Optimizing Sparse Matrix-Matrix Multiplication for the GPU

Steven Dalton[†] *Nathan Bell*[‡] *Luke N. Olson*[§]

Abstract

Sparse matrix-matrix multiplication (SpMM) is a key operation in numerous areas from information to the physical sciences. Implementing SpMM efficiently on throughput-oriented processors, such as the graphics processing unit (GPU), requires the programmer to expose substantial fine-grained parallelism while conserving the limited off-chip memory bandwidth. Balancing these concerns, we decompose the SpMM operation into three, highly-parallel phases: expansion, sorting, and compression, and introduce a set of complementary bandwidth-saving performance optimizations. Our implementation is fully general and our optimizations lead to substantial efficiencies for a SpMM product.

Keywords: parallel, sparse, gpu, matrix-matrix

1 Introduction

Operations on sparse data structures abound in all areas of information and physical science. In particular, the sparse matrix-matrix multiplication (SpMM) is a fundamental operation that arises in many practical contexts, including graph contractions [11], multi-source breadth-first search [6], matching [23], and algebraic multigrid (AMG) methods [3]. In this paper we focus on the problem of computing matrix-matrix products efficiently for general sparse matrices in data parallel environments.

1.1 Sparse Matrices and Algorithms

While algorithms operating on sparse matrix and graph structures are numerous, a small set of operations, such as SpMM and sparse matrix-vector multiplication (SpMV), form the foundation on which many complex operations are

[†] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, `dalton6@illinois.edu`

[‡] Google, `nathanbell@google.com`, `http://www.wnbell.com`

[§] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, `lukeo@illinois.edu`, `http://www.cs.illinois.edu/homes/lukeo`

built. An analysis of sparse matrix-vector multiplication (SpMV) reveals that the operation has very low computational intensity — i.e., the ratio of the floating point operations (FLOPs) to memory accesses — which severely limits the potential throughput of the operation on contemporary architectures [25, 26]. A common strategy for improving SpMV performance is to exploit *a priori* knowledge of the sparsity structure of the matrix in order to utilize memory access optimizations. Since the cost of reformatting the data is non-trivial, generally on the order of 10-20 SpMV operations, this approach is profitable when the number of subsequent SpMV operations is relatively large.

Although SpMV is a useful starting point for understanding SpMM, we emphasize that the latter is a qualitatively different problem with unique complexities and trade-offs in performance. In particular, whereas the computational structure of SpMV is fully described by the matrix sparsity pattern, SpMM adds another level of indirection and depends on the detailed interaction of two sparse matrices. Indeed, simply computing the number of floating point operations required by the SpMM, or even the size of the output matrix, is not substantially simpler than computing the SpMM itself.

The recent demand for high performance SPMM operations is driven by the increasing size of sparse linear systems [5, 7]. AMG is an important example because the setup phase of the method relies on a sparse triple-matrix product (ie., the Galerkin product). AMG solvers are generally divided into two phases: setup and solve [3]. The relative cost of each phase varies, but the setup phase often represents a significant portion (e.g. 25-50%) of the total solve time. Within the AMG setup phase, the SpMM is the central performance bottleneck, often accounting for more than 50% of the total setup cost as shown in Figure 1. In contrast, the AMG solution phase is comprised of SpMV and level 1 BLAS operations and therefore readily accelerated by employing existing highly-optimized GPU implementations[3].

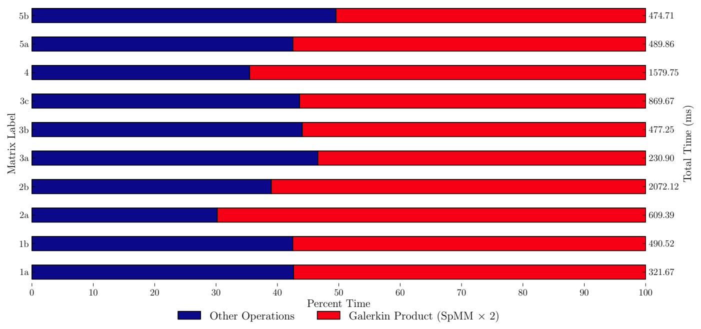


Fig. 1: Relative cost of the SpMM during the AMG setup phase for a series of matrices (see Table 3).

The approach to SpMM in [3] is based on a decomposition of the operation into 3 phases: expansion, sorting, and compression (ESC). The ESC formulation of the SpMM operation is naturally implemented with data parallel primitives, such as those provided by the Thrust parallel algorithms library[14]. In this work we implement a set of optimizations that enhance SpMM performance by exploiting on-chip memory, whenver possible, and reducing the cost associated with the sorting phase.

2 Background

The emergence of “massively parallel” many-core processors has inspired interest in algorithms with abundant fine-grained parallelism. Modern GPU architectures, which accommodate tens of thousands of concurrent threads, are at the forefront of this trend towards massively parallel throughput-oriented execution. While such architectures offer higher absolute performance, in terms of theoretical peak FLOPs and bandwidth, than contemporary (latency-oriented) CPUs, existing algorithms need to be reformulated to make effective use of the GPU [10, 18, 16, 24].

Modern GPUs are organized into tens of multiprocessors, each of which is capable of executing hundreds of hardware-scheduled threads. Warps of threads represent the finest granularity of scheduled computational units on each multiprocessor with the number of threads per warp defined by the underlying hardware. Execution across a warp of threads follows a data parallel SIMD (single instruction, multiple data) model and performance penalties occur when this model is violated as happens when threads within a warp follow separate streams of execution — i.e., *divergence* — or when atomic operations are executed in order — i.e., *serialization*. Warps within each multiprocessor are grouped into a hierarchy of fixed-size execution units known as blocks or cooperative thread arrays (CTAs); intra-CTA computation and communication may be routed through a shared memory region accessible by all threads within the CTA. At the next level in the hierarchy CTAs are grouped into grids and grids are launched by a host thread with instructions encapsulated in a specialized GPU programming construct known as a kernel.

GPUs sacrifice serial performance of single thread tasks to increase the overall throughput of parallel workloads. Effective use of the GPU depends on four key features: an abundance of fine-grained parallelism, uniform work distribution, high arithmetic intensity [26], and regularly-structured memory access patterns. Workloads that do not have these characteristics often do not fully utilize the available computational resources and represent an opportunity for further optimization. In this work we seek to characterize the nature of SpMM and to decompose the computational work to suit the GPU architecture. In particular, by concentrating on the *intersection* of the input matrices and slightly coarsening the degree of parallelism we greatly reduce the number of off-chip memory references to improve the arithmetic intensity of the bandwidth-limited SpMM operation.

2.1 SpMM

Given two sparse matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$, for $k, m, n \in \mathbb{N}$, SpMM multiplication computes

$$C = AB, \tag{1}$$

where $C \in \mathbb{R}^{m \times n}$. We denote $\texttt{nnz}(A)$ as the number of nonzeros in sparse matrix A. The sparsity of A and B implies that both input matrices are represented in a space-efficient format that avoids storing explicit zero values. Although SpMM is related to both the SpMV operation and to dense matrix-matrix multiplication — e.g., GEMM in BLAS — the formulations and optimizations are fundamentally different. Both SpMV and GEMM have achieved near optimal implementations on GPUs through regularization of the data access patterns and algorithmic reformulations [15, 8], approaching the (theoretical) peak limits of memory bandwidth and arithmetic throughput, respectively.

In contrast to GEMM, SpMM operations are highly irregular and may exhibit considerably lower arithmetic intensity. Techniques to improve performance through sparsity pattern analysis, such as those for SpMV, are less effective because SpMM is in general a *fleeting* operation, meaning that they are called at most once for a given set of matrices in most applications. Indeed, whereas the same sparse matrix participates in hundreds of SpMV operations in the context of a single iterative solver, SpMM operations are generally outside the innermost solver loop.

Efficient sequential SpMM algorithms [2, 12], generally operate on sparse matrices stored in the Compressed Sparse Row (CSR) format, which provides $\mathcal{O}(1)$ indexing of the matrix rows, but $\mathcal{O}(\texttt{nnz}(A))$ access to columns. Therefore, these methods focus on constructing the output matrix and accessing both input operands on a per row basis. Sequential methods operate by iterating over the rows of A and for each column entry in each row scaling the values in the corresponding row from B and accumulating the results. To accomplish this the sequential algorithms rely on a large amount, $\mathcal{O}(N)$, of temporary storage. Parallel SpMM algorithms generally decompose the matrix into relatively large submatrices and distribute the submatrices across multiple processors for parallel computation, a strategy used in many computational software packages which use MPI such as Trilinos and PETSc [13, 1].

The reliance in sequential methods on $\mathcal{O}(N)$ storage renders these methods untenable on GPUs, which thrive on workloads in which the per thread state is considerably smaller — i.e., on the order of tens of values. Furthermore, the traditional parallel approaches to SpMM on the GPU require a decomposition of the matrices on a per thread or CTA basis which may be advantageous but require complex decompositions to avoid unnecessarily high imbalances in the work distribution.

The contribution of this work is the study and proposal of a SpMM algorithm which exposes abundant *fine-grained* parallelism and is amenable to execution on the GPU architecture. In particular, we develop parallelism at the level of individual matrix rows and nonzero entries while respecting GPU performance considerations, such as execution divergence and memory locality.

3 ESC Algorithm

A direct implementation of the ESC algorithm using parallel primitives is "work-efficient" and insensitive to the sparsity pattern of the matrices [3]. Although SpMM is highly unstructured and gives rise to complex and unpredictable data access patterns, the ESC algorithm distills the computation into a small set of data-parallel primitives such as `gather`, `scatter`, `scan`, and `stable_sort_by_key`, whose performance characteristics are readily understood [19]. Since the whole method is simply the sum of its parts, and great care is taken when implementing the individual parts, the ESC algorithm with parallel primitives is robust, reliable and efficient. The high-level structure of the ESC algorithm is summarized in Algorithm 1.

```
Algorithm 1: SpMM: Reference
  parameters: $A$, $B$
  return: $C$

1 $M \leftarrow$ slice($A$)                         {decompose rows into slices}
  for $k = 0, \ldots, M$
2     $\hat{C}_k \leftarrow$ expand($A_k, B$)            {expand intermediate matrix}
3     $\hat{C}_k \leftarrow$ sort($\hat{C}_k$)            {radix sort $\hat{C}_k$ by row and column}
4     $\hat{C}_k \leftarrow$ compress($\hat{C}_k$)        {compress duplicate $\hat{C}_k(row, col)$ entries}
5 $C \leftarrow$ construct($\hat{C}$)                {concatenate slices to form final matrix}
```

As an example, consider the matrices

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 20 & 30 & 40 \\ 0 & 0 & 0 & 50 \\ 0 & 60 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 3 \\ 4 & 5 & 0 & 0 \\ 0 & 6 & 0 & 7 \end{bmatrix}, C = AB = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 120 & 430 & 0 & 340 \\ 0 & 300 & 0 & 350 \\ 0 & 120 & 0 & 180 \end{bmatrix}, \tag{2}$$

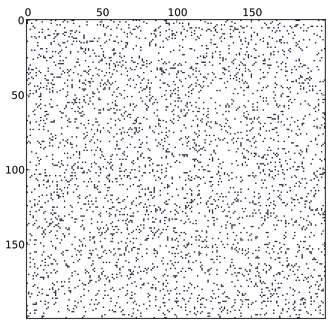
where the COO representation is given by the tuples

$$A = \begin{bmatrix} (0,0,10) \\ (1,1,20) \\ (1,2,30) \\ (1,3,40) \\ (2,3,50) \\ (3,1,60) \end{bmatrix} \quad B = \begin{bmatrix} (0,0,1) \\ (1,1,2) \\ (1,3,3) \\ (2,0,4) \\ (2,1,5) \\ (3,1,6) \\ (3,3,7) \end{bmatrix} \quad C = AB = \begin{bmatrix} (0,0,10) \\ (1,0,120) \\ (1,3,340) \\ (1,1,430) \\ (2,3,350) \\ (2,1,300) \\ (3,3,180) \\ (3,1,120) \end{bmatrix}. \tag{3}$$

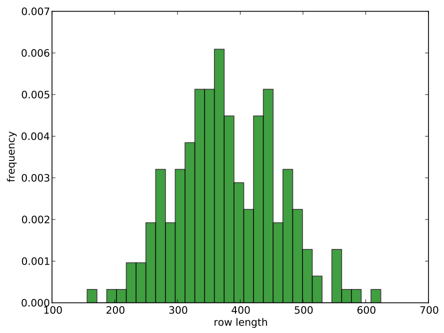
Then the expansion, sorting, and compression phases yield

$$\hat{C} = \begin{bmatrix} (0,0,10) \\ (1,3,60) \\ (1,1,40) \\ (1,1,150) \\ (1,0,120) \\ (1,3,280) \\ (1,1,240) \\ (2,3,350) \\ (2,1,300) \\ (3,3,180) \\ (3,1,120) \end{bmatrix} \xrightarrow[\text{sort}]{} \begin{bmatrix} (0,0,10) \\ (1,0,120) \\ (1,1,40) \\ (1,1,150) \\ (1,1,240) \\ (1,3,60) \\ (1,3,280) \\ (2,1,300) \\ (2,3,350) \\ (3,1,120) \\ (3,3,180) \end{bmatrix} \xrightarrow[\text{compress}]{} \begin{bmatrix} (0,0,10) \\ (1,0,120) \\ (1,1,430) \\ (1,3,340) \\ (2,1,300) \\ (2,3,350) \\ (3,1,120) \\ (3,3,180) \end{bmatrix} = C. \quad (4)$$

Here we see that general sparsity patterns lead to a variety of row lengths in $\hat{C}$. To further illustrate this point consider a sparse random matrix of size $n = 200$ with an average of 20 nonzeros-per-row (see Figure 2a), yielding a minimum and maximum sort length of 156 and 624, respectively, as shown in Figure2b. Here $\mathtt{nnz}(A) = \mathtt{nnz}(B) = 3812$, and $\mathtt{nnz}(C) = 33678$, while $\hat{C}$ contains 75786 entries. In the following, for a sparse matrix A we denote by $A_{\mathtt{row}_i}$ the i^{th} row of A (and similar for columns), while $\mathtt{NNZ}(A_{\mathtt{row}_i})$ denotes the set of nonzero column indices.



(a) Random sparse matrix pattern.



(b) Distribution of row lengths.

Fig. 2: Sparse matrix with $n = 200$ yielding a range of row lengths in $\hat{C}$.

The ESC process for (1) follows from the inner product view of multiplication:

$$C_{i,j} = A_{\mathtt{row}_i} \cdot B_{\mathtt{col}_j} = \sum_k A_{i,k} B_{k,j}. \quad (5)$$

From this we see that simultaneous access to $A_{\mathtt{row}_i}$ and $B_{\mathtt{col}_j}$ is necessary to construct entry $C_{i,j}$. Yet, there are two issues to address when considering the inner product formulation of SpMM: intersecting sparsity patterns and sparse storage formats. Intersecting sparsity patterns requires the categorization of all of the entries in C into zero and nonzero values. The work performed in the SpMM should avoid operations on zero values of $C_{i,j}$, which implies row i of A and column j of B have non-intersecting sparsity patterns. However, to identify non-intersecting sparsity patterns the naïve inner product formulation requires explicit checking of all possible mn entries in C thereby generating excessive data movement when C is also sparse. Another potential problem with the inner product formulation is the access pattern of entries in A and B. As noted earlier, if B is stored in CSR format then accessing entries on a per column basis results in significant overhead and should therefore be avoided whenever possible.

In the following, we consider the basic form of our ESC algorithm [3] in Algorithm 1 as the reference implementation. However, there are several limitations with this approach. First, implementing the operation using parallel primitives forces many data movement operations in global memory between primitives. Moving through global memory between operations ignores more efficient use of registers and shared memory to seamlessly process data from successive phases locally. Second, by staging values in global memory and relying on radix sorting, which is not in-place, to order the intermediate matrix the amount of temporary global memory required by the method is significant. Indeed, to process large matrices we use a decomposition method which partitions A row-wise into multiple slices based upon the maximum amount of available temporary memory to form any slice of $\hat{C}$. Lastly, although radix sorting on GPUs is fast and efficient it is a $\mathcal{O}(kN)$ algorithm, with k representing the number of passes, and requires random accesses to reorder data in global memory. In addition, the costs are compounded by requiring two sorting operations, first by column index and then by row index, to ensure the intermediate format is in proper format for compression.

To motivate where in the ESC algorithm (Algorithm 1) we focus our optimizations, we consider a set of matrices for A that result from a discretization of a Poisson problem, $-\nabla \cdot \nabla u = 0$, with Dirichlet boundary conditions and an average mesh diameter h in the case of unstructured tessellations. The matrices considered are outlined in Table 3, along with several additional test problems previously found in GPU SpMV data sets [3, 4]. Here, cases 1 and 2 are structured, while 3–5 are unstructured tessellations. For matrix B, we generate an interpolation matrix through smoothed aggregation-based AMG[3].

Figure 4 shows the per-phase-cost associated with the reference ESC implementation. Note the negligible overhead in the analysis (setup) phase, where the GPU memory constraints are used to decompose the formation of $\hat{C}$ row-wise, in contrast to the substantial overhead associated with the sorting phase. In the following sections we detail a method of work decomposition to increase the use of shared memory through all phases of the operation and improving the sorting performance by reducing N in the radix sorting algorithm.

Matrix	n	nnz	Matrix	n	nnz
1a. 2D FD	1 048 576	5 238 784	Cantilever	4 007 383	62 451
1b. 2D FE	1 048 576	9 424 900	Spheres	6 010 480	83 334
2a. 3D FD	1 030 301	7 150 901	Accelerator	2 624 331	121 192
2b. 3D FE	1 030 301	27 270 901	Economics	1 273 389	206 500
3a. 2D FE	550 387	3 847 375	Epidemiology	2 100 225	525 825
3b. 2D FE	1 182 309	8 268 165	Protein	4 344 765	36 417
3c. 2D FE	2 185 401	15 287 137	Wind Tunnel	11 634 424	217 918
4. 3D FE	1 088 958	17 095 986	QCD	1 916 928	49 152
5a. 2D FE	853 761	5 969 153	Webbase	3 105 536	1 000 005
5b. 2D FE	832 081	5 817 905			

Tab. 3: Test problems of square matrices ($n \times n$) with nnz nonzeros.

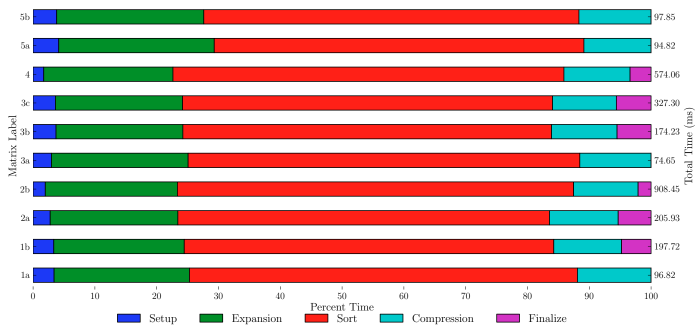


Fig. 4: Component-wise performance of the reference ESC SpMM operation.

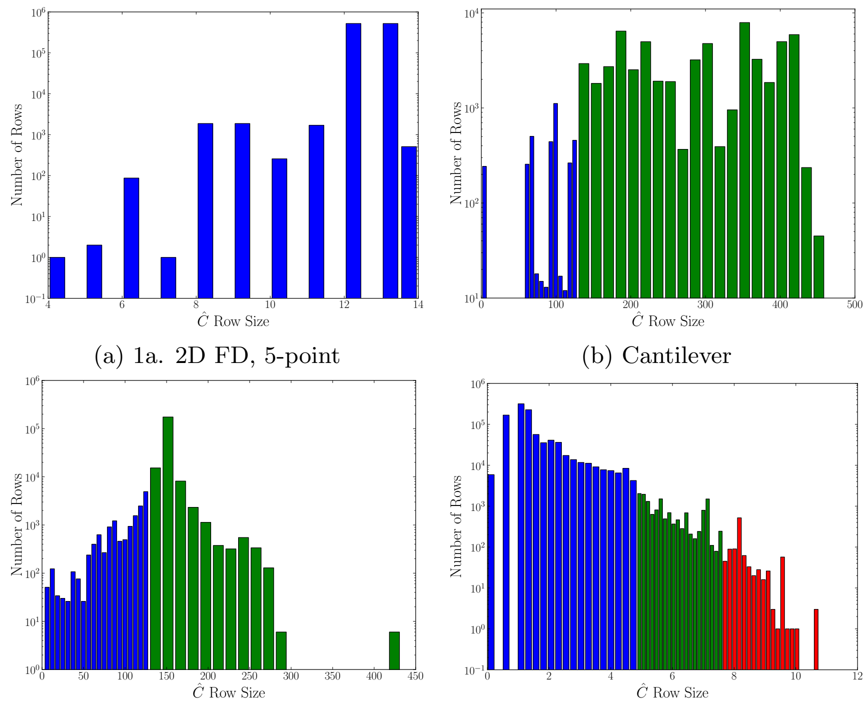
3.1 Analysis

For SpMM, traditional approaches process the input matrices using the natural ordering of the operands and assign a fixed number of threads and memory per row of the output matrix. If C is constructed row-wise then assigning a fixed number of computational units per row of the output matrix may result in significant load imbalance. To illustrate, the minimum number of FLOPs associated with forming $C_{\texttt{row}_i}$ is proportional to

$$\sum_{j \in \texttt{NNZ}(A_{\texttt{row}_i})} \texttt{nnz}(B_{\texttt{row}_j}).$$

This quantity represents the total number of products required to scale each row of B referenced by each column entry within the row.

As depicted in Figure 5, with respect to the products, the computational work per row of C may vary substantially. Consequently, any static assignment of computational units to rows of the matrix leads to arbitrarily poor load balance and possible degradation in performance. One strategy for avoiding this load imbalance is to implement the entire algorithm in terms of parallel primitives [3]. While this approach thoroughly eliminates load imbalances it does so at the cost of significant data movement between stages. An alternative method relies on dynamically scaling computational units to address the data-dependent workloads. In a GPU architecture, the allocation of computational units should account for processing small workloads completely in shared memory as opposed to global memory, and scaling should take advantage of the parallel execution across arbitrary groups of threads within a CTA and multiple CTAs. While ideal, this dynamic scaling is difficult to implement effectively in real hardware. Therefore we use a different approach based on the work distribution model that groups work into several categories that are processed using the most appropriate method.



(a) 1a. 2D FD, 5-point

(b) Cantilever

(c) Wind Tunnel

(d) Webbase

Fig. 5: Distribution of $\hat{C}$ row lengths for SpMM operations in Tables 11,13. Rows are grouped by color: 0-256 (Blue), 257-1024 (Green), and 1025$\geq$ (Red).

We observe that C may be assembled in any order, thus we may permute the input matrices to achieve a grouping of the output rows which yield a favorable use of the computational units. Our scheme is based on reordering the output matrix rows by the amount of computational work in the model. This sorting yields a permutation matrix P for C and implies that $PC = PAB$ which translates into processing the rows of A in permuted order. The permuted order of A groups rows of similar total work and places the rows in non-decreasing order of the work-per-row, Algorithm 2. Identification of rows to be processed by individual threads, warps, or CTAs may be carried out using a model parameterized by the size and number of rows fulfilling predefined conditions. Our strategy is to use a splitting to decompose the rows into units that are processed within a targeted group of threads using parallel primitives. One drawback of this approach is that C is generated in a permuted order and must be sorted by row before the final output is generated. However, in practice we find that this reassembly cost is relatively low.

```
Algorithm 2: SpMM: reorder
parameters: $A$, $B$                                   {$A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$}
return: $P$                                            {reordering vector}

$P \leftarrow 0$
foreach row $i$ in $C$ do
    for $j \in \mathit{NNZ}(A_{row_i})$
        $P_i \leftarrow P_i + \mathtt{nnz}(B_{\mathtt{row}_j})$   {gather $B$ row lengths based on $A$ column indices}
$P \leftarrow \mathtt{sort}(P)$                        {set $P$ to permutation of $P$ in non-decreasing order}
```

3.2 Expansion

As illustrated in Figure 6, the expansion phase expands scaled rows of B into an intermediate buffer. Expanding B row-wise ensures efficient access when the underlying sparse storage format is CSR and all expanded entries contribute to the nonzero entries in C. The expanded memory buffer consists of row indices $\hat{I}$, column indices $\hat{J}$, and values $\hat{V}$, which we collectively denote as $\hat{C} = (\hat{I}, \hat{J}, \hat{V})$. The formation of $\hat{C}$ requires gathering possibly disparate rows from B dictated by the column indices of A. Loading rows of B in an incoherent or random manner limits the benefit of coalescing and therefore precludes fully utilizing the memory bandwidth of the GPU. In particular, if a fixed unit of threads are assigned to load rows from B, and the average row length from is significantly smaller than the number of assigned threads, then many threads are idle or load unrelated entries from adjacent rows. In contrast if the average row length of B is significantly larger than the number of assigned threads, then multiple sequential loading phases are required to process the row.

To address the deficiencies in the expansion phase we adopt a formulation of SpMM as a layered graph model [9]. Each input matrix is represented as a bipartite graph with vertices defined by the individual rows and columns in the matrix. For each nonzero entry in the matrix, a directed edge is created from the row to the column vertices in the bipartite graph. The bipartite graphs are then concatenated — i.e., *layered* — by joining the graphs along the inner dimension vertex sets. The equality of the cardinality of the joined vertex sets is assured by assuming the proposed multiplication is well-posed — i.e., the

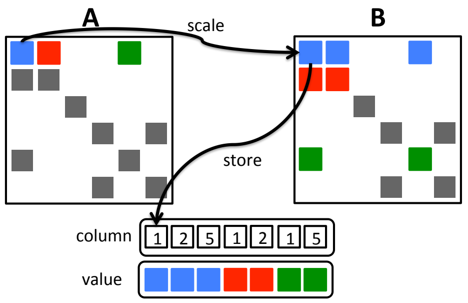
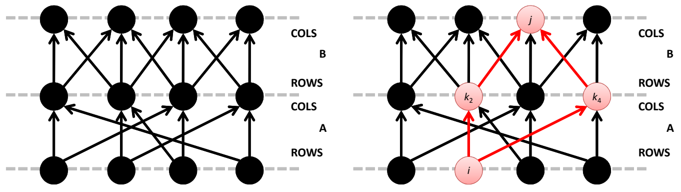


Fig. 6: Scaling rows of B by columns of A and storage in intermediate buffer.

inner matrix dimensions agree. As an example, we illustrate the layered model diagram in Figure 7 using matrices A and B defined as

$$\mathtt{A} = \begin{bmatrix} x & 0 & x & 0 \\ 0 & x & 0 & x \\ 0 & x & x & 0 \\ x & 0 & 0 & x \end{bmatrix} \quad \text{and} \quad \mathtt{B} = \begin{bmatrix} x & x & 0 & 0 \\ x & x & x & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix}. \tag{6}$$



(a) Layered graph model of $A*B$.

(b) Paths contributing to an entry.

Fig. 7: Schematic of graph-based sparse matrix multiplication.

In the layered graph model a nonzero, $C_{i,j}$, in the output matrix corresponds to a path to vertex j in the column set of B from vertex i in the row set of A. There may be an arbitrary number of paths bounded by the product of the maximum in- and out-degrees of the second level intermediate vertices. A weight is attributed to all paths according to a binary operation — e.g., multiplication in the case of SpMM — on the weight of the individual edges traversed by the path. Based on this formulation the expansion phase is an operation on graphs rather than algebraic structures and enumerating the entries which contribute to all output nonzeros is recast as computing all-pairs-all-paths to the column set of B from the row set of A.

By viewing the expansion phase from a graph perspective we see that expansion is a candidate for a breadth-first-search (BFS) of the levels in the layered model. BFS traversals are effectively mapped to GPUs using efficient expansion methods designed to dynamically scale the number of threads expanding the frontier of a single vertex within a CTA [16]. Starting from the source vertices in the layered model — i.e., vertices in the bipartite graph with an in-degree of zero — is unnecessary because the first expansion is implicitly defined as all the column indices in A. Therefore the column indices of A identify the vertices in the frontier from rows of B which must be expanded. However, in contrast to previous BFS implementations [16], the edges in the layered model are weighted. Moreover, although duplicate vertices appear in the frontier, the distinct weights associated with the edges prevent the removal of duplicates, to reduce redundant expansion operations.

Algorithm 3: SpMM: `Expansion`

```
  parameters: $A, B$
  return: $\hat{C} = (\hat{I}, \hat{J}, \hat{V})$

  foreach row $i$ in $C$ do
      for $k \in NNZ(A_{row_i})$                 {Note $A_{i,k}$ is stored in shared memory for reuse}
          for $j \in NNZ(B_{row_k})$
              $\hat{I} \leftarrow [\hat{I},\ i]$                    {implicit row index}
1             $\hat{J} \leftarrow [\hat{J},\ j]$                    {append column index}
2             $\hat{V} \leftarrow [\hat{V},\ A_{i,k} * B_{k,j}]$    {append value}
```

The work in the expansion phase is decomposed at the granularity of one thread per nonzero entry in A. Each thread within the warp or CTA computes the length of the row referenced from B and expansion proceeds using either fine-grained scan-based or cooperative warp or CTA expansion routines[16]. The expansion phase is therefore efficient and the imbalance between CTAs is negligible. To reduce the costs of repeated loading of values from A, each thread stores their entry from A to shared memory. Once the row corresponding to the given thread is expanded the column indices are stored in either local registers in preparation for the impending sorting operation outlined in the following section or streamed to global memory along with the floating point values if global memory processing is necessary. Prior to streaming the floating-point values from shared memory the per-thread values from A are broadcast to shared memory and the entries from B are scaled appropriately. A high-level description of the expansion phase is outlined in Algorithm 3 where the loop over entries in each row of A on lines 1 and 2 is decomposed at the granularity of the thread group, which may be a thread, warp, or CTA.

3.3 Sorting

The expansion phase generates a partially sorted matrix, $\hat{C}$, in coordinate format (see 4) with duplicate entries. Since there are an undetermined number of duplicates for any column entry j within the extent of row i, the partial ordering creates a bottleneck in reducing the duplication. To do this efficiently, $\hat{C}$ is first sorted by row and column, transforming $\hat{C}$ into a sorted format with duplicate entries in adjacent locations. Figure 4 underscores the expense of the sorting performance, which shows it is the dominant cost in the reference ESC algorithm.

Since sorting is the dominant expense we focus on improved SpMM performance by employing a faster sorting algorithm or exploiting our knowledge about the range of input values. Figure 8 illustrates the potential speedup in the sorting performance yielded by two such improvements. By default the Thrust sorting algorithms allocate and free large amounts of temporary memory each time they are invoked, which represents a non-trivial cost. Using the preallocated memory interface[*] we improve the sorting performance of our previous SpMM implementation by minimizing the number of allocations. As a comparison, Figure 8 also captures the performance of the back40computing (B40C) implementation [17] from which the Thrust sorting implementation was derived. The B40C radix sort allows specializations in the number and location of the sorting bits. We exploit this feature to achieve optimal sorting by noting that the total number of bits in the row and column indices of $\hat{C}$ are $\lceil \log_2(m) \rceil$ and $\lceil \log_2(n) \rceil$.

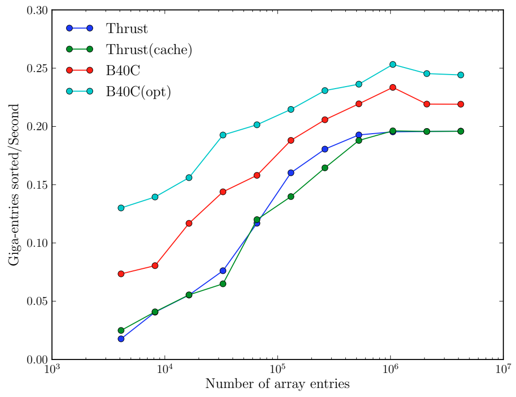


Fig. 8: Sorting performance comparison of Thrust and B40C routines.

Following the marginal improvements realized using optimized global sorting methods we instead focus on sorting within the GPU's higher-bandwidth shared memory for increased efficiency. We observe that $\hat{C}$ consists of a collection of rows of various lengths which may be processed in parallel since there are no dependencies between matrix rows. There are two advantages of operating on a per row basis: 1) two global sorting operations over millions of entries are replaced by numerous operations over possibly tens to thousands of entries and 2) sorting the intermediate entries using shared memory improves overall per-

[*] Introduced in version 1.6 of Thrust

formance. However, row-wise sorting using shared memory places tight bounds on the number of intermediate entries which may be processed per thread group precluding the use of such a method for rows which exceed the maximum shared memory space. Consequently, we identify the rows that violate this space constraint during the analysis phase and process them using the global memory ESC algorithm available within the CUSP library.

The localized shared memory sorting routine is implemented using the highly efficient CTA oriented radix sorting implementation exposed by the CUB branch of B40C library[17]; we implement thread and warp variants. To address the possible inefficiency of attempting to sort a highly varying workload using a static number of threads we scale the number of threads per row of $\hat{C}$ proportionally with the maximum number of entries produced during the expansion. Therefore if $\texttt{nnz}(\hat{C}_{\texttt{row}_i}) \leq \alpha_{\text{thread}}$ we sort each row within a single thread using a sorting network in an "embarrassingly parallel" manner. This optimization dramatically reduces the overall costs of the sorting phase by completely decoupling the threads and preferring the execution of the sorting phase in registers over shared memory. Similarly if $\texttt{nnz}(\hat{C}_{\texttt{row}_i}) \in (\alpha_{\text{thread}}, \alpha_{\text{warp}}]$ then each row is assigned to 1 (32-thread) warp and ordered using radix sort. The remaining rows in the range of $(\alpha_{\text{warp}}, \alpha_{\text{cta}}]$ are processed using an entire CTA. By scaling the number of registers and threads on a per row basis our approach reduces the number of the wasted memory operations caused by rows whose size does not perfectly match any of our targeted sorting boundaries and allows the cost of the sorting pass to scale proportionally with the size of the row. The values of $[\alpha_{\text{thread}}, \alpha_{\text{warp}}, \alpha_{\text{cta}}]$ are parameters that may be set — e.g., we use $[32, 736, 3840]$ in our tests.

A naïve implementation of the sorting implementation described above requires that a key (column index) and value (floating point number) pair be exchanged using shared memory during each pass of the radix sorting routine . To avoid this cost we implement an optimization which sacrifices $\lceil \log_2(m) \rceil$ bits, where m is the intermediate row size, and stores the position corresponding to each entry within the upper region of each column index. To illustrate this point if the maximum column index in the matrix is 2^{20} (1,048,576) then the intermediate row may consist of no more than 2^{12} (4,096) entries in order for the position index to be placed within the sacrificed bit field. Although this optimization is applicable over a more constrained set of rows than those which fit in shared memory, as shown in Algorithm 4, the reduction of key-value to keys-only sorting reduces the total number of shared memory operations and provides a notable improvement in the localized sorting performance.

3.4 Compression

The next computationally intensive phase compresses duplicate entries in $\hat{C}$ using pairwise addition. In contrast to the predictable nature of the total work required to construct $\hat{C}$ in the expansion phase, the number of duplicates, and therefore the number of FLOPs to form any row of C is not easily known *a priori* and often varies significantly between rows in $\hat{C}$. As noted previously in

Algorithm 4: SpMM: `Sorting`

```
parameters:
                    $\beta$:  number of bits per column index
                    $n$:  number of columns in $B$
    $\hat{C} = (\hat{I}, \hat{J}, \hat{V})$ :  column indices, $\hat{J}$, reside in shared memory
return: $\hat{C}, P$                    {$\hat{J}$ sorted row-wise and permutation vector $P$}

foreach row $i$ in $C$ do
      $J$, $V \leftarrow$ extract$_i$ $\hat{J}$, $\hat{V}$          {extract entries where $\hat{I} \equiv i$}
1     $m \leftarrow$nnz$(J)$          {$m$ is the number of expanded entries}
      if $\log_2 m \leq \beta - \lceil \log_2 n \rceil$          {if row not long}
          for $(j, k) \in (J, [0, m])$
2             $j \leftarrow [\underbrace{\quad k \quad}_{\text{upper } \log_2 m \text{ bits}}, \underbrace{\quad j \quad}_{\text{lower } \lceil \log_2 n \rceil \text{ bits}}]$          {store index in upper bits}
3         $J \leftarrow$ lower bit sort$(J)$          {keys-only sort}
4         $P \leftarrow$ upper bit indices          {extract upper bits to form permutation}
      else
5         $J, P \leftarrow$ key_value_sort$(J, [0, m])$          {keys-value sort}
```

Section 3.2 the structure of row $\hat{C}_{\texttt{row}_i}$ and is dependent on the set of rows from B referenced by the column indices in $A_{\texttt{row}_i}$.

The irregularity of the work required to reduce duplicate entries per row in $\hat{C}$ causes a severe imbalance in the compression phase if rows are compressed using a fixed number of threads. The `reduce_by_key` function in Thrust avoids imbalance [3] by reducing duplicates in adjacent locations at the granularity of a fixed number of entries per CTA irrespective of the duplicates per region. While `reduce_by_key` is general and avoids excessive imbalance, it relies on constructing keys and values in global memory.

Storing the keys and values in global memory for relatively long rows allows multiple processing units to work cooperatively to reduce values but ignores possible optimizations associated with utilizing local shared memory storage. Following the sorting phase outlined in Section 3.3 for short rows, the column indices are stored in nondecreasing order in shared memory and the permutation which achieves the sorted ordering is stored within the bit field of each index. As outlined in Algorithm 5 a simple scan operation over the column indices per row may be used to identify unique entries which define the nonzero structure of C. The final task requires the reduction of values corresponding to each duplicate column index and relies on the permutation computed during the sorting phase. The scaled values, which are computed and stored in temporary global memory during the expansion phase, are streamed into registers according to the permutation indices. Then, duplicate values are reduced using a segmented scan operation and tail values are stored to the final row of C. The most ineffi-

Algorithm 5: SpMM: `Compression`

```
   parameters: $\hat{C} = (\hat{I}, \hat{J}, \hat{V})$, $P$        {$P$ permutation which sorts $\hat{C}$ row-wise}
   return: $C$

   foreach row $i$ in $C$ do
1      $v \leftarrow 0$                                          {initialize output value}
       $J$, $V \leftarrow$ extract$_i$ $\hat{J}$, $\hat{V}$                      {extract entries where $\hat{I} \equiv i$}
       for $j = 0, \ldots, \mathtt{nnz}(\hat{C}_{row_i})$
2          $v \leftarrow v + V[P_j]$                             {reduce consecutive values}
           if $J[j] \neq J[j+1]$                 {$J[j+1]$ marks beginning of new nonzero entry}
3              $C_{i,J[j]} \leftarrow v$                     {Store accumlated value to output row}
4              $v \leftarrow 0$                              {re-initialize output value}
```

cient portion of this value compression algorithm is the loading of values from global memory in permuted ordering, however this penalty is mitigated by the implicit spacial locality of the referenced values for short rows.

4 Evaluation

In this section, we examine the performance of a GPU implementation of the proposed ESC method. All of the operations are performed using double precision with error-correcting code (ECC) memory support disabled and the times reported are the average for 10 iterations. We refer to our proposed approach as "Optimized" [†] and compare against the reference ESC variant within the CUSP library as well as the CUSPARSE SpMM implementation[20]. Our system is configured with CUDA v5.0 [22] and Thrust v1.7 [14], and all tests are performed using an Nvidia Tesla C2075[21].

4.1 SPMM

4.1.1 Intermediate Factors

We characterize the SpMM multiplication pairs by the expansion and contraction factors associated with the intermediate matrices. The expansion factor, $\mathtt{nnz}(\hat{C})/\mathtt{nnz}(A)$, describes the ratio of the number of memory references from A to the number data movement operations from B. A relatively large expansion factor indicates that the number of load operations per memory reference is high. The compression factor, $\mathtt{nnz}(C)/\mathtt{nnz}(\hat{C})$, describes the ratio of the number unique entries in C to the number of duplicates in the expanded format. A relatively large compression factors indicate a compression phase with relatively few FLOPs.

[†] Code is available in the CUSP library: `https://github.com/cusplibrary/spmm`

Table 9 illustrates the variation in the expansion and contraction factors possible for computing the inner and outer products of a matrix with dimensions $1024^2 \times 1024$ and a density of 10^{-3}. For the inner product the expansion phase consists a large collection of sparse rows resulting in a contraction phase with a large number of duplicates. In contrast the outer product expansion phase consists of a small collection of rows with many entries which do not contain duplicates and the contraction phase therefore has relatively little work.

	Expand: $\texttt{nnz}(\hat{C})/\texttt{nnz}(A)$				Contract: $\texttt{nnz}(C)/\texttt{nnz}(\hat{C})$			
	Min	Max	Mean	Std	Min	Max	Mean	Std
Inner	1.00	1.14	1.05	**0.02**	7.50	108.00	20.19	**9.49**
Outer	73.0	140.0	105.99	**10.80**	1.00	1.01	1.00	**0.00**

Tab. 9: Expansion and contraction factors for a $1024^2 \times 1024$ matrix.

4.1.2 Performance

Table 10 outlines the performance for each phase of the ESC algorithm for a few representative matrices. The companion operator, $B = P$, used in all operations is generated using a smoothed aggregation interpolation matrix P found in algebraic multigrid methods[3]. It is clear that the cost of the analysis phase varies with the properties of the input matrices but remains a relatively small overhead compared to the overall cost of SpMM. Although for some matrices, such as the anisotropic horseshoe and square matrices, the analysis consumes more than 20% of the total execution time the total improvement in the performance compared to the CUSP version is evident from Table 11. The SpMM portion of the processing time completely encompasses the time required to process the rows of C in a batch oriented manner based on the entries of the $\hat{C}$. As a consequence of processing the rows of C in ascending order according to the number of entries in $\hat{C}$ there is an additional overhead in the form of reordering the final matrix. Though reordering increases the total time per operation it is negligible compared to both the analysis and SpMM times. We note that in the special case where all $\hat{C}$ row lengths are less than 32, processing of rows uses the natural ordering which avoids the overhead of reordering C.

Table 11 presents the speedup of the optimized SpMM over the dataset outlined in Table 3. The average speedup of the our proposed method over the global processing approach utilized in the CUSP version of the ESC algorithm is 3.1 for AP and 6.5 compared to CUSPARSE. The properties of the P operator allow the product AP to be favorable for the ordered approach for several reasons. As illustrated in Figure 5 many of the intermediate row lengths are relatively small and may be processed completely within a single thread, warp, or CTA, thus avoiding the cost of resorting to global memory operations.

In addition, the small row lengths coupled with the fact that $P \in \mathbb{R}^{n\times k}$, where k is typically a constant factor smaller then n, allows the intermediate

Matrix	Analysis		Expand Sort Compress		Reorder	
1a. 2D FD, 5-point	4.6	**14**	28.7	**86**	0.0	**0**
1b. 2D FE, 9-point	7.5	**16**	39.8	**84**	0.0	**0**
2a. 3D FD, 7-point	5.8	**10**	52.0	**90**	0.0	**0**
2b. 3D FE, 27-point	17.9	**11**	146.7	**87**	3.4	**2**
3a. 2D FE, $h \approx 0.03$	4.5	**18**	20.2	**82**	0.0	**0**
3b. 2D FE, $h \approx 0.02$	8.0	**7**	110.6	**92**	1.7	**1**
3c. 2D FE, $h \approx 0.015$	14.5	**7**	198.0	**91**	4.1	**2**
4. 3D FE, $h \approx 0.15$	13.1	**8**	142.7	**89**	4.2	**3**
5a. 2D FE, horseshoe	4.9	**17**	24.3	**83**	0.0	**0**
5b. 2D FE, square	5.3	**17**	26.1	**83**	0.0	**0**

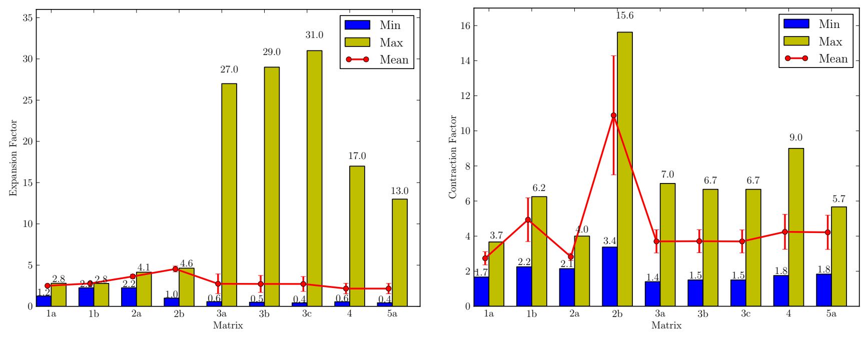
Tab. 10: Time (*ms*) and **percentage** in each phase of the optimized algorithm.

Matrix	CUSPARSE	Ref	Opt	
1a. 2D FD, 5-point	135.5	98.2	33.3	**4.1** / **3.0**
1b. 2D FE, 9-point	206.2	186.6	47.3	**4.4** / **4.0**
2a. 3D FD, 7-point	428.9	196.6	57.8	**7.4** / **3.4**
2b. 3D FE, 27-point	1633.0	820.1	168.0	**9.7** / **4.9**
3a. 2D FE, $h \approx 0.03$	168.2	75.7	24.7	**6.8** / **3.1**
3b. 2D FE, $h \approx 0.02$	368.8	166.6	120.3	**3.1** / **1.4**
3c. 2D FE, $h \approx 0.015$	682.4	323.9	216.6	**3.2** / **1.5**
4. 3D FE, $h \approx 0.15$	1269.2	567.2	160.0	**7.9** / **3.5**
5a. 2D FE, horseshoe	162.4	94.5	29.2	**5.6** / **3.2**
5b. 2D FE, square	412.9	95.0	31.4	**13.2** / **3.0**

Tab. 11: $C = AP$ times *(ms)* and **speedups** (h is an average diameter).

sorting routines to utilize the faster keys-only version as discussed in Section 3.3. In Figure 12 the intermediate expansion and contraction factors for each of the matrices in Table 3 are presented as well as the corresponding standard deviation. It is clear that the maximum and minimum intermediate factors may vary substantially between rows of $\hat{C}$ and therefore to achieve high efficiency the SpMM method must adapt at runtime to accomodate these features.

The matrices outlined in Table 3 exhibit negligible variations in the number of entries per row in the intermediate matrix. As shown in Figure 12 the standard deviation of the expansion phase is moderate for many of the matrices and the mean expansion factor is less than 5 in all cases. These two factors impact the sorting phase because together they imply that many of the intermediate rows are roughly of equal length with the total number of entries in each row a small constant factor larger than the corresponding row from A. As such these matrices may not fully capture the the imbalances which may be present



(a) Expansion (b) Contraction

Fig. 12: SpMM expansion and contraction factors for test matrices.

in more general sparsity patterns giving rising to intermediate matrices with highly varying expansion, sorting, and contraction components.

To address this we conduct a similar set of tests using a small subset of the matrices outlined in the GPU SpMV dataset [4]. The P operator was generated in a similar manner outlined in the previous dataset — i.e., through an AMG interpolation matrix. This class of SpMM operations are susceptible to extreme variations in the all phases which places an increased concern on the applicability of our proposed optimizations to improve the performance.

In Table 13 we note the first instance of our optimized SpMM operation failing to improve over the CUSP version. The *Webbase* matrix originates from a scale free graph and therefore generates a intermediate matrix with a rich diversity of row lengths. However, since many of the rows are small (due to a power law) the total number of intermediate entries in $\hat{C}$ in total is not expected to be large. This allows for the CUSP ESC method to process the entire operation in a single pass and removing any sensitivity to the jagged nature of the workload.

		Total Time		
Matrix	CUSPARSE	Ref	Opt	
Cantilever	61.9	57.6	21.6	**2.8 / 2.7**
Spheres	131.3	90.3	19.3	**6.8 / 4.7**
Accelerator	108.9	39.7	15.4	**7.1 / 3.6**
Economics	67.8	50.6	26.0	**2.6 / 2.0**
Epidemiology	72.3	57.0	17.4	**4.2 / 3.3**
Protein	92.0	56.2	39.4	**2.3 / 1.4**
Wind Tunnel	182.5	107.1	28.1	**6.5 / 3.8**
QCD	97.4	83.6	17.1	**5.7 / 4.9**
Webbase	3086.3	154.2	190.8	**16.2 / 0.8**

Tab. 13: *AP* times *(ms)* and **speedups**.

Finally in Table 14 we present data for $C = A^2$ (cf. Table 13) to illustrate the effectiveness of our method outside of the context of computing $C = AP$. Notably our method consistently outperforms CUSP on this dataset by utilizing shared memory more efficiently and achieving up to almost four times performance improvement. Compared to CUSPARSE our new method is comparable in many cases and substantially outperforms CUSPARSE for matrices such as the *Accelerator*. Though we cannot state definitively the reason for this considerable improvement we speculate it is connected to the analysis phase of our optimized approach. During analysis it is discovered that 75% of the $\hat{C}$ rows generated by A^2 are less than 1024 elements. Adapting to this knowledge our method is capable of processing this set of short rows in approximately 14 milliseconds. Conversely we find that for the *Protein* matrix our approach still outperforms the CUSP version but is twice as slow as CUSPARSE. During the analysis phase for this matrix we find that only half of the input rows are capable of being processed in shared memory. The remaining rows must be processed using the global memory variant which mimics the performance of CUSP, yielding only modest performance.

	Total Time			
Matrix	CUSPARSE	Ref	Opt	
Cantilever	233.1	486.3	204.2	**1.1 / 2.4**
Spheres	305.1	838.1	308.6	**1.0 / 2.7**
Accelerator	238.7	108.1	25.0	**9.5 / 4.3**
Economics	136.9	63.7	30.1	**4.6 / 2.1**
Epidemiology	55.2	65.4	21.3	**2.6 / 3.1**
Protein	280.5	912.4	612.6	**0.5 / 1.5**
Wind Tunnel	374.9	1090.5	477.0	**0.8 / 2.3**
QCD	333.7	527.9	185.7	**1.8 / 2.8**
Webbase	1545.5	583.4	563.2	**2.7 / 1.0**

Tab. 14: A^2 *(ms)* and **speedups**.

5 Conclusion

In conclusion we have presented a new formulation of our global sort based SpMM operation that exhibits notable speedup by exploiting the row-wise processing of the intermediate matrix. In order to study and process the intermediate matrix more effectively we presented a reordering scheme to identify the number of total entries per row of the intermediate matrix and adaptively tune the sorting implementation to reduce the costs of global sorting in favor to localized schemes. While our method does not provide speedup in all cases, we have shown that by performing a lightweight analysis phase it is possible to mitigate the overhead of global memory in favor of shared memory operations.

References

[1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser Press, 1997, pp. 163–202.

[2] R. E. Bank and C. C. Douglas, *Sparse matrix multiplication package (SMMP)*, Advances in Computational Mathematics, 1 (1993), pp. 127–137.

[3] N. Bell, S. Dalton, and L. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM Journal on Scientific Computing, 34 (2012), pp. C123–C152.

[4] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA, 2009, ACM, pp. 18:1–18:11.

[5] A. Buluç and J. R. Gilbert, *On the representation and multiplication of hypersparse matrices*, 2008.

[6] A. Buluç and J. R. Gilbert, *The combinatorial blas: design, implementation, and applications*, IJHPCA, 25 (2011), pp. 496–509.

[7] ——, *Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments*, CoRR, abs/1109.3739 (2011).

[8] J. W. Choi, A. Singh, and R. W. Vuduc, *Model-driven autotuning of sparse matrix-vector multiply on gpus*, in Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, New York, NY, USA, 2010, ACM, pp. 115–126.

[9] E. Cohen, *On optimizing multiplications of sparse matrices*, in IPCO, 1996, pp. 219–233.

[10] M. Garland and D. B. Kirk, *Understanding throughput-oriented architectures*, Commun. ACM, 53 (2010), pp. 58–66.

[11] J. R. Gilbert, S. Reinhardt, and V. B. Shah, *A unified framework for numerical and combinatorial computing*, Computing in Science and Engg., 10 (2008), pp. 20–25.

[12] F. G. Gustavson, *Two fast algorithms for sparse matrices: Multiplication and permuted transposition*, ACM Trans. Math. Softw., 4 (1978), pp. 250–269.

[13] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams,

An Overview of Trilinos, Tech. Rep. SAND2003-2927, Sandia National Laboratories, 2003.

[14] J. Hoberock and N. Bell, *Thrust: A parallel template library*, 2011. Version 1.4.0.

[15] J. Kurzak, S. Tomov, and J. Dongarra, *Autotuning gemms for fermi* , 2011.

[16] D. Merrill, M. Garland, and A. Grimshaw, *Scalable gpu graph traversal*, in 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, New York, NY, USA, 2012, ACM, pp. 117–128.

[17] D. Merrill and A. Grimshaw, *High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing*, Parallel Processing Letters, 21 (2011), pp. 245–272.

[18] D. G. Merrill and A. S. Grimshaw, *Revisiting sorting for gpgpu stream architectures*, Tech. Rep. CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010.

[19] H. Nguyen, *Gpu gems 3*, Addison-Wesley Professional, first ed., 2007.

[20] Nvidia, *CUSPARSE : Users guide.* `http://developer.nvidia.com/cusparse`.

[21] NVIDIA Corporation, *TESLA C2050/C2070 GPU Computing Processor Supercomputing at 1/10th the Cost*, July 2010. www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf.

[22] ——, *NVIDIA CUDA Programming Guide*, Dec. 2012. Version 5.0.

[23] M. O. Rabin and V. V. Vazirani, *Maximum matchings in general graphs through randomization*, Journal of Algorithms, 10 (1989), pp. 557 – 567.

[24] C. Vasconcelos and B. Rosenhahn, *Bipartite graph matching computation on gpu*, in Energy Minimization Methods in Computer Vision and Pattern Recognition, D. Cremers, Y. Boykov, A. Blake, and F. Schmidt, eds., vol. 5681 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 42–55.

[25] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*, Parallel Comput., 35 (2009), pp. 178–194.

[26] S. Williams, A. Waterman, and D. Patterson, *Roofline: an insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.