Efficient SpMM Using Dynamic-CSR on GPUs

Abstract

Sparse matrices are a core component in many numerical simulations, and their efficiency is essential to achieve high performance. Dynamic sparse-matrix allocation (insertion) can benefit a number of problems such as sparse-matrix factorization, sparse-matrix-matrix addition, static analysis (e.g., points-to analysis), computing transitive closure, and other graph algorithms. Existing sparse-matrix formats are poorly designed to handle dynamic updates. The compressed sparse-row (CSR) format is fully compact and must be rebuilt after each new entry. Ellpack (ELL) stores a constant number of entries per row, which allows for efficient insertion and sparse matrix-vector multiplication (SpMV) but is memory inefficient and strictly limits row size. The coordinate (COO) format stores a list of entries and is efficient for both memory use and insertion time; however, it is much less efficient at SpMV. Hybrid ellpack (HYB) compromises by using a combination of ELL and COO but degrades in performance as the COO portion fills up. Rows that use the COO portion require it to be completely traversed during every SpMV operation.

In this paper we introduce dynamic compressed sparse row (DCSR), a sparse matrix format that allows for asynchronous dynamic updates. Through the use of DCSR we demonstrate an improved sparse matrix-matrix multiplication algorithm and apply this to algebraic multigrid (AMG). AMG is a multigrid operation in which the hierarchy of operators is created from the matrix itself as opposed to the geometry of the mesh. This operation can be expressed in terms of SpMM, SpMV, and primitive parallel operations.

1. Introduction

Sparse matrix-vector multiply (SpMV), the workhorse operation of many numerical simulations, has seen use in a wide variety of areas such as data mining [17] and graph analytics [13]. In scientific computing and numerical algorithms, a majority of the total processing is frequently spent on SpMV operations. Iterative computations such as the power method and conjugate gradient are commonly used in numerical simulations and require successive SpMV operations [29]. GPUs are increasingly used for computing these operations as they are, in principle, highly parallelizable. GPUs have both a high computational throughput and a high memory bandwidth. Operations on sparse matrices are generally memory bound, which makes the GPU a good target platform due to its higher memory bandwidth compared to that of the CPU. However, it is still difficult to attain high performance with sparse matrices because of thread divergence and non-coalesced memory accesses.

Some applications require dynamic updates to the matrix; generally construed, updates may include inserting or deleting entries. Fully compressed formats such as compressed sparse row (CSR) cannot handle these operations without rebuilding the entire matrix. Rebuilding the matrix is orders of magnitude more costly than performing an SpMV operation. The ellpack (ELL) format allocates a fixed amount of space for each row, allowing fast insertion of new entries and fast SpMV but limits each row to a predetermined number of entries and can be highly memory inefficient. The coordinate (COO) format stores a list of entries and permits both efficient memory use and fast dynamic updates but is unordered and slow to perform SpMV operations. The hybrid-ellpack (HYB) format attempts a compromise between these by combining an ELL matrix with a COO matrix for overflow. Operations over rows may require examination of this overflow matrix however and efficiency suffers.

Matrix representations of sparse graphs sometimes exhibit a power-law distribution (when the number of nodes with a given number of edges scales as a power of the number of edges). This distribution results in a long tail in which a few rows have a relatively high number of entries but the rest have a relatively low number. Real-world phenomena often exhibits the power-law distribution. For example, their corresponding matrices can represent adjacency graphs, web communication, and finite-state simulations. Such a matrix is also the pathological case for memory efficiency in the ELL format and requires significant use of the COO portion of a HYB matrix, making neither particularly well suited for dynamic sparse-graph applications.

One motivating application for our work is control-flow analysis (CFA), a general approach to static program analysis of higher-order languages [24, 30]. These algorithms use an approximate interpretation of their target code to yield an upper bound on the propagation of data and control through a program across all possible actual executions. A CFA involves a series of increasing operations on a graph (extending it with nodes and edges), terminating when a fixed point is reached (a steady state where the analysis is self-consistent).

Recent work has shown how to implement this kind of static analysis as linear-algebraic operations on the sparse-matrix representation of a function [14, 27]. Other recent work shows how to implement an inclusion-based points-to analysis of C on the GPU by applying a set of semantic rules to the adjacency matrix of a sparse-graph [23]. These algorithms may be likened to finding the transitive closure of a graph encoded as an adjacency matrix. The matrix is repeatedly extended with new entries derived from SpMV until a fixed point is reached (no more edges need to be accumulated). These approaches to static analysis on the GPU are very different; both however, require performant sparse-matrix operations and dynamic insertion of new entries.

Sparse-matrix factorization is the essential step in direct methods for solving linear systems. This process is highly time and memory consuming, and could benefit from efficient dynamic updates to the factors being built or reduced. Existing methods for LU-decomposition [16] and Cholesky decomposition [3] make frequent use of sparse-matrix addition to union components of the overall workload during a recursive merging step. This union of matrices is done by allocating a fresh matrix all at once or by proprietary ad-hoc methods, which have gone undiscussed in the literature. Our work provides a general matrix format that allows such merging steps to incrementally extend an existing matrix.

Sparse matrix-matrix multiplication (SpMM) is another application for efficient dynamic updates. Existing approaches use an intermediate COO format matrix to compile a list of partial results before building the final product [4]. A more efficient approach might dynamically extend the final product with these intermediate results as they are accumulated.

1.1. Contributions

Existing matrix formats are ill-suited for such dynamic allocation with many being fully compressed or otherwise unable to be efficiently extended with new entries. Our contribution in this paper is to present a fast, dynamic method for sparse matrix allocation:

1. **We review existing matrix formats and present an alternative approach to dynamic matrix allocation that allows an existing matrix to be modified arbitrarily in-place.**

2. **We use this approach to design a specific matrix format *dynamic compressed sparse row* (DCSR) that exhibits easy conversion with standard CSR, fast dynamic updates, and fast SpMV.**

3. **We implement DCSR and demonstrate its efficacy, benchmarking SpMV and insertions using the adjacency matrices for a suite of sparse-graph benchmarks.**

4. **We apply DCSR to SpMM and demonstrate it when applied to AMG. We compare our results to the standard parallel algorithm for computing SpMM using CSR and COO matrices.**

2. Background

In this paper we are concerned with dynamic updates to sparse matrices. As SpMV is arguably the most important sparse-matrix operation, we want to maintain efficient times for the problem $Ax = y$. A major goal of sparse-matrix formats is to reduce irregularity in the memory accesses. We provide a brief overview of some of the most commonly used sparse-matrix formats.

The *coordinate* (COO) format is the simplest sparse-matrix format. It represents a matrix with three vectors holding the row indices, column indices, and values for all nonzero entries in the matrix. The entries within a COO format must be sorted by row in order to efficiently perform an SpMV operation. SpMV operations are conducted in parallel through segmented reductions over the length of the arrays. Tracking which thread has processed the final entry in a row requires explicit inter-thread communication.

The *compressed sparse row/column* (CSR/CSC) formats are similar to COO in that they have arrays that fully store two of the three sets, either the column indices or the row indices in addition to the values. Either the rows or columns (in CSR or CSC, respectively) are compressed to store only the offsets into the other two arrays. For CSR, entry i and $i+1$ in the row offsets array will store the starting and ending offsets for row i. CSR has been shown to be one of the best formats in terms of memory usage and SpMV efficiency due to its fully compressed nature, and thus it has become widely used [15]. CSR has a greater memory efficiency than COO, which is a significant factor in speeding up SpMV operations due to decreased memory bandwidth usage.

The *ellpack* (ELL) format uses two arrays, each of size $m \times k$ (where m is the number of rows and k is a fixed width), to store the column indices and the values of the matrix [11, 12]. These arrays are stored in column-major order to allow for efficient parallel access across rows. This format is best suited for matrices that have a fixed number of entries per row.

Allocating enough memory in each row to store the entire matrix is prohibitively expensive for ELL when a matrix contains even one long row. The *hybrid-ellpack* (HYB) format offers a compromise by using a combination of ELL and COO. It stores as many entries as possible in an ELL portion, and the overflow from rows with a number of entries greater than the fixed ELL width is stored in a COO portion. ELL and HYB have become popular on SIMD architectures due to the ability of thread warps to look through consecutive rows in an efficient parallel manner [5].

A number of other specialized sparse-matrix formats have been developed, including jagged diagonal storage (JDS), block diagonal (BDIA), skyline storage (SKS), tiled COO

(TCOO), block ELL (BELL), and sliced-ELL (SELL) [25], all of which offer improved performance for specific matrix types. Blocked variants of these and other formats work by storing localized entries in blocks for better data locality and a reduction in index storage. "Cocktail" frameworks that mix and match matrix formats to fit specific subsets of the matrix have been developed, but they require significant preprocessing and are not easily modified dynamically [31]. Garland et al. have provided detailed reviews of the most common sparse matrix formats [11, 12, 32], as well as an analysis of their performance on throughput-oriented many-core processors [6].

Block formats such as BRC [2] and BCCOO [34] have limited ability to add in additional entries. BRC can add new entries only if those entries correspond to zeros within blocks that have been stored. BCCOO can handle the addition of new entries, but it suffers from many of the same problems as COO. Also, new insertions will not always follow a blocked structure, so additional blocks may be sparse, which lowers memory efficiency.

Many sparse matrix formats are fully compressed and do not allow additional entries to be added to the matrix dynamically. Adding additional entries to a CSR matrix requires rebuilding the entire matrix, since there is no free space between entries. Of existing formats, COO is the most amenable to dynamic updates because new entries can be placed at the end of the data structure. However, updating a COO matrix in parallel requires atomic operations to keep track of currently available memory locations. The ELL/HYB formats allow for some additional entries to be added in a limited fashion. ELL cannot add in more entries per row than the given width of the matrix, and while the HYB format has a COO matrix to handle overflow from the ELL portion, it cannot be efficiently updated in parallel since atomic operations are required and the COO portion must maintain the sorted property.

2.1. Sparse Matrix Algorithms on the GPU

A great deal of research has been devoted to improving the efficiency of SpMV, which has been studied on both multi-core and many-core architectures. Williams et al. demonstrated the efficacy of using architecture-specific data structures to optimize performance [21, 33]. Since SpMV is a bandwidth-limited operation, research has also produced other methods, such as automatic tuning, blocking, and tiling, to increase cache hit rates and decrease bandwidth usage [9, 28, 35].

The two most common CSR SpMV algorithms are CSR-scalar and CSR-vector. CSR-scalar assigns one thread per row and CSR-vector assigns a vector of threads to each row. On SIMD architectures the vector size generally never exceeds a full warp (to avoid explicit synchronization between threads). A vectorized approach allows for more efficient coalesced memory accesses. A hybrid approach has been shown to be effective. This method selectively picks between CSR-scalar and CSR-vector based on the row length [15]. Adaptive algorithms that group rows together by length and assign separate kernels to each group have also been explored [1].

Graph applications often use sparse binary adjacency matrices to represent graphs and translate graph operations to linear algebraic operations [18]. Finding the transitive closure of a graph can be done through repeated multiplication of its adjacency matrix. The transitive closure of an adjacency matrix R calculates $R^+ = \bigcup_{i \in \{1,2,3,\ldots\}} R^i$, where R^i is the i^{th} power of the matrix. The result is R^i having a nonzero between any pair of nodes connected by a path of length i. Thus, the union (addition/binary-or) of all $R, \ldots R^n$ will have a nonzero entry for every pair of nodes that are connected by a path of length $\leq n$. This process of unioning successive powers of R can be continued until a fixed point is reached. All nodes that are connected by a path of any length will be marked in the matrix.

Bandwidth limited sparse matrix-matrix operations such as sparse matrix-matrix addition $A + B = C$ and sparse matrix-matrix multiplication $AB = C$ remain difficult to compute efficiently. These operations require creating a new sparse matrix C whose entries and sparsity will depend on the sparsity patterns of A and B, and often will have a differing number of elements than either. Current implementations generally look globally at both matrices and find the intersection patterns using temporary workspace memory, after which the new matrix C can be generated [7, 19]. This often involves format conversions that consume additional time and memory.

3. Dynamic Allocation

We present a dynamic sparse-matrix allocation method that allows for efficient dynamic updates while still maintaining fast SpMV times. Our dynamic allocation uses a *row offset array*, representing a dense array of ordered rows, and for each a fixed number of *segment* offsets. The column indices and values are stored in arrays that are logically divided into these data segments in the same way that CSR row offsets partition the column indices and values. Each such segment is a contiguous portion of memory that stores entries within a row. Segments may contain more space than entries to allow for future insertions. The contiguous layout of entries within the set of segments for a given row is equivalent to the corresponding row in CSR format. In the following subsection we illustrate how dynamic allocation is performed, after which we provide details of how DCSR operations are implemented. We then present our implementation of an improved SpMM algorithm that utilizes DCSR for asynchronous dynamic writes to the resulting C matrix.

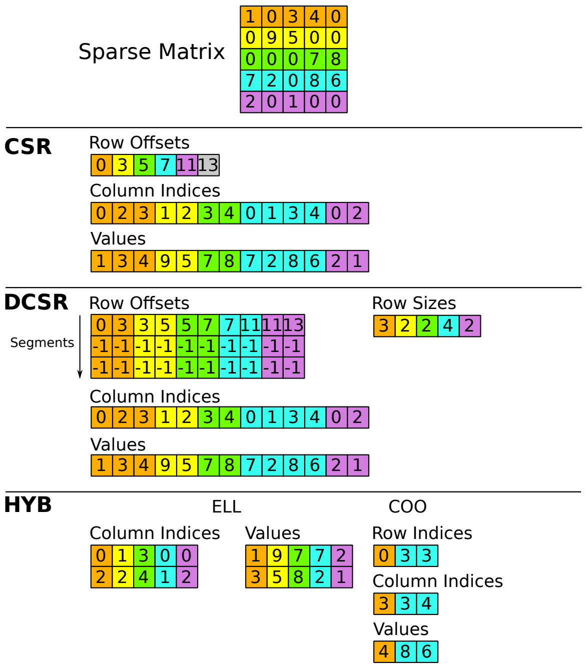


Figure 1: Comparison of CSR, DCSR, and HYB formats.

3.1. Dynamic CSR

Initializing the matrix can be done in one of two ways. Either a matrix can be loaded from another format (e.g., COO or CSR) or the matrix can be initialized as blank. In the latter case, each row is assigned an initial number of entries (an initial segment size) in the column indices and values arrays. The row offset array is initialized with space for k segment offset pairs, with either no allocated segments or a single allocated segment of size μ per row. The latter case will consume the same amount of memory as an ELL matrix with a row width of μ, except in row-major order instead of column-major order. To allow for dynamic allocation we maintain a larger memory buffer than needed and use simple bump-pointer allocation to add new segments. This allocation pointer is set to the end of the currently used space ($rows \times \mu$ in the case of a new matrix). A maximum size of memory buffer for the columns and values arrays is specified by the user. Figure 1 provides a illustrative comparison of CSR, HYB, and DCSR formats.

The format consists of four arrays for column indices, values, row offsets, and row sizes, in addition to a memory allocation pointer. The row offsets array functions in a similar manner to that of its CSR counterpart, except that both a beginning and ending offset are stored and space exists for up to k such pairs per row. This table is encoded as a strided array where the starting and ending offsets of segment k in row i are indexed by $(i * 2 + k * pitch)$ and $(i * 2 + k * pitch + 1)$, respectively. The $pitch$ may be defined as a value convenient for cache performance such that $pitch \geq 2 * rows$. Each set of offsets for a given segment lies within a different cache line, which serves to increase memory aligned accesses. The number of memory segment offset pairs (the max k) is an adjustable parameter specified at matrix construction. The column indices and values correspond 1:1, just as in CSR. Unlike CSR, however, there may be more than one memory segment assigned to a given row, and the segments need not be contiguous. As the last segment for a row may not be full, the actual row sizes are maintained so the used portion of each segment is known.

Explicitly storing row sizes allows for optimization techniques such as *adaptive CSR* (ACSR) [1] (of which we take advantage). This optimization implements customized kernels to process bins of specified row-lengths. During this binning process we create a permuted set of row indices that are sorted according to these bin groupings. We launch each bin-specific kernel with these permuted indices on its own stream, which allows each kernel to easily access the rows that it needs to process without scanning over the matrix.

When inserting new elements within a row, the last allocated segment for that row is located and if space is available the new elements are inserted in a contiguous fashion just after current entries. If that segment does not have enough room, a new segment will be allocated with the appropriate size plus an additional amount α. The α value represents additional "slack space" and allows for a greater number of entries to be inserted without the creation of a new segment. If dynamic updates follow a power-law distribution, there will be a higher probability of additional entries being inserted into longer rows. Although we experimented with setting α to be a factor of the previous segment size, for our tests we settled on a value of μ (average row size of matrix). When a new segment is allocated, the memory allocation pointer is atomically increased by the size of the new segment. A hard limit on these additions, before defragmentation is required, is fixed by the number of segments k. The defragmentation operation always reduces the number of segments in each row to one, which allows the format to scale to an arbitrary number of allocations.

When inserting new elements into the matrix, it is possible that duplicate nonzero entries (i.e., two or more entries with the same row and column index) will be added. Duplicate entries are handled in one of two ways. The first method is to simply let them accumulate, which does not pose a problem for many operations. SpMV operations are tolerant of duplicate entries due to the distributive property of the inner product and will yield the same result to within floating point tolerance. For binary matrices the row-vector inner products will produce the same result irrespective of duplicate nonzeros. A second solution is to perform a segmented reduction on the entries after sorting by row and column, which combines all entries with matching row and column indices into a single entry. This full reduction is generally not needed when performing only SpMV and addition operations. Sparse matrix-matrix multiplication (SpMM) operations may cause significant fill-in which would require such a reduction to be performed. In our SpMV tests we let the values accumulate for all formats as they do not hinder the SpMV operations that are performed.

An SpMV operation works as follows. The first pair of segment offsets is fetched. The entries within the corresponding segment are multiplied by the appropriate values in x according to the algorithm being used (CSR-scalar, CSR-vector, etc.). If the row size is greater than the capacity of the current memory segment, the next pair of offsets is fetched. If the size of the current segment plus the running sum of the previous segment sizes is greater than or equal to the row size, this is the final segment of the row. In case the final segment is not full, the location of the last entry can be determined by the difference of the row size and the running sum. This process continues until the entire row has been read.

As the matrix accumulates more segments, SpMV performance decreases slightly. A fixed number of segments also means this process cannot continue forever. Our solution to both problems is to implement a defragmentation operation that compacts all the entries within the column indices and values arrays, eliminating empty space. This defragmentation combines all the segments in a row into a single segment that compactly stores the entire row. This operation may be invoked periodically, or more conservatively when a row has reached its maximum capacity of segments. In practice we do the latter and set a flag when any row reaches its maximum segment count. At this point we consider defragmentation to be required.

Defragmentation performs the equivalent to a sort by row operation on the entries of the matrix; we formulated a method that does not require an actual sort and is significantly faster than doing so. Since we explicitly store row sizes, we perform a prefix-sum operation on them to calculate the new row offsets in a compacted CSR form. The entries are then shuffled from their current indices to their new indices in newly allocated column indices and values buffers, after which we set a pointer in our data structure to these new arrays and free the old buffers (shallow copy). By using the knowledge of the row sizes to compute resulting offsets and indices, we eliminate the need to do any comparisons in this operation, which greatly improves performance.

Figure 2 illustrates an example of inserting new elements into a DCSR matrix. Initially, the matrix has four populated rows with the memory allocation pointer being 16. Row 0 can insert 1 additional entry in its current segment before a new segment would need to be allocated. Rows 1 and 2 have enough room for two additional entries, but row 3 is full. Figure 2 shows a set of new entries that are inserted into rows 0, 2, and 3. In this case a new segment of size 4 is allocated for row 0 and row 3. The additional segments need not be consecutive nor in order of row since the exact offsets are stored for each segment. Finally, the defragmentation operation computes new segment offsets from the row sizes. The entries are shuffled to their new indices, which results in a single compacted segment for each row.

Figure 2: Illustration of insertion and defragmentation operations with DCSR.

As CSR is the most commonly used sparse matrix format, we designed DCSR to be compatible with CSR algorithms and to allow for easy conversion between the formats. Minimal overhead is required to convert from CSR to DCSR and vice versa. When converting from CSR to DCSR, the column indices and values arrays are copied directly. For the row offsets array, the i^{th} element is copied to indices $i*2-1$ and $i*2$ for all elements except the first and last one. A simple subtraction must be performed to calculate the row sizes from the row offsets. Converting back is equally simple, assuming the matrix is first defragmented; the column indices and values arrays are copied back, and the starting segment offset from each row is copied into the row offsets array.

3.2. Sparse Matrix-Matrix Multiplication (SpMM)

It is a difficult task to efficiently compute $C = AB$ for sparse matrices in parallel. The sequential sparse matrix-matrix multiplication algorithm is not suitable for fine-grained parallelization. Sequential algorithms are efficient, but they rely on a large amount of (per thread) temporary storage. Specifically, to compute the sparse product $C = AB$, the sequential methods use $O(N)$ additional storage, where N is the number of columns in C. The parallel approach to sparse matrix-matrix multiplication is formulated in terms of highly scalable parallel primitives with no such limitations. As a result, a straightforward parallelization of the sequential scheme requires $O(n)$ storage per thread, which is not possible when using tens of thousands of independent threads of execution. Although it is possible to construct variations of the sequential method with lower per-thread storage requirements, any method that operates on the granularity of matrix rows (i.e., distributing matrix rows over threads), requires a nontrivial amount of per-thread state and suffers load imbalances for certain input [4].

The standard algorithm for parallel SpMM that exposes fine-grained parallelism is:

1. Expansion of $A * B$ into an intermediate coordinate format T.
2. Sorting of T by row and column indices to form $\hat{T}$.
3. Compression of $\hat{T}$ by summing duplicate values for each matrix entry.

All three stages of the algorithm expose fine-grained parallelism that the GPU can take advantage of. The algorithm can be formulated in terms of efficient data-parallel computations — gather, scatter, scan, sort, etc. Like the sequential algorithm, this formulation is work efficient. It computes the exact number of partial products required for each nonzero without performing any additional operations with zero entries. It has the same computational complexity as the sequential method $O(nnz(T))$. The complexity is proportional to the size of the intermediate format T, and the work required at each stage is linear with respect to T. This process results in a relatively even load balancing across the GPU regardless of the sparsity patterns of the input matrices.

A limitation of this method is that the memory required to store the intermediate format is potentially large. If A and B are both square, $n \times n$ matrices with exactly K entries per row, then $O(nK^2)$ bytes of memory are needed to store T. Since the input matrices are generally large themselves ($O(nK)$ bytes), it is not always possible to store a K-times larger intermediate result in memory. In the limit, if A and B are dense matrices (stored in sparse format), then $O(n^3)$ storage is required. In such a case the matrix-matrix multiplication $C = AB$ can be decomposed into several smaller operations that are computed in a workspace of bounded size. The resulting slices are then concatenated together to produce the final result. This technique introduces some overhead, but in practice it is relatively small as the workspace can be sized appropriately to saturate the device.

Our implementation of SpMM follows the same principles as the general algorithm, but we assign specialized kernels to process rows grouped by size. This algorithm allows for a more efficient use of shared memory when performing the sort and reduction operations. DCSR allows for asynchronous dynamic memory allocations when storing the rows products into C. This property of DCSR allows computation of the rows in any order. In the standard algorithm the result of each previous row is required to know the offset when writing the final result into C. We precompute the number of partial products per row i following:

$$\sum_{k=1}^{ARS_i} BRS_j$$

where ARS_i is the number of entries in row i of matrix A, and j is the column index of element $a_{i,j}$. We then assign specific kernels, based on this row size, to process rows of length 1-32, 33-64, 65-128, 129-256, 257-512, 513-1024, 1025-2048, and 2049+.

The kernels process a row by computing the partial products, sorting them by column index, and reducing them before storing them in the resulting C matrix. Since this is done on a per row basis, the row is implicit and we need only store the column indices and values for the sorting and reduction phases. For all kernels except the 2049+ kernel, the operations are computed within shared memory on the GPU, which provides a significant performance improvement over global memory. For the 2049+ kernel we use dynamic parallelism to assign a compute kernel to each row, which performs these operations using global memory.

4. Experimental Results

To benchmark SpMV, update, and conversion performance, we used a node with an Intel Xeon E5-2640 processor running at 2.50GHz, 128GB of memory, and a NVIDIA Tesla K20c GPU. We compiled using g++ 4.7.2, CUDA 7.5, CUSP 0.5.1, and Thrust 1.8.1, comparing our method against modern implementations in CUSP [7] and cuSPARSE [26]. Table 1 provides a list of the matrices that we used in our tests as well as their sizes, number of nonzeros, and row entry distributions. All the matrices can be found in the University of Florida sparse-matrix database [10].

Memory consumption is a major concern for sparse matrix formats, as one of the primary reasons for eliminating the storage of zeros is to reduce the memory footprint. The ELL component of HYB is best suited to store rows with an equal number of entries. If there is a large variance in row size, much of the ELL portion may end up storing zeros, which is inefficient. We provide a comparison of memory consumption for HYB, DCSR (using 2, 3, and 4 segments), and CSR formats in Table 2. We compute the storage size of the HYB format using an ELL width equal to the average number of nonzeros per row (μ) for the given matrix. CSR has the smallest memory footprint since its row indices have been compressed to the number of rows in the matrix. We see

Matrix	Abbr.	NNZ	Rows \ Cols	$\mu \setminus \sigma \setminus$ Max
amazon-2008	AMA	5M	735K	7 \ 4 \ 10
cnr-2000	CNR	3M	325K	9 \ 21 \ 2716
dblp-2010	DBL	807K	326K	2 \ 4 \ 154
enron	ENR	276K	69K	3 \ 28 \ 1392
eu-2005	EU2	19M	862K	22 \ 29 \ 6985
flickr	FLI	9M	820K	11 \ 87 \ 10K
hollywood-2009	HOL	57M	1139K	50 \ 160 \ 6689
in-2004	IN2	16M	1382K	12 \ 37 \ 7753
indochina-2004	IND	194M	7414K	26 \ 216 \ 6985
internet	INT	207K	124K	1 \ 4 \ 138
kron-18	KRO	10M	262K	40 \ 261 \ 29K
ljournal-2008	LJO	79M	5363K	14 \ 37 \ 2469
rail4284	RAL	11M	4K \ 1M	2633 \ 4K \ 56K
soc-LiveJournal1	SOC	68M	4847K	14 \ 35 \ 20K
webbase-1M	WEB	3M	1000K	3 \ 25 \ 4700
wikipedia-2005	WIK	19M	1634K	12 \ 31 \ 4970

Table 1: Matrices used in tests. NNZ: total number of nonzeros, μ: average row size, σ: standard deviation of row sizes, Max: maximum row size

Matrix	HYB size	DCSR 2 segs.	DCSR 3 segs.	DCSR 4 segs.	CSR
AMA	54M	0.924	1.026	1.128	0.77
CNR	47M	0.626	0.679	0.732	0.547
DBL	12M	0.86	1.052	1.245	0.572
ENR	4M	0.653	0.762	0.871	0.489
EU2	236M	0.675	0.703	0.731	0.633
FLI	160M	0.546	0.585	0.624	0.487
HOL	859M	0.531	0.541	0.551	0.516
IN2	229M	0.654	0.7	0.746	0.585
IND	2791M	0.571	0.591	0.612	0.541
INT	4M	0.761	0.969	1.177	0.449
KRO	171M	0.493	0.505	0.516	0.475
LJO	1152M	0.594	0.63	0.665	0.541
RAL	149M	0.577	0.577	0.577	0.576
SOC	1009M	0.595	0.631	0.668	0.54
WEB	40M	0.966	1.155	1.344	0.682
WIK	276M	0.635	0.68	0.725	0.567

Table 2: Comparison of memory consumption between HYB, CSR, and DCSR formats. Size of HYB is listed in bytes (using ELL width of μ), and sizes for DCSR and CSR are listed as a percent of the HYB size.

that DCSR has a significantly smaller memory footprint in almost all test cases. Test cases such as AMA and DBL have lower memory consumption for HYB than for DCSR (with 3 and 4 segments), because these matrices have a low row size variance. DCSR with 4 segments uses 20% less memory on average than HYB.

Conversion times between formats are often a key factor when determining the efficacy of a particular format. High conversion times can be a significant hindrance to efficient performance. Architecture-specific formats may provide better performance, but unless the rest of the code base uses that format, the conversion time must be accounted for. We provide the overhead required to convert to and from CSR and COO matrices in Table 3. The conversion times have been normalized against the time required to copy CSR $\rightarrow$ CSR. The conversion times to DCSR are only slightly higher compared to that of CSR. HYB requires significant overhead as the entries must first be distributed throughout the ELL portion and the remaining overflow entries distributed into the COO portion.

4.1. Matrix Updates

To measure the speed of dynamic updates, we ran two series of tests, which involved streaming updates and iterative updates. In the streaming updates test, we incrementally build up the matrix by continuously inserting new entries. The elements are first buffered into three arrays, representing the rows indices, column indices, and values. We initialize the matrix sizes according to the average number of nonzeros for the given input. The entries are then added in a streaming parallel fashion to the matrices.

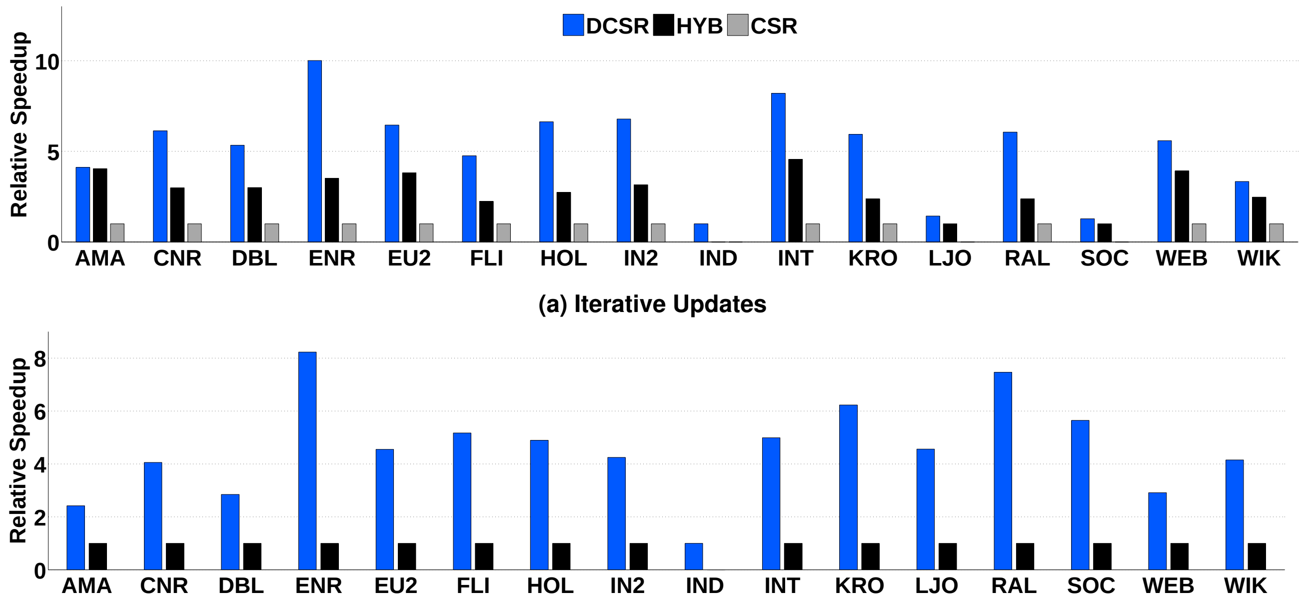
Updating a HYB matrix first requires checking the ELL portion, and if the row in question is full, inserting the new entry into the COO portion. Any updates to the COO portion require atomic operations to ensure synchronous writes between multiple threads. These atomic updates are prohibitive for fast parallel updates as all threads are contending to insert entries onto the end of the COO matrix.

Updating a DCSR matrix requires finding the last occupied (current) segment within a row. If that segment is not full, the new entry is added into it and the row size is increased. When the current segment for a row fills up, a new segment is allocated dynamically. Since atomic operations are required only for the allocation of new segments, and not for each individual element, synchronization overhead is kept low. By allowing for dynamically sized slack space within a row, we dramatically reduce the number of atomic operations that are required to allocate new entries. In this way DCSR was designed to be updated in an efficient parallel manner.

The number of segments, initial row width, and α value can be tuned for the problem to give a reasonable limit on updates. In our tests we used four segments and α value of μ (average row size of the matrix). When a row nears its limit, a defragmentation is required in order to reduce that row to a single segment.

Figure 3 provides the results of our iterative and streaming matrix update tests. We do not compare to CSR in the latter case, since it is not possible to dynamically add entries without rebuilding the matrix. This operation only loads the matrix and does not perform any insertion checks. DCSR saw an average speedup of $4.8\times$ over HYB with streaming updates. In the case of IND only DCSR was able to perform the operation within memory capacity.

We also executed an iterative update test to compare the ability of the formats to perform a combination of dynamic



(a) Iterative Updates

(b) Streaming Updates

Figure 3: Top: Relative speedup of DCSR compared to HYB for iterative updates with SpMV operations. The speedup is compared to a normalized CSR baseline. Bottom: Relative speedup of DCSR compared to HYB for matrix updates.

From To	COO CSR	COO DCSR	COO HYB	CSR DCSR	CSR HYB	DCSR CSR
AMA	2.93	3.03	9.22	1.06	9.25	0.9
CNR	2.24	2.62	14.84	1.04	13.62	0.87
DBL	4.34	5.74	18.07	1.17	16.83	1.1
ENR	5.56	5.95	27.15	1.29	26.95	1.14
EU2	2.1	2.29	16.08	1.06	15.67	0.99
FLI	2.13	2.5	23.29	1.06	19.74	0.96
HOL	1.82	1.9	20.37	1.01	20.3	0.99
IN2	2.15	2.42	18.12	1.06	18.15	0.98
IND	1.93	1.98	∞	1.03	∞	1.01
INT	12.07	13.74	21.38	1.3	15.12	1.0
KRO	1.78	2.09	24.01	1.0	20.14	0.91
LJO	2.09	2.19	19.96	1.02	19.97	0.98
RAL	1.73	2.03	20.67	1.0	17.97	0.91
SOC	2.22	2.35	20.47	1.06	20.41	1.01
WEB	2.89	3.19	11.45	1.16	11.56	0.86
WIK	2.18	2.42	20.13	1.07	20.11	0.98

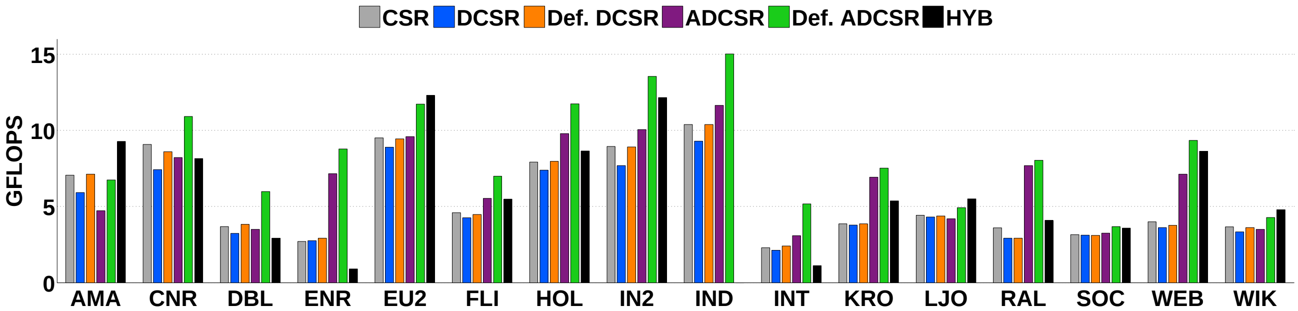
Table 3: Comparison of relative conversion times. Conversions are normalized against time to copy CSR→CSR.

updates and SpMV operations. This test is analogous to what would be done in a graph application (such as CFA) where the graph is updated at periodic intervals. In the iterative updates test we perform a series of iterations consisting of a matrix addition operation $(A = A + B)$ followed by several SpMV operations $Ax = y$. Part (a) of Figure 3 provides the results for our iterative updates. Within each iteration, the matrix is updated with an additional 0.2% random nonzeros followed by 5 SpMV operations, which is repeated 50 times. This process yields a total increase of 10% to the number of nonzeros. We compare the DCSR and HYB results to a normalized CSR baseline. In the CSR case a new matrix must be created to update the original matrix, which causes a significant amount of overhead (in terms of computation and memory). In the cases of LJO and SOC, CSR was not able to complete within memory capacity, so we normalized against HYB.

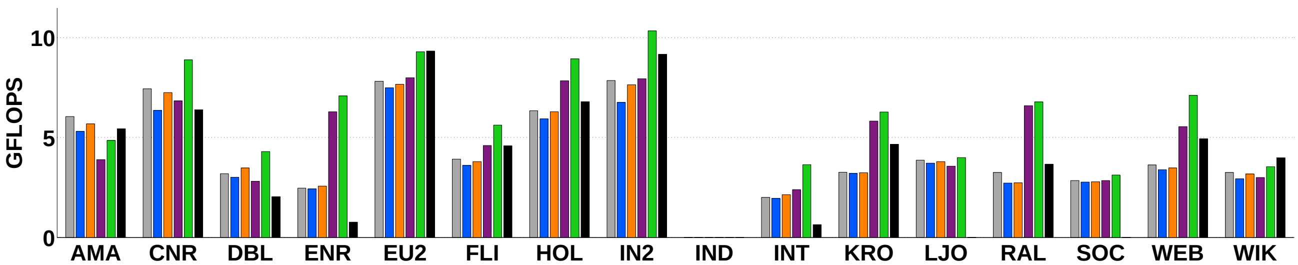
DCSR shows significant improvement over HYB on streaming updates in all test cases (in some by as much as $8\times$). DCSR also outperforms HYB in all test cases on iterative updates, and in some cases by as much as $2.5\times$. The Amazon-2008 matrix has a low standard deviation, and the majority of its entries fit nicely into the ELL portion, which greatly speeds up SpMV operations. However, even in this case DCSR slightly outperforms HYB on iterative updates due to having lower overhead for defragmentation. In all other cases DCSR exhibits noticeable performance improvements over HYB and CSR.

4.2. SpMV Results

In the SpMV tests we take the same set of matrices and perform SpMV operations with randomly generated dense vectors. We performed each SpMV operation $100\times$ times and averaged the results. Figure 4 provides the results for these SpMV tests run using both single- and double-precision floating-point arithmetic. We implemented the adaptive binning optimization (ACSR) outlined in [1], which we labeled ADCSR. This optimization requires relatively little overhead and provides noticeable speed improvements by using spe-



(a) Single Precision



(b) Double Precision

Figure 4: FLOP ratings of SpMV operations for CSR, DCSR, and HYB.

cialized kernels on bins of rows with similar row sizes. In these tests we compare across several variants of our format, including DCSR, defragmented DCSR, ADCSR, and defragmented ADCSR, in addition to standard implementations of HYB and CSR.

The fragmented DCSR times are 8% slower than the defragmented DCSR times on average. When the DCSR format is defragmented, it sees SpMV times competitive with those of CSR (1% slower on average). With the adaptive binning optimization applied, we see that ADCSR outperforms HYB in many cases. ADCSR performs 9% better on average than HYB across our benchmarks.

4.3. Post-Processing Overhead

Post-processing overhead is a concern when dealing with dynamic matrix updates. Dynamic segmentation allows for DCSR to be updated with new entries without requiring the entries to be defragmented. SpMV operations can be performed on the DCSR format regardless of the number and order of segments, in contrast to HYB matrices where a sort is required anytime an entry is added that overflows into the COO portion. The SpMV operation for HYB matrices assumes the COO entries are sorted by row (without this property the COO SpMV would be dramatically slower). Table 4 provides post-processing times for HYB and DCSR formats relative to a single SpMV operation. In the case of IND, HYB was unable to sort and update due to insufficient memory (overhead represented as ∞).

The defragmentation operation gives us an opportunity to internally order rows by row-size at no additional cost. Our defragmentation algorithm is similar to the row sorting technique illustrated in [20], although we use a global sorting scope as opposed to a localized one. Because we explicitly manage segments within the columns and values arrays by both starting and ending index, the internal order of segments may be changed arbitrarily, and this permutation re-

Matrix	DCSR defrag	HYB sort	DCSR update	HYB update
AMA	3.9	2.12	2.02	4.89
CNR	5.13	6.75	3.77	15.26
DBL	5.69	4.66	3.6	10.23
ENR	5.49	8.0	2.21	18.2
EU2	2.32	4.28	2.65	12.05
FLI	1.58	4.22	1.94	10.01
HOL	1.54	5.57	2.55	12.45
IN2	2.58	5.85	3.14	13.34
IND	2.15	∞	3.36	∞
INT	6.74	6.19	1.76	8.78
KRO	1.02	3.43	1.82	11.3
LJO	1.45	3.02	1.34	6.1
RAL	0.72	2.04	1.82	13.61
SOC	1.05	3.74	1.02	5.74
WEB	2.65	1.93	2.54	7.39
WIK	1.39	2.54	1.32	5.49

Table 4: Overhead of DCSR defragmentation and HYB sorting is measured as the ratio of one operation against a single CSR SpMV. Update time is measured as the ratio of 1000 updates to a single CSR SpMV.

mains invisible from the outside. To accomplish this optimization we permute row sizes according to the permuted row indices (which have already been binned and sorted by row size). The permuted row sizes can then be used to create new offsets for the monolithic segments produced by defragmentation. The column and value data can be internally reordered by row size at no additional cost. We observed this internal reordering to provides a noticeable SpMV performance improvement of 12%. This improvement is from an increased cache-hit rate via better correlation between bin-specific kernels and the memory they access.

The DCSR defragmentation incurs a lower overhead than HYB sort because entries can be shuffled to their new index without a sort operation. A DCSR defragmentation step is $2\times$ faster on average than the HYB sorting step. More importantly this is required infrequently, while HYB sorting must be performed at every insertion, which means that DCSR requires significantly lower total post-processing overhead.

4.4. SpMM

We test the efficiency of our SpMM method through its application to algebraic multigrid. We compare our method to a similar version that computes SpMM using CSR and COO matrices. AMG can be formulated in terms of SpMM, SpMV, and primitive parallel operations. Algorithm 1 illustrates the structure of the AMG preconditioner setup phase of AMG given a sparse matrix A and a set of vectors B, which may represent low eigen modes of the problem. In our tests we used a constant vector, which is a common default. The $(R_k A_k P_k)$ operation computes the Galerkin product of the three matrices using SpMM by first computing $A * P = AP$ followed by $R * AP = RAP$.

Algorithm 1: AMG Setup

Input: A, B
Output: $A_0,\ldots,A_M$, $P_0,\ldots,P_M$
$A_0 \leftarrow A$, $B_0 \leftarrow B$;
for $k = 0, \ldots, M$ **do**
    $C_k \leftarrow$ strength(A_k);
    $Agg_k \leftarrow$ aggregate(C_k);
    $T_k, B_{k+1} \leftarrow$ tentative(Agg_k, B_k);
    $P_k \leftarrow$ prolongator(A_k, T_k);
    $R_k \leftarrow P_k^T$;
    $A_{k+1} \leftarrow (R_k A_k P_k)$;

We compare the results for AMG on 2D and 3D Poisson problems with Dirichlet boundary conditions. It is known that AMG performs well as a preconditioner on such problems, which allows us to focus on the merits of the SpMM method rather than on weather AMG is suited for the problem. Table 5 lists the set of matrices used in our tests as well as the number of unknowns and nonzeros. These tests were all computed with double precision.

Matrix	Abbr.	Unknowns	Nonzeros
2D Poisson 5pt	2-5-a	262144	1310720
2D Poisson 9pt	2-9-a	262144	2359296
3D Poisson 7pt	3-7-a	262144	1810432
3D Poisson 27pt	3-27-a	262144	6859000
2D Poisson 5pt	2-5-b	1048576	5238784
2D Poisson 9pt	2-9-b	1048576	9424900
3D Poisson 7pt	3-7-b	2097152	14581760
3D Poisson 27pt	3-27-b	2097152	55742968

Table 5: List of matrices used for AMG tests.

Figure 5 illustrates the results of our AMG tests with both the individual SpMM times and the overall AMG preconditioner times. Our method outperforms the baseline method by upwards of $3\times$ in some cases. The Galerkin product represents 30% – 50% of total time required by the setup phase of the preconditioner. Results shown in [4] indicate that the Galerkin product occupies 50% – 60% of the run time on similar matrices using a Nvidia Tesla C2050 GPU. This seems to indicate that the underlying architecture plays a role in the relative processing times across stages. In the case of matrix 3-7-a, the Galerkin product occupies roughly half of the setup time, and our SpMM method is nearly $3\times$ faster in that case, resulting in a speedup of 40%. There is no guarantee what the resulting fill will be in the C matrix, but in practice the resulting fill is relatively sparse for multiplication with Poisson matrices.

By taking advantage of asynchronous updates enabled by DCSR, we are able to employ specialized kernels based on row lengths. These row length optimized kernels perform the sort and reduction operations within shared memory, which is notably faster than performing these operations within global memory. The efficient use of shared memory leads to significant performance gains for the overall SpMM operation. The Galerkin product is by far the largest single component of the setup phase, so improvements in this area will lead to the greatest gains.

5. Conclusion

We have described a fast, flexible, and memory-efficient strategy for dynamic sparse-matrix allocation. The design of current formats limits the extension of an existing matrix with new entries. As many applications require or would benefit from efficient dynamic updates, we have proposed a strategy of explicitly managed dynamic segmentation that makes this operation inexpensive. We demonstrate this approach with a new sparse matrix format (DCSR), that provides a robust method for allocating streaming updates while maintaining fast SpMV times on par with CSR. The format gracefully degrades in performance upon dynamic extension, but does not require a sort to be performed after inserting new entries (as opposed to COO-based formats such as HYB).

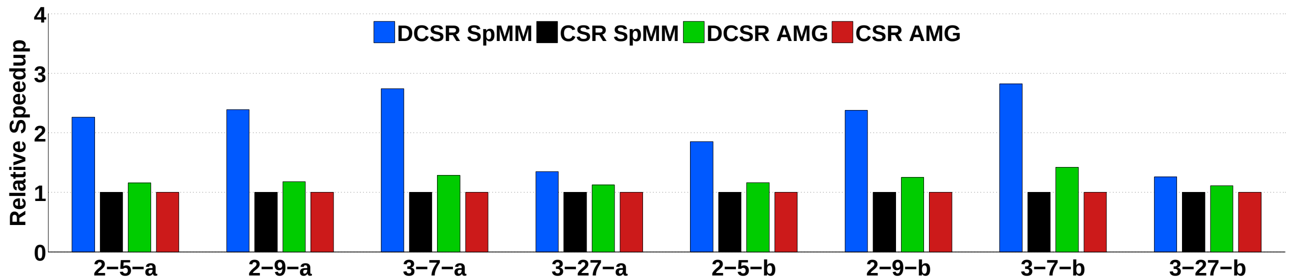


Figure 5: Relative speedup for SpMM and AMG using DCSR and CSR.

Without defragmentation, SpMV times are only marginally slower than that of a fully constructed CSR matrix, and after defragmentation they are roughly equal. With adaptive binning applied, DCSR gives faster overall SpMV times compared to the HYB format. DCSR is significantly more efficient in terms of memory use as well. ELL must allocate enough room in every row for the longest row in a matrix. The HYB format improves in this area by allowing long rows to overflow into its COO portion, but DCSR exhibited lower memory consumption on every benchmark when set to allow two segments per row, and still used 20% less memory on average when allowing four segments per row.

A key advantage of DCSR is compatibility with CSR-scalar, CSR-vector, and other CSR algorithms. Only minor modifications are required to account for a difference in the format of the row offsets array. We have demonstrated how CSR-specific optimizations, such as adaptive binning, can be easily applied to DCSR. Other optimizations such as tiling and blocking could also be used. This compatibility also means that minimal overhead is required to convert to and from CSR. Numerous sparse-matrix formats have been developed that are specifically tailored to GPU architectures. These formats offer improved performance but require converting from whatever previous format was being used. As CSR is the most commonly used sparse-matrix format, and large amounts of software already incorporate it into their code bases, it is often not worth the conversion cost to introduce another format. DCSR reduces this barrier with a low cost of conversion.

We demonstrated that DCSR significantly improves SpMM for some matrices by as much as $2\times - 3\times$, and applied the format along with our SpMM method to algebraic multigrid. The ability to asynchronously update the C matrix allows for the rows to be processed independently in any order. This asynchronous property is key to enabling the row binning technique which provides significant speedup to the sorting and reduction operations.

To the best of our knowledge, no other work has created a dynamic format like DCSR for iterative updates to sparse matrices. Some dynamic graph algorithms, such as approximate betweenness centrality [22], require dynamic updates but do not specify how the graph should be represented and modified. A matrix encoding would require a format such as DCSR to be efficient. Dynamic insertion algorithms, such as those described in [8], use a modified insertion sort that disperses gaps throughout the data in order to reduce insertion time from $O(n)$ to $O(log\ n)$ with high probability. This method probabilistically reduces the overall cost of the insertion sort from $O(n^2)$ to $O(n\ log\ n)$. The defragmentation operation we implement can be done in $O(n)$, and insertions require $O(1)$, which is better than insertion sort. Also, leaving many intermittent gaps between the data would slow SpMV times. We mitigate this problem by grouping entries contiguously within segments.

We believe our strategy fits certain operations and problems, such as graph algorithms, that require periodically updating the graph with new entries. These are applications that have not previously been well addressed by sparse-matrix formats. Our work also opens up a number of interesting research questions as to whether existing algorithms that rebuild matrices between iterations could be improved by a matrix format that permits these dynamic updates directly.

We pursue this hypothesis in continuing work that applies DCSR to GPU-based CFA (static analysis of functional programs). CFA was a primary motivation behind this work and, much like the transitive-closure problem, progress is made through a series of small updates to a graph. Recent work has shown how such program analyses may be more efficiently implemented via an encoding to linear algebra; however, existing implementations compromise run time by using a CSR format that must be constantly rebuilt, or memory efficiency by using an ELL format that must be statically allocated with sufficient space. In the latter case this is impossible for all but the most trivial inputs.

References

[1] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 781–792, Piscataway, NJ, USA, 2014. IEEE Press.

[2] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. An Efficient Two-dimensional Blocking Strategy for Sparse Matrix-vector Multiplication on GPUs. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 273–282, New York, NY, USA, 2014. ACM.

[3] Haim Avron and Anshul Gupta. Managing Data-movement for Effective Shared-memory Parallelization of Out-of-core Sparse Solvers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 102:1–102:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[4] Nathan Bell, Steven Dalton, and Luke N. Olson. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing*, 2012. accepted.

[5] Nathan Bell and Michael Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[6] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[7] Nathan Bell and Michael Garland. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, 2012. Version 0.3.0.

[8] Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion Sort is O(n log n). *Theory of Computing Systems*, 39(3):391–397, 2006.

[9] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. *SIGPLAN Not.*, 45(5):115–126, January 2010.

[10] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[11] Michael Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 2–6, New York, NY, USA, 2008. ACM.

[12] Michael Garland and David B. Kirk. Understanding Throughput-oriented Architectures. *Commun. ACM*, 53(11):58–66, November 2010.

[13] JohnR. Gilbert, Steve Reinhardt, and ViralB. Shah. High-Performance Graph Algorithms from Parallel Sparse Matrices. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 260–269. Springer Berlin Heidelberg, 2007.

[14] Thomas Gilray, Jim King, and Matthew Might. Partitioning 0-CFA for the GPU. *Workshop on Functional and Constraint Logic Programming*, September 2014.

[15] Joseph L. Greathouse and Mayank Daga. Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 769–780, Piscataway, NJ, USA, 2014. IEEE Press.

[16] Anshul Gupta, Seid Koric, and Thomas George. Sparse Matrix Factorization on Massively Parallel Computers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 1:1–1:12, New York, NY, USA, 2009. ACM.

[17] Eun-jin Im, Eun-jin Im, Katherine Yelick, and Katherine Yelick. Optimization of Sparse Matrix Kernels for Data Mining. In *First SIAM Conf. on Data Mining*, 2000.

[18] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011.

[19] Khronos Group. *The OpenCL Specification*, September 2011.

[20] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR*, abs/1307.6209, 2013.

[21] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.

[22] A. McLaughlin and D.A. Bader. Revisiting Edge and Node Parallelism for Dynamic GPU Graph Analytics. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1396–1406, May 2014.

[23] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices*, 47(8):107–116, 2012.

[24] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 44(3):10:1–10:33, June 2012.

[25] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In YaleN. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg, 2010.

[26] NVIDIA. *CUDA CUSPARSE Library*, August 2010.

[27] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the Symposium on the Principals of Programming Languages*, pages 511–522, January 2010.

[28] I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, May 2012.

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

[30] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991.

[31] Bor-Yiing Su and Kurt Keutzer. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 353–364, New York, NY, USA, 2012. ACM.

[32] Richard Wilson Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, 2003. AAI3121741.

[33] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[34] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 107–118, New York, NY, USA, 2014. ACM.

[35] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining. *Proc. VLDB Endow.*, 4(4):231–242, January 2011.