High Performance Multi-GPU SpMV for Multi-component PDE-based Applications

Ahmad Abdelfattah, Hatem Ltaief, and David Keyes

Extreme Computing Research Center

Division of Computer, Electrical, and Mathematical Sciences and Engineering
King Abdullah University of Science and Technology, Thuwal, KSA
{Ahmad.Ahmad,Hatem.Ltaief,David.Keyes}@kaust.edu.sa

Abstract. Leveraging optimization techniques (e.g., register blocking and double buffering) introduced in the context of KBLAS, a Level 2 BLAS high performance library on GPUs, the authors implement dense matrix-vector multiplications within a sparse-block structure. While these optimizations are important for high performance dense kernel executions, they are even more critical when dealing with sparse linear algebra operations. The most time-consuming phase of many multicomponent applications, such as models of reacting flows or petroleum reservoirs, is the solution at each implicit time step of large, sparse spatially structured or unstructured linear systems. The standard method is a preconditioned Krylov solver. The Sparse Matrix-Vector multiplication (SpMV) is, in turn, one of the most time-consuming operations in such solvers. Because there is no data reuse of the elements of the matrix within a single SpMV, kernel performance is limited by the speed at which data can be transferred from memory to registers, making the bus bandwidth the major bottleneck. On the other hand, in case of a multi-species model, the resulting Jacobian has a dense block structure. For contemporary petroleum reservoir simulations, the block size typically ranges from three to a few dozen among different models, and still larger blocks are relevant within adaptively model-refined regions of the domain, though generally the size of the blocks, related to the number of conserved species, is constant over large regions within a given model. This structure can be exploited beyond the convenience of a block compressed row data format, because it offers opportunities to hide the data motion with useful computations. The new SpMV kernel outperforms existing state-of-the-art implementations on single and multi-GPUs using matrices with dense block structure representative of porous media applications with both structured and unstructured multi-component grids.

1 Introduction

Simulating the flow in porous media enables the petroleum industry to characterize the production potential of oil fields and to optimize their development. It is also used in geological sequestration of carbon dioxide to mitigate anthropogenic

climate change. The most time-consuming operation (up to 80% of the total simulation time) in such simulations is the solution of large, sparse spatially structured or unstructured linear systems at each integration time step. The Sparse Matrix-Vector multiplication (SpMV) is the innermost computational kernel in such solvers and therefore, its performance impacts directly the solvers' overall performance. As opposed to more compute-intensive Level 3 BLAS kernels (e.g., matrix-matrix multiplication), the SpMV performance suffers from lack of data reuse and is thus limited by the speed at which data can be transferred from memory to registers. For the multi-species versions of the aforementioned applications and many others that drive investment in high performance, the resulting Jacobian has a dense block structure. For contemporary petroleum reservoir simulations, the block size typically ranges from three to a few dozen, and still larger blocks are relevant within adaptively model-refined regions of the domain. This structure can be exploited beyond the convenience of a block compressed row data format, because it offers opportunities to hide data motion with useful computations.

We leverage optimization techniques, such as register blocking and double buffering, introduced in the context of KBLAS [2], a Level 2 BLAS high performance library on GPUs, originally designed for dense matrix-vector multiplications. While these optimizations are important for high performance dense kernel executions, they are even more critical when dealing with sparse linear algebra operations, due to irregular memory accesses and low compute-intensity kernels. The new SpMV kernel outperforms existing state-of-the-art implementations on GPUs using matrices with dense multi-component blocks on structured-grid and random spatial block distributions. A multi-GPU SpMV interface allows simulation of larger problem sizes, while increasing the level of concurrency.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 reviews the source of sparse with dense block Jacobian structure in mesh-based PDE applications. Section 4 describes the framework for a uniform design strategy for such matrices and presents its different features and functionalities. The implementation details of the high performance SpMV kernels are given in Section 5. Section 6 shows the SpMV performance results on GPUs and compares it against state-of-the-art high performance SpMV implementations with various data layouts. Section 7 illustrates the performance impact after integrating our SpMV kernel into a sparse iterative solver library and we conclude in Section 8.

2 Related Work

The literature is rich in contributions for GPU-accelerated SpMV. Bell and Garland [7] proposed SpMV implementations for several formats including Compressed Sparse Row (CSR), ELLPACK [14], and the Coordinate (COO) format. They also proposed HYB, which is a hybrid format that combines both the ELLPACK format with the COO format, in an effort to reduce the padding overhead of the ELLPACK format.

Monakov et al. [18] proposed a sliced version of the ELLPACK format, where each slice is separately stored in the ELLPACK format. Vázquez et al. [20] proposed the ELLPACK-R format that adds auxiliary information to avoid computing the extra padded zeros. Choi et al. [9] proposed a parameterized blocked version of the ELLPACK format that proves to be competitive for block-sparse matrices, although it is restricted to certain block sizes, and targets mainly Fermi generation GPUs. Kreutzer et al. [16] generalized the sliced ELLPACK format to the SELL-C- σ format [15], in an effort to provide a unified sparse storage format across different architectures. The SELL-C- σ format has been improved and optimized for GPUs by Antz et al. [3], by introducing some zero padding to satisfy the memory constraints of the GPU architecture, hence called the SELL-P format.

Ashari et al. [4] proposed an adaptive algorithm for SpMV using the CSR format (called ACSR), where additional metadata are used with the standard CSR format that help achieve better GPU utilization. ACSR is mainly proposed for adaptive graph applications, where the structure of the graph adjacency matrix changes frequently, thus making the preprocessing step a serious bottleneck.

We are mainly interested in the Blocked Sparse Row (BSR) format, which is the blocked version of the CSR format. It was first introduced for CPU architectures by Im et al. [11][12]. The BSR format targets sparse matrices that are naturally blocked, as shown in Figure 1. It uses one integer per block to store its column index, as well as an integer to denote the start of every block row. In cases specific to structured grid problems, Godwin et al. [10] proposed a format called Column Diagonal Storage (CDS), which assigns only one integer for a group of blocks located at the same diagonal/off-diagonal. The work by Choi et al. [9] suggested GPU specific optimizations for the BSR format that were not enough to outperform the cuSPARSE HYB kernel [7]. They concluded that the BSR will be dominated by a reduction step that is affected by the number of blocks per block row. This work revisits the BSR formats and proposes some optimization techniques for a wide range of block sizes. The BSR format supports any structure of a block-sparse matrix.

Among all the aforementioned sparse formats, the cuSPARSE HYB format along with the SELL-P format usually achieve the best performance on the GPU across several matrices, as long as the matrix structure does not change during the simulation.

Fig. 1. The BSR format.

3 Multi-component Applications

Numerous applications result in sparse matrices of dense blocks, where the first nontrivial block size is 2 (e.g., streamfunction and vorticity in fluid dynamics) and the block size ranges up to hundreds in realistic contemporary applications that drive high performance computing (e.g., detailed kinetics models of hydrocarbon-air combustion). In the applications expressed as PDEs that motivate this work, the number of components is related to the number of fields defined over the domain. The blocks are square because each field (e.g., density, pressure, momentum, internal energy, concentration of a given species in a given phase in a given charge state) has its own conservation equation.

If the conservation equations were decoupled, all blocks would be diagonal and the data structures designed for this paper would not be relevant for high performance. However, most systems of conservation laws (see Eq. 1) couple the fields defined at each point through possibly several types of physical interdependencies.

$$\frac{\partial(\rho\phi_k)}{\partial t} + \nabla \cdot (\rho \mathbf{v} \phi_k) - \nabla \cdot (\mu_k \nabla \phi_k) = F_k(\phi_1, \phi_2, \dots, \phi_K), \quad k = 1, 2, \dots, K. \quad (1)$$

In the typical convection-diffusion-reaction system shown, the convection terms couple the momenta to all convected components. The momenta are products of density (ρ) and velocities ($\mathbf{v}$), and the density is a function of the mass fractions and thermodynamic state of all of the species ($\phi_1, \phi_2, \dots, \phi_K$) in the system. The gradient operator acting on the density couples degrees of freedom across the grid points in the stencil, so the typical off-diagonal component of the off-diagonal blocks is nonzero. The diffusion terms couple the degrees of freedom to each other because the diffusion coefficients (μ_k) are also complex functions of the mass fractions and thermodynamic state at each point. Again, the gradient operator couples the degrees of freedom across the grid points in the stencil, so that the off-diagonal blocks are best regarded as fully dense. The structure of the reaction terms for the creation and consumption of each component (F_k) may lead to some exploitable sparsity within the diagonal blocks since not all components react with all others. However, the diagonal blocks are often factored as part of a block preconditioner to pre-scale the system and the blocks are best regarded as full in this case.

Equation 1 is a simplified schema of systems described by first principles in, e.g., [8] for porous media applications or [21] for reacting flows. In turn, such systems may be regarded as embedded in multiphysics applications for which computational modelers increasingly prefer fully implicit solvers [13] for reasons of numerical efficiency, stability, and/or robustness. Past generations of modelers lacking powerful high performance solvers have tended to employ operator splitting to solve such systems in a series of steps that leave behind first-order temporal splitting errors and potentially destabilizing mechanisms. Splitting also weakens temporal locality and arithmetic intensity. Contemporary high performance solver software allows such users to more fully exploit the inexpensive flops of a GPU and reduce expensive memory thrashing.

4 A Uniform Design Strategy for SpMV

We propose a uniform design strategy for the SpMV kernel based on the BSR format. Such strategy builds on the same design ideas used in the KBLAS library[1][2].

Fig. 2. Hierarchical register blocking.

Hierarchical Register Blocking.

Using the BSR format, the matrix is naturally described using square dense *blocks* of size $bs \times bs$, with no padding assumed. We define the term *working set* to denote the minimum amount of work assigned to a thread block (TB) at one time. A working set is generally different from a *block*. A working set can be equivalent to: 1) one block, 2) multiple adjacent blocks in the same block row, or 3) part of a block. The block size bs drives how a working set is defined. A working set should always fit into registers. Its dimensions are assumed to be $nb \times width$, where nb and $width$ are two design parameters. A working set is always processed using a *thread array*. In general, a TB consists of N_{ta} thread arrays ($N_{ta} \geq 1$). This allows a TB to process multiple working sets concurrently. Each thread array is restructured to $nb \times N_{tg}$ threads, where every nb threads are called a *thread group* (nb can be any value, not necessarily a warp). A working set is further subdivided into vertical slices. Each vertical slice is assigned to one thread group. A vertical slice is further subdivided horizontally among threads in a thread group. As shown in Figure 2, each thread needs ept registers to store a segment of a row of the matrix. The $width$ is equal to $N_{tg} \times ept$.

Double Buffering. The proposed strategy incorporates a double buffering scheme in order to hide memory latency and overlap computation with data prefetching. Each TB requires a storage large enough to fit at least two working sets. Considering Figure 2, each thread needs two independent buffers, each one is ept in length.

5 High Performance Kernel Implementations

This section discusses how the aforementioned design ideas can be applied for the SpMV kernel.

Dividing Block Size Range. We propose three kernels to divide the spectrum of the block size. All kernels are available through the KSPARSE library¹. Kernel *K1* can process small values of bs , strictly 2 through 5. Kernel *K2* is

¹ <http://ecrc.kaust.edu.sa/Pages/ksparse.aspx>

applicable to medium *blocks* starting from 5 up to 45. Kernel *K3* is assigned for large *blocks* starting from 45 and beyond.

These ranges are not strict, except for *K1*, which cannot be applied to blocks larger than 5×5 . The above ranges are specified according to our experiments on a Kepler K20c GPU. With very little programming effort, ranges can be adapted to the test environment.

Small Blocks (*K1*). Kernel *K1* assumes that a warp can read one or more *blocks* in one memory transaction. So, a working set spans multiple adjacent blocks in a block row. So $nb = bs$, but *width* is multiples of *bs*. Figure 3 shows the hierarchy of the overall kernel design, where thread arrays are strictly warps. Each warp is truncated to be fully divisible by bs^2 , then restructured to $nb \times N_{tg}$ threads, where $N_{tg} = \lfloor \frac{32}{bs^2} \rfloor \times bs$. This restricts *ept* to be 1. A truncated warp can process $Nbrows_{ta}$ consecutive block rows ($Nbrows_{ta} \geq 1$). If, for example, $bs = 3$, then a warp is truncated into 27 threads, that read 3 blocks at one memory transaction.

Medium Blocks (*K2*). The design of *K2* is similar to *K1* (Figure 3), except that thread arrays are not strictly warps. *K2* assumes that a working set is always equivalent to one block. This implies $nb = width = bs$. The value N_{tg} is in a tuning parameter $\leq nb$. Since $width = bs$, it is not always possible to subdivide the working set into vertical slices of the same width (e.g. consider $bs = 17$). Given a block size bs , and a number of thread groups $N_{tg} \leq bs$, we define, 1) $ept_{max} = \lceil \frac{bs}{N_{tg}} \rceil$, and 2) $threshold = (N_{tg} - (bs \bmod N_{tg})) \bmod N_{tg}$. To hold one block in registers, every thread needs at maximum ept_{max} registers. However, threads in thread groups $< threshold$ use only $ept_{max} - 1$ registers. This amount is different from the total register usage, which is decided solely by the compiler.

Large Blocks (*K3*). Since *K3* targets very large blocks, each block can be subdivided into multiple working sets. Therefore, *K3* uses multiple TBs to process a block. The dimensions of the working set are exposed as tuning parameters. Each group of $\lceil \frac{bs}{nb} \rceil$ TBs behave like a mini-grid of a dense GEMV kernel around every matrix block. Figure 4 shows the structure of the *K3* grid. The next section presents the performance results of the proposed SpMV kernels.

Fig. 3. *K1/K2* structure.

6 Performance Results and Analysis

We denote an SpMV operation using the BSR format as BSRMV. The performance of KSPARSE is compared against the NVIDIA BSRMV kernel available in cuSPARSE. We also compare against the NVIDIA HYB format [7], and the SELL-P format [3] (which is based on the

SELL-C- σ format [15]). The two latter formats are known for their high performance and friendliness to GPU architectures. We consider only real matrices. Performance is shown for double precision. However, we emphasize that the same performance behavior, including speedups and achieved memory bandwidth, holds also for single precision. The online version of KSPARSE supports both single and double precisions.

System Setup. The performance test experiments are conducted on a system with 16-core Intel Xeon CPU E5-2650 (2.00GHz) and four Tesla K20c GPU (ECC off). The system runs Ubuntu 14.04.1 LTS, CUDA driver version 340.32, and CUDA Toolkit 5.5. Through a set of bandwidth micro-benchmarks, the sustained memory performance is measured at 184.18 GB/s. This is 88.5% from the theoretical memory bandwidth (208 GB/s). Results are properly averaged among multiple runs.

Matrix Test Suite. We use synthetically generated matrices using a matrix generator that produces two extreme cases for matrices with dense block substructure. The first is a structured matrix that has exactly seven block diagonals, which is typical for matrices arising from 3D scalar structured-grid problems from low-order finite differences. The second matrix type has a nonzero block main diagonal and otherwise random column placements of the remaining non-zero blocks per row. The number of non-zero blocks per block row is randomly generated between two input values to the generator. Sparsity is kept at least at 99.9%. The number of rows/columns of the matrix is at least 1M, with at least 99.9% sparsity. Figure 5 shows spy plots for sample structured and random matrices produced from our matrix generator. PDE applications on grids arising from low-order local discretizations of control volume or finite element type, unstructured by virtue of irregular domain geometry or adaptive refinement, lie between the structured and random cases. They tend to have superior regularity in the number of blocks per row and superior spatial locality relative to the random matrices. For space considerations, they are not considered herein.

Single GPU Performance. The single GPU performance is shown in Figure 6. For structured matrices, the cuSPARSE-HYB and SELL-P kernels can

Fig. 4. $K3$ structure.

Fig. 5. Spy plots for sample structured/random matrices.

exploit the structure and match or outperform KSPARSE for relatively small block sizes. As the block sizes gets bigger, KSPARSE can achieve up to 42% and 49% speedups against cuSPARSE-HYB and SELL-P respectively. Against cuSPARSE-BSR, and starting block sizes 6 and up, KSPARSE is between 1.34x to 4.11x faster. For the random case tests, KSPARSE outperform all other kernels in most cases, achieving speedups up to 2.15x, 1.52x, and 4.08x against cuSPARSE-HYB, SELL-P, and cuSPARSE-BSR respectively. We note that the cuSPARSE-HYB kernel fails to run for some tests, where its memory footprint exceeds the GPU memory. **Multi-GPU Scaling.** A multi-GPU BSRMV can be broken down into individual calls to the single GPU BSRMV kernel, provided that matrix distribution among GPUs allows the local sparse submatrix to be individually described using the BSR format. As an example, we present the multi-GPU scaling when block rows and their respective column indices are assigned in a 1D cyclic manner among GPUs. The integer array that holds row pointers has to be reevaluated for each GPU. After building the local row pointer array, it is straightforward to launch a BSRMV kernel on each GPU. Figure 7 shows the double precision performance of KSPARSE for 1, 2, and 4 GPUs on the same node. The figure shows an almost linearly scaling performance. The only overhead of using multi-GPUs is a synchronization point to ensure that all GPUs are finished.

7 Impact on Sparse Iterative Solver

This section presents the impact of using KSPARSE on a sparse iterative solver based on GMRES [19]. The purpose of this part is to show the speedup achieved by using KSPARSE instead of cuSPARSE when calling the SpMV kernel based on the BSR format. Solver-related issues like implementation quality, convergence rate, and others are beyond the scope of this work. We use CUSP [6] as the library implementing GMRES. This is an open-source C++ library that provides a high level template implementation of sparse linear solvers. CUSP is used by PETSc [5] within its GPU component [17]. We use an example implementation of GMRES from CUSP that uses a single GPU. We have added

(a) Structured case.

(b) Random case.

Fig. 6. Single GPU performance.

support for the BSR format inside CUSP. We show the GMRES execution time for a non-preconditioned system with at least a million unknowns. The solver stops at a residual norm of 10^{-5} or after reaching 1000 iterations. The solver is also set to restart every 50 iterations. We use synthetically generated structured matrices similar to Figure 5(a). Figure 8 shows the execution time of GMRES. As expected from the SpMV performance, cuSPARSE achieves better execution time for GMRES using block sizes 2 and 4, achieving speedups up to 12%. For block size 3, the KSPARSE-based solver almost matches its cuSPARSE counterpart. For block sizes 5 and up, the KSPARSE-based GMRES achieves speedups ranging 1.14x to 2.62x, depending on the performance of the BSRMV kernel given a particular block size.

8 Conclusion and Future Work

We have designed an SpMV kernel (part of the KSPARSE library) for solving large sparse structured or unstructured linear systems, in the context of applications that feature multi-component models, typified by contemporary porous media simulations, resulting in dense block Jacobian structure. Using a uniform

(a) Structured case.

(b) Random case.

Fig. 7. Multi-GPU performance.

design strategy based on hierarchical register blocking and double buffering optimization techniques, the new SpMV kernel outperforms existing state-of-the-art implementations on GPUs by achieving significant speedups using matrices with large dense block structure from applications with structured and unstructured multi-component grids. The proposed multi-GPU SpMV interface highlights an almost linearly scaling performance. Integrating our KSPARSE SpMV kernel into a sparse iterative solver library improves the overall performance up to 2.62x speedup, as the block size increases. We plan to investigate the implementation of KSPARSE on distributed memory environments and to study the impact of high network interconnect linking remote GPUs on the overall iterative solver.

Acknowledgment

The support of Saudi Aramco through research project RGC/3/1438 is gratefully acknowledged. The authors would like also to thank NVIDIA for their support and generous hardware donations as well as Pascal Hnon from TOTAL S.A. for fruitful technical discussions.

Fig. 8. Execution time of CUSP-GMRES.

References

1. KAUST BLAS. <http://ecrc.kaust.edu.sa/Pages/Res-kblas.aspx>
2. Abdelfattah, A., Keyes, D., Ltaief, H.: KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators. *ACM Transactions on Mathematical Software* (accepted subject to revision) (2014), <http://arxiv.org/abs/1410.1726>, <http://arxiv.org/abs/1410.1726>
3. Antz, H., Tomov, S., Dongarra, J.: Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C- σ formats on NVIDIA GPUs. Tech. rep. (2014), <http://www.icl.utk.edu/sites/icl/files/publications/2014/icl-utk-772-2014.pdf>
4. Ashari, A., Sedaghati, N., Eisenlohr, J., Parthasarathy, S., Sadayappan, P.: Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. pp. 781–792. SC '14 (2014), <http://dx.doi.org/10.1109/SC.2014.69>
5. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zhang, H.: PETSc Web page. <http://www.mcs.anl.gov/petsc> (2014)
6. Bell, N., Garland, M.: CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. <http://cusplibrary.github.io/>
7. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. pp. 18:1–18:11. SC '09, ACM, New York, NY, USA (2009), <http://doi.acm.org/10.1145/1654059.1654078>
8. Chen, Z., Huan, G., Ma, Y.: *Computational Methods for Multiphase Flows in Porous Media*. Society for Industrial and Applied Mathematics, Philadelphia (2006)
9. Choi, J.W., Singh, A., Vuduc, R.W.: 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*. pp. 115–126. PPoPP '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1693453.1693471>
10. Godwin, J., Holewinski, J., Sadayappan, P.: High-performance Sparse Matrix-vector Multiplication on GPUs for Structured Grid Computations. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. pp. 47–56. GPGPU-5, ACM, New York, NY, USA (2012), <http://doi.acm.org/10.1145/2159430.2159436>

11. Im, E.J., Yelick, K.: Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In: Alexandrov, V., Dongarra, J., Juliano, B., Renner, R., Tan, C. (eds.) Computational Science ICCS 2001, Lecture Notes in Computer Science, vol. 2073, pp. 127–136. Springer Berlin Heidelberg (2001), http://dx.doi.org/10.1007/3-540-45545-0_22
12. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: Optimization Framework for Sparse Matrix Kernels. *Int. J. High Perform. Comput. Appl.* 18(1), 135–158 (Feb 2004), <http://dx.doi.org/10.1177/1094342004041296>
13. Keyes, D.E., McInnes, L.C., Woodward, C., Gropp, W.D., Myra, E., Pernice, M., Bell, J., Brown, J., Clo, A., Connors, J., Constantinescu, E., Estep, D., Evans, K., Farhat, C., Hakim, A., Hammond, G., Hansen, G., Hill, J., Isaac, T., Jiao, X., Jordan, K., Kaushik, D., Kaxiras, E., Koniges, A., Lee, K., Lott, A., Lu, Q., Magerlein, J., Maxwell, R., McCourt, M., Mehl, M., Pawlowski, R., Peters, A., Reynolds, D., Riviere, B., Rüde, U., Scheibe, T., Shadid, J., Sheehan, B., Shephard, M., Siegel, A., Smith, B., Tang, X., Wilson, C., Wohlmuth, B.: Multiphysics simulations: Challenges and opportunities. *International Journal of High Performance Computing Applications* 27, 4–83 (2013)
14. Kincaid, D., Oppe, T., Young, D.: ITPACKV 2D User’s Guide (1989), <http://www.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/>
15. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Bishop, A.: A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing* 36(5), C401–C423 (2014), <http://dx.doi.org/10.1137/130930352>
16. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Basermann, A., Bishop, A.R.: Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation. In: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops. pp. 1696–1702. IPDPSW ’12, IEEE Computer Society, Washington, DC, USA (2012), <http://dx.doi.org/10.1109/IPDPSW.2012.211>
17. Minden, V., Smith, B., Knepley, M.: Preliminary implementation of petsc using gpus. Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering (2010)
18. Monakov, A., Lokhmotov, A., Avetisyan, A.: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In: Patt, Y., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) High Performance Embedded Architectures and Compilers, Lecture Notes in Computer Science, vol. 5952, pp. 111–125. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11515-8_10
19. Saad, Y., Schultz, M.: GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing* 7(3), 856–869 (1986), <http://dx.doi.org/10.1137/0907058>
20. Vázquez, F., Fernández, J.J., Garzón, E.M.: A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 23(8), 815–826 (2011), <http://dx.doi.org/10.1002/cpe.1658>
21. Williams, F.A.: Combustion Theory. Benjamin/Cummings, Menlo Park (1985)