Optimizing Sparse Matrix-Matrix Multiplication for the GPU

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

Abstract

Sparse matrix-matrix multiplication (SpGEMM) is a key operation in numerous areas from information to the physical sciences. Implementing SpGEMM 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 SpGEMM operation into three, highly-parallel phases: expansion, sorting, and contraction, and introduce a set of complementary bandwidth-saving performance optimizations. Our implementation is fully general and our optimization strategy adaptively processes the SpGEMM workload row-wise to substantially improve the performance by decreasing the work complexity and utilizing the memory hierarchy more effectively.

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 (SpGEMM) is a fundamental operation that arises in many practical contexts, including graph contractions [12], multi-source breadth-first search [6], matching [24], 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.

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

[†] 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`

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 [26, 27]. A common strategy for improving SpMV performance is to exploit *a priori* knowledge of the sparsity structure of the matrix in order to minimize expensive off-chip memory operations. 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 SpGEMM, 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, SpGEMM adds another level of complexity through its depenency on the detailed interaction of two sparse matrices. Indeed, simply computing the number of floating point operations required by the SpGEMM, or even the size of the output matrix, is not substantially simpler than computing the SpGEMM itself.

The recent demand for high performance SpGEMM 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, SpGEMM 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 solve phase is comprised of SpMV and level 1 BLAS operations and therefore readily accelerated by employing existing highly-optimized GPU implementations [3].

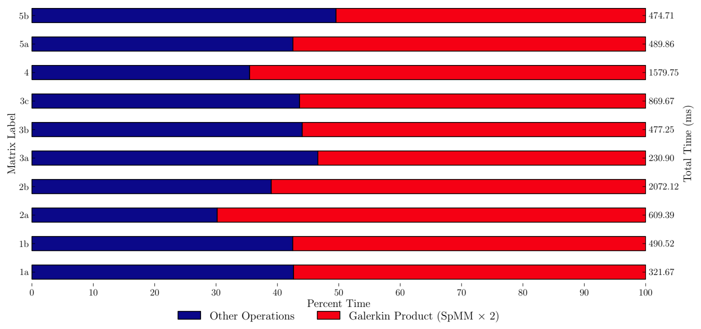


Fig. 1: Relative cost of the SpGEMM during the AMG setup phase for a series of matrices (see Table 3).

The approach to SpGEMM in [3] is based on a decomposition of SpGEMM into 3 phases: expansion, sorting, and contraction (ESC). The ESC formulation of the SpGEMM operation is naturally implemented with data parallel primitives, such as those provided by the Thrust parallel algorithms library [15].

In this work we implement a set of optimizations that enhance SpGEMM performance by exploiting on-chip memory, whenever possible, and reducing the cost associated with the sorting phase. The contribution of this work is the study and development of a SpGEMM 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 by studying the interaction of the sparsity patterns of the input matrices and avoid arbitrarily poor decisions based upon either input matrix individually. The algorithm takes an adaptive approach to using the shared memory in the architecture, thereby increasing utilization of the memory locality in the method.

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 [11, 19, 17, 25].

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 [27], 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 SpGEMM 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 SpGEMM operation.

Given two sparse matrices $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$, for $k, m, n \in \mathbb{N}$, SpGEMM 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 SpGEMM 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 [16, 8], approaching the (theoretical) peak limits of memory bandwidth and arithmetic throughput, respectively.

In contrast to GEMM, SpGEMM 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 SpGEMM 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, SpGEMM operations are generally outside the innermost solver loop.

Sequential SpGEMM algorithms [2, 13], 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. These methods construct each output row by iterating over the rows of A and for each column entry performing a scale and reduction operation on the values in the corresponding row from B. To accomplish this sequential algorithms rely on a large amount, $\mathcal{O}(N)$, of temporary storage to efficiently store and reduce unique entries on any row of C. Therefore, sequential methods focus on constructing the output matrix and accessing both input operands on a per row basis. Parallel SpGEMM 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 [14, 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, a straight-forward parallel approach to SpGEMM on the GPU would decompose the matrices on a per thread or CTA basis which may be advantageous but requires complex decompositions to avoid unnecessarily high imbalances in the work distribution. The focus of the work here is on developing fine-grained parallelism to avoid these bottlenecks on the GPU.

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 SpGEMM 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 [20]. As a result, the ESC algorithm with parallel primitives is robust, reliable and efficient. The high-level structure of the ESC algorithm is summarized in Algorithm 1. To facilitate the memory constrained nature of the GPU large problems are decomposed into smaller more manageable slices. The ESC algorithm achieves this decomposition by partitioning A into M submatrices, where each row of A is in one and only one submatrix. The operation `slice`(A) in Algorithm 1 generates subsets of contiguous rows by considering the row-wise memory requirements and selecting a set of rows to process in parallel based on the amount of available global memory. Next, `expand`(A_k, B) generates the products associated with slice A_k and B in coordinate (COO) format, described in further detail in Section 3.2. Then the `sort`$(\hat{C}_k)$ operation orders these expanded products by row and column indices. The sorted products are subsequently processed by the `contract`(C_k) operation to compute the sum of duplicate entries and store the reduced set of nonzero entries produced by A_k. Finally, the `construct`(C) operation allocates memory sufficiently large to store the number of entries in the output matrix C and concatenates the slices, C_k, together.

Algorithm 1: SpGEMM: `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     $\hat{C}_k \leftarrow$ contract($\hat{C}_k$)         {contract 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

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

Then the expansion, sorting, and contraction phases yield

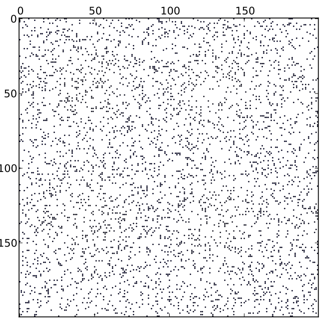
$$\hat{C} = \begin{bmatrix} (0,0,10) \\ (1,3,60) \\ (1,1,40) \\ (1,1,150) \\ (1,0,120) \\ (1,3,280) \\ (1,1,240) \\ (2,3,350) \\ (2,1,300) \\ (3,3,180) \\ (3,1,120) \end{bmatrix} \xrightarrow[\text{sort}]{} \begin{bmatrix} (0,0,10) \\ (1,0,120) \\ (1,1,40) \\ (1,1,150) \\ (1,1,240) \\ (1,3,60) \\ (1,3,280)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

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 $\texttt{nnz}(A) = \texttt{nnz}(B) = 3812$, and $\texttt{nnz}(C) = 33678$, while $\hat{C}$ contains 75786 entries.

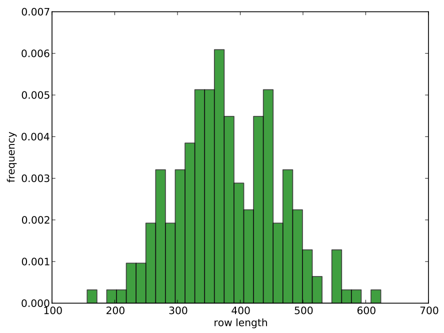
In the following, for a sparse matrix A we denote by $A_{\texttt{row}_i}$ the i^{th} row of A (and similar for columns), while $\texttt{NNZ}(A_{\texttt{row}_i})$ denotes the set of nonzero column indices. The ESC process for (1) follows from the inner product view of multiplication:

$$C_{i,j} = A_{\texttt{row}_i} \cdot B_{\texttt{col}_j} = \sum_k A_{i,k} B_{k,j}. \tag{5}$$

From this we see that simultaneous access to $A_{\texttt{row}_i}$ and $B_{\texttt{col}_j}$ is necessary to construct entry $C_{i,j}$. Yet, there are two issues to address when considering the inner product formulation of SpGEMM: 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 SpGEMM 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



(a) Random sparse matrix pattern.



(b) Distribution of row lengths.

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

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. We focus on CSR based approaches in this work but note that our algorithms are equally applicable to CSC based data structures since the transpose of any matrix stored in CSR is equivalent to the original matrix stored in CSC format. Therefore if the underlying data-structure for both matrices is CSC, then multiplying the matrices in reverse order yields the transpose of our CSR base approach.

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. 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 contraction.

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 from the University of Florida Sparse Matrix Collection and previously found in GPU SpMV data sets [3, 4, 10]. 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].

				Row-wise stats		
Matrix		n	`nnz`	min	max	mean
1a. 2D FD		1 048 576	5 238 784	3	5	4.99
1b. 2D FE		1 048 576	9 424 900	4	9	8.99
2a. 3D FD		1 030 301	7 150 901	4	7	6.94
2b. 3D FE		1 030 301	27 270 901	8	27	26.47
3a. 2D FE		550 387	3 847 375	4	11	6.99
3b. 2D FE		1 182 309	8 268 165	4	11	6.99
3c. 2D FE		2 185 401	15 287 137	4	11	6.99
4. 3D FE		1 088 958	17 095 986	5	38	15.69
5a. 2D FE		853 761	5 969 153	3	7	6.99
5b. 2D FE		832 081	5 817 905	4	10	6.99
Cantilever	cant	62 451	4 007 383	1	78	64.17
Spheres	consph	83 334	6 010 480	1	81	72.13
Accelerator	cop20k_A	121 192	2 624 331	0	81	21.65
Economics	mac_econ_fwd500	206 500	1 273 389	1	44	6.17
Epidemiology	mc2depi	525 825	2 100 225	2	4	3.99
Protein	pdb1HYS	36 417	4 344 765	18	204	119.31
Wind Tunnel	pwtk	217 918	11 634 424	2	180	53.39
QCD	qcd_5_4	49 152	1 916 928	39	39	39
Webbase	webbase-1M	1 000 005	3 105 536	1	4700	3.11

Tab. 3: Test problems of square matrices ($n \times n$) with `nnz` nonzeros. The average number of entries for the companion B matrices are 2.93 and 3.79 for the AMG and SpMV matrices respectively.

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.

3.1 Setup

A straight-forward approach to processing SpGEMM operations in parallel would 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. However, 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.

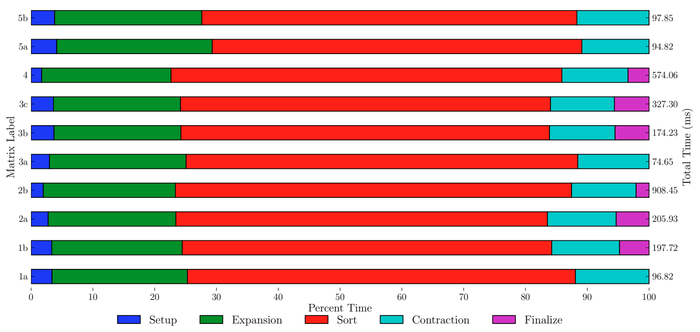


Fig. 4: Component-wise performance of the reference ESC SpGEMM operation.

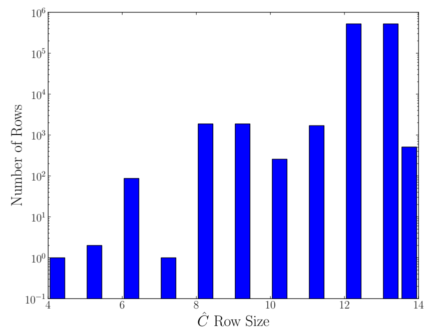
To illustrate, the minimum number of FLOPs associated with forming $C_{\texttt{row}_i}$ is proportional to

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

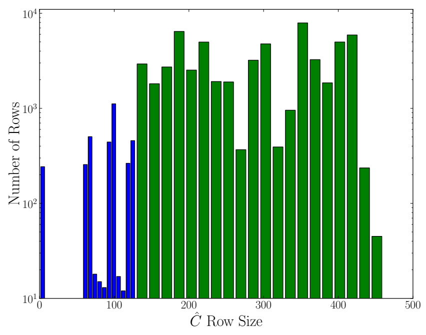
This quantity represents the total number of products required to scale each row of B referenced by each column entry within the row.

In Figure 5 the variation in the intermediate matrix, with respect to the number of products, is depicted for several test instances described in Section 4. The histograms are formed by grouping each row of $\hat{C}$ according to total number of products. We use a coloring scheme to draw attention to regions of interest in which processing rows at various granularities of parallelism on the GPU may be advantageous. For example the regions in blue represent rows of length less than 128 which could be processed by a single warp, while regions in green range from 128 to 1024 benefit from a CTA oriented processing scheme. Finally, rows larger than 1024 are highlighted in red to indicate processing with multiple CTAs necessitating the use of global memory to communicate intermediate results. Figure 5d highlights the particularly challenging nature of some SpGEMM instances, note that the row lengths of $\hat{C}$ vary so dramatically in size that we plot the logarithm of the row sizes and therefore this SpGEMM instance generates substantially diverse worloads per row.

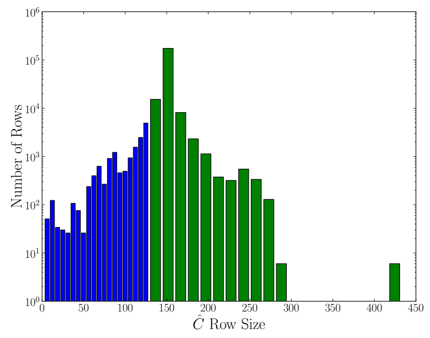
Based on Figure 5 we conclude that any static assignment of computational units to process a fixed number of rows of the input matrix, A, could lead to arbitrarily poor load balance and possible degradation in performance. One strategy for avoiding this load imbalance is to implement the entire algorithm 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 software. Therefore we use a different approach based on the work distribution model that groups rows into several categories that are processed using the most appropriate method.



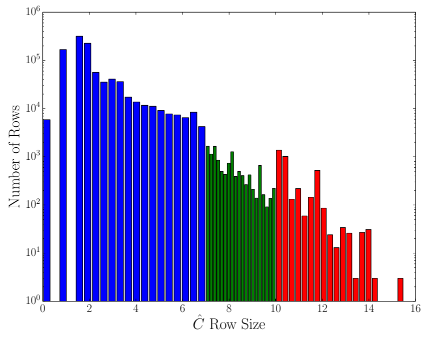
(a) 1a. 2D FD, 5-point



(b) Cantilever



(c) Wind Tunnel



(d) Webbase

Fig. 5: Distribution of $\hat{C}$ row lengths for SpGEMM operations in Tables 11,13. Rows are grouped by color: 0–127 (Blue), 128–1024 (Green), and ≥ 1024 (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: SpGEMM: `reorder`

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

$F_i = 0$ for $i = 1$ to $m$
foreach row $i$ in $C$ do
    for $j \in NNZ(A_{row_i})$
        $F_i \leftarrow F_i + \mathtt{nnz}(B_{\mathtt{row}_j})$        {gather $B$ row lengths based on $A$ column indices}
$P \leftarrow \mathtt{sort}(F)$                       {set $P$ to permutation of $F$ 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 a 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 is assigned to load rows from B, and the average row length is significantly smaller than the number of assigned threads, then many threads are either idle or loading 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 SpGEMM 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,(i, j, v), in the matrix, a directed edge is added between the row i and column j vertices in the bipartite graph with weight v. 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 inner matrix dimensions agree. As an example, we illustrate the layered model diagram in Figure 7 using matrices A and B defined as

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

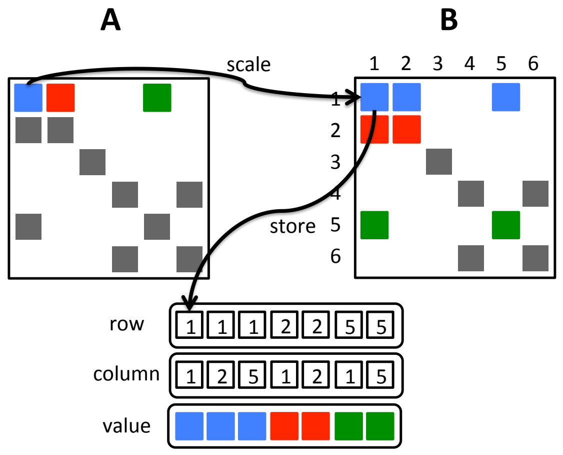
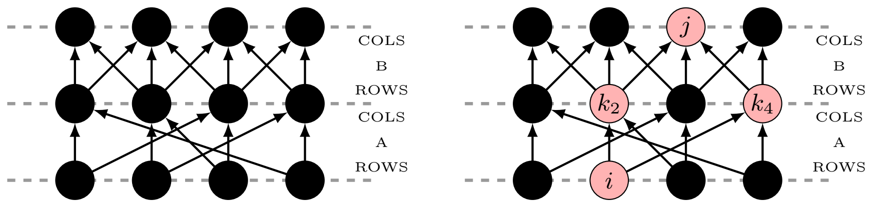


Fig. 6: Scaling rows of B by column indices corresponding to nonzero entries of A and storing in intermediate buffer.



(a) Layered graph model of AB. (b) Paths contributing to an entry.

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

In the layered graph model any 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. A weight is attributed to all paths according to a binary operation — e.g., multiplication in the case of SpGEMM — 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 between the column set of B and 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 [17]. 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 which corresponds to rows of B which must be expanded.

However, in contrast to previous BFS implementations [17], 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: SpGEMM: `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[17]. 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 (cf. (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 SpGEMM performance by either 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 SpGEMM implementation by minimizing the number of allocations. As a comparison, Figure 8 also captures the performance of the back40computing (B40C) implementation [18] 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 $\lceil \log_2(n) \rceil$.

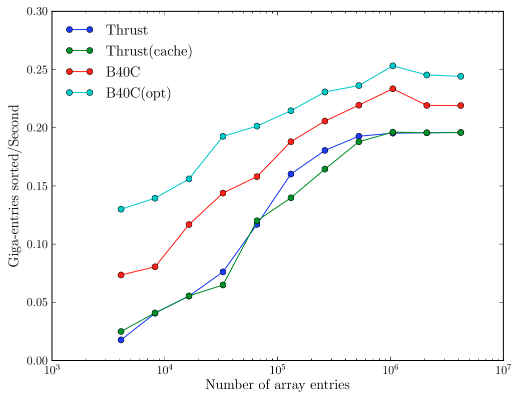


Fig. 8: Sorting performance comparison of Thrust and B40C routines.

Figure 8 highlights the limited gains achieved by focusing on improving global sorting methods, therefore we concentrate 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 reduces global memory operations. 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

* Introduced in version 1.6 of Thrust

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 programming library, a collection of CTA primitives[18]; 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 $[32, 736, 6144]$ in our tests. We note that by sorting rows independently in registers or shared memory that additional optimization techniques, such as packing permutation into column indices, are applicable though highly dependent on the specific SpGEMM instance.

Algorithm 4: SpGEMM: `Sorting`

```
parameters:                 $n$:   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
      $J$, $V \leftarrow \texttt{extract}_i\ \hat{J}$, $\hat{V}$              {extract entries where $\hat{I} \equiv i$}
1     $m \leftarrow \texttt{nnz}(J)$                        {$m$ is the number of expanded entries}
2     $J, P \leftarrow \texttt{key\_value\_sort}(J, [0, m])$            {keys-value sort}
```

The $\texttt{extract}_i$ function in Algorithm 4 refers to the subset of entries on row i of $\hat{C}$. Note that because short rows are generated in shared memory using cooperative warp or CTA methods during the expansion phase then references to the subset of entries on any particular row of $\hat{C}$ simply refers to the shared memory region with no additional processing required. In the case of global memory processing any specific row of $\hat{C}$ may be extracted by computing the prefix-sum of the number of expanded products per row. This information is readily available as a byproduct of the analysis phase and therefore $\texttt{extract}_i$ represents a CSR style referencing of row i by the offset of the first entry in that row.

3.4 Contraction

The next computationally intensive phase contracts the values associated with duplicate column indices 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 Section 3.2 the structure of row $\hat{C}_{\texttt{row}_i}$ 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 $\hat{C}$ may cause imbalance in the contraction phase if rows are contracted 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 number of duplicates per row. While `reduce_by_key` is general and avoids excessive imbalance, it relies on constructing keys and values in global memory.

Algorithm 5: SpGEMM: `Contraction`

```
  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}
      $J, V \leftarrow \texttt{extract}_i\ \hat{J}, \hat{V}$        {extract entries where $\hat{I} \equiv i$}
      for $j = 0, \ldots, \texttt{nnz}(\hat{C}_{row_i})$           {Segmented scan by keys, $J$, over values in $V$}

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 accumulated value to output row}
4             $v \leftarrow 0$                                     {re-initialize output value}
```

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 shared memory storage. Following the sorting phase outlined in Section 3.3 the column indices, in nondecreasing order, and the permutation which achieves the sorted ordering are stored in shared memory. Algorithm 5 describes the construction of the reduced set of entries on any row of C by performing both reduction and store operations. This contraction phase may therefore be decomposed into two phases, the reduction of the scaled values corresponding to the same column index and the storage of entries into C. First, the scaled values, which are computed and stored in temporary global memory during the expansion phase, are streamed into registers in sorted order using the precomputed permutation indices. Then, the summation of values associated with identical column indices are computed using a specialized segmented scan operation which immediately stores the column index and value corresponding to the last duplicate entry in each segment. The most inefficient portion of this value contraction 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.

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. Though ECC is an important feature to mitigate random errors in DRAM memory it also decreases the peak achievable bandwidth on the device. We refer to our proposed approach as "Optimized"[†] and compare against the reference ESC variant within the Cusp library as well as the Cusparse SpGEMM implementation[21]. Our system is configured with CUDA v5.0 [23] and Thrust v1.7 [15], and all tests are performed using an Nvidia Tesla C2075[22].

4.1 SpGEMM

4.1.1 Intermediate Factors

We characterize the SpGEMM 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 contraction factor, $\mathtt{nnz}(\hat{C})/\mathtt{nnz}(C)$, describes the ratio of the number of unique entries in C to the number of duplicates in the expanded format. A relatively large contraction factors indicate a contraction phase with relatively few FLOPs.

Table 9 illustrates the variation in the expansion and contraction factors possible for computing the inner, A^TA, and outer, AA^T, products of a random, sparse matrix with dimensions $1024^2 \times 1024$ and a density of 10^{-3}. For the inner product the expansion phase consists of 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. The large variation in the intermediate factors of this example illustrates the need for an adaptive method.

[†] Test matrices and code are avaiable at `http://lukeo.cs.illinois.edu/spgemmdata/index.html`

	Expand: $\mathtt{nnz}(\hat{C})/\mathtt{nnz}(A)$				Contract: $\mathtt{nnz}(\hat{C})/\mathtt{nnz}(C)$			
	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, used in all operations is generated using a smoothed aggregation interpolation matrix found in algebraic multigrid methods [3]. The interpolation operators typically have the form of a scattering operation. In particular, in this example interpolation uses a distance-2 maximal independent set (MIS-2) to generate disjoint vertex subsets. This pattern would result in one entry per row of P with the column indices indicating the MIS-2 set each vertex is assigned. In smoothed aggregation these subsets are extended to overlap to improve numerical properties of the solver at the expense of increasing the work during multiplication. 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 SpGEMM.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 SpGEMM 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 SpGEMM 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 SpGEMM over the dataset outlined in Table 3. The average speedup of our proposed method over the global processing approach utilized in the Cusp version of the ESC algorithm is 3.1 for AB.The properties of the B operator, in this case, allow the product AB 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 $B \in \mathbb{R}^{n\times k}$, where k is typically a constant factor smaller then n, allows the intermediate sorting routines to exploit the faster keys-only sorting variant by implanting permutation indices in the upper bitfield of the column indices.

Matrix	Analysis		Expand Sort Contract		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**

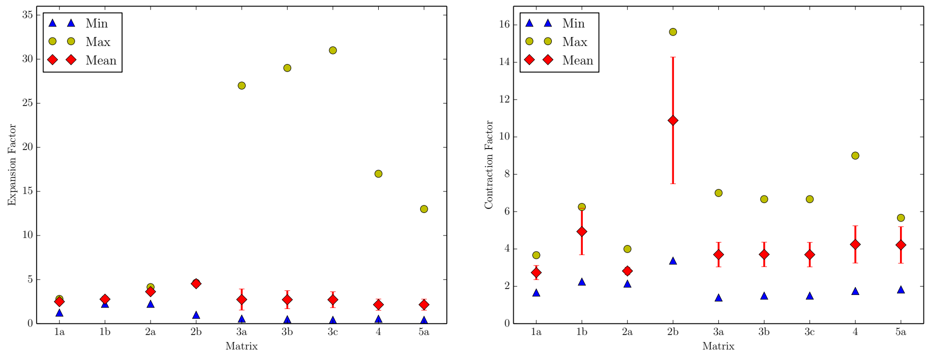
Tab. 10: Time (ms) and **percentage** in each phase of the optimized algorithm.

	Total Time				
Matrix	Ref	Opt		Cusparse	
1a. 2D FD, 5-point	98.2	33.3	**3.0**	135.5	**0.78**
1b. 2D FE, 9-point	186.6	47.3	**4.0**	206.2	**0.90**
2a. 3D FD, 7-point	196.6	57.8	**3.4**	428.9	**0.46**
2b. 3D FE, 27-point	820.1	168.0	**4.9**	1633.0	**0.50**
3a. 2D FE, $h \approx 0.03$	75.7	24.7	**3.1**	168.2	**0.45**
3b. 2D FE, $h \approx 0.02$	166.6	120.3	**1.4**	368.8	**0.45**
3c. 2D FE, $h \approx 0.015$	323.9	216.6	**1.5**	682.4	**0.47**
4. 3D FE, $h \approx 0.15$	567.2	160.0	**3.5**	1269.2	**0.45**
5a. 2D FE, horseshoe	94.5	29.2	**3.2**	162.4	**0.58**
5b. 2D FE, square	95.0	31.4	**3.0**	412.9	**0.23**

Tab. 11: $C = AB$ run time *(ms)* and **speedups** (h is an average diameter).

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 SpGEMM method must adapt at runtime to accommodate 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 5.0 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



(a) Expansion (b) Contraction

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

these matrices may not fully capture the imbalances which may be present in more general sparsity patterns giving rise 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 B operator was generated in a similar manner outlined in the previous dataset — i.e.,.,through an AMG interpolation matrix. This class of SpGEMM 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 that our optimized ESC achieves performance at least on par with the reference Cusp version. A particularly challenging test case involves the Webbase matrix which 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 $\hat{C}$ 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.

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 $C = AB$. Notably our method consistently outperforms Cusp on this dataset by utilizing shared memory more efficiently and achieving up to almost seven 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 80% of the $\hat{C}$ rows generated by A^2 are less than 1024 elements for the Accelerator matrix. Adapting to this knowledge the optimized ESC algorithm is capable processes this set of short rows in approximately 60 milliseconds. Conversely we find that for the Protein matrix our approach still outperforms the Cusp version but is considerably

	Total Time				
Matrix	Ref	Opt		Cusparse	
Cantilever	115.9	33.9	**3.4**	145.5	**0.80**
Spheres	170.2	41.8	**4.1**	247.3	**0.69**
Accelerator	67.9	30.1	**2.2**	216.2	**0.31**
Economics	45.0	37.5	**1.2**	67.5	**0.67**
Epidemiology	53.3	16.6	**3.2**	71.7	**0.74**
Protein	113.7	36.0	**3.2**	181.7	**0.63**
Wind Tunnel	205.7	50.2	**4.1**	446.2	**0.46**
QCD	83.2	28.5	**2.9**	96.7	**0.86**
Webbase	152.9	149.1	**1.0**	3091.3	**0.05**

Tab. 13: AB run time *(ms)* and **speedups**.

slower than 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	Ref	Opt		Cusparse	
Cantilever	1979.6	390.5	**5.1**	469.1	**4.2**
Spheres	3410.6	706.5	**4.8**	1127.3	**3.0**
Accelerator	629.5	113.4	**5.6**	1052.9	**0.6**
Economics	63.2	40.9	**1.5**	136.5	**0.5**
Epidemiology	65.3	19.8	**3.3**	54.9	**1.2**
Protein	4111.9	3442.5	**1.2**	1227.7	**3.4**
Wind Tunnel	4641.0	628.5	**7.4**	1580.9	**2.9**
QCD	528.4	78.4	**6.7**	333.3	**1.6**
Webbase	571.9	308.5	**1.9**	1558.2	**0.4**

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

5 Conclusion

In conclusion we have presented a new formulation of our global sort based SpGEMM 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. We note that naïve strategies to selectively process the SpGEMM computations based on isolated analysis of the input matrices does not adequately capture the complexity of the SpGEMM and a combined approach considering contributions from both yields more optimization opportunities. Though our experiments consisted of SpGEMM instance originating from AMG we stress that this feature is immaterial with respect to our analysis and processing scheme because of our explicit focus on the intermediate characteristics of the histogram model.

Though effective at utilizing faster shared on-chip registers and shared memory our method does not achieve substantial improvement over the global memory approach when the total number of entries per row in the intermediate format exceeds the maximum shared memory size. Specifically, with respect to the AMG two SpGEMM orderings are possible $(P^TA)P$ and $P^T(AP)$. We perform our computations using the later AP formulation as it naturally exposes more parallelism, in terms of rows, then the former. In future work we plan to selectively process large rows in global memory more effectively by modeling the performance trade-offs of more sophisticated schemes versus any additional analysis or processing overhead.

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. Buluc and J. Gilbert, *On the representation and multiplication of hypersparse matrices*, in Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, 2008, pp. 1–11.

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

[7] A. Buluç and J. R. Gilbert, *Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments*, SIAM Journal of Scientific Computing (SISC), 34 (2012), pp. 170 – 191.

[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] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25.

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

[12] 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.

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

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

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

[25] 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.

[26] 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.

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