Accelerating the Conjugate Gradient Algorithm with GPUs in CFD Simulations

Hartwig Anzt[1], Marc Baboulin[2], Jack Dongarra[1], Yvan Fournier[3], Frank Hulsemann[3], Amal Khabou[2], and Yushan Wang[2]

[1] University of Tennessee, Innovative Computing Laboratory, Knoxville, USA
[2] Université Paris-Sud, Laboratoire de Recherche en Informatique, Orsay, France
[3] EDF R&D, Clamart, France

Abstract. This paper illustrates how GPU computing can be used to accelerate computational fluid dynamics (CFD) simulations. For sparse linear systems arising from finite volume discretization, we evaluate and optimize the performance of Conjugate Gradient (CG) routines designed for manycore accelerators and compare against an industrial CPU-based implementation. We also investigate how the recent advances in preconditioning, such as iterative Incomplete Cholesky (IC, as symmetric case of ILU) preconditioning, match the requirements for solving real world problems.

1 Introduction

A significant gap exists in-between the availability of open-source software libraries for sparse linear algebra computations on accelerators, and what is actually used in an industrial environment. An example is the *Code_Saturne* [5] package, a general purpose Computational Fluid Dynamics (CFD) software developed and used at Electricité de France (EDF). Among the main reasons behind this situation is the limited experience of how open-source packages, often coming from an academic environment, fit the demands of an industrial setting. Another concern is whether the accelerator hardware specifications, in particular the limited memory bandwidth of graphics processing units (GPUs), are suitable for real world applications. In this position paper, we address these two concerns by evaluating the performance of different implementations of the Conjugate Gradient (CG) method for two benchmarks with a finite volume origin. As the iterative solution process plays a key role in the simulation algorithm — it can account for up to 80% of the computational time in *Code_Saturne* — the performance improvements are quickly reflected in the overall runtime of the CFD simulation. Thus the main contribution of this work is to show that the use of GPU-enabled sparse linear algebra libraries in the framework of industrial applications allows for significant performance improvements with minimal implementation effort.

The rest of the paper is organized as follows. In Section 2 we provide some background about the industrial code, the software libraries, and the benchmarks that we consider. Section 3 reviews some strategies known to enhance the performance of iterative solvers on GPUs. This includes the optimization of the sparse matrix vector product (SpMV) which typically dominates the performance of Krylov solvers such as CG, and the use of kernel fusion for enhanced data locality. We also review some of the latest ideas on preconditioning techniques suitable for fine-grained hardware parallelism. In Section 4, we report some experimental results obtained using the different software packages to solve the CFD problems. We conclude in Section 5.

2 Problem setting and software framework

Code_Saturne [5] is a general purpose Computational Fluid Dynamics (CFD) software package developed and used at Electricité de France (EDF). It is based on a co-located finite volume approach, using a fractional time step method. This allows for any type of polyhedral mesh, though best results are usually obtained with regular, hexahedral meshes. The flux discretization uses a 2-point scheme, with contributions due to mesh non-orthogonalities added at the right-hand side and solved through sub-iterations. The matrix graph is thus based on face to cell adjacencies, leading to very sparse matrices. For a scalar variable on a hexahedral mesh, we have 7 non-zero entries per row (6 face neighbors + 1 diagonal). For a tetrahedral mesh, this even goes down to 5 non-zero entries per row. The benchmark problems we consider in this paper originate from Code_Saturne. As the problems are all symmetric and positive definite, they can be solved efficiently with the CG iterative solver. Enhancing the CG with a Jacobi preconditioner (diagonal scaling) typically improves both convergence and performance. We note that, in *Code_Saturne*, parallelism is handled via MPI and OpenMP. However, in our evaluation, we limit the parallelism to OpenMP, as we are considering single node performance only.

The **CUsparse** [10] software library is a collection of routines for sparse linear algebra computations on NVIDIA GPUs. It provides the main building blocks, such as the sparse matrix vector product kernel, matrix conversion routines, and incomplete LU (ILU) preconditioning techniques. Some basic iterative solvers such as CG are also available. Developed by NVIDIA, this library typically achieves very good performance on NVIDIA architectures.

MAGMA [8] is an accelerator-focused linear algebra library developed at the University of Tennessee. It provides backends for NVIDIA GPUs, Intel's Xeon Phi many-core accelerators (MIC), and any OpenCL-compatible system such as AMD GPUs. In addition being well-known for the dense linear algebra routines, MAGMA also contains a large variety of solvers, preconditioners, and eigensolvers for sparse linear systems. Comprehensive support for NVIDIA GPUs is provided, some basic routines and functionalities are also available in OpenCL and for the Xeon Phi.

ViennaCL [11] is a free open-source linear algebra and solver library written in C++. The functionality provided by ViennaCL overlaps significantly with the functionality provided by MAGMA. However, ViennaCL provides a unified interface for three fully supported compute backends using CUDA (for NVIDIA GPUs), OpenCL (for cross-vendor GPU-support), and OpenMP (for multi-core CPUs). Also, in contrast to MAGMA, the compute backends in ViennaCL can be switched at runtime.

In the experimental evaluation, we consider two benchmarks from the EDF application:

– The `bundle` problem is generated using one regular hexahedral mesh. The matrices are built from a matrix of size 16384 (average nnz per row is 7) for which we duplicate the initial mesh to obtain larger systems and study the scalability. Fig. 1 (left) shows the geometry of the domain in the numerical simulation.

- The bora problem originates from a mostly hexahedral mesh, but includes face subdivisions at non-conformal mesh joining interfaces. As a result, most matrix rows have 7 non-zero entries, but some rows have a higher number of non-zero entries. In this benchmark, we solve a linear system of size 10196476 and the geometry of the compute domain is shown in Fig. 1 (right).

For both benchmarks, the bundle and bora, the linear system to be solved is extracted from the Laplacian operator within the pressure correction step, with a right-hand side corresponding to the flow initialization. The matrix structure does not change over time, but the coefficients may change whenever the fluid properties vary. For these test matrices, we assume constant temperature and constant fluid properties, which means that the matrix coefficients remain constant.

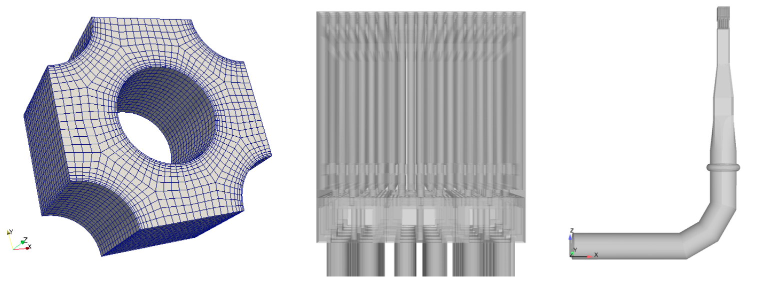


Fig. 1: Domain representation of bundle (left) and bora (right). The middle figure is a close-up of the upper part of the right figure. The discretization in this part is different from the rest of the domain, leading to a sparse matrix with an irregular pattern.

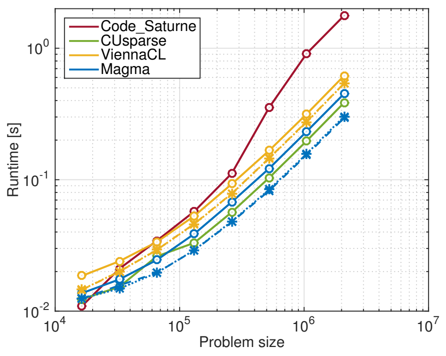
3 Sparse linear algebra on GPUs

The performance of Krylov methods like the Conjugate Gradient is generally bounded by the memory bandwidth of the hardware architecture used. Hence, optimizing the performance for these solvers is usually equivalent to optimizing the access to the GPU main memory. The implication is twofold: reducing the total amount of data that is read and written to the main memory, and organizing the memory access as coalescent reads [9]. As the CG and its preconditioned variant arise as a combination of matrix-vector and vector-operations, the optimization for coalescent memory reads boils down to the sparse matrix vector product. There exists extensive work on optimizing storage format and sparse matrix vector performance for GPUs, and in this work we focus on using the CSR, ELL, and SELLP formats, known to provide good performance [4]. To reduce the memory traffic, it is necessary to use algorithm-specific kernels that apply kernel-fusion to the basic linear algebra operations whenever possible [1]. More precisely, consecutive vector operations sharing some of the input or output data are merged into a single kernel, such that data, once loaded into the fast multiprocessor memory, is reused. See [2] for details on how this is achieved for the Conjugate Gradient solver used in this study. The Magma library implementations feature kernel fusion for the basic CG as well as the preconditioned variant. In ViennaCL, the concept of kernel fusion is applied to the basic CG, not yet for the preconditioned variant. This optimization will be included in a future release, which will bring the performance of ViennaCL closer to that of MAGMA for the preconditioned CG as well. Beside Jacobi, another preconditioner suitable for a large variety of problems is an incomplete LU factorization [12]. A drawback of ILU preconditioners is the sequential nature of both the preconditioner generation via Gaussian-Elimination, and the sparse triangular solves in the preconditioner application. Also, approaches using level-scheduling or multi-color ordering for enhancing the concurrency often fail to exploit the fine grained parallelism provided by manycore architectures. Given this background, the recently proposed iterative approach to ILU preconditioning has attracted much attention [6, 7]. On GPUs in particular, the forward and backward substitutions traditionally used to solve the sparse triangular systems in every outer Krylov iteration are expensive. Replacing those with a few Jacobi sweeps can accelerate the overall solution process significantly [3]. ViennaCL and MAGMA both provide an iterative ILU, and we include this option in the experimental evaluation although, the *Code_Saturne* reference software does not contain an ILU preconditioner. In the experiments, as we are dealing with symmetric positive definite systems, we use the symmetric variant of ILU, the Incomplete Cholesky (IC).

4 Experimental results

In this section, we analyze the convergence and performance of the Conjugate Gradient method using different preconditioners when solving the real-world CFD benchmarks previously described. The solvers are taken from different software libraries: *Code_Saturne* version 4.0.0 is compared against MAGMA release 2.0.0 and ViennaCL version 1.7.0. The GPU implementations are based on CUDA and CUsparse version 7.5 [10], and use an NVIDIA Tesla K40c GPU. The default block size is 256, which is also the size of the matrix slices in the SELLP format. *Code_Saturne* is using a 6-core Intel Xeon E5-2620 (Ivy Bridge) with hyperthreading enabled. In our experiments, we use 8 or 10 OpenMP threads, whichever provides the best performance.

In Figure 2, we analyze how well the different CG implementations scale with respect to the problem size. As described in Section 2, we replicate the `bundle` problem to generate linear systems of larger dimensions. For a comprehensive evaluation, we use the MAGMA and ViennaCL solvers with different matrix storage formats. The intention is to identify the most suitable format for this problem. The right side shows the runtime of 100 iterations using a Jacobi-preconditioned CG (JCG). In this case, the preconditioner setup time is included as well. Note that CUsparse does not contain a pre-coded JCG implementation. ViennaCL only allows for the use of the CSR format, and does not provide a JCG version featuring kernel fusion in version 1.7.0. This explains the larger difference between the JCG runtime for ViennaCL and MAGMA when using the CSR format. As expected, SELLP again gives the best performance. A Jacobi preconditioner increases the pressure on the memory bandwidth, which is the performance-limiting factor for the GPU implementations. Nevertheless, the MAGMA JCG using SELLP format solves the largest problem about 8 times faster than *Code_Saturne*. For the small problems, the multicore JCG should be preferred. As SELLP gives also the best performance for the `bora` problem, we choose this format for the CG implementations of MAGMA and ViennaCL. On the left side, we show the time needed to execute 100 CG iterations using different combinations of software and matrix formats. For the GPU-based solvers (CUsparse, ViennaCL, MAGMA), the time needed for transferring the matrix and vectors between host and GPU is also included. 100 iterations are typically insufficient for convergence (also for this problem), which emphasizes the impact of these data transfers.



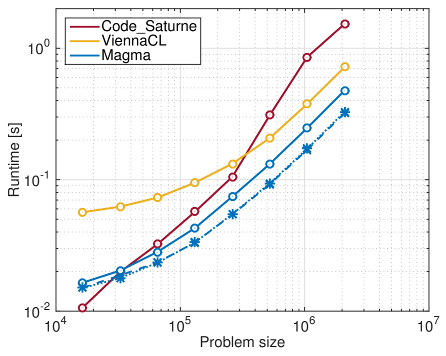


Fig. 2: Solver execution time for 100 iterations of different implementations of CG (left) and JCG (right). The target problem is the replicated `bundle`. Solid lines (with circle marker) are for CSR format , dotted lines (with star marker) for SELLP.

For small problems, the overhead of the data transfers plays an important role. Also, the parallel compute power of the GPU cannot be exploited, as the size of the linear system is smaller than the parallelism provided. For these problems, *Code_Saturne* is much faster than all GPU codes. With increasing problem size, the runtime of *Code_Saturne* grows much faster than for the GPU implementations, and for the largest problem (2 million unknowns), ViennaCL, CUsparse ,and MAGMA run between 5 and 10 times faster than the multicore CG. NVIDIA's CUsparse implementation is highly optimized, and its performance for the CSR format is unmatched by either MAGMA or ViennaCL. At the same time, it does not support the ELL and the SELLP matrix format, which gives much better performance for this class of test matrices. For MAGMA and ViennaCL, using SELLP gives the fastest CG execution time. The higher backend flexibility of ViennaCL comes along with some performance decrease. The MAGMA implementation of CG is optimized in CUDA, and using the SELLP format in this routine is the overall winner for larger problems (15000 unknowns).

In Figure 3, we compare the runtime for the different software libraries and solver settings when solving the `bora` problem. Notice that, in contrast to Figure 2, we do not show the execution time for a fixed number of iterations, but show the timings of the preconditioner setup phase, the data transfer, and the iteration phase when solving the linear system for a relative residual stopping criterion of 10^{-10}. This implies that using a Jacobi preconditioner improves convergence, but makes every CG iteration more expensive. The validity of the results is ensured as the iteration counts are consistent across the different software libraries.

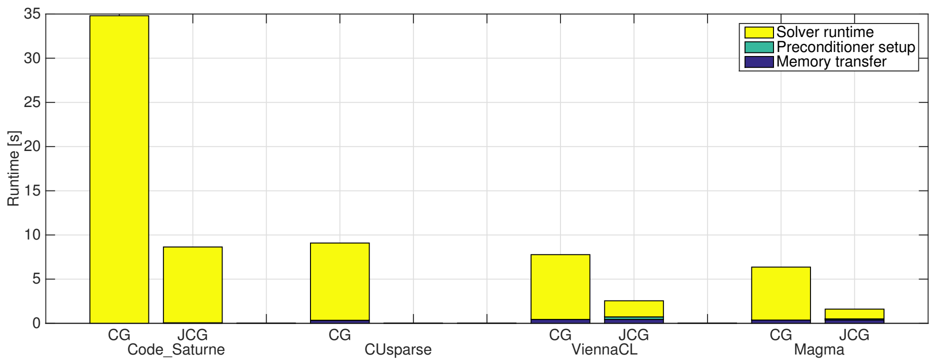


Fig. 3: Execution time of different implementations of CG and JCG for `bora`.

Despite the additional cost of the preconditioner setup, all implementations benefit from using a Jacobi preconditioner. The execution time for *Code_Saturne* improves by a factor of 4. Similarly, the ViennaCL and MAGMA JCG solve `bora` significantly faster than the corresponding CG implementations. The acceleration is smaller for ViennaCL, as the JCG is not enhanced with kernel fusion in the current release. All the GPU implementations are significantly faster than the multicore implementation. The overall winners are the MAGMA implementations for CG and JCG. Compared to the multicore *Code_Saturne* implementation, MAGMA improves the execution times of CG and JCG from 34.81s to 6.36s and from 8.64s to 1.61s, respectively. This includes the expensive preconditioner setup phase. In a scenario where a sequence of similar problems has to be solved, the reuse of a generated preconditioner would provide even larger benefits.

Although not available in *Code_Saturne*, we want to investigate whether the recent advances in iterative ILU preconditioning are suitable for the given real-world problems. ILU preconditioners are well-known to significantly reduce the iteration count for a large range of problems, and replacing the exact sparse triangular solves in the preconditioner application with approximate triangular solves can also make incomplete factorization preconditioners attractive for GPUs [3].

In Figure 4, we compare iteration count (left) and execution time (right) for solving the `bora` problem with different preconditioners. Although also available in ViennaCL, in this experiment we focus on the MAGMA software package, as we exclusively target NVIDIA GPUs, and previously identified the implementations in MAGMA as performance winners for this problem. As previously mentioned, despite the ILU-notation, we internally use an incomplete Cholesky factorization, the symmetric variant of the incomplete LU factorization [12].

The left side shows the iteration count needed for the relative residual stopping criterion of 10^{-10} when using a plain CG, a Jacobi preconditioner (JCG), an exact ILU preconditioner, or the variant using approximate triangular solves. The incomplete factorization preconditioners are generated as level-ILU [12] using different fill-in levels. For simplicity, the factorizations themselves are generated as exact factorizations, despite MAGMA also providing the functionality for iterative ILU-factor generation [6].

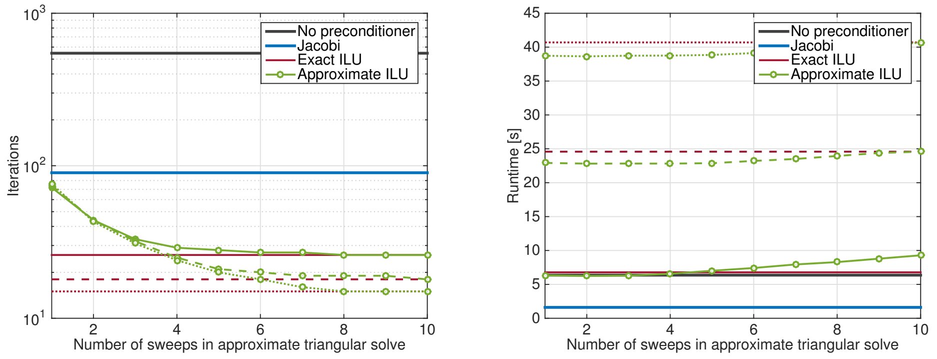


Fig. 4: CG iteration count (left) and execution time (right) for solving `bora` with different preconditioner configurations available in the MAGMA library. For the ILU preconditioner, solid lines are ILU(0), dashed lines are ILU(1), dotted lines are ILU(2). Exact ILU uses exact triangular solves based on level scheduling, approximate ILU uses different numbers of Jacobi sweeps.

Using a Jacobi preconditioner decreases the number of iterations significantly, from 547 to 90. An exact factorization preconditioner provides even larger convergence improvement, reducing the iteration count to 26, 18, and 15, for ILU(0), ILU(1), and ILU(2), respectively. Using approximate triangular solves requires some additional iterations of the outer CG solver, but depending on the fill-in level, 3-6 sweeps in the approximate triangular solves are sufficient to bring the CG iteration count close to the exact ILU.

More relevant than the iteration count is the execution time, as this is the metric of interest when optimizing CFD simulations. The right side of Figure 4 shows the corresponding execution time of the different configurations. Despite the higher iteration count, approximate triangular solves accelerate the ILU-preconditioned CG. Also, the lower iteration count for higher fill-in levels is not reflected in execution time, and despite the significantly lower iteration count (15 vs. 547), the ILU(2) using exact triangular solves needs about 8 times longer than the unpreconditioned CG. This comes partly from the higher cost of the preconditioner setup and data transfers. Also, higher fill-in levels make the sparse triangular solves (exact and approximate) more expensive. For using an incomplete factorization preconditioner, the runtime winner is the setting of an ILU(0) and two Jacobi sweeps in the approximate triangular solves. This configuration needs 3.55s for the preconditioner setup, 0.37s for the data transfers, and 2.38s for the PCG iterations. In the execution time of the iterations, 1.97s are needed for the approximate triangular solves. Due to the expensive preconditioner setup, the overall performance hardly matches the performance of the unpreconditioned CG. Using the iterative ILU generation would improve the results, but real benefits can only be expected when solving a sequence of linear systems that allow for reusing a generated preconditioner. In the end, it is the Jacobi-preconditioned CG that gives the best performance when solving the `bora` problem.

5 Summary and future work

In this position paper, we evaluated whether the available open-source software libraries for sparse linear algebra computations on GPUs are suitable for real-world problems arising from an industrial application. For two CFD simulations, we compared the performance of the solvers from *Code_Saturne*, an in-house developed multicore computational fluid dynamics code from EDF, to that of CUsparse, MAGMA and ViennaCL. The results reveal the superiority of *Code_Saturne* for small problems. For large problems, the GPU codes run up to $5\times$ faster. In the future, we will address sequences of linear systems and evaluate the benefits of reusing a preconditioner for problems with similar properties. We will also address non-symmetric problems, where the benefits of the Jacobi preconditioner are typically smaller, and the iterative ILU preconditioner may become the method of choice.

Acknowledgements

This work was funded by the contract 8610-5910139657 between Université Paris-Sud and EDF. We are grateful to Karl Rupp (TU Wien) for his support in using the ViennaCL library.

Bibliography

[1] J.I. Aliaga, J. Perez, and E.S. Quintana-Orti. Systematic fusion of CUDA kernels for iterative sparse linear system solvers. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 675–686, 2015.

[2] J.I. Aliaga, J. Perez, E.S. Quintana-Orti, and H. Anzt. Reformulated conjugate gradient for the energy-aware solution of linear systems on GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 320–329, Oct 2013.

[3] H. Anzt, E. Chow, and J. Dongarra. Iterative sparse triangular solves for preconditioning. In J. L. Träff, S. Hunold, and F. Versaci, editors, *Euro-Par 2015: Parallel Processing*, volume 9233 of *Lecture Notes in Computer Science*, pages 650–661. Springer Berlin Heidelberg, 2015.

[4] H. Anzt, S. Tomov, and J. Dongarra. Energy efficiency and performance frontiers for sparse computations on gpu supercomputers. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 1–10, New York, NY, USA, 2015. ACM.

[5] F. Archambeau, N. Méchitoua, and M. Sakiz. Code Saturne: A Finite Volume Code for the computation of turbulent incompressible flows - Industrial Applications . *International Journal on Finite Volumes*, 1(1), 2004.

[6] E. Chow, H. Anzt, and J. Dongarra. Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs. In *Lecture Notes in Computer Science*, volume 9137, pages 1–16, July 12 – 16 2015.

[7] E. Chow and A. Patel. Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing*, 37:C169–C193, 2015.

[8] MAGMA Web page. `http://icl.cs.utk.edu/magma/index.html`.

[9] NVIDIA Corporation. CUDA C best practices guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[10] NVIDIA Corporation. *CUDA Toolkit Documentation v7.5*, September 2015.

[11] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*, pages 51–56, 2010.

[12] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.