Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices

Mayank Daga Joseph L. Greathouse
AMD Research
Advanced Micro Devices, Inc., USA
{*Mayank.Daga, Joseph.Greathouse*}*@amd.com*

***Abstract*—Sparse matrix vector multiplication (SpMV) is an important linear algebra primitive. Recent research has focused on improving the performance of SpMV on GPUs when using compressed sparse row (CSR), the most frequently used matrix storage format on CPUs. Efficient CSR-based SpMV obviates the need for other GPU-specific storage formats, thereby saving runtime and storage overheads. However, existing CSR-based SpMV algorithms on GPUs perform poorly on irregular sparse matrices, limiting their usefulness.**

We propose a novel approach for SpMV on GPUs which works well for both regular and irregular matrices while keeping the CSR format intact. We start with CSR-Adaptive, which dynamically chooses between two SpMV algorithms depending on the length of each row. We then add a series of performance improvements, such as a more efficient reduction technique. Finally, we add a third algorithm which uses multiple parallel execution units when operating on irregular matrices with very long rows.

Our implementation dynamically assigns the best algorithm to sets of rows in order to ensure that the GPU is efficiently utilized. We effectively double the performance of CSR-Adaptive, which had previously demonstrated better performance than algorithms that use other storage formats. In addition, our implementation is 36% faster than CSR5, the current state of the art for SpMV on GPUs.

I. Introduction

Sparse matrix vector multiplication (SpMV) is a fundamental sparse linear algebra primitive [6]. A common refrain when optimizing this algorithm is that its performance is tightly coupled with the data structure used to store the sparse matrix [21, 22]. Many applications use the compressed sparse row (CSR) matrix storage format because it yields good performance on CPUs and good compression for both structured and unstructured matrices [23].

Substantial research has been conducted to improve the performance of SpMV on graphics processing units (GPUs) [2, 9, 14, 18, 20, 24]. Previous research has proposed more than *fifty* new storage formats for sparse matrices, since CSR seemed unsuitable for GPUs [3, 11]. This is because traditional CSR-based algorithms: (i) did not yield enough parallelism when working on short rows, (ii) were poorly load balanced on irregular matrices that have rows of varying widths, and (iii) relied on performing slow uncoalesced memory accesses [7].

While new storage formats can yield improved performance on GPUs, they present two major concerns. First, software that already uses CSR must be rewritten to utilize the new format. Second, other algorithms also use CSR, necessitating frequent transformations to and from the GPU-optimized formats. These transitions can take large amounts of time and space, reducing the appeal of new formats [11].

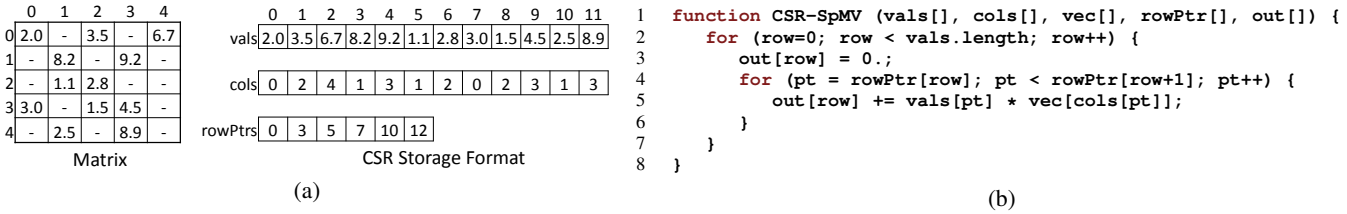
Recent proposals have described efficient CSR-based SpMV algorithms for GPUs [1, 7, 10, 13, 14]. These new algorithms reduce or eliminate the need to modify the original CSR data structure, while, at the same time, they improve the performance over GPU-specific formats. CSR-Adaptive, for instance, is over twice as fast as the eleven SpMV algorithms implemented in the clSpMV auto-tuning framework while requiring substantially lower format-conversion overheads [7].

One limitation of the CSR-Adaptive algorithm is its poor performance on irregular matrices that have a small number of very long rows, such as those found in graph analytics. Calculating the result of these long rows can become the serial bottleneck for the entire computation. An efficient SpMV solution for irregular matrices is thus needed.

We propose a novel approach to compute SpMV on GPUs and integrate it with the existing CSR-Adaptive algorithm. This improved CSR-Adaptive keeps the CSR format intact and works well for both regular and irregular sparse matrices. The new CSR-Adaptive *dynamically* chooses from among three algorithms for every row of the sparse matrix. These three algorithms are optimized for short, long, and very long rows. We optimize the algorithm for short rows, *CSR-Stream*, to incorporate a logarithmic reduction technique to fully utilize the GPU's resources. We also introduce a configuration parameter to choose the optimal number of rows that should be processed by CSR-Stream.

The other algorithm, *CSR-Vector*, can lead to load imbalance with very long rows. We propose a third algorithm that uses multiple parallel execution units on these rows. The three different algorithms create a distinct mapping between the GPU execution units and the rows of the sparse matrix in order to serve the entire spectrum of SpMV.

We evaluate our solution on an AMD FirePro™ W9100 GPU using a set of 32 sparse matrices. Our improvements to CSR-Adaptive double its performance. This allows it to achieve 36% higher performance than CSR5 [13], the current state of the art for SpMV on GPUs, while reducing the overhead of generating data structures by 10×.




```
function CSR-SpMV (vals[], cols[], vec[], rowPtr[], out[]) {
   for (row=0; row < vals.length; row++) {
      out[row] = 0.;
      for (pt = rowPtr[row]; pt < rowPtr[row+1]; pt++) {
         out[row] += vals[pt] * vec[cols[pt]];
      }
   }
}
```

Figure 1. Example of the Compressed Sparse Row (CSR) sparse matrix storage format. (a) The CSR structure for the sparse matrix on the left. (b) Pseudocode of a basic serial algorithm that can be used to perform SpMV on a CSR-based sparse matrix.

We make the following contributions in this paper:

- We propose a new GPU algorithm to compute sparse matrix-vector multiplication for matrices with very long rows that are stored in the CSR format.
- We optimize the existing algorithms in CSR-Adaptive using novel reduction techniques to improve the performance of rows with fewer non-zero values.
- We demonstrate that the improved CSR-Adaptive performs better than all previous GPU-based SpMV algorithms while incurring negligible overheads.

The remainder of this paper is arranged as follows. Section II provides a background on SpMV and previous CSR-based algorithms, while Section III details our algorithm. Section IV explains our experiments, and Section V shows results. We present related work in Section VI, followed by future directions and conclusions in Section VII.

II. Background

A. Sparse Matrix-Vector Multiplication

The goal of SpMV is to find the dense vector product of a matrix stored in a sparse format and a dense vector: $y = Ax$. The data structure used to hold the sparse matrix defines the SpMV algorithm, and the most popular format is compressed sparse row (CSR). This format is built from three data structures: two arrays that hold the non-zero values in the sparse matrix ($vals$) and the column offsets of each non-zero value ($cols$), as well as a smaller array which points into the other two arrays and denotes where each row starts ($rowPtrs$). An example of this is shown in Figure 1a.

CSR is popular because it performs well on CPUs and compresses matrices well regardless of their sparsity pattern. Pseudocode for a CSR-based SpMV is shown in Figure 1b. This can be parallelized by having each thread work independently on a row (an iteration of the outermost loop).

Previous works concluded that CSR was poorly suited for SpMV on GPUs [3]. Having each GPU thread work on a different row, a technique dubbed CSR-Scalar, results in poor performance because of uncoalesced memory accesses. Using a full workgroup to operate on a row, dubbed CSR-Vector, can increase performance by allowing more efficient memory accesses. Unfortunately, CSR-Vector only utilizes the GPU's wide vector units if there are many non-zero values in each row. Parallelizing over a row with fewer values than the vector width of the processor leaves execution units idle, again hampering performance.

These difficulties led Bell and Garland to posit that CSR was an unsuitable storage format for GPU-based SpMV. They proposed a hybrid of ELLPACK and the COO format, a decision that carried forward to the CUSPARSE library [16]. This led to a deluge of research into new formats, ranging from sliced and blocked versions of ELLPACK [4, 15] to complex coordinate schemes [24].

Conversion to these new formats must often take place dynamically, since other algorithms, like sparse matrix-matrix multiplication, are built to use CSR [12]. As such, Langr and Tvrdík argue that conversion times are a vital metric for new formats; it is not always possible to amortize these times through successive SpMV iterations [11]. They also recommend that memory overheads be measured for new formats and mention that *in-place* formats, those that do not require extra storage during conversion, are ideal.

B. Previous CSR-based GPU Algorithms

There has been a renewed interest in CSR-based SpMV on GPUs. Reguly and Giles described a CSR-based algorithm that is a mix of CSR-Scalar and CSR-Vector [17]. They use multiple threads per row but avoid using entire workgroups if the average row lengths are short. They pick a static number of threads per row based on the average number of non-zero values (NNZs) in each row. This static, matrix-wide choice makes it difficult to achieve maximum performance on irregular matrices with varying row lengths. This algorithm is similar to the CSR-Vector implementation available in the CUSP library [19].

Koza et al. describe an extension to CSR called compressed multi-row storage (CMRS) that helps solve this problem [10]. They allocate a static number of rows, called a *strip*, to each workgroup. Within a strip, they stream the adjacent values into the GPU, allowing better coalescing and load balancing. The static size of each strip can still lead to performance loss for irregular matrices, however.

CSR-Adaptive alleviates this difficulty by statically fixing the *NNZs* per workgroup and dynamically calculating the number of rows each workgroup will handle [7]. Multiple short rows can thus be assigned to a single workgroup, while long rows can be given to individual workgroups to help load balance. The CSR-Adaptive algorithm leaves the CSR structure unchanged but adds a *rowBlocks* buffer to delineate which rows are assigned to each workgroup. CSR-Adaptive is faster, on average, than other matrix storage formats while maintaining comparability with CSR.

Liu et al. showed that CSR-Adaptive's performance was limited on irregular matrices that contained very long rows [13] and therefore developed CSR5. Their algorithm adds extra data structures to CSR and performs an in-place transpose of parts of the matrix to maximize performance on both regular and irregular matrices. Their results showed higher performance than any previous CSR-based SpMV. The limitations of CSR5 are its complexity and the matrix transpose, which can cause large transformation overheads.

III. Implementation Details

The objective of our work is to obtain good SpMV performance regardless of the structure of the input matrices. This section describes our complete SpMV solution, which dynamically switches between three algorithms - CSR-Stream, CSR-Vector, and CSR-VectorL. We first describe optimizations for CSR-Stream and CSR-Vector, followed by a detailed description of our new CSR-VectorL algorithm.

We analyzed a non-public implementation of CSR-Adaptive in a previous paper [7], which also described it in pseudocode. Soon after that work was published, the authors of ViennaCL used this pseudocode to implement CSR-Adaptive in version 1.6.1 of their library [18]. This forms our *baseline* implementation in this paper.

Pseudocode for this baseline is shown in Figure 2. The number of rows assigned to a workgroup is used to determine the best algorithm for that workgroup. CSR-Stream is used if the number of rows is greater than one, otherwise CSR-Vector is used. CSR-Stream first performs coalesced loads of `NNZ_PER_WG` values into the GPU's local memory (LDS) and then uses a scalar reduction technique to compute the final result for every row of that workgroup. CSR-Vector loads a row's values from global memory to local registers in parallel. Afterwards, a parallel reduction is performed through the LDS to calculate a row's final result [3].

A. Optimized CSR-Stream

The baseline CSR-Stream has two main limitations – (1) it may not yield the best performance for workgroups with few rows, and (2) the scalar reduction technique employed to compute the final result may leave a large number of GPU threads idle, thereby inefficiently utilizing hardware resources. This section describes how we optimized CSR-Stream to substantially improve its performance.

1) Logarithmic Reduction: CSR-Stream computes the output for each row by reducing the multiplication results held in the LDS. The baseline implementation of CSR-Stream employs a scalar reduction technique to compute this final result, as shown in Figure 2. This reduction uses only one thread per row (`line 16`).

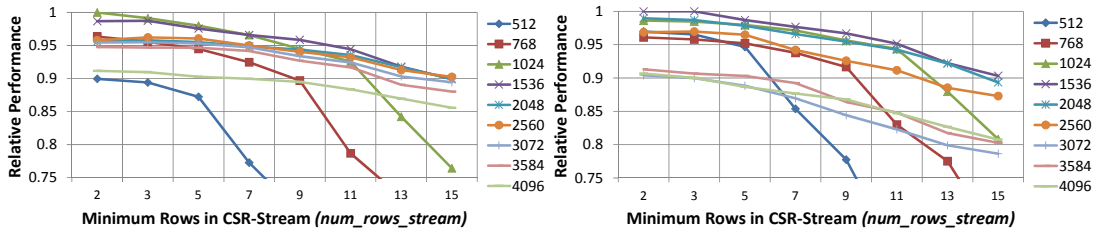
We augment this scalar reduction with a logarithmic reduction that uses multiple threads to reduce each row. Pseudocode for this two-step reduction process is shown in Figure 3. First, threads are evenly distributed to each row of

```
1  function CSR-Adaptive-Baseline (vals[], cols[], vec[],
2                    rowPtrs[], rowBlocks[], out[]) {
3     startRow = rowBlocks[workgroupID];
4     stopRow = rowBlocks[workgroupID+1];
5     numRows = stopRow - startRow;
6     local TYPE LDS[NNZ_PER_WG];
7     // Choose between CSR-Stream and CSR-Vector
8     if (numRows > 1) // CSR-Stream case
9        for i ∈ BLOCKSIZE
10          localCol = rowPtrs[startRow]+localthreadID+i;
11          LDS[localthreadID+i] = vals[localCol];
12          LDS[localthreadID+i] *= vec[cols[localCol]];
13          i += WGSIZE;
14       end for
15       firstCol = rowPtrs[startRow];
16       localRow = startRow + localthreadID;
17       // # numRows threads perform Scalar Reduction out
18       // of LDS to compute final output
19       while(localRow < stopRow) do
20          temp = 0;
21          i = rowPtrs[localRow] - firstCol;
22          // Loop over non_zeroes for this row
23          for i ∈ (rowPtrs[localRow+1] - firstCol) do
24             temp += LDS[i];
25          end for
26          // Write computed result to the output buffer
27          out[localRow] = temp;
28          localRow += workgroupSize;
29       end while
30    else  // CSR-Vector case
31       /* OMITTED: CSR-Vector is well known */
32    end if
33 }
```

Figure 2. Pseudocode for our Baseline CSR-Adaptive. CSR-Adaptive uses the `rowBlocks` buffer to determine how many rows are being computed by each workgroup. CSR-Stream works on at least two rows and the scalar reduction is carried out using one thread per row.

```
1  function Logarithmic-CSR-Stream-Reduction {
2     // # threads that can be allocated to each row
3     numThreadsRed = numThreadsInWG / numRows;
4     numThreadsRed = lowerPowerOf2(numThreadsRed);
5     // # values which each thread needs to reduce
6     threadInRow = localthreadID & (numThreadsRed - 1);
7     /* Step 1 -- Compact all rows into WGSIZE entries*/
8     /* OMITTED: stride and offset computations are
9      * abstracted for clarity. */
10    localRow = startRow + (localthreadID/numThreadsRed);
11    workPerThread = nnzInLocalRow / numThreadsRed;
12    for i ∈ workPerThread do
13       /* OMITTED: a parallel group of numThreadsRed
14        * threads sums their entire localRow into 'temp'
15        * registers with a stride of numThreadRed. */
16    end for
17    LDS[localthreadID] = temp;
18    /* Step 2 -- Parallel logarithmic reduction */
19    i = WGSIZE / 2;
20    while (i > 0)
21       if (i < numThreadsRed)
22          LDS[localthreadID] += LDS[localthreadID + i];
23       end if
24       i = i / 2;
25    end while
26    if (threadInRow == 0 && localRow < stopRow)
27       out[localRow] = LDS[localthreadID];
28    end if
29 }
```

Figure 3. Pseudocode for logarithmic reduction in CSR-Stream. Step 1 uses `numThreadsRed` adjacent threads to reduce a row into `numThreadsRed` LDS locations. Step 2 then does a logarithmic reduction of each of these values, such that each row's output is contained in the LDS entry corresponding to the first thread in the group working on that row.



(a) Single Precision (b) Double Precision

Figure 4. Relative performance of CSR-Adaptive at varying minimum rows in CSR-Stream for a range of NNZ_PER_WG. The legend depicts the value of NNZ_PER_WG. The values plotted are the harmonic mean of the performance achieved with over 48 different sparse matrices.

the workgroup. The threads assigned to a row then reduce all of its results into fewer locations in the LDS. Next, these values are reduced in parallel since the initial step aligns the values for each thread. This results in multiple outputs that can then be stored to global memory in parallel.

For explanatory purposes, let us assume that there are 16 rows assigned to a 256-thread workgroup, each with 1024 non-zero values. This new reduction will allocate $256/16 = 16$ threads to each row. Each of these 16 threads evenly divide the non-zero values for that row, meaning that each thread would reduce $1024/16 = 64$ non-zero values into a single LDS entry. All of the threads access their 64 non-zero values and cooperatively reduce them into 16 LDS locations. This marks the completion of Step 1. In Step 2, all 256 threads perform parallel reductions until there are 16 output values remaining, one for each row.

The modified reduction technique overcomes the limitation of scalar reduction and improves the performance for workgroups which calculate fairly small numbers of rows with many non-zero values per row. The logarithmic reduction is efficient only when more than one thread is allocated to every row. If this is not the case, scalar reduction is more efficient. We therefore check the number of threads that are available for reduction and choose between logarithmic or scalar reductions, as shown in Figure 9.

We found that this reduction technique fared somewhat poorly when the length of rows in a workgroup varied greatly. As such, it is best to modify the `rowBlocks` generation algorithm to break a row-block when moving between relatively long (e.g. > 128 NNZ) and relatively short (e.g. < 32 NNZ) rows. We implemented this mechanism but do not further describe or study it in this paper.

2) Hyper-Parameter: `num_rows_stream`: The baseline CSR-Adaptive uses CSR-Stream when more than one row is processed by a workgroup. This may not yield the best performance, because the overheads associated with reduction may not be amortized. CSR-Vector can be better at processing workgroups with a small number of rows due to its simpler reduction method.

Determining the minimum number of rows to send to CSR-Stream in order to achieve the best overall SpMV performance depends on the characteristics of individual sparse matrices. The optimum value can be empirically deduced by evaluating the performance at varying values of minimum rows and `NNZ_PER_WG` across a variety of sparse matrices. We introduce a hyper-parameter called *num_rows_stream* to choose the optimal number of rows for CSR-Stream.

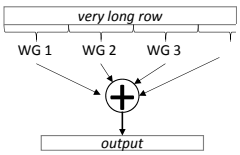
The improved CSR-Adaptive uses CSR-Stream when the number of rows is greater than or equal to `num_rows_stream` and uses CSR-Vector otherwise. Figure 4 illustrates the relative performance of CSR-Adaptive at varying values of both `num_rows_stream` and `NNZ_PER_WG` using the harmonic mean for 48 different matrices. From the figure, we show that for single precision, this value is best set to the default of 2. Double precision, on the other hand, sees a minor benefit at a value of 3. On our particular GPU, we see little average performance difference, but we note that this is not always the case on different GPUs or on all matrices. Another important point is that the best performing `NNZ_PER_WG` for single and double precision is 1024 and 1536, respectively.

The optimal values of `num_rows_stream` and `NNZ_PER_WG` can change with the GPU micro-architecture and, and experiments should be rerun to find the best value for any particular GPU.

B. SpMV for Very Long Rows (CSR-VectorL)

CSR-Adaptive used the CSR-Vector algorithm (which uses one workgroup per row) for long rows with many non-zero values. However, if a row is exceptionally long, then the corresponding workgroup might take longer than any other row, causing a performance bottleneck.

1) Multiple Workgroups Per Row: A higher-performing solution is to use multiple workgroups to compute SpMV on a long row, as shown in Figure 5a. Each workgroup working on a long row must therefore know which subset of that row it should to work on. For instance, workgroup 1 will load the first `NNZ_PER_WG` values, while workgroup 2 will load the next `NNZ_PER_WG` values, etc. If multiple entries in the `rowBlocks` buffer point to the same row, then those workgroups can cooperate on that row. However, the existing CSR-Adaptive structure do not have enough information to accurately complete this operation. All of the workgroups processing the same row would start at the first non-zero in that row and end at its last non-zero.



(a) Multiple Workgroups on a Single Row

row_delimiters	0	2048	5,120	5,632
row block buffer	0	1	2	3

(b) Existing rowBlocks Buffer

0	0	0	1	0	2	0	3	1	0	1	1	1	2	1	3	1	4	1	5	2	0	3	0

(c) Improved rowBlocks Buffer

Figure 5. Illustration of multiple workgroups working on a single long row. Three rows contain 2048, 3072, and 512 non-zeroes, respectively. The NNZ_PER_WG is assumed to be 512 for both existing and improved rowBlocks buffers. In the improved rowBlocks buffer, each entry consists of row and workgroup numbers for workgroups working on that row.

```
|6666 5555 5555 5544 4444 4444 3333 3333|3322 2222|2222 1111 1111 1100 0000 0000|
|3210 9876 5432 1098 7654 3210 9876 5432|1098 7654|3210 9876 5432 1098 7654 3210|
|------------Row Information------------|----flag^|---WG ID within a long row---|
```

Figure 6. Single entry of the improved rowBlocks buffer demonstrating the bits to store the row number, the flag, and the workgroup number. The exact number of bits used to store each can be implementation dependent. The flag allows workgroups to synchronize.

For illustration purposes, let us assume that three rows in a sparse matrix consist of $2048, 3072$, and 512 non-zeros. The `rowBlocks` buffer for such an example is shown in Figure 5b. The baseline CSR-Adaptive would use one workgroup for each row, leading to load imbalance, since the first two workgroups must do more work than the third.

To solve this problem, we assign multiple workgroups to rows such as these. We then extend the `rowBlocks` structure so that each workgroup assigned to a long row can calculate where in that row it should work. This can be accomplished by using one set of bits of a 64-bit integer to store the row number (like the original `rowBlocks` structure) and another set of bits to store the workgroup number for that particular row, as shown in Figure 5c.

In this example, each solid black box denotes a 64-bit integer. The blue line shows a demarcation of the set of bits used to store the row and workgroup numbers. Workgroup 0 works on the first 512 non-zeroes of row 0, workgroup 2 works on the second 512 non-zeroes of row 0, and so on. Workgroup 5 works on first 512 elements of row 1, and workgroup 11 works on row 2. The number of bits chosen to store the row number and workgroup number can be implementation dependent, and these values could be stored in different structures for simplicity. Using the improved `rowBlocks` buffer, each workgroup can determine the start and end of its working region in a long row.

2) Reducing Partial Outputs: The second part of the long rows algorithm is to combine the answers from each of the workgroups which cooperate on a long row to produce the final output (the reduce operator in Figure 5a). Since multiple workgroups are working towards the same final output, a regular addition would result in a data race.

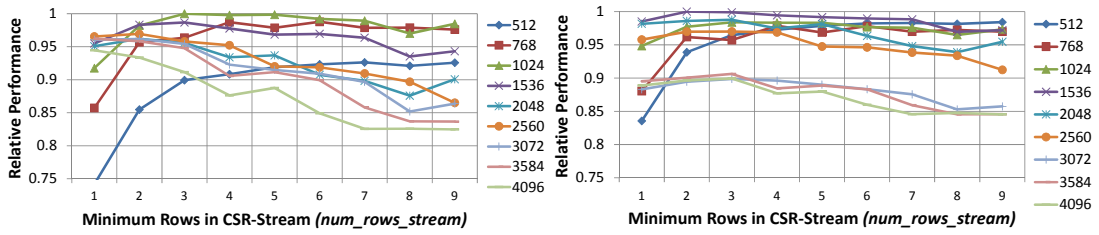
```
1  function Multi-Workgroup-Barrier {
2     compare_value = rowBlocks[WG_ID].flag;
3     if(WG_ID == first_wg_in_row)
4        // First WG handles output initialization
5        out[row] = 0;
6        // Release other workgroups
7        xor(rowBlocks[first_wg_in_row].flag, 1);
8     end if
9     // For other workgroups, compare_value holds
10    // what to wait on. If your flag==first workgroup's
11    // flag, you spin loop
12    while(WG_ID != first_wg_in_row &&
13       rowBlocks[first_wg_in_row].flag == compare_value);
14    // After the barrier, update your flag to ensure
15    // you know what to wait on the next time through
16    if(WG_ID != first_wg_in_row)
17       xor(rowBlocks[WG_ID].flag, 1);
18    end if
19 }
```

Figure 7. Pseudocode for implementation of a GPU barrier which allows all of the workgroups that are cooperative on a single row to wait for the output to be properly initialized.

We use atomic additions to enforce the correct final output. However, simply using atomic additions will produce incorrect answers if the output buffer is not initialized before every iteration. Similarly, if the goal of the SpMV algorithm is to calculate $y = \alpha Ax + \beta y$, rather than simply $y = Ax$ (where α and β are scalar multipliers), then it is difficult to initially multiply y by β. In this case, the output buffer must first be initialized before allowing any other workgroups to write out their partial sums. We do this by means of a flag-based barrier. Generally, a workgroup can know if it is taking part in the operation on a "long row" by checking the values which occur before and after its own corresponding value in the `rowBlocks` buffer. If either points to the same row, then we must perform this specialized initialization.

The first workgroup in a cooperative performs the initialization while other workgroups wait at a barrier to ensure that they do not read the uninitialized value. After the first workgroup completes the initialization, it releases all other workgroups from the barrier. We have each of these workgroup wait on a flag stored in the `rowBlocks` entry of that row's first workgroup, as shown in Figure 6. All of the workgroups spin-loop while the value of the first workgroup's flag is 0. Meanwhile, the first workgroup sets its flag to 1 once it completes the initialization.

Another complexity arises when the SpMV kernel is called multiple times. In this case, the cooperative workgroups will not wait at the barrier because the previous flag still remains. Because every workgroup has its own `rowBlocks` entry and its own flag bit, a solution is to have every workgroup independently store (in its own flag) the value it should wait on. After going through the barrier, it flips its own flag. As such, each workgroup checks its flag to know whether value means "wait at this barrier" or "go ahead". The pseudocode to implement multi-workgroup barrier in a GPU kernel is show in Figure 7.



(a) Single Precision (b) Double Precision

Figure 8. Relative performance of CSR-Adaptive at varying NNZ_multipliers for a range of NNZ_PER_WG. The legend depicts the value of NNZ_PER_WG. The values plotted are the harmonic mean of the performance achieved with over 48 different sparse matrices.

3) Hyper-Parameter: `NNZ_multiplier`: Computing SpMV for a long row by merely dividing it across multiple workgroups and having each workgroup process `NNZ_PER_WG` non-zero values is not optimal. This is because (1) the total number of workgroups to be launched can be high, which can cause substantial workgroup scheduling overheads, and (2) the number of atomic writes to global memory linearly increase with the increase in number of workgroups processing a row. The number of atomic writes increase because every workgroup needs to atomically update its partial value to compute the final output.

One way to reduce these overheads is to increase the number of non-zero values processed by every workgroup. However, we have previously found the values for `NNZ_PER_WG` that achieve best performance for CSR-Stream and CSR-Vector, as shown in Section III-A. Therefore, we should alter the `NNZ_PER_WG` for workgroups working only on CSR-VectorL. This is achieved by introducing another hyper-parameter, called *NNZ_multiplier*, which increases the `NNZ_PER_WG` by a predefined value. Choosing the appropriate value of `NNZ_multiplier` is challenging because a high value may adversely affect performance, similar to why CSR-Vector performs poorly on long rows. In contrast, a low value might still incur high scheduling and atomic overheads. We empirically determine the best value of `NNZ_multiplier` by comparing the performance of 48 sparse matrices at varying values of `NNZ_PER_WG` and `NNZ_multiplier`. These results, for both single and double precision, are shown in Figure 8.

From the figure, it is noted that an `NNZ_multiplier` of 2 provide the best performance for double precision at the `NNZ_PER_WG` of 1536. For single precision, a `NNZ_multiplier` of 3 at a `NNZ_PER_WG` of 1024 performs the best. Larger values of both `NNZ_PER_WG` and `NNZ_multiplier` reduce performance as the algorithm tends to be more like CSR-Vector. Contrarily, smaller values of `NNZ_multiplier` do not successfully amortize the overheads associated with CSR-VectorL. These optimal values may change for different GPUs. It is worth reiterating that the `NNZ_multiplier` is used for only those workgroups which use CSR-VectorL.

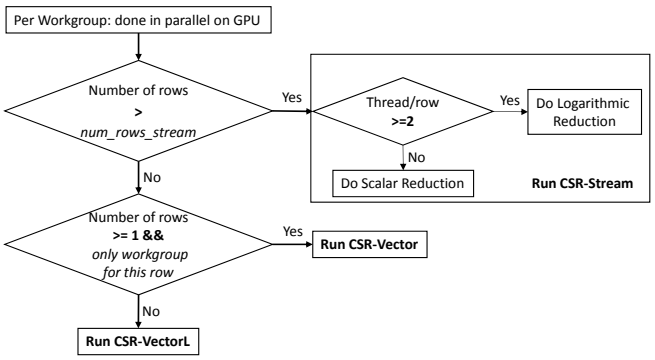


Figure 9. Flowchart for improved CSR-Adaptive.

C. Bringing Everything Together

We have described three different SpMV algorithms, CSR-Stream, CSR-Vector, and CSR-VectorL, which are advantageous for short, long, and very long rows, respectively. We combine the three algorithms into an improved CSR-Adaptive to efficiently compute SpMV regardless of the structure of sparse matrices. The pseudocode for the improved CSR-Adaptive is shown in Figure 9.

We should also mention that other low-level optimizations are possible. For example, splitting row blocks when the row lengths change (as mentioned in Section III-A1) can help performance. Removing the division operations to compute the number of available threads per row in Figure 3 and replacing them with appropriate shifts or calculating these values on the CPU also helps. A version of our source code with all such optimizations is available as part of the open source clSPARSE library, which can be found at https://github.com/clMathLibraries/clSPARSE/.

IV. Experimental Setup

We implemented the improved CSR-Adaptive algorithm starting from the baseline described in Figure 2. Our CSR-Adaptive hyper-parameters are shown in Table I. We compare the improved CSR-Adaptive to CSR5, the current state of the art for SpMV on GPUs. The implementation of CSR5 was provided by its authors.

We test our improved CSR-Adaptive on 32 different matrices, shown in Table II, from previous works [3, 13, 20].

We performed all of our experiments on an AMD FirePro™ W9100 GPU with ECC disabled. Table III details its important characteristics. The host machine, which we also use to measure matrix format conversion times, uses an AMD A10-7850K APU with 32 GB of DDR3-2133 SDRAM. The GPU was programmed using OpenCL™ 1.2 with the AMD APP SDK v2.9 and AMD FirePro driver v14.20 Beta on Ubuntu 14.04.2. We used 256 threads per workgroup, and all of the performance numbers are an average of 1000 runs.

Table I
VALUES OF HYPER-PARAMETERS USED IN CSR-ADAPTIVE

Hyper-Parameter	Single Precision	Double Precision
NNZ_PER_WG	1024	1536
num_rows_stream	1	2
NNZ_multiplier	3	2

Table II
OVERVIEW OF SPARSE MATRICES USED FOR EVALUATION

Name	Size	NNZ	NNZ/row	Max
Dense2	2K * 2K	4,000,000	2,000	2,000
Protein	36K * 36K	4,344,765	119	204
FEM/Spheres	83K * 83K	6,010,480	72	81
FEM/Cantilever	62K * 62K	4,007,383	64	78
Wind Tunnel	218K * 218K	11,634,424	53	180
FEM/Harbor	47K * 47K	2,374,001	51	145
QCD[1]	49K * 49K	1,916,928	39	40
FEM/Ship	141K * 141K	7,813,404	55	102
Economics	207K * 207K	1,273,389	6	44
Epidemiology	526K * 526K	2,100,225	4	4
FEM/Accelerator	121K * 121K	2,624,331	22	81
Circuit	171K * 171K	958,936	6	353
Webbase	1,000K * 1,000K	3,105,536	3	4700
LP	4K * 1,097K	11,284,032	2,634	56,182
circuit5M	5,558K * 5,558K	59,524,291	11	1,290,501
eu-2005	863K * 863K	19,235,140	22	4240
Ga41As41H72	268K * 268K	18,488,476	69	702
in-2004	1,383K * 1,383K	16,917,053	12	7753
mip1	66K * 66K	10,352,819	156	66,396
Si41Ge41H72	186K * 186K	15,011,265	81	662
ASIC_680k	683K * 683K	3,871,773	6	395,259
dc2	117K * 117K	766,396	7	114,190
FullChip	2,897K * 2,897K	26,621,990	9	2,312,481
ins2	309K * 309K	2,751,484	9	309,412
bone010	986K * 986K	47,851,783	48	81
crankseg_2	121K * 121K	2,624,331	22	3423
ldoor	952K * 952K	42,493,817	45	77
rajat31	4,690K * 4,690K	20,316,253	4	1252
Rucci1	1,978K * 109K	7,791,168	4	5
boyd2	466K * 466K	1,500,397	3	93,262
sls	1,748K * 63K	6,804,304	4	5
transient	179K * 179K	961,790	5	60,423

Table III
OVERVIEW OF AN AMD FIREPRO™ W9100 GPU

Compute Units (CU)	44
Core Clock Rate	930 MHz
GDDR5 Memory Clock Rate	1250 MHz
Peak Memory Bandwidth	320 GB/s
L2 Cache Size	1024 KB
L1 Cache Size per CU	16 KB
Local Data Store (LDS) Size per CU	64 KB

[1]This matrix stores complex numbers, but only the real portions were used in our tests. CSR5 and ViennaCL do the same.

V. EVALUATION

A. Effect of Optimizations

Figure 10 presents the effect of various optimizations in both double and single precision. The performance achieved with our simple implementation of CSR-Adaptive is denoted by `Baseline`. We note that the double precision analysis in Figure 10b uses a row block size of 1024. As shown in Figure 8b, the final performance is roughly the same as with a size of 1536. The performance of `Baseline` was much worse at 1536, however, so we felt that it was unfair to compare our optimizations against it. All other double precision results use a row block size of 1536.

The first optimization, denoted by `NRS+opt` or `num_rows_stream`, divides the SpMV computation between CSR-Stream and CSR-Vector depending on the number of rows assigned to a workgroup. This also includes the various optimizations we mentioned at the end of Section III-C. `NRS+opt` improves the performance by an average of 1% over `Baseline` for single and double precision. The speedups for `mip1` are 3.2× and 4.5×, since many of its workgroups compute only one or two rows each, and they are more amenable to CSR-Vector.

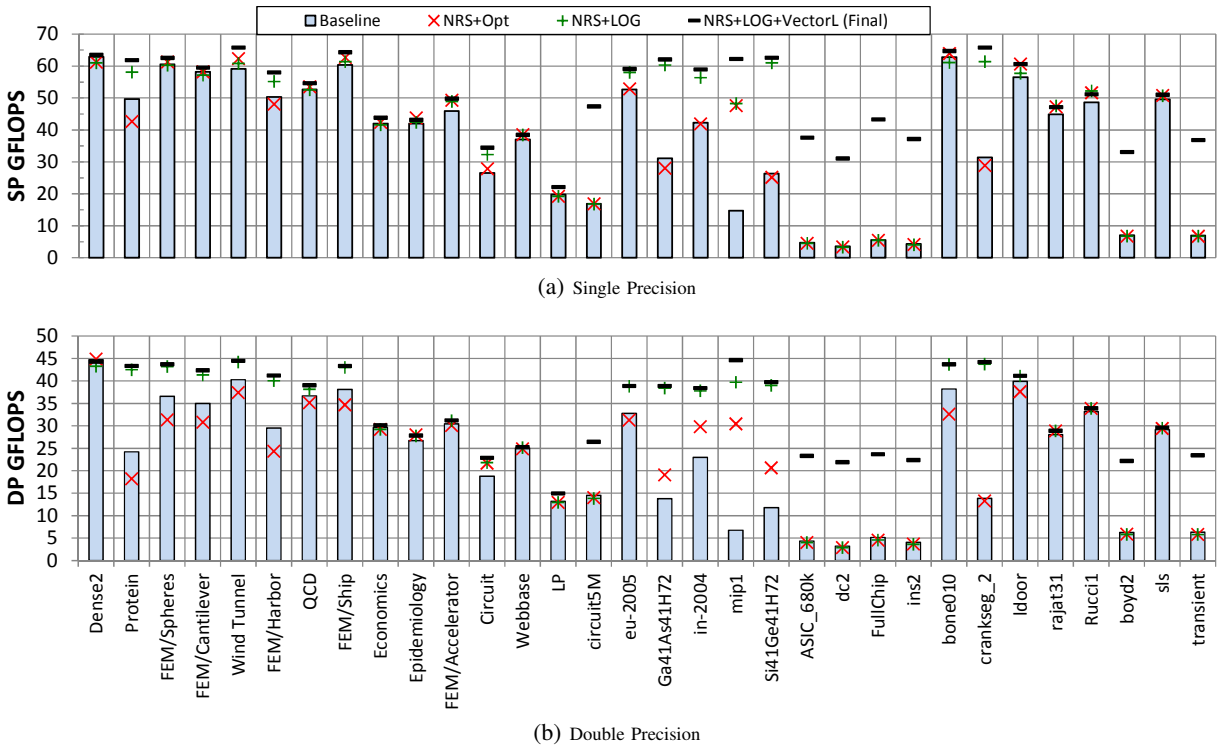
We then add the logarithmic reduction technique (`LOG`) in CSR-Stream. Adding `LOG` improves the performance by 4% and 9% over `NRS+opt` in single and double precision, respectively. This combination of `NRS+LOG` more doubles the performance for three matrices (`Ga41As41H72`, `Si41Ge41H72`, `crankseg_2`). These matrices leave many threads idle when performing a scalar-style reduction, which can become a performance bottleneck.

The performance drops for some matrices when adding `NRS+opt`. However, when combined with `LOG`, there are no performance losses. This shows that combinations of optimizations are vital to maximize performance.

Lastly, we integrate the CSR-VectorL algorithm into CSR-Adaptive to improve the performance of matrices which contain a few very long rows. Several such matrices (`circuit5m`, `ASIC_680k`, `dc2`, `FullChip`, and `ins2`) achieve a speedup of up to 8.2× in single precision and up to 6.6× in double precision. Integrating CSR-VectorL doubles the average performance for all matrices over the combination of only `NRS+LOG`.

Overall, the improved CSR-Adaptive achieves an average speedup of 2.8× (SP) and 2.4× (DP) over the `Baseline`.

While the algorithms presented here are optimized to access the matrix as quickly as possible, uncoalesced accesses to the vector can prevent some matrices from reaching the same performance as others. This can be seen in matrices that are greatly helped by CSR-VectorL. As shown in Table II, these matrices have some number of very long rows. The parallel accesses to the dense vector are therefore spread over a wide range of addresses, resulting in uncoalesced, slow loads. This problem also manifests in `LP`.



(a) Single Precision

(b) Double Precision

Figure 10. Effect of various optimizations in CSR-Adaptive. NRS: num_rows_stream, denotes the performance achieved by optimally choosing the minimum number of rows needed in a row block before using the CSR-Stream algorithm. LOG: logarithmic reductions in CSR-Stream. NRS+LOG denotes the performance achieved by combining NRS and LOG. Lastly, the optimized version combines NRS, LOG, and CSR-VectorL.

Along the same lines, some tests can cache the input vector better than others. The vector access pattern depends on the input matrix and can affect cache hit rates. The matrix `sls`, for instance, is heavily affected by changes in the cache size. Those matrices that do not clearly fit into our GPU's 1MB L2 cache will generally perform slightly worse due to higher memory bandwidth requirements.

B. Comparison to CSR5

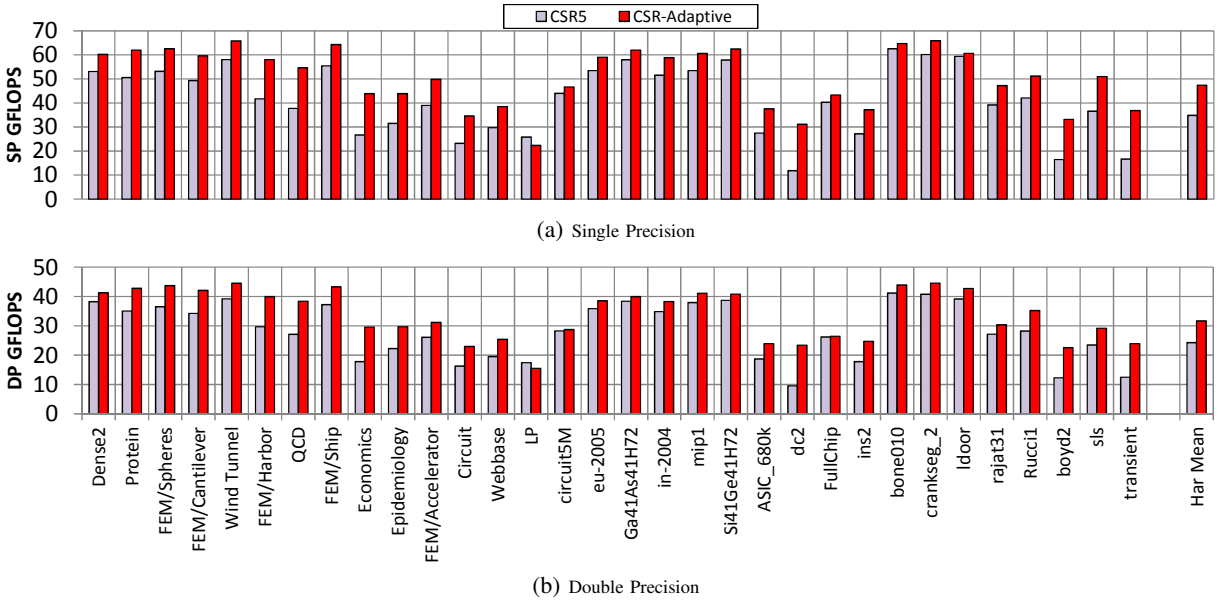
Our previous work on CSR-Adaptive showed that our baseline algorithm was better than a plethora of other formats (such as ELLPACKR, BCSR, SELLPACK, and others in the clSpMV suite). Similarly, the authors of CSR5 showed that their algorithm was better than any other CSR-based solution, including the baseline version of CSR-Adaptive implemented in ViennaCL. As such, we compare the improved CSR-Adaptive against this state-of-the-art SpMV implementation.

Figure 11 demonstrates the performance achieved by CSR5 and CSR-Adaptive. On average, CSR-Adaptive is faster than CSR5 by 36% in single precision and by 30% in double precision. However, for individual matrices, CSR-Adaptive improves the performance by 2.6× like for `dc2` in single precision and 2.4× for double precision. The only case when CSR-Adaptive is slower is for `LP`, by 16%. In both algorithms, the performance of `LP` is limited by the access pattern to the input vector. CSR-Adaptive is slightly more aggressive at accessing the vector and causes more memory contention.

Both CSR5 and CSR-Adaptive use specialized data structures (e.g. the `rowBlocks` buffer in CSR-Adaptive) to improve the performance of CSR-based SpMV on GPUs. The overhead of generating these structures may reduce the usefulness of the SpMV [11]. We therefore compare the single-precision data structure generation times of CSR-Adaptive and CSR5. The CSR-Adaptive row blocking algorithm is single-threaded CPU code, while the CSR5 generation algorithm parallelizes some parts of the generation on the GPU. Figure 12 illustrates the overhead incurred by CSR5 normalized to the overhead incurred by CSR-Adaptive.

From the figure, CSR5 can take up to 105× longer to generate its required data structures compared to CSR-Adaptive. An interesting data point is `LP`. CSR5 is 16% faster to calculate a single SpMV iteration, as shown in Figure 11. However, CSR5 incurs a setup overhead 65× greater than CSR-Adaptive. Therefore, CSR5 would take 31 SpMV iterations after generating the data structures before it gained performance over CSR-Adaptive. None of these matrices incur more overhead with CSR-Adaptive than with CSR5. On an average, CSR-Adaptive incurs 10× less overhead than CSR5.

To summarize, CSR-Adaptive is 36% faster than CSR5, while incurring 10× less data structure generation overhead.



(a) Single Precision

(b) Double Precision

Figure 11. Performance comparison between CSR-Adaptive and CSR5.

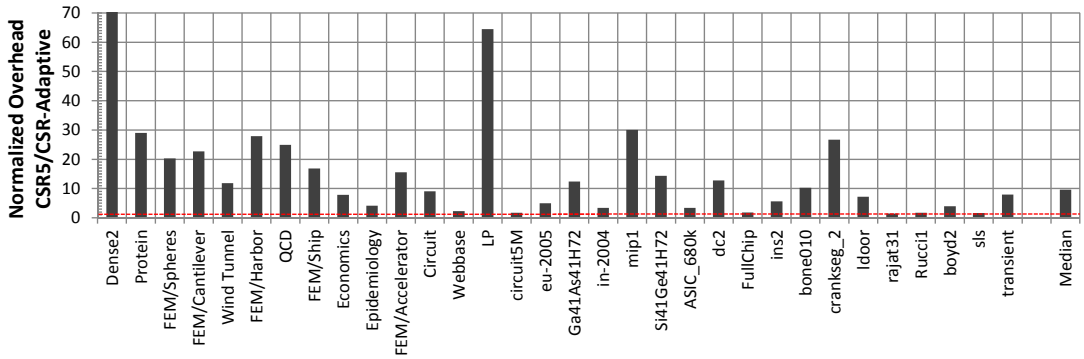


Figure 12. Overhead incurred for generation of specialized data structures in CSR5 and CSR-Adaptive, normalized to CSR-Adaptive (red line)

C. Overhead for Storage Format

As previously mentioned, CSR-Adaptive improves the performance of SpMV on GPUs by augmenting the CSR data structures with the `rowBlocks` buffer. Our experiments for single precision showed that `rowBlocks` increases the storage requirement by an average of only 0.0603% (and a maximum of 0.0716%) compared to the storage requirements of the regular CSR data structures. For double precision (with its different block size), these numbers become 0.0611% and 0.0823%, respectively.

Several applications are known to iteratively call the SpMV routine. Hence, rather than just comparing the rowBlock generation time with CSR5, we also compare the generation time versus the performance of a single SpMV iteration. This gives a general idea of the number of SpMV iterations required to amortize the cost of generating the `rowBlocks` structure. This data is presented in Figure 13.

From the figure, the average matrix takes less than 2.5 SpMV iterations to overcome the overhead of `rowBlocks` generation. This is substantially better than the overheads incurred by either other CSR-based SpMV algorithms or more complicated formats, thereby making CSR-Adaptive an extremely lightweight SpMV solution.

D. Memory Bandwidth Performance Bounds

Rather than simply analyzing the computational performance of SpMV, evaluating how close it is to the maximum bandwidth of a processor is also relevant. This is a measure of optimality of the algorithm for a particular device [11].

Gropp et al. showed a formula for calculating the bandwidth bounds for SpMV [8]. They assume that every value from both the matrix and the input vector must be loaded into the cache exactly once, and that each entry of the output vector must be written to once. Koza et al. call this value β_+ [10]. This is a lower bound on the memory throughput because it makes the idealistic assumption that the input vector is perfectly cached. Dividing this estimated data transfer by the SpMV execution time gives a memory throughput estimate that, ideally, should be as close as possible to the GPU's maximum memory bandwidth.

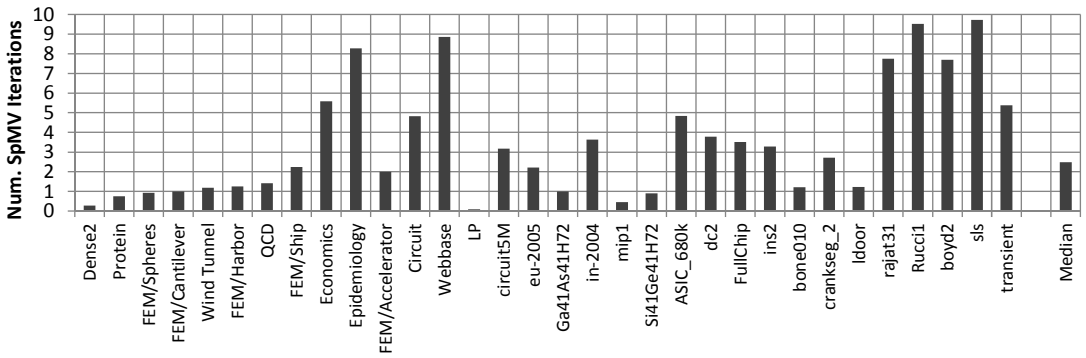


Figure 13. Number of SpMV iterations required to amortize the cost of generation of the rowBlocks buffer using a single CPU core.

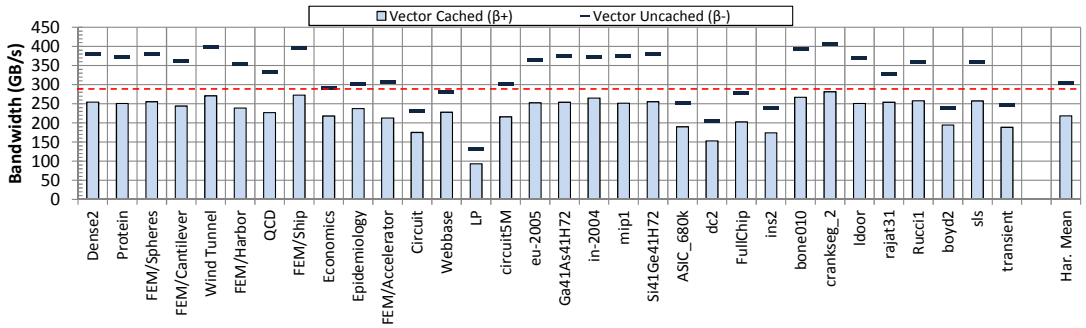


Figure 14. Bandwidth achieved by CSR-Adaptive assuming the vector is either fully cached or uncached. The red dashed line indicates the bandwidth achieved using a STREAM-style microbenchmark. Results are from single precision tests; double precision bandwidth is roughly the same.

Figure 14 shows these calculations for our optimized CSR-Adaptive kernel. The bars in this chart show the lower bound on memory bandwidth (calculated using the lower bound on memory throughput, β_+). The red line shows the GPU's achievable memory bandwidth in the DeviceMemory microbenchmark from the SHOC benchmark suite [5]. In this case, many of the input matrices operate within 15% of this GPU's maximum achievable performance.

The upper bar for each input matrix shows the estimated memory bandwidth using the pessimistic assumption that the input matrix values are never cached, a value Koza et al. refer to as β_- [10]. This is often higher than can actually be achieved if every access went to DRAM, demonstrating the benefit of the GPU's caches on vector accesses.

VI. Related Work

Designing good CSR-based SpMV algorithms for GPUs has turned out to be surprisingly difficult. The first algorithms, CSR-Scalar and CSR-Vector, each had its own limitations. The former performed poorly due to memory serialization, the latter due to underutilizing the hardware.

Bell and Garland thus came to the conclusion that new formats were required for SpMV GPUs [3]. This resulted in proposals for a plethora of other formats; Langr and Tvrdík list over 50 different formats [11]. This has led to auto-tuning frameworks that dynamically analyze the matrices to pick the best format [4, 15, 20]. We also discussed a number of advanced CSR-based works in Section II.

We previously showed that CSR-Adaptive performed better than these other mechanisms [7]. Though some matrices were better stored in specialized formats, CSR-Adaptive performed better in general. Liu et al showed that CSR-Adaptive performed poorly on matrices with a small set of very long rows [13]. This work set out to fix that issue.

LightSpMV [14] accelerated CSR-Vector through intelligent caching and reducing workgroup launch overheads by dynamically assigning rows to each workgroup. They show up to 2× higher performance compared to the CUSP library, though this was lower than CSR5 for the matrices they tested. One benefit of LightSpMV (and a number of the algorithms discussed in Section II) is that it requires no matrix preprocessing.

Guo and Gropp previously demonstrated using more than one workgroup on long rows [9]. Their algorithm is based on CSR-Vector, but it requires an expensive sort of the matrix and vector in order to group rows of similar lengths. Our solution shows superior performance and requires no sorting.

Adaptive CSR (ACSR) is a recent proposal for high-performance CSR-based SpMV [1]. ACSR also bins rows of similar lengths, but it holds the binning information in new data structures rather than sorting the arrays. Because the array remains unsorted, disparate rows of similar sizes can be accesses in parallel, causing uncoalesced memory accesses. Liu et al. found that CSR5 performed better than ACSR for both regular and irregular matrices.

VII. Future Research and Conclusions

CSR-based SpMV on GPUs is desirable because it avoids the runtime and storage overheads that afflict implementation-specific storage formats. Existing CSR-based approaches like CSR-Adaptive have improved the performance of SpMV on regular matrices but demonstrate poor performance on irregular matrices with very long rows.

We improve CSR-Adaptive to better utilize the GPU resources. We also propose a novel algorithm to efficiently compute SpMV on long rows. The improved CSR-Adaptive achieves high performance regardless of the matrix structure by dynamically choosing among three different algorithms that cater to different row lengths. Our novel techniques effectively double the performance of CSR-Adaptive and enable it to attain 36% higher performance at 10× less overhead than CSR5, the current state of the art.

We found that, while our new algorithm offers superior performance, some matrices do not perform as well as others. This is because they have rows with widely dispersed values; the access to the associated input vector entries becomes the new bottleneck. Studying mitigation techniques for this problem can be fruitful in future works.

Because CSR-Adaptive allows the same data structures on both CPUs and GPUs, there are interesting future studies in CSR-based algorithms on heterogeneous processors. For example, perhaps rows with poor vector access patterns could execute on CPUs, which usually have larger caches and more advanced prefetching mechanisms than GPUs.

The algorithm described in this paper is available as part of the open source clSPARSE library. The source code can be found at https://github.com/clMathLibraries/clSPARSE/.

Acknowledgements

We wish to thank Weifeng Liu for providing his CSR5 data and source code; this greatly helped our comparisons. Thanks also to Kent Knox for all of his work on clSPARSE and for pushing to include initialization in CSR-VectorL.

AMD, the AMD Arrow logo, FirePro and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple, Inc. used by permission by Khronos. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[2] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan. An Efficient Two-dimensional Blocking Strategy for Sparse Matrix-Vector Multiplication on GPUs. In *Proc. of the Int'l Conf on Supercomputing (ICS)*, 2014.

[3] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-oriented Processors. In *Proc. of the Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[4] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[5] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proc. of the Workshop on General-Purpose Computing on Graphics Processing Units (GPGPU)*, 2010.

[6] I. S. Duff, M. A. Heroux, and R. Pozo. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *Trans. on Mathematical Software*, 28(2):239–267, 2002.

[7] J. L. Greathouse and M. Daga. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *Proc. of the Int'l Conf for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[8] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward Realistic Performance Bounds for Implicit CFD Codes. In *Proc. of the Int'l Parallel Computational Fluid Dynamics Conf. (PARCFD)*, 1999.

[9] D. Guo and W. Gropp. Adaptive Thread Distributions for SpMV on a GPU. In *Proc. of the Extreme Scaling Workshop*, 2012.

[10] Z. Koza, M. Matyka, S. Szkoda, and Ł. Mirosław. Compressed Multiple-Row Storage Format for Sparse Matrices on Graphics Processing Units. *SIAM Journal on Scientific Computing*, 32(2):C219–C239, 2014.

[11] D. Langr and P. Tvrdík. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Trans. on Parallel and Distributed Systems*, PP(99):1–1, 2015.

[12] W. Liu and B. Vinter. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *Proc. of the Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2014.

[13] W. Liu and B. Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proc. of the Int'l Conf. on Supercomputing (ICS)*, 2015.

[14] Y. Liu and B. Schmidt. LightSpMV: Faster CSR-based Sparse Matrix-Vector Multiplication on CUDA-enabled GPUs. In *Proc. of the Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, 2015.

[15] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *Proc. of the Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[16] M. Naumov, L. S. Chien, P. Vandermersch, and U. Kapasi. CUSPARSE Library. Presented at the GPU Technology Conference, 2010.

[17] I. Reguly and M. Giles. Efficient Sparse Matrix-Vector Multiplication on Cache-based GPUs. In *Proc. of Innovative Parallel Computing (InPar)*, 2012.

[18] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *Int'l Workshop on GPUs and Scientific Applications (GPUScA)*, 2010.

[19] L. O. Steven Dalton, Nathan Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2015.

[20] B.-Y. Su and K. Keutzer. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proc. of the Int'l Conf. on Supercomputing (ICS)*, 2012.

[21] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huynh, X. Li, and R. S. M. Goh. Optimizing and Auto-tuning Scale-free Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *Proc. of the Int'l Symp. on Code Generation and Optimization (CGO)*, 2015.

[22] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Proc. of SciDAC, Journal of Physics: Conf. Series*, 2005.

[23] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.

[24] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2014.