2015 IEEE International Parallel and Distributed Processing Symposium Workshop

A Novel Heterogeneous Algorithm for Multiplying Scale-Free Sparse Matrices

Kiran Raj Ramamoorthy[1], Dip Sankar Banerjee[2], Kannan Srinathan[3] and, Kishore Kothapalli[4]
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India, 500 032.
[1]kiran.raj@research.iiit.ac.in, [2]dipsankar.banerjee@research.iiit.ac.in,
[3]srinathan@iiit.ac.in, [4]kkishore@iiit.ac.in

***Abstract*—Multiplying two sparse matrices, denoted `spmm`, is a fundamental operation in linear algebra with several applications. Hence, efficient and scalable implementation of `spmm` has been a topic of immense research. Recent efforts are aimed at implementations on GPUs, multicore architectures, FPGAs, and such emerging computational platforms. Owing to the highly irregular nature of `spmm`, it is observed that GPUs and CPUs can offer comparable performance (Lee et al. [12]).**

In this paper, we study CPU+GPU heterogeneous algorithms for `spmm` where the matrices exhibit a scale-free nature. Focusing on such matrices, we propose an algorithm that multiplies two sparse matrices exhibiting scale-free nature on a CPU+GPU heterogeneous platform. Our experiments on a wide variety of real-world matrices from standard datasets show an average of 25% improvement over the best possible algorithm on a CPU+GPU heterogeneous platform. We show that our approach is both architecture-aware, and workload-aware.

I. Introduction

Sparse matrix operations are listed as one the seven dwarfs in parallel computing as identified in the Berkeley report [2]. Of these, multiplying two sparse matrices, usually denoted `spmm`, is one of the important problems for its numerous applications. Examples include numerical applications such as climate modeling, molecular dynamics, CFD solvers, and the like [8]. In fact, most current multi-, and many-core architectures include optimized library support for `spmm` such as `cusparse` [15] on NVidia GPUs and MKL [11] for Intel CPUs.

Given the importance of `spmm`, there has been a significant number of works targeted at efficient algorithms and their implementations on a variety of architectures. Prominent examples include the work of Buluc [5] that studied `spmm` on multicore architectures.

There are two noticeable trends that are affecting parallel computing research in the recent years. Firstly, the current architectural trend is towards a heterogeneous collection of devices involving CPUs and accelerators such as GPUs and Intel Xeon-Phi. Hence, the design and development of heterogeneous algorithms aimed at such commodity heterogeneous computing platforms are of immense research interest. Heterogeneous algorithms for a variety of problems from domains such as sorting [4], graph algorithms [6], sparse matrix computations [13], are reported in recent literature.

A second trend that is being witnessed recently is to customize algorithms and/or their implementations according to the characteristics of the input. Such studies are gaining research attention in recent times for problems such as finding the strongly connected components of real-world graphs by Hong et al. [9], mapping graph traversals to a CPU+GPU heterogeneous platform by Gharibieh et al. [6], sparse matrix-vector multiplication of scale-free matrices by Indarapu et al. [10], and the like.

In the context of sparse matrices, it can be observed that several sparse matrices arising in practical scenarios exhibit a scale-free nature. A matrix exhibiting a scale-free nature has several rows with very few nonzero elements and very few rows with a large number of nonzero elements. An example is shown in Figure 1 for the matrix webbase-1M from the collection of sparse matrices in [18]. In Figure 1, the X-axis is the number of nonzeros (NNZ) and the Y-axis is the number of rows. As can be noted from Figure 1, of the 1,000,005 rows in this matrix, there are very few rows with at least 60 nonzero elements per row, and the large number of rows have less than than 60 nonzeros. (Note that the Y-axis of Figure 1 is on a logarithmic scale.) This distribution of nonzeros across rows can have significant implications for algorithm design and implementation as we will see in this work.

Indeed, the `spmm` computation offers a lot of data parallelism that can be exploited in a heterogeneous setting too. Different elements, or rows/columns, of the output matrix can be computed in parallel independently on all the devices in the computing platform. However, in the matrix product $C = A \times B$, the amount of computation required with respect to an element $C[i, j]$ in the product matrix depends on the number of indices of the i^{th} row of A containing nonzero elements that overlap with the indices of the j^{th} column of B containing nonzero elements. In a given sparse matrix, as the number of nonzeros per row can vary significantly across rows, it is difficult to know the amount of computation required for producing a row/column of the product matrix. The above difficulty can persist even for matrices that exhibit a scale-free nature in their row sizes. Thus, even when one restricts to special classes of sparse matrices, it is challenging to design efficient heterogeneous algorithms for sparse matrix multiplication.

In this paper, we aim at efficient heterogeneous algorithms for multiplying two scale-free sparse matrices. We propose novel techniques that are aimed at alleviating two challenges:

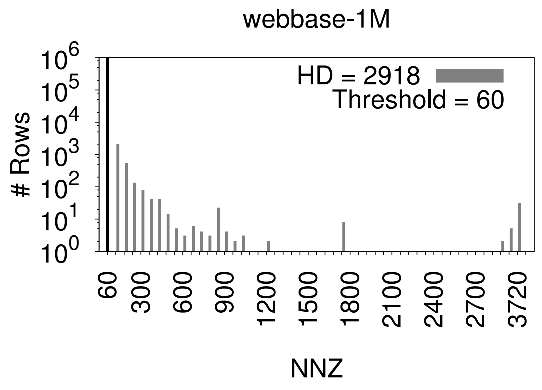

Fig. 1. Row histogram of the matrix webbase-1M from the dataset of [18]. The figure does indicate that very few rows have at least 60 nonzeros per row. The gray bars indicate these high density rows. The black bar represents the rows with low density.

(i) load balancing across the devices in the computing platform, (ii) assigning the *"right"* work to the *"right"* processor. Addressing these twin challenges, we show significant improvements in the performance compared to corresponding best known implementations on a wide variety of real-world sparse and scale-free matrices. We summarize our main technical contributions of this work as follows.

- We propose a heterogeneous algorithm, called HH-CPU, for multiplying two scale-free sparse matrices on a CPU+GPU heterogeneous platform.
- We conduct experiments on a wide range of scale-free matrices and show the efficiency of our algorithm. Our results indicate that the HH-CPU algorithm offers a 25% improved performance over the present best-known heterogeneous algorithm for multiplying two sparse matrices from [13].
- We explain the performance improvement based on the scalefree nature of the matrix.
- We also conduct experiments on synthetic scalefree matrices to understand the impact of scalefreeness on the performance of Algorithm HH-CPU.

A. Related Work

The multiplication of two matrices, dense and sparse alike, is an important primitive with applications to many areas of computing. It is therefore not surprising that a lot of research attention is devoted in this direction. Focusing on sparse matrices, one of the first notable works is that of Gustavson et al. [7]. They present a Row-Row fashion `spmm` algorithm for general sparse matrices. Park et al. [16] gave space efficient data structures and algorithms based on the proposed data structures for a class of sparse matrices which have nonzero elements adjacent to each other.

Buluc et al. [5] conducted a detailed study of `spmm` on distributed memory systems. In this direction they analyse 1D and 2D block distribution algorithms. Siegel et al. [17] designed a run-time framework for `spmm` 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. [19] present cache optimized algorithms on sequential machines for sparse matrix multiplication. They explore Row-Row and Column-Row formulations of matrix multiplications.

More recently, in [13], heterogeneous algorithms for `spmm` are designed on CPU+GPU systems. Here, the heterogeneous algorithm does not consider the nature of the matrix and is aimed at generic sparse matrices. In this work, we show that more performance gains can be obtained when one takes the nature of the input into account.

Matching workload to a computational device based on the characteristics of the workload is an emerging line of research. In [6], Gharaibeh et al. consider three graph algorithms and suggest that for large, sparse graphs, it is advisable to process vertices of low degree on the GPU and vertices of high degree on the CPU. The authors of [6] also show that such a choice can help improve the hit ratio of the last level cache on current multicore architectures. In this work, we show that such effects can be seen for `spmm` also.

B. Organization of the Paper

In the rest of the paper, we first describe some background material in Section II. Section III and Section IV discuss our algorithm and its implementation details. The results of our algorithm are presented in Section V followed by concluding remarks in Section VI.

II. Preliminaries

In this section, we discuss some preliminary notions. Section II-A describes the row-row matrix multiplication formulation. Section II-B describes the characteristics of the GPU and the CPU used in our experiments.

A. Matrix Multiplication Formulation

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 its i^{th} row and i^{th} column of respectively. The multiplication of A with B can be achieved in four different ways: multiplying the rows/columns of A with the rows/columns of B. As noted in [13], multiplying the rows of A with the columns of B, called the Row-Column Formulation, is not well suited for sparse matrices on current parallel architectures in general. In this work, we use the Row-Row Formulation that is summarized below.

a) The Row-Row Formulation: In the Row-Row formulation, to compute the i^{th} row in C, $C(i,:)$, we proceed as follows. Let $S = \{j_1, j_2, \cdots, j_r\}$ be the column indices of nonzero elements in the i^{th} row of A. For $k = 1, 2, \cdots, r$, we scale the j_k^{th} row in B by $A[i, j_k]$. We then add all the scaled B rows to get the $C(i,:)$. Thus, $C(i,:) = \sum_{k=1}^{r} A(i, j_k) \cdot B(j_k,:)$. For the example matrix shown in Figure 2 [a], the working of the Row-Row formulation is shown in Figure 2 [b].

Even when using the Row-Row formulation, estimating the work volume per row of the output matrix a-priori is still a

(a) **Example**

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

(b) **Row - Row Formulation**

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

Fig. 2. Matrix multiplication via the Row-Row Formulation.

challenging task. In fact, asymptotically, it amounts to actually performing matrix multiplication. The Row-Row formulation however can greatly help in maximizing the usage of elements that are read from memory.

b) GPU Algorithm for Row-Row based Matrix Multiplication: Consider two sparse matrices A and B. The basic approach of the GPU algorithm (from [13]) is as follows. The ith row of C, $C(i,:)$, is constructed as a row of N elements of which only a few are nonzero. The nonzero values of $C(i,:)$ are copied to the output. On the GPU, a fixed number of warps, W, are launched. Each row of A is assigned to one warp. The M rows of A are iterated over in multiples of W. Warp i computes the i^{th} row of C. For this, warp i accumulates the nonzero 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 may not be 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. We refer the reader to [13] for a detailed pseudocode.

B. A Brief Overview of our Experimental Platform

In this section, we briefly describe our heterogeneous computing platform. Our heterogeneous platform consists of an Intel i7 980 CPU and an Nvidia Tesla K20c (Kepler) GPU 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 4.1 [14]. The CUDA API Version 4.1 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.

The Tesla K20c GPU is a current generation Kepler micro-architecture from NVidia with 13 streaming multi-processors (SMX) with each having 192 cores for a total of 2496 compute cores. Each compute core is clocked at 706 MHz. Each SMX has a hardware scheduler which schedules 32 threads at a time. This group is called a warp and a half-warp is a group of 16 threads that execute in a SIMD fashion. Each of the cores of the GPU now has a fully cached memory access via an L2 cache, 1.25MB in size. In all, the K20c will provide up to 3.52 TFLOPS of single- and 1.17 TFLOPS of double-precision floating-point performance.

The Intel i7 980 is based on the Intel Westmere micro-architecture and has six cores with each core running at 3.4 GHz. With active SMT (hyper-threading), the i7 980 can handle twelve logical threads. Other features of the Core i7 980 include a 32 KB instruction and a 32 KB data L1 cache per core, a 256 KB per-core L2 cache, and an L3 cache of 12 MB that is shared by all 6 cores.

III. Multiplication of Two Sparse Scale-Free Matrices (`spmm`)

In this section, we describe our algorithm, Algorithm HH-CPU, that multiplies two sparse matrices which exhibit a scale-free behavior. Given two scale-free matrices A and B, we wish to compute their product $C = A \times B$. (Throughout, we assume that A and B are compatible for multiplication).

Our algorithm has four phases as shown in Algorithm 1. A brief description of the phases is given below followed by a detailed description in Sections III-A–III-D. In Algorithm HH-CPU, see Algorithm 1, the labels `CPU::`, `GPU::`, refer to the computations done in an overlapped manner on the CPU and the GPU respectively. The label `GPU` $\rightarrow$ `CPU`, refers to data transfer from the GPU to the CPU. The label `CPU, GPU::` refers to computations that is done on both the CPU and the GPU in an overlapped manner, where the volume of computation done on the CPU and the GPU is not known a-priori.

In Algorithm HH-CPU, as part of our preprocessing, we rearrange the rows of matrices A and B according to their size (number of nonzero elements). We call the matrix A_H (B_H) as the submatrix consisting of the rows of A (*resp.* B) that have a large number of nonzeros. The rest of the matrix, that is the submatrix of A (B) consisting of rows that contain fewer nonzeros are called as A_L (*resp.* B_L). With a reordering of the rows of A and the rows of B, the matrix A (B) can be written as $[A_H, A_L]^T$ (*resp.* $[B_H, B_L]^T$).

The matrix C can be obtained by multiplying the matrix products A_H with B_H, A_L with B_L, A_L with B_H, and A_H with B_L. These matrix products are computed in Phases II and III of Algorithm HH-CPU, followed by consolidation of the

Algorithm 1 Algorithm HH-CPU

```
1: /* Phase I */
2: CPU, GPU :: Identify thresholds $t_A$, $t_B$ and the matrices $A_H, A_L, B_H$, and $B_L$.
3: /* Phase II */
4: CPU:: Compute $A_H \times B_H$ using [13, Algorithm 1] rewritten for CPU.
5: GPU:: Compute $A_L \times B_L$ using [13, Algorithm 1].
6: /* Phase III */
7: CPU, GPU:: Compute $A_H \times B_L$ and $A_L \times B_H$.
8: /* Phase IV */
9: CPU, GPU:: Combine the results of Phases II and III.
10: GPU → CPU:: Transfer the partial results from the GPU to the CPU.
```

$$A = \begin{bmatrix} 0 & 2 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 3 & 2 & 2 & 1 \\ 0 & 0 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 2 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 3 & 2 & 2 & 1 \\ 0 & 0 & 0 & 5 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 4 & 2 & 1 \\ 0 & 1 & 0 & 0 \\ 6 & 12 & 7 & 7 \\ 0 & 0 & 0 & 25 \end{bmatrix}$$

$$\underset{A_H \times B_H}{\begin{bmatrix} 3 & 2 & 2 & 1 \\ 0 & 0 & 0 & 0 \\ 6 & 10 & 7 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}} \quad \underset{A_L \times B_L}{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 25 \end{bmatrix}} \quad \underset{A_L \times B_H}{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}} \quad \underset{A_H \times B_L}{\begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}}$$

Fig. 3. Figure explaining Algorithm HH-CPU. The matrices A and B have two rows containing at least two nonzeros in A_H and B_H and the other two rows are in A_L and B_L. The figure also shows the four partial products $A_H \times B_H$, $A_L \times B_L$, $A_H \times B_L$, and $A_L \times B_H$. The product of A and B is shown as matrix C, which is the sum of the four partial products. Rows in the four partial products that are not computed as part of our algorithm are filled with 0's for illustration.

four partial results. An example is shown in Figure 3. In the following, we explain each phase of the algorithm in detail.

A. Phase I

Phase I essentially involves identifying the threshold for matrices A and B and the matrices A_H, A_L and B_H, B_L. We do this as follows. Keeping t small may mean that the work done by the CPU in Phase II would increase, whereas keeping t large may tilt the balance towards the GPU. Hence, we chose to identify t empirically.

Once the threshold is identified, we prepare a Boolean array of size equal to the number of rows of the matrix. For a matrix A, a value of 1 at index i in this array indicates that the ith row of A is a high dense row and hence should belong to the matrix A_H. Since, computing the Boolean array is embarrassingly parallel operation we perform this computation on GPU. For this computation we need only row sizes of the rows of A and B matrices to be transferred to GPU.

B. Phase II

Since the matrices A_H and B_H have a large number of nonzero elements in their rows, we envisage that good cache blocking techniques can be used when multiplying A_H with B_H. This suggests that this product be computed on the CPU. Further, as A and B are scale-free in nature, the sizes of A_H and B_H are likely to be such that the time taken by the CPU to compute the product $A_H \times B_H$ is near-equal to the time taken by the GPU to compute the product $A_L \times B_L$. In a similar fashion, the product $A_L \times B_L$ consists of several subproblems each of which has a small computational intensity. These subproblems are also all independent of each other. Thus, these numerous subproblems can be assigned to individual threads of the GPU by grouping them into a reasonable number of thread blocks.

Therefore, in Phase II, we overlap the computation of $A_H \times B_H$ with that of computing $A_L \times B_L$. For computing $A_H \times B_H$ on the CPU, we cannot use the Intel MKL library routine as the matrices are not compatible for multiplication. For example, consider matrix A has k_1 high density rows and B has k_2 high density rows according to thresholds t_A, t_B respectively. The dimensions of A_H and B_H matrices are $k_1 \times n$ and $k_2 \times n$ respectively, which are not compatible for multiplication. Hence we have rewritten [13, Algorithm 1] to run on CPU with minor modifications to handle compatibility issue. In order to compute $A_H \times B_H$, we have the list of rows in A_H computed during phase I. For each row in A_H we multiply with the corresponding rows of B only if that row is classified as high dense. This effectively calculates $A_H \times B_H$. Our modified CPU matrix multiplication algorithm performs around 15% to 20% slower than the Intel MKL library routine. More optimization techniques could be added to improve the performance of the same which in turn will lead to significant improvement of HH-CPU algorithm.

For computing $A_L \times B_L$, we use the row-row based matrix multiplication as it is shown empirically in [13] that the row-row method can outperform the other methods of matrix multiplication on modern architectures, especially for sparse matrices. Similar to the modified CPU routine, Algorithm 1 from [13] is modified to make suitable workarounds to address issues around compatibility of matrices under multiplication.

C. Phase III

The computation in Phase III involves the multiplication of A_H with B_L and A_L with B_H. In our implementation, we make the following observations to perform these computations efficiently. Firstly, it is not clear as to which of the two products above are best suited to be executed on the CPU and the GPU respectively. Further, there are no additional input behavioral characteristics that dictate the assignment of these products to the CPU and the GPU. Secondly, the amount of time taken by the CPU and the GPU on these products cannot be estimated easily.

Given the above observations, we start computing the product of A_H with B_L in GPU. Note that, in Phase II we have already transferred matrices A and B. Hence GPU has A_H and no transfer is required. The CPU starts computing product of A_L with B_H.

To balance the work between the CPU and the GPU, we use a custom workqueue. In our custom workqueue, the CPU and GPU dequeue work-units from opposite ends of the queue.

In our case, a work-unit corresponds to the multiplication of a contiguous set of rows of A_H (A_L) with a contiguous set of rows of B_L (*resp.* B_H). We have the CPU and the GPU dequeue work-units from opposite ends of the queue so that the time taken to synchronize the dequeue operations is also minimal. Therefore, on the CPU end of the queue, we fill the queue with work-units corresponding to the product $A_L \times B_H$ and on the GPU end of the queue, we fill the queue with work-units that correspond to product $A_H \times B_L$.

We launch separate threads on the CPU to handle the CPU `spmm` operation and also launch the GPU kernel for `spmm` from within a CPU thread. We maintain two global variables $cpuOffset$ and $gpuOffset$ on the CPU side to track working units of CPU and GPU. Similarly, variables $cpuRows$ and $gpuRows$ denote the number of rows computed in that call made by CPU and GPU respectively. In our implementation, we use the modified [13, Algorithm 1] for computation on the CPU and the GPU algorithm [13, Algorithm 1] on the GPU.

D. Phase IV

In Phase IV, the outputs of the CPU and the GPU in Phases II and III are merged together and stored on the CPU. The computation in Phases II and III of Algorithm 1 produces tuples of the form $\langle r, c, v \rangle$ such that a contribution of v has to be added to the element at index (r, c) of the output matrix. For a given index (r, c) of the output matrix, there may be several tuples all of which have to be added together to get the final result. Moreover, the tuples generated in the CPU have to be combined with the tuples generated in the GPU. We can see that the tuples generated in CPU and the GPU during Phase II are for different rows. Hence merging the results of the CPU and the GPU from Phase II is straight-forward.

The steps of the algorithm we use in Phase IV are described as follows. We first merge the tuples based on r and c values. At the end of this step, we have essentially grouped all tuples based on their row and column index. The rest of the computation in this phase is to add like-tuples, that is tuples with identical row and column index. Such tuples are now in neighboring indices after the merging is done. Since we expect that there will be very few tuples for any row and column index, we process these tuples sequentially for each row and column index. However, care has to be taken so that there is exactly one thread for each such (r, c) tuple. For this purpose, we use standard techniques such as marking the indices of like-tuples, and `scan` the marked array to identify the first index for each row, column index. For each row, column index, let us call this index as the master index. We now associate a thread to each master index. The job of this thread is to add the values of the tuples with the same row and column index. Once this is done, we now produce a new output array which contains distinct row, column indices. The process is illustrated in Figure 4 for a small example.

IV. Implementation Details

In this section, we discuss a few implementation details of our algorithm during Phases II and III. The implementation of the other phases is rather straightforward from the description of the algorithm.

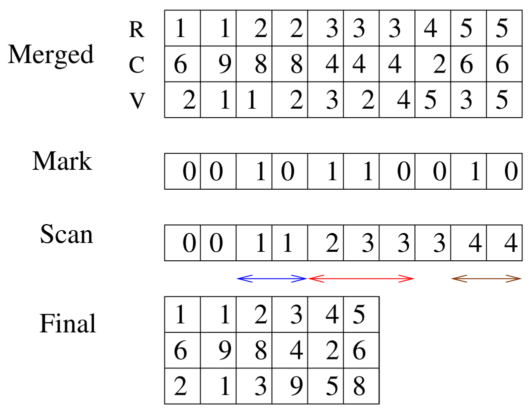

Fig. 4. Figure shows the process of merging the output of Phase II and III. The labels R, C, and V in the figure refer to the row indices, column indices and values of the nonzero elements respectively. The colored arrows in the figure indicate contiguous blocks of tuples with the same row and column index.

Matrix	Rows	NNZ	α
scircuit	170,998	958,936	3.55
Webbase-1M	1,000,005	3,105,536	2.1
cop20kA	121,192	2,624,331	143.8
web-Google	916,428	5,105,039	3.75
p2p-Gnutella31	62,586	147,892	48.9
ca-CondMat	23,133	186,936	3.58
roadNet-CA	1,971,281	5,533,214	133.80
internet	124,651	207,214	4.63
dblp2010	326,186	1,615,400	5.79
email-Enron	36,692	367,662	2.1
wiki-Vote	8,297	103,689	3.88
cit-Patents	3,774,768	16,518,948	3.90

TABLE I
List of sparse matrices used in our experiments. NNZ refers to the total number of nonzeros in the matrix. The number of columns and rows are equal for all the matrices. The column labeled α indicates the exponent of the power law distribution that the row sizes of the matrix fit to.

A. Phase II : Computing $A_L \times B_L$ on the GPU

Notice from Algorithm 1 that Phase I is performed both on the CPU and GPU. This means that before we start Phase II, the GPU needs to know the submatrices A_L and B_L of A and B respectively. In our implementation, we transfer A_L and B_L to the GPU from the CPU. Since we don't split the matrices physically, transferring A_L and B_L means transferring A and B entirely along with the Boolean array that classifies each row as high dense or low dense. Approximately, it takes around 25-30 milliseconds to transfer a matrix with around 5 Million nonzero entries to the GPU.

As mentioned earlier, we have used Algorithm 1 from [13] for this computation. Additionally, we have checked for the possibility that some of the GPU specific parameters such as the block size, the number of warps, can be varied for arriving at the best possible results. We experimented with various values for the above parameters and finally chose a block size of 1024, with 32 warps per block. Also in parallel, we compute $A_H \times B_H$ on CPU using Algorithm 1 from [13] rewritten for CPU on the sub-matrices obtained in phase I of our algorithm.

B. Phase III

Recall that in Phase III, we compute the products $A_L \times B_H$ and $A_H \times B_L$. As mentioned earlier, we also use a work queue in this phase to balance the work between the CPU and the GPU. The size of the work-unit on the CPU, variable $cpuRows$, is set at 1000 rows as this yields the best possible result. Similarly, the variable $gpuRows$ that the GPU uses to indicate the size of the work-unit is set to 10,000 rows when contributing to the product $A_L \times B_H$. According to our workqueue model, the GPU (CPU) can contribute to the product $A_L \times B_H$ (*resp.* $A_H \times B_L$) after finishing the product $A_H \times B_L$ (*resp.* $A_L \times B_H$).

V. Experimental Results and Analysis

A. Datasets

For our experiments, we use 12 matrices from standard datasets from a variety of sources listed in [18]. The matrices and their properties are shown in Table I. Figure 5 shows the row density histogram of 12 matrices from Table I. Figure 5 also shows the number of rows that are dense, that is having nonzeros more than the threshold indicated in each plot. In all our experiments we multiply the matrix with itself.

B. Results

In this section, we compare our results with the corresponding current best known results from [13], and also analyze our results. All these results are obtained on identical platforms by running the programs from [13] also on the experimental platform described in Section II-B. Each experiment is run several times. In all the plots, the label "HiPC2012" refers to the implementation of Algorithm 1 from [13]. The label "HH-CPU" refers to Algorithm 1 from Section III.

a) **Overall Improvement***:* Figure 6 shows that our algorithm achieves comparable speedup over the results of [13]. As can be noticed from Figure 6, the HH-CPU method is able to perform on average 25% faster compared to the results of [13]. Our results also outperform the results of `cusparse` and Intel MKL by 4x and 3.6x respectively.

b) **Profiling***:* We now turn our attention to understanding and interpreting the results of our experiments. In Figure 7, we show the breakup of the overall time taken by our implementation over the various phases of the algorithm. In Figure 7, the time for each phase is taken as the maximum time spent by either device on that phase. As can be noticed from Figure 7, Phases II and III dominate the overall time taken and add up to more than 96% of the overall time. This indicates that our approach has a low overhead including both Phase I and Phase IV. Hence, even if we only use the `spmm` operation for a single time on a given matrix, there are significant savings to be had when one does a brief amount of preprocessing. We could observe that the difference between the GPU and the CPU runtime within each phase is on average under 2% of the overall runtime which demonstrates a better load balancing between CPU and GPU.

c) **Analysis of the Overall Result***:* The column labeled α in Table I indicates the exponent of the power law distribution that the row sizes of the matrix fit to. This number is obtained using the toolkit developed by Alstott et al. [1]. The smaller the exponent, one can say that the underlying data exhibits a bigger degree of scalefreeness.

The relative improvement of our Algorithm HH-CPU can be understood in terms of the α value of the matrix. For instance, matrices such as webbase-1M and email-Enron have α value of 2.1. On these matrices, our algorithm has a speedup of 37% compared to that of [13]. In the next category of matrices, we have five matrices scircuit, web-Google, ca-Condmat, wiki-vote, and cit-patents with their α value between 3 and 4. Of these five, except for web-Google, the rest of the matrices show a speedup in the range of 20% to 25% over the algorithm of [13]. This speedup is a little lower compared to matrices with a lower α as a lower value of α indicates a better scalefreeness in general.

For the matrix web-Google, the speedup compared to [13] is much higher at 45%. This can be explained by the fact that the matrix has nearly a million rows with under 25 nonzeros as shown in Figure 5. This suggests that there is a better load balance in Phase II, with the CPU and the GPU getting near identical share of work. (Notice that as we use a workqueue in Phase III, the time spent by the CPU and the GPU in Phase III is near identical in most cases).

Two matrices, internet, and dblp-2010 have their α value between 4 and 5. On these two matrices, the speedup is in the range of 30%, which is higher than what is expected. These two matrices too have a structure similar to that of the matrix web-Google. So, the speedup on these matrices is higher than anticipated.

Let us now consider the matrices p2p-Gnutella31, roadNet-CA, and cop20kA with their α values at 48.9, 133.8, and 143 respectively. The high α value can also be observed from the fact that in these matrices the relative difference in the NNZ between high dense and low dense rows is small. This small difference explains their high α value. See also the row histogram plots of these matrices in Figure 5. The varying nature of improvement, ranging from just over 5% on p2p-Gnutella31 and roadNet-CA, to over 20% on cop20kA indicates that on matrices that are not scalefree, the performance benefit of our algorithm cannot be judged based on the value of α.

Finally, we note that studies on our dataset indicate that even on matrices that are not scale-free, Algorithm HH-CPU is likely to offer an advantage. The additional steps in Algorithm HH-CPU such as Phase I and Phase IV are not consuming any significant amount of time. On matrices from Table I, these two steps consume under 4% of the overall time. The actual work performed in Algorithm HH-CPU is not more than the work performed in other approaches to multiply sparse matrices in parallel. So, it can be seen that Algorithm HH-CPU does not have disadvantages compared to other approaches even on matrices that are not scale-free.

d) **Trade-off Between Phases II and III***:* Recall from Section III-A that we identify the threshold for a row to be called as a high density row by relying on the histogram of

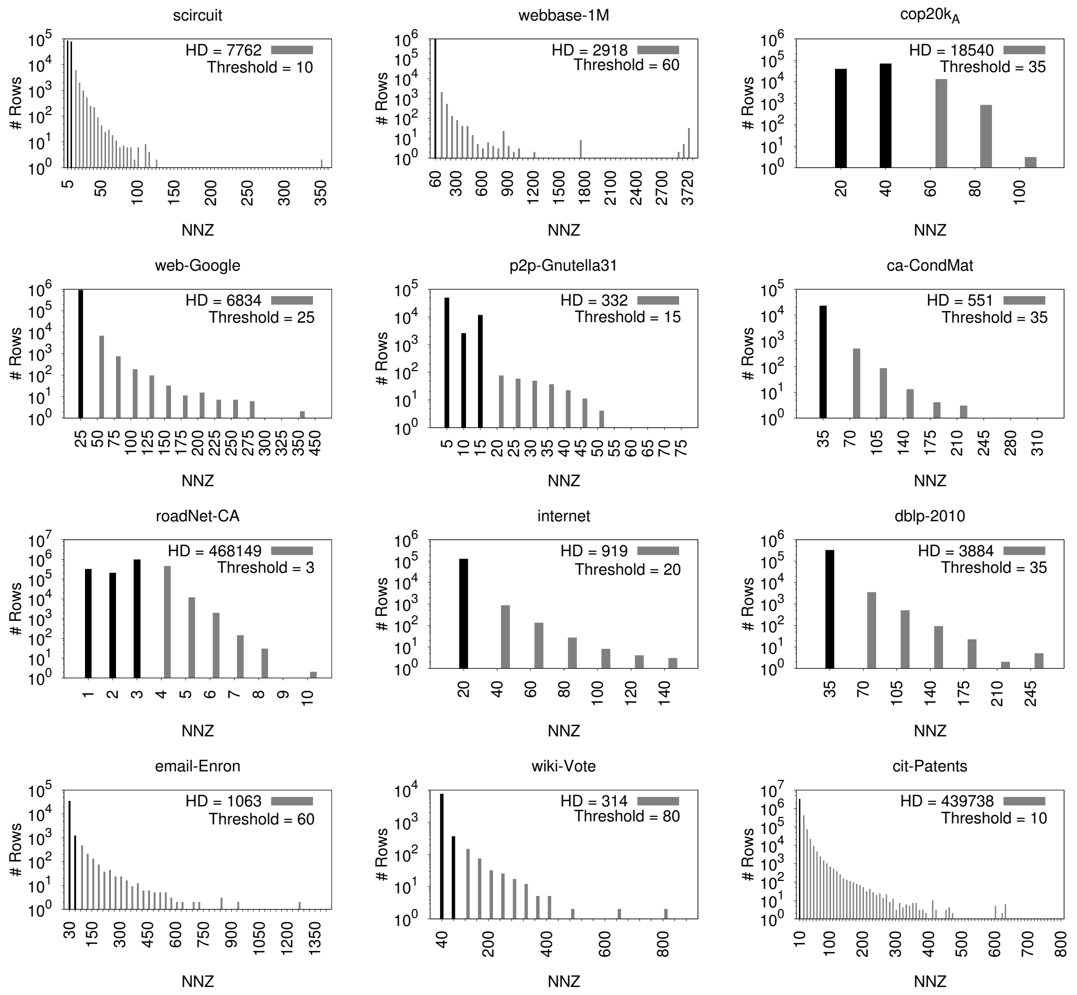

Fig. 5. Row density histogram of 12 sparse scale-free matrices listed in Table I. In the legend, the value "Threshold" indicates the threshold used to identify the high density rows in our experiments. The black bars (to the left) indicate the low density rows. The gray bars (to the right) represent rows with high density. The number of such high density rows is also indicated in the legend as "HD". Note that the Y-axis on all the plots are in logarithmic scale.

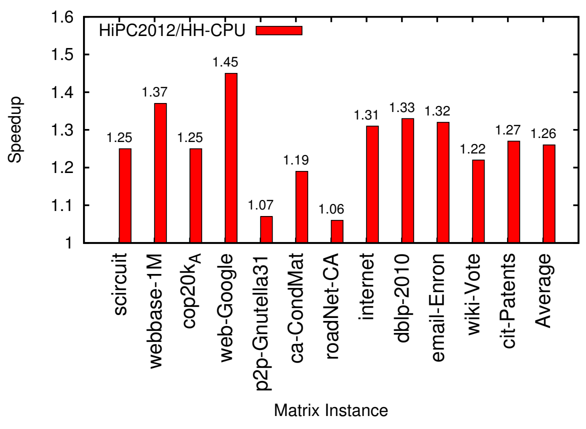

Fig. 6. Figure shows the result of applying our algorithm on the matrices from Table I. The last instance labeled Average indicates the average speedup over the 12 matrices in the dataset.

the row densities. This threshold t plays a crucial role in our algorithm. Let t^* be the best possible threshold for a given matrix. If the chosen threshold, t, is larger than t^*, then our algorithm assigns more work to the GPU during Phase II, and vice-versa. Further, if t is chosen such that all rows are characterized as being low density rows, then our algorithm is identical to Algorithm 1 of [13]. Similarly, if $t = 0$, then our algorithm assigns all of the work to the CPU. Hence, as we increase t from 0 to the largest possible value, the overall time taken by our algorithm should exhibit a convex behavior.

This is shown also experimentally in Figure 8 for the 12 matrices from our dataset. In Figure 8, we also show the time taken by Phases II and III apart from the overall time. In all cases, as the threshold increases, the time taken by Phase II decreases initially and then starts increasing again. The overall time taken also follows a similar trend. In fact, it is also noticed that the time corresponding to a threshold of 0 is close to the time taken by MKL on the instance, and the time taken

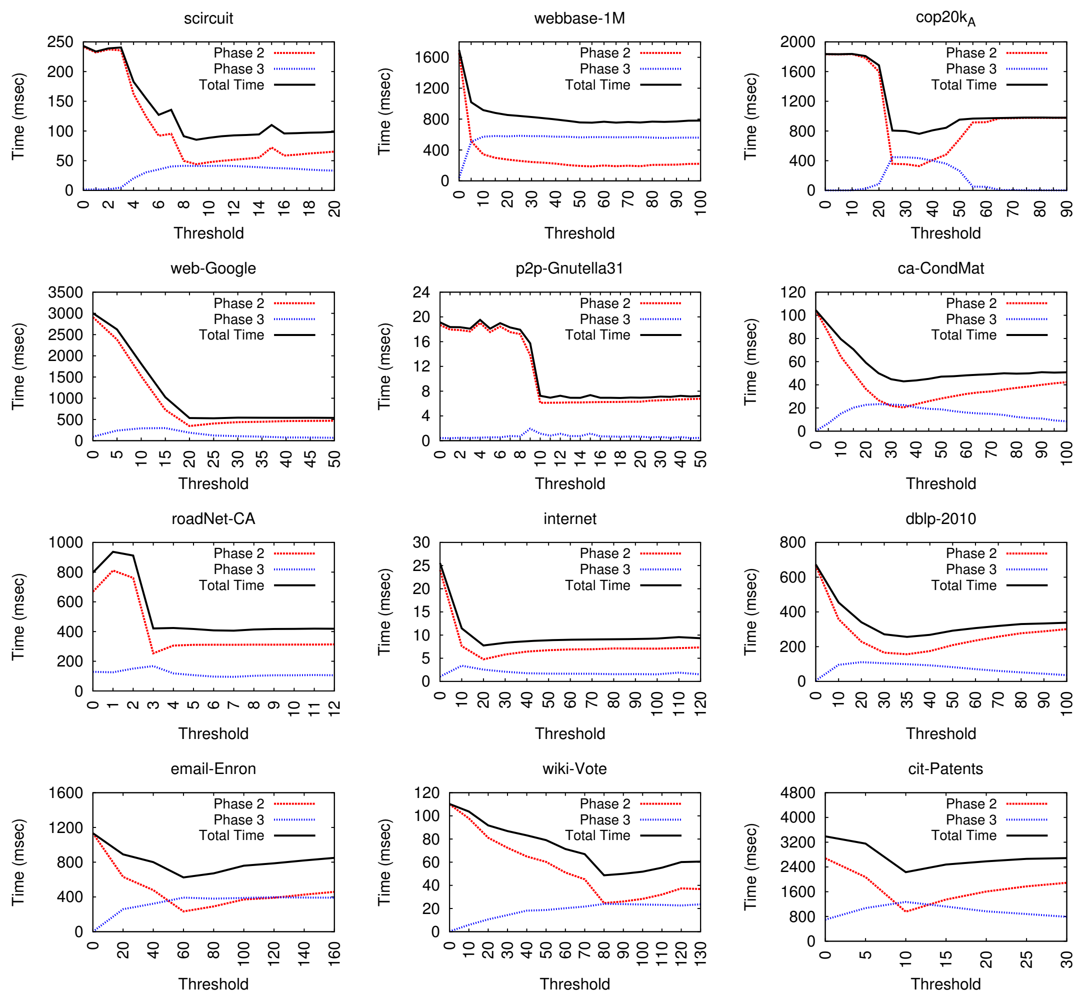

Fig. 8. Figure shows the effect of the threshold on the overall time and the time in Phases II and III for twelve matrices from Table I. The solid black line represents the total time, the dashed red line represents the time taken in Phase II, and the dashed blue line represents the time taken in Phase III. Only Phases II and III were included as they consume the bulk of the overall time taken by our algorithm in any instance. Further, the threshold in each case is varied according to the matrix instance.

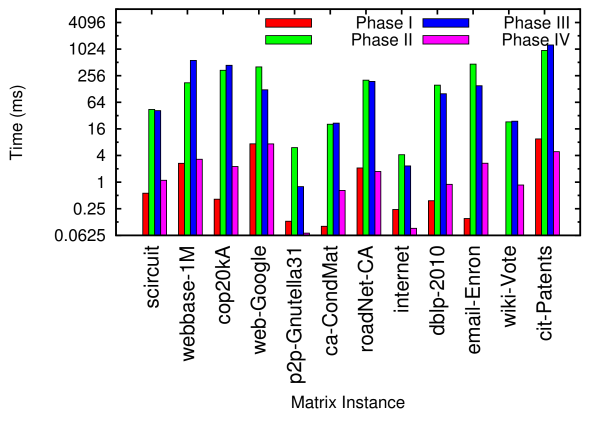

Fig. 7. Figure shows the breakdown of time across various phases of our algorithm. Notice that the Y-axis is on a logarithmic scale.

corresponding to the largest applicable threshold is close to the time taken by [13].

It is also worthwhile to note that the time taken by Phase III does not show any structured behavior as opposed to time taken by Phase II. This can be explained as follows. Even though the computation in Phase III depends on the value of the chosen threshold t, it is not clear as to how the volume of computation corresponding to $A_L \times B_H$ and $A_H \times B_L$ change with varying t.

C. Comparison with Other Approaches

In this section, we show that Algorithm HH-CPU is better suited compared to other possible approaches. We consider two alternatives as follows. In what we call as Algorithm Unsorted-Workqueue, we use a workqueue model for the entire computation of $A \times B$. In other words, in Algorithm Unsorted-Workqueue, the CPU and the GPU multiply independent and contiguous sets of rows of A with the rows of

B. Each such contiguous set of rows (of A) corresponds to a work-unit. The CPU and the GPU access the work-units from opposite ends of the workqueue as described in Section IV-B. This approach can provide for dynamic load balancing across the devices and differs from that of [13] as the heterogeneous algorithm from [13] does a static work partitioning across the CPU and the GPU.

In a second approach, called as Algorithm Sorted-Workqueue, we sort the rows of A according to their sizes, and then apply a workqueue model to compute the product $A \times B$. In Algorithm Sorted-Workqueue, each work-unit corresponds to a contiguous set of rows of A such that the sizes of the rows are in sorted order. This approach also will achieve load balancing across the devices.

The performance of these algorithms compared to Algorithm HH-CPU is shown in Figure 9. For purposes of this comparison, for both algorithm Unsorted-Workqueue and Sorted-Workqueue, we have used empirically obtained best possible values for the sizes of the work-units on the CPU and the GPU. We notice that the overall time taken for Algorithm HH-CPU is 15% smaller on average compared to either of Algorithm Unsorted-Workqueue and Algorithm Sorted-Workqueue on scale-free matrices.

With respect to Algorithm Unsorted-Workqueue, the improvement can be attributed to the load imbalance that algorithms Unsorted-Workqueue and Sorted-Workqueue induce on threads within a warp of the GPU. This load imbalance across threads within a warp of the GPU can result in suboptimal utilization of the GPU.

With respect to Algorithm Sorted-Workqueue, the improvement can be attributed to the following reasons. Observe that a thread on the CPU, or a group of threads on the GPU assigned to a workunit corresponding to a sorted set of rows of the A matrix has to finish multiplying all the rows of the B matrix before proceeding to another workunit. Since the sizes of the rows of the B matrix vary, although in a sorted manner, it becomes difficult to make effective load balancing techniques within a workunit. This also means that the performance of Algorithm Sorted-Workqueue as compared to that of Algorithm HH-CPU can vary across matrices without any observable behavior.

This comparison provides evidence for the fact that mere load balancing across devices may not be sufficient when one considers heterogeneous systems. The algorithm should also be architecture-aware so that the "right" work is assigned to the "right" processor. In our case, the CPU is more appropriate for multiplying dense matrices where it can use techniques such as cache-blocking, and the GPU is more appropriate for multiplying rows with small density where it can make good use of shared memory.

D. Experiments on Synthetic Datasets

To understand the impact of scalefreeness of matrix on Algorithm HH-CPU, we conduct experiments on synthetic matrices where we can control the nature of the matrix. A measure of scalefreeness of a dataset is its α value that indicates the exponent in the power law distribution to which the data

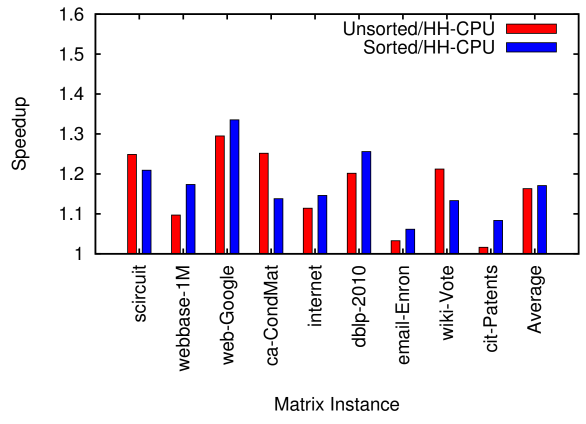

Fig. 9. Figure shows the speedup of our algorithm compared to that of Algorithm Unsorted-Workqueue and Algorithm Sorted-Workqueue. The last instance labeled 'Average' shows the average performance on scale-free matrices from Table I.

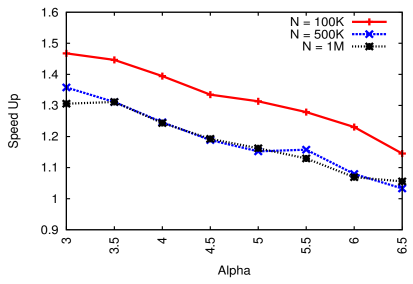

Fig. 10. Figure shows the speedup of Algorithm HH-CPU on synthetic matrices as a function of α.

fits to. The smaller the α, the more scalefree the underlying data is, and as α increases, the degree of scalefreeness of the dataset decreases. For a given dataset, the value of α can be found as described in [1].

In this experiment, we use the GT graph generator [3] to generate graphs whose degree sequence exhibits a scalefree nature. The degree of scalefreeness can be controlled by using the parameters of the generator. These graphs are then interpreted as matrices in a natural way. Now, the row sizes of the matrix exhibit the same scalefree behavior as the corresponding graph does.

We consider three different sizes of matrices, 100K, 500K, and 1M, and α values in the range of $[3, 6.5]$ in steps of 0.5. Unlike the earlier experiments, we now multiply two different matrices A and B with the same α value. For each matrix size and α value, we generate a scalefree matrix that has the given size and an α that is close to the desired value. (GT generator in general does not have a way to produce a distribution with a given α. In turn, one has to specify the number of nonzeros, equivalently the number of edges in the graph, that result in a particular α.) For matrices generated in this manner, we compare the speedup achieved by Algorithm HH-CPU with respect to the algorithm from [13]. The results of the experiment are shown in Figure 10.

As can be noticed from Figure 10, as α increases, the speedup achieved by Algorithm HH-CPU decreases as expected. For matrices of 100 K rows, the speedup is higher than the other two matrix sizes considered in the experiment. This difference is due to several reasons, some of which are stated below. In Phase IV of Algorithm HH-CPU, the tuples generated in $\langle r, c, val \rangle$ format are converted to the CSR format. This step involves sorting based on row indices as explained in Section III-D. We noticed that the matrices with 500K and 1M rows have consistently more tuples in the product matrix. This increase in the number of tuples leads to a bigger portion of time spent in Phase IV that contributes to the relative drop in the speedup.

VI. Conclusions

In this paper, we have proposed efficient heterogeneous algorithms for `spmm` on a CPU+GPU platform targeting real-world and scale-free graphs. We showed that our algorithms in general outperform the corresponding best known implementations. Our techniques are directed at solving two problems: mapping the "right" workload to the "right" processor, and achieving load balancing. Thus, our approach is both architecture-aware, and workload-aware. In future, we would like to study analytical techniques to identify the threshold in Phase I of Algorithm HH-CPU.

It is easy to see that a similar algorithm can be designed for also `csrmm`, which multiplies a sparse matrix A with a dense matrix B. In this case, we imagine that, since B is dense, the work can be divided as multiplying the high-density submatrix A_H of A with B on the CPU and the low-density submatrix A_L of A with B on the GPU. In essence, the algorithm would be very similar to the one we used for `spmm`. We therefore expect that such an algorithm for `csrmm` would be also architecture- and workload-aware.

VII. Acknowledgment

The first author in this work was supported in part by research grants from the Department of Biotechnology, Government of India project BT/65 PR-14715/PBD/16/903/2010.

References

[1] J. Alstott, E. Bullmore, and D. Plenz. Powerlaw: a Python package for analysis of heavy-tailed distributions. PLoS ONE 9(1): e85777, 2014.
[2] 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.
[3] D. A. Bader and K. Madduri. GTgrpah: A suite of synthetic graph generators. Available at https://sdm.lbl.gov/~kamesh/software/GTgraph/
[4] Banerjee, D. S., Sakurikar, P., and Kothapalli, K. Fast, scalable parallel comparison sort on hybrid multicore architectures. In *Proc. AsHES, 2013*.
[5] A. Buluc and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In Proc. ICPP, pp 503–510, 2008.
[6] A. Gharaibeh, B. Costa, E. Santos-Neto, and M. Ripeanu. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In in Proc. IEEE IPDPS (2013).
[7] F. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. ACM T. Math. Soft.,4(3):250-269, 1978.
[8] G.H. Golub, C.F. Van Loan. Matrix Computations. 2nd ed. 1989.
[9] Hong, S., Rodia, N. C., and Olukotun, K. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In *Proc. of SC'13* (2013).
[10] S. Indarapu, M. Maramreddy, and K. Kothapalli. Architecture- and Workload-aware algorithms for Spare Matrix- Vector Multiplication, Under submission, 2014.
[11] Intel Math Kernel Library, https://software.intel.com/en-us/intel-mkl
[12] 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.
[13] K. Matam, S. Indarapu, and K. Kothapalli. Sparse Matrix Matrix Multiplication on Modern Architectures, in Proc. of HiPC, 2012.
[14] NVIDIA Corporation. CUDA: Compute unified device architecture programming guide. Technical report, NVIDIA.
[15] NVIDIA cuSPARSE Library, https://developer.nvidia.com/cusparse
[16] S. C. Park, J. P. Draayer, and S.-Q. Zheng. Fast sparse matrix multiplication. Computer Physics Communications, 70:557.568, July 1992.
[17] 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.
[18] Stanford Network Analysis Platform dataset , http://www.cise.ufl.edu/research/sparse/matrices/SNAP/
[19] Sulatycke, P.D.; Ghose, K.; , Caching-efficient multi-threaded fast multiplication of sparse matrices, in Proc. of IPDPS, pp.117-123, 1998.