Advances in Engineering Software 156 (2021) 102997

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

# A new AXT format for an efficient SpMV product using AVX-512 instructions and CUDA☆,☆☆,★

E. Coronado-Barrientos [1,*,a], M. Antonioletti [2,b], A. Garcia-Loureiro [3,a]

[a] Department of Electronics and Computer Science and Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela Santiago de Compostela, España
[b] Edinburgh Parallel Computing Centre (EPCC), University of Edinburgh, Edinburgh, Scotland, UK

ARTICLE INFO

*Keywords:*
Sparse Matrix Vector product
AVX-512 instructions
MKL Library
CUDA
cuSPARSE Library
Segmented Scan algorithm

ABSTRACT

The Sparse Matrix-Vector (SpMV) product is a key operation used in many scientific applications. This work proposes a new sparse matrix storage scheme, the AXT format, that improves the SpMV performance on vector capability platforms. AXT can be adapted to different platforms, improving the storage efficiency for matrices with different sparsity patterns. Intel AVX-512 instructions and CUDA are used to optimise the performances of the four different AXT subvariants. Performance comparisons are made with the Compressed Sparse Row (CSR) and AXC formats on an Intel Xeon Gold 6148 processor and an NVIDIA Tesla V100 Graphics Processing Units using 26 matrices. On the Intel platform the overall AXT performance is 18% and 44.3% higher than the AXC and CSR respectively, reaching speed-up factors of up to x7.33. On the NVIDIA platform the AXT performance is 44% and 8% higher than the AXC and CSR performances respectively, reaching speed-up factors of up to x378.5.

## 1. Introduction and related work

Sparse matrices arise in numerous scientific and engineering problems ranging from electrophysiological simulations of the heart [1], including mechanical simulations of the deformation and the stress distribution in O-rings under pressure [2], to numerical simulations of lattice Quantum Chromodynamics (QCD) [3] or semiconductor devices [4]. Their efficient manipulation plays a major role in the performance of numerical applications developed to solve this kind of problems. Particularly, the Sparse Matrix-Vector (SpMV) product is of special interest because it often represents the largest time consuming operation in iterative methods needed to solve sparse linear systems and eigenvalue problems central to such applications [5–7]. Indeed, a myriad of numerical simulation applications, commercial and *ad hoc* solutions, uses non stationary iterative methods because of their high effectiveness and robustness whenever solving linear systems of equations [8]. The most popular solvers included in this category are: the Conjugated Gradient (CG) which requires one SpMV product per iteration [9], the Generalized Minimum Residual Method (GMRES) that also uses one SpMV product per iteration [4], the BiConjugate Gradient (BiCG) that needs two SpMV products per iteration [10], and the BiConjugate Gradient Stabilised (BiCGS) that also needs two SpMV products per iteration [11]. Optimising the SpMV product on modern multi and many-core processors for general sparse matrices is not a trivial task because, in order to harness the strong parallel-processing capabilities of these devices, the computations require to have regular execution paths

☆ This work is a result of support from the following projects: the FEDER funds, Xunta de Galicia ED431C 2018/19 and ED431F 2020/008, Spanish Ministry of Science and Technology PID2019-104834GB-I00, the Spanish Ministry of Science and Technology TIN2016-76373-P, and the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme; in addition, the authors gratefully acknowledges the support of Mr. Adrian Jackson EPCC and the computer resources and technical support provided by EPCC at the University of Edinburgh☆☆ This work used the Cirrus UK National Tier-2 HPC Service at EPCC Cirrus@EPCC. ★ The authors wish to acknowledge the Irish Centre for High-End Computing (ICHEC) for the provision of computational facilities and support.

* Corresponding author.
*E-mail addresses:* edoardoemilio.coronado@usc.es (E. Coronado-Barrientos), m.antonioletti@epcc.ed.ac.uk (M. Antonioletti), antonio.garcia.loureiro@usc.es (A. Garcia-Loureiro).
[1] [orcid=0000-0003-2274-0395]
[2] [orcid=0000-0002-2486-7990]
[3] [orcid=0000-0003-0574-1513]

https://doi.org/10.1016/j.advengsoft.2021.102997
Received 2 April 2020; Received in revised form 17 March 2021; Accepted 1 April 2021
Available online 18 April 2021

and memory access patterns, which are hardly ever present in the sparse matrices generated by real life numerical applications. Thus, it becomes necessary to rearrange the data structure of the matrix in such a way that this regularity can be enforced. In addition, in order to achieve maximum performance, the algorithms employed to perform the SpMV product using the rearranged data needs to take into account the technical specifications of the hardware that it is being targeted at and, if available, use the platform proprietary software tools for coding them.

The ELLPACK (ELL) format, introduced by Grimes in 1979 [12] for vector machines, was the first attempt to provide a sparse matrix with a regular data structure to force coalesced memory access. Since then, two major events have propelled the interest of researchers into developing new sparse matrix storage schemes for multi and many-core processors: the introduction of the Compute Unified Device Architecture (CUDA [13]) model in 2006 by NVIDIA for their Graphics Processing Units (GPUs) and the introduction of the 512-bit Advanced Vector Extensions (AVX-512 [14]) proposed by Intel in 2013 for their Knights Landing and Skylake architectures [15].

Even though both platforms are used in High Performance Computing (HPC), there seems to be a preference towards the NVIDIA GPUs evidenced by the greater number of formats developed for these devices when compared to those that include the Intel Xeon devices. This could be triggered because programmers were already using GPUs for General Purpose computing and the introduction of the CUDA model facilitated the use of GPUs because problems do not longer required to be masked as computer graphics tasks [16]. The list of formats proposed for GPUs is quite long, it is not the objective of this work to cover them all, but it is worth tackling some of these works that have provided new approaches that improved the performance of the SpMV product.

The early work of Bell and Garland in 2009 [5,6] covered the CUDA implementation of six formats: the Diagonal (DIA) format, the ELL format, the Coordinate (COO) format, the Compressed Sparse Row (CSR) format, the Hybrid (HYB) format, and the Packet (PKT) format. This paper demonstrated the importance of having not only a vector-friendly storage format, but also the importance of having the computational kernel customised to the platform, taking into account the technical specifications of the latter. This was highlighted by the introduction of two CUDA functions coded using the CSR format: the first function, referred to as the scalar kernel, executed a row computation per thread; the second function, referred to as the vector kernel, split a row computation between a group of threads (warp).

Despite the homogeneous computation and the coalesced memory access provided by the ELL format and kernel, its performance rapidly degrades when the number of non-zero elements per matrix row varies or when a matrix has an uncommonly large row. This shortcoming produced several variants that attempted to fix this problem. In 2009, Vazquez et al. proposed the ELLPACK-R (ELL-R) format whose aim was to avoid useless computations by indicating the matrices row lengths using an auxiliary array `rl[]` [17]. In 2010, Vazquez et al. revisited the ELL-R format and proposed the ELLR-T format that split the row computations between $T$ number of threads [18]. In 2010, Monakov et al. proposed their sliced ELLPACK (SELL) format that partitioned a matrix into strips of $S$ adjacent rows, and stored each strip in ELL format in order to reduce the zero over padding [19]. In the same year, Choi et al. published their blocked ELLPACK (BELL) format that stored a matrix into dense sub-blocks sorted in descending order by the number of blocks per row [20].

The Jagged Diagonals Storage (JDS) format [21,22] was another important format developed for vector machines that is at the same level as the ELL format because it served as base for important variants. The JDS format used the same arrays of the ELL format, but it introduced a sorting stage for the rows of the sparse matrix in order to reduce the memory footprint and avoid useless computation. Its variant, the padded Jagged Diagonals Storage (pJDS) format, introduced by Kreutzer et al. in 2012 [7], reduced the memory footprint of the ELL-R format by padding blocks of consecutive rows to the length of the longest row within each block.

In 2014, Kreutzer et al. introduced their sliced ELLPACK-C-$\sigma$ (SELL-C-$\sigma$) format [23]. This is the first work that proposed a single format for a variety of processor designs using some form of Single Instruction Multiple Data (SIMD) parallel model. The SELL-C-$\sigma$ was customised for different processors by selecting the parameter C ("chunk size") in accordance with the SIMD units width and fixing $\sigma$ ("sorting scope") for a multiple of C that was not too large. Despite the fact that this work used the Advanced Vector Extensions (AVX) for the Intel "Many Integrated Core" (MIC) architecture and CUDA for the NVIDIA platform, it only provided the computational kernel for the Intel platform. In 2015, Wong et al. published their ELLWARP format and their K1 algorithm (the CUDA equivalent to the SELL-C-$\sigma$ kernel) [1], but this work only targeted general purpose GPUs. In 2016, Altinkaynak proposed its ELLFEM format [24], based on the BELL format, for matrices generated by Finite Element Method (FEM) simulations on GPUs of uni-dimensional mechanical problems. Liu *et al.* developed the Compressed Sparse Row 5 (CSR5) format in 2017 [25], this is a variant of the CSR format that store the matrix in 2D arrays. Each array is split into segments called tiles, so a computational core can consume one or more 2D tiles, and each SIMD lane of a core can deal with the column of a tile. The CSR5 format used two parameters (one hardware-dependent and the other matrix sparsity-dependent) to customise the format for SIMD machines and for regular and irregular matrices. In summary, most of these formats were developed and tested only for GPUs, to the author's knowledge only the SELL-C-$\sigma$ and CSR5 formats have been developed and tested on two different programming model platforms.

Turning away from GPU platforms, the authors proposed the AXC format for sparse matrices oriented mainly at the Intel Xeon Phi coprocessor [11,26] in 2018. This format exploited the utilisation of the 512-bit registers of the coprocessor and the cache memory due to the data arrangement (64-Byte aligned segments to store the matrix entries and vector values).

Hence, in trying to widen the application of the AXC format, this work proposes a new sparse matrix storage that depends on three parameters to adapt itself to different architectures without affecting the scope for general sparse matrices. The proposed format can spawn four different data patterns depending on the values selected of these parameters. This proposal is tested on two devices: the Intel Xeon Gold 6148 processor with Skylake architecture and the NVIDIA Tesla V100 GPU with Volta architecture. Each of them has very distinct programming models. The AVX-512 instructions are used, in combination with OpenMP, to code the SpMV computational kernels for the Intel processor and CUDA is used for the computational kernels used on the NVIDIA GPU. It is worthwhile mentioning that there are other approaches for improving the performance of the SpMV product, e.g. Cataln *et al.* in [27] improved the SpMV product by optimising the workload balance among tasks by making better use of the platform resources using OmpSs directives.

The remainder of this paper is structured as follows: in Section 2 a brief overview of the formats used to compare the new proposal is given and an in depth description of the new AXT format. Section 3 describes the testing framework, along with all the information regarding the set of matrices used in this paper and the performance comparison of all the evaluated formats in the mentioned devices. Finally, Section 4 presents a summary of this paper.

## 2. Evaluated formats

This section covers the sparse matrix storage schemes evaluated in this work. The CSR and AXC formats are briefly described in 2.1 and 2.2 respectively, and the new proposal is introduced in detail in 2.3.

### 2.1. The CSR format

The CSR format is one of the most popular schemes used in a myriad

of numerical applications to date mainly due to its storage efficiency. In spite of its weaknesses, such as the unbalanced workload assigned to threads, the format still manages to be in demand as many relevant numerical libraries, such as the Intel Math Kernel Library (MKL) for Intel processors or the cuSPARSE for NVIDIA GPUs, still invest effort improving their CSR-based functions to perform the SpMV product. The CSR uses three arrays to store the matrix: `val[]` stores the matrix entries, `col[]` stores the column indices and `rwp[]` is used to pinpoint the starting position of the matrix's rows. An example of its data arrangement is shown in Figure 1. This work used the current state-of-the-art functions from the platforms' own libraries to achieve the maximum performance for the CSR format. For the Intel Xeon Gold processor, the Intel MKL function `mkl_sparse_d_mv` was used in conjunction with its optimisation functions and for the NVIDIA Tesla V100 GPU, the cuSPARSE function `cusparseCsrmvEx` was used.

### 2.2. The AXC format

The AXC format is a very simple scheme designed to target the Intel Xeon Phi coprocessor architecture. This format uses two arrays to store the matrix: `ax[]` stores the matrix's entries along with the vector's corresponding values contiguously in segments named *bricks*, and `brp[]` is used to indicate the starting position of the first brick belonging to each row. In addition to the two arrays, the AXC format requires the parameter `HBRS` to set the offset between a matrix entry and its corresponding vector value; and consequently the length of the bricks. An example of the AXC format is shown in Fig. 1. Because two elements per non zero entry of the matrix are stored in the `ax[]` array, the memory requirement becomes the major drawback of the AXC format compared to the CSR memory footprint. The computational kernel to perform the SpMV product using the AXC format and the AVX-512 instructions using `HBRS=8` is shown in Listing 1 (also in Listing 2 of [11]). The CUDA version of the SpMV function for the AXC format using `HBRS=32` is shown in Listing 2. It is worth remarking that because CUDA does not have a native function to perform the reduction step required by the SpMV product, this work used warp-level primitives to achieve the maximum performance on the NVIDIA GPU.

### 2.3. Proposal: the AXT format

The new AXT format (where the A and X stands for the matrix and vector respectively as commonly used in algebraic notation, and the T stands for *tiled*) which continues using the idea as presented by the AXC format of storing the matrix's entries and vector values contiguously in one array (`ax[]`) to avoid the performance penalties imposed by indirect memory accesses in order to improve the SpMV performance. This new proposal still uses only two arrays to store the sparse matrix, however, it now incorporates three tuning parameters in its design to reduce its memory footprint, adapt itself to different architectures and improve the load balance. The tuning parameters for the AXT format are: the tile's half width (`THW`), the tile's height (`TH`) and the mode (`MODE`).

The tile's half width (`THW`) is a machine dependent parameter set to the size of the SIMD execution unit of the targeted device. Thereby the SIMD unit can process a tile in `TH` steps without using any synchronisation barriers. As shown in Fig. 1, `THW` helps to set the width of the tile and consequently sets the space in the `ax[]` array between a matrix entry and its corresponding vector value. For double precision arithmetic the chosen values are: `THW=8` for the Intel Xeon Gold processor and `THW=32` for the NVIDIA Tesla V100 GPU, although other values (4, 8 and 16) were also tried for the GPU case.

The mode (`MODE`) parameter is key to the AXT format because it activates, or deactivates, the usage of zero padding between different

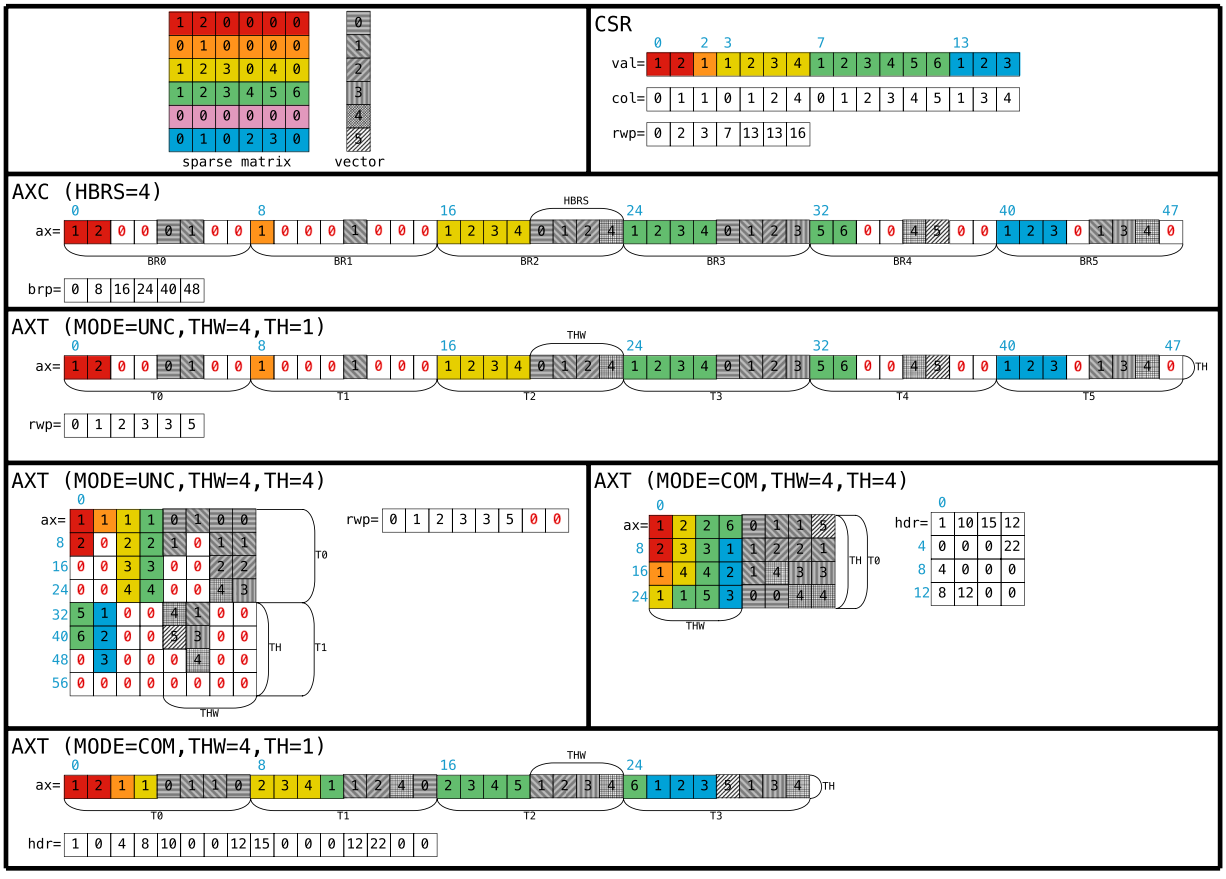


**Fig. 1.** The sparse matrix and the vector at the top left rectangle are used to illustrate the data arrangement for the CSR, AXC and AXT formats (there are four variants of the latter). Blue numbers indicate memory positions within the array. Red zeros are null elements required by the zero padding.

row elements in the ax[] array, impacting directly on the required memory footprint to store a matrix and establishing the function for the auxiliary array. There are only two options for this parameter: uncompacted (UNC) and compacted (COM). The uncompacted mode indicates that the elements in a row are going to be fitted to as many tiles, or tile's columns, as needed without mixing elements from a different row through the use of zero padding. It also sets the auxiliary array to store row indices (rwp[]). The compacted version joins the non zero elements of a matrix and their corresponding vector values together without any zero padding, which enables a tile, or tile's column, to contain elements from different rows. The compacted version changes the function of the auxiliary array (hdr[]), which contains a combination of row indices and offsets in this mode. Fig. 1 shows the data structure generated by each option of this parameter. Clearly, the uncompacted mode of the AXT has larger memory requirements than the compacted variants, but the memory overhead is counterbalanced by simpler kernels which leads to a better performance in most cases.

The tile's height (TH) parameter is more complex to identify because its correct value not only depend on the device's on-chip memory specifications, but its value also depends on the matrix's structure. The value of TH should be large enough to amortise the cost of the algorithms at a thread-level and yet small enough to avoid a performance reduction through an excess of null operations due to the zero padding in the uncompacted variants or a shared memory access penalisation in the compacted variants. Due to the fact that some of the matrices used in this work exhibit a large variability within their row population (e.g. matrices M07, M20, M21 and M24 in Table 1 on Subsection 3.2) the value of TH was obtained empirically, rather than assigning a value according to some matrix characteristic, such as the average number of elements per row.

The AXT format can generate four different data arrangements according to the values selected for the parameters TH and MODE, each of these data arrangements, or variants, require their own algorithm to perform the SpMV product. They are enumerated in the following list:

- the *AXTUH1* variant is an AXT format with MODE=UNC and TH=1,
- the *AXTUH* variant is an AXT format with MODE=UNC and TH>1,
- the *AXTCH1* variant is an AXT format with MODE=COM and TH=1 and
- the *AXTCH* variant is an AXT format with MODE=COM and TH>1;

in the remaining part of this subsection, each of these variants is described, along with an exhibition of the computational kernels implemented for each of the targeted devices used in this work.

### 2.3.1. The AXTUH1 variant

The uncompacted mode of the AXT format with TH=1 generates a data arrangement of the ax[] array identical to the one generated by the AXC format, as can be seen in Fig. 1. Both schemes store THW matrix entries with the corresponding THW vector values contiguously in segments named tiles (bricks in the AXC format) in the ax[] array in row major order. The difference between these two schemes lies in the auxiliary array. The new rwp[] array assigns the corresponding row index of its elements to each tile, hence for an ax[] array with NT tiles, the rwp[] array requires NT elements. This modification allows race conditions to appear as different threads processing different elements in the same row could try to write their partial results at the same time and position in the resulting array (y[]). This requires the use of atomic functions to ensure the correct computation of the result. The introduction of atomic functions reduced the performance of this variant on the Intel processor, but it helped to make the tiles' computation independent which favoured a better load balance and improved its overall performance on the NVIDIA GPUs. The Intel version of the SpMV kernel used an AVX-512 instruction for the reduction step of the partial results (Listing 3). Its CUDA counterpart (Listing 4) implemented the warp-level parallel reduction algorithm presented in [28] and is shown in Fig. 2. The warp-level primitives (__shfl_down_sync) enables a thread to access a value stored in another thread's register within the same warp (group of 32 threads).

### 2.3.2. The AXTUH variant

This is the 2D variant of the previous scheme, thus here the same arrays are used in the same way as its 1D predecessor (Fig. 1). Following the approach applied in [25] for the CSR5 format, this scheme stores the

**Table 1**
Matrices characteristics: *NNZ* is the number of non zero elements, *NROWS* is the number of rows, *RMIN* is the minimum number of elements in any row, *RAVE* is the average number of elements per row, *RMAX* is the maximum number of elements in any row, *RSD* is the standard deviation of the number of elements per row and *RSDP* is the normalised standard deviation of the number of elements per row.

| *MATRIX* | *NAME* | *NNZ* | *NROWS* | *RMIN* | *RAVE* | *RMAX* | *RSD* | *RSDP* |
|---|---|---|---|---|---|---|---|---|
| M00 | e001 | 159119 | 11931 | 5 | 13.3 | 29 | 3.2 | 24.1 |
| M01 | scircuit | 958936 | 170998 | 1 | 5.6 | 353 | 4.4 | 78.3 |
| M02 | mac_econ_fwd500 | 1273389 | 206500 | 1 | 6.2 | 44 | 4.4 | 71.9 |
| M03 | e002 | 1615118 | 121316 | 6 | 13.3 | 32 | 3.3 | 24.5 |
| M04 | mc2depi | 2100225 | 525825 | 2 | 4.0 | 4 | 0.1 | 1.9 |
| M05 | rma10 | 2329092 | 46835 | 1 | 49.7 | 145 | 27.7 | 55.7 |
| M06 | cop20k_A | 2624331 | 121192 | 0 | 21.7 | 81 | 13.8 | 63.7 |
| M07 | webbase-1M | 3105536 | 1000005 | 1 | 3.1 | 4700 | 25.4 | 816.1 |
| M08 | shipsec1 | 3568176 | 140874 | 4 | 25.3 | 68 | 10.7 | 42.1 |
| M09 | e003 | 3922721 | 279255 | 5 | 14.1 | 31 | 3.1 | 21.9 |
| M10 | cant | 4007383 | 62451 | 1 | 64.2 | 78 | 14.1 | 21.9 |
| M11 | pdb1HYS | 4344765 | 36417 | 18 | 119.3 | 204 | 31.9 | 26.7 |
| M12 | hamrle3 | 5514242 | 1447360 | 2 | 3.8 | 6 | 1.6 | 40.7 |
| M13 | consph | 6010480 | 83334 | 1 | 72.1 | 81 | 19.1 | 26.5 |
| M14 | g3_circuit | 7660826 | 1585478 | 2 | 4.8 | 6 | 0.6 | 13.3 |
| M15 | thermal2 | 8580313 | 1228045 | 1 | 7.0 | 11 | 0.8 | 11.6 |
| M16 | e004 | 9160735 | 635375 | 6 | 14.4 | 32 | 2.8 | 19.6 |
| M17 | pwtk | 11524432 | 217918 | 2 | 52.9 | 180 | 5.5 | 10.3 |
| M18 | kkt_power | 12771361 | 2063494 | 1 | 6.2 | 90 | 6.5 | 104.7 |
| M19 | memchip | 13343948 | 2707524 | 2 | 4.9 | 27 | 1.2 | 23.7 |
| M20 | in-2004 | 16917053 | 1382908 | 0 | 12.2 | 7753 | 37.2 | 304.3 |
| M21 | fullChip | 26621983 | 2987012 | 1 | 8.9 | 2312478 | 23.0 | 258.0 |
| M22 | e005 | 27367235 | 1873377 | 6 | 14.6 | 30 | 2.6 | 18.0 |
| M23 | delaunay_n23 | 50331568 | 8388608 | 3 | 6.0 | 28 | 1.3 | 22.3 |
| M24 | circuit5M | 59524291 | 5558326 | 1 | 10.7 | 1290501 | 99.3 | 927.3 |
| M25 | Serena | 64131971 | 1391349 | 1 | 46.1 | 249 | 9.9 | 21.5 |

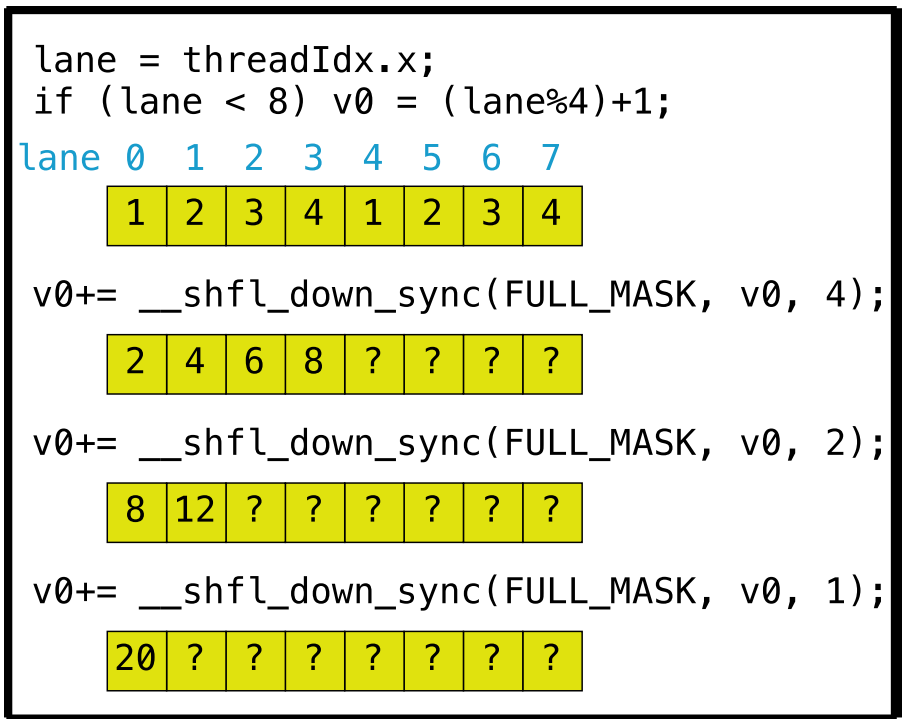


**Fig. 2.** Naive reduction using warp-level primitives for eight elements. Yellow cells contain the values of the registers where the values of the variable `v0` are stored. A question mark indicates unknown values because the threads read memory positions that are not being displayed in this example.

non zero elements of a matrix together with their corresponding vector values in a column major order separated by `THW` positions. Because this variant's mode is uncompacted, if a row has `NNZR` non zero elements this scheme will store its elements in `(NNZR + TH - 1)/TH` consecutive columns, avoiding that a column should have elements from different rows. If there are unused positions in the last column they will be padded with zeros as shown in Fig. 1. The 2D variant of the AXT format has another difference from its 1D counterpart, instead of storing one element per tile on the `rwp[]` array, it will store `THW` elements per tile, for a total of `NT*THW`, where `NT` is the number of tiles contained in the `ax[]` array. Unused columns will have a zero on their corresponding position on the `rwp[]` array as shown in Fig. 1. Storing the elements in column major order will guarantee coalesced memory access from SIMD lanes and enforces a one-step accumulation of the partial result from each column. The computational kernels for this variant are shown in Listing 5 and Listing 6 for an Intel processor and an NVIDIA GPU respectively.

Note that the CUDA version (Listing 6) uses the function `atomicAdd` to accumulate the result but the Intel version (Listing 5) does not use an atomic operation for the same task. The reason for this is that, despite the fact that the `rwp[]` array may point to the same position in the resulting array (`y[]`) for different columns in a tile, to the author's knowledge, there is no AVX-512 instruction [14] that performs an atomic write operation within the lanes of the same 512-bits register. Therefore, the required atomic add for the final step on the Intel version is substituted by a loop that performs the reading of a small static array (`tmp[]`) to accumulate correctly its results in the resulting array (`y[]`).

### 2.3.3. The AXTCH1 variant

Despite the simplicity of the data structure of the 1D compacted variant of the AXT format; it involves more complex algorithms than its uncompacted version to perform the SpMV product. Because this variant does not separate the elements from different rows through zero padding, the `ax[]` array may contain tiles that are mixing elements from different rows (see tiles T0, T1 and T3 for this variant in Fig. 1). Thus, this variant requires an algorithm that can perform more than one reduction within individual tiles, a sort of *segmented scan*. The segmented scan is an operation for performing separate parallel scans simultaneously on arbitrary contiguous partitions ("segments") on a given vector or array of numbers. This generalises the scan primitive by simultaneously performing separate parallel scans on arbitrary contiguous partitions of an input sequence. Segmented sequences require a segment descriptor that encodes how a sequence is divided into segments besides the sequence of values [29]. The left section of Fig. 3 shows an example of a segmented scan using the sum operation. The input arguments `is[]` and `sd[]` arrays contain the elements of the input sequence and the non zero elements (ones) that indicate the starting position of each partition within the sequence respectively. The result of the scan is stored in the `os[]` array.

Sengupta et al. in [29] presented an efficient parallel algorithm to perform the scan and segmented scan operations optimised for GPUs. However, their algorithm for the segmented scan operation has an overhead requiring two extra reads, one extra write and the use of two extra conditional statements for each element computation over their algorithm for the scan operation. Liu et al. in [25] published an algorithm named *fast segmented sum* as a simpler alternative to the more complex segmented scan algorithm. The right section of Fig. 3 shows an example of the fast segmented sum. The input arguments `is[]` and `off[]` arrays store the input sequence and the distance between the first and last element of each partition respectively; then step 1 (s1) stores the elements of `is[]` in the `tmp[]` array; step 2 (s2) performs a full scan operation on the `is[]` array; and step 3 (s3) applies Eq. (1):

$$os[i] = is[i + off[i]] - is[i] + tmp[i], \tag{1}$$

to the elements of `is[]` corresponding to the positions of the non zero values in the `off[]` array.

As described in the above paragraph, the fast segmented sum algorithm requires to perform a full scan operation on the input sequence. Hence, the computational kernels in Listings 8 and 9 implement the *block scan* algorithm presented in [29] to scan sequences of a fixed size. In order to use the block scan algorithm, there is an additional value that needs to be set, the block's length. For this reason the 1D compacted

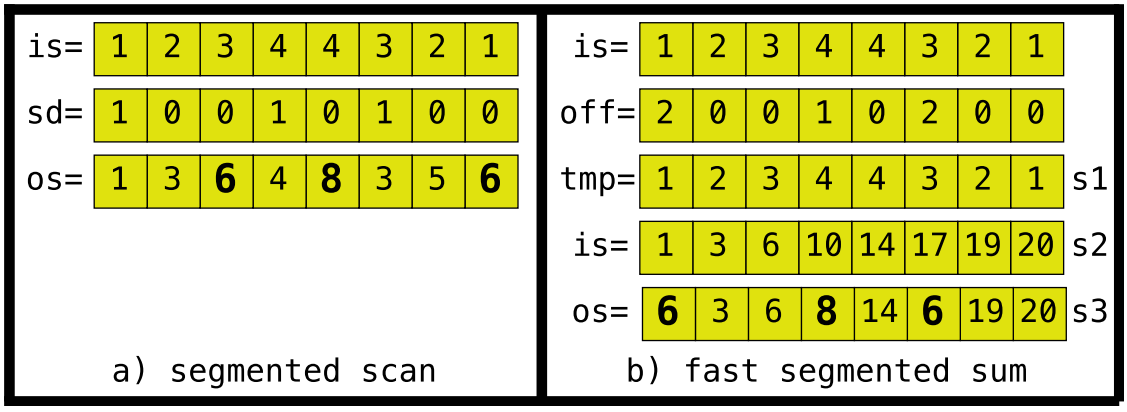


**Fig. 3.** a) The `is`[] and the `sd`[] arrays are the required input arguments for the segmented scan algorithm using the sum operation. b) The `is`[] and the `off`[] arrays are the required input arguments for the fast segmented sum algorithm. s1 (step 1) stores elements from the original sequence `is`[] in `tmp`[]. s2 performs a scan operation on the elements of `is`[]. s3 applies Eq. (1) to positions corresponding to the non zero values from the `off`[] array. The `os`[] arrays contain the output from the algorithms. The numbers in bold are the results for each partition within `os`[].

variant of the AXT format requires one additional parameter, the block's size (`BS`), to indicate the length of the sequence to be scanned. The block scan algorithm follows five steps:

1. scan the input sequence at the SIMD unit level,
2. collect the SIMD unit level results,
3. scan the SIMD unit level results,
4. accumulate the SIMD unit level results,
5. write and return the final results,

which are identified in Listing 9. Before going further into the kernels in Listings 8 and 9, the auxiliary array `hdr`[] must be described because it is used differently from the two previous variants of the AXT format.

The `hdr`[] array has `THW` elements for every tile contained in the `ax`[] array, that is, one element per matrix's entry. A non zero value is put at the beginning of each tile and wherever the first element of a row is found, the remaining positions are filled with zeros in order to use the fast segmented sum algorithm. This can be appreciated in the *AXTCH1* section from Fig. 1; where the tiles `T0` and `T2` each have three and one non zero entries in the `hdr`[] array respectively as the tiles have elements from three different rows and from the same row respectively. By structuring the `hdr`[] array in this way each partition formed from different rows can be distinguished within the tiles.

At this point, it can be deduced that the elements of the `hdr`[] array require to provide two pieces of data: the distance between the first and last element of each partition (referred to as the offset) and the row's index to which this partition belongs to. Therefore, the 32 bits (for a typical integer data type) of each non zero element in the `hdr`[] array are split into two parts through bit operations. The first bits of an element belonging to the `hdr`[] array (counting from the least significant bit (LSB)) are dedicated to specifying the offset of the partition. The number of bits for this task depends on the value of the `BS` parameter. For example, if `BS=1024`, then the maximum offset within a block is 1023, and it would require 10 bits to cover this value as $1023_{10} = 1111111111_2$. In order to obtain the offset of a partition, a bitwise `AND` operator needs to be applied between the `hdr`[] element and the maximum offset for the fixed block's size. The last bits of an element from the `hdr`[] array are dedicated to indicate the row's index of the partition, thus following the previous scenario, 22 bits would remain to indicate a row index, enough to cover matrices whose number of rows is lower than $2^{22} = 4,194,304$. This work covers matrices with higher number of rows than 4,194,304 by using a maximum block's size of 512 (for example, matrices M23 and M24 in Table 1). To obtain the row's index it is only necessary to shift the `hdr`[] element to the right by the number of positions equal to the number of digits employed for the offset.

After this explanation, the extraction of the offset and the row's index can be calculated using `BS=4` for the `hdr`[] elements of the compacted versions of the AXT format in Fig. 1. Also, both bitwise operations can be easily located in Listings 8 and 9.

The fast segmented sum requires to do a scan operation at a SIMD level, as it is indicated by the first item in the previous list. Two algorithms were employed in the SIMD level scan implementation. The first algorithm employed for the CUDA kernel is the warp scan [29] that uses the threads in a warp for 32 elements scans (Fig. 4 a)). The second algorithm implemented using the AVX-512 instructions (Listing 7) is the work-efficient scan [30,31] that uses the 512-bit registers for 8 double precision elements scans (see Fig. 4 b)).

Up to this point the kernels shown in Listings 8 and 9 are implementations of the algorithms described for this variant. Another factor to highlight is that the CUDA version only needs to use two shared memory arrays in order to use a maximum block size of 1024 elements, while the AVX-512 version requires four static memory arrays in order to use a maximum block size of 512 elements. This fact and the use of local memory from the registers favour the CUDA performance as the numerical results in Section 3 show.

#### 2.3.4. The AXTCH variant

The 2D compacted variant of the AXT format follows the same approach as the 2D uncompacted variant, which stores the matrix's entries and the corresponding vector values separated by `THW` elements in column-major order in the `ax`[] array. However, as with its 1D predecessor, this variant allows elements from more than one row to be mixed in the same column to reduce the memory footprint required. In order to perform the SpMV product, this variant still uses the fast segmented sum algorithm on the tile's columns as an alternative to the segmented scan algorithm. Therefore, the `hdr`[] array adopt the same ideas of the previous variant, but its elements are stored in column-major order producing a 2D array (Fig. 1). This way, the offset extracted from the `hdr`[] elements indicates the "vertical" distance between the first and last element of a partition, but not its position in the `hdr`[] array. Once the position of the last element of a partition is needed, it is calculated by multiplying the offset by `THW` as can be seen in Listings 10 and 11. Because the fast segmented sum is applied in column order, the maximum offset is set by the parameter `TH`, hence this variant does not require the parameter `BS`. The compacted mode allows the value for the parameter `TH` to be selected as large as desired without the storing penalty the uncompacted version could generate. However, having a large `TH` could lead to more partial sums that would produce a performance reduction due to more memory writes. Therefore the parameter

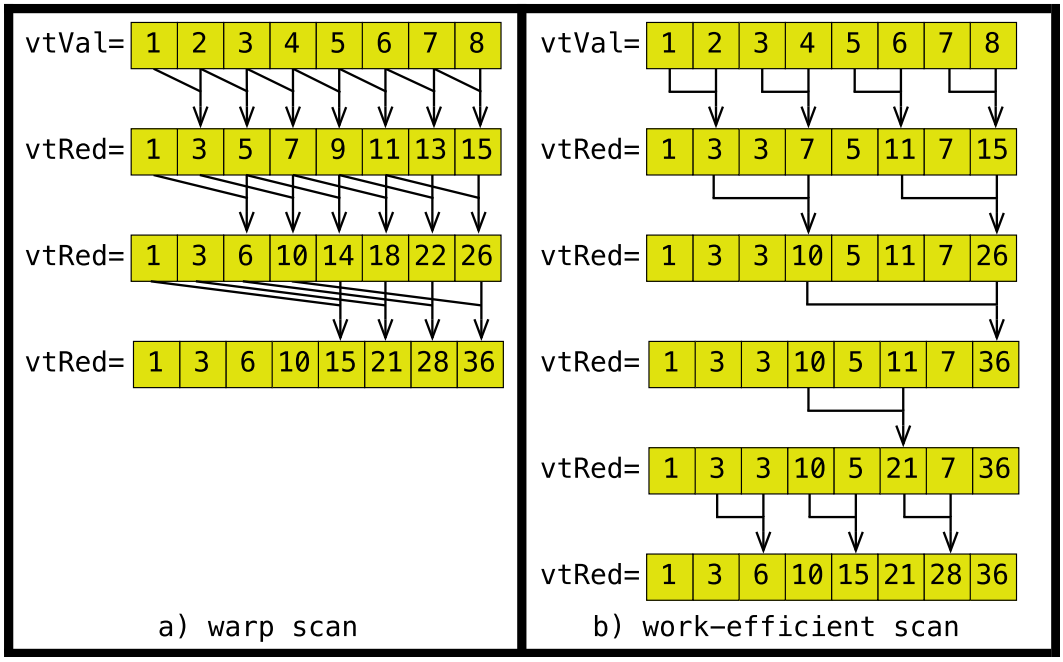


**Fig. 4.** a) The warp scan algorithm. Its implementation in CUDA using warp primitive instructions is shown in the first `for` loop in Listing 9. b) the work-efficient scan algorithm. Its implementation using the AVX-512 instructions is shown in Listing 7.

`TH` was not selected to be as large as the block size in the 1D compacted variant. This fact and the column-major order of the accumulations reduced the shared memory requirements of the computational kernels. Notice that in the Intel kernel Listing 10 only requires two static arrays: (`blk1`[] and `blk2`[]).

The CUDA kernel Listing 11 can do the algorithm using only local variables that are stored in the thread registers. The performance of the CUDA kernel is higher than the performance of the Intel kernel as can be seen by comparing the performances of this variant in Figs. 5 and 6. One of the factors for this development is that the CUDA kernel has all the data in the thread registers avoiding the use of shared memory.

## 3. Numerical results

This section provides information about the framework used for this study and the results obtained from the testing stage. Subsection 3.1 briefly covers the technical specifications of the devices where the tests were performed. Subsection 3.2 lists the matrices employed for testing

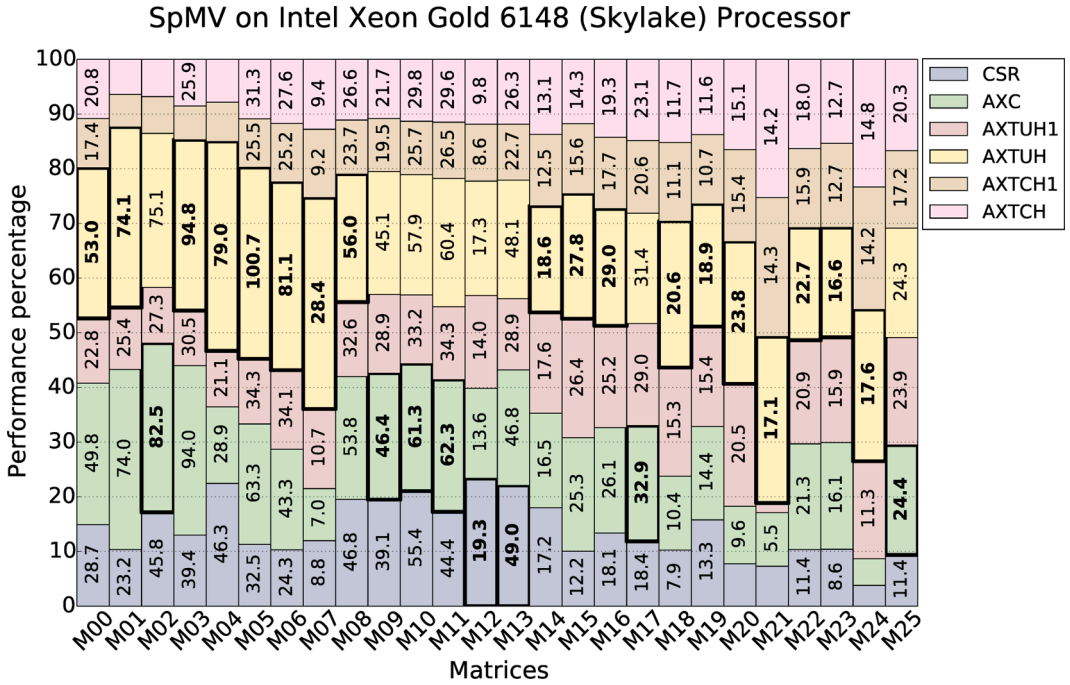


**Fig. 5.** The performance of the different formats on the Intel Xeon Gold 6148 processor. The values within the bars indicate the performance achieved by each variant in GFLOPS. Values corresponding to a performances inferior to 8 GFLOPS are not displayed. Bold fonts indicate the best performer for each matrix. The performances for each matrix are normalised to 100%.

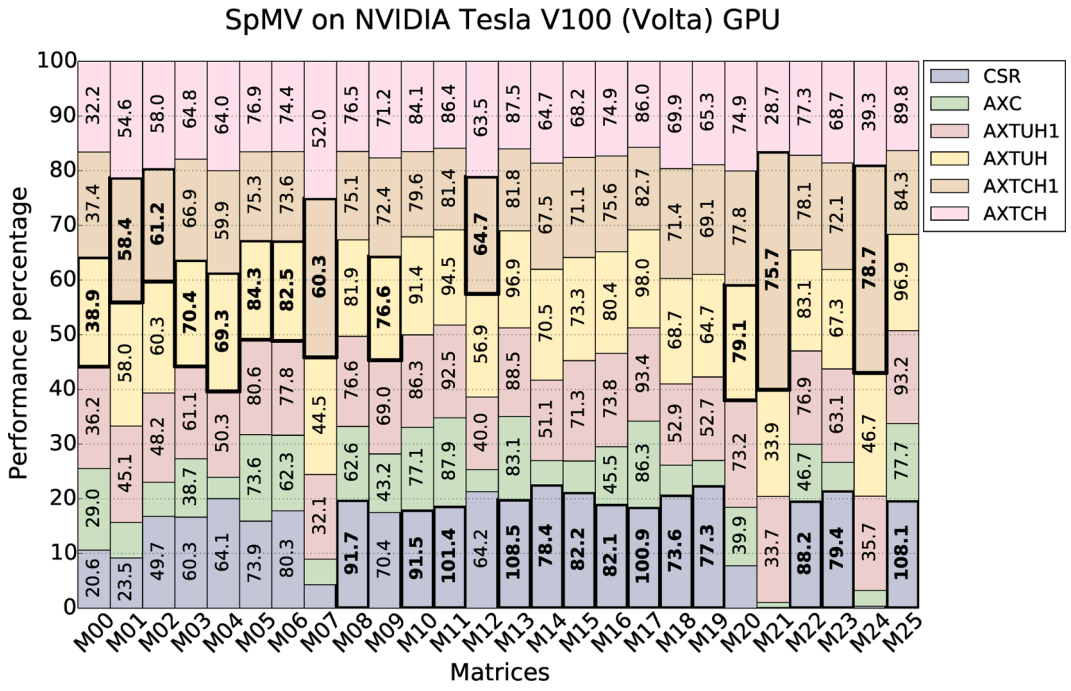


**Fig. 6.** The performance of the formats on an NVIDIA Tesla V100 GPU. The values within the bars indicate the performance achieved by each variant in GFLOPS. Values corresponding to performances below 8 GFLOPS are not displayed. Bold fonts indicate the best performer for each matrix. The performances for each matrix are normalised to 100%.

```
typedef unsigned int UIN;
typedef double FPT;
void axc( const UIN nt, const UIN hbs, const UIN nrows, const FPT *
      ax, const UIN * brp, FPT * y )
{
     const UIN stride = 2 * hbs;
           UIN rowID, pax;
           FPT red, sum;
     __m512d val, vec, pro;
     #pragma omp parallel for default(shared) private(rowID,pax,val,
          vec,pro,red,sum) num_threads(nt)
     for ( rowID = 0; rowID < nrows; rowID++ )
     {
          sum =  0;
          for ( pax = brp[rowID]; pax < brp[rowID+1]; pax = pax +
               stride )
          {
               val = _mm512_load_pd( &ax[pax]       );
               vec = _mm512_load_pd( &ax[pax + hbs] );
               pro = _mm512_mul_pd( val, vec );
               red = _mm512_reduce_add_pd( pro );
               sum = sum + red;
          }
          y[rowID] = sum;
     }
     return;
}
```

**Listing. 1.** AVX-512 kernel for AXC format (HBRS=8).

```
typedef unsigned int UIN;
typedef double FPT;
__global__ void gaxc( const UIN NROWS, const FPT * ax, const UIN *
      brp, FPT * y )
{
     const UIN tidGRID = blockIdx.x * blockDim.x + threadIdx.x;
     const UIN widGRID = tidGRID >> 5;
     if ( widGRID < NROWS )
     {
          const UIN tidWARP = tidGRID & 31;
          const UIN p1      = brp[widGRID]   + tidWARP;
          const UIN p2      = brp[widGRID+1] + tidWARP;
                UIN pax;
                FPT val = 0.0, red = 0.0;
          for ( pax = p1; pax < p2; pax = pax + 64 )
          {
               val = ax[pax] * ax[pax+32];
               val = val + __shfl_down_sync( FULL_MASK, val, 16 );
               val = val + __shfl_down_sync( FULL_MASK, val,  8 );
               val = val + __shfl_down_sync( FULL_MASK, val,  4 );
               val = val + __shfl_down_sync( FULL_MASK, val,  2 );
               val = val + __shfl_down_sync( FULL_MASK, val,  1 );
               red = red + val;
          }
          if (tidWARP == 0) y[widGRID] = red;
     }
     return;
}
```

**Listing. 2.** CUDA kernel for AXC format (HBRS=32).

the computational kernels, some pertinent characteristics regarding their structure, and the optimal values of the parameters used for each format's variant to achieve the maximum performance. Lastly, Subsection 3.3 provides a graphical comparison of the performance achieved by all variants in the two platforms utilised in this work.

### 3.1. Devices

The results shown in Figs. 5 and 6 were obtained from running the computational kernels listed in Section 2 on the GPU nodes of the Cirrus facility [32] at the EPCC, of the University of Edinburgh. Cirrus is a SGI/HPE 8600 Cluster that consists of 282 nodes, 280 of these are standard compute nodes and 2 of these contain GPU accelerators. The nodes that contain GPUs are referred as GPU nodes, an each of them contains two Intel Xeon Gold 6148 (Skylake) processors and four NVIDIA Tesla V100-PCIe (Volta) GPU accelerators. Therefore, the Cirrus facilities allowed the kernels to be tested using the Intel AVX-512 instructions set for the Skylake processors and CUDA version 9.1 on the same node. In order to avoid interference from other tasks, each SpMV job launched requested to use the two Intel Xeon Gold processors and the four GPUs, despite the fact that only one processor or GPU was targeted at a time.

The Intel Xeon Gold 6148 processor [33] has 20 cores running at a base frequency of 2.40 GHz with 27.5 MBytes L3 cache. Each core is able to support 2 hardware threads to give a total of 40 threads per processor (the parameter `nt` is set to 40 for Intel kernels). The maximum memory is 768 GBytes type DDR4 at 2666 MHz with a maximum bandwidth of 119.21 GBytes/s [34]. The processor supports the AVX-512 instructions set which allows simultaneous (vectorised) operations on eight double

```
typedef unsigned int UIN;
typedef double FPT;
void axt_unc_th1( UIN nt, UIN tn, UIN thw, FPT * ax, UIN * rwp, FPT
      * y )
{
     UIN stride = 2 * thw;
     UIN tid, pax, rid;
     FPT red;
     __m512d vtMat, vtVec, vtPro;
     #pragma omp parallel for default(shared) private(tid,pax,vtMat,
           vtVec,vtPro,red,rid) num_threads(nt)
     for ( tid = 0; tid < tn; tid++ )
     {
          pax   = tid * stride;
          vtMat = _mm512_load_pd( &ax[pax]       );
          vtVec = _mm512_load_pd( &ax[pax + thw] );
          vtPro = _mm512_mul_pd( vtMat, vtVec );
          red   = _mm512_reduce_add_pd( vtPro );
          rid   = rwp[tid];
          #pragma omp atomic
          y[rid] = y[rid] + red;
     }
     return;
}
```

**Listing. 3.** The AVX-512 kernel for the AXT format (with `MODE=UNC`, `THW=8`, `TH=1`), AXTUH1 variant.

```
typedef unsigned int UIN;
typedef double FPT;
__global__ void gaxt_unc_th1_thw16( const UIN TPW, const UIN TN,
       const FPT * ax, const UIN * rwp, FPT * y )
{
     const UIN tidGRID = blockIdx.x * blockDim.x + threadIdx.x;
     const UIN tidWARP = tidGRID & 31;
     const UIN ll      = (tidGRID >> 5) * TPW;
     const UIN ul      = ll + TPW;
           UIN t1, t2, r1, r2, p_ax1, p_ax2;
           FPT v0, v1, v2;
     for ( t1 = ll; t1 < ul; t1 = t1 + 2 )
     {
          t2    = t1 + 1;
          r1    = rwp[t1];
          r2    = rwp[t2];
          p_ax1 = t1 * 32 + tidWARP;
          p_ax2 = t2 * 32 + tidWARP;
          v1    = ax[p_ax1];
          v1    = v1 * __shfl_down_sync( FULL_MASK, v1, 16 );
          v2    = ax[p_ax2];
          v2    = v2 * __shfl_up_sync  ( FULL_MASK, v2, 16 );
          if (tidWARP < 16) v0 = v1;
          else              v0 = v2;
          v0    = v0 + __shfl_down_sync( FULL_MASK, v0, 8  );
          v0    = v0 + __shfl_down_sync( FULL_MASK, v0, 4  );
          v0    = v0 + __shfl_down_sync( FULL_MASK, v0, 2  );
          v0    = v0 + __shfl_down_sync( FULL_MASK, v0, 1  );
               if (tidWARP ==  0) atomicAdd( &y[r1], v0 );
          else if (tidWARP == 16) atomicAdd( &y[r2], v0 );
     }
     return;
}
```

**Listing. 4.** The CUDA kernel for the AXT format (with `MODE=UNC`, `THW=16`, `TH=1`), AXTUH1 variant.

precision data type, and Fused Multiply-Add (FMA) instructions which executes two floating point operations (FLOPS) per cycle. The theoretical maximum peak performance can be calculated by multiplying the number of operations per cycle (two assuming only FMA operations are being executed) by the number of cores, by the SIMD unit length by the processor speed [35], that is: $2 \times 20 \times 8 \times 2.4 = 768$ GFLOPS.

The NVIDIA Tesla V100 PCIe GPU has 5120 CUDA cores with 16 GBytes of memory. Its theoretical maximum peak performance is 7 TFLOPS for double precision data type [36]. This GPU has a compute capability of 7.0 which includes CUDA atomic functions for double precision data type.

### 3.2. Matrices

Table 1 list the matrices used in this work. Most of these matrices belong to the Williams, Boeing and Janna suites whose characteristics along with their sparsity patterns graphs can be consulted in [37]. The matrices M00, M03, M09, M16 and M22 were generated using the semiconductor device numerical simulator used in [4]; these matrices

```
typedef unsigned int UIN;
typedef double FPT;
void axt_unc( UIN nt, UIN tn, UIN th, UIN thw, FPT * ax, UIN * rwp,
      FPT * y )
{
     UIN stride = 2 * thw;
     UIN ts     = th * stride;
     UIN tid, pax, prwp, rid, i;
     FPT tmp[thw];
     FPT red;
     __m512d vtMat, vtVec, vtPro, vtSum;
     __m512i vtRid;
     #pragma omp parallel for default(shared) private(tid,vtSum,pax,
           vtMat,vtVec,vtPro,tmp,i,prwp,rid) num_threads(nt)
     for ( tid = 0; tid < tn; tid++ )
     {
          vtSum = _mm512_setzero_pd();
          for ( pax = tid * ts; pax < (tid + 1) * ts; pax = pax +
                stride )
          {
               vtMat = _mm512_load_pd( &ax[pax]       );
               vtVec = _mm512_load_pd( &ax[pax + thw] );
               vtPro = _mm512_mul_pd( vtMat, vtVec );
               vtSum = _mm512_add_pd( vtSum, vtPro );
          }
          _mm512_store_pd( tmp, vtSum );
          for ( i = 0, prwp = tid * thw; i < thw; i++, prwp++ )
          {
               rid    = rwp[prwp];
               y[rid] = y[rid] + tmp[i];
          }
     }
     return;
}
```

**Listing. 5.** The AVX-512 kernel for the AXT format (with `MODE=UNC`, `THW=8`, `TH>1`), AXTUH variant.

```
typedef unsigned int UIN;
typedef double FPT;
__global__ void gaxt_unc( const UIN TPW, const UIN TN, const UIN TH,
       const FPT * ax, const UIN * rwp, FPT * y )
{
     const UIN tidGRID = blockIdx.x * blockDim.x + threadIdx.x;
     const UIN tidWARP = tidGRID & 31;
     const UIN widGRID = (tidGRID >> 5);
     const UIN ll      = widGRID * TPW;
     const UIN ul      = ll + TPW;
           UIN t, p1, p2, p_ax, r;
           FPT val;
     for ( t = ll; t < ul; t++ )
     {
          p1  = t * TH * 64 + tidWARP;
          p2  = p1 + TH * 64;
          val = 0.0;
          for ( p_ax = p1; p_ax < p2; p_ax = p_ax + 64 )
               val = val + ax[p_ax] * ax[p_ax+32];
          r = rwp[t*32 + tidWARP];
          atomicAdd( &y[r], val );
     }
     return;
}
```

**Listing. 6.** The CUDA kernel for the AXT format (with `MODE=UNC`, `THW=32`, `TH>1`), AXTUH variant.

have an arrow-head sparsity pattern. All the matrices are square and ordered by the number of non zero elements (*NNZ*) in ascending order in Table 1. Besides the arrow-head sparsity pattern, two more patterns were identified: the band diagonal (e.g. M04, M10, M25) and irregular sparsity patterns such as those exhibited by matrices M07, M21 and M24 that have abnormally large rows within their ranks. These different sparsity patterns served to test the AXT format as a general matrix format.

Tables 2 and 3 show the memory footprint required to store the matrices (in MBytes) using double precision by each variant and the optimal parameter's values for the schemes tested for the Intel and NVIDIA platforms respectively. Most of these parameters have already been introduced and explained in Section 2. However, there is an important metric that needs to be introduced at this point in order to interpret all the information from Tables 2 and 3. Kreutzer et al. in [23] introduced the storage occupancy ($\beta$) as:

$$\beta = \frac{NNZ}{SE}, \tag{2}$$

```
__m512d inc_scan8( __m512d vtVal )
{
     __m512i  vtIdx = _mm512_set_epi64( 6, 6, 4, 4, 2, 2, 0, 0 );
     __m512d  vtAux = _mm512_permutexvar_pd( vtIdx, vtVal );
     __mmask8 cmask = 0xAA;
     __m512d  vtRed = _mm512_mask_add_pd( vtVal, cmask, vtVal, vtAux
             );
              vtIdx = _mm512_set_epi64( 5, 6, 5, 4, 1, 2, 1, 0 );
              vtAux = _mm512_permutexvar_pd( vtIdx, vtRed );
              cmask = 0x88;
              vtRed = _mm512_mask_add_pd( vtRed, cmask, vtRed, vtAux
                      );
              vtIdx = _mm512_set_epi64( 3, 6, 5, 4, 3, 2, 1, 0 );
              vtAux = _mm512_permutexvar_pd( vtIdx, vtRed );
              cmask = 0x80;
              vtRed = _mm512_mask_add_pd( vtRed, cmask, vtRed, vtAux
                      );
              vtIdx = _mm512_set_epi64( 7, 6, 3, 4, 3, 2, 1, 0 );
              vtAux = _mm512_permutexvar_pd( vtIdx, vtRed );
              cmask = 0x20;
              vtRed = _mm512_mask_add_pd( vtRed, cmask, vtRed, vtAux
                      );
              vtIdx = _mm512_set_epi64( 7, 5, 5, 3, 3, 1, 1, 0 );
              vtAux = _mm512_permutexvar_pd( vtIdx, vtRed );
              cmask = 0x54;
              vtRed = _mm512_mask_add_pd( vtRed, cmask, vtRed, vtAux
                      );
     return( vtRed );
}
```

**Listing. 7.** The `inc_scan8` function is a 512-bit register scan operation that uses the workload balancing algorithm.

where *SE* is the number of elements stored in the scheme. The higher the number of elements stored above *NNZ* by the format the lower the storage occupancy is, which indicates an inferior storage efficiency for the format. Clearly, the compacted variants of the AXT format have the best storage efficiency (1.0) as they store all non zero elements in contiguous positions. The Tables 1, 2 and 3are self-explanatory, however, in order to clarify the information two examples of matrix information reading are given here for Tables 2 and 3. For the first example, the information from Table 2, row M21 and column *AXTUH* reads 575.9 [0.76] ( 8), this means that the uncompacted variant of the AXT format requires 575.9 MBytes to store the matrix, it has a storage efficiency of $\beta = 0.76$ (138.2 MBytes are wasted on zero padding). In this case, its maximum performance for the Intel Xeon Gold processor is achieved with *TH*=8 and *THW*=8. For the second example, the information from Table 3, row M14 and column *AXTCH1* is 153.2 $< 1024 >$, this is interpreted as, the compacted variant of the AXT format needs 153.2 MBytes to store the matrix. Note that this format has an optimal storage efficiency of $\beta = 1.0$, because there is no overhead due to zero padding. Under this configuration, its maximum performance on the NVIDIA GPU is achieved with *TH*=1, *THW*=32 and *BS*=1024.

### 3.3. Performance

The performance, shown in Figs. 5 and 6, was calculated through the application of the following formula:

$$P = \frac{2 \times NNZ}{T}, \tag{3}$$

where $P$ is the performance (in FLOPS), $2 \times NZZ$ is the total number of floating point operations required to carry out the SpMV product and $T$ is the mean time (in seconds) it takes to complete 100 executions for each scheme described in Section 2. In order to ensure the time is being correctly measured, the function `clock_gettime` with the `CLOCK_MONOTONIC` option was employed on the Intel Xeon Gold processor and the function `cudaEventElapsedTime` on the NVIDIA Tesla V100 GPU. Because there are large differences in performance between the matrix products, the overall performance by each matrix was normalised to 100, see Figs. 5 and 6. As an example of how to interpret these figures consider the M00 matrix in Fig. 5 for the Intel Xeon Gold processor; it can be observed that the overall performance for this matrix is: $28.7 + 49.8 + 22.8 + 53.0 + 17.4 + 20.8 = 192.5$ GFLOPS and that the *AXTUH* variant performed best achieving 53.0 GFLOPS that is approximately

```
typedef unsigned int UIN;
typedef double FPT;
void axt_com_th1_bs512( const UIN nt, const UIN bn, const UIN lenHDR
      , const FPT * ax, const UIN * hdr, FPT * y )
{
          UIN bid, blkOff, tid, pax, pblk, ep, ro, r, o;
       __m512d vtMat, vtVec, vtVal, vtRed, vtAcu;
          FPT v;
          FPT blk1[512];
          FPT blk2[512];
          FPT blk3[64];
          FPT blk4[8];
    #pragma omp parallel for default(shared) private(bid,blkOff,tid
           ,pax,pblk,ep,ro,r,o,vtMat,vtVec,vtVal,vtRed,vtAcu,v,blk1
           ,blk2,blk3,blk4) num_threads(nt)
    for ( bid = 0; bid < bn; bid++ )
    {
         blkOff = bid * 1024;
         for ( tid = 0; tid < 64; tid++ )
         {
              pax       = blkOff + tid * 16;
              vtMat     = _mm512_load_pd( &ax[pax]     );
              vtVec     = _mm512_load_pd( &ax[pax + 8] );
              vtVal     = _mm512_mul_pd( vtMat, vtVec );
              pblk      = tid *  8;
              _mm512_store_pd( &blk1[pblk], vtVal );
              vtRed     = inc_scan8( vtVal );
              _mm512_store_pd( &blk2[pblk], vtRed );
              blk3[tid] = blk2[pblk+7];
         }
         for ( tid = 0; tid < 8; tid++ )
         {
              pblk      = tid *  8;
              vtVal     = _mm512_load_pd( &blk3[pblk] );
              vtRed     = inc_scan8( vtVal );
              _mm512_store_pd( &blk3[pblk], vtRed );
              blk4[tid] = blk3[pblk+7];
         }
         vtVal = _mm512_load_pd( blk4 );
         vtRed = inc_scan8( vtVal );
         _mm512_store_pd( blk4, vtRed );
         for ( tid = 1; tid < 8; tid++ )
         {
              pblk   = tid * 8;
              vtAcu  = _mm512_set1_pd( blk4[tid-1]   );
              vtVal  = _mm512_load_pd( &blk3[pblk] );
              vtRed  = _mm512_add_pd( vtAcu, vtVal );
              _mm512_store_pd( &blk3[pblk], vtRed );
         }
         for ( tid = 1; tid < 64; tid++ )
         {
              pblk   = tid * 8;
              vtAcu  = _mm512_set1_pd( blk3[tid-1]   );
              vtVal  = _mm512_load_pd( &blk2[pblk] );
              vtRed  = _mm512_add_pd( vtAcu, vtVal );
              _mm512_store_pd( &blk2[pblk], vtRed );
         }
         blkOff = bid * 512;
         ep     = blkOff + 512;
         ep     = (ep>lenHDR) ? lenHDR : ep ;
         for ( pblk = blkOff, tid = 0; pblk < ep; pblk++, tid++ )
         {
              ro = hdr[pblk];
              if (ro!=0)
              {
                   r = ro >> 9;
                   o = ro & 511;
                   v = blk2[tid+o] - blk2[tid] + blk1[tid];
                   #pragma omp atomic
                   y[r] = y[r] + v;
              }
         }
    }
    return;
}
```

**Listing. 8.** The AVX-512 kernel for the AXT format (with `MODE=COM`, `THW=8`, `TH=1`, `BS=512`), AXTCH1 variant.

28% of the overall performance $\left(\frac{53.0}{192.5} \times 100 = 27.5\%\right)$.

#### 3.3.1. Intel Xeon Gold processor

The following list enumerates the observations of the performances obtained for the Intel Xeon Gold processor shown in Fig. 5:

1. The best performer was the *AXTUH* variant that achieved a total of 1139.4 GFLOPS averaging 43.8 GFLOPS per matrix; it outperformed the other schemes in 18 out of 26 matrices.

```cuda
typedef unsigned int UIN;
typedef double FPT;
__global__ void gaxt_com_h1( const UIN LOG, const FPT * ax, const
      UIN * hdr, FPT * y )
{
     const UIN tidBLCK = threadIdx.x;
     const UIN widBLCK = tidBLCK >> 5;
     const UIN tidWARP = tidBLCK & 31;
     const UIN pAX     = blockIdx.x * 2 * blockDim.x + widBLCK * 64
          + tidWARP;
     const UIN ro      = hdr[blockIdx.x * blockDim.x + tidBLCK];
           UIN r, o, i;
           __shared__ FPT blk1[32];
     extern __shared__ FPT blk2[];
           FPT vo = 0.0, v1 = 0.0, v2 = 0.0, v3 = 0.0;
     blk1[tidWARP] = 0.0;
     blk2[tidBLCK] = 0.0;
     __syncthreads();
     vo = ax[pAX] * ax[pAX+32];
     v1 = vo;
     __syncthreads();
     // step 1
     for ( i = 1; i <=16; i = i * 2 )
     {
          v2 = __shfl_up_sync( FULL_MASK, v1, i );
          if (tidWARP >= i) v1 = v1 + v2;
     }
     __syncthreads();
    // step 2
     if (tidWARP == 31) blk1[widBLCK] = v1;
     __syncthreads();
     // step 3
     if (widBLCK == 0)
     {
          v2 = blk1[tidWARP];
          for ( i = 1; i <=16; i = i * 2 )
          {
               v3 = __shfl_up_sync( FULL_MASK, v2, i );
               if (tidWARP >= i) v2 = v2 + v3;
          }
          blk1[tidWARP] = v2;
     }
     __syncthreads();
     // step 4
     if (widBLCK > 0) v1 = v1 + blk1[widBLCK-1];
     __syncthreads();
     // step 5
     blk2[tidBLCK] = v1;
     __syncthreads();
     if (ro)
     {
          r  = ro >> LOG;
          o  = ro & (blockDim.x - 1);
          v1 = blk2[tidBLCK + o] - v1 + vo;
          atomicAdd( &y[r], v1 );
     }
     return;
}
```

**Listing. 9.** The CUDA kernel for the AXT format (with `MODE=COM, THW=32, TH=1, BS=512`), AXTCH1 variant.

2. The second best performer was the *AXC* kernel that achieved a total 932.6 GFLOPS averaging 35.9 GFLOPS per matrix; it won over the remaining competitors in 6 out 26 matrices.
3. The third best performer was the *CSR* function which achieved a total of 635.1 GFLOPS averaging 24.4 GFLOPS per matrix; the *CSR* function registered 2 out 26 winning cases.
4. The *AXTUH* kernel variant represents an 18.1% increase in overall performance over the *AXC* kernel and a 5.1% reduction in the total memory footprint.
5. The *AXTUH* variant represents a 44.3% increase in the overall performance over the *CSR* function and a 34.6% increase in the total memory footprint.
6. Despite the similarities between the *AXC* kernel and the *AXTUH1* variant, the latter performed a 35.6% worse than the former. This could be caused by race conditions generated from splitting the tiles belonging to the rows, instead of performing the accumulation of each row's result sequentially.
7. The worst performers on the Intel Xeon Gold processor were the *AXTCH1* and *AXTCH* variants. This could be explained by the fact that their kernels, and algorithms, require access to off-chip memory storage because of the `blk1[]`-`blk4[]` static arrays.
8. Most of the formats achieved their best performances on small matrices with less than 6M non zero elements.

```cuda
typedef unsigned int UIN;
typedef double FPT;
void axt_com( const UIN nt, const UIN tn, const UIN th, const UIN
      log, const UIN thw, const FPT * ax, const UIN * hdr, FPT * y
      )
{
     const UIN ths = th * thw;
     const UIN ts  =  2 * ths;
           UIN tid, off, f, pax, pblk, ro, r, o;
      __m512d vtMat, vtVec, vtVal, vtAcu;
           FPT blk1[th*thw];
           FPT blk2[th*thw];
           FPT v;
     #pragma omp parallel for default(shared) private(tid,off,f,pax,
          pblk,ro,r,o,vtMat,vtVec,vtVal,vtAcu,v,blk1,blk2)
          num_threads(nt)
     for ( tid = 0; tid < tn; tid++ )
     {
          off = tid * ts;
          vtAcu = _mm512_setzero_pd();
          for ( f = 0; f < th; f++ )
          {
               pax   = off + f * 16;
               vtMat = _mm512_load_pd( &ax[pax]     );
               vtVec = _mm512_load_pd( &ax[pax + 8] );
               vtVal = _mm512_mul_pd( vtMat, vtVec );
               pblk  = f * 8;
               _mm512_store_pd( &blk1[pblk], vtVal );
               vtAcu = _mm512_add_pd( vtAcu, vtVal );
               _mm512_store_pd( &blk2[pblk], vtAcu );
          }
          off = tid * ths;
          for ( f = 0; f < ths; f++ )
          {
               ro = hdr[off + f];
               if (ro != 0)
               {
                    r    = ro >> log;
                    o    = ro & (th-1);
                    v    = blk2[f+(o*thw)] - blk2[f] + blk1[f];
                    #pragma omp atomic
                    y[r] = y[r] + v;
               }
          }
     }
     return;
}
```

**Listing. 10.** The AVX-512 kernel for the AXT format (with `MODE=COM, THW=8, TH>1`), AXTCH variant.

```cuda
typedef unsigned int UIN;
typedef double FPT;
__global__ void gaxt_com( const UIN LOG, const UIN TH, const FPT *
      ax, const UIN * hdr, FPT * y )
{
     const UIN tidGRID = blockIdx.x * blockDim.x + threadIdx.x;
     const UIN widGRID = tidGRID >> 5;
     const UIN tidWARP = tidGRID & 31;
     const UIN THS     = TH * 32;
     const UIN ul_hdr  = (widGRID + 1) * THS;
     const UIN TS      = TH * 64;
     const UIN ul_ax   = (widGRID + 1) * TS;
           UIN a1_hdr, ro, r, o, a1_ax, a2_ax, p_ax;
           FPT red;
     a1_hdr = widGRID * THS + tidWARP;
     ro     = hdr[a1_hdr];
     r      = ro >> LOG;
     o      = (ro & (TH-1)) * 64;
     a1_ax  = widGRID * TS + tidWARP;
     a2_ax  = a1_ax + o;
     do {
          red    = 0.0;
          for ( p_ax = a1_ax; p_ax <= a2_ax; p_ax = p_ax + 64 )
          {
               red    = red + ax[p_ax] * ax[p_ax+32];
               a1_hdr = a1_hdr + 32;
          }
          atomicAdd( &y[r], red );
          if (a1_hdr < ul_hdr)
          {
               ro = hdr[a1_hdr];
               r      = ro >> LOG;
               o      = (ro & (TH-1)) * 64;
               a1_ax  = p_ax;
               a2_ax  = a1_ax + o;
          }
     } while (p_ax < ul_ax);
     return;
}
```

**Listing. 11.** The CUDA kernel for the AXT format (with `MODE=COM, THW=32, TH>1`), AXTCH variant.

**Table 2**
Storage and parameter information for all format variants on the Intel Xeon Gold 6148 processor. The memory footprint is indicated by the numbers without brackets in MBytes. The storage occupancy $\beta$ Eq. (2) is indicated by the numbers enclosed in square brackets. The storage occupancy is 1.00 for the variants *CSR, AXTCH1* and *AXTCH*. The brick's half size (HBRS) of the AXC format is equivalent to the tile's half width (THW) of the AXT format. Both parameters are set to 8 for the variants *AXC, AXTUH1, AXTUH, AXTCH1* and *AXTCH*. The tile's height (TH) is indicated by the numbers enclosed in parentheses. The tile's height is 1 for the variants *AXTUH1* and *AXTCH1*. Lastly, the variant *AXTCH1* uses block's size (BS) equals to 512.

| *MATRIX* | *CSR* | *AXC* | *AXTUH1* | *AXTUH* | *AXTCH1* | *AXCTCH* |
|---|---|---|---|---|---|---|
| M00 | 2.1 | 3.3 [0.79] | 3.3 [0.79] | 3.6 [0.72] (16) | 3.2 | 3.2 (64) |
| M01 | 13.6 | 25.3 [0.62] | 25.4 [0.62] | 25.4 [0.62] (8) | 19.2 | 19.2 (64) |
| M02 | 17.8 | 32.7 [0.64] | 32.9 [0.64] | 32.9 [0.64] (8) | 25.5 | 25.5 (64) |
| M03 | 20.8 | 33.7 [0.78] | 34.3 [0.78] | 36.8 [0.72] (12) | 32.3 | 32.3 (64) |
| M04 | 31.5 | 69.4 [0.50] | 69.4 [0.50] | 35.8 [1.00] (4) | 42.0 | 42.0 (64) |
| M05 | 28.5 | 39.7 [0.94] | 40.7 [0.94] | 41.9 [0.91] (12) | 46.6 | 46.6 (64) |
| M06 | 33.0 | 45.9 [0.92] | 46.9 [0.92] | 48.1 [0.89] (12) | 52.5 | 52.5 (64) |
| M07 | 49.3 | 144.9 [0.35] | 145.3 [0.35] | 84.3 [0.63] (4) | 62.1 | 62.1 (8) |
| M08 | 44.5 | 65.7 [0.88] | 67.2 [0.88] | 64.4 [0.94] (4) | 71.4 | 71.4 (64) |
| M09 | 50.4 | 79.3 [0.80] | 80.6 [0.80] | 80.6 [0.80] (8) | 78.5 | 78.5 (64) |
| M10 | 48.8 | 69.3 [0.93] | 71.2 [0.93] | 71.6 [0.91] (20) | 80.2 | 80.2 (64) |
| M11 | 52.6 | 71.7 [0.97] | 73.8 [0.97] | 73.7 [0.96] (12) | 86.9 | 86.9 (256) |
| M12 | 83.5 | 191.1 [0.48] | 191.1 [0.48] | 155.5 [0.60] (4) | 110.3 | 110.3 (8) |
| M13 | 73.1 | 104.0 [0.93] | 106.9 [0.93] | 102.6 [0.96] (12) | 120.2 | 120.2 (128) |
| M14 | 111.0 | 209.3 [0.60] | 209.3 [0.60] | 191.5 [0.68] (4) | 153.2 | 153.2 (512) |
| M15 | 117.7 | 166.4 [0.85] | 166.5 [0.85] | 166.5 [0.85] (8) | 171.6 | 171.6 (128) |
| M16 | 117.6 | 182.0 [0.82] | 185.1 [0.82] | 172.0 [0.91] (4) | 183.2 | 183.2 (128) |
| M17 | 140.9 | 192.9 [0.96] | 198.0 [0.96] | 198.0 [0.96] (8) | 230.5 | 230.5 (512) |
| M18 | 178.0 | 349.2 [0.60] | 351.6 [0.60] | 266.9 [0.81] (4) | 255.4 | 255.4 (64) |
| M19 | 192.6 | 358.6 [0.61] | 358.6 [0.61] | 290.2 [0.78] (4) | 266.9 | 266.9 (64) |
| M20 | 219.6 | 346.3 [0.79] | 351.4 [0.79] | 318.3 [0.90] (4) | 338.3 | 338.4 (128) |
| M21 | 355.3 | 570.4 [0.76] | 575.9 [0.76] | 575.9 [0.76] (8) | 532.4 | 532.5 (128) |
| M22 | 350.9 | 538.0 [0.83] | 547.0 [0.83] | 512.9 [0.91] (4) | 547.3 | 547.4 (256) |
| M23 | 704.6 | 1149.1 [0.72] | 1150.4 [0.72] | 1095.7 [0.78] (4) | 1006.6 | 1006.6 (128) |
| M24 | 781.0 | 1253.0 [0.77] | 1269.2 [0.77] | 1269.2 [0.77] (8) | 1190.5 | 1190.5 (128) |
| M25 | 786.3 | 1106.8 [0.93] | 1135.6 [0.93] | 1126.0 [0.97] (4) | 1282.6 | 1282.7 (512) |

9. The *AXTUH* variant achieved outstanding speed up factors over the *AXC* kernel for the following matrices: M04 (x2.73), M07 (x4.06), M18 (x1.98), M20 (x2.48), M21 (x3.11) and M24 (x5.68).
10. The *AXTUH* variant achieved outstanding speed up factors over the *CSR* function for the following matrices: M01 (x3.19), M03 (x2.41), M05 (x3.10), M06 (x3.34), M07 (x3.23), M15 (x2.28), M18 (x2.61), M20 (x3.35), M21 (x4.17), M24 (x7.33) and M25 (x2.73).
11. The *AXTUH* variant achieved the best performance using TH=4 or 8 for most cases. While the *AXTCH* did the same task using TH=64 or 128 for most cases.

In summary the *AXTUH* variant gave the best performance on the Intel Xeon Gold 6148 processor of all the schemes tested using TH=4 or 8 for most cases. It presented an overall improvement of 18.1% over its closest competitor the *AXC* kernel and a 44.3% over the *CSR* function. There is no clear preference of this variant for a specific type of matrix because it achieved outstanding speed up factors over different type of matrices like on: the M03 matrix with an arrow-head sparsity pattern (x2.41), the M24 matrix with an irregular sparsity pattern and its abnormally large row (x7.33) and on the M25 matrix with a band sparsity pattern (x2.73). The *AXTUH* variant managed to reduce the memory requirements of the AXC format by 5.1%, but it failed to reduce them over the CSR format.

### 3.3.2. NVIDIA V100 GPU

These are the observations for the performances obtained for the NVIDIA Tesla V100 GPU:

1. The best overall performer was the *AXTUH* variant that achieved 1869.0 GFLOPS averaging 71.9 GFLOPS per matrix. It obtained the best performances on 7 out of 26 matrices.
2. The second best performer was the *AXTCH1* variant that achieved 1852.1 GFLOPS averaging 71.2 GLOPS per matrix. It won in 6 out of 26 matrices.
3. The third best performer was the *AXTCH* variant that achieved 1753.8 GFLOPS averaging 67.5 GFLOPS per matrix. This variant did not register any winning case.
4. The fourth best performer was the *CSR* function that achieved 1709.1 GFLOPS averaging 65.7 GFLOPS per matrix. The *CSR* function registered 13 out of 26 winning cases. However, most of the speed up factors achieved by this format over the *AXTUH* variant ranged between x1.02 to x1.19, which reflects a close performance between these two schemes on matrices though the *CSR* function got the upper hand.
5. The performance between the *AXTUH* and *AXTCH1* variants is very similar. However, the former achieved speed up factors over the latter from x1.02 up to x1.19, while the *AXTCH1* variant

**Table 3**
Storage and parameter information for all format variants on the NVIDIA Tesla V100 GPU. The memory footprint is indicated by the numbers without brackets in MBytes. The storage occupancy $\beta$ Eq. (2) is indicated by the numbers enclosed in square brackets. The storage occupancy is 1.00 for the variants *CSR, AXTCH1* and *AXTCH*. The tile's half width (THW) is indicated by the numbers enclosed in curly brackets. The tile's half width is 32 for the variants *AXTUH, AXTCH1* and *AXTCH*. The brick's half size (HBRS) is 32 for the *AXC* variant. The tile's height (TH) is indicated by the numbers enclosed in parentheses. The tile's height is 1 for the variants *AXTUH1* and *AXTCH1*. The block's size (BS) for variant *AXTCH1* is indicated by the numbers enclosed in angle brackets.

| *MATRIX* | *CSR* | *AXC* | *AXTUH1* | *AXTUH* | *AXTCH1* | *AXCTCH* |
|---|---|---|---|---|---|---|
| M00 | 2.1 | 6.2 [0.42] | 3.6 [0.72] {16} | 3.2 [0.84] (6) | 3.2 < 256> | 3.2 (4) |
| M01 | 13.6 | 88.6 [0.17] | 25.4 [0.62] { 8} | 19.6 [0.83] (4) | 19.2 < 512> | 19.2 (4) |
| M02 | 17.8 | 106.8 [0.19] | 32.9 [0.64] { 8} | 26.0 [0.83] (4) | 25.5 < 512> | 25.5 (4) |
| M03 | 20.8 | 62.6 [0.42] | 37.2 [0.71] {16} | 30.6 [0.90] (4) | 32.3 < 512> | 32.3 (4) |
| M04 | 31.5 | 271.3 [0.12] | 35.8 [1.00] { 4} | 35.8 [1.00] (4) | 42.0 < 512> | 42.0 (4) |
| M05 | 28.5 | 47.1 [0.79] | 42.7 [0.89] {16} | 40.7 [0.94] (8) | 46.6 <1024> | 46.6 (4) |
| M06 | 33.0 | 61.2 [0.69] | 46.9 [0.92] { 8} | 46.9 [0.92] (8) | 52.5 < 512> | 52.5 (4) |
| M07 | 49.3 | 525.4 [0.10] | 84.3 [0.63] { 4} | 84.3 [0.63] (4) | 62.1 < 256> | 62.1 (4) |
| M08 | 44.5 | 88.7 [0.65] | 67.2 [0.88] { 8} | 64.4 [0.94] (4) | 71.4 <1024> | 71.4 (4) |
| M09 | 50.4 | 144.1 [0.44] | 80.6 [0.80] { 8} | 73.9 [0.90] (4) | 78.5 <1024> | 78.5 (4) |
| M10 | 48.8 | 83.0 [0.77] | 73.1 [0.89] {16} | 68.8 [0.95] (11) | 80.2 < 128> | 80.2 (4) |
| M11 | 52.6 | 78.7 [0.88] | 75.1 [0.94] {16} | 73.7 [0.96] (12) | 86.9 < 128> | 86.9 (4) |
| M12 | 83.5 | 746.8 [0.12] | 191.1 [0.48] { 8} | 129.4 [0.72] (5) | 110.3 < 256> | 110.3 (4) |
| M13 | 73.1 | 118.8 [0.81] | 106.9 [0.93] { 8} | 99.5 [0.99] (9) | 120.2 <1024> | 120.2 (4) |
| M14 | 111.0 | 818.1 [0.15] | 209.3 [0.60] { 8} | 145.9 [0.88] (5) | 153.2 <1024> | 153.2 (4) |
| M15 | 117.7 | 633.7 [0.22] | 166.5 [0.85] { 8} | 166.5 [0.85] (8) | 171.6 <1024> | 171.6 (4) |
| M16 | 117.6 | 327.9 [0.45] | 185.1 [0.82] { 8} | 172.0 [0.91] (4) | 183.2 <1024> | 183.2 (4) |
| M17 | 140.9 | 222.8 [0.83] | 198.0 [0.96] { 8} | 190.9 [0.99] (9) | 230.5 < 128> | 230.5 (4) |
| M18 | 178.0 | 1077.3 [0.19] | 351.6 [0.60] { 8} | 266.9 [0.81] (4) | 255.4 <1024> | 255.4 (4) |
| M19 | 192.6 | 1397.1 [0.15] | 358.6 [0.61] { 8} | 290.2 [0.78] (4) | 266.9 <1024> | 266.9 (4) |
| M20 | 219.6 | 677.1 [0.40] | 351.4 [0.79] { 8} | 318.3 [0.90] (4) | 338.3 <1024> | 338.3 (4) |
| M21 | 355.3 | 1627.1 [0.26] | 872.9 [0.50] {16} | 658.3 [0.66] (11) | 532.4 <1024> | 532.4 (16) |
| M22 | 350.9 | 966.7 [0.46] | 547.0 [0.83] { 8} | 512.9 [0.91] (4) | 547.4 <1024> | 547.4 (4) |
| M23 | 704.6 | 4328.5 [0.19] | 1150.4 [0.72] { 8} | 1095.7 [0.78] (4) | 1006.6 < 512> | 1006.6 (4) |
| M24 | 781.0 | 3384.8 [0.28] | 1971.4 [0.49] {16} | 1574.8 [0.62] (12) | 1190.5 < 512> | 1190.5 (16) |
| M25 | 786.3 | 1415.0 [0.73] | 1126.0 [0.97] { 8} | 1104.4 [0.97] (6) | 1282.6 <1024> | 1282.6 (4) |

achieved speed up factors from x1.01 up to x2.23 over the *AXTUH* variant.

6. The performance improvement of the *AXTCH1* variant on the NVIDIA GPU can be explained because the kernel was less penalised by lower latency accesses to memory due to the fact that variables and dynamic arrays required by the fast segmented sum algorithm were stored in the processors registers and on-chip memory.
7. The *AXTUH* variant over the CSR format represents an overall performance improvement of 8.6% and an increment on memory requirements of 36.9%.
8. The *AXTCH1* variant over the CSR format represents an overall performance improvement of 7.7% and an increment in memory requirements of 34.1%.
9. The *AXTUH* variant reached notable speed up factors over the CSR format for the matrices: M00 (x1.89), M01 (x2.47), M07 (x5.00), M20 (x2.73), M21 (x169.50) and M24 (x66.71).
10. The *AXTCH1* variant reached notable speed up factors over the CSR format for the matrices: M00 (x1.82), M01 (x2.49), M07 (x6.78), M20 (x2.68), M21 (x378.50), and M24 (x112.43).
11. The *CSR* function exceeded 100 GFLOPS for the matrices: M11 (101.4 GFLOPS), M13 (108.5 GFLOPS), M17 (100.9 GFLOPS) and M25 (108.1 GFLOPS). The M11, M17 and M25 matrices exhibit a band sparsity pattern while the M13 matrix has a multi-band sparsity pattern.
12. The *AXTUH* variant achieved its best performance mostly with TH=4 or 8 and the *AXTCH1* variant achieved its best performance using BS=512 or 1024.

In summary, the *AXTUH* and *AXTCH1* variants achieved the best overall performance on the NVIDIA Tesla V100 GPU using TH=4 or 8 and BS=512 or 1024 respectively for most cases. However, their overall performance improvement is lower than 10% while their memory requirements represent an increment of approximately 35% compared to the CSR format. Despite a 10% of performance improvement may not justify the selection of the AXT variants over the CSR format for most matrices due to their larger memory requirement, these variants may be found useful for matrices possessing abnormally large rows within their ranks (e.g. M07, M20, M21 and M24) where these variants reach speed up factors from x2.68 up to x378.50. Another fact worth to highlight is that the AXT variants managed to overcome the AXC format's performance on the NVIDIA GPU, successfully accomplishing one of the objectives of this work.

## 4. Conclusions

This paper proposes a new sparse matrix storage scheme designated as AXT that is compared with the existing AXC and the CSR formats, and tested on two different multi/many-core platforms: the Intel Xeon Gold

6148 processor using the Intel AVX-512 instructions set and the NVIDIA Tesla V100 GPU using warp primitives from CUDA. In order to adapt the new AXT format to these platforms and to the matrix being targeted, three parameters are used: the tile's half width (THW), the mode (MODE), and the tile's height (TH). Depending on the value of the MODE and TH parameters the AXT format can spawn one of four possible data arrangement variants tagged as: *AXTUH1, AXTUH, AXTCH1* and *AXTCH*. The computational kernels needed to do the SpMV product required the adoption of novel and complex algorithms developed to perform reduction and scan operations on parallel devices resulting in eight different implementations.

The numerical results on the Intel Xeon Gold 6148 processor (Fig. 5) have shown that the *AXTUH* variant was the best performer using TH=4 or 8 for most cases. *AXTUH* variant achieved a performance improvement of 18.1% and it managed to reduce the memory footprint a 5.1% over the AXC format. Compared to the CSR format, the *AXTUH* variant achieved a performance improvement of 44.3% and needed 34.6% more memory. This variant did not show any preference for a specific sparsity pattern since it reached speed up factors for different types of matrices: x2.41 for the M03 matrix with an arrow-head sparsity pattern, x7.33 for the M24 matrix with an irregular sparsity pattern and its abnormally large row and x2.73 for the M25 matrix with a band sparsity pattern.

On the other hand, the results on the NVIDIA Tesla V100 GPU (Fig. 6) have shown that the *AXTUH* and *AXTCH1* variants were the best overall performers using TH=4 or 8 and BS=512 or 1024 respectively for most matrices. These variants achieved approximately the same overall performance, 1869.0 GFLOPS for the *AXTUH* variant and 1852.1 GFLOPS for the *AXTCH1* variant, which represent an improvement of 8% over the CSR format and a 44% over the AXC format. The improvement over the AXC format proves that the AXT has successfully extended the range of application of its predecessor to the NVIDIA platform, despite the 35% larger memory requirements than the CSR format to store all the matrices. The *AXTUH* and *AXTCH1* variants excelled processing matrices with irregular sparsity patterns with abnormally large rows within their ranks like the M07, M20, M21 and M24 matrices where they reached speed up factors from x2.68 up to x378.5 over the CSR format.

The *AXTUH* good performance can be explained because it stores the elements of a matrix's rows in major column order. By doing so, this variant guarantees coalesced memory access from SIMD lanes and enforces a one-step accumulation by a simple sum operation of the partial result of THW different segments, this exploits the vector capabilities of the platforms. On the other hand, the major handicap of the *AXTUH1* is that it requires an explicit reduction stage that reduces its performance. Also, this variant cannot process as many different segments in parallel as the *AXTUH* variant because each tile stores the elements of one row. The *AXTCH* poor performance is mainly due to the branch divergence introduced by the conditional instruction used to locate the beginning and end of a row within a segment (column). Despite the *AXTCH1* variant shares the same handicap of the *AXTCH* scheme, it excels at storing matrices with extremely large rows, because it splits the large rows into segments that are processed in parallel improving the workload balance among threads. This is especially observed on the NVIDIA platform whose implementation of a kernel for a block size of 1024 requires less additional memory than an Intel kernel using a block size of 512.

Lastly, this work has tested several algorithms presented in Section 2 to do parallel reduction and scan operations (e.g. naive reduction, work-efficient scan, fast segmented scan) required to perform the SpMV product and it has evidenced their weaknesses and strengths on two different platforms. Evidently, the fast segmented sum algorithm employed for the compacted variants of the AXT format has delivered a poor performance on the Intel platform while it generated faster execution times on the NVIDIA platform. Thus, this fact provides an area for future work, that is, to research for alternative algorithms that improve the compacted variants performance on the Intel platform in order to reduce the memory footprint of the AXT format.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References


[1] Wong J, Kuhl E, Darve E. A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems. International Journal for Numerical Methods in Engineering 2015;102(12):1784–814. https://doi.org/10.1002/nme.4865.

[2] Ahamed A-KC, Magoulès F. Conjugate gradient method with graphics processing unit acceleration: Cuda vs opencl. Advances in Engineering Software 2017;111:32–42. https://doi.org/10.1016/j.advengsoft.2016.10.002.http://www.sciencedirect.com/science/article/pii/S096599781630477X

[3] Kanamori I, Matsufuru H. Practical Implementation of Lattice QCD Simulation on SIMD Machines with Intel AVX-512. Lecture Notes in Computer Science 2018:456–71. https://doi.org/10.1007/978-3-319-95168-3_31.

[4] Coronado-Barrientos E, Indalecio G, Seoane N, García-Loureiro A. Implementation of numerical methods for nanoscaled semiconductor device simulation using OpenCL. Proceedings of the 2015 Spanish Conference on Electron Devices. IEEE; 2015. p. 1–4. https://doi.org/10.1109/CDE.2015.7087476.http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=7087476&queryText=indalecio&newsearch=true&searchField=Search_All

[5] Bell N, Garland M. Efficient Sparse Matrix-Vector Multiplication on CUDA. Tech. Rep. Santa Clara, CA: NVIDIA; 2008.

[6] Bell N, Garland M. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. New York, NY, USA: ACM; 2009, ISBN 978-1-60558-744-8. p. 1–11. https://doi.org/10.1145/1654059.1654078.

[7] Kreutzer M, Hager G, Wellein G, Fehske H, Basermann A, Bishop AR. Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation. Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. Washington, DC, USA: IEEE Computer Society; 2012, ISBN 978-0-7695-4676-6. p. 1696–702. https://doi.org/10.1109/IPDPSW.2012.211.

[8] Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. second. Philadelphia, PA: SIAM; 1994.

[9] Catalán S, Martorell X, Labarta J, Usui T, Toledo Díaz LA, Valero-Lara P. Accelerating conjugate gradient using ompss. 2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). 2019. p. 121–6.

[10] Ortega G, Garzón EM, Vázquez F, García I. The BiConjugate gradient on GPUs. The Journal of Supercomputing 2012;64:49–58.URL https://doi.org/10.1007/s11227-012-0761-2

[11] Coronado-Barrientos E, Indalecio G, García-Loureiro AJ. Improving Performance of Iterative Solvers with the AXC Format Using the Intel Xeon Phi. J Supercomput 2018;74(6):2823–40. https://doi.org/10.1007/s11227-018-2325-6.

[12] Grimes R, Kincaid D, Young DM. Itpack 2.0 User's Guide. Center for Numerical Analysis, The University of Texas at Austin; 1979.https://books.google.es/books?id=EseZwgEACAAJ

[13] NVIDIA Developer Zone. CUDA Toolkit documentation. https://docs.nvidia.com/cuda/index.html; Accessed: 2020-02-06.

[14] Intel. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#; Accessed: 2020-02-06.

[15] Jeffers J, Reinders J, Sodani A. Intel Xeon Phi Processor High Performance Programming. Knights Landing Edition. Morgan Kaufmann; 2016, ISBN 978-0-12-809194-4.

[16] Sanders J, Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. first. Addison-Wesley; 2011, ISBN 978-0-13-138768-3.

[17] Vázquez F, Garzón EM, Martinez A, Fernández JJ. The sparse matrix vector product on GPUs. Proceedings of the 2009 International Conference on Computational and Mathematical Methods in Science and Engineering 2009;2:1081–92.

[18] Vázquez F, Ortega G, Fernández JJ, Garzón EM. Improving the performance of the sparse matrix vector with GPUs. Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology. Washington, DC, USA: IEEE Computer Society; 2010, ISBN 978-0-7695-4108-2. p. 1146–51. https://doi.org/10.1109/CIT.2010.208.

[19] Monakov A, Lokhmotov A, Avetisyan A. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. High Performance Embedded Architectures and Compilers: 5th International Conference. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010, ISBN 978-3-642-11515-8. p. 111–25. https://doi.org/10.1007/978-3-642-11515-8_10.

[20] Choi JW, Singh A, Vuduc R. Model-driven autotuning of sparse matrix-vector multiply on GPUs.. ACM Sigplan Notices. 2010. p. 115–26.

[21] Hossain S. On efficient data structures for sparse matrix storage. 2006.URL http://www.mathematik.hu-berlin.de/~gaggle/EVENTS/2006/BRENT60/presentations/Shahadat%20Hossain%20-%20On%20efficient%20data%20structures%20for%20sparse%20matrix%20storage.pdf

[22] Hager G, Wellein G. Introduction to High Performance Computing for Scientifics and Engineers. CRC Press; 2011, ISBN 978-1-4398-1192-4. https://doi.org/10.1201/EBK1439811924-f.
[23] Kreutzer M, Hager G, Wellein G, Fehske H, Bishop AR. A unified sparse matrix data format for modern processors with wide SIMD units. SIAM Journal on Scientific Computing 2014;36:C401–23.
[24] Altinkaynak A. An efficient sparse matrix-vector multiplication on CUDA-enabled graphic processing units for finite element method simulations. International Journal for Numerical Methods in Engineering 2017;110(1):57–78. https://doi.org/10.1002/nme.5346.
[25] Liu W, Vinter B. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. 2015.
[26] Coronado-Barrientos E, Fernández GI, García-Loureiro AJ. AXC: A new format to perform the SpMV oriented to Intel Xeon Phi architecture in OpenCL. Concurrency and Computation: Practice and Experience 2019;31. https://doi.org/10.1002/cpe.4864.
[27] Catalán S, Usui T, Toledo L, Martorell X, Labarta J, Valero-Lara P. Towards an Auto-Tuned and Task-Based SpMV (LASs Library). OpenMP: Portable Multi-Level Parallelism on Modern Systems. Cham: Springer International Publishing; 2020, ISBN 978-3-030-58144-2. p. 115–29.
[28] Lin Y., Grover V.. Using CUDA Warp-Level Primitives. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/; Accessed: 2020-02-06.
[29] S. Sengupta and M. Harris and M. Garland. Efficient Parallel Scan Algorithms for GPUs. Tech. Rep. NVIDIA Corporation; 2008.
[30] Sengupta S, Harris M, Owens JD. Parallel Prefix Sum (Scan) with CUDA. In: Nguyen H, editor. GPU Gems 3. Addison Wesley Professional; 2007. p. 39–69.
[31] McCool M, Robison AD, Reinders J. Structured Parallel Programming Patterns for Efficient Computation. Morgan Kaufmann; 2012, ISBN 978-0-12-415993-8.
[32] Edinburgh Parallel Computing Centre (EPCC). Cirrus. http://www.cirrus.ac.uk/; Accessed: 2020-03-02.
[33] Intel. Intel Xeon Gold 6148 Processor. https://ark.intel.com/content/www/us/en/ark/products/120489/intel-xeon-gold-6148-processor-27-5m-cache-2-40-ghz.html; Accessed: 2020-03-02.
[34] WikiChip. Intel Xeon Gold 6148. https://en.wikichip.org/wiki/intel/xeon_gold/6148#Memory_controller.
[35] Reinders J, Jeffers J. High Performance Parallelism Pearls Multicore and Many-core Programming Approaches. ElSci; 2015, ISBN 978-0-12-802118-7.
[36] NVIDIA. NVIDIA Tesla. V100 GPU. 2020.https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf; Accessed
[37] Texas A&M University. SuiteSparse Matrix Collection. 2020.https://sparse.tamu.edu/