Sparse Matrix Matrix Multiplication on Hybrid CPU+GPU Platforms

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

Center for Security, Theory and Algorithmic Research (CSTAR), IIIT-Hyderabad
Gachibowli, Hyderabad, India, 500 032.
[1]kiranm@research.iiit.ac.in
[2]sivaramakrishna.i@research.iiit.ac.in [3]kkishore@iiit.ac.in

ABSTRACT

Sparse matrix-sparse/dense matrix multiplications, spgemm and csrmm, among other applications find usage in various matrix formulations of graph problems. GPU based supercomputers are presently experiencing severe performance issues on the Graph-500 benchmarks, a new HPC benchmark suite focusing on graph algorithms. 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 output size. It is even more challenging when combined with the GPU architectural constraints in memory accesses, limited shared memory, and thread execution.

To address these challenges and perform the operations efficiently on a GPU, we evaluate three possible variations of matrix multiplication (Row-Column, Column-Row, Row-Row) and perform suitable optimizations. Our experiments indicate that the Row-Row formulation outperforms the other formulations. We extend the Row-Row formulation to a CPU+GPU hybrid algorithm that simultaneously utilizes the CPU as well. In this direction, we present a heuristic to find the right amount of work division between the CPU and the GPU. For a subclass of sparse matrices, namely band matrices, we present a heuristic that can identify the right amount of work division. Our hybrid approach outperforms the corresponding pure GPU or pure CPU implementations.

Our hybrid row-row formulation of the spgemm operation performs up to 6x faster when compared to the optimized multi-core implementation in the Intel MKL library. In a similar fashion, our GPU csrmm operation performs up to 5x faster when compared to corresponding implementation in the cusparse library, which outperforms the Intel MKL library implementation.

Keywords

sparse matrix multiplication, hybrid algorithms, GPGPU, band matrices

1. 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 [11], sorting [12], and the like are available on varied architectures such as Intel CPUs [10], GPUs [12], and also the IBM Cell [6]. 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 [29], 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 parllel 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 [32, 22, 20], numerical applications climate modeling, molecular dynamics, CFD solvers, and the like [14, 8, 21]. It is therefore not surprising that this operation is part of several vendor supported libraries such as the Intel MKL [9], and the NVidia cusp [18]. The spgemm kernel is also considered as an important kernel in the class of throughput oriented applications [29].

Implementing sparse matrix-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 introduces 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 comes with a huge performance penalty.

Previous works on spgemm considered distributed memory systems [2] and multicore CPUs [26], apart from optimal algorithms [31]. However, as the architecture and the programming model of GPUs is very different from that of distributed systems, and other architectures, many of the algorithms and optimization strategies may not apply to spgemm on GPU. The library implementation for spgemm in cusp has a few shortcomings (see also Section 4).

To the best of our knowledge there has been no previous work on designing efficient algorithms for spgemm on GPU's and also on a heterogeneous collection of CPUs and GPUs.

In this paper, we present GPU algorithms and CPU+GPU algorithms for spgemm along with their efficient implementations. Let A, B, and C denote matrices of sizes $M \times P$, $P \times N$, and $M \times N$ respectively. Consider the matrix product $C = A \times B$. Our algorithms explore the space of possibilities for spgemm such as multiplying the rows/columns of matrix A with the rows/columns of B. We evaluate our implementation on two datasets: a standard dataset for sparse matrices from [23], and a representative subset of the sparse matrices included in the University of Florida SNAP sparse matrix collection [25]. Our implementation achieves a speed-up up to 6x compared to Intel MKL running on a quad-core CPU.

While GPU algorithms are interesting, better resource efficiency can be achieved by also including the CPUs in the computation process. This can be called as *hybrid computing* or *heterogeneous computing* and involves designing algorithms that run simultaneously on a heterogeneous collection of computing devices. We extend our results to arrive at a CPU+GPU hybrid algorithm for spgemm. Our hybrid algorithm combines Intel MKL with relevant GPU algorithms for spgemm. Our hybrid algorithm achieves an average speedup of 30% compared to a pure GPU algorithm thereby indicating the benefits of the hybrid approach.

1.1 Related work

Buluc et al. [2] 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.

Gustavson et al [7] developed algorithm for spgemm. They presented spgemm for CSR based format in Row-Row fashion. Yuster el al [31] considered spgemm for matrices over a ring. They presented algorithms which use fast dense matrix multiplication algorithms and are near optimal.

Siegel et al [24] 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 [26] present cache optimized algorithms on sequential machines for matrix matrix multiplication on sparse matrices. They explore Row-Row, and Column-Row formulations of matrix multiplications.

1.2 Our Results

In this paper, we first investigate efficient algorithms and implementation for spgemm . Recall that we are computing the product $C = A \times B$ where A is an $M \times P$ matrix and B is an $P \times N$ matrix. We explore four different approaches to perform the above product, such as:

- Multiplying the rows of A with the columns of B,
- Multiplying the rows of A with the rows of B,
- Multiplying the columns of A with the rows of B, and
- Multiplying the columns of A with the columns of B.

We investigate GPU algorithms and their corresponding efficient implementations for the above approaches.We evaluate our implementations with respect to two datasets: a standard dataset of sparse matrices from an influential paper by Williams et al [23], and a subset of the sparse matrices from the University of Florida SNAP sparse matrix collection [25]. Our results indicate that our implementations are up to 6x faster compared to Intel MKL running on a quad-core CPU.

With the aim of improving the efficiency, we then extend our implementations to consider hybrid approaches involving both CPUs and GPUs simultaneously in the computation. Experimental results indicate that this results in a further 30% improvement in the performance of spgemm compared to our pure GPU implementations. Our hybrid algorithms assign a portion of the overall work to CPUs based on a threshold. We also design heuristics to find the best threshold given an input instance. As this is quite difficult for general unstructured sparse matrices, we show that for a subclass of sparse matrices called *band matrices*, such a heuristic can be obtained analytically.

We then consider the operation of multiplying a sparse matrix with a dense matrix.In this case, we provide a GPU algorithm and its corresponding efficient implementation. Our implementation is up to 5x faster when compared to the csrmm implementation in cusparse running on a quad-core CPU.

A brief summary of our results is given below.

- We provide efficient algorithms and implementations for spgemm on GPU. Our implementation is up to 6x faster 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 banded 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 up to 5x faster when compared to the csrmm implementation in cusparse running on a quad-core CPU.

1.3 Organization of the Paper

The rest of the paper is organized as follows. In Section 2 we discuss preliminary concepts. Section 3 describes our GPU algorithms for spgemm. Section 4 discusses implementation details and results for GPU spgemm. Section 4 also described our hybrid approach and our hybrid algorithms for band sparse matrices. Section 5 describes our GPU algorithms for multiplying a sparse matrix with a dense matrix along with experimental results for such multiplication. Concluding remarks are presented in Section 6.

2. PRELIMINARIES

In this section, we discuss some preliminary notions. Section 2.1 describes the various matrix multiplication formulations. Section 2.2 describes the characteristics of the GPUs and the CPUs used in our experiments.

2.1 Matrix Multiplication Formulation

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 i^{th} column of A respectively.

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.

In the Row-Column formulation of `spgemm`, 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, $C(i,j) = \sum_{k \in I(i,j)} A(i,k).B(k,j)$. Thus, only elements of the ith row of A whose column indices appear in $I(i,j)$ and elements of the j^{th} column of B whose indices appear in $I(i,j)$ contribute to $C(i,j)$. However, to obtain $I(i,j)$, we need to bring from the memory all the elements in the i^{th} row of A and j^{th} column of B. Therefore, we bring in elements which may not contribute to the output. In the worst case, we would bring in the entire a 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.

The Row-Row Formulation.

In the Row-Row formulation, to compute the i^{th} row in C, $C(i,:)$, we multiply each element in $A(i,:)$ with corresponding row in B. We then add all the scaled B rows to get the $C(i,:)$. Thus, $C(i,:) = \sum_{j \in A(i,:)} A(i,j).B(j,:)$. In this formulation, we access only the elements which contribute to the output. For the example matrix shown in Figure 1 [a], the working of the Row-Row formulation is shown in Figure 1 [b].

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., $C = \sum_{i=1}^{N} C_i$. In this formulation, we access only the elements which contribute to the output. For an example matrix shown in Figure 1 [a], the working of the Column-Row formulation is shown in Figure 1 [c].

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. An example is shown in Figure 1 [d] for the matrices in Figure 1 [a].

In the Column-Row formulation, the Row-Row formulation, and the Column-Column formulation we bring only the elements which contributes to the output. So these three formulations have have optimal number of operations. As the Column-Column, and the Row-Row formulation have similar issues we investigate the Row-Row, and the Column-Row formulations on GPU.

2.1.1 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 seminal work of Bell and Garland [13]. In this work, we represent the input sparse matrices in the CSR format and produce output in the COO format.

(a) **Example matrices**

$$A = \begin{bmatrix} 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 3 & 4 \\ 8 & 0 & 0 \\ 0 & 0 & 6 \\ 0 & 7 & 0 \end{bmatrix}$$

(b) **Row - Row method**

$C(1,:) = 2 \times \begin{bmatrix} 8 & 0 & 0 \end{bmatrix} + 1 \times \begin{bmatrix} 0 & 0 & 6 \end{bmatrix} = \begin{bmatrix} 16 & 0 & 6 \end{bmatrix}$

$C(2,:) = 1 \times \begin{bmatrix} 0 & 7 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 7 & 0 \end{bmatrix}$

$C(3,:) = 1 \times \begin{bmatrix} 2 & 3 & 4 \end{bmatrix} + 1 \times \begin{bmatrix} 0 & 0 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 10 \end{bmatrix}$

$C(4,:) = 2 \times \begin{bmatrix} 2 & 3 & 4 \end{bmatrix} + 4 \times \begin{bmatrix} 0 & 7 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 34 & 8 \end{bmatrix}$

$C = (C(1,:); C(2,:); C(3,:); C(4,:)) = \begin{bmatrix} 16 & 0 & 6 \\ 0 & 7 & 0 \\ 2 & 3 & 10 \\ 4 & 34 & 8 \end{bmatrix}$

(c) **Column - Row method**

Let C_i denote the matrix obtained by multiplication of i^{th} column of A and i^{th} row of B.

$$C_1 = \begin{bmatrix} 0 & 0 & 1 & 2 \end{bmatrix}^T \times \begin{bmatrix} 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 2 & 4 \\ 0 & 0 & 3 & 6 \\ 0 & 0 & 4 & 8 \end{bmatrix}^T$$

$$C_2 = \begin{bmatrix} 2 & 0 & 0 & 0 \end{bmatrix}^T \times \begin{bmatrix} 8 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 16 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}^T$$

$$C_3 = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}^T \times \begin{bmatrix} 0 & 0 & 6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 6 & 0 & 6 & 0 \end{bmatrix}^T$$

$$C_4 = \begin{bmatrix} 0 & 1 & 0 & 4 \end{bmatrix}^T \times \begin{bmatrix} 0 & 7 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 28 \\ 0 & 0 & 0 & 0 \end{bmatrix}^T$$

$$C = \sum_{i=1}^{4} C_i = \begin{bmatrix} 16 & 0 & 2 & 4 \\ 0 & 7 & 3 & 34 \\ 6 & 0 & 10 & 8 \end{bmatrix}^T$$

(d) **Column - Column method**

$C(:,1) = 2 \times \begin{bmatrix} 0 & 0 & 1 & 2 \end{bmatrix}^T + 8 \times \begin{bmatrix} 2 & 0 & 0 & 0 \end{bmatrix}^T = \begin{bmatrix} 16 & 0 & 2 & 4 \end{bmatrix}^T$

$C(:,2) = 3 \times \begin{bmatrix} 0 & 0 & 1 & 2 \end{bmatrix}^T + 7 \times \begin{bmatrix} 0 & 1 & 0 & 4 \end{bmatrix}^T = \begin{bmatrix} 0 & 7 & 3 & 34 \end{bmatrix}^T$

$C(:,3) = 4 \times \begin{bmatrix} 0 & 0 & 1 & 2 \end{bmatrix}^T + 6 \times \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}^T = \begin{bmatrix} 6 & 0 & 10 & 8 \end{bmatrix}^T$

$C = (C(:,1); C(:,2); C(:,3)) = \begin{bmatrix} 16 & 0 & 6 \\ 0 & 7 & 0 \\ 2 & 3 & 10 \\ 4 & 34 & 8 \end{bmatrix}$

Figure 1: Examples of various matrix matrix multiplication formulations.

2.2 A Brief Overview of NVidia GPUs and the Intel i7 920 CPU

NVidia GPUs.

NVidia's unified architecture (see also the left half of Figure 2) 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*). Thirty two cores, also known as *Symmetric Processors* (SPs) are grouped in an SIMD fashion into a *Symmetric Multiprocessor* (SM). The Tesla C2050 has 14 such SMs, which makes for a total of 448 processing cores. Each core can store a number of thread contexts. Data fetch latencies are tolerated by switching between thread contexts. Nvidia features a zero-overhead scheduling system by quick switching of thread contexts in the hardware.

The CUDA API allows a user to create a large number of threads to execute code on the GPU. Threads are also grouped into *blocks* and blocks make up a *grid*. Blocks are serially assigned for execution on each SM. The blocks themselves are divided into SIMD groups called *warps*, each containing 32 threads on current hardware.which enables warps that are stalled on a memory fetch to be swapped for another warp.

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. The 64 KB L1 cache per SM can be configured as either 48 KB of user managed cache and 16 KB of hardware managed cache, or 16 KB of user managed cache and 48 KB of hardware managed cache. It also features 768 KB unified L2 cache that services all load, store, and texture read/write requests. 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. Global memory can also be accessed through two read-only caches known as the *constant memory* and *texture memory* for efficient access for each thread of a warp.

Computations that are to be performed on the GPU are specified in the code as explicit *kernels*. Prior to launching a kernel, all the data required for the computation must be transferred from the *host* (CPU) memory to the GPU *global* memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data. Barrier synchronization for all the threads in a block can be defined by the user in the kernel code. Apart from this, all the threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be performed across separate kernel launches. For more details, we refer the interested reader to [16].

The Intel i7 920 CPU.

The Intel i7 920 CPU that we use in our experiments is a quad-core Intel CPU. It is shown in the right half of Figure 2. The Core i7-920 has four cores and with active SMT eight logical threads. The maximum clock speed of i7-920 is 2.66 GHz. The L3 cache has a size of 8 MB. The L1 cache size is 64 KB per core and L2 is 256 KB. Other features of the Core i7 920 include a 32 KB instruction and a 32 KB data L1 cache per core and the L3 cache is shared by all 4 cores. The memory bandwidth is up to 25.6 GB/s MHz.

Our Hybrid Platform.

Our hybrid platform as shown in Figure 2 is a coupling of the two devices described above, the Intel i7 920 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 [16]. For programming the CPU, we use OpenMP 4.2 and ANSI C [5]. The CUDA API Version 3.2 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.

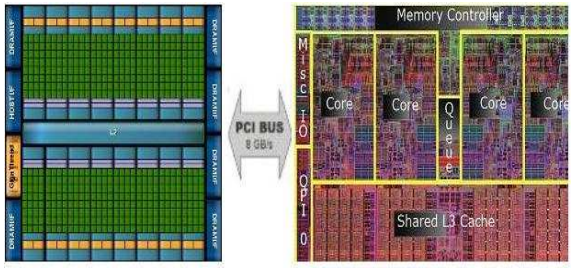

Figure 2: The GPU-CPU hybrid platform.

3. 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.

3.1 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 $C(i,:) = \sum_{j \in A(i,:)} A(i,j).B(j,:)$. 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 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 $PartialOutput$ array.

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 3.

To improve efficiency, we use shared memory to store the elements of A. Given that also shared memory is limited, we iterate

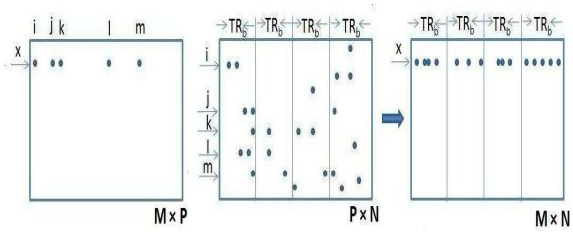

Figure 3: Diagram illustrating our Row-Row method

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 element with column index in TR_b range of the j^{th} row of B are multiplied with $A(i,j)$ and are added to the array $PartialOutput$ at the appropriate index and this index is also stored in the array $NonZeroIndices$. 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⟩. Our choice of storing the output on the CPU also helps in reducing the need of atomic operations in the GPU global memory as follows. 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.

Since the GPU global memory is also limited and the available storage on CPU is typically large enough to hold the output matrix, we choose to store the output on the CPU. 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. 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.

Algorithm 1 Row-Row GPU Algorithm

repeat
  $OutputBuffer = Buf[0]$;
  $TransferBuffer = Buf[1]$;
  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
            $PartialOutput[j] += A(rowid,k) \times B(k,j)$
            $NonZeroIndices$ U $= j$, in an atomic manner.
            $j = j + 1$;
          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
  $OutputBuffer = Buf[1]$;
  $TransferBuffer = Buf[0]$;
  Transfer from $OutputBuffer$ to CPU storage;
until all rows of A are processed

3.2 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

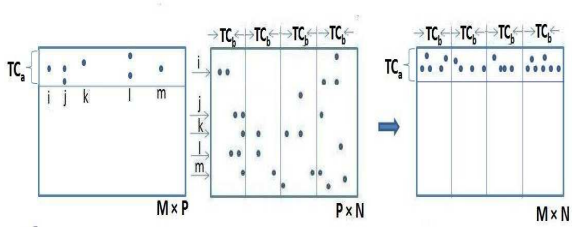

Figure 4: Diagram illustrating our Column-Row method

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

In a given GPU block, we assign each element of the rows of A considered to a thread. 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 can also be used here. The GPU Algorithm is shown as Algorithm 2.

4. IMPLEMENTATION DETAILS

In this section we discuss the 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 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 avaialble. In most of the cases setting $BlockSize$ to 768 gives best performance. For 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.

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
          $bitem = B(acolumn, \ell)$
          $PartialOutput(arow, \ell) += aitem \times bitem$; in an atomic manner
          $NonzroIndices \cup = \{arow, \ell\}$; in an atomic manner
          $\ell = \ell + 1$
        until $\ell \leq j + TC_b$
      end for
    end for
  end for
  $OutputBuffer = Buf[1]$;
  $TransferBuffer = Buf[0]$;
  Transfer from $OutputBuffer$ to CPU storage;
until all parts of A are processed

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

Figure 5: 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 $1,092,610$**.**

4.1 Results

In this section, we report the results of our experiments. The experiments were run on the following systems:

- **CPU:** An Intel Core i7 920, with 8 MB cache, 3 GB RAM and a 4.8 GT/s Quick path interface, with maximum memory bandwidth of 25 GB/s.
- **GPU:** A Tesla C2050 GPU card with 3 GB memory and 144 GB/s memory bandwidth. It is attached to an Intel Core i7 920 CPU, running CUDA Toolkit/SDK version 4.0. [16].

In the rest of this paper, when we refer to the label **CPU**, we mean the above CPU. Similarly, when we use the label **GPU**, we refer to the GPU mentioned above.

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 [23]. These instances are shown in Figure 5.

Another dataset we use is a subset of the sparse matrices under the SNAP sparse matrix collection maintained by the University of Florida. The SNAP collection contains 9 different classes of sparse matrices, and we considered one instance from each class. The instances considered are shown in Figure 6.

Results.

We first show the time taken by the Row-Row formulation and the Column-Row formulation on the GPU. We compare these timings with that of an Intel MKL routine running on the CPU mentioned in Section 2.2 and cusp library routine running on the GPU mentioned in Section 2.2. These results are shown in Figure 7 for the dataset from Figure 5 and in Figure 8 for the dataset in Figure 6. 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 further outperforms Column-Row in most of the cases. Our Row-Row, and Column-Row methods performs up to 6x, and

Instance	Multi-core time	Cusp time	GPU Row-Row time	GPU Column-Row time
Dense	5971.46	—	1436.23	24052.73
Protein	703.90	3121.88	103.16	299.29
FEM/Spheres	599.02	2602.61	110.789	218.27
FEM/Cantilever	229.67	1539.74	90.766	123.44
Wind Tunnel	488.78	—	133.624	435.01
FEM/Harbor	719.97	882.945	136.362	394.62
QCD	638.77	411.662	136.499	303.44
FEM/ship	355.30	—	110.326	333.73
Economics	198.47	42.92	81.176	78.38
Epidemiology	44.23	45.526	95.727	75.63
FEM/Accelerator	121.44	454.635	64.266	77.72
Circuit	242.89	47.697	69.159	90.86
Webbase	1025.20	—	251.825	495.93
LP	829.62	619.104	270.167	474.99

Figure 7: Results of MKL, cusp libarry routine, Row-Row, Column-Row methods. Times are in milli seconds.

Instance	Multi-core time	Cusp time	GPU Row-Row time	GPU Column-Row time
roadNet-CA.mtx	127.3	24.557	254.62	349.03
web-Google.mtx	2500.19	356.07	416.43	605.41
email-Enron.mtx	86.72	290.644	48.29	90.42
amazon0312.mtx	638.89	160.861	170.70	204.25
ca-CondMat.mtx	32.89	26.264	17.90	10.85
p2p-Gnutella31.mtx	11.49	8.22	14.88	65.814
wiki-Vote.mtx	172.34	28.50	37.58	86.59
cit-Patents.mtx	8744.38	—	1700.74	2280
as-Skitter.mtx	43012.95	—	1495	3283.43

Figure 8: Results of MKL, cusp libarary routine, Row-Row, Column-Row methods for SNAP dataset matrices. Times are in milli seconds.

Collection	Instance	Rows	NNZ	NNZ/Row
Road Networks	roadNet-CA	1,971,281	5,533,214	2.8
Web Graphs	web-Google	916,428	5,105,039	5.57
Communication networks	email-Enron	36,692	367,662	10.02
Product co-purchasing networks	amazon0312	400,727	3,200,440	7.98
Collaboration networks	ca-CondMat	23,133	186,936	8.08
Internet peer-to-peer networks	p2p-Gnutella	62,586	147,892	2.36
Social networks	wiki-Vote	8,297	103,689	12.49
Citation networks	cit-Patents	3,774,768	16,518,948	4.37
Autonomous systems graphs	as-Skitter	1,696,415	22,190,596	13.08

Figure 6: List of sparse matrices from SNAP dataset

2.5x 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.

4.1.1 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 fewer 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 5 and the roadnet-CA and p2p-Gnutella31 instances in the dataset from Figure 6. The cusp library approach of storing the output on the GPU works only on instances whose output size (minus the storage required for storing the input and the auxiliary data structures) is less than the available global memory. On instances whose output size exceeds the available gloabl memory, e.g. instances from Figures 7–8 where the cusp time is marked –, our approach is clearly advantageous. On other instances, our method rarely fails to outperform the cusp library routine.

4.2 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. Since `spgemm` is an example of irregular computations, this phenomenon can be observed also from our experimental results shown in Figure 7. 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 of [27, 28, 15], list ranking and graph connected components [3], image dithering [4], to name a few. 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. [3, 4] for 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. We experiment with two different heuristics to obtain the threshold in our case.

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 is $\sum_{j \in I_i(A)} |N_j(B)|$. From Figure 7, we see that the average GPU performance on the dataset from Figure 5 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 9.

In the result shown in Figure 9, the column labelled 'Best hybrid time' refers to the hybrid runtime at the best possible threshold. This is obtained by experimentation using various thresholds. 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 MKL and the GPU Row-Row formulation into account.

Heuristic II:.

Instance	Heuristic I hybrid time	Heuristic II hybrid time	Best hybrid time
Dense	1436.41	1168.94	1165.56
Protein	94.93	95.2	84.15
FEM/Spheres	118.921	124.15	98.94
FEM/Cantilever	70.032	71.87	70.66
Wind Tunnel	169.871	141.78	109.335
FEM/Harbor	95.06	104.711	94.622
QCD	131.54	118.5	80.20
FEM/Ship	116.554	117.39	91.257
Economics	62.373	59.23	44.037
Epidemiology	75.414	55.63	31.729
FEM/Accelerator	49.24	51.09	44.797
Circuit	52.451	55.15	40.8643
Webbase	635.482	557.85	226.984
LP	207.216	205.688	202.594

Figure 9: Results using the proposed heuristic.Times are in milli seconds

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 9. As can be observed, this heuristic performs better than Heuristic I 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 5 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.

4.3 A Hybrid Approach for Band Matrices

Band matrices are a kind of sparse matrices whose nonzero entries are present 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)[13]. 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. Figure 10 illustrates the DIA representation of an example matrix with four diagonals. In our implementation data is stored in column-major order so that diagonals placed adjacently from left to right. Entries with the symbol * are stored with 0. It turns out 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 of i^{th} diagonal elements of A with j^{th} diagonal elements of B contribute output to the diagonal whose offset is $offset\_A[i]+offset\_B[j]$.

(a) **Example matrix**

$$A = \begin{bmatrix} 1 & 6 & 0 & 0 & 0 \\ 8 & 2 & 9 & 0 & 0 \\ 2 & 6 & 3 & 8 & 0 \\ 0 & 1 & 5 & 4 & 7 \\ 0 & 0 & 3 & 2 & 5 \end{bmatrix} \quad data\_A = \begin{bmatrix} * & * & 1 & 6 \\ * & 8 & 2 & 9 \\ 2 & 6 & 3 & 8 \\ 1 & 5 & 4 & 7 \\ 3 & 2 & 5 & * \end{bmatrix}$$

$$offset\_A = \begin{bmatrix} -2 & -1 & 0 & 1 \end{bmatrix}$$

Figure 10: DIA format representation

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 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 diagonal computed above}
    for $k = 1$ to $Crows$ do in parallel **do**
      $data\_C(k, outDiagNumber) += data\_A(k, i) \times data\_B(k + i + offset\_A[0], j)$
    end for
  end for
end for

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.

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 $endBrow$ 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 for a hybrid `spgemm` algorithm for band matrices.

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 CPU 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

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**
      $outDiagoffset = offset\_A[i] + offset\_B[j]$
      $outDiagNumber = outDiagoffset - offset\_A[0] - offset\_B[0]$
      $data\_C(tid, outDiagNumber) += data\_A(tid, i) \times data\_B(tid + i + offset\_A[0], j)$
    end for
  end for
end for

to $\frac{t\dot{R}}{100}$. Similarly, time taken by the GPU on a $\frac{(100-t)\dot{R}}{100}$. Let us assume that the final output would be available on the CPU. This requires transferring the output computed by the GPU to the CPU. This requires 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. These evaluations help us identify the parameters α, β, and γ such that:

$$\begin{aligned} CPUtime &= \alpha \times R \\ GPUtime &= \beta \times R \\ Copytime &= \gamma \times S \end{aligned}$$

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 PCIExpress 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 set of matrices with varying A_r, A_d, and B_d. The results of the study are shown in Figure 11. It can be observed that we are able to calculate the split percentage with an accuracy of 96% percentage.

5. SPARSE MATRIX AND DENSE MATRIX

In this section we discuss a GPU algorithm and implementation for multiplying sparse matrix with a dense matrix (`csrmm`). `csrmm` is used in widely used Krylov subspace methods such as the block Lanczos method and the conjugate gradient method. In the matrix product $C = A \times B$, we consider the case where A is sparse and B is dense. For tall and skinny dense matrices, i.e., dense matrices with very few columns, a possibility is to use `spmv` with A and each column of B. The large body of research on computing `spmv` on GPUs can be made use of [13, 17, 30]. However,

A_r	A_d	B_d	CPU Time	GPU Time + Copy Time	Split %
51200	3	4	0.56	0.67	32
51200	3	5	0.66	0.8	31
51200	3	6	0.75	0.89	30
51200	2	5	0.53	0.62	36
51200	4	5	0.8	0.92	28
51200	5	5	0.89	1.03	26
512000	3	4	4.82	4.94	32
512000	3	5	5.76	5.86	31
512000	3	6	6.74	6.71	30
512000	2	5	4.51	4.7	36
512000	4	5	6.96	6.93	28
512000	5	5	8.06	7.96	26
5120000	3	4	47.4	47.11	32
5120000	3	5	57.36	55.61	31
5120000	3	6	66.56	64.58	30
5120000	2	5	44.43	43.86	36
5120000	4	5	69.23	66.47	28
5120000	5	5	80.14	77.46	26

Figure 11: Results of hybrid Band matrix multiplication. All times are in milli seconds

in the case of sparse matrix-dense matrix multiplication, we can perform optimizations such as reusing the A matrix for each column in B, and so on. These optimizations suggest that one can indeed think of a separate GPU algorithm for mutiplying a sparse matrix with a dense matrix.

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 into the shared memory. Each half-warp brings 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 resulting from the fact that the B matrix is dense. Unlike in Algorithm 1, we need not maintain auxiliary arrays to collect partial outputs. We can also expect the size of the output dense matrix.

5.1 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 5. 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 of this experiment are shown in Figure 12.

From Figure 12, it can be observed that we outperform a repeated `spmv` time by a large factor. We compare our results with the cusparse library implementation for multiplying a sparse matrix with a dense matrix. 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 utilised. 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 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.

In our algorithm as the arithmetic intensity is low we are bound by the available bandwidth. The emperical peak bandwidth reported by the CUDA SDK `bandwidhtTest` benchmark is 102GB/s. We see that for some of the matrices we are able to achieve the emperical peak bandwidth.

From [19], we note that the cusparse library routine for the `csrmm` kernel outperforms the Intel MKL implementation. Since our results from Table 12 are in general better than the cusparse implementation, we can infer that our implementation would also outperform the Intel MKL implementation.

6. 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. Our algorithms outperform comparable implementations by a factor of up to 6x. Some of the interesting aspects that one can extend our work are as follows. Our `spgemm` kernel is targeted at general sparse matrices. Knowing the sparsity nature of the input can sometimes help. So, it would be interesting to study efficient algorithms for `spgemm` for a specific class of sparse matrices such as block structured matrices or power-law based sparsity patterns. Our work on `spgemm` for band matrices is positive evidence that such an approach can lead to interesting algorithmic optimizations. Our heuristics to find the right threshold to use in a CPU+GPU hybrid `spgemm` kernel fails to work well for all matrices considered. This is in general difficult given the nature of the computation. Hence, it would be interesting to see how to arrive at analytical arguments for classes of sparse matrices.

7. REFERENCES

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley,

[2] Aydin Buluc and John R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In The 37th International Conference on Parallel Processing (ICPP'08), pages 503-510, Portland, Oregon, USA, 2008.

[3] Dip Sankar Banerjee and Kishore Kothapalli. Hybrid Algorithms for List Ranking and Graph Connected Components, in the Proc. of 18th Annual International Conference on High Performance Computing (HiPC), Bangalore, India, 2011.

[4] Aditya Deshpande, Ishan Misra, P. J. Narayanan, Hybrid Implementation of Error Diffusion Dithering, n the Proc. of 18th Annual International Conference on High Performance Computing (HiPC), Bangalore, India, 2011.

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

[6] Bugra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. 2007. CellSort: high performance sorting on the cell processor. In Proceedings of the 33rd international conference on Very large data bases (VLDB '07). VLDB Endowment 1286-1297.

[7] F. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. ACM Transactions on Mathematical Software (TOMS), 4(3):250-269, 1978.

[8] G.H. Golub, C.F. Van Loan. Matrix Computations. 2nd ed. (Johns Hopkins Univ. Press, Baltimore, 1989).

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

[10] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. Proc. VLDB Endow. 1, 2, 2008, 1313-1324.

[11] Long Chen; Ziang Hu; Junmin Lin; Gao, G.R. Optimizing the Fast Fourier Transform on a Multi-core Architecture, in Proc. of IEEE IPDPS 2007, pp.1-8, 26-30.

[12] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In Proc. of IPDPS '09.

[13] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proc. of SC '09, ACM, New York, NY, USA, , Article 18 , 11 pages.

[14] J.K. Cullum, R.A. Willoughby. Lanczos, "Algorithms for Large Symmetric Eignenvalue Computations" Vol. 1, Birkhauser, 1985.

[15] H. Ltaief, S. Tomov, R. Nath, and J. Dongarra, Hybrid Multicore Cholesky Factorization with Multiple GPU Accelerators, IEEE Transaction on Parallel and Distributed Computing, 2010.

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

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

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

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

[20] G. Penn. EfiňĄcient transitive closure of sparse matrices over closed semirings. Theoretical Computer Science, 354(1):72-81, 2006.

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

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

[23] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In Proc. of SC '07. ACM, New York, NY, USA, , Article 38 , 12 pages.

[24] Siegel, J.; Villa, O.; Krishnamoorthy, S.; Tumeo, A.; Xiaoming Li; , "Efficient sparse matrix-matrix multiplication on heterogeneous high performance systems," Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on , vol., no., pp.1-8, 20-24 Sept. 2010.

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

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

[27] Stanimire Tomov, Jack Dongarra, and Marc Baboulin,

Column size →	Column size 8		Column size 16		Column size 32		Column size 64	
Matrix ↓	Our GPU	cusparse	Our GPU	cusparse	Our GPU	cusparse	Our GPU	cusparse
Dense	3.45	1.49	3.45	3.03	5.61	4.46	9.59	9.02
Protein	3.98	3.3	3.98	5.78	6.15	10.4	10.36	20.01
FEM/Spheres	5.89	6.42	5.94	11.48	9.04	20.73	15.1	40.14
FEM/Cantilever	3.96	4.61	3.98	8.21	6.08	14.92	10.18	28.85
Wind Tunnel	11.31	15.01	11.44	27.24	17.26	49.81	28.69	96.95
FEM/Harbor	2.58	3.33	2.58	5.89	3.89	10.75	6.41	20.72
QCD	2.02	3.38	2.02	6.05	3.09	10.9	5.1	21.15
FEM/Ship	7.53	9.9	7.54	17.81	11.67	32.52	19.52	63.04
Economics	3.04	3.1	3.05	5.37	4.23	9.74	6.51	18.65
Epidemiology	4.61	3.51	4.59	5.88	6.22	10.78	8.98	20.44
FEM/Accelerator	3.22	7.72	3.28	13.79	4.89	25.61	8.28	49.88
Circuit	2.29	2.58	2.31	4.47	3.21	8.04	5.05	15.41
Webbase	12.49	7.77	12.62	16.54	17.23	24.07	24.13	49.11
LP	33.39	18.43	36.54	37.9	55.83	68.94	99.42	138.47

Figure 12: Results comparing our Row-Row method and implementation in CUSPARSE for various column sizes of B. All times are in milliseconds.

Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems, Parallel Computing, Volume 36, Issues 5-6, pp:232-240, 2010.

[28] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra, Dense Linear Algebra Solvers for Multicore with GPU Accelerators, Proceedings of IPDPS 2010.

[29] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10). ACM, New York, NY, USA, 451-460.

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

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

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