Control Flow Vectorization for ARM NEON

Angela Pohl, Nicolás Morini, Biagio Cosenza and Ben Juurlink
Technische Universität Berlin
Berlin, Germany
{angela.pohl, cosenza, b.juurlink}@tu-berlin.de

Abstract

Single Instruction Multiple Data (SIMD) extensions in processors enable in-core parallelism for operations on vectors of data. From the compiler perspective, SIMD instructions require automatic techniques to determine how and when it is possible to express computations in terms of vector operations. When this is not possible automatically, a user may still write code in a manner that allows the compiler to deduce that vectorization is possible, or by explicitly define how to vectorize by using intrinsics.

This work analyzes the challenge of generating efficient vector instructions by benchmarking 151 loop patterns with three compilers on two SIMD instruction sets. Comparing the vectorization rates for the AVX2 and NEON instruction sets, we observed that the presence of control flow poses a major problem for the vectorization on NEON. We consequently propose a set of solutions to generate efficient vector instructions in the presence of control flow. In particular, we show how to overcome the lack of masked load and store instruction with different code generation strategies. Results show that we enable vectorization of conditional read operations with a minimal overhead, while our technique of *atomic select stores* achieves a speedup of more than 2x over state of the art for large vectorization factors.

CCS Concepts **•Computer systems organization → Single instruction, multiple data;** *Embedded systems;* **•Software and its engineering → Compilers;**

ACM Reference format:
Angela Pohl, Nicolás Morini, Biagio Cosenza and Ben Juurlink
. 2018. Control Flow Vectorization for ARM NEON. In *Proceedings of International Workshop on Software and Compilers for Embedded Systems, Germany, St Goar (SCOPES'18),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Code vectorization is an optimization technique to exploit data level parallelism (DLP). Starting in the 1970s, vector processors grouped together data independent instructions and applied vector operations instead of scalar ones. Depending on the Vectorization Factor (VF), i.e. the number of data elements that can be merged into one vector, vectorization can reach high speedups, especially on codes exposing DLP in loops.

However, the task of producing efficient vectorized code is challenging. Manual approaches, where the programmer directly indicates which vectorial instruction to use, require huge efforts and produce results that are not portable when targeting different Instruction Set Architectures (ISAs). That is why auto-vectorizers have been added to compilers to perform this task automatically. As of today, a compiler can contain multiple vectorization passes, targeting loops and straight line code. During compilation, the vectorizer has to find a valid code transformation which can be mapped to the underlying ISA. In addition, the transformation has to be deemed beneficial, i.e. the added overhead should not efface the performance gain.

Auto-vectorization has made tremendous improvements in the last decades, with different compiler techniques addressing different forms of parallelism (e.g. Superword-Level Parallelism for straight-line code vectorization) as well as increasingly complex code patterns. However, the success of vectorization does not purely depend on the compiler and the code to be vectorized. The target ISA plays a key role by providing instructions that enable efficient vectorization as well.

This paper starts with a quantitative and qualitative analysis of state-of-the-art techniques provided by production compilers (GCC, ICC, and LLVM) on two different SIMD ISAs, i.e. Intel's AVX2 and ARM's NEON. Investigating 151 loops with a broad range of code patterns, our numbers show that the compilers do not perform adequately on the processor supporting NEON, specifically for loops that contain control flow. It is a genuine example of a code pattern whose vectorization can be greatly enhanced by the availability of specific instructions, e.g., masked load and store. Unfortunately, these instructions are missing in the NEON instruction set, therefore limiting the vectorization in existing production compilers. Consequently, we propose two techniques to enable the automatic vectorization of conditional load and store operations, increasing the vectorization rates for NEON platforms.

The contributions of this paper are:

- a detailed quantitative and qualitative analysis of the vectorization rate and quality of GCC, ICC, and LLVM on a test benchmark of 151 loops, targeting AVX2 and NEON platforms
- an automatic approach to vectorize conditional load operations where compilers currently fail due to the inability to assume the safety of memory accesses
- an automatic approach to vectorize conditional store operations, resulting in an improved code performance compared to the state-of-the-art scalar predicated store currently employed in compilers.

Both vectorization techniques can be applied to ARM NEON platforms, as well as other ISAs that do not support masked load/store instructions natively.

The paper is organized as follows: in Section 2 we discuss current options to handle control flow for vectorization. In Section 3, we show the results of our analysis of the most popular C/C++ compilers' auto-vectorizers. Afterwards, we describe the implementation of conditional load/store operations for the state of the

Table 1. Overview of compiler flags and versions

	gcc 7.2.0	**LLVM 5.0.0**	**icc 18.0.1**
Vectorized Setup	`-std=(c11|c++11) -O3 -ffast-math -march=native`		`-std=(c11|c++11) -Ofast (-xavx|-xavx2)`
Added for Scalar Setup	`-fno-tree-vectorize -fno-tree-slp-vectorize`	`-fno-vectorize -fno-slp-vectorize`	`-no-vec`
Added for Reports	`-fopt-info-vec-all=report.lst`	`-Rpass=loop-vectorize -mllvm -debug-only=loop-vectorize,SLP`	`-qopt-report=1 -qopt-report-phase=vec`

art, as well as our new proposed solutions in Section 4. In Section 5, we show the performance improvements achieved by our proposed approaches, and the paper is concluded in Section 6.

2 Related Work

Manually programming SIMD units with either intrinsics or in assembly language is an error prone and time consuming activity, whose output highly depends on the target architecture. A more productive and portable alternative is to rely on compiler-based automatic vectorization (*auto-vectorization*), which tries to replace scalar instructions with vectorial ones. Auto-vectorization mainly relies on two approaches: Loop-Level Vectorization (LLV) [3] and Superword- Level Parallelization (SLP) [13].

LLV, used since the advent of vector processors [22], is the preferred technique in today's compilers due to the potentially low vector utilization in SLP [18]. In a related study from 2011, Maleki et al. [14] analyzed the state of the art of the gcc, icc, and XLC vectorizers with three different benchmarks. A second study by Mitra et al. from 2013 [15] performed a comparison of performance gains on platforms with SSE and NEON SIMD extensions. However, the authors did not utilize auto-vectorizers, but manually vectorized the codes.

An important aspect of LLV is the generation of efficient vector instructions in the presence of control flow. Control flow may diverge because a condition might be true for some scalar instances and false for others, therefore requiring specific solutions such as masking. Allen et al. [2] first worked on solving control flow vectorization by converting control flow dependences to data dependences. They developed a translator, the Parallel Fortran Converter, which implemented an *if-conversion* phase that attempts to eliminate all goto statements in the program. Erosa and Hendren [7] introduced an algorithm that eliminates each goto in control flows by first applying a sequence of goto-movement transformations followed by the appropriate goto elimination. Karrenberg [11] describes a multiphase algorithm for OpenCL codes where *mask*s are computed for every edge of the control flow graph, storing information about the flow of control in the function; subsequently, select instructions that discard results of inactive instances are introduced where necessary. Parts of the control flow graph where the instances may take different paths are linearized, i.e., all branches except for loop back edges are removed and code of originally disjoint paths is merged into one path.

If-conversions have been used to in the context of improving instruction level parallelism with (non-vectorial) predicated instructions as well. August et al. [5] investigated how compilers should appropriately balance control flow and predication to achieve efficient execution, and studied how this is tightly coupled with scheduling decisions and processor characteristics.

Control flow issues also arise from irregular codes, which have been handled with specific vectorization approaches; examples are trees [10], irregular data structures [19], or irregular strides [12]. Attempts to support control-flow have been investigated also for SLP vectorization [20].

An alternative approach is the design of better programming models that addresses the code generation of such patterns (an overview of programming models for vectorization has been conducted by Pohl et al. [17]). A particular interesting model is provided by the ispc compiler [16], which implements special features for control flow: masking is handled with an explicit execution mask and keywords (e.g., `unmasked`); special support for coherent control flow statements (e.g., the `cif` keyword); an efficient way to handle data races within a gang (a group of program instances running together): any side effect from one program instance is visible to other program instances in the gang after the next sequence point in the program (for vectorization, this semantic is more efficient than OpenCL's `barrier()`).

Efficient vectorization also depends on the ISA design. Intel vectorial instructions [8], for instance, provide masked load and store [9], which drastically simplify the generation of vector code in control flow. The Scalable Vector Extension (SVE) [21], a novel instruction set designed for HPC workloads on ARMv8-A, emphasizes the importance of predication for vectorization with dedicated predication registers to efficiently handle active and inactive lanes of non-fixed-length vectors. Unfortunately, the current NEON instruction set [4] does not have similar features, and is lacking the masked load and store operations present in AVX2. This work starts from the observation that this is a major problem for vectorization and requires different code generation strategies in order to produce efficient vectorized code.

3 Vectorization Analysis

3.1 Experimental Setup

To understand the state of the art of the most popular C/C++ compilers' auto-vectorizers, we ran a set of benchmark kernels on x86 and aarch64 architectures which support Intel's AVX2 or ARM's NEON vector instruction set. For this purpose, we used one of the benchmarks from Maleki et al.'s related study.

The Test Suite for Vectorizing Compilers (TSVC) [1] was originally published by Callahan, Dongarra, and Levine in 1988 and contained 135 Fortran loops [6]. Those were ported to C, outdated language constructs were removed, and 23 loop patterns were added [14]. Currently, the benchmark consists of 151 basic loops, grouped and numbered by vectorization challenge, such as loop peeling, statement reordering or data dependences. In TSVC, each loop itself contains only a few statements, i.e. loop bodies are short in terms of instructions. This synthetic setup is needed to test vectorization for very specific code patterns.

In our assessment, we used three popular C/C++ production compilers: the Gnu Compiler Collection (gcc), the Intel C Compiler (icc), and LLVM using the Clang frontend in their respective latest version (see Table 1). Only standard compiler flags were applied for benchmarking, and no further code annotations were added with the exception of pragma annotations to enforce data alignment. However, we did use a debug build of LLVM's release to obtain detailed vectorization reports. An overview of the compiler setups is provided in Table 1.

The most commonly used SIMD extensions today are Intel's AVX2 and ARM's NEON. Although newer SIMD extensions have been introduced, they are not yet commonly available. For example, Intel's AVX-512 is only available on their Xeon Phi platform, while ARM's successor of NEON, SVE, has been announced, but no product is on the market yet.

We therefore chose a server and a desktop processor with AVX2, as well as an embedded processor supporting NEON. The profiled server processor is an Intel Xeon E5-2679, while the desktop processor is an Intel i5-7500. Both support 256 bit floating point and integer operations, i.e. a vectorization factor of 8 is the maximum for our single-precision floating point benchmark. Choosing two processors with the same SIMD extension furthermore allows us to compare different implementations of the same instruction set and analyze their impact on vectorization rate and quality. Due to the backwards compatibility of x86 architecture, we were also able to run the benchmark targeting the older SSE4.2. These results are useful to understand if the progress in vectorization stems from an improved compiler, or an improved hardware ISA when comparing to previously published analysis numbers.

In order to compare the x86 results with a NEON ISA, we ran the benchmarks on an ARM Cortex-A53 as well. This ARMv8 architecture supports 128 bit vector operations, i.e. a maximum vectorization factor of 4 for single-precision floating point operations. Since the icc compiler does not generate code for aarch64 architectures, we only present gcc's and LLVM's results for that hardware. All numbers are generated on a single core without further parallelization techniques that could overshadow the vectorization results, for example multi-threading.

3.2 Vectorization Rate

The first metric to analyze is the vectorization rate, i.e. the number of loops that have been vectorized. Here, we test if a compiler is able to find a legal code transformation to vectorize a pattern. This incorporates the code analysis, transformation and a profitability analysis. For each compiler, results may vary across platforms due to the different underlying ISAs. For example, a certain instruction type can be essential for vectorization, but it may not be supported on all platforms. The ISAs also impact the profitability analysis. For example, larger SIMD vectors can allow for a bigger VF, which can result in a vectorization to be profitable despite the added overhead.

The obtained vectorization rates, sorted by TSVC's pattern groups, are shown in Table 2. In this table, we listed the number of vectorized loops for the AVX2 and NEON instruction set (if applicable), as well as the average speedup for the vectorized loops. As a comparison, we furthermore added results for the SSE4.2 extension, which highlights the progress made in vectorization due to improved SIMD ISAs.

Out of the 151 TSVC loop patterns, GCC is able to vectorize 64 loops for SSE4.2 (42%), 72 loops for AVX2 (48%), and 63 loops on the A53 (42%). In the related study from 2011, GCC was able to vectorize 59 loops (39%) on an Intel Nehalem i7 processor, utilizing the SSE4.2 SIMD ISA. It therefore shows an improvement in the compiler's

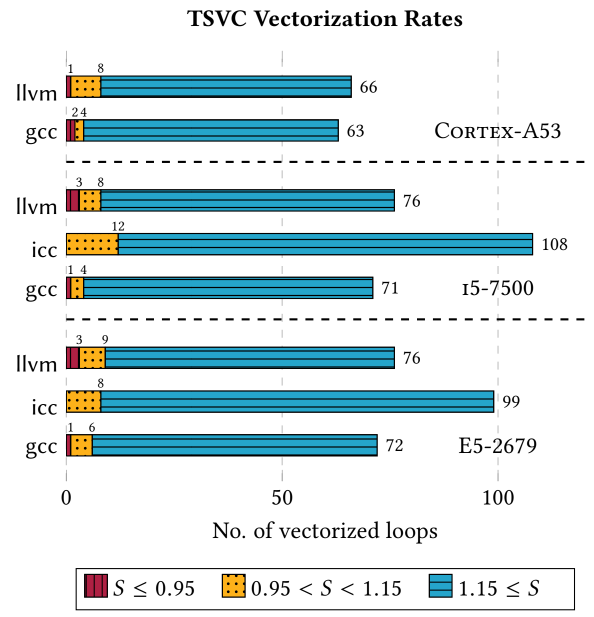


Figure 1. Vectorization rates of the TSVC benchmark, classified by speedup factor

auto-vectorization by 3%. However, Maleki et al.'s numbers are not solely based on the vectorization reports, but also on resulting speedup. A loop is considered vectorized if a speedup of at least 15% is achieved. In contrast, we base our analysis on the compilers' vectorization report and therefore capture codes that do not benefit from vectorization or even exhibit a slowdown. Nonetheless, we classified our results by speedup factor as well, but limited this analysis to the current SIMD extensions AVX2 and NEON. The results are depicted in Figure 1. Assuming the same minimum speedup factor of 1.15 and removing loops that exhibit slowdowns or scalar performance, gcc is able to vectorize 58 loops on the E5 (44%), 67 loops on the i5 (44%), and 59 loops on the A53 (39%).

The vectorization rates of icc are significantly higher than gcc's. It is able to vectorize 107 loops (72%) for SSE4.2 and 108 loops (72%) for AVX2 on the i5 platform. For SSE4.2, this marks an increase from 90 to 96 beneficially vectorized loops, i.e. +6%. However, the vectorization rate is not the same for AVX2 on the E5 Xeon platform. Here, only 99 loops (66%) are vectorized. When looking at the performance classification in Figure 1, it can be seen that the number of loops that were vectorized but show now speedup is higher on the i5 platform, reducing the difference in beneficial vectorization to four loops. A further analysis of the runtimes show that these differences are indirect adressing patterns, which were not deemed beneficial on the E5 platform, but on the i5. LLVM's vectorization rates are also higher than the ones achieved by gcc's, albeit not as high as icc's. It is able to vectorize 63 loops for SSE4.2 (42%), 76 loops for AVX2 (50%), and 66 loops on the A53 (44%). Applying the metric to filter loops with speedup only, 67 loops are profitable on the E5 (44%), 68 on the i5 (44%), and 58 on the A53 (38%). Furthermore, LLVM produces the highest rate of slowdowns on the AVX2 architectures. An analysis showed that this is likely due to a bug in the address calculation scheme of the scalar fall-back option.

Table 2. Number of vectorized patterns sorted by pattern groups for three different SIMD extensions; numbers in parentheses indicate average speedups on the E5-2679 and Cortex-A53, respectively

			LLVM			gcc			icc	
Pattern Group	Pattern IDs	#	SSE4.2	AVX2	NEON	SSE4.2	AVX2	NEON	SSE4.2	AVX2
Dependences	11*, 14*	15	7 (1.52x)	7 (1.49x)	7 (2.81x)	7 (2.16x)	8 (1.78x)	8 (2.52x)	10 (1.90x)	8 (2.02x)
Induction Var Recognition	12*	8	3 (1.59x)	4 (1.49x)	3 (2.90x)	5 (1.50x)	5 (1.55x)	5 (1.94x)	5 (1.53x)	4 (1.62x)
Data Flow Analysis	13*, 15*	4	3 (2.22x)	3 (2.00x)	4 (3.45x)	2 (2.38x)	2 (3.89x)	3 (2.79x)	4 (2.38x)	4 2.91(x)
Control Flow	16*, 27*, 44*, 48*	22	1 (0.87x)	11 (1.46x)	2 (1.25x)	4 (2.06x)	4 (2.59x)	4 (1.70x)	18 (1.66x)	18 (1.77x)
Symbolic Resolution	17*	6	5 (2.38x)	5 (2.41x)	5 (3.79x)	3 (2.22x)	3 (2.54x)	3 (3.07x)	3 (2.59x)	3 (2.71x)
Statement Reordering	21*	3	—	—	—	—	—	—	3 (1.68x)	3 (1.62x)
Loop Distribution	22*	3	1 (2.94x)	1 (2.95x)	1 (4.38x)	1 (2.91x)	1 (2.89x)	1 (0.81x)	3 (1.31x)	3 (1.40x)
Loop Interchange	23*	6	1 (1.04x)	1 (1.03x)	1 (1.04x)	3 (2.79x)	3 (4.09x)	3 (2.08x)	4 (1.49x)	4 (1.43x)
Node Splitting	24*	6	2 (1.62x)	2 (1.71x)	2 (2.07x)	1 (1.65x)	1 (1.92x)	1 (1.99x)	6 (2.08x)	6 (2.13x)
Scalar/Array Expansion	25*, 26*	12	5 (1.52x)	6 (1.66x)	5 (3.10x)	3 (1.36x)	3 (1.46x)	3 (1.69x)	6 (1.78x)	6 (1.73x)
Index Set Splitting	28*	2	1 (1.57x)	1 (1.62x)	1 (1.88x)	1 (1.83x)	1 (1.85x)	1 (1.87x)	1 (1.70x)	1 (1.63x)
Loop Peeling	29*	3	—	—	—	—	—	—	2 (2.54x)	2 (2.61x)
Diagonals	210*	3	1 (1.00x)	1 (1.00x)	—	1 (1.00x)	1 (1.00x)	—	1 (1.00x)	1 (0.99x)
Reduction	31*, v*	28	18 (11.75x)	19 (15.93x)	18 (3.91x)	20 (6.91x)	22 (12.08x)	19 (2.84x)	23 (3.00x)	22 (4.68x)
Search Loops	33*	2	—	—	—	—	—	—	1 (1.91x)	1 (3.25x)
Loop Rerolling	35*	4	4 1.43(x)	4 (2.48x)	4 (1.76x)	2 (1.75x)	2 (1.80x)	2 (2.54x)	2 (2.04x)	2 (1.85x)
Indirect Addressing	411*	6	—	—	2 (1.88x)	—	5 (2.46x)	—	4 (1.37x)	—
Statement Function Calls	412*, 47*	2	2 (1.48x)	2 (1.56x)	2 (2.75x)	2 (1.60x)	2 (1.88x)	2 (2.29x)	2 (1.78x)	2 (1.62x)
Equivalencing	42*	5	5 (1.49x)	5 (1.49x)	5 (2.01x)	5 (2.11x)	5 (2.03x)	5 (2.99x)	5 (1.97x)	5 (2.24x)
Parameters	43*	1	1 (2.19x)	1 (2.11x)	1 (3.98x)	1 (1.91x)	1 (2.24x)	1 (2.88x)	1 (2.55x)	1 (2.33x)
Intrinsic Functions	45*	3	3 (1.77x)	3 (6.81x)	3 (3.17x)	3 (2.55x)	3 (31.0x)	3 (3.15x)	3 (2.95x)	3 (3.84x)
Other	32*, 34*, 49*	7	—	—	—	—	—	—	—	—
Sum		151	63	76	66	64	72	63	107	99

Overall, the results also show that there are pattern groups that are not vectorized by any compiler (recurrences, packing, and vector semantics) and pattern groups that were vectorized by only one compiler (statement reordering, search loops) or only for specific platforms (indirect addressing). We will discuss further details for the differences in target hardware in section 3.4.

3.3 Vectorization Quality

Apart from the vectorization rate and the speedup classification, the execution times must be examined in more detail. This is necessary because speedup is a relative measure, i.e. it is related to the scalar code execution time. Hence measuring the average speedup provides an insight of how well a compiler can improve its own scalar code. However, this metric is not suitable to compare the quality of the produced code with the results from other compilers since they do not share the same baseline. We therefore calculated the geometric means of all loop execution times, including the non-vectorized ones. Including all test loops is critical because each compiler vectorizes a different subset of patterns (a trait that we will discuss in section 3.4). With these numbers, it is possible to determine an average speedup factor that the compiler can achieve. It is also possible to compare these numbers to a common baseline and we chose GCC's scalar execution time for this purpose. The results are presented in Figure 2.

The measurements show that ICC is significantly better for the basic loop patterns of TSVC on x86 platforms, which is consistent with the vectorization rate analysis from the previous discussion. Furthermore, GCC is ahead of LLVM for x86 platforms, but shows comparable performance on the ARM processor.

3.4 Differences between AVX2 and NEON Vectorization

The subsets of TSVC loops that are vectorized depend on the compiler and the target ISA, while our analysis of vectorization rates in Section 3.2 indicates that different implementations of the same ISA only have a minor impact. The only exception is icc with a different benefit analysis for indirect addressing schemes when vectorizing for AVX2. In our case, only gcc vectorized one loop more on the E5 platform, and icc did not vectorize indirect addressing patterns on the E5. Both compilers decided to forgo vectorization due to their cost models deeming the transformation not beneficial. It indicates that the vectorizers are able to find correct code transformations, however. Otherwise the same set of loops was vectorized on the two AVX2 platforms, although with different speedups. Nonetheless, there are significant differences between the compilers and their vectorization rates for the two instruction sets. These differences are depicted with a set of Venn diagrams in Figure 3.

When looking at the AVX2 ISA, icc is able to vectorize significantly more loops than its two competitors (108 vs. 72 and 76, respectively). However, there are a few loops that either gcc (3) or LLVM (5) can vectorize, but where icc fails. These patterns require certain dependence testing or loop-rerolling to be vectorized. LLVM, for example, is able to vectorize the unrolled loops with its SLP vectorization pass.

The difference in the vectorization algorithms of gcc and LLVM shows for both ISAs, AVX2 and NEON. In both cases, either compiler is able to vectorize 10-15 codes exclusively. When examining the pattern types of these exclusively vectorized loops, it is not possible to narrow them down to specific groups. As ICC does not compile for ARM ISAs, such a comparison cannot be performed.

When looking at the vectorization rates of each compiler for the two SIMD ISAs, the results show that it is significantly lower

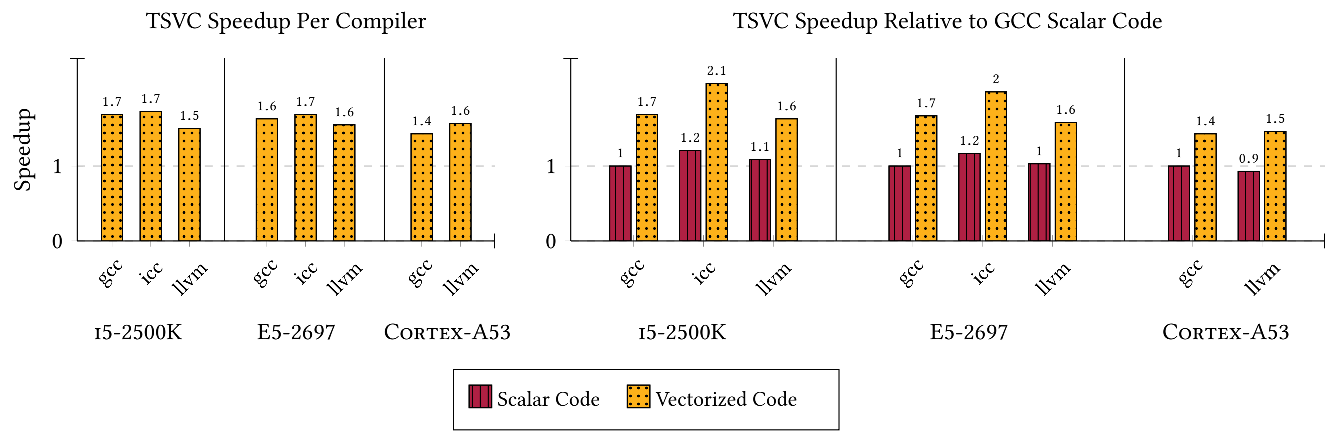


Figure 2. Speedups and relative execution times after vectorization; results are based on the geometric mean of the full benchmark execution time (including non-vectorized codes)

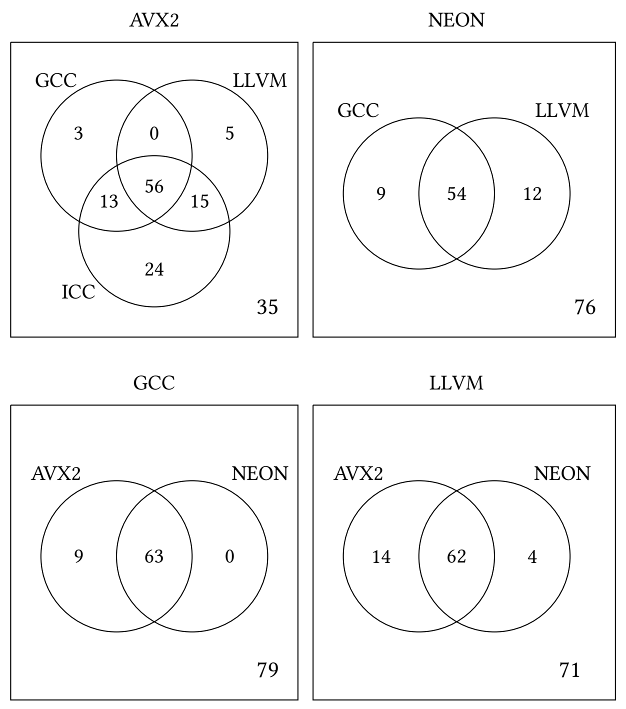


Figure 3. Sets of TSVC loops vectorized by different compilers on different platforms

for the NEON platform. GCC vectorizes 9 loops only on the AVX2 processor, but none exclusively for the NEON ISA. LLVM is able to optimize 14 loops on AVX2 only, and 4 exclusively on the ARM processor. Two of the four loops on ARM have indirect addressing patterns. Another interesting observation is that for both compilers, **13 out of the 14 patterns which were exclusively vectorized on AVX2 contain control flow**. This shows that both compilers are able to find a vectorization in general, but either cannot implement it with the NEON instruction set or deem it not beneficial.

4 Control Flow Vectorization

Based on the observation that loops with control flow are only vectorized for AVX2 ISAs, we analyzed the LLVM compiler to understand the limitations regarding the NEON instruction set.

4.1 State of the Art in LLVM

Listing 1 presents a basic loop containing control flow.

Listing 1. Basic loop containing control flow

```
int cond[n], in[n], out[n];
...
for (int i = 0; i < n; i++){
   if (cond[i]){
      out[i] = in[i] + 1;
   }
}
```

In this example, the value of `cond[i]` determines if the following basic block is executed. As a consequence, the memory operations to read `in[i]` and write `out[i]` are executed conditionally, i.e. they are not performed for every loop iteration. The same applies to the arithmetic operation of `in[i]+1`. To vectorize such a code pattern, the compiler thus has to

1. conditionally read data
2. vectorize the arithmetic operations within the basic block
3. and conditionally store data.

These three steps can be solved independently within the compiler. In general, the second step of vectorizing the arithmetic operations is the same as for every other basic block or loop body and therefore does not require any modifications to the compiler. This is not the case for the memory operations. Here, three specific techniques are available:

- scalarizing
- hoisting/sinking
- and masking.

For load operations, *scalarizing* means a gathering of single data elements and aggregating them into a vector before applying the vectorized arithmetic operations. A scalarized store will use a regular store operation to write back only those elements of loop iterations where the conditional will hold true. It can be used with any type of load operation, i.e. a vector load, masked load, or scalarized load. A scalarized load, however, requires a scalarized store and adds a significant overhead, which is why it is currently not used in LLVM.

Hoisting and sinking are further options currently utilized by LLVM. *Hoisting a load* is a technique to move memory read operations outside (*above*) the predicated basic block. This is done if the memory location is either read outside of the predicated block anyway, or if it is accessed in both branches of an if-then-else statement. In both cases, the compiler can assume that it is always safe to access this memory location and it can perform a vector load. The same conditions apply when *sinking a store*. If the memory location is written to outside of the predicated block, or a value is assigned to it in both cases of an if-then-else statement, the store instruction can be moved outside (*below*) the basic block; it is *sunk*. Since the memory accesses are no longer predicated after being hoisted/sunk, they can be vectorized. An example where hoisting and sinking can be applied is shown in Listing 2.

Listing 2. Example of a loop where hoisting and sinking of memory operations can be applied

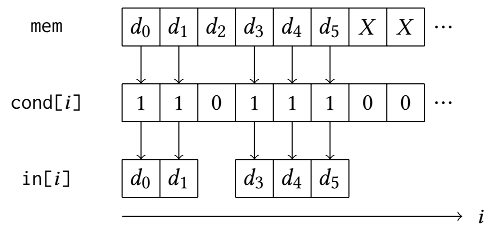
```
int cond[n], in[n], out[n];
...
for (int i = 0; i < n; i++){
    if (cond[i]){
        out[i] = in[i] + 1;
    }
    else{
        out[i] = in[i] - 2;
    }
}
```

Nonetheless, further instructions, such as a select or vector shuffling, might be required to obtain the correct write-back data. In the shown example, a write back vector has to be created from the results of the if and else branches.

The third, and most straight-forward approach, are masked load/store operations. These vector instructions accept as an argument a binary mask that defines which elements of a vector should be read from/written to. They thereby assure that only these memory locations will be accessed or modified. However, this class of instructions is not available for all SIMD extensions. In our case, the AVX2 based platforms support them, while the NEON ISA extension does not. This is the root cause for the difference in vectorization rates on the two platforms, which we observed in section 3.4. Besides the missing masked operations for NEON, LLVM was not able to apply hoisting of read instructions to the benchmark codes. If the read operation is not vectorized, however, LLVM will not apply a partial vectorization of the basic block, for example by performing a scalarized load, vectorized arithmetic instructions, and a sunk store. As a consequence, most of the control flow loops were not vectorized for the NEON platform.

4.2 Vectorizing Load Operations for NEON

The challenge with vectorizing predicated load operations is to identify if it is legal to access all memory locations within the vector, which is the rationale behind load hoisting. If this is not possible and masked load operations are not available, the programmer can annotate the code by using pragmas such as `#pragma clang loop vectorize (assume_safety)`. This pragma explicitly declares



mem
d_0 d_1 d_2 d_3 d_4 d_5 X X ···
cond[i]
1 1 0 1 1 1 0 0 ···
in[i]
d_0 d_1 d_3 d_4 d_5
i

Figure 4. Example of using a conditional statement to mask out memory read accesses

all memory accesses to be safe and will result in a hoisted load. However, this is not an automatic approach and requires insight, knowledge and effort by the programmer. We therefore present an automatic methodology that does not rely on pragma annotations, but can be applied to any predicated load operation.

Figure 4 shows an example of a conditional load, where the conditional statement is used to mask out memory accesses. In this example, it is safe to access all six elements of `in[i]` if they are located in the same memory page. That does not mean that all elements have to be accessed, as shown for data element d_2, where the conditional resolves to false despite the element being present in memory. For indices of $i > 5$, the kernel does not have access permissions for `in[i]` and the content of `mem` is undefined (X). Trying to access the successive element of d_6 would therefore lead to a memory access violation. Hence the conditional statement `cond[i]` prevents these accesses by masking out all memory accesses for larger indices. However, when vectorizating this code, such a transition from legal to illegal memory accesses can lie within a vector. Assuming a vectorization factor of four for the example, the second vectorized load would try to access elements $d_4 - d_7$, where trying to read the latter two can lead to access violations during execution. Nonetheless, information such as the size of the loaded array is not necessarily present and thus cannot be statically evaluated. The same applies to the trip count of the loop, i.e. the number of iterations. We therefore propose adding a runtime check for each vectorized loop execution to assess if it is safe to perform a vector load based on the evaluated conditional `cond[i]`, which is independent of the input array sizes or loop iteration counts.

For our approach, we make the following two assumptions:

- allocated memory is contiguous, i.e. an array is stored in a contiguous memory region without gaps or jumps
- accesses to memory can be expressed as a linear affine expressions, i.e. addresses are strictly increasing or decreasing.

Both of these conditions can be statically identified by the compiler today. As a first step to determine if it safe to perform a vector load, it needs to be understood how the vector elements are located in memory. For example, if all vector or array elements lie within one page of memory, it is safe to perform a vector load if at least one conditional evaluates to true. If all vector elements lie within two consecutive pages of memory, the first and last element of the conditional mask need to be set. In other words, per page of memory that is accessed within a vector, at least one conditional mask element needs to evaluate to true. To enable a vectorized load, a runtime check to test the conditional mask for a specific pattern can then be created based on the individual memory access pattern. These masks could potentially be tuned to each individual kernel, e.g. by determining the iteration with a page break and splitting the loop accordingly at runtime to allow for a simple conditional check. However, this can also introduce significant overhead, which needs to be evaluated further. We therefore initially limit our approach to patterns where **all vector elements lie within one or two pages of memory**. Handling the case of all vector elements in two memory pages is the more generic approach and works for the first case, all elements in one page, as well. This approach can be applied if the compiler can determine a constant distance between vector elements of less than $\frac{pagesize}{n \cdot distance}$. In this case, it is sufficient to test if the first and last element of the conditional evaluates to true (or at least one element per page, which would have to be checked at runtime, thus adding more overhead). This is shown in Listing 3.

Listing 3. Proposed runtime check to enable partial vectorization of conditional read operations for vectors whose elements lie within two pages of memory

```
vec cond_v [n];
cond_v = ...

if (cond_v[0] & cond_v[n-1])
      do_vec;
else
      do_scalar;
}
```

First, the conditional vector cond_v is calculated for VF loop iterations. If it evaluates to true for the first and last iteration/element, cond_v[0] and cond_v[n-1] in the example, it is safe to vectorize the load and the vectorized code block can be executed. If it evaluates to false, it falls back to scalar code execution. The overhead added is therefore a straight forward binary test of one and operation per VF loop iterations. However, the conditional calculation is still vectorial, and our results in Section 5 show that it compensates the 3% of added overhead.When assuming a probability of 50% for the conditional to evaluate to true, the benefit of vectorization outweights the performance impact of the added runtime check by far, especially for kernels with a high arithmetic intensity. Furthermore, the compiler is able to determine if adding overhead can be compensated by vectorization through its cost analysis, as the runtime check is static and does not depend on the basic block's code.

However, for small distances between elements, e.g. a consecutive access as shown in Figure 4, a page break within a vector will rarely occur. Here it would be sufficient to test if at least one element in the conditional mask evaluates to true for the majority of iterations. In such cases, it is more efficient to determine the number of iterations within a memory page based on the start addresses of the read vectors and split the loop accordingly. This will add overhead at runtime, but this calculation is executed only once per accessed memory page. With this additional calculation, a simple runtime check of (|cond_v) suffices to determine if the vectorized code should be executed; otherwise, the next vector iteration can be evaluated.

With these proposals, it is possible to vectorize load operations on architectures without masked load instructions, such as processors supporting NEON. The approach is automatic and does not require code annotations or further code analysis, while adding limited overhead that is compensated by the vectorized conditional calculation. However, the approach is only applicable to loops where the stride between vector elements can be determined.

4.3 Vectorizing Store Operations

As mentioned in Section 4.1, there are three techniques available for conditional store operations: using scalar predicated stores, sinking the store or applying masked store instructions. Sinking the store is dependent on the code and cannot be applied in general, while masked store store instructions are not available for all architectures, such as those supporting NEON. As a consequence, the scalar predicated store is the only globally applicable technique to foster conditional store instructions. But for the NEON instruction set, we have implemented an additional option: the *select store*.

The idea stems from the fact that the NEON instruction set supports select instructions, which merge two vectors based on a binary mask. Using the loop in Listing 1 as an example, the *select store* is implemented in a read-modify-write sequence:

1. the original data of out is read from memory with a vector load for the next VF elements
2. the results of the basic block, i.e. in[i] + 1, are caculated for VF iterations
3. a write back vector is determined by merging the results of the basic block and the orginal values of out that were read in the first step, using the conditional cond_v as a select mask
4. the write back vector is written to memory with a vector store, overwriting all data elements in memory.

Listing 4 shows how to transform a masked store to a *select store*.

Listing 4. Example of transforming a masked store to a *select store*

```
vec cond_v [n], result_v [n], out_v [n];

// masked store
masked_store(& out_v, cond_v, result_v);

// select store
vec tmp_v [n], wb_data_v [n];

tmp_v     = vec_load(&out_v);
wb_data_v = select(cond_v, result_v, tmp_v);
vec_store (wb_data_v);
}
```

In order to apply a *select store*, it must be safe to read the data before modifying it. With our approach of vectorizing load operations based on the evaluated conditional, we can guarantee this safety. However, updating the data/writing in memory has an additional set of requirements. As long as the loop is executed in a single thread, it is safe to apply the above *select store* approach. But as soon as multi-threading is applied, it cannot be guaranteed that other threads do not access those data elements masked out by the conditional. In those cases, data of a masked out iteration could be modified after the read, but before the write back of the *select store*. As a consequence, the *select store* would overwrite elements with outdated content, causing corrupt data in memory. One way to avoid such scenario is to ensure that each thread processes a chunk of the loop that is a multiple of the vectorization factor. One options is annotating the code, for example by using the OpenMP pragma `pragma omp parallel for (kind, chunk size)`. However, code annotation means that the approach is no longer fully automatic.

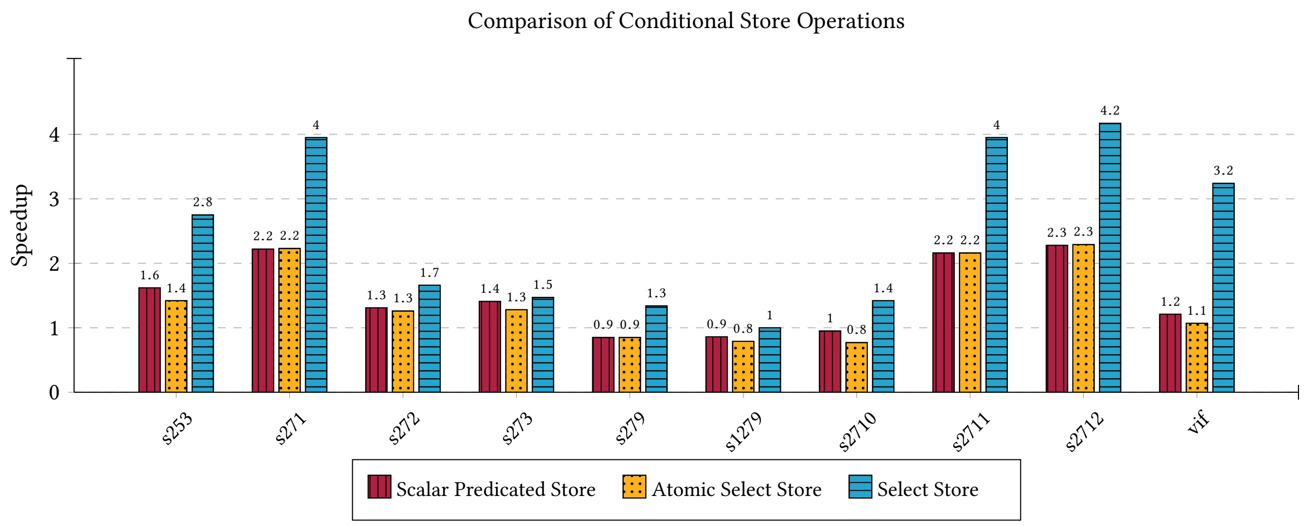


Figure 5. Performance comparison of different options to implement conditional store for loops with control flow, measured on the Cortex-A53 with VF = 4

We therefore implemented another option: making the sequence of read-modify-write atomic. This *atomic select store* is translated to a `load-acquire-store-release` loop and a `bit select` instruction in the compiler. Due to the restrictions of atomic operations, it is now not possible for other threads to modify the data processed by the locking thread. It therefore ensures data coherency and the *atomic select store* can be applied automatically to all loops where it cannot be ensured at compile time that the program is either single threaded or the chunk size is a multiple of the VF.

Atomic operations do come with an overhead, however. We thus compared the performance of the *select store*, the *atomic select store*, and the – already existing – predicated scalar store. The results will be discussed in the next section.

5 Results

5.1 Overhead Analysis for Conditional Load Operations

The goal of our approach is to enable the vectorization of loops while adding only a minimal overhead. We therefore need to analyze three aspects in particular:

- the overhead added
- the worst case performance
- the best case performance

The overhead can be assessed when the vectorized code is never executed, i.e. when the test of `(cond_v[0] & cond_v[3])` always evaluates to false and the fall back option of scalar code is executed for all loop iterations. To test this scenario, we used kernel pattern `s271`, shown in Listing 5.

Listing 5. Kernel for overhead analysis of conditional load

```
// code pattern s271
for (int i = 0; i < n; i++){
   if (b[i] > 0){
      a[i] += b[i] * c[i];
   }
}
```

Furthermore, we set all values of `b` to zero and implemented two intrinsics based versions of the code, one with and one without the added runtime check; both versions use a scalar predicated store. The measured execution times show that the runtime check adds an overhead of merely 3%. This number is based on the vectorized calculation of `b[i] > 0`. For regular scalar code, each element would be evaluated separately, and the three percent overhead added by the runtime check will be compensated by the vectorized conditional calculation.

To emulate the worst case scenario, we set the values in `b` so the conditional statement would evaluate to the pattern

$$\texttt{cond} = \{1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1 \ldots\}$$

Here, the full overhead of vectorization is added for each iteration, although only half the number of elements per vector are executed. However, the vectorized version including the runtime check still achieves a speedup over scalar code of 1.34x.

The best-case scenario is when the vectorized code is always taken, i.e. the full advantage can be taken from the vectorized code. For this purpose, all values of `b` were set to values greater than zero. We also enabled the auto-vectorizer to vectorize the loop by annotating the code so that load hoisting could be performed. When comparing the execution times, we are almost able to reach the auto-vectorizer's performance (a speedup of 1.91x vs 1.79x). So although we are exploiting the full speedup potential, we are able to vectorize loops that are currently not vectorized. Hence we achieve a significant speedup over state of the art with our approach.

These numbers are generated with the straight-forward code pattern shown in Listing 5. For even simpler patterns, the overhead might be slightly higher. However, the overhead is static, as the conditional check does not depend on the actual kernel code within the basic block. For kernels with a higher arithmetic intensity, the percentage of overhead will therefore decrease further.

5.2 Performance Comparison of Select Store Operations

Out of the 13 loop patterns containing control flow that were not vectorized, three of them could be vectorized after ensuring the safety of hoisting the load via code annotation. We therefore limited our performance analysis to the remaining ten loops and benchmarked the codes after compiling them with a modified LLVM to apply our *select store* and *atomic select store* techniques. Figure 5 shows the results of this analysis.

For all loops, the *select store* approach outperforms the *scalar predicated store* and the *atomic select store*. The achieved speedup is higher by at least 5%, and for some patterns, it is up to a factor of 2x better. Hence it is always recommendable to apply this technique when a code is executed in single-thread mode, or a thread chunk size of a multiple of the VF can be guaranteed.

For multi-threaded applications where this cannot be ensured, the *scalar predicated store* and the *atomic select store* need to be compared. The figure shows that the performance of the two is comparable for most of the loops, with the *atomic select store* being slightly behind the performance of the *scalar predicated store*. This is due to the overhead added by making the instruction atomic. Furthermore, there are loops (s279, s1279, and s2710), where both approaches show a slowdown. Here, the profitability analysis of the compiler fails and code is vectorized despite an overhead effacing all performance gains.

It must be noted that all speedup measurements are relative to the scalar code performance and depend on the branching probability, its distribution and the vectorization factor. We therefore performed a second evaluation to understand these factors better in the next subsection.

5.3 Profitability Analysis

The focus of this work has been to enable auto-vectorization of loops containing control flow for architectures without native support for masked instructions. As seen in Figure 5, however, vectorization is not necessarily profitable. We therefore analyzed the profitability of our approaches as well and identified the key contributing factors as:

- **the largest vectorized data type**: the available vector registers have a fixed length, e.g. 128 bit for the Cortex-A53. The data type therefore directly impacts the maximum possible vectorization factor, which again impacts the possible speedup. So far, all results are based on single-precision floating point arrays, i.e. vectors of 4x32 bit. For this analysis, we scale the dataypes from characters (8 bit, VF = 16) to double precision floating point (64 bit, VF = 2).
- **the branching probability**: when vectorizing basic blocks with control flow, iterations will be executed that are masked out by the conditional statement. Their results will be discarded before the write back. Therefore, when an if-branch is rarely taken, the vectorized code might not be beneficial due to the redundantly executed iterations. In the experiment, we scaled the branching probability from 0% to 100% assuming an interleaved scheme, i.e. for a ratio of 25%, one out of every four iterations was set to be taken.
- **the arithmetic intensity**: vectorization in general adds overhead. For example, an aggregation of vector elements might be needed. Thus it can be faster to execute code scalarly, especially if there are few arithmetic operations that can be sped up within the processor. To understand this impact, we executed our analysis with two different kernels showing a varying arithmetic intensity. They are shown in Listing 6.

Listing 6. Kernels with different arithmetic intensity for profitability analysis

```
type cond[n], a[n], b[n], c[n], d[n];

// Kernel 1
for (int i = 0; i < n; i++){
   if (cond[i]){
      a[i] += 1;
   }
}

// Kernel 2
for (int i = 0; i < n; i++){
   if (cond[i]){
      a[i] = b[i] + c[i] * d[i];
   }
}
```

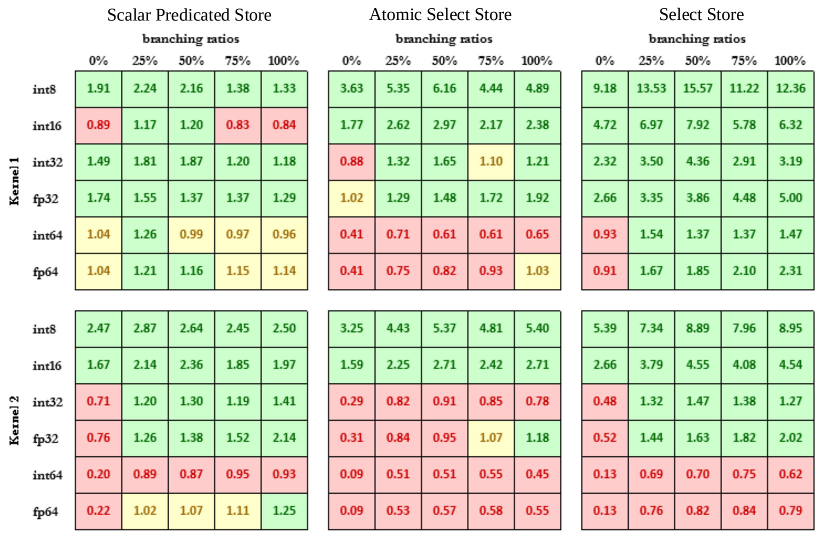
The results of this experiment are shown in the heatmaps of Figure 6. It can be seen that the profitability of vectorization is limited for the larger data types, i.e. 64 bit integers and floating point numbers. For the first kernel with the lower arithmetic intensity only the *select store* is profitable, while the *scalar predicated store* is in the range of scalar performance and the overhead of the *atomic select store* causes slowdowns. Another observation is that all techniques cause slowdowns if the vectorized code is never executed, i.e. the ratio is 0%. Again, especially large data types are affected.

It can also be seen, however, that all three techniques show significant speedups for smaller data types, such as characters (8 bit) or shorts (16 bit). The only exception is the *scalar predicated store* for 16 bit integers, where the branch predictor causes significant performance impacts. When taking a closer look at the magnitude of these speedups, both select store approaches outperform the state of the art, the *scalar predicated store*. For the kernel with the lower arithmetic intensity, the speedup over the state of the art ranges between 1.9x and 9.3x, while for the second kernel the speedup over state of the art ranges between 0.95X and 3.58x. In this case, the slowdown results from an execution probability of 0% and the added overhead of the never-executed code.

In today's auto-vectorizers, the branching probability is assumed to be 50% when performing the profitability analysis. Based on this assumption, our methods of *select store* and *atomic select store* always outperform the state of the art for conditional store operations of small data types. It is thus safe to replace it with our approach, yielding a significantly higher speedup by a factor of up to 7x.

6 Conclusions

Vectorization is an important opitmization technique in today's compilers. We analyzed the capabilities of the most popular C/C++ compilers' auto-vectorizers. The analysis was performed on hardware platforms supporting AVX2 and NEON. On the AVX2 platforms, GCC and ICC were able to increase their vectorization rates compared to a similar analysis performed in 2011. However, we show in our analysis that the vectorization rates are lower for NEON platforms. We identified the problem to be the missing masked load/store operations needed to vectorize basic blocks that contain control flow, i.e. that are wrapped by an if(-else) statement.



		Scalar Predicated Store					Atomic Select Store					Select Store				
		branching ratios					branching ratios					branching ratios				
		0%	25%	50%	75%	100%	0%	25%	50%	75%	100%	0%	25%	50%	75%	100%
Kernel 1	int8	1.91	2.24	2.16	1.38	1.33	3.63	5.35	6.16	4.44	4.89	9.18	13.53	15.57	11.22	12.36
	int16	0.89	1.17	1.20	0.83	0.84	1.77	2.62	2.97	2.17	2.38	4.72	6.97	7.92	5.78	6.32
	int32	1.49	1.81	1.87	1.20	1.18	0.88	1.32	1.65	1.10	1.21	2.32	3.50	4.36	2.91	3.19
	fp32	1.74	1.55	1.37	1.37	1.29	1.02	1.29	1.48	1.72	1.92	2.66	3.35	3.86	4.48	5.00
	int64	1.04	1.26	0.99	0.97	0.96	0.41	0.71	0.61	0.61	0.65	0.93	1.54	1.37	1.37	1.47
	fp64	1.04	1.21	1.16	1.15	1.14	0.41	0.75	0.82	0.93	1.03	0.91	1.67	1.85	2.10	2.31
Kernel 2	int8	2.47	2.87	2.64	2.45	2.50	3.25	4.43	5.37	4.81	5.40	5.39	7.34	8.89	7.96	8.95
	int16	1.67	2.14	2.36	1.85	1.97	1.59	2.25	2.71	2.42	2.71	2.66	3.79	4.55	4.08	4.54
	int32	0.71	1.20	1.30	1.19	1.41	0.29	0.82	0.91	0.85	0.78	0.48	1.32	1.47	1.38	1.27
	fp32	0.76	1.26	1.38	1.52	2.14	0.31	0.84	0.95	1.07	1.18	0.52	1.44	1.63	1.82	2.02
	int64	0.20	0.89	0.87	0.95	0.93	0.09	0.51	0.51	0.55	0.45	0.13	0.69	0.70	0.75	0.62
	fp64	0.22	1.02	1.07	1.11	1.25	0.09	0.53	0.57	0.58	0.55	0.13	0.76	0.82	0.84	0.79

Figure 6. Heatmaps of experiment to analyze vectorization profitability; shown are speedups when scaling data type sizes and branching ratios for two kernels with different arithmetic intensities

We therefore proposed two techniques: one to safely vectorize conditional load, and one to vectorize conditional store. These approaches have been implemented in the LLVM compiler, as they are independent of the code to be vectorized and can be applied automatically during compilation. Furthermore, the overhead added for the vectorized load is neglectible, while it enables loop vectorization by the auto-vectorizers, i.e. without further code annotations. We also showed significant improvements by our technique of *select stores*. Even when applying *atomic select stores*, we demonstrate that there are use cases where a significant speedup over the state-of-the-art scalar predicated store can be achieved.

References

[1] 2011. Extended Test Suite for Vectorizing Compilers. (2011). http://polaris.cs.uiuc.edu/~maleki/TSVC.tar.gz

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '83)*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/567067.567085

[3] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.* 9, 4 (Oct. 1987). DOI: http://dx.doi.org/10.1145/29873.29875

[4] ARM Limited. 2018. NEON. (2018). https://developer.arm.com/technologies/neon

[5] D. I. August, W. W. Hwu, and S. A. Mahlke. 1997. A framework for balancing control flow and predication. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. DOI: http://dx.doi.org/10.1109/MICRO.1997.645801

[6] D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing Compilers: A Test Suite and Results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing (Supercomputing '88)*. IEEE Computer Society Press. http://dl.acm.org/citation.cfm?id=62972.62987

[7] A. M. Erosa and L. J. Hendren. 1994. Taming control flow: a structured approach to eliminating goto statements. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on*. DOI: http://dx.doi.org/10.1109/ICCL.1994.288377

[8] Intel Corp. 2018. Intel Architecture Instruction Set Extensions and Future Features Programming Reference, 319433-032. (2018).

[9] Intel Corp. 2018. Intrinsics for Masked Load/Store Operations. (2018). https://software.intel.com/en-us/node/695108

[10] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. 2013. Automatic vectorization of tree traversals. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*.

[11] Ralf Karrenberg. 2015. *Whole-Function Vectorization*. Springer Fachmedien Wiesbaden, Wiesbaden, 85–125. DOI: http://dx.doi.org/10.1007/978-3-658-10113-8_6

[12] Seonggun Kim and Hwansoo Han. 2012. Efficient SIMD Code Generation for Irregular Kernels. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM. DOI: http://dx.doi.org/10.1145/2145816.2145824

[13] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM. DOI: http://dx.doi.org/10.1145/349299.349320

[14] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society. DOI: http://dx.doi.org/10.1109/PACT.2011.68

[15] Gaurav Mitra, Beau Johnston, Alistair P Rendell, Eric McCreath, and Jun Zhou. 2013. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 1107–1116.

[16] Matt Pharr and William R. Mark. 2012. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Proceedings Innovative Parallel Computing (InPar)*.

[17] Angela Pohl, Biagio Cosenza, Mauricio Alvarez Mesa, Chi Ching Chi, and Ben Juurlink. 2016. An Evaluation of Current SIMD Programming Models for C++. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing (WPMVP '16)*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/2870650.2870653

[18] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society. http://dl.acm.org/citation.cfm?id=2738600.2738625

[19] Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. DOI: http://dx.doi.org/10.1109/CGO.2013.6494989

[20] Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society. DOI: http://dx.doi.org/10.1109/CGO.2005.33

[21] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (March 2017). DOI: http://dx.doi.org/10.1109/MM.2017.35

[22] Hans Zima and Barbara Chapman. 1991. *Supercompilers for Parallel and Vector Computers*. ACM, New York, NY, USA.