SuperGraph-SLP Auto-Vectorization

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

***Abstract*—SIMD vectors help improve the performance of certain applications. The code gets vectorized into SIMD form either by hand, or automatically with auto-vectorizing compilers. The Superword-Level Parallelism (SLP) vectorization algorithm is a widely used algorithm for vectorizing straight-line code and is part of most industrial compilers. The algorithm attempts to pack scalar instructions into vectors starting from specific seed instructions in a bottom-up way. This approach, however, suffers from two main problems: (i) the algorithm may not reach instructions that could have been vectorized, and (ii) atomically operating on individual SLP graphs suffers from cost overestimation when consecutive SLP graphs share data. Both issues lead to missed vectorization opportunities even in simple code.**

In this work we propose SuperGraph-SLP (SG-SLP), an improved vectorization algorithm that overcomes these limitations of the existing algorithm. SG-SLP operates on a larger region, called the SuperGraph. This allows it to reach and successfully vectorize code that was previously unreachable. Moreover, the new region helps eliminate the inaccuracies in the cost-calculation as it allows for a more holistic view of the code. Our experiments show that SG-SLP improves the vectorization coverage and outperforms the state-of-the-art SLP across a number kernels by 36% on average, without affecting the compilation time.

***Keywords*-SIMD; Vectorization; SLP**

I. Introduction

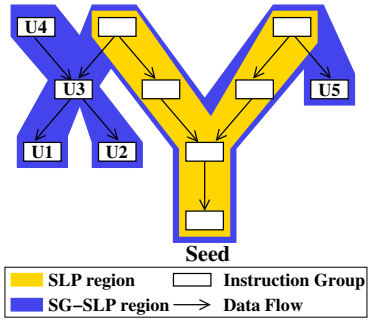
All modern high-performance general purpose processors support short SIMD (Single Instruction Multiple Data) vector instructions in their instruction sets. It allows them to boost performance in applications with vector computations, (usually scientific or signal-processing workloads). These vector instructions perform the same operation as multiple scalar instructions in fewer cycles while using less energy. Vector ISAs are being improved regularly, supporting wider data types (e.g., 512 bits in Intel's AVX-512) and a larger variety of opcode types.

Making use of the SIMD units, however, is not a trivial task. Software developers must explicitly express the vector parallelism in their code, taking into consideration the capabilities of the target platform. This is not ideal for several reasons: (i) The code may not be portable to other target architectures if compiler intrinsics are used. (ii) Even if the code is portable, its performance may not be. (iii) It is harder to program, more error prone and time consuming. Alternatively the developers can rely on an auto-vectorizing compiler to convert the regular scalar code into vector code. This is the preferred method for most software projects, except perhaps for highly tuned kernels in high performance libraries.

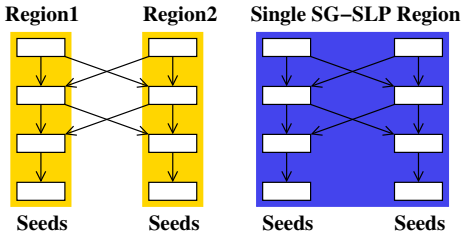
SLP [32] is the state-of-the-art straight-line code auto-vectorizing algorithm and has been implemented in several compilers, including GCC [10] and LLVM [18]. This algorithm is based on concepts from the original Super-word Level Parallelism paper [17]. It works by first scanning the code for scalars that can become the seeds of vectorization (usually consecutive stores). Once found they are grouped together to form the first potentially vectorizable group. Then, SLP walks up the data-flow (towards the definitions), attempting to group the predecessors together, as long as they can be potentially vectorized. After collecting all these groups of instructions (of the same opcode), SLP checks whether converting the grouped scalar instructions into vectors is better for performance than keeping them scalar. The cost calculation factors in the costs of gathering/scattering data into/out of the vector registers.

SLP has been designed with computational complexity in mind. The major design decision that contributes to the fast run-time of this algorithm and thus its wide adoption by industrial compilers is the concept of *seed instructions*. Even though any potential group of scalars could be vectorized, the widely adopted SLP algorithm will only consider groups that are rooted at specific seeds (usually consecutive stores and reductions). The SLP algorithm will then only explore instructions that are data-flow connected to the seeds. Exploring all potential vectorizable sets of scalars in the code is computationally unaffordable. Although this design limits the vectorization coverage, it reduces the complexity, allowing the algorithm to be used in production compilers. Compiler developers consider that the compile-time complexity of industrial compilers is of utmost importance. This is true to such an extent that for example in LLVM's [18] implementation of SLP there are potential groups of seed instructions that have been deliberately left out and are not even explored (32 x int8 for AVX2). This is done in an attempt to limit the computational complexity, even though this limits the vectorization coverage.

In this work we identify a major limitation of the SLP algorithm in the way it forms and explores the SLP graph starting from the given seeds. The state-of-the-art SLP algorithms will only build the SLP graph in a bottom-up way (from the uses to the definitions) and operate on it atomically



(a) SG-SLP region reaches previously unreachable instructions.



(b) SLP forms multiple fragmented regions leading to cost over-estimation. SG-SLP's single region solves the problem.

Figure 1. SG-SLP overcomes SLP limitations.

without considering its future neighboring (and potentially vectorizable) graphs. In this process, it: (i) conservatively considers the side-uses (side-exits from the SLP graph) as external uses of scalar code, and (ii) does not attempt to explore vectorization on any other path beyond the basic bottom-up region. This strategy leads to two significant problems (illustrated in Figure 1):

1) There may well be instructions that could be potentially vectorized but remain unreachable if there is no descending data-flow path from them to some seed. In Figure 1(a) the groups from U1 to U5 are unreachable by SLP. Allowing seeds to form at more instructions could fix this issue, but it has prohibitively high complexity.
2) The additional cost introduced by the side-uses could potentially render the code non-profitable for vectorization, even though in reality it may be perfectly vectorizable if considered together with the neighboring graphs. There is no easy fix for this, as the algorithm cannot foresee the future in order to tell whether the side-uses will be vectorized or not. In Figure 1(b) if the cost of side-entrances and exits is high enough, SLP will fail to vectorize both Region1 and Region2.

We explain both of these problems in more detail and provide insightful examples in Section III.

SuperGraph-SLP (SG-SLP) solves both problems. It is a novel SLP-based automatic vectorization algorithm that operates on a larger region compared to SLP (referred to as the SuperGraph). The SG-SLP supergraph region is not only a superset of several neighboring SLP graphs, but it also extends to instructions unreachable by SLP. Initially the supergraph forms bottom-up, but it will also attempt to extend downwards from successfully grouped instructions. The supergraph enables us to vectorize parts of the code that were either: (i) unreachable by vanilla SLP due to its limited region (see Section III-B), or (ii) non-vectorizable due to the over-estimation of the SLP cost (see region fragmentation Section III-C). SG-SLP results in improved vectorization coverage and better performance, with little impact on the execution time of the algorithm, which renders it suitable for use in industrial compilers.

II. Background

SG-SLP builds upon the state-of-the-art SLP automatic vectorization algorithm. In this section we briefly present the state-of-the-art of automatic vectorization in modern compilers with the main focus on the operation of SLP.

A. Automatic Vectorization Algorithms

Modern compilers have two distinct types of automatic vectorization algorithms:

1) Loop-based algorithms (e.g., [23], [24]). These require that: (i) the loop has well defined induction variables, usually affine (ii) all inter- and intra-loop dependencies are statically analyzable (or dynamically evaluated for multi-versioning). Consecutive loop iterations are fused together into a single vectorized iteration in a strip-mining fashion.
2) Straight-line code algorithms, the most widely used being Superword-Level Parallelism (SLP, e.g., [17], [32]). Their main features are that: (i) they operate on straight-line code, not loops (ii) they can even vectorize code within non-vectorizeable loops

Straight-line code algorithms scan the code for repeated sequences of isomorphic scalar instructions. Instructions of the same type are grouped together and replaced by a single vector instruction. The name Super-Word Level Parallelism (SLP) refers to vector parallelism for short vector ISAs (e.g., Intel's AVX2). This is to differentiate it from traditional vector processing on machines of the past (with vector widths in the tens or hundreds [27], [33]).

Although the SLP algorithm could be considered as a superset algorithm of broader scope compared to the loop-based vectorizer [17], in practice this is not the case. The algorithms are complementary and industrial strength compilers (e.g., GCC and LLVM) implement both algorithms. A common configuration is to run the SLP pass after the loop-vectorization pass.

B. SLP Vectorization

The most widely used straight-line code vectorizer is the Superword-Level Parallelism algorithm (SLP) [32]. It

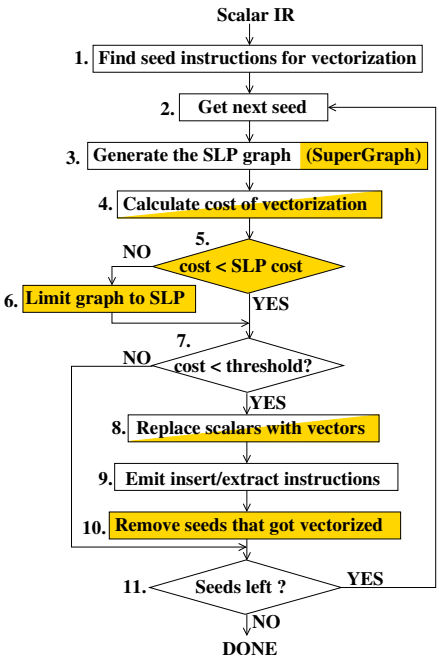


Figure 2. Overview of the SLP algorithm. The highlighted sections belong to the SG-SLP algorithm.

scans the compiler's intermediate representation, identifying specific type of instructions, referred to as *seeds*. The seeds are instructions of a small subset of the instruction set, 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 through the code above the seeds, following the data-flow, to form the rest of the vectorizable groups.

The main difference of SLP from traditional vectorization techniques is that SLP does not operate on loops, making it more generally applicable than loop-based vectorizers, e.g., on loop-unrolled code or within basic-blocks of the non-vectorizable loops. The code to be vectorized can span multiple basic blocks, as long as each group of instructions to be vectorized belongs to the same basic block.

A high-level overview of the SLP algorithm is shown in Figure 2 (the highlighted parts have been added by SG-SLP). The SLP algorithm first scans for vectorizable seed instructions (step 1), which are instructions of the same type and bit width that are either: (i) non-dependent memory instructions that access adjacent memory locations (scalar evolution analysis [7], [4], [37] is commonly used to test for this); (ii) instructions that form a reduction tree (e.g., a reduction tree of additions). Adjacent memory instructions are the most promising seeds and therefore most compilers look for these first [32].

The algorithm then grabs a seed from the seed list (step 2) and starts to build the SLP graph (step 3). Building the SLP graph involves forming groups of potentially vectorizable instructions by following the data dependence graph that starts at the seed instructions. The state-of-the-art method for generating the graph is to start from store seed instructions and build the graph from the bottom-up. This is the approach followed in both GCC's and LLVM's SLP vectorizers [32]. Each group contains the scalar instructions that are candidates for vectorization, but it also carries some additional data such as the group's cost (see next step). Once the algorithm encounters scalar instructions that cannot form a vectorizable group it forms a final non-vectorizable group which will carry the cost of collecting the data from scalars and inserting them into the vector. At this point the algorithm stops exploring the code in this direction as this path cannot be vectorized any further.

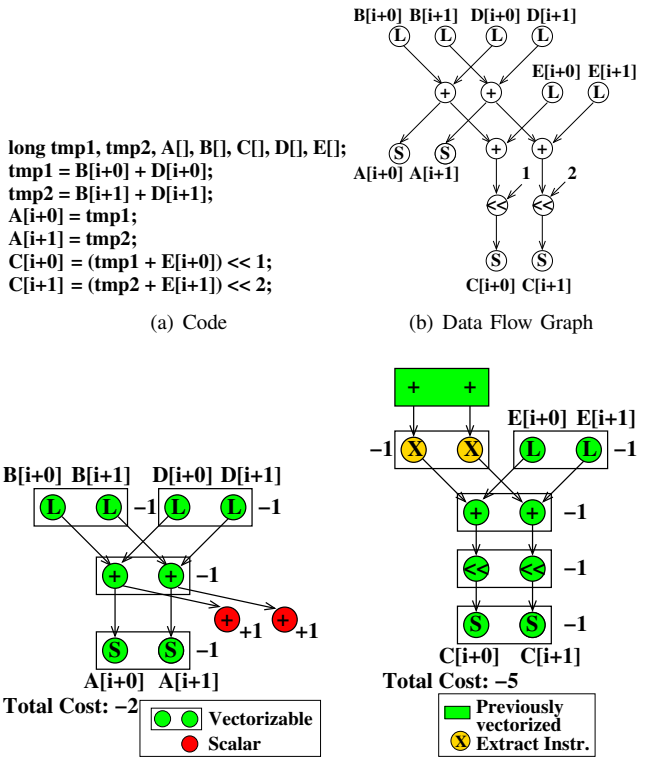
Once the graph has been constructed, SLP estimates the code's performance (step 4). This is done with the help of the compiler's target-specific cost model. The group's cost is equal to the savings from converting each group of scalar instructions into vector form (negative numbers mean better performance). The cost of the whole graph is equal the sum of the group costs (the lower the cost the better). The graph cost is compared against a threshold (usually 0) to determine whether vectorization should proceed (step 7). If so, the compiler modifies the intermediate representation code by replacing the groups of scalar instructions with their equivalent vector instructions (step 8), and emits any *insert* or *extract* instructions required for the flow of data between the vector and scalar instruction (step 9). If, however, the cost of vectorization is higher than that of the scalar code, then the code remains unmodified. This process repeats for all the seed instructions that the SLP front-end has collected (step 11).

III. Motivation

This section motivates SG-SLP with the help of three examples that highlight the weaknesses of the existing SLP algorithm, while demonstrating how SG-SLP overcomes them. We show that the fragmented region of the original SLP algorithm is causing vectorization opportunities to be missed. SG-SLP introduces a larger vectorization region (the supergraph) which allows the algorithm to successfully vectorize the code.

A. SLP Builds and Vectorizes Standalone Regions

SLP is a bottom-up algorithm since the SLP graph grows from the seeds upwards following the instruction data flow predecessors. The front-end of SLP scans the code for the appropriate seeds and, groups them together. Then the core of the algorithm (Figure 2 steps 3 to 10) builds a graph rooted at each seed group. In other words, SLP will build a standalone SLP graph for each group of seeds. For example consider the code of Figure 3(a), with the corresponding data-flow graph of Figure 3(b). A single run of the core of the algorithm will build the SLP graph of Figure 3(c) for



(a) Code (b) Data Flow Graph

(c) SLP graph 1. Seeds: A[i+0],A[i+1] (d) SLP graph 2 for seeds C[i+0],C[i+1]

Figure 3. Example 1: SLP successfully vectorizes both SLP graphs.

the A[i+0] and A[i+1] seeds, and another run will build the graph in Figure 3(d). This is a fragmented view of the vectorized region. Nevertheless, it usually succeeds, as shown in the example of Figure 3 and as discussed below.

Each SLP graph grows upwards forming potential vector groups (represented by rectangular boxes in Figure 3), until it encounters either loads, or instructions that cannot form a vectorizable group (e.g., instructions of different opcodes, instructions that have no vectorizable form in the target ISA, or memory non-consecutive loads).

Each potentially vectorizable group has a cost value associated with it (shown on the side of the instruction groups) of Figure 3). This cost represents the added cost of vectorizing the instruction in the group compared to the cost if they remained scalar. A negative cost means that vectorization is profitable. For example, vectorizing the stores to A[i+0] and A[i+1] in Figure 3(c) has a cost of -1, which means that the vectorized stores have a lower cost than the two scalar stores.

The SLP graph is treated as a single atomic vectorization entity. All instructions within the graph are to be vectorized, while flow of data into or out of the graph is considered external input or output. Edges to/from code outside the SLP graph: (i) are specifically tagged and (ii) introduce additional cost because of the additional instructions required to transfer the data between scalars and vectors. For example, the SLP graph of Figure 3(c) has two external uses: the additions that lead to the stores to C[i+0] and C[i+1], each of which requires an extract instruction which introduces a cost of +1 each.

The decision on whether or not to vectorize the code of the SLP graph is made atomically for that graph alone, by calculating the cost of the whole SLP graph and comparing it to a threshold value, (usually 0). Only graphs with a negative total cost are to be vectorized. In the example of Figure 3(c), the SLP graph has a cost of -2, and therefore vectorization is profitable. Once the graph is considered vectorizable, its scalar instructions in the intermediate representation are replaced with their vectorized counterparts.

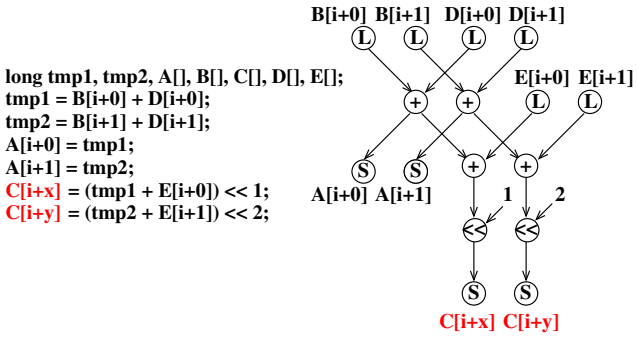
After the graph of Figure 3(c) is vectorized and its intermediate representation code of the corresponding scalars has been converted into vectors, SLP moves to the next seeds (as shown in steps 11 and 2 of Figure 2) and builds a graph starting from the new seeds. It therefore builds the graph rooted in C[i+0] and C[i+1], as shown in Figure 3(d). The graph includes the store seeds (C[i+0], C[i+1]), the left shifts ($\ll$), the additions, the loads from E[i+0],E[i+1], the extract instructions (X) and finally the vector addition of the previously vectorized graph (the top left vectorized node corresponds to the group of additions of Figure 3(c)). The total cost is -5, as all instructions, including the group of extracts are vectorizable [1] . The end result is that even though SLP formed two distinct graphs for the seeds and attempted to vectorize them in isolation, the whole code of Example 1 gets fully vectorized.

To summarize, in Example 1 (Figure 3), the SLP algorithm generates two graphs, one rooted at the stores to A[i+0] and A[i+1], and a second one rooted at the stores to C[i+0] and C[i+1]. Even though there is data flowing across the graphs (through the data-flow edges as shown in Figure 3(b)), the SLP vectorizer will attempt to vectorize them in isolation, one at a time. Each time, SLP is treating the shared edges as an external data transfer with additional cost. Despite this, in this particular example the cost for each individual graph turns out to be profitable and the SLP algorithm successfully vectorizes the code as a whole.

B. SLP Limitation 1 of 2: Unreachable instructions

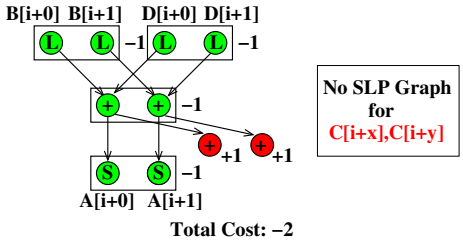
In Example 1 (Figure 3), SLP was lucky enough that instructions like the loads from E[i+0] and E[i+1] (see Figure 3(b)) were reachable by the second seed group (C[i+0],C[i+1]). This may not always be the case. Consider the scenario of Example 2, where the stores are not consecutive (say C[i+x] and C[i+y] of Figures 4(a) and 4(b)). In this case the non-consecutive stores to C[ ] do not qualify

[1]These extracts are redundant and will be optimized away once the code gets vectorized.

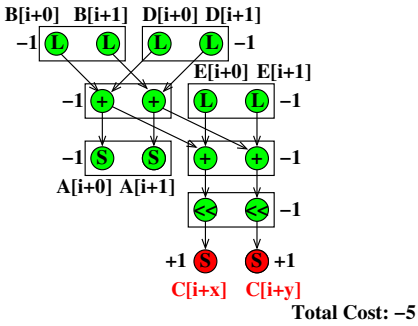


(a) Code. Notice the non-consecutive stores to C[ ]. They cannot form a seed group.

(b) Data Flow Graph



(c) SLP graph for seeds A[i+0],A[i+1]

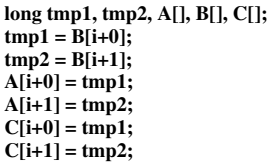


(d) SG-SLP graph for seeds A[i+0],A[i+1]. It includes all the isomorphic instruction groups reachable from the seeds.

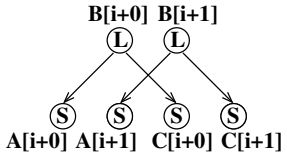
Figure 4. Example 2: SLP only vectorizes the graph rooted at A[i+0],A[i+1] and completely misses the rest of the code. On the other hand, SG-SLP fully vectorizes the code.

as seeds and therefore they are not grouped together. As a result, SLP will only build a single graph, that of Figure 4(c), rooted at A[i+0],A[i+1]. The code beyond this graph is unreachable to SLP and is not considered for vectorization (the unreachable nodes includes both loads from E[ ], the additions and the shifts). SLP will simply vectorize the nodes of Figure 4(c) and the rest of the code will remain scalar. This leads to a vectorization cost of -2.

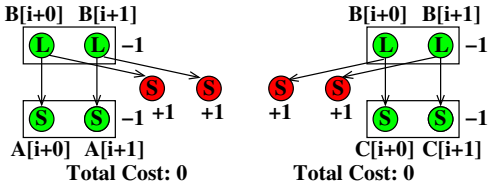
SG-SLP, on the other hand, builds a larger region. The SG-SLP supergraph grows across all paths towards both definitions and uses. The region extends to any isomorphic candidate instruction groups that could be potentially vectorized along those paths. The SG-SLP region for the seeds



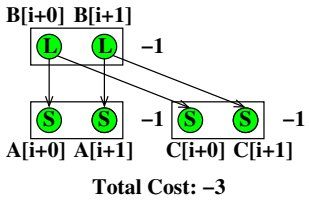
(a) Code



(b) Data Flow Graph



(c) SLP graphs. The left one for seeds A[i+0],A[i+1] and the right one for seeds C[i+0],C[i+1]



(d) SG-SLP graph for seeds A[i+0],A[i+1]. It includes all nodes reachable from the seeds all the way to C[i+0],C[i+1]

Figure 5. Example 3: SLP fails to vectorize the two graphs due to region fragmentation. SG-SLP succeeds.

A[i+0],A[i+1] is shown in Figure 4(d). It includes all the nodes that are present in the vanilla SLP graph, but it extends all the way to the non-vectorizable nodes C[i+x] and C[i+y]. In this graph all the groups in the rectangles are vectorizable, leading to a total cost of -5, a much better cost compared to original SLP cost of -2.

C. SLP Limitation 2 of 2: Region Fragmentation

As described in Section III-A, SLP may succeed in vectorizing standalone graphs, even though they may share edges with subsequent graphs. The problem arises when neighboring SLP graphs are connected to each other via multiple data-flow edges, or to be more precise, when the ratio of these edges to the total nodes is high enough. Let's consider another example which demonstrates the limitation of attempting to vectorize the graphs in isolation under such conditions.

The code of Example 3 (Figure 5(a)) corresponds to the data flow graph of Figure 5(b). Just like in Example 1, there are two groups of seeds: stores to A[i+0],A[i+1] and stores to C[i+0],C[i+1]. As explained in Section III-A, the existing SLP algorithm will generate separate graphs for each seed group and will attempt to vectorize each one of them individually. Once again, this will introduce additional extract/insert instructions at each graph's boundaries. For

example, in Figure 5(c), the left graph has two external uses (the stores to C[i+0],C[i+1]) which requires two individual extract instructions. The total cost of the graph is now equal to 0, which means that the graph on the left will remain scalar. Similarly, SLP will also fail to vectorize the graph on the right hand side, as its total cost is also 0.

The reason why SLP fails to vectorize the otherwise perfectly vectorizable code, is that SLP forms multiple graphs, one for each individual seed group. This leads to an overestimation of the extraction/insertion costs and therefore an inaccurate calculation of the total cost. Once the communication edges between individual SLP graphs becomes large enough compared to the number of nodes in the graph, the accuracy of the cost calculation drops to such an extent that SLP fails to vectorize each individual graph even though all of the graphs could have been fully vectorized.

SG-SLP overcomes this limitation. It builds a larger region that expands to all directions, including the nodes of the neighboring standalone SLP graphs. Just like SLP, it is rooted in the same seeds, and it starts by growing bottom-up. Unlike SLP though, SG-SLP will attempt to grow the region towards both the definitions and uses as long as they can form a vectorizable group. Therefore once SG-SLP reaches the top of the graph (e.g., the loads B[i+0],B[i+1] in Figure 5(b)), it will scan the uses and attempt to grow the region towards them (that is the stores to C[i+0],C[i+1]). SG-SLP generates a single graph originating from the seeds A[i+0],A[i+1] that spans all the nodes in the data-flow graph all the way to the succeeding seeds C[i+0],C[i+1] (Figure 5(d)).

The cost evaluation of this SG-SLP super-graph is done in a similar logic as SLP. As shown in Figure 5(d), all groups are vectorizable with a total cost of -3.

IV. SuperGraph-SLP

This section describes the details of the SG-SLP algorithm as implemented on an SSA-based IR. At a high-level view the new SG-SLP specific parts are shown in Figure 2 (the highlighted parts).

A. SG-SLP SuperGraph Construction

At the core of the SG-SLP algorithm lies the construction of the supergraph. This extends the vanilla SLP graph creation in step 3 of Figure 2. In short, the algorithm extends the graph creation to form larger regions that can include instructions that would otherwise be unreachable, and may even span multiple seeds. A summarized (and significantly simplified) version of the algorithm is listed in Algorithm 1.

The buildSG() function (line 6) triggers the creation of the graph for the given group of seed instructions by calling the recursive function buildSGrec() (line 11). Initially the inputs of buildSGrec() are the seeds (the root of the graph). If the candidate group instructions can be vectorized, the candidates are appended to the graph (lines 12 to 14). In vanilla SLP this function builds the graph recursively by performing a bottom-up depth-first traversal of the use-def chains, in a direction from the uses to the definitions (lines 22 to 23). The data-flow predecessors become the new candidates for vectorization and are passed as operands to the buildSGrec() function (line 23). In SG-SLP, the graph extends towards both the definitions and the uses. To avoid increased complexity, the algorithm will only consider a small number of users (in line 26). Then the algorithm goes through the users in the candidate groups formed with the subset of users and recursively calls buildSGrec using the users as the new candidates (lines 27 to 28). The generation of *defGroups* and *useGroups* of candidates (lines 21 and 26) is described in Section IV-B.

Of course, not all candidates end up being vectorized (line 12). The candidates for a vector group need to be:

1) Scalar (not already vectorized)
2) Isomorphic (same opcode)
3) Instructions (not constants)
4) Unique instructions (not multiple instances of the same instruction)
5) In the same control paths (all instructions in the same BB)
6) Able to be scheduled (not breaking dependences)
7) Not in the SLP graph already

If the group fails any of the conditions in the list of restrictions, then the group is rendered inappropriate for vectorization. In that case a special terminator group (tagged as non-vectorizable) is added to the graph and the recursion stops (lines 15 to 18). Otherwise the group is generated and appended to the graph (line 14). These restrictions not only prevent the algorithm from attempting to vectorize bad instructions, but it also prevents builsSGrec() from getting stuck in an infinite loop visiting the same nodes over and over again.

As a final step, SG-SLP needs to remove any vectorized seeds from all the list of non-visited seeds collected by the front-end of the algorithm (Figure 2 step 10). This is only a requirement for SG-SLP as unlike vanilla SLP it can vectorize in all directions, potentially vectorizing seed instructions that have not been considered for vectorization yet. These are removed to prevent the algorithm from attempting to vectorize them again.

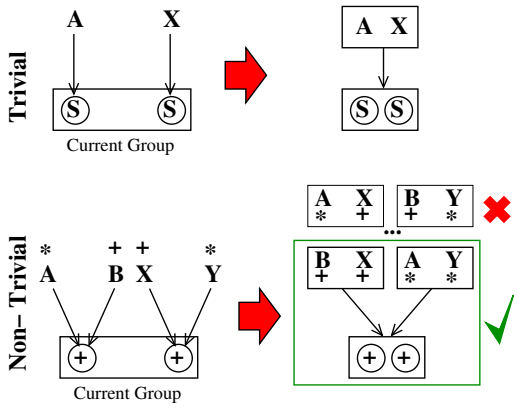
B. Pruning Candidate Groups

While growing the SLP graph in a bottom-up manner, we need to generate the best possible groups out of the predecessor instructions (Algorithm 1 line 21) such that vectorization can apply to that group too. Generating these vectorization groups in SLP can be challenging.

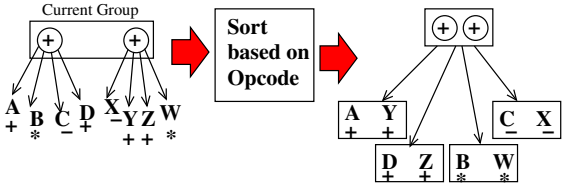
In the trivial case, the instructions in the current group have a single predecessor, therefore the candidate group contains only the immediate predecessors in the same order.

Algorithm 1. Building the SG-SLP supergraph.

```
1  /******* Building SG-SLP SuperGraph *******/
2  /* Input : Group of seed instructions      */
3  /* Output: SG-SLP supergraph               */
4
5  // The driver function for building the graph
6  buildSG(seeds) {
7   buildSGrec(seeds);
8  }
9
10 // Recursively build the graph
11 buildSGrec(candidateGroup) {
12  if (candidateGroup is legal to vectorize)
13    // Safe to vectorize, create vectorizable group
14    addGroupToGraph(candidateGroup, VEC)
15  else
16    // Mark the new group as non-vectorizable
17    addGroupToGraph(candidateGroup, NOVEC)
18    return
19
20  // SLP Recursion: defs
21  defGroups = get pruned definition candidate group
22  for each defGroup in defGroups
23    buildSGrec(defGroup)
24  // SG-SLP only recursion: uses
25  if (SG-SLP is enabled)
26    useGroups = get pruned user candidate groups
27    for each useGroup in useGroups
28      buildSGrec(useGroup)
29 }
```



(a) Pruning definition candidate groups in SLP (and SG-SLP).



(b) Pruning user candidate groups in SG-SLP.

Figure 6. Efficient generation of def/use candidate groups in SLP and SG-SLP.

This is shown in Figure 6(a) top, where each of the two stores has a single definition (A and X respectively). The only possible group of the definition instruction is [A, X] and this is the one being formed by SLP.

When the current group contains commutative instructions with two operands, then we have 2^{VL-1} possible definition groups to choose from (VL is the Vector Length). In practice though, LLVM's SLP will only form one of all these groups: The definition instructions per lane are sorted based on opcode and the groups are created using this ordering. This is shown in Figure 6(a) bottom: Instructions A(*), B(+) of lane 1 and X(+), Y(*) of lane 2 are sorted based on their opcode, leading to B(+), A(*) and X(+), Y(*). Then the groups generated are [B(+), X(+)] for lane 1 and [A(*), Y(*)] for lane 2.

SG-SLP has the additional task of generating groups out of the successors. Since the number of successors can be an arbitrary number N, the number of candidate groups is N^{VL-1}. SG-SLP performs a similar pruning for the successors as the one performed for the predecessors. It sorts the users based on their opcode and forms the candidate groups in a single attempt (Algorithm 1 line 26). This is shown in Figure 6(b). The instructions using the data produced by the two additions of the current group are of various opcodes (A is add, B is multiply etc.). The instructions are sorted based on their opcode, which gives us the following orderings : A(+), D(+), B(*), C(-) for lane 1 and Y(+), Z(+), W(*), X(-) for lane 2. Obviously, this technique will not always generate optimal groups, but exploring its effectiveness is beyond the scope of this paper.

C. SG-SLP Specific Details

Although the SG-SLP algorithm shares a lot of SLP's infrastructure, it has to address some unique challenges, for its correct operation. This section highlights the most important of these issues.

Vanilla SLP only considers gathering from scalars into vectors, since the SLP graph is built bottom-up and the leaf group nodes in the graph can only define values used by other nodes in the graph. In SG-SLP, however, the border group nodes of the graph can also be using values defined by other vectorized nodes within the graph. This requires step 4 of Figure 2 to be able to estimate the cost of such scatter instructions and step 9 to be able to generate scatter code.

Secondly, unlike vanilla SLP, SG-SLP needs to be able to swizzle the vector data to match the sequence expected by the store seed nodes whose order is specified by the memory addresses. This can happen in case the SG-SLP graph spans across multiple seeds. This is never an issue with the original SLP as stores are only present at the root of the SLP graph. This type of swizzling is to be considered in all steps 3, 4 and 9 of Figure 2.

Thirdly, SG-SLP needs to be able to revert to vanilla SLP if that proves more profitable (steps 4, 5 and 6 of Figure 2). Although this is not frequent, there is no guarantee that larger graphs will always improve the cost [29]. For example, it might happen that as the SG-SLP graph extends further than SLP, that particular section of

the code may introduce many additional overheads due to extensive insert/extract instructions, leading to worse overall performance. To avoid this situation, the SG-SLP algorithm marks the nodes that correspond to the limits of the SLP graph while it is building the supergraph. This allows it to calculate both SLP and SG-SLP costs without having to use a second graph specifically for SLP.

Finally, the step of converting the scalar instructions into vectors (step 8 of Figure 2) needs some SG-SLP specific adjustments. Just like in SLP, the nodes of SG-SLP are getting converted into vectors in a top-down fashion, with the definitions being vectorized before the uses. This allows for a simple conversion of the code in a single pass (post-order traversal). In LLVM this is done with a recursive function that first calls itself recursively for the predecessor groups and then converts the current group into vector form. However, SG-SLP needs to visit the uses as well. Therefore the recursive function needs to first calls itself recursively for all the predecessors groups, then to vectorize the current group and finally to call itself recursively again for all the successor groups.

D. Complexity Considerations

There are two common sources of complexity for the SLP and SG-SLP algorithm: 1. collecting the vectorization seeds and 2. attempting to vectorize the code that is rooted at the seeds. The first problem is the dominant source of complexity and therefore requires several simplifications to render the problem solvable in acceptable time (such as looking for specific instruction types to be used as seeds rather than any possible instruction type). The SLP algorithm solves the second problem quite fast: The SLP graph grows greedily bottom-up until it encounters instructions that do not meet the vectorization conditions. Even though in the worst case each SLP graph could contain all instructions of the program, in practice this is extremely rare and the SLP graph size does not grow large. To avoid this corner case, the LLVM implementation has a hard-coded size limit for the SLP graph size.

SG-SLP makes the second problem slightly more complex as it allows for the graph to grow in all directions. The worst case is equivalent to the worst case of vanilla SLP, that is a graph containing all the instructions of the program. In practice though, the same factors that limit the complexity of vanilla SLP apply to SG-SLP too. Encountering instructions that can be potentially vectorized is quite rare, thus practically limiting the complexity of the algorithm to similar levels to SLP.

Finally, SG-SLP introduces a third source of complexity: visiting a node's users. While the bottom-up exploration is bounded by the single producer (at least in SSA), SG-SLP will also explore the successors which could potentially be as many as the number of nodes in the program in the worst case. To limit the complexity, SG-SLP limits its exploration to a small number of successors (Algorithm 1 line 26). We empirically found that a limit of 4 users is usually adequate.

Section V-E discusses the complexity considerations further and provides data to support our claim that the complexity remains low.

Kernel	Benchmark	Filename:Line	Description
motivation1	Section III	Figure 3(a)	SLP succeeds
motivation2	Section III	Figure 4(a)	SLP succeeds partially
motivation3	Section III	Figure 5(a)	SLP fails
dc_coeff	464.h264	block.c:605	Pick out DC coefficients
vertical	464.h264	block.c:2054	DCT luma SP vert transform
horizontal	464.h264	block.c:914	Horizontal transform
inverse_dc	464.h264	block.c:687	Inverse DC transform
sub_four_vecs	433.milc	sub4vecs.c:26	Subtract su3 vectors
lattice_basis	444.namd	Lattice.h:275	Calc reciprocal lattice vectors
translation	453.povray	matrices.cpp:788	Compute translation transform

Table I
BRIEF DESCRIPTION OF KERNELS USED IN EVALUATION.

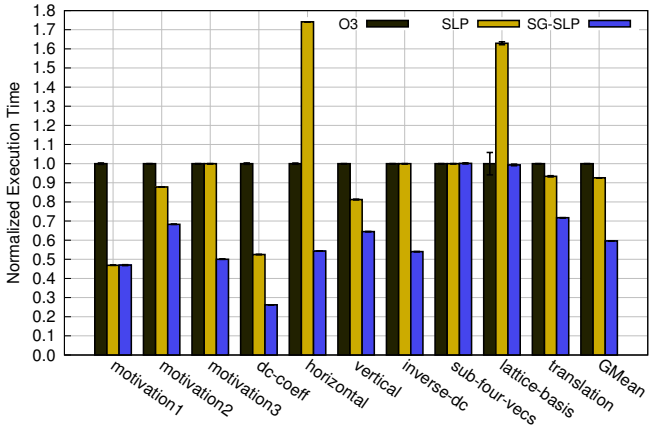


Figure 7. Execution time (normalized to O3).

V. RESULTS

A. Experimental Setup

We implemented SG-SLP in LLVM 3.8.1. The tests were compiled with one of the three configurations: The first one is O3, which corresponds to *-O3 -unroll-allow-partial* and the SLP vectorizer disabled, then there is SLP which is O3 but with the SLP vectorizer enabled, and finally SG-SLP which is O3 but with the SG-SLP algorithm enabled instead. All configurations were compiled with these additional options: *-march=skylake -mtune=skylake -mavx2*, and with the loop vectorizer disabled. The target platform is a desktop system running Linux-4.4.0 on an Intel Core i5-6600K 3.5GHz Skylake CPU, 16GB RAM. We evaluated our approach on kernels extracted from SPEC CPU2006 [36] as shown in (Table I). and included the motivating examples of Section III in these tests for completeness. For all performance results we executed the test 11 times, after skipping one initial run.

B. Performance

The first three tests are the motivating examples of Figures 3, 4 and 5. They show that the performance of these

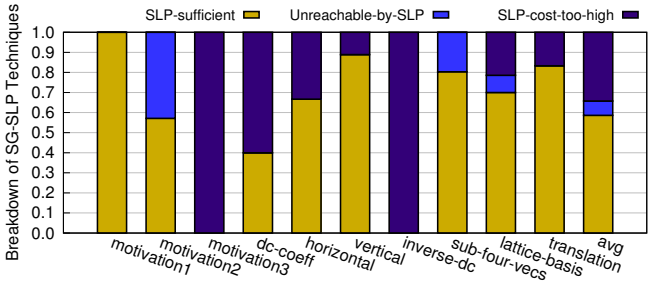


Figure 8. Breakdown of the individual techniques that enable SG-SLP.

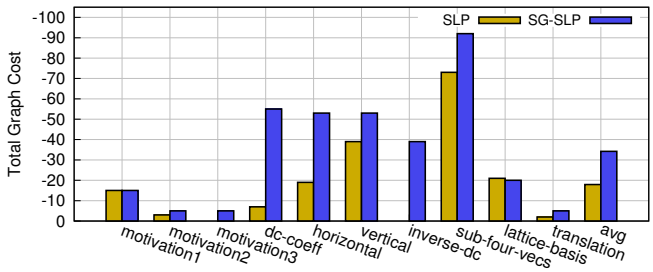


Figure 9. Static cost of vectorization graph (the more negative the better).

tests is what we expected: In *motivation1*, SLP is adequate (SG-SLP cannot do any better), in *motivation2* SLP succeeds but SG-SLP can do better, and finally in *motivation3* SLP fails to vectorize and SG-SLP succeeds. The rest of the results show that SG-SLP consistently improves upon SLP except for *sub-four-vecs* which shows no performance difference.

Figure 8 provides more insights into the individual factors that contribute to the success of SG-SLP. It shows the breakdown of the individual techniques responsible for the successful vectorization of instructions by SG-SLP: (i) for some of the instructions SLP is sufficient, (ii) some instructions are completely unreachable by SLP due to its restricted region formation (as described in Section III-B), and finally (iii) some of the instructions are reachable by SLP but are not vectorized due to the over-estimation of the vectorization cost, caused by the region fragmentation (see Section III-C).

Finally, in some cases vectorization proves harmful for performance. There are two such cases: 1) *horizontal* where SLP performs about 75% worse than O3, but performs significantly better with SG-SLP (45% faster than O3), and 2) *lattice-basis* where both SLP is about 65% slower than O3 and SG-SLP performs similarly to O3. Such slowdowns are not uncommon [2] . They are usually caused by one or more of the following: (i) poor cost model (that is wrong instruction costs), (ii) inability of the compiler to have a good knowledge of code generated, and therefore estimating the cost of instruction sequences that may resemble little the final assembly code. (iii) disabling succeeding optimizations due to converting the code into vector form (iv) succeeding compiler passes degrading the code quality. Nevertheless, SG-SLP does outperform SLP even in that case.

C. Static Cost

The static cost of the vectorization graph is the only metric that the vectorization algorithm possesses for estimating the code's performance. Subtracting the scalar cost from the vector cost gives us the cost savings (negative costs are associated with improved performance). The sum of all the savings is the total static cost of the vectorization graph. This cost tells us whether the vectorized code is more profitable than the original scalar, and therefore whether the code should get vectorized or not.

The static costs for the tests we evaluated are shown in Figure 9. SG-SLP improves the overall cost consistently. This is expected behavior since SG-SLP: (i) will attempt to vectorize at least as much as SLP, and (ii) it will only vectorize the code if it can improve the static cost over SLP (steps 5 and 6 of Figure 2).

There are, however, some exceptions. Upon a closer look we notice that in *lattice-basis* SLP has a slightly lower cost than SG-SLP. The reason is that the cost calculation is not restricted to original instructions only. It also includes instructions that were emitted by previous runs of the core SLP algorithm on sections of the code. This results in an artificial improvement of the cost of SLP. For example a first run of SLP may emit insert/extract instructions which can get vectorized by a succeeding run of SLP on neighboring code. In SG-SLP, however, these additional instructions would not have been emitted in the first place.

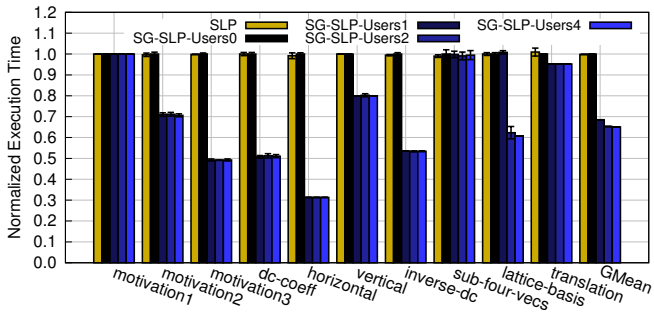


Figure 10. Execution time (normalized to SG-SLP-Users0) for 0, 1, 2 and 4 users. SLP is shown for reference. SG-SLP-Users4 is equivalent to SG-SLP.

D. Sensitivity to Number of Users

As we have discussed in Section IV-D, SG-SLP's region extends towards the users. We only consider a small number of them in order to reduce the complexity. We empirically found that a value of 4 provides good coverage. Figure 10 shows the execution time when we vary the number of users

[2] The performance results for these tests on an older Sandybridge system show a significantly lower slowdown (less than 20%).

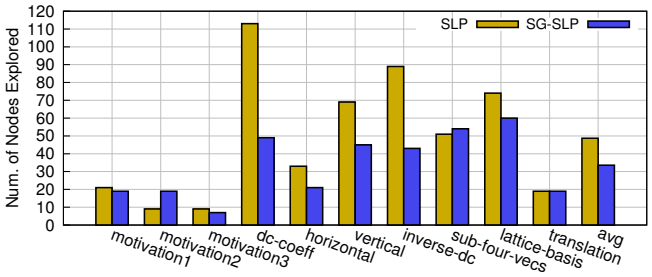


Figure 11. Total number of potentially vectorizable nodes explored.

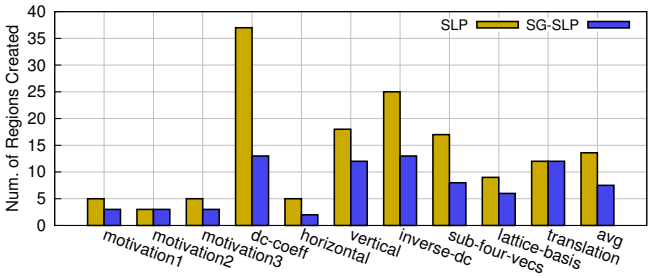


Figure 12. Number of graphs (regions) created.

from 0 to 4. SG-SLP-Users4 is referred to as SG-SLP by the rest of the text. As expected, exploring no users (SG-SLP-Users0) performs the same as SLP . Increasing the number of users to 1, reaches peak performance for all but lattice-basis which needs 2 users to perform best.

E. Complexity

SG-SLP operates on a larger region compared to the original SLP. As already discussed in Section IV-D, SG-SLP does not increase the number of seeds, which is the main source of complexity of the algorithm. It does, however, increase the size of the explored region. In practice, these regions do not grow to large sizes because of the numerous restrictions associated with vectorization (same number of instruction candidates, same instruction types, etc.).

We measured the total number of nodes considered for vectorization by each of the techniques and show the data in Figure 11. The figure shows that although SG-SLP can explore larger regions compared to SLP, it is not uncommon for it to attempt to vectorize fewer nodes than SLP. This can happen because while SG-SLP builds a single large supergraph, SLP builds multiple individual graphs. Several of these SLP graphs may share nodes, therefore adding to the times a node gets considered for vectorization. Moreover, SLP may repeatedly attempt and fail to vectorize these small graphs for various vector widths, adding to the number of times these nodes are considered for vectorization. SG-SLP, on the other hand, may succeed to vectorize a larger supergraph containing several of the smaller SLP graphs, saving from the extended exploration.

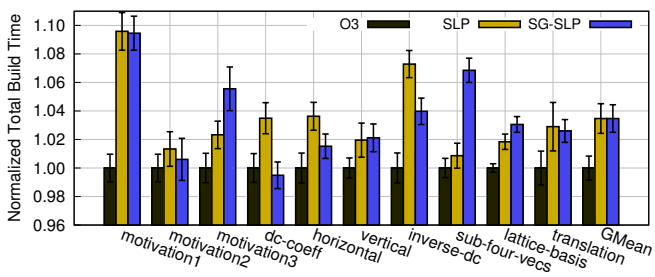


Figure 13. Total compilation time (normalized to O3).

Note that when SG-SLP succeeds in vectorizing a supergraph, it does not need to re-visit the vectorized seeds, nor build a graph rooted at those seeds. Figure 12 quantifies this data. It shows the number of times each of the algorithm grabs a group of seeds and proceeds to build a vectorization graph. On average, SG-SLP creates fewer (and larger) graphs compared to SLP. As expected, there is some correlation between the number of regions created and the exploration size of Figure 11: the more the created regions, the more extended the exploration performed by the algorithm. For example, SG-SLP generates about half the graphs for *inverse-dc* as compared to SLP and this is reflected in the number of nodes considered for vectorization (about half as well).

Finally, we we show the total compilation time in Figure 13 to show that the run-time of SG-SLP is comparable to SLP. We compiled each of the tests 11 times, skipping the first compilation. On average SG-SLP takes the same time to compile as SLP (within the noise margin). It is interesting to note that SG-SLP is often faster than vanilla SLP. This is attributed to the smaller exploration SG-SLP is required to do in those cases. This explains the strong correlation between the build time and the number of potentially vectorizable groups of instructions explored by each of the techniques (Figure 11). For example *dc-coeff* shows much greater exploration in SLP compared to SG-SLP, and at the same time it takes longer to compile.

In one case the vectorized code appears to compile slightly faster than O3 (e.g., *dc-coeff*). The reason is that the vectorizer, if successful, can reduce the total number of instructions in the program. Since the complexity of most compiler algorithms is a function of the size of the code, therefore the time taken by the optimizations that follow is reduced. The overall compilation time with the vectorizer enabled becomes similar (or lower) than with the vectorizer disabled.

VI. Related Work

A. Vector Processors

Vector machines have been the focus of high performance computing in both the industry and academia for several decades. Commercial wide vector machines machines like

[33], [27] or experimental ones like [15] have been used to accelerate scientific vector code.

Modern graphics processors (GPUs) implement similar type of wide vector execution, similar to the old vector machines [19]. Computation is performed in groups of 32 (on Nvidia), 64 (on AMD) or any of 8/16/32 (on Intel [14]) adjacent threads executing in lock-step. Such large vector widths are possible thanks to data-parallel graphics APIs (e.g., OpenGL [35], DirectX [11]) or languages like CUDA [25] or OpenCL [22], where the programmer explicitly exposes the available parallelism. Generating vector code for such inputs is straight forward, since vector parallelism is already exposed by the programming model to the compiler.

General purpose CPUs have been supporting short SIMD vector ISAs for several decades. Each vendor supports its own SIMD ISA, most of which are similar in functionality (examples are MMX/SSE4.x/AVX2/AVX-512 [13], 3DNow! [26], VMX/AltiVec [12], NEON [3]). These SIMD ISAs are being updated frequently with wider vectors and more powerful instructions being introduced every few processor generations.

B. Loop Vectorization

Loops are the main target of vectorization techniques [39]. The basic implementation strip-mines the loop by the vector factor and widens each scalar instruction in the body to work on multiple data elements. Early works of Allen and Kennedy on the Parallel Fortran Converter [1], [2], the works of Kuck et al. [16], Wolfe [38] and Davies et al. [8] solve many of the fundamental problems of automatic vectorization. Numerous improvements to the basic algorithm have been proposed in the literature and implemented in production compilers. Efficient run-time alignment has been proposed by Eichenberger et al. [9], while efficient static alignment techniques were proposed by Wu et al. [40]. Ren et al. [31] propose a technique that reduces the count of data permutations by optimizing them in groups. Nuzman et al. [23] describe a technique to overcome non-contiguous memory accesses and a method to vectorize outer loops without requiring loop rotation in advance [24]. A review of the effectiveness of loop vectorizing compilers has been studied by Maleki et al. [21]. Recently, architectural ISA extensions have been proposed by Baghsorkhi et al. [5], along with novel code generation techniques to allow vectorizing otherwise unvectorizable loops with cyclic dependences.

C. SLP Vectorization

SLP (Super-word Level Parallelism) has been introduced as a complementary auto-vectorization step for straight-line code. Larsen and Amarasinghe [17] were the first to present an auto-vectorization technique based on vectorizing independent isomorphic scalar instructions with no knowledge of any surrounding loop. Variants of this algorithm have been implemented in most major compilers including GCC [10] and LLVM, with a widely used algorithm being Bottom-Up SLP [32]. This is the state-of-the-art SLP algorithm and in this paper we use its LLVM implementation as a baseline for comparison for our SG-SLP work.

Shin et al. [34] propose an SLP algorithm with a control-flow extension that makes use of predicated execution to convert the control flow into data-flow, thus allowing it to become vectorized. A vectorizer in the instruction selection phase based on dynamic programming was introduced by Barik et al. [6], an approach different from most vectorizers. An automatic vectorization approach that works on straight-line code is presented by Park et al. [28]. It succeeds in reducing the overheads associated with vectorization such as data shuffling and inserting/extracting elements from the vectors. Liu et al. [20] present a vectorization framework that improves SLP by performing a more complete exploration of the instruction selection space while building the SLP tree. Porpodas et al. [30] apply SLP after first padding the scalar code with redundant instructions, to convert non-isomorphic instruction sequences into isomorphic ones, while [29] proposes limiting vectorization to a smaller section of the SLP graph in order to remove harmful parts. More recently, Zhou et al. [42] proposed a vectorization technique that aims at reducing the data re-organization overhead by considering both intra- and inter-loop parallelism. In another work [41], the same authors proposed a scheme that enables vectorization of SIMD widths that are not supported by the target hardware, by widening the vector instructions accordingly.

The techniques discussed so far are orthogonal to SG-SLP. None of them recognized or studied the missed vectorization opportunities caused by (i) unreachable instructions and (ii) excessive data sharing between consecutive SLP graphs. SG-SLP is the first to tackle both problems in a simple yet effective way by vectorizing on a larger region and without increasing the time complexity.

VII. Conclusion

In this paper we presented SuperGraph-SLP (SG-SLP), a novel SLP-based auto-vectorization algorithm with higher vectorization coverage and improved performance compared to the state-of-the-art bottom-up SLP. The state-of-the-art algorithm works on bottom-up regions and operates on them atomically, one by one. We showed that this leads to missed vectorization opportunities. SG-SLP addresses the limitations by introducing a vectorization region that extends both bottom-up and top-down when profitable. This SuperGraph region allows the SG-SLP vectorizer to: (i) extend the vectorization scope to code which would otherwise be unreachable, and (ii) successfully vectorize code that was previously considered non-profitable, due to extensive data sharing with neighboring graphs. SG-SLP improves code quality with little impact on compilation time.

REFERENCES

[1] J. R. Allen and K. Kennedy, "PFC: A program to convert fortran to parallel form," Department of Mathematical Sciences, Rice University, Tech. Rep., 1982.

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

[3] ARM Ltd, "ARM NEON," http://www.arm.com/products/processors/technologies/neon.php, 2014.

[4] 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.

[5] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu, "Flexvec: auto-vectorization for irregular loops," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[6] R. Barik, J. Zhao, and V. Sarkar, "Efficient selection of vector instructions using dynamic programming," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2010.

[7] J. L. Birch, "Using the chains of recurrences algebra for data dependence testing and induction variable substitution," Master's thesis, Department of Computer Science, The Florida State University, 2002.

[8] 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.

[9] 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.

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

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

[12] IBM PowerPC Microprocessor Family, "Vector/SIMD Multimedia Extension Technology Programming Environments Manual," 2005.

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

[14] Intel Corporation, "The Compute Architecture of Intel Processor Graphics Gen9," 2015.

[15] 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*, 1997.

[16] 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 (POPL)*, 1981.

[17] 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.

[18] 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.

[19] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, 2008.

[20] 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.

[21] 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.

[22] A. Munshi, "The OpenCL specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[23] 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.

[24] 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.

[25] Nvidia Corporation, "The CUDA specification."

[26] S. Oberman, G. Favor, and F. Weber, "AMD 3DNow! technology: Architecture and implementations," *IEEE Micro*, 1999.

[27] W. Oed, "Cray Y-MP C90: System features and early benchmark results," *Parallel Computing*, 1992.

[28] 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.

[29] V. Porpodas and T. M. Jones, "Throttling automatic vectorization: When less is more," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[30] 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.

[31] 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.

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

[33] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, 1978.

[34] 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.

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

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

[37] R. van Engelen, "Symbolic evaluation of chains of recurrences for loop optimization," Department of Computer Science, Florida State University, Tech. Rep. TR-000102, 2000.

[38] M. Wolfe, "Vector optimization vs. vectorization," in *Supercomputing*. Springer, 1988.

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

[40] P. Wu, A. Eichenberger, and A. Wang, "Efficient SIMD code generation for runtime alignment and length conversion," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.

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

[42] 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 (CGO)*, 2016.