Mixed and Multi-Precision SpMV for GPUs with Row-wise Precision Selection

Erhan Tezcan*, Tugba Torun*, Fahrican Koşar*, Kamer Kaya+, Didem Unat*
*Department of Computer Science and Engineering
Koç University, Istanbul, Turkey
+Department of Computer Science and Engineering
Sabancı University, Istanbul, Turkey
etezcan19@ku.edu.tr, ttorun@ku.edu.tr, fkosar21@ku.edu.tr, kamer.kaya@sabanciuniv.edu, dunat@ku.edu.tr

***Abstract*—Sparse Matrix-Vector Multiplication (SpMV) is one of the key memory-bound kernels commonly used in industrial and scientific applications. To improve its data movement and benefit from higher compute rates, there are several efforts to utilize mixed precision on SpMV. Most of the prior-art focus on performing the entire SpMV in single-precision within a bigger context of an iterative solver (e.g., CG, GMRES). In this work, we are interested in a more fine-grained mixed-precision SpMV, where the level of precision is decided for each element in the matrix to be used in a single operation. We extend an existing entry-wise precision based approach by deciding precisions per row, motivated by the granularity of parallelism on a GPU where groups of threads process rows in CSR-based matrices. We propose mixed-precision CSR storage methods with row permutations and describe their greater efficiency and load-balancing compared to the existing method. We also consider a multi-precision case where single and double precision copies of the matrix are stored priorly and further extend our mixed-precision SpMV approach to comply with it. As such, we leverage a mixed-precision SpMV to obtain a multi-precision Jacobi method which is faster than yet almost as accurate as double-precision Jacobi implementation, and further evaluate a multi-precision Cardiac modeling algorithm. We demonstrate the effectiveness of the proposed SpMV methods on an extensive dataset of real-valued large sparse matrices from the SuiteSparse Matrix Collection using an NVIDIA V100 GPU.**

***Index Terms*—spmv, mixed-precision, gpu, cuda**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is one of the key kernels in many scientific and engineering applications. Performance of iterative solvers for large linear systems are greatly dependent on their SpMV kernels [1]–[4]. Neural networks such as CNN heavily utilize SpMV [5], and graph analysis algorithms such as PageRank [6] and HITS [7] use SpMV within. With the ever-increasing data size, this memory-bound kernel continues to challenge high-performance applications.

SpMV has been extensively studied under a variety of storage formats and GPU kernel optimizations [8]–[13]. Independent of the formats and the respective kernels, a possible optimization is utilizing mixed-precision, such as double precision (FP64) and single precision (FP32) together. This is motivated by the fact that processors are equipped with compute units that are faster and more power-efficient at lower precisions; and lower levels of precision yield less memory and network traffic compared to higher precision arithmetic, which has been conventionally preferred due to its higher accuracy [14]–[16].

One of the mixed-precision SpMV techniques is to have different precisions at different levels of the entire algorithm using SpMV within. This is often used within the iterative solvers where a less-precise inner solver is kept at lower precision, and an outer, more-precise solver can tolerate the errors introduced, an approach known as *defect-correction* [17], [18]. Lower precisions have also been utilized in compute-intensive but error-tolerant parts of iterative solvers such as factorization and preconditioning [19]–[21]. It is shown that FP64 accuracy can be maintained for the overall application while lower precision is used for certain parts of the application [14].

Another perspective on mixed-precision SpMV is to use different precisions at a finer granularity within a single SpMV [22]–[24]. Ahmad et al. [22] decides the precision for each nonzero value of the sparse matrix stored in Compressed Sparse Row (CSR) format, and then they split it into separate FP32 and FP64 CSR matrices. They also modify the CSR-Vector kernel (originally developed by Bell & Garland [25]) to obtain a mixed-precision CSR-Vector kernel. However, their performance varies greatly from matrix to matrix, most of the time resulting in execution slower than FP64 SpMV, likely due to a load-imbalance issue inherent in this method.

In this work, we consider row-wise precision selection to better balance the workload among the threads and extend CSR and ELLPACK-R to multi-precision with slightly modified SpMV kernels [25]. We base our modified kernels on top of the open-source CUSP [26] library for NVIDIA, which is an implementation of CSR-Vector SpMV that has been used in numerous works on comparing different kernels [22], [27]–[29]. Upon deciding the precision for each row, we permute the rows such that those with the same precision type are clustered together. We demonstrate that deciding the precision for each row, and then permuting the rows to cluster the ones with the same precision is remedial to load-balancing problems within

This work has received funding from the European High-Performance Joint Undertaking under grant agreement no. 956213 and TUBITAK grant no. 120N003.

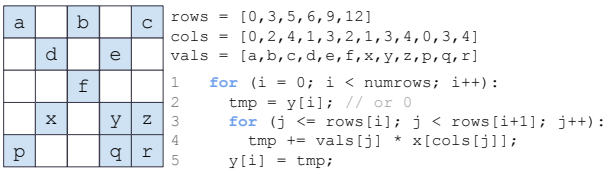


Fig. 1: CSR format example with SpMV host code

```
for (row=vec_id; row<num_rows; row+=num_vecs):
```

Fetch `rows[row]` & `rows[row+1]`
Calculate local sum with `vals`, `cols` and `x`
Reduce sums and write to `y[row]`

Fig. 2: CSR-Vector kernel

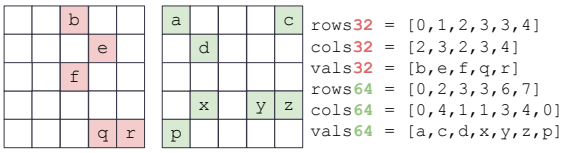


Fig. 3: Entry-wise Split example

```
for (row=vec_id; row<num_rows; row+=num_vecs):
```

Fetch `rows32[row]` & `rows32[row+1]` Fetch `rows64[row]` & `rows64[row+1]`
Calculate local sum with `vals32`, `cols32` and `x32`
Calculate local sum with `vals64`, `cols64` and `x64`
Reduce sums in FP64 and write to `y[row]`

Fig. 4: Entry-wise Split kernel

and among thread-groups.

We employ our proposed mixed-precision SpMV to be multi-precision compliant, and demonstrate its capability in two applications: a multi-precision sparse Jacobi method that can be used as a preconditioner within other iterative solvers, and a multi-precision Cardiac modeling [30] as an application of solving diffusion equations with finite volumes. Our main contributions can be listed as follows:

- We propose an easy mixed-precision method for SpMV and its CSR and ELLPACK-R based storage formats and GPU kernels.
- For a multi-precision setting where FP32 and FP64 matrices are stored in advance, we further extend the row-wise mixed-precision SpMV and implement a multi-precision Jacobi pre-conditioner and Cardiac modeling as use-cases.
- Over a dataset of 105 real-valued large matrices from SuiteSparse, for the CSR format, we demonstrate an average $1.06\times$ and up to $1.49\times$ speedup over FP64 SpMV, while the prior-art [22] achieves no speedup. FP32 SpMV with FP64 sum-reduction achieves an average speedup of $1.16\times$, which serves as an upper bound for the achievable speedup.
- For ELLPACK-R, we achieve an average $1.07\times$ and up to $1.74\times$ speedup, which is again very close to the upper bound of FP32 SpMV (with FP64 sum-reduction) with an average speedup of $1.13\times$.
- We demonstrate that multi-precision Jacobi is faster than yet as accurate as the FP64 Jacobi method. We achieve an average $1.05\times$ and $1.11\times$ and up to $1.11\times$ and $1.40\times$ speedup for Jacobi and Cardiac modeling, respectively.
- Our code is open-source and is available at https://github.com/ParCoreLab/mixed-and-multi-spmv.

II. Background

A. CSR-Vector SpMV Kernel

CSR is a widely used storage format for sparse matrices. CSR-based SpMV methods have shown reliable performance on many platforms [9], and there are a variety of CSR-based SpMV kernels for GPU. An example of CSR format and the sequential code of a CSR-based SpMV is shown in Fig. 1 and Fig. 2 respectively. In CSR format, the column indices (`cols`) and values (`vals`) are stored for each nonzero value similar to the Coordinate Format (COO); but instead of row indices, the information about number of elements per row is captured in `rows`.

In this work we focus on *CSR-Vector SpMV* [25] which has an open-source implementation (CUSP [26]), is relatively simple compared to LightSpMV [29] or Perfect-CSR SpMV [27], and requires a modest development effort for mixed-precision integration. The idea behind the CSR-Vector SpMV is to assign a group of threads called "vector" to each row, with possible vector sizes being 2, 4, 8, 16, and 32, chosen to be the largest power of two not larger than the average number of nonzeros per row. The number of blocks per grid is computed as $M/(t/v)$, the ratio of rows to vectors per block. To avoid ambiguity between the vector of this kernel and the dense vector in SpMV, we will refer to the former as a "thread group". For efficiency, consequent thread groups should access consequent memory locations, and one can partially expect this from the CSR-Vector kernel. However, using mixed-precision within a single SpMV may introduce additional irregularities to the access pattern, as well as change the thread workloads. Our work is aimed towards solving such problems.

B. Prior Art: Entry-wise Split

The prior art by Ahmad et al. [22] introduced a precision selection method as follows: if the value is in range $(-r, r)$ for some r then it is stored in FP32, otherwise FP64; where smaller r yields less error. In [22], a fixed range ($r = 1$) is applied to all matrices.

Assuming that a range has been chosen and the precisions are determined for each nonzero value, one CSR matrix with the FP64 values and another with the FP32 values are created such that each of them have the same number of rows and columns as the original. An example of this split is shown in

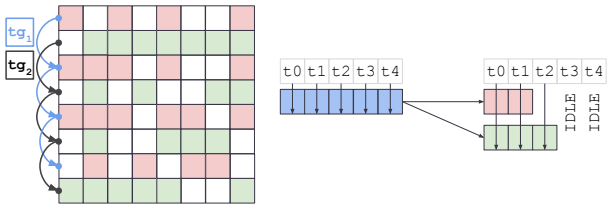


Fig. 5: Prior art: $\texttt{tg}_1$ processes 4 FP32 rows, $\texttt{tg}_2$ processes 4 FP64 rows. Some threads may be idle within a thread group.

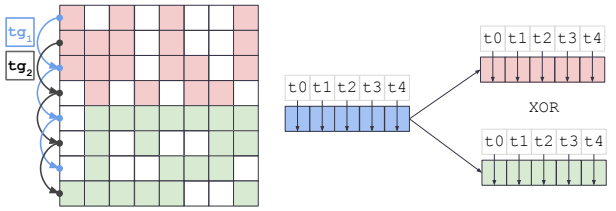


Fig. 6: Proposed: Both $\texttt{tg}_1$ & $\texttt{tg}_2$ process 2 FP32 & 2 FP64 rows. Thread work loads do not change within a thread group.

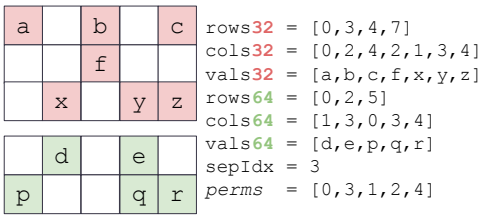


Fig. 7: Row-wise Split example

```
for (row=vec_id; row<num_rows; row+=num_vecs):
  if (row < sepIdx)
    T: Fetch rows32[row] & rows32[row+1]
       Calculate local sum with vals32, cols32 and x32
    F: Fetch rows64[row-sep] & rows64[row-sep+1]
       Calculate local sum with vals64, cols64 and x64
  Reduce sums in FP64 and write to y[row]
```

Fig. 8: Row-wise Split kernel

Fig. 3. The suffixes 32 and 64 indicate that the pointer belongs to FP32 and FP64 matrices.

In the prior art's CSR-Vector kernel implementation (Fig. 4), a thread group fetches the row pointers for both matrices and then calculates the dot product for both rows, one in FP64 and the other in FP32. This may incur load-balancing problems both within a thread group and among the thread groups. We depict these problems in Fig. 5. It is possible that certain splits result in some thread groups doing more work. Moreover, due to the way the values are indexed, threads with smaller id's may do more work when a row is split into smaller rows.

The second problem can be construed as if the matrix density (ratio of the number of nonzeros to matrix size) is getting smaller since it yields twice as many rows. Adjusting the global thread group size with respect to the new density is an option, but other rows might have different split ratios between single and double precision, so one may be better off keeping the thread group size as it is.

In summary, the entry-wise split does not necessarily take advantage of the used storage format. This is expected since their precision selection happens at the granularity of each nonzero value, but such a selection provides the leeway of being extendable to respect storage formats. This is where our new row-wise methods come forth with regards to the row-compressed formats. We propose row-level precision selection methods instead of entry-level precision to obtain more efficient mixed- and multi- precision SpMV implementations.

III. Proposed Mixed and Multi-Precision Methods

A. Row-wise Split

We propose a row-level precision selection method for SpMV, namely *row-wise split*, which determines the precision for each row separately. A row is chosen to be in FP32 if at least p percentage of its values are in some range $(-r, r)$, and all of its values are within the range of FP32 (i.e., `FLT_MAX` in C language).

We also propose a *matrix-specific range* calculated for each matrix prior to computations, instead of a fixed range. This is because although a fixed and small range would incur less absolute error, it can cause all values to be stored in FP32 for matrices with small or scaled values. For a given matrix with set of nonzero values $\mathcal{V}$, we set the range as $r = f \times (\sum_{v \in \mathcal{V}} |v|)/|\mathcal{V}|$ where f is a predefined shrinking factor chosen to adjust the range such that smaller f may lead to less error but fewer values to be stored in FP32.

After precision is decided per row, two matrices are created from the rows, one type of precision each. This has a row permutation effect (shown with `perms` in Fig.7) where the rows with same precisions are clustered together. As shown in the kernel overview in Fig. 8, a thread group checks the precision by comparing its row to the separation index (`sepIdx`), and proceeds with the dot product at the respective precision.

Row-wise split method guarantees no empty rows, by permuting the existing ones to be clustered at the bottom and ignored during SpMV. This saves redundant global accesses to the row pointer in the presence of empty rows. It also effectively solves the load-balance problems. As depicted in Fig. 6, row permutation ensures consecutive thread groups to have similar loads. Moreover, a thread group's load does not change after the split as elements of a row stay the same.

B. Row-wise Composite: A Multi-precision Compliant Method

Multi-precision and mixed-precision techniques are occasionally used interchangeably [14]. In this work, mixed-precision refers to the case that an operation uses multiple precisions; whereas multi-precision means that FP64, FP32 and mixed-precision implementations may be used throughout an application at different levels. For instance, a multi-

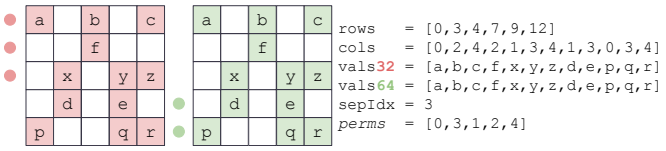


Fig. 9: Row-wise Composite Split example

```
for (row=vec_id; row<num_rows; row+=num_vecs):
    Fetch rows[row], rows[row+1]
    if (row < sepIdx)
        T  Calculate local sum with vals32, cols and x32
        F  Calculate local sum with vals64, cols and x64
    Reduce sums in FP64 and write to y[row]
```

Fig. 10: Row-wise Composite kernel

TABLE I: Memory Capacity and Bandwidth Requirements.

Storage Format	Storage Cost (Bytes)*	Bandwidth Requirement (Bytes)†
CSR FP64	$4M+12V+4$	–
CSR FP32	$4M+8V+4$	–
Prior Art	$8M+8V+4V_{64}+8$	$8M+8V+4V_{64}+8$
Row-wise Split	$4M+8V+4V_{64}+12$	$4M+8V+4V_{64}+12$
Row-wise Composite	$4M+16V+8$	$4M+8V+4V_{64}+12$

* M: #rows, V: #nonzeros, V_{64}: #nonzeros stored in FP64. † bandwidth required by the kernel

precision GMRES method using FP32 and FP64 is recently described [15], and this method requires the matrix to be stored in both precisions at the same time. Note that the extra FP32 matrix just requires a value pointer, as the row and column pointers are equivalent for both matrices.

Suppose one is to extend the multi-precision method by incorporating mixed-precision SpMV in between FP32 and FP64 steps. The aforementioned methods need two different matrices with their own row, column and value pointers, differing from those of FP64 and FP32 matrices. Consequently, these mixed-precision splits require their own memory allocations and transfers for a GPU-based execution in addition to the existing FP32 and FP64 matrices. This can either mean allocating all types once at the start or transferring the data to GPU during execution, which would greatly hinder the performance due to large amounts of data movement.

To address this issue, we propose *row-wise composite* to be used in multi-precision cases. Instead of creating two matrices as done in row-wise split, here we just create an FP32 copy of the values array of the permuted FP64 matrix. Fig. 9 and Fig. 10 show how this enables accessing the value pointer in both precisions with the same row and column pointers. Similar to the row-wise split kernel, the separation index (`sepIdx`) is used to find the precision of a row.

In Table I, we present the total storage cost and bandwidth requirement of each described mixed-precision split in bytes. Compared to the prior art, the row-wise split saves $4M$ bytes as the total number of rows stay constant. Row-wise composite has the cost of storing FP32 and FP64 CSR matrices together

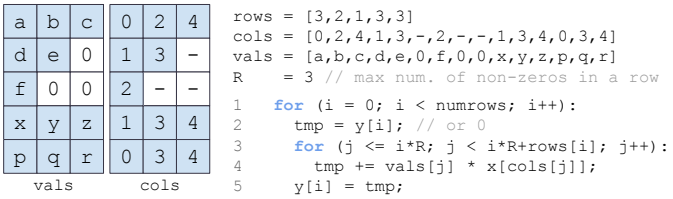


Fig. 11: ELLPACK-R format example with SpMV host code

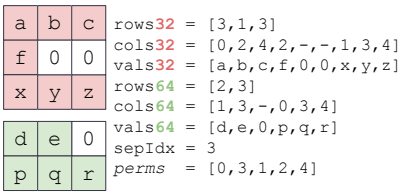


Fig. 12: ELLPACK-R Row-wise Split example

(with common row and column pointers) will cost $4M+16V+4$ bytes. The bandwidth requirements of row-wise split and row-wise composite are equivalent for mixed-precision SpMV, and they save $4M$ compared to the prior art.

C. Extending to ELLPACK-R

We extend our mixed-precision approach for ELLPACK-R [31] format. The value and column index arrays have a size equal to number of nonzeros in CSR, but here the size is $M \times R$, where M is the row count and R is the maximum number of nonzeros in a row. This enables to index a row i just with $R \times i$. The row pointers in CSR become row lengths for ELLPACK-R, where index i is the number of nonzeros in row i. Consequently, the SpMV kernels are almost identical.

Provided that the number of nonzeros per row does not vary dramatically throughout the matrix, ELLPACK-R provides better memory coalescing and is more efficient than CSR format. However, if a row has a lot more nonzeros compared to the others, this becomes memory-consuming. The splitting methodology is analogous to what is done for CSR format, the rows are treated the same way and two new ELLPACK-R matrices are created from the split (Fig. 11 and Fig. 12). For brevity, we omit the examples of other splits for ELLPACK-R.

D. Case Study: Jacobi Method

Jacobi is a well-known iterative method that can be used to solve linear systems [32], [33], and is most widely used as a preconditioner within other iterative solvers (e.g., Krylov subspace methods) for faster convergence [34]–[36]. Using Jacobi as a preconditioner is appealing since its structure it allows utilizing high parallelism in contrast to its counterparts such as ILU and SSOR [37]. The method aims to obtain an approximate solution for a linear system of equations $Ax = b$ where $A \in \mathbb{R}^{N \times N}$ is a sparse nonsingular matrix. Considering A is decomposed as $A = D + R$ where D is the diagonal component and R consists of the remaining off-diagonal entries, it iteratively solves $x^{(k+1)} = D^{-1}(b - Rx^{(k)})$ where $x^{(k)}$ is the solution at the k^{th} iteration. The SpMV operation

Algorithm 1 Mixed-Precision Jacobi Method

Require: Diagonal (D) and remaining (R) components with splitting $R = R_{32} + R_{64}$
1: Choose an initial guess $x^{(0)}$
2: **for** $k = 1, 2, ...,$ `ITERS` **do**
3: $\quad \bar{x} \leftarrow R_{64} \times x_{64}^{(k-1)} + R_{32} \times x_{32}^{(k-1)}$
4: $\quad \bar{x} \leftarrow D^{-1}(b - \bar{x})$
5: $\quad x_{64}^{(k)} \leftarrow \bar{x}, \quad x_{32}^{(k)} \leftarrow$ `float(`$\bar{x}$`)`
6: **end for**

Algorithm 2 Mixed-Precision Cardiac Modeling

Require: Diagonal (D) and remaining (R) components with splitting $R = R_{32} + R_{64}$
1: Initial guess $x^{(0)}$ read from disk.
2: **for** $k = 1, 2, ...,$ `ITERS` **do**
3: $\quad \bar{x} \leftarrow R_{64} \times x_{64}^{(k-1)} + R_{32} \times x_{32}^{(k-1)}$
4: $\quad \bar{x} \leftarrow \bar{x} + \bar{x} \odot D$
5: $\quad x_{64}^{(k)} \leftarrow \bar{x}, \quad x_{32}^{(k)} \leftarrow$ `float(`$\bar{x}$`)`
6: **end for**

in Jacobi ($Rx^{(k)}$) is required at each iteration, dominates the runtime, and constitutes the main bottleneck. We propose to utilize the mixed-precision SpMV to obtain a mixed-precision Jacobi method which is faster than the conventional FP64 Jacobi while attaining a decent accuracy.

The pseudocode of the proposed mixed-precision Jacobi method is given in Algorithm 1. Here x_{64} and x_{32} represent the solution vector x stored in FP64 and FP32, respectively. The R matrix is split as $R_{32} + R_{64}$, where R_{32} and R_{64} contain the nonzeros which are selected to be stored in FP32 and FP64, respectively. In Line 3, the mixed-precision SpMV is performed. In practice, instead of storing D as an $N \times N$ matrix, the diagonal array of D is stored in a vector d (in FP64).

We implement a single CUDA kernel for the Jacobi iteration and solution updates (lines 4, 5) where thread i computes $(b[i] - \bar{x}[i])/d[i]$ and stores it to register. It then writes this to $x_{64}[i]$ and $x_{32}[i]$, while casting the value to `float` for the latter. Although there is a slight overhead of updating the FP32 solution in addition to FP64 solution, the speedup gained from using mixed-precision SpMV is expected to amortize it. All computations take place on the GPU and there is no extra cost of transferring the solution vectors back and forth to the host. As described in the row-wise split and row-wise composite methods, the rows of the coefficient matrix are permuted prior to the computations. In the Jacobi method, the columns are further permuted in the same order with rows, i.e. a symmetric permutation is applied to make the diagonal values remain on the diagonal after reordering.

Different multi-precision implementations are possible for the Jacobi method. In a 1-step method, only the mixed-precision Jacobi is used. In a *2-step* Jacobi method, it starts with mixed-precision and moves to FP64 after a number of iterations. Similarly, in a *3-step* method, a number of iterations are done in FP32, then the method is upgraded to use mixed-precision, and the final set of iterations are done in FP64.

E. Case Study: Cardiac Modeling

In cardiac modeling, diffusion equations with finite volumes are used, where the computations for a single time-step can be shown as a matrix multiplication operation $x^{(i)} = A \times x^{(i-1)}$ [30]. The mixed precision SpMV is utilized in a similar fashion to the Jacobi, where a solution time-step becomes $\bar{x} = A_{64} \times x_{64}^{(i)} + A_{32} \times x_{32}^{(i)}$. We then update both $x_{64}^{(i)}$ and $x_{32}^{(i)}$ solution vectors by assigning $\bar{x}$. Generally, the solution update could be made just by pointer swapping; however, when we have two solution vectors to be updated, swapping alone does not suffice. Instead, an additional copy kernel that is issued after the SpMV must update both solutions. We have also observed that the diagonal values in this dataset are much larger than off-diagonals. Our absolute mean based range-selection method would therefore end up with a large range that covers most of, if not all, the off-diagonal values; the only diagonal value which is outside the range in that row would be tolerated by the 1% margin. Consequently the whole matrix would likely be stored in FP32.

Considering both the large diagonals, and the need of an extra copy kernel to update solution vectors, we extract the diagonal values and store them in an array; whereas the off-diagonal values are kept in the matrix. After SpMV, the copy kernel adds the product of solution vector and diagonal values at their respective rows, and assigns the result to both FP64 and FP32 solution vectors as described in Algorithm 2. By utilizing the row-wise composite method, we also propose multi-precision Cardiac modeling methods *1-step*, *2-step* and *3-step* similar to the way that we construct in multi-precision Jacobi.

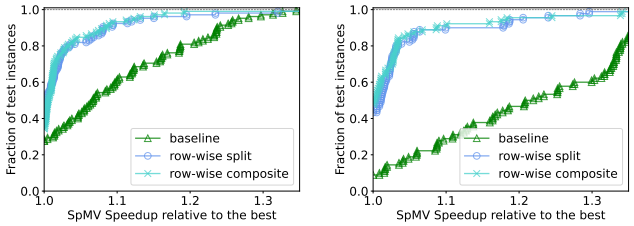
IV. Evaluations

All the performance experiments were conducted on a single node with an NVIDIA Tesla V100 GPU. CUDA 11.2 and GCC 9.3.0 with `-O3` optimization are used to compile our program. For all CUDA kernels, threads per block is set to 128.

In all mixed-precision SpMV operations, an FP32 copy of the FP64 dense vector is used in the dot-product of FP32 matrix rows. Otherwise the CUDA runtime would cast the computations to be FP64, therefore preventing any performance improvements from FP32 computations [22]. Similar to the prior research [22], [38], we do the global sum reductions in FP64, which is a well-known technique to increase the accuracy regardless of the precision used in lower precision mixed-precision methods [18].

We run 200 warm-up iterations of FP64 SpMV for each matrix. The runtime measurements are from the beginning until the end of iterations; data transfer to/from GPU are not included. Throughout this section, the *baseline* refers to the prior art [22] or the application using it for the SpMV within, where the range value r changes according to the application.

Our row-wise methods use the proposed adaptive range, based on an absolute mean with $f = 0.1$ for SpMV and Jacobi, and $f = 2$ for Cardiac modeling. A larger value is used for Cardiac since the matrix values are closer, and shrinking the calculated range often results in all values to be out of range. For all methods, the percentage p is 99, i.e., in a row 1% out-of-range values are tolerated. These are chosen empirically. In particular, $p = 100$ is too strict, making most of the rows kept in FP64, and values such as $p = 90$ result in greater errors.



(a) CSR (b) ELLPACK-R

Fig. 13: SpMV Speedup performance profiles

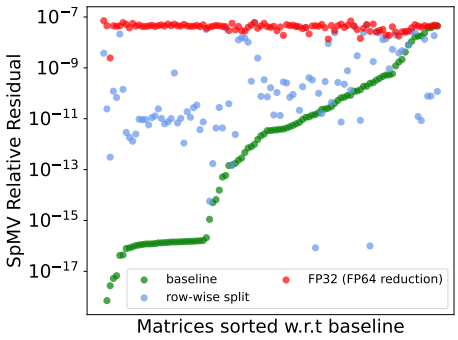


Fig. 14: SpMV CSR Relative Residuals

In the proposed row-wise methods, the matrix splitting and permuting cost is very lightweight and at around a thousandth of a single iteration of SpMV time, approximately. Moreover, in many applications, the coefficient matrix stays the same throughout the iterations, which means the splitting and permutation incur only one-time pre-processing cost.

A. SpMV Results

For SpMV, we consider $y \leftarrow y + Ax$ where the initial y is all zeros, and the x entries are (uniformly) random in the range $(-5, 5)$. The experiments are conducted on real-valued matrices from the SuiteSparse Collection [39], where the matrices have between 100K and 40M nonzeros. The range value of baseline method is $r = 1$, as in [22]. For reliable measurements, we filter out matrices whose FP64 SpMV runtime is less than 0.1 seconds. We also ignore the matrices where the row-wise split stores less than 10% of nonzeros in FP32, as the speedup will not be considerable. As such, we have used 105 matrices (out of 2,794).

Fig. 13a shows the performance profiles comparing different methods in terms of the relative speedup of SpMV CSR with respect to FP64 for 105 matrices. A performance profile [40] represents the comparison of different methods for each data instance relative to the best-performing one. A point (α, β) on the line associated to method X means that the performance of X is within α factor of the best result for a fraction β of the instances. For example, the point (1.30, 0.75) on the curve of method X means that X yields 30% better result than the best result achieved for 75% of the dataset. Therefore, the method closest to the top left corner is interpreted as the best-performing one.

As shown in Fig. 13a, our proposed row-wise split and row-wise composite methods are performing the best. Due to their close performance, further SpMV experiments are conducted with the row-wise split method. Fig. 14 shows the plot for relative residuals of SpMV with respect to FP64 for the same 105 matrices sorted in the order of the baseline method's residuals. Here, we refer the relative residual as $\|y' - y_{64}\| / \|y_{64}\|$ where y_{64} is the result of FP64 SpMV, and y' is the result of mixed-precision SpMV. FP32 SpMV with FP64 reduction serves as the upper bound of relative residual, which is incurred by storing all nonzeros in FP32.

With its small fixed range $(-1, 1)$, the baseline often has small relative residuals. Nevertheless, there are numerous instances where the row-wise split yields near or lesser residual compared to the baseline. Considering the greater speedup of the proposed method and the possibility of correcting errors with a more precise iteration that follows the mixed-precision steps, our residuals to be acceptable.

TABLE II: Average speedup and relative residuals with respect to the FP64 SpMV.

density (λ) in 10,000 *	matrix count	avg. speedup w.r.t. FP64			avg. relative residual w.r.t. FP64		
		FP32†	baseline	row-split§	FP32†	baseline	row-split§
λ <0.5	26	1.14	0.97	1.03	3.90e-08	8.31e-12	1.03e-09
0.5 <λ <1	16	1.13	1.02	1.06	3.68e-08	7.22e-12	6.53e-10
1 <λ <2.5	17	1.11	0.95	1.05	4.14e-08	8.79e-12	1.52e-11
2.5 <λ <5	24	1.13	0.98	1.07	3.30e-08	8.46e-12	6.81e-11
λ >5	22	1.26	1.10	1.11	4.49e-08	7.37e-12	4.19e-11
ALL	105	1.16	1.00	1.06	3.87e-08	8.04e-12	1.33e-10

* density $\lambda = \text{NNZ}/(M \times N) \times 10{,}000$. † FP32 SpMV with FP64 sum reduction. § row-wise split.

For the chosen 105 matrices, ELLPACK-R goes out-of-memory for 15 of them, as ELLPACK-R stores a dense matrix of size $M \times R$ for a matrix with M rows and maximum of R elements in a row. If a row has many elements, but the rest are a lot sparser, this format becomes inefficient. The speedup results for the remaining 90 matrices are given in Fig. 13b. We omit ELLPACK-R relative residuals as they are invariant to the storage format and they are similar to what is observed with CSR. Moreover, the average and maximum speedup for ELLPACK-R is higher than that of CSR, with 1.07x and 1.74x respectively. Note that FP32 with double sum-reduction serves as an upper bound for speedup with average $1.16\times$ and $1.13\times$ for CSR and ELLPACK-R respectively.

In Table II, we present the (geometric) average speedup and relative residuals grouped into five different matrix density categories of 105 matrices. For the row-wise split method, a density-speedup correlation is observed. Though its reason is not immediately obvious, the sparser matrices may have lower speedups as the irregular access to the dense vector becomes more pronounced. Note that the access pattern is not within the scope of mixed-precision SpMV nor our row permutations. FP32 with double sum reduction, serving as an upper bound of speedup and relative residual for each group, achieves up to 26% improvement with 16% on average. The table also shows that the baseline has no or insignificant speedup. We

TABLE III: Warp execution efficiency profiled by `nvprof` for 5 matrices that the percentages of nonzeros stored in FP32 (FP32%) are close between baseline and row-wise split.

matrix	avg nnz per row	FP32%		speedup		warp exec. efficiency	
		baseline	row-split	baseline	row-split	baseline	row-split
shipsec1	55.5	58.37	58.99	1.00	1.08	87.23	94.16
shipsec5	56.2	58.07	55.48	0.97	1.08	87.06	94.39
pwtk	53.4	13.18	13.96	0.90	1.01	83.07	94.19
barrier2-4	33.7	75.10	75.56	0.95	1.09	77.78	91.41
barrier2-9	33.7	75.82	76.09	0.92	1.07	77.73	91.43

have also experimented on 2,794 real-valued matrices from SuiteSparse on which the experiments of [22] are conducted, and from these, in 999 matrices, row-wise split has speedup ≥ 1; whereas this number is 354 for the baseline method, contrary to the findings in [22].

To show the effect of load-balancing, we focus on 5 matrices with FP32 nonzero percentages close to the baseline and row-wise split method. Using the NVIDIA's `nvprof`, we profile the warp execution efficiency which is defined as: "*Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor*". The results are given in Table III. For the selected matrices, the average number of nonzeros are all greater than 32, and therefore the thread group size within the mixed-precision CSR-Vector kernel is equal to the warp size 32. Hence, the profiled metric reflects the efficiency of the thread group itself. For all matrices here, row-split performs better, even though the percentages of FP32 in the matrices are similar to the baseline. As such, the performance improvement can be attributed to the more efficient warps and load-balanced threads.

B. Jacobi Method Results

We consider large real-valued sparse square matrices with no zero values in the diagonal. We applied HSL MC64 [41] permutation and scaling for diagonal-dominance, which is a sufficient condition for Jacobi convergence [42]. Permuting aims to maximize the product of the diagonal entries, and scaling aims to make the absolute value of diagonal entries 1 and the others not larger than 1. We use $r = 0.01$ as a range for the baseline method (Jacobi using SpMV with entry-wise split) since otherwise it would choose everything in FP32 after scaling.

We set the desired solution vector as $x^* \leftarrow [1/N, 2/N, \ldots, 1]^T$ to find $y \leftarrow Ax^*$ with FP64 SpMV. We set the initial guess as $x \leftarrow [0,0,\ldots,0]^T$ and solve $Ax = y$ using Jacobi. Since Jacobi is often used as a preconditioner, and to provide an explicit accuracy comparison, we run Jacobi for a fixed number (2,000) of iterations. The iterations are divided equally among the steps in multi-precision runs. The relative residual is calculated as $\|y - Ax'\|/\|y\|$ where x' is the computed solution. The comparison experiments are conducted on 8 matrices whose relative residual for FP64 Jacobi after 2000 iterations is less than 10^{-1}.

Fig. 15 shows the speedups of baseline and our methods compared to the FP64 Jacobi. In all cases, 3-step multi-precision Jacobi is the fastest (avg. 5% and up to 11% speedup)

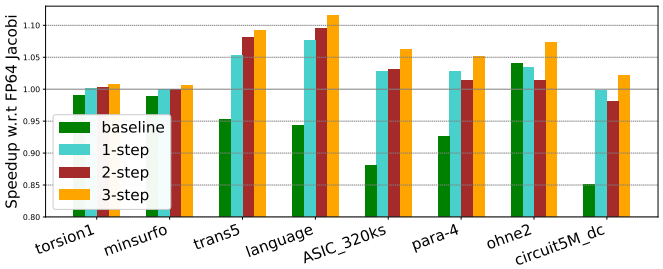


Fig. 15: Speedups for Jacobi method

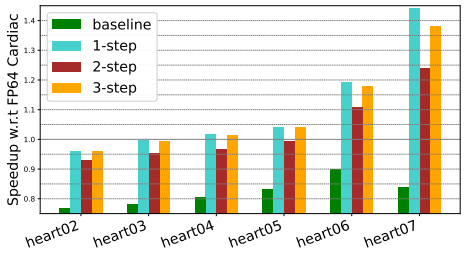


Fig. 16: Speedups for Cardiac modeling

TABLE IV: Relative residuals of Jacobi method after 2,000 iterations.

matrix	FP64	FP32*	baseline	1-step	2-step	3-step
torsion1	1.49e-16	1.24e-07	1.49e-16	1.38e-08	1.58e-16	1.58e-16
minsurfo	1.19e-16	1.97e-07	1.40e-10	5.33e-09	2.27e-16	2.44e-16
trans5	1.07e-05	1.06e-05	1.07e-05	1.07e-05	1.07e-05	1.07e-05
language	7.14e-18	1.04e-08	1.59e-11	7.28e-09	7.16e-18	7.15e-18
ASIC_320ks	4.89e-07	4.89e-07	4.89e-07	4.89e-07	4.89e-07	4.89e-07
para-4	7.68e-02	7.68e-02	7.68e-02	7.68e-02	7.68e-02	7.68e-02
ohne2	2.12e-02	2.12e-02	2.12e-02	2.12e-02	2.12e-02	2.12e-02
circuit5M_dc	6.01e-17	3.25e-08	5.36e-11	6.34e-09	6.39e-17	6.68e-17

* FP32 Jacobi refers to Jacobi using FP32 SpMV with FP64 reduction.

due to the performance gains from the lower precision steps. 2-step method also has speedup, but for some cases less than 1-step; possibly due to the additional cost of updating the FP32 copy of the solution vector, and low number of FP32 values used in split. For all cases, baseline is the slowest except for *ohne*, where the baseline keeps considerably high number of FP32 values.

Table IV shows the relative residuals for FP64 and FP32 Jacobi, the baseline, and our methods. The residual is affected by the precision selection, but not from the split that comes afterwards; since, the split itself does not change the accuracy of the computations. Although the 1-step method has relative residuals higher than that of FP64, the ones of 2-step and 3-step are close to FP64 since they can tolerate the accumulated errors in the prior less-precise steps, as expected. This is in line with our expectations that lower precision steps can be corrected by the following higher precision steps.

C. Cardiac Modeling Results

We use a set of 6 generated meshes (matrices) from [43] each having larger than 100K nonzeros and an initial dense solution vector. The ground-truth FP64 Cardiac modeling does not extract the diagonal, and uses pointer swapping to update

the solution. Each method runs for 8,000 iterations. All off-diagonals for this dataset are less than 1, so again the baseline range $r = 1$ is infeasible, instead we use $r = 1e-5$, which is close to the absolute mean value for all 6 matrices.

Figure 16 shows the speedups of baseline and our methods with respect to the FP64 Cardiac modeling for 6 matrices sorted in increasing order of number of nonzeros. Despite the extra copy kernel overhead, 1-step achieves an average of $1.11\times$ and up to $1.40\times$ speedup, while also being faster than 2-step and 3-step. This indicates that FP64 steps are slow enough to affect the entire execution time. The speedup increases with the number of nonzeros, which may be due to better cache utilization of lower precision. The relative residuals are computed as described in SpMV evaluations, and are found to be not much different among methods (unlike Jacobi), thus are omitted for brevity.

V. Related Work

Scientific applications often benefit from high precision floating point arithmetic since it has the advantage of leading higher accuracy. On the other hand, it has higher data transfer and computational costs. One solution for this trade-off is the utilization of mixed-precision arithmetic, which has been well studied for both dense and sparse numerical methods [19], [44]–[49]. For sparse linear algebra methods, which are mostly bandwidth-bound, mixed-precision algorithms are favored to reduce the communication volume and memory access [14].

Mixed-precision arithmetic is utilized for both direct and iterative methods to get the accuracy as good as FP64 and cost as low as the FP32 arithmetic [50]. Utilizing mixed-precision arithmetic is shown to improve the runtime and energy consumption of iterative solvers on GPU [51]. Buttari et al. [21] utilize mixed-precision iterative refinement for sparse direct and iterative solvers by performing the most expensive steps in FP32 and the less important ones in FP64.

Anzt et. al. [50] implement an adaptive precision Jacobi iterative method by utilizing varying mantissa lengths. Then in [52], a mixed-precision block-Jacobi preconditioner is proposed by using high precision for all the computations after handling a part of the preconditioner in low precision. Although this approach has the potential for reducing the runtime and energy costs, its implementation is not practical since it requires extra knowledge of the matrix properties (e.g. condition number) and an optimized data conversion procedure.

Amestoy et al. [23] demonstrate mixed-precision low-rank approximations, and they use bfloat16, FP32 and FP64 to obtain a mixed-precision Singular Value Decomposition (SVD). Ooi et al. [24] implement a mixed-precision (FP32 and FP64) Hierarchical Matrix (H-Matrix) vector multiplication. H-Matrix is a dense matrix approximation with low-rank approximated submatrices.

Mixed-precision SpMV is implemented in libraries such as NVIDIA cuSPARSE; however, these focus on lower-precisions and they only allow one type of precision for the matrix and vector during SpMV. To our best knowledge, multiple precisions within the inputs in a single SpMV is not available yet.

Ahmad et al. [22] proposed a mixed-precision SpMV kernel for iterative solvers by storing some entries in FP32 and the rest in FP64. Their approach accelerates both the computation and the data movement, and its accuracy loss with respect to the FP64 SpMV is low. Although speedup is reported for some matrices, the performance varied greatly for each matrix. We believe the performance fluctuations can be attributed to the load-balancing issues discussed in this work.

VI. Conclusion & Future Work

This work has shown how selecting precision with respect to the storage format improves the performance of SpMV. Although the speedups might be considered as modest, considering the competitiveness of SpMV research, we believe even small amounts of speedup should be noted. Our methodology is based on CSR format; nevertheless, we demonstrate its effectiveness with ELLPACK-R, and we further believe that the same methodology will work on various other formats (e.g., CSC, DIA). Furthermore, our methods are orthogonal to the underlying SpMV implementation and can easily be integrated to other implementations of SpMV.

With the multi-precision compliant splits, our objective has been to obtain a matrix split such that changing precisions would not affect the row and column pointers, and thus one can adjust the precision effortlessly by accessing their respective value pointers. We have demonstrated the applicability of our methods in an extensive set of sparse matrices, on a Jacobi method that uses multiple precisions at different times to achieve FP64 accuracy with lesser execution time, and a Cardiac modeling application that yields speedup with modest change in residuals.

Since FP16 (half-precision) is increasingly becoming prevalent among HPC and ML, it is a clear direction of future work to extend our methodology to accommodate lower-precisions and possibly leverage Tensor Cores in doing so. A further Multi-GPU extension of this work, although an orthogonal direction of optimization, is certainly possible. Scenarios where each GPU works on a different precision, or where the split matrices are further split to fit into multiple GPUs are of consideration; however, effort must go into load-balancing the work among them.

Acknowledgments

This work has received funding from the European High-Performance Joint Undertaking under grant agreement no. 956213 and TUBITAK grant no. 120N003. We thank Simula Research Labarotory in Norway for providing access to the eX3 machine, where the experiments are performed. We especially thank Prof. Xing Cai for his immense help throughout this project, as well as Dr. James Trotter and Julie Johanne Uv for their assistance and providing the dataset for Cardiac modeling.

REFERENCES

[1] A. Cevahir, A. Nukada, and S. Matsuoka, "Fast conjugate gradients with multiple GPUs," in *ICCS*, 2009.

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, p. 917–924, Jul. 2003.

[3] D. Boland and G. A. Constantinides, "Optimizing memory bandwidth use and performance for matrix-vector multiplication in iterative methods," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 3, Aug. 2011.

[4] H. Anzt, S. Tomov, P. Luszczek, W. Sawyer, and J. Dongarra, "Acceleration of GPU-based Krylov solvers via data transfer reduction," *Int. J. High Perf. Comp. Appl.*, vol. 29, no. 3, pp. 366–383, 2015.

[5] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," *SIGPLAN Not.*, vol. 53, no. 1, p. 94–108, Feb. 2018.

[6] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.

[7] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, p. 604–632, Sep. 1999.

[8] S. AlAhmadi, T. Mohammed, A. Albeshri, I. Katib, and R. Mehmood, "Performance analysis of sparse matrix-vector multiplication (SpMV) on graphics processing units (GPUs)," *Electronics*, vol. 9, no. 10, 2020.

[9] G. Tan, J. Liu, and J. Li, "Design and implementation of adaptive SpMV library for multicore and many-core architecture," *ACM Trans. Math. Softw.*, vol. 44, no. 4, Aug. 2018.

[10] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[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 J. Sci. Comp.*, vol. 36, no. 5, pp. C401–C423, 2014.

[12] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *Proc. 1st Int. Conf. High Perf. Comput. Commun.*, ser. HPCC'05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 807–816.

[13] H.-V. Dang and B. Schmidt, "CUDA-enabled sparse matrix–vector multiplication on GPUs using atomic operations," *Parallel Computing*, vol. 39, no. 11, pp. 737–750, 2013.

[14] A. Abdelfattah *et al.*, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *Int. J. High Perf. Comp. Appl.*, vol. 35, no. 4, pp. 344–369, 2021.

[15] J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman, and S. Rajamanickam, "Experimental evaluation of multiprecision strategies for GMRES on GPUs," in *IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2021, pp. 469–478.

[16] S. F. McCormick, J. Benzaken, and R. Tamstorf, "Algebraic error analysis for mixed-precision multigrid solvers," *ArXiv*, vol. abs/2007.06614, 2020.

[17] E. Carson and N. J. Higham, "Accelerating the solution of linear systems by iterative refinement in three precisions," *SIAM J. Sci. Comput.*, vol. 40, no. 2, pp. A817–A847, 2018.

[18] M. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi, "Solving lattice QCD systems of equations using mixed precision solvers on GPUs," *Comput. Phys. Commun.*, vol. 181, no. 9, pp. 1517–1528, 2010.

[19] A. Abdelfattah, S. Tomov, and J. Dongarra, "Investigating the benefit of FP16-enabled mixed-precision solvers for symmetric positive definite matrices using GPUs," in *Comput. Sci.* Cham: Springer Int. Publ., 2020, pp. 237–250.

[20] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Comput. Phys. Commun.*, vol. 180, no. 12, pp. 2526–2533, 2009.

[21] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Trans. Math. Softw.*, vol. 34, no. 4, Jul. 2008.

[22] K. Ahmad, H. Sundar, and M. Hall, "Data-driven mixed precision sparse matrix vector multiplication for GPUs," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Dec. 2019.

[23] P. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary, "Mixed Precision Low Rank Approximations and their Application to Block Low Rank LU Factorization," 2021, hal-03251738v2.

[24] R. Ooi, T. Iwashita, T. Fukaya, A. Ida, and R. Yokota, "Effect of Mixed Precision Computing on H-Matrix Vector Multiplication in BEM Analysis," in *Proc. Int. Conf. High Perf. Comput. Asia-Pacific Region*. ACM, 2020, p. 92–101.

[25] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Conf. High Perf. Comput. Netw. Storage Anal.*, ser. SC '09. New York, NY, USA: ACM, 2009.

[26] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2014, version 0.5.0. [Online]. Available: http://cusplibrary.github.io/

[27] G. He and J. Gao, "A Novel CSR-Based Sparse Matrix-Vector Multiplication on GPUs," *Math. Prob. Eng.*, vol. 2016, p. 8471283, Apr 2016.

[28] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," *Proc. 29th ACM Int. Conf. Supercomp.*, 2015.

[29] Y. Liu and B. Schmidt, "LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs," in *IEEE 26th Int. Conf. Appl.-Specif. Syst. Archit. Process. Proc.*, 2015, pp. 82–89.

[30] J. Langguth, M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai, "Scalable heterogeneous cpu-gpu computations for unstructured tetrahedral meshes," *IEEE Micro*, vol. 35, no. 4, pp. 6–15, 2015.

[31] F. Vázquez, J. J. Fernández, and E. M. Garzón, "A new approach for sparse matrix vector product on nvidia gpus," *Concurr. Comput.*, vol. 23, no. 8, p. 815–826, jun 2011.

[32] P. P. Pratapa, P. Suryanarayana, and J. E. Pask, "Anderson acceleration of the Jacobi iterative method: An efficient alternative to Krylov methods for large, sparse linear systems," *J. Comput. Physics*, vol. 306, pp. 43–54, 2016.

[33] A. D. Gunawardena, S. Jain, and L. Snyder, "Modified iterative methods for consistent linear systems," *Linear Algebra Appl.*, vol. 154, pp. 123–143, 1991.

[34] E. Turkel, "Preconditioning techniques in computational fluid dynamics," *Annu. Rev. Fluid Mech.*, vol. 31, no. 1, pp. 385–416, 1999.

[35] S. Chan, K. Phoon, and F. Lee, "A modified Jacobi preconditioner for solving ill-conditioned biot's consolidation equations using symmetric quasi-minimal residual method," *Int. J. Numer. Anal. Methods. Geomech.*, vol. 25, no. 10, pp. 1001–1025, 2001.

[36] F.-H. Lee, K. Phoon, K. Lim, and S. Chan, "Performance of Jacobi preconditioning in Krylov subspace solution of finite element equations," *Int. J. Numer. Anal. Methods. Geomech.*, vol. 26, no. 4, pp. 341–372, 2002.

[37] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[38] P. Grigoraş, P. Burovskiy, W. Luk, and S. Sherwin, "Optimising Sparse Matrix Vector multiplication for large scale FEM problems on FPGA," in *26th Int. Conf. Field-Program. Log. Appl.*, 2016, pp. 1–9.

[39] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[40] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Math. Prog.*, vol. 91, no. 2, pp. 201–213, 2002.

[41] "HSL. A collection of Fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk/."

[42] R. Bagnara, "A unified proof for the convergence of Jacobi and Gauss–Seidel methods," *SIAM Review*, vol. 37, no. 1, pp. 93–97, 1995.

[43] H. Martinez-Navarro, B. Rodriguez, A. Bueno-Orovio, and A. Minchole, "Repository for modelling acute myocardial ischemia: simulation scripts and torso-heart mesh, University of Oxford," 2019.

[44] A. Alvermann *et al.*, "Benefits from using mixed precision computations in the ELPA-AEO and ESSEX-II eigensolver projects," *Jpn. J. Ind. Appl. Math.*, vol. 36, no. 2, pp. 699–717, 2019.

[45] M. Arioli and I. S. Duff, "Using FGMRES to obtain backward stability in mixed precision," *Electron. Trans. Numer. Anal.*, vol. 33, pp. 31–44, 2009.

[46] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, "Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores," *SIAM J. Sci. Comp.*, vol. 42, no. 3, pp. C124–C141, 2020.

[47] M. Emans and A. van der Meer, "Mixed-precision AMG as linear equation solver for definite systems," *Procedia Comput. Sci.*, vol. 1, no. 1, pp. 175–183, 2010.
[48] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers," in *Int. Conf. High Perf. Comp. Netw. Storage Anal.* IEEE, 2018, pp. 603–613.
[49] J. D. Hogg and J. A. Scott, "A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems," *ACM Trans. Math. Soft.*, vol. 37, no. 2, pp. 1–24, 2010.
[50] H. Anzt, J. Dongarra, and E. S. Quintana-Ortí, "Adaptive precision solvers for sparse linear systems," in *Proc. 3rd Int. Ws. Energy Effic. Supercomp.*, 2015, pp. 1–10.
[51] H. Anzt and E. Quintana-Ortí, "Improving the energy efficiency of sparse linear system solvers on multicore and manycore systems," *Philos. Trans. Royal Soc. A*, vol. 372, 2014.
[52] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, "Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers," *Concurr. Comput.*, vol. 31, no. 6, p. e4460, 2019.