Exploring Mixed and Multi-Precision SpMV for GPUs

by

Erhan Tezcan

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

February 17, 2022

Exploring Mixed and Multi-Precision SpMV for GPUs

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Erhan Tezcan

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Assist. Prof. Didem Unat (Advisor)

Assoc. Prof. Kamer Kaya

Prof. Öznur Özkasap

Date:

Dedicated to my family, to whom I owe everything.

ABSTRACT

Exploring Mixed and Multi-Precision SpMV for GPUs
Erhan Tezcan
Master of Science in Computer Science and Engineering
February 17, 2022

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 for SpMV. Most of the prior-art focus on performing the SpMV in different precisions throughout the entire application, such as an iterative solver (e.g., CG, GMRES) where certain steps can be done with lower precisions. More recently, methods of using mixed-precision within a single SpMV has been of consideration. For instance, one can decide precision for each non-zero value in the matrix, and then split the input into multiple matrices with their respective precisions; or, decide the precision for each block in a given block-diagonal matrix format.

In this work, we are interested in this more fine-grained approach of mixed-precision SpMV. To this extent, we extend an existing entry-wise precision based approach by deciding precision for each row, motivated by the granularity of parallelism on a GPU where groups of threads process rows in row-compressed sparse matrices. We propose mixed-precision CSR storage methods with row permutations and describe its greater load-balance 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.

To evaluate our methods, we apply them in two real-life applications: a multi-precision Jacobi method and a multi-precision Cardiac modeling application. We further extend our mixed-precision methodology to be used with the ELLPACK-R format. 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.

ÖZETÇE

GPU'lar için CSR Tabanlı Karışık ve Çoklu-Hassasiyetli SpMV
Erhan Tezcan
Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans
17 Şubat 2022

Seyrek Matris-Vektör Çarpımı (SpMV), endüstriyel ve bilimsel uygulamalarda sıklıkla kullanılan bellek-bağımlı kilit çekirdeklerden biridir. Veri hareketini iyileştirip daha yüksek işlem gücünden yararlanmak üzere karışık-hassasiyetin SpMV'de kullanılması için birtakım çalışmalar yapılmıştır. Geçmiş çalışmaların çoğunluğu, bir iteratif çözücü (ör. CG, GMRES) gibi içerisinde düşük hassasiyet kullanılabilen, daha büyük bir uygulama boyunca farklı hassasiyetlerde SpMV kullanmaya odaklanmıştır. Daha yakın zamanda ise, sadece bir SpMV içerisinde karışık-hassasiyet kullanmanın yolları değerlendirilmiştir. Örneğin, matris içerisindeki her bir sıfır-olmayan değer için hassasiyet seçilip, girdi bu hassasiyetlere göre birden fazla matrise ayrılabilir; veya, bir blok-diagonal matrisin her bloğu için hassasiyete karar verilebilir.

Bu çalışmada, bahsedildiği gibi daha ince-taneli karışık-hassasiyet yaklaşımı ile ilgileniyoruz. Bu minvalde, iplik gruplarının satır-sıkıştırılmış seyrek matrisleri satır bazında işlediği GPU'lardaki paralellik taneliğinden yola çıkarak, var olan bir eleman bazında hassasiyet karar verme yaklaşımı, satır bazında karar verebilmek üzere genişletiyoruz. Satır permutasyonları kullanarak karışık-hassasiyetli CSR depolama metodları öneriyor, ve bu metodların var olan metoda göre daha üstün yük-dengeleme özelliğini açıklıyoruz. Dahası, tek ve çift hassasiyetli matrislerin halihazırda depolandığı bir çoklu-hassasiyet senaryosunu göz önünde bulundurarak, önerdiğimiz karışık-hassasiyet SpMV yaklaşımımızı bu senaryoda da kullanılmak üzere genişletiyoruz.

Metodlarımızı değerlendirmek için, bir çoklu-hassasiyet Jacobi yöntemi ve bir de Kardiyak modelleme uygulaması olmak üzere iki uygulamada kullanıyoruz. Ek olarak, karışık-hassasiyet yöntemimizi ELLPACK-R formatıyla kullanılmak üzere genişletiyoruz. Önerilen SPMV metodumuzun etkililiğini, NVIDIA V100 GPU kullanarak SuiteSparse Matrix Collection içerisindeki gerçel-değerli büyük seyrek matrislerden oluşan kapsamlı bir veri seti üzerinden gösteriyoruz.

ACKNOWLEDGMENTS

I can not express enough my gratitude towards my research advisor *Asst. Prof. Didem Unat*, who was always there to pick me up no matter how tough times were. Although I was reluctant to study under this domain at first, with her guidance and mentor-ship, I have grown to be a better computer engineer, incomparable to what I was when I first started this degree. I am further grateful to *Dr. Tuğba Torun* and *Assoc. Prof. Kamer Kaya*, whom have shown perennial guidance throughout my work.

I thank *Prof. Xing Cai* from Simula, for his immense help throughout this project and providing access to Simula eX3 machine, where the experiments discussed in this work were performed. I further thank *Dr. James Trotter* and *Julie Johanne Uv* from Simula for their assistance on this project, and providing the dataset for Cardiac modeling.

This project has received funding from the European High-Performance Joint Undertaking under grant agreement no. 956213 and TUBITAK grant no. 120N003. Prior to that, it had received funding from TUBITAK grant no. 118E801.

I am grateful to have been a member of *Parallel and Multi-core Computing Laboratory* of Koç University, and I thank every and each of my colleagues there for our pleasurable time together. Finally, I thank my family for their never-ending support, and helping me be the best version of myself.

TABLE OF CONTENTS

List of Tables ix

List of Figures x

Abbreviations xi

Chapter 1: Introduction 1

Chapter 2: Background 5

2.1 Compressed Sparse Row 5

2.2 Accelerating SpMV using GPUs 5

Chapter 3: Mixed-Precision SpMV Methods 8

3.1 Prior Art: Entry-wise Split 9

3.2 Row-wise Split 10

3.3 Row-wise Composite: A Multi-Precision Compliant Row-wise Split . 11

3.4 Row-wise Dual: Multi-Precision Compliance without Row Permutations 13

3.5 Extending to ELLPACK-R 15

Chapter 4: Case Studies 16

4.1 Case Study: Jacobi Method 16

4.2 Case Study: Cardiac Modeling 18

Chapter 5: Evaluations 20

5.1 SpMV Results 21

5.1.1 ELLPACK-R Results 25

5.2 Jacobi Method Results 25

5.3 Cardiac Modeling Results 27

Chapter 6: Related Work **30**

Chapter 7: Conclusions **32**

Bibliography **33**

LIST OF TABLES

3.1 Memory Capacity Requirements. M: number of rows, V: number of nonzeros, V_{64}: number of nonzeros stored in FP64. 14

5.1 Average speedup of different precision SpMV methods with respect to the FP64 SpMV implementation. The matrices are grouped according to their density $\lambda = NNZ/(M \times N) \times 10{,}000$. Here, row-split refers to row-wise split; and FP32 refers to FP32 SpMV with FP64 sum reduction. 23

5.2 Warp execution efficiency profiled by `nvprof` for 5 matrices where the percentage of nonzeros stored in FP32 (shown as FP32%) are close between baseline and row-wise split (referred to as row-split in the table). 24

5.3 Relative residuals of Jacobi method after 2,000 iterations. 27

5.4 Relative residuals of Cardiac modeling after 8,000 iterations. 28

LIST OF FIGURES

2.1 CSR format example and CSR-Vector [Bell and Garland, 2009] SpMV kernel overview . . . 5

3.1 An example and kernel overview for prior art (entry-wise split) [Ahmad et al., 2019] . . . 9
3.2 An example and kernel overview for Row-wise Split method . . . 10
3.3 Workload distribution among thread groups with mixed-precision SpMV, demonstrated with 2 thread groups ($\mathtt{tg}_1$ and $\mathtt{tg}_2$) processing an 8×8 matrix. . . . 11
3.4 An example and kernel overview for Row-wise Composite Split method 12
3.5 An example and kernel overview for Row-wise Dual Split method . . 13
3.6 ELLPACK-R format example and row-wise split example . . . 15

5.1 Speedup performance profile and relative residual plot for SpMV (CSR) 21
5.2 Speedup performance profile and relative residual plot for SpMV (ELLPACK-R) . . . 25
5.3 Jacobi Speedups with respect to FP64 Jacobi method . . . 27
5.4 Cardiac Speedups with respect to FP64 Cardiac method . . . 28

ABBREVIATIONS

SpMV	Sparse Matrix-Vector Multiplication
CSR	Compressed Sparse Row
FP32	32-bit Floating-Point Representation
FP64	64-bit Floating-Point Representation
GPU	Graphics Processing Unit
CG	Conjugate Gradient
GMRES	Generalized Minimal Residual Method
IEEE	Institute of Electrical and Electronics Engineers
CUDA	Compute Unified Device Architecture

Chapter 1

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 [Cevahir et al., 2009, Bolz et al., 2003, Boland and Constantinides, 2011, Anzt et al., 2015b]. Neural networks such as CNN heavily utilize SpMV [Zhao et al., 2018], and graph analysis algorithms such as PageRank [Page et al., 1999] and HITS [Kleinberg, 1999] use SpMV within. With the ever-increasing data size, this memory-bound kernel continues to challenge high-performance applications.

SpMV has been extensively studied by many researchers under a variety of sparse storage formats and GPU kernel optimizations [AlAhmadi et al., 2020, Tan et al., 2018, Bell and Garland, 2008, Kreutzer et al., 2014, Vuduc and Moon, 2005]. Independent of the storage format and its respective kernel, an optimization technique is to utilize mixed-precision, such as double precision (FP64) and single precision (FP32) together in solving a linear system. 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 [Abdelfattah et al., 2021, Loe et al., 2021, McCormick et al., 2020].

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 in the domain of iterative solvers where a less-precise inner solver is kept at lower precision, and an outer more-precise solver can tolerate the errors introduced within,

an approach known as *defect-correction* [Carson and Higham, 2018, Clark et al., 2010]. Lower precisions have also been utilized in compute-intensive but error-tolerant parts of iterative solvers such as factorization and preconditioning [Abdelfattah et al., 2020, Baboulin et al., 2009, Buttari et al., 2008]. As such, it has been demonstrated that FP64 accuracy can be maintained for the overall application while lower precision is used for certain parts of the application [Abdelfattah et al., 2021]. Nevertheless, there are caveats to using lower precision. Ill-conditioned matrices when used within mixed-precision iterative solvers, may suffer from the lost precision [Baboulin et al., 2009]. Furthermore, casting to lower precision may introduce overflow, underflow or subnormal numbers; working with such values have been shown to reduce performance [Zounon and Higham, 2020].

A more recent perspective on mixed-precision SpMV optimization is to use different precisions at a finer granularity within a single SpMV [Grigoraş et al., 2016, Ooi et al., 2020, Amestoy et al., 2021, Ahmad et al., 2019]. We defer further discussion of these papers to the related work. Nevertheless, it is important to note the work by Ahmad et al. [Ahmad et al., 2019], where precision is decided for each non-zero value in the sparse matrix, and then the CSR matrix is split into separate FP32 and FP64 CSR matrices. They also modify the CSR-Vector kernel (originally developed by Bell & Garland [Bell and Garland, 2009]) to obtain a mixed-precision CSR-Vector kernel.

In this work we consider row-wise precision selection to maintain a more balanced workload among threads and extend the capabilities of the CSR format with slightly modified CSR-Vector SpMV kernel implementations [Bell and Garland, 2009]. We base our modified kernels on top of the open-source CUSP [Dalton et al., 2014] library for NVIDIA, which is an implementation of CSR-Vector SpMV that has been used in numerous works on comparing different kernels [Ahmad et al., 2019, He and Gao, 2016, Liu and Vinter, 2015, Liu and Schmidt, 2015]. Upon deciding the precision for each row, we permute the rows such that those with the same precision type are clustered together. Considering the focus on SpMV per se, together with the usage of FP32 and FP64, the state-of-the-art data-driven approach is the work by Ahmad et al. [Ahmad et al., 2019]. 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. We identify the load-imbalance within and among thread groups, which are orthogonal problems related to the number of elements in a row and the rows processed by a thread group. We demonstrate that deciding precision for each row, and then permuting rows to cluster those with the same precision is remedial to both problems. Lastly, we introduce 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 as an application of solving diffusion equations with finite volumes. Although our discussions are based on CSR format, we demonstrate its extension to ELLPACK-R format.

Our main contributions can be listed as follows:

- We propose an easy mixed-precision method for SpMV and its CSR-based storage format and GPU kernel.

- We describe row-wise matrix split methods that improve warp efficiency of the mixed-precision SpMV using row permutations.

- Taking into account a multi-precision setting where FP32 and FP64 are stored in advance, we further describe row-wise mixed-precision SpMV and implement a mixed-precision Jacobi pre-conditioner as a use-case.

- Over a dataset of 105 real-valued large sparse matrices from SuiteSparse, we demonstrate an average 1.06x and up to 1.49x speedup over FP64 SpMV with CSR format, where the prior-art [Ahmad et al., 2019] achieves no speedup.

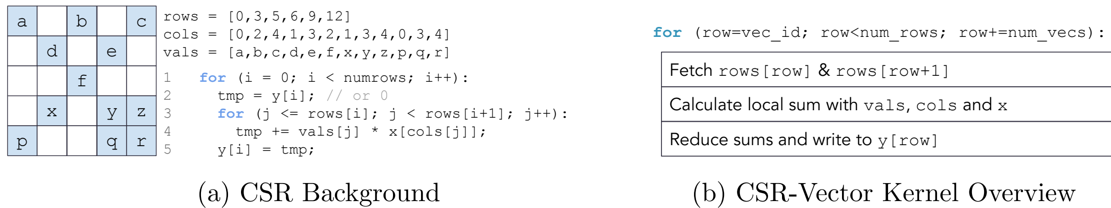
- We demonstrate that multi-precision Jacobi with mixed-precision SpMV within can be faster than yet as accurate as the FP64 Jacobi method.

- We implement multi-precision Cardiac modeling, as an application of solving diffusion equations with finite volumes [Langguth et al., 2015]. We observe an

average 1.11x and up to x1.40 speedup.

- We apply the proposed CSR-based methodology to do SpMV with ELLPACK-R format, and obtain an average 1.07x and up to x1.74 speedup.

The rest of this thesis is organized as follows: In chapter 2 we give a background of CSR format and the CSR-Vector SpMV kernel, together with brief details on GPU parallelization with CUDA. Next, in chapter 3 we present the prior art [Ahmad et al., 2019], and our proposed mixed-precision SpMV method, as well as multi-precision compliant SpMV methods in. In chapter 4 we describe the multi-precision Jacobi method and Cardiac modeling case-studies. In chapter 5 we discuss our evaluations on all presented methods and the prior art. Following the related work in chapter 6, we finally make our conclusions in chapter 7.



(a) CSR Background (b) CSR-Vector Kernel Overview

Figure 2.1: CSR format example and CSR-Vector [Bell and Garland, 2009] SpMV kernel overview

Chapter 2

BACKGROUND

In this section, we first describe Compressed Sparse Row (CSR) sparse storage format. Then, we show the CSR-Vector SpMV kernel and briefly describe how CUDA is used for GPU parallelization.

2.1 *Compressed Sparse Row*

The CSR format is a common storage format for sparse matrices, and CSR-based SpMV methods have shown reliable performance on many platforms [Tan et al., 2018]. An element is accessed with respect to its order within a row; for instance, the kth element in row r is indexed by $\texttt{rows}[r]+k$. This index can be used to get the element itself from `vals`, or to get its column index from `cols`. The total number of elements in row r is given by $\texttt{rows}[r+1] - \texttt{rows}[r]$. An example of CSR format and the sequential code of a CSR-based SpMV is shown in Figure 2.1a.

2.2 *Accelerating SpMV using GPUs*

Today, GPUs are used beyond their graphics processing capabilities, in the form of general-purpose GPUs (GPGPU) [Luebke et al., 2006]. To this extent, there are frameworks to enable GPU programmability. Of these frameworks one of the

most popular, if not the most, is CUDA [Nickolls et al., 2008] by NVIDIA. CUDA programming follows single-instruction multiple-data paradigm, and threads are distributed across the data, to process it independently. The GPU is thought of as a grid of blocks, and each block has a number of threads. Threads within a block can be synchronized at the block level; however, synchronization across the entire GPU is not possible without the more advanced features such as cooperative groups, or host-side barriers between kernel launches. At the smaller scale within a block, threads are issued in groups of 32, which is called a warp. Threads within a warp may execute one type of operation at a time, so the most efficient case is when all 32 threads are on the same instruction [NVIDIA, 2022]. This in effect indicates that threads are synchronized within a warp by definition, as they are unable to work on different instructions at a time.

From a variety of CSR-based GPU kernels on SpMV, we focus on *CSR-Vector SpMV* [Bell and Garland, 2009], based on the open-source implementation of CUSP [Dalton et al., 2014]. CSR-Vector is relatively simple and intuitive compared to LightSpMV [Liu and Schmidt, 2015] or Perfect-CSR SpMV [He and Gao, 2016] and requires a minuscule development effort for the mixed-precision approaches, that are described in the following section. The idea behind the CSR-Vector SpMV kernel is to assign a group of threads called "vector" to each row, with possible vector sizes being 2, 4, 8, 16, and 32. Effectively, the largest thread group is a warp. The vector size is chosen to be the largest of the possible values less than or equal to the average number of nonzeros per row. For instance, if the average number of nonzeros is 14.5, then the vector size to be 8. Suppose vector size is v and threads per block is t, then the blocks per grid is computed as $\#\texttt{rows}/(t/v)$, which can be interpreted as 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".

Each thread group has an ID, respecting the number of thread groups. Based on this ID, they fetch the respective row pointers, and compute the dot product of that row. Each thread keeps a local sum as they compute the dot product, which then sum-reduced to the first thread of that thread group. Only then, that first

thread writes the result to the respective index of the output vector. 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 operation may introduce additional irregularities to the access pattern, as well as change the thread workloads. Our work is aimed towards solving such problems.

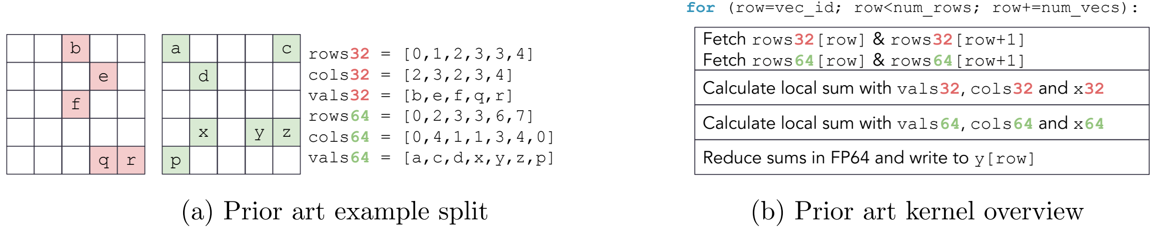
Chapter 3

MIXED-PRECISION SPMV METHODS

The prior art [Ahmad et al., 2019] introduced a precision selection method for a given nonzero value as follows: if the value is in range $(-r, r)$ for some r then it is stored in FP32, otherwise FP64; where a smaller r yields a lesser error. The idea here is based on the definitions of the IEEE 754 floating-point standard [IEEE, 2019], which defines a floating-point value n to be represented as $(-1)^s \times 2^{e-b} \times (1.m)$ where s is the sign bit, e is the exponent and m is the mantissa. An added bias b serves the purpose of negative exponents, for example FP64 is shown as $(-1)^s \times 2^{e-1023} \times (1.m)$. The number of bits reserved for s, e, m are $1, 8, 23$ for FP32 and $1, 11, 52$ for FP64, respectively.

The same number of mantissa bits are used to represent a number in the range $[2^e, 2^{e+1})$, regardless of e. With this, for a given exponent e, a precision is defined in that interval as $p_e = (2^{e+1} - 2^e)/2^\mu$, where μ is the number of mantissa bits. A smaller p_e indicates better precision since there are more values that can be represented with the given mantissa bits at that specific range. In smaller ranges, the loss in μ can be tolerated (i.e., casting FP64 to FP32).

The prior work had used a fixed range of $r = 1$ that applied to all matrices. Although a fixed and small range such as 1 would incur less absolute error [Ahmad et al., 2019], it can cause all values to be stored in FP32 for matrices with small values or for matrices that are scaled. Instead, in this work we implement a *matrix-specific range* that is calculated for each matrix prior to computations. For a given matrix with set of nonzero values $\mathcal{V}$, we calculate the range r as $r = f \times (\sum_{v \in \mathcal{V}} |v|)/|\mathcal{V}|$ where f is a shrinking factor chosen by user. The role of f is to adjust the mean, where smaller values may incur less errors but also choose a smaller amount of singles.



(a) Prior art example split

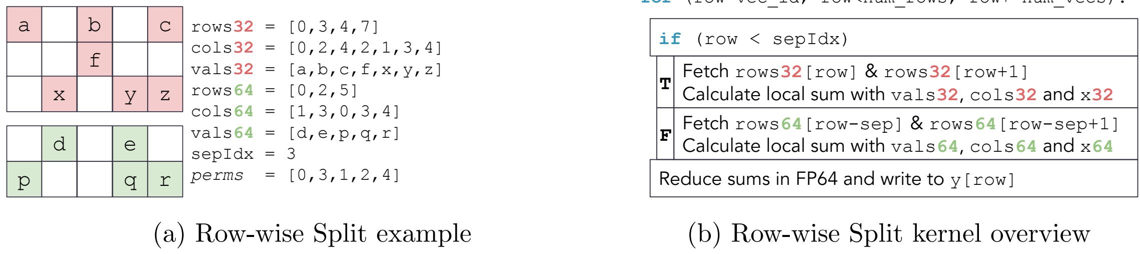
(b) Prior art kernel overview

Figure 3.1: An example and kernel overview for prior art (entry-wise split) [Ahmad et al., 2019]

3.1 Prior Art: Entry-wise Split

Assuming that a range has been chosen and the precisions are determined for each nonzero value, Ahmad et al. [Ahmad et al., 2019] create one CSR matrix with the FP64 values and another with the FP32 values, both of these matrices have the same number of rows and columns as the original. They had used a fixed range of $(-1, 1)$ for all matrices. An example of matrix split is shown in Figure 3.1a, the suffixes 32 and 64 indicate that the pointer belongs to FP32 and FP64 CSR matrices.

In the prior art's CSR-Vector kernel implementation (Figure 3.1b), 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. However, this implementation may incur load-balancing problems, both within a thread group and among the thread groups. To demonstrate the first problem this, let us describe how the first row is processed by a thread group in Figure 3.1a. If this row was not split, it would have three elements a, b, c processed by three consecutive threads. When the row is split, the first thread will process b from the FP32 matrix and a from the FP64 matrix, the second thread will only process c from the FP64 matrix and the third thread will be idle. This side-effect can be construed as if the matrix density (ratio of the number of nonzeros to matrix size) is getting smaller because now there are 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.



(a) Row-wise Split example (b) Row-wise Split kernel overview

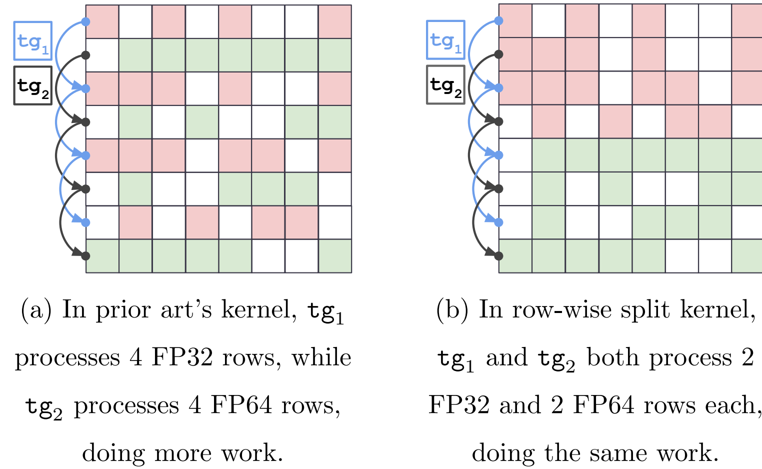
Figure 3.2: An example and kernel overview for Row-wise Split method

Secondly, the thread groups may have different loads based on the split, such as in a case where a thread group keeps processing rows with mostly FP64 values, whereas another processes rows with mostly FP32 values. Such a case is demonstrated in Figure 3.3a. As shown in the figure, thread groups have drastically different loads to process.

3.2 Row-wise Split

Although its kernel is based on CSR, the prior art does not necessarily take advantage of the storage format. This is expected since entry-wise precision selection happens at the granularity of each nonzero value, but such a selection provides the leeway of being extendable to respect other storage formats. This is where our new row-wise methods come forth with regards to the CSR format: we extend the prior art to choose a precision for each row as follows: a given row is chosen to be in FP32 if at least p percentage of its values are in some range $(-r, r)$. However, all nonzeros in the row must be within the range of FP32 (i.e., `FLT_MAX` in C language). If these conditions are met, the row is stored in FP32.

After precisions are decided per row, we permute the rows such that FP32 rows are clustered on the top, and FP64 rows are clustered below it. For instance, in Figure 3.2a, the rows are mapped from $0, 1, 2, 3, 4$ to $0, 3, 1, 2, 4$ w.r.t the matrix in Figure 2.1a; this permutation is indicated with the `perms` array. We then create a CSR matrix by separating the top and bottom by their precisions. The row index where this separation happens is denoted as `sepIdx` and is used to differentiate



(a) In prior art's kernel, $\mathtt{tg}_1$ processes 4 FP32 rows, while $\mathtt{tg}_2$ processes 4 FP64 rows, doing more work.

(b) In row-wise split kernel, $\mathtt{tg}_1$ and $\mathtt{tg}_2$ both process 2 FP32 and 2 FP64 rows each, doing the same work.

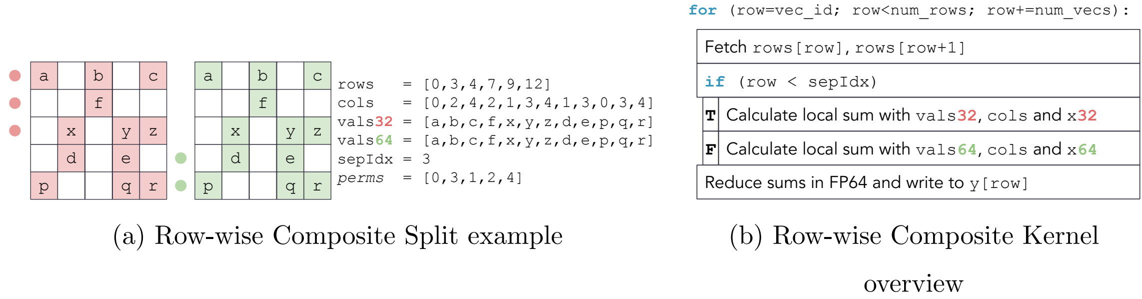
Figure 3.3: Workload distribution among thread groups with mixed-precision SpMV, demonstrated with 2 thread groups ($\mathtt{tg}_1$ and $\mathtt{tg}_2$) processing an 8×8 matrix.

the precision type based on the row index alone, shown in Figure 3.2b. Within the kernel, a thread group first checks the precision by comparing its row to the separation index, and then fetches the respective row pointers to proceed with the dot product. Unlike prior art, row-wise split method guarantees that there will be no empty rows processed during SpMV, and it further permutes existing empty rows to be clustered at the bottom where they can be ignored during SpMV. Thus, it saves redundant global accesses to the row pointer in case of empty rows.

With the row-wise split method, the matrix density is kept constant, and the order of elements in a row stays the same, effectively solving the load-balancing problem within a thread group. Moreover, row permutation ensures load-balance among thread groups as demonstrated in Figure 3.3b. Consecutive rows will have the same precision and thus consecutive thread groups will have a similar amount of work, with the only exception being near the separation index.

3.3 Row-wise Composite: A Multi-Precision Compliant Row-wise Split

Multi-precision and mixed-precision techniques are occasionally used interchangeably [Abdelfattah et al., 2021], though we describe them as follows regarding SpMV: mixed-precision SpMV is when a single SpMV operation uses more than one preci-



(a) Row-wise Composite Split example

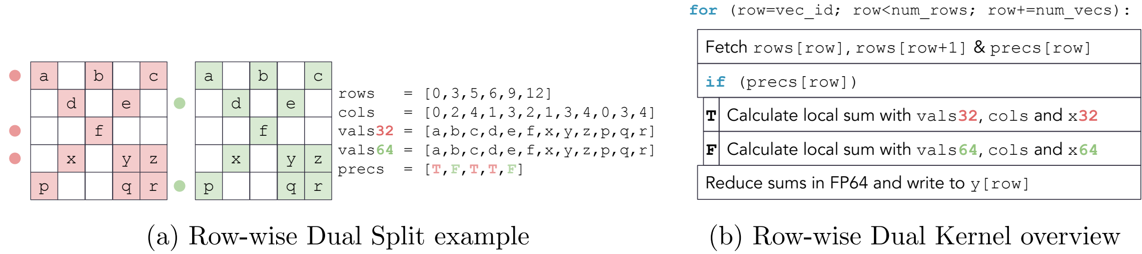
(b) Row-wise Composite Kernel overview

Figure 3.4: An example and kernel overview for Row-wise Composite Split method

sion; whereas multi-precision SpMV is when FP64, FP32 and mixed-precision SpMV may be used throughout an entire application at different levels. Recently, a multi-precision GMRES method using FP32 and FP64 was described by Loe et al. [Loe et al., 2021], and this method required both the original FP64 matrix and its FP32 copy to be stored at the same time. Note that this only requires creating the FP32 value pointer and its matrix, as the row and column pointers are equivalent for both.

Suppose one is to extend this multi-precision method by incorporating mixed-precision SpMV in between FP32 and FP64 iterations. With the methods described so far, two separate CSR matrices need to be created which have their own row, column and value pointers, differing from those of FP64 and FP32 matrices. Consequently, these mixed-precision splits will require their own memory allocations and transfers in case of a GPU execution in addition to 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. In this section, we describe two methods that can utilize the row-wise mixed-precision SpMV for multi-precision execution in the presence of FP64 and FP32 matrices.

In row-wise split, the matrix is permuted and then split it into two separate CSR matrices. Now, instead of creating two CSR matrices we will just create a copy of the entire permuted value pointer in FP32. Figure 3.4a shows 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 differentiate



(a) Row-wise Dual Split example

(b) Row-wise Dual Kernel overview

Figure 3.5: An example and kernel overview for Row-wise Dual Split method

the precision of a row, shown in Figure 3.4b. We denote this method as *row-wise composite*.

The row-wise composite method can be even more storage efficient if only FP32 values required during the mixed-precision step are needed (i.e. the values above the separation index). This is simply done by truncating the unused portion of the memory via the `malloc` and `memcpy` size parameters for the FP32 value pointer. Though the row-wise split methodology is agnostic to the order of precisions in the permuted matrix, for this type of extra optimizations keeping FP32 at the top of the permuted matrix is a requirement.

3.4 Row-wise Dual: Multi-Precision Compliance without Row Permutations

We have also implemented a more straightforward row-wise split method where instead of permuting the matrix and using a separation index, an auxiliary Boolean array `precs` is used to store the precision information of each row (true and false indicate FP32 and FP64, respectively). We denote this as *row-wise dual*, demonstrated in Figure 3.5a. For computations with FP64 or FP32 values per se, choosing respective value pointer and using the same row and column pointers for both would suffice; whereas for a mixed-precision computation the auxiliary array is utilized to infer which value pointer is needed for a given row, as shown in Figure 3.5b. The advantage of this implementation choice is to have a row-wise method without necessarily permuting the matrix. Note that row-wise dual still suffers from load-balance

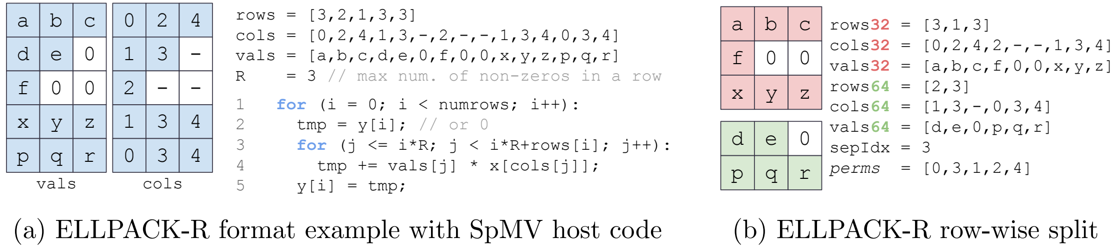
Table 3.1: Memory Capacity Requirements. M: number of rows, V: number of nonzeros, V_{64}: number of nonzeros stored in FP64.

Storage Format	Storage Cost (Bytes)
CSR FP64	$4M + 12V + 4$
CSR FP32	$4M + 8V + 4$
Prior Art	$8M + 8V + 4V_{64} + 8$
Row-wise Split	$4M + 8V + 4V_{64} + 12$
Row-wise Composite	$4M + 16V + 8$
Row-wise Dual	$5M + 16V + 4$

among thread groups, but not within a thread-group.

In Table 3.1 we present the total storage cost of each described mixed-precision split in bytes. Let us denote V as the number of non-zero values, and M as the number of rows. The CSR FP64 has a value pointer of V FP64 values, and a column index pointer of V integers. For the row pointers, $M + 1$ integers are used. We take the integer type to be 4 bytes, as is the case for most systems, and FP64 is 8 bytes by definition. Summing these up, we have $4(M + 1) + 4V + 8V = 4M + 12V + 4$ for the CSR FP64 storage. The CSR FP32 only differs by $4V$, which is due to FP32 being 4 bytes and saving $4V$ bytes from the value pointer array. The prior art splits the input in two matrices, but the matrix dimensions stay the same. This means that now the row pointers take $2 \times 4(M + 1)$ bytes. Let V_{32} and V_{64} denote the number of non-zeros stored in FP32 and FP64 respectively. Then, the value and column index storage will be $4V_{32} + 4V_{32}$ and $4V_{64} + 8V_{64}$ for each. Summing these up, we get $8V + 4V_{64}$, as $V = V_{32} + V_{64}$. This is also the case in row-wise split; however, there we keep the total number of rows constant, and thus only $4M$ is used for the row pointer. An extra 4 bytes is used to store the separation index.

Considering the multi-precision case, the FP32 and FP64 CSR matrices together (with common row and column pointers) will cost $4M + 16V + 4$ bytes. Row-wise composite uses only an extra 4 bytes for the separation index on top of this storage. Row-wise dual however, uses an extra M bytes instead, to store the precision information of each row in Boolean type. The bandwidth requirements for the methods



(a) ELLPACK-R format example with SpMV host code (b) ELLPACK-R row-wise split

Figure 3.6: ELLPACK-R format example and row-wise split example

above are equal to their storage costs, except for the multi-precision compliant methods (row-wise composite and row-wise dual). For them during the mixed-precision SpMV step, the memory accesses will be equivalent to that of row-wise split, with row-wise dual accessing an additional M bytes.

3.5 *Extending to ELLPACK-R*

Our methodology so far has been described for CSR format; notwithstanding, we follow the same approach to obtain mixed-precision SpMV for ELLPACK-R [Vázquez et al., 2011] format. The value and column index arrays had a size equal to number of non-zeros in CSR format, but here the size is $M \times R$, where M is the number of rows and R is the maximum number of non-zeros in a row. This enables us to index a row i just with $R \times i$. The row pointers in CSR then becomes row lengths for ELLPACK-R, where index i gives the number of non-zeros in row i. Consequently, the SpMV kernels are almost identical.

Provided that the number of non-zeros in a row does not vary dramatically throughout the matrix, ELLPACK-R provides better memory coalescing and is more efficient that CSR format. However, if a row has a lot more non-zeros compared to the others, this format 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, shown in Figure 3.6b. For brevity, we omit the examples of other splits for ELLPACK-R.

Chapter 4

CASE STUDIES

4.1 Case Study: Jacobi Method

Jacobi method is a well-known iterative method that can be used to solve linear systems [Pratapa et al., 2016, Gunawardena et al., 1991], and it is most widely used as a preconditioner within other iterative solvers (e.g., Krylov subspace methods) to accelerate their convergence [Turkel, 1999, Chan et al., 2001, Lee et al., 2002]. Using Jacobi as a preconditioner is appealing since its structure allows utilizing high parallelism in contrast to its counterparts such as ILU and SSOR preconditioners [Saad, 2003]. Jacobi aims to obtain an approximate solution for a given sparse linear system of equations $Ax = b$ where $A \in \mathbb{R}^{N \times N}$ is a sparse nonsingular matrix, $x \in \mathbb{R}^N$ is the unknown and $b \in \mathbb{R}^N$ is the right-hand side vector. Considering A is decomposed as $A = D + R$ where D is the diagonal component and R consists of the remaining off-diagonal entries, Jacobi method iteratively solves

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \tag{4.1}$$

where $x^{(k)}$ denotes the solution at the kth iteration. Note here that the calculations regarding D^{-1} are straightforward since D consists of only the diagonal entries. Therefore the SpMV operation in Jacobi ($Rx^{(k)}$), which is required to be solved at each iteration, dominates the runtime and constitutes the main bottleneck in the layout. We propose to utilize our mixed-precision implementation of SpMV to obtain a mixed-precision Jacobi which achieves speedup with respect to the conventional double-precision 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

Algorithm 1 Mixed-Precision Jacobi Method to solve $Ax = b$.

Require: Diagonal (D), remaining (R) components, and the splitting $R = R_{32} + R_{64}$

1: Choose an initial guess $x^{(0)}$
2: **for** $k = 1, 2, ...,$ `ITERS` **do**
3: $\bar{x} \leftarrow R_{64} \times x_{64}^{(k-1)} + R_{32} \times x_{32}^{(k-1)}$ SpMV (Mixed)
4: $\bar{x} \leftarrow D^{-1}(b - \bar{x})$ Jacobi Iteration (FP64)
5: $x_{64}^{(k)} \leftarrow \bar{x}$ Solution Update (FP64)
6: $x_{32}^{(k)} \leftarrow$ `float`$(\bar{x})$ Solution Update (FP32)
7: **end for**

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), that is $d_i = D_{i,i}$ for $i = 1, \ldots, N$.

We implement a single CUDA kernel for the Jacobi iteration and Solution Updates (lines 4,5,6) 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. Furthermore, all computations here take place on the GPU and thus 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 Jacobi. For example in a *2-step* Jacobi method, the solver can start with mixed-precision and move to double precision after a number of iterations. Similarly, in a *3-step* method, a number of iterations can be done in FP32, then the solver can be upgraded to use mixed-precision, and the final set of iterations can be done in FP64.

Algorithm 2 Mixed-Precision Cardiac Modeling

Require: Diagonal (D), remaining (R) components, and the splitting $R = R_{32} + R_{64}$

Require: Initial guess x_0 read from disk.

1: **for** $k = 1, 2, ..., \texttt{ITERS}$ **do**

2: $\bar{x} \leftarrow R_{64} \times x_{64}^{(k-1)} + R_{32} \times x_{32}^{(k-1)}$ SpMV (Mixed)

3: $\bar{x} \leftarrow \bar{x} + \bar{x} \odot D$ Add Diagonal (FP64)

4: $x_{64}^{(k)} \leftarrow \bar{x}$ Solution Update (FP64)

5: $x_{32}^{(k)} \leftarrow \texttt{float}(\bar{x})$ Solution Update (FP32)

6: **end for**

4.2 Case Study: Cardiac Modeling

Cardiac modeling is an application of solving diffusion equations with finite volumes, where the computations for a single time-step can be shown as a matrix multiplication operation [Langguth et al., 2015] in the form:

$$x^{(i)} = A \times x^{(i-1)} \tag{4.2}$$

. The mixed precision SpMV is utilized in a similar fashion to the Jacobi, where a solution time-step becomes

$$\bar{x} = A_{64} \times x_{64}^{(i)} + A_{32} \times x_{32}^{(i)} \tag{4.3}$$

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 is issued after the SpMV must update both solutions.

Furthermore, we have 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, if not all, of 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; the off-diagonal values are kept in the matrix similar to what is done in Jacobi. 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. This way, the range selection is more fair and the large diagonals are kept in FP64. Our proposed mixed-precision cardiac modeling is thus as described in Algorithm 2. In a fashion similar to multi-precision Jacobi described priorly, the *2-step* Cardiac modeling starts in mixed-precision and moves to double precision after a number of iterations. Similarly, in a *3-step* method: a number of iterations are done in FP32, then in mixed-precision, and finally in FP64.

Chapter 5

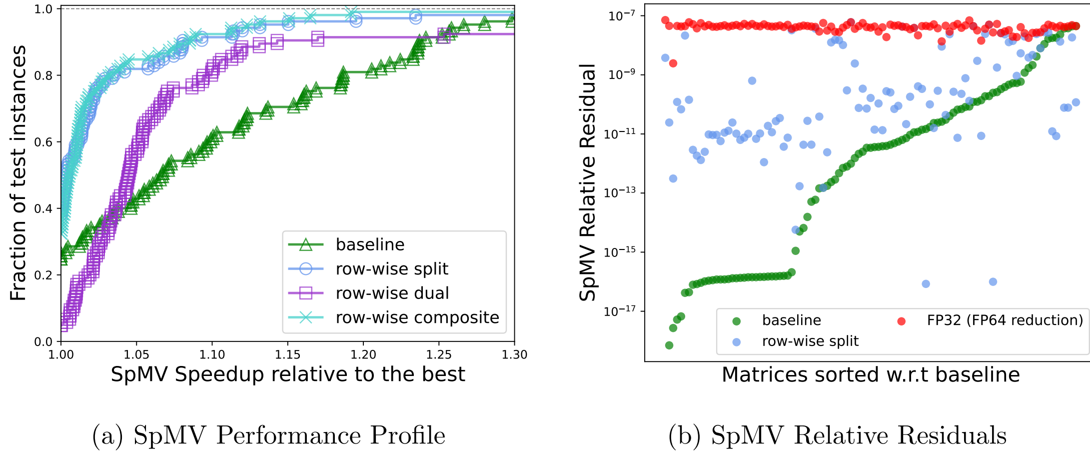
EVALUATIONS

All evaluations were conducted on a single node with an NVIDIA Tesla V100 GPU. CUDA 11.2 and GCC 9.3.0 with `-O3` optimization is used to compile our program. GFORTRAN 9.3.0 has also been used to build the HSL MC64 v2.3.1 [HSL, nd] static library that is used prior to the Jacobi solver for permutation and scaling. 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. As noted in [Ahmad et al., 2019], if this is not done then the CUDA runtime will cast the computations to be FP64, therefore preventing any performance improvements from FP32 computations. Similar to the prior mixed-precision SpMV research [Grigoraş et al., 2016, Ahmad et al., 2019], 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 [Clark et al., 2010].

We run 200 warm-up iterations of FP64 SpMV for each matrix in each experiment. The runtime measurements are from the beginning until the end of iterations; data transfer to and from GPU are not included. The matrix splitting and permuting times are also excluded from the runtime calculations, as this is a one-time cost that is done prior to the computations. Throughout this section, our *baseline* method is the prior art (entry-wise split) with $p = 100$, and its range value r changes with respect to the application.

Our row-wise methods use the proposed adaptive range based on absolute mean with $f = 0.1$ for SpMV and Jacobi method, and $f = 2$ for Cardiac modeling. The reason behind increased factor for Cardiac modeling is that the matrix values are very close to each-other, and shrinking the calculated range often results in all values to be outside the range. For all methods, the percentage value p is 99, i.e.,



(a) SpMV Performance Profile

(b) SpMV Relative Residuals

Figure 5.1: Speedup performance profile and relative residual plot for SpMV (CSR)

in a row 1% out-of-range values are tolerated. These values are chosen empirically. In particular, $p = 100$ turned out to be too strict, allowing most of the row to be kept in FP64, and lower p values such as 90 tolerated too many values, resulting in greater errors.

5.1 SpMV Results

For SpMV, we consider the $y \leftarrow y + Ax$ operation where the initial y is all zeros, and x has values that are uniformly random in the range $(-5, 5)$. The experiments are conducted on an extensive set of real-valued matrices from the SuiteSparse Matrix Collection [Davis and Hu, 2011], where the matrices have between 100,000 and 40,000,000 nonzeros. The range value of baseline method is $r = 1$, as is done in [Ahmad et al., 2019]. For reliable time measurements, we filter out matrices whose FP64 SpMV runtime is less than 100 milliseconds. Furthermore, we ignore matrices where the row-wise split results in less than 10% of nonzeros to be stored in FP32, as the speedup effect will not be considerable with a low percentage of FP32 values. As such, 105 matrices (out of 2,794) have been used in the evaluations of mixed-precision SpMV methods.

Figure 5.1a 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 [Dolan and Moré, 2002] 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. In a performance profile, the method closest to the top left corner is interpreted as the best-performing one.

As shown in Figure 5.1a, our proposed row-wise split and row-wise composite methods are performing the best. Due to the close performance of these two, and the fact that row-wise composite is designed for multi-precision algorithms, the further mixed-precision SpMV experiments are conducted with the row-wise split method. In this figure and hereafter, *baseline* refers to the prior art with range $(-1, 1)$ [Ahmad et al., 2019]. Note that for a fraction of instances, row-wise dual is even slower than baseline. This can be attributed to the fact that memory access pattern of row-wise dual may be more irregular compared to the baseline depending on the split. Consecutive warps may access the FP32 and FP64 matrices in an alternating fashion, having a negative impact on coalescing; especially when the number of elements in a row is small.

Figure 5.1b shows the plot for relative residuals of SpMV with respect to FP64 for the same 105 matrices sorted with respect to the baseline method's residuals. Here, we refer the relative residual as $||y' - y_{64}||_2/||y_{64}||_2$ 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, the baseline method often has small residuals. Nevertheless, there are numerous instances where the row-wise split method 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, we find our residuals to be acceptable. FP32 SpMV with FP64 reduction has the highest residual in all cases as expected.

In Table 5.1, we present the average speedup and relative residuals grouped into

Table 5.1: Average speedup of different precision SpMV methods with respect to the FP64 SpMV implementation. The matrices are grouped according to their density $\lambda = NNZ/(M{\times}N){\times}10{,}000$. Here, row-split refers to row-wise split; and FP32 refers to FP32 SpMV with FP64 sum reduction.

density (λ) in 10,000	matrix count	speedup w.r.t FP64			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

five different matrix density categories of 105 matrices.

All average values here are the geometric mean. For the row-wise split method, a correlation between speedup and density is observed. Though its reason is not immediately obvious, the matrices that are less dense may have lower speedup as the irregular access to the dense vector becomes more pronounced. Its access pattern is defined by the column indices, and that is not within the scope of mixed-precision SpMV nor our row permutations. Combined with the more reliable performance of our method, we notice that the speedup increases together with the density.

In Table 5.1, it should further be noted that the FP32 with double sum reduction serves as an upper bound of speedup and relative residual for each group. FP32 achieves up to 26% improvement with 16% on average. The table also shows that the baseline has no speedup or insignificant speedup. We have also experimented on 2,794 real-valued matrices from SuiteSparse on which the experiments of [Ahmad et al., 2019] 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 [Ahmad et al., 2019]. We chose to report detailed experimental results for 105 matrices mainly because of their large sizes.

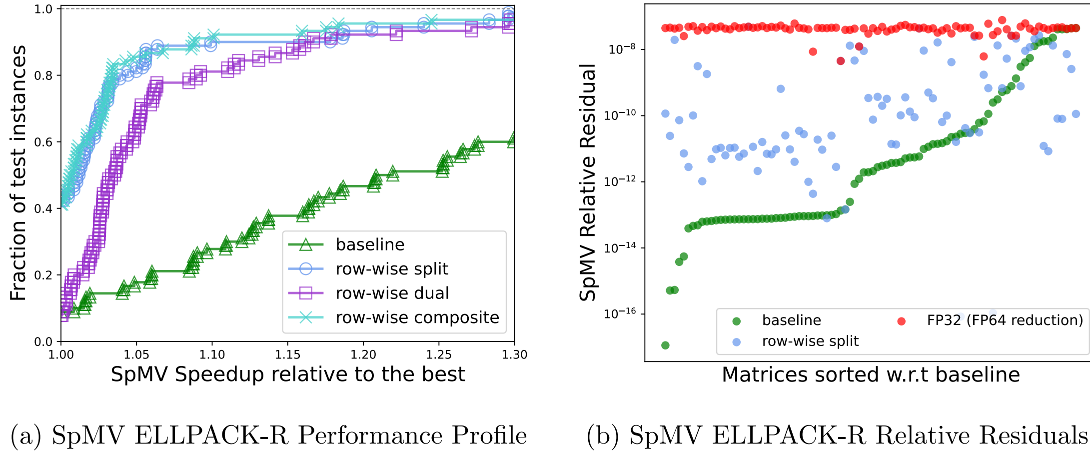
To demonstrate the effect of load-balancing, we focus on 5 matrices where the percentage of nonzeros chosen in FP32 are close between baseline and row-wise

Table 5.2: Warp execution efficiency profiled by `nvprof` for 5 matrices where the percentage of nonzeros stored in FP32 (shown as FP32%) are close between baseline and row-wise split (referred to as row-split in the table).

matrix	avg nnz	FP32%		speedup		warp exec. efficiency	
	per row	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

split method. Using the NVIDIA profiler `nvprof`, we profile one iteration of mixed-precision SpMV operation for both, and focus on 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*". As we have mentioned in chapter 2, threads within a warp can only execute one type of instruction at a time. If some threads are idle while others are processing elements for the dot product, we would see that reflect as a worse warp execution efficiency.

Our kernel profiling results are given in Table 5.2. 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-wise split has higher warp execution efficiency and speedup, 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. Moreover, for the barrier2-4 and barrier2-9 matrices the baseline method has significantly lower warp execution efficiency compared to its performance in the other three matrices; whereas row-wise split has an overall similar performance. This can be explained from the fact that both barrier2-4 and barrier2-9 has an average 33.7 nonzeros per row. With the baseline method, these rows will be split into smaller rows, which results in many rows with less number of



(a) SpMV ELLPACK-R Performance Profile (b) SpMV ELLPACK-R Relative Residuals

Figure 5.2: Speedup performance profile and relative residual plot for SpMV (ELLPACK-R)

elements that the warp size. This may cause additional inefficiency within a warp, as a warp would now have to work on different rows.

5.1.1 ELLPACK-R Results

When we use ELLPACK-R for the chosen 105 matrices, 15 matrices of these become out-of-memory, 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 Figure 5.2a, and the residuals are given in Figure 5.2b. The results are similar to what is observed for CSR format, with the only difference being that the gap between baseline and our proposed methods is larger in the performance profile for ELLPACK-R. Moreover, the average and maximum speedup for ELLPACK-R is higher than that of CSR, with 1.07x and 1.74x respectively.

5.2 Jacobi Method Results

We consider large real-valued sparse square matrices with no explicit zeros, no empty values in the diagonal, or is not made of a diagonal only. We applied HSL

MC64 [HSL, nd] permutation and scaling in order to increase the chance of encountering diagonal-dominance, which is a sufficient condition for Jacobi to converge [Bagnara, 1995]. The aim of permuting is to maximize the product of the diagonal entries, whereas the aim of scaling is to make the absolute value of diagonal entries one and the absolute value of off-diagonal entries less than or equal to one. Due to this scaling, the baseline method's range can not be $r = 1$, which would choose everything in FP32; instead, we use $r = 0.01$.

For the Jacobi method, first $x^* \leftarrow [1/N, 2/N, \ldots, 1]^T$ is set as the desired solution vector, and the right-handside is calculated as $y \leftarrow Ax^*$ with an FP64 SpMV. We then set the initial guess as $x \leftarrow [0, 0, \ldots, 0]^T$ and run the Jacobi method with different mixed-precision SpMV within to solve $Ax = y$. Since Jacobi is often used as a preconditioner in iterative solvers, and to provide an explicit comparison in terms of accuracy, we run Jacobi for a fixed number of (2000) iterations. With a computed solution x', the relative residual is calculated as $(||y - Ax'||_2/||y||_2)$. Note that the residual is affected by the precision selection, but not from the split that comes afterwards; the reason is that the split itself does not change the accuracy of the computations. For our evaluations, we have a selection of 8 matrices that concisely demonstrate various cases of differing speedups and residuals.

In Figure 5.3 we show the speedups of baseline and our methods with respect to the FP64 Jacobi method. In all cases, 3-step multi-precision Jacobi is the fastest. 2-step multi-precision Jacobi also has speedup, though it is slower than 3-step in all cases. For the instances where 1-step is slower than 2-step, the higher runtime of the former can be attributed to the additional cost of updating the FP32 copy of the solution vector, and a low number of FP32 values used within a split. With all matrices, the baseline is the slowest except for *ohne*, where the baseline keeps considerably higher percentage of FP32 values.

Table 5.3 shows the relative residuals for baseline and our methods, as well as Jacobi methods that use FP64 and FP32 SpMV (with FP64 reduction). Although the 1-step method has relative residuals higher than that of FP64, the multi-precision methods are able to tolerate the accumulated errors in the prior less-precise steps within, and consequently their relative residuals are close to that of FP64 Jacobi.

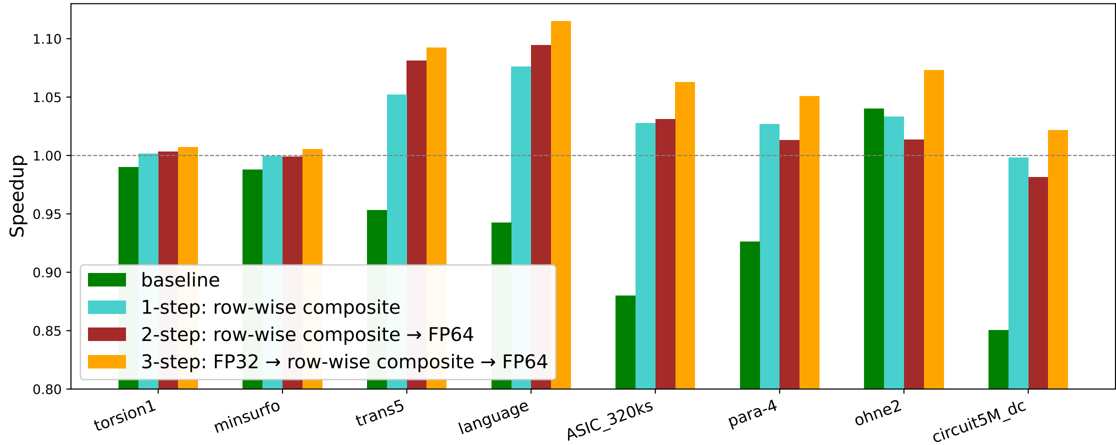


Figure 5.3: Jacobi Speedups with respect to FP64 Jacobi method

Table 5.3: 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

This is in line with our expectations that lower precision steps can be corrected by the higher precision steps that follow. As such, the speedup gained from the lower precision steps yield an overall runtime improvement.

5.3 *Cardiac Modeling Results*

We have a set of 6 matrices, which are generated meshes from a dataset [Martinez-Navarro et al., 2019]; each having more than 100,000 non-zeros. With each matrix, an initial dense solution vector is also provided. Note that for a fair comparison, 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.

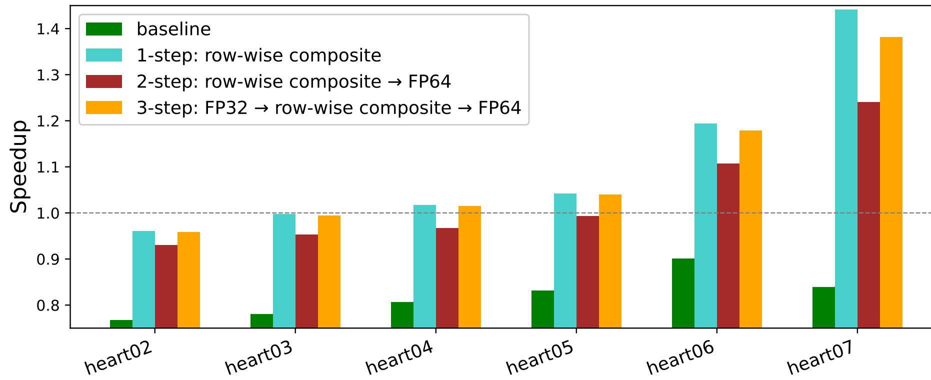


Figure 5.4: Cardiac Speedups with respect to FP64 Cardiac method

Table 5.4: Relative residuals of Cardiac modeling after 8,000 iterations.

matrix	FP32	baseline	1-step	2-step	3-step
heart02	1.68e-08	1.55e-09	1.16e-08	5.89e-09	9.26e-09
heart03	1.99e-08	2.67e-10	1.64e-08	1.13e-08	1.37e-08
heart04	1.98e-08	1.51e-10	1.68e-08	1.27e-08	1.39e-08
heart05	1.98e-08	6.96e-11	1.69e-08	1.36e-08	1.40e-08
heart06	1.97e-08	2.44e-11	1.70e-08	1.39e-08	1.40e-08
heart07	3.06e-08	1.35e-11	2.89e-08	2.73e-08	2.73e-08

All off-diagonals for this dataset are less than 1, so again the baseline range $r = 1$ is infeasible, instead we use $1e - 5$ which is around the absolute mean value for all matrices in the dataset.

In Figure 5.4 we show the speedups of baseline and our methods with respect to the FP64 Cardiac modeling method for the 6 matrices, with increasing number of non-zeros from left to right. Despite the extra copy kernel overhead of our mixed-precision methods, we observe an average speedup of x1.11 and up to x1.40. Furthermore, speedup increases with the number of non-zeros, which may attributed to better cache utilization of lower precisions. Note that 1-step is faster than 2-step and 3-step, indicating that FP64 steps are slow enough to affect the entire execution time.

Table 5.4 demonstrates the relative residuals of our methods with respect to the result of FP64 Cardiac modeling. We do not see much difference among the

mixed and multi-precision methods as was the case for Jacobi. Here, instead the final solutions have an error of around the same magnitude. The baseline method, although has lesser relative residual, suffers from longer execution times. It can be inferred that for some applications, sticking to just 1-step may be better, since the relative residuals among it and multi-precision methods are close to each-other, and 1-step is faster overall.

Chapter 6

RELATED WORK

Scientific applications often benefit from high precision floating point arithmetic since it has the advantage of leading higher accuracy [Bailey, 2005]. On the other hand, it results in larger data movement 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 [Abdelfattah et al., 2020, Alvermann et al., 2019, Arioli and Duff, 2009, Blanchard et al., 2020, Emans and van der Meer, 2010, Haidar et al., 2018, Hogg and Scott, 2010]. For sparse linear algebra methods, which are mostly bandwidth-bound, mixed-precision algorithms are favored to reduce the communication volume and memory access [Abdelfattah et al., 2021].

Mixed-precision arithmetic is utilized for both direct and iterative methods with the aim of obtaining the accuracy as good as FP64 and cost as low as the FP32 arithmetic [Anzt et al., 2015a, Barrachina et al., 2009]. The usage of mixed-precision arithmetic is shown to improve the execution time and energy consumption of iterative solvers on graphics processors [Anzt et al., 2012, Anzt and Quintana-Ortí, 2014]. Buttari et.al. [Buttari et al., 2008] utilize mixed-precision iterative refinement for sparse direct and iterative solvers by performing the most expensive steps in half precision and the less important ones in double precision.

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

Anzt et. al. [Anzt et al., 2015a] implement an adaptive precision Jacobi iterative method by utilizing varying mantissa lengths. Then in [Anzt et al., 2019], Anzt et. al. propose a mixed-precision block-Jacobi preconditioner 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.

Ahmad et. al. [Ahmad et al., 2019] 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 they achieve some speedup for certain 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.

Grigoraş et al. [Grigoraş et al., 2016] propose an architecture and automated method to perform SpMV on FPGAs within the context of the Finite Element Method where the block diagonals have customisable precision. For this, they propose a matrix-vector multiplication unit (MVMU) for FPGAs.

Amestoy et al. [Amestoy et al., 2021] demonstrate mixed-precision low-rank approximations, and they use bfloat16, FP32 and FP64 to obtain a mixed-precision Singular Value Decomposition (SVD). They apply their findings to sparse matrices and obtain a mixed-precision block low-rank (BLR) representation, and BLR LU factorization.

Ooi et al. [Ooi et al., 2020] implement a mixed-precision Hierarchical Matrix (H-Matrix) vector multiplication using FP32 and FP64. H-Matrix is a dense matrix approximation with low-rank approximated submatrices. They further apply their mixed-precision method within a Krylov iterative solver.

Chapter 7

CONCLUSIONS

In this work, we have shown how selecting precision with respect to the storage format improves the performance of SpMV. Although the speedups are modest, considering the competitiveness of SpMV research, we believe even small amounts of speedup should be noted. Our methodology was 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). 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 multi-precision methods in 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 residuals.

FP16 (half-precision) is increasingly becoming prevalent among HPC, though initially it was utilized in deep learning. With this, it is a clear direction of future work to extend our methodology to accommodate lower-precisions. The row permutation techniques can permute more precision types with the same approach, and lower precisions can be decided with smaller ranges. In doing so, more specialized hardware with greater capabilities at lower precisions, such as NVIDIA's Tensor Cores, are to be considered in future implementations.

Although an orthogonal direction of optimization, extension of this work to use multiple GPUs 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.

BIBLIOGRAPHY

[Abdelfattah et al., 2021] Abdelfattah, A., Anzt, H., Boman, E. G., Carson, E., Cojean, T., Dongarra, J., Fox, A., Gates, M., Higham, N. J., Li, X. S., et al. (2021). A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *International Journal of High Performance Computing Applications*, 35(4):344–369.

[Abdelfattah et al., 2020] Abdelfattah, A., Tomov, S., and Dongarra, J. (2020). Investigating the benefit of FP16-enabled mixed-precision solvers for symmetric positive definite matrices using GPUs. In *Computational Science – ICCS 2020*, pages 237–250, Cham. Springer International Publishing.

[Ahmad et al., 2019] Ahmad, K., Sundar, H., and Hall, M. (2019). Data-driven mixed precision sparse matrix vector multiplication for GPUs. *ACM Trans. Archit. Code Optim.*, 16(4).

[AlAhmadi et al., 2020] AlAhmadi, S., Mohammed, T., Albeshri, A., Katib, I., and Mehmood, R. (2020). Performance analysis of sparse matrix-vector multiplication (SpMV) on graphics processing units (GPUs). *Electronics*, 9(10).

[Alvermann et al., 2019] Alvermann, A., Basermann, A., Bungartz, H.-J., Carbogno, C., Ernst, D., Fehske, H., Futamura, Y., Galgon, M., Hager, G., Huber, S., et al. (2019). Benefits from using mixed precision computations in the ELPA-AEO and ESSEX-II eigensolver projects. *Japan Journal of Industrial and Applied Mathematics*, 36(2):699–717.

[Amestoy et al., 2021] Amestoy, P., Boiteau, O., Buttari, A., Gerest, M., Jézéquel, F., L'Excellent, J.-Y., and Mary, T. (2021). Mixed Precision Low Rank Approximations and their Application to Block Low Rank LU Factorization. working paper or preprint.

[Anzt et al., 2012] Anzt, H., Castillo, M., Fernández, J. C., Heuveline, V., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. (2012). Optimization of power consumption in the iterative solution of sparse linear systems on graphics processors. *Computer Science-Research and Development*, 27(4):299–307.

[Anzt et al., 2019] Anzt, H., Dongarra, J., Flegar, G., Higham, N. J., and Quintana-Ortí, E. S. (2019). Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers. *Concurrency and Computation: Practice and Experience*, 31(6):e4460.

[Anzt et al., 2015a] Anzt, H., Dongarra, J., and Quintana-Ortí, E. S. (2015a). Adaptive precision solvers for sparse linear systems. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing*, pages 1–10.

[Anzt and Quintana-Ortí, 2014] Anzt, H. and Quintana-Ortí, E. (2014). Improving the energy efficiency of sparse linear system solvers on multicore and manycore systems. *Philosophical Trans. of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2018):20130279.

[Anzt et al., 2015b] Anzt, H., Tomov, S., Luszczek, P., Sawyer, W., and Dongarra, J. (2015b). Acceleration of GPU-based Krylov solvers via data transfer reduction. *The International Journal of High Performance Computing Applications*, 29(3):366–383.

[Arioli and Duff, 2009] Arioli, M. and Duff, I. S. (2009). Using FGMRES to obtain backward stability in mixed precision. *Electronic Transactions on Numerical Analysis*, 33:31–44.

[Baboulin et al., 2009] Baboulin, M., Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Langou, J., Luszczek, P., and Tomov, S. (2009). Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533.

[Bagnara, 1995] Bagnara, R. (1995). A unified proof for the convergence of Jacobi and Gauss–Seidel methods. *SIAM review*, 37(1):93–97.

[Bailey, 2005] Bailey, D. H. (2005). High-precision floating-point arithmetic in scientific computation. *Computing in science & engineering*, 7(3):54–61.

[Barrachina et al., 2009] Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., and Quintana-Ortí, G. (2009). Exploiting the capabilities of modern GPUs for dense matrix computations. *Concurrency and Computation: Practice and Experience*, 21(18):2457–2477.

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

[Bell and Garland, 2009] Bell, N. and Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA. ACM.

[Blanchard et al., 2020] Blanchard, P., Higham, N. J., Lopez, F., Mary, T., and Pranesh, S. (2020). Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores. *SIAM Journal on Scientific Computing*, 42(3):C124–C141.

[Boland and Constantinides, 2011] Boland, D. and Constantinides, G. A. (2011). Optimizing memory bandwidth use and performance for matrix-vector multiplication in iterative methods. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3).

[Bolz et al., 2003] Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. (2003). Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924.

[Buttari et al., 2008] Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P., and Tomov, S. (2008). Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4).

[Carson and Higham, 2018] Carson, E. and Higham, N. J. (2018). Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J Sci Comput*, 40(2):A817–A847.

[Cevahir et al., 2009] Cevahir, A., Nukada, A., and Matsuoka, S. (2009). Fast conjugate gradients with multiple GPUs. In *ICCS*.

[Chan et al., 2001] Chan, S., Phoon, K., and Lee, F. (2001). A modified Jacobi preconditioner for solving ill-conditioned biot's consolidation equations using symmetric quasi-minimal residual method. *Int J Numer Anal Methods Geomech*, 25(10):1001–1025.

[Clark et al., 2010] Clark, M., Babich, R., Barros, K., Brower, R., and Rebbi, C. (2010). Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications*, 181(9):1517–1528.

[Dalton et al., 2014] Dalton, S., Bell, N., Olson, L., and Garland, M. (2014). Cusp: Generic parallel algorithms for sparse matrix and graph computations. Version 0.5.0.

[Davis and Hu, 2011] Davis, T. A. and Hu, Y. (2011). The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1).

[Dolan and Moré, 2002] Dolan, E. D. and Moré, J. J. (2002). Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213.

[Emans and van der Meer, 2010] Emans, M. and van der Meer, A. (2010). Mixed-precision AMG as linear equation solver for definite systems. *Procedia Computer Science*, 1(1):175–183.

[Grigoraş et al., 2016] Grigoraş, P., Burovskiy, P., Luk, W., and Sherwin, S. (2016). Optimising Sparse Matrix Vector multiplication for large scale FEM problems on FPGA. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9.

[Gunawardena et al., 1991] Gunawardena, A. D., Jain, S., and Snyder, L. (1991). Modified iterative methods for consistent linear systems. *Linear Algebra and Its Applications*, 154:123–143.

[Haidar et al., 2018] Haidar, A., Tomov, S., Dongarra, J., and Higham, N. J. (2018). Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613. IEEE.

[He and Gao, 2016] He, G. and Gao, J. (2016). A Novel CSR-Based Sparse Matrix-Vector Multiplication on GPUs. *Mathematical Problems in Engineering*, 2016:8471283.

[Hogg and Scott, 2010] Hogg, J. D. and Scott, J. A. (2010). A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Transactions on Mathematical Software (TOMS)*, 37(2):1–24.

[HSL, nd] HSL (n.d.). A collection of Fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk/.

[IEEE, 2019] IEEE (2019). IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84.

[Kleinberg, 1999] Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632.

[Kreutzer et al., 2014] Kreutzer, M., Hager, G., Wellein, G., Fehske, H., and Bishop, A. R. (2014). A unified sparse matrix data format for efficient general sparse

matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423.

[Langguth et al., 2015] Langguth, J., Sourouri, M., Lines, G. T., Baden, S. B., and Cai, X. (2015). Scalable heterogeneous cpu-gpu computations for unstructured tetrahedral meshes. *IEEE Micro*, 35(4):6–15.

[Lee et al., 2002] Lee, F.-H., Phoon, K., Lim, K., and Chan, S. (2002). Performance of Jacobi preconditioning in Krylov subspace solution of finite element equations. *International journal for numerical and analytical methods in geomechanics*, 26(4):341–372.

[Liu and Vinter, 2015] Liu, W. and Vinter, B. (2015). CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. *Proceedings of the 29th ACM on International Conference on Supercomputing.*

[Liu and Schmidt, 2015] Liu, Y. and Schmidt, B. (2015). LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 82–89.

[Loe et al., 2021] Loe, J. A., Glusa, C. A., Yamazaki, I., Boman, E. G., and Rajamanickam, S. (2021). Experimental evaluation of multiprecision strategies for GMRES on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 469–478.

[Luebke et al., 2006] Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., and Buck, I. (2006). Gpgpu: General-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, page 208–es, New York, NY, USA. Association for Computing Machinery.

[Martinez-Navarro et al., 2019] Martinez-Navarro, H., Rodriguez, B., Bueno-Orovio, A., and Minchole, A. (2019). Repository for modelling acute myocardial ischemia: simulation scripts and torso-heart mesh.

[McCormick et al., 2020] McCormick, S. F., Benzaken, J., and Tamstorf, R. (2020). Algebraic error analysis for mixed-precision multigrid solvers. *ArXiv*, abs/2007.06614.

[Nickolls et al., 2008] Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53.

[NVIDIA, 2022] NVIDIA (2022). Cuda c++ programming guide.

[NVIDIA cuSPARSE, nd] NVIDIA cuSPARSE (n.d.). https://docs.nvidia.com/cuda/cusparse/index.html#cusparse-generic-function-spmv, (accessed on 29 January 2022).

[Ooi et al., 2020] Ooi, R., Iwashita, T., Fukaya, T., Ida, A., and Yokota, R. (2020). Effect of mixed precision computing on h-matrix vector multiplication in bem analysis. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2020, page 92–101, New York, NY, USA. Association for Computing Machinery.

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

[Pratapa et al., 2016] Pratapa, P. P., Suryanarayana, P., and Pask, J. E. (2016). Anderson acceleration of the Jacobi iterative method: An efficient alternative to Krylov methods for large, sparse linear systems. *Journal of Computational Physics*, 306:43–54.

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

[Tan et al., 2018] Tan, G., Liu, J., and Li, J. (2018). Design and implementation of adaptive SpMV library for multicore and many-core architecture. *ACM Trans. Math. Softw.*, 44(4).

[Turkel, 1999] Turkel, E. (1999). Preconditioning techniques in computational fluid dynamics. *Annual Review of Fluid Mechanics*, 31(1):385–416.

[Vázquez et al., 2011] Vázquez, F., Fernández, J. J., and Garzón, E. M. (2011). A new approach for sparse matrix vector product on nvidia gpus. *Concurr. Comput.: Pract. Exper.*, 23(8):815–826.

[Vuduc and Moon, 2005] Vuduc, R. W. and Moon, H.-J. (2005). Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proceedings of the First International Conference on High Performance Computing and Communications*, HPCC'05, page 807–816, Berlin, Heidelberg. Springer-Verlag.

[Zhao et al., 2018] Zhao, Y., Li, J., Liao, C., and Shen, X. (2018). Bridging the gap between deep learning and sparse matrix format selection. *SIGPLAN Not.*, 53(1):94–108.

[Zounon and Higham, 2020] Zounon, M. and Higham, N. J. (2020). Performance evaluation of mixed precision algorithms for solving sparse linear systems.