Sparse Matrix-Matrix Multiplication on Modern Architectures

Kiran Matam [1], Siva Rama Krishna Bharadwaj Indarapu [2], Kishore Kothapalli [3]

Center for Security, Theory and Algorithmic Research (CSTAR),
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India, 500 032.
{{ [1]kiranm, [2]sivaramakrishna.i}@research., [3]kkishore@}iiit.ac.in

***Abstract*—**

Sparse matrix-sparse/dense matrix multiplications, `spgemm` and `csrmm`, respectively, among other applications find usage in various matrix formulations of graph problems. Considering the difficulties in executing graph problems and the duality between graphs and matrices, computations such as `spgemm` and `csrmm` have recently caught the attention of HPC community. These computations pose challenges such as load balancing, irregular nature of the computation, and difficulty in predicting the output size. It is even more challenging when combined with the GPU architectural constraints such as memory accesses, limited shared memory, strict SIMD and thread execution.

To address these challenges on a GPU, we evaluate three possible variations of matrix multiplication (Row-Column, Column-Row, Row-Row) and perform suitable optimizations targeted at sparse matrices. Our experiments indicate that the Row-Row formulation, which mostly outperforms the other formulations, is 3.5x faster on average compared to an optimized multi-core implementation in the Intel MKL library. We extend the Row-Row formulation to a CPU+GPU hybrid algorithm that simultaneously utilizes the CPU also. In this direction, we present heuristics to find the right amount of work division between the CPU and the GPU. Our hybrid row-row formulation of the `spgemm` operation performs 5.5x faster on average when compared to the optimized multi-core implementation in the Intel MKL library.

Our experience indicates that it is difficult to identify right amount of work division between the CPU and the GPU. We therefore investigate a subclass of sparse matrices, *band matrices*, and present an analytical method to identify a good work division when multiplying two *band matrices*.

Our GPU `csrmm` operation performs 2.5x faster on average when compared to a corresponding implementation in the `cusparse` library, which outperforms the Intel MKL library implementation.

***Index Terms*—sparse matrix multiplication, hybrid algorithms, GPGPU, band matrices**

I. Introduction

Advances in multi- and many-core architectures are driving powerful changes in computing. Presently, efficient and scalable solutions for several challenge problems in parallel computing such as FFT [17], sorting [18] are available on varied architectures such as Intel CPUs [16], GPUs [18], and also the IBM Cell [11]. Of these, CPU and GPU based solutions stand-out for contrasting reasons. Current generation GPUs offer the best performance per price of more than 1 TFLOP for as little as $400. Modern multicore CPUs are not far behind, and in some computations offer a near-matching performance compared to GPUs. In fact, in a recent work [33], the authors show evidence to indicate that on a class of throughput-oriented problems, the average GPU performance is only three times faster than a 6-core CPU performance.

Sparse matrix operations are some of the fundamental problems in parallel computing. Sparse matrix operations are included in the original seven dwarfs of parallel computing identified in the Berkeley report [1]. Of these, the multiplication of two sparse matrices is particularly relevant for its myriad applications. Multiplying two sparse matrices finds several applications in varied domains such as graph algorithms [38], [26], numerical applications such as climate modeling, molecular dynamics, CFD solvers, and the like [14], [25]. It is therefore not surprising that this operation is part of several vendor supported libraries such as the Intel MKL [15], and the NVidia `cusp` [22].

Implementing sparse matrix-sparse matrix multiplication, denoted `spgemm` in the rest of this paper, on modern architectures is challenging for various reasons. Due to variations in the sparsity nature of the matrices, `spgemm` poses severe load balancing problems amongst threads. Variations in the sparsity nature also introduce irregularity in memory access patterns that are difficult to optimize. Further, it is difficult to predict the size of the output, which poses difficulties in managing the memory required for producing the output. While the above problems are applicable to generic modern architectures, using GPUs for `spgemm` poses further unique challenges. GPUs are limited in the amount of shared memory available, thereby necessitating serious workarounds. The SPMD nature of GPU thread execution means that any divergence in the execution paths of a warp of threads has a huge performance penalty.

Previous works on `spgemm` considered distributed memory systems [3] and multicore CPUs [30], apart from optimal algorithms [36]. The library implementation for `spgemm` in `cusp` has a few shortcomings (see also Section IV). Further, to the best of our knowledge there has been no previous work on designing efficient algorithms for `spgemm` on a tightly coupled hybrid platform of CPUs and GPUs[1].

In this paper, we present GPU algorithms and CPU+GPU hybrid algorithms for `spgemm` along with their efficient im-

[1]We envisage a hybrid platform as a computing platform that consists of a collection of multicore CPUs plus a few accelerators. For more details of a hybrid platform, please see Section II.g

978-1-4673-2371-0/12/$31.00 ©2012 IEEE

plementations. Our GPU algorithm achieves a speed-up of 3.5x on average compared to Intel MKL running on a quad-core CPU. It is shown in Figure 5. Our hybrid algorithm achieves an average speedup of 30% compared to a pure GPU algorithm thereby indicating the benefits of the hybrid approach. Our experience indicates that designing efficient hybrid algorithms for `spgemm` on general unstructured sparse matrices is very challenging. Therefore, we focus on a subclass of sparse matrices called *band matrices*, and show that very efficient hybrid algorithms can be designed for multiplying band matrices.

We then consider the operation of multiplying a sparse matrix with a dense matrix, denoted as `csrmm`. In this case, we provide a GPU algorithm and its corresponding efficient implementation that is 2.5x faster on average compared to the `csrmm` implementation in `cusparse`. It is shown in Figure 10.

A. Related work

Matrix-matrix multiplication is an important primitive in many areas of computer science. Considering the importance, a lot of attention has been given to it in high performance computing. Efficient solutions for dense matrices are proposed on different architectures such as GPU [34], FPGA [37], and the like.

For sparse matrix multiplication the first important work was done by Gustavson et al. [13]. They present a Row-Row fashion `spgemm` algorithm for general sparse matrices. This algorithm is being used in CSparse [5] software. Yuster el al. [36] considered `spgemm` for matrices over a ring. They presented algorithms which use fast dense matrix multiplication algorithms and are near optimal. Park et al. [24] gave space-efficient data structures and algorithms based on the proposed data structures for a class of sparse matrices which have non-zero elements adjacent to each other.

Buluc et al. [3] extensively worked on `spgemm`. They explore scalable parallel algorithms for `spgemm` on distributed memory systems. In this direction they analyse 1D and 2D algorithms and show that existing 1D algorithms are not suitable for thousands of processors. They then present 2D block distribution algorithms and data structures for hypersparse matrices. Hypersparse matrices are where the ratio of nonzeros to it dimensions is asymptotically zero.

Siegel et al. [28] designed a run-time framework for `spgemm` on heterogeneous clusters. For addressing load balancing problem they present a task based allocation model where multiplication of block of matrices represents a task.

Sulatycke et al. [30] present cache optimized algorithms on sequential machines for sparse matrix multiplication. They explore Row-Row, and Column-Row formulations of matrix multiplications.

As the architecture and the programming model of GPUs is very different from that of distributed systems, and other architectures, many of the previous algorithms and optimization strategies may not apply to `spgemm` on GPU. Also one has to note that the 2D matrix multiplication algorithms [4], [9] that are applicable in distributed systems may not be suitable for standalone systems.

Sparse matrix dense matrix multiplication is used in blocked versions of important iterative algorithms such as the Lanczos method [6] and the Conjugate gradient method [6]. Considering its importance it is provided as a primitive in the Intel MKL library [15] and the `cusparse` library [23].

B. Our Results

Recall that we are computing the product $C = A \times B$ where A is an $M \times P$ matrix and B is a $P \times N$ matrix. We explore four different approaches to perform the above product, such as multiplying the rows/columns of A with the rows/columns of B. We evaluate our implementations with respect to two datasets: a standard dataset of sparse matrices from an influential paper by Williams et al. [27], and a subset of the sparse matrices from the University of Florida SNAP sparse matrix collection [29]. A brief summary of our results is given below.

- We provide efficient algorithms and implementations for `spgemm` on GPU. Our implementation is 3.5x faster on average compared to an Intel MKL running on a quad-core CPU. Our implementation is also scalable and can handle inputs that the NVidia `cusp` library cannot handle.
- Hybrid solutions for `spgemm` using a CPU+GPU combination. Our hybrid solution is up to 30% faster compared to a pure GPU solution.
- Heuristics for a proper division of work between the CPU and the GPU. For a subclass of matrices, namely band matrices, we identify the right work division between the CPU and the GPU analytically.
- An algorithm and an efficient implementation for multiplying a sparse matrix with a dense matrix using the GPU. Our implementation is 2.5x faster on average when compared to the `csrmm` implementation in `cusparse` library.

C. Organization of the Paper

The rest of the paper is organized as follows. In Section II we discuss preliminary concepts. Section III describes our GPU algorithms for `spgemm`. Section IV discusses implementation details and results for GPU `spgemm`. Section IV also describes our hybrid approach and our hybrid algorithms for band sparse matrices. Section V describes our GPU algorithm for `csrmm` along with experimental results. Concluding remarks are presented in Section VI.

II. Preliminaries

In this section, we discuss some preliminary notions regarding matrix multiplication, sparse matrix representations and provide a brief overview of the architectures used in our experiments.

A. Matrix Multiplication Formulations

In this section we discuss the four formulations of matrix multiplication. Consider the product $C = A \times B$, where A, B, and C are matrices of size $M \times P$, $P \times N$, and $M \times N$ respectively. For a matrix A, let $A(i,:)$, and $A(:,i)$ denote the

i^{th} row and the i^{th} column of A respectively. Let $I_i(A)$ denote the indices of the nonzero elements in the i^{th} row of A.

a) The Row-Column Formulation: In the Row-Column formulation, to get one element in C, we multiply a row in the A matrix with a column in the B matrix, i.e., $C(i,j) = A(i,:) \times B(:,j)$ for $i = 1, 2, \cdots, M$ and $j = 1, 2, \cdots, N$. This is the standard matrix multiplication approach.

For a given i,j, let $I(i,j)$ denote the set of indices k such that both the elements $A(i,k)$ and $B(k,j)$ are nonzero. Then, $C(i,j) = \sum_{k \in I(i,j)} A(i,k).B(k,j)$. However, to obtain $I(i,j)$, we need to access all the elements in the i^{th} row of A and j^{th} column of B. In the worst case, we would access the entire row i of A and a column j of B whereas $I(i,j) = \Phi$. Hence, this approach is not suited for sparse matrices in general.

b) The Row-Row Formulation: In the Row-Row formulation, to compute the i^{th} row in C, $C(i,:)$, we multiply each nonzero element in $A(i,:)$ with the corresponding row in B. We then add all the scaled B rows to get the $C(i,:)$. Thus, $C(i,:) = \sum_{j \in I_i(A)} A(i,j).B(j,:)$.

c) The Column-Row Formulation: In the Column-Row formulation, for $i = 1, 2, \cdots, P$, we multiply the i^{th} column of A with the i^{th} row of B to get a matrix $C_i = A(:,i) \times B(i,:)$. The output matrix C is sum of all such matrices obtained, i.e., $C = \sum_{i=1}^{N} C_i$.

d) The Column-Column Formulation: The Column-Column formulation is similar to the Row-Row formulation. Here column elements of B are used to scale the corresponding columns of A.

In the Row-Row formulation, the Column-Row formulation, and the Column-Column formulation we access only the elements which contribute to the output. As the Column-Column and the Row-Row formulation have similar issues we investigate the Row-Row, and the Column-Row formulations on GPU.

B. Data Structures for Sparse Matrix Representations

To represent sparse matrices, one often uses special data structures such as the compressed sparse row (CSR) format, the coordinate (COO) format, the diagonal format (DIA) and the like. For details of these representations, we refer the reader to the work of Bell and Garland [2]. In the present work, we represent the input sparse matrices in the CSR format and produce output in the COO format.

C. A Brief Overview of Architectures

e) NVidia GPUs: NVidias unified architecture for its current line of GPUs supports both graphics and general computing. In general purpose computing, the GPU is viewed as a massively multi-threaded architecture containing hundreds of processing elements (cores). Each core comes with a four stage pipeline. Eight cores, also known as Symmetric Processors (SPs) are grouped in an SIMD fashion into a Symmetric Multiprocessor (SM), so that each core in an SM executes the same instruction. Each core can store a number of thread contexts. Data fetch latencies are tolerated by quickly switching between threads. The GPU also has various memory types at each level. A set of 32-bit registers is evenly divided among the threads in each SM. In Tesla C2050, the size of L1 cache in every SM is 64KB. This L1 cache can be configured as either 48 KB of user managed cache and 16 KB of hardware managed cache, or vice versa. It also features a unified L2 cache of 768kB that services all load, store, and texture read/write requests. A maximum of 48 Kilobyte of *shared memory* per SM acts as a user-managed cache and is available for all the threads in a block. The Tesla C2050 is equipped with 3 GB of off-chip *global memory* which can be accessed by all the threads in the grid, but may incur hundreds of cycles of latency for each fetch/store.

f) The Intel i7 920 CPU: The Intel i7 920 CPU that we use in our experiments is a quad-core Intel CPU. The i7 920 has four cores and with active SMT eight logical threads. Maximum clock speed of i7 920 is 2.66 GHz. The i7 920 has a three level, L1, L2, L3, cache hierarchy of sizes 64KB, 256KB, and 8MB respectively. The L3 cache is shared by all the four cores. The memory bandwidth is up to 25.6 GB/s MHz.

g) Our Hybrid Platform: Our hybrid platform is a combination of an Intel i7 920 CPU and the NVidia Tesla C2050 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU. To program the GPU we use the CUDA API Version 4.0 [20]. For programming the CPU, we use OpenMP 4.2 and ANSI C [10]. The CUDA API Version 4.0 supports asynchronous concurrent execution model so that a GPU kernel call does not block the CPU thread that issued this call. This also means that execution of CPU threads can overlap a GPU kernel execution. Asynchronous transfer of data from the CPU to the GPU is also supported through *streams*. This facility allows one to overlap not only executions on the CPU and the GPU but also data transfers between the CPU and the GPU.

III. GPU Algorithms

In this section we describe our GPU algorithms for the Row-Row and the Column-Row formulations. Recall that in $C = A \times B$ we have that A is an $M \times P$ matrix, B is a $P \times N$ matrix and C is an $M \times N$ matrix.

A. The Row-Row Formulation

Recall that in the Row-Row formulation of sparse matrix multiplication, we produce rows of the matrix C. For $i = 1, 2, \cdots, M$, we have that $C(i,:) = \sum_{j \in I_i(A)} A(i,j).B(j,:)$. For our GPU algorithm, we consider matrices A and B in CSR format and produce C in the COO format.

The basic approach of our GPU algorithm is as follows. We construct $C(i,:)$ as a row of N elements of which only a few are nonzero. We then copy only the nonzero values of $C(i,:)$ to the output. On the GPU, we launch a fixed number of warps, W. Each row of A is assigned to one warp. If $M > W$, then we iterate over the rows of A in multiples of W. Warp i computes the i^{th} row of C. For this, warp

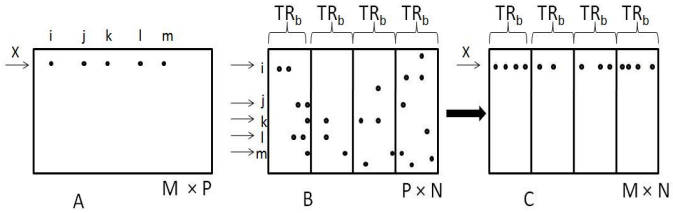

Fig. 1. Diagram illustrating our Row-Row method. i,j,k,l,m denote column indices of non-zeroes in X^{th} row of A. In each iteration a part of length TR_b of i^{th},j^{th},k^{th},l^{th},m^{th} rows in B are scaled by element present at column index i,j,k,l,m of X^{th} row of A respectively. Addition of these scaled rows of length TR_b will give a portion of TR_b size of X^{th} row of C.

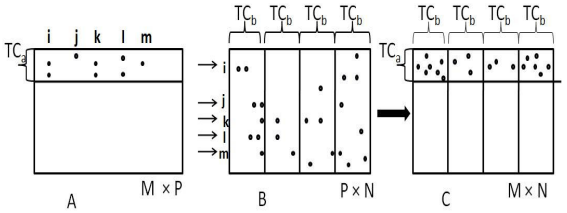

Fig. 2. Diagram illustrating our Column-Row method. Each strip of TC_a rows in A matrix is multiplied with B matrix iterating over TC_b size strip of columns. The multiplication is done in column-row fashion as described in II-A. i, j, k, l, m indicate the column indices of columns in A which have non-zero elements. These columns are multiplied with corresponding rows in B to get the output.

i accumulates the non-zero values and their indices in $C(i,:)$ using auxiliary arrays $PartialOutput$ and $NonZeroIndices$. The array $PartialOutput$ is used to accumulate the nonzero elements of $C(i,:)$. The array $NonZeroIndices$ is used to store the indices of nonzero elements in the $PartialOutput$.

From the above, we can see that the size of the array $PartialOutput$ should be N. We should create this array in the global memory of the GPU as it is not feasible to create this in the shared memory. It can be also noted that because of this reason, writes to $PartialOutput$ may be uncoalesced in nature. Even then, the size of the global memory of the GPU is not enough to store the $PartialOutput$ and the $NonZeroIndices$ arrays for all the W warps that are all active at the same time. Hence, we consider groups of TR_b columns of B in an iterative manner. In this case, the size of the auxiliary arrays $PartialOuput$ and $NonzeroIndices$ is TR_b for each warp. This is illustrated in Figure 1.

To improve efficiency, we use shared memory to store the elements of A. Given that also shared memory is limited, we iterate over the elements of a row of A in units of $PartRow$. If the number of elements in a row of A is less than the $PartRow$, we bring the elements in to the shared memory only once and use them in the next iterations over B columns.

Suppose that a warp has brought in $PartRow$ nonzero elements of $A(i,:)$ into the shared memory. For each such element, say $A(i,j)$, the warp accesses the jth row of B in groups of nonzero elements with column indices in TR_b range. Each of these nonzero elements with column index in TR_b range of the j^{th} row of B are multiplied with $A(i,j)$. The resultant elements are added to the array $PartialOutput$ at the appropriate indices. Then the non-zero indices are updated in the $NonZeroIndices$ array. As threads in the warp may simultaneously add the new nonzero indices into the $NonZeroIndices$ array, we use atomic operations.

Notice that the array $PartialOutput$ is sparse in nature. When producing the output matrix, we need to copy only the nonzero items in the $PartialOutput$ array. Each warp therefore compacts the $PartialOutput$ array using the array $NonZeroIndices$ and writes the result to $OutputBuffer$ in the GPU global memory as tuples of ⟨row index, column index, value⟩. Recall that during compaction of the $PartialOutput$ array, if we produce the output matrix in the COO format as tuples of the form ⟨row index, column index, value⟩, warps have to synchronize with each other so as to produce the correct output. This synchronization requires coordination in the global memory across warps from different blocks. Such global synchronization, though can be achieved using global atomic operations, is however very expensive on the GPU. To reduce the need of global atomic operations, in our GPU algorithm each block stores the output in global memory and the final output is accumulated in the CPU.

Since the GPU global memory is also limited and the available storage on CPU is typically large enough to hold the output matrix, our choice of storing the output in CPU helps in handling larger matrices. Otherwise, our implementation would be limited by the amount of available GPU global memory, which constrains us to handle sparse matrices of a very small limited output size only. For instance on the Tesla C2050 GPU, `spgemm` in CUSP library which stores the output in global memory of GPU cannot handle four matrices in the dataset shown in Figure 3. The GPU Row-Row Algorithm is shown as Algorithm 1.

To continue with the rest of the rows of A, we need to free up the global memory by copying the output tuples of the form ⟨row index, column index, value⟩ to the CPU storage. However, this copying can be overlapped in a double buffering fashion with the GPU executing on the rest of the rows of A. As it is difficult to determine the size of the output prior to the computation, we store the output of each iteration in a buffer at CPU, and finally combine them to produce the output in the COO format.

B. Column Row Formulation

In the Column-Row formulation of `spgemm`, recall that we accumulate all the partial output matrices obtained by multiplying a column of A with the corresponding row of B. When the i^{th} column of A consisting of n_{ai} non-zero elements is multiplied with the i^{th} row of B consisting of n_{bi} non-zero elements we get a matrix of size $n_{ai} \times n_{bi}$. It is not feasible to store all the partial output matrices and then add them to get the output matrix. Therefore, we compute output matrices in blocks of $TC_a \times TC_b$. We multiply $TC_a \times P$ part of A matrix with $P \times TC_b$ part of B to get non-zero elements in block of $TC_a \times TC_b$ matrix. See Figure 2 for an illustration.

If a portion of the rows of A fit in the registers of the GPU, then these can be reused across multiple iterations over the columns of B. This motivates us to divide the A matrix into parts of contiguous rows. To improve efficiency, we consider up to TC_a rows of A with the total number of nonzero elements not exceeding the block size. We also

Algorithm 1 Row-Row GPU Algorithm

$OutputBuffer = Buf[0]$;
$TransferBuffer = Buf[1]$;
repeat
  Assign each row in A to a warp.
  //warpid is the index of the warp in the launched kernel.
  //rowid is the row index that the warp numbered warpid operates on
  for warpid = 1 to W warps **do**
    for ℓ = 0 to N in increments of TR_b **do**
      repeat
        Load $PartRow$ elements of the rowid row of A into the shared memory.
        for Each element $A(rowid, k)$ in $PartRow$ **do**
          $j = \ell$;
          repeat
            $PartialOutput[j] += A(rowid, k) \times B(k, j)$
            $NonZeroIndices \cup= j$, atomically
            $j = j + 1$;
          until $j < \ell + TR_b$
        end for
        Write tuples ⟨rowid, column id, value⟩ to the $OutputBuffer$ using $PartialOutput$ and $NonZeroIndices$.
      until $A(rowid, :)$ is processed
    end for
  end for
  Swap pointers of $OutputBuffer$,$TransferBuffer$.
  Transfer from $TransferBuffer$ to CPU storage;
until all rows of A are processed

process rows with more than $BlockSize$ number of non-zeros in an individual block of GPU.

In a given GPU block, we assign a thread to each element of A. Each thread multiples the element with the corresponding row elements in B in iterations of TC_b column indices. Similar to the row-row formulation, we use auxiliary matrices $PartialOutput$, and $NonZeroIndices$. We use atomic operations on matrices $PartialOutput$ and $NonZeroIndices$ as threads in the block may simultaneously access these matrices. As in Row-Row method, if the output does not fit into global memory we launch kernels in iterations. In each iteration, the computed output is written to $OutputBuffer$ in global memory from which the CPU copies. A similar double buffering technique used here. The GPU Algorithm is shown as Algorithm 2.

IV. Implementation Details, Results, and Discussion

In this section we discuss implementation issues and also identify the values we should use for parameters used in our GPU algorithms. For finding the best configuration parameters of the GPU kernel, we performed several experiments.

Profiling the Row-Row method, we noticed that each thread is using about 34 registers. So it limits the block size to

Algorithm 2 Column-Row GPU Algorithm

$OutputBuffer = buf[0]$;
$TransferBuffer = buf[1]$;
repeat
  Assign each part of A matrix to a block in GPU.
  //$blockId$, $threadId$ indicate the index of the block in the launched kernel, and thread index in the block respectively.
  for $i = blockId$ to number of parts in A in increments of $TotalNumberOfBlocks$ **do**
    k' = starting index of this part of A rows
    for $j = 0$ to N in increments of TC_b **do**
      for $k = threadId$ to $\#nonzeros$ in this part of A in increments of $BlockSize$ **do**
        //Data and Indices refer to the data structures used in the CSR format representation of A
        $aitem = Data[k' + k]$;
        $acolumn = Indices[k' + k]$;
        $arow$ = row number of the element $aitem$
        $\ell = j$;
        repeat
          $bitem = B(acolumn, \ell)$
          $PartialOutput(arow, \ell) += aitem \times bitem$; atomically
          $NonzroIndices \cup= \{arow, \ell\}$; atomically
          $\ell = \ell + 1$
        until $\ell \leq j + TC_b$
      end for
    end for
  end for
  Swap pointers of $OutputBuffer$,$TransferBuffer$.
  Transfer from $TransferBuffer$ to CPU storage;
until all parts of A are processed

approximately 960. So we varied the $BlockSize$ from 128 to 960. In most of the cases assigning $BlockSize$ to 768 gives best performance. As we are allocating auxiliary memory for each warp we launch only 14 blocks (one per SM) and warps in these blocks iterate over all the rows in A. For each block to store its auxiliary data we need about $768 \cdot TR_b \times 12/32$ Bytes of storage. Since this storage has to be given for each of the 14 blocks that run simultaneously, the total space for auxiliary data is about $4000TR_b$ Bytes. The $OutputBuffer$ and the $TransferBuffer$ is each given a space of 24×10^6B per block of threads. Choosing $TR_b = 2 \times 10^5$, the overall space used for all the auxiliary data is about 1 GB.

Similar calculations were considered for the Column-Row formulation. In the Column-Row method, the BlockSize is constrained by the number of registers available. In most of the cases setting $BlockSize$ to 768 gives best performance. For the Column-Row method we kept the $Buffersize$ to 24×10^6 per block of threads. Given the limits on the available global memory, we choose $TC_a = 100$ and $TC_b = 24000$ so that the the auxiliary memory for each block is about 1 GB.

Matrix	Rows	NNZ	NNZ/Row
Dense	2,000	4,000,000	2000.0
Protein	36,417	4,344,765	119.3
FEM/Spheres	83,334	6,010,480	72.1
FEM/Cantilever	62,451	4,007,383	64.1
Wind Tunnel	217,918	11,634,424	53.3
FEM/Harbor	46,835	2,374,001	50.6
QCD	49,152	1,916,928	39.0
FEM/Ship	140,874	7,813,404	55.4
Economics	206,500	1,273,389	6.1
Epidemiology	525,825	2,100,225	3.9
FEM/Accelerator	121,192	2,624,331	21.6
Circuit	170,998	958,936	5.6
Webbase	1,000,005	3,105,536	3.1
LP	4,284	11,279,748	2632.9

Fig. 3. List of sparse matrices. Number of columns and rows are equal for all the matrices except for the matrix LP, where the number of columns is equal to $1,092,610$.

Collection	Instance	Rows	NNZ/Row
Road Networks	roadNet-CA	1,971,281	2.8
Web Graphs	web-Google	916,428	5.57
Communication networks	email-Enron	36,692	10.02
Product co-purchasing networks	amazon0312	400,727	7.98
Collaboration networks	ca-CondMat	23,133	8.08
Internet peer-to-peer networks	p2p-Gnutella	62,586	2.36
Social networks	wiki-Vote	8,297	12.49
Citation networks	cit-Patents	3,774,768	4.37
Autonomous systems graphs	as-Skitter	1,696,415	13.08

Fig. 4. List of sparse matrices from SNAP dataset

A. Results

In this section, we report the results of our experiments. The experiments were run on the systems described in Section II. In the rest of this paper, when we refer to the label **CPU**, we mean the Intel i7 920 CPU described in Section II.6 and when we use the label **GPU**, we refer to the GPU described in Section II.5.

*a) **Datasets**:* Our experiments consider two datasets for sparse matrices. The first dataset is a popular dataset for sparse matrices from the work of Williams et al. [27]. These instances are shown in Figure 3. Another dataset we use is a subset of the sparse matrices under the SNAP sparse matrix collection [29] maintained by the University of Florida. The SNAP collection contains 9 different classes of sparse matrices. We considered one instance from each class as shown in Figure 4.

*b) **Results**:* We compare the performance of Row-Row formulation on **GPU** with Column-Row formulation on **GPU**, Intel MKL routine running on the **CPU** and the `cusp` library routine running on the **GPU**. These comparisions are shown in Figure 5 for the dataset from Figure 3 and in Figure 6 for the dataset in Figure 4. We multiply each matrix with itself except for the case LP where we multiply the matrix with its transpose as it is a rectangular matrix.

We see that in most of the cases both our Row-Row, and Column-Row methods outperform the Intel MKL implementation. Our Row-Row method outperforms the Column-Row in most of the cases. Our Row-Row, and Column-Row methods perform up to 6x, and 2.5x faster respectively when compared to `spgemm` in Intel MKL. Such a behaviour can be observed in the instances from both the datasets considered.

In the Epidemiology, roadNet-CA, and p2p-Gnutella matrices where many rows have uniformly fewer non-zero elements the Row-Row method does not perform well because in our Row-Row method each warp acting on a row in A matrix brings it corresponding rows in B matrix. So for rows with fewer non-zero elements many threads are idle which degraded the performance. Unlike the Column-Row method, the Row-Row method takes advantage of coalescing accesses. Hence it performs better in instances such as FEM/Harbour, FEM/Ship, Wind Tunnel, and Protein. In the Column-Row method each threads brings its corresponding elements from row in B. After each iteration over columns in B all the threads in the block need to be synchronized. Due to this, variation in row sizes has effect on the performance of the Column-Row method. As instances Webbase, LP, Scircuit introduce load balancing problems, the column-Row method does not work well compared to the Row-Row method.

1) Comparison with cusp: We now present a comparison of our method with the NVidia `cusp` library routine for the `spgemm` kernel. The `cusp` library routine stores the entire output on the GPU itself. So, there is no need to copy output to the CPU which we do in our GPU algorithms. For this reason, for instances with a smaller output size, the library routine performs better than our GPU algorithms. Examples of such instances include the Economics and the Epidemiology instances from the dataset of Figure 3 and the roadnet-CA and p2p-Gnutella31 instances in the dataset from Figure 4. The `cusp` library approach of storing the output on the GPU works only on instances whose output size fits in the available global memory. On instances whose output size exceeds the available global memory, e.g. instances from Figures 5–6 where no bar is shown for `cusp` time/Row-Row time, our approach is clearly advantageous. On other instances, our method rarely fails to outperform the `cusp` library routine.

B. Towards an Hybrid Approach

GPU algorithms tend to outperform best known CPU implementations in most cases. However, as multicore CPUs evolve, it is not beneficial to keep the CPUs idle in the computation process. More so, in cases where the GPU performance is only a small factor away from the CPU performance, as it happens in most irregular computations. The `spgemm` computation is an example of irregular computations and hence we seek ways to simultaneously utilize the CPU and the GPU in our computation. We call this as *hybrid computing*. Such hybrid approaches are studied for other matrix operations in the works

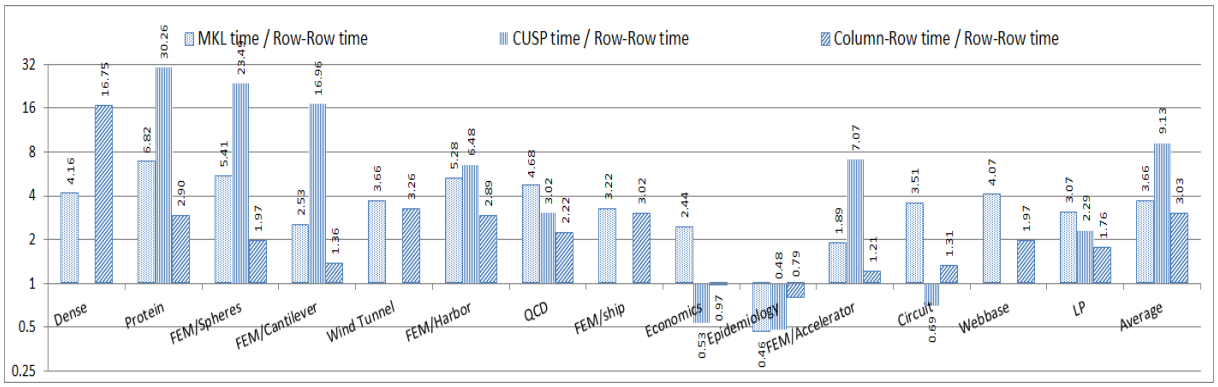

Fig. 5. Performance comparision of *MKL routine*, *cusp library routine*, and *Column-Row methods* w.r.t *Row-Row method* on the dataset shown in Figure 3. X-axis represents instances in dataset and Y- axis represents performance w.r.t *Row-Row* formulation. No bar line is shown for instances where the routine failed to work. The last instance *Average* shows the average value of the series.

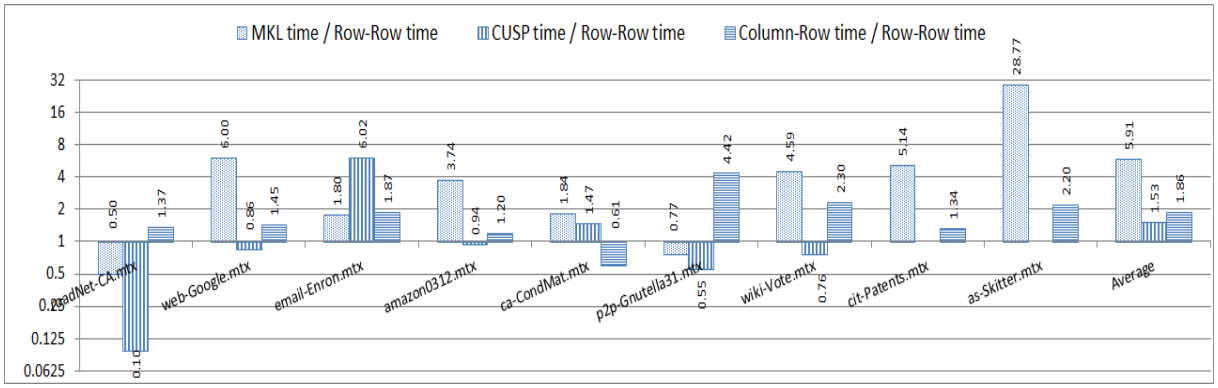

Fig. 6. Performance comparison of *MKL routine*, *cusp library routine*, and *Column-Row methods* w.r.t *Row-Row method* on the dataset shown in Figure 4. X-axis represents instances in dataset and Y- axis represents performance w.r.t *Row-Row* formulation. No bar line is shown for instances where the routine failed to work. The last instance *Average* shows the average value of the series.

of [31], [32], [19]. One of the standard approaches to design hybrid algorithms is to compute on a portion of the input on the CPU and perform the remaining part of the computation on the GPU, (cf. also [7], [8] for more examples).

To this end, we now extend our Row-Row algorithm to work as a hybrid algorithm. In our hybrid algorithm for `spgemm`,we choose a threshold $t\%$ and assign the computation corresponding to $t\%$ of the rows of A to the CPU. The remaining computation is performed on the GPU. The challenge in designing efficient hybrid algorithms then lies in finding the right threshold. A good value of t can be obtained by exhaustive experimentation. We call the corresponding time as `Best hybrid time`. The results of the best hybrid times are given in Figure 7. However, exhaustive experimentation is not an ideal solution. Hence, we start with identifying heuristics to find a good value for t. We experiment with two different heuristics.

a) Heuristic I: In our first heuristic, we find the threshold based on the number of multiplications involved in an instance of `spgemm` when using the Row-Row formulation. For a sparse matrix A, let $N_i(A)$ to denote the number of nonzero elements in the i^{th} row of A. Let $I_i(A)$ denote the indices of the nonzero elements in the i^{th} row of A. According to the Row-Row formulation, the number of multiplications for processing the i^{th} row of A in $A \times B$ is $\sum_{j \in I_i(A)} |N_j(B)|$. From Figure 5, we see that the average GPU performance on

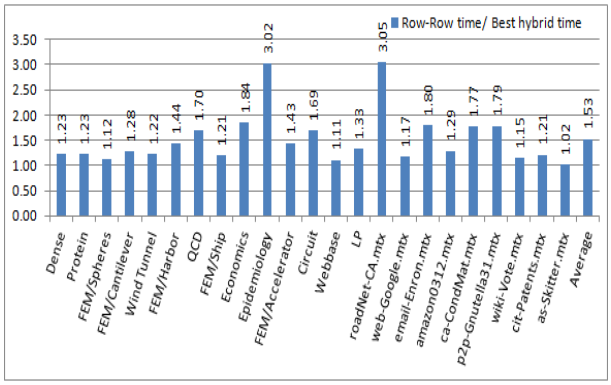

Fig. 7. Performance comparison of Hybrid method w.r.t Row-Row method on datasets shown in Figure 3, Figure 4.

the dataset from Figure 3 is around 3x. So, we set t to be 25% of the total number of multiplications. We find r which refers to the row number by which $t\%$ of the multiplications occur. We then assign rows indexed 1 to r to be processed on the CPU and rows $r+1$ to M are processed on the GPU. The results of this heuristic are presented in Figure 8.

As can be observed, the best hybrid run time using the hybrid approach outperforms the hybrid run time obtained by using the proposed heuristic. Our heuristic considers only the average speedup to arrive at a value of t and the weakness of our heuristic can be attributed to that. To remedy this situation, we propose a better heuristic that takes the run time of Intel

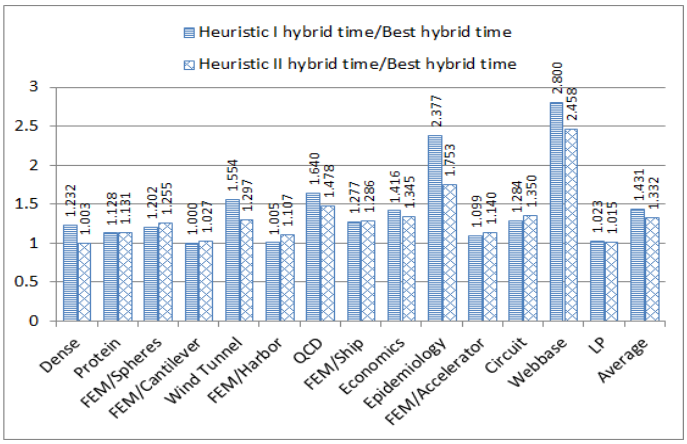

Fig. 8. Performance comparision of two presented heuristics w.r.t the best hybrid timings on the dataset shown in Figure 3. X-axis represents instances in dataset and Y-axis represents performance w.r.t best hybrid time. The last instance *Average* shows the average value of the series.

MKL and the GPU Row-Row formulation into account.

b) Heuristic II: In this heuristic, we delve a bit into each instance. We take the run time of the instance on CPU and also the GPU. Let these run times be t_c and t_g. We take the threshold t to be $\frac{t_g}{t_c+t_g}\%$. As earlier, we find a value of r so that the first r rows account for $t\%$ of the multiplication operations. The results of using this heuristic are shown in the last column of Figure 8. As can be observed, this heuristic performs better than Heuristic I in general but still cannot meet the performance of the best possible hybrid approach.

The difficulty can be partly explained by the fact that `spgemm` is a highly irregular computation. Moreover, it is difficult to estimate the number of rows that are required to make up a given percentage of the total number of operations. Knowing this, one can indeed estimate the size of the output matrix, which is one of the difficulties of the `spgemm` computation. Further, the highly unstructured sparsity nature of the matrices in the dataset from Figure 3 makes the tasks of estimating the threshold very difficult. It may therefore help if there is any prior knowledge on the nature of sparsity of the input matrices, which we explore in the coming section.

C. A Hybrid Approach for Band Matrices

Band matrices are a kind of sparse matrices where nonzero entries appear uniformly in a diagonal band. This allows one to use more efficient data structures to store band matrices and also arrive at suitable algorithms that work better than formulations such as the Row-Row and the Column-Row. We store band matrix in the diagonal format (DIA) [2]. The diagonal format consists of two arrays: for a matrix A, the $data\_A$ array stores the nonzero values, the $offset\_A$ array stores the offset of each diagonal from the main diagonal. The i^{th} column of $data\_A$ indicates i^{th} diagonal of matrix and $offset\_A[i]$ indicates the offset of i^{th} diagonal. In our implementation data is stored in column-major order so that diagonals placed adjacently from left to right. notice that multiplying two band matrices results in another band matrix.

Let A, B, and C be band matrices with $C = A \times B$. Let $Adiagonals$, $Bdiagonals$, and $Cdiagonals$ indicate number of diagonals in A, B, and C respectively. We can see that $Cdiagonals = Adiagonals + Bdiagonals - 1$. In general, multiplying the i^{th} diagonal elements of A with j^{th} diagonal elements of B contribute output to the diagonal whose offset is $offset\_A[i]$+$offset\_B[j]$.

The DIA format allows for more efficient algorithms to multiply two band matrices on the CPU and also on the GPU. The CPU Algorithm and the GPU Algorithm are presented as Algorithm 3 and Algorithm 4 respectively. Algorithm 3 iterates over the diagonals of A and the diagonals of B. For a given pair of such diagonals, all the applicable multiplications are done in parallel.

Algorithm 3 CPU Algorithm

```
for i = 1 to Adiagonals do
  for j = 1 to Bdiagonals do
    outDiagoffset = offset_A[i] + offset_B[j]
    outDiagNumber = outDiagoffset − offset_A[0] − offset_B[0]
    {writing output to the diagonal computed above}
    for k = 1 to Crows do in parallel do
      data_C(k, outDiagNumber) += data_A(k, i) × data_B(k + i + offset_A[0], j)
    end for
  end for
end for
```

In the GPU algorithm, each block of threads processes $BlockSize$ rows of the A matrix. Every block of threads brings the applicable portion of the B matrix into the shared memory. We use variables such as $Arow$ to denote the starting row number of A corresponding to the block in GPU, $startBRow$ and $endBrow$ to denote starting row value and ending row value of the applicable portion of B. Every block computes a portion of the output and writes to C. The computation is similar to that of the CPU algorithm, except for calculating the indices and offsets used.

Our experimental results on synthetically generated band matrices indicate that the CPU and the GPU algorithms presented above for band matrices outperform the corresponding CPU and GPU algorithms for `spgemm` as expected. The hybrid approach we study is similar to the hybrid approach of the Row-Row formulation where a certain $t\%$ of the rows of A are processed on the CPU and the remaining $(100-t)\%$ rows of A are processed on the GPU. We now show how to use the prior knowledge of inputs being band matrices to obtain the right threshold, t, for a hybrid `spgemm` algorithm for band matrices.

c) Heuristic for Band Matrices: To identify the correct threshold to use in the hybrid approach, we proceed as follows. Let A_r denote the number of rows in the A matrix, A_d and B_d denote the number of diagonals in the A matrix and the B matrix respectively. Let $R = A_r \cdot A_d \cdot B_d$ and $S = A_r \cdot (A_d + B_d - 1)$. It can be seen that the time taken by the

Algorithm 4 GPU Algorithm

Every $BlockSize$ rows of A is assigned to a block of GPU threads.
for each thread with index tid in the Block **do**
  $startBrow = Arow + offset\_A[0]$
  $endBrow$=$startBrow + Adiagonals - 1 + BlockSize$
  Bring rows of B from $startBrow$ to $endBrow$ into shared memory.
  for $i = 1$ to $Adiagonals$ **do**
    for $j = 1$ to $Bdiagonals$ **do**
      $outDiagoffset = offset\_A[i] + offset\_B[j]$
      $outDiagNumber = outDiagoffset - offset\_A[0] - offset\_B[0]$
      $data\_C(tid, outDiagNumber) \mathrel{+}= data\_A(tid, i) \times data\_B(tid + i + offset\_A[0], j)$
    end for
  end for
end for

CPU Algorithm (see also Algorithm 3) is proportional to R. If we process $t\%$ of the rows on the CPU, then the number of operations performed on the CPU is proportional to $\frac{t \cdot R}{100}$. Similarly, time taken by the GPU is proportional to $\frac{(100-t) \cdot R}{100}$. Let us assume that the final output would be available on the CPU by transferring the output from the GPU to the CPU in time proportional to S.

For a few input matrices, we evaluate the performance of the CPU Algorithm, the GPU Algorithm and the copy time of the output from the GPU to the CPU. This helps us identify the parameters α, β, and γ such that: $CPUtime = \alpha \times R$, $GPUtime = \beta \times R$, and $Copytime = \gamma \times S$.

In the above, the parameter α is a constant that depends on the CPU, β is a constant that depends on the GPU, and γ depends on the bandwidth of the PCI Express link connection the CPU and the GPU. The $Copytime$ in the above refers to the time taken to transfer the GPU part of the output to the CPU. If we use $t\%$ as the threshold, to minimize the hybrid execution time, we require: $CPUtime = GPUtime + Copytime$. This translates to $t \cdot \alpha R = (100 - t) \cdot (\beta R + \gamma S)$. Solving the above equation for t gives us that $t = \frac{100(\beta R + \gamma S)}{(\alpha + \beta)R + \gamma S}$.

To study our methodology we experimented on a set of synthetic matrices with varying A_r, A_d, and B_d. The synthetic dataset is generated with different combinations of A_r, A_d, and B_d sizes, so as to study the effect of varying one or more values among A_r, A_d, and B_d. The results of the study are shown in Figure 9. We observed that predicted time is nearer to best time.

V. Sparse Matrix and Dense Matrix Multiplication

In this section we discuss a GPU algorithm and implementation for multiplying a sparse matrix with a dense matrix (`csrmm`). In the matrix product $C = A \times B$, we consider the case where A is sparse and B is dense in row major format. `csrmm` is widely used in Krylov subspace methods

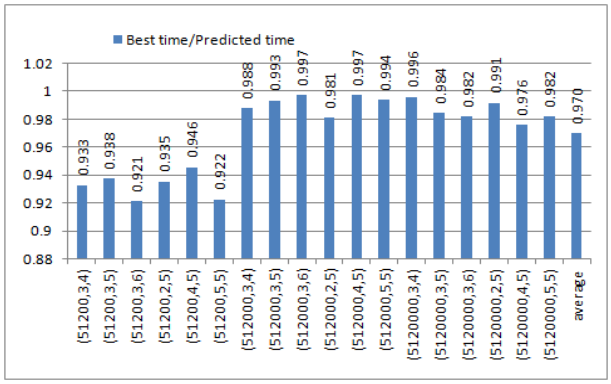

Fig. 9. Graph showing the performance comparison of best time, predicted time using formulae, for various combinations of A_r, A_d, and B_d. A tuple $\langle l, m, n \rangle$ in the x-axis indicates $\langle A_r, A_d, B_d \rangle$. The last instance *Average* shows the average value of the series.

such as the block Lanczos method [6] and the conjugate gradient method [6]. For tall and skinny dense matrices, i.e., dense matrices with very few columns, one possibility is to use `spmv` with A and each column of B. The large body of research on computing `spmv` on GPUs can be used [2], [21], [35]. However, in the case of sparse matrix-dense matrix multiplication, we can perform optimizations such as reusing the A matrix for each column in B. This suggests that one can indeed think of a separate GPU algorithm for `csrmm`.

In our algorithm, we assign a half-warp of threads to process a row in A. Each half-warp brings its elements of the A matrix and the corresponding rows from B into the shared memory. The computation is performed in the shared memory and the output is written to the global memory of the GPU. As the shared memory is limited, we iterate over B in chunks of TR_B. The algorithm is similar to Algorithm 1, with a few simplifications as the B matrix is dense.

A. Results

We experimented with assigning half-warp / warp to a row in A matrix. We see that assigning a half-warp performs better. We implement the above algorithm on our GPU and evaluate it on the sparse matrices from the dataset in Figure 3. The B matrix is chosen as follows. The number of rows in a B matrix is bound by the number of columns in the A matrix we consider. We vary the number of columns of B from 8 to 64 in multiples of 2. The results are shown in Figure 10.

We compare our results with the `cusparse` library implementation for multiplying a sparse matrix with a dense matrix. The comparison shown in Figure 10. It can be seen that we outperform the `cusparse` library implementation in most cases. We also observe that our method performs better as the number of columns in B increases. This can be attributed to the possibility that GPU memory transactions are done in sizes of 128 bytes and for smaller column sizes of B, all the 128 bytes fetched by the half-warp may not be utilized. This effect can also be seen from our results where our method performs better as the column size of B increases. Also as memory transactions are done in sizes of 128 bytes and half-warp is given to a row we expect our implementation to change

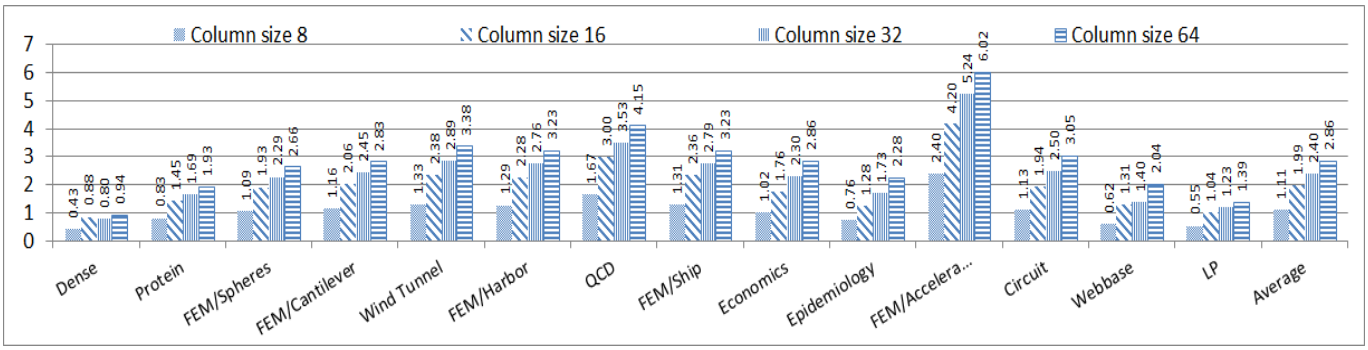

Fig. 10. Performance comparison of our `csrmm` with implementation in `CUSPARSE` on the dataset shown in Figure 3. Each bar line shows the speedup of our `csrmm` when compared to the implementation in `CUSPARSE`. The last instance *Average* shows the average value of the series.

timings for every 16 column sizes of B. We can observe this pattern in the timings for B column sizes of 8, 16, and 32.

As the arithmetic intensity is low in our computation, one can note that the computation is bound by the available bandwidth. We notice from our experiments that our computation utilizes close to the empirical peak bandwidth of 102 GB/s, reported by the CUDA SDK `bandwidthTest` benchmark [20]. Since our results from Figure 10 are in general better than the `cusparse` implementation, we can infer from [23] that our implementation outperforms Intel MKL implementations.

VI. Conclusion and Future Work

In this paper, we have studied the `spgemm` kernel extensively and provided GPU algorithms and CPU+GPU algorithms for the same. It would be interesting to seek further optimizations of the `spgemm` computation for specific classes of sparse matrices. Another direction to extend our work is to analytically obtain efficient hybrid algorithms for classes of sparse matrices.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley Technical Report No. UCB/EECS-2006-183, UC Berkeley.

[2] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proc. of SC '09.

[3] A. Buluc and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In Proc. ICPP'08, pp 503–510, 2008.

[4] L. E. Cannon. A cellular computer to implement the kalman filter algorithm. PhD thesis, Montana State University, 1969.

[5] T. A. Davis. Direct Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, 2006.

[6] J. W. Demmel, Applied Numerical Linear Algebra. Society for Industrial and Applied Mathematics, 1997.

[7] D. S. Banerjee and K. Kothapalli. Hybrid Algorithms for List Ranking and Graph Connected Components, in the Proc. of HiPC, 2011.

[8] A. Deshpande, I. Misra, P. J. Narayanan, Hybrid Implementation of Error Diffusion Dithering, in Proc. HiPC, 2011.

[9] R. A. V. D. Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. Concurrency: Practice and Experience, 9(4):255-274, 1997.

[10] Brian W. Kernighan. 1988. The C Programming Language (2nd ed.). Prentice Hall Professional Technical Reference.

[11] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: high performance sorting on the cell processor. In Proc. VLDB, pp. 1286–1297, 2007.

[12] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: Design and implementation. SIAM J. Mat. Anal. & App., 333-356, 1992.

[13] F. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. ACM T. Math. Soft.,4(3):250-269, 1978.

[14] G.H. Golub, C.F. Van Loan. Matrix Computations. 2nd ed. 1989.

[15] Intel Math Kernel Library, http://software.intel.com/en-us/articles/intel-mkl/.

[16] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. Proc. VLDB, 2008,

[17] L. Chen, Z. Hu, J. Lin, G. R. Gao. Optimizing the Fast Fourier Transform on a Multi-core Architecture, in Proc. of IEEE IPDPS 2007, pp.1-8, 26-30.

[18] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In Proc. of IPDPS '09.

[19] H. Ltaief, S. Tomov, R. Nath, and J. Dongarra, Hybrid Multicore Cholesky Factorization with Multiple GPU Accelerators, IEEE T. on Par. and Dist. Comp, 2010.

[20] NVIDIA Corporation. CUDA: Compute unified device architecture programming guide. Technical report, NVIDIA.

[21] A. Monakov, A. Avetisyan, and A. Lokhmotov. Automatically tuning sparse matrix-vector multiplication for GPU Architectures, in Proc. of HiPEAC, pp. 111–125, 2010.

[22] Nvidia cusp-library, http://code.google.com/p/cusp-library/

[23] Nvidia cusparse Library, http://developer.nvidia.com/cusparse

[24] S. C. Park, J. P. Draayer, and S.-Q. Zheng. Fast sparse matrix multiplication. Computer Physics Communications, 70:557-568, July 1992.

[25] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery. Numerical Recipes, The Art of Scientific Computing. 2nd ed., 1992.

[26] M. O. Rabin and V. V. Vazirani. Maximum matchings in general graphs through randomization. Journal of Algorithms, 10(4):557-567, 1989.

[27] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In Proc. of SC '07.

[28] Siegel, J.; Villa, O.; Krishnamoorthy, S.; Tumeo, A.; Xiaoming Li; , Efficient sparse matrix-matrix multiplication on heterogeneous high performance systems, IEEE CLSUTER, pp.1–8, 2010.

[29] Stanford Network Analysis Platform dataset , http://www.cise.ufl.edu/research/sparse/matrices/SNAP/

[30] Sulatycke, P.D.; Ghose, K.; , Caching-efficient multithreaded fast multiplication of sparse matrices, in Proc. of IPDPS, pp.117-123, 1998.

[31] S. Tomov, J. Dongarra, and M. Baboulin, Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems, Parallel Computing, Volume 36, Issues 5-6, pp:232-240, 2010.

[32] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, Dense Linear Algebra Solvers for Multicore with GPU Accelerators, Proceedings of IPDPS 2010.

[33] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In Proc. ISCA, 2010.

[34] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In Proc. SC, 2008.

[35] X. Yang, S. Parthasarathy, and P. Sadayappan, Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining, 12 pp. OSU-CISRC-2/10-TR05.

[36] R. Yuster and U. Zwick. Fast sparse matrix multiplication. ACM Trans. Algorithms, 1(1):2-13, 2005.

[37] Zhuo, L.; Prasanna, V.K.; , Scalable and modular algorithms for floating-point matrix multiplication on FPGAs, in Proc. IPDPS, 2004.

[38] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. Journal of the ACM, 49(3):289-317, 2002.