ON IMPLEMENTING SPARSE MATRIX-VECTOR MULTIPLICATION ON INTEL PLATFORM

BY

MOHAMMAD ALMASRI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Advisers:

Adjunct Associate Professor Walid Abu-Sufah
Professor Wen-Mei W. Hwu

ABSTRACT

Sparse matrix-vector multiplication, SpMV, can be a performance bottleneck in iterative solvers and algebraic eigenvalue problems. In this thesis, we present our sparse matrix compressed chunk storage format (CCF) and SpMV CCF kernel that realizes high performance on Intel Xeon multicore and Phi processors for unstructured matrices. CCF kernel exploits the properties of CCF to enhance load balancing and SIMD efficiency. Moreover, we present the CCF auto-tuner that selects the most effective parameters and the SpMV kernel to achieve the highest possible performance that CCF can attain on a target architecture. Using 151 unstructured matrices from 38 application areas, we compare the performance of the CCF kernel to that of MKL 2018u1 SpMV CSR, MKL 2018u2 Inspector-executor SpMV CSR, and Compressed Vectorization-oriented sparse Row (CVR) SpMV. We execute the kernels on a dual 24-core Skylake Xeon Platinum 8160 and a 68-core KNL Xeon Phi 7250. Executing on the dual 24-core Skylake Xeon Platinum 8160, and compared to MKL SpMV CSR, our kernel achieves superior execution throughputs for 135 matrices (89%) with an average speed improvement of 2.3x and maximum speed improvement of 27.5x. Our kernel outperforms MKL Inspector-executor SpMV CSR for 109 matrices (73%) with an average speed improvement of 1.5x and maximum speed improvement of 3.0x. Moreover, SpMV CCF outperforms SpMV CVR for 81% of the matrices with an average speed improvement of 1.8x and maximum speed improvement of 4.2x. Executing on the 68-core KNL Xeon Phi 7250, CCF achieves high average and maximum speed improvements compared to the other three kernels but for slightly smaller percentages of matrices. Lastly, we show that auto-tuning CCF parameters improves the performance for more than 50 matrices compared to the default CCF on Skylake and KNL with an average speed improvement of 1.2x.

To Mama, Baba, Alaa, Huda, Haneen, Ahmad, and Issa for their unconditional love and support

ACKNOWLEDGMENTS

All praise is due to God.

I would like to thank my parents for their endless love and support, for their dedication to help me overcome all the obstacles and succeed, and for being such great role models.

I would like to thank my wife Alaa and my daughter Huda for their tremendous support and unconditional love during the past two years. The sacrifices they have made and the care they have provided helped me become who I am today.

I would like to thank my adviser, Professor Walid Abu-Sufah, for his great mentorship and support. He has always motivated my work and been patient with me. His wisdom will keep inspiring me in my professional career and personal life.

Intel Corporation supported this work through Grant No 552147-239012-191100. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), Stampede 2 at Texas Advanced Computing Center through allocation which is supported by National Science Foundation grant number ACI-1548562.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CCF	Compressed Chunk Format
CSR	Compressed Sparse Row
DRAM	Dynamic Random-Access Memory
I-e	Inspector-executor
MKL	Math Kernel Library
OpenMP	Open Multi-Processing
SIMD	Single-Instruction Multiple-Data
SpMV	Sparse Matrix-Vector Multiplication

CHAPTER 1

INTRODUCTION

1.1 Sparse Matrix-Vector Multiply Overview

Sparse matrix-vector multiplication (SpMV) is a fundamental performance bottleneck in solving sparse linear systems and eigenvalue problems. In SpMV, the operation y=A*x+y is performed, where A is a sparse matrix and x, y are dense vectors. Sparse matrices use special data structures that store only the nonzero elements and hence eliminate unnecessary storage and computation. This leads to low computational intensity and poor performance. Moreover, the emergence of modern processors with high thread count and wide vector units has introduced new performance bottlenecks that any new storage format must address and mitigate to achieve significant improvement in performance. These performance bottlenecks are: (a) low utilization of vector units (low SIMD efficiency), (b) load imbalance, (c) irregular memory access pattern, and (d) low utilization of thread resources.

1.2 Common Storage Formats

Compressed Sparse Row (CSR) is a general-purpose storage format that is commonly used due to its compact memory requirements. Parallel and vectorized CSR SpMV kernels divide the rows evenly among threads and each thread processes its assigned rows in row-major order. Each row is processed by a single vector unit. Figure 1.1 illustrates how a row is processed by an 8-lane vector unit. A CSR kernel can suffer from low SIMD efficiency (figure 1.2) and load imbalance (figure 1.3) when it is parallelized and vectorized. Low SIMD efficiency is caused by processing a row with the number of nonzero elements less than the SIMD width (SIMDW, the number of avail-

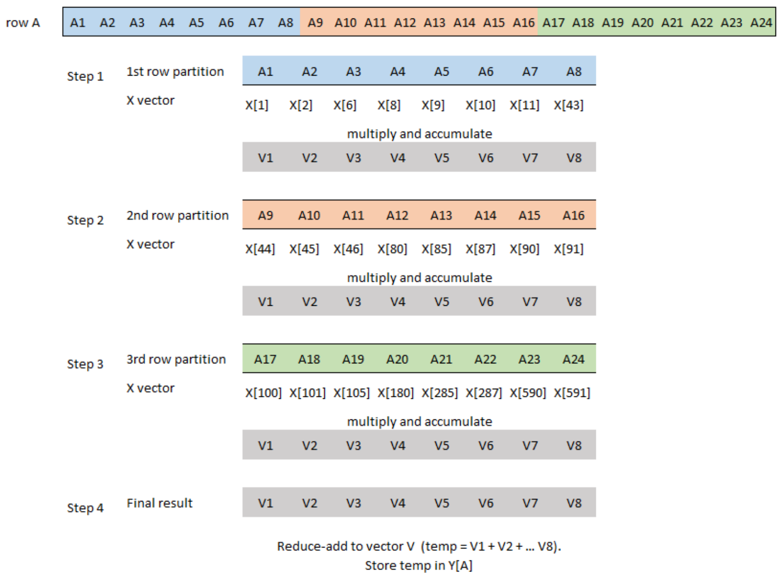

row A
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16 A17 A18 A19 A20 A21 A22 A23 A24
Step 1
1st row partition
A1 A2 A3 A4 A5 A6 A7 A8
X vector
X[1] X[2] X[6] X[8] X[9] X[10] X[11] X[43]
multiply and accumulate
V1 V2 V3 V4 V5 V6 V7 V8
Step 2
2nd row partition
A9 A10 A11 A12 A13 A14 A15 A16
X vector
X[44] X[45] X[46] X[80] X[85] X[87] X[90] X[91]
multiply and accumulate
V1 V2 V3 V4 V5 V6 V7 V8
Step 3
3rd row partition
A17 A18 A19 A20 A21 A22 A23 A24
X vector
X[100] X[101] X[105] X[180] X[285] X[287] X[590] X[591]
multiply and accumulate
V1 V2 V3 V4 V5 V6 V7 V8
Step 4
Final result
V1 V2 V3 V4 V5 V6 V7 V8
Reduce-add to vector V (temp = V1 + V2 + ... V8).
Store temp in Y[A]

Figure 1.1: SpMV CSR: processing of a single row. SIMDW=8.

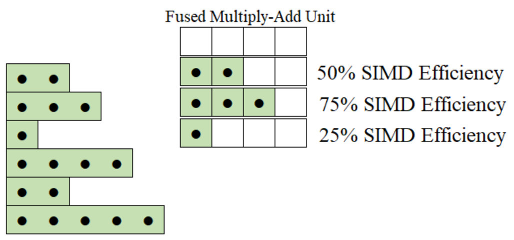


Figure 1.2: CSR bottlenecks: SIMD efficiency.

able lanes in the vector unit). Dividing rows evenly between threads can lead to load imbalance when the number of nonzero elements per row (nnzr) is irregular across rows. In such a case, threads will have different numbers of nonzero elements to process.

The ELLPACK [1] (or ELL) format is particularly well suited to vector architectures. ELL converts an H x W matrix with a maximum nnzr of M into a dense matrix of H x M by padding zeros to shorter rows. ELL uses transposition and zero-padding to improve SIMD efficiency. A vectorized ELL kernel loads elements from sequential rows, performs vector fused-multiply add, and accumulates to a temporary vector. With transposition (or column-major ordering), ELL eliminates the need for the reduction operation (needed by a CSR kernel at the end of each row processing, see figure 1.1) since each entry of the temporary vector has the final value for a different y element (see

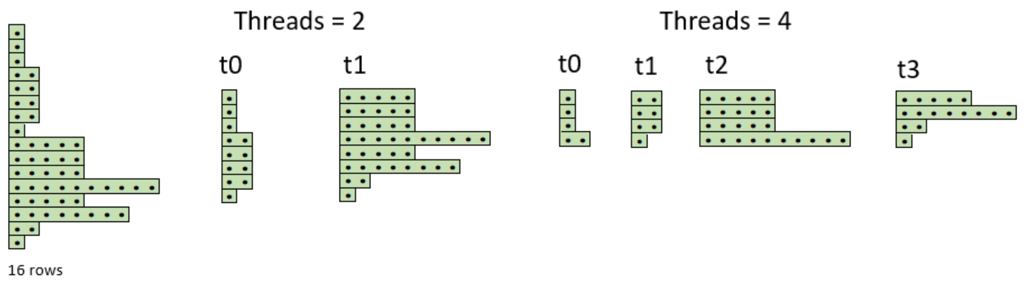


Figure 1.3: CSR bottlenecks: load imbalance. In the case of two threads, t1 has more nonzero elements to deal with than t0. In the case of four threads, t2 and t3 have more nonzero elements than t0 and t1.

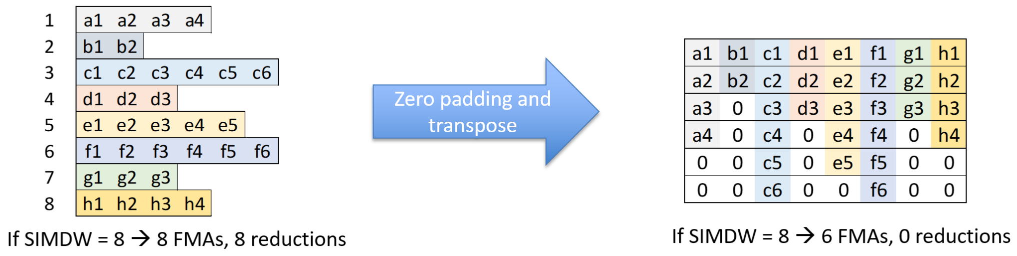


Figure 1.4: Transforming a matrix into ELLPACK format using zero-padding and transposition.

figure 1.4). The fundamental challenge in ELL is the excessive zero-padding when a matrix has rows with irregular nnzr or when few rows are very long. Thus, many storage formats such as [2, 3] were developed trying to process the matrix in a transposed form with minimal zero-padding. Nevertheless, these formats have an inescapable zero-padding overhead for unstructured matrices.

1.3 Unstructured Matrices

Performing the SpMV computation on unstructured matrices is a challenging problem especially as large and highly irregular sparse matrices (scale-free matrices) are emerging from many application areas such as data analytics, social networks, and transportation networks [4, 5, 6]. For scientific and engineering unstructured matrices, [2, 3, 7, 8] proposed formats to achieve high performance. Moreover, several formats were proposed to deal with scale-free matrices such as [6, 9]. There is no single storage format that can achieve the best performance for every unstructured matrix.

Xie et al. devised the Compressed Vectorization-oriented sparse Row (CVR) [9] storage format and showed that SpMV CVR achieves good speed improvements for unstructured matrices compared to five kernels: Intel MKL SpMV CSR, Intel MKL SpMV CSR(I) [10], and kernels presented in [8], [7], and [6]. SpMV CVR processes each row in the sparse matrix by a single SIMD lane to improve SIMD efficiency and data locality. SpMV CVR replaces the data overhead of zero-padding by control overhead logic that keeps track of when a row accumulation is completed and the index in the output vector to which it will be written. Moreover, SpMV CVR allows work stealing (in rare cases) between the lanes of vector units to enhance load balancing.

1.4 Compressed Chunk Format (CCF) and CCF Auto-tuner Overview

In 2013, a storage format advisor which takes a sparse matrix as an input and identifies the best storage format to use for executing an auto-tuned SpMV on NVIDIA GPUs was presented in [11]. Using a similar approach, Intel released in 2015 the Math Kernel Library Inspector-executor (MKL I-e) Sparse BLAS Routines. Users call a storage advisor subroutine which stores the matrix in Intel classified data structures. Users then call an auto-tuner subroutine which delivers an optimized SpMV kernel for the matrix stored in the structures delivered by the storage advisor. In the last step, users call the optimized SpMV kernel delivered by the auto-tuner to perform the SpMV computation.

Independently, we have been working to develop an automatic storage format advisor/auto-tuning software to deliver high performance SpMV kernels for Intel Xeon processors including Intel Xeon Phi. As a first step, we designed and developed our sparse matrix compressed chunk storage format (CCF) and its SpMV CCF kernel. CCF storage format and SpMV CCF enhance load balancing and SIMD efficiency for unstructured matrices. CCF divides the matrix into chunks of rows. A chunk can have multiple rows of the same length or a single row. SpMV CCF works as a hybrid kernel that uses ELLPACK to process multi-row chunks and CSR to process single-row chunks. This technique enhances SIMD efficiency and suppresses the need for zero-padding. Load balancing is achieved by dividing the nonzero elements of the matrix evenly between the execution threads and partitioning of very long rows on several threads. CCF has two parameters that can be tuned for a given matrix and a given architecture to achieve higher performance. CCF auto-tuner software automatically tries all the possible combinations of the parameters and provides the optimal parameter values that would generate the highest possible performance that can be achieved by the SpMV CCF kernel on the given architecture.

For performance evaluation, we use 151 unstructured matrices from 38 application areas and two platforms: a dual 24-core Skylake Intel Xeon Platinum 8160 and a 68-core KNL Intel Xeon Phi 7250. We compare the performance of our SpMV CCF kernel with Intel MKL SpMV CSR, Intel MKL Inspector-executor SpMV CSR, and SpMV CVR kernels. Our results show that CCF significantly outperforms MKL CSR kernels and the CVR kernel for a high percentage of matrices on both platforms. We show that auto-tuning CCF improves the performance of CCF by an average speed improvement of 1.2x on both platforms. Lastly, the dual 24-core Skylake Intel Xeon Platinum 8160 is faster than the 68-core KNL Intel Xeon Phi 7250 for executing SpMV.

1.5 Contributions and Organization

In this thesis, we make the following contributions:

- We propose a novel storage format, CCF, that improves the SIMD efficiency and load balancing for unstructured matrices.

- We present our highly optimized SpMV kernel that benefits from the properties of CCF format to enhance SIMD efficiency and load balancing.

- We propose an auto-tuner software for CCF that further improves the performance of CCF.

- We compare the performance of our SpMV CCF kernel with the latest MKL SpMV CSR kernels and the SpMV CVR kernel using 151 unstructured matrices on the latest Intel HPC platforms. We show that SpMV CCF outperforms the other three SpMV kernels for a high percentage of matrices with high average and maximum speed improvements. We also show that the auto-tuned CCF kernel is superior to the other formats on the dual Skylake.

This thesis is organized as follows. In chapter 2, we present related work. In chapter 3, we introduce our sparse matrix compressed chunk storage format (CCF) and our SpMV CCF kernel. In chapter 4, we analyze the performance of CCF. In chapter 5, we compare the performance of our SpMV CCF kernel to that of MKL 2018u1 SpMV CSR, MKL 2018u1 Inspector-executor SpMV CSR, and SpMV CVR. In chapter 6, we present CCF auto-tuner. We present our conclusions in chapter 7.

CHAPTER 2

RELATED WORK

The introduction of multicore, integrated many-core, and graphics processing units (GPU) triggered a substantial amount of research on development and evaluation of SpMV algorithms for such platforms [12, 13, 14, 2, 15, 16].

Matrix blocking is a widely used optimization technique for SpMV on CPUs [12, 13, 15, 17]. This is because matrices of block sub-structures are encountered in important applications [18]. Furthermore, blocking improves the SIMD efficiency [7] and data locality [17, 19] of the SpMV computation and reduces bandwidth requirement [18, 20].

For wide SIMD architectures, the ELLPACK, ELL [1] storage format is an attractive choice for exploiting vector units efficiently. ELLPACK uses zero-padding and transposition to improve SIMD efficiency. However, performing SpMV for unstructured matrices suffers from significant overhead because of the use of excessive zero-padding. Monakov et al. [21] devised the Sliced ELLPACK format that divides the matrix into slices and packs each slice into ELLPACK format to reduce zero-padding. Although Sliced ELLPACK reduces zero-padding significantly compared to ELL, matrices with irregular nonzero elements per row (nnzr) can still suffer from excessive zero-padding within slices.

SELL-C-σ [3] and ELLPACK Sparse Block (ESB) [7] are variants of Sliced ELLPACK. Both formats divide the matrix into slices and pack each slice into ELLPACK format. Both kernels sort rows by the number of nonzero elements per row (nnzr) in descending order within a finite window. Row sorting is performed before creating slices to help increase the nonzero element intensity in slices and reduce the number of padded zeros. The number of rows per slice and the number of rows per window are tunable parameters. ESB replaces zero-padding in a slice with a bit array that marks nonzero element positions inside the slice. Different flavors of load balancers are studied in [7].

Liu and Vinter [8] proposed the Compressed Sparse Row 5 (CSR5) storage format that divides the matrix into tiles with fixed height and width to improve load balancing and SIMD efficiency. Rows in tiles are stored and processed in column-major order. A row can span multiple columns in a tile or multiple tiles. A column can have elements from multiple consecutive rows. CSR5 redesigned a segmented sum algorithm with wider SIMD utilization [22]. Tile height and width are tunable parameters.

Vectorized hybrid COO+CSR (VHCC) [6] is designed to improve the performance of highly unstructured scale-free matrices on the Xeon Phi architecture. VHCC employs a 2D jagged partitioning method, tiling, and efficient prefix sum computations to achieve high performance.

Compressed Vectorization-oriented sparse Row (CVR) [9] processes each row of the matrix using a single lane in the vector unit. Once the row in a SIMD lane has been processed, the next non-empty row in a matrix would be processed. When all rows are added, CVR steals nonzero elements from SIMD lanes that have more elements and supplies them to shorter lanes to improve load balancing. CVR has a structure called record that keeps track of when a row accumulation is completed and the index in the output vector to which it will be written. Moreover, CVR incurs low pre-processing overhead for scale-free matrices.

CHAPTER 3

THE SPARSE MATRIX COMPRESSED CHUNK STORAGE FORMAT

3.1 Mapping a Sparse Matrix into the Compressed Chunk Storage Format, CCF

CCF is designed for use when executing SpMV on multi/many core vector processors and aims at enhancing load balancing and SIMD efficiency. To store a matrix in CCF for a given processor and runtime system, the values of the following parameters are used before kernel execution:

1. The number of nonzero elements in the matrix, NNZ.
2. The number of threads that will be used to perform SpMV, T.
3. Vector units' width in cores, SIMDW.

CCF groups rows of a matrix in the following three-hierarchical collections:

1. A "*set*" is a collection of consecutive rows assigned for processing by a single thread. The NNZ elements of the matrix are divided equally between sets. A set is made of a collection of bins.
2. A "*bin*" is a collection of rows that belong to the same set and have the same number of nonzero elements per row, nnzr. A bin is a collection of chunks.
3. A "*chunk*" is a collection of rows that belong to the same set and bin and has the same nnzr for all rows. The maximum number of rows in a chunk equals to SIMDW.

The following algorithm stores a matrix in CCF:

1. Create T sets of matrix rows. The number of nonzero elements in each set is NNZ/T.

2. In each set, sort the rows in ascending order based on the number of nonzero elements per row. Rows with the same nnzr are assigned to a bin of rows.

3. In each bin, the collection of the first SIMDW rows forms the first chunk and the second SIMDW rows form the second chunk. This repeats until the number of rows in a bin is less than SIMDW. We refer to each row in such a bin as a “tail” row. For tail rows:

   (a) If the nnzr is less than or equal to SIMDW, then accumulate these rows in a new chunk.

   (b) If the nnzr is greater than SIMDW, then accumulate each row in a new chunk. That is, create single-row chunks.

4. Nonzero matrix elements in a multi-row chunk are reordered. CCF stores the first element of the second row after the first element of the first row and the first element of the third row after the first element of the second row. This repeats until all the first elements of rows are stored sequentially in “the first elements sequence”. The same reordering technique is used for the second elements of rows and CCF stores “the second elements sequence” sequentially following the “the first elements sequence”. This repeats for “the third elements sequence” and the remaining elements sequences until all elements in a chunk are stored using this ordering. In case there is a row with nnzr greater than NNZ/T, CCF divides this row into multiple parts each of length NNZ/T. Each part is assigned to a single set and a single thread.

Figure 3.1 demonstrates the decomposition of a matrix into chunks. Figure 3.2 shows an example matrix, A, in its sparse and compressed forms. Figure 3.3 shows the steps to map A into CCF on a platform using two threads and SIMDW =8.

3.2 CCF Data Structures

The following are the data structures used in CCF storage format:

1. Values Vector. This vector stores the sparse matrix nonzero elements in their final order after performing rows sorting to construct chunks of

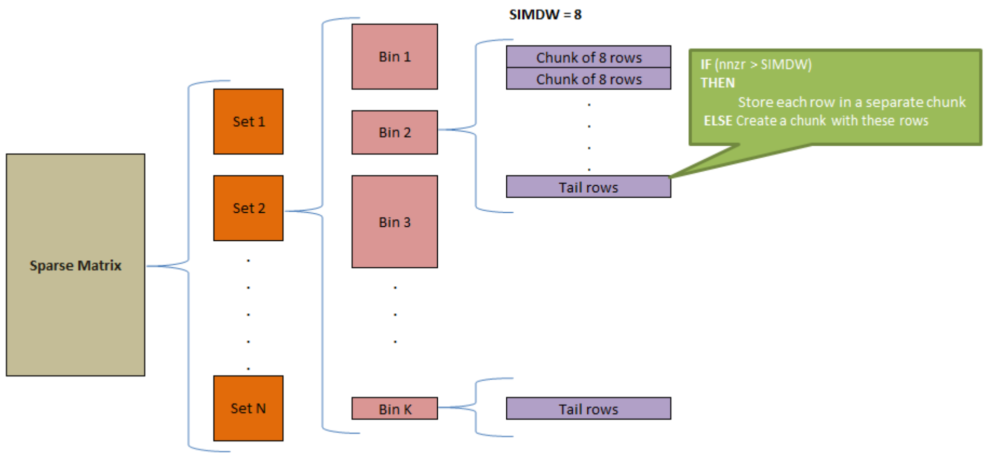
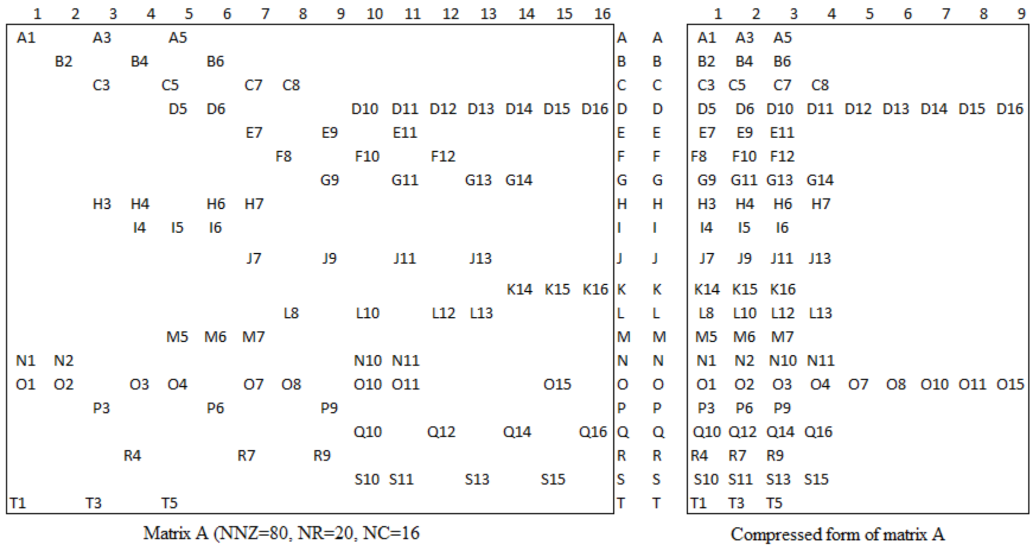


Figure 3.1: Decomposing a matrix into chunks.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
A1		A3		A5												A
	B2		B4		B6											B
		C3		C5		C7	C8									C
				D5	D6				D10	D11	D12	D13	D14	D15	D16	D
						E7		E9		E11						E
							F8		F10		F12					F
								G9		G11		G13	G14			G
		H3	H4		H6	H7										H
			I4	I5	I6											I
						J7		J9		J11		J13				J
													K14	K15	K16	K
							L8		L10		L12	L13				L
				M5	M6	M7										M
N1	N2								N10	N11						N
O1	O2		O3	O4		O7	O8		O10	O11				O15		O
		P3			P6			P9								P
									Q10		Q12		Q14		Q16	Q
			R4			R7		R9								R
									S10	S11		S13		S15		S
T1		T3		T5												T

Matrix A (NNZ=80, NR=20, NC=16)

	1	2	3	4	5	6	7	8	9
A	A1	A3	A5						
B	B2	B4	B6						
C	C3	C5	C7	C8					
D	D5	D6	D10	D11	D12	D13	D14	D15	D16
E	E7	E9	E11						
F	F8	F10	F12						
G	G9	G11	G13	G14					
H	H3	H4	H6	H7					
I	I4	I5	I6						
J	J7	J9	J11	J13					
K	K14	K15	K16						
L	L8	L10	L12	L13					
M	M5	M6	M7						
N	N1	N2	N10	N11					
O	O1	O2	O3	O4	O7	O8	O10	O11	O15
P	P3	P6	P9						
Q	Q10	Q12	Q14	Q16					
R	R4	R7	R9						
S	S10	S11	S13	S15					
T	T1	T3	T5						

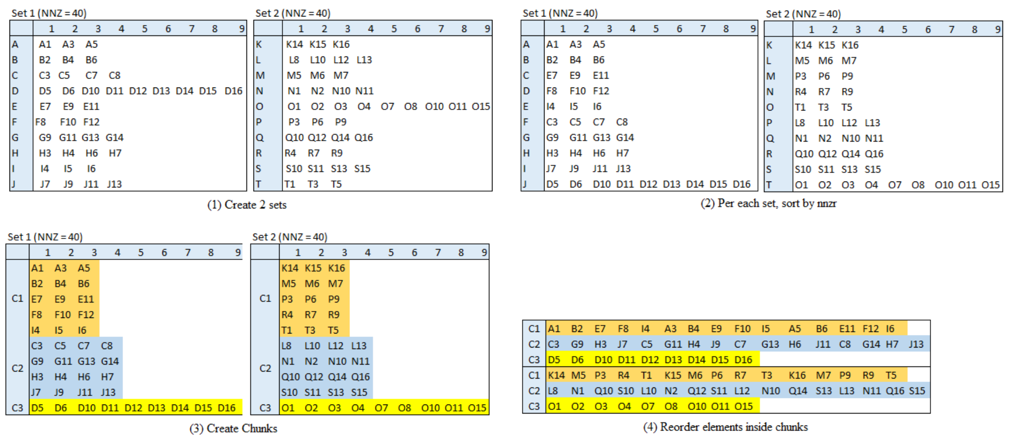
Compressed form of matrix A

Figure 3.2: Matrix A.

each set and performing nonzero elements reordering to build elements sequences inside chunks.

2. Columns Vector. This vector stores the column index for each nonzero element of the sparse matrix. The same index is used to obtain a nonzero element from Values Vector and its actual column index from the Columns Vector. The column index of a nonzero element maps to the proper x vector element that the kernel multiplies with this nonzero element.

3. Y Mapping Vector. This vector maps the sorted row index to its row


Set 1 (NNZ = 40)

	1	2	3	4	5	6	7	8	9
A	A1	A3	A5						
B	B2	B4	B6						
C	C3	C5	C7	C8					
D	D5	D6	D10	D11	D12	D13	D14	D15	D16
E	E7	E9	E11						
F	F8	F10	F12						
G	G9	G11	G13	G14					
H	H3	H4	H6	H7					
I	I4	I5	I6						
J	J7	J9	J11	J13					

Set 2 (NNZ = 40)

	1	2	3	4	5	6	7	8	9
K	K14	K15	K16						
L	L8	L10	L12	L13					
M	M5	M6	M7						
N	N1	N2	N10	N11					
O	O1	O2	O3	O4	O7	O8	O10	O11	O15
P	P3	P6	P9						
Q	Q10	Q12	Q14	Q16					
R	R4	R7	R9						
S	S10	S11	S13	S15					
T	T1	T3	T5						

(1) Create 2 sets

Set 1 (NNZ = 40)

	1	2	3	4	5	6	7	8	9
A	A1	A3	A5						
B	B2	B4	B6						
C	E7	E9	E11						
D	F8	F10	F12						
E	I4	I5	I6						
F	C3	C5	C7	C8					
G	G9	G11	G13	G14					
H	H3	H4	H6	H7					
I	J7	J9	J11	J13					
J	D5	D6	D10	D11	D12	D13	D14	D15	D16

Set 2 (NNZ = 40)

	1	2	3	4	5	6	7	8	9
K	K14	K15	K16						
L	M5	M6	M7						
M	P3	P6	P9						
N	R4	R7	R9						
O	T1	T3	T5						
P	L8	L10	L12	L13					
Q	N1	N2	N10	N11					
R	Q10	Q12	Q14	Q16					
S	S10	S11	S13	S15					
T	O1	O2	O3	O4	O7	O8	O10	O11	O15

(2) Per each set, sort by nnzr

Set 1 (NNZ = 40)

	1	2	3	4	5	6	7	8	9
C1	A1	A3	A5						
	B2	B4	B6						
	E7	E9	E11						
	F8	F10	F12						
	I4	I5	I6						
C2	C3	C5	C7	C8					
	G9	G11	G13	G14					
	H3	H4	H6	H7					
	J7	J9	J11	J13					
C3	D5	D6	D10	D11	D12	D13	D14	D15	D16

Set 2 (NNZ = 40)

	1	2	3	4	5	6	7	8	9
C1	K14	K15	K16						
	M5	M6	M7						
	P3	P6	P9						
	R4	R7	R9						
	T1	T3	T5						
C2	L8	L10	L12	L13					
	N1	N2	N10	N11					
	Q10	Q12	Q14	Q16					
	S10	S11	S13	S15					
C3	O1	O2	O3	O4	O7	O8	O10	O11	O15

(3) Create Chunks

C1	A1	B2	E7	F8	I4	A3	B4	E9	F10	I5	A5	B6	E11	F12	I6	
C2	C3	G9	H3	J7	C5	G11	H4	J9	C7	G13	H6	J11	C8	G14	H7	J13
C3	D5	D6	D10	D11	D12	D13	D14	D15	D16							
C1	K14	M5	P3	R4	T1	K15	M6	P6	R7	T3	K16	M7	P9	R9	T5	
C2	L8	N1	Q10	S10	L10	N2	Q12	S11	L12	N10	Q14	S13	L13	N11	Q16	S15
C3	O1	O2	O3	O4	O7	O8	O10	O11	O15							

(4) Reorder elements inside chunks

Figure 3.3: Mapping matrix A into CCF storage format.

index in the output y vector.

4. Chunks Information Vector. Each entry of this vector is composed of the following values:

   (a) Number of rows in a chunk. The maximum number of rows in a chunk is SIMDW.

   (b) Row length of all rows in the chunk. All rows of any chunk must have the same length.

   (c) Start input index, which is the index of the first nonzero element of the chunk in the Values Vector.

   (d) "Are rows in this chunk contiguous?" This indicates whether the rows of the chunk are contiguous or there are gaps in their indices.

   (e) Start y index. This value refers to either the start row index in the Y Mapping Vector in case that the rows are not contiguous or start row index in the actual y vector in case that the rows are contiguous.

5. Sets Pointer. This points to the first chunk index of each set in the Chunks Information Vector.

3.3 SpMV CCF Kernel Description

The SpMV CCF kernel uses a single thread to process one set of rows. A single thread processes chunks of a set in sequential order. To process a multi-row chunk, a thread loads elements of the first elements sequence from the Values Vector and gathers the corresponding x vector elements by using the actual column indices loaded from the Columns Vector (indirect memory access of vector x). The thread multiplies the elements of the first elements sequence by the x vector elements and accumulates the results in an accumulation vector. Next, the thread loads the elements of the second elements sequence from Values Vector and collects the corresponding x vector elements by using the actual column indices loaded from the Columns Vector (indirect memory access of vector x). The thread multiplies the elements of the second elements sequence by the x vector elements and accumulates the results in an accumulation vector. The SpMV CCF kernel repeats until all elements sequences in a chunk are processed. If rows are contiguous in the chunk, the kernel stores the temporary accumulation vector elements directly into the y vector. If rows are not contiguous, the kernel loads the actual y indices from the Y Mapping Vector and uses them to scatter the temporary accumulation vector elements to their actual y vector values. Figure 3.4 is a high-level illustration of processing a multi-row chunk by SpMV CCF.

If a chunk has a single row, SpMV CCF processes the row completely and stores the result into the y vector. Figure 1.1 illustrates processing of a single-row chunk by SpMV CCF.

3.4 CCF Performance Discussion

CCF enhances the SIMD efficiency by grouping rows with the same nnzr in a single chunk. The formation of chunks is the fundamental concept in CCF design. The intuition behind this design is to create chunks with SIMDW of rows (or as close as possible to SIMDW). This enhances the utilization of the vector units and eliminates the reduce-add vector instructions needed at the end of each row processing in the generic SpMV CSR kernel processing. Figure 1.1 shows how SpMV CCF processes a single-row chunk. It is the same way the SpMV CSR kernel processes rows. SpMV CCF does not

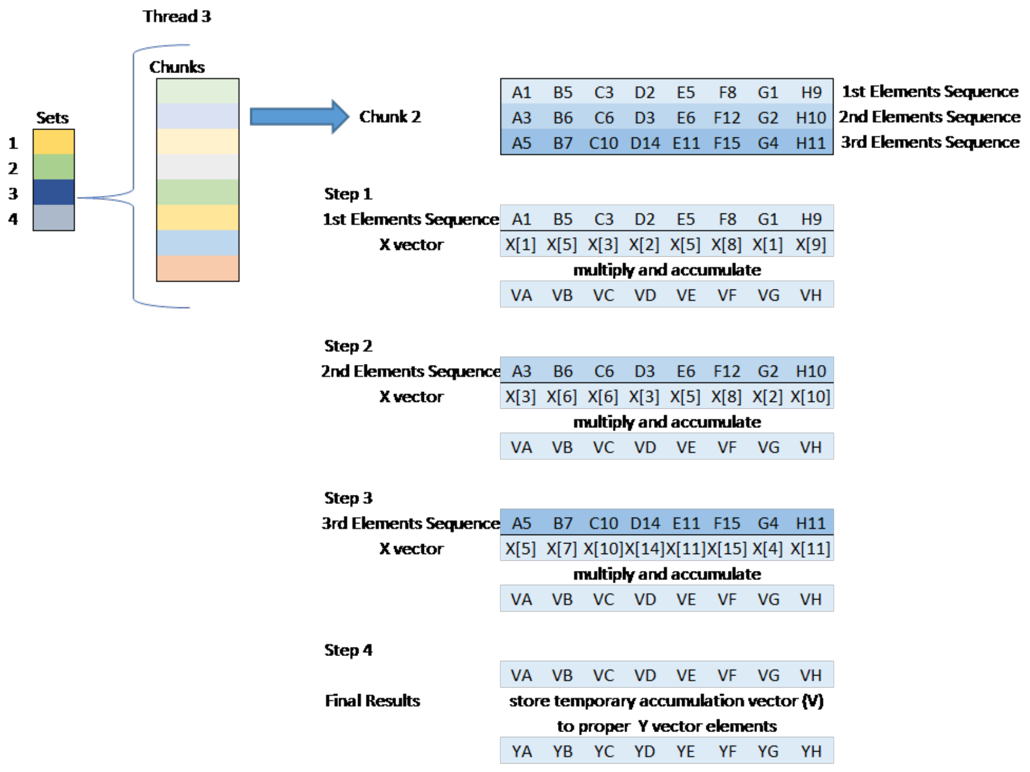


Figure 3.4: High-level graphical illustration of SpMV CCF: processing a multi-row chunk.

need the reduce-add instruction for multi-row chunks because the final vector has results for different rows (figure 3.4). Furthermore, grouping rows with the same nnzr helps suppress the need for padding zeros or using mask bit array and masked vector instructions as is done for matrices stored in the ELLPACK Sparse Block (ESB) storage format introduced in [7].

Collecting rows with the same nnzr in a chunk is the default transformation of CCF during the storage format preparation. However, when the number of tail rows of a bin is less than SIMDW and the nnzr is large, processing one row at a time utilizes vector units more efficiently than collecting these rows in a chunk and processing multiple elements from different rows simultaneously. As a heuristic, we use SIMDW nnzr as a threshold to identify long rows, however this is a tunable parameter.

Load balancing is achieved by two techniques. The first is to divide the nonzero elements of the matrix evenly between threads. The second is used when processing very long rows. The nnzr for such rows exceeds NNZ/T and hence the kernel uses multiple threads instead of a single thread. This is the only case for which a row is processed by more than a single thread in SpMV CCF. If a matrix has a single very long row, processing this row by a single thread will dominate the execution time and any improvement to the kernel requires reducing the processing time of this row.

Moreover, sorting rows within the set boundaries helps avoid the false sharing [23] problem that can occur when threads store results to the y vector. If all the rows of a matrix are sorted without using sets (sorting across the entire matrix), there is a possibility that threads on different cores write to different y elements that belong to the same cache block. In this case, unnecessary cache coherency messages would hurt the performance. Sets represent safeguards that divide the matrix into non-overlapping y vector regions. Thus, no false share will occur because each thread writes to its designated region of the y vector that does not intersect with any other thread's region.

CHAPTER 4

EVALUATION: PERFORMANCE BREAKDOWN

In this chapter, we analyze all the techniques used to achieve high performance in CCF. We study four incremental versions of CCF; each version has one more technique than its predecessor. In chapter 5, we compare the performance of SpMV CCF with three SpMV formats on two Intel platforms.

4.1 Methodology

4.1.1 CCF Incremental Versions

Table 4.1 shows CCF incremental versions along with the techniques used in each version. To study the SIMD efficiency, we manually instrumented the CSR, CCF0, and CCF1 kernels to count the number of vector arithmetic (fused multiply-add and reduce-add) operations. Intuitively, reducing the number of vector operations means higher SIMD efficiency. To verify load balancing improvements, we measured the execution time of each thread in each kernel and compared it to longest execution time of all threads.

We compare the performance of CSR, CCF0, and CCF1 to show the effectiveness of the SIMD efficiency enhancement techniques used in CCF. Moreover, we compare the performance of CCF1, CCF2, and CCF to show the effectiveness of the load balancing techniques used in CCF. Summary tables that compare the performance of each incremental version and the CSR kernel are provided in the comparison sections (4.2, 4.3, 4.4, and 4.5) to show the performance benefits achieved by each technique.

Table 4.1: CCF incremental versions.

	Technique	**Optimization Target**
CCF0	Groups and processes rows in chunks (core functionality of CCF)	SIMD efficiency
CCF1	CCF0 + processes long tail rows using CSR	SIMD efficiency
CCF2	CCF1 + divides nonzero elements evenly across threads	Load balancing
CCF	CCF2 + partitions very long rows	Load balancing

4.1.2 Platform

We use a single node dual 24-core Skylake Intel Xeon Platinum (table 5.1) and one thread per core (48 threads). This choice of threads is sufficient to analyze SpMV CCF performance; however, the maximum performance of CSR and CCF might be attained by using different thread count. We implemented the CSR kernel using AVX512 Intrinsics. The initialization of the CSR vectors on the dual Skylake is NUMA-aware. We manually instrumented CSR and all CCF incremental kernels to count the number of vector operations and time the execution of threads.

4.1.3 Test Matrices

We use a test set of 151 unstructured sparse matrices taken from the University of Florida collection [24] and Stanford Network Analysis Platform (SNAP) [25]. The matrices cover 38 scientific and engineering applications. Our test set includes the unstructured matrices used in the Intel SpMV benchmark [26] and all the matrices used by Xie et. al [9]. The appendix lists all 151 matrices and some of their characteristics.

4.1.4 Matrices Classification

Figure 4.1 shows our hierarchical classification of our set of matrices.

The HPC matrices are from scientific and engineering applications and are regular in nature (i.e. the distribution of the nonzero elements is regular and the nnzr for each row is similar) while the scale-free matrices are from more practical applications such as social networking, data analytics, and transportation [4, 5, 6]. These matrices are highly irregular [6]. The same-nnzr matrices have the same nnzr for every row in the matrix. These matrices exemplify the best sample to show how ELLPACK-based formats such as

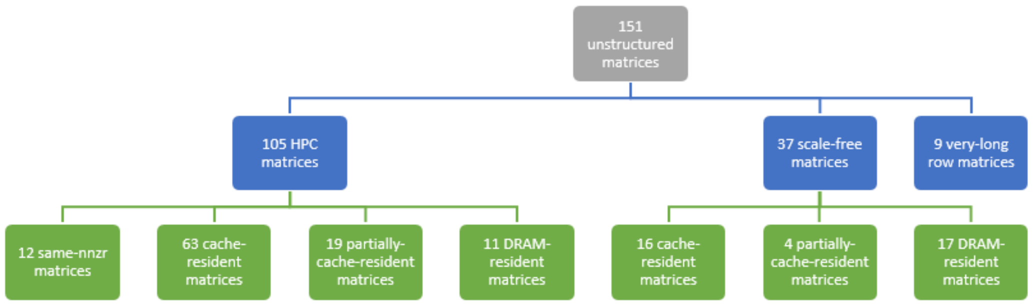

151 unstructured matrices
105 HPC matrices
37 scale-free matrices
9 very-long row matrices
12 same-nnzr matrices
63 cache-resident matrices
19 partially-cache-resident matrices
11 DRAM-resident matrices
16 cache-resident matrices
4 partially-cache-resident matrices
17 DRAM-resident matrices

Figure 4.1: Hierarchical classification of the matrices using a dual 24-core Skylake with 114 MB cache.

Table 4.2: CSR vectors along with their sizes in bytes.

Vector	**Size in bytes**
Nonzero elements	NNZ x 8
Column index	NNZ x 4
Row Index	(NR+1) x 4
X vector	NC x 8
Y vector	NR x 8
Approx. Total size	**12NNZ + 12NR + 8NC**

CCF improve the SIMD efficiency compared to CSR.

We use the CSR data structure to express the approximate required memory for an unstructured matrix (see table 4.2). The nonzero elements in the matrix need to be transferred to the cores before being processed. If the x and y vectors and other auxiliary data fit the cache and incur no memory traffic, the SpMV kernel would be bounded by streaming the nonzero elements and the column indices. Therefore, the achieved performance depends significantly on whether these two data structures reside in the very fast cache or need to be streamed from the slower memory.

Using CSR data structures, we define the following classes of matrices:

- Cache-resident matrices: All CSR data structures for these matrices fit in the cache (12NNZ + 12NR + 8NC <= cache size).

- Partially-cache-resident matrices: The data structures representing such matrices partially fit in the cache. This occurs when the total size of the CSR data structures is greater than the cache size but the size of summation of the sizes of the column index, x, y, and row index is less than the cache size.

Table 4.3: Comparing the performance of CCF0 and CSR on Skylake.

	#(%) of matrices	Max	Avg.
CCF0 faster than CSR	55(36%)	2.3x	1.4x
CCF0 same as CSR	48(32%)	-	-
CSR faster than CCF0	48(32%)	2.2x	1.2x

- DRAM-resident matrices: For these matrices neither the nonzero elements vector nor the column index vector fits in the cache. The performance of these matrices is memory bandwidth bound and is characterized as follows:
  - Flop-to-byte-ratio = 2 (add and multiply) / 12 bytes = 1/6. This assumes that the x and y vectors and other auxiliary data incur no memory traffic.
  - Maximum possible GFLOP/s = Flop-to-byte-ratio x Maximum memory bandwidth.

The dual 24-core Skylake has a total cache size of 114 MB (48 MB L2 cache + 66 MB L3 non-inclusive cache) and a maximum theoretical memory bandwidth of 240 GB/s.

4.2 CCF0 vs. CSR

CCF0 improves the SIMD efficiency by grouping rows with the same nnzr in a single chunk. CCF0 processes the rows of each chunk in column-major order like ELLPACK but without using zero-padding. Table 4.3 shows a comparison between CCF0 and CSR. Each kernel outperforms the other for almost the same percentages of matrices. This indicates that each kernel is effective for different groups of matrices.

4.2.1 HPC Matrices

To study the effectiveness of CCF0, we use the 12 same-nnzr matrices. The chunks created for these matrices always have SIMDW rows and the workload of all threads is balanced. Table A.1 in the appendix lists these 12 matrices along with some of their characteristics. Figure 4.2 shows that CCF0 reduces

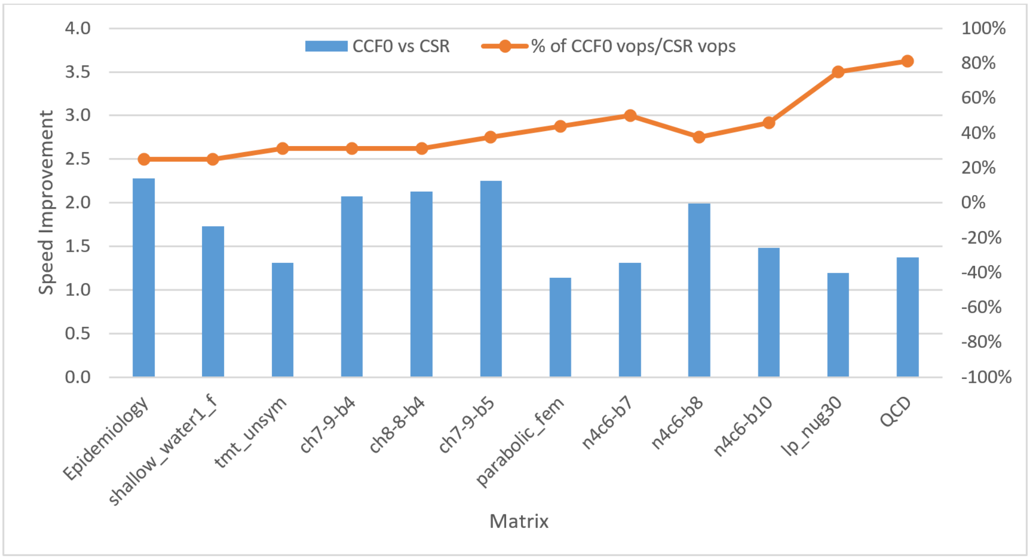


Figure 4.2: Speed improvement of CCF0 compared to CSR and the percentage of CCF0 vector operations compared to CSR vector operations for the 12 same-nnzr matrices.

the number of vector operations and improves the performance for all the matrices in this group.

For HPC cache-resident matrices, we observe that there is a relation between the reduction in the vector operations and the achieved performance improvement of CCF0 compared to CSR. For the 33 matrices for which CCF0 outperforms CSR, CCF0 always reduces the number of vector operations compared to CSR. This can be clearly noticed for the matrices with small mean nnzr. Table 4.4 lists 8 matrices with mean nnzr less than 8. Figure 4.3 shows the comparison between performance and the vector operation percentages of CSR and CCF0 for these matrices.

On the other hand, CSR is faster than CCF0 either when the numbers of vector operations for the two kernels are close or when CCF0 executes many more vector operations. Table 4.5 shows five cache-resident matrices for which CCF0 achieves the lowest compared CCF0. These matrices have high mean and standard deviation nnzr. Figure 4.4 shows the speed improvement of CSR compared to CCF0 and the percentage of CCF0 vector operations compared to CSR.

For HPC partially-cache-resident and the HPC DRAM-resident matrices, we observe that the performances of CCF0 and CSR are close to each other for all the matrices.

Table 4.4: An example of cache-resident matrices with small mean nnzr (mean nnzr <8).

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr
Raj1	1,302,464	263,743	263,743	4.9	88.3	40,468
darcy003	2,101,242	389,874	389,874	5.4	2	7
scircuit	958,936	170,998	170,998	5.6	4.4	353
ASIC_320ks	1,827,807	321,671	321,671	5.7	7.9	412
Economics	1,273,389	206,500	206,500	6.2	4.4	44
helm2d03	2,741,935	392,257	392,257	7	0.1	9
tmt_sym	5,080,961	726,713	726,713	7	1	9
hvdc2	1,347,273	189,860	189,860	7.1	3.8	60

Table 4.5: Top five HPC cache-resident matrices for which CSR is faster than CCF0.

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr
std1_Jac3	1,455,848	21,982	21,982	66.2	169.3	1,030
std1_Jac2	1,248,731	21,982	21,982	56.8	145.6	898
crankseg_2	7,106,348	63,838	63,838	111.3	108.5	3,423
appu	1,853,104	14,000	14,000	132.4	36.5	294
nd6k	6,897,316	18,000	18,000	383.2	89.2	514

4.2.2 Scale-free Matrices

For scale-free matrices, CSR is faster than CCF0 for 20 matrices with an average speed improvement of 1.2x (maximum 1.6x). There is no correlation between the mean nnzr, matrix size, ratio of CCF0 vector operations to CSR vector operations, and the delivered performance; these matrices are highly irregular. For example, matrix road_usa mean nnzr is 2.4 and the maximum nnzr is 9. Theoretically, this matrix is a potential target of CCF0 to improve SIMD efficiency and hence performance. The first part of the hypothesis is correct; CCF0 eliminates 85% of the CSR vector operations. However, CCF0 performs slower than CSR; CSR is 1.6x times faster than CCF0. We investigated this issue and found out that sorting rows in a huge sorting window (i.e. sorting within a set per thread) for highly irregular matrices achieves low performance for some matrices. Table 4.6 shows profiling information using "perf" tool [27] for the cache traffic when using CSR and CCF0. These counters are collected for the whole program execution with 1000 times SpMV kernel runs. CSR L1 dcache load hits are 3.2x higher than CCF0, which means that CSR has better L1 data locality than CCF0. In

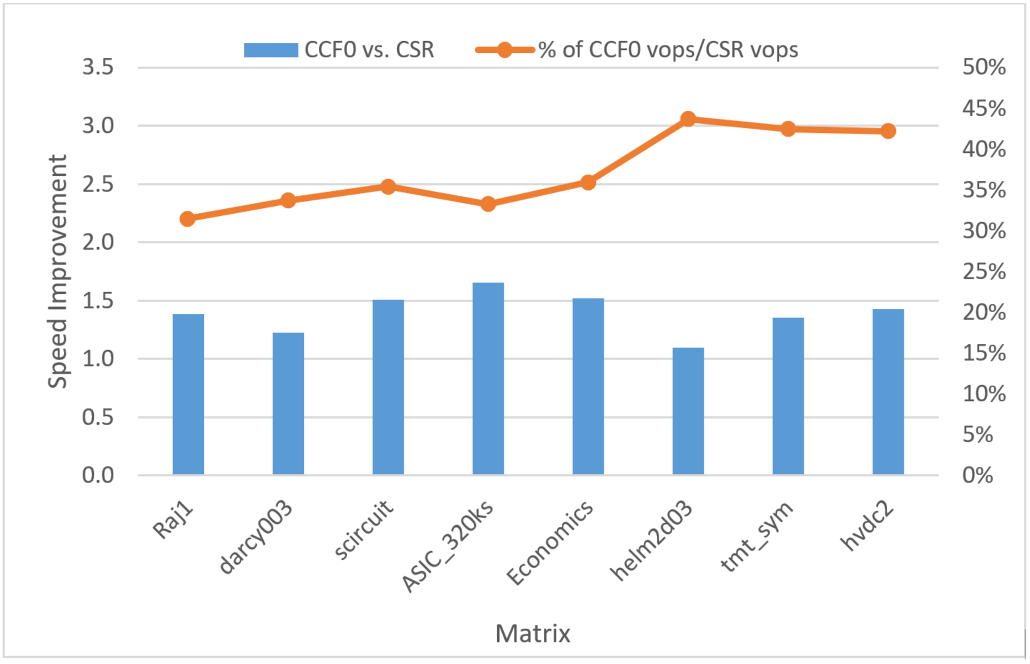


Figure 4.3: HPC matrices with small mean nnzr. CCF0 significantly reduces the number of vector operations and improves the performance.

Table 4.6: Profiling counters for road_usa matrix using CSR and CCF0 (in billions).

Performance Counter	CSR	CCF0
L1 dcache load hits	372.4	115.6
L1 dcache load misses	30.8	47.7
LLC load hits	4.5	17.0
LLC load misses	3.1	12.8

chapter 6, we show that increasing the number of sets per thread improves the performance of this matrix (see table 6.1).

4.3 CCF1 vs. CCF0

CCF1 tries to improve the SIMD efficiency of CCF0 by using a hybrid execution model for the tail rows of each bin. The tail rows can be grouped in a single chunk or separated into single-row chunks based on their length (nnzr). Table 4.7 shows a summary comparison between CCF1 and CSR. On comparing with table 4.3, one sees that CCF1 increased the number of matrices for which CCF is faster than CSR, and reduced the number of matrices

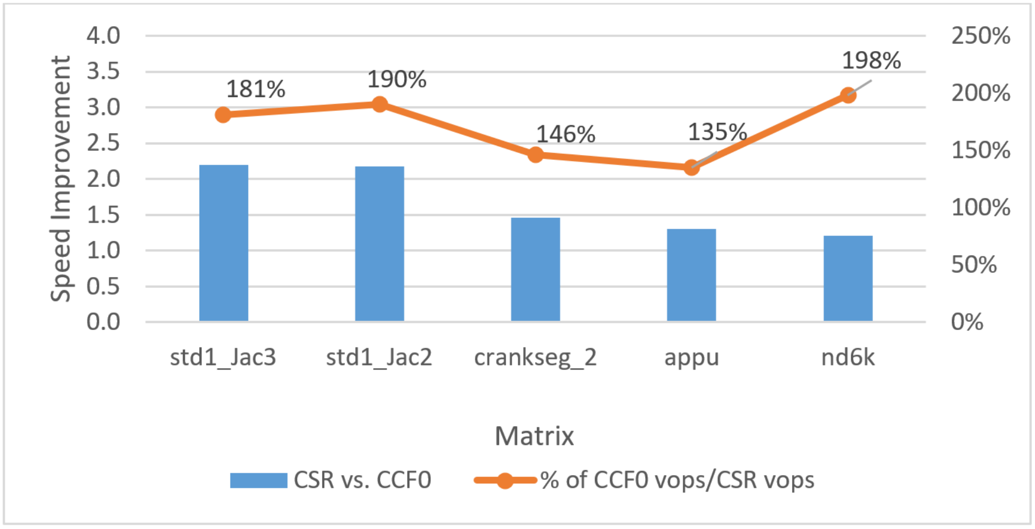


Figure 4.4: The five HPC matrices for which CCF0 achieves the lowest compared to CSR.

Table 4.7: Comparing the performance of CCF1 and CSR on Skylake.

	#(%) of matrices	Max	Avg.
CCF1 faster than CSR	79(52%)	2.3x	1.3x
CCF1 same as CSR	47 (31%)	-	-
CSR faster than CCF1	25 (17%)	1.4x	1.1x

for which CCF is slower than CSR.

4.3.1 HPC Matrices

For the same-nnzr matrices, CCF1 has no effect on the performance because all the chunks have SIMDW rows (no tail rows). In addition, CCF1 has a negligible performance improvement for the matrices with small mean nnzr. For the HPC cache-resident matrices, CCF1 outperforms CCF0 for 25 matrices with an average speed improvement of 1.3x (maximum 2.2x). Moreover, CCF1 outperforms CSR for 44 matrices with an average speed improvement of 1.3x (maximum 2.1x). Table 4.8 lists 5 matrices for which CCF0 has less performance than CSR and CCF1 has higher performance than CSR. CCF1 further reduces the number of vector operations compared to CCF0. These matrices have high standard deviation nnzr and their mean nnzr is greater than the threshold we set to decide whether CCF preprocessor (the routine that transforms a sparse matrix into CCF format) should compress the tail

Table 4.8: An example of 5 HPC cache-resident matrices for which CCF0 is slower than CSR and CCF1 is faster than CSR. CCF1 further reduces the vector operations and improves the performance.

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr
bbmat	1,771,722	38,744	38,744	45.7	38.4	126
li	1,350,309	22,695	22,695	59.5	29.2	108
vanbody	2,336,898	47,072	47,072	49.6	17.6	232
Protein	4,344,765	36,417	36,417	119.3	31.9	204
ct20stif	2,698,463	52,329	52,329	51.6	17	207

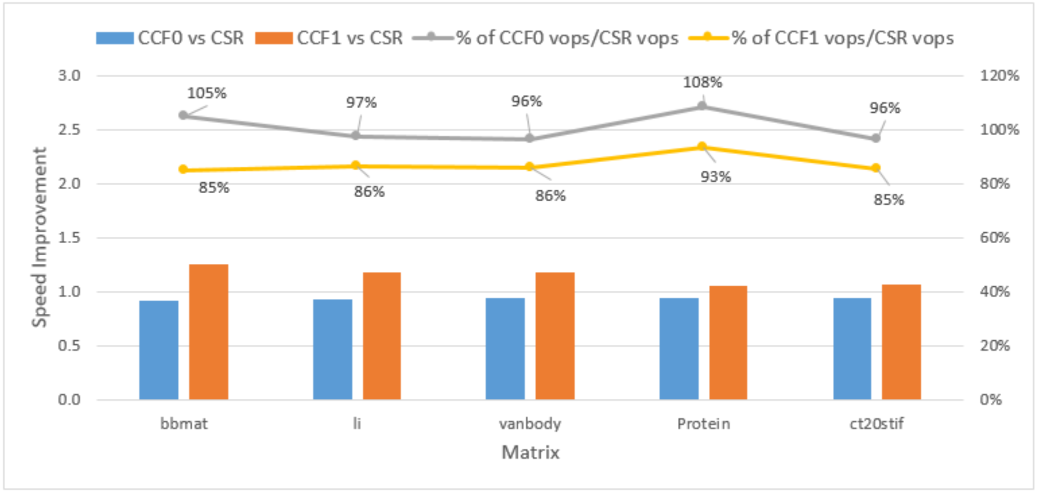


Figure 4.5: A comparison of CSR, CCF0, and CCF1 for example five matrices for which CCF0 is slower than CSR and CCF1 is faster than CSR. CCF1 further improves the performance by reducing the number of vector operations.

rows into a multi-row chunk or separate them into single-row chunks. Figure 4.5 shows how CCF1 reduces the number of vector operations compared to CCF0 and improves the performance of CCF0.

Most of the matrices with improved performance have a high standard deviation nnzr. For the matrices listed earlier in table 4.5 for which CSR has a higher performance than CCF0, figure 4.6 shows how CCF1 significantly reduces the number of vector operations compared to CCF0 and improves the performance of CCF0.

For the HPC partially-cache-resident and the HPC DRAM-resident matrices, we observe that the two SIMD efficiency techniques implemented in CCF0 and CCF1 provide no performance improvement over CSR.

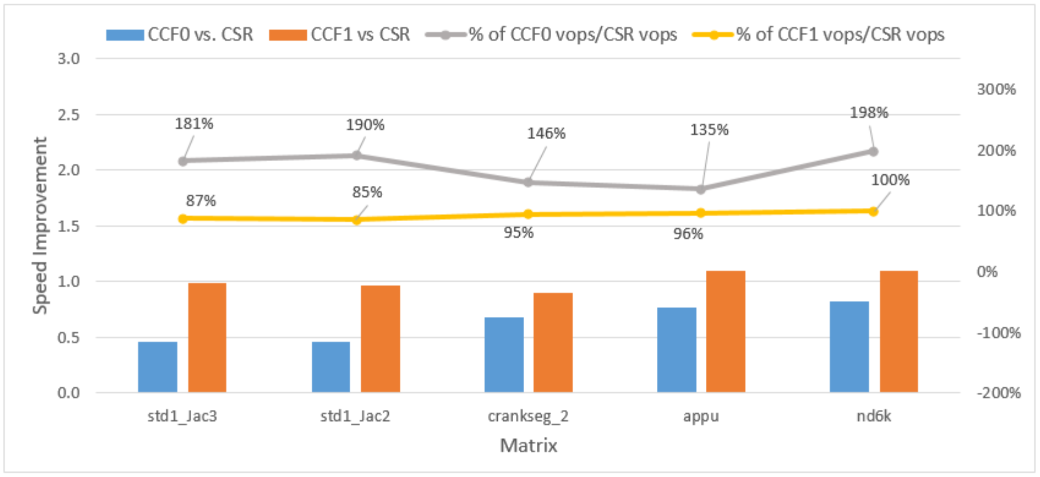


Figure 4.6: A performance comparison between CCF1, CCF0, and CSR for the five HPC matrices for which CCF0 achieves the lowest compared to CSR.

4.3.2 Scale-free Matrices

For the scale-free matrices, we observe that the two SIMD efficiency techniques implemented in CCF0 and CCF1 provide no performance improvement over CSR.

4.4 CCF2 vs. CCF1

CSR, CCF0, and CCF1 divide the rows evenly across threads while CCF2 divides the nonzero elements evenly across threads. Table 4.9 shows that CCF2 significantly improves the performance of CSR when compared to CCF1. CCF2 considerably improves the performance of the scale-free and the very-long-row matrices. Table 4.10 shows the performance breakdown for the top three classes of matrices. The scale-free matrices and the very-long-row matrices are highly irregular, which is why dividing the nonzero elements evenly helps unify the workload assigned to each thread. For the HPC matrices, which are naturally more uniform, CCF2 also improves the performance for 40% of matrices.

Table 4.11 lists two matrices, one from the HPC group and one from the scale-free group, for which CCF2 considerably outperforms CCF1. Figure 4.7 shows the execution time of each thread for matrix TSC_OPF_1047. Appar-

Table 4.9: Comparing the performance of CCF2 and CSR on Skylake.

	#(%) of matrices	Max	Avg.
CCF2 faster than CSR	79(52%)	2.3x	1.3x
CCF2 same as CSR	47 (31%)	-	-
CSR faster than CCF2	25 (17%)	1.4x	1.1x

Table 4.10: The effect of CCF2 on performance compared to CCF1 for the three groups of matrices.

Group	CCF2 faster than CCF1		
	#(%) of matrices accelerated	Max	Avg.
105 HPC matrices	42 (40%)	5.0x	1.6x
37 scale-free matrices	30 (81%)	23.4x	2.9x
9 very-long-row matrices	9 (100%)	4.1x	2.2x

ently, there are many idle threads in CCF1 and one thread is taking a very long time to finish compared to the other threads. CCF2 with load balancing technique mitigates this problem and distributes the workload fairly. CCF2 is five times faster than CCF1. Figure 4.8 demonstrates a worse case for matrix wiki-Talk, in which CCF1 is almost executing serially. CCF2 alleviates this problem and creates a load-balanced execution. CCF2 is 23 times faster than CCF1. We observe that even in CCF2, there is one thread that finishes much later than the other threads. We think that this thread has too many short rows that will need to go more frequently to the memory for loads and stores. We leave this topic for future work.

4.5 CCF vs. CCF2

CCF employs all the four techniques that improve the SIMD efficiency and load balancing. Very-long-row matrices suffer from few rows that are extremely long when compared to the matrix other rows. CCF seamlessly detects these rows and partitions them into multiple segments. Each segment is processed by a single thread. CCF preprocessor divides any very long row into segments when the row's nnzr is greater than the total nonzero elements of the matrix divided by the number of threads assigned for the kernel execution (row nnzr >(NNZ/T)). This means that the nnzr of this row exceeds the share of nonzero elements assigned to each thread. Table 4.12

Table 4.11: Two matrices for which CCF2 considerably outperforms CCF1.

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr
TSC_OPF_1047	2,016,902	8,140	8,140	247.8	323.6	1,526
wiki-Talk	5,021,410	2,394,385	2,394,385	2.1	99.9	100,022

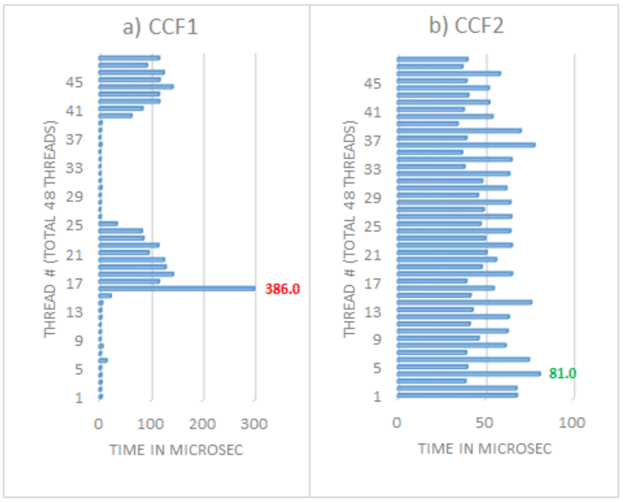


Figure 4.7: The execution time per each kernel thread of CCF1 and CCF2 for matrix TSC_OPF_1047.

shows that 9 matrices with very long rows are detected by CCF preprocessor when using 48 threads. Figure 4.9 shows the delivered GFLOP/s by all the kernels used in this section. CCF achieves impressive speed improvements for most of the matrices.

To demonstrate the effectiveness of this technique, we select two matrices from table 4.12, dc2 and rajat30, and show the execution time for each thread in the CCF2 and CCF kernels. For matrix dc2, figure 4.10 shows that CCF2 has some idle threads and the first two threads do most of the work. CCF, on the other hand, has a more balanced workload distribution. Thus, CCF is 3.3x faster than CCF2. Figure 4.11 shows the same story for matrix rajat30.

Table 4.13 shows the final summarized comparison between CCF and CSR using 48 threads on the Skylake. CCF accelerates the SpMV computation for 82% of the matrices. CSR outperforms CCF for 5% of the matrices; we will show that our auto-tuned CCF reduces this percentage to zero.

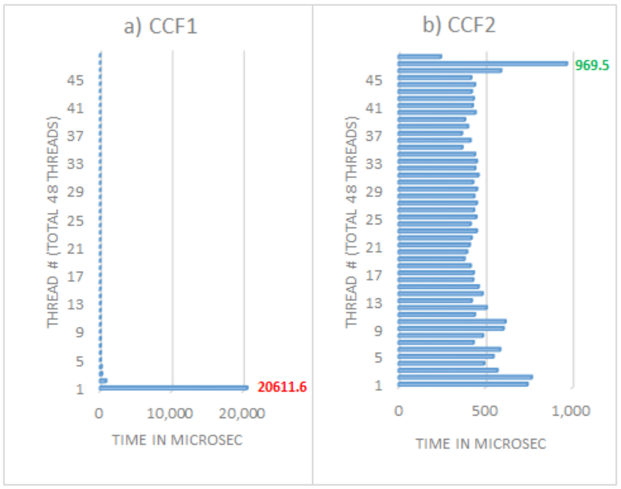


Figure 4.8: The execution time per each kernel thread of CCF1 and CCF2 for matrix wiki-Talk.

Table 4.12: Matrices with very long rows on the Skylake using 48 threads.

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr
rajat24	1,948,235	358,172	358,172	5.4	180.1	105,296
rajat29	4,866,270	643,994	643,994	7.6	773.9	454,521
rajat30	6,175,377	643,994	643,994	9.6	784.6	454,746
ASIC_680k	3,871,773	682,862	682,862	5.7	659.8	395,259
circuit5M	59,524,291	5,558,326	5,558,326	10.7	1,356.60	1,290,501
connectus	1,127,525	512	394,792	2,202.20	7,584.40	120,065
dc2	766,396	116,835	116,835	6.6	361.5	114,190
FullChip	26,621,990	2,987,012	2,987,012	8.9	1,806.80	2,312,481
ins2	2,751,484	309,412	309,412	8.9	590.4	309,412

4.6 Summary

In this chapter, we discussed and illustrated how CCF improves SIMD efficiency and load balancing. We introduced four incremental versions of CCF (each version employs one more technique than its predecessor). We divided our set of test matrices into different classes based on their application areas and memory residency (cache or DRAM). CCF0 implements the core functionality of CCF. It groups and processes rows with the same nnzr in chunks. CCF0 improves the performance of the SpMV computation for matrices with the same nnzr and for small standard deviation (less than eight) nnzr cache-resident HPC matrices. CCF1 enhances CCF0 by processing long tail rows using CSR and improving the SIMD efficiency for cache-resident matrices with high nnzr standard deviation (in tens or hundreds). CC2 improves load

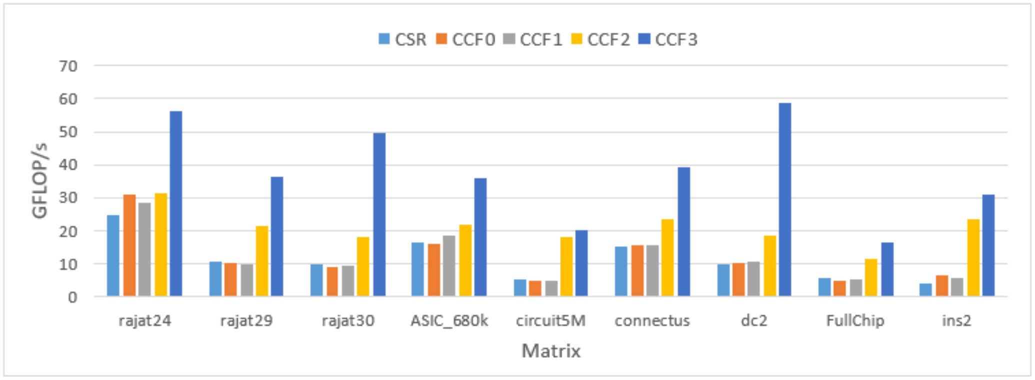


Figure 4.9: GFLOP/s of CSR, CCF0, CCF1, CCF2, and CCF for very-long-row matrices.

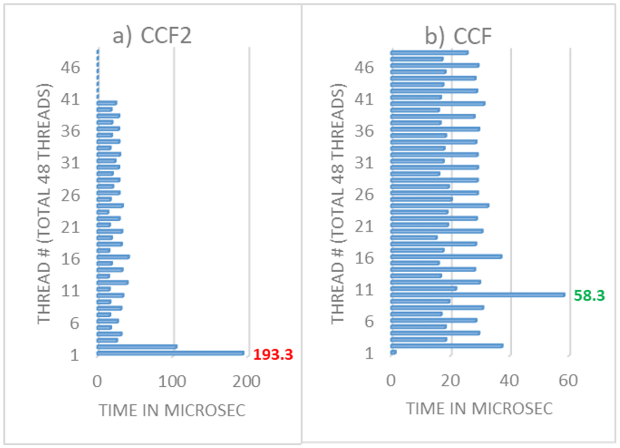


Figure 4.10: The execution time per each kernel thread of CCF2 and CCF for matrix dc2.

balancing by dividing the nonzero elements evenly between threads. CCF2 significantly improves the performance for 52% of matrices. Moreover, CCF2 considerably improves the performance of the scale-free and the very-long-row matrices. CCF, which employs the four techniques, improves load balancing by partitioning very long rows into multiple segments. Each segment is processed by a single thread. CCF significantly improves the performance of matrices with very long rows.

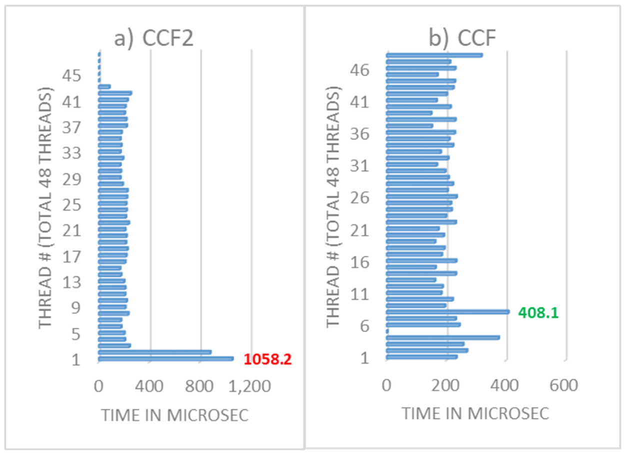


Figure 4.11: The execution time per each kernel thread of CCF2 and CCF for matrix rajat30.

Table 4.13: Comparing the performance of CCF and CSR on Skylake.

	#(%) of matrices	Max	Avg.
CCF faster than CSR	124 (82%)	20.9x	2.1x
CCF same as CSR	20(13%)	-	-
CSR faster than CCF	7(5%)	1.6x	1.1x

CHAPTER 5

EVALUATION: COMPARISON WITH OTHER FORMATS

In this chapter, we compare the double precision performance of the SpMV CCF kernel with three other SpMV kernels on two Intel platforms using 151 unstructured matrices.

5.1 Methodology

5.1.1 Platforms

The SpMV kernels are evaluated using two platforms: a 68-core KNL Intel Xeon Phi 7250 and a dual 24-core Skylake Intel Xeon Platinum 8160 (table 5.1). In this thesis, we refer to the Xeon Phi 7250 as KNL. Moreover, we refer to the dual Xeon Platinum 8160 as Skylake. KNL is a standalone machine and does not need a host to operate. Intel C++ v18.0.1 compiler and OpenMP are used. Thread scheduling is static. For KNL, we vary the number of threads per core from 1 to 4 and report the highest measured performance. For Skylake, we report the highest performance for 1 and 2 sockets using 1 and 2 threads per core. We calculate the average execution time of 1000 SpMV kernel runs. We compute the GFLOP/s by dividing twice the number of nonzero elements in the sparse matrix by the average execution time. We exclude the first SpMV kernel execution since it warms up caches. We use OpenMP to capture the start and end wall-clock times of kernel execution. Our threshold for considering a SpMV kernel faster than another is 5%.

Table 5.1: Specifications of the KNL and the Skylake platforms used for performance evaluation.

	KNL	Dual-socket Skylake
Processor	Intel Xeon Phi 7250	Intel Xeon Platinum 8160
Cluster	XSEDE [28] Stampede 2	XSEDE Stampede 2
Core microarchitecture	Silvermont	Skylake
Launch date	Q2'16	Q3'17
Peak double precision performance	3 TFLOPS	2x3 TFLOPS
Processor base frequency	1.4 GHz	2x2.1 GHz
Number of cores	68	2x24
Maximum number of threads	272 (4 per core)	96 (2 per core)
Vectorization	AVX512	AVX512
L3 cache size	None	2x33 MB
L2 cache size	34 MB	2x24 MB
Main memory	MCDRAM (16GB)	DDR4-2666 (768 GB)
Main memory bandwidth	450 GB/s	2x120 GB/s
Memory mode	Flat	N/A
Clustering mode	Quadrant	N/A

5.1.2 Formats for Comparison

We compare our SpMV CCF kernel with the following three existing storage formats:

- Intel MKL 2018u1 CSR which is widely used for sparse matrix representation and commonly adopted by many works for comparison. CSR has been deprecated in Intel MKL 2018u2. This is why we used the CSR in MKL 2018u1. The initialization of the CSR vectors is NUMA-aware on the dual Skylake (CSR vectors should be explicitly created and initialized by the user).

- Intel MKL 2018u2 Inspector-executor CSR that divides the operation into two stages: analysis and execution. The analysis phase starts by inspecting the matrix sparsity pattern and applying matrix structure changes. The execution stage improves the performance by reusing the information generated in the analysis phase. We only consider the performance of the execution phase in the performance comparisons.

- CVR [9] that achieves high performance for unstructured matrices, especially the scale-free matrices. CVR kernel achieves good speed improvements for unstructured matrices compared to five existing kernels: Intel MKL SpMV CSR, Intel MKL SpMV CSR(I) [10], and kernels presented in [8], [7], and [6].

In this chapter, we refer to MKL 2008u1 SpMV CSR, MKL 2008u2 Inspector-executor SpMV CSR, SpMV CVR, and our SpMV CCF as CSR, I-e, CVR, and CCF respectively. We do not consider the preprocessing time of any format in our performance evaluation.

5.1.3 Test Matrices

We use a test set of 151 unstructured sparse matrices from 38 scientific and engineering applications obtained from the University of Florida collection [24] and Stanford Network Analysis Platform (SNAP) [25]. Our test set includes the unstructured matrices used in the Intel SpMV benchmark [26] and all the matrices used in [9]. The appendix lists all 151 matrices and some of their characteristics. This is the same test set used in chapter 4.

5.2 Performance Comparison on Skylake

In this section, we use the same matrix classification used in figure 4.1. All the following figures show the speed improvement achieved by SpMV CCF compared to the other three kernels.

5.2.1 HPC Matrices

The HPC matrices are divided into four categories: same-nnzr, cache-resident, partially-cache-resident, and DRAM-resident. Figure 5.1 shows that CCF outperforms all the other kernels for all the matrices with the same nnzr. As discussed in chapter 4, grouping rows into chunks is the only technique that improves the performance of these matrices compared to CSR.

Figures 5.2, 5.3, 5.4, and 5.5 show the performance comparison between CCF and the other three kernels. CCF outperforms the other kernels for most of the matrices. The geomeans of the speed improvement of CCF compared to CSR, I-e, and CVR are 1.7x, 1.3x, and 1.8x respectively. This indicates

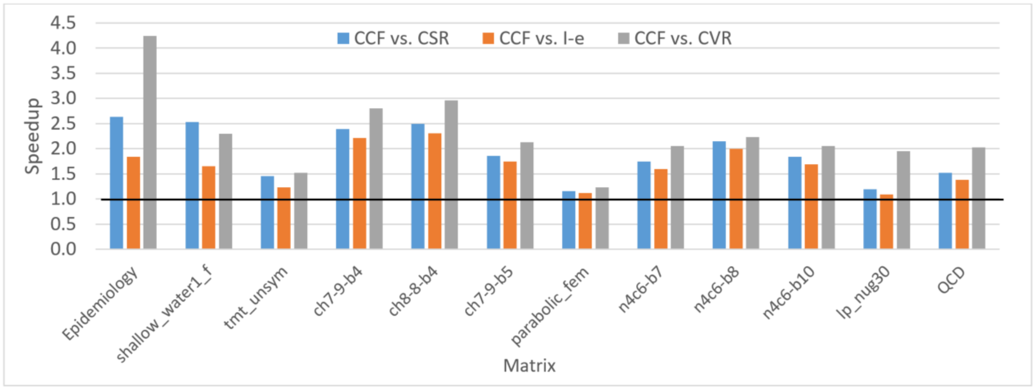


Figure 5.1: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the matrices with same nnzr on Skylake.

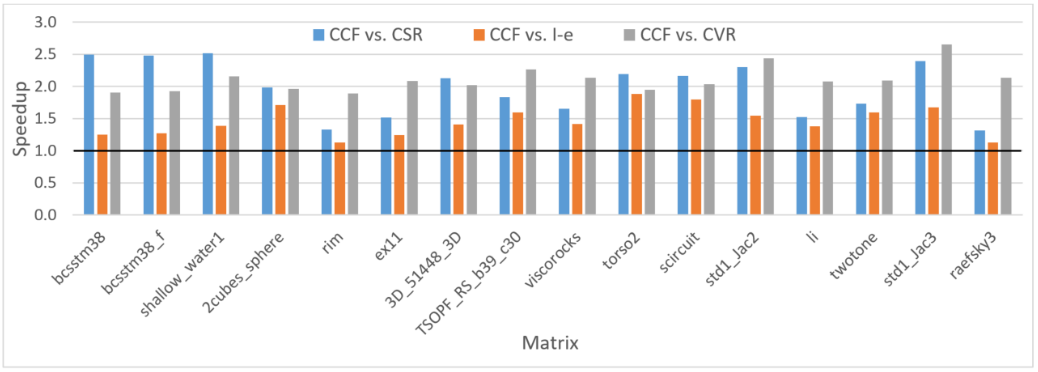


Figure 5.2: The speed improvement of SpMV CCF compared to the other three SpMV kernels for HPC cache-resident matrices (1/4).

that the techniques used by CCF effectively enhance the SIMD efficiency and load balancing for these matrices which are not memory bandwidth bound.

For the partially-cache-resident matrices, figure 5.6 shows that CCF outperforms the other three kernels for most of the matrices but with less speed improvement compared to the cache-resident matrices. The geomeans of the speed improvement of CCF compared to CSR, I-e, and CVR are 1.4x, 1.1x, and 1.1x respectively.

For the DRAM-resident matrices, all the kernels have similar performance because the performance of these matrices is bounded by the memory bandwidth of Skylake. Figure 5.7 demonstrates the achieved performance of CCF compared to the other three kernels.

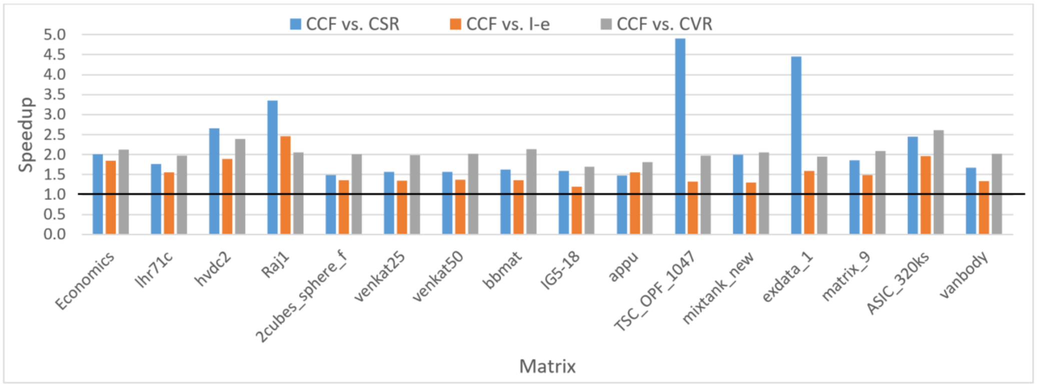


Figure 5.3: The speed improvement of SpMV CCF compared to the other three SpMV kernels for HPC cache-resident matrices (2/4).

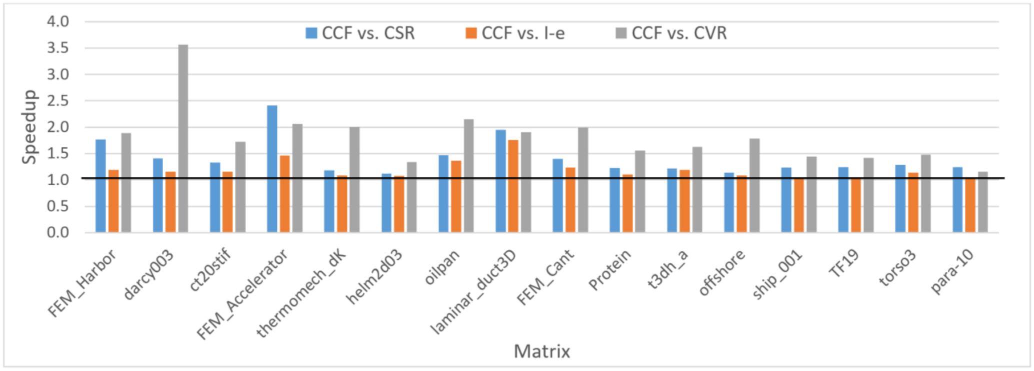


Figure 5.4: The speed improvement of SpMV CCF compared to the other three SpMV kernels for HPC cache-resident matrices (3/4).

5.2.2 Scale-free Matrices

The scale-free matrices are divided into three categories: cache-resident, partially-cache-resident, and DRAM-resident. Figure 5.8 shows the performance of CCF compared to the other three kernels for the cache-resident matrices. CCF is even more effective than the other kernels for these highly irregular matrices. The geomeans of the speed improvement of CCF compared to CSR, I-e, and CVR are 2.7x, 1.4x, and 2.0x respectively. For matrix wiki-Talk, CSR almost executes sequentially because of the kernel load imbalance. CCF, CVR, and I-e have better load balancing techniques but CCF has higher performance than CVR and I-e.

Figure 5.9 shows the performance of CCF compared to the other three kernels for the partially-cache-resident matrices. The geomeans of the speed improvement of CCF compared to CSR, I-e, and CVR are 2.3x, 1.1x, and

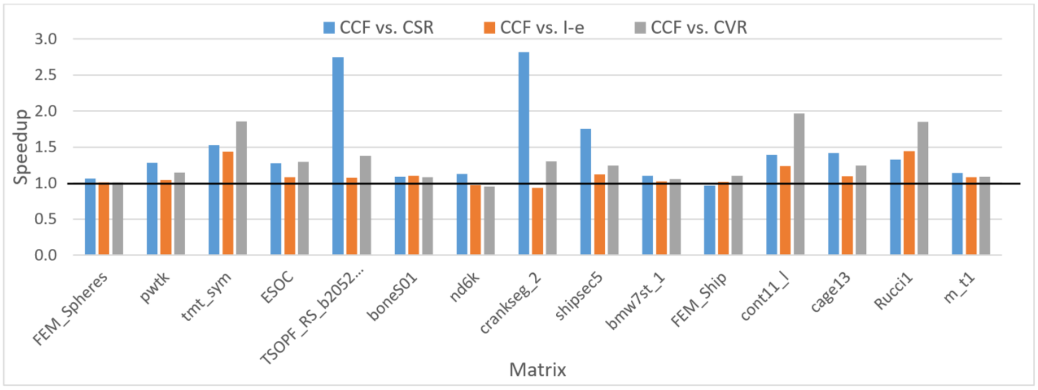


Figure 5.5: The speed improvement of SpMV CCF compared to the other three SpMV kernels for HPC cache-resident matrices (4/4).

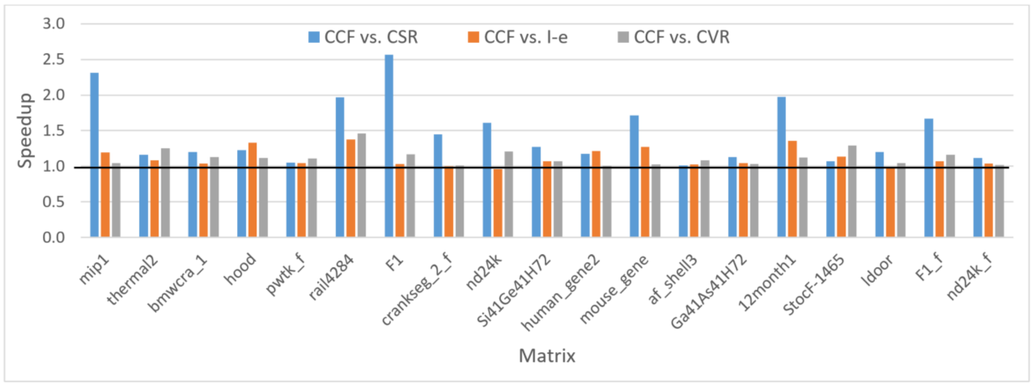


Figure 5.6: The speed improvement of SpMV CCF compared to the other three SpMV kernels for HPC partially-cache-resident matrices.

1.4x respectively.

Figure 5.10 shows the performance of CCF compared to the other three kernels for the DRAM-resident matrices. Compared to CSR, CCF significantly improves the performance of most of the matrices due to the alleviated load balancing. Furthermore, CCF is slightly faster than I-e and CVR for several matrices. However, we observe that CCF is slower than all of the other kernels for three matrices: road_central, wb-edu, and road_usa. In general, scale-free DRAM-resident matrices are highly irregular and also very large. CCF uses one set per thread as its sorting window (sets are mainly used for load balancing and false sharing elimination). For these irregular and large matrices, this sorting window (set) is quite large and sorting all the rows can reduce data locality. This is the reason behind low CCF performance for some matrices in this category. In chapter 6, we show that increasing the

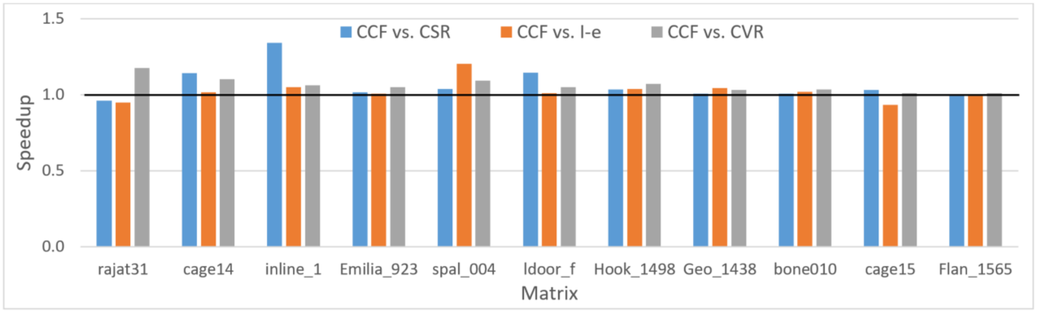


Figure 5.7: The speed improvement of SpMV CCF compared to the other three SpMV kernels for HPC DRAM-resident matrices.

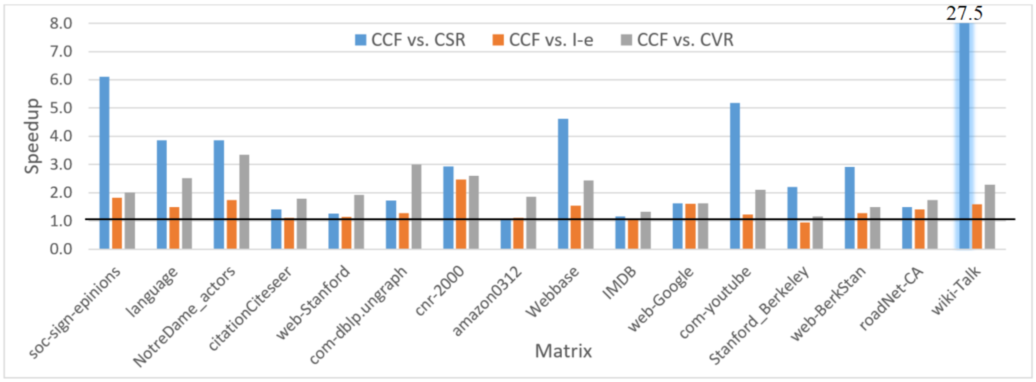


Figure 5.8: The speed improvement of SpMV CCF compared to the other three SpMV kernels for scale-free cache-resident matrices.

number of sets per thread reduces the sorting window size and improves the performance.

5.2.3 Very-long-row Matrices

Figure 5.11 shows the performance comparison between CCF and the other kernel for matrices with very long rows. CCF significantly outperforms CSR and I-e. However, CCF is slightly faster than CVR because CVR also mitigates the challenge of very long rows by strictly dividing the nonzero elements of the matrix evenly across threads, which means that rows can be partitioned among multiple threads. The geomeans of the speed improvement of CCF compared to CSR, I-e, and CVR are 4.8x, 2.1x, and 1.2x respectively.

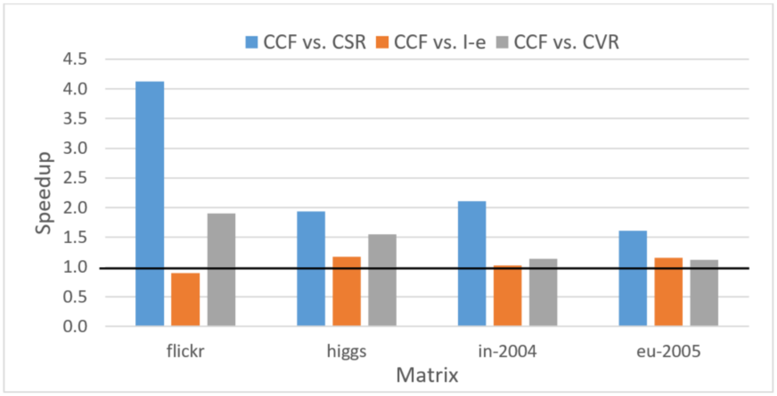


Figure 5.9: The speed improvement of SpMV CCF compared to the other three SpMV kernels for scale-free partially-cache-resident matrices.

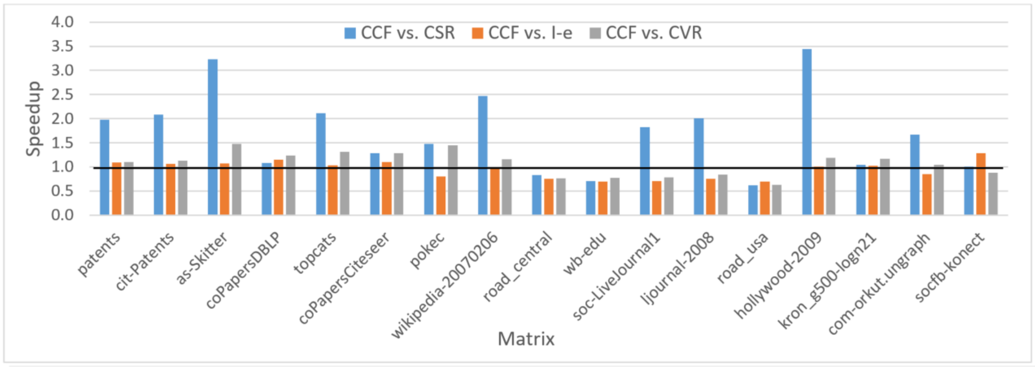


Figure 5.10: The speed improvement of SpMV CCF compared to the other three SpMV kernels for scale-free DRAM-resident matrices.

5.3 Performance Comparison on KNL

KNL has a different cache size than Skylake. The total cache size of KNL is 34 MB. Thus, the matrices that belong to each class of matrices are different on each platform. Figure 5.12 shows the matrix classification on the KNL. The number of cache-resident matrices on KNL is much lower than Skylake because of the huge difference in cache size between the two platforms. Descriptions of both categories are given in section 4.1.3.

5.3.1 HPC Matrices

The HPC matrices are divided into four categories: same-nnzr, cache-resident, partially-cache-resident, and DRAM-resident.

Figure 5.13 shows the performance comparison between CCF and the other

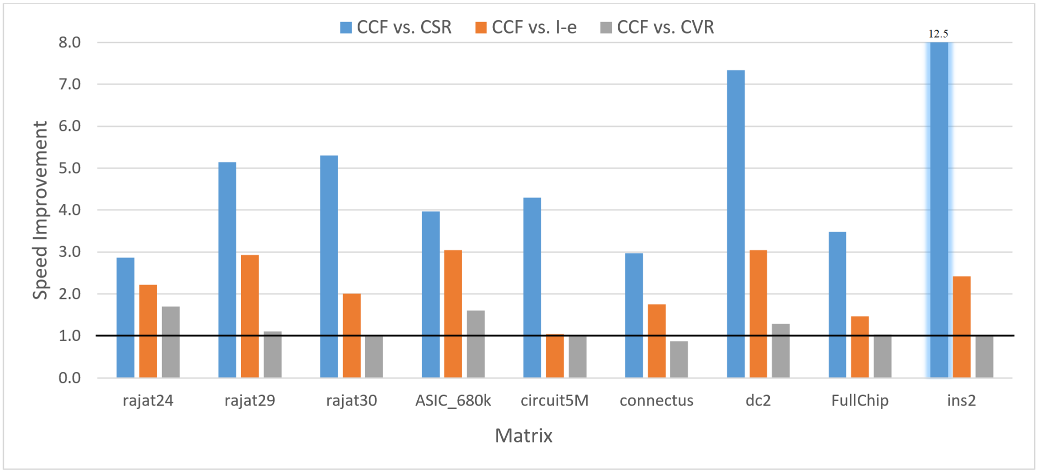


Figure 5.11: The speed improvement of SpMV CCF compared to the other three SpMV kernels for very-long-row matrices.

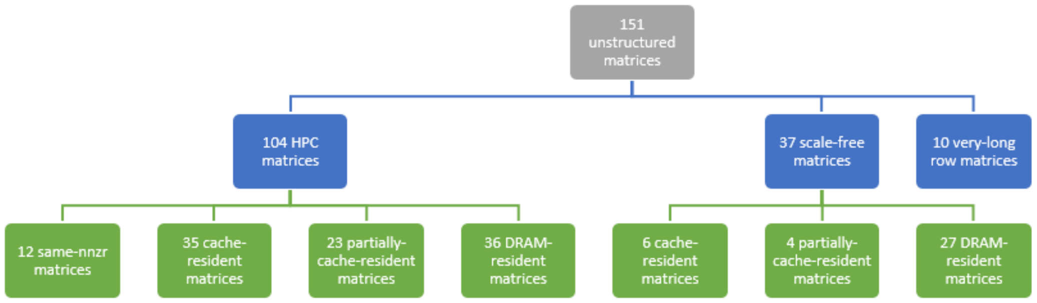

151 unstructured matrices
104 HPC matrices
37 scale-free matrices
10 very-long row matrices
12 same-nnzr matrices
35 cache-resident matrices
23 partially-cache-resident matrices
36 DRAM-resident matrices
6 cache-resident matrices
4 partially-cache-resident matrices
27 DRAM-resident matrices

Figure 5.12: Hierarchical grouping of the matrices using the 68-core KNL with 34 MB cache.

three kernels for the matrices with the same nnzr. CCF significantly outperforms CSR and CVR with a geomean of 2.0x each. I-e slightly outperforms CCF with a geomean of 1.0x.

For the cache-resident matrices, CCF exhibits a superior performance for most of the matrices compared to the other kernels. These matrices reside in the cache, which means that they are not bounded by the memory bandwidth. As discussed in chapter 4, CCF excels in exploiting the available vector units and creating a load balanced workloads. CCF significantly outperforms CSR and CVR with a geomean of 2.1x each. CCF outperforms I-e with a geomean of 1.4x. Figure 5.14 shows a comparison between the CCF and the other three kernels.

Partially-cache-resident matrices do not completely reside in the cache during the kernel execution. Therefore, CCF techniques are still effective

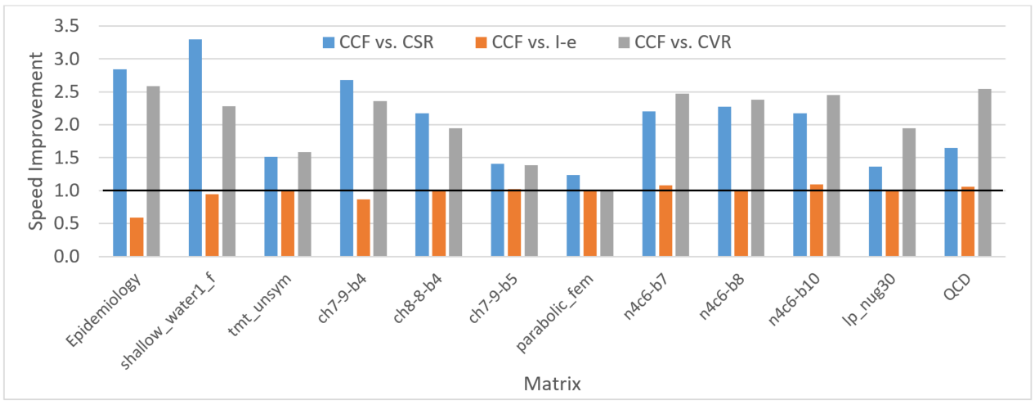


Figure 5.13: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the matrices with same nnzr on KNL.

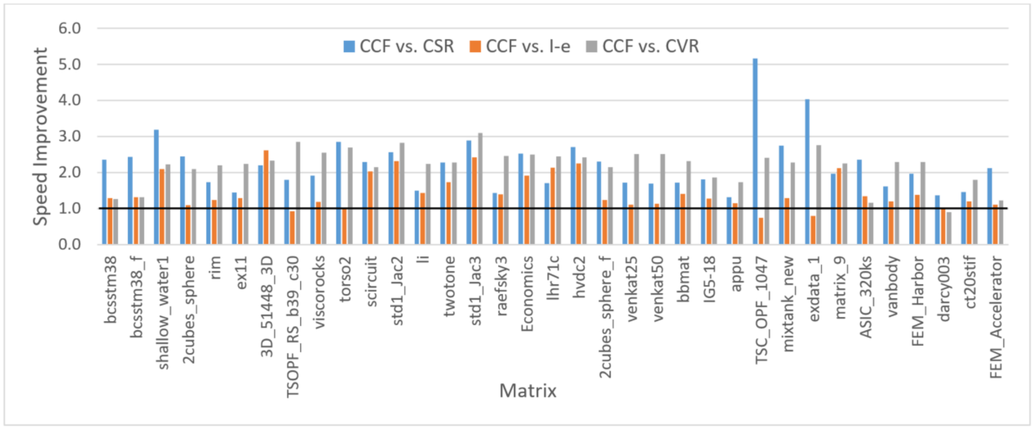


Figure 5.14: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the HPC cache-resident matrices on KNL.

but not as effective as for the cache-resident matrices. The geomeans of the speed improvement of CCF compared to CSR, I-e, and CVR are 1.3x, 1.1x, and 1.2x respectively. Figure 5.15 shows the comparison between the CCF and the other three kernels.

Figure 5.16 shows the comparison between CCF and the other three kernels for the DRAM-resident matrices. The geomeans of the speed improvement of CCF compared to CSR, I-e, and CVR are 1.3x, 1.0x, and 1.0x respectively. There are several matrices for which CSR, I-e and CVR outperform CCF. This is due to the poor data locality caused by sorting rows within sets, which are considered as a huge sorting window for the KNL small cache. In chapter 6, we will show some examples of how to improve the performance by increasing the number of sets per thread.

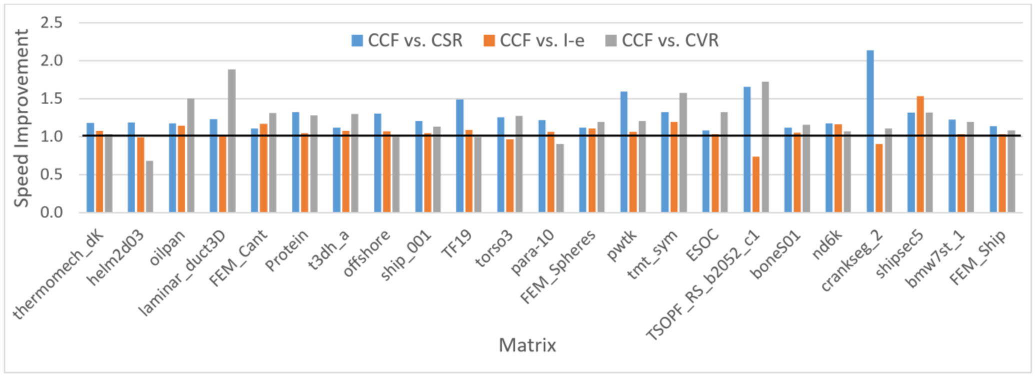


Figure 5.15: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the HPC partially-cache-resident matrices on KNL.

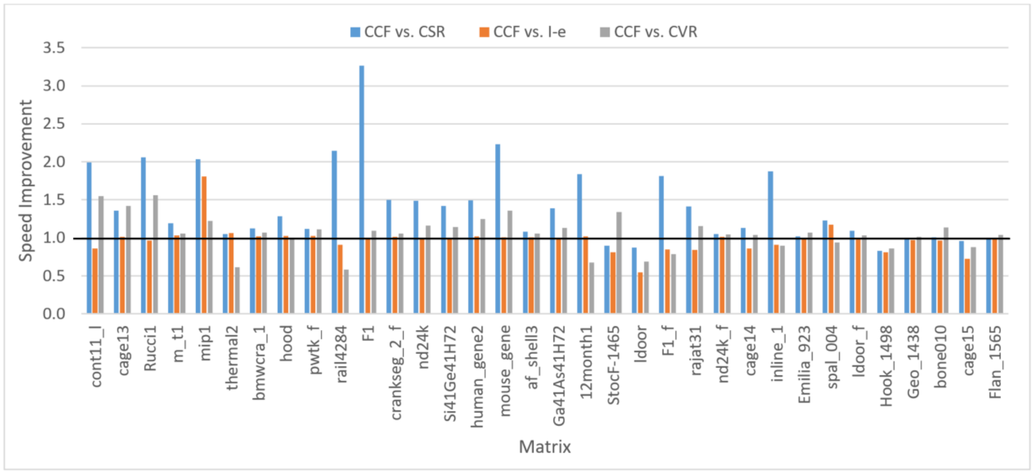


Figure 5.16: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the HPC DRAM-resident matrices on KNL.

5.3.2 Scale-free Matrices

The scale-free matrices are divided into three categories: cache-resident, partially-cache-resident, and DRAM-resident.

For the cache-resident matrices, CCF achieves a higher performance than CSR for all the matrices with a geomean of 2.4x. Compared to I-e, CCF is faster for 4 out of 5 matrices. The geomean for all the matrices is 1.3x. CVR and CCF perform comparably to each other; however, the geomean of CCF performance compared to CVR is 1.1x. Figure 5.17 shows the comparison between CCF and the other three kernels.

Figure 5.18 shows a comparison between CCF and the other three kernels for the partially-cache-resident matrices, CCF outperforms CSR and I-e for

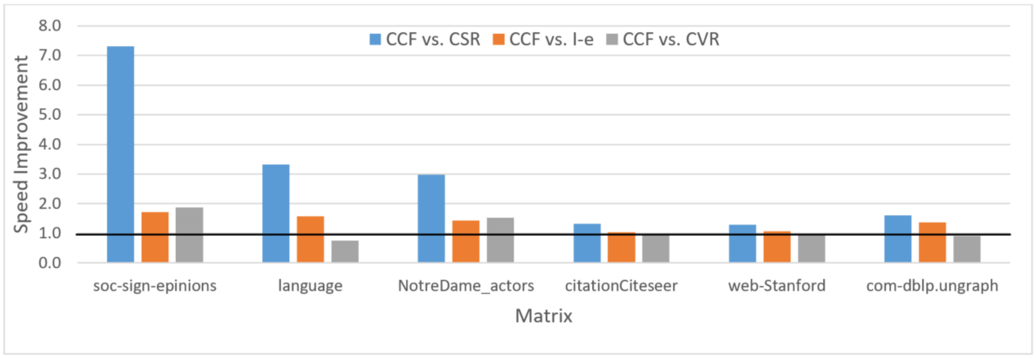


Figure 5.17: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the scale-free cache-resident matrices on KNL.

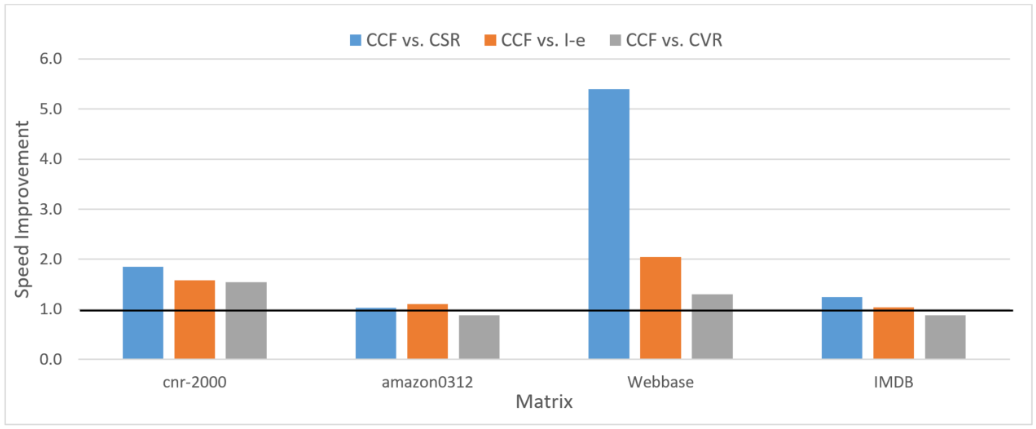


Figure 5.18: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the scale-free partially-cache-resident matrices on KNL.

all the matrices. The geomeans of the speed improvement of CCF compared to CSR, I-e, and CVR are 1.9x, 1.4x, and 1.1x respectively.

For the DRAM-resident matrices, CCF outperforms CSR and I-e for most of the matrices with geomeans of 2.3x and 1.1x respectively. However, we note that CVR is superior to the other three kernels for most of the matrices. The geomean of CVR performance compared to CCF is 1.4x (see figure 5.19).

5.3.3 Very-long-row Matrices

Figure 5.20 shows the performance of CCF compared to CSR, I-e, and CVR for the very-long-row matrices on KNL. Note that one matrix (Raj1) is added to very-long-row matrices because KNL has more threads than the dual Sky-

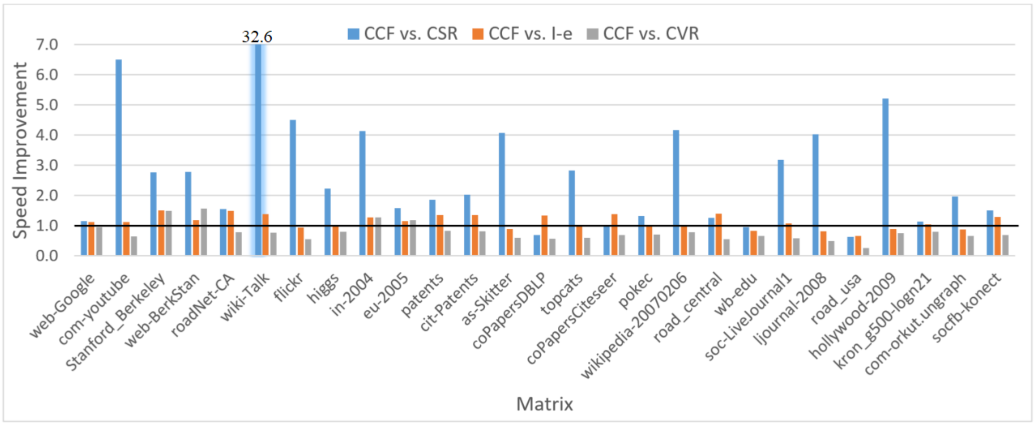


Figure 5.19: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the scale-free DRAM-resident matrices on KNL.

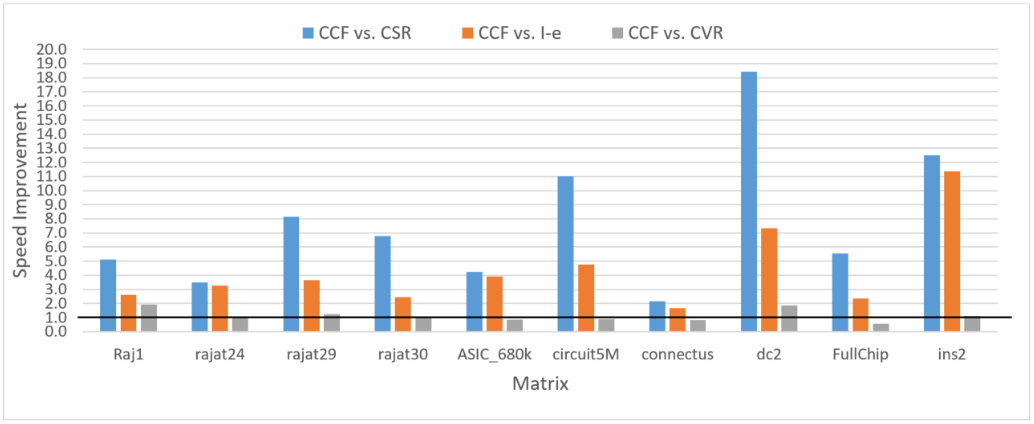


Figure 5.20: The speed improvement of SpMV CCF compared to the other three SpMV kernels for the very-long-row matrices on KNL.

lake. CCF significantly outperforms CSR and I-e for all the matrices with geomeans of 6.5x and 3.7x. CCF is faster than CVR for 50% of the matrices. The geomean of CCF performance compared to CVR is 1.1x.

5.4 Performance Summary on KNL and Skylake

5.4.1 Achieved GFLOP/s

Table 5.2 shows a summary of the achieved performance of each kernel on each platform. CCF has the highest maximum, average, and minimum GFLOP/s on both platforms. CCF achieved 3.5% of KNL and the dual

Table 5.2: Maximum, average, and minimum delivered GFLOP/s for our 151 unstructured matrices on KNL and Skylake.

	KNL			Skylake		
kernel	Max	Avg.	Min	Max	Avg.	Min
CSR	73.12	25.09	0.16	145.51	47.93	0.66
I-e	102.59	36.07	1.24	169.53	60.26	4.80
CVR	53.71	28.74	1.64	103.22	48.07	3.16
CCF	104.96	41.27	2.06	209.71	79.90	5.25

Table 5.3: The performance of CCF compared to CSR on KNL and Skylake.

	KNL			Skylake		
	#(%) of matrices	Max	Avg.	#(%) of matrices	Max	Avg.
CCF faster than CSR	137(90%)	32.6x	2.7x	135(89%)	27.5x	2.3x
CCF same as CSR	8 (6%)	-	-	13 (9%)	-	-
CSR faster than CCF	6 (4%)	1.6x	1.3x	3(2%)	1.6x	1.4x

Skylake peak performance.

5.4.2 The Effectiveness of CCF Compared to CSR, I-e, and CVR on KNL and Skylake

Tables 5.3, 5.4, and 5.5 summarize the speed improvement achieved by CCF compared to each of the other three kernels. The effectiveness of CCF compared to the other three kernels on Skylake is noticeably higher than on KNL. This is because Skylake has a larger cache which increases the number of matrices that belong to the cache-resident and partially-cache-resident categories. As we observed previously, CCF is superior to the other kernels for cache-resident and partially-cache-resident matrices. For the HPC DRAM-resident matrices, CCF, I-e, and CVR have similar performance on Skylake and KNL. On KNL, CVR outperformed all other kernels for most scale-free DRAM-resident matrices.

5.5 Skylake vs. KNL

To compare KNL and Skylake, we took the maximum delivered performance by any kernel on the two platforms and compared the performance of KNL

Table 5.4: The performance of CCF compared to I-e on KNL and Skylake.

	KNL			Skylake		
	#(%) of matrices	Max	Avg.	#(%) of matrices	Max	Avg.
CCF faster than I-e	81(53%)	11.4x	1.8x	109(73%)	3.0x	1.5x
CCF same as I-e	45(30%)	-	-	30(19%)	-	-
I-e faster than CCF	25(17%)	1.8x	1.2x	12(8%)	1.4x	1.2x

Table 5.5: The performance of CCF compared to CVR on KNL and Skylake.

	KNL			Skylake		
	#(%) of matrices	Max	Avg.	#(%) of matrices	Max	Avg.
CCF faster than CVR	95(63%)	3.1x	1.7x	124(81%)	4.2x	1.8x
CCF same as CVR	12(8%)	-	-	20(13%)	-	-
CVR faster than CCF	44(29%)	3.8x	1.4x	7(5%)	1.6x	1.3x

Table 5.6: The performance of KNL compared to Skylake using the best/fastest of the four kernels (CSR, I-e, CVR, and CFF) for each matrix.

	#(%) of matrices	Max	Avg.
KNL faster than Skylake	24(16%)	2.0x	1.4x
KNL same as Skylake	8	-	-
Skylake faster than KNL	119(79%)	4.4x	2.3x

and Skylake for each matrix. Table 5.6 summarizes the performance comparison of KNL and Skylake. It is apparent that Skylake outperforms KNL for most of the matrices using any of the four SpMV kernels. Skylake is faster than KNL for all the matrices that are cache-resident on Skylake. This is expected because Skylake is computationally more powerful than KNL (the dual Skylake peak performance is twice the KNL peak performance). On the other hand, KNL is faster than Skylake for 24 matrices. These matrices are either partially-cache-resident or DRAM-resident matrices. This is also expected since the memory bandwidth of KNL is almost twice that of dual Skylake.

5.6 Preprocessing Overhead

In this section, we measure the overhead of CCF preprocessing that transforms a matrix into the CCF format (described in chapter 3). We use the dual 24-core Skylake (table 5.1) and one thread per core (48 threads). For each matrix in our set, we divide the preprocessing time by a single SpMV

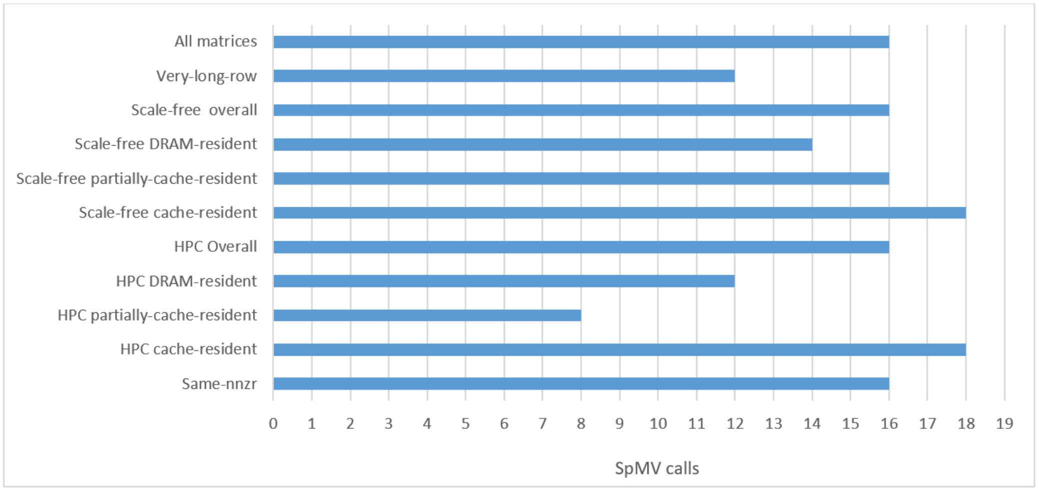


Figure 5.21: Number of SpMV CCF calls that amortize the preprocessing overhead on Skylake.

call time. This gives the number of SpMV calls that amortize the preprocessing overhead. Figure 5.21 shows the average number of SpMV calls that amortize the preprocessing overhead for each category of matrices on Skylake. For our set of matrices, the average preprocessing overhead is 16 SpMV calls, which is a small overhead for real world applications that call SpMV kernels hundreds or thousands of times.

5.7 Summary

In this chapter, we compared the performance of CCF with CSR, I-e, and CVR on Skylake and KNL. We showed that CCF has a superior performance compared to the other kernels for cache-resident and partially-cache-resident matrices. For HPC DRAM-resident matrices, CCF, I-e, and CVR have similar performance on Skylake and KNL. On KNL, CVR outperformed all other kernels for most scale-free DRAM-resident matrices. Moreover, we showed that the dual Skylake is faster than KNL for all the matrices that are cache-resident on Skylake due to its higher computational power. KNL is faster than Skylake for 24 matrices (partially-cache-resident or DRAM-resident matrices). This is due to KNL's higher memory bandwidth. Lastly, we showed that the average preprocessing overhead is 16 SpMV calls.

CHAPTER 6

CCF AUTO-TUNING

In this chapter, we describe the CCF auto-tuner and show a revised performance comparison between the auto-tuned CCF and the other three kernels: CSR, I-e, and CVR on Skylake and KNL.

6.1 Parameters for Auto-tuning

CCF has two main parameters whose values partake in the delivered performance: a) the number of assigned sets per thread (SPT) and b) the chunk size (CS), which determines the maximum number of rows that a chunk can accommodate. The default value of the chunk size is SIMDW of the target architecture. CCF also can use multiples of SIMDW as chunk sizes but SpMV CCF must be aware of the chunk size used in the preprocessing step. For the SPT parameter, the default value is one. This is due to the intuition behind using sets, which is load balancing. Tuning SPT and CS improves performance for several matrices. Table 6.1 shows examples illustrating that tuning these parameters can improve the performance of SpMV CCF significantly.

CCF auto-tuner seamlessly tries all the possible parameter combinations and delivers the optimal parameter values that CCF uses to achieve the highest performance on a given architecture.

6.2 Auto-tuner Description

Figure 6.1 shows a pseudocode of the CCF auto-tuner skeleton. For a given matrix, the CCF auto-tuner exhaustively tries all combinations of the parameters and identifies their optimal values. Using each combination of parameters, the CCF auto-tuner calls the CCF preprocessor routine. Furthermore,

Table 6.1: Example on how the optimal choice of CCF parameters can improve the performance of CCF SpMV kernel on dual 24-core Skylake Platinum.

Matrix	CCF GFLOP/s using the default parameter values	CCF GFLOP/s using tuned parameter values	Speed improvement of the auto-tuned CCF kernel	Optimal SPT	Optimal CS
road_usa	6.65	11.79	1.8x	4	32
soc-LiveJournal1	11.95	19.03	1.6x	32	32
wb-edu	15.1	22.47	1.5x	32	8
connectus	44.77	65.98	1.5x	32	16
rajat30	49.51	66.52	1.3x	8	16

the preprocessor automatically detects whether or not the matrix has very long rows and creates CCF data structures accordingly. Using the given parameters and whether or not the matrix has very long rows, the auto-tuner executes the proper SpMV kernel multiple times and records the elapsed execution time. After CCF auto-tuner tries all possible combinations, the combination with the least execution time represents the optimal values of the parameters.

For the SPT parameter, the auto-tuner searches the values: 1, 2, 4, 8, 16, 32, and 64. We observed no benefit of using any SPT value greater than 64 for our current set of matrices and the two platforms (Skylake and KNL). SpMV CCF kernel does not need to know the SPT value in advance to execute; OpenMP API automatically detects the number of sets and divides them evenly across threads. For the CS parameter, the auto-tuner uses the values: 8, 16, 24, and 32 for its search space. There is a different SpMV kernel for each CS value. Using kernels that are specialized for particular chunk sizes avoids extra loop overhead in SpMV CCF. Based on the values of CS and the very long rows decision (see figure 6.1), CCF auto-tuner selects from the following SpMV kernels:

1. For matrices without very long rows:
    (a) Spmv-ccf-8: supports CCF data structures with CS = 8.
    (b) Spmv-ccf-16: supports CCF data structures with CS = 16.
    (c) Spmv-ccf-24: supports CCF data structures with CS = 24.
    (d) Spmv-ccf-32: supports CCF data structures with CS = 32.
2. For matrices with very long rows:
    (a) Spmv-ccf-8-long: supports CCF data structures with CS = 8.

```
setsChoice = 7
chunkSizeChoice = 4
setsPerThread = {1, 2, 4, 8, 16, 32, 64}
chunkSize = {8, 16, 24, 32}

nt = 10 //Number of trials per each combination of parameters
For i= 0 to setsChoice:
  For j=0; to chunkSizeChoice:
       HasLongRows = Ccf-preprocess(setsPerThread[i], chunkSize[j])
       If HasLongRows Equals true:
             If chunkSize[j] Equals 8:
               start = start-timing
                Spmv-ccf-8-long(times:nt)
               end = end-timing
             Else If chunkSize[j] Equals 16:
             /*…. tries 16, 24, 32*/

             End IF
       Else:
             If chunkSize[j] Equals 8:
               start = start-timing
                Spmv-ccf-8(times:nt)
               end = end-timing
             Else If chunkSize[j] Equals 16:
             /*… tries 16, 24, 32*/

             End If


       LogTime(setsPerThread[i], chunkSize[j],end - start)

  End For

End For

seed = logtimes[0].time
OptimalSetsNumber = logTimes[0].sets
OptimalChunkSize = logTimes[0].chunkSize
For i = 1 to setsChoice * chunkSizeChoice:
       speedup = seed/times[i]
       If speedup GreaterOrEquals 1.05:
             seed = times[i].time
             OptimalSetsNumber = logTimes[i].sets
             OptimalChunkSize = logTimes[i].chunkSize
       End If
End For
```

Figure 6.1: The skeleton of the CCF auto-tuner pseudocode.

(b) Spmv-ccf-16-long: supports CCF data structures with CS = 16.

(c) Spmv-ccf-24-long: supports CCF data structures with CS = 24.

(d) Spmv-ccf-32-long: supports CCF data structures with CS = 32.

6.3 Auto-tuner Performance Analysis

In this section, we show some cache profiling data for three matrices that achieved higher performance when using the CCF auto-tuner on the Skylake. “Perf” tool is used [27]. Tuning the SPT and CS parameters improves the

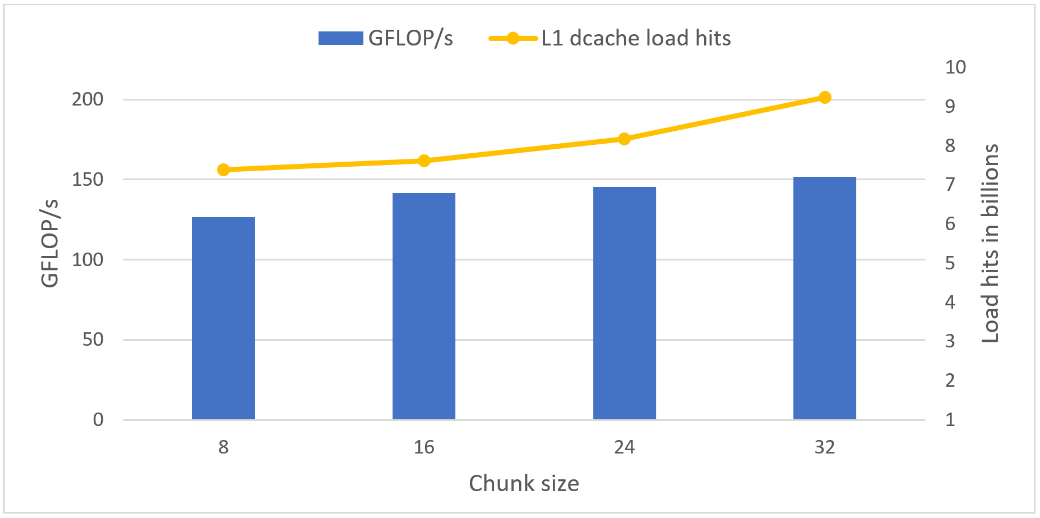


Figure 6.2: Performance and L1 dcahce hits as the CS parameter varies for matrix TSOPF_RS_b39_c30.

data locality of the SpMV kernel for these matrices. For simplicity, we study the tuning of each parameter separately.

6.3.1 Tuning the Chunk Size and Fixing the Number of Sets per Thread to One

By increasing the chunk size, the number of rows per chunk increases, which means that processing in the column direction involves more rows. For some matrices, processing more elements in the column direction improves data locality. Figure 6.2 shows how the number of L1 dcache hits increases and performance improves as the CS parameter increases for matrix TSOPF_RS_b39_c30. Figure 6.3 shows performance and LLC load misses for matrix rajat29. The LLC misses decrease until CS=24, then start to increase again. This, of course, reflects on the delivered performance. We used different performance counters in figures 6.2 and 6.3 because the CCF kernel exhibits different cache traffic behavior for each matrix. For example, for matrix rajat29 we observed that the L1 cache traffic does not significantly change as CS is increased. Furthermore, matrix rajat29 also benefits from tuning SPT, but we do not show it here.

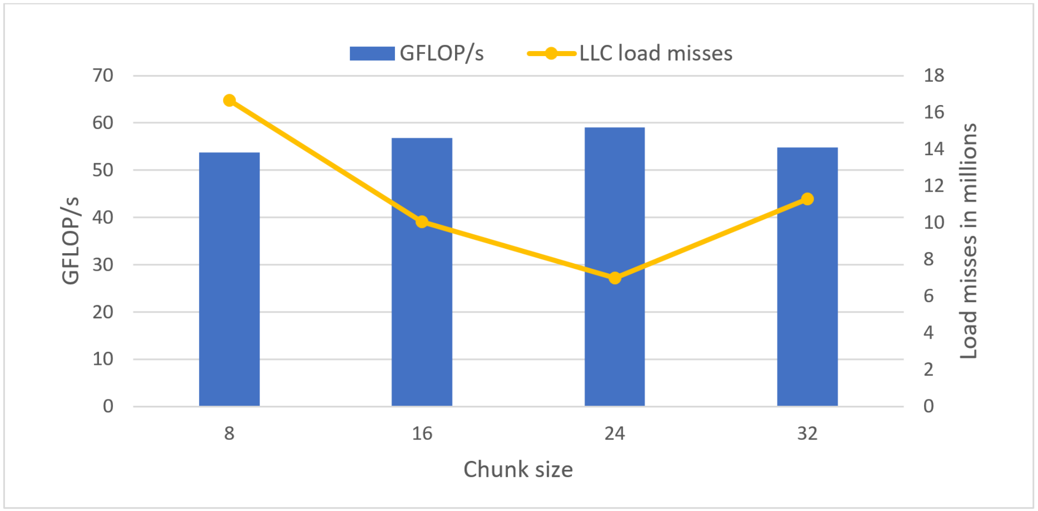


Figure 6.3: Performance and LLC misses as the CS parameter varies for matrix rajat29.

6.3.2 Tuning the Number of Sets per Thread and Fixing the Chunk Size to Eight

CCF divides the matrix into sets that have the same number of nonzero elements for load balancing. Moreover, each thread sorts rows with the set boundary to avoid false sharing of the y vector elements. For most of the matrices, CCF outperforms CSR, I-e, and CVR kernels. However, we note that CCF performance is very low compared to the other three kernels for some of the scale-free DRAM matrices. These matrices are highly irregular and very large. Hence, sorting rows with one set per thread is considered as a huge sorting window that reduces the inherent data locality of the neighboring nonzero elements. The solution is to increase the number of sets per each thread which reduces the boundaries of the sorting windows. Figure 6.4 illustrates how LLC load misses decrease and performance improves as the number of sets per thread increases. The auto-tuned CCF outperforms CSR, I-e, and CVR by 1.1x, 1.2x, and 1.1x respectively.

6.4 Performance Improvement Summary

In this section, we show revised summary tables that compare the performance of the auto-tuned CCF kernel with CSR, I-e, and CVR kernels on Skylake and KNL. We also show a summary table that compares the auto-

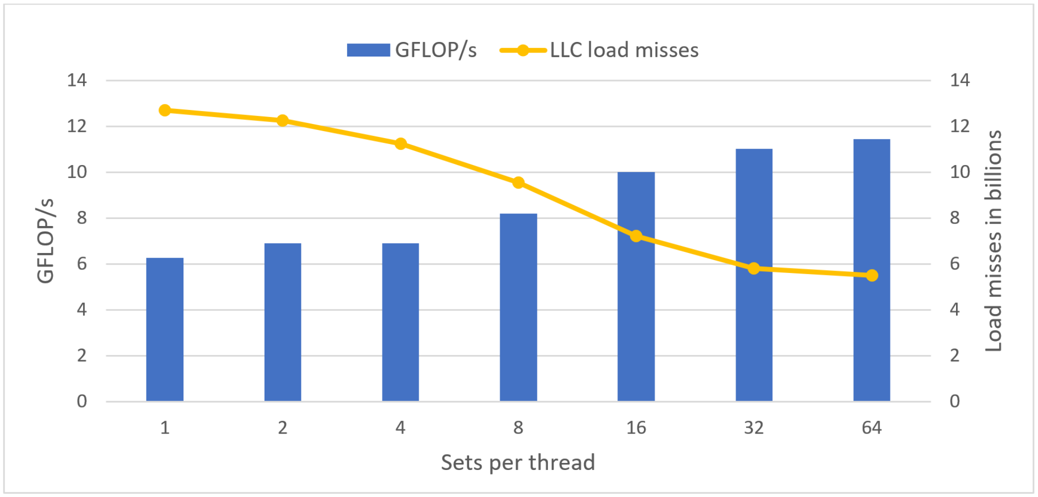


Figure 6.4: Performance and LLC misses as the CS parameter varies for matrix road_usa.

tuned CCF kernel with the default CCF kernel on Skylake and KNL. In this section we refer to the auto-tuned CCF kernel as CCF-AT.

6.4.1 Auto-tuned CCF vs. CSR, I-e, and CVR

Tables 6.2, 6.3, and 6.4 show a summary comparison between CCF-AT and the other three kernels on KNL and Skylake. Compared to the summary tables presented in chapter 5 (tables 5.3, 5.4, and 5.5), we note that CCF-AT improved the performance of CCF on both platforms. On Skylake, CVR and CSR are not faster than CCF-AT for any matrix. Compared to I-e, I-e is only faster for two matrices with 10% improvement. This shows that CCF-AT is the winning kernel for all matrices in our set on Skylake. On KNL, CCF-AT is significantly faster than CSR. CSR is only faster for one matrix. Compared to I-e and CVR, CCF-AT outperforms both kernels for high percentages of matrices. However, we observe that I-e is faster than CCF-AT for 8 matrices, 7 of which are HPC matrices. This indicates that I-e is taking advantage of the regular structure of these matrices. Furthermore, CVR outperforms CCF-AT for 37 matrices, 22 of which are scale-free DRAM-resident matrices. CVR is more effective than CCF-AT for the scale-free DRAM-resident matrices on KNL. CVR incurs high overhead tracking row accumulation and indexing the output vector, however, processing rows in their original order is more effective on KNL.

Table 6.2: The performance of the auto-tuned CCF kernel (CCF-AT) compared to the CSR kernel on KNL and Skylake.

	KNL			Skylake		
	#(%) of matrices	Max	Avg.	# (%) of Matrices	Max	Avg.
CCF-AT faster than CSR	144(95%)	32.7x	2.8x	142(94%)	32.5x	2.5x
CCF-AT same as CSR	6 (4%)	-	-	9 (6%)	-	-
CSR faster than CCF-AT	1	1.1x	1.1x	0	-	-

Table 6.3: The performance of the auto-tuned CCF kernel (CCF-AT) compared to the I-e kernel on KNL and Skylake.

	KNL			Skylake		
	#(%) of matrices	Max	Avg.	# (%) of Matrices	Max	Avg.
CCF-AT faster than I-e	91(60%)	11.5x	1.8x	120(80%)	3.9x	1.5x
CCF-AT same as I-e	52(34%)	-	-	29(19%)	-	-
I-e faster than CCF-AT	8(6%)	1.7x	1.2x	2(1%)	1.1x	1.1x

Figure 6.5 shows how CCF and CCF-AT compare to CVR for the scale-free DRAM-resident matrices on KNL. It is obvious that CCF-AT improved the performance of CCF for most of the matrices. The geomean of CVR performance compared to CCF is reduced from 1.4x to 1.1x using the auto-tuner.

6.4.2 Auto-tuned CCF vs. the Default CCF

Table 6.5 demonstrates the performance improvement achieved by auto-tuning the CCF storage format and its kernel on KNL and Skylake. The CCF auto-tuner improves the performance of more than 50 matrices with an average speed improvement of 1.2x on both platforms.

Table 6.4: The performance of the auto-tuned CCF kernel (CCF-AT) compared to the CVR kernel on KNL and Skylake.

	KNL			Skylake		
	#(%) of matrices	Max	Avg.	# (%) of Matrices	Max	Avg.
CCF-AT faster than CVR	105(70%)	3.6x	1.7x	139 (92%)	4.5x	1.8x
CCF-AT same as CVR	9(6%)	-	-	12 (8%)	-	-
CVR faster than CCF-AT	37(24%)	1.6x	1.2x	0	-	-

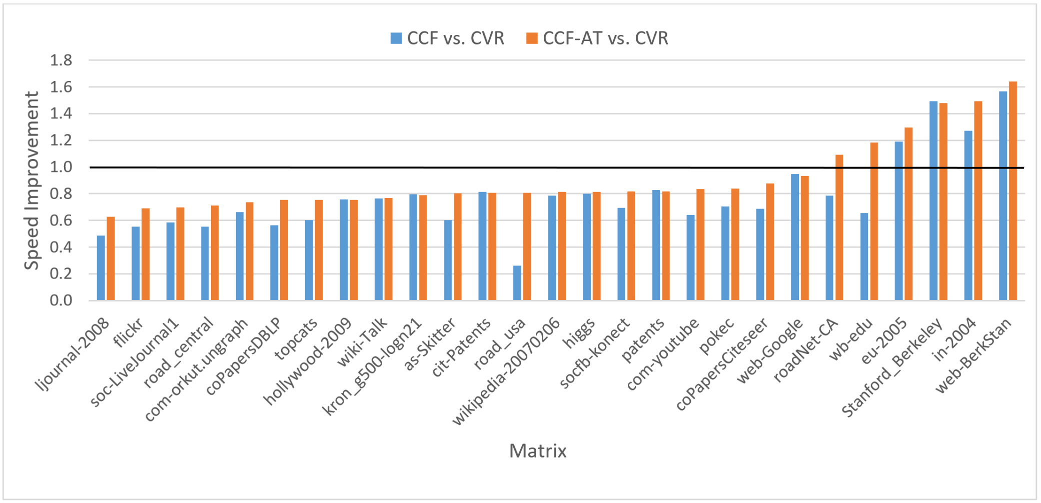


Figure 6.5: The performance of CCF and CCF-AT compared to CVR for the scale-free DRAM-resident matrices on KNL.

Table 6.5: The performance of the auto-tuned CCF kernel compared to the default CCF kernel on KNL and Skylake.

	KNL			Skylake		
	#(%) of matrices	Max	Avg.	# (%) of Matrices	Max	Avg.
CCF-AT faster than CCF	52(34%)	3.1x	1.2x	59(40%)	1.8x	1.2x
CCF-AT same as CCF	99(66%)	-	-	92 (60%)	-	-
CCF faster than CCF-AT	0	-	-	0	-	-

CHAPTER 7

CONCLUSION

This thesis presented our novel sparse matrix compressed chunk storage format (CCF) and its optimized SpMV kernel for Intel many-core and Intel multi-core platforms. CCF improves the SIMD efficiency and load balancing using four techniques: collecting rows in chunks, dividing nonzero elements evenly across threads, using CSR processing when the number of rows in a bin is less than SIMDW, and partitioning very long rows.

We presented a thorough performance analysis of CCF by breaking down CCF into incremental versions and classifying matrices into categories based on their applications and residency (cache or DRAM). We compared each incremental version with its predecessor per each matrix category and explained the strengths and weaknesses in CCF.

We compared the performance of our SpMV CCF kernel with Intel MKL 2018u1 SpMV CSR, Intel MKL 2018u2 Inspector-executor SpMV CSR, and SpMV CVR kernels on two platforms: a dual 24-core Skylake and a 68-core KNL. On the dual 24-core Skylake, and compared to MKL SpMV CSR, our kernel achieves superior execution throughputs for 135 matrices (89%) with an average speed improvement of 2.3x and maximum speed improvement of 27.5x. Our kernel outperforms MKL Inspector-executor SpMV CSR for 109 matrices (73%) with an average speed improvement of 1.5x and maximum speed improvement of 3.0x. Moreover, CCF outperforms CVR for 81% of the matrices with an average speed improvement of 1.8x and maximum speed improvement of 4.2x. On the 68-core KNL, CCF achieves high average and maximum speed improvements compared to the other three kernels but for slightly smaller percentages of matrices.

We studied the impact of tuning the number of sets per thread and the chunk size on performance and presented the CCF auto-tuner that automatically chooses the optimal values for these parameters. The auto-tuned CCF improved the performance for more than 50 matrices on Skylake and KNL with and average improvement of 1.2x. On the dual 24-Skylake, the auto-tuned CCF is guaranteed to achieve the best possible performance compared to the other kernels.

REFERENCES

[1] Y. Saad, “Krylov subspace methods on supercomputers,” *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 6, pp. 1200–1232, 1989.

[2] K. Yelick, “poski: An extensible autotuning framework to perform optimized spmvs on multicore architectures,” Ph.D. dissertation, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2009.

[3] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, “A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units,” *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[4] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang et al., “Bigdatabench: A big data benchmark suite from internet services,” in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on.* IEEE, 2014, pp. 488–499.

[5] A.-L. Barabási and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

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

[7] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, “Efficient sparse matrix-vector multiplication on x86-based many-core processors,” in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing.* ACM, 2013, pp. 273–282.

[8] W. Liu and B. Vinter, “Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication,” in *Proceedings of the 29th ACM on International Conference on Supercomputing.* ACM, 2015, pp. 339–350.

[9] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization.* ACM, 2018, pp. 149–162.

[10] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi.* Springer, 2014, pp. 167–188.

[11] W. Abu-Sufah and A. A. Karim, "Auto-tuning of sparse matrix-vector multiplication on graphics processors," in *International Supercomputing Conference.* Springer, 2013, pp. 151–164.

[12] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[13] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007.

[14] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* ACM, 2009, p. 18.

[15] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International.* IEEE, 2011, pp. 721–733.

[16] W. Abu-Sufah and A. A. Karim, "An effective approach for implementing sparse matrix-vector multiplication on graphics processing units," in *IEEE 14th International Conference on High Performance Computing.* IEEE, 2012, pp. 453–460.

[17] J. Mellor-Crummey and J. Garvin, "Optimizing sparse matrix–vector product computations using unroll and jam," *The International Journal of High Performance Computing Applications*, vol. 18, no. 2, pp. 225–236, 2004.

[18] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *International Conference on High Performance Computing and Communications.* Springer, 2005, pp. 807–816.

[19] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[20] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing.* ACM, 1999, p. 30.

[21] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *International Conference on High-Performance Embedded Architectures and Compilers.* Springer, 2010, pp. 111–125.

[22] G. E. Blelloch, M. A. Heroux, and M. Zagha, "Segmented operations for sparse matrix computation on vector multiprocessors," Carnegie-Mellon Univ School of Computer Science, Tech. Rep., 1993.

[23] J. Torrellas, H. Lam, and J. L. Hennessy, "False sharing and spatial locality in multiprocessor caches," *IEEE Transactions on Computers*, vol. 43, no. 6, pp. 651–663, 1994.

[24] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[25] J. Leskovec and A. Krevl, "Snap datasets: Stanford network analysis platform," 2014. [Online]. Available: http://snap.stanford.edu/data

[26] Intel Corporation, "Performance benchmarks — intel$^{(R)}$ mkl — intel$^{(R)}$ software," https://software.intel.com/en-us/mkl/features/benchmarks, (Accessed on 04/05/2018).

[27] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, 2013.

[28] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson et al., "Xsede: accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

APPENDIX A

THE UNSTRUCTURED MATRICES USED FOR EVALUATION

A.1 Same-nnzr Matrices

Table A.1: Same-nnzr matrices.

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr
Epidemiology	2,100,225	525,825	525,825	4	0	4
shallow_water1_f	327,680	81,920	81,920	4	0	4
tmt_unsym	4,584,801	917,825	917,825	5	0	5
ch7-9-b4	1,587,600	317,520	105,840	5	0	5
ch8-8-b4	1,881,600	376,320	117,600	5	0	5
ch7-9-b5	2,540,160	423,360	317,520	6	0	6
parabolic_fem	3,674,625	525,825	525,825	7	0	7
n4c6-b7	1,305,720	163,215	104,115	8	0	8
n4c6-b8	1,790,055	198,895	163,215	9	0	9
n4c6-b10	1,456,422	132,402	186,558	11	0	11
lp_nug30	1,567,800	52,260	379,350	30	0	30
QCD	1,916,928	49,152	49,152	39	0	39

A.2 HPC Matrices

The matrices in this section are sorted by the CSR data structure size. The following capital letters indicate the category of the matrix on each platform:

- L: the matrix has very-long rows
- C: the matrix is cache-resident
- P: the matrix is partially-cache-resident
- D: the matrix is DRAM-resident

Table A.2: HPC matrices.

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr	Category on Skylake	Category on KNL
bcsstm38	7,842	8,032	8,032	1.00	1.30	15	C	C
bcsstm38_f	10,485	8,032	8,032	1.30	1.90	20	C	C
shallow_water1	204,800	81,920	81,920	2.50	0.80	4	C	C
dc2	766,396	116,835	116,835	6.60	361.50	114,190	L	L
2cubes_sphere	874,378	101,492	101,492	8.60	3.80	29	C	C
rim	1,014,951	22,560	22,560	45.00	26.60	112	C	C
ex11	1,096,948	16,614	16,614	66.00	16.20	90	C	C
3D_51448_3D	1,056,610	51,448	51,448	20.50	28.40	5,671	C	C
TSOPF_RS_b39_c30	1,079,986	60,098	60,098	18.00	14.00	32	C	C
viscorocks	1,162,244	37,762	37,762	30.80	7.70	42	C	C
torso2	1,033,473	115,967	115,967	8.90	0.60	10	C	C
scircuit	958,936	170,998	170,998	5.60	4.40	353	C	C
std1_Jac2	1,248,731	21,982	21,982	56.80	145.60	898	C	C
li	1,350,309	22,695	22,695	59.50	29.20	108	C	C
twotone	1,224,224	120,750	120,750	10.10	15.00	185	C	C
std1_Jac3	1,455,848	21,982	21,982	66.20	169.30	1,030	C	C
raefsky3	1,488,768	21,200	21,200	70.20	6.30	80	C	C
Economics	1,273,389	206,500	206,500	6.20	4.40	44	C	C
lhr71c	1,528,092	70,304	70,304	21.70	26.30	63	C	C
hvdc2	1,347,273	189,860	189,860	7.10	3.80	60	C	C
Raj1	1,302,464	263,743	263,743	4.90	88.30	40,468	C	L
2cubes_sphere_f	1,647,264	101,492	101,492	16.20	2.70	31	C	C
venkat25	1,717,792	62,424	62,424	27.50	2.30	44	C	C
venkat50	1,717,792	62,424	62,424	27.50	2.30	44	C	C
bbmat	1,771,722	38,744	38,744	45.70	38.40	126	C	C
IG5-18	1,790,490	47,894	41,550	37.40	32.90	120	C	C
appu	1,853,104	14,000	14,000	132.40	36.50	294	C	C
TSC_OPF_1047	2,016,902	8,140	8,140	247.80	323.60	1,526	C	C
mixtank_new	1,995,041	29,957	29,957	66.60	38.30	154	C	C
exdata_1	2,269,501	6,001	6,001	378.20	649.60	1,503	C	C
matrix_9	2,121,550	103,430	103,430	20.50	17.80	4,057	C	C
ASIC_320ks	1,827,807	321,671	321,671	5.70	7.90	412	C	C
vanbody	2,336,898	47,072	47,072	49.60	17.60	232	C	C
FEM_Harbor	2,374,001	46,835	46,835	50.70	27.80	145	C	C
rajat24	1,948,235	358,172	358,172	5.40	180.10	105,296	L	L
darcy003	2,101,242	389,874	389,874	5.40	2.00	7	C	C
ct20stif	2,698,463	52,329	52,329	51.60	17.00	207	C	C
FEM_Accelerator	2,624,331	121,192	121,192	21.70	13.80	81	C	P
thermomech_dK	2,846,228	204,316	204,316	13.90	1.40	20	C	P
ins2	2,751,484	309,412	309,412	8.90	590.40	309,412	L	L
helm2d03	2,741,935	392,257	392,257	7.00	0.10	9	C	P
oilpan	3,597,188	73,752	73,752	48.80	13.50	70	C	P
laminar_duct3D	3,833,077	67,173	67,173	57.10	37.90	89	C	P
FEM_Cant	4,007,383	62,451	62,451	64.20	14.10	78	C	P
Protein	4,344,765	36,417	36,417	119.30	31.90	204	C	P
t3dh_a	4,352,105	79,171	79,171	55.00	14.50	81	C	P
offshore	4,242,673	259,789	259,789	16.30	2.80	31	C	P
ship_001	4,644,230	34,920	34,920	133.00	55.20	438	C	P
TF19	4,370,721	241,029	317,955	18.10	9.80	90	C	P
torso3	4,429,042	259,156	259,156	17.10	4.40	22	C	P
ASIC_680k	3,871,773	682,862	682,862	5.70	659.80	395,259	L	L
para-10	5,416,358	155,924	155,924	34.70	22.30	6,931	C	P
rajat29	4,866,270	643,994	643,994	7.60	773.90	454,521	L	L
FEM_Spheres	6,010,480	83,334	83,334	72.10	19.10	81	C	P
pwtk	5,926,171	217,918	217,918	27.20	6.20	90	C	P
tmt_sym	5,080,961	726,713	726,713	7.00	1.00	9	C	D
ESOC	6,019,939	327,062	37,830	18.40	0.80	19	C	P
TSOPF_RS_b2052_c1	6,761,100	25,626	25,626	263.80	310.50	635	C	P
boneS01	6,715,152	127,224	127,224	52.80	17.60	81	C	P
nd6k	6,897,316	18,000	18,000	383.20	89.20	514	C	P
crankseg_2	7,106,348	63,838	63,838	111.30	108.50	3,423	C	P
rajat30	6,175,377	643,994	643,994	9.60	784.60	454,746	L	L
shipsec5	7,236,289	179,860	179,860	40.20	27.20	126	C	P
bmw7st_1	7,339,667	141,347	141,347	51.90	12.70	435	C	P

Table A.2 continued from previous page

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr	Category on Skylake	Category on KNL
FEM_Ship	7,813,404	140,874	140,874	55.50	11.10	102	C	D
cont11_l	5,382,999	1,468,599	1,961,394	3.70	0.90	5	C	D
cage13	7,479,343	445,315	445,315	16.80	5.10	39	C	D
Rucci1	7,791,168	1,977,885	109,900	3.90	0.30	4	C	D
m_t1	9,753,570	97,578	97,578	100.00	28.60	237	C	D
mip1	10,352,819	66,463	66,463	155.80	350.70	66,395	P	D
thermal2	8,580,313	1,228,045	1,228,045	7.00	0.80	11	P	D
bmwcra_1	10,644,002	148,770	148,770	71.50	18.50	351	P	D
hood	10,768,436	220,542	220,542	48.80	12.80	77	P	D
pwtk_f	11,634,424	217,918	217,918	53.40	4.70	180	P	D
rail4284	11,284,032	4,284	1,096,894	2634.00	4209.30	56,182	P	D
F1	13,590,452	343,791	343,791	39.50	42.20	378	P	D
crankseg_2_f	14,148,858	63,838	63,838	221.60	95.90	3,423	P	D
nd24k	14,393,817	72,000	72,000	199.90	101.00	483	P	D
Si41Ge41H72	15,011,265	185,639	185,639	80.90	127.00	662	P	D
human_gene2	18,068,388	14,340	14,340	1260.00	1375.10	7,229	P	D
mouse_gene	18,221,931	45,101	45,101	404.00	645.80	6,790	P	D
af_shell3	17,588,875	504,855	504,855	34.80	1.30	40	P	D
Ga41As41H72	18,488,476	268,096	268,096	69.00	105.40	702	P	D
12month1	22,624,727	12,471	872,622	1814.20	4554.40	75,355	P	D
StocF-1465	21,005,389	1,465,137	1,465,137	14.30	2.60	189	P	D
ldoor	23,737,339	952,203	952,203	24.90	19.70	77	P	D
F1_f	26,837,113	343,791	343,791	78.10	40.80	435	P	D
rajat31	20,316,253	4,690,002	4,690,002	4.30	1.10	1,252	D	D
nd24k_f	28,715,634	72,000	72,000	398.80	76.90	520	P	D
cage14	27,130,349	1,505,785	1,505,785	18.00	5.40	41	D	D
FullChip	26,621,990	2,987,012	2,987,012	8.90	1806.80	2,312,481	L	L
inline_1	36,816,342	503,712	503,712	73.10	35.60	843	D	D
Emilia_923	41,005,206	923,136	923,136	44.40	3.70	57	D	D
spal_004	46,168,124	10,203	321,696	4525.00	1492.00	6,029	D	D
ldoor_f	46,522,475	952,203	952,203	48.90	11.90	77	D	D
Hook_1498	60,917,445	1,498,023	1,498,023	40.70	14.00	93	D	D
Geo_1438	63,156,690	1,437,960	1,437,960	43.90	4.40	57	D	D
circuit5M	59,524,291	5,558,326	5,558,326	10.70	1356.60	1,290,501	L	L
bone010	71,666,325	986,703	986,703	72.60	15.80	81	D	D
cage15	99,199,551	5,154,859	5,154,859	19.20	5.70	47	D	D
Flan_1565	117,406,044	1,564,794	1,564,794	75.00	11.40	81	D	D

A.3 Scale-free Matrices

Table A.3: Scale-free matrices.

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr	Category on Skylake	Category on KNL
soc-sign-epinions	841,372	131,828	131,828	6.40	32.90	2,070	C	C
connectus	1,127,525	512	394,792	2202.20	7584.40	120,065	L	L
language	1,216,334	399,130	399,130	3.00	20.70	11,555	C	C
NotreDame_actors	1,470,404	392,400	127,823	3.70	10.30	646	C	C
citationCiteseer	2,313,294	268,495	268,495	8.60	16.30	1,318	C	C
web-Stanford	2,312,497	281,903	281,903	8.20	11.30	255	C	C
com-dblp.ungraph	2,099,732	426,000	426,000	4.90	9.10	343	C	P
cnr-2000	3,216,152	325,557	325,557	9.90	20.50	2,716	C	P
amazon0312	3,200,440	400,727	400,727	8.00	3.10	10	C	P
Webbase	3,105,536	1,000,005	1,000,005	3.10	25.30	4,700	C	P
IMDB	3,782,463	428,440	896,308	8.80	15.30	1,334	C	P
web-Google	5,105,039	916,428	916,428	5.60	6.60	456	C	D
com-youtube	5,975,248	1,157,830	1,157,830	5.20	50.30	28,754	C	D
Stanford_Berkeley	7,583,376	683,446	683,446	11.10	284.80	83,448	C	D
web-BerkStan	7,600,595	685,230	685,230	11.10	16.40	249	C	D
roadNet-CA	5,533,214	1,971,281	1,971,281	2.80	1.00	12	C	D
wiki-Talk	5,021,410	2,394,385	2,394,385	2.10	99.90	100,022	C	D
flickr	9,837,214	820,878	820,878	12.00	87.70	10,272	P	D

Table A.3 continued from previous page

Matrix	NNZ	NR	NC	Mean nnzr	SD nnzr	Max nnzr	Category on Skylake	Category on KNL
higgs	14,855,842	456,627	456,627	32.50	49.10	1,259	P	D
in-2004	16,917,053	1,382,908	1,382,908	12.20	37.20	7,753	P	D
eu-2005	19,235,140	862,664	862,664	22.30	29.30	6,985	P	D
patents	14,970,767	3,774,768	3,774,768	4.00	5.30	36	D	D
cit-Patents	16,518,948	3,774,768	3,774,768	4.40	7.80	770	D	D
as-Skitter	22,190,596	1,696,415	1,696,415	13.10	136.90	35,455	D	D
coPapersDBLP	30,491,458	540,486	540,486	56.40	66.20	3,299	D	D
topcats	28,511,807	1,791,489	1,791,489	15.90	30.40	3,907	D	D
coPapersCiteseer	32,073,440	434,102	434,102	73.90	101.30	1,188	D	D
pokec	30,622,564	1,632,804	1,632,804	18.80	32.10	8,763	D	D
wikipedia-20070206	45,030,389	3,566,907	3,566,907	12.60	33.00	7,061	D	D
road_central	33,866,826	14,081,816	14,081,816	2.40	0.90	8	D	D
wb-edu	57,156,537	9,845,725	9,845,725	5.80	20.30	3,841	D	D
soc-LiveJournal1	68,993,773	4,847,571	4,847,571	14.20	36.10	20,293	D	D
ljournal-2008	79,023,142	5,363,260	5,363,260	14.70	37.00	2,469	D	D
road_usa	57,708,624	23,947,347	23,947,347	2.40	0.90	9	D	D
hollywood-2009	113,891,327	1,139,905	1,139,905	99.90	271.90	11,468	D	D
kron_g500-logn21	182,082,942	2,097,152	2,097,152	86.80	755.60	213,905	D	D
com-orkut.ungraph	234,370,166	3,072,600	3,072,600	76.30	154.80	33,313	D	D
socfb-konect	185,044,029	59,216,215	59,216,215	3.10	22.60	4,960	D	D