VW-SLP: Auto-Vectorization with Adaptive Vector Width

Vasileios Porpodas
Intel Corporation, USA
vasileios.porpodas@intel.com

Rodrigo C. O. Rocha
University of Edinburgh, UK
r.rocha@ed.ac.uk

Luís F. W. Góes
PUC Minas, Brazil
lfwgoes@pucminas.br

ABSTRACT

Auto-vectorization techniques allow the compiler to automatically generate SIMD vector code out of scalar code. SLP is a commonly-used algorithm for converting straight-line code into vector code, which complements the loop-based traditional vectorizers. It works by scanning the input code looking for groups of instructions that can be combined into vectors and replacing them with the corresponding vector instructions. The state-of-the-art SLP algorithm works by attempting to vectorize blocks of code with a fixed vector width and falling back to smaller widths for the whole block upon failure.

In this work we remove this limitation and introduce Variable-Width SLP (VW-SLP), a novel algorithm that is capable of adjusting the vector width at an instruction granularity. This allows the algorithm to better adapt to the code's SIMD parallelism characteristics, thus exposing more vector parallelism than before. We implemented VW-SLP in LLVM and our evaluation on a real system shows that it considerably improves the performance of real benchmark code, with a small increase in compilation time.

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; • **Software and its engineering** → **Compilers**;

KEYWORDS

SIMD, SLP, Auto-Vectorization

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243189

ACM Reference Format:
Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. VW-SLP: Auto-Vectorization with Adaptive Vector Width. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243176.3243189

1 INTRODUCTION

Auto-vectorization is a performance-critical optimization in modern optimizing compilers. Its goal is to automatically generate SIMD instructions out of scalar code, allowing parts of the program to run on the processor's high-throughput SIMD units, without any effort from the programmer's side. An alternative approach is to move the burden of both expressing the vector parallelism and tuning its performance from the compiler to the programmer. There are several ways of expressing vector parallelism, including target-specific intrinsics, a vector-aware language, or more commonly, a programming model (e.g. OpenMP [8]). Auto-vectorization is the preferred approach for most software projects, except for some highly tuned library kernels or for specific High Performance Computing workloads.

There are two main approaches for auto-vectorizing scalar code: the traditional loop-based algorithms (e.g., [2, 3]) and those operating on straight-line code (e.g., [18, 35]).

Superword-Level Parallelism (SLP), first introduced by Larsen and Amarasinghe [18], is the algorithm originally proposed for vectorizing straight-line code. Instead of relying on a loop structures, it looks for groups of isomorphic instruction sequences that can be converted into vectors. Similar algorithms to the original SLP are implemented in several optimizing compilers. The bottom-up SLP algorithm [35] is the one implemented in both GCC [11] and LLVM [19] due to its good coverage and fast run-time. Throughout the rest of the paper, with the term "SLP", we are referring to this bottom-up SLP algorithm present in GCC and LLVM.

SLP works by first scanning the code for scalar instructions that can become the seeds of vectorization and grouping them together. This group becomes

the root of the SLP graph, a graph structure that holds all groups of potentially vectorizable scalar instructions. Then, SLP walks up the use-def chains, towards definitions, attempting to group more isomorphic instructions together, as long as they can be potentially vectorized. This process repeats until the SLP graph is completed. The next step is to evaluate whether converting the groups of the SLP graph into vectors can improve performance based on the compiler's built-in cost model. This cost calculation factors in the overheads of inserting/extracting data into/out of the vector registers. If vectorization is shown to be profitable, vector instructions are generated to replace the groups of scalars.

In this paper, we extend the SLP algorithm with a strategy for varying the vector width[1] (i.e. the number of scalars that fit within each group) at an instruction granularity while building the SLP graph. This allows the SLP graph to branch out into either narrower branches or wider ones as dictated by the underlying code. The state-of-the-art algorithm, on the other hand, maximizes the vector width at the root of the graph (the seed group node) and maintains the same vector width for the whole SLP graph. The proposed Variable-Width SLP algorithm avoids early termination when either shorter or wider widths are required by the underlying code, thus leading to greater coverage than before. The end result is a more powerful vectorizer, capable of generating faster code.

2 BACKGROUND

2.1 Loop Vectorization Versus SLP

Traditional loop vectorization works by conceptually fusing consecutive loop iterations and replacing the fused instructions by their vectorized form. This requires a well-formed loop structure with statically analyzable loop dependencies (or dynamically evaluated with multi-versioning) and simple control flow.

Straight-line code algorithms, on the other hand, like the SLP vectorizer [18, 35], (*i.*) are not restricted to operate within loops, i.e., they can handle straight-line code anywhere in the program, and (*ii.*) can vectorize code within loops where the loop vectorizer fails.

Both the loop-based and the straight-line code algorithms conceptually perform the same operation of reducing VL (Vector-Length) isomorphic instructions into VL-wide vector instructions. However, they follow different approaches to achieve this result. In loop-based algorithms, the presence of the loop structure implies the presence of multiple copies of each instruction in neighboring iterations. Straight-line code algorithms, on the other hand, do not rely on the presence of a loop, so they have to explore the code to find repeated sequences of isomorphic scalar instructions.

The straight-line code vectorization can be considered as a superset algorithm of broader scope compared to the loop-based vectorizer, in particular with the support of loop unrolling [18]. However, in practice this is not yet the case, and both the loop vectorizer and the SLP vectorizer are executed for the best coverage. A common configuration is to run SLP after the loop vectorizer.

2.2 SLP Vectorization

The bottom-up SLP algorithm [35] has been implemented in both GCC and LLVM. Its goal is to find isomorphic instruction sequences and vectorize them if profitable. It works by first scanning the compiler's intermediate representation (IR), identifying specific type of instructions, referred to as *seeds*. The seeds are instructions which are likely to form vector sequences. These are usually stores or instructions that form reduction trees. The seeds become the first potential vector group and are the starting point of the algorithm. The algorithm then searches for more instructions to group together by following the use-def chains, starting from the seeds and then following the operands, in an attempt to extend the SLP graph. The code to be vectorized can span multiple basic blocks, as long as all instructions in the group belong to the same basic block.

The flowchart of Figure 1 shows an overview of the SLP algorithm, with the highlighted section showing where VW-SLP has introduced its changes. The SLP algorithm first scans the input IR for vectorizable seed instructions (step 1), which are instructions of the same type and bit-width that are likely to form vectorized code e.g.: (*i.*) non-dependent store instructions that access adjacent memory locations (scalar evolution analysis [5] is commonly used to test for this), (*ii.*) instructions that lead to vectorization idioms such as reduction trees (e.g. a reduction tree of additions), 9*iii.*) address calculation instructions that feed into non-consecutive loads, etc. Compilers commonly look first for adjacent store seeds first [35], as they are the most promising seeds. The seeds are then inserted into a worklist.

The algorithm will then attempt to vectorize the code at the widest possible vector size and will fall

[1] With the term vector width we refer to the number of lanes in the SIMD vector. Vector size, vector length and vector factor are also commonly used to describe this.

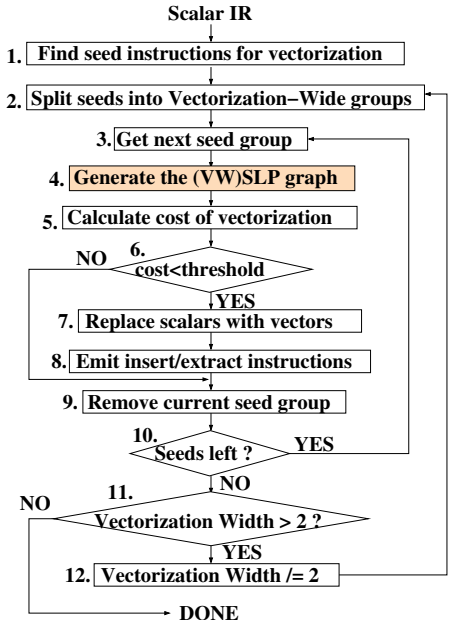


Figure 1: Overview of Bottom-up SLP.

back to a smaller size if vectorizing at the wide vector failed. The intuition behind this is that wider vectors lead to better performance and should therefore be tried out first. To that end, the algorithm splits the seeds into vectorization-width sizes (step3). Initially, the vector width is set to the largest valid size that the target architecture can execute (for that particular data type), limited by the number of available seeds. For example, for Intel AVX2 the target architecture can execute 4-wide vectors of 64-bit integers, so if we have more than 4 seeds available, our seeds will form groups of four.

Next, the algorithm gets a seed group from the worklist (step 3) and starts to build the SLP graph (step 4). Building the SLP graph involves forming groups of potentially vectorizable instructions by following the dependence graph (use-def chains) towards the definitions (bottom-up). This is the approach followed in both GCC's and LLVM's implementations of the SLP vectorizer [35]. Each group contains the scalar instructions that are candidates for vectorization, but it also carries some additional auxiliary data such as the group's cost (see next step). Once the graph-building process encounters instructions that cannot form a vectorizable group (e.g, due to non-matching opcodes), it forms a non-vectorizable group which indicates that scalar-to-vector data movement is required. Moreover this group carries the additional cost introduced by these instructions that will insert the scalar data into vectors. At this point the algorithm stops exploring this path any further as vectorization cannot proceed.

Listing 1: Simplified SLP graph generation.

```
1  build_graph(instrs) {
2   // i. Termination conditions
3   if instrs not vectorizable: return
4   // ii. Append new node to graph
5   graph.add(new_group_node(instrs))
6   // iii. Operand ordering based on opcode
7   reorder_operands(instrs.operands)
8   // iv. Recursion for each group of operands
9   for operands in instrs.get_operands():
10    build_graph(operands)
11 }
```

VW-SLP improves the SLP-graph formation by not bailing out immediately upon encountering such non-vectorizable instruction groups. Instead, it attempts to extend the graph towards both narrower and wider instruction groups, leading to better vectorization coverage.

After constructing the SLP graph, the algorithm needs to estimate the performance benefits of the vectorized code (step 5) in order to decide whether or not to actually use the vectorized code or keep the scalar version. This is done with the help of the compiler's target-specific cost model. The cost of the graph is equal to the sum of the savings from converting each group of scalar instructions into vector form (the lower the cost the better) plus the overheads for gathering the inputs of the vector instructions. In step 6 the cost is compared against a threshold (usually 0) to determine whether code generation of the vector code should proceed (steps 7 and 8). If so, the compiler modifies the intermediate representation code by replacing the groups of scalar instructions with their equivalent vector instructions (step 7), and emits any *insert* or *extract* instructions required for the flow of data between the vector and scalar instructions (step 8). But if the cost of vectorization is higher than that of the scalar code, the code remains unmodified. Next, the current seed group is removed from the worklist (step 9) and the process repeats until we have processed all seeds in the worklist (step 10).

Once all seed groups of the current Vectorization Width have been processed, the algorithm falls back to a smaller width of half the size if possible (steps 11, 12 and back to step 2).

2.3 SLP Graph Generation

The SLP Graph is generated in step 3 with the help of the recursive function `build_graph()`, as shown in Listing 1. This function is initially called with the group of seed instructions as its inputs.

The `build_graph()` recursive function (line 1) has four distinct steps: *(i.)* check the termination

conditions[2] (line 2), *(ii.)* build the vectorizable nodes and connect them to the rest of the SLP graph (line 4), *(iii.)* perform operand reordering if it is legal and required (line 6). Only commutative instructions are legal candidates for reordering (not shown for brevity). In vanilla SLP, reordering considers the opcode of the operands of the current instruction, and in the case of loads whether or not they are consecutive[3].

(iv.) Finally, the function calls itself recursively, once for each operand group (commonly a group for the left operands and another group for the right operands). In this way, the function continues growing the graph up the use-def chains (line 8).

2.4 Existing Variable-Width Strategy

The existing SLP algorithm can adapt to the variable vector widths in two separate ways: (*1.*) It considers the gathering points (i.e. the points where vectorization stops) as new seeds. It will consequently attempt to form a new SLP-graph starting from these seeds, but at smaller vector widths. The end result is code that may be vectorized with two different vector widths: *i.* a larger width until the new seeds that form gathering point, and *ii.* a narrower width from the new seeds forward. (*2.*) It can fall back to narrower widths when it fails to generate a profitable SLP-graph with the current width.

Even though this strategy is adequate for some cases, it has a fundamental weakness. If the wide SLP-graph fails to be profitable, no new seeds will be generated for it, so the first point of its strategy becomes ineffective. Now SLP will fall back to the narrower width (point 2 of the strategy). But, in this way, it will not generate vector code with mixed width, but rather narrow vector code in the best case. This is sub-optimal compared to the proposed per-instruction variable-width vectorization approach, as we show in Section 3.

3 MOTIVATION

This section motivates VW-SLP with examples that highlight the weaknesses of the existing SLP algorithm, while demonstrating how VW-SLP overcomes them.

[2] The instructions must be: i) scalars, ii) isomorphic, iii) unique, iv) in the same basic block, v) schedulable, and vi) not yet in the SLP graph.

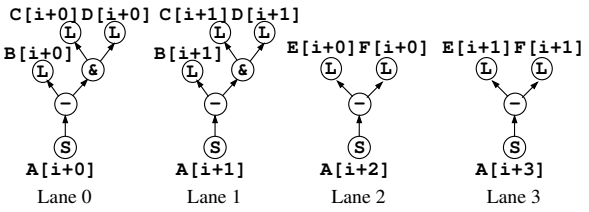
[3] As an example of operand reordering, if our input is $A[i] = B[i] + C[i]$ and $A[i+1] = C[i+1] + B[i+1]$, the second statement will become $A[i+1] = B[i+1] + C[i+1]$ to help match the operands of the first statement.

3.1 Shortening the Vector Width

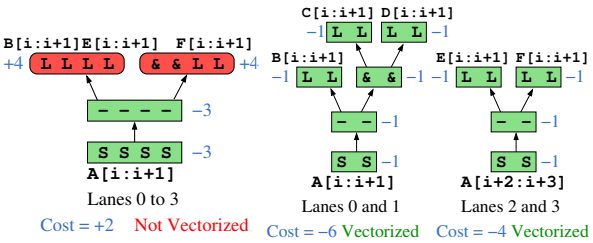
This example shows how vanilla SLP fails to generate optimal code when the source code's available vector parallelism changes from 4-wide to 2-wide within the block. The source code is in Figure 2(a) and the use-def dependence DAG for it is shown in Figure 2(b).

```
uint64_t A[],B[],C[],D[],E[],F[];
A[i+0]=B[i+0]-(C[i+0]&D[i+0]);
A[i+1]=B[i+1]-(C[i+1]&D[i+1]);
A[i+2]=E[i+0]- F[i+0];
A[i+3]=E[i+1]- F[i+1];
```

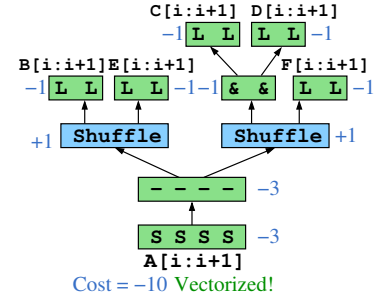
(a) Source Code



(b) Use-def chains DAG



(c) 4-wide SLP (d) 2-wide SLP



(e) VW-SLP

Figure 2: Vectorization Width Shortening.

Initially, state-of-the-art SLP will attempt to vectorize the code using the widest possible vectors for the seeds it has collected. In this example, the seeds are the four consecutive stores to `A[]`. It will, therefore, attempt 4-wide vectorization, as shown in Figure 2(c). The four stores (`S`) are vectorizable and they form a vectorizable (green) group node in

Figure 2(c). Next, SLP follows the use-def chains towards the definitions and attempts to group the four subtractions (−), which also lead to another vectorizable (green) group node. At this point both the left and right operands, are non-vectorizable. On the left hand side we have loads from B[i], B[i+1], E[i] and E[i+1], which cannot form a 4-wide vector load because they are not accessing consecutive memory addresses; instead they need to remain scalar (red group node) and their values have to be gathered into the consuming vector. On the right hand side we have two bit-wise ands (`&`) and two loads from F[i] and F[i+1]. These cannot form a 4-wide vector either (because the opcodes do not match), so they are also placed inside a red group node. Since all paths ended in non-vectorizable groups, the graph construction cannot proceed any further.

At this point SLP needs to determine whether it is profitable to generate vector code, given the SLP graph of Figure 2(c). This is the job of the cost model. It computes the cost of each node in the SLP graph (integers near the group nodes of Figure 2(c)). The cost is calculated as the difference VectorCost − ScalarCost, with negative cost values implying better performance of the vector code compared to the equivalent scalar code[4]. It is a metric of the overhead this group would introduce if converted into a vector. A typical ALU instruction (e.g., an integer ADD) has a cost of 1 in both scalar and vector form, therefore a group cost of −3 (VectorCost = 1, ScalarCost = 4) is quite common for a vectorizable group of four ALU instructions. The actual cost values come from querying the compiler's cost model[5]. Figure 2(c) has a total cost of +2 for the whole graph, which suggests that the code should remain scalar.

Since 4-wide vectorization failed, SLP will fall back to 2-wide vectorization (that is steps 11, 12 and back to 2 in Figure 1). The newly formed graph is shown in Figure 2(d). The algorithm operates twice, once for the stores to `A[i], A[i+1]` which creates the SLP graph of the left hand side of Figure 2(d), and once for the stores to `A[i+2], A[i+3]` creating the SLP graph of the right hand side. Both graphs are fully vectorizable, one with a cost of -6 and the other with a cost of -4. The 2-wide vectorized code is clearly faster than scalar code.

[4]The vectorization cost also accounts for the additional cost of extracting intermediary values with external uses.

[5] The compiler's cost model provides a target-dependent cost estimation that approximates the cost of an intermediate representation (IR) instruction when lowered to machine instructions. Our examples are based on LLVM's target-transformation interface (TTI) for the target Intel processor.

If we take a closer look into both Figures 2(c) and 2(d), we can observe that neither the 4-wide or the 2-wide approach follow the best vectorization strategy. In the 4-wide SLP graph of Figure 2(c) SLP managed to successfully vectorize the code, up until the subtractions (−). While in the 2-wide SLP graph of Figure 2(d), vectorization works fine for all nodes above the subtractions.

VW-SLP combines the strong points of both. We argue that the algorithm should adjust to the needs of the input code and adapt the vector width dynamically while building the SLP graph, at an instruction-level granularity. This is shown in Figure 2(e). VW-SLP starts, as usual, with 4-wide vectors up to the second level of the SLP graph (similarly to Figure 2(c)). It then realizes that 2-wide vectors are profitable and switches to 2-wide vectorization for each of the remaining branches. The transition from 4-wide to 2-wide is not free and requires the use of one or more data reorganization instructions[6], which we represent with a single `Shuffle` group node (light blue box). Please note that the `Shuffles` rever to LLVM's `shufflevector` IR instructions which can perform both (i) the common "shuffling" of operands, and (ii) "blending" of two vectors into one by selecting the individual lanes among them. The end result is an SLP-graph that combines the best parts of the fixed 4-wide and the 2-wide vectorization attempts of the state-of-the-art SLP algorithm.

3.1.1 Shortening with Repeated Instructions. A sub-problem of the generic shortening case is when a sub-group of instructions repeats within a group. For example if the 4-wide group contains (`I1 I2 I1 I2`), then VW-SLP is able to shorten it as well, but instead of creating two 2-wide vector branches of (`I1 I2`), only one gets created.

3.1.2 Comparing 2-wide vs 4-wide Costs. A closer look into the costs reported in Section 3.1, and more specifically in Figures 2(d) and 2(e), can raise questions about whether the 2-wide SLP cost of $-6 + (-4) = -10$ would be similar to the −10 cost of VW-SLP. This is a legitimate concern since the total cost savings of VW-SLP are comparable to that of SLP. However, in practice, longer vectors will usually lead to higher performance compared to shorter ones, due to the higher throughput that the wider vectors can achieve on the target architecture. This also explains why SLP will always try

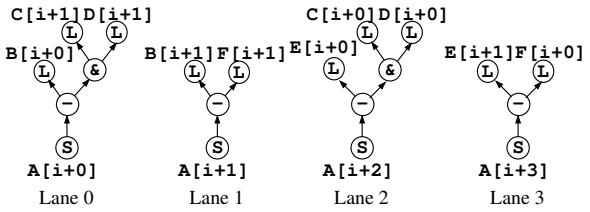
[6]When the `Shuffle` node gets lowered, it can generate different instructions, depending on the actual reorganization performed. Therefore its cost can vary, depending on the action performed.

to form longer vectors before forming shorter ones. To further support our claim, the performance of all the examples, including this code, are reported in Section 5.1.

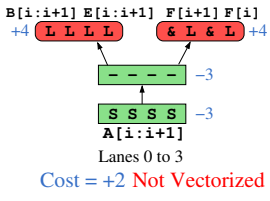
3.2 Shortening with Permutation

```
uint64_t A[],B[],C[],D[],E[],F[];
A[i+0]=B[i+0]-(C[i+1]&D[i+1]);
A[i+1]=B[i+1]- F[i+1];
A[i+2]=E[i+0]-(C[i+0]&D[i+0]);
A[i+3]=E[i+1]- F[i+0];
```

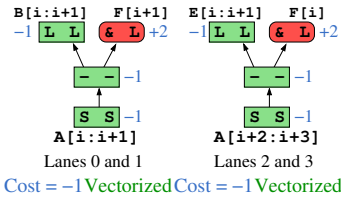
(a) Source Code



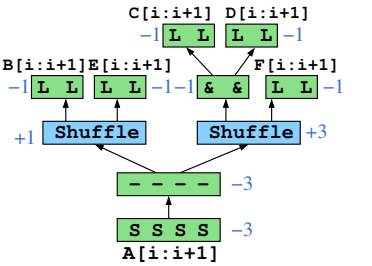
(b) Use-def chains DAG



(c) 4-wide SLP



(d) 2-wide SLP



(e) VW-SLP

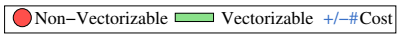


Figure 3: Vectorization Width Shortening with Permutation.

The example of Section 3.1 showed that vanilla SLP failed to vectorize some code at 4-wide SIMD, but succeeded in fully vectorizing it at 2-wide SIMD. Falling back, however, to shorter SIMD sizes will not always guarantee full vectorization. This is demonstrated by the example in Figure 3.

The input code in Figure 3(a) is very similar to that of Figure 2(a), with the difference being that the second operand of the subtractions are shuffled across lanes. This is more obvious if you compare the DAGs of Figure 3(b) against Figure 2(b). Therefore, we expect a very similar 4-wide SLP graph in Figure 3(c) to the one in Figure 2(c). The main difference is that the second operand of the subtraction group contains a different mix of instructions than before (`& L & L`) instead of (`& & L L`), but it is still not vectorizable. Therefore the 4-wide SLP still fails with a cost of +2.

Once again vanilla SLP falls back to 2-wide vectorization. It generates a pair of SLP graphs as shown in Figure 3(d). In this example, however, they do not get fully vectorized due to an opcode mismatch on the second operand of the subtraction group in both graphs. The non-vectorizable group is (`& L`) for both graphs.

VW-SLP will attempt to shorten the vectors at the points where it encounters non-vectorizable groups of the current vectorization width. Since shortening is not free and requires data shuffling anyway, VW-SLP will attempt to get the best permutation of the new shorter vectors, which may or may not add an additional cost as its overhead could be similar to the existing shuffling. Therefore, when the new shorter groups are created for the right-hand side operands of the subtraction group (`& L & L`), it will form a group of (`& &`) and (`L L`), instead of the naive (`& L`) and (`& L`). This allows VW-SLP to form vectorizable shortened groups and keep vectorizing further on, as illustrated in Figure 3(e). The end result is an identical SLP-graph compared to Figure 3(e) with a very similar cost.

3.3 Widening the Vector Width

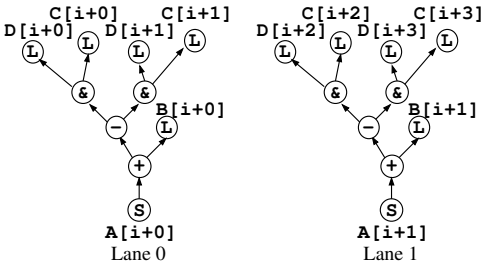
So far, in Sections 3.1 and 3.2, we showed how the shortening strategy enables the vectorization to continue with shorter vector widths while still benefiting from larger vector widths. In this example, we show that increasing the vectorization width can also be beneficial.

Given the code of Figure 4(a) that corresponds to the DAG of Figure 4(b), vanilla SLP will build a 2-wide SLP graph like the one shown in Figure 4(c). Vectorization proceeds fine until the loads are reached, which are not accessing consecutive memory addresses, and therefore remain scalar, which requires a gather operation. The cost model returns a cost of +2 which is not profitable for vectorization. Please note that there is neither a shorter vector width to try (we cannot go narrower than 2-wide), nor is there a 4-wide seed available to initiate a 4-wide vectorization attempt. Therefore the decision

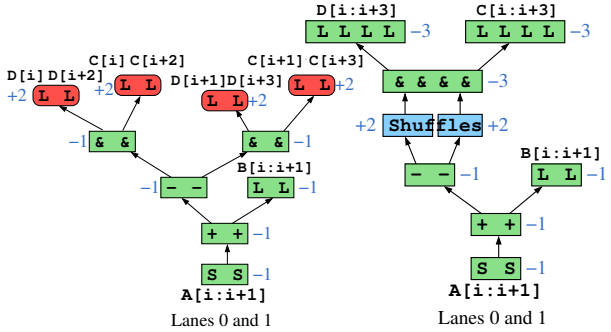
is final and this code does not get vectorized by the existing SLP algorithm.

```
uint64_t A[],B[],C[],D[];
uint64_t tmp0=C[i+0]&D[i+0];
uint64_t tmp1=C[i+1]&D[i+1];
uint64_t tmp2=C[i+2]&D[i+2];
uint64_t tmp3=C[i+3]&D[i+3];
A[i+0]=B[i+0]+(tmp0-tmp1);
A[i+1]=B[i+1]+(tmp2-tmp3);
```

(a) Source Code



(b) Use-def chains DAG



(c) SLP (2-wide only) (d) VW-SLP

Figure 4: Vectorization Width Widening.

Let's consider how VW-SLP will attempt to vectorize this code (see Figure 4(d)). Once again, vectorization starts at 2-wide mode until it reaches an instruction group with all operands of the same opcode which may be considered for widening. This group is the 2-wide group of subtractions. All of its operands are bit-wise and (&) operations which could potentially form a 4-wide vector. VW-SLP will consider both this new 4-wide vector and the default 2-wide ones and keep the best. In this example, switching to the 4-wide vectors proves profitable since all the loads are consecutive and the gains from vectorizing 4-wide more than make up for the additional +4 shuffling cost. According to the cost model, the total cost is -9, which is profitable to vectorize.

As expected, widening can also explore the permutations of the input at the widening/shuffling point, similarly to Section 3.2. This can make a big difference in the applicability of widening as it can improve its chances of succeeding.

4 VARIABLE-WIDTH SLP

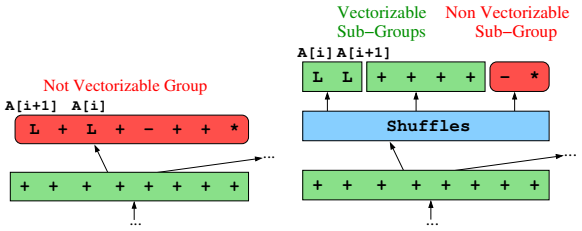
VW-SLP introduces several changes at the core of the SLP algorithm, the formation of the SLP-graph (the highlighted step 4 of Figure 1). As already shown in the examples of Section 3, the graph formation is critical for the effectiveness of the vectorizer as it is the step where the code's isomorphism is explored. The changes introduced by VW-SLP improve the algorithm's capability in extracting SIMD parallelism by adjusting to the vector width requirements of the code at an instruction granularity.

4.1 Shortening Strategy

While building the SLP-graph, SLP may reach instructions that cannot be grouped into a vectorizable group node (Listing 2, line 50). A common reason for this is that the opcodes do not match. At this point vanilla SLP will create a non-vectorizable group and will bail out (line 55).

Instead, VW-SLP considers such cases as opportunities for vector-width resizing, since there is a chance of finding isomorphism within a sub-group of these instructions. It is the job of `buildGraphVW()` (line 53) to create multiple narrower instruction subgroups (line 26) that could be potentially vectorized. If any of these narrower branches proves both vectorizable and profitable (lines 27 to 28), then the algorithm could potentially continue vectorizing beyond such points, with a narrower vector width (line 29), as shown in Figure 2(e).

Figuring out which sub-groups to create involves the following steps. First, the non-vectorizable instructions have to be grouped into vectorizable subgroups using the `createSubGroups()` function (Listing 2, line 26). The body of this function is not listed in the Listing 2, due to lack of space, but its operation is demonstrated in Figure 5. Given a group of instructions (Instrs), they are partitioned based on their opcodes into sub-groups as shown in Figure 5(c). Then the sub-groups are pruned. All sub-groups with illegal sizes are removed and the rest are shrinked down to the nearest power of two (or any number that is allowed by the target architecture). This results in zero or more vectorizable groups of smaller size, compared to the original group, and in a set of gathers for the remaining non-vectorizable scalar instructions. In Listing 2, line



(a) SLP (no shortening). (b) VW-SLP (shortening).

Opcodes	Sub–Groups
L	(L) A[i+1] (L) A[i]
+	(+) (+) (+) (+)
-	(-)
*	(*)

(c) VW-SLP formation of sub-groups.

Figure 5: Forming vectorizable sub-groups for shortening.

26, the sub-groups are returned and appended onto SubOps.

Shortening does not come for free, since depending on the actual data shuffling performed, it may require one or more additional instructions. In order to maximize the profitability of this step, VW-SLP can explore a number of possible permutations for each of the shorter groups[7] as shown in Listing 2, line 5 with the `getPermutations()` function.

Next, VW-SLP needs to perform a local performance evaluation of these options, that is: (*i.*) gathering, just like vanilla SLP, (*ii.*) forming shortened vectors, and (*iii.*) forming shortened vectors that are a permutation of the ones in the previous step. To that end, it temporarily continues building the graph for all candidates (with the loop of line 6) until a fixed exploration depth (lines 8 and 9), and runs the cost model on it (line 11). By comparing the costs returned for each case (lines 13 to 15), it selects the best option, flushes the temporary graph state (line 17), and continues building the graph as usual using the best instructions (lines 28 and 29). Please note that VW-SLP evaluates both the newly created variable-width graphs and the original SLP one. Therefore, VW-SLPis conceptually a superset of SLP.

A similar strategy is followed when instructions repeat within an instruction group, i.e., when the exact same instruction is present in more than one place in the group. Once again, several sub-groups are generated, including any gathers, and are evaluated with or without their permutations.

[7]We have a strategy for reducing the complexity. Please see Section 4.3.

Listing 2: VW-SLP algorithm for SLP-Graph

```
1  // Explores AllInstrs, and returns the one
2  // with the best cost.
3  getBestInstrs(AllOps, Depth) {
4   if (ExplorePermutations)
5     AllOps.append(getPermutations(AllOps, LIMIT))
6   for (TmpOps in AllOps)
7     // Build Graph temporarily until MaxDepth
8     MaxDepth = Depth + DEPTH_EXPLORATION_LIMIT
9     buildGraph(TmpOps, Depth)
10    // Run the cost model to get its cost.
11    Cost = getGraphCost()
12    // Keep the best cost.
13    if (Cost < MinCost)
14      MinCost = Cost
15      BestSubInstrs = TmpOps
16    // Discard the temporary state of the graph.
17    resetGraphState()
18  return BestSubInstrs
19 }
20
21 // VW-SLP-specific buildGraph()
22 // Returns true if we changed the vector width.
23 buildGraphVW(Instrs, Depth, Type) {
24  if (Type == SHORTEN) // Shortening
25   SubOps.append(Instrs)
26   SubOps.append(createSubGroups(Instrs))
27   BestSubOps = getBestInstrs(SubOps, Depth)
28   if (BestSubOps != Instrs)
29     buildGraph(BestSubOps, Depth+1)
30     return true
31  if (Type == WIDEN) // Widening
32   WidenedOps.append(Instrs)
33   WidenedOps.append(createWidenedGroup(Instrs))
34   BestWidenedOps = getBestInstrs(SubInstrs)
35   if (BestWidenedOps != Instrs)
36     buildGraph(BestWidenedOps, Depth+1)
37     return true)
38  return false
39 }
40
41 // Entry point for building the SLP-Graph.
42 // In:  The vector of seed instructions
43 // Out: The SLP-graph
44 void buildGraph(Instrs, Depth) {
45  // If we reached the depth limit, return
46  if (Depth > MaxDepth)
47    createNonVectorizableGroup(Instrs)
48    return
49  // Group of Instrs is not vectorizable
50  if notVectorizable(Instrs)
51    // Try Shortening
52    if (VW-SLP
53        && buildGraphVW(Instrs, Depth, SHORTEN))
54      return
55    createNonVectorizableGroup(Instrs)
56    return
57  // Try Widening
58  if (VW-SLP && areBinaryInstructions(Instrs)
59      && buildGraphVW(Instrs, Depth, WIDEN))
60    return
61  // Vanilla SLP recursion
62  createVectorizableGroupFor(Instrs)
63  for (Operands in Instrs' operands)
64    buildGraph(Operands, Depth+1)
65 }
```

4.2 Widening Strategy

VW-SLP attempts to widen the vector width at each vectorizable node of binary instructions, whenever it is both possible and profitable compared to vanilla SLP. After creating a group node of binary operations, VW-SLP will attempt to flatten all the instructions of the group and continue with a 2x

(a) SLP (no widening) (b) VW-SLP (widening)

Figure 6: Widening of both operands of the group into a single wider vector.

wider vector. For this to be legal, all the opcodes of the wider vector need to be identical (and should obviously be allowed to be scheduled together). When this is the case, VW-SLP proceeds with either: (*i.*) this new wider vectorizable group, (*ii.*) one of its permutations, or (*iii.*) the original groups. VW-SLP explores all three cases and keeps the most profitable one using the exact same method as in Section 4.1 (Listing 2 lines 32 to 34). With the best vectorizable group in `BestWidenedOps`, the construction of the SLP graph resumes (line 37).

For widening to be profitable, vectorization should succeed for at least a few more nodes above the new wider vector. This is to amortize the cost of the wide data shuffling. As expected, and as shown in Section 5, widening does not succeed as often as shortening.

4.3 Complexity

VW-SLP adds more complexity to the existing SLP algorithm in two ways:

1. As already discussed in Sections 4.1 and 4.2, the evaluation of profitability for both shortening and widening is based on a local backtracking strategy. That is, we are temporarily extending the SLP-graph (Listing 2 line 9) using either wider or shorter vectors, and then we compare the cost of the graph against the default scheme without variable-width vectors. If the new cost is better, we discard the current state. We can limit the complexity added by this backtracking in two ways. (*i.*) By building a temporary SLP-graph of fixed maximum depth (line 8). The reasoning behind this is that bad graphs fail early. (*ii.*) By limiting the number of nested explorations to a very small number. The reasoning is that nesting of shortening and widening is not that common.

2. Exploring the permutations at the shuffling points adds yet another dimension of complexity. This exploration is particularly critical for compilation-time as it also works in a backtracking fashion to select the best permutation possible. However, the complexity can be controlled by limiting the exploration space as follows: (*i.*) By gradually reducing the number of permutations we evaluate at each exploration (Listing 2 line 5), we can allow for more exploration towards the root of the graph and less towards the leaves. The intuition is that the decisions made close to the root are more critical than those made at the leaves. (*ii.*) We can limit the number of the nested (outstanding) permutation explorations that we are evaluating. The intuition behind this is that nested explorations provide diminishing returns. (*iii.*) We can impose a global limit the overall number of permutations that are explored to a fixed number.

By controlling these knobs, the user has full control over the trade-off between exploration time and performance benefits. We have empirically found that reducing the permutation explorations from 8 to 0 as the depth increases, using the formula $(8 - depth)$, is a good trade-off. Similarly, disabling both nested permutation explorations and nested widening/shortening explorations is not harming performance significantly. For more insights into how these knobs affect compilation time and performance, please refer to Section 5.2, where we show a detailed breakdown of both performance and compilation time while varying some of these tuning knobs.

5 RESULTS

We implemented VW-SLP in the development branch of LLVM 7 as an extension to the existing SLP vectorizer. We compiled the workloads with the following configurations: (*i.*) O3, which corresponds to *-O3* with all vectorizers disabled, (*ii.*) SLP, which is *-O3* with only the SLP vectorizer enabled (this is the state-of-the-art SLP), and (*iii.*) VW-SLP, which is *-O3* with only the VW-SLP algorithm enabled instead of SLP.

All C/C++ workloads were compiled with `clang` using *-O3 -march=skylake -mtune=skylake*. The fortran benchmarks were compiled with the GCC fortran front-end with the help of the DragonEgg [1] project. The loop vectorizer was disabled across all configurations. The target platform is a Linux-4.9.0, glibc-2.23 based system with a 6th generation Intel® Core™ i5-6440HQ CPU and 8 GB RAM. We evaluated our approach on unmodified SPEC CPU2006 and CPU2017 [39]. We also evaluated VW-SLP on TSVC[7] and NAS[6]. All TSVC kernels shown were manually unrolled. We applied code massaging on s319 to expose the 4-wide reduction seed SLP, and for s123 we assume that the loop condition can be statically optimized to true. We are not showing

Table 1: Kernels used for evaluation.

Kernel	Benchmark	Filename:Line
444-lattice-recalc	SPEC'06 444.namd	Lattice.H:267
435-inl3220	SPEC'06 435.gromacs	innerc.c:18895
435-inl3330	SPEC'06 435.gromacs	innerc.c:19210
435-inl3430	SPEC'06 435.gromacs	innerc.c:21238
464-satd8x8	SPEC'06 464.h264ref	mv-search.c:1118
464-find-sad16x6	SPEC'06 464.h264ref	macroblock.c:4226
h264enc-dct-chroma	MediabenchII h264enc	block.c:1517

Table 2: Variants used for evaluation.

Variant	Description
O3	-O3 (No vectorizer enabled)
SLP	-O3 + SLP (No loop vectorization)
VW-SLP-S	Only Shortening (No Permutation)
VW-SLP-SP	Only Shortening with Permutation
VW-SLP-W	Only Widening (No Permutation)
VW-SLP-WP	Only Widening with Permutation
VW-SLP	VW-SLP-WP + VW-SLP-SP
VW-SLP-NESTx	VW-SLP + nested width explorations
VW-SLP-PERMx	VW-SLP + nested permutations
VW-SLP-NESTx-PERMy	VW-SLP-NESTx + VW-SLP-PERMy

the few benchmarks that did not trigger VW-SLP at all, as they perform exactly the same as SLP. We also extracted whole functions (kernels) from SPEC CPU2006 and Mediabench II [12] (Table 1) to help focus on code that triggers VW-SLP. In the kernel results we included the motivating examples of Section 3 for completeness. For all results, we report the average of 10 executions, after skipping the first warm-up run. The error bars show the standard deviation. The static cost we report is LLVM's TTI-based cost (see Section 2.2).

5.1 Performance

In order to provide more insights into the individual contribution of each proposed technique, we provide a full breakdown. The variants we used are described in Table 2. The performance breakdown is shown in Figure 7, while the sum of the static cost collected by the vectorizer is reported in Figure 8.

For all the VW-SLP results reported we use: i) no nesting of the shortening/widening techniques (only one allowed per graph), ii) no nested permutation exploration, iii) a maximum of 8 permutations per permutation point, and iv) an exploration depth limit of 12.

On average, the variants of VW-SLP do improve performance compared to SLP. The most effective variant tends to be the shortening one. This is expected, as it is easier to find small sets of isomorphic instructions compared to a large set, i.e, shortening is easier than widening. Nevertheless, 464-find-sad16x16 improves explicitly due to widening.

Permutation is beneficial for 464-satd8x8, suggesting that: (*i.*) it is common for vectorization to stop

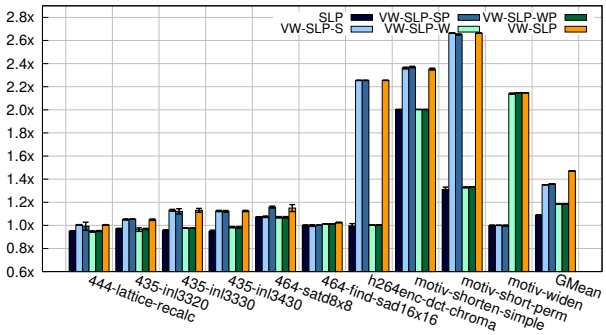


Figure 7: Speedup Normalized to O3.

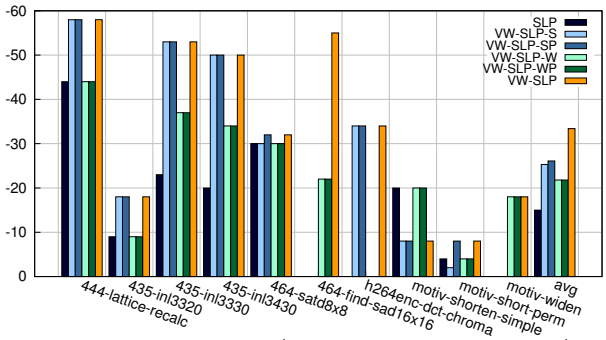


Figure 8: Static Cost (the higher the better).

because of only some of the lanes becoming non-isomorphic, and (*ii.*) the additional overhead of the permutation is hard to get amortized, except for large graphs.

5.1.1 Full Benchmarks. The VW-SLP algorithm gets triggered numerous times across several benchmarks in the SPEC 2006 and 2017 suites, as shown in Figures 9 and 11. When triggered, it improves the cost in the majority of benchmarks, as shown in Figures 9 and 11. On average, the benchmarks get a cost improvement of almost 20%, while many of them get a lot more than that. A notable exception is 456.hmmer, where VW-SLP has a cost of -181, which is worse than SLP's -221. Such cost regressions are possible because the algorithm is greedy by design. If VW-SLP improves the cost of one region, it will accept this solution even though that may harm vectorization for a future region. More holistic approaches would fix such issues, at the expense of higher compilation time.

Whether these static cost improvements lead to run-time performance improvements depends on whether the code that got vectorized was in a hot execution path. As shown in Figures 10 and 12, both SPECint and SPECfp improve with VW-SLP. Integer workloads that operate on RGB data are usually good candidates for SLP-style algorithms, therefore they can potentially improve further with the help of

VW-SLP. There are several benchmarks from both the floating-point and the integer suites that improve considerably, such as 447.dealII and 525.x264_r. It is interesting to note that even though 456.hmmer had a worse static score, because VW-SLP favored one code region over a future one, this turned out to be beneficial for performance. On the other hand, 444.namd and 433.milc had worse performance, even though their scores were increased by about 4% and 28%, respectively. Such discrepancies between the static cost and the actual performance are expected due to the inaccuracies of the cost model. Overall, VW-SLP provides significant improvements over SLP.

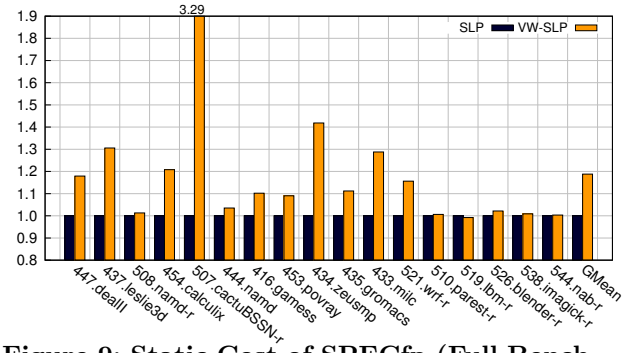


Figure 9: Static Cost of SPECfp (Full Benchmarks) normalized to SLP (the higher the better).

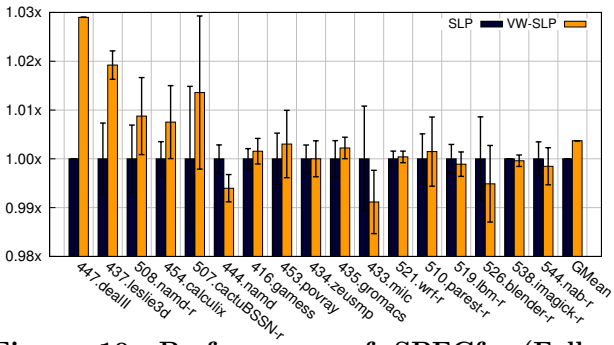


Figure 10: Performance of SPECfp (Full Benchmarks) Normalized to SLP.

5.1.2 TSVC and NAS. Our evaluation on these smaller benchmarks shows that VW-SLP can achieve larger speedups. Please note that, just like in Section 5.1.1, we are only showing those cases where VW-SLP triggered at least once, as for the rest of them both VW-SLP and SLP perform exactly the same.

The plot of static cost in Figure 13 shows that VW-SLP improves benchmarks from both suites. In case of s123 and s127 SLP is unable to vectorize

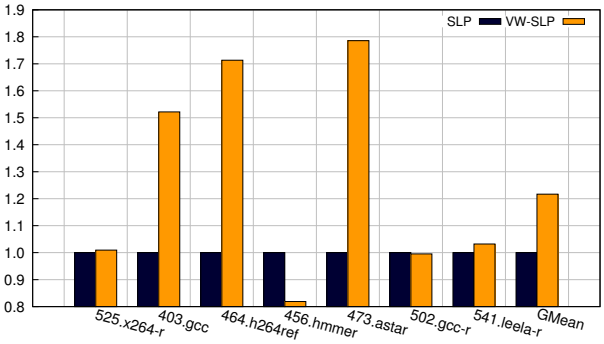


Figure 11: Static Cost of SPECint (Full Benchmarks) normalized to SLP (the higher the better).

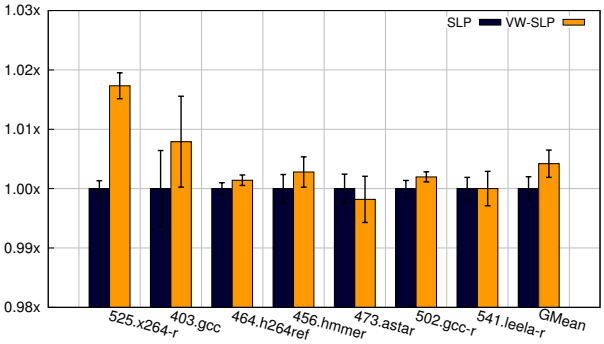


Figure 12: Performance of SPECint (Full Benchmarks) Normalized to SLP.

the code at all (leading to infinite normalized cost improvement). However, as Figure 14 shows, since the tests in TSVC are much smaller, the speedups that we get are far greater. Both BT and LU from the NAS suite show static cost improvements with VW-SLP compared to SLP. However, BT shows a 3% improvement while LU performs the same as SLP, since the optimized code was not in a *hot* code path.

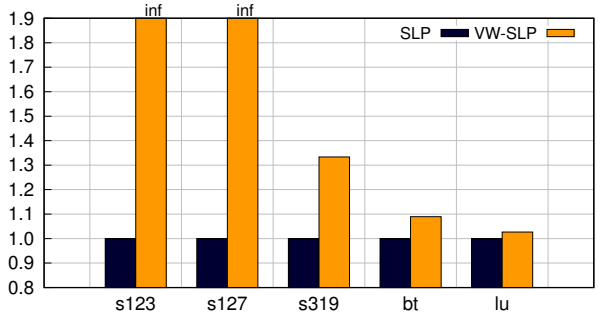


Figure 13: Static Cost of TSVC and NAS normalized to SLP (the higher the better).

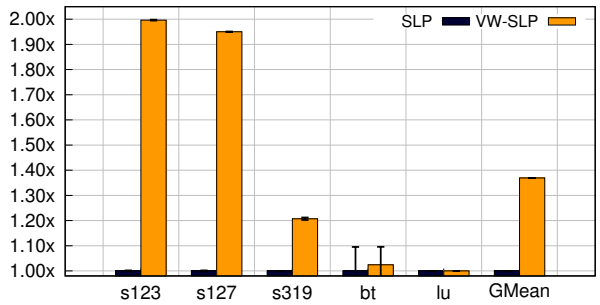


Figure 14: Performance of TSVC and NAS normalized to SLP.

5.2 Compilation Time

We measured the compilation wall time, normalized it to O3, and showed the results in Figure 15. The kernel compilation time tests show the worst-case scenario, since the ratio between "code with SLP opportunities" and "total code" is very high. When compiling full benchmarks, the compilation time increase is usually insignificant.

SLP has an overall overhead of about 14% over O3, while VW-SLP about 31% across the kernels. We observe that when the overhead of SLP over O3 is high, then the overhead of VW-SLP over SLP is also high. This is expected because the more time the compiler spends in SLP, the higher the number of non-vectorized groups encountered, and therefore the more the exploration that VW-SLP needs to perform.

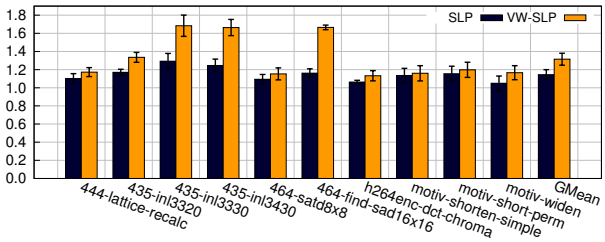


Figure 15: Normalized Compilation time.

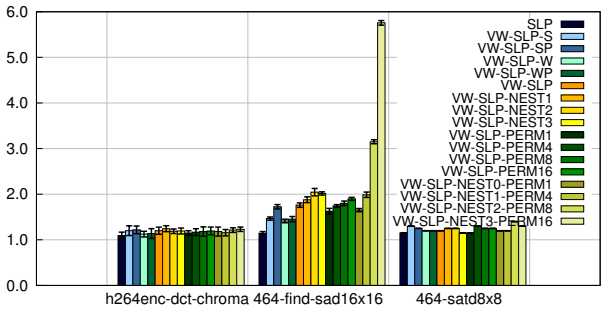


Figure 16: Compilation time breakdown normalized to O3.

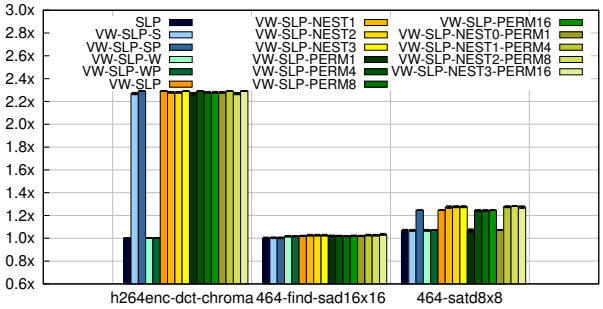


Figure 17: Performance breakdown normalized to O3.

Compilation time can be traded off for more complete exploration (and hopefully better performance), by adjusting one of the several knobs that control VW-SLP. In Figures 16 and 17 we show a more complete breakdown of the compilation time and the corresponding performance for three of the workloads. We varied two of the exploration parameters to provide more insights into the trade-offs involved. The first one is the maximum nesting allowed for explorations (that is whether we allow an exploration for shortening or widening, while a previous one is already ongoing, and how many such overlapping explorations are allowed). It is marked with the suffix `-NESTx`. Please note that `VW-SLP` is equivalent to `VW-SLP-NEST0`. The second one is the maximum number of permutations allowed per permutation point, and is marked with the `-PERMx` suffix. Finally we enabled both at once (`-NESTx-PERMy`).

The compilation time overhead (Figure 16) varies depending on the workload. Some workloads, like 464-find-sad16x16, can consume up to 5.8x more compilation time if we allow many levels of nested explorations for vector-width resizing along with a large number of permutations. However, as shown in Figure 17, the additional compilation time for this workload has diminishing performance returns. We empirically found that no nesting and 8 permutations provide a good trade-off.

6 RELATED WORK

High Performance Computing (HPC) has relied on vector machines to accelerate HPC workloads for several decades, while scientific workloads have been accelerated by both commercial [28, 36] and experimental [16, 28, 36] vector machines. General purpose CPUs have also adopted vectorization technology through the use of short SIMD vector instructions [15]. Graphics processors (GPUs) [20] implement similar vector datapaths for achieving high throughput. Wide parallel computation on GPUs is possible thanks to data-parallel graphics APIs (e.g., OpenGL [38], DirectX [13]) or languages like

CUDA [27] or OpenCL [24], where the programmer explicitly exposes the available parallelism.

6.1 Loop Auto-Vectorization

Auto-vectorization techniques have traditionally focused on vectorizing loops [41]. The basic implementation conceptually strip-mines the loop by the vector factor and widens each scalar instruction in the body to work on multiple data elements. The effectiveness of loop vectorizing compilers has been studied by Maleki et al. [22]. Many fundamental problems of loop vectorization have been addressed by early work on the Parallel Fortran Converter [2, 3] and others [9, 17, 40]. Since then, numerous improvements to the basic algorithm have been proposed in the literature and production compilers [4, 10, 25, 26, 34].

6.2 SLP Auto-Vectorization

SLP Vectorization was introduced by Larsen and Amarasinghe [18]. It is a complementary technique to loop vectorization which focuses on vectorizing straight-line code instead of loops. Similar straight-line code vectorization algorithms have been implemented in compilers such as GCC [11] and LLVM, with bottom-up SLP (Rosen et al. [35]) being widely adopted due to its low run-time overhead and its good coverage. In this paper we use the LLVM implementation of this state-of-the-art bottom-up SLP algorithm as the baseline.

Since its original work, several improvements have been proposed for the straight-line-code (SLP-style) vectorization. Shin et al. [37] propose an SLP-based framework that makes use of predicated execution to convert the control flow into data-flow dependence, thus allowing it to become vectorized by a straight-line code vectorizer. Liu et al. [21] present a vectorization framework that generates vectorizable groups by exploring pairs of vectorizable instructions and forming vectors out of the most profitable ones. [23] improves upon this algorithm with an ILP solver to better explore the space, which results in better performance, but with impractically long compilation times. Huh and Tuck [14] propose a different approach for identifying isomorphism in SLP vectorization based on growing the vectorizable graph from small predefined patterns. The Park et al. [29] approach succeeds in reducing the overheads associated with vectorization such as data shuffling and inserting/extracting elements from the vectors.

The widely used bottom-up SLP algorithm has been improved in several ways. Porpodas et al. [32] propose PSLP, a technique that pads the scalar code with redundant instructions, to convert non-isomorphic instruction sequences into isomorphic ones, thus extending the applicability of SLP. Just like VW-SLP-S, PSLP can vectorize code when some of the lanes differ, but it is most effective when the non-isomorphic parts are only a short section of the instruction chain. VW-SLP, on the other hand, works even if the chain never converges. In [31], the SLP region is pruned to scalarize groups of instructions that harm the vectorization cost, while in [30] a larger unified SLP region is used, that overcomes limitations associated with the inter-region communication and unreachable instructions. Zhou et al. [43] present a vectorization technique that reduces the data re-organization overhead by considering both intra- and inter-loop parallelism, that improves upon the loop-aware SLP approach of [35], while in [42] vectorization is enabled for SIMD widths that are not supported by the target hardware. Finally, Look-Ahead SLP [33] proposes an improved SLP algorithm that focuses on better vectorizing chains of commutative operations. It introduces the concept of the *multi-node* and an improved operand reordering heuristic based on knowledge collected by scanning the code beyond the operations being considered for vectorization.

VW-SLP is orthogonal to these techniques. It is the first algorithm, that we are aware of, that introduces a more powerful exploration of the underlying IR by adapting the vector width to the requirement of the input at an instruction granularity.

7 CONCLUSION

We presented VW-SLP, a bottom-up SLP-based auto-vectorization algorithm that can adjust the vector width at an instruction granularity. This allows the algorithm to vectorize at either wider or narrower widths within the same block of code, as dictated by the available vector parallelism. The end result is a more powerful code exploration for isomorphic instructions, and therefore increased vectorization coverage and better performance compared to the state-of-the-art. VW-SLP was implemented in LLVM and was evaluated on wide-range of benchmarks on a real machine. The performance results show that it improves performance considerably without a prohibitive compilation-time overhead.

ACKNOWLEDGMENTS

Rodrigo C. O. Rocha is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/L01503X/1.

REFERENCES

[1] DragonEgg: Using LLVM as a GCC backend. http://dragonegg.llvm.org.

[2] J. R. Allen and K. Kennedy. PFC: A program to convert fortran to parallel form. Technical Report 82-6, Department of Mathematical Sciences, Rice University, 1982.

[3] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *Tranactions on Programming Languages and Systems (TOPLAS)*, 9(4), 1987.

[4] A. Anderson, A. Malik, and D. Gregg. Automatic vectorization of interleaved data revisited. *ACM Trans. Archit. Code Optim.*, 12(4):50:1–50:25, Dec. 2015.

[5] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences: A method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 1994.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 1991.

[7] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Supercomputing'88.[Vol. 1]., Proceedings.*, pages 98–105. IEEE, 1988.

[8] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[9] J. Davies, C. Huson, T. Macke, B. Leasure, and M. Wolfe. The KAP/S-1- an advanced source-to-source vectorizer for the S-1 Mark IIa supercomputer. In *Proceedings of the International Conference on Parallel Processing*, 1986.

[10] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[11] Free Software Foundation. GCC: GNU compiler collection. http://gcc.gnu.org, 2015.

[12] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. MediaBench II video: Expediting the next generation of video systems research. *Microprocessors and Microsystems*, 2009.

[13] K. Gray. *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[14] J. Huh and J. Tuck. Improving the effectiveness of searching for isomorphic chains in superword level parallelism. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 718–729, New York, NY, USA, 2017. ACM.

[15] Intel Corporation. IA-32 Architectures Optimization Reference Manual, 2007.

[16] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9), 1997.

[17] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Symposium on Principles of Programming Languages*, 1981.

[18] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[20] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2), 2008.

[21] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[22] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[23] C. Mendis and S. Amarasinghe. goSLP: Globally Optimized Superword Level Parallelism Framework. *arXiv preprint arXiv:1804.08733*, 2018.

[24] A. Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[25] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[26] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[27] C. Nvidia. Compute unified device architecture programming guide. 2007.

[28] W. Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8), 1992.

[29] Y. Park, S. Seo, H. Park, H. Cho, and S. Mahlke. SIMD defragmenter: Efficient ILP realization on data-parallel architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[30] V. Porpodas. SuperGraph-SLP Auto-Vectorization. In *2017 International Conference on Parallel Architecture and Compilation (PACT)*, pages 330–342. IEEE, 2017.

[31] V. Porpodas and T. M. Jones. Throttling automatic vectorization: When less is more. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 432–444. IEEE, 2015.

[32] V. Porpodas, A. Magni, and T. M. Jones. PSLP: Padded SLP automatic vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2015.

[33] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. Look-ahead slp: Auto-vectorization in the presence of commutative operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 163–174, New York, NY, USA, 2018. ACM.

[34] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[35] I. Rosen, D. Nuzman, and A. Zaks. Loop-aware SLP in GCC. In *GCC Developers Summit*, 2007.

[36] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1), 1978.

[37] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.

[38] D. Shreiner. *OpenGL reference manual: The official reference document to OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[39] SPEC. Standard Performance Evaluation Corp Benchmarks. http://www.spec.org, 2014.

[40] M. Wolfe. Vector optimization vs. vectorization. In *Supercomputing*. Springer, 1988.

[41] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[42] H. Zhou and J. Xue. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[43] H. Zhou and J. Xue. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 59–69. ACM, 2016.