# Assessing the State of Autovectorization Support based on SVE

1st Bine Brank
*Jülich Supercomputing Centre*
*Forschungszentrum Jülich*
Jülich, Germany
b.brank@fz-juelich.de

2nd Dirk Pleiter
*PDC Center for High Performance Computing*
*KTH Royal Institute of Technology*
Stockholm, Sweden
0000-0001-7296-7817

***Abstract*—So-called SIMD instructions, which trigger operations that process in each clock cycle a data tuple, have become widespread in modern processor architectures. In particular, processors for high-performance computing (HPC) systems rely on this additional level of parallelism to reach a high throughput of arithmetic operations. Leveraging these SIMD instructions can still be challenging for application software developers. This challenge has become simpler due to a compiler technique called auto-vectorization. In this paper, we explore the current state of auto-vectorization capabilities using state-of-the-art compilers using a recent extension of the Arm instruction set architecture, called SVE. We measure the performance gains on a recent processor architecture supporting SVE, namely the Fujitsu A64FX processor.**

***Index Terms*—ISA, auto-vectorization, Arm, SVE**

## I. Introduction

Typical state-of-the-art CPU core architectures implement instructions that follow the Single Instruction-Multiple Data (SIMD) paradigm. We use the term SIMD instruction to refer to instructions where a single instruction triggers operations that process in each clock cycle a data tuple, i.e. multiple data. An early example of an Instruction Set Architectures (ISA) comprising SIMD instructions is Intel's Streaming SIMD Extensions (SSE), which was introduced in 1999 as an extension to the x86 ISA. The operands of SSE instructions have a size of 128 bits. Later, Intel quadrupled the width of the operands when introducing AVX512.

SIMD instructions have meanwhile been included in all ISAs used for general-purpose processors suitable for high-end computing. This also includes Arm, which recently introduced the Scalable Vector Extension (SVE) [1].[1] A remarkable feature of this SIMD ISA is that the operands do not have a width fixed by the ISA. SVE operands could have any width between 128 and 2048 bits, but the width must be a multiple of 128 bits. The first processor supporting SVE instructions has been Fujitsu's A64FX processor, which has been developed in a co-design process with RIKEN [2] and features an SVE operand width of 512 bits.

To fully exploit the performance of processor architectures like the A64FX or Intel processors supporting AVX512, this SIMD parallelism cannot be ignored. These SIMD ISAs can be explicitly exploited using special functions, supported by popular compilers and called intrinsic functions (or built-in functions). This approach results, however, in non-portable code. This may be acceptable if additional efforts for maintaining such codes can be justified. But for application software, it is typically much more attractive to rely on the ability of compilers to generate code that leverages a SIMD ISA. This technique is called auto-vectorization.

We use the advent of the SVE ISA as an opportunity to assess the auto-vectorization capabilities of today's compilers. This investigation is based on the TSVC2 benchmark suite, which has been developed and extended as a benchmark suite to test and challenge the auto-vectorization capabilities of compilers [3], [4]. We measure the achieved performance gain using the A64FX processor architecture. In selected cases, we perform a comparison with manual vectorization using intrinsic functions. This analysis allows us to reason about instructions that are not available in the current SVE ISA but could be exploited for vectorizing one or more of the explored benchmarks. Given that we focus on a set of synthetic benchmarks, we would like to stress that from the speed-up factors reported in this short paper one cannot infer the speed-up achieved for real-life applications.

This paper makes the following contributions:

1) We provide results that allow us to compare the auto-vectorization capabilities of different state-of-the-art compilers using the TSVC benchmark suite. This includes a manually vectorized version of 136 benchmarks using SVE intrinsics.
2) The benchmarks are executed on the A64FX hardware to obtain quantitative results on the speed-up.
3) We explore possible gaps in the SVE ISA and identify three instructions that could be added to SVE to create additional opportunities for vectorization.

This paper is organised as follows: In section II we present related work. Next, we document our methodology in section III, followed by a documentation of the results in section IV. These results are analysed in section V before we present a summary and conclusions in section VI.

[1]Meanwhile an extended version called SVE2 is available, which, however, is not considered here.

## II. Related Work

The Test Suite for Vectorizing Compilers (TSVC), developed by Callahan et al. [3], is a common benchmark suite to evaluate auto-vectorization. In 2011, the original version was expanded to 151 loops by Maleki et al. [4]. In this work, vectorization capabilities were evaluated for different compilers including ICC, XLC and GCC. 124 to 127 loops were identified as vectorizable loops with compilers vectorizing 59 to 90 loops. Additionally, he classified loops by optimization techniques. Moldovanova et al. [5] studied the auto-vectorization of loops on Intel64 and Intel Xeon Phi. Here, a higher degree of vectorization (79-100) is reported. Another benchmark suite developed for testing the auto-vectorization capabilities of compilers is the PolyBench/C benchmark suite [6]. Other benchmark suites, like Video SIMDBench [7], have been designed according to typical code patterns of a specific domain.

These benchmarks have not only been used to test compilers but also to compare improvements that can be obtained on different architectures. For instance, in [4] the speed-up has been measured for processors from the Intel Nehalem as well as the IBM Power 7 generation of processors. While this work focuses on results obtained with compiler-generated code, we also compare it with manually vectorized code using intrinsic functions.

Since Arm proposed the SVE ISA [1], a lot of work has been performed on assessing the benefits of this novel SIMD ISA. The focus has been mainly on full or simplified applications, so-called mini-applications. In [8] the generation of SVE instructions by different compilers is explored for different benchmarks and mini-applications using an instruction emulator. Leveraging the vectorization opportunities based on SVE has been explored for different applications, e.g. applications from the area of LQCD [9], or different numerical kernels, e.g. FFT [10]. This work does, however, not systematically evaluate the auto-vectorization capabilities of the used compilers, explore to which extent the SVE ISA is exploited and/or investigate possible extensions of this SIMD ISA.

Due to its unique features, the A64FX processor has attracted significant interest and various publications have become available that evaluate this novel processor architecture, e.g. [11]–[13]. The A64FX processor was the first to support the SVE ISA. Meanwhile, AWS's Graviton3 became available, which is not considered here. It supports SVE instructions with operands of a width of 256 bits, i.e. half the width compared to the A64FX. An alternative strategy for exploring performance is to use simulators like gem5 (see, e.g., [14]).

## III. Methodology

We evaluate the compilers' ability to vectorize loops for SVE using the TSVC2. TSVC is a benchmark suite for evaluating a compiler's auto-vectorization capabilities. The original version, written by Callahan et al. [3], contained 135 synthetic loops. In 2011, TSVC was extended to 151 loops (TSVC2). The benchmarks test various strategies for vectorization like dependence testing, statement reordering, loops interchange, scalar expansion, etc. The loops have not been extracted from any real-life application but were explicitly written for compiler analysis. The benchmark is designed to cover various types of loops with access to 1- or 2-dimensional arrays. It covers, e.g., various cases of linear dependence testing with or without jumps in data access. One example is the loop *s111* shown in Listing 1 where the loop counter is incremented by two in each iteration. Furthermore, cases of induction variable recognition are tested, where array access depends on a variable that is increased (or decreased) by a variable amount in each iteration. Also, loops used for packing arrays conditionally are tested. One such example is the loop *s341* shown in Listing 4.

We analyze three compilers: GCC 11.1.0, Arm Compiler for Linux (ACfL) 22.0.1 and the Clang version of the Fujitsu Compiler 4.7 (FCC). SVE is targeted with `-march=armv8.2-a+sve`. For each compiler, we enable **all** optimizations which we deem relevant for vectorization. Most importantly, we enable `-Ofast` which sets the `-ffast-math` flag. This flag enables the biggest set of optimizations and may break strict compliance with the IEEE 754 standard for floating-point arithmetic. In the case of GCC, we also enable `-fivopts` which enables induction variable optimization on trees and use `aarch64-auto-vectorization-preference` to only vectorize loops with SVE instructions and not NEON. For ACfL and FCC, `-ffp-contract=fast` was used to enable fused multiply-add operations. In the case of GCC this is enabled by default. In the case of ACfL, the option `-fsimdmath` allows the compiler to generate calls to vectorized ArmPL library routines. Table I documents the specific flags that have been used for each of the compilers. The array lengths were set to `LEN_1D=8000` and `LEN_2D=80` to ensure that the entire data footprint fits into the L2 cache. Each loop was executed 1000 times. To compare results with a fixed-size SIMD ISA, we have also performed experiments targeting AVX-512. For this, we used GCC and changed the target with `-march=cascadelake --mprefer-vector-width=512`.

We perform both, a static and dynamic analysis. First, we check the compiler-generated executables by inspecting the assembly and reviewing the compiler optimisation reports. Here, we wrote a script that reads the optimisation reports and parses the result for each TSVC loop. For cases where compilers produce different SVE codes, we use the LLVM machine code analyzer (llvm-mca) to analyze different compilations. We target SVE with `llvm-mca -march=arm64 -mcpu=a64fx`. [2] As a metric for comparing the compilers, we focus on the number of loops that have been vectorized.

For the dynamic analysis, we rely on the A64FX processor. The A64FX is the first processor that implements SVE and features two 512-bit SVE units. We compare the benchmark

[2]We use llvm-mca (26.6.2022 trunk) on Godbolt compiler explorer available at http://www.godbolt.org.

execution times for the three different compilers. For this purpose, we introduce the vectorization speed-up

$$\eta = \frac{\Delta t_{\mathrm{scalar}}}{\Delta t_{\mathrm{vec}}}\,, \tag{1}$$

where $\Delta t_{\mathrm{scalar}}$ and $\Delta t_{\mathrm{vec}}$ are the execution time for the scalar and vectorized versions of the benchmarks. Scalar binaries are produced using the same flags but adding `-fno-tree-vectorize` or `-fno-vectorize`. Instead of the vectorization speed-up, one may also consider the efficiency

$$\epsilon = \frac{\eta}{N_{\mathrm{lane}}}\,, \tag{2}$$

where $N_{\mathrm{lane}}$ is the size of the data tuples that fit into a SIMD operand. Since all loops use single-precision floating-point numbers and we are considering SVE operands with a width of 512 bits, we here have $N_{\mathrm{lane}} = 16$.

## IV. Results

Table II reports the number of SVE vectorized loops in TSVC by different compilers. FCC vectorizes most loops (97), followed by GCC and ACfL. The latter does identify more opportunities for vectorization. However, based on an internally computed metric, the ACfL compiler decides in 11 cases that vectorization for SVE target is not beneficial. In 5 of 11 cases, loops are instead vectorized with NEON instructions. On the other hand, GCC and FCC do not report any cases where vectorization is not beneficial. Figure 1 shows how vectorized loops overlap between different compilers. (For ACfL, 11 above-mentioned loops are included.) A total of 115 loops were vectorized by at least one compiler, and 80 loops were vectorized by all three. Recent studies [5] [15] also report similar numbers (95-111) for x86's AVX-512 SIMD set. Our experiment with AVX-512 shows that GCC vectorizes 97 loops of which 4 loops were not vectorized for SVE and 1 loop was vectorized for SVE but not AVX-512. Four extra loops were, however, vectorized with 256-bit vectors (AVX-256) and were also vectorized by ACfL and FCC for SVE. Therefore, the VLA paradigm does not introduce any significant drawbacks in terms of vectorization. On top of that, we manually vectorized 21 more loops with intrinsic functions giving 136 vectorizable loops. This code is publicly available.[3]

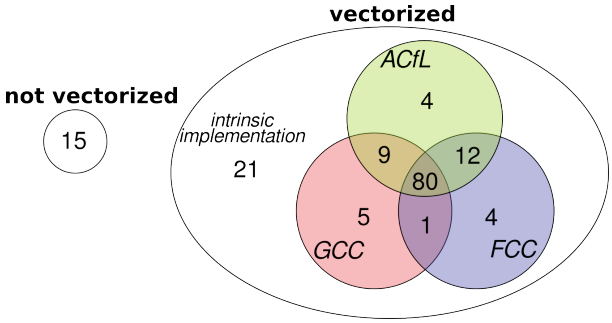


Fig. 1. Vectorization in TSVC (151 loops)

[3]https://gitlab.jsc.fz-juelich.de/brank1/tsvc_sve

Table III shows the total run-time of the entire TSVC2 benchmark suite (151 loops) on the A64FX processor. Here, $\Delta T_{\mathrm{scalar}}$ and $\Delta T_{\mathrm{vec}}$ refers to the accumulated execution time of all benchmarks without and with vectorization by the compiler enabled. $\Delta T_{\mathrm{intrinsic}}$ shows the total execution time where all 136 vectorizable loops were manually vectorized with intrinsic functions. However, for loops autovectorized by compilers, the intrinsic code is practically the same as compiler-generated code. The fastest binaries are produced by FCC (2.68 s), followed by ACfL (2.92 s) and GCC (3.73 s). Although this corresponds with the number of vectorized loops, the result is, partly, also a consequence of faster scalar code. We also notice that the GCC code is approximately 40% slower than FCC. We observe that FCC uses aggressive loop-unrolling and many loops are automatically unrolled over a factor of two or four. This is reflected in the size of the FCC's binary, which is 2.2 and 2.9 times bigger than those generated by the ACfL and GCC compilers. Also, we note that some loops take significantly longer time than others, so the total average time difference is significantly weighted towards a few computationally more expensive loops. Our intrinsic implementation takes 1.64-1.69 seconds with compilers showing little difference in performance. Intrinsic code is 40% faster than FCC, however, for the majority of loops, the difference between auto-vectorization and intrinsic implementation is significantly smaller. The 40% difference mostly comes from additional vectorization, because $\Delta T_{\mathrm{vec}}$ is dominated by loops which are not vectorized.

In Table IV, we count the number of loops that achieve a vector speed-up of at least 1.15, 2, 4, and 8. In all cases, FCC reports less loops with bigger speed-up, which is a consequence of a faster scalar code. Additionally, we observed several loops where $\epsilon$ is close to 1 ($\eta$ close to 16).

## V. Analysis

From inspecting the code generated by the compilers for the TSVC2 benchmarks we conclude that key features of the SVE ISA are exploited. All three compilers, e.g., generated gather-load and scatter-store instructions in loops with complex memory patterns. Furthermore, all compilers used reduction instructions, and predicate register as a mask for loops with conditional statements. A strange compilation was observed for reversed loops (i.e. loops with decrementing loop counter). In SVE, predicate driven loop control is achieved with *whilele* and *whilelt* instructions that create a predicate register equivalent to loop exit condition. However, *whilege* or *whilegt* are not defined[4] and compilers rely on the *rev* instruction to reverse the contents of vector registers in each iteration. Other than that, we did not notice any drawbacks that could be attributed to the VLA feature of SVE. We observed that GCC produces in various cases significantly different assembly compared to the LLVM-based compilers (ACfL and FCC). For example, consider the loop *s111*, which is shown in Listing 1. The loop counter is incremented by two

[4]Instructions *whilegt* and *whilege* were introduced in SVE2.

TABLE I
COMPILER FLAGS

| GCC (gcc) | ACfL (armclang) | FCC (fcc -Nclang) |
|---|---|---|
| `-march=armv8.2-a+sve -Ofast -fivopts --param aarch64-autovec-preference=2 (-fno-tree-vectorize)` | `-march=armv8.2-a+sve -Ofast -fsimdmath -ffp-contract=fast (-fno-vectorize)` | `-march=armv8.2-a+sve -Ofast -ffp-contract=fast (-fno-vectorize)` |

TABLE II
THE NUMBER OF LOOPS THAT HAVE BEEN VECTORIZED (IN BRACKETS IS THE NUMBER OF ADDITIONAL LOOPS THAT HAVE BEEN IDENTIFIED AS VECTORIZABLE BUT HAVE NOT BEEN VECTORIZED FOR SVE).

| GCC | ACfL | FCC | Intrinsic |
|---|---|---|---|
| 94 (+0) | 93 (+11) | 97 (+0) | 136 |

TABLE III
TSVC TOTAL TIME IN UNITS OF SECONDS.

| | GCC | ACfL | FCC |
|---|---|---|---|
| $\Delta T_{\text{scalar}}$ | 7.58 | 8.84 | 6.10 |
| $\Delta T_{\text{vec}}$ | 3.73 | 2.92 | 2.68 |
| $\Delta T_{\text{intrinsic}}$ | 1.64 | 1.68 | 1.69 |

in each iteration and requires handling of data with stride two. All three compilers vectorize this loop, but with a different approach. GCC uses *ld2* instructions to load elements of `a` and `b`, interleaving odd and even into separate registers. Afterwards, a scatter-store instruction is used to write the data to `a[i]` (see Listing 9 in the Appendix). On the other hand, ACfL and FCC use normal *ld1* load and *st1* store instructions (see Listing 10 in Appendix). This requires four additional *zip* instructions to separate even and odd elements (two for data registers and two for predicate registers). llvm-mca predicts that GCC code is 2.42 times faster than ACfL, however, our measurement shows only a difference of 20%.

```
1  for (int i = 1; i < LEN_1D; i += 2) {
2      a[i] = a[i - 1] + b[i];
3  }
```

Listing 1. Loop *s111*

Another example of different code generation is observed for loop *s124* (see Listing 2). An important observation in this loop is that `j` is incremented during each iteration and that all elements of array `a` are overwritten. All three compilers generate vector compare instructions to create a predicate register, whose active lanes satisfy the condition of positive `b` values. However, such predicate is used differently by GCC than by Clang based compilers. GCC uses two *fmla* instructions, where the second predicated *fmla* overwrites those lanes where the condition evaluates to true (see Listing 11 in Appendix). ACfL and FCC rather use a predicated *sel* instruction to copy `b[i]` and `c[i]` in the right lanes of an SVE register. Afterwards, only a single *fmla* operations is required (see Listing 12 in Appendix). In this case, llvm-mca predicts the code generated by GCC to be 24% faster than ACfL, while measurement on the A64FX hardware shows that GCC is 12% faster than ACfL.

TABLE IV
TSVC SPEED-UP.

| | GCC | ACfL | FCC |
|---|---|---|---|
| $\eta \geq 1.15$ | 86 | 83 | 76 |
| $\eta \geq 2$ | 81 | 77 | 73 |
| $\eta \geq 4$ | 72 | 67 | 51 |
| $\eta \geq 8$ | 42 | 46 | 23 |

```
1  for (int i = 0; i < LEN_1D; i++) {
2      if (b[i] > (real_t)0.) {
3          j++;
4          a[j] = b[i] + d[i] * e[i];
5      } else {
6          j++;
7          a[j] = c[i] + d[i] * e[i];
8      }
9  }
```

Listing 2. Loop *s124*

For loops that were vectorized by only one compiler, we noticed that GCC is slightly better at loops involving `goto` constructs (see Listing 3 as an example). For ACfL and FCC, we did not observe any patterns that would suggest them being better for a particular family of loops. Table V shows loops vectorized by only one compiler.

```
1  for (int i = 0; i < LEN_1D-1; ++i) {
2      if (b[i] < (real_t)0.) {
3          goto L20;
4      }
5      a[i] = c[i] + d[i] * e[i];
6      goto L10;
7  L20:
8      c[i+1] = a[i] + d[i] * d[i];
9  L10:
10     ;
11 }
```

Listing 3. Loop *s161*

TABLE V
TSVC LOOPS VECTORIZED BY ONLY ONE COMPILER

| GCC | ACfL | FCC |
|---|---|---|
| *s122 s231 s235 s257 s331* | *s115 s118 s119 s2102* | *s212 s241 s256 s351* |

### A. Limits of the auto-vectorizers

We identified 21 loops that were not vectorized by any compiler, but we managed to vectorize these using SVE intrinsics. Loop *s341* (see Listing 4) was an example of a loop where the compilers fail to spot an opportunity of leveraging the SVE ISA.

In this loop, all values of `b` which are greater than zero are packed in the array `a`. At first, the loop seems hard to vectorize due to the increment `j++` which makes `j` depend on previous

iterations. However, SVE includes a *compact* instruction which concatenates active elements of the register and fills the rest with zero. Therefore, we can first use a *cmpgt* instruction to find all lanes where `b[i] > 0.0f`, followed by a *cntp* instruction to count number of such elements. Afterward, this information is used to store the right number of elements to `a` and correctly increment `j`. The intrinsic implementation is shown in Listing 5.

```
1  j = -1;
2  for (int i = 0; i < LEN_1D; i++) {
3      if (b[i] > (real_t)0.) {
4          j++;
5          a[j] = b[i];
6      }
7  }
```

Listing 4. Loop *s341*

```
1  do {
2      svfloat32_t bv = svld1_f32(pg, &b[i]);
3      svbool_t cg = svcmpgt(pg, bv, zerov);
4      svfloat32_t res = svcompact_f32(cg, bv);
5      int inc_j = svcntp_b32(svptrue_b32(), cg);
6      svbool_t tg = svwhilelt_b32(0, inc_j);
7      svst1(tg, &a[j], res);
8      j += inc_j;
9      i += svcntw();
10     pg = svwhilelt_b32(i, LEN_1D);
11 } while (svptest_any(svptrue_b32(), pg));
```

Listing 5. SVE vectorization of loop *s341*

In other loops, the compiler failed to realise vectorization opportunities for reasons that are not specific to SVE. In these cases, loops require a specific transformation to enable vectorization. The compilers do not recognize these opportunities in various cases. Here we give a brief overview of the transformations that we used:

- **Loop splitting** is a technique where the loop is split into two separate loops. We use this to vectorize loops where straightforward vectorization fails due to a specific iteration. This is because other iterations depend on it or the particular iteration has different properties than other iterations. In most cases, vectorization can still be applied if the special iteration is handled with scalar instructions. This applies to loops *s1113* and *s281*.
- **Loop peeling** is a special case of loop splitting where the first (or last) few iterations are split from the loop and performed outside the loop. A missed opportunity for vectorization that requires loop peeling was observed for loops *s244*, *s254*, *s255*, *s291*, *s292* and *s293*.
- **Pre-loading** is a technique where data is pre-loaded into a SIMD vector to store the copy of the original data before it gets overwritten. A missed opportunity for vectorization involving pre-loading was observed in *s211*, *s1213*, *s241*, *s243*, and *s1244*.
- **Searching loops** are loops that search the first value in an array (and its corresponding index) that satisfies a certain condition. This applies to loops *s332*, *s481*, *s482*, *s3110*, and *s13110*. A way to vectorize this loop is to use the SVE vector compare operation to see if a searched value is in the current iteration. If it is found then it is manually checked one by one for the first value that fits the condition. The performance of such vectorization depends on the values of a[i].
- Some loops involve a combination of multiple techniques for a successful vectorization (for example, *s126*, *s232*, *s2251*).

### B. Limits of the ISA

The TSVC2 benchmark suite also allows identifying cases, where vectorization does not fail due to the lacking capabilities of the compiler's auto-vectorizer but rather due to limitations of the ISA. These cases can be used for identifying possible new instructions to facilitate vectorization also in these cases.

One example is loop *s3112* (see Listing 6), which computes a prefix sum. Although there exist algorithms that compute a prefix sum using SIMD instructions, they usually do that with partial sums in multiple sweeps. This could be avoided with a hardware implementation of a prefix sum SVE instruction[5] (see Figure 2). Since there is already the *fadda* instruction, which computes a sum-reduction in order, prefix sums already appear naturally during the execution of this operation. Therefore, a modification to prefix sum does not seem far-fetched.

```
1  sum = (real_t)0.0;
2  for (int i = 0; i < LEN_1D; i++) {
3      sum += a[i];
4      b[i] = sum;
5  }
```

Listing 6. Loop *s3112*

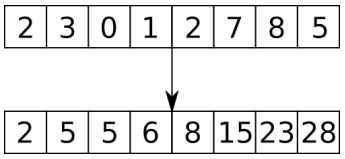


Fig. 2. Possible prefix sum instruction

Loop *s342* (see Listing 7) leads to the opposite case. Array element `b[j]` is only unpacked if the array element *a[i]* is positive. SVE does not include instructions that would unpack the vector under predicate control (see Figure 3). Such an instruction, which would have an opposite effect than the *compact* instruction, would be needed for vectorizing *s342*.

```
1  for (int i = 0; i < LEN_1D; i++) {
2      if (a[i] > (real_t)0.) {
3          j++;
4          a[i] = b[j];
5      }
6  }
```

Listing 7. Loop *s342*

Loop *s258* is shown in Listing 8. The temporary variable `s` only changes value during specific iterations. Such a loop could be vectorized if instructions existed that would copy the last active element under predicate control. An example of such a potential instruction is presented in Figure 4.

[5]The idea of prefix sum hardware implementation was first introduced for vector computers in 1990 by Chatterjee et al.

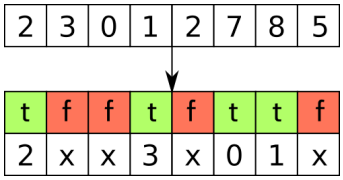


Fig. 3. Possible unpack instruction

```
for (int i = 0; i < LEN_2D; ++i) {
    if (a[i] > 0.) {
        s = d[i] * d[i];
    }
    b[i] = s * c[i] + d[i];
    e[i] = (s + (real_t)1.) * aa[0][i];
}
```

Listing 8. Loop *s258*

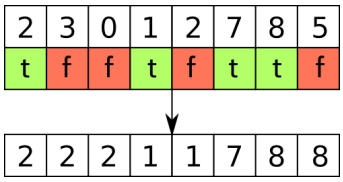


Fig. 4. Possible last active instruction

## VI. Summary and Conclusions

In this work, we have assessed current compiler auto-vectorization capabilities based on SVE. SVE is a novel SIMD ISA for Arm that introduces the novel feature of vector-length-agnostic (VLA) SIMD instructions. We noticed that the investigated compilers, namely GCC, Arm Compiler for Linux (ACfL), and the Fujitsu C Compiler (FCC), have no problems adapting to the VLA target. The results of TSVC2 benchmark show that these compilers can vectorize up to 115 of 151 loops and generate SVE SIMD instructions. Additionally, key features of the SVE ISA including gather/scatter, reduction, and predicate instructions are exploited. Although the number of vectorized loops is similar, the fastest code was produced by FCC, followed by ACfL and GCC. In certain cases, Clang-based compilers (ACfL and FCC) produce significantly different codes than GCC. We have also shown that compilers fail to vectorize many loops which can be vectorized manually. We provided a vectorized implementation of 136 benchmarks using SVE intrinsic functions. The execution time of the entire TSVC2 benchmark suite can be improved by 40% when performing such a manual vectorization. Finally, we identified three loops where a small change of the SVE ISA, namely the addition of three instructions, could open new vectorization opportunities.

## Acknowledgements

The authors would like to thank the Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a $5M National Science Foundation grant (#1927880). Furthermore, we want to thank the Open Edge and HPC Initiative for access to an Arm-based development environment through the HAICGU cluster at the Goethe University of Frankfurt. Funding for parts of this work has been received from the European Commission H2020 program under Grant Agreement 779877 (Mont-Blanc 2020) as well as from the Swedish e-Science Research Centre (SeRC).

## Appendix

In the following we document different examples that illustrate the large differences in code generated by GCC and ACfL.

### A. *Different compilation of loop s111*

```
.L337:
    ld2w    {z0.s - z1.s}, p0/z, [x2]
    ld2w    {z2.s - z3.s}, p0/z, [x3]
    add x5, x2, 4
    fadd    z0.s, z0.s, z2.s
    st1w    z0.s, p0, [x5, z4.s, sxtw 2]
    incw    x0
    mov w5, 3999
    add x3, x3, x4
    add x2, x2, x4
    whilelo p0.s, w0, w5
    b.any   .L337
```

Listing 9. GCC-generated assembly

```
.LBB153_2:
    ld1w    { z2.s }, p0/z, [x22, x9, lsl #2]
    ld1w    { z0.s }, p0/z, [x20, x9, lsl #2]
    ld1w    { z1.s }, p0/z, [x21, x9, lsl #2]
    pfalse  p1.b
    fmul    z3.s, z4.s, z2.s
    add x9, x9, x24
    fmul    z3.s, z1.s, z3.s
    fadd    z1.s, z1.s, z0.s
    fadd    z1.s, z1.s, z2.s
    fmad    z0.s, p3/m, z1.s, z3.s
    add x10, x23, w8, sxtw #2
    zip1    p2.s, p0.s, p1.s
    zip2    p1.s, p0.s, p1.s
    zip1    z1.s, z0.s, z0.s
    whilelo p0.s, x9, x19
    zip2    z0.s, z0.s, z0.s
    add w8, w8, w25
    st1w    { z1.s }, p2, [x10]
    st1w    { z0.s }, p1, [x10, #1, mul vl]
    b.mi    .LBB153_2
```

Listing 10. ACfL-generated assembly

### B. *Different compilation of loop s124*

```
.L519:
        ld1w    z0.s, p0/z, [x1, x0, lsl 2]
        ld1w    z1.s, p0/z, [x25, x0, lsl 2]
        ld1w    z2.s, p0/z, [x22, x0, lsl 2]
        ld1w    z3.s, p0/z, [x21, x0, lsl 2]
        fcmle   p1.s, p2/z, z0.s, #0.0
        fmla    z0.s, p2/m, z1.s, z2.s
        movprfx z0.s, p1/m, z3.s
        fmla    z0.s, p1/m, z1.s, z2.s
        st1w    z0.s, p0, [x19, x0, lsl 2]
        add     x0, x0, x23
        whilelo p0.s, w0, w20
        b.any   .L519
```

Listing 11. GCC-generated assembly

```
.LBB168_2:
        ld1w    { z0.s }, p0/z, [x20, x8, lsl #2]
        ld1w    { z1.s }, p0/z, [x22, x8, lsl #2]
        ld1w    { z3.s }, p0/z, [x23, x8, lsl #2]
        fcmgt   p1.s, p3/z, z0.s, #0.0
        not     p2.b, p3/z, p1.b
        and     p2.b, p3/z, p0.b, p2.b
        ld1w    { z2.s }, p2/z, [x21, x8, lsl #2]
        sel     z0.s, p1, z0.s, z2.s
        fmla    z0.s, p3/m, z3.s, z1.s
        st1w    { z0.s }, p0, [x24, x8, lsl #2]
        add     x8, x8, x25
        whilelo p0.s, x8, x19
        b.mi    .LBB168_2
```

Listing 12. ACfL-generated assembly

## REFERENCES


[1] A. Rico, J. A. Joao, C. Adeniyi-Jones, and E. Van Hensbergen, "Arm hpc ecosystem and the reemergence of vectors: Invited paper," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 329–334. [Online]. Available: https://doi.org/10.1145/3075564.3095086

[2] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-design for A64FX manycore processor and "fugaku"," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[3] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: a test suite and results," in *Supercomputing '88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*, 1988, pp. 98–105.

[4] S. Maleki, Y. Gao, M. J. Garzar´n, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 372–382.

[5] O. V. Moldovanova and M. G. Kurnosov, "Auto-vectorization of loops on intel 64 and intel xeon phi: Analysis and evaluation," in *Parallel Computing Technologies*, V. Malyshkin, Ed. Cham: Springer International Publishing, 2017, pp. 143–150.

[6] T. Yuki, "Understanding PolyBench/C 3.2 kernels," in *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, ser. IMPACT '14, Jan. 2014.

[7] M. Alvanos and P. Trancoso, "Video SIMDBench: Benchmarking the compiler vectorization for multimedia applications," in *2016 Euromicro Conference on Digital System Design (DSD)*, 2016, pp. 168–175.

[8] A. Poenaru and S. McIntosh-Smith, "Evaluating the effectiveness of a vector-length-agnostic instruction set," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds. Cham: Springer International Publishing, 2020, pp. 98–114.

[9] N. Meyer, P. Georg, D. Pleiter, S. Solbrig, and T. Wettig, "SVE-enabling lattice QCD codes," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 623–628.

[10] D. Takahashi and F. Franchetti, "Ffte on sve: Spiral-generated kernels," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPCAsia2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 114–122. [Online]. Available: https://doi.org/10.1145/3368474.3368488

[11] F. Banchelli, K. Peiro, G. Ramirez-Gargallo, J. Vinyals, D. Vicente, M. Garcia-Gasulla, and F. Mantovani, "Cluster of emerging technology: evaluation of a production hpc system based on A64FX," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 741–750.

[12] M. A. S. Bari, B. Chapman, A. Curtis, R. J. Harrison, E. Siegmann, N. A. Simakov, and M. D. Jones, "A64FX performance: experience on Ookami," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 711–718.

[13] B. Brank, S. Nassyr, F. Pouyan, and D. Pleiter, "Porting applications to Arm-based processors," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 559–566.

[14] L. Zaourar, M. Benazouz, A. Mouhagir, F. Jebali, T. Sassolas, J.-C. Weill, C. Falquez, N. Ho, D. Pleiter, A. Portero, E. Suarez, P. Petrakis, V. Papaefstathiou, M. Marazakis, M. Radulovic, F. Martinez, A. Armejach, M. Casas, A. Nocua, and R. Dolbeau, "Multilevel simulation-based co-design of next generation hpc microprocessors," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021, pp. 18–29.

[15] M. Rajan, D. W. Doerfler, M. Tupek, and S. Hammond, "An investigation of compiler vectorization on current and next-generation Intel processors using benchmarks and Sandia's Sierra applications," 2015.