Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2016, Article ID 4596943, 14 pages
http://dx.doi.org/10.1155/2016/4596943

Research Article

Efficient CSR-Based Sparse Matrix-Vector Multiplication on GPU

Jiaquan Gao,[1] Panpan Qi,[2] and Guixia He[3]

[1]*School of Computer Science and Technology, Nanjing Normal University, Nanjing 210023, China*
[2]*College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China*
[3]*Zhijiang College, Zhejiang University of Technology, Hangzhou 310024, China*

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

Received 9 July 2016; Revised 1 September 2016; Accepted 29 September 2016

Academic Editor: Sebastian López

Sparse matrix-vector multiplication (SpMV) is an important operation in computational science and needs be accelerated because it often represents the dominant cost in many widely used iterative methods and eigenvalue problems. We achieve this objective by proposing a novel SpMV algorithm based on the compressed sparse row (CSR) on the GPU. Our method dynamically assigns different numbers of rows to each thread block and executes different optimization implementations on the basis of the number of rows it involves for each block. The process of accesses to the CSR arrays is fully coalesced, and the GPU's DRAM bandwidth is efficiently utilized by loading data into the shared memory, which alleviates the bottleneck of many existing CSR-based algorithms (i.e., CSR-scalar and CSR-vector). Test results on C2050 and K20c GPUs show that our method outperforms a perfect-CSR algorithm that inspires our work, the vendor tuned CUSPARSE V6.5 and CUSP V0.5.1, and three popular algorithms clSpMV, CSR5, and CSR-Adaptive.

1. Introduction

Given their many-core structures, graphics processing units (GPUs) have sufficient computation power for scientific computations. Processing big data by using GPUs has drawn considerable attention over the recent years. Following the introduction of the compute unified device architecture (CUDA), a programming model that supports the joint CPU/GPU execution of applications, by NVIDIA in 2007 [1], GPUs have become strong competitors as general-purpose parallel programming systems.

Sparse matrix-vector multiplication (SpMV) has proven to be of great importance in computational science. It represents the dominant cost in iterative methods for solving linear systems and eigenvalue problems which arise in a wide variety of scientific and engineering applications [2–5]. Recently, with the increasing size of matrices, GPU-accelerated SpMVs have attracted considerable attention.

SpMV performance is heavily dependent on the format that is used to store the sparse matrix in the memory. Compressed sparse row (CSR) has historically been a frequently used format because it efficiently compresses both structured and unstructured matrices. The naive CSR-based parallel SpMV, known as CSR-scalar, assigns each row of the sparse matrix to a separate thread [6]. However, the memory-access pattern of CSR-scalar is rarely coalesced, which results in disappointing performance. CSR-vector [7, 8] improves the performance of CSR-scalar by using warps to access the CSR structure in a contiguous but not generally aligned fashion, which implies partial coalescing. Since then, researchers have developed many highly efficient CSR-based SpMV implementation techniques on the GPU by optimizing the memory-access pattern of the CSR structure [9–19].

Among the existing CSR-based algorithms on the GPU, a representative, called a perfect-CSR algorithm (PCSR) [13], achieves high performance by fully coalesced accesses to the CSR arrays. PCSR involves two kernels. The first kernel is used to perform many parallel coalesced loads from the CSR arrays and place values into global memory *temp*. The second kernel is composed of two stages. The first stage is to load the global memory *temp* into the shared memory *temp_s* in a fully coalesced manner. Next, the second stage performs a

scalar-style reduction (with each thread working on a single row), and each row in *temp_s* is reduced to an output value. We can observe that the process of loading and reading the global memory *temp* is coalesced in PCSR but it still needs some cost. If a matrix has a great number of nonzero values, the operation that is similar to the vector sum in the first kernel will bottleneck the PCSR performance [20]. Moreover, when each row has large and significantly different number of nonzero values for a matrix, PCSR may leave many threads inactive during the scalar-style reduction process for each block and the thread with a long row often has a slower reduction speed than that with a short row. This greatly decreases the reduction efficiency.

These observations motivate us to enhance the performance of PCSR. We propose an improved PCSR called IPCSR to calculate SpMV on the GPU. IPCSR reduces two kernels in PCSR to one kernel while keeping the coalesced loads of the CSR arrays intact and saves the cost of loading and reading global memory *temp*. IPCSR only includes two stages. In the first stage, IPCSR calculates dynamically the suitable number of rows for each block and quickly loads values from the CSR arrays into the shared memory *temp_s*. This stage merges operations in the first kernel of PCSR and the first stage of the second kernel of PCSR. This process of loads is coalesced and effectively utilizes the GPU's DRAM bandwidth, alleviating the bottleneck of PCSR. IPCSR supports two reduction approaches: a scalar-style reduction (with each thread working on a single row) and a multiple scalar-style reduction (with multiple threads working on a single row). How many threads are assigned to a single row for each block is dynamically determined by a decision tree. Thus, IPCSR improves the reduction efficiency by keeping threads active as much as possible.

IPCSR loses efficiency when a block operates a row, and it becomes inoperative if a row has more values than can be allocated in the shared memory. To overcome this limitation, we specially design a highly efficient algorithm for this case and can dynamically determine which block executes IPCSR with a set of rows or a long row. Our experiments on a set of 20 sparse matrices show that IPCSR outperforms PCSR for all test cases. IPCSR achieves average performance improvement of 1.76x (single precision) and 1.53x (double precision) over PCSR on the C2050 GPU and 1.79x (single precision) and 1.56x (double precision) over PCSR on the K20c GPU. In addition, compared to CUSPARSE V6.5 [21], our proposed IPCSR achieves performance improvement by up to 2.76x on average in single precision and 1.98x on average in double precision on the C2050 GPU and 3.52x on average in single precision and 2.52x on average in double precision on the K20c GPU. Compared to CUSP V0.5.1 [22], which chooses the best of six algorithms that include COO, CSR-scalar, CSR-vector, DIA, ELL, and HYB, IPCSR achieves a performance gain of up to 1.43x on average in single precision and 1.33x on average in double precision on the C2050 GPU and 1.75x on average in single precision and 1.62x on average in double precision on the K20c GPU. Compared to clSpMV [23], which combines advantages of many existing algorithms, IPCSR achieves performance improvement by up to 1.55x on average in single precision and 1.22x on average in double precision on the C2050 GPU and 1.56x on average in single precision and 1.26x on average in double precision on the K20c GPU. Compared to CSR5 [18], IPCSR achieves average performance improvement of 1.21x (single precision) and 1.11x (double precision) over PCSR on the C2050 GPU and 1.30x (single precision) and 1.20x (double precision) over PCSR on the K20c GPU. IPCSR has a slightly better behavior than CSR-Adaptive [17] and achieves performance improvement by up to 1.02x on average in single precision and 1.01x on average in double precision on the C2050 GPU and 1.07x on average in single precision and 1.08x on average in double precision on the K20c GPU.

The main contributions in this paper are summarized as follows:

(i) We propose a novel CSR-based SpMV on the GPU. Our proposed algorithm is inspired by the work of Gao et al. [13] but achieves performance improvement by up to 1.76x on average on the C2050 GPU and 1.79x on average on the K20c GPU compared to the work of Gao et al.

(ii) In our proposed algorithm, each block may have different number of rows, and the optimal number of rows each block involves is dynamically calculated.

(iii) Each block can dynamically determine whether to execute our proposed algorithm with a set of rows or a long row on the basis of the number of rows it involves.

The rest of this paper is organized as follows. Section 2 addresses the related work. Section 3 introduces the CSR format and the PCSR algorithm. Section 4 details our IPCSR algorithm. Experimental results are presented in Section 5. Section 6 summarizes our conclusions and suggestions for future research.

2. Related Work

Sparse matrix-vector multiplication (SpMV) is so important that there has existed a great deal of work on accelerating it. We only discuss the most relevant ones here. Initial work about accelerating the SpMV on the CUDA-enabled GPU is presented by Bell and Garland [7]. They implement several well-known formats including DIA, ELL, CSR, and COO and a new hybrid format HYB on the GPU. Each storage format has its ideal matrix type. They conclude that the best performance of SpMV on the GPU depends on the storage format and advocate the column-major ELL and HYB. These conclusions are carried forward to the CUSP [22] library, which relies heavily on HYB for SpMV.

A lot of work has been proposed for GPUs using the variants of the CSR, ELL, and COO formats such as the compressed sparse eXtended [24], bit-representation-optimized compression [25], block CSR [26], ELLPACK-R [27], sliced ELL [28, 29], SELL-C-σ [30], sliced COO [31], and blocked compressed COO [32]. Specialized storage formats provide definitive advantages. However, as a great number of software programs use CSR, the conversion from CSR to these other formats will present a large engineering hurdle and can incur large runtime overheads and require extra storage space.

In [7, 8], Bell and Garland declare that CSR-scalar loses efficiency due to its rarely coalesced memory accesses. CSR-vector improves the performance of CSR-scalar, but it accesses the CSR structure in a contiguous but not generally aligned fashion, which implies partial coalescing. Compared to CSR-scalar and CSR-vector, ELL, DIA, and HYB implementation techniques on the GPU benefit from full coalescing. Following the idea of optimizing the memory-access pattern of the CSR structure, researchers have developed many algorithms. Lu et al. [33] optimize CSR-scalar by padding the CSR arrays and achieve 30% improvement of the memory-access performance. In [10], Dehnavi et al. propose a prefetch-CSR method that partitions the matrix nonzeros into blocks of the same size and distributes them among GPU resources. The 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, authors increase the performance of this method by replacing it with three subkernels in [11]. Gao et al. [13] propose a perfect-CSR (PCSR) algorithm on the GPU, which consists of two kernels. This algorithm implements the fully coalesced accesses to the CSR arrays by introducing a middle array. Greathouse and Daga suggest a novel algorithm, CSR-Adaptive, which keeps the CSR format intact and maps well to GPUs [14]. Thus, authors enhance CSR-Adaptive and make it work well for both regular and irregular matrices while keeping the CSR format unchanged [17]. Liu and Schmidt present LightSpMV, a novel fast CUDA-compatible SpMV algorithm using the standard CSR format, which achieves high parallelism by benefiting from the fine-grained dynamic distribution of matrix rows over warps/vectors [16]. Other related work can be found in [9, 12, 18].

Our proposed algorithm is inspired by Gao et al.'s work [13]. It alleviates the bottleneck of PCSR that is suggested by Gao et al. while keeping fully coalesced accesses to the CSR arrays and thus outperforms PCSR.

3. The CSR Format and PCSR

3.1. The CSR Format. The compressed sparse row (CSR) format is a popular, general-purpose sparse matrix representation. CSR stores a sparse matrix A via three arrays: (1) the array AA contains all the nonzero entries of A, (2) the array JA contains column indices of the nonzero entries stored in AA, and (3) entries of the array IA point to the first entry of subsequence rows of A in the arrays AA and JA.

For example, the matrix

$$A = \begin{bmatrix} 4 & 1 & 0 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 & 1 & 0 \\ 0 & 1 & 4 & 0 & 0 & 1 \\ 1 & 0 & 0 & 4 & 1 & 0 \\ 0 & 1 & 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 0 & 1 & 4 \end{bmatrix} \quad (1)$$

Input: AA, JA, x, $total\_nonzeros$;
Output: $temp$;
(1) $tid \leftarrow threadIdx.x + blockIdx.x * blockDim.x$;
(2) $icr \leftarrow blockDim.x * gridDim.x$;
(3) **while** $tid < total\_nonzeros$
(4) $\quad temp[tid] \leftarrow AA[tid] * x[JA[tid]]$;
(5) $\quad tid \mathrel{+}= icr$;
(6) **end while**

Algorithm 1: Kernel I in PCSR.

is stored in the CSR format by

$$\begin{aligned} &AA: \\ &\quad [4\ 1\ 1\ 1\ 4\ 1\ 1\ 1\ 4\ 1\ 1\ 4\ 1\ 1\ 1\ 4\ 1\ 1\ 1\ 4], \\ &JA: \\ &\quad [0\ 1\ 3\ 0\ 1\ 2\ 4\ 1\ 2\ 5\ 0\ 3\ 4\ 1\ 3\ 4\ 5\ 2\ 4\ 5], \\ &IA: [0\ 3\ 7\ 10\ 13\ 17\ 20]. \end{aligned} \quad (2)$$

3.2. PCSR. PCSR is CSR-based SpMV implementation on the GPU and involves two kernels. Algorithm 1 shows the main procedure of the first kernel. This kernel is to load values from the CSR arrays into the global memory $temp$ in a coalesced manner. Each thread only calculates a nonzero entry. Algorithm 2 shows the main procedure of the second kernel. We observe that the second kernel is composed of two stages. The first stage performs many parallel coalesced loads from $temp$ and quickly places values into the shared memory $temp\_s$. The second stage completes a scalar-style reduction (with each thread working on a single row). Each row in $temp\_s$ is reduced to an output value.

4. Improved PCSR

On the basis of the above discussions, we propose an improved PCSR on the GPU, called IPCSR, as shown in Algorithm 3. Compared to PCSR, IPCSR reduces two kernels to one kernel and only involves two stages. The first stage is to have each thread block perform many parallel coalesced loads from the JA and AA arrays. This quickly places values into the shared memory $temp\_s$ (see lines (7)–(9) in Algorithm 3). This stage combines operations in the first kernel of PCSR (see lines (3)–(6) in Algorithm 1) and the first stage of the second kernel of PCSR (see lines (12)–(17) in Algorithm 2). The second stage, similar to the second stage in the second kernel of PCSR, completes a scalar-style reduction (with each thread working on a single row).

For IPCSR, the size of shared memory is set as follows for a given nt (number of threads per block):

$$SHARED\_SIZE = \begin{cases} \dfrac{N^{mem}/\min\left(N^{td}/nt, N^{tb}\right)}{4}, & \text{for single precision} \\ \dfrac{N^{mem}/\min\left(N^{td}/nt, N^{tb}\right)/4}{2}, & \text{for double precision,} \end{cases} \quad (3)$$

```
Input: temp, IA;
Output: res;
(1)  tid ← threadIdx.x;
(2)  gid ← threadIdx.x + blockIdx.x * blockDim.x;
(3)  _shared_temp_s[];
(4)  _shared_IA_s[];
(5)  IA_s[tid] ← IA[gid];
(6)  if(tid == 0)
         IA_s[THREADS_BLOCK] ← IA[gid + THREADS_BLOCK];
(7)  _syncthreads();
(8)  sum ← 0; i_s ← IA_s[0]; i_e ← IA_s[THREADS_BLOCK];
(9)  for i ← i_s to i_e − 1 with i += SHARED_SIZE do
(10)     index ← i + tid;
(11)     _syncthreads();
         //Load temp into the shared memory temp_s
(12)     for j ← 0 to SHARED_SIZE/THREADS_BLOCK − 1 do
(13)        if index < SHARED_SIZE then
(14)           temp_s[tid + j * THREADS_BLOCK] ← temp[index];
(15)           index += THREADS_BLOCK;
(16)        end
(17)     done
(18)     _syncthreads();
         //Perform a scalar-style reduction
(19)     if (IA_s[tid + 1] <= i || IA_s[tid] > i + SHARED_SIZE − 1) is false then
(20)        row_s ← max(IA_s[tid] − i, 0);
(21)        row_e ← min(IA_s[tid + 1] − i, SHARED_SIZE);
(22)        for j ← r_s to r_e − 1 do
(23)           sum += temp_s[j];
(24)        done
(25)     end
(26)     res[gid] ← sum;
(27) done
```

Algorithm 2: Kernel II in PCSR.

where N^{td}, N^{mem}, and N^{tb} are the maximum number of threads per multiprocessor, maximum amount of shared memory per multiprocessor, and maximum number of blocks per multiprocessor, respectively. N^{mem}, N^{td}, and N^{tb} are constant for a specific GPU.

The number of rows each block calculates for PCSR is an invariable value that is equal to the number of threads per block. However, in IPCSR, how many rows are assigned to a block is dynamically calculated. IPCSR splits the problem into row blocks, where the nonzero values within a block fit into a statically sized amount of shared memory (i.e., *SHARED_SIZE*). For example, assume that there are 128 rows for a matrix, each with 16 nonzero values, and *SHARED_SIZE* = 1024, two blocks, each with 64 rows, are required. If the first row instead has 32 nonzero values and the remaining 127 rows still have 16 nonzero values, three blocks will be needed. The first, second, and third blocks work on 63, 64, and 1 rows, respectively. With the assumption that the number of nonzero values per row is not more than *SHARED_SIZE*, the main procedure of generating row blocks is shown in Algorithm 4.

Here, the procedure in Algorithm 4 is executed on the CPU. Of course, this procedure can also be performed on the GPU. However, the test results show that the use of the GPU for this procedure does not increase the performance. For a specific matrix, the blocks need to be calculated only once. The complexity of generating the blocks is related to the number of rows, and the cost of generating the blocks is generally less than 1% of the cost that it takes to generate the CSR data structure.

In summary, compared to PCSR, IPCSR has the following improvement:

(i) Reducing two kernels to one kernel and avoiding loading/reading the global memory *temp* by merging operations in the first kernel of PCSR and the first stage of the second kernel of PCSR, which saves the cost of loading/reading the global memory.

(ii) Fitting the number of rows dynamically to the amount of *temp_s* space, which simplifies the work done on the GPU.

4.1. Optimizing the Scalar-Style Reduction. For the scalar-style reduction in Algorithm 3, a single row is only calculated by one thread in a block. If a sparse matrix has nearly equal and small number of nonzero values per row, the scalar-style reduction will be efficient because most threads for each block

```
Input: AA, JA, IA, x, blockRows;
Output: res;
(1)  _shared_temp_s[];
(2)  tid ← threadIdx.x; bid ← blockIdx.x;
(3)  startRow ← blockRows[bid];
(4)  endRow ← blockRows[bid + 1];
(5)  firstCol ← IA[startRow];
(6)  nnz ← IA[endRow] − IA[startRow];
     //Assemble AA[] ∗ x[] into shared memory
(7)  for i ← tid to nnz − 1 with i += blockDim.x do
(8)      temp_s[i] ← AA[firstCol + i] ∗ x[JA[firstCol + i]];
(9)  done
(10) _syncthreads();
     //Perform a scalar-style reduction from temp_s
(11) num_rows ← endRow − startRow;
(12) if tid < num_rows then
(13)     sum ← 0.0;
(14)     row_s ← IA[startRow + tid] − firstCol;
(15)     row_e ← IA[startRow + tid + 1] − firstCol;
(16)     for i ← row_s to row_e − 1 do
(17)         sum += temp_s[i];
(18)     done
(19)     res[startRow + tid] = sum;
(20) end
```

Algorithm 3: Improved PCSR.

```
Input: IA, n, nt, SHARED_SIZE;
Output: blockRows, CTR;
(1)  blockRows[0] ← 0; nRows ← 0; CTR ← 1; sum ← 0;
(2)  for i ← 0 to n − 1 do
         //Compute non-zeros and the total rows
(3)      sum += IA[i + 1] − IA[i];
(4)      nRows++;
(5)      if sum == SHARED_SIZE ‖
             (nRows == nt && sum <= SHARED_SIZE) then
             //This row fills up SHARED_SIZE or threads per block
(6)          blockRows[CTR] ← i + 1; CTR++; nRows ← 0; sum ← 0;
(7)      else if sum > SHARED_SIZE then
             //This row is an extra one that is excluded
(8)          blockRows[CTR] ← i; CTR++; nRows ← 0; sum ← 0; i − −;
(9)      end
(10) done
     //Extra case
(11) if blockRows[CTR − 1] != n then
(12)     blockRows[CTR] ← n;
(13) else
(14)     CTR − −;
(15) end
```

Algorithm 4: Main procedure of generating row blocks.

are active and can almost synchronously complete the scalar-style reduction. However, when a sparse matrix has a large or significantly different number of nonzero values per row, many threads for each block are inactive and the thread with a long row has a slower reduction speed than that with a short row, which greatly decreases the performance. In order to improve the efficiency of the scalar-style reduction, we take full advantage of threads per block and propose a decision tree to fit the number of threads dynamically to a single row, as shown in Figure 1. For each block, this decision tree enables selecting the appropriate number of threads to calculate a single row on the basis of nt/num_rows (number of rows that are

```
    ...
(1) _shared_bVAL_s[];
(2) warpNums ← warps_block[bid];
    //Perform a multiple scalar-style reduction from temp_s
(3) num_rows ← (endRow − startRow) ∗ warpNums;
(4) warp_id ← tid/warpNums; lane ← tid & (warpNums − 1);
(5) if tid < num_rows then
      //Perform a partial reduction from temp_s
(6)   bVAL ← 0.0;
(7)   row_s ← IA[startRow + warp_id] − firstCol + lane;
(8)   row_e ← IA[startRow + warp_id + 1] − firstCol;
(9)   for i ← row_s to row_e − 1 with i += warpNums do
(10)    bVAL += temp_s[i];
(11)  done
(12)  bVAL_s[tid] = bVAL;
(13)  _syncthreads();
      //Perform a warp reduction from bVAL_s
(14)  if lane < blockDim.x/4 && warpNums >= blockDim.x/2
(15)    {bVAL_s[tid] += bVAL_s[tid + blockDim.x/4];
(16)    _syncthreads();}
      ...
(17)  if lane < 8 && warpNums  >= 16
(18)    {bVAL_s[tid] += bVAL_s[tid + 8]; _syncthreads();}
(19)  if lane < 4 && warpNums  >= 8
(20)    {bVAL_s[tid] += bVAL_s[tid + 4]; _syncthreads();}
(21)  if lane < 2 && warpNums >= 4
(22)    {bVAL_s[tid] += bVAL_s[tid + 2]; _syncthreads();}
(23)  if lane < 1 && warpNums >= 2
(24)    res[startRow + warp_id] = bVAL_s[tid] + bVAL_s[tid + 1];
(25) end
```

Algorithm 5: Multiple scalar-style reduction.

assigned to it). When using multiple threads to calculate a single row in a block, the scalar-style reduction in Algorithm 3 is modified to the multiple scalar-style reduction (Algorithm 5).

The multiple scalar-style reduction consists of two steps: a partial-reduction step and a warp-reduction step. In the partial-reduction step, each one of the threads in each thread group (*warpNums* threads are grouped into a thread group) performs a partial reduction and saves the reduction value to the shared memory. Then, in the warp-reduction step, the partial-style reduction values in the shared memory for each thread group are reduced to an output value in parallel. The main procedure of IPCSR with the adaptive number of threads per block is shown in Algorithm 7.

4.2. Optimizing Very Long Rows. If a single row has nonzero values that are more than *SHARED_SIZE*, IPCSR in Algorithm 7 will not be suitable for calculating the extremely long individual row because it cannot move these values into the shared memory *temp_s*.

This is solved by giving each of the very long rows to a single block. To assign each of these very long rows to a single block, the code in the 8th line of Algorithm 4 is modified as Algorithm 6.

The main procedure where a single long row is calculated by one block is listed in Algorithm 8. First, the thread block performs many parallel coalesced loads from the *JA* and *AA* arrays. This quickly places values into the shared memory *bVAL_s* (see lines (2)–(4) in Algorithm 8). Second, the values in *bVAL_s* are reduced in parallel to an output value (see lines (7)–(15) in Algorithm 8).

The main procedure of IPCSR with very long rows is shown in Algorithm 9. By experiments, if one block only involves a single row whose nonzeros are not more than *SHARED_SIZE*, the performance that is obtained by the multiple scalar-style reduction in Algorithm 5 is always less than that obtained by a single long row reduction in Algorithm 8. Therefore, we also use the single long row reduction in Algorithm 8 for each block involving a single row whose nonzeros are not more than *SHARED_SIZE*.

4.3. Optimizing the Accesses to x. In IPCSR, the vector x in the global memory is randomly accessed, which results in decreasing its performance. Therefore, the texture memory is utilized to alleviate the high latency of randomly accessing the vector x in the global memory [34]. For single precision,

$$AA\left[firstCol + i\right] * x\left[JA\left[firstCol + i\right]\right] \tag{4}$$

can be rewritten as

$$AA\left[firstCol + i\right] * \text{text1Dfetch}\left(floatTexRef, JA\left[firstCol + i\right]\right). \tag{5}$$

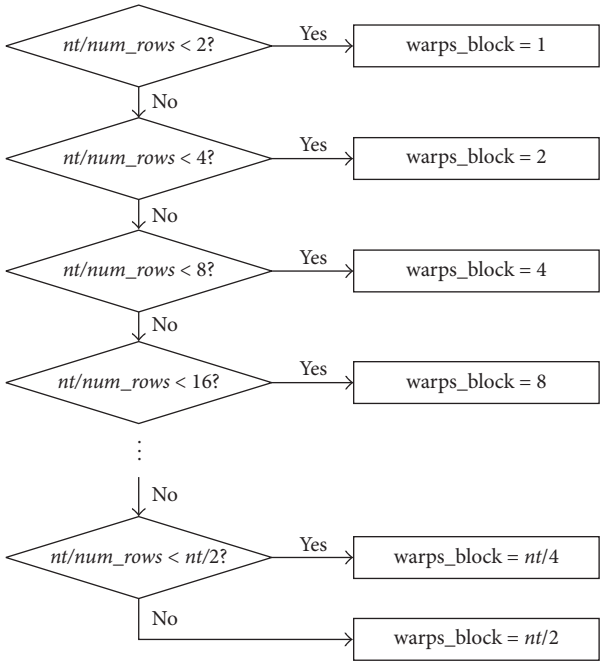

Figure 1: Decision tree.

```
if nRows > 1 then
    //This row is an extra one that is excluded
    blockRows[CTR] ← i; i − −;
else
    //This row is more than SHARED_SIZE non-zeros
    blockRows[CTR] ← i + 1;
end
CTR++; nRows ← 0; sum ← 0;
```

Algorithm 6

Because the texture does not support the double value, the following function *fetch_double*() shown in Algorithm 10 is adopted to transfer the int2 value to the double value.

Furthermore, for double precision, based on the function *fetch_double*(), we rewrite

$$AA\left[firstCol + i\right] * x\left[JA\left[firstCol + i\right]\right] \tag{6}$$

as

$$AA\left[firstCol + i\right] * fetch\_double\left(doubleTexRef, JA\left[firstCol + i\right]\right). \tag{7}$$

5. Experimental Results

Here, in addition to comparing IPCSR with PCSR, we also use CUSPARSE V6.5 [21], CUSP V0.5.1 [22], clSpMV [23], CSR-Adaptive [17], and CSR5 [18] for the performance comparison. CSRMV, a CSR-based algorithm, is only considered for CUSPARSE. CUSP supports six algorithms that include COO, CSR-scalar, CSR-vector, DIA, ELL, and HYB. For each test matrix, we choose the best of the six CUSP algorithms. For clSpMV, we test all the single formats and choose the best performing single format for each matrix.

Table 1: Sparse matrices used in the experiments.

Matrix	Rows	Columns	Nonzeros (NZ)	NZ/Row
Dense	2,000	2,000	4,000,000	2000.0
Protein	36,417	36,417	4,344,765	119.3
FEM/Spheres	83,334	83,334	6,010,480	72.1
FEM/Cantilever	62,451	62,451	4,007,383	64.1
Wind Tunnel	217,918	217,918	11,634,424	53.3
FEM/Harbor	46,835	46,835	2,374,001	50.6
QCD	49,152	49,152	1,916,928	39.0
FEM/Ship	140,874	140,874	7,813,404	55.4
Economics	206,500	206,500	1,273,389	6.1
Epidemiology	525,825	525,825	2,100,225	3.9
FEM/Accelerator	121,192	121,192	2,624,331	21.6
Circuit	170,998	170,998	958,936	5.6
Webbase	1,000,005	1,000,005	3,105,536	3.1
LP	4,284	1,092,610	11,279,748	2632.9
circuit5M	5,558,326	5,558,326	59,524,291	10.7
eu-2005	862,664	862,664	19,235,140	22.3
Ga41As41H72	268,096	268,096	18,488,476	69.0
in-2004	1,382,908	1,382,908	16,917,053	12.2
mip1	66,463	66,463	10,352,819	155.8
Si41Ge41H72	185,639	185,639	15,011,265	80.9

Table 2: Overview of GPUs.

Hardware	C2050	K20c
Cores	448	2496
Clock speed (GHz)	1.15	0.71
Memory type	GDDR5	GDDR5
Memory size (GB)	3	5
Max bandwidth (GB/s)	144	208
Compute capability	2.0	3.0

5.1. Experimental Setup. We use a total of 20 sparse matrices: 14 of them are from [35] and 6 of them are from the University of Florida Sparse Matrix Collection [36]. Table 1 summarizes the information of the sparse matrices, including the number of rows, number of columns, number of nonzeros, and number of nonzeros per row. These matrices have been widely used in some previous work [8, 14, 23, 28, 29, 32, 37].

Experimental environments include one machine that is equipped with an Intel Xeon Quad-Core CPU and an NVIDIA Tesla C2050 GPU and another machine with an Intel Xeon Quad-Core CPU and an NVIDIA Tesla K20c GPU. Table 2 shows an overview of NVIDIA GPUs.

The performance of all algorithms is measured in terms of GFlop/s (second) or GByte/s (GB/s). Measured performance does not include the data transfer (from GPU to CPU or from CPU to GPU). For each algorithm, we randomly execute it 50 times and then take the average performance as its performance.

```
Input: AA, JA, IA, x, blockRows, warps_block;
Output: res;
(1)  _shared_temp_s[];
(2)  _shared_bVAL_s[];
(3)  tid ← threadIdx.x; bid ← blockIdx.x;
(4)  startRow ← blockRows[bid]; endRow ← blockRows[bid + 1];
(5)  firstCol ← IA[startRow]; nnz ← IA[endRow] − IA[startRow];
     //Assemble AA[] ∗ x[] into shared memory
(6)  for i ← tid to nnz − 1 with i += blockDim.x do
(7)      temp_s[i] ← AA[firstCol + i] ∗ x[JA[firstCol + i]];
(8)  done
(9)  _syncthreads();
(10) warpNums ← warps_block[bid];
(11) if warpNums < 2 then
(12)     //Omitted: Perform a scalar-style reduction
(13) else
         //Omitted: Perform a multiple scalar-style reduction
(14) end
```

Algorithm 7: IPCSR with the adaptive number of threads per block.

```
     ...
     //a long row is calculated
(1)  bVAL ← 0.0;
(2)  for i ← tid to nnz with i += blockDim.x do
(3)      bVAL += AA[firstCol + i] ∗ x[JA[firstCol + i]];
(4)  done
(5)  bVAL_s[tid] = bVAL;
(6)  _syncthreads();
(7)  if tid < blockDim.x/2
(8)      {bVAL_s[tid] += bVAL_s[tid + blockDim.x/2]; _syncthreads();}
(9)  ...
(10) if tid < 32 {bVAL_s[tid] += bVAL_s[tid + 32]; _syncthreads();}
(11) if tid < 16 {bVAL_s[tid] += bVAL_s[tid + 16]; _syncthreads();}
(12) if tid < 8 {bVAL_s[tid] += bVAL_s[tid + 8]; _syncthreads();}
(13) if tid < 4 {bVAL_s[tid] += bVAL_s[tid + 4]; _syncthreads();}
(14) if tid < 2 {bVAL_s[tid] += bVAL_s[tid + 2]; _syncthreads();}
(15) if tid < 1 res[startRow] = bVAL_s[tid] + bVAL_s[tid + 1];
```

Algorithm 8: Single long row reduction.

```
Input: AA, JA, IA, x, blockRows, warps_block;
Output: res;
(1)  _shared_temp_s[];
(2)  _shared_bVAL_s[];
(3)  tid ← threadIdx.x; bid ← blockIdx.x;
(4)  startRow ← blockRows[bid]; endRow ← blockRows[bid + 1];
(5)  firstCol ← IA[startRow]; nnz ← IA[endRow] − IA[startRow];
(6)  num_rows ← endRow − startRow;
(7)  if nnz < SHARED_SIZE && num_rows != 1 then
(8)      //Omitted: With the adaptive number of threads per block;
(9)  else
         //Omitted: With very long rows
(10) end
```

Algorithm 9: IPCSR with very long rows.

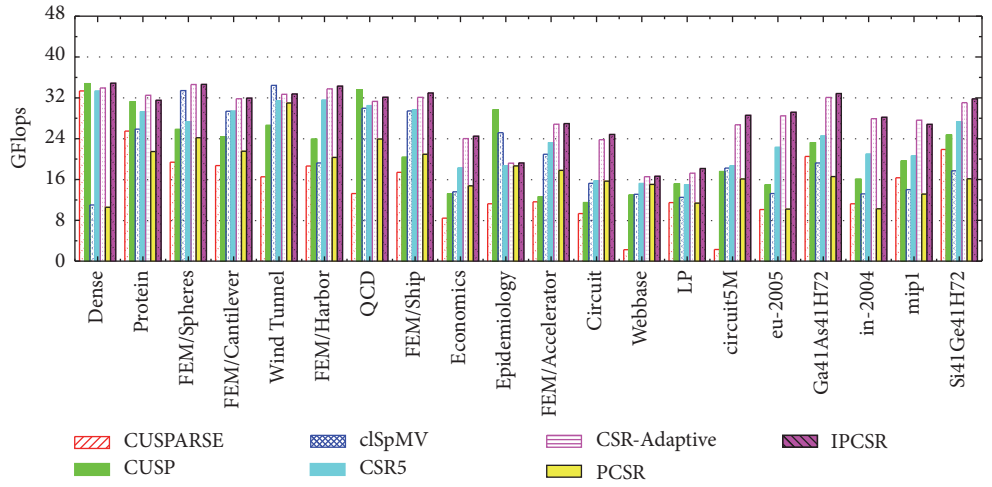

Figure 2: Performance comparison for single precision on C2050.

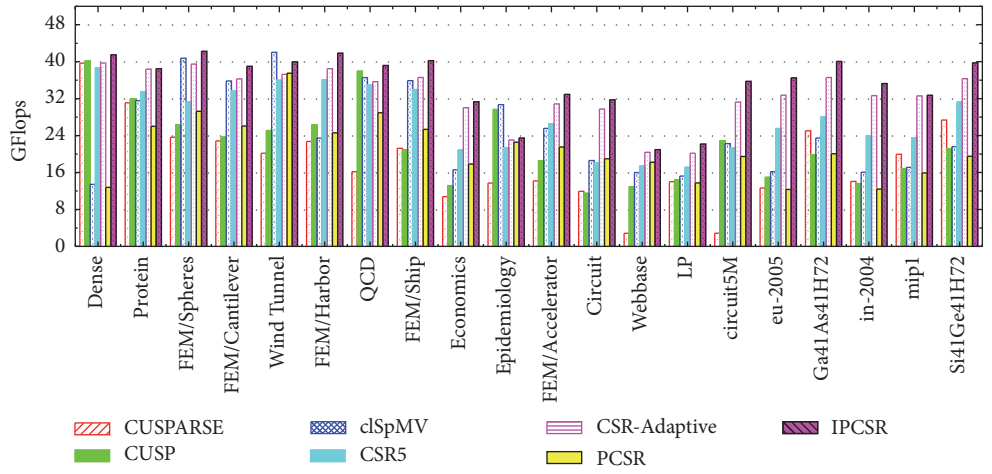

Figure 3: Performance comparison for single precision on K20c.

```
(1) _device_double fetch_double(texture⟨int2, 1⟩t, int i){
(2)    int2 v = tex1Dfetch(t, i);
(3)    return_hiloint2double(v · y, v · x);
(4) }
```

Algorithm 10

5.2. Single Precision. Figures 2 and 3 illustrate the performance of various implementation techniques in single precision in terms of GFlop/s for all test matrices on C2050 and K20c, respectively. GFlop/s values are calculated on the basis of the assumption of two flops per nonzero entry for a matrix [8]. Table 3 presents speedups that are achieved by IPCSR over other implementation techniques on both GPUs for all test matrices. We can observe that the performance of our proposed IPCSR for each test matrix is much better than that of PCSR on both GPUs. IPCSR achieves speedups, ranging from 1.03 to 3.30 on C2050 and 1.04 to 3.25 on K20c, over PCSR (Table 3). The average speedups on the C2050 and K20c are 1.76 and 1.79, respectively (Figure 4).

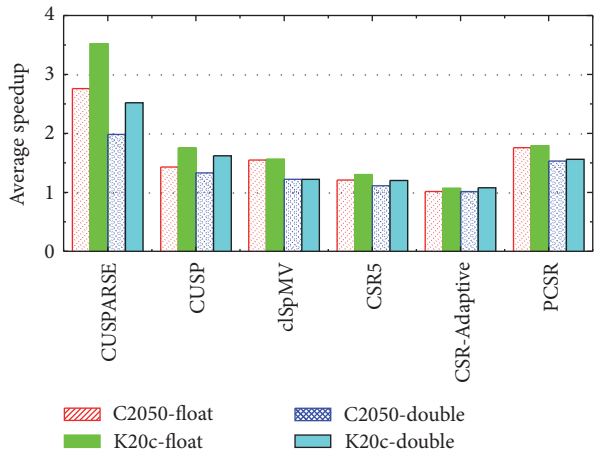

Figure 4: Average speedups of IPCSR over other implementation techniques.

Furthermore, from Figure 2, we can see that IPCSR outperforms CUSPARSE, CUSP, and clSpMV on C2050 for all

Table 3: Speedups of IPCSR over other implementation techniques in single precision.

Matrix	Speedups on C2050						Speedups on K20c					
	CUSPARSE	CUSP	clSpMV	PCSR	CSR5	CSR-Adaptive	CUSPARSE	CUSP	clSpMV	PCSR	CSR5	CSR-Adaptive
Dense	1.04	1.00	3.16	3.30	1.05	1.03	1.24	1.03	3.09	3.25	1.07	1.04
Protein	1.24	1.01	1.22	1.47	1.08	0.97	1.55	1.21	1.22	1.48	1.15	1.00
FEM/Spheres	1.79	1.34	1.04	1.43	1.27	1.00	2.24	1.61	1.04	1.44	1.35	1.07
FEM/Cantilever	1.71	1.31	1.09	1.48	1.09	1.00	2.14	1.65	1.09	1.50	1.16	1.08
Wind Tunnel	1.98	1.23	0.95	1.06	1.04	1.00	2.49	1.59	0.95	1.07	1.11	1.07
FEM/Harbor	1.84	1.44	1.78	1.69	1.09	1.02	2.31	1.59	1.78	1.70	1.16	1.08
QCD	2.42	0.96	1.07	1.34	1.05	1.03	3.04	1.03	1.07	1.35	1.12	1.05
FEM/Ship	1.89	1.62	1.12	1.57	1.11	1.03	2.38	1.92	1.12	1.59	1.18	1.10
Economics	2.91	1.86	1.80	1.66	1.34	1.02	3.83	2.38	1.89	1.75	1.50	1.04
Epidemiology	1.71	0.65	0.76	1.03	1.03	1.00	2.15	0.79	0.76	1.04	1.10	1.02
FEM/Accelerator	2.32	2.14	1.29	1.52	1.16	1.01	2.91	1.77	1.29	1.53	1.24	1.07
Circuit	2.66	2.16	1.63	1.58	1.58	1.04	3.51	2.76	1.71	1.67	1.76	1.07
Webbase	7.43	1.29	1.27	1.11	1.09	1.01	9.64	1.62	1.31	1.15	1.20	1.03
LP	1.58	1.20	1.45	1.60	1.21	1.05	1.99	1.54	1.45	1.61	1.29	1.10
circuit5M	12.55	1.63	1.78	1.78	1.53	1.06	16.16	1.56	1.61	1.83	1.67	1.14
eu-2005	2.89	1.95	2.20	2.86	1.31	1.02	3.71	2.44	2.25	2.96	1.43	1.11
Ga41As41H72	1.60	1.41	1.71	1.98	1.34	1.02	2.01	2.02	1.71	2.00	1.43	1.09
in-2004	2.51	1.76	2.14	2.75	1.35	1.01	3.23	2.59	2.20	2.84	1.47	1.08
mip1	1.64	1.37	1.91	2.04	1.30	0.97	2.06	1.95	1.91	2.06	1.39	1.00
Si41Ge41H72	1.45	1.29	1.80	1.97	1.17	1.02	1.87	1.88	1.84	2.03	1.27	1.09

the matrices except for Wind Tunnel, QCD, and Epidemiology. IPCSR has only slightly lower performance than clSpMV and CUSP for Wind Tunnel and QCD, respectively. For Epidemiology, the number of rows in many blocks is nearly half the number of threads, and other rows have small and nearly equal number of nonzeros except that a few rows have large number of nonzeros. This greatly decreases the performance of performing the scalar-style reduction in IPCSR. For Wind Tunnel, QCD, and Epidemiology, IPCSR has some performance degradation, but it achieves average performance improvement of 2.76x over CUSPARSE, 1.43x over CUSP, and 1.55x over clSpMV (Figure 4). In addition, IPCSR has also better behavior than CSR5 for all test cases and achieves average performance improvement of 1.21x over CSR5 (Figure 4). The performance of IPCSR is slightly higher than that of CSR-Adaptive for all test matrices except for Protein and mip1, and IPCSR only achieves average performance improvement of 1.02x over CSR-Adaptive (Figure 4). For Protein and mip1, there are many rows that have a large number of nonzeros. CSR-Adaptive can use several blocks to calculate a row whereas IPCSR at most uses a block to calculate a row, which can be the reason resulting in the improvement of the CSR-Adaptive performance compared to our proposed IPCSR.

Similar to C2050, we also observe that IPCSR has higher performance than CUSPARSE, CUSP, and clSpMV for all test matrices except for Wind Tunnel and Epidemiology on K20c from Figure 3. IPCSR achieves an average speedup of 3.52x over CUSPARSE, 1.75x over CUSP, and 1.56x over clSpMV, respectively (Figure 4). IPCSR outperforms CSR5 and the performance of IPCSR is slightly better than that of CSR-Adaptive for all test cases. IPCSR achieves average performance improvement of 1.30x over CSR5 and 1.07x over CSR-Adaptive, respectively (Figure 4).

5.3. Double Precision. The performance of CUSPARSE, CUSP, clSpMV, CSR5, CSR-Adaptive, PCSR, and IPCSR in double precision for all test matrices on C2050 and K20c is listed in Figures 5 and 6, respectively. Table 4 presents speedups that are achieved by IPCSR over other implementation techniques on both GPUs for all test matrices. Compared to PCSR, IPCSR achieves higher performance for all the matrices on both GPUs. The speedups of IPCSR over PCSR range from 1.07 to 2.72 on C2050 and 1.08 to 2.68 on K20c. The average speedups on C2050 and K20c are 1.53 and 1.56, respectively.

For CUSPARSE, CUSP, clSpMV, CSR5, CSR-Adaptive, and IPCSR, there is not a single approach that is best for all the matrices on both GPUs. However, IPCSR usually equals or outperforms other approaches and is the best approach on average. Similar to single precision, the major exceptional matrices are still Wind Tunnel, QCD, and Epidemiology in double precision. Although the performance of IPCSR is not the best for these exceptional matrices, IPCSR achieves average performance improvement of 1.98x over CUSPARSE, 1.33x over CUSP, 1.22x over clSpMV, 1.11x over CSR5, and 1.01x over CSR-Adaptive on C2050 (Figure 4). Similarly, IPCSR achieves average performance improvement of 2.52x over CUSPARSE, 1.62x over CUSP, 1.26x over clSpMV, 1.20x over CSR5, and 1.08x over CSR-Adaptive on K20c (Figure 4).

Therefore, we can conclude that IPCSR greatly improves the performance of PCSR and shows better behavior than

Table 4: Speedups of IPCSR over other implementation techniques in double precision.

Matrix	Speedups on C2050						Speedups on K20c					
	CUSPARSE	CUSP	clSpMV	PCSR	CSR5	CSR-Adaptive	CUSPARSE	CUSP	clSpMV	PCSR	CSR5	CSR-Adaptive
Dense	0.91	0.95	2.06	2.72	1.01	1.00	1.08	0.97	2.01	2.68	1.05	1.01
Protein	0.87	0.92	0.94	1.08	1.01	0.95	1.09	1.10	0.94	1.09	1.08	0.99
FEM/Spheres	1.37	1.39	1.29	1.36	1.22	1.00	1.72	1.66	1.29	1.37	1.30	1.07
FEM/Cantilever	1.35	1.41	1.26	1.55	1.17	1.04	1.70	1.77	1.26	1.57	1.25	1.11
Wind Tunnel	1.34	1.21	0.82	1.27	0.86	0.83	1.82	1.69	0.88	1.38	1.00	0.97
FEM/Harbor	1.13	1.22	1.05	1.28	1.06	1.02	1.42	1.35	1.05	1.29	1.14	1.09
QCD	1.49	0.71	0.96	1.16	1.05	1.01	1.87	0.77	0.70	1.17	1.13	1.09
FEM/Ship	1.43	1.60	1.19	1.23	1.08	1.01	1.79	1.89	1.19	1.24	1.15	1.08
Economics	2.28	1.68	1.67	1.59	1.15	1.056	2.87	2.04	1.67	1.60	1.23	1.08
Epidemiology	1.42	0.90	0.74	1.07	1.01	1.00	1.79	1.10	0.74	1.08	1.08	1.03
FEM/Accelerator	1.70	1.65	0.92	1.48	1.07	1.00	2.13	1.37	0.92	1.49	1.15	1.07
Circuit	1.94	1.85	1.25	1.11	1.06	1.02	2.56	2.36	1.31	1.17	1.19	1.06
Webbase	6.11	1.29	0.93	1.19	1.17	1.05	7.93	1.63	0.96	1.24	1.29	1.08
LP	1.39	1.28	1.29	1.79	1.28	1.01	1.75	1.64	1.18	1.81	1.37	1.08
circuit5M	7.61	1.71	1.15	1.74	1.10	1.06	9.80	1.64	1.18	1.80	1.21	1.16
eu-2005	1.90	1.78	1.52	2.06	1.17	1.03	2.44	2.22	1.56	2.13	1.29	1.13
Ga41As41H72	1.06	1.19	1.09	1.23	1.08	1.03	1.33	1.70	1.09	1.24	1.16	1.11
in-2004	1.69	1.41	1.31	1.92	1.23	1.14	2.18	2.08	1.35	1.98	1.35	1.26
mip1	1.10	1.11	1.16	1.87	1.08	0.98	1.39	1.59	1.16	1.88	1.15	1.03
Si41Ge41H72	1.45	1.29	1.97	2.50	1.26	1.01	1.82	1.84	1.80	1.99	1.35	1.08

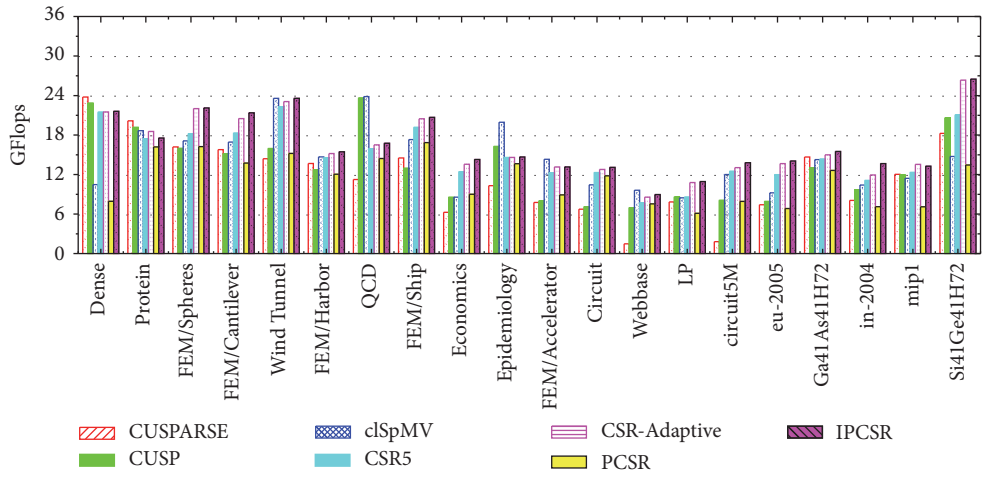

Figure 5: Performance comparison for double precision on C2050.

CUSPARSE, CUSP, clSpMV, CSR5, and CSR-Adaptive for all test matrices on both GPUs.

5.4. Effective Bandwidth and Evaluations. Here, we only take single precision on C2050 to test the effective bandwidth of all algorithms. The effective bandwidth is calculated by the following:

$$\text{Effective bandwidth} = \frac{\left(\left(B_r + B_w\right)/10^9\right)}{\text{time}}. \quad (8)$$

Here, the effective bandwidth is in units of GB/s; B_r is the number of bytes read per kernel and B_w is the number of bytes written per kernel. Figure 7 lists the effective bandwidth of CUSPARSE, CUSP, clSpMV, CSR5, CSR-Adaptive, PCSR, and IPCSR for all test matrices on C2050. The unit is GB/s. The red line is the peak theoretical bandwidth (144 GB/s) of C2050. We can observe that IPCSR achieves effective bandwidth ranging from 72.63 to 139.48 GB/s for all test matrices on C2050 and its average effective bandwidth of IPCSR is 119.22 GB/s. On the basis of the definition of the bandwidth [20], we can conclude that IPCSR has high parallelism. For CUSPARSE, CUSP, clSpMV, CSR5, CSR-Adaptive, and PCSR, their average effective bandwidth for all test cases on C2050 is 61.64, 89.92, 85.35, 100.29, 117.43,

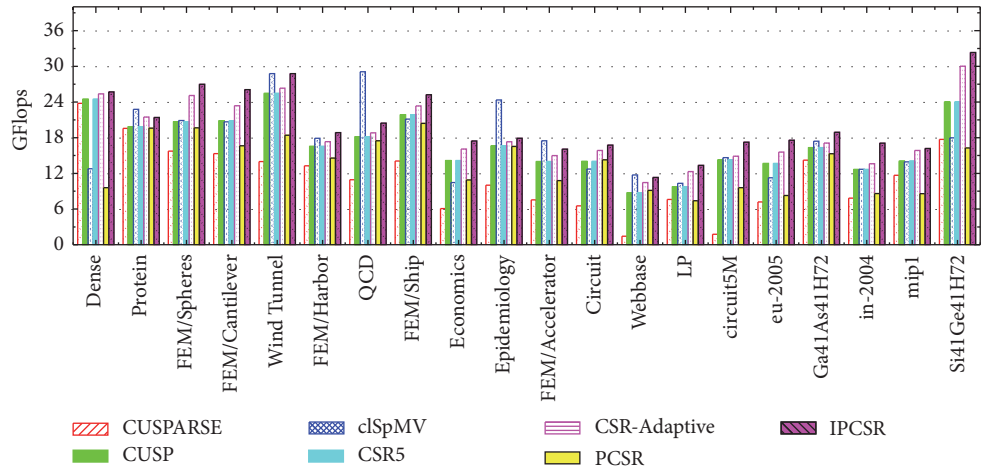

Figure 6: Performance comparison for double precision on K20c.

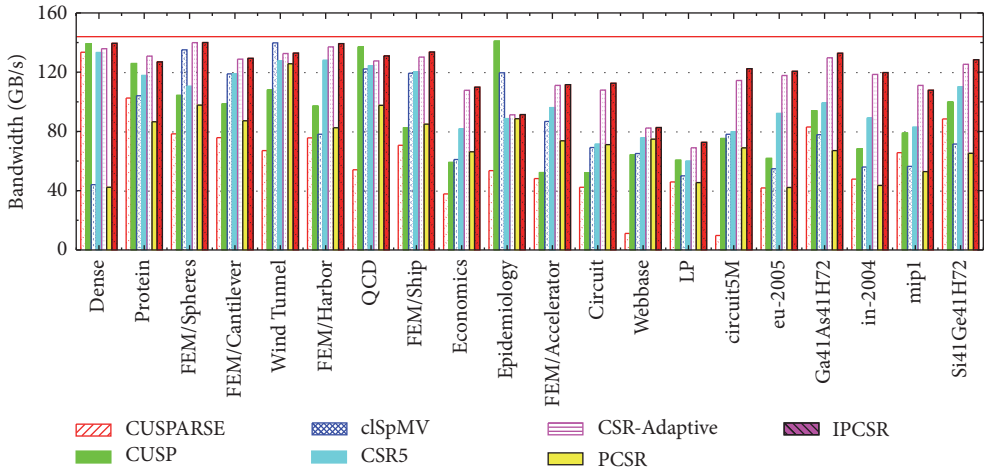

Figure 7: Effective bandwidth of all algorithms on C2050.

and 73.19 GB/s, respectively. This further verifies that IPCSR is much better than CUSPARSE, CUSP, clSpMV, CSR5, and PCSR and slightly outperforms CSR-Adaptive.

Compared to CSR5, the implementation of IPCSR is simpler. IPCSR keeps the CSR arrays intact and only incorporates an extra array (i.e., *blockRows*) whereas CSR5 only leaves one of the three CSR arrays unchanged, stores the other two arrays in an in-place tile-transposed order, and adds two groups of extra auxiliary information. Therefore, the limitations of CSR5 are its complexity and the matrix transpose, which can cause large transformation overheads [17]. CSR-Adaptive involves three algorithms: CSR-Stream, CSR-vector, and CSR-VectorL, which correspond to the matrices with short, long, and very long rows, respectively. Their switches are complicated because some thresholds are required (i.e., *num_rows_stream* and *NNZ_multiplier*). IPCSR uses one algorithm, and the only difference is that the reduction method can be different for different types of rows. And the reduction method can be easily chosen utilizing a decision tree. For very long rows, compared to a row per block that is used in IPCSR, CSR-Adaptive uses multiple blocks to calculate a row. However, the efficiency is often offset by the overhead of the atomic operation and block synchronization.

Like other GPU-accelerated SpMV algorithms, the performance of IPCSR is also sensitive to the GPU architecture. For different GPU architectures, IPCSR usually has a little difference. There are the following two main reasons: (1) some parameters (i.e., *SHARED_SIZE*) that have a great influence on the IPCSR performance depend on the GPU feature and (2) the GPU features and performance parameters usually are different for different GPU architectures. For IPCSR, based on (3), the *SHARED_SIZE* value in single precision is twice that in double precision. If so, each block in single precision will usually involve more rows and have higher reduction efficiency than that in double precision. Other algorithms do not have this feature. This can result in the performance improvement of IPCSR being faster than other algorithms from double precision to single precision.

6. Conclusion

In this study, we propose a novel CSR-based algorithm on the GPU, which improves the performance of PCSR. Compared

to PCSR, our proposed algorithm IPCSR has the following distinct characteristics: (1) each thread block is assigned to different number of rows, (2) a multiple scalar-style reduction is involved besides the single scalar-style reduction, and (3) optimization implementation is specially designed for very long rows. Experimental results on C2050 and K20c show that IPCSR outperforms PCSR on both GPUs. Furthermore, IPCSR also shows higher performance than the vendor tuned CUSPARSE V6.5, CUSP V0.5.1, and three popular algorithms clSpMV, CSR5, and CSR-Adaptive.

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 and the Natural Science Foundation of Zhejiang Province, China, under Grant no. LY17F020021.

References

[1] NVIDIA, "CUDA C Programming Guide 6.5," 2014, http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, Pa, USA, 2nd edition, 2003.

[3] J. Gao, Z. Li, R. Liang, and G. He, "Adaptive optimization l_1-minimization solvers on GPU," *International Journal of Parallel Programming*, 2016.

[4] J. Li, X. Li, B. Yang, and X. Sun, "Segmentation-based image copy-move forgery detection scheme," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 3, pp. 507–518, 2015.

[5] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 340–352, 2016.

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

[7] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Tech. Rep., NVIDIA, 2008.

[8] 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, ACM, Portland, Ore, USA, November 2009.

[9] M. M. Baskaran and R. Bordawekar, "Optimizing Sparse Matrixvector Multiplication on GPUs," Tech. Rep., IBM Research, Yorktown Heights, NY, USA, 2009.

[10] 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.

[11] 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.

[12] I. Reguly and M. Giles, "Efficient sparse matrix-vector multiplication on cache-based GPUs," in *Proceedings of the IEEE Innovative Parallel Computing*, pp. 1–12, Piscataway, NJ, USA, 2012.

[13] J. Gao, R. Liang, and J. Wang, "Research on the conjugate gradient algorithm with a modified incomplete Cholesky preconditioner on GPU," *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2088–2098, 2014.

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

[15] Z. Koza, M. Matyka, and S. Szkoda, "Compressed multirow storage format for sparse matrices on graphics processing units," *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C219–C239, 2014.

[16] Y. Liu and B. Schmidt, "LightSpMV: faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs," in *Proceedings of the IEEE 26th International Conference on Application-Specific Systems, Architectures and Processors (ASAP '15)*, pp. 82–89, IEEE, Toronto, Canada, July 2015.

[17] M. Daga and J. L. Greathouse, "Structural agnostic SpMV: adapting CSR-adaptive for irregular matrices," in *Proceedings of the IEEE 22nd International Conference on High Performance Computing (HiPC '15)*, pp. 64–74, Bengaluru, India, December 2015.

[18] W. Liu and B. Vinter, "CSR5: an efficient storage format for crossplatform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM International Conference on Supercomputing (ICS '15)*, pp. 339–350, ACM, Newport Beach, Calif, USA, June 2015.

[19] G. He and J. Gao, "A novel CSR-based sparse matrix-vector multiplication on GPUs," *Mathematical Problems in Engineering*, vol. 2016, Article ID 8471283, 12 pages, 2016.

[20] NVIDIA, *CUDA C Best Practices Guide 6.5*, 2014, http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[21] NVIDIA, *CUSPARSE Library 6.5*, 2014, https://developer.nvidia.com/cusparse.

[22] 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.

[23] B.-Y. Su and K. Keutzer, "clSpMV: a cross-platform OpenCL SpMV framework on GPUs," in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*, pp. 353–364, Venice, Italy, June 2012.

[24] 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.

[25] 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 on High Performance Computing, Networking, Storage and Analysis (SC '13)*, article 26, pp. 1–12, IEEE, 2013.

[26] 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.

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

[28] 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.

[29] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *High Performance Embedded Architectures and Compilers*, vol. 5952 of *Lecture Notes in Computer Science*, pp. 111–125, Springer, Berlin, Germany, 2010.

[30] 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.

[31] 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.

[32] 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, ACM, Austin, Tex, USA, February 2014.

[33] 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.

[34] 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.

[35] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[36] 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.

[37] J. Gao, Y. Wang, and J. Wang, "A novel multi-graphics processing unit parallel optimization framework for the sparse matrix-vector multiplication," *Concurrency and Computation: Practice and Experience*, 2016.