GPU-ACCELERATED SPARSE MATRIX-MATRIX MULTIPLICATION BY ITERATIVE ROW MERGING*

FELIX GREMSE†, ANDREAS HÖFTER‡, LARS OLE SCHWEN§, FABIAN KIESSLING†, AND UWE NAUMANN‡

Abstract. We present an algorithm for general sparse matrix-matrix multiplication (SpGEMM) on many-core architectures, such as GPUs. SpGEMM is implemented by iterative row merging, similar to merge sort, except that elements with duplicate column indices are aggregated on the fly. The main kernel merges small numbers of sparse rows at once using sub-warps of threads to realize an early compression effect which reduces the overhead of global memory accesses. The performance is compared with a parallel CPU implementation as well as with three GPU-based implementations. Measurements performed for computing the matrix square for 21 sparse matrices show that the proposed method consistently outperforms the other methods. Analysis showed that the performance is achieved by utilizing the compression effect and the GPU caching architecture. An improved performance was also found for computing Galerkin products which are required by algebraic multigrid solvers. The performance was particularly good for 7-point stencil matrices arising in the context of diffuse optical imaging and the improved performance allows to perform image reconstruction at higher resolution using the same computational resources.

Key words. Sparse Matrix-Matrix Multiplication, GPU Programming, Algebraic Multigrid, Fluorescence-mediated Tomography

AMS subject classifications. 65F50, 65Y20, 65M06

1. Introduction.

This paper presents a GPU-accelerated method for general sparse matrix-matrix multiplication (SpGEMM). We denote it by RMerge because it merges rows using sub-warps of threads. The method was developed in the context of fluorescence-mediated tomography, where we required a fast SpGEMM implementation for an algebraic multigrid (AMG) solver [21, 30].

1.1. Sparse matrix-matrix multiplication.

Sparse matrix-matrix multiplication (SpGEMM) is an essential component for many problems arising in combinatorial and scientific computing. Graphs can be represented as sparse matrices and vice versa [34]. Many graph processing operations such as graph clustering [42], subgraph extraction [10], breadth first search [18], or transitive closure [37] can be solved using sparse matrix-matrix multiplication. SpGEMM is an essential part of multigrid solvers [8] and, in particular, of algebraic multigrid solvers [5, 8], which are the fastest known sparse linear solvers for many applications. Furthermore, SpGEMM has been used for quantum modelling [38], to evaluate the chained product of sparse Jacobians [24], and to compute joins with duplicate elimination for relational databases [1]. While special algorithms may be advantageous for special types of graphs or matrices, a fast and general SpGEMM implementation turns out to be desirable for abstract programming and high-level languages for linear algebra and graph processing [18]. Consequently, parallel implementations for SpGEMM are provided by several libraries, such as the Intel MKL (CPU-based) or Nvidia's Cusparse [36] and Cusp [6] for GPU

*This article was published in SIAM Journal of Scientific Computing, Vol. 37, No. 1, pp. C54–C71, 2015, DOI 10.1137/130948811, http://www.siam.org/journals/sisc/37-1/94881.html.

This work was supported RWTH Aachen University (I3TM-Seed-Fund) and the German Federal Ministry of Education and Research (BMBF, Virtual Liver Network, grant number 0315769).

†Experimental Molecular Imaging, RWTH Aachen University

‡LuFG Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen University

§Fraunhofer MEVIS, Bremen

processing. For an evaluation of several alternative SpGEMM approaches on the GPU, we refer to [12, 31].

The matrix product $\mathbf{C} = \mathbf{AB}$ is defined as $c_{ij} = \sum_{v=0}^{k-1} a_{iv} b_{vj}$ for $\mathbf{A} \in \mathbb{R}^{m \times k}$, $\mathbf{B} \in \mathbb{R}^{k \times n}$, and $i = 0, \dots m-1, j = 0, \dots n-1$. The naive implementation requires $O(n^3)$ floating point operations (flops), if $n = m = k$, which can be reduced to approximately $O(n^{2.807})$ using Strassen's classical algorithm [39], to $O(n^{2.375})$ using the Coppersmith–Winograd algorithm [11], and further to $O(n^{2.373})$ using the asymptotically best currently known method [43]. For sparse matrices, many of these multiplications can be omitted whenever a_{iv} or b_{vj} is zero. Thus, the number of multiplications (and additions) is much smaller than the one required for dense matrices. An efficient sequential algorithm for SpGEMM is described in [3, 25]. Here, the term efficient means that the processing time is proportional to the number of required multiplications. This method computes each output row at a time and requires a large temporary array of size n to keep track of the resulting elements of the output row. The array is accessed in an unstructured way, which turns out to be inefficient on modern memory architectures due to numerous cache misses [41]. The processing time becomes more strongly determined by memory accesses instead of arithmetic operations [10, 41]. Furthermore, the output rows are computed in an unordered way. While the commonly used CSR (compressed sparse rows) format does not enforce sorted rows, many applications require this property. To achieve this, the rows need to be sorted requiring loglinear computational cost. Ignoring this sorting step, the complexity for sparse matrix squaring is bounded by $O(n \operatorname{nnz}(\mathbf{A}))$ [45] where $\operatorname{nnz}(\mathbf{A})$ denotes the number of nonzero entries in the matrix $\mathbf{A}$. This bound may not be relevant for many applications, because often the matrices are so sparse that each output row can be computed within a cost of much less than $O(n)$, e.g., $O(1)$. For sufficiently dense matrices, $O(n \operatorname{nnz}(\mathbf{A}))$ becomes $O(n^3)$ making dense algorithms more efficient [45].

1.2. GPU Programming. Graphics processing units (GPUs) provide a high computational power and are increasingly used in computing centers due to their high performance-to-price and performance-to-energy ratios. While CPUs are traditionally designed for general purpose computations, processing single or few threads at high speed, GPUs are designed to process thousands of relatively slow threads in parallel to achieve a higher total throughput [17]. To reduce the overhead due to instruction fetching and scheduling, these threads are grouped into so-called warps, e.g., containing 32 threads, which operate in a synchronized fashion [35]. Therefore, optimal performance is typically achieved if all threads of a warp follow the same code path, i.e., perform the same operation on possibly different data. Diverging code paths are allowed but cause some threads to be idle while the code paths of the other threads are processed. Fortunately, this is handled by the compiler and the hardware.

Many warps are active at the same time and are managed by a scheduler which assigns them to different units of the GPU-processor to perform arithmetic, logic, special, or memory operations. The idea is to achieve a high throughput by keeping many parts of the silicon busy. This serves the purpose to hide long memory latencies which may stall warps for hundreds of cycles. For this to be effective, thousands of threads need to be running concurrently. This requires special parallel programming techniques for many problems [5] because the number of concurrent threads is limited by resources required per thread, such as registers and shared memory. While the memory bandwidth between GPU memory and GPU-processors is very high (e.g., 288 GB/s) [35], the bandwidth between CPU and GPU memory is much lower and can often become be a limiting factor [20]. Furthermore, the GPU memory is limited, e.g., to 12 GB [35] which should be taken into account when designing GPU-accelerated systems.

Fortunately, high level compilers for languages such as CUDA (which is mostly compatible with C++) are available and allow abstract (e.g., template-based) programming; incorporation of assembly code is rarely needed. Nevertheless, a good understanding of the underlying hardware is essential. Even though features such as recursion, dynamic memory allocation, and function pointers are provided by the latest generation of GPUs [35], simple code avoiding these features often performs better.

GPU-based implementations of SpGEMM are available from Cusparse [36] and Cusp [6]. The approach of Cusparse is described in [14] and available as library with the Nvidia Cuda toolkit [35]. Cusp is described in [5] and the code is available online [6]. It expands all values arising from scalar multiplications, sorts them, and finally compresses the entries with duplicate column indices. Therefore, this has been named an expansion, sorting, and compression (ESC) method. The computational load is therefore shifted to finding an efficient parallel sorting algorithm for the GPU, which is performed by Radix-sort as implemented in [27]. Another approach is described in [32], it uses blocking to avoid the large intermediate array required by the sequential algorithm [3, 25]. The approach by Cusp has been improved in [12], by introducing a set of bandwidth saving operations, however, the method is not available as code or library. The performance of parallel algorithms is difficult to predict analytically, because sparse matrices exhibit very different structures, which may impact the performance due to irregular memory accesses. The sparsity structures of a heterogeneous subset of 8 matrices are shown in Fig. 1.1 and the same subset is used throughout the text for illustration purposes. Furthermore, the distribution of row lengths of the left and right hand sides may vary strongly (Fig. 1.2), which causes challenges for load balancing. Therefore, the performance is often measured experimentally for a broad set of matrices [12, 31, 32].

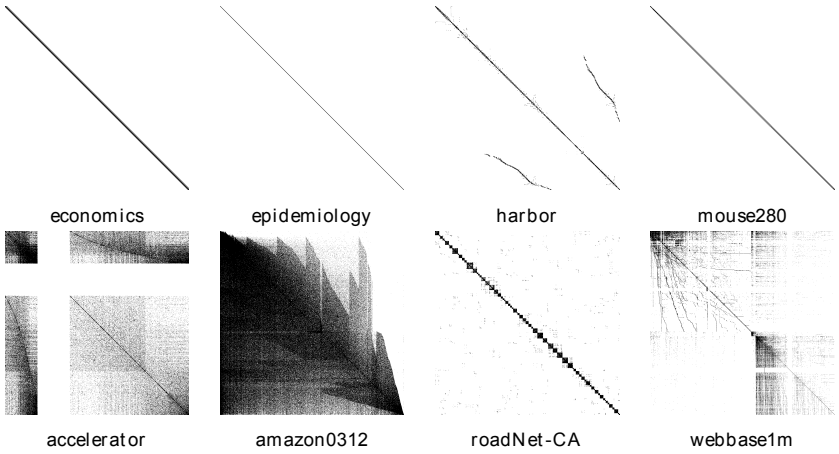


FIG. 1.1. *Common Sparse Matrix Structures. The nonzero structure is shown for a heterogeneous selection of sparse matrices: economics is a macroeconomic model, epidemiology is a 2D Markov model of an epidemic, harbor is used in 3D CFD simulations of the Charleston Harbor, mouse280 was generated to model diffuse light propagation inside a mouse for fluorescence-mediated tomography, accelerator holds the cavity design of a particle accelerator, amazon0312 represents a directed graph based on the 'Customers Who Bought This Item Also Bought' feature of the Amazon website, roadNet-CA represents an undirected graph of the road network of California, and webbase1m is a web connectivity matrix.*

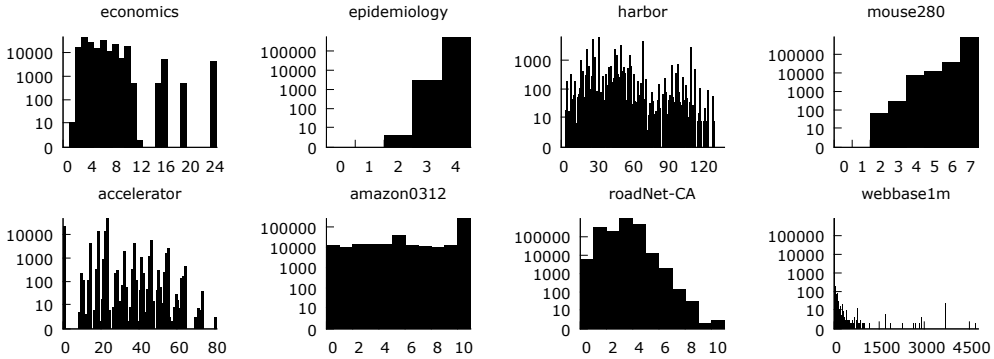


FIG. 1.2. *Row length histograms. The histograms of the row lengths are often heterogeneous. Heterogeneous row length distributions of left and right hand sides cause challenges for load balancing for parallel SpGEMM implementations. Furthermore, the histograms vary strongly between matrices and little assumptions can be made to optimize parallel SpGEMM implementations.*

1.3. Algebraic Multigrid. Multigrid methods belong to the fastest known sparse linear solvers for many applications [8, 33]. They iteratively solve a sparse linear system $\mathbf{Ax} = \mathbf{b}$ by reducing the residuum on different scales. High-frequency components of the residuum are reduced considering the original system. Low-frequency components are removed by corrections obtained using coarsened approximations of the system. A pyramid of coarse approximations can be obtained by different approaches. They range from geometrically coarsening the underlying problem (geometric multigrid) [8] to schemes only considering the sparsity structure and entries of the system matrix (algebraic multigrid) [8, 16, 40]. Geometric coarsening is specific to the geometry and the physics of the problem being considered. This domain-specific knowledge can often be used to parallelize the coarsening scheme and port it to GPUs [2, 4, 7, 19].

Algebraic coarsening is typically used as a 'black box' method [15] and the coarsening requires computation of Galerkin products which are a kind of sparse matrix-matrix multiplications. Creating the pyramid of coarsened problems, which dominates the preparation time, has also undergone parallelization [22, 23, 26], but is difficult to implement efficiently for GPUs [5, 44]. Many GPU implementations of AMG perform the preparation on the CPU [29] or make use of special auxiliary grids [44]. A fully GPU-based AMG implementation has been presented in [5] and uses the Cusp library [6] for SpGEMM which constitutes the main bottleneck in the prepare phase.

Once an AMG has been prepared, the cost of each iteration is linear in $\mathrm{nnz}(\mathbf{A})$ and typically a constant number of iterations is sufficient for achieving a fixed solver tolerance [8]. Therefore, the system can be solved in $O(\mathrm{nnz}(\mathbf{A}))$ time. The general applicability and the efficiency makes AMG solvers very attractive as solvers [8] or as preconditioners, e.g., for the conjugate gradient method [5].

1.4. Notation. In the following, the terms vector and matrix always refer to sparse vectors and matrices, unless explicitly noted otherwise. Vectors (e.g., $\mathbf{a}, \mathbf{x}$) and matrices (e.g., $\mathbf{A}, \mathbf{B}, \mathbf{C}$) are denoted using lower and upper case roman letters, respectively. $\mathbf{a}_{i:}$ denotes row i of $\mathbf{A}$. As introduced above, $\mathrm{nnz}(\mathbf{A})$ denotes the number of nonzeros of $\mathbf{A}$. We define $\mathrm{flops}(\mathbf{A},\mathbf{B})$ as the number of nontrivial arithmetic operations required to multiply $\mathbf{A}$ and $\mathbf{B}$ using a naive dense implementation. When

splitting a warp into sub-warps, the sub-warp size is denoted by W.

1.5. Overview. In the following sections, we describe how SpGEMM can be implemented using row merging. A GPU-accelerated function `MulLimited()` is introduced which computes the product **AB** for left hand sides with limited elements per row by merging rows using sub-warps of threads. Then we show how general SpGEMM is reduced to this function. The performance of this method is evaluated and analyzed for sparse matrix squaring the sparse matrices listed in Table 1.1. Additionally, we assess the performance of computing the Galerkin product which is required for algebraic multigrid solvers. Finally, the results are discussed.

TABLE 1.1

*Sparse matrices used for performance experiments, available from the University of Florida Sparse Matrix Collection [13]. Several properties, i.e., width, number of nonzeros, as well as the maximum and average number nonzeros per row are listed. Another column lists the arithmetic workload of matrix squaring, i.e. flops(***A**,**A***). The last column lists the compression factor achieved during matrix squaring, which is defined as the ratio of the number of nonzero multiplications to* nnz(**AA**). *The matrices are sorted in somewhat regular (upper 10) and more irregular (lower 11) matrices.*

Name	Width (=Height)	*nnz*	Max Row *nnz*	Mean Row *nnz*	Work-load [Mflop]	Com-pression
cantilever	62 451	4 007 383	78	64.2	539.0	15.5
economics	206 500	1 273 389	44	6.2	15.1	1.1
epidemiology	525 825	2 100 225	4	4.0	16.8	1.6
harbor	46 835	2 374 001	145	50.7	313.0	19.8
mouse280	901 972	6 227 648	7	6.9	86.3	2.0
protein	36 417	4 344 765	204	119.3	1 110.7	28.3
qcd	49 152	1 916 928	39	39.0	149.5	6.9
ship	140 874	7 813 404	102	55.5	901.3	18.7
spheres	83 334	6 010 480	81	72.1	927.7	17.5
windtunnel	217 918	11 634 424	180	53.4	1 252.1	19.1
accelerator	121 192	2 624 331	81	21.7	159.8	4.3
amazon0312	400 727	3 200 440	10	8.0	56.8	2.0
ca-CondMat	23 133	186 936	280	8.1	8.3	1.8
cit-Patents	3 774 768	16 518 948	770	4.4	164.3	1.2
circuit	170 998	958 936	353	5.6	17.4	1.7
email-Enron	36 692	367 662	1 383	10.0	103.0	1.7
p2p-Gnutella31	62 586	147 892	78	2.4	1.1	1.0
roadNet-CA	1 971 281	5 533 214	12	2.8	35.0	1.4
webbase1m	1 000 005	3 105 536	4 700	3.1	139.0	1.4
web-Google	916 428	5 105 039	456	5.6	121.4	2.0
wiki-Vote	8 297	103 689	893	12.5	9.1	2.5

2. SpGEMM implementation. In this section, the data structures and algorithmic parts for the sparse matrix-matrix multiplication by row merging are explained. The product $\mathbf{C} = \mathbf{AB}$ can be split into many vector-matrix products $\mathbf{c} = \mathbf{aB}$, where $\mathbf{a}$ and $\mathbf{c}$ are corresponding rows of $\mathbf{A}$ and $\mathbf{C}$. This product $\mathbf{aB}$ is defined as a linear combination of some rows of $\mathbf{B}$ which are selected and weighted by $\mathbf{a}$. This can be understood as a merging operation similar to merge sort [28], except that elements with the same index are combined, resulting in a compression effect (Fig. 2.1). We explain how the sparse matrix-matrix product can be computed efficiently for left hand sides with limited elements per row, by performing the row merging using sub-warps of GPU threads. Then we show how the general sparse matrix-matrix product is reduced to this operation.

2.1. Matrix formats. The most commonly used sparse matrix format is the terminated compressed sparse row (CSR) format which is also used in the presented algorithm. This format stores the nonzero values sequentially in an array *val*, as well as the corresponding column indices in another array *col*. A third array *rowStarts* stores the indices of *val* and *col* where a new row starts. Therefore, nnz values and $n + nnz$ indices are stored for a matrix of height n. Due to the available row starts and consecutively saved nonzeros, CSR permits fast indexing of row vectors in $O(1)$. Access to individual elements is slower and should be avoided. We assume that the rows are always sorted by the column indices, because this is required for efficient row merging. 'Terminated' means that an additional entry at the end of *rowStarts*

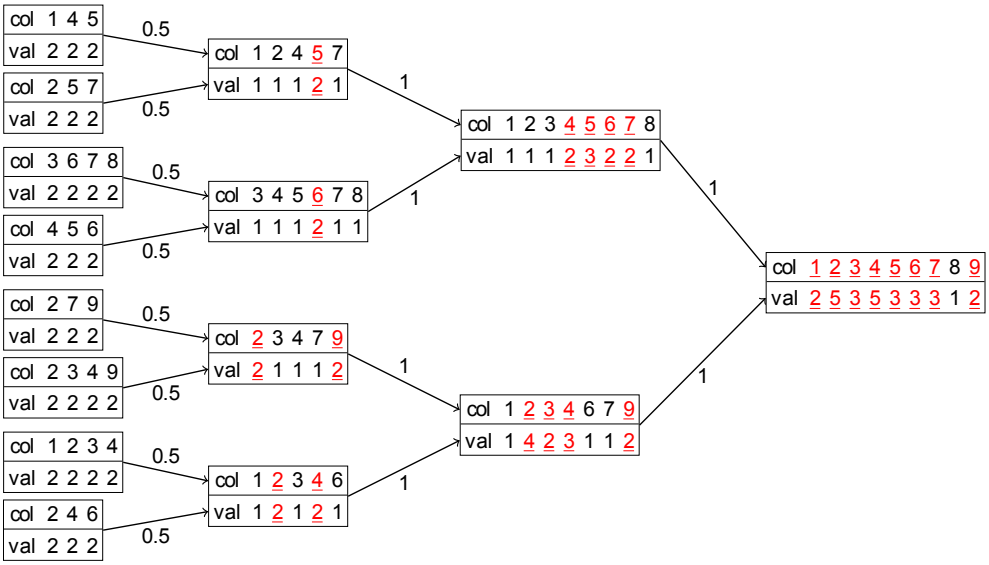


FIG. 2.1. *Row merging. Sparse vector-matrix multiplication* $\mathbf{c} = \mathbf{aB}$ *can be implemented by row merging. The sparse rows of* $\mathbf{B}$*, selected and weighted by* $\mathbf{a}$*, are merged in a hierarchical way similar to merge sort. For simplicity, the values of* $\mathbf{B}$ *and* $\mathbf{a}$ *are assumed to be* 2 *and* 0.5*, respectively. Multiplications are only required at the first merging level, because at later levels the weights are* 1*. The sparse vectors remain sorted throughout the computation. At each level, the number of rows is reduced by half but the rows become longer due to merging. The total size is reduced, depending on the amount of overlap (shown in red and underlined). Values with identical column indices are added up causing an increasing degree of compression at later levels compared to the first level. Here, 27 value-index pairs are compressed into 9 value-index pairs, resulting in a compression factor of* 3.0*.*

holds the total number of nonzeros. This allows fetching the row length of row r as $rowStarts[r+1] - rowStarts[r]$ without having to worry to go out of bounds. Therefore, the maximum row length can be computed using a reduce-transformed operation which is available in the thrust library [27]. The CSC (compressed sparse column) format is equivalent to CSR, except that columns are stored consecutively. The coordinate format (COO) stores two arrays of length nnz, one for the row indices and one for the column indices, in addition to the nnz matrix elements. Matrices can be easily converted from CSR or CSC to COO, since only the row or column indices need to be filled in. The conversion from COO into CSR or CSC is more complicated and requires a sorting operation of the COO entries. The Cusp library [6] provides such functionality.

2.2. Special sparse matrix-matrix product. We implemented a GPU-accelerated function named `MulLimited()` which computes the product $\mathbf{C} = \mathbf{AB}$ for a left hand side $\mathbf{A}$ with a limited number of nonzeros per row. The maximum number of nonzeros per row is limited by the sub-warp size W, i.e., 2, 4, 8, 16, or 32, which is a parameter to `MulLimited()`. `MulLimited()` consists of three steps. First the structure of the result $\mathbf{C}$ is computed using one GPU kernel call which computes the row lengths of $\mathbf{C}$. Then the rowStarts vector of $\mathbf{C}$ is computed using a parallel prefix sum [27] and the memory of $\mathbf{C}$ is allocated. Finally, the values and column indices of $\mathbf{C}$ are computed using another kernel call. The first kernel call, computing the row lengths, is similar to the last kernel call. We therefore only describe the latter in the following. For further details we refer to the source code, available as supplemental material.

The kernel has a template parameter for the sub-warp size W, which can be 2, 4, 8, 16, or 32. Each sub-warp computes one output row $\mathbf{c} = \mathbf{aB}$, where $\mathbf{a}$ and $\mathbf{c}$ are corresponding rows of $\mathbf{A}$ and $\mathbf{C}$, by merging up to W rows of the right hand side $\mathbf{B}$

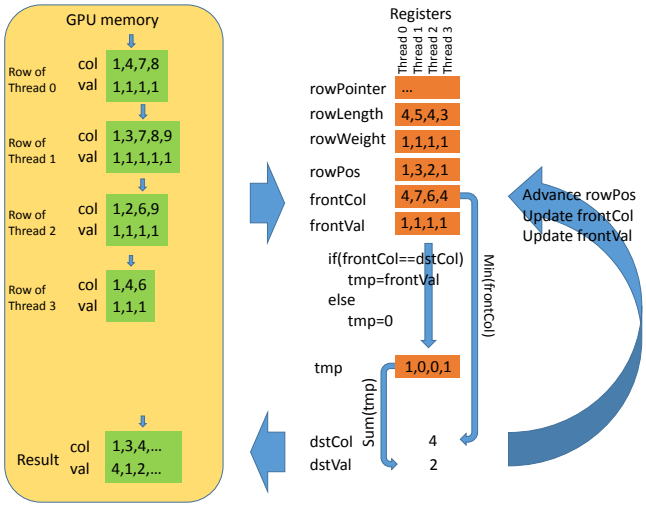


FIG. 2.2. *Row merging by a GPU sub-warp. A sub-warp, here consisting of 4 threads, merges 4 sparse input rows into one output row. Each thread manages one input row, i.e., advances and fetches the front value and index. All values and weights are assumed to be one in this example. At each iteration, one element of the resulting output row is computed using a sub-warp min and sub-warp sum operation and stored into the global GPU memory.*

Listing 1
CUDA code for sub-warp reduction using the shuffle operation. A sub-warp must be of size 2, 4, 8, 16, or 32.

```
template<int SubWarpSize, typename T>
static __device__ __host__ T WarpSum(T value){
  for (int i=SubWarpSize/2; i>=1; i/=2)
    value+=__shfl_xor(value, i);
  return value;
}

template<int SubWarpSize, typename T>
static __device__ __host__ T WarpMin(T value){
  for (int i=SubWarpSize/2; i>=1; i/=2)
    value=Min(value,__shfl_xor(value, i));
  return value;
}
```

which are selected and weighted by $\mathbf{a}$. The kernel requires that each row of $\mathbf{A}$ has at most W elements per row. The algorithm for the sub-warps, illustrated in Fig. 2.2, is relatively simple. Each thread of a sub-warp is associated to one input row and pulls the front value and column index of its row. To compute the next element of $\mathbf{c} = \mathbf{aB}$, the minimal column index is computed and all values belonging to this index are accumulated using a sub-warp reduction (Listing 1). The simple code has the advantage that each thread requires a moderate amount of registers and no shared memory at all. Therefore, an occupancy of 100 % is achieved, meaning that enough warps are running to keep the GPU busy [35]. We use CUDA thread blocks of 128 threads, with $blockDim.x = W$ and $blockDim.y = 128/W$. Since up to 16 thread blocks can run concurrently on the used GPU architecture, this results in 2048 threads per GPU processor.

Let us assume a maximum row length of 4 as for the epidemiology matrix (Table 1.1). Using a sub-warp size of 4 means that 8 rows can be computed by each warp in parallel, i.e., one by each of the 8 sub-warps. Therefore, all threads are busy. If a row was computed by the whole warp, i.e., 32 threads, only 4 threads would be working and 28 threads would be idle. Splitting the warp into sub-warps works well because the sub-warps follow mostly the same code path but work on different data, which is compatible with the SIMD (single instruction, multiple data) architecture of GPUs. For some matrices from Table 1.1 the matrix square can be computed using a single call to `MulLimited()`, e.g., epidemiology, mouse280, and amazon0312 have a maximum row length of 4,7, and 10, respectively, and the matrix square can be computed using sub-warp sizes of 4, 8, and 16, respectively. Since `MulLimited()` allocates memory solely for the output matrix, these matrix-matrix multiplications can be performed without memory overhead.

The algorithm is somewhat wasteful with respect to the number of arithmetic operations performed for the sub-warp reductions, however. For each element of the result $\mathbf{c}$, a sub-warp min and sub-warp sum reduction is required (Listing 1). The number of operations for these sub-warp reductions is $W \cdot \operatorname{ld} W$, i.e., $\{2, 8, 24, 64, 160\}$ where W is the sub-warp size of $\{2, 4, 8, 16, 32\}$, respectively. A serial implementation could maintain a small sorted list or a priority queue to compute the reductions of the contributing values with less operations [9]. For GPUs, however, much more arithmetic operations than global memory operations per time are available. The double precision peak performance for the used GPU is 1500 billion operations per second (Gflops), while the memory throughput is by comparison only 36 billion doubles per second

Listing 2

RMerge algorithm. First, RMerge determines the maximum row length of the left hand side $\mathbf{A}$*. Then the left hand side is iteratively split into two matrices* $\mathbf{tmp}$ *and* $\mathbf{G}$ *so that* $\mathbf{G}$ *has at most* W *elements per row. Then* $\mathbf{G}$ *is premultiplied to* $\mathbf{B}$ *using the function* `MulLimited()`*. At each iteration the maximum row length of the remaining left hand side is updated using an integer division with rounding up (DivUp). For the final multiplication the smallest sufficient sub-warp size is used.*

```
template<typename T>
static MatrixCSR<T> RMerge(MatrixCSR<T> A,MatrixCSR<T> B,int subWarpSize){
  int maxRowLength=MaxRowLength(A);
  while(maxRowLength>subWarpSize){
    MatrixCSR<T> tmp,G;
    SpmmDecompose(tmp,G,A,subWarpSize);
    A=tmp;
    B=MulLimited(G,B,subWarpSize);
    maxRowLength=DivUp(maxRowLength,subWarpSize);
  }
  return MulLimited(A,B,SufficientSubWarpSize(maxRowLength));
}
```

(288GB/s) [35], i.e., 41 times more arithmetic operations are available per time than transfer operations to and from the global memory. Therefore, it makes sense to spend these operations since the arithmetic units would otherwise be idle.

2.3. General sparse matrix-matrix product. To be able to use the `MulLimited()` function for the general sparse matrix-matrix product $\mathbf{C} = \mathbf{AB}$, the matrix $\mathbf{A}$ is split into $\mathbf{A} = \mathbf{A_k G_{k-1} ... G_1}$, such that all matrices have at most W elements per row. The splitting is performed iteratively in the sequence $\mathbf{A} = \mathbf{A_1} = \mathbf{A_2 G_1} = \mathbf{A_3 G_2 G_1} = \mathbf{A_4 G_3 G_2 G_1}$ and so forth, where $\mathbf{A_k}$ are the parts that can be further split, while $\mathbf{G_k}$ are finished once they are computed. Then, the chain of matrix-matrix multiplications $\mathbf{C} = \mathbf{A_k G_{k-1} ... G_1 B}$ can be computed from right to left using `MulLimited()`.

In the following, this is illustrated for a matrix $\mathbf{A}$ with two rows (red and blue/underlined), assuming a sub-warp size $W = 2$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} 2 & 0 & 7 & 4 & 6 & 0 & 2 & 2 \\ \underline{3} & 0 & 0 & \underline{2} & 0 & \underline{5} & 0 & \underline{3} \end{bmatrix} \mathbf{B}$$

Since $\mathbf{A}$ contains more than 2 elements in at least one row, $\mathbf{A}$ is split into $\mathbf{A} = \mathbf{A_2 G_1}$ by splitting each row $\mathbf{a}$ of $\mathbf{A}$ into $\lceil \mathrm{nnz}(\mathbf{a})/W \rceil$ rows.

$$\mathbf{C} = \mathbf{A_2 G_1 B} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & \underline{1} & \underline{1} \end{bmatrix} \begin{bmatrix} 2 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ \underline{3} & 0 & 0 & \underline{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \underline{5} & 0 & \underline{3} \end{bmatrix} \mathbf{B}$$

$\mathbf{G_1}$ can be understood as a merging operation on the rows of $\mathbf{B}$. This creates some intermediate rows (3 for the red row and 2 for the blue row) which need to be further merged as encoded by $\mathbf{A_2}$. $\mathbf{G_1}$ contains the same column indices and values as $\mathbf{A}$, allowing to reuse the corresponding data structures. Furthermore, the values of $\mathbf{G_2}, \ldots, \mathbf{G_k}$ are guaranteed to be one which is used to reduce memory consumption

and to avoid trivial multiplications. Now $\mathbf{A_2}$ is further split into $\mathbf{A_3G_2}$:

$$\mathbf{C} = \mathbf{A_3G_2G_1B} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & \underline{1} \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \underline{1} & \underline{1} \end{bmatrix} \begin{bmatrix} 2 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ \underline{3} & 0 & 0 & \underline{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \underline{5} & 0 & \underline{3} \end{bmatrix} \mathbf{B}$$

Finally, `MulLimited()` can be used to compute $\mathbf{C} = \mathbf{A_3G_2G_1B}$ from right to left. The algorithm (Listing 2) iteratively splits the remaining left hand side $\mathbf{A_i}$ into two parts, $\mathbf{A_{i+1}}$ and $\mathbf{G_i}$, and multiplies $\mathbf{G_i}$ with the current right hand side to compute an intermediate result $\mathbf{C_i}$. Therefore, at most two intermediate matrices are kept in memory at a time. Each matrix splitting reduces the maximum row length of $\mathbf{A_i}$ by a factor W, therefore, the number of calls to `MulLimited()` depends logarithmically on the maximum row length of $\mathbf{A}$.

The size of the intermediate results, i.e., sparse matrices, is largest for the first multiplication $\mathbf{G_1B}$ and shrinks for later levels because more and more overlap occurs. It is possible, however, that no overlap occurs, if the merged rows have non-overlapping column indices. The size of each intermediate matrix is larger or equal to the resulting matrix.

Based on performance experiments discussed later, we chose a sub-warp size of 16 as default for all experiments. For the last multiplication, the lowest sufficient sub-warp size is used, however. Therefore, for 7-point stencil matrices as left hand sides, this automatically results in a single call to `MulLimited()` with sub-warp size 8.

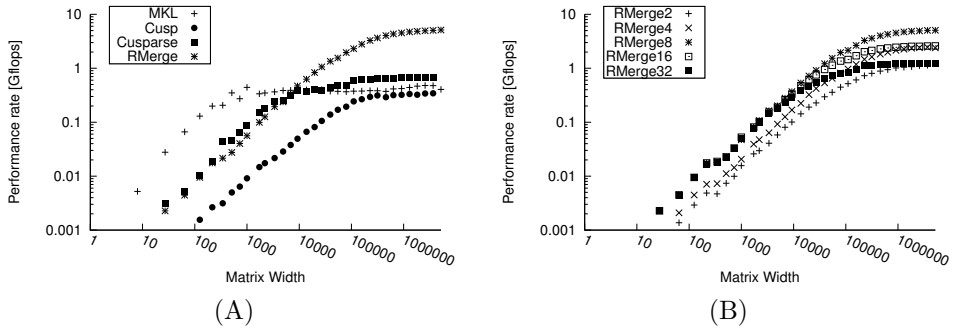
3. Performance Measurements. In the following, we report several performance measurements. For all experiments RMerge was used as a black-box operation, i.e., without tuning for special applications. All matrix-matrix multiplications were performed in double precision. To allow a comparison between matrices, we report the performance rate instead of the processing time, because the former is normalized with respect to the problem size. The performance rate is defined as the ratio of the arithmetic workload and the measured processing time. The arithmetic workload flops($\mathbf{A}$,$\mathbf{B}$) is defined as twice the number of nontrivial scalar multiplications (to account for the additions) which can be computed as $\sum_{j \in \hat{\mathbf{a}}_{i:}} \text{nnz}(\mathbf{b}_{j:})$ for each result row $\mathbf{c}_{i:}$ [12, 31], where $\hat{\mathbf{a}}_{i:}$ denotes the nonzero indices of row $\mathbf{a}_{i:}$. All performance rate measurements were repeated 11 times and the median was used because of its robustness with respect to outliers. The median absolute deviation from the median was used as error bars.

3.1. Data sets. The sparse matrices, listed in Table 1.1, were taken from the University of Florida Sparse Matrix Collection [13]. Inspired by [31], we sorted the matrices into regular (the upper 10) and irregular matrices (the lower 11) and sorted these subsets alphabetically. Regular matrices result from problems involving mesh approximations, e.g., from finite element methods, while irregular matrices mostly result from network structures. These matrices were also used for performance tests by [32] and therefore provide a basis for comparison. The matrix mouse280 originates from a finite difference mesh (using a 7-point stencil) which models the diffuse light propagation inside a mouse [21]. It is a 901 972 × 901 972 band matrix with a band-diameter of 13 224 elements. The fill-in would result in roughly 901 972 × 13 224 values, i.e., around 95 GB of data, being prohibitively large for a naive direct method.

3.2. Devices and software. A PC (Dell PrecisionWorkstation T7500) equipped with two Intel Xeon X5677 (3.47 GHz) quad-core processors, 96GB of DDR3 RAM (CAS latency 9, DRAM Frequency 665.1 MHz), and an Nvidia GeForce GTX Titan (14 Kepler cores, 2688 CUDA cores, 876 MHz, 6 GB memory at 1502 MHz) was used for performance measurements. The used operating system was Windows 7 (64-bit). The CUDA Toolkit 6.0 was used, which includes the Cusparse library. Cusp version 0.4 available from [6] was used. To measure the GPU caching efficiency, we used the Nvidia Nsight 4.0 profiler. The Intel MKL library 11.1 Update 3 was used for CPU-based SpGEMM. The C++ code was compiled with Visual Studio 2012 Ultimate. The results of all functions were checked against a reference implementation to assert correct structure and floating point values. The source code of the RMerge algorithm is provided as supplemental material. For the linear correlation analysis we used GraphPad Prism 5.

3.3. Matrix squaring: performance comparison. Considering the same example application as in [12, 31, 32], we measured the time to compute the square of a sparse matrix $\mathbf{C} = \mathbf{AA}$. To assess the scalability, the performance was measured as a function of the matrix width for 3D Poisson matrices (Fig. 3.1). All four methods show an increasing performance rate with increasing matrix sizes, eventually reaching a plateau. The CPU method reaches its plateau earlier than the GPU methods and outperforms these for small matrices. Asymptotically, RMerge outperforms MKL, Cusp, and Cusparse by the factors 10.5, 14.5, and 7.2, respectively. These ratios are computed from the average performance rate at the highest 5 sizes. Despite being used as a black box method, RMerge works particularly well for these simple matrices. Furthermore, for this case, our method does not require any memory overhead, i.e., no intermediate matrices are allocated, because the maximum row length of 7 allows to compute the result $\mathbf{C}$ directly using a single call to `MulLimited()` with sub-warp size 8, which actually turned out to be the optimal sub-warp size (Fig. 3.1).

Next, we compared the performance for matrix squaring for all matrices listed in Table 1.1. Our method outperformed all other methods for all matrices (Fig. 3.2), achieving an average speedup of 4.1, 6.9, and 9.2, for MKL, Cusp, and Cusparse, respectively. In comparison to the other methods, Cusp achieved a very constant



(A) (B)

FIG. 3.1. *Performance of matrix squaring of 7-point stencil matrices. 3D Poisson matrices of increasing sizes were used. (A) Performance comparison of MKL, Cusp, Cusparse, and RMerge. RMerge outperforms the other methods for large matrices. The CPU method (MKL) works best on small matrices. (B) The effect of the sub-warp size on RMerge is shown. Sub-warp size 8, which is automatically selected by RMerge, achieves the best performance, because the result can be computed with a single call to* `MulLimited()` *and because less threads are idle than when using sub-warp size 16 or 32.*

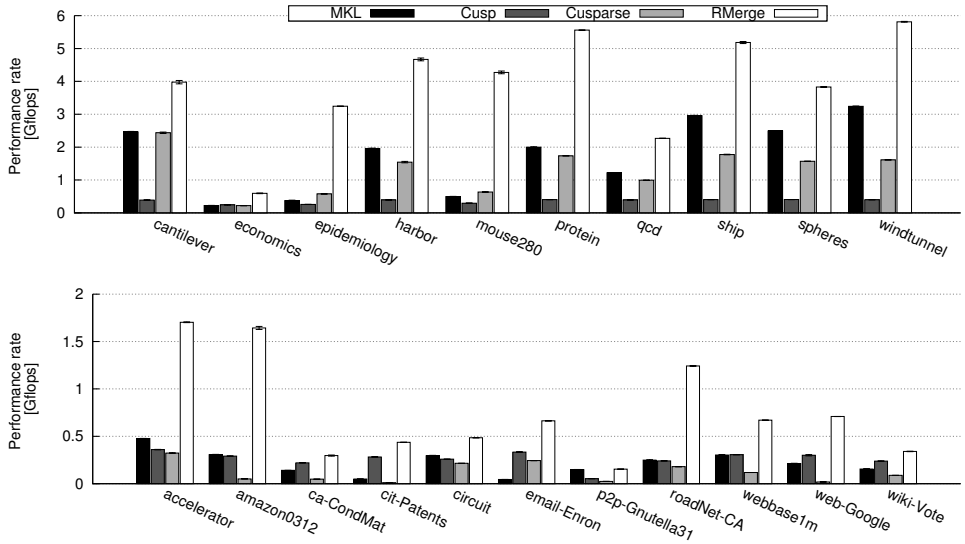


FIG. 3.2. *Performance rate for matrix squaring. The performance is highly heterogeneous. The results are split into results for regular (top) and irregular matrices (bottom) from Table 1.1. Cusparse performs better than Cusp for most of the regular matrices but Cusp is better for many irregular matrices. RMerge consistently outperforms the other methods. Performance was measured using double precision.*

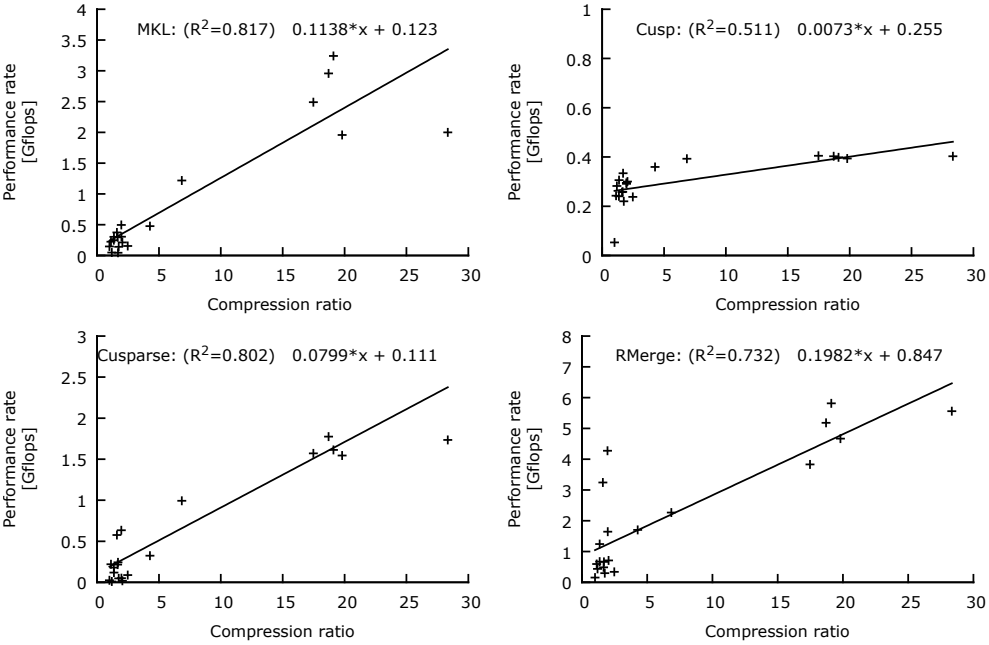


FIG. 3.3. *Matrix squaring performance rate over the compression factor. All methods show a statistical dependency ($P \leq 0.001$) between the compression factor and the performance rate when squaring the matrices from Table 1.1.*

performance, which can be explained by the ESC (Expansion, Sorting, Compression) approach, requiring allocation of an intermediate array with a size equal to the number of multiplications. The array is then sorted in $O(n)$ using radix sort which dominates the computational cost. Therefore, Cusp is hardly affected by irregular matrix structures and outperforms Cusparse or MKL for many of the irregular matrices.

Furthermore, we noticed that the performance rate was higher for matrices with a higher compression factor. A linear (Pearson) correlation analysis showed that, for all methods, the performance rate correlated significantly ($P \leq 0.001$) with the compression factor (Fig. 3.3).

We also compared the performance of our method with a more recent implementation [31]. Using the same GPU, our method was faster for each matrix of the set used in [31], with an average speed-up of 2.6.

3.4. Matrix squaring: performance analysis. We performed several experiments to investigate the performance of our method. Applying our matrix splitting approach with Cusp and Cusparse reduced the performance by average factors of 1.6 and 1.4, respectively, and therefore does not explain the higher performance of RMerge. Measuring the execution time of the first matrix-matrix multiplication $\mathbf{A}_1\mathbf{B}$, we found that 65% of the computation time is spent here (Table 3.1). We also measured the execution time of the MKL, Cusp, and Cusparse for this multiplication and found that RMerge achieves a speed-up of 6.5, 9.5, and 13.3, respectively. Apparently, it is our implementation of the `MulLimited()` function which causes the high performance of RMerge. Therefore, we further investigated this.

Each call to `MulLimited()` consists of three parts. First, the resulting matrix structure is computed, then allocated, and finally filled with values and column indices. The kernels for computing the structure and filling it are very similar, however, the former only writes the number of nonzero elements of each result row to the global memory and does not consider the values. Computation and allocation of the matrix structure requires 39% of `MulLimited()` on average as we measured for the first matrix-matrix multiplication (Table 3.1). After allocation, the column indices and values are filled into the preallocated matrix. Each sub-warp of this kernel merges up to W rows of $\mathbf{B}$ to compute and store one output row. Therefore, three aspects affect the performance: reading the input rows, processing, and storing the output rows.

When reading the input rows, the row merging benefits from the caching architecture in two ways. First, each thread benefits when successively fetching values and indices from the row associated to the thread. Second, threads of different sub-warps may require the same input row and the cache might be able to avoid some global memory accesses. Therefore, we measured the caching ratio of the kernel, i.e., the ratio of the amount of data processed by the threads to the amount of data fetched from global memory (Table 3.1), where we used flops($\mathbf{AB}$) times sizeof(*double*) for the former and a value provided by the Nvidia Nsight profiler for the latter. A caching ratio below one can occur because the cache always load an entire cache lines (e.g. 32 bytes) which may be more than needed. A caching ratio greater than one can only occur if multiple threads require some of the same input rows which can be served by the cache avoiding global memory accesses. For most matrices, a caching ratio above one was achieved (Table 3.1), with an average of 8.2 ± 9.6. We found that the caching ratio correlates significantly with the performance rate ($R^2 = 0.60$, $P \leq 0.001$).

To process each output element, a sub-warp needs to perform two sub-warp reductions (min and sum) and two storage operations (column index and value). The output size strongly depends on the compression that is realized. A high compression results

TABLE 3.1
Performance measurements for the first matrix-matrix multiplication $\mathbf{G_1B}$. *The second column shows the caching ratio, i.e., the ratio of the amount of data processed by the threads to the amount data read from global memory. The third column shows the compression factor. A high compression factor results in a higher efficiency of the sub-warp reduction and in less storing operations for the output rows. On average, 2.2 calls to* `MulLimited()` *are required, the first of which consumes 65 % of the computation time.*

Name	Performance rate [Gflops]	Cache Ratio	Compression	Iterations	First Mul[%]	Prepare[%]
cantilever	6.0	32.0	6.0	2	66	34
economics	0.9	2.9	1.1	2	63	44
epidemiology	3.2	1.9	1.6	1	100	64
harbor	7.0	20.8	7.0	2	66	35
mouse280	4.4	1.7	2.0	1	100	49
protein	8.2	27.4	8.2	2	68	33
qcd	4.2	7.7	3.9	2	54	40
ship	8.0	10.6	7.9	2	64	34
spheres	6.1	14.4	6.1	2	63	34
windtunnel	8.4	19.6	8.2	2	69	33
accelerator	2.4	1.6	3.1	2	71	33
amazon0312	1.7	0.7	2.0	1	100	38
ca-CondMat	0.8	1.3	1.5	3	36	42
cit-Patents	0.8	0.4	1.2	3	54	33
circuit	1.1	2.3	1.6	3	43	41
email-Enron	1.3	2.9	1.4	3	50	35
p2p-Gnutella31	0.3	1.1	1.0	2	49	55
roadNet-CA	1.2	1.2	1.4	1	100	42
webbase1m	1.6	6.6	1.3	4	42	34
web-Google	1.2	0.5	2.0	3	58	32
wiki-Vote	0.8	14.3	1.5	3	42	40
mean	3.3	8.2	3.3	2.2	65	39
standard deviation	2.8	9.6	2.6	0.8	20	8

in fewer output values and therefore less reduction and memory storage operations. A correlation analysis between the compression factor and the performance rate for computing $\mathbf{A}_1\mathbf{B}$, as reported in Table 3.1, showed a strong correlation ($R^2 = 0.93$, $P \leq 0.001$), showing that our kernel strongly benefits from a high compression factor.

The choice of the sub-warp size has several effects. A small sub-warp size, e.g., 2, causes more loads and stores, because at each iteration only two rows are merged and the intermediate result needs to be stored and loaded again. A larger sub-warp size allows immediate merging of many rows which reduces the number of load and store cycles into global memory. To assess the compression factor, we determined the number of nonzeros of the intermediate matrices when using a merge factor of 2, because this includes the intermediate sizes that are created by larger merge factors. The evolution of the compression is shown in Fig. 3.4. It can be seen that the highest compression gains are in the early levels.

If the sub-warp size is larger than the actual number of input rows, some threads of the sub-warp are not used. Furthermore, a large sub-warp size may become inefficient

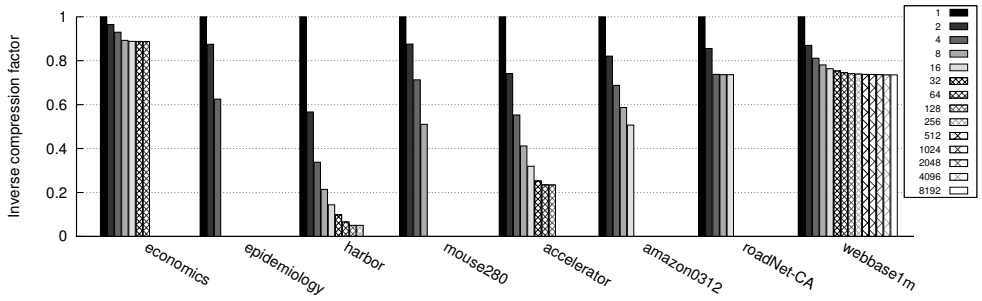


FIG. 3.4. *Compression effect during row merging. For a subset of 8 heterogeneous matrices, the number of nonzeros of intermediate matrices relative to the first level is reported, using a sub-warp size 2, i.e., merging 2 rows at each level. The strongest compression is achieved at early levels. Sizes are reported relative to level 1, which corresponds to the size of the intermediate array allocated by an Expansion-Sorting-Compression (ESC) method such as CUSP. It is not allocated by our method, instead the rows are fetched from the original matrix* $\mathbf{B}$ *when computing* $\mathbf{AB}$*. The last level has the output size. By using a larger sub-warp size than 2, e.g., 32, some intermediate levels can get skipped and a higher immediate compression is achieved at the first matrix-matrix multiplication.*

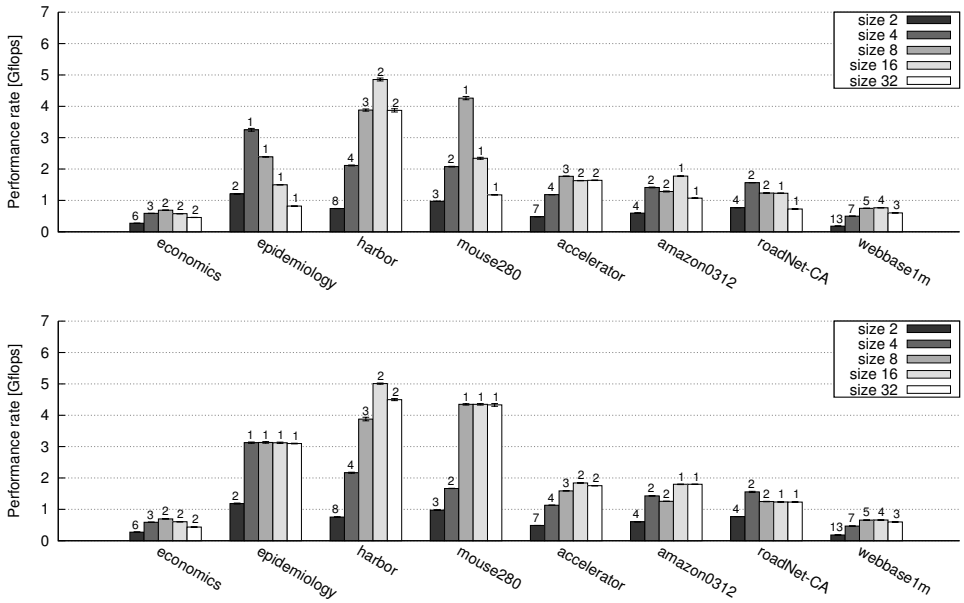


FIG. 3.5. *Effect of the sub-warp size on performance. Top: When using a fixed sub-warp size, the optimal size strongly depends on the matrix type. Bottom: By adjusting the sub-warp size, an improvement can be achieved and the performance depends less on the default sub-warp size. The numbers above the bars indicate the number of calls to* `MulLimited()`*.*

because the sub-warp sum and sub-warp min reductions (Listing 1) require $W \cdot \mathrm{ld}\, W$ operations per output, which eventually becomes inefficient for large sub-warps. If sub-warp sizes below the warp size (currently 32) are used, the sub-warps belonging to one warp will run synchronously only until the first sub-warp is finished, which may be inefficient if their computational load is heterogeneous. The combined effect of these aspects is difficult to predict analytically. Therefore, some tuning experiments

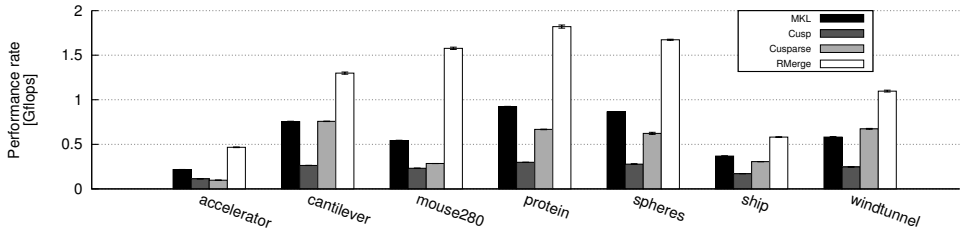


FIG. 3.6. *Performance rate for computing the Galerkin product pyramid during the AMG prepare step (for the symmetric positive definite matrices in Table 1.1). RMerge consistently outperforms the other methods.*

were performed for all possible sub-warp sizes (Fig. 3.5), showing, that adjusting the warp size for the last iteration is important and that larger sub-warp sizes (8, 16, and 32) result in better performance than small sub-warp sizes. The performance rates for sub-warp sizes 2, 4, 8, 16, and 32 were 0.65, 1.52, 2.10, 2.33, and 2.22 Gflops, respectively, on average (Fig. 3.5). Therefore, we used sub-warp size 16 as default.

3.5. Galerkin products. The necessity to compute Galerkin products arises during the preparation phase of of multigrid solvers, in particular AMG solvers. A coarse matrix $\mathbf{A_{i+1}}$ is computed as $\mathbf{A_{i+1}} = \mathbf{P}_i^T \mathbf{A}_i \mathbf{P}_i$ using the restriction operators $\mathbf{P}_i$. To measure the performance improvements for this operation, the time to compute this chain of products was measured. When $\mathbf{A_i}$ is small enough, e.g., width below 1000, a direct solver is more efficient and should be used. Therefore, the chain stops at this point. The measurements were performed for the symmetric positive definite matrices of Table 1.1 and the results are summarized in Figure 3.6. The average speed-up vs. MKL, Cusp, and Cusparse is 2.8, 5.4, and 3.4, respectively. The performance gains are not as strong as for matrix squaring, which may be related to the fact that the restriction operators are usually more balanced than the matrices used for the matrix squaring experiment.

4. Discussion. Our method for GPU-accelerated sparse matrix-matrix multiplication computes the product $\mathbf{C} = \mathbf{AB}$ by iterative row merging, similar to merge sort, except that elements with the same index are merged on the fly. Our performance measurements for matrix squaring, using a broad selection of 21 example matrices, show that SpGEMM is a highly heterogeneous problem. We noticed that all methods on average delivered higher performance for matrices with a higher compression factor. Cusp was the least affected by this and worked comparably well for irregular matrices with low compression factors. For eight selected matrices, we illustrate the heterogeneity with respect to the matrix structure, the distribution of nonzeros per row, the memory compression during row merging, and the effect of the sub-warp size. Despite the heterogeneity, our implementation performs better than the other CPU and GPU implementations for all 21 matrices, indicating that it performs well under broad conditions. The method might even be applicable for CPU processing since it optimizes the caching mechanisms by streaming a few rows at a time through the cache.

Our proposed method can be improved in several ways. Currently we keep the sub-warp size, i.e., the number of rows to be merged simultaneously, fixed, except for the last matrix-matrix multiplication, where the smallest possible sub-warp size is used. A heuristic could be developed to vary the sub-warp size depending on the

distribution of row sizes. The heuristic should be simple and fast to compute, because otherwise its benefits would be outweighed by the computational overhead. Currently, the largest row of **A** determines the number of row-merging iterations. Rows finished in early iterations are copied over to the next iteration, which is fast but can be avoided completely. An improvement could be to compute subsets of the rows with different kernels, because often 32 or less rows need to be merged which can be performed with single a kernel call, reducing the intermediate memory overhead. For situations where the intermediate memory is prohibitively large or even beyond the GPU memory size, the memory overhead could be further reduced by vertical blocking [5]. Currently, our method is implemented for devices providing an intrinsic 'shuffle' operation to exchange registers between threads within the same sub-warp. Thus, sub-warp reduction can be performed without using shared memory. For earlier Nvidia devices, the sub-warp reduction would require a small amount of shared memory, e.g., 8 bytes per thread for double precision, which would still be tolerable.

Our main motivation for developing this SpGEMM method was the Galerkin products required for AMG in the context of fluorescence-mediated tomography [21]. We showed that our method performs well for this application, now allowing to perform fluorescence reconstruction at higher quality using the same computational resources.

Acknowledgments. The authors would like to thank Weifeng Liu for providing data to allow comparison with his work [31].

REFERENCES

[1] R. R. Amossen and R. Pagh, *Faster join-projects and sparse matrix multiplications*, in Proceedings of the 12th International Conference on Database Theory, ACM, 2009, pp. 121–126.

[2] E. Attardo and A. Borsic, *GPU acceleration of algebraic multigrid for low-frequency finite element methods*, in IEEE Antennas and Propagation Society International Symposium, 2012, pp. 1–2.

[3] R. E. Bank and C. C. Douglas, *Sparse matrix multiplication package (SMMP)*, Adv. Comput. Math., 1 (1993), pp. 127–137.

[4] J. Becerra-Sagredo, C. Málaga, and F. Mandujano, *A novel and scalable multigrid algorithm for many-core architectures*, 2011. arXiv:1108.2045v1 [cs.NA].

[5] N. Bell, S. Dalton, and L. N. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM J. Sci. Comput., 34 (2012), pp. C123–C152.

[6] N. Bell and M. Garland, *Cusp: Generic parallel algorithms for sparse matrix and graph computations*, 2013. Version 0.4.0, http://cusp-library.googlecode.com.

[7] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, *Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*, ACM Trans. Graph., 22 (2003), pp. 917–924.

[8] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial*, SIAM, Philadelphia, PA, 2000.

[9] A. Buluç and J. Gilbert, *On the representation and multiplication of hypersparse matrices*, in Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, April 2008, pp. 1–11.

[10] A. Buluç and J. R. Gilbert, *Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments*, SIAM J. Sci. Comput., 34 (2012), pp. C170–C191.

[11] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[12] S. Dalton, N. Bell, and L. Olson, *Optimizing sparse matrix-matrix multiplication for the gpu*, tech. report, NVIDIA, 2013.

[13] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1–25.

[14] J. Demouth, *Sparse matrix-matrix multiplication on the gpu*, presentation, Nvidia, GTC, 2012.

[15] J. E. Dendy Jr., *Black box multigrid*, J. Comput. Phys., 48 (1982), pp. 366–386.

[16] R. D. Falgout, *An introduction to algebraic multigrid*, tech. report, Lawrence Livermore National Laboratory, 2006.

[17] M. Garland and D. B. Kirk, *Understanding throughput-oriented architectures*, Commun. ACM, 53 (2010), pp. 58–66.
[18] J. R. Gilbert, V. B. Shah, and S. Reinhardt, *A unified framework for numerical and combinatorial computing*, Comput. Sci. Eng., 10 (2008), pp. 20–25.
[19] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek, *Using GPUs to improve multigrid solver performance on a cluster*, International Journal of Computational Science and Engineering, 4 (2008), pp. 36–55.
[20] C. Gregg and K. Hazelwood, *Where is the data? Why you cannot debate CPU vs. GPU performance without the answer*, in IEEE International Symposium on Performance Analysis of Systems and Software, 2011, pp. 134–144.
[21] F. Gremse, B. Theek, S. Kunjachan, W. Lederle, A. Pardo, S. Barth, T. Lammers, U. Naumann, and F. Kiessling, *Absorption reconstruction improves biodistribution assessment of fluorescent nanoprobes using hybrid fluorescence-mediated tomography*, Theranostics, 4 (2014), pp. 960–971.
[22] M. Griebel, B. Metsch, D. Oeltz, and M. A. Schweitzer, *Coarse grid classification: A parallel coarsening scheme for algebraic multigrid methods*, Numer. Linear Algebra Appl., 13 (2006), pp. 193–214.
[23] M. Griebel, B. Metsch, and M. A. Schweitzer, *Coarse grid classification—Part II: Automatic coarse grid agglomeration for parallel AMG*, Tech. Report 271, Universität Bonn, 2006.
[24] A. Griewank and U. Naumann, *Accumulating Jacobians as chained sparse matrix products*, Math. Program., 95 (2003), pp. 555–571.
[25] F. G. Gustavson, *Two fast algorithms for sparse matrices: Multiplication and permuted transposition*, ACM Trans. Math. Software, 4 (1978), pp. 250–269.
[26] V. E. Henson and U. Meier Yang, *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Appl. Numer. Math., 41 (2002), pp. 155–177.
[27] J. Hoberock and N. Bell, *Thrust: A parallel template library*, 2013. Version 1.7.0, http://thrust.github.io/.
[28] D. E. Knuth, *The art of computer programming. Vol. 3. Sorting and Searching*, Addison-Wesley, Reading, Mass.; Harlow, 1998.
[29] J. Kraus and M. Förster, *Efficient AMG on heterogeneous systems*, in Facing the Multicore-Challenge II, no. 7174 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 133–146.
[30] S. Kunjachan, F. Gremse, B. Theek, P. Koczera, R. Pola, M. Pechar, T. Etrych, K. Ulbrich, G. Storm, F. Kiessling, and T. Lammers, *Noninvasive optical imaging of nanomedicine biodistribution*, ACS Nano, 7 (2013), pp. 252–262.
[31] W. Liu and B. Vinter, *An efficient gpu general sparse matrix-matrix multiplication for irregular data*, in IEEE 28th International Parallel & Distributed Processing Symposium, 2014.
[32] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, *Sparse matrix-matrix multiplication on modern architectures*, in 19th International Conference on High Performance Computing, 2012, pp. 1–10.
[33] A. Napov and Y. Notay, *An algebraic multigrid method with guaranteed convergence rate*, SIAM J. Sci. Comput., 34 (2012), pp. A1079–A1109.
[34] U. Naumann and O. Schenk, eds., *Combinatorial Scientific Computing*, Computational Science Series, Chapman & Hall / CRC Press, Taylor and Francis Group, Boca Raton, USA, 2012.
[35] NVIDIA Corporation, *CUDA C programming guide*, 2014. Version 6.0, http://developer.nvidia.com/cuda.
[36] ———, *Cusparse library*, 2014. Version 6.0, http://developer.nvidia.com/cusparse.
[37] G. Penn, *Efficient transitive closure of sparse matrices over closed semirings*, Theoret. Comput. Sci., 354 (2006), pp. 72–81.
[38] E. H. Rubensson, E. Rudberg, and P. Sałek, *Sparse matrix algebra for quantum modeling of large systems*, in Applied Parallel Computing. State of the Art in Scientific Computing, B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, eds., no. 4699 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 90–99.
[39] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.
[40] K. Stüben, *An introduction to algebraic multigrid*, in Multigrid, Academic Press, 2001, pp. 413–532.
[41] P. Sulatycke and K. Ghose, *Caching-efficient multithreaded fast multiplication of sparse matrices*, in Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, 1998, pp. 117–123.
[42] S. Van Dongen, *Graph clustering via a discrete uncoupling process*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 121–141.

[43] V. Vassilevska Williams, *Multiplying matrices faster than Coppersmith–Winograd*, in Proceedings of the 44th Symposium on Theory of Computing, ACM, 2012, pp. 887–898.
[44] L. Wang, X. Hu, J. Cohen, and J. Xu, *A parallel auxiliary grid algebraic multigrid method for graphic processing units*, SIAM J. Sci. Comput., 35 (2013), pp. C263–C283.
[45] R. Yuster and U. Zwick, *Fast sparse matrix multiplication*, ACM Trans. Algorithms, 1 (2005), pp. 2–13.