

**ANALYSIS OF HIGH PERFORMANCE SPARSE MATRIX-VECTOR
MULTIPLICATION
FOR SMALL FINITE FIELDS**

by

Matthew A. Lambert

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Summer 2020

© 2020 Matthew A. Lambert
All Rights Reserved

**ANALYSIS OF HIGH PERFORMANCE SPARSE MATRIX-VECTOR
MULTIPLICATION
FOR SMALL FINITE FIELDS**

by

Matthew A. Lambert

Approved: _____
Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer & Information Sciences

Approved: _____
Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____
Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

B. David Saunders, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Jeremy Johnson, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Stephen F. Siegel, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Sunita Chandrasekaran, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Andrew Novocin, Ph.D.

Member of dissertation committee

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my parents for their support over the years and in particular, for tolerating my presence over these last four months of rather stressful work.

I would also, of course, thank my advisor, Dave, for getting me to finally get this thesis done, only a bit later than I really should have gotten it done.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	xiii
ABSTRACT	xv
 Chapter	
1 INTRODUCTION	1
1.1 Sparse Matrix Formats	2
1.2 Sparse Matrix-Vector Multiplication Considerations	5
1.2.1 Specialized Sparse Matrix Formats and Blocking	6
1.2.2 Per-Matrix Optimizations	11
1.2.3 Tuning, Benchmarking, and Heuristics	12
1.3 Small Field Optimizations	16
1.3.1 The Method of the Four Russians	18
1.4 Blackbox Linear Algebra	20
2 ANALYSIS OF THE METHOD OF THE FOUR RUSSIANS	22
2.1 Modeling Costs of the Method of the Four Russians	22
2.1.1 Dense Approximations	25
2.1.2 Sparse Operation Approximations	33
2.2 Sparse Methods of the Four Russians	37
2.2.1 Fixed-Size Lookup Tables	39
2.2.2 Pattern-Generated Lookup Tables	44
2.2.3 Combining Fixed-Size and Pattern-Generated Lookup Tables	45

3	SPARSE MATRIX-VECTOR MULTIPLICATION ANALYSIS	47
3.1	Introduction	48
3.2	Dense Multiplication Analysis	52
3.3	Sparse Format Analysis	64
3.4	Block Sparse Format Analysis	78
3.5	Experiment: Block Coordinate Loop Orderings	98
3.6	Dense Method of the Four Russians Analysis	106
3.7	Sparse Panel Analysis	116
3.8	Analysis Conclusions	128
4	SPARSE METHOD OF THE FOUR RUSSIANS: TRANSPOSED PANEL FORMAT	131
4.1	Transposing Panels	131
4.2	llvm-mca Analysis of Sparse Transposed Panel Format	142
4.3	Further Sparse Method of the Four Russians Considerations	146
4.4	Benchmarks	147
4.4.1	Experimental Setup	148
5	CONCLUSIONS AND FUTURE WORK	151
5.1	Summary	151
5.2	Future Work	152
	BIBLIOGRAPHY	154
	Appendix	
	SOURCE CODE AND COMPILER OUTPUT	159

LIST OF TABLES

2.1	Numerically solved solutions for k from Figure 2.10 for select values of d and m	37
2.2	A 6-by-6 matrix with 21 non-zero entries over GF_2 : $t = k = 1$	39
2.3	The same 6-by-6 matrix in Table 2.2, partitioned into panels of two columns with up to two non-zeroes per row within a panel: $t = k = 2$.	40
2.4	The same 6-by-6 matrix in Table 2.2 and Table 2.3, partitioned into panels of three columns with up to two non-zeroes per row within a panel: $t = 3, k = 2$. Some rows within a panel have multiple segments.	40
2.5	The same 6-by-6 matrix in Table 2.2, 2.3, and 2.4, partitioned into panels of three columns per row within a panel: $t = k = 3$	44
3.1	llvm-mca analysis for classical dense multiplication in the i-j-k ordering shown in Listing 3. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.	55
3.2	Cost and reciprocal throughputs (cycles until another operation can be launched that requires the same execution units) for common operations in this dissertation on the Skylake architecture, based on values from Agner Fog’s instruction tables [18].	60

3.3	llvm-mca analysis for classical dense multiplication in the i-k-j ordering shown in Listing 8. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.	63
3.4	llvm-mca analysis for sparse matrix-vector multiplication using coordinate format with three vectors shown in Listing 9. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.	66
3.5	llvm-mca analysis for sparse matrix-vector multiplication using coordinate format with a vector of tuples shown in Listing 13 compared to a coordinate format with three vectors shown in Listing 9.	71
3.6	llvm-mca analysis for sparse matrix-vector multiplication using the compressed sparse row format shown in Listing 14. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.	72
3.7	llvm-mca analysis for sparse matrix-vector multiplication using the compressed sparse column format shown in Listing 16. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.	75

4.1	llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the sparse transposed panel format presented in Listing 36 with blocks of 1, 8, 32, and 64 rows. The cost, in cycles, noted in the “other” column is a cost present for each unique pattern.	143
4.2	llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the modified coordinate format presented in Listing 37 with blocks of 1, 8, 32, and 64 rows. The cost, in cycles, noted in the “other” column is a cost present for each complete block.	143
4.3	llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the alternate sparse transposed panel format presented in Listing 38 with blocks of 1, 8, 32, and 64 rows. The cost, in cycles, noted in the “other” column is a cost present for each unique pattern. Missing values in the first three columns denote that no clear main loop was found.	144
4.4	llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the modified coordinate format presented in Listing 39 with blocks of 1, 8, 32, and 64 rows. The cost, in cycles, noted in the “other” column is a cost present for each complete block.	145
4.5	Running times, in nanoseconds-per-entry, for the specified sparse formats and given matrix dimensions and densities. coo is coordinate format, csr is compressed sparse row, csc is compressed sparse column, csp is a sparse panel format, and tcsp is our coordinate-like transposed format. csp and tcsp specify the parameter of the Method of the Four Russians used. All vectors used are vectors of 64-bit integers.	149

LIST OF FIGURES

1.1	Sketch of the Method of the Four Russians: the matrix A is divided into panels of k (here, 2) columns and the right-hand operand into panels of k rows. For each panel, all 2^k combinations of rows of the right-hand operand are computed. Then a k -bit entry from A is used to index the table. Thus, each 11 in the right panel of the matrix requires just one operation to add to our result, since the $c + d$ combination is already computed in our lookup table.	19
2.1	The cost, in number of additions, to apply the densely-represented m -by- n matrix A to the n -by- p operand B using the Method of the Four Russians.	26
2.2	Finding a solution for k from the partial derivative of the cost found in Figure 2.1.	27
2.3	The cost, in approximate reads and writes, to build lookup tables in the Method of the Four Russians.	28
2.4	Approximate count of reads and writes using the lookup table and destination matrix in the Method of the Four Russians. This, with Figure 2.3 represent all costs except reading entries from matrix A	29
2.5	Number of reads required to read matrix A in a dense representation or a coordinate format sparse representation, assuming k divides n and k divides the number of bits per machine word.	29
2.6	Number of reads and writes in the dense Method of the Four Russians.	30
2.7	Finding a k that minimizes the number of reads for a dense representation.	31
2.8	Finding a k that minimizes the number of reads+writes for a dense representation.	32

2.9	Finding a k that minimizes the number of reads for a sparse representation storing a dense matrix.	32
2.10	Summary of expected costs in the sparse Method of the Four Russians, for a random matrix with uniformly distributed entries.	35
2.11	Finding a solution for k with our relaxed model of costs in the Sparse Method of the Four Russians.	36
4.1	Relevant values for applying a single panel in a sparse panel format.	132
4.2	Transposed variant of the panel in Figure 4.1.	132
4.3	Block, masked transposed variant of the panel in Figure 4.1.	135

ABSTRACT

This thesis explores the intricacies of obtaining high performance sparse matrix-vector multiplication on modern hardware, with an emphasis on operating over data from small finite fields. We develop and present novel adaptations of classical, fast matrix multiplication algorithms over these fields and apply them in a sparse setting. Further, we compare these novel formats and algorithms to a wide array of standard sparse matrix formats. In particular, we use modern code analysis tools to show how data layouts, vector/panel widths, and choice of compiler all substantially influence the efficiency of the underlying arithmetic of these various sparse formats. These analyses are performed in a data-agnostic manner, meaning we focus solely on the efficiency of the arithmetic and not on higher-level aspects of performance such as cache access patterns. In particular, we analyze the assembly code produced by compilers, removing matrix-specific intangibles from the discussion of format. These are still important considerations when considering any specific matrix, but they make comparing general formats to one another difficult, without exhaustive benchmarking. These results show the theoretical peak arithmetic performance, which we discuss in this abstract, analytic perspective. We see similar trends in synthetic performance benchmarks. Ultimately, we show that the Method of the Four Russians can be directly adapted to sparse matrix-panel (a matrix with relatively few columns) multiplication and that a custom, high-performance variant can achieve high performance in sparse matrix-vector multiplication, with potential to perform better in real-world matrices.

Chapter 1

INTRODUCTION

This thesis focuses on the theoretical and high-performance aspects of sparse matrix-vector multiplication. Sparse matrix-vector multiplication is a fundamental kernel of high performance computation. Indeed it is one of the “seven dwarfs” identified as the most important kernels in the highly influential paper [6]. It is also the essence of the PageRank algorithm [33], the original core of Google search. Also, it is the basis of blackbox exact linear algebra used in computer algebra applications as discussed in Section 1.4.

In this chapter, we will present an overview of related and motivating work in the field of exact linear algebra and sparse matrix-vector multiplication, particularly focusing on general-purpose and matrix-specific optimizations.

In the second chapter, we review the costs of the Method of the Four Russians, a fast matrix multiplication algorithm specialized for dense arithmetic over GF2, discuss and model the costs for a sparse version of this algorithm, and propose some practical optimizations for this new sparse variant.

In the third chapter, we launch into an analysis of the theoretical real performance of sparse matrix-vector multiplication. Specifically, given C++ implementations of common and novel sparse matrix-vector routines, we use a tool to measure the theoretical performance of the assembly generated by the Clang and GCC compilers and determine what advantageous and disadvantageous decisions the processors make, given the different formats and different vector widths.

In the fourth chapter, we propose another new format, this one specifically designed to address the shortcomings of the sparse Method of the Four Russians developed in the second and third chapters. We apply our same analysis procedure and

perform some benchmarks on randomly-generated sparse matrices, generally yielding better performance for sparse matrix-vector multiplication than the straightforward sparse Method of the Four Russians format and better performance than standard sparse formats for larger cases.

Finally, we conclude with a review of the work done and results we have seen in this thesis and discuss future extensions of this work.

1.1 Sparse Matrix Formats

In this section, we will present an overview of the “standard” sparse matrix formats used in many applications. These formats are those that are not ill-suited to sparse matrix-vector multiplication. Other formats such as Skyline involve re-ordering matrices in order to efficiently perform steps of Gaussian Elimination. SPARSKIT [36], a sparse matrix computation package supports these formats and some formats are also supported by LinBox, an exact linear algebra library. In nearly all cases, the storage requirements are proportional to the number of nonzero entries in the matrix or the number of nonzero entries and one dimension of the matrix. Typically, these are ordered in some fashion, with row-major or column-major order being prominent. However, additional metadata are required and access within the matrix is often indirect. Many of these formats are designed to take advantage of structure such as prominent diagonals or small blocks of non-zero entries in order to minimize this additional overhead. In all cases, these sparse matrices can be efficiently iterated over, although random access of entries is often expensive. Although random access is not common for sparse matrix-vector computations, it is a factor when finding information about the matrix such as distribution of non-zero entries.

Selection of the most appropriate format is an additional area of frequent research. Further, *a priori* knowledge of matrix structures and system characteristics may lead to further optimizations.

Coordinate format (COO) consists of three arrays: an array of row indices, an array of column indices, and an array of non-zero values. Alternatively, data can be

stored in one array of tuples of three values. This format is attractive because of its simplicity and is often used when constructing matrices. However, efficient access still requires an ordering placed on all vectors and 3NNZ (number of non-zero entries) space is required, which is less efficient than other formats.

Compressed Sparse Row (CSR) is the most widely-used sparse matrix format. The matrix is stored in row-major order (or column-major order in compressed sparse column format), with one array, A , storing non-zero entries and another array, JA , storing column indices. A third array, IA , stores the index of the first non-zero within each row. Thus, the entries within A and JA belonging to the i -th row are bounded by the i -th and $i+1$ -th entries of IA . The total storage requirement is $2\text{NNZ} + N_r + 1$, where NNZ is the number of non-zero entries and N_r is the number of rows in the matrix.

Diagonal storage (DIAG) is a specialization used for matrices with only a few, prominently used diagonals. Entries are stored in a 2D array the length of which is equal to the longest diagonal and the height is one row per each non-zero diagonal. An additional array keeps track of the offset of each row of the entry array. For example, if the first, third, and fifth diagonals are stored in the entry array, then the offset array would contain 1, 3, and 5.

Ellpack format, or Ellpack-Itpack format, (ELL) is a similar format to DIAG, adapted for matrices with few non-zeros per row. There are two 2-dimensional storage arrays of the same size: one containing non-zero entries and one containing column indices. Each row of the entry array has all non-zero entries within the row stored contiguously. If a row has fewer than that maximum number of non-zero entries, it is padded with zeros. This format gives predictable working sizes and can be fairly divided more easily than CSR, though storage will be less efficient if there are different number of non-zero entries per row.

Jagged diagonal storage (JDS) is a space-efficient format similar to DIAG and ELL. Rows are written with all zero entries removed and permuted by length, with the permutation stored. All first entries in each row are stored with corresponding

column indices, then all second entries, if present, and so forth. An additional array stores pointers to the first entry of each column if written in un-padded ELL format. This format is slightly more complicated than other formats, but achieves good space efficiency and accesses entries in orders that are often efficient for some numerical iterative methods.

Hybrid format (HYB) is traditionally a combination of ELL and COO formats. A constant, b , is chosen and the first up to b non-zero entries within each row are stored within ELL format. All remaining non-zeros are stored in a COO format. This format gives the predictable sizes of ELL along with better memory efficiency if there are unequal numbers of non-zero entries among rows. The general concept of a hybrid matrix can be applied with any number of formats, as we can express any matrix as the sum of matrices.

Block sparse row format (BSR) is a generalization of CSR. In this format, the elements of a CSR matrix are instead fixed-sized dense matrices of relatively small size. This can be useful for hardware or algorithms that can comfortably manipulate multiple elements at once and requires correspondingly fewer entries in the IA and JA arrays, although it can possibly introduce the storage of zeros, if the distribution of non-zero entries does not perfectly fit the blocks.

Variable block row (VBR) is yet another generalization of CSR. However, blocks can be of variable sizes. We can think of this format as drawing any number of vertical or horizontal lines through the entire matrix and considering the blocking and non-zero blocks that result. In this case, additional metadata arrays are created and store the locations of the blocks. An additional array contains pointers to the first element in each block. Blocks are stored consecutively in CSR, with IA and JA containing the typical indices from the CSR format. Finding blocks is a separate issue and heuristics are often used.

These formats are generally data-agnostic. Variations of these and heuristics for parameter selections are discussed in the next section.

1.2 Sparse Matrix-Vector Multiplication Considerations

Operating with sparse matrices is fundamentally different from operating with dense matrices. With dense matrices, we have predictable sizes and access patterns, but we have fewer guarantees when working with sparse matrices. This problem is exacerbated when considering sparse matrix-vector multiplication, as we have no opportunities to re-use entries in the matrix. A typical sparse matrix-vector multiplication algorithm iterates over non-zero entries in a row of the matrix and performs a dot product of these entries with the corresponding entries in the vector. It is widely recognized that standard sparse matrix-vector multiplication is typically bandwidth bound [22, 32], as the cost of loading these matrix elements and storing the result often dominates the cost of one floating point or integer operation, especially if these loads follow irregular patterns. When extending our problem to that of a sparse matrix times a panel (SpMP multiplication), there is opportunity to re-use entries in the sparse matrix, but there is still substantial overhead compared to dense matrix multiplication.

Goumas et al. [22] identify four major bottlenecks of sparse matrix-vector multiplication, applicable to both vectors and panels as products. First, there is substantial memory intensity compared to the number of arithmetic operations performed. For a matrix stored in COO format, a single entry is represented by three values, so working with these three values and one value from the vector is significant for multiplication and addition of only one value. Thus, we see a large memory access-to-operation ratio. Second, there is often poor cache performance of the sparse matrix due to indirect accesses and additional metadata, in addition to the low amount of computation. Third, there is irregular access to the vector operand, yielding more poor cache performance. Since consecutive entries in the sparse matrix may be very far apart in terms of coordinates, reads from the vector may be unpredictable and far apart. Finally, short and irregular row lengths in the sparse matrix can lead to slight, but measurable performance losses as there is increased overhead and loop prediction conditions are not often predicted correctly.

Optimizations for sparse matrix-vector multiplication aim at addressing different

aspects of these shortcomings. Some formats aim at decreasing the amount of data needed to represent the sparse matrix, while others reorder or block the matrix into predictably-sized blocks to allow for better cache performance and aligned reads and vectorization. Here, we present some standard sparse matrix formats. Later, we will discuss existing work in optimization of sparse matrix-vector multiplication.

1.2.1 Specialized Sparse Matrix Formats and Blocking

In this section, we will discuss select sparse matrix formats beyond the common formats presented in the background section. These specialized formats aim at decreasing the amount of data needed to represent the sparse matrix, representing data in a compute-friendly format, or exploiting particular structures. Most sparse matrices are not uniformly sparse, but instead exhibit dense clusters of non-zero elements. Identifying these dense areas allows us to use arithmetic for dense matrices, making much better use of caching and pipelining on modern hardware. In some cases, it may be even advantageous to permute rows or columns of the matrix to group denser regions together. Another consideration is when to explicitly represent zeroes. Much of the benefit of sparse formats stems from not representing zero entries. There is often some cutoff at which it is beneficial to represent a small dense block with very few non-zero entries densely, rather than sparsely.

Tang et al. [37] introduce a compression scheme for hybrid, ELLPACK, and COO formats aimed at minimizing storage of additional metadata such as column indices. This bit-optimized representation is aimed at creating efficient streams that can be efficiently uncompressed on GPUs. This bit-optimized format uses techniques similar to bitpacking from stream processing to divide the matrix into slices or panels. By having panels of known sizes, we can encode more information in less space at the cost of direct usability. For example, the gaps between column indices are stored instead of the column indices, themselves. If there are non-zeros at indices 1, 2, 3, 4, and 5, the gaps of 1, 1, 1, 1, 1 are recorded instead, requiring just 5 bits, instead of 20 bits (5 entries of 4 bits each). If the gaps were 1, 3, 7, then this would require 9 bits (3 bits

per gap). Provided the largest gap size requires fewer bits to represent than the largest index, this format requires fewer bits than storing indices directly. The authors discuss more complex strategies and reordering procedures for the three formats. Over their benchmarked matrices from the University of Florida Sparse Matrix collection, they obtained speedups of 20-110% over the standard ELLPACK format. In each case, the bit-optimized representation decompression still leaves the operations memory-bound.

Yzelman and Bisseling [45] propose a zigzag CSR format. Rather than having every row stored left-to-right as in CSR, rows alternate direction for potential cache performance improvements, as it is more likely that the same vector elements will be reused. This approach applied recursively on blocks of the matrix can potentially yield exploitable block structure, if not already present in the matrix. However, it can also remove structures already present. Combined with blocking strategies, however, reorganizing the order of rows of non-zero entries improved performance in a majority of test cases.

The level of blocking performed is another consideration. Nishtala et al. [32] suggest that blocking so that the entire working set of entries of the matrix and corresponding entries within the vector fit into cache is beneficial if the matrix is dense enough that the vector entries are reused. The authors conclude that cache misses represent the single largest source of lost performance in sparse matrix-vector multiplication on modern hardware. This level of blocking is compatible with many formats and other blocking strategies, but still has some overhead and possibilities for unequal block sizes, as it may be difficult to block some sparse matrix formats equally.

On the other hand, work by Wozniak et al. [43] suggests that for simple dot product-based implementations of sparse matrix-vector multiplication on modern processors that implement prefetching and automatic cache management, explicit blocking for cache size may not be necessary if access patterns are regular. However, depending in the target hardware, especially in the case of accelerators, explicitly setting block sizes to fit SIMD or working group sizes may be crucial [27, 29, 20].

Whether explicitly blocked or not, the order of element access is still important

in reducing the total number of cache loads. Morton ordering (or Z-ordering, exploring the two dimensional space in a fractal Z order) is used in dense matrix multiplication for more efficient reuse of items in cache [31].

The variable block row (VBR) sparse matrix format [36] previously presented, for example, is a block-based generalization of many CSR-based formats. There are indices in both dimensions and extra metadata to capture block sizes. Effectively, any number of lines are drawn horizontally or vertically through the entire matrix. These individual blocks created this way are then stored in CSR. The advantage of this format is that small blocks in this form can be fed directly into custom dense multiplication code. Vuduc and Moon [40] show that many matrices from finite element method models can have most non-zeros captured in just a couple small block sizes. The conversion to VBR, however, requires candidate block sizes or performs a simplistic search that requires exact matches of consecutive rows or columns. Vuduc and Moon instead introduce an unaligned block CSR (UBCSR) format. Different block sizes are stored in separate matrices, with code optimized for each block size. This version is unaligned, as patterns of the same size are not necessarily multiples of their dimension apart, as they would tend to be with the default format conversion. The end result of this work is lower memory requirements, as no zeros are stored and custom code for each block size can be optimized. This method showed great speedups over CSR and even over block CSR for matrices from finite element methods.

Using fixed-size blocks of small sizes is well-studied [35] [38] and often exhibits comparable, if not better, performance compared to more adaptive blocking algorithms. Simple 1-by-2 (or 2-by-1) blocks of non-zero entries in a CSR-based format, even without the data compression used in small finite fields can offer over 20% improvements over standard CSR [35]. These 1-by-2 blocks are stored in one matrix, while all remaining single entries are stored within their own matrix. Further work by Pinar et al. [35] suggests that reordering columns (or rows) to create larger dense blocks can be advantageous. Additionally, the authors recognize that even for a single fixed block size, the search process when one dimension is larger than 1 is computationally prohibitive

in most cases, especially when considering potential reorderings. They suggest a reduction to the Traveling Salesman Problem and then the use of a fast heuristic-based solver to find a reordering and block size. The TSP-based reordering and blocking shows improvement over standard reorderings. In certain cases, the TSP-based reordering and blocking yields dramatic performance increases compared to a CSR, but the average case is comparable to using 1-by-2 blocks which shows an average of 26% improvement over CSR over a set of standard benchmark matrices.

Zhang et al. [47] apply a standard quadtree blocking strategy to sparse matrices, recursively dividing the matrix into quadrants until one quadrant, when represented in a COO sparse format is small enough to fit into cache. This format and process they refer to as cache-oblivious extension quadtree storage. COO is chosen over CSR because it offers better insensitivity to data distribution and a fixed, computable size. CSR is dependent on number of non-zero entries and dimension, while COO's size is dependent only on number of non-zero entries. This method achieved a modest 10-50% performance increases over CSR for large datasets for sparse matrix-vector multiplication. When extended to a distributed system, the parallel quadtree approach averaged a 63% speedup over the standard parallel CSR approach. Yzelman and Roose [46] discuss cache-oblivious strategies for shared-memory parallel implementations of sparse matrix-vector multiplication. This work provides general ideas for distributing blocks among multiple processes. These blocks can be represent in any format, as the authors suggest an ordering of element access to best re-use elements in both the sparse matrix and vector. Large unstructured matrices benefited relatively more from this approach than small matrices or large structured matrices, achieving an average of approximately a 5.5 times speedup over 12 threads for most underlying formats. The search process for finding this ordering is not included in these timings.

CSR5 [29] is a blocking format aimed at exploiting the storage efficiency of CSR and the automatic load balancing of segmented sum sparse matrix-vector multiplication. Elements are partitioned into column-major order within SIMD-sized blocks.

These blocks are in row-major order. As with CSR, column indices are stored. Additional bit-sized flags and metadata are used to perform segmented sums within an operation. While generally comparable to the performance of CSR-based multiplication for regular matrices, the load-balancing aspects of the segmented sum operations yields better relative performance for irregular sparse matrices over multiple types of hardware.

Liu and Vinter [30] extend the idea of a segmented-sum based multiplication well-suited for GPUs. This variation uses speculative execution of sums to remove unnecessary work from empty rows by keeping track of potential empty rows when executing code on the GPU. Any unnecessary partial sums are handled back on the CPU. Rather than expensive preprocessing to address locations of empty rows as done in CSR5, these locations are handled at runtime and are more efficient the matrix is used for only a few vector multiplications.

OSKI (Optimized Sparse Kernel Interface) [39], a library built around tuning sparse matrices for fast computation allows the user to input candidate block sizes and likelihoods for VBR and UBCSR. In practice, this is likely an ideal middle ground between the user generating his or her own custom code and performing an exhaustive search, as the user may have some insights into existing block structure of the code. OSKI further offers implementations of many of these optimizations including blocking and row or column reordering. Further, it is capable of determining if certain optimizations would be beneficial based both on models and adaptive methods.

Finally, standard clustering algorithms like DBSCAN [17] can be used to find dense blocks without the stipulations required by VBR or UBCSR. A review of literature in the field shows very few substantial instances of standard clustering algorithms used in sparse matrix-vector multiplication. This is likely because multiple large dense blocks are unlikely to appear in sparse matrices.

Ultimately, blocking techniques can serve several purposes. First, they provide ways to partition the matrix into blocks of entries for efficient cache usage. Second, they can identify contiguous blocks of non-zeros, allowing for better vectorization and lower

overhead. However, applicable strategies are dependent on storage format, hardware, and matrix structure, so not every blocking technique can be applied to all matrices and all systems. Further, the overhead required for many of these optimizations requires that the matrix be used for sparse matrix-vector multiplication many times to make it worthwhile.

1.2.2 Per-Matrix Optimizations

In this subsection, we will review optimizations that can be applied on a matrix-by-matrix basis. While the effectiveness of blocking and partitioning strategies varies based on matrix characteristics, these optimizations require analysis or extensive modification of the original matrix. Matrices used in exact linear problems are typically rank-deficient [14] and are likely to exhibit exploitable structure, so identifying and capturing these characteristics is one direction of research. In many numerical computations, the value 1 (and -1, to a lesser degree) often occurs far more frequently than other non-zero values. In examples typically used in finite fields, these values occur more often than other values, but not to the same extent seen numerically [9]. A common optimization is to separate all occurrences of 1 into their own sparse matrix. Having a fixed value means that the value vector used in most formats does not need to be stored. The space savings yield good GPU speedups, with Boyer et al. [9] showing up to a 20% speedup for word-sized primes in a GPU-based sparse matrix-vector multiplication routine. This is a particularly-motivating example for specializing work in small fields, as if 1 occurs very frequently, speeding the portions of the matrix that just contain 1 will still result in savings across the whole matrix.

The second-most common optimization for sparse matrix-vector multiplication concerns code generation. Rather than read an entire sparse matrix into memory and apply traditional sparse matrix-vector multiplication routines, the matrix is analyzed and custom multiplication kernels are generated. The GiMMiK project [43] uses code generation to generate custom kernels for fluid modeling applications. These sparse matrices are highly-structured and contain only a few non-zero entries per row with

frequently-repeated patterns. These matrices are fixed and repeatedly multiplied by a wide matrix of panels. By reordering rows and generating custom kernels, the authors achieve up to a 100% speedup for their workflows compared to their existing CSR-based sparse matrix-vector multiplication routine. Boyer et al. [9] also consider code generation in the case of finite fields. Their approach is to partition the matrix into blocks of 1000 non-zero entries and generate kernels from these entries. These custom kernels are compiled separately and dynamically called as needed at runtime. Unfortunately, the authors do not present any performance details for these optimizations independent of other work, so the exact performance benefit cannot be gleaned.

1.2.3 Tuning, Benchmarking, and Heuristics

Given the vast number of possible optimizations, high memory-to-operation cost of sparse matrix-vector multiplication, and high dependency on data layout, there is significant research on benchmarking and application of heuristics in optimizations. For applications where the matrix is used only once, there is little opportunity for optimization, as reading the matrix represents a large portion of the real cost of sparse matrix-vector multiplication. However, many sparse applications such as iterative methods in numerical modeling and Wiedemann’s method in exact linear algebra use a fixed sparse matrix for repeated sparse matrix-vector products. However, there is still a tradeoff between time spent optimizing and the benefits gained.

A large amount of work regarding sparse matrix heuristics involves determining a format to use, often among the standard formats. This is especially critical when using accelerators, as the bottleneck is typically in communicating to and from the accelerator. Thus, there may be benefits in compressing the matrix before transmitting, as we saw have discussed in work by [37].

Guo et al. [23] and Armstrong and Rendell [5] present two different approaches to this selection process. Guo et al. address the problem of using a GPU for acceleration and selecting the fastest format. The authors introduce an idea of a strip of

a matrix, which is the largest matrix that can be represented and operated on efficiently given the compute units on the accelerator. Benchmark matrices are generated and timings computed. Based on the strips of the matrix, a time is calculated based on the benchmark times. Predicted times for each format and actual compute times when applied to select matrices differ by less than 9% in nearly all cases. The model primarily predicts that ELLPACK or HYB storage should be used and that for the selected parameters, about a 40% speedup over COO or CSR can be achieved. The costs of prediction, once the benchmarks are run for the target hardware, depend only on matrix dimension and number of non-zero rows and number of non-zero entries, so prediction is quite efficient once the benchmarks are run.

Further existing work over finite fields is done by Giorgi and Vialla [21], although this work involves representations of word-size primes and multiprecision primes. This work uses heuristics to determine data representation, accommodating the extra space needed for values relative to indices.

Armstrong and Rendell [5] consider a runtime process, primarily targeted at CPUs. Their machine learning system uses 65 formats, primarily consisting of the standard formats with various fixed block sizes for BCSR, so the cost of conversion is minimal. Their system integrates with existing multiplication routines, taking a matrix and a number of multiplications to perform. Attributes of row count, column count, density, average row density deviation, and mean number of neighbors for each non-zero are used with fixed weights selected by the authors from previous work. The model is initially built by performing enough multiplications in a starting format to achieve verifiable timings. Then for future iterations, with some probability, the matrix format is changed, k multiplications in the current format are performed, or the model is consulted for the best course of action considering available information. A weighed reward parameter controls selection of k and is updated as information is gained. With this process, the state space is explored with parameterized probability, but good formats are likely to be used for most multiplications. Further, since the model considers only the cost of conversion and the five attributes, it can be constantly updated or

even adapt in real-time to changing hardware. When considering the cost of transformations and calculation of attributes, the runtime method is approximately 13% faster on average for 1000 sparse matrix-vector multiplications than a static method, which selects only the single best format used on the benchmark matrices. Further, the method is only 8% slower on average for these multiplications when compared to individually selecting the single best format among the 65. The total overhead of the attribute computation and learning algorithm is approximately 0.6% the cost of 1000 sparse matrix-vector products.

In further work, Zein et al. [24] gave an initial successful look at selecting among a few sparse formats based on dimension and average number of non-zeros per row. [28] uses a larger set of features (measurable characteristics of the matrix) to classify a new matrix into a common sparse format based on training and analyzing two thousand training matrices, but at significant overhead when calculating these additional attributes. The authors recognize that this is likely too slow for runtime usability, but it can be run online if the classifier is already trained. Thus, if enough time is spent on training, there is some potential in querying this for a new matrix, but many matrices are required to amortize this cost.

[16] takes a slightly different approach. While recognizing that selecting the proper format is critical, it is possible to change formats at run-time, though this is not beneficial in the majority of cases. These examples are illustrative of the approaches taken in deciding on matrix formats. They all test their performance against the University of Florida Sparse Matrix Collection and Guo et al. construct synthetic matrices for their predictions. With the wide variety of sparse matrices, selection of both benchmark and test matrices is not trivial, as it is in dense cases where only dimension is considered.

Gahvari et al. [19] aim to address a deficiency in benchmarking for sparse matrix-vector multiplication. As sparse matrix-vector multiplication depends heavily on underlying matrix layouts and representations, selection of matrices and tuning of

these matrices for benchmarking purposes as is done with BLAS for LINPACK benchmarks should be performed systematically. The authors select a collection of sparse matrices from existing sources that they believe is a suitable representation of the variety of common workloads sparse matrix-vector multiplication is used for. These matrices cover a variety of dimension, density, common block size, and bandedness. From statistics on these matrices, a matrix generator was constructed. Matrices constructed from this generator can be scaled, so benchmarking can be applied to multiple machines as caches and memory grow. Generated matrices had similar performance to matrices from the data set and were tuned with OSKI [39]. The authors noted that beyond certain dimensions, based on the machine, performance leveled out, so very large matrices did not need to be tested, as the tuning targets cache performance. The conclusion is that five minutes of benchmarking these generated matrices can explore the input space about as well as 150 minutes of benchmarking with real matrices from the initial data set. The importance of accurate benchmarking within a domain is shown by [32], [9], and [23] who extensively use models to predict performance and select parameters.

Tuning sparse matrix-vector multiplication routines is yet another application of benchmarking. OSKI [39] can use extensive tuning, covering a wide variety of parameters to partition matrices into efficient blocks. Much of this work we have already referred to. Other work done by Heinecke [3] performs just-in-time generation of Intel-specific micro-op code for specific matrices for the base case of numerical matrix multiplication, much like the compilation-time work by Boyer [9]. This approach is intended to be done at run-time for very small systems. Tuned kernels are created based on matrix-specific information like stride and dimension to use optimized stores, loads, and vectorized instructions, even in unaligned cases. While quite specific to Intel processors, performance of matrix multiplication was at worst comparable to Intel's MKL and better in most cases. Further, it showed great scaling potential across many cores on a shared-memory system.

Ultimately, predictions require benchmarks on representative data and the number of matrices and similarity to desired matrices greatly influences models and parameter selection. Given the potential benefits of selecting block sizes and different formats, performing some analysis or testing on matrices may be beneficial if the matrix is used for many computations. Further, there are both run-time and compilation-time approaches that can be selected, depending on the application.

1.3 Small Field Optimizations

Small finite fields, for our purposes, are finite fields in which multiple elements can be stored within a single machine word. For practical consideration, we will use the range of primes between 2 and 13, inclusive, with special emphasis on GF2 and GF3. With very small entries, requiring only one bit per element in the case of GF2, we can treat machine words such as 64-bit integers as vectors containing multiple entries over these fields and effectively perform operations on small vectors of these elements efficiently. With these compact representations, we can achieve significantly more throughput and accommodate much larger problems than we can when using one machine word per element. Furthermore, the small quantity of unique field elements allows us to achieve additional speedups through optimized vector axpys (scalar multiplication and addition: $y+ = ax$) and multiplication.

The simplest way of representing these small fields is assign each element over GF_P a certain number of bits, k , (such that $k > \log_2 P$) and store as many of these as possible within a machine word, sometimes referred to as bitpacking [15, 26, 44]. We choose k specifically to be larger than the minimum size in order to prevent overflow from occurring. For brevity, our examples here will consider 8-bit integers. The concepts can be applied to any size integer word (or the mantissa of a floating point word) or even vector registers on processors that support vectorized arithmetic, although exact selections of k will depend on the field and the hardware.

As an example, if we are working over GF_3 and select $k = 4$, we can fit two field elements within a single 8-bit integer: the lower four bits dedicated to the first entry

and the upper four bits dedicated to the second entry. Thus, adding two of these vectors together is done with standard integer addition. Scaling a vector is done by integer multiplication. Since we have 4 bits for each entry in this example, we can accumulate several in-place operations before worrying about overflow, as our maximum value for each entry is 15 and we are over GF_3 . We must normalize our value before overflow occurs, which is much more expensive than arithmetic operations. Thus, there is a trade-off between normalization frequency and storage efficiency.

At an extreme, if we use three bits for each entry over GF_3 , we can store 21 elements in a 64-bit word, or 84 in a 256-bit vector unit. We can perform addition of two vectors 84-entry vectors with just one hardware operation, although we have to normalize our result after every other operation, before overflow occurs. Again, finding the optimal tradeoff is dependent on multiple factors and is ever-changing as hardware advances.

An alternative to bitpacking is bitslicing [15, 8, 44, 26]. Rather than store the bits of a single element contiguously, multiple machine words are used, with the first word holding the first bits of each entry, the second word holding the second bits of each entry, and so forth. Additionally, specialized arithmetic is employed to ensure no overflow occurs. Boothby and Bradshaw [8] describe a scheme for GF_3 where 6 logical operations can add or subtract two machine word-sized vectors of elements over GF_3 using just two bits per entry. With two 64-bit integers (one containing the first bits of 64 elements and one containing the second bits of 64 entries), we can store 64 elements and add or subtract another vector of 64 elements in just 6 operations with no normalization necessary. This is in contrast to the bitpacked representation where at most 42 elements can be stored in two 64-bit integers, while only 16 are usually stored [44]. In practice, the times spent loading and storing the values dominate the computation time [26, 44, 8]. Boothby and Bradshaw also detail bitsliced schemes for some larger prime fields, although the complexity of the arithmetic increases significantly beyond GF_5 . Further, Boothby and Bradshaw reduce fields of powers of primes to arithmetic using coefficient matrices from the base fields, so fast arithmetic over prime fields translates to fast

arithmetic over fields that are powers of primes.

1.3.1 The Method of the Four Russians

Matrix multiplication over these small fields is often expressed as a series of axpy operations, since operating on rows of the right-hand operand is much more efficient than manipulating individual elements. To that end, the (i, j) entry of A in the product $C = AB$ leads to the axpy operation $C_{i,+} = A_{i,j} \times B_{j,:}$, allowing for use of the efficient axpy operations over our small fields. The Method of the Four Russians [4, 2] is a sub-cubic boolean matrix multiplication algorithm first developed by Arlazarov et al. At its core, the algorithm efficiently computes a lookup table of boolean linear combinations of t rows of the right-hand operand at a time and uses t entries at a time from the left-hand operand as indices into this table. We divide our left-hand operand into vertical panels of width t columns and our right-hand operand into horizontal panels of height t rows. We compute a lookup table containing all 2^t linear combinations of these t rows. These combinations can be computed with a number of row operations proportional to 2^t using a standard ordering or Gray Code ordering [1]. We then use all t -length entries in our panel of t columns as an index into our lookup table, which we add into our product. This process is illustrated in Figure 1.1. For example, if $t = 5$ and we read 10111 from $A_{i,j:j+4}$, this can be thought of as adding $C_{i,+} = 1 \times B_j + 0 \times B_{j+1} + 1 \times B_{j+2} + 1 \times B_{j+3} + 1 \times B_{j+4}$, which would involve four or five row operations, depending on how the 0 is handled. Since we already computed all linear combinations, these four or five adds can be performed in just one.

For multiplication of two $n \times n$ matrices, we divided A into $n \frac{n}{t}$ entries of t elements. Each of these entries is an index into a lookup table for a row operation, so we perform $n \frac{n}{t}$ row operations using these entries. Further, we build $\frac{n}{t}$ lookup tables, each requiring 2^t row operations to build. Thus, the total number of row operations performed in the Method of the Four Russians is $\frac{n}{t}(2^t + n)$. If we let $t = \log_2 n$, our total number of row operations performed is $\frac{2n^2}{\log_2 n}$, yielding a logarithmic speedup over traditional cubic matrix multiplication. This speedup comes at the expense of storing

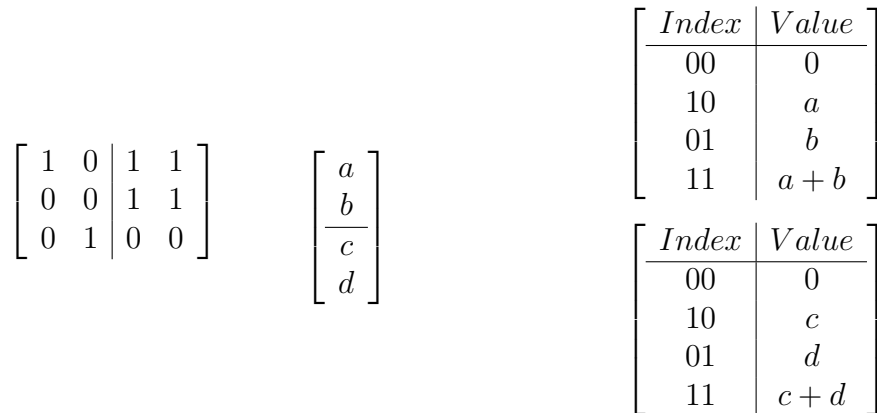


Figure 1.1: Sketch of the Method of the Four Russians: the matrix A is divided into panels of k (here, 2) columns and the right-hand operand into panels of k rows. For each panel, all 2^k combinations of rows of the right-hand operand are computed. Then a k -bit entry from A is used to index the table. Thus, each 11 in the right panel of the matrix requires just one operation to add to our result, since the $c + d$ combination is already computed in our lookup table.

the lookup table, whose size is roughly the same size as the matrices we are operating on. In practice, exact selection of t depends heavily on available cache space [1].

For matrices over GF_2 , the Method of the Four Russians offers a clear advantage over traditional multiplication methods for matrices small enough that $\log_2 n$ rows can be easily stored. For slightly larger fields, the method is still applicable, though there is falloff in benefit. Boothby and Bradshaw [8] illustrate the effectiveness over GF_3 using their bitsliced representation. Larger fields pose difficulties. For example, over GF_3 , should all 3^t combinations of rows be calculated, or just the same 2^t combinations and then use the table once for each bit of the field elements (e.g., perform one index on the 1s and one on the 2s)? Parker [34] uses a specialized GF_3 representation requiring one additional operation in order to keep table sizes between 2^t and 3^t , while maintaining just one index. For larger fields, Lambert and Saunders [26] show benefits of using 2^t -sized tables for bitpacked primes up to GF_{13} compared to traditional methods, while performing one table lookup and add for each bit in the prime.

In summary, the Method of the Four Russians offers a speedup because of the

computation of all binary combinations of t rows of a matrix. Thus, all repeated entries are not unnecessarily recomputed and a set of t entries with multiple non-zero elements will require just one addition instead of up to t . The realized benefits of this method increases as the height of the left-hand side matrix increases, as patterns will be repeated more frequently. An overarching goal of this thesis is to adapt this algorithm from dense matrix-matrix multiplication to sparse matrix-vector multiplication.

1.4 Blackbox Linear Algebra

A fundamental algorithm in black box linear algebra is Wiedemann’s Algorithm [42]. Wiedemann’s algorithm probabilistically finds a solution to a matrix, the minimal polynomial, rank, or determinant. It repeatedly computes $a_i = u^t A^i v$ for random vectors u, v and powers of A until a recurrence for a_i is found using the Wiedemann algorithm. Each iteration is performed a matrix-vector product and can be done in $O(\text{NNZ})$ (number of non-zero entries) time, so for the up to $2n$ iterations that are required, the total work performed is $O(n \times \text{NNZ})$.

The Coppersmith Block Wiedemann [11, 25] variant instead computes $a^i = U^t A^i V$, where U, V are random panels m columns wide. For this computation, m applications of sparse matrix-vector product are performed for each iteration, i , thus achieving faster matrix multiplication, particularly in panel cases, is of significant importance. The Block Berlekamp-Massey algorithm can be used to find this recurrence. Probability of success depends on the matrix, field, and random vectors chosen. Matrices called preconditioners exist that are applied before and after each computation of a_i that increase these probabilities. Each of these, in turn, requires multiple sparse matrix-vector multiplications. Chen et al. [10] presents structured preconditioners for various problems and discusses the factors influencing the probability of success for each. Thus, algorithms for solution, rank, determinant, minimal polynomial, and potentially characteristic polynomial reduce to finding recurrences of these a_i sequences. Each sequence, in turn requires multiple sparse matrix-vector product computations.

As an illustration of Block Wiedemann in action we next report on the computation of the rank modulo 3 of a quite large matrix. This example serves to emphasize the importance of high performance in sparse matrix vector multiplication. When the needed block size is relatively small, as in this case, matrix vector product cost dominates this rank computation. The setting is the study of 3-regular graphs. Mathematicians have created a number of families of graphs whose 3-ranks in each family satisfy a recurrence relation [41]. It thus was expected to find a recurrence relation on the family designed by Ding and Yuan [13]. This has proven elusive and remains an open problem. Either the recurrence is of high degree or doesn't exist. The i -th matrix in the Ding/Yuan family has dimension 3^{2i-1} . The ninth (dimension $3^{17} = 129,140,163$) is the largest for which we have been able to compute the rank. Because the rank is about one million (it is 968152), two million matrix vector products were needed. These were blocked into 32 at a time. Rather than sparse matrix times a single vector, the basic operation is sparse matrix times a $3^{17} \times 32$ "fat vector", what we call a *panel* in the sequel. The computation became feasible (it ran for only 12 days) when we fashioned a multithreaded matrix panel product that ran in about 10 seconds on four threads, exploiting the highly special structure of this matrix. Thread parallelism is not further explored in this thesis, but the cache friendly blocking that plays an important role in our analyses provides an appropriate granularity for multithreading.

Chapter 2

ANALYSIS OF THE METHOD OF THE FOUR RUSSIANS

In this chapter, we will discuss the challenges a sparse variant of the Method of the Four Russians faces, propose a novel, generic solution, and explore a breadth of work aimed at achieving faster sparse matrix-vector multiplication. Beyond these challenges, our end goal is to find a format and implementation that shows an analogous approximately logarithmic speed-up compared to standard sparse matrix multiplication that the standard, dense Method of the Four Russians sees to classic matrix multiplication. As speed-ups and performance of sparse matrix-vector multiplication are highly data-sensitive, we will comprehensively discuss processing, analysis, and performance trade-offs. At all times we consider not just how design decisions relate to the four pillars of sparse matrix multiplication performance [22], but also how to efficiently utilize the vector registers of modern hardware.

First, we will discuss a general sparse version of the Method of the Four Russians, noting its areas of similarity and difference with the classic, dense algorithm and how this sparsification is reflected in the four pillars of sparse matrix multiplication. From this general sparse problem, we will model a simplified matrix and show that our desired pseudo-logarithmic speed-up may be possible. Next, we will present simple, practical matrix formats and sparse multiplication techniques that directly follow from the dense case.

2.1 Modeling Costs of the Method of the Four Russians

In this section, we will model the application of a sparse, generic version of the Method of the Four Russians. These models rely on our matrix being pre-processed for a particular k , denoting the constant used in the Method of the Four Russians (panel

width). As mentioned, this pre-processing may actually yield better performance, even in some fully dense cases, as the cost of isolating, masking, and shifting bits is expensive compared to basic addition. This extraction process would be untenable when considering any standard sparse matrix format that stores individual elements (bits) as separate matrix entries.

This modeling is not inherently novel. [7] and [2] both model the costs of the Method of the Four Russians for dense matrix-matrix multiplication, with the latter going into far more detail than we will in this work. As our goal is to develop a practical, applicable version of the Method of the Four Russians, analysis on a fictitious random matrix with uniformly distributed entries is only a starting point. These models will provide some insight into the baseline conditions in which it would be feasible to attempt to apply the Method of the Four Russians to sparse matrix-vector multiplication. However, we note that we can extend these models to matrices with various distributions of entries, provided that we only assume uniform distribution within each panel. We note that within a single panel, we believe that a uniform distribution of entries is likely a pathological case for the sparse Method of the Four Russians. Non-uniform distributions of entries are more likely to have longer patterns and thus more empty rows at the same density. For a given panel width, k , longer patterns yield increased speed-ups by applying the Method of the Four Russians, while more empty rows leads to fewer operations performed in the sparse case.

In the following subsections, we will model the arithmetic cost of a dense Method of the Four Russians, followed by a sparse Method of the Four Russians in terms of number of additions. As we have stated, our goal is to show that there are dimensions and densities that would be amenable to a sparse variant of the Method of the Four Russians, so these incomplete models of unlikely matrices serve as a starting point for future work, not as concrete claims of success or failure.

To be precise, we will use Listing 2 as our baseline algorithm for computing lookup tables and Listing 1 as our baseline algorithm for our sparse Method of the Four Russians multiplication. We note that these are perhaps not strictly optimal, as

```

1  #include <stdint>
2  using DT = std::uint64_t;
3  using M = std::uint64_t *__restrict;
4  using T = DT *__restrict;
5  #define P 1
6
7  void build_table(T, std::size_t, T);
8  void mul_panel(M, T, T, std::size_t, std::size_t);
9
10 void mul_four_russians(M A, T B, T C, std::size_t m, std::size_t n,
    ↪ std::size_t k, std::size_t stride) {
11     auto table = new DT[P*(1 << k)];
12     for (auto i = 0; i < n; i += k) {
13         build_table(B + i*P, k, table);
14         mul_panel(A+i*stride, table, C+i*P, stride, bit_offset);
15     }
16 }

```

Listing 1: Skeleton of the Method of the Four Russians. For each panel of k columns of A and rows of B , we compute all 2^k combinations of these rows of B into a lookup table and then use k – *bit* entries of A as indices into this table, rather than separately performing k operations for each row in a panel.

```

1  inline void build_table(T B, std::size_t k, T table) {
2      for (auto i = 0, start = 1, stop=2; i < k; ++i, start<<=1,
    ↪ stop<<=1) {
3          for (auto j = 0; j < P; ++j) {
4              table[start*P + j] = B[i*P + j];
5          }
6          for (auto j = 1; j < start; ++j) {
7              for (auto l = 0; l < P; ++l) {
8                  table[(j+start)*P + l] = table[j*P + l] + B[i*P + l];
9              }
10         }
11     }
12 }

```

Listing 2: One possible method of computing all 2^k combinations of rows from B to build a lookup table in the Method of the Four Russians.

computing all combinations of 2^k rows requires $2^k - k - 1$ additions, as no addition is required in the zero-row or in any of the k rows consisting of combinations of one row and that the initial application of the lookup table could be a copy, not a true binary addition. In practice, the extra additions in the table construction are easily omitted, while the difference of one copy compared to one addition over a large sparse matrix is inconsequential. In terms of Method of the Four Russians details, we note that these table constructions differ from those of [1], as we do not build our tables in Gray Code order. However, the total number of operations required is the same, so for the purposes of evaluating costs, this does not matter.

Finally, we make note of the following conventions in this section: k denotes the constant of the Method of the Four Russians (width of a panel), A denotes our binary (pre-processed) sparse matrix and is of dimension m -by- n , B denotes our right-hand operand (a vector or panel, in our particular use cases) and is of dimension n -by- p , and C is our product of dimension m -by- p .

2.1.1 Dense Approximations

For purposes of comparison, we will present simple models of the costs of a dense Method of the Four Russians. First, we will use a standard dense matrix representation. Then, we will compare this to a sparse representation, of presumed full density (i.e., we read all entries, independent of value). These two formats require the same number of domain operations, as the same lookup tables are generated and the same number of applications of each table are performed. Regardless of format and density, for a given constant, k , all lookup table generation is the same in both arithmetic operation count and memory accesses. The only difference arises when we consider the number of items read from the matrix and the corresponding number of applications of our generated lookup tables using these entries. Our subsequent models of a sparse Method of the Four Russians will be adaptations of this sparse representation.

Through Figure 2.1, we denote the costs of panel-wise and complete applications of the Method of the Four Russians with a dense matrix. The arithmetic cost is

Arithmetic cost of generating a lookup table	$=p(2^k - 1)$
Arithmetic cost of generating all lookup tables	$=\frac{np(2^k - 1)}{k}$
Arithmetic cost of applying one panel of dense matrix A	$=mp$
Arithmetic cost of applying all panels of dense matrix A	$=\frac{mnp}{k}$
Total arithmetic cost of to multiply dense matrix A	$=\frac{np(m + 2^k - 1)}{k}$

Figure 2.1: The cost, in number of additions, to apply the densely-represented m -by- n matrix A to the n -by- p operand B using the Method of the Four Russians.

independent of matrix representation. This is a simplified model, requiring $2^k - 1$ operations to generate all 2^k combinations of rows. Then, we use this table once for each row within each panel. We note that the combination of zero rows of B requires no operations. Optimally, additions may be omitted when “computing” combinations of single rows, but it is notationally simpler to ignore this. For further simplicity, we assume that k divides n . Further, for our IO considerations, we will assume that k also divides the number of bits in a machine word so that a read of k -consecutive bits are performed in a single operation.

To select an “optimal” value for k , we can take the partial derivative of the equation in Figure 2.1 with respect to k and find a root of this partial derivative, with the assumption of positive matrix dimensions.

In Figure 2.2, we illustrate this process of finding an optimal value for k to minimize our operation count. This yields a value of $k = \frac{W(\frac{m-1}{e})+1}{\log(2)}$.¹ This is not quite the logarithmic value that often appears in literature concerning the Method of the Four Russians, but it is quite close. In Theorem 1, we note that for sufficiently

¹ W is the Lambert W function, also known as the product logarithm [12]

$$\begin{aligned} \frac{\partial}{\partial k} \frac{np(m + 2^k - 1)}{k} &= \frac{np(-2^k + 2^k \log(2) - m + 1)}{k^2} \\ \text{and setting} & \\ \frac{np(-2^k + 2^k \log(2) - m + 1)}{k^2} &= 0 \\ \text{yields} & \\ m = 2^k(k \log(2) - 1) + 1 & \\ \text{and solving for } k \text{ yields} & \\ k = \frac{W\left(\frac{m-1}{e}\right) + 1}{\log(2)} & \end{aligned}$$

Figure 2.2: Finding a solution for k from the partial derivative of the cost found in Figure 2.1.

large values of m (say, greater than 8), the product logarithm, $W(x)$, is bounded by $\log x - 0.5 \log \log x$. Ignoring this $0.5 \log \log x$ term, we are left with $k = \frac{1 + \log m}{\log 2}$, effectively the $\log_2(m)$ choice that is popular. However, this process also illustrates why most experimental choices of k are usually less than $\log_2(m)$. For instance, Albrecht [2] decided on $0.75 \log_2(m)$, as a good choice by experiment, while Bard [7] did indeed note that $O(\log m - \log \log m)$ was a good fit for experimental results in modeling his Method of the Four Russians for matrix inversion. We point out that the standard $\log_2(m)$ speedup typically comes from equating the two halves of our total cost: when $\frac{n2^k}{k} = \frac{mn}{k}$ when $k = \log_2(m)$.

Theorem 1. *When $m \geq e^2 + 1$, $k = O(\log m - \log \log m)$, directly following from bounds of the Lambert W function in [12].*²

Substituting this value of k back into our equation model, we see that we obtain the roughly logarithmic speedup that the Method of the Four Russians is known for.

² $\log x - \log \log x < W(x) < \log x - 0.5 \log \log x$, when $x \geq e$.

Number of reads from matrix per panel	$=kp$
Number of reads from table per panel	$=2p(2^k - 1) - kp$
Total reads per panel	$=2p(2^k - 1)$
Total reads	$=\frac{2pn(2^k - 1)}{k}$
Number of writes to table per panel	$=p(2^k - 1)$
Total writes	$=\frac{np(2^k - 1)}{k}$

Figure 2.3: The cost, in approximate reads and writes, to build lookup tables in the Method of the Four Russians.

Now, our next objective is to see how or if this selection of k changes as we modify our model.

In Figures 2.3 and 2.4 we note the number of reads and writes performed in both representations of the dense Method of the Four Russians. The number of writes corresponds to the number of arithmetic operations performed, while twice as many reads are required because our additions require three operands: two sources and a destination. Where these values begin to differ is when we consider how we read k -bit entries from our source matrix.

In Figure 2.5, we illustrate the significant difference in the number of reads required to read the matrix, comparing reading k bits at a time to extracting single bits at a time from a coordinate format sparse matrix. Without some method of reading k bits efficiently, a very naïve implementation of a sparse Method of the Four Russians for matrix-vector multiplication ($p \approx 1$) will be dominated by cost of reading the matrix. Thus, for any practical implementation, the sparse matrix *must* be pre-processed so that these k bits can be read in a single operation, or, at the very least, in no more than p operations to match the cost of reads using the generated lookup tables. Further, we must be able to access these entries in some efficient ordering, presumably

Number of reads from table	$= \frac{mnp}{k}$
Number of reads from destination	$= \frac{mnp}{k}$
Number of writes to destination	$= \frac{mnp}{k}$

Figure 2.4: Approximate count of reads and writes using the lookup table and destination matrix in the Method of the Four Russians. This, with Figure 2.3 represent all costs except reading entries from matrix A .

Number of reads for k bits from dense A	$=1$
Number of reads for k bits from COO sparse A	$=3k$
Total number of reads when reading dense A	$= \frac{mn}{k}$
Total number of reads when reading COO sparse A	$=3mn$

Figure 2.5: Number of reads required to read matrix A in a dense representation or a coordinate format sparse representation, assuming k divides n and k divides the number of bits per machine word.

Total writes	$= \frac{np(m + 2^k - 1)}{k}$
Shared reads	$= \frac{2np(m + (2^k - 1))}{k}$
Dense format, total reads	$= \frac{mn + 2np(m + 2^k - 1)}{k}$
Sparse format, total reads	$= \frac{3mnk + 2np(m + 2^k - 1)}{k}$

Figure 2.6: Number of reads and writes in the dense Method of the Four Russians.

in exactly the order required when performing the Method of the Four Russians.

In a standard dense application of the Method of the Four Russians, we are performing strided reads from our matrix: the i -th k bits from row 1, then the i -th k bits from row 2, and so on, followed by the $(i + 1)$ -st k bits from row 1, then the $(i + 1)$ -st k bits from row 2, and so on. High-performance implementations of the Method of the Four Russians are aware of this and prefer to work on cache-friendly size matrices to ameliorate this [1]. This sort of strided access is inherently incompatible with standard COO, CSR, or CSC sparse formats, necessitating some form of panel-ordering as opposed to row- or column-ordering.

Following this same procedure for optimizing k that we performed with our arithmetic model, we will find values of k considering optimizing the number of writes, reads, and combined writes and reads. Notably, optimizing the number of writes is identical to optimizing the arithmetic count, as we only perform write operations when performing additions. In Figure 2.6, we list our final values of memory operation counts before beginning our minimization process.

In Figure 2.7, we obtain $k = \frac{W(\frac{2pm+m-2p}{2ep})+1}{\log(2)}$ as a value to minimize the number of reads we perform when using a dense representation. As before, choice of k is

$$\frac{\partial}{\partial k} \frac{mn + 2np(m + (2^k - 1))}{k} = \frac{2^{k+1}np \log(2)}{k} - \frac{2np(2^k + m - 1) + mn}{k^2}$$

Setting

$$\frac{2^{k+1}np \log(2)}{k} - \frac{2np(2^k + m - 1) + mn}{k^2} = 0$$

yields

$$k = \frac{W\left(\frac{2pm+m-2p}{2ep}\right) + 1}{\log(2)}$$

Figure 2.7: Finding a k that minimizes the number of reads for a dense representation.

evidently independent of the number of columns of A , but this model instead depends on both m and p . When considering the two “extreme” cases of $p = m$ and $p = 1$, this simplifies to $k = \frac{W\left(\frac{2m-1}{2e}\right)+1}{\log(2)}$ and $k = \frac{W\left(\frac{3m-2}{2e}\right)+1}{\log(2)}$, respectively. Provided, again, that we have m more than a few entries, the first expression is nearly the same as it was in the arithmetic case ($\frac{2m-1}{2e}$ compared to $\frac{m-1}{e}$), while the vector case suggests that we may want to consider a k that is larger than what we would choose for the square case, balancing the relative decrease in work in building and applying the lookup tables with the relatively more expensive reads. While this is not a typical application of the Method of the Four Russians, this analysis suggests that using the Method of the Four Russians for matrix-vector arithmetic should still yield similar speedups, even though we are performing relatively less work in the vector case.

We see in Figure 2.8 that changing the model to incorporate the total number of reads and writes unsurprisingly carries threes rather than twos into our value for k . For particular values of m , this may involve rounding differently when computing k , but the general costs are similar. This form suggests that we could readily model combined reads and writes given a relative cost between the two, by accordingly adjusting our coefficient. While the total costs may change, the selection of k is going to change very little in this model.

$$\frac{\partial}{\partial k} \frac{mn + 2np(m + 2^k - 1)}{k} = \frac{3 \times 2^{k+1}np \log(2)}{k} - \frac{3np(2^k + m - 1) + mn}{k^2}$$

Setting

$$\frac{3 \times 2^{k+1}np \log(2)}{k} - \frac{3np(2^k + m - 1) + mn}{k^2} = 0$$

yields

$$k = \frac{W\left(\frac{3pm+m-3p}{3ep}\right) + 1}{\log(2)}$$

Figure 2.8: Finding a k that minimizes the number of reads+writes for a dense representation.

$$\frac{\partial}{\partial k} \frac{3mnk + 2np(m + (2^k - 1))}{k} = \frac{2np(-2^k + 2^k k \log(2) - m + 1)}{k^2}$$

Setting

$$\frac{2np(-2^k + 2^k k \log(2) - m + 1)}{k^2} = 0$$

yields

$$k = \frac{W\left(\frac{m-1}{e}\right) + 1}{\log(2)}$$

Figure 2.9: Finding a k that minimizes the number of reads for a sparse representation storing a dense matrix.

In our model of the sparse representation of the dense Method of the Four Russians, requiring us to perform three reads per bit from a sparse coordinate format, we see in Figure 2.9 that the optimal selection of k is identical to that in the arithmetic count model. Our choice of k is strictly driven by the dominant term of the leading $3mnk$ in this model and accounting for writes in addition to reads does not change our choice of k . We reiterate that this choice of k determines our speedup, not the total work done, as this format requires more reads than the dense format, so even with the same speedup, we are still relatively slower, as the total work is higher.

Ultimately, these dense models of arithmetic, read, and write operation counts are presented as comparison points for our following analyses with sparse formats and sparse data. However, these models suggest that optimal choice of k is fairly robust and will be likely depend solely on m in both the vector and matrix operand cases, although particular changes may result in rounding differences or ± 1 . Our aim is to show that we can select a similar value of k in the true sparse matrix case.

2.1.2 Sparse Operation Approximations

In this subsection, we model the costs of a sparse Method of the Four Russians in terms of number of additions and number of memory operations performed, as shown above with a dense version. We assume that $A \in \{0, 1\}^{m \times n}$ with density d and uniformly distributed entries and $B \in D^{n \times p}$.

As mentioned in the introduction to this section, we assume that matrix A has been pre-processed in such a way that we read entries in the order they are used when applying our lookup tables. Given the abstract nature of working with random matrices, we will discuss our costs in terms of expected numbers of operations within an arbitrary panel and carry this through for each panel.

We make the following assumptions about our sparse matrix format, which will be expanded upon in Section 2.2: we have a modified CSC-like format. This format contains the vector KA , of length $1 + \frac{n}{k}$ that incrementally counts the number of entries in each panel. IA stores the row index of a k -bit entry, while A stores the k -bit value.

Thus, IA and A both have lengths equal to the number of non-zero k -bit entries within our sparse matrix.

Our significant difference in modeling this sparse version is that we must determine the expected number of entries within any given panel in our sparse matrix. In our dense case, we read all m k -bit entries or all $m \times k$ single-bit entries within a panel, but in a real sparse representation, we will only read those entries that are present.

To that end, we will use a naïve model of our uniformly distributed random matrix. Within any particular panel, we anticipate $\frac{kmd}{n}$ entries. An individual bit within a row within a panel is expected to be non-zero, then, with probability d . Thus, counting the number of non-zero k -bit entries is the task of finding n minus the number of expected zero entries within a given panel. The expected probability of a k -bit entry being 0 is $(1 - d)^k$, so the expected number of *zero* entries within a panel is $m(1 - d)^k$, while the expected number of *non-zero* entries is then $m - m(1 - d)^k$. For the purposes of applying the Method of the Four Russians, the cost is independent of the number of set bits > 1 within a particular k -bit entry, although the more set bits we have within an entry, the more work we save by generating our lookup tables.

Thus, when reading from or applying with entries in a panel, we expect to encounter $m - m(1 - d)^k$ entries, carried forward to $\frac{n(m - m(1 - d)^k)}{k}$ expected entries within the entire matrix. Thus, for a 10,000-by-10,000 matrix that is 5% dense with $k = 10$, we expect our 5,000,000 non-zero entries to be contained within approximately 4,012,631 10-bit entries.

With these assumptions in place, we can begin our model of the number of arithmetic operations required to apply a sparse Method of the Four Russians in a sparse matrix-vector product. As in the dense case, we will perform the same work in constructing our lookup tables. However, we now perform the $\frac{n(m - m(1 - d)^k)}{k}$ expected additions, rather than the $\frac{mn}{k}$ additions in the fully-dense case. We illustrate this by listing the number of expected additions performed in Figure 2.10.

As before, we would like to take the partial derivative of this total sum with respect to k , equate this to zero, and find a closed-form of k . However, due to the

Additions per table	$=p(2^k - 1)$
Total additions in table generation	$=\frac{np(2^k - 1)}{k}$
Table applications per panel	$=m - m(1 - d)^k$
Table additions per panel	$=p(m - m(1 - d)^k)$
Total table applications	$=\frac{np(m - m(1 - d)^k)}{k}$
Total additions	$=\frac{np((2^k - 1) + m - m(1 - d)^k)}{k}$

Figure 2.10: Summary of expected costs in the sparse Method of the Four Russians, for a random matrix with uniformly distributed entries.

dependence on k for the number of entries within a panel, finding a closed-form solution to this is intractable. The closest our analytical solver will take us is $2^k k \log(2) + 1 = 2^k + m + (1 - d)^k m(-1 + k \log(1 - d))$, for which it is unable to isolate k . However, our solver is able to rearrange this into the equation $m = \frac{1 - 2^k + 2^k k \log(2)}{1 - (1 - d)^k + (1 - d)^k k \log(1 - d)}$, when $d < 1$.

With no closed form solution, we will consider our requirements for this model to show a speedup over classical multiplication, requiring $\frac{np(2^k - 1 + m - m(1 - d)^k)}{k} < mnpd$. In other words, the Method of the Four Russians will only be advantageous if our format reduces the number of entries in our matrix by at least the number of operations spent building lookup tables.

With this stipulation, we have the constraint $2^k < m((1 - d)^k + dk - 1) + 1$. Further, we claim that $(1 - d)^k + dk < dk + 1$, simplifying our inequality to $2^k < mdk + 1$, which we can rearrange to $\frac{2^k - 1}{k} < dm$, which is close to the $k < \log(dm)$ we hope to see.

Given the interdependence between the number of entries in a panel and k , we need some alternative to $m - m(1 - d)^k$ to represent this term. Instead, we will

$$\begin{aligned}
\text{Number of additions} &= \frac{np(2^k - 1 + \alpha dm)}{(1 + \alpha)k} \\
\frac{\partial}{\partial k} \frac{np(2^k - 1 + \alpha dm)}{(1 + \alpha)k} &= \frac{-np(\alpha dm + 2^k - 2^k k \log(2) - 1)}{(\alpha + 1)k^2} = 0 \\
&\text{implies} \\
2^k(k \log(2) - 1) + 1 &= \alpha dm \\
&\text{solving for } k \text{ yields} \\
k = \frac{W\left(\frac{\alpha dm - 1}{e}\right) + 1}{\log(2)}
\end{aligned}$$

Figure 2.11: Finding a solution for k with our relaxed model of costs in the Sparse Method of the Four Russians.

introduce some parameter α , with the claim that the number of entries we will ever consider within a panel is αdm . Similarly, we may require additional blocks, so we will have $(1 + \alpha)k$ panels. This gives a new model of $\frac{np(2^k - 1 + \alpha dm)}{(1 + \alpha)k}$.

In Figure 2.11, we repeat our optimization process with this less exact, new model, and obtain $k \approx \log_2(\alpha dm) - \log_2 \log_2(\alpha dm) + 1 \approx \log_2(\alpha dm)$, for some $0 < \alpha < 1$. While not precisely modeling the exact arithmetic cost, we believe this model suggests that we can choose k proportionally to some percent of dm .

Rather than solely rely on this very crude approximation, given m and d , Mathematica is quite capable of numerically solving for k for our exact model, so instead we will present some numerical solutions for selected row dimensions and densities. We illustrate these in Table 2.1. We stop evaluations once we hit a $k = 1$, although we realistically need $k \geq 2$ to obtain a speedup, as we cannot compute all $2^{1.32}$ combinations of rows with much success. This numerical solution suggests that for $m = 1000$, we need a matrix that is at least 4% dense. With 10,000 rows, we only require a density of about 1.5% to have $k \geq 2$.

Ultimately, we believe these models, however crude, show that despite the sparse

m/d	0.1	0.05	0.04	0.03	0.02	0.01	0.0028	0.001	0.0005	0.00009
1,000	5.33	3.18	2.39	1.32						
10,000	8.86	7.05	6.38	5.46	4.09	1.52				
100,000	12.27	10.68	10.08	9.25	8.01	5.74	1.03			
1,000,000	15.60	14.19	13.64	12.87	11.72	9.59		1.52		
100,000,000	22.17	21.04	20.56	19.88	18.84			9.67	7.36	1.10

Table 2.1: Numerically solved solutions for k from Figure 2.10 for select values of d and m .

context, there is some theoretical backing that suggests that a Method of the Four Russians can be advantageous in a sparse environment. As is the case in most dense implementations, practical considerations will dictate our sparse implementations. In the remainder of this chapter, we will continue to develop theoretical and practical details concerning a sparse Method of the Four Russians.

2.2 Sparse Methods of the Four Russians

The Method of the Four Russians offers a speedup for matrix multiplication because it reduces the amount of repeated work in the dense case. By precomputing all binary linear combinations of k rows, we save operations by not recomputing combinations of rows if these combinations occur more than once and we further save in the event of combinations containing more than one row (e.g., if the entry of the matrix is 11101, we perform only one add instead of four after building the table). Thus, if the entry 11101 occurs multiple times within a panel of five columns, we still only require one addition each time it occurs, rather than four. For sparse matrices, we are both less likely to have repeated entries within a panel row and less likely to have panel rows with multiple occurrences.

For sparse matrices, we would like to make k much larger, as we have fewer entries present, so we need to bring in additional columns to our panels for benefit. However, this would require even more unnecessary work when computing lookup tables. For example, we may want our panel width to be 16 for our sparse matrix to divide our matrix into as many panels as a smaller, dense matrix with a width of,

say, 6-8. We cannot generate all 2^{16} combinations of rows. To address this issue, we consider building partial tables. Rather than all 2^k combinations, we consider a couple of approaches that result in smaller tables. First, we consider generating lookup tables containing up to k entries. To do thus, we introduce a second parameter, t , controlling the width of a panel, while k refers to the number of entries in a portion of this panel. For example, if $t = 20$ and $k = 3$, we build a lookup table containing all combinations of 1, 2, and 3 entries for a total size of $\binom{20}{1} + \binom{20}{2} + \binom{20}{3} = 1350$ combinations of rows. We will later discuss more details on these parameters.

The second approach is to analyze the entries within a panel and record only the patterns that occur. Then, we generate a list of instructions to produce a lookup table containing only these patterns. If very few patterns occur within the matrix, then we have huge opportunity for speedup at the cost of a more complicated table-building process.

In both of these cases, we also lose the original values for space efficiency and efficiency when using values as lookups into our tables. For example, if $t = 5$ and the only patterns that occur are 10001, 11111 and 11110, we replace occurrences of these values with 0, 1, and 2, corresponding to the first three entries in the lookup table we generate.

A standard CSR representation of a sparse matrix might dedicate one 64-bit integer for each entry, one 64-bit integer for each column index for each entry, and $N_{rows} + 1$ 64-bit integers for the row counts. A standard 2-by-1 blocking strategy might dedicate 128-bits for every two entries, one 64-bit integer for every other entry, and maintain the same row counts. Over GF_2 , we might drop this requirement to one 8-bit integer for each entry, but the other costs remain the same. However, we wish to use less storage, thus we compress multiple entries into one integer, since we do not need to access individual entries at any point in the Method of the Four Russians. By compressing entries, we can remove our column storage entirely and require just one 8-64 bit integer for every row in a panel. The only additional information we need is to keep track of the number of rows within each panel, which requires $N_{panels} + 1$ 64-bit

1	1	1	1	0	1
0	1	0	0	1	0
1	1	0	1	1	1
1	1	1	1	0	0
1	0	0	1	0	0
0	0	1	1	1	0

Table 2.2: A 6-by-6 matrix with 21 non-zero entries over GF_2 : $t = k = 1$.

integers. For the example where $t = 5$ in the previous paragraph, our compressed storage would require just one byte for each 10001, 11111, or 11110 while not requiring any column indices. Where a normal CSR representation would require 11 bytes for entries and 11 64-bit integers for column indices, we use only 3 bytes for entries.

2.2.1 Fixed-Size Lookup Tables

Generating fixed-size lookup tables can be viewed as an extension of unaligned block CSR (UBCSR). The standard idea of blocking is not inherently useful for us when working over very small fields. Since we can fit multiple entries into a single machine word, our blocking places multiple entries in one word. This has the added benefit of simplifying the Method of the Four Russians which extracts multiple entries at once from a row in a matrix.

After selecting parameters t and k , we reorganize our matrix into panels of t columns. Each panel is in row-major order. Each row within a panel is broken into segments of up to k non-zero entries. For example, if we have 8 non-zero entries within a row and $k = 3$, we break this row into segments of 3, 3, and 2 non-zero entries. Each of these segments is replaced by a value corresponding to a row of our partial lookup table. If our first segment is 1110..., it is assigned the row index of the row in the lookup table containing the sum of the first three rows. The exact value of this will depend on t and k . For example, if we have $t = 22$ and $k = 2$, then there are 253 possible values. Each non-zero segment is replaced by one of these 253 values.

1	1	1	1	0	1
0	1	0	0	1	0
1	1	0	1	1	1
1	1	1	1	0	0
1	0	0	1	0	0
0	0	1	1	1	0

Table 2.3: The same 6-by-6 matrix in Table 2.2, partitioned into panels of two columns with up to two non-zeroes per row within a panel: $t = k = 2$.

1	1	0	1	0	1
0	0	1	0	1	0
0	1	0	1	1	0
1	1	0	0	0	1
1	1	0	1	0	0
0	0	1	1	0	0
1	0	0	1	0	0
0	0	1	1	1	0

Table 2.4: The same 6-by-6 matrix in Table 2.2 and Table 2.3, partitioned into panels of three columns with up to two non-zeroes per row within a panel: $t = 3, k = 2$. Some rows within a panel have multiple segments.

To illustrate these storage formats, we refer to Table 2.2, Table 2.3 and Table 2.4. The reader can extend these small examples to much larger matrices. Some matrices and parameter selections will yield smaller or larger costs. Table 2.2 shows a simple 6-by-6 matrix with 21 non-zero entries. In Table 2.3 we separate this matrix into panels of 2 columns each ($t = 2$) with up to 2 entries per row within a panel ($k = 2$). In this table, we have only 14 non-zero entries, as we pack two entries into one integer. Our Four Russians lookup table will contain 3 entries for each panel: 01, 10, and 11. Building this table requires only one operation: adding two rows together. Thus, multiplying this with a vector will require 1 operation for each panel (3 operations total) to build the lookup tables and 14 total operations to add using the lookup tables, for 17 total operations, 4 fewer than the standard matrix.

In Table 2.4, we separate the matrix into panels of 3 columns each ($t = 3$) with up to 2 entries per row within a panel ($k = 2$), thus some rows within a panel have multiple entries. For example, the first row of the first panel is 111. When we limit an entry to only two set bits, we must break this 111 into 110 and 001. Each horizontal line in Table 2.4 corresponds to a row in our original matrix. Two entries within a row notes that we have broken our $t = 3$ -bit entry into two, up to $k = 2$ -non-zero entries. In this representation, we still have 15 non-zero entries. Building a lookup table requires computing all combinations of $\binom{3}{1} + \binom{3}{2}$ rows. This will require 3 additions for each lookup table to compute 110, 101, and 011. With two tables each requiring 3 operations and 15 total entries, we require 21 operations to multiply this matrix by a vector.

Then, our CSR format needs to record the number of segments that occur within a row, rather than the number of entries. We further save by not needing to record column indices, as these are indirectly captured by the segment-numbering scheme. If necessary (and it is not necessary for our Four Russians-based SpMV multiplication), we can produce a mapping from an integer to the component column indices when generating the lookup table. Indeed, when working with bitpacked, bitsliced, or our custom storage in sparse matrices, we do not want to store column indices unless necessary, as the size of a column index is considerably larger than the size of an entry.

Performing multiplication with this matrix is straightforward. For each panel, we compute our partial, fixed-size lookup table. We then iterate across each row of the panel and each segment within a row and use this segment as an index into our lookup table.

If $t = k$, then this approach effectively degenerates to the dense Method of the Four Russians with the additional overhead of our modified CSR format recording the number of segments in each row in each panel and the number of segments in each panel. If $k = 1$, then this approach is equivalent to standard CSR multiplication, as our generated lookup table contains no combinations of rows, only single rows. Thus, selection of t and k will depend on the density, distribution of non-zero entries, and dimension of the sparse matrix.

Additionally, note that k directly limits how effective this approach is. If $k = 2$, and our matrix is fully dense, each row within the panel will require $t/2$ lookup table additions. Further, each segment only captures k non-zero entries, so our storage requirements increase as k decreases. However, we can never require more storage than standard CSR, as, if $k = 1$, we have a standard CSR format matrix. The primary trade-off is that our lookup table size increases as k increases. Thus, we require a matrix suitably dense that we have multiple non-zero entries occurring within a row in a panel frequently enough that our selection k matters. Further, selection of t and k have practical considerations, as the number of bits needed to represent the total number of combinations determines the integer type to store a segment. If $t = 22$ and $k = 2$, for instance, we have 253 possible combinations, so one byte can be used for an entry. If $t = 10$ and $k = 3$, we still require just one byte, as we have fewer than 257 combinations. As with the dense Method of the Four Russians, it is also critical for performance that the lookup table fit into a low level of cache.

Actual parameter selection is difficult to predict, as t , k , number of rows, number of columns, and density all contribute. Considering a 10000-by-10000 uniformly dense matrix with $t = 22$ and $k = 2$, the expected break-even density of this format is 1.1% dense. We can estimate this by considering the frequency we see each of our 2^{22}

patterns in a uniform distribution. At this density, a CSR representation will require 2.3 to 3.3 MB of storage (depending on whether 1 or 8 bits per entry are used) and require 1,100,000 row operations (for a vector right-hand side, this is just one operation, but for a panel, it might be several operations) to apply the matrix to the vector. The fixed-sized lookup table approach will require less than 1.1 MB of storage for the sparse matrix (plus the space needed for 253 rows of the look-up tables). Constructing the lookup tables will require around 105,000 row operations (455 panels, requiring 231 operations each for all combinations of two rows), while applying the sparse matrix will require approximately 990,000, for a total number of row operations of about 1,095,000. The cost to generate the lookup table depends only on t and k , while the cost of applying the matrix depends on t , k , dimension, and density.

If we increase density to 3%, then we expect to require 2.4 MB of storage (for the compressed matrix), and about 2,450,000 row operations to apply the matrix with a fixed-size lookup table, compared to approximately 6.4 MB to 9.1 MB (1 bit or 1 byte per entry) and 3,000,000 row operations with a CSR representation. The increase in storage is not uniform, as storing our column indices and row counts requires more memory than storing the values of entries. While an element of GF_2 requires just 1 bit (and may be stored in just a byte), other information such as indices or accumulated non-zero counts in CSR will still require 32 or 64 bits per entry. If we were to set $t = 64$ and $k = 2$, we would still require 3.8 MB of storage and perform only 2,198,000 row operations. The performance trade-offs between matrix size, lookup table size, and number of row operations will depend on hardware. If the matrix is fully dense, we expect to improve field operation count by a factor of k compared to CSR.

Ultimately, these smaller, fixed-size lookup tables can give predictable performance for uniformly random matrices of modest densities. Further, we can compute expected performance for chosen parameters when given density and dimension. For matrices with very few non-zeros per row or with few patterns repeated many times, this approach is unlikely to be as beneficial.

1	1	1	1	0	1
0	1	0	0	1	0
1	1	0	1	1	1
1	1	1	1	0	0
1	0	0	1	0	0
0	0	1	1	1	0

Table 2.5: The same 6-by-6 matrix in Table 2.2, 2.3, and 2.4, partitioned into panels of three columns per row within a panel: $t = k = 3$.

2.2.2 Pattern-Generated Lookup Tables

Posited by Dumas et al. [14], most matrices used in exact linear algebra problems have some form of exploitable structure. We can exploit repeated patterns within panels in a Method of the Four Russians-based approach. Rather than construct all row combinations or a fixed number of row combinations, we select a panel size and enumerate only the patterns that occur within the matrix at this panel size. After finding these patterns, we generate an optimal list of instructions that produce these patterns. As with the fixed-size tables, we then replace entries in the panel with the corresponding row of the lookup table. In this case, however, an entry is going to be a t -bit pattern, as we have only one or zero patterns in each row of a panel.

Unlike the fixed-size tables that work best over uniform distributions of elements, these pattern-generated tables require the underlying matrix to be highly structured to be efficient. By this, we mean that each pattern used must appear repeatedly within a panel. Also unlike the fixed-size table approach, this approach requires a very expensive analysis stage. For each t we consider, we must reorder our matrix into panels of t columns. We then record all t -bit patterns that occur across all panels. After recording all patterns, we then generate a list of instructions to generate these combinations. Before this step, it is difficult to predict the cost of building a lookup table. Currently, the reordering, conversion, and instruction generation process for a single value of t is very expensive compared to the data manipulation of the fixed-pattern approach. However, for some matrices it may be worthwhile to consider this.

As an example, if we refer back to Table 2.3, we see that there are three patterns that occur across the panels: 10, 01, and 11. Thus, for each panel we generate only these patterns, requiring one operation for each panel to add two rows for the pattern 11. This yields an identical situation to our previous case for $t = k = 2$. Instead, let us look at the case where $t = 3$ shown in Table 2.5. Here, we note that the pattern 011 does not occur. So instead of generating all seven non-zero patterns of three rows, we instead only generate the six we need, requiring just three operations to compute 110, 101, and 111 (which can be built from 110 and 001). By generating only the patterns that occur, this saves us one operation for each panel. Thus, we have 12 entries and two panels requiring three operations each, for 18 total operations, compared to 20 operations if we generated all patterns. Even this modest decrease in field operations is significant, as we perform $O(n)$ sparse matrix-vector multiplications for an n -by- n matrix. Matrices with highly repeated patterns could realize larger speedups, while those with very few repeated patterns could see worse performance.

2.2.3 Combining Fixed-Size and Pattern-Generated Lookup Tables

We also consider combining the ideas behind both approaches. We reformat rows of panels to combine k entries into one, as with fixed-size tables. Among these patterns, we only populate the lookup table with patterns that occur. While random matrices would have many such patterns still present, structured matrices may have relatively few.

For example, consider the matrix shown in Table 2.4, where $t = 3$ and $k = 2$. Before, we generated all three patterns of two entries of length three. However, the pattern 011 does not occur in this matrix. Thus, if we generate only the patterns 110 and 101, we require only two operations for each of our two panels, yielding a total number of operations of 19 (15 entries and 2 operations per panel) to multiply this matrix by a vector.

Using the fixed-size lookup table may be beneficial over a full lookup table, in cases where we decrease the size of the lookup table significantly. Since our entries

have a fixed size, the cost of computing the patterns and schedule for the patterns is lower. However, if we do not generate all patterns, our cost may be higher. Generating all combinations of k entries from t rows, for example, can be done in a tight loop for a fixed-size table. If we wish to generate only some of these patterns, then we must enumerate a complete list of instructions instead of using a systematic construction. This may result in poorer performance.

The actual number of operations performed when applying the combination of these two approaches is no greater than the number of operations required to apply the fixed-size table approach, though there is extra cost in generating the instructions and reforming the matrix if we have fewer than the maximum number of patterns. Further investigation into combining these approaches is ongoing. For matrices where we can accurately select t and k and quickly generate all patterns, this combination of approaches could yield the highest potential speedups. However, these potential speedups are tempered by the cost of both reforming the matrix into panels and computing occurring patterns. As we saw with Table 2.4, this selection of parameters still requires more field operations than computing all patterns in Table 2.3.

What we ultimately see, however, is that the difficult part of our work involves selecting these parameters. Exhaustively processing a matrix is intractable if we will only see marginal benefits. The actual task of applying our panels of values is no different than with any other format, the real problem is in generating our lookup tables.

Chapter 3

SPARSE MATRIX-VECTOR MULTIPLICATION ANALYSIS

In this chapter, we will delve into theoretical performance aspects of various sparse matrix formats. By theoretical, we mean analysis, not necessarily asymptotic, predicting performance of code without directly benchmarking it. As sparse matrix-vector multiplication is much more data-dependent (dependent on location of non-zero entries) than dense matrix multiplication, objectively evaluating a format is inherently difficult as there are use cases favorable for every format described in Chapter 2 as well as for our sparse Method of the Four Russians variants. Thus, we will introduce a data-agnostic evaluation that measures theoretical peak performance of applying these sparse matrix formats to vectors and thin panels, matching our motivating use case of sparse matrix-vector multiplication.

This data-agnostic approach will use the `llvm-mca` tool developed by Sony ¹ to evaluate the assembly code produced by our selected compilers (Clang 10.0.0 and GCC 10.1) for each sparse multiplication format. Originally developed to track data dependencies, this tool is capable of scheduling its given assembly code and compute expected running time and reciprocal throughput (both measured in cycles), subject to assumptions that our loads and stores operate uniformly efficiently on L1 cache. As blocking for cache performance is commonly applied as discussed in Chapter 1, this approach instead considers a very low-level analysis of the “quality” of the code produced by the compilers and the true costs of performing arithmetic.

Through this process of compiling sparse multiplication kernels and analyzing the assembly code produced by the compilers, we aim to identify qualities of sparse

¹ `llvm-dev` mailing list: <https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html>

matrix formats and multiplication routines that are best-suited to our sparse matrix-vector products. Further, we will show that our sparse Method of the Four Russians formats are no worse than traditional sparse formats, in general, but require stricter assumptions to produce true block versions. Additionally, we will show that the pre-processing necessary to use a sparse Method of the Four Russians ameliorates one of the major performance bottlenecks of a standard dense implementation.

Moreover, this analysis will show that choices of format, loop ordering, block size, data type, vector/panel width, and compiler are all variables to consider when looking to perform a sparse matrix-vector product.

We do not intend this type of analysis to be used in isolation, but in conjunction with some knowledge of the intended operation (i.e., vector or panel multiplication), some knowledge of the distribution of entries, and with some expected costs for pre-processing. For example, while the theoretical per-entry performance of a 4-by-4 block may be higher than that of a 2-by-2 block, these gains will likely only be realized if blocks are suitably dense, on average. Simply padding a 2-by-2 block with twelve zeroes will not result in better performance for the four “actual” operations. However, that 4-by-4 block may perform better than four 2-by-2 blocks, even discounting intangible factors such as requiring fewer loop iterations.

In the upcoming sections, we will present C++ code (trivially convertible to C) and the modeled performance of the primary loop for each format, each considered panel width and data type, and for a selection of block sizes. For each set of analyses, we will discuss what the compilers do and how these analyses compare to those for other formats.

3.1 Introduction

We will perform this data-agnostic analysis using `llvm-mca`, the LLVM machine code analyzer to model the execution time, in cycles, and the reciprocal throughput of given segments of assembly code. This tool, originally developed at Sony, schedules and statically analyzes execution of given code with the knowledge LLVM has about the

target hardware. Thus, for supported instruction sets and CPU targets, we can obtain realistic, execution unit-aware and cost-aware schedules for given code. This tool, at this time, is unaware of cache performance or misses, assuming a fixed, optimal cost for any loads performed. Given that this is a major concern in sparse matrix multiplication, this is unfortunate. However, cache access is necessarily data-dependent and is an issue shared among every sparse format. Instead, we can view this tool as something capable of measuring theoretical peak performance of a given multiplication algorithm.

Our workflow, executed on <https://godbolt.org> is to write a C++ implementation of each of our considered matrix formats, compile these to our target platform, isolate the inner loop, if any such loop exists, in the generated assembly for this algorithm and analyze the execution time of one iteration of this loop using `llvm-mca`. For each format, we will also endeavor to describe any peculiarities or significant costs that may be present, but not captured by this inner-most loop.

We will perform this analysis across a range of vector widths, block sizes (where relevant), and for recent versions of GCC and Clang. In addition to considering the reciprocal throughput of our inner loops and computed running times, we are also interested in what the compilers themselves do to optimize or potentially hinder performance of these sparse matrix multiplication algorithms. For instance, we are interested in seeing when the compilers generate vector instructions or unroll loops. This is important to consider and perhaps something that is often not predicted, as if a compiler generates two versions of a given inner loop, one handling one element and one handling sixty-four elements, the actual execution profile of this may look substantially different with different matrices, while a loop handling eight elements and a loop handling a single element will be less sensitive to the underlying matrix. Thus, we use this process with the understanding that it should be used in conjunction with some level of analysis of actual matrices.

For this work, we will use Clang 10.0.0 and GCC 10.1 as our two compilers as we have access to these compilers outside of the environment on <https://godbolt.org> and they represent the two most-common open-source compilers. We will compile our

C++ code with the options `-Ofast -march=skylake -mtune=skylake -std=c++14`, with the Intel Skylake architecture representing the hardware on which later benchmarks will be performed at the end of Chapter 4. Additionally, we will use `llvm-mca` with the options `-iterations=1 -mcpu=skylake`.

We will test our algorithms against the following panel widths: one 64-bit unsigned integer, four 64-bit unsigned integers, eight 32-bit unsigned integers, one 32-bit float, eight 32-bit floats, one 64-bit double, and four 64-bit doubles. These sizes encapsulate a variety of expected use cases in applying Wiedemann’s algorithm or the block-Wiedemann algorithm and will be hard-coded within our C++ code. For our main use-cases, we expect these vector sizes to be known to or selected by the user at compile time. The unsigned integer cases directly correspond to arithmetic over GF_2 , as integer and bit-wise arithmetic are executed in the same execution units and have the same running time. Thus, a 32-bit (or greater) integer addition can also be viewed as 32 single-element adds over GF_2 . Since we recognize that we are interested in domains other than GF_2 , we specifically include floats here. Notably, the `-Ofast` compilation argument disables strict IEEE-754 floating point rule adherence. For our expected exact linear algebra use cases of using a float to store an entry or entries of a finite field, these relaxations should not pose any problem. However, the results of this `llvm-mca` analysis might not translate to numerical domains that depend on strict adherence to lack of floating point operation associativity.

Additionally, any pointers used in our code will be explicitly marked with `__restrict` or `restrict`. While not C++ keywords, these are commonly-supported compiler extensions that are critical in achieving high-performing code in this context. This keyword allows the compiler to make the assumption that our pointers are not aliases for one another. For uniformity, we will use `std::size_ts` for all index vectors. Compressed entries used in the Method of the Four Russians will be `std::uint64_ts`. For standard formats, we will assume that the data type of the matrix matches that of the panel.

For block formats, we will consider a few small block sizes. GCC is hesitant

about unrolling loops and Clang will seldom unroll loops that process more than 64 elements, so larger sizes are more likely than not going to be similar to very small loops. For CSC, we will only consider n-by-1 patterns, while CSR will consider only 1-by-n patterns. Intermediate sizes and two-dimensional blocks for CSC and CSR may prove quite practical in some cases, but a more exhaustive list of all block sizes for each panel width and type would be overwhelming in this work. For implementations of the sparse Method of the Four Russians, we will consider a broader range of block sizes, as this is novel work.

Our workflow follows. We compile the given C++ code for our particular target, specifying our vector types and widths, in addition to any block sizes present in the algorithm. We analyze the given assembly output and manually extract the innermost loop of the algorithm and give these to `llvm-mca`. We manually tally the number of entries that each loop processes and report this information. For this inner loop, we will report the number of operations performed, the reciprocal throughput of this inner loop, the number of cycles per operation, and any additional local costs. These additional costs are usually once per-row (column, block) and usually involve some re-ordering or distribution of values at the beginning of iterations or horizontal sums of partial results at the end of iterations

In the following sections, we will consider this analysis on classical dense matrix-vector multiplication, the standard sparse matrix formats, block variants of these sparse matrix formats, a dense Method of the Four Russians implementation, and the panel sparse Method of the Four Russians described in Section 2.2. When considering multiple parameters, we will use `DT` to refer to the type of our vector (int or float), `P` to refer to our panel width, `BR` to refer to the width of a block in number of rows, `BC` to refer to the width of a block in number of columns, `M` to refer to the type of a pointer to our matrix, and `I` to refer to a pointer of indices (`std::size_ts`). `DT`, `P` and `BR` and `BC` are compile-time constants and modified as needed to perform the requisite analyses.

3.2 Dense Multiplication Analysis

While not the focus of this thesis, we will begin this chapter by applying our llvm-mca analysis technique to the problem of dense matrix-vector multiplication to illustrate our workflow and to serve as a comparison point when discussing the relative performance of upcoming sparse matrix-vector multiplication.

In this section and later sections, we will refer to some assembly code to illustrate some common operations. One optimization that compilers can provide is to emit code that uses vector registers on hardware that supports them, which is nearly-universal on any hardware on which we are interested in performing sparse matrix-vector products. These vector operations, frequently referred to as SIMD², operate on 128 or 256 bits worth of their underlying data type. For instance, one PADD³ can add eight 32-bit integers to eight other 32-bit integers, typically with the same cycle cost as just a single addition⁴. The logistical downside is that these vector operations work best when loading and storing directly from the given number of packed items. Accessing or manipulating individual entries that are packed into a vector register is relatively expensive. However, successful use of these vector registers is crucial to “fast” arithmetic.

² Single Instruction, Multiple Data

³ Packed ADD Double word integer, <https://www.felixcloutier.com/x86/paddb:paddw:paddd:paddq>

⁴ Agner Fog, Instruction Tables: https://www.agner.org/optimize/instruction_tables.pdf

```

1  #include <cstdint>
2  using DT = std::uint32_t;
3  using M = DT * __restrict;
4  using T = DT * __restrict;
5  #define P 1
6
7  void mul_ijk(M A, T B, T C, std::size_t m, std::size_t n) {
8      for (auto i = 0; i < m; ++i) {
9          for (auto j = 0; j < n; ++j) {
10             for (auto k = 0; k < P; ++k) {
11                 C[i*P + k] += A[i*n + j] * B[j*P + k];
12             }
13         }
14     }
15 }

```

Listing 3: C++ implementation of dense matrix-vector multiplication. DT and P control our vector type and width.

We begin this analysis by considering the textbook multiplication code presented in Listing 3. By convention, m will refer to a row count and n to a column count. As a fairly standard algorithm, we loop over each row of A, then each column of A, then each column of C, computing $C_{i,k} += A_{i,j} \times B_{j,k}$. Although rather simple, this example compiles to 124 lines of assembly in Clang 10.0.1. Notably, we see Clang perform a common optimization: loop unrolling. For each row, Clang performs some checks and then jumps to one of two loops: one handling 64 columns of A (and 64 rows of B) at a time, or one handling just a single entry at a time. If the unrolled loop is chosen, all multiples of 64 in the column count of A are processed, then if at least 32 columns remain, those 32 are processed, before finally jumping into the single-entry loop.⁵ Thus,

⁵ Clang will check to do the 32-column block of code before jumping to the single-entry

if we have 31 columns of A, we do a couple checks and then perform 31 iterations of the single entry loop shown in Listing 5. If we have at least 64, we will do at least one iteration of the unrolled loop shown in Listing 4. In this x86 assembly code, we see a sequence of instructions beginning with “vp,” signifying they are vector (operating in the 128-bit or 256-bit registers) and packed (the operations are repeated for each component 8-, 16-, 32-, or 64-bit element). Our goal here is not to fully discuss all intricacies of SIMD instructions, but we see a similar pattern between the two loop versions: we do some loads (vmovdqu or mov), some multiplications (vpmulld or imul), some additions (vpaddq or add), and then add/increment followed by a comparison and jump. The only structural difference between the two loops is the number of entries read, multiplied, and added. There is one caveat, however. Even if we have identically a multiple of 64 elements, the unrolled loop has not fully computed the dot product. Instead, it has accumulated however many 32-bit multiplications and additions into the vector registers ymm0, ymm1, ymm2, and ymm3 at the end of this loop. Since we are performing a dot product, these 32 entries (eight 32-bit entries per 256-bit vector register) need to be horizontally added together to form a single entry. This is the “other” extra cost we reference in Table 3.1. This additional cost, however, is only “paid” once per row, in this circumstance. In Listing 7, we see how Clang generates code to horizontally sum these 32 partial products into a single entry: two 16-entry additions, then one 16-entry addition to yield a single 256-bit partial result. The vextracti128 instruction separates the upper and lower 128 bits of this 256-bit result into two 4-item components which are added together. The masks and adds then apply the necessary two-by-two and one-by-one element additions to leave the desired result in the lower 32 bits of the xmm0 register which are then copied to the non-vector eax register to be stored. While rather dense and involved, this sort of horizontal addition appears frequently in the assembly output for many of the formats we will consider.

loop, even if not jumping into the 64-column loop.

Vector width	Clang 10.0.0				GCC 10.1			
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1x32-bit int.	64	8	0.45	15	8	1.3	2.63	15
8x32-bit int.	64	42.5	2.83	42/19	64	72	1.44	33
1x64-bit int.	32	24	1.38	13	4	3	5	13
4x64-bit int.	32	25.5	4.06	15	4	4	3.5	13
1xfloat	64	8	0.33	28	8	1	1.75	24
8xfloat	64	43	2.38	46/17	64	120	2.27	47
1xdouble	32	8	0.66	23	4	1	3.5	19
4xdouble	32	20	2.53	22	32	16	0.97	22

Table 3.1: llvm-mca analysis for classical dense multiplication in the i-j-k ordering shown in Listing 3. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.

By comparison, GCC 10.1, in Listing 6 produces somewhat similar code, although it is far less eager at unrolling the main loop. Instead, for each row, GCC produces a single loop that process eight elements at a time, followed by checks for remaining four, two, and one entry segments and any necessary horizontal adds.

This is where compilers can be both friend and enemy, depending on the number of columns, n , of A we actually have. For this case, Clang processes $n = 64x + 32z + 1y$, while GCC processes this as $n = 8x + 4z + 2y + 1$, where question marks denote unrolled blocks of code that exist outside of the loops that process 64, 1, or 8 entries noted by x , y , and z . If we have 31 entries, we will perform 31 single-entry iterations with Clang, but three 8-entry iterations, then four, then two, then one with GCC. While this is not a sparse matrix, we still see the same possibility for poorly-predicted or seemingly unpredictable loop counts. Further, we see that in Clang’s 64-element case, the amount of work lost if we incorrectly predict to continue speculative execution in the unrolled loop is large. However, with dense matrices, it is more probable that always predicting to continue the loop is the correct prediction. This sort of intangible quantifier is why

```

1  .LBB0_7:
2      vmovdqu ymm4, ymmword ptr [rsi + 4*rax]
3      vmovdqu ymm5, ymmword ptr [rsi + 4*rax + 32]
4      vmovdqu ymm6, ymmword ptr [rsi + 4*rax + 64]
5      vpmulld ymm4, ymm4, ymmword ptr [rbx + 4*rax - 224]
6      vmovdqu ymm7, ymmword ptr [rsi + 4*rax + 96]
7      vpaddd  ymm0, ymm0, ymm4
8      vpmulld ymm4, ymm5, ymmword ptr [rbx + 4*rax - 192]
9      vpaddd  ymm1, ymm1, ymm4
10     vpmulld ymm4, ymm6, ymmword ptr [rbx + 4*rax - 160]
11     vpmulld ymm5, ymm7, ymmword ptr [rbx + 4*rax - 128]
12     vpaddd  ymm2, ymm2, ymm4
13     vpaddd  ymm3, ymm3, ymm5
14     vmovdqu ymm4, ymmword ptr [rsi + 4*rax + 128]
15     vmovdqu ymm5, ymmword ptr [rsi + 4*rax + 160]
16     vmovdqu ymm6, ymmword ptr [rsi + 4*rax + 192]
17     vmovdqu ymm7, ymmword ptr [rsi + 4*rax + 224]
18     vpmulld ymm4, ymm4, ymmword ptr [rbx + 4*rax - 96]
19     vpmulld ymm5, ymm5, ymmword ptr [rbx + 4*rax - 64]
20     vpaddd  ymm0, ymm0, ymm4
21     vpaddd  ymm1, ymm1, ymm5
22     vpmulld ymm4, ymm6, ymmword ptr [rbx + 4*rax - 32]
23     vpaddd  ymm2, ymm2, ymm4
24     vpmulld ymm4, ymm7, ymmword ptr [rbx + 4*rax]
25     vpaddd  ymm3, ymm3, ymm4
26     add    rax, 64
27     add    rbp, -2
28     jne    .LBB0_7

```

Listing 4: Clang 10.0.1 output of unrolled loop from Listing 3.

```

1  .LBB0_11:
2      mov    r9d, dword ptr [rsi + 4*rbp]
3      imul  r9d, dword ptr [r13 + 4*rbp]
4      add    eax, r9d
5      inc   rbp
6      cmp   r8, rbp
7      jne   .LBB0_11

```

Listing 5: Clang 10.0.1 output of single-entry loop from Listing 3.

```

1  .L5:
2      vmovdqu ymm2, YMMWORD PTR [r10+rax]
3      vpmulld ymm0, ymm2, YMMWORD PTR [rdi+rax]
4      add    rax, 32
5      vpaddd ymm1, ymm1, ymm0
6      cmp    rax, r14
7      jne    .L5

```

Listing 6: GCC 10.1 output of main loop from Listing 3.

```

1  vpaddd ymm1, ymm1, ymm3
2  vpaddd ymm0, ymm0, ymm2
3  vpaddd ymm0, ymm0, ymm1
4  vextracti128    xmm1, ymm0, 1
5  vpaddd xmm0, xmm0, xmm1
6  vpshufd xmm1, xmm0, 78      # xmm1 = xmm0[2,3,0,1]
7  vpaddd xmm0, xmm0, xmm1
8  vpshufd xmm1, xmm0, 229    # xmm1 = xmm0[1,1,2,3]
9  vpaddd xmm0, xmm0, xmm1
10 vmovd  eax, xmm0

```

Listing 7: Clang 10.0.1 horizontal adds from 32 32-bit entries to a single 32-bit entry occurring after the loop in Listing 4.

we are performing this analysis on the aspects of the problem that can be quantified in a data-agnostic fashion.

To that end, we repeat this process for our selection of data types and panel sizes, performing this same analysis using `llvm-mca` and record our results in Table 3.1. The two particularly important values in this table are the reciprocal throughput and the cycles-per-operation count. Both of these values are listed in number of cycles, again, assuming perfect loads from L1 cache for all required loads. Particularly for 32-bit integer, float, and double vectors, Clang produces code that performs more than one multiplication and addition per cycle. In fact, our bottleneck in these cases stems from performing eight vector loads from the row of A and column of B which combine to a reciprocal throughput of 8 cycles, as we can launch two vector loads per-cycle⁶. However, when we extend our problem to the panel case, Clang (and especially GCC in the 8xfloat case) are unhappy with this change in pattern. While the panel case seems as if it should be a sequence of row operations with sizes that are amenable to vectorization (all 256 bits), something about the loop ordering or access pattern causes the compilers to emit strangely inefficient code for these cases. Curiously, the case with a 4xdouble panel is the “best” GCC performs, at least from a cycles-per-operation perspective.

However, cycles-per-entry is not the only factor to consider. The reciprocal throughput is also an important number. This is, in effect, the number of cycles into our loop that must pass before we can begin executing another cycle, based on data dependencies and available resources. Thus, if the reciprocal throughput is shorter than our processor’s instruction pipeline and the process correctly predicts that we wish to perform another iteration, it may be able to begin another one (again, provided resources are available). Thus, Clang’s 64-element 1x32-bit integer loop completes in approximately 28.8 cycles, while GCC’s code can theoretically execute eight 8-element cycles in $7 \times 1.3 + 8 \times 2.63 = 31.44$ cycles, so if we exist in a world of perfect predictions

⁶ Agner Fog, Instruction Tables

with these small numbers of instructions and large pipelines, the real-world difference is likely to be smaller than shown. However, this situation becomes more important in the case of sparse matrices, where we (the processor) are less likely to correctly predict whether or not to execute a loop again. In that case, executing GCC's loop eight times is seven opportunities to get a branch prediction incorrect and lose any work speculatively done. However, this also affects us with Clang, as if we do not have 64 elements, we are not using this loop and likely jumping into a single-entry version with the same problem.

As a final couple notes regarding this dense multiplication example, as these issue occur frequently enough. There *is* a difference between the vector and panel operations when it comes to SIMD operations. In the case of a vectorized dot product, we can directly multiply an 8-entry register against an 8-entry register. This result then needs to be horizontally added a couple times to produce the single-entry result. In the panel-case, we are performing an axpy operation, scaling the row of B added to a row of our panel by one entry of the matrix A. If we wanted to perform this multiplication with our vector registers, we would need to load and broadcast our single value from A into all segments of a vector register which is a more expensive operation than simply loading a 256-bit chunk of memory into a register. Second, not all instructions are equally supported or equally fast. Notably, there is no desired 4x64 by 4x64 bit integer multiplication available (that is, $c_i = a_i \times b_i$, where we truncate results to keep all operands the same size). This exists for 32-bit integers, doubles, and floats, but not for 64-bit integers. We can only multiply two of the four integers in a 256-bit register and store two 128-bit results, of which we ignore the upper 64 bits of both results. Thus, a sequence of multiplies, shifts, and adds are required to correctly obtain the pairwise product of two vector registers of four 64-bit integers.

Using Agner Fog's instruction tables [18] and an x86 assembly guide⁷ as reference, there are a couple observations that can be made regarding "speed" of some of

⁷ <https://www.felixcloutier.com/x86/index.html>

Operation	Cycles	Reciprocal throughput
Multiplying 32/64-bit ints	3	1
Adding 32/64-bit ints	1	0.25
Multiplying eight 32-bit ints	10	1
Adding eight 32-bit ints	1	0.33
Adding four 64-bit ints	1	0.33
Multiplying lower halves of four 64-bit ints	5	0.5
Adding floats/doubles	4	0.5
Multiplying floats/doubles	4	0.5
Fused multiply-add of floats/doubles	4	0.5
Adding eight floats	4	0.5
Multiplying eight floats	4	0.5
Fused multiply-add of eight floats	4	0.5
Adding four doubles	4	0.5
Multiplying four doubles	4	0.5
Fused multiply-add of four doubles	4	0.5

Table 3.2: Cost and reciprocal throughputs (cycles until another operation can be launched that requires the same execution units) for common operations in this dissertation on the Skylake architecture, based on values from Agner Fog’s instruction tables [18].

these operations. We note these costs in Table 3.2. Of particular note, all floating-point operations execute in the same registers and all common operations require the same amount of time. Integer additions are quite fast, but multiplication is expensive (and we are not considering division at all in this work, which is in a league of its own). From these numbers, we can infer a couple results. First, we will generally expect floating point arithmetic to be slightly faster. Second, we expect 64-bit integer arithmetic to be slower. Third, we expect the addition-based Method of the Four Russians to be faster in the integer case.

This initial table is indicative of some of the oddities we will see throughout the remainder of our analyses. First, Clang tends to try to unroll loops to process 32 or 64 operations (256 bits of each operand). If GCC unrolls loops, it opts to do so just two or four times. Second, row operations (panel operations), at first glance, should not disrupt optimizations or patterns the compilers are looking for, despite

the apparent poor performance we saw with dense multiplication. Finally, sometimes certain parameters or orderings just yield abysmal code, as in the case of GCC and the 8xfloat panel case in Table 3.1.

In the spirit of the suite of comparisons we will be performing in this chapter, we will repeat our process of compilations, assembly-disassembly, and llvm-mca analysis after swapping the order of our inner two loops (looping over the rows and columns of our destination). For dense matrix-matrix multiplication, this “optimization” is frequently cited for its efficiency because of better memory access patterns. For the vector case, this should not change much as the vector width is known at compile time. The question is what changes in assembly code do we see in the panel case. For reference, the C++ code for this loop transposition is in Listing 8 and our tabular compilation of our analysis process is in Table 3.3.

```

1  #include <cstdint>
2  using DT = std::uint32_t;
3  using M = DT * __restrict;
4  using T = DT * __restrict;
5  #define P 1
6
7  void mul_ikj(M A, T B, T C, std::size_t m, std::size_t n) {
8      for (auto i = 0; i < m; ++i) {
9          for (auto k = 0; k < P; ++k) {
10             for (auto j = 0; j < n; ++j) {
11                 C[i*P + k] += A[i*n + j] * B[j*P + k];
12             }
13         }
14     }
15 }

```

Listing 8: C++ implementation of dense matrix-vector multiplication. DT and P control our vector type and width.

For easy comparison, we repeat our initial “i-j-k” ordering table after our new “i-k-j” table. What we find, in this case, is that even with the vector width hard-coded and available to the compiler to use in any optimizations it wishes, we do not even see the same code produced in the vector case for all input combinations. With GCC, in the 1xdouble case, there is a slight change of instructions in the initial computation of each row, while in the 1x64-bit integer case, GCC unrolls the loop an additional four times, yielding a loop comparable to its 1x32-bit case, albeit it with lower reciprocal throughput due to requiring access to twice as much memory. The panel cases, on the other hand, have changed substantially. While the reciprocal throughputs of the panel cases have, as a whole, halved, this is sometimes due to the loops being unrolled fewer times. The cycles-per-entry has risen substantially in every case. Part of this increase

Vector width	Clang 10.0.0				GCC 10.1			
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
i-k-j ordering								
1x32-bit int.	64	8	0.45	15	8	1.3	2.63	15
8x32-bit int.	32	20.5	3.69	14	8	35	8.13	15
1x64-bit int.	32	24	1.38	13	8	3	2.5	13
4x64-bit int.	16	14.5	6.13	14	1	1.3	12	
1xfloat	64	8	0.33	28	8	1	1.75	24
8xfloat	32	21.5	5.31	24	8	35	7.63	24
1xdouble	32	8	0.66	23	4	1	3.5	19
4xdouble	16	12	6.31	27	4	6	7	19
i-j-k ordering								
1x32-bit int.	64	8	0.45	15	8	1.3	2.63	15
8x32-bit int.	64	42.5	2.83	42/19	64	72	1.44	33
1x64-bit int.	32	24	1.38	13	4	3	5	13
4x64-bit int.	32	25.5	4.06	15	4	4	3.5	13
1xfloat	64	8	0.33	28	8	1	1.75	24
8xfloat	64	43	2.38	46/17	64	120	2.27	47
1xdouble	32	8	0.66	23	4	1	3.5	19/4
4xdouble	32	20	2.53	22	32	16	0.97	22

Table 3.3: llvm-mca analysis for classical dense multiplication in the i-k-j ordering shown in Listing 8. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.

is due to loops being unrolled fewer times, too. In the 4x64-bit panel case, GCC does not even process the entire four-element panel in a single iteration.

As this particular example is considering a dense case, this variant ordering is seemingly less effective in most cases where there are differences. However, the fact that there are significant differences is in itself a result moving forward with our analyses of sparse formats. Given that loop unrolling, reordering, and data access pattern considerations are all considerations that compilers routinely do, seeing a significant difference when changing the order of the two loops when one has a fixed length suggests that there are sensitivities beyond data access patterns when working with matrix-vector products. We will address this problem again in Section 3.5 with a sparse format, where the picture becomes even more muddled.

As a whole, this first section serves as an introduction and gentle exposition to our analysis workflow and aspects of assembly code we consider. As we reach the end of the chapter, we will refer back to this section as a comparison point when considering the arithmetic efficiency of our sparse formats.

3.3 Sparse Format Analysis

In this section, we continue our process of compilation and assembly analysis using `llvm-mca` presented in Section 3.2. Here, we will consider most of the standard formats mentioned in Section 1.1, namely coordinate format (COO), compressed sparse row (CSR), and compressed sparse column (CSC). As we will see, even with the craziness that occurs in some dense cases, we will never reach 0.33 cycles-per-operation we saw with vectors of floats. However, we will see some significant differences between these formats.

```

1  #include <cstdint>
2  using DT = std::uint32_t;
3  using M = DT * __restrict;
4  using I = std::size_t * __restrict;
5  using T = DT * __restrict;
6  #define P 1
7
8  void mul_coo_vecs(M A, I IA, I JA, T B, T C, std::size_t nnz) {
9      for (auto i = 0; i < nnz; ++i) {
10         auto v = A[i];
11         auto r = IA[i];
12         auto c = JA[i];
13         for (auto j = 0; j < P; ++j) {
14             C[r*P + j] += v * B[c*P + j];
15         }
16     }
17 }

```

Listing 9: C++ implementation of sparse matrix-vector multiplication using a coordinate format with three vectors. DT and P control our vector type and width.

We will begin with coordinate format. Namely, coordinate format using three vectors: A, IA, and JA. Respectively, these store the coefficient, row index, and column index of the non-zero entries of A. We illustrate this in Listing 9 and immediately jump into the results of our analysis in Table 3.4.

In many respects, this is actually the simplest code possible, as there is only one loop. With no guarantee of contiguous reads from our right-hand panel vectors or writes to our destination, the only unrolling Clang is able to do in the vector case is to explicitly perform the given number of single-entry operations. There are some savings here, of course, chiefly that we have fewer opportunities for mispredicting a branch if

Vector width	Clang 10.0.0			GCC 10.1				
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1x32-bit int.	4	10	6.75		1	2.5	23	
8x32-bit int.	16	5	1.88		8	9.5	4	
1x64-bit int.	4	10	6.75		1	2.5	23	
4x64-bit int.	8	6.5	4.5		4	5.5	7	
1xfloat	4	10	5.75		1	2.5	19	
8xfloat	16	5	1.44		8	10.2	3.5	
1xdouble	4	10	5.75		1	2.5	19	
4xdouble	8	5	2.88		4	6.2	6	

Table 3.4: llvm-mca analysis for sparse matrix-vector multiplication using coordinate format with three vectors shown in Listing 9. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.

we have fewer iterations. This is especially true, as without knowledge of the number of rows or columns, there is only one loop in the compiler output, not just a main loop within an outer loop (in the dense analysis, for example, our main loop iterated over the columns of a row of the matrix A). As we see in Listing 10 and Listing 11, when considering the vector case (here, a vector of 32-bit integers), neither compiler produces particularly exciting code. Clang’s is simply a repetition of GCC’s code four times, thus with marginal savings in the number of iterations and increments performed. As we see, again, we cannot necessarily compare the cycles-per-operation number without also considering the reciprocal throughput. As we saw with our operation costs in Table 3.2, we see floating point arithmetic slightly out-performing integer arithmetic. This is primarily a dependency issue, as the fused multiply-add can begin its four-cycle process immediately, while the integer procedure must wait for the multiplication to finish before performing the addition. This does not necessarily affect our reciprocal throughput, as this is gated by the loads from memory into registers.

Compared to our dense cases, we are handling fewer entries and handling them

more slowly. As we see in the 32-bit integer vector case in Listing 11, our assembly instructions directly mirror our C++ code, with repeated triple loads from our three coordinate vectors, a multiply (and load), add (and store), and comparison. Compared to the dense multiplication Clang produced in its single-element loop in Listing 5, we see that we are performing three additional loads and a store. In the dense case, the store is postponed until the end of the row. In coordinate format, the store occurs after each operation.

However, there is some potentially good news from this sparse format. Instead, we consider the code Clang produces for its eight 32-bit integer panel case in Listing 12. Here, we see Clang emitting code using 256-bit vector instructions to perform row operations on operands of eight 32-bit integers. For each operation, we broadcast our coefficient into all eight 32-bit segments of a 256-bit register, then perform our multiplication and addition. As we see in Table 3.4, we achieve 1.88 cycles-per-operation with a reciprocal throughput of 5 cycles without requiring any horizontal adds. If we compare this to our dense multiplication of the same panel, the best achieved by clang was 2.83 cycles-per-operation with a staggering reciprocal throughput of 42.5 cycles (for 64 entries). Notably, GCC does achieve a better cycle-per-operation count, albeit it a reciprocal throughput of 72 cycles for 64 entries. If our sparse case is able to directly execute four times without any contention or mispredictions, we would expect it to take $5 \times 3 + 1.88 \times 16 \approx 45$ cycles, or around 0.7 cycles-per-operation. With only one loop (compared to one per row), a misprediction should be unlikely. However, astonishingly good performance is more likely a user error in not formatting loops correctly, or a compiler error in not optimizing the dense case correctly, not some inherent advantage in the sparse coordinate format, as our vector cases are much slower with coordinate format than with the dense multiplication.

Notably, GCC does not emit code that uses vector operations for this coordinate format, even for the panel cases.

For comparison, we redo this procedure using a coordinate format of a vector of tuples to see if this format has any impact on our performance. This slight modification

```

1  .LBB0_8:
2      mov     r11, qword ptr [rsi + 8*rax]
3      mov     rbx, qword ptr [rdx + 8*rax]
4      mov     ebx, dword ptr [rcx + 4*rbx]
5      imul   ebx, dword ptr [rdi + 4*rax]
6      add     dword ptr [r8 + 4*r11], ebx
7      mov     rbx, qword ptr [rdx + 8*rax + 8]
8      mov     r11d, dword ptr [rcx + 4*rbx]
9      imul   r11d, dword ptr [rdi + 4*rax + 4]
10     mov     rbx, qword ptr [rsi + 8*rax + 8]
11     add     dword ptr [r8 + 4*rbx], r11d
12     mov     r11, qword ptr [rsi + 8*rax + 16]
13     mov     rbx, qword ptr [rdx + 8*rax + 16]
14     mov     ebx, dword ptr [rcx + 4*rbx]
15     imul   ebx, dword ptr [rdi + 4*rax + 8]
16     add     dword ptr [r8 + 4*r11], ebx
17     mov     rbx, qword ptr [rdx + 8*rax + 24]
18     mov     r11d, dword ptr [rcx + 4*rbx]
19     imul   r11d, dword ptr [rdi + 4*rax + 12]
20     mov     rbx, qword ptr [rsi + 8*rax + 24]
21     add     dword ptr [r8 + 4*rbx], r11d
22     add     rax, 4
23     cmp     r9, rax
24     jne     .LBB0_8

```

Listing 10: Clang 10.0.1 output of unrolled loop from Listing 9 with a vector of 32-bit integers.

```

1  .L3:
2      mov     r10, QWORD PTR [rdx+rax*8]
3      mov     r11, QWORD PTR [rsi+rax*8]
4      mov     r10d, DWORD PTR [rcx+r10*4]
5      imul   r10d, DWORD PTR [rdi+rax*4]
6      inc     rax
7      add     DWORD PTR [r8+r11*4], r10d
8      cmp     r9, rax
9      jne     .L3

```

Listing 11: GCC 10.1 output of loop from Listing 9 with a vector of 32-bit integers.

```

1  .LBB0_7:
2      mov     r11, qword ptr [rsi + 8*rax]
3      mov     rbx, qword ptr [rdx + 8*rax]
4      shl     rbx, 5
5      shl     r11, 5
6      vpbroadcastd  ymm0, dword ptr [rdi + 4*rax]
7      vpmulld ymm0, ymm0, ymmword ptr [rcx + rbx]
8      vpadd  ymm0, ymm0, ymmword ptr [r8 + r11]
9      vmovdqu ymmword ptr [r8 + r11], ymm0
10     mov     rbx, qword ptr [rsi + 8*rax + 8]
11     mov     r11, qword ptr [rdx + 8*rax + 8]
12     shl     r11, 5
13     shl     rbx, 5
14     vpbroadcastd  ymm0, dword ptr [rdi + 4*rax + 4]
15     vpmulld ymm0, ymm0, ymmword ptr [rcx + r11]
16     vpadd  ymm0, ymm0, ymmword ptr [r8 + rbx]
17     vmovdqu ymmword ptr [r8 + rbx], ymm0
18     add     rax, 2
19     cmp     r9, rax
20     jne     .LBB0_7

```

Listing 12: Clang 10.0.1 output of unrolled loop from Listing 9 with a panel of eight 32-bit integers.

is noted in Listing 13, with its corresponding llvm-mca analysis table in Table 3.5. For reference, we follow this table with the previous format’s table.

```
1  #include <stdint>
2  #include <tuple>
3  using DT = std::uint32_t;
4  using M = std::tuple<DT, std::size_t, std::size_t> *__restrict;
5  using T = DT *__restrict;
6  #define P 1
7
8  void mul_coo_tup(M A, T B, T C, std::size_t nnz) {
9      for (auto i = 0; i < nnz; ++i) {
10         auto v = std::get<0>(A[i]);
11         auto r = std::get<1>(A[i]);
12         auto c = std::get<2>(A[i]);
13         for (auto j = 0; j < P; ++j) {
14             C[r*P + j] += v * B[c*P + j];
15         }
16     }
17 }
```

Listing 13: C++ implementation of sparse matrix-vector multiplication using a coordinate format with a vector of tuples. DT and P control our vector type and width.

Unsurprisingly, there are no significant differences. If anything, the vector of tuples format is ever-so-slightly worse, likely due to more awkward loads and alignments, packing three values together. With Clang, we see our loop unrolled two fewer times in the vector cases. While this superficially impacts our cycles-per-operation, by halving the reciprocal throughput, there is little likelihood of real-world impact. Potentially, one might think there could be some possibility of using vector loads and shuffling

Vector width	Clang 10.0.0			GCC 10.1				
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
Vector of tuples								
1x32-bit int.	2	5	12		1	2.5	23	
8x32-bit int.	16	5	1.88		8	9.5	4	
1x64-bit int.	2	5	12		1	2.5	23	
4x64-bit int.	8	6.7	4.5		4	5.5	7	
1xfloat	2	5	10		1	2.5	19	
8xfloat	16	5	1.44		8	10.2	3.5	
1xdouble	2	5	10		1	2.5	19	
4xdouble	8	5	2.88		4	6.2	6	
Three vectors								
1x32-bit int.	4	10	6.75		1	2.5	23	
8x32-bit int.	16	5	1.88		8	9.5	4	
1x64-bit int.	4	10	6.75		1	2.5	23	
4x64-bit int.	8	6.5	4.5		4	5.5	7	
1xfloat	4	10	5.75		1	2.5	19	
8xfloat	16	5	1.44		8	10.2	3.5	
1xdouble	4	10	5.75		1	2.5	19	
4xdouble	8	5	2.88		4	6.2	6	

Table 3.5: llvm-mca analysis for sparse matrix-vector multiplication using coordinate format with a vector of tuples shown in Listing 13 compared to a coordinate format with three vectors shown in Listing 9.

Vector width	Clang 10.0.0				GCC 10.1			
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1x32-bit int.	32	24	4.88	11	8	6	6.13	15
8x32-bit int.	32	22.5	4.94	36	8	10	4	0/16
1x64-bit int.	32	28	4.03	13	4	3.7	10	13
4x64-bit int.	32	26	4.69	15	4	4	5	13/4
1xfloat	32	24	5.62	21	8	6	5.25	24
8xfloat	32	22.5	1.78	33	8	9	3.38	
1xdouble	32	24	3.22	23	4	3.5	8.75	19
4xdouble	32	19.5	2.03	21	4	3	4.75	9

Table 3.6: llvm-mca analysis for sparse matrix-vector multiplication using the compressed sparse row format shown in Listing 14. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.

operations to infinitesimally speed up an unrolled loop, but having three values (of potentially different sizes and types) likely makes this a non-starter. Again, this analysis considers a perfect world where all loads are successfully from L1 cache. There may be practical differences between these formats when considering how to block them, especially if there are alignment concerns with different-sized items within a tuple. However, while coordinate format is often viewed as the least-efficient sparse format due to its memory requirements, we see that it may compare favorably to a naïve dense multiplication implementation for panel multiplication, at least with Clang.

We follow our foray into coordinate format by considering the most popular sparse matrix format: compressed sparse row. As before, our reference implementation is given in Listing 14 and the results of our analysis in Table 3.6.

```

1  #include <stdint>
2  using DT = std::uint32_t;
3  using M = DT *__restrict;
4  using I = std::size_t *__restrict;
5  using T = DT *__restrict;
6  #define P 1
7
8  void mul_csr(M A, I IA, I JA, T B, T C, std::size_t m) {
9      for (auto i = 0; i < m; ++i) {
10         for (auto j = IA[i]; j < IA[i+1]; ++j) {
11             auto v = A[j];
12             auto r = i;
13             auto c = JA[j];
14             for (auto j = 0; j < P; ++j) {
15                 C[r*P + j] += v * B[c*P + j];
16             }
17         }
18     }
19 }

```

Listing 14: C++ implementation of sparse matrix-vector multiplication using a compressed sparse row format. DT and P control our vector type and width.

The analysis of the assembly produced for this CSR-based multiplication is somewhat puzzling. For Clang, performance seems rather uniform in every case except the 8xfloat and 4xdouble panel cases (we have half the cycles-per-operation in the 4x64-bit int case because we handle half the operations due the elements requiring twice as many bits as 32-bit integers). Despite Clang’s (and GCC’s, to a lesser extend) loops being unrolled, our reciprocal throughput per operation is fairly high, even though we are achieving approximately triple the cycles-per-entry performance of coordinate

format, in the vector case. The panel cases with CSR do not appear to be significantly different from its vector cases, except for Clang with the panel of eight floats. This loop is too large to print on a single page, but we will briefly refer to the standard (not unrolled) loop version Clang also produces for this case, shown in Listing 15. As we saw with coordinate format, the compiler can produce what appears to be optimal code, but things become complicated in the unrolled loop due to a large number of shuffles and shifts, likely due to some sort of unaligned accesses going on, possibly due to different data type sizes when comparing our indices to our coefficients. The fact that the vector of doubles executes faster than the vector of floats definitely lends some support to the idea that there is some poor interaction between 64-bit indices and 32-bit values. Further, we have a clean-up step at the end of each row to horizontally sum our partial results.

Compared to coordinate format, the unrolled loops in CSR likely will outperform in the vector cases, but not for panel multiplication. While the lower memory overhead of CSR and more uniform access patterns may make it easier to block into cache-friendly submatrices, this analysis suggests that it, at best, may marginally outperform coordinate format for some cases with Clang, but not panel cases. With GCC, however, we see some potential for modest speed-ups, but these results are almost uniformly slower than Clang’s, even accounting for differences in operation count and reciprocal throughput. However, even fractions of a cycle per entry may yield noticeable savings for a large enough matrix, especially if the lower memory footprint of CSR is critical. However, we have no particular expectations for much better performance, as we still face the same potential problems as coordinate format: namely, our reads from our right-hand operand (B , in $A \times B = C$) are still potentially non-adjacent.

Moving on from CSR, we consider our final “standard” format: compressed sparse column, CSC. As before, we provide our reference implementation in Listing 16, with the completed llvm-mca analysis in Table 3.7.

```

1 .LBB0_9:
2     mov     rax, qword ptr [rdx + 8*rbx]
3     shl     rax, 5
4     vbroadcastss    ymm0, dword ptr [rdi + 4*rbx]
5     vfmadd231ps    ymm1, ymm0, ymmword ptr [rcx + rax]
6     inc     rbx
7     cmp     rbx, r12

```

Listing 15: Clang 10.0.0 output of its standard loop from Listing 14 with a panel of eight floats.

Vector width	Clang 10.0.0			GCC 10.1				
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1x32-bit int.	4	6	5.25		1	1.5	18	
8x32-bit int.	32	6.5	0.81		8	8	3.25	
1x64-bit int.	4	6	5.25		1	1.5	18	
4x64-bit int.	16	10	2.06		4	4.7	5.5	
1xfloat	4	6	5.25		1	1.7	19	
8xfloat	32	6	0.75		8	9.3	3.5	
1xdouble	4	6	5.25		1	1.7	19	
4xdouble	16	6	1.5		4	5.3	6	

Table 3.7: llvm-mca analysis for sparse matrix-vector multiplication using the compressed sparse column format shown in Listing 16. The first column for each compiler notes the number of element axpys performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.

```

1  #include <cstdint>
2  using DT = std::uint32_t;
3  using M = DT * __restrict;
4  using I = std::size_t * __restrict;
5  using T = DT * __restrict;
6  #define P 1
7
8  void mul_csc(M A, I IA, I JA, T B, T C, std::size_t n) {
9      for (auto i = 0; i < n; ++i) {
10         for (auto j = JA[i]; j < JA[i+1]; ++j) {
11             auto v = A[j];
12             auto r = IA[i];
13             auto c = i;
14             for (auto j = 0; j < P; ++j) {
15                 C[r*P + j] += v * B[c*P + j];
16             }
17         }
18     }
19 }

```

Listing 16: C++ implementation of sparse matrix-vector multiplication using a compressed sparse column format. DT and P control our vector type and width.

Compared to CSR there is nothing surprising here for Clang or GCC. Interestingly, we see GCC does not unroll any loops in the vector cases, while Clang unrolls every loop with this format four times. Within each assembly output, all versions perform loads, then a fused multiply-add for floats and multiplication then addition for integers, then a store. For Clang and the panel cases, these are all vectorized versions. For Clang’s vector versions and all of GCC’s versions, these are simply single-element operations (repeated the requisite number of times).

As with coordinate format, CSC directly yields assembly code that is very close to the original C++ code. In circumstances where we do not expect many entries per column, these less-aggressively unrolled loops may be more advantageous than Clang’s aggressive unrolling present in CSR. However, the performance of CSC and Clang for the vector cases is, on a cycle-per-entry and reciprocal throughput basis potentially lower than CSR. However, our panel cases for both GCC and Clang compile to the expected vectorized broadcast, multiply, and add without the strange shifting and shuffling Clang saw with CSR in its unrolled panel versions. GCC still does not produce any vectorized code, yielding slightly better performance than coordinate format.

Ultimately, these three standard sparse formats may have their own pros and cons. The strong advantage, from an arithmetic perspective, for coordinate format is that we have a single loop, while we have nested loops for CSR and CSC (per row and per column). With both coordinate format and CSC, we obtain main loops for both GCC and Clang that closely match our original C++ code. Clang is capable of emitting vectorized code for the panel case with both formats, while GCC does not. In general, our lower metadata requirements yield lower reciprocal throughput for CSC compared to coordinate format, suggesting that memory has arithmetic impact, not just memory impact. Finally, with CSR we saw that both compilers can attempt more aggressive optimizations. However, with these attempted optimizations came some unexpected inefficiencies that may require many multiples of the unrolled number of elements per row to justify the extra cost. Curiously, we saw that both COO and CSC formats with Clang can allegedly handle our panel cases more efficiently than our reference dense matrix-panel multiplication code, even though the corresponding vector cases perform significantly worse. At this point, it is unclear why both compilers struggle to produce seemingly-obvious vector instructions with the fixed-width vector in the dense case. It is not unexpected that these sparse formats, at least in the vector cases, do not nearly approach the efficiency of the dense formats. However, we will see these converge slightly in our next section as we branch into block variants of these sparse formats.

3.4 Block Sparse Format Analysis

In this section, we extend the analyses of the standard sparse formats considered in Section 3.3 to block variants of these formats. While we already consider the effects of compiler, vector width, and data type, we will limit ourselves to a small number of potential block sizes for each format in order to keep our discussion and presentation tractable. However, we recognize that there are potentially certain block sizes that are well suited to certain matrices and certain block sizes that may simply perform well with our two compilers that we are not considering. Further, as these formats become more complicated, as we saw with dense matrix-vector multiplication in Section 3.2, the effects of loop ordering may also play a significant role in the efficiency of code generated by our compilers.

Where relevant, we use BR to denote the number of rows in a block and BC to denote the number of columns. For coordinate format, we consider two-dimensional blocks, while for CSC and CSR we only consider one-dimensional blocks.

With these caveats out of the way, we will begin this section by revisiting our coordinate formats and consider block variants of both our three vectors and vector of tuples format. In this circumstance, we do expect differences between the two formats, as a tuple that contains a row index, column index, and a block of data is intuitively more likely to behave differently as we increase our block dimensions compared to handling three vectors.

```

1  #include <cstdint>
2  using DT = std::uint32_t;
3  using M = DT * __restrict;
4  using I = std::size_t * __restrict;
5  using T = DT * __restrict;
6  #define P 1
7  #define BR 2
8  #define BC 2
9
10 void mul_coo_block_vec(M A, I IA, I JA, T B, T C, std::size_t nb) {
11     for (auto i = 0; i < nb; ++i) {
12         for (auto j = 0; j < BR; ++j) {
13             auto r = IA[i] + j;
14             for (auto k = 0; k < BC; ++k) {
15                 auto c = JA[i] + k;
16                 auto v = A[i*BR*BC + j*BC + k];
17                 for (auto l = 0; l < P; ++l) {
18                     C[r*P + l] += v * B[c*P + l];
19                 }
20             }
21         }
22     }
23 }

```

Listing 17: C++ implementation of sparse matrix-vector multiplication using a block coordinate format with three vectors. DT and P control our vector type and width. BR and BC note the number of rows and columns of each block. nb represents the number of blocks in the matrix.

```

1  #include <cstdint>
2  #include <tuple>
3  #include <array>
4  using DT = std::uint32_t;
5  using I = std::size_t * __restrict;
6  using T = DT * __restrict;
7  #define P 1
8  #define BR 2
9  #define BC 2
10 using M = std::tuple<std::array<DT, BR*BC>, std::size_t, std::size_t>
    ↪ * __restrict;
11
12 void mul_coo_block_tup(M A, T B, T C, std::size_t nb) {
13     for (auto i = 0; i < nb; ++i) {
14         for (auto j = 0; j < BR; ++j) {
15             auto r = std::get<1>(A[i]) + j;
16             for (auto k = 0; k < BC; ++k) {
17                 auto c = std::get<2>(A[i]) + k;
18                 auto v = std::get<0>(A[i])[j*BC + k];
19                 for (auto l = 0; l < P; ++l) {
20                     C[r*P + l] += v * B[c*P + l];
21                 }
22             }
23         }
24     }
25 }

```

Listing 18: C++ implementation of sparse matrix-vector multiplication using a block coordinate format with a vector of tuples. DT and P control our vector type and width. BR and BC note the number of rows and columns of each block. nb represents the number of blocks in the matrix.

In Listing 17 and Listing 18, respectively, we present the code for our block format using three vectors and using a vector of tuples. While there is little literature involving the block-tuple format, we consider it anyway, to see if this results in any change in what the compilers do. Rather than the storing the coefficients, row-block indices, and column-block indices separately, the tuple version alternates tuples of these three values for each block. In the 2-by-2 block case, this means we store four coefficients and two indices, repeating this for each block.

For each format, we consider 2-by-2, 4-by-4, and 8-by-8 blocks of coefficients. As we increase the block sizes, we note that Clang is typically unwilling to unroll loops to perform more than 256 bytes worth of operations (typically, 32 64-bit operations or 64 32-bit operations). In the 4-by-4 block with panels of 8 32-bit integers or 8 floats, it is willing to unroll the loop to handle all 128 operations. In the 4-by-4 8xfloat case, we surpass the peak cycles-per-operation count we saw in dense multiplication with a better reciprocal throughput-per-operations count as well (0.33 cycles-per-operation and 8 reciprocal throughput for Clang’s dense multiplication with a vector of floats).

We summarize our llvm-mca analyses in Table 3.8 and Table 3.9 for the vector case and tuple case, respectively. For the coordinate format with three vectors, we generally see better cycles-per-operation and reciprocal throughput per operation as we increase our block size with Clang. In the 8-by-8 block cases of panel multiplication, we perform slightly worse in the case of four 64-bit integers, worse in the case of eight floats, and substantially worse in the case of eight 32-bit integers. Likely due to the large size and more complicated arithmetic, GCC does not produce a simple loop for this size, while Clang spends the majority of its time shuffling data around, presumably because it is unable to recognize the data access patterns we are using. While the panel of doubles is not fully unrolled with Clang, we note that given the difference in reciprocal throughput, the expected performance of eight iterations of the 32-operation loop would still yield very good performance. Given the heavy utilization of the vector registers, however, it is unclear if all eight iterations could be successfully contiguously scheduled and pipelined without some delays.

Block size	Vector width	Clang 10.0.0				GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
2-by-2	1x32-bit int.	4	5	5.5		4	5	6.5	
	8x32-bit int.	32	7	1		32	49	2.34	
	1x64-bit int.	4	5	5.5		4	5	6.5	
	4x64-bit int.	16	16	2.38		16	19	3	
	1xfloat	4	5	5.75		4	5	6.25	
	8xfloat	32	6.5	0.84		32	34	1.97	
	1xdouble	4	4.5	6.25		4	4.2	7.5	
	4xdouble	16	5	1.56		16	17.7	2.31	
4-by-4	1x32-bit int.	16	13.5	2.31		16	8.5	2.75	
	8x32-bit int.	128	16	0.34		32	8	1.13	0/20
	1x64-bit int.	16	16.3	2.38		16	16	2.38	
	4x64-bit int.	64	42	1.08		16	16	2.06	
	1xfloat	16	12.2	2.69		16	8.5	2.31	
	8xfloat	128	13	0.31		32	6	1	
	1xdouble	16	12.2	2.69		16	8.7	2.5	
	4xdouble	64	13	0.63		64	86	1.84	
8-by-8	1x32-bit int.	64	64	1.42		64	24	0.95	
	8x32-bit int.	64	55	3.88	0/35	-	-	-	
	1x64-bit int.	64	64	1.42		8	8	3.13	
	4x64-bit int.	32	24	1.41		32	32	1.78	0/51
	1xfloat	64	41	1.11		64	24	0.94	
	8xfloat	64	24	0.86	0/81	8	8.5	2.5	13
	1xdouble	64	41	1.11		64	20	0.88	
	4xdouble	32	4.5	1.41		8	17	6.38	0/37

Table 3.8: llvm-mca analysis for sparse matrix-vector multiplication using the block coordinate format with three vectors as shown in Listing 17 with block sizes of 2-by-2, 4-by-4, and 8-by-8. Missing values in the first three columns denote that no clear main loop was found.

Block size	Vector width	Clang 10.0.0				GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
2-by-2	1x32-bit int.	4	5	5.5		4	5	6.5	
	8x32-bit int.	32	6	0.94		32	48	2.19	
	1x64-bit int.	4	5	5.5		4	5	6.5	
	4x64-bit int.	16	16	2.13		16	18.3	2.63	
	1xfloat	4	5	6		4	5	6.25	
	8xfloat	32	6	0.84		32	33	1.81	
	1xdouble	4	4.5	6		4	4.2	7.5	
	4xdouble	16	5	1.69		16	17.5	2.31	
4-by-4	1x32-bit int.	4	6	5		4	2.5	4.25	
	8x32-bit int.	64	7	0.44		32	47	2.13	
	1x64-bit int.	8	5	2.38		4	2.5	4.5	
	4x64-bit int.	32	6	0.72		16	16	2.25	
	1xfloat	8	6	3.25		4	3.3	6.25	
	8xfloat	32	7	1.06		32	33	1.78	
	1xdouble	8	5	3.13		4	2.5	6	
	4xdouble	32	6	0.91		16	16	2.25	
8-by-8	1x32-bit int.	64	46.5	2.33		64	24	0.92	
	8x32-bit int.	64	55	4.72	0/38	-	-	-	
	1x64-bit int.	64	54.5	2.92		8	8	3.13	0/17
	4x64-bit int.	32	24	1.34	0/22	32	20	1.66	0/22
	1xfloat	64	70	1.22		64	24	0.94	
	8xfloat	64	24	0.75	0/82	-	-	-	
	1xdouble	64	47	3.16		64	20	0.88	
	4xdouble	32	4.5	1.34	0/22	32	13	1.59	0/37

Table 3.9: llvm-mca analysis for sparse matrix-vector multiplication using the block coordinate format with a vector of tuples as shown in Listing 18 with block sizes of 2-by-2, 4-by-4, and 8-by-8. Missing values in the first three columns denote that no clear main loop was found.

In fact, we note the near-entirety of this 8-by-8 block times a panel of doubles in Listing 19. This assembly represents everything except an initial check if the entire loop is empty and pushing some pointers into registers. Our analysis considers the loop between lines 16 and 38, consisting of eight broadcasts for four doubles into the four 64-bit segments of a 256-bit register and eight fused multiplies and adds, covering 32 of our 256 total operations per block. This loop repeats eight times, as we see in lines 36 and 37, before we jump to the first label and load eight new rows of our panel in the first portion of this code. The cost of the eight loads in the first label is not represented in the “other” column in our table, although it is paid once per 8-by-8 block. This section is not performing any extraneous work, just loads, so we do not count it when considering “other” costs. If we consider this assembly, it is difficult to find any unnecessary work. Clang’s code performs a series of vector loads, broadcasts, fused multiply-adds, and some looping. It is hard to imagine that we could significantly improve on this. Thus, while the cycles-per-operation count is not as low as it is for the 4-by-4 block case, the very low reciprocal throughput suggests that more than one iteration could be scheduled simultaneously, if there are enough available registers.

```

1  .LBB0_2:
2      mov     r14, qword ptr [rsi + 8*r10]
3      mov     rbx, qword ptr [rdx + 8*r10]
4      shl     rbx, 5
5      vmovupd ymm0, ymmword ptr [rcx + rbx]
6      vmovupd ymm1, ymmword ptr [rcx + rbx + 32]
7      vmovupd ymm2, ymmword ptr [rcx + rbx + 64]
8      vmovupd ymm3, ymmword ptr [rcx + rbx + 96]
9      vmovupd ymm4, ymmword ptr [rcx + rbx + 128]
10     vmovupd ymm5, ymmword ptr [rcx + rbx + 160]
11     vmovupd ymm6, ymmword ptr [rcx + rbx + 192]
12     vmovupd ymm7, ymmword ptr [rcx + rbx + 224]
13     shl     r14, 5
14     add     r14, r8
15     xor     ebx, ebx
16  .LBB0_3:
17     lea     eax, [r11 + rbx]
18     and     eax, -8
19     vbroadcastsd ymm8, qword ptr [rdi + 8*rax]
20     vfmadd213pd ymm8, ymm0, ymmword ptr [r14 + 4*rbx]
21     vbroadcastsd ymm9, qword ptr [rdi + 8*rax + 8]
22     vfmadd213pd ymm9, ymm1, ymm8 # ymm9 = (ymm1 * ymm9) + ymm8
23     vbroadcastsd ymm8, qword ptr [rdi + 8*rax + 16]
24     vfmadd213pd ymm8, ymm2, ymm9 # ymm8 = (ymm2 * ymm8) + ymm9
25     vbroadcastsd ymm9, qword ptr [rdi + 8*rax + 24]
26     vfmadd213pd ymm9, ymm3, ymm8 # ymm9 = (ymm3 * ymm9) + ymm8
27     vbroadcastsd ymm8, qword ptr [rdi + 8*rax + 32]
28     vfmadd213pd ymm8, ymm4, ymm9 # ymm8 = (ymm4 * ymm8) + ymm9
29     vbroadcastsd ymm9, qword ptr [rdi + 8*rax + 40]
30     vfmadd213pd ymm9, ymm5, ymm8 # ymm9 = (ymm5 * ymm9) + ymm8
31     vbroadcastsd ymm8, qword ptr [rdi + 8*rax + 48]
32     vfmadd213pd ymm8, ymm6, ymm9 # ymm8 = (ymm6 * ymm8) + ymm9
33     vbroadcastsd ymm9, qword ptr [rdi + 8*rax + 56]
34     vfmadd213pd ymm9, ymm7, ymm8 # ymm9 = (ymm7 * ymm9) + ymm8
35     vmovupd ymmword ptr [r14 + 4*rbx], ymm9
36     add     rbx, 8
37     cmp     rbx, 64
38     jne     .LBB0_3
39     inc     r10
40     add     r11, 64
41     cmp     r10, r9
42     jne     .LBB0_2

```

Listing 19: Main body of both loops produced by Clang 10.0.0 when multiplying an 8-by-8 block by a panel of four doubles using block coordinate format of vectors in Listing 17.

Considering the rest of Table 3.8, we see that GCC’s code typically lags far behind Clang’s in terms of efficiency. However, we do see GCC finally emitting code using vector instructions. In particular, we will highlight the 4-by-4 panel case when multiplying a panel of eight floats. Clang fully unrolled this loop, while GCC processes just one eighth of it per iteration. Note that if we were able to fully pipeline all eight iterations, we would expect this to take $3 \times 6 + 1 \times 32 \approx 50$ cycles, or 0.78 cycles-per-operation. Certainly faster than the smaller block size and the standard coordinate format, but far from the efficiency of Clang’s code. In Listing 20 we see the assembly GCC produces for this main loop. We see in the four broadcast operations, that we are handling one row of our 4-by-4 block, while performing eight 128-bit multiplies and adds, covering 32 operations of the desired 128. Curiously, we see a mix of separate multiplications and additions along with fused multiply-adds. Due to some reuse of data in some registers, GCC does not emit the expected series of fused multiply-adds. Other than this oddity, we still see a format that generally follows what we would expect, with broadcasts, multiplies, and adds. However, we are only performing 128-bit operations, despite our panels being 256 bits wide. This difference is why Clang’s code for this case is twice as fast, on a cycles-per-operation basis. Clearly, GCC is happier with our block format than our standard format, but we are using a pattern it does not recognize or loops in unexpected orderings that prevent it from fully optimizing this to the same degree Clang does. As a final note, we see that GCC does fully unroll the 4-by-4 block with four doubles, despite a suspiciously-high reciprocal throughput. The generated loop is far too long to include inline, but it curiously performs its 64 operations with a combination of 16 128-bit operations and 32 64-bit operations, along with a large amount of seemingly unnecessary data movement.

```

1  .L3:
2      vbroadcastss    xmm12, DWORD PTR [rdx+4]
3      vbroadcastss    xmm0, DWORD PTR [rdx]
4      vmulps    xmm2, xmm8, xmm12
5      vmulps    xmm12, xmm7, xmm12
6      vbroadcastss    xmm1, DWORD PTR [rdx+8]
7      add    rax, 32
8      add    rdx, 16
9      vfmadd231ps    xmm2, xmm10, xmm0
10     vfmadd132ps    xmm0, xmm12, xmm9
11     vmovaps xmm13, xmm2
12     vmovaps xmm11, xmm0
13     vbroadcastss    xmm0, DWORD PTR [rdx-4]
14     vmulps    xmm2, xmm4, xmm0
15     vmulps    xmm0, xmm3, xmm0
16     vfmadd231ps    xmm2, xmm6, xmm1
17     vfmadd231ps    xmm0, xmm5, xmm1
18     vaddps    xmm2, xmm2, xmm13
19     vaddps    xmm0, xmm0, xmm11
20     vaddps    xmm1, xmm2, XMMWORD PTR [rax-32]
21     vaddps    xmm0, xmm0, XMMWORD PTR [rax-16]
22     vmovups XMMWORD PTR [rax-32], xmm1
23     vmovups XMMWORD PTR [rax-16], xmm0
24     cmp    rcx, rax
25     jne    .L3

```

Listing 20: Main body of both loops produced by GCC 10.1 when multiplying a 4-by-4 block by a panel of eight floats using block coordinate format of vectors in Listing 17.

By comparison, we move onto Table 3.9. As we saw in the unblocked version,

Clang is generally less aggressive at loop unrolling. This is likely due to the interleaved nature of coefficients and indices within the tuple, so it is difficult to access the next indices without also loading the entire next block of coefficients. However, most sets of parameters do not differ significantly between the two formats with Clang. The notable exception is with the Clang vector case with an 8-by-8 block size. The 4-by-4 block size with Clang and a panel of 64-bit integers yields code that is very similar to the float case we saw with GCC in Listing 20. Interestingly, this partially-unrolled loop seems to perform better than its fully-unrolled counterpart in the vector coordinate format cast. However, we believe that any difference between the vector and tuple case is likely caused by one of two reasons. The first is that the explicit use of the tuple and array of packed elements is handled differently by the compilers when considering possible optimizations. Second, the modified memory layout, particularly if we have 32-bit and 64-bit elements together may result in some poor memory access patterns or alignment issues.

Aside from some peculiarities in some panel cases, we see a general trend of achieving better throughput with larger block sizes. While the cycles-per-operation are still higher than in dense matrix-vector multiplication due primarily to heavier memory requirements (explicitly storing row and column indices) and less potential for loop unrolling. What we do not see, however, is consistently better performance to justify a larger block size instead of a similar number of smaller blocks. For instance, four iterations of Clang’s 32-bit integer vector of application of 2-by-2 blocks could take as low as $3 \times 5 + 4 \times 5.5 = 37$ cycles, while a single iteration of Clang’s 4-by-4 block could take approximately 35.3 cycles. When Clang finds nearly-optimal sequences of arithmetic, the only savings we can expect to realize by moving to a larger block size are due to a reduction in loop counter overhead. While a reduced number of iterations may be helpful on the whole due to branch predictions, we see that a larger block size does not necessarily yield truly faster arithmetic. Thus, it is unlikely that a situation of trying to take three 2-by-2 blocks and store them densely into a larger 4-by-4 block will enable us to take advantage of faster arithmetic to amortize away the extra added zeros

in order to use faster arithmetic. Instead, we generally only see vector arithmetic arise in panel cases and those only perform sequences of axpy operations using vectorized broadcast instructions. For matrices with suitably dense structure, a block coordinate format may present small improvements over an unblocked counterpart, but unlikely not enough benefit to account for many explicitly-stored zeros, if they must be added in the block case.

Moving on, we now consider block CSR and block CSC formats. Again, we will consider only blocks of 2, 4, and 8 columns with CSR and blocks of 2, 4, and 8 rows with CSC. Thus, we are guaranteed to be either accessing contiguous rows of our result in the CSR case or contiguous rows of our right-hand operand in the CSC case. What we will examine is which of these access patterns is more efficient for vector and panel operations.

```

1  #include <cstdint>
2  using DT = std::uint32_t;
3  using M = DT * __restrict;
4  using I = std::size_t * __restrict;
5  using T = DT * __restrict;
6  #define P 1
7  #define BC 2
8
9  void mul_csr_block(M A, I IA, I JA, T B, T C, std::size_t m) {
10     for (auto i = 0; i < m; ++i) {
11         for (auto j = IA[i]; j < IA[i+1]; ++j) {
12             auto r = i;
13             for (auto k = 0; k < BC; ++k) {
14                 auto c = JA[j] + k;
15                 auto v = A[j*BC + k];
16                 for (auto l = 0; l < P; ++l) {
17                     C[r*P + l] += v * B[c*P + l];
18                 }
19             }
20         }
21     }
22 }

```

Listing 21: C++ implementation of sparse matrix-vector multiplication using a block compressed sparse row format. DT and P control our vector type and width. BC notes the number of columns of each block. m represents the number of rows in the matrix

```

1  #include <stdint>
2  using DT = std::uint32_t;
3  using M = DT *__restrict;
4  using I = std::size_t *__restrict;
5  using T = DT *__restrict;
6  #define P 1
7  #define BR 2
8
9  void mul_csc_block(M A, I IA, I JA, T B, T C, std::size_t n) {
10     for (auto i = 0; i < n; ++i) {
11         for (auto j = JA[i]; j < JA[i+1]; ++j) {
12             auto c = i;
13             for (auto k = 0; k < BR; ++k) {
14                 auto r = IA[j] + k;
15                 auto v = A[j*BR + k];
16                 for (auto l = 0; l < P; ++l) {
17                     C[r*P + l] += v * B[c*P + l];
18                 }
19             }
20         }
21     }
22 }

```

Listing 22: C++ implementation of sparse matrix-vector multiplication using a block compressed sparse column format. DT and P control our vector type and width. BR notes the number of rows of each block. n represents the number of columns in the matrix.

As before, we reference the C++ code we will be using in Listing 21 for block

CSR and Listing 22 for block CSC, while reporting the outcomes of our llvm-mca analyses in Table 3.10 and Table 3.11, respectively, for block CSR and block CSC.

With these two formats and blocking, we begin to see some more substantial differences in the code these compilers generate. As we saw with the standard CSR format, with block CSR, whenever loops are unrolled, there is some amount of clean-up necessary to horizontally sum partial results from the dot product of part of a row of our matrix with a column of our vector or panel. The cost of this horizontal sum scales with the number of adds necessary to reduce results to our vector or panel width. The start-up costs, the second value listed in the “other” column denote the amount of work performed for each row of CSR and generally involves loading and/or broadcasting the matrix coefficients, sometimes with some amount of shuffling, especially in Clang’s unrolled loops. These clean-up and start-up costs for each row generally correspond to the cost of just a few arithmetic operations and the start-up cost is primarily loads of reusable values. Notably, with Clang, we do not see the same pattern we saw with our block coordinate formats: as our block size increases, we do not see generally lower cycles-per-operation counts. Additionally, we see very high reciprocal throughputs in both absolute counts and in terms of reciprocal throughput per operation. Particularly, with the 128-operation loops Clang performs with 1-by-4 blocks of eight 32-bit ints, eight floats, and 1-by-8 blocks of four 64-bit ints and four doubles, we see reciprocal throughputs over 100 cycles long. These loops also require several hundred cycles to complete, but there is little opportunity to pipeline these operations and with these large reciprocal throughputs and required clean-up and start-up costs for each row, mispredictions begin to look relatively expensive with this format. With Clang, none of these vector/panel types and sizes and block sizes yield code that, with this sort of analysis, would appear to perform better than what we saw with block coordinate format or even those seen with our unblocked CSR format in Table 3.6, where we nearly-uniformly saw 4-5 cycles-per-operation costs for 32 operations and reciprocal throughputs of about 24 cycles. With standard CSR, the best-performing code was with floats and doubles and the standard CSR code outperforms the blocked variants

Block size	Vector width	Clang 10.0.0				GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1-by-2	1x32-bit int.	16	17.5	4.94	10/14	2	2.5	9.5	
	8x32-bit int.	64	55	4.67	35/22	16	18	0.69	19
	1x64-bit int.	32	28	3.91	14/14	2	2.5	9.5	0/11
	4x64-bit int.	32	25.5	5.78	15/10	8	8	3	20/16
	1xfloat	32	17	4.84	17/14	2	2.5	13	0/8
	8xfloat	64	51	2.95	38/23	16	17.5	2.19	10/19
	1xdouble	16	24	7.13	27/14	2	1.8	13.5	0/10
	4xdouble	32	20	2.16	22/18	8	6	3.38	9/15
1-by-4	1x32-bit int.	32	39	6.44	10/14	4	2.5	7.75	0/10
	8x32-bit int.	128	108.5	4.68	34/22	32	35	1.88	11/17
	1x64-bit int.	32	40.5	5.19	12/14	4	4.5	5.75	0/11
	4x64-bit int.	64	55	2.88	17/18	16	16	2.44	22/17
	1xfloat	32	37.5	5.19	17/14	4	2.2	8	0/10
	8xfloat	128	105.5	4.88	38/23	32	34.5	1.63	10/16
	1xdouble	32	38	7.72	19/14	4	2.2	8.75	0/10
	4xdouble	64	53	2.91	22/18	16	12	2.06	9/11
1-by-8	1x32-bit int.	64	83	6.2	10/14	8	3	4.5	0/10
	8x32-bit int.	64	54	3.33	15/16	-	-	-	
	1x64-bit int.	64	83	6.86	12/14	8	8.5	3.38	0/10
	4x64-bit int.	128	110.7	2.99	17/18	32	32	1.91	23/20
	1xfloat	64	79.5	5.52	17/14	8	3	5	0/10
	8xfloat	64	84	1.97	12/15	-	-	-	
	1xdouble	64	79.5	6.78	19/14	8	2.7	4.88	0/11
	4xdouble	128	111.5	3.23	29/18	32	32	1.88	9/11

Table 3.10: llvm-mca analysis for sparse matrix-vector multiplication using the block compressed sparse row format as shown in Listing 21 with block sizes of 1-by-2, 1-by-4, and 1-by-8. Missing values in the first three columns denote that no clear main loop was found.

Block size	Vector width	Clang 10.0.0				GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
2-by-1	1x32-bit int.	4	5	5.25	0/21	2	2.5	9.5	0/10
	8x32-bit int.	32	5.8	0.81	0/28	16	16	2.13	10/23
	1x64-bit int.	4	5	5.25	0/23	2	2.7	9.5	0/11
	4x64-bit int.	16	10	2.06	0/33	8	8.2	3.25	0/15
	1xfloat	4	5	5.25	0/21	2	2.8	10.5	0/10
	8xfloat	32	5	0.75	0/27	16	17.3	2.25	0/14
	1xdouble	8	6	2.75	0/22	2	1.8	10	0/10
	4xdouble	16	5	1.5	0/24	8	9.5	3.5	0/12
4-by-1	1x32-bit int.	16	6	1.56	0/24	4	2.2	5.25	0/10
	8x32-bit int.	32	15	1.09	0/20	32	32	1.56	0/20
	1x64-bit int.	16	10	2.06	0/30	4	3	6.75	0/12
	4x64-bit int.	16	13.5	2.75	0/22	16	16	2.13	0/15
	1xfloat	16	6	1.38	0/22	4	1.8	5	0/10
	8xfloat	32	7	0.75	0/16	32	33.8	1.63	0/17
	1xdouble	16	6	1.44	0/23	4	1.8	5.25	0/11
	4xdouble	16	4.5	1.5	0/12	16	17.8	2.25	0/11
8-by-1	1x32-bit int.	32	6	0.81	0/25	8	2.2	2.75	0/11
	8x32-bit int.	64	10.3	0.48	0/12	-	-	-	
	1x64-bit int.	16	10	2.06	0/33	8	5	3.63	0/12
	4x64-bit int.	32	23.7	1.88	0/26	32	32	1.56	0/15
	1xfloat	32	6	0.72	0/23	8	1.8	2.63	0/11
	8xfloat	64	8.5	0.44	0/12	-	-	-	
	1xdouble	16	5	1.44	0/24	8	2.7	2.75	0/11
	4xdouble	32	8.5	0.88	0/19	-	-	-	

Table 3.11: llvm-mca analysis for sparse matrix-vector multiplication using the block compressed sparse column format as shown in Listing 22 with block sizes of 2-by-1, 4-by-1, and 8-by-1. Missing values in the first three columns denote that no clear main loop was found.

for all float and double vectors, although panels are comparable.

As we have seen with other formats, GCC is less aggressive in unrolling and versioning loops. With this block CSR format, GCC does not version loops at all, instead processing only the values specified by the block size, although it does not create a clean looping pattern when more than 32 operations are to be performed. Despite lower apparent cycles-per-operation when compared to Clang and no vectorized instructions present, the low reciprocal throughputs and adherence to the specified block size seem beneficial in this case. Notably, in the 1-by-2 block case with a panel of eight 32-bit integers, GCC’s fully unrolled loop of single-entry integer loads, multiplications, and additions achieves very low cycles-per-operation, albeit at a relatively high reciprocal throughput compare to the number of operations, as performing all of these loads takes time. With both compilers, we see our reciprocal throughputs match the number of operations fairly evenly, suggesting that little vectorization (and thus relatively more efficient loads) is performed in loading elements. Fairly conclusively, we believe this particular format is not well-suited to the problem of sparse matrix-vector multiplication. As we see relatively better performance, although not as good as the non-blocked version, with the panel cases, this format may fare better with wider panel widths than those we consider.

By comparison, if we consider the performance of the generated block CSC code in Table 3.11, we see a similar pattern to the results of block coordinate form. As block sizes increase, both compilers generally emit code that performs no worse than smaller block sizes. More importantly, with Clang, we often do not see reciprocal throughputs increasing, even as operation counts increase and cycles-per-operation decrease, suggesting that more efficient, vectorized instructions are being successfully generated for these kernels. This is not the case for GCC, though, as it still struggles to generate vector instructions in some cases. As a rule of thumb, single-entry loads from memory take 2 cycles with a reciprocal throughput of 0.5 and operations that involve three elements must perform these loads on our two vector/panel operands, while the matrix coefficients are usually loaded in the start-up “other” cost. These loads are our

bottlenecks between iterations and since two can be launched at a time and take about two cycles to complete, we see a rough one-to-one correspondence between reciprocal throughput and operation counts. However, when vectorized loads can be used, we are loading 128 or 256 bits of values in 3 cycles with the same reciprocal throughput of 0.5. Thus, when the reciprocal throughput of our multiplication kernels tends to be noticeably smaller than the operation count, we expect to see vectorized instruction present.

Notably, we do not see GCC produce vectorized instructions in the panel cases, although it does in most vector cases with block CSC. The ease of optimization for the vector case likely stems from our data access patterns. Both compilers seem quite adept at recognizing this order of operations and accesses better than the seemingly-similar block CSR loop, suggesting that with this loop ordering, writing to contiguous rows of our destination is easier to optimize than reading from contiguous rows of our vector/panel operand, although broadcasting and/or shuffling must be performed before or after the arithmetic in both cases. In fact, if we view the entire kernel (save for initially pulling function inputs and returning) for GCC’s 8-by-1 block case with a vector of floats where we suspect the use of vectorized instructions, we see very similar code to that in our coordinate format, except that loads and broadcasts have been swapped, with the broadcast occurring in the outer loop, once per block of eight rows, rather than in the inner loop. We see this in Listing 23, where we broadcast one row (entry) of our right-hand operand into a vector register, then use the vectorized fused multiply-add to scale an eight-row segment of our sparse matrix by this amount and add it directly into the corresponding eight-row segment of our destination vector. This process of “vertically” adding into our destination, from observation, is critical in achieving vectorized instructions with GCC. Clang, while similarly able to optimize for this case, is also generally happy to perform the inverse operation we saw in block coordinate format with panels, scaling the panel by a coefficient of our sparse matrix and adding it to a corresponding row of the destination panel.

```

1  .L5:
2      mov     rdx, QWORD PTR [r11+r9*8]
3      mov     rcx, QWORD PTR [r11+8+r9*8]
4      cmp     rdx, rcx
5      jnb     .L3
6      lea    rax, [r10+rdx*8]
7      vbroadcastss    ymm1, DWORD PTR [rbx+r9*4]
8      sal    rdx, 5
9      add    rdx, rdi
10     lea    rsi, [r10+rcx*8]
11  .L4:
12     mov    rcx, QWORD PTR [rax]
13     vmovups ymm0, YMMWORD PTR [rdx]
14     lea    rcx, [r8+rcx*4]
15     vfmadd213ps    ymm0, ymm1, YMMWORD PTR [rcx]
16     add    rax, 8
17     add    rdx, 32
18     vmovups YMMWORD PTR [rcx], ymm0
19     cmp    rsi, rax
20     jne    .L4
21  .L3:
22     inc    r9
23     cmp    r12, r9
24     jne    .L5

```

Listing 23: Main body of both loops produced by GCC 10.1 when multiplying an 8-by-1 block by a vector of floats using a block CSC format described in Listing 22.

Among all formats considered so far, this block CSC format is handled well

by both compilers and comes closest to approaching the performance (in cycles-per-operation and reciprocal throughput) of dense matrix-vector multiplication. In fact, this is the first time where we see, effectively “free work” being done in sparse matrices. If we consider both the 4-by-1 and 8-by-1 blocks for vectors of 32-bit integers, we see both compilers execute the 8-by-1 block in identical cycle counts and reciprocal throughputs compared to the 4-by-1 blocks. This is due to the cost of using 128-bit and 256-bit vectorized instructions being the same. Thus, if moving from a 4-by-1 block size to an 8-by-1 block size decreases the number of blocks, less time would be spent on arithmetic (although, as always, cache must be considered). Moving forward into panel formats and orderings with the Method of the Four Russians, we will see that this access pattern of blocks of rows accessed per-column is a pattern both compilers generally treat favorably.

Ultimately, among the block variants of the standard formats we have considered with this analysis, we see that block CSC and, to a lesser degree, Clang with small coordinate blocks, both scale well, yielding to arithmetic performed with vectorized instructions. However, we have also seen some block, compiler, and data pairings yield horrendously inefficient results, particularly with block CSR and unblocked CSR. Perhaps there are some variants to these implementations that could be considered or these formats lend themselves well to convenient cache usage, but we can see with this code analysis that parameter choices can lead to significant cost savings or increases for this sparse matrix-vector multiplication problem.

3.5 Experiment: Block Coordinate Loop Orderings

Before considering the Method of the Four Russians and how its panel-ordered access patterns compare to those we have seen, we will briefly address the issue of loop orderings when it comes to these fixed-width matrix-vector and matrix-panel products. As we saw with dense matrix multiplication at the beginning of Section 3.2, the order of loops may have some impact on the code the compilers produce, even when one dimension is known at compile time.

We would be remiss in not considering the effects of this effectively additional parameter. However, the scope of considering all variants for all formats that already may have as many as three nested inner loops takes an already information-dense problem into a nearly intractable one. Instead, we will examine all variants for block coordinate format with three vectors. This is the only format we have considered thus far that is two-dimensional. With this in mind, we will conduct two sets of miniature experiments in the effects of loop ordering on block sparse matrix-vector and matrix-panel multiplication. First, we will consider both orderings for the vector case and then all three orderings for the panel case. By convention, we will refer to these as “i-j” or “j-i” orderings for the vector case, and by one of the six permutations of “i-j-k” for the panel case, where “i” loops over the rows of a block of our matrix or the destination, “j” over the columns of our matrix or rows of the right-hand operand, and “k” over columns of the right-hand operand or destination. Thus, the “k” is fixed at one in the vector case and at four or eight in the panel case. In terms of details, we recompile the code in Listing 24. For brevity, we will only perform this experiment on our 32-bit operands. These most-closely match our anticipated use cases and as we have already seen, 64-bit vectorized integer multiplication is inherently slow due to the lack of an efficient instruction for our desired operations.

```

1  #include <cstdint>
2  using DT = std::uint32_t;
3  using M = DT *__restrict;
4  using I = std::size_t *__restrict;
5  using T = DT *__restrict;
6  #define P 1
7  #define BR 2
8  #define BC 2
9
10 void mul_coo_block_vec(M A, I IA, I JA, T B, T C, std::size_t nb) {
11     for (auto i = 0; i < nb; ++i) {
12         for (auto j = 0; j < BR; ++j) {
13             auto r = IA[i] + j;
14             for (auto k = 0; k < BC; ++k) {
15                 auto c = JA[i] + k;
16                 auto v = A[i*BR*BC + j*BC + k];
17                 for (auto l = 0; l < P; ++l) {
18                     C[r*P + l] += v * B[c*P + l];
19                 }
20             }
21         }
22     }
23 }

```

Listing 24: C++ implementation of sparse matrix-vector multiplication using a block coordinate format. For our experiments, we consider all permutations on ordering the inner three loops.

First, we consider our analysis procedure on the vector case for each of our

Format	Block size	Data	Clang 10.0.0				GCC 10.1			
			ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
i-j	2-by-2	1xint.	4	5	5.5		4	5	6.5	
		1xfloat	4	5	5.75		4	5	6.25	
	4-by-4	1xint.	16	13.5	2.31		16	8.5	2.75	
		1xfloat	16	12.2	2.69		16	8.5	2.31	
	8-by-8	1xint.	64	64	1.42		64	24	0.95	
		1xfloat	64	41	1.11		64	24	0.94	
j-i	2-by-2	1xint.	4	5	5.5		4	6	6.5	
		1xfloat	4	5	6		4	5	6.25	
	4-by-4	1xint.	16	12	2.19		16	16	2.19	
		1xfloat	16	12	2		16	13	2.19	
	8-by-8	1xint.	64	36	1.05		-	-	-	
		1xfloat	64	29	0.94		-	-	-	

Table 3.12: llvm-mca analysis for sparse matrix-vector multiplication using the block coordinate format shown in Listing 24 with a vector of one 32-bit integer or one float. The order of the two outer loops is swapped in each half of the table, corresponding to i-j and j-i orderings of the rows and columns of the sparse matrix. Missing values in the first three columns denote that no clear main loop was found.

standard block sizes (2-by-2, 4-by-4, and 8-by-8), summarized in Table 3.12 for both orderings. We immediately see, somewhat unexpectedly, that both compilers strictly adhere to our given block sizes. While Clang often versions loops to handle multiple unrolled iterations, it does not do so in this circumstance. What we glean from this small experiment is that we see a lot of inconsistency among compilers and block sizes. For 2-by-2 blocks, all sets of parameters produce approximately identical code. However, when we expand to 4-by-4 blocks, we obtain noticeably different code for each compiler-ordering pairing. Clang produces vectorized instructions in both cases, while GCC only does for the “i-j” ordering. However, as we see in the cycles-per-entry column in the “j-i” ordering, its total throughput for a single iteration is not worse than Clang’s vectorized version, so it is understandable why GCC might not have emitted these vector instructions. The differences between Clang’s two versions seems to be minimal, with alternating data movement and arithmetic in the “j-i” variant compared to a couple dozen lines of data movement followed by four 128-bit fused multiply-adds in the “i-j” variant.

Bizarrely, the 8-by-8 block is where we see the strangest inconsistencies. In the “i-j” ordering, Clang produces a loop that performs 64 single-element operations, while this variant is where GCC performs its best and creates vectorized code that handles this case more efficiently than any other in this experiment. Conversely, if we swap the order of the loops, Clang produces code that is very similar to GCC’s “i-j” 8-by-8 block code, despite its own poor performance with “i-j” ordering. However, GCC does not create a simple loop structure in this case, thus we cannot directly compare it to the others with our analysis. We recognize that compilers are quite complex and perform these optimizations staggeringly well, so just because we do not obtain a simple loop structure does not inherently mean the code is not good, but without a simple loop structure, we cannot directly analyze behavior in this data-agnostic manner.

Overall, we see that that even for this simple format, a choice of compiler, loop ordering, or block size are all somewhat dependent on one another. However, whatever inconsistencies we see here are just teasers for what we will see when we consider all

permutations of the panel cases.

For the panel cases, we will consider all six permutations of the inner loops of this block coordinate multiplication and we summarize our llvm-mca analyses in Table 3.13. The one clear, immediate takeaway from this table is that for this particular problem the “i-k-j” ordering is universally the best format for both compilers and the only format optimized well for by compilers. Not surprisingly, the eight-by-eight case is not fully unrolled for either compiler, but what both compilers produce is code that processes 64 operations, very similarly to the code we viewed for the four doubles panel in Listing 19. Outside of the two-by-two blocks with Clang, we are performing more than one multiply and add per cycle with this format, performing more than three per cycle in the 4-by-4 float case. We improve ever-so-slightly on this cycle count for 8-by-8 “j-i-k” and “j-k-i” and 4-by-4 “k-j-i” orderings with Clang, but these minor differences are roughly one cycle across an iteration of the loop. In fact, many of these cycles-per-operation values are better than those we saw in dense matrix-vector and matrix-panel multiplication. Of course, for large blocks and panels, what we are doing is closer to dense multiplication than what we do in unblocked sparse multiplication, but it appears that something about coordinate format works better for panel multiplication than what we were seeing with dense multiplication.

Even though the “i-k-j” ordering is seemingly the strongest contender for both compilers at nearly all block sizes, we still must note that the 8-by-8 case for “j-k-i” and “j-i-k” do perform nearly two to three times faster. Oddly, these orderings fare extremely poorly with 4-by-4 blocks. This suggests that if we definitively wanted 8-by-8 blocks, we should choose one of these other orderings with Clang. However, we again point out that the performance by GCC and Clang for 4-by-4 blocks in this case approach the theoretical maximum efficiency, which is 0.25 cycles-per-operation if no loads are performed. If we recall our table of operations in Table 3.2, we can simultaneously execute two 256-bit fused multiply-adds that take four cycles to complete, thus performing 16 operations in 4 cycles, or 0.25 cycles per operation. Thus, rather than condition a choice of multiplication kernel on block size, using the smaller block size

still nearly reaches our theoretical maximum throughput. This is another case, too, where applying a 4-by-4 block is more than four times faster, in theory, than applying four 2-by-2 blocks, as even if all four two-by-two blocks of integers can be pipelined and executed in the GCC case, we still require $3 \times 5.7 + 32 \times 1 \approx 49.1$ cycles, or 0.38 cycles-per-operation, which is slightly slower than the 0.35 cycles-per-operation of the 4-by-4 block. Over the course of 128 integer multiplies and adds, this is savings of just one integer multiply and utterly insignificant to what would occur if we saw a mispredicted branch or a cache miss.

Regardless, the efficient formats are quite efficient, even if there may be a difference of a few cycles here and there across iterations. What is also interesting is the inefficient cases, both relatively and absolutely. Basically, this is any case with a reciprocal throughput of more than $\frac{1}{4}$ the number of operations performed. In these cases, we usually have no vectorized instructions emitted. Or, if we do, a huge number of data shuffles are also performed, yielding staggeringly poor reciprocal throughputs and execution times. For instance, with “j-i-k” and “j-k-i” orderings, Clang handles 2-by-2 and 8-by-8 cases nearly as well, if not better than in the “i-k-j” ordering. However, 4-by-4 blocks pose a significant problem, with five times the reciprocal throughput and about five times the cycles-per-operation. A similar problem arises when we consider GCC’s code with the “j-i-k” ordering and 4-by-4 blocks, along with Clang’s 8-by-8 blocks with “i-j-k” ordering (with integers only!) and “k-i-j” ordering. This sensitivity to loop ordering and block size may suggest that the blocks we considered previously may not represent the best possible output from our compilers. This sensitivity may not be apparent, even in real-world benchmarks, if not applied to all block sizes, as Clang’s “j-k-i” ordering is quite good for 2-by-2 and 8-by-8 blocks. Further, in the missing cases, the compilers did not generate a simple, nested loop structure. Thus, we cannot evaluate the potential performance of these formats.

Ultimately, our goal in this section was to explore the impact of loop ordering within sparse matrix-vector multiplication in a controlled manner. Due to the wide range of formats and blocks tested, this is another variable to account for that we

Format	Block size	Data	Clang 10.0.0				GCC 10.1			
			ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
i-j-k	2-by-2	8xint.	32	7	1		32	49	2.34	
		8xfloat	32	6.5	0.84		32	34	1.97	
	4-by-4	8xint.	128	16	0.34		32	8	1.13	0/20
		8xfloat	128	13	0.31		32	6	1	0/20
	8-by-8	8xint.	64	55	3.88	0/35	-	-	-	
		8xfloat	64	24	0.86	0/81	-	-	-	
i-k-j	2-by-2	8xint.	32	7.5	1.03		32	5.7	1	
		8xfloat	32	7	1.09		32	5	0.84	
	4-by-4	8xint.	128	16	0.34		128	16	0.35	
		8xfloat	128	13	0.31		128	13	0.3	
	8-by-8	8xint.	64	8	0.52	0/19	64	8	0.56	0/20
		8xfloat	64	4.5	0.75	0/19	64	6	0.59	0/20
j-i-k	2-by-2	8xint.	32	11.7	1.16		32	9.7	1.19	
		8xfloat	32	8.7	1.13		32	7.3	1.03	
	4-by-4	8xint.	128	91	1.44		32	37.5	1.63	0/53
		8xfloat	128	78	1.33		32	32.5	1.69	0/44
	8-by-8	8xint.	64	8	0.44	11/19	-	-	-	
		8xfloat	64	4.5	0.28	11/19	-	-	-	
j-k-i	2-by-2	8xint.	32	6.2	0.97		32	6.5	0.97	
		8xfloat	32	5	0.84		32	5.3	0.88	
	4-by-4	8xint.	128	87.3	1.6		-	-	-	
		8xfloat	128	77.8	1.23		-	-	-	
	8-by-8	8xint.	64	8	0.44	11/19	-	-	-	
		8xfloat	64	4.5	0.28	0/19	-	-	-	
k-i-j	2-by-2	8xint.	32	6.2	0.97		32	5.8	0.97	
		8xfloat	32	5	0.81		32	5	0.88	
	4-by-4	8xint.	128	19.3	0.34		-	-	-	
		8xfloat	128	14	0.31		-	-	-	
	8-by-8	8xint.	64	64	1.27	0/78	64	8	0.56	0/20
		8xfloat	64	36.5	1.13	0/67	64	6	0.59	0/20
k-j-i	2-by-2	8xint.	32	6.5	0.97		32	5.8	0.97	
		8xfloat	32	5.2	0.81		32	5	0.88	
	4-by-4	8xint.	128	21.2	0.35		-	-	-	
		8xfloat	128	16	0.28		-	-	-	
	8-by-8	8xint.	-	-	-		-	-	-	
		8xfloat	-	-	-		-	-	-	

Table 3.13: llvm-mca analysis for sparse matrix-vector multiplication using the block coordinate format shown in Listing 24 with vectors of eight 32-bit integers or floats. The order of the three loops is swapped in each section of the table. Missing values in the first three columns denote that no clear main loop was found.

cannot apply to every format. However, we do see that for block coordinate format, there may be substantial differences in loop orderings. While the “i-k-j” ordering may be a solid choice, it does not perform as well with 8-by-8 blocks as it does with 4-by-4 blocks or as well as other orderings with 8-by-8 blocks. Further, this analysis was limited to 32-bit data types. We anticipate that these results would carry forward to doubles, but not to 64-bit integers, due to the difference in vectorized multiplication with 64-bit integers.

3.6 Dense Method of the Four Russians Analysis

In this section, we apply our `llvm-mca` analysis to the dense Method of the Four Russians to serve as a comparison point when considering sparse implementations in Section 3.7 and in Chapter 4. Further, this analysis will allow us to see if we obtain analogous performance when comparing dense multiplication with the dense Method of the Four Russians and sparse multiplication with a sparse Method of the Four Russians.

Admittedly, the structural differences in code between the Method of the Four Russians and standard matrix multiplication require us to make some assumptions about what we are comparing. In particular, the workflow of generating a lookup table and then applying it for each panel of k columns of our matrix is one that is difficult to directly compare to standard multiplication. This work depends on the value of k and in the sparse case, may vary considerably more. Instead, we will compare the cost of applying a single lookup table with a panel of our matrix. Further, we note that in the Method of the Four Russians we are performing additions, not multiplications and additions. Thus, we will expect some inherent differences between the analysis results of the Method of the Four Russians and our other formats.

First, we consider a skeleton outline of the entire Method of the Four Russians process in Listing 25 and the corresponding assembly output produced by Clang 10.0.0 in Listing 26. While there is ultimately the chance that the table creation and application calls can be inlined and/or combined and optimized, we will continue on to

consider only the analysis of applying a panel of matrix A to a pre-generated lookup table.

```

1  #include <stdint>
2  using DT = std::uint64_t;
3  using M = std::uint64_t *__restrict;
4  using T = DT *__restrict;
5  #define P 1
6
7  void build_table(T, std::size_t, T);
8  void mul_panel(M, T, T, std::size_t, std::size_t);
9
10 void mul_four_russians(M A, T B, T C, std::size_t m, std::size_t n,
    ↪ std::size_t k) {
11     auto table = new DT[P*(1 << k)];
12     auto word_offset = 0;
13     auto bit_offset = 0;
14     auto stride = (n + 63) / 64;
15     for (auto i = 0; i < n; i += k) {
16         build_table(B + i*P, k, table);
17         mul_panel(A+word_offset, table, C, stride, bit_offset);
18         bit_offset += k;
19         word_offset += bit_offset;
20         if (bit_offset >= 64) {
21             bit_offset -= 64;
22             ++word_offset;
23         }
24     }
25 }

```

Listing 25: Skeleton code detailing the workflow of the Method of the Four Russians.

```

1  .LBB0_2:
2      lea    rdi, [r15 + 8*r13]
3      mov    rsi, rbx
4      mov    rbx, rcx
5      mov    rdx, rcx
6      call   build_table(unsigned long*, unsigned long, unsigned long*)
7      movsxd r15, r14d
8      lea    rdi, [r12 + 8*r15]
9      movsxd r12, ebp
10     mov    rsi, rbx
11     mov    rdx, qword ptr [rsp + 48] # 8-byte Reload
12     mov    rcx, qword ptr [rsp + 24] # 8-byte Reload
13     mov    r8, r12
14     call   mul_panel(unsigned long*, unsigned long*, unsigned long*,
    ↪ unsigned long, unsigned long)
15     mov    rcx, rbx
16     mov    rax, qword ptr [rsp + 8] # 8-byte Reload
17     add    rax, r12
18     add    r15d, eax
19     xor    r14d, r14d
20     cmp    eax, 63
21     setg   r14b
22     mov    rdx, qword ptr [rsp + 8] # 8-byte Reload
23     lea    rbp, [r12 + rdx - 64]
24     mov    r12, qword ptr [rsp + 32] # 8-byte Reload
25     cmovle rbp, rax
26     add    r14d, r15d
27     mov    rbx, qword ptr [rsp + 8] # 8-byte Reload
28     mov    r15, qword ptr [rsp + 40] # 8-byte Reload
29     add    r13d, ebx
30     movsxd r13, r13d
31     cmp    r13, qword ptr [rsp + 16] # 8-byte Folded Reload
32     jb    .LBB0_2

```

Listing 26: Clang 10.0.0 output of the main loop of Listing 25.

```

1  #include <stdint>
2  using DT = std::uint32_t;
3  using M = std::uint64_t *__restrict;
4  using T = DT *__restrict;
5  #define K 4 // param of Four Russians
6  #define P 1
7
8  void mul_dense_panel_4rK(M A, T table, T C, std::size_t m,
9                          std::size_t stride, std::size_t offset) {
10     auto mask = ((uint64_t)1 << K) - 1;
11     for (auto i = 0; i < m; ++i) {
12         auto idx = (A[i*stride] >> offset) & mask;
13         for (auto j = 0; j < P; ++j) {
14             C[i*P + j] += table[idx*P + j];
15         }
16     }
17 }

```

Listing 27: C++ code for applying a panel of dense matrix A with a pre-generated lookup table using the Method of the Four Russians. stride is the stride between rows of matrix A , while offset is the starting bit index of the k -bit entry we are extracting from each row.

In Listing 27, we present a sample of the Method of the Four Russians with a given $k = 4$. We note that we do not observe any difference when providing k as a parameter or if it is fixed. We only make the assumption that each k -bit entry is contained within a single 64-bit integer (i.e., we do not have two bits in one integer and two in the next). The general order of operations is that, for each row in our panel, we isolate the necessary bits and use these as an index into our lookup table, adding the

Vector width	Clang 10.0.0			GCC 10.1				
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1x32-bit int.	16	12	6.06		4	7	10.75	
8x32-bit int.	32	8.2	0.78		8	9.5	3.88	
1x64-bit int.	16	12	6.06		4	3	9.25	
4x64-bit int.	16	8.2	1.56		4	4.5	6.75	
1xfloat	16	12	6.25		8	5.5	2.13	
8xfloat	32	8.2	0.88		32	8.5	0.91	
1xdouble	16	12	6.25		4	3	10	
4xdouble	16	8.2	1.75		4	5	6.25	

Table 3.14: llvm-mca analysis for dense matrix-vector multiplication using the Method of the Four Russians with a fixed $k = 4$ as shown in Listing 27. The first column for each compiler notes the number of element additions performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.

corresponding row of this table to our current row of our output vector or panel.

As before, we compile and analyze this code with our standard suite of vector/panel widths and types, summarizing the results of the llvm-mca analysis in Table 3.14. At first glance, these results do not appear particularly good. For vector cases, we are over a dozen times worse than with classical dense matrix multiplication. The panel cases are marginally better, but generally not nearly as efficient as those we saw with sparse formats that were frequently lower than one cycle-per-operation with better reciprocal throughputs. Further, we are doing less work in applying this panel, as we are simply adding. Thus, we would expect better performance with integers than with floats due to slightly faster addition operations. This slight speedup is present with Clang, but not with GCC. What is more interesting is that GCC does produce vectorized code when it unrolls loops. For example, the main loop that handles four unrolled rows of our panel in the 32-bit integer vector case does use vectorized instructions, as we see in Listing 28

```

1  .L6:
2      vmovq   xmm7, QWORD PTR [rax]
3      vmovq   xmm5, QWORD PTR [rax+r13*2]
4      vpinsrq xmm0, xmm7, QWORD PTR [rax+r13], 1
5      vmovdqa xmm6, xmm3
6      vpsrlq  xmm0, xmm0, xmm2
7      vmovdqa xmm7, xmm3
8      vpand   xmm0, xmm0, xmm4
9      vpgatherqd    xmm1, DWORD PTR [rsi+xmm0*4], xmm6
10     vpinsrq  xmm0, xmm5, QWORD PTR [rax+r12], 1
11     inc     rbx
12     vpsrlq  xmm0, xmm0, xmm2
13     add     rax, r14
14     vpand   xmm0, xmm0, xmm4
15     vpgatherqd    xmm5, DWORD PTR [rsi+xmm0*4], xmm7
16     vshufps xmm0, xmm1, xmm5, 68
17     vpadd   xmm0, xmm0, XMMWORD PTR [r11]
18     add     r11, 16
19     vmovdqu XMMWORD PTR [r11-16], xmm0
20     cmp     rbx, r15
21     jne     .L6

```

Listing 28: GCC 10.1 output of the main loop of Listing 27 for a 32-bit integer vector.

Ultimately, we see the `vpad` instruction, but only comes after a series of bit fiddling to perform the isolation and mask of the index bits from the panel followed by vector gather. This seems to be a lot of work to simply perform what could be four independent loads, adds, and stores. Instead, this seems to be a case of overzealous optimization, where a pattern is recognized, but the vectorization is likely not beneficial here. In the case of a vector of doubles and vector of 64-bit integers, we do see this sequence of independent loads and stores instead of this complicated shuffling just to use vector registers. Clang, on the other hand, produces identical code for each case, save for differences in data type. While Clang correctly identifies and performs row operations in the panel case using vectorization, GCC only does in the case of a panel of floats.

While these are not great results, we can draw two inferences from this work. First, the work done shifting and masking, while inexpensive, likely is a cause of problems in achieving good vectorization, as we are using 64-bit values for our matrix, while our data types vary. This may be why the 64-bit case is better than the 32-bit case for GCC. Second, our access patterns, like those in CSR format, are not ideal for adding vectors. Even though we are adding to contiguous elements of our destination vector, our loads occur randomly as they are determined by the index. Thus, loading four values from the panel corresponds to four random reads from our lookup table. In order to perform a vectorized addition, we must separately insert these four values into a vector register before adding this to our destination. Individually inserting values into a vector register from a non-vector register can only be performed one value at a time, so gathering and packing four scattered 32-bit values is more expensive than loading eight contiguous values from memory. Unfortunately, this problem is a characteristic of the Method of the Four Russians, as our speedup requires uses of this lookup table.

While these results are, perhaps, disheartening, we must keep in mind that each operation we perform is using our lookup table, so each operation is the result of 0 to k sums of rows, so while we may require 6.06 cycles-per-operation with Clang and 32-bit integer vectors, each of these operations may, in fact, represent up to k true operations.

While this lessens the blow of expensive arithmetic somewhat, we see that our factor of k speed-up, in practice, may not be as good as expected, as our cost of arithmetic is higher than that in classical dense or even many sparse cases.

```
1  #include <cstdint>
2  using DT = std::uint32_t;
3  using M = std::uint64_t *__restrict;
4  using T = DT *__restrict;
5  #define P 1
6
7  void mul_dense_panel(M A, T table, T C, std::size_t m,
8                      std::size_t stride, std::size_t offset) {
9      for (auto i = 0; i < m; ++i) {
10         auto idx = A[i*stride];
11         for (auto j = 0; j < P; ++j) {
12             C[i*P + j] += table[idx*P + j];
13         }
14     }
15 }
```

Listing 29: C++ code for applying a panel of dense matrix A with a pre-generated lookup table using the Method of the Four Russians. `stride` is the stride between rows of matrix A . A has been pre-processed so its entries directly correspond to indices in our lookup table.

Before addressing this issue further, we will first tackle the cost of isolating and extracting bits, as this step would be handled in the pre-processing stage for sparse matrices. We can easily wave away the cost of this bit extraction by simply assuming that the input panel A has been pre-processed so that we can directly use its elements

Vector width	Clang 10.0.0			GCC 10.1				
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1x32-bit int.	32	24	3.56		4	6.5	9.75	
8x32-bit int.	32	6.8	0.69		8	8.5	3.63	
1x64-bit int.	32	24	3.16		4	3	8	
4x64-bit int.	16	6.8	1.38		4	4.5	6.25	
1xfloat	32	24	3.25		4	6.5	10.5	
8xfloat	32	6.8	0.78		8	8.5	3.38	
1xdouble	32	24	3.25		4	3	8.75	
4xdouble	16	6.8	1.56		4	4.7	5.75	

Table 3.15: llvm-mca analysis for dense matrix-vector multiplication using the Method of the Four Russians with a pre-processed matrix as shown in Listing 29. Each entry of the sparse matrix has been pre-processed so it can be directly used to index into a generated lookup table.

as indices into our table. We illustrate this change in Listing 29 and the results of our llvm-mca analysis in Table 3.15.

At first glance, it may appear that we have somewhat better results than those we saw without pre-processing. Our cycles-per-operation values are generally lower for Clang. However, our number of operations performed has generally doubled, along with reciprocal throughput. As we see with GCC and the 32-bit integer vector case, we are only shaving a half cycle off of our reciprocal throughput and saving about one cycle per entry. The sequence of shifts and masks, while representing a measurable cost, are dwarfed by the expense of loading and inserting non-contiguous values from the lookup table into vector registers (or not using any vectorization at all). While this is technically a speedup, it does not address our primary area of concern: our random access into the table. However, we must always keep in mind that we cannot necessarily compare operations directly, as if $k = 8$ and our matrix is 50% dense, then each operation we are performing actually corresponds to four real operations, yielding less than one cycle-per-entry in the Clang vector cases. However, with sparse matrices, we anticipate smaller values of k , so having expensive arithmetic is still a problem we seek to address.

3.7 Sparse Panel Analysis

We begin our analysis of the sparse panels using `llvm-mca`, by looking at the ELLPACK format. ELLPACK format, while a standard sparse format is a bit peculiar as it is more like a block format, being inherently data-dependent, as selection of its parameter depends on knowledge of the number of non-zero entries per row of the matrix. While we do not expect it to perform well for general cases, if we consider what ELLPACK does, it is somewhat similar to the panels we wish to use with the Method of the Four Russians.

```

1  #include <stdint>
2  using DT = std::uint32_t;
3  using M = DT *__restrict;
4  using I = std::size_t *__restrict;
5  using T = DT *__restrict;
6  #define P 1
7  #define BC 2
8
9  void mul_ellpack(M A, I JA, T B, T C, std::size_t m) {
10     for (auto i = 0; i < m; ++i) {
11         for (auto j = 0; j < BC; ++j) {
12             auto r = i;
13             auto c = JA[i*BC + j];
14             auto v = A[i*BC + j];
15             for (auto k = 0; k < P; ++k) {
16                 C[r*P + k] += v * B[c*P + k];
17             }
18         }
19     }
20 }

```

Listing 30: C++ code for sparse matrix-vector multiplication using the ELLPACK format with small block sizes.

In Listing 30, we present C++ code for performing sparse matrix-vector multiplication using the ELLPACK format. We see our corresponding llvm-mca analysis results in Table 3.16. The panels used in the Method of the Four Russians are almost like repeated ELLPACK blocks. If we recall with ELLPACK, for each row we explicitly store the same number of entries (possibly, some are padded zeros) equal to

Block size	Vector width	Clang 10.0.0				GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
2	1x32-bit int.	32	19	3.78		2	3.5	12.5	
	8x32-bit int.	16	4	1.88		16	16	2.81	
	1x64-bit int.	16	20	6.06		2	3.5	12.5	
	4x64-bit int.	8	6	4.25		8	8	3.88	
	1xfloat	16	19	7.19		2	3.5	18.5	
	8xfloat	16	3.5	1.56		16	14	2.13	
	1xdouble	16	20	4.63		2	3.2	21	
	4xdouble	8	3.5	3.13		8	7	4	
4	1x32-bit int.	16	25	15.63		4	6	14.75	
	8x32-bit int.	32	7.5	1.06		32	10.2	1.34	
	1x64-bit int.	4	6.5	6.25		4	6.5	5.5	
	4x64-bit int.	16	12	2.44		16	16	2.38	
	1xfloat	16	26	11.56		4	6	14	
	8xfloat	32	6.5	1.09		32	9	1.31	
	1xdouble	16	24.5	13.63		4	3.5	13	
	4xdouble	16	6.5	2.25		16	12.5	2.69	
8	1x32-bit int.	16	54	22.75		8	6	8.13	
	8x32-bit int.	64	61	3.66		8	8	3.13	21/16
	1x64-bit int.	8	12.5	3.63		8	12.5	3.63	
	4x64-bit int.	32	24	1.66		32	41.5	1.88	
	1xfloat	16	53.5	26.88		8	6	8.13	
	8xfloat	64	45.5	2.56		8	5	2.75	13/11
	1xdouble	32	53.5	11.63		8	6.5	7	
	4xdouble	32	12.5	1.63		32	9.5	1.5	

Table 3.16: llvm-mca analysis for sparse matrix-vector multiplication using the ELL-PACK format shown in Listing 30 with 2, 4, or 8 entries per row.

the most non-zeros in a row. In all fairness, the format did arise to solve a particular problem, not as a general format, as aside from a couple panel cases, the code Clang produces yields the worst cycles-per-operation count we have seen across all formats. GCC, while better, is still fairly poor compared to all other formats. Despite being somewhat blocked, all loads from the right-hand operand are random, as they are in the Method of the Four Russians. While they may be ordered, there is no opportunity to ever perform adjacent reads from the right-hand operand. Further, performing vectorized writes by using values in multiple rows is difficult, as loads from consecutive rows of the coefficient matrix are not contiguous in memory. While significantly worse than anything we have seen for most cases, we include this format for completeness.

Now, we will consider how to multiply sparse panels, based on the format described in Section 3.7. This is very similar to the dense case, except we do not perform action on every row, just those in the IA vector. Our reference code, as always, is in Listing 31, while our llvm-mca analysis is summarized in Table 3.17.

```

1  #include <stdint>
2  using DT = std::uint32_t;
3  using M = std::uint64_t *__restrict;
4  using I = std::size_t *__restrict;
5  using T = DT *__restrict;
6  #define P 1
7
8  void mul_sparse_panel(M A, I IA, T table, T C, std::size_t nnz) {
9      for (auto i = 0; i < nnz; ++i) {
10         auto r = IA[i];
11         auto idx = A[i];
12         for (auto j = 0; j < P; ++j) {
13             C[r*P + j] += table[idx*P + j];
14         }
15     }
16 }

```

Listing 31: C++ code for sparse matrix-vector multiplication with a panel and pre-generated lookup table using a sparse panel format with the Method of the Four Russians.

Despite some differences in loop unrolling factors, this table could be a copy of the dense, processed case. Both Clang and GCC produce nearly-identical code. Clang, as always, vectorizes its panel operations, while GCC does not in this circumstance. Otherwise, particularly for the vector cases, we obtain assembly code that produces the expected number of single-entry loads, adds, and stores. GCC’s values in the table appear slightly worse. They are, again, in the panel case due to having no vectorization, but in the vector case, if we were to pipeline four iterations to match Clang’s unrolled loops, we would obtain similar cycles-per-operation.

Vector width	Clang 10.0.0			GCC 10.1				
	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1x32-bit int.	4	8	5.75		1	2	20	
8x32-bit int.	32	8	0.69		8	9	3.63	
1x64-bit int.	4	8	5.75		1	2	20	
4x64-bit int.	16	8	1.38		4	5	6.25	
1xfloat	4	8	5.25		1	2	19	
8xfloat	32	8	0.78		8	10	3.5	
1xdouble	4	8	5.25		1	2	19	
4xdouble	16	8	1.56		4	6	6	

Table 3.17: llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the sparse panel format shown in Listing 31. The first column for each compiler notes the number of element additions performed in the main loop of the multiplication code. The second column is reciprocal throughput, in cycles. The third column is the number of cycles required to execute the main loop divided by the number of operations performed. The final column denotes cycles spent in clean-up stages at the end of a row or at the beginning of a row, respectively.

Immediately, this suggests that our sparse Method of the Four Russians is not necessarily worse than in the dense case. However, due to having fewer entries and smaller values of k , the parameter of the Method of the Four Russians, we would like to out-perform the dense case to make up for this decreased benefit.

To that end, we will present a sparse block-panel format in both one- and two-dimensions. Blocking multiple rows together means we are reading lookup table indices from contiguous rows of our sparse matrix and correspondingly adding two contiguous rows of our destination vector or panel. Blocking multiple columns together, while seemingly strange as the classic Method of the Four Russians has one entry per row within a panel, corresponds to the $\binom{t}{k}$ tables presented in Section 2.2, where we break rows of t entries into potentially multiple entries of no more than k bits each. For a $\binom{7}{3}$ table, for instance, we could have three entries within a single row of our panel. The standard sparse panel format, with its IA vector can handle this format just fine, but we will consider explicitly telling the compiler to expect multiple entries in a row to see

if this additional operation can be optimized. Finally, we will consider combining both blocking factors. This blocking requires pre-processing our sparse matrix on creation, so it is not an optimization that can be readily applied to the dense Method of the Four Russians. Ideally, this would give us the potential for a relative speed-up to ameliorate the decrease in efficiency due to having fewer entries.

```

1  #include <stdint>
2  using DT = std::uint32_t;
3  using M = std::uint64_t *__restrict;
4  using I = std::size_t *__restrict;
5  using T = DT *__restrict;
6
7  #define BR 2
8  #define BC 1
9  #define P 1
10
11 void mul_sparse_block_panel(M A, I IA, T table, T C, std::size_t nnzb)
    ↪ {
12     for (auto i = 0; i < nnzb; ++i) {
13         for (auto k = 0; k < BR; ++k) {
14             auto row = IA[i] + k;
15             for (auto l = 0; l < BC; ++l) {
16                 auto m_idx = A[i*BR*BC + k*BC + l];
17                 for (auto j = 0; j < P; ++j) {
18                     C[row*P + j] += table[m_idx*P + j];
19                 }
20             }
21         }
22     }
23 }

```

Listing 32: C++ code for block sparse matrix-vector multiplication with a panel and pre-generated lookup table using a blocked sparse panel format with the Method of the Four Russians.

In Listing 32 we present C++ code that shows this blocking in action. This appears to have more complicated indexing, but as our unblocked sparse panel format does not perform vectorized loads from the panel, these more complicated access patterns should not be worse. Due to the larger number of blocks we consider, we separate our llvm-mca analysis summaries into three tables: Table 3.18 has the results of blocking for rows, Table 3.19 has the results of blocking for columns, while Table 3.20 has a combination of the two blocking dimensions.

When looking at the results for the row-blocked sparse panel multiplication, we see a fairly similar table to what we saw with the unblocked sparse panel multiplication. By explicitly giving a block size, we have loops unrolled different numbers of times to the unblocked case, but the general patterns within each block size are consistent to the unblocked version. Further, we do not realize any true speedups with either compiler: the lower cycles-per-operation values with larger block sizes come with larger reciprocal throughputs. While this will, as always, result in fewer branches and thus fewer possibilities for mispredicted branches, we are still not achieving vectorization in the vector cases due to the random reads from the lookup table. By comparison, the blocked variant of CSC with an 8-by-1 block size and 32-bit vectors with Clang performed 32 operations with lower reciprocal throughput and cycle count than the eight performed in this blocked sparse panel format. Compared to the dense Method of the Four Russians, we are not performing any worse with this block format, but blocking in this dimension has not yielded any new use of vectorization.

Sadly, we see similar patterns as we look at the results of blocking in the column dimension and in two-dimensional blocking. Admittedly, we do begin to see some new vectorization from GCC in the 4-by-4 panel case, the panel case was not problematic. While the panel cases do improve as we increase block sizes, this is again due to having larger unrolled loops: the reciprocal throughputs increase correspondingly. However, efficient panel cases are good to have. Unfortunately, we still do not find any vectorization with our vector cases. Despite blocking, the issue of performing random, single-element reads makes vectorization intractable. In Chapter 4, we will approach

Block size	Vector width	Clang 10.0.0			GCC 10.1				
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
2-by-1	1x32-bit int.	4	7	5.75		2	3.5	10.5	
	8x32-bit int.	32	7	0.69		16	17.5	2.56	
	1x64-bit int.	4	6	5.75		2	3	15.5	
	4x64-bit int.	16	7	1.38		8	9.5	3.63	
	1xfloat	4	7	5.5		2	3.5	10.5	
	8xfloat	32	7	0.78		16	18.8	2.63	
	1xdouble	4	6	6.5		2	3	17	
	4xdouble	16	7	1.56		8	10.8	3.5	
4-by-1	1x32-bit int.	8	10	3.25		4	6	9.5	
	8x32-bit int.	32	6.5	0.75		8	8.5	3.63	4/10
	1x64-bit int.	8	10	4.13		4	3.5	8.5	
	4x64-bit int.	16	6.5	1.5		16	18.5	2.38	
	1xfloat	8	10	3.63		4	4.2	9.75	
	8xfloat	32	6.5	0.84		8	8.5	3.38	4/10
	1xdouble	8	10	4.63		4	3.5	9.25	
	4xdouble	16	6.5	1.69		16	20.5	2.63	
8-by-1	1x32-bit int.	8	9	3.88		8	6	5	
	8x32-bit int.	64	12.5	0.48		8	8.5	3.63	4/10
	1x64-bit int.	8	9.5	3.75		8	6.5	6.63	
	4x64-bit int.	32	12.5	0.97		4	4.5	6.25	4/10
	1xfloat	8	9	4.38		8	6	5.38	
	8xfloat	64	12.5	0.53		8	8.5	3.38	4/10
	1xdouble	8	9.5	4.63		8	6.5	7.38	
	4xdouble	32	12.5	1.06		4	4.7	5.75	4/10

Table 3.18: llvm-mca analysis for sparse matrix-vector multiplication using the block sparse panel format shown in Listing 32 with block sizes of 2-by-1, 4-by-1, and 8-by-1. These block dimensions indicate that we are accessing consecutive rows of the matrix, with one (potentially different) pattern per row. Note: operations performed are all additions.

Block size	Vector width	Clang 10.0.0			GCC 10.1			other
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	
1-by-2	1x32-bit int.	4	6	5	2	3	8.5	
	8x32-bit int.	32	6.2	0.75	16	17.5	2.56	
	1x64-bit int.	4	6	5	2	3	8.5	
	4x64-bit int.	16	6.2	1.5	8	9.5	3.75	
	1xfloat	4	6	6.25	2	3	11.5	
	8xfloat	32	6.2	0.88	16	17.5	2.25	
	1xdouble	4	6	6.25	2	3	11.5	
	4xdouble	16	6.2	1.75	8	10.2	3.5	
1-by-4	1x32-bit int.	8	10	3.13	4	5	4.75	
	8x32-bit int.	32	5.2	0.75	32	5	0.91	
	1x64-bit int.	8	10	3.13	4	5	4.75	
	4x64-bit int.	16	5.2	1.5	16	18.5	2.69	
	1xfloat	8	10	4.5	4	6	11.75	
	8xfloat	32	5.2	1.06	32	5	1.09	
	1xdouble	8	10	4.5	4	3.7	12.5	
	4xdouble	16	5.2	2.13	16	18.5	2.69	
1-by-8	1x32-bit int.	4	5.5	5.25	4	5.5	6.25	
	8x32-bit int.	32	5.5	0.72	16	13.5	2	4/16
	1x64-bit int.	4	5.5	5.25	4	5.5	6.25	
	4x64-bit int.	16	5.5	1.44	16	18.5	2.69	
	1xfloat	4	5.5	5.5	4	5.5	6.25	
	8xfloat	32	5.5	0.88	16	13	2.31	4/10
	1xdouble	8	10	4	4	5.5	6.25	
	4xdouble	16	5.5	1.75	16	18.8	2.63	

Table 3.19: llvm-mca analysis for sparse matrix-vector multiplication using the block sparse panel format shown in Listing 32 with block sizes of 1-by-2, 1-by-4, and 1-by-8. These block dimensions indicate that we are accessing one row of our matrix and applying multiple patterns to this row. This situation may arise when applying $\binom{t}{k}$ tables as described in Section 2.2. Note: operations performed are all additions.

Block size	Vector width	Clang 10.0.0			GCC 10.1				
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
2-by-2	1x32-bit int.	4	5.5	5.25		4	5.5	6.25	
	8x32-bit int.	32	5.5	0.72		16	13.5	2	4/16
	1x64-bit int.	4	5.5	5.25		4	5.5	6.25	
	4x64-bit int.	16	5.5	1.44		16	18.5	2.69	
	1xfloat	4	5.5	5.5		4	5.5	6.25	
	8xfloat	32	5.5	0.88		16	13	2.31	4/10
	1xdouble	8	10	4		4	5.5	6.25	
	4xdouble	16	5.5	1.75		16	18.8	2.63	
4-by-2	1x32-bit int.	8	9	2.88		8	10.5	3.38	
	8x32-bit int.	64	10.5	0.41		16	13	1.75	4/20
	1x64-bit int.	8	9	3.5		8	10.5	3.38	
	4x64-bit int.	32	10.5	0.91		8	7	2.75	4/10
	1xfloat	8	9	3.5		8	10.5	3.63	
	8xfloat	64	10.5	0.5		16	13	2.31	4/10
	1xdouble	8	9	4.75		8	10.5	3.63	
	4xdouble	32	10.5	1.06		8	7	3.5	4/10
4-by-4	1x32-bit int.	16	17	2.25		16	18.5	2.25	
	8x32-bit int.	128	18.5	0.29		32	4.5	0.88	4/10
	1x64-bit int.	16	17	2.38		16	18.5	2.25	
	4x64-bit int.	64	18.5	0.58		16	12	2.25	4/10
	1xfloat	16	17	2.81		16	20	9.19	
	8xfloat	128	18.5	0.34		32	4.5	1.06	4/10
	1xdouble	16	17	2.81		16	12.5	8.44	
	4xdouble	64	18.5	0.69		16	12	2.31	4/10

Table 3.20: llvm-mca analysis for sparse matrix-vector multiplication using the block sparse panel format shown in Listing 32 with block sizes of 2-by-2, 4-by-2, and 4-by-4. The first number of the block size denotes the number of consecutive rows of our matrix we access. The second number is the number of patterns per row applied to each row. Note: operations performed are all additions.

this problem from a new angle, but for now we must be content with efficient panel operations.

3.8 Analysis Conclusions

In this chapter, we presented a novel approach to considering the efficiency of sparse matrix-vector multiplication in a data-agnostic way. Rather than concern ourselves with distributions of non-zero entries, cache performance, and the intangible problems such as loop predictions and irregular vector lengths, we used the `llvm-mca` analysis tool to exhaustively analyze the actual arithmetic performed in ideal cases. Thus, if we have guarantees that our data are accessed in an efficient manner, we focus solely on how well each format can actually perform a matrix-vector or matrix-panel multiplication.

If we consider any one of our Method of the Four Russians-inspired formats in this chapter or the previous chapter, the task of applying our sparse panels to a generated lookup table remains constant, even if differences in data may drastically change what the indices or table-generation costs look like. Thus, identifying and selecting an efficient multiplication kernel is still critical.

By considering the reciprocal throughputs, number of operations performed, and cycles-per-operation, we can choose formats for this problem in general. If we have some information about entry distributions for a particular matrix, we can use these analyses to select an appropriate format, block size, or compiler for the task.

Overall, what we have gleaned is that for the task of matrix-vector multiplication, sparse formats do not come particularly close to the performance of dense matrix-vector multiplication in theoretical efficiency per operation. However, we find that sparse formats, especially using Clang, are fairly efficient at performing matrix-panel multiplication. Curiously, they nearly all perform better than classic dense matrix-vector multiplication, at least in the lens of this form of analysis.

Compressed sparse column and especially block compressed sparse column formats come closest to the performance of dense matrices in performing matrix-vector

multiplication, while also performing admirably for matrix-panel multiplication. However, we saw better theoretical performance for panels with large coordinate format blocks, achieving nearly theoretically perfect performance with an 8-by-8 block with Clang and a “j-k-i” or “j-i-k” ordering of inner loops in our multiplication kernels.

Ultimately, speed in terms of reciprocal throughput and in terms of cycle count both require the use of vectorized instructions. Clang, in these experiments, produces more vectorized code. However, we see that changes in data type, panel width, compiler, and loop ordering can cause significant changes in the quality of code produced, even for a simple format like coordinate form with three vectors. Presumably, if we were to consider additional compilers or target different hardware, we may find different results.

What we have also seen is that the sparse Method of the Four Russians does not perform its arithmetic any less efficiently than the dense Method of the Four Russians. Thus, if we have the density and entry distribution to justify the creation of lookup tables, a sparse panel representation will not be any slower when it comes to applying the lookup tables. While generally not as efficient as other formats, Clang’s vectorized panel-panel multiplication achieved vectorization, acceptable reciprocal throughput, and less than one cycle-per-operation performance. Unfortunately, the inherent structure of randomly reading from the lookup table severely limits the possibility of performing matrix-vector operations using vectorized instructions. Thus, we draw a somewhat inconclusive conclusion when it comes to comparing the dense Method of the Four Russians to dense matrix multiplication and the sparse Method of the Four Russians to sparse matrix multiplication, as when it comes to the underlying arithmetic, both are surpassed by other sparse formats for the panel and vector case, while neither comes close to dense matrix-vector multiplication. This inconclusive result also suggests that selection of compiler, format, vector type, and vector width are all factors to consider when performing sparse matrix-vector multiplication. Further, we were not able to achieve any speedup that addresses the potential limitations of the $\binom{t}{k}$ table format, despite some likely practical opportunities.

However, we conclude this chapter with a reminder that differences in sparse formats are more important when it comes to their ability to be blocked into cache-sized pieces. The memory savings of compressed or block formats are not modeled here, beyond the need for differing numbers of loads when applying them to our vectors and panels. What we will strive for in the next chapter is some manner of achieving efficient sparse matrix-vector multiplication using the Method of the Four Russians.

Chapter 4

SPARSE METHOD OF THE FOUR RUSSIANS: TRANSPOSED PANEL FORMAT

In this chapter, we present a modified sparse Method of the Four Russians that performs well when multiplying by a vector operand. The chief hurdle in the way of performing efficient sparse matrix-vector multiplication with a sparse panel format is that our loads from lookup tables are random, rather than contiguous, so we cannot take advantage of vector registers in order to speed up our arithmetic. To counter this, we will propose a new format, perform our llvm-mca analysis on this format, and discuss the concerns that arise with this new form.

4.1 Transposing Panels

Our proposed new format is to transpose our problem. Within a panel, rather than store indices in row-major order, we store rows in index-major order. With sparse matrices, we expect to have relatively few patterns. Further, we require repeated patterns within panels in order to obtain any speedup from the generation of our lookup tables. Thus, if we have some block format that is capable of finding and applying the same table index in contiguous-enough rows of our panel, we may be able to take advantage of vectorization.

We illustrate this idea simply in Figure 4.1 and Figure 4.2, taking the sparse panel in the first example and transposing it to the example in the second. In this case, we have three vectors: A , an optional vector that contains the values of the patterns we are interested in. It is helpful to think about this vector when viewing this format, but we only need to know that the order of counts in PA aligns with the order of our patterns. With this representation we could just walk down the rows of our lookup

m	$=5$
IA	$=[0, 1, 5, 7, 8]$
A	$=[10, 01, 11, 11, 01]$

Figure 4.1: Relevant values for applying a single panel in a sparse panel format.

np	$=3$
A	$=[01, 10, 11]$
PA	$=[0, 2, 3, 5]$
IA	$=[1, 4, 0, 2, 3]$

Figure 4.2: Transposed variant of the panel in Figure 4.1.

table, since we are explicitly storing the number of row for which each pattern. Thus, also storing the index is not necessary. PA stores a running count of the number of rows in which a given pattern is present. Thus $PA[i]$ up to, but not including $PA[i+1]$ is the range of indices in IA that represent rows in which pattern $A[i]$ is present in the panel. With this transposition of access, we now no longer perform random accesses to the lookup table (for each panel). The downside is that we now potentially traverse our destination once for each pattern. This access is sequential, so it remains difficult to intuitively compare these access patterns.

In Listing 33 we present C++ code to apply this transposed panel. The total amount of work we perform is no different from the standard sparse panel format, we are just accessing items in different orders. In Listing 34, we consider a block variant of this, where IA is assumed to be blocked, with BR consecutive entries for each value of j in the middle loop.

```

1  #include <stdint>
2  using DT = std::uint32_t;
3  using M = std::uint64_t *__restrict;
4  using I = std::size_t *__restrict;
5  using T = DT *__restrict;
6  #define P 1
7
8  void mul_transposed_panel(M A, I PA, I IA, T table, T C, std::size_t
↪  np) {
9      for (auto i = 0; i < np; ++i) {
10         for (auto j = PA[i]; j < PA[i+1]; ++j) {
11             for (auto k = 0; k < P; ++k) {
12                 C[IA[j]*P + k] += table[A[i]*P + k];
13             }
14         }
15     }
16 }

```

Listing 33: C++ code to apply a transposed panel using a pre-generated lookup table.

Using this block format, we immediately obtain vectorized code when looking at the output of both compilers. For reference, we see what GCC, our compiler that has been notoriously difficult to get to produce vectorized code, produces in Listing 35.

The glaring issue with this format is that we have no ability to store explicit zeros. Each block contains implicit contiguous row indices. Instead of directly adding all rows in a block, we would like to modify our format to allow us to selectively zero out some of our implicit rows.

In Figure 4.3, we introduce yet another variant, where we have added a fourth vector: MA, standing for mask. Each value of MA is a block-sized bitmask with a one wherever the given row of the block should be used and zero where no operation should be performed. We also note that the IA vector is smaller, as each value of IA now corresponds to the start of a block. However, our new vector's size is equal to the number of blocks within this panel of the matrix. Our total size of this format is the number of patterns (recall that we do not need to explicitly store the vector A)

```

1  #include <stdint>
2  using DT = std::uint32_t;
3  using M = std::uint64_t *__restrict;
4  using I = std::size_t *__restrict;
5  using T = DT *__restrict;
6  #define BR 8
7  #define P 1
8
9  void mul_transposed_panel(M A, I PA, I IA, T table, T C, std::size_t
  ↪ np) {
10     for (auto i = 0; i < np; ++i) {
11         for (auto j = PA[i]; j < PA[i+1]; ++j) {
12             for (auto b = 0; b < BR; ++b) {
13                 for (auto k = 0; k < P; ++k) {
14                     C[(b+IA[j*BR])*P + k] += table[A[i]*P + k];
15                 }
16             }
17         }
18     }
19 }

```

Listing 34: Block variant of the transposed panel multiplication code in Listing 33

```

1  .L4:
2      mov     rdx, QWORD PTR [rax]
3      add     rax, 64
4      lea    rdx, [r8+rdx*4]
5      vpaddd ymm0, ymm1, YMMWORD PTR [rdx]
6      vmovdqu YMMWORD PTR [rdx], ymm0
7      cmp     rcx, rax
8      jne    .L4

```

Listing 35: Main loop produced by GCC with the C++ code in Listing 34

np	$=3$
A	$=[01, 10, 11]$
PA	$=[0, 1, 2, 3]$
MA	$=[1001, 1000, 1100]$
IA	$=[1, 0, 2]$

Figure 4.3: Block, masked transposed variant of the panel in Figure 4.1.

plus two times the number of blocks, compared to our normal sparse panel form which was two times the number of non-zero entries. If blocks are sufficiently full, then these formats are similar in size.

However, we have found that using this mask in an efficient manner requires a great deal of engineering. Unfortunately, there is no 128- or 256-bit broadcast-subject-to-mask vectorized instruction that would directly align with our format. This exists for the few processors that support 512-bit SIMD instructions, but we must find some alternative.

```

1  #include <stdint>
2  #include <array>
3  using DT = std::uint32_t;
4  using I = std::size_t *__restrict;
5  using M = std::uint64_t *__restrict;
6  using T = DT *__restrict;
7  #define P 1
8  #define BR 64
9
10 void mul_sparse_transposed_panel_csp (M A, I PA, I IA, T table, T C,
    → std::size_t m, std::size_t np)
11 {
12     std::array<std::uint64_t, BR> offsets;
13     for (auto i = 0; i < BR; ++i) {
14         offsets[i] = (std::uint64_t)1 << i;
15     }
16     for (auto i = 0; i < np; ++i) {
17         std::array<DT, BR*P> val;
18         for (auto b = 0; b < BR; ++b) {
19             for (auto k = 0; k < P; ++k) {
20                 val[b*P + k] = table[i*P + k];
21             }
22         }
23
24         for (auto j = PA[i]; j < PA[i+1]; ++j) {
25             auto row = IA[j];
26             std::array<DT, BR*P> temp;
27             for (auto k = 0; k < BR*P; ++k) {
28                 temp[k] = A[BR*j];
29             }
30             for (auto k = 0; k < P; ++k) {
31                 for (auto b = 0; b < BR; ++b) {
32                     temp[k*BR + b] &= offsets[b];
33                 }
34             }
35             for (auto k = 0; k < P; ++k) {
36                 for (auto b = 0; b < BR; ++b) {
37                     temp[k*BR + b] = !temp[k*BR + b];
38                 }
39             }
40             for (auto k = 0; k < P; ++k) {
41                 for (auto b = 0; b < BR; ++b) {
42                     temp[k*BR + b] -= 1;
43                 }
44             }
45             for (auto k = 0; k < P; ++k) {
46                 for (auto b = 0; b < BR; ++b) {
47                     temp[k*BR + b] &= val[b*P + k];
48                 }
49             }
50             for (auto b = 0; b < BR; ++b) {
51                 for (auto k = 0; k < P; ++k) {
52                     C[P*(row+b) + k] += temp[b*P + k];
53                 }
54             }
55         }
56     }
57 }

```

Listing 36: C++ implementation of the transposed sparse panel format.

```

1  #include <stdint>
2  #include <array>
3  using DT = std::uint32_t;
4  using I = std::size_t *__restrict;
5  using M = std::uint64_t *__restrict;
6  using T = DT *__restrict;
7  #define P 1
8  #define BR 32
9
10 void mul_sparse_transposed_panel_csp2 (M A, I PA, I IA, T table, T C,
    → std::size_t m, std::size_t np)
11 {
12     std::array<std::uint64_t, BR> offsets;
13     for (auto i = 0; i < BR; ++i) {
14         offsets[i] = i;
15     }
16     for (auto i = 0; i < np; ++i) {
17         std::array<DT, BR*P> val;
18         for (auto b = 0; b < BR; ++b) {
19             for (auto k = 0; k < P; ++k) {
20                 val[b*P + k] = table[i*P + k];
21             }
22         }
23
24         for (auto j = PA[i]; j < PA[i+1]; ++j) {
25             auto row = IA[j];
26             std::array<DT, BR*P> temp;
27             for (auto k = 0; k < P; ++k) {
28                 for (auto b = 0; b < BR; ++b) {
29                     temp[k*BR + b] = A[BR*j] >> offsets[b];
30                 }
31             }
32             for (auto b = 0; b < BR*P; ++b) {
33                 temp[b] &= 1;
34             }
35             for (auto b = 0; b < BR*P; ++b) {
36                 temp[b] = temp[b] - 1;
37             }
38             for (auto b = 0; b < BR*P; ++b){
39                 temp[b] &= val[b];
40             }
41             for (auto b = 0; b < BR; ++b) {
42                 for (auto k = 0; k < P; ++k) {
43                     C[P*(row+b) + k] += temp[b*P + k];
44                 }
45             }
46         }
47     }
48 }

```

Listing 37: Alternative C++ implementation of the transposed sparse panel format.

To that end, we present two C++ multiplication routines, shown in Listing 36 and Listing 37. In these codes, we do not explicitly store the patterns in the vector

A, but read down our lookup table. Instead, A corresponds to MA in our previous example, a 64-bit integer that serves as a bitmask. Effectively, both of these routines are doing the same work, but there are differences in the exact operations required to achieve our outcome.

The main idea is that for each row of our lookup table, we iterate across all blocks of rows in our panel that contain this pattern. For each BR-sized block, we load the BR-bit bitmask from A and broadcast the bitmask into BR separate integers. Then, we isolate each individual bit: the 0-th in the 0-th copy, the first in the first copy, and so on. This occurs in lines 32 and 25 in the two versions, respectively. Getting this process vectorized required creation of a helper array, offsets, that is the main difference between the two formats. In the first version, this offsets array contains BR items, each of which is the value one shifted accordingly (e.g., 1, 10, 100, 100, ... BR times). In the second case, we just store the index of these ones (0, 1, 2, ...). The overarching goal is to take this bitmask for each of our BR values in a block and transform the bitmask into either a mask of all zeros or a mask of all ones, which we then use to mask out all a broadcast copy of our pattern. This masking occurs in lines 37 and 49, respectively. The intermediate work is trying different ways at achieving this conversion from bitmask to all zeros or all ones in a way both compilers can optimize. We note that in line 20 of each listing, we explicitly broadcast the contents of our given row of our lookup table into the temporary array val. While Clang was generally happy about vectorizing code, GCC required some of these additional temporary arrays, even though it does not actually build them, when considering the generated assembly.

```

1  #include <stdint>
2  #include <array>
3  using DT = std::uint32_t;
4  using I = std::size_t *__restrict;
5  using M = std::uint64_t *__restrict;
6  using T = DT *__restrict;
7  #define P 1
8  #define BR 1
9
10 void mul_sparse_transposed_panel_coo (M A, I JA, I IA, T table, T C,
    → std::size_t nb)
11 {
12     std::array<std::uint64_t, BR> offsets;
13     for (auto i = 0; i < BR; ++i) {
14         offsets[i] = i;
15     }
16     for (auto i = 0; i < nb; ++i) {
17         auto pattern = JA[i];
18         auto row = IA[i];
19         auto mask = A[i];
20
21         std::array<DT, BR*P> temp;
22         for (auto k = 0; k < P; ++k) {
23             for (auto b = 0; b < BR; ++b) {
24                 temp[k*BR + b] = mask & (1 << offsets[b]);
25             }
26         }
27         for (auto k = 0; k < P; ++k) {
28             for (auto b = 0; b < BR; ++b) {
29                 temp[k*BR + b] >>= offsets[b];
30             }
31         }
32         for (auto k = 0; k < P; ++k) {
33             for (auto b = 0; b < BR; ++b) {
34                 temp[k*BR + b] -= 1;
35             }
36         }
37         for (auto k = 0; k < P; ++k) {
38             for (auto b = 0; b < BR; ++b) {
39                 C[P*(row+b) + k] += table[pattern*P + k] & (~temp[b*P + k]);
40             }
41         }
42     }
43 }

```

Listing 38: A coordinate-like alternative C++ implementation of the transposed sparse panel format.

To expand on this even more, we consider a coordinate-like variant of this, too, rather than this CSC-like format. From our previous analysis, coordinate format generally produced clearer loops, so we present this third alternative in Listing 38. In

this example, we do not store a running count of the number of items for each panel, but instead have a triple of pattern (table index), starting row of block, and bitmask. When we consider our panel in the context of an entire matrix of an unknown number of panels, this coordinate-like format may be beneficial if we have many panels or many empty panels in our matrix. Fundamentally, the work is the same. In the first 34 lines we are primarily performing bit-fiddling to broadcast and convert our bitmasks. In line 39, we actually perform our addition. Since our pattern may be different between blocks, we do not actively place it into a temporary array in this case.

For further explanation, if we consider all panels within our matrix, we would need to store a count of the number of blocks for each pattern for each panel, requiring us to have one array of length equal to the number of panels, while our original JA array's length would be equal to the number of patterns times the number of panels. This would be $\frac{2^k n}{k}$. It is unclear how large this value might be for a sparse matrix with large n and small k , so the simpler coordinate-like triple vector format yields a simpler-to-compute size, as the size of each of its three components is definitely no larger than the number of non-zeros in the matrix.

```

1  #include <stdint>
2  #include <array>
3  using DT = float;
4  using ST = std::uint32_t;
5  using I = std::size_t *__restrict;
6  using M = std::uint64_t *__restrict;
7  using T = DT *__restrict;
8  #define P 1
9  #define BR 32
10
11 void mul_sparse_transposed_panel_coo (M A, I JA, I IA, T table, T C,
    ↪ std::size_t nb)
12 {
13     std::array<std::uint64_t, BR> offsets;
14     for (auto i = 0; i < BR; ++i) {
15         offsets[i] = i;
16     }
17     for (auto i = 0; i < nb; ++i) {
18         auto pattern = JA[i];
19         auto row = IA[i];
20         auto mask = A[i];
21
22         std::array<ST, BR*P> temp;
23         for (auto k = 0; k < P; ++k) {
24             for (auto b = 0; b < BR; ++b) {
25                 temp[k*BR + b] = (mask >> offsets[b]) & 1;
26             }
27         }
28
29         auto temp2 = reinterpret_cast<DT*>(&temp[0]);
30         for (auto k = 0; k < P; ++k) {
31             for (auto b = 0; b < BR; ++b) {
32                 C[P*(row+b) + k] += table[pattern*P + k] * (temp2[b*P + k]
    ↪ -1);
33             }
34         }
35     }
36 }

```

Listing 39: A coordinate-like alternative C++ implementation of the transposed sparse panel format adapted for floating point vectors and lookup tables.

To confound things further, we will point out that all three methods presented thus far only work for integer vectors and panels. We are converting our bitmask to a mask of all ones or all zeros and using a bitwise AND to keep or clear copies of our lookup table row. While we cannot perform bit operations on floats (well, we could reinterpret cast our pointers), we actually have a simpler task with floating point numbers. Instead of converting ones and zeros into masks of all ones or all zeros, we

can take advantage of the fact that our floating point fused multiply-add is no more expensive than a floating point add, so instead of masking and adding, we can just do a multiply and add without fussing over how to efficiently convert a single bitmask into an array of individual bitmasks. This shortcut does rely on the not-necessarily-universal assumption that the unsigned integer 1 is equal to the float 2. We illustrate this final version in Listing 39. When building our integer masks, we can use the same data types and have conversions work appropriately. However, we need to pay a little bit of extra attention to the float case, as we use 64-bit values for our masks, thus if we build an array of zeros and ones to be used for scaling, we want to avoid the very expensive integer-to-float conversion function, so we use a reinterpret cast in line 29. For this to work, we must make sure that the width of an element of the temp array is the same as our floating point type. Likewise, we would not want to use a 32-bit mask with a 64-bit float, as our goal is to directly multiply the values, so we want them to be the same types. The ST definition of line 4 should match the size of the floating point type in line 3 for this reinterpret cast to be successful.

4.2 llvm-mca Analysis of Sparse Transposed Panel Format

As we have done so many times in this document, we will present our llvm-mca analyses for each of these four transposed sparse panel formats. In no particular order of importance, Table 4.1 contains our first transposed CSC-like version for integers, Table 4.2 contains our alternative format, Table 4.3 contains our coordinate-like format for integers, and Table 4.4 contains the coordinate-like format modified for floating point operands. For each, we consider block sizes of 1, 8, 32, and 64. For the larger block sizes, the panel operand cases are omitted, as neither compiler produces code that handles the entire unrolled block.

For each format, the single-element block case is, understandably poor. For many cases, our panel-width multiplication, with Clang, is on par with that of other formats. However, the standard CSC-like (i.e., not coordinate-like) formats come with the usual costs we have seen with these types of formats. For each pattern, we have

Block size	Vector width	Clang 10.0.0				GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1	1x32-bit int.	4	6	5.5		1	2	18	
	8x32-bit int.	32	8.5	0.91		8	12	4.5	
	1x64-bit int.	4	6	5.5		1	2	18	
	4x64-bit int.	16	8.5	1.81		4	5.2	6.75	
8	1x32-bit int.	8	2.2	2.63	33	8	5	3.38	11
	8x32-bit int.	64	16	0.47	72	-	-	-	
	1x64-bit int.	8	3.3	2.75	30	8	10	2.88	10
	4x64-bit int.	32	24	1.28	38	32	36	1.88	23
32	1x32-bit int.	32	8.2	1.03	14	32	5.8	0.75	11
	1x64-bit int.	32	51	1.97	33	32	11.8	1	17
64	1x32-bit int.	64	52	0.97	25	64	10.8	0.52	51
	1x64-bit int.	64	96	1.64	31	64	23.5	0.69	11

Table 4.1: llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the sparse transposed panel format presented in Listing 36 with blocks of 1, 8, 32, and 64 rows. The cost, in cycles, noted in the “other” column is a cost present for each unique pattern.

Block size	Vector width	Clang 10.0.0				GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1	1x32-bit int.	4	10	6.75		1	2.5	22	
	8x32-bit int.	16	5.5	1.5		8	5	3.38	
	1x64-bit int.	4	10	6.75		1	2.5	22	
	4x64-bit int.	8	5.5	3		4	3.3	5.25	
8	1x32-bit int.	8	2.5	3		8	10.5	3.5	
	8x32-bit int.	64	25	0.61		8	8.5	2.88	78
	1x64-bit int.	8	3.8	3.13		8	10.5	3.5	
	4x64-bit int.	32	62	2.78		8	8.5	2.88	46
32	1x32-bit int.	32	21	1.53		32	6.7	0.84	
	1x64-bit int.	32	25.3	1.63		32	12.7	1.22	
64	1x32-bit int.	64	36	0.77		32	6.7	0.84	
	1x64-bit int.	64	52	1.05		32	12.7	1.22	

Table 4.2: llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the modified coordinate format presented in Listing 37 with blocks of 1, 8, 32, and 64 rows. The cost, in cycles, noted in the “other” column is a cost present for each complete block.

Block size	Vector width	Clang 10.0.0				GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other
1	1x32-bit int.	4	6	5.5	20	1	1.8	18	10
	8x32-bit int.	32	8.5	0.91	28	8	8.3	3.25	15
	1x64-bit int.	4	6	5.5	20	1	1.8	18	10
	4x64-bit int.	16	8.5	1.81	28	4	5	5.5	10
8	1x32-bit int.	8	3	3.63	36	8	19	5.25	11
	8x32-bit int.	64	90	1.53	26	-	-	-	
	1x64-bit int.	8	3.7	2.88	31	8	10.5	3.63	10
	4x64-bit int.	32	24	1.16	27	32	48	2.34	19
32	1x32-bit int.	32	18	1.34	20	32	20	1.34	11
	1x64-bit int.	32	19	1.34	22	32	11.8	1.06	11
64	1x32-bit int.	64	32	0.8	32	64	40	0.98	14
	1x64-bit int.	64	70	1.27	36	64	23.3	0.72	14

Table 4.3: llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the alternate sparse transposed panel format presented in Listing 38 with blocks of 1, 8, 32, and 64 rows. The cost, in cycles, noted in the “other” column is a cost present for each unique pattern. Missing values in the first three columns denote that no clear main loop was found.

Block size	Vector width	Clang 10.0.0			GCC 10.1			
		ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$	other	ops.	$\frac{1}{\text{throughput}}$	$\frac{\text{cycles}}{\text{op.}}$
1	1xfloat	4	10	6.25	1	2.5	19	
	8xfloat	16	5	1.56	8	3.2	3.13	
	1xdouble	4	10	6.25	1	2.5	19	
	4xdouble	8	5	3.13	4	3	5.5	
8	1xfloat	8	3	3.75	8	13	3.88	
	8xfloat	64	20	0.61	8	8.5	2.63	11
	1xdouble	8	3.2	3	8	13	3.88	
	4xdouble	32	29	1.66	8	8.5	2.63	11
32	1xfloat	32	12	1	32	20	1.38	
	1xdouble	32	16	1.06	32	9.3	0.94	
64	1xfloat	64	24	0.66	64	40	1.02	
	1xdouble	64	32	0.73	64	18	0.7	

Table 4.4: llvm-mca analysis for sparse matrix-vector multiplication using the Method of the Four Russians with the modified coordinate format presented in Listing 39 with blocks of 1, 8, 32, and 64 rows. The cost, in cycles, noted in the “other” column is a cost present for each complete block.

some amount of work performed in our initial setup, broadcasting our lookup table entries into registers that we then use later. With Clang, this is a considerable cost in many cases, as GCC performs much less data movement for most of its cases, especially those at larger block sizes. Not to disappoint, we also see inconsistent performance among our different formats and compilers. For example, multiplying panels of eight floats or eight 32-bit ints with a block a block size of 8 is done very efficiently by Clang in three of the four formats. In the alternative CSC-like format it fares quite poorly in this case. However, nearly all formats handle our 64-row blocks in less than one cycle-per-operation. Particularly with GCC in our first CSC-like format and in the integer coordinate-like format, we also see very low reciprocal throughputs in addition to good cycles-per-operation. This low reciprocal throughput is especially beneficial in the coordinate-like formats, as if our processor serves us well, it should predict to continue to subsequent iterations and should be able to pipeline multiple iterations, provided we have available registers.

Additionally, GCC’s very low cycles-per-operation in the CSC-like format in Table 4.1 for the 64-row block size may yield very good performance, despite the relatively high extra cost per pattern if we have few patterns. Further, in Table 4.4 for 64 doubles and 32 doubles in the coordinate-like format we see GCC have exceptionally low reciprocal throughputs and cycles-per-operation. Curiously, GCC handles the double cases better than it handles the float cases, suggesting that there is some alignment issue going on again between 32-bit and 64-bit operands in functions that Clang is not concerned with.

Ultimately, we see vector cases that approach the 0.33, 0.45, and 0.66 cycles-per-entry costs we saw with dense matrix-vector multiplication, all while faring better in the panel cases. While our reciprocal throughputs are higher (due to the overhead of index data not present in dense matrices), this transposed format is the closest we have come to this performance.

4.3 Further Sparse Method of the Four Russians Considerations

Ultimately, the good on-paper performance for the larger block sizes of these transposed formats comes with some concerns. First, these are fairly large block sizes. If we compare Clang’s 64-row float to its 1-row float, while we take approximately 42 cycles for the 64 entry version, we can perform five 4-times unrolled 1-row applications in about the same time, if scheduling permits. This begs the question about how full these blocks are and how close we expect duplicate patterns within rows of our matrices. While we expect and, indeed, require duplicate patterns within panels to successfully use the Method of the Four Russians, requiring that duplicate patterns be adjacent is a stricter requirement than just simple row- and/or column-blocking.

However, the fast arithmetic, particularly compared to other sparse formats in the vector cases suggests that purposefully keeping our number of patterns low (i.e., small values of k), while not just being a natural choice with sparse matrices, may, in some cases, paradoxically allow us to perform these vector applications faster.

At the very least, if we do have neighboring duplicated patterns, we generally outperform the standard sparse panel multiplication formats that cannot take advantage of vectorized code for the vector-operand case. With our standard sparse format, we saw reciprocal throughputs of two-per-operation with greater than five cycles-per-operation with Clang (for four operations). Thus, if we consider GCC's integer coordinate-like code for 32 or 64 rows operating on 32 rows in about 27 cycles with a reciprocal throughput of 6.7 cycles, we are breaking even if we have four non-zeros in our block of 32. Through our benchmarks, we see that this holds true, at large enough sizes, as 5% density is just over 3 entries in a block size of 64 entries.

Considering most of our workflows that would involve the Method of the Four Russians and repeated sparse matrix-vector products would allow us some amount of preconditioning or permutation of our matrix, we could feasibly envision some heuristic sort to group similar rows or adjacent values together. Coupled with choosing a small parameter of the Method of the Four Russians, this could easily yield circumstances where these large block sizes can be applied. This is, of course, a data-centric concern, as these parameters are highly sensitive to dimension and distribution of elements. What we see is a format that can yield efficient sparse matrix-vector applies and potentially do so better than other sparse matrix formats which exceeds our goal of obtaining a Method of the Four Russians speedup in sparse computations.

4.4 Benchmarks

In this section, we test the efficiency of our proposed format in a synthetic benchmark in efforts to identify if its potential can be realized. For this test, we will focus only on Clang, as it had the most ubiquitous use of vectorization, what we rely on in our llvm-mca analysis for high potential performance.

Our synthetic benchmarks involve multiplication against randomly-generated matrices with a specified density. This is not, therefore, a test of the highest possible efficiency of these formats, as a random case tends to be unfavorable to the Method

of the Four Russians, but whether or not there are situations we can obtain practical speed-ups and if any patterns in these situations can be determined.

4.4.1 Experimental Setup

For this experiment, we use the Clang 10.0.0. compiler with compiler flags of `-Ofast -march=skylake -mtune=skylake -std=c++14`, with our code running on an Intel i6770k processor with 32GB of memory, although we limit ourselves to relatively small matrices, in keeping with our analytical method of assuming we have cache-friendly operands.

For a range of comparison points, hoping to mirror our `llvm-mca` analyses, we test against coordinate format, compressed sparse row, compressed sparse column, a “standard” sparse Method of the Four Russians, and our coordinate-like “transposed” format introduced in this chapter. For simplicity, we use generic parameters of the Method of the Four Russians of 2, 4, and 8. We use a block size of 64 for our transposed format.

We test between 25 and 100,000 consecutive trials, guarded by a volatile-declared iterator, depending on the number of entries in order to gather consistent times with our machine’s low clock resolution of 0.004 seconds.

We generate our matrices with identically $[m * n * \text{density}]$ non-zero elements chosen as uniformly as the provided standard library allows. For comparison purposes, times are reported in nanoseconds per coefficient of the matrix. To use the Method of the Four Russians, all coefficients are 1. Panel times include time spent building lookup tables in the Method of the Four Russians.

In Table 4.5, we summarize these results. In general, we see a couple results that differ from our theoretical analysis. Namely, we see about a 1-to-2 ratio between coordinate format and CSR and CSR and CSC. From our `llvm-analysis`, we would have expected CSC and COO to perform better than CSR for this vector case. The great performance relative to the traditional formats supports our `llvm-mca` analyses that CSC, as presented, is well-suited for sparse matrix-vector multiplication. Second, we

Format	1024 × 1024	8192 × 8192	8192 × 8192	512 × 512	16384 × 16384
	5%	5%	1%	25%	5%
coo	1.144	1.454	1.311	1.221	1.460
csr	0.595	0.727	0.715	0.488	0.780
csc	0.343	0.393	0.358	0.305	0.393
csp, k=2	1.991	1.118	1.728	1.099	1.204
tcsp, k=2	2.060	0.262	1.252	0.793	0.131
csp, k=4	1.320	1.001	1.371	0.732	1.055
tcsp, k=4	1.549	0.238	0.894	0.793	0.143
csp, k=8	1.053	0.870	1.311	0.488	0.900
tcsp, k=8	2.418	0.465	0.954	2.319	0.340

Table 4.5: Running times, in nanoseconds-per-entry, for the specified sparse formats and given matrix dimensions and densities. coo is coordinate format, csr is compressed sparse row, csc is compressed sparse column, csp is a sparse panel format, and tcsp is our coordinate-like transposed format. csp and tcsp specify the parameter of the Method of the Four Russians used. All vectors used are vectors of 64-bit integers.

see that the “standard” sparse panel and and the transposed format generally trade off in which performs better, despite analysis showing that the latter should have faster arithmetic.

Our transposed panel format improves relative to the other formats as our matrix sizes increase and density remains constant or increases. Consistently, when we have larger dimensions and relatively high density (5%), our transposed format outperforms the others. At smaller dimensions or at lower densities, CSC overtakes this format, likely due to the overhead of the Method of the Four Russians. Some preliminary traces suggest that far more time is spent generating lookup tables than applying them.

For the most part, the standard formats show relatively constant performance across dimension and density. Our two formats that use the Method of the Four Russians, as our analysis showed, are much more dependent on these factors and the selection of k . For the most part, we see that $k = 4$, provided our matrix is large enough works well for our 5% dense 8192 × 8192 and 16384 × 16384 cases, showing roughly 65% and nearly 175% speedup over CSC in these cases, respectively. Interestingly, we

see that we do slightly better when $k = 2$ in this larger case.

These benchmarks only serve to illustrate the potential of this format for particular matrices. On one hand, these random matrices are likely poor candidates for the Method of the Four Russians, as the method works best with repeated entries. Non-random matrices are likely to contain specific repeated patterns. On the other hand, our our past theoretical analysis suggested that there are certain dimension and density combinations for which the Method of the Four Russians could be advantageous, even in the random case. Further, optimizations such as $\binom{t}{k}$ tables or custom lookup table generation code are possible optimizations for the Method of the Four Russians. Second, with our use of 64-bit operations, our traditional formats slightly suffer. Additionally, both the “traditional” sparse format in applying its lookup tables and both Method of the Four Russians formats do not make use of vectorized instructions in table generation for the vector case, as row operations are used when building lookup tables. Finally, we limited ourselves to just a couple parameters of the Method of the Four Russians. Given the sensitivity to k that we see, especially at $k = 8$, it is possible that another choice of k could yield better results than those we see here.

Ultimately, we see that despite the potential for further optimizations, there are some cases where we can expect improved performance over general formats, even with arbitrarily-chosen parameters.

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Summary

In this thesis, we presented an overview of the current and historical work regarding sparse matrix-vector multiplication, particularly with regards to small finite fields. Specifically, we focused on optimizations and formats that lend themselves well to these cases.

The task of sparse matrix-vector multiplication is recognized as one of the seven most important kernels in computation [6]. Thus, improving its performance and obtaining performance that scales to advanced hardware is critically important. In this thesis, we have thoroughly analyzed common sparse matrix formats and how efficiently their operations can be performed in isolation, showing that we can apply the Method of the Four Russians to obtain a speedup that can be realized without slower arithmetic and data access patterns.

From this, we launched into a discussion specifically regarding the Method of the Four Russians and presented initial, novel approaches at sparse variants to this algorithm. We modeled the arithmetic costs of standard dense and sparse Methods of the Four Russians consistently with previous work. The sparse case proved more difficult to optimize, due to the interdependence between number of entries within a panel and the number of panels. From this, we recounted some potential, possibly-practical considerations that should be regarded when transitioning from theory to practice, as not all operations are equal and keeping memory access and cache performance in mind are crucial.

To illustrate this point, we spent the majority of this thesis in a novel, in-depth examination of what compilers actually produce for our sparse matrix multiplication

kernels. Through this analysis, we could see why certain formats may perform better than others, even discounting factors such as cache performance. Further, we saw that, regardless of the packing efficiency or potential logarithmic or pseudo-logarithmic speedup the Method of the Four Russians may offer, the random accesses, performed when accessing our table rob us of other practical speedups achievable with vectorization on modern hardware.

To counter this, we proposed a new sparse matrix format that is directly compatible with the Method of the Four Russians. Using our same analysis technique, we identified the circumstances necessary to obtain the desired performance in matrix-vector products. Specifically, we have identified that a direct, panel-based sparse Method of the Four Russians is competitive with classic sparse formats in the general case, with the potential to perform much better in specific matrices. Further, we have shown that, for suitably large matrices, our transposed format can compute matrix-vector products faster than classical formats, with similar matrix-specific optimizations available.

We believe that both this process of code-driven analysis and applying the Method of the Four Russians to sparse environments enables us to advance the performance of sparse matrix-vector multiplication and thus tackle large scale linear algebra problems over these small fields.

5.2 Future Work

The intersection of achieving general vectorized multiplication kernels and the additional optimizations available to particular matrices (i.e., not general cases) offer many potential areas of future work. The eventual goal is a generic format such as our $\binom{t}{k}$ format that offers at least acceptable performance in most cases, or at least if certain easily-verifiable conditions are met. Ideally, this general format could be tuned for performance if features such as data distribution and density are known. The requirement for expensive analysis or pre-processing to select parameters for the Method of the Four Russians is a steep cost. Further exploring the practical trade-offs between purposefully increasing the density and decreasing the number of patterns

within a sparse matrix in order to make better advantage of faster arithmetic is one avenue.

Additionally, more investigation into the effects, benefits, or possible heuristics of row and/or column reordering when processing our matrix can be performed. Further, it is possible that some form of hybrid format or factorization of a matrix into matrices using different block sizes may be of some benefit.

Particularly, if we find that there are certain densities or sizes that can be done well, it may make a certain amount of sense add entries to our matrix and compute the difference of matrix products, rather than the typical sum of products if we factor a matrix into component pieces and process these pieces separately.

Further, extending our analysis method to encompass all arithmetic that may occur within our problem is another item of future work. Beyond applying panels to lookup tables, we also want to consider the arithmetic efficiency of generating our lookup tables. Given the likelihood of producing smaller tables, we have additional optimization possibilities such as building multiple tables in parallel. Additionally, we would like to extend this analysis process to normalization and partial normalization procedures that may occur if using operating over most finite fields beyond GF2 and GF3. Ideally, automating the process and identifying major loops should be feasible, even though the work in this thesis was done by hand. If possible, generating test suites that use these tools to rapidly find optimal block sizes, test code for deployment on different hardware configurations, or create a recommendation engine for fast formats are all possible extensions of this data-agnostic work. Ideally, this would be done in conjunction with some form of auto-tuning to pair cache-sized blocks with efficient matrix formats for the available space.

BIBLIOGRAPHY

- [1] Martin Albrecht and Gregory Bard. *The M4RI Library – Version 20140914*. The M4RI Team, 2014. <https://bitbucket.org/malb/m4ri/src/master/>.
- [2] Martin Albrecht, Gregory Bard, and William Hart. Algorithm 898: Efficient Multiplication of Dense Matrices over GF(2). *ACM Trans. Math. Softw.*, 37(1):9:1–9:14, January 2010. <http://doi.acm.org/10.1145/1644001.1644010>.
- [3] Heinecke Alexander and Trinitis Carsten. Cache-oblivious matrix algorithms in the age of multicores and many cores. *Concurrency and Computation: Practice and Experience*, 27(9):2215–2234. <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.2974>.
- [4] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics—Doklady*, 11(5):1209–1210, 1970. <http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=dan&paperid=35675>.
- [5] Warren Armstrong and Alistair P. Rendell. Runtime sparse matrix format selection. *Procedia Computer Science*, 1(1):135 – 144, 2010. <https://doi.org/10.1016/j.procs.2010.04.016>.
- [6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiawicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A View of the Parallel Computing Landscape. *Commun. ACM*, 52(10):56–67, October 2009. <https://doi.org/10.1145/1562764.1562783>.
- [7] Gregory V Bard. Accelerating Cryptanalysis with the Method of Four Russians. 2006. <https://eprint.iacr.org/2006/251>.
- [8] Tomas J. Boothby and Robert W. Bradshaw. Bitslicing and the Method of Four Russians Over Larger Finite Fields. *CoRR*, abs/0901.1413, 2009. <http://arxiv.org/abs/0901.1413>.
- [9] Brice Boyer, Jean-Guillaume Dumas, and Pascal Giorgi. Exact Sparse Matrix-Vector Multiplication on GPU’s and Multicore Architectures. In Marc Moreno Maza & Jean-Louis Roch, editor, *PASCO’10: 4th International Symposium on Parallel Symbolic Computation*, pages 80–88, Grenoble, France, July 2010. Association for Computing Machinery. <https://doi.org/10.1145/1837210.1837224>.

- [10] Li Chen, Wayne Eberly, Erich Kaltofen, B. David Saunders, William J. Turner, and Gilles Villard. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and its Applications*, 343–344:119–146, 2002. [https://doi.org/10.1016/S0024-3795\(01\)00472-4](https://doi.org/10.1016/S0024-3795(01)00472-4).
- [11] Don Coppersmith. Solving Homogeneous Linear Equations Over $GF(2)$ via Block Wiedemann Algorithm. *Mathematics of Computation*, 62(205):333–350, 1994. <https://dl.acm.org/doi/10.2307/2153413>.
- [12] Robert M Corless, Gaston H Gonnet, David EG Hare, David J Jeffrey, and Donald E Knuth. On the LambertW function. *Advances in Computational mathematics*, 5(1):329–359, 1996. <https://doi.org/10.1007/BF02124750>.
- [13] C. Ding and J. Yuan. A family of skew Hadamard difference set. *J. Comb. Theory, Ser. A*, 113:1526–1535, 2006. <https://dl.acm.org/doi/10.1016/j.jcta.2005.10.006>.
- [14] Jean-Guillaume Dumas, Philippe Elbaz-Vincent, Pascal Giorgi, and Anna Urbánska. Parallel Computation of the Rank of Large Sparse Matrices from Algebraic K-theory. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, PASCOCO '07*, pages 43–52, New York, NY, USA, 2007. ACM. <http://doi.acm.org/10.1145/1278177.1278186>.
- [15] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. Dense Linear Algebra over Word-Size Prime Fields: The FFLAS and FFPACK Packages. *ACM Trans. Math. Softw.*, 35(3):19:1–19:42, October 2008. <http://doi.acm.org/10.1145/1391989.1391992>.
- [16] I. R. eguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *2012 Innovative Parallel Computing (InPar)*, pages 1–12, 2012. <https://doi.org/10.1109/InPar.2012.6339602>.
- [17] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, KDD'96*, pages 226–231. AAAI Press, 1996. <https://dl.acm.org/doi/10.5555/3001460.3001507>.
- [18] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering*, 93:110, 2011. https://www.agner.org/optimize/instruction_tables.pdf.
- [19] Hormozd Gahvari, Mark Hoemmen, James Demmel, and Katherine Yelick. Benchmarking Sparse Matrix-Vector Multiply in Five Minutes. In *SPEC*

- Benchmark Workshop 2007*, 2007. <https://bebop.cs.berkeley.edu/pubs/gahvari2007-spmvbench-spec.pdf>.
- [20] Michael Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference, DAC '08*, pages 2–6, New York, NY, USA, 2008. ACM. <http://doi.acm.org/10.1145/1391469.1391473>.
- [21] Pascal Giorgi and Bastien Vialla. Generating Optimized Sparse Matrix Vector Product over Finite Fields. In Hoon Hong and Chee Yap, editors, *Mathematical Software – ICMS 2014*, pages 685–690, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-44199-2_102.
- [22] Georgios I. Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing*, 50:36–77, 2008. <https://doi.org/10.1007/s11227-008-0251-8>.
- [23] Ping Guo, Wansong Chen, and Po Chen. A Performance Modeling and Optimization Analysis Tool for Sparse Matrix-Vector Multiplication on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25:1112–1123, 2014. <https://dl.acm.org/doi/abs/10.1109/TPDS.2013.123>.
- [24] El Zein Ahmed H. and Rendell Alistair P. Generating optimal CUDA sparse matrix–vector product implementations for evolving GPU hardware. *Concurrency and Computation: Practice and Experience*, 24(1):3–13. <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1732>.
- [25] Erich Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. In Gérard Cohen, Teo Mora, and Oscar Moreno, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 195–212, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. <https://dl.acm.org/doi/abs/10.2307/2153451>.
- [26] Matthew A. Lambert and B. David Saunders. Compiler auto-vectorization of matrix multiplication modulo small primes. In *Proceedings of the International Workshop on Parallel Symbolic Computation, PASCO 2017*, pages 7:1–7:10, New York, NY, USA, 2017. ACM. <http://doi.acm.org/10.1145/3115936.3115943>.
- [27] Daniel Langr and Pavel Tvrdik. Evaluation criteria for sparse matrix storage formats. *IEEE Trans. Parallel Distrib. Syst.*, 27(2):428–440, February 2016. <http://dx.doi.org/10.1109/TPDS.2015.2401575>.
- [28] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. SMAT: An Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication. *SIGPLAN Not.*, 48(6):117–126, June 2013. <http://doi.acm.org/10.1145/2499370.2462181>.

- [29] Weifeng Liu and Brian Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 339–350, New York, NY, USA, 2015. ACM. <http://doi.acm.org/10.1145/2751205.2751209>.
- [30] Weifeng Liu and Brian Vinter. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Comput.*, 49(C):179–193, November 2015. <https://doi.org/10.1016/j.parco.2015.04.004>.
- [31] K. Patrick Lorton and David S. Wise. Analyzing Block Locality in Morton-order and Morton-hybrid Matrices. *SIGARCH Comput. Archit. News*, 35(4):6–12, September 2007. <http://doi.acm.org/10.1145/1327312.1327315>.
- [32] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Appl. Algebra Eng., Commun. Comput.*, 18(3):297–311, May 2007. <http://dx.doi.org/10.1007/s00200-007-0038-9>.
- [33] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.5427>.
- [34] Richard Parker. Meataxe64: High Performance Linear Algebra over Finite Fields. In *Proceedings of the International Workshop on Parallel Symbolic Computation*, PASCO 2017, pages 11:1–11:3, New York, NY, USA, 2017. ACM. <http://doi.acm.org/10.1145/3115936.3115947>.
- [35] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM. <http://doi.acm.org/10.1145/331532.331562>.
- [36] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2, 1994. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.3853>.
- [37] W. T. Tang, W. J. Tan, R. S. M. Goh, S. J. Turner, and W. Wong. A Family of Bit-Representation-Optimized Formats for Fast Sparse Matrix-Vector Multiplication on the GPU. *IEEE Transactions on Parallel and Distributed Systems*, 26(9):2373–2385, 2015. <https://doi.org/10.1109/TPDS.2014.2357437>.
- [38] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. Res. Dev.*, 41(6):711–726, November 1997. <http://dx.doi.org/10.1147/rd.416.0711>.

- [39] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005. <https://doi.org/10.1088/1742-6596/16/1/071>.
- [40] Richard W. Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proceedings of the First International Conference on High Performance Computing and Communications, HPCC'05*, pages 807–816, Berlin, Heidelberg, 2005. Springer-Verlag. http://dx.doi.org/10.1007/11557654_91.
- [41] G. Weng, W. Qiu, Z. Wang, and Q. Xiang. Pseudo-Paley graphs and skew Hadamard difference sets from commutative semifields. *Designs, Codes and Cryptography*, 44(1):49–62, 2007. <https://doi.org/10.1007/s10623-007-9057-6>.
- [42] D H Wiedemann. Solving Sparse Linear Equations over Finite Fields. *IEEE Trans. Inf. Theor.*, 32(1):54–62, January 1986. <http://dx.doi.org/10.1109/TIT.1986.1057137>.
- [43] Bartosz D. Wozniak, Freddie D. Witherden, Francis P. Russell, Peter E. Vincent, and Paul H.J. Kelly. GiMMiK—generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics. *Computer Physics Communications*, 202:12 – 22, 2016. <https://doi.org/10.1016/j.cpc.2015.12.012>.
- [44] Bryan Youse. *High performance exact linear algebra*. PhD thesis, University of Delaware, 2015. <https://udspace.udel.edu/handle/19716/17203>.
- [45] A. N. Yzelman and Rob H. Bisseling. Cache-Oblivious Sparse Matrix–Vector Multiplication by Using Sparse Matrix Partitioning Methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009. <https://doi.org/10.1137/080733243>.
- [46] Albert-Jan N. Yzelman and Dirk Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):116–125, January 2014. <https://doi.org/10.1109/TPDS.2013.31>.
- [47] Jilin Zhang, Jian Wan, Fangfang Li, Jie Mao, Li Zhuang, Junfeng Yuan, Enyi Liu, and Zhuoer Yu. Efficient sparse matrix–vector multiplication using cache oblivious extension quadtree storage format. *Future Generation Computer Systems*, 54:490–500, 2016. <https://doi.org/10.1016/j.future.2015.03.005>.

Appendix

SOURCE CODE AND COMPILER OUTPUT

Source .cpp files, compiler assembly output, isolated loops, and benchmark generation, at time of writing, are available on the University of Delaware EECIS research servers (hoek and catan) at </usa/lambert/sirena/thesis>.