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., 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 Superword 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 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

![Fig. 1. Overview of the SLP algorithm. The highlighted sections are updated by the SN-SLP algorithm.](image-url)
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: \( D[i+1] \) 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 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
long A[], B[], C[], D[];
A[i+0]=B[i+0]-C[i+0]+D[i+0];
A[i+1]=B[i+1]+D[i+1]-C[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. 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. 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). 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 vectoriz-
able 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 successful, 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
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

1These are vectorized with the use of additional instructions, including
select or shufflevector, similarly to how it is described in
PSL [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
2This 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 // In: Candidate scalar seeds for Super-Node
23 // Out: The new Super-Node and massaged code
24 buildSuperNode(Instrs) {
25     // If already building a Super-Node, grow it
26     if (!SN.empty()) {
27         // Legality checks for candidate Instrs
28         if (SN.areCompatible(Instrs) {
29             // Append the operands to the Super-Node
30             SN.append(operands)
31             // Compute the APOs for each lane
32             Graph.computeAccumulatedPathOps()
33             // Continue the recursion toward the Defs
34             for (Ops in values.getOperands())
35                 buildGraph(operands)
36             return;
37         }
38     }
39     // Build a new Super-Node
40     else if SN.areCompatible(Instrs) {
41         // Initialize the Super-Node
42         SN.init(Instrs)
43         // Try to grow Super-Node
44         for (operands in values.getOperands())
45             buildGraph(operands)
46         // The Super-Node has now been built.
47         // Find best order of trunks and leaves
48         SN.reorderLeavesAndTrunks()
49         // Apply changes to the underlying IR
50         SN.generateCode()
51         // Save (for undoing) and cleanup the state
52         SN.cleanup()
53         }
54     }
```

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
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 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...
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).

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 (-sloop-vectorize-hor) for both LLVM 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.

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 (-sloop-vectorize-hor) for both LLVM 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.
This work introduces the concept of the Super-Node, which is a 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.

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
time for all kernels (10 runs + 1 warm-up) and we show the normalized results in Figure 11. The figure shows that SuperNode 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. 10. Average Multi/Super-Node size.](image)

![Fig. 11. Compilation time normalized to O3.](image)

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


