Optimizing Sparse Matrix-Vector Multiplication in NEC SX-Aurora Vector Engine

Constantino Gómez[1], Marc Casas[1], *Filippo Mantovani*[1], *and* Erich Focht[2]

[1]Barcelona Supercomputing Center, {first.last}@bsc.es
[2]NEC HPC Europe (Germany), {first.last}@EMEA.NEC.COM

August 14, 2020

Abstract

Sparse Matrix-Vector multiplication (SpMV) is an essential piece of code used in many High Performance Computing (HPC) applications. As previous literature shows, achieving efficient vectorization and performance in modern multi-core systems is nothing straightforward. It is important then to revisit the current state-of-the-art matrix formats and optimizations to be able to deliver high performance in long vector architectures. In this tech-report, we describe how to develop an efficient implementation that achieves high throughput in the NEC Vector Engine: a 256 element-long vector architecture. Combining several pre-processing and kernel optimizations we obtain an average 12% improvement over a base SELL-*C*-σ implementation on a heterogeneous set of 24 matrices.

1 Introduction

The Sparse Matrix-Vector (SpMV) product is a ubiquitous kernel in the context of High-Performance Computing (HPC). For example, discretization schemes like the finite differences or finite element methods to solve Partial Differential Equations (PDE) produce linear systems with a highly sparse matrix. Such linear systems are typically solved via iterative methods, which require an extensive use of SpMV across the whole execution. In addition, emerging workloads from the data-analytics area also require the manipulation of highly irregular and sparse matrices via SpMV. Therefore, the efficient execution of this fundamental linear algebra kernel is of paramount importance.

The performance of the SpMV $y = Ax$ is strongly correlated to several factors. First accesses to data structures containing the A matrix and the y vector are typically regular, which means that they benefit from hardware resources like memory bandwidth capacity and structures like hardware pre-fetchers. The access on the x vector are driven by the sparsity pattern of A, which makes them irregular and hard to predict. In addition, x is the only element of SpMV where some degree of data reuse can be exploited, although the irregular nature of its accesses prevent the reuse of the x vector. Another important performance aspect when parallelizing SpMV is the control flow divergence driven by the different number of non-zero entries of the A matrix. Such divergence can become a really important issue to leverage the computing power of high-end numerical accelerators like Graphic Processor Units (GPUs) or long vector architectures.

Many different approaches have been proposed to efficiently store sparse matrices and efficiently run SpMV $y = Ax$. One of the most common approaches, Compressed Sparse-Row (CSR), and its column counterpart Compressed Sparse-Column (CSC), efficiently stores sparse matrices and enables simple stride-1 memory access patterns on A and y. However, accesses on x are highly irregular and it suffers from flow divergence issues since each row-vector product depends on the number of non-zeros it contains. As such, it is not an appropriate format to manipulate sparse matrices on GPUs or long vector architectures.

Other approaches aim to mitigate the drawbacks of CSR by enlarging its storage requirements to increase the locality on x. SELL-C-σ [1] and ELLPACK Sparse Block [2] make use of row sorting and column blocking to improve both storage requirements and locality on x. This tech-report demonstrates that, although some of these approaches are very good abstractions to represent and manipulate sparse matrices, there are many unexploited opportunities to improve their performance on long vector architectures.

The main contributions of this tech-report are:

– We develop a highly efficient SpMV implementation for long vector architectures based on the state-of-the-art SELL-C-σ format. Our implementation reaches 117 GFlops saturating 69% of the peak memory bandwidth in a NEC SX-Aurora Vector Engine [3]. This accelerator leverages a vector instruction set that goes beyond the SIMD approach (of e.g., AVX-512 in x86 CPUs) with an 16-kbit long vector registers and instructions. Our evaluation assumes therefore an exploratory role for other vector ISA gaining momentum such as Arm SVE and RISC-V vector extension. Since there was no SELL-C-σ implementation for such class of architectures, we contribute to the HPC community with a highly optimized vector implementation of the SELL-C-σ format that can easily be ported to other vector ISA whenever they are ready.

– We implement, evaluate and discuss the performance impact of several optimizations targeting vector architectures making extensive use of the available vector instructions. We improve the SELL-C-σ baseline by 12%.

– We compare our novel approach for long vector architectures with other state-of-the-art approaches targeting SpMV in cutting-edge multi-core CPU and GPU devices. We demonstrate that our approach is 3$\times$ and 1.71$\times$ faster, respectively, than these approaches since it achieves an outstanding average SpMV efficiency of 4.19% of the peak performance.

The remaining part of the document is structured as follows: in Section 2 we introduce the background of SpMV, drawbacks, and benefits of several previously proposed formats that we use as a starting point for our work; in Section 3 we explain in detail the added contributions of this tech-report; in Section 4 we describe our experimental setup, i.e., the hardware and the system software used to run our tests; in Section 5 we present and analyze the results of our experiments; Section 6 contains an overview of other relevant related work; Lastly, in Section 7, based on our analysis, we offer conclusions about the suitability of SELL-*C*-σ and other SpMV optimizations for very long vector architectures.

2 Background

In this section, we introduce relevant state-of-the-art sparse matrix formats that set the grounds to build our long-vector architecture targeted optimizations for SpMV that conform the contributions of this tech-report.

CSR – The Compressed Sparse Row (CSR) is one of the most commonly used formats to represent sparse matrices. It stores the values and column indices of all the number of non-zero (*NNZ*) elements in two separate arrays in row order. A third array keeps a pointer to the starting position of every row in those arrays. In practice, the main advantages of CSR are that it has a very good compression ratio for any type of matrix element distribution and performs accesses to the A values and column indices in a streaming fashion. The main disadvantages are that it is not a format suitable for vector architectures, and that the accesses to x have poor locality which ends up producing a high number of outstanding requests to main memory.

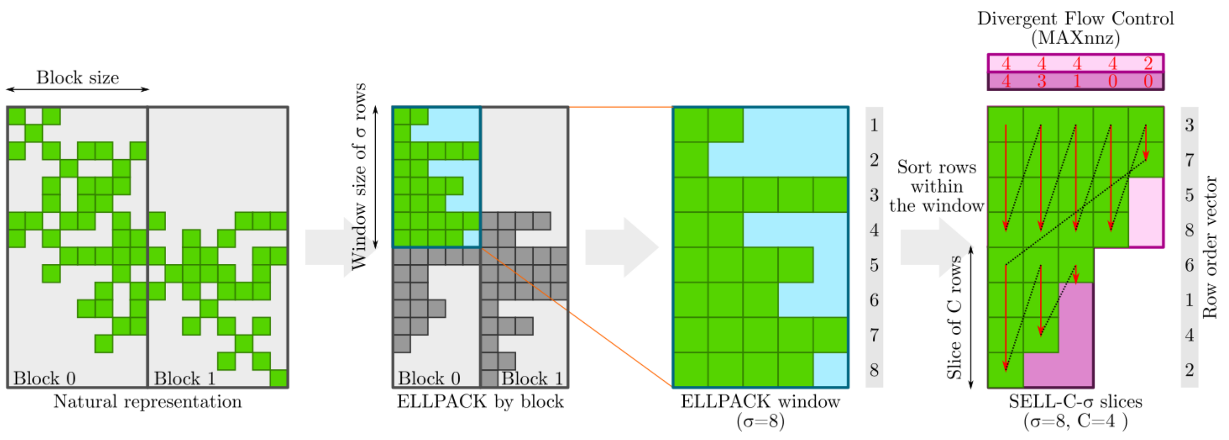


Figure 1: *Data layout representation of SELL-C-σ with* Column Blocking *and* Divergence Flow Control *optimizations.*

ELLPACK – ELLPACK [4] is designed to perform efficiently in GPUs and vector architectures. Compared to CSR, it offers improved locality of memory accesses to x by storing and accessing the non-zero elements in column order at the cost of additional storage. For a matrix A of size $M \times N$ with a maximum row size of K, it requires an array of size $M \times K$ to store A. The two left-most drawings in Figure 1 visually describe how a sparse matrix is compressed and represented in ELLPACK. The main downside of ELLPACK is that it only offers good performance and compression as long as the matrix being stored has a regular *NNZ* elements per row. As matrices become more irregular, the performance is expected to decrease and the number of zero elements stored to increase.

Sliced ELLPACK – SELLPACK [5] optimizes the ELLPACK format to improve storage efficiency and throughput for both regular and irregular matrices. In this format, matrix rows are ordered by their *NNZ*. Then, the matrix is logically divided into slices of a fixed number of rows. Rows are padded with zeroes to match the longest row within the same slice, as opposed to ELLPACK that matches the longes row in the whole matrix.

SELL-*C*-σ – SELL-*C*-σ [1] introduces the idea of limiting the sorting window to avoid reordering the whole matrix. SELL-*C*-σ addresses two main issues: i) the large cost of sorting matrices with a large number of rows; and ii) the exploitation of the natural locality in accesses to x of adjacent rows, which approaches that apply a total sorting of the matrix can reduce. The sorting window size is defined by the σ parameter, which is typically a multiple of the maximum length of the SIMD or vector unit.

ELLPACK Sparse Block (ESB) – ESB [2] introduces two additional optimizations to reduce bandwidth requirements: column blocking, and the use of a bit array to mask instructions. In an effort to improve the locality of accesses to x, ESB subdivides the matrices by blocks of columns. Each of those blocks is processed like an independent matrix stored with the SELL-*C*-σ format and contiguously allocated. ESB uses a bit array data structure to store one bit mask for each column of the slice. A bit is set to one for each non-zero element in the column. This information is later used to mask the elements of the SIMD operations. It also allows to compress the matrix even more as zero padding is no longer required.

Figure 1 represents the logical steps to convert a sparse matrix to the SELL-*C*-σ (with blocking) format during the preprocessing phase. In this simplified example we consider the parameters: $C = 4$, $\sigma = 8$ and *num. blocks* = 2. On the leftmost part, the natural representation of the matrix is divided in two separate blocks of *Block size* columns. Then, the rows of each block are ordered by size within a so-called ELLPACK window, i.e., within the blue region containing contiguous chunks of σ rows. The rightmost drawing (SELL-*C*-σ slices) shows the final data layout within a window where two slices are highlighted in light-pink and light-purple. Elements within each slice, including the zero-padding if no optimization is applied, are stored contiguously in *column-major* order. The red arrows describe the access pattern to the matrix elements if our Divergent Flow Control or the Bit-Array optimizations are enabled. We elaborate on this optimization further in this work in Section 3.2.

3 Contributions

In this section, we describe each of the contributions that improve the performance of SpMV. Section 3.1 presents our proposals to improve the SELL-*C*-σ format. These proposals can be applied to a wide range of scenarios. Section 3.2 describes our contributions to accelerate SpMV on long vector architectures.

3.1 Implementing the SELL-C-σ format

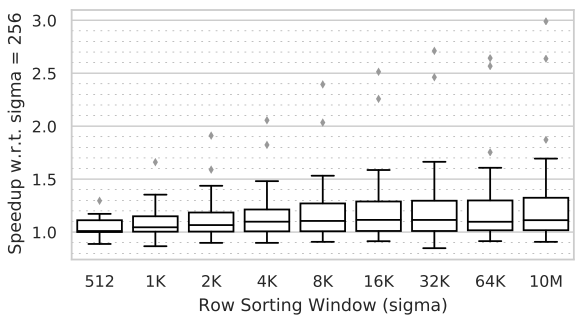
SELL-C-σ is an efficient sparse matrix format for vector architectures. ESB extends this format with optimizations that target a particular type of matrices or better suited for specific accelerators like the Intel Xeon Phi (KNC). Because of that, we choose SELL-C-σ as a starting point to develop our SpMV implementation for Vector Engine. From there, we revisit and adapt some optimizations previously proposed in the literature extending them with new approaches targeting long vector architectures. In detail, we explore: *i)* the adequate sorting strategy based on the trade-off between performance and preprocessing overhead as the σ parameter increases; *ii)* the use of task-based parallelism and the impact of the task granularity in the scaling performance of SELL-C-σ; and *iii)* the impact of column blocking in matrices to improve locality on vector x.

In Listing 1 we provide detailed pseudo-code of our basic implementation of SELL-C-σ for the Vector Engine. The rest of the optimizations are built on top of this code. Our codes are optimized using low-level intrinsics which allow us to leverage NEC specific architectural features such as arithmetics using long vectors and memory access policies.

```
void kernel_SELLCS(ELLPACKmtx matrix, double x, double y,
        int64 start_row, int64 end_row, int64 vrow_order,
        int64 slices_width, int64 slices_ptr) {
  int vlen = 256;
  // Outer loop: iterates over rows in the matrix
  for (int64 rowid = start_row; rowid < end_row; rowid += 256)
  {
    //nl is the number of active vector lanes, always 256 except a the end of the matrix
    int nl=((end_row - rowid) < vlen)? (end_row - rowid): 256;
    // Set results = {0,...0}
    vr results = _vxor(results, results, nl);
    // Pointers to values and col. indices
    int64 slice_idx = rowid >> 8;
    double *values_ptr = &matrix->values[slices_ptr[slice_idx]];
    int64 *colidx_ptr = &matrix->column_indices[slices_ptr[slice_idx]];
    // Compute scatter addresses
    vr y_sc_addr = _vld(8, &vrow_order[rowid], nl);
    y_sc_addr = _vmulul(8, y_sc_addr, nl);
    y_sc_addr = _vaddul(y, y_sc_addr, nl);
    // Get the width of the slice
    int64 swidth = slices_width[slice_idx];
    /* Inner loop: iterates over columns in the slice */
    for (int64 i = 0; i < swidth; i++)
    {
      // Vector load matrix values and col. indices
      vr A_values = _vld(8, values_ptr, nl);
      vr A_colidx = _vld(8, colidx_ptr, nl);
      // Gather X vector values
      vr xgather_addr = _vmulul(8, A_colidx, nl);
      xgather_addr = _vaddul(x, xgather_addr, nl);
      vr x_val = _vgt(xgather_addr, &x[0], &x[ncols], nl);
      // Multiply
      results = _vfmadd(results, x_val, A_values, nl);
      values_ptr += 256; colidx_ptr += 256;
    }
    // Scatter the results back to Y
    _vsc(results, y_sc_addr,&y[0], &y[nrows], nl);
  }
}
```

Listing 1: *Basic Vector Engine SELL-C-σ implementation*

Sorting strategies – The SELL-*C*-σ format requires matrix rows to be sorted in terms of their *Number of Non-Zero* elements, NNZ, which may incur a significant overhead. We mitigate the cost of SELL-*C*-σ in terms of sorting by applying the non-comparative Radix Sort algorithm. It has a worst-case cost of $\mathcal{O}(w \times n)$, where w is the number of digits of the largest value, and is well suited for sorting integers and long-vector architectures [6]. Since reordering the whole matrix can be extremely costly, SELL-*C*-σ splits the matrix into several subsets of rows and requires each one of them to be sorted independently. We call the number of rows per subset *reorder window*, which corresponds to the σ parameter. To select an optimal reorder window size for our experiments, we run our SELL-*C*-σ implementation with Divergent Flow Control optimization, as it requires an extra preprocessing step in which the $MAXnnz$ structure is computed (see Figure 1). We evaluate the trade-off between the performance benefits of SELL-*C*-σ during the SpMV computation, and the cost of sorting the matrix rows during the preprocessing phase. If the σ parameter is bigger than the total number of rows of a matrix, the second is used instead. We use the optimal task partitioning configuration for each matrix. As Figure 2 shows, for some matrices a bigger reorder window can provide up to 2× performance improvements, but for most of them, a reorder window of 16 k-rows is enough to perform optimally with improvements between 0% and 30% compared to σ = 256. In general, reordering the matrix in groups of 16K rows takes about 40% to 50% additional time during the preprocessing phase. For reference, in our experiments, converting most matrices to SELL-*C*-σ takes the equivalent in time of 70 to 150 iterations. To avoid algorithmic errors due to the sorting, we use the data structure, *row_order[num_rows]*, that keeps the initial order of the matrix, to store back the results in y.



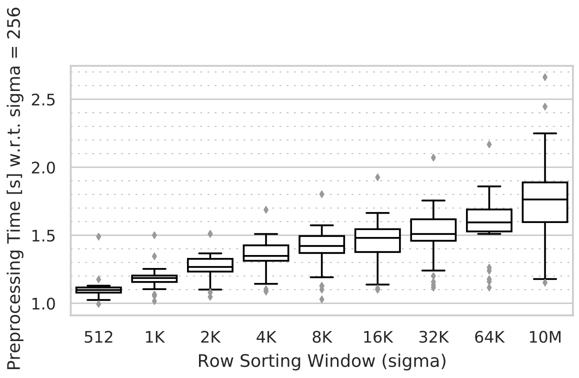


Figure 2: *Performance increase and preprocessing overhead increasing the sigma window.*

Task-Based Parallelism – We orchestrate the parallel execution of our workloads by splitting them into several sequential pieces, called tasks, and letting an underlying runtime system to dynamically schedule them as computing resources become available. OpenMP supports this parallel execution model via task constructs [7]. In this context, the programmer must assign an appropriate amount of work to each parallel task to expose a significant amount of concurrency to the parallel hardware while, at the same time, avoid incurring in too much overhead in terms of task creation or synchronization. As the ideal task granularity changes from matrix to matrix, we perform a scalability test up to 8 OpenMP threads, with different number of tasks ranging from 8, which would be the minimum for 8 cores, to 256 in steps in powers of two. In any task partition configuration all tasks contain similar NNZ. The results of these experiments allow us to understand the level of task parallelism needed to achieve the best workload balancing without creating too many tasks instances. In general, we observe that most matrices achieve its best performance when the workload is divided between 8 to 64 tasks. Only few experiments (around 10%) show better performance when the workload is divided between 128 and 256 tasks. In such cases the benefits are less than 1% compared to dividing it by 8 to 64 tasks. For each matrix, we select the number of tasks configuration that achieves the highest performance. We use such configurations in our performance experiments, further, in the evaluation section.

Matrix Column Blocking – As we mention in Section 2, previous work [2] analyzes the effects of column blocking on SpMV.

However, that study is limited to Intel Xeon Phi (KNC) which has a SIMD width of 512-bits. We extend our SELL-*C*-σ base implementation with a similar column blocking approach on top of SELL-*C*-σ to test its suitability with longer vectors. We comment on the results in Section 5.1.

3.2 Optimizations Targeting Long-Vector Architectures

In this section, we describe the optimizations we propose targeting long-vector architectures like SX-Aurora and others. Our proposals to accelerate SpMV on long vector architectures are: *i)* the use of cache allocation to improve the reuse of x and deprioritization of store dependencies; *ii)* divergence flow control adapting the length of vector operations to avoid loading and computing *zero-padded* elements; *iii)* enabling loop unrolling in SELL-*C*-σ using partial loop fusion; *iv)* efficient computation of gather and scatter addresses with special instructions.

Cache allocation and store relaxation policies – In the context of SpMV, data structures corresponding to the matrix A have very different data access patterns and reuse properties than the vector x. Indeed, the matrix *values* and *column_indices* data structures are accessed with a very simple stride-1 memory access pattern and never reused, while the vector x is accessed randomly but often reused. One of the fundamental aspects to achieve good performance for SpMV is to exploit all opportunities for data reuse to alleviate the pressure on the memory bandwidth. Therefore, it is critical to reuse all vector x coefficients stored in the cache hierarchy as much as possible. Long-vector architectures typically offer support to explicitly guide from the source code the cache replacement policy [3, 8]. We exploit such support to indicate to the architecture that cache lines containing pieces of both the *values* and *column_indices* arrays must be evicted before than any other cache line. Therefore, we reduce the chances of evicting a cache line occupied by the vector x an access to a non-reused data structure misses. We refer to this policy as *Non-Cacheable* load.

Another important aspect is the way we handle stores, which exploits the memory access pattern driven by the SELL-*C*-σ format. We use a register to accumulate the intermediate results of a *rows slice*. Once the pass over a certain slice finishes, the vector register holding intermediate results is stored to several memory addresses that will not be accessed again until many other slices are computed. Storing the whole vector register in memory is done via a scatter store instruction. Typically, dependencies on scatter store instructions are computed considering the whole range between the initial and final addresses they access, which can delay subsequent memory instructions that access addresses within this range, although not necessarily the same ones as the scatter store. Since we know that memory instructions immediately following the scatter do not access the same memory addresses, we instruct the hardware to not check dependencies across the scatter and some subsequent instructions. We refer to this policy as Store *Overtake*. A memory fence instruction is inserted to define where this relaxation period finishes. Modern vector architectures support this kind of memory scatter dependencies relaxation. [8].

Table 1 summarizes how we apply this concepts in our implementation of SELL-*C*-σ and include cache allocation and store relaxation policies in each memory access to obtain improvements in performance.

Loop	Vector Mem. Access	Policy applied
Inner	Load Values	Non-Cacheable
Inner	Load Col. Indices	Non-Cacheable
Inner	Gather X coef.	< none >
Outer	Load Row Order	< none >
Outer	Scatter result to Y	Overtake

Table 1: *Policies applied to optimize SELL-C-σ.*

Handling Flow Divergence by Adapting the Vector Length – One major aspect when computing SpMV on long vector architectures is the management of flow divergence. It mainly arises from the varying number of non-zero elements per matrix row, which forces the loop iterating over matrix non-zero coefficients to produce different control flow scenarios per row. Therefore, vectorizing matrix rows containing very different numbers of non-zero elements wastes computing resources, particularly the ones assigned to rows with small amounts of non-zero coefficients.

One very popular technique to handle flow divergence is the use of predicated registers. They contain an array of bits specifying whether its corresponding vector element is zero or not. Predication is supported by commercial SIMD ISAs like AVX512 [9] or SVE [10] and is a valuable approach to let the compiler vectorize irregular loops. However, it requires a large amount of storage, since every single vector element needs its corresponding predicated mask, and many of its implementations do not avoid processing zeroed vector entries but just discard the output of these meaningless computations.

We propose a new approach to handle flow divergence on vector architectures that relies on the ratio between the number of vector elements v_{el} and the number of vector lanes v_{la}. We call this new approach *Divergent Flow Control* (DFC). Vector architectures have the capacity of processing vector elements in batches of v_{la}, that is, they require processing $\frac{v_{el}}{v_{la}}$ batches of v_{la} to finish the pass over the whole set of v_{el} elements. Our approach instructs the hardware to process just $MAXnnz$ elements of the vector, which implies that the hardware will need to process just $\left\lceil\frac{MAXnnz}{v_{la}}\right\rceil$ batches of v_{la} elements, where $MAXnnz$ is the maximum number of non-zeros over all rows involved in the vector operation. Importantly, this information can be defined in just 8 bits. For example, the SX-Aurora architecture, which is described in Section 4, has 256-element vector registers, hence the possible values are $1 \leq MAXnnz \leq 256$, which means that in a single byte we can encode the vector length that we need for each vector instruction.

As we describe in the previous section, we modify our code to enable this optimization by adding an extra data structure, the `active_lanes` vector, which contains the $MAXnnz$ for every *column-wise* vector operation to do inside each slice. The right-most drawing of Figure 1 shows a basic example of how $MAXnnz$ are counted and stored in a vector. The computation of zero-padded elements in yellow is avoided.

Applying loop unrolling – Loop unrolling is a well-know optimization to reduce control flow overhead and maximize the use of the register file. Our loop unrolling approach particularly maximizes locality on vector x by increasing the depth of vertical or *column-wise* vector operations while, at the same time, maximizes data reuse on the register file. This is a very natural optimization in our context since SELL-*C*-σ also targets the same kind of locality on vector x without writing back to memory partial results until all the rows in a slice are completed.

An efficient implementation of loop unrolling is not straight forward in our context. In Listing 2, we show an example including unrolling two slices of the algorithm. This example also implements the *DFC* optimization that we describe above. The main issue when unrolling slices is that each slice may have a different width. However, we know that within the same row order window, each slice has a width less or equal to the previous one. That is, in on our example $swidth1 \geq swidth2$.

It is possible to fuse from the *bottom-up* the inner loop iteration space of the slices as we show in lines 25 to 41 in Listing 2. We must add an additional loop to handle the remaining elements of slice 1. This loop is represented in lines 43-47 in our example. Note that, if unrolling is applied, the sigma reorder window has to be a multiple of the number of rows the unroll covers.

The unroll size is limited by the number of vector registers. Since the number of local variables increase each time we unroll, it is important to declare them in the most immediate context where they are consumed, which makes it easier for the compiler to manage register dependencies, apply architecture-specific optimizations and avoid spilling. Our implementation is able to unroll up to 8 times without producing any spilling access on a vector architecture with 64 architectural registers available.

Efficient computation of gather/scatter addresses – Gather and scatter instructions fetch and store, respectively, to the addresses provided in a vector. In our implementation, gather is used to access non-contiguous elements of the x vector, while scatter is used to store the results back to the corresponding element of the y vector. The addresses are generated by multiplying the index of the vector element we access by the corresponding data type size. (e.g., by 8 if we use 64-bit elements). To handle the integer arithmetic involved in address computation, a multiply instruction followed by and add operation can be replaced by a single shift and add instruction. Listing 2, (lines 20 and 30) exemplifies how we apply this optimization.

To evaluate the impact of our proposals, we created five implementations which incrementally include the optimizations we describe above. The optimizations included in every implementation are specified in Table 2. In addition to those five, we also created an implementation of SELL-*C*-σ including the column blocking, sorting strategies, DFC and efficient computation of gather/scatter addresses, that we evaluate in a

```
kernel_SELLCS(...) {
  int vlen = 256;
  for (int64 rowid = start_row; rowid < (end_row - 511); rowid += 512) {
    int64 slice_idx = rowid >> 8;
    double *values_ptr = &matrix->values[slices_ptr[slice_idx]];
    uint64 *colidx_ptr = &matrix->column_idx[slices_ptr[slice_idx]];
    // Duplicate variables
    vr results = _vxor(results, results, vlen);
    vr results2 = _vxor(results2, results2, vlen);
    int64 swidth = slices_width[slice_idx];
    int64 swidth2 = slices_width[slice_idx + 1];
    int64 act_lanes_idx = actlanes_ptr[slice_idx];
    int64 act_lanes_idx2 = act_lanes_idx + swidth;
    double *values_ptr2 = values_ptr + (swidth*vlen);
    int64 *colidx_ptr2 = colidx_ptr + (swidth*vlen);
    vr y_sc_addr = _vld(8, &vrow_order[rowid], vlen);
    y_sc_addr = _vsfa(y_sc_addr, 3UL, y, vlen);
    vr y_sc_addr2 = _vld(8, &vrow_order[rowid + vlen], vlen);
    y_sc_addr2 = _vsfa(y_sc_addr2, 3UL, y, vlen);
    /* Partial loop fusion: Slices 1 & 2 */
    for (int64 i = 0; i < swidth2; i++) {
      // Slice 1
      int nl_1 = vactive_lanes[act_lanes_idx++] + 1;
      vr A_values = _vld(8, values_ptr, nl_1);
      vr A_colidx = _vld(8, colidx_ptr, nl_1);
      vr xgather_addr = _vsfa(A_colidx, 3UL, x, nl_1);
      vr x_val = _vgt(xgather_addr, &x[0], &x[ncols], nl_1);
      results = _vfmadd(results, x_val, A_values, nl_1);
      // Slice 2
      int nl_2 = vactive_lanes[act_lanes_idx2++] + 1;
      vr A_values2 = _vld(8, values_ptr2, nl_2);
      vr A_colidx2 = _vld(8, colidx_ptr2, nl_2);
      vr xgather_addr2 = _vsfa(A_colidx2, 3UL, x, nl_2);
      vr x_val2 = _vgt(xgather_addr2, &x[0], &x[ncols], nl_2);
      results2 = _vfmadd(results2, x_val2, A_values2, nl_2);
      values_ptr += vlen; colidx_ptr += vlen; values_ptr2 += vlen; colidx_ptr2 += vlen;
    }
    /* Finish the vector ops remaining in slice 1*/
    for (int64 i = swidth2; i < swidth; i++) {
      // [ ... ]
      results = _vfmadd(results, x_val, A_values, nl_1);
      values_ptr += vlen; colidx_ptr += vlen;
    }
    _vsc(results, y_sc_addr, &y[0], &y[nrows], vlen);
    _vsc(results2, y_sc_addr2, &y[0], &y[nrows], vlen);
  }
  // [Omitted Unroll Epilogue]
}
```

Listing 2: *Manually unrolling the SELL-C-σ by 2 slices*

dedicated subsection of Section 5.1.

3.3 Optimization generalization

We examine the portability of these optimizations to three additional mainstream vector (or SIMD) architectures: Intel AVX-512, ARM SVE and RISC-V vector extension. Following, we describe the necessary changes in the code to port each optimization, and comment about the possible limitations or drawbacks in performance.

Some optimizations like the sorting strategies, column blocking and loop unrolling, are implemented with standard C and not using any architecture dependent instruction. Of course, the optimal σ, unroll and number of blocks parameter configurations are expected to differ across input sets and architectures. In a similar way, implementing the task-based parallelism approach just requires a parallel runtime with OpenMP

Optimization	SELLCS	SELLCS DFC	SELLCS U8-DFC	SELLCS U8-NC	SELLCS U8-NC-DFC
Sorting strategies	●	●	●	●	●
Task-Based Parallelism	●	●	●	●	●
Matrix Column Blocking					
Cache Allocation & Store relaxation policies				●	●
Divergent Flow Control		●	●		●
Loop unrolling			●	●	●
Efficient gather/scatter address computation	●	●	●	●	●

Table 2: *Optimizations applied on each of the implementations evaluated in the tech-report.*

4.0 support available in the system. Our Divergent Flow Control optimization leverages the *load vector length* from the Vector Engine ISA to adjust the vector length on every operation as needed; in RISC-V, the length of each vector operation can also be dynamically set with a similar instruction. Intel AVX nor ARM SVE specifications do not support an exact equivalent mechanism, but they support masking of operations. A partial version of this optimization then, can be implemented using masks using an additional *Bit-Array* data structure [2]. However, the use of masks has two disadvantages compared to adjusting the vector length. The major one is that, although masking will prevent unnecessary vector loads of zero elements, it will not reduce the latency of the vector instructions. Also but less relevant, a bit array requires 256 bits to control the 256 elements of the vector while *DFC* would use just 8 bits.

In other architectures it is also possible to annotate instructions with cache allocation and store relaxation hints, referred also as non-temporal hints, like in SX-Aurora. In Intel architectures this behavior is enabled by using the *movnt** instruction for stores and loads. In this case, the *mfence* instruction can be used for synchronization. ARM SVE supports also non-temporal loads (e.g., load1d becomes loadnt1d). The current specification of RISC-V vector extension does not support non-temporal hints by default so it remains implementation dependant.

Finally, shift and add is an interesting instruction to speedup gather and scatter logical address computation in aurora. However, AVX512, ARM SVE and RISC-V instructions use indices or offsets relative to a base address specified in an instruction field. The logical address computation is done internally.

4 Methodology

In this section we introduce the hardware and software infrastructure used for our evaluation. The main system on which we evaluated our implementation is the **NEC SX-Aurora** (described in the following section), while for performance and energy comparison we considered:

- **Intel Xeon Platinum 8160 CPU** with 24 cores each running at 2.10 GHz. Each core houses 32 kB L1 and 1024 kB L2 data cache. The L3 cache is shared among the 24 cores and its size is 33792 kB. Also, each core is powered by a AVX-512 SIMD unit, allowing operating with registers of up to 512 bits (i.e., 16 floats, 8 doubles).

– **NVIDIA V100 (Volta) GP-GPU** with 84 Volta Streaming Multiprocessor (SM) running at a maximum frequency of 1.5 GHz. The 84 SMs share 6144 kB L2 cache and are connected to 4096 GB HBM2.

4.1 SX-Aurora Vector Engine

The NEC SX-Aurora Vector Engine (VE) is the latest incarnation of NEC's long vector architecture which combines SIMD and pipelining. Vector units and vector registers use a 32×64-bit wide SIMD front in a 8 cycles deep pipeline resulting in a maximum vector length of 256×64-bit elements or 512×32-bit elements. The VE10B processor used for this publication has been released in 2018, its characteristics were presented at the IEEE HotChips 2018 [11] and first evaluations of performance were discussed in [3]. Due to its 6 HBM2 8 high stacks the Vector Engine has a very high memory bandwidth of 1.22 TB/s out of the 48 GB on-chip RAM, shared by only 8 powerful cores. The VE10AE and VE10BE models released at the end of 2019 have improved the memory bandwidth to 1.35 TB/s but were not accessible for this publication's work.

Sparse matrix operations performance benefits of large memory bandwidth, but equally important are other characteristics of the processor like mechanisms for memory latency hiding or caches. Each of the 8 Vector Engine cores consists of a scalar processing unit (SPU) and a vector processing unit (VPU) and is connected to a common last level cache (LLC) of 16 MB. The core's bandwidth to the LLC is 406.9 GB/s, bidirectional, therefore the memory bandwidth (995 GB/s peak measured with STREAM) can be saturated by 4 cores. Each VPU has 64 architectural vector registers of 256×64-bit elements and the threefold amount implemented in hardware, used for register renaming. Three fused multiply-add vector units deliver a peak performance of 269 GLFOPS (double precision) per core at 1.4 GHz, and 307 GFLOPS for the VE10A model running at 1.6 GHz. The peak performance of the used Vector Engine variant is 2.15 TFLOPS, which is not impressive when compared to the latest GPGPUs, but the 0.56 byte/FLOP represent a well-balanced CPU with coarse grain parallelism in 8 cores and fine-grain parallelism at vector level.

Vector Engines are integrated as PCIe cards into their host machines and offload the operating system functionality entirely to the host. They run in multitasking and multiprocessing mode, like standard CPUs, programs can run entirely on the Vector Engine (native programming model), offload parts of code to the host machine (reverse offloading) or run on the host and offload compute kernels to the Vector Engine (accelerator, offload model). Heterogeneous programs can also be built with the hybrid MPI provided by NEC that connects processes running on the host and the VEs. In all cases programmers can use languages like C, C++, Fortran, and parallelize with MPI as well as OpenMP, while accelerator code can still use almost any Linux system call transparently.

The proprietary compilers from NEC support automatic vectorization aided by directives. They are capable of using most features of the extensive vector engine ISA [12] from high-level languages loop constructs. For the work presented in this article, we needed even tighter control over Vector Engine features like vector masks generation and control, vector registers and LLC cache affinity of data. Therefore, we turned to the open-source LLVM-VE project [13] which supports intrinsics allowing full control over the generated code [14].

4.2 Experimental Setup

For our study, we select matrices, used frequently in recent literature [1,15–17] that represent a wide range of problems in the area of HPC applications. In Table 3 we include the list of matrices and some of its characteristics, all of them can be downloaded at the SuiteSparse Matrix Collection repository[1].

Name	Row×Col	NNZ	NNZ/row	Density
scircuit	170K × 170K	958K	5	2.92E-05
mc2depi	525K × 525K	2.1M	3	5.71E-06
webbase-1M	1000K × 1000K	3.1M	3	3.00E-06
s4dkt3m2	90K × 90K	3.7M	41	4.53E-04
dense2	2K × 2K	4.0M	2000	1.00E+00
bmw7st-1	141K × 141K	7.3M	51	3.61E-04
torso1	116K × 116K	8.5M	73	6.28E-04
mip1	66K × 66K	10.3M	155	2.33E-03
fcondp2	201K × 201K	11.2M	55	2.73E-04
pwtk	217K × 217K	11.5M	52	2.39E-04
fullb	199K × 199K	11.7M	58	2.91E-04
halfb	224K × 224K	12.3M	55	2.45E-04
BenElechi1	245K × 245K	13.1M	53	2.16E-04
crankseg-2	63K × 63K	14.1M	221	3.46E-03
Si41Ge41H72	185K × 185K	15.0M	80	4.31E-04
TSOPF-RS-b2383	38K × 38K	16.1M	424	1.11E-02
msdoor	415K × 415K	19.1M	46	1.11E-04
bundle-adj	513K × 513K	20.2M	39	7.60E-05
ML-Laplace	377K × 377K	27.5M	73	1.94E-04
ldoor	952K × 952K	42.4M	44	4.62E-05
bone010	986K × 986K	47.8M	48	4.86E-05
af-shell10	1.5M × 1.5M	52.2M	34	2.25E-05
ML-Geer	1.5M × 1.5M	110M	73	4.85E-05
nlpkkt240	27M × 27M	760M	27	9.65E-07
12month1	12K × 837K	22.6M	1814	2.25E-03
spal-004	10K × 322K	46.1M	4524	1.46E-02

Table 3: *Sparse matrices used in the evaluation.*

For our SELL-C-σ implementations in the Vector Engine, we compile with LLVM-VE v1.8 and link NEC's proprietary compiler NCC v3.0.1. We also use the following math libraries for performance comparisons between platforms: NLC 2.0, cuSPARSE v10.2, and MKL v2020.0 for the Vector Engine, NVIDIA V100 and Intel Xeon systems described above. All of them released less than a year ago. All codes are compiled with the *-O3* optimization level. For easy reproducibility, all the SpMV implementations, the benchmarking tool developed and the exact environment configuration files used for each system are provided in our repository[2]

[1]https://sparse.tamu.edu/

[2]https://repo.hca.bsc.es/gitlab/cgomez/spmv-long-vector

5 Evaluation

5.1 Performance of Long Vector Optimizations

In this section, we evaluate the performance impact of the different optimizations we describe in Section 3. They are implemented on top of our SELL-C-σ base implementation for the Vector Engine. As several studies show [18], SpMV performance is highly dependent on the sparse matrix structure. Therefore, our optimizations are not expected to deliver the exact same performance for all the matrices evaluated. To analyze in detail our performance results, we take into account several aspects like: matrix size and density, information we provide in Table 3; and microarchitecture event information provided by the hardware Performance Monitoring Counters (PMCs). The PMCs available in the Vector Engine, report several relevant metrics such as: the total number of dynamically executed scalar and vector instructions, the average vector instruction length, the elapsed cycles, or the hit/miss ratio of the different cache hierarchy levels. We access the PMCs right before and after the SpMV computation. We analyze these metrics and we relate their values with the performance that the different optimizations achieve.

Figure 3 displays performance results in terms of GFLOP/s considering all the matrices described in Section 4.2 except the last two. We evaluate six different implementations of the SpMV kernel: *NLC, SELLCS, SELLCS-DFC, SELLCS-U8-DFC, SELLCS-U8-NC* and *SELLCS-U8-NC-DFC*. The *NLC* category represents results obtained with the NEC math library, a proprietary software developed by NEC and particularly tailored to the SX-Aurora Vector Engine. Table 2 specifies which of the optimizations described in Section 3 are included in *SELLCS, SELLCS-DFC, SELLCS-U8-DFC, SELLCS-U8-NC* and *SELLCS-U8-NC-DFC*. Results in Figure 3, are executed in SX-Aurora running in a single Vector Engine, with optimal σ and task partitioning configurations as described in section 3. The performance measures exclusively the SpMV computation without any pre- or post-process.

There are some missing values for the dense2 and nlptkk240 matrices in Figure 3. In the case of dense2, implementations containing the 8-slice unroll optimization, require at least 2048 rows to execute correctly and this matrix has only 2000. In the case of nlptkk240, the implementation using NLC runs out of memory while trying to allocate the data structures. Despite these issues, we include these matrices since they extend the variety of matrices considered in this part of the evaluation.

Cases with significant performance issues – In Figure 3 we observe a particular set of matrices where the performance of our SELL-C-σ implementation is clearly sub-optimal compared to the NLC math library. Those matrices are: mip1 and torso1. In such matrices, we have been able to identify clear issues when scaling the number of cores. To do so, we compare the total number of instructions executed between 1 and 8 cores. Ideally, if we run a parallel workload using 1 or 8 cores, the sum of total instructions executed in each core should be roughly the same for every run. For the case of mip1, when running on 8 cores, the total number of scalar instructions is one order of magnitude larger than the single-core run, while the number of vector instructions executed is roughly the same. This large increment of scalar instructions is introduced by the OpenMP runtime due to workload imbalance. Matrices with a large

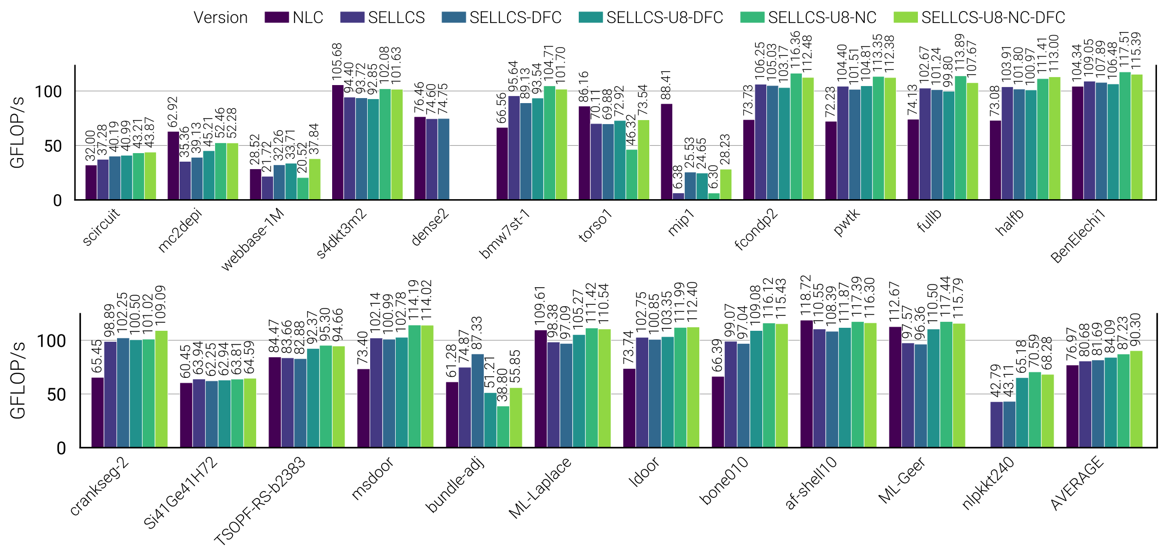


Figure 3: *Performance comparison of NLC vs our SELL-C-σ implementations for regular matrices.*

difference in NNZ element density among sorting regions may incur in these issues. A more refined data partitioning should be considered for such cases.

Divergent Flow Control evaluation – To understand the impact in performance of the *DFC* optimization, we compare the performance of *SELLCS* with *SELLCS-DFC*. These two implementations only differ in the use of the *DFC* optimization. Only the second one includes it. On average, the overall performance benefits of adapting each vector length instruction to $MAXnnz$ are almost negligible. However, it has a large impact in some scenarios. For example, for webbase-1M, which represents a website connectivity matrix and has a very low non-zero element density, *SELLCS-DFC* is 50% faster than *SELLCS*. For that matrix, PMC data reveals that the average length of vector instructions when using *SELLCS* is 256, while it drops to 148 for *SELLCS-DFC*. The performance difference between *SELLCS-DFC* and *SELLCS* is explained by the vector length reduction achieved by *SELLCS-DFC*, driven by the ratio of *NNZ* to zero-padding in webbase-1M. We observe a very similar behavior for the rest of the matrices that benefit from this optimization, including bundle-adj. While *DFC* does not offer any benefit in regular matrices where an average vector length of 256 does not waste resources, it can be critical for performance in highly irregular matrices. As the minimum required length of vector instructions can be precomputed before running SpMV, *DFC* potential benefits can be easily predicted.

Unrolling by 8 evaluation – To evaluate the benefits of unrolling we compare the performance of *SELLCS-DFC* with *SELLCS-U8-DFC*, since they are equivalent with the only exception that *SELLCS-U8-DFC* implements the unroll optimization. *SELLCS-U8-DFC* unrolls its most inner loop 8 times to process eight slices in the same iteration. *SELLCS-U8-DFC* is in average $\sim 3\%$ faster than *SELLCS-DFC*. For 13 out of 24 matrices, unrolling yields benefits ranging from 1% to 15%, while in the particular case of nlpkkt240 it brings a 51% performance increase. Now, unrolling introduces a constraint during the partitioning of the slices into tasks. The way we implement it, requires that the first slice of the 8 unrolled slices is the *longest* of them. After that, every next slice must be equal or shorter than the previous. To avoid breaking this constraint, we do not allow slices from two different sorting windows to concur in the same *unrolled iteration* and group them in multiples of 8. However, this forces our partitioner to create coarser than optimal tasks, which might end up causing load imbalance between cores. The drop in performance we observe in bundle-adj on the unrolled implementations is consequence of that: the first slices of the matrix contain a very high amount of non-zeros compared to the rest of it; to satisfy our partitioning constraint, those first slices are included in a single suboptimal large task. Figure 3 shows that, in general, matrices with fewer rows barely benefit from the unrolling optimization while the largest ones can obtain noticeable benefits. PMC data indicates that these performance benefits come from a reduction in the time spent computing vector operations. For the ML-Geer and ML-Laplace matrices, unrolling optimization yields 3% and 3.5% improvements in LLC hit ratio, respectively. It also brings 14% and 7% improvement in vector load throughput for ML-Geer and ML-Laplace, respectively. These benefits in load throughput are not only due to improved locality of the accesses on x, but also in the vector ILP, since unrolling exposes more instructions to the hardware.

Cache allocation and store relaxation policies – We evaluate the performance impact of using techniques that control cache eviction policies when data is loaded, and the relaxation of instruction dependencies when data are stored. We compare technique *SELLCS-U8-NC-DFC* with *SELLCS-U8-DFC*. Figure 3 reports that two thirds of the matrices obtain improvements ranging between 5% to 12% when using *SELLCS-U8-NC-DFC* compared to *SELLCS-U8-DFC*. We examine in detail two cases that represent the behavior observed in matrices that benefit from this optimization. When Cache allocation and store relaxation policies are enabled, ldoor and pwtk obtain an increase in LLC hit ratio of 8% and 6%, and an increase of the vector loaded elements per cycle of 10% and 8%, respectively. We also observe a reduction of $\sim 50\%$ of the L1 cache misses in both cases. These results show how preventing the eviction of the x vector is beneficial for performance. We do not observe any major drawback in performance for any of the matrices we tested, so cache allocation and store relaxation policies can be applied in any scenario without having to add complex logic to enable or disable it. We do not observe any correctness issue when using the store relaxation policies.

Applying all optimizations – We obtain in average 90.3 GFLOPs across all matrices by enabling all optimizations, which constitutes a significant improvement of ~12% and ~17% compared to the baseline SELL-C-σ and NEC math library implementations, respectively. The significant performance increase that we obtain over the NEC proprietary software, which is specially tailored to SX-Aurora Vector Engine, demonstrates the relevance of our optimizations in long vector architectures.

Applicability of column blocking – We revisit the matrix column blocking optimization (see Figure 1) to evaluate the impact in long vector architectures. We implement another separate version of our *SELLCS* base implementation including the sorting, task-based parallelism, cache and store policies, *DFC* and matrix column blocking optimizations. We perform two groups of experiments. First we evaluate the matrices used previously in the previous experiments (see Figure 3). For such matrices we do not observe performance benefits when increasing the number of blocks in which the columns are divided. However, previous studies [2] with standard SIMD architectures, show that this optimization is only effective in *skewed* matrices with very long rows, which have higher opportunities to exploit x vector locality. For purposes of comparison, we test a second group composed of two additional matrices with *skewed* shape: *spal_004* and *12month1*. We consider optimal σ task granularity for these two matrices, as in our previous experiments. If Figure 4, *12month1* achieves up to $2\times$ performance if the matrix is divided in 16 blocks instead of one. In the case of *spal_004*, we measure a steady degradation in performance as the number of blocks increases. Our results suggest that, while the applicability of this optimization seems to be very limited, it still can have a huge performance impact in few particular cases running on long vector architectures.

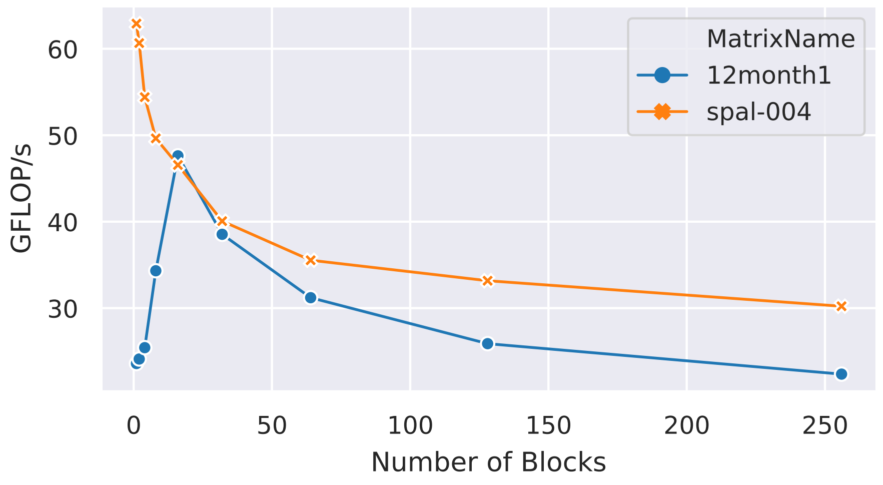


Figure 4: *Column Blocking:* 12month1 *obtains up to* $2\times$ *performance improvement.*

5.2 Comparison with state-of-the-art HPC architectures

We compare the performance and the energy efficiency of our *SELLCS-U8-NC* implementation in Vector Engine against the Intel an NVIDIA platforms we describe in 4 using their respective *MKL* and *cuSPARSE* math libraries. Both math libraries provide two methods to partition and schedule workload among cores. To ensure fairness, we run experiments performance results with both methods and keep the best result for each matrix. We report energy to solution metrics measuring the whole execution of the SpMV including matrix loading and preprocessing. The Intel platform reports *total energy* consumed by a run based on information provided by the *RAPL* monitoring counters. The nvidia-smi tool, provides information about the *power* drained by the GPU every second. In Vector Engine, we collect power information using a command-line program that reports instantaneous *current and voltage*. Power samples are then integrated over time to obtain the total energy consumption of each run. In all power measurement experiments, we run 600K iterations of the SpMV algorithm so the time spent loading the matrix from disk and pre-processing is less than 10% of the full duration. *MKL* experiments run with OMP_NUM_THREADS=48 to utilize all the cores in both sockets. *cuSPARSE* experiments using a single NVIDIA Volta V100 device.

The upper plot in Figure 5 shows our performance evaluation considering the three platforms. The lower bars represent the normalized energy-to-solution with respect to *MKL* results. On average, *SELLCS-U8-NC* achieves a 71% and 200% speedup over the *cuSPARSE* and *MKL*, respectively. In terms of energy efficiency, *SELLCS-U8-NC* consumes 22% less energy compared to *cuSPARSE*, and $9.09\times$ less energy compared to *MKL*. Also, a careful reader can detect that the performance figures of the Vector Engine presented in the upper part of Figure 5 are slightly lower than the ones presented in Figure 3. The mismatch is on average below 5% and it is due to the instrumentation overhead introduced by the power monitoring on the Vector Engine. We excluded two matrices from this comparison: dense2, due to incompatibilities with the unroll optimization; and nlpkkt240, due to memory management errors in the instrumentation libraries.

	x86 Skylake		NVIDIA v100		NEC SX-Aurora	
	GFlops	% of peak	GFlops	% of peak	GFlops	% of peak
Peak DP	3200		7800		2150	
Best SpMV	81.33	2.54%	86.15	1.11%	115.69	5.38%
Worst SpMV	3.05	0.09%	7.61	0.09%	28.07	1.3%
Average SpMV	29.84	0.93%	52.61	0.67%	79.32	4.19%

Table 4: *Percentage of the DP peak performance reached by SpMV on state-of-the-art HPC architectures.*

In Table 4 we report the percentage of the double-precision peak performance achieved by each of the three architectures.

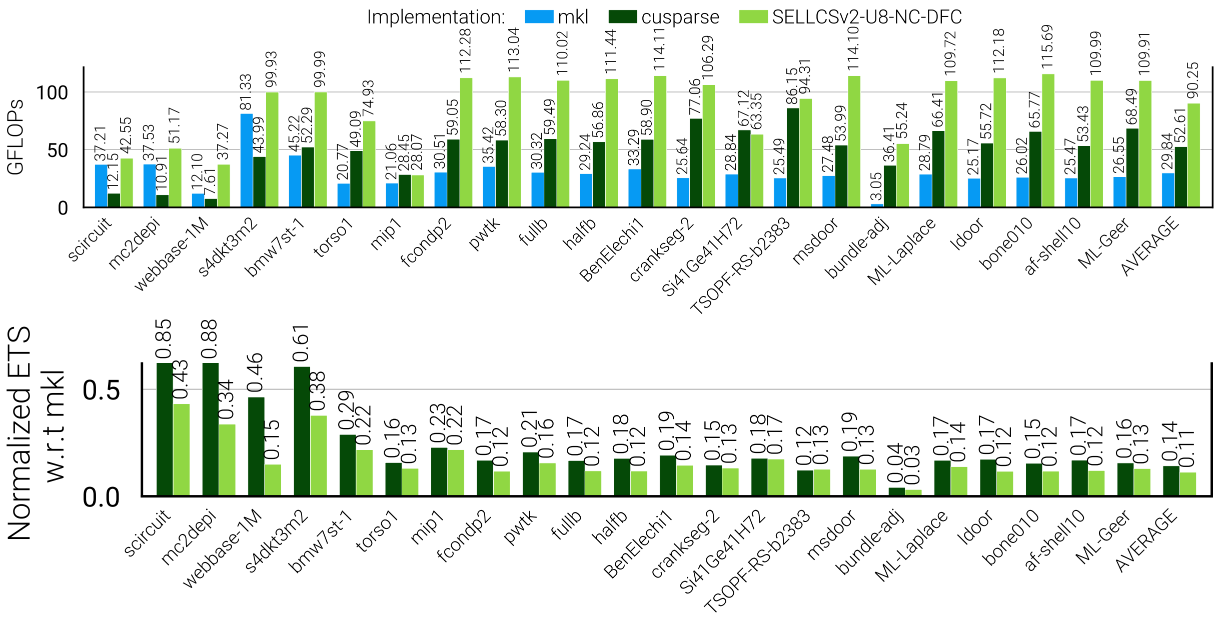


Figure 5: *Performance and energy-to-solution comparison between different computing platforms.*

6 Related work

Since SpMV is a kernel of paramount importance in several algorithms of scientific computing, a large body of research about SpMV optimization on many architectures has been published in the last twenty years. Most recently, the research community is focusing on developing efficient SpMV implementations targeting emerging architectures with different degrees of parallelism, e.g., high number of cores and long SIMD or vector units. However, most of those studies are limited to 8 elements SIMD units in CPUs or 32 elements warps in GPUs [1,2,15,16,18,19]. Even if this approach covers state-of-the-art Intel CPU with the AVX-512 extension and the latest GP-GPUs it falls short with a 256 double-precision elements vector length platform such as the SX-Aurora analyzed in our study.

Several studies propose new formats capable of exploiting the SIMD/vector units available in those new architectures. Some of them are based in CSR and use some kind of blocking with padding of zeroes to create contiguous segments of elements [15, 16, 20], however, as those formats rely on having big enough groups of NNZ elements close to each other to be efficient they are not suitable for long vector architectures. Both Beamer et al. in [21] and Buono et al. in [17] propose a very similar two-step SpMV algorithm that shows good performance improvements for big-data matrices. Using the code provided by the authors of [21] we were able to reproduce the results in an x86 system similar to the one used in their study. However, we were not able to develop an efficient implementation using intrinsics close in performance to our SELL-C-σ code in the Vector Engine.

The SELL-C-σ [1] and ESB [2] matrix formats build on top of the Sliced ELLPACK format [5]. They introduce several optimizations like limiting the reorder window of rows, division in blocks along the X vector to increase locality and bit-masks to avoid unnecessary memory accesses. Our work extends those studies by revisiting and optimizing such techniques in a long vector architecture like the SX-Aurora Vector Engine. Furthermore, we propose and evaluate two extra optimizations that produce an additional increase in performance.

Not all matrices perform optimally with a single format. A recent study shows how machine learning can be used to select automatically the most adequate format for each matrix [18]. While similar auto-tuning techniques can be combined with our work, e.g., by training a neural network to pick the adequate σ and task size parameters for us, that alone is a challenge on itself and is out of the scope of our study. Still, our results show that the chosen parameters allow a general performance improvement compared to the NEC optimized library over a wide range of sparse layouts.

7 Conclusions

Our implementation of SpMV for the SX-Aurora long vector architecture shows very competitive performance results which mostly overtake the highly optimized proprietary vendor implementation found in the NEC Library Collection. We used the open-source compiler LLVM for the kernel of the computation and discussed various optimizations that can be applied to other algorithms as well. The implementation is for a Vector Engine native library, callable from Vector Engine programs but can be easily transformed into an offload library callable directly from programs running on the host machine, in order to accelerate host-native programs. That use case would actually bring tuning opportunities for the preprocessing step which has received less focus with respect to optimization. With the sorting running on the host side we would expect a reduction of the matrix setup times.

When compared to other vector architectures the performance results in figure 5 show that the SX-Aurora performance is very competitive, in average winning over both standard libraries from the competing architectures Xeon Skylake Platinum and NVIDIA Volta: MKL and cuSparse. Although the Vector Engine has the lowest peak performance, 2.15 TFLOPS compared to 3.2 TFLOPS of two Skylakes and 7.8 TFLOPS of a V100, it can leverage its superior memory bandwidth and reach higher SpMV performance than its competitors. The results in table 4 show that the SX-Aurora has the most balanced architecture, its 0.567 byte/FLOP paired with the long vector ISA that exposes parallelism in a very explicit way lead to the highest percentage of performance efficiency of up to 5.38% of the peak.

In terms of energy-to-solution, the accelerators clearly beat the general-purpose CPU with SIMD extensions and highly optimized libraries by factors mostly larger than 9. Combining these with the absolute performance values the SX-Aurora appears to be a very attractive platform for memory bandwidth demanding data-parallel workloads.

References

[1] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "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*, vol. 36, no. 5, pp. C401–C423, Jan. 2014.

[2] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. Eugene, Oregon, USA: Association for Computing Machinery, Jun. 2013, pp. 273–282.

[3] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2018, pp. 685–696.

[4] D. R. Kincaid, T. C. Oppe, and D. M. Young, "Itpackv 2d user's guide," 5 1989.

[5] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 2010, pp. 111–125.

[6] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 712–721.

[7] O. A. R. Board, "Openmp 5.0 specification," Tech. Rep., November 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[8] N. CORPORATION, "X-aurora tsubasa architecture guide revision 1.1," Tech. Rep., 2018.

[9] C. Lomont, “Introduction to intel advanced vector extensions. intel white paper,” 2011.

[10] 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

[11] Y. Yamada and S. Momose, “Vector engine processor of nec’s brand-new supercomputer sx-aurora tsubasa,” in *Proceedings of A Symposium on High Performance Chips, Hot Chips*, vol. 30, 2018, pp. 19–21.

[12] “Sx-aurora tsubasa architecture guide,” https://www.hpc.nec/documents/guide/pdfs/Aurora_ISA_guide.pdf, 2018.

[13] “Llvm-ve github repository,” https://github.com/sx-aurora-dev/llvm-project - last accesses April 2020.

[14] “Llvm ve intrinsics,” https://sx-aurora-dev.github.io/velintrin.html - last accesses April 2020.

[15] X. Chen, P. Xie, L. Chi, J. Liu, and C. Gong, “An efficient simd compression format for sparse matrix-vector multiplication,” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, p. e4800, 2018, e4800 CPE-18-0532.R1. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4800

[16] Y. Li, P. Xie, X. Chen, J. Liu, B. Yang, S. Li, C. Gong, X. Gan, and H. Xu, “VBSF: a new storage format for SIMD sparse matrix–vector multiplication on modern processors,” *The Journal of Supercomputing*, Apr. 2019.

[17] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, “Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics,” in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS ’16. Istanbul, Turkey: Association for Computing Machinery, Jun. 2016, pp. 1–12.

[18] S. Chen, J. Fang, D. Chen, C. Xu, and Z. Wang, “Optimizing Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures,” May 2018. [Online]. Available: http://arxiv.org/abs/1805.11938

[19] H. Anzt, S. Tomov, and J. Dongarra, “Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs,” *University of Tennessee, Tech. Rep. ut-eecs-14-727*, 2014.

[20] W. Liu and B. Vinter, “CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*. Newport Beach, California, USA: Association for Computing Machinery, Jun. 2015, pp. 339–350.

[21] S. Beamer, K. Asanović, and D. Patterson, “Reducing Pagerank Communication via Propagation Blocking,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 820–831.