Computer Science and Information Systems 21(1):203–221 https://doi.org/10.2298/CSIS230819005R

SPC5: an efficient SpMV framework vectorized using ARM SVE and x86 AVX-512

Evann Regnault[1] and Bérenger Bramas[2,3]

[1] Strasbourg University, UFR de Mathématique et d'Informatique
7, rue René Descartes, 67084 Strasbourg, France
evann.regnault@etu.unistra.fr
[2] Inria Nancy, CAMUS Team, 615 Rue du Jardin-Botanique
54600 Villers-lès-Nancy, France
[3] ICube laboratory, ICPS Team, 300 bd Sébastien Brant
67412 Illkirch Cedex, France
berenger.bramas@inria.fr

Abstract. The sparse matrix/vector product (SpMV) is a fundamental operation in scientific computing. Having access to an efficient SpMV implementation is therefore critical, if not mandatory, to solve challenging numerical problems. The ARM-based AFX64 CPU is a modern hardware component that equips one of the fastest supercomputers in the world. This CPU supports the Scalable Vector Extension (SVE) vectorization technology, which has been less investigated than the classic x86 instruction set architectures. In this paper, we describe how we ported the SPC5 SpMV framework on AFX64 by converting AVX512 kernels to SVE. In addition, we present performance results by comparing our kernels against a standard CSR kernel for both Intel-AVX512 and Fujitsu-ARM-SVE architectures.

Keywords: SpMV, vectorization, AVX-512, SVE.

1. Introduction

The sparse matrix/vector product (SpMV) is a fundamental operation in scientific computing. It is the most important component of iterative linear solvers, which are widely used in finite element solvers. This is why SpMV has been and remains studied and improved.

Most of the studies work on the storage of sparse matrices, the implementation of SpMV kernels for novel hardware, or the combination of both.

In a previous work [7], we proposed a new sparse matrix storage format and its corresponding SpMV kernel in a framework called SPC5. The implementation was for x86 CPUs using the AVX512 instruction set architectures, and it was efficient for various types of data distribution.

In the current work, we are interested in porting this implementation on ARM SVE [21,4,3] architecture. In other words, we aim at keeping the SPC5 storage format but create computational kernels that are efficient on ARM CPUs with SVE.

AVX512 and SVE instruction set architectures are different in their philosophies and features. Consequently, as it is usually the case with vectorization, providing a new computational kernel is like solving a puzzle: we have the operation we want to perform on one side and the existing hardware instructions on the other side.

The contribution of the paper is to depict a new SpMV kernel for ARM SVE, and to demonstrate its performance on several sparse matrices of different shapes. A secondary contribution is the description of our new AVX512 implementation, which is much simpler than the previous assembly implementation, while still delivering the same performance.

This paper is organized as follows. In Section 2, we start by describing the vectorization principle, then the SpMV operation and the challenges of its efficient implementation, and finally provide the specificities of SPC5. Then, in Section 2.4, we present our SPC5 implementation with SVE. Finally, we study the performance of our implementation in Section 4.

2. Background

2.1. Vectorization

Vectorization, also named SIMD for single instruction multiple data [11], is a key mechanism of modern processing units to increase the performance despite the clock frequency stagnation. As its name suggests, the idea consists in working on several elements stored in vectors instead of scalar distinct elements. As such, instead of performing operations on one element at a time, we perform the operations on vectors of elements using a vector instruction set architecture (ISA) that supports vector instructions. We provide a schematic view of the concept in Figure 1.

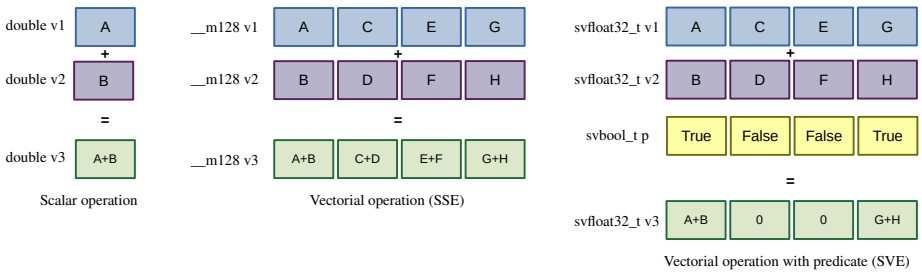


Fig. 1. Illustration of a scalar operation, a vectorial operation and a vectorial operation with predicate. Each of the three instructions is performed with a single instruction

Vectorization is straightforward when we aim to apply the same operation on all the elements of a vector. However, the principle is challenging when we have divergence, i.e., we do not apply the exact same operations on all the elements, or when we need to perform data layout transformations, i.e., the input/output data blocks from the main memory that are loaded from (stored to) vectors are not contiguous, or we need to shuffle the data inside the vectors.

Moreover, not all instruction sets support the same operations, making each implementation specific to a given hardware. Consequently, what could be done with a single

instruction in a given instruction set, might need several instructions in another. For example, non-contiguous stores (scatter), non-contiguous loads (gathers), or internal permutation/merging of vectors are not available in all existing instruction sets and not necessarily similar when they are supported.

Many computational algorithms use conditional statements, therefore several solutions have been proposed to manage vector divergences. The first one is the single instruction multiple thread (SIMT) programming model, as used in CUDA and OpenCL. While the programmer expresses its parallel algorithm as if independent execution threads would be used, it is actually large vector units that will perform the execution, where each thread will be an element of the vector. The hardware takes care of the coherency during the execution.

The second mechanism is the use of a vector of predicates, where each predicate tells if an operation should be applied on an element of the vector. When the elements of a vector should follow different execution paths (branches), all paths will be executed but predicate vectors will ensure to apply the correct operations. The ARM SVE technology uses this mechanism, and most instructions can be used with a predicate vector. Similar behavior can be obtained with classic x86 instruction sets using, for example, binary operations to merge several vectors obtained through different execution branches.

2.2. Related Work on Vectorized with SVE

Developing optimized kernels with SVE is a recent research topic [18,14,2,25,10]. A previous study [1] has focused on the modelling and tuning of the A64FX CPU. The authors implemented the SELL-C-σ SpMV kernels and tuned it for this hardware. This kernel was originally made for GPUs but works well on CPUs too. However, the format is very different from the CSR format and requires a costly conversion step, which we aim to avoid. Additionally, the authors have performed important tuning for each matrix, by permuting the matrix or performing costly parameter optimization, where we want to provide a unique solution.

2.3. SpMV

The SpMV operation has been widely studied. This operation is memory bound in most cases with a low arithmetic intensity. Consequently, a naive vectorization usually does not provide significant benefits if the arithmetic intensity remains unchanged. This is why the storage of the sparse matrix is usually the central point of improvement.

Each new ISA can potentially help to create new storage formats that take less memory and/or that can be vectorized more efficiently.

For example, consider the more simple storage format called *coordinates* (COO) or IJV, where each non-zero value (NNZ) is stored with a triple row index, column index and floating point value. In this case, for each NNZ we need two integers and one floating point value. Not only this format is heavy but it is difficult to vectorized its corresponding SpMV kernel.

Another well-known storage format is the *compressed sparse row* (CSR), where the values of the same row are stored contiguously such that there is no need to store an individual row index per value. With the CSR, each NNZ needs a single integer, which is the column index, decreasing the memory footprint up to 33% compared to COO/IJV.

Following this idea, plenty of storage formats have been proposed. Many of them also tried to obtain a format that can be computed efficiently for a given architecture.

Some of the first block-based formats are the block compressed sparse row storage (BCSR) [20] and its extensions to larger blocks of variable dimension [24,13] or to unaligned block compressed sparse row (UBCSR) [23]. However, in these formats, the blocks have to be filled with zeros to be full. For these formats, the blocks were aligned (the upper-left corner of the blocks start at a position multiple of the block size). While the blocks are well suited for vectorization, the extra zeros can dramatically decrease the performance.

More recent work has focused on GPUs and manycore architectures. Among them, the references are the ELLAPACK format [17], SELL-C-σ [15] defined as a variant of Sliced ELLPACK, and the CSR5 [16] format that we used as reference in our previous study.

The Cuthill-McKee method from [8] is a well-known technique for improving the bandwidth of a matrix to have good properties for LU decomposition. It does so by applying a breadth-first algorithm on a graph which represents the matrix structure. While the aim of this algorithm is not to improve the SpMV performance, the generated matrices may have better data locality.

Another method [20] has been specifically designed to increase the number of contiguous values in rows and/or columns. This method works by creating a graph from a matrix, where each column (or row) is a vertex and all the vertices are connected with weighted edges. The weights represent the interest of putting two columns (or rows) contiguously. By solving the traveling salesman problem (TSP) to obtain a path that goes through all the nodes but only once and that minimizes the total weight of the path, we can find a permutation of the sparse matrix that should be better divided into blocks. This means that we should have fewer blocks and the blocks should contain more NNZ elements.

Several updates to the method have been presented in [23,19,5] using different formulas. While the current study does not focus on the permutation of matrices, it is worth noting that enhancing the matrix's shape, as in other approaches, would likely lead to improved kernel efficiency by reducing the number of blocks.

2.4. SPC5

The SPC5 format consists in using a block scheme without adding additional zeros. SPC5 can be seen as an extension of the CSR format, but where the values of each row are split into blocks. Each block starts with a NNZ at column c and includes the next NNZ values until column c+*VEC_SIZE-1* if they exist. Consequently, in the worst case a block contains a single value, and in the best case *VEC_SIZE* values. Then, for each block, we use a mask of bits to indicate which of the NNZ values in the block exist. As a result, in a poor configuration, SPC5 will have the same memory footprint as the CSR plus one bit mask per NNZ. On the other hand, in the other extreme scenario, SPC5 will save one integer for each value added to a block since we can retrieve the corresponding column index from c and the position of the NNZ in the block.

The SPC5 format has been extended so that a block is mapped to several rows. This is helpful if there are NNZ values closed (NNZ of consecutive rows that have closed column index) such that the values loaded from the vector x can be used more than once and that the column index of the block is reused for more NNZ.

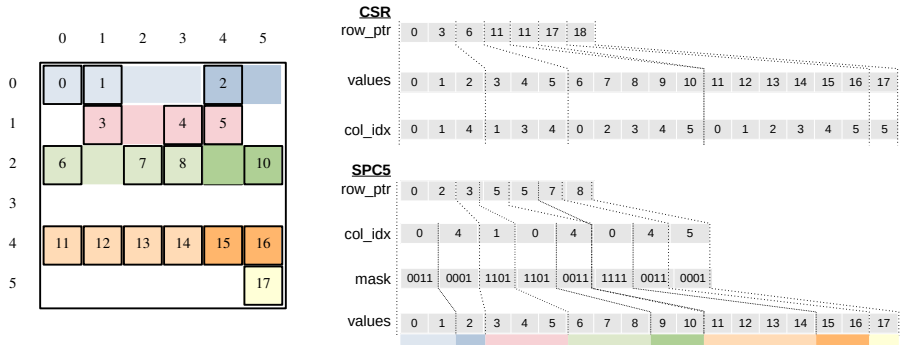


Fig. 2. Illustration of the CSR and SPC5 formats. In this figure, we use the $\beta(1,4)$ format, which means that each block is on a single row and of length 4. In the CSR format, the original *row_idx* array is compacted to have a single index per block instead of an index per NNZ. The mask array indicates the positions of the next NNZ in the block, and the corresponding column index can be obtained by summing the block's column index with the corresponding bit position in the mask

In the rest of the document, we refer to β(r, VEC_SIZE) when the blocks are over *r* rows and of length *VEC_SIZE*. In the original study, we were also using blocks of $VEC\_SIZE/2$ but not in the current study. We give an example of CSR and SPC5 in Figure 2.

3. SPC5 Implementation

We provide the SPC5 SpMV pseudocode in Algorithm 1.

First, we initialize an index to progress in the array of NNZ values, at line 3. Since we have no way to know where the values of a given block are located in the array, we have to increase the index with the number of values of each block that has been computed. This is visible at line 16 for the scalar version, line 21 for AVX512, and line 28 for SVE.

At line 5, we iterate over the rows with a step r. For each row segment, we iterate over the blocks at line 8. For each block, we load its column index at line 9.

Then, we process each row of the block. We start by getting the mask, at line 11, that tells us what are the NNZ that exist in the row of the block. A naive implementation consists in testing the existence of each possible value, at line 14, and performing the computation if needed, at line 15. However, this loop over the NNZ can be done with a few instructions in AVX512, at line 20. In this case, we load a vector from x that matches the column index, and expand the NNZ from the value array into a vector.

The expand operation does not exist in SVE. Consequently, the same behavior is obtained using different instructions. First, we need a filter vector that contains 2^i at position i, see line 4. Then, we compute a binary *and* operation between the filter vector and the mask, such that only the position for which a NNZ exist will not be zero. We do this operation at line 23 and get the active elements at line 24. Second, instead of expanding

the NNZ values, as with AVX512, we compact the values from x, at line 26. Doing so, we can simply load the right number of NNZ and leave them contiguous in the resulting vector, before performing the computation. A schematic view of the two approaches is provided in Figure 3.

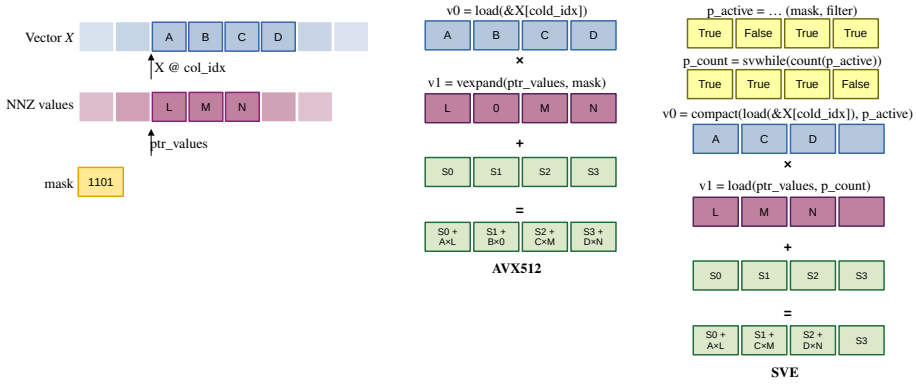


Fig. 3. Illustration of the loading and computation of one row of a block. The mask is represented from the most significant bit (MSB) to the least significant bit (LSB), whereas the vector elements are represented from the first element to the last element. Hence, the 1s in the mask 1101 correspond to the elements N, M, and L (in this order)

Finally, we update y at the end of the algorithm (at line 32) for each of the rows that have been proceeded.

3.1. Optimizing the Loading of x

In AVX512, the values from x are loaded into a SIMD vector without pruning. This means that no matter how many NNZ are in the block or how many values we need from x, VEC_SIZE values will be loaded from memory. It is possible to prune/filter the values, but this will imply an extra cost (i.e., using a more expensive instruction like gather) and would certainly have no benefit as the AVX512 SIMD load instruction is translated into an efficient memory transaction. Consequently, in AVX512, the main optimization consists in loading the values from x once for all the rows of a block, which allows accessing the memory once and using the resulting vector r times.

ALGORITHM 1: SpMV for a matrix *mat* in format $\beta(r, c)$. The lines in blue • are to compute in scalar and have to be replaced by the line in green • to have the vectorized equivalent in SVE or in red • with AVX

Input: x : vector to multiply with the matrix. mat : a matrix in the block format $\beta(r, c)$. r, c : the size of the blocks.
Output: y : the result of the product.

```
1  function spmv(x, mat, r, c, y)
2      // Index to access the array's values
3      idxVal ← 0
4      filter ← [1 << 0, ...., 1 << VS-1]
5      for idxRow ← 0 to mat.numberOfRows-1 inc by r do
6          sum[r] ← init_scalar_array(r, 0)
7          sum[r] ← init_simd_array(r, 0)
8          for idxBlock ← mat.block_rowptr[idxRow/r] to mat.block_rowptr[idxRow/r+1]-1
            do
9              idxCol ← mat.block_colidx[idxBlock]
10             for idxRowBlock ← 0 to r do
11                 valMask ← mat.block_masks[idxBlock × r + idxRowBlock]
12                 // The next loop can be vectorized with vexpand
13                 for k ← 0 to c do
14                     if bit_shift(1 , k) BIT_AND valMask then
15                         sum[idxRowBlock] += x[idxCol+k] * mat.values[idxVal]
16                         idxVal += 1
17                     end
18                 end
19                 // To replace the k-loop AVX512
20                 sum[idxRowBlock] += simd_load(x[idxCol]) *
                    simd_vexpand(mat.values[idxVal], valMask)
21                 idxVal += popcount(valMask)
22                 // To replace the k-loop SVE
23                 mask_vec = svand(svdup(valMask), filter)
24                 active_elts = svcmpne(mask_vec, 0)
25                 increment = count(active_elts)
26                 xvals = svcompact(active_elts, simd_load(active_elts, x[idxCol]))
27                 block = simd_load(svwhile(0, increment), mat.values[idxVal])
28                 idxVal += increment
29                 sum[idxRowBlock] += block * xvals
30             end
31         end
32         for idxRowBlock ← idxRow to r+idxRow do
33             y[idxRowBlock] += sum[r]
34             y[idxRowBlock] += simd_hsum(sum[r])
35         end
36     end
```

With SVE, it is different and we have mainly three alternatives:

- Loading the values from x without pruning, as with AVX512, and then compacting the obtained vector for each row of the block. We refer to this strategy as *single x load*.
- Loading a different vector for each row of the block, as shown in Algorithm 1. We refer to this strategy as *partial x load*.
- Combining the predicates of several rows of the block by merging their predicates/masks to load all the values that are needed by the block, but not more. In our study, we left this approach aside, as different tests we have conducted have shown poor performance.

The performance gains we can expect from the different strategies depend on the way the load is actually performed by the hardware. In fact, the main point is to know if the hardware can make faster memory transactions when some elements of the predicate vector used in the load are false. If the hardware actually loads VEC_SIZE values from the memory but then only copies the ones that have their corresponding predicate value true to the SIMD vector, we should not expect any benefit. Moreover, ARM SVE can be seen as an interface, hence it can be implemented by the hardware differently such that the behavior can also change from one vendor to another.

3.2. Optimizing the Writing of the Result in y

In the SIMD implementation, we have to perform the reduction (i.e., the horizontal sum) of r vectors to add them to y and store the result.

A straightforward approach is to call a single reduction instruction per vector, as both AVX512 and SVE support such an operation. However, this means that we will perform r individual summations between the reduced values and the values from y, and that we will also access the memory r times (actually $2 \times r$, since we load values from y, perform the summation, and write back to y).

To avoid this, we propose a possible optimization that consists in performing the reduction of all the vectors manually to obtain a single vector as output, and then performing a vectorial summation with y.

With AVX512, this manual multi-reduction can be implemented by playing with AVX and SSE registers and using the horizontally add adjacent pairs instruction (*hadd*). The operation is done without any loop.

With SVE, we do this using odd/even interleave instructions (*svuzp1* and *svuzp2*). In this case, we need a loop because the length of the vectors is unknown at compile time.

4. Performance Study

4.1. Configuration

We assess our method on two configurations:

- Fujitsu-SVE: it is an ARMv8.2 A64FX - Fujitsu with 48 cores at 1.8 GHz and 512-bit SVE [12], i.e. a vector can contain 16 single precision floating-point values or 8

double precision floating-point values. The node has 32 GB HBM2 memory arranged in four core memory groups (CMGs) with 12 cores and 8GB each, 64KB private L1 cache, and 8MB shared L2 cache per CMG. We use the GNU compiler 10.3.0.

– Intel-AVX512: it is a 2 $\times$ 18-core Cascade Lake Intel Xeon Gold 6240 at 2.6 GHz with AVX-512 (Advanced Vector 512-bit, Foundation, Conflict Detection, Byte and Word, Doubleword and Quadword Instructions, and Vector Length). The main memory consists of 190 GB DRAM memory arranged in two NUMA nodes. Each CPU has 18 cores with 32KB private L1 cache, 1024KB private L2 cache, and 25MB shared L3 cache. We use the GNU compiler 11.2.0 and the MKL 2022.0.2.

4.2. Test Cases

We evaluated the performance of our proposed SPC5 SpMV kernels [4] on a set of sparse matrices taken from the University of Florida sparse matrix collection (UF Collection) [9]. It includes a dense matrix of dimension 2048. The results of the dense matrix are expected to provide an upper bound on the performance of the kernels, as all blocks will be full. Of course, our kernels are not well-designed or optimized for a dense matrix-vector product. The properties of the matrices are given in Table 1.

We evaluated the performance of four kernels: β(1, VS), β(2, VS), β(4, VS), and β(8, VS), where VS is the vector size. We also provide the filling of the blocks when we format the matrices to the corresponding block sizes. The filling can be up to 80% for *nd6k* but as low as 1% for *wikipedia-20060925*. (It is obviously 100% for the dense matrix.)

We performed the computation in single precision (*float/f32*) and double precision (*double/f64*).

The original AVX implementation was written in assembly language, while our current implementation is written in C++ with intrinsic functions. Consequently, the AVX and SVE kernels have very similar structures.

4.3. Results

The results are organized as follows. In the first part, we evaluate the difference of using the manual multi-reduction (described in Section 3.2) vs. the native SIMD horizontal summation in Figures 2 (a) and 2 (b), for Fujitsu-SVE and Intel-AVX respectively. For Fujitsu-SVE, we also evaluate the use of full vector load of x (described in Section 3.2). Then, we provide the detailed results for all the matrices and the selected configuration in Figures 5 and 7. Finally, we provide an overview of the parallel performance when the computation is naively divided among the threads in Figure 8.

Comparisons of the Different Optimizations. We provide the performance results of the different implementations in Table 2.

In Table 2 (a), we evaluate the use of manual multi-reduction and the single load of the x vector for the Fujitsu-SVE architecture. There is no difference in most cases, and it is always meaningless for β(2,VS). The β(4,VS) and β(8,VS) kernels react differently with the optimizations and improve slightly. The small change in using the multi-reduction comes from the small difference in latencies between the two approaches. The *reduce*

[4] The code is available at `https://gitlab.inria.fr/bramas/spc5`

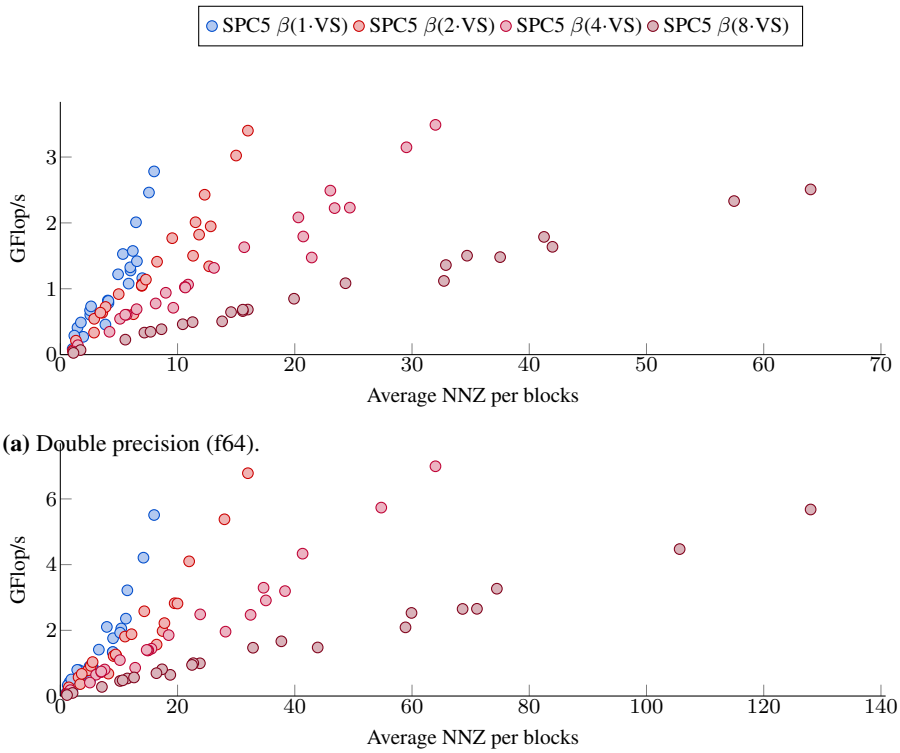
Table 1. Matrix set for computation and performance analysis. We provide the percentage of filling of the blocks for double (left) and single (right) precision

Name	Dim	NNZ	$\frac{NNZ}{N_{rows}}$	$\beta(1,VS)$	$\beta(2,VS)$	$\beta(4,VS)$	$\beta(8,VS)$
bundle	513351	20208051	39.365	72% | 55%	70% | 54%	64% | 50%	51% | 46%
CO	221119	7666057	34.6694	18% | 9%	18% | 9%	17% | 9%	16% | 8%
crankseg	63838	14148858	221.637	66% | 49%	59% | 44%	49% | 37%	38% | 29%
dense	2048	4194304	2048	100% | 100%	100% | 100%	100% | 100%	100% | 100%
dielFilterV2real	1157456	48538952	41.9359	31% | 20%	22% | 14%	15% | 10%	11% | 7%
Emilia	923136	41005206	44.4195	50% | 31%	43% | 28%	34% | 24%	24% | 18%
FullChip	2987012	26621990	8.91258	24% | 13%	17% | 10%	13% | 7%	8% | 5%
Hook	1498023	60917445	40.6652	51% | 34%	43% | 29%	33% | 23%	24% | 17%
in-2004	1382908	16917053	12.233	48% | 31%	38% | 25%	30% | 19%	21% | 14%
ldoor	952203	46522475	48.8577	87% | 55%	79% | 51%	67% | 44%	51% | 34%
mixtank	29957	1995041	66.5968	31% | 20%	24% | 16%	17% | 11%	12% | 8%
nd6k	18000	6897316	383.184	80% | 71%	76% | 68%	71% | 64%	64% | 58%
ns3Da	20414	1679599	82.2768	14% | 7%	8% | 4%	4% | 2%	2% | 1%
pdb1HYS	36417	4344765	119.306	77% | 65%	72% | 60%	63% | 54%	54% | 46%
pwtk	217918	11634424	53.389	74% | 56%	74% | 55%	73% | 54%	65% | 53%
RM07R	381689	37464962	98.1557	61% | 41%	51% | 34%	40% | 28%	31% | 25%
Serena	1391349	64531701	46.3807	51% | 34%	43% | 29%	33% | 23%	24% | 17%
Si41Ge41H72	185639	15011265	80.8627	32% | 18%	31% | 17%	28% | 15%	22% | 13%
Si87H76	240369	10661631	44.3553	21% | 11%	21% | 11%	20% | 10%	17% | 9%
spal	10203	46168124	4524.96	74% | 69%	45% | 37%	25% | 23%	13% | 12%
torso1	116158	8516500	73.3182	81% | 63%	80% | 62%	77% | 59%	58% | 55%
TSOPF	38120	16171169	424.217	94% | 88%	93% | 87%	92% | 85%	89% | 82%
wikipedia-20060925	2983494	37269096	12.4918	13% | 6%	6% | 3%	3% | 1%	1% | 0%

instruction (*addv*) has a latency of 12 cycles [12]. Our multi-reduction has a latency of around 96 cycles for two vectors (considering the following latencies *uzp1* 6, *uzp2* 6, *whilelt* 4 and full *vadd* 22), and it is almost the same cost for 4 or 8 vectors. Disabling the x load optimization almost always degrades performance for the β(4,VS) kernel but seems to improve the performance for the β(8,VS). This is surprising, as we would expect that the larger the blocks would be, the more benefit we would have to load the vector from x completely. From our understanding, the cost of a load depends on the location of the data it requests but not on the fact that the data it requests could be located in different cache lines. This explains why the optimization has a limited impact, as a partial load will move the data to the cache and speedup the next partial loads. Since the β(4,VS) is faster than β(8,VS), we consider the best configuration to be with both optimizations turned on.

In Table 2 (b), we evaluate the use of manual multi-reduction on Intel-AVX512 architecture. The performance increases slightly with the use of manual multi-reduction in some cases. For instance, the best performance on average is obtained with β(4,VS) and for this configuration, using the manual multi-reduction has no impact (double) or increases the speedup by 0.1 (float). The explanation is as follows: the *reduce* intrinsic function (*_mm512_reduce_add_ps*) is not actually a real hardware instruction, but a call to a function provided by the compiler [22]. Its implementation [5] is very similar to our manual multi-reduction, with the main difference being that we try to factorize the in-

[5] https://github.com/gcc-mirror/gcc/blob/9d7e19255c06e05ad791e9bf5aefc4783a12c4f9/gcc/config/i386/avx512fintrin.h#L15928



(a) Double precision (f64).

(b) Single precision (f32).

Fig. 4. Performance in Giga Flop per second for sequential computation in double and single precision for our SPC5 kernels on Fujitsu-SVE architecture for all the matrices of the test set

structions to reduce several SIMD vectors at the same time. This allows us to obtain a SIMD vector as output, which can then be used to update y with vectorized instructions. In the end, the performance difference between the two approaches is limited. However, for the rest of the study, we consider that the best Intel-AVX512 configuration is to use manual multi-reduction.

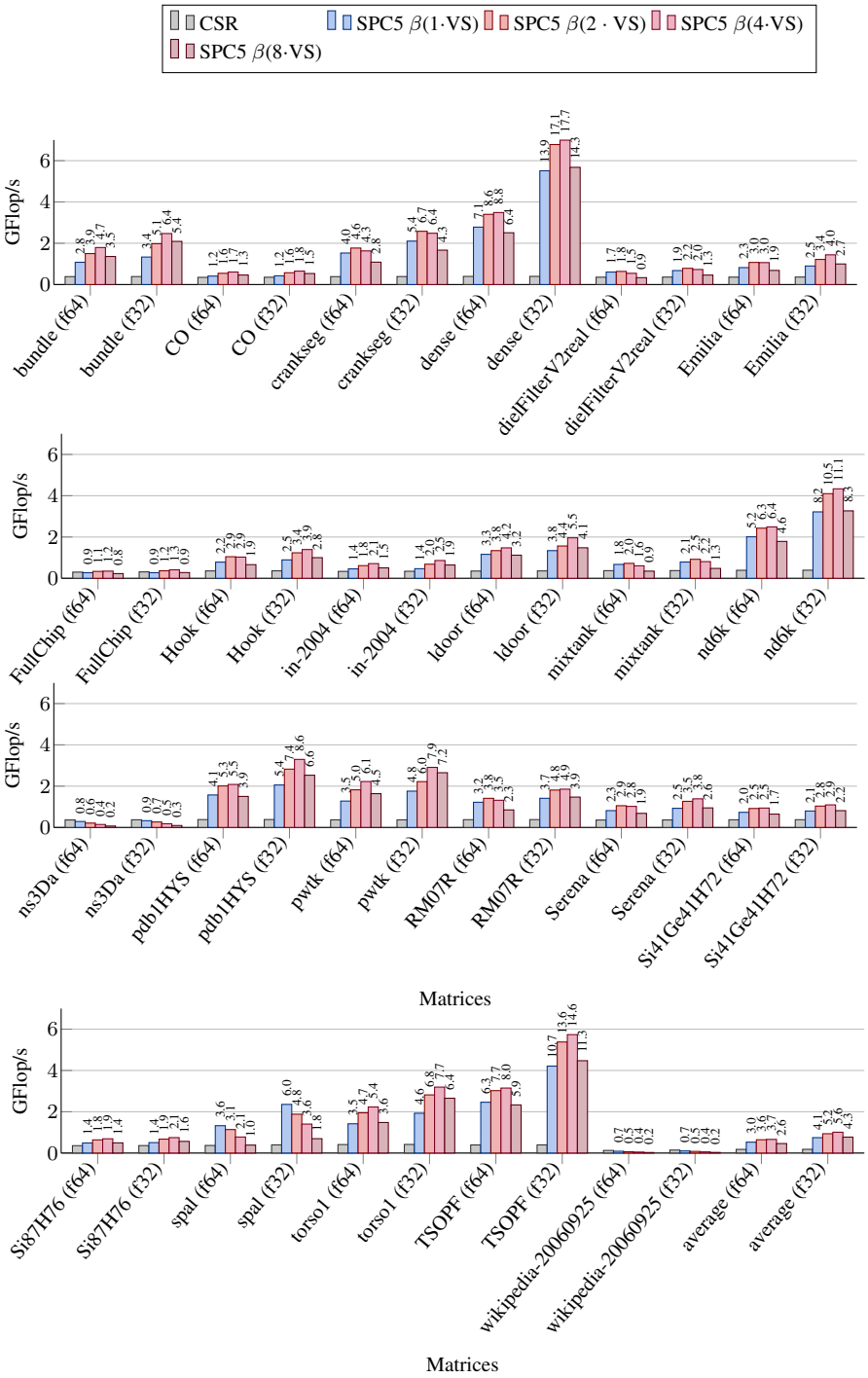


Fig. 5. Performance in Giga Flop per second for sequential computation in double and single precision for our SPC5 kernels on Fujitsu-SVE architecture. Speedup of SPC5 is computed against the scalar sequential version and written above the bars

	x load /	CSR		β(1,VS)		β(2,VS)		β(4,VS)		β(8,VS)	
	reduction	f64	f32	f64	f32	f64	f32	f64	f32	f64	f32
CO	Yes/Yes	0.3	0.4	0.4 [x1.2]	0.4 [x1.2]	0.5 [x1.6]	0.6 [x1.6]	0.6 [x1.7]	0.6 [x1.8]	0.5 [x1.3]	0.5 [x1.5]
	Yes/No										0.5 [x1.4]
	No/Yes					0.5 [x1.5]		0.5 [x1.5]	0.6 [x1.6]	0.5 [x1.5]	0.6 [x1.6]
	No/No					0.5 [x1.5]		0.5 [x1.5]	0.6 [x1.6]	0.5 [x1.5]	0.6 [x1.6]
dense	Yes/Yes	0.4	0.4	2.8 [x7.1]	5.5 [x13.9]	3.4 [x8.6]	6.8 [x17.1]	3.5 [x8.8]	7.0 [x17.7]	2.5 [x6.4]	5.7 [x14.3]
	Yes/No						6.9 [x17.5]		6.5 [x16.4]	2.5 [x6.3]	6.4 [x16.0]
	No/Yes			2.8 [x7.0]		3.3 [x8.5]	6.8 [x17.2]	3.0 [x7.7]	6.4 [x16.1]	3.1 [x7.9]	6.4 [x16.1]
	No/No			2.8 [x7.0]	5.5 [x13.8]	3.3 [x8.5]	6.8 [x17.2]	3.0 [x7.7]	6.4 [x16.2]	3.1 [x7.9]	6.4 [x16.1]
nd6k	Yes/Yes	0.4	0.4	2.0 [x5.2]	3.2 [x8.2]	2.4 [x6.3]	4.1 [x10.5]	2.5 [x6.4]	4.3 [x11.1]	1.8 [x4.6]	3.3 [x8.3]
	Yes/No										
	No/Yes							2.2 [x5.8]	3.9 [x9.9]	2.0 [x5.2]	3.6 [x9.3]
	No/No							2.2 [x5.8]	3.9 [x9.9]	2.0 [x5.2]	3.6 [x9.3]
average	Yes/Yes	0.2	0.2	0.5 [x3.0]	0.7 [x4.1]	0.6 [x3.6]	0.9 [x5.2]	0.7 [x3.7]	1.0 [x5.6]	0.5 [x2.6]	0.8 [x4.3]
	Yes/No								1.0 [x5.5]		
	No/Yes					0.6 [x3.5]	0.9 [x5.0]	0.6 [x3.3]	0.9 [x4.8]	0.5 [x2.9]	0.8 [x4.6]
	No/No					0.6 [x3.5]	0.9 [x5.0]	0.6 [x3.3]	0.9 [x4.8]	0.5 [x2.9]	0.8 [x4.6]

(a) Fujitsu-SVE

	x load /	CSR		MKL		β(1,VS)		β(2,VS)		β(4,VS)		β(8,VS)	
	reduction	f64	f32	f64	f32	f64	f32	f64	f32	f64	f32	f64	f32
CO	No/Yes	1.4	1.9	1.9 [x1.3]	2.3 [x1.2]	1.3 [x0.9]	1.4 [x0.8]	1.6 [x1.1]	1.9 [x1.0]	1.7 [x1.2]	1.9 [x1.0]	1.6 [x1.1]	1.9 [x1.0]
	No/No												
dense	No/Yes	1.2	1.3	2.3 [x1.9]	3.6 [x2.8]	3.7 [x3.0]	8.3 [x6.4]	4.1 [x3.4]	9.5 [x7.3]	4.3 [x3.6]	11.2 [x8.6]	4.4 [x3.6]	10.8 [x8.3]
	No/No						9.4 [x7.2]		10.6 [x8.1]	4.2 [x3.4]	11.0 [x8.5]		11.0 [x8.5]
nd6k	No/Yes	1.2	1.4	2.2 [x1.8]	2.8 [x2.0]	2.9 [x2.4]	6.2 [x4.5]	3.4 [x2.8]	7.3 [x5.4]	3.4 [x2.8]	7.4 [x5.4]	3.4 [x2.8]	7.4 [x5.4]
	No/No				2.8 [x2.1]		6.3 [x4.6]		7.3 [x5.3]		7.6 [x5.6]		7.1 [x5.2]
average	No/Yes	0.6	0.8	0.9 [x1.5]	1.2 [x1.6]	1.0 [x1.6]	1.7 [x2.3]	1.2 [x1.8]	2.0 [x2.6]	1.2 [x1.8]	2.0 [x2.7]	1.1 [x1.7]	1.9 [x2.5]
	No/No					1.1 [x1.7]	1.8 [x2.4]				2.0 [x2.6]		1.8 [x2.4]

(b) Intel-AVX512

Table 2. Performance in Giga Flop per second for sequential computation in double and single precision for our SPC5 kernels on Fujitsu-SVE and Intel-AVX512 architectures. Speedup of SPC5 is computed against the scalar sequential version, we print the values only when there is a difference with the first version (above one digit difference in the speedup). We provide the results for the CO, dense and nd6k matrices, and the average based on all the matrices from the test set. We compare the the loading of full x vectors per block (SVE and AVX512), and the use of manual multi-reduction against vectorial reduction (SVE only). The scalar and β(1,VS) and MKL versions are expected to remain unchanged, differences for these kernels are from noise.

Best Configuration Detailed Results. We provide the complete results for Fujitsu-SVE in Figures 5 and 4. The results for Intel-AVX are shown in Figures 7 and 6.

In Figure 5, we can see that the performance of the SPC5 kernels is clearly related to the block filling. The performance model can be described as a constant cost per block that seems independent of the number of blocks or the number of NNZ. This means that the performance can be easily predicted from the block filling.

We also note that the performance increases as we increase the size of the blocks up to 4×VS, but then it decreases for $8 \times VS$. This is more visible in Figure 5, where β(8, VS) is the slowest SPC5 kernel in most cases.

The behaviors in single and double precision are similar. For some matrices, such as ns3Da, SPC5 is even slower than a simple CSR implementation. This means that the overhead of using vectorial instructions outweighs the benefits of vectorization since the block filling is very low.

The computation on the matrix TSOPF, which has a very high block filling, achieves performance almost equivalent to the dense matrix case. Finally, we can see the average performance in Figure 5 (last bars). While the speedup against CSR is significant, the raw performance is low compared to the peak performance of the machine.

The results for Intel-AVX are slightly different. In Figure 7, we can see that while there is a correlation between the block filling and the performance, the relationship is less clear than for Fujitsu-SVE.

We also note that the performance increases with the block size, such that the best performance is achieved with β(8, VS). This is even more visible in Figure 7.

Contrary to Fujitsu-SVE, the performance obtained for TSOPF is not close to the dense matrix case. This means that while the blocks are almost full, the fact that we have to jump over the vector x has a negative impact on the performance.

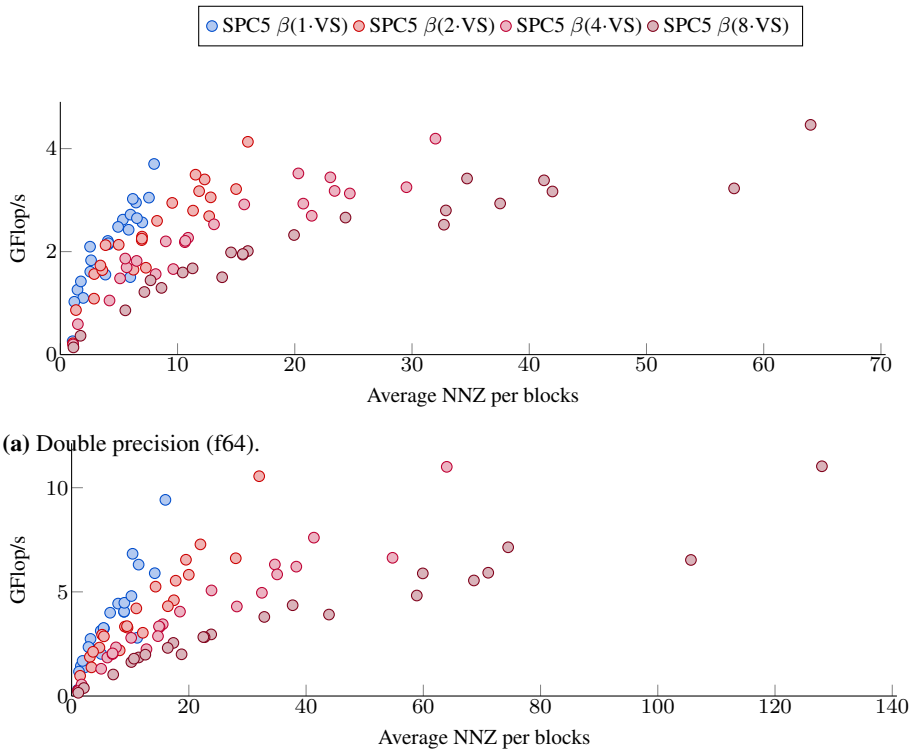
The performance of SPC5 on Intel-AVX is higher than those obtained with Fujitsu-SVE for almost all matrices. Finally, SPC5 is faster than the Intel MKL CSR kernel for most matrices, but can be slower if there are less than two values per block.

Parallel Performance Overview. In Figure 8, we provide the results for the parallel executions. For the Fujitsu-SVE hardware, Figure 8a, for some matrices the speedup is above 42 (the number of cores). This is possible because the matrices are split and allocated by the threads such that each thread has its data on the memory nodes that correspond to its CPU core. In addition, the split of the matrices and the use of all the cores can result in using the cache more efficiently.

For the Intel-AVX512 hardware, Figure 8b, the executions on the dense matrix have poor performance for small blocks. This is clear that the x vector will be fully loaded for each row, such that the cache performance is tied to the final execution performance. We can notice that the speedup around 15 is far from the number of CPU cores (36). The workload balance between the threads is similar to the Fujitsu-SVE configuration, therefore, we consider that the difference comes from the memory organization and use.

5. Conclusion

We have presented a new version of our SPC5 framework, which remains efficient on architectures with AVX512 and is now compatible with ARM architectures with SVE. The



(a) Double precision (f64).

(b) Single precision (f32).

Fig. 6. Performance in Giga Flop per second for sequential computation in double and single precision for our SPC5 kernels on Intel-AVX architecture for all the matrices of the test set

same sparse matrix format can be used to target both ISA, allowing for interoperability and the use of a single framework on x86 and ARM-based architectures.

The SPC5's SpMV kernels are implemented differently, as they rely on an expand mechanism of the NNZ (AVX512) or a compaction of the x vector (SVE). The performances we obtained are usually higher than a simple CSR kernel if there is more than a single NNZ per block. The $\beta(1,*)$ format has a low conversion cost as it leaves the array of NNZ unchanged compared to CSR, which makes it easy to plug in existing CSR-based applications.

In a future work, we would like to investigate if we could use a hybrid format, i.e., a format where we could have blocks of different sizes including blocks of scalar, to avoid using vectorial instructions when there is no benefit.

Acknowledgments. This work used the Isambard 2 UK National Tier-2 HPC Service [6] operated by GW4 and the UK Met Office, which is an EPSRC project (EP/T022078/1). We also used the

[6] `http://gw4.ac.uk/isambard/`

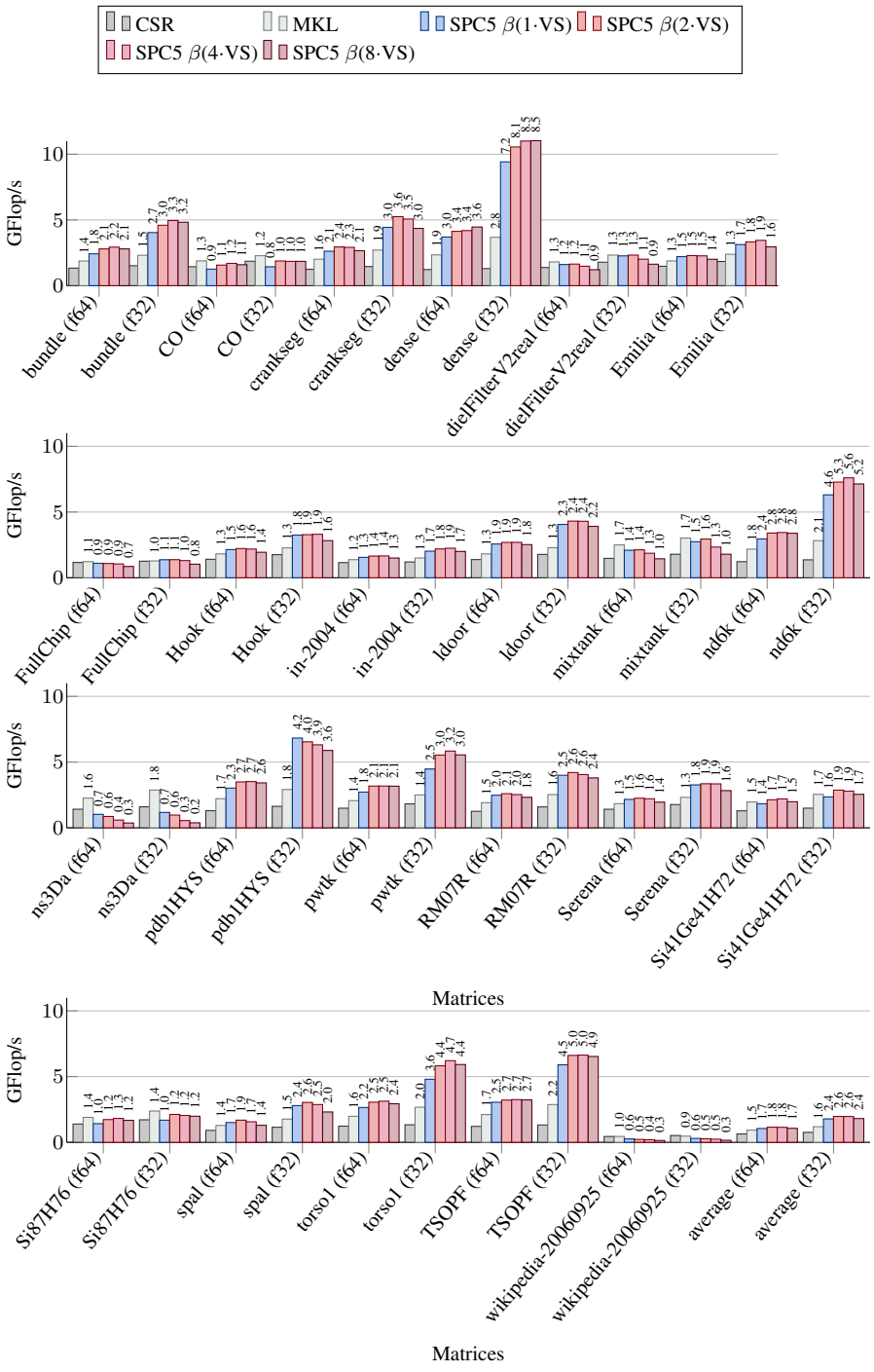
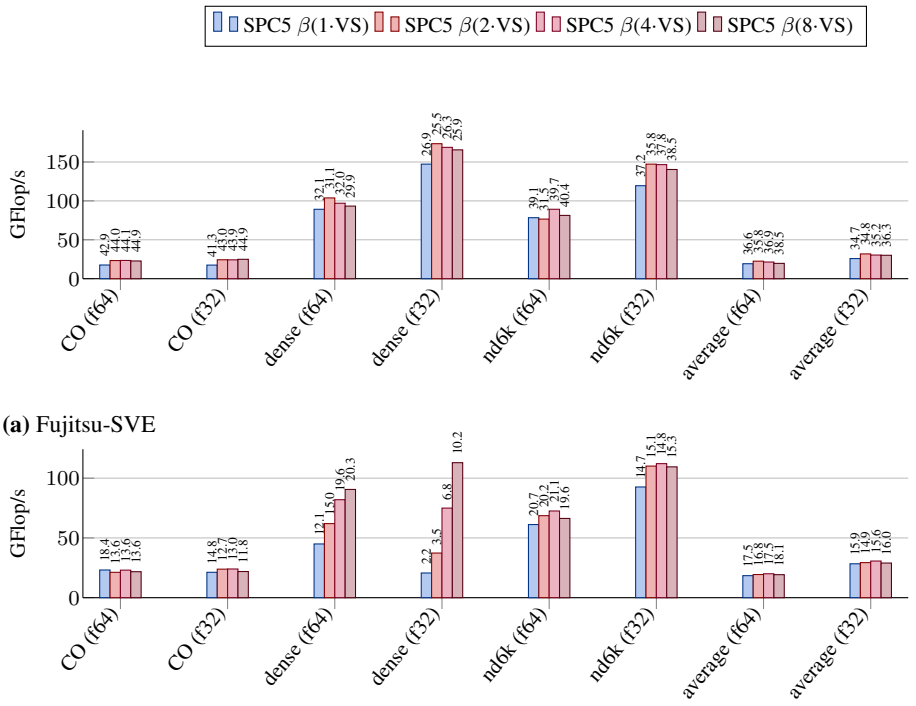


Fig. 7. Performance in Giga Flop per second for sequential computation in double and single precision for our SPC5 kernels on Intel-AVX512 architecture. Speedup of SPC5 is computed against the scalar sequential version and written above the bars



(a) Fujitsu-SVE

(b) Intel-AVX512

Fig. 8. Performance in Giga Flop per second for parallel computation in double and single precision for our SPC5 kernels on Fujitsu-SVE and Intel-AVX512 architectures. Speedup of parallel SPC5 is computed against the same sequential version and written above the bars. We provide the results for the CO, dense and nd6k matrices, and the average based on all the matrices from the test set.

PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Universite de Bordeaux, Bordeaux INP and Conseil Regional d'Aquitaine [7]. In addition, this work used the Farm-SVE library [6].

References

1. Alappat, C., Meyer, N., Laukemann, J., Gruber, T., Hager, G., Wellein, G., Wettig, T.: Ecm modeling and performance tuning of spmv and lattice qcd on a64fx. arXiv preprint arXiv:2103.03013 (2021)
2. AOKI, R., MURAO, H.: Optimization of x265 encoder using arm sve
3. ARM: Arm architecture reference manual supplement, the scalable vector extension (sve), for armv8-a. https://developer.arm.com/documentation/ddi0584/ag/, accessed: July 2020 (version Beta)

[7] `https://www.plafrim.fr`

4. ARM: Arm c language extensions for sve. https://developer.arm.com/documentation/100987/0000, accessed: July 2020 (version 00bet1)
5. Bramas, B.: Optimization and parallelization of the boundary element method for the wave equation in time domain. Ph.D. thesis, Université de Bordeaux (2016)
6. Bramas, B.: Farm-SVE: A scalar C++ implementation of the ARM® Scalable Vector Extension (SVE) (Jul 2020), https://inria.hal.science/hal-02906179
7. Bramas, B., Kus, P.: Computing the sparse matrix vector product using block-based kernels without zero padding on processors with avx-512 instructions. PeerJ Computer Science 4, e151 (Apr 2018), https://doi.org/10.7717/peerj-cs.151
8. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th national conference. pp. 157–172. ACM (1969)
9. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS) 38(1), 1 (2011)
10. Domke, J.: A64fx–your compiler you must decide! arXiv preprint arXiv:2107.07157 (2021)
11. Flynn, M.J.: Very high-speed computing systems. Proceedings of the IEEE 54(12), 1901–1909 (1966)
12. Fujitsu: A64fx microarchitecture manual. https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.8.1.pdf (2022), accessed on 26 July 2023 from https://www.fujitsu.com/global/products/computing/servers/supercomputer/a64fx/
13. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. International Journal of High Performance Computing Applications 18(1), 135–158 (2004)
14. Kodama, Y., Odajima, T., Matsuda, M., Tsuji, M., Lee, J., Sato, M.: Preliminary performance evaluation of application kernels using arm sve with multiple vector lengths. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 677–684 (2017)
15. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Bishop, A.R.: A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. SIAM Journal on Scientific Computing 36(5), C401–C423 (2014)
16. Liu, W., Vinter, B.: Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In: Proceedings of the 29th ACM on International Conference on Supercomputing. pp. 339–350. ACM (2015)
17. Liu, X., Smelyanskiy, M., Chow, E., Dubey, P.: Efficient sparse matrix-vector multiplication on x86-based many-core processors. In: Proceedings of the 27th international ACM conference on International conference on supercomputing. pp. 273–282. ACM (2013)
18. Meyer, N., Georg, P., Pleiter, D., Solbrig, S., Wettig, T.: Sve-enabling lattice qcd codes. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp. 623–628 (2018)
19. Pichel, J.C., Heras, D.B., Cabaleiro, J.C., Rivera, F.F.: Performance optimization of irregular codes based on the combination of reordering and blocking techniques. Parallel Computing 31(8), 858–876 (2005)
20. Pinar, A., Heath, M.T.: Improving performance of sparse matrix-vector multiplication. In: Proceedings of the 1999 ACM/IEEE conference on Supercomputing. p. 30. ACM (1999)
21. Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., Reid, A., Rico, A., Walker, P.: The arm scalable vector extension. IEEE Micro 37(2), 26–39 (Mar 2017), https://doi.org/10.1109/MM.2017.35
22. Varela, M.H.: Manycore Architectures and SIMD Optimizations for High Performance Computing. Ph.D. thesis, Universidade da Coruña (2022)
23. Vuduc, R.W., Moon, H.J.: Fast sparse matrix-vector multiplication by exploiting variable block structure. In: High Performance Computing and Communications, pp. 807–816. Springer (2005)
24. Vuduc, R.W.: Automatic performance tuning of sparse matrix kernels. Ph.D. thesis, Citeseer (2003)

25. Wan, X., Gu, N., Su, J.: Accelerating level 2 blas based on arm sve. In: 2021 4th International Conference on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE). pp. 1018–1022 (2021)

Evann Regnault holds a Bachelor of Science degree and is currently pursuing a Master of Science in Computer Science at the University of Strasbourg. His research focuses on low-level optimizations and software engineering for High-Performance Computing (HPC).

Bérenger Bramas is a permanent research scientist at Inria Nancy and the ICube laboratory. He possesses dual Master of Science degrees: one in Computer Science from the University Blaise Pascal and the other in Software Engineering from ISIMA. He earned his Ph.D. from Bordeaux University. His research focuses on High-Performance Computing (HPC), including optimizations, complex algorithm development, accelerators, and software engineering for HPC. Additionally, his work involves scientific computing, runtime systems, scheduling, and the development of tools for automatic vectorization and parallelization.

Received: August 19, 2023; Accepted: December 12, 2023.