Using Arm's Scalable Vector Extension on Stencil Codes

Adrià Armejach[1,2] · Helena Caminal[3]
Juan M. Cebrian[1] · Rubén Langarita[1]
Rekai González-Alberquilla[4]
Chris Adeniyi-Jones[4] · Mateo Valero[1]
Marc Casas[1] · Miquel Moretó[1,2]

Received: date / Accepted: date

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 of 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 1.57×. In addition, we show that certain optimizations can hurt performance due to reduced arithmetic intensity and instruction overheads, and provide insight useful for compiler optimizers.

Keywords data-level parallelism · scalable vector extension · vector length agnostic · stencil computations

Adrià Armejach
adria.armejach@bsc.es

[1] Barcelona Supercomputing Center, Barcelona, Spain
[2] Universitat Politècnica de Catalunya, Barcelona, Spain
[3] Cornell University, Ithaca, NY, USA
[4] Arm, Cambridge, UK

This is a post-peer-review, pre-copyedit version of an article published in *Journal of supercomputing*.
The final authenticated version is available online at: http://dx.doi.org/10.1007/s11227-019-02842-5

1 Introduction

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) [1], 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 [2]. 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.* [1]. 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 [3, 4]. Stencils are typically memory bound, which is also a challenge for vector architectures [5]. 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, on top of the MPI-enabled version of miniAMR, 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 multi-core architectural simulations that faithfully model the SVE architecture. We employ roofline models to study the performance impact of different vector lengths, ranging from 128 to 2048 bits, on the evaluated code optimizations. In addition, by using metrics of interest we also compare the effectiveness of each code optimization. We find that loop fusion and data reuse unlock the largest performance improvements, up to 1.45× and 1.57× respectively.
- We provide a memory bandwidth sensitivity study to show the performance impact additional bandwidth can have on the base 7-point and 27-point stencils, as well as on their best performing optimized version.
- 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 to solve large-scale high-dimensional problems using finite-difference methods (FDM). PDEs are the base to provide numerical approximations to complex computational problems from science and engineering [6, 7, 8, 9, 10]. 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 [11]. 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 resides 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 neighborhoods of radius 1.

Vector/SIMD Architectures: Vector architectures have been present almost since the beginning of the history of parallel computing [12]. 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 [13] has masked operations and gather/scatter memory operations, and PowerPC AltiVec [14] implements the *compare* operation to create field masks.

There are also recent proposals from academia, such as the Hwacha Design [15], which extends the RISC-V ISA [16]. Its philosophy is based on traditional vector architectures, similar to Cray1 [17], 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 need of 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 [18]. Additionally, stencils usually suffer 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 [19, 20, 21, 22, 23, 24]. There are also stencil-specific optimization frameworks [4] and compiler support [25] to ease the optimization process for developers.

Optimization of stencil computations via SIMD instructions is also a common approach [26, 27], and the usage of GPUs has also been considered [28]. 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 mimic SIMD acceleration capabilities offered by GPUs, by 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, SVE encoding fits in a 28-bit region, occupying only one sixteenth of all the available opcodes that can be represented with the 32-bit encoding Armv8-A has, leaving

```
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
```

Fig. 1 3D Jacobian method pseudo-code (T time-steps) with *computational* and *copy* loops.

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 accomplishes 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 SVE features include: (i) serialized reductions that 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 [26, 27, 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 alternative solutions that prevent having a copy loop, and cover it in detail when introducing the loop fusion optimization.

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.

Listing 1 NEON version (128-bit)

```
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
```

Listing 2 SVE version (VLA)

```
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:
```

Fig. 2 Code fragment of the 7-point stencil computation loop - only the innermost loop (**k**).

– We use the notation (i, j, k) to denote a specific iteration of the three loops, from outermost to innermost.
– We assume there is memory allocation for the *halo* of the stencil, that it is initialized and never updated.

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.

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 its equivalent vector version. In addition, we typically need to operate 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 2 shows the NEON and SVE baseline code of the innermost loop of a 7-point stencil. 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 structure would check if there are more elements in the k dimension to continue execution. 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 tail loop performs the same computations but on a single element.

	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	$\frac{6}{N} + \frac{6}{M} + \frac{12}{NM} + 3$

Table 1 Vector loads-per-VL elements ratio.

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 a 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 $(z\_size \ mod \ VL) \neq 0$.

We want to outline that although mask operations exist in other current SIMD architectures (such as Intel AVX-512 [13]), 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.

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 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, the benefits unrolling unlocks in terms of load reduction are limited by the amount of architectural resources. There are different ways of

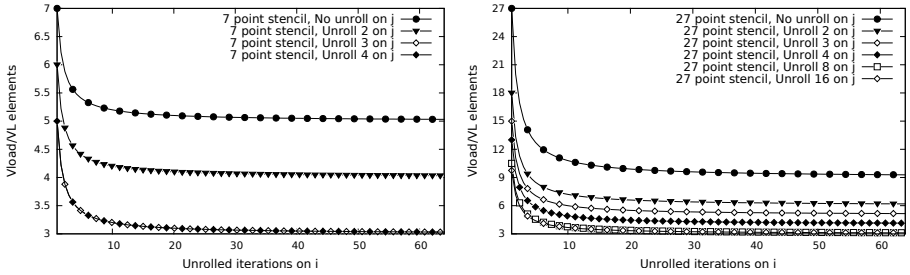


Fig. 3 Vector loads-per-VL elements ratio when unrolling i and j for a 7-point (left) and a 27-point (right) stencil.

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.

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 $\mathbf{A}[\mathbf{i}][\mathbf{j}][\mathbf{k}]$ is $\mathbf{A}[\mathbf{i}+\mathbf{1}][\mathbf{j}][\mathbf{k}]$. 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 $\mathbf{A}[\mathbf{i}][\mathbf{j}][\mathbf{k}]$. A pseudo-code of this optimization is shown in Figure 5, where *tmp* is

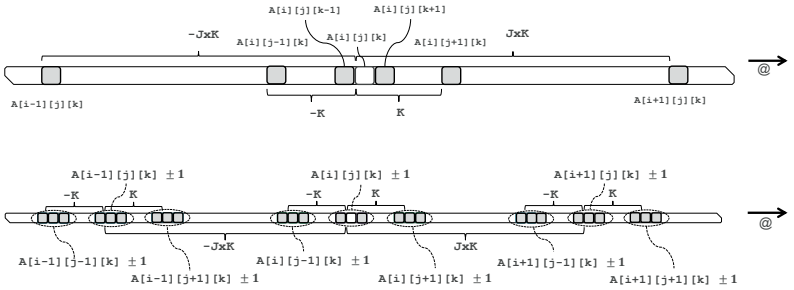


Fig. 4 Linear representation of the accessed (loaded) elements in iteration (i, j, k) on array $A[I][J][K]$ in the *computation loop* on a 7-point (top) and a 27-point (bottom) stencils.

```
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
```

Fig. 5 Loop fusion (*computation* and *copy*) on a 3D Jacobian method, T time-steps.

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 we update the value of its last dependency $\mathbf{A[i-1][j][k]}$ to the original array and then store it to the temporal array ($\mathbf{tmp[j][k]}$). 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 $\mathbf{A[i][j][k]}$ is element $\mathbf{A[i+1][j+1][k+1]}$, 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.

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 $\mathbf{A[i][j][k]}$ are stored in consecutive memory locations. As a consequence, given a tuple of elements starting at address $\mathbf{A[i][j][k-1]}$, we can build tuples starting at $\mathbf{A[i][j][k]}$ (and $\mathbf{A[i][j][k+1]}$) by re-

moving the first (and second) elements from the tuple, and then concatenating with the first (and second) elements from the tuple starting at $\mathbf{A[i][j][k-1+VL]}$. 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: $\mathbf{A[i][j][k-1]}$ and $\mathbf{A[i][j][k-1+VL]}$. Then we combine them properly to obtain $\mathbf{A[i][j][k]}$ and $\mathbf{A[i][j][k+1]}$. It is worth pointing out that in the next iteration, k^+ will equal $k + VL$. Therefore, $\mathbf{A[i][j][k^+-1]}$ is the same as $\mathbf{A[i][j][k-1+VL]}$. 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 $\mathbf{A[i][j][k]}$ and $\mathbf{A[i][j][k+1]}$ 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 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 [13] 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. Because the *valignq* 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*.

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, as the amount of reuse is much higher in this stencil type. 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 $(\mathbf{i, j, k-2VL})$, $(\mathbf{i, j, k-VL})$ and $(\mathbf{i, j, k})$. 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,

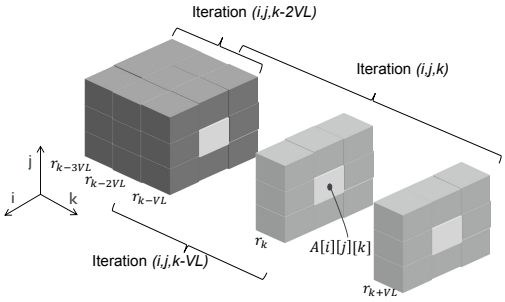


Fig. 6 Elements (and partial results $r_{<iter.>}$) used in three consecutive iterations of the 27-point stencil, each cube representing a VL array of elements in the k dimension.

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.

4 Methodology

We implemented the SVE baseline and subsequent optimizations over a well-known stencil code found in the Mantevo *miniapps* suite - *miniAMR* [30, 31]. 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 $(X, Y, Z) = (64, 64, 64)$ (does not account for *halo*) and a single block per each dimension x, y and z. There is 1 mesh refinement level and a maximum of 18 blocks per core. We employ 8 MPI processes in our executions, 2 processes for each of the three directions (2^3).

Processor size	8 cores - 2 clusters of 4 cores each.
Cores	*out-of-order:* 3-wide issue/retire, 92-entry instruction queue, 192-entry ROB, 48 LDQ + 48 STQ, 2GHz *in-order:* 2-wide issue, 5-entry store buffer, 2GHz
Private Caches	*out-of-order:* L1I: 48KB, 3-way, 2 cycle, 2 ports, 8MSHRs LID: 32KB, 2-way, 2 cycle, 2 ports, 16MSHRs L2: 256KB, 8-way, 9 cycle, 24MSHRs *in-order:* L1I: 32KB, 2-way, 2 cycle, 1 port, 8MSHRs LID: same as out-of-order but with 1 port L2: same as out-of-order
Last-level Cache	16MB, 16-way, 64B lines, 8 banks, 32MSHRs per bank Data bank access latency of 14 cycles.
NoC	Coherent crossbar, 128-bit wide, 2 cycles
Main Memory	8 HBM channels, 128-bit width, 8 banks/channel 128-entry write and 64-entry read buffers per channel 128GB/s peak bandwidth. Bank conflicts and queuing delays modeled

Table 2 Parameters for full-system simulations.

We use gem5 [32] 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, HBM bank conflicts, etc. The simulator models the Armv8-A ISA and boots a recent linux kernel v4.15 that has support for SVE. The parameters modeled are representative of modern mobile cores. Out-of-order and in-order cores employ parameters extracted from the *Arm* Cortex-A72 and *Arm* Cortex-A53 technical reference manuals [33, 34], respectively. All results are obtained using out-of-order cores unless otherwise stated. Table 2 details simulated architectural parameters.

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 Baseline implementations

Figure 7 shows the roofline model for the simulated system. A roofline model ties together floating-point performance, arithmetic intensity, and memory performance in a two-dimensional graph. The Y-axis is GFlops per second (performance). Theoretical ceilings can be derived using the hardware specifications; in our case, the simulated system can achieve 32GFlops/s for a 128- bit vector length: 1 vector processing unit $\times$ 2 double floating-point operations per unit $\times$ 2 GHz $\times$ 8 cores. The X-axis is arithmetic intensity, i.e., operations per byte of DRAM traffic

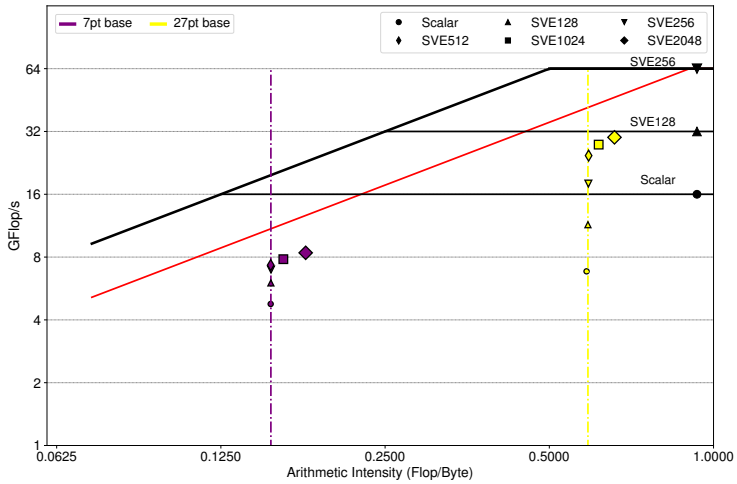


Fig. 7 Roofline model of the simulated system for the 7 and 27 point baseline stencils. For each baseline we plot performance for scalar and all the different vector lengths.

($AI = \frac{\text{flops}}{\text{bytes read DRAM}}$). Therefore, we measure traffic between the caches and memory rather than between the processor and the caches. Thus, arithmetic intensity suggests the DRAM bandwidth needed by a kernel on a particular computer. We can then plot memory performance by calculating the maximum floating-point performance that the memory system of that computer can support for a given arithmetic intensity. As can be seen in the figure, we plot 2 memory bandwidth ceilings, one for the theoretical peak pin bandwidth (black line) and another one for the measured bandwidth the system delivered (red line). In addition, we plot multiple peak performance ceilings depending on the vector length employed.

As can be seen in the figure, for the 7-point baseline implementation, performance gains are significant for 128 bit and 256 bit vector lengths; however, performance improvements stagnate at 512 bits. Performance scaling is hindered due to low arithmetic intensity (AI). An AI of 0.15 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 transactions) are necessary to satisfy vectorized loads.

For the 27-point baseline stencil, performance improvements from vectorization are significantly larger than in the 7-point stencil, showing speed-ups in terms of GFlop/s with respect to scalar of 2.63× and 3.59× for 256 and 512 bits, respectively. At wider 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, which is now around 0.63.

Note that both baseline stencil codes are far from their theoretical peak performance ceilings, this is expected due to the low arithmetic intensity these benchmarks have, which makes them become memory bound quickly.

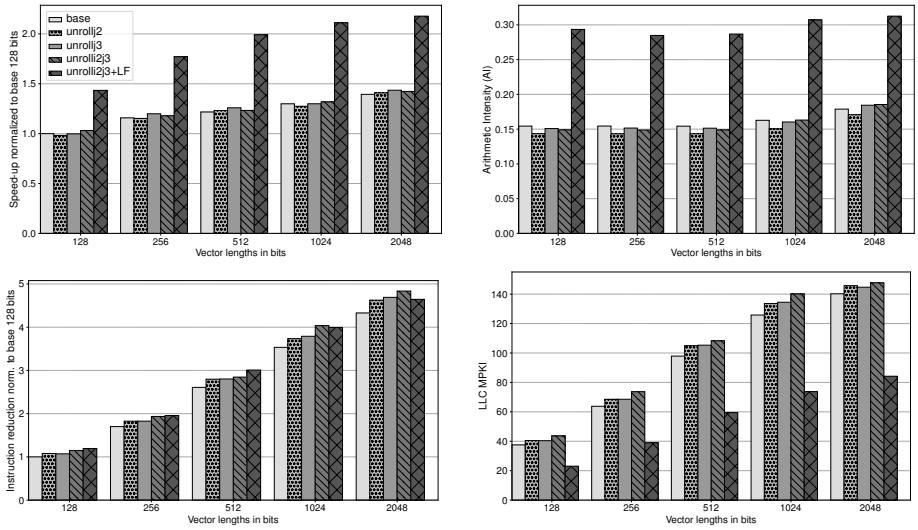


Fig. 8 Performance, arithmetic intensity, instruction reduction and LLC MPKI for the 7-point stencil baseline and optimizations.

5.2 Optimizations over the 7-point baseline stencil

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 8 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 all the unrolling configurations, performance stays on par with *base*. Even though instructions are reduced due to unrolling, the LLC MPKI slightly increases, leading to a reduction in AI that hinders potential performance improvements from reducing the instruction footprint. 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.

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 8. 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 1.45× better than *base*, and for 2048 bits it is 1.52× better. This optimization yields similar improvements accross all vector lenghts, which means it is helping mitigate the main underlying issue, memory contention.

Figure 9 shows the roofline model for all the evaluated 7pt stencil variants. As can be seen, *unrolli2j3+LF* is able to attain higher performance in terms of

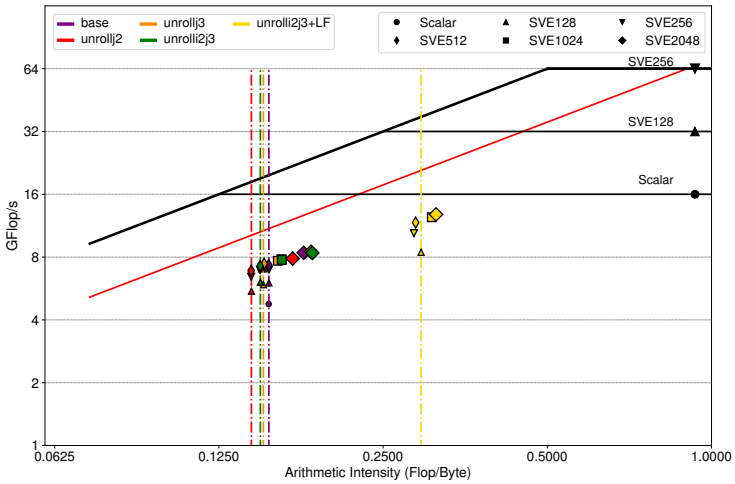


Fig. 9 Roofline model for all the evaluated 7pt stencil variants.

GFlop/s due to its higher AI. Which also translates into better overall execution time as seen before in Figure 8. This optimization allows the 512 bit configuration to extract additional performance. The rest of optimizations all fall within the same lower AI range, and are therefore more memory bound. In these cases, going beyond 256 bits does not provide significant benefits.

5.3 Optimizations over the 27-point baseline stencil

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 a significant performance improvement (1.28× for 128 bits) that diminishes as vector length increases (1.03× 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 slight performance degradation. We 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. Nontheless, performance improves significantly with respect to the *base* SVE implementation, as the amount of reused data and partial computations is significant, i.e., two thirds per iteration. As expected, and similarly to the *unrollj2* optimization, performance improvements are larger with narrower vector lengths (1.57× for 128 bits), and diminish as the vector length increases (1.08× for 2048 bits) due to higher memory contention (see MPKI in Figure 10).

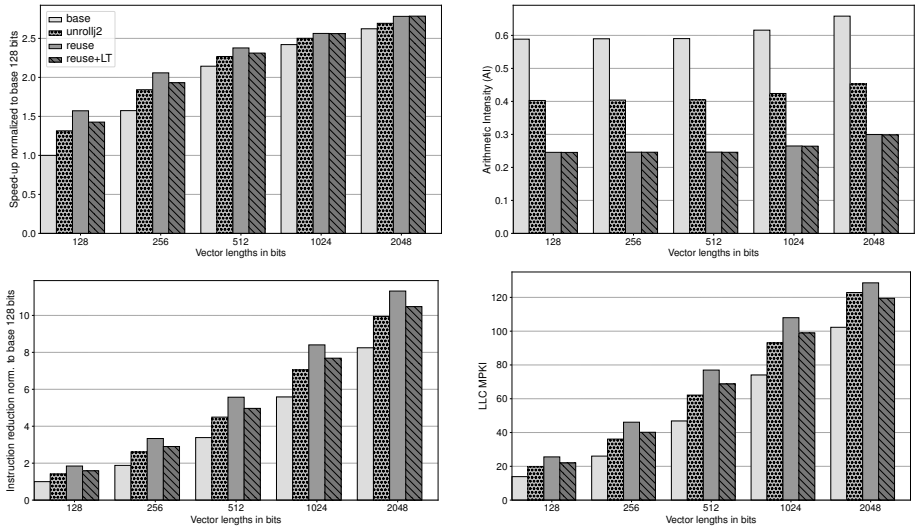


Fig. 10 Performance, arithmetic intensity, instruction reduction and LLC MPKI for the 27-point stencil.

Load Trading: Load trading is implemented on top of the previous optimization, therefore it is termed *reuse+LT*. This optimization 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. To do the reconstruction, 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, and others to reuse data on the next iteration. Therefore, a non-destructive `splice` instruction with different source and destination registers would be beneficial for this particular optimization.

In addition, to preserve vector length agnosticism, we need to recompute the address of $\mathbf{A[i][j][k-1+VL]}$ to be able to reconstruct elements $\mathbf{A[i][j][k]}$ and $\mathbf{A[i][j][k+1]}$. 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. It is interesting to see that performance degradation with respect to *reuse* fades as vector length increases, since as the benchmark becomes more memory bound, the additional move operations become less relevant.

Figure 11 shows the roofline model for all the evaluated 27pt stencil variants. We observe that the *base* implementation has the highest AI, and as a consequence it has also the highest GFlop/s. However, as seen before, the *reuse* optimization, which has the lowest AI, is the vartiant that has the best overall execution time (see Figure 10). This is explained by the fact that *reuse* performs significantly less floating-point operations to achieve the same result as *base*. For all the 27-point stencil variants we observe how the 256 bit SVE unlocks significant improvements with respect to 128 bits. In particular, for *reuse*, 1.31×; while SVE 512 bits further improves over 256 bits by 1.16×.

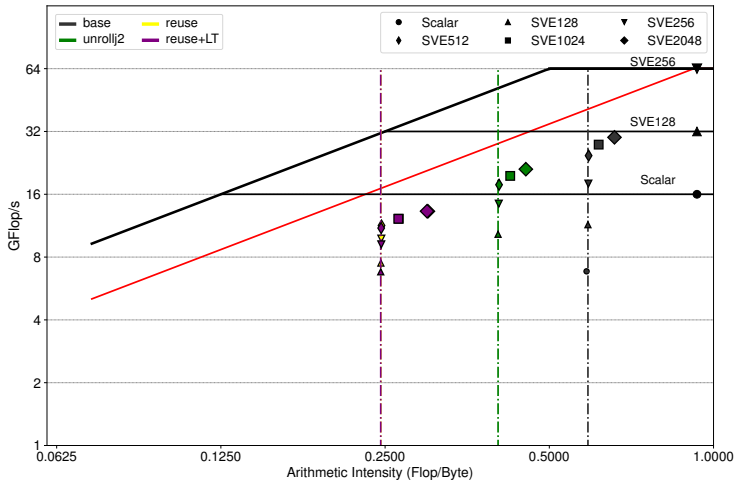


Fig. 11 Roofline model for all the evaluated 27pt stencil variants.

5.4 Memory bandwidth sensitivity analaysis

Figure 12 shows the speed-up obtained when doubling the amount of available peak off-chip memory bandwidth by doubling the amount of avaialble HBM memory channels - effectively adding another HBM stack. Leading to 256 GB/s peak bandwidth, 32 GB/s per core. The figure shows numbers for the two baselines, the best optimization for each stencil, and the average. We can see that improvements are larger as the vector length increases with diminishing returns above 512 bits.

Although we have doubled the available bandwidth, performance improvements are modest, especially for narrow vector lengths. For 128, 256 and 512 bits performance improves 1.08×, 1.17×, and 1.25× on average, respectively. A key observation here is that with 8 cores it is difficult to saturate the available 256 GB/s - 32 GB/s per core. Conventional prefetchers and cache hierarchies are not aggressive enough to saturate 16 HBM memory channels with 8 cores, which is an expected result, as with a single core it is only possible to achieve a peak bandwidth of about 19 GB/s in our simulated system. Note that this is on par with a real top tier high-performance server processor.

Therefore, a large portion of the available bandwidth is unused and performance improvements are not as large as one would expect, because the base bandwidth of 128 GB/s already delivered 16GB/s per core.

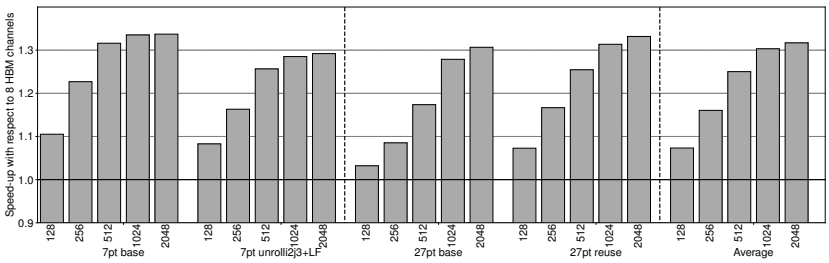
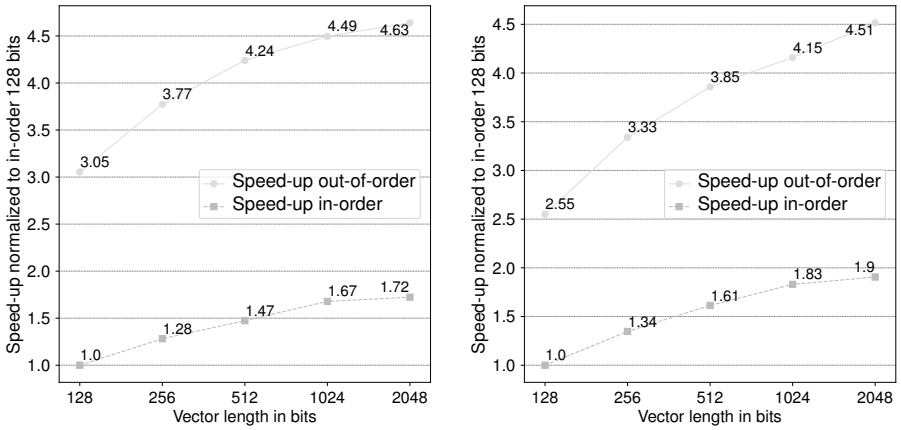


Fig. 12 Performance comparison when doubling the number of HBM memory channels - 256 GB/s peak bandwidth.



(a) 7-point *unrolli2j3+LF* (b) 27-point *reuse*

Fig. 13 Out-of-order versus in-order cores on best performing optimization for each stencil.

5.5 Comparison using in-order cores

All the presented results so far used out-of-order cores. Figure 13 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 *unrolli2j3+LF*, the out-of-order core is 3.05× more performant on 128 bits, while in the 27-point *reuse* is 2.55× 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 wider 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.

5.6 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 the 7-point stencil and its low AI, 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, wider vector lengths were able to deliver additional performance due to higher AI of the loop body. Unrolling and data reuse did provided further performance benefits, however, load trading hurts performance due to 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 1.45× on 128 bits, while data reuse on the 27-point stencil improves performance by 1.57×. On the other hand, when applying load trading we were not able to improve performance. In this case, support for a `splice` instruction that is non-destructive for the source registers

would have helped to make this optimization perform better. 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.

Acknowledgements 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. M. Casas has been partially supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the European Union (Contract 2013 BP B 00243). 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. N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker, "The ARM scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.35
2. T. Yoshida, "Introduction of Fujitsu's HPC Processor for the Post-K Computer," in *Hot Chips 28 Symposium (HCS)*, ser. Hot Chips '16. IEEE, 2016.
3. K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick, "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*, 2008, p. 4. [Online]. Available: http://doi.acm.org/10.1145/1413370.1413375
4. C. Yount, J. Tobin, A. Breuer, and A. Duran, "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*, 2016, pp. 30–39. [Online]. Available: https://doi.org/10.1109/WOLFHPC.2016.08
5. M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005*, 2005, pp. 361–366. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088197
6. U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*. Academic press, 2000.
7. V. T. Zhukov, M. M. Krasnov, N. D. Novikova, and O. B. Feodoritova, "Multigrid effectiveness on modern computing architectures," *Programming and Computer Software*, vol. 41, no. 1, pp. 14–22, 2015. [Online]. Available: https://doi.org/10.1134/S0361768815010077
8. D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *J. Comput. Physics*, vol. 229, no. 20, pp. 7692–7714, 2010. [Online]. Available: https://doi.org/10.1016/j.jcp.2010.06.024
9. A. Heimlich, A. Mol, and C. Pereira, "Gpu-based monte carlo simulation in neutron transport and finite differences heat equation evaluation," *Progress in Nuclear Energy*, vol. 53, no. 2, pp. 229–239, 2011.
10. F. Molnár, F. Izsák, R. Mészáros, and I. Lagzi, "Simulation of reaction–diffusion processes in three dimensions using cuda," *Chemometrics and Intelligent Laboratory Systems*, vol. 108, no. 1, pp. 76–85, 2011.

11. T. Toffoli and N. Margolus, *Cellular automata machines - a new environment for modeling*, ser. MIT Press series in scientific computation. MIT Press, 1987.
12. R. Espasa, M. Valero, and J. E. Smith, "Vector Architectures: Past, Present and Future," in *Proceedings of the 12th international conference on Supercomputing, ICS 1998, Melbourne, Australia, July 13-17, 1998*, 1998, pp. 425–432. [Online]. Available: http://doi.acm.org/10.1145/277830.277935
13. Intel Architecture instruction set extensions programming reference. Intel Corporation (2016). [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf
14. S. Fuller, "Motorola's altivec^{TM}technology," Motorola Inc,, Tech. Rep., 1998. [Online]. Available: http://www.nxp.com/assets/documents/data/en/fact-sheets/ALTIVECWP.pdf
15. Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanovìc, "The hwacha vector-fetch architecture manual," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep., 2015. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.pdf
16. A. Waterman, Y. Lee, D. Patterson, and K. Asanovìc, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep., 2014. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf
17. R. M. Russell, "The cray-1 computer system," *Commun. ACM*, pp. 63–72, 1978.
18. J. Reinders and J. Jeffers, *High Performance Parallelism Pearls, Multicore and Many-core Programming Approaches*. Morgan Kaufmann, 2014, ch. Characterization and Autotuning of 3DFD, pp. 377–396.
19. S. Kronawitter and C. Lengauer, "Optimization of two Jacobi Smoother Kernels by Domain-Specific Program Transformation," in *HiStencils 2014*, 2014, pp. 75–80.
20. M. Christen, O. Schenk, and H. Burkhart, "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*, May 2011, pp. 676–687.
21. L. Szustak, K. Rojek, R. Wyrzykowski, and P. Gepner, "Toward efficient distribution of MPDATA stencil computation on Intel MIC architecture," *Proce. HiStencils*, vol. 14, pp. 51–56, 2014.
22. S. Kamil, C. P. Chan, L. Oliker, J. Shalf, and S. Williams, "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*, 2010, pp. 1–12. [Online]. Available: https://doi.org/10.1109/IPDPS.2010.5470421
23. S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. A. Yelick, "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*, 2005, pp. 36–43. [Online]. Available: http://doi.acm.org/10.1145/1111583.1111589
24. S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick, "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*, 2006, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/1178597.1178605
25. Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. Luk, and C. E. Leiserson, "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)*, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508
26. H. Dursun, K. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, "A multilevel 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*, 2009, pp. 642–653. [Online]. Available: https://doi.org/10.1007/978-3-642-03869-3_61
27. L. Peng, R. Seymour, K. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong, "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*, 2009, pp. 1–11. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5161011

28. N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "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*, 2011, pp. 11:1–11:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063398
29. C. Yount, "Vector folding: Improving stencil performance via multi-dimensional simd-vector representation," in *IEEE 17th International Conference on High Performance Computing and Communications*, Aug 2015, pp. 865–870.
30. M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.
31. The mantevo suite release version 3.0. [Online]. Available: https://mantevo.org/download/
32. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718
33. ARM Cortex-A72 MPCore Processor Technical Reference Manual. [Online]. Available: https://static.docs.arm.com/100095/0003/cortex_a72_mpcore_trm_100095_0003_05_en.pdf?_ga=2.187644577.805846766.1551351186-1814310934.1538732624
34. ARM Cortex-A53 MPCore Processor Technical Reference Manual. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf