International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:6, No:1, 2012

Accelerating Sparse Matrix Vector Multiplication on Many-Core GPUs

Weizhi Xu, Zhiyong Liu, Dongrui Fan, Shuai Jiao, Xiaochun Ye, Fenglong Song, and Chenggang Yan

***Abstract*—**Many-core GPUs provide high computing ability and substantial bandwidth; however, optimizing irregular applications like SpMV on GPUs becomes a difficult but meaningful task. In this paper, we propose a novel method to improve the performance of SpMV on GPUs. A new storage format called HYB-R is proposed to exploit GPU architecture more efficiently. The COO portion of the matrix is partitioned recursively into a ELL portion and a COO portion in the process of creating HYB-R format to ensure that there are as many non-zeros as possible in ELL format. The method of partitioning the matrix is an important problem for HYB-R kernel, so we also try to tune the parameters to partition the matrix for higher performance. Experimental results show that our method can get better performance than the fastest kernel (HYB) in NVIDIA's SpMV library with as high as 17% speedup.

***Keywords*—**GPU, HYB-R, Many-core, Performance Tuning, SpMV

I. Introduction

MANY-CORE architectures like GPUs become more and more popular, because they can offer both high peak computational throughput and high peak bandwidth, which are much higher than those of conventional multi-core platforms based on general-purpose CPUs. That is good news for high performance computing community, but there is no free lunch. We can get high performance more easily with regular applications like dense matrix multiplication on GPUs, while it is more difficult to efficiently exploit the advantages of GPUs for irregular applications like Sparse Matrix Vector Multiplication (SpMV).

SpMV can be described as follows. y←Ax+y, where A is a sparse matrix, x and y are both dense vectors. SpMV is a very important kernel used in scientific and engineering computations. Methods for efficiently computing SpMV are often critical to the performance of many applications.

SpMV is memory bandwidth-bound and GPUs offer sufficiently high bandwidth, so it is an opportunity to improve the performance of SpMV on GPUs. But it is also a challenge to optimize SpMV on GPUs because SpMV is an irregular computation which requires many indirect and irregular memory accesses. Irregular memory accesses usually lead to low bandwidth efficiency on GPUs, so we need to develop new data formats to store sparse matrices on GPUs in order to make good use of GPUs' high bandwidth.

There are three contributions in this paper.

Authors are with Key Lab of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences. (e-mail: {xuweizhi, zyliu,fandr,jiaoshuai,yexiaochun,songfenglong,yanchenggang}@ict.ac.cn).

Weizhi Xu, Shuai Jiao and Chenggang Yan are also with Graduate University of Chinese Academy of Sciences, Beijing, China.

First, after analyzing the experimental data, we find that the best kernel in NVIDIA's SpMV library, HYB[7], does not work perfectly, and there is still some space to improve the performance of HYB kernel because there are still a large number of non-zeros in the COO portion of HYB for some matrices.

Second, we propose a new storage format called HYB-R. HYB partitions a matrix into two parts, ELL and COO, while HYB-R partitions a matrix recursively into ELL and COO. That means the COO portion of the matrix can be partitioned into ELL and COO recursively to form the HYB-R format only if there are enough non-zeros in the COO portion. So there are usually more non-zeros placed in ELL portion for HYB-R format than HYB format. Because the ELL kernel is roughly three times faster than COO kernel, we can conclude that the HYB-R kernel is faster than HYB kernel. Experimental results also show that HYB-R kernel can get better performance than HYB kernel, with as high as 17% improvement.

Third, we find that the partition method used in HYB format is not the optimal, so we try to tune the parameters on how to partition the matrix. The parameter K is used to partition the matrix into ELL and COO. K is the number of columns in the ELL portion. Two methods are used to choose K in the experiment, but it is found that neither method is the optimal and further efforts are still needed to tune the parameter K.

II. SpMV On GPUs

In this section, several storage formats for sparse matrices are introduced, CSR, COO, ELL and HYB. Optimization methods using these storage formats on GPUs are also represented.

A. CSR

The compressed sparse row (CSR) format is one of the most popular general-purpose sparse matrix representations. The CSR format stores a variable number of non-zeros per row without wasting memory space to store zeros. Fig. 1 (b) illustrates the CSR representation of an example matrix. CSR explicitly stores column indices and non-zero values in arrays *col_idx* and *values*. The *row_ptr* array stores the pointers to the first non-zero of each row.

The scalar CSR kernel [7] assigns one thread to each matrix row. Each thread reads the elements of its row sequentially. The problems of the scalar CSR kernel are (1) Threads within a warp access the arrays, *col_idx* and *values*, with uncoalesced memory accesses. That leads to low memory bandwidth efficiency. (2) When the scalar kernel is applied to a matrix with a highly variable number of non-zeros per row, many

threads within a warp may stay idle while the threads with longer rows continue running.

The vector kernel [7] assigns one warp to process each matrix row. The vector kernel accesses indices and data contiguously. Therefore, it overcomes the first deficiency of the scalar approach. The problems of the vector kernel are (1) It requires an additional intra-warp parallel reduction to sum per-thread results together. (2) Efficient execution of the vector kernel demands that matrix rows contain a number of non-zeros greater than the warp size, so the performance of the vector kernel is sensitive to matrix row size.

B. COO

The coordinate (COO) format is another general-purpose sparse matrix representation. As shown in Fig. 1(c), the three arrays *values*, *row_idx* and *col_idx* store the values, row indices and column indices, respectively, of the non-zeros.

COO kernel [7] assigns one thread to each non-zero, and then performs a segmented reduction operation to sum values across threads. The primary advantage of the COO kernel is that its performance is insensitive to irregularity of the data structure. Therefore, COO method offers robust performance across a wide variety of sparse matrices. One drawback of COO format is that it needs more storage space to store *row_idx* array than CSR format.

C. ELL

The ELL format is well-suited to GPUs. As shown in Fig. 1(d), an M-by-N sparse matrix with at most K non-zeros per row is stored as a denser M-by-K *values* array and a corresponding M-by-K *col_idx* array. All rows with less than K non-zeros are zero-padded to length K.

The two ELL arrays are stored in column-major order and zero-padded for alignment. ELL is most efficient when the maximum number of non-zeros per row does not substantially differ from the average. Otherwise, more zeros are padded, the memory space and bandwidth are not used efficiently and the performance degrades.

D. HYB

Hybrid (HYB, Fig. 1(e)) format [7] overcomes the drawback of ELL. The purpose of HYB format is to store the typical number of non-zeros per row in the ELL format and the remaining entries of the rows with more than the typical number of non-zeros in the COO format.

It is important to determine how to partition the original matrix. Based on empirical results, the fully-occupied ELL format is roughly three times faster than COO, except when the number of rows is approximately less than 4K. So it is profitable to add a K-th column to the ELL portion when the number of matrix rows with K (or more) non-zeros is at least max(4096, M/3), where M is the total number of matrix rows. The remaining non-zeros in the rows with more than K non-zeros are put into COO.

HYB is the fastest kernel in NVIDIA's SpMV library, but the HYB format needs to be improved further, which will be analyzed in section IV.

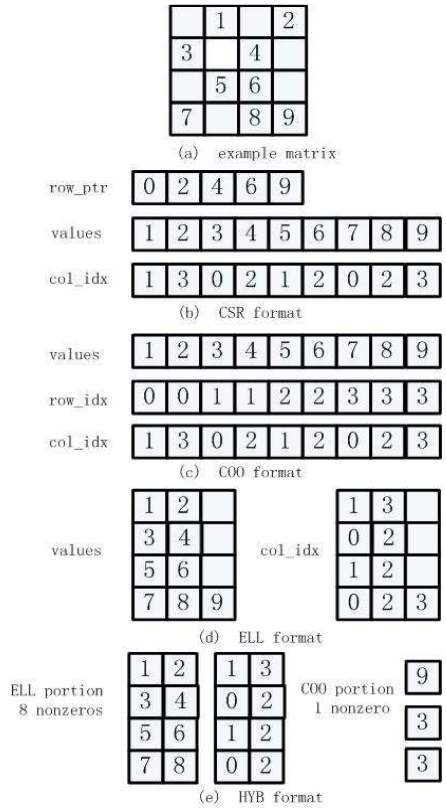


Fig. 1 Storage formats for an example matrix.

III. Related Work

There are a lot of researches on optimizing SpMV because of its importance. Many optimizing methods on various platforms have been proposed.

Eun-Jin Im [1] proposed two techniques, register blocking and cache blocking, to reuse the data in registers and cache respectively and reduce the traffic between on-chip and off-chip memory. A performance model was created to choose the block size in the register blocking method. A variation of basic SpMV in which a sparse matrix is multiplied by a set of dense vectors is also considered. Vuduc [2] proposed an improved heuristic for the tuning parameter of register blocking optimization, developed performance bounds for a specific matrix on a specific architecture and auto-tuned performance on single-core CPUs.

Williams [3] presented a performance model called Roofline Model to guide the performance auto-tuning for SpMV on multi-core platforms. Williams et al. [4] tested SpMV kernels with many kinds of optimizing methods on five multi-core/many-core platforms.

The research on GPU-based SpMV started in 2003[5]. Sengupta, et al. developed more generic approaches using parallel prefix/scan primitives [6], though this implementation did not outperform CPU. Bell and Garland [7] proposed parallel algorithms for several storage formats on GPU platform, including CSR, COO, ELL, HYB, DIA and PKT. HYB is the most closely related work to our research, in which the matrix is empirically partitioned into two portions, ELL and

COO. Baskaran and Bordawekar [8] optimized SpMV with four techniques on two GPU platforms. Ali Cevahir et al. [9] optimized SpMV with the jagged diagonal format (JAD) on GPUs. JAD is a more general format than DIA and ELL, but JAD can still guarantee coalesced memory access. But JAD kernel demands that the matrix should be preprocessed. ELLPACK-R[10] and Sliced ELLPACK[11] are two variants of ELL, which were proposed to save the storage space and reduce useless operations on zeros. Choi et al. [12] proposed BELLPACK format and used the register blocking method on GPUs. A model-driven auto-tuning framework was also proposed. Ping Guo, et al. [13] tuned CUDA parameters like BLOCK_SIZE, NUM_THREADS and WARP_SIZE for SpMV on GPUs to achieve higher performance. Xintian Yang, et al. [17] tried to use cache blocking method to optimize SpMV on GPU.

There are also some researches on new storage formats for sparse formats in recent years [14, 15, 16, 19].

IV. Fast SPMV Based On HYB-R Format

A. Observations:Problems of HYB Format

HYB format, which was introduced in section II, may not fully exploit the characteristics of some matrices. Two observations below show the problems of HYB format.

TABLE I
NON-ZEROS RATIO IN HYB FORMAT (%)

Matrices	COO	ELL
1	0.3	99.7
2	0.0	100.0
3	17.2	82.8
4	18.9	81.1
5	0.0	100.0
6	4.0	96.0
7	0.6	99.4
8	0.0	100.0
9	99.1	0.9
10	18.7	81.3
11	21.8	78.2
12	6.9	93.1
13	35.8	64.2

Observation 1: A large number of non-zeros may be left in the COO portion of the HYB format for some matrices. Table I shows that non-zeros left in the COO portion account for more than 17% of the whole matrix for 6 out of the 13 matrices when the partition method in HYB format is used. As a result, we can conclude that the performance of the COO portion can be improved further if we continue to partition the COO portion into two parts, ELL and COO.

Observation 2: The partition method used in HYB may not be the best choice. As mentioned in section II D, the number of columns in the ELL portion is K, which is determined empirically. But there are some other choices for K, such as average non-zeros per row.

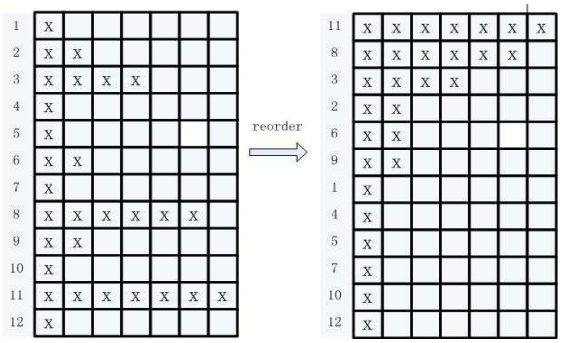
B. HYB-R Format and HYB-R Kernel

In this section, we represent a new storage format for sparse matrices, recursive hybrid (HYB-R) format. HYB-R format is proposed based on observation 1 in section IV A to put as many non-zeros as possible in the ELL portion. HYB-R kernel aims to fully exploit the advantage of ELL kernel on GPUs. Experimental results show that HYB-R kernel can improve the performance of HYB kernel further. HYB-R format and HYB-R kernel are represented below separately.

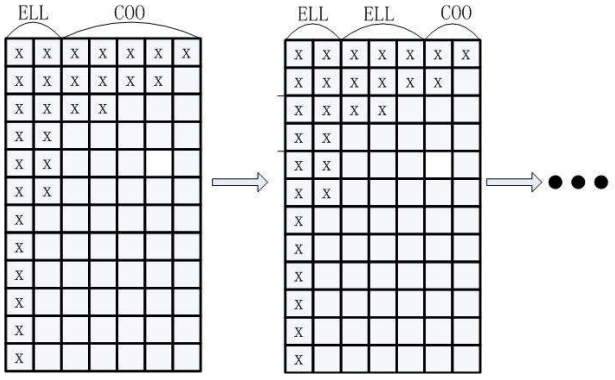
1. HYB-R Format

There are one COO portion and one ELL portion in HYB format, while there are one COO portion and more than one ELL portions in HYB-R format (Fig. 3 (a)). So the number of non-zeros in ELL format is usually larger for HYB-R format than HYB format.

The process of creating HYB-R format is described in Fig. 2(b). The matrix is first partitioned into two parts, ELL portion and COO portion. Then the COO portion is partitioned recursively into ELL portion and COO portion until there are few non-zeros left in COO portion or the number of rows left is less than 4096.



a) Reorder the matrix



b) The process of creating HYB-R format.

Fig. 2 Rorder the matrix and create HYB-R format

After several partitions, there are usually no non-zeros left for many rows which should not be stored in the next ELL portion to save the memory space. As a result, the matrix rows are reordered in decreasing order of length before the process of creating the HYB-R format if the matrix will be partitioned at least twice. For example, only Row 11, 8 and 3 are stored in the second ELL portion after reordering in Fig. 2 (a), but all the rows should be stored in the second ELL portion if the matrix rows are not reordered previously so that each matrix row corresponds with the right element of vector y. The reorder information of the matrix rows is recorded in a permutation

array. After the HYB-R kernel finishes, vector y is reordered according to the permutation array.

2. HYB-R Kernel

The HYB-R kernel is divided into several ELL kernels and one COO kernel, which are launched one by one. Fig. 3 (a) shows the pseudo-code for HYB-R data structure, and Fig. 3 (b) shows the execution process of HYB-R kernel. The matrix in HYB-R format and the vectors are first transferred to the global memory, then the kernels are started one by one, and at last the vector y is copied from the global memory to the host memory. The number of threads in the ELL kernels becomes smaller gradually because the number of matrix rows left becomes smaller.

```
struct hyb-r_matrix{
  ell_matrix    ell1;
  ell_matrix    ell2;
            …
  coo_matrix    coo;
};
```

(a) HYB-R data structure

```
ELL kernel1
ELL kernel2
…
COO kernel
```

(b) HYB-R kernel

Fig. 3 HYB-R data structure and HYB-R kernel

C. Parameter Tuning

First, we should decide when to stop the partition process. In other words, when the number of non-zeros in COO portion is less than X% of the number of non-zeros in the whole matrix, the process of partitioning the matrix should be stopped. In the implementation, the partition process is stopped when X=0.5 or the number of rows left is less than 4096, but X can be carefully tuned to get better performance.

Second, as mentioned in observation 2 in section IV A, we should decide how to choose the parameter K, which is used to partition the matrix into ELL portion and COO portion. Besides the method mentioned in Section II D (Method 1), there are two other methods to determine the value of K as follows.

1. K equals average non-zeros per row (Method 2). This is a simple method but it can be effective when the numbers of non-zeros per row do not highly vary across the matrix.
2. If the non-zeros account for more than 1/3 storage space of ELL portion after the column is added to the ELL portion, the column should be added (Method 3). Method 3 characterizes the non-zeros ratio in the ELL portion more exactly than Method 1.

V. Experiments Setup

A. Introduction to NVIDIA CUDA

A NVIDIA GPU usually consists of several streaming multiprocessors, and each streaming multiprocessors consists of eight streaming processors. There is a private local memory in the form of registers for each thread and a low-latency on-chip memory called the shared memory for a group of threads. The main memory of a GPU is a high-bandwidth DRAM shared by all threads. There are also two types of read-only memory called constant memory and texture memory both with on-chip cache.

NVIDIA's CUDA is a programming model designed for NVIDIA GPUs. A CUDA program consists of a host program running on the CPU, and a kernel program running on the GPU. The host program transfers the data from CPU to GPU, the kernel program processes that data, and then the host program transfers the results from GPU to CPU. The kernel program is partitioned into a grid of thread blocks, each including a group of threads. A warp is a group of 32 threads, and a thread block may include several warps. The execution of the threads follows a single instruction multiple threads (SIMT) model. Threads within a block share the shared memory and can synchronize via barriers. But there is no such synchronization mechanism for threads in different thread blocks.

There are many techniques to improve performance on GPUs. We just list three of them and suggest the CUDA Programming Guide [18] for more information. (1) Maximize the bandwidth of global memory with aligned and coalesced accesses. (2) Reuse data in on-chip memories, such as registers, shared memory, texture cache and constant cache. (3) Reduce thread divergence.

TABLE II
Platforms Used In The Experiments

GPU	GeForce 9800 GX2	GeForce GTX 295
CUDA Cores	256 (128 per GPU)	480 (240 per GPU)
GPU clock	1500 MHz	1242 MHz
Memory Clock	1000MHz	999 MHz
Memory capacity	1GB (512MB per GPU)	1792 MB (896MB per GPU)
Memory Bandwidth	128 (64 per GPU) GB/sec	223.8 GB/sec
Compute Capability	1.1	1.3

B. Experiment Platform

The experiments in this paper were run on the two systems listed in Table II. There are two GPUs on a single card for both GPU platforms, but only one GPU was used in the experiments.

C. Sparse Matrices in the Experiments

The sparse matrices used in the experiments are listed in Table III, which are also used in prior work [7]. Every matrix in the table is given a serial number so that we can use the number to represent the matrix. The column Rows and Columns shows the number of rows and columns for each matrix. The column NNZ represents the total number of non-zeros for each matrix. The column NNZ/R represents the number of non-zeros per row on average.

TABLE III
MATRICES USED IN THE EXPERIMENTS

No.	Matrix Name	Rows	Columns	NNZ	NNZ/R
1	FEM/Cantilever	62,451	62,451	4,007,383	64.1
2	FEM/Spheres	83,334	83,334	6,010,480	72.1
3	FEM/Accelerator	121,192	121,192	2,624,331	21.6
4	Economics	206,500	206,500	1,273,389	6.1
5	Epidemiology	525,825	525,825	2,100,225	3.9
6	Protein	36,417	36,417	4,344,765	119.3
7	Wind Tunnel	217,918	217,918	11,634,424	53.3
8	QCD	49,152	49,152	1,916,928	39.0
9	LP	4,284	1,092,610	11,279,748	2632.9
10	FEM/Harbor	46,835	46,835	2,374,001	50.6
11	Circuit	170,998	170,998	958,936	5.6
12	FEM/Ship	140,874	140,874	7,813,404	55.4
13	Webbase	1,000,005	1,000,005	3,105,536	3.1

VI. EXPERIMENTAL RESULTS AND ANALYSES

The experiments were carried out with the matrices and the platforms which were introduced in section III.

TABLE IV
NON-ZEROS RATIO IN HYB-R FORMAT AFTER THE MATRIX IS PARTITIONED TWICE

Matrices	ELL1	ELL2	COO
1	99.7	0.0	0.3
2	100.0	0.0	0.0
3	82.8	15.2	2.0
4	81.1	9.7	9.2
5	100.0	0.0	0.0
6	96.0	3.2	0.8
7	99.4	0.5	0.1
8	100.0	0.0	0.0
9	0.9	0.0	99.1
10	81.3	16.2	2.5
11	78.2	12.1	9.7
12	93.1	5.9	1.0
13	64.2	8.2	27.6

In the experiments, the matrices were only partitioned twice (Table IV) for simplicity. But we can conclude that performance will be better for some matrices if the matrices are partitioned three or more times, because there are still a lot of non-zeros left in COO portion for some matrices after two partitions, such as Matrix 4, 11 and 13, all with more than 9% of the non-zeros in the COO portion.

As shown in Table I and IV, non-zeros of COO portion are less than 0.5% of non-zeros in the whole matrix for Matrix 1, 2, 5 and 8, which are partitioned only one time, so there are no non-zeros in ELL2 portion in Table IV for these matrices. There are only 4284 rows in Matrix 9, and less than 4096 rows are left after the first partition, so there is also no non-zero in its ELL2 portion. As a result, the HYB-R kernel is approximately as fast as the HYB kernel for the above five matrices (Fig. 4 and 5).

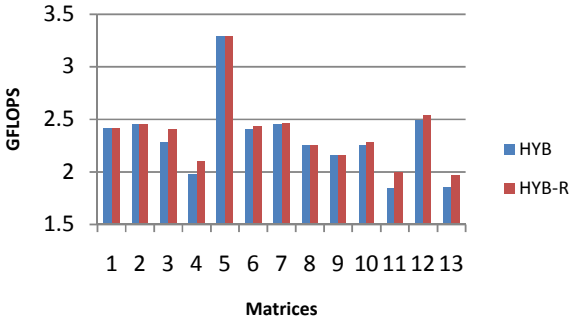


Fig. 4 Single precision performance without cache on GeForce 9800 GX2

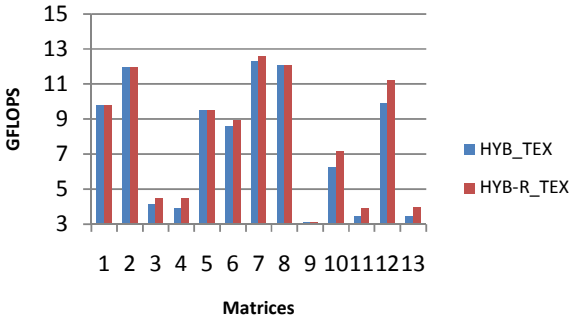


Fig. 5 Single precision performance with cache on GeForce 9800 GX2.

Vector x can be put in texture memory so that some elements of x will be reused in texture cache by different threads. Fig. 4 shows the single precision performance without cache(x was placed in global memory) on GeForce 9800 GX2, and Fig. 5 shows the single precision performance with cache(x was placed in texture memory) on GeForce 9800 GX2. Both HYB kernel and HYB-R kernel can achieve higher performance with cache than without cache. From Fig. 4 and 5, we can find that the HYB-R kernel is faster than the HYB kernel for all the eight matrices which are partitioned twice. If there are enough non-zeros and enough rows in COO portion, the performance can always be further improved by another partition to the COO portion.

Table V shows the speedup of HYB-R kernel over HYB kernel on GeForce 9800 GX2. HYB-R kernel outperformed HYB kernel obviously both with cache and without cache. In HYB-R kernel with cache, 5 out of the 13 matrices achieved more than 13% speedup over the HYB kernel. For the best case, Matrix 12, performance was improved by 14.5% with cache. The average speedups of the eight matrices which are partitioned twice are 3.9% without cache and 10.4% with cache respectively.

More performance improvement can be achieved with cache than without cache. The possible reason is that ELL2 kernel can reuse vector x in texture cache better than ELL1 kernel. ELL2 kernel only accesses a part of the vector x and ELL1

kernel may access the whole vector x, while the texture cache cannot hold the whole vector x. From this point of view, carefully tuning the parameter K may also lead to better reuse of vector x.

TABLE V
PERFORMANCE IMPROVEMENT OF HYB-R OVER HYB (%) ON GEFORCE 9800 GX2

Matrices	without cache	with cache
1	0	0
2	0	0
3	5.7	7.7
4	6.0	13.8
5	0	0
6	1.2	4.0
7	0.4	2.5
8	0	0
9	0	0
10	0.9	14.4
11	8.7	13.3
12	2.0	13.0
13	5.9	14.5
Average Speedup	3.9	10.4

Fig. 6 shows the bandwidth of HYB kernel and HYB-R kernel without cache on GeForce 9800 GX2, and Fig. 7 with cache. More bandwidth improvement of HYB-R over HYB can also be achieved with cache than without cache, just as the case of single precision performance.

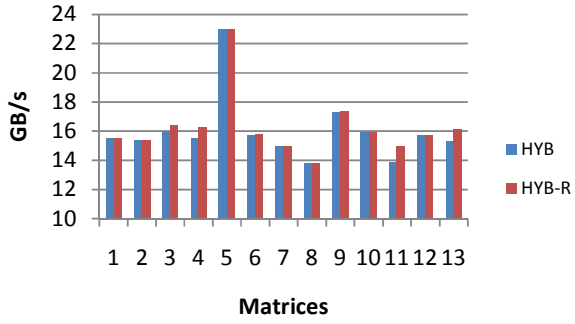


Fig. 6 Bandwidth without cache on GeForce 9800 GX2

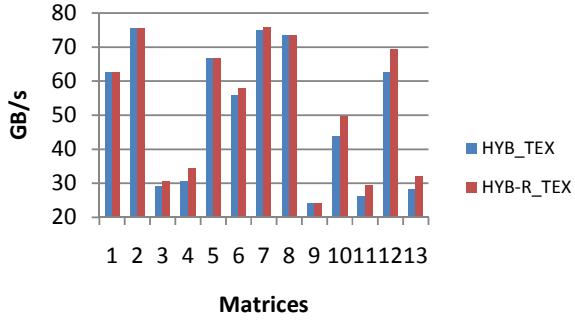


Fig. 7 Bandwidth with cache on GeForce 9800 GX2

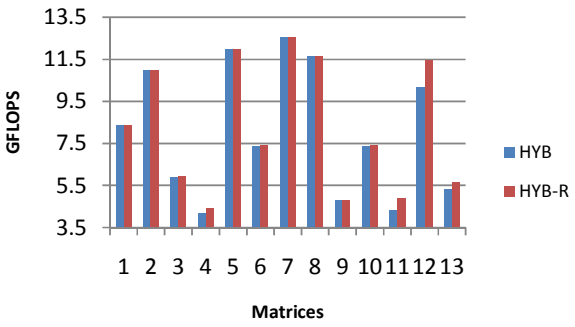


Fig. 8 Single precision performance without cache on GeForce GTX 295

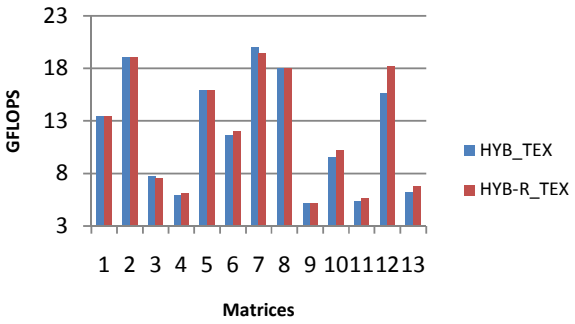


Figure 9 Single precision performance with cache on GeForce GTX 295

Fig. 8 shows the single precision performance without cache(x in global memory) on GeForce GTX 295. Fig. 9 shows the single precision performance with cache(x in texture memory) on GeForce GTX 295. The HYB-R kernel is also as fast as the HYB kernel for Matrix 1,2,5,8 and 9 with the same reason as on GeForce 9800 GX2. The performance of the HYB-R kernel is higher than the HYB kernel without using texture cache for all the other eight matrices. It is surprising that the HYB-R kernel is slower than the HYB kernel for Matrix 3 and 7 with texture cache. The possible reason is that one more ELL kernel is launched in HYB-R kernel than in HYB kernel, and the cost of launching one more ELL kernel can lead to lower performance on GeForce GTX 295.

TABLE VI
PERFORMANCE IMPROVEMENT OF HYB-R OVER HYB (%) ON GEFORCE GTX 295

Matrices	without cache	with cache
1	0	0
2	0	0
3	0.5	-2.5
4	5.5	3.7
5	0	0
6	0.9	3.1
7	0.2	-2.6
8	0	0
9	0	0
10	0.4	7.5
11	13.5	4.5
12	12.7	16.5
13	6.0	9.2
Average Speedup	5.0	4.9

Table VI shows the speedup of HYB-R kernel over HYB kernel on GeForce GTX 295. For the best case, Matrix 12, performance was improved by 16.5% with cache. The average speedup of the eight matrices is 5.0% without cache, and 4.9% with cache. Minus signs are used for Matrix 3 and 7 because of the lower performance. Combining Table V with Table VI, we can find that higher speedup was achieved without cache on GeForce GTX 295, 5.0% versus 3.9%, while higher speedup is achieved with cache on GeForce 9800 GX2, 10.4% versus 4.9%.

Fig. 10 shows the bandwidth of HYB kernel and HYB-R kernel without cache on GeForce GTX 295, and Fig. 11 with cache.

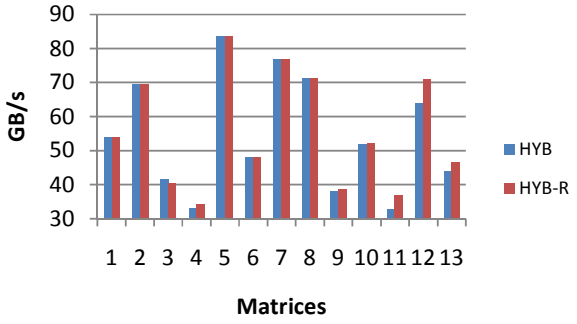


Fig. 10 Bandwidth without cache on GeForce GTX 295

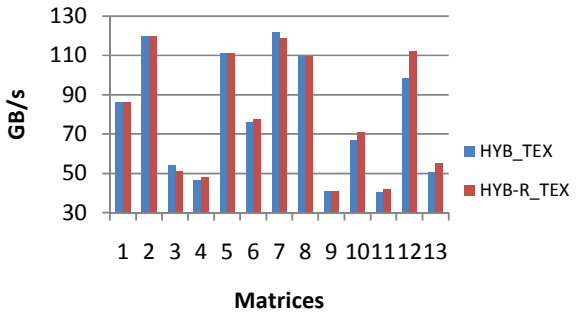


Fig. 11 Bandwidth with cache on GeForce GTX 295

Table VII shows the parameter tuning results of two methods and the best value of K which was found exhaustively. In this experiment, each matrix was partitioned only once, and only the GeForce 9800 GX2 platform was used. Method 1 is the method used in HYB format. Method 2 is the average non-zeros per row method. Because average non-zeros per row is not a decimal, the value of K in Method 2 includes a smaller integer and a greater integer. Each entry of the table includes the single precision performance with cache and the value of K (in parenthesis). The best performances of the three methods are shown in red. If the best performance was achieved by Method 1 or 2, the content of Best column would not be in red.

TABLE VII
PERFORMANCE OF TUNING K - GFLOPS(K)

Matrices	Method 1	Method 2		Best
1	9.75 (75)	7.97(64)	8.15(65)	9.75 (75)
2	11.97 (81)	8.59(72)	8.47(73)	11.97 (81)
3	4.15 (23)	3.15(21)	3.99(22)	4.15 (23)
4	3.91 (7)	3.77(6)	3.91(7)	4.31 (9)
5	9.52 (4)	3.02(3)	9.51(4)	9.52 (4)
6	8.60 (138)	8.00(119)	8.41(120)	8.80 (140)
7	12.27 (54)	10.43(53)	12.27(54)	12.27 (54)
8	12.08 (39)	12.08(39)	11.93(40)	12.08 (39)
9	3.12 (23)	1.39(2632)	1.39(2633)	3.12 (23)
10	6.24 (55)	5.92(50)	6.07(51)	6.24 (55)
11	3.45 (5)	3.46(5)	3.76(6)	3.76 (6)
12	9.93 (54)	9.92(55)	9.61(56)	9.93 (54)
13	3.44 (2)	3.61(3)	3.69(4)	3.69 (4)

It can be found that neither method is optimal, 9 matrices using Method 1 get best performances, 2 matrices using Method 2. Method 1 seems better than Method 2 for the 13 matrices. But the value of K still needs to be better tuned to achieve higher performance, which is mainly determined by the characteristics of the specific matrix and the GPU architecture. Maybe the Method 3 represented in section IV C is a better choice, which is not included in the experiment.

VII. Conclusions And Future Work

In this paper, a novel method is proposed to improve the performance of SpMV on GPUs. A new storage format called HYB-R is proposed to fully exploit GPU architecture. Experiments were carried out with thirteen matrices on two GPU platforms. Experimental results show that our method can get better performance than the fastest kernel (HYB) in NVIDIA's SpMV library. We also tried to tune the parameters on how to partition the matrix into ELL and COO, and find that the partition method used in HYB format is better than the non-zeros per row method for the 9 out of13 matrices.

However, the partition method used in HYB format is not perfect. So we will test other methods to partition the matrix into ELL and COO for the HYB-R format, considering the best reuse of vector x in texture cache. And we will also try cache blocking method with different storage formats to represent sub-blocks on GPUs in future.

Acknowledgment

This work is in part supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302500, State Key Program of National Natural Science Foundation of China under Grant No. 60736012, National Science Foundation for Distinguished Young Scholars of China under Grant No. 60925009, Foundation for Innovative Research Groups of the National Natural Science Foundation of China under Grant No. 60921002, Beijing Natural Science Foundation under Grant No.4092044, National Core-High Tech-basic Program under Grant No.2011ZX01028-001-002.

References

[1] E.-J. Im, "Optimizing the performance of sparse matrix-vector multiplication", PhD thesis, University of California, Berkeley, May 2000.

[2] R. Vuduc, " Automatic Performance Tuning of Sparse Matrix Kernels", PhD thesis, University of California, Berkeley, December 2003.
[3] S. Williams , "Auto-tuning Performance on Multicore Computers", PhD thesis, University of California, Berkeley, 2008.
[4] Sam Williams, Richard Vuduc, Leonid Oliker, John Shalf, Katherine Yelick, and James Demmel, "Optimizing sparse matrix-vector multiply on emerging multicore platforms," Journal of Parallel Computing, vol. 35, no. 3, pp.178–194, March 2009.
[5] Jeff Bolz, Ian Farmer, et al. , "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," In Proc. Special Interest Group on Graphics Conf. (SIGGRAPH), San Diego, CA, USA, July 2003.
[6] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, "Scan primitives for GPU computing," In Proc. ACM Dense Protein QCD Cantilever Spheres Harbor Ship Wind Tunnel SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware, San Diego, CA, USA, 2007.
[7] Nathan Bell and Michael Garland, "Efficient sparse matrix-vector multiplication on CUDA," In Proc. ACM/IEEE Conf. Supercomputing (SC), Portland, OR, USA, November 2009.
[8] Muthu Manikandan Baskaran and Rajesh Bordawekar, "Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies," Technical Report RC24704 (W0812-047), IBM T.J.Watson Research Center, Yorktown Heights, NY, USA, December 2008.
[9] Ali Cevahir , Akira Nukada , Satoshi Matsuoka, "Fast Conjugate Gradients with Multiple GPUs," Proceedings of the 9th International Conference on Computational Science: Part I, Baton Rouge, LA, May 25-27, 2009.
[10] F. Vazquez, E. M. Garzon, J.A.Martinez, and J.J.Fernandez, "The sparse matrix vector product on GPUs," Technical report, University of Almeria, June 2009.
[11] Monakov, A., A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," High Performance Embedded Architectures and Compilers, Lecture Notes in Computer Science, Vol. 5952, pp.111–125, 2010.
[12] J. W. Choi, A. Singh, and R. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," In PPOPP, pp.115–126, 2010.
[13] Ping Guo, Liqiang Wang, "Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs," The 2010 International Conference on Computational and Information Sciences, Chengdu, China.
[14] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and E. Leiserson, "Parallel sparse matrix-vector and matrixtranspose-vector multiplication using compressed sparse blocks," In SPAA, pp. 233–244, 2009.
[15] A. Buluç, et.al., "Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication," In IPDPS, 2011.
[16] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An Extended Compression Format for SpMV on Shared Memory Systems,"In PPoPP, pp. 247–256, San Antonio, Texas, USA, 2011.
[17] Xintian Yang, Srinivasan Parthasarathy, P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining," In VLDB, 2011.
[18] NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide, Version 2.1, December 2008.
[19] J. Willcock, A. Lumsdaine, "Accelerating sparse matrix computations via data compression," In ICS, Cairns, Australia, June 2006.