Stencil Codes on a Vector Length Agnostic Architecture

Adrià Armejach
Barcelona Supercomputing Center
Universitat Politécnica de Catalunya
adria.armejach@bsc.es

Helena Caminal*
Cornell University
hc922@cornell.edu

Juan M. Cebrian
Barcelona Supercomputing Center
juan.cebrian@bsc.es

Rekai González-Alberquilla
Arm, Cambridge, UK
rekai.gonzalezalberquilla@arm.com

Chris Adeniyi-Jones
Arm, Cambridge, UK
chris.adeniyi-jones@arm.com

Mateo Valero
Barcelona Supercomputing Center
mateo.valero@bsc.es

Marc Casas
Barcelona Supercomputing Center
marc.casas@bsc.es

Miquel Moretó
Barcelona Supercomputing Center
miquel.moreto@bsc.es

ABSTRACT

Data-level parallelism is frequently ignored or underutilized. Achieved through vector/SIMD capabilities, it can provide substantial performance improvements on top of widely used techniques such as thread-level parallelism. However, manual vectorization is a tedious and costly process that needs to be repeated for each specific instruction set or register size. In addition, automatic compiler vectorization is susceptible to code complexity, and usually limited due to data and control dependencies. To address some these issues, Arm recently released a new vector ISA, the Scalable Vector Extension (SVE), which is Vector-Length Agnostic (VLA). VLA enables the generation of binary files that run regardless of the physical vector register length.

In this paper we leverage the main characteristics of SVE to implement and optimize stencil computations, ubiquitous in scientific computing. We show that SVE enables easy deployment of textbook optimizations like loop unrolling, loop fusion, load trading or data reuse. Our detailed simulations using vector lengths ranging from 128 to 2,048 bits show that these optimizations can lead to performance improvements over straight-forward vectorized code of up to 56.6% for 2,048 bit vectors. In addition, we show that certain optimizations can hurt performance due to a reduction in arithmetic intensity, and provide insight useful for compiler optimizers.

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; • **Theory of computation** → *Parallel computing models*;

KEYWORDS

data-level parallelism, scalable vector extension, vector length agnostic, stencil computations

*Work done while at the Barcelona Supercomputing Center and during an internship at Arm Reseach, Cambridge.

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM Reference Format:
Adrià Armejach, Helena Caminal, Juan M. Cebrian, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. 2018. Stencil Codes on a Vector Length Agnostic Architecture. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3243176.3243192

1 INTRODUCTION

One of the major constraints when designing high-performance parallel architectures is energy efficiency. Achieving good energy efficiency requires scalable architectures, that not only exploit thread-level parallelism, but also parallelism at instruction and data level via vector/SIMD instructions. In fact, under area constraints, the usage of vector architectures clearly outperforms chip multi-processors (CMPs) in flops per watt [10]. Power savings come mainly from reducing front-end pressure, since a single instruction encodes multiple operations, and from the reduction in execution time achieved by vectorized code.

While multi-threaded implementations have become the *de facto* solution to software design, developers commonly neglect the usage of vector capabilities, *e.g.* SIMD. The underutilization of vector/SIMD features usually lies in the limited capabilities of automatic vectorization and the complexity of handcrafted vectorization. The quality of automatically generated vector code is closely related to the complexity of the code being vectorized, and often limited by intra-vector dependencies. In addition, manually-vectorized codes need to be maintained to support new features or vector lengths.

To address these issues, Arm recently released a new vector Instruction Set Architecture (ISA), the Scalable Vector Extension (SVE) [22], which extends the Armv8-A ISA. SVE targets complex parallel codes that go beyond the workloads that typically run on embedded or mobile systems. In particular, one of the areas that is expected to be more impacted by SVE is HPC. Some of the most ambitious research projects aiming to build exascale systems are based on Arm architectures that will feature SVE [28]. Rather than having a fixed vector length (VL), SVE gives flexibility to hardware designers to implement their vector length of choice from 128 to 2048 bits. The *Vector-Length Agnostic (VLA)* programming model adjusts dynamically to the available VL.

SVE improves manual, automatic and user-directed (pragma-based) code generation by easing the vectorization process to both compilers and developers. In particular, SVE has been tested to improve vectorized code generated on applications from different fields of research, including: sorting, dense and sparse linear algebra, n-body methods, LU-decomposition, graph traversal, stencil codes, *etc.* [22]. Stencil codes are of particular interest, since they are the basis for HPC applications targeting problems from many scientific domains such as fluid dynamics, structural mechanics, and image processing. Stencils are iterative kernels that operate over N-dimensional data structures with a fixed computational pattern. These kernels are commonplace in finite-difference methods used to solve large-scale and highly-dimensional partial differential equations (PDEs).

The implementation of stencil computations to efficiently exploit the resources available in the system is a difficult task that has been previously studied [4, 30]. Stencils are typically memory bound, which is also a challenge for vector architectures [7]. VLA facilitates the deployment of well-known code optimizations that significantly benefit from the semantics offered by SVE. By using SVE the loop control flow is driven by predicates, therefore, porting the *while(cond)-end* control statement is straight-forward and automatically applies VLA to our baseline (non-optimized) codes.

This paper makes the following contributions:

- We present a novel analysis on how the Arm SVE vector ISA can be used to increase the performance of a highly relevant group of numerical kernels - stencil computations. We describe how different optimizations such as loop unrolling, loop fusion, data reuse, and load trading can easily be implemented using the SVE ISA. We have implemented the scalar baseline and all the SVE-enabled optimizations on 7-point and 27-point stencils using hand-coded assembly to ensure controlled and optimized code generation.
- A comprehensive performance evaluation based on detailed architectural simulations that faithfully model the SVE architecture. We study the impact of different vector lengths ranging from 128 to 2048 bits, as well as the performance impact of each of the different code optimizations. We find that loop fusion unlocks the largest performance improvements, up to 56.6% for 2048-bit vectors.
- We compare performance of out-of-order and in-order cores using different vector lengths and find that out-of-order capabilities offer significant performance advantages. This is because in vectorized codes the time spent in arithmetic operations decreases and higher memory contention is usually present, leading to stalls in in-order pipelines.
- We report our experience vectorizing and optimizing stencil codes using SVE, which serves a two-fold purpose: (i) detail how VLA and per-lane predication aid programming certain optimizations, providing a recipe for manual vectorization; and (ii) useful insight to train automatic vectorization tools. In addition, we provide guidelines on the appropriate vector lengths to employ depending on workload characteristics.

2 RELATED WORK

Stencil Codes: Partial differential equations (PDEs) are used for solving large-scale and high-dimensional problems using finite-difference methods (FDM). PDEs are the base to provide numerical approximations to complex computational problems from science and engineering [9, 14, 18, 26, 31]. FDMs use an N-dimensional array of elements in which each element is updated every time-step based on the weighted contribution of neighboring elements (the *stencil*).

The most commonly used sets of neighboring elements are called *Von Neumann* and *Moore* neighborhoods [25]. Both the Von Neumann and Moore neighborhoods of an element x and radius d are formed by all the elements, y, such that $dist(x, y) <= d$. The difference strives in how the distance is calculated. For Von Neumann, the distance between two elements is the addition of the distances in each dimension. For Moore, the distance between two elements is the max of the distances in each dimension. In this paper, we consider two neighborhoods, *7-point* and *27-point* stencil, which are the names commonly used to designate, respectively, Von Neumann and Moore neighborhood of radius 1.

Vector/SIMD Architectures: Vector architectures have been present almost since the beginning of the history of parallel computing [6]. The first vector supercomputers, TI-ASC and STAR-100, were released in 1970 and consisted of a powerful vector unit that was served by the scalar unit, which comparatively had poor performance. Both were memory-to-memory machines, equipped with a very high bandwidth memory system. As opposed to present architectures, specially RISC-based ones, the instruction sets of these machines had so much semantic content that it would be very difficult for a compiler to auto-vectorize programs. Nevertheless, some of the features we see in today's vector architectures were already present in TI-ASC and STAR-100. Bit masks to implement conditional operations were one of the features implemented in these first machines. Current architectures offer similar functionalities: AVX-512 [3] has masked operations and gather/scatter memory operations, and PowerPC AltiVec [8] implements the *compare* operation to create field masks.

There are also recent proposals from academia, such as the Hwacha Design [16], which extends the RISC-V ISA [27]. Its philosophy is based on traditional vector architectures, similar to Cray1 [21], allowing to have a variable vector length configured through the *vector length register*. The key difference is that vector instructions execute in their own *vector fetch block*, while the scalar control processor continues doing useful work independently. Additionally, the instruction set has predicate registers to mask vector operations. SVE shares with Hwacha a VLA approach without the needing a specific vector length register

The paradigm implemented in GPUs, *single instruction multiple thread* (SIMT) has similar functionalities, as it also permits handling divergent threads by predicating operations. In general, codes targeting GPUs are independent of micro-architectural parameters such as warp size, which could relate to the VLA feature of SVE.

Stencils on Vector/SIMD Architectures: Naive implementations of stencil computations usually achieve only a fraction of the system's peak performance [20]. Additionally, stencils usually suffer

```
for (t←1; t≤T; ++t)
  for (i←1; i≤I; ++i)
    for (j←1; j≤J; ++j)
      for (k←1; k≤K; k+=VL)
        B[i][j][k] ← (A[i][j][k] +
                A[i][j][k+1] + A[i][j][k−1] +
                A[i][j+1][k] + A[i][j−1][k] +
                A[i+1][j][k] + A[i−1][j][k])/7
  A ← B
```

Figure 1: Pseudo-code of a 3D Jacobian method (T time-steps) with *computational* and *copy* loops.

from a high cache miss-rate, and their performance drops drastically once input sizes exceed the size of the last level cache. Memory-boundedness of a stencil depends on the arithmetic intensity of the computations done over the neighbors (computations per loaded byte). Many optimizations try to improve data locality, data reuse and other performance-critical factors of stencil computations [2, 11–13, 15, 23]. There are also stencil-specific optimization frameworks [30] and compiler support [24] to ease the optimization process for developers.

Optimization of stencil computations via SIMD instructions is also a common approach [5, 19], and the usage of GPUs has also been considered [17]. Wider register sizes for CPU ISAs clearly offer some advantages in terms of data migration reduction and ease the programming effort. However, programming for GPUs is complex and error-prone. Programming models for GPUs, like CUDA and OpenCL, require programmers with expertise on the target micro-architecture. The vector length agnosticism of SVE is a key feature to match the acceleration capabilities that GPUs offer, exploiting long SIMD units without the need for generating binaries for specific lengths.

Scalable Vector Extension: SVE is Arm's response to the increasing needs of energy efficient computing systems in the HPC domain. One of the highlights of SVE is *Vector-Length Agnosticism (VLA)*. VLA enables the generation of binary files that run independently of the underlying physical vector register length. The immediate consequence of VLA is performance portability, exploiting wider registers in high-end implementations of the architecture with the same binary. Additionally, VLA also comes with the benefit of having an efficient utilization of instruction *opcodes* throughout the wide range of vector lengths: the same opcode is used for a given instruction independently of the vector length. In fact, only one sixteenth of the available opcodes are needed to encode SVE instructions. SVE encoding fits in a 28-bit region (Armv8-A is encoded in 32 bits), leaving room for future extensions. SVE supports vector lengths ranging from 128 to 2048 bits in multiples of 128.

VLA is achieved using a predicate register driven loop control flow. Predicate registers are constructed and/or modified when the loop condition is checked. This functionality results in accomplishing two tasks with a single instruction: a) setting condition flags, and b) preparing predicate registers to be used as masks for instructions in the loop body. As a consequence, there is no need for treating loop prologues and epilogues aside of the main loop, thus allowing the vectorization of variable trip-count loops. It also helps preventing faults due to uninitialized data or accesses to out-of-bound addresses. Other remarkable features of SVE include: (i) serialized reductions that allow SVE to ensure the same rounding behavior as scalar codes and (ii) vector partitioning, that enables speculative vector loads, among other uses.

3 STRATEGIES TO OPTIMIZE STENCIL CODES WITH SVE

Stencil optimization via SIMD instructions has been widely researched in the past [5, 19, 29]. From the many strategies available in the literature, we have cherry-picked those that are most used and that could potentially be challenging for the ISA. Selected strategies are thoroughly described in this section in the Armv8-A ISA context. To the best of our knowledge this is the first study that leverages specific SVE properties to optimize stencil computations. Figure 1 shows the pseudo-code of a 3D Jacobi method, which uses a 7-point stencil scheme to update the values of the elements composing the 3D array A. Since the Jacobi method uses the values computed during iteration $(i-1, j, k)$ to obtain the values of iteration (i, j, k), it is not possible in general to override the elements of A as they may be used in subsequent computations. This is the reason why the algorithm displayed in Figure 1 contains two main loops within each time-step t: the *computational loop* where we store the results to array B, and the *copy loop* where we copy back the previous results to the original array A. We are aware of the alternative solutions that prevent having a copy loop, and cover it in detail in Section 3.3.

We adhere to the following conventions:

- The memory layout of arrays is row-major order.
- Vectorization is applied on the k axis unless otherwise stated
- **A[i][j][k]** represents the tuple of elements, consecutive along the k axis, $< A[i][j][k], \ldots, A[i][j][k+VL-1] >$, that fit in a vector.
- We overload the meaning of i, j and k to both the index variable of the for-loops, as well as the value of the index during a given iteration. k^+ denotes the value of the index for the next iteration.
- We use the notation (i, j, k) to denote the iteration in which the index variables of the loops, from outermost to innermost, are i, j and k.
- We assume there is memory allocation for the *halo* of the stencil, that it is initialized and never updated.
- Stencils are iterative algorithms that compute until a certain convergence criteria is matched. As such, they allow a certain degree of error in the computations. We can take advantage of that property and allow associativity of floating point computations.

The stencils employ double precision floating point elements, therefore we can operate on 2, 4, 8, 16 and 32 elements using vector lengths from 128 to 2048 bits.

3.1 Baseline Codes

The simplest way to vectorize a stencil code is to apply vector instructions to the innermost loop. Vector instructions compute 'vector-length (VL)' data elements simultaneously, so the innermost loop index variable will now be incremented in VL time-steps. Starting from the scalar code, each instruction is replaced by their equivalent vector version. In addition, we typically need to operate

```
mov   x1, #1                  // k:=1
loop:
cmp   x1, z_size
b.eq  end
ldr   d4, [A, x1, lsl #3] //A[i][j][k]
add   x2, x1, offs_north
ldr   d8, [A, x2, lsl #3] //A[i][j][k+1]
fadd  d4, d4, d8
sub   x2, x1, offs_north
ldr   d8, [A, x2, lsl #3] //A[i][j][k-1]
fadd  d4, d4, d8
add   x2, x1, offs_front
ldr   d8, [A, x2, lsl #3] //A[i][j+1][k]
fadd  d4, d4, d8
sub   x2, x1, offs_front
ldr   d8, [A, x2, lsl #3] //A[i][j-1][k]
fadd  d4, d4, d8
add   x2, x1, offs_east
ldr   d8, [A, x2, lsl #3] //A[i+1][j][k]
fadd  d4, d4, d8
sub   x2, x1, offs_east
ldr   d8, [A, x2, lsl #3] //A[i-1][j][k]
fadd  d4, d4, d8
fmul  d4, d4, constant
str   d4, [B, x1, lsl #3] //B[i][j][k]
add   x1, x1, #1
b     loop
end:
```

(a) Scalar version

```
mov   x1, #1
sub   x7, z_size, z_size mod 2
loop:
cmp   x1, x7
b.eq  scalar
ld1   v4, [A, x1, lsl, #3]
add   x2, x1, offs_north
ld1   v8, [A, x2, lsl, #3]
fadd  v4.2d, v4.2d, v8.2d
sub   x2, x1, offs_north
ld1   v8, [A, x2, lsl, #3]
fadd  v4.2d, v4.2d, v8.2d
add   x2, x1, offs_front
ld1   v8, [A, x2, lsl, #3]
fadd  v4.2d, v4.2d, v8.2d
sub   x2, x1, offs_front
ld1   v8, [A, x2, lsl, #3]
fadd  v4.2d, v4.2d, v8.2d
add   x2, x1, offs_east
ld1   v8, [A, x2, lsl, #3]
fadd  v4.2d, v4.2d, v8.2d
sub   x2, x1, offs_east
ld1   v8, [A, x2, lsl, #3]
fadd  v4.2d, v4.2d, v8.2d
fmul  v4.2d, v4.2d, constant
str   v4, [B, x1, lsl, #3]
add   x1, x1, #2
b     loop
scalar: // Last iteration
```

(b) NEON version (128-bit SIMD)

```
mov   x1, #1
loop:
whilelt p0.d, x1, z_size
b.eq  end
ld1d  z4.d, p0/z, [A, x1, lsl #3]
add   x2, x1, offs_north
ld1d  z8.d, p0/z, [A, x2, lsl #3]
fadd  z4.d, z4.d, z8.d
sub   x2, x1, offs_north
ld1d  z8.d, p0/z, [A, x2, lsl #3]
fadd  z4.d, z4.d, z8.d
add   x2, x1, offs_front
ld1d  z8.d, p0/z, [A, x2, lsl #3]
fadd  z4.d, z4.d, z8.d
sub   x2, x1, offs_front
ld1d  z8.d, p0/z, [A, x2, lsl #3]
fadd  z4.d, z4.d, z8.d
add   x2, x1, offs_east
ld1d  z8.d, p0/z, [A, x2, lsl #3]
fadd  z4.d, z4.d, z8.d
sub   x2, x1, offs_east
ld1d  z8.d, p0/z, [A, x2, lsl #3]
fadd  z4.d, z4.d, z8.d
fmul  z4.d, p0/m, z4.d, constant
st1d  z4.d, p0, [B, x1, lsl #3]
incp  x1, p0.d
b     loop
end:
```

(c) SVE version (VLA)

Figure 2: Code fragment of the 7-point stencil computation loop - only showing the innermost loop (k).

on the prologue or epilogue aside of the main loop to compute the remaining values, that is, the initial/final values that do not completely fill in the vector register. SVE's per-lane predication enables treating prologues and epilogues within the main loop. Figure 2a shows the scalar baseline code of the innermost loop of a 7-point stencil. Its NEON and SVE counterparts are shown in Figure 2b and 2c, respectively. As a reminder, for data manipulation instructions (i.e., additions) Arm assembly syntax places the destination operand immediately after the mnemonic (similarly to Intel and opposite to AT&T assembly syntax).

In the scalar code, the for-loop (*cmp–b.eq–b* structure) checks *x1* and *z_size* to continue execution (elements in k axis, without the halo). If true, the memory addresses of the 6 neighboring elements are computed and data is loaded into registers. Then, the average of the 6 neighbors is computed and stored to array *B*.

On the other hand, NEON code needs to check if the number of iterations is a multiple of the number of elements it can fit into the vector register. In this example NEON can store 128 bits, since we use double precision floating point it can fit 2 elements. If it is not multiple of 2, we need to do one iteration less of the vectorized loop, and then jump to the *scalar* label to compute the last element - the scalar code would be identical to the body loop in Figure 2a.

Finally, SVE uses the *whilelt* instruction to iterate over the for-loop. This instruction allows operating over the loop independently of the VL and the number of iterations. *whilelt* constructs the predicate register, *p0*, by evaluating the condition *lt* (less than) on the content of registers *x1* and *z_size*. *p0* can be seen as a mask that tells the architecture if an specific vector lane is enabled ('1'), or disabled ('0'). Instructions *ld1d*, *st1d*, *fadd* and *fmul* are the vector equivalent of the scalar instructions *ldr*, *str*, *fadd* and *fmul*, respectively. These new instructions operate on vector registers (*z4.d* and *z8.d*), which contain a set of double precision floating point elements. The *incp* instruction increments the content of register *x1* by the number of active elements in *p0*. SVE executes the code inside the loop z_size/VL times, with an additional predicated iteration if ($z\_size \bmod VL$)$\neq 0$.

We want to outline that although mask operations exist in other current SIMD architectures (such as Intel AVX-512 [3]), SVE's per-lane predication is already integrated into the control flow. Predicate registers drive loop control flow, reducing loop management overhead and controlling both vector and scalar instructions.

3.2 Loop Unrolling

One way to improve performance is to unroll the outer loops, as the innermost loop is vectorized. By doing so, we reuse loaded data for more than one iteration, thus reducing the pressure on the memory subsystem.

For instance, each computation on the 7-point stencil requires 7 $load/element$, or 7 $Vload/VLelements$ in vectorized code. Unrolling one iteration on the *j* dimension increases the number of required neighbors to 12, but two elements would be computed in the process, so the ratio goes down to $12/2 = 6$ $load/element$. Table 1 shows the ratio variation for several configurations as we unroll the outer loops *i* and *j*, *N* and *M* times, respectively. The right-most column

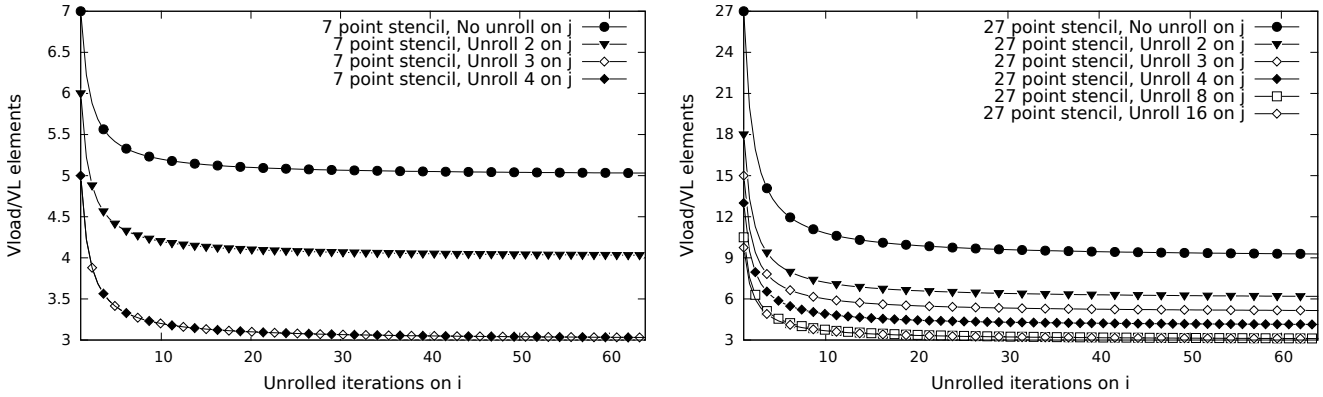


Figure 3: Vector loads-per-*VL* elements ratio when unrolling *i* and *j* for a 7-point (left) and a 27-point (right) stencil.

Table 1: Vector loads-per-VL elements ratio.

	Unroll factor	$\frac{\text{Vloads}}{\text{Iteration}}$	$\frac{\text{VLelements}}{\text{Iteration}}$	$\frac{\text{Vload}}{\text{VLelements}}$
7-point	Baseline	7	1	7
	j by 2	12	2	6
	j by 3	17	3	5.67
	i by 2 & j by 3	28	6	4.67
	i by N & j by M	$2M + 2N + 3NM$	NM	$\frac{2}{N} + \frac{2}{M} + 3$
27-point	Baseline	27	1	27
	j by 2	36	2	18
	j by 3	45	3	15
	i by 2 & j by 3	60	6	10
	i by N & j by M	$3(2 + M)(2 + N)$	NM	$\frac{6}{N} + \frac{6}{M} + \frac{12}{NM} + 3$

of this Table ($Vload/VLelements$) is computed as the division of the left-most column ($Vload/Iteration$) and the middle column ($VLelements/Iteration$).

The progress of this ratio between neighborhood shapes follows a curve with an asymptotic behavior (Figure 3), becoming almost flat after few unrolled iterations (< 10) on both of the outer dimensions *i* and *j*. The studied 27-point stencil is more sensitive to unrolling, specially in the first unrolled iterations, explained by the multiplicative factor ($12/NM$, Table 1). For higher degrees of unrolling on both dimensions there is little variation on the number of loads, which matches the number of dimensions we are operating on, in this case 3. As a general conclusion, unrolling on both *i* and *j* dimensions at the same time provides the largest gains.

On the practical side, unrolling is limited by the amount of architectural resources. There are different ways of implementing an unrolled code, nevertheless, to update the value for more than one element on the same iteration, we need at least one register per element to accumulate the result. Therefore, on a 3D space for both a 7 and a 27-point stencil unrolling N and M iterations on the *i* and *j* dimensions, respectively, we need at least $N \times M$ registers. Also, we need at least one additional register to operate on the data before accumulating it. In addition, we should save some registers to build optimizations on top of the unrolling. SVE supports up to 32 vector registers. As an example, if we chose to unroll symmetrically a 27-point stencil to a level of $N = M = 5$ iterations, 25 registers are required to live through the iteration to store the result.

As loop unrolling is based on a replication of the loop body and a non-unitary advance of the innermost loop index, the same SVE benefits as the baseline code apply to this version. Control flow is done identically given that unrolling is done to dimensions other than the vectorized one (**k**), thus, VLA is naturally maintained.

3.3 Loop Fusion

Stencil codes have cyclic data dependencies between elements in sequential time-steps. An element needs the former value of its neighbors to compute its new value. In turn, neighbors need the element value to calculate their new value. Implementations of Jacobi usually avoid this problem by writing the elements of the next time-step to a temporal array *B*, and once the calculation is complete, copy back to the original array *A* (Figure 1). Another possible implementation is using two arrays which, alternatively behave as *previous* and *current* by switching the pointer values every time-step. Both implementations require the use of an auxiliary array of the same size as the original.

Exploiting parallelism without using multiple copies of the data requires a detailed study of the dependencies between elements. Figure 4-top shows a representation of the elements used in a single iteration of a 7-point stencil in memory, row major order. Note that Figure 4 shows the accesses for the scalar case and that in a vectorized code we would have the same accesses with their consecutive right $VL - 1$ neighbors. In each iteration, and for this linear memory layout, the elements required for the current computations will be their consecutive right neighbors. Since we are working with stencils of order 1, that is, only use their closest neighbors for their computations, the last element that requires of element $\mathbf{A[i][j][k]}$ is $\mathbf{A[i + 1][j][k]}$. A solution to overcome this dependency is to store temporarily the values required for the next iteration of the outermost loop, $i + 1$, until we reach its last consumer. Then, we update this value in the original array $\mathbf{A[i][j][k]}$. A pseudo-code of this optimization is shown in Figure 5, where *tmp* is the 2-D temporary array where we store the elements between dependencies. Note that the computation is first stored to an SVE register ($z_i.d$) before

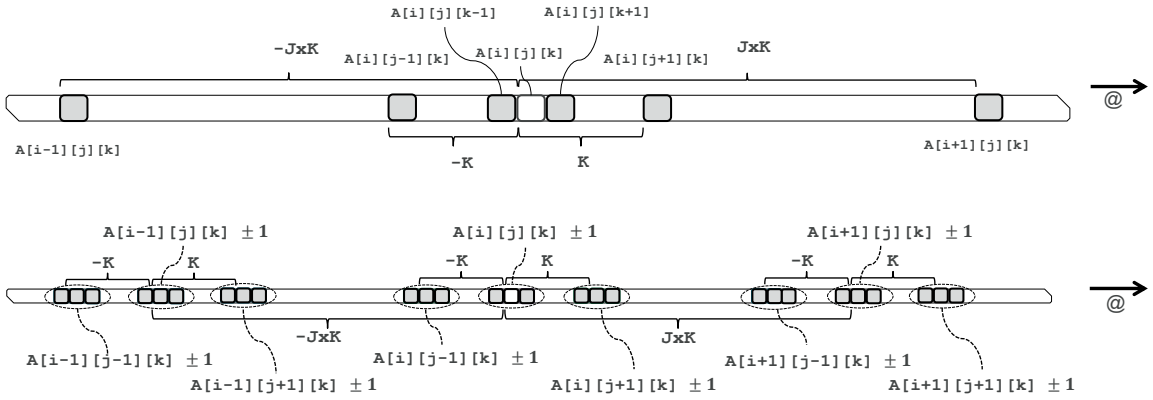


Figure 4: Linear representation of the accessed (loaded) elements in iteration (i, j, k) on array $A[I][J][K]$ in the *computation loop* on a 7-point (top) and a 27-point (bottom) stencils.

```
for (t←1; t≤T; ++t)
  for (i←1; i≤I; ++i)
    for (j←1; j≤J; ++j)
      for (k←1; k≤K; k+=VL)
        z_i.d← (A[i][j][k] +
                A[i][j][k+1] + A[i][j][k−1] +
                A[i][j+1][k] + A[i][j−1][k] +
                A[i+1][j][k] + A[i−1][j][k])/7
        A[i−1][j][k] ← tmp[j][k]
        tmp[j][k] ← z_i.d
```

Figure 5: Pseudo-code of loop fusion (*computation* and *copy*) on a 3D Jacobian method, T time-steps.

we update the value of its last dependency **A[i − 1][j][k]** to the original array and then store it to the temporal array (**tmp[j][k]**). This solution only requires a 2-dimensional array in addition to the original 3D array. In general, this implementation requires a $d-1$-dimension array. This represents a K times reduction in storage compared to naive implementations, where K is the size of the *last* dimension. In case of a 27-point stencil, the last consumer for element **A[i][j][k]** is element **A[i + 1][j + 1][k + 1]**, as shown in Figure 4-bottom. That requires storing $JxK + K + 1$ elements to honor the dependencies.

In both neighborhood shapes, we are reducing the memory footprint of the stencil computation. In the case of the 7-point stencil, we are only using one d dimensional array and one $d-1$ dimensional array, compared to the original code: two d-dimensional arrays. As a side effect, this optimization also improves the locality of the memory accesses.

This optimization is mainly a re-organization of the code that results in a single 3D loop control. As opposed to the previous optimization, now with a single loop management (*whilelt–b.eq–incp–b* structure) we are able to control all vector and scalar instructions. Moreover, if the loop body grows to support further optimizations, loop control overhead will be even less significant.

3.4 Load Trading

Performance of stencil codes is typically limited by memory bandwidth and latency. Thus, trading memory instructions for register-level computations may improve performance. In the 7-point stencil, there is an inter-iteration reuse of blocks when using the current memory layout, depicted in Figure 4-top. For iteration (i, j, k), the elements of vector **A[i][j][k]** are stored in consecutive memory locations. As a consequence, given a tuple of elements starting at address **A[i][j][k − 1]**, we can build tuples starting at **A[i][j][k]** (and **A[i][j][k + 1]**) by removing the first (and second) elements from the tuple, and then concatenating with the first (and second) elements from the tuple starting at **A[i][j][k − 1 + VL]**. The 27-point stencil (Figure 4-bottom) brings even more opportunity for improvement. With a layout of nine blocks of three adjacent neighbors in memory, loads can be reduced by $\frac{2}{3}$.

The approach we use to minimize memory instructions in the 7-point stencil is loading contiguous non-overlapping blocks of data, that is, the tuples that start at memory locations: **A[i][j][k − 1]** and **A[i][j][k − 1 + VL]**. Then we combine them properly to obtain **A[i][j][k]** and **A[i][j][k + 1]**. It is worth pointing out that in the next iteration, k^+ will equal $k + VL$. Therefore, **A[i][j][k$^+$ − 1]** is the same as **A[i][j][k − 1 + VL]**. Storing the vector in a register and carrying it over one iteration reduces load instructions from 7 to 5.

SVE provides several instructions that enable the reconstruction of a block given its contiguous neighbors. After considering the different options, we constructed our solution using the instruction *splice*. This instruction takes two vector registers and a predicate register. It constructs the destination register taking from the first source vector register the first to last active elements in the predicate register, and then filling the remaining with the lower elements of the second source. Note that the predicate register is constant through the execution of the stencil, thus, it can be constructed only once, at the beginning of the execution. The predicates used to reconstruct **A[i][j][k]** and **A[i][j][k + 1]** have the form of p_k = (1..110) and p_{k+1} = (1..100), respectively, being the right-most the least significant bit. The process to obtain the predicates consists of two instructions: *ptrue*, which activates as many elements as indicated, starting from the least significant bit (1 for p_k and 2 for

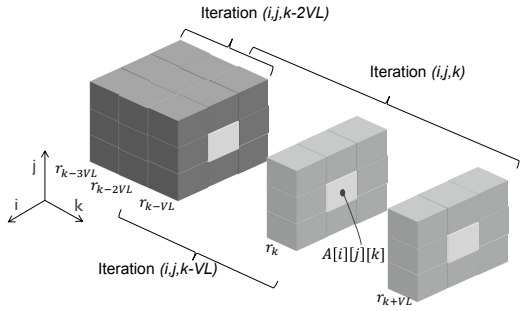


Figure 6: Elements (and partial results $r_{<iter.>}$) used in three consecutive iterations of the 27-point stencil, each cube representing a VL array of elements in the k dimension.

p_{k+1}) and *not* inverts each of the predicate bits. Note that these predicates would work for any architectural VL.

Other architectures offer similar instructions. As an example, AVX-512 [3] offers *valignq* which accepts also two SIMD 512-bit registers and a scalar value that indicates how many 64-bit elements to shift after concatenating the source vectors. Although Intel's instruction does not enforce the building of any mask register, it offers less flexibility in terms of functionality. For this special case SVE's predicate registers were only performing 64-bit element shift movements. Nevertheless, more complex movements could be made by constructing the predicates differently using *splice*.

3.5 Data Reuse

Given the symmetry of the neighborhood elements, we can reuse loaded data and partial computations across different iterations. The proposed optimization is only tested for the 27-point stencil. This optimization is implemented on top of the baseline, without any interaction with any other previous optimization.

Figure 6 shows the data usage in three contiguous iterations $(\mathbf{i}, \mathbf{j}, \mathbf{k} - \mathbf{2VL})$, $(\mathbf{i}, \mathbf{j}, \mathbf{k} - \mathbf{VL})$ and $(\mathbf{i}, \mathbf{j}, \mathbf{k})$. In each iteration, we only need to load 9×VL new elements to 9 SVE registers and add them in a SIMD manner to obtain a partial result, stored to an SVE register. This last partial result for element $\mathbf{A[i][j][k]}$ is defined as r_{k+VL}. By adding up r_{k+VL} to r_k and r_{k-VL}, obtained similarly in the two previous iterations, we compute the final result for this iteration. Finally, we move r_k to the register storing r_{k-VL} (and r_{k+VL} to r_k) to reuse the two newest partial results and get rid of the oldest.

This optimization allows to reduce the body of the loop significantly, replacing memory accesses and arithmetic operations by register movements. Additionally, only 4 SVE registers need to be alive simultaneously. This allows to apply optimizations on top of data reuse since it barely affects the architectural registers limitation. Data reuse could be done in any of the three dimensions, applying it to the vectorized dimension (**k**) is enabled by the use of *incp* that predicates the loading of the 9×VL new elements. This is clearly interesting in the case of having sizes of dimension **k** not multiple of VL, resulting in a classical loop tail for other ISAs. Additionally, in irregular grids where the borders in the **k** dimension are a function of the other dimensions, **i** and **j**, could benefit from VLA. In that case, the border condition can be dynamically computed to produce predicated registers for each **i** and **j** coordinates.

Table 2: Parameters for full-system simulations.

Processor size	4 cores.
Cores	*out-of-order:* 3-wide issue/retire, 40-entry instruction queue, 128-entry ROB, 32 LDQ + 32 STQ, 2GHz *in-order:* 2-wide issue, 5-entry store buffer, 2GHz
L1 I/D Caches	*out-of-order:* L1I: 48KB, 3-way, 2 cycle, 2 ports, 8MSHRs LID: 32KB, 2-way, 2 cycle, 2 ports, 16MSHRs *in-order:* L1I: 32KB, 2-way, 2 cycle, 1 port, 8MSHRs LID: same as out-of-order but with 1 port
Last-level Cache	4MB, 16-way, 64B lines, 8 banks, 64MSHRs Data bank access latency of 9 cycles.
NoC	Coherent crossbar, 128-bit wide, 2 cycles
Main Memory	4 DDR4-2400 channels, 2 ranks/channel, 16 banks/rank, 8KB row-buffer 128-entry write and 64-entry read buffers per channel 75GB/s peak bandwidth. Bank conflicts and queuing delays modeled

4 METHODOLOGY

We implemented the SVE baseline and subsequent optimizations over a well-known stencil code found in the Mantevo *miniapps* suite - *miniAMR*. This miniapp offers both Von Neumann and a Moore neighborhood shapes with a radius of 1 on a $3D$ space. The computation over the *stencil* is an average of all the neighbors. Reported performance and statistics in our evaluation are obtained measuring the entire region of interest, which includes the main stencil computation routine called *stencil_calc* and an additional routine that performs refinement. We focused our vectorization efforts only on the *stencil_calc* routine. The first step was to manually write an Armv8-A assembly version of the original scalar code of *stencil_calc*, both for 7-point and 27-point stencils. Using a manually written scalar baseline ensures fairness when comparing code versions and prevents the compiler from affecting the results. The *base* SVE-vectorized versions and subsequent optimizations (i.e.,*loop unrolling*, *loop fusion*, *load trading* and *data reuse*) are also manually written in assembly.

The input parameters used in the experiments are as follows: stencil type of 7 or 27 points, a single object (sphere) with no bounce ('0'), initially with its center at a position (x, y, z) of $(-1.1, -1.1, -1.1)$, a radius of 1.5 and a speed $(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{z})$ of $(0.03, 0.03, 0.03)$. This object has a null change rate of size: $(0.0, 0.0, 0.0)$. We simulate with 10 time-steps and 10 stages per time-step. The 3D space has dimensions $(X, Y, Z) = (70, 70, 70)$ (does not account for *halo*) and a single block per each dimension x, y and z. There are 2 mesh refinement levels and a maximum of 18 blocks per core. We only simulate one variable per object, since the algorithm does not change.

We use gem5 [1] for cycle-accurate full-system simulations. Gem5 is an open-source simulator that has received significant contributions from the industry (i.e., both *Arm* and *AMD*) in recent years. The simulator faithfully models microarchitectural details of the out-of-order core (including all SVE-related architectural details), the cache hierarchy and the memory subsystem (including the on-chip interconnect), contention for shared resources, off-chip memory channels, DRAM bank conflicts, etc. The simulator models the Armv8-A ISA and boots a recent linux kernel v4.15 that has support for SVE. Out-of-order cores are modeled after an *Arm* Cortex-A72, while in-order cores resemble an *Arm* Cortex-A53. All results are obtained using out-of-order cores unless otherwise stated. Table 2 details simulated architectural parameters.

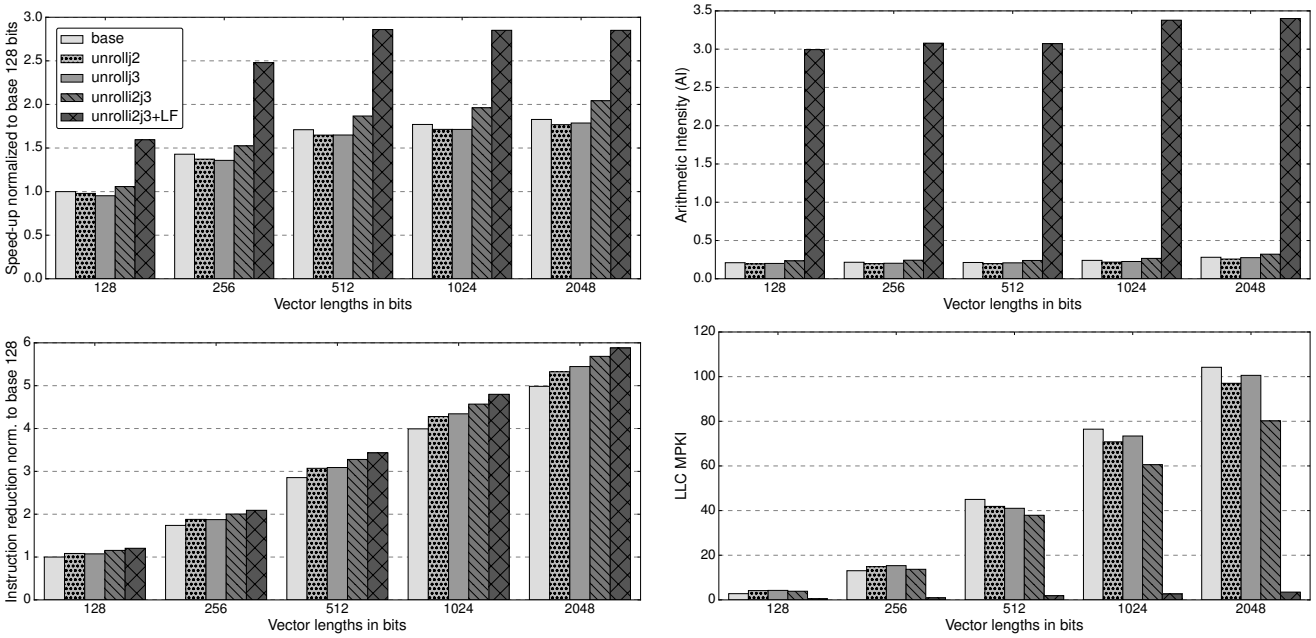


Figure 7: Performance, arithmetic intensity, instruction reduction and LLC MPKI for the 7-point stencil.

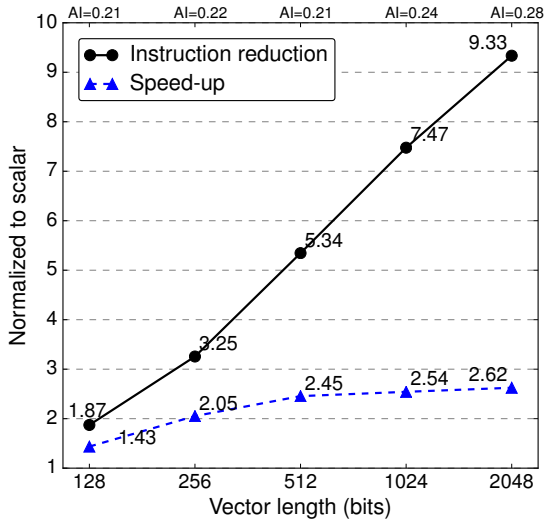


Figure 8: Instruction reduction and speed-up for a 7-point stencil - *base* SVE implementation.

5 EVALUATION

This section presents the main experimental results for the two studied stencil types. We start with a straightforward SVE implementation and apply the strategies presented in Section 3. For the 7-point stencil we apply multiple levels of loop unrolling as well as loop fusion; whereas for the 27-point stencil we apply one level of unrolling, data reuse and load trading.

5.1 7-point stencil

SVE Baseline: Ideally, developers would expect an initial instruction reduction and a speedup close to 2× when using a vector length of 128 bits compared to scalar code. However, scalar instructions such as loop index computations and control operations hinder these expectations, as they remain constant regardless of the vector length employed. Amdahl's Law is emphasized as scalar instructions translate into a bigger percentage of the total instruction count. Figure 8 shows instruction reduction and speed-up normalized to the scalar version. For 128 bits instruction reduction reaches 1.87×, indicating that the number of non-vectorized loop instructions executed is significant. As vector length increases, instruction reduction obtains diminishing returns due to scalar instructions becoming more relevant. In terms of speed-up, gains are significant for 128 bit and 256 bit vector lengths, but performance improvements stagnate at 512 bits. Performance scaling is hindered due to low arithmetic intensity (AI), calculated as $AI = \frac{\text{flops}}{\text{bytes read DRAM}}$ and shown at the top of the figure. An AI of 0.21 floating point operations per byte of data read from DRAM quickly limits performance, especially in configurations with more than 512 bits, as multiple cachelines (memory operations) are necessary to satisfy vectorized loads.

Loop Unrolling: We implement three configurations, namely, unrolling two and three iterations along the *j* axis (*unrollj2* and *unrollj3*, respectively), and unrolling two along the *i* and three iterations along the *j* axis (*unrolli2j3*). Figure 7 shows performance and instruction reduction normalized to the *base* SVE implementation using 128 bits, as well as arithmetic intensity (AI) and last-level cache (LLC) misses per kilo-instruction (MPKI). As can be seen, for *unrollj2* and *unrollj3*, performance degrades marginally. Even though instructions are reduced due to unrolling, the LLC MPKI

for 128 and 256 bit vector lengths slightly increases, leading to a reduction in AI that impacts performance. For all vector lengths, unrolling also leads to a higher percentage of memory operations within the loop body, and we find that the L1D cache performance is negatively impacted due to the additional contention, especially for vector lengths above 512 bits, as loading data into vector registers requires more than one memory access.

On the other hand, for *unrolli2j3* we observe performance improvements that are significant with vector lengths of 512 bits and above, reaching 12.1% with 2048 bits. This unroll optimization introduces reuse of loaded elements when unrolling in the *i* dimension, as opposed to the previous two unrolling optimizations. This time the observed instruction reduction is not only on scalar and control instructions, but also some memory operations are reduced, for example, there is a 28.2% reduction in load instructions when comparing the *base* SVE implementation with *unrolli2j3* using 2048 bits. This reduction in the number of memory operations significantly improves the LLC MPKI as can be seen in Figure 7.

Loop Fusion: This optimization, denoted *unrolli2j3+LF*, is built on top of *unrolli2j3*. Merging the computation and the copy loop further reduces the total amount of control instructions as can be seen in Figure 7. In addition, fusing the loops significantly increases spatial locality of the memory accesses, which translates into a lot less LLC MPKI and, consequently, a boost in terms of AI. We can observe notable performance improvements with respect to the *base* SVE implementation and previous optimizations across all vector lengths. For example, for 128 bits it is over 50% better than *base*, and for 2048 bits it is 56.6% and 39.7% better than *base* and *unrolli2j3* respectively.

5.2 27-point stencil

SVE Baseline: For the studied 27-point stencil, the instruction reduction factor is closer to the theoretical optimal point (Figure 9) than for the 7-point stencil, with reductions of 1.95× for 128 bits and 15.9× for 2048 bits. In addition, performance improvements from vectorization are also higher than in the 7-point stencil, showing good vectorization speed-ups of 4.86× at 512 bits. At larger vector lengths vectorization loses efficiency due to increased memory contention to service vector memory operations, as they need additional memory accesses. For each iteration, the amount of loaded elements and computations is higher now, as we are computing with 27 elements instead of just 7. This reduces the importance of scalar index computations and other scalar instructions. These larger performance improvements are possible due to higher AI (shown at the top of the figure).

Loop Unrolling: As can be seen in Figure 10, the *unrollj2* optimization presents a different behaviour than the one observed for the 7-point stencil, with significant performance improvements (28.7% for 128 bits) that diminishing as vector length increases (1.4% for 2048 bits). This is due to a higher AI that still offers enough computation with respect to memory operations despite the reduction of control and index calculation instructions that occurs with unrolling. Therefore, loop unrolling needs to be carefully applied depending on the characteristics of the loop body, as seen before in the 7-point stencil, it can lead to performance degradation. We

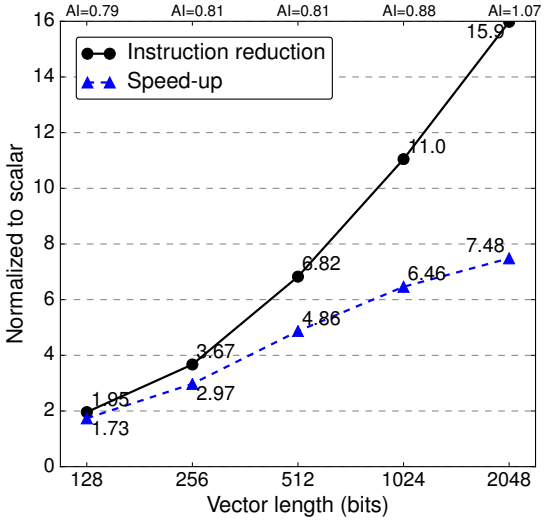


Figure 9: Instruction reduction and speed-up for a 27-point stencil - *base* SVE implementation.

limit our study to unrolling twice on *j* due to the complexity of manually unrolling the 27-point loop body.

Data Reuse: Data reuse, termed *reuse*, aims at reusing loaded data and partial computations across different iterations. As a consequence the number of instructions is reduced significantly as can be seen in Figure 10. However, this has a negative effect on AI, since many arithmetic operations are replaced by register movements. In addition, the memory access pattern presents poor spatial locality which increases LLC MPKI significantly, leading to performance degradation with respect to the *base* SVE implementation. We observe that applying optimizations that reduce instruction count is not always beneficial. Optimizations that a priori appear to be good candidates not always lead to performance improvements due to poor memory behaviour, i.e., worse locality or an increase in memory contention.

Load Trading: Load trading is implemented on top of the previous optimization, therefore it is termed *reuse+LT*. This optimization also fails to provide the expected improvement in performance. The main reason is the significant increase in instruction count when compared to *reuse*, needed to support the merging of operations that reconstruct the vector blocks. Several additional moves are required, some to preserve the content of vector registers that must be used both as a source and destination of the `splice` instruction, others used to reuse data on the next iteration.

In addition, to preserve vector length agnosticism, we need to recompute the address of **A[i][j][k − 1 + VL]** to be able to reconstruct elements **A[i][j][k]** and **A[i][j][k + 1]**. This optimization could probably benefit from additional fine-tunning to squeeze a bit more performance, by reordering operations and refining some of the movements, but the additional effort to undertake these modifications manually was too steep for the potential gains.

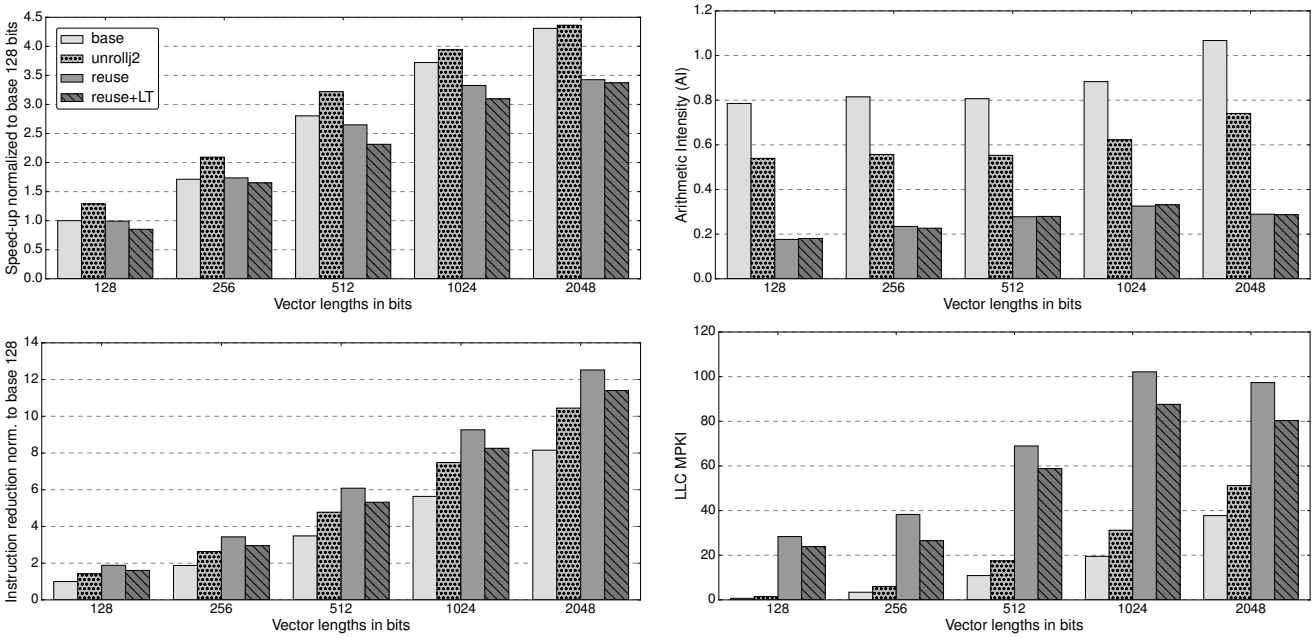
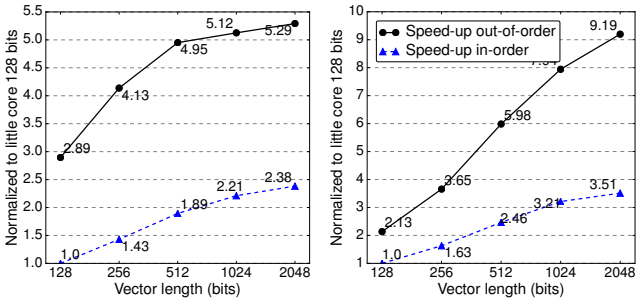


Figure 10: Performance, arithmetic intensity, instruction reduction and LLC MPKI for the 27-point stencil.



(a) 7-point stencil **(b) 27-point stencil**

Figure 11: Out-of-order versus in-order cores on *base* SVE implementation.

5.3 Comparison using in-order cores

All the presented results so far used out-of-order cores. Figure 11 shows a performance comparison of out-of-order and in-order cores normalized to in-order using 128 bits. As can be seen in the figures, the increase in performance when employing out-of-order cores is significant. For the 7-point stencil, the out-of-order core is 2.89× more performant on 128 bits, while in the 27pt is 2.13× better. The 7-point stencil has a higher percentage of memory operations, therefore, out-of-order capabilities are able to extract more performance.

In the 7-point stencil, performance for the out-of-order configuration does not improve significantly after 512 bits. This is due to a smaller number of instructions in the loop body and the additional memory accesses needed for each vectorized load, which quickly increases memory contention and the percentage of memory accesses within the loop. On the other hand, the 27-point stencil can achieve significant improvements using out-of-order cores when compared to in-order at long vector lengths. Nonetheless, our results show that out-of-order capabilities on vectorized codes are necessary, as the amount of time spent executing arithmetic instructions is lowered due to vectorization, increasing memory contention and stalls suffered by in-order cores.

5.4 Summary of the SVE Experience

SVE enables operating on different iteration counts with a single control-flow structure. In addition to having neat programming codes, loop-tail free algorithms allow to keep operating in a SIMD manner even if the vector does not have all the elements active. To the contrary, other architectures enforce us to have scalar loops that complete the remaining iterations. For example, a 2048-bit vector that operates on 8-bit integers can hold 256 elements; even if only half of them are active, it can avoid doing 128 iterations using scalar instructions.

The control flow structure (*whilelt–b.cond–incp–b*) integrates the functionality of building and reading the predicates to be used as masks in the inner loop body instructions. This is a clear reflection of the symbiosis between VLA and per-lane predication in SVE's model. Our use case benefited from per-lane predication by deactivating off-border elements in a regular grid, among others. Stencil codes on irregular grids would be also a good target for SVE, for example, recalculating a more complex condition each iteration to re-construct the predicates could be flexibly implemented by the predicate modifying a set of instructions.

The one optimization that is vector/SIMD aware is load trading, that is, it relies on having a vector code that loads repetitive data. As a consequence, the direct translation between scalar and vector

instructions does not exist. This is a second use case of per-lane predication. The two neighborhoods used (Moore and Von Neumann, radius 1) just require to create a constant predicate for the whole execution in order to re-construct a new vector by combining two other vectors. For other neighborhood shapes that have different inter-vector reconstruction opportunities, we could use other predicate modifying instructions.

Finally, our analysis shows that for loops that have low AI like the 7-point stencil, going beyond 512 bit vector lengths is not beneficial. This is due to the additional memory accesses needed to load data into the vectors, which further increases memory contention, leading to small returns in terms of performance. Applying loop unrolling on the *j* dimension did not provide benefits, however, unrolling on the *i* dimension and loop fusion increased performance significantly due to better spatial locality and higher AI. In the 27-point stencil, all long vector lengths were able to deliver additional performance due to higher AI of the loop body. However, only unrolling provided further performance benefits and both data reuse and load trading hurt performance due to poor cache behaviour and overheads in terms of additional instructions. This results highlight the difficulties in applying optimizations. Both programmers and compiler writers need to carefully select when and what optimizations to apply to maximize performance, taking into account things like AI and data locality.

6 CONCLUSIONS

Through vector lane agnosticism and per-lane predication, SVE enables programmers to write and compile applications only once, but execute the binary on any vector length, greatly improving code portability. In addition, per-lane predication simplifies codes by treating loop prologues and epilogues within the main loop. As a result, SVE's VLA adds value to the process of vectorizing an application and, in our experience, enables good productivity for developers.

This paper describes the ability of SVE to map stencil applications. We have implemented the scalar baseline and all the SVE-enabled optimizations, i.e., loop unrolling, loop fusion, data reuse and load trading, on 7-point and 27-point stencils using Armv8-A hand-coded assembly. Our performance evaluation using vector lengths ranging from 128 to 2048 bits shows that certain optimizations can boost performance significantly, i.e., loop unrolling combined with loop fusion boost performance of the 7-point stencil by more than 50% on 128 bits and by 56.6% on 2048 bits. On the other hand, certain optimizations can hurt performance due to worse data locality. Finally, we report our experiences vectorizing and optimizing stencil codes using SVE, highlighting when VLA and per-lane predication is useful and providing guidelines on what are the vector lengths that should be used depending on workload characteristics. We expect our findings to serve as a recipe for both programmers and compiler writers that would like to port and optimize stencil codes to SVE.

ACKNOWLEDGMENTS

This work has been partially supported by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), and by the Generalitat de Catalunya (contracts 2017-SGR-1328 and 2017-SGR-1414). The Mont-Blanc project receives funding from the EUs H2020 Framework Programme (H2020/2014-2020) under grant agreements no. 671697 and no. 779877. M. Moreto has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104. Finally, A. Armejach has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Juan de la Cierva postdoctoral fellowship number FJCI-2015-24753.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[2] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011 - Conference Proceedings*. 676–687. https://doi.org/10.1109/IPDPS.2011.70

[3] Intel Corporation. 2016. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf

[4] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David A. Patterson, John Shalf, and Katherine A. Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*. 4. https://doi.org/10.1145/1413370.1413375

[5] Hikmet Dursun, Ken-ichi Nomura, Liu Peng, Richard Seymour, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. 2009. A Multi-level Parallelization Framework for High-Order Stencil Computations. In *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*. 642–653. https://doi.org/10.1007/978-3-642-03869-3_61

[6] Roger Espasa, Mateo Valero, and James E. Smith. 1998. Vector Architectures: Past, Present and Future. In *Proceedings of the 12th international conference on Supercomputing, ICS 1998, Melbourne, Australia, July 13-17, 1998*. 425–432. https://doi.org/10.1145/277830.277935

[7] Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005*. 361–366. https://doi.org/10.1145/1088149.1088197

[8] S. Fuller. 1998. *Motorola's AltiVec™Technology*. Technical Report. Motorola Inc,. http://www.nxp.com/assets/documents/data/en/fact-sheets/ALTIVECWP.pdf

[9] A. Heimlich, ACA Mol, and CMNA Pereira. 2011. GPU-based Monte Carlo simulation in neutron transport and finite differences heat equation evaluation. *Progress in Nuclear Energy* 53, 2 (2011), 229–239.

[10] J. L. Hennessy and D. A. Patterson. 2006. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[11] Shoaib Kamil, Cy P. Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470421

[12] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine A. Yelick. 2006. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory System Performance and Correctness, San Jose, California, USA, October 11, 2006*. 51–60. https://doi.org/10.1145/1178597.1178605

[13] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine A. Yelick. 2005. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 workshop on Memory System Performance, Chicago, Illinois, USA, June 12, 2005*. 36–43. https://doi.org/10.1145/1111583.1111589

[14] Dimitri Komatitsch, Gordon Erlebacher, Dominik Göddeke, and David Michéa. 2010. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *J. Comput. Physics* 229, 20 (2010), 7692–7714. https://doi.org/10.1016/j.jcp.2010.06.024

[15] S. Kronawitter and C. Lengauer. 2014. Optimization of two Jacobi Smoother Kernels by Domain-Specific Program Transformation. *HiStencils 2014*, 75–80.

[16] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanovìc. 2015. *The Hwacha Vector-Fetch Architecture Manual.* Technical Report. Electrical Engineering and Computer Sciences, University of California at Berkeley. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.pdf

[17] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. 2011. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*. 11:1–11:12. https://doi.org/10.1145/2063384.2063398

[18] F. Molnár, F. Izsák, R. Mészáros, and I. Lagzi. 2011. Simulation of reaction–diffusion processes in three dimensions using CUDA. *Chemometrics and Intelligent Laboratory Systems* 108, 1 (2011), 76–85.

[19] Liu Peng, Richard Seymour, Ken-ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R. Volz, and Chap C. Wong. 2009. High-order stencil computations on multicore clusters. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. 1–11. https://doi.org/10.1109/IPDPS.2009.5161011

[20] J. Reinders and J. Jeffers. 2014. *High Performance Parallelism Pearls, Multicore and Many-core Programming Approaches.* Morgan Kaufmann, Chapter Characterization and Auto-tuning of 3DFD, 377–396.

[21] R. M. Russell. 1978. The CRAY-1 Computer System. *Commun. ACM* (1978), 63–72.

[22] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (2017), 26–39. https://doi.org/10.1109/MM.2017.35

[23] L. Szustak, K. Rojek, R. Wyrzykowski, and P. Gepner. 2014. Toward efficient distribution of MPDATA stencil computation on Intel MIC architecture. *Proce. HiStencils* 14 (2014), 51–56.

[24] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The pochoir stencil compiler. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*. 117–128. https://doi.org/10.1145/1989493.1989508

[25] Tommaso Toffoli and Norman Margolus. 1987. *Cellular automata machines - a new environment for modeling.* MIT Press.

[26] U. Trottenberg, C. W Oosterlee, and A. Schuller. 2000. *Multigrid.* Academic press.

[27] A. Waterman, Y. Lee, D. Patterson, and K. Asanovìc. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report. Electrical Engineering and Computer Sciences, University of California at Berkeley. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf

[28] Toshio Yoshida. 2016. Introduction of Fujitsu's HPC Processor for the Post-K Computer. In *Hot Chips 28 Symposium (HCS) (Hot Chips '16)*. IEEE.

[29] Charles Yount. 2015. Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*. 865–870. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.27

[30] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK - Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning. In *Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC@SC 2016, Salt Lake, UT, USA, November 14, 2016*. 30–39. https://doi.org/10.1109/WOLFHPC.2016.08

[31] V. T. Zhukov, Mikhail M. Krasnov, N. D. Novikova, and O. B. Feodoritova. 2015. Multigrid effectiveness on modern computing architectures. *Programming and Computer Software* 41, 1 (2015), 14–22. https://doi.org/10.1134/S0361768815010077