electronics

MDPI

Article

Leveraging Memory Copy Overlap for Efficient Sparse Matrix-Vector Multiplication on GPUs

Guangsen Zeng and Yi Zou *

School of Microelectronics, South China University of Technology, Guangzhou 510641, China; mizengguangsen@mail.scut.edu.cn
* Correspondence: zouyi@scut.edu.cn

Abstract: Sparse matrix-vector multiplication (SpMV) is central to many scientific, engineering, and other applications, including machine learning. Compressed Sparse Row (CSR) is a widely used sparse matrix storage format. SpMV using the CSR format on GPU computing platforms is widely studied, where the access behavior of GPU is often the performance bottleneck. The Ampere GPU architecture recently from NVIDIA provides a new asynchronous memory copy instruction, `memcpy_async`, for more efficient data movement in shared memory. Leveraging the capability of this new `memcpy_async` instruction, we first propose the *CSR-Partial-Overlap* to carefully overlap the data copy from global memory to shared memory and computation, allowing us to take full advantage of the data transfer time. In addition, we design the dynamic batch partition and the dynamic threads distribution to achieve effective load balancing, avoid the overhead of fixing up partial sums, and improve thread utilization. Furthermore, we propose the *CSR-Full-Overlap* based on the CSR-Partial-Overlap, which takes the overlap of data transfer from host to device and SpMV kernel execution into account as well. The CSR-Full-Overlap unifies the two major overlaps in SpMV and hides the computation as much as possible in the two important access behaviors of the GPU. This allows CSR-Full-Overlap to achieve the best performance gains from both overlaps. As far as we know, this paper is the first in-depth study of how `memcpy_async` can be potentially applied to help accelerate SpMV computation in GPU platforms. We compare CSR-Full-Overlap to the current state-of-the-art cuSPARSE, where our experimental results show an average 2.03x performance gain and up to 2.67x performance gain.

Keywords: sparse matrix-vector multiplication (SpMV); CSR; NVIDIA Ampere GPUs; shared memory; overlap; `memcpy_async`; dynamic batch partition; dynamic threads distribution

Citation: Zeng, G.; Zou, Y. Leveraging Memory Copy Overlap for Efficient Sparse Matrix-Vector Multiplication on GPUs. *Electronics* **2023**, *12*, 3687. https://doi.org/10.3390/electronics12173687

Academic Editor: Shinichi Yamagiwa

Received: 2 August 2023
Revised: 24 August 2023
Accepted: 30 August 2023
Published: 31 August 2023

1. Introduction

Sparse matrix-vector multiplication (SpMV) is a commonly used operation in computer science and numerical computation with a wide range of applications in many fields [1,2], such as large-scale linear systems [3], graph analytics [4,5], machine learning [6], and so on. SpMV is often the performance bottleneck of these applications, so it is important to study how to improve the performance of SpMV. Sparse matrices generally only have a small number of nonzero elements, which are distributed in different positions of the matrix. This irregular distribution results in uncoalesced memory access behavior, which leads to huge memory access overhead [7]. In order to reduce such memory overhead to improve the memory access efficiency, a variety of sparse matrix storage formats have been proposed, such as Coordinate (COO) [7], Compressed Sparse Row (CSR) [7], ELLPACK (ELL) [7], Hybrid ELL COO (HYB) [7], CSR5 [8], etc. Among these various storage formats, CSR is the most widely used and the de facto default format. Other storage formats are inevitably involved in the problem of format conversion from CSR, causing considerable performance overhead [9]. Therefore, it is essential to focus on improving the performance of the SpMV algorithm based on the CSR storage format.

Graphics processing units (GPUs) provide efficient parallel computing capability through multiple stream processors and high memory bandwidth. GPU computing platforms are very attractive in the field of high performance computing [10]. In this paper, we focus particularly on SpMV performance accelerations on NVIDIA GPU-based computing platforms. Without loss of generality, we use the term *GPUs* throughout the paper to refer to NVIDIA GPUs, which are most commonly used in SpMV calculations. However, the unstructured nature of sparse matrices introduces new challenges, such as difficulty in maintaining the thread load balance and coalesced memory accesses [7]. Such problems inevitably all contribute to constraining the GPU to achieve the desired performance acceleration for SpMV.

In this paper, following convention, we use the term *device* throughout the paper to refer to NVIDIA GPUs and *host* to refer to CPUs [11], where *global* and *shared* memory are used in the context of GPUs. Based on our observations, there exist two major overlaps in the SpMV data movement, namely the *global-to-shared* data copy overlap and *host-to-device* data copy overlap. The global-to-shared overlap refers to the overlap of data copy from global memory to shared memory and computation. The device-to-host overlap refers to the overlap of data transfer from host to device and SpMV kernel execution. For convenience of discussion, we use *GSOL* and *HDOL* for short in this paper to refer to these two overlaps.

In this paper, we propose a new method to unify these two overlaps to achieve the best performance improvement for SpMV. The main contributions of this paper are summarized as follows:

- We introduce the *CSR-Partial-Overlap*, a novel CSR-based SpMV acceleration algorithm on CUDA-enabled GPUs. This serves as the first in-depth study of how the new `memcpy_async` in NVIDIA Ampere GPU architecture can be applied to performance acceleration in SpMV;
- We design the *dynamic batch partition* and the *dynamic threads distribution* algorithms to further improve the performance of the CSR-Partial-Overlap;
- We propose the *CSR-Full-Overlap* based on CSR-Partial-Overlap to unify two major overlaps in SpMV data movement. CSR-Full-Overlap hides the SpMV computation as much as possible in two major types of data transfers, gaining performance acceleration from both overlaps;
- We present a detailed comparative analysis of the proposed CSR-Partial-Overlap and CSR-Full-Overlap with a variety of well-known SpMV implementations. We demonstrate that CSR-Partial-Overlap outperforms all other approaches by about 1.4× on average, where CSR-Full-Overlap is superior to the widely used NVIDIA cuSPARSE, with a speedup of an average of 2.03× and up to 2.67×.

The rest of the paper is organized as follows. Section 2 provides a literature review of related work in this context. Section 3 offers a brief background review on SpMV, sparse matrix storage structures, NVIDIA Ampere GPU architecture, and CUDA. In Section 4, we describe the design philosophy of the proposed CSR-Partial-Overlap and CSR-Full-Overlap methods in detail. Next, we introduce the experimental setup in Section 5, as well as analysis and discussions on evaluation results. We conclude this paper in Section 6 with our thoughts on future research directions.

2. Related Work

GPU-based SpMV acceleration has been studied extensively [7–10,12–21]. Different optimization techniques have been proposed, focusing on introducing new storage format [7,8], threads distribution [9,16], shared memory leverage [12], automatic format selection [13,20], performance analysis and autotuning [14,18,19,21], and load balancing [15,17].

Bell and Garland propose CSR-Scalar and CSR-Vector [7]. In CSR-Scalar, each thread processes one row of a sparse matrix. If each row of the sparse matrix has a low number of average nonzero elements, such as 1∼2, CSR-Scalar offers better performance. However, when the number of average nonzero elements per row is large, the performance is lower

due to uncoalesced memory accesses. On the other hand, unlike CSR-Scalar, in CSR-Vector each warp processes one row of the matrix. This coalesced memory access improves memory access efficiency greatly. However, when short rows are encountered, the redundant threads in the warp remain idle, leading to low computational resource utilization.

In an effort to solve the shortcomings of CSR-Scalar and CSR-Vector, Liu and Schmidt propose LightSpMV [9], splitting warps into vectors, where each vector owns *vector_size* threads and processes one row of the matrix. LightSpMV first calculates the number of row-averaged nonzero elements of the sparse matrix to determine the appropriate *vector_size* before performing the SpMV. This approach alleviates the problems of low thread utilization and uncoalesced memory access. However, if the variance of the row lengths of the matrix is very large, using a fixed *vecctor_size* is not effective in solving the above problem.

As regards to the memory accesses in GPUs, the access latency to global memory is the largest. For this reason, each streaming multiprocessor in a GPU is equipped with shared memory, accessible to CUDA cores within the streaming multiprocessor with a lower access latency than that of the global memory. Based on this hardware feature of GPUs, Greathouse and Daga propose the CSR-Stream [12], which first batch-copies nonzero elements from global memory to shared memory, then accumulates the nonzero elements of each row and writes the result back to global memory. It accesses nonzero elements in a coalesced manner, greatly improving the memory access efficiency. The CSR-Stream proves that copying data to shared memory at first and then distributing threads to compute the sum of each row is a better solution, largely thanks to shared memory having low access latency and friendly to uncoalesced memory access. However, copying data from GPU global memory to shared memory takes up thread resources. Meanwhile, as compared with other methods, it has two more accesses to shared memory, leading to poor performance on sparse matrices with an average row length greater than 32 without fully utilizing the advantages of shared memory for SpMV.

NVIDIA has recently proposed the Ampere GPU architecture [22], which implements a new asynchronous copy instruction, `memcpy_async`. It allows data loading asynchronously from global memory to shared memory. In other words, no additional thread resources are required for copying data from global memory to shared memory, thus freeing up the thread to completely focus on computational operations during such data movement. We believe that the new capability provided by `memcpy_async` in the Ampere architecture can be applied for accelerating SpMV operation. Particularly in this paper, we explore the opportunity of overlapping the processes of "copying nonzero elements from global memory to shared memory" and "vector processes of each matrix row" using `memcpy_async` to take full advantage of shared memory for SpMV to achieve overall improved performance.

In addition, before performing a SpMV operation, nonzero elements are copied from host to device. Actually, in our own experiments, we note that this process is very time-consuming, as much as more than 10 times that of the SpMV operation itself. The NVIDIA GPU attempts to solve this via the CUDA library with the concept of streams [11], where a stream is a sequence of commands that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently. Streams using the `cudaMemcpyAsync` function are capable of overlapping the "copy nonzero elements from host to device" and "SpMV kernel execution", thus improving performance by hiding the latency of data movement for SpMV.

3. Background

3.1. Sparse Matrix-Vector Multiplication

In this paper, we investigate the general SpMV equation $\vec{y} = \alpha A\vec{x} + \beta\vec{y}$ [23], where A is a sparse $m \times n$ matrix. A has N_{nz} nonzero elements. $\vec{x}$ is a dense vector of size n and $\vec{y}$ is a dense vector of size m. α and β are scalars. Let A_{ij} denote the element of A at position (i, j), x_i and y_i denote the i-th element of vector $\vec{x}$ and $\vec{y}$. Note that the result of SpMV is finally written back into the vector $\vec{y}$, thus the vector $\vec{y}$ on the right-hand side of

the equation denotes the old value, and the vector $\vec{y}$ on the left-hand side denotes the new value. SpMV can be expressed as

$$y_i = \beta \cdot y_i + \alpha \cdot \sum_{A_{ij} \neq 0} A_{ij} \cdot x_j \quad (1)$$

From Equation (1), we know that there are a total of $2 \times (N_{nz} + m)$ floating point operations (FLOPs) for a single execution of SpMV.

3.2. Compressed Sparse Row

In this paper, we focus on the SpMV optimization using the widely used sparse matrix storage format, namely the *Compressed Sparse Row* (CSR). Using the same $m \times n$ matrix A with N_{nz} nonzero elements, Figure 1 below illustrates that CSR achieves efficient memory usage by storing nonzero values in three arrays $row\_offsets$, $column\_indices$ and $values$.

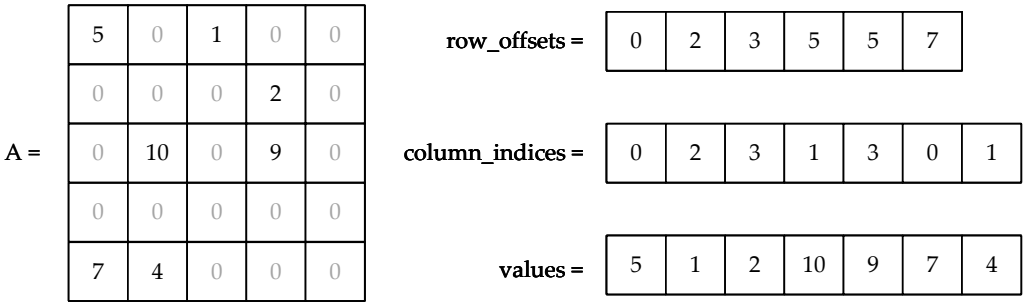

Figure 1. CSR representation of an example sparse matrix A.

Note that the CSR format uses 0-based indices, i.e., the index for the first row or column starts from 0. The arrays $values$ and $column\_indices$ are of length N_{nz}, and contain the nonzero values and the column indices of those values, respectively. The array $row\_offsets$ is of length $m + 1$ and encodes the index in $values$ and $column\_indices$ where the given row starts. This is equivalent to $row\_offsets[j]$ encoding the total number of nonzeros above row j. $row\_offsets[m]$ is equal to N_{nz}. Figure 1 is an example. Algorithm 1 shows the pseudo-code of CSR-based SpMV using CPU.

Algorithm 1: Sequential SpMV based on CSR format

```
1 for i = 0; i < m; ++i do
2     sum = 0;
3     for j = row_offsets[i]; j < row_offsets[i + 1]; ++j do
4         sum += values[j] * x[column_indices[j]];
5     y[i] = α * sum + β * y[i];
```

3.3. Ampere GPU Architecture

NVIDIA GPUs contain multiple streaming multiprocessors (SMs), where each SM contains all necessary computation resources including CUDA cores, scheduler, dispatch units, shared memory, register files, L1 cache, etc. Shared memory is accessed by the corresponding CUDA cores in the SM they belong to with support for uncoalesced accesses [10,11]. This avoids considerable access latency as compared to global memory when memory accesses are uncoalesced. Therefore, it is natural to design the SpMV computation such that we partition the data into batches to batch-copy these data from the global memory to the shared memory for the SM to perform compute on the data [12].

Recently, NVIDIA has released a new GPU architecture, namely the Ampere. The Ampere comes with a new powerful asynchronously shared memory copy capability, which is implemented in CUDA as the `memcpy_async` instruction [22]. As its name suggests, it offers efficient asynchronous data copy from global memory to shared memory. This means

that copying data using `memcpy_async` no longer occupies thread resources, freeing up the thread during data movement for computation. Thus, we can leverage this mechanism to achieve efficient SGOL to improve the performance. In addition, the Ampere implements a neat hardware acceleration capability to bypass for L1 cache in `memcpy_async`, copying data directly from global memory to shared. Figure 2 shows the difference between `memcpy_async` and without `memcpy_async` when copying data.

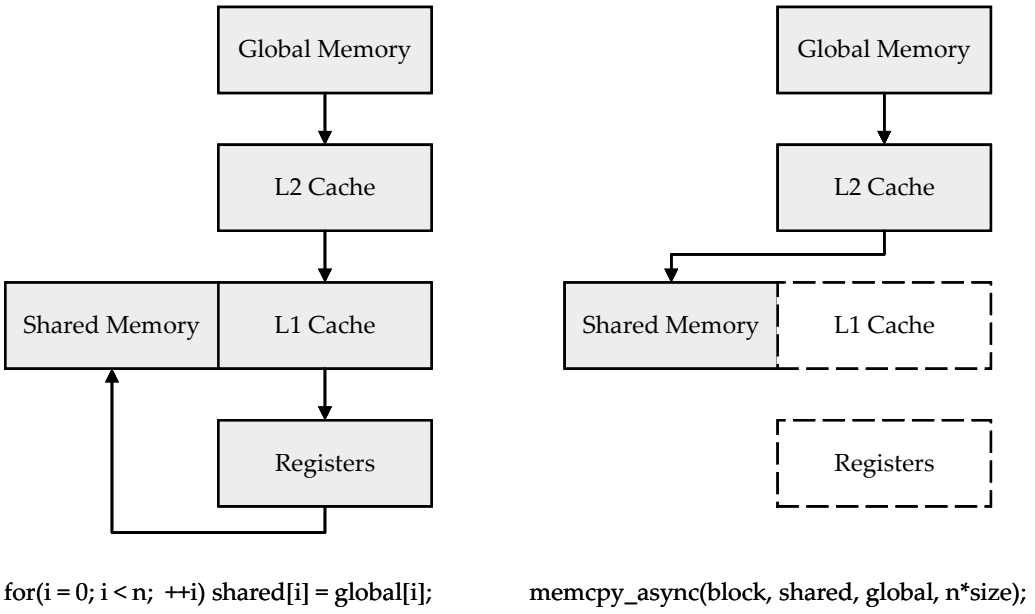

Figure 2. Illustration of the process with and without `memcpy_async` for data copying from global memory to shared memory.

Figure 2 offers an illustrative example of two ways for data copying from global memory to shared memory, where the array *shared* is stored in shared memory and the array *global* is stored in global memory. When using `memcpy_async`, data avoids taking a long journey through the memory hierarchy, freeing the thread block as well as related registers from the task of moving data to focus on compute-centric tasks.

3.4. CUDA Streams

Operations within streams in CUDA are executed sequentially, while operations in different streams can be interleaved and, if possible, can be executed concurrently. The `cudaMemcpyAsync` function is used to copy data from the host memory to the device global memory asynchronously, designed to only be called by the host side. The `memcpy_async` function, on the other hand, can only be called by the device side, copying data asynchronously between global memory and shared memory in the device. Streams use the `cudaMemcpyAsync` function to achieve the overlap of data transfer from host to device and SpMV kernel execution. Algorithm 2 provides a simple code example showing how `cudaMemcpyAsync` is used, while Figure 3 provides an illustrative view of how the data movement and kernel execution are overlapped.

Algorithm 2: Example of overlapping data transfer with kernel execution

```
1 for i = 0; i < 2; ++i do
2     cudaMemcpyAsync(dev_in + i * size, host_in + i * size, size,
        cudaMemcpyHostToDevice, stream[i]);
3     kernel<<<128, 512, 0, stream[i]>>>((dev_in + i * size, dev_out + i * size, size);
4     cudaMemcpyAsync(host_out + i * size, dev_out + i * size, size,
        cudaMemcpyDeviceToHost, stream[i]);
```

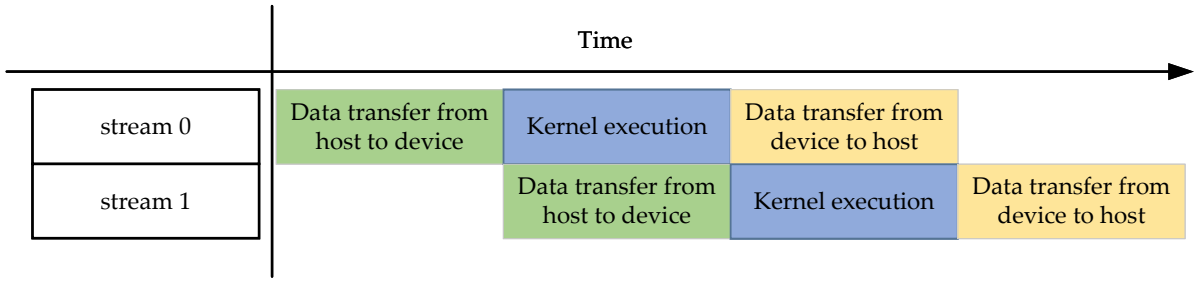

Figure 3. Example view of overlapping the data transfer and kernel execution for two streams.

4. Proposed Methods

The goal of this work is to describe a high-performance SpMV using CSR storage format on GPU, hiding the computation as much as possible in the two important access behaviors of the GPU. We first design the CSR-Partial-Overlap, with a two-staged pipeline design using `memcpy_async`, targeted directly at GSOL-type overlapping. We further design the *dynamic batch partition* algorithm for the CSR-Partial-Overlap to remove the overhead of fixing up partial sums and allow low cost and effective load balancing. In addition, we design the *dynamic threads distribution* algorithm for the computation task in the two-staged pipeline, to improve thread utilization. Finally, based on the CSR-Partial-Overlap, we propose the CSR-Full-Overlap, which is not only designed for HDOL-type overlapping but also further unifies the two major overlap types for best overall performance improvement.

4.1. CSR-Partial-Overlap

To solve the shortcoming of existing SpMV CSR algorithms discussed in Section 2, we propose the CSR-Partial-Overlap algorithm with both faster data copying and concurrency in computing and data copying. We aim to achieve these benefits by carefully and smartly applying `memcpy_async` to SpMV in CSR-Partial-Overlap.

Since the size of shared memory is fixed, the data is partitioned into batches before performing SpMV. We design a two-staged pipeline for CSR-Partial-Overlap to overlap data copying and computation. In addition, we dynamically distribute threads for batches to obtain better performance in data computing.

4.1.1. Partition Nonzero Elements into Batches

An intuitive strategy for batch partition is to have a fixed number of nonzero elements per batch, as shown in Figure 4. However, this approach leads to nonzero elements belonging to the same row being partitioned into two different batches. For example, in Figure 4, row4 is partitioned into batch1 and batch2, where batch1 and batch2 may be assigned to different GPU blocks for computation. The batch would have to save the partial sums of the split rows to the global memory during the computation and wait for all batch computations to finish before starting another kernel to accumulate the partial sums and write them back. Consequently, this approach results in additional global memory accesses and additional kernel overhead for partial sums accumulation. Moreover, writing the computation result of row back to $\vec{y}$ is also a non-trivial overhead since batches are likely to differ in number of rows, implying that loads from different batches are not sufficiently well-balanced.

To mitigate the above problem, we design a dynamic batch partition algorithm (executed on the host side), where batches have different numbers of nonzero elements, limited only by *max_batch_size*. The steps to partition the batches are as follows:

1. Pick out the extra-long rows whose row length is greater than *max_batch_size* and record them in the $long\_row\_infos$ array, as shown in line 3 in Algorithm 3;
2. Fill each row into current the batch sequentially. Stop when the number of nonzero elements in the current batch is larger than *max_batch_size* or when it encounters an

extra-long row. Record the start and end rows of the current batch in the $batch\_infos$ array, as shown in line 7 to 19 in Algorithm 3;

3. Continue to fill the remaining rows into the next batch until all rows have been partitioned into the corresponding batch.

Algorithm 3: Dynamic nonzero elements batch partition algorithm

```
1  // find extra-long rows and save them to long_row_infos
2  for i = 0; i < m; ++i do
3      if (row_offsets[i + 1] - row_offsets[i]) > max_batch_size then long_row_infos.append(i);
4  // define a stack that can accumulate row lengths and check for overflow when
       adding elements
5  stack s(max_batch_size);
6  for current_row = 0; current_row < m; ++current_row do
7      if current_row is long row then
8          // skip long row and save stack information to batch_infos
9          if s.empty() == false then
10             batch_infos.append(s.start, s.end + 1);
11             s.clean();
12         continue;
13     current_row_len = row_offsets[current_row + 1] - row_offsets[current_row];
14     // use s.push() to accumulated row length and check for overflow
15     if s.push(current_row, current_row_len) != 0 then
16         // the currently accumulated row length is greater than max_batch_size.
               save stack information to batch_infos
17         batch_infos.append(s.start, s.end + 1);
18         s.clean();
19         s.push(current_row, current_row_len);
20 // save remaining rows to batch_infos
21 if s.empty() == false then batch_infos.append(s.start, s.end + 1);
```

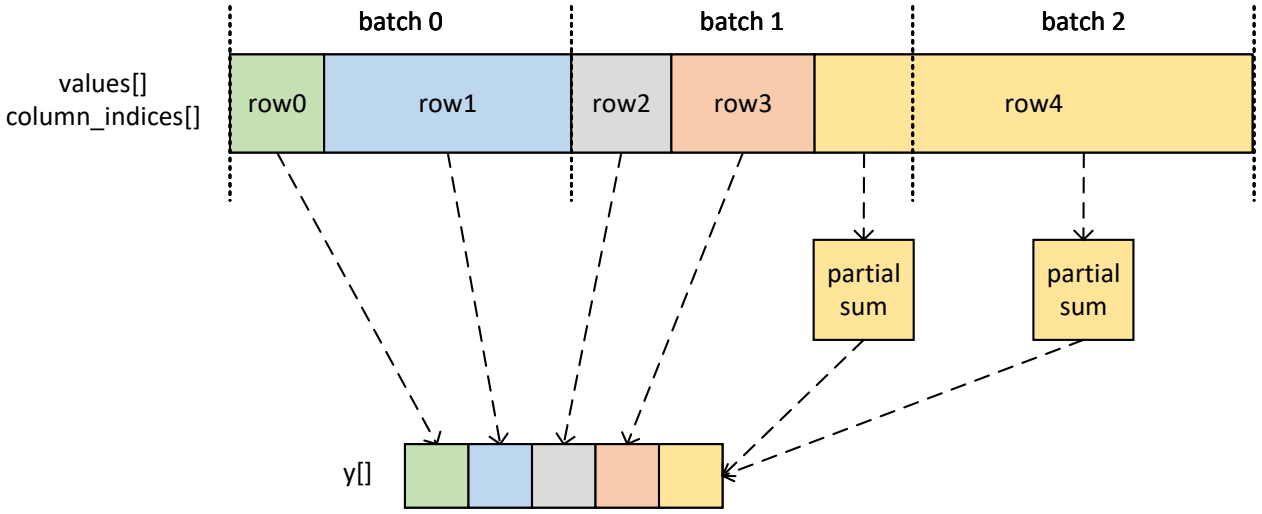

Figure 4. A static batch partition results in the same number of nonzero elements in all batches.

Figure 5 shows an example of the working principle of the above dynamic batch partition algorithm, with the corresponding pseudo-code presented in where Algorithm 3.

As we can see, such a dynamic batch partition method ensures that the same row is not partitioned into different batches, avoiding the overhead from accumulating partial row sums. In our observation, when max_batch_size is large enough, e.g., max_batch_size is larger than 1024, after eliminating extra-long rows larger than max_batch_size, interestingly, batches do not differ much in their numbers of nonzero elements. In addition, the complexity of the batch partition is only related to m, not to N_{nz}. This implies that the overhead for performing the batch partition does not increase significantly when the sparse

matrix size increases. Moreover, since the result of batch partition is only related to the number of nonzero elements in each row of the given matrix, there is no need to re-partition the batches whenever the values of its nonzero elements are changed. In addition, the latency overhead of using the CPU to partition the batches can be hidden in the data transfer process between the host and the device. This partitioning method makes each batch have a similar load, and the batches are evenly distributed among GPU blocks to execute. Instead of striving for extreme load balancing, like merge-based, we achieved effective load balancing with a low-cost partitioning method.

max_batch_size	64						
row_id	0	1	2	3	4	5	6
row_length	32	15	16	11	8	38	256
row_id	7	8	9	10	11	12	
row_length	25	16	130	2	22	33	

batch_num	4				
batch_infos	batch_id	0	1	2	3
	start_row	0	3	7	10
	end_row	3	6	9	13

long_row_num	2		
long_row_infos	long_row_id	0	1
	long_row	6	9

Figure 5. Illustration of dynamical batch partition, where each resulting batch can differ in their number of nonzero elements.

4.1.2. Two-Staged Pipeline Processing

As CSR-Partial-Overlap is a CUDA kernel, We assume that the kernel is configured with B GPU blocks and T threads per block. In practice, we usually set B to an integer multiple of the number of SMs on the GPU and correspondingly set T to the number of CUDA cores for each SM. To balance the load, each block processes *block_batch_num* batches, where $block\_batch\_num = \frac{total\ number\ of\ batches}{B}$.

For convenience of discussion, Figure 6 illustrates an example two-staged pipeline design used in CSR-Partial-Overlap. In this example, each block processes three batches. All batches are added to a queue and go through the pipeline in order. Stage 1 of the pipeline is to copy the *values* array and *column_indices* array from the global memory to shared memory using `memcpy_async`, as shown in line 12 to 17 in Algorithm 4. Stage 2 of the pipeline is the computation task, i.e., computing the sum of the nonzero elements of each row in the batch and writing the result back to the $\vec{y}$ vector, as shown in line 19 to 23 in Algorithm 4. We use the term *computation* to refer to this computation task. We configure shared memory of size S for each block, where $S = 2 \times max\_batch\_size \times (\texttt{sizeof}(T\_VAL) + \texttt{sizeof}(T\_COL))$. This shared memory space is partitioned into two equal-sized sub shared memory spaces, as denoted by $S0$ and $S1$ in Figure 6. $S0$ and $S1$ form a shared memory resources pool for pipeline to use.

Note that in the beginning, both $S0$ and $S1$ are free. In the above example, we load the data of *batch*0 into $S0$. After $S0$ completes the data loading, the block performs computation for *batch*0, while $S1$ loads the data of *batch*1. The *batch*0 stays in $S0$ while the computation is being performed, which is released when the computation is complete. Whenever $S0$ is released, it can immediately be used to load data of *batch*2. The pipeline repeats this behavior until all batches have been computed. Using the same example earlier, we overlap the loading of *batch*1 and the computation of *batch*0, and the loading of *batch*2 and the computation of *batch*1. Algorithm 4 shows the pseudo-code for this process.

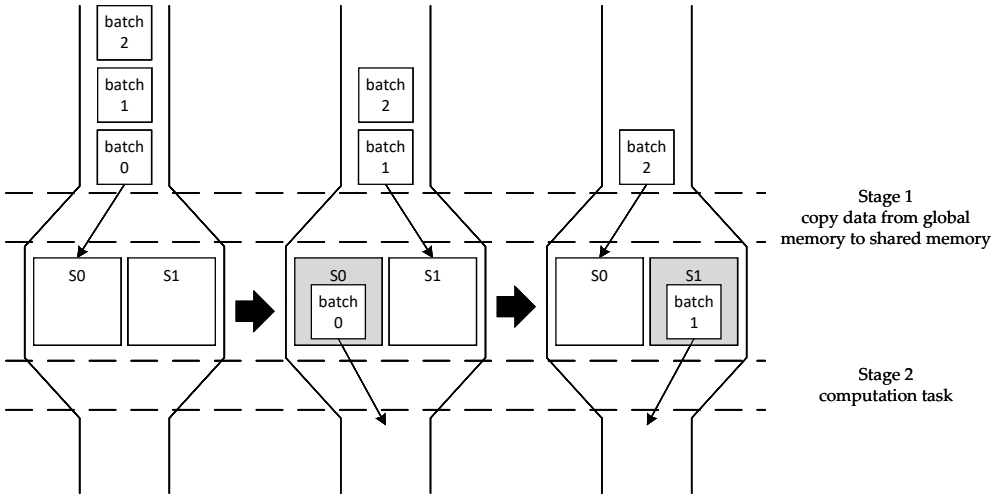

Figure 6. Illustration of two-staged pipeline. All batches are added to a queue and go through the pipeline in order.

Algorithm 4: Copying data from global memory to shared memory overlapping with computation

```
// this is csr_partial_overlapping() kernel
__shared__ sval[max_batch_size * 2], scol[max_batch_size * 2];
shared_offset[2] = {0, max_batch_size};
// extra block return
start_batch = blockIdx.x * block_batch_num;
end_batch = min(start_batch + block_batch_num, batch_num);
if end_batch <= start_batch then return;
// overlapping data copying and computation
for compute_batch=0,fetch_batch=0; compute_batch<(end_batch-start_batch);
 ++compute_batch do
    for ; fetch_batch<(end_batch-start_batch)&&fetch_batch<(compute_batch+2);
     ++fetch_batch do
        // fetch data
        pipeline.producer_acquire();
        shared_idx = fetch_batch % 2;
        // copy data from global memory to shared memory using memcpy_async
        cuda::memcpy_async(block, sval + shared_offset[shared_idx], values +
          batch_start_nnz, batch_nnz_len * sizeof(T_VAL), pipeline);
        cuda::memcpy_async(block, scol + shared_offset[shared_idx], column_indices +
          batch_start_nnz, batch_nnz_len * sizeof(T_COL), pipeline);
        pipeline.producer_commit();
    // compute while the block is copying data
    pipeline.consumer_wait();
    block.sync();
    shared_idx = compute_batch % 2;
    compute(batch_start_row, batch_end_row, batch_start_nnz, batch_end_nnz,
      row_offsets, scol + shared_offset[shared_idx], sval + shared_offset[shared_idx], ...);
    pipeline.consumer_release();
```

4.1.3. Dynamic Thread Distribution For Batches

In this part, we introduce the dynamic threads distribution when batches computing mentioned above. The SM in the NVIDIA GPU executes threads in groups of 32 parallel threads called warps [11]. In Section 2, we mentioned that the distribution of one warp

per matrix row does not work well for all matrices, since the average row lengths of these matrices are varied. Therefore, we split warps into smaller vectors, each having *vector_size* threads, where $vector\_size \in \{1, 2, 4, 8, 16, 32\}$. Each vector computes one row of the matrix at a time, multiplies and accumulates the nonzero elements of that row, and writes the sum back to the vector $\vec{y}$, as shown in line 8 to 19 in Algorithm 5. After the vector completes the computation of a row, it will distribute the next matrix row until all rows within the batch have been computed, as shown in line 21 in Algorithm 5. Algorithm 5 presents the detail of this logic in pseudo-code. Note that the average row length of each batch is variable, to minimize thread idling, we dynamically select *vector_size* based on the average row length of the batch, as shown in Algorithm 6. This method improves thread utilization, which improves the overall performance of the CSR-Partial-Overlap.

Algorithm 5: Vector process nonzero elements in the batch

```
1  // this is vector_compute() function
2  block_vector_id = threadIdx.x / VECTOR_SIZE;
3  block_vector_num = blockDim.x / VECTOR_SIZE;
4  lane_id = threadIdx.x & (VECTOR_SIZE - 1);
5  // get first row
6  current_row = batch_start_row + block_vector_id;
7  while current_row < batch_end_row do
8      nnz_start = row[current_row];
9      nnz_end = row[current_row + 1];
10     // compute the dot product of the vector
11     sum = 0;
12     for i = nnz_start + lane_id; i < nnz_end; i += VECTOR_SIZE do
13         sum += sval[i - batch_start_nnz] * x[scol[i - batch_start_nnz]];
14     // intra-vector reduction
15     for i = VECTOR_SIZE ≫ 1; i > 0; i ≫= 1 do
16         sum += __shfl_down_sync(__activemask(), sum, i, 32);
17     // save the result
18     if lane_id == 0 then
19         y[current_row] = α * sum + β * y[current_row];
20     // get a new row
21     current_row += block_vector_num;
```

It is not too difficult to see that the proposed CSR-Partial-Overlap focuses only on the GSOL type of overlapping. It fully utilizes the time of data transfer between the global and the shared memory for computation. To the best of our knowledge, this is the first time this aspect has been considered in accelerating SpMV computations. In addition, as described in Section 4.1.1, CSR-Partial-Overlap benefits from a dynamic batch partition scheme that fixes the *max_batch_size* but not the number of nonzero elements within the batch, where each batch dynamically selects the *vector_size* based on the average row length. Compared to existing merge-based SpMV alternatives, CSR-Partial-Overlap achieves a low cost and effective load balancing and great performance for matrices with low N_{nz}/m, and is much faster than CSR-Stream in the case of a large N_{nz}/m.

4.2. CSR-Full-Overlap

Next, based on CSR-Partial-Overlap, we describe how to unify the HDOL type of overlapping, maximizing the benefits for accelerating SpMV performance as much as we can. Note that before using GPUs for SpMV, the data needs to be copied from the host to the device. This process causes considerable latency overhead, as much as more than 10 times that of SpMV computation itself. Therefore, we propose CSR-Full-Overlap to utilize the asynchronous copying feature of CUDA Streams to hide the SpMV kernel execution in the data transfer to improve the performance greatly.

Algorithm 6: Select vector_size to compute according to mean

```
// this is compute() function
row_num = batch_end_row - batch_start_row;
nnz_num = batch_end_nnz - batch_start_nnz;
mean = nnz_num/row_num;
// set vector size according to mean
if mean < SWITCH_POINT_1 then
    vector_compute<1>(...);
else if mean < SWITCH_POINT_2 then
    vector_compute<2>(...);
else if mean < SWITCH_POINT_4 then
    vector_compute<4>(...);
else if mean < SWITCH_POINT_8 then
    vector_compute<8>(...);
else if mean < SWITCH_POINT_16 then
    vector_compute<16>(...);
else
    vector_compute<32>(...);
```

The core of the proposed CSR-Full-Overlap relies on the fact that it unifies two types of overlapping in GPUs as introduced in Section 1. By hiding the SpMV computation in both types of data transfer, we are able to achieve substantial performance acceleration for SpMV.

The data transfer overhead between the host and the device is not only the transfer time overhead, but also the startup time overhead. Therefore, the amount of data transferred at a time has to be large enough to be cost-effective. Note that the transfer mentioned in this part refers to the transfer between the host and the device. Based on this, we partition the batches into big batches. During the experiments, we realize that when the amount of data is different for each transfer, it will lead to an additional time overhead. This overhead may seem to be relatively a small portion as compared to the total data transfer time, but nonetheless contributes to the execution time of the SpMV kernel and thus cannot be ignored.

Therefore, unlike the partition method in Section 4.1.1, where each big batch transfers the same *big_batch_size* nonzero elements (except for the last big batch), Figure 7 shows an example of the big batch partition, where nine batches are partitioned into two big batches and $big\_batch\_size = 1000$. Note that a batch can be partitioned into two parts to transfer data, but it can only be processed by one kernel, in which case we assign the batch to the latter big batch kernel to process, as shown in line 9 and 15 in Algorithm 7. In this example, $batch4$ is processed by the $big\ batch1$'s kernel. The range of transfer data of $big\ batch0$ is $[0, 1000)$, and the range of transfer data of $big\ batch1$ is $[1000, 2000)$. The batches processed by the kernel of $big\ batch0$ are $[batch0, batch3]$, and the batches processed by the kernel of $big\ batch1$ are $[batch4, batch8]$. We use big_batch_infos array to store the partition information mentioned above. The pseudo-code is shown in Algorithm 7.

After partitioning batches into big batches, we overlap the data transfer and kernel execution using CUDA's streams feature. Each big batch is distributed to a stream, and the *column_indices* array and *values* array within the big batch are first copied to the device using `cudaMemcpyAsync`. After that, the csr_partial_overlapping kernel in Algorithm 4 is called to perform the SpMV operation. The pseudo-code is shown in Algorithm 8. Since commands are executed concurrently between different streams, the SpMV kernel execution of a stream overlaps with the data transfer of the next stream.

At this point, we have completed the unification of the two overlaps in SpMV, and CSR-Full-Overlap can gain performance improvement by two overlaps at the same time. It is worth mentioning that if the *big_batch_size* is set to N_{nz}, which means that a single

big batch contains all the batches, in this case CSR-Full-Overlap will be equivalent to CSR-Partial-Overlap.

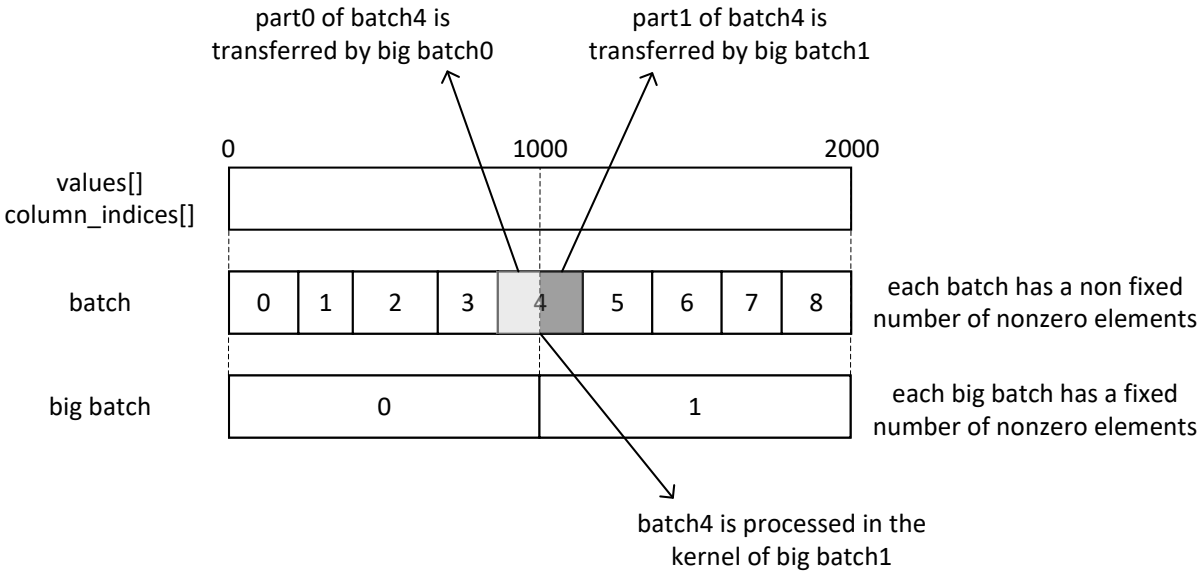

Figure 7. Illustration of big batch partition, where the *big_batch_size* is fixed.

Algorithm 7: Partition batches into big batches

```
1  big_batch_num = ceil(Nnz / big_batch_size);
2  // compute the start_nnz and end_nnz of big_batchs
3  for i = 0; i < big_batch_num; ++i do
4      big_batch_infos[i].start_nnz = i * big_batch_size;
5      big_batch_infos[i].end_nnz = min((i + 1) * big_batch_size, Nnz);
6  // determine which batches the kernel will execute for each big batch
7  for i = 0; i < big_batch_num; ++i do
8      for j = 0; j < batch_num; ++j do
9          if row_offsets[batch_infos[j].end_row] > big_batch_infos[i].start_nnz then
10             big_batch_infos[i].start_batch = j;
11             break;
12         if j == batch_num - 1 then
13             big_batch_infos[i].start_batch = batch_num;
14     for j = batch_num - 1; j >= 0; j = j - 1 do
15         if row_offsets[batch_infos[j].end_row] <= big_batch_infos[i].end_nnz then
16             big_batch_infos[i].end_batch = j + 1;
17             break;
```

4.3. Processing of Extra-Long Rows

For rows with a number of nonzero elements larger than *max_batch_size*, we process them after all streams have finished transferring data. Due to the large load of extra-long rows, we distribute a row to a block for processing. Multiple warps within the block process a row together, using shared memory as a temporary space to save the partial sums of each warp, as shown in line 9 to line 22 in Algorithm 9. Finally, the first warp within the block accumulates these partial sums and writes the result back to the $\vec{y}$ vector, as shown in line 24 to 32 in Algorithm 9. The pseudo-code for this process is shown in Algorithm 9. Having so many nonzero elements within a matrix is a rare case in practice, and launching an extra kernel to process these rows causes additional overhead. We can reduce the probability of this happening by increasing *max_batch_size*.

Algorithm 8: Nonzero elements transfer from host to device overlapping with SpMV kernel execution

```
for i = 0; i < big_batch_num; ++i do
    start_nnz = big_batch_infos[i].start_nnz;
    end_nnz = big_batch_infos[i].end_nnz;
    len = end_nnz - start_nnz;
    // data transfer from host to device
    cudaMemcpyAsync(dcolumn_indices + start_nnz, column_indices + start_nnz, len *
      sizeof(T_COL), cudaMemcpyHostToDevice, streams[i]);
    cudaMemcpyAsync(dvalues + start_nnz, values + start_nnz, len * sizeof(T_VAL),
      cudaMemcpyHostToDevice, streams[i]);
    batch_offset = big_batch_infos[i].start_batch;
    current_batch_num = big_batch_infos[i].end_batch - big_batch_infos[i].start_batch;
    // kernel execution
    csr_partial_overlapping<<<B, T, S, streams[i]>>>(batch_infos + batch_offset, current_batch_num,
      ...);
```

Algorithm 9: Each long row is allocated to a block to process

```
// this is long row process kernel
warp_id = threadIdx.x / 32;
lane_id = threadIdx.x & (32 - 1);
__shared__ shared_sum[gridDim.x / 32];
// get a long row id
current_long_row_id = blockIdx.x;
while current_long_row_id < long_row_num do
    // get the row
    current_row = long_row_infos[current_long_row_id];
    nnz_start = row[current_row];
    nnz_end = row[current_row + 1];
    // compute the dot product of the warp
    sum = 0;
    for i = nnz_start + threadIdx.x; i < nnz_end; i += blockDim.x do
        sum += values[i] * x[column_indices[i]];
    // intra-warp reduction
    for i = 32 >> 1; i > 0; i >>= 1 do
        sum += __shfl_down_sync(__activemask(), sum, i, 32);
    // save intermediate results to shared memory
    if lane_id == 0 then
        shared_sum[warp_id] = sum;
    __syncthreads();
    // intra-block reduction
    if threadIdx.x < 32 then
        block_sum = 0;
        for i = threadIdx.x; i < block_warp_num; i += 32 do
            block_sum += shared_sum[i];
        for i = 32 >> 1; i > 0; i >>= 1 do
            block_sum += __shfl_down_sync(__activemask(), block_sum, i, 32);
        // save the result
        if threadIdx.x == 0 then
            y[current_row] = α * block_sum + β * y[current_row];
    // get a new long row id
    __syncthreads();
    current_long_row_id += gridDim.x;
```

5. Performance Evaluation

5.1. Experimental Setup

We use the SuiteSparse Matrix Collection, a publicly available and widely used set of sparse matrix benchmarks collected from a wide range of applications [2]. To evaluate the performance of the proposed method, we select a set of matrices with representative

matrix sparsity characteristics from the SuiteSparse dataset. Details of this set are shown in Table 1. The N_{nz} of these matrices ranges from 73 K to 14.8 M. The average row length ranges from 2.6 to 158.5. These matrices represent sparse features that are most common in a wide variety of applications.

Table 1. Information of sparse matrices used.

Name	Nonzeros (N_{nz})	N_{nz}/m	m	n
ncvxqp1	73,963	6.107	12,111	12,111
aug2d	76,832	2.649	29,008	29,008
case9	147,972	10.237	14,454	14,454
delaunay_n16	393,150	5.999	65,536	65,536
ch7-9-b3	423,360	4	105,840	17,640
tandem_dual	460,493	4.895	94,069	94,069
epb3	463,625	5.479	84,617	84,617
ch8-8-b3	470,400	4	117,600	18,816
cage11	559,722	14.322	39,082	39,082
Si5H12	738,598	37.123	19,896	19,896
nemeth19	818,302	86.083	9506	9506
tube1	897,056	41.727	21,498	21,498
kim1	933,195	24.292	38,415	38,415
2D_54019_highK	996,414	18.446	54,019	54,019
nemeth21	1,173,746	123.474	9506	9506
nemeth22	1,358,832	142.945	9506	9506
nemeth23	1,506,810	158.511	9506	9506
Hamrle3	5,514,242	3.810	1,447,360	1,447,360
pre2	5,959,282	9.042	659,033	659,033
pkustk14	14,836,504	97.656	151,926	151,926

All tests are conducted on a workstation with two AMD EPYC 7713 64-Core CPUs and 377 GB RAM, running the Ubuntu 20.04 Linux operating system. The workstation has been further equipped with an Ampere-based A40 GPU, which comprises 84 SMs with a total of 10,752 CUDA cores. It also comes with a total of 48 GB global memory. Each SM has 48 KB shared memory. All of the CUDA-based programs used for evaluation in this paper are compiled using the CUDA 11.7 toolkit with standard practice switches "`-arch sm_86 -O3`", corresponding to the target GPU architecture and level 3 optimization.

5.2. Optimal Dynamic vector_size Selection

In Section 2, we mentioned that the larger the average row length of the matrix was, the larger the *vector_size* for obtaining optimal performance would be. In other words, it is advantageous to process matrices with shorter average row lengths when $vector\_size = 1$ and, on the contrary, it is advantageous to process matrices with longer average row lengths when $vector\_size = 32$. Therefore, we segment the average row length into intervals, and each interval has a corresponding *vector_size* that makes the performance optimal on that interval. The boundary point between intervals we call the optimal switching point. In this experiment, we generate a series of random matrices with average row lengths ranging from 1 to 256. The range of average row lengths is large enough to cover all the optimal switching points. We then evaluate the performance of the CSR-Partial-Overlap on these random matrices. In each round of experiments, unlike Algorithm 4, we set *vector_size* to a fixed number in turn, where $vector\_size \in \{1, 2, 4, 8, 16, 32\}$. In this way, we can measure the optimal *vector_size* corresponding to each interval. The result is shown in Table 2.

Table 2. The optimal *vector_size* for each average row length interval.

N_{nz}/m	vector_size
[0, 17)	1
[17, 34)	2
[34, 64)	4
[64, 122)	8
[122, 216)	16
[216, ∞)	32

5.3. Evaluation of CSR-Partial-Overlap

For practical evaluation, we use the standard metric of billion floating-point operations per second (GFLOPs), which is computed as $\frac{2(N_{nz}+\mathrm{m})}{t_{\mathrm{kernel}}\times 10^9}$ for Equation (1), where t_{kernel} is the execution time of the CUDA kernel measured in seconds, excluding the host to device data transfer time.

cuSPARSE is a publicly available library from NVIDIA with GPU-accelerated basic linear algebra subroutines for handling sparse matrices [23]. cuSPARSE has excellent performance among the existing SpMV algorithms based on the CSR storage format and is thus used in this paper as a baseline for our comparative study.

We compare the proposed CSR-Partial-Overlap with three well-known SpMV algorithms based on CSR storage formats in the literature, namely cuSPARSE [23], LightSpMV [9], and merge-based [17]. Figure 8 illustrates the performance of various implementations in single precision on the collection of matrices listed in Table 1. Note that there is no single algorithm which is best for all the matrices. However in most cases, CSR-Partial-Overlap either outperforms all other algorithms or performs equally well. CSR-Partial-Overlap's performance is far ahead of other algorithms for matrices with average row lengths less than 16. For example, CSR-Partial-Overlap's performance on aug2d matrices is 2.2 times that of cuSPARSE. CSR-Partial-Overlap attains an average throughput of 100.1 GFLOPs compared to 71.6 for cuSPARSE [23], 71.7 for LightSpMV [9], and 71.1 for merge-based [17] GFLOPs. On all matrices CSR-Partial-Overlap obtains the best average performance of 1.4× that of cuSPARSE.

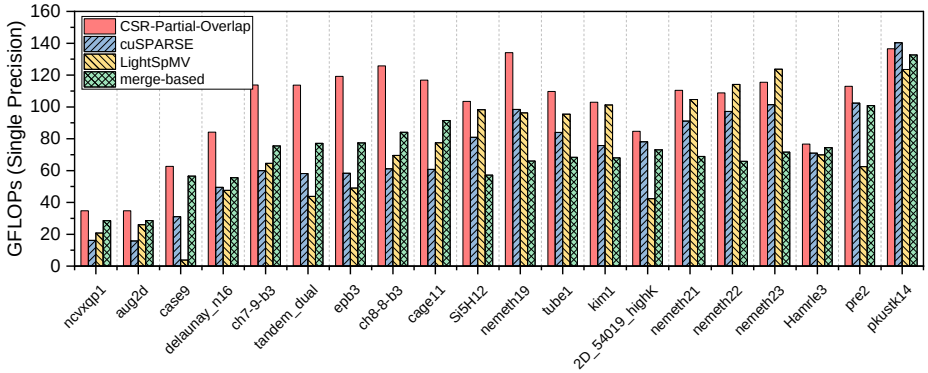

Figure 8. Performance evaluation of CSR-Partial-Overlap.

5.4. Impact from big_batch_size

One interesting observation from our experiment is the impact from *big_batch_size* on the proposed CSR-Full-Overlap algorithms. Recall, as we have discussed in Section 4.2, that there is a host to device nonzero elements transfer time overhead of up to 10 times of SpMV computation itself. In CSR-Full-Overlap, the data transfer of each big batch overlaps with the SpMV kernel execution of the next big batch. Therefore, the SpMV kernel execution of the last big batch is not overlapped, i.e., its time overhead is not hidden in data transfer. Intuitively, the smaller the execution time of the last big batch kernel, the shorter the total time is, thus a smaller *big_batch_size* is always preferred. However, on

the contrary, we find that this is not the case since the smaller the *big_batch_size*, the larger the *stream_num* leading to additional overhead due to the increase in streams switching overhead and startup data transfer is. In our estimation, these overheads are equivalent to 0.2 $\sim$ 7% of the total transfer time and thus should not be ignored. Therefore, in this paper, we choose $stream\_num = 2$, i.e., correspondingly $big\_batch\_size = \frac{N_{nz}}{2}$, to avoid above potential overheads.

5.5. Evaluation of CSR-Full-Overlap

To evaluate the performance of the proposed CSR-Full-Overlap, we compare it with cuSPARSE. For a more objective and realistic view, we design the speedup metric using the following formula:

$$\text{Speedup} = \frac{t_{\text{cuSPARSE}} - t_{\text{CSR-Full-Overlap}}}{t_{\text{CSR-Partial-Overlap}}} + 1, \tag{2}$$

The $t_{cuSPARSE}$ in Equation (2) is the total time for cuSPARSE to perform data transfer and SpMV, where the data transfer time is the time spent to transfer the *column_indices* array and *values* array from the host to the device. Similarly, the $t_{\text{CSR-Full-Overlap}}$ is the total time for CSR-Full-Overlap to perform data transfer and SpMV. The $t_{\text{CSR-Partial-Overlap}}$ is the time taken by this matrix to perform SpMV with CSR-Partial-Overlap, which excludes the time spent on data transfer between host and device. Using Speedup as a metric can effectively show the performance improvement effect brought by CSR-Full-Overlap. For example, if Speedup is 2, it means that the time for one SpMV operation using cuSPARSE is enough for CSR-Full-Overlap to perform 2 SpMV operations.

Figure 9 illustrates the Speedup of CSR-Full-Overlap versus cuSPARSE in single precision in the matrix set of Table 1. CSR-Full-Overlap, which unifies the two overlaps in SpMV, outperforms cuSPARSE in every case, with a maximum Speedup of 2.67 and an average Speedup of 2.03. This speedup is quite considerable, as Equation (2) indicates when cuSPARSE has completed 1 SpMV operation, CSR-Full-Overlap has already completed 2.03 SpMV operations. This proves that CSR-Full-Overlap does its best to hide the computation in two major types of data transfers to achieve the acceleration from the two overlaps in SpMV.

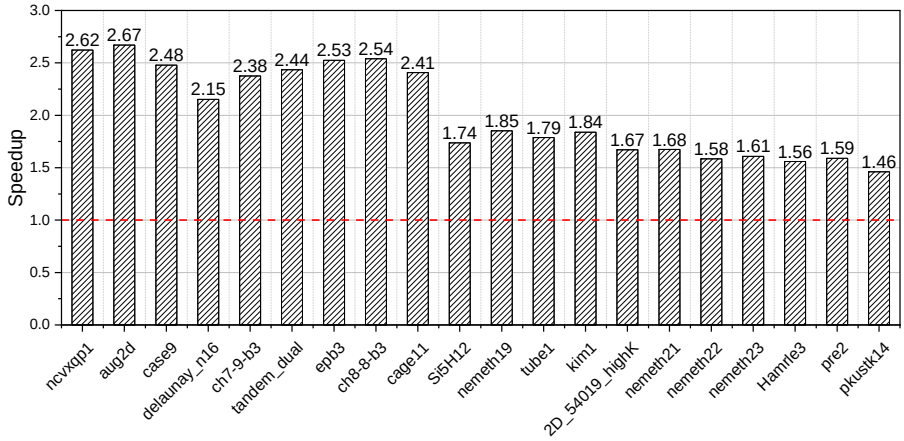

Figure 9. Performance evaluation of CSR-Full-Overlap.

6. Conclusions

As a basic essential scientific computing operation which is widely used in various fields, improving SpMV performance on its dominant CSR format on popular GPU platforms is of great significance and interest to both academia and industry communities.

In this paper, we present observations and in-depth study of effective application of `memcpy_async` from recent NVIDIA Ampere GPUs to achieve substantial performance gain for SpMV. Firstly, we propose the CSR-Partial-Overlap for GSOL-type overlapping. After

that, to remove the overhead of fixing up partial sums and allow low cost and effective load balancing, we design the dynamic batch partition algorithm for the CSR-Partial-Overlap. To further improve the performance of the CSR-Partial-Overlap, we design the dynamic threads distribution algorithm to improve thread utilization. Finally, based on the CSR-Partial-Overlap, we propose the CSR-Full-Overlap, which unifies the two major overlap types and hides the computation as much as possible in the two important access behaviors of the GPU.

We also notice that there are several aspects for future improvement. Particularly, we plan to focus on exploration in the following:

- Note that the load of batch is not completely determined by the number of nonzero elements, but is also related to factors including the number of rows, the dispersion of column coordinates, etc. This suggests that it is promising to explore ways for building a low-cost and efficient batch load prediction model, potentially using machine learning methods, such that a more accurate and better load balancing is achieved;
- In this paper, we only consider single matrix single vector multiplication operation. In practice, the same sparse matrix may reside in the GPU for single matrix multiple vectors multiplication operation, i.e., multiple SpMV operations. Particularly in the observation of the host to device data transfer being 10 times more than that of SpMV computation itself, we plan to explore multiple SpMV operations to bring even greater potential in performance improvement;
- Our approach works under two prerequisites: 1. the GPU supports the GSOL, i.e., the threads can be freed to perform other computational tasks when copying data between shared memory and global memory; and 2. the GPU supports the HDOL, i.e., the GPU can execute other kernels when transferring data between the host memory and the device memory. Therefore, our approach can be applied to any GPUs which support the above two prerequisites. Thus, to improve the generalizability of our approach, applying our approach to GPUs provided by other manufacturers (especially AMD) is one of our future research directions.

It is our belief that this paper not only serves as the first of its kind, but also more importantly, it helps pave the way for more fruitful explorations in this dimension.

Author Contributions: Conceptualization, G.Z.; methodology, G.Z. and Y.Z.; software, G.Z.; validation, G.Z. and Y.Z.; formal analysis, G.Z. and Y.Z.; investigation, G.Z. and Y.Z.; resources, Y.Z.; data curation, G.Z.; writing—original draft preparation, G.Z.; writing—review and editing, G.Z. and Y.Z.; visualization, G.Z.; supervision, Y.Z.; project administration, G.Z.; funding acquisition, Y.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research is supported partly by the SCUT Research Startup Fund No. K3200890, as well as partly by the Guangzhou Huangpu District International Research Collaboration Fund No. 2022GH13. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data that support the findings of this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Asanovic, K.; Bodik, R.; Catanzaro, B.C.; Gebis, J.J.; Husbands, P.; Keutzer, K.; Patterson, D.A.; Plishker, W.L.; Shalf, J.; Williams, S.W.; et al. *The Landscape of Parallel Computing Research: A View from Berkeley*; Technical Report UCB/EECS-2006-183; EECS Department, University of California: Berkeley, CA, USA, 2006.
2. Davis, T.A.; Hu, Y. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* **2011**, *38*, 1–25. [CrossRef]
3. Ahamed, A.K.C.; Magoulès, F. Efficient implementation of Jacobi iterative method for large sparse linear systems on graphic processing units. *J. Supercomput.* **2016**, *73*, 3411–3432. [CrossRef]

4. Gilbert, J.R.; Reinhardt, S.; Shah, V.B. High-performance graph algorithms from parallel sparse matrices. In Proceedings of the Name of the 8th International Workshop (PARA 2006), Umea, Sweden, 18–21 June 2006. [CrossRef]
5. Yang, X.; Parthasarathy, S.; Sadayappan, P. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *arXiv* **2011**, arXiv:1103.2405. [CrossRef]
6. Demirci, G.V.; Ferhatosmanoglu, H. Partitioning Sparse Deep Neural Networks for Scalable Training and Inference. In Proceedings of the ACM International Conference on Supercomputing, Virtual, 14–17 June 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 254–265. [CrossRef]
7. Bell, N.; Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, OR, USA, 14–20 November 2009; pp. 1–11. [CrossRef]
8. Liu, W.; Vinter, B. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In Proceedings of the 29th ACM on International Conference on Supercomputing, Newport Beach, CA, USA, 8–11 June 2015. [CrossRef]
9. Liu, Y.; Schmidt, B. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In Proceedings of the 2015 IEEE 26th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), Toronto, ON, Canada, 27–29 July 2015; pp. 82–89. [CrossRef]
10. Filippone, S.; Cardellini, V.; Barbieri, D.; Fanfarillo, A. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Trans. Math. Softw.* **2017**, *43*, 1–49. [CrossRef]
11. CUDA C++ Programming Guide. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#the-benefits-of-using-gpus (accessed on 15 July 2023).
12. Greathouse, J.L.; Daga, M. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In Proceedings of the SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014; pp. 769–780. [CrossRef]
13. Nisa, I.; Siegel, C.; Rajam, A.S.; Vishnu, A.; Sadayappan, P. Effective Machine Learning Based Format Selection and Performance Modeling for SpMV on GPUs. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, Canada, 21–25 May 2018; pp. 1056–1065. [CrossRef]
14. Choi, J.W.; Singh, A.; Vuduc, R.W. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. *SIGPLAN Not.* **2010**, *45*, 115–126. [CrossRef]
15. Flegar, G.; Anzt, H. Overcoming Load Imbalance for Irregular Sparse Matrices. In Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms, Denver, CO, USA, 12–17 November 2017; ACM: New York, NY, USA, 2017; pp. 1–8. [CrossRef]
16. Ashari, A.; Sedaghati, N.; Eisenlohr, J.; Parthasarathy, S.; Sadayappan, P. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014; IEEE Press: Piscataway, NJ, USA, 2014; pp. 781–792. [CrossRef]
17. Merrill, D.; Garland, M. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In Proceedings of the SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 13–18 November 2016. [CrossRef]
18. Tsai, Y.M.; Cojean, T.; Anzt, H. Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse Linear Algebra Computations. *arXiv* **2020**, arXiv:2008.08478. [CrossRef]
19. Anzt, H.; Tsai, Y.M.; Abdelfattah, A.; Cojean, T.; Dongarra, J. Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse and Batched Computations. In Proceedings of the 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Atlanta, GA, USA, 12 November 2020; pp. 26–38. [CrossRef]
20. Niu, Y.; Lu, Z.; Dong, M.; Jin, Z.; Liu, W.; Tan, G. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Portland, OR, USA, 28 June 2021; pp. 68–78. [CrossRef]
21. Ahmed, M.; Usman, S.; Shah, N.A.; Ashraf, M.U.; Alghamdi, A.M.; Bahadded, A.A.; Almarhabi, K.A. AAQAL: A Machine Learning-Based Tool for Performance Optimization of Parallel SPMV Computations Using Block CSR. *Appl. Sci.* **2022**, *12*, 7073. [CrossRef]
22. Tuning CUDA Applications for NVIDIA Ampere GPU Architecture. Available online: https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#asynchronous-data-copy-from-global-memory-to-shared-memory (accessed on 15 July 2023).
23. cuSPARSE Library. Available online: https://docs.nvidia.com/cuda/cusparse/index.html (accessed on 15 July 2023).