Vectorization-Aware Loop Unrolling with Seed Forwarding

Citation for published version:
Rocha, RR, Porpodas, V, Petoumenos, P, Goes, LFW, Wang, Z, Cole, M & Leather, H 2020, Vectorization-Aware Loop Unrolling with Seed Forwarding. in *CC 2020: Proceedings of the 29th International Conference on Compiler Construction.* Association for Computing Machinery (ACM), pp. 1-13, ACM SIGPLAN 2020 International Conference on Compiler Construction , San Diego, United States, 22/02/20. https://doi.org/10.1145/3377555.3377890

Digital Object Identifier (DOI):
10.1145/3377555.3377890

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
CC 2020: Proceedings of the 29th International Conference on Compiler Construction

Vectorization-Aware Loop Unrolling with Seed Forwarding

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

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

Pavlos Petoumenos
University of Manchester, UK
pavlos.petoumenos@manchester.ac.uk

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

Zheng Wang
University of Leeds, UK
z.wang5@leeds.ac.uk

Murray Cole
University of Edinburgh, UK
mic@inf.ed.ac.uk

Hugh Leather
University of Edinburgh, UK
hleather@inf.ed.ac.uk

Abstract

Loop unrolling is a widely adopted loop transformation, commonly used for enabling subsequent optimizations. Straight-line-code vectorization (SLP) is an optimization that benefits from unrolling. SLP converts isomorphic instruction sequences into vector code. Since unrolling generates repeatead isomorphic instruction sequences, it enables SLP to vectorize more code. However, most production compilers apply these optimizations independently and uncoordinated. Unrolling is commonly tuned to avoid code bloat, not maximizing the potential for vectorization, leading to missed vectorization opportunities.

We are proposing VALU, a novel loop unrolling heuristic that takes vectorization into account when making unrolling decisions. Our heuristic is powered by an analysis that estimates the potential benefit of SLP vectorization for the unrolled version of the loop. Our heuristic then selects the unrolling factor that maximizes the utilization of the vector units. VALU also forwards the vectorizable code to SLP, allowing it to bypass its greedy search for vectorizable seed instructions, exposing more vectorization opportunities.

Our evaluation on a production compiler shows that VALU uncovers many vectorization opportunities that were missed by the default loop unroller and vectorizers. This results in more vectorized code and significant performance speedups for 17 of the kernels of the TSVC benchmarks suite, reaching up to 2× speedup over the already highly optimized -O3. Our evaluation on full benchmarks from FreeBench and MiBench shows that VALU results in a geo-mean speedup of 1.06×.

CCS Concepts • **Software and its engineering → Compilers**.

Keywords SIMD, SLP, Auto-Vectorization, Loop Unrolling

ACM Reference Format:
Rodrigo C. O. Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís F. W. Góes, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Vectorization-Aware Loop Unrolling with Seed Forwarding. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377555.3377890

1 Introduction

Modern high-performance processors include short SIMD vector units to support higher computational throughput. Making effective use of the vector units is critical for extracting maximum performance from these processors.

There are two general classes of vectorizers. Traditional loop-based vectorizers [2, 3] detect instructions that can be vectorized across loop iterations. Superword-Level Parallelism (SLP) [23, 43], on the other hand, are not limited by the loop structure. They identify isomorphic groups of instructions that can be vectorized within any straight-line code sequence, whether in a loop body or outside loops altogether.

Loop unrolling is commonly applied before the SLP vectorization pass. Unrolling the loop body generates straight-line code with repeating computational and memory access patterns. This makes finding vectorizable instructions much more likely. The motivation for this work comes from the realization that, in state-of-the-art compilers, unrolling and SLP vectorization are completely independent and uncoordinated. Unrolling is guided by its own heuristic, mainly considering how unrolling affects code size. As a result, this

CC '20, February 22–23, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7120-9/20/02…$15.00
https://doi.org/10.1145/3377555.3377890

heuristic makes good unrolling decisions with regards to vectorization only incidentally.

In this work, we propose Vectorization-Aware Loop Unrolling (VALU), a novel unrolling approach that offers a strong coupling with SLP vectorization. Our approach is two-fold. First, VALU uses a novel analysis, named *Potential SLP*, that performs vectorization and profitability analyses that would be performed by SLP as if the loop had been unrolled (without unrolling it yet). If vectorization is deemed profitable, the loop is then actually unrolled by a factor that maximizes utilization of the vector units on the target architecture. Second, VALU has a seed forwarding mechanism that keeps track of unrolled copies of vectorizable seed instructions identified in the original context and forwards them directly to the SLP vectorizer. VALU knows by definition that unrolled instructions are isomorphic, while the SLP vectorizer needs to discover which group of instructions in the unrolled loop will lead to isomorphic use-def graphs, without an expensive search. By forwarding this information, we can bypass SLP's greedy seed collection, improving vectorization.

Our approach uncovers many vectorization opportunities that were completely missed by LLVM's loop unroller. Unlike traditional unrolling, VALU only unrolls loops when enough code will be vectorized away. Therefore, it can afford to make aggressive unrolling decisions, when that is estimated to pay off. When evaluated on the TSVC [6] benchmark suite, VALU improves SLP vectorization by up to 6× and 30% on average, enabling SLP to outperform the loop vectorizer for 26 kernels of the TSVC suite. VALU also improves performance by up to 2×, with a geometric mean of 5%, compared to the highest optimization setting (`-O3`). We have also evaluated VALU on two full benchmarks, FreeBench and MiBench, where it achieves a geo-mean percentual speedup of 6%.

To summarize, our main contribution is providing a strong coupling between loop unrolling and the SLP vectorizer, with a two-way communication channel between the two passes.

- We enable much better vectorization by analyzing instructions in the rolled context.
- We choose better unroll factors by knowing how vectorization will be applied.
- We find better vectorization seeds before loop unrolling and forward them directly to the SLP vectorizer.

2 Background

2.1 Loop Unrolling

Loop unrolling creates multiple copies of the loop body, in order to perform multiple iterations at once, adjusting the loop control accordingly to preserve its original semantics. The number of copies is called the *unrolling factor* [9, 16, 30]. The immediate benefit comes from reducing the loop control overhead. By converting loops into straight-line code, loop unrolling also enables or improves subsequent optimizations.

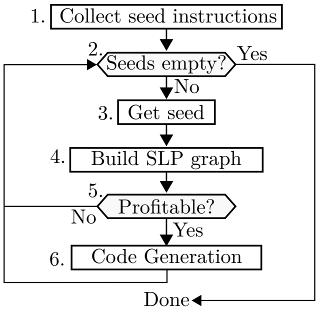


Figure 1. Overview of Bottom-up SLP.

Excessive unrolling may impair performance, mainly due to increased register pressure and instruction cache misses [10, 46]. For this reason, most unrolling heuristics will not unroll a loop above a certain factor, if the estimated size of the unrolled loop body exceeds an empirically set threshold.

2.2 SLP Vectorization

Superword-Level Parallelism (SLP) is a straight-line-code vectorizer that was first introduced by Larsen and Amarasinghe [23]. SLP tries to find isomorphic instruction sequences and vectorize them if profitable. Some variants of this algorithm have been implemented in production compilers, with Bottom-Up SLP [43] being widely adopted due to its low runtime overhead and its good coverage.

Figure 1 shows a diagram of the bottom-up SLP algorithm [43]. It first identifies instructions, called *seed instructions*, that are likely to form vectorizable sequences, such as *stores* instructions or reductions trees (step 1). Starting from a group of seeds (step 3), the algorithm follows their use-def chains towards their operands to grow the SLP graph (step 4). Once this process encounters instructions that cannot form a vectorizable group (e.g, due to non-matching opcodes), it forms a non-vectorizable group and it stops further exploring this path. Non-vectorizable groups indicate that scalar-to-vector data movement is required.

Next, the algorithm estimates the profitability of vectorizing the instructions in the SLP graph (step 5). The total profit is the one of converting groups of scalar instructions into vectors minus the overhead of gathering the inputs of the vector instructions. If profitable, SLP replaces each group of scalar instructions in the graph with their equivalent vector version (step 6). Otherwise, the code remains unmodified. The process then continues with the next seed group until all seeds have been explored (step 2).

3 Motivating Example

In this section, we present an example to demonstrate that existing unrolling heuristics are ineffective in exposing vectorization opportunities for SLP. Instead, an ideal loop unroller would be able to identify exactly which loops are profitable

```
float Af[N], Bf[N], Cf[N], Df[N], Ef[N];
double Ad[N], Bd[N], Cd[N], Dd[N], Ed[N];
for (int k = 0; k < N; k++) {
 Af[k] = Bf[k]*Cf[k] + Df[k]*Ef[k];
 Ad[k] = Bd[k]*Cd[k] + Dd[k]*Ed[k];
}
```

(a) Source code of a loop that is unrolled twice by the default loop unroller.

```
for (int k = 0; k < SIZE; k+=2) {
 Af[k:k+1] = Bf[k:k+1]*Cf[k:k+1] +
             Df[k:k+1]*Ef[k:k+1];
 Ad[k:k+1] = Bd[k:k+1]*Cd[k:k+1] +
             Dd[k:k+1]*Ed[k:k+1];
}
```

(b) After unrolling the loop by a factor of 2, the SLP vectorizer will generate this sub-optimal vectorized code, underutilizing the vector units available in the target architecture.

```
for (int k = 0; k < SIZE; k+=8) {
 Af[k:k+7]   = Bf[k:k+7]*Cf[k:k+7] +
               Df[k:k+7]*Ef[k:k+7];
 Ad[k:k+3]   = Bd[k:k+3]*Cd[k:k+3] +
               Dd[k:k+3]*Ed[k:k+3];
 Ad[k+4:k+7] = Bd[k+4:k+7]*Cd[k+4:k+7] +
               Dd[k+4:k+7]*Ed[k+4:k+7];
}
```

(c) The loop could instead be unrolled by 8 times, resulting in a SLP-vectorized code that is around 2× faster than when unrolling only twice, by fully utilizing the vector units available.

Figure 2. Example where the default loop unroller uses a sub-optimal unrolling factor.

to be vectorized by SLP after unrolling and what is the unrolling factor that uncovers enough code to maximize the utilization of the target vector units.

Figure 2a shows a loop with a small loop body, just two statements long. Loop unrolling uses its heuristics to determine the unrolling factor, comparing the expected code size increase against a threshold. In this particular example, LLVM unrolls it only by a factor of two, because the cost of unrolling it further exceeds the threshold. Although the unrolled loop may be vectorized, as shown in Figure 2b, this can result in poor utilization of the vector units.

A vectorization-aware technique should be able to anticipate that the unrolled loop can be vectorized by SLP and select an unroll factor that maximizes performance. In our example of Figure 2a, we achieve this by unrolling as many times as needed by the smallest data type to fill the vector register. For a 256-bit vector length and a smallest data type of 32 bits (array `Ai`), this leads to an unrolling factor of 8. After unrolling and vectorizing with SLP, we get the code in Figure 2c, which does better than that of Figure 2b.

Code with bigger data types will also be unrolled more times than can fit in the vector registers. For example, the instructions operating on doubles will still be unrolled 8 times instead of the just 4 times. This is not a concern, however, because the vectorizer will generate more vector instructions for them, twice as many in this case, as shown in Figure 2c.

For slightly bigger loops or slightly lower unroll thresholds, the default loop unroller may completely bail-out on unrolling and prevent SLP from vectorizing the loop altogether. Just changing the unroll threshold to improve vectorization is not a reasonable strategy. Loops can be vectorizable regardless of their sizes, then some vectorization opportunities might be missed for any unrolling threshold. At the same time, high thresholds would unroll scalar loops by very large factors impacting performance. SLP vectorization cannot rely on the default loop unroller, because its heuristics may decide not to unroll profitable loops for vectorization.

The end result shown in Figure 2c also differs from that produced by LLVM's loop vectorizer. The loop vectorizer selects a single vector length, based on the largest data type, for the whole loop body, so that all instructions in the loop can be vectorized with the same vector length. The ideal loop unroller should choose the best unrolling factor to maximize performance. Usually, the version with mixed vector lengths tends to be faster as it better utilizes the vector units [40].

4 Vectorization-Aware Loop Unrolling

In this section, we describe our vectorization-aware loop unrolling (VALU). The core idea is to perform an analysis on the original loop that looks for code that could be vectorized by SLP once the loop gets unrolled. After unrolling, VALU forwards to SLP the instructions that are profitable for vectorization, bypassing SLP's greedy seed collection.

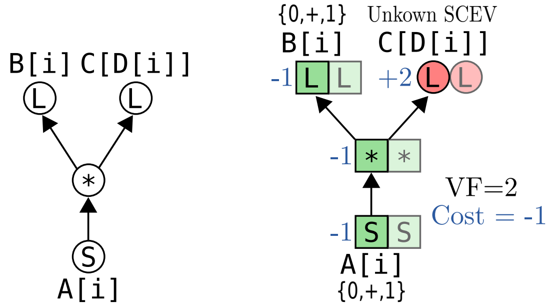
4.1 Potential SLP Graph

In order to identify if loop unrolling would be beneficial for vectorization, VALU performs an analysis inspired by the SLP algorithm. Traditional SLP analysis builds an SLP graph that represents the combined use-def graphs of the groups of scalar instructions that are considered for vectorization[1]. VALU uses a different data-structure, called *Potential SLP graph*. This is built from one use-def graph of the scalar instructions in the rolled loop. However, Potential SLP graph reproduces the state of an equivalent SLP graph that would be built if the loop was unrolled by a specific unrolling factor. For example, Figure 3c shows the Potential SLP graph obtained by VALU when applied to the loop from Figure 3a, which contains the use-def graph shown in Figure 3b. The Potential SLP graph is able to estimate the same profitability cost as the one computed by the SLP graph in Figure 3e, which is built from the already unrolled loop shown in Figure 3d. This is a key aspect of how VALU is able to precisely unroll loops that are profitable for SLP vectorization.

[1] Each node in the SLP graph contains the group of scalars that are considered for vectorization, and the edges represent the combined dependencies among the groups of scalars.

```
for (int i = 0; i < N; ++i) {
 A[i] = B[i]*C[D[i]];
}
```

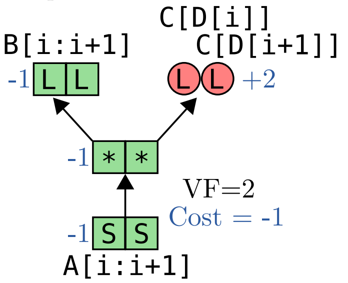
(a) Source code.



(b) Use-def graph. **(c)** Potential SLP graph.

```
for (int i = 0; i < N; i+=2) {
 A[i]   = B[i]   * C[D[i]];
 A[i+1] = B[i+1] * C[D[i+1]];
}
```

(d) Loop unrolled by a factor of two.



(e) SLP graph built from the unrolled loop.

🔴 Non–Vectorizable 🟩 Vectorizable +/–#Cost

```
for (int i = 0; i < N; i+=2) {
 int id0 = D[i];
 int id1 = D[i+1];
 CVec = { C[id0], C[id1] };
 A[i:i+1] = B[i:i+1]*CVec;
}
```

(f) Partially vectorized loop by SLP after loop unrolling.

Figure 3. Example shows how the Potential SLP graph obtained by VALU mirrors the SLP's analysis in the unrolled loop.

Figure 4 shows a diagram of the algorithm for the VALU heuristic. VALU starts by scanning the loop body and collecting seed instructions. At the moment, we only consider *store* instructions and reduction trees, but other instructions can also be used as seeds. Contrary to SLP that only collects vectorizable store instructions, VALU collects all store instructions, as detailed in Section 4.7. After collecting these seed instructions, we calculate the best vectorization factor (VF) based on the data type of the seed instructions. This factor is required for building the Potential SLP graph. VALU selects a vectorization factor that maximizes the utilization of vector units in the target architecture. This can be computed based on the bit-size of the instruction's data type and the maximum size of vectors supported by the target architecture. For example, a 64-bit *store* on a 256-bit vector architecture corresponds to a vectorization factor of 4. Please, note that the vectorization factor selected particularly for a Potential SLP graph corresponds to the desired unrolling factor of the enclosing loop. Consequently, VALU's unrolling factor is bounded by the target-vector length.

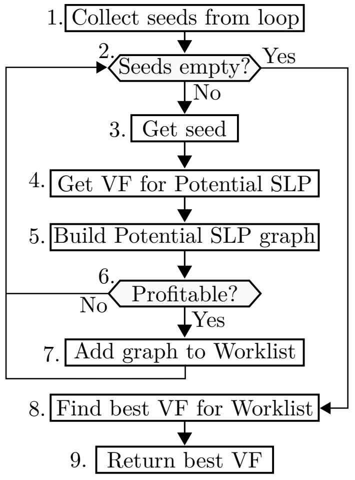

1. Collect seeds from loop
2. Seeds empty?
Yes
No
3. Get seed
4. Get VF for Potential SLP
5. Build Potential SLP graph
6. Profitable?
No
Yes
7. Add graph to Worklist
8. Find best VF for Worklist
9. Return best VF

Figure 4. The high-level algorithm for the VALU heuristic.

Next, given the seed instructions and their corresponding vectorization factors, VALU builds a Potential SLP graph for each one of them. This is done by following the use-def chain, towards the definitions, inserting the instructions to the Potential SLP graph. Once the algorithm encounters instructions that cannot possibly form a vectorizable group by the SLP pass after unrolling, it forms a final non-vectorizable node. The green nodes represent all vectorizable nodes. The red node for `C[D[i]]` in Figure 3c is an example of a non-vectorizable node, due to its indirect memory addressing. This process repeats until we have reached non-vectorizable nodes or *load* instructions. This completes the Potential SLP graph.

While finding isomorphic code is an expensive task for SLP, VALU does not suffer from this same problem since the unrolled copies of the loop will inevitably contain isomorphic code. For this reason, most nodes in the Potential SLP graph are trivially vectorizable, such as those formed by arithmetic, logical, or casting instructions. Memory operations and function calls, on the other hand, require some special treatment. In particular, VALU needs to analyze if the memory instructions can be widened, i.e., whether or not their unrolled copies will form groups with vectorizable access patterns. Section 4.3 describes this analysis in more detail. Function calls are vectorizable if their callees are known vectorizable intrinsics.

Since each Potential SLP graph has its own vectorization factor, we may end up with many profitable Potential SLP graphs in the same loop, each with a different vectorization factor. This introduces a conflict, as the vectorization factor corresponds to the desired unrolling factor of the enclosing loop. We need a way for choosing a single unrolling factor from multiple vectorization factors. The solution is simple: we select the *least common multiple* among the vectorizaton factors, since this is the only way to ensure that all of them will get vectorized in the future by SLP, while also fully utilizing the vector units of the target architecture. Because all vectorization factors are powers of two, this means that, in practice, we can simply select the maximum among them for the unrolling factor.

4.2 Profitability of Potential SLP Graph

As it was mentioned in Section 4.1, a necessary step is deciding whether the Potential SLP graph is profitable, i.e., whether the unrolled scalar code will be considered profitable by the SLP vectorizer. This is done with the help of the compiler's target-specific cost model. The cost of each node is calculated as the difference $VectorCost - ScalarCost$, with negative cost values implying that the vector code performs better than the equivalent scalar code. The $ScalarCost$ of a node in the Potential SLP graph is the cost of its scalar instruction multiplied by the number of copies that will be produced after the loop is unrolled by VF times. The $VectorCost$ is estimated assuming that all unrolled copies of the scalar instruction will be packed into a VF-wide vector. We also account for any additional costs related to inserting/extracting data to/from the potential vector instructions. For example, a vectorizable instruction in our Potential SLP graph may have uses outside the graph. In this case we would have to extract the data from the vector (possibly with the help of some additional instructions) and feed it to its uses.

4.3 Widening Memory Instructions

While arithmetic, logical, and casting instructions are trivially vectorizable by simply widening the data type, memory instructions are more challenging. The best performing vector memory instructions are the ones accessing consecutive memory addresses. Therefore, we consider a memory instruction in the Potential SLP graph as vectorizable, only if its unrolled copies point to consecutive memory addresses. If the addresses are not consecutive but instead follow a strided pattern with small constant strides, these memory instructions may also be vectorized, but this is not currently handled well by the SLP pass, so we are considering them as non-vectorizable. Modern processors do provide support for non-consecutive memory access patterns, but these are usually more costly than their consecutive counterparts, therefore we need to account for this when widening [4].

This memory access analysis is performed symbolically using chains of recurrences [5, 13], implemented by LLVM's scalar evolution framework (SCEV). Chains of recurrences (CR) is a formalism used to represent closed-form functions at regular intervals [52]. In compilers, it is largely used to represent induction variables and memory access patterns, allowing the compiler to reason about loops and memory operations in a systematic way. We are using LLVM's SCEV analysis to perform the memory access analysis, which determines which of the memory instructions in the Potential SLP graph will be vectorized or not.

4.4 Dependence Analysis

The SLP pass relies on dependence analysis to check that the code semantics are not violated by vectorization. LLVM's SLP implements this as part of a scheduling step, which tests whether the groups of instructions to be vectorized, can be moved to a single point in the code, without violating any dependencies. During the construction of the SLP graph, SLP tests whether the instructions are schedulable, and will only form a vectorizable group if they are. If not, the group node is labeled as non-vectorizable.

Before actually unrolling the loop, VALU needs to perform a similar analysis to check whether the unrolled code will have data dependencies that prevent vectorization. Some of the tooling for this analysis is also common to the loop vectorizer, which we adapted for VALU. While the loop vectorizer analyzes the whole loop at the same time, VALU analyzes the data dependency of individual instructions. During the construction of the Potential SLP graph, VALU can analyze if the unrolled replicas of the instructions will be schedulable, preserving all dependencies.

4.5 Partial Vectorization

VALU handles partial vectorization seamlessly. The Potential SLP graph grows until a *load* instruction or non-vectorizable node is found. As long as the cost model estimates that it is profitable to vectorize a Potential SLP graph, it will be considered for vectorization, regardless if the Potential SLP graph is fully vectorizable or not. Figure 3 shows such an example where both VALU and SLP coordinate to partially vectorize a loop that contains indirect memory accesses. As we show in Section 5, this is an important advantage over the loop vectorizer.

4.6 Code Size Concerns

Although VALU will temporarily increase the size of the code and potentially increase the register pressure after unrolling, we rely on the SLP vectorization to bring the code of the unrolled loops close to their initial sizes. However, we cannot always avoid code size increase.

First, partially unrolling a loop may create extra code for maintaining the program's semantics. For example, if the trip count is not divisible by the unrolling factor or the trip count is not statically known, we need to create a cloned

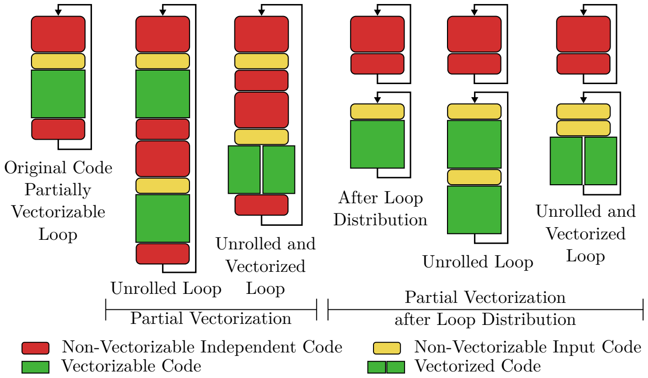


Figure 5. An illustrative example of a partially vectorizable loop that shows how the non-vectorizable part of the loop is also replicated. If some of this code is independent from the vectorizable part of the loop, then loop distribution could reduce unnecessary replication.

loop to perform the remainder iterations after the unrolled loop [44].

Significant code size increase can also result from partially vectorizable loops. When a fully vectorizable loop is unrolled, all unrolled copies will be grouped together in a vector form, canceling out the effects in code size. However, when an unrolled loop is only partially vectorizable, all copies of the non-vectorizable code will remain scalar. This is illustrated in Figure 5. After the loop gets unrolled and vectorized, the resulting loop will still contain multiple copies of the non-vectorizable code.

There is a way to mitigate this code increase if part of the non-vectorizable code is completely independent of the vectorizable code in the loop. We can perform loop distribution and only unroll the loop that contains the vectorizable code, as shown in Figure 5. This loop may still contain non-vectorizable code that interacts directly with the vectorizable part of the loop, but the impact on code size increase will be smaller. When this is not possible, we provide a threshold that specifies the minimum proportion of vectorizable code in a loop to consider unrolling it. Loops with little vectorizable code are ignored.

4.7 Forwarding Seeds to SLP

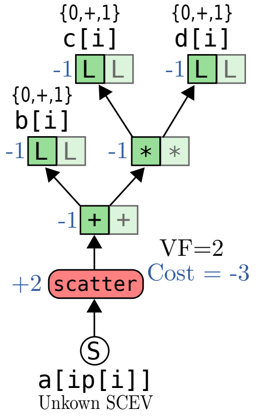
Straight-line code vectorization is a graph isomorphism problem and as such, an optimal solution has exponential time complexity. SLP Vectorizers [43] in production compilers are designed around heuristic-based algorithms that limit the exploration to instructions that have a good chance of success. They collect seed instructions (e.g., stores to consecutive memory addresses) and perform a localized exploration on the use-def chains rooted at these seeds. The collection of seed instructions, however, is both computationally expensive and is itself guided by heuristics whenever multiple grouping alternatives are available. This can lead to missed vectorization opportunities if the seed collection does not form a seed group with the instructions generated by unrolling. VALU can help by forwarding the seed instructions that drive its unrolling decision to SLP, effectively bypassing SLP's seed collection for these instructions, and increasing the probability of success.

VALU collects the seeds during its Potential SLP graph formation. The Potential SLP graph is built from a single seed instruction. The unrolled copies of this single seed instruction will then become the seed instructions to form the first group node of an SLP graph. Instead of expecting SLP's seed collection to find these same instructions and group them correctly, VALU can assist the SLP vectorizer. To achieve that, VALU keeps track of the unrolled copies of the profitable seed instructions while performing the unrolling and shares them with the SLP vectorizer. This guarantees that SLP will be applied on unrolled copies of instructions that are trivially isomorphic and profitable for vectorization. This is preferred to relying on SLP's greedy seed collection, which may miss these vectorization opportunities in the unrolled code.

There are two cases where seed forwarding is extra helpful: non-vectorizable stores and reduction computations.

```
for (int i = 0; i < LEN; i++) {
  a[ip[i]] = b[i] + c[i] * d[i];
}
```

(a) Kernel S491 with an example of a non-vectorizable store instruction that leads to a partially vectorizable SLP graph.



(b) Potential SLP graph.

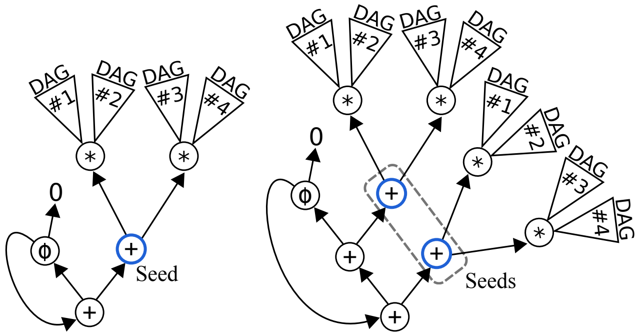
Figure 6. VALU partial vectorization with non-vectorizable store instructions as seeds.

4.7.1 Non-Vectorizable Stores

Figure 6 shows a loop with a store instruction which is non-vectorizable, due to its indirect addressing, but its value operand is part of a profitable SLP graph for vectorization. Since unrolling generates copies of the loop body, VALU is aware that although the store is non-vectorizable, it is possible that the unrolled copies of its value operand will result

```
int sum = 0;
for (int i = 0; i<SIZE; i++) {
 sum += ((DAG#1)*(DAG#2)) + ((DAG#3)*(DAG#4))
}
```

(a) Reduction loop before unrolling. The DAGs represent subexpressions that may be different from one another.



(b) Use-def graph with reduction before unrolling.

(c) Use-def graph with reduction after unrolling by a factor of 2.

Figure 7. Horizontal reduction before and after unrolling. We highlight the seeds for isomorphic graphs.

in isomorphic use-def graphs that are profitable for SLP vectorization. For this reason, if the store is non-vectorizable, VALU builds the Potential SLP graph starting from its value operand, as shown in Figure 6b. If this Potential SLP graph is profitable for vectorization, VALU forwards these as seed instructions for SLP.

Without seed forwarding from VALU, SLP performs seed collection in the already unrolled loop. LLVM's SLP does not track these non-vectorizable store instructions, for complexity reasons. As such it fails to collect its predecessors as seeds and will not vectorize it. The loop vectorizer cannot handle this loop either as it requires partial vectorization.

4.7.2 Reduction Computations

VALU seed forwarding can also improve SLP vectorization of reductions. Figure 7b shows the use-def graph for the reduction from the loop shown in Figure 7a.

Currently, the SLP seed collection is performed by following the use-def chains, starting from the ϕ-node, grouping the first set of nodes that differ from the reduction operator. SLP considers all instructions with the same opcode of the reduction operator as part of the reduction computation. In the example shown in Figure 7, the SLP vectorizer collects all multiplication instructions as seeds and proceeds to form the SLP graph. A major problem arises with loop unrolling (Figure 7c), which generates copies of the loop body and makes it harder to identify the reduction and its immediate operands. SLP may greedily select a group of additions as seeds, which may be a non-profitable group. However, it is trivial for VALU to identify the seeds highlighted in Figure 7c and forward them to the SLP vectorizer.

5 Experimental Results

5.1 Experimental Setup

Our evaluation platform is a Linux 4.4.27, glibc-2.22 based system with an Intel®Core i7-4770 CPU and 16 GiB of RAM. We implemented VALU as a standalone pass in LLVM 8 and was placed just before the SLP vectorizer in the compilation pipeline. We compiled all benchmarks using `clang` with the following flags: `-O3 -ffast-math -march=native -mtune=native -mllvm -slp-vectorize-hor`. These options enable the default loop unroller (DU) as well as both SLP and the Loop Vectorization (LV).

We evaluate our approach on three benchmark suites[2]: TSVC [6], FreeBench [18], and MiBench [15]. First, we provide a detailed analysis on several of the TSVC kernels, which were specifically designed for evaluating vectorizing compilers. Then, we provide performance results on both FreeBench and MiBench, which include full benchmark programs from a wide range of application domains.

SLP, being a straight-line-code vectorizer, is not expected to find many opportunities for vectorization in the TSVC kernels, which is exactly what makes it a great suite for evaluating the effectiveness of VALU. Since the TSVC suite contains a large number of kernels (151), we only show the kernels with a performance difference of at least 2% or more compared to the baseline. In total, 52 kernels are hidden from the plots. Regardless, geometric means and averages refer to all 151 TSVC kernels. For our performance results we ran each workload 25 times and we show the arithmetic average of the speedup across all runs, as well as the 95% confidence interval of the speedup as a min-max bar.

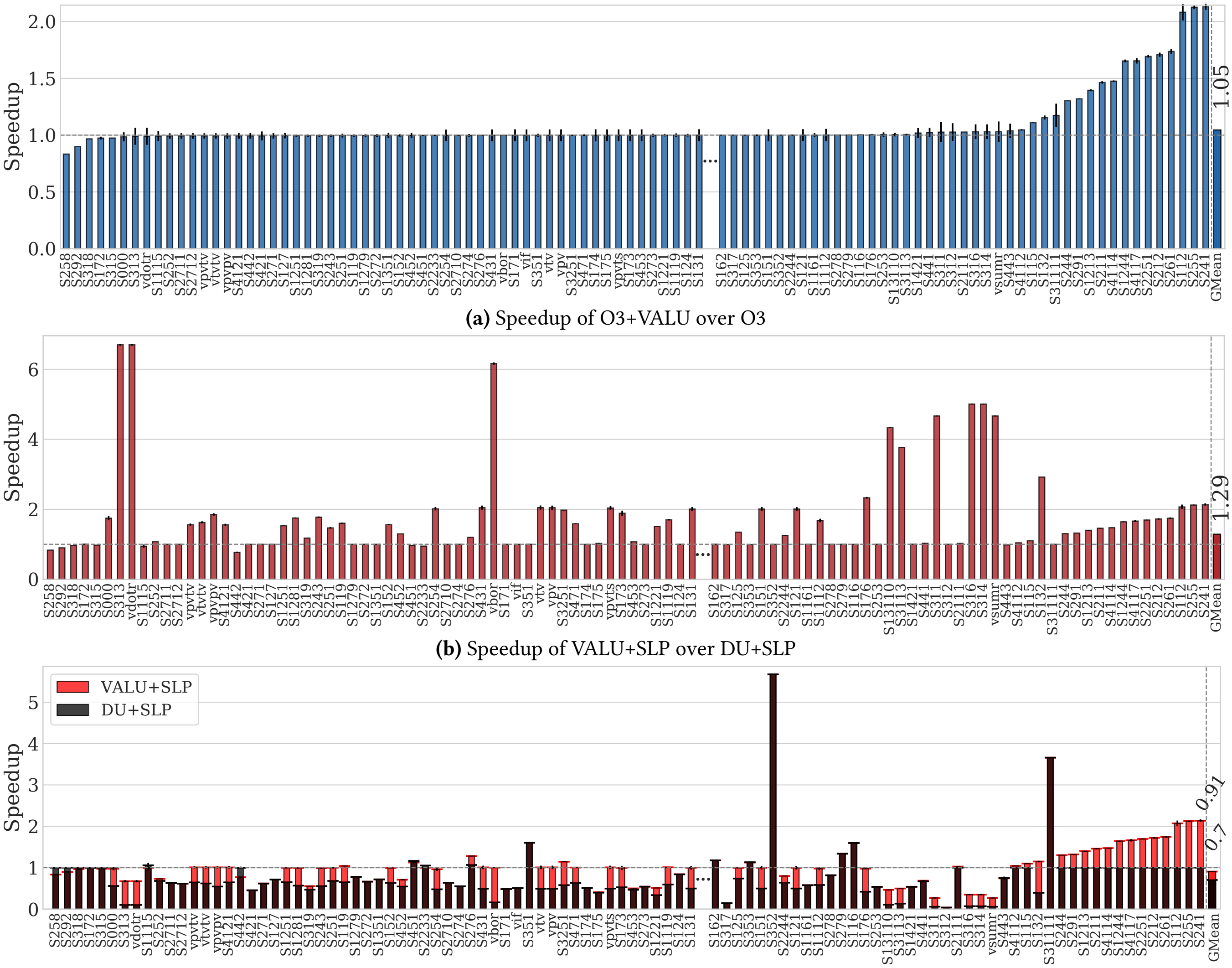
5.2 Overall Performance

The performance speedup of enabling VALU over `-O3` is shown in Figure 8a. VALU significantly improves the LLVM baseline with a speedup of up to 2×, and a geometric mean of 1.05× (5% improvement) across the whole benchmark suite. This is a promising result, given the heavily optimized baseline and that for most kernels there is little room for improvement when applying SLP.

As we dicuss later in Section 5.3, many of the significant speedups shown in Figure 8a are due to partial vectorization enabled by VALU, such as the kernel S255. However, the few regressions observed, more specifically the kernels S258 and S292, also represent two loops that get unrolled by VALU and later partially vectorized by SLP. Both VALU and the SLP vectorizer rely on the compiler's built-in cost model when checking for profitability, which can cause performance regressions when the cost model contains inaccuracies. The rest of the results show the expected behavior: better costs lead to better performance.

Figure 8b isolates the effect of more intelligent unrolling on SLP vectorization. It shows the speedup of VALU over

[2] These benchmarks can also be found in the LLVM benchmark suite.



(a) Speedup of O3+VALU over O3

(b) Speedup of VALU+SLP over DU+SLP

(c) Speedup of VALU+SLP and DU+SLP over the loop vectorizer (LV)

Figure 8. Evaluation of the effect of VALU when applied on top of the baseline O3 or on top of the standalone SLP. To simplify presentation, we only show kernels that have a speedup or a slowdown of more than 2% in any of the plots. Geometric means include all kernels, whether shown or not.

LLVM's default loop unroller with SLP vectorization enabled but loop vectorization disabled. In other words, the baseline is using the additional `-fno-vectorize` and `-fslp-vectorize` flags, and we show the speedup due to enabling VALU over this setting. Since VALU is well coordinated with the requirements of SLP, it is expected that more code will get vectorized compared to the default loop unroller. This figure supports our argument that the default loop unrolling heuristics are inappropriate for preparing code for the SLP vectorizer. VALU uncovers vectorization opportunities that result in speedups of up to 6× compared to the default loop unroller, with a geometric mean of 1.29× (29% improvement) across all 151 kernels in the TSVC benchmarks.

Figure 8c compares SLP against loop vectorization. The baseline is `-O3` with loop vectorization but without SLP (`-fno-slp-vectorize`). The figure shows the speedup over this baseline with the loop vectorizer disabled, SLP enabled, and either the default loop unroller or VALU enabled. The figure highlights two key points that were motivated in Section 3: (*i*) VALU enables SLP to handle loops where the loop vectorizer fails, and (*ii*) VALU helps to close the performance gap between SLP and the loop vectorizer. A good coordination between the loop unroller and the SLP vectorizer is essential for SLP to reach, or even exceed, the performance of the loop vectorizer.

Although SLP combined with VALU can cover many of the same loops covered by the loop vectorizer, there are still multiple cases where the loop vectorizer generates faster code than SLP even when combined with the VALU unroller. In most of them, it is due to some missing features in LLVM's

```c
for (int i = 0; i < LEN-1; i++) {
 a[i] = b[i] * c[i] * d[i];
 b[i] = a[i] * a[i+1] * d[i];
}
```

Figure 9. Kernel S241 with complex data dependences that require instruction reordering before vectorization. VALU+SLP vectorized version results in gains higher than 2× over -O3.

specific implementation of SLP and a better SLP implementation would be able to vectorize the code. This means that the best configuration should still include both vectorizers, in addition to our new VALU loop unroller that provides significant speedup on top of -O3, as shown in Figure 8a. In the following sections, we discuss key strengths of VALU and also how to improve the LLVM's SLP implementation in more details. Finally, we report the compilation overhead of our approach.

5.3 Overall Analysis of the Performance Results

As expected, the loop vectorizer performs very well on this loop-only benchmark suite. However, there are two classes of loops where VALU+SLP outperforms the loop vectorizer: (1) loops that contain *loop-independent dependences*; and (2) loops that can only be partially vectorized. Because SLP operates on groups of use-def graphs separately, it is able to handle loop-independent dependences out of the box, leaving the problem of placing the vectorized instructions to the scheduler (see Section 4.4). Similarly, because SLP grows its graph until the point where it is no longer vectorizable, partial vectorization is intrinsic to it. As long as the SLP graph is considered profitable, it will be vectorized.

We can also assign the loops where VALU+SLP misses performance opportunities in two classes: (1) reductions computations; and (2) loops with control flows that require predication. Overall, LLVM's loop vectorizer supports more idioms than its SLP implementation, resulting in missed opportunities for VALU+SLP. We discuss in detail all these cases in the subsequent subsection.

5.3.1 Loop-Independent Dependences

Loop-independent dependences are between different instructions within the same iteration of a loop. This adds complexity to the code and frequently inhibits vectorization by the LLVM's loop vectorizer, especially as this requires instruction reordering to allow vectorization. The two main examples are kernels S241 and S112, where VALU+SLP gets more than 2× speedup on top of -O3. Other significant examples are the kernels S1213, S1244, S211, and S212.

For the kernel S241 shown in Figure 9, while the loop vectorizer fails, VALU+SLP can successfully vectorize this loop. SLP only needs to make sure that it loads a[i+1:i+8] before updating a[i:i+7]. Because each statement in this loop is handled separately, by analyzing their use-def chains, after VALU unrolls it, SLP is able to schedule the vectorizable instructions and preserve data dependencies. The vectorized code results in a speedup higher than 2× over -O3.

```c
for (int i = n1-1; i < LEN; i++) {
 k = ip[i];
 a[i] = b[i] + c[LEN-k+1-2] * d[i];
 k += 5;
}
```

Figure 10. Kernel S4114 with indirect addressing. VALU+SLP version achieves about 1.5× speedup over -O3.

5.3.2 Partially Vectorizable Loops

One benefit of VALU+SLP over LLVM's LV is that it can partially vectorize loops containing non-vectorizable code. The loop in Figure 10, taken from the kernel S4114, is such a case. It contains an indirect memory access c[LEN-k+1-2] that cannot be vectorized. While the loop vectorizer bails out completely, VALU+SLP vectorizes it partially, improving the performance of this loop by about 50%.

Specifically, if VALU unrolls the loop, SLP can partially vectorize the code and leave the indirect memory access. This means that the scalar loads c[LEN-k+1-2] must be inserted into a vector, but this overhead is taken into account by our Potential SLP analysis and is found to be profitable.

Other kernels that also include indirect addressing are S4112 and S4117, which also result in significant speedups. In addition to indirect memory accesses, there are many other loops that are partially vectorized by VALU+SLP that LV is unable to handle, such as S2251, S244, S255, and S291.

5.3.3 Seed Forwarding

VALU's seed forwarding mechanism is an effective way of overcoming major limitations in existing vectorizers. Figure 11 shows a loop that is poorly vectorized by SLP without the assistance of VALU's seed forwarding, as the computation being stored in adjacent addresses is not fully isomorphic. However, VALU groups the interleaved stores that are fully isomorphic, resulting in a vectorized code with a performance equivalent to that produced by the loop vectorizer. Benchmarks S1111 and S491 (Figure 6) are loops that neither one of the vectorizers were able to handle because of the access pattern used by the store instruction. However, VALU also collects store instructions that cannot be widened, using its value operand as the seed for a Potential SLP graph.

5.3.4 Reduction Computations

The loop vectorizer in LLVM is able to generate efficient code for reductions. This accounts for all exceptionally well performing cases of LV. Although VALU is able to identify reductions, especially because max- or min-reductions are lowered into select-based reductions, LLVM's SLP implementation has a limited support for reductions. The two most serious limitations are that it cannot handle product-based

```
    j = -1;
    for (int i = 0; i < LEN/2; i++) {
      j++;
      a[j] = b[i] + c[i] * d[i];
      j++;
      a[j] = b[i] + d[i] * e[i];
    }
```

Figure 11. Kernel S127. Loop shows an induction variable with multiple increments. Example where forwarding seeds makes life easier for the SLP vectorizer.

```
for (int i = 0; i < LEN; i++) {
  if (e[i] >= t) {
    a[i] += c[i] * d[i];
    b[i] += c[i] * c[i];
  }
}
```

Figure 12. S272. This loop has a conditional branch. LLVM's loop vectorizer is able to vectorize this loop using predication, which is not yet supported by the SLP implementation.

reductions and it reduces the vector lanes *inside the loop* instead of outside the loop. The former makes it impossible to vectorize cases that the loop vectorizer does, the latter reduces the benefits of vectorization.

The list of kernels with reduction in Figure 8c includes: S13110, S31111, S311, S312, S313, S314, S316, S317, S319, S3113, S352, vdotr, and vsumr. Because the kernels S31111 and S352 are a reduction where the inner loop has already been unrolled, the loop vectorizer is unable to handle it. For the other kernels with reduction, however, the loop vectorizer is able to generate very efficient code.

5.3.5 Predicated Vectorization

The loop vectorizer is also able to effectively handle loops that contain conditional branches, such as the loop in Figure 12, taken from the kernel S272. In these cases, it generates a vectorized code that uses masks to predicate the execution for particular vector lanes.

Similar cases of predication, with varied levels of complexity, can be found in the kernels: S1161, S124, S1279, S253, S271, S2710, S2711, S2712, S272, S273, S274, S441, S443, and vif. For all of them, we are limited by the implementation of SLP in LLVM which does not support predicted SLP vectorization, despite proposed techniques to achieve this [45]. On such cases, our unrolling technique has any effect, so we only consider single-block loops in our heuristic.

5.4 Compilation Time

We measured the wall clock time for compiling the full TSVC benchmark suite using O3+VALU and normalizing it to O3. Enabling VALU leads to a modest overall compilation overhead of 16% over O3, considering the whole compilation pipeline. Most of this overhead is due to the fact that after loop unrolling, subsequent optimizations, including the SLP vectorizer, and the backend will have more code to process.

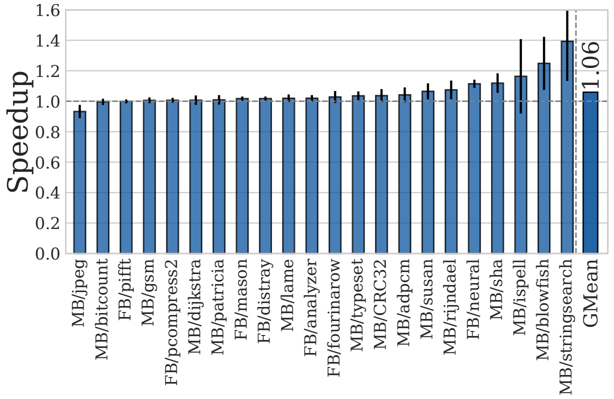


Figure 13. Speedup of O3+VALU over O3 on full benchmarks.

Interestingly, if we compare VALU+SLP with the loop vectorizer (LV), VALU+SLP results in about 8% faster compilation. This shows that the compilation overhead of VALU+SLP is within acceptable bounds. The difference in compilation time comes from different sources, which includes the time spent during the vectorization itself, but also because loop unrolling can still be applied after the loop vectorizer.

5.5 Performance on Full Benchmarks

The kind of code accelerated by VALU is not found only in benchmarks suites designed to test vectorizers. We tested VALU on the benchmarks of the FreeBench and MiBench suites, on top of the baseline -O3 which already includes both vectorizers and the default loop unrolling. Shown in Figure 13, VALU achieves a significant speedup on five of these benchmarks, with stringsearch getting 45% faster, and an overall geometric mean speedup of 1.06.

6 Related Work

6.1 Loop Unrolling

Loop unrolling is a well-studied code transformation technique, implemented in most compilers. There is a wide range of studies on loop unrolling [9, 30]. Traditionally, this was applied only to FOR-loops at the source level [1]. Later, more general techniques have been proposed to perform loop unrolling [16, 44], including nested and remainder loops.

Unroll-and-jam is a loop unrolling technique for outer loops, unrelated to vectorization. With unroll-and-jam, the compiler unrolls outer loops and then fuses the unrolled copies of the inner loops [7, 8, 34]. Similarly, Ferrer et al. [14] shows how to unroll loops that already contain OpenMP task parallelism, fusing the tasks after unrolling to reduce unnecessary multi-threading overheads.

VALU is the first loop-unrolling technique, to the best of our knowledge, that provides a strong coupling between loop unrolling and SLP vectorization. Unlike prior unrolling work that aims at balancing code size increase with improving the applicability of generic optimizations, VALU is able to identify loops that are valid and profitable to be vectorized.

There has also been a significant amount of work on iterative optimization or other approaches for tuning the unrolling factor [20, 21, 24, 46]. However, even if these approaches manage to find the best unrolling factor to uncover SLP vectorization, which is usually infeasible on a per loop basis, they are still insufficient to vectorize those loops that require VALU's seed forwarding. As described in Section 4.7, there are cases where SLP can be unable to properly identify the seed instructions in order to vectorize the unrolled loop.

6.2 Loop and Function Auto-Vectorization

Auto-vectorization techniques have traditionally focused on vectorizing loops [49]. The basic implementation conceptually strip-mines the loop by the vector factor and widens each scalar instruction in the body to work on multiple data elements. The effectiveness of loop vectorizing compilers has been studied by Maleki et al. [26]. Many fundamental problems of loop vectorization have been addressed by early work on the Parallel Fortran Converter [2, 3, 11, 22, 48]. Since then, numerous improvements to the basic algorithm have been proposed in the literature and production compilers [4, 12, 31, 32, 42]. For example, Stock et al. [47] uses machine learning to train a profitability model for the loop vectorizer.

Whole function vectorization has been proposed by Karrenber et al. [19, 41]. This is particularly important for mapping programming models like OpenCL onto vector units. A different approach is presented by Masten et al. [27] which discusses how function/kernel vectorization could be presented as a loop-vectorization problem. Finally, Moll et al. [29] present a novel control-flow linearization algorithm, for use in function/kernel vectorizers.

6.3 SLP Auto-Vectorization

A complementary technique to the loop vectorizer has been introduced by Larsen and Amarasinghe [23], the SLP vectorizer, which focuses on straight-line code. Since its original work, several improvements have been proposed for the straight-line-code (SLP-style) vectorization [17, 25, 28, 33, 45].

The bottom-up SLP algorithm has recently been improved in several ways. [37] introduces padding the code with redundant instructions to generate isomorphism and improve vectorization. In [36] the SLP region is pruned to scalarize groups of instructions that harm the vectorization cost, while in [35] a larger unified SLP region is used, to overcome limitations caused by the region formation. In [50] vectorization is enabled for SIMD widths that are not supported by the target hardware. Finally, extensions to SLP that focus on commutative operations are presented in [38, 39].

Combining loop-vectorization with SLP was proposed in loop-aware SLP [43] and implemented in GCC. This work combines SLP-style parallelism with the loop vectorizer, which allows it to vectorize both across iterations and within a single iteration. Zhou et al. [51] improve this technique by extending the exploration performed by the algorithm, improving the effectiveness of the mixed inter and intra-loop vectorization. Both approaches rely on SLP-style parallelism that must already be exposed in the loop body, which means that VALU would be complementary to them. This is different from our work.

7 Conclusion

This paper presented Vectorization-Aware Loop Unrolling (VALU), a novel compiler heuristic for identifying loop unrolling opportunities to enable the straight-line-code vectorization. VALU does so by identifying if loop unrolling will be profitable for the SLP vectorizer and what loop unroll factor can maximize the utilization of the target architecture's vector units. VALU determines the unroll factor by employing *Potential SLP*, a novel vectorization and profitability analysis on the original rolled loop as if it was unrolled. We implemented VALU in LLVM. and evaluated it on the TSVC vectorization testing suite. Our experimental results show a great SLP vectorization improvement compared to the LLVM's default loop unrolling heuristic, and very significant performance improvements over O3.

Acknowledgment

This work has been supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/L01503X/1 (CDT in Pervasive Parallelism), EP/P003915/1 (SUMMER), and EP/M01567X/1 (SANDeRs). This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

References

[1] FE Allen and J Cocke. 1971. *A catalogue of optimizing transformations.* IBM Research Center, Thomas J. Watson.

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

[3] John Randy Allen and Ken Kennedy. 1987. Automatic Translation of Fortran Programs to Vector Form. *Tranactions on Programming Languages and Systems (TOPLAS)* 9, 4 (1987).

[4] Andrew Anderson, Avinash Malik, and David Gregg. 2015. Automatic Vectorization of Interleaved Data Revisited. *ACM Trans. Archit. Code Optim.* 12, 4 (Dec. 2015), 50:1–50:25.

[5] Diego Andrade, Manuel Arenaz, Basilio B. Fraguela, Juan Touriño, and Ramón Doallo. 2007. Automated and accurate cache behavior analysis for codes with irregular access patterns. *Concurrency and Computation: Practice and Experience* 19, 18 (2007), 2407–2423.

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

[7] Steve Carr and Yiping Guan. 1997. Unroll-and-jam Using Uniformly Generated Sets. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 30).* IEEE Computer Society, Washington, DC, USA, 349–357.

[8] Steve Carr and Ken Kennedy. 1994. Improving the Ratio of Memory Operations to Floating-point Operations in Loops. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1768–1810.

[9] Keith Cooper and Linda Torczon. 2003. *Engineering a Compiler: International Student Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[10] Jack W. Davidson and Sanjay Jinturkar. 1996. Aggressive loop unrolling in a retargetable, optimizing compiler. In *Compiler Construction*, Tibor Gyimóthy (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 59–73.

[11] James Davies, Christopher Huson, Thomas Macke, Bruce Leasure, and Michael Wolfe. 1986. 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.*

[12] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).*

[13] Robert van Engelen. 2001. Efficient Symbolic Analysis for Optimizing Compilers. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01).* Springer-Verlag, London, UK, UK, 118–132.

[14] Roger Ferrer, Alejandro Duran, Xavier Martorell, and Eduard Ayguadé. 2010. Unrolling Loops Containing Task Parallelism. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing (LCPC'09).* Springer-Verlag, Berlin, Heidelberg, 416–423.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization.*

[16] J. C. Huang and T. Leng. 1999. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122).* 244–248.

[17] Joonmoo Huh and James Tuck. 2017. Improving the Effectiveness of Searching for Isomorphic Chains in Superword Level Parallelism. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17).* ACM, New York, NY, USA, 718–729.

[18] J. Hwang, S. Zeng, F. y. Wu, and T. Wood. 2013. A component-based performance comparison of four hypervisors. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013).* 269–276.

[19] Ralf Karrenberg and Sebastian Hack. 2011. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* IEEE Computer Society, 141–150.

[20] Toru Kisuki, Peter M. W. Knijnenburg, Mike F. P. O'Boyle, François Bodin, and Harry A. G. Wijshoff. 1999. A feasibility study in iterative compilation. In *High Performance Computing*, Constantine Polychronopoulos, Kazuki Joe Akira Fukuda, and Shinji Tomita (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–132.

[21] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. 2002. *Iterative Compilation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 171–187.

[22] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the Symposium on Principles of Programming Languages.*

[23] S. Larsen and S. Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).*

[24] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2009. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09).* IEEE Computer Society, Washington, DC, USA, 81–91.

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

[26] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT).*

[27] Matt Masten, Evgeniy Tyurin, Konstantina Mitropoulou, Hideki Saito, and Eric Garcia. 2018. Function/Kernel Vectorization via Loop Vectorizer. *Proceedings of the 5th Workshop on The LLVM Compiler Infrsatructure in HPC (LLV-HPC)* (2018).

[28] Charith Mendis and Saman Amarasinghe. 2018. GoSLP: Globally Optimized Superword Level Parallelism Framework. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 28.

[29] Simon Moll and Sebastian Hack. 2018. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 543–556.

[30] Steven S. Muchnick. 1997. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, San Fransisco, California, USA.

[31] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).* ACM, 132–143.

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

[33] Y. Park, S. Seo, H. Park, H.K. Cho, and S. Mahlke. 2012. 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).*

[34] D. Petkov, R. Harr, and S. Amarasinghe. 2002. Efficient pipelining of nested loops: unroll-and-squash. In *Proceedings 16th International Parallel and Distributed Processing Symposium.* 6 pp–.

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

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

[37] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO).*

[38] Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. 2019. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019).* IEEE Press, Piscataway, NJ, USA, 206–216.

[39] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. Look-ahead SLP: Auto-vectorization in the Presence of Commutative Operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018).* ACM, New York, NY, USA, 163–174.

[40] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. VW-SLP: Auto-vectorization with Adaptive Vector Width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18).* ACM, New York, NY, USA, 12:1–12:15.

[41] R. Karrenberg and S. Hack. 2012. Improving performance of OpenCL on CPUs. In *International Conference on Compiler Construction.* Springer, 1–20.

[42] Gang Ren, Peng Wu, and David Padua. 2006. Optimizing Data Permutations for SIMD Devices. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).*

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

[44] Vivek Sarkar. 2000. Optimized Unrolling of Nested Loops. In *Proceedings of the 14th International Conference on Supercomputing (ICS '00)*. ACM, New York, NY, USA, 153–166.

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

[46] Mark Stephenson and Saman Amarasinghe. 2005. Predicting Unroll Factors Using Supervised Classification. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, Washington, DC, USA, 123–134.

[47] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. 2012. Using Machine Learning to Improve Automatic Vectorization. *ACM Trans. Archit. Code Optim.* 8, 4, Article 50 (Jan. 2012), 23 pages.

[48] Michael Wolfe. 1988. Vector optimization vs. vectorization. In *Supercomputing*. Springer.

[49] Michael Joseph Wolfe. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley.

[50] Hao Zhou and Jingling Xue. 2016. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization (TACO)* (2016).

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

[52] Eugene V. Zima. 1995. Simplification and Optimization Transformations of Chains of Recurrences. In *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*. ACM, New York, NY, USA, 42–50.