Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications

Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, P. Sadayappan
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{ashari,sedaghati,eisenlohr,srini,saday}@cse.ohio-state.edu

Abstract—Sparse matrix-vector multiplication (SpMV) is a widely used computational kernel. The most commonly used format for a sparse matrix is CSR (Compressed Sparse Row), but a number of other representations have recently been developed that achieve higher SpMV performance. However, the alternative representations typically impose a significant preprocessing overhead. While a high preprocessing overhead can be amortized for applications requiring many iterative invocations of SpMV that use the same matrix, it is not always feasible—for instance when analyzing large dynamically evolving graphs.

This paper presents ACSR, an adaptive SpMV algorithm that uses the standard CSR format but reduces thread divergence by combining rows into groups (bins) which have a similar number of non-zero elements. Further, for rows in bins that span a wide range of non-zero counts, dynamic parallelism is leveraged. A significant benefit of ACSR over other proposed SpMV approaches is that it works directly with the standard CSR format, and thus avoids significant preprocessing overheads. A CUDA implementation of ACSR is shown to outperform SpMV implementations in the NVIDIA CUSP and cuSPARSE libraries on a set of sparse matrices representing power-law graphs. We also demonstrate the use of ACSR for the analysis of dynamic graphs, where the improvement over extant approaches is even higher.

Keywords—SpMV, GPU, CSR, HYB, ACSR

I. INTRODUCTION

In the last decade, there has been a significant increase in the use of many-core throughput-oriented architectures in scientific computing and data analytics. In particular, with the emergence of GPU APIs such as OpenCL [16] and CUDA [19] [8], scientists and analysts from a broad range of disciplines have begun to leverage GPUs. For instance, Sparse Matrix-Vector Multiplication (SpMV), a core kernel that finds use in a host of scientific [11] and graph analytic [28] domains, has received much attention within the community.

GPUs are very well suited for dense matrix computations, but several challenges are faced in achieving high performance for sparse matrix computations. In the case of SpMV \( y = Ax \), the sparsity and irregularity of the matrix \( A \) raises multiple challenges: a) irregular and non-coalesced accesses to both matrix \( A \) and vector \( x \), b) load imbalance among threads and warps, and c) thread divergence at the warp level.

A number of efforts [1], [4], [5], [18], [27], [28] have sought to address these challenges. The well-maintained high-performance libraries, cuSPARSE [10] and CUSP [4], [5], [9], are two of the most widely-used CUDA libraries and support different sparse matrix formats (e.g. Diagonal (DIA), ELLPACK (ELL), Compressed Sparse Row (CSR), Coordinate (COO), and Hybrid (HYB), which combines ELL and COO). The representation of the sparse matrix affects the performance of SpMV on GPUs, but none of the above formats is consistently superior. The best performing representation depends on the structure of non-zero elements in the matrix.

CSR is the most compact format for unstructured sparse matrices, and is the predominantly used representation for sequential and parallel computing on CPUs. It is the main format employed in widely used scientific libraries such as PETSc [2], [3] and Hypre [12]. However, most efforts at efficient SpMV for GPUs [1], [4], [5], [18], [27], [28] have sought the use of alternate representations for sparse matrices in order to address one or more of the above challenges. Although alternate formats may improve the performance of SpMV on GPUs, they generally impose two overheads: 1) a higher space requirement than CSR, and 2) higher set-up time than CSR, for creation of the sparse matrix in the alternate format.

The preprocessing time for alternate formats can be very significant, being one or two orders of magnitude higher than the time for actually performing an SpMV. A high set-up cost is acceptable in application contexts where it can be amortized over repeated SpMVs using the same sparse matrix or a sequence of sparse matrices with identical sparsity structure. However, a very high preprocessing overhead is unacceptable in scenarios where the structure of the sparse matrix changes quite frequently. Such is often the case for adaptive graph applications, where a graph is represented as an adjacency matrix, and many common operations on graph data structures are expressed using sparse-matrix operations [15].

In this paper, we focus on efficient SpMV for dynamic graph applications, where the sparsity structure of the matrix changes frequently. Instead of using an alternative format to CSR, we devise an approach that builds on top of the standard CSR, using additional meta-data whose creation imposes low overhead in terms of both space and time. The new adaptive SpMV method built on CSR, called ACSR, addresses the thread divergence, while avoiding redundant computation and data transfer needed by alternative formats that use padding. It also avoids the excessive synchronization overhead due to reduction/atomic operations with formats like COO.

ACSR groups rows of the sparse matrix into different bins, based on the number of non-zero elements in the rows, and launches bin-specific SpMV kernels. For bins containing rows with a large number of non-zero elements, a special master kernel is invoked, whose purpose is to decrease thread
store the row indices of the non-zero elements. Instead, the non-zero elements in each row are stored contiguously in a vector, along with a vector that maintains the offset for the first of each row’s elements in the vector. The CSR representation of $A$ is shown in Figure 1(a). Non-zero elements of row $i$ and the corresponding column indices are located respectively in the data and column index vectors at index $r : \text{RowOffset}[i] \leq r < \text{RowOffset}[i+1]$. The ELLPACK (ELL) [23] representation is similar to CSR, but each row is zero-padded so that all rows occupy the same size as the one with the largest number of non-zero elements.

To perform SpMV on a matrix in COO format, each thread computes the product of a non-zero element of the matrix and the corresponding element in vector $x$. To generate the final dot product of a row of the matrix by the vector, a reduction step accumulates the partial products. This operation requires atomic access to the result element and causes serialization that imposes a performance penalty. The overhead is alleviated to some extent by use of efficient segmented reduction [5], where threads in a warp span multiple rows and each thread handles multiple non-zero elements of a row.

With the CSR format, one approach (Scalar-CSR) is to make each thread responsible for calculating the product of a matrix row with the vector $x$. But this kernel suffers from thread divergence and uncoalesced access to the sparse matrix elements. An alternate approach (Vector-CSR [5]) improves performance by having all threads of a warp collectively process a row. But this algorithm wastes resources when rows have far fewer non-zero elements than a warp size. In the most recent and efficient implementation of Vector-CSR in the cuSPARSE [10] and CUSP [9] libraries, the idea of segmented reduction is employed, where threads of a warp span multiple rows when the average number of non-zeros per row is small.

Hybrid COO-ELL (HYB) [5] is a hybrid format that uses a combination of COO and ELL. The matrix representation has two distinct parts: the ELL part and the COO part. The ELL part is used for rows whose non-zero count is less than or equal to some chosen threshold $k$. The ELL part is represented as a full dense $\text{row} \times k$ matrix. Rows with fewer than $k$ non-zero elements are zero-padded. If a row has more than $k$ non-zeros, the remaining elements are placed in the COO part. A heuristic is used for selecting $k$ as the maximum value such that there are at least $R = \max(4096, \frac{4}{k})$ rows with $k$ or more non-zero elements, in a matrix with $M$ non-zero elements. Figure 1(b) shows the HYB format for matrix $A$, with $k = 2$.
III. ADAPTIVE CSR (ACSR) FORMAT

The main challenge with CSR-based approaches for SpMV is that a fixed number of threads are assigned to each row. This works well for matrices with low standard deviation in the number of non-zeros per row. However, execution is unbalanced for matrices with high deviation, where warps working on rows with few non-zeros finish much earlier compared with those that are assigned to rows with many non-zeros. For load-balanced execution with matrices with high deviation (i.e., highly varying sparsity across different rows), different number of threads should work on different rows. In other words, careful consideration of the matrix sparsity is required when distributing work (i.e., rows) across different threads. This is especially the case for power-law matrices that constitute a very important class of real-world matrices. They have a small number of rows with a huge number of non-zeros (the long tail in Figure 3). Such wide rows have proven to be the bottleneck when the SpMV kernel assigns fixed number of threads to each row.

In order to address the load imbalance issue for such matrices, we develop Adaptive CSR (ACSR) – an adaptive scheme that is based on the CSR matrix representation but considers the sparsity of the rows of the matrix when choosing the number of threads. Because it is based on the CSR format, ACSR imposes very low preprocessing overhead (meta-data generation, data transformation, etc.). ACSR combines two separate mechanisms: row binning and dynamic parallelism. It uses row-binning to place matrix rows into bins so that rows within any bin have roughly the same number of non-zeros (within a factor of two). This grouping of rows into bins enables processing by bin-specific kernel variants that reduce thread divergence. The binning also helps isolate the bottleneck long-tail rows to be processed by a different mechanism that exploits dynamic parallelism capabilities of modern GPUs, where kernels can be dynamically launched from within an executing kernel. Dynamic parallelism is supported by recent NVIDIA GPUs (i.e. Kepler) with device compute capability of 3.5 or higher.

Algorithm 1: ACSR, the Driver

| input | CSR matrix A: (values, col_idx, row_off), Vector: x, Vector: y, Vector: BIN#N_Rows |
| output | y, BinMax, RowMax, ThreadLoad |
| begin | if (this is the first iteration) then foreach row r do b_r ← b_r ∪ {r}; 2^{i-1} ≤ nnz(r) ≤ 2^i; n ← max(i): size(b_i) > 0; G_1 ← {b_0, b_1, ..., b_k}; i = 0; k ≥ BinMax; G_2 ← {b_1, b_2, ..., b_k}; do SpMV << gridG_r, >> (b_r); // Launch a bin-specific kernel: Algorithm 2 DP_Parent << gridG_r, >> (G_1); // Launch the parent kernel for rows in G_1: Algorithm 3 |

A. ACSR: Binning

The standard vector-CSR algorithm involves the cooperative handling of each row by a set of threads in a warp. The size of a thread-group to handle each row is set to be a perfect power of two close to μ, the average number of non-zeros per row in the matrix. For rows that have fewer non-zeros than the thread-group size, load imbalance exists during execution. With ACSR, this load imbalance problem is addressed by grouping matrix rows into bins as follows: Bin_1 contains all rows with 1 or 2 non-zeros, Bin_2 all rows with 3 or 4 non-zeros, Bin_3 the rows with number of non-zeros in the range 5-8, and so on. Bin-specific kernels are used for rows in each bin, with the size of thread-group cooperating on each row in a bin being customized for that bin size.

The row-binning idea is illustrated by Figure 2. In the example, the sparse matrix is preprocessed to place the rows into three different bins (shown in Figure 2-b). The preprocessing step is done once by scanning the matrix to place each row in the appropriate bin. It is very inexpensive and does not require any movement and restructuring of the matrix data.

Algorithm 2: ACSR, Bin-Specific SpMV for Bin-N

| input | CSR matrix A: (values, col_idx, row_off), Vector: x, Vector: y, Vector: BIN#N_Rows |
| output | Vector y |
| begin | $2^{N-1}$ (N = 1, 2, ..., N) threads work on each row (vector size) for Bin N begin tid ← thread index; lid ← tid % 32; // lane index vid ← tid / 32; // vector (warp) index sum ← 0; row ← BIN#N_Rows[tid]; row_idx ← row_off[row]; next_row_idx ← row_off[row + 1]; foreach (i = row_idx + vid, i < next_row_idx, i = 2^{N-1}) do sum += values[i] × column_indices[i]; // Intra vector reduction (example using shuffle) for (i = N, i > 0, i =) do sum += shfl_down(sum, i); after Reduction, the first thread has the final result. if (vid == 0) then y[row] += sum; |

After extracting the bins, a separate bin-specific kernel is launched for each bin to perform SpMV for the rows assigned to that bin. For example, three separate kernels (BIN1, BIN2, BIN3) are invoked for the three bins that are found for the example sparse matrix (shown in Figure 2-b). Consequently, since rows 2, 5 and 7 are assigned to the second bin, they are processed by the BIN2 kernel. However, it is noteworthy that, as described in Algorithm 1, bins are partitioned into two separate groups: bins whose rows have large average number of non-zeros (G1), and those whose rows have small average non-zeros (G2). The partitioning is done to enable effective utilization of dynamic parallelism, as described in the next subsection. The number of rows in G1 is limited by the parameter RowMax, to stay within the limits on the maximum number of simultaneous dynamically launched kernels supported by the GPU. If dynamic parallelism is not enabled (i.e., binning-only) or not supported on the target device (e.g., NVIDIA Fermi), group G2 will contain all the bins (i.e., RowMax = 0).

Each bin has a unique pre-defined range for the number of non-zero elements and that range is used at the time of preprocessing. Each bin-specific kernel (and thus the rows that are mapped to that bin) also has a pre-defined dimensionality.
(i.e., number of threads in a group that cooperatively processes a row). For example, if a bin contains rows with number of non-zeros in the range of $[1..2]$ (first bin in Figure 2-b), then the assigned kernel (BIN1) utilizes only one thread to work on each row. Similarly, if a bin contains rows with number of non-zeros in the range of $[33..64]$, then the number of cooperating threads per row is 32. Generally, bin $i$ covers the range of $[2^{i-1}+1..2^i]$, for $i > 0$. For simplicity (and consistency with CSR-Vector [5]), the bin ranges have been chosen to be powers of two.

The generated kernels require a final reduction step where the threads that have worked on the same row accumulate the computed partial results, as with CSR-Vector [5]. The reduction operation is intra-warp and does not require inter-warp synchronizations (i.e., can be done by a shuffle instruction on NVIDIA Kepler GPUs). Pseudocode for the bin-specific kernel for bin “N” is shown in Algorithm 2.

Algorithm 3: ACSR, Dynamic Parallelism Parent Kernel

```
input : CSR matrix A: (values, col_idx, row_off), Vector: x, Vector: y, Vector: G_i_Row
output: Vector: y

begin
    tid ← the thread global ID in the grid;
    if tid < size(G_i_Row) then
        row ← G_i_Row[tid];
        row_idx ← row_off[row];
        next_row_idx ← row_off[row + 1];
        NNZ ← next_row_idx – row_idx;
        bSize ← NNZ/ThreadLoad;
        str ← a CUDA stream;
        SpMV <<< 1, bSize, str >>> (argv) where:
            // SpMV is the row specific kernel; Algorithm 4
        begin
            argv[0] : values[row_idx..next_row_idx);
            argv[1] : col_idx[row_idx..next_row_idx);
            argv[3] : x;
        end
```

B. ACSR: Dynamic Parallelism

Dynamic parallelism is a new feature in NVIDIA GPUs (compute capability 3.5 or higher) that enables threads on the device to launch a new grid without having to return to the host. The control mechanism between the thread and the new grid is a parent-and-child relationship. Figure 2-c shows an example of dynamic parallelism in which the host launches a grid on the device (the parent grid) and each thread in this grid launches a new grid with an arbitrary size (i.e. ROW0 to ROW7, each working on a different row of the example matrix). The parent-child thread block relationship ensures that the parent thread block does not exit before all children threads finish.

We leverage dynamic parallelism to further improve the adaptivity of ACSR. In order to have an efficient dynamic parallelism, it is not beneficial to launch a parent kernel (plus the new child kernels) for the bins with small average number of non-zeros per row (group G2 in Algorithm 1). Doing so will not create enough compute work for the new child kernels (i.e. due to rows having few non-zeros each). Thus, after creating the G1 and G2 groups (as described in Algorithm 1), and for every bin inside G2, a separate SpMV grid is launched. This bin-specific grid assigns a constant number of threads to each
row in the bin. For the remaining bins (group G2), a single parent kernel is launched from the host (shown in Figure 2-d).

![Fig. 3: Power Law distribution (histogram).](image)

Each thread inside the parent grid works as a parent thread itself and launches a row specific grid of worker threads that do the actual work. Algorithm 3 describes the required steps and how a parent kernel creates and launches worker grids. Algorithm 4 shows the structure of the worker (child) SpMV kernel working on a row. As described by the algorithm, multiple thread groups (i.e., warps) coordinate to perform SpMV on a single row. Consequently, in addition to the intra-warp reduction (seen previously for the bin-specific kernels), a final inter-warp reduction is needed at the end.

The size of each worker grid is determined by the number of non-zeros of the corresponding row and a pre-defined parameter that specifies compute load per thread (i.e., \textit{ThreadLoad}). This is considered as the \textit{thread coarsening} knob in our algorithm. Note that parent threads are only used for control purposes and do not perform any actual computations. Also note that the bin partitioning in Algorithm 1 (into G1 and G2) is done using two defined parameters: \textit{BinMax}, the largest bin index for which we launch a bin specific grid; and \textit{RowMax}, the largest number of rows for which we launch a row specific grid. As an example, Figure 2-d demonstrates a case where \textit{BinMax} = 2, \textit{RowMax} = 16, and there are total of two rows mapped to the third bin (i.e., representing group G1). For this case, bin-specific SpMV kernels are launched for bins \textit{BIN}1 and \textit{BIN}2 (that belong to G2) and a parent grid for all rows in bin \textit{BIN}3.

For the NVIDIA GPUs with dynamic parallelism enabled, there is a limit on the number of dynamic kernel invocations; \textit{cudaLimitDevRuntimeP endingLaunchCount} : 2048. We account for such a device-specific limit in the binning mechanism (in Algorithm 1, \textit{RowMax} is set to this limit), to ensure that the size of \textit{G1} (i.e., total number of threads) does not exceed this limit. Failing to meet this constraint (e.g., launching more than 2048 concurrent dynamic kernels) will force the system to reserve memory for pending launches and thus lead to performance degradation.

The binning and dynamic parallelism techniques make ACSR most effective on large matrices with a diverse range of non-zero elements in rows. Many real-world matrices (e.g., adjacency graphs in social networks and web communication, circuit simulation, etc.) exhibit the power-law property. Figure 3 shows the distribution (histogram) of power law matrices – the \(x\) axis shows the number of non zero elements per row and the \(y\) axis shows the probability (frequency) distribution of such rows. As shown in the figure, there is a long tail on the right side of the distribution. These very long rows are a bottleneck when performing SpMV with standard CSR. The figure also shows that there is a very heavy concentration of very small rows. The ACSR algorithm has been designed to handle both these extremes. The bins corresponding to the smaller rows are handled by the bin-specific kernels, while the long rows are effectively processed by dynamic kernel launching.

### IV. Evaluation Methodology

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Abbrev.</th>
<th>NNZ</th>
<th>Rows / Cols</th>
<th>(\mu) / (\sigma) / Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon-2008</td>
<td>AMZ</td>
<td>5M</td>
<td>735K</td>
<td>7.7 / 4.7 / 10</td>
</tr>
<tr>
<td>cee-2000</td>
<td>CEE</td>
<td>6M</td>
<td>845K</td>
<td>10.2 / 7.8 / 2216</td>
</tr>
<tr>
<td>dblp-2010</td>
<td>DBLP</td>
<td>1.5M</td>
<td>320K</td>
<td>5.8 / 5.3 / 238</td>
</tr>
<tr>
<td>enron</td>
<td>ENR</td>
<td>276K</td>
<td>69K</td>
<td>4.7 / 28 / 1392</td>
</tr>
<tr>
<td>eu-2005</td>
<td>EU</td>
<td>19M</td>
<td>862K</td>
<td>22.7 / 29 / 6985</td>
</tr>
<tr>
<td>flickr</td>
<td>FL</td>
<td>22M</td>
<td>1.8M</td>
<td>12 / 101 / 2615</td>
</tr>
<tr>
<td>hollywood-2009</td>
<td>HOL</td>
<td>113M</td>
<td>1M</td>
<td>100 / 272 / 11468</td>
</tr>
<tr>
<td>in-2004</td>
<td>IN</td>
<td>16M</td>
<td>1M</td>
<td>12 / 101 / 753</td>
</tr>
<tr>
<td>indochina-2004</td>
<td>IND</td>
<td>194M</td>
<td>78M</td>
<td>26 / 216 / 6985</td>
</tr>
<tr>
<td>internet</td>
<td>INT</td>
<td>104K</td>
<td>65K</td>
<td>2.7 / 24 / 693</td>
</tr>
<tr>
<td>livejournal</td>
<td>LV</td>
<td>77M</td>
<td>5M</td>
<td>13 / 22 / 9186</td>
</tr>
<tr>
<td>ljournalex-2008</td>
<td>LJ</td>
<td>79M</td>
<td>5M</td>
<td>15 / 37 / 2469</td>
</tr>
<tr>
<td>ok-2002</td>
<td>OK</td>
<td>298M</td>
<td>18M</td>
<td>16 / 27 / 2450</td>
</tr>
<tr>
<td>wikipedia</td>
<td>Wik</td>
<td>20M</td>
<td>1.3M</td>
<td>31 / 42 / 20975</td>
</tr>
<tr>
<td>youtube</td>
<td>YO</td>
<td>54M</td>
<td>1M</td>
<td>4.7 / 48 / 2894</td>
</tr>
<tr>
<td>webbase-1M</td>
<td>WEB</td>
<td>3M</td>
<td>1M</td>
<td>37 / 25 / 4700</td>
</tr>
<tr>
<td>raid4284</td>
<td>RAL</td>
<td>11M</td>
<td>4K / 1M</td>
<td>2833 / 2409 / 56181</td>
</tr>
</tbody>
</table>

TABLE I: Matrices used in this study; NNZ: total number of non-zeros, \(\mu\): average of non-zeros per row, \(\sigma\): standard deviation, Max: maximum number of non-zeros in rows. \textit{RAL} is a rectangular matrix (i.e., not power-law).

The effectiveness of the ACSR algorithm was evaluated on a collection power-law matrices, including those used in previous studies [5], [6], [26], [28]. The matrices are all available from the University of Florida Sparse Matrix Collection [22]. Table I shows characteristics of each matrix: total number of non-zeros, number of rows/ columns, mean \((\mu)\), standard deviation \((\sigma)\), and maximum number of non-zeros per row. With the exception of AMZ and DBL, the maximal non-zero per row in these matrices can be seen to be much larger than the mean, and the standard deviation is also larger than the mean. AMZ and DBL do not follow the same trend as the others, and were selected to contrast ACSR performance with non-power-law matrices.

We report performance in terms of computation rate (as number of floating point operations per second, or GFLOPs). Each SpMV experiment was repeated 50 times and the average (arithmetic mean) is reported. The experiments were run on three different generations of NVIDIA GPUs, \textit{GTX580} (Fermi G104), \textit{TeslaK10} (Kepler GK104s) and \textit{GTXTitan} (Kepler GK110), each hosted by an Intel Core i7 CPU. Details about the GPUs are provided in Table II. As dynamic parallelism is available only on GPUs with compute capability \(\geq 3.5\) (only available on the \textit{GTXTitan}), ACSR with dynamic parallelism is only reported for the \textit{Titan} GPU. For the others, we only report ACSR with binning. Since the \textit{TeslaK10} has two GPUs, we also report dual-GPU performance in Section VIII.

We used the NVIDIA compiler (\textit{nvcc}) version 5.5, with optimizations enabled using -O3. The input vector \(x\) was placed in texture memory, which in general improves memory access (i.e., by utilizing the \textit{Texture} cache on the device). Such a strategy is also employed by the SpMV implementations in cuSPARSE [10] and CUSP [9].
We compare different algorithms, reporting both preprocessing time and SpMV time. This enables an assessment of the number of SpMV invocations required for other formats to outperform ACSR. Table III shows speed-up of ACSR compared to BCCOO [27], BRC [1], TCOO [28], and HYB [5], when performing a single SpMV. The speed-ups are generally very high, due to the much higher preprocessing time of other schemes. Figure 4 shows the ratio of preprocessing overhead to the time for one SpMV operation. In case of BRC, TCOO and HYB the preprocessing overhead is the time for data transformation, while for BCCOO it is the time for auto-tuning. For the latter format we compare against the best performance achieved for all matrices. Every matrix we tested achieve its best performance with different settings and this configuration space has more than 300 different settings. Also, in the case of TCOO, we performed an exhaustive search to find the best number of tiles, the input parameter to the algorithm, and time reported here is for the best performance of TCOO. As Figure 4 shows, in the studied formats, preprocessing time is a dominant factor in the overall SpMV time, while for ACSR it is in the order of few SpMVs. On average, the ratio of preprocessing time to SpMV time, is 161k, 87, 3k, 21, and 3, for BCCOO [27], BRC [1], TCOO [28], HYB [5], and ACSR respectively.

In Table IV, we show the actual SpMV time, along with how many iterations, \( n \), are needed for other formats to outperform ACSR. To calculate this number, first we model the total time for repeated SpMVs, for example in an iterative solver as:

\[
T = PT + n \times ST \tag{2}
\]

where \( ST \) is the execution time of one SpMV calculation and \( PT \) is the preprocessing time. For an algorithm \( A \) to outperform ACSR, we must have:

\[
PT_A + n \times ST_A \leq PT_{ACSR} + n \times ST_{ACSR} \tag{3}
\]

Simplifying this inequality, we have:

\[
n \geq \frac{PT_A - PT_{ACSR}}{ST_{ACSR} - ST_A} \tag{4}
\]

A \( \emptyset \) cell in the table indicates that the format is not able to handle the matrix due to memory limitation. Also, an \( \infty \) entry means that ACSR outperforms the other format for any number of iterations. As the table shows, BCCOO and TCOO outperform ACSR when we use SpMV in a solver that iterates many times. The same is true for BRC, but with fewer iterations. ACSR outperforms HYB, except for matrix AMZ, DBL, and WIK. For AMZ and DBL, due to the small maximum number of non-zeros per row, ACSR does not leverage dynamic parallelism. Later in this section, we show more results comparing ACSR against HYB. We do not have the data on how much padding cost is paid by BCCOO. However, for the matrices we have tested in HYB format, it is 33% of the total number of non-zeros on average. For BRC and TCOO, the space overhead is around 1% and 5%, respectively. ACSR uses the same amount of memory as CSR except for the bin data. Since BCCOO and TCOO are only available for single precision, data in Figure 4 and Tables III and IV are only for single precision computation. The experiments were performed on a GTX Titan GPU.

In the case of power-law matrices, in general the HYB format is the best among all the formats in the NVIDIA SpMV library [28]. Therefore we compare ACSR against HYB on different devices. We also compare ACSR against CSR, the
ACSRR is up to 1.67× and 1.75× (and average 1.18×) faster than the HYB format in single and double precision, respectively. This performance improvement of ACSR is obtained without the average 33% padding cost and transformation time of HYB. The HYB transformation costs 21× SpMV on average while for ACSR it is less than 3 SpMV. Furthermore, due to memory limits and padding applied by HYB, there are matrices which cannot be transformed into the HYB format. ACSR also achieves a maximal speedup of 5.34× and 6.02× (and average 2.09× and 2.34×) over straightforward SpMV for CSR for single and double precision, respectively. Table V shows how many bin specific grids (BS column) and row specific grids (RS column) are launched by ACSR for different matrices.

The same experiment was performed on a GTX580 GPU. However, as the compute capability of this GPU is 2.0 and does not support dynamic parallelism, ACSR only leverages binning (i.e., \( \text{RowMax} = 0 \)) on this GPU. Figure 5-center shows results on this GPU. Results on this device are similar to the same results on the GTX Titan with the difference that there are large matrices, such as HOL and UK2, which could not be run because of the memory limit of GTX580. Such cases are shown by Ø (no bar). There is some loss of performance due to the inability to apply dynamic parallelism. On this device, ACSR is up to 1.45× and 1.32× (and average 1.1× and 1.05×) faster than the HYB format in single and double precision, respectively. It also achieves a maximal speedup of 6.3× and 5.65× (and average 2.23× and 2×) over the CSR format for single and double precision, respectively. Figure 5-bottom shows GFLOPs obtained for the same experiment on a Tesla K10 with a single GPU.
VI. ACSR in Graph Mining Application

Algorithm 5: PageRank Algorithm

\begin{verbatim}
input: Row normalized adjacency matrix $A^{(n \times n)}$, damping factor $d$, convergence rate $\epsilon$
output: Vector $PR^{(n)}$
begin
    $PR^{(0)}(i) \leftarrow \frac{1}{n}, \forall i \in \{1..n\}$;
    $k \leftarrow 0$;
    repeat
        $PR^{(k+1)} \leftarrow (1-d)PR^{(0)} + d(A^T \times PR^{(k)})$;
    until distance($PR^{(k+1)}, PR^{(k)}$) > $\epsilon$;
end
\end{verbatim}

In order to illustrate the practical use of the developments reported in this paper, we evaluated ACSR with three well known graph algorithms: PageRank [20], HITS [17] and Random Walk with Restart (RWR) [24]. These algorithms apply the power method to the adjacency matrices of large graphs. Such matrices generally exhibit the power law property. The total run time of these algorithms is dominated by the SpMV calculation that is executed during each iteration until the algorithm converges.

A. PageRank

PageRank is an iterative algorithm that assigns an initial rank to pages and updates it based on the links between pages until it converges. Algorithm 5 shows pseudocode for the PageRank method. The function distance can be any distance measure; we use Euclidean distance, the most common distance measure. We evaluated performance using different SpMV methods: CSR, HYB, ACSR. We used a damping factor $d = 0.85$ [20]. Figure 6-top shows the speedup of ACSR over CSR and HYB for applying PageRank on power law matrices. It may be seen that ACSR outperforms both CSR and HYB on all matrices, except AMZ. As discussed earlier, AMZ’s structure is better suited to use of the HYB format. For each matrix, the number of iterations for convergence (i.e., the number of SpMV operations performed) is also shown.

B. HITS

HITS is a link analysis method that assigns two scores, authority and hub, to each web page. If $A$ is the adjacency matrix of a graph $G = (V, E)$, where $V$ represents the set of web pages, and $E$ the set of links between web pages, the authority ($a$) and hub ($h$) score vectors are calculated as follows:

\begin{equation}
a^{k+1} = A^T \times h^k
\end{equation}

\begin{equation}
h^{k+1} = A \times a^k
\end{equation}

Here $k$ is the iteration index, and $a$ and $h$ are initialized with equal score of $\frac{1}{|V|}$ for all pages. As in [28], we combine the above computations into a single SpMV as follows:

\begin{equation}
\begin{bmatrix} a \\ h \end{bmatrix}^{k+1} = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} \times \begin{bmatrix} a \\ h \end{bmatrix}^{k}
\end{equation}

Like PageRank, the algorithm iterates until both the $a$ and $h$ score vectors converge. We implemented Equation 7 using CSR, HYB and ACSR as the SpMV method and results are shown in Figure 6-center.
C. Random Walk with Restart

Random Walk with Restart (RWR) [24] is another graph algorithm that calculates the relevance vector $r_i$ between node $i$ and all other nodes in the graph. Vector $r_i$ is defined by:

$$r_i^{k+1} = c(W \times r_i^k) + (1 - c)e_i$$  \hspace{1cm} (8)

Here $W$ is the column-normalized version of the adjacency matrix of the underlying graph, $c$ is a restart probability (similar to damping factor in PageRank), and $e_i$ is a column vector with $e_i[i] = 1$ and $e_i[j] = 0; \forall j \neq i$. RWR is also an iterative method, and its total time is dominated by SpMV. Figure 6-bottom shows speedup in RWR algorithm using ACSR over using CSR and HYB on power law matrices.

For all the algorithms, Euclidean distance was used as the convergence measure, with $\epsilon = 10^{-6}$. In recording the time, the time for copying data to the device was not included. HYB data transformation cost was also not included. The average speedups are shown in the rightmost set of bars of Figure 6 (AVG) for the three applications.

VII. ACSR WITH DYNAMIC GRAPHS

In many applications such as web mining and analytics on social communication networks, the graph that defines the network is subject to changes over time. Therefore, the sparse adjacency matrix that encodes the graph changes accordingly. Algorithms like page ranking receive the updated matrix and iterate on the new matrix to update their rank/score. Compared to a “cold start” with a graph, where an initial guess of the iterated solution vector must be used, as the page connectivity graph evolves, the previous page rank vector can be used as the initial guess for the updated matrix. Doing so reduces the number of iterative steps to convergence. Thus the number of SpMV operations per page rank calculation in a dynamic setting is lower than that in a static setting, often just tens of iterations. In these scenarios, SpMV formats that require high preprocessing costs are not suitable for use. Since ACSR does not incur a high preprocessing cost and works directly on top of CSR data, it is very attractive for such applications.

Updates to dynamic graphs can be transmitted in two ways: 1) the complete updated matrix, or 2) just the changes to the previous matrix. In the first case, SpMV algorithms which transform the matrix need to redo this step each time the matrix is updated. For this model, due to the very low preprocessing overhead, as shown in the previous section, ACSR outperforms other algorithms. In the second case, other formats must first apply the change to the matrix, and then perform the transformation. This also incurs the cost of copying the updated matrix to the device again. For the dynamic update scenario, we develop an approach that works on top of CSR format and copies only update information to the device. Therefore it reduces the updating cost as well.

For the incremental approach, some additional memory is reserved at the end of each CSR row, to be used when nonzeros get added to the row. A matrix update is defined by specifying the rows to be updated, and for each row, which
columns are to be added or deleted. This information is copied to the device and a device kernel applies the changes to the matrix. We have implemented this kernel, with one thread being responsible for updating a row. To avoid intra-warp thread divergence, we assign a warp to each row, but only the first thread of the warp performs the update. This thread first deletes columns of the delete list from the row, and compresses (shifts columns) the row to fill up the deleted spaces. Then, it extends the row by adding columns from the insert list. The kernel assumes that the delete and insert column lists are sorted.

To test the effectiveness of this kernel, we synthetically generated changes on large power law matrices. We randomly selected 10% of the rows to be updated. Scanning the columns of a row, we either remove a column or add another column to the row, each with equal probability. The total number of non-zeros in the matrix is thus kept nearly constant. We encode the changes into an array of rows to be updated, a list of columns to be deleted and a list of columns to be added, both in CSR format. This data is copied to the device and the matrix is updated accordingly.

In this experiment, PageRank is run till it converges in iteration \( k \). Then the matrix is changed by 10% as described above, and PageRank is applied to the new matrix, but starting off with \( PR^{k+1} \) set to the converged \( PR^k \) from the previous time epoch. This is repeated 10 times. With ACSR, after the first time period, instead of copying the complete updated matrix to device, only the change list is copied and the update to the matrix is performed on the device. In contrast, for CSR and HYB, each time that matrix is changed, the complete updated matrix needs to be copied to device. With the HYB format, there is also an additional cost for re-transforming the matrix. Figure 7-top shows the speedup for matrix FLI, which is a representative of our test set. The different sets of bars show the speedup trend over the time epochs. The speed-up in all later time periods is larger than the first time period, because the cost of copying the complete matrix for ACSR is only paid in the first time period. It may also be observed that fewer iterations are needed for convergence in later steps, making the preprocessing and copying overheads more significant.

The same experiment was performed on all the other matrices. Figure 7-bottom shows the average speed-up across all time points, corresponding to the right-most set of bars for FLI in Figure 7-top. Comparing speed-up in this figure with Figure 6-top, it can be seen that the performance improvement from use of ACSR with PageRank on dynamic graphs is more significant than with static graphs.

VIII. DIVIDING WORK AMONG MULTIPLEGPUS

In this section, we evaluate multi-GPU execution of sparse matrix-vector multiplication using the ACSR algorithm. We report performance on a dual-GPU NVIDIA Tesla K10, with two GPU devices on the same card. In order to leverage dynamic parallelism, we require GPUs with compute capability \( \geq 3.5 \); but Tesla K10 has compute capability 3.0. Therefore we were constrained to evaluating only the row-binning benefits of ACSR but not the leveraging of dynamic parallelism. By extending the number of bins in the long tail, we can simulate the behavior of ACSR with static/hard-coded parallelism. The partitioning algorithm for ACSR is a simple division of each bin among GPUs. For two GPUs, we simply map half of the rows in each bin to each device. Each GPU is thus responsible for computing half the results. Such a partitioning approach can be used with any number of GPUs.

Performance results for single and double precision are shown in Figure 8. This approach on average achieves 1.64X and 1.68X improvement over ACSR on a single GPU, for single and double precision respectively. As the figure shows, there are cases in which ACSR scaled perfectly, e.g. \( AMZ, EU2, HOL, IN2, \) and \( LJ2 \). However, there are cases for which using two GPUs does not double the performance of using a single GPU. In order to achieve perfect scaling using multiple GPUs, first there must be enough workload (both computation and memory access) to saturate all GPUs. Then the division algorithm must apportion such tasks equally among GPUs. Our partitioning algorithms respect the second requirement, but in cases where there is not sufficient workload, perfect scaling cannot be achieved. The matrices \( ENR, FLI, INT, \) and \( YOT \) are examples with insufficient workload on each GPU to saturate available resources and hide data latency. In fact, for matrices \( ENR \) and \( INT \) the workload is smaller than required to saturate a single GPU of the Tesla K10.
In these cases, using multi-GPU not only does not improve performance, but adds the overhead of synchronizing two GPUs. Excluding these matrices, partitioning across two GPUs on average results in \( \frac{1}{1.79} \) and \( \frac{1}{1.80} \) improvement over ACSR on a single GPU, for single and double precision respectively.

IX. RELATED WORK

The numerical linear algebra community has developed many data storage formats and algorithms for performing the SpMV operation. The different formats often represent trade-offs between initial preprocessing overhead and performance of repeated SpMV. Further, different representations may be preferable for matrices from different application domains because of differing sparsity characteristics. In this section, we briefly discuss a number of approaches that have been developed for efficient execution on GPUs.

CSR and COO are the most commonly used formats. However other formats achieve higher performance in some scenarios. For example, Bell and Garland [5] demonstrate that DIA is the superior format for structural matrices which have non-zeros on only a few diagonals. For blocked diagonal matrices, such as those which arise in structured grid computations with many degrees of freedom, Godwin et al. [14] show that CDS outperforms DIA. If the sparse matrix is symmetric, PKT [4] is the best format, but it is not optimized for SpMV. In the case of general blocked sparse matrices, formats such as BCSR and BELLPACK [7] are superior. For unstructured matrices, formats such as ELL, HYB [5], BRC [1], BCCOO [27] and TCOO [28] have been shown to be effective. However, many of these formats require very significant preprocessing time.

As with ACSR, Feng et al. [13] also group rows based on non-zero count. They propose the use of Compressed Sparse Row with Segmented Interleave Combination (SIC), to put rows into 3 segments and combine data in each segment by interleaving rows into blocks. Since their implementation was not available, it was not feasible to perform an experimental performance comparison with ACSR. However, as is the case with several other schemes experimentally evaluated in this paper (BCCOO [27], BRC [1], TCOO [28]), the approach of Feng et al. [13] also requires expensive preprocessing operations such as sorting and re-formatting. A critical difference between ACSR and most proposed SpMV approaches is that it uses the standard CSR representation and thus incurs minimal preprocessing overheads. A low preprocessing overhead is very important for dynamic graph applications where the graph dynamically evolves over time and changes the sparse matrix structure, so that the number of SpMV applications with a fixed matrix sparsity structure is relatively low. The performance results demonstrate that ACSR is preferable to previous SpMV approaches for such applications.

X. CONCLUSION

This paper describes ACSR, a fast adaptive approach to sparse matrix-vector multiplication on GPUs, well suited for graph processing applications. A significant benefit of the proposed algorithm is that it works with data in CSR format, the most memory efficient and commonly used format for general sparse matrices. ACSR does not require any data transformation, and its preprocessing is limited to efficient scanning of row-lengths to determine the workload of different bins. The ACSR SpMV implementation was demonstrated to outperform state of the art implementations from the NVIDIA CUSP and cuSPARSE libraries, using both the base CSR format and the HYB format.

ACSR is very well suited for processing of large power-law matrices, commonly encountered with adjacency graphs in social networks and web communication. For such matrices, the cost of reformatting the matrix, and memory padding cost are often relatively high. Further, applications which process such matrices often have to deal with sparsity structure that is dynamically changing at a slow rate. ACSR is the especially advantageous for such contexts, since such adaptations can be easily incorporated incrementally with a very low overhead. The benefits of ACSR were demonstrated using several data analytics applications that repeatedly use sparse matrix-vector multiplication.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. We also thank Xintian Yang [28], and Shengen Yan [27] for providing us with the source codes of their implementations. This work was supported in part by the National Science Foundation through awards ACI-0904549, CCF-1217353, IIS-1111118, and CCF-1240651.
REFERENCES


[22] Research and software development in sparse matrix algorithms. The university of florida sparse matrix collection.


