

ScholarWorks@GSU

Optimizing Sparse Matrix-Matrix Multiplication on a Heterogeneous CPU-GPU Platform

Authors	Wu, Xiaolong
Citation	Wu, Xiaolong (2015). "Optimizing Sparse Matrix-Matrix Multiplication on a Heterogeneous CPU-GPU Platform." Thesis, Georgia State University. https://doi.org/10.57709/7906679
DOI	https://doi.org/10.57709/7906679
Download date	2026-03-12 17:09:46
Link to Item	https://hdl.handle.net/20.500.14694/4190

OPTIMIZING SPARSE MATRIX-MATRIX MULTIPLICATION ON A HETEROGENEOUS CPU-GPU PLATFORM

by

XIAOLONG WU

Under the Direction of Sushil K. Prasad, PhD

ABSTRACT

Sparse Matrix-Matrix multiplication (SpMM) is a fundamental operation over irregular data, which is widely used in graph algorithms, such as finding minimum spanning trees and shortest paths. In this work, we present a hybrid CPU and GPU-based parallel SpMM algorithm to improve the performance of SpMM. First, we improve data locality by element-wise multiplication. Second, we utilize the ordered property of row indices for partial sorting instead of full sorting of all triples according to row and column indices. Finally, through a hybrid CPU-GPU approach using two level pipelining technique, our algorithm is able to better exploit a heterogeneous system. Compared with the state-of-the-art SpMM methods in cuSPARSE and CUSP libraries, our approach achieves an average of 1.6x and 2.9x speedup separately on the nine representative matrices from University of Florida sparse matrix collection.

INDEX WORDS: Sparse matrix-matrix multiplication, Data locality, Pipelining, GPU

OPTIMIZING SPARSE MATRIX-MATRIX MULTIPLICATION ON A HETEROGENEOUS
CPU-GPU PLATFORM

by

XIAOLONG WU

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2015

Copyright by
Xiaolong Wu
2015

OPTIMIZING SPARSE MATRIX-MATRIX MULTIPLICATION ON A HETEROGENEOUS
CPU-GPU PLATFORM

by

XIAOLONG WU

Committee Chair: Sushil K. Prasad

Committee: Yingshu Li

Yanqing Zhang

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

December 2015

DEDICATION

This thesis is dedicated to my mother, my father and my girlfriend.

ACKNOWLEDGEMENTS

This thesis work would not have been possible without the support of many people. I want to express my gratitude to my advisor Dr Sushil K. Prasad, for providing me an opportunity to work on this thesis. He has been guiding me through all the obstacles encountered in my research work and has been a constant source of motivation.

I must extend my thanks to all the committee members of this thesis, Dr. Yingshu Li and Dr. Yanqing Zhang, for their valuable suggestions to help in shaping this thesis.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
1 INTRODUCTION.....	1
1.1 Background and Problem.....	1
1.2 Motivation of this Study	2
<i>1.2.1 Irregular Data Access.....</i>	<i>2</i>
<i>1.2.2 Redundant Sorting.....</i>	<i>3</i>
<i>1.2.3 Expensive Data Transfer.....</i>	<i>4</i>
1.3 Main Contribution	4
1.4 Organization of the Thesis.....	5
2 PRELIMINARY AND RELATED WORK	6
2.1 ESC-SpMM.....	6
2.2 GPU Architecture and Concurrent Kernels.....	7
2.3 Related Work.....	8
3 Hybrid CPU-GPU SpMM	9
3.1 Data Structure	9
3.2 Partial Sorting	12
3.3 Algorithm Optimization	13

3.3.1	<i>Pipelining</i>	14
3.3.2	<i>Data Partition</i>	16
4	RESULTS	18
4.1	Data Sets and Platforms	18
4.2	Performance	19
5	CONCLUSIONS AND FUTURE WORK	21
	REFERENCES	22

LIST OF TABLES

Table 4.1 Platform configurations	18
Table 4.2 Benchmark matrices	19

LIST OF FIGURES

Figure 1.1 Cache miss ratio of CUSP-SpMM.	3
Figure 3.1 New data structure for hySpMM.....	10
Figure 3.2 Resource occupancy of the five stages (pre-processing, data transfer, multiplication, sorting, and compression) of hySpMM algorithm.	14
Figure 3.3 Pipelining strategies on pre-processing, data transfer, and multiplication stages.....	15
Figure 4.1 Performance.....	19
Figure 4.2 Speedup compared to cuSPARSE and CUSP libraries.	20
Figure 4.3 Cache behavior comparison between hySpMM and CUSP-SpMM.	20

1 INTRODUCTION

Sparse matrix-matrix multiplication (SpMM) is one of the most critical operations in numerous application areas, such as International Thermonuclear Experimental Reactor (ITER) [1] and climate prediction [2] in computational sciences and social network, national security, and system biology in data sciences. For example, SpMM is a fundamental building block for algebraic multigrid method in ITER and shortest path problem [20] in social network applications. Compared to well-studied sparse matrix-vector multiplication (SpMV) [3-8], SpMM is more challenging because of its sparser feature.

1.1 Background and Problem

SpMM operation multiplies a sparse matrix A of size $m \times k$ with a sparse matrix B of size $k \times n$, and produces sparse matrix C of size $m \times n$. The output matrix C is usually denser than both the two input sparse matrices A and B , sometimes even totally becomes a dense matrix if A or B has a high nonzero element ratio. Compressed storage formats are used in SpMM, like coordinate (COO) and compressed row storage (CSR) formats, to make it possible to process very large sparse matrices. Compressed formats save storage space, but the performance of matrix-matrix multiplication in sparse cases is not comparable to dense cases. That's because of the irregular data accesses pattern brought by sparse data structures. Research on sparse matrix vector multiplication (SpMV) also shows similar behavior [3-8].

In recently years, graphics processing units (GPUs) have brought a new chance to high performance computing, which promise much higher peak floating-point performance and memory bandwidth than traditional CPUs. Plenty of research [3-8] has improved its application's performance on GPUs. Several literatures [9-15] optimize SpMM performance on GPUs, greatly improving its performance compared to CUDA libraries (like cuSPARSE [10] and CUSP [11]).

However, when applying these optimizations to applications we also need to consider the time of data transfer, rather than assuming that data is already located in GPU memory.

Data transfer part occupies more than 50% in the SpMM of CUSP library according to our experiments, making it a bottleneck when calling them in real applications. Our work designs a new SpMM algorithm, a hybrid CPU-GPU SpMM (hySpMM) to separate regular operations from the general ESC algorithm in one aspect, and the other is to make it possible to overlap between data transfer and computation. Thus, SpMM can be more applicable to real applications especially for large-scale data.

1.2 Motivation of this Study

We analyze and profile ESC SpMM algorithm, and get three observations, which are irregular data access, redundant sorting, and expensive data transfer.

1.2.1 Irregular Data Access

Irregular data access is observed in expansion stage, where the multiplication between elements of A and B needs indirect memory access. From algorithm 1, expansion stage of ESC-SpMM multiplies each of A (a_{ik}) with a corresponding nonzero element of B (b_{kj}) . The row index of b_{kj} is the same with the column index of a_{ik} . Since the nonzero elements in A are always stored in a row-major pattern, e.g. in CSR and COO formats, nonzero elements in A are processed row by row. Whereas nonzero elements in the same row have a wide range of column indices, making the access of b_{kj} be not contiguous. Besides, matrix B is accessed column by column. If CSR format is used, like in ESC-SpMM and CUSP-SpMM, element accesses are very inefficient.

GPU architecture suffers more from data irregularity, because of its much smaller cache size than CPU. For our test platforms, Intel Xeon i7-2660K has 8MBytes L2 cache, while

NVIDIA is only configured with 1.5MBytes L2 cache. We measure cache miss ratio of SpMM in CUSP library using NVIDIA visual profiler [16]. The cache miss ratios are shown in figure 1.1, tested on NVIDIA GeFore Titan. X-axis represents the names of input sparse matrices, the details of which are given in table 4.2 in section 4. Y-axis shows the L2 cache hit ratios. The average cache miss ratio is about 25%, showing potential space for optimization. Data irregularity feature of SpMM drives us to design new algorithm for GPU. One approach is to reorganize nonzeros into a more regular memory access according to SpMM algorithm. That is to design a data structure which is in accordance with program behavior.

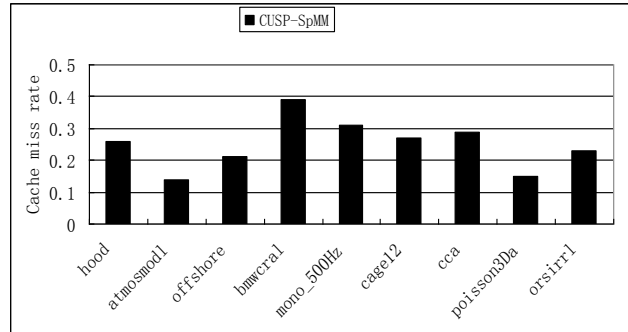


Figure 1.1 Cache miss ratio of CUSP-SpMM.

1.2.2 Redundant Sorting

After the expansion stage, ESC-SpMM obtains an intermediate triple list $C_{triples}$, including duplicated elements. Sorting is employed to get an ordered triple list $C_{triples}^{\zeta}$. Then the following compression stage can do reduction on the triples which are contiguous in location and also have the same indices. Though row indices of $C_{triples}$ are already sorted according to the input matrix A Which is stored row-by-row in CSR/COO format, two-key sorting is still necessary, otherwise only sorting by column indices will ruin the sorted order of row indices.

ESC-SpMM uses radix sort for the two-key sorting, and counting sort is employed in it. As we know, Radix sort has a limitation that the keys are small integers because the time complexity become un-linear from $O(nk)$ to $O(n \log n)$ when the index is very large, which is not satisfied by large matrix. Our aim is to design an algorithm suitable for both large and small-scale matrices, radix sort and counting sort are not suitable any more.

In ESC-SpMM, since the input matrices A and B are stored row-by-row, the intermediate matrix is already sorted in their row indices. If we separate rows and consider only one row at a time, generally only one-key sorting is needed, which is more timesaving than two-key sorting. And we also eliminate the limitation of matrix size.

1.2.3 Expensive Data Transfer

Many researches on SpMM assume sparse matrices are already located on GPU memory [10-13]. This is reasonable when running SpMM as a single kernel, while it is opposite in an application view. As a real application, data (sparse matrices) are generated inside an application. Take algebraic multi-grid as an example, the sparse matrices used in iterative solvers are built in its setup stage. Different matrices may be generated due to different coarsen strategies used in setup stage [17]. Before running SpMM, data transfer is an indispensable stage.

We profile the runtime of the SpMM algorithm in CUSP library using matrix “cage12” from university of Florida collection. Data transfer occupies about 86% of total running time, which is the most expensive part in SpMM algorithm. To push SpMM optimization ahead and make it applicable to real applications, we need to reduce the data movement overhead.

1.3 Main Contribution

In this work, our contributions are as follows: Firstly, we present a hybrid CPU-GPU SpMM algorithm (four-stage algorithm) to isolate the irregular memory access from

multiplication kernel, and allocate different algorithm stage to different platforms. In this way, our algorithm improves data locality and parallelization. Secondly, we employ pipelining strategy to overlap data transfer and computation, fully utilizing CPU and GPU resources. This optimization makes data transfer insignificant to the whole SpMM algorithm. Thirdly, compared to CUSP and cuSPARSE, our algorithm obtains 1.6x and 2.9x speedup on average respectively.

1.4 Organization of the Thesis

The rest of the paper is organized as follows. Introduction and motivation is presented in section 1. Section 2 gives an outline of ESC SpMM algorithm, and illustrates GPU architecture features. What's more related work is also presented in this part. The hybrid CPU-GPU SpMM algorithm (hySpMM) is proposed in section 3. We first describe our new data structure, and then explain the four stages of hySpMM algorithm, which are pre-processing, multiplication, partial sorting, and compression. Furthermore, detailed optimization approaches are given, one is pipelining between data transfer and pre-processing and multiplication stages. We show experiment results on representative sparse matrices in section 4, as well as analyze the cache and memory behavior of our SpMM algorithm. Conclusion is presented in section 5.

2 PRELIMINARY AND RELATED WORK

We introduce a popular ESC algorithm of sparse matrix-matrix multiplication (SpMM). ESC-SpMM is used in CUSP library [10], but CUSP employs an improved version of it, like optimization on load balance, sorting algorithm, and etc. We use the basic ESC algorithm to get its fundamental idea in this section, while in section 3 and section 4 we'll bring in more details about SpMM in CUSP to compare experiment results and illustrate the differences from our new algorithm. Then we illustrate GPU architecture features, especially the ones related to our optimizations.

2.1 ESC-SpMM

ESC-SpMM algorithm follows from the inner product view of multiplication. Each nonzero entry of C is computed as follows:

$$C_{i,j} = A_{i,:} \times B_{:,j} = \sum_k A_{i,k} B_{k,j}$$

Whether $c_{i,j}$ is a nonzero entry depends on both the i^{th} -row in A and the j^{th} -column in B. ESC-SpMM consists of three stages, expansion, sorting, and compression, which are shown in algorithm 1. The first stage implements the inner product between rows of A and columns of B, and then generates an intermediate matrix Ck which is represented by a triple list (COO format). Ck is usually larger than the final result C, because of more than one values are computed for a $c_{i,j}$. Thus, we need to combine triples with the same row and column indices together. Due to the disordered distribution of the triples, sorting algorithm is employed to sort the triple list according to the two indices. Then the triples with the same indices will be continuous, and then compressed to get the result matrix C.

Algorithm 1: SpMM

Input: A, B

Output: C

```

1 M <- slice(A)
  for k=0,...,M
2   Ck <-expand (Ak, B)
3   Ck <- sort (Ck)
4   Ck <- compress (Ck)
5   C <- construct (C)

```

ESC-SpMM algorithm gives a basic approach to SpMM operation, whereas there is space to improve its performance. CUSP optimized load balance, sorting algorithm of ESC-SpMM. In this paper, we illustrate three observations on ESC-SpMM and show our solution to improve it.

2.2 GPU Architecture and Concurrent Kernels

Modern general-purpose graphics processing units (GPUs) are fully programmable many-core platforms. NVIDIA's Fermi GPU architecture consists of multiple streaming multiprocessors (SMs) each consisting of 32 cores, each of which can execute one floating point or integer instruction per clock. SMs employ a Single Instruction Multiple Thread (SIMT) architecture. A group of 32 threads called a warp is the minimum execution unit. Once scheduled on a SM, the threads in a warp share the same instruction and can execute in a fairly synchronous fashion. In addition, The SMs are supported by a second-level cache (L2 cache).The L2 cache covers GPU local DRAM as well as system memory.

From version 5.0, Compute Unified Device Architecture (CUDA) enables dynamic parallelism by letting a kernel function launch new kernels. Moreover, it also enables creation and use of streams and events without CPU involvement. A stream, which is a sequence of commands that execute in order, allows the overlapping of data transfer and computation in CPU and GPU. Different streams may execute concurrently. The Fermi architecture supports the simultaneous execution of kernels. The benefits are 1) utilization of whole GPU by simultaneous

execution of small kernels and 2) time savings by overlapping kernel execution with device to host memory copy.

2.3 Related Work

There has been a flurry of work [9-15] related to optimization sparse matrix operations, mainly focusing on sparse matrix-vector multiplication (SpMV). Some research is optimizing sparse matrix-dense matrix multiplication, which is similar to SpMV. In this paper, we studied sparse matrix-sparse matrix multiplication (SpMM), which is the most irregular kernel among sparse operations. Research on SpMM is not too much, but there are some great results by far.

NVIDIA CUSP library [11] realized SpMM based on ESC algorithm, and also improved it in three aspects: adding a reorder stage, graph-based multiplication, sorting improvement, and etc. The reorder stage is grouping rows of similar total work and placing the rows in a non-decreasing order of the work-per-row. This stage improves load balance, but ruins the sorted row indices at the same time. The sorting algorithm should be two-key sorting, which is more expensive than one-key partial sorting in hySpMM. Besides, our data partitioning for pipelining approach also insures a good load balance. Another NVIDIA library, cuSPARSE, also includes SpMM kernel, but it is not open-source and opaque to us. Recently, W. Liu et.al optimized SpMM on GPU architecture, their main optimization methods are hybrid method for the result matrix pre-allocation, fast merging, and heuristic load-balancing. We don't care too much about the result matrix pre-allocation is because of the hybrid algorithm, that we don't need to allocate all data in GPU memory. Through data partitioning, we pre-calculate the size of sub-matrices and use the upper bound strategy to pre-allocate the space for intermediate data. In partial sorting stage, we also use concurrent kernel techniques to enhance parallelism on GPU. Combinatorial BLAS [15] also supports SpMM, but they focus on a distributed platform.

3 Hybrid CPU-GPU SpMM

We propose a hybrid CPU-GPU SpMM algorithm, hySpMM, trying to solve irregular data access, redundant sorting, and expensive data transfer problems we observed. In this section, we introduce hySpMM algorithm stage by stage, and also specify which platform (CPU or GPU) this stage is designed for.

3.1 Data Structure

As mentioned in section 1, though CSR format usually achieves good performance on SpMV, its advantage is not so obvious on SpMM. The main reason is that matrix B is accessed in column-major pattern, which is very inefficient for CSR format. Besides, although ESC-SpMM uses CSR format for matrices A and B, but the intermediate results are also stored in a triple list, which is COO format. Thus, we choose COO format as the basic sparse format, and design a new data structure based on it.

In SpMM algorithm, there are three sparse matrices, and each of them needs to be stored in a sparse format. To avoid irregular data access as much as possible, we design a different data structure based on COO format for matrix A and B.

Generally, data structures are independent with programs/algorithms, only used to store data and show some data characteristics. Like COO format, you cannot catch the features of SpMM algorithm based on it. This independence makes sure of the isolation of data and algorithm, which is good for software engineering. But sometimes, auxiliary data structure (or attributes, or information) is very helpful to algorithm's performance. Our idea is to expose more information of SpMM process by data structures.

repeat_A	COO format: A	Duplicated B
		(0,0,1)
		(1,1,2)
1	$\begin{bmatrix} (0, 0, 10) \\ (1, 1, 20) \\ (1, 2, 30) \\ (1, 3, 40) \\ (2, 3, 50) \\ (3, 1, 60) \end{bmatrix}$	(1,3,3)
2		(2,0,4)
2		(2,1,5)
2		(3,1,6)
2		(3,3,7)
2		(3,1,6)
		(3,3,7)
		(1,1,2)
		(1,3,3)

Figure 3.1 New data structure for hySpMM

For matrix A, we add an additional array to mark the repeated times each nonzero are involved in the SpMM algorithm. For matrix B, we extend it by storing duplicated triples. The order of the triples in B is in accordance with the SpMM multiplication order, from top to down, left-to-right for matrix A. The data structure for previous SpMM example is shown in figure 3.1. Obviously, the new data structure takes more storage space. We trade space for shorter running time. Since each triple in $A[i]$ directly multiplies with triple $B[j+i]$, making sure of the contiguous data access for both A and B during multiplication operations. Multiplication is the most critical point for SpMM performance excluding data transfer stage. From a first glance, this new data structure may not be attractive. However, it opens more possibilities to further optimization, especially to solve data transfer problem.

Algorithm 2: Pre-processing algorithm

Input: A, B, Repeat_A[]

Output: DuplicateB

```

1 for each column_index_A i in matrix A do
2   RowIdB<- COOcollIndexA[i]
3   sum[i]<- Prefix_sum(Repeat_A[i])
4   Insert COO_B [RowIdB] to COO_B[sum[i-1]]
5 end for

```

To build the new data structure, an additional stage is introduced as pre-processing stage (algorithm 2). Pre-processing stage transforms matrices A and B from COO formats to our data structure. According to different triple pairs of A and B operating on, the new data structure is built to reflect multiplication information. Based on the new data structure, the expansion stage of ESC-SpMM is split into two stages, pre-processing and multiplication stages. Multiplication stage is shown in algorithm 3. Multiplication stage simply loops all triple pairs (A and B), multiplies the two values, and then stores them into the template matrix $C_{triples}$ as triples. The row and column indices of $C_{triples}$ are the row indices of A and column indices of B accordingly. The new multiplication stage is not only simple, but has more regular data access because of no indirect indexing.

Algorithm 3: Element-wise multiplication

Input: new data structure

Output: intermediate matrix C'

```

k = 0;
for i=1 to nnz(A) do
  times = repeat_A[i];
  ele_A = mat_A[i];
  for j=1 to times do
    ele_B = dup_mat_B[k];
    C'[k] = ele_A * ele_B;
    k++;
  end
end
End

```

Except building a new data structure, the main difference from ESC-SpMM algorithm (algorithm 1) is splitting one expansion stage into two stages. The irregular behavior is remained in pre-processing stage, operating on CPU, while the multiplication stage is processed on GPU because of its more regular behavior after pre-processing. Since GPU shows much more performance potential on regular data than CPU (like dense matrix-matrix multiplication), the new multiplication stage may get higher performance. As we mentioned in section 2, GPU has

smaller cache sizes than CPU, moreover GPU is not good at dealing with branches. These features make CPU a better choice to execute pre-processing stage. Another benefit of the new two stages is providing larger possibilities for overlapping between data transfer and computation. We'll show this in section 4.

3.2 Partial Sorting

In ESC-SpMM radix sort is used, taking both row and column indices as keys. Since row indices are already sorted due to the row-by-row execution order in matrix A, we only need to sort within a row of matrix $C_{triples}$. In this paper, we consider input matrix in COO format also ordered in row indices, which is the general case of real data sets. However, if we consider all triples as a whole, two-key sorting is still necessary since we need to keep row indices still ordered after the sorting.

We propose partial sorting approach in hySpMM. Partial sorting is a straightforward idea that we separate triples of $C_{triples}$ into different groups by row indices. Each group only consists of triples from the same row. In this way, we can only implement a general one-key sorting algorithm on one group, which is more time efficient, and this stage is executed on GPU. Though partial sorting has benefit due to its one-key sorting characteristics, there is a problem to implement it efficiently on GPU. Because of the diverse number of nonzero elements per row, GPU cannot be fully utilized if there are not many nonzeros in a row. One method to resolve this problem is to use different sorting algorithm for different group.

After sorting stage, $C_{triples}^c$ is generated from $C_{triples}$ with row and column indices both sorted. Compression stage of hySpMM is the same with that of ESC-SpMM. Compression stage is also executed on GPU. Other than the four stages, we also need data transfer stage to copy input matrices from CPU to GPU.

By now, we have designed a hybrid CPU-GPU algorithm based on ESC-SpMM, to extract irregularity and use partial sorting to increase parallelization. The main problem coming with the new algorithm is more storage space. We can solve this problem using pipelining strategy, which will be introduced in section 3.3. Apart from this, our algorithm shows several advantages. First, our algorithm splits the expansion stage of ESC-SpMM into two stages, pre-processing and multiplication. In this way, we distinguish regular data locality from the irregular part, allowing GPU to develop better performance. Second, this hybrid algorithm allows us to execute two different stages (pre-processing and data transfer stages) on different platforms according to their characteristics. In this way, we are able to use pipelining strategy to overlap among different stages. Last, partial sorting avoids two-key sorting and allows concurrent GPU kernels, which is more flexible for sparse matrices.

Apart from the advantages our algorithm brings, there are also challenges to pursue high performance of it. First, to design a good pipelining approach is critical to hide data transfer overhead. Only if the data transfer time is insignificant, it is possible to apply our algorithm to real applications. Second, though partial sorting decreases the sorting burden by only sorting column indices, there are not so many elements to sort in each row, making it hard to full utilize GPU resources, which may harm the performance. Also a big variation of nonzero elements in each row requires the sorting algorithm be efficient on both short and long arrays. Last, further optimization considering GPU architecture is needed. In the next section, we'll address our solution to these challenges.

3.3 Algorithm Optimization

We state detailed optimization methods for hySpMM algorithm in section 3.3, including two main aspects: pipelining and GPU architecture-specific optimization. Inspired from J. Li's

work [19] about how to design pipelining algorithm for dense matrix-matrix multiplication on a heterogeneous CPU-GPU platform, we state our pipelining strategy as follows.

3.3.1 Pipelining

We first analyze the resource usage of the five stages of hySpMM, which are pre-processing, data transfer, multiplication, sorting, and compression. Resources we considered are CPU, CPU memory, PCIe, GPU, and GPU memory. Figure 3.2 shows the resource usage of each pipelining step. Except pre-processing stage, all the other four stages are executed on GPU, thus they need both GPU and GPU memory. Pre-processing stage executes on CPU, it needs CPU and CPU memory. Data transfer can be executed in Direct Memory Access (DMA) pattern, so it occupies CPU memory, GPU memory and PCIe bus. There is data dependence between every two continuous stages, pipelining can only occurs among the first three stages without resource conflicts. Although data transfer stage shares CPU memory with pre-processing stage and GPU memory with multiplication stage, since the three stages use different data in CPU (GPU) memory between different pipelining steps, CPU (GPU) memory can be shared by more than one steps without conflicts.

	CPU	CPU memory	PCIe bus	GPU memory	GPU
pre-processing					
data transfer					
multiplication					
sorting					
compression					

Figure 3.2 Resource occupancy of the five stages (pre-processing, data transfer, multiplication, sorting, and compression) of hySpMM algorithm.

The pipelining strategy is shown in figure 3.3. X-axis is a time line, and Y-axis shows different stages. The time of each data transfer block (red one) occupies more than 5 times of the

sum of the time of pre-processing and multiplication stages. About 9% time of data transfer stage can be overlapped with pre-processing stage, and 11% overlapped with multiplication stage. This is because data transfer is very time-consuming; it occupies most of the execution time. Though there is still part of data transfer overhead exposed, most of the computation time is overlapped by data transfer. Figure 3.2 and 3.3 show that our pipelining strategy can improve algorithm performance by fully utilizing resources.

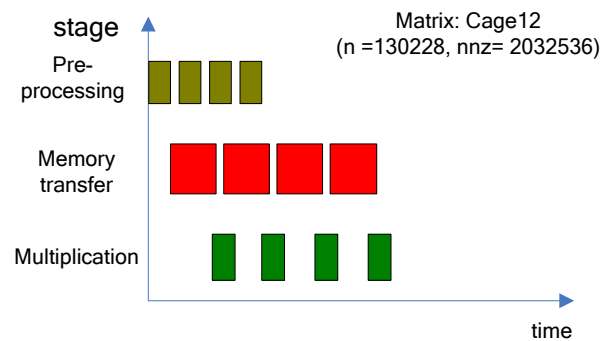


Figure 3.3 Pipelining strategies on pre-processing, data transfer, and multiplication stages.

Two strategies are used to implement our pipelining method. First, the input matrices are split into plenty of blocks (data partitioning is stated in section 3.3.2), and each thread is responsible for each block pair (including blocks of A and B). Direct Memory Access (DMA) method is employed for data transfer. Thus, when a thread is running pre-processing stage, data transfer can be executed simultaneously. Furthermore, we use multiple threads to parallelize this process, decrease the CPU idle time as much as possible. Second, for the pipelining between data transfer and multiplication, we use CUDA streams to concurrently execute asynchronous CUDA memory copy functions with CUDA kernels (multiplication stage).

Pipelining strategy has three benefits. First, it is more promising to integrate optimized SpMM into real applications. Since in real applications data transfer time is not negligible, this strategy makes it possible for optimized SpMM kernels in previous research to be applied in real

applications. Second, our algorithm is easily tolerant large-scale data. When sparse matrices are too large to reside in GPU memory, our algorithm can split the input and output matrices and processing them independently and in pipeline. Last, pipelining strategy helps alleviate the increased storage space in our algorithm. Our algorithm can make use of the larger CPU memory instead of only GPU memory.

3.3.2 Data Partition

We partition matrices A and B due to the size of GPU global memory, to make all the three sub-matrices fully stored in it. Actually, we also need to consider the intermediate triple list $C_{triples}$, which consumes more memory than C, and reserve space for it. Upper bound method is used to pre-allocate space for $C_{triples}$, computing an upper bound of the number of nonzero entries in $C_{triples}$ and allocating corresponding memory space. We consider matrices A and B are square matrices both with the size of $n \times n$, and with nnz_A and nnz_B as their number of nonzeros.

$$nnz(C_{triples}) = n^2 - \max(p_A, p_B, p_A \times p_B) \quad (1)$$

where p_A is the number of rows of A with all zeros in it, and p_B is the number of columns of B with all zeros in it. This equation means when there is a zero row in A and a zero column in B, the corresponding C element in the cross point of this row and this column is zero. Even if there is no zero columns in B ($p_B = 0$), p_A rows in C are filled with zeros. We allocate space for $C_{triples}$ by excluding these positions. In real applications, especially graph applications, zero rows (or columns) are existed in general. The nonzero ratio of $C_{triples}$ is

$$ratio(C_{triples}) = 1 - \frac{\max(p_A, p_B, p_A \times p_B)}{n^2} \quad (2)$$

If both P_A and P_B equals to 0, we have to allocate memory space as a dense matrix. From equation 2, the result matrix C usually will be denser than input matrices A and B.

We use the same method to pre-allocate space for intermediate matrix blocks ($C_{triples}^{Block}$). The following equation is used to calculate the partition size.

$$size(A^{Block}) + size(B^{Block}) + size(C_{triples}^{Block}) \leq 0.9M \quad (3)$$

where M is the size of GPU memory, A^{Block} and B^{Block} are blocks to be processed in each step. We only consider 90% available memory space for other temporary data. If we assume nb blocks are generated, and use nM to represent the numbers can be stored in GPU memory, equation 3 transforms to

$$\frac{(nnz_A + nnz_B)}{nb} + nnz(C_{triples}^{Block}) \leq 0.9nM \quad (4)$$

Only nb is need to decide in formation 4, since $nnz(C_{triples}^{Block})$ can be estimated its upper bound from equation 1, on matrix blocks A^{Block} and B^{Block} .

Another thing we need to considered is the parallelization on CPU. Assume we use np CPU threads, and each thread manages a block pair of (A^{Block} , B^{Block}). Thus, we need to divide another np factor from nM, to make sure all tasks on GPU can be allocated at the same time. So the limitation now is

$$\frac{(nnz_A + nnz_B)}{nb} + nnz(C_{triples}^{Block}) \leq \frac{0.9nM}{np} \quad (5)$$

Using inequation 5, we can partition data as even as possible, to make sure the resources are fully utilized and insure a good load balance.

4 RESULTS

We test hySpMM performance and compare its performance with both cuSPARSE and CUSP libraries. We also give analysis on data locality by observing cache behavior.

4.1 Data Sets and Platforms

Our experiments are tested on a heterogeneous platform, with Intel Xeon i7-2600K CPU and NVIDIA GeForce Titan GPU. The parameters of the two platforms are listed in table 4.1. As we mentioned, GPU has much higher peak floating-point performance, but its cache and memory sizes are not comparable to CPU.

Table 4.1 Platform configurations

Parameters	Intel Xeon i7-2600K	NVIDIA GeFore Titan
Frequency	3.4 GHz	876 MHz
# of cores	4	2048
LLC [#] Size	8 MB	1.5MB
Memory Size	32 GB	6 GB
Memory Bandwidth	21 GB/s	288 GB/s
System software and Library	Operating system is ubuntu and kernel version is linux-3.2.0. CUDA 7.0, CUSP v0.4.0, CUSPARSE v2	

[#] LLC: Last Level Cache.

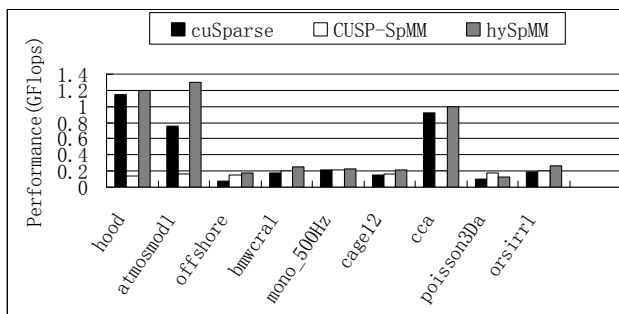
We choose 9 sparse matrices from the University of Florida sparse matrix collection [18], to show diverse sparsity features. The matrix set is given in table 4.2. The matrices have various sizes and nonzeros per row.

Table. 4.2 Benchmark matrices

Matrices	n	nnz	nnz per row
Hood	220542	9895422	44
Atmosmodl	1489752	10319760	6
Offshore	259789	4242673	16
Bmwcral	148770	10641602	71
mono_500Hz	169410	5033796	29
cage12	130228	2032536	15
Cca	49152	139264	3
Poisson3Da	13514	352762	26
Orsirrl	1030	6858	7

4.2 Performance

Our hySpMM performance is shown in figure 4.1, counting data transfer time. For the nine matrices, their performance numbers vary from 100 MFLOP/s to 1GFLOP/s, due to various sparsity features. Our performance number is not attractive because we count data transfer time in, to simulate the environment in real applications. Matrices “hood”, “atmosmodl” and “cca” achieve relative higher performance, because there matrix size is enough large to benefit from the pipeline optimization, what’s more the relatively small number nonzero elements per row reduce their pre-processing time.

**Figure 4.1 Performance**

We also compare our performance with that of cuSPARSE and CUSP libraries in figure 4.2. Due to the large overhead of data transfer, cuSPARSE and CUSP also show low performance.

HySpMM achieves an average speedup of 1.6 times and 2.9 times over cuSPARSE and CUSP respectively, which shows the benefit of hySpMM.

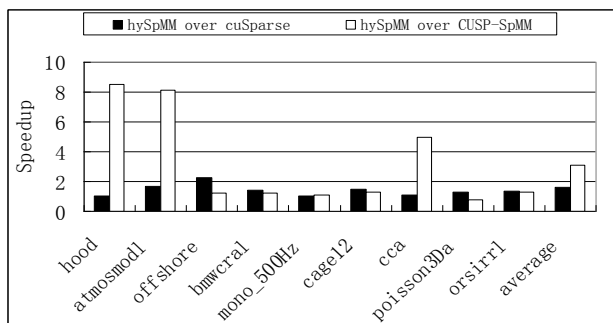


Figure 4.2 Speedup compared to cuSPARSE and CUSP libraries.

HySpMM separates expansion stage into pre-processing and multiplication stages, the first one is executed on CPU and the second one on GPU. Pre-processing stage conserves the irregularity, while multiplication stage only contains regular memory accesses, which is more beneficial to GPU. We prove hySpMM having better data locality by measuring cache behavior. L2 cache hit ratio is measured by NVIDIA visual profiler on the nine matrices (figure 4.3). Compared to CUSP, hySpMM obtains close to 90% cache hit ratio on average, which is much better than 75% from CUSP. hySpMM improves data locality to improve its overall performance.

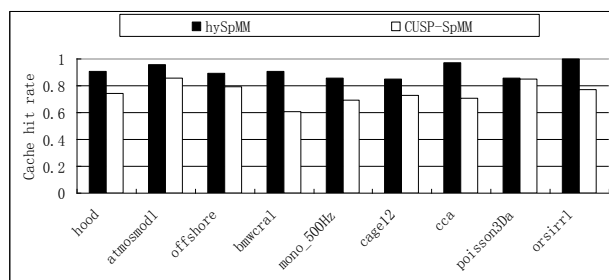


Figure 4.3 Cache behavior comparison between hySpMM and CUSP-SpMM.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we optimized the widely used Sparse Matrix-Matrix multiplication (SpMM) on a heterogeneous CPU-GPU platform. Based on basic ESC algorithm of SpMM, we observed three problems which affect its performance: (1) data locality problem in the expansion stage, (2) redundant operations in the sorting stage, and (3) poor utilization of CPU-GPU pair in processing irregular data simultaneously. The new designed hySpMM algorithm solves these problems through algorithm optimization, pipelining strategy, and GPU architecture-specific optimization. Compared with the state-of-the-art SpMM methods in cuSPARSE and CUSP libraries, our approach achieves an average of 1.6x and 2.9x speedup separately on the nine representative matrices from University of Florida sparse matrix collection. However, there is still optimization space for SpMM.

REFERENCES

- [1] M. D. Adam Hill. The international thermonuclear experimental reactor. Technical report, 2005.
- [2] M. F. Khairoutdinov and D. A. Randall. A cloud resolving model as a cloud parameterization in the near community climate system model: Preliminary results. *Geophys. Res. Lett.*, 28(18):3617C3620, 2001
- [3] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Vol. 2. No. 5. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.
- [4] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '10*, pages 115–126. ACM, 2010.
- [5] B.-Y. Su and K. Keutzer. clspmv: A cross-platform opencl spmv framework on gpus. In *Proceedings of the 26th ACM international conference on Supercomputing, ICS '12*, pages 353–364. ACM, 2012
- [6] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Physics: Conf. Ser.*, volume 16, pages 521–530, 2005.
- [7] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing, SC '07*, pages 38:1–38:12. ACM, 2007.
- [8] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM*

SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13). ACM, New York, NY, USA, 117-126.

- [9] E. Cohen, "On optimizing multiplications of sparse matrices," in *Integer Programming and Combinatorial Optimization*, ser. *Lecture Notes in Computer Science*, W. Cunningham, S. McCormick, and M. Queyranne, Eds. Springer Berlin Heidelberg, 1996, vol. 1084, pp. 219–233.
- [10] CUDA CUSPARSE Library. NVIDIA, 2010. <https://developer.nvidia.com/cuSPARSE>
- [11] Bell, Nathan, and Michael Garland. "CUSP: Generic parallel algorithms for sparse matrix and graph computations." *Version 0.3.0* (2012): 35.
- [12] Dalton Steven, Nathan Bell, and Luke Olson. "Optimizing sparse matrix-matrix multiplication for the GPU." (2013). Technical Report.
- [13] Weifeng Liu; Vinter, B., "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data," *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp.370,381, 19-23 May 2014.
- [14] K. Matam, S. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *High Performance Computing (HiPC), 2012 19th International Conference on*, 2012, pp. 1–10.
- [15] Buluc, A.; Gilbert, J.R., "Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication," *Parallel Processing, 2008. ICPP '08. 37th International Conference on* pp.503-510, 9-12 Sept. 2008. doi: 10.1109/ICPP.2008.45
- [16] CUDA Visual Profiler
- [17] R. Falgout. *An introduction to algebraic multigrid computing*. *Computing in Science Engineering*, 8(6):24–33, nov.-dec. 2006. ISSN 1521-9615.

- [18] T. A. Davis and Y. Hu, The University of Florida Sparse Matrix Collection, ACM Transactions on Mathematical Software, Vol 38, Issue 1, 2011, pp 1:1 - 1:25.
<http://www.cise.ufl.edu/research/sparse/matrices>
- [19] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2012. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs. In Proceedings of the 26th ACM international conference on Supercomputing (ICS '12). ACM, New York, NY, USA, 377-386.
- [20] Timothy M. Chan. 2007. More algorithms for all-pairs shortest paths in weighted graphs. In Proceedings of the thirty-ninth annual ACM symposium on Theory of computing (STOC '07). ACM, New York, NY, USA, 590-598. DOI=10.1145/1250790.1250877.