Throttling Automatic Vectorization: When Less Is More

Vasileios Porpodas, Timothy M. Jones
Computer Laboratory, University of Cambridge
vp331@cl.cam.ac.uk, tmj32@cl.cam.ac.uk

***Abstract*—SIMD vectors are widely adopted in modern general purpose processors as they can boost performance and energy efficiency for certain applications. Compiler-based automatic vectorization is one approach for generating code that makes efficient use of the SIMD units, and has the benefit of avoiding hand development and platform-specific optimizations. The Superword-Level Parallelism (SLP) vectorization algorithm is the most well-known implementation of automatic vectorization when starting from straight-line scalar code, and is implemented in several major compilers.**

The existing SLP algorithm greedily packs scalar instructions into vectors starting from stores and traversing the data dependence graph upwards until it reaches loads or non-vectorizable instructions. Choosing whether to vectorize is a one-off decision for the whole graph that has been generated. This, however, is sub-optimal because the graph may contain code that is harmful to vectorization due to the need to move data from scalar registers into vectors. The decision does not consider the potential benefits of throttling the graph by removing this harmful code. In this work we propose a solution to overcome this limitation by introducing Throttled SLP (TSLP), a novel vectorization algorithm that finds the optimal graph to vectorize, forcing vectorization to stop earlier whenever this is beneficial. Our experiments show that TSLP improves performance across a number of kernels extracted from widely-used benchmark suites, decreasing execution time compared to SLP by 9% on average and up to 14% in the best case.

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

I. Introduction

In recent years vectorization for general purpose processors, in the form of Single Instruction Multiple Data (SIMD) instruction set extensions, has gained increasing popularity, especially for applications in the signal-processing and scientific-computing domains. These vector instructions provide energy-efficient and high-performance execution by exploiting fine-grained data parallelism, and vector ISAs are provided by all major processor vendors. Their effectiveness for certain application domains has led to regular improvements, with designers increasing both the width of the data paths (e.g., 512 bits in Intel's AVX-512 extensions) and the diversity of instructions supported.

However, to make best use of these SIMD units, software developers must extract significant amounts of data-level parallelism from their applications, either manually by hand or automatically within the compiler. An automatic vectorization pass available within mature compilers performs a code analysis to identify profitable instructions for conversion from scalar form to vector, and then transforms the code appropriately. For all but the most highly-tuned codes (e.g., certain library functions), compiler-based automatic vectorization is the preferred method.

Vector extraction is generally performed either on loops or across straight-line code. For loops with well-defined induction variables, usually affine, and inter- and intra-loop dependences that are all statically analyzable[1], loop-based algorithms [1], [2] can combine multiple iterations of the loop into a single iteration of vector instructions. However, these restrictions frequently prohibit loop vectorization in general-purpose workloads.

For straight-line code, there exist algorithms that operate on repeated sequences of scalar instructions, regardless of whether they are in a loop or not [3], [4]. These do not require sophisticated dependence analysis and have more general applicability, succeeding in cases where the loop vectorizers would fail. In situations where the loop vectorizer is unable to transform the loop (e.g., due to complicated control-flow), the straight-line code vectorizer may still be able to vectorize basic blocks from within the loop.

SLP [3] is the state-of-the-art straight-line code vectorizer and has been implemented in several compilers, including GCC [5] and LLVM [6]. It works by scanning the code for scalars that can be grouped together into vectors. After collecting all these groups, it evaluates their performance, checking whether converting all groups into vectors is better than leaving all them scalar. In making its calculations it factors in the costs of gathering data into the vector registers and scattering it back again afterwards. This one-off check for all groups is a fundamental limitation of the SLP algorithm because some groups, although vectorizable, may require an excessive number of gather and scatter instructions to be inserted, which end up harming performance.

To overcome this limitation, we propose Throttled SLP (TSLP), a novel SLP-based automatic vectorization algorithm that performs an exploration of the vectorizable groups, evaluating their performance in steps, and deciding whether vectorization should stop prematurely. We refer to this early termination of vectorization as "vectorization

[1]Strictly speaking some of these data dependence checks can be performed at run-time. The compiler can then generate multiple versions of the code (assuming various states of the dependences), one of which gets executed depending on the outcome of the run-time check.

throttling". Throttling helps reduce the performance penalties caused by sections of the code that, if vectorized, would cost more than their scalar form. TSLP results in more densely vectorized code with all the harmful parts completely removed. Although counter-intuitive, this improves vectorization coverage and leads to better performance.

The rest of this paper is structured as follows. Section II gives an overview of the SLP vectorization algorithm and motivates the need for throttling the instructions to vectorize. Section III then describes TSLP. In section IV we present our experimental setup before showing the results from running TSLP in section V. Finally, section VI describes related work before section VII concludes.

II. Background and Motivation

An automatic vectorization pass within the compiler identifies regions of code that would be better executed as vectors rather than scalars, according to a particular cost model. We first give an overview of SLP, which is the baseline the vectorization algorithm, then identify reasons for sub-optimal performance of the final vector code that TSLP can overcome.

A. Straight-Line Code Vectorization

The most well-known straight-line code vectorizer is the Superword-Level Parallelism algorithm (SLP [3]). It scans the compiler's intermediate representation, identifying sequences of scalar instructions that are repeated multiple times and fusing them together into vector instructions. The main difference of SLP from traditional vectorization techniques is that SLP does not operate on loops, making it much more generally applicable than loop-based vectorizers. In fact, in its usual form, the algorithm does not interact with the loop that bounds the target code. The code to be vectorized can span multiple basic blocks within the compiler's control flow graph, as long as each group of instructions to be vectorized belongs to the same basic block.

A high-level overview of the SLP algorithm is shown in figure 1 where non-highlighted parts belong to the original algorithm and the orange-highlighted parts belong to TSLP. We give an overview of SLP below and describe the TSLP extensions in section III.

The SLP algorithm first scans for vectorizable seed instructions (step 1), which are instructions of the same type and bit width that are either: *i.* non-dependent memory instructions that access adjacent memory locations (scalar evolution analysis [7], [8], [9] is commonly used to test for this); *ii.* instructions that form a reduction; or *iii.* simply instructions with no dependencies between them. Adjacent memory instructions are the most promising seeds and therefore most compilers look for these first [10].

The algorithm then forms groups of potentially vectorizable instructions by following the data dependence graph that starts at the seed instructions (step 2). A common

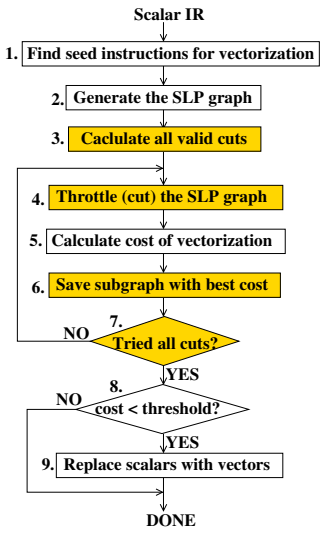


Figure 1. Overview of the TSLP algorithm. The highlighted boxes refer to the structures introduced by TSLP.

method for generating the graph is to start from store seed instructions and build the graph from the bottom-up, although it could also be created by starting at loads and building top-down. Both GCC's and LLVM's SLP vectorizers start at stores [10]. Each group primarily points to the scalar instructions that could be vectorized, but it also carries some auxiliary data such as the group's cost (see next step). As soon as the algorithm encounters scalar instructions that cannot form a vectorizable group it terminates its traversal.

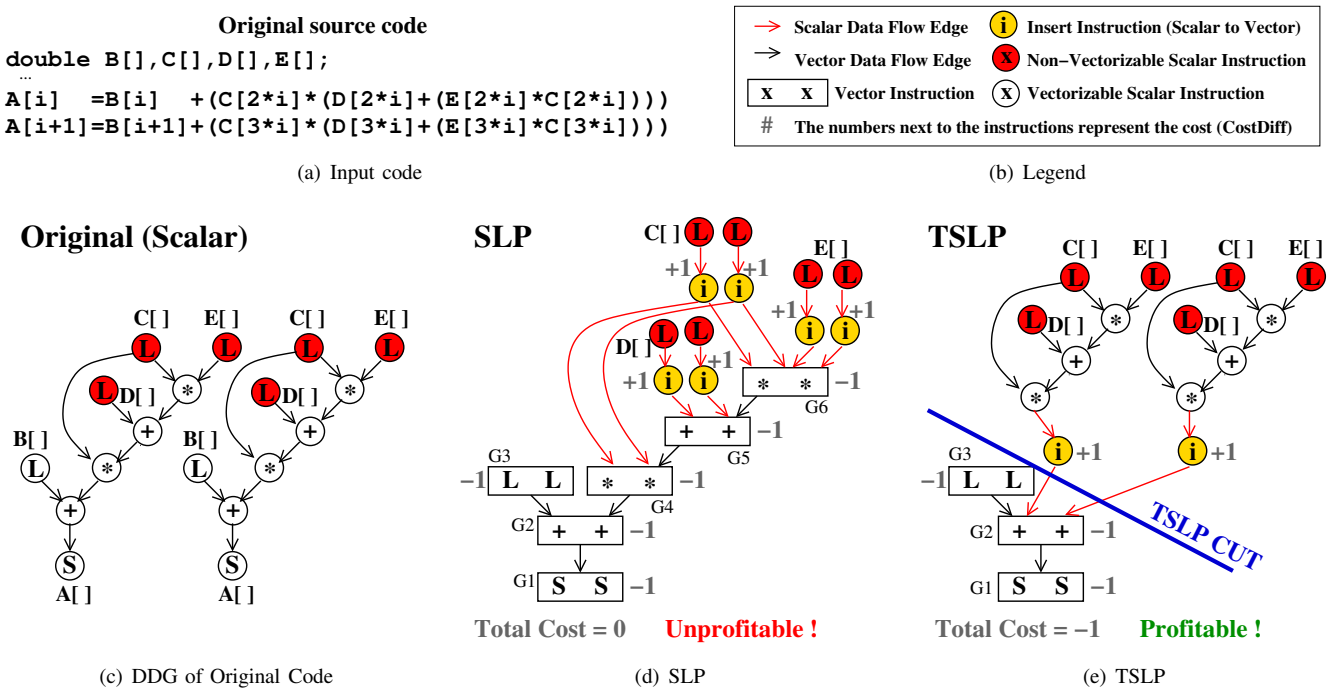
Once the graph has been constructed, SLP statically estimates the code's performance (step 5). This involves querying the compiler's target-specific cost model for the cost of each individual instruction in either scalar ($ScalarCost$) or vector form ($VectorCost$). Each group is tagged with the cost difference between vector and scalar form, i.e.,

$$CostDiff = VectorCost - ScalarCost$$

A negative result suggests that vectorization is beneficial for this specific group. For an accurate cost calculation the algorithm also takes into account any additional instructions required for data movement between scalar and vector units ($ScatterGatherCost$). The total cost ($TotalCost$) for the whole region under consideration is computed as the sum of all group costs along with the cost of additional instructions, i.e.,

$$TotalCost = \sum_{g=1}^{groups} CostDiff\,(g) + GatherScatterCost$$

In step 8, $TotalCost$ is compared to a threshold to determine whether vectorization should proceed. This threshold is normally set to 0, meaning that if $TotalCost < 0$ then vectorization is profitable, otherwise it is not. In other words, vectorization is only profitable if the vectorized form of the



(a) Input code (b) Legend

(c) DDG of Original Code (d) SLP (e) TSLP

Figure 2. TSLP enables vectorization by forcing the code beyond the cut to remain scalar.

code has a lower total cost than its scalar form. Finally, if vectorization should go ahead, the compiler modifies the intermediate representation code by replacing the groups of scalar instructions with their equivalent vector instructions (step 9). If not, then the code remains unmodified.

This process repeats for every potential seed and whenever vectorization succeeds the decision is greedily considered final. Therefore, upon success, the code gets vectorized in its entirety and on failure none of it does.

B. Unrealized Benefits of Vectorization

Vanilla SLP performs well on codes which benefit from vectorization uniformly; it does not realize the full benefits of vectorization when one or more subsets of the code benefit but others do not. SLP treats its graph as a single region and will only check whether vectorizing it as a whole can improve performance (step 8 in figure 1). However, if sections of the graph cannot be vectorized then they may harm vectorization, causing the whole graph to underperform. This can mean that the graph is not vectorized, or that the final vectorization is sub-optimal. Figure 2 shows an example and the solution to this problem.

In figure 2(c) we show the dependences graph for the code in figure 2(a). The value stored in A[] is the result of some computation on B[], C[], D[] and E[]. Note that the stores to A[i] and A[i+1] are to consecutive memory locations, as are the loads from B[i] and B[i+1]. However, the loads from the other arrays are not (the indices $2*i$ and $3*i$ guarantee this). These non-consecutive loads correspond to the nodes that are shown in red and cannot be vectorized using many vector architectures[2]. We now consider how SLP optimizes this code, as shown in figure 2(d).

As described in section II-A, we first locate the seed instructions, in this case the stores into A[i] and A[i+1], which are to adjacent memory locations. These form group 1 (G1, the root of the SLP graph in figure 2(d) that contains stores). Next the algorithm follows the data dependences upwards and tries to form more groups from instructions of same type. The rest of the groups (G2 to G6), consist of additions, multiplications and the loads from B[]. The loads from non-consecutive memory locations remain scalar. Each scalar node that produces data used by a vectorizable group requires a special instruction to insert the scalar data into the vector register. These insert instructions are represented by the orange nodes marked with the letter "i" in figure 2. Once data has been inserted into a vector register it can be reused whenever needed by reading from that register, hence only one set of insert instructions are required after the loads from C[] in figure 2(d); both G6 and G4 will read from the register written by the inserts.

The performance of the code is evaluated using the compiler's cost model that estimates the cost of each of the instructions. In LLVM's cost model for our target,

[2]Vectorization could be performed if the target ISA supported gather memory addressing, such as Intel's AVX2 [11]. However, implementations of SLP in LLVM and GCC would also need to be updated to take advantage of these instructions; currently only operations on contiguous memory locations are considered for vectorization.

each instruction shown has a cost of 1. The nodes of the graph in figure 2(d) are marked with the cost difference we should expect by vectorizing, as described in section II. For example, group G1 has a *CostDiff* of -1 (calculated as 1 - 2), meaning that if the two stores are vectorized they have a cost of 1, but if they remain scalar they have a cost of 2. Instructions that remain scalar have a *CostDiff* of 0 (their cost difference is not shown in the graph). Any additional nodes required to support the vectors, like the insert nodes, have a positive cost difference because they are not present if their consumers remain scalar. We then compute *TotalCost*, which is 0 for this graph, meaning that there are no benefits from vectorizing it.

The underlying reason that this graph is not profitable to vectorize is that the subgraph rooted at G4 upwards is harmful (it has a positive *TotalCost*). The SLP algorithm is unable to identify this problem as it treats the whole graph as single region. To rectify this, we propose TSLP, which isolates subsections of code that cause more harm than good. We refer to this as SLP *throttling* since it stops SLP from considering the entire graph.

Consider figure 2(e) where we have performed this throttling at the point of the cut, right above group G2. All instructions above the cut remain scalar. At the point of the cut we require additional insert instructions to bring in data from scalar registers to vectors. The *TotalCost* of this TSLP graph is -1, which means that it is beneficial to perform vectorization. We measured the performance of the SLP and TSLP versions of this code and found that TSLP is almost 17% faster than SLP (see section V for more information).

III. TSLP

TSLP is an automatic vectorization algorithm that improves performance by throttling the amount of vectorization that occurs. It discards regions of code that prove harmful to vectorization, even though they could actually be vectorized. Counter-intuitively, this results in smaller amounts of vectorized code yet higher performance. We first give an overview of the algorithm and then describe how throttling is performed in more detail.

A. Overview

An overview of the TSLP algorithm is shown in figure 1. As previously mentioned, the sections that belong to the original SLP algorithm are in white boxes while the TSLP-specific parts are highlighted in orange.

TSLP largely reuses SLP's infrastructure. It initially builds the SLP graph (step 2), rooted at the seeds (the graph is similar to that in figure 2(d)). The nodes in the graph represent groups of scalars that can be potentially vectorized; the edges represent data dependences. Unlike SLP, the graph in TSLP is used to explore the cost of different parts of the graph and to locate those that are harmful to vectorization. This exploration is performed by generating cuts within the

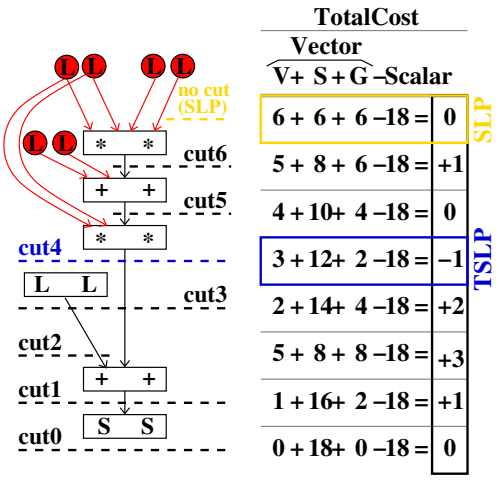


Figure 3. Throttling vectorization at "cut4" leads to the best performance, saving a cost of 1 compared to SLP. Vector costs V, S and G refer to costs for vector, scalar and gather instructions in the vectorized code.

original graph (steps 3 and 4) and evaluating the cost of vectorizing the subgraph below the cut. I.e., the nodes within it are considered vectorizable whereas the nodes beyond it are treated as scalars. Valid cuts are those that create a connected subgraph containing the root (seed instructions). The cost of each these subgraphs is evaluated (step 4) and that with the minimum cost is recorded (step 6). This process repeats (step 7) until we have explored all valid cuts within the original graph.

In comparison, vanilla SLP evaluates the cost of the graph only once and it performs its evaluation on the whole graph. It will not attempt to modify the graph in any way. The downside to this approach is that it will not remove any parts of the graph, even if they are harmful to the performance of the final code.

TSLP's final step is to check whether vectorization is beneficial for the best of the subgraphs (step 8). If it is then the algorithm will perform vectorization on the groups of scalars within the subgraph, below the cut (step 9). Otherwise the code remains unmodified.

B. (T)SLP Graph Construction

The graph for both SLP and TSLP (figure 1, step 2) is constructed bottom-up, starting from the seed instructions. Instructions that can be vectorized form a group and occupy a single node in the graph. Data dependences connect the nodes in the graph. The graphs get terminated (i.e., do not grow further upwards) once loads or non-vectorizable instructions are encountered. Each node contains information about the group, the most relevant for TSLP being:

- The scalar instructions that compose it;
- The dependences with other groups;
- A flag showing whether the group needs to gather (i.e., move data in from scalar registers); and

Algorithm 1. TSLP's method for generating throttled subgraphs.

```
1  /******  Generate Throttled Subgraphs ******/
2  /* Input : SLP Graph                       */
3  /* Output: gset set of throttled SLP Graphs*/
4
5  // Front-end function
6  gen_throttled_subgraphs (SLPGraph) {
7    root = SLPGraph.get_root()
8    Graph init_g
9    init_g.add_node (root)
10   gen_throttled_subgraphs_rec (init_g)
11 }
12
13 // Recursive function (back-end)
14 gen_throttled_subgraphs_rec (subg) {
15   // Early exit if already have SUBG in gset
16   if (subg in gset)
17     return
18   // Insert SUBG to set of throttled graphs
19   gset.insert(subg)
20   // Iterate over all SUBG's neighbors
21   for (neighbor in subg's neighbors) {
22     // Skip nodes already added
23     if (neighbor in subg)
24       continue
25     // SUBG_CP = SUBG + NEIGHBOR
26     Graph subg_cp = copy of subg
27     subg_cp.add_node (neighbor)
28     // Recurse using the subgraph copy
29     gen_throttled_subgraphs_rec (subg_cp)
30   }
31 }
```

- The costs associated with the group ($CostDiff$, $ScalarCost$, and $VectorCost$).

The group node format is shown in figure 4(a).

C. Throttling

The aim of TSLP is to find the most profitable subgraph to vectorize, given a graph of vectorizable instructions. It must stop vectorizing when the $TotalCost$ is at its minimum, before harmful code is included in the graph, which is caused by excessive gather/scatter instructions to move data from/to scalars. This is the job of the function that calculates all possible valid cuts within the original graph (step 3 of figure 1). The cut splits the original SLP graph into two subgraphs: one with vectorizable nodes and one with scalar nodes. A valid cut is a connected vectorizable subgraph that includes the root node (the seeds). A cut will introduce new insert instructions to bring in data to the vectors from scalars where data travels across the cut.

The example in figure 3 shows how throttling is explored for the graph in figure 2. The table on the right side of the figure shows how the cost is calculated in detail for each cut (*cut0* to *cut6*) and how the decision is made on where to throttle. SLP implicitly stops at the top (after *cut6*), since all inputs of the topmost node "[* *]" are non-vectorizable and there is nothing to gain by proceeding further. The total cost of performing vectorization ($TotalCost$) is calculated as the difference between the vector cost ($Vector$) and the scalar cost ($Scalar$). The vector cost is the sum of the costs of all the individual vector (V), scalar (S) and gather/scatter instructions (G) with the given cut. For example at *cut1* we have formed a single vector instruction containing the two stores "[S S]" so V=1. The rest of the code is scalar with a cost of 16 (S=16). At the point of the cut we need 2 insert instructions to insert the scalar data into the vector, therefore G=2. This leads to a vector cost of $1 + 16 + 2$. The scalar cost is constant and equal to the total cost of all the scalars (18); the cuts do not affect scalar code.

It is not uncommon for gathering (or scattering) costs to become higher the deeper we get into the graph, because the higher we get the more points where gathering (or scattering) is required. At the same time the higher the cut, the more vectors that we have formed (which leads to a significantly lower S). The total cost is negative if vectorization is profitable and positive if it is not. In the example, vectorization is profitable only for *cut4* with a cost of -1. In the general case, TSLP will explore all profitable cuts and select the one that gives the minimum cost. Compared to SLP, TSLP's decision to throttle vectorization at *cut4* in figure 3 not only saves a cost of 1, but it also makes vectorization profitable.

Algorithm: Algorithm 1 lists the core function to generate the set of all valid throttled subgraphs from the SLP graph. Its input is the SLP graph and its output is *gset*, the set of throttled subgraphs, that is the set of connected subgraphs that contain the root node. The outer edges of the throttled subgraphs correspond to the cut introduced by the throttling algorithm. The algorithm consists of two functions: the front-end function (line 6) and the back-end function (line 14) that calls itself recursively. The front-end creates a new graph *init_g* which contains only the initial root node (lines 7-9). Then it calls the recursive function using this new graph as a parameter (line 10).

The recursive function (line 14) is where the core of the computation lies. In short, this function saves the input subgraph into the output set, then expands it by adding each neighboring node one at a time and recursing. The computation begins with an early exit if the subgraph is already in the set (line 16) to avoid duplicate computation. Then the subgraph is inserted into the output set of subgraphs (*gset*, line 19). Next the function iterates over all neighbors of the subgraph (line 21) in order to add them to the subgraph. If the neighbor is already in the subgraph (since the graph is a DAG, this is a possibility) then it is skipped (line 23). Otherwise a copy of the subgraph is created (*subg_cp*, line 26), including the neighboring node (line 27), and the resulting subgraph is used as a parameter for recursion (line 29).

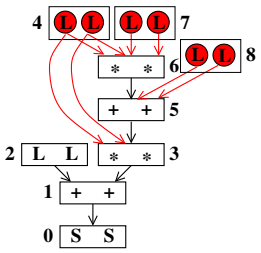
D. Implementation Details

We use LLVM's implementation of SLP and reuse its vector representation of the SLP graph (figures 4(a) and 4(b)), where each entry in the vector represents a node in the graph

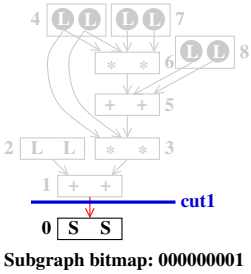
groups_vector:

Node format	0	1	2	3	4	5	6	7	8
Scalars	S,S	+,+	L,L	*,*	L,L	*,*	+,+	L,L	L,L
Dependencies	1,	2,3	–	4,5	–	6,8	4,7	–	–
NeedToGather	NO	NO	NO	NO	YES	NO	NO	YES	YES
Cost (V,S,Total)	1,2,−1	1,2,−1	1,2,−1	1,2,−1	2,0,+2	1,2,−1	1,2,−1	2,0,+2	2,0,+2

(a) Vector representation of TSLP graph

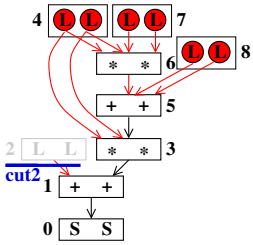


(b) Example TSLP graph



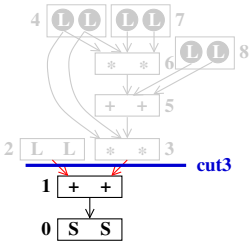
Subgraph bitmap: 000000001

(c) Subgraph 1



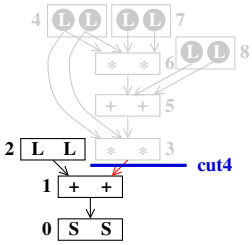
Subgraph bitmap: 111111011

(d) Subgraph 2



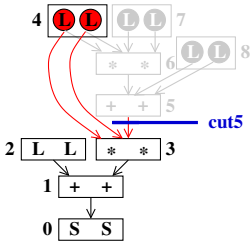
Subgraph bitmap: 000000011

(e) Subgraph 3



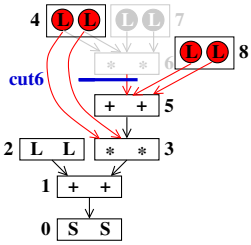
Subgraph bitmap: 000000111

(f) Subgraph 4



Subgraph bitmap: 000011111

(g) Subgraph 5



Subgraph bitmap: 010011111

(h) Subgraph 6

Figure 4. TSLP data structures and representation of subgraphs with bitmaps.

that is a vectorizable group. The vanilla SLP representation does not keep track of data dependences in the graph data structures: it uses the dependences in order to build the graph but nothing further. We extend the structure of each node to include the data dependence edges (figure 4(a): "Node format"). We use a simple short vector of integers as the representation for the edges: each number "N" in the vector represents a dependence with the groups_vector[N] node. For example, group 1 depends on both group 2 and group 3, so the dependences field in groups_vector[1] contains the vector 2,3.

In order to get a fast implementation, we need an efficient representation of the subgraphs, so we use bitmaps. The bit at position "i" shows whether groups_vector[i] node is part of the subgraph (figure 4(c) to 4(h)). Looking up node "i"s data in the groups_vector takes constant time. This representation also allows for constant time comparison of subgraphs and constant time checking whether the subgraph is already included in *gset* (the set of all subgraphs generated so far, algorithm 1, line 16).

In order to avoid complexity explosion for large input graphs, we limit exploration after reaching a threshold number of nodes in *gset*. After reaching this threshold, each new subgraph is constructed by appending *all* neighbors at once. For our experiments, we empirically set the threshold value to 50. We tried several threshold values ranging from 10 to 1000 and for each of these we evaluated TSLP. We found that for threshold values of about 30 or more, TSLP performed the best. Our experimentation showed that after this threshold value the gains start to level off. The intuition behind this is that finding cuts near the root are more important than near the leaves, because at the root you can potentially cut-off large chunks of the graph that would degrade performance.

E. Cost Model

Having throttled the graphs, we must decide whether to proceed with vectorization or to simply keep the code scalar. To do this without degrading performance, TSLP requires an accurate cost model for estimating the latency of each case. We use LLVM's cost model (*TargetTransformInfo*) without modification, which is already used by the original SLP pass. The cost is computed as the sum of all individual instruction costs. Each instruction cost is the execution cost of that instruction on the target processor (and for the x86/AVX2 target, this is usually 1). If there is scalar data flowing in or out of vector code, or vector data flowing in or out of the scalar code, then there is an extra cost to be added: the cost of the additional instructions required to perform these data movements. This is also target-dependent and its actual execution cost is reflected in the model.

This cost model is straightforward but basic; a more precise model, especially for simpler in-order architectures, would involve instruction scheduling on the target pipeline. The schedule depth would then provide a better estimate of the actual execution cost. In performing this analysis, the scalar cost on wide-issue architectures would be estimated more accurately, which would benefit SLP and TSLP alike. Although the existing cost model is not perfect, improving it further is beyond the scope of this work.

Kernel	Description
compute_rhs	Xi-direction fluxes (NPB2.3, BT)
mult_su3_mat_vec_sum_4dir	Su3 matrix by vector mult. (433.milc)
ewald_LRcorrection	Kernel from CPU2006 (435.gromacs)
compute_triangle_bbox	Triangle bounding box (453.povray)
lbm_handleInOutFlow	Kernel from CPU2006 (470.lbm)
shift_LRcorrection	Kernel from CPU2006 (435.gromacs)
motivation	Section II-B code

Table I
DESCRIPTION OF THE KERNELS.

F. Summary

We have presented TSLP, a straight-line code vectorization algorithm that throttles the scope of vectorization to improve performance. Throttling removes code that is harmful to performance even when it could have successfully been vectorized. The algorithm relies on generating a set of subgraphs from the original SLP graph. A cost model is applied to find the subgraph with the best performance that is worth vectorizing. Only code within the subgraph is vectorized; all other code is left in scalar form.

IV. EXPERIMENTAL SETUP

We implemented TSLP in the trunk version of the LLVM 3.6 compiler [6] as an extension to the existing SLP pass. The front-end of the compiler is *clang* and the back-end is *llc*. We evaluated TSLP on several kernels extracted from various C/C++ benchmarks from SPEC CPU 2006 [12] and NPB2.3-C [13]. A brief description of them is given in table I. We also included the motivation example code from section II-B as a reference. We compiled the workloads with the following options: *-O3 -allow-partial-unroll -march=core-avx2 -mtune=core-i7* and refer to these options as O3. We evaluated the following cases:

- O3 with loop, SLP and TSLP vectorizers disabled (O3)
- O3 with only the SLP vectorizer enabled (SLP)
- O3 with only the TSLP vectorizer enabled (TSLP)

Our target system was an Intel Core i5-4570 at 3.2GHz with 16GB of RAM and an SSD hard drive, running Linux 3.10.17 and glibc 2.17. We ran each kernel in a loop for as many iterations as required such that they executed for several hundred milliseconds.

V. RESULTS

For a complete evaluation of the optimization we measured a wide range of metrics: performance (section V-A), static cost savings according to the cost model (section V-B), subgraph exploration overhead (section V-C) and the utilized vector lanes (section V-D). Finally, in section V-E, we describe two case studies in detail to explain why TSLP improves performance.

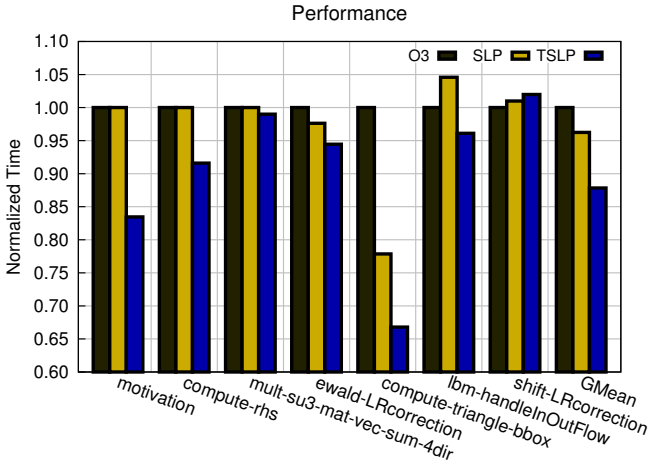


Figure 5. Execution time of the kernels under O3, SLP and TSLP, normalized to the baseline O3.

A. Performance

Execution time on a real machine, normalized to O3, is shown in figure 5. These show that TSLP improves performance over SLP in the majority of the kernels. The first three kernels (motivation, compute-rhs and mult-su3-mat-vec-sum-4dir) benefit from TSLP but not from SLP. This happens because the SLP graphs include sequences of code that harm vectorization so much that the cost model shows scalar code is better than vector. TSLP throttles the graph before these harmful sequences, leading to code that actually benefits from vectorization.

The next two kernels (ewald-LRcorrection and compute-triangle-bbox) perform better than O3 when SLP is enabled. In both cases the cost of vectorizing the whole region is less than the scalar cost, which is why SLP succeeds. Nevertheless, throttling (TSLP) removes code that incurs high overheads when vectorized, leading to better performance.

Finally there are two benchmarks where the cost-model is not accurate enough and actually causes performance degradation. In lbm-handleInOutFlow, this results in SLP being slower than O3 even though the cost-model has calculated that vectorization is profitable. TSLP, having removed the harmful code, performs better than both O3 and SLP, increasing performance. Of particular interest is shift-LRcorrection, where both SLP and TSLP perform slightly worse than O3, with TSLP being slightly worse than SLP. This can once again be attributed to an imprecise cost model that assigns lower cost to both SLP and TSLP even though scalar code proves to be faster when actually executed.

TSLP provides an average 12% improvement in performance over O3 across all kernels, which is approximately 9% better than SLP, where SLP is already 4% faster than O3.

B. Static Cost Savings

Although the performance results are a good indication of whether TSLP works well, static results are also important because they show how the technique works without noise from an imprecise cost model. Static results, based on the total cost of each vectorized graph, show how SLP and TSLP compare for one particular cost model. TSLP always minimizes the total cost of a graph, whereas SLP uses the whole graph's cost to determine whether to vectorize or not. Improving the cost model would therefore benefit both algorithms, but TSLP would always be at least as good as SLP, and usually better.

Figure 6 shows the total static costs for scalar, SLP and TSLP code according to LLVM's cost model. To get a more meaningful measurement, the costs have been normalized to the total $ScalarCost$ for each workload. This shows that, regardless of real performance, TSLP always succeeds in improving the cost of vectorization. On average, TSLP provides approximately a 19% decrease in the code's cost, while SLP achieves a 9% reduction.

The static costs in figure 6 do not show a strong correlation with the real performance results from figure 5. There are two factors that contribute to this:

1) The accuracy of the cost model: even if we save in static cost, there is always the possibility that the code generated and executed could be slower;
2) The coverage of the vectorized code as a part of the total run-time: it could be that the vectorized region accounts for only a small fraction of the program's execution time.

The kernels lbm-handleInOutFlow and shift-LRcorrection are examples of (1). On the other hand, ewald-LRcorrection and mult-su3-mat-vec-sum-4dir are examples of (2). In ewald-LRcorrection, only 2 minor functions get fully vectorized and these account for a small percentage of the total runtime, whereas in mult-su3-mat-vec-sum-4dir the main loop is mainly scalar with only a very small region being vectorized.

Nevertheless there is some predictable behavior. The workloads where SLP does not provide any cost savings do not show any run-time performance change (motivation, compute-rhs, mult-su3-mat-vec-sum-4dir). Also when both SLP and TSLP contribute to static cost savings, we tend to see performance improvements for both.

C. Number of Subgraphs Explored

To provide an estimate of the additional complexity that TSLP introduces, we measured the number of subgraphs that TSLP generates and evaluates (figure 7). This is the total number of subgraphs generated for all regions that are candidates for vectorization, both for successful and unsuccessful attempts. The number shown varies considerably across benchmarks, from a few tens (compute-triangle-bbox) to a few thousands (lbm-handleInOutFlow). The reason is that the number of subgraphs depends on:

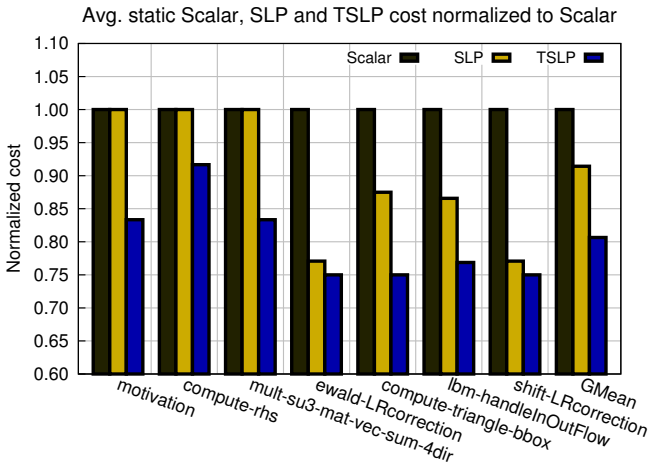


Figure 6. Static cost of Scalar, SLP and TSLP normalized to the scalar cost of the region.

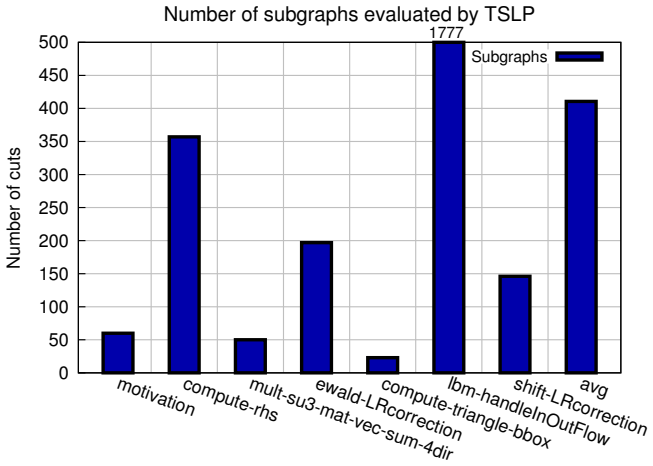


Figure 7. The total number of subgraphs explored by TSLP. Each subgraph corresponds to a cut of the SLP graph.

1) The amount of unrolling: the more code that gets unrolled the more candidates for SLP/TSLP;
2) The number of times SLP gets triggered: the more seeds that SLP starts from, the more times SLP, and consequently TSLP, get triggered;
3) The size and type of the TSLP graph: the larger the graph, the greater the number of possible cuts. Additionally, the more connected the graph, the greater the number of available cuts.

The average number of subgraphs evaluated in our benchmarks is around 400, which does not add significant complexity into the vectorization pass.

Figure 8 provides more insight into the subgraphs explored by TSLP for each of the kernels. Each subfigure

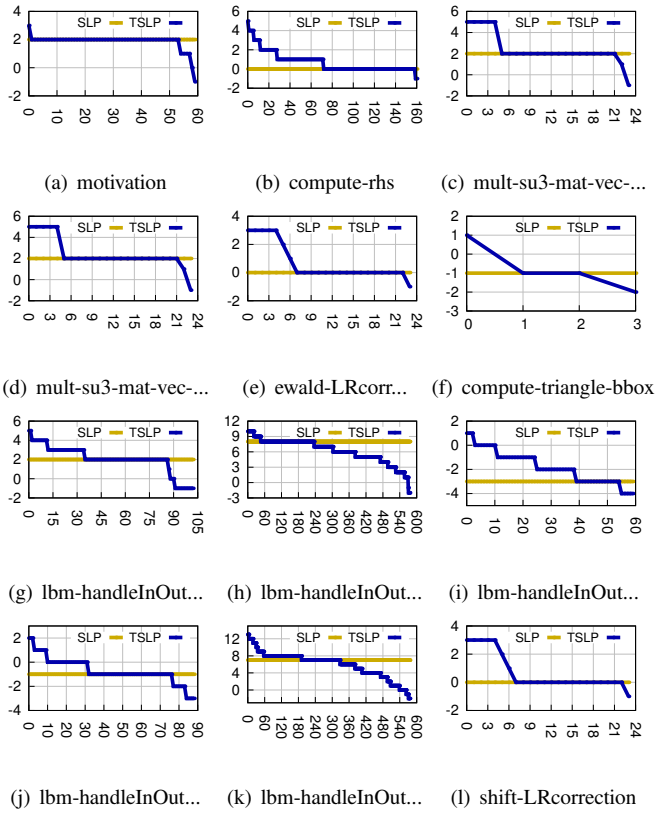


Figure 8. *TotalCost* (vertical axis) for each subgraph explored by TSLP (horizontal axis) that improves upon SLP. The subgraphs are sorted in terms of decreasing cost.

shows the cost gains (*TotalCost*) across all subgraphs explored for a different vectorization attempt. We show only graphs that are successfully vectorized and where TSLP outperforms SLP, which is shown for reference (straight horizontal line). The subgraphs are sorted in terms of decreasing cost (increasing performance) along the horizontal axis and the vertical axis gives that cost. The subgraph chosen by TSLP is always that on the far right, with minimum cost.

A range of behaviors are shown. As can be seen from the figure, TSLP explores a varying number of different subgraphs at each attempt and there are always graphs that have a positive *TotalCost*, as well as those with a negative value. In three quarters of the graphs, SLP does not vectorize the code because the full graph has a cost of 0 or more, whereas TSLP manages to pick a subgraph that is beneficial to vectorize. The most extreme case is shown in figure 8(h) where SLP's graph has a cost of 8, but TSLP manages to reduce that to -2.

D. Vector Widths

We measured the average vector widths generated by SLP and TSLP (shown in figure 9), to gain an understanding of how much vectorization saves on scalar computation.

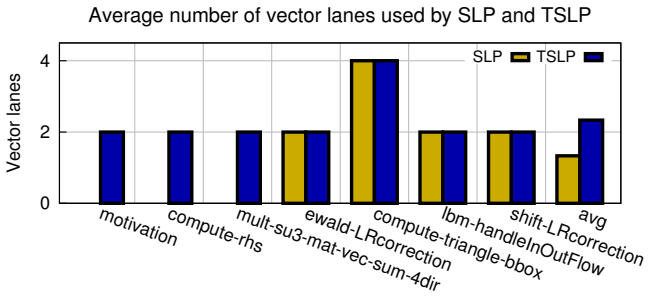


Figure 9. Average SLP and TSLP vector widths. The missing bars suggest that the vectorization technique did not succeed even once.

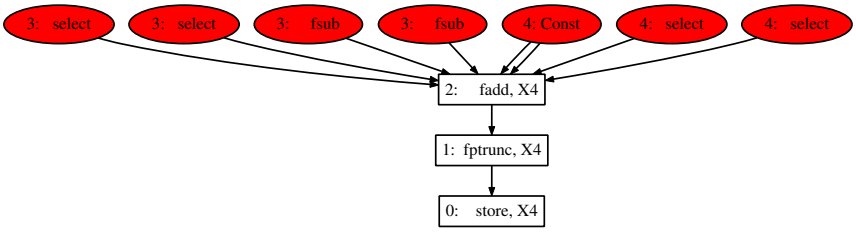
For each vectorization performed, we measured the number of lanes enabled, then summed them and divided by the number of times vectorization succeeded. This gives an average value for the benchmark. The plots show that for the first three workloads SLP does not get triggered, at all. This validates the observation in section V-A that the first three workloads do not trigger SLP, whereas they do trigger TSLP. For the remaining workloads both SLP and TSLP get triggered and they both generate vectors of equal widths.

As expected, the kernel with the highest performance improvement from both SLP and TSLP is compute-triangle-bbox with an average vector width of 4. This achieves a speedup of 33% compared to O3, or 14% compared to SLP. This should be of no surprise as the wider the vectors, the larger the performance gap between scalar and vector execution. The most common vector width, however, is 2, meaning that these kernels do not contain sufficient similar scalar code to enable optimal use of the vector units.

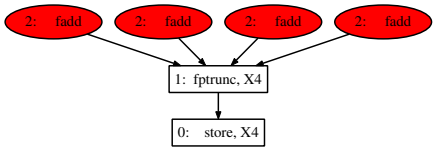
E. Case Studies

In this section we take a closer look at the actual graphs generated by the compiler for both SLP and TSLP when compiling two of the workloads: compute-triangle-bbox (the best performing of the workloads) and compute-rhs (a kernel where SLP does not vectorize anything, but TSLP does). Figures 10 and 11 show these graphs, where red elliptic nodes are scalar instructions and clear rectangular nodes are vector. The vector nodes also show their width in the form "X<width>", i.e., "X2" or "X4" for widths 2 and 4 respectively.

We first focus on compute-triangle-bbox which has a vector width of 4. Figure 10(a) shows the SLP graph has a *TotalCost* of -1, meaning that vectorization is profitable. This comes from a *VectorCost* of 17 (3 vectors + 8 gathers + 6 scalars) and a *ScalarCost* of 18 ($3 \times 4 +$ 6 scalars). Note that the gathers (8) are more than the scalars (6) because one of the inputs of node 2 (*fadd*) comes from both constants and regular instructions (two *Const* and two *Select* nodes). This, according to the target, requires 4 gather instructions. Although vectorization is beneficial,

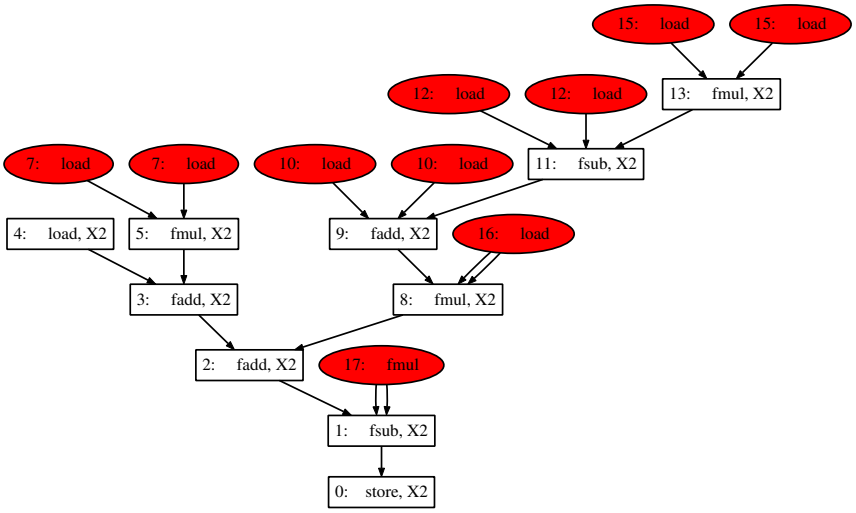


(a) SLP: $(3+8+6)-(3\times 4+6)=-1$

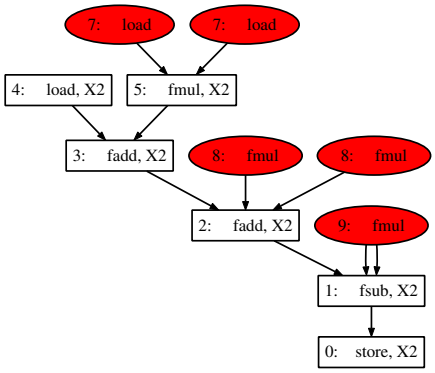


(b) TSLP: $(2+4+4)-(2\times 4+4)=-2$

Figure 10. SLP and TSLP graphs for compute-triangle-bbox. Rectangles are instructions vectorized across 4 lanes, ovals are scalar and need gather instructions to use their outputs in the vector code. Costs are $VectorCost - ScalarCost$. TSLP improves the graph's performance by saving a cost of 1.



(a) SLP: $(10+10+10)-(10\times 2+10)=0$



(b) TSLP: $(6+5+5)-(6\times 2+5)=-1$

Figure 11. SLP and TSLP graphs for compute-rhs. Rectangles are instructions vectorized across 2 lanes, ovals are scalar and need gather instructions to use their outputs in the vector code. Costs are $VectorCost - ScalarCost$. With TSLP the graph becomes profitable.

the topmost node (*fadd*) must gather from 6 out of its 8 inputs, meaning that there are significant overheads and sub-optimal performance. Using TSLP (figure 10(b)) we can see that throttling vectorization early reduces these overheads, leading to speedups. The graph has been cut between nodes 1 (*fptrunc*) and 2 (*fadd*), which gives a $TotalCost$ of -2, from a $VectorCost$ of 10 (2 vectors + 4 gathers + 4 scalars) and a $ScalarCost$ of 12 (2×4 + 4 scalars).

The second workload, compute-rhs, has a much larger, 2-wide SLP graph (figure 11(a)) which has a $TotalCost$ of 0 and is therefore not vectorized by SLP ($VectorCost$ of $10+10+10=30$ and $ScalarCost$ of $10\times 2+10=30$). Note that the cost of gathering is 10, not 12 (if we simply count the edges from scalar nodes). The reason is that nodes 16 and 17 both have two outgoing edges. This can be represented as a single broadcast instruction in the target ISA, with a cost of 1. However, TSLP (figure 11(b)) can identify a portion of the graph that is harmful to vectorization and throttle it early with a cut between nodes 2 (*fadd*) and 8 (*fmul*). In this case the $TotalCost$ of TSLP is -1, from a $VectorCost$ of $6+5+5=16$ and a $ScalarCost$ of $6\times 2+5=17$, meaning that vectorization should proceed. Once again the gather cost is lower than expected (5 instead of 6) since node 9 needs a single broadcast instruction to insert the value it produces to both lanes of a vector register.

F. Summary

We have evaluated TSLP in comparison to O3 and SLP, showing that TSLP's performance is, on average, approximately 12% better than O3 and 9% better than SLP. According to the static cost evaluation, TSLP saves 19% of the scalar cost, and on average 12% more than SLP. Depending on the workload, TSLP explores up to several thousand subgraphs to identify the best place to cut the graph.

VI. Related Work

A. Vector Processing

Various commercial (for example [14], [15]) and experimental (e.g., [16]) wide vector machines have been built in the past. These machines were used to accelerate scientific vector code, usually written in some dialect of Fortran.

More recently short SIMD vectors have become a standard feature of all commodity processors for most desktop and mobile systems. All major processor manufacturers (Intel, AMD, IBM and ARM) support some sort of short-vector ISA (e.g., MMX/SSE*/AVX/AVX2 [17], 3DNow! [18], VMX/AltiVec [19] and NEON [20] respectively). These ISAs are constantly under improvement and get updated every few years with more capable vector instructions and/or wider vectors.

Modern graphics processors (GPUs), like old vector machines, implement hardware vectorization [21]. They do so by executing groups of 32 (on Nvidia) or 64 (on AMD) adjacent threads in *warps* in lock-step. Such large vector widths are possible thanks to data-parallel input languages like CUDA or OpenCL, where the programmer explicitly exposes the available parallelism to the hardware. This effectively overcomes intrinsic limitations of compiler-based analysis, leading to substantial runtime and energy improvements over traditional CPU execution for suitable workloads.

B. Loop Vectorization

Loops are the main target of vectorization techniques [22]. The basic implementation strip-mines the loop by the vector factor and widens each scalar instruction in the body to work on multiple data elements. Early works of Allen and Kennedy on the Parallel Fortran Converter [23], [24], the works of Kuck et al. [25], Wolfe [26] and Davies et al. [27] solve many of the fundamental problems of automatic vectorization. Numerous improvements to the basic algorithm have been proposed in the literature and implemented in production compilers. Efficient run-time alignment has been proposed by Eichenberger et al. [28], while efficient static alignment techniques were proposed by Wu et al. [29]. Ren et al. [30] propose a technique that reduces the count of data permutations by optimizing them in groups. Nuzman et al. [1] describe a technique to overcome non-contiguous memory accesses and a method to vectorize outer loops without requiring loop rotation in advance [2].

An evaluation of loop vectorization performed by Maleki et al. [31] shows the limits of current implementations. State-of-the-art compilers, like GCC and ICC, can vectorize only a small fraction of loops in standard benchmarks like *Media Bench*. The authors explain these poor results as (1) lack of accurate compiler analysis, (2) failure to perform preliminary transformations on the scalar code and (3) lack of effective cost models.

C. SLP Vectorization

Super-word level parallelism (SLP) has been recently introduced to take advantage of SIMD ISAs for straight-line code. Larsen and Amarasinghe [3] were the first to present an automatic vectorization technique based on vectorizing parallel scalar instructions with no knowledge of any surrounding loop. Variants of this algorithm have been implemented in all major compilers including GCC and LLVM [10]. This is the state-of-the-art SLP algorithm and in this paper we use its LLVM implementation as a baseline for comparison and as a starting-point for our TSLP work.

Shin et al. [32] introduce an SLP algorithm with a control-flow extension that makes use of predicated execution to convert the control flow into data-flow, thus allowing it to become vectorized. They emit `select` instructions to perform the selection based on the control predicates.

In another work, Porpodas et al. [4] apply SLP after first padding the scalar code with redundant instructions, to convert non-isomorphic instruction sequences into isomorphic ones. They identify differences between graphs of scalar code using the maximum common subgraph between all sequences of instructions, then add all differences into the code to generate the minimum common supergraph. New `select` instructions are also inserted to choose the correct path through the graph, and optimizations are introduced to remove any redundant `select`s that are added.

Other straight-line code vectorization techniques which depart from the SLP algorithm have also been proposed in the literature. A back-end vectorizer in the instruction selection phase based on dynamic programming was introduced by Barik et al. [33]. This approach is different from most of the vectorizers as it is close to the code generation stage and can make more informed decisions on the costs involved with the instructions generated. An automatic vectorization approach that works on straight-line code is presented by Park et al. [34]. It succeeds in reducing the overheads associated with vectorization such as data shuffling and inserting/extracting elements from the vectors. Holewinsky et al. [35] propose a technique to detect and exploit more parallelism by dynamically analyzing data dependences at runtime, and thus guiding vectorization. Liu et al. [36] present a vectorization framework that improves SLP by performing a more complete exploration of the instruction selection space while building the SLP tree.

None of these approaches identify the problem of atomic evaluation of vectorization performance. TSLP is the first approach that tackles this problem by throttling the scope of vectorization to improve performance.

D. Vectorization Portability

Another relevant issue for vectorization is portability across platforms. The various types of SIMD instructions available on different architectures require the definition of suitable abstractions in the compiler's intermediate representation. These must be general enough to embrace various vectorization patterns without sacrificing the possibility of generating efficient code. Nuzman et al. targeted this problem by proposing improvements to the GIMPLE GCC intermediate representation [37] and through JIT compilation [38].

VII. Conclusion

In this paper we presented TSLP, a novel automatic vectorization algorithm that improves the performance of vectorized code. It achieves this by throttling the graphs that are built when assessing the scalar instructions to be vectorized. Instead of greedily accepting all instructions that are vectorizable, it explores the subgraphs to choose the one that will realize the most benefits, pruning out code that is harmful to vectorization. An evaluation of our technique in an industrial-strength compiler and on a real machine shows improved coverage and performance gains across a range of kernels, achieving speedups of 9% on average.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through grant reference EP/K026399/1. Additional data related to this publication is available in the data repository at https://www.repository.cam.ac.uk/handle/1810/250381.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

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

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

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

[11] Intel Corporation, "Intel advanced vector extensions programming reference," 2011.

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

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

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

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

[16] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: IRAM," *Computer*, vol. 30, no. 9, 1997.

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

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

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

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

[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, 2008.

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

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

[24] ——, "Automatic translation of Fortran programs to vector form," *Tranactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 4, 1987.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Dynamic trace-based analysis of vectorization potential of applications," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[36] 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.

[37] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[38] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks, "Vapor SIMD: Auto-vectorize once, run everywhere," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2011.