Sequences of Sparse Matrix-Vector Multiplication on Fugaku's A64FX processors

Jérôme Gurhem, Maxence Vandromme, Miwako Tsuji, Serge Petiton, Mitsuhisa Sato

▶ To cite this version:

Jérôme Gurhem, Maxence Vandromme, Miwako Tsuji, Serge Petiton, Mitsuhisa Sato. Sequences of Sparse Matrix-Vector Multiplication on Fugaku's A64FX processors. CLUSTER 2021 - IEEE International Conference on Cluster Computing, Sep 2021, Portland, United States. pp.751-758, 10.1109/Cluster48925.2021.00111 . hal-03450283

HAL Id: hal-03450283
https://hal.archives-ouvertes.fr/hal-03450283

Submitted on 25 Nov 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Sequences of Sparse Matrix-Vector Multiplication on Fugaku's A64FX processors

Jérôme Gurhem*, Maxence Vandromme*, Miwako Tsuji†, Serge G. Petiton*‡, Mitsuhisa Sato†
*USR 3441 - Maison de la Simulation, CNRS
Saclay, France
†RIKEN Center for Computational Science
Kobe, Japan
‡Univ. Lille, UMR 9189 CRIStAL, CNRS
F-59000 Lille, France

Abstract—We implement parallel and distributed versions of the sparse matrix-vector product and the sequence of matrix-vector product operations, using OpenMP, MPI, and the ARM SVE intrinsic functions, for different matrix storage formats. We investigate the efficiency of these implementations on one and two A64FX processors, using a variety of sparse matrices as input. The matrices have different properties in size, sparsity and regularity. We observe that a parallel and distributed implementation shows good scaling on two nodes for cases where the matrix is close to a diagonal matrix, but the performances degrade quickly with variations to the sparsity or regularity of the input.

Index Terms—Parallel computing, distributed computing, matrix-vector multiplication, sparse matrix, A64FX, network on chip.

I. Introduction

Sequences of sparse matrix multiplication are a key part of many numerical applications, notably iterative methods such as the conjugate gradient [1]. In addition, with the increasing use of sparse matrices in Deep Learning [2], the training of deep neural networks relies more and more on sparse matrix-vector multiplication. Therefore, implementing an efficient sparse matrix-vector product (SpMV) is necessary to ensure reliable performance for applications using it.

In this article, we focus on the Fujitsu A64FX processor that powers the Fugaku supercomputer [3], which is #1 in the TOP500 list at the time of the study. This processor contains 48 compute cores, split into four groups of 12. Each such group is called Core Memory Group (CMG), and has its own L2 cache and memory. The CMGs inside a processor are linked by a network-on-chip that handles the communications between them. The processors are linked by a 6D topology Tofu interconnect. For large-scale computations, we need to use parallel and distributed applications. In these cases, the communications often become a limiting factor for the performance, and therefore require thorough investigation and optimization. Single node sparse matrix-vector product with OpenMP and global addressing on the network-on-chip has already been considered in a recent study [4]. Therefore, we focus here on multi-node implementations with MPI and MPI+OpenMP. We also evaluate the efficiency of using MPI on a single node to perform communications between the CMGs of the A64FX chip instead of using global memory addressing. We consider several standard storage formats for sparse matrices such as compressed sparse rows (CSR), Ellpack-itpack (ELL), and coordinates (COO). These formats are not always ideal for SpMV computation, but they are largely used in widespread packages and applications, and as such they deserve to be studied as a baseline to better understand the performance patterns and bottlenecks.

This paper is organized as follows: in Section II, we introduce the block decomposition used to distribute the sparse matrices on the different MPI processes, and the algorithms used to perform the sparse matrix-vector product with several sparse storage formats. In Section III, we describe our distributed application (sequence of SpMV) and the test matrices used to compare the performance of the different implementations and storage formats. Then, in Section IV, we detail the experiments and the results obtained. Finally, in Section V, we give a summary and an overview for future work.

II. Data Distribution and Sparse Storage Formats

In this section, we introduce the matrix distribution and the sparse storage formats used in our applications. They are used to store matrices distributed across multiple compute nodes, which induces communication issues depending on the distribution of the matrix. The SpMV algorithms were introduced previously and are available in more detail, for instance, in [5]. We choose to implement the compressed sparse row (CSR) and two versions of the coordinates (COO) storage formats, which are used, for instance, in TensorFlow, Pytorch, and cuSPARSE from Nvidia [6]. We also implemented the Ellpack-itpack (ELL) storage format since it is shown that it has been shown to obtain good performance on GPUs [7].

A. 2D block decomposition

In this section, we consider different ways of distributing sparse matrices across several nodes. We use a 2D block

decomposition of the matrix. Each sub-matrix is a sparse matrix compressed in memory with a sparse storage format. We are not looking into the load balancing issues that may arise if some blocks are more populated than others. This is a different complex issue which is not the focus of this study.

The block distribution consists of both a distribution by rows and a distribution by columns. The $Nc \times Nr$ matrix is split in $Ngc \times Ngr$ sub-matrices. The sub-matrices are stored in a sparse storage format locally. In this case, the input vector is split across the columns of the matrix and the sub-vectors are duplicated on the sub-rows of the same column. The resulting vector has the same size as the number of rows in the sub-matrices of the corresponding row. However, each computing resource contains a part of a sub-vector. To obtain the global result, all the distributed results of the same row have to be summed then each row has to be gathered if the full result vector is needed in one place.

B. COO

Algorithm 1: COO format data structure and matrix vector product

```
struct {
  | Vector row, col, val
} MatrixCOO

Function spmv_coo()
  | Data: m : MatrixCOO, v : Vector
  | Result: r : Vector
  | For i from 0 to m.val.size() - 1 do
  |   r[m.row[i]] += m.val[i] * v[m.col[i]]
```

The COO format stores three vectors of equal size: row index, column index, and value. Algorithm 1 shows the formula for the matrix-vector product using this format. In this implementation, the sub-matrix does not have to be sorted, i.e. this implementation does not expect the coordinates of the values of the matrix to be in a given range. Therefore, this implementation can process any value in any position in any sub-matrix (which is not the case with the other storage formats). However, we cannot deduce the range of position in the full input vector as well as the range of position for the output vector since there are no restrictions on the range of coordinates of the values in the matrix.

It can allow a better load balancing at the cost of a full vector in input and output, which will change how the output vector will be processed to construct the full output vector.

C. SCOO

The SCOO format uses the same three vectors as COO, and two additional numbers/integers to locate the sub-matrix within the overall matrix. Algorithm 2

Algorithm 2: SCOO format data structure and matrix vector product

```
struct {
  | Vector row, col, val
  | Integer fr, fc
} MatrixSCOO

Function spmv_scoo()
  | Data: m : MatrixCOO, v : Vector
  | Result: r : Vector
  | For i from 0 to m.val.size() - 1 do
  |   r[m.row[i] - m.fr] += m.val[i] * v[m.col[i] - m.fc]
```

describes the sparse matrix vector product for SCOO. These informations are the first row (fr) and the first column (fc) in the sub-matrix. The algorithm for the SCOO storage format is similar to the algorithm for COO; the supplementary information is used in the algorithm to properly position the output vector. In this case, the range of position for the input and output vectors can be computed since the range of coordinates for the values in the SCOO storage format are restricted to the data distributions discussed in Section II-A. Therefore, depending on the data distribution, the input and output vector shapes change.

In this case, the construction of the output vector will depend on the division of the matrix as specified in Section II-A.

D. CSR

Algorithm 3: CSR format data structure and matrix vector product

```
struct {
  | Vector idx, col, val
  | Integer fc
} MatrixCSR

Function spmv_csr()
  | Data: m : MatrixCSR, v : Vector
  | Result: r : Vector
  | For i from 0 to m.idx.size() - 1 do
  |   | For j from m.idx[i] to m.idx[i+1] - 1 do
  |   |   r[i] += m.val[j] * v[m.col[j] - m.fc]
```

The CSR format also uses three vectors to store the matrix. Here, the vector containing the indexes of the row is replaced by a vector that contains the position of the beginning of each row in the column vector. Algorithm 3 describes the sparse matrix vector product for CSR. The data structure contains the three vectors necessary to store the matrix in the CSR storage format as well as useful information about the position of the sub-matrix in the global matrix, i.e. the first column (fc) in the sub-matrix. As for the SCOO storage format, the ranges of position for the input and output vectors can be computed.

E. ELL

Algorithm 4: ELL format data structure and matrix vector product

```
struct {
    Vector col, val
    Integer fc, max_col
} MatrixELL

Function spmv_ell()
    Data: m : MatrixELL, v : Vector
    Result: r : Vector
    For i from 0 to m.lrs - 1 do
        For j from 0 to m.max_col - 1 do
            r[i + m.rpos] += m.val[i * m.max_col + j]
              * v[m.col[i * m.max_col + j]- m.fc]
```

The Ellpack format uses two rectangular matrices. The first vector stores the position of the values in the columns and the second stores the corresponding values. Algorithm 4 describes the sparse matrix vector product for the Ellpack sparse storage formats. The data structure contains the number of columns needed in the vectors and the two vectors necessary to store the matrix in the ELL storage format as well the first column (fc) in the sub-matrix which is used in this algorithm. As for the SCOO and CSR storage formats, the ranges of position for the input and output vectors can be computed.

III. $A(Ax + x) + x$ Application

In this section, we describe out target application: the sequence of SpMV with vector addition $A(Ax+x)+x$. We also describe the different sparse matrices that are used in the experiments.

A. Distributed $A(Ax + x) + x$

The sparse operation $A(Ax + x) + x$ consists of a call to the sparse matrix-vector product followed by the sum of the output vector with x, then a second iteration of these processes. This application is more representative of the iterative methods using the SpMV as kernel, since the result vector has to be redistributed to all processes after the first iteration. This operation $A(Ax+x)+x$ has been implemented in MPI and MPI+OpenMP on top of a sparse matrix-vector product kernel that supports the CSR, ELL, COO, and SCOO storage formats as well as the different matrix distribution possibilities. The communications, the building of the output vector depending on the distribution of the matrix, and the sum are managed with the programming models in which the application is implemented (e.g. reductions from OpenMP or MPI)[1]. The program is compiled using the Fujitsu compiler in Clang mode, with flags '-Kopenmp -fPIC -Ofast -mcpu=native -funroll-loops -fno-builtin -march=armv8.2-a+sve'.

[1] Code is available at https://github.com/jgurhem/TBSLA/tree/dev_array

The $Nc \times Nr$ input matrix is distributed over a $Ngc \times Ngr$ decomposition, as seen in Section II-A. This kernel is called in the MPI processes to perform the sparse matrix-vector product on the local matrices. $Ngc \times Ngr$ is equal to the number of available processes in the application. This allows each process to manage its sub-matrix.

B. Test Matrices

We use several sparse matrices with different properties, in order to get a broader view of the performance of our application in a variety of situations (size, sparsity, regularity...).

The first test matrix we use to evaluate performance is Matrix Market's nlpkkt120. It is a square matrix with 3,542,400 rows and 96,845,792 entries. The number of entries per row is regular (about 27), and the matrix has a strong diagonal pattern. This matrix was used in the previous study about SpMV on the A64FX processor [4].

We use two additional matrices, drawn from the SuiteSparse Matrix Collection [8]: cage14 and cage15. These two matrices are structurally similar to one another, but their sizes are different (see Table I for dimensions). Compared to nlpkkt120, they have high sparsity, and higher deviation from the diagonal for the nonzero entries. Therefore, we expect them to highlight the cost of communications since the data will be distributed across more nodes and processes.

matrix	rows	columns	nnz
cage14	1,505,785	1,505,785	27,130,349
cage15	5,154,859	5,154,859	99,199,551
nlpkkt120	3,542,400	3,542,400	96,845,792

TABLE I
Matrices informations

Finally, we use C-diagonal Q-perturbed matrices introduced in [9]. These matrices consist of C values above the diagonal including the diagonal with a probability of Q to change the column position of the value. Figure 1 represents a 30×30 C-diagonal Q-perturbed sparse matrix with C=4 and several values of Q (0.05, 0.5 and 0.9) where each black square is a non-zero value. The deviation from the diagonal increases quickly with the parameter Q.

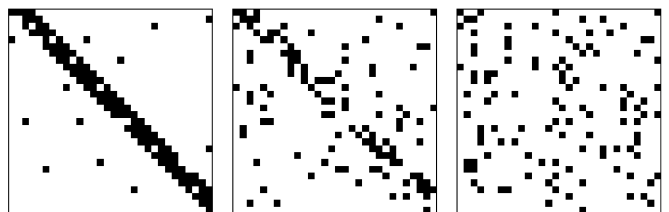

Fig. 1. C-diagonal Q-perturbed sparse matrix with C=4 and Q=0.05 on the left, Q=0.5 on the middle and Q=0.9 on the right

IV. Experiments

In this section, we experiment with our different implementations of the sparse matrix-vector product. First, we

focus on the improvements provided by using ARM SVE for the sparse matrix product. Then, we present the results of experiments with Matrix Market's nlpkkt120 matrix example for COO, CSR, ELL, and SCOO. We compare them with the results on cage14 and cage15. Finally, we use C-diagonal Q-perturbed matrices as benchmarks for the CSR and ELL implementations.

A. ARM SVE

A major feature of the recent ARM-based processors, including the A64FX, is the Scalable Vector Extension (SVE) that enables support for SIMD operations with per-lane prediction, which allows for efficient vectorization [10]. The vector length can be specified as a multiple of 128 bits. We use a vector of 512 bits, which is the default for the processor and allows for the simultaneous computation of 8 double-precision values (64 bits each). The SVE instructions can be generated automatically by the compiler for simple functions. However, this is not currently supported for the SpMV, which require indirect access to the stored arrays. Instead, we used the ARM SVE intrinsic functions to implement a vectorized version of the SpMV on different matrix formats. Then, we used the nlpkkt120 matrix to benchmark the efficiency of the vectorization.

also be that the optimizations done by the compilers on the non-vectorized code (software pipelining, etc.) already bring this code close to the best reachable performance for this application.

There are several possible ways to improve the efficiency of the vectorization. A first possibility is to fine-tune the prefetching options of the compiler for the input data. A second option is to apply manual unrolling to the inner loop, which can improve performance noticeably on some applications. Another option is to use data storage formats better suited to vectorization, such as the SELL-C-sigma [11] format used in a previous study on this processor [4], or other proposed formats [12]. We did not explore these possibilities in this study, and instead wanted to evaluate the possible gain of using intrinsic functions "off-the-shelf". As shown in the rest of this section, we focus more on the general behavior of the applications for different input matrices and different implementations. We believe this is a valuable first step before diving more in case-by-case optimization and tuning.

B. Matrix Market's nlpkkt120

For the experiments in this section, we used the nlpkkt120 matrix as a benchmark for our implementations of the sparse Ax as well as the sparse $A(Ax+x)+x$.

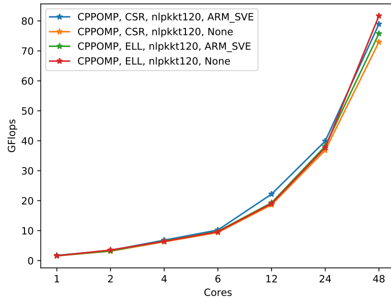

Fig. 2. Ax with ARM SVE vs no SVE for CSR and ELL with nlpkkt120 as example

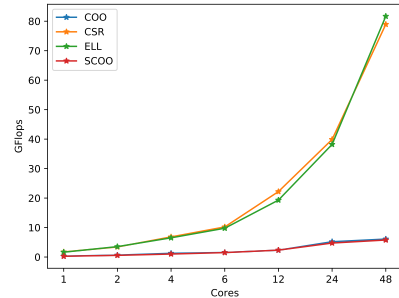

Fig. 3. Ax for COO, CSR, ELL and SCOO with nlpkkt120 as example

Figure 2 shows the results (in GFlop/s) obtained with the two implementations of ELL and CSR, on the sparse Ax operation. We can see that ELL without vectorization runs slightly faster than CSR for this matrix. We can also see that CSR with vectorization runs slightly faster than CSR without. In CSR's case, the addition of vectorization improves the performance of the sparse matrix-vector product by about 8% whereas, for ELL, the vectorized code does not always improve the performance. Therefore, we choose to use vectorization for CSR in the experiments presented in the rest of this section. For ELL, we choose to run both implementations and show the results of the implementation that provides the best results.

The lack of significant speedup when using a vectorized implementation could mean that the latency associated with the SVE intrinsics (see analysis in [4]) compensates the benefits of computing several values at once. It could

Figure 3 shows the performance obtained with each storage format for the sparse Ax with the OpenMP implementation. We can see that the results for CSR and ELL are similar since their implementation with the vectorization is based on a similar approach. However, COO and SCOO are not vectorized and they need a sum on the output vectors of each thread to compute the resulting vector. Because of the COO/SCOO format, the time spent to compute the sum increases with the number of cores (= OpenMP threads) used, leading to minimal scaling.

Table II shows the number of floating-point operations per second for the MPI and the MPI+OpenMP implementations, for the sparse matrix-vector product on 1 and 2 nodes. The last row of the table represents the pure MPI implementation whereas the other rows show the results for the MPI+OpenMP implementation. The thread

threads	COO		CSR		ELL		SCOO	
	1	2	1	2	1	2	1	2
1	7.9	10.0	75.6	127.2	81.7	145.5	6.0	11.9
2	3.8	4.4	75.0	136.4	72.4	135.0	6.0	11.9
4	5.4	4.0	73.6	142.2	80.3	135.0	5.9	11.7
6	5.9	2.6	73.8	130.0	64.2	131.7	5.8	11.4
12	3.1	3.1	72.5	131.5	69.9	129.4	5.5	11.0
24	3.0	2.4	71.9	130.9	69.1	124.5	3.2	5.4
48	1.6	2.6	69.8	127.9	68.8	123.2	2.4	3.3
MPI	14.5	27.4	78.1	119.1	84.4	121.2	6.8	13.0

TABLE II
MPI and MPI+OpenMP performance (in GFlop/s) for Ax with nlpkkt120 matrix on 1 and 2 nodes for COO, CSR, ELL and SCOO storage formats with different number of threads per MPI process, keeping 48 threads total on each node

column gives the number of threads per MPI process used during the execution while keeping all the cores busy (e.g. when there are 2 threads per process, there are 24 MPI processes).

We can see that COO does not scale well with MPI+OpenMP, with the performance on 2 nodes being sometimes lower than on 1 node. It is mainly due to the OpenMP reduction on the resulting vector after the execution of the OpenMP threads. However, the MPI implementation scales well, even if the performance is behind CSR and ELL, since there are more memory accesses due to the format design. For CSR, ELL, and SCOO, the best performance is obtained with pure MPI for one node. For CSR, they are close to the ones obtained with OpenMP only (78.9 GFlop/s for OpenMP and 78.1 for MPI). However, for ELL, the performance is better with pure MPI than OpenMP, whereas it is the opposite for SCOO where the performance is better with OpenMP. CSR and ELL's best performance on two nodes is obtained with MPI+OpenMP with 12 MPI processes and 4 OpenMP threads per process, whereas for SCOO it is with pure MPI.

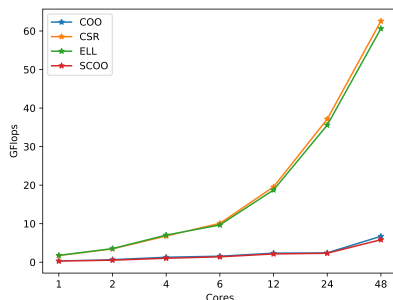

Fig. 4. $A(Ax+x)+x$ for COO, CSR, ELL and SCOO with nlpkkt120 as example

We then ran experiments for the sparse operation $A(Ax+x)+x$, which should better approximate the iterative methods. Figure 4 shows its performance with the OpenMP implementation up to 48 threads. We obtain similar performance patterns as for Ax, which is expected since it is based on this operation. However, the performance is lower than for Ax since there are two more additions between the resulting vector of Ax and the input vector.

threads	COO		CSR		ELL		SCOO	
	1	2	1	2	1	2	1	2
1	2.6	1.1	8.9	10.4	8.3	10.6	3.8	5.9
2	2.8	1.7	8.2	9.2	8.1	9.1	3.7	5.9
4	3.2	3.1	13.1	15.0	13.9	14.8	4.3	7.2
6	2.3	1.9	16.7	17.6	16.0	17.9	4.6	7.5
12	2.1	1.9	18.1	20.8	17.2	22.5	4.6	8.0
24	3.6	1.8	27.5	25.9	29.6	25.1	3.6	5.6
48	1.6	3.4	34.1	35.8	33.4	36.1	3.3	4.3
MPI	3.2	1.2	9.0	9.9	8.1	9.9	3.6	5.6

TABLE III
MPI and MPI+OpenMP performance (in GFlop/s) for $A(Ax+x)+x$ with nlpkkt120 matrix on 1 and 2 nodes for COO, CSR, ELL and SCOO storage formats with different number of threads per MPI process, keeping 48 threads on each node

With distributed sparse matrices, the sparse matrix-vector product resulting vector is local, thus, it has to be combined with the outputs in the other processes to form the complete solution then perform the addition. The creation of the full output vector is costly, as we can see in Table III, in which the GFlop/s of the sparse $A(Ax+x)+x$ are given depending on the number of OpenMP threads used to run the application. Indeed, the best performance obtained with MPI or MPI+OpenMP is about half the one with just OpenMP. Moreover, the scaling of the application is poor since the improvement from using 2 nodes is minimal at best. This is due to the construction of the resulting vector to perform the addition, since it is the only difference with the regular Ax.

C. cage14 and cage15 matrices

As discussed in Section III, the two cage matrices have a different pattern from nlpkkt120. The matrix is more sparse, and the entries are more spread out from the diagonal, which means the data will be distributed differently.

Tables IV and V show the performance on these two matrices compared to nlpkkt120, for OpenMP, MPI, and OpenMP+MPI implementations, for Ax and $A(Ax+x)+x$ respectively. We only consider CSR and ELL as storage formats, since the other two have shown poor scaling and performance in the previous experiments.

matrixtype	CPPOMP		MPI		MPIOMP	
	CSR	ELL	CSR	ELL	CSR	ELL
cage14	29.5	23.1	31.7	24.7	32.4	25.2
cage15	28.8	22.2	31.0	24.5	33.2	26.2
nlpkkt120	78.9	81.7	78.1	84.4	75.6	81.7

TABLE IV
Performance (in GFlop/s) for several Matrix Market matrices for Ax on 1 node

We can make two observations about these results. First, the performance is lower than on nlpkkt120, at least by a factor 2. The pure MPI implementation is also

matrixtype	CPPOMP		MPI		MPIOMP	
	CSR	ELL	CSR	ELL	CSR	ELL
cage14	27.9	22.0	5.1	4.2	15.3	14.0
cage15	26.0	20.5	5.8	5.9	17.1	14.7
nlpkkt120	62.6	60.6	9.0	8.1	34.1	33.4

TABLE V
Performance (in GFlop/s) for several Matrix Market matrices for $A(Ax + x) + x$ on 1 node

never the best option for these two matrices, compared to nlpkkt120. As expected, this is probably due to the increased burden on communications caused by the data distribution. Second, ELL performs noticeably worse than CSR, whereas for nlpkkt120 the two formats give similar results or ELL is slightly better. This may be due to the number of nonzero entries being different across the rows of the matrix This would lead to the storage of useless zeroes, since ELL aligns on the highest number of nonzeroes per row.

In order to get a clearer view of the effect of having nonzero entries deviating from the diagonal on the performance of our application, we now present experiments on synthetic C-diagonal Q-perturbed matrices.

D. C-diagonal Q-perturbed matrices

1) C = 100: We used $8,000,000 \times 8,000,000$ matrices with C = 100 non-zero elements per row, with different values of Q which represents the dispersion of the values in the rows of the matrix. When Q is small the nonzero elements of the matrix are concentrated close to the diagonal of the matrix. When Q increases, the probability of exchanging the position in the rows with another one increases. Therefore, the values are more dispersed when Q increases. Thus, when Q is small, the values in the rows are close to each other. The values of the input vector, accessed during the dot product between the row of the matrix and the input vector, are also close in memory. However, when Q increases, this is not the case anymore and the accesses to the memory containing the input vector are more random and more prone to cache misses. Therefore, when Q increases, the performance of the sparse matrix-vector decreases. This can be confirmed by Figure 5 in which we represent the performance of Ax depending on the values of Q for CSR and ELL.

Indeed, Figure 5 shows that the performances of CSR and ELL are similar and that they depend on the values of Q. When Q is small, the performance is very high with up to 70 GFlop/s. However, when Q increases, the performance of the sparse matrix-vector product degrades quickly.

Tables VI and VII show the performances of CSR and ELL for the sparse Ax with the MPI and MPI+OpenMP implementations. We can see that the MPI and MPI+OpenMP implementations are performing better than the OpenMP implementation for CSR and ELL. MPI+OpenMP with 1 MPI process and 48 OpenMP threads per process obtains the same performance as pure

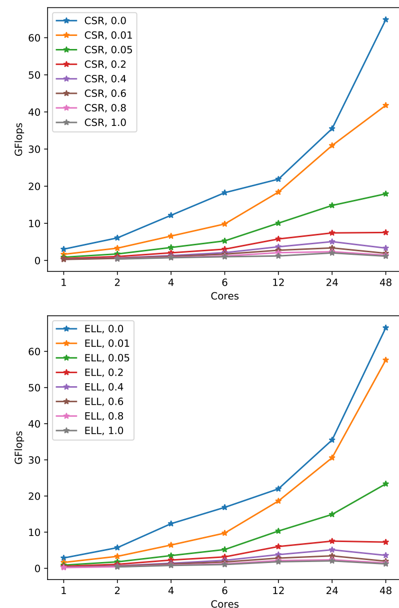

Fig. 5. Ax with OpenMP for C-diagonal Q-perturbed matrices with C = 100

threads	0.0		0.4		0.8	
	1	2	1	2	1	2
1	138.2	275.5	12.0	24.0	7.2	14.4
2	139.0	272.9	12.3	24.0	7.3	14.4
4	134.3	270.7	12.1	24.6	8.2	16.2
6	136.7	264.7	12.2	24.7	8.7	15.1
12	136.6	260.0	13.0	25.6	10.7	20.0
24	130.3	248.4	9.0	22.0	8.8	20.4
48	65.9	234.9	4.0	11.8	1.6	6.5
MPI	138.0	274.5	11.9	23.8	7.2	14.3

TABLE VI
MPI and MPI+OpenMP performance (in GFlop/s) for Ax with C-diagonal Q-perturbed matrices on 1 and 2 nodes for CSR storage format with different number of threads per MPI process, keeping 48 threads per node

OpenMP, which is not surprising since it is almost pure OpenMP code. We can also see the influence of the value of Q. When Q = 0, we can get up to 140 GFlop/s for ELL whereas when Q = 0.8, we can only get 7 GFlop/s with 1 node.

With $A(Ax + x) + x$, Q influences the performance in a similar way as for Ax. However, there is no decrease in performance compared to the experiments on nlpkkt120. This is shown by Figure 6 that gives the performances of $A(Ax + x) + x$ with OpenMP for CSR and ELL.

Table VIII and Table IX show the performances of CSR and ELL for $A(Ax+x)+x$ with MPI and MPI+OpenMP with different values of Q for C-diagonal Q-perturbed matrices. We can see that the performance of $A(Ax+x)+x$ is lower than for Ax, similar to what happened with nlpkkt120. However, the scaling to 2 nodes is better. Indeed, the best performance for 1 node and Q = 0 is around 70 GFlop/s and 100 GFlop/s for 2 nodes and Q = 0. We can see similar results for CSR and ELL as well as

threads	0.0		0.4		0.8	
	1	2	1	2	1	2
1	133.3	265.2	12.2	24.4	7.3	14.5
2	132.1	260.0	12.2	24.4	7.3	14.5
4	131.0	261.4	12.3	25.0	7.4	14.9
6	130.8	254.4	12.4	25.3	7.6	15.3
12	131.0	253.9	12.9	26.3	9.0	17.1
24	126.5	255.5	8.6	16.8	7.5	14.1
48	64.5	241.5	3.3	11.0	1.3	5.5
MPI	140.7	279.6	12.0	24.1	7.2	14.4

TABLE VII
MPI and MPI+OpenMP performance (in GFlop/s) for Ax with C-diagonal Q-perturbed matrices on 1 and 2 nodes for ELL storage format with different number of threads per MPI process, keeping 48 threads per node

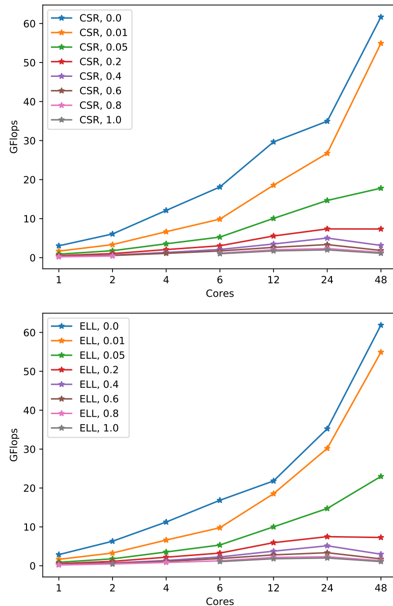

Fig. 6. $A(Ax+x)+x$ with OpenMP for C-diagonal Q-perturbed matrices with C = 100

for the different values of Q. Therefore, it means that the scaling issues are not coming from the dispersion of the non-zero elements in the matrix. In this case, we used a matrix of size $8,000,000 \times 8,000,000$, whereas nlpkkt120 has a size of $3,542,400 \times 3,542,400$. Our matrix also has a larger number of non-zero elements ($800,000,000$ vs $96,845,792$), which makes it larger both in terms of the vector sizes and the number of non-zero elements. Our application is scaling better with the C-diagonal Q-perturbed matrices than with nlpkkt120 since the matrix is larger in this case.

2) Lower number of nonzeroes per row: The observations so far on C-diagonal Q-perturbed matrices do not confirm our initial intuition that the dispersion of nonzero entries from the diagonal caused an increase in the cost of communications and lower performance. Instead, it appears that the number of nonzeroes is the main determinant. This could explain the results from Section IV.C. under a new light, since cage14 and cage15 are

threads	0.0		0.4		0.8	
	1	2	1	2	1	2
1	32.5	35.7	9.4	15.3	6.2	10.7
2	32.2	36.3	9.4	15.3	6.2	10.8
4	35.8	55.4	9.7	18.2	6.6	12.2
6	42.5	55.0	10.2	18.2	7.0	12.3
12	51.7	64.2	11.2	19.8	8.7	13.6
24	72.4	81.3	8.1	14.4	7.7	13.2
48	50.3	103.9	2.7	9.8	1.3	5.2
MPI	32.5	33.6	9.6	14.5	6.2	10.4

TABLE VIII
MPI and MPI+OpenMP performance (in GFlop/s) for $A(Ax+x)+x$ with C-diagonal Q-perturbed matrices on 1 and 2 nodes for CSR storage format with different number of threads per MPI process, keeping 48 threads per node

threads	0.0		0.4		0.8	
	1	2	1	2	1	2
1	32.5	35.2	9.5	15.3	6.2	10.7
2	31.9	36.8	9.5	15.6	6.2	10.9
4	35.4	53.8	9.9	18.5	6.5	12.2
6	41.5	55.3	10.3	18.6	6.7	12.4
12	51.3	59.3	11.4	20.3	8.7	13.0
24	71.8	83.3	8.2	14.7	7.2	10.5
48	50.3	106.1	2.9	10.3	1.3	5.6
MPI	32.6	33.1	9.6	14.8	6.3	10.4

TABLE IX
MPI and MPI+OpenMP performance (in GFlop/s) for $A(Ax+x)+x$ with C-diagonal Q-perturbed matrices on 1 and 2 nodes for ELL storage format with different number of threads per MPI process, keeping 48 threads per node

also more sparse than nlpkkt120, and showed decreased performance for our application.

In order to confirm this observation, we repeat the experiments with the OpenMP implementation on C-diagonal Q-perturbed matrices, with C = 16 and C = 8 instead of C = 100, in order to simulate cases with fewer nonzero entries. Results for C = 16 are presented in Tables 7 and 8, respectively for Ax and $A(Ax+x)+x$. Results for C = 16 are presented in Tables 9 and 10, respectively.

For C = 16, performances on both applications are slightly lower than for C = 100, and degrade faster even for minor deviations from the diagonal (Q = 0.01). For C = 8, performance is noticeably lower, at around 40 GFlop/s for Ax and 30 GFlo/ps for $A(Ax+x)+x$. This confirms that the number of nonzeroes is an important factor in the performance.

V. Conclusion

We implemented a sparse matrix-vector product and $A(Ax+x)+x$ with several sparse formats that we used to benchmark the performance of the vectorization on A64FX chips. We have seen that our simple implementation of the vectorization, using SVE intrinsics for regular Ellpack and CSR, provides only a small improvement in performance (less than 10%). We also used coordinates-based sparse storage formats but they were less efficient than CSR and ELL due to the required reduction on the resulting vector during the OpenMP parallel loop.

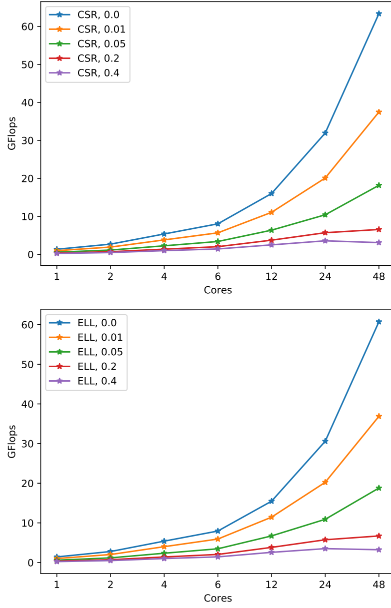

Fig. 7. Ax with OpenMP for C-diagonal Q-perturbed matrices with C = 16

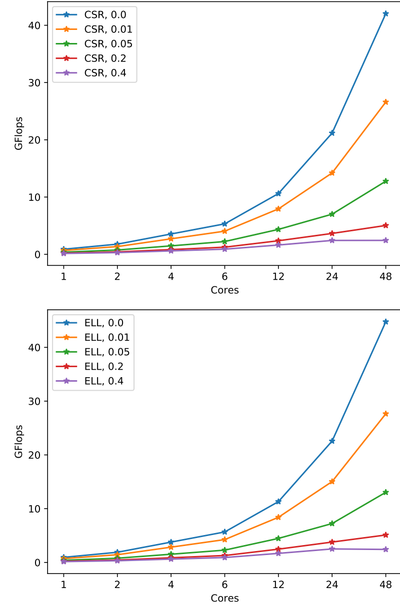

Fig. 9. Ax with OpenMP for C-diagonal Q-perturbed matrices with C = 8

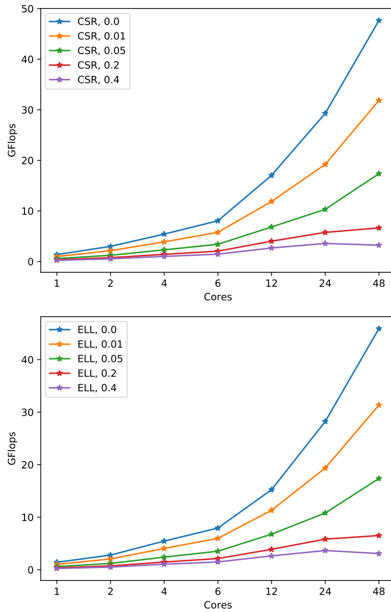

Fig. 8. $A(Ax + x) + x$ with OpenMP for C-diagonal Q-perturbed matrices with C = 16

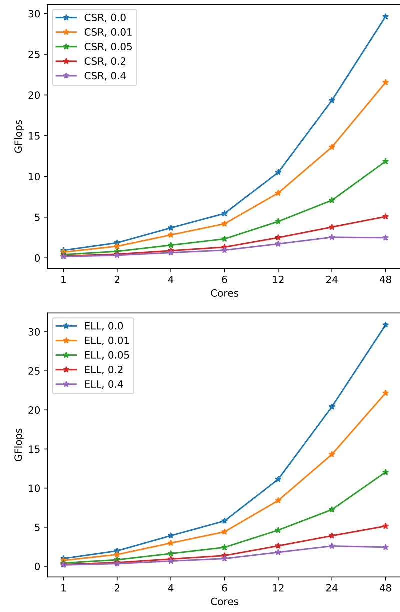

Fig. 10. $A(Ax + x) + x$ with OpenMP for C-diagonal Q-perturbed matrices with C = 8

We have shown that Ax scales well up to 2 nodes with matrices of different shapes and sizes. However, the sparse $A(Ax+x)+x$ is not scaling well with nlpkkt120, but scales well with our larger C-diagonal Q-perturbed matrices. Moreover, we have shown that MPI and MPI+OpenMP provide better performance on multiple core memory groups than OpenMP for large matrices such as our C-diagonal Q-perturbed matrices, whereas OpenMP obtained better performance on nlpkkt120. Finally, we showed that the sparsity and dispersion of the values in the matrices greatly influences the performance of the sparse matrix-vector product. Indeed, matrices with values localized in groups obtained better performance than matrices without localization.

In future work, we will focus on large-scale and multi-node applications based on the sparse matrix-vector product such as page rank and conjugate gradient. We will also investigate sparse storage formats more adapted to

the vectorization of operations. We will experiment with larger and application extracted sparse matrices. Another avenue is to explore the several compiler options for the A64FX processor that could help improve the efficiency of the vectorization, notably the tuning of prefetching for the different cache levels.

References

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, Templates for the solution of linear systems: building blocks for iterative methods. SIAM, 1994.

[2] T. Dettmers and L. Zettlemoyer, "Sparse Networks from Scratch: Faster Training without Losing Performance," CoRR, vol. abs/1907.04840, 2019. [Online]. Available: http://arxiv.org/abs/1907.04840

[3] J. Dongarra, "Report on the Fujitsu Fugaku system," University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06, 2020.

[4] C. L. Alappat, J. Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig, "Performance Modeling of Streaming Kernels and Sparse Matrix-Vector Multiplication on A64FX," CoRR, vol. abs/2009.13903, 2020. [Online]. Available: https://arxiv.org/abs/2009.13903

[5] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd ed. Society for Industrial and Applied Mathematics, 2003.

[6] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in GPU Technology Conference, 2010.

[7] M. R. Hugues and S. G. Petiton, "Sparse Matrix Formats Evaluation and Optimization on a GPU," in 2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC), Sep. 2010, pp. 122–129.

[8] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," ACM Transactions on Mathematical Software (TOMS), vol. 38, no. 1, pp. 1–25, 2011.

[9] S. Petiton and C. Weill-Duflos, "Massively parallel preconditioners for the sparse conjugate gradient method," in Parallel Processing: CONPAR 92—VAPP V. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 373–378.

[10] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-design for A64FX Manycore Processor and "Fugaku"," in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 1–15.

[11] 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, 2014.

[12] B. Bian, J. Huang, R. Dong, L. Liu, and X. Wang, "CSR2: a new format for SIMD-accelerated SpMV," in 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). IEEE, 2020, pp. 350–359.