SPMM-BENCH: PERFORMANCE CHARACTERIZATION OF SPARSE FORMATS FOR SPARSE-DENSE MATRIX MULTIPLICATION

by

Patrick Flynn

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Science

Charlotte

2024

Approved by:

Dr. Yonghong Yan

Dr. Erik Saule

Dr. Dong Dai

©2024
Patrick Flynn
ALL RIGHTS RESERVED

ABSTRACT

PATRICK FLYNN. SpMM-Bench: Performance Characterization of Sparse Formats for Sparse-Dense Matrix Multiplication. (Under the direction of DR. YONGHONG YAN)

Sparse linear algebra operations are very important across many areas of science. Therefore, efficiently finding ways to perform these matrix computations is essential. The rapid development over the past few years in heterogeneous computing with or including accelerators has made finding new ways to process these operations important and challenging. However, the primary focus of this goal is through sparse matrix vector (SpMV) operations or sparse-sparse matrix operations (SpGEMM). Little attention is paid to sparse-dense matrix multiplication (SpMM), which is also used in important areas. In this paper, we propose to conduct a performance analysis of sparse matrix formats, specifically COO, CSR, ELLPACK, and BCSR. We aim to study these formats in SpMM operations in various conditions and provide a corresponding performance analysis. To facilitate this, we first propose a set of benchmarks for this evaluation. We then conduct a study of the formats themselves to provide an understanding of their behavior in various conditions.

DEDICATION

I dedicate this to my friends and family, who have always supported me and encouraged me all these years.

ACKNOWLEDGEMENTS

I would like to firstly thank my advisor, Dr. Yonghong Yan, for his guidance on this project. I have worked with him since my sophmore year of undergraduate, and I would like to thank him for his support and guidance through the years here at UNCC.

I would also like to thank Dr. Erik Saule, who co-advised me on this project and served as a committee member. I have also known Dr. Saule for a few years, and I would like to thank him for his guidance through my time here at UNCC.

I would also like to thank Dr. Dong Dai, my third committee member. I would like to also thank Dr. Ron Sass, who although did not directly advise me on this project, has given me a lot of guidance through my academic career here at UNCC, and much direction for the future.

Finally, I would like to thank my friends and family who have provided valuable support and encouragement throughout my time here at UNCC, and especially this past year as I have worked on this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Sparse linear algebra is widely used across many domains. It is used for various scientific applications [1], machine learning [2] [3], graph analytics [4] [5], and others [6]. Given the ubiquity and current prevalence of these fields, finding ways to optimize computation of sparse data sets is essential. The operations needed to perform these computations, especially matrix multiplication, are simple. The challenge lies in the representation and algorithms of these matrices.

To help facilitate such huge and expensive computations, GPUs and other accelerators are commonly used. This is in part due to the plateauing of Moore's law and the new trend towards heterogeneous computing as the method to solve this problem. The downside of these devices is they can be challenging to program. Their paradigms compared to conventional CPUs are quite different and challenging to reason about in context of parallelizing a sparse matrix.

The current challenge lies in the intersection of this problem. To help facilitate sparse computation, many formats have been devised to compress the matrices down to their nonzero values. Some formats such as CSR have proven to be very effective in conventional CPU environments. However, these formats are not always effective in parallel environments such as GPUs or even CPU acceleration. To solve this, various blocked sparse formats have been developed. These aim to create easy parallelization and improve memory locality with the trade off of padding the matrix, therefore doing some unnecessary computation. Depending on the matrix and the algorithm, these trade offs will ideally be limited. Regardless of the environment, there is no formula to choosing the right format. While generalities and patterns can be derived, choosing the right format depends on the matrix properties, the algorithm, the implementation,

and the device.

In studying this problem, there has been a lot of work done on sparse matrix vector multiplication (SpMV) and some work done on sparse matrix matrix (SpGEMM) multiplication [7] [8] [9] [10] [11] [12] [13]. Little study has been done on sparse dense matrix multiplication (SpMM) in terms of performance analysis and benchmarking.

Analyzing SpMM is important for general information needed by those who use the algorithm, but it is also important for those who are developing implementations. Much of the implementation focus for SpMM and other sparse operations is on run-time libraries. However, there is interest and desire in implementing sparse operations in the compiler space. Implementing operations at compile time is most desirable as it introduces predictability early on.

In this paper, we aim to present a study of the blocked-sparse matrix formats with particular focus on SpMM. We choose the ELLPACK and BCSR formats as they are the most common formats in the parallel space. We first aim to implement ELLPACK and BCSR in a compile-time environment to get a sense of how they compare against other general formats already implemented and tested. Following this, we will conduct a study of these formats with focus on SpMM. We will develop a set of benchmarks to help facilitate our study and hopefully facilitate the studies of others who wish to study SpMM in any environment with any format. Finally, we will provide an evaluation of the results we have obtained and and attempt to draw some conclusions from the data we have gathered. To gain a thorough understanding of the formats and the operation, we will run the benchmarks on a variety of inputs on various architectures and environments. To present a clear picture, we will arrange our data into nine studies.

We aim to make two main contributions here. Our first contribution is the benchmark suite itself. We aim to make our benchmark suite easily extensible for a wide variety of sparse matrix formats, while providing a set of common functions for per-

formance analysis and cost model analysis. The default kernels we will provide encompass the main general matrix formats and blocked matrix formats. Our second contribution is the performance analysis we perform with these benchmarks. While SpMM is algorithmically similar to SpMV, there are key differences especially with parallelization. We aim to perform a comprehensive study of SpMM itself and SpMM in context of the formats and parallelization. We hope this will be useful for those trying to perform SpMM operations and are looking for ways to best optimize their code through algorithm and format.

The rest of this paper will be divided as follows: Chapter 2 contains the background and motivation of our work. Chapter 3 contains an analysis of the existing literature on what has been studied in this space. Chapter 4 describes the implementation of our benchmark suite. Chapter 5 contains our evaluation. Finally, chapter 6 describes trends in our evaluation and lists future improvements and evaluations we wish to make.

CHAPTER 2: BACKGROUND & MOTIVATION

2.1 Sparse Formats

Despite the ubiquity of sparse formats developed over the years, there is no universal or ideal format. The format is dependent on the problem you are trying to solve, the hardware and execution environment you are targeting, and the properties of the matrix. The biggest factor in what ultimately decides which format is used is memory access. While the matrix itself may not change, the sparse format can dictate how memory is accessed in the course of fetching the nonzero elements. However, a specific sparse format even in this case is not always inherently good or bad. A format that is really bad in terms of memory access for one matrix may not necessarily be bad for another matrix.

Many of the classic sparse formats such as coordinate (COO) and compressed sparse row (CSR) perform really well in general CPU environments, but do not lend themselves well to the parallel environments that have become dominant at the end of Moore's law. Sparse operations are run on very specific pieces of hardware such as GPUs and vector machines. Multi-threading is often used as well. Parallelizing a problem requires an understanding of the data and a clear, self-evident way to partition it. If the partitioning method is not obvious, time and computation is wasted as an algorithm attempts to figure it out. In cases where you may not even have the opportunity to figure it out as you go, such as with GPU and vector machines, these formats may be outright prohibitive to parallel processing.

To remedy this, formats have been devised specifically for parallel environments. These formats are known as blocked formats. Traditional formats only represent the nonzero elements in as compact a method as possible (COO and CSR both represent

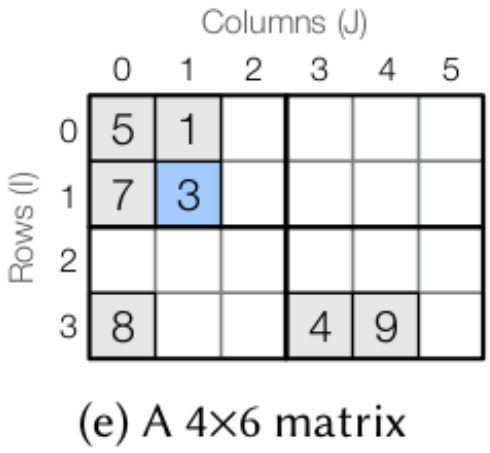

Figure 2.1: A dense matrix

nonzero elements only, but CSR compresses COO). This introduces the problem of parallelization we spoke of above. To solve this, the blocked formats pad the compressed representations to create blocks around the nonzero elements that can be more easily parallelized. Many formats have been developed, but the most common are ELLPACK[14], BCSR[15], and Blocked-ELL[16].

2.2 Sparse Format Processing

Choosing the ideal format is difficult, and the choice is not easier with blocked formats. While ELLPACK (ELL), BCSR, and Blocked-ELL (BELL) all use the same underlying principle of blocking, their methods of doing so are quite different. The choice of which format to use largely comes down to matrix properties. Consider the matrix shown in figure 2.1.

ELL is the simplest format to implement, but it is also the format that most easily degrades performance. ELL builds an array that contains the nonzero column indices for every row in the matrix. Every row will have a constant number of columns, meaning the size of each row is dictated by the row in the matrix with the most nonzero elements. Padding is done on all the other rows to make them the same length as the longest row. Ideally, the padding is done in proximity to the nonzero

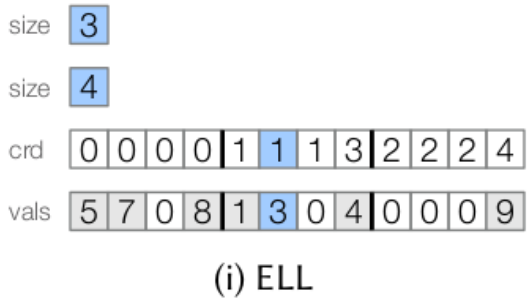

Figure 2.2: The dense matrix above in ELL format

elements to introduce spatial locality. The algorithm for computing this is very simple and easily vectorizable. The downside is this format only works with matrices that have similar and consistent numbers of nonzero elements in each row. Once you have a row that has vastly more elements than the rest of the matrix, your performance degrades substantially. See figure 2.2 for an example of the dense matrix above in ELL format.

BCSR and BELL attempt to solve this problem. BCSR is basically an extension of CSR to allow for blocking. Of the three formats, this format allows for the most control over how the elements are blocked. This can alleviate excessive padding, but care must be taken not to create blocks too small to not be worth parallelizing. In any case, this format is also the most expensive in terms of loops and format-specific computation. BELL is halfway between ELL and BCSR. It partitions the matrix into groups of rows, and then performs ELL padding by block. This gives you less control than BCSR, but ideally with less looping and computation. See figure 2.3 for an example of the dense matrix above in BCSR format.

The blocked sparse formats highlight the importance of knowing the properties of the matrix you are computing. Understanding how the data is laid out in the matrix helps determine the efficacy of a blocking method and whether to use blocking at all.

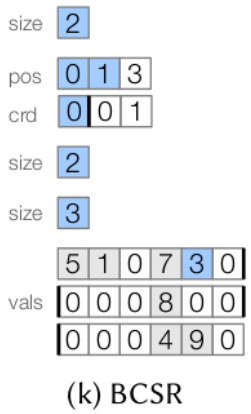

Figure 2.3: The dense matrix in BCSR format

2.3 Sparse Matrix Dense Matrix Multiplication

Sparse dense matrix (SpMM) multiplication is a common operation used in machine learning [2] [3], recommendation systems [17], and other scientific environments. SpMM can be utilized in these fields in two ways. The first and most obvious is when you have a sparse matrix and a dense matrix that need to be multiplied. The second utilization is through batching vectors. It is often necessary to multiply several vectors by the same matrix. Although this would usually be an SpMV problem, these vectors can be "stacked" and multiplied with the sparse matrix as SpMM. This is potentially more efficient than performing several SpMV operations (at the very least, you save the time of having to format your sparse matrix over and over). Benchmarks and performance studies exist for SpMV, SpGEMM, and other operations, but no general benchmark for SpMM that we know of exists, and little work has been done on performance characterizations. Because of the relevance of the fields and the ways SpMM can be utilized, having a stable baseline for SpMM operations is necessary.

In matrix matrix multiplication, you receive as input matrix A and matrix B, and receive as output matrix C. Every row of matrix A is multiplied by every column of matrix B and saved to matrix C. Below is the general algorithm for matrix matrix multiplication given a matrix of size $N \times P$. Matrix matrix multiplication algorithms

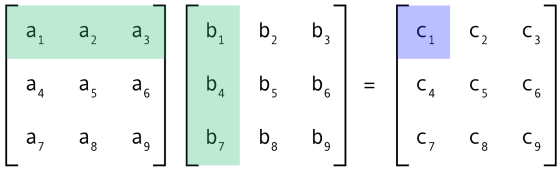

Figure 2.4: Matrix Matrix Multiplication

with sparse matrices have a similar structure. A visual representation of matrix matrix multiplication is seen in figure 2.4.

```
#Input: matrices A and B
let C = new array[p*n]
For i from 1 to n:
    For j from 1 to p:
        Let sum = 0
            For k from 1 to m:
                Set sum <- sum + A[i][k] * B[k][j]
        Set C[i][j] <- sum
Return C
```

The primary challenge with matrix matrix multiplication is the data intensity. In matrix vector multiplication, each row of matrix A is multiplied by vector B. It is exactly the same as in matrix matrix multiplication, except that only one column instead of several exists. This means memory access is sequential. Each row of matrix A is loaded, and once it is used, it is not needed again. In matrix matrix multiplication, each row of A is loaded and used several times. If the matrix was small enough, it could be kept in the cache, but in real world matrices, this is never the case. Additionally, in matrix vector multiplication, the vector is laid out as a row in programs, making memory access much faster. In matrix matrix multiplication, the column has to be gathered from matrix B, which almost certainly thrashes the cache.

SpMM is very similar algorithmically to SpMV. The only difference is the presence of the k loop and the extra operations that it entails. SpMM is conceptually similar to SpGEMM, but unlike SpMV, the algorithms are varied and different. SpGEMM is the multiplication of two sparse matrices. Multiplying two sparse matrices involves the additional complexity of dealing with sparse formats, possibly multiple sparse formats. The algorithms here will vary greatly depending on the combination of formats being used. Benchmarks exist for SpMV and SpGEMM, and both operations have been well studied with different sparse formats [7] [8] [9] [10] [11] [12] [13].

We believe an evaluation of SpMM is necessary for a few reasons. First of all, we do not believe that performance will necessarily be relative to SpMV because of the redundant, data intensive memory operations needed to gather each column from matrix B ([5] identifies this problem as well). The extra loop and the memory problems present different challenges for parallelism than what we would expect in SpMV. Secondly, much of the current work on SpMM that we are aware of focuses on specific problems and is focused towards one format (most commonly CSR) and one architecture (generally the GPU). We are not aware of any general performance characterizations of SpMM across the difference sparse formats on different architectures. We aim to make this contribution through the development of our benchmark suite and through our initial performance study.

CHAPTER 3: RELATED WORK

[18] and [9] present studies of sparse matrix operations and formats in an attempt to create a machine learning framework for selecting the ideal sparse matrix format. While they focus on several formats, they consider the blocked formats, especially ELLPACK. Both authors present metrics for evaluating the input matrices in question. These metrics are used by the machine learning frameworks to determine the format. As an example, one metric presented is the ELL ratio, which is the maximum number of nonzero elements in a row compared to the average number of elements per row. A high ratio would indicate that ELL is probably not the best format for the matrix in question. However, both of these papers focus on SpMV.

[9] additionally focuses on heterogeneous and parallel environments, including GPU computation and CPU parallelization. It presents excellent information and provides a set of metrics applicable to our work, but they focus solely on SpMV. No evaluation of SpMM is done, nor is there an indication in future work that it will be.

[8] and [10] presents studies and evaluations using the blocked sparse formats we chose here. However, [8] also focuses on SpMV, while [10] focuses on SpGEMM. [7] also focuses on SpMV, but narrows the focus to BCSR.

[19] presents a set of sparse format abstractions that are applicable across general formats such as COO and CSR and specific formats such as the blocked formats like ELL and BCSR. Building upon this work, [20] and [21] implement an LLVM/MLIR-based compiler directly using the TACO formats. Because the COMET compiler introduces a completely compiler-based solution, full runtime analysis can be done, and future benchmarking work can influence compiler development for blocked formats. While not a follow-on work to either COMET or TACO, Triton [22] presents

a similar example of how LLVM can be leveraged for GPU code generation and used for reasoning about complex abstractions.

[23] present a study of SpMV and SpMM kernels on the Intel Xeon Phi. This paper was a study focused on the Intel Xeon Phi processor rather on sparse matrix formats or their properties. Nevertheless, the experimental setup is very good and well laid out, with a number of properties and experimental parameters. We used the authors' setup as inspiration in designing our benchmark suite. However, our focus is on the sparse formats rather than the hardware, so adjustments were made for that.

[24], [5], [25], and [17] present studies of SpMM for various applications. However, these and other studies generally focus on the GPU and do not consider multiple formats. [24] and [5] only focus on variations of CSR, and [17] focus on the compressed sparse column (CSC) format.

CHAPTER 4: IMPLEMENTATION

4.1 Design Rationale

Many benchmark suites, including suites for sparse operations, are written in C or Fortran. The benefit of using these languages lies in their performance. However, both languages are relatively simple and low-level, so they are not always great for generic programming.

Sparse formats all differ in their data structures, formatting, and multiplication algorithms. For example, COO requires an integer and three arrays. CSR also requires an integer and three arrays, but one of these arrays is much shorter than the other two. ELLPACK requires two integers and two arrays. More advanced formats such as BCSR may require even more structures. Sparse matrices are generally stored in a COO-like format, meaning that all formats require a preprocessing function to format the sparse data into the specific representation being used. In a benchmark suite, the goal is to have a core library to handle as many preprocessing and data collection tasks as possible so the focus can be on the algorithms. Implementing functions in C to perform these tasks with the idiosyncrasies of each format is certainly possible, but this would be tedious to implement, and difficult for others to extend for additional formats.

To address these issues, we wrote the suite in C++ and used an object oriented approach in the implementation. C++ provides the advantages of the performance of C and Fortran (when done properly), with the benefit of more abstract data structures. With our design, the benchmark suite is designed as a core library that includes all the performance collection and reporting methods. The entire library is defined as a C++ class which defines formatting and calculation functions that will be specific

to every format. By default, the library defines the COO format. All other formats will format their structures based on the COO representation. A custom format will simply extend the class, and re-implement the calculation and formatting functions. This design makes the interface easy to implement and read. Benchmarking is done from within the suite, so any potential overhead is eliminated.

The second benefit to this approach is implemented formats can be partially extended to test things such as different formatting methods or different multiplication algorithms. We made extensive use of this feature in our evaluation. In our evaluation, we tested serial, CPU parallel, and GPU versions of each format. The initial implementation of each format has a serial multiplication algorithm. For the parallel and GPU versions, we simply re-implemented the calculation function for each version. We tested this with CUDA, and re-implementation for CUDA algorithms is possible.

The third benefit lies in manual optimizations, which we demonstrated later in our evaluation. In our library, some features such as the k loop can be adjusted at runtime to study different performance characteristics. However, this removes information from compile time that could potentially be used by the compiler to make optimizations. The C++ template feature can be utilized to essentially trick the compiler into performing these optimizations while still being generic enough to maintain one algorithm (the same compile time trick can be utilized in C, but this would require copying and pasting the function for every value we wish to embed). We will provide an example of this in the last study of our evaluation. We believe there are potentially other areas in which these features could be utilized.

4.2 Formats

The initial format provided by the suite is the COO format. The core suite also provides implementations of CSR, BCSR, and ELLPACK. For each format implementation, we provide serial, parallel, GPU, serial transpose, parallel transpose, and GPU

transpose kernels. The parallel and GPU kernels are implemented using OpenMP.

The decision to use C++ in our implementation was partly influenced by the fact that we already had implementations of the formatting and multiplication algorithms for ELLPACK and BCSR. Formatting a large matrix into the ELLPACK or BCSR format is potentially very time consuming since padding is required. When the matrix is padded, we want to make sure to include the zeros as close to the non-zeros as possible in order to maintain spatial locality. Our initial algorithms were very expensive, to the point of not being usable. We solved this problem for ELLPACK and partly solved it for BCSR by using the containers and algorithms within the C++ standard library (especially maps) that allowed us to cache data and easily query it. There is still room for improvement with BCSR, but ELLPACK formatting time is comparable to CSR and COO.

4.3 Metrics

Our output metrics are a combination of runtime data, matrix data, and parameter information. Parameters are input as command line arguments, which the suite defines and parses. We currently have parameters for controlling the number of times the calculation function will be called; the thread count for parallel kernels; the block size for applicable block formats (currently just BCSR); and the length of the k-loop. A debug flag is also provided for development purposes.

The primary performance measurement is reported in floating-point operations per second (FLOPS). The suite reports FLOPS, mega-FLOPS, and giga-FLOPS. This is calculated against the average run time of the multiplication algorithm, which is calculated by the benchmarking function. The formatting time and total runtime is also reported. The suite has a built-in verification function for verifying the accuracy of the calculation. We originally tried to implement this using a pure matrix-matrix multiplication algorithm, but this took too long. We decided instead to use the COO multiplication algorithm for verification.

For the matrix properties, rows, columns, number of non-zeros, maximum columns, average columns, column ratio, variance, and standard deviation are reported. Rows and columns represent the size of the input matrix, and number of non-zeros represent the non-zero values of the input matrix. All other metrics are related to the number of non-zero values (columns) per row. This is an important metric to understanding blocked sparse format performance. ELLPACK is the simplest blocked format because it creates a single block based on the row with the most non-zeros. However, if you have one or a few rows with a lot of non-zeros, you will end up with very poor performance. For all formats, this may also impact parallelism and memory performance. The maximum column metric gives the number of non-zeros in the row with the greatest number. The average column metric gives the average number of non-zeros per row. The column ratio metric is a the ratio of maximum columns to average columns. The variance and standard deviation represent the number of non-zeros across all the rows. We will show this data for our matrices in the evaluation section.

CHAPTER 5: EVALUATION

In this evaluation, we perform a total of nine studies. Our first study lists our input matrices and their attributes. Our second and third studies present overviews of the formats across our input matrices from two different perspectives. Our fourth study looks at CPU level parallelism. Our fifth study looks at the impact of adjusting our k loop. Our sixth study focuses on the BCSR format. Our seventh study takes a look at the CPU and GPU architectures we used. Our eighth study compares our algorithms against the cuSparse algorithms. Our ninth study looks at taking the transpose of matrix B and the impact that has on performance. Our tenth and final study takes a look at the impact of some manual optimizations we made to the algorithms.

5.1 Environment

In this evaluation, we study four formats: COO, CSR, ELLPACK, and BCSR. For each format, we run a serial, parallel, and GPU version of each kernel. For the eighth study, we have transpose variations of each of these three kernels. We perform this evaluation on the Arm and x86 architectures. The entire benchmark suite is compiled with Clang 16 with version 12.3 of the Nvidia toolkit. We run our benchmark suite on 14 matrices, all from the SuiteSparse matrix collection. We will look at the matrices in more depth in the next section.

Our Arm machine is an Nvidia Grace Hopper Superchip (we refer to it in this paper as Grace Hopper or Arm). Grace Hopper has 72 Nvidia Grace CPUs, based on the Arm architecture. It has an Nvidia H100 Tensor Core GPU. The system as a whole has 574 GB of RAM. Our x86 machine (referred to as Aries) is based on the AMD EPYC 7413. The machine has two AMD EPYC Milan 7413 CPUs with

24-Core, hyperthreaded up to 48 Threads. It has 504 GB of RAM with an NVIDIA A100 Ampere GPU.

In our evaluation, our goal was to consider CPU and GPU performance for each matrix and format. For both the CPU and GPU, we used OpenMP as our runtime. OpenMP on the CPU generally yielded good performance, and we had no issues with the runtime. Unfortunately, this was not the case for the GPU. We were not able to write CUDA kernels, so we decided to use OpenMP for the advantage of ease and portability. However, the OpenMP target offload runtime had a lot of issues. It worked perfectly on our Grace Hopper machine, but the exact same version of Clang and Cuda on our Aries machine did not work. At first, it randomly failed, but eventually it always failed. We tried multiple versions of Clang, but this yielded no results. While a code issue is certainly possible, given that the same code with the same compiler and runtime worked on Grace Hopper, we believe it could be an environment issue. We did eventually find that some matrices worked with the runtime on Aries, so we limited our evaluation to those matrices.

In the evaluations that follow, all runtime results are reported in MFLOPs. As a result, higher is better on all our graphs. Except for the K-loop study, all benchmarks were run with k set to 128. Unless otherwise noted, all OMP (parallel) kernels were run with 32 threads. Except for the BCSR study, all BCSR kernels were run with a block size of 4.

5.2 Matrix Properties

Table 5.1 shows the properties of each matrix we evaluated. We present this table primarily to understand the properties of the matrices we are using. However, we also predict that matrices with a low column ratio should perform the best with the blocked sparse formats especially on the GPU.

All the matrices we used were square, so the "Size" column refers to the number of rows and columns respectively. The "Non-zeros" column refers to the number of non-

sparse values across the matrix. The "Max" column refers to the maximum number of non-zeros in a row in the matrix. The "Avg" (average) column shows the average number of non-zeros per row. The "Ratio" (column ratio) column presents a ratio of the "Max" to the "Avg" column. The "Variance" and "Std Dev" (standard deviation) columns show the variance and standard deviation of the number of columns per row respectively. The column ratio is probably the most useful column since it presents the clearest image of how the non-zeros are distributed by row.

Table 5.1: Properties of Each Matrix

	Size	Non-zeros	Max	Avg	Ratio	Variance	Std Dev
2cubes_sphere	101492	874378	24	8	3	14	3
af23560	23560	484256	21	20	1	1	1
bcsstk13	2003	42943	84	21	4	197	14
bcsstk17	10974	219812	108	20	5	79	8
cant	62451	2034917	40	32	1	54	7
cop20k_A	121192	1362087	24	11	2	45	6
crankseg_2	63838	7106348	297	111	2	2339	48
dw4096	8192	41746	8	5	1	0	0
nd24k	72000	14393817	481	199	2	6652	81
pdb1HYS	36417	2190591	184	60	3	753	27
rma10	46835	2374001	145	50	2	772	27
shallow_water1	81920	204800	4	2	2	0	0
torso1	116158	8516500	3263	73	44	176054	419
x104	108384	5138004	204	47	4	313	17

5.3 Study 1: Formats

In this study, we give an overview of all formats across all input matrices divided by architecture and kernel type. In all versions, we set the k value to 128, and set the BCSR block size to 4. Our goal for this study is to see which format in each environment (serial CPU, multicore CPU, GPU) does the best overall. Figure 5.1 shows the Arm results, and figure 5.2 shows the Aries results.

Let us consider the serial results. On Arm, the CSR format generally did best, scoring the highest for over half. BCSR did surprisingly well, scoring the highest for five matrices. In general, the single core computations on Arm average around 5k

MFLOPs. On Aries, the results were almost evenly divided between COO and CSR. The block formats did not perform well serially. The average computational speed for Aries was around 7k MFLOPs.

Next, let us consider the parallel results. We set the thread count to 32 for this study. On Arm, COO generally did the best in a parallel environment. On Aries, the results depended on the matrix. They were fairly evenly divided across all formats, although the results tended slightly towards CSR and COO. In general, the parallel to serial speedup on Arm was 5-6x faster, and stayed fairly consistent across the matrices. For Aries, the speedup was around 4x faster, but in some cases they did substantially better, with one matrix doing almost 15x better.

Finally, let us consider the GPU results. On Arm, the GPU results were dependent on the matrix. In general, there was not a huge speedup, but a few matrices did almost 2x better. The GPU results for Aries were strange, so we will not consider those. We suspect there is a runtime bug or an environment issue since the same code performed so poorly when it ran at all.

5.4 Study 2: Kernels

In this study, we look at all versions of each format across all matrices. This study is divided by format, and then by architecture. Our goal here is to see which form of each kernel (serial CPU, parallel CPU, or GPU) does best for each format. For this study, we set k to 128, we run the parallel kernels with 32 cores, and we set BCSR to a block size of 4.

We will consider the Arm benchmarks first. Figure 5.3 shows the Arm results. On Arm, across all formats, the results were almost always evenly divided between CPU parallelism and the GPU. The best versions generally average 10-30k MFLOPS depending on the format.

Let us consider the Aries benchmarks. Figure 5.4 shows the results. We were unable to consider the GPU results on Aries. However, the CPU parallelism almost

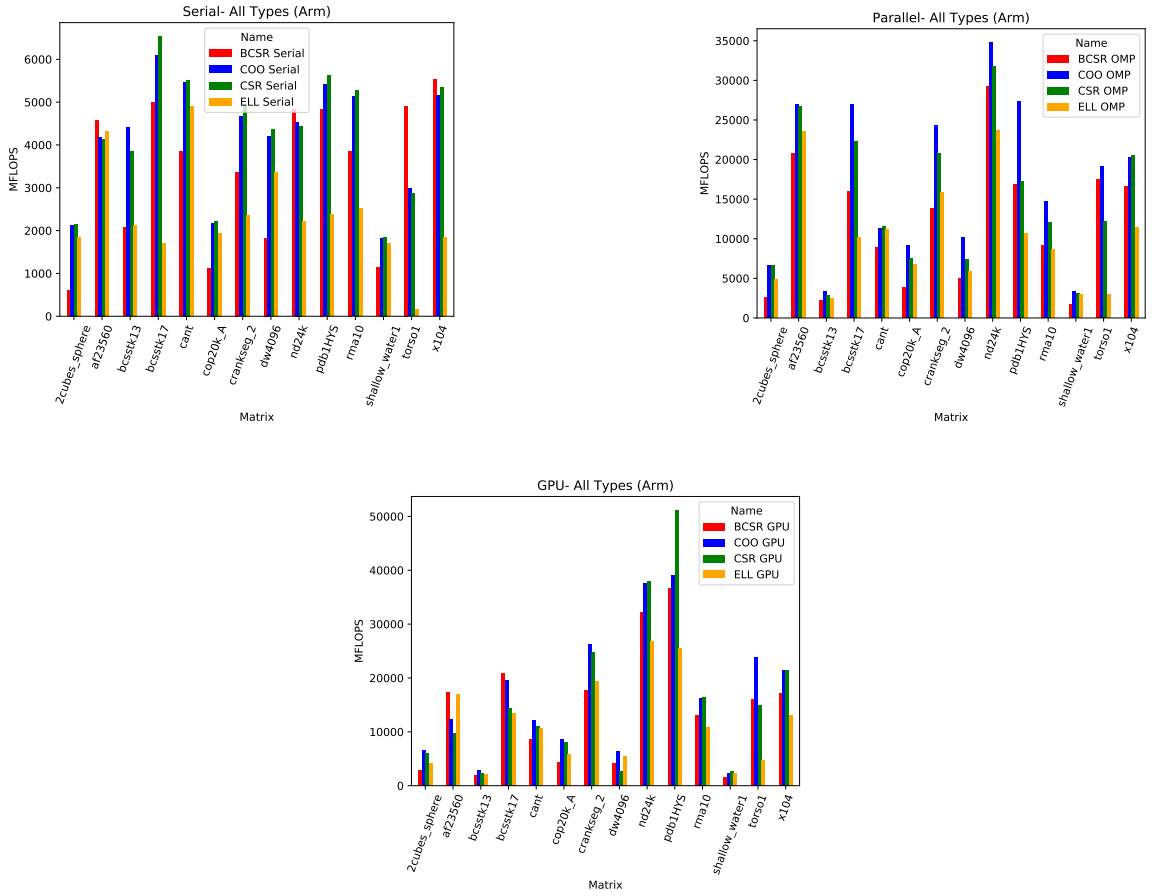

Figure 5.1: Study 1: All Formats- Arm

always did better. The best versions generally averaged around 15-30k MFLOPs.

There were a few instances of the serial kernels doing the best on both architectures. However, this was confined to just a few matrices, and in no case did they substantially outperform the parallel versions. This generally occurred with the COO and CSR formats.

5.5 Study 3: CPU Parallelism

In this section, we will consider the performance impact of the thread count for the parallel kernels across each format. All kernels were run with a thread count of 8, 16, and 32. The k loop was set to 128. Our study is divided by architecture, and subdivided by format. We evaluate our data based on our kernel doing well with a

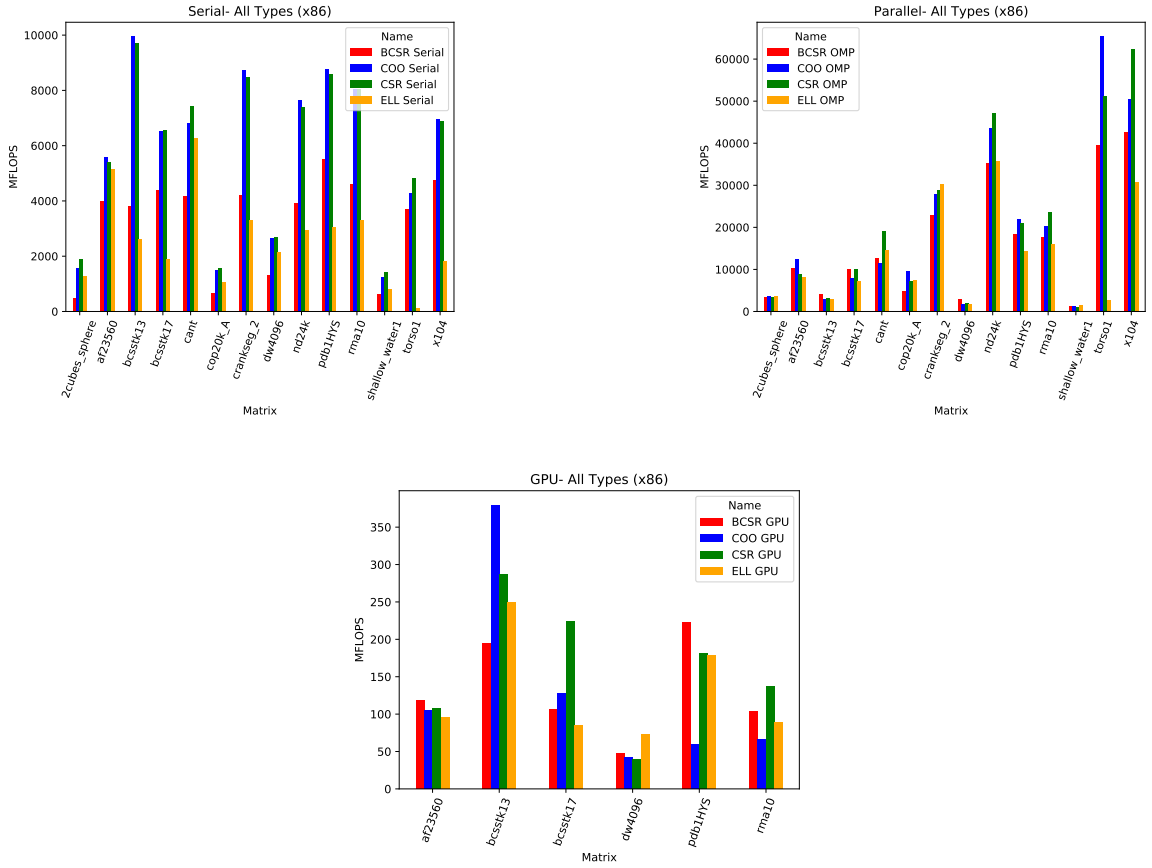

Figure 5.2: Study 1: All Formats- x86

high thread count, a low thread count, or it generally being equal across all thread counts. Our goal for this study is to see the impact of thread count for our formats and matrices.

We will start with the Arm results. Figure 5.5 shows the Arm results. In general, all formats did the best with a high thread count on Arm. We did not observe any significant difference between the performance of the blocked formats versus COO/CSR. In general, all formats averaged between 10-20K MFLOPs in performance, hitting 30-35K on the high end.

Let us examine the Aries results. Figure 5.6 shows the Aries results. In general the results across all formats were divided between higher and lower thread counts. Depending on the matrix, some formats did well with a high thread count, while

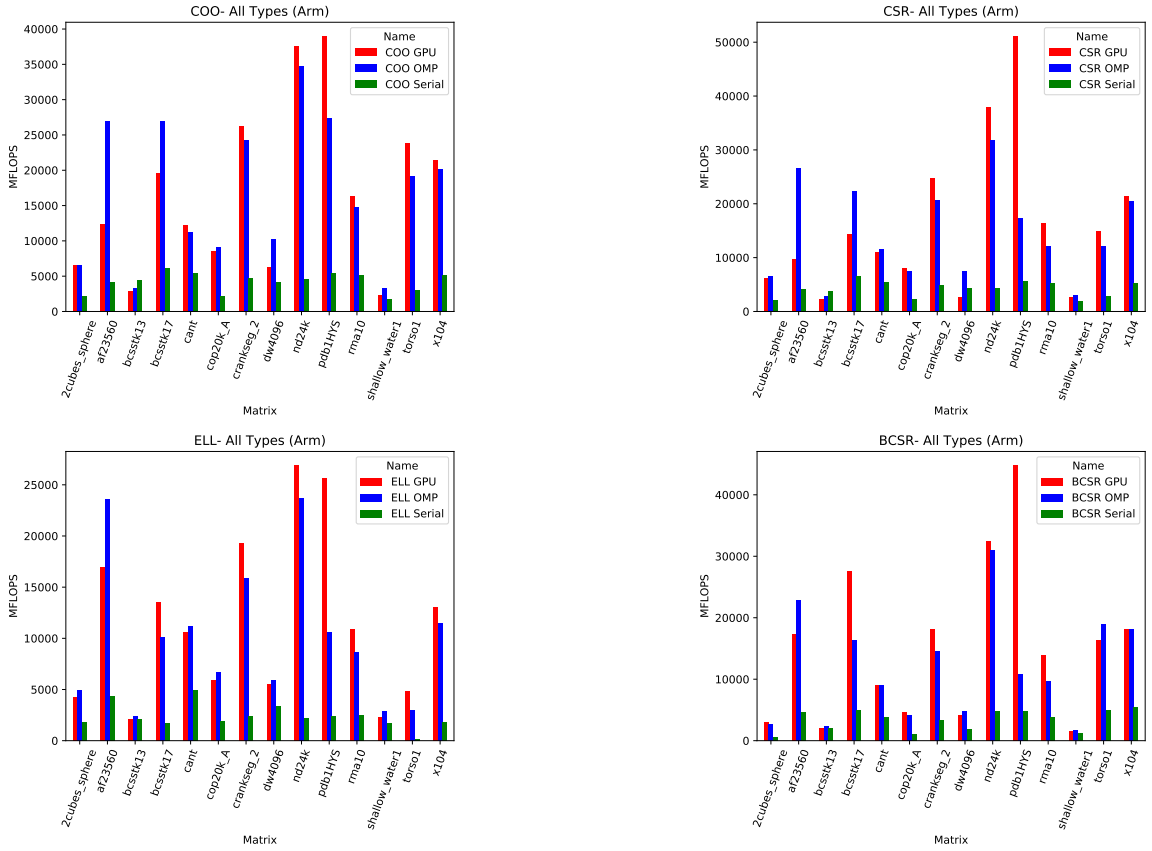

Figure 5.3: Study 2: Best Form of Each Format- Arm

others did well with a lower thread count. BCSR did the best overall with a high thread count. CSR came in second. In general, all formats average around 15K MFLOPS in performance, hitting 40-60K on the high end.

5.5.1 Study 3.1: Best Thread Count

This section is a follow-up study to study 3. Because of our results from study 3, we were curious to test the overall effect of thread count on each format for each matrix. In this study, we modified our benchmark suite to include a feature that will run the benchmark for a user-designated set of thread counts. The suite will iterate through the thread count list, and pick the best thread count for the given inputs. For this study, we kept k at 128, and set our thread list input to 2, 4, 8, 16, 32, 48, 64, and 72. Because our machines differed slightly in their core counts, we chose 72 as

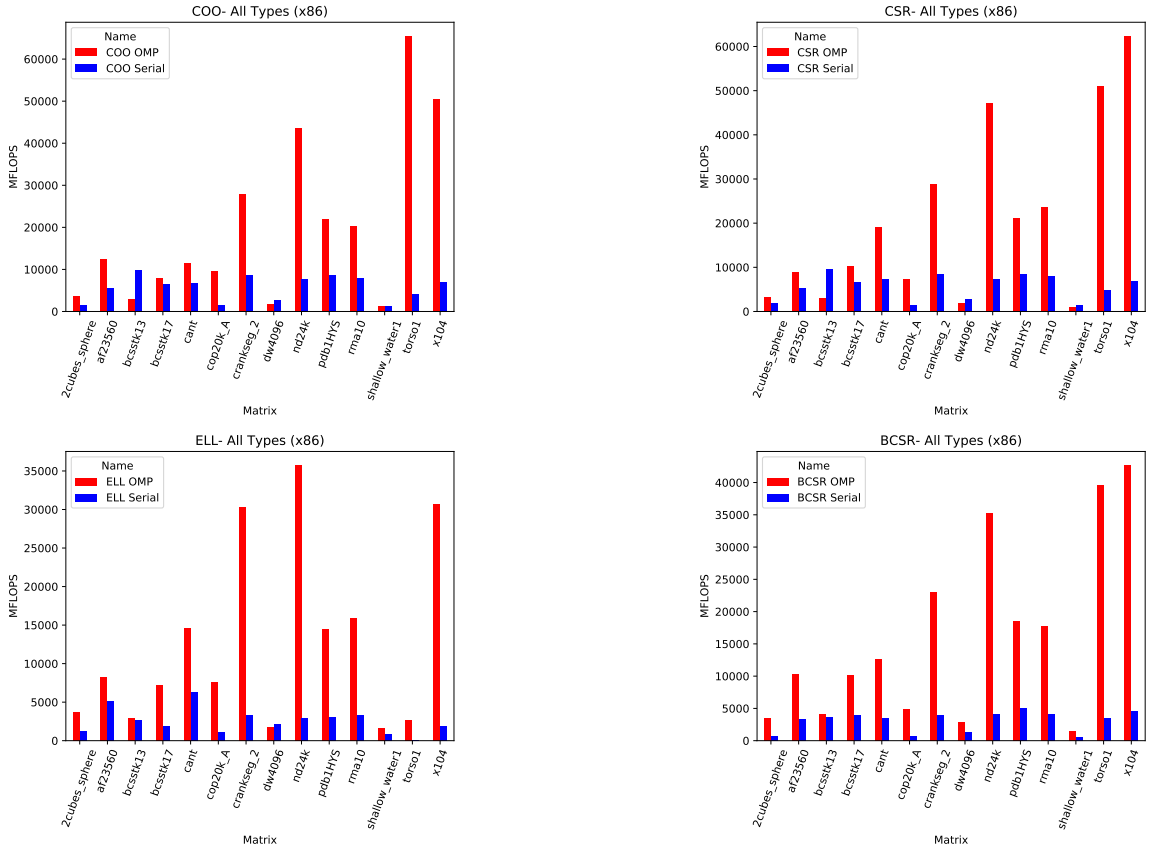

Figure 5.4: Study 2: Best Form of Each Format- x86

our consistent upper bound. To evaluate this study, we looked at how many matrices of each format did best on the thread count of 72.

Let us consider the Arm results first. Figure 5.7 shows the results. Out of 14 matrices, COO achieved the 72 core count on 10 matrices; CSR for 9; ELL for 12; and BCSR for 6. These results are generally consistent with our previous observations in study 3 of COO and ELL doing well on higher thread counts, and our observation in general that using high thread counts yielded the best results.

Figure 5.8 shows the results for Aries. Aries was interesting because although 96 cores were technically available, there were only 48 physical cores (the 48 cores were hyperthreaded to 96 cores). As a result, the Aries results appeared to trend towards less cores. In reality, high thread counts worked up to the number of physical cores (32 to 48, accounting for other factors). However, 10 of the 14 matrices had at least

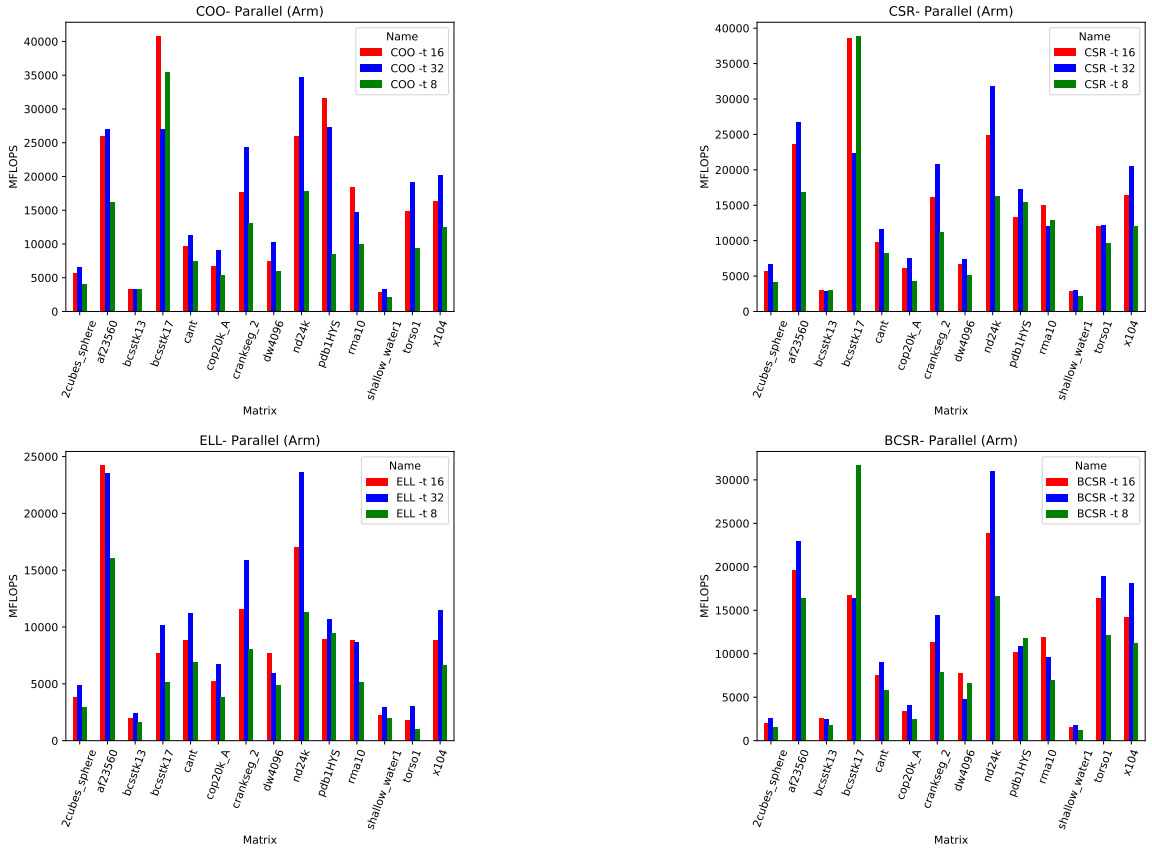

Figure 5.5: Study 3: Parallelism- Arm

one format that did very well with hyperthreading. BCSR in particular seemed to do the best with hyperthreading.

5.6 Study 4: K-Loop

In this section, we consider the impact of adjusting the bounds of the innermost loop in the matrix multiplication algorithm, commonly known as the k loop. Adjusting this controls how much of the multiplication we wish to do. For this study, we consider the CPU parallel version of each format with a thread count of 32. We use k values of 8, 16, 64, 128, 256, 512, and 1028. Figure 5.9 shows the Arm results, and figure 5.10 shows the Aries results.

This study is based on the assumption that we can pick a k value in our problem. In theory, this leads to better performance since we can limit the amount of multi-

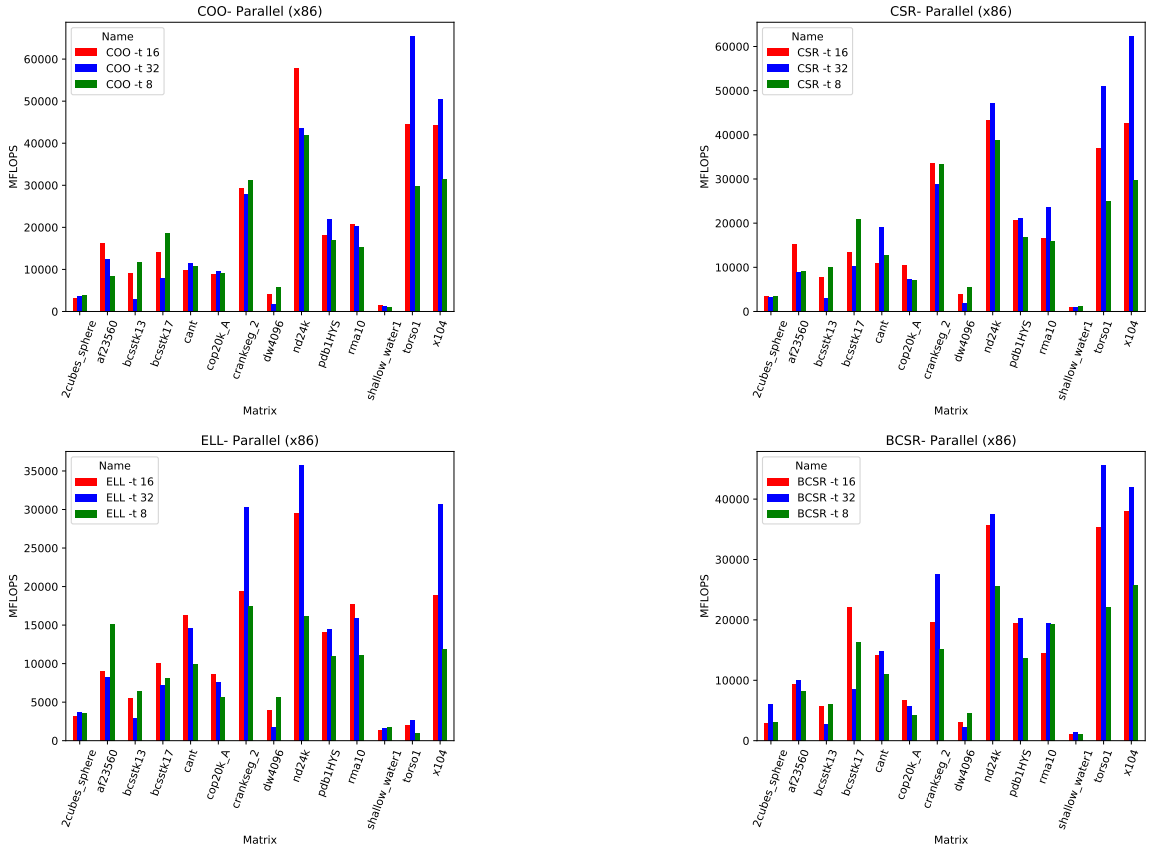

Figure 5.6: Study 3: Parallelism- x86

plication done. Additionally, we might expect that the performance increases of this will cap at a certain point since more work is done. On Arm, for the values of k we picked, we did not observe this to be the case. In general, a higher value of k seemed to lead to more performance.

For Aries, there were several instances were performance for k capped, usually around the 512 mark. Given some of the peculiarities of the x86 architecture like hyperthreading and CPU throttling, there could be multiple explanations for this. More investigation is necessary, but adjusting k on x86 might require an adjustment to the number of threads used.

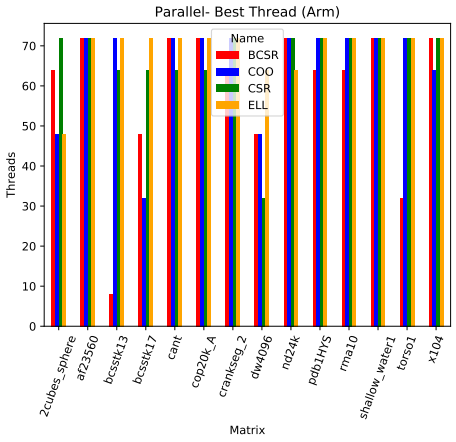

Figure 5.7: Best Thread Count (Arm)

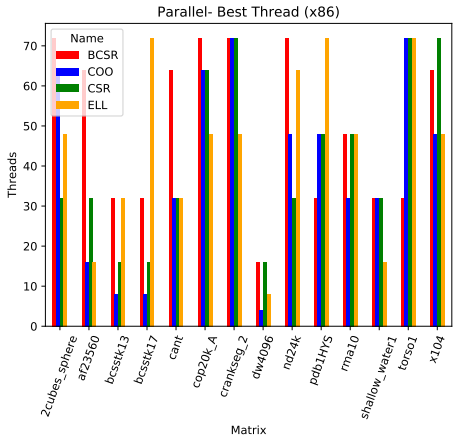

Figure 5.8: Best Thread Count (Aries)

5.7 Study 5: BCSR Study

In this study, we only consider the BCSR format. BCSR allows us to configure the size of the sub-blocks in our matrix. Our goal here is to see what effect changing the block size has on performance. We will consider this in a serial, parallel, and GPU environment. Figure 5.11 shows the Arm results, and figure 5.12 shows the Aries results.

As expected, the serial versions did increasingly worse as the block size got bigger. This remained the case on both Aries and Arm. The overall performance averaged around 5k MFLOPs. The parallel versions also tended towards a lower block size. However, in a few cases, increasing the block size yielded better performance. This was the case two times on Arm, and four times on Aries. Overall performance averaged 15k MFLOPs with the 25-35k on the high end for Arm. On Aries, it averaged 20k MFLOPS with the high end being around 40-50k. We only considered the GPU results on Arm. In general, the trend of a lower block size remained the case, but it did well with a higher block size on a few additional matrices.

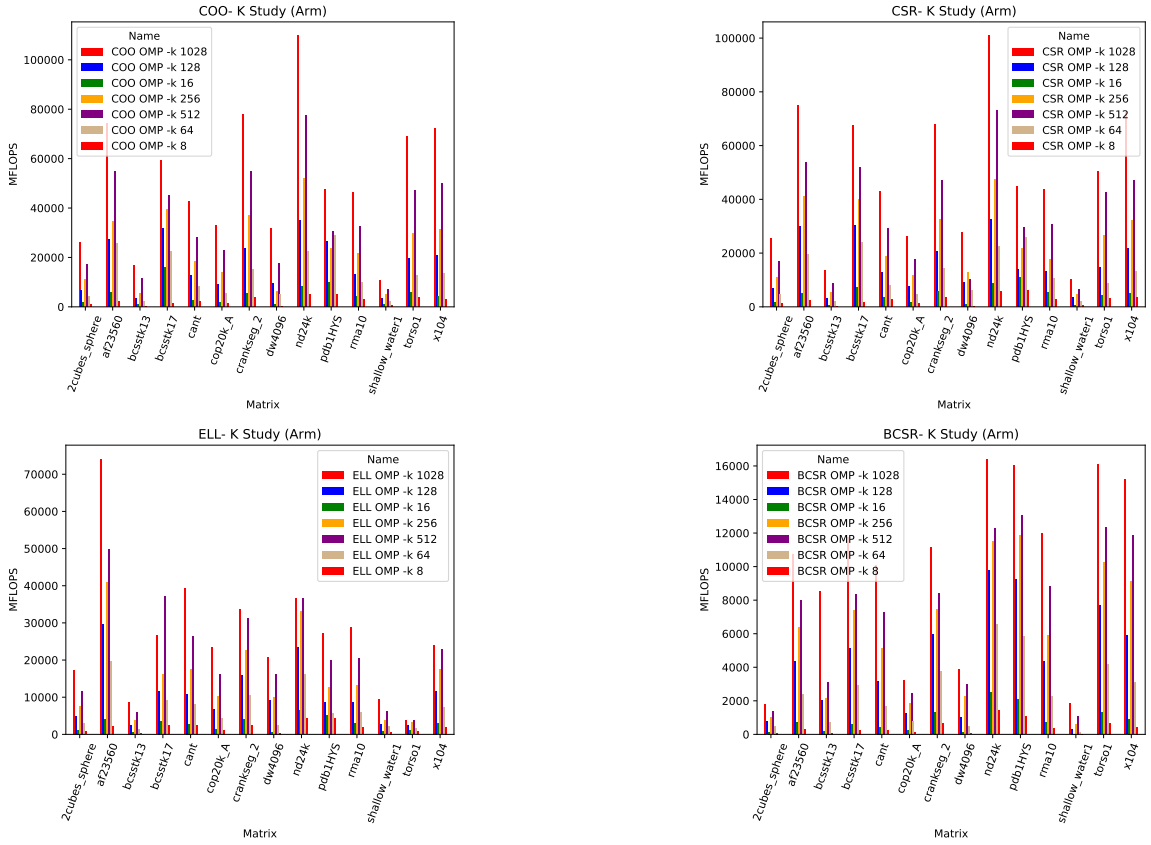

Figure 5.9: Study 4: Setting -k (Arm)

5.8 Study 6: Architecture Study

In this section, we consider how each format performs on different architectures. We evaluate the serial versions of each format on our Aries and Arm machines to evaluate the single core performance of each. Figure 5.13 shows all formats across both architectures, and figure 5.14 shows the BCSR formats for all block sizes on both architectures. For this study, we used all three of our BCSR block size configurations for more data.

For COO, CSR, and ELLPACK, the Aries versions all performed better. In general, the overall performance was around 5k MFLOPS, except for ELLPACK, which was 3k MFLOPS. The opposite was true on BCSR. All three versions of BCSR performed better on Arm. The performance averaged 5k, 4k, and 1.5k MFLOPs for 2, 4, and 16 block sizes respectively. For COO, CSR, and ELL, the Aries results were significantly

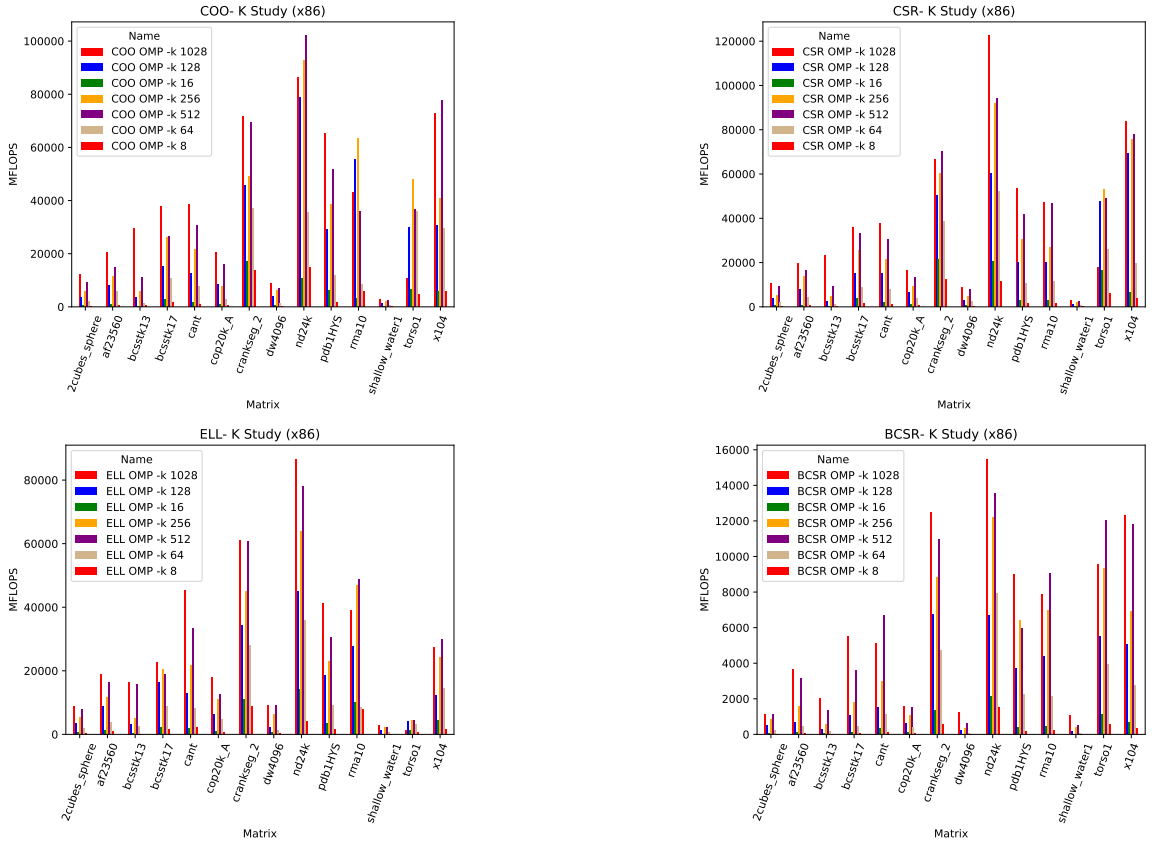

Figure 5.10: Study 4: Setting -k (x86)

better than Arm. On BCSR, which did better with Arm, the results for Aries were still close. For pure individual core performance, Aries seems to yield better results across the board than Arm.

5.9 Study 7: cuSparse Study

In this section, we compare our COO and CSR GPU performance with the CUDA cuSparse library. The purpose of this study is primarily to compare Nvidia libraries with our algorithms. We select COO and CSR since they are the only two formats provided by cuSparse that provide a direct comparison to our formats. We also intend for this test to demonstrate CUDA interoperability with our benchmark suite. For the test, we do not set k. We also used only 9 of our 14 matrices. We omitted the other 5 because they required more memory than what the device could support. On Aries, we had to omit five more matrices because of the OpenMP target offloading

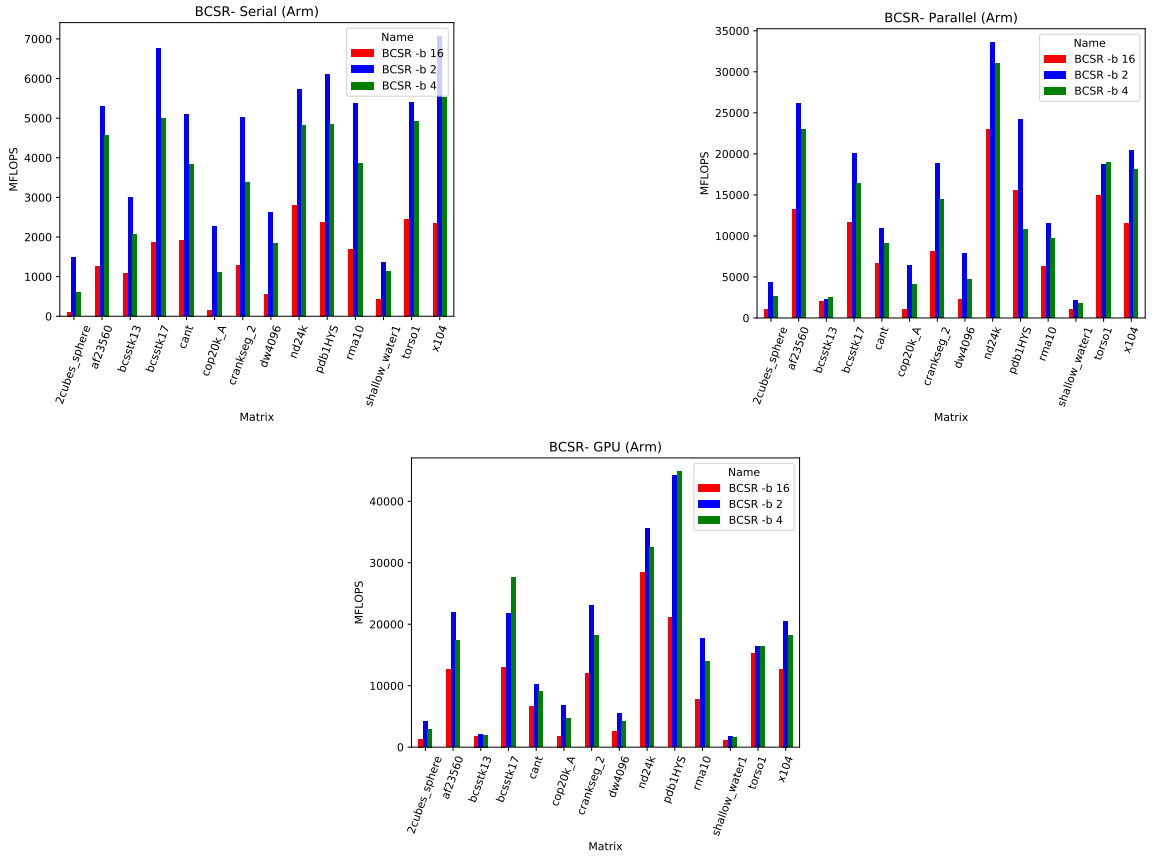

Figure 5.11: Study 5: BCSR (Arm)

issues.

Figure 5.15 shows the Arm results. For COO, cuSparse did better on all but two of the matrices. For CSR, it did better on all but one. Figure 5.16 shows the x86 results. Surprisingly, of the three matrices we tested, the OpenMP versions did better. However, because the sample set is so small, and because of the other issues we have had with the runtime, we hesitate to draw conclusions.

We do not find the results particularly surprising since the OpenMP target offload library is not known to do well on the GPU. While OpenMP offload provides convenience and in many cases, far better performance than any CPU equivalent, it is probably not the best solution when optimal performance is required.

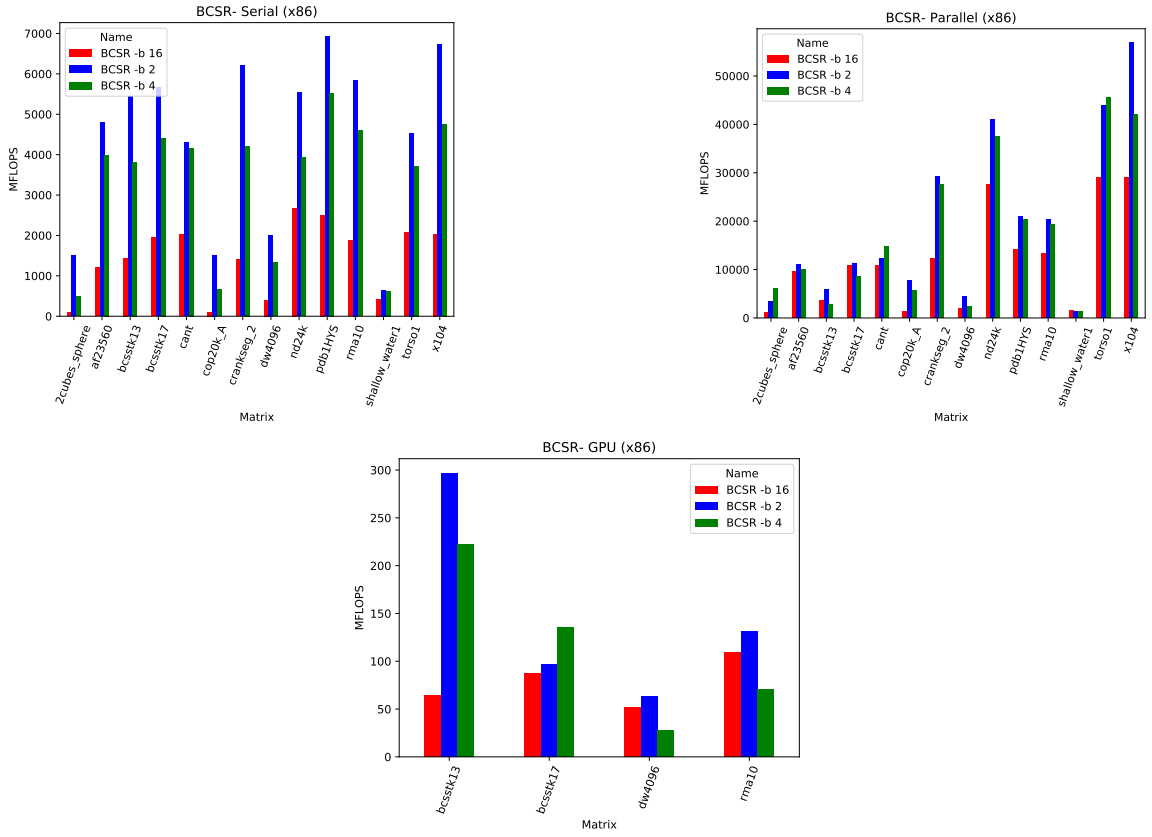

Figure 5.12: Study 5: BCSR (x86)

5.10 Study 8: Transpose Study

In this section, we consider what performance impact transposing matrix B has on performance. In theory, transposing matrix B should yield performance improvements since it allows B to be accessed in a linear manner (row by row as opposed to column by column). However, there is a potential performance cost because B has to be transposed before we can perform the calculation. Our goal is to see whether or not transposed matrix multiplication with the cost of transposing B yields any performance improvements.

For this study, we only considered the parallel results since doing the transpose serially and then multiplying would have been very time consuming, and realistically not something that would be done in the real world. Figure 5.17 shows the Arm results, and figure 5.18 shows the Aries results. The results here are interesting in that

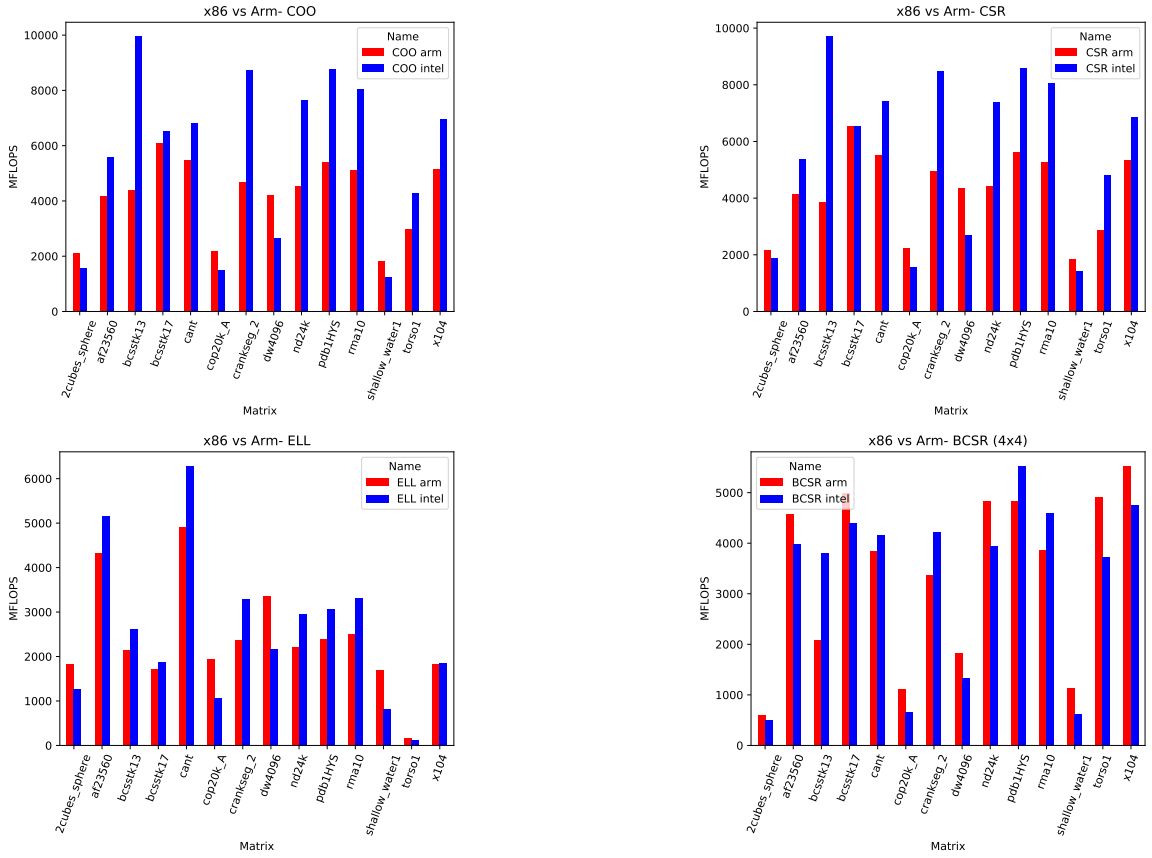

Figure 5.13: Study 6: All Formats (Arm vs x86)

only a few matrices have a noticeable speedup on either architecture. These matrices tended to be consistent across architectures, which indicates that the position of the sparse values could influence whether or not transposing works. For example, in theory, accessing a matrix by row would be faster, but if the nonzeros were spread across the rows with a wide gap, it would still thrash the cache. Depending on the problem, there is also the potential overhead of having to transpose matrix B.

In dense matrix multiplication, we would expect a performance increase because of the access patterns for matrix B. However, since we are working with sparse matrices, we are using a different access pattern in which B is accessed linearly. Transposing B effectively requires us to use the same access pattern as dense multiplication, which thrashes the case and leads to worse performance in most cases.

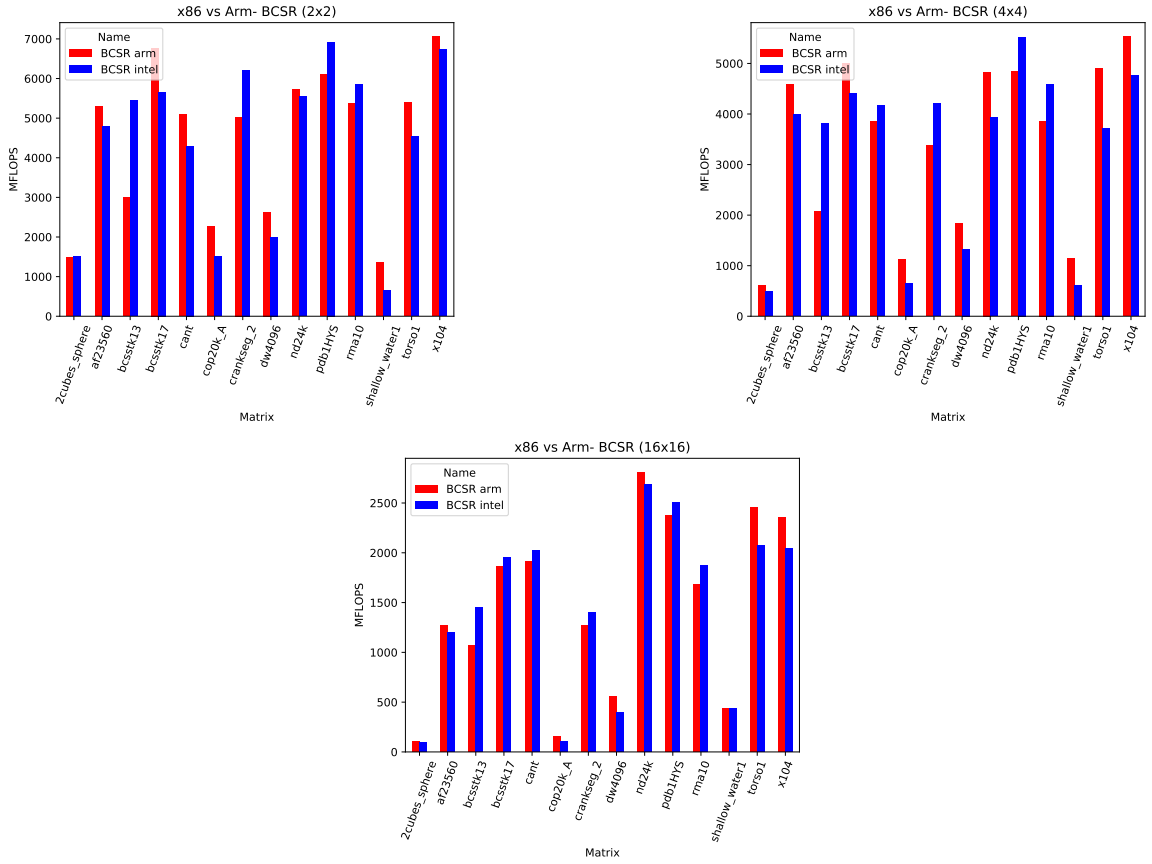

Figure 5.14: Study 6: BCSR: Block Sizes 2, 4, 16 (Arm vs x86)

5.11 Study 9: Manual Optimization Study

Near the end of the evaluation, we made some manual optimizations to the individual calculation kernels. We moved the values load from outside the k loop, and we used C++ templates to hard-code the value of k in the loop. In theory, this should create a slight performance improvement since the number of loads are reduced. Prior to doing this, we examined the code generation of a serial kernel, and observed that SIMD instructions were not being used mainly since k was not known at compile time. After making these changes, we notice that SIMD instructions were much more and better utilized. More loop unrolling was also done.

We ran the benchmarks with some of these improvements, but we did not have time to run and evaluate the entire suite with the entire dataset again, and additionally we

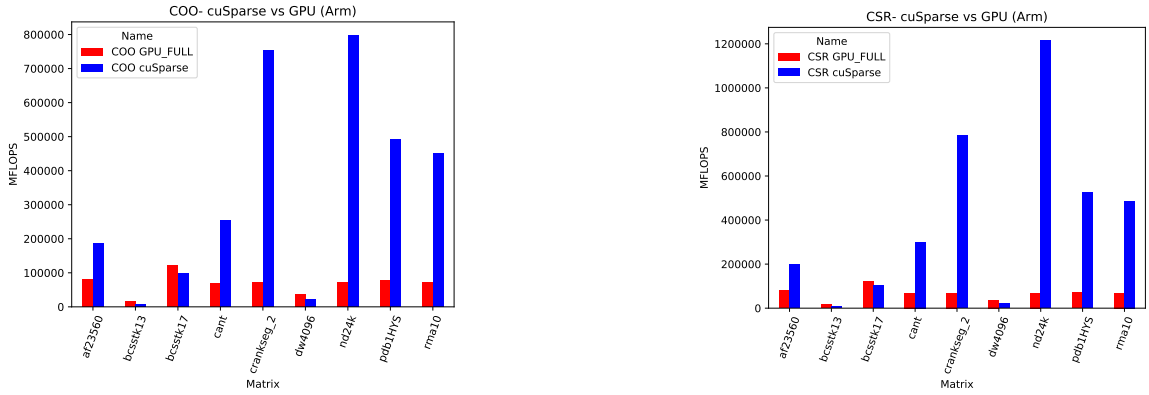

Figure 5.15: Study 7: cuSparse vs OpenMP GPU (Arm)

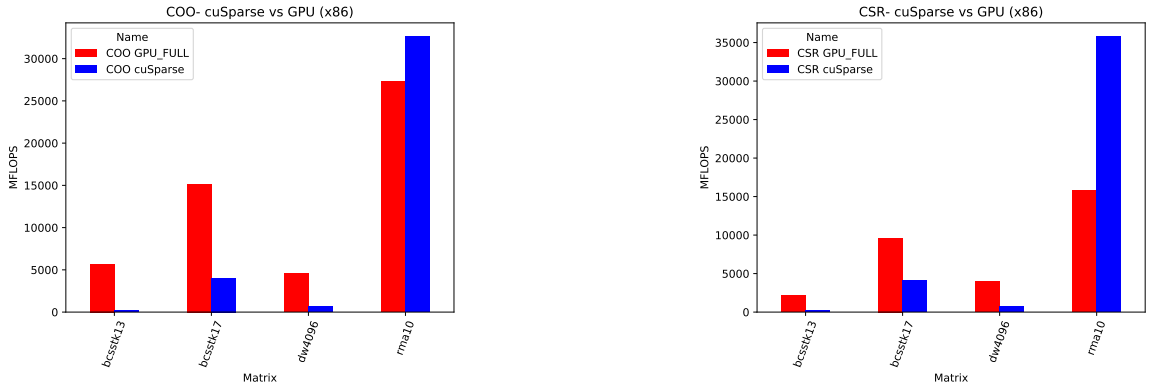

Figure 5.16: Study 7: cuSparse vs OpenMP GPU (x86)

began encountering OpenMP target offload issues on our Arm machine. Therefore, we decided to use what data we could collect and evaluate it separately compared to our original data. Figure 5.19 shows the manual optimizations for the serial and parallel kernels on both architectures.

The serial Arm versions did not lead to any positive performance improvements for any format except COO. However, the opposite was true on Aries. Almost every format showed positive performance increases with the modifications. However, a total of six formats showed negative performance impacts. The parallel Arm versions showed positive performance improvements for 5 of the fourteen matrices with the COO format, but 9 had a negative performance impact. This was only the case for the COO format (except for one isolated CSR format that also performed negatively

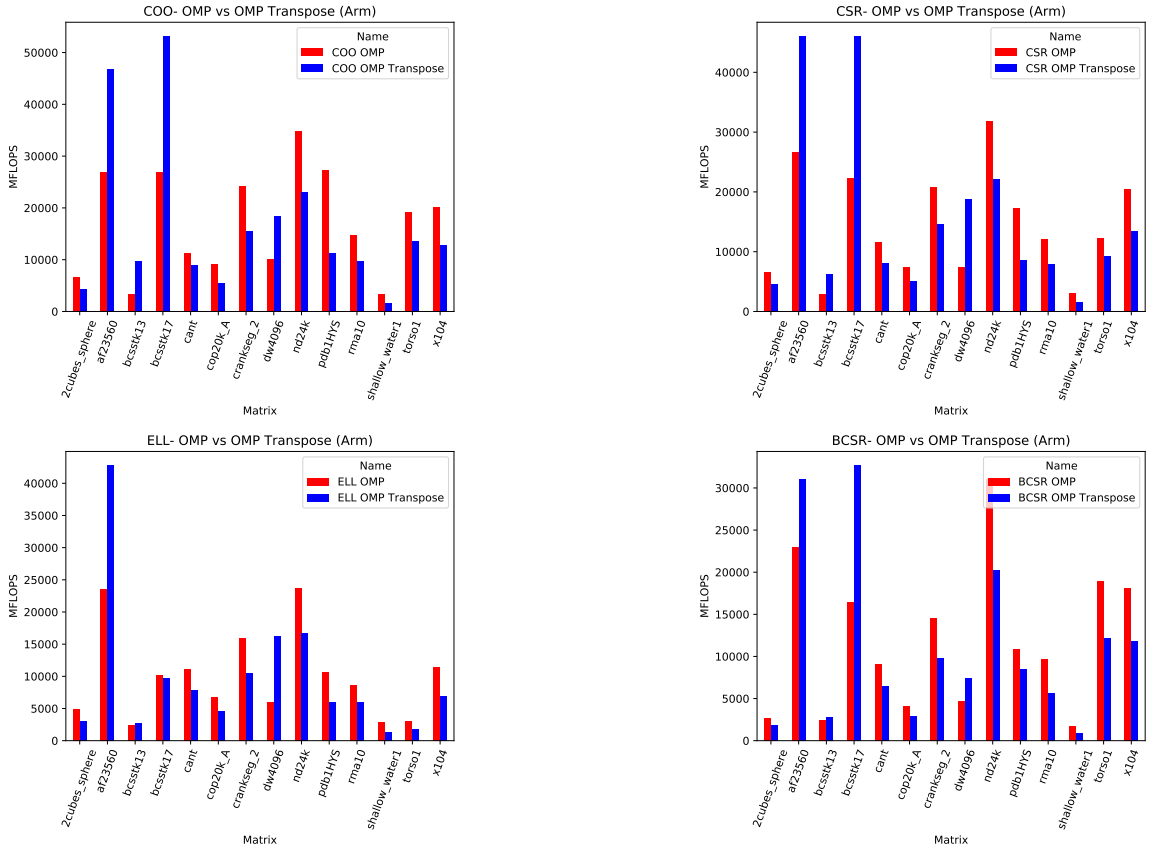

Figure 5.17: Study 8: Transpose (Arm)

with the modifications). On Aries, the results were evenly divided between positive and negative performance impacts for all the formats. Note that we do not consider the parallel BCSR in this data because we made a change to which loop is parallelized. This change clearly made the overall performance worse, so we do not attribute this to the modifications were are considering in this study.

Because this study is more of a compiler problem than an algorithm problem, it is hard to draw definitive conclusions with this study, especially for the two CPU parallelism versions. We believe it is best to consider the serial versions when making a determination. On Arm, the changes were always neutral or better. On Aries, the improvements were generally positive. Because the methods here reduce the loads and give more information to the compiler, it is probably best to make these manual optimizations when possible.

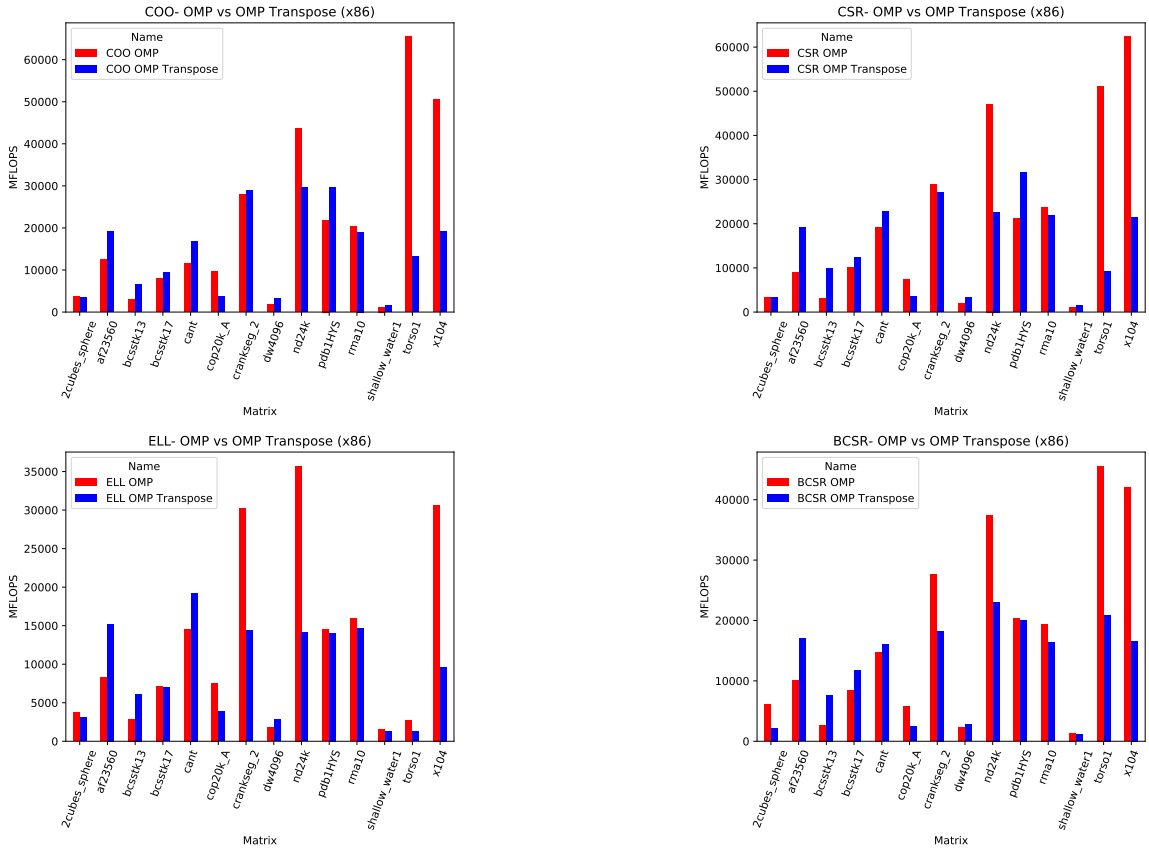

Figure 5.18: Study 8: Transpose (x86)

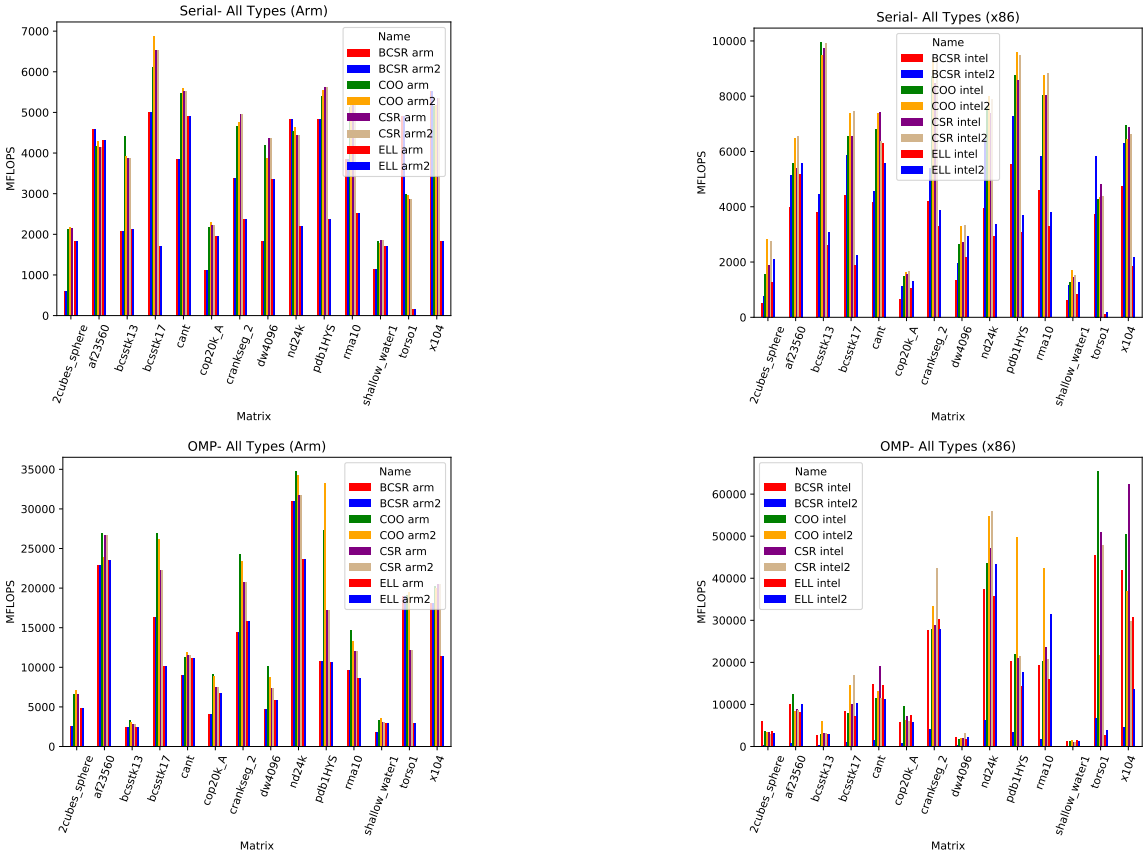

Figure 5.19: Study 9: Manual Optimizations (Arm and x86)

CHAPTER 6: CONCLUSION

6.1 Evaluation Conclusion

While this initial evaluation presented a number of studies on several matrices, we acknowledge that a more rigorous study would be needed to draw firm conclusions. We will discuss what this could look like in the next section. However, we can still draw some general trends from our evaluation. We hope these trends along with the future work we discuss in the next section can provide some starting points for additional study.

Our GPU studies are the weakest part of our evaluation owing to the issues with the OpenMP runtime. From the data we could collect, we observed that GPU offloading is often better, and depending on the matrix, even dramatically better. Because the OpenMP runtime is considered to have poor performance, it is likely that if a matrix does better with the runtime, it would have even greater performance improvements when using CUDA or some other solution.

Our most thorough study was with CPU parallelism. On Arm, all formats performed about equally in terms of how well they worked with a higher thread count and their average performance. On Aries, COO did not do well, but CSR and the blocked formats performed well. We also noticed some interesting trends when choosing a thread count. OpenMP defaults to using all threads available on the system. On Arm, this is generally the best solution regardless of format. On Aries, this is not always the best solution. x86 CPUs use hyperthreading, which basically turns one physical core into two virtual cores, doubling the number of CPUs available from the operating system's perspective. This does not mean better performance, however. Our studies showed that many matrices tended to do best with a thread count closer

to the number of physical cores present. However, there were a few instances of certain matrices gaining huge performance increases with hyperthreading. Interestingly, this generally happened with the blocked formats.

It is widely known in the existing literature on sparse formats that there is not one best format. The best format depends on your problem and your environment. In our studies, we found that COO and CSR often did very well, and in many cases, they did better than BCSR or ELLPACK regardless of the environment. While not always true, the difference between COO and CSR was often not big. When performance between COO and CSR for a given matrix is comparable or even slightly less in the case of CSR, CSR may be better since it has a smaller memory footprint. ELLPACK and BCSR generally did the best in parallel environments, whether it be on the CPU or the GPU. Interestingly, the blocked formats seemed to do quite well on Aries. When hyperthreading worked, the blocked formats generally did the best.

BCSR is the only format we tested in which the formatting could be modified. In theory, this would be the strength of BCSR in parallel environments since you can create block sizes conducive to the peculiarities of your matrix. In this study, we kept constant block sizes to draw some general trends. In general, a smaller block size tends to do better. Instances were a larger block size yielded positive performance improvements indicates that it is more conducive to the layout of your matrix. However, if the block size is too small, you should use CSR since the algorithm is much simpler. We did notice that Aries seemed to do a better job with larger block sizes than Arm.

Arm is new to high performance computing, meaning that it is still being evaluated and developed. x86 by contrast has been developed, tested, and studied for decades. The Grace Hopper Arm machine we used is one of the most recent Arm HPC machines, so we hoped to gain a sense of its performance in this work. In terms of individual core speed, our Aries machine was still faster, sometimes significantly

faster. In terms of parallelism, Arm seemed to be better. As long as the workload of each thread is still great enough to offset the overhead of threading, more Arm cores appear to equal more speed. However, in instances where an Aries machine has a lot of physical cores and hyperthreading works well for the specific problem, the total speed was often greater than Arm.

6.2 Blocked Sparse Conclusion

In the previous section, we made a general conclusion of all formats compared to the results. In this section, we will focus specifically on the blocked sparse formats.

As expected, ELL and BCSR do not perform well in serial environments. They perform best in CPU parallel environments and on the GPU. We were unable to conduct GPU tests in depth, but from the GPU data we could collect, we observed that when given the option, the blocked formats bring at least comparable performance on the GPU. We were surprised with how well the blocked formats especially BCSR did on Aries even when hyperthreaded. COO and CSR generally only performed well up to the number of physical cores, but when a format did well in a hyperthreaded environment, it was usually one of the blocked formats.

We compared the instances where the block formats did well with the data we collected in table 5.1. As we expected, ELLPACK generally did best with matrices that have a low column ratio. BCSR generally did best with a low column ratio, but there were a few matrices with higher column ratios that it did well on (indicating that the blocking is potentially effective here). However, the data in our table presents an overly simplistic view of the data. A low column ratio does help, but spatial locality of the non-zeros is ultimately best. If the data is sparse and widely scattered, any blocking will become irrelevant because of the cache misses. We looked at some of the other metrics in the table including the variance and standard deviation of columns per row, but we could not draw any pattern here. Understanding your matrix data is probably best done with a graphical representation.

6.3 Future Work

This paper represents the first version of the benchmark suite. Our initial goal was to create a core benchmark suite that could be extended for additional uses in the future. In the process of implementing it and testing it through initial evaluations, we have identified some features and improvements we would like to do in the future.

6.3.1 Additional Formats

Our initial implementation provides the COO, CSR, BCSR, and ELLPACK formats. While this provides a diverse starting point, we recognize that other formats have been proposed and evaluated in recent literature with promising results. The two most commonly cited that we would like to implement next are the Blocked-ELLPACK (BELL)[16] and the CSR5 formats[26]. Our original implementation contained an initial draft of the BELL format, but we ran into several issues with it, and had to put it on hold for now.

6.3.2 BCSR Formatting Algorithm

We fortunately began this project with the ELLPACK and BCSR formatting and multiplication algorithms already completed. We also did initial evaluations on them in the month before we started this project. While the ELLPACK algorithm performs very well, the BCSR formatting algorithm is very slow. We significantly improved it after our first round of testing, but it still very slow (formatting all 14 matrices here into three block configurations each took 40 hours total on the Grace Hopper machine). To address this issue, we created a small tool that would format the BCSR matrix into a given block configuration, and then save that to a file, which the BCSR kernels could quickly load and use.

We intend for this to be an interim solution until we can look at the algorithm more deeply and hopefully optimize it. Until then, we provide the formatted BCSR matrices for anyone who wishes to use them. We believe these matrices could be useful

irrespective of this algorithm since they allow anyone who would wish to do a similar evaluation to easily create a BCSR matrix without using a formatting algorithm.

6.3.3 Building and Running the Suite

The suite is designed to implement and benchmark a single kernel. Running all the individual kernels is currently done through bash scripting. This method works fairly well, but it can be tedious and error-prone to maintain long term. It is possible to extend the benchmark suite to support multiple calculation functions per kernel, but this would negate the flexibility of our design. One possible solution would be to devise a Python script to generate a runtime script for a given configuration. We have one such script for generating data visualization plots from the CSV.

Another future goal is to improve the build system. Currently we maintain two Makefiles, one for Arm machines and one for Intel machines. The Makefiles are reasonably organized and easy to read, but they are not easily scalable to new configurations. We would like to replace this with some kind of build system that can be easily tailored for a given environment.

6.3.4 Support for SpMV

While benchmarks exist for SpMV, we can foresee cases where one would wish to do SpMV and SpMM benchmarking for a single study. In such a case, using a common set of benchmarks is preferable in order to get consistent data. For this reason, we would like to add support to our benchmark suite for SpMV. Modifying our suite for this should be trivial. At the moment, the suite automatically generates a dense matrix. Modifying it to generate a vector rather than a matrix should be relatively straightforward. Supporting SpGEMM would be interesting, but doing so would likely require significant modification (unless the operation is on one type of format). Supporting pure matrix-matrix multiplication is theoretically possible in the current implementation.

6.3.5 Memory Footprint

While we did not quantify or study this directly, when monitoring the performance of our benchmarks of both machines, we noticed that they used a huge amount of the available RAM. Additionally, in our cuSparse study, we had to eliminate several matrices because we ran out of memory, despite the GPUs we were using having a huge amount of memory.

We believe there are a few factors influencing this. When the benchmark suite loads the sparse matrix, it loads it as a COO matrix (which directly corresponds to the original Matrix Market format). This COO matrix is formatted into CSR, ELLPACK, or BCSR depending on the benchmark. However, the original COO matrix is retained for verification after the calculation, along with the formatted matrix A and dense matrices B and C. While this does not affect the GPU issues, it does use a lot of memory on the host machine, especially with large matrices.

The biggest source of the memory issue is probably due to the data types being used. In our preliminary work, we used 64-bit integers to store matrix coordinates and doubles for our values, meaning that everything is essentially 64-bit. This will naturally use a huge amount of memory, and possibly exceed what many GPUs can store. For the majority of matrices and problems, 32-bit integers and floats should be sufficient. Making this change would cut our memory use in half, possibly solving some of the issues we faced here.

REFERENCES

[1] G. Galli, G. Galli, and G. Galli, "Linear scaling methods for electronic structure calculations and quantum molecular dynamics simulations," *Current Opinion in Solid State Materials Science*, 1996.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, p. 84â90, may 2017.

[3] Z. Wang, J. Wohlwend, and T. Lei, "Structured pruning of large language models," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, 2020.

[4] T. Hoefler, T. Hoefler, M. Snir, and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," *null*, 2011.

[5] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2020.

[6] C. G. Petra, C. G. Petra, O. Schenk, O. Schenk, M. Lubin, M. Lubin, K. GÃ€ertner, and K. GÃ€ertner, "An augmented incomplete factorization approach for computing the schur complement in stochastic optimization," *SIAM Journal on Scientific Computing*, 2014.

[7] R. Eberhardt and M. Hoemmen, "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 663–672, 2016.

[8] C. Stylianou and M. Weiland, "Optimizing sparse linear algebra through automatic format selection and machine learning," 2023.

[9] M. Ashoury, M. Loni, F. Khunjush, and M. Daneshtalab, "Auto-spmv: Automated optimizing spmv kernels on gpu," 2023.

[10] Z. Xie, G. Tan, W. Liu, and N. Sun, "Ia-spgemm: an input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, (New York, NY, USA), p. 94â105, Association for Computing Machinery, 2019.

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

[12] G. Erlebacher, E. Saule, N. Flyer, and E. Bollig, "Acceleration of derivative calculations with application to radial basis function: finite-differences on the intel mic architecture," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, (New York, NY, USA), p. 263â272, Association for Computing Machinery, 2014.

[13] H. Anzt, S. Tomov, and J. J. Dongarra, "Implementing a sparse matrix vector product for the sell-c / sell-c-σ formats on nvidia gpus," 2014.

[14] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second ed., 2003.

[15] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in sparsity," in *Computational Science — ICCS 2001* (V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, eds.), (Berlin, Heidelberg), pp. 127–136, Springer Berlin Heidelberg, 2001.

[16] W. Yang, K. Li, and K. Li, "A parallel computing method using blocked format with optimal partitioning for spmv on gpu," *Journal of Computer and System Sciences*, vol. 92, pp. 152–170, 2018.

[17] U. Choi and K. Lee, "Dense or sparse : Elastic spmm implementation for optimal big-data processing," *IEEE Transactions on Big Data*, vol. 9, pp. 637–652, apr 2023.

[18] Q. Sun, Y. Liu, M. Dun, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "Sptfs: sparse tensor format selection for mttkrp via deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, IEEE Press, 2020.

[19] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proceedings of the ACM on Programming Languages*, vol. 2, p. 1â30, Oct. 2018.

[20] E. Mutlu, R. Tian, B. Ren, S. Krishnamoorthy, R. Gioiosa, J. Pienaar, and G. Kestor, "Comet: A domain-specific compilation ofÂ high-performance computational chemistry," in *Languages and Compilers for Parallel Computing* (B. Chapman and J. Moreira, eds.), (Cham), pp. 87–103, Springer International Publishing, 2022.

[21] R. Tian, L. Guo, J. Li, B. Ren, and G. Kestor, "A high performance sparse tensor algebra compiler in mlir," in *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pp. 27–38, 2021.

[22] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, (New York, NY, USA), p. 10â19, Association for Computing Machinery, 2019.

[23] E. Saule, K. Kaya, and U. V. Catalyurek, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," 2013.

[24] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O'Connor, "Learning sparse matrix row permutations for efficient spmm on gpu architectures," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 48–58, 2021.

[25] G. Dai, G. Huang, S. Yang, Z. Yu, H. Zhang, Y. Ding, Y. Xie, H. Yang, and Y. Wang, "Heuristic adaptability to input dynamics for spmm on gpus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, (New York, NY, USA), p. 595â600, Association for Computing Machinery, 2022.

[26] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," 2015.