2012 IEEE 14th International Conference on High Performance Computing and Communications

An Efficient Sparse Matrix Multiplication for skewed matrix on GPU

Monika Shah
Computer Science & Engineering Department
Nirma University
monikag.shah@gmail.com

Vibha Patel
Department of Computer Science & Engineering
Indian Institute of Technology
vibhadp@cse.iitk.ac.in

***Abstract*—This paper presents a new sparse matrix format *ALIGNED_COO*, an extension to *COO* format to optimize performance of large sparse matrix having skewed distribution of non-zero elements. Load balancing, alignment and synchronization free distribution of work load are three important factors to improve performance of sparse matrices representing power-law graph. Coordinate (*COO*) format is selected for extension in this paper as it is the most suitable format for sparse matrices representing power-law graph. The *ALIGNED_COO* format tries to set maximum alignment across the computing resources. Our heuristic to decide degree of concurrency is different from the existing approaches. Dispite the availability of other popular sparse formats, *ALIGNED_COO* format helps to gain better performance without any extra memory overhead. Our approach not only achieves higher performance on skewed matrices with power-law distribution, but also gives appreciable performance for wide range of sparse matrix patterns. The proposed implementation of SpMV kernel for *ALIGNED_COO* sparse format helps to achieve 1.0-25.72 times higher performance than *COO_flat* kernel with increase in the level of accuracy. The average performance gain over other sparse formats is in tolerable range of 0.89-48.8.**

Keywords: *SpMV*, Power-law graph, Load balance

I. Introduction

Sparse matrix Vector multiplication (SpMV) is a very prominent and highly used kernel in scientific and engineering applications. Many of these applications use sparse matrix having power-law distribution. Optimizing performance of such matrix on NVIDIA GPU is a challenging task. This has motivated us to design a data structure which can be useful to increase performance for such matrices. There are two key advantages of our proposed data structure *ALIGNED_COO* and its respective SpMV implementation over past effort: (i) It achieves higher performance for matrices with skewed distribution of non-zeros. (ii) It also provides acceptable performance for wide range of sparse matrix patterns.

Performance of SpMV kernel on GPU using the formats presented by BELL et al.[1] is limited due to three main reasons: (i) All these formats are applicable in specific sparse matrix pattern. For example, Coordinate (COO) format is suitable for sparse matrix having large power-law distribution, CSR and ELL formats are suitable for sparse matrix which have rows with similar row length and DIA format is suitable for diagonal matrix pattern only. (ii) Lack of an important factor of equal load distribution to reduce threads scheduling overhead. (iii) Existence of update lost problem due to lack of synchronization.

We propose following essential factors responsible for improvement in the performance of SpMV kernel on GPU.

i) Load balancing among computational resources
ii) Synchronization free load distribution
iii) Increasing reuse of the input vector
iv) Heuristic to work with wide range of sparse pattern
v) Reduce fetch operation to avoid drawback of low latency memory access in GPU

These factors are influenced by the past research on matrices following power-law distribution. Considering these factors, we have proposed a new storage format ALIGNED_COO for sparse matrix and its respective SpMV implementation to optimize performance on GPU. The remaining paper is structured as follows: Section II deals with CUDA programming principles in brief. The trajectory of Optimizing sparse format and its SpMV implementation is traced in section III. Section IV brings forth our attempt to optimize SpMV performance on NVIDIA GPU. Section V exhibits the performance evaluation of our work and speedup comparison with well-known sparse formats. Conclusion of the paper along with furture work is described in section VI.

II. Parallel Programming using CUDA

Compute Unified Device Architecture (CUDA) is a programming model and the APIs required for performing general purpose computation tasks on NVIDIA GPUs. GPUs are highly multithreaded architectures that provide high order of magnitude of speedup for regular applications with high computation requirements. CUDA program is a sequential program that runs program module (kernel) in parallel devices. CUDA kernel executes on a grid of thread blocks, where a thread block is a group of threads that work in SIMT (Single Instruction Multiple Thread) fashion.

The SIMT execution model is designed to work efficiently on regular application, where each independent computation on independent data can be executed in parallel to reduce time complexity from $O(n)$ to $O(1)$ with n parallel computation

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.192

resources. Kernel designed to handle data dependency with control statements may produce thread divergence. These thread divergences cause serialization of threads with different control path which attribute towards increase in execution time.

We have used the following programming principles in CUDA to achieve higher performance:

- Maximize Thread concurrency
- Synchronization free data distribution among concurrent thread to achieve accurate result
- Data reordering to obtain coalesced memory access
- Data reordering and use of shared memory access to overcome drawback of low latency memory access on GPU
- Avoiding thread divergence to improve performance

III. Related Work

BELL et al. [1] have presented various compressed sparse storage formats, and its respective SpMV kernel implementation for CUDA environment. These compressed sparse storage formats include Coordinate (COO) format, Compressed Sparse Row format (CSR), ELLPACK (ELL) format, Diagonal (DIA) format. CSR(vector) and COO_flat kernels in [1] provide higher concurrency through vectorization. However, the inherent problem of vectorization leads to (i) Splitting the row across warp boundary causing update loss problem, and (ii) Increased overhead of parallel reduction for insufficient data to be processed by a warp.

In last few years, researchers and academicians have presented many more optimization of SpMV on GPU. [2],[3] presented data reordering methods to reduce memory space for sparse matrix, and to increase reuse of input vector. [4] eliminate use of zero padding in BCSR format, but it has drawback of using additional indirection index and thread divergence issues. [5] describe method of splitting rows into groups and allocation of appropriate warp size to increase performance. Dziekonski [6] describe method of splitting large rows in ELL format to equalize load distribution among threads. But, applying the concept of splitting large rows will not suffice the requirement of achieving high degree of load balancing. [7] explore splitting of block rows to equalize load among active threads. This past effort to split data computations into multiple kernel has resulted in excess overhead of kernel scheduling.

Sadayappan et al.[3] have explored an efficient transformation of sparse matrix storage format to maximize load balancing. [8] emphasize on importance of load balancing among threads for optimizing SpMV performance. [9] have shown use of Travelling salesman problem in data reordering to optimize storage space and performance. Data reordering and computation reordering are two powerful techniques to optimize performance of sparse applications. But complex data reordering reduces performance in absence of recurrent use of SpMV kernel.

IV. Proposed Work

In this section, we have described our proposed sparse storage format ALIGNED_COO for skewed sparse matrix and its respective SpMV kernel implementation. The proposed format is an extension of COO format with simplified data reordering transformation. We have considered important factors listed in section I, which are responsible for higher performance of SpMV kernel on GPU. The detailed description of these factors is as follows.

A. Load balancing among computational resources

We performed load balancing to equally distribute computational tasks among active threads of streaming multiprocessors on GPU. One of the prime reason to introduce load balancing is to reduce thread scheduling overheads without compromising the performance. In order to balance distribution of computational task, we create equal size segments containing elements for concurrent computation. Segments are executed in sequential order similar to CSR and ELLPACK format. Each element of segments is executed in parallel to achieve synchronization free concurrent execution. The process of segmentation

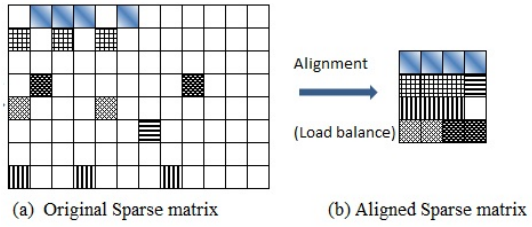

Fig. 1. Alignment using data reordering

ensures that the segment contains only one element of each row for synchronization free concurrent execution. Each column shown in fig.1.(b) represents a segment. Proper selection of segment size and number of segments helps to achieve high degree of concurrency and robust load balancing. The applied approach of determining segment size and number of segments in the proposed format is described in Algorithm 1. To support coalesced memory access, ALIGNED_COO format stores data in column major order. Fig. 2(b) demonstrates segmented ALIGNED_COO storage representation for matrix shown in fig. 2(a). The Aligned_COO data structure consists of three vectors I, J, and V as similar to COO format. Algorithm 1 returns 4 as the segment size, and 3 as the number of segments for the given matrix. Character 'p' in vector represents padding value. 0 is considered as default padding value.

Algorithm 1 Determining Segment Size

Input : NNZ, NR, *Row_len*
Output: Seg_size,Num_of_segs

$Max\_row\_length = 0$, $Avg\_length\_rows = 0$
$Avg\_length = NNZ/NR$
for $i = 0 \rightarrow NR$ **do**
  $Max\_row\_length = max(Max\_row\_length, Row\_len[i])$
  if $Row\_len[i] \geq Avg\_length$ **then**
    $Avg\_length\_rows+ = 1$
  end if
end for
$Num\_of\_seg = Max\_row\_len$
$Approx\_seg\_size = NNZ/Num\_of\_seg$
$Seg\_size = max(Approx\_seg\_size, Avg\_length\_rows)$
while $Row\_len[Seg\_size] + Row\_len[Seg\_size + 1] \geq Num\_of\_segs$ AND $Seg\_size < NR$ **do**
  $Seg\_size = Seg\_size + 1$
end while
return

$$\begin{bmatrix} 0 & 2 & 0 & 4 & 5 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 6 & 0 & 0 \\ 0 & 7 & 0 & 8 & 0 \\ 0 & 0 & 0 & 9 & 0 \\ 10 & 0 & 0 & 0 & 0 \end{bmatrix}$$

a) Matrix A

```
Row Index Vector    I:{0,1,3,5,0,2,3,p,0,2,4,p}
Column Index Vector J:{1,0,1,0,3,1,3,p,4,2,3,p}
Data Vector         V:{2,1,7,10,4,3,8,p,5,6,9,p}
Input Vector        X:{1,2,3,4,5}
```

b) Aligned_COO format and Input Vector X

```
Row Index     vector I:{0,1,3,5,0,2,3,0,0,2,4,0}
Column Index  vector J:{2,1,2,1,4,2,4,1,5,3,4,1}
Data          vector V:{2,1,7,10,4,3,8,0,5,6,9,0}
Input         vector X:{1,2,3,4,5}
```

c) Modified J in Aligned_COO

Fig. 2. Modified COO storage presentation to avoid use of Input Vector

B. Synchronization free load distribution

We have divided the task of synchronization free load distribution in two phases. The first phase is concerned with constraining distribution of row elements into different segments as mentioned in section IV-A. The second phase is concerned with proper padding value to create equal size segments. We use the following concept to ensure proper selection of padding values.

i) Synchronization-free execution of segments, where no two segments can have same row index. For example, as shown in fig.3, use of 0 padding value in segment 1 of I vector raise the need of synchronization logic. Considering this criteria, values 1 and 4 are possible padding options for segment 1, and values 1,3,5 are possible padding values for segment 2.
ii) Avoiding additional memory fetch operation for padding value.
iii) Avoiding memory conflicts. In the given example, 4 and 5 are selected as padding values for segment 1 and segment 2 respectively as shown in fig.3.

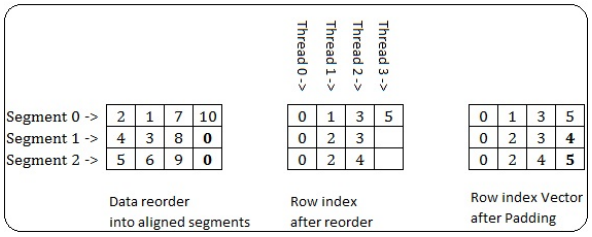

Fig. 3. Modified Aligned_COO format to avoid use of Input Vector

Algorithm 2 Aligned COO segment kernel

Input : Input Vector(*X*), Data Vector (*V*)
Row Index Vector(*I*),seg_size, num_seg, stride, min_rl
Output: Output Vector(*Y*)

$idx = calculate\_thread\_id()$
if $idx \geq seg\_size$ **then**
  return
end if

$row\_id = I[idx]$
$i = idx, sum = 0, cnt = 0$
while $(cnt < min\_rl)$ **do**
  $sum+ = V[i] * X[i]$
  $cnt+ = 1$
  $i = i + stride$
end while
$y[row\_id]+ = sum$

while $idx < seg\_size$ **do**
  while $i < mem\_size$ **do**
    $rowi\_d = I[i]$
    $Y[row\_id] = Y[row\_id] + V[i] * X[i]$
    $i = i + stride$
  end while
  $idx = idx + MAX\_THREADS$
  $i = idx$
end while
return

C. Increasing reuse of the input vector and reducing fetch operations

Most optimization associated with SpMV talks about reducing storage space, and thread allocation. [2],[3] presented concept of column reordering to improve reuse of input vector. Wang

et al.[10] have applied an approach of substituting column array with actual element of input vector X corresponding to the column index. This helps to achieve reduction in fetch operation for input vector. We have applied both these concepts in the proposed format, which is demonstrated in fig. 2(c). The ALIGNED_COO storage format shown in fig. 2(b) is modified by substituting column array J with actual elements of input vector X for corresponding J index.

D. Heuristic to work with wide range of sparse pattern

COO format is suitable for a large sparse matrix with skewed distribution of non-zeros[1]. The proposed SpMV implementation is a hybridization of SpMV_ALIGNED_COO_seg kernel and SPMV_ALIGNED_COO_flat kernels. This heuristic is designed for distributing computations having requirements of different concurrency degree. The rows with average length less than warp size will use SpMV_ALIGNED_COO_seg kernel, and the remaining rows having higher concurrency need will use COO_flat kernel for computation as shown in fig.4. Further, we also modified the COO_flat kernel as SPMV_ALIGNED_COO_flat to reduce the error rate, for cases when the warp access, crosses the row boundary. To eliminate this drawback of COO_flat kernel, trailing elements of each row which are not in multiple of warpsize are assigned to SpMV_ALIGNED_COO_seg portion of AlIGNED_COO. The implementation of SpMV_ALIGNED_COO_seg kernel is described in Algorithm 2.

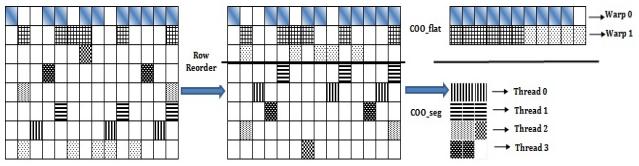

Fig. 4. ALIGNED_COO Hybrid kernel

Runtime cost estimation for COO_flat partition, and COO_seg partition are used in our heuristic to perform thread allocation. Computation of runtime cost for ALIGNED_COO_seg, and ALIGNED_COO_flat kernel is carried out using following system of equations:

Runtime cost for COO_flat kernel

$$threads\_required = NNZ$$
$$active\_threads = min(MAX\_THREADS, threads\_required)$$
$$warps = \lceil \frac{threads\_required}{WARP\_SIZE} \rceil$$
$$iteration = \lceil \frac{threads\_required}{active\_threads} \rceil$$
$$thread\_cost = log_2 WARP\_SIZE$$
$$serial\_cost = NNZ \mod WARP\_SIZE$$
$$blocks\_reduction\_cost = log_2(blocksize)$$
$$T(COO\_flat) = iteration * thread\_cost + serial\_cost + blocks * blocks\_reduction\_cost + warps * WARP\_SCHEDULE\_TIME$$

Runtime cost for Aligned_COO_seg kernel

$$threads\_required = segment\_size$$
$$active\_threads = max(threads\_required, MAX\_THREADS)$$
$$warps = \lceil \frac{active\_threads}{WARP\_SIZE} \rceil$$
$$iteration = \lceil \frac{threads\_required}{active\_thread} \rceil$$
$$T(Aligned\_COO\_seg) = iteration * segments + warps * WARP\_SCHD\_TIME$$

Runtime cost for partition COO_flat, Aligned_COO_flat and Aligned_COO_seg are used in heuristic to construct an efficient format and its respective kernel implementation. Detailed description of the applied heuristic is shown in Algorithm 3.

Algorithm 3 Aligned COO Conversion

Input : NNZ, NR
Output: Segment_size
for $i = 1 \rightarrow NR$ **do**
  $rl[i] = find\_row\_length(i)$
  $trail\_elements = rl[i] \ mod \ WARP\_SIZE$
  $flat\_elements = rl[i] - trail\_lelements$

  move $trail\_elements$ to aligned_coo_seg part
  move $flat\_elements$ to aligned_coo_flat part

  $aligned\_coo\_seg\_nz \ += trail\_elements$
  $aligned\_coo\_seg\_rows \ += 1$

  $aligned\_coo\_flat\_rows += 1$
  $aligned\_coo\_flat\_nz \ += flat\_elements$
end for
$coo\_seg\_part\_cost$ = Find T($aligned\_coo\_seg$ part)
$coo\_flat\_part\_cost$ = Find T($aligned\_coo\_flat$ part)
$complete\_coo\_seg\_cost$= Find T($aligned\_coo\_seg$) for complete data set
$complete\_coo\_flat\_cost$ = Find T(coo_flat) for complete data set
$aligned\_coo\_hyb\_cost = coo\_seg\_part\_cost + coo\_flat\_part\_cost$
if ($aligned\_coo\_hyb\_cost \geq complete\_coo\_seg\_cost$ AND $aligned\_coo\_hyb\_cost \geq complete\_coo\_flat\_cost$) **then**
  copy entire matrix data to $aligned\_coo\_seg$ part
end if
if ($complete\_coo\_flat\_cost \geq complete\_coo\_seg\_cost$ AND $complete\_coo\_flat\_cost \geq aligned\_coo\_hyb\_cost$) **then**
  copy entire matrix data to coo_flat part
end if
return

V. Experimental Result

We have tested performance of our proposed implementation on dataset listed in table 1. The matrices selected for experimentation are having varied sparse pattern. We have designed our sparse format and its implementation and added it to NVIDIA cusp-library for evaluating the performance against well-known formats provided in NVIDIA cusp-library.

A. Test Platform

We performed our experiments on Intel(R) Core(TM) i3 CPU @ 3.20GHz with 4GB RAM, 2 x 256KB(L2 Cache) and 4 MB (L3 Cache), and NVIDIA C2070 GPU device using CUDA version 4.0 on Ubuntu 11.

TABLE I
Sparse Matrix collection used in experimentation

Matrix	NR	NC	NNZ	%NNZ
G2_circuit	1,50,102	1,50,102	7,26,674	0.00003225
3D_51448_3D	51,4484	51,4484	1,056,610	0.00003992
lung2	1,09,460	1,09,460	4,92,564	0.00004111
dc1	1,16,835	1,16,835	7,66,396	0.00005614
debr_G_14	65,536	65,536	2,62,138	0.00006103
bayer01	57,735	57,735	2,77,774	0.00008333
aug3dcqp	35,543	35,543	1,28,115	0.00010141
bloweybl	30,003	30,003	1,20,000	0.00013331
bips07_3078_iv	21,128	21,128	75,729	0.00016965
c64b	51,035	51,035	7,17,841	0.00027561
sit100	10,262	10,262	61,046	0.00057969
Hamrle2	5,952	5,952	22,162	0.00062558
aircraft	3,754	7,517	20,267	0.00071821
airfoil_2d	14,214	14,214	2,59,688	0.00128534
FEM_3D_thermal1	17,880	17,880	4,30,740	0.00134735
lns_3937	3,937	3,937	25,407	0.00163916
gupta1	31,802	31,802	21,64,210	0.00213989
net100	29,920	29,920	20,33,200	0.00227121
epb0	1,794	1,794	7,764	0.00241235
lhr07	7,337	7,337	1,56,508	0.00290736
Zd_Jac6	22,835	22,835	17,11,983	0.00328320
jagmesh3	1,089	1,089	7,361	0.00620699
jagmesh2	1,009	1,009	6,865	0.00674308

B. Performance Comparision and Analysis

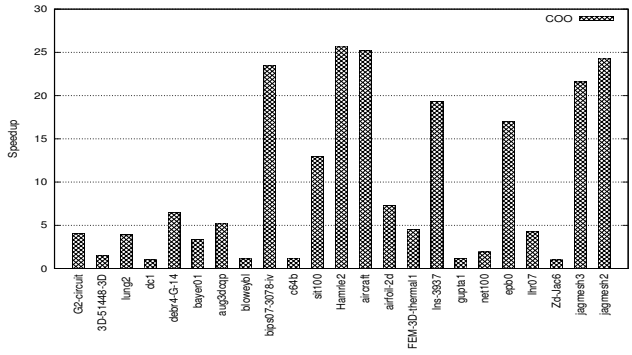

Fig. 5. Speedup of Aligned_COO over COO_flat

We have focused on improving performance of COO format by applying load balancing alignment for skewed sparse matrix. COO_flat supports higher concurrency degree. But, fig.5 shows speedup of ALIGNED_COO over COO_flat on NVIDIA GPU with more accurate result. The result shows that matrix with highly skewed distribution of nonzero gains higher performance using proposed implementation.The ALIGNED_COO format achieve 1x to 25.72x performance gain over COO_flat kernel for highly skewed sparse matrices.

COO format has an major drawback of multiple index

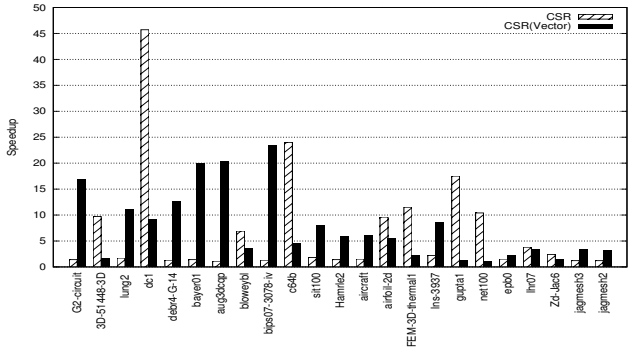

Fig. 6. Speedup of Aligned_COO over CSR kernels

indirection, while CSR is designed as compressed storage format. But, CSR format has its own limitation to work efficiently for large number of rows having row length less than warp size. CSR is suitable to provide high concurrency degree among sparse rows, where CSR vector utilizes maximum computational resources. And, CSR(vector) kernel is used for large size rows to provide higher concurrency degree. Fig. 6 shows speedup of ALIGNED_COO over CSR kernels. The performance gain in ALIGNED_COO over CSR and CSR(vector) kernel shows that this format can be applied to wide range of sparse patterns.

ELLPACK format is well-known compressed storage format for sparse matrix. Log based Performance speedup chart in fig. 9 shows that ALIGNED_COO format gain higher performance over ELLPACK for large and highly skewed matrices. The proposed format gains upto 90x speedup over ELLPACK format for highly skewed data distribution. ELLPACK format is suitable for matrix having average number of nonzero elements. Hybrid format is designed to work with wide range of sparse patterns. Hence, we have also performed comparitive analysis of ALIGNED_COO format with Hybrid format. Comparitive analysis represented in fig.8 shows that Load balancing prime factor for optimizing performance for irregular application like SpMV. This speedup chart shows upto 29.2x speedup of ALIGNED_COO over Hybrid kernel implementation. Performance comparasion of ALIGNED_COO with other well-known sparse formats are given in fig.7. This represents that ALIGNED_COO format gives appreciable performance over other sparse formats for large class of sprase patterns.

Fig.10 represents speedup of GPU based Aligned_COO im-

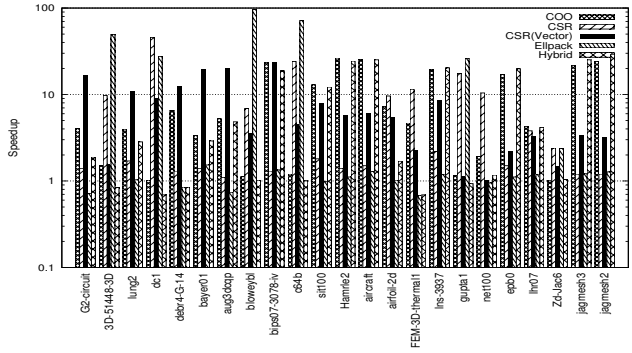

Fig. 7. Log based performance comparison of Aligned_COO and other formats

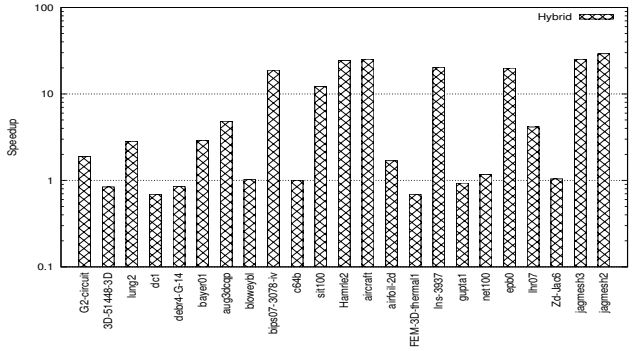

Fig. 8. Log based speedup of Aligned_COO over Hybrid format

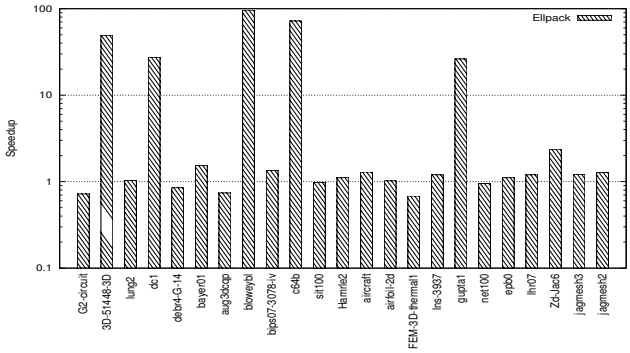

Fig. 9. Log based speedup of Aligned_COO over ELLPACK format

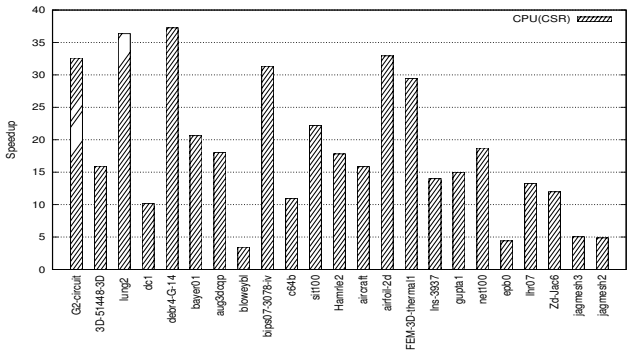

Fig. 10. Speedup of Aligned_COO(GPU) over CSR(CPU)

plementation over serialized execution on CPU. Parallel processing on GPU has major drawback of data transfer rate with low memory bandwidth. We propose a SpMV implementation on GPU for requirement of high concurrency degree, and openMP-based implementation on multi-core CPU for need of low concurrency degree.

VI. Conclusion and Future Work

In this paper, we have discussed various factors responsible for achieving higher performance of SpMV on GPU for matrices with skewed distribution. We have revealed that synchronization-free load distribution, load balancing alignment are most important factor for increasing performance of SpMV kernel and other kernels on GPU. We applied load balancing optimization in Coordinate (COO) format, and achieved 1x to 25.72x performance optimization. The performance of ALIGNED_COO is also found appreciable over other sparse formats for large set of sparse patterns. It has proved importance of load balancing in SpMV kernels for GPU.

Performance gain of ALIGNED_COO kernel over other kernels shows a direction to apply hybrid and heuristic approach to handle wide range of sparse data distribution. In Future, we would like to apply load balancing concept to ELLPACK and CSR format for improving performance. We would like to support heuristics to select most appropriate kernel based on given sparse matrix pattern.

References

[1] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC*, 2009.

[2] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus," *Engineering*, vol. 24704, no. RC24704, 2008.

[3] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus: Implications for graph mining," *CoRR*, vol. abs/1103.2405, 2011.

[4] R. Shahnaz and A. Usman, "Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers," *Int. Arab J. Inf. Technol.*, 2011.

[5] F. Vazquez, G. Ortega, J. Fernandez, and E. Garzon, "Improving the performance of the sparse matrix vector product with gpus," *Computer and Information Technology, International Conference on*, vol. 0, 2010.

[6] A. Dziekonski, A. Lamecki, and M. Mrozowski, "A memory efficient and fast sparse matrix vector product on a gpu," *Progress In Electromagnetics Research*, vol. 116, pp. 49–63, 2011.

[7] 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, 2010.

[8] W. Cao, L. Yao, Z. Li, Y. Wang, and Z. Wang, "Implementing sparse matrix-vector multiplication using cuda based on a hybrid sparse matrix format," in *International Conference on Computer Application and System Modeling*, 2010.

[9] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, 1999.

[10] Z. Wang, X. Xu, W. Zhao, Y. Zhang, and S. He, "Optimizing sparse matrix-vector multiplication on cuda," in *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, vol. 4, june 2010.