CES Seminar

Documentation

GPU-accelerated method for general sparse matrix-matrix multiplication

Daniel Becker

daniel.becker4@rwth-aachen.de
Matr.-No. 312312

December 3, 2016

Supervisor: Dr. rer. medic. Dipl.-Inf. Felix Gremse*†

*Experimental Molecular Imaging, RWTH Aachen University
†Software and Tools for Computational Engineering, RWTH Aachen University

1 Introduction

In the field of solving discretized differential equations the multiplication of matrices arises as a computational problem, e.g. in algebraic multigrids [1]. Usually these matrices are sparse which should be exploited when developing an algorithm for matrix-matrix multiplication. Moreover every entry in the resulting matrix of such a multiplication is independent from other entries. This knowledge should be used by computing the entries in parallel. Since the rows and columns are sparse the number of arithmetic operations for each entry is rather small which can be taken into account by computing the operations on a graphical processing unit (GPU). In the particular case of this study the sparse matrix-matrix multiplication is needed for the coarsening step (Galerkin products) of an algebraic multigrid solver for absorption and scattering reconstruction in small animal fluorescence-mediated tomography [2, 3]. Other applications could be graph algorithms like finding cycles, subgraphs or shortest paths as described in [4].

In [5] an algorithm (RMerge) for sparse matrix-matrix multiplication using iterative row merging on a GPU is presented and it actually outperforms other prominent methods like MKL (Intel), Cusp [6] or Cusparse [7]. This is why in this document a possible improvement of RMerge should be analyzed. The following sections describe the most important mathematical techniques and computational tools used by RMerge. Conclusively some measurements and tests of the new approach are performed.

1.1 General Matrix-Matrix Multiplication

1.1.1 Dense Case

When multiplying two dense matrices ($C = A \cdot B$ with $A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{k \times n}, C \in \mathbb{R}^{m \times n}$) the number of floating point operations (Flops) is in $\mathcal{O}\left(n^3\right)$ if A and B are square (i. e. $m = n = k$) since for every entry in C one row of A and one column of B are combined:

$$\begin{aligned} c_{i,j} = & \sum_{l=0}^{k-1} a_{i,l} \cdot b_{l,j} \\ & i \in 0..m-1 \,\text{and}\, j \in 0..n-1 \end{aligned} \tag{1}$$

There exist methods which further reduce the number of floating point operations. The first algorithm to mention is an algorithm which was published in 1969 by Volker Strassen who showed that one can compute such a matrix multiplication in $\mathcal{O}\left(n^{2.807}\right)$ by performing ring matrix multiplications recursively [8]. Further studies based on

this idea lead to the Coppersmith-Winograd algorithm which is in $\mathcal{O}\left(n^{2.375}\right)$ [9] and the best currently known method published by Virginia Vassilevska Williams and has order $\mathcal{O}\left(n^{2.373}\right)$ [10].

1.1.2 Sparse Case

In cases where sufficiently enough entries $a_{i,l}$ and $b_{l,j}$ are zero the matrices A and B are called sparse. Obviously one can observe that the majority of the multiplications and additions in equation 1 are neglectable since their result is zero. This fact should be exploited and several combinatorial problems arise how to multiply two sparse matrices in the most efficient way (cf. [11–14]). In most cases these methods compute each output row of a matrix squaring parallel and lead to an order which is $\mathcal{O}\left(n \cdot \mathrm{nnz}(A)\right)$ where $\mathrm{nnz}(A)$ denotes the number of non-zero entries in A. However, this is a very conservative assessment which is difficult to analyze because matrices can be very non uniformly. Note that since $1 < \mathrm{nnz}(A) < n^2$ the order of those algorithms reach from $\mathcal{O}\left(n\right)$ to $\mathcal{O}\left(n^3\right)$ which implies that at some point the algorithms in section 1.1.1 are again more efficient.

1.2 Algebraic Multigrid

Multigrid methods can be used to approximate the solution of the linear equation system $Au = b$. First the error on the finest grid is smoothed by e. g. a Gauss Seidel relaxation which leads to reduction of the residuum's high-frequency parts. Second the new residuum is translated to a more coarse grid with a restriction operator. Those new more coarse grids can be obtained by taking the geometric information into account (geometric multigrid) or by exploiting the structure of the matrix A (algebraic multigrid) [1]. A coarsened system matrix A_c for an algebraic multigrid (AMG) can be computed with the so called Galerkin operator [15]

$$\begin{aligned} & A_c = P^T A P \\ \text{with:} \quad & A : \text{original system matrix} \in \mathbb{R}^{n \times n} \\ & A_c : \text{coarsed system matrix} \in \mathbb{R}^{n_c \times n_c} \\ & P : \text{restriction operator} \in \mathbb{R}^{n \times n_c} \end{aligned} \tag{2}$$

Using this operator it is possible to iteratively create the pyramid of coarsened problems until a direct solver can be used efficiently, e. g. when A_c is in $\mathbb{R}^{1000 \times 1000}$. When this system is solved the coarse solution can be propagated back to the finest grid which removes the low-frequency parts of the residuum. When performing these steps iteratively this results in a solution which converges and is exact. [15, 16].

Creating the pyramid of coarsened problems requires the biggest part of the preparation time but it is still in linear order of $\mathrm{nnz}(A)$. Once the AMG is prepared for

a given problem the costs of one iteration is linear in $\mathrm{nnz}(A)$ too, which leads to a solving time in $\mathcal{O}(\mathrm{nnz}(A))$. Furthermore it is to point out that AMG solvers are not only interesting for solving linear equation systems but also as preconditioners for other algorithms like the conjugate gradient method [17].

1.3 GPU Programming

Graphics Processing Units (GPUs) offer massive computational power by using a large number of lightweight but relatively slow parallel threads [18]. This results in a high performance-to-price as well as performance-to-energy ratio when compared to CPUs. The threads are organized in so called warps which currently are an accumulation of 32 threads. All threads in a warp operate in synchronization to reduce the overhead that occurs due to instruction fetching and scheduling. To work optimally, all threads in a warp should follow the same code path while using different data. Additionally, warps can be further subdivided into subwarps. The number of threads that can run on a GPU device is limited by its resources, such as its shared memory or registers size. The memory used on GPUs is very fast but also limited (most powerful units have 16 to 24 GB). For every task that is to be executed on the GPU the relevant data has to be transferred from the CPU environment to the GPU. This represents a bottleneck for the performance of an application as the transfer rate is limited by the connection to the GPU (PCI Express). This means that data transfers should be as sparse as possible [19].

To cope with these difficulties and the many-thread environment, special programming techniques are required. These are provided by several high-level frameworks for compiling on the GPU such as CUDA [20] or OpenCL [21]. For this project we solely use NVIDIA graphics cards and consequently we make use of NVIDIAs own CUDA framework.

CUDA executes code in parallel on a given number of parallel threads with specifically defined functions called kernels. Each thread has a unique ID that can be accessed by the kernel which assigns the thread computing tasks. For convenience, the threads can be arranged in one-, two- or three-dimensional blocks which allows a much easier access to data represented in arrays of the respective dimensions. They should be constructed in such a way that the occupancy of all threads is maximized. For current GPUs there is a limit of 2048 threads per block. These blocks are again organized in a up to three-dimensional grid (figure 1). The number of blocks in a grid is determined by the total amount of threads required to process the data.

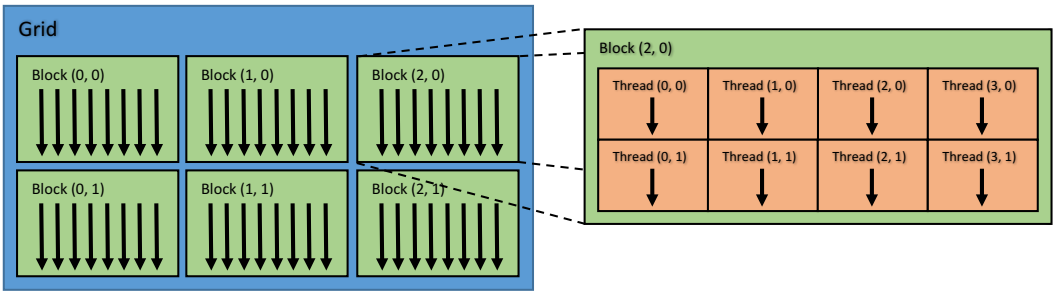

Figure 1: On the left a grid of six thread blocks organized in two dimensions. On the right a representation of one block in the grid. They each contain eight threads organized in two dimensions for a total of 48 threads.

The GPU has several layers of memory available. Each thread has a very limited but also very fast private memory, the registers. Each block of threads has a shared memory and above that every thread, no matter in which block or grid, has considerably slower access to the global memory. An important factor in the effectiveness of a kernel is which memory it reads from and writes to, so of course only little memory activity per kernel is to be desired.

In the field of sparse general matrix multiplication there exist several approaches which are implemented using GPUs e. g. [7, 20, 22]. Measurements and comparisons to the algorithm described in this document can be found in [5].

2 Iterative Row Merging

The matrix-matrix product $C = AB$ with $C \in \mathbb{R}^{m \times n}$ can be split up into m row merging operations which means that each row of C is computed as $c = aB$ where a and c are the rows of A and C respectively. In the following this operation is analyzed in more detail. The product $c = aB$ represents a linear combination of the rows of B which are selected and weighted by a. This is illustrated in figure 2 where three rows of the matrix B (2, 3, 5) are crucial for the resulting row c because only columns 2, 3 and 5 of a are non-zero. Note that c has the sparsity pattern of the summed up selected and weighted rows of B.

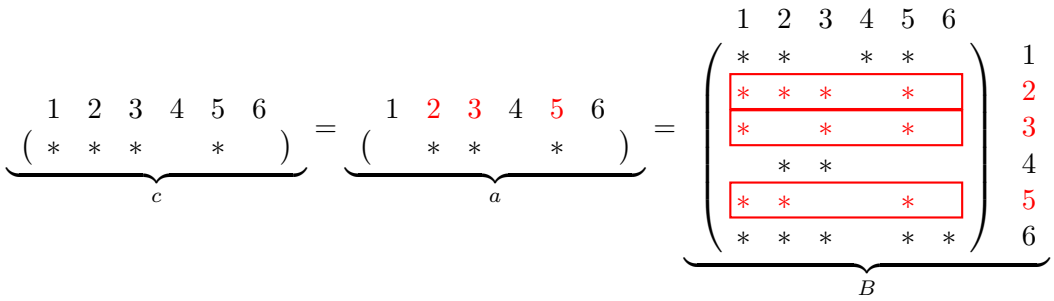

Figure 2: Example of the resulting pattern of a row when multiplying a sparse row vector with a matrix. Rows 2,3 and 5 are merged into one row.

The reason why this is a good approach in the context of GPUs is that each resulting row c can be computed by one subwarp which stores the input rows of B in one of its threads. The next step is to find a convenient way to implement a Cuda kernel which performs such a row merging operation. In the following the row merging alogrithm is called RMerge [5].

2.1 Implementation Limited Row Length Multiplication

The matrices which RMerge uses are stored in the so called CSR (compressed sparse row) format. This format consists of three vectors, first a vector that contains all $\text{nnz}(A)$, second the corresponding column indices of the values and third a vector which indicates when a new row starts in matrix A. This yields a total number of $\text{nnz}(A)$ floating values plus $\text{nnz}(A) + n$ index values which have to be stored for a matrix with height n. In a first step the matrix matrix product $C = AB$ (each row like in figure 2) can be calculated for a limited number of nonzero elements per row in A. The function is called MulLimited and consists of three steps:

1. Compute structure of result C: One kernel call calculates the row lengths of C
2. Compute vector with first index of each row and allocate memory for C
3. Values and columns indices of C are calculated with another kernel call.

As mentioned in section 1.3 the number of threads in a subwarp is limited to a given number. Therefore, RMerge uses a template variable for the subwarp size, we call it W with value 2, 4, 8, 16, or 32. Those values state the maximum number of nonzero entries in the longest row of A. Now for each row of A a function is called in which

every thread reads one row of B in the cache weights it with the corresponding entry of a and adds the outcome to the resulting row c. For a detailed description cf. [5].

2.2 Implementation for Arbitrary Row Length Multiplication

To make sure that the algorithm can handle matrices A with maximum row length greater than 32 those matrices A are split up into chain matrix matrix products on which multiple MulLimited calls are performed from right to left. The splitting is done iteratively in the sense that rows which are too long are split up into $\lceil \text{nnz}(a)/W \rceil$ rows. Using this procedure there emerge two matrices A_2 and G_1 where G_1 has at most W nonzero entries per row and A_2 may be divided further more:

$$C = AB = A_2G_1B = A_3G_2G_1B = ... = A_kG_{k-1}...G_1B \tag{3}$$

The splitting is done such that it is enforced that $G_2...G_{k-1}$ and A_k only contain one valued elements which is exploited in the computation of the intermediate result. The number of splits depends logarithmically on the maximum row length of A since each split reduces the maximum row length of A_i by a factor W.

2.3 Extension by Taking Multiple Rows per Thread

As a possible improvement for the existing RMerge algorithm two new versions were implemented. The idea is to minimize the communication between GPU threads which is realized is by assigning each thread multiple rows of B. The kernel functions of MulLimited are implemented in two new varieties, one which pulls two rows of B in each thread and one which pulls four rows, respectively. For the same subwarp size now different numbers of rows are merged in one iteration, which is why internally there is a new quantity *mergeFactor* replaces *subWarpSize* which simply specifies how many rows of B should be merged. As a consequence a merge factor of for example 32 results in the subwarp sizes 32, 16 and 8 for the three versions of RMerge. A second effect is that now the version with two threads per row has maximum merge factor of 64 instead of 32 and for the four row version it is possible to set the merge factor equal to 128. This property could be beneficial for matrices with maximum row length between 33 and 128 because they can be computed in one iteration instead of being split up (cf. section 2.2).

3 Measurements

Since the RMerge algorithm was presented and compared to other algorithms in [5] it has been used or cited in several publications. Liu et al. [23] use RMerge in

a framework for general sparse matrix-matrix multiplication and compare it with several other algorithms. They come to the conclusion that RMerge offers significant speedups in some of the matrices of table 1, mostly those which have short rows or are evenly distributed. On the NVIDIA Titan RMerge was the best considered algorithm with double precision. Liu uses this in his PhD thesis as well [24]. In other publications the algorithm is mentioned as an alternative to compute sparse matrix-matrix multiplications: [25–28]

The data which is used to test the performance of the new RMerge versions (cf. section 2.3) is the same as in [5] and is taken from the Florida sparse matrix collection [29] (cf. table 1). As test cases all matrices are squared and the performance rate is the measured quantity. Performance rate means the ratio of the arithmetic workload and the processing time, where the arithmetic workload Gflops(A, B) equals twice the number of nontrivial scalar multiplications [30]. All measurements are executed on a PC with *Windows 8.1 Enterprise* as operation System equipped with an *i5-4570* 3.20 GHz, 24 GB RAM and a *NVIDIA GeForce GTX TITAN*.

Table 1: Extract of the Florida sparse matrix collection [29] which is used to test the algorithm. Alongside the name of each matrix the following columns are the size, the number of non-zero entries as well as the maximum and average number of non-zero entries per row. In addition the floating point operations to square the matrices are stated. The last column gives the compression as the ratio of number of non-zero multiplications to nnz(AA).

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

To evaluate which version of RMerge works the best with which merge factor all combinations are measured. The results can be compared in different ways, first the smallest common possible merge factor is analyzed (cf. figure 3). Looking at the resulting performance rates it is obvious that the original RMerge with one row per thread outperforms the new implementations in every matrix square product. In a second investigation the results for the best average merge factor from [5] is visualized in figure 4. Here the performance rises in general but the one row per thread RMerge is still mostly faster than the new versions, except for some matrices. In a last plot (cf. figure 5) the highest possible merge factor for each

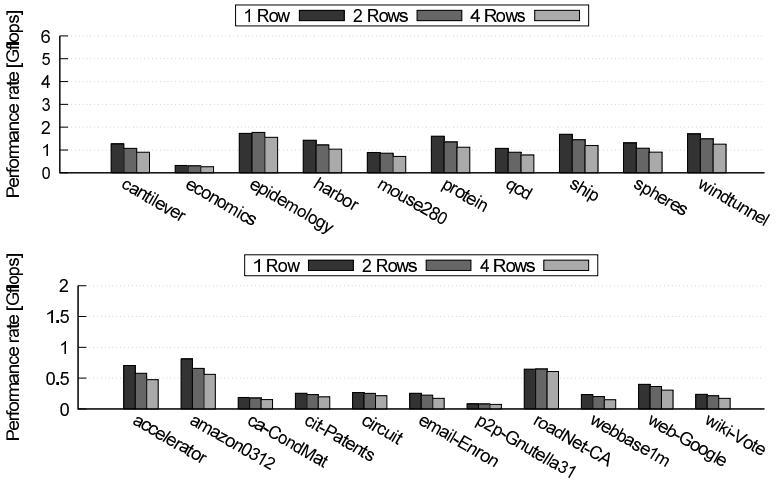

Figure 3: Performance rates for merge factor equal to 4. For each matrix one bar represents the different RMerge versions with one, two or four rows per thread. On top the more regular test matrices, at the bottom the more irregular matrices. Note that both plots are scaled to different maximum values due to better visibility. In both plots the bars are rather small because a merge factor of four mostly does not result in the best performance but one can see that with such a small merge factor the original RMerge outperforms the other versions.

version is analyzed, it turns out that now for the matrices *qcd* and *spheres* a huge performance boost is reached due to the merge factor of 128. On average both two and four rows per thread implementations offer a small speedup but there are also matrices for which one row per thread is the most suitable setup. Nevertheless, there are matrices like *qcd*, *webbase1m*, *roadNet-CA*, or *spheres* where multiple rows per thread offer better performance of 20 % to 44 %.

As a final step default merge factors for RMerge using 2 and 4 rows per thread need to be estimated. Although, for some matrices the speedup is quite high it is the case that the average speedup is 3.25 % and 2.18 % for RMerge with 2 and 4 rows per thread, respectively, using a merge factor of 32 in both cases. Those speedups result from the average performance rates of the three versions with their best merge factors. 1.67 GFlop/s for the 1 row per thread version compared to 1.73 GFlop/s and 1.71 GFlop/s for the 2 and 4 rows per thread versions.

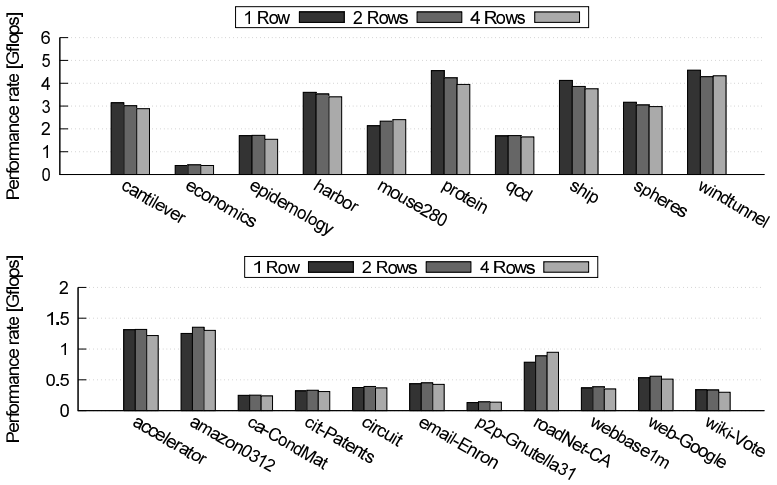

Figure 4: Performance rates for merge factor equal to 16. Distribution analogously to figure 3. The performance is much better compared to merge factor four and e.g. for *roadNet-CA* RMerge with four rows per thread works better than the others.

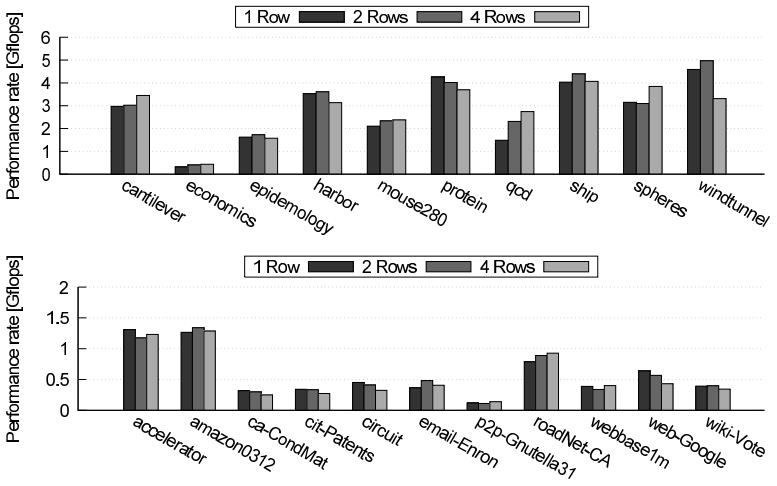

Figure 5: Performance rates for each RMerge version with the highest possible merge factors, i.e. 32, 64 and 128. Again the plots constructed like figure 3. Note that for *qcd* and *spheres* the now possible merge factor of 128 offers a huge benefit compared to the original RMerge.

4 Conclusion & Outlook

In this document a general algorithm for sparse matrix matrix multiplication was analyzed and extended to decide whether the performance could be increased. After describing the idea of iterative row merging the new extension was introduced where one thread of a GPU gets more work done, namely process more than one row for the merge operation. Finally test cases were discussed to determine if the new versions outperform or worsen the performance rate. It turns out that there is a minor average speedup but it is highly dependent on the input matrix if a performance boost is achieved or not.

As future work one could analyze whether it is useful to let a GPU thread take even more rows (i.e. 8, 16 or even arbitrary). Another possible point of interest could be the handling of non uniform matrices which means that there are some rows with many nonzero entries but in average the number is smaller. Those rows could be identified and treated differently from the regular row merging. This could lead to less idle GPU use.

References

[1] William L Briggs, Van Emden Henson, and S. F McCormick. *A multigrid tutorial*. SIAM, Philadelphia, PA, 2000. 2, 3

[2] Felix Gremse, Benjamin Theek, Sijumon Kunjachan, Wiltrud Lederle, Alessa Pardo, Stefan Barth, Twan Lammers, Uwe Naumann, and Fabian Kiessling. Absorption reconstruction improves biodistribution assessment of fluorescent nanoprobes using hybrid fluorescence-mediated tomography. *Theranostics*, 4:960–971, 2014. 2

[3] Felix Gremse and Fabian Kiessling. Noninvasive optical imaging: Hybrid optical imaging. In Anders Brahme, editor, *Comprehensive Biomedical Physics*. Elsevier, Amsterdam, Netherlands, 2014. 2

[4] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005. 2

[5] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, 2015. 2, 5, 6, 7, 8, 9

[6] Nathan Bell and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2013. Version 0.4.0, http://cusp-library.googlecode.com. 2

[7] NVIDIA Corporation. Cusparse library, 2014. Version 6.0, http://developer.nvidia.com/cusparse. 2, 5

[8] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, 1969. 2

[9] Don Coppersmith and Samuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9(3):251–280, 1990. 3

[10] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith–Winograd. In *Proceedings of the 44th Symposium on Theory of Computing*, pages 887–898. ACM, 2012. 3

[11] Aydin Buluç and John R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM J. Sci. Comput.*, 34(4):C170–C191, 2012. 3

[12] P.D. 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*, pages 117–123, 1998. 3

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

[14] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Software*, 4(3):250–269, 1978. 3

[15] R. D. Falgout. An introduction to algebraic multigrid. Technical report, Lawrence Livermoore National Laboratory, 2006. 3

[16] Artem Napov and Yvan Notay. An algebraic multigrid method with guaranteed convergence rate. *SIAM J. Sci. Comput.*, 34(2):A1079–A1109, 2012. 3

[17] Nathan Bell, Steven Dalton, and Luke N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM J. Sci. Comput.*, 34(4):C123–C152, 2012. 4

[18] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53(11):58–66, 2010. 4

[19] 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*, pages 134–144, 2011. 4

[20] NVIDIA Corporation. CUDA C programming guide, 2014. Version 6.0, http://developer.nvidia.com/cuda. 4, 5

[21] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Design & Test*, 12(3):66–73, May 2010. 4

[22] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2013. Version 1.7.0, http://thrust.github.io/. 5

[23] Weifeng Liu and Brian Vinter. A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85:47–61, 2015. 7

[24] Weifeng Liu. Parallel and scalable sparse basic linear algebra subprograms. 2015. 8

[25] Grey Ballard, Christopher Siefert, and Jonathan Hu. Reducing communication costs for sparse matrix multiplication within algebraic multigrid. *SIAM Journal on Scientific Computing*, 38(3):C203–C231, 2016. 8

[26] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *arXiv preprint arXiv:1510.00844*, 2015. 8

[27] K Rupp, J Weinbub, F Rudolf, A Morhammer, T Grasser, and A Jüngel. A performance comparison of algebraic multigrid preconditioners on cpus, gpus, and xeon phis. 2015. 8

[28] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jüngel, and Siegfried Selberherr. Viennacl—linear algebra library for multi-and many-core architectures. *SIAM Journal on Scientific Computing*, 38(5):S412–S439, 2016. 8

[29] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Software*, 38(1):1–25, 2011. 8, 9

[30] Steven Dalton, Nathan Bell, and Luke Olson. Optimizing sparse matrix-matrix multiplication for the gpu. Technical report, NVIDIA, 2013. 8

List of Figures