A New Segmentation-Based GPU-Accelerated Sparse Matrix-Vector Multiplication

Kai He*, Sheldon X.-D. Tan*, Esteban Tlelo-Cuautle†, Hai Wang‡ and He Tang‡
*Department of Electrical Engineering, University of California, Riverside, CA, 92521
†Institute National Astrophysics, Optical and Electrics (INAOE), Puebla, Mexico
‡School of Microelectronics and Solid-State Electronics, UESTC, Chengdu, China, 610051

***Abstract*—In this paper, we propose a new fast parallel sparse matrix-vector multiplication (SpMV) algorithm on GPU platforms. The new algorithm, called *segSpMV*, is based on the compressed sparse row (CSR) format and can be applied to wide computational applications with both structured and unstructured matrices. The SpMV operation has very low computing to communication ratio and is bandwidth-limited. The new SpMV algorithm tries to reduce the memory access by partitioning the rows, whose nonzero patterns are irregular in general, into a number of fixed-length segments. As a result, both multiplication and summation phases now can enjoy the coalesced memory access and they can be finished in one kernel launch. The summation phase can also be further improved by using GPU reduction techniques for large segment lengths. The resulting SpMV method constantly outperforms all published algorithms and the SpMV method in the recent CUSPARSE library based on a set of public matrix benchmarks.**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is of crucial importance in sparse linear algebra as it plays an important role in many numerical and scientific computing applications such as finite difference and finite element based methods. SpMV operation represents the dominant computing cost in those problems and it is very important to improve the efficiency of the SpMV algorithms.

Modern computer architecture has shifted towards the multi-core processor [1], [2] and many-core architectures [3]. The family of graphic processing units (GPU) are among the most powerful many-core computing systems in mass-market use [4]. For instance, the state-of-the-art NVIDIA Kepler K20X GPU with 2688 cores has a peak performance of over 4 TFLOPS versus about 80–100 GFLOPS of Intel i7 series Quad-core CPUs [5], [6]. In addition to the primary use of GPUs in accelerating graphics rendering operations, there has been considerable interest in exploiting GPUs for general purpose computation (GPGPU) [7].

Modern NVIDIA GPUs are throughput-oriented many-core processors that can offer very high peak computational throughput. They favor computations exhibiting sufficient regularity of execution paths and memory access patterns. However, SpMV operations (actually most of BLAS level 1 and level 2 operations for sparse matrices) are typically much less regular in their access patterns and consequently are generally limited purely by bandwidth. Those operations has very low computing to communication ratio and less data sharing, which can be exploited. As a result, they remain a challenge for GPU-based fine-grained parallel computing [5].

Several research efforts have been proposed for parallelizing SpMV on the GPU platforms. There are many formats used for storing sparse matrices. Among them, Compressed Sparse Row (CSR) format has been widely used for many sparse matrices computing especially for the iterative and the direct matrix solving. As a result, we focus on the CSR format in our work. Bell and Garland [8] had a first comprehensive study on this topic and proposed a number of SpMV algorithms on CUDA GPU for different sparse matrices formats, which become part of the cusp library now [9]. Two SpMV algorithms have been proposed for the CSR format in their work. The first algorithm, called row-based $B\&G$ method, assigns one thread per row to compute results, which may not be efficient if the length of rows is large. To further improve its efficiency, the second algorithm, called the warp-based $B\&G$ method, applies one warp (32 threads) to compute a row instead to gain better memory access as warp is the thread scheduling unit in GPU. However, for matrices with many rows of small number of nonzeros, the warp-based $B\&G$ does not perform well. Deng *et al* later proposed [10] an improved SpMV method in which the SpMV operation is first transferred into two vectors, the remaining operations become two vector multiplication and partial summation over rows. However, the method requires two sequential kernel launches, one for element-wise multiplication (or production), which can enjoy fast coalesced memory access, and the other one for partial summation, which still can't avoid irregular memory access due to different length of rows. In the sequel, we call this method the $P\&S$ method.

In this paper, we propose a new SpMV algorithm for the CSR format on GPUs. The new algorithm, called the *segSpMV* method, can be applied to wide computational applications with both structured and unstructured matrices. The new algorithm tries to overcome the irregular memory access problem in the $P\&S$ method. The main idea is to partition rows, which are irregular in general, into a number of fixed-length segments and the length of segments can be determined based on the matrix structures. As a result, both multiplication and summation phases now can enjoy the coalesced memory access and they can be finished in just one kernel launch. The summation phase can also be further improved by using GPU reduction techniques for large segment lengths.

Experimental results show that the resulting SpMV method constantly outperforms all published algorithms on *ALL* the public matrix benchmarks given by Bell and Garland [11]. In some cases, the new algorithm can deliver order of magnitude speedup over existing approaches.

II. Review of Existing SpMV methods on GPUs

Before we present our new SpMV algorithm, we review several relevant SpMV algorithms on GPU platforms. There are many sparse matrices formats such as DIA, ELL, CSR, HYB, PKT, COO with applications ranging from highly structured matrices (DIA, ELL) to unstructured matrices (HYB, COO) [8]. Among them, CSR can be used for both structured and unstructured sparse matrices. As a result, we focus on the CSR format for our new SpMV algorithm.

A. Compressed sparse row (CSR) format

The compressed sparse row (CSR) format is a popular, general-purpose sparse matrix representation. CSR explicitly stores column indices and nonzero values in arrays *col_indices* and *data*. A third array of row pointers, *row_point*, takes the CSR representation as shown in Fig. 1 for a 5×5 sparse matrix. For a $M \times M$ matrix,

This work was funded in part by NSF grants under No. NSF OISE-1130402 and CCF-1017090.

978-1-4799-4132-2/14/$31.00 ©2014 IEEE

the *row_point* with length $M+1$, stores the offset into *data* for the start pointing of each row, with the convention that $row\_point[M] = N_{nz}$, where N_{nz} is the number of nonzeros in the matrix.

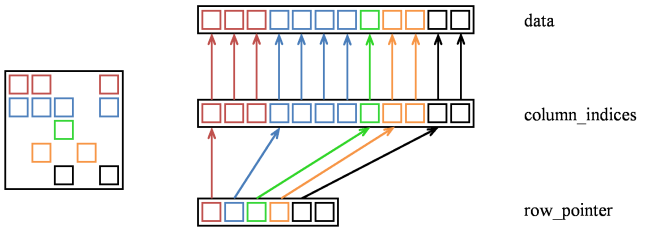

Fig. 1. The CSR format of a sparse matrix

B. Existing sparse matrix vector multiplication algorithms

In this section, we describe SpMV implementations on the the CSR format. The SpMV computing consists of two phases: the first *product* phase, which performs the element-element production between the matrix and the vector, the second *summation* phase adds the results for each row to get the final result.

1) The row-based B&G method: Bell and Garland [8] first proposed a straightforward implementation, in which each row will take care of all the computation (multiplication and summation) by a single thread as shown in Fig. 2. The algorithm only requires one kernel launch (one kernel launch means one CPU-to-GPU invocation). The main drawback of this approach is that each thread will read many sequential data from a *data* vector in the CSR format from the global memory of GPUs, which lead to slow non-coalesced memory access.

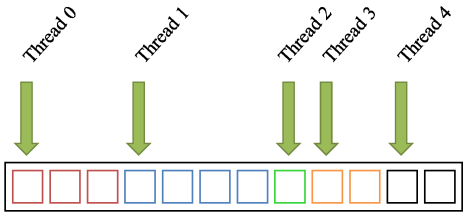

Fig. 2. The row-based $B\&G$ method

2) The warp-based B&G method: The row-based $B\&G$ method is further improved by the warp-based $B\&G$ method in [8] in which one warp is assigned to each row of a matrix. After the multiplication phase, the intra-warp thread reduction is performed to compute the per-thread result. The algorithm is illustrated in Fig. 3. Compared to the row-based $B\&G$ method, its memory accesses can be coalesced because 32 continuous threads in the same warp could work together to load the non-zero elements in one row. This method, however, may suffers low performance when the number of nonzeros in each row is smaller than 32, which can be the case for many finite difference and finite element based methods.

3) The P&S method: Deng *et al* later proposed [10] an improved SpMV method, called $P\&S$ method for many electronic design automation (EDA) related problems. The approach will not directly operate on the CSR data structure. Instead, it creates a new vector, called *expanded vector*, of the same size of the *data*, as shown in Fig. 4. The expanded vector consists of the elements from the multiplication vector $[b1, ..., b3]$ and each element in the vector, says *expended_vector[i]*, corresponds to the element in *data*, which is *data[i]*. They will be multiplied in the production phase.

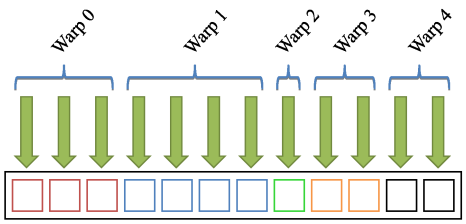

Fig. 3. The warp-based $B\&G$ method

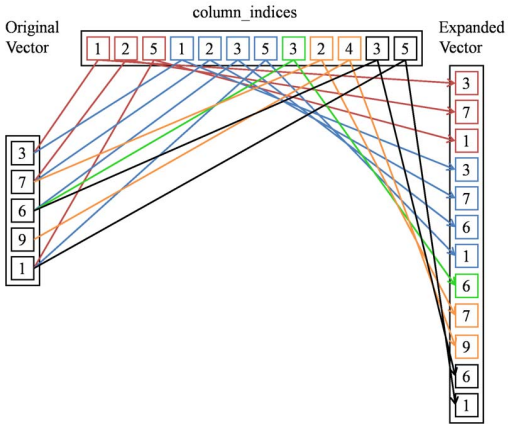

Fig. 4. The vector expansion concept in the $P\&S$ method

After the generation of expande vector, the remaining operations become two vector multiplication and partial summation over rows. However, the method requires two sequential kernel launches, one for element-wise multiplication for the two vectors, which can enjoy fast coalesced memory access. Another one for carrying out partial summation for each row after the vector multiplication as shown in Fig. 5. The second phase, however, can't avoid irregular memory access due to different length of rows.

III. New sparse matrix-vector multiplication method – segSpMV

In this section, we present our new SpMV algorithm, called *segSpMV* method. As we can see, the $P\&S$ method mitigates the irregular memory access for the multiplication phase by using expended vector concept. But for the summation phase, it still suffers the irregular memory access issue as the length of rows are irregular. Using shared memory can partially mitigate this problem. However, given the fact that the shared memory is limited, the number of nonzeros per row can't be too larger. To see this, let's assume that we have 768 threads per block and 16KB shared memory, and every element has 4 byte. Then for every block, we will need to access 48 KB memory to obtain all required data. If we can only have 16KB shared memory (for instance in Tesla T10 GPU), the hit rate (probability of required data in the shared memory) would be only about 33%.

In addition, the $P\&S$ method uses only one thread per row for the summation after loading the data into the shared memory. As a result, the row with less non-zero elements would get the summation result faster compared to the row with more non-zero elements. Therefore, for the different rows, they would store their summation results in different time, which cause the threads in the same warp in a less-coalesced memory access way (in terms of timing). Furthermore, it requires two sequential kernel launches. In the second launch for the

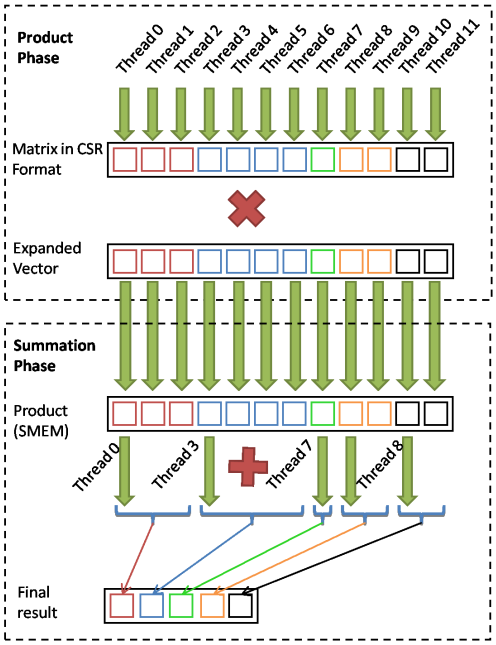

Fig. 5. The $P\&S$ method

summation, one have to load intermediate production result vector and the *row_point* vector from global memory.

In this paper, we present a new SpMV algorithm, which can overcome the aforementioned problems in the existing $P\&S$ method. The new algorithm is also based on the expanded vector concept for the multiplication phase. But different from the $P\&S$ method, the new algorithm can mitigate the irregular memory access problems in the summation phase, and thus lead to more simple implementation and yet better performance. The main idea is to partition the rows into a number of fixed-length regular segments before the operation. The length of the segment typically is selected to be just bigger than the average number of nonzero elements per row in a matrix and they also should be the power of 2 for easy reduction operation. For instance, if the average number of nonzero elements is 14, then segment length $2^4 = 16$ is selected. For rows with more nonzeores than the average number, multiple segments will be needed.

After the segment length is determined, the each row is partitioned into a number of regular segments. If a segment does not fully filled by the elements from the given row, 0 is padded to the rest of the empty elements in the segment as shown in Fig. 6. In this figure, one 0 is padded at the end of seg1 and seg4. We perform this segment-based expansion for both original *data* vector and the *expanded vector*. After this step, the two segment-expanded vectors are sent to GPU global memory for multiplication and addition phases with just one kernel launch as shown in Fig. 6. Note that it takes $O(N_{nz})$ to do the zero padding. In the product phase, each thread first will read two elements from the two segment-expended vectors respectively via the coalesced memory access from the GPU global memory. Then each thread multiplies one pair of elements from the two segment-expanded vectors. But it saves the product result immediately into the shared memory instead. In this case, all the partial product results from all threads are stored in shared memory, which is ready for the second phase of addition operation right away. We note that the new method will never run out of shared memory as the amount of memory needed is 4 times of number of threads in each block as each thread takes care of one segment. So given 1K maximum thread allowed in each block in Tesla C2070 CPU, the maximum memory is just 4K, which is far less than the 49K shared memory in each multiple stream processors(MP). This is also the case for other GPUs as well.

As a result, we do not need to write the product results back to global memory and then read them back again, which leads to one more kernel launch saving. In the addition phase, each thread sums products in one segment and each block is responsible of the same number of segments. The number of non-zero elements in each row may be different, but all segments are with the same length. Compared to the $P\&S$ method, we do not need to check if a data is cached in shared memory and do not need to worry about the low hit rate when the number of non-zero elements per row is large as we make full use of the shared memory and has equivalent 100% hit rate in a sense.

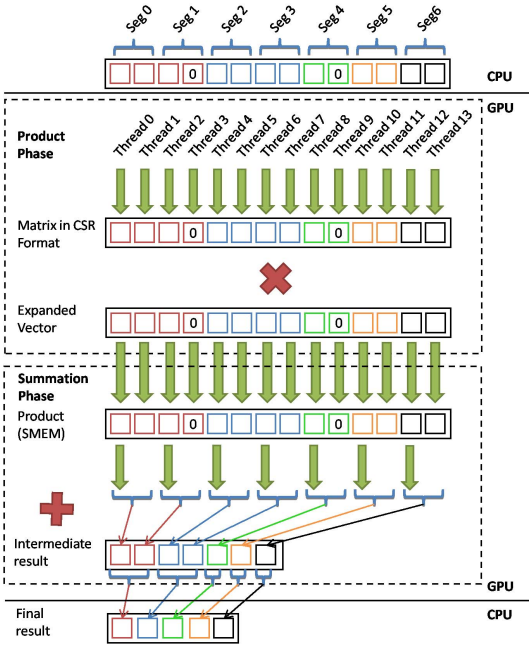

Fig. 6. The proposed segment-SpMV method or segSpMV method

In the summation phase, the new algorithm does not need to check the boundaries of each row any more, which causes the irregular memory access, as it can simply add all the results for each regular segment instead. Because the segment size is fixed, the summation can be very easily done by one thread or by multiple threads via reduction. Also the addition operation will take almost same time for all the threads. We add the $synchronize()$ to ensure all the partial results from each segment finish first before they are written back into global memory using the coalesced memory access. Finally, *segSpMV* adds up the immediate results of segments corresponding to the same row to get the final results in the CPU side, which can be done very efficiently.

IV. Numerical Results and Discussions

The proposed method has been prototyped in CUDA 5.0 and the experimental results where carried out in a Linux server with two 8-Core Xeon E5-2670 CPUs, DDR3-1600 64GB memory. The server also consists of two C2070 GPUs, which serve as the GPU platforms for the proposed algorithms. To perform the comparisons, several algorithms have been implemented or obtained from the published sources as listed below:

- segSpMV, the proposed method where summation is done one thread per segment.

TABLE I
THE PERFORMANCE COMPARISON OVER BELL AND GARLAND MATRICES

Matrices	$B\&G$-s (ms)	$B\&G$-w (ms)	$P\&S$ (ms)	CUSPARSE (ms)	segSpMV (ms)	Speedup o/v $B\&G$-s	Speedup o/v $B\&G$-w	Speedup o/v $P\&S$	Speedup o/v CUSPARSE
webbase-1M	7.662	14.100	11.068	5.775	0.920	8.36	15.39	12.08	6.30
mc2depi	0.435	6.324	0.514	0.632	0.350	1.24	18.07	1.47	1.81
scircuit	0.597	2.355	0.328	0.432	0.250	2.43	9.57	1.33	1.76
mac-econ-fwd500	1.000	3.112	0.496	0.638	0.340	2.93	9.13	1.45	1.87
cop20k-A	2.165	1.794	0.906	0.514	0.336	8.23	6.82	3.44	1.95
Average						4.64	11.80	3.96	2.74
	$B\&G$-s (ms)	$B\&G$-w (ms)	$P\&S$ (ms)	CUSPARSE (ms)	segSpMV-r (ms)	Speedup o/v $B\&G$-s	Speedup o/v $B\&G$-w	Speedup o/v $P\&S$	Speedup o/v CUSPARSE
pwtk	10.035	4.195	4.408	1.623	0.913	10.99	4.59	4.83	1.78
shipsec1	6.590	2.871	2.752	1.138	0.606	10.88	4.74	4.54	1.88
cant	3.680	1.365	1.944	0.639	0.236	15.59	5.78	8.24	2.71
consph	5.809	1.966	3.232	0.891	0.455	12.77	4.32	7.10	1.96
qcd5-4	3.819	1.247	2.218	0.594	0.272	14.04	4.58	8.15	2.18
rma10	4.467	1.464	2.313	0.591	0.345	12.95	4.24	6.70	1.71
pdb1HYS	4.096	1.286	2.300	0.523	0.322	12.72	3.99	7.14	1.62
Average						12.85	4.61	6.67	1.98

TABLE II
THE MATRICES AND THEIR PROPERTIES FROM BELL AND GARLAND [11]

Matrices	size	nzsize	nzperrow	seg_length
webbase-1M	1000005	3105536	3.11	4
mc2depi	525825	2100225	3.99	4
scircuit	170998	958936	5.61	8
mac-econ-fwd500	206500	1273389	6.17	8
cop20k-A	121192	1362087	11.24	16
pwtk	217918	5926171	27.19	32
shipsec1	140874	3977139	28.23	32
cant	62451	2034917	32.58	32
consph	83334	3046907	36.56	32
qcd5-4	49152	1916928	39.00	32
rma10	46835	2374001	50.69	64
pdb1HYS	36417	2190591	60.15	64

- segSpMV-r, the proposed method where summation is performed by reduction.
- $P\&S$, the $P\&S$ method.
- $B\&G$-s, the $B\&G$ method using single thread per row [9].
- $B\&G$-w, the $B\&G$ method using one warp per row [9].

For our segSpMV and the $P\&S$ method, we set the number of thread per blocks to be 256, which is the best choice based on our observation. For shared memory configuration, the new segSpMV method only requires (thread per block)*4 = 1024 bytes per block, so it is always satisfied for C2070 GPU. $P\&S$ method needs much more shared memory when number of nonzeros per row is large. As a result, we set 2048*4 bytes for each block, which is the largest one we can set for C2070 GPU.

We perform the comparison on the set of matrices from Bell and Garland [11] as shown in Table II, in which *nzsize* means number of nonzeros and *nzperrow* is the average number of nozeros per row, *seg_length* is the segment length used for the proposed methods. All the matrices are ranked with increasing number of *nzperrow* from top to bottom and those matrices represent various matrix structures from wild applications.

Table I shows the performance comparison on the matrices in Table II for five algorithms and CUSPARSE library. Since segSpMV and segSpMV-r methods need to get the final results at CPU side, the time of the two methods is the total time of GPU and CPU part. It can be seen that the proposed segSpMV or segSpMV-r methods beat all the other algorithms on *ALL* the matrices cases with various structures. When the segment length is shorter than 32, for segSpMV, the average speedups over $B\&G$-s, $B\&G$-w, $P\&S$ methods and CUSPARSE are 4.64, 11.80, 3.96 and 2.74 respectively. Speedup in some cases such as *webbase-1M* can be order of magnitude faster over three other algorithms. For the matrices with long segment length, segSpMV-r is faster than segSpMV as the two matrices has very large numbers of average nonzeros per row and large segment size (32 or 64) is used. In this case, the speedup over the $B\&G$-w method becomes small as warp-based coalesced memory access shows better performance for matrices with large nonzeros per row. But the fact that the proposed methods constantly beat all the algorithms on *ALL* matrices clearly demonstrates the advantage of the proposed method.

V. CONCLUSION

In this paper, we have proposed a fast parallel sparse matrix-vector multiplication (SpMV) algorithm on GPU platforms. The new SpMV algorithm tries to mitigate the low computing to communication ratio issues in the SpMV operations by regularizing the access patterns during the two operations of the SpMV. Experimental results showed that the resulting SpMV method constantly outperforms all published algorithms and the SpMV method in the CUSPARSE library over the set of public matrix benchmark.

REFERENCES

[1] Intel Corporation, "Intel multi-core processors, making the move to quad-core and beyond (White Paper)," 2006. http://www.intel.com/multi-core.

[2] AMD Inc., "Multi-core processors—the next evolution in computing (White Paper)," 2006. http://multicore.amd.com.

[3] S. Borkar, "Thousand core chips: a technology perspective," in *Proc. Design Automation Conf. (DAC)*, pp. 746–749, 2007.

[4] NVIDIA Corporation, 2011. http://www.nvidia.com.

[5] D. B. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach, 2ed*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2013.

[6] "NVIDIA Tesla's Servers and Workstations." http://www.nvidia.com/object/tesla-servers.html.

[7] D. Göddeke, "General-purpose computation using graphics harware." http://www.gpgpu.org/, 2011.

[8] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[9] "Cusp-library – generic parallel algoritgms for sparse matrix and graph computations." http://code.google.com/p/cusp-library.

[10] Y. Deng, B. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pp. 539–546, 2009.

[11] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA." http://www.nvidia.com/object/nvidia_research_pub_001.html.