Acceleration of Turbomachinery Steady Simulations on GPU

Mohamed Hassanine Aissa[1(✉)], Lasse Müller[1], Tom Verstraete[1], and Cornelis Vuik[2]

[1] Von Karman Institute for Fluid Dynamics, Waterloosesteenweg 72, 1640 Sint-Genesius-Rode, Belgium
aissa@vki.ac.be
[2] Delft University of Technology, 2628 CD Delft, The Netherlands
http://www.vki.ac.be

Abstract. Steady state simulations in Computational Fluid Dynamics (CFD), which rely on implicit time integration, are not experiencing great accelerations on GPUs. Moreover, most of the reported acceleration effort concerns solving the linear system of equations while neglecting the acceleration potential of running the entire simulation on the GPU. In this paper, we present the software implementation of an implicit RANS CFD solver, which is fully running on GPU. We use the GMRES linear solver of the Paralution package combined with the incomplete LU factorization for the preconditioning. We propose also a control mechanism - *on-demand* factorization - capable of reducing the number of times an incomplete LU factorization is performed. The *on-demand* factorization accelerates the linear solver without altering the flow convergence. The GPU implementation achieved a speedups of 9.2x compared to a single-core CPU and 3.5x compared to a 4-cores CPU for 3-D flow predictions in turbine applications.

Keywords: Steady CFD · Linear systems · GPU · ILU · Krylov subspace · GMRES

1 Introduction

1.1 Sparse Linear Systems in Turbomachinery

Turbomachinery components are nowadays designed by using optimization algorithms, which scan the design space guided by CFD simulations [1]. These algorithms require therefore a large number of simulations making any time gain on the CFD level very beneficial for the overall optimization procedure. These steady CFD simulations advance an initial flow solution based on an explicit or implicit numerical time integration scheme. Implicit schemes are more stable and faster to converge due to a larger allowed time step. This property comes however at a high cost of assembling and solving a linear system of equations

F. Desprez et al. (Eds.): Euro-Par 2016 Workshops, LNCS 10104, pp. 814–825, 2017.
DOI: 10.1007/978-3-319-58943-5_65

$Ax = b$ at every flow iteration. The system assembly comprises the computation of the system matrix A and the right-hand side b. The linear solver, due to the sparsity character, uses an iterative solver such as the Generalized Minimal Residual Algorithm for Solving Non-symmetric Linear Systems (GMRES) [6].

For CFD problems in turbomachinery, this system of equations is large but sparse. With the growth of the problem size and complexity, the use of High Performance Computing (HPC) becomes inevitable. In this field, Graphics Processing Units (GPUs) are gaining in importance through the reported speedups of many CFD applications [2,4]. While dense matrix vector operations are very efficient on GPUs [5], solving a sparse linear system of equations is more challenging, since there are less independent operations for the large GPU computational power. Moreover, most linear systems require a factorization-based preconditioner to converge, which enhances the serial aspect of the algorithm and thus reduces drastically the GPU performance gain.

1.2 Related Work

In some GPU-accelerated applications [9,20] with a major part of the execution time for the linear solver, the CPU is used for the system assembly, for which the high porting effort is not worth the performance gain. A linear solver is in general implemented on a GPU using a low-level programming language. The flexibility of the low-level approach makes it possible to adapt the data storage and the algorithm to the sparsity pattern (non-zero elements distribution) of the system matrix in order to enhance the performance. In this context, the effort is concentrated on accelerating the sparse matrix-vector product (SpMV), which constitutes the core of many linear solvers. Bell and Garland [20] examined the optimization possibilities for SpMV on GPU without reordering the system matrix. He identified the diagonal format (DIA) as suitable for structured meshes and the Hybrid matrix format (HYB) for unstructured ones. The optimization is part of the CUSP library. Cecka et al. [10] did similar work for problems based on Finite Element Methods (FEM). He examined the effect of the memory optimization on the overall performance comparing local, global and shared memory. Istvan and Giles [11] reviewed relevant research for SpMV on GPU and concentrated on GPU tuning of SpMV operations for the Compressed Sparse Row (CSR) matrix format making use of the L1-cache locality, shared memory, and thread cooperation. The author presented a speedup of 1.4x over cuSparse and suggested that cache hit maximization was the key method behind the observed performance gain.

GPU iterative solver performance has been gradually increasing but the bottleneck remains the serial preconditioners such as the Incomplete LU factorization (ILU). These functions have been the subject of extended research [15]. In order improve the performance, the system matrix has to be reordered. This expose more fine-grained parallelism and thus provide the GPU with more independent instructions. Level-scheduling is one established alternative to elevate the parallelism of the factorization, where independent rows of the system matrix are implicitly grouped in the same level. Graph-coloring is another method

where an explicit reordering is performed giving independent matrix elements the same color, then every color is thread-safe for a massively parallel linear solver. Naumov et al. [12] showed a parallel graph coloring method reaching a higher parallelism than in level scheduling. His work is included in the cuSparse Nvidia library. Another method to extract more parallelism, introduced by Chow and Patel [19], is to transform the ILU factorization in a minimization problem of a set of equations that could be computed in groups independently. Groups can be so small to contain only one equation making it possible for every non-zero element of the incomplete L and U matrices to be computed asynchronously and in parallel. This ILU version can be found in ViennaCL[1].

1.3 Contributions

In this work, the reference CFD simulation is performed on CPU using PETSc [8] and 70% of the execution time is spend on the system assembly, while the rest is for the system linear solver. The same balance is also found in some FEM applications, e.g. Darve et al. [13] ported a CPU application based on PETSc with 80% of execution time for the system assembly. This observation motivated us to port the assembly part to the GPU to avoid any data transfer to the CPU during the simulation. The linear solver is the preconditioned GMRES solver of the Paralution library[2], which uses building blocks of the efficient cuSparse library. This library has been reported [18] to allow a speedup of factor 5x for a neutron diffusion problem. Paralution performs, however, the assembly of the system matrix on the host, which implies a data transfer from GPU to the host CPU. To address this issue we developed an interface to connect the system of equations, which is assembled on the GPU, to the linear solver. We propose an algorithm - *on-demand* LU factorization- to optimize the frequent use of linear solvers in steady simulations. The algorithm is capable of reducing the number of times an ILU preconditioner matrix is built for the linear solver without altering the flow accuracy. This new technique enables the linear solver to use previously computed LU matrix as preconditioner instead of computing a new one in every iteration. We combine this technique with standard ILU to deliver the best speedups for coarse and fine meshes.

Our contributions are:

- A GPU solver based on implicit time stepping with no CPU-GPU data transfer.
- An *on-demand* ILU preconditioner build to reduce the computational time.
- Analysis of the advantages and drawbacks of the GPU for implicit solvers.
- An interface to Paralution and ViennaCL linear solvers.
- A sorting algorithm to transform unordered matrix entries to COO then CSR.

The rest of the paper is structured as follow: Sect. 2 introduces the numerical scheme used by the CFD solver while Sect. 3 describes the implementation of

[1] Rupp, K. "ViennaCL." http://viennacl.sourceforge.net.
[2] PARALUTION Labs "PARALUTION v1.0.0", 2015, http://www.paralution.com.

the solver on the GPU. Results are shown in Sect. 4 and main findings are summarized in Sect. 5.

2 Numerical Scheme

The flow solver uses a cell-centered finite volume discretization on multiblock structured grids. It solves the Reynolds-Averaged Navier Stokes (RANS) equations in time-dependent integral form [16]:

$$\frac{\partial}{\partial t}\int_{\Omega} \boldsymbol{W} d\Omega + \oint_{\partial\Omega} (\boldsymbol{F_c} - \boldsymbol{F_v}) dS = \int_{\Omega} \boldsymbol{Q} d\Omega, \tag{1}$$

with $\boldsymbol{W} = \{\rho, \rho V_x, \rho V_y, \rho V_z, \rho E\}$ the vector of conservative variables, Ω the cell volume and S the cell surface. The convective fluxes $\boldsymbol{F_c}$ are computed using a Roe upwind approximation of a Riemann Solver while second order accuracy is achieved through the MUSCL approach (Monotone Upstream-Centered Schemes for Conservation Law). The viscous fluxes $\boldsymbol{F_v}$ are approximated using a central discretization scheme. The source term $\boldsymbol{Q}$ contains contributions from the Spalart-Allmaras (SA) one-equation turbulence model.

The implicit time integration on steady simulations follows the equation below:

$$\left[\frac{(\Omega I)}{\Delta t} + \left(\frac{\delta \boldsymbol{R}}{\delta \boldsymbol{W}}\right)\right] \Delta \boldsymbol{W}^n = -\boldsymbol{R}^n. \tag{2}$$

with $\boldsymbol{R}$ the residual containing the fluxes and the source term, $\Delta \boldsymbol{W} = \boldsymbol{W}^{n+1} - \boldsymbol{W}^n$ the solution change, I the identity matrix and $\frac{\delta \boldsymbol{R}}{\delta \boldsymbol{W}}$ an approximate *Jacobian* matrix. When Eq. 2 is applied to the entire mesh a large linear system is build with the form $Ax = b$. Residuals and Jacobian are first evaluated on cell surfaces and then summed in a local assembly procedure (see Eq. 1). The global assembly concatenates the local items to a large global matrix and right-hand side containing all the problem unknowns. A multistage time stepping method such as implicit Runge-Kutta [17] solves multiple successive linear systems for every flow iteration, in which only the right-hand side is updated then multiplied by a different stage coefficient α. The nature of the flow solved in this work and the mesh complexity leads to a stiff system matrix that requires further treatment, e.g. preconditioning, to enhance the linear system convergence. A preconditioner is any form of modification to the original linear system, which accelerates the convergence of an iterative method [7]. The linear system of equations is modified as follow:

$$M^{-1}Ax = M^{-1}b, \tag{3}$$

with M the preconditioning matrix. M can be filled by an incomplete factorization of the original system matrix: $A = LU - R$, where L and U are upper and lower matrices respectively while R is the residual of the factorization. The general algorithm of the incomplete LU factorization can be found in [7]. This factorization - involving a Gaussian elimination process - is inherently serial with recursive computations, in which every value of the L and U matrices depends on the computation of several values of previous rows and columns. This dependency makes any parallelization difficult.

3 Flow Solver Implementation

The reference CPU-based implicit solver, written in C++, solves the linear system of equations using the PETSc package. The residual and flux Jacobians are computed serially in a loop over all mesh faces. Profiling has revealed that the ILU preconditioner is not the bottleneck in the CPU implementation taking a small portion of the execution time. Three libraries have been considered for solving the linear system of equations on GPU: PETSc (GPU version), viennaCL and Paralution. While PETSc requires only a small change on the data type of the system matrix and right-hand side to run the linear solver on GPU, the library does not provide a GPU implementation of incomplete LU factorization. Moreover, it does not accept external data computed on GPU, which reduces the scope of the parallelization to the linear solver minimizing the expected global speedup. A second alternative is to use ViennaCL. While this OpenCL-based library can process data residing on the GPU, it performs a costly data copy from CUDA type of data to OpenCL. The third alternative is Paralution, which can process data residing on the GPU and is at the same time based on CUDA cuSparse library. The latter library has been chosen for the linear solver. To describe the flow solver implementation, we first introduce briefly some GPU computing techniques used in this work before we present the two main parts of the GPU flow solver (see Fig. 1) namely the system assembly and the linear solver.

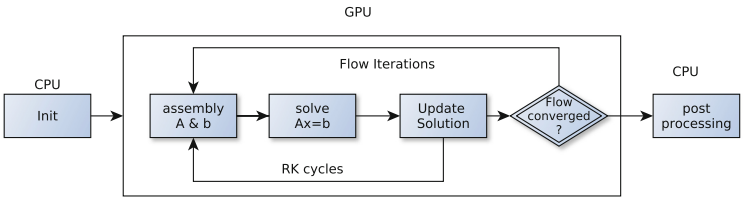

Fig. 1. Flow solver algorithm with an outer loop for the flow iteration: $\boldsymbol{W}^{n+1} = \boldsymbol{W}^{n} + \boldsymbol{\Delta W}$, and an inner loop of Runge-Kutta cycles for the computation of $\boldsymbol{\Delta W}$

3.1 GPU Computing

The GPU is a co-processor featuring a large number of cores organized in streaming multiprocessors, which access directly a global memory. Every multiprocessor is a set of scalar processors with access to a shared memory local to the multiprocessor. Each of these processors has its own local and register memory. Programs running on GPU are called kernels. When calling a kernel the GPU starts a large number of threads (unit of execution) grouped in blocks of threads. Threads among the same block are grouped in warps of 32 threads with consecutive thread ID that execute the same instructions. When threads of the same warp execute different operations, they are executed serially and this performance decreasing situation is called *thread divergence*. The GPU acceleration

is based on overlapping the memory access time (latency) with computations. When a warp is blocked waiting for data the GPU schedules another warp to take over with no overhead for the scheduling. This technique is more effective, if a kernel with a large number of blocks is executed, as more warps are likely to be available for the scheduler. An accurate measure of code performance on GPU is the throughput as floating operations per second which combines arithmetical and memory performance. A first hint to optimize a GPU code then is through increasing the number of active warps, which can run simultaneously (occupancy). At the same time occupancy should not be the only key of performance assessment, as it can be misleading for some cases [3]. In the second place, the algorithm should ensure that neighboring threads, which run together in one warp, access neighbor memory positions in order to avoid long wait times for variables load. This access is called a *coalesced* access.

The number of active warps defining the occupancy is proportional to the number of started threads and the memory consumption per thread in terms of registers and shared-memory. The variables declared in a kernel are locally saved in fast access registers until there are no registers anymore and the rest of the data is stored in global memory. Since all threads share a certain amount of registers the kernel consumption on registers limits the number of blocks of threads that could run simultaneously. In case the kernel needs to start few threads, a technique called multi-streaming can be used to increase the number of active warps by starting multiple independent kernels at the same time. Every kernel contributes to the occupancy by providing active warps. This is different from the standard one-stream approach, in which kernels are executed one after the other. As this section is intended to provide a short overview of techniques used in this work, further details to the GPU architecture and the programming model along with some applications can be found in: [20,24].

3.2 System Assembly

The global system matrix is a concatenation of local block matrices, which are divided in diagonal and off-diagonal blocks. The dominance of the diagonal blocks, which contain the inverse of the time step, improves the convergence of the linear solver. Therefore, when small time steps are used (see Eq. 2) GMRES converges with fast Jacobi preconditioner without the need for factorization. However, large time steps decrease the diagonal dominance and with it the condition number requiring thus the incomplete LU factorization to accelerate the linear solver convergence. The off-diagonal blocks contain mainly the flux Jacobians defining the bandwidth of the matrix.

Within the finite volume scheme, the global assembly of the linear system is made by looping over the cell faces in the mesh. On every cell face a contribution to the cell local system matrix is computed along with a residual. Since every cell receives the contributions of six faces, a risk of *race condition* is eminent, in which up to six threads simultaneously update the system matrix of the same cell. To avoid race conditions *atomic add* or *graph-coloring* are generally used. These techniques are known for deteriorating the coalesced access.

In this work another alternative that conserves the data coalescence has been chosen, in which the contributions are stored along with their positions in the system matrix (row, column). The face contribution belonging to two neighbor-cells is stored twice with the belonging cell index and sign. Computing and storing all face contributions leads to three large arrays: two for indices (column array, row array) and a third array for the contribution's value. Contributions belonging to the same cell are identified over identical index in column and row arrays then summed up using *sort* and *reduce* functions of the `THRUST` library [14]. This library generates 3 arrays free of repetition hosting the positions and values of all non-zero elements (nnz) of the system matrix. This data arrangement is known under Coordinate format (COO). The COO format stores nnz values in double precision and $2 * nnz$ integers. To reduce the storage size while keeping the same information content, the row array can be transformed in row offset array, in which the column offset of the first non-zero element in every row is stored. This operation is performed by the CUSP library, which provides the CSR arrays that constitute the input for the iterative solver of Paralution. A similar but less complicated algorithm allows to sort and scan the right-hand side for duplicated entries. Finally, all Kernels in this work are based on the same global memory access pattern and the coalesced access is assured by using the thread index as an offset for the array index.

3.3 Linear Solver with *on-demand* Factorization

The flow solver has a modular design with an interface to PETSc, Paralution and ViennaCL libraries. We use the GMRES linear solver of Paralution library along with the incomplete LU preconditioner (ILU). To accelerate the linear solver while preserving the accuracy of the solution, the LU matrix should be provided for a lower cost. As reported by many authors [7,19], the accuracy of the Lower Upper matrices affects the conditioning of the system leading to a larger number of linear system iterations to convergence. Since iterations of the linear solver are faster on the GPU than the incomplete LU factorization, the additional inner iterations cost generally less time than performing the incomplete LU factorization. The accuracy of the factorization is here traded against performance.

To decrease the time spent in the factorization, the linear solver uses the LU matrix of previous flow iteration. As a result, the linear solver skips the factorization for some flow iterations. The factorization is performed only *on-demand*, when the LU quality is so decreased that the linear solver needs more iterations to converge than a user defined threshold:

Pseudo-code of the on-demand LU factorization

```
if (itr> MAX_ITR ) M <-LU_Factorization (A)
(x, itr) <- GMRES (A,M,b)
```

where A, M and b are defined in Eq. 3. The maximum number of iterations MAX_ITR depends on the condition number and thus on the time step. As

the time step depends on the CFL and the mesh cell size a relation between MAX_ITR and CFL number can be found for a given mesh:

$$MAX\_ITR = a + b * CFL, \tag{4}$$

with a and b two tuning parameters. Parameter a plays an important role for applications with a low CFL number and b increases with the mesh refinement. The *on-demand* factorization changes only the entries of L and U matrices not the ordering of the non-zero elements, therefore it does not affect the flow solver convergence and accuracy.

4 Results

The numerical results were obtained using a Tesla K40 GPU with a theoretical peak performance of 1,682 Gflops in double precision and 12 GB of global memory. The GPU implementation is realized with CUDA 7.0. The host CPU (double quad-core) is an Intel(R) Xeon(R) CPU E5-2640 with a clock rate of 2.50 GHz and a 15 MB cache size. The CPU parallelization is performed on mesh block level, as blocks are distributed to processors (1 to 4) assuring a good load balancing. For the benchmark case with seven mesh blocks of different sizes, using more than 4 processors deteriorates the load balancing which damage the CPU performance. Therefore a maximum of 4 CPU cores is used.

The test case is a transonic flow over the LS89 inlet guide vane cascade [21], which experiences a turning of 74° through the NGV geometry and a passage shock with a peak Mach number of 1.15. The validation of the flow solver against experimental data can be found in [23]. The stopping criterion for the linear solver is a 10^{-6} reduction of the relative Residual and the flow solver stops when the minimization of the L_2 norm of the residual reaches 10^{-6}. The 2-stages Runge-Kutta (RK) time stepping method has been chosen for the benchmark, since the RK methods with more stages presented no flow convergence acceleration in the treated CFD case while costing extra execution-time. Two types of meshes are treated (coarse and fine) to explore the GPU potential (see Table 1).

Table 1. Characteristics of used meshes and underlying linear systems

Mesh	N_{Cells}	N_{Rows}	nnz	nnz/row
Coarse	40k	200k	5.7M	[20 . . . 30]
Fine	300k	1500k	52.6M	[20 . . . 35]

4.1 Assembly Acceleration

The assembly phase on the GPU experiences a 7x acceleration for the coarse mesh compared to a single-core CPU and 12x acceleration for the fine one (see Figs. 2 and 3). The multi-streaming contributed to the speedup by 10% improve

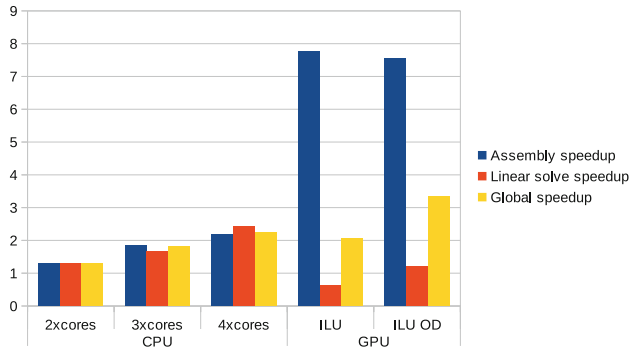

Fig. 2. Speedups of the flow solver on the coarse mesh with GPU ILU and *on-demand* ILU compared to a single-core to 4-cores CPU

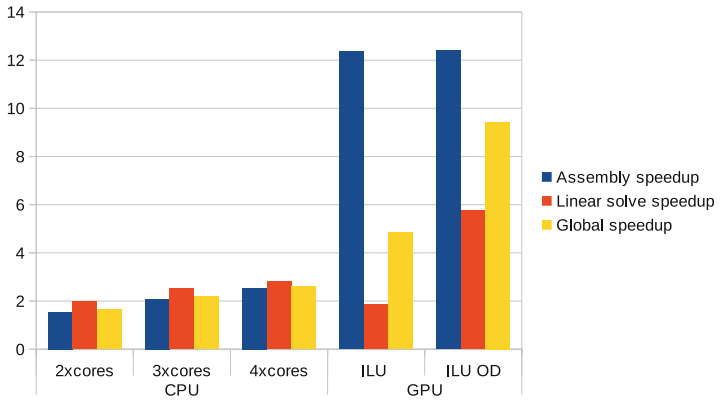

Fig. 3. Speedups of the flow solver on the fine mesh with GPU ILU and *on-demand* ILU compared to a single-core to 4-cores CPU

of the performance compared to the one-stream GPU version for the coarse mesh. The coalesced memory access has more impact on the performance with an improve of 23% compared to a striped access for the same coarse mesh. A multiblock mesh layout originates, in general, from the mesh generator designed to improve the mesh quality towards accurate CFD results. For complicated geometries it leads to multi-block meshes presenting blocks of different sizes and many interfaces between the mesh blocks.

An analysis of the achieved acceleration is proposed by addressing possibilities for further improvements considering: first large, then small mesh blocks and finally the interface update between all kind of blocks. Large blocks provide the GPU kernels with a high amount of independent operations for processing at the same time, which maximizes the number of active threads. The limiting factor in this case is the register usage. Since the kernels are starting large number of threads and computing long algorithms, the total number of used registers is very

high. The register consumption limits the achieved occupancy (see Sect. 3.1). A way to improve the occupancy for these kernels is to divide them when possible into small, less memory demanding, sub-kernels. Blocks with few cells on the other hand, are in fact not limited by register usage but by the small number of started threads. The GPU is not provided with enough active threads to hide the memory latency. In this case, the multi-streams technique (see Sect. 3.1) can improve the occupancy by starting more than one kernel at the same time. The mesh block interfaces require a cell update between blocks and this procedure involves few cells proportional to the *surface to volume* ratio ($r_{StoV} * N_{Cells}$). A solution is to use a mesh generator that takes into account the reduction of the number of blocks and neighboring blocks along with the increase of block size in terms of cells (e.g. `hMETIS` [22]). The higher speedup of the assembly phase on the GPU for the fine mesh is then due to the larger blocks and lower *surface to volume* ratio.

4.2 Linear Solver Acceleration

The linear solver on the coarse mesh is 40% slower than on the single-core CPU. This is mainly caused by: (1) the ILU preconditioner, (2) the total number of linear solver calls and (3) the size of the system matrix. The ILU preconditioner contains low fine-grained parallelism and is more efficient on CPU. Moreover, the CPU implementation of ILU factorization has a set of techniques to improve the linear solver convergence, which decreases dramatically the GPU performance once ported to the GPU. This results on the flow solver using GPU ILU to perform 32% more linear solver iterations. The flow convergence is on the other hand exactly the same for CPU and GPU implementation in terms of number of flow iterations, this for the sake of a fair comparison.

For 2-stages RK, the flow requires 827 flow iterations to converge. The standard ILU performs one factorization per flow iteration, while the *on-demand* ILU (ILU-OD) reduces the total number of factorization to only 113. This corresponds to a decrease of 86%, which explains the improved speedup for the linear solver when ILU-OD is used. The *on-demand* ILU is only as fast as a 2-cores CPU, because the size of the system matrix is not enough to observe the advantage of GPUs for sparse matrix-vector products (SpMV). The fine mesh presents, on the other hand, a larger system matrix with more non-zero elements. While the standard ILU implementation is 1.8x faster than the single-core ILU, the ILU-OD is 5.5x faster than single-core ILU and 2.05x faster than a 4-cores CPU. The *on-demand* mechanism decreased here also the number of factorization by 86%. In addition to that the size of the matrix showed the advantage of GPU for SpMV.

4.3 Global Acceleration of the Flow Solver

The global speedup depends heavily on the mesh size. For the fine mesh the GPU performance reaches a speedup of 4.8 and 9.43 for the flow solver using ILU and ILU-OD respectively compared to a single-core CPU. ILU and ILU-OD

are 1.8x and 3.4x faster than 4-cores CPU. On the coarse mesh, the acceleration is 2.07x and 3.35x for ILU and ILU-OD respectively compared to a single-core CPU. This correspond to a speedup of 1.15x and 1.5x compared to a 4-cores CPU. The larger contribution of the speedup is done in the assembly phase.

GPUs are rather adapted for system assembly as a stencil-based operation and for solving very large sparse linear systems not exceeding the storage capacity of GPUs. Small linear systems are solved more efficiently on cache-based machines. Moreover GPUs are inherently co-processor and cannot replace a CPU for the entire simulation including pre- and post-processing. Therefore, the cooperation between the two architectures is more of interest rather than the competition for speedups as the latter can be misleading [25].

5 Conclusion

In this paper, we presented a flow solver with one order of magnitude acceleration on GPU compared to an optimized serial CPU version. We demonstrated that implicit time stepping in CFD applications can profit from the GPU computational power, provided an appropriate GPU occupancy is reached and a good mesh in terms of surface to volume ratio is used. As the bottleneck of the GPU flow solver is the incomplete LU factorization, the *on-demand* ILU factorization presented in this work improved the overall speedup by 60% to 80%. The *on-demand* ILU can be applied as well on cache-based processors (x86) but it is expected to have a very limited effect since factorization is not a bottleneck for serial execution. On the other hand it is expected to improve the performance of other SIMD machines (e.g. Xeon Phi). This once again shows that acceleration techniques can be very different on various architectures.

Acknowledgments. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013), Marie Curie Initial Training Networks (ITN) action, under grant agreement no. 316394, AMEDEO. We are also grateful to NVIDIA for the hardware donation.

References

1. Shahpar, S., Caloni, S.: Aerodynamic optimization of high-pressure turbines for lean-burn combustion system. J. Eng. Gas Turbines Power **135**(5), 055001 (2013)
2. Brandvik, T., Pullan, G.: Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. Proc. Inst. Mech. Eng. Part C: J. Mech. Eng. Sci. **221**(12), 1745–1748 (2007)
3. Volkov, V.: Better performance at lower occupancy. In: Proceedings of the GPU Technology Conference, GTC, vol. 10 (2010)
4. Lin, F., et al.: A multi-block viscous flow solver based on GPU parallel methodology. Comput. Fluids **95**, 19–39 (2014)
5. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 739–748. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85451-7_79

6. Saad, Y., Schultz, M.H.: GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **7**(3), 856–869 (1986)
7. Saad, Y.: Iterative Methods for Sparse Linear Systems. Siam, New Delhi (2003)
8. Balay, S., et al.: PETSc Users Manual Revision 3.5. No. ANL-95/11 Rev. 3.5. Argonne National Laboratory (ANL) (2014)
9. Serban, G., et al.: GPU acceleration for FEM-based structural analysis. Arch. Comput. Methods Eng. **20**(2), 111–121 (2013)
10. Cecka, C., et al.: Assembly of finite element methods on graphics processors. Int. J. Numer. Methods Eng. **85**(5), 640–669 (2011)
11. Istvan, R., Giles, M.: Efficient sparse matrix-vector multiplication on cache-based GPUs. In: Innovative Parallel Computing (InPar). IEEE (2012)
12. Naumov, M., et al.: Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU. NVIDIA TR NVR-2015-001, May 2015
13. Wong, J., Kuhl, E., Darve, E.: A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems. Int. J. Numer. Methods Eng. **102**(12), 1784–1814 (2015)
14. Bell, N., Hoberock, J.: Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems: Jade Edition (2012)
15. Saad, Y.V., der Vorst, H.A.: Iterative solution of linear systems in the 20th century. J. Comput. Appl. Math. **123**, 1–33 (2000)
16. Blazek, J.: Computational Fluid Dynamics: Principles and Applications. Elsevier, Amsterdam (2005)
17. Xu, S., et al.: Stabilisation of discrete steady adjoint solvers. J. Comput. Phys. **299**, 175–195 (2015)
18. Trost, N., et al.: Accelerating COBAYA3 on multi-core CPU and GPU systems using PARALUTION. Ann. Nucl. Energy **82**, 252–259 (2014)
19. Chow, E., Patel, A.: Fine-grained parallel incomplete LU factorization. SIAM J. Sci. Comput. **37**(2), C169–C193 (2015)
20. 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. ACM (2009)
21. Arts, T., et al.: Aero-thermal Investigation of a highly loaded transonic linear Turbine Guide Vane Cascade von Karman Institute for Fluid Dynamics TN-174 (1990)
22. Karypis, G., Kumar, V.: hMETIS 1.5: a hypergraph partitioning package. Technical report, Department of Computer Science, University of Minnesota (1998)
23. Aissa, M.H., Verstraete, T., Vuik, C.: Aerodynamic optimization of supersonic compressor cascade using differential evolution on GPU. In: 13th International Conference of Numerical Analysis and Applied Mathematics (ICNAAM 2015) September 23–29 2015, Rhodes, Greece (2015)
24. Garland, M., et al.: Parallel computing experiences with CUDA. IEEE Micro **4**, 13–27 (2008)
25. Lee, V.W., et al.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: ACM SIGARCH Computer Architecture News, vol. 38, no. 3. ACM (2010)