG), this IR represents the input program as a flat list of instructions and loops and tracks the control dependence of each instruction explicitly with a boolean formula that expresses whether the instruction should execute. We refer to such formulas as *control predicates*.[1] Using Predicated SSA, inter-basic block code motion is straightforward: We just move an instruction (or loop) together with its control predicate, and no information is lost because we can recover control flow from equivalent control predicates.

We make the following contributions in this paper:

- We introduce Predicated SSA, an IR that we developed to simplifying the pervasive code motion required to uncover arbitrary SLP that spans different basic blocks and loops.
- We present SuperVectorization, our generalization of SLP vectorization to vectorize (pack) instructions from different basic blocks and loops. Together with loop unrolling, SuperVectorization effectively subsumes the roles of inner-loop, outer-loop, and straight-line vectorizers.
- We show that our single prototype implementation, written from scratch, is competitive with and, in many cases, significantly outperforms LLVM's vectorization pipeline, which includes both a loop and SLP vectorizer.

2 Background

Our goal is to develop a vectorization strategy that retains the simplicity and flexibility of SLP vectorization while generalizing it to systematically handle control flow. To this end, we review the background on loop and SLP vectorization and motivate the technical challenge of extending SLP vectorization to handle control flow.

[1]Predicated SSA does not perform predicated execution and executes instructions conditionally similar to a traditional IR with CFG.

Loop Vectorization. Loop vectorization targets loop-level parallelism by mapping successive loop iterations to successive vector lanes. While traditional loop vectorization focuses on inner loops by executing the inner-loop iterations in parallel, outer-loop vectorization instead transforms the program to execute the outer-loop iterations in parallel [20].

SLP Vectorization. Larsen and Amarasinghe [14] proposed superword-level parallelism (SLP) as a model of the short-vector parallelism implemented by modern vector extensions. SLP vectorization works in three steps. First, the vectorizer runs an SLP packing heuristic to select groups of independent, isomorphic instructions to pack together. Second, the vectorizer reorders the instructions so that the dependences of any group of packed instructions appear before the group. Finally, the vectorizer replaces each group of packed instructions with an equivalent vector instruction.

SLP vectorization is relatively simpler and more flexible. SLP vectorization is simpler because it does not target parallelism that spans different loop iterations, thus not requiring loop dependence analysis to reason with loop-carried dependence. SLP vectorization can nonetheless exploit inner-loop parallelism using loop unrolling [14, 29].

SLP vectorization is also more flexible at exploiting the type of short-vector parallelism implemented by existing multimedia architecture extensions. The SLP framework defines a search space that specifies which instructions can be packed together—subject to dependences and the capability of the target architecture, leaving the SLP heuristic free to vectorize however it sees fit [16, 17, 24–27, 30]. In contrast, loop vectorization mechanically transforms each loop instruction to a wider vector instruction. Adapting loop vectorization to handle cases that deviate from this assumption requires research in itself.

Consider the loop in Figure 1a as an example of SLP vectorization's flexibility. A naive loop vectorizer would emit instructions similar to those in Figure 1b, vectorizing the even and odd accesses separately, losing performance because the memory accesses are not contiguous and are less efficient. To address this inefficiency, Nuzman et al. [19] extended loop vectorization to support interleaving and generate instructions similar to the ones in Figure 1c. In contrast, an SLP vectorizer (with unrolling) vectorizes such loops effectively without any extensions because SLP is loop agnostic.

2.1 Handling Control Flow in SLP Vectorization

SLP vectorization has so far been limited to target parallelism within individual basic blocks, despite SLP captures a more general form of parallelism. Consider, for example, the sequential program in Figure 2, which performs two independent linear searches over the same array for two different elements. No existing vectorizers can exploit the available (superword-level) parallelism among the loops.

```
for (i = 0; i < n; i+=2) {
  a[i]   = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

(a) An example loop with interleaved accesses

```
for (i = 0; i < n; i+=8) {
  t  = b[i:i+8];
  t2 = c[i:i+8];
  b_even = shfl-even t;
  c_even = shfl-even t2;
  e = b_even + c_even;
  b_odd = shfl-odd t;
  c_odd = shfl-odd t2;
  o = b_odd + c_odd;
  a[i:i+8] =
    interleave e, o;
}
```

(b) Result of applying loop vectorization naively

```
for (i = 0; i < n; i+=4) {
  /*
   After unrolling,
   before SLP vectorization:
   a[i]   = b[i] + c[i];
   a[i+1] = b[i+1] + c[i+1];
   a[i+2] = b[i+2] + c[i+2];
   a[i+3] = b[i+3] + c[i+3];

   After SLP vectorization:
  */
  a[i:i+4] = b[i:i+4]
             + c[i:i+4];
}
```

(c) Result of interleave-aware loop vectorization. SLP vectorization can discover the same vectorization scheme natively.

Figure 1. An example of vectorizing a loop with interleaved accesses. Nuzman et al. [19] proposed an extension to loop vectorization to vectorize such an example. SLP vectorization can discover the equivalent vectorization scheme from the first principle.

Code Motion. The key challenge to targeting SLP that spans arbitrary control-flow regions is inter-basic block code motion. Exploiting the SLP available in Figure 2, for example, requires moving instructions from one loop to another. Code motion in a traditional IR with CFG is challenging because it requires restructuring the CFG and the complexity of doing so increases with the complexity of control dependences. Consider the task of hoisting a store to an earlier location in the CFG as an example. Because relaxing the condition under which the store executes may lead to segfault, we need to preserve the condition of the store. Consequently, hoisting a store in general entails 1) finding a basic block that is *control-flow equivalent* to the original basic block—and, if such a basic block does not exist, restructuring the CFG to create the block—and 2) recursively hoisting the dependences of the store to preserve data dependences, which may require further changes to the CFG.

If-conversion. If-conversion [5] converts control dependences to data dependences by transforming the program to unconditionally execute both sides of every conditional branch and replaces control-flow joins (i.e., the ϕ-nodes in SSA) with select instructions (which chooses two alternative values based on a boolean condition, similar to the ternary operator in C/C++). Because if-conversion unconditionally executes instructions that may not otherwise execute, naively applying if-conversion can decrease performance [18, 31]. Nonetheless, if-conversion is necessary for vectorizing divergent control flow (i.e., branches with different conditions).

```
int haystack[];
int needles[2];
int needle_idxs[2];

for (int i = 0; i < n; i++)
  if (haystack[i] != needles[0]) {
    needle_idxs[0] = i;
    break;
  }

for (int i = 0; i < n; i++)
  if (haystack[i] != needles[1]) {
    needle_idxs[1] = i;
    break;
  }
```

Figure 2. Running example of a sequential C code that we will vectorize. The program performs two linear searches over the same array twice to find the indices of two different elements (`needles[0]` and `needles[1]`). The two loops are independent, and we will exploit the parallelism between the two loops by packing their instructions together.

Prior Work. Shin et al. [31] proposed handling control flow in SLP vectorization with if-conversion. After applying if-conversion to eliminate all (forward) control flow, they use SLP vectorization as a black box to vectorize instructions within the relatively few large basic blocks. However, without directly addressing the code motion problem, their approach cannot target parallelism that spans different loops nests (e.g., the program in Figure 2).

Our Approach. The key to our approach is Predicated SSA, an IR that we developed to simplify the pervasive code motion required to target SLP that spans different basic blocks (and loops). Instead of using a CFG, which complicates code motion, Predicated SSA represents the input program as a flat list of instructions and loops while tracking the control dependences of each instruction separately with a symbolic boolean formula that expresses whether the instruction should execute. Code motion in Predicated SSA only entails reordering an instruction and its dependences as one would for intra-basic block code motion; such reordering in Predicated SSA is safe because we always reorder an instruction together with its control predicate, and no information is lost. Predicated SSA does not commit to any code layout decision (e.g., assigning an instruction to a particular basic block). Only after vectorization and deciding an overall execution schedule for the instructions, do we revert back to a traditional IR with CFG.

2.2 Vector Instruction Sets

Our approach targets vector instruction sets with fixed vector width. This covers Intel's vector extensions (SSE, AVX2, and AVX-512) and Arm's Neon instruction set. In the rest of this section, we discuss the hardware features that our approach either relies on or benefits from.

Selection/Blending. Depending on their control dependences, our approach may independent control-flow joins into data-flow joins in the form of *vector selects* (*blending*). Intel's vector extensions support this operation with their `blend` family of instructions (e.g., `blendvps` for blending two packed vectors of single-precision floats). Arm's Neon extension supports this operation with the `bsl` instruction (bitwise select).

Gather/Scatter. Traditional vector instruction sets only support loading/storing contiguous memory locations, and vector gather/scatter allows loading/storing a vector of arbitrary pointers. Although not a required feature for our technique, vector gathers improve performance for applications with irregular memory access patterns. When the target hardware does not support vector gather/scatter, our approach emits equivalent (but less efficient) scalar loads/stores and vector insertion instructions.

Predication. While vectorizing conditional load/store instructions (with distinct conditions), we require the ability to dynamically suppress the effect of vector lanes whose conditions evaluate to false. If a target architecture does not support predicated vector load/store instructions, our approach will emit sequential scalar stores with branches instead. AVX-512 supports predicated (masked) loads and stores.

2.3 Definitions

In this section, we review the compiler terminologies that we will use in Section 3.

Let b_1 and b_2 be two basic blocks in a given CFG. b_1 *dominates* b_2 if every path from the entry basic block to b_2 must go through b_1. Relatedly, b_1 *post-dominates* b_2 if every path from b_1 to the exit node goes through b_2.

The *post-dominance frontier* of a basic block b is the set of all basic blocks b' such that b' is not post-dominated by b but has a predecessor post-dominated by b.

Let b be a basic block. The set of basic blocks that b is *control-dependent* on is exactly the *post-dominance frontier* of b [9].

Gated SSA. Our vectorizer uses Predicated SSA, an IR that borrows many ingredients from gated SSA [21]. Gated SSA extends SSA [9] by differentiating its ϕ-nodes into two categories.

- For ϕ-nodes placed at loop headers, gated SSA renames them as μ-nodes (which have the same semantics as the loop-header ϕ-nodes in SSA).
- For each n-nary ϕ-nodes used for forward control-flow join, gated SSA replaces it with $n-1$ gating operators called γ-nodes. A γ node selects two alternative incoming values (of the original ϕ-node) depending on a branch condition (similar to C's ternary operator).

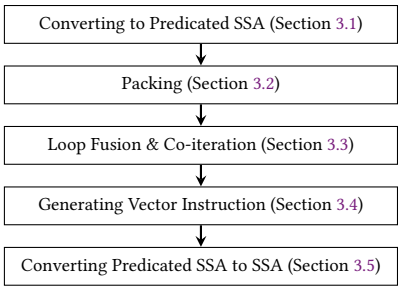

Figure 3. Vectorization workflow in our framework

3 SuperVectorization

SuperVectorization is our generalization of SLP vectorization to pack arbitrary, independent instructions from different basic blocks and loops. Such loops can be imperfectly nested, have different trip counts, or have multiple side exits.

Workflow. Figure 3 shows our workflow. We begin by transforming an input sequential program from SSA to Predicated SSA, an IR that we developed to simplify the pervasive code motion that is required for vectorizing instructions across basic blocks and loops. Once the input is in Predicated SSA, we find profitable vector packs as one does in traditional SLP vectorization (Section 3.2). The precise algorithm for finding the vector packs is orthogonal to our work; in our implementation, we use the bottom-up SLP algorithm [30]. Once we have a set of vector packs, we identify loops that share common vector packs and transform the loops so that all instructions from any given pack reside in a single new loop (Section 3.3). Having ensured that all instructions from the same packs will be in the same loops, we replace the packed instructions with equivalent vector instructions (Section 3.4). So far, we have done everything in Predicated SSA (i.e., packing, fusion, and vectorization). Once we replace scalar instructions with vector instructions, we finish by lowering Predicated SSA back to a traditional IR with CFG (Section 3.5).

We will use the program in Figure 2 as a running example. This example contains two disjoint, independent loops. We will pack instructions from the loops together into vector instructions.

3.1 Predicated SSA

Figure 4 shows the syntax of Predicated SSA, and Figure 5 shows the first loop in the running example (Figure 2) translated to Predicated SSA.

Control Predicate. Predicated SSA forgoes the use of CFG entirely and instead represents the input program as a flat list of instructions and loops while tracking the control dependence of individual instructions and loops with first-class constructs that we refer to as *control predicates*. Figure 6 shows the definition of control predicates. A control

$$
\begin{aligned}
fn &::= item_1 : p_1, \ldots, item_n : p_n \\
p &::= \textbf{control predicate} \\
item &::= \textbf{instruction} \mid loop \\
loop &::= \textbf{with } v_1 = \mu_1, \ldots, v_m = \mu_m \textbf{ do} \\
&\qquad item_1 : p_1, \ldots, item_n : p_n \\
&\quad \textbf{while } p_{cont} \\
\mu &::= \textbf{mu}(v_{init}, v_{rec}) \\
\phi &::= \textbf{phi}(v_1 : p_1, \ldots, v_n : p_n) \\
v &::= \mu \mid \textbf{instructions} \mid \textbf{constant} \mid \textbf{argument}
\end{aligned}
$$

Figure 4. Definition of Predicated SSA. The root of the grammar is *fn*, representing a given input function. Similar to gated SSA, Predicated SSA represents IR values defined recursively in loops with μ-nodes, where v_{init} represents the value flowing from the loop pre-header, and v_{rec} represents the value flowing from the back edge.

```
with i = mu(0, i') do
  t          = load haystack[i]  : true
  needle     = load needles[0]   : true
  found      = cmp eq, t, needle : true
  i'         = add i, 1          : true
  not_found  = not found         : true
  lt_n       = cmp lt i', n      : true
while not_found and lt_n         : true
store i, needle_idxs[0]          : found
```

Figure 5. The first loop in Figure 2 translated to Predicated SSA. The *true* control predicates indicate that the items execute unconditionally. For items inside the loop, the predicate indicates that they should execute as long as the loop is not terminated. Observant readers may notice that the load of `needles[0]` is loop-invariant and can be hoisted; we do not hoist the load here because loop-invariant code motion is not part of the IR conversion process.

predicate is a symbolic boolean expression that indicates whether an instruction (or loop) should execute. Predicated SSA represents each function as a list of *items*. Each item is either an instruction or a loop, and each loop also contains a list of *items*. Each item has its own *control predicate* that indicates whether that item should execute. With this representation, code motion is straightforward: We can reorder an instruction (or loop) together with its control predicate without losing information.

Although our use of control predicates is inspired by predicated execution [22], Predicated SSA does not perform predicated execution. In Predicated SSA, instructions and loops execute conditionally, depending on their control predicates. In contrast, in predicated executions, predicated instructions execute unconditionally but with their side-effects suppressed conditionally.

$$
\begin{aligned}
c &::= \textbf{branch-conditions } \textit{(IR values used for branching)} \\
p &::= \textbf{true} \mid c \mid \overline{c} \mid p_1 \wedge p_2 \mid p_1 \vee p_2
\end{aligned}
$$

Figure 6. Definition of Control Predicates.

Similar to gated SSA, Predicated SSA uses μ-nodes to represent control-flow joins placed at loop headers. For forward control-flow joins, Predicated SSA uses *gated* ϕ-nodes (rather than the γ-nodes from gated SSA); our gated ϕ-nodes are similar to the ϕ nodes in classical SSA, except that the incoming basic block labels are replaced with *control predicates*. The gated ϕ-nodes provide us with a simple way to vectorize ϕ-nodes. While packing multiple ϕ-nodes, if the incoming values of the ϕ-nodes have the same predicates, then we retain control flow and replace them with a single vector ϕ; if the incoming values of the ϕ-nodes have different predicates, then we lower them into vector selects, emitting the select conditions according to the control predicates.

Converting from SSA to Predicated SSA. We begin by converting the input IR into a canonical form. We assume that the CFG of the input program is reducible and that we can transform each loop into a canonical form, with each loop having:

- a single incoming edge
- a single back edge
- a dedicated loop header
- a dedicated loop pre-header (i.e., the unique predecessor of the loop pre-header)
- a dedicated loop latch (i.e., the unique source of the back edge)

We also normalize each loop in a rotated form where all loops execute at least once (i.e., similar to do-while loops in C).

Once we transform the input into this canonical form, we convert it into Predicated SSA by visiting the input loops recursively. We convert the forward ϕ-nodes into gated ϕ-nodes by replacing the incoming basic block labels with the control predicates of the basic blocks. (We will discuss how to compute control predicates next.) Because all loops are in the canonical form that we just discussed, we translate the ϕ-nodes in the loop headers into μ-nodes by labeling the values flowing from the loop pre-headers as the initial values (v_{init} in Figure 4) and the values flowing from the loop latches as the recursive values (v_{rec}).

Computing Control Predicates. One tempting approach to computing the control predicates would be to perform symbolic execution over the branch instructions of the input program, namely, emitting conjunction when taking a branch and emitting disjunction when encountering a control-flow join. However, this approach can introduce unnecessarily complicated predicates that require further simplification.

```
entry:
  br c, if_true, if_false
if_true:
  br join
if_false:
  br join
join:
  ...
```

Figure 7. An example of *control-flow equivalent* basic blocks. The blocks `entry` and `join` are control-flow equivalent because `entry` dominates `join` and because `join` post-dominates `entry`. Consequently, the control predicate of the `join` block is the same as `entry`'s predicate (*true*).

Consider the example in Figure 7 where there is a branch on the condition c. When we join both sides of the branch, we would get the predicate $c \vee \overline{c}$, but we want the more concise predicate *true*. To achieve this aim, we use an algorithm that uses control dependence to directly produce simplified predicates.

Figure 8 shows the algorithm for computing the control predicate of a given basic block. The control predicate of an instruction is simply the predicate of its basic block. The idea of the algorithm follows from the observation that whether a basic block b executes depends exactly on two conditions: (1) whether any of its control-dependent basic blocks b' executes, and (2) whether b' takes the branch that leads to b. For condition (1), we simply compute the post-dominance frontier of b, which is equivalent to the control-dependent blocks of b [9]. For condition (2), we observe that because b' is in the post-dominance frontier of b, the branch that leads from b' to b must be the branch $b' \rightarrow b''$, such that b'' is a successor of b' and b post-dominates b'' (i.e., any control flow leaving b'' must end at b); such successor b'' must exist uniquely because b' (the predecessor of b'') is in the post-dominance frontier of b. For the example program in Figure 7, our algorithm directly computes a *true* predicate for the `join` block because it post-dominates all preceding blocks and, therefore, has no control dependences.

When computing control predicates, we use the concept of control-flow equivalence. *Control-flow equivalence* is a relation over basic blocks that holds when the execution of one basic block implies that of the other and vice versa. This relation can be difficult to establish for arbitrary pairs of basic blocks. Instead, we will use an incomplete heuristic and say two basic blocks b_1 and b_2 are control-flow equivalent if b_1 dominates b_2 and b_2 post-dominates b_1. Figure 7 shows an example of control-flow equivalent basic blocks.

We ignore backward control flows when we compute the control predicates, and we assign a *true* predicate to basic blocks that are known to be *control-flow equivalent* to the loop header. Consequently, if a predicate is used within a loop, the truthness of the predicate is conditioned on whether the execution reaches a given loop iteration. We track the

$$cp_{\text{block}}(b) = \begin{cases} true & \text{If } \textit{ctrl-flow-equivalent}(b, \textit{header}(b)) \\ \bigvee_{b' \in PDF(b)} cp_{\text{edge}}(b', \textit{succ-pdom}(b', b)) & \text{Otherwise} \end{cases}$$

$$cp_{\text{edge}}(b_1, b_2) = \begin{cases} true & \text{If } b_1 \rightarrow b_2 \text{ is back edge} \\ cp_{\text{block}}(b_1) & \text{If } \textit{preds}(b_2) = \{b_1\} \\ cp_{\text{block}}(\textit{preheader}(b_1)) \wedge cp_{\text{block}}(b_1) \wedge \textit{cond}(b_1, b_2) & \text{If } b_1 \rightarrow b_2 \text{ loop-exiting} \\ cp_{\text{block}}(b_1) \wedge \textit{cond}(b_1, b_2) & \text{Otherwise} \end{cases}$$

Figure 8. Algorithm for computing the control predicate of a basic block b, where $PDF(b)$ is the post-dominance frontier of b; $\textit{succ-pdom}(b', b)$ is the successor of b' that is post-dominated by b (there uniquely exists such a basic block because b' is in the post-dominance frontier of b); and $\textit{cond}(b_1, b_2)$ denotes the branch condition for when the control-flow edge $b_1 \rightarrow b_2$ is taken.

continue predicate (i.e., whether the back edge is taken) as a special case (p_{cont} in Figure 4) in Predicated SSA. We compute the *continue predicate* of a loop as the conjunction of the control predicate of the loop latch (i.e., whether we reach the latch) and the branch condition of the back edge.

We compute the control predicate of a loop exit—a loop may have multiple exits—with the intuition that the execution only reaches the exit all of the following is true: (1) if we enter the loop in the first place, (2) if we reach the predecessor of the exit, and finally (3) if the conditional branch preceding the exit is taken. Therefore, the control predicate of a loop exit is the conjunction of (1) the control predicate of the pre-header of the loop from which it is exiting, (2) the control predicate of the predecessor of the basic block (which is inside the loop), and (3) the branch condition of the edge leading to the loop exit.

3.2 Vector Packing

Once we convert the input program into Predicated SSA, we run a packing heuristic to decide which instructions should be packed together. Deciding which instructions to pack is orthogonal to our work. SuperVectorization is compatible with arbitrary packing heuristics as long as program dependence is not violated [16, 17, 24–27, 30]. For this paper, we adapt the bottom-up SLP heuristic [30] for its simplicity. Both GCC and Clang use variants of the same heuristic.

The bottom-up SLP heuristic works as follows. First, it identifies groups of instructions that are known to be vectorizable, which are termed *seed instructions*; common seeds are stores to contiguous memory locations and reductions. Next, the algorithm attempts to find more vectorizable instructions by traversing the use-def chains of the seed instructions packing the operands of the instructions that it encounters along

the way. This process stops when it encounters operands that cannot be directly produced by vector instructions (e.g., a group of instructions with different opcodes). At this point, to ensure that the final vectorization decision does not violate program dependence, the heuristic checks that the set of selected vector packs do not have any circular dependences. Finally, the algorithm uses a cost model to estimate the cost benefit of vectorizing the candidate instructions, with the benefit coming from replacing scalar instructions with more efficient vector instructions, and the cost coming from using vector data-movement instructions (e.g., vector shuffle) to produce the operands that are not vectorized.

Once we have decided which instructions to pack, we lower the selected packs into vector instructions as follows. First, if we are packing instructions from different loops, we transform the loops so that the instructions of any given vector pack belong to the same loops. Then, we lower the packed instructions into vector instructions (e.g., a pack of additions becomes a single vector addition). At this point, we have done everything in Predicated SSA. Finally, we lower the IR back to a traditional IR with CFG (e.g., LLVM IR).

A Note on Pattern Matching. Many SLP vectorizers use pattern matching. For example, LLVM's SLP vectorizer employs pattern matching to (among other use cases) identify and vectorize reductions. Doing pattern matching in Predicated SSA is the same as that in a traditional SSA-based IR (e.g., LLVM) because the former maintains the same use-def information that SSA tracks.

3.3 Loop Fusion and Co-iteration

Our framework supports packing instructions from different loop nests. This capability allows, for example, vectorizing the two independent search loops in the running example from Figure 2. More importantly, when coupled with a vectorization-aware loop unroller to unroll outer loops, packing instructions across loops is equivalent to outer-loop vectorization. This approach is similar to how Nuzman and Zaks [20] applied unroll-and-jam to achieve outer-loop vectorization. One major difference here is that in our framework, a packing heuristic need not be aware of the loop structures.

While packing instructions from different loops, we must generate a new loop so that the instructions execute in the same loop. To accomplish this, we either fuse or *co-iterate* the loops, depending on whether the loops are control-flow equivalent (i.e., have the same control predicates) and whether they have the same trip counts.

Loop Fusion. We apply loop fusion if we can prove that loops are independent,[2] they have the same iterations, they execute under the same condition (i.e., they have identical control predicates), and their parent loops can also be fused Fusing loops in Predicated SSA requires only creating a new loop whose μ-nodes and loop items are the concatenations of those of the original loops.

Predicated SSA also increases the applicability of loop fusion, despite the latter being a well-studied optimization. Traditionally, compilers only fuse loops that are adjacent [7, 13]. Predicated SSA removes this restriction by making it straightforward to hoist (or sink) any intervening instructions and control flow.

Loop Co-iteration. When we cannot fuse some loops because they execute under different control predicates or because they have different trip counts, we apply a transformation that we call *loop co-iteration*. The intuition behind co-iteration is to interleave the iterations of multiple loops in a single new loop while preserving the execution condition of the original instructions. It is safe to co-iterate when the loops are independent and when their parent loops are also safe to co-iterate. Although we use co-iteration to enable packing instructions from different loops, co-iteration (similar to fusion) is a standalone scalar optimization that is applicable independent of vectorization.

Figure 9 shows the algorithm for co-iterating multiple loops. Similar to loop fusion, co-iteration entails moving the μ-nodes and the loop items of the original loops into a new loop and the following additional steps:

- For each loop, introduce a boolean μ-node to track whether the loop still has remaining iterations (lines 19-27).
- For each loop item, strengthen its control predicate so that the item only executes when its original loop is active and its original predicate evaluates to true (lines 44-48).
- For each loop live-out (i.e., an instruction with users outside of the loop), introduce a new μ-node to keep the value alive across the iterations even when the loop is no longer active (lines 51-65).
- Set the continue predicate of the new loop to be the disjunction of the active conditions of the co-iterating loops.

Figure 10 shows the result of co-iterating the loops in the running example. In the example, we use the booleans `active` and `active2` to indicate whether the original loops are still active. We also insert the new μ-nodes `i_out` and `i2_out` to keep the values `i` and `i2` alive across the iterations where their original loops would have exited already.

[2]While this is not necessary for fusion in general, it simplifies our implementation.

```
def coiterate(loops: the loop to coiterate):
  if identical(loops):
    return

  parents = {l.parent for l in loops}
  parent = coiterate(parents)
  mus = concat(l.mus for l in loops)
  items = concat(l.items for l in loops)
  # A map of IR values that tells
  # if a given loop is active
  active_mus = {}
  # Create a boolean Mu node to indicate
  # whether a co-iterating loop is active
  for l in loops:
    active = Mu()
    mus.append(active)
    active_mus[l] = active

    pred = parent.control_pred_of(l)
    active_init = GatedPhi(
      {pred: true, negate(pred): false})
    active_next = GatedPhi({
      And(active, l.cont_pred): true,
      negate(active): false,
      negate(l.cont_pred): false})
    active.set_init(active_init)
    active.set_rec(active_next)

    items.append(active_next)
    parent.items.append(active_init)
    parent.remove_loop(l)

  # We continue iterating
  # as long as there is an active loop
  cont_pred = Or(
    [And(active_mus[l], l.cont_pred) for l in loops]
  )
  new_loop = Loop(
    mus=mus, items=items,
    cont_pred=Or(cont_pred, parent=parent)

  # Strengthen the predicates so that
  # an item only executes when the parent is active
  for l in loops:
    for item in l.items:
      new_loop.set_control_pred(item,
        And(active_mus[l],
            l.control_pred_of(item)))

  # Introduce mu nodes to guard loop live-outs
  for l in loops:
    active = active_mus[l]
    for x in l.live_outs():
      x_out = Mu()
      x_pred = l.control_pred_of(x)
      x_out_next = GatedPhi({
          And(active, x_pred): x,
          negate(active): x_out,
          negate(x_pred): x_out
        })
      x_out.set_init(undef)
      x_out.set_rec(x_out_next)
      for user in x.users():
        if user not in new_loop.items:
          user.replaceUsesOfWith(x, x_out)

  return new_loop
```

Figure 9. Algorithm to co-iterate multiple loops

```
with i = mu(0, i'),
     i2 = mu(0, i2'),
     i_out = mu(undef, i_out'),
     i2_out = mu(undef, i2_out'),
     found_out = mu(undef, found_out'),
     found2_out = mu(undef, found2_out'),
     active = mu(true, active'),
     active2 = mu(true, active2')
do
  t         = load haystack[i]      : active
  needle    = load needles[0]       : active
  found     = cmp eq, t, needle     : active
  not_found = not found             : active
  i'        = add i, 1              : active
  lt_n      = cmp lt, i', n         : active

  t2         = load haystack[i2]       : active2
  needle2    = load needles[1]         : active2
  found2     = cmp eq, t2, needle2     : active2
  not_found2 = not found2              : active2
  i2'        = add i2, 1               : active2
  lt_n2      = cmp lt i2', n           : active2

  active'  = phi(
    active and not_found and lt_n    : true,
    _                                : false)     : true
  active2' = phi(
    active2 and not_found2 and lt_n2 : true,
    _                                : false)     : true
  found_out'  = phi(active: found, _: found_out)  : true
  found2_out' = phi(active2: found2,_: found2_out): true
  i_out'      = phi(active: i', _: i_out)         : true
  i2_out'     = phi(active2: i2', _: i2_out)      : true

  cont = or active', active2' : true
while cont : true
store i_out, needle_idxs[0]  : found_out
store i2_out, needle_idxs[1] : found2_out
```

Figure 10. The two loops in Figure 2 after co-iteration.

3.4 Generating Vector Instructions from Packs

After selecting a profitable set of vector packs, we generate an optimized vector program, according to the packing decision, as follows:

1. Schedule (reorder) the instructions and loops to satisfy the dependences of the packed instructions.
2. Replace vector packs with vector instructions.
3. Assign control predicates to the vector instructions.

Scheduling. We first schedule the instructions and loops so that for any given vector pack, the dependences of the packed instructions precede the packed instructions (while also satisfying the dependences of the other scalar instructions). We find such schedules by doing topological sort on the dependence graph formed by the instructions and loops.

Scheduling in our framework is relatively straightforward because we are using Predicated SSA, which enables us to freely reorder instructions and loops. Scheduling instructions and loops with arbitrary control dependences directly on an IR with CFG requires multiple coordinated changes to the CFG and is more complicated.

```
with i = vmu({0, 0}, i'),
     i_out = vmu({undef, undef}, i_out'),
     found_out = vmu({undef, undef}, found_out'),
     active = vmu({true, true}, active'),
do
  addrs      = vadd haystack, i                : true
  t          = vgather addrs with mask=active  : true
  needle     = vload needles with mask=active  : true
  found      = vcmp eq, t, needle              : true
  not_found  = vnot found                      : true
  i'         = vadd i, {1, 1}                  : true
  lt_n       = vcmp lt, i', {n, n}             : true
  active'    = vand active, not_found, lt_n    : true
  i_out'     = vselect active, i, i_out        : true
  found_out' = vselect active,
                       found,
                       found_out               : true
  cont       = vreduce or, active'             : true
while cont                                     : true
vstore needle_idxs, i_out with mask=found_out  : true
```

Figure 11. Result of packing and vectorizing the loops in Figure 2. `vmu` denotes vector μ-node. For vector load/store instructions that require masking, we use their original control predicates (e.g., `active` and `active2` in Figure 10) to compute the masks.

Emitting Vector Instructions. After scheduling, we replace the packed instructions with their corresponding vector instructions. For most packs, lowering into vector instructions is straightforward. For example, a pack of additions becomes a single vector addition. We need to take special care of packs of ϕ-nodes and packs of loads or stores. Figure 11 shows the result of packing the instructions in Figure 10 and translating them into vector instructions.

Within a pack of ϕ-nodes, if the control predicate of i'th ϕ operand is different from the control predicate of i'th operand of a different ϕ-node, we lower the pack of ϕ-nodes into a vector select; otherwise, we lower the pack ϕ-nodes into a single vector ϕ-node. To lower a pack of n-ary ϕ-nodes into vector select, we emit a chain of $n-1$ selects and emit the select conditions according to the incoming control predicates of the ϕ-nodes. If we pack loads (or stores) that have different control predicates, we emit masked vector loads (instead of ordinary vector loads or stores).

Assigning Control Predicate. Because we support packing instructions with different control predicates, we need a mechanism to assign a new control predicate for each new vector instruction. For a pack of instructions with identical control predicates, we simply assign the same control predicate. For a pack of instructions that have different control predicates, we set the new control predicate to the strongest control predicate which is also the necessary condition of (i.e., implied by) the original predicates. For example, if a pack contains two instructions with the predicates $c \wedge c_2$ and $c \wedge c_3$, we set the new control predicate to c.

3.5 Lowering to IR with control flow

Figure 12 shows the algorithm for converting Predicated SSA to a lower-level IR with control flow. We first convert the gated ϕ-nodes into move instructions, whereby we temporarily destroy the single-definition invariant of SSA (which we will restore later). Once we remove all ϕ-nodes, we proceed to reconstruct the CFG level by level, from the top-level function to the deepest nested loop. For each item, we first create (or reuse) a basic block whose control predicate is equivalent to that of the item's (line 16). If the item is a loop, we lower the loop recursively and use the basic block as the loop's pre-header (line 18). If the item is an instruction, we simply move the instruction to the basic block (line 20).

Finally, we generate the back edge and exits of the loop as follows. We first allocate a special variable that indicates whether the loop should continue (line 35). We initialize this variable to *false* in the loop header (i.e., the loop exits by default). Next, we create a basic block based on the control predicate of the loop continue condition (p_{cont} in Figure 4), and in this basic block, we set the variable to *true*. Finally, in the loop latch, we create a conditional branch that goes to the header if the variable is true and otherwise to the loop exit (line 45).

We use a utility data structure that we call *block builder* (line 6) to encapsulate the operation of recreating basic blocks from control predicates. While lowering the predicates into basic blocks, the builder maintains a hash table that maps the source predicate of every block to the block itself. We refer to this hash table as the predicate table. For each conjunction of the form $p \wedge c$, the builder queries the predicate table for an existing block equivalent to the predicate p (and recursively creates such a block if it does not exist) and creates a conditional branch (based on the condition c) to the new basic block. For each disjunction, the builder similarly queries the predicate table for basic blocks equivalent to the sub-terms of the disjunctions and joins those basic blocks.

4 Implementation

We implemented our vectorization framework within the LLVM compiler infrastructure [15].[3] We implemented our prototype with 7,831 lines of C++. All components of the prototype (i.e., the new IR infrastructure, dependence analysis, and transformation) are implemented from scratch, independent of LLVM's existing vectorization infrastructure.

4.1 Loop Unrolling

Our vectorizer exposes loop-level parallelism with unrolling. Because SuperVectorization can pack instructions from different loop nests, it achieves outer-loop vectorization by unrolling outer loops and then packing the instructions from the duplicated inner loops. We unroll loops following an approach similar to Rocha et al. [29]'s. We traverse a given

[3] LLVM 12.0.0.

loop nest top-down (parents before children), unrolling each loop virtually and running an SLP packing algorithm (without actually lowering the packs into vector instructions) to determine whether unrolling exposes more parallelism.

4.2 Dependence Analysis

Unlike a loop vectorizer, our framework does not need to reason with loop-carried dependence because it does not pack an instruction with a different instance of itself (from a different iteration). Similar to a traditional SLP vectorizer, we only need to check for loop-independent dependence. Nonetheless, our framework requires testing whether any two given loops are independent. To prove that two loops are independent, we first check that there are no use-def dependences (through registers). We then prove that there are no read-write memory dependences between the loops.

For any two memory instructions (with at least one being a write) from the two loops, we overapproximate the ranges of memory locations that could be accessed by the instructions and verify that the ranges do not overlap (using LLVM's ScalarEvolution analysis). In the cases where the memory accesses are unpredictable and, therefore, have compile-time uncomputable bounds, LLVM's alias analysis framework also allows checking whether two memory regions with unknown sizes overlap. Finally, to improve analysis precision, we have special-case support to distinguish memory accesses that may have overlapping ranges but are nonetheless separated by a sufficient offset in their access strides (e.g., two loops may write to the same buffer but with one writing to the odd indices and the other writing to the even indices).

4.3 Cost Model

Our implementation models the cost-saving of vectorization similar to existing SLP vectorizers and only vectorizes when the saving outweighs the cost.

The saving of vectorization comes from replacing the scalar instructions with fewer, more efficient vector instructions, and its cost comes from the overhead of data-movement instructions such as vector shuffles, which are necessary when the vectorizer is unable to directly produce the vector operands for some of the vector instructions. Consider the following example, where the vectorizer is able to vectorize everything except for the two function calls.

```
  t1   = call f();
  t2   = call g();
  add1 = add t1, 1
  add2 = add t2, 1
  store add1, x[0]
  store add2, x[1]
=>
  t1    = call f();
  t2    = call g();
  t     = pack {t1, t2}
  vadd0 = vadd t, {1, 1}
  vstore vadd0, x
```

```
def lower(func_or_loop, entry):
  eliminate_phis(func_or_loop)

  # Utility structure to create basic blocks
  # from control predicates
  block_builder = BlockBuilder(entry)
  if func_or_loop.is_loop():
    # Allocate dedicated latch and exit blocks
    latch = BasicBlock()
    exit = BasicBlock()
    # the header executes unconditionally
    header = block_builder.get_block(true)

  for item in func_or_loop.items:
    pred = func_or_loop.control_pred_of(item)
    b = block_builder.get_block(pred)
    if item.is_loop():
      loop_exit = lower(item.as_loop(), entry=b)
    else:
      item.as_inst().move_to(b)

  if func_or_loop.is_loop():
    loop = func_or_loop.as_loop()

    # finally convert the mu-nodes back to phi-nodes
    for mu in loop.mus:
      phi = Phi(
        incoming_blocks=[entry, latch],
        incoming_values=[mu.init, mu.rec],
        insert_at=header)
      replaceAllUsesWith(mu, phi)

    # Allocate a dedicate, flag variable,
    # indicating if we should continue
    should_continue = Alloca(type=bool, insert_at=header)
    # Set the flag to false
    assign_false(should_continue, insert_at=header)

    # Create a block that
    # has the same predicate as the continue predicate
    temp_block = block_builder.get_block(loop.cont_pred)
    # and set the continue flag there
    assign_true(should_continue, insert_at=temp_block)

    # Create a block that executes unconditionally
    pre_latch = block_builder.get_block(true)
    # and load the continue flag
    cont = create_load(should_continue,
                       insert_at=pre_latch)
    # Then exit or continue depending on the flag
    create_branch(cond=cont,
                  if_true=latch, if_false=exit,
                  insert_at=pre_latch)

  restore_ssa()
  return None
```

Figure 12. Algorithm for lowering Predicated SSA to a traditional IR with CFG (e.g., LLVM IR)

Because the vectorizer cannot directly vectorize the calls to `f` and `g`, it then needs to emit an additional instruction to explicitly pack the result of the calls into a vector using the highlighted instruction.

For each input function, we evaluate the cost saving from vectorization as follows and only vectorize when the saving is greater than zero.

$$c_{scalar} - c_{vector} - c_{shuffle}$$

where the terms c_{scalar}, c_{vector}, and $c_{shuffle}$ are the total cost of the packed scalar instructions, the cost of vector instructions, and the cost of vector shuffling (and other vector data-movement instructions), respectively. For all three terms, we use LLVM's target-specific cost model [3], which estimates the expected cost of individual IR instructions after instruction selection. Our implementation does not currently model the change of cost from restructuring the input CFG (e.g., co-iterating loops).

5 Evaluation

We evaluated our framework on three benchmark suits: TSVC [8], PolyBench [28], and a collection of sequential C++ programs that Pharr and Mark [23] used as baselines to evaluate `ispc`, a data-parallel language they designed to target SIMD extensions.

For all the experiments, we used a machine with an Intel® Core™ Platinum 8180 CPU with AVX-512 support and 32 GB of memory and with hyperthreading disabled. Unless otherwise noted, we use the same optimization pipeline as the LLVM `-O3` optimizations, except that we run our vectorizer instead of LLVM's loop and SLP vectorizers (which are disabled in our pipeline). Although the experiment machine comes with AVX-512 support, LLVM 12, by default, sets the maximum vector width to 256-bit (less than the full 512-bit) to avoid performance scaling.

5.1 TSVC

Figure 13 shows the results on TSVC, a comprehensive benchmark suite for evaluating automatic vectorizers [8]. Our vectorizer is on par with LLVM and yields an average (geomean) speedup of 1.04× over LLVM's vectorizers on TSVC. Our vectorizer gets significant speedups (more than 1.5× and up to 4.3×) on 17 benchmarks and significant slowdowns (more than 20%) on 18 benchmarks. For the 17 benchmarks with significant speedup, the speedups come from partial vectorization and outer-loop vectorization with complex control dependences. Partial vectorization comes naturally for SLP vectorization because the packing heuristic is not required to pack all instructions and can selectively pack any instructions that it finds profitable and legal. In contrast, a traditional vectorizer only vectorizes if all of the loop iterations are independent. Figure 16 shows an example benchmark where we obtain a 2.91× speedup by vectorizing an outer loop that also contains data-dependent control flow.

The slowdowns mainly come from benchmarks that have potential data dependences that prevent vectorization. In such a case, LLVM's vectorizer versions the loop into two duplicates, with one of the duplicated loops vectorized and guarded with a run-time check to ensure that there are no unsafe dependences. We have not yet implemented code versioning and do not vectorize in these cases.

5.2 PolyBench

PolyBench [28] is a benchmark for polyhedral optimizations, and we use it here as a vectorization benchmark because its applications also benefit from vectorization and because they have data-access patterns that benefit from outer-loop vectorization, and we want to validate that we can effectively perform outer-loop vectorization with unrolling plus SLP.

Figure 14 shows the PolyBench results. PolyBench comes with a configurable dataset size; for our experiments, we use the *large* dataset size. Our vectorizer yields a 1.46× average speedup over LLVM's vectorizers (Loop + SLP) and a 1.80× average speedup over LLVM's scalar baseline (`-O3` without vectorization).

Because PolyBench is a benchmark for polyhedral optimizers, we also compared our approach with Polly [10], a polyhedral optimizer for LLVM IR. Polly performs aggressive cache optimizations and vectorization, achieving significant speedup over LLVM (`-O3`). To separate the effects of Polly's cache optimization from its vectorization, we run Polly in three settings: Polly with LLVM vectorization, Polly with its own vectorization, and Polly with our vectorization. Figure 15 shows the comparison against Polly. Compared to LLVM's full optimizations (`-O3` with vectorization), Polly's cache optimizations gets a 1.41× average speedup; Polly's vectorization gains a 1.49× average speedup (1.06× over Polly's cache optimization); and Polly combined with our vectorizer obtains a 1.82× average speedup (1.22× over Polly's cache optimization).

5.3 ISPC

Pharr and Mark [23] originally developed their benchmarks to motivate the need for `ispc`, a data-parallel programming model and language that they proposed to target applications with complex data and control dependences (e.g., graphics simulations) unsuitable for auto-vectorization. For all the ISPC benchmarks, we use their baseline, scalar implementations; in other words, the benchmarks have sequential semantics, and the vectorizers must uncover the parallelism from scratch. Figure 17 shows some code snippets from Volum Rendering, one of the benchmarks. Volume Rendering consists of five imperfectly nested loops with early exits.

Figure 18 shows the results on the `ispc` benchmarks [23]. Our vectorizer yields a 1.88× average speedup over LLVM's vectorizers (Loop + SLP) and a 2.43× average speedup over LLVM's scalar baseline (`-O3` without vectorization).

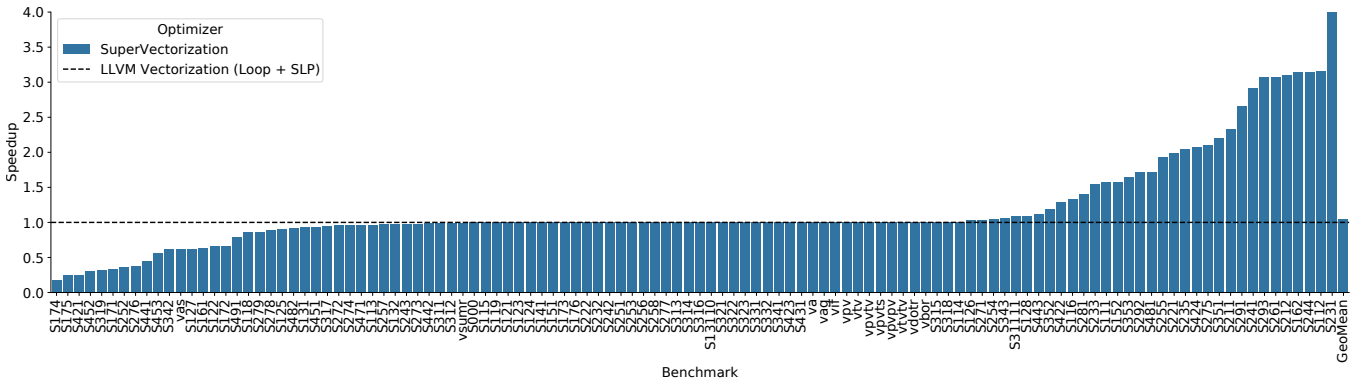

Figure 13. Speedup over LLVM scalar optimizations (-O3) on TSVC. The benchmarks are sorted (in increasing order) according to speedup from SuperVectorization over LLVM's scalar optimizations.

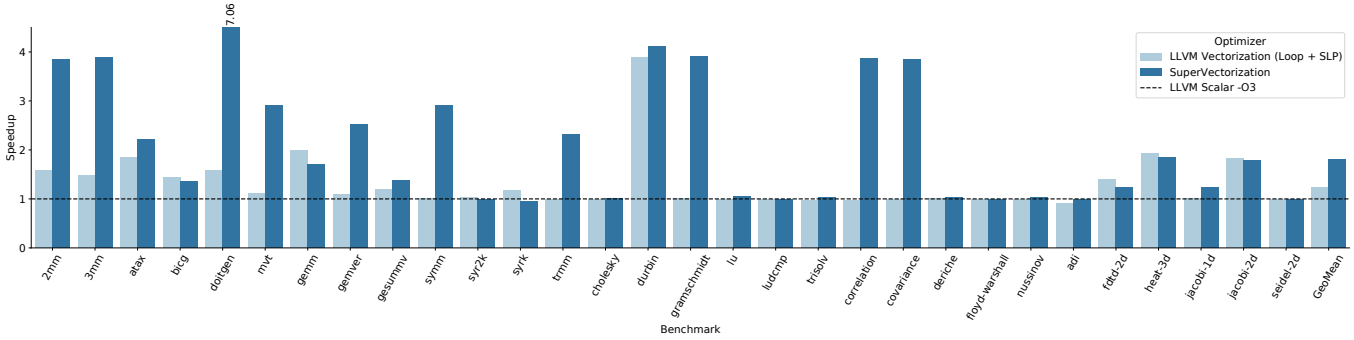

Figure 14. Speedup over LLVM scalar optimizations (-O3) on PolyBench

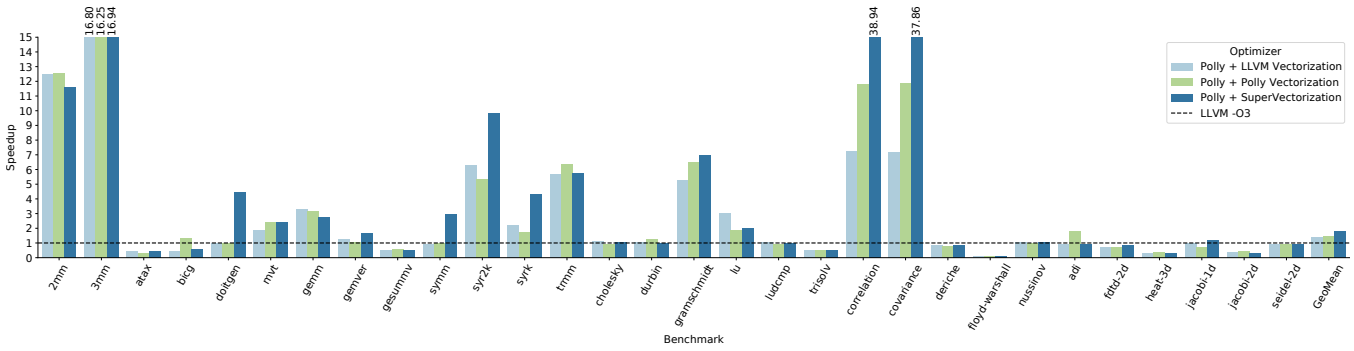

Figure 15. Speedup over full LLVM optimizations (-O3 + vectorization) on PolyBench using Polly

```
for (int i = 0; i < LEN2; i++)
  if (aa[0][i] > (float)0.)
    for (int j = 1; j < LEN2; j++)
      aa[j][i] = aa[j-1][i] + bb[j][i] * cc[j][i];
```

Figure 16. An example benchmark from TSVC where our vectorizer gains a 2.91× speedup over LLVM's vectorizers by unrolling the outer i-loop and packing instructions across the inner j-loops and their loop guards.

```
  ...
static float transmittance(...) {
  ...
  if (!IntersectP(ray, pMin, pMax, &rayT0, &rayT1))
      return 1.;
  ...
  while (t < rayT1) {
      tau += ...
      pos = pos + dirStep;
      t += stepT;
  }
  ...
  return expf(-tau);
}

static float raymarch(...) {
  ...
  if (!IntersectP(ray, pMin, pMax, &rayT0, &rayT1))
      return 0.;
  ...
  while (t < rayT1) {
      ...
      // terminate once attenuation is high
      float atten = expf(-tau);
      if (atten < .005f)
          break;

      // direct lighting
      float Li = ... / transmittance(...)
      ...
  }

   // Gamma correction
  return powf(L, 1.f / 2.2f);
}

void volume_serial(...) {
  int offset = 0;
  for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x, ++offset) {
      Ray ray;
      generateRay(..., ray);
      image[offset] = raymarch(density, nVoxels, ray);
    }
  }
}
```

Figure 17. Code snippets from Volume Rendering, which contains five imperfectly nested loops with early exits and multiple live-outs. Our vectorizer achieves a 3.28× speedup by vectorizing the outer x-loop. No production compiler vectorized this program.

.

For the ISPC benchmarks, we extended SuperVectorization to recognize and vectorize the random number generator `drand48` (used by aobench). Without the modification, the vectorizer cannot reorder different calls to `drand48` and does not vectorize aobench. Nonetheless, `drand48` is not the sole reason that LLVM does not vectorize. In a separate experiment, we modify aobench to use a compile-time constant instead of calling `drand48`. Even in this setting, LLVM's vectorizer does not vectorize, whereas our vectorizer reproduces the speedup.

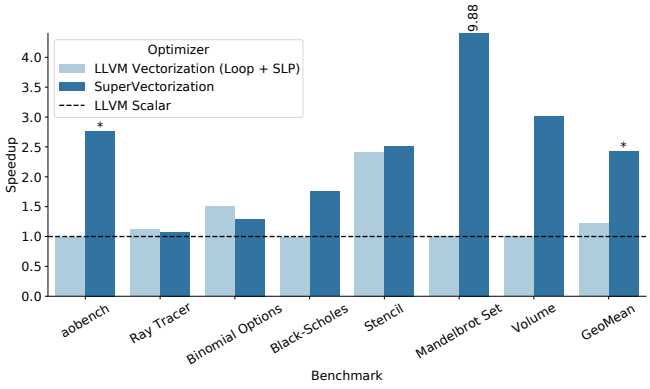

Figure 18. Speedup over LLVM vectorization (Loop + SLP) on the ISPC benchmarks. aobench (marked with *) uses the random number generator `drand48`; We modified SuperVectorization to recognize `drand48` and to assume that it is safe to reorder different calls to `drand48`. Without this modification, aobench is not vectorizable, and the geomean speedup becomes 2.09× (instead of 2.43×).

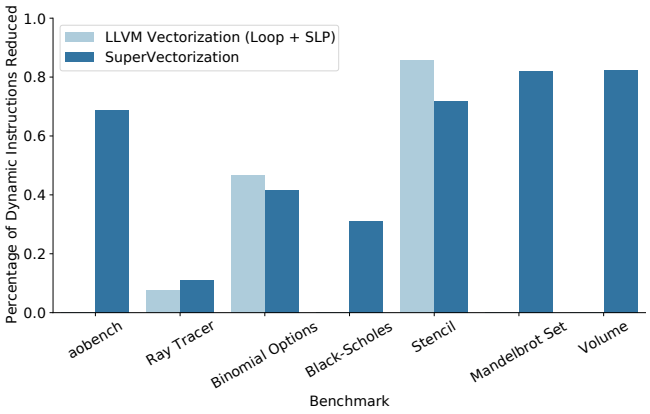

Figure 19. Reduction in dynamic instruction counts in the ISPC benchmarks

The only ISPC benchmarks that our vectorizer does not vectorize is Ray Tracer. Ray Tracer reuses a single array across computations that are otherwise logically independent. Consequently, our dependence analysis concludes that all reads and writes to the array are dependent, thus preventing vectorization.

To analyze the source of the speedups, we collected hardware performance counters. Figure 19 shows the reduction of dynamically executed instructions for the ISPC benchmarks. The speedups correlate with the reduction in the number of instructions executed at runtime. We observed similar correlations in TSVC and PolyBench. In some PolyBench benchmarks such as *2mm*, not only did our vectorizer reduce the number of instructions executed but also reduced L1 data cache miss rates from 60% to 40%.

6 Related Work

There has been extensive work on compiler auto-vectorization. We list some of the key developments here.

Loop Vectorization. Allen and Kennedy [4] pioneered loop vectorization. Their seminal paper also introduced applying if-conversion to vectorize control flow. Nuzman et al. [19] extended the traditional loop-based vectorization algorithm to support interleaved data accesses. Baghsorkhi et al. [6] proposed an architectural extension and a joint vectorization for algorithm detecting and speculating over data dependences at runtime. Nuzman and Zaks [20] proposed an outer-loop vectorization approach based on unroll-and-jam.

SLP Vectorization. Larsen and Amarasinghe [14] developed SLP vectorization as an alternative to traditional loop-based vectorization. SLP vectorization targets short-vector parallelism and requires a much simpler dependence analysis in comparison to traditional loop vectorizers. Almost all subsequent work following Larsen and Amarasinghe [14] proposed algorithmic improvements to the original algorithm to uncover more parallelism. Examples include Holistic SLP vectorization [16], Super-node SLP [27], TSLP [24], PSLP [25], VW-SLP [26], and applying integer linear programming (ILP) solver to select vector packs [17]. Shin et al. [31] extended SLP to vectorize control flow with if-conversion.

Compilation for Data-Parallel Languages. There is a related line of work on compiling data-parallel languages for vector architectures. In contrast to auto-vectorization, compilers for such language do not need to extract vector parallelism or prove its safety because their source semantics are already data-parallel. OpenCL provides a data-parallel programming model that maps to SIMD extensions [11]. Karrenberg and Hack [12] proposed a set of techniques for vectorizing arbitrary reducible CFGs in a data-parallel language. Moll and Hack [18] proposed techniques for recovering control flow from if-conversion, thereby reducing the amount of speculative computation and improving utilization. Pharr and Mark [23] proposed `ispc`, a C-like data-parallel language that targets SIMD extensions.

Compiler Intermediate Representation. There has been extensive work towards developing IRs to simplify the downstream compiler optimizations. Cytron et al. [9] developed a seminal algorithm for efficiently computing SSA form, which is now used by production compilers such as GCC and LLVM. Ottenstein et al. [21] originally proposed gated SSA as an IR for their Fortran compiler that targets dynamic data-flow architectures. Tu and Padua [33] developed an algorithm for efficiently computing gated SSA. Tristan et al. [32] used gated SSA for translation validation.

7 Conclusion

We have introduced a new vectorization framework that targets arbitrary superword-level parallelism that spans different basic blocks and loops. With loop unrolling, our vectorizer matches and, in many cases, outperforms LLVM's vectorization pipeline, which uses both loop and SLP vectorizers. For example, our vectorizer vectorizes and accelerates several of the benchmarks that Pharr and Mark [23] argued as being unsuitable for automatic vectorization due to their complex control and data dependences. Our vision for the future is that a single vectorizer based on our framework would be sufficient for a vectorizing compiler.

Acknowledgments

We thank our shepherd Laure Gonnord and the anonymous reviewers for their valuable suggestions. We thank Teodoro Collin, Logan Weber, Jesse Michel, Alex Renda, Daniel Donenfeld, and Changwan Hong for reading early drafts of this paper and providing feedback. Our work is supported by the DARPA/SRC JUMP ADA Center; the Toyota Research Institute; the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DESC0008923 and DESC0018121; NSF 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] 2022. Auto-Vectorization in GCC. https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[2] 2022. Auto-Vectorization in LLVM. https://llvm.org/docs/Vectorizers.html.

[3] 2022. llvm::TargetTransformInfo Class Reference. https://llvm.org/doxygen/classllvm_1_1TargetTransformInfo.html.

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

[5] Randy Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Symposium on Principles of Programming Languages*.

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

[7] Bob Blainey, Christopher Barton, and José Nelson Amaral. 2002. Removing impediments to loop fusion through code transformations. In *International Workshop on Languages and Compilers for Parallel Computing*.

[8] David Callahan, Jack J Dongarra, and David Levine. 1988. Vectorizing Compilers: A Test Suite and Results. In *ACM/IEEE Conference on Supercomputing*.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* (1991).

[10] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly – Performing polyhedral optimizations on a low-level intermediate

representation. *Parallel Processing Letters* (2012).
[11] Khronos Group. 2009. OpenCL 1.0 Specification. http://khronos.org/registry/cl/specs/opencl-1.0.pdf.
[12] Ralf Karrenberg and Sebastian Hack. 2011. Whole Function Vectorization. In *International Symposium on Code Generation and Optimization*.
[13] Ken Kennedy and Kathryn S McKinley. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 301–320.
[14] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Programming Language Design and Implementation*.
[15] 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*.
[16] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. 2012. A Compiler Framework for Extracting Superword Level Parallelism. In *Programming Language Design and Implementation*.
[17] Charith Mendis and Saman Amarasinghe. 2018. goSLP: Globally Optimized Superword Level Parallelism Framework. *Proceedings of the ACM on Programming Languages* (2018).
[18] Simon Moll and Sebastian Hack. 2018. Partial Control-Flow Linearization. In *Programming Language Design and Implementation*.
[19] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of Interleaved Data for SIMD. In *Programming Language Design and Implementation*.
[20] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *International Conference on Parallel Architectures and Compilation Techniques*.
[21] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Programming Language Design and Implementation*.
[22] Joseph CH Park and Mike Schlansker. 1991. *On predicated execution*.
[23] Matt Pharr and William R. Mark. 2012. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing*.
[24] Vasileios Porpodas and Timothy M. Jones. 2015. Throttling Automatic Vectorization: When Less is More. In *Conference on Parallel Architecture and Compilation*.
[25] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP Automatic Vectorization. In *International Symposium on Code Generation and Optimization*.
[26] 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*.
[27] 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*.
[28] Louis-Noël Pouchet. 2021. PolyBench/C: the polyhedral benchmark suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.
[29] Rodrigo C. O. Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís F. W. Góes, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Vectorization-Aware Loop Unrolling with Seed Forwarding. In *International Conference on Compiler Construction*.
[30] Ira Rosen, Dorit Nuzman, and Ayal Zaks. 2007. Loop-aware SLP in GCC. In *GCC Developers Summit*.
[31] Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *International Symposium on Code Generation and Optimization*.
[32] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-Graph Translation Validation for LLVM. In *Programming Language Design and Implementation*.
[33] Peng Tu and David Padua. 1995. Efficient Building and Placing of Gating Functions. In *Programming Language Design and Implementation*.