A UNIFIED FRAMEWORK FOR BENCHMARKING SPARSE MATRIX-VECTOR MULTIPLICATION METHODS

A Thesis

by

Erdem Sarılı

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
January 2019

A UNIFIED FRAMEWORK FOR BENCHMARKING SPARSE MATRIX-VECTOR MULTIPLICATION METHODS

Approved by:

Asst. Professor T. Barış Aktemur, Advisor
Department of Computer Science
Özyeğin University

Assoc. Professor Hasan Sözer
Department of Computer Science
Özyeğin University

Assoc. Professor Mehmet Aktaş
Department of Computer Engineering
Yıldız Technical University

Date Approved: 10 January 2019

To my beloved wife

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is an important sparse linear algebra kernel that has a wide range of application domains, including computational science, graph analytics, machine learning and many more. Due to its significance, numerous studies have been conducted and are still being proposed to improve the performance of SpMV. Most of the studies evaluate the performance of their method in a custom experimental environment, which weakens the reproducibility of the empirical results, and also makes it hard to compare the proposed method to a wide range of existing methods. In this study, we address this problem by introducing an easy-to-integrate benchmarking framework that is able to unify SpMV methods in a single experimental environment to obtain consistent evaluation results. As a proof-of-concept, we have integrated several state-of-the-art CPU and GPU-based SpMV methods in our framework. We make the framework available as an open-source software for the convenience of researchers.

ÖZETÇE

Seyrek matris-vektör çarpımı (SpMV) hesaplama bilimi, çizge analitiği ve makine öğrenmesi de dahil pek çok kullanım alanı olan önemli bir seyrek doğrusal cebir çekirdeğidir. Bu önemi sebebiyle pek çok çalışma SpMV performansını arttırmayı hedeflemektedir. Yapılan çalışmaların çoğu, önerdikleri metotların performansını özel deney ortamlarında değerlendirmektedir. Özel deney ortamlarının kullanılması, değerlendirme sonuçlarının tekrar edilebilirliğini olumsuz etkilemekte ve önerilen metotların mevcut metotlarla kıyaslanmasını zorlaştırmaktadır. Sunduğumuz bu çalışmada, tutarlı deney sonuçları elde etmek için SpMV metotlarını tek bir deney ortamında birleştiren, kolay entegre edilebilir bir performans değerlendirme çerçevesi oluşturarak problemi çözmeyi hedefledik. Bu kavramın bir kanıtı olarak, en gelişmiş CPU ve GPU tabanlı metotlarını sunduğumuz çerçeveye entegre ettik. Geliştirdiğimiz çerçeveyi açık kaynak bir yazılım olarak araştırmacıların kullanımına sunuyoruz.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Asst. Professor T. Barış Aktemur for all his valuable guidance. I consider myself lucky to have had a chance to work with him.

I would also like specially thank my dear spouse Ayşe Gülcemal for her limitless support and continuous encouragement during my studies. This accomplishment would not have been possible without her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is a fundamental sparse linear algebra operation that takes role in a wide range of science and engineering problems, where substantial amount of time is spent performing the SpMV operations. Hence, improving the performance of SpMV has been the concern of a large body of work (see Langr and Tvrdik [1], and Filippone et al. [2] for two recent surveys). Because the matrix operand of SpMV is sparse, it is stored in a sparse storage format to reduce the space requirements – a dense format causes extremely large space penalties. The downside of a sparse format is irregular memory access patterns, causing under-utilization of the peak performance available on the computer. For this reason, many studies aimed at improving SpMV's performance by attacking various aspects of execution limitations including efficient parallelization of SpMV execution (e.g. [3, 4]), maximization of cache reuse (e.g. [5, 6, 7, 8]), design of new sparse storage formats (e.g. [9, 10, 11]), and dynamic or static generation of SpMV programs optimized for a given set of situations (e.g. [12, 13, 14, 10]).

No matter what approach is used to improve the performance of SpMV, a study needs to provide experimental evaluation results to demonstrate the impact on efficiency. Although a general benchmarking structure is usually followed, to the best of our knowledge, there does not exist a common benchmarking framework that allows researchers to evaluate the performance of their new SpMV approach with respect to existing methods. Researchers manually craft their benchmarking software. This not only causes valuable time to be spent on rudimentary tasks, but also makes it difficult and error-prone to compare evaluation results coming from different sources. In this work, we propose a unified benchmarking framework that fills this gap in SpMV research. We developed this framework with the motivation and goal that

- researchers will be able to integrate their new SpMV methods conveniently.
- researchers will not have to spent time implementing the experimental setup.
- industrial-strength SpMV libraries such as Intel MKL [15], cuSPARSE [16], and CUSP [17] are available built-in as common baseline methods to enable comparability.
- researchers will be encouraged to provide their SpMV methods to others for reproducibility of the results.

We have examined several existing SpMV work whose benchmarking code is available. We developed our framework by considering the common benchmarking flow in these work, the execution stages that they profile, the timing procedures they use, the metrics they report, the SpMV method that they use as the baseline, etc. We aimed to benchmark both CPU and GPU based SpMV methods. We have integrated ViennaCL [18] (for both CPU and GPU), several methods from Yzelman [19, 6], and methods from Thundercat library [14] into our framework as proof-of-concept.

This paper is organized as follows: In Chapter 2 we provide background information about SpMV, followed by a discussion of related previous work in Chapter 3. We present our proposed approach in Chapter 4, and in Chapter 5 we explain how to integrate a new SpMV method into this framework. We demonstrate integrations in framework in Chapter 6. Finally, we give our conclusions in Chapter 7.

Our framework is publicly and freely available at the following URL:

```
https://github.com/ozusrl/spmv-benchmarking
```

CHAPTER II

BACKGROUND

SpMV takes two inputs, a sparse matrix A and a dense vector x, and accummulates their multiplication on an output vector y. The mathematical notation for this operation is $y \leftarrow y + Ax$. In a computer program, the sparse matrix A is stored using a sparse storage format. Probably the most basic of these formats is the coordinate (COO) format, also known as the triplet representation, where the nonzero elements of A are stored in a contiguous array, *values*. Two other arrays, named *rows* and *cols*, store the row and column indices of the elements in *values* such that $values[k] = A_{rows[k],cols[k]}$ for each k such that $0 \le k < NNZ$, where NNZ is the number of nonzero elements in A. Figure 1 shows a sparse matrix in COO format, and Figure 2 shows SpMV using the COO format.

$$A = \begin{array}{|c|c|c|c|} \hline 0 & 4.0 & 2.1 & 0 \\ \hline 0 & 0 & 1.3 & 0 \\ \hline 0 & 5.7 & 0 & 3.6 \\ \hline 6.8 & 0 & 0 & 0 \\ \hline \end{array}$$

(a) Matrix A.

	0	1	2	3	4	5
values :	4.0	2.1	1.3	5.7	3.6	6.8
rows :	0	0	1	2	2	3
cols :	1	2	2	1	3	0

(b) Matrix A in the COO format.

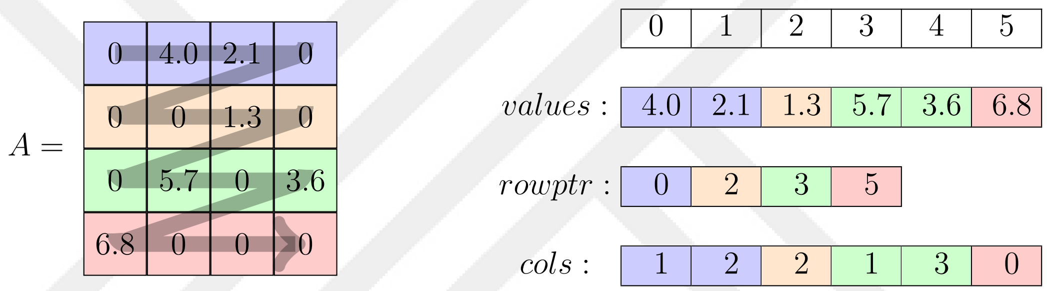
Figure 1: A sparse matrix in the COO format.

To take advantage of spatial locality for cache, matrix values can be ordered in row-major or column-major order. This causes many repetitions in the *rows*, or respectively *cols*, array that can be compressed. This leads to a compressed storage format such as Compressed Sparse Row (CSR) [20], also known as Compressed Row

```
input  : NNZ: number of nonzero elements in matrix
         values: nonzero elements of matrix
         rows: row indices of elements in values
         cols: column indices of elements in values
         x: input vector
output: y: output vector

for $i \leftarrow 0$ to $NNZ$ do
    $y[rows[i]] \leftarrow values[i] \times x[cols[i]] + y[rows[i]]$
end
```

Figure 2: SpMV using the COO format.



(a) Row major order on matrix A.

(b) Matrix A in the CSR format.

Figure 3: A sparse matrix in the CSR format .

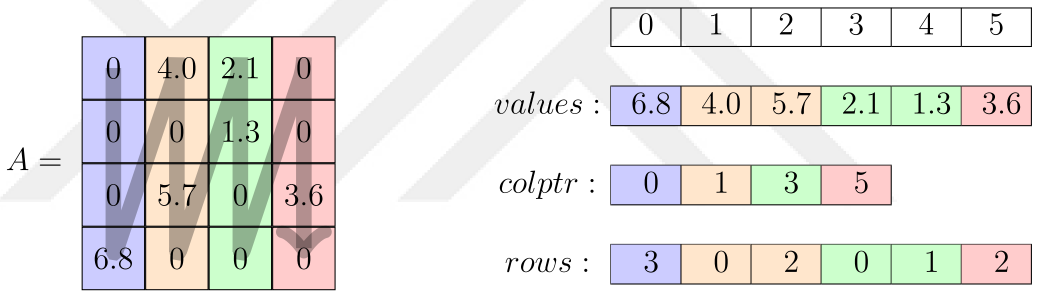
Storage (CRS). It stores the nonzero elements in row major order, again in a single-dimensional array, *values*. Similar to COO, another array, *cols*, stores the column indices. A third array, *rowptr*, stores for each row the index of the first nonzero element. In CSR, for each row i of the matrix A, we have $values[j] = A_{i,cols[j]}$ for each j such that $rowptr[i] \leq j < rowptr[i+1]$. Figure 3 demonstrates an example matrix in CSR format. Figure 4 shows SpMV using the CSR format.

Compressed Sparse Column (CSC), also known as Compressed Column Storage (CCS), is highly analogous to CSR format, exercising column major ordering for nonzero elements instead of row-major. Figure 5 shows an example matrix in the CSC format and Figure 6 outlines the algorithm of SpMV using the CSC format.

```
input  : M: number of rows in matrix
         values: nonzero elements of matrix in row major order
         rowptr: pointers to row beginnings on values
         cols: column indices of elements in values
         x: input vector
output: y: output vector

for $i \leftarrow 0$ to $M$ do
    for $j \leftarrow rowptr[i]$ to $rowptr[i+1]$ do
        $y[i] \leftarrow values[j] \times x[cols[j]] + y[i]$
    end
end
```

Figure 4: SpMV using the CSR format.



$$A = \begin{bmatrix} 0 & 4.0 & 2.1 & 0 \\ 0 & 0 & 1.3 & 0 \\ 0 & 5.7 & 0 & 3.6 \\ 6.8 & 0 & 0 & 0 \end{bmatrix}$$

	0	1	2	3	4	5
values :	6.8	4.0	5.7	2.1	1.3	3.6
colptr :	0	1	3	5		
rows :	3	0	2	0	1	2

(a) Column major order on matrix A.

(b) Matrix A in the CSC format.

Figure 5: A sparse matrix in the CSC format.

```
input  : N: number of columns in matrix
         values: nonzero elements of matrix in row major order
         colptr: pointers to column beginnings on values
         rows: row indices of elements in values
         x: input vector
output: y: output vector

for $j \leftarrow 0$ to $N$ do
    for $i \leftarrow colptr[j]$ to $colptr[j+1]$ do
        $y[rows[i]] \leftarrow values[i] \times x[j] + y[rows[i]]$
    end
end
```

Figure 6: SpMV using the CSC format.

CSR is arguably the defacto standard representation for sparse matrices. Several SpMV libraries accept matrix inputs in CSR format, including Intel MKL [15], cuSPARSE [16], CUSP [17], and ViennaCL [18]. For this reason, we opted in our framework for using CSR as the format of input matrices.

CHAPTER III

RELATED WORK

SpMV is an extensively studied problem. A complete overview of this problem is out of the scope of this work. For two recent surveys, see Langr and Tvrdik [1], and Filippone et al. [2]. In this section we give an overview of previous SpMV work with benchmarks, focusing more on those whose code is available online. In general, we notice that there are subtle differences that make SpMV evaluations incomparable to each other directly, such as

- Number and diversity of matrices included in the experiment
- Number of experiments done per input matrix
- Use of cache warm up iterations before experiments
- Reported performance measurement unit (GFlops, milliseconds, speed up ratio)
- Reported measurement sampling (best performing, mean, median)
- Mechanism used for measuring execution times (wall-clock, profiler)
- The performed operation ($y \leftarrow y + Ax$ vs. $y \leftarrow Ax$)
- Whether explicit zeros in the input matrix are discarded or not

Yzelman proposes an SpMV method named zz-crs, for partitioning an input matrix in such a way that the SpMV can be executed in a cache-friendly manner [19]. The study compares the proposed method with self implemented CSR and incremental CSR (ICSR) variants as well as the OSKI [5] library. 14 different matrices are used in experiments which are classified in two groups; the ones that benefit from cache reuse due to their size and structure, and the ones that do not. Experiments are carried on two different architectures, an Intel Q6600 CPU and a supercomputer named Huygens. Execution times are measured by wall-clock and normalized average of 1000 SpMV operations for each matrix are considered. This work reports the speedup ratios between the proposed method and others.

Yzelman and Bisseling perform another cache optimization study [6], where they introduce a new matrix storage format, Bi-directional Incremental Compressed Row Storage (BI-CRS) that uses Hilbert curve to order nonzero elements in matrix. A comparison is done between CSR, ICSR, and the proposed method BI-CRS. In this study, authors prefer to experiment with 9 matrices, for which the input/output vectors do not fit in the cache, so the cache reuse is not possible. Experiments are carried on two architectures, Intel q6600 and AMD 945e. Wall-clock is used to time the experiments; 100 SpMV executions are performed for each matrix and the average of those timings are reported in milliseconds.

Yzelman's SpMV benchmarking library is available online[1]. The library contains 15 sequential and 5 parallel SpMV schemes. We noticed that the CSR method performed a replacing SpMV operation (i.e. $y \leftarrow Ax$) while others performed accumulative (i.e. $y \leftarrow Ax + y$). We also noticed that CSR, McCSR, and SVM methods ignored explicit zeros when reading matrix data as input, while other methods did not. These inconsistencies jeopardize the reliability of performance comparison of these methods with respect to each other.

CSR5 [9] is a sparse matrix format designed for obtaining high throughput SpMV execution on different platforms with low conversion cost from CSR. The format is based on the idea of partitioning the input matrix into small, same-sized, 2D tiles to improve multicore/multithreaded execution. In this study CSR5 method is compared with Intel MKL, NVIDIA cuSPARSE, CUSP, and ViennaCL on 4 different platforms:

[1]Yzelman Sparse Library, `http://albert-jan.yzelman.net/software.php`

Intel Xeon, NVIDIA Geforce, AMD Radeon R9, and Intel Xeon Phi. 24 different matrices are used in tests. For each matrix, 10 experiments are performed. Each experiment involves 1000 iterations of which the average run time is recorded. The best timing of 10 experiments is reported. For the execution time measurements, OpenCL is used on the AMD platform, and wall-clock is used for the others. Time measurements are reported in GFlops. The source code of this study is publicly available[2].

Kourtis et al. introduce a storage format, Compressed Sparse eXtended (CSX), that is used to take advantage of recurring substructures in sparse matrices and generate specialized code at run time [10]. Experiments are conducted on an 8-core Intel Harpertown and a 24-core Intel Dunnington. Authors compile a list of 15 matrices to experiment with. 128 executions of SpMV are done for each matrix with randomly generated input vectors. No measures are taken to prevent cache reuse in-between each iteration. The speed up of CSX against CSR, BCSR, and CSR-DU formats are reported. The time measurement approach is not stated in the study. However the source code of the study[3] reveals that the time measurements are performed based on a custom method using CPU frequencies and time stamp counter registers.

Merrill and Garland propose a merge-based scheduling method for parallel execution of SpMV [3]. The essence of their approach is to merge row indices and values into a single sequence, and do the partitioning accordingly to remedy the downsides of row or nonzero element partitioning. The partitioning is designed to have very low preprocessing overhead on the CSR format. The study conducts experiments on two different platforms, Intel Xeon E5-2690v2 CPU and NVIDIA Tesla K40 GPU, using 4,201 different matrices. Authors compare the performance of their approach

[2]CSR5 Benchmark, `https://github.com/bhSPARSE/Benchmark_SpMV_using_CSR5`
[3]CSX Library, `https://github.com/cslab-ntua/csx`

against Intel MKL and NVIDIA cuSPARSE. The study reports execution times in milliseconds and also performance in GFlops. The paper does not explicitly state how execution times are measured, how many iterations are done per experiment, or which measurements are reported – best of all or average. However, based on our inspection of their source[4], we have found out that

- The number of iterations is adaptively set in between 100 and 5000 depending on the number of nonzero elements in matrix, trying to achieve 16 billion nonzeros per experiment.
- Platform-dependant profilers are used for timing: For the experiments on CPUs OpenMP timers are used if OpenMP is present. Otherwise, *getrusage* system calls are used on Linux platforms and *Performance Counters* are used on Windows. For the GPU experiments CUDA events are used to measure elapsed time.
- Average elapsed time is reported per experiment.

The study also provides the "Stream Triad" [21] throughput scores for both platforms to give a comparative view of how good each method is in utilizing the processor.

Xie et al. propose Compressed Vectorization-oriented sparse Row (CVR) format which aims to improve SIMD lane utilization and cache efficiency by assigning each indivudual matrix row to a different SIMD lane [22]. They conduct experiments on Intel Xeon Phi (Knights Landing 7250) platform using 58 different matrices. Performance of CVR is compared to Intel MKL CSR, Intel MKL CSR(I), ESB [23], VHCC [24], CSR5 [9] with respect to average execution time of 1000 iterations and preprocessing overheads. The `gettimeofday` system call is used to measure the wall-clock time. The source code for this work is available online[5]

[4]Merge Based Parallel SpMV, `https://github.com/dumerrill/merge-spmv`
[5]CVR, `https://github.com/puckbee/CVR`

Buluç et al. propose Compressed Sparse Blocks (CSB) scheme which partitions the input matrix into fixed sized blocks organized similar to COO format [25]. This scheme is tested on AMD Opteron 8214 and Intel Xeon X5460 platforms with 14 different matrices. Subsequently they improve CSB with Bitmasked Compressed Sparse Block scheme (BiCSB) by introducing bitmasked indexing mechanism for better bandwidth utilization [11]. BiCSB is benchmarked on AMD Opteron 8431 and Intel Xeon 7550 using 7 matrices. Neither of the studies reports how many iterations are performed or how the time is measured. But since both schemes are published as part of Compressed Sparse Block library[6], it is possible to find out that library reports average GFlops achieved for 10 iterations and timing functions from Cilk++ library are used for time interval profiling.

Augusto et al. [26] conduct a performance comparison of popular linear algebra libraries; Intel MKL, CUSP, NVIDIA cuSPARSE, and ViennaCL. They experiment with relatively small set of five matrices on two different platforms: Intel Xeon E5-2650v2 CPU and NVIDIA Titan GPU. They report average GFlops performance of 1000 SpMV executions.

Another study that benchmarks popular linear algebra libraries in terms of SpMV performance is done by Kasmi et al. [27]. This study compares CUSP and Intel MKL libraries on NVIDIA GTX580 GPU and Intel Core i7 CPU. Five different matrices are included in the experiments and the performance results are reported as GFlops. However there is no information provided on details of the experiments such as how many iterations are done or how the performance values are obtained – average or best of all iterations.

Kamin et al. [28] evaluate various code specialization techniques for optimizing performance of SpMV and compare performance of these techniques to Intel MKL

[6]Compressed Sparse Block library, `https://people.eecs.berkeley.edu/~aydin/csb/html/index.html`

library, BiCSB [11] and CSX [10]. The experiments are run on 23 different matrices. Each experiment involves 10,000 SpMV runs and is repeated 5 times. The fastest of those five experiment is chosen for comparison. Results of each experiment are reported as GFlops and also speed up ratio compared to Intel MKL.

Langr and Tvrdik define a set of criteria to evaluate sparse storage matrix formats [1]. These criteria can be classified as in Table 1 into 4 groups: Runtime Performance, Memory Footprint, Preprocessing Runtime Overhead, Preprocessing Memory Overhead. Although memory-related criteria are hard to measure programmatically in an accurate and reliable way, runtime-related criteria can easily be used to evaluate any SpMV Method.

Table 1: SpMV format evaluation criterion by Langr and Tvrdik.

Runtime Performance	Maximum Performance in Flops Performance w.r.t. Standard CSR Performance w.r.t. Platform Upper Bound
Memory Footprint	Total Footprint Footprint per Nonzero Element Footprint per Nonzero w.r.t Standard CSR
Preprocessing Overhead	Conversion from Standard CSR Finding Optimal Parameters
Memory Overhead	Conversion from Standard CSR Finding Optimal Parameters

Langr and Tvrdik do not provide an implementation that other researchers can use to integrate their methods. In our framework, we provide timing probes to measure the maximum performance (reported in microseconds and in flops), the overhead of preprocessing the input, and the cost of conversion to the new format. SpMV using CSR is a built-in method, so it is straightforward to find the relative performance.

In our work, we deliberately refrain from providing a pre-compiled data set of input matrices, because the ideal set depends on target-specific features such as the cache size of the machine on which the experiments are run, or the matrix properties for which the new SpMV format is optimized.

CHAPTER IV

PROPOSED FRAMEWORK

In this section we present the overview and details of our SpMV benchmarking framework. From the user point of view, it is a tool with a command-line interface (CLI) that is executed by providing the required and optional command-line arguments. The help message and the CLI parameters are shown on Figure 7. The framework takes the name of the matrix file and the SpMV method to use as the required parameters. With optional parameters, the user can set the number of threads, turn on the debug mode, set the number of iterations for invoking the SpMV function and the number of warm-up iterations, and dump the contents of the output to a file for post-processing. The matrices are read from Matrix Market exchange files [29] that typically have the `.mtx` extension. The output of an example benchmark session with 8 threads, 10 warm-up iterations, and 100 benchmark iterations using the plain CSR SpMV method “pcsr” on the matrix file “s3dkt3m2.mtx” is shown on Figure 8.

```
> ./spmv_benchmarking --help

OzU SRL SpMV Benchmarking.
  Usage:
    spmv_benchmarking <mtxFile> <method>
      [--threads=<num>] [--debug] [--iters=<count>]
      [--dump-output] [--warmups=<count>]
    spmv_benchmarking (-h | --help)
    spmv_benchmarking --version
  Options:
    -h --help             Show this screen.
    --version             Show version.
    --debug               Turn debug mode on.
    --dump-output         Dump output vector to <method>.out
    --threads=<num>       Number of threads to use [default: 1].
    --iters=<count>       Number of iterations for benchmarking [default: 10].
    --warmups=<count>     Number of warmup iterations before benchmarking [default: 5].
```

Figure 7: Command-line interface parameters of the framework.

```
> ./spmv_benchmarking s3dkt3m2.mtx pcsr  --threads=8 --warmups=10 --iters=100

Options:
========
matrix file: s3dkt3m2.mtx
method     : pcsr
threads    : 8
iters      : 100
warmups    : 10
debug      : false
dump output: false
=================

1    1378494 usec.    ReadInputFile
1          3 usec.    Init
2     953203 usec.    ConversionToCSR
2          0 usec.    Analysis
2          0 usec.    Conversion
1     953479 usec.    Preprocess
1     155307 usec.    Warm up
1    1395486 usec.    Spmv

       13954 usec.    perIteration
    0.537976 GFlops   perIteration
         100 times    iterated
```

Figure 8: An example run with 8 threads, 10 warm-up iterations, and 100 benchmark iterations.

An SpMV benchmarking session involves multiple stages that are interesting to time individually for the purposes of benchmarking. A session starts with data-independent initialization steps such as initialization of the threading framework, initialization of the GPU, querying of the CPU or GPU features. Following this stage, the input matrix is read from the specified file, and converted to the CSR format. Next come the method-specific initialization steps such as converting the input matrix to a custom format and processing the matrix for collecting features that will be used by the method for tuning the operation. This initialization phase is followed by a small number of invocations of the SpMV function for eliminating the effects of cold-start of the cache. This phase is called “warm-up”. Next comes the actual SpMV execution phase, where the function is invoked for many times consecutively. The exact number of iterations can be adjusted, and may range from tens to thousands, depending on the duration. The aim in making many SpMV invocations is to time a total duration

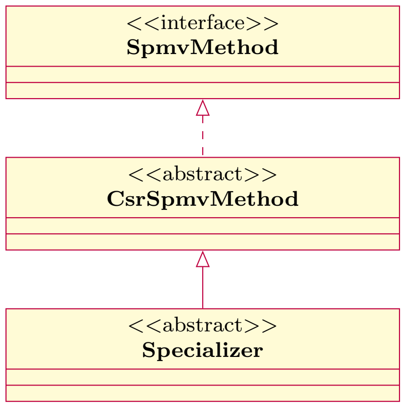


Figure 9: SpMV methods hierarchy.

that is long enough that the effects of interference on individual function calls can be diminished. Finally, GPU-based methods also include data-copying phases that are also timed. We classify the SpMV stages in accordance with the performance criterion defined by Langr and Tvrdik [1] in terms of time measurement as overheads and performance measurements. The timing probes for the phases we discussed above are already inserted in our framework. It is straightforward to insert new probes should the user desire to do so.

4.1 SpMV Method Interfaces

At the center of the framework, there are three interfaces at different refinement levels that constitute a contract to integrate new SpMV methods into the framework: `SpmvMethod`, `CsrSpmvMethod`, and `Specializer`. The inheritance hierarchy of these interfaces is shown in Figure 9. `SpmvMethod` is the most generic definition of an SpMV method. `CsrSpmvMethod` derives from `SpmvMethod`, and is designed for methods that take an input matrix in CSR format. `Specializer` further derives from `CsrSpmvMethod`, and is an abstraction of methods that generate code at runtime. Below we discuss details of these interfaces.

- `SpmvMethod`: This interface is the most basic interface to implement an SpMV

```
class SpmvMethod {
public:
  virtual void init(unsigned int numThreads) = 0;
  virtual void preprocess(MMMatrix<double> &matrix) = 0;
  virtual void spmv(double * v, double * w) = 0;
};
```

Figure 10: `SpmvMethod` interface definition.

method. As depicted in Figure 10, the interface defines three functions to execute an SpMV operation. The `init` function is the first function that is called during a benchmarking run. It only accepts a single parameter from the framework, `numThreads`, which is the factor of parallelization. The `init` function performs all the data independent initialization, such as initialization of the threading framework. The second function is `preprocess`. It receives the input matrix in type `MMMatrix`. This is an in-house implementation of the Matrix Market format, and offers conversions to the common COO, CSC, and CSR formats. The `preprocess` function executes data-dependent initialization steps, such as converting the matrix to a different format, reordering the matrix data, or tuning the optimization parameters. Lastly, the `spmv` function is the actual function that performs the SpMV calculation. Figure 11 shows the generation execution flow of any method as driven by the framework.

- `CsrSpmvMethod`: This interface is an extension on top of `SpmvMethod`. It includes some default behaviour for executing SpMV on CSR formatted matrices and defines means of customization of preprocessing step. Figure 12 shows the definition for this interface.

  `CsrSpmvMethod` converts the input matrix to CSR format then performs a one-dimensional, row-based partitioning to split the matrix into stripes. The number of stripes is determined according to the number of threads. The matrix

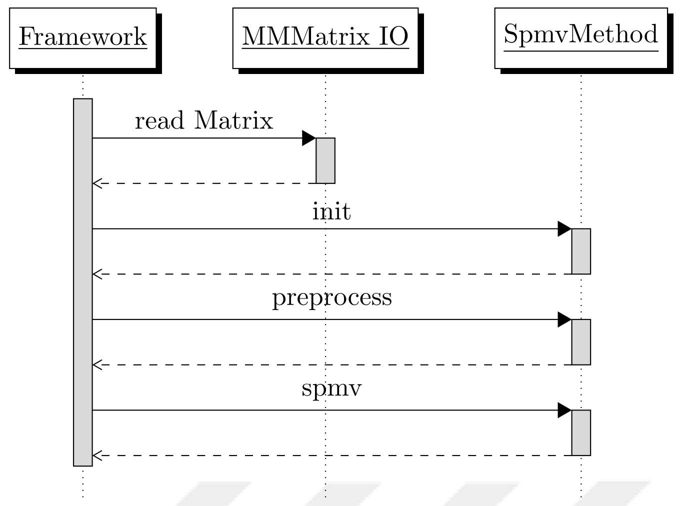


Figure 11: `SpmvMethod` execution flow.

```
class CsrSpmvMethod : public SpmvMethod {
public:
  virtual void init(unsigned int numThreads);
  virtual void preprocess(MMMatrix<double> &matrix);
  virtual void spmv(double * v, double * w) = 0;
protected:
  virtual void analyzeMatrix();
  virtual void convertMatrix();
  std::vector<MatrixStripeInfo> *stripeInfos;
  unsigned int numPartitions;
  std::unique_ptr<CSRMatrix<double>> csrMatrix;
};
```

Figure 12: `CsrSpmvMethod` definition.

in CSR format is accessible via the member variable `csrMatrix`, and the partitioning information is available via `stripeInfos`. `CsrSpmvMethod` allows further refinements on preprocessing steps. The `analyzeMatrix` function can be used to extract features from the matrix that will be useful in the later stages, e.g. recurring non-zero patterns. The `convertMatrix` function shall be implemented to convert the matrix from CSR to another format, or to reorder the matrix data. These two functions are profiled by the framework for the convenience of researchers.

Figure 13 shows the execution flow for a `CsrSpmvMethod`.

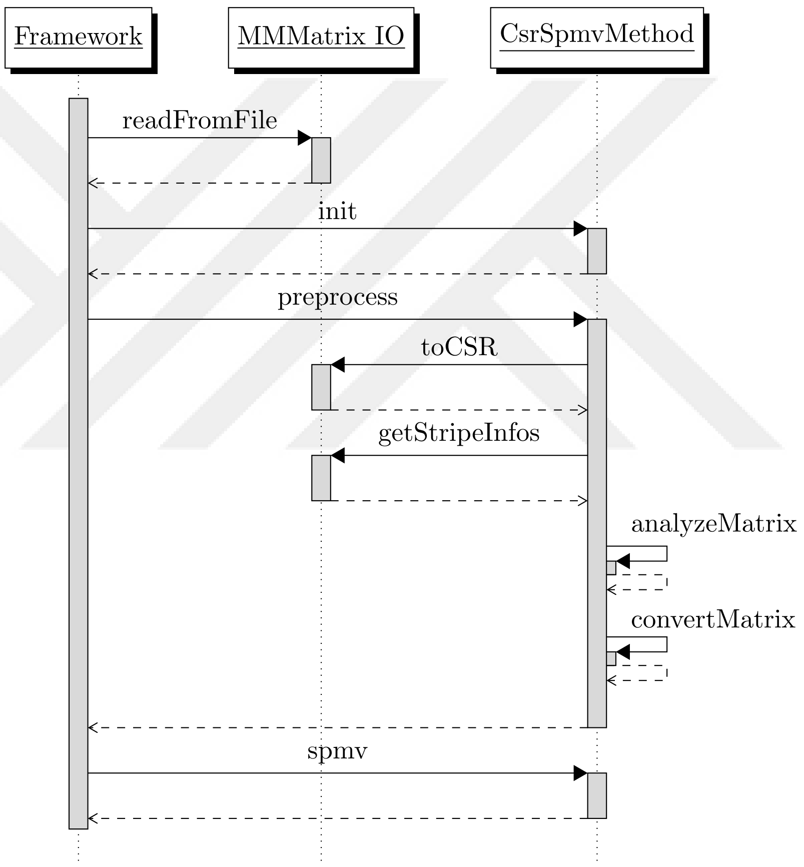


Figure 13: `CsrSpmvMethod` execution flow.

```
class Specializer : public CsrSpmvMethod {
public:
  virtual void init(unsigned int numThreads) final;
  virtual void preprocess(MMMatrix<double>& matrix);
  virtual std::vector<asmjit::CodeHolder*> *getCodeHolders() final;
  virtual void spmv(double*  v, double*  w) final;

protected:
  virtual void emitMultByMFunction(unsigned int index) = 0;
  std::vector<asmjit::CodeHolder*> codeHolders;
  std::vector<MultByMFun> functions;
  std::unique_ptr<CSRMatrix<double>> matrix;

private:
  virtual void emitCode() final;
  asmjit::JitRuntime rt;
};
```

Figure 14: `Specializer` definition.

- `Specializer`: This interface is derived from `CsrSpmvMethod` with the purpose of generating SpMV code specialized for the given input matrix. The definition is given in Figure 14. The most important function in this interface is `emitMultByMFunction`, which is responsible for code generation. `Specializer` interface is an abstraction of the SpMV methods implemented in Thundercat [30, 14], and uses the `asmjit`[1] just-in-time assembler. In addition to the time measurements taken for `CsrSpmvMethod`, the framework includes probes for measuring the costs of code generation.

  The execution flow of this method is similar to the `CsrSpmvMethod`. As the first step of preprocessing, the input matrix is converted to CSR format. Then the input matrix is analyzed to obtain useful information for code specialization. Following analysis, input matrix can be converted to any other form if necessary. Lastly, specialized code for the final form of input matrix is generated. When the preprocessing is done and specialized code is generated, framework calls the `Specializer::spmv` function to initiate the SpMV calculation with specialized functions. Figure 15 illustrates execution flow of `Specializer`.

[1] `asmjit` JIT and Remote Assembler, `https://github.com/asmjit/asmjit`

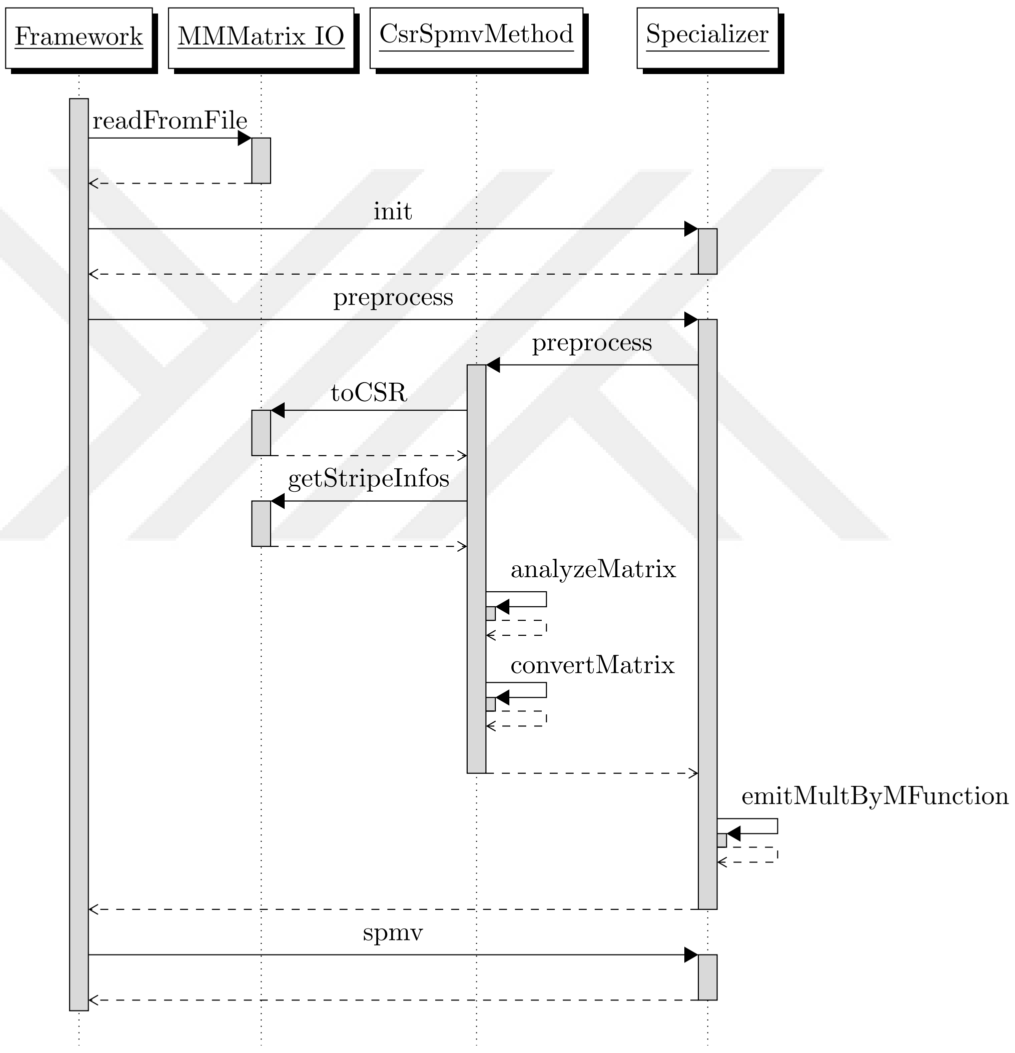


Figure 15: `Specializer` execution flow.

```
class Profiler {
public:
  static void recordTime(std::string description,
                         std::function<void()> codeBlock);
  static void recordSpmv(std::function<void()> codeBlock);
  static void recordSpmvOverhead(std::string description,
                                 std::function<void()> codeBlock);
  static void print(unsigned int numIters,
                    unsigned int NNZ,
                    unsigned int flopPerNNZ = 2);
};
```

Figure 16: `Profiler` definition.

4.2 Method Registry

`SpmvMethodRegistry` is the component responsible for storing and providing SpMV methods for benchmarking. The framework makes use of template based helpers to register existing methods via `REGISTER_METHOD` macro. This macro defines a method provider type for a given method and then creates a static instance of this method specific provider. During instantiation, a method provider registers itself to `SpmvRegistry`. Auto instantiation and registering of methods provides a straightforward and convenient way of integrating new methods; this way an SpMV method easily makes itself available in the command-line interface. When the framework queries registry for a method by method name, `SpmvMethodRegistry` fetches the registered method provider and uses provider to create an instance of the queried method.

4.3 Profiling Time Intervals

In our framework, we provide an execution time profiler. The profiler is able keep track of wall-clock, and measures execution time in *microseconds* as well as in *GFlops*. The definition of our `Profiler` class is shown in Figure 16. Our profiler implementation supports three different types of time measurement:

- The `recordTime` function is a general-purpose time profiler. As arguments, it takes a string as the description, and the code block in the form of a lambda function, whose execution time will be measured. The `recordTime` function can be used for *nested* time measurements. In other words the code block that is being measured can also call `recordTime` to introduce further fine grained measurements. In this case, each of the nested measurements is recorded as a child of the enclosing measurement in a children-first (postorder) fashion. Figure 17 shows an example usage of the `recordTime` function. In our framework, we use this type of measurements to profile initialization and preprocessing overheads.

- The `recordSpmv` function tracks the entire SpMV calculation, namely execution of `SpmvMethod::spmv`. This function is designed to be called only once and the framework should be the only caller. The `recordSpmv` creates the measurement named `"Spmv"` once the SpMV exection is completed.

- The framework is able to record the overheads via `recordSpmvOverhead` function. This is especially useful for the cases where overhead tasks cannot be easily moved out of the SpMV logic. One of those cases might be the tasks that move the data between CPU and GPU. Similar to `recordTime` function, `recordSpmvOverhead` also accepts a code block and a description; however, it does not support nested measurements. Total overhead time measured by `recordSpmvOverhead` function is deducted from the SpMV execution time in order to find the net SpMV execution time. Figure 18 depicts an example usage scenario for the `recordSpmvOverhead` function.

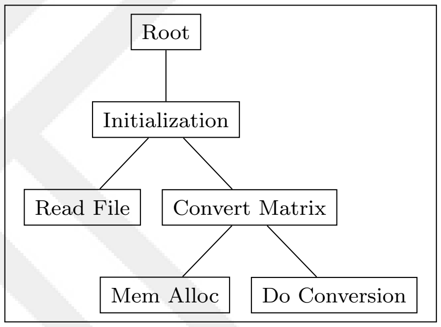
```
void readFile() {
  /* read file */
}

void convertMatrix() {
  Profiler::recordTime("Mem Alloc", [&]() {
    /* allocate memory */
  });

  Profiler::recordTime("Do Conversion", [&]() {
    /* do conversion */
  });
}

void initilize() {
  Profiler::recordTime("Initialization", [&]() {
    Profiler::recordTime("Read File", [&]() {
        readFile();
    });
    Profiler::recordTime("Convert Matrix", [&]() {
        convertMatrix();
    });
  });
}
```

(a) Nested calls to `Profiler::recordTime` function.



(b) Measurement hierarchy.

```
2   1300 usec.   Read File
3    500 usec.   Mem Alloc
3    200 usec.   Do Conversion
2    700 usec.   Convert Matrix
1   2000 usec.   Initialization
```

(c) Measurement report in postorder (children-first).

Figure 17: `Profiler::recordTime` example.

```
void copyToGPU(double * data) {
  /* copy data to GPU */
}

void copyToCPU(double * data) {
  /* copy data to CPU */
}

void spmv(double * inVector, double * outVector) {

  Profiler::recordSpmvOverhead("Copy to GPU", [&]() {
      copyToGPU(inVector);
  });

  /* Do SpMV on GPU */

  Profiler::recordSpmvOverhead("Copy to CPU", [&]() {
    copyToCPU(outVector);
  });

}
```

(a) Measuring the SpMV overhead.

```
1   1300 usec.  Read File
1    750 usec.  Init
1   1050 usec.  Preprocess
1    850 usec   Spmv

Spmv Overheads:
    100 usec.   Copy to GPU
    120 usec.   Copy to CPU

    630 usec.   Spmv w/o Overheads
```

(b) Measurement report with overheads.

Figure 18: `Profiler::recordSpmvOverhead` example.

CHAPTER V

EXTENDING THE FRAMEWORK

The implementation or integration of a new method begins with deciding which SpMV interface to use as the base. As discussed in Section 4.1, we provide three different interfaces for integration:

- `SpmvMethod` is the most basic interface applicable for any SpMV method or format, but provides no builtin functionality other than basic profiling.
- `CsrSpmvMethod` is convenient for CSR-based methods since it provides automatic conversion to the CSR format and defines two injection points for analyzing the input matrix and transforming the CSR matrix to an internal format.
- `Specializer` supports run time code generation on top of `CsrSpmvMethod`.

New SpMV methods should be derived from one of the above framework interfaces and implement the functions of the parent according to the guidelines described in Section 4.1.

Another important point for implementing a new SpMV method is how to profile it. All of the interfaces in our framework provide basic profiling intervals. These particularities are discussed in the previous chapter. The default profiling should be sufficient for most of the cases. However it is at developers' will to introduce more granular profiling or mark certain intervals of SpMV execution as overheads.

Rest of this chapter describes how to add a hypothetical SpMV method to our framework.

Suppose we want to implement an SpMV method that uses the CSC format. Let us also assume that our method reorders columns by the number of nonzero elements

```
class NewMethod : public SpmvMethod {
public:
    virtual void init(unsigned int numThreads);
    virtual void preprocess(MMMatrix<double> &matrix);
    virtual void spmv(double * v, double * w);
private:
    std::unique_ptr<CSCMatrix<double>> cscMatrix;
};
```

Figure 19: `NewMethod` definition.

they contain. In order to implement this method we need to:

1. Convert input matrix to CSC format
2. Reorder the converted matrix
3. Align input vector to the reordered matrix
4. Run the SpMV function
5. Align the output matrix to the initial order

Note that this method is not the most efficient and performant SpMV method; however, it provides enough variety to cover different scenarios for this tutorial.

Since the most basic interface is `SpmvMethod`, we are going to use this interface for our implementation. Figure 19, shows the definition of our method. We inherit all the functions from `SpmvMethod`. We also have a member variable to hold the matrix CSC format after conversion.

As mentioned before, `init` function is responsible for initialization steps that are independent of the input data. Suppose our method runs in a multithreaded fashion. Thus, we need to initialize the threading framework. This operation suits well to the `init` function, whose implementation is given in Figure 20.

Continuing with our todo list, converting the input matrix to CSC and reordering the columns are considered as preprocessing steps, since they only depend on the input matrix. We use the `MMMatrix` argument to convert the matrix, and suppose that

```
void NewMethod::init(unsigned int numThreads){
  MyThreadingFramework::init();
  MyThreadingFramework::setNumberOfThreads(numThreads);
}
```

Figure 20: `init` function.

```
void NewMethod::preprocess(thundercat::MMMatrix<double> &matrix) {
  Profiler::recordTime("Conversion to CSC", [&] () {
       cscMatrix = matrix.toCSC();
  };

  Profiler::recordTime("Sort Input Matrix", [&] () {
      SmartSorter::sortByNnz(cscMatrix);
  });
}
```

Figure 21: `preprocess` function.

there is already a sorter provided that can track all the sort actions and revert them. Although the framework measures the execution time of the `preprocess` function, we might still want to introduce more granular timings, by measuring conversion and sorting separately. Figure 21 outlines the implementation of the `preprocess` function.

Now we can perform the actual SpMV calculation. In order to achieve this, we need to first sort the input vector, do the SpMV execution, and sort the output vector to align with the initial indices. Again, for simplicity, we assume that all this functionality is provided to us. However, profiling the `spmv` function might need tweaking. The framework measures the entire execution of `spmv`, which also includes all sorting sections. If we want to avoid this and only measure the actual SpMV execution, we should mark sorting parts as overhead using `recordSpmvOverhead`, so that the framework will deduct the time spent during sorting from the total execution time of `spmv`. We suppose actual SpMV logic is already provided. Figure 22 shows the `spmv` function.

The last step to complete our method integration is registering the method to

```
void NewMethod::spmv(double * v, double * w){
  Profiler::recordSpmvOverhead("Sort Input Vector", [&] (){
    SmartSorter::sortInputVector(v);
  });

  w = SpmvLogic::multiply(cscMatrix, v);

  Profiler::recordSpmvOverhead("Revert Output Vector", [&] (){
      SmartSorter::revertOutputVector(w);
  });
}
```

Figure 22: spmv function.

```
REGISTER\_METHOD(NewMethod, "newmethod")
```

Figure 23: Registering method to the framework.

the framework. This can be achieved by calling the `REGISTER_METHOD` macro as in Figure 23. Following that, our method is fully integrated in the framework with the name `"newmethod"`. It is ready to run for benchmarking and to be compared to all the other state of the art methods which are already integrated into the framework. Figure 24 shows the entire implementation of our method

```
#include "method.h"

class NewMethod : public SpmvMethod {
public:
  virtual void init(unsigned int numThreads);
  virtual void preprocess(MMMatrix<double> &matrix);
  virtual void spmv(double * v, double * w);
private:
  std::unique_ptr<CSCMatrix<double>> cscMatrix;
};
```

(a) Header file for `NewMethod`.

```
#include "newmethod.h"
#include "spmvRegistry.h"
#include "profiler.h"

REGISTER_METHOD(NewMethod, "NewMethod")

void NewMethod::init(unsigned int numThreads){
  MyThreadingFramework::init();
  MyThreadingFramework::setNumberOfThreads(numThreads);
}

void NewMethod::preprocess(thundercat::MMMatrix<double> &matrix) {
  Profiler::recordTime("Conversion to CSC", [&] () {
       cscMatrix = matrix.toCSC();
  };
  Profiler::recordTime("Sort Input Matrix", [&] () {
      SmartSorter::sortByNnz(cscMatrix);
  });
}

void NewMethod::spmv(double * v, double * w){
  Profiler::recordSpmvOverhead("Sort Input Vector", [&] (){
    SmartSorter::sortInputVector(v);
  });

  w = SpmvLogic::multiply(cscMatrix, v);

  Profiler::recordSpmvOverhead("Revert Output Vector", [&] (){
      SmartSorter::revertOutputVector(w);
  });
}
```

(b) Source file for `NewMethod`.

Figure 24: Implementation of `NewMethod`.

CHAPTER VI

BUILTIN SPMV INTEGRATIONS

We provide several SpMV methods already integrated into the framework. These built-in methods are particularly useful because they allow users to benchmark their own methods against the existing ones without any effort. They also constitute concrete examples of how to integrate an SpMV method; a user can implement their own method by duplicating an existing one and modifying it appropriately. The available SpMV methods are listed in Table 2. In this section we discuss these methods and their integration.

Table 2: Built-in SpMv methods.

Reference	*CPU*	*GPU*	*Specializer*
PlainCSR	Intel MKL	ViennaCL	CSRbyNZ
PlainCSR4	ViennaCL	cuSPARSE	CSRLenWithGOTO
PlainCSR8	Yzelman - Hilbert	CUSP	CSRWithGOTO
PlainCSR16	Yzelman - HTS		GenOSKI
PlainCSR32	Yzelman - zzcrs		RowPattern
Incremental CSR			Unfolding

6.1 Reference Benchmarking Methods

Reference benchmarking methods are a set of CSR-based SpMV methods that establish a baseline for a comparison. They form the most basic implementation of SpMV with a one-dimensional partitioning that aims to split the matrix into stripes with as equal number of nonzero elements per stripe as possible. Each stripe is processed by a thread concurrently. Parallelization is achieved by means of OpenMP pragmas. The implementations themselves are derived from the Thundercat library [30, 14] and

```
for(int i = 0; i < rowCount; i++){
  double res = 0.0;
  for (int j = rows[i]; j < rows[i + 1]; j++) {
    output += vals[j] * x[cols[j]];
  }
  y[i] += res;
}
```

(a) PlainCSR method (unrolling factor: 1).

```
for(int i = 0; i < rowCount; i++){
  double res = 0.0;
  for (int j = rows[i]; j < rows[i + 1] - 3; j += 4) {
    res += vals[j] * x[cols[j]];
    res += vals[j + 1] * x[cols[j + 1]];
    res += vals[j + 2] * x[cols[j + 2]];
    res += vals[j + 3] * x[cols[j + 3]];
  }

  for (; j < rows[i + 1]; j++) {
    res += vals[j] * v[cols[j]];
  }
  y[i] += res;
}
```

(b) PlainCSR4 method (unrolling factor: 4).

Figure 25: PlainCSR and PlainCSR4 variant.

integrated into the framework as an instance of `CsrSpmvMethod`.

- *Plain CSR:* This method is the most straightforward implementation of SpMV over the CSR format. In our integration, there are 5 variants; each having a different unrolling factor for the inner loop – 1, 4, 8, 16 and 32. Figure 25a and Figure 25b illustrate *PlainCSR*, where the unrolling factor is simply 1, and *PlainCSR4*, where the factor is 4, respectively.

- *Incremental CSR:* This method is a CSR variant that attempts to decrease the memory footprint of row pointers [31]. Rather than storing absolute position of row beginnings, it stores displacement with respect to the previous row beginning. Figure 26 compares a normal CSR row pointer array and an Incremental CSR row pointers array. As the values stored in the array become smaller, this

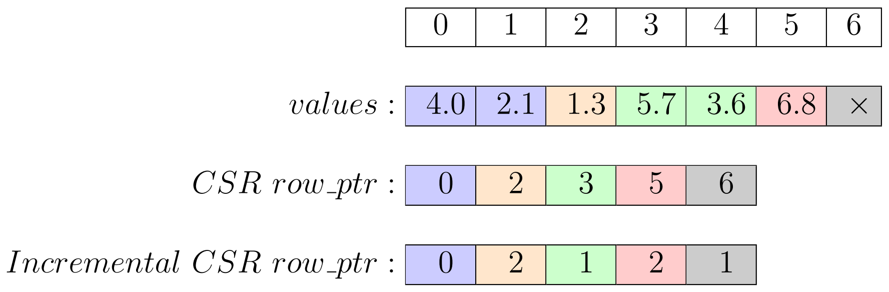

0 1 2 3 4 5 6
values : 4.0 2.1 1.3 5.7 3.6 6.8 ×
CSR row_ptr : 0 2 3 5 6
Incremental CSR row_ptr : 0 2 1 2 1

Figure 26: CSR and Incremental CSR row pointers.

method enables the chance that the data type of the *rowptr* array is two-byte or even one-byte integer, instead of four. Although this has the potential of reducing the memory footprint, it comes with a runtime penalty, since additional steps are needed to reconstruct the row pointer values.

6.2 Integrated CPU and GPU methods

- *Intel MKL:* Intel Math Kernel Library (Intel MKL) offers a rich collection of highly optimized vastly threaded math routines supporting a wide range of usage scenarios including but not limited to Linear Algebra, Fast Fourier transform and Deep Neural Network primitives [15]. The library is designed to run on CPU, is available on Windows, Linux, and MacOS platforms, and has support for C/C++ and FORTRAN.

  MKL provides a number of different variants for SpMV. We integrate matrix vector multiplication routine for double precision sparse matrices in CSR format, `mkl_dcsrmv`, as an `CsrSpmvMethod` to our framework. Being a BLAS level 2 [32] routine, the `mkl_dcsrmv` function supports SpMV in the following forms:

$$y \leftarrow \alpha A \times x + \beta y \qquad \text{or} \qquad y \leftarrow \alpha A^T \times x + \beta y$$

  In our integration we omit the transpose operation on input matrix and fix the values for α and β to 1. This yields a more simple and comparable SpMV operation: $y \leftarrow A \times x + y$.

The MKL integration follows the general flow of the `CsrSpmvMethod`; in the `init` phase we inform MKL about the parallelization factor, in `preprocessing` we convert the input matrix to CSR format, and finally for `spmv`, we call the `mkl_dcsrmv` function to compute $y \leftarrow A \times x + y$.

- *Yzelman:* Yzelman put together a collection of various research oriented sparse matrix formats and computations in his "Sparse Library"[1]. The library not only consists of methods and formats that is studied by Yzelman and his colleagues such as ZZCRS and Hilbert-curve ordered BICRS, but also others like COO, CSR, HTS, ICRS.

  The methods in Sparse Library follow a common approach. Each method accepts a COO formatted matrix – referred as Triplet Scheme in the library. Upon its instantiation, a method converts the input data to the format on which it operates. Then user can call the SpMV function named `zax` which computes $z \leftarrow A \times x + z$.

  This common approach proves to be helpful in our integration. We define a base template method for all methods to be integrated. Our template method, `YzelmanMethod`, is derived from `SpmvMethod`. It defines a function to inject an SpMV method from the library. In preprocessing, `YzelmanMethod` converts input matrix to COO format (Triplet) and instantiates a library method using the injection function. The SpMV execution is as simple as calling the `zax` function of the injected method. With this approach it becomes convenient to integrate any method since integration is reduced to just defining an injection function. Our integration includes ZigZag CRS (ZZ_CRS), Hilbert curve ordered BICSR (Hilbert), and Hilbert Triplet Scheme (HTS) from Yzelman's Sparse Library.

[1]Sparse Library, `http://albert-jan.yzelman.net/software.php`

- *cuSPARSE:* NVIDIA cuSPARSE library [16] consists of a series of basic linear algebra routines for dealing with sparse matrices on NVIDIA GPUs. The library is built on top of NVIDIA general purpose processing API, CUDA[2]. cuSPARSE runs on Windows, Linux and MacOS and can be called from C/C++. The library is organized around four groups of operations:

  - Operations on a sparse vector and a dense vector
  - Operations on a sparse matrix and a dense vector
  - Operations on a sparse matrix and a set of dense vector
  - Conversions between different matrix formats

  Our framework integrates `cusparseDcsrmv` function which is the double precision SpMV function offered by cuSPARSE. Analogous to MKL, this function also computes $y \leftarrow \alpha \; op(A) \times x + \beta y$, where $op()$ is one of identity function, transpose, or conjugate transpose on A. In our integration we again set value of α and β to 1 and use identity function to calculate a simpler form of SpMV; $y \leftarrow \alpha A \times x + \beta y$.

  Nevertheless, the integration of cuSPARSE is still not simple for two main reasons. First, cuSPARSE assumes that the input data for the routines are already on GPU memory. If not, developers are expected to move the data between CPU and GPU. Although these copy operations have to be performed in every iteration of a benchmark run, they should not be considered as part of the SpMV execution, but rather as overheads. Second, the sources for GPU and CPU need to be compiled separately and then linked together, since they are usually compiled with different compilers – icpc or gcc for CPU, nvcc for GPU. Furthermore, the compilers for CPU generally support the most recent

[2]CUDA Zone, `https://developer.nvidia.com/cuda-zone`

```
class CusparseAdapter {
public:
  void init();
  void preprocess(int nnz, int m, int n, int *rowPtr,
                  int *colIdx, double *values);
  void setX(double *x);
  void getY(double *y);
  void spmv();
};
```

Figure 27: `CusparseAdapter` definition.

C++ standards whereas GPU compilers often do not. Therefore, compile unit isolation for GPU and CPU sources is essential.

In order to overcome these issues, we divide the cuSPARSE integration into two. We develop an adapter that performs all cuSPARSE and CUDA related operations, outlined in Figure 27, and a light proxy layer that implements `SpmvMethod`. The proxy layer is responsible for the communication between framework and the adapter, and for profiling the operations performed by the adapter.

The flow of execution starts with a call to `init` method of proxy and it is delegated to adapter in order to initialize the cuSPARSE environment. Subsequently, preprocessing starts. Proxy converts the matrix to CSR format and hands the data over to adapter. Adapter allocates memory for input matrix, input vector and output vector on GPU, then copies input matrix to GPU. Once the preprocessing ends, SpMV execution is performed in three steps. First, the proxy passes the input matrix to adapter, and the adapter copies the input vector to GPU. Following that, proxy initiates the SpMV execution and the adapter calls the corresponding cuSPARSE function. Finally, the output vector is copied from GPU to CPU. One important point to note in SpMV execution is, all copy operations in-between CPU and GPU are profiled as SpMV execution overheads by our profiler and not included in SpMV execution timings.

Figure 28 illustrates this flow.

- *CUSP:* CUSP library [17] is a template based C++ library for sparse linear algebra and graph computations that is built on top of NVIDIA CUDA and Thrust [33].

  CUSP library offers two SpMV functions for several formats. The `multiply` function that performs $y \leftarrow A \times x$, and the `generalized_spmv` function that can be used to perform $z \leftarrow A \times x + y$. We initially integrated the latter function. However, CUSP does not allow accumulating on the same vector to calculate $y \leftarrow A \times x + y$; therefore, we used an intermediary vector which slowed down the execution significantly. We integrate both functions as different methods, and let the users of our framework choose the integration they need.

  The integration follows a similar approach to cuSPARSE since it has the identical requirements for compile unit isolation. CUSP is also integrated as an `SpmvMethod`, and the implementation is again divided into two as a proxy and an adapter. However, since CUSP works with Thrust pointers, all the raw CUDA pointers are translated into Thrust pointers during preprocessing.

  The execution flow of CUSP integration is also very much like cuSPARSE, since they both operate on top of CUDA. First, the CUDA environment is initialized, then matrix is converted to CSR format. Following that, all input data is copied to GPU and SpMV is executed. Finally, the result is fetched from GPU memory to CPU memory. The time spent in moving data between CPU and GPU is marked as overhead and excluded from the SpMV execution time.

- *ViennaCL:* ViennaCL[3] is a scientific computing library introduced by Rupp et al. [18]. It is designed for providing common data types for linear algebra

[3]ViennaCL, `http://viennacl.sourceforge.net/`

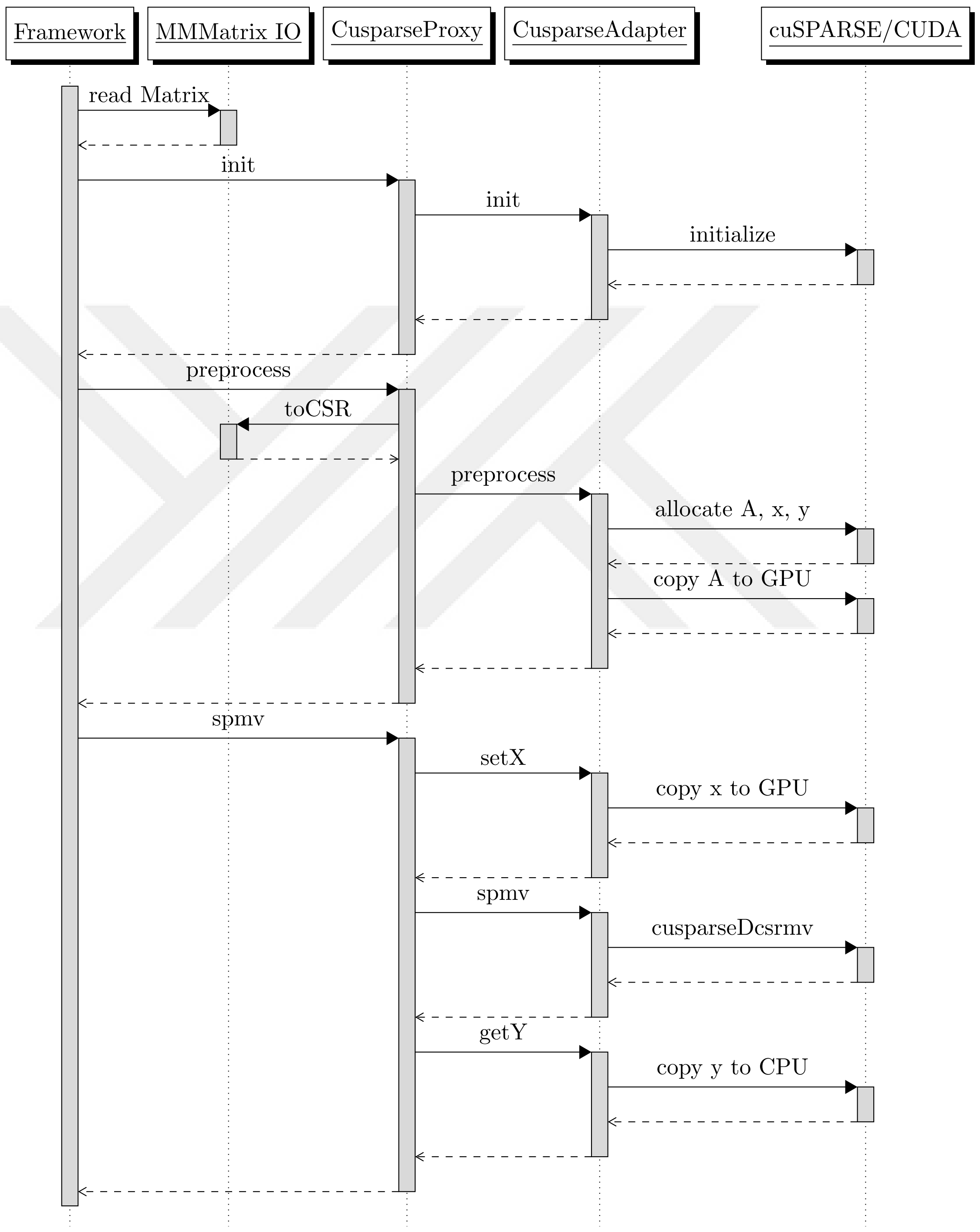


Figure 28: cuSPARSE SpMV execution flow.

operations on CPU and GPU, and is based on OpenCL. Recently it also supports CUDA and OpenMP backends [34] and even has Python and MatLab Interfaces.

The linear algebra operations are designed as a collection of algorithm functions and operators on data types. The SpMV in $y \leftarrow \alpha A \times x + \beta y$ form can be performed as `y = alpha * prod(A, x) + beta * y` in ViennaCL. Our integration uses the slightly simplified version `y += prod(A, x)` which computes $y \leftarrow A \times x + y$.

In order to fully support all available ViennCL backends —CUDA, OpenCL and, OpenMP— we organized our ViennaCL integration as proxy and adapter layers much the same as cuSPARSE and CUSP integrations, since supporting CUDA backend and others at the same time requires compile unit segregation. This also yields a similar execution flow as cuSPARSE and CUSP integrations with data exchange in-between CPU and GPU. Surely the data exchanges are considered as overheads, and do not affect the SpMV performance measurement.

6.3 Specializers

Specializer methods are also CSR-based methods that generate SpMV code specialized for a given matrix at runtime. They use this generated code to carry out the calculation. We integrate the following specialization methods from the Thundercat library [30, 14]. All of these specializer methods are derived from the `Specializer` interface.

- *CSRbyNZ:* This method groups the rows of the input matrix according to the number of nonzeros they contain, and creates a specialized code for each group by unrolling the innermost loop in plain CSR based SpMV [28, 14].

- *CSRWithGOTO:* Aktemur introduces this method as an intermediary method to *CSRLenWithGOTO* [35]. The main idea in this method is to improve the

CSRbyNZ method by unrolling the inner loop just once according to the length of the row with maximum number of nonzero elements, and then using this code again for the rows with fewer nonzero elements. In order to reuse the same code for the rows with fewer elements, this method calculates a jump address based on the row length and then starts execution from the calculated jump address, basically skipping instructions for longer rows.

- *CSRLenWithGOTO:* This method is also proposed by Aktemur as an improvement on top of *CSRWithGOTO* [35]. Compared to *CSRWithGOTO*, this method makes use of the *rowptr* array of CSR to pre-compute the jump addresses.

- *RowPattern:* This method analyzes the matrix and looks for patterns of nonzero entries. It groups rows by the patterns they contain, and generates specialized code for each group accordingly.

- *GenOSKI:* This method also analyzes the input matrix as in *RowPattern*; however, it looks for patterns in $r \times c$ sized blocks, rather then individual rows. It generates specialized code for each block pattern. We have integrated the 3×3 and 4×4 block size variants of this method.

- *Unfolding:* This method completely unfolds the entire CSR SpMV loops, generates a program that consists of individual calculation of every element of the output vector, without any loops.

For the specializer methods, the framework measures the time spent for runtime emission of code, so that it can also be used in evaluation of the performance.

CHAPTER VII

CONCLUSION

In this study we presented a framework for benchmarking SpMV methods. We designed our framework to be extensible with new SpMV methods in order for researchers to integrate their SpMV methods conveniently. Such a framework is needed to enable reproducibility of experimental SpMV evaluations and consistency throughout experiments. We have integrated several state-of-the-art SpMV methods and libraries not only to show the framework's extensibility, but also to provide a ready-to-use experiment environment. We make our framework available as open source to the use of researchers.

Bibliography

[1] D. Langr and P. Tvrdík, “Evaluation criteria for sparse matrix storage formats,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 428–440, Feb 2016.

[2] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, “Sparse matrix-vector multiplication on gpgpus,” *ACM Trans. Math. Softw.*, vol. 43, pp. 30:1–30:49, Jan. 2017.

[3] D. Merrill and M. Garland, “Merge-based parallel sparse matrix-vector multiplication,” in *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 678–689, Nov 2016.

[4] D. Guo and W. Gropp, “Optimizing sparse data structures for matrix-vector multiply,” *Int. J. High Perform. Comput. Appl.*, vol. 25, pp. 115–131, Feb. 2011.

[5] R. Vuduc, J. W. Demmel, and K. A. Yelick, “Oski: A library of automatically tuned sparse matrix kernels,” *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 521, 2005.

[6] A.-J. N. Yzelman and R. H. Bisseling, “A cache-oblivious sparse matrix–vector multiplication scheme based on the hilbert curve,” in *Progress in Industrial Mathematics at ECMI 2010* (M. Günther, A. Bartel, M. Brunk, S. Schöps, and M. Striebel, eds.), (Berlin, Heidelberg), pp. 627–633, Springer Berlin Heidelberg, 2012.

[7] J. Pichel, D. Heras, J. Cabaleiro, and F. Rivera, “Improving the locality of the sparse matrix-vector product on shared memory multiprocessors,” in *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 66–71, Feb 2004.

[8] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, “Optimizing sparse matrix-vector multiplication for large-scale data analytics,” in *Proceedings of the 2016 International Conference on Supercomputing*, ICS ’16, (New York, NY, USA), pp. 37:1–37:12, ACM, 2016.

[9] W. Liu and B. Vinter, “CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication,” *CoRR*, vol. abs/1503.05032, 2015.

[10] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, “Csx: An extended compression format for spmv on shared memory systems,” *SIGPLAN Not.*, vol. 46, pp. 247–256, Feb. 2011.

[11] A. Buluç, S. Williams, L. Oliker, and J. Demmel, “Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication,” in *In Proc. IPDPS*, 2011.

[12] M. Belgin, G. Back, and C. J. Ribbens, "A library for pattern-based sparse matrix vector multiply," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 62–87, 2011.

[13] D. Grewe and A. Lokhmotov, "Automatically generating and tuning gpu code for sparse matrix-vector multiplication from a high-level representation," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 12:1–12:8, ACM, 2011.

[14] B. Yilmaz, B. Aktemur, M. J. Garzarán, S. Kamin, and F. Kiraç, "Autotuning runtime specialization for sparse matrix-vector multiplication," *ACM Trans. Archit. Code Optim.*, vol. 13, pp. 5:1–5:26, Mar. 2016.

[15] "Intel® Math Kernel Library (Intel® MKL)." `https://software.intel.com/en-us/mkl`.

[16] "cuSPARSE Library Documentation." `https://docs.nvidia.com/pdf/CUSPARSE_Library.pdf`.

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

[18] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs," in *Intl. Workshop on GPUs and Scientific Applications*, pp. 51–56, 2010.

[19] A. Yzelman and R. Bisseling, "Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009.

[20] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2003.

[21] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[22] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: Efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, (New York, NY, USA), pp. 149–162, ACM, 2018.

[23] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 273–282, ACM, 2013.

[24] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 136–145, Feb 2015.

[25] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, (New York, NY, USA), pp. 233–244, ACM, 2009.

[26] D. A. Augusto, L. M. Carvalho, P. Goldfeld, A. E. F. Muritiba, and M. Souza, "A performance comparison of linear algebra libraries for sparse matrix-vector product," SBMAC, aug 2015.

[27] N. Kasmi, S. A. Mahmoudi, M. Zbakh, and P. Manneback, "Performance evaluation of sparse matrix-vector product (spmv) computation on gpu architecture," in *2014 Second World Conference on Complex Systems (WCCS)*, pp. 23–27, Nov 2014.

[28] S. Kamin, M. J. Garzarán, B. Aktemur, D. Xu, B. Yilmaz, and Z. Chen, "Optimization by runtime specialization for sparse matrix-vector multiplication," *SIGPLAN Not.*, vol. 50, pp. 93–102, Sept. 2014.

[29] NIST, "Matrix Market." `https://math.nist.gov/MatrixMarket/`.

[30] The Software Research Lab at Özyegin University, "Thundercat." `https://github.com/ozusrl/thundercat`.

[31] A. N. Yzelman and R. H. Bisseling, "Two-dimensional cache-oblivious sparse matrix-vector multiplication," *Parallel Comput.*, vol. 37, pp. 806–819, Dec. 2011.

[32] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, pp. 1–17, Mar. 1988.

[33] N. Bell and J. Hoberock, "Chapter 26 - thrust: A productivity-oriented library for cuda," in *GPU Computing Gems Jade Edition* (W. mei W. Hwu, ed.), Applications of GPU Computing Series, pp. 359 – 371, Boston: Morgan Kaufmann, 2012.

[34] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr, "ViennaCL—linear algebra library for multi- and many-core architectures," *SIAM Journal on Scientific Computing*, vol. 38, pp. S412–S439, jan 2016.

[35] B. Aktemur, "A sparse matrix-vector multiplication method with low preprocessing cost," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 21, p. e4701, 2018. e4701 cpe.4701.

VITA

Erdem Sarılı received Bachelor of Science in Computer Engineering from Izmir Institute of Technology Izmir, Turkey in 2010. He is working as Big Data Developer for SAP as of January 2019.