A Performance Analysis of Vector Length Agnostic Code

Angela Pohl, Mirko Greese, Biagio Cosenza and Ben Juurlink
Embedded Systems Architecture
Technische Universität Berlin
Berlin, Germany
{angela.pohl, cosenza}@tu-berlin.de, mirko.greese@campus.tu-berlin.de

***Abstract*—Vector extensions are a popular mean to exploit data parallelism in applications. Over recent years, the most commonly used extensions have been growing in vector length and amount of vector instructions. However, code portability remains a problem when speaking about a compute continuum. Hence, vector length agnostic (VLA) architectures have been proposed for the future generations of ARM and RISC-V processors. With these architectures, code is vectorized independently of the vector length of the target hardware platform. It is therefore possible to tune software to a generic vector length. To understand the performance impact of VLA code compared to vector length specific code, we analyze the current capabilities of code generation for ARM's SVE architecture. Our experiments show that VLA code reaches about 90% of the performance of vector length specific code, i.e. a 10% overhead is inferred due to global predication of instructions. Furthermore, we show that code performance is not increasing proportionally with increasing vector lengths due to the higher memory demands.**

***keywords*—vectorization, SIMD, vector length agnostic, SVE**

I. Introduction

Exploiting data level parallelism by code vectorization is a common technique to speed up applications. Consequently, general purpose CPUs and server processors come equipped with Single Instruction Multiple Data (SIMD) Instruction Set Architectures (ISAs). While the trend over the past decade has been to keep increasing the vector length, i.e. the number of bits in a vector register, and adding new vector instructions, recent vector ISAs go down a different path: they revisit techniques such as loop predication to enable vector length agnostic (VLA) code generation. Albeit not being a new technique either [1], it solves the problem of code portability across hardware platforms and ISA generations. Using VLA programming, it is possible to run the same code on an embedded processor with a small vector length, as well as an HPC processor with large vector registers without re-coding or even re-compilation.

The most recently proposed VLA ISAs are ARM's Scalable Vector Extensions (SVE) [2] and RISC-V's V-Module [3]. Both architectures offer instructions whose vector length is not fixed and can be determined at execution time, thus enabling the generation of vector-length agnostic code by the compiler. RISC-V's V-Module furthermore allows the configuration of a specific vector length that will be most beneficial for a kernel. This vector length can be modified for different kernels within an application. Both ISAs provide different solutions for avoiding scalar loop tails when vectorizing code. RISC-V solves the problem by using smaller vector instructions for loop tails that would otherwise not utilize a full vector. ARM's SVE relies on loop predication, i.e. masking out vector elements and supporting partial vector execution. Gobal predication of instructions, however, causes an overhead that impacts performance. We therefore set out to understand the performance impact of vector length agnostic code compared with vector length specific code.

Although both ISAs have been proposed a few years ago, no hardware implementation is available on the market yet. While the specifications for the V-Module are still work in progress, Fujitsu has announced a first product that will support SVE. The *Post-K* processor [4] will support the SVE instruction set with a vector length of up to 512 bit. To get developers started, ARM provides tools to simulate the SVE ISA. Using this infrastructure, we

- benchmarked 151 auto-vectorized loop patterns in a full-system simulator
- compare the performance of VLA code run on a 128 bit architecture with state-of-the-art NEON code
- assess the overhead induced with global predication of instructions in SVE
- analyze the scaling of speedups with scaling vector lengths.

Our results show that, while SVE is competitive with NEON on average, the overhead added by predication is around 10%. We furthermore found that, despite the ability to scale vector lengths to large values, the added stress on the memory subsystem diminishes the return on investment.

II. Related Work

Although no product is available on the market, first application and performance evaluation papers have been published about SVE. Rico et al. [5] present an overview of the available HPC ecosystem for ARM processors and briefly introduce the scalable vector extensions. They present a simulation-based study of vectorization rates and speedups for a variety of benchmark kernels from HPC applications. Findings include that a significant speedup could be achieved for about a third

of the kernels, while a higher vectorization rate does not necessarily translate into a performance gain.

Kodama et al. [6] analyzed the impact of vector scaling on code performance. In their experiment, they increased the hardware's vector length while keeping all other hardware resources constant. They are able to show that compute-bound applications do benefit from larger vectors as long as sufficient hardware registers are available.

Meyer et al. [7] ported a framework for Lattice Quantum Chromodynamic Codes to SVE and evaluated the existing toolchain for their HPC problem. They were able to implement and prove correctness of their code using the ARM instruction emulator, but were not able to give a performance assessment due to a lacking HPC simulation environment.

Armejach et al. [8] implemented stencil codes using SVE assembly. They thoroughly discuss strategies for further performance improvements and present example codes that did not benefit from large vector lengths due to the increased number of memory accesses.

However, none of the aforementioned works analyze the impact of VLA code on performance. We therefore present results of two further experiments: comparing performance of 128 bit NEON code with 128 bit SVE VLA code, as well as comparing performance of SVE VLA code with SVE vector length specific code for large vector lengths.

III. ARM Scalable Vector Extensions

A thorough discussion of all features can be found in ARM's inaugural SVE paper [2]. A few of the most significant extensions compared to ARM's current NEON ISA are [5]:

- *Vector length scalability*, which enables code to adapt dynamically to a hardware's vector length by supporting vector length agnostic instructions
- *Enhanced addressing modes* that support gather-load and scatter-store vector operations
- *Per-lane predication*, with predicated instructions eliminating the need for scalar loop tails and enabling partially executed loop iterations
- *Extended horizontal operations*, which include floating-point, integer and bitwise logical reductions
- *Vector partitioning* to allow speculative vector loads and vectorization of data-dependent loops

When looking at SVE assembly, it can be seen that the concept of VLA code is implemented by predicating all vector instructions. An example is shown in Listing 4, where all instructions are invoked with an additional predicate register `p`. Furthermore, loop increments are not done with a fixed number that denotes the vector length. Instead, the loop variable is incremented with a constant value, which is indicated by the `incw` instruction that only takes a variable as an argument, but not the vector length.

IV. Experimental Setup

As no SVE hardware is available on the market yet, we used the gem5 simulator for our measurements [9]. gem5 is a modular platform for processor and system architecture simulation. It is actively supported by ARM and a branch to simulate hardware with SVE is publicly available [10]. Within this branch, there is a choice of three different CPU models: atomic, in-order, and out-of-order. Since SVE is targeted for HPC applications, we chose the out-of-order CPU model, which resembles an ARM Cortex-A72.

As a benchmark for our experiments, we used the *Test Suite for Vectorizing Compilers* [11], [12]. It contains 151 short loops that were originally intended to test the auto-vectorization capabilities of a compiler. All loops use 32 bit floating point data types and test different loop patterns, such as loops with control flow or indirect addressing schemes. For our experiment, we separated each loop into its own source file and ran it once to measure the execution time of each code pattern. Running the simulation once was sufficient due to its deterministic nature. The execution time was derived from the absolute number of simulated clock ticks per execution run.

To generate SVE vectorized code, we used GCC 8.2.0. It supports auto-vectorization for SVE as well as in-line assembly. In our experiments, we relied on GCC's auto-vectorization capabilities and used the `-msve-vector-bits` flag to switch between VLA and vector length specific code generation. Possible values for this flag are `scalable`, `128`, `256`, `512`, `1024`, and `2048`. Contrary to SVE's specifications, it is not yet possible to generate code for all vector lengths that are multiples of 128 bit, e.g. 384 bit. The default configuration is `scalable`, which generates VLA code. Furthermore, there is a restriction in GCC 8 when specifying the flag value as `128`; it will always generate VLA code.

V. Comparison with NEON

As a first experiment, we compared two metrics: the overall number of auto-vectorized loops, i.e. the vectorization rate, and the speedups achieved by NEON and SVE VLA code. As a second step, we analyzed the overhead that is introduced by SVE's global predication.

A. Vectorization Rates and Speedups

Out of the 151 loop patterns, 66 could be auto-vectorized for both architectures. GCC is furthermore able to vectorize 16 additional loops for the SVE hardware, i.e. 82 loops in total. There were no loops that were vectorized for NEON exclusively. This is depicted in the Venn diagram in Figure 2 below.

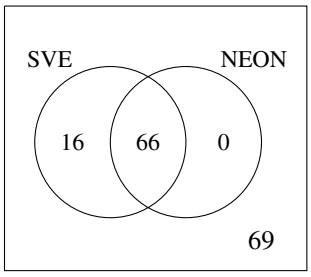

Fig. 2. Auto-vectorized TSVC loops

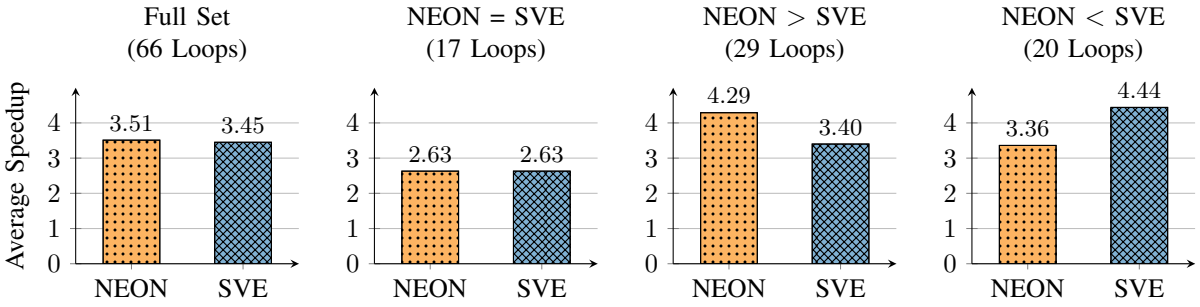

Fig. 1. Speedup comparison of loops vectorized with NEON and vector length agnostic SVE (128 b vector length); average speedup is determined by the geometric mean of individual loop speedups

The 16 additional loops were auto-vectorized due to the support of predication and scatter/gather operations in the SVE ISA, which has been lacking in the NEON ISA so far. For example, loops with indirect addressing schemes were not vectorized. There are techniques to implement the missing scatter/gather instructions for NEON, too, such as assembling vectors element by element or emulating a predicated store with existing instructions [13]. However, GCC did not apply them to the loops in questions.

We measured the benchmark's average speedup by calculating the geometric mean of the 66 individual loop speedups that were vectorized for both ISAs, SVE and NEON. All individual loop speedups are based on the runtimes of each loop on the out-of-order processor with a vector width of 128 bit. The results are depicted in Figure 1.

On average, the speedup achieved by SVE VLA code is 0.06 lower than for NEON. This could be attributed to the overhead introduced by loop predication, but a deeper analysis shows that the 66 loops can actually be split into three different sets:

1) `NEON = SVE` (17 loops): speedups are the same, i.e. within 5 percentage points of each other.
2) `NEON > SVE` (29 loops): loop speedup is more than 5 percentage points higher for NEON than for SVE.
3) `NEON < SVE` (20 loops): loop speedup is more than 5 percentage points lower for NEON than for SVE.

The geometric mean of the first set's speedups is 2.63x for both architectures. However, this set also contains three loops that are vectorized, but fall back to scalar code during execution due to unresolved dependences at runtime. When removing these three loops from set 1, the geometric mean of the set's speedups increases from 2.63x to 3.23x. For the second set, the average speedup is 0.81 higher for NEON ($S_{NEON} = 4.29$, $S_{SVE} = 3.40$), while it is 0.90 lower for the third set ($S_{NEON} = 3.36$, $S_{SVE} = 4.44$).

When looking at the type of loop patterns that fall into each set, we see a broad distribution among pattern types as shown in Table I. There are pattern types where one ISA achieves higher performance for all auto-vectorized loops of that category. For NEON, these categories are loops with control flow, index set splitting, jump in data accesses and node splitting, i.e. five loops in total. However, the difference in speedups is only 18 percentage points on average (geometric mean of $S_{NEON} - S_{SVE}$).

TABLE I
NUMBER OF LOOPS PER PATTERN TYPE IN LOOP SETS 1 (`NEON = SVE`), 2 (`NEON > SVE`), AND 3 (`NEON < SVE`)

Pattern Type	Set 1	Set 2	Set 3
reduction	4	5	2
linear dependence testing	2	2	4
symbolics	2	1	1
scalar and array expansion	1	1	1
storage classes and equivalencing	1	4	—
call statements	1	1	—
data flow analysis	1	—	3
loop interchange	4	—	—
loop re-rolling	1	—	—
misc. control loops	—	6	4
induction variable recognition	—	4	3
control flow	—	2	—
index set splitting	—	1	—
jump in data access	—	1	—
node splitting	—	1	—
parameters	—	—	1
intrinsic functions	—	—	1
Sum	17	29	20

The two loops that are vectorized from the parameters and intrinsic functions categories perform better on SVE. While the difference for the loop with intrinsic functions is again 18 percentage points ($S_{NEON} = 1.61$, $S_{SVE} = 1.79$), it is significantly higher for the loop from the *parameters* category. Here the speedup increases from 6.06x for NEON to 8.45x for SVE. The code for this loop, `s431`, is shown in Listing 1.

Listing 1
LOOP PATTERN `s431`

```
int k1 = 1;
int k2 = 2;
int k = 2 * k1 - k2;

for (int i = 0; i < 32000; i++){
      array[i] = a[i + k] + b[i];
   }
}
```

Listing 2
LOOP PATTERN s125

```
int k = -1;

for (int i = 0; i < 256; i++){
   for (int j = 0; j < 256; j++){
      k++;
      res[k] = aa[i][j]
             + bb[i][j] * cc[i][j];
   }
}
```

Listing 3
NEON ASSEMBLY

```
.L3:
 ldr q2, [x4, x0]
 ldr q1, [x3, x0]
 ldr q0, [x1, x0]
 fmla v0.4s, v2.4s, v1.4s
 str q0, [x2, x0]
 add x0, x0, 16
 cmp x0, 1024
 bne .L3
```

Listing 4
SVE ASSEMBLY

```
.L3:
 ldlw z0.s, p0/z, [x3, x0, lsl 2]
 ldlw z1.s, p0/z, [x8, x0, lsl 2]
 ldlw z2.s, p0/z, [x4, x0, lsl 2]
 fmla z0.s, p1/m, z2.s, z1.s
 stlw z0.s, p0, [x1, x0, lsl 2]
 incw x0
 whilelo p0.s, x0, x2
 bne .L3
```

The difference in the generated assembly is the calculation of the iteration variable `i` and the test if all loop iterations have been executed. While the SVE version applies the vector length agnostic increment to `i` and subsequently utilizes its predication registers to decide if loop execution can be terminated, the NEON version needs one addition and two subtractions for the same calculation. Due to the short loop body —it consists of two loads, a floating point addition and a store—, the performance impact of the additional instructions is measurable. It must be noted, however, that the compiler should be able to produce more efficient code for the NEON platform, consequently reducing the difference in speedups between platforms.

B. Predication Overhead

To understand the performance impact of SVE's global predication of instructions, we analyzed code pattern `s125` in more detail. It belongs to the *induction variable recognition* pattern group and its code is presented in Listing 2. The generated assembly for NEON and SVE is shown in Listings 3 and 4, respectively. For both architectures, an identical code sequence of three load operations, a fused-multiply-add instruction and a store is generated. The difference between the two code snippets is that for SVE, all instructions are predicated, which can be seen by the additional `p` registers being passed as arguments to the instructions. Furthermore, there is no specific induction variable increment in SVE, but it is incremented by the hardware specific vector width. The loop is then executed until the lowest element of the predicate register is set to false, i.e. all elements have been processed. In terms of performance, the NEON based version achieves a speedup of 1.54x, while the SVE based version speeds up scalar code by a factor of 1.39x, i.e. 90% of NEON's performance. The overhead introduced by the general predication of registers can therefore be assumed to be around 10% when compared to state-of-the-art NEON code.

VI. PERFORMANCE OF VLA CODE

We performed a third experiment to understand the performance impact of VLA code for larger vector lengths and analyze how well performance scales with increasing vector lengths. For this purpose, we determined the speedups for 82 loops that could be auto-vectorized by the compiler for vector length specific and VLA code. We then calculated the geometric mean of the loops' individual speedups. Results are shown in Figure 3.

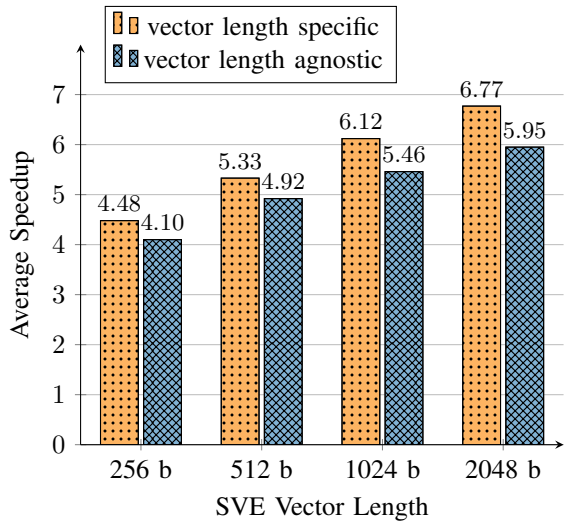

Fig. 3. Comparison of average speedups of 88 loops for VLA and vector length specific code built for varying vector lengths

Based on these numbers, two observations can be made:

- The compiler is able to produce higher performing code when the vector length is known.
- Speedups do not grow proportionally with vector length for neither vector length specific nor vector length agnostic code.

A. NEON as Fall-Back Solution

The first observation is that VLA code reaches about 90% of the performance of vector length specific code. When looking at individual loop speedups, performance is similar for the majority of codes. There are loops, however, where the compiler uses NEON code as a fall-back solution, i.e. it is not utilizing the full vector length. This applies to loops with instructions that might not be supported on all vector lengths and/or data types, such as horizontal adds, which add all elements across a vector. In this case, it is safe to fall back to 128 bit NEON instructions, since all vector lengths in SVE are multiples of 128 bit and it can be guaranteed that all

instructions will be supported by any hardware. The measured results in Figure 3 therefore show the average impact of the fall-back solution on overall performance when compared with vector length specific code that uses the full vector length. This performance impact is significantly higher for the affected individual loops. Consequently, when code portability is not an issue, a specific target architecture should be specified.

B. Memory Limitations for Large Vector Lengths

Figure 3 also shows that the average speedup does not scale with vector length. All loops in this experiment operate with 32 bit single precision floating point numbers. Therefore, the highest possible vectorization factor (VF), i.e. the highest number of vector elements, is given by $VF = \frac{vector\ length}{32bit}$. It also denotes the theoretical maximum speedup for compute-bound loops. Ideally, when doubling the vector length, the average speedup would double as well. However, this is not the case for the 88 SVE loops. For a vector length of 256 bit, an average speedup of 4.5 is achieved for vector length specific code (4.1 for VLA) despite an upper bound of eight. When doubling the vector length, however, the average speedup only increases by a factor of 1.18x (1.20x for VLA) and the return diminishes further when continuing to double vector lengths: when increasing the vector length from 1024 bit to 2048 bit, code is sped up by a factor of merely 1.10x (1.09 for VLA). This is due to code hitting memory limitations.

We therefore analyzed the memory limitations for large vector lengths in terms of L1 cache size. To understand its impact on code performance, we scaled the L1 cache size in the gem5 simulator and ran the benchmarks with different vector lengths. The L2 cache size was kept constant at a large value ($> 100MB$) to remove its impact from the experiment. In general, all benchmark loops operate on one or multiple data arrays of a constant size. For one dimensional arrays, this size is $32000 * 4B = 128kB$, while it is $256 * 256 * 4B = 256kB$ for two dimensional arrays. Hence a cache size of 1 MB would fit all array data into the cache for the majority of loops. The baseline of this experiment, however, is code performance for a L1 cache size of 32 kB, which is a common size in today's processors, even in the high performance computing market. We then quadrupled the cache size up to a size of 8 MB, keeping the cache line size constant, and ran the experiment for three different vector lengths. Average speedups are determined based on the each vector length's performance with a L1 cache size of 32 kB. The results are shown in Figure 4.

The experiment shows that increasing the cache size by a factor of four from 32 kB to 128 kB yields only limited speedups of 6 - 14 percentage points. For the smaller vector length of 128 b, scaling the cache size further improves performance up to a factor of 1.91x (8 MB cache size). The larger vector lengths benefit significantly more from the larger cache sizes, with maximum speedups of 2.69x (512 b vector, 8 MB cache size) and 5.65x (2048 b, 8 MB cache size). However, the return diminishes substantially. While performance improves greatly when moving from 128 kB to 512 kB to 2 MB, increasing the cache size to 8 MB increases the speedups by a mere 4% on average.

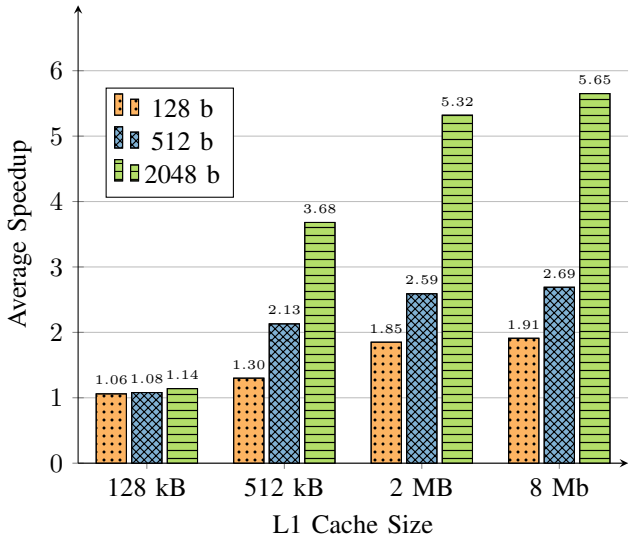

Fig. 4. L1 cache size scaling showing the average speedup compared to baseline cache size of 32 kB for increasing vector lengths

Overall, the experiment highlights the increased pressure that is added to the memory subsystem by the growing vector lengths. With large cache sizes, we see speedups up to a factor of 5.65x for large vectors, while performance increases by a factor of 1.91x for small vectors. It indicates that applications will be even more memory bound as of today, since higher speedups are possible with a more performant memory subsystem. To achieve these high speedups, however, an increase in L1 cache size of at least a factor of eight is needed for this benchmark. The use of large vectors therefore opens new challenges for memory design, and more research is required in this direction to utilize large vectors to their full potential.

VII. Conclusion

After several years of ever-increasing vector lengths and vector instruction sets, new ISAs are introduced for vector length agnostic programming and code generation. Based on the known technique of loop predication, the proposed ISAs allow optimizations for a hardware- independent instruction set, which is portable across platforms with varying vector widths. In this work, we analyzed the performance impact of VLA code compared to vector length specific code. For this purpose, we ran a loop benchmark in a gem5 SVE simulation environment and measured the obtained speedups. Our results show that the current processor model assumes a 10% overhead that is added by loop predication.

For large vector lengths, we found that the compiler is not able to utilize the full vector length for all code patterns, falling back to NEON instructions and thus missing performance improvements. Our experiment furthermore highlights the stress that is put on the memory subsystem by growing vector lengths. It will therefore become critical to understand

and satisfy memory requirements that such large vector lengths impose on the overall system, as it will not be possible to capitalize on large vector lengths otherwise.

REFERENCES

[1] B. Juurlink, D. Tcheressiz, S. Vassiliadis, and H. A. Wijshoff, "Implementation and Evaluation of the Complex Streamed Instruction Set," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 73–82, IEEE, 2001.

[2] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, *et al.*, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[3] "RISC-V Vector Extension Proposal." https://riscv.org/wp-content/uploads/2015/06/riscv-vector-workshop-june2015.pdf. Accessed: 2019-04-05.

[4] T. Yoshida, "Fujitsu high performance CPU for the Post-K Computer," in *Hot Chips 30 Symposium (HCS), Series Hot Chips*, vol. 18, 2018.

[5] A. Rico, J. A. Joao, C. Adeniyi-Jones, and E. Van Hensbergen, "ARM HPC Ecosystem and the Reemergence of Vectors," in *Proceedings of the Computing Frontiers Conference*, pp. 329–334, ACM, 2017.

[6] Y. Kodama, T. Odajima, M. Matsuda, M. Tsuji, J. Lee, and M. Sato, "Preliminary Performance Evaluation of Application Kernels Using ARM SVE with Multiple Vector Lengths," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 677–684, IEEE, 2017.

[7] 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)*, pp. 623–628, IEEE, 2018.

[8] A. Armejach Sanosa, H. Caminal Pallarés, J. M. Cebrián González, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero Cortés, M. Casas, and M. Moreto Planas, "Stencil codes on a vector length agnostic architecture," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques: Limassol, Cyprus, November 01-04, 2018*, pp. 1–12, Association for Computing Machinery (ACM), 2018.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[10] "gem5 SVE Branch." https://gem5.googlesource.com/arm/gem5/+/sve/beta1. Accessed: 2019-04-05.

[11] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An Evaluation of Vectorizing Compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pp. 372–382, IEEE Computer Society, 2011.

[12] "TSVC Benchmark Sources." http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz. Accessed: 2019-05-06.

[13] A. Pohl, B. Cosenza, and B. Juurlink, "Control Flow Vectorization for ARM NEON," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pp. 66–75, ACM, 2018.