Optimizing Sparse Matrix-Matrix Multiplication for Graph Computations on GPUs and Multi-Core Systems

A Thesis

Presented in Partial Fulfillment of the Requirements
for the Degree Master of Science in the
Graduate School of The Ohio State University

By

Vineeth Reddy Thumma

Graduate Program in Computer Science and Engineering

The Ohio State University

2018

Master's Examination Committee:

Professor P. Sadayappan, Advisor

Professor Srinivasan Parthasarathy

© Copyright by

Vineeth Reddy Thumma

2018

Abstract

General sparse matrix-matrix multiplication (SpGEMM) is a fundamental building block and a core component for many data analytics and graph algorithms. An efficient parallel SpGEMM implementation has to handle challenges such as irregular nature of the computation and determination of the non-zero entries in the result matrix. In order to overcome these challenges and to exploit the characteristics of the hardware, various algorithms are devised to improve SpGEMM performance on GPUs and multi-core systems. An experimental study is done on Regularized Markov Clustering(R-MCL) algorithm which has SpGEMM as an important primitive and a parallel algorithm has been devised to improve its performance. A new approach to do K-Truss decomposition of a Graph using a variant of SpGEMM has been proposed which uses adjacency matrix formulation.

To my parents and my brother for their love and support without whom none of my success would be possible

Acknowledgments

This thesis would not have been possible without the guidance and support of several individuals. First and foremost, I would like to express my deepest gratitude to Professor P. Sadayappan for giving me an opportunity to work with him. I am grateful for the valuable advice, constant guidance and motivation he gave me through out the work. One could not wish for a better advisor and I am indebted to him in many ways.

I am grateful to Professor Srinivasan Parthasarathy for his valuable insights and advice. His involvement with the work triggered my interest in the area of graph mining and I couldn't thank him enough.

I am very thankful to Dr. Aravind Sukumaran Rajam for his constant guidance and feedback. I learnt a lot from him and I am grateful for the support he gave me through out the thesis.

I would like to thank my lab mates Emre, Rakshith, Kunal, Rohit, Jinsung, Miheer, Rui, Israt, Changwan, Gordon, Prashant and Wenlei for the memorable time I had at HPCRL.

I would like to thank my friends Venkat, Kalyan, Anirudh, Dhanvi, Prithvi, Sankeerth and Harsha without whom life in Columbus wouldn't have been so fun. I will always cherish the time I have spent with them.

I would like to thank my friends from BITS - Srikanth, Rakesh, Sriteja, Goutham, Dileep, Srinath, Sujith, Sai Krishna, Jeevan, Jithendra, Nivedith, Karthik, Gokul, Praneeth, Arun, Sampath, Swamy and Sourav. They will be an inseparable part of my life.

I would like to thank my buddies from Freescale - Albert, Siva, Lohit and Abhinav. I will miss all the intellectual and fun conversations that I had with them. I would like to thank my school friends - Nikhil, Rajashekar and Avinash for always being with me.

Finally, I would like to thank my family for all the love and support and for being a pillar of strength to me. I attribute all the success in my life to them.

Vita

2014	B.E. Electronics and Communication, BITS Pilani - Hyderabad, India
2014-2016	Software Engineer, Freescale Semiconductor, India
2017-present	Graduate Research Associate, The Ohio State University

Publications

Research Publications

S. E. Kurt, V. Thumma, C. Hong, A. Sukumaran-Rajam and P. Sadayappan "Characterization of Data Movement Requirements for Sparse Matrix Computations on GPUs".
2017 IEEE 24th International Conference on High Performance Computing (HiPC)

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

Abstract ii

Dedication iii

Acknowledgments iv

Vita vi

List of Tables ix

List of Figures x

1. Introduction 1

    1.1 Motivation 1
    1.2 SpGEMM formulation 2
    1.3 Parallel SpGEMM Challenges 4
    1.4 Organization of this Thesis 5

2. Background 6

    2.1 2-Phase Approach 7
    2.2 Load Balancing using Binning 8
    2.3 Scatter Vector Approach 9

3. Improving SpGEMM on GPUs 11

    3.1 Roofline Model 11
    3.2 Experiments with Synthetic Banded Matrices 14
    3.3 Dynamic Virtual Warping 16
    3.4 Memory Management 18
    3.5 Implementation Details and Results 20

4. Improving SpGEMM on Multi-core Systems 22

4.1 Matrix Storage Format 22
4.1.1 CSR format 22
4.1.2 Dynamic Sparse Row format 25
4.2 Memory Management and Alignment 27
4.3 Zero filling and Vectorization 27
4.4 Vectorized SpGEMM Algorithm 29
4.5 Implementation Details and Results 33

5. SpGEMM Applications 40

5.1 Regularized Markov Clustering(R-MCL) 40
5.1.1 Background 40
5.1.2 Implementation Details and Results 43
5.2 K-Truss Decomposition 51

6. Conclusion 57

Appendices 58

A. Test Set of Matrices 58

Bibliography 61

List of Tables

Table **Page**

4.1 Matrices Used 34

List of Figures

Figure **Page**

1.1 CSR representation of a Sparse Matrix 2

3.1 Roofline plot: Dense MV vs Dense MM 12

3.2 SpMV vs SpGEMM: Performance and Operational Intensity 13

3.3 Banded Matrices: Original vs Randomized 14

3.4 Virtual Warp Experiment on Banded Matrices 17

3.5 Performance comparison: low-throughput matrices 21

3.6 Performance comparison: high-throughput matrices 21

4.1 CSR representation 24

4.2 Dynamic Sparse Row representation 26

4.3 Splitting matrix B into Dense B and Sparse B 28

4.4 Performance comparison on Machine 1 - Set I 35

4.5 Performance comparison on Machine 1 - Set II 35

4.6 Performance comparison on Machine 1 - Set III 36

4.7 Ratio of times of (split B)/(1 iteration of SpGEMM) on Machine 1 . 36

4.8 Performance comparison on Machine 2 - Set I 37

4.9 Performance comparison on Machine 2 - Set II 37

4.10 Performance comparison on Machine 2 - Set III 38

4.11 Ratio of times of (split B)/(1 iteration of SpGEMM) on Machine 2 . 38

5.1 Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 1 44

5.2 Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 1 44

5.3 Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 1 45

5.4 Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 2 45

5.5 Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 2 46

5.6 Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 2 46

5.7 Ratio of R-MCL time for (MKL/DSR) on Machine 1 48

5.8 Ratio of R-MCL time for (MKL/DSR) on Machine 1 48

5.9 Ratio of R-MCL time for (MKL/DSR) on Machine 1 49

5.10 Ratio of R-MCL time for (MKL/DSR) on Machine 2 49

5.11 Ratio of R-MCL time for (MKL/DSR) on Machine 2 50

5.12 Ratio of R-MCL time for (MKL/DSR) on Machine 2 50

5.13 Truss decomposition of a graph. Each edge is labeled with it's truss number 51

Chapter 1: Introduction

1.1 Motivation

Sparse matrix computations are at the core of many compute-intensive applications, both in scientific/engineering modeling/simulation as well as large-scale data analytics. A large number of graph algorithms can also be formulated in the language of sparse linear algebra. Many portable implementations of graph algorithms are being developed using efficient implementations of key sparse matrix operations.

General Sparse Matrix-Matrix multiplication(SpGEMM) is a fundamental building block and an important primitive for many applications. Applications like algebraic multi-grid solvers [3] and linear solvers [36] have SpGEMM as an important sub-routine in them. It is also a core component for many graph analytics algorithms like Markov clustering [21], dynamic reachability in directed graphs [25], all-pair shortest path [6], cycle detection [37], sub graph detection [31], maximum graph matching [20]. Apart from these, SpGEMM also has its applications in computing Jacobian products [11], quantum modelling[26], optimizing joins on relational databases [2]. A variant of SpGEMM can be used in applications like Triangle Enumeration and K-Truss Decomposition. Thus, improving the performance of SpGEMM using efficient and parallel algorithms is of critical importance.

1.2 SpGEMM formulation

General Sparse Matrix-Matrix Multiplication (SpGEMM) multiplies a sparse matrix A of size $m \times k$ with a sparse matrix B of size $k \times n$ and gives a result matrix C of size $m \times n$.

If the input sparse matrices are represented using the standard CSR (Compressed Sparse Row) format, SpGEMM can be formulated as operations on row vectors of the input matrices. Figure 1.1 shows CSR representation of an example sparse matrix and Algorithm 2 shows the high-level pseudo code for SpGEMM.

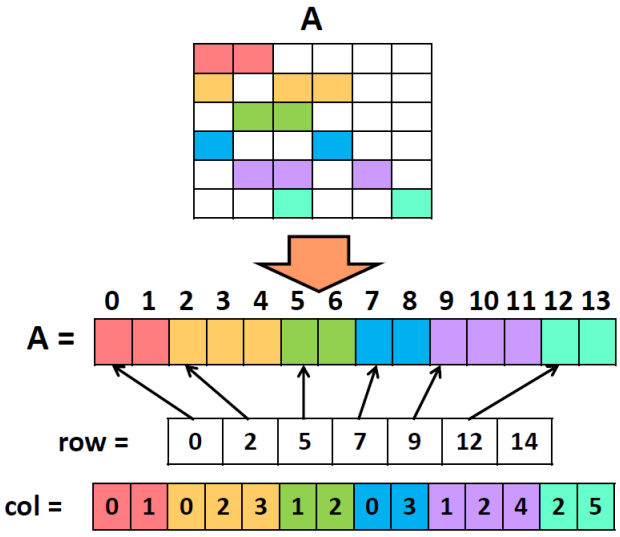

Figure 1.1: CSR representation of a Sparse Matrix

In CSR format, efficient contiguous access to elements in any row is possible, but access to the elements in a column is not efficient. In order to compute the elements of a row i of C, all nonzero elements A_{i*} must be accessed, and for each such nonzero A_{ik}, all elements B_{k*} need to be accessed. For each such nonzero element B_{kj} , the

Algorithm 1: Dense Matrix-Matrix Multiplication

```
input  : DenseMatrix A[M][N], DenseMatrix B[N][P]
output: DenseMatrix C[M][P]
for i = 0 to M-1 do
    for j = 0 to P-1 do
        C[i][j] = 0
        for k = 0 to N-1 do
            C[i][j] += A[i][k] * B[k][j]
        end
    end
end
```

Algorithm 2: Sparse-Matrix Sparse-Matrix Multiplication

```
input  : SparseMatrix A[M][N], SparseMatrix B[N][P]
output: SparseMatrix C[M][P]
for each A[i][*] in matrix A do
    for each non-zero entry A[i][k] in A[i][*] do
        for  each non-zero entry B[k][j] in B[k][*] do
            value = A[i][k] * B[k][j]
            if C[i][j] ∉ C[i][*] then
                Insert C[i][j] in C[i][*]
                C[i][j] = value
            end
            else
                C[i][j] += value
            end
        end
    end
end
```

product A_{ik}*B_{kj} must be computed and it contributes to a nonzero element C_{ij} in the resultant matrix C.

When multiplying two sparse matrices, the operations for each row of the resultant matrix correspond to floating point (Fuse Multiply Add) computations. There is a large variance in the number of floating point operations across each of these rows depending upon the sparsity pattern of the input matrices and the variance of number of non-zeros in each row. Various algorithms have been developed for SpGEMM to optimize for performance on GPUs and multi-core systems.

1.3 Parallel SpGEMM Challenges

Some major challenges in the parallel implementation of SpGEMM are described in this section.

The number of non-zero elements in the resultant matrix is not known beforehand. This is because of the variance in the sparsity pattern of the input matrices. Due to this fact, the memory for the resultant matrix cannot be allocated a priori without scanning through the input matrices.

The work division among threads to perform the computations is not trivial as the amount of work to compute a row of the resultant matrix is non-uniform and is dependent on the sparsity of both the input matrices.

Another significant challenge in computing the sparse matrix product is in efficiently gathering together the various additive contributions to an element C_{ij} from different rows of B.

Various strategies and algorithms to overcome these challenges and to improve SpGEMM performance on GPUs and multi-core systems have been discussed in the subsequent chapters.

1.4 Organization of this Thesis

The rest of the Thesis is organized as follows. Chapter 2 gives a background about various approaches previously used to do SpGEMM. Chapter 3 discusses the approaches used and improvements done to perform SpGEMM on GPUs. Chapter 4 discusses the contributions done to improve SpGEMM on multi-core systems. Chapter 5 discusses about the experimental study done to optimize performance of applications like Regularized Markov Clustering that have SpGEMM as an important primitive. It also discusses about K-Truss decomposition of a graph using a variant of SpGEMM algorithm. Chapter 6 presents the conclusion for the Thesis.

Chapter 2: Background

Existing implementations on GPUs for dense matrix-vector multiplication and dense matrix-matrix multiplication achieve performance quite close to the roof-line bounds based on operational intensity(the ratio of number of floating-point operations and measured data volume). For large dense matrices, performance of matrix-vector multiplication is significantly lower than matrix-matrix multiplication. However, in contrast, the performance of state-of-the art GPU implementations of sparse matrix-matrix multiplication(SpGEMM) is generally much lower than that of sparse matrix-vector multiplication(SpMV).

In the case of multi-core platforms, existing benchmark implementations for SpGEMM like Intel MKL use a two-phase approach, where the number of non-zeroes is computed in the first phase and the computation is done in the second phase after allocating the memory for the resultant matrix.

In the following sections, some of the key approaches previously used to do SpGEMM on GPUs and multi-core systems have been discussed briefly. Improvements over these and other new approaches will be discussed in the subsequent chapters.

2.1 2-Phase Approach

Sparse matrix multiplication operations like SpMM(Sparse Matrix*Dense Matrix) and SpMV(Sparse Matrix*Dense Vector) allocate memory for the resultant dense matrix and dense vector respectively as the memory required for the result is known prior to the computation. Also, the resultant entries in these computations are mapped to predictable memory addresses. However due to the nature of SpGEMM, the exact memory needed for the resultant matrix is not known beforehand. To overcome this, a 2-phase approach is used. In the first phase, the memory is allocated based on the estimate on the number of non-zeroes in the resultant matrix. In the second phase, the computations to form the result are performed. Different solutions like precise method, probabilistic method, upper bound method and progressive method have been proposed implementing this approach.

In the precise method, a simplified version of SpGEMM is done which has the same computational pattern to know the exact number of non-zeroes. Intel MKL and CUSPARSE library use this method.

In the probabilistic method, non-zeroes in the result matrix are estimated based on random sampling and probability analysis [1], [4] of the input matrices. In case the estimation fails extra memory has to be allocated.

In the upper bound method, an upper bound on the non-zeroes in the result matrix is computed and corresponding memory is allocated [16]. The upper bound equals the number of FMA operations(half the floating point operations).

In the progressive method, memory of some size is allocated initially and the memory is reallocated if more memory is needed as the sparse matrix computation

progresses. Sparse matrix computations in Matlab [13], some CPU sparse matrix libraries and some GPU implementations [18] use this approach.

2.2 Load Balancing using Binning

Due to non-uniform distribution pattern of non-zeroes in the sparse matrices, it is challenging to do SpGEMM as the amount of work done by each thread will be uneven. In order to handle the problem of load balancing and to have good concurrency among all the threads, some previous SpGEMM implementations on GPUs have tried grouping the rows of A initially into different bins.

Depending on the total number of operations it is involved in, each row of the matrix A is grouped into a bin. Some previous GPU implementations [16] grouped the bins with range of operations being powers of 2. The memory in matrix C for all the rows that correspond to a particular bin is allocated in upper bound fashion; i.e. all the rows in the bin having ops in the range of (a,b] are pre-allocated with memory that can hold b elements . The rationale behind this is that the number of non-zeros in that row cannot be more than the number of ops i.e, b. All the rows with the number of operations beyond a certain threshold are put in the last bin. As the upper bound for the rows in the last bin cannot be estimated, memory allocation and computations happen in different phases. Memory can be allocated progressively as the need arises. Other implementations [18] have tried different ways of binning.

In the next stage, each of the bins are executed with different kernel programs that are customized for that bin. The results are computed and stored in the temporary

matrix memory allocated for the bins. In the final stage, the total number of non-zeroes in the entire C matrix is computed and memory is allocated. The non-zeroes from the temporary matrix is then copied to the result matrix C.

2.3 Scatter Vector Approach

Scatter Vector is one of the approaches to solve the index matching problem in SpGEMM. It was originally devised for CPUs [14]. In this approach, a vector of size n, where n is the number of columns in matrix B is used to store the pointers to compacted elements in a row of the resultant matrix C. A row of A is processed sequentially. Before processing each row of A, the entire scatter vector is initialized to NULL value. For each A element in the current row, the corresponding B elements are identified and partial products are formed. For each such partial product, the corresponding column in the scatter vector is accessed. If the scatter vector contains a non-NULL value, then the current partial contribution is added to the location pointed by the scatter vector. If the value is NULL, then a unique location is obtained from a memory pool and is initialized with the current partial product. The address of the obtained unique location is written to the corresponding column of the scatter vector.

Algorithm 3: Sequential SpGEMM using Scatter-Vector Approach [15] [16]

```
input : A[M][K], B[K][N] in CSR format
output: C[M][N] in CSR format
SV[:] = -1
nnz_count = 0
C.row[0] = 0
for i = 0 to M-1 do
    for a_index = A.row[i] to A.row[i+1]-1 do
        a_val = A.val[a_index]
        j = A.col[a_index]
        for b_index = B.row[j] to B.row[j + 1] - 1 do
            k = B.col[b_index]
            if SV[k] == -1 then
                C.col[nnz_count] = k
                C.val[nnz_count] = a_val * B.val[b_index]
                SV[k] = nnz_count
                nnz_count = nnz_count+1
            end
            else
                index = SV[k]
                C.val[index] += a_val * B.val[b_index]
            end
        end
    end
    C.row[i+1] = nnz_count
    SV[:] = -1
end
```

Chapter 3: Improving SpGEMM on GPUs

Achieving high performance for SpGEMM on GPUs has been extremely challenging as opposed to the Dense Matrix Multiplication. Experimental data on achieved performance with state-of-the-art implementations of dense and sparse matrix-vector (MV) and matrix-matrix (MM) multiplications shows that the performance of dense MM is much higher than MV, while the opposite is true for the sparse case.

3.1 Roofline Model

A roofline plot [35] provides insightful visual illustration of the extent to which algorithms are constrained by the data-movement bandwidth limits of a system. It contains two asymptotic lines that represent upper bounds on performance: the maximum computational rate of processor cores and, the bandwidth from main memory to cores. The horizontal line represents peak computational performance (in GFLOPS), and the inclined line has a slope corresponding to the memory bandwidth (in Gbytes/sec). The y-axis of the roofline plot represents performance (in GFLOPs), while the x-axis represents the operational intensity (OI) of a computation, defined as the ratio of number of computational operations performed per byte of data moved between main memory and the processor cores. A code will be memory-bandwidth limited unless OI is sufficiently high, greater than a critical intensity corresponding to

the point of intersection of the two rooflines. This is because the product of the OI and the peak memory-bandwidth (slope of the inclined roofline) imposes an upper-bound on the number of computational operations that can be performed

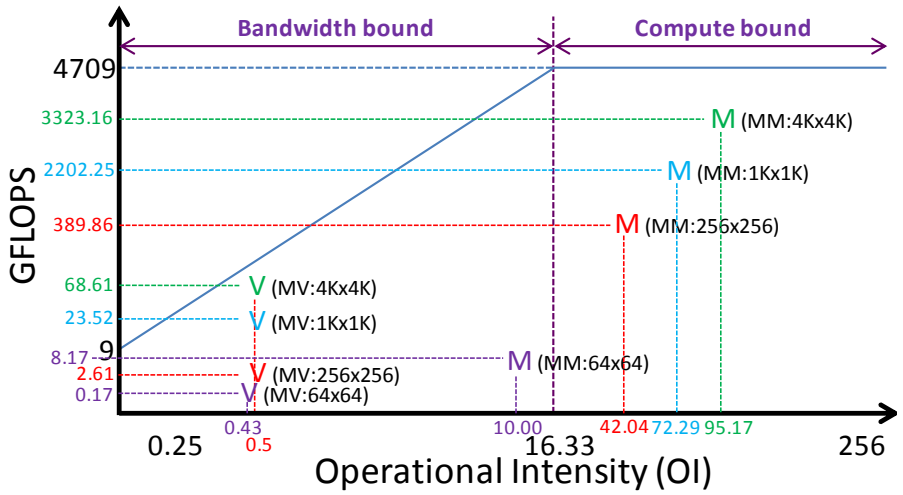

Figure 3.1: Roofline plot: Dense MV vs Dense MM

Figure 3.1 shows roofline plot for single-precision dense matrix-vector and dense matrix-matrix multiplication using Nvidia's CuBLAS library. The experiments are run on Nvidia Kepler K 20c GPU.

The following can be concluded from the roofline plot:

- Dense MV is bandwidth-bound, with achieved performance being quite close to the asymptotic sloping roofline representing the memory-bandwidth-based performance limit.

- The performance of Dense MM is over 30x than dense MV for large matrices, and the computation is clearly compute-bound for large enough problem sizes, the plotted points are far to the right of the point where the rooflines intersect.

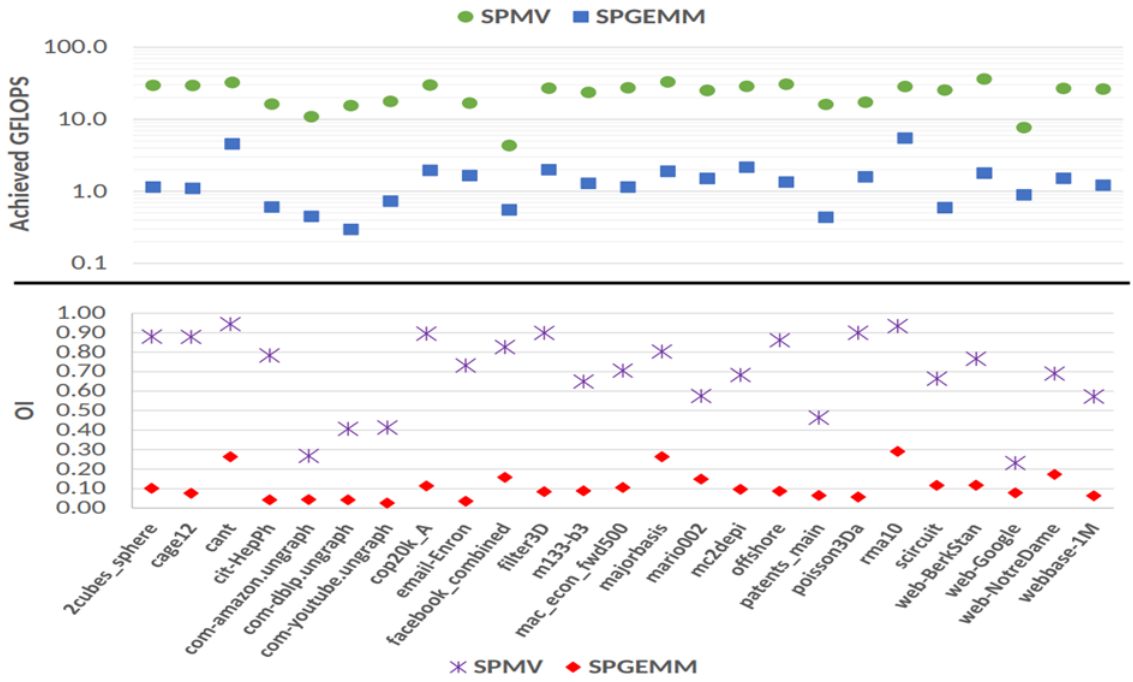

Figure 3.2: SpMV vs SpGEMM: Performance and Operational Intensity

A similar plot for sparse matrix-vector (SpMV) and sparse matrix-matrix (SpGEMM) multiplication plotted using a collection of 25 sparse matrices from the Suite Sparse Matrix Collection [8] is shown in Figure 3.2. For SpMV, a state-of-the-art implementation using the CSR5 variant [19] of the compressed sparse row (CSR) data structure was used. For SpGEMM, the HybridSparse [16] code was used which has been shown to achieve higher performance than any other publicly available GPU SpGEMM code.

The following conclusions can be drawn regarding SpMV versus SpGEMM:

- As opposed to the dense case, the performance of SpGEMM is considerably lower than performance of SpMV across all tested sparse matrices.
- The measured OI for SpGEMM is much lower than the measured OI for SpMV, which already is more memory bandwidth bound than dense MV. This is very

different from the relative operational intensities achieved by dense MM versus dense MV.

3.2 Experiments with Synthetic Banded Matrices

A challenge to gleaning insights into performance bottlenecks for SpGEMM is the fact that data elements from three sparse matrices are used in each elementary operation. Even if two of the matrices are kept the same to perform C = A*A, different rows of A (with different sparsity patterns) are involved. In order to better control the variability and gain insights, a set of experiments are devised to perform SpGEMM on banded matrices, but represented using a CSR representation. Further, a random symmetric row/column permutation was performed for each tested banded matrix, and the randomized variant was also tested.

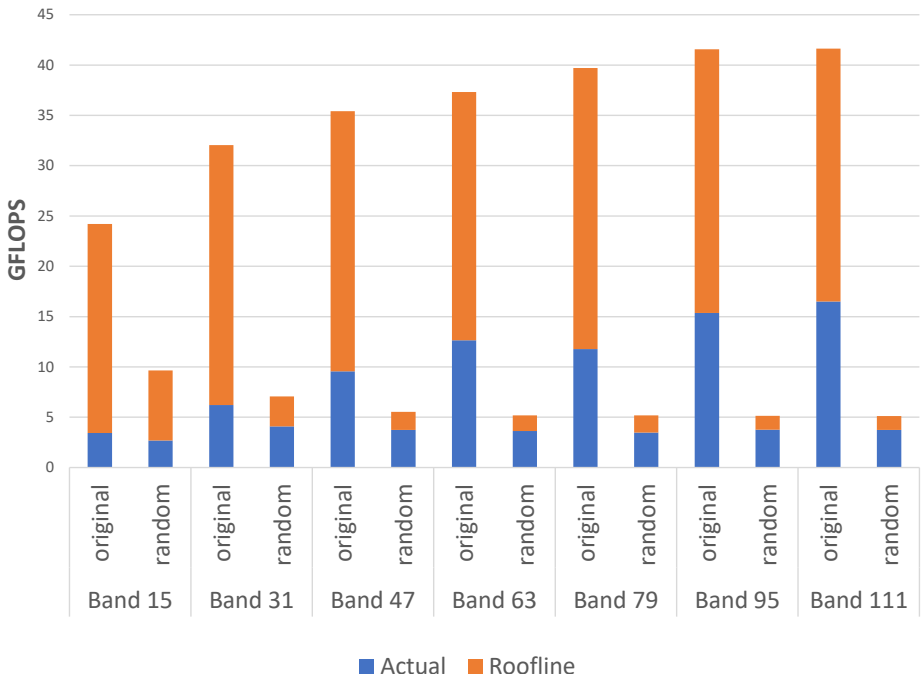

Figure 3.3: Banded Matrices: Original vs Randomized

Figure 3.3 presents the results of these experiments. The set of matrices had half-band sizes of 15, 31, 47, 63, 79, 95, and 111. For each matrix and its randomized variant, the stacked bar-chart shows actual achieved performance (blue) as well as the roofline performance bound based on measured data volume to/from global memory.

The main observations are as follows:

- The randomized variants achieve significantly lower performance than the contiguously represented banded matrices. This is a consequence of the significantly worse temporal locality in accessing rows of B for the randomized variant, as well as worse data coalescing for accesses to elements of C with the scatter-vector approach.

- As the band size increases (going from left to right in the chart), the ratio of roofline to actual performance decreases noticeably. This is more prominent for the randomized variants, which have a much higher data volume than the corresponding contiguous variants. This trend is suggestive that inadequate thread-level concurrency is a likely factor in the low performance of the SpGEMM implementation. As the band size increases, the average degree of concurrency with the scatter vector approach increases, proportionally with the band size. At high band sizes, the execution is moving closer to a bandwidth-limited regime, as indicated by the lower ratio of roofline performance bound to actual achieved performance.

The latter observation suggests the development of an adaptive work distribution scheme which is discussed in the next section.

3.3 Dynamic Virtual Warping

As shown in Figure 3.3, for many matrices Hybrid SpGEMM approach is far from the roofline limit which indicates that the approach is latency limited. Latency effects can be reduced by exposing more parallelism. The amount of available parallelism in GPUs is limited by the total number of warps that can be simultaneously active. Achieved Occupancy metric from Nvprof indicate that the kernels achieve near optimal occupancy. This suggests that even with high occupancy, the effective parallelism is less. From Figure 3.3, it can be observed that the gap between achieved GFLOPS and theoretical GFLOPS is smaller for larger bands. The reason for it is that in the existing Hybrid SpGEMM implementation, for each element in a row of A matrix, all the threads in a thread block are assigned to process the corresponding B elements. The number of threads assigned to process each element in a given row of A depends on the total number of operations corresponding to that row (bin id). For example, bin 12 is assigned 128 threads. It may happen that even though the number of ops is high the number of B elements is small. Since the elements in a row of matrix "A" are processed sequentially, if the number of elements in the corresponding rows in B matrix is less many threads will be idle. Even though the threads are idle, they have not exited the kernel. This is reflected in the measured achieved occupancy metric as Nvprof considers these threads as active and reports a high occupancy. When compared to smaller bands, higher bands have higher number of non-zeros in B. Hence, most of the threads are not idle which results in higher performance.

In order to improve effective concurrency, virtual warping scheme was implemented. Each thread block processes multiple rows of A simultaneously and the entire threads in a thread block are equally divided and assigned to process each

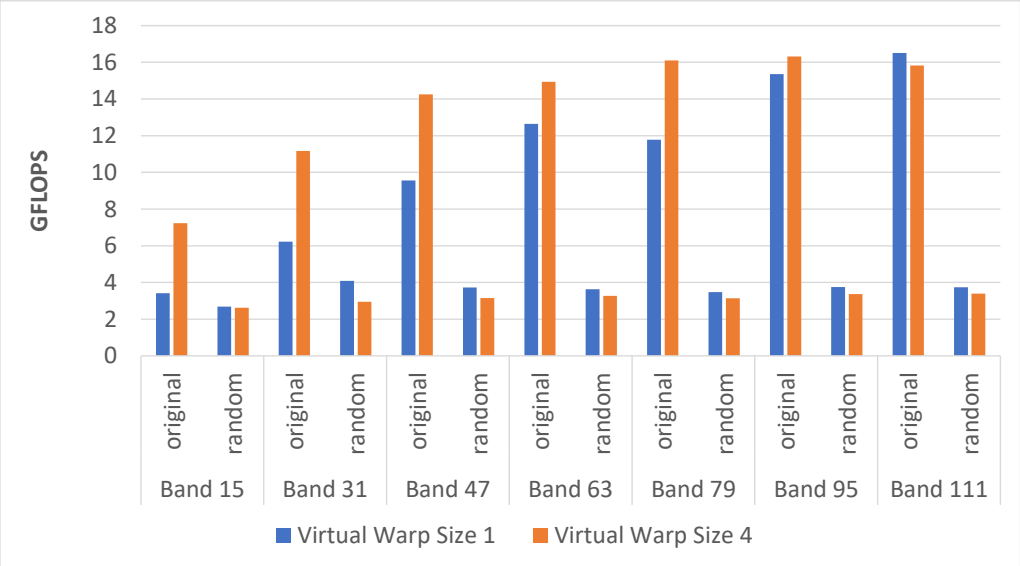

Figure 3.4: Virtual Warp Experiment on Banded Matrices

row of A. For example, for a thread block of size 128, and virtual warp of size 4, each thread block will process 4 rows of A simultaneously. For each row of A, 32 (128/4) threads are assigned to cyclically process the corresponding elements in B. Virtual warping improved the performance of the approach. Figure 3.4 compares the performance of our approach when virtual warp size is 1 and 4. Note that virtual warping is not beneficial for all bands sizes, which motivates an adaptive scheme. If the virtual warp size is greater than 4, then accesses to B elements may be partially un-coalesced. For example, if the virtual warp size is 8, then 16 (128/8) threads will work on each row of A. The 32 elements of B (corresponding to two rows of A) that are simultaneously required by a warp may not be contiguous which results in un-coalesced accesses. If the average number of non zeros in B is greater than 32, then the latter choice may not be beneficial. Consider another example when the average number of non-zeros in B is 128. In the latter case, a virtual warp size of 4 or 1 can

achieve fully coalesced access; however the performance of virtual warp of size 1 can be better due to the following reason. When the virtual warp size is 1, each thread block is processing one row of A at a time. In contrast, when the virtual warp size is 4, four rows of A are processed at the same time, which increases the cache pressure and reduces the cache hit rate.

The adaptive virtual warping scheme decides the virtual warp size depending on the average number of non-zeros in B for a given row of A. For a given C row, the average number of non-zeros in B is determined by dividing the total ops by the number of elements in corresponding row of A. To enable this, along with binning the rows are also sub-binned based on the total ops for a row of A divided by non-zeroes in that row.

3.4 Memory Management

As discussed in Section 2.2, as the upper bound for the rows in last bin cannot be estimated, memory allocation has to be done in two phases. In the first phase, the number of non-zeroes in the last bin is determined by going through the rows of A and B corresponding to the last bin. In the second phase, once the memory has been allocated for the last bins, computations are done by going through the same corresponding rows in A and B. Since these two phases necessarily have the same computational pattern, it might be efficient to avoid two phases. To achieve this, a huge block of memory is initially allocated before the SpGEMM kernels for the last bins are executed. This block of memory is divided into chunks of equal size. The idea is that, for the last bin, each virtual warp will be responsible for computing a row of C. Each virtual warp will initially have a chunk of memory to start with and it

computes the elements of matrix C by traversing the corresponding rows of the input matrices and writes them simultaneously to these memory chunks. When a virtual warp runs out of memory, it will get the next chunk of memory by doing an atomic on the global memory lock so that no two virtual warps get the same chunk. Each warp will store the pointer to the next chunk at the end of the current chunk. This is needed as the values in these chunks has to be copied back to a resultant CSR matrix C in the next phase. In this implementation, Scatter Vector approach(discussed in Section 2.3) is used for storing the pointers to the memory locations(in the chunks). Each virtual warp has a scatter vector that store the values corresponding to the columns of a row in C that it is computing. Once the virtual warp finishes computing a row of C, it re-sets the entire scatter vector to NULL. In order to reduce shared memory usage and get better performance, warp shuffle instructions are used as they are faster.

3.5 Implementation Details and Results

Figures 3.5, 3.6 present a performance comparison (single precision) of the new SpGEMM code (HybridSparse DVW) with other publicly available SpGEMM codes for GPUs (HybridSparse [16], bhSPARSE [18], cuSPARSE [23], KKMEM [9])and multi-core CPUs (Intel MKL and KKMEM [16]). The trend is similar for double precision data type. The GPU was an Nvidia GTX TITAN (14 Kepler SMs, 192 cores/MP, 6 GB Global Memory, 876MHz, 1.5 MB L2 cache, ECC off). The multi-core CPU was a quad-core (8 thread) Intel Core i7-6700K CPU @ 4.00GHz, 8MB L3 cache. The matrices used are the set of sparse matrices used in recent GPU-SpGEMM studies, dividing the set into a low-throughput group (Fig.3.5) and a high-throughput group (Fig.3.6). Nvidia's CUSP or cuSPARSE libraries are not used for comparison, since the implementations compared have shown to be consistently superior to CUSP[7] and cuSPARSE[23].

The leftmost two bars in the graphs show performance of HybridSparse DVW and HybridSparse, respectively. HybridSparse DVW is consistently better than HybridSparse for all high-throughput matrices (Fig.3.6). It is also superior for about half of the low-throughput matrices (Fig.3.5) and equal in performance for the other half. The middle two bars in the charts show performance of KKMEM-GPU (third bar from left) and bhSPARSE (fourth bar from left). For all matrices, HybridSparse DVW is consistently and often significantly faster than either KKMEM-GPU or bhSPARSE. The rightmost two bars show performance on the multi-core CPU for KKMEM-CPU (rightmost bar) and Intel MKL (second bar from the right). Again, for all tested matrices, HybridSparse DVW is consistently faster. However the peak performance of the multi-core CPU is considerably lower than that of the GPU.

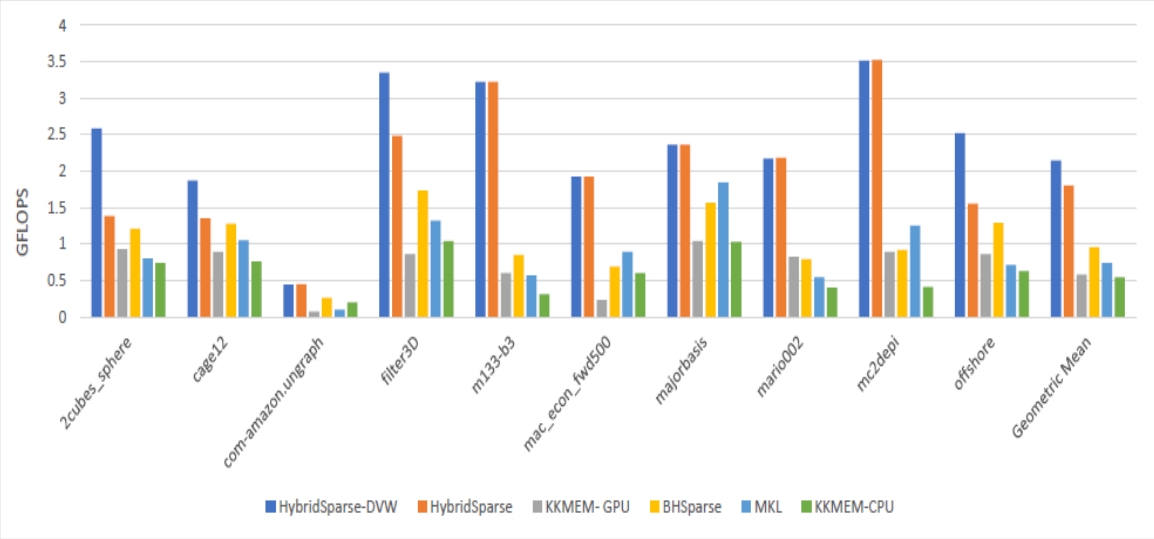

Figure 3.5: Performance comparison: low-throughput matrices

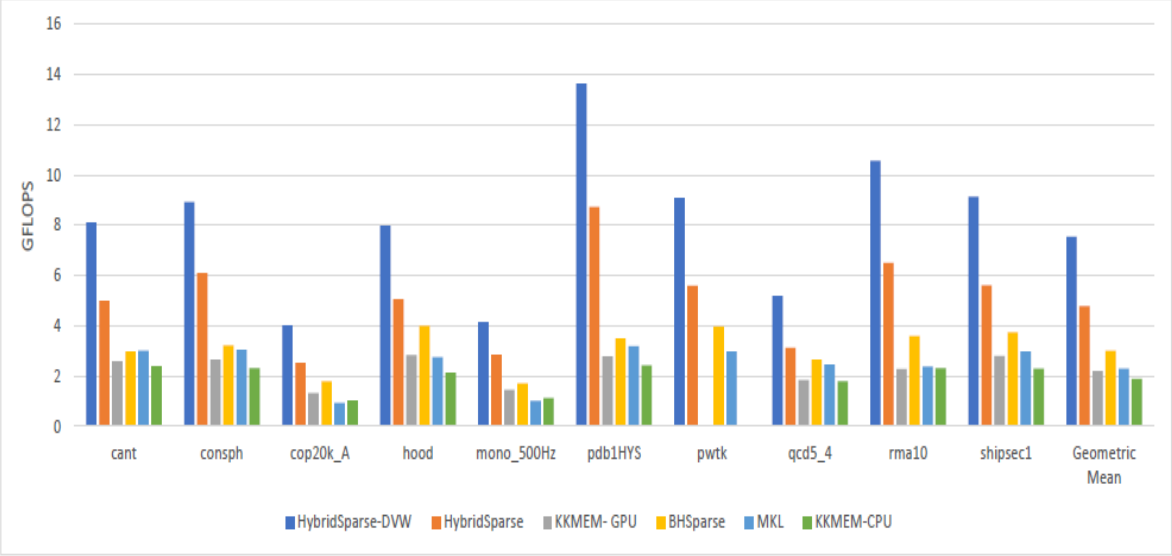

Figure 3.6: Performance comparison: high-throughput matrices

Chapter 4: Improving SpGEMM on Multi-core Systems

Existing multi-core libraries like Intel MKL use a two-phase approach, where the number of non-zeroes is computed in the first phase and the computation is done in the second phase after allocating the memory for the resultant matrix. Some implementations like Patwary et al.[24] explored various ideas like different partitioning techniques, cache optimizations and dynamic scheduling. Sulatcke and Ghose[30] proposed cache-efficient parallel algorithm for SpGEMM after analyzing the behavior of different loop orderings.

In this work, a new approach for SpGEMM on multi-core and many-core CPUs is proposed that uses a dynamic matrix storage representation to perform the computations in one phase. Later, approaches to exploit the benefits of SIMD vectorization on processors like Intel KNL is discussed.

4.1 Matrix Storage Format

4.1.1 CSR format

The standard compressed sparse row (CSR) format is specified by three arrays: the values, columns, and rowIndex. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix.

Values

An array that contains the non-zero elements of the matrix. The values of the non-zero elements of the matrix are mapped into the values array using the row-major storage mapping.

Column Indices

Element i of the integer array columns is the column number in the matrix that contains the i-th value in the values array.

RowIndex

Element j of this integer array gives the index of the element in the values array that is first non-zero element in a row j of matrix.

The length of the values and columns arrays is equal to the number of non-zero elements in the matrix. The length of the rowIndex array is equal to the number of rows in the matrix plus one.

The disadvantage of this format for the SpGEMM implementation is that, to allocate memory for resultant C matrix, an initial pass has to be made on the Input matrices (A and B) to determine the total no of non-zeroes in C. After that, memory needs to be allocated for the entire CSR C and then another pass has to be made on matrices A and B to fill the matrix C with calculated values. Because of the CSR matrix format used to represent a matrix, the memory needed for the non-zeroes of the entire CSR matrix C has to be allocated before computing the values.

Matrix:

5			1												
	2						-7						9		
2		4	-5	8		-1					81				
					-3										
	7								1	2	3	4	5		

values:

5	1	2	-7	9	2	4	-5	8	-1	81	-3	7	1	2	3	4	5

columns:

0	3	1	7	13	0	2	3	4	6	11	5	1	9	10	11	12	13

rowIndex:

0	2	5	11	12	12	18

Figure 4.1: CSR representation

4.1.2 Dynamic Sparse Row format

To overcome the disadvantage of previous format, a new matrix storage format is proposed where values and column indexes of all the rows in a matrix need not be contiguous in memory. The new format is specified by three arrays: the valuesPtr, columnsPtr, and nnzRow. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix A.

valuesPtr

An array that contains pointers to the values of non-zero elements of rows of a matrix. A pointer (element i of the valuesPtr array) points to an array that contains values of the non-zero elements of the row i.

columnsPtr

An array that contains pointers to the column indexes of non-zero elements of rows of a matrix. A pointer (element i of the columnsPtr array) points to an array that contains column indexes of the non-zero elements of the row i.

nnzRow

Element i of this integer array gives the number of non-zeroes in a row i of the matrix. The length of the valuesPtr, columnsPtr and nnzRow arrays is equal to the number of rows in a given matrix. Each of the arrays pointed to by the valuesPtr and columnsPtr is equal to the number of non-zeroes in that row of the matrix.

As all the rows of the matrix need not be contiguous in memory with the new storage format, the two-pass approach described above can be avoided. In the new approach, each thread independently computes the non-zeroes for a row its currently

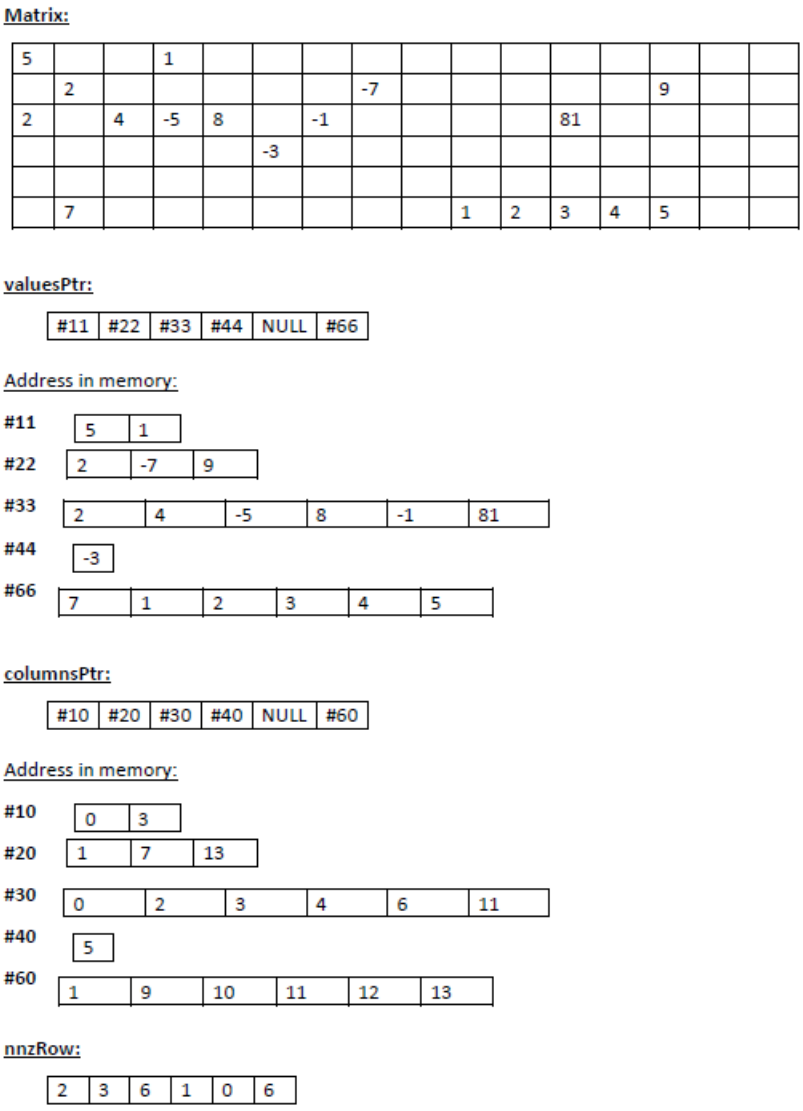

Figure 4.2: Dynamic Sparse Row representation

processing, gets the memory required to store the corresponding column indexes and values for those elements and computes those values at the same time. A memory manager is used which simultaneously allocates memory required by the threads.

4.2 Memory Management and Alignment

A thread-safe memory manager is implemented so that all the threads can simultaneously get memory required for the corresponding rows processed by them at the same time. The input matrix data and memory allocated by the memory manager is ensured to be memory-aligned. This helps to reduce memory access latency by efficient usage of fast memory(Cache).

4.3 Zero filling and Vectorization

Modern architectures have SIMD units with widths up to 512 bits. Exploiting these SIMD units is critical to achieving good performance. Consider a single element of A which may contribute to multiple elements in the same row of B. Each such partial product corresponds to same row but different columns in C. This computation can be vectorized. However, even though the elements of B are contiguous in Dynamic CSR format, the contributions to C may not be contiguous due to; i) the scatter vector approach does not guarantee column ordered memory allocation, and ii) the column structure of B and C can be different. Thus only scatter-gather vectorization is possible, limiting the vectorization potential. Scatter-gather vectorization has two disadvantages: i)low efficiency and ii) low availability.

To increase the vectorization opportunity, the implementation uses the idea of zero filling. In the pre-processing phase each row of B is split into sparse and dense

Original Matrix B:

5			1												
	2						-7						9		
2		4	-5	8		-1					81				
					-3										
	7								1	2	3	4	5		

Dense B:

2	0	4	-5	8	0	-1	0								
								0	1	2	3	4	5	0	0

Sparse B:

5			1												
	2						-7						9		
											81				
					-3										
	7														

Figure 4.3: Splitting matrix B into Dense B and Sparse B

parts (Figure 4.3). Each row of B is viewed as a set of windows of size equal to the vector width (VW). For a given row i^{th} window corresponds to the column range of $[i * VW, (i + 1) * VW]$. If the number of non-zero elements in a window is greater than a threshold, then those elements are moved to the dense part (after zero-filling the empty columns). The remaining elements are kept in the sparse part. The dense parts of B can now be efficiently processed using vector instructions and does not rely on the availability of scatter-gather instruction vector instructions.

For computing A*B=C, during the pre-processing stage zero-filling is done for matrix B and it is split into Sparse B and Dense B. All the buckets satisfying the threshold go to Dense B and the other buckets contribute to Sparse B. Because of doing SpGEMM using the scatter-vector approach, doing zero-filling and splitting the matrix B can be beneficial to exploit SIMD vectorization. The threshold and vector width(VW) are chosen based on the processor's SIMD unit width and the type of the floating point computations done(single or double precision).

4.4 Vectorized SpGEMM Algorithm

The Algorithm to compute each row in resultant C matrix can be described as below:

- Each row of the resultant matrix C is assigned to a thread using dynamic scheduling.

- The number of non-zeroes in the current row of C contributed by Dense B and Sparse B is computed by traversing the corresponding rows of A, Dense B and Sparse B by making use of the Scatter Vector.

- The memory required to store the column indexes and values of the non-zeroes in

the current row being processed is allocated using Memory Manager described above.

- Column Index Pointer and Value Pointer corresponding to this row in the resultant C matrix are updated.

- The actual values and column indexes of the current Row contributed by the Dense B and Sparse B are computed respectively.

- While computing the contribution from Dense B, SIMD vectorization is used as all the non-zero elements within a bucket of size VECTOR_LENGTH have continuous indexes.

- Once the current row is entirely computed, the Scatter Vector is reset and the next Row is processed.

Algorithm 4: Vectorized SpGEMM Algorithm to compute a Row in C

```
i ← rowId
thread_id ← omp_thread_num()
nnz_row_count ← 0
SV[:] ← −1

for vp ← 0; vp < nnzARow[i]; vp++ do
    v ← *(JAStart[i] + vp)
    for kp ← 0; kp < nnzBDenseRow[v]; kp+ = vectorLength do
        k ← *(JBDenseStart[v] + kp)
        if SV[k] == −1 then
            #pragma simd
            for j ← 0; j < vectorLength; j++ do
                SV[k + j] = nnz_row_count + j
            end
            nnz_row_count+ = vectorLength
        end
    end
end

for vp ← 0; vp < nnzARow[i]; vp++ do
    v ← *(JAStart[i] + vp)
    for kp ← 0; kp < nnzBSparseRow[v]; kp++ do
        k ← *(JBSparseStart[v] + kp)
        if SV[k] == −1 then
            SV[k] = nnz_row_count
            nnz_row_count++
        end
    end
end

//get memory from allocator to hold columnIndex and values for this row of C
JC ← columnIndexAllocator.allocate(thread_id, nnz_row_count)
C ← valueAllocator.allocate(thread_id, nnz_row_count)

JC[:] ← 0
C[:] ← 0

JCStart[i] ← JC
CStart[i] ← C
```

```
//Adding Dense B contribution
for vp ← 0; vp < nnzARow[i]; vp++ do
    v ← *(JAStart[i] + vp)
    aVal ← (*(AStart[i] + vp))
    JBDS ← JBDenseStart[v]
    nnzBRow ← nnzBDenseRow[v]

    for kp ← 0; kp < nnzBRow; kp+ = vectorLength do
        bPtr ← BDenseStart[v] + kp
        col ← *(JBDS + kp)
        cIndex ← SV[col]

        cPtr ← &C[cIndex]
        jcPtr ← &JC[cIndex]

        #pragma simd
        for j ← 0; j < vectorLength; j++ do
            jcPtr[j] ← (col + j)
        end

        #pragma simd
        for j ← 0; j < vectorLength; j++ do
            cPtr[j]+ = (aVal * bPtr[j])
        end
    end
end

//Adding Sparse B contribution
for vp ← 0; vp < nnzARow[i]; vp++ do
    v ← *(JAStart[i] + vp)
    aVal ← (*(AStart[i] + vp))
    JBSS ← JBSparseStart[v]
    nnzBRow ← nnzBSparseRow[v]
    bPtr ← BSparseStart[v]

    for kp ← 0; kp < nnzBRow; kp++ do
        col ← *(JBSS + kp)
        pos ← *(SV[col])
        JC[pos] = col
        C[pos]+ = aVal * (*(bPtr + kp))
    end
end
```

4.5 Implementation Details and Results

The below machines were used for benchmarking the results:

- Machine 1: Intel Xeon Phi CPU 7250 @ 1.40GHz with 68 cores, AVX-512 (512 bit vector instruction set) and 34 MB L2 Cache.
- Machine 2: Intel Xeon CPU E5-2680 v4 @ 2.40GHz with 28 cores, AVX-2(256 bit vector instruction set) and 35 MB L3 Cache.

Table 4.1 has the list of matrices used for the tests. Appendix A has description about these matrices.

Figures 4.4, 4.5, 4.6 show the performance of Intel MKL vs. DSR_AVX(Vectorized SpGEMM) on Machine 1. Figures 4.8, 4.9, 4.10 show the performance of Intel MKL vs. DSR_AVX(Vectorized SpGEMM) on Machine 2. The time for splitting B is not included in the above figures. The matrices are divided into three sets based on the performance(GFLOPS) range. The results presented are for single precision computations (similar trends for double precision).

Figures 4.7 and 4.11 show the ratio of the time taken for splitting matrix B into sparse B and Dense B divided by the time for 1 iteration of SpGEMM for Machine 1 and Machine 2 respectively.

Table 4.1: Matrices Used

1	2cubes_sphere
2	BioGRID_unweighted_graph
3	cage12
4	cant
5	cit-HepPh
6	com-amazon.ungraph
7	com-dblp.ungraph
8	com-youtube
9	consph
10	cop20k_A
11	DIP_unweighted_graph
12	email-Enron
13	facebook_combined
14	filter3D
15	hood
16	loc-gowalla_edges
17	m133-b3
18	mac_econ_fwd500
19	majorbasis
20	mario002
21	mc2depi
22	mono_500Hz
23	offshore
24	patents_main
25	pdb1HYS
26	poisson3Da
27	pwtk
28	qcd5_4
29	rma10
30	roadNet-CA
31	scircuit
32	shipsec1
33	twitter_combined
34	webbase-1M
35	web-BerkStan
36	web-Google
37	web-NotreDame
38	WIPHI_graph

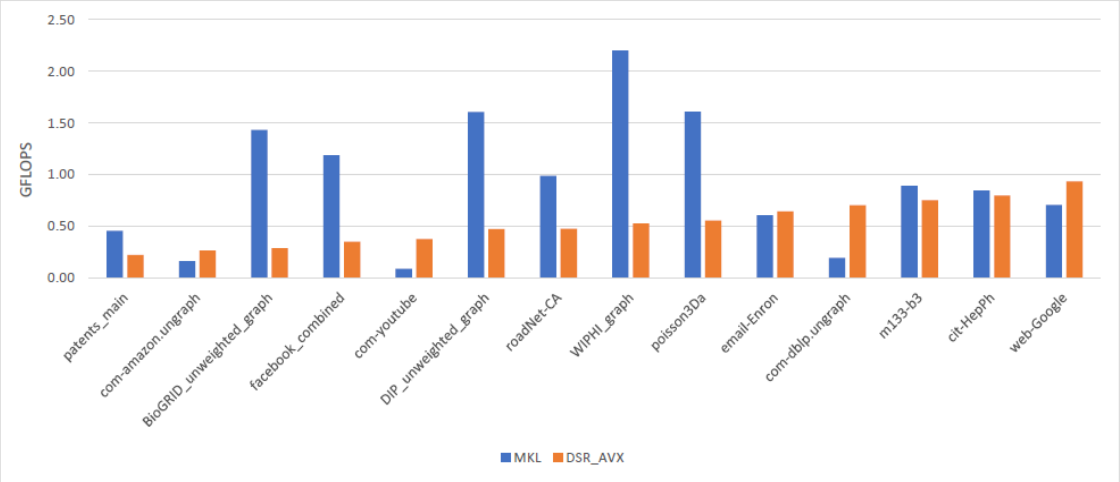

Figure 4.4: Performance comparison on Machine 1 - Set I

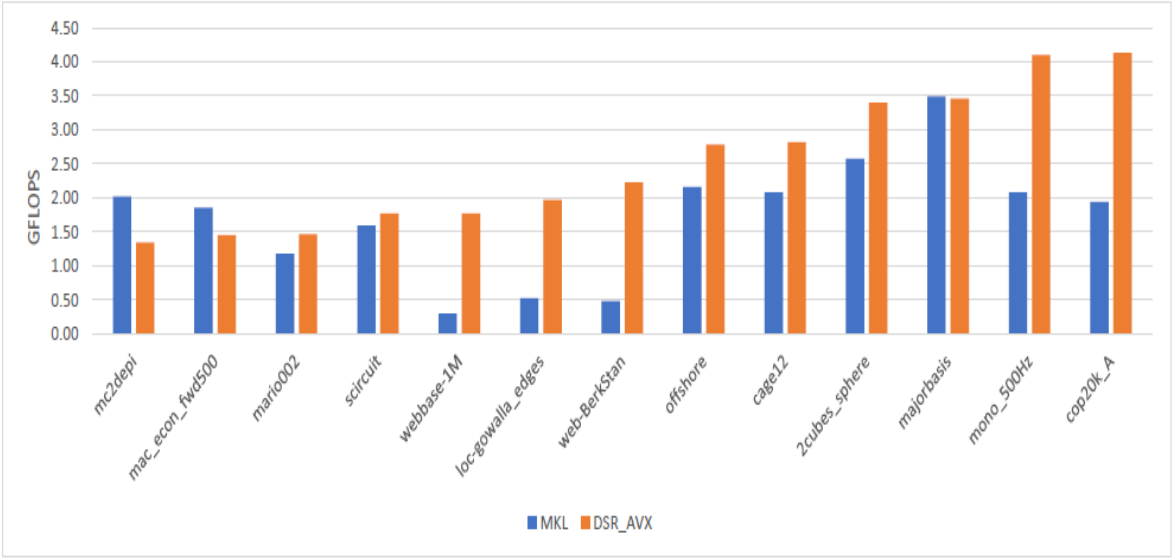

Figure 4.5: Performance comparison on Machine 1 - Set II

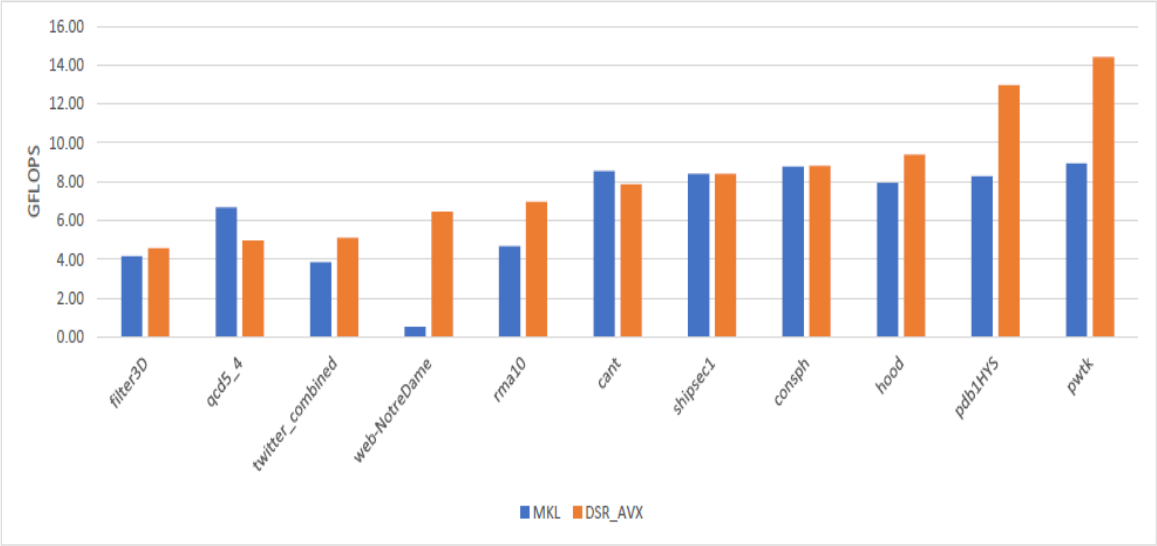

Figure 4.6: Performance comparison on Machine 1 - Set III

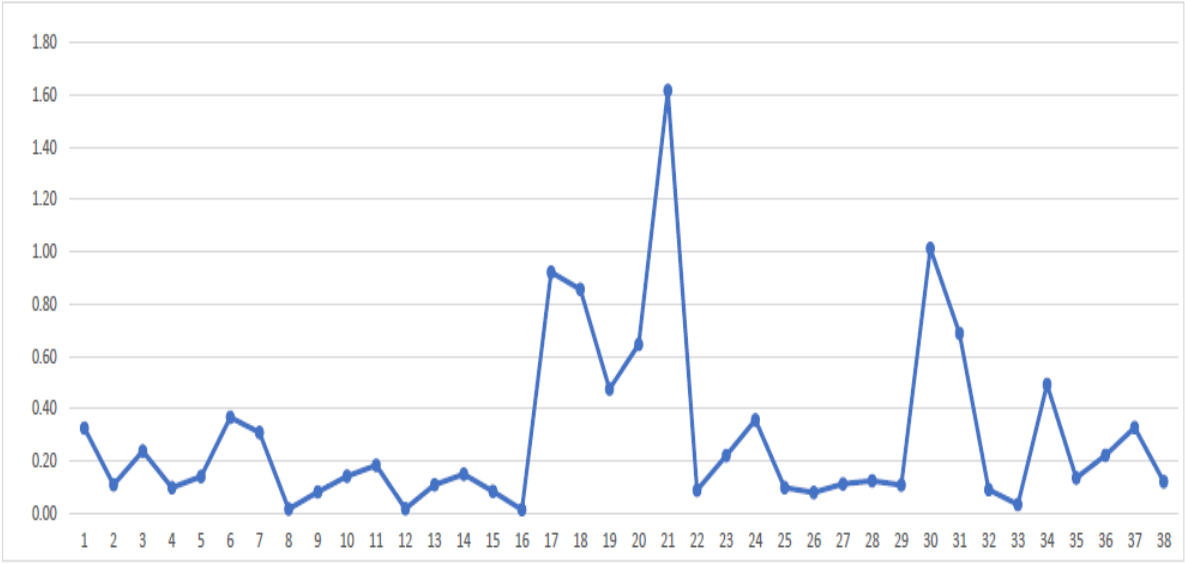

Figure 4.7: Ratio of times of (split B)/(1 iteration of SpGEMM) on Machine 1

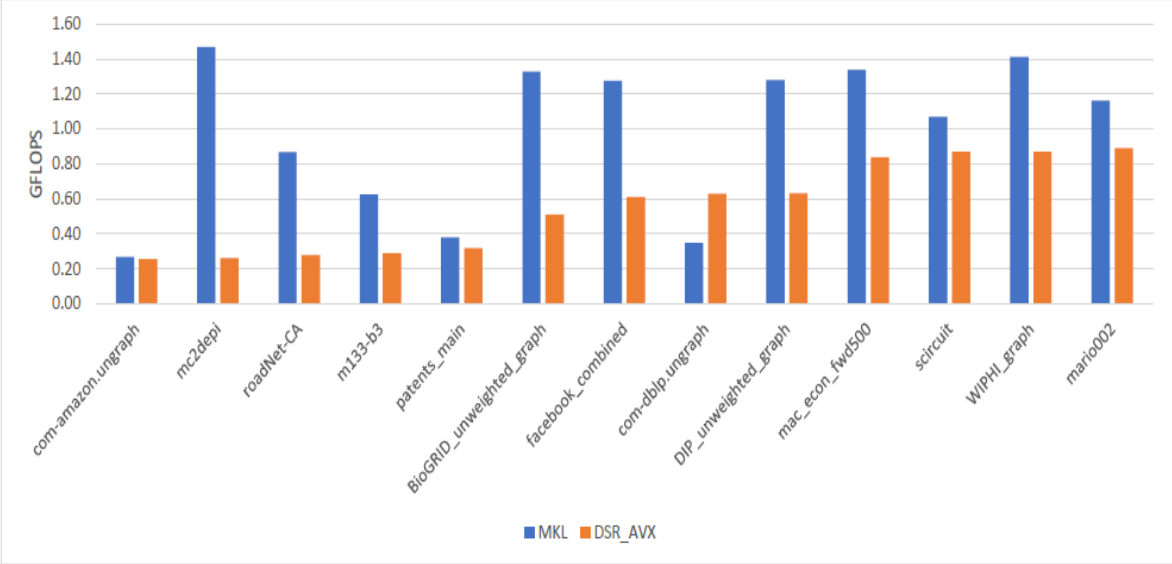

Figure 4.8: Performance comparison on Machine 2 - Set I

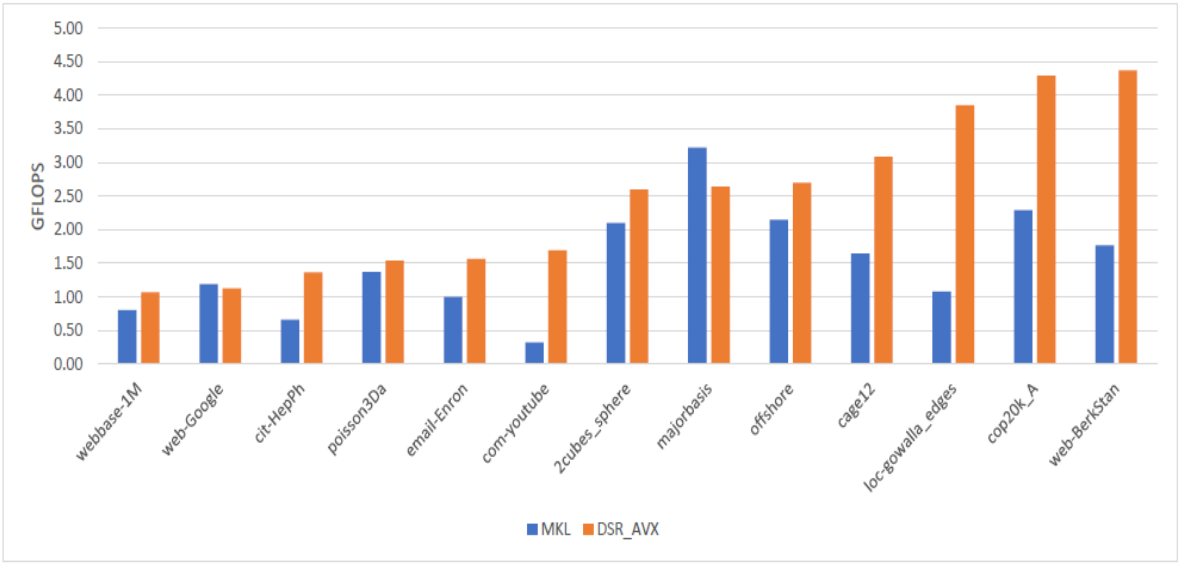

Figure 4.9: Performance comparison on Machine 2 - Set II

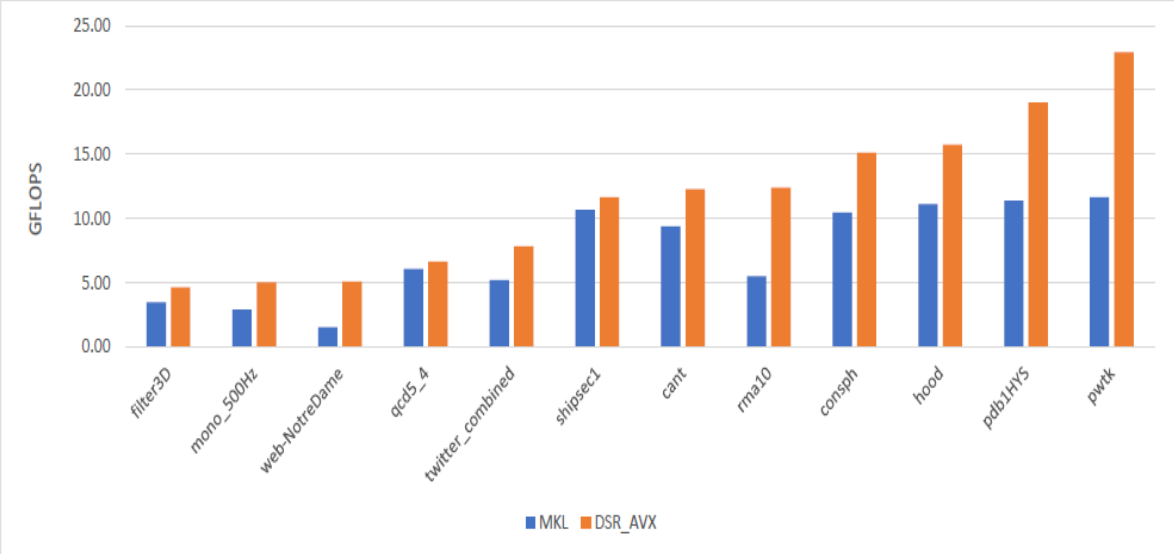

Figure 4.10: Performance comparison on Machine 2 - Set III

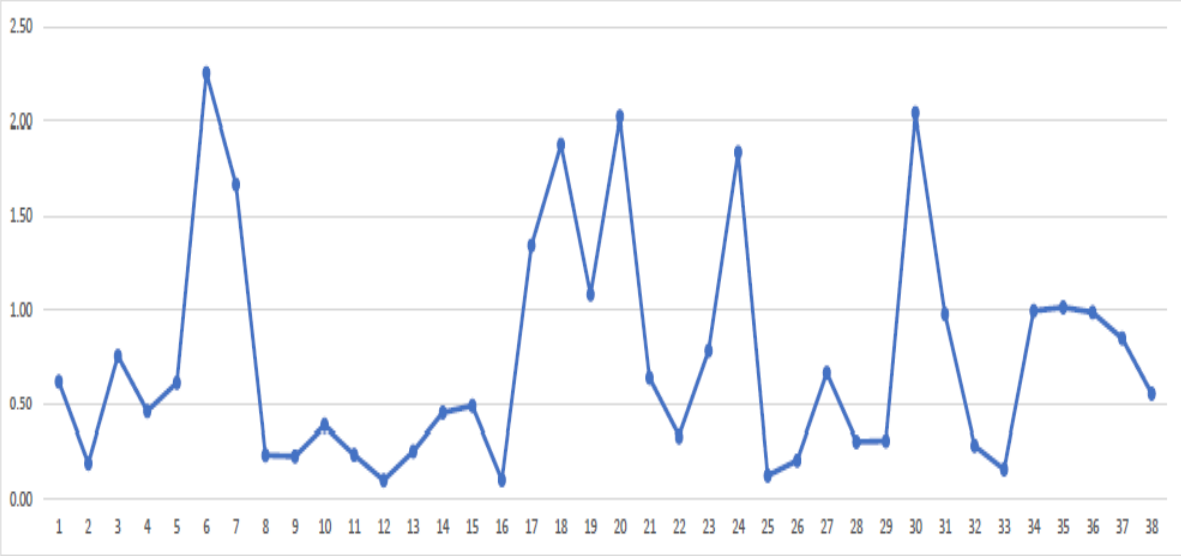

Figure 4.11: Ratio of times of (split B)/(1 iteration of SpGEMM) on Machine 2

It can be observed that the new approach does better than MKL for majority of the matrices in the test set. DSR_AVX loses out to MKL for low-throughput matrices(Set I) on both the machines. It it observed that for most of the matrices in Set I, very few number of non-zeroes go into the Dense B matrix after splitting B. Since DSR_AVX is highly optimized for exploiting Vectorization in Dense B, these matrices do not tend to perform well compared to the rest of the matrices. Further, a hybrid scheme can be implemented that selects the DSR_AVX approach(Vectorized SpGEMM) or DSR approach(using DSR representation without split B) depending on the sparsity nature of the matrices.

Some algorithms like graph clustering which have the property of iterative convergence use SpGEMM as a core primitive. As there is some pre-processing overhead for splitting matrix B, it will be beneficial to use this approach in algorithms which have repeated SpGEMM with one of the matrices fixed, so that the pre-processing time can be amortized over some iterations. This idea is explored in Chapter 5.

Chapter 5: SpGEMM Applications

General Sparse Matrix-Matrix multiplication(SpGEMM) is a fundamental building block and an important primitive for many graph analytics applications. An efficient implementation of this key primitive is critical to the performance of many of these applications. In this chapter, experiments on Regularized Markov Clustering(RMCL)[28] are discussed which uses SpGEMM in its algorithm. Later, a new approach to do K-Truss decomposition of a Graph (using it's adjacency matrix formulation) using a variant of SpGEMM is explored.

5.1 Regularized Markov Clustering(R-MCL)

5.1.1 Background

Graph clustering is one of the key operations in graph mining which is used to detect communities in networks pertaining to many domains. Many algorithms have been proposed to solve the problem of graph clustering. In the bioinformatics community, Markov Clustering(MCL) [10] algorithm based on stochastic flow simulation has gained prominence. Satuluri et al. [28] pointed out that MCL algorithm tends to produce too many clusters and proposed Multi-Level Regularized Markov Clustering(MLR-MCL) [28] which addresses the limitations of MCL. Some previous

studies [21] analyzed that Regularized Markov Clustering (R-MCL) is the most time-consuming component in MLR-MCL algorithm and have improved it's performance by parallelizing R-MCL.

R-MCL has iterative SpGEMM as a core component. An attempt has been made to improve the performance of R-MCL algorithm using the SpGEMM implementations discussed in Chapter 4.

Algorithm 5: R-MCL Algorithm adapted from [28]

```
A = A + I
M = Mg = AD^-1
repeat
    M = M*Mg
    M = inflate(M, r)
    M = prune(M)
until M converges;
```

Algorithm 5 [28] shows the high-level pseudo code of R-MCL. A is the adjacency matrix representation corresponding to the graph. M is first initialized as the canonical transition matrix Mg, by multiplying A and the inverse of the diagonal degree matrix D. R-MCL algorithm involves the following three operations in each iteration.

- The regularize operation calculates M = $M \times Mg$.
- The inflate operation raises each element of M to the power r (typically 2) and normalizes each column (such that sum of each column is 1).
- The prune operation removes elements whose values are smaller than a threshold, which is heuristically computed based on the average and maximum values within a column of M.

In the R-MCL implementation each of the three steps described above can be parallelized. The SpGEMM implementation discussed in Chapter 4 is used to perform the regularize operation discussed above. As threshold and prune operations for a column are independent of other columns, they can be parallelized as well. The work corresponding to all the columns is dynamically distributed across the threads using dynamic scheduling.

Algorithm 6: Parallel R-MCL Algorithm

```
A = A + I
M = Mg = AD^-1
repeat
    #pragma omp parallel for schedule(dynamic, stride)
    for colIdx = 0; colIdx < M.cols; colIdx++ do
        compute_column(M, Mg, colIdx)   //uses SpGEMM kernel
        inflate(M, r, colIdx)
        prune(M, colIdx)
    end
until M converges;
```

Depending on the matrix format used (row or column major representation), the R-MCL algorithm can be adapted accordingly.

5.1.2 Implementation Details and Results

The below machines were used for benchmarking the results:

- Machine 1: Intel Xeon Phi CPU 7250 @ 1.40GHz with 68 cores, AVX-512 (512 bit vector instruction set) and 34 MB L2 Cache.
- Machine 2: Intel Xeon CPU E5-2680 v4 @ 2.40GHz with 28 cores, AVX-2(256 bit vector instruction set) and 35 MB L3 Cache.

The results presented are for single precision computations (similar trends for double precision). The RMCL algorithm is run for 5 iterations in all the results presented.

Figures 5.1, 5.2 and 5.3 show the ratio of 5 iterations of RMCL time for MKL divided by time for DSR_AVX on Machine 1. Figures 5.4, 5.5 and 5.6 show the ratio of 5 iterations of RMCL time for MKL divided by time for DSR_AVX on Machine 2. It can be observed that in majority of the cases the speed-up is below 1.0. The reason is as follows: For the DSR_AVX algorithm, the matrix B has to be fixed so that the pre-processing is done only once. This matrix B is in row-major format. But as the R-MCL algorithm requires the matrix to be column stochastic, there is lot of overhead to do threshold, pruning and normalization stages. This is because of having a row major representation and matrix has to be column stochastic. Due to this, there is a lot of synchronization overhead and additional work that each thread has to do which is impacting the performance.

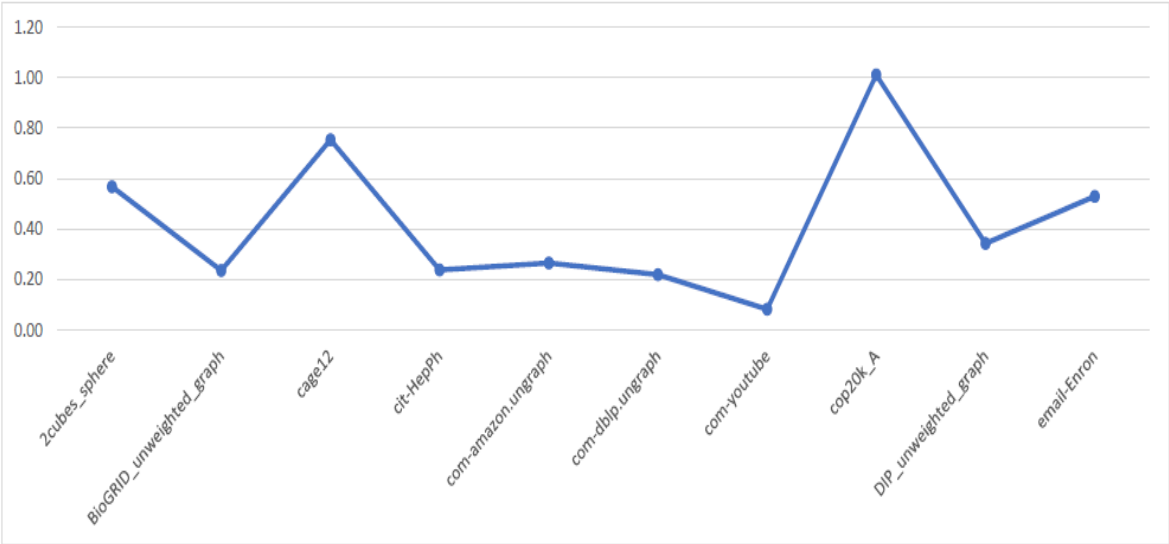

Figure 5.1: Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 1

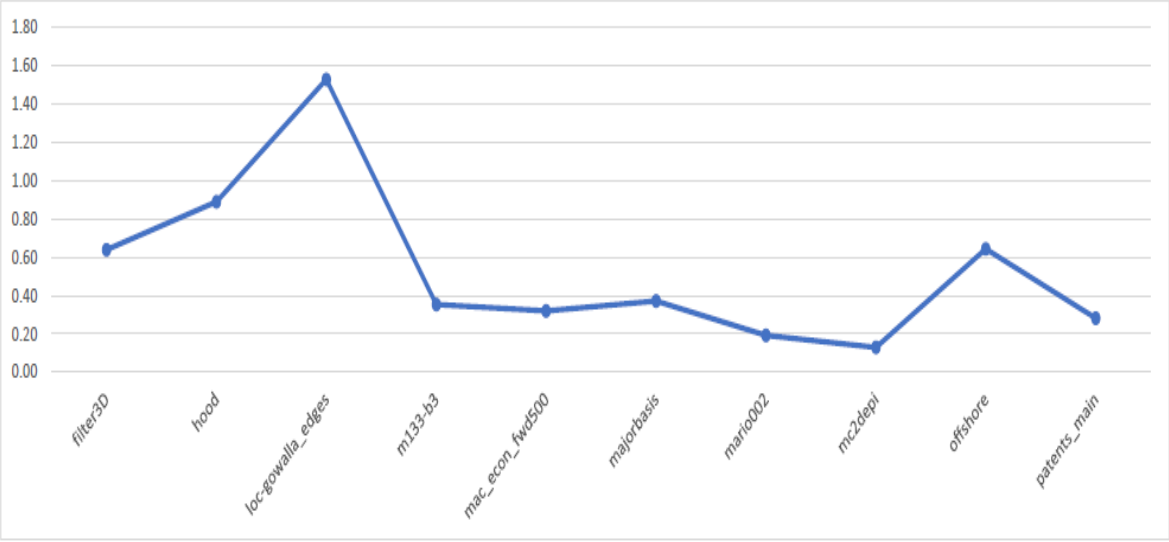

Figure 5.2: Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 1

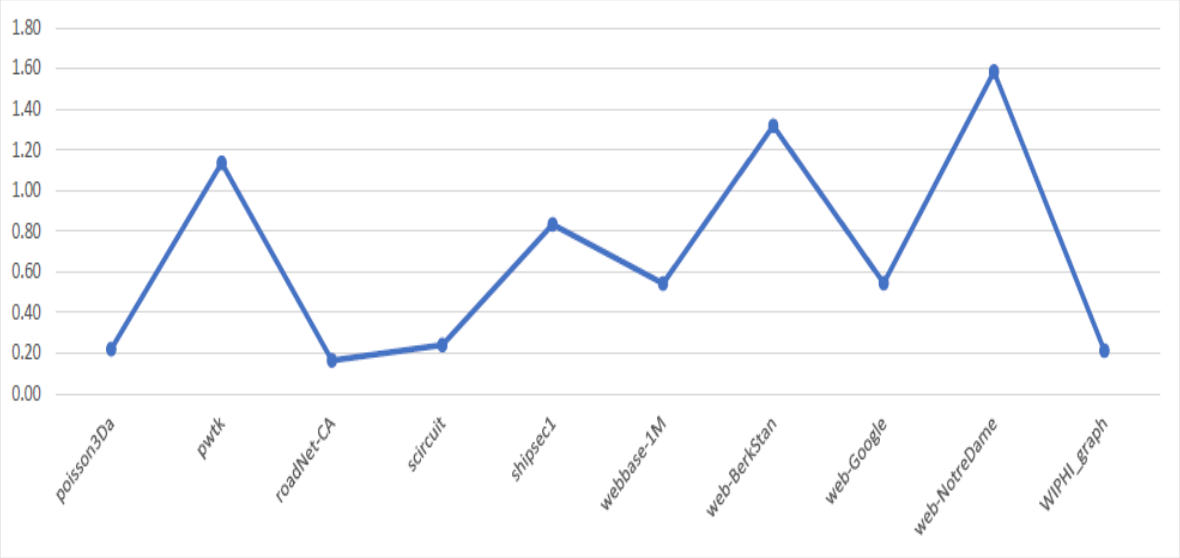

Figure 5.3: Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 1

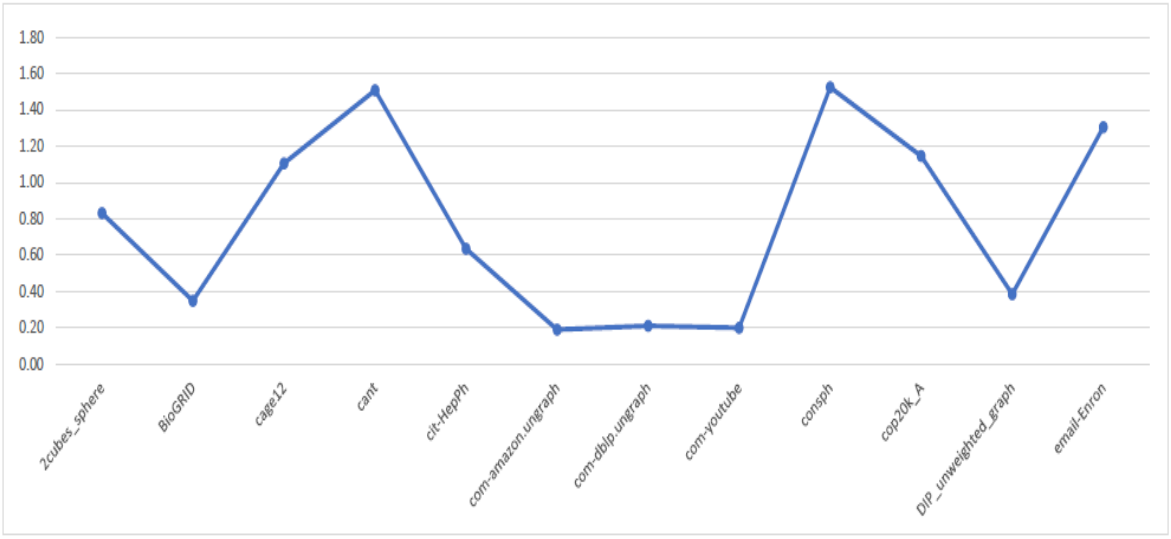

Figure 5.4: Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 2

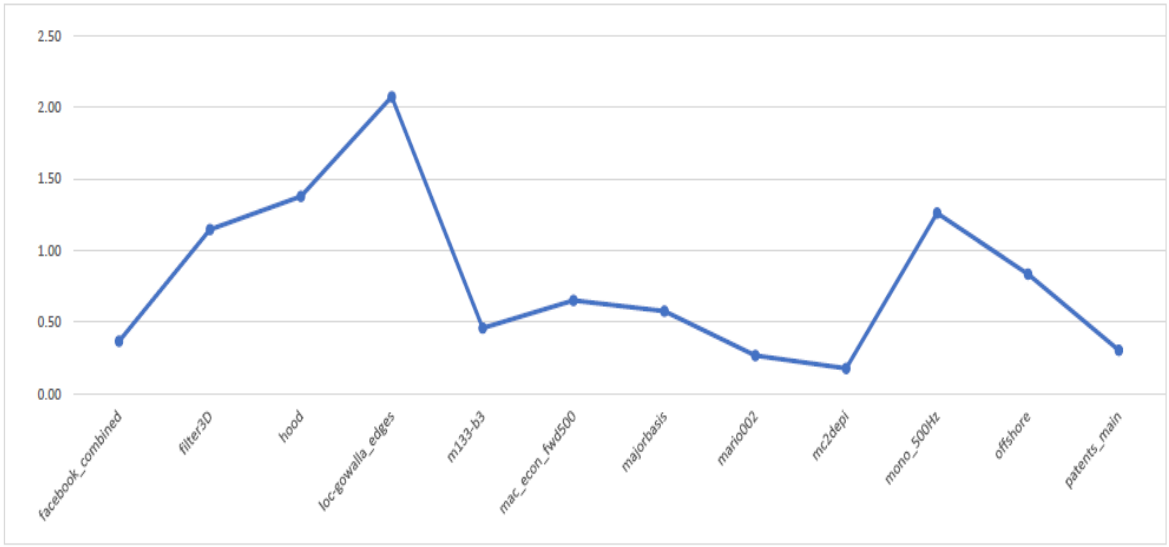

Figure 5.5: Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 2

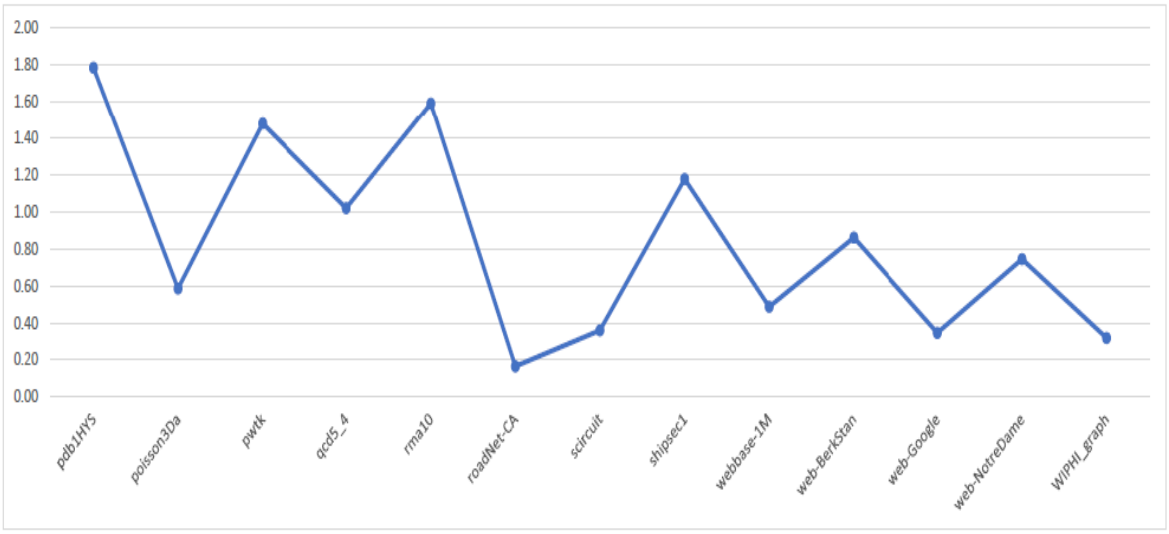

Figure 5.6: Ratio of R-MCL time for (MKL/DSR_AVX) on Machine 2

As the DSR_AVX algorithm is losing out due to the reason mentioned above, computing SpGEMM using DSR format without splitting B is considered. The advantage with this is that there is no constraint that B has to be fixed. The property that $A * B = (B^T * A^T)^T$ is used and R-MCL algorithm is run with row-major representation. Due to the transpose, the rows are the columns in the original matrix. Now each thread can independently compute C and do threshold, prune, normalization steps without any synchronization overhead and without any additional work.

Figures 5.7, 5.8 and 5.9 show the ratio of 5 iterations of RMCL time for MKL divided by time for the case of DSR on Machine 1. Figures 5.10, 5.11 and 5.12 show the ratio of 5 iterations of RMCL time for MKL divided by time for the case of DSR on Machine 2. It can be observed that in majority of the cases the speed-up is above 1.5 and the maximum speed-up achieved is 4.5.

Though, DSR_AVX algorithm gave less performance for this particular case due to the constraints explained above, it is worthy to explore other applications where repeated SpGEMM is done.

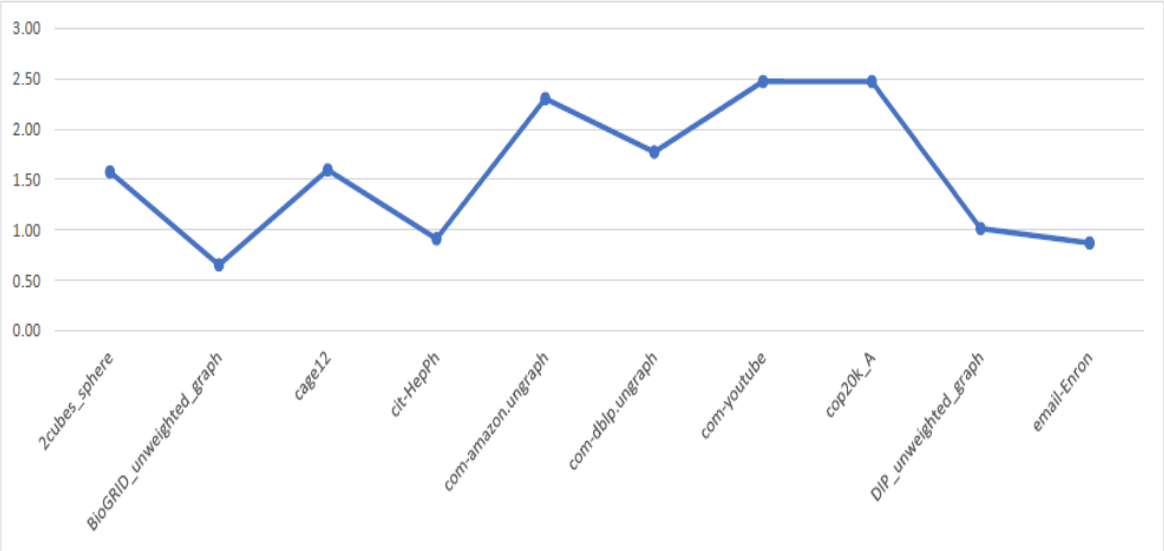

Figure 5.7: Ratio of R-MCL time for (MKL/DSR) on Machine 1

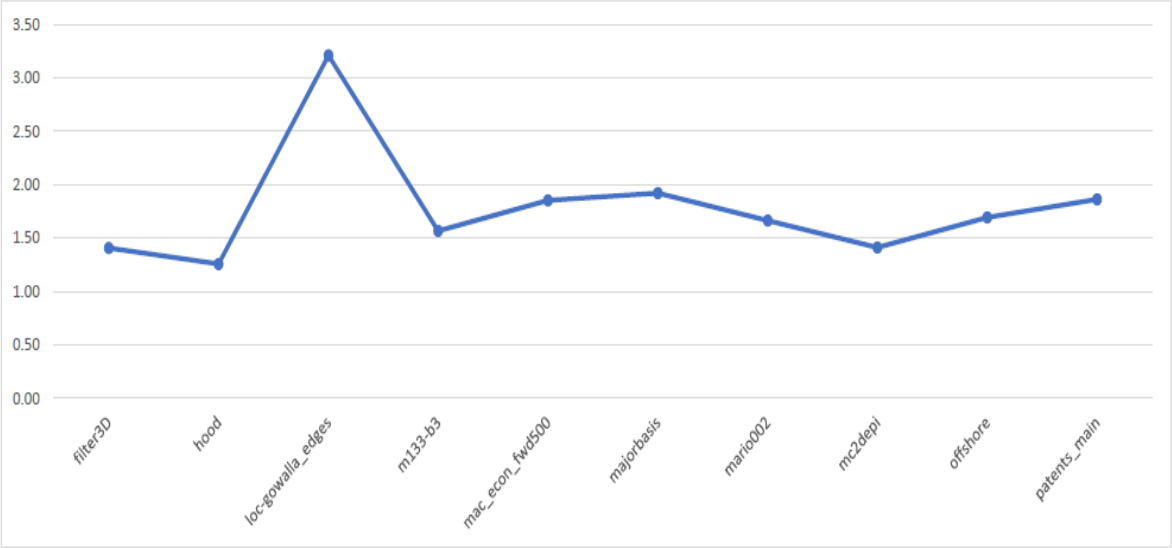

Figure 5.8: Ratio of R-MCL time for (MKL/DSR) on Machine 1

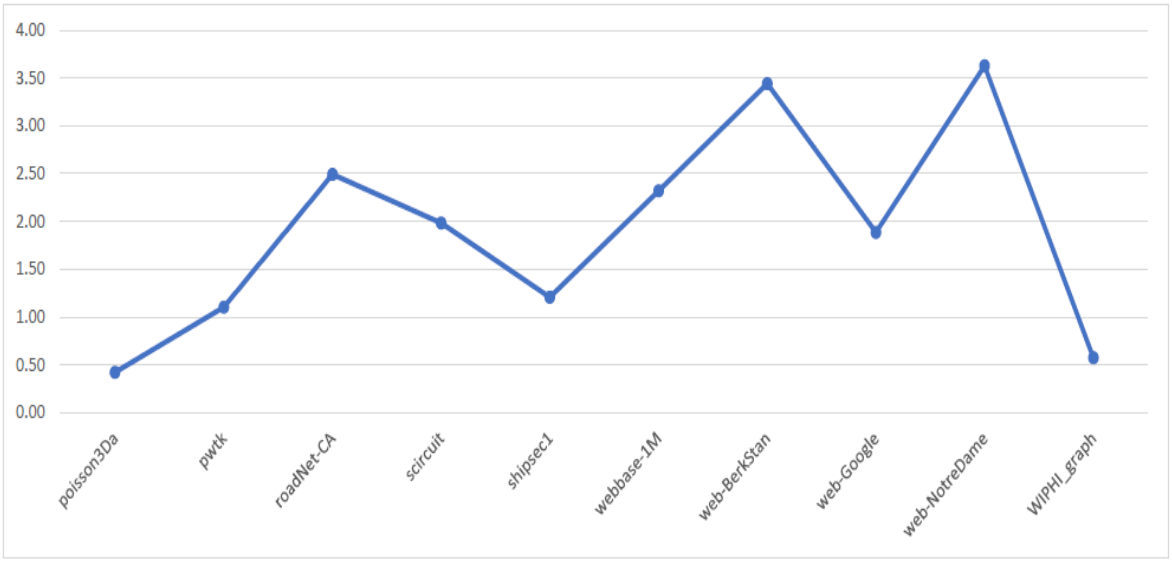

Figure 5.9: Ratio of R-MCL time for (MKL/DSR) on Machine 1

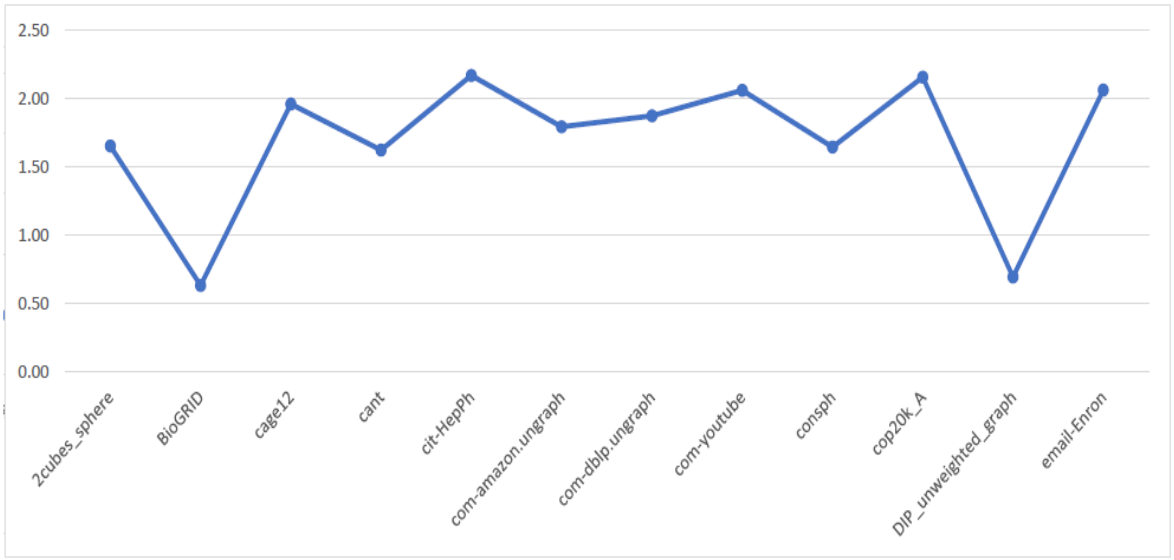

Figure 5.10: Ratio of R-MCL time for (MKL/DSR) on Machine 2

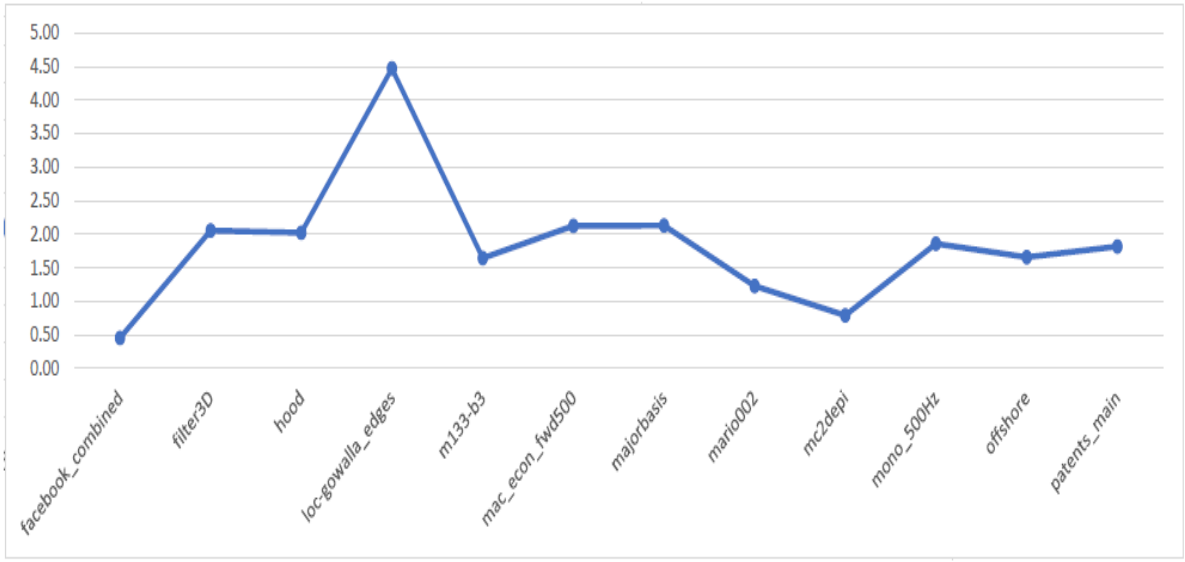

Figure 5.11: Ratio of R-MCL time for (MKL/DSR) on Machine 2

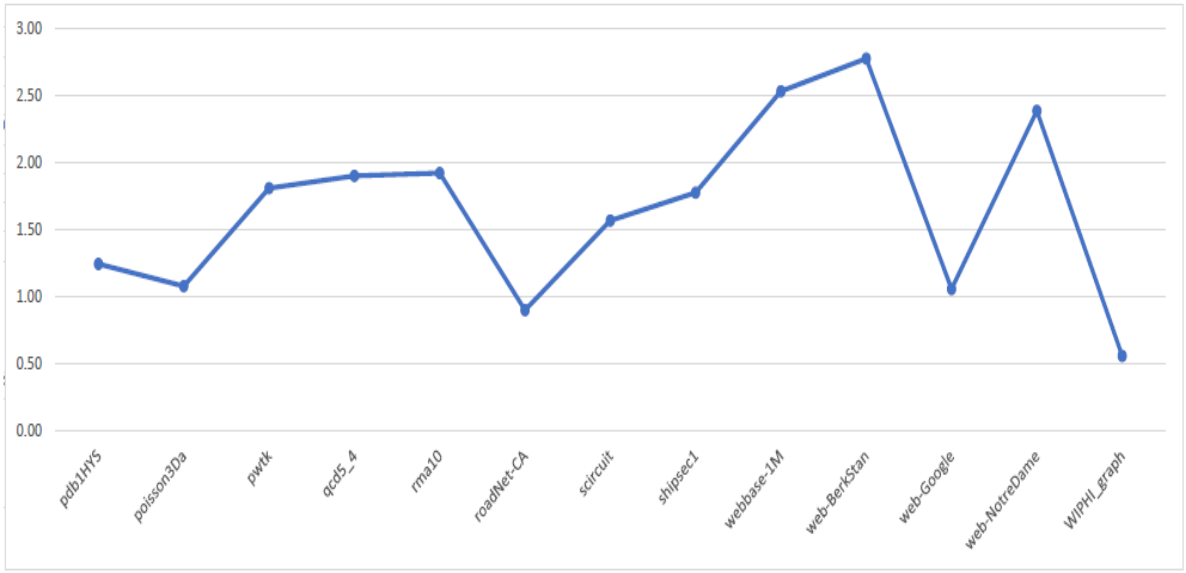

Figure 5.12: Ratio of R-MCL time for (MKL/DSR) on Machine 2

5.2 K-Truss Decomposition

Given a graph G, a k-truss is a sub-graph G_k such that each edge is contained in at least (k-2) triangles in the same sub-graph [5], [32]. The truss number of an edge e in G is defined as the maximum value k such that $e \in \mathrm{G}_k$. k_{max} denotes the maximum truss number of any edge in the graph. The problem of Truss decomposition of a graph G is finding the (non-empty) k-trusses of G for all $2 \leq k \leq k_{max}$ [12].

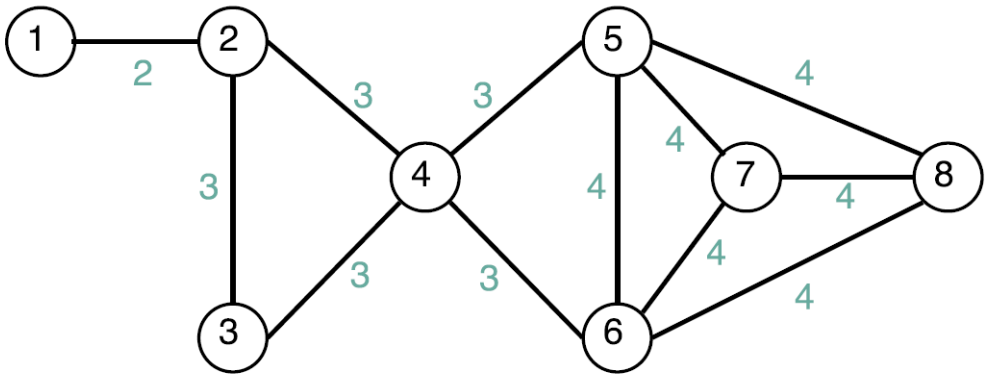

Figure 5.13: Truss decomposition of a graph. Each edge is labeled with it's truss number

Figure 5.13 [29] illustrates truss decomposition of a graph. Each of the edges is labeled with it's truss number which represents the number of triangles that edge is a part of.

In the analysis of large networks, it is more feasible to focus on smaller areas of the network that reflect important properties of the network. The definition of k-truss is based on triangles which are the fundamental building blocks of a network [34], [33]. In the context of a social network, a triangle implies a strong tie among three

friends, or two friends having a common friend. The k-truss strengthens every (edge) connection in it by at least (k - 2) ties. It implies that the more common friends two people have, the stronger their connection.

Algorithm 7: Common incidence matrix K-Truss Decomposition algorithm[27]

$R \leftarrow EA$
$x \leftarrow find((R == 2)1 < k - 2)$
while x **do**
  $E_x \leftarrow E(x, :)$
  $E \leftarrow E(x_c, :)$
  $R \leftarrow E(X_c, :)A$
  $R \leftarrow R - E(E_x E_x^T - diag(E_x E_x^T))$
  $x \leftarrow find((R == 2)1 < K - 2)$
end

Algorithm 8: Algorithm to do K-Truss Decomposition of a Graph

$K_Truss \leftarrow 3$

while $nnzC! = 0$ **do**
  while $nnzC! = nnzA$ **do**
    //compute support values and remove edges
    //with support less than (K_Truss - 2)
    $C = (A * A).A$
  end
  $K_Truss++$
end

A common matrix based implementation for K-Truss [27] operates over the edge or incidence matrix, and keeps an updated adjacency matrix. Each iteration removes edges from E that do not have sufficient support (i.e., the set x). The algorithm requires a multiplication of E and A (where E is the remaining edges, which is reduced

Algorithm 9: Algorithm to process a row of A in Truss Decomposition

```
input : A, K_Truss
output: C
i = rowIdx
thread_id = omp_get_thread_num()
SV[:]

for vp = 0; vp < A.nnzRow[i]; ++vp do
    int k = *(A.JStart[i] + vp)
    SV[k] = 0
end
for vp = 0; vp < A.nnzRow[i]; ++vp do
    v = *(A.JStart[i] + vp)
    for kp = 0; kp < A.nnzRow[v]; ++kp do
        k = *(A.JStart[v] + kp)
        SV[k]++
    end
end
index=0
for vp = 0; vp < A.nnzRow[i]; ++vp do
    k = *(A.JStart[i] + vp)
    if SV[k] >= K_Truss - 2 then
        *(C.JStart[i]+index)=k
        index++
    end
end
```

as the algorithm progresses), one of E_x and E_x^T, and one of E and a matrix of size E_x. The algorithm is an incremental approach, that removes edges from E and updates the support at each iteration.

In developing an improved algorithm, a formulation that only requires the adjacency matrix A is considered. Algorithm 8 shows the high level pseudo-code for K-Truss Decomposition of a graph G represented by an adjacency matrix representation A. The Algorithm starts with enumerating a 3-Truss. The outer most *while* loop in the algorithm keeps increasing the K_Truss value till K_max for the graph

is reached, i.e., till there are no more edges in the graph ($nnz = 0$). The function $(A * A).A$ computes the support of all the edges and removes those that do not have the corresponding support. As an edge is removed, it also affects the support of all other edges connected to its vertices. The innermost while loop keeps removing the edges until all remaining edges satisfy the current support value ($A.nnz \neq nnz$, i.e., no additional edges drop out in this iteration).

The $(A * A).A$ multiplication has the benefit of a known output structure (i.e., the output structure is masked by the $.A$ operation), and so the output memory can be pre-allocated. This is a significant benefit, as computing the output data structure is a major source of overhead in an SpGEMM operation. Algorithm 9 shows the pseudo-code for computing $(A * A).A$. For each vertex, all the neighbors are visited, and for each such neighbor, the intersection is calculated. If there is an intersection, the supports are updated using the Scatter Vector. At the end, those edges that have support less than the threshold are removed.

The benefits of this approach are a reduced data structure size (A instead of E) and the prior knowledge of the output structure. The achieved performance relative to existing approaches will change based on the structure of the graph being processed. In order to improve performance for a wider range of graphs (especially graphs that have a low number of edges "dropping out" at each iteration), an incremental approach that can update A without performing the multiplication can be used in the cases where the drop-out is low.

The main idea behind the optimization is that instead of forming a reduced matrix (by removing the entries corresponding to edges without sufficient support) and explicitly forming its product with itself, it is better to deduce how much less the

product values would have been at various elements if the to-be-removed edges had not added their contributions. For a row i, the adjustments to the contributions from any to-be-removed edge $A(i,j)$ can be determined simply by traversing the non-zeros of row j in A.

An analysis of the amount of work with the direct versus incremental approach is as follows. Let d_i denote the degree of vertex i in the graph, which is the number of nonzero elements in row i of A. Since A is an adjacency matrix, it is symmetric, i.e., there are also d_i non-zeros in column i. Let r_i denote the number of edges of vertex i that do not have sufficient support and get removed after the matrix product $(A * A).A$ is computed. The number of operations in forming $(A * A).A$ is $\sum_{i=1}^{N} d_i^2$. For the next iteration, with the direct approach, the same product $(A * A).A$ would be computed with a reduced matrix with $d_i - r_i$ non-zeros in row/col i, with cost $\sum_{i=1}^{N} (d_i - r_i)^2$. Instead, with the use of incremental approach, each to-be-removed entry $A(i,j)$ would form a product with the original non-zeros in A in row j. Since the removed entries are symmetric, this means that r_i entries would perform d_i operations each, i.e., the total number of operations would be $\sum_{i=1}^{N} (d_i * r_i)$. When the drop-out is low, significant savings can be expected from the use of the incremental approach. Further, a hybrid scheme can be implemented that selects the direct or incremental approach depending of which would require fewer operations.

Algorithm 10: GPU Kernel - Compute K Truss

```
__shared__ count[WARP_NUM]
local_id = (threadIdx.x)%(WARP_SIZE)
warp_id = (threadIdx.x)/(WARP_SIZE)
global_warp_id = (WARP_NUM*blockIdx.x) + warp_id
local_size = WARP_SIZE
SV[:]
if local_id == 0 then
    count[warp_id]=0
end
__syncthreads()
for rowId = global_warp_id; rowId < A.rows;
rowId+=gridDim.x*WARP_NUM do
    for vp = local_id; vp < A.nnzRow[rowId]; vp+ = local_size do
        k = *(A.JStart[rowId]+vp)
        SV[k]=0
    end
    __syncthreads()
    for vp = 0; vp < A.nnzRow[rowId]; vp + + do
        v = *(A.JStart[rowId]+vp)
        for kp = local_id; kp < A.nnzRow[v]; kp+ = local_size do
            k = *(A.JStart[v] + kp)
            SV[k]++
        end
        __syncthreads()
    end
    for vp = local_id; vp < A.nnzRow[rowId]; vp+ = local_size do
        k = *(A.Start[rowId]+vp)
        if SV[k] >= K_Truss − 2 then
            idx=atomicAdd(count[warp_id], 1)
            *(C.JStart[rowId]+idx) = k
        end
    end
    __syncthreads()
    if local_id == 0 then
        C.nnzRow[rowId] = count[warp_id]
        count[warp_id] = 0
    end
    __syncthreads()
end
```

Chapter 6: Conclusion

An attempt has been made to gain insights into the cause of low performance of GPU implementations of SpGEMM relative to sparse matrix-vector multiplication. Experiments are performed on synthetic banded matrices to gain insights into the low performance. Inadequate concurrency was identified as a root cause of Scatter Vector based SpGEMM and an improved implementation which outperforms existing implementations was devised.

In the case of multi-core systems, a novel matrix storage representation was devised which aids to do SpGEMM in one phase as opposed to the existing libraries like MKL which have two phases (where number of non-zeroes are computed in the first phase and the computation is done in the second phase). A new approach was devised to exploit SIMD vectorization in processors like Intel KNL. Experimental results show that the new approach demonstrates better performance than Intel MKL for a majority of the matrices in the test set. An attempt has been made to improve the performance of Regularized Markov Clustering(R-MCL) algorithm using the new SpGEMM implementation.

A new approach to do K-Truss decomposition of a Graph using a variant of SpGEMM has been proposed which uses adjacency matrix formulation.

Appendix A: Test Set of Matrices

The test set of sparse matrices includes web connection graphs, protein models, structural matrices, circuit simulation graphs and macroeconomic models. These matrices are diverse and inherently irregular in their sparsity structure. The set includes all the matrices used in implementations by Liu et al.[18], Rakshith et al.[16], Qingpeng [22] and few matrices from SNAP data set[17].

Description about all the matrices studied in this thesis is presented below.

1. dblp: BLP bibliography provides lists of research papers and citations in the field of computer science. Each node represents an author, and an edge between two nodes indicates that the authors have published at least one joint paper
2. amazon: amazon product co-purchase network
3. facebook: facebook anonymized friend network
4. patents_main : patent network of US 240,547 patents
5. cit-HepPh: Arxiv HEP-PH (high energy physics) citation graph
6. scircuit: circuit simulation problem
7. youtube: Friendships between users collected from Youtube
8. DIP : An unweighted PPI network of cere- visiae (a species of yeast) obtained from database of interacting proteins

9. web-Google: This dataset is provided by the Google Programming Contest 2002. Each node is a webpage, and a directed edge between nodes is a hyperlink between two webpages

10. BioGRID: An unweighted PPI network of cerevisiae obtained from Database of Protein and Genetic Interactions

11. WIPHI_graph: weighted PPI network with weight adjusted by min-max normalization

12. email-Enron: Enron email communication network

13. roadNet-CA: A road network of California. Nodes represent intersections and edges indicate roads connecting the intersections

14. m133-b3: Simplicial complexes from Homology

15. poisson3Da: computational fluid dynamics problem

16. cage12: DNA electrophoresis network

17. epidemiology(mc2depi): 2D markov model of epidemic

18. protein(pdb1HYS): protein data bank

19. mario002: matrix from mario for which MA47 analyze is slow.

20. 2cubes_spheres: FEM, electromagnetics, 2 cubes in a sphere

21. web-NotreDame: Nodes represent pages from University of Notre Dame and directed edges represent hyperlinks

22. offshore: 3D FEM, transient electric field diffusion

23. economics(mac_econ_fwd500) : macroeconomic model

24. webbase-1M: web connectivity matrix

25. majorbasis: mixed complementarity optimization problem.

26. loc-gowalla: check-in locations shared by the users of Gowalla.

27. twitter: twitter friends network.

28. filter3D: tunable optical filter.

29. QCD(qcd): Quantum chromodynamics.

30. cop20k_A: Accelerator cavity design.

31. web-BerkStan: Nodes represent pages from berkely.edu and stanford.edu domains and directed edges represent hyperlinks.

32. mon_500Hz: 3D vibro-acoustic problem, aircraft engine.

33. hood: INDEED Test Matrix (DC-mh).

34. harbor(rma10): 3D CFD model, Charleston harbor.

35. pwtk: pressurised wind tunnel. stffness matrix.

36. spheres(consph): FEM concentric spheres.

37. ship(shipsec-1): Ship section/detail from production run.

38. cant: FEM cantilever.

Bibliography

[1] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better size estimation for sparse matrix products. In *Proceedings of the 13th International Conference on Approximation, and 14 the International Conference on Randomization, and Combinatorial Optimization: Algorithms and Techniques*, APPROX/RANDOM'10, pages 406–419, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 121–126, New York, NY, USA, 2009. ACM.

[3] Nathan Bell, Steven Dalton, and Luke N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C.123–C.152, 2012. Copyright - 2012, Society for Industrial and Applied Mathematics; Last updated - 2012-09-17.

[4] Edith Cohen. On optimizing multiplications of sparse matrices. In *Proceedings of the 5th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 219–233, London, UK, UK, 1996. Springer-Verlag.

[5] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. 04 2018.

[6] Paolo D'Alberto and Alexandru Nicolau. R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. *Algorithmica*, 47(2):203–213, February 2007.

[7] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[8] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[9] M. Deveci, C. Trott, and S. Rajamanickam. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 693–702, May 2017.

[10] S. V. Dongen. Graph clustering by flow simulation. PhD thesis, University of Utrecht, 2000.

[11] T F Coleman and J J Mor. Estimation of sparse jacobian matrices and graph coloring problems. 20:187–209, 01 1984.

[12] Vijay Gadepally, Jake Bolewski, Dan Hook, Dylan Hutchison, Benjamin A. Miller, and Jeremy Kepner. Graphulo: Linear algebra graph kernels for nosql databases. *CoRR*, abs/1508.07372, 2015.

[13] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, January 1992.

[14] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978.

[15] Rakshith Kunchum. On improving sparse matrix-matrix multiplication on gpus. Master's Thesis, The Ohio State University, 2017.

[16] Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam, Qingpeng Niu, Israt Nisa, and P Sadayappan. On improving performance of sparse matrix-matrix multiplication on gpus. In *Proceedings of the International Conference on Supercomputing*, page 14. ACM, 2017.

[17] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[18] W. Liu and B. Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381, May 2014.

[19] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 339–350, New York, NY, USA, 2015. ACM.

[20] Marcin Mucha and Piotr Sankowski. Maximum matchings in planar graphs via gaussian elimination. *Algorithmica*, 45(1):3–20, April 2006.

[21] Q. Niu, P. W. Lai, S. M. Faisal, S. Parthasarathy, and P. Sadayappan. A fast implementation of mlr-mcl algorithm on multi-core processors. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.

[22] Qingpeng Niu. Characterization and enhancement of data locality and load balancing for irregular applications. PhD Dissertation, The Ohio State University, 2015.

[23] NVIDIA. Nvidia cuSPARSE library, 2017.

[24] Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G. Pudov, Vadim O. Pirogov, and Pradeep Dubey. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, pages 48–57, Cham, 2015. Springer International Publishing.

[25] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 679–688, 2002.

[26] Emanuel H. Rubensson, Elias Rudberg, and Paweł Sałek. Sparse matrix algebra for quantum modeling of large systems. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 90–99, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[27] Siddharth Samsi, Vijay Gadepally, Michael B. Hurley, Michael Jones, Edward K. Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William S. Song, Diane Staheli, and Jeremy Kepner. Static graph challenge: Subgraph isomorphism. *CoRR*, abs/1708.06866, 2017.

[28] Venu Satuluri and Srinivasan Parthasarathy. Scalable graph clustering using stochastic flows: Applications to community discovery. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 737–746, New York, NY, USA, 2009. ACM.

[29] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis. Truss decomposition on shared-memory parallel systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2017.

[30] P. D. Sulatycke and K. Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices. In *Proceedings of the First Merged International Parallel*

Processing Symposium and Symposium on Parallel and Distributed Processing, pages 117–123, Mar 1998.

[31] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. Finding heaviest h-subgraphs in real weighted graphs, with applications. *CoRR*, abs/cs/0609009, 2006.

[32] Jia Wang and James Cheng. Truss decomposition in massive networks. *CoRR*, abs/1205.6693, 2012.

[33] S. Wasserman and K. Faust. Social network analysis: Methods and applications.

[34] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[35] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[36] Ichitaro Yamazaki and Xiaoye S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes, editors, *High Performance Computing for Computational Science – VECPAR 2010*, pages 421–434, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[37] Raphael Yuster and Uri Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 254–260, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.