Research Article

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

Guixia He¹ and Jiaquan Gao²

¹Zhijiang College, Zhejiang University of Technology, Hangzhou 310024, China
²College 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 CSRMMV and HYBMMV 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 needs to 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-CSF 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-CSF 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 xTended [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

\[
A = \begin{bmatrix}
4 & 1 & 0 & 1 & 0 & 0 \\
1 & 4 & 1 & 0 & 1 & 0 \\
0 & 1 & 4 & 0 & 0 & 1 \\
1 & 0 & 0 & 4 & 1 & 0 \\
0 & 1 & 0 & 1 & 4 & 1 \\
0 & 0 & 1 & 0 & 1 & 4
\end{bmatrix}
\]

is stored in the CSR format by

\[
data: [4 1 1 1 1 4 1 1 1 4 1 1 1 4 1 1 1 4]
\]

\[
indices: [0 1 3 0 1 2 4 1 2 5 0 3 4 1 3 4 5 2 4 5]
\]

\[
ptr: [0 3 7 10 13 17 20].
\]

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.
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 \( v_i = data[i] \cdot x[indices[i]], \) for \( i = 1, 2, \ldots, N, \) and then save it to global memory.

(ii) **Kernel 2**: accumulate element values of \( v \) according to the following formula: \( \sum_{i \in \text{ptr}(j) \cap \text{ptr}(j+1)} v_i, \) \( j = 0, 1, \ldots, n - 1, \) 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

\[
v[tid] \\ \leftarrow data[tid] \cdot tex1Dfetch(floatTexRef, indices[tid]) .
\]

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

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

\[
v[tid] \\ \leftarrow data[tid] \cdot fetch\_double(doubleTexRef, indices[tid]).
\]

### 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).
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: \( n, \text{ptr} \)

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 \( \text{sizeSharedMemory} \)
(02) define shared memory \( \text{ptr}_s \) with size \( \text{threadsPerBlock} + 1 \)
(03) \( gid \leftarrow \text{threadIdx.x} + \text{blockIdx.x} \times \text{blockDim.x} \);
(04) \( tid \leftarrow \text{threadIdx.x} \);
(05) \( \text{ptr}_s[tid] \leftarrow \text{ptr}[gid] \);
(06) \( \text{if } tid == 0 \text{ then } \text{ptr}_s[\text{threadsPerBlock}] \leftarrow \text{ptr}[\text{gridDim.x}] \);
(07) \( \text{syncthreads}() \);
(08) \( \text{temp} \leftarrow (\text{ptr}_s[\text{threadsPerBlock}] - \text{ptr}_s[0]) / \text{threadsPerBlock} + 1 \);
(09) \( \text{nlen} \leftarrow \text{min} (\text{temp} \cdot \text{threadsPerBlock}, \text{sizeSharedMemory}) \);
(10) \( \text{sum} \leftarrow 0.0; \text{maxlen} \leftarrow \text{ptr}_s[\text{threadsPerBlock}] \);
(11) \( \text{for } i \leftarrow \text{ptr}_s[0] \text{ to maxlen } - 1 \text{ with } i += \text{nlen} \text{ do} \)
(12) \( \text{index} \leftarrow i + tid \);
(13) \( \text{syncthreads}() \);
(14) \( \text{Load } v \text{ into the shared memory } v_s^* / \)
(15) \( \text{for } j \leftarrow 0 \text{ to maxlen } / \text{threadsPerBlock } - 1 \text{ do} \)
(16) \( v_s[\text{tid} + j \cdot \text{threadsPerBlock}] \leftarrow v[\text{index}] \);
(17) \( \text{index} += \text{threadsPerBlock} \);
(18) \( \text{end} \)
(19) \( \text{done} \)
(20) \( \text{syncthreads}() \);
(21) \( \text{Perform a scalar-style reduction } / \)
(22) \( \text{if } (\text{ptr}_s[\text{tid} + 1] < i \text{ or } \text{ptr}_s[\text{tid}] > i + \text{nlen} - 1) \text{ is false then} \)
(23) \( \text{row}_s \leftarrow \text{max} (\text{ptr}_s[\text{tid}] - i, 0) \);
(24) \( \text{row}_e \leftarrow \text{min} (\text{ptr}_s[\text{tid} + 1] - i, \text{nlen}) \);
(25) \( \text{for } j \leftarrow \text{row}_s \text{ to row}_e - 1 \text{ do} \)
(26) \( \text{sum} += v_s[j] \);
(27) \( \text{done} \)
(28) \( \text{done} \)
(29) \( y[gid] \leftarrow \text{sum} \);
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, CSRVM, 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 CSRVM for all test cases, and average speedups of 4.24x, 2.18x, and 1.62x compared to CSR-scalar, CSR-vector, and CSRVM can be obtained, respectively. Furthermore, PCSR has a slightly better behavior than HYBMV for all the matrices except for af_shell9 and
5.2 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...
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 PCSR 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 PCSR and PCSRII on two and four GPUs, respectively.

On two GPUs, we observe that PCSRII has better parallel efficiency than PCSR 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 PCSR 98.06%, 88.16%, and 72.20%. Moreover, PCSRII has a smaller standard deviation than PCSR 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 PCSR 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 96.21%, 78.89%, and 64.17% and are advantageous over the corresponding maximum, average, and minimum parallel efficiency of PCSR 96.21%, 78.89%, 64.17%.

---

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

<table>
<thead>
<tr>
<th>Matrix</th>
<th>ET (GPU)</th>
<th>PCSRI (2 GPUs)</th>
<th>PCSRII (2 GPUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ET</td>
<td>SD</td>
<td>PE</td>
</tr>
<tr>
<td>2cubes_sphere</td>
<td>0.4444</td>
<td>0.2670</td>
<td>0.0178</td>
</tr>
<tr>
<td>scircuit</td>
<td>0.3484</td>
<td>0.2413</td>
<td>0.0322</td>
</tr>
<tr>
<td>Ga41As41H72</td>
<td>4.2387</td>
<td>2.3084</td>
<td>0.0446</td>
</tr>
<tr>
<td>Fl</td>
<td>6.5544</td>
<td>3.8865</td>
<td>0.7012</td>
</tr>
<tr>
<td>ASIC_680ks</td>
<td>0.8196</td>
<td>0.4567</td>
<td>0.0126</td>
</tr>
<tr>
<td>ecology2</td>
<td>1.2321</td>
<td>0.6665</td>
<td>0.0140</td>
</tr>
<tr>
<td>Hamrle3</td>
<td>1.7684</td>
<td>0.9651</td>
<td>0.0478</td>
</tr>
<tr>
<td>thermal2</td>
<td>2.0708</td>
<td>1.0559</td>
<td>0.0056</td>
</tr>
<tr>
<td>cage14</td>
<td>5.9177</td>
<td>3.4757</td>
<td>0.5417</td>
</tr>
<tr>
<td>Transport</td>
<td>4.7305</td>
<td>2.4665</td>
<td>0.0391</td>
</tr>
<tr>
<td>G3_circuit</td>
<td>1.9731</td>
<td>1.0485</td>
<td>0.0364</td>
</tr>
<tr>
<td>kkt_power</td>
<td>4.3465</td>
<td>2.7916</td>
<td>0.7454</td>
</tr>
<tr>
<td>CurlCurl_4</td>
<td>5.1605</td>
<td>2.7107</td>
<td>0.0347</td>
</tr>
<tr>
<td>memchip</td>
<td>3.8257</td>
<td>2.1905</td>
<td>0.3393</td>
</tr>
<tr>
<td>Freescale1</td>
<td>5.0524</td>
<td>3.0235</td>
<td>0.5719</td>
</tr>
</tbody>
</table>
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 PCSR 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 PCSR and PCSRII parallel efficiency on two and four GPUs, respectively.

On two GPUs, PCSR and PCSRII have almost close parallel efficiency for most matrices (Figure 9 and Table 5). As a comparison, PCSRII slightly outperforms PCSR. 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 PCSR for all the matrices except that PCSR has slightly good parallel efficiency for thermal2, G3_circuit, and Hamrle3 and slightly small standard deviation for thermal2, G3_circuit, ecology2, and CurlCurl4 (Figure 10 and Table 6). The maximum, average,
## Table 5: Comparison of PCSRI and PCSRII with communication on two GPUs.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>ET (GPU)</th>
<th>PCSRI (2 GPUs)</th>
<th>PCSRII (2 GPUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ET</td>
<td>SD</td>
<td>PE</td>
</tr>
<tr>
<td>2cubes, sphere</td>
<td>0.4444</td>
<td>0.2494</td>
<td>6.00E-04</td>
</tr>
<tr>
<td>scircuit</td>
<td>0.3484</td>
<td>0.2234</td>
<td>0.0154</td>
</tr>
<tr>
<td>Ga₄₁As₄₁H₇₂</td>
<td>4.2387</td>
<td>2.3516</td>
<td>0.0030</td>
</tr>
<tr>
<td>F₁</td>
<td>6.5544</td>
<td>3.9252</td>
<td>0.6948</td>
</tr>
<tr>
<td>ASIC₆₈₀ks</td>
<td>0.8196</td>
<td>0.4890</td>
<td>0.0113</td>
</tr>
<tr>
<td>ecology2</td>
<td>1.2321</td>
<td>0.6865</td>
<td>3.00E-04</td>
</tr>
<tr>
<td>Hamrle3</td>
<td>1.7684</td>
<td>1.0221</td>
<td>0.0209</td>
</tr>
<tr>
<td>thermal2</td>
<td>2.0708</td>
<td>1.1403</td>
<td>0.0230</td>
</tr>
<tr>
<td>cage4</td>
<td>5.9177</td>
<td>3.5756</td>
<td>0.5644</td>
</tr>
<tr>
<td>Transport</td>
<td>4.7305</td>
<td>2.4623</td>
<td>0.0203</td>
</tr>
<tr>
<td>G₃_circuit</td>
<td>1.9731</td>
<td>1.1215</td>
<td>0.0189</td>
</tr>
<tr>
<td>kkt_power</td>
<td>4.3465</td>
<td>2.9539</td>
<td>0.6973</td>
</tr>
<tr>
<td>CurlCurl₄</td>
<td>5.1605</td>
<td>2.7064</td>
<td>0.0092</td>
</tr>
<tr>
<td>memchip</td>
<td>3.8257</td>
<td>2.3218</td>
<td>0.3467</td>
</tr>
<tr>
<td>Freescale1</td>
<td>5.0524</td>
<td>3.1216</td>
<td>0.5868</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Matrix</th>
<th>ET (GPU)</th>
<th>PCSRI (4 GPUs)</th>
<th>PCSRII (4 GPUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ET</td>
<td>SD</td>
<td>PE</td>
</tr>
<tr>
<td>2cubes, sphere</td>
<td>0.4444</td>
<td>0.1567</td>
<td>0.0052</td>
</tr>
<tr>
<td>scircuit</td>
<td>0.3484</td>
<td>0.1544</td>
<td>0.0204</td>
</tr>
<tr>
<td>Ga₄₁As₄₁H₇₂</td>
<td>4.2387</td>
<td>1.7157</td>
<td>0.7909</td>
</tr>
<tr>
<td>F₁</td>
<td>6.5544</td>
<td>2.1149</td>
<td>0.3833</td>
</tr>
<tr>
<td>ASIC₆₈₀ks</td>
<td>0.8196</td>
<td>0.3449</td>
<td>0.0187</td>
</tr>
<tr>
<td>ecology2</td>
<td>1.2321</td>
<td>0.4257</td>
<td>0.0048</td>
</tr>
<tr>
<td>Hamrle3</td>
<td>1.7684</td>
<td>0.6231</td>
<td>0.0087</td>
</tr>
<tr>
<td>thermal2</td>
<td>2.0708</td>
<td>0.6922</td>
<td>0.0267</td>
</tr>
<tr>
<td>cage4</td>
<td>5.9177</td>
<td>1.9339</td>
<td>0.3442</td>
</tr>
<tr>
<td>Transport</td>
<td>4.7305</td>
<td>1.3323</td>
<td>0.0279</td>
</tr>
<tr>
<td>G₃_circuit</td>
<td>1.9731</td>
<td>0.7234</td>
<td>0.0408</td>
</tr>
<tr>
<td>kkt_power</td>
<td>4.3465</td>
<td>1.7277</td>
<td>0.5495</td>
</tr>
<tr>
<td>CurlCurl₄</td>
<td>5.1605</td>
<td>1.5065</td>
<td>0.0253</td>
</tr>
<tr>
<td>memchip</td>
<td>3.8257</td>
<td>1.3804</td>
<td>0.1768</td>
</tr>
<tr>
<td>Freescale1</td>
<td>5.0524</td>
<td>2.0711</td>
<td>0.4342</td>
</tr>
</tbody>
</table>

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
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


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


