Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements

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

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

Evgueni Brevnov
Intel Corporation, USA
evgueni.v.brevnov@intel.com

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

Timothy Mattson
Intel Corporation, USA
timothy.g.mattson@intel.com

***Abstract*—SLP Auto-vectorization converts straight-line code into vector code. It scans input code for groups of instructions that can be combined into vectors and replaces them with their corresponding vector instructions.**

This work introduces Super-Node SLP (SN-SLP), a new SLP-style algorithm, optimized for expressions that include a commutative operator (such as addition) and its corresponding inverse element (subtraction). SN-SLP uses the algebraic properties of commutative operators and their inverse elements to enable additional transformations that extend auto-vectorization to cases difficult for state-of-the-art auto-vectorizing compilers.

We implemented SN-SLP in LLVM. Our evaluation on a real system demonstrates considerable performance improvements of benchmark code with no significant change in compilation time.

***Index Terms*—SIMD, SLP, Auto-Vectorization**

I. Introduction

Modern high-performance processors include short SIMD vector units to support higher computational throughput. Making efficient use of the vector units is critical for achieving a large fraction of the available performance from a processor. The programmer can make use of the vector resources in several ways: (i.) using a vector-aware language or high-level programming model (e.g., OpenMP [1]), (ii.) using low-level target-dependent intrinsics or assembly, or (iii.) relying on the compiler's auto-vectorizer for converting unmodified scalar code into higher performing SIMD code. In practice, most general purpose applications rely on the auto-vectorizer to generate SIMD code, as it requires no effort from the programmer. For this reason, improving the coverage of auto-vectorizers is crucial for extracting the most out of modern processors. The extra vector units available in modern processors are wasted every time the compiler misses vectorization opportunities.

In order to leverage the processor's vector units, major compilers provide two main types of auto-vectorization: (1) the traditional loop vectorization (e.g., [2], [3]), and (2) a vectorizer that operates on straight-line code [4], [5], [6]. This work focuses on the second type, and more specifically on the Superword-Level Parallelism (SLP) vectorizer, as implemented in the GCC [7] and LLVM [8] compilers.

SLP does not operate directly on loop structures. Instead it explores straight-line code to find groups of isomorphic instruction sequences that can be converted into vectors. A typical implementation works by first scanning the code to find scalar instructions that can become the seeds of vectorization. If found, they are grouped and marked as candidates for vectorization. This group becomes the root of the SLP graph, which holds all candidate groups of scalar instructions. Then, SLP walks up the use-def chains, towards definitions, in an attempt to form more groups of isomorphic instructions. This process repeats until the SLP graph is fully formed. The next step evaluates whether converting the groups of the SLP graph into vector instructions will improve performance based on the compiler's cost model. This cost calculation factors in the overheads of inserting/extracting data into/out of the vector registers. If profitable, vector code gets generated, replacing the corresponding scalar code in each group.

Solving SLP optimally is computationally intensive as the underlying problem is a graph isomorphism problem. Therefore, all industrial-quality SLP implementations rely on heuristics to achieve the best outcome within reasonable compilation time. Hence, the goal of SLP research is to improve the algorithms that explore the code and collect vectorizable instructions, without increasing the compilation time.

Even with an optimal code exploration, SLP can fail to vectorize the code because the original scalar instructions form patterns that prevents vectorization. It is a known fact that performing some code massaging on the scalar code can sometimes help vectorize the code, like when reordering the operands of commutative operations (e.g., as shown in [6] and [9]). In this paper, we improve upon such techniques by introducing an SLP-based algorithm that performs on-the-fly code morphing to convert non-vectorizable code into a vectorizable form. We specifically target expressions trees composed of operations that are both commutative and associative (e.g. additions) and their corresponding inverse elements (e.g. subtractions), and we use a combination of code motion and operand reordering in a single coordinated step. The resulting vectorizer has better coverage than the state-of-the-art [9].

II. Background

There are two distinct types of auto-vectorization algorithms:

1) Loop-based vectorizations (e.g., [2], [3], [10], [11]) that operate on loops and perform widening of each instruction in the loop. This is equivalent to fusing together consecutive loop iterations into a single vectorized iteration. For this to work, the loop has to be well structured enough so that the compiler is able to analyze the data dependencies and prove that the transformation is legal.
2) Straight-line code algorithms, the most common being the fast bottom-up SLP [5] inspired by [4]. These algorithms can handle straight-line code anywhere in the program and, although not focused on loops, they can also vectorize code within loops where the loop-vectorizer fails.

SN-SLP improves upon the state-of-the-art bottom-up SLP, while the concepts introduced can also be applied to other types of vectorization algorithms too. This section is a short introduction to the bottom-up SLP algorithm, as implemented in GCC [7] and LLVM [8] .

A. SLP Overview

Vectorization algorithms based on SLP works by scanning the code for repeated sequences of isomorphic scalar instructions, aiming at replacing each one of them for their vector counterpart. Although SLP could be considered as a superset algorithm of broader scope than the loop-based vectorizer [5], in practice this is not the case. The algorithms are complementary and major compilers (e.g. GCC and LLVM) implement both algorithms. A common configuration is to run the SLP pass after the loop-vectorization pass.

The bottom-up SLP [5] algorithm, is an industrial-strength algorithm, sharing some fundamental concepts with the Super-word Level Parallelism paper [4]. This algorithm is present in both GCC and LLVM and is enabled by default for the higher optimization levels (-O2+ for LLVM).

As a first step, it scans the compiler's intermediate representation, identifying specific type of instructions, referred to as *seeds*. The seeds are scalar instructions sequences that have a high probability of giving us profitable vectorization if they get vectorized along with their dependent instructions. Commonly used seeds are *stores* or instructions that form reduction trees. The seeds become the first potential vector group and are the starting point of the algorithm's exploration. Then, the algorithm follows the use-def chains towards the definitions, to continue forming as many vectorizable groups as possible.

B. SLP Algorithm

A summarized overview of the SLP algorithm is shown in Figure 1 (the highlighted parts have been added or modified by SN-SLP). It starts by scanning the code for vectorizable seed

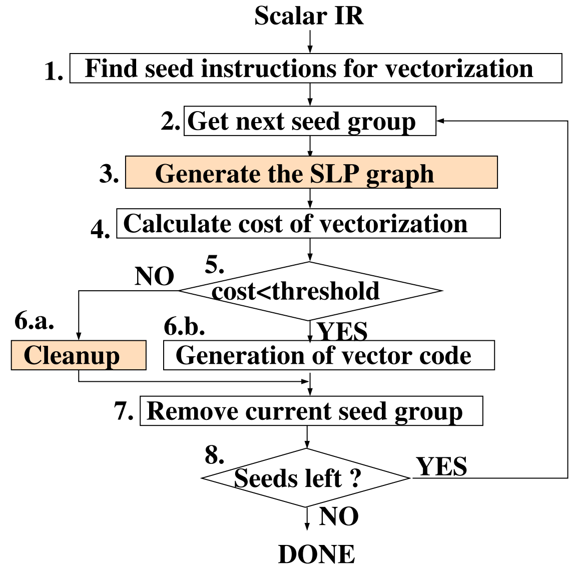


Fig. 1. Overview of the SLP algorithm. The highlighted sections are updated by the SN-SLP algorithm.

instructions (step 1), such as stores accessing adjacent memory locations, or the instructions feeding into a reduction tree (e.g., a reduction tree of additions). Adjacent memory instructions are some of the most promising seeds and therefore most compilers look for these first [5].

Next, the algorithm grabs a group of seeds from the seed work-list (step 2) and creates the first node in the SLP graph (step 3). Each node of the SLP graph is a group of vectorizable scalar instructions. For building the rest of the graph, the algorithm follows the use-def chains, towards the definitions and keeps repeating this process until it reaches non-isomorphic instructions, or until the instructions are not legal to vectorize. Each node contains not only the scalar instructions that are candidates for vectorization, but also 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 holds the cost of collecting the data from scalars and inserting them into a vector. At this point the algorithm stops exploring the code in this direction as this path cannot be vectorized any further.

Once the SLP graph has been constructed, SLP estimates the code's performance (step 4), with the help of the compiler's target-specific cost model. The cost of the graph is equal the sum of the savings from converting each group of scalar instructions into vector form (the lower the cost the better). Next, the cost is compared against a threshold (usually 0) to determine whether vectorization should proceed (step 5). If so, the compiler schedules the code and updates the intermediate representation (IR), replacing the groups of scalar instructions with their equivalent vector instructions, and emitting any *insert* or *extract* instructions required for communicating any required data between vectors and scalars outside the graph (step 6.b.). If the cost is not profitable, then the code remains unmodified. Since we are done using this seed group, we remove it from the work-list (step 7), and the whole process repeats for all the seed instructions in the work-list (step 8).

III. Motivation

This section explains SN-SLP algorithm through the use of examples. We compare it directly against the state-of-the-art, showing how SN-SLP can successfully vectorize code that state-of-the-art vectorizers cannot.

A. Algebraic Background

This work introduces the concept of the Super-Node, which is an improved and generalized version of the Multi-Node introduced by [9]. The Super-Node extends the concept of Multi-Node by allowing inverse operations to be included into a non-interrupted chain of operations with an operator that is both commutative and associative. Recall that an inverse element in abstract algebra generalizes the concept of sign reversal relative to addition or the reciprocal operation relative to multiplication. For example, Super-Nodes can be formed of expressions such as `A + B - C` or `A * B / C`. Algebraically, these expressions involve an operator that is both commutative and associative, i.e., addition or multiplication, and terms that include the corresponding inverse elements. Specifically, the expression `A + B - C` can be rewritten as `A + B + (-C)`, where `(-C)` is the inverse element of `C` under addition. This fact enables us to reorder all terms of this expression, namely, `A`, `B`, or `(-C)`. Note that the operand in the right-hand-side of the subtraction needs to be reordered with the unary negation operator, e.g., `A + B + (-C)` can be reordered as `A + (-C) + B`. A similar argument can be given to the example `A * B / C`, which can be written as `A * B * (1 / C)`. We can then leverage the algebraic properties of these operations in order to improve vectorization.

SN-SLP uses these properties to modify both the instructions internal to the Super-Node (referred to as trunk nodes), and its immediate predecessors (referred to as leaf nodes) to improve vectorization coverage. In addition to being able to reorder these *leaf* nodes across the whole Super-Node SLP, we may also change the order of the internal (*trunk*) nodes, if needed, to improve isomorphism. We explain how each of these actions is performed in the examples that follow.

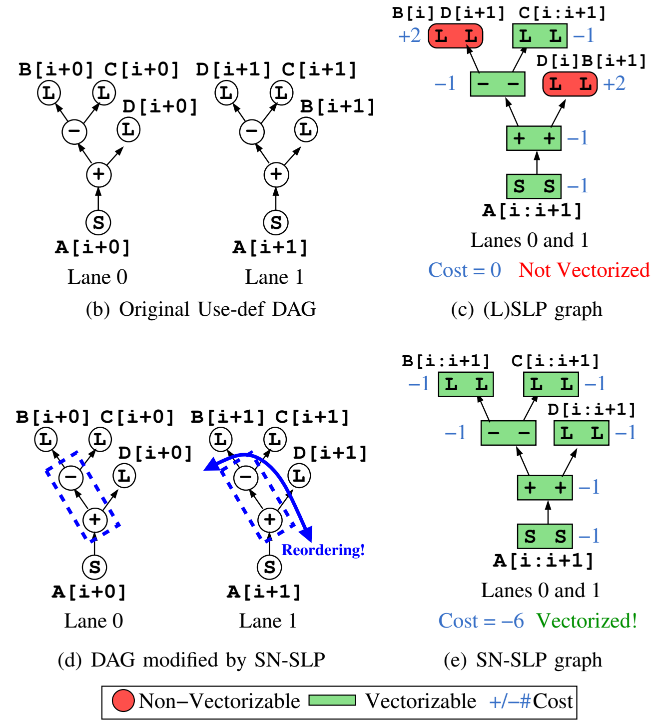
B. Reordering the Leaf Nodes

In this example we consider the C-style source code of Figure 2(a) which corresponds to the use-def DAG of Figure 2(b).

The state-of-the-art SLP algorithm will build the SLP graph as shown in Figure 2(c). Each green rectangular node corresponds to a group of scalars that can be potentially vectorized, while each red oval node represents those that cannot. The right-hand-side operand of the addition node `[+ +]` is a non-vectorizable group of non-adjacent *loads*: `D[i]` and `B[i+1]`. Similarly, the left operand of the subtraction node `[- -]` is non-vectorizable since `B[i]` and `D[i+1]` are not adjacent in memory. Each of these non-vectorizable nodes incur a cost

```
long A[],B[],C[],D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=D[i+1]-C[i+1]+B[i+1];
```

(a) Source Code



(b) Original Use-def DAG (c) (L)SLP graph

(d) DAG modified by SN-SLP (e) SN-SLP graph

Fig. 2. Reordering the *leaf* nodes of a Super-Node.

penalty of +2, negating the gains from the vectorizable nodes. The total cost is 0, which renders the whole SLP graph non-profitable to vectorize.

Super-Node SLP is able to massage the code on-the-fly to convert it fully vectorizable. It first forms the Super-Node out of the addition and subtraction operations of both lanes (the dashed rectangular around the + and − nodes). It then performs operand reordering across the whole Super-Node, following some legality rules to maintain the original semantics. The *leaf loads* from `B[i+1]` and `D[i+1]` are swapped, which results in all groups becoming isomorphic and therefore vectorizable. The final cost reflects this, since the total cost is now a profitable −6.

This leaf reordering across both additions and subtractions is not supported by the state-of-the-art Look-Ahead SLP (LSLP) algorithm [9]. LSLP can only operate on an uninterrupted chain of a single opcode (both commutative and associative, e.g, addition), since it is unable to check for reordering legality across the inverse elements (e.g., subtractions). For both motivating examples, vanilla SLP and LSLP perform identically.

C. Reordering the Trunk Nodes

In the example of Section III-B, the *trunk* operations in the Super-Node were isomorphic across lanes even in the original code (see Figures 2(b) and 2(d)). This is not always

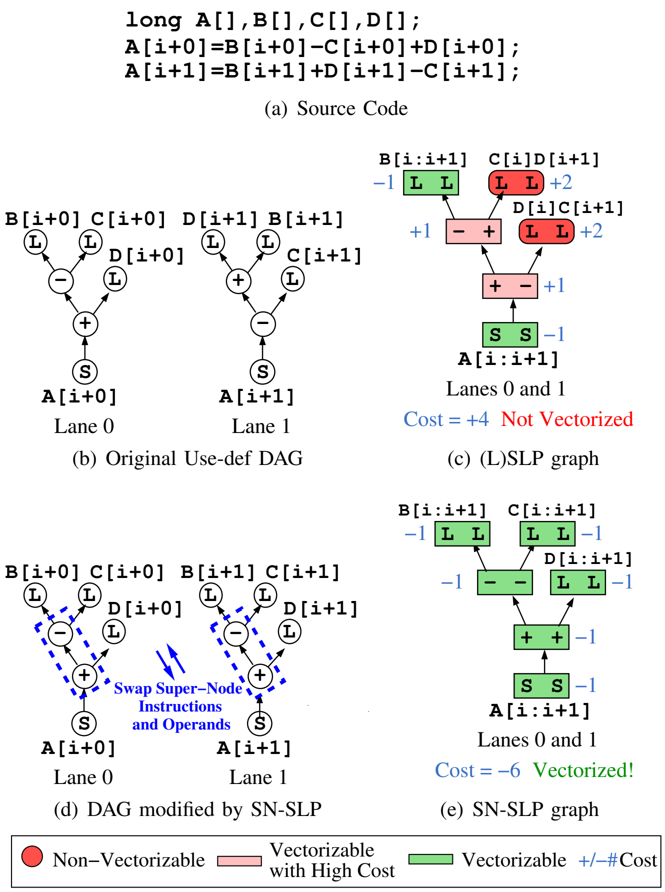


Fig. 3. Swapping both *trunk* nodes and leaves of a Super-Node.

the case. The DAG of Figure 3(b) would generate the SLP graph of Figure 3(c) under the state-of-the-art SLP algorithm. The `[+ -]` and `[- +]` nodes are vectorizable but with a higher cost of $+1$, since they are add/sub alternate sequences[1]. The state-of-the-art SLP can successfully form a vectorizable group of *loads* from `B[i:i+1]` by reordering the operands of the top-most addition node of Lane 1 in Figure 3(b) [2]. The other two *load* groups remain non-vectorizable. The total cost of SLP is $+4$ which is not profitable for vectorization.

Again, Super-Node SLP is able to fully vectorize this code. Initially it forms the Super-Node that includes the additions and subtractions of each lane. Then it tries to reorder the *leaf* nodes in an attempt to improve isomorphism. While doing so, it checks the legality of the transformation, and realizes that a simple *leaf* reordering will break the semantics of the computation. For example, the optimal Lane 1 order (from left to right) of `B[i+1]`, `C[i+1]` followed by `D[i+1]` would change the program semantics, as it would correspond to this code: `A[i+1] = B[i+1] + C[i+1] - D[i+1]`, which is different than the original code for this lane. Nevertheless, Super-Node SLP is able to legally form the groups of vectorizable *loads* by also reordering the *trunk* nodes themselves. The result of this reordering is shown in Figure 3(d). The final cost of Super-Node SLP is -6 which is profitable for vectorization.

In this particular example, the trunk nodes also become fully isomorphic, as they match perfectly after *trunk* reordering took place. This, however, is incidental and the algorithm does not rely on it. Please note, that even without this happening, the code would still be vectorizable with a small overhead due to the add/sub alternate *trunk* nodes, just like the *trunk* nodes of Figure 3(c). The assumption is that the *leaf* nodes matter more than the *trunk* nodes, so their ordering has a higher precedence.

IV. Super-Node SLP

A. Overview

SN-SLP introduces several changes at the core of the SLP algorithm. It changes the part of the algorithm that forms the SLP graph (that is the highlighted step 3 "Generate the SLP graph" of Figure 1). As already discussed in the examples of Section III, the graph formation is critical for the effectiveness of the vectorizer as it is the step where the code isomorphism is explored. SN-SLP improves the vectorizer's ability to massage the code and expose the underlying isomorphism better than before.

B. Construction of the Super-Node

The construction of the Super-Node is inspired by the construction of the Multi-Node in [9]. There are two distinct phases in our construction process. The first one grows the SLP graph like in vanilla SLP, by performing a group-wide recursive depth-first search into the use-def chains. This is shown in the `buildGraph` function (Listing 1, line 3). Super-Node SLP introduces the call to `buildSuperNode` of line 12, that attempts to build a Super-Node (if it finds good instruction candidates) and, if succesful, it performs the necessary code morphing. In the usual execution `buildSuperNode` returns early and `buildGraph` resumes building the vanilla SLP graph as usual (line 14 onwards. The algorithm recursively calls itself (line 19) growing the SLP-graph towards the definitions.

If a valid instruction group for the root of the Super-Node is identified by `buildSuperNode`, (by analyzing all the legality tests of line 41), the main search of `buildGraph` pauses and the algorithm switches to the second phase, the construction of the Super-Node (line 43 onwards). In this second phase, an independent bottom-up depth-first recursion is performed, until all the Super-Node-compatible instructions are collected for the *trunk* and *leaf* nodes of Super-Node.

The `buildSuperNode` function has two parts. The top part (lines 27 to 38) executes while the Super-Node creation is in progress and the bottom one (lines 41 to 54) handles both the initialization and finalization of the Super-Node. A new Super-Node is initialized with the set of `Instrs` (line 43), and a recursive call `buildGraph` is attempting to grow the Super-Node towards the definitions. Next, when the execution

[1]These are vectorized with the use of additional instructions, including `select` or `shufflevector` instructions, similarly to how it is described in PSLP [12]. Please also note that the x86 platforms implement the family of `addsub` vector instructions in the SSE instruction set. These can execute an alternate sequence of additions and subtractions across the vector lanes

[2]This reordering is a standard feature of LLVM's SLP (and LSLP [9]) and enables more effective vectorization of expressions with commutative operations

Listing 1. Super-Node graph construction

```
1 // In: Candidate scalar array for vectorization
2 // Out: SLP graph of grouped scalars
3 buildGraph(Instrs) {
4  // Legality check if Instrs can get vectorized
5  if (nonVectorizable(Instrs) {
6   // Create Non-Vec node and stop growing graph
7   Node = createNewGroup(instrs, NO_VEC)
8   graph.addNode(Node)
9   return
10  }
11  // Try to build a Super-Node
12  buildSuperNode(Instrs)
13  // Create Vectorized node and add to Graph
14  Node = createNewGroup(Instrs, VEC)
15  // Add the node to the SLP-graph
16  Graph.addNode(Node)
17  // Normal SLP recursion
18  for operands in instrs.getOperands() {
19   buildGraph(operands)
20  }
21 }
22
23 // In: Candidate scalar seeds for Super-Node
24 // Out: The new Super-Node and massaged code
25 buildSuperNode(Instrs) {
26  // If already building a Super-Node, grow it
27  if (!SN.empty()) {
28   // Legality checks for candidate Instrs
29   if (SN.areCompatible(Instrs)) {
30    // Append the operands to the Super-Node
31    SN.append(operands)
32    // Compute the APOs for each lane
33    Graph.computeAccumulatedPathOps()
34    // Continue the recursion toward the Defs
35    for (Ops in values.getOperands())
36     buildGraph(operands)
37    return;
38   }
39  }
40  // Build a new Super-Node
41  else if SN.areCompatible(Instrs) {
42   // Initialize the Super-Node
43   SN.init(Instrs)
44   // Try to grow Super-Node
45   for (operands in values.getOperands())
46    buildGraph(operands)
47   // The Super-Node has now been built.
48   // Find best order of trunks and leaves
49   SN.reorderLeavesAndTrunks()
50   // Apply changes to the underlying IR
51   SN.generateCode()
52   // Save (for undoing) and cleanup the state
53   SN.cleanup()
54  }
55 }
```

flow reaches the top part (line 27), further legality checks take place to ensure that the candidate instructions have compatible opcodes, that the instructions have no external uses to the Super-Node, and that they are not already part of the Super-Node (line 29). A legal group is added to the Super-Node in line 31, and the APO (Accumulated Path Operation) table is updated for this entry (line 33). This process continues recursively towards the definitions (line 36), until there are no more instructions to add to the Super-Node.

The *leaf* and *trunk* node reordering takes place in line 49, and then the IR is updated to reflect this new state of the code in line 51. This code transformation is a critical component of SN-SLP and is discussed in detail in Section IV-C. Since the IR may need to be reverted upon profitability failure, the state is saved and can be restored (line 53), allowing SLP to get back to its normal operation within `buildGraph` (line 16). This cleanup stage is also shown in Figure 1, step 6.a.

C. Reordering of Leaves and Trunks and Legality Checks

Unlike the Multi-Node of LSLP[9], operand reordering in a Super-Node is not always legal, and can also require additional transformations within the *trunk* nodes of the Super-Node itself. This section describes in detail how both of these tasks are performed by Super-Node SLP.

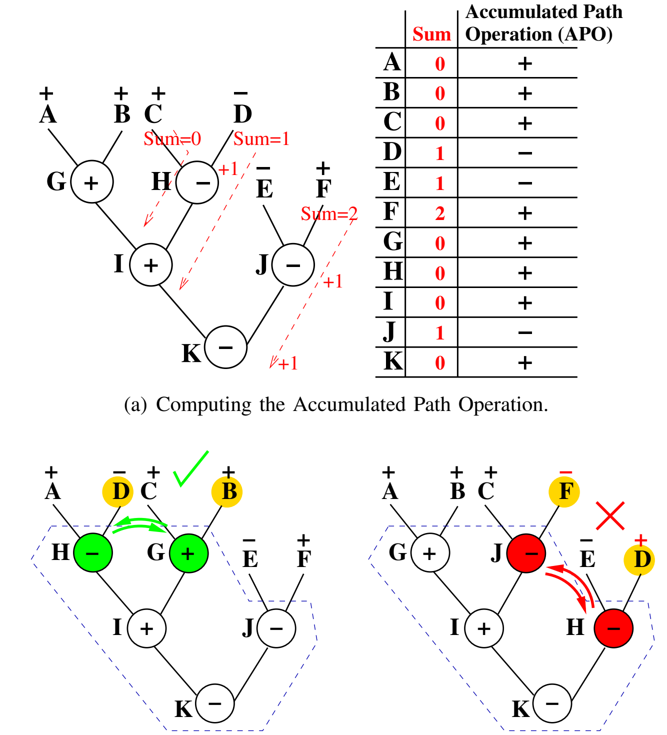
1) Accumulated Path Operation (APO): During the Super-Node construction, we annotate each node with the APO for each lane. This is the unary operation performed on each element that results in the same computation as the original expression. For example, the APOs in `A - (B + C)` are: '+' for `A`, '−' for `B`, and '−' for `C`, since `+A + (-B) + (-C)` is computationally equivalent to the original expression. Each of these unary operations can be calculated by walking down the expression tree and counting the number of right-hand-side edges of subtractions that we encounter. If this sum is even, then the APO is a '+', otherwise it is a '−'. The same property is obviously also used for multiplication and division.

Figure 4(a) shows a slightly more complicated example of an expression tree (left-hand-side) and the corresponding APO for each node (right-hand-side). The calculation path for nodes `C`, `D`, and `F` are shown with the red dashed lines.

2) Legality of Leaf Reordering: Since the Super-Node may contain inverse operators, *leaf* nodes can only move across the Super-Node in a restricted way. The legality rule in Super-Node SLP is that a *leaf* can only be placed at an operand position with the same *Accumulated Path Operation* (see Section IV-C1). For example, in Figure 3(d) Lane 0, `B[i+0]` can only move legally to the position of `D[i+0]` (both have a '+' APO), while `C[i+0]` cannot legally move to some other location, since it is the only operand with a '−' APO.

It turns out that this rule is quite restrictive. This legality rule would not allow the *leaf* order shown in Figure 3(d), Lane 1, since `C[i+1]` would not be allowed to be swapped with any other node. Similarly, we are not allowed to reorder nodes `B` and `D` of Figure 4(a). To relax this, we introduce *trunk* node reordering which, in effect, reorders the position of the APOs across the border of the Super-Node.

3) Improving Reordering with Trunk Movement: Given that reordering the *leaf* nodes by themselves is rather restrictive, under certain conditions, Super-Node SLP allows for bundles of *leaf* and *trunk* nodes to be reordered. It is legal to move a *trunk* node across the Super-Node as long as the APO of all nodes remains the same. For example, in Figure 3(d) Lane 1, both the Add and Sub locations have the same APO of '+', therefore it is legal to swap them. Swapping them causes the APOs of their operands to swap too, therefore allowing the `C[i+1]` *leaf* node to move up to match the position of `C[i+0]` in Lane 0. The rest of the *leaf* nodes on Lane 1, are



	Sum	Accumulated Path Operation (APO)
A	0	+
B	0	+
C	0	+
D	1	−
E	1	−
F	2	+
G	0	+
H	0	+
I	0	+
J	1	−
K	0	+

(a) Computing the Accumulated Path Operation.

(b) Legal *trunk* swap.

(c) Illegal *trunk* swap.

Fig. 4. How the Accumulated Path Operation (APO) restricts which code motions are legal.

now also free to match their Lane 0 counterparts, resulting in full isomorphism, as shown in Figure 3(d).

Figure 4(b) illustrates a valid swap between two *trunk* nodes, `G` and `H`, which allows swapping *leaf* nodes `B` and `D`. After this valid transformation, all APOs remain the same. Figure 4(c), on the other hand, shows an example of an invalid *trunk* swap of `H` and `J`, while trying to swap `D` and `F`. If the *trunk* reordering was allowed to take place (as shown in the figure), then the APOs of the *leaf* nodes would change (as shown), thus changing the semantics of the expression.

4) Putting it all together: The implementation of the Super-Node SLP *leaf* and *trunk* reordering is shown in Listing 2.

The method definition of Listing 2 line 3 implements the Super-Node's `reorderLeavesAndTrunks` method, which is called by `buildGraph` in Listing 1, line 49). It operates on the Super-Node `SN` object which was just created by `buildGraph`. Its job is to perform the necessary transformations on the Super-Node's *trunk* instructions and its operands *leaf* nodes, that will maximize the isomorphism and allow SLP to vectorize best.

It starts by iterating through the operand indexes. The Super-Node is considered as a single "fat" node with as many operands as its incoming edges. The indexes are sorted (line 5) such that we first visit the ones close to the root node of the Super-Node. The intuition behind this is that the closest to the root node, the more important it is to find good matching *leaf* nodes, as the further away from the root, the probability of finding isomorphism within the Super-Node's *trunk* nodes

Listing 2. Reordering *leaf* and *trunk* nodes

```
1  // In: A Super-Node
2  // Out: The order of leaf and trunk nodes
3  SuperNode::reorderLeavesAndTrunks() {
4   // Visit the operands closest to the root first
5   for (OpI in SN.getSortedOperandIdxs()) {
6    BestScore = 0
7    BestGroup = none
8    LeftOperands = SN.getSortedOperands()
9    // Try out all operands for the Left-most lane
10   for (LeftOp in LeftOperands) {
11    // Build the best group starting with LeftOp
12    Group = buildGroup(OpI, LeftOp)
13    // Get the Look-Ahead score of the Group
14    Score = getGroupLookAheadScore(Group)
15    // Keep the group with the best score
16    if (Score > BestScore)
17     BestGroup = Group
18   }
19   if (BestGroup == none) continue
20   // Apply BestGroup's ordering onto Super-Node
21   for (Lane in Lanes) {
22    // Swap the leaf and trunk nodes if needed
23    swapleafstrunkAndAPOs(BestGroup.getOperand(
        Lane), SN.getOperand(Lane, OpI))
24    // Mark the operand as 'used'
25    BestGroup.getOperand(Lane).setUsed()
26   }
27  }
28 }
```

decreases. The goal is to find the best possible group of *leaf* nodes, for all lanes, that should be moved to the `OpI`'th input of the Super-Node. In order to find the best group, it tries all possible *leaf* nodes at Lane 0, and using that as a starting point, asks for `buildGroup` (line 12) to return the best possible group for `OpI`. Next, it uses the look-ahead scoring routine as listed in LSLP [9], to calculate the sum of the pair-wise cost of each pair of instructions in `Group` (line 14). The score is then compared against the current best in order to keep the best group in `BestGroup`.

Once the best group is found, it is applied onto the Super-Node, meaning that the ordering of its *leaf* and *trunk* nodes is updated such that the node's `OpI`'th operands across all lanes are the nodes listed in `BestGroup`. This is performed within the loop of line 21. The old *leaf* operands are swapped with the ones in `BestGroup`, and the *trunk* nodes are swapped too if needed, along with the corresponding APO tables. Finally, the instructions in `BestGroup` are all marked as 'used', to avoid being used as parts of future searches. This process is greedy, since (*i.*) we need to cap compilation time for large Super-Nodes, and (*ii.*) the existing scoring system that guides the operand reordering has limited accuracy.

Now, let's focus on the implementation of the `buildGroup` function shown in Listing 3. As already mentioned, its goal is to get the instructions that lead to the best possible score, using `LeftOp` as the value in Lane 0 and given that this group of instructions will be the `OpI`'th operand of the Super-Node. The first check is to make sure that `LeftOp` can be legally moved to the `OpI`'th operand (line 7). If not, then there is an early exit. Next, the initial `Group` is formed (line 10), using `LeftOp` as its first

Listing 3. Optimizing the operands group score

```
1 // In: OpI: The Super-Node operand number,
2 //     LeftOp: The Lane 0 operand
3 // Out: The group with the best score
4 buildGroup(OpI, LeftOp) {
5  // Early exit if illegal to move LeftOp at OpI
6  OrigLeftOp = MN.getOperand(0, OpI)
7  if (!isLegalToMove(LeftOp, OrigLeftOp)
8   return none
9  // Initialize MaxGroup with the LeftOp
10  Group.append(LeftOp)
11  // Find the best RightOp, for all lanes
12  for (Lane : Lanes[1:]) {
13   // The original operand at OpI
14   OrigRightOp = MN.getOperand(Lane, OpI)
15   // Find the RightOp with the best score
16   BestScore = 0
17   BestRightOp = none
18   for (RightOp : MN->getOperands(Lane)) {
19    // Skip already used nodes
20    if (RightOp.isUsed()) continue
21    // If illegal to move the leaf by itself
22    if (! isLegalToMoveleaf(RightOp,
23                            OrigRightOp)) {
24     RTrunk = RightOp.gettrunk()
25     OrigRTrunk = OrigRightOp.gettrunk()
26     // If we cannot move its trunk node
27     if (RTrunk.getAPO() != OrigRTrunk.getAPO())
28      continue
29    }
30    // Get the score of RightOp candidate
31    Score = getLookAheadScore(LeftOp, RightOp)
32    if (Score > BestScore) {
33     BestScore = Score
34     BestRightOp = RightOp
35    }
36   }
37   if (BestRightOp == none)
38    return none
39   Group.append(BestRightOp)
40   LeftOp = BestRightOp
41  }
42  return Group
43 }
```

value. The rest of the code iterates through all remaining lanes (line 12), in an attempt to get the best sequence of instructions.

Building the best sequence of values is done by iterating through all possible operands for the current lane (`RightOp`) (line 18), skipping the used ones (line 20), and the one ones that are illegal to move (lines 22 and 27). The legality check is a two-step check, one that checks the movement of the *leaf* node alone (line 22), and if that fails, one that checks the movement of the *trunk* node that would allow the *leaf* to move (line 27). If all checks pass, then the score of matching `RightOp` with `LeftOp` from the previous lane is evaluated, using the look-ahead calculation of LSLP [9]. This score is compared against the `BestScore` found so far, and the `BestRightOp` is appended to the group (line 39). The current `BestRightOp` will become the `LeftOp` of the next iteration for the following lane (line 40). If no best *leaf* is found, an empty group is returned (line 37), otherwise the complete group is returned (line 42).

TABLE I
KERNELS USED IN THE EVALUATION.

Kernel	Benchmark	Filename:Line
433-mult-su3-mat-hwvec	433.milc	m_mat_hwvec.c:23
433-mult-su3-mat-vec	433.milc	m_matvec.c:64
433-mult-su3-nn	433.milc	m_mat_nn.c:90
453-minvers	453.povray	matcies.cpp:1331
454-solveSparseColumns	454.calculix	SubMtx_solveH.c:257
454-solveDenseSubColumns	454.calculix	SubMtx_solveH.c:9
motivation-leaf	Figure 2	
motivation-trunk	Figure 3	

V. Results

We implemented Super-Node SLP on the latest LLVM trunk, and we modified the existing implementation of the SLP Vectorizer (SLPVectorizer.cpp).

We evaluated the following configurations: (*i.*) O3, with all vectorizers disabled, (*ii.*) LSLP, which is vanilla SLP + the Multi-Node support from [9], and (*iii.*) SN-SLP which implements the full Super-Node. We compiled all benchmarks using `clang` or `clang++`, and we used the following compilation flags: `-O3 -ffast-math -march=native -mtune=native`. We have also enabled the horizontal reductions support (`-slp-vectorize-hor`) for both LSLP and SN-SLP. The target system is an Intel® Core™i5-6440HQ CPU @ 2.60GHz base, up to 3.50GHz.

For the evaluation we compiled and executed the unmodified C/C++ benchmarks from the SPEC CPU2006 [13] suite. Super-Node SLP was activated in a numerous functions. A small number of these functions were extracted as kernels (Table I) to help us focus the evaluation on code that triggers Super-Node SLP. We also included the motivating examples of Section III to the list of kernels for completeness. For all performance and compilation time results, we report the average of 10 executions, after skipping the first warm-up run. The error bars show the standard deviation.

The node structures of both LSLP and SN-SLP support integer and floating point additions and floating point multiplications. On top of these, SN-SLP also supports their inverse elements, that is integer and floating point subtractions and floating point divisions.

A. Performance on the Kernels

The normalized speedup over O3 for the kernels is shown in Figure 5. The first thing to note is that LSLP is on average the same as O3. In a few tests it performs worse than O3. This is expected because the cost model's static predictions about the performance of scalar code versus the equivalent vector code is not guaranteed to be correct. It is usually more accurate when comparing relative cost between vector variants, like LSLP versus SN-SLP. As expected, SN-SLP improves, with statistical significance, upon the LSLP, in most cases. This figure also includes the measurements of the code of the motivating examples of Section III. Since this code is just a simple loop, the speedup is very significant.

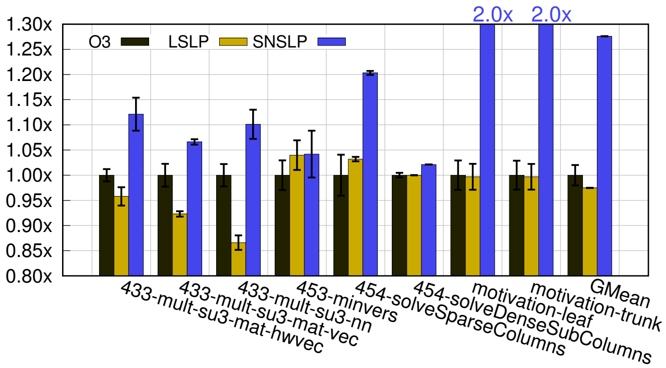


Fig. 5. Execution speedup, normalized to O3

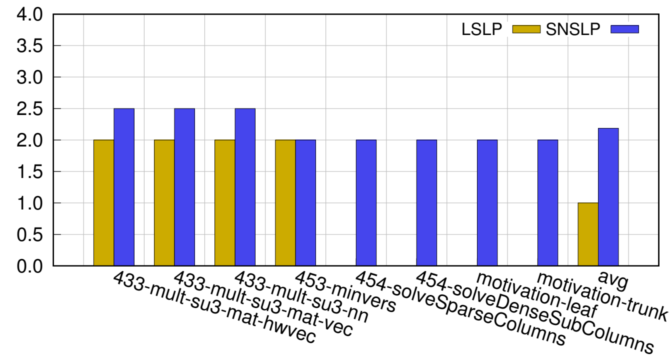


Fig. 7. Average Multi/Super-Node size.

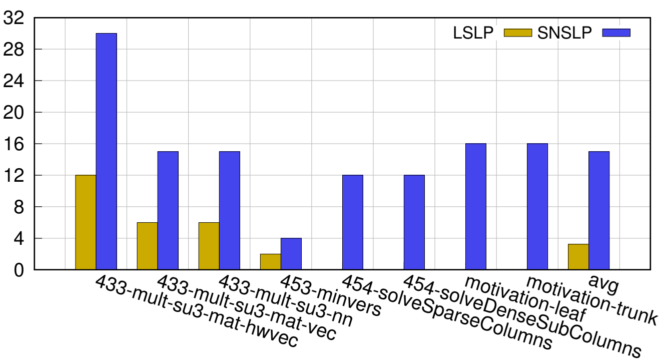


Fig. 6. Total Multi/Super-Node size.

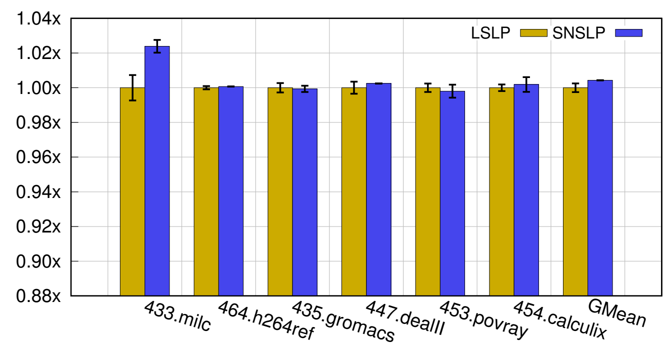


Fig. 8. Execution speedup normalized to LSLP.

This work introduces the concept of the Super-Node, which is an generalized Multi-Node [9]. In order to evaluate the effectiveness of each node type, we measured the total aggregate node size (depth) of each case (Figure 6), across all successfully vectorized code. Super-Node is clearly a more effective node structure, achieving much greater aggregate size compared to LSLP's Multi-Node. This can be attributed not only to the larger size of each individual node per SLP-graph (shown in Figure 7), but also to the fact that this more effective structure allows us to successfully vectorize more often. The average node size is about 2.2 instructions deep, which is intuitive because: (i) 2 is the minimum legal size for a Multi/Super-Node, and (ii) shorter chains are much more likely to be isomorphic than longer ones.

B. Performance on Benchmarks

We measured the performance across all C/C++ SPEC CPU2006 full benchmarks. Super-Node SLP activates only in six of them, the ones shown in Figure 5. Since Super-Node SLP is a generic optimization, not one that targets specific hot loops, the performance improvements across whole benchmarks were not expected to be significant. However, as it turns out, 433.milc achieves 2% speedup over LSLP, which is a very significant improvement. The rest of the benchmarks have no statistical difference between the two versions.

We measured the aggregate node size for the full benchmarks as well, and we present the results in Figure 9. As expected, Super-Node SLP creates more and nodes, but not always larger on average (Figure 10). Since Super-Node SLP gets activated more times than LSLP, it is not impossible that these frequent activations can pull the average down, towards the most common node sizes. Just like in the kernels, the average node size is close to 2.5 instructions.

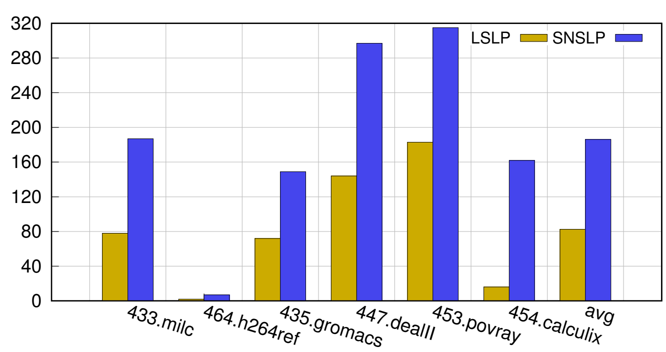


Fig. 9. Total Multi/Super-Node size.

C. Compilation Time

In compilers it is important to balance the generated code's performance and the time spent compiling. Since we are introducing a new design for the SLP algorithm, we also need to evaluate its execution time. We measured the wall compilation

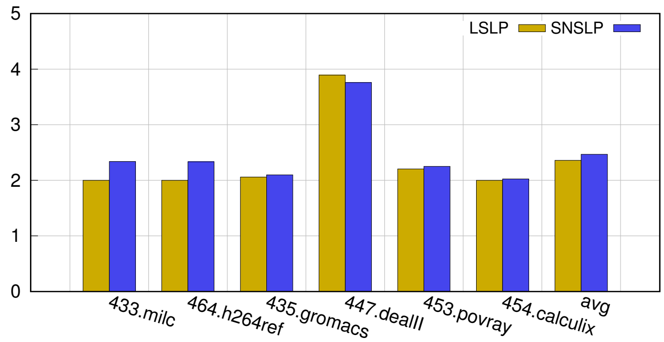


Fig. 10. Average Multi/Super-Node size.

time for all kernels (10 runs + 1 warm-up) and we show the normalized results in Figure 11. The figure shows that Super-Node SLP does not introduce any significant compilation-time overhead, which is expected, as we have not introduced any compilation-time intensive component. Moreover, when there is a significant code size reduction due to vectorized code (for example in the motivation kernels) we see a large reduction in wall time since there is less code to process for the remaining compiler passes.

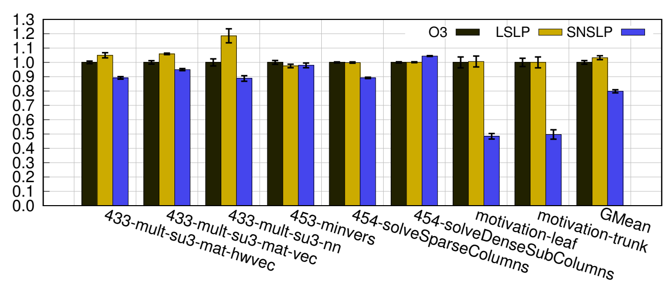


Fig. 11. Compilation time normalized to O3.

VI. Related Work

Traditionally, vector machines were the choice of preference for supercomputers [14], with scientific workloads being accelerated by both commercial [15], [16] and experimental [15], [16], [17] vector machines. Modern graphics processors (GPUs) implement similar types of wide vector execution, where computation is performed in groups of 32 (on Nvidia), 64 (on AMD) or any of 8/16/32 (on Intel [18]) adjacent threads executing in lock-step. Wide parallel computation on GPUs is possible thanks to data-parallel graphics APIs (e.g., OpenGL [19], DirectX [20]) or General Purpose GPU (GPGPU) languages such as CUDA [21] or OpenCL [22], where the programmer explicitly exposes the available parallelism.

General purpose CPUs have been supporting short SIMD vector ISAs for several decades (e.g, AVX* [23], 3DNow! [24], VMX/AltiVec [25], NEON [26]).

A. Loop Auto-Vectorization

Auto-vectorization techniques have traditionally focused on vectorizing loops [27]. These loop-based algorithms work by fusing consecutive loop iterations into a single vectorized iteration in a strip-mining fashion, widening each scalar instruction in the loop body to work on vector elements. Early works of Allen and Kennedy on the Parallel Fortran Converter [2], [3], the works of Kuck et al. [28], Wolfe [29], and Davies et al. [30] solve many of the fundamental problems of loop vectorizers. Since then, numerous other improvements have been proposed in the literature and implemented in production compilers [31], [32], [10], [11], [33]. Nuzman et al. [10] describe a technique to overcome non-contiguous memory accesses and a method to vectorize outer loops without requiring loop rotation in advance [11]. The overall effectiveness of loop vectorizing compilers has been studied by Maleki et al. [34]. Whole function vectorization has been the focus of Karrenberg et al. [35], [36], while recent improvements on control-flow linearization were presented by Moll et al. [37]. Finally, Masten et al. [38] proposes a loop vectorization-based technique for whole function vectorization.

B. SLP Auto-Vectorization

The SLP Vectorization was first proposed by Larsen and Amarasinghe [4]. It is a complementary technique to loop vectorization which focuses on vectorizing straight-line code instead of loops. Bottom-up Variants of this straight-line code vectorization algorithm have been implemented in compilers such as GCC [7] (Rosen et al. [5]) and LLVM [8] (Rotem et al. [6]). This bottom-up algorithm is widely adopted due to its low run-time overhead while still providing good vectorization coverage. In this paper, we use LLVM's bottom-up SLP algorithm as the baseline, after we extended it with support for Multi-Nodes, as described in [9].

Since its conception, several improvements have been proposed for the SLP vectorizer and straight-line-code vectorization in general. Shin et al. [39] propose a framework that makes use of predicated execution to convert the control flow into data-flow dependence, which enables a straight-line-code vectorizer to analyze and transform the predicated scalar code to vector code. Liu et al. [40] present a framework for a holistic SLP vectorizer that uses an iterative grouping mechanism to explore groups of vectorizable instructions and then forming vectors out of the most profitable ones. [41] improves upon this algorithm with an ILP solver to better explore the optimization space, which results in better performance, but at the cost of impractically long compilation times. Huh and Tuck [42] propose a different approach for identifying isomorphism in SLP vectorization based on growing the vectorizable graph from small predefined patterns. The Park et al. [43] 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. [12] propose PSLP, a technique that converts non isomorphic instruction sequences into isomorphic ones through instruction padding. In [44], the SLP region is pruned to scalarize groups of instructions that harm the vectorization cost, while in [45] a larger unified SLP region is used, that overcomes limitations associated with the inter-region communication and unreachable instructions. Zhou et al. [46] 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 [5], while in [47] vectorization is enabled for SIMD widths that are not supported by the target hardware. Variable-Width SLP [48] adjusts the SIMD width and performs the necessary shuffles to allow SLP to vectorize more code.

Look-Ahead SLP (LSLP) [9] focuses on improving vectorization of commutative operations. It introduces the concept of the *Multi-Node*, which is limited to commutative operations only. It also introduces the Look-Ahead operand reordering methodology, uses knowledge from instructions beyond the immediate predecessors, to improve the effectiveness of operand reordering. LSLP is our baseline comparison for our work. Super-Node SLP introduces the Super-Node, which Multi-Node which includes both commutative operations and their corresponding inverse elements. This allows the algorithm to perform more effective code massaging, leading to improved vectorization.

VII. Conclusion

We presented Super-Node SLP (SN-SLP), an SLP-based auto-vectorization algorithm capable of optimizing expressions of commutative operations and their inverse elements (for example additions and subtractions). SN-SLP performs aggressive code motion across such expressions to expose the underlying isomorphism. Our SN-SLP implementation in LLVM shows considerable performance improvements over the state-of-the-art on real benchmark code, without any significant change in compilation time.

Acknowledgment

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

References

[1] OpenMP Architecture Review Board, "OpenMP Application Program Inteface," https://www.openmp.org/specifications/.

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

[3] J. R. Allen and K. Kennedy, "Automatic translation of Fortran programs to vector form," *Tranactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 4, 1987.

[4] 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.

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

[6] N. Rotem and A. Schwaighofer, "Vectorization in LLVM https://llvm.org/devmtg/2013-11/slides/Rotem-Vectorization.pdf," *LLVM Developer's meeting, 2013*.

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

[8] 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.

[9] 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*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 163–174.

[10] 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.

[11] 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.

[12] 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.

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

[14] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: Past, present and future," in *Proceedings of the 12th International Conference on Supercomputing*, ser. ICS '98. New York, NY, USA: ACM, 1998, pp. 425–432. [Online]. Available: http://doi.acm.org/10.1145/277830.277935

[15] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, 1978.

[16] W. Oed, "Cray Y-MP C90: System features and early benchmark results," *Parallel Computing*, vol. 18, no. 8, 1992.

[17] 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*, vol. 30, no. 9, 1997.

[18] Intel Corporation, "The Compute Architecture of Intel Processor Graphics Gen9," https://software.intel.com, 2015.

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

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

[21] Nvidia, "Compute unified device architecture programming guide," 2007.

[22] Khronos OpenCL Working Group, "The OpenCL Specification," https://www.khronos.org/opencl/.

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

[24] S. Oberman, G. Favor, and F. Weber, "AMD 3DNow! technology: Architecture and implementations," *IEEE Micro*, vol. 19, no. 2, 1999.

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

[26] ARM Ltd, "ARM NEON," https://developer.arm.com/technologies/neon, 2014.

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] A. Anderson, A. Malik, and D. Gregg, "Automatic vectorization of interleaved data revisited," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 50:1–50:25, Dec. 2015.

[34] 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.

[35] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011, pp. 141–150.

[36] R. Karrenberg and S. Hack, "Improving performance of opencl on cpus," in *International Conference on Compiler Construction*. Springer, 2012, pp. 1–20.

[37] S. Moll and S. Hack, "Partial control-flow linearization," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 543–556.

[38] M. Masten, E. Tyurin, K. Mitropoulou, H. Saito, and E. Garcia, "Function/Kernel Vectorization via Loop Vectorizer," *Proceedings of the 5th Workshop on The LLVM Compiler Infrsatructure in HPC (LLV-HPC)*, 2018.

[39] 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.

[40] 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.

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

[42] 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*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 718–729.

[43] 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.

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

[45] V. Porpodas, "SuperGraph-SLP Auto-Vectorization," in *2017 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2017, pp. 330–342.

[46] 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*. ACM, 2016, pp. 59–69.

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

[48] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes, "VW-SLP: Auto-vectorization with Adaptive Vector Width," in *2018 International Conference on Parallel Architecture and Compilation (PACT)*, ser. PACT 2018.