2015 IEEE 29th International Parallel and Distributed Processing Symposium

Optimizing Sparse Matrix Operations on GPUs using Merge Path

Steven Dalton, Luke Olson
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, 61821
{*dalton6, lukeo*}*@illinois.edu*

Sean Baxter, Duane Merrill, Michael Garland
Nvidia Research
Santa Clara, CA, 95050
{*sbaxter, dmerrill, mgarland*}*@nvidia.com*

***Abstract*—Irregular computations on large workloads are a necessity in many areas of computational science. Mapping these computations to modern parallel architectures, such as GPUs, is particularly challenging because the performance often depends critically on the choice of data-structure and algorithm. In this paper, we develop a parallel processing scheme, based on Merge Path partitioning, to compute segmented row-wise operations on sparse matrices that exposes parallelism at the granularity of individual nonzeros entries. Our decomposition achieves competitive performance across many diverse problems while maintaining predictable behavior dependent only on the computational work and ameliorates the impact of irregularity. We evaluate the performance of three sparse kernels: SpMV, SpAdd and SpGEMM. We show that our processing scheme for each kernel yields comparable performance to other schemes in many cases and our performance is highly correlated, nearly 1, to the computational work irrespective of the underlying structure of the matrices.**

***Keywords*-parallel; gpu; sparse matrix-matrix; sorting;**

I. Introduction

Operations that are naturally decomposed into a fixed number of equally sized components are inherently amenable to parallel processing. However, processing operations in which the workload is highly irregular and data-dependent is substantially more challenging. For *fine-grained* parallel processing environments, such as GPUs, the issues of effectively decomposing and mapping irregular work to computational units is amplified due to the hierarchical structure of the threads in the architecture and the relatively small working set size of each individual thread. In this work we study the performance of three operations on sparse matrices, sparse matrix-vector multiplication (SpMV), sparse matrix-sparse matrix addition (SpAdd), and sparse matrix-sparse matrix multiplication (SpGEMM) and propose several strategies to achieve predictable performance, meaning the processing time is highly correlated with the total work and independent of the underlying structure of the input matrices.

The primary challenge of computations involving sparse matrices originates from workload imbalances induced by data segmentation. To address irregularity many methods have been introduced that structure memory accesses to improve the performance of sparse matrix operations. Arguably the most studied and optimized, in terms of data-structures and algorithms, is SpMV. Peak SpMV performance is achieved by coupling sparsity pattern analysis with specialized, and in some cases exotic, storage schemes tuned for a particular class of matrices. The drawbacks of this approach are the need to perform analysis of the input matrix and the use of storage formats that are not amenable for use in other operations. For a particular set of matrices a specialized algorithm-storage pair may yield optimal performance on a given architecture but a highly tuned pair may be invalid or exhibit unpredictable behavior when applied to general matrices. We consider these approaches as sparsity or segmentation aware since they incorporate matrix structure into the implementation.

In this work we avoid the use of segmentation aware algorithms or specialized data structures. Our approach places load-balancing first and performs segmented row-wise operations progressively by introducing intermediate work. To achieve this we study balanced decompositions of each sparse matrix workload and perform processing at the granularity of nonzero entries during all phases. In the case of SpAdd and SpGEMM our processing scheme builds on highly regular merge-based sorting routines to partition work into smaller units of roughly equal size for parallel processing. In contrast with traditional metrics of interest, such as giga-floating point operations per second (GFLOPs/s), our method is not designed to achieve the best absolute performance compared to other segmentation aware algorithms. Indeed we show the performance of our approach may be notably lower than other methods in some cases. However, we also show that our method is easily predictable for a wide diversity of inputs. The contributions of this work are as follows:

- Balanced decompositions of three sparse matrix kernels: SpMV, SpAdd, and SpGEMM.
- Extension of merge-path partitioning to perform set unions.

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.98

II. Background

Modern GPUs are representative of a class of processors that seek to maximize the performance of massively data parallel workloads by employing hundreds of hardware-scheduled threads organized into tens of processor cores. Such architectures are considered throughput-oriented in contrast to traditional, latency-oriented, CPU cores which seek to minimize the time to complete individual tasks [1]. GPUs organize processors into a hierarchy of computational blocks starting from a fixed size group of threads, known as warp, executing in a SIMD fashion. Warps are aggregated into large blocks sharing a fast local memory buffer, shared memory, executing within a thread block or concurrent thread array (CTA). CTAs are further grouped into a larger set, known as a grid, executing a single program. This hierarchical execution model exhibits the highest utilization when computations are regularly-structured and evenly distributed among CTAs within a grid.

Sorting is a important class of computational algorithms and efficient GPU implementations of radix and merge sort have been presented [2], [3]. Though radix sorting an array of N elements, with an average key-length k, may achieve lower complexity bounds, $\mathcal{O}(kN)$, than comparison based methods, $\mathcal{O}(N\log N)$, it has limited applicability and fails to exploit approximate sorted-ness of the input sequence. Recently merge path has been introduced as a efficient means of improving the performance of merge based sorting algorithms [4], [5]. Merge path, illustrated in Figure 1a, operates by partitioning the inputs, A and B, among parallel threads in a way that provides:

1) Equal amounts of work for all threads.
2) Minimal communication and synchronization between parallel threads.

SpMV operations are at the core of many sparse iterative solvers and as such have been the focus of a large body of research to optimize performance [6], [7]. Bell et al. give a comprehensive overview of SpMV considerations on the GPU and propose a novel hybrid format to balance the trade-off of performance and generality [8]. Subsequent research on GPU SpMV have considered sparsity aware adaptive processing and various algorithmic enhancements to reduce memory traffic [9]–[11].

Sequential SpGEMM algorithms [12], rely on a large amount, $\mathcal{O}(n)$, of temporary storage to efficiently store and reduce unique entries on any row of the output matrix $C \in \mathbb{R}^{m\times n}$. Although techniques have been developed to expand the scalability and generality of such parallel decompositions [13], the level of parallelism remains too coarse-grained for GPU acceleration. One novel GPU SpGEMM algorithm decomposes the operation into expansion, sorting, and compression (ESC) [14]. The ESC algorithm was improved by processing C row-wise according to the distribution of work, which is a measure of the number of products, computed during analysis of the input matrices [15]. Though highly efficient for sparse-matrix pairs generating intermediate products readily processed in shared memory their approach did not impact the performance of long intermediate rows processed in global memory. To address long intermediate rows Liu et al. presented a merge-based processing scheme that distributed entries on long intermediate rows evenly for parallel processing in shared memory which achieved notable performance improvement for certain matrices [16].

In this work we explore segmentation oblivious methods to process general reductions on sparse matrices in order to reduce the impact of irregularity while maintaining performance. For complicated decompositions involving sparse matrix pairs, SpAdd and SpGEMM, we leverage the merge path strategy to exploit the partial ordering of the input and perfectly balance processing of the intermediate work across CTAs.

III. Sparse Matrix Operations

Given two matrices $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$, and a vector $x \in \mathbb{R}^{p\times 1}$ then SpMV, SpAdd, and SpGEMM compute Ax, $A + B$, $A \times B$, respectively. In contrast with general (dense) matrix-matrix operations, A and B are assumed to be sparse. Typical storage formats include coordinate (COO), where each nonzero is represented as a `(row, column, value)` tuple, and compressed sparse formats that sort and compress the COO format along either the row (CSR) or column (CSC) axis and provides offsets to the first entry in each segment. We consider the following sparse matrices in our algorithmic descriptions for Sections III-A, III-B and III-C,

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 20 & 30 & 40 \\ 0 & 0 & 0 & 50 \\ 0 & 60 & 0 & 0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 3 \\ 4 & 5 & 0 & 0 \\ 0 & 6 & 0 & 7 \end{bmatrix},$$

with tuple form given by

$$A = \begin{bmatrix} (0,0,10) \\ (1,1,20) \\ (1,2,30) \\ (1,3,40) \\ (2,3,50) \\ (3,1,60) \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} (0,0,1) \\ (1,1,2) \\ (1,3,3) \\ (2,0,4) \\ (2,1,5) \\ (3,1,6) \\ (3,3,7) \end{bmatrix}.$$

A. SpMV

For SpMV we consider $Ax = y$, where A is in CSR format and $y \in \mathbb{R}^{m\times 1}$, and the total work is two times the number of nonzeros in A, denoted $|A|$. The obvious parallelization, known as scalar CSR, is to assign one thread

to each row of A and process each row independently. Unfortunately this work decomposition may exhibit a large amount of imbalance between threads if the number of nonzeros per rows of A vary significantly. One solution is to vectorize the row-wise processing scheme and enlist a fixed number of threads within a warp or CTA to process each row of the A. This scheme is advantageous when: the average number of entries per row does not vary significantly, each row contains a relatively large number of entries relative to the number of threads, and there are enough rows to fully saturate the device. However, it may be impossible to globally select a static number of threads appropriate for every row of A.

We avoid possible irregularities associated of row-wise processing by assigning a fixed number of nonzeros, N_{prods}, per CTA. To implement this strategy we decompose the SpMV kernel into three phases: partition, reduction, and update. Although a flat nonzero based decomposition is perfectly balanced row-wise segmentation is required to enact the reduction correctly. Processing the matrices in COO format is one alternative but requires the additional storage and movement of one row entry per nonzero. We address this issue, for sparse matrices in CSR format, during the partitioning phase which computes the row-wise limits of the entries processed per CTA. For $|A|$ entries we process the products using $m = \lceil |A|/N_{prods} \rceil$ CTAs. During the partitioning phase we enlist m threads to cooperatively perform one binary search per CTA segment to locate the last offset in the row offsets array processed by each CTA and storing these offsets in an auxiliary global memory buffer, S.

After partitioning the rows the reduction phase launches m CTAs to reduce products within each segment of A. A given CTA, i, processes entries in the range $[i * N_{prods}, \min(|A|, (i+1) * N_{prods})]$ and row offsets corresponding to $S[i, i+1]$ are loaded into shared memory to generate the expanded row indices corresponding to each nonzero. To compute the reduction we first load the column indices and values of A, in strided order to reduce the penalty of global memory operations, into register. Then we dereference entries from x according to the column indices and compute the scaled products. The products are transposed from strided to blocked order using shared memory and a CTA wide segmented scan is performed. On encountering discontinuities in the row indices partial sums are stored to y and the CTA remainder, corresponding to the last row, is stored to a global memory buffer, r.

Lastly, during the update phase we accumulate the inter-CTA updates by performing a segmented scan on r and augmenting the first reduction for each CTA with the carry out value from the previous CTA. This approach exposes parallelism without regard for segment geometry allowing it to scale over a wide range of sparsity patterns. We statically tune the number of entries per thread empirically to maximize throughput.

Algorithm 1: Tuple Ordering

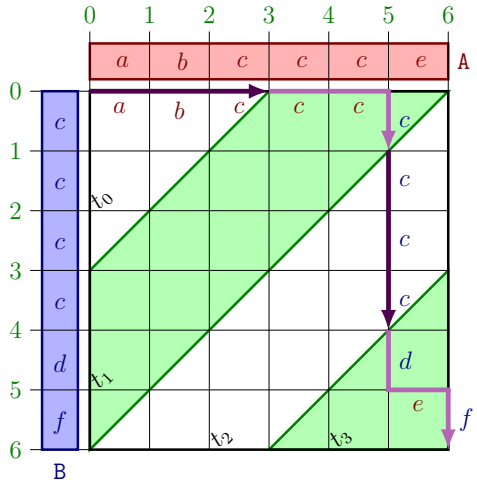
parameters: T1=($\texttt{row}_1$,$\texttt{col}_1$), T2=($\texttt{row}_2$,$\texttt{col}_2$)
return: MIN(T1,T2)
if $row_1 < row_2$
  return T1
if $row_2 < row_1$
  return T2
if $col_1 <= col_2$
  return T1
return T2

B. Addition

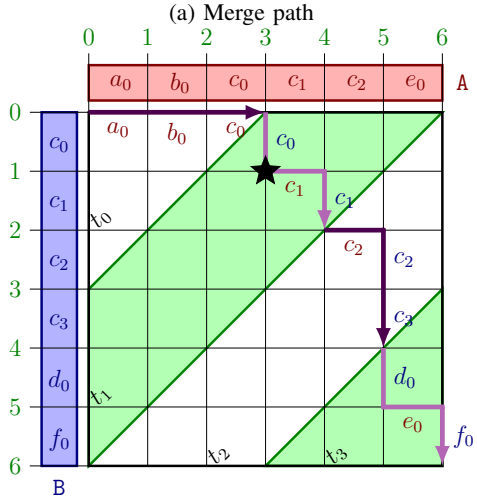
For SpAdd we consider the total work to form C as the sum of the number of nonzero entries in A and B, $|A| + |B|$. Two approaches for computing SpAdd are row-wise segmented reductions and global sorting. In the row-oriented scheme each row of C is processed by a thread group. Though the decomposition of work is simple and processing any output row i requires $\mathcal{O}(|A_{row_i}| + |B_{row_i}|)$ work, it is challenging to choose the number of cooperative threads per row. Conversely, with respect to the global sorting scheme the total work to generate C is considered collectively as a single monolithic unit by combining nonzero entries from both A and B into a intermediate matrix $\hat{T}$. By lexicographically sorting $\hat{T}$ by $(\texttt{row}, \texttt{column})$ tuples all duplicate entries are adjacent and performing a simple tuple-wise reduction generates C. Though not susceptible to penalties associated with row-wise irregularities of A or B the complexity of globally sorting $\hat{T}$ is $\mathcal{O}(k(|A| + |B|)$ which is k times more expensive than the accumulated cost of the row-wise approach, $\mathcal{O}(|A| + |B|)$.

Ideally a method to construct C should mitigate the impact of load imbalance while attaining near optimal work complexity. To achieve this requires a decomposition of work composed of non-overlapping ranges from A and B such that each thread processes a unique set of entries in C, thus corresponding to $\mathcal{O}(|A| + |B|)$ work complexity. We observe that when both input matrices are sorted by row and row-wise internally sorted by column then sparse matrix addition may be formulated as a more general operation requiring the union of sets. Considering the natural comparison of tuples outlined in Algorithm 1 and the ordering of the input matrices then parallel merging using merge path appears to be a natural fit for SpAdd.

We forgo a detailed description of the merge path algorithm and provide only a short introduction but refer to Green et al. for additional information [5]. Merge path decomposes the work of merging two sorted list by performing a binary search along the diagonals formed by orienting the lists on the x and y axes as shown in Figure 1a. The ordering



(a) Merge path



(b) Balanced path

Figure 1: Comparison of decision paths for merge path and balanced path given two sorted sets A and B having six elements each. The paths, and thus the outputs, are evenly partitioned among four threads. For balanced path, the partition boundary between thread t0 and t1 is starred so that zipped pairs are never split across threads.

of entries along diagonals prescribe a partition of both lists into non-overlapping regions of uniform size that can be processed in parallel by individual threads. Though SpAdd appears merge-like, parallel merge path partitioning is inadequate. Each consecutive range of matching $(\texttt{row}, \texttt{column})$ tuples must be available to the same thread and therefore always appear on the same side of any diagonal. We therefore introduce balanced path partitioning, illustrated in Figure 1b, to extend the merge path decomposition to partition the inputs according to this additional constraint.

We describe the balanced path methodology by considering two sorted lists, A and B, consisting of duplicate keys in place of sparse matrix tuples. The normal merge path

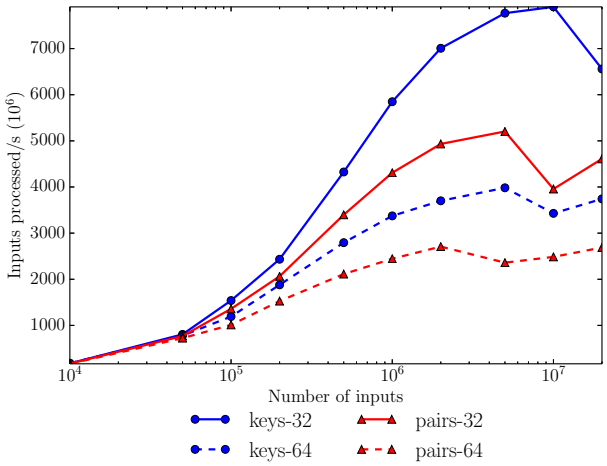


Figure 2: Performance of union operation on sorted sets. Entries per input array are divided evenly.

implementation consumes all duplicate keys in A before matching entries in B therefore constructing a merged result with duplicate entries. In contrast balanced path partitioning assigns a rank to duplicate keys within each duplication range of the input sequences. This enables the assignment of keys with matching ranks to the same partition for serial merging within individual threads therefore ensuring the key-rank pairs emitted by each thread during construction of the result is a unique non-duplicated member of the output list. In Figure 1b the stair-step pattern of balanced path snaking its way through keys for two sorted lists highlights the unique differences between the decompositions where we see a starred diagonal indicating a translation of thread t1's partition to include a matching key from B.

As in merge path the balanced path diagonals begin at fixed intervals. However, the intervals may shift to consume one additional or fewer elements than the prescribed number processed per thread to ensure both elements of a matched key-rank pair are found on the same side of a diagonal partition. Diagonals, mapping to entries $(i, diag - i)$ in A and B, that need to be extended to include an additional element from B, corresponding to a matched pair, are starred. The starred balanced path diagonal now maps to matched entries $(i, diag - i + diag_\star)$ in A and B therefore stealing one entry from the right partition and depositing it into the left partition.

Note that performing this key-rank decomposition allows the implementation of not only unions but many other set operations, such as intersection, difference and symmetric difference [4]. For SpAdd we focus on the union of sets therefore the output list is comprised of only zeroth ranked entries from A and B. In Figure 2 we illustrate the performance of our set oriented union operation on various array sizes and data types. Using our balanced path approach we perfectly decompose the work required to identify and

reduce duplicates.

With respect to SpAdd we reduce the global memory overhead by computing the union of the matrices in two phases. During the first phase the unique tuples in the union of A and B are counted and space for C is allocated. In the second balanced path invocation the entries from each matrix are loaded into shared memory and duplicate values are reduced within the CTA. Note that although for well-formed sparse matrices A and B there are at most two duplicates that must be reduced to form each unique entry of C our implementation is general enough to process a much wider range of input sets consisting of an arbitrary number of duplicates.

C. Multiplication

SpGEMM is substantially more challenging than SpMV and SpAdd because of the added irregularity of the workload to dynamically expand, during formation of the products, and contract, during reduction of duplicates. If we consider the row-wise products expanded for our simple example matrices we see the following intermediate representation:

$$\hat{C} = \begin{bmatrix} (0,0,\ 10) \\ (1,3,\ 60) \\ (1,1,\ 40) \\ (1,1,\ 150) \\ (1,0,\ 120) \\ (1,3,\ 280) \\ (1,1,\ 240) \\ (2,3,\ 350) \\ (2,1,\ 300) \\ (3,3,\ 180) \\ (3,1,\ 120) \end{bmatrix} = \begin{bmatrix} (0,0,\ 10) \\ (1,0,\ 120) \\ (1,1,\ 40) \\ (1,1,\ 150) \\ (1,1,\ 240) \\ (1,3,\ 60) \\ (1,3,\ 280) \\ (2,1,\ 300) \\ (2,3,\ 350) \\ (3,1,\ 120) \\ (3,3,\ 180) \end{bmatrix} = \begin{bmatrix} (0,0,\ 10) \\ (1,0,\ 120) \\ (1,1,\ 430) \\ (1,3,\ 340) \\ (2,1,\ 300) \\ (2,3,\ 350) \\ (3,1,\ 120) \\ (3,3,\ 180) \end{bmatrix},$$

which reduces to the sparse matrix

$$C = A \times B = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 120 & 430 & 0 & 340 \\ 0 & 300 & 0 & 350 \\ 0 & 120 & 0 & 180 \end{bmatrix}.$$

This depiction of SpGEMM decomposes the FLOPs into two phases, expansion and contraction. We consider the total work as a function of the number of products, N_{prods}, in the expansion phase since it is a measure of the total number of global memory operations required to fetch data from the input matrices and it also describes the number of insertions/reductions required to form C. By far the identification and reduction of duplicates during the contraction phase is the most challenging phase of the computation and forms the primary performance limiting routine in many SpGEMM implementations.

To combat the irregularity of contracting the intermediate matrix we propose a balanced approach based on merge path partitioning of the total number products and split

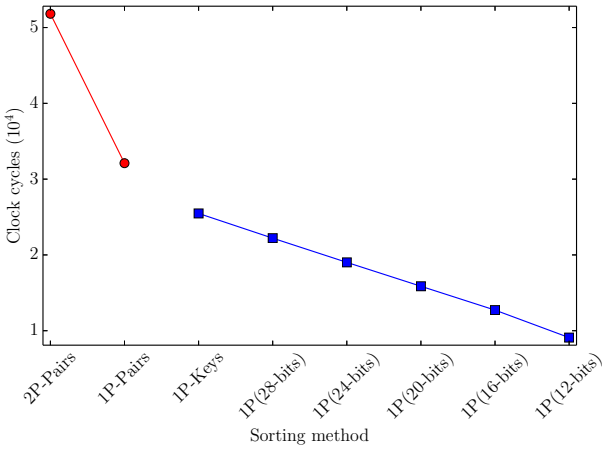


Figure 4: Clock cycles per CTA radix sorting operations for two-pass (2P) key-value pairs (red), one-pass (1P) key-value pairs, and one-pass keys-only routines as the number of sorting bits are decreased (blue).

the processing into separate but performance stable sorting phases, the first phase performed in shared memory and a second phase performed in global memory. To partition the products we first specify the number of products that may be processed per CTA, N_{CTA}, then we compute the segmented prefix-sum, S, of the size of each row of B, $|B_{row_i}|$, referenced by the column indices of A. Following this preprocessing phase $\left\lceil \frac{N_{prods}}{N_{CTA}} \right\rceil$ CTAs are launched to independently expand and process N_{CTA} products. Each CTA locates the range of products it is assigned by performing a binary search on S to locate the first corresponding entry nonzero entry from A greater than or equal to its CTA number times N_{CTA}. By decomposing A at the granularity of products our approach balances the work perfectly across CTAs irrespective of the row-wise reductions dictated by the input matrices.

The first sorting phase reduces duplicates within each CTA using radix sort. Our key observation is that because we expand entries in order according to column entries of A then contiguous entries within a CTA remain ordered by row and duplicates are in adjacent locations following a single radix sort on the column indices as illustrated in Figure 3, this is in contrast to two phase sorting approaches [14]. Figure 4 illustrates the performance improvement achieved by reducing the total number of radix-sort operations within a CTA. We benchmark the performance using the radix-sort routine in the open-source CUB programming framework [17] for 32-bit data types with 128 threads per CTA and 11 entries per thread. For key-value pairs we observe that performing a single radix-sort pass reduces the number clock cycles by approximately a factor of two yielding notable performance improvement.

In Figure 4 we also compare the performance of radix-

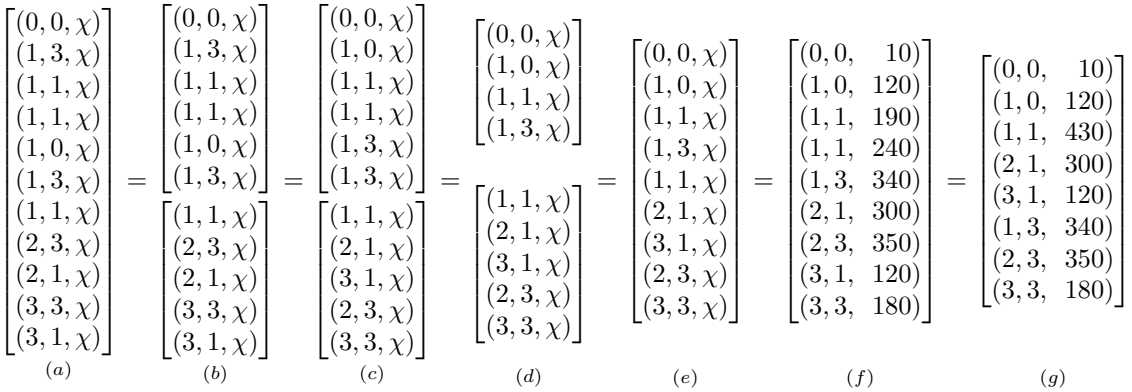


Figure 3: All of the intermediate indices are formed, (a), and partitioned into subsets of approximately equal size, (b). χ represents unformed products corresponding to each intermediate entry. Each subset is sorted by column index independently, (c), and duplicate entries are reduced, (d). All the subsets are aggregated into a intermediate matrix that contains duplicates but is not sorted by row or column, (e). The reduced matrix is sorted, (f), the products of the reduced entries are computed and the remaining duplicates are reduced to form C, (g).

sort when the number of sorted bits are reduced from 28 to 12 bits. Based on this observation we aggressively optimize our sorting implementation to minimize the processing time by exploiting the limited range of the column indices and sorting the minimum number of bits required by computing $\lceil \log_2(n) \rceil$, the number of columns in B. We further optimize our performance by embedding the permutation bits in the unused upper range of the column indices when possible. Using this approach we reduce the number of shared memory operations and perform a keys-only CTA radix-sort. The permutation computed by each block is then stored to global memory using 16-bit integers to minimize the memory traffic and storage overhead. The first phase concludes with each CTA scanning the sorted entries to identify unique entries and asynchronously storing the reduced set to global memory.

Note that following the first phase no products have been formed and the reduced intermediate entries in global memory consist of possible duplicates that are completely unordered. To compute the unique duplicates in C we perform a two pass global radix-sort routine similar to the method outline by the ESC algorithm. The notable difference is that the global memory sorting routine computes only the permutation that sorts the pairs and does not perform any reordering of the, currently unformed, intermediate products.

In the third phase the reduced intermediate products are formed by performing the expansion phase a second time. In this version each CTA loads the permutation computed during the first expansion and a segmented scan is performed to reduce permuted products according to the precomputed duplicate indicators. To reduce the number of global memory load and store operations the reduced values are not stored randomly back to global memory. The output location that orders the reduced products according to global duplication, computed during the second phase, is used to store entries in sorted order. The final step consists of forming the values in C by performing a global memory reduce-by-key operation on the ordered products.

A key feature of this two-level decomposition is that both the CTA-wide and global memory sorting operations are oblivious to the irregularity of the underlying input matrices. By preprocessing many of the duplicates in shared memory prior to the global memory pass the total work required to perform the expensive two-pass radix-sort operation is significantly reduced in cases where each block yields only a small number of unique pairs, i.e., a large number of duplicate entries per CTA.

IV. Numerical Results

In this section we evaluate the performance of our parallel processing strategy and compare our results with two packages for processing sparse matrix operations on GPUs, Cusp, open-source, and Cusparse, closed source. Our evaluation is performed on a set of SpMV matrices taken from the University of Florida (UFL) sparse matrix collection outlined in Table II and the configuration of our testing environment is described in Table I. All of the performance data was collected using double precision arithmetic with the input matrices resident in GPU memory.

CPU	i7-3820 CPU	3.60GHz
GPU	GTX Titan	0.88GHz
CUDA	6.5	
GCC	4.8	-O3

Table I: System configuration settings. On the GPU error checking and correction (ECC) is disabled.

Matrix	rows	columns	nonzeros	avg/row	std
Dense	2000	2000	4 000 000	2000.00	0.00
Protein	36 417	36 417	4 344 765	119.31	31.86
Spheres	83 334	83 334	6 010 480	72.13	19.08
Cantilever	62 451	62 451	4 007 383	64.17	14.06
Wind Tunnel	217 918	217 918	11 634 424	53.39	4.74
Harbor	46 835	46 835	2 374 001	50.69	27.78
QCD	49 152	49 152	1 916 928	39.00	0.00
Ship	140 874	140 874	7 813 404	55.46	11.07
Economics	206 500	206 500	1 273 389	6.17	4.44
Epidemiology	525 825	525 825	2 100 225	3.99	0.08
Accelerator	121 192	121 192	2 624 331	21.65	13.79
Circuit	170 998	170 998	958 936	5.61	4.39
Webbase	1 000 005	1 000 005	3 105 536	3.11	25.35
LP	4284	1 092 610	11 279 748	2632.99	4209.26

Table II: Unstructured matrices taken from the UFL sparse matrix collection for performance testing.

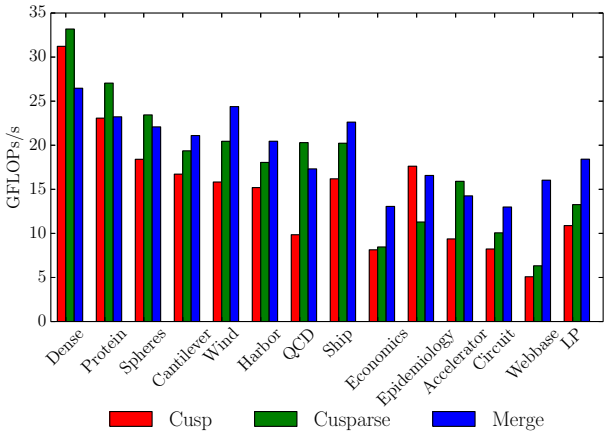


Figure 5: Sparse matrix-vector performance, measured in GFLOPs/s, comparison for CSR format in double precision for three implementation using Cusp, Cusparse, and Merge.

A. SpMV

In Figure 5 we compare the SpMV performance of three different implementations operating on an input matrix stored in CSR format. Although the performance for some matrices is substantially higher using specialized storage formats our goal is to compare the achieved performance using a general and standard storage format. For each matrix the performance reported is the average of 100 iterations. The Cusp SpMV implementation uses the vectorized CSR implementation discussed in Section III-A. In the Merge implementation we detect the presence of empty rows in the input matrices and adaptively switch between a faster method that assumes there are no empty rows and a slightly slower method that compacts the CSR row offsets prior to performing each SpMV operation.

From Figure 5 we see that the relative performance between the three implementations varies with respect to each matrix. We note that the performance of our Merge scheme remains competitive with the best cases in all tests except the Dense matrix. For challenging matrices that exhibit a large degree of irregularity in the number of entries per row, Webbase and LP, we see that our flat decomposition achieves markedly better performance compared to the other implementations. To illustrate the connection between the required work, proportional to $|A|$ for SpMV, and the performance we plot the processing time versus $|A|$ for each matrix in Figure 6. From Figure 6 we see that the performance of the Merge approach is highly correlated, with a correlation coefficient of 0.97, with number the number of nonzeros in each matrix and exhibiting only small perturbations from the least-squares fit of the data points.

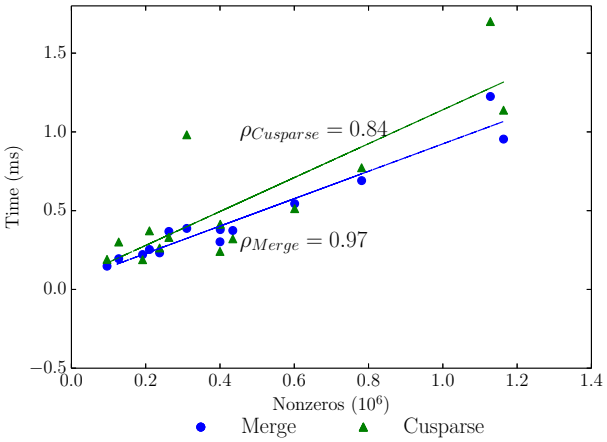


Figure 6: Comparison of SpMV performance to $|A|$. We use the correlation coefficient , ρ, as a measure of the performance predictability as a function of $|A|$. For our Merge scheme ρ is 0.97 indicating that predicting the SpMV performance of untested matrices should be relatively accurate.

B. SpAdd

In Figure 7 we compare SpAdd, $A + A$, performance for the previously mentioned packages. The performance illustrated is the speedup with respect to the sequential implementation using CSR format on the CPU. We note that the storage format for these tests are different between the packages. The Cusp and Merge approaches operate on the input matrices stored in expanded COO format to facilitate processing at the granularity of individual nonzero entries while Cusparse operates on the matrices in CSR format. The Cusp implementation uses the two-pass global sorting approach described in Section III-B.

In contrast with SpMV we see that both Cusparse and Merge perform significantly better than Cusp on all matrices in the test suite. Another notable difference is the significant performance improvement of Cusparse over the Merge implementation for the Dense, Protein, and Wind matrices. However, for the remaining matrices in the test suite the performance of the two methods are comparable with the exception of the Webbase and LP matrices.

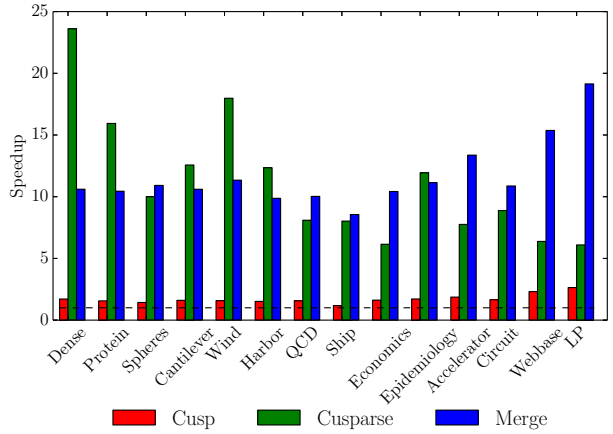


Figure 7: SpAdd performance, measured in speedup versus sequential CPU implementation, for Cusp(COO), Cusparse(CSR), and Merge(COO).

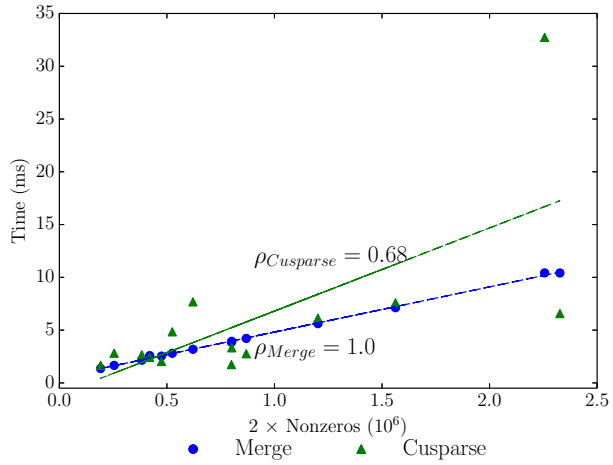


Figure 8: Comparison of SpAdd performance to the number of nonzeros. For the merge implementation our correlation coefficient of 1 implies a nearly perfect match between the work and processing time irrespective of the underlying structure of the input matrices.

In Figure 8 we again plot the performance versus the total work to understand the correlation between the required work and the total processing time. Not surprisingly the Merge approach is perfectly correlated with the number of nonzeros in the input matrices. This is expected since the approach is based on parallel decompositions that yield perfect balance irrespective of the segmentation of the underlying data. Note that for two of the largest SpAdd instances Cusparse achieves speedup for one and dramatic slowdown for the other compared to the Merge implementation.

C. SpGEMM

In Figure 9 we compare SpGEMM, $A \times A$, performance for all three implementations. In the special case of the

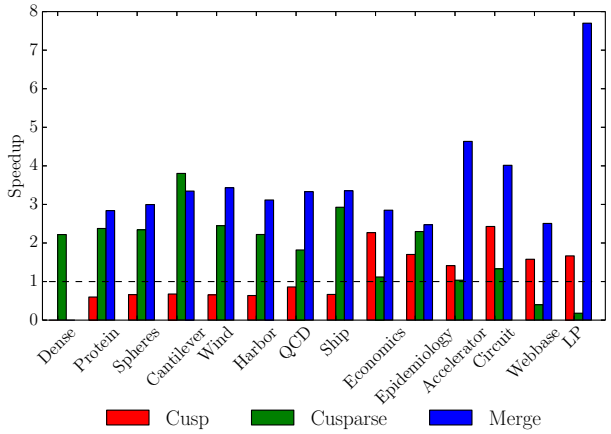


Figure 9: SpGEMM performance, measured in speedup versus sequential CPU implementation, for double precision for Cusp(COO), Cusparse(CSR), and Merge(CSR).

nonsquare LP matrix we transpose the matrix and perform $A \times A^T$. The performance is tabulated in terms of the average speedup for 10 iterations versus the sequential CPU implementation in CSR format. The Cusp implementation uses a global memory ESC algorithm and although the performance is inherently independent of the input matrices the absolute time is high because of global operations involving the total number of products, specifically radix sort. For many of the matrices the Cusparse and Merge approaches outperform Cusp by a significant margin. However, the performance of Cusparse degrades for the Economics, Circuit, Webbase, and LP matrices. In contrast the Merge approach sustains performance improvement compared to Cusp in all instances.

The weakness of sort based SpGEMM methods operating on intermediate products is also illustrated in Figure 9 where both the Cusp and Merge approaches required more physical memory than the resource constrained GPU could support. For the Dense test case the duplicates per CTA is close to zero therefore the global memory pass requires a significant amount of temporary storage to order the expanded entries. For matrices with a relatively dense number of entries per row segmented processing presents a notable advantage. Although both the Webbase and LP matrices also have rows containing a relatively large number of entries the power-law distribution of the row lengths in the case of Webbase and the small number of rows in LP, since it is computing $A \times A^T$, ensures that on average many of the CTAs contract a substantially large number of intermediate entries during the first sorting pass.

In Figure 10 we plot the performance of the Merge and Cusparse implementations versus the total number of products required to form C. Note that this analysis ignores the FLOPs performed to reduce duplicate entries to form C. From Figure 10 we see that the performance of the two-

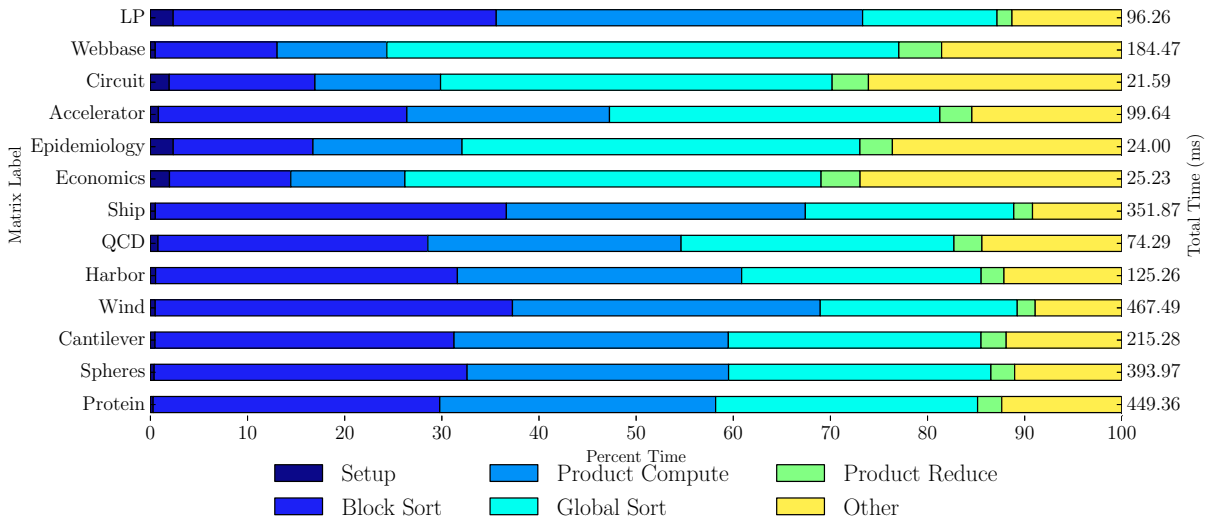


Figure 11: SpGEMM performance breakdown.

level sorting approach based on merge path decompositions is highly correlated with the total number of products. This regularity implies that the performance of the Merge approach is easily predictable based on simple analysis of the input matrices prior to execution of the operation. The observed regularity in the Merge scheme performance is in contrast with Cusparse which diverges considerably from the anticipated performance in two cases. We do not claim the Cusparse implementation is inherently unpredictable but that the performance appears to be unrelated to our interpretation of the required work, the number of products.

In Figure 11 we decompose the performance of the merge SpGEMM performance into several phases. During the Setup phase the number of entries on each row of B referenced by each column of A is scanned to compute the number of products required for each nonzero entry in A. During Block Sort the products are decomposed into a fixed number of entries and processed using a single radix-sort pass within each CTA and the resulting permutation is stored to global memory. The Global Sort phase organizes the reduced number of entries identified by each CTA using two-radix passes and the Product Compute phase combines the local CTA permutation with the global duplicate permutation to form the ordered reduced products in global memory. In the last phase the final products are formed by performing a reduce-by-key operation on duplicates in global memory. Additional overheads in terms of allocation and miscellaneous memory operations are aggregated into Other. From Figure 11 it is clear that the two sorting passes and the computation of the output products constitute the bulk of the processing time for each matrix. Although there are significant variations in the total relative time of each operation based on the matrix it is important to consider these variations with respect to the total processing time, shown on the right axis.

V. Conclusion

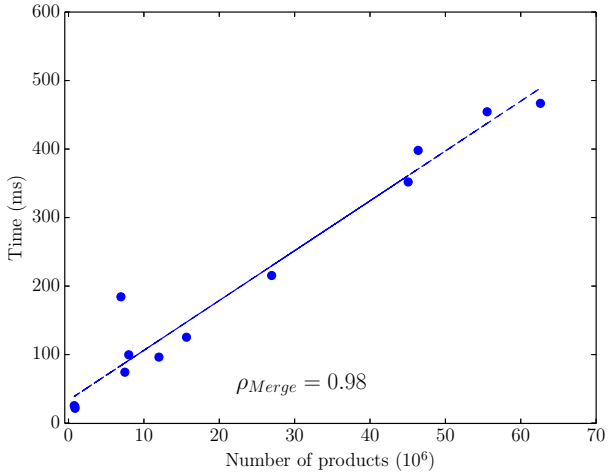
Although the specific techniques employed to perform each sparse matrix operation varied substantially the focus of our study was the performance implications of using a flat decomposition of the work irrespective of the underlying segmentation dictated by the input operands. Note that our method does *not* achieve the best performance in all cases for any of our tests. Indeed, we expect when comparing any two algorithms processing such irregular computations that subtle data-dependent performance characteristics may have a large impact on the overall processing time. Our primary focus in this work is achieving predictable performance across a vast landscape of input instances.

In future work we plan to address the deficiencies of sort based SpGEMM methods by adaptively introducing segmented approaches when necessary. Detecting specific cases like the Dense matrix is relatively simple but would also require a more detailed model to accurately predict the trade-off between the number of entries processed in the segmented and unsegmented regions of the algorithm.

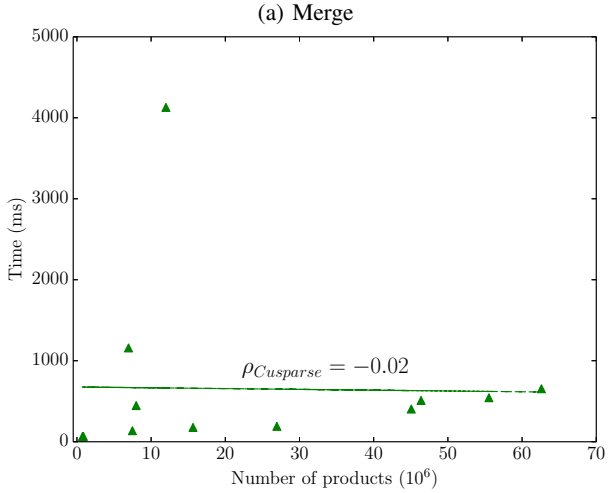
References

[1] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures," *Commun. ACM*, vol. 53, pp. 58–66, November 2010. [Online]. Available: http://doi.acm.org/10.1145/1839676.1839694

[2] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens, "Efficient parallel merge sort for fixed and variable length keys," in *Innovative Parallel Computing*, May 2012, p. 9.



(a) Merge



(b) Cusparse

Figure 10: Comparison of SpGEMM performance to the number of products. Flat two-level sorting results in performance that is highly correlated, $\rho = 0.98$, with the work expressed as a function of number of products in the intermediate matrix.

[3] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011. [Online]. Available: http://www.worldscinet.com/ppl/21/2102/S0129626411000187.html

[4] "Mgpu : Design patterns for gpu computing," http://nvlabs.github.io/moderngpu/, version 1.1.

[5] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: A gpu merging algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 331–340. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304621

[6] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, 2003, aAI3121741.

[7] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178 – 194, 2009, revolutionary Technologies for Acceleration of Emerging Petascale Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819108001403

[8] N. Bell and M. 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*. New York, NY, USA: ACM, 2009, pp. 1–11.

[9] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," *SIGPLAN Not.*, vol. 45, no. 5, pp. 115–126, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693471

[10] J. L. Greathouse and M. 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*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 769–780. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.68

[11] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *High Performance Embedded Architectures and Compilers*, ser. Lecture Notes in Computer Science, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds. Springer Berlin Heidelberg, 2010, vol. 5952, pp. 111–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11515-8_10

[12] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, pp. 250–269, September 1978. [Online]. Available: http://doi.acm.org/10.1145/355791.355796

[13] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal of Scientific Computing (SISC)*, vol. 34, no. 4, pp. 170 – 191, 2012. [Online]. Available: http://gauss.cs.ucsb.edu/~aydin/spgemm_sisc12.pdf

[14] N. Bell, S. Dalton, and L. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/110838844

[15] S. Dalton, N. Bell, and L. Olson, "Optimizing sparse matrix-matrix multiplication for the gpu," Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, Illinois, Tech. Rep., February 2013.

[16] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 370–381.

[17] "Cub : Reusable software for cuda programming," http://nvlabs.github.io/cub/, version 1.3.2.