Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2016, Article ID 8471283, 12 pages
http://dx.doi.org/10.1155/2016/8471283

Research Article

A Novel CSR-Based Sparse Matrix-Vector Multiplication on GPUs

Guixia He[1] and Jiaquan Gao[2]

[1]*Zhijiang College, Zhejiang University of Technology, Hangzhou 310024, China*
[2]*College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China*

Correspondence should be addressed to Jiaquan Gao; springf12@163.com

Received 4 January 2016; Accepted 27 March 2016

Academic Editor: Veljko Milutinovic

Copyright © 2016 G. He and J. Gao. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Sparse matrix-vector multiplication (SpMV) is an important operation in scientific computations. Compressed sparse row (CSR) is the most frequently used format to store sparse matrices. However, CSR-based SpMVs on graphic processing units (GPUs), for example, CSR-scalar and CSR-vector, usually have poor performance due to irregular memory access patterns. This motivates us to propose a perfect CSR-based SpMV on the GPU that is called PCSR. PCSR involves two kernels and accesses CSR arrays in a fully coalesced manner by introducing a middle array, which greatly alleviates the deficiencies of CSR-scalar (rare coalescing) and CSR-vector (partial coalescing). Test results on a single C2050 GPU show that PCSR fully outperforms CSR-scalar, CSR-vector, and CSRMV and HYBMV in the vendor-tuned CUSPARSE library and is comparable with a most recently proposed CSR-based algorithm, CSR-Adaptive. Furthermore, we extend PCSR on a single GPU to multiple GPUs. Experimental results on four C2050 GPUs show that no matter whether the communication between GPUs is considered or not PCSR on multiple GPUs achieves good performance and has high parallel efficiency.

1. Introduction

Sparse matrix-vector multiplication (SpMV) has proven to be an important operation in scientific computing. It need be accelerated because SpMV represents the dominant cost in many iterative methods for solving large-sized linear systems and eigenvalue problems that arise in a wide variety of scientific and engineering applications [1]. Initial work about accelerating the SpMV on CUDA-enabled GPUs is presented by Bell and Garland [2, 3]. The corresponding implementations in the CUSPARSE [4] and CUSP libraries [5] include optimized codes of the well-known compressed sparse row (CSR), coordinate list (COO), ELLPACK (ELL), hybrid (HYB), and diagonal (DIA) formats. Experimental results show speedups between 1.56 and 12.30 compared to an optimized CPU implementation for a range of sparse matrices.

SpMV is a largely memory bandwidth-bound operation. Reported results indicate that different access patterns to the matrix and vectors on the GPU influence the SpMV performance [2, 3]. The COO, ELL, DIA, and HYB kernels benefit from full coalescing. However, the scalar CSR kernel (CSR-scalar) shows poor performance because of its rarely coalesced memory accesses [3]. The vector CSR kernel (CSR-vector) improves the performance of CSR-scalar by using warps to access the CSR structure in a contiguous but not generally aligned fashion [3], which implies partial coalescing. Since then, researchers have developed many highly efficient CSR-based SpMV implementations on the GPU by optimizing the memory access pattern of the CSR structure. Lu et al. [6] optimize CSR-scalar by padding CSR arrays and achieve 30% improvement of the memory access performance. Dehnavi et al. [7] propose a prefetch-CSR method that partitions the matrix nonzeros to blocks of the same size and distributes them amongst GPU resources. This method obtains a slightly better behavior than CSR-vector by padding rows with zeros to increase data regularity, using parallel reduction techniques, and prefetching data to hide global memory accesses. Furthermore, Dehnavi et al. enhance the performance of the prefetch-CSR method by replacing it with three subkernels [8]. Greathouse and Daga suggest a CSR-Adaptive algorithm that keeps the CSR format intact

and maps well to GPUs [9]. Their implementation efficiently accesses DRAM by streaming data into the local scratchpad memory and dynamically assigns different numbers of rows to each parallel GPU compute unit. In addition, numerous works have proposed for GPUs using the variants of the CSR storage format such as the compressed sparse eXtended [10], bit-representation-optimized compression [11], block CSR [12, 13], and row-grouped CSR [14].

Besides using the variants of CSR, many highly efficient SpMVs on GPUs have been proposed by utilizing the variants of the ELL and COO storage formats such as the ELLPACK-R [15], ELLR-T [16], sliced ELL [13, 17], SELL-C-σ [18], sliced COO [19], and blocked compressed COO [20]. Specialized storage formats provide definitive advantages. However, as many programs use CSR, the conversion from CSR to other storage formats will present a large engineering hurdle and can incur large runtime overheads and require extra storage space. Moreover, CSR-based algorithms generally have a lower memory usage than those that are based on other storage formats such as ELL, DIA, and HYB.

All the above observations motivate us to further investigate how to construct efficient SpMVs on GPUs while keeping CSR intact. In this study, we propose a perfect CSR algorithm, called PCSR, on GPUs. PCSR is composed of two kernels and accesses CSR arrays in a fully coalesced manner. Experimental results on C2050 GPUs show that PCSR outperforms CSR-scalar and CSR-vector and has a better behavior compared to CSRMV and HYBMV in the vendor-tuned CUSPARSE library [4] and a most recently proposed CSR-based algorithm, CSR-Adaptive.

The main contributions in this paper are summarized as follows:

(i) A novel SpMV implementation on a GPU, which keeps CSR intact, is proposed. The proposed algorithm consists of two kernels and alleviates the deficiencies of many existing CSR algorithms that access CSR arrays in a rare or partial coalesced manner.

(ii) Our proposed SpMV algorithm on a GPU is extended to multiple GPUs. Moreover, we suggest two methods to balance the workload among multiple GPUs.

The rest of this paper is organized as follows. Following this introduction, the matrix storage, CUDA architecture, and SpMV are described in Section 2. In Section 3, a new SpMV implementation on a GPU is proposed. Section 4 discusses how to extend the proposed SpMV algorithm on a GPU to multiple GPUs. Experimental results are presented in Section 5. Section 6 contains our conclusions and points to our future research directions.

2. Related Techniques

2.1. Matrix Storage. To take advantage of the large number of zeros in sparse matrices, special storage formats are required. In this study, the compressed sparse row (CSR) format is only considered although there are many varieties of sparse matrix storage formats, such as the ELLPACK (or ITPACK) [21], COO [22], DIA [1], and HYB [3]. Using CSR, an $n \times n$ sparse matrix A with N nonzero elements is stored via three arrays: (1) the array *data* contains all the nonzero entries of A, (2) the array *indices* contains column indices of nonzero entries that are stored in *data*, and (3) entries of the array *ptr* point to the first entry of subsequence rows of A in the arrays *data* and *indices*.

For example, the following matrix

$$A = \begin{bmatrix} 4 & 1 & 0 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 & 1 & 0 \\ 0 & 1 & 4 & 0 & 0 & 1 \\ 1 & 0 & 0 & 4 & 1 & 0 \\ 0 & 1 & 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 0 & 1 & 4 \end{bmatrix} \tag{1}$$

is stored in the CSR format by

$$\begin{aligned} &\textit{data}: \\ &\quad [4\ 1\ 1\ 1\ 4\ 1\ 1\ 1\ 4\ 1\ 1\ 4\ 1\ 1\ 1\ 4\ 1\ 1\ 1\ 4] \\ &\textit{indices}: \\ &\quad [0\ 1\ 3\ 0\ 1\ 2\ 4\ 1\ 2\ 5\ 0\ 3\ 4\ 1\ 3\ 4\ 5\ 2\ 4\ 5] \\ &\textit{ptr}: [0\ 3\ 7\ 10\ 13\ 17\ 20]. \end{aligned} \tag{2}$$

2.2. CUDA Architecture. The compute unified device architecture (CUDA) is a heterogenous computing model that involves both the CPU and the GPU [23]. Executing a parallel program on the GPU using CUDA involves the following: (1) transferring required data to the GPU global memory; (2) launching the GPU kernel; and (3) transferring results back to the host memory. The threads of a kernel are grouped into a grid of thread blocks. The GPU schedules blocks over the multiprocessors according to their available execution capacity. When a block is given to a multiprocessor, it is split in warps that are composed of 32 threads. In the best case, all 32 threads have the same execution path and the instruction is executed concurrently. If not, the execution paths are executed sequentially, which greatly reduces the efficiency. The threads in a block communicate via the fast shared memory, but the threads in different blocks communicate through high-latency global memory. Major challenges in optimizing an application on GPUs are global memory access latency, different execution paths in each warp, communication and synchronization between threads in different blocks, and resource utilization.

2.3. Sparse Matrix-Vector Multiplication. Assume that A is an $n \times n$ sparse matrix and x is a vector of size n, and a sequential version of CSR-based SpMV is described in Algorithm 1. Obviously, the order in which elements of *data*, *indices*, *ptr*, and x are accessed has an important impact on the SpMV performance on GPUs where memory access patterns are crucial.

```
Input: data, indices, ptr, x, n
Output: y
(01) for i ← 0 to n − 1 do
(02)   row_start ← ptr[i];
(03)   row_end ← ptr[i + 1];
(04)   sum ← 0;
(05)   for j ← row_start to row_end − 1 do
(06)     sum += data[j] · x[indices[j]];
(07)   done
(08)   y[i] ← sum;
(09) done
```

Algorithm 1: Sequential SpMV.

3. SpMV on a GPU

In this section, we present a perfect implementation of CSR-based SpMV on the GPU. Different with other related work, the proposed algorithm involves the following two kernels:

(i) *Kernel* 1: calculate the array $v = [v_1, v_2, \ldots, v_N]$, where $v_i = data[i] \cdot x[indices[i]]$, $i = 1, 2, \ldots, N$, and then save it to global memory.

(ii) *Kernel* 2: accumulate element values of v according to the following formula: $\sum_{ptr[j] \leqslant i < ptr[j+1]} v_i$, $j = 0, 1, \ldots, n-1$, and store them to an array y in global memory.

We call the proposed SpMV algorithm PCSR. For simplicity, the symbols used in this study are listed in Table 1.

3.1. Kernel 1. For *Kernel* 1, its detailed procedure is shown in Algorithm 2. We observe that the accesses to two arrays *data* and *indices* in global memory are fully coalesced. However, the vector x in global memory is randomly accessed, which results in decreasing the performance of *Kernel* 1. On the basis of evaluations in [24], the best memory space to place data is the texture memory when randomly accessing the array. Therefore, here texture memory is utilized to place the vector instead of global memory. For the single-precision floating point texture, the fourth step in Algorithm 2 is rewritten as

$$v[tid] \longleftarrow data[tid] \cdot text1Dfetch(floatTexRef, indices[tid]). \quad (3)$$

Because the texture does not support double values, the following function *fetch_double*() is suggested to transfer the int2 value to the double value.

```
(01) __device__ double fetch_double(texture⟨int2, 1⟩t, int i){
(02)   int2 v = tex1Dfetch(t, i);
(03)   return __hiloint2double(v · y, v · x);
(04) }
```

Furthermore, for the double-precision floating point texture, based on the function *fetch_double*(), we rewrite the fourth step in Algorithm 2 as

$$v[tid] \longleftarrow data[tid] \cdot fetch\_double(doubleTexRef, indices[tid]). \quad (4)$$

3.2. Kernel 2. *Kernel* 2 accumulates element values of v that is obtained by *Kernel* 1 and its detailed procedure is shown in Algorithm 3. This kernel is mainly composed of the following three stages:

(i) In the first stage, the array *ptr* in global memory is piecewise assembled into shared memory *ptr_s* of each thread block in parallel. Each thread for a thread block is only responsible for loading an element value of *ptr* into *ptr_s* except for thread 0 (see lines (05)-(06) in Algorithm 3). The detailed procedure is illustrated in Figure 1. We can see that the accesses to *ptr* are aligned.

(ii) The second stage loads element values of v in global memory from the position *ptr_s*[0] to the position *ptr_s*[TB] into shared memory *v_s* for each thread block. The assembling procedure is illustrated in Figure 2. In this case, the access to v is fully coalesced.

(iii) The third stage accumulates element values of *v_s*, as shown in Figure 3. The accumulation is highly efficient due to the utilization of two shared memory arrays *ptr_s* and *v_s*.

Obviously, *Kernel* 2 benefits from shared memory. Using the shared memory, not only are the data accessed fast, but also the accesses to data are coalesced.

From the above procedures for PCSR, we observe that PCSR needs additional global memory spaces to store a middle array v besides storing CSR arrays *data*, *indices*, and *ptr*. Saving data into v in *Kernel* 1 and loading data from v in *Kernel* 2 to a degree decrease the performance of PCSR. However, PCSR benefits from the middle array v because introducing v makes it access CSR arrays *data*, *indices*, and *ptr* in a fully coalesced manner. This greatly improves the speed of accessing CSR arrays and alleviates the principal deficiencies of CSR-scalar (rare coalescing) and CSR-vector (partial coalescing).

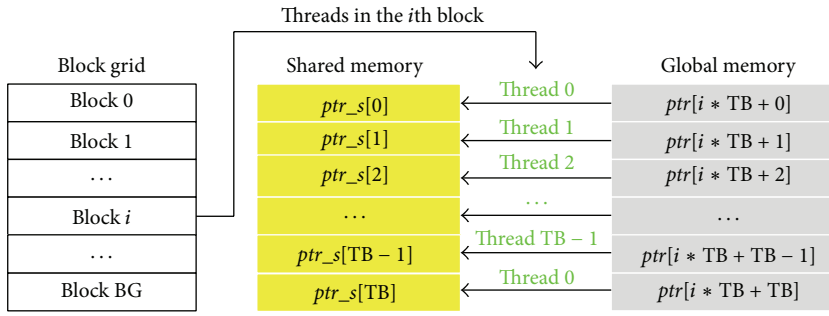


Figure 1: First stage of *Kernel* 2.

Table 1: Symbols used in this study.

Symbol	Description
A	Sparse matrix
x	Input vector
y	Output vector
n	Size of the input and output vectors
N	Number of nonzero elements in A
threadsPerBlock (TB)	Number of threads per block
blocksPerGrid (BG)	Number of blocks per grid
elementsPerThread	Number of elements calculated by each thread
sizeSharedMemory	Size of shared memory
M	Number of GPUs

Input: $data, indices, x, N$
CUDA-specific variables:
(i) threadId.x: a thread
(ii) blockId.x: a block
(iii) blockDim.x: number of threads per block
(iv) gridDim.x: number of blocks per grid
Output: v
(01) $tid \leftarrow$ threadId.x + blockId.x · blockDim.x;
(02) $icr \leftarrow$ blockDim.x · gridDim.x;
(03) **while** $tid < N$
(04) $v[tid] \leftarrow data[tid] \cdot x[indices[tid]]$;
(05) $tid \mathrel{+}= icr$;
(06) **end while**

Algorithm 2: *Kernel* 1.

4. SpMV on Multiple GPUs

In this section, we will present how to extend PCSR on a single GPU to multiple GPUs. Note that the case of multiple GPUs in a single node (single PC) is only discussed because of its good expansibility (e.g., also used in the multi-CPU and multi-GPU heterogeneous platform). To balance the workload among multiple GPUs, the following two methods can be applied:

(1) For the first method, the matrix is equally partitioned into M (number of GPUs) submatrices according to the matrix rows. Each submatrix is assigned to one GPU, and each GPU is only responsible for computing the assigned submatrix multiplication with the complete input vector.

(2) For the second method, the matrix is equally partitioned into M submatrices according to the number of nonzero elements. Each GPU only calculates a submatrix multiplication with the complete input vector.

In most cases, two partitioned methods mentioned above are similar. However, for some exceptional cases, for example, most nonzero elements are involved in a few rows for a matrix, the partitioned submatrices that are obtained by the first method have distinct difference of nonzero elements, and those that are obtained by the second method have different rows. Which method is the preferred one for PCSR?

If each GPU has the complete input vector, PCSR on multiple GPUs will not need to communicate between GPUs. In fact, SpMV is often applied to a large number of iterative methods where the sparse matrix is iteratively multiplied by the input and output vectors. Therefore, if each GPU only includes a part of the input vector before SpMV, the communication between GPUs will be required in order to execute PCSR. Here PCSR implements the communication between GPUs using NVIDIA GPUDirect.

5. Experimental Results

5.1. Experimental Setup. In this section, we test the performance of PCSR. All test matrices come from the University of Florida Sparse Matrix Collection [25]. Their properties are summarized in Table 2.

All algorithms are executed on one machine which is equipped with an Intel Xeon Quad-Core CPU and four NVIDIA Tesla C2050 GPUs. Our source codes are compiled and executed using the CUDA toolkit 6.5 under GNU/Linux Ubuntu v10.04.1. The performance is measured in terms of GFlop/s (second) or GByte/s (second).

5.2. Single GPU. We compare PCSR with CSR-scalar, CSR-vector, CSRMV, HYBMV, and CSR-Adaptive. CSR-scalar and

Input: v, ptr
CUDA-specific variables:
(i) threadId.x: a thread
(ii) blockId.x: a block
(iii) blockDim.x: number of threads per block
(iv) gridDim.x: number of blocks per grid
Output: y
(01) define shared memory v_s with size $sizeSharedMemory$
(02) define shared memory ptr_s with size $(threadsPerBlock + 1)$
(03) $gid \leftarrow$ threadIdx.x + blockIdx.x × blockDim.x;
(04) $tid \leftarrow$ threadIdx.x;
/* Load ptr into the shared memory ptr_s */
(05) $ptr\_s[tid] \leftarrow ptr[gid]$;
(06) **if** $tid == 0$ **then** $ptr\_s[threadsPerBlock] \leftarrow ptr[gid + threadsPerBlock]$;
(07) __syncthreads();
(08) $temp \leftarrow (ptr\_s[threadsPerBlock] - ptr\_s[0])/threadsPerBlock + 1$;
(09) $nlen \leftarrow \min(temp \cdot threadsPerBlock, sizeSharedMemory)$;
(10) $sum \leftarrow 0.0$; $maxlen \leftarrow ptr\_s[threadsPerBlock]$;
(11) **for** $i \leftarrow ptr\_s[0]$ **to** $maxlen - 1$ **with** i += $nlen$ **do**
(12) $index \leftarrow i + tid$;
(13) __syncthreads();
/* Load v into the shared memory v_s */
(14) **for** $j \leftarrow 0$ **to** $nlen/threadsPerBlock - 1$ **do**
(15) **if** $index < nlen$ **then**
(16) $v\_s[tid + j \cdot threadsPerBlock] \leftarrow v[index]$;
(17) $index$ += $threadsPerBlock$;
(18) **end**
(19) **done**
(20) __syncthreads();
/* Perform a scalar-style reduction */
(21) **if** $(ptr\_s[tid + 1] \leqslant i$ **or** $ptr\_s[tid] > i + nlen - 1)$ is *false* **then**
(22) $row\_s \leftarrow \max(ptr\_s[tid] - i, 0)$;
(23) $row\_e \leftarrow \min(ptr\_s[tid + 1] - i, nlen)$;
(24) **for** $j \leftarrow row\_s$ **to** $row\_e - 1$ **do**
(25) sum += $v\_s[j]$;
(26) **done**
(27) **end**
(28) **done**
(29) y[gid] $\leftarrow sum$;

Algorithm 3: *Kernel* 2.

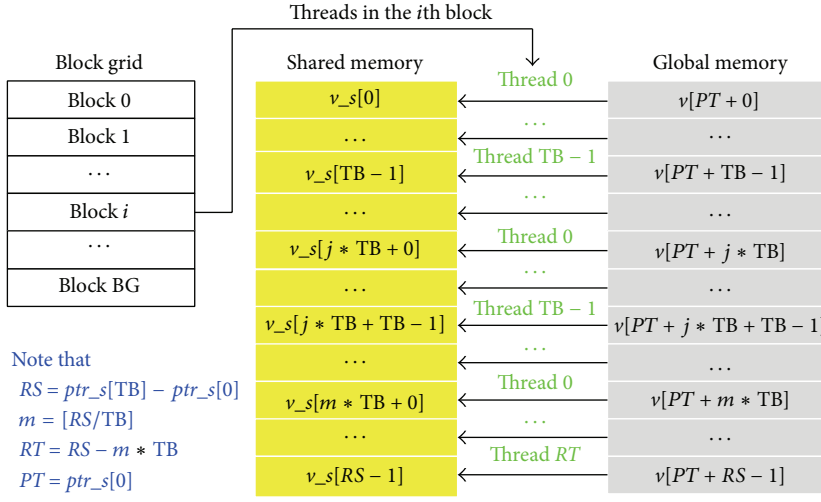


Figure 2: Second stage of *Kernel* 2.

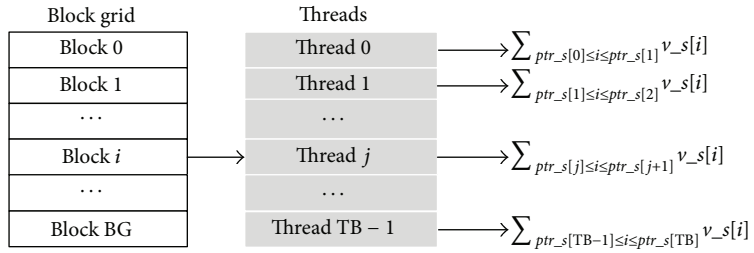


Figure 3: Third stage of *Kernel* 2.

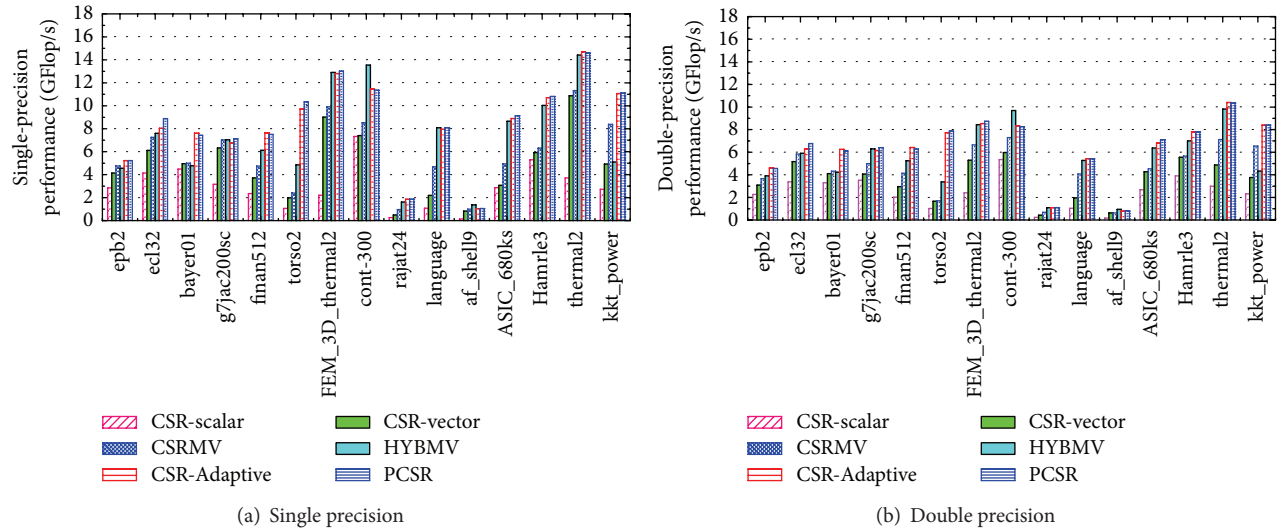
Table 2: Properties of test matrices.

Name	Rows	Nonzeros (nz)	nz/row	Description
epb2	25,228	175,027	6.94	Thermal problem
ecl32	51,993	380,415	7.32	Semiconductor device
bayer01	57,735	277,774	4.81	Chemical process
g7jac200sc	59,310	837,936	14.13	Economic problem
finan512	74,752	335,872	4.49	Economic problem
2cubes_sphere	101,492	1,647,264	16.23	Electromagnetics
torso2	115,967	1,033,473	8.91	2D/3D problem
FEM_3D_thermal2	147,900	3,489,300	23.59	Nonlinear thermal
scircuit	170,998	958,936	5.61	Circuit simulation
cont-300	180,895	988,195	5.46	Optimization problem
Ga41As41H72	268,096	18,488,476	68.96	Pseudopotential method
F1	343,791	26,837,113	78.06	Stiffness matrix
rajat24	358,172	1,948,235	5.44	Circuit simulation
language	399,130	1,216,334	3.05	Directed graph
af_shell9	504,855	17,588,845	34.84	Sheet metal forming
ASIC_680ks	682,712	2,329,176	3.41	Circuit simulation
ecology2	999,999	4,995,991	5.00	Circuit theory
Hamrle3	1,447,360	5,514,242	3.81	Circuit simulation
thermal2	1,228,045	8,580,313	6.99	Unstructured FEM
cage14	1,505,785	27,130,349	18.01	DNA electrophoresis
Transport	1,602,111	23,500,731	14.67	Structural problem
G3_circuit	1,585,478	7,660,826	4.83	Circuit simulation
kkt_power	2,063,494	12,771,361	6.19	Optimization problem
CurlCurl_4	2,380,515	26,515,867	11.14	Model reduction
memchip	2,707,524	14,810,202	5.47	Circuit simulation
Freescale1	3,428,755	18,920,347	5.52	Circuit simulation

CSR-vector in the CUSP library [5] are chosen in order to show the effects of accessing CSR arrays in a fully coalesced manner in PCSR. CSRMV in the CUSPARSE library [4] is a representative of CSR-based SpMV algorithms on the GPU. HYBMV in the CUSPARSE library [4] is a finely tuned HYB-based SpMV algorithm on the GPU and usually has a better behavior than many existing SpMV algorithms. CSR-Adaptive is a most recently proposed CSR-based algorithm [9].

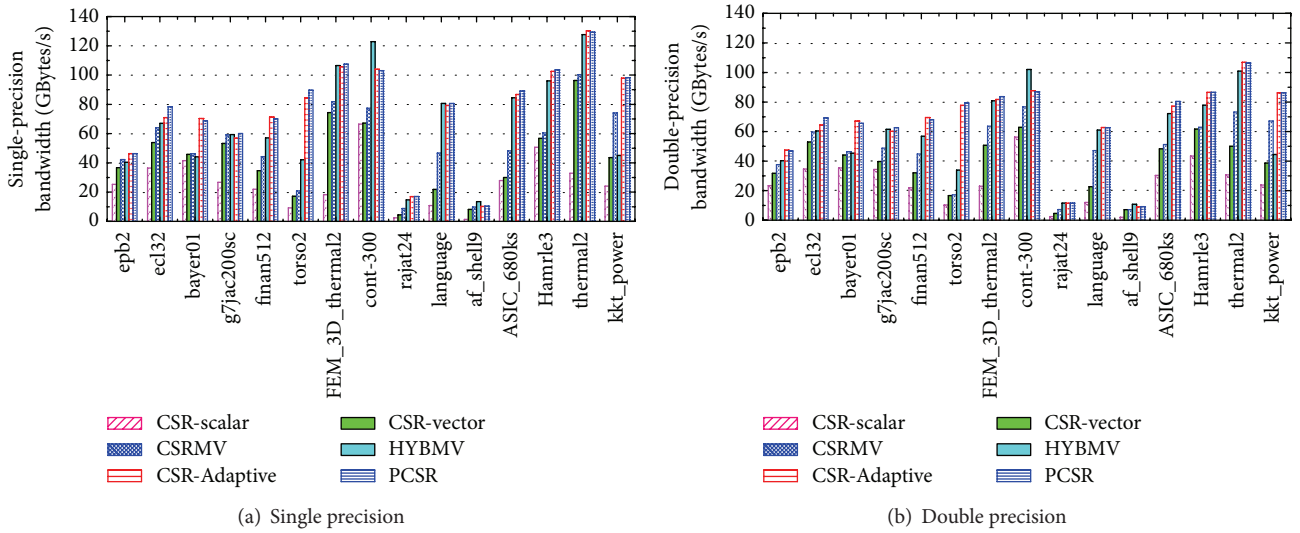
We select 15 sparse matrices with distinct sizes ranging from 25,228 to 2,063,494 as our test matrices. Figure 4 shows the single-precision and double-precision performance results in terms of GFlop/s of CSR-scalar, CSR-vector, CSRMV, HYBMV, CSR-Adaptive, and PCSR on a Tesla C2050. GFlop/s values in Figure 4 are calculated on the basis of the assumption of two Flops per nonzero entry for a matrix [3, 13]. In Figure 5, the measured memory bandwidth results for single precision and double precision are reported.

5.2.1. Single Precision. From Figure 4(a), we observe that PCSR achieves high performance for all the matrices in the single-precision mode. In most cases, the performance of over 9 GFlops/s can be obtained. Moreover, PCSR outperforms CSR-scalar, CSR-vector, and CSRMV for all test cases, and average speedups of 4.24x, 2.18x, and 1.62x compared to CSR-scalar, CSR-vector, and CSRMV can be obtained, respectively. Furthermore, PCSR has a slightly better behavior than HYBMV for all the matrices except for af_shell9 and



(a) Single precision (b) Double precision

Figure 4: Performance of all algorithms on a Tesla C2050.



(a) Single precision (b) Double precision

Figure 5: Effective bandwidth results of all algorithms on a Tesla C2050.

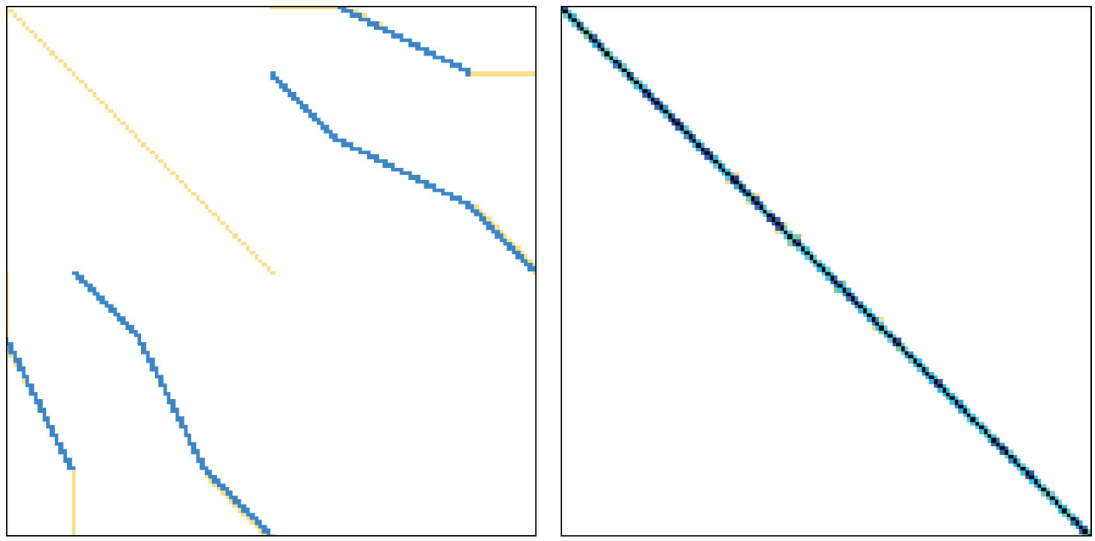
cont-300. The average speedup is 1.22x compared to HYBMV. Figure 6 shows the visualization of af_shell9 and cont-300. We can find that af_shell9 and cont-300 have a similar structure, and each row for two matrices has a very similar number of nonzero elements, which is suitable to be stored in the ELL section of the HYB format. Particularly, PCSR and CSR-Adaptive have close performance. The average performance of PCSR is nearly 1.05 times faster than CSR-Adaptive.

Furthermore, PCSR almost has the best memory bandwidth utilization among all algorithms for all the matrices except for af_shell9 and cont-300 (Figure 5(a)). The maximum memory bandwidth of PCSR exceeds 128 GBytes/s, which is about 90 percent of peak theoretical memory bandwidth for the Tesla C2050. Based on the performance metrics [26], we can conclude that PCSR achieves good performance and has high parallelism.

5.2.2. Double Precision. From Figures 4(b) and 5(b), we see that, for all algorithms, both the double-precision performance and memory bandwidth utilization are smaller than the corresponding single-precision values due to the slow software-based operation. PCSR is still better than CSR-scalar, CSR-vector, and CSRMV and slightly outperforms HYBMV and CSR-Adaptive for all the matrices. The average speedup of PCSR is 3.33x compared to CSR-scalar, 1.98x compared to CSR-vector, 1.57x compared to CSRMV, 1.15x compared to HYBMV, and 1.03x compared to CSR-Adaptive. The maximum memory bandwidth of PCSR exceeds 108 GBytes/s, which is about 75 percent of peak theoretical memory bandwidth for the Tesla C2050.

5.3. Multiple GPUs

5.3.1. PCSR Performance without Communication. Here we take the double-precision mode, for example, to test the PCSR



(a) cont-300 (b) af_shell9

Figure 6: Visualization of the af_shell9 and cont-300 matrix.

Table 3: Comparison of PCSRI and PCSRII without communication on two GPUs.

Matrix	ET (GPU)	PCSRI (2 GPUs)			PCSRII (2 GPUs)		
		ET	SD	PE	ET	SD	PE
2cubes_sphere	0.4444	0.2670	0.0178	83.21	**0.2640**	**0.0156**	**84.17**
scircuit	0.3484	0.2413	0.0322	72.20	**0.2250**	**0.0207**	**77.41**
Ga41As41H72	4.2387	2.3084	0.0446	91.81	**2.3018**	**0.0432**	**92.07**
F1	6.5544	3.8865	0.7012	84.32	**3.5710**	**0.2484**	**91.77**
ASIC_680ks	0.8196	0.4567	0.0126	89.72	**0.4566**	**0.0021**	**89.74**
ecology2	1.2321	0.6665	**0.0140**	92.42	**0.6654**	0.0152	**92.58**
Hamrle3	1.7684	0.9651	0.0478	91.61	**0.9208**	**5.00E − 05**	**96.02**
thermal2	2.0708	1.0559	0.0056	98.06	**1.0558**	**0.0045**	**98.06**
cage14	5.9177	3.4757	0.5417	85.13	**3.1548**	**0.0458**	**93.78**
Transport	4.7305	2.4665	**0.0391**	95.89	**2.4655**	0.0407	**95.93**
G3_circuit	1.9731	**1.0485**	**0.0364**	**94.08**	1.1061	0.1148	89.18
kkt_power	4.3465	2.7916	0.7454	77.85	**2.2252**	**0.0439**	**97.66**
CurlCurl_4	5.1605	2.7107	0.0347	95.18	**2.7075**	**0.0244**	**95.30**
memchip	3.8257	2.1905	0.3393	87.32	**2.0975**	**0.2175**	**91.19**
Freescale1	5.0524	3.0235	0.5719	83.55	**2.8175**	**0.2811**	**89.66**

performance on multiple GPUs without considering communication. We call PCSR with the first method and PCSR with the second method PCSR-I and PCSR-II, respectively. Some large-sized test matrices in Table 2 are used. The execution time comparison of PCSRI and PCSRII on two and four Tesla C2050 GPUs is listed in Tables 3 and 4, respectively. In Tables 3 and 4, ET, SD, and PE stand for the execution time, standard deviation, and parallel efficiency, respectively. The time unit is millisecond (ms). Figures 7 and 8 show the parallel efficiency of PCSRI and PCSRII on two and four GPUs, respectively.

On two GPUs, we observe that PCSRII has better parallel efficiency than PCSRI for all the matrices except for G3_circuit from Table 3 and Figure 7. The maximum, average, and minimum parallel efficiency of PCSRII are 98.06%, 91.64%, and 77.41%, which wholly outperform the corresponding maximum, average, and minimum parallel efficiency of PCSRI 98.06%, 88.16%, and 72.20%. Moreover, PCSRII has a smaller standard deviation than PCSRI for all the matrices except for ecology2, Transport, and G3_circuit. This implies that the workload balance on two GPUs for the second method is advantageous over that for the first method.

On four GPUs, for the parallel efficiency and standard deviation, PCSRII outperforms PCSRI for all the matrices except for G3_circuit (Table 4 and Figure 8). The maximum, average, and minimum parallel efficiency of PCSRII for all the matrices are 96.35%, 85.14%, and 64.17% and are advantageous over the corresponding maximum, average, and minimum parallel efficiency of PCSRI 96.21%, 78.89%,

Table 4: Comparison of PCSRI and PCSRII without communication on four GPUs.

Matrix	ET (GPU)	PCSRI (4 GPUs)			PCSRII (4 GPUs)		
		ET	SD	PE	ET	SD	PE
2cubes_sphere	0.4444	0.1560	0.0132	71.23	**0.1527**	**0.0111**	**72.78**
scircuit	0.3484	0.1453	0.0262	59.94	**0.1357**	**0.0130**	**64.17**
Ga41As41H72	4.2387	1.6123	0.7268	65.72	**1.3410**	**0.1846**	**79.02**
F1	6.5544	2.5240	0.6827	64.92	**1.9121**	**0.1900**	**85.69**
ASIC_680ks	0.8196	0.2944	0.0298	69.59	**0.2887**	**0.0264**	**70.98**
ecology2	1.2321	0.3593	0.0160	85.72	**0.3554**	**0.0141**	**86.67**
Hamrle3	1.7684	0.5114	0.0307	86.45	**0.4775**	**0.0125**	**92.59**
thermal2	2.0708	0.5553	0.0271	93.22	**0.5546**	**0.0255**	**93.33**
cage14	5.9177	1.8126	0.3334	81.62	**1.5386**	**0.0188**	**96.15**
Transport	4.7305	1.2292	0.0270	96.21	**1.2275**	**0.0158**	**96.35**
G3_circuit	1.9731	**0.5804**	**0.0489**	**84.99**	0.6195	0.0790	79.63
kkt_power	4.3465	1.4974	0.5147	72.57	**1.1584**	**0.0418**	**93.80**
CurlCurl_4	5.1605	1.3554	0.0153	95.18	**1.3501**	**0.0111**	**95.56**
memchip	3.8257	1.1439	0.1741	83.61	**1.1175**	**0.1223**	**85.59**
Freescale1	5.0524	1.7588	0.4039	71.81	**1.4806**	**0.1843**	**85.31**

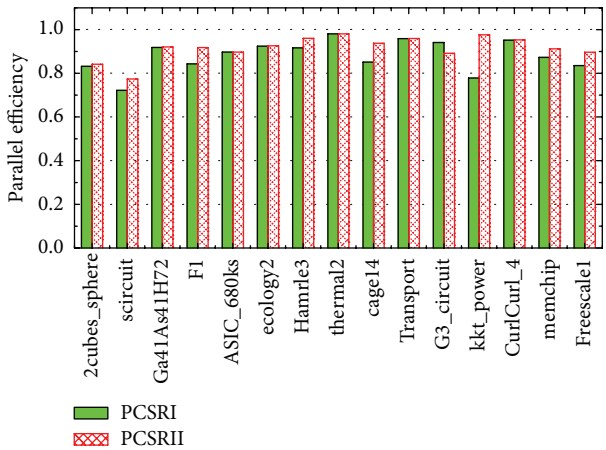


Figure 7: Parallel efficiency of PCSRI and PCSRII without communication on two GPUs.

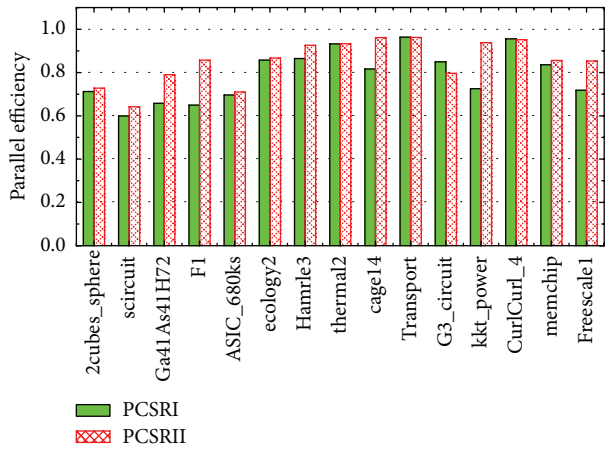


Figure 8: Parallel efficiency of PCSRI and PCSRII without communication on four GPUs.

and 59.94%. Particularly, for Ga41As41H72, F1, cage14, kkt_power, and Freescale1, the parallel efficiency of PCSRII is almost 1.2 times that obtained by PCSRI.

On the basis of the above observations, we conclude that PCSRII has high performance and is on the whole better than PCSRI. For PCSR on multiple GPUs, the second method is our preferred one.

5.3.2. PCSR Performance with Communication. We still take the double-precision mode, for example, to test the PCSR performance on multiple GPUs with considering communication. PCSR with the first method and PCSR with the second method are still called PCSR-I and PCSR-II, respectively. The same test matrices as in the above experiment are utilized. The execution time comparison of PCSRI and PCSRII on two and four Tesla C2050 GPUs is listed in Tables 5 and 6, respectively. The time unit is ms. ET, SD, and PE in Tables 5 and 6 are as the same as those in Tables 3 and 4. Figures 9 and 10 show PCSRI and PCSRII parallel efficiency on two and four GPUs, respectively.

On two GPUs, PCSRI and PCSRII have almost close parallel efficiency for most matrices (Figure 9 and Table 5). As a comparison, PCSRII slightly outperforms PCSRI. The maximum, average, and minimum parallel efficiency of PCSRII for all the matrices are 96.34%, 88.51%, and 80.44% and are advantageous over the corresponding maximum, average, and minimum parallel efficiency of PCSRI 96.05%, 86.03%, and 73.57%.

On four GPUs, for the parallel efficiency and standard deviation, PCSRII is better than PCSRI for all the matrices except that PCSRI has slightly good parallel efficiency for thermal2, G3_circuit, and Hamrle3 and slightly small standard deviation for thermal2, G3_circuit, ecology2, and CurlCurl_4 (Figure 10 and Table 6). The maximum, average,

Table 5: Comparison of PCSRI and PCSRII with communication on two GPUs.

Matrix	ET (GPU)	PCSRI (2 GPUs)			PCSRII (2 GPUs)		
		ET	SD	PE	ET	SD	PE
2cubes_sphere	0.4444	**0.2494**	**6.00E − 04**	**89.09**	0.2503	5.00*E* − 04	88.75
scircuit	0.3484	0.2234	0.0154	77.95	**0.2165**	**0.0070**	**80.44**
Ga41As41H72	4.2387	**2.3516**	**0.0030**	**90.12**	2.3795	0.0521	89.07
F1	6.5544	3.9252	0.6948	83.49	**3.6076**	**0.2392**	**90.84**
ASIC_680ks	0.8196	**0.4890**	**0.0113**	**83.80**	0.4998	0.0178	81.99
ecology2	1.2321	0.6865	3.00*E* − 04	89.74	**0.6863**	**8.00E − 04**	**89.76**
Hamrle3	1.7684	1.0221	0.0209	86.50	**1.0066**	**0.0170**	**87.84**
thermal2	2.0708	1.1403	0.0230	90.80	**1.1402**	**0.0203**	**90.81**
cage14	5.9177	3.5756	0.5644	82.75	**3.2244**	**0.0196**	**91.76**
Transport	4.7305	2.4623	0.0203	96.05	**2.4550**	**0.0183**	**96.34**
G3_circuit	1.9731	**1.1215**	**0.0189**	**87.96**	1.1766	0.0896	83.84
kkt_power	4.3465	2.9539	0.6973	73.57	**2.4459**	**0.0356**	**88.85**
CurlCurl_4	5.1605	2.7064	0.0092	95.34	**2.7049**	**1.00E − 03**	**95.39**
memchip	3.8257	2.3218	0.3467	82.39	**2.2243**	**0.1973**	**85.99**
Freescale1	5.0524	3.1216	0.5868	80.92	**2.9367**	**0.3199**	**86.02**

Table 6: Comparison of PCSRI and PCSRII with communication on four GPUs.

Matrix	ET (GPU)	PCSRI (4 GPUs)			PCSRII (4 GPUs)		
		ET	SD	PE	ET	SD	PE
2cubes_sphere	0.4444	0.1567	0.0052	70.89	**0.1531**	**0.0028**	**72.54**
scircuit	0.3484	0.1544	0.0204	56.39	**0.1495**	**0.0073**	**58.27**
Ga41As41H72	4.2387	1.7157	0.7909	61.76	**1.4154**	**0.2178**	**74.87**
F1	6.5544	2.1149	0.3833	77.48	**2.0022**	**0.1941**	**81.84**
ASIC_680ks	0.8196	0.3449	0.0187	59.39	**0.3423**	**0.0147**	**59.87**
ecology2	1.2321	0.4257	**0.0048**	72.35	**0.4257**	0.0056	**72.35**
Hamrle3	1.7684	**0.6231**	0.0087	**70.95**	0.6297	**0.0085**	70.21
thermal2	2.0708	**0.6922**	**0.0267**	**74.78**	0.6959	0.0269	74.39
cage14	5.9177	1.9339	0.3442	76.50	**1.6417**	**0.0067**	**90.12**
Transport	4.7305	1.3323	0.0279	88.77	**1.3217**	**0.0070**	**89.48**
G3_circuit	1.9731	**0.7234**	**0.0408**	**68.19**	0.7458	0.0620	66.14
kkt_power	4.3465	1.7277	0.5495	62.89	**1.3791**	**0.0305**	**78.79**
CurlCurl_4	5.1605	1.5065	**0.0253**	85.63	**1.5004**	0.8789	**85.99**
memchip	3.8257	1.3804	0.1768	69.29	**1.3051**	**0.1029**	**73.28**
Freescale1	5.0524	2.0711	0.4342	60.98	**1.8193**	**0.2262**	**69.43**

and minimum parallel efficiency of PCSRII for all the matrices are 90.12%, 74.50%, and 58.27%, which are better than the corresponding maximum, average, and minimum parallel efficiency of PCSRI 88.77%, 65.69%, and 56.39%.

Therefore, compared to PCSRI and PCSRII without communication, although the performance of PCSRI and PCSRII with communication decreases due to the influence of communication, they still achieve significant performance. Because PCSRII overall outperforms PCSRI for all test matrices, the second method in this case is still our preferred one for PCSR on multiple GPUs.

6. Conclusion

In this study, we propose a novel CSR-based SpMV on GPUs (PCSR). Experimental results show that our proposed PCSR on a GPU is better than CSR-scalar and CSR-vector in the CUSP library and CSRMV and HYBMV in the CUSPARSE library and a most recently proposed CSR-based algorithm, CSR-Adaptive. To achieve high performance on multiple GPUs for PCSR, we present two matrix-partitioned methods to balance the workload among multiple GPUs. We observe that PCSR can show good performance with and

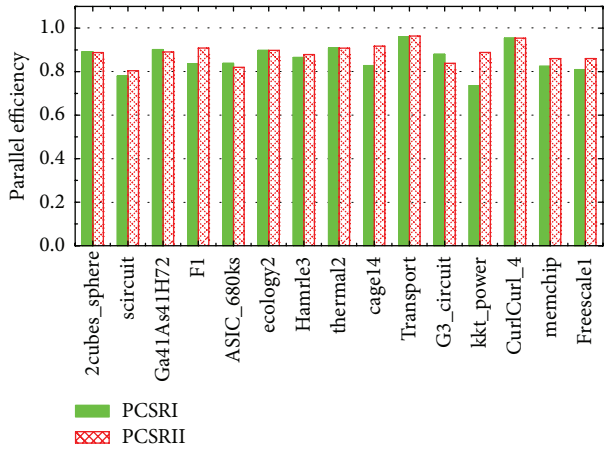


Figure 9: Parallel efficiency of PCSRI and PCSRII with communication on two GPUs.

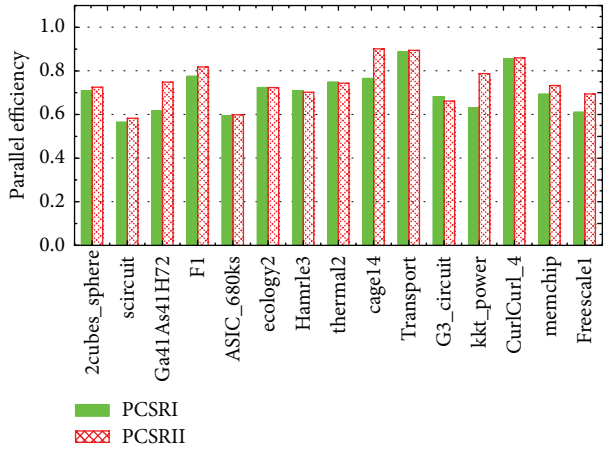


Figure 10: Parallel efficiency of PCSRI and PCSRII with communication on four GPUs.

without considering communication using the two matrix-partitioned methods. As a comparison, the second method is our preferred one.

Next, we will further do research in this area and develop other novel SpMVs on GPUs. In particular, the future work will apply PCSR to some well-known iterative methods and thus solve the scientific and engineering problems.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

The research has been supported by the Chinese Natural Science Foundation under Grant no. 61379017.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, Pa, USA, 2nd edition, 2003.

[2] N. Bell and M. Garland, "Efficient Sparse Matrix-vector Multiplication on CUDA," Tech. Rep., NVIDIA, 2008.

[3] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, pp. 14–19, Portland, Ore, USA, November 2009.

[4] NVIDIA, CUSPARSE Library 6.5, 2015, https://developer.nvidia.com/cusparse.

[5] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations, version 0.5.1," 2015, http://cusp-library.googlecode.com.

[6] F. Lu, J. Song, F. Yin, and X. Zhu, "Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters," *Computer Physics Communications*, vol. 183, no. 6, pp. 1172–1181, 2012.

[7] M. M. Dehnavi, D. M. Fernández, and D. Giannacopoulos, "Finite-element sparse matrix vector multiplication on graphic processing units," *IEEE Transactions on Magnetics*, vol. 46, no. 8, pp. 2982–2985, 2010.

[8] M. M. Dehnavi, D. M. Fernández, and D. Giannacopoulos, "Enhancing the performance of conjugate gradient solvers on graphic processing units," *IEEE Transactions on Magnetics*, vol. 47, no. 5, pp. 1162–1165, 2011.

[9] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, pp. 769–780, New Orleans, La, USA, November 2014.

[10] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, "An extended compression format for the optimization of sparse matrix-vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 10, pp. 1930–1940, 2013.

[11] W. T. Tang, W. J. Tan, R. Ray et al., "Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*, pp. 1–12, Denver, Colo, USA, November 2013.

[12] M. Verschoor and A. C. Jalba, "Analysis and performance estimation of the conjugate gradient method on multiple GPUs," *Parallel Computing*, vol. 38, no. 10-11, pp. 552–575, 2012.

[13] 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)*, pp. 115–126, ACM, Bangalore, India, January 2010.

[14] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, "MPI-CUDA sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner," *Computers & Fluids*, vol. 92, pp. 244–252, 2014.

[15] F. Vázquez, J. J. Fernández, and E. M. Garzón, "A new approach for sparse matrix vector product on NVIDIA GPUs," *Concurrency Computation Practice and Experience*, vol. 23, no. 8, pp. 815–826, 2011.

[16] F. Vázquez, J. J. Fernández, and E. M. Garzón, "Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach," *Parallel Computing*, vol. 38, no. 8, pp. 408–420, 2012.

[17] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *High Performance Embedded Architectures and*

Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25–27, 2010. Proceedings, pp. 111–125, Springer, Berlin, Germany, 2010.

[18] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[19] H.-V. Dang and B. Schmidt, "CUDA-enabled sparse matrix-vector multiplication on GPUs using atomic operations," *Parallel Computing. Systems & Applications*, vol. 39, no. 11, pp. 737–750, 2013.

[20] S. Yan, C. Li, Y. Zhang, and H. Zhou, "YaSpMV: yet another SpMV framework on GPUs," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*, pp. 107–118, February 2014.

[21] D. R. Kincaid and D. M. Young, "A brief review of the ITPACK project," *Journal of Computational and Applied Mathematics*, vol. 24, no. 1-2, pp. 121–127, 1988.

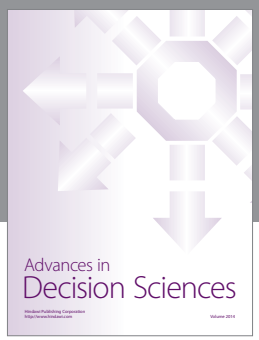
[22] G. Blelloch, M. Heroux, and M. Zagha, "Segmented operations for sparse matrix computation on vector multiprocessor," Tech. Rep., School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa, USA, 1993.

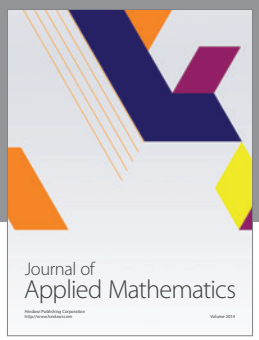
[23] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

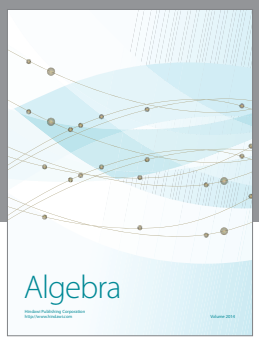
[24] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105–118, 2011.

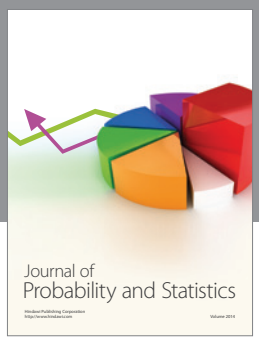
[25] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

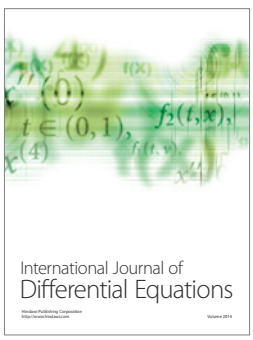
[26] NVIDIA, "CUDA C Programming Guide 6.5," 2015, http://docs.nvidia.com/cuda/cuda-c-programming-guide.

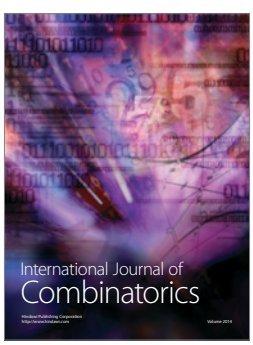


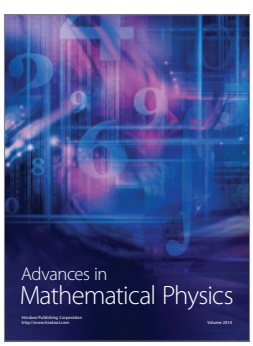


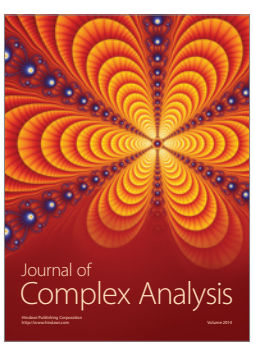


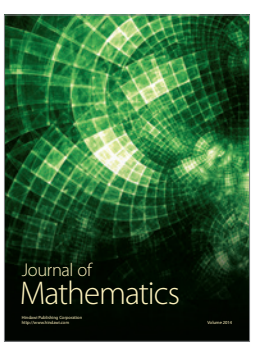


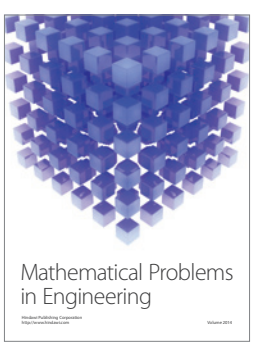


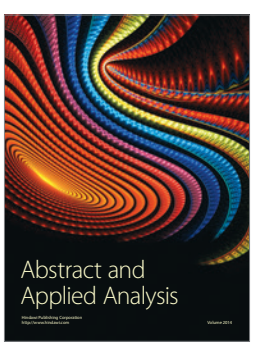


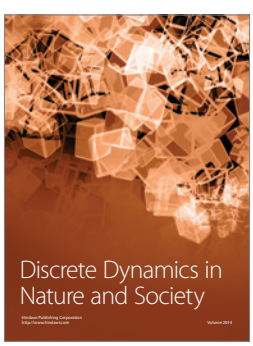


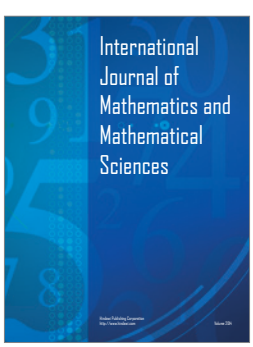


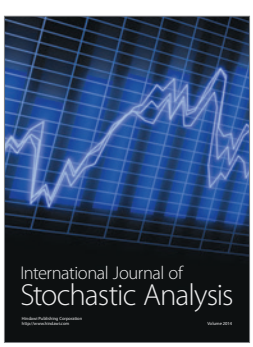












Hindawi

Submit your manuscripts at
http://www.hindawi.com