Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 13, Number 2, pp. 159–171. http://www.scpe.org

NEW SPARSE MATRIX STORAGE FORMAT TO IMPROVE THE PERFORMANCE OF TOTAL SPMV TIME

NEELIMA B.*, PRAKASH S. R.* AND G. RAM MOHANA REDDY*

Abstract. Graphics Processing Units (GPUs) are massive data parallel processors. High performance comes only at the cost of identifying data parallelism in the applications while using data parallel processors like GPU. This is an easy effort for applications that have regular memory access and high computation intensity. GPUs are equally attractive for sparse matrix vector multiplications (SPMV for short) that have irregular memory access. SPMV is an important computation in most of the scientific and engineering applications and scaling the performance, bandwidth utilization and compute intensity (ratio of computation to the data access) of SPMV computation is a priority in both academia and industry. There are various data structures and access patterns proposed for sparse matrix representation on GPUs and optimizations and improvements on these data structures is a continuous effort. This paper proposes a new format for the sparse matrix representation that reduces the data organization time and the memory transfer time from CPU to GPU for the memory bound SPMV computation. The BLSI (Bit Level Single Indexing) sparse matrix representation is up to 204% faster than COO (Co-ordinate), 104% faster than CSR (Compressed Sparse Row) and 217% faster than HYB (Hybrid) formats in memory transfer time from CPU to GPU. The proposed sparse matrix format is implemented in CUDA-C on CUDA (Compute Unified Device Architecture) supported NVIDIA graphics cards.

Key words: Graphics Processing Unit (GPU), data parallelism, sparse matrix, SPMV computation, compute intensity, memory transfer time, CUDA-C, NVIDIA Graphics Card.

1. Introduction. Graphics processors (GPUs) are proved to be good for data parallel applications. GPUs have also been proved as a good choice for irregular memory access applications that have high data parallelism. Example of such applications include sparse matrix vector multiplication, graph algorithms etc. Several scientific computations use SPMV computation as a main kernel. Improving and optimizing SPMV computation is still a research focus for the new hardware architectures. The sparse storage format used in SPMV determines the performance of the application. The steps involved in SPMV computation on GPU are: data organization (to make memory access efficient on GPU), memory transfer of input data from CPU to GPU and SPMV computation on GPU. The result is very small in size, that need to be sent back to CPU from GPU and this small value is not considered in this work. The sparse storage formats used for CPUs cannot deliver good performance when used for SPMV computation on GPU. So, many GPU specific new formats and optimizations are evolving. Most of the formats and optimization methods have taken only SPMV computation time on GPU into consideration and tried to optimize GPU performance. As SPMV computation is performed on GPU, there are common overheads in terms of data organization time by GPU or CPU, data transfer from CPU to GPU. At the same time, the computation power available on GPU is not negligible and should be utilized for the high performance applications. This paper proposes a new sparse storage format that can reduce the time of data organization and memory transfer, reducing the overall computation time of SPMV. The new format is called as Bit Level Single Indexing (BLSI). BLSI implementation is done on CUDA and proved good for GPU architecture. This format reduces the number of bytes required per flop, reducing the compute intensity or ratio of bytes to flops. It also saves on cache and/or register usage per thread on GPU. The total time comparison shows that BLSI is 2x to 112.6x faster than HYB (HYBrid format) when total time is considered as the SPMV time. The BLSI sparse matrix representation is upto 204% faster than COO, 104% faster than CSR and 217% faster than HYB formats in memory transfer time from CPU to GPU

The paper is organized as follows. An overview of the GPU architecture and sparse formats are given in Sect. 1 and 2 respectively. Section 4 highlights the importance of SPMV optimization on GPU by giving the related work. Section 5 details the new format generation and uses. Section 6 gives the experimental set up. Section 7 gives results and analysis and concludes with future work in Sect. 8.

2. GPU Architecture. Usage of GPUs for general purpose computations have accelerated when NVIDIA introduced CUDA, a general purpose parallel computing architecture. A CUDA device or the GPU is connected to CPU through host interface. CUDA device consists of a set of Streaming Multiprocessors (SMs), each consists of an instruction unit and a shared memory along with a set of Streaming Processors (SPs). Each core can preserve number of thread contexts, specific to the architecture. CUDA has zero-overhead scheduling, that is Fmaintained by tolerating the data fetch latency by switching between threads [1].

*Department of Information Technology, National Institute of Technology, Karnataka, India. (reddy_neelima@yahoo.com) Questions, comments, or corrections to this document may be directed to this email address.

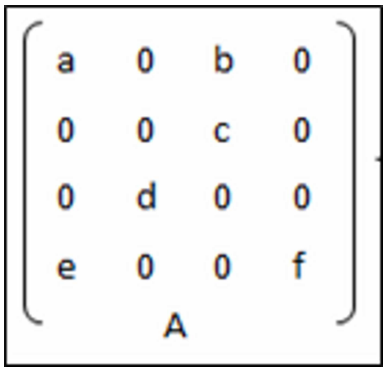

Fig. 3.1: Example sparse matrix A

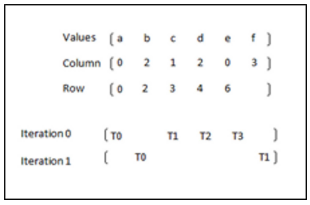

Fig. 3.2: CSR representation sparse matrix and its thread access pattern on GPU

Parallel region of CUDA program is called as kernel. The CUDA API (Application Program Interface) is used to create parallel threads to be executed on the hardware. The kernel is partitioned into a grid of thread blocks that can execute in parallel. The programmer can define the dimensions of the grid and block. The SPs are grouped in SIMD (Single Instruction Multiple Data) fashion and CUDA follows SIMD. Thread blocks are distributed evenly on the multiprocessors. Threads are logically grouped into warps. A warp consists of 32 threads that can execute a single instruction. Each SM executes one warp at a time. Different warps within SM are time shared on the hardware resources. Thread divergence is created with the usage of conditional instructions that serializes the threads.

CUDA device has hierarchy of memories. The device memory is called as global memory. Memory request of a half warp (16 threads) are served together, this is called as coalescing. The request from all the threads of a warp is coalesced into one memory transaction if they are accessing the addresses in the same segment. Once the addresses are accessed by the half warp in one segment, it is called as fully coalesced. This is one of the optimization that is looked into for any CUDA computations. The shared memory is accessible by threads of the same block. Along with this, set of registers are shared by the threads of a block. The constant and texture memories are read only memories in global space with on-chip caches. The programmer can bind these regions to read only data before launching a kernel [2].

3. Sparse Matrix Formats on GPU. The SPMV computation involves a sparse matrix A multiplied by a dense vector x, represented as $y{=}Ax$. The standard formats like CO-Ordinate (COO), Compressed Sparse Row (CSR), ELL (ELLPACK) and HYB (HYBrid) [10] formats are considered in this paper to compare with the proposed format and explained briefly here. The other formats are built on these standard formats that are given under related work section. Bell and Garland [10] gave a detailed study of sparse formats and their access pattern on GPU.

3.1. Compressed Sparse Row (CSR) Format. A sample sparse matrix A is shown in Figure 3.1 and Figure 3.2 shows the CSR representation and its thread assignment on GPU hardware. Here, one thread per row is launched. In Figure 3.2, T0 through T3 reads the elements in each row, first iteration will give full throughput, but in the next iteration only two threads read the elements. The column indices are not accessed simultaneously even though they are stored contiguously, which causes poor performance of this format.

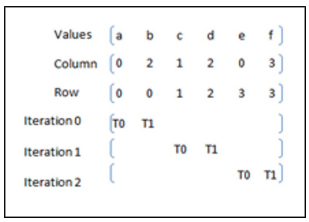

Fig. 3.3: COO representation of sparse matrix and its thread access pattern on GPU

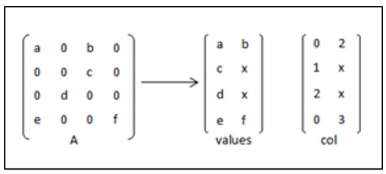

Fig. 3.4: ELL representation of sparse matrix

3.2. CO-Ordinate (COO) Format. In COO format, there are three one-dimensional arrays of same size as that of the number of non-zeros in the matrix. It causes an overhead in terms of memory transfer from CPU to GPU. The COO representation and thread access pattern is shown in Figure 3.3. It can be noted that warp sized (here warp size is 2) number of threads work on the non-zeros in each iteration. Reduction or atomic operations can be used across threads. Reduction operation is used to compute the sum of an array of numbers in parallel. Atomic operations allow multiple threads to perform concurrent read-modify-write operations in memory without conflicts. The syntax of atomic operation in CUDA is as follows: float atomicAdd(float* address, float val).

3.3. ELL Format. ELL representation is shown in Figure 3.4 and thread assignment is given in Figure 3.5. This technique is suitable for vector architectures. Column major access is preferred as it offers better coalescing, and shared memory can be used with ease since there wont be any bank conflicts. As shown in Figure 3.5, the threads are launched in column major order. After each iteration, the threads advance to the next column for execution.

3.4. Hybrid (HYB) Format. Hybrid format is combination of sparse matrix formats proposed by [10]. HYB uses combination of ELL and COO. The HYB structure is shown in Figure 3.6. ELL and COO is faster for SPMV in many cases, but it has a CPU-GPU memory transfer overhead when compared with other formats, since it requires five memory transfers; two for ELL and three for COO. The Thread access pattern is a combination of the access pattern of ELL and COO.

ELL and COO combination as used in HYB format is preferred because ELL is proved to be good when the difference in number of non-zeros in each row is negligible, and COO is proved to give a modest performance when the number of non-zeros per row is variable. It can be seen that the portion of the row till size L (calculated empirically) is considered to be the ELL portion of the row and if the size of the row exceeds L, it is considered as the COO portion of the row and is stored in ELL format. Alternatively, the format can take entire row as ELL if its size is less than or equal to L or as COO if its size is more than L.

4. Related Work. SPMV performance improvements and optimization based publications are on rise in recent times. The importance of communication overhead in high performance application and the need of optimizing this overhead were studied extensively in the literature as follows. Ravi et. al [3] have proposed a heterogeneous BLAS library for SPMV computation considering communication bandwidth as one of the parameters to tune the applications parameters according to the architecture in a heterogeneous system. Our work optimizes this bandwidth limitation by using a new data structure for the sparse matrix representation for a single GPU. Xingfu wu et. al [4] proposed hybrid optimization methods for scientific and compute intensive applications for CMP clusters. They map processors per node optimally and similarly this work also maps computation to the threads to increase the performance of the applications. Vuduc et.al [5] discussed three main applications for which GPU has some limitations. One of the applications discussed is SPMV operation which

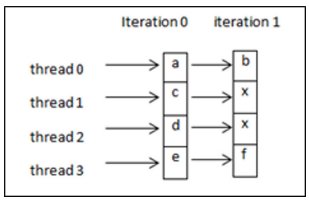

Fig. 3.5: ELL thread access pattern on GPU

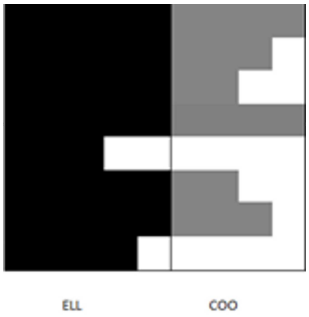

Fig. 3.6: HYB structure of sparse matrix

is bandwidth limited and mentioned that porting this application is beneficial if CPU to GPU communication is reduced or removed by some methods. The method proposed in this paper considers the memory transfer time and improves it.

Lee et.al [6] have analysed main kernels that appear in most of the applications and mentioned the requirement of optimizations on and off the device. SPMV is one kernel that can improve performance if bandwidth limitations are overcome. BLSI method optimizes data organization time and memory transfer time that can strengthen the usage and need of GPU for general purpose computation. Gregg and Kim [7] proposed that moving computation to data improves the application performance than always moving data so that data movement overheads are reduced. They proposed to use system intelligence and develop an automated tool to control the assignment of the computation. BLSI method also optimizes data movement by reducing the amount of data to be given to the computation with easy and flexible implementation. Michael et. al [8] have also considered bandwidth bound applications for GPU and proposed an API for creating DMA warps that exclusively handles memory transfers from off chip memory to on chip memory. Our method of optimizing memory bandwidth is external to the device. Thomas et.al [9] proposed an automatic tool that consists of runtime library and compiler optimizations to optimize the number of communications from CPU to GPU. The tool proposed by them combines all the memory copies to the GPU of multiple kernels. This method is beneficial if the kernels use few common data. Our method is to optimize communication especially for SPMV, which is very commonly used in most of the scientific and engineering applications.

Storage format of sparse matrix is very important in determining the performance of SPMV. Various proposals of sparse matrix storage formats are discussed here. Bell and Garland [10] proposed a new GPU suitable sparse matrix storage format namely HYB which is a combination of ELL and COO. The kernel time of HYB is improved a lot at the cost of high data organization and memory transfer time. The data structure proposed in this paper gives improvements in overall time of SPMV computation, of course at the cost of SPMV kernel time. Blelloch et al. [11] studied SPMV on vector machines. Choi et al. [12] proposed the BELLPACK representation that suits for matrices with dense blocks. Yang et al. [13] proposed optimizations using texture to increase data locality that improves SPMV performance. Monakov et al. [14] [15] implemented blocked SPMV and Sliced ELLPACK in which a slice of the matrix, a set of adjacent rows, are stored in ELL format.

Vazquez et al. [16] proposed ELLPACK-R format that uses ELL format with an array containing the length of each row. They assigned multiple threads to a row to balance the computations of threads. Anirudh et al. [17] and Kiran and Kishore [18] considered combination of CSR and ELL formats for storing the matrix. Dziekonski et al. [19] have proposed sliced ELLR-T data structure. Most of the formats proposed increases the

performance on chip and not considered other overheads like memory transfer and data organization time. The communication optimized sparse data structure by name CSPR was proposed by Neelima and Prakash [20]. The method was not scalable and not tested with CUSP results.

Various optimizations to the existing formats can further improve the performance of SPMV computation on GPU. The work by Yelicki et al. [21] and Vuduc et al. [22, 23, 24, 25] gave extensive optimizations and auto-tuning for sequential machines. Low-level code optimizations and data structure optimizations for single core and parallelized optimizations for multi-core architectures is given by Williams et al. [26], [27]. Brahme et al. [28] proposed a greedy extraction of dense sub matrices that load balances and overlaps communication and computation, so, reduces memory traffic and hides communication latencies. It is an on chip optimization, uses greedy algorithm and has less scope for scalability. Zein and Rendell [29] proposed an analysis and selection tool that selects the best performing implementation for SPMV. Wang et al. [30] proposed optimizations for SPMV computation on CUDA by giving optimized CSR storage, thread mapping etc. Baskaran and Bordawekar [31] also proposed optimizations to improve SPMV performance by using synchronization free parallelism, optimized threads mapping, data fetch and data reuse. This work proposed a new storage format for the sparse matrix that improves the performance of the application by optimizing the data organization and memory transfer time from CPU to GPU. This work also optimizes the computation by optimizing the thread mapping.

5. New Format for Sparse Matrix Representation. The concept of single index representation is implemented at the bit level and hence the name Bit-Level Single Indexing (BLSI) is given. This is also implemented at the integer level by using division and modulus operations, but for the ease of programming and optimal index generation, bit level implementation is chosen for observing the results. The results or the total time is same in both bit level and integer level implementations.

5.1. Index Generation. Contrast to many standard sparse matrix formats like COO, CSR etc., which use one array or one data structure for column index and another to keep information about the row, BLSI method uses only single array or data structure to store the indices by embedding the column information in the bits of row indices information. Hence, this method needs only one array of size equal to the number of non-zero elements to represent the indices. If the column index is also big and could not fit into the remaining bits that are available, then offset is used to keep track of column index while using small value to represent the column index that fits into the array. Offset size will be much smaller compared to the size of the second array in COO format (ITER<<<nnz) and smaller than CSR format pointer array (ITER<<ptr) to mention a few. The size of the offset array, ITER, is computed as:

$$ITER = (numEle + (THPB * BLOCKS) - 1)/(THPB * BLOCKS) \tag{5.1}$$

Terminology used:

- ITER : Number of iterations required and size of offset array
- THPB : Threads per block
- BLOCKS : Number of blocks launched in the grid
- numEle : Number of non-zero-elements
- BIT_SHIFT : The power of two taken (in the example given, it is 17)
- Offset[] : Array to store the offsets for different ITER
- newRow[] : Array that contains the value of row index of corresponding non-zero element
- newCol[] : Array that contains the value of column index of the corresponding non zero element
- index[] : Array that contains the row index and column index embedded into few bits
- value[] : Array that contains the non-zero-value
- REM_AND : (2 power of BIT_SHIFT)-1
- B[] : Dense vector array
- dotd : Contains result of multiplication of non-zero-value with the corresponding element in the vector

5.2. BLSI: Bit Level Single Indexing. BLSI is a new format proposed to represent sparse matrix indices to reduce memory transfer overhead in the memory bound SPMV computations. The main operations involved in SPMV computation on a GPU are organizing the data (for enabling global coalescing by changing the data layout and other optimizations to get performance benefit from GPU), sending the data from CPU to GPU and executing SPMV computation on GPU. The memory transfer time for the results from GPU to CPU is very minimal and not considered here. The SPMV computation time is given as total time of all these operations and through this new data structure improvements are seen in all these steps of SPMV computation.

Through this new method, this work presents optimizations at three levels as follows:

- At the data format or data mapping level
- At the communication level between CPU and GPU
- At the computation level while assigning threads for computation

Algorithm 5.2.1 Index generation in BLSI format

```
for(j=0;j<ITER;j++)
{
offset[j]=newCol[j*THPB*BLOCKS];
for(i=j*THPB*BLOCKS;i<((j+1)*THPB*BLOCKS) && (i<numEle);i++)
{
index[i]=((newCol[i]-offset[j])<< BIT_SHIFT)+newRow[i];
}
}
```

The number of bits required for the single index representation is the sum of BIT_SHIFT size and the offset [] data size. The size of BIT_SHIFT is the immediate next power of 2 of the row size. For example, if the number of rows are 86, 000, then 2^{16} = 65,536 and 2^{17} = 131,072. So BLSI considers BIT_SHIFT size as 17. As mentioned earlier, BLSI needs offset array to reduce the number of bits to represent row and column information into a single value. The column index is left shifted by BIT_SHIFT size and then added with row index. The index generation algorithm is given in Algorithm 1. The column index added with row index and the row index is extracted on GPU in SPMV GPU kernel as given in Algorithm 2.

Algorithm 5.2.2 SPMV computation using BLSI format

```
unsigned int i =blockDim.x * blockIdx.x + threadIdx.x;
for( ;i<N; i+=BLOCKS * THPB )
{
row = index[i] & REM_AND;
col = (( index[i] >> BIT_SHIFT) + offset[i / (THPB*BLOCKS)]);
dotd = value[i] * B [col];
atomicAdd( result+row, dotd); }
```

The BLSI index can also be computed as row-index * n + col-index, which is equivalent to the index calculated in Algorithm 1. The index of row and column computation from single index as shown in Algorithm 2 is equivalent to obtaining row by doing division and modulus operation on BLSI index with size of the matrix to get row and column index respectively as shown in Figure 5.1.

5.2.1. Data Format Optimization. The proposed BLSI format optimizes the data organization time and also the data storage requirement for the memory bound SPMV computation. The input data to SPMV kernel on GPU needs reorganizing the data to enable the global coalescing and other optimizations to get the actual performance improvements on the GPU. BLSI method requires pre-processing to restructure the matrix into a single index based matrix from .mtx, Matrix Market Format, a standard file extension used in many benchmark matrix data that uses COO format to represent the values in the matrix. But this pre-processing is involved with any other sparse data representation other than COO, which needs to be generated from COO. For example ELL, CSR etc. formats have to be generated form the Matrix Market Format i.e, COO. The overhead involved in generating BLSI format is still very less compared to the other formats. Table 1 shows the total time for BLSI and HYB format. Total time includes data reorganization time, Sending data from CPU to GPU and the kernel computation on GPU. HYB format is highly optimized format on the GPU so far. The

```
val             = [a  b  c  d  e   f]
BLSI-Index      = [0  2  6  9  12  15]
BLSI-Index      = row-index * n + column-index
row-index       = BLSI-Index / n
column-index    = BLSI-Index % n
```

Fig. 5.1: Sample data format representation of sparse matrix A in BLSI method, single index calculation and extraction of the row and column index calculations represented at the data type level. n is the size of the square matrix.

Table 5.1: Total and kernel time executions for BLSI and HYB formats

Input Matrix	$BLSI-total$ time in ms	$HYBtotal$ time in ms	*Ratio* : HYB/BLSI	$HYB-Kernel$ time in ms	$BLSI-Kernel$ time in ms
web	21.307	581.689	27.301	0.847	4.637
ship	21.399	2191.501	102.411	1.426	3.396
scircuit	7.478	248.329	33.208	0.459	1.371
rma	10.098	741.014	73.379	0.617	2.207
rail	55.504	108.001	1.946	3.772	20.656
qcd	8.686	546.219	62.888	.227	2.287
pwtk	31.701	3344.274	105.493	1.502	3.607
pdb	12.674	1427.531	112.633	0.971	1.479
mc2depi	14.084	604.106	42.894	0.246	2.472
mac	11.134	420.245	37.744	0.639	2.178
dense	13.949	29.763	2.134	0.849	1.994
cop20k	9.851	796.334	80.838	1.047	2.412
consph	17.356	1951.671	112.452	0.889	2.427
cant	12.384	1361.669	109.959	0.765	1.693

total time comparison shows that BLSI is 2x to 112.6x faster than HYB when total time is considered as the SPMV time. If we compare the total time (that includes the kernel time) with the kernel time of the respective sparse formats then total time is almost 2.6x to 8x times the kernel time on GPU for BLSI and 28x to 2452x times the kernel time for HYB format. The total time is observed by using the CUDA event recoder for the CUSP library operations. The improvements in communication between CPU and GPU are shown in Sect. 7. BLSI kind of new formats are desired for the GPUs that are massively data parallel architectures to show the overall benefit of using the GPU for data parallel computations. The problem of CPU-GPU communication scales up as the number of GPUs used increases in a system. If the matrix is very large and does not fit into the chip storage, then SPMV performance still degrades and BLSI format gives better overall timing in this case too. Hence the proposed new data representation is an optimized data format for the GPUs.

5.2.2. Communication Optimization. The Matrix Market format is the most used standard format that uses COO data format. An optimized library like CUSP [32] etc., uses .mtx files and builds other formats from COO. So, the communication time between CPU and GPU is the time taken to transfer two index arrays for any format. The BLSI sparse matrix representation is up to 204% faster than COO, 104% faster than CSR and 217% faster than HYB formats, in memory transfer time from CPU to GPU. The results of comparison for the different matrices are given in Sect. 7.

5.2.3. Computation Optimization. BLSI uses atomicAdd() computation where all the threads need to synchronize to add the row-wise product to single sum. To optimize even this computation time using atomic operations, the thread assignment is modified as follows. In the pre-processing stage, BLSI does some changes to the COO matrix (from Matrix Market file) [33], to change the data access pattern that in turn optimizes the thread computations. The detailed explanation is as follows. For the matrix given in Figure 3.1, the thread assignment for an SPMV kernel is shown in Table 5.2.

If the data access pattern is not changed, two consecutive threads take the computation of values a and b of row 1. Assume that both finish the computation at the same time. These two values need to be added to a single value that represents the sum of the products of that row. Only one thread can access the sum

Table 5.2: Thread assignment: continuous threads are assigned for values with in a row

Old Thread Assignment						
Thread ID	1	2	3	4	5	6
Val	a	b	c	d	e	f
Row	1	1	2	3	4	4

Table 5.3: Data representation of .mtx file after changed by BLSI method

Row	Col	Val
1	1	a
4	1	e
3	2	d
1	3	b
2	3	c
4	4	f

as it is an atomic operation. The problem here is two threads finished the work at the same time but sum of products delays the overall result because of the atomic operation. To overcome this, BLSI uses a different data access pattern that delays the single row product computations so that the conflict to write to the row wise sum is delayed and in turn the row wise sum is available to all the products when they are completed with the computation. This technique has boosted the performance to much higher values. The time of execution is dominated by the row that has large non-zero-values.

To improve the performance using BLSI, BLSI changes the data representation of the .mtx file as follows. The Matrix market file is sorted based on column indices rather than row indices. Hence the .mtx representation will be changed as shown in Table 5.3 for the matrix given in Figure 3.1.

The thread assignment is done column wise here. Threads that belong to the same row need not wait for atomic operation, reducing computation time. The atomic operation of multiple threads is delayed that in turn improves overall performance of SPMV kernel. The thread assignment is shown in Table 5.4. By changing the access pattern, it gives up to 92% improvement in SPMV kernel computation than the previous access pattern.

These optimizations used are external to the device. As shown in Sect. 7, the results are promising in-terms of memory transfer time form CPU to GPU, overall GPU computation time that includes memory transfer and kernel executions time, new format generation time and also overall program execution time. Hence BLSI can be used as one of the sparse matrix formats that better utilizes the device.

6. Experimental Setup. This section describes the experimental setup used. The workload selected and workload parameters used in the experiments are given. Monitors used for the observation of outputs are listed. The specifications of hardware and software used for the experimentation are given in Table 6.1.

6.1. Workload. The input matrices used are the same workbench used by William et al. [26], [27]. These are the real data observed form the experiments and posted in University of Florida Sparse Matrix Collection [33]. The input workload use and its characteristics are given in Table 6.2.

6.2. Workload Parameters. The input matrices are evaluated with the proposed methods. The workload is characterized by performance observed in GFlops (10^9 Flops) and bandwidth in GBytes (10^9 Bytes). The time is measured for the kernel execution and these values are derived with the details of the matrices used.

GFlops is computed as follows:
GFlops = ((2* nnz)/(kernel execution time in milliseconds * 1000000))

Bandwidth is computed as follows:
Bandwidth = ((3*nnz) + (2*# of rows)*4)/(kernel execution time in milliseconds * 1000000)

To compare the communication time between CPU and GPU, only the time of communication or time for the memory transfer is taken into the consideration. The total time is observed as the time for the data organization of the input, memory transfer time form CPU to GPU and the SPMV kernel execution time on the GPU. For observations, CUDA event recorder is used for GPU related computations and CPU timer is used for the computations on CPU.

Table 5.4: Thread assignment: thread 1 access first value of row1, thread 2 access first value of row 2 and goes on

New Thread Assignment						
Thread ID	1	2	3	4	5	6
Val	a	e	d	b	c	f
Row	1	4	3	1	2	4

Table 6.1: Specifications of hardware and software used in experiments

System/Hardware Specifications	
Processor name and code name	Intel Core i7 2600, Sandy Bridge
Processor specification	Intel (R) Core(TM) i7-2600 CPU @ 3.40GHz
Graphics Interface	PCI-Express
Graphics processor name and code name	NVIDIA GeForce GTX 470-GF100
PCI-E link width	X16
Memory type	DDR3
Memory size	1280MB
Software Specifications	
Windows Version	Microsoft Windows 7(6.1) Service pack 1(Build 7601)
DirectX Version	11.0
Programming platform	Visual Studio 2010
CUDA SDK version	3.2 and 4.0

6.3. Monitors. NVIDIA provides ParallelNSight to profile the CUDA programs. The Communication time between CPU to GPU is taken from the profiler memory copy time from host to device. CUDA C also provides an event recorder to observe the elapsed time. For the GFlops and Bandwidth computation that are given as part of program, CUDA event record is being used to measure the kernel time. It is also been verified that the time taken by the CUDA event and the profiler for the kernel execution are same. Hence the mode of observation done is valid.

7. Analysis and Interpretation of Results. This section compares the proposed BLSI format against the most commonly used formats like COO, CSR and HYB. The results are given for the following comparisons.

Communication time (or) Memory transfer time: The memory copy (memCopy for short) time between CPU and GPU is compared for all the matrices given in the workload. They are compared by considering the time of memCopy in milliseconds. Figure 7.1, shows that BLSI takes less time for memCopy in all the cases. BLSI is up to 107% better than CSR, 204% better than COO and 217% better than HYB when compared for memCopy time of sparse matrix data from CPU to GPU. The percentage of variations in memory transfer time of COO, CSR and HYB with respect to BLSI is shown in Figure 7.2. In scircuit, the number of elements per row is very small and hence CSR ptr and BLSI offset sizes become same and also the kernel time of CSR is better for such matrices. In general, for a matrix with very few elements per row or few rows with large elements, BLSI method will not perform well.

Performance Observation in time of execution by considering the kernel + memCopy time of SPMV Computation: Figure 7.3 compares four formats with respect to SPMV computation that is, CPU to GPU communication time plus the kernel time. The total time taken by BLSI format in matrices scircuit, rail is more because, they have more nonzero elements distributed in very few rows. So the computation time taken by BLSI is higher, because BLSI uses atomicAdd() for row wise sum. The BLSI (our method) outperforms than all other methods for different structures of the matrices, when SPMV computation time and device memCopy time are considered. As explained earlier, this format was proposed to reduce the CPU-GPU communication, this optimization has resulted in overall better performance also. BLSI is 80% better than CSR, 164% better than COO and 161% better than HYB (as shown in Figure 7.3) when both the memory transfer time and kernel time are considered.

Figure 7.4 compares different sparse storage formats by considering total time as the data organization time, memory transfer time and the kernel time. The HYB results are not shown in Figure 7.4, because it deviates the graph. The values are given in Table 5.1. The total time in mac, scircuit, web and mc2depi of BLSI is little more than total time of CSR because offset array used in BLSI and kernel time dominates by the longest row reduction. As the number of elements per row is very small in these matrices, CSR kernel performance is better

Table 6.2: Specifications of hardware and software used in experiments

Matrix	Rows	NNZ	NNZ/Row	Description
cant	62,451	4,007,383	64	FEM cantilever
consph	83,334	6,010,480	72.1	FEM concentric spheres
cop20k_A	121,192	2,624,331	21.6	Accelerator cavity design
dense2	2000	4,000,000	2,000	dense matrix in sparse format
mac_econ_fwd500	206,500	2,100,225	3.9	Macroeconomic model
mc2depi	525,825	2,100,225	3.9	2D Markov model of epidemic
pdb1HYS	36,417	4,334,765	119.3	protein data bank 1HYS
pwtk	217,918	11,634,424	53.3	pressurized wind tunnel
qcd5_4	49,152	1,916,928	39	quark propagators (QCD/LGT)
rail4284	4,284	11,279,748	2,632.9	Railways set cover, constraint matrix
rma10	46,835	46,835	50.6	3D CFD of Charleston Harbor
scircuit	170,998	958,936	5.6	Motorola circuit simulation
shipsec1	140,874	7,813,404	55.4	FEM Ship section / detail
webbase-1M	1,000,005	3,105,536	3.1	Web connectivity matrix

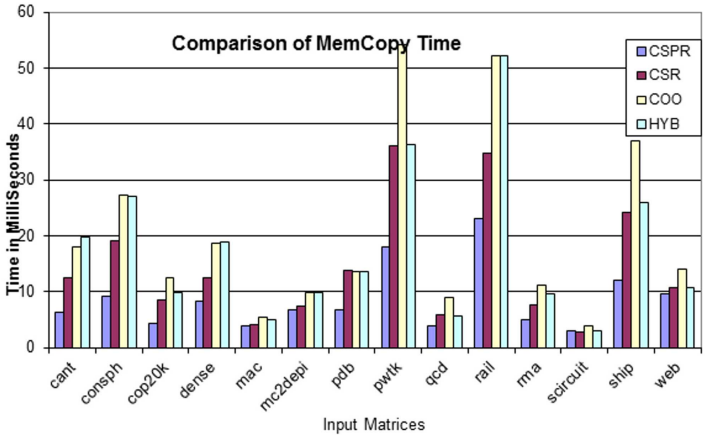

Fig. 7.1: Comparison of memCopy time between CPU and GPU

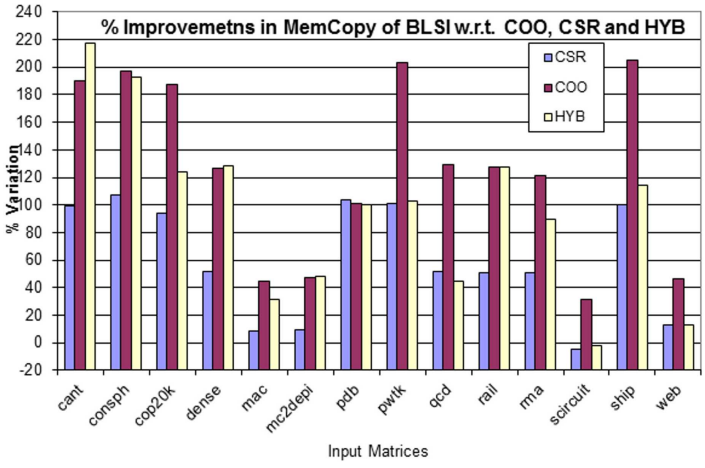

Fig. 7.2: Percentage of variation in Memory Transfer Time between CPU and GPU

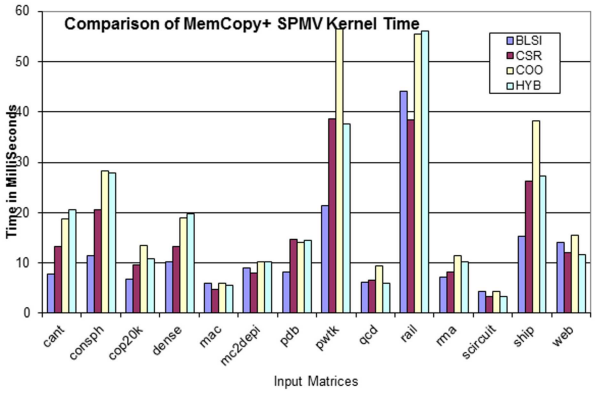

Fig. 7.3: Performance comparison in time that includes memCopy time and kernel execution time

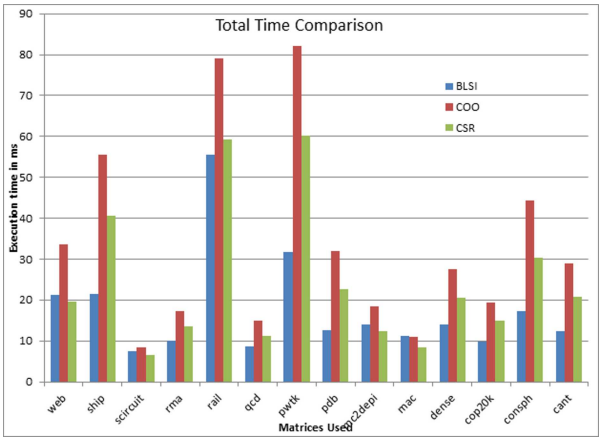

Fig. 7.4: Performance comparison of different formats by considering the total time as time that includes data organization time, memCopy time and kernel execution time

than the BLSI format. These graphs show that how the memory bound computations affect the performances of GPU.

Performance in terms of GFlops and GBytes by considering only kernel execution: The performance in GFlops is computed using only the kernel time. As BLSI do not improve the kernel time of SPMV than the HYB etc., the GFlops computed considering the kernel time is less for BLSI format. BLSI improves the overall time of the SPMV computation at the cost of increased kernel execution time. GFlops graph is shown in Figure 7.5 and bandwidth measurement in GBytes is shown in Figure 7.6. These graphs show the performances of various formats on the GPU.

8. Conclusions. Our experiments have shown that single indexed sparse matrix representation can give substantial improvement in performance while considering the total time of SPMV computation. Total time includes the time for the main operations involved in SPMV, i.e. time of data organization, time of memory transfer and time of SPMV computation on the GPU. The improvement in performance is because of reducing the complexity of data organization and reducing the data to be transferred at the cost of GPU execution time. The improvements shown are not negligible and even for an iterative solution the improvement is beyond the multiple of number of iterations.

The idea of reducing the memory transfer overhead and data organization overhead by using a new data structure for the sparse matrix is novel and it can be improved further. The SPMV performance on the GPU can further be improved by using various optimizations like parallel reduction for row wise sum calculation to mention a few. BLSI or any other formats performance is determined with respect to the input matrix

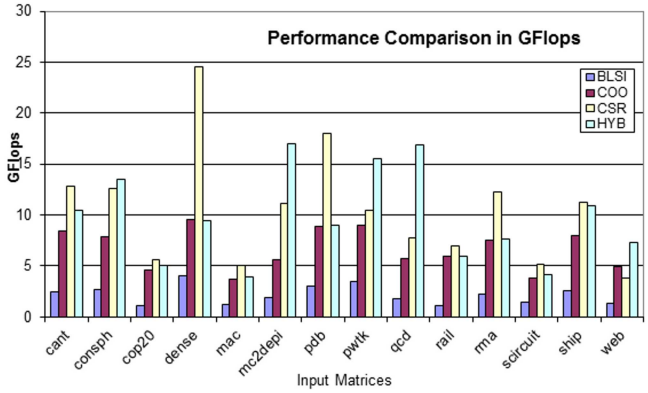

Fig. 7.5: Performance comparison in GFlops considering only kernel execution time of SPMV

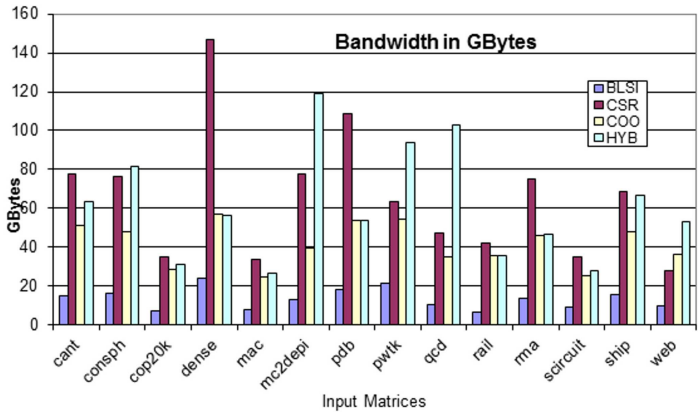

Fig. 7.6: Comparison of bandwidth in GBytes for the different formats considered

sparseness. BLSI kernel performance on GPU is less compared to other formats when the matrix has few rows with large number of elements. An automated tool that can suggest a best data structure from the existing sparse formats based on the input matrix properties like number of elements per row is an on-going work. This can be integrated at runtime so that on the fly data structure to be used can be decided based on the input matrix and also architecture of the device to give the best performance for the SPMV computation.

REFERENCES

[1] *http://developer.nvidia.com/*

[2] Sanders, J and Edward, K. ,*CUDA by example: an introduction to genral purpose GPU programmingCompanion*, Addison-Wesley Professional, 2010,PP: 312.

[3] Ravi Reddy, Malexey Lastovetsky and Pedro Alonso, *HETEROPBLAS: a set of parallel basic linear algebra subprograms optimized for hetergeneous computational clusters Companion*, Scalable Computing: Practice and Experience,10(2)(2009), pp. 201–216.

[4] Xingfu Wu, Valerie Taylor, Charles Lively and Smeh Sharka*Performance analysis and optimization of parallel scientific applications on CMP clusters Companion*, Scalable Computing: Practice and Experience, 10(1)(2009), pp. 61–74.

[5] Vuduc, W. R., Chandramowlishwaran, A., Choi, J., Guney, M., and Shringarpure, A. *On the limits of GPU acceleration Companion*, in USENIX conference on hot topics in parallelism-HotPar-10. Berkeley, CA, USA, 2010, pp. 13–19.

[6] Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, M., Smelyanskiy, M., Chennupaty, S., and Hammarlund, P. *Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU Companion*,in 37th Annual International Symposium on Computer Architecture, Saint-Malo, France, 2010, pp. 451–460

[7] Gregg, C., and Kim, H. *Where is the data? why you cannot debate CPU vs. GPU performance without the answer Companion*,

in 11th IEEE International Symposium on Performance Analysis of Systems and Software- ISPASS-11. Austin, Texas, 2011, pp. 134–144.

[8] Bauer, H., Cook, H., and Khailany, B. *CudaDMA: optimizing GPU memory bandwidth via warp specialization Companion*, in International Conference on High Performance Computing, Networking and Storage Analysis. (SC-11), Seattle, Washington, USA, 2011, Article 12.

[9] Thomas, B. J, Prabhu, J.P., Jablin, J.A., Johnson, P.N., Beard, R.S., and August, I.B. *Automatic CPU-GPU communication management and optimization Companion*,in 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI-11), san Jose, California, USA, 2011, pp. 142–151.

[10] Bell, N., and Garland, M. *Implementing sparse matrix-vector multiplication on throughput-oriented processors Companion*,in International Conference on High Performance Computing, Networking and Storage Analysis (SC-09), Portland, OR, 2009, Article 18.

[11] Blelloch, G. E., Heroux, M. A., and Zagha, M. *Segmented operations for sparse matrix computations on vector multiprocessors Companion*,Technical Report, Department of Computer Science, Carnegie Mellon University (CMU), Pittsburgh, PA, USA, CMU-CS-93-173.

[12] Jee, W. C., Singh, A., and Vuduc, W.R. *Model-driven autotuning of sparse matrix-vector multiply on GPUs Companion*,in 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP-10), Bangalore, India, pp. 115–126.

[13] Yang, X., Parthasarathy, S., and Sadayappan, P. *Fast sparse matrix-vector multiplication on GPUs: implications for graph mining Companion*, Proceedings of the VLDB Endowment VLDB Endowment Hompagearchive, 4(4)(2011), pp. 231–242.

[14] Monakov, A., and Arutyun, A. *Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs Companion*,in 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS-09), Samos, Greece, 2009, pp. 289–297.

[15] Monakov, A., Avetisyan, A., and Lokhmotov, A *Automatically tuning sparse matrix-vector multiplication for GPU architectures Companion*, in 5th international conference on High Performance Embedded Architectures and Compilers (HiPEAC-10), Pisa, Italy, 2010, pp. 11–125.

[16] Vazquez, F.,Ortega, G., Fernndez, J.J and Garzn, E.M. *Improving the performance of the sparse matrix vector product with GPUs Companion*,in 10th IEEE International Conference on Computer and Information Technology (CIT-10), Bradford, UK, 2010, pp. 1146–1151.

[17] Maringanti, A., Athavale, V., and Patkar, S. *Acceleration of conjugate gradient method for circuit simulation using CUDA Companion*,in 16th annual IEEE International Conference on High Performance Computing (HiPC 2009), Kochi, India, 2009, pp. 438-444.

[18] Mateti, K.K., and Kothapalli, K. *Accelerating sparse matrix vector multiplication in iterative methods using GPU Companion*,in International Conference on Parallel Processing (ICPP-11), Taipei, Taiwan, 2011, pp. 612–621

[19] Dziekonski, A., Lamecki, A., and Mrozowski, M. *A memory efficient and fast sparse matrix vector product on a GPU Companion*,Progress in Electromagnetic Resaerch, 116 (2011), pp. 49–63.

[20] Neelima, B., and Prakash, S. R. *Effective sparse matrix representaiton for the GPU architectures Companion*,International Journal of Computer Sceince, Engineering and Applications, 2(2),(2012), pp. 151–165.

[21] Im, E. J., Yelick, K., and Vuduc, W. R. *Sparsity: optimization framework for sparse matrix kernels Companion*,International Journal of High Performance Computing Applications, Sage Publications, CA, 18(1), (2004), pp. 135–158.

[22] Vuduc, W. R., and Hyun-Jin, M. *Fast sparse matrix vector multiplication by exploiting variable block structure Companion*, in High- Performance Computing and Communications (HPCC-05), Sorrento, Italy, 2005, pp. 807–816.

[23] Vuduc, W.R. *Automatic performance tuning of sparse matrix kernels Companion*, Ph. D. Thesis, University of California, Berkeley, CA, USA, 2003.

[24] Vuduc, W.R., James, W. D., and Katherine, A. Y. *OSKI: a library of automatically tuned sparse matrix kernels Companion*, J. Phys.: Conf. Series, IOP Science, 16 (2005), pp. 521–530.

[25] Lee, B. C., Vuduc, W. R., Demmel, J., and Yelick, K. *Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply Companion*,in International Conference on Parallel Processing (ICPP04), Montreal, Quebec, Canada, 2004, pp. 169–176.

[26] Williams, S., Oliker, L., Vuduc, W.R., Shalf, J., Yelick, K ., and Demmel, J. *Optimization of sparse matrix-vector multiplication on emerging multicore platforms Companion*,in International Conference on High Performance Computing, Networking and Storage Analysis (SC-07), Reno, Nevada, 2007, Article 38, 12 pages.

[27] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. *Scientific computing kernels on the Cell processor Companion*, International Journal of Parallel Programming, Kluwer Academic Publishers Norwell, MA, USA, 35(3), (2007), pp. 263–298.

[28] Brahme, D., Mishra, R.B., and Barve, A. *Parallel sparse matrix vector multiplication using greedy extraction of boxes Companion*,in International Conference on High Performance Computing (HiPC-11), Goa, India, 2011, pp. 1–10.

[29] Zein, H.E.A., and Rendell, P. A. *From sparse matrix to optimal GPU CUDA sparse matrix vector product implementation. Companion*, in 10th IEEE/ ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, Victoria, Australia, 2010, pp. 808–813

[30] Wang, W., Xu, X., Zhao, W., Zhang, Y., and He, S. *Optimizing sparse matrix-vector multiplication on CUDA Companion*,in 2nd International conference on Education Technology and Computer (ICETC-10), Shanghai, 2010, pp. 109–113

[31] Baskaran, M.M., and Bordawekar, R. *Optimizing sparse matrix- vector multiplication on GPU Companion*, IBM Research, 24704 (2008).

[32] *Cusp-library, http://code.google.com/p/cusp-library/*

[33] *http://www.cise.ufl.edu/research/sparse/matrices/*

Edited by: Marcin Paprzycki
Received: May 1, 2012
Accepted: June 15, 2012