Efficient Sparse Matrix-Vector Multiplication on GPUs using the CSR Storage Format

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

***Abstract*—The performance of sparse matrix vector multiplication (SpMV) is important to computational scientists. Compressed sparse row (CSR) is the most frequently used format to store sparse matrices. However, CSR-based SpMV on graphics processing units (GPUs) has poor performance due to irregular memory access patterns, load imbalance, and reduced parallelism. This has led researchers to propose new storage formats. Unfortunately, dynamically transforming CSR into these formats has significant runtime and storage overheads.**

We propose a novel algorithm, CSR-Adaptive, which keeps the CSR format intact and maps well to GPUs. Our implementation addresses the aforementioned challenges by (i) efficiently accessing DRAM by streaming data into the local scratchpad memory and (ii) dynamically assigning different numbers of rows to each parallel GPU compute unit. CSR-Adaptive achieves an average speedup of 14.7× over existing CSR-based algorithms and 2.3× over clSpMV cocktail, which uses an assortment of matrix formats.

Keywords: Sparse matrix-vector multiplication (SpMV); general purpose computation on graphics processing units (GPGPU); compressed sparse row (CSR); AMD; performance acceleration;

I. Introduction

Sparse matrices are extensively used in areas such as linear algebra [27], data mining [14], and graph analytics [11]. As such, accelerating sparse primitives is an important goal with wide-reaching consequences. Sparse matrix-vector multiplication (SpMV) is one of the fundamental primitives that form sparse basic linear algebra subprogram (BLAS) libraries [8] and is used extensively in iterative methods for linear systems and eigenvalue problems. SpMV performance is heavily dependent on the format used to store the sparse matrix in memory. Compressed sparse row (CSR) has historically been a frequently used format because it efficiently compresses both structured and unstructured matrices, and it is amenable to CPU-based algorithms [31].

Graphics Processing Units (GPUs) are routinely used to accelerate regular linear algebra problems. However, SpMV implementations that use CSR have shown disappointing performance due to issues such as load imbalance, lack of parallelism, and irregular, uncoalesced memory accesses [3].

The naïve CSR-based parallel SpMV, known as CSR-Scalar, assigns each row of the sparse matrix to a separate thread [10]. This works well on CPUs, but causes uncoalesced, slow memory accesses on GPUs. In addition, hardware resources may be underutilized when long rows cause a thread to have more work than its neighbors.

Previous work has described an improved GPU-based approach, CSR-Vector [3]. This algorithm assigns one unit of single instruction, multiple data (SIMD) execution (a wavefront, also known as a warp) to work on a single row of the matrix. This allows wavefronts to access consecutive memory locations in parallel, resulting in fast coalesced loads. However, CSR-Vector can lead to poor GPU occupancy for short rows. For example, a 64-wide SIMD unit will have numerous unused execution resources on a row with only 32 non-zero values.

Numerous proposals have claimed that new storage formats are required for good GPU SpMV performance [3, 5, 18, 26, 28]. The conversion to these other formats has two major issues - (i) changing software to use a new format presents a large engineering hurdle, as enormous amounts of software already use CSR, and (ii) using CSR matrices for parts of the application and converting to a different format for GPU-based SpMV incurs large runtime overheads and requires extra storage space. These issues jeopardize the widespread adoption of GPUs for sparse linear algebra.

We propose a novel algorithm called *CSR-Stream* to compute SpMV on GPUs while keeping the CSR format intact. CSR-Stream statically fixes the number of non-zero values that will be processed by one wavefront and streams all of these values into the local scratchpad memory. This stream of loads is coalesced and effectively utilizes the GPU's DRAM bandwidth, alleviating the bottleneck of CSR-Scalar. CSR-Stream also efficiently utilizes the GPU's parallel resources and mitigates the bottleneck of CSR-Vector by dynamically determining the number of rows on which each wavefront will operate.

CSR-Stream loses efficiency when a wavefront operates on rows with a large number of non-zero values, and it becomes inoperative if a row has more values than can be allocated in the scratchpad. To overcome this limitation, we dynamically determine whether to execute a set of rows with CSR-Stream or with the traditional CSR-Vector, which offers good performance on long rows. We call

SC14, November 16–21, 2014, New Orleans
978-1-4799-5500-8/14/$31.00

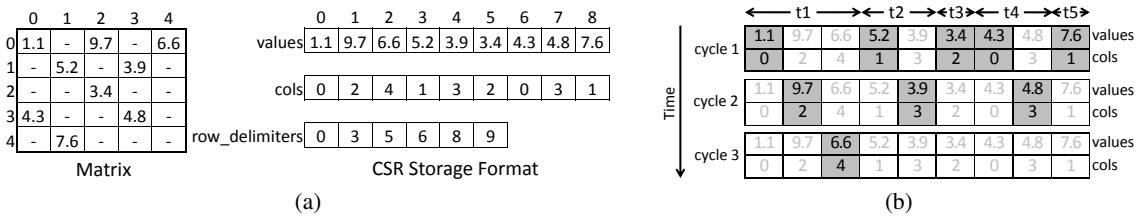

Figure 1. Example of the Compressed Sparse Row (CSR) sparse matrix storage format. (a) The CSR storage for the sparse matrix on the left. (b) An illustration of how multiple CPU threads would concurrently access the CSR data structure when performing SpMV with one thread per row.

this combination *CSR-Adaptive*. We evaluate CSR-Adaptive on an AMD FirePro™ W9100 discrete GPU. Over the set of sparse matrices we tested, CSR-Adaptive shows an average performance increase of 14.7× over existing CSR-based SpMV implementation and a speedup of 2.3× over clSpMV's cocktail, which is built from collections of other GPU-amenable storage formats.

Our work makes the following contributions:

- We propose a new algorithm, CSR-Stream, to compute sparse matrix-vector multiplication on GPUs using the CSR storage format when the matrix rows are short.
- We develop CSR-Adaptive, an algorithm that dynamically decides whether to use our CSR-Stream or the traditional CSR-Vector for each row of the matrix.
- We demonstrate that our algorithm performs better than previous GPU-based SpMV algorithms, both those that use CSR as well as those that use other GPU-optimized storage formats. This is the case *even when discounting the significant data transformation overheads required by these other formats*.

The remainder of this paper is arranged as follows. Section II provides a background on various sparse matrix storage formats and AMD GPUs. Section III details our CSR-Adaptive algorithm. Section IV explains our experiments methodology and Section V shows the results. We present related work in Section VI, followed by future research directions and our conclusions in Section VII.

II. Background

A. Sparse Matrix-Vector Multiplication

Modern high-performance computing (HPC) systems rely on SpMV for numerous tasks. SpMV operates on compressed *sparse matrices*, which are primarily filled with zeroes. This compression is especially important to the HPC community because it allows extremely large matrices to be used in modern computers that have relatively small amounts of storage.

The storage format used for the sparse matrix defines the SpMV algorithm, and it significantly impacts performance. CSR is a widely used format that offers excellent compression of both structured and unstructured sparse matrices. The performance of CSR-based SpMV is good on CPUs, and several other sparse BLAS algorithms also support CSR. As such, a great deal of existing software has been written to use CSR.

Figure 1(a) illustrates the CSR storage format for an example matrix. The non-zero values are stored in a `values` array. Each non-zero value also has an entry in the `cols` array in order to help describe its location in the matrix. Finally, rather than holding the row index of every non-zero value (as is done in the COO format), CSR has a `row_delimiters` array that holds the indices for only the first value of every row. This row-oriented compression is the source of the format's name.

Figure 1(b) shows how SpMV is commonly performed on CPUs when using CSR, where each thread operates on a separate row or chunk of rows. As the threads iterate over the non-zero values, they pull sections of the CSR data structures into their private caches, where the next few iterations can access them. SpMV is traditionally bandwidth bound (i.e., its arithmetic intensity is very low), so good memory access patterns greatly improves performance [12, 31].

B. Graphics Processing Units

GPUs have evolved over the last decade from relatively fixed-function circuits to highly parallel general-purpose processors. Modern GPUs concurrently execute thousands of threads and quickly switch between them to hide long-latency operations. In order to reduce hardware complexity (and thus allow more parallel compute units in a chip), GPUs bundle numerous threads together and require them to execute in a SIMD fashion. In AMD GPUs, each of these wavefronts contains 64 threads. Figure 2 shows an illustration of a modern AMD GPU. Each of the parallel compute units (CUs) itself has some number of parallel execution resources. These resources are shared across a wavefront, meaning that a wavefront which does not have all of its threads active leaves some hardware unused.

GPUs are capable of utilizing large amounts of memory bandwidth. Modern discrete GPUs can utilize 200-300 GB/s, while CPUs of similar cost reach 30-60 GB/s. To further increase usable bandwidth, modern GPUs also contain a series of on-chip caches. In AMD GPUs, each compute unit has a private L1 data cache that is connected to shared L2 caches and memory controllers through a crossbar.

The L1 caches can communicate up to 64 bytes to their CU every cycle, but they have microarchitectural limitations

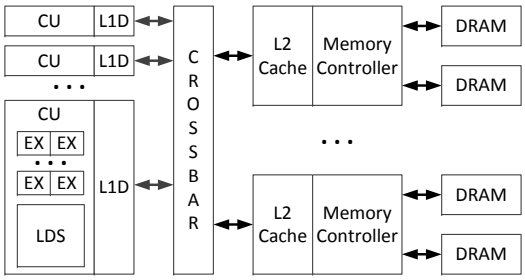

Figure 2. The memory and interconnect of AMD Graphics Core Next GPUs. Each CU has a private L1 data cache (L1D) and a local data store (LDS) that is shared amongst its execution units. L2 caches are associated with memory channels and are shared across all CUs.

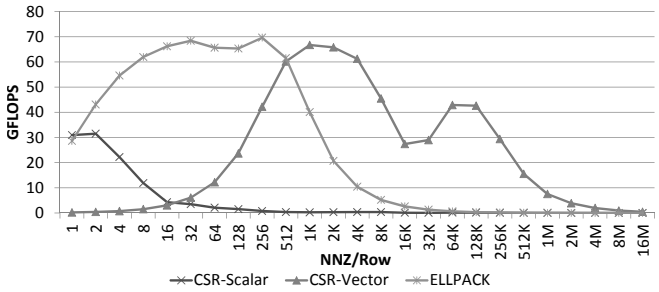

Figure 3. The SpMV performance on an AMD FirePro™ W9100 GPU using different sparse matrix formats. CSR-Scalar performs poorly for almost all row lengths, while CSR-Vector performs well on long rows. ELLPACK performs well on short rows, but requires changing the matrix storage format.

on the number of cache lines that can be accessed each cycle. If a single wavefront's access requires touching many lines, the requests are serialized and L1 bandwidth is reduced. The per-CU local data store (LDS, also known as shared memory) is a software-addressed scratchpad cache. In AMD GPUs, the LDS is highly banked and can quickly service uncoalesced (scatter/gather) accesses. A *workgroup* is a collection of wavefronts that execute on the same CU and that are able to communicate through resources like the LDS.

C. Previous CSR-based GPU Algorithms

A simple CSR-based SpMV algorithm, as illustrated in Figure 1, assigns a row to each thread. While this algorithm, *CSR-Scalar*, works well on CPUs, it is ill-suited for GPUs [3]. If each row contains a large number of non-zero values (NNZ), then the threads will access entries in both `values` and `cols` that are far apart. Such accesses cause each thread in a wavefront to read from different L1 cache lines and touch numerous DRAM pages. This serializes the wavefront's accesses and reduces the achievable bandwidth.

Figure 3 illustrates this problem on an AMD FirePro™ W9100 GPU using an experiment similar to the one described by Bell and Garland [3]. We perform SpMV on a dense matrix (stored in a sparse format) with a fixed number of non-zero values, 16 million in this case. Each point on the X axis has a different number of rows and thus a different NNZ/row. This demonstrates how the performance of CSR-Scalar plummets as the NNZ/row increases beyond a small number. At 16 NNZ/row, each thread in a wavefront accesses a different 64-byte segment of memory during a vector load, resulting in poor memory system performance.

Bell and Garland described *CSR-Vector*, a CSR-based SpMV algorithm where threads in a wavefront all work on a single row [3]. For rows with many non-zero values, this results in good performance by using large coalesced memory accesses followed by a short reduction phase. Unfortunately, many sparse matrices have few non-zeroes per row. CSR-Vector performs poorly in this case, as there is little parallel work in each wavefront to amortize the cost of the reduction. This is demonstrated in Figure 3, where the performance of CSR-Vector drops when the NNZ/row is small. (The performance of CSR-Vector is also low for the longest rows in Figure 3; in this case, there are not enough rows to completely utilize all of the CUs.)

D. Other Sparse Matrix Formats

Because of the poor GPU performance of these CSR algorithms, the common consensus is that GPU-based SpMV is best served by other storage formats. Bell and Garland presented some of the first work in this direction when they showed that the column-major ELLPACK format presented much better performance for matrices with short rows [3]. This algorithm is also demonstrated in Figure 3. They further described a hybrid ELLPACK+COO format that remedied the difficult problems of using ELLPACK on long rows or in matrices with variable NNZ/row.

Other formats have been proposed across the literature [5, 18, 28]. To this end, Su and Keutzer described the clSpMV framework, which automatically converts sparse matrices into any of nine different formats (or a "cocktail" format that uses different formats on subsets of the matrix) [26]. The main limitation in each of these cases is the expensive conversion from CSR to the new matrix format.

Existing CPU-centric software relies heavily on the CSR format. As such, transforming the matrix to one of these GPU-optimized formats has both space and time overheads. While SpMV is often used in iterative algorithms, the conversion time from CSR to even simple alternative formats can still require hundreds or thousands of SpMV iterations to amortize. Changing storage formats would require overwriting the original CSR data using costly in-place algorithms, or the system must have enough available storage for both copies. HPC applications are regularly limited by available system memory, so the latter is a disagreeable proposition.

While the entire application could be rewritten to use these other formats, this would likely require great effort. Requiring different data structures between the CPU and GPU also presents problems for heterogeneous processors that include both types of cores [22]. It becomes harder to share work between the heterogeneous units on these devices if they do not use the same data structures.

The desire to continue to use CSR leads us to study algorithms that can attain good performance on GPUs without needing a new sparse matrix storage format.

III. A Better CSR-based SpMV for GPUs

The goal of this work is to describe a high-performance CSR SpMV, no matter the NNZ/row. We begin by focusing on matrices with "short" rows, since many interesting physical systems are represented by such matrices [10].

A. CSR SpMV on Short Sparse Rows

For the purposes of discussion, we loosely define "short" rows as those that have so few non-zeroes per row that CSR-Vector performs poorly. Since CSR-Scalar performs poorly almost everywhere, this is a region where CSR is believed to be a bad storage format for GPUs. The problem for CSR-Vector is a lack of parallel work, while the problem for CSR-Scalar is poor memory coalescing. The first stage of our CSR-based GPU SpMV solves both of these problems.

If a large sparse matrix has few non-zeroes per row, there are, by definition, abundant rows upon which to work. CSR-Scalar takes advantage of this fact by spreading these rows to the parallel execution units. Each thread thus accesses values from different rows, which will be stored far apart in the row-major CSR format. GPU memory accesses perform best when simultaneously accessing contiguous regions of memory, rather than random or distant accesses. As such, the uncoalesced accesses to both the `cols` and `values` arrays hamper CSR-Scalar performance.

If we momentarily focus on sparse matrices where every row has the same, small NNZ, we can solve this problem using GPU hardware designed to simultaneously access non-contiguous locations. We will generalize this solution later.

Modern GPUs contain software-addressed scratchpad memories that are local to each parallel compute unit. These are called LDS in AMD GPUs. The LDS is specifically built to help with scatter and gather operations – it is highly banked and allows many simultaneous accesses to different locations without stalling [1]. The primary limitations of the LDS are that data must be manually moved into the structure and that kernels must define at compile time the amount of LDS each workgroup will use.

These are not a problem if we are writing an algorithm for a sparse matrix with a constant, small NNZ/row. By way of example, if each row had 16 non-zero values, then a LDS partition of 1024 entries will suffice to hold the values for 64 rows. AMD wavefronts are 64 threads wide, so this offers enough parallelism to perform an algorithm like CSR-Scalar fully in parallel using the LDS. The only step left is to move data from the DRAM into the LDS, which can be done at high performance by using the parallel threads to stream all 1024 values into the LDS.

The first step in our new CSR-based SpMV algorithm is thus to have each workgroup perform many parallel, coalesced loads from the `col` and `value` arrays. This quickly places values into the LDS, whereupon a scalar-style reduction (with each thread working on a single row) completes the SpMV at full speed.

```
   input : values[], cols[], row_delimiters[], x[]
   output: output[]
 1 local float LDS[NNZ_PER_WG];
 2 startRow←workgroupID * ROWS_PER_WG;
 3 stopRow←(workgroupID + 1) * ROWS_PER_WG;
 4 localRow←startRow + localTid;
 5 first_col←row_delimiters[startRow];
 6 /* Stream from values[] and cols[]        */
 7 for i←localTid to NNZ_PER_WG - 1 do
 8 |   LDS[i]←values[first_col+i] * x[ cols[first_col+i] ];
 9 |   i + = THREADS_PER_WG;
10 end
11 /* Scalar-style reduction from LDS        */
12 if startRow + localTid < stopRow then
13 |   temp←0, i←(row_delimiters[localRow] - first_col);
14 |   for i to row_delimiters[localRow+1] - first_col - 1 do
15 |   |   temp + = LDS[i++];
16 |   end
17 end
18 output[startRow + localTid] = temp;
```

Algorithm 1. Partial GPU CSR Algorithm. This algorithm assigns a fixed number of rows per workgroup (ROWS_PER_WG) and assumes that, at most, NNZ_PER_WG non-zero values are in those rows.

The solution as currently described only works if the NNZ/row is fixed and small. The second half of the algorithm (the scalar reduction) statically knows which values each thread should read from the LDS. If the NNZ/row varies, this must change slightly. Instead of hard-coding the LDS entries that each thread will access, we can use the `row_delimiters` array to dynamically calculate each row's start and end point. The first LDS entry a thread should access is defined by the difference between its `row_delimiters` value and that of the first row in the workgroup. The last value each thread will load is defined similarly. This partial solution is shown in Algorithm 1.

This example code has neither of the major problems previously discussed. Unlike CSR-Scalar, the memory accesses are fast and coalesced. Unlike CSR-Vector, there is enough work for multiple threads even when there are few NNZ/row because each workgroup operates on multiple rows.

B. Variable Number of Rows per Workgroup

Algorithm 1 works only if the non-zero values for the *fixed number of rows* fits within the statically allocated LDS space. If, for instance, a set of 64 rows has 1024 non-zeroes, the algorithm would work even if each row was not exactly 16 values wide. It does not work when the 64 rows have more than 1024 values. If 63 rows had 16 non-zero values, but one row had 17, the current algorithm would not stream that last remaining value into the LDS.

We solve this by dynamically calculating *how many rows* will fit into the LDS. For example, assume there are 128 rows in a matrix, each with 16 non-zero values. In this

```
input : totalRows, row_delimiters[]
output: row_blocks[]
rowBlocks[0]←0, tempSum←0, last_i←0, ctr←1;
for i←1 to totalRows - 1 do
    /* Count non-zeroes in this row        */
    sum += row_delimiters[i] - row_delimiters[i-1];
    if sum == LOCAL_SIZE then
        /* This row fills up LOCAL_SIZE    */
        last_i←i, row_blocks[ctr++]←i, sum←0
    else if sum > LOCAL_SIZE then
        if i - last_i > 1 then
            /* This extra row will not fit */
            row_blocks[ctr++] ← (i−1), i−−;
        else if i - last_i == 1 then
            /* This one row is too large   */
            row_blocks[ctr++] ← i;
        end
        last_i←i, sum←0;
    end
end
rowBlocks[ctr++] ← totalRows;
```

Algorithm 2. This CPU code calculates the number of rows of a CSR matrix that can fit into LOCAL_SIZE LDS entries.

case, two workgroups, both with 1024 LDS entries, would each work on 64 rows. If the first row instead had 32 entries, and the remaining 127 rows had 16 entries, then three workgroups would be needed. The first would operate on 63 rows, the second on 64, and the third would work on only on the last row. This has the benefit of dynamically fitting the number of rows to the amount of LDS space, which simplifies the work done on the GPU.

This new part of our algorithm splits the problem into *row blocks*, where the non-zero values within a block fit into a statically sized amount of local storage. Algorithm 2 shows simple code for generating row blocks on the CPU, which we found performs better than computing the information in the GPU. The blocks need to be calculated only once when generating the matrix, or whenever the NNZ/row changes, rather than when the values in the matrix change. The complexity of calculating the blocks is related to the number of rows, not the number of non-zeroes in the matrix; this generally takes less than 1% of the time that it takes to generate the CSR data structure, as shown in Section V-C. The row blocks can also be calculated while transferring the matrix to the GPU's memory, hiding the extra latency.

Algorithm 3 describes at a high level the GPU code that utilizes these row blocks to perform SpMV. We refer to this algorithm as CSR-Stream, since it streams values from the memory into the LDS. We note that some of the finer details of this algorithm (such as implementation-specific optimizations) are abstracted away in this pseudocode description.

Essentially, the CPU uses the `row_delimiters` structure to quickly calculate how many rows each workgroup will access. After this, each GPU workgroup, which has a static amount of LDS storage, uses its parallel threads to efficiently load values from memory into the LDS in a

```
startRow ← row_blocks[workgroupID];
nextStartRow ← row_blocks[workgroupID + 1];
num_non_zeroes ← row_delimiters[nextStartRow] -
row_delimiters[startRow];
/* Omitted: Stream num_non_zeroes values
   into LDS                                  */
num_rows ← nextStartRow − startRow;
thread_start_point = row_delimiters[startRow + localTID];
thread_end_point = row_delimiters[startRow + localTID + 1];
/* Perform reduction on num_rows, rather
   than a static number.                     */
while rows_done < num_rows do
    /* Omitted: Scalar reduction out of the
       LDS for values between
       thread_start_point and thread_end_point */
    rows_done += workgroupSize;
end
output[startRow + localTid] = temp;
```

Algorithm 3. High-level pseudocode for CSR-Stream. We abstract away many of the implementation-specific details for the sake of clarity.

coalesced manner. Finally, `num_rows` threads perform a reduction of each row before writing the value to the output.

CSR-Stream may leave some threads inactive during the reduction phase, but this rarely reduces the peak performance. SpMV is heavily bandwidth bound, and the coalesced loads that put data into the LDS result in efficient bandwidth usage. The reduction can mostly be hidden under the bandwidth-related stalls. Logarithmic reductions may be preferable when there are many inactive threads.

C. Optimizing Very Long Rows

We do not use CSR-Stream to calculate extremely long individual rows. If a single row has more non-zero values than will fit into the LDS, a simple implementation of CSR-Stream will not move these values into the local memory.

This could be solved by giving each of these very long rows to a single CSR-Stream workgroup. The blocking code in Algorithm 2 causes this to happen – very large rows are assigned to a single row block. The workgroup handling that block could continuously add the values from memory to the previous values contained in the LDS. The final reduction step would then finish adding these values together.

We found that it was better to find these long rows and hand them off to the traditional CSR-Vector algorithm, which only loads the exact number of needed values into the LDS and performs a logarithmic (rather than linear) reduction at the end. After all, CSR-Vector already results in good performance for these long rows. This worked better than expanding CSR-Stream to work on such long rows.

This then completes our CSR-Based GPU SpMV algorithm, which we call *CSR-Adaptive*. CSR-Adaptive is shown in Figure 4 and described in pseudocode in Algorithm 4. When generating the CSR matrix on the CPU, a `row_blocks` structure is also generated that holds the start row for each block that fits into a workgroup's LDS. We adaptively decide that when the number of rows is small

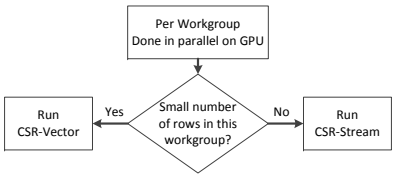

Figure 4. High-level flow chart of the CSR-Adaptive algorithm. The decision is made inside the single CSR-Adaptive GPU kernel.

```
1 startRow ← row_blocks[workgroupID];
2 nextStartRow ← row_blocks[workgroupID + 1];
3 num_rows ← nextStartRow − startRow;
4 if num_rows > SMALL_VALUE then
5 |  /* Execute CSR-Stream                    */
6 else
7 |  /* Execute CSR-Vector                    */
8 end
```

Algorithm 4. Pseudocode for CSR-Adaptive. The calls to either CSR-Vector or CSR-Stream do not require launching a new GPU kernel, since this decision is made on the GPU.

(e.g., if there is only one row in this block), the CSR-Vector algorithm is used to calculate the SpMV for that block. Otherwise, CSR-Stream streams values from memory into the LDS and then reduces the LDS values into the outputs.

CSR-Adaptive dynamically finds the right number of rows for each workgroup and uses the best SpMV method for rows of different length. In addition, the fundamental CSR data structure for the matrix is unchanged.

The generation of the `row_blocks` structure is much faster than transforming the CSR structure into another format, and it takes very little extra space. If the number of LDS entries per workgroup is 1024, it takes less than 0.1% the space of the `values` structure; it can be much smaller if the rows are very long.

IV. Experimental Setup

We test CSR-Adaptive on twenty different input matrices that have been studied in previous works [3, 26]. These sparse matrices, shown in Table I, demonstrate varied characteristics and the average NNZ/row ranged from 3 - 2634.

We implemented both CSR-Stream and CSR-Adaptive within the OpenCL™ version of the SpMV sample in the Feb., 2014, version of the Scalable HeterOgeneous Computing (SHOC) suite [6]. This allowed us to directly compare our algorithms against the SHOC versions of CSR-Vector, CSR-Scalar, and ELLPACK. We slightly modified the implementation of CSR-Vector in order to increase the performance of its parallel reduction step. We then also use this as the vector algorithm in CSR-Adaptive.

In addition to the SpMV implementations in SHOC, we also compare the performance of CSR-Adaptive against clSpMV v0.1 [26] and ViennaCL v1.5.2 [25]. clSpMV supports a number of different specialized matrix storage formats: CSR (with both Scalar and Vector algorithms), BCSR, DIA, BDIA, ELLPACK, SELL, BELL, SBELL, and

Table I
Overview of sparse matrices used for evaluation

Name	Size	Non-Zeroes (NNZ)	NNZ / Row
Dense2	2K * 2K	4,000,000	2,000
Protein	36K * 36K	4,344,765	119
FEM/Spheres	83K * 83K	6,010,480	72
FEM/Cantilever	62K * 62K	4,007,383	64
Wind Tunnel[1]	218K * 218K	11,634,424	53
FEM/Harbor[1]	47K * 47K	2,374,001	51
QCD[2]	49K * 49K	1,916,928	39
FEM/Ship[1]	141K * 141K	7,813,404	55
Economics	207K * 207K	1,273,389	6
Epidemiology	526K * 526K	2,100,225	4
FEM/Accelerator	121K * 121K	2,624,331	22
Circuit	171K * 171K	958,936	6
Webbase	1,000K * 1,000K	3,105,536	3
LP	4K * 1,097K	11,284,032	2,634
circuit5M	5,558K * 5,558K	59,524,291	11
eu-2005	863K * 863K	19,235,140	22
Ga41As41H72	268K * 268K	18,488,476	69
in-2004	1,383K * 1,383K	16,917,053	12
mip1	66K * 66K	10,352,819	156
Si41Ge41H72	186K * 186K	15,011,265	81

COO. We compare against all of these but combine the results into a single data point, "clSpMV Best Single," to improve readability. For each matrix, we run all ten algorithms but only display the fastest in this bar. clSpMV also supports a custom cocktail format, which splits the matrix into blocks and applies different formats to each block based on hardware-specific heuristics tuned by an offline training run. Because clSpMV only has the capability for single-precision computation, we perform all of our tests (unless otherwise noted) using single-precision floating point values.

ViennaCL supports three different versions of CSR-Scalar, two of which zero-pad the data structure so that rows contain multiples of four or eight values. This allows for faster scalar loads, but could also result in significant storage overheads. Nonetheless, we compare against all three in addition to the other three formats supported by ViennaCL: COO, ELLPACK, and the ELLPACK+COO HYB format. Like clSpMV, we present the best of the ViennaCL results in a single bar labelled "ViennaCL Best."

We performed all of our experiments on an AMD FirePro™ W9100 GPU with ECC disabled. Table II 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 operating system was a 64-bit version of CentOS 6.4, kernel version 2.6.32-358.23.2. The

[1]The Matrix Market files for these matrices contain elements whose value are zero. The input functions for SHOC, clSpMV, and ViennaCL treat these as non-zero values and thus perform some needless SpMV computation. We do the same for consistency with previous works, but this would be accounted for in production code.

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

Table II
OVERVIEW OF AN AMD FIREPRO™ W9100 GPU

Compute Units (CU)	44
Processing Elements	2816
Core Clock Rate	930 MHz
GDDR5 Memory Clock Rate	1250 MHz
Memory Size	16 GB
Peak Memory Bandwidth	320 GB/s
L2 Cache Size	1024 KB
L1 Cache Size per CU	16 KB
Local Data Store (LDS) per CU	64 KB
Single Precision Peak Performance	5238 GFLOPS
Double Precision Peak Performance	2619 GFLOPS

GPU was programmed using OpenCL™ 1.2 [19] with the AMD APP SDK v2.9 and AMD FirePro driver v14.20 Beta. We used 256 threads per workgroup, and all of the performance numbers are an average of 100 runs.

V. EVALUATION

In this section, we first tune the row block size and then present performance results of CSR-Adaptive compared to prior SpMV implementations on the GPU using synthetically generated and unstructured sparse matrices.

A. Tuning Block-size

As mentioned in Section III, CSR-Adaptive processes a statically fixed NNZ per workgroup. The number of values, the "block size," will affect the performance due to changing the parallelism in the kernel and latency hiding due to LDS capacity constraints. In this section, we empirically determine the best NNZ to be included in each block and use that value for the number of entries assigned to the LDS.

Figure 5 illustrates the harmonic mean of the single-precision SpMV performance for the matrices listed in Table I for block sizes ranging from $512 - 8192$. These values are normalized to the block size with best performance. We did not use block sizes greater than 8192 because our GPU only allows up to 32 KB of local data storage per workgroup. As shown in the figure, we find that processing blocks of 1024 non-zero values performs the best.

Using a block-size of 1024 results in enough threads in flight to efficiently hide memory latency on the GPU, as well as sufficiently utilize LDS to improve memory bandwidth. 1024 non-zero values utilize 4 KB of local data-store in single-precision, which allows 16 workgroups to be simultaneously executed. Larger block sizes reduce this capability. In contrast, smaller blocks inefficiently use the LDS, and threads access data from global memory most of the time, leading to poor memory bandwidth utilization.

After tuning the block-size, we also determined the number of rows within a block that would serve as the cutoff point between the CSR-Stream and CSR-Vector algorithm ("SMALL_VALUE" in Algorithm 4). We chose this value to be two. In other words, if a row block contains one or two rows, CSR-Vector will be used, while a row block with more rows will use CSR-Stream.

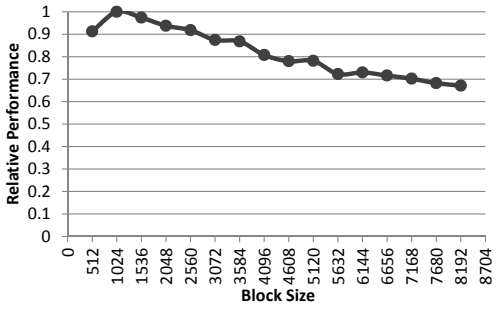

Figure 5. Performance of CSR-Adaptive with varying number of non-zero values processed in a workgroup.

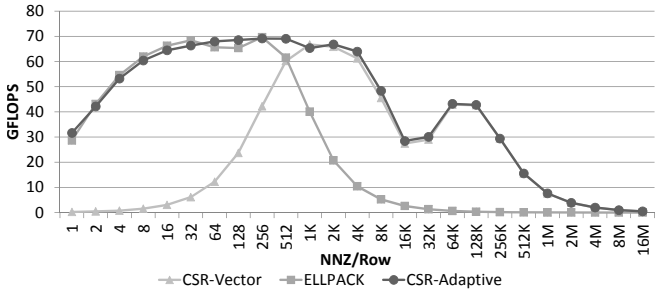

Figure 6. SpMV performance on dense matrices with a fixed number of non-zero values per row and variable number of rows.

B. Synthetically Generated Matrices

We explore the performance of various SpMV implementations using a collection of synthetically generated dense matrices stored in different sparse storage formats. These matrices have the same NNZ, $2^{24} \approx 16.7$ million, but have different number of rows and thus different NNZ/row. Each row has the same NNZ as all other rows: the matrices range from one row with 16 million non-zero values to 16 million rows with one non-zero value per row. This is the same experiment used to generate Figure 3. In this section, we look at three different SpMV implementations - (i) CSR-Vector, (ii) ELLPACK, and (iii) CSR-Adaptive. Figure 6 demonstrates how these implementations perform with varying number of rows and NNZ/row.

As also discussed in Section II-B, Figure 6 shows that the performance of SpMV is highly dependent on the matrix characteristics. CSR-Vector performs best when there are enough rows in the matrix to launch sufficient wavefronts to fully occupy the GPU and when the rows have enough non-zero values to keep the threads busy. CSR-Vector is inefficient when either of these conditions are not true, as demonstrated by the performance when the number of non-zero values per row is $>= 256$K and $<= 256$, respectively.

ELLPACK performance peaks when the matrix consists of enough rows to maximize the number of threads active on a GPU. This is slightly different than CSR-Vector, since ELLPACK needs enough rows to keep all *threads* active, rather than *workgroups*. As such, ELLPACK needs more

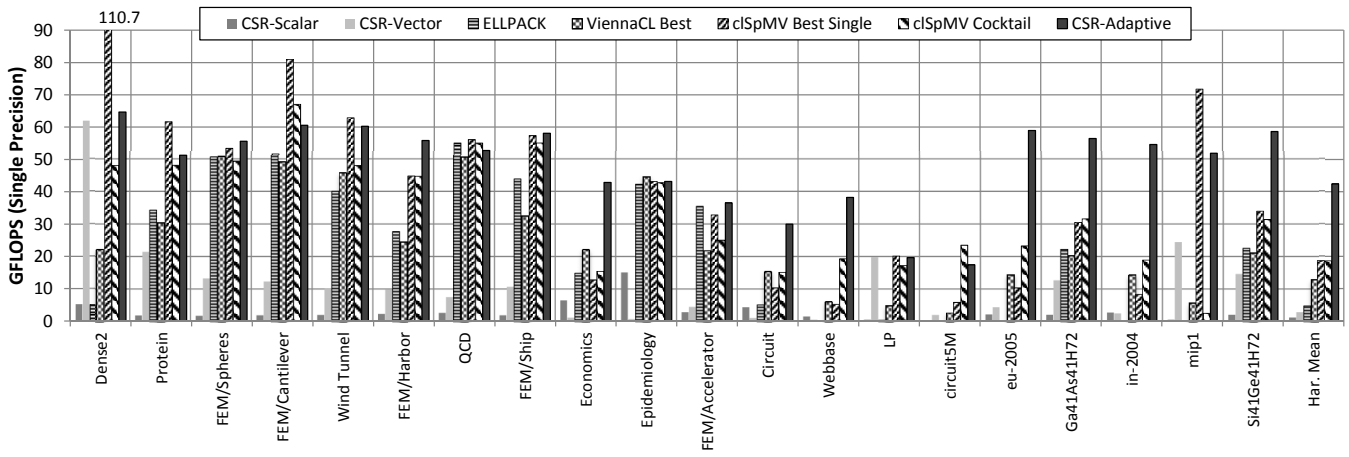

Figure 7. Performance of various implementations of SpMV in single precision using unstructured matrices.

rows to reach good performance. This is why the performance of ELLPACK degrades for long rows with >= 1K non-zero values. In these tests, ELLPACK coalesces memory accesses well and yields good performance for matrices with few NNZ/row. For these benchmarks, CSR-Vector performs better for long rows and ELLPACK is better for short rows. However, neither of these implementations is optimal for a spectrum of non-zero values per row.

CSR-Adaptive bridges the gap and matches the performance of CSR-Vector for long rows since it uses the same algorithm. For short rows, however, it matches the performance of ELLPACK by using the CSR-Stream algorithm. In this case, both algorithms produce the same style of fast coalesced memory accesses and become DRAM bandwidth bound. For 512 NNZ/row, CSR-Adaptive performs better than either of the other implementations.

C. Unstructured Matrices

In this section, we compare performance of CSR-Adaptive against the SHOC suite's CSR and ELLPACK implementations, the six algorithms supported by ViennaCL (CSR-Scalar, CSR-Scalar-4, CSR-Scalar-8, COO, ELLPACK, HYB), the ten individual algorithms used by clSpMV (CSR-Scalar, CSR-Vector, BCSR, DIA, BDIA, ELLPACK, SELL, BELL, SBELL, COO), and the clSpMV Cocktail format. For each matrix, the best of the six ViennaCL algorithms is denoted as "ViennaCL Best" and the best of the clSpMV algorithms is denoted as "clSpMV Best Single".

Figure 7 illustrates the performance of various implementations in single precision on the collection of unstructured matrices listed in Table I. There is no single format which is best for all the matrices. However, CSR-Adaptive often equals or outperforms all other implementations and is the best individual format on average. The major exceptions to this are `Dense2`, `Protein`, `mip1` and `FEM/Cantilever`.

`FEM/Cantilever` is a strongly diagonal matrix and benefits from the significant storage reduction offered by the specialized DIA format (this is the only matrix we tested that could successfully use the DIA format). `Dense2` and `mip1` are best served by the BCSR storage format because they can easily be blocked without requiring a great deal of zero-padding. This reduces the amount of data that must be read per non-zero value, yielding better performance. `Protein` performs best with SBELL for similar reasons.

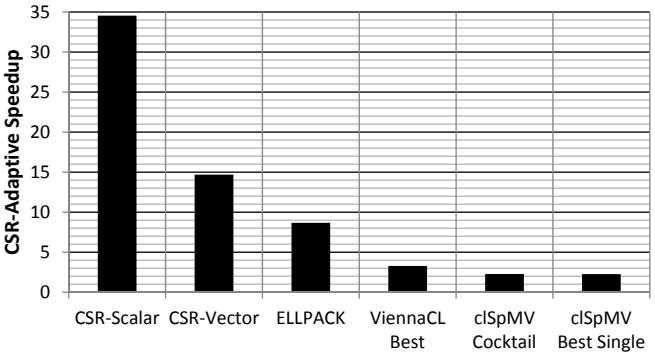

Figure 8. Average speedup of CSR-Adaptive over other implementations.

Even though these specialized format work well for particular matrices, there are cases when CSR-Adaptive achieves a speedups of 5-8× over the best of the ten non-Cocktail clSpMV algorithms and the six ViennaCL algorithms, as exemplified by `Webbase`, `eu-2005` and `in-2004`. Figure 8 presents the average speedup achieved by CSR-Adaptive over other SpMV implementations, which range from 34× for CSR-Scalar, to 2.3× for both clSpMV Cocktail and Best Single.

Figure 9 compares the performance of CSR-Adaptive in single and double precision. From the figure, we note that on an average the double precision performance is 30% slower than single precision. As shown in Table II, the double precision throughput of the AMD FirePro™ W9100 GPU is ½ its single precision throughput. Therefore, only a 30% slowdown corroborates that performance of SpMV is limited to available bandwidth rather than computational throughput.

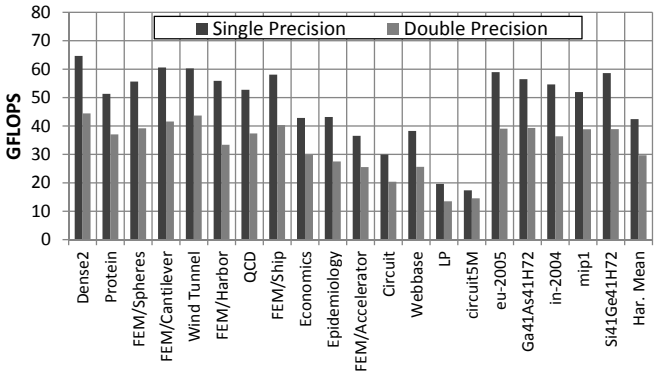

Figure 9. Performance of CSR-Adaptive in single and double precision on AMD FirePro^{TM} W9100 GPU.

Memory Bandwidth: Figure 10 depicts the theoretical memory bandwidth achieved by CSR-Adaptive compared to previous CSR-based SpMV implementations and ELLPACK. The CSR bandwidth is derived using the formula for total bytes transferred, as presented by Gropp et al., divided by the measured runtime [12]. This formula assumes that the CSR structure and vector are loaded from DRAM, but that any subsequent accesses to the vector values are retrieved from the cache. Essentially, it assumes a cache of infinite size that only takes demand misses. The ELLPACK values use a similar formula that assume an infinite cache and that NNZ values must be loaded from each of the `data` and `indices` arrays (which discounts needless loads of padding zeroes). We do not compare against other storage formats because they are difficult to theoretically characterize.

If the idealizations (infinite cache size, no padding values) hold true, the bandwidths shown in Figure 10 should equal the GPU's stream bandwidth. The SHOC `DeviceMemory` program, a good approximation of a GPU stream test, shows that our GPU can transfer about 280 GB/s from its DRAM. Due to protocol overheads and other inefficiencies, this test fails to reach the peak theoretical bandwidth of 320 GB/s. Figure 10 shows that, while no graph perfectly meets this idealism, CSR-Adaptive can greatly increase the achievable bandwidth compared to previous CSR-based algorithms. The average bandwidth achieved by CSR-Adaptive is 189 GB/s, roughly 68% of the maximum stream bandwidth. This is an order of magnitude greater than the other implementations.

Matrices such as `LP` and `circuit5M` fall further below the card's maximum achievable bandwidth due to other inefficiencies in the memory system. For example, accessing a single 4-byte vector entry first requires loading a 64-byte cache line from memory. This load, in a theoretical model with an infinite-sized cache, would only require four or zero bytes to be loaded from memory or cache, respectively.

Storage Format Generation: In addition to achieving better performance than other implementations, another important benefit of CSR-Adaptive is that it maintains the CSR data structure, which is commonly used and relatively fast to generate. SpMV is often used in iterative algorithms where the matrix is generated once and SpMV is performed numerous times. If the data structure generation time is low and the iteration count is high, the cost of generating the matrix can be amortized. However, many of these algorithms are input dependent; they may only run a small number of iterations, making it difficult to amortize the matrix generation overhead.

Figure 11 shows the overhead of converting from the simple COO format to a number of other formats. Because CSR is a common format, and most algorithms likely strive to amortize the CSR generation cost, we normalize all of this data to the time taken to convert from COO to CSR. Data structure generation is generally considered to be amortized when it takes a small fraction of the total time spent in SpMV iterations. Let us assume that N SpMV iterations are required to amortize the cost of CSR generation. Amortizing the generation of a complex format, therefore, will require $Y \times N$ SpMV iterations, assuming the SpMV kernel runtimes do not change. This multiplicative factor, Y, is the ratio of the generation times of a complex format and CSR and is shown in Figure 11. The figure demonstrates that generating the row-block structure for CSR-Adaptive is extremely lightweight compared to generating other complex formats. Hence, CSR-Adaptive is especially useful for problems that iterate few times.

A concrete example is the matrix `mip1`. In this case, BCSR is 38% faster than CSR-Adaptive. However, the BCSR data structure takes 18× longer to generate than the row-block values needed for CSR-Adaptive. In this case, it actually takes BCSR 25× as many iterations compared to CSR-Adaptive to amortize its data structure generation cost. This means that BCSR is only beneficial if the algorithm using this graph iterates a large number of times. For most other matrices, in contrast, CSR-Adaptive takes less time to generate the data structure and operates faster, making it the better choice regardless of iteration count.

VI. Related Work

Due to its importance in the computational sciences, there exists a great deal of literature on accelerating SpMV. Williams et al. study SpMV across a number of modern parallel CPU architectures and found that the best performance came from tuning the algorithm and data structures to match the underlying hardware [32]. Tools like OSKI [30] attempt to take advantage of this insight by automatically optimizing the computation in order to increase cache hit rates and reduce bandwidth demands.

Because of the copious DRAM bandwidth available to GPUs, there has been significant interest in using these chips to accelerate SpMV. Garland initially described CSR-Scalar with a code example and an early variant of CSR-Vector at a higher level [10]. That work presented no performance values, but mentions the possibility of using the scratchpad

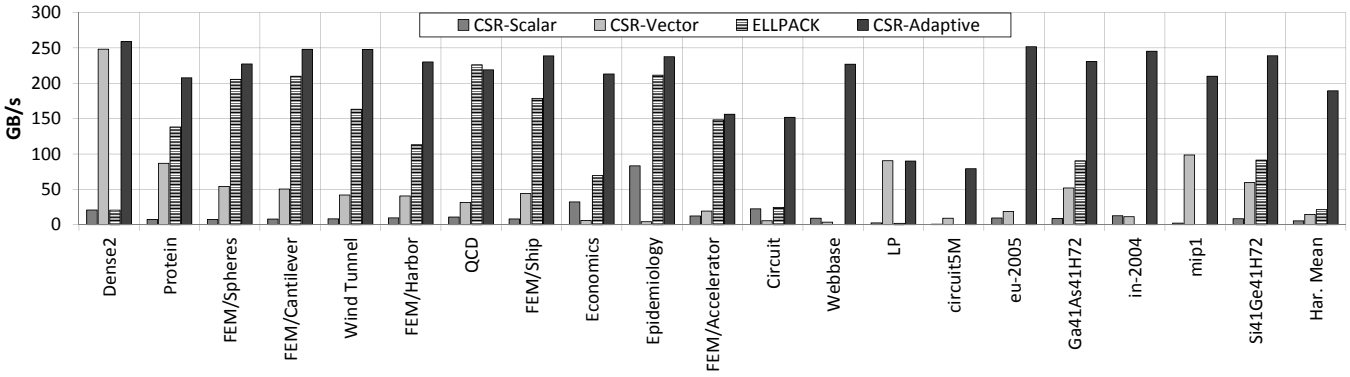

Figure 10. Theoretical memory bandwidth achieved by various CSR-based SpMV implementations on AMD FirePro™ W9100 GPU assuming that all vector accesses are cached.

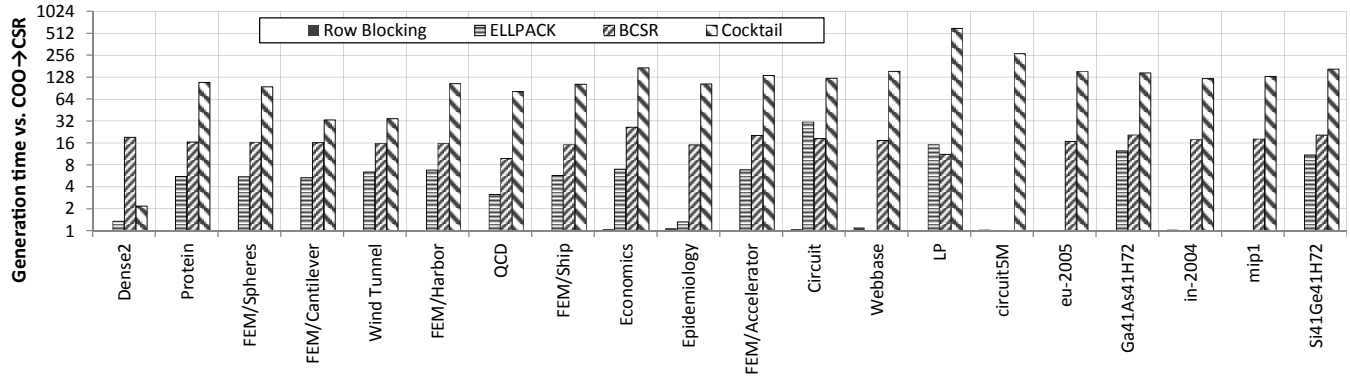

Figure 11. Cost of generating sparse matrix data structures when starting in the COO format. These costs are normalized to converting from COO to CSR. Note that ELLPACK could not be used for `Webbase`, `circuit5m`, `eu-2005`, `in-2004` or `mip1`, so these numbers bars are not shown. Similarly, BCSR could not hold `circuit5M`.

to manually cache elements of the vector. Nickolls et al. demonstrated the benefits of this when the entries in a row are near to each other [21]. Garland further mentions that the scratchpad can be used to hold contiguous chunks of memory (akin to CSR-Stream), but provides no details and does not discuss how to handle long or variable-length rows.

Bell and Garland presented perhaps the best-known work on GPU-based SpMV [3]. They describe CSR-Scalar and CSR-Vector in depth and then discuss the limitations of Garland's previous proposal to use the scratchpad to hold vector values for CSR-Scalar. They come to the conclusion that the best GPU performance depends on the storage format, and they advocate for the column-major ELLPACK or a hybrid of ELLPACK and COO. These conclusions carried forward to the cuSPARSE [20] and Cusp [4] libraries, which both rely heavily on this hybrid format for SpMV.

Numerous works have proposed various other GPU-optimized storage formats, including blocked and sliced versions of CSR, ELLPACK, and COO [16, 23, 28]. This has in turn led to the introduction of multi-format libraries with auto-tuning frameworks to dynamically pick the storage format which yields the best performance [5, 18, 26]. Specialized storage formats provide definitive advantages (e.g., DIA for strongly diagonal matrices). However, as described in Section II-D, the requirement to change from CSR in general presents difficulties.

There is also a selection of work dedicated to accelerating CSR-based SpMV on GPUs. Baskaran and Bordawekar discuss a number of changes to CSR-Vector and show a nearly $2\times$ performance increase by assuring access alignment and by caching values [2]. Some of this performance comes from zero-padding the CSR, which we tried to avoid in order to reduce storage requirements. Reguly and Giles describe an extension to CSR-Scalar that uses a fixed number of threads per row in order to better coalesce the loads [24]. They pick this number statically, which can result in reduced performance compared to CSR-Vector for long rows and underutilized execution units for short ones.

Gropp and Guo showed how to extend CSR-Vector to better work on very long rows [13]. If they find very long rows in a matrix, they statically allocate more than one workgroup per row. At the end, workgroups that share the same row communicate amongst each other to finalize their reduction. This is not a portable technique, as OpenCL™ 1.x

does not guarantee coherence between workgroups. While it may work on some devices, two workgroups within a single kernel are not guaranteed to be able to communicate with one another, even through atomics. Nonetheless, this is a promising technique, and may increase the performance of SpMV on matrices such as `LP`.

Feng et al. show another algorithm that loads more than one row per workgroup, but they require changing the CSR data structure by reordering and sorting the rows [9].

Koza et al. show an algorithm similar to CSR-Stream [15]. They statically allocate a number of rows, called a strip, to each workgroup. Strips are then streamed into the scratchpad before a parallel reduction step. Long rows cannot utilize the GPU's parallelism unless the number of LDS entries per row is greater than or equal to the wavefront width. Otherwise, two parallel threads that both need to write to the same LDS location would need to synchronize to avoid a data race. However, using many LDS entries per row can result in poor performance on matrices with short rows, as this limits the number of active workgroups while also resulting in many wasted loads. These limitations mean their algorithm is often slower than hybrid ELLPACK+COO for matrices with short rows. High-level comparisons with their reported results imply that CSR-Adaptive is much more effective for graphs with short rows like `Webbase` and `Epidemiology`. They must also store NNZ entries containing information about which row within a strip each non-zero value belongs. They hide this in unused upper bits of the values in the `cols` array, but this is not a portable solution.

A recent work that uses a new storage format, yaSpMV, showed very promising performance results on GPUs by using a block-compressed COO [33]. Besides having significant overhead for data structure generation (we found it was higher than the conversions shown in Figure 11), the authors also rely on non-portable communication between workgroups [34]. Their code deadlocks on modern AMD GPUs due to this. One workaround would be to launch a different kernel for the inter-workgroup reduction, but this will negatively affect performance without the use of low-overhead work queues, such as those specified in the Heterogeneous System Architecture (HSA) [17].

VII. Future Research and Conclusions

SpMV is a frequently used linear algebra primitive. CSR is perhaps the most popular format to store sparse matrices. Conventional wisdom says that other storage formats are needed to achieve good SpMV performance on GPUs. In this paper, we introduce a novel CSR-Stream algorithm to optimize SpMV performance for matrices where existing CSR implementations have exhibited poor performance. We also describe CSR-Adaptive, which adaptively switches between using CSR-Stream for rows with relatively few non-zero values and the traditional CSR-Vector for rows with a large number of non-zero values.

Our implementation eliminates both the space and time overheads incurred by SpMV implementations that use other formats, while at the same time, demonstrating comparable or better performance. CSR-Adaptive achieves an average speedup of 14.7× over existing CSR-based SpMV implementations and a speedup of 2.3× over the clSpMV Cocktail, which uses a variety of different formats. It does this while also avoiding the overhead of transforming the sparse matrix to those formats.

The GPU-accelerated CSR-based SpMV described here opens a number of interesting research questions. It is now possible to attain good SpMV performance on both CPUs and GPUs using the same underlying storage format and data structures. This is particularly interesting for heterogeneous processors that have both types of cores [22]. Previously, switching between the CPU and GPU would require a costly data structure change. The removal of this constraint will enable research to properly balance linear algebra workloads across cores for power, performance, and energy.

We also believe that heterogeneous processors will help alleviate some of the concerns raised in the literature about the viability of GPU-based SpMV [7, 29]. GPUs on HSA-complaint processors, for instance, will no longer have the memory capacity constraints or interconnect bandwidth limitations of discrete GPUs, opening up more applications-level research into the best places to use GPU-based SpMV.

Acknowledgements

We thank Jonathan Gallmeier for providing the impetus for this work. Thanks also to Arka Basu, Shuai Che, and the anonymous reviewers for their feedback.

References

[1] *AMD Accelerated Parallel Processing OpenCL™ Programming Guide*, Nov. 2013.

[2] M. M. Baskaran and R. Bordawekar, "Optimizing Sparse Matrix-Vector Multiplication on GPUs," IBM Research, Tech. Rep., 2009.

[3] N. Bell and M. Garland, "Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors," in *Proc. of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[4] ——, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2012, version 0.3.0. [Online]. Available: http://cusp-library.googlecode.com

[5] 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.

[6] 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.

[7] J. D. Davis and E. S. Chung, "SpMV: A Memory-Bound Application on the GPU Stuck Between a Rock and a Hard Place," Microsoft Research, Tech. Rep., 2012.

[8] 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*, vol. 28, no. 2, pp. 239–267, 2002.

[9] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao, "Optimization of Sparse Matrix-Vector Multiplication with Variant CSR on GPUs," in *Proc. of the Int'l Conf. on Parallel and Distributed Systems (ICPADS)*, 2011.

[10] M. Garland, "Sparse Matrix Computations on Many-core GPU's," in *Proc. of Design Automation Conf. (DAC)*, 2008.

[11] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance Graph Algorithms from Parallel Sparse Matrices," in *Proc. of the Int'l Workshop on Applied Parallel Computing*, 2006.

[12] 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.

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

[14] E.-J. Im and K. Yelick, "Optimization of Sparse Matrix Kernels for Data Mining," in *Proc. of the Workshop on Text Mining*, 2001.

[15] 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*, vol. 32, no. 2, pp. C219–C239, 2014.

[16] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A Unified Sparse Matrix Data Format for Modern Processors with Wide SIMD Units," *CoRR*, vol. abs/1307.6209, 2014.

[17] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," HSA Foundation, Tech. Rep., 2012.

[18] 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.

[19] A. Munshi, "The OpenCL Specification," 2012, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

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

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

[22] S. Nussbaum, "AMD "Trinity" APU," in *Hot Chips*, 2012.

[23] T. Oberhuber, A. Suzuki, and J. Vacata, "New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA," *CoRR*, vol. abs/1012.2270, 2010.

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

[25] 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.

[26] 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.

[27] L. N. Trefethen and D. Bau, III, *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[28] F. Vázquez, J.-J. Fernández, and E. M. Garzón, "A New Approach for Sparse Matrix Vector Product on NVIDIA GPUs," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 8, pp. 815–826, 2011.

[29] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the Limits of GPU Acceleration," in *Proc. of the USENIX Conf. on Hot Topics in Parallelism (HotPar)*, 2010.

[30] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proc. SciDAC, J. Physics: Conf. Ser.*, 2005.

[31] R. W. Vuduc, "Automatic Performance Tuning of Sparse Matrix Kernels," Ph.D. dissertation, University of California, Berkeley, 2003.

[32] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," in *Proc. of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2007.

[33] 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.

[34] S. Yan, G. Long, and Y. Zhang, "StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization," in *Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2012.