Computers and Mathematics with Applications 74 (2017) 201–217

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes

Mohamed Aissa [a,*], Tom Verstraete [a], Cornelis Vuik [b]

[a] Von Karman Institute for Fluid Dynamics, Sint-Genesius-Rode, 1640, Belgium

[b] Delft University of Technology, 2628 CD Delft, The Netherlands

ARTICLE INFO

Article history:
Available online 17 March 2017

Keywords:
Time integration
GPU
CFD
GMRES
Preconditioning
ILU

ABSTRACT

A computational Fluid Dynamics (CFD) code for steady simulations solves a set of non-linear partial differential equations using an iterative time stepping process, which could follow an explicit or an implicit scheme. On the CPU, the difference between both time stepping methods with respect to stability and performance has been well covered in the literature. However, it has not been extended to consider modern high-performance computing systems such as Graphics Processing Units (GPU). In this work, we first present an implementation of the two time-stepping methods on the GPU, highlighting the different challenges on the programming approach. Then we introduce a classification of basic CFD operations, found on the degree of parallelism they expose, and study the potential of GPU acceleration for every class. The classification provides local speedups of basic operations, which are finally used to compare the performance of both methods on the GPU. The target of this work is to enable an informed-decision on the most efficient combination of hardware and method when facing a new application. Our findings prove, that the choice between explicit and implicit time integration relies mainly on the convergence of explicit solvers and the efficiency of preconditioners on the GPU.

1. Introduction

CFD is commonplace in engineering activities, as many products are nowadays designed by relying heavily on numerical preconditions with reduced wind tunnel testing in order to cut down the product development cost. Computations are, nevertheless, still expensive and a trade-off is continuously sought between fast turn around time and high accuracy. New hardware with high computational power, such as the GPU, can be effectively employed to target fast computations without compromising the accuracy. The GPU, originally developed from graphics pipelines, excels on processing a large amount of independent data with a very regular and simple memory access pattern.

The basic task of CFD solvers is to advance iteratively an initial solution by performing a space and a time integration of the governing equations. In order to update the solution, the solver requires memory access to neighboring cells. For explicit time integration, the update depends only on few neighbor cells, while for implicit time integration, the update depends on all cells and could be formulated as a solution of a linear system of equations.

* Corresponding author.
E-mail address: aissa@vki.ac.be (M. Aissa).

http://dx.doi.org/10.1016/j.camwa.2017.03.003
0898-1221/

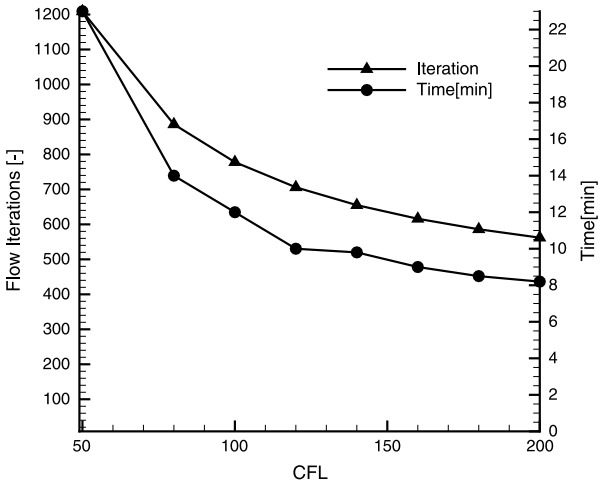


Fig. 1. Saturation of the iterations number with the increase of CFL number (case: turbine t106c [8]).

The locality of explicit solvers reflects on the memory usage, which is very low compared to the implicit solver memory footprint (apart from special cases of matrix-free implicit solvers [1]). The computations performed within this scheme have a stencil-based character, for which neighbor cells are used to update a central cell following a regular pattern. As the operations are repeated for all mesh cells, it generates a large number of independent operations. Due to this simple regular workflow, explicit solvers are suited to the GPU massive parallel architecture and can benefit largely from its computational power. Interesting speedups have been reported in the literature of 1 and 2 orders of magnitudes [2–6]. In general, explicit solvers are stable only when a small time step is used, as the latter is controlled by a relatively low Courant–Friedrichs–Lewy (CFL) conditions (e.g. CFL $= 0.92$ [7]). This method requires, thus, a large number of iterations to converge. When combining the GPU acceleration with convergence acceleration techniques, such as residual smoothing or multigrid and local time stepping, the explicit method can be very efficient.

Implicit solvers have less stringent stability limits, allowing to speed up the transient process with an increased CFL number. The reduction reaches, however, asymptotically a limit, due to the inherent non-linearity of the equations, as depicted in Fig. 1. Implicit solvers benefit less from the GPU acceleration especially when factorization based preconditioners that rely on Gaussian elimination are used such as the incomplete LU factorization (ILU). New algorithms are, however, trying to expose more parallelism and improve the performance of linear solvers on the GPU by accelerating the incomplete factorization [9] and its use in the linear solver [10,11]. Reported speedups for implicit solver are of 1 order of magnitude [12–14].

In this paper, we examine the choice between explicit and implicit time stepping both on CPU and GPU, which results in four different approaches. Few authors compared explicit to implicit performance. Niemeyer [15] implemented a Finite Volume flow solver for chemical reactions on a GPU with explicit time stepping. He compared the performance on a CPU and a GPU of the explicit time integration to a commercial implicit solver on a CPU. He showed, however, no results on the GPU version of the implicit solver. Brock [3] developed an explicit solver for an astrophysics application and compared the execution time of one iteration with estimations of the implicit performance on CPU as the memory requirement for the solved case where prohibitive for the implicit integration. He showed a peak speedup of $2.5\times$ for the GPU explicit over the CPU explicit and expected a speedup of $15\times$ for the GPU explicit over the CPU implicit. He has not considered the possible faster convergence of the implicit method.

To bring the comparison forward we introduce a classification of basic CFD operations following their suitability to the GPU architecture and study their speedups. The classification of different CFD operations is inspired by the major differences of both hardware. The CPU is a general purpose processor able to handle different type of tasks. The large cache for a reduced number of CPU cores (e.g. Xeon E5-2640 has 15 MB of cache memory for 8 cores) is an efficient cure to the irregular data flow of some algorithms. In fact, it minimizes the cache misses defined as calls to variables relocated from the cache memory due to overuse [16]. The processor high clock rate of the CPU is an indicator for the speed, at which computations are executed and the memory is accessed. The GPU, on the other hand, is very specialized with a large computational power coming from the high number of GPU cores. The clock rate and cache capacities are in general lower than what CPUs offer. Time-consuming cache misses cannot be avoided for a dispersed memory accesses. Under these circumstances, a regular data flow pattern is essential in order to achieve good performance. As the computation power exceeds by far the GPU memory bandwidth, algorithms need to balance the slow memory access with a large number of computations. The large number of cores with a reduced power for each core are efficiently used, when a large number of independent data is available with a relatively simple and regular calculation for every piece of data. These specificities of the GPU motivated us to classify the CFD operations regarding 3 criteria: (1) the amount of data to process, (2) the data dependency level and (3) the regularity of the access pattern. The amount of data is used to maximize the possibilities for the GPU to hide slow memory access with computations. A low data dependency provides a large number of independent instructions ideal for the large number of

GPU cores. For a regular access pattern, the GPU can combine multiple expensive memory loads into one single load. The only condition is that consecutive threads of half a warp access consecutive memory positions. The entire explicit/implicit comparison is based on the proportion in execution time of each class in every scheme, which is used to estimate the GPU speedup. A similar classification has been introduced by Elsen [5] for vehicle aerodynamics. He classified kernels based only on memory access pattern.

The CPU and GPU solvers are used in this work to accumulate enough benchmark data of basic operations to correlate it with the global performance of CFD simulations on the GPU. Performance estimation for the GPU is an active domain. Some authors analyze the GPU code [17,18] to build an estimate of the GPU performance. Li [19] predicts the performance of Sparse Matrix-Vector operations (SpMV) using a trained probabilistic model. Baghsorhi [18] interprets a GPU kernel as a workflow graph and estimates its execution time. We cover steady CFD simulation on turbomachinery with structured meshes. The knowledge over the underlying CFD operations enabled us to match the performance bottlenecks on GPU with their sources from the basic CFD operations. We avoid in that way the use of automated methods to analyze program's code or executable.

The classification is used also to empower readers for a better appreciation of reported speedups in the literature. As it is discussed by Lee et al. [20], reported GPU speedups are not to be taken as raw numbers. The reference CPU code optimization is important along with the used GPU. At the same time, some reported accelerations are only local and specific to a certain operation with sometimes little influence on the global speedup. Our approach is rather qualitative focusing on transmitting key know-how on CFD operations on the GPU. The aim is to help the developer to estimate the expected speedup of steady CFD simulations on GPU using only the profiling results of the CFD application on a single CPU.

The remainder of the paper is organized as follows. Section 2 introduces the GPU device and governing equations as a prerequisite for the classification and the qualitative performance model in Section 3. Section 4 presents the 2 GPU-accelerated RANS solvers, which are used for the benchmark. Numerical results are presented in Section 5 for the explicit/implicit comparison and a discussion is in Section 6.

2. Background

2.1. GPU computation

The GPU is a massively parallel device with a large number of cores organized in multi-streaming processors with access to a relatively large but slow global memory. Every multiprocessor hosts a set of processors and provides a small and fast shared memory along with a reduced set of fast registers. A GPU program is based on kernels, which launch a large number of lightweight threads grouped in blocks. Inside a block, threads are grouped in warps, for which CUDA uses the SIMT model (single instruction multiple threads). The same instruction is broadcasted to all threads of a warp for execution. Unlike the SIMD model (Single Instruction Multiple Data), threads in a warp are able to execute different instructions since every thread is provided with an individual register set and instruction counter. Although *thread divergence* is tolerated it is very damaging to the performance and a fully diverged warp can run 32 times slower than a divergence-free warp.

The GPU memory hierarchy proposes different levels of speed and capacity. The data residing in the global memory has higher latency for the access than the data residing in a cache or a register. Moreover, the access itself to the global memory should follow certain patterns to guarantee the best memory bandwidth. The GPU loads, indeed, an entire *word*[1] when a thread accesses a memory position and the loaded *word* is broadcasted to all threads of the warp. Memory transactions of threads within a warp accessing the same *word* are then fused or *coalesced* into one transaction.

The overuse of registers by a kernel limits its theoretical occupancy, defined as the ratio between concurrent warps versus maximum warps per multiprocessor.[2] The more demanding a kernel the lower is its theoretical occupancy. The achieved occupancy, on the other hand, depends on the number of started threads. Therefore, a kernel should start a large number of warps so the warp scheduler can dispatch an idling warp (e.g. waiting for a memory transaction) and switch to an eligible warp ready for execution. This context switch is used to hide the latency of operations on the GPU. Volkov [21] proposes an extensive analysis and a new performance model for latency hiding in the GPU.

While an efficient use of the available memories is the principle source to leverage the performance of a program, the GPU offers also a coarse-grained parallelism. It allows indeed multiple kernels with grids of threads to run concurrently, which increases the number of active warps and consequently the GPU utilization. Multi-streaming is, however, limited by the hardware resources of the GPU.

An extensive and detailed treatment of the subject of GPU programming can be found in the CUDA user manual[3] and some valuable textbooks [22–24].

1 A word is a piece of data with a fixed-size managed as a unit by processor.

2 https://docs.nvidia.com/cuda/cuda-c-programming-guide.

3 https://docs.nvidia.com/cuda/cuda-c-programming-guide, retrieved February 2017.

2.2. CFD RANS solver with finite volume discretization on GPU

The integral form of the Reynolds Averaged Navier–Stokes equations (RANS) in terms of conservative variables is listed below [25]:

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{W}d\Omega+\oint_{\partial\Omega}(\vec{F}_c-\vec{F}_v)dS=\int_{\Omega}\vec{Q}d\Omega, \tag{1}$$

where Ω is the cell volume, $\delta\Omega$ is the cell surface and $\vec{W}$ is the conservative variable vector : $\vec{W}=[\rho,\rho u,\rho v,\rho w,\rho E]$. Convective and viscous fluxes along with the source term are defined as follows:

$$\vec{F}_c=\begin{bmatrix}\rho V\\ \rho uV+n_xp\\ \rho vV+n_yp\\ \rho wV+n_zp\\ \rho HV\end{bmatrix} \tag{2}$$

$$\vec{F}_v=\begin{bmatrix}0\\ n_x\tau_{xx}+n_y\tau_{xy}+n_z\tau_{xz}\\ n_x\tau_{yx}+n_y\tau_{yy}+n_z\tau_{yz}\\ n_x\tau_{zx}+n_y\tau_{zy}+n_z\tau_{zz}\\ n_x\Theta_x+n_y\Theta_y+n_z\Theta_z\end{bmatrix} \tag{3}$$

$$\vec{Q}=\begin{bmatrix}0\\ \rho f_{e,x}\\ \rho f_{e,y}\\ \rho f_{e,z}\\ \rho\vec{f}_e\cdot\vec{v}+\dot{q}_h\end{bmatrix}. \tag{4}$$

The Roe scheme [26] is used to compute the convective fluxes, which are evaluated at the face of a cell as follows:

$$(\vec{F}_c)_{I+1/2}=\frac{1}{2}[\vec{F}_c(\vec{W}_R)+\vec{F}_c(\vec{W}_L)-|\bar{A}_{Roe}|_{I+1/2}(\vec{W}_R-\vec{W}_L)], \tag{5}$$

with $\bar{A}_{Roe}$ the *Roe matrix* defined as a combination of the conservative variables $\vec{W}_R$ and $\vec{W}_L$ [26]. The subscripts L and R refer to the left and right state of the considered face. $\vec{F}_c(\vec{W}_R)$ and $\vec{F}_c(\vec{W}_L)$ are evaluated following Eq. (2). The evaluation of the right and left conservative variables determines the degree of the space integration. Second-order accuracy is achieved in this work through the Monotone Upstream-Centered Schemes for Conservation Law (MUSCL [27]) with the *Venkatakrishnan* slope limiter function [28], which enlarges the stencil from 2 to 4 cells in every space dimension. The central scheme is used for the viscous flux and turbulence is modeled with Spalart–Allmaras model [29].

The "method of lines" is used and thus space and time integration are treated separately. After the space integration, which consists of summing the fluxes and source term in a residual R, a time integration advances the flow toward a stationary state. Eq. (1) can be reformulated to highlight the time integration as follows:

$$\frac{(\Omega\bar{I})_I}{\Delta t_I}\Delta\vec{W}_I^n=-\beta\vec{R}_I^{(n+1)}-(1-\beta)\vec{R}_I^n, \tag{6}$$

with $\Delta\vec{W}=\vec{W}^{n+1}-\vec{W}^n$ the solution update, which depends on a combination of residuals of time point n and $n+1$. While the residual R^n at time point n is available right after the space integration, the residual R^{n+1} depends on the solution at time point $n+1$, which is not available before the time integration. For $\beta=0$, the right-hand side (RHS) of Eq. (6) is known and updates can be computed explicitly. If however $\beta\neq0$, the residual is linearized to allow to formulate R^{n+1} as a function of R^n and the Jacobian $\frac{\delta\vec{R}}{\delta\vec{W}}$ to first order accuracy:

$$\vec{R}_I^{n+1}\approx\vec{R}_I^n+\left(\frac{\delta\vec{R}}{\delta\vec{W}}\right)_I\Delta\vec{W}^n. \tag{7}$$

Substituting the linearization into the initial equation gives a linear system of equations of the form $Ax=b$:

$$\left[\frac{(\Omega\bar{I})_I}{\Delta t_I}+\left(\frac{\delta\vec{R}}{\delta\vec{W}}\right)\right]\Delta\vec{W}^n=\vec{R}_I^n. \tag{8}$$

The implicit time integration requires the same residual, as for the explicit solver, and additionally the Jacobians, which are combined to form the global system matrix (depicted in Fig. 2). Therefore, the building of the linear system costs extra

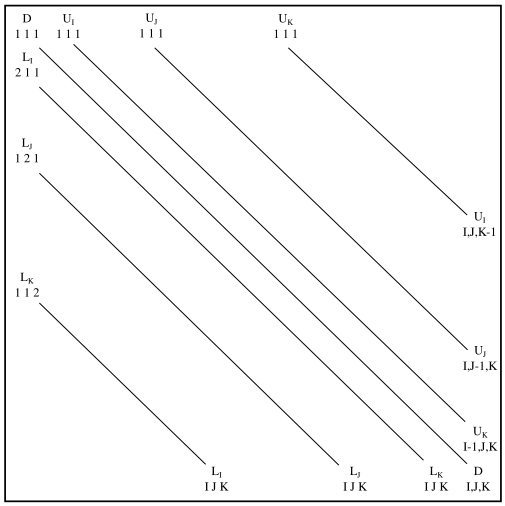


Fig. 2. Structure of the system matrix showing 7 diagonals filled with block matrix of $N_v * N_v$ with N_v the number of flow variables. I, J and K are the number of cells in the first, second and third directions respectively.

memory and computation, due to the large number of entries to be first calculated then inserted into the global matrix. Every cell face has 7 contributions for 3D problems: one Diagonal, one upper and one lower for every direction. The number of elements per row is not constant and varies depending on the number of diagonals crossing the row. Jacobians of cell faces in the interior of the computational domain have neighbors in all directions (corresponding row has 7 diagonals for 3D problems). On the other hand, a Jacobian of a face on the block boundary has fewer neighbors depending on how many boundaries are attached to this face. The number of elements per row plays a role on the performance of sparse matrix-vector operations on the GPU. A large spectrum of storage formats exists for the system matrix. While they all store the values in the same way, the columns and rows indices are handled differently. The compressed row format (CSR) for instance stores the offset of every row along with the column indices.

In the system matrix of Eq. (8), the diagonal has the Jacobian $\frac{\delta \vec{R}}{\delta W}$ added to a term inversely proportional to the time step. This implies that an increase of the time step, which depends on the CFL number, the mesh refinement and the flow properties [25], increases the condition number ($Cond = \|A\| * \|A^{-1}\|$) and makes the system matrix harder to invert. Solving such a linear system requires, in general, more involved computations. The condition number has to be lowered to accelerate the convergence. This is performed by the preconditioner, a matrix $M \approx A$ approximating the original system matrix while being easily inverted.

$$M^{-1}Ax = M^{-1}b. \tag{9}$$

The matrix M can be filled by an incomplete factorization, which is a process able to reformulate the system matrix into:

$$A = CD - E, \tag{10}$$

with C and D two matrices easy to invert and E a residual matrix. The ILU factorization, for instance, produces two triangular matrices L and U and can be used as a preconditioner. Some computationally lighter preconditioners, such as Jacobi, approximates the inverse of the system matrix. These preconditioners expose better parallelism for the GPU, since they are based on SpMV [30], but are not suitable for matrices with a large condition number.

3. Methodology: the classification

In computer science, problems easily run in parallel, are called *embarrassingly parallel*. For those type of problems, it is clear how to divide the algorithm into small independent pieces of calculations, which are performed on different data. The algorithm shows a low level of data dependency peculiar to this type of operations. An *inherently sequential* problem, on the other hand, has highly interdependent operations, which have to be executed in a given order to get accurate results. A practical approach to classify CFD operations as *embarrassingly parallel* or *inherently sequential* is to follow the GPU utilization of the operation given by the GPU profiler. The NVidia Visual Profiler (NVVP) can deliver information about the utilization of the memory and the computation units of the GPU for every kernel. If both unit utilizations are high the kernel is then very efficiently written and executed and the underlying algorithm is suited to the GPU architecture. Fig. 3 depicts this ideal case in addition to 3 realistic cases of performance limitations for GPU kernels: a latency-limited, a compute-limited and a memory-limited kernel. Kernels with low memory and low computation utilizations are latency-limited. Such a low utilization can be caused by a non efficient code (memory non coalesced, important thread divergence) but for carefully

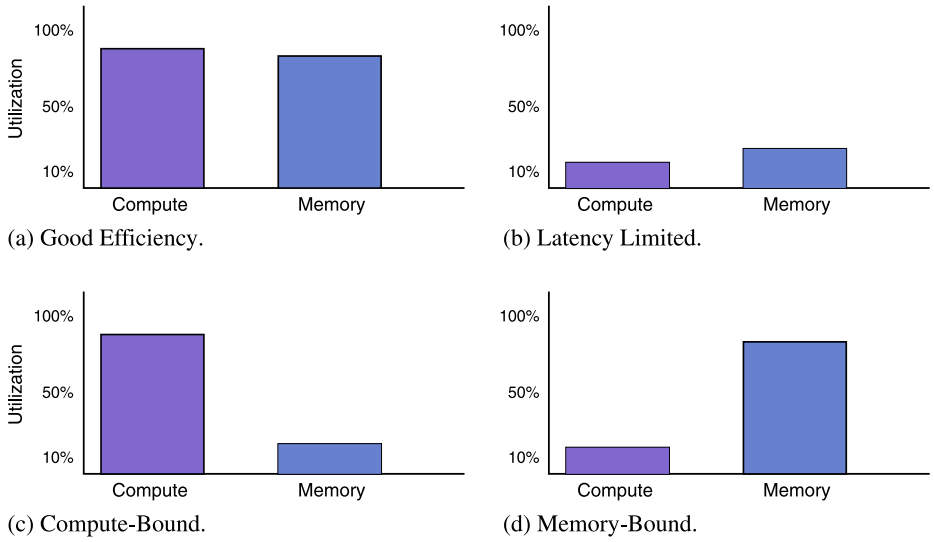


Fig. 3. Four cases of utilization of the memory and compute unit of a GPU delivered by the NVida Visual Profiler.

designed code the major cause is the use of a low number of threads. Indeed, when a kernel starts only few threads, these are unable to hide the memory latency. When warps of threads are waiting for memory loads, not enough warps are available to use the waiting time for computations. Kernels with a high compute utilization and a poor memory utilization are said to be compute-bound. For such a kernel the performance optimization should first target the reduction of the number of computations per memory load. Kernels with a high memory utilization and a low compute utilization are said to be bandwidth-bound. For such a kernel few computations are performed per memory load. In that case, improving the memory coalescence and using the shared-memory could improve the performance of the kernel.

3.1. Classification of basic operations

In this section, elementary functions used by explicit or implicit CFD solvers are first classified into three classes: Compute-bound, Memory-bound and Latency-bound kernels. The classification reflects the differentiation between *embarrassingly parallel* and *inherently sequential* algorithms using the GPU utilization as a measure of the GPU suitability of studied CFD operations. Every class is analyzed regarding solely its performance on the GPU.

Compute-Bound Embarrassingly Parallel operations are suited to the GPU architecture and make an intensive use of arithmetic operations boosting the compute utilization of the GPU (see Fig. 3). In this class, we list explicit Runge–Kutta time stepping, convective and viscous flux calculation, turbulence calculation and Jacobian calculation. These operations are stencil-based, they involve few neighbor cells for the computations related to a central cell. This class of functions is able to provide a large amount of independent data, which increases with the mesh size. The data dependency is relatively low, as only a first-order scheme is used for the Jacobian (requiring access to the 6 neighbor cells) and a second-order scheme for the space integration (requiring access to 8 neighbors). Most of these functions involve an intensive use of arithmetic for a reduced stencil leading to a set of compute bound functions. As the access pattern is regular on all data it is very rare to have a thread divergence. Even though some algorithms can still impose divergence through a conditional statement. The Roe scheme, for instance, performs an entropy correction to better capture flow shocks as the original formulation does not recognize the sonic point [31]. This could lead to different execution paths within a warp. These kernels can have a high performance in terms of updated cells per second, which leads to a relatively high speedup of 2 orders of magnitude (see Figs. 4 and 5).

The convective flux calculation is a compute-bound kernel, as the roe scheme (see Eq. (5)) has to be evaluated at the cell face involving a lengthily computation with a large amount of arithmetic operations. The flux calculation is based on a summation of surface contributions and thus not thread-safe as two or more faces could add face contributions at the same time to the same cell. One possibility to avoid this race condition is to compute the flux cell-wise which is thread-safe but requires a redundant calculation of the flux. Recalling that the kernel is compute-bound, redundant computation has to be avoided. The second approach is the multi-coloring, which consists of creating groups (colors) of faces, which do not share cells in common. For every color, the computation is thread-safe at the expense of less coalesced access. The second approach is proven to be more efficient (see Fig. 4).

Memory-Bound Embarrassingly Parallel operations provide welldefined independent work units for the GPU but make few computations per loaded byte of memory. This class includes sparse matrix-vector operations (SpMV), as a dot product has to be performed for every row of the matrix. In CFD time integration, the rows correspond to mesh cells, even though they can be reordered to favor a better performance. SpMV operations are known for being memory bound [32] and benefit

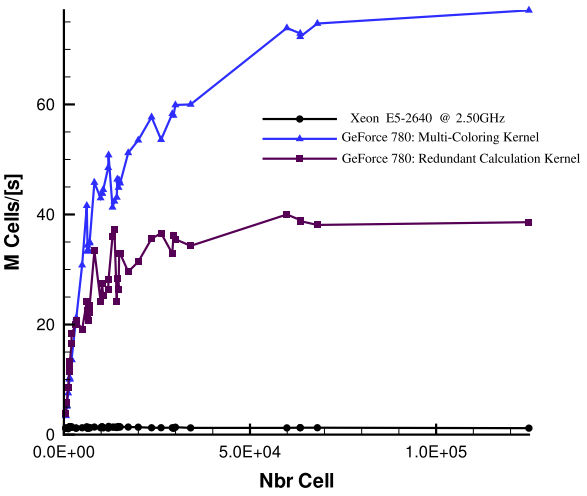


Fig. 4. Performance in terms of updated cells per second for one inviscid flux calculation on a single CPU compared to multi-coloring based kernel and redundant calculation on GeForce 780 for a RANS simulation of 2D flow on a supersonic compressor cascade. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

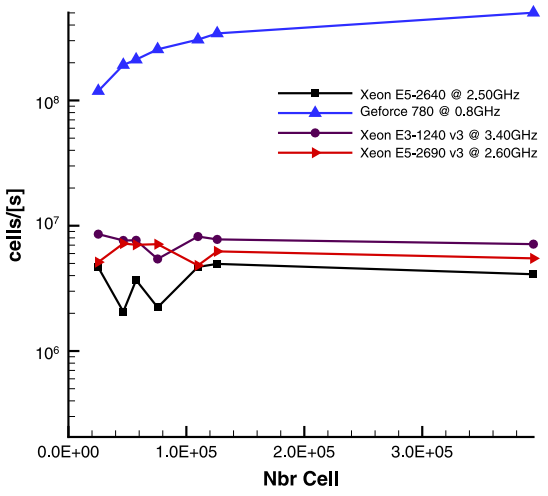


Fig. 5. Performance in terms of updated cells per second for one Runge–Kutta stage on a Geforce 780 compared to 3 other single CPUs for a RANS simulation of 2D flow on a supersonic compressor cascade.

averagely from the GPU. It is very important here to specify the data storage layout, as it can improve the data dependency and the memory access. Linear solvers, such as GMRES, are based on SpMV operations and belong thus to the same class.

Latency-Bound Inherently Sequential operations provide a very limited fine-grained parallelism, insufficient to run the GPU in a profitable regime. This third class deals with functions operating on a small amount of data with a high dependency and a non-regular memory access. Classical factorization algorithms, used for instance in implicit solvers as preconditioner, belong to this class since they are based on Gaussian elimination [33,34,9] and handle sparsely populated matrices. The peculiar aspect of factorization is that the algorithm, in general, is recursive and the entries are computed serially. Some linear solvers, such as the GPU version of PETSc [35], perform the ILU factorization on the CPU as a response to its difficult adaptation to massively parallel hardware. In this case, the entire system matrix needs to be transferred to the host and back. The communication through the PCI bus between host and GPU is not encouraged for optimized performance. For large systems, this alternative has no benefits as the fast CPU ILU factorization is not compensating the expensive communication.

The other approach is to optimize the ILU factorization on the GPU. In order to expose more parallelism during the assembly of the ILU matrices, it is possible to identify independent rows that can be updated concurrently [33]. Few options are available to improve the performance of the ILU factorization on the GPU among them multi-coloring [36,37,34] and level-scheduling [38].

While the methods cited above increase slightly the parallelism of the factorization other methods such as the iterative ILU [39] propose a novel algorithm. The factorization is replaced by a minimization problem able to provide an approximate L and U with more fine-grained parallelism. The method relies on a fixed point iteration $x^{n+1} = G(x^n)$ which is guaranteed

to converge [39,40] after a set of sweeps.[4] The GPU implementation of this method, presented in [9, p. 5], evokes a trade-off between convergence and parallelism as the fixed point iteration tends to use less frequently updated values when more threads are used.

3.2. Qualitative analysis of time integration methods on CPU and GPU

In this section, we present a qualitative model able to give some insights into the performance of the two integration methods on the GPU and on the CPU. It is based on the performance of all methods for one flow iteration which can be then extrapolated to cover realistic cases. For that purpose, this section will introduce a set of ratios, which will be used to link the performance of different methods.

The explicit time integration is not different from the space integration in terms of the underlying type of computations. Both are embarrassingly parallel and only few memory loads and few arithmetic operations are done per cell for the explicit time integration compared to the space integration. The execution time of this operation (t_T^{Exp}) can be thus neglected for the definition of the ratio of execution time of implicit to explicit solver for a flow iteration on CPU:

$$R_{CPU}^{ITR} \approx \frac{t_S + t_T^{Imp}}{t_s} = 1 + \frac{t_T^{Imp}}{t_S}, \tag{11}$$

with t_S the space integration for both solvers. The approximation (see Eq. (11)) guarantees a faster explicit flow iteration on the CPU ($R_{CPU}^{ITR} > 1$) as it assumes the explicit time stepping is less time consuming than the implicit time stepping. Theoretically, very high values of R_{CPU}^{ITR} are possible, but in practice researchers [41,14] report a 30%–80% of global execution time for the time integration. Consequently, R_{CPU}^{ITR} could reach values of 5 and possibly one order of magnitude for a dominant implicit time integration of 90%. The study of the different execution times of both methods on the CPU revealed the well-known fact: Explicit solvers cannot compete with implicit solvers unless the convergence rates are similar which is rather unusual.

In the following the ratio of execution time of implicit to explicit solvers for a flow iteration on a GPU is considered. Eq. (12) relates R_{GPU}^{ITR} to the same ratio on the CPU (R_{CPU}^{ITR}) as follows:

$$R_{GPU}^{ITR} = R_{CPU}^{ITR} * \frac{a_{Exp}^{GPU}}{a_{Imp}^{GPU}}, \tag{12}$$

with a_{Exp}^{GPU} and a_{Imp}^{GPU} the speedups of explicit and implicit solvers respectively on the GPU. Explicit solvers profit more from the GPU as they have only *compute-bound embarrassingly parallel* operations. Implicit solvers on the GPU profit averagely from the GPU, as they contain *memory-bound embarrassingly parallel* operations for the linear solver and *latency-bound inherently sequential* operations when standard[5] factorization-based preconditioners are used. The more an implicit solver is dominated by the time integration the less it benefits from the GPU favoring the explicit solver on the GPU. A single explicit flow iteration is already faster on the CPU than an implicit flow iteration ($R_{CPU}^{ITR} > 1$) and the GPU increases the ratio by 1–2 orders of magnitude ($a_{Exp}^{GPU}/a_{Imp}^{GPU} \gg 1$). The decisive aspect in the performance comparison is the portion of the linear solver from the total execution time of the implicit solver.

The speedup of 1 single flow iteration is, however, not determinant for the global performance as both methods have different convergence rates. The final comparison depends more on the ratio of convergence of both methods defined as:

$$R_C = \frac{N_{ITR}^{Exp}}{N_{ITR}^{Imp}}, \tag{13}$$

with N_{ITR}^{Exp} and N_{ITR}^{Imp} the number of flow iterations required to reach a converged solution for an explicit and an implicit solver respectively. This ratio is in practice large as explicit solvers are severely restricted in terms of CFL condition (see Section 2). If results are available for the two solvers both on the CPU and on the GPU for 1 flow iteration, different conclusions can be drawn depending on the ratio of convergence.

If we apply the above-introduced model on a case of a turbine nozzle guide vane, depicted in Fig. 6, we observe that one flow iteration of the GPU explicit solver is the fastest followed by the CPU explicit solver, the GPU implicit solver and finally the CPU implicit solver. When R_C increases, the execution time of one flow iteration for the implicit solver is the same but the execution time of the equivalent number of explicit iterations is linearly increasing. The explicit solver on the GPU is the fastest alternative to a certain value of R_C ($R_C \approx 20$). After this value, the implicit GPU is the fastest alternative. The maximum value of R_C, for which the explicit solver is still the fastest choice is equal to its acceleration for one flow iteration compared to the direct competitor, here the GPU implicit solver. In the results section more parameters will be considered such as the CFL number and the linear solver stopping condition, as both have an influence on the implicit solver.

4 A sweep is one full update of the L and U matrices.

5 Iterative incomplete factorization are not included.

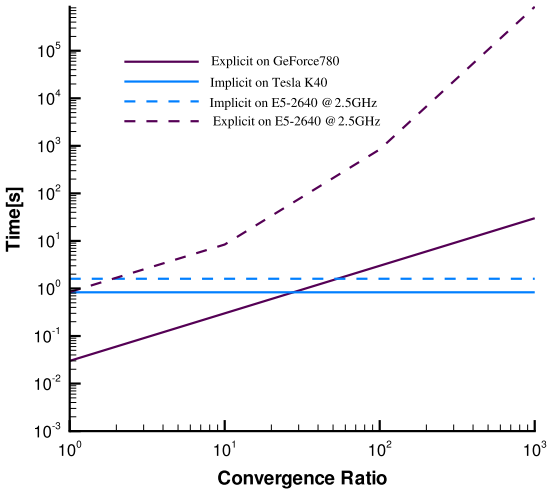


Fig. 6. Execution time of implicit solver for one flow iteration and explicit solver for equivalent flow iterations both on the CPU and the GPU as a function of the convergence ratio for a turbine nozzle guide Vane (Blade definition from [42]).

Even though the current work does not provide results for the CPU parallelization nor for the effect of the convergence acceleration (e.g. Residual smoothing and multigrid), the model shown in Fig. 6 can handle these cases. The parallelization for both explicit and implicit solvers on both GPU and many/multi-core CPU corresponds, indeed, to a translation of the performance curve[6] to regions of shorter execution times. Depending on the degree of acceleration the fastest combination of method and hardware can change. On the other hand, a convergence acceleration of the explicit or the implicit solver will change the convergence ratio R_C, defined in Eq. (13). A faster converging explicit solver for instance will have a lower R_C and a different value is read from the same performance curve in Fig. 6.

4. Implementation

4.1. Explicit GPU solver

In this work, we examine the GPU parallelization of explicit solvers using a CUDA implementation, which is an optimized version of an initial GPU solver ported earlier [43] from an in-house serial CFD solver. The CPU reference solver is running on a single core CPU and profiting from standard compiler optimization. The explicit time stepping algorithm, depicted in Fig. 7, is based on a set of functions (space, time integration and boundary updates) which are iteratively applied until the flow converges. The convergence is defined as a relative residual drop of six orders of magnitude. First, the space integration is performed in all cells by gathering fluxes on the cell faces and summing them up, followed by adding the source term. The time integration follows the Runge–Kutta scheme:

$$\begin{aligned}
\vec{W}_I^{(0)} &= \vec{W}_I^{n} \\
\vec{W}_I^{(1)} &= \vec{W}_I^{(0)} - \alpha_1 \frac{\Delta t_I}{\Omega_I} \vec{R}(\vec{W}^{(0)}) \\
\vec{W}_I^{(2)} &= \vec{W}_I^{(0)} - \alpha_2 \frac{\Delta t_I}{\Omega_I} \vec{R}(\vec{W}^{(1)}) \\
&\vdots \\
\vec{W}_I^{n+1} &= \vec{W}_I^{(m)} = \vec{W}_I^{(0)} - \alpha_m \frac{\Delta t_I}{\Omega_I} \vec{R}(\vec{W}^{(m-1)}).
\end{aligned} \tag{14}$$

While typical Runge–Kutta scheme stores intermediate values $W_I^{(k)}$, this formulation stores only the initial solution $W^{(0)}$ and the current solution $W^{(k)}$ along with the residual. The memory footprint is consequently independent of the number of stages (m) leading to a constant memory consumption of 3 arrays of $N_{cells} * N_{var}$ instead of $2 + m$ arrays. The speedup for explicit solvers depends on the level of CUDA code optimization and on the specifications of the used GPU (mainly the memory bandwidth and the computational power in double precision). The register consumption of every kernel in the implementation has been kept to a minimum to boost the occupancy, which in turn produces enough blocks of threads that hid the memory latency. Some memory-bound kernels such as the Runge–Kutta function reach 60%–70% of the memory

6 Curve relating the execution time of a simulation to its flow convergence.

Table 1
Execution time of 100 iterations of the explicit solver with 4 Runge–Kutta stages on 2 different CPUs and 2 different GPUs (used Hardware listed in Table 3).

NCells	K40 (1 Stream) [min]	K40 (7 Streams) [min]	GTX780 [min]	E5-2640 [min]	E3-1240 [min]	Speedup $\frac{T_{E3}}{T_{K40}}$	Speedup $\frac{T_{E5}}{T_{K40}}$
116k	0.055	0.034	0.039	1.85	1.31	38.5	54.4
290k	0.105	0.089	0.102	6.22	4.41	49.5	69.9
552k	0.179	0.155	0.199	12.99	9.15	59.0	83.8
1070k	0.32	0.304	0.393	27.04	19.45	64.0	88.9

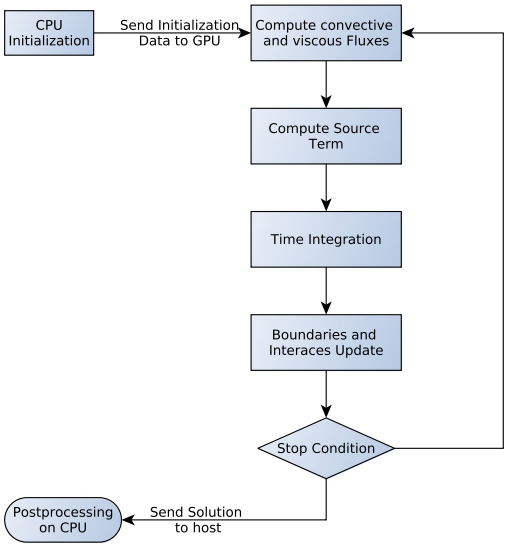


Fig. 7. The solver computes first inviscid and viscous residuals and source term. Second the time integration takes place and finally, boundaries and mesh interfaces are updated.

bandwidth.[7] Compute-bound kernels such as the convective flux evaluation have been improved by using multi-coloring. Speedups ranging from 1 to 2 orders of magnitude have been reached over the serial execution (see Table 1). Multi-streaming enables every mesh block to be assigned to a different stream increasing the level of parallelism. The effect of multi-streams fades, however, with the size of the treated meshes because of the scarce GPU resources.

4.2. Implicit GPU solver

In this work we use a GPU accelerated implicit solver [14] ported earlier from a serial C/C++ in-house code, which uses ILU preconditioner with flexible GMRES from PETSc package [44] for the time integration. The Runge–Kutta [45] multistage method is used for the time integration. It solves multiple successive linear systems for every flow iteration, in which only the right-hand side is updated and then multiplied by a different stage coefficient α:

$$\begin{aligned} \vec{W}^{(0)} &= \vec{W}^{n} \\ A^{(0)}[\vec{W}^{(1)} - \vec{W}^{(0)}] &= -\alpha_1 \vec{R}(\vec{W}^{(0)}) \\ A^{(0)}[\vec{W}^{(2)} - \vec{W}^{(0)}] &= -\alpha_2 \vec{R}(\vec{W}^{(1)}) \\ &\vdots \\ A^{(0)}[\vec{W}^{(m)} - \vec{W}^{(0)}] &= -\alpha_m \vec{R}(\vec{W}^{(m-1)}) \\ \vec{W}^{n+1} &= \vec{W}^{(0)} + [\vec{W}^{(m)} - \vec{W}^{(0)}]. \end{aligned} \tag{15}$$

The assembly of the system matrix is carried out with CUDA and the sorting is performed by the efficient GPU-based library THRUST [46]. Inserting the values in the system matrix requires more attention because a large number of cell Jacobians are computed at the same time. There are two strategies for the assembly procedure: the first approach is to gather all the entries and store them scattered in different arrays for row, column and values, then sort them out to the required storage format. The second approach computes the entries and inserts them directly in their final positions in the global matrix. While the first method has a clear pattern for the memory access during the data gathering, it requires a costly sorting

[7] Both on Geforce GTX780 and Kepler K40 with 288 GB/s the vendor provided peak memory bandwidth.

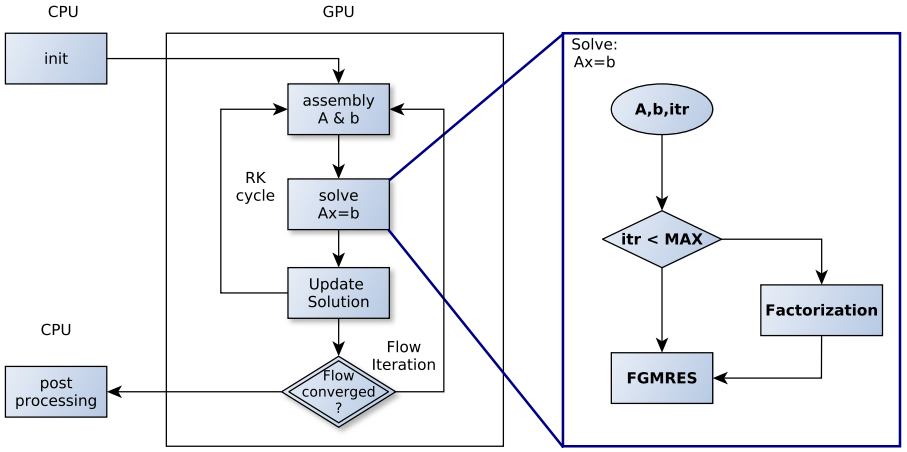


Fig. 8. Flow solver algorithm showing the full GPU assembly and linear solver with *on-demand* factorization.

function. The second approach avoids the sorting function but it cannot guarantee a coalesced access while inserting the entry in the global matrix. The extra cost of the uncoalesced access depends on the algorithm used to get the global index and the targeted storage format. In practice, the first approach is sometimes faster on a GPU and has been used in this work. The cell Jacobians have been stored in a large array with an index based on the cell index in the mesh in order to keep coalesced memory access for the reading/writing on the global memory. The array receives first all entries, which are next sorted by the row index then the column index.

Before choosing the linear solver we performed a benchmark for different algorithms, which is shown in Table 2. The tested GPU libraries are Paralution [47] and MAGMA [48]. The benchmark shows also the performance of PETSc [44] on the CPU (Xeon E3-1240) for the same meshes. Both GPU libraries have functions that rely on the cuSparse[8] implementation of the ILU factorization with level scheduling. For the application of the preconditioner within the linear solver iteration, Paralution relies only on the cuSparse triangular solver while MAGMA [11] proposes also an Incomplete Sparse Approximate Inverse method (ISAI). All methods use the flexible GMRES algorithm. For small meshes on the GPU, an ILU with ISAI triangular solver is faster while for large meshes the Paralution library provides the best GPU performance. The superiority of the ILU CPU performance is expected (cf. Section 3.1). We should precise, however, that the CPU is fast on building the ILU preconditioner and spends most of the execution running the linear solver. The GPU, on the other hand, dedicates more time to the preconditioner building rather than the linear iterations. In a context of multi-stages Runge–Kutta (see Eq. (15)) the same preconditioner is reused in all stages, which improves the GPU performance. Based on the results of the benchmark the chosen linear solver is the preconditioned GMRES of the Paralution library [47] with a restart Krylov subspace of 10 to 30 vectors.

In addition to the standard algorithm for the iterative solver, we use an *on-demand* factorization [14] to improve the speedup. The *on-demand* feature, depicted in Fig. 8, reduces the building time of the linear system by freezing the ILU factorization for some flow iterations. When skipping the ILU factorization, the linear solver uses an outdated preconditioner based on the system matrix of previous flow iteration. This causes the linear solver to perform some extra linear iterations. A control mechanism of the *on-demand* factorization, with the number of performed linear solver iterations as an input, keeps the balance between saved factorization time and extra iterations. A threshold value is defined as a limit for the tolerated increase of the number of iterations, after which an update of the preconditioner is performed. The limit depends on the time step, the major contributor to the system matrix diagonal. Consequently, it is a function of the CFL number with a dependency also on the mesh cell size:

$$Itr_{\max} = a + b * CFL, \tag{16}$$

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 key metric to assess the *on-demand* effect is the total extra iterations compared to the nominal case of a full ILU update. The optimal case is to skip the maximum number of updates while keeping the extra iterations as low as possible.

Speedups of 1 order of magnitude have been reached on 3D flow simulation [14]. The *on-demand* strategy reduced the number of calls for the preconditioner by 86% accelerating the linear solver by a factor of 2× to 3×. As it will be detailed in the benchmarks the *on-demand* feature is essential to have a competitive GPU performance for preconditioned linear solvers.

[8] http://docs.nvidia.com/cuda/cusparse/.

Table 2
Execution time of the ILU preconditioned FGMRES(30) for 33 iterations using different GPU libraries.

NCells	N_{cols}	nnz	Paralution [s] cuSparse ILU	MAGMA Sparse [s] cuSparse ILU	MAGMA Sparse [s] ISAI ILU	PETSc [s] ILU
116k	264 640	7.8M	1.8	2.19	1.224	0.30
290k	1 058 560	35.2M	2.85	4.87	5.652	1.35
552k	2 249 440	76.3M	4.7	9.1	12.96	2.92
1070k	–	–	–	–	–	6.36

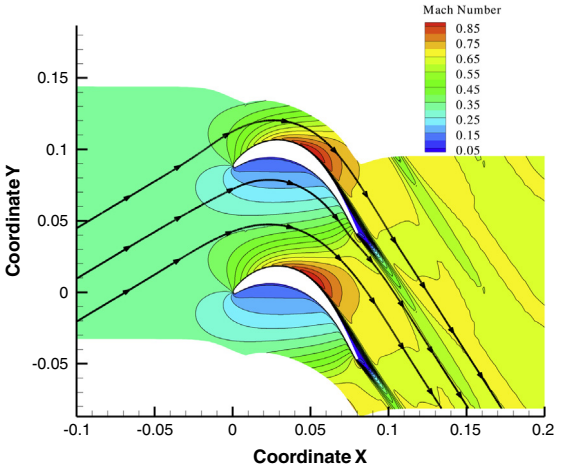


Fig. 9. Mach number contours around the t106c nozzle guide vane.

Table 3
Hardware used in the benchmark.

Reference	Clock rate [GHz]	Cache size [MB]	Memory bandwidth [GB/s]	Global memory [GB]
E3-1240	3.4	8	21	–
E5-2640	2.5	15	42.6	–
Geforce GTX 780	0.863	0.064	288.4	3
Tesla K40	0.745	0.064	288	12

5. Numerical experiments

For the benchmark two Intel Xeon CPUs are used with different clock rates and two different GPUs are used with different local memory capacities (see Table 3). The Geforce GTX 780 with only 3 GB of local memory is running the GPU explicit solver and the Kepler k40 card with 12 GB of local memory is running the GPU implicit solver.

5.1. Subsonic turbine cascade: T106c

T106c is a very high-lift, mid-loaded low-pressure turbine blade [8]. The blade turns an incoming flow and reduces its pressure as depicted in Fig. 9. Different mesh sizes are used for the benchmark. In a first step, an average execution time for one flow iteration has been calculated from 20 iterations. The explicit solver uses a four-stages Runge–Kutta scheme and the implicit solver uses a two stages of Jacobian-Trained Krylov Implicit-Runge–Kutta scheme (JT-KIRK) with a CFL number of 50. The execution time of one flow iteration for different mesh sizes is depicted in Fig. 10.

We first observe a difference of one to two order of magnitude between the execution time of the explicit solver on the GPU and the implicit solver on the CPU. This is due to the combination of two facts: first explicit solvers are inherently faster per iteration and secondly, they benefit largely from the GPU acceleration as they include mostly operation very suited to the GPU architecture (cf. Section 3). We observe also that the performance of the implicit solver on the CPU can be improved with a higher clock frequency, as Xeon E3 slightly outperforms the Xeon E5. The GPU is not able to run the one million mesh case. The GPU solver used in this work (cf. Section 4) uses ghost cells adding 4 layers of cells for every direction of mesh blocks which can increase the mesh size by 20% to 50%. Numerical fluxes and Jacobians are also stored explicitly in large arrays before being sorted out for the linear solver. An optimization of the memory footprint of the solver can let the GPU compute larger meshes.

In order to compare all alternatives from different hardware, the slowest combination is chosen as a common reference, which is in this case the implicit solver on the CPU with the lower clock rate (Xeon E5). This approach leads to high speedups

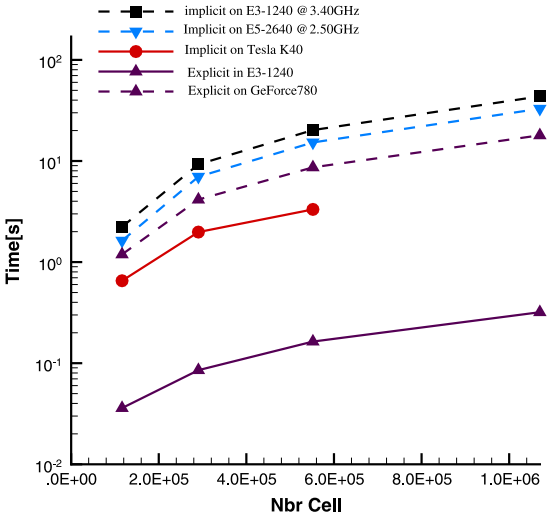


Fig. 10. Execution time for one flow iteration of the implicit solver and the explicit solver both on different CPUs (dashed line) and the GPU (full line).

Table 4
Speedup of explicit and implicit solvers on the GPU and the CPU over the implicit solver on the Xeon E5-2640 CPU for 1 flow iteration.

N_{Cells} [−]	Implicit E5-2640	Implicit E3-1240	Explicit E5-2640	Implicit Tesla k40	Explicit Geforce 780
116k	1	1.37	1.87	3.41	62.0
290k	1	1.32	2.23	4.67	108.5
552k	1	1.32	2.33	6.09	123.6
1070k	1	1.32	2.42	N.A	136.4

which do not reflect a GPU acceleration only, as usual speedups do. It reflects also the fact that an explicit flow iteration is lighter by nature while the implicit flow iteration solves a linear system of equations. Independently of how efficiently the linear system is solved, an explicit Runge–Kutta stage is much lighter with its very few operations. Table 4 summarizes the performance of the combinations highlighting the growing gap between the explicit solver on the GPU and the other alternatives with the increase of the mesh size. The GPU, as a throughput-oriented device, is used more efficiently when the workload increases of embarrassingly parallel operations. The CPU, on the other hand, has rather a constant performance independently of the mesh size.

Explicit solvers are faster since they perform fewer operations and the GPU version of both methods is faster, even though the explicit solver takes more advantage from the GPU. The execution times of 1 flow iteration are used to extrapolate a performance comparison for different convergence ratios (see Fig. 11). The convergence ratio is used to cover a wider range of applications as in some areas implicit solvers converge much faster than the explicit and in other areas the difference is not very pronounced. The initial ranking is valid for the unrealistic convergence ratios of 1 for which explicit solver and implicit solver converge after the same amount of flow iterations. Realistic ratios are in general between 20 to 100 for turbomachinery simulations. A common pattern is repeated for all mesh sizes predicting the explicit GPU solver to be the fastest alternative for low convergence ratios and the implicit GPU solver for large convergence ratios.

In the next step, the flow around t106c is solved for a relative residual drop of six order of magnitude. Depending on the mesh size, the implicit solver required between 1328 and 1366 flow iterations for a CFL of 50. The explicit solver required between 22 900 and 23 400 flow iterations, which leads to a convergence ratio[9] $R_C = 17$. For this R_C value, the extrapolation based on the execution time of one flow iteration (see Fig. 11) predicts the explicit solver on the GPU to be the fastest solver followed by the GPU implicit solver for all mesh sizes. This approximation does not cover, however, the *on-demand* factorization (cf. Section 4). This technique acts on reducing the global execution time of implicit solvers on the GPU by skipping most of the ILU factorization used for the preconditioner. It does not change the number of flow iterations and thus the convergence ratio is untouched. Fig. 12(a) depicts the speedup (with reference to the explicit solver on E5-2640) of an entire flow simulation around the t106c blade including the performance of GPU solvers with the *on-demand* factorization (ODILU). The CPU delivers similar performance independently from the mesh size while the GPU has a better performance for larger meshes.

For a higher CFL number, every single implicit flow iteration is slower on both the GPU and CPU but the convergence ratio (see Eq. (13)) is larger favoring the implicit solver. For a $CFL = 100$ the implicit solver requires between 756 and 781 flow

[9] Defined in Eq. (13).

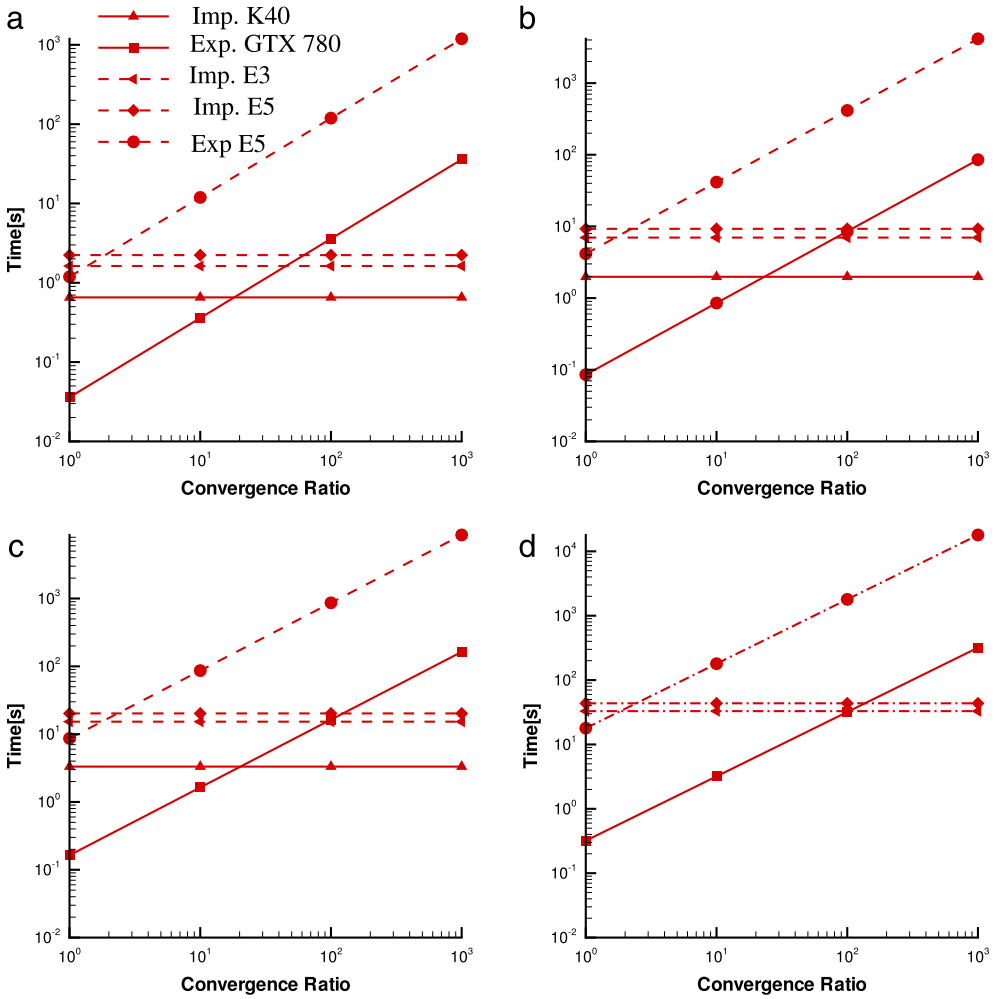


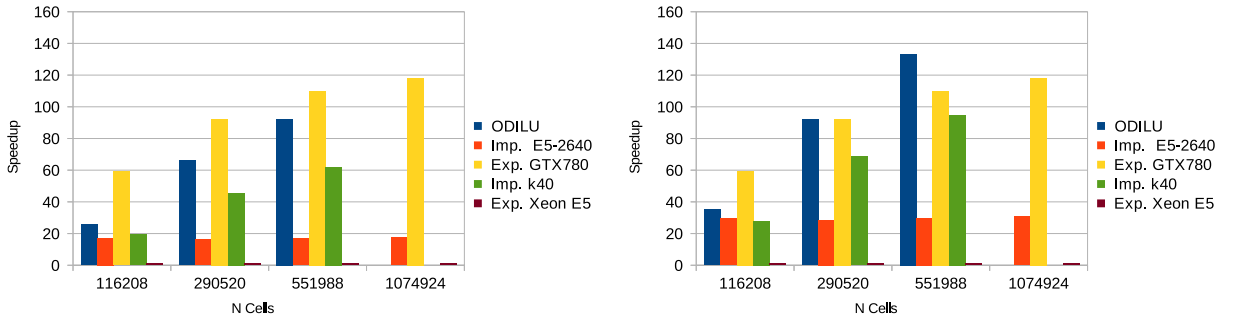
Fig. 11. Case t106c: Execution time of implicit solver for one flow iteration and explicit solver for equivalent flow iterations both on the CPU (dashed line) and the GPU (full line) as a function of the convergence ratio for increasing mesh resolution (a: smallest mesh, d: largest mesh).

iterations, which corresponds to a drop of 42% in the number of flow iterations compared with the number of flow iterations of the implicit solver with $CFL = 50$. Fig. 12(b) summarizes the execution time until convergence of the used solvers scaled by the execution time of the explicit solver on the CPU for a $CFL = 100$. Depending on the mesh size the fastest combination is changing. The mesh with 1 M cells was not run by the implicit solver on the GPU because of memory limitations. The convergence ratio contributed largely to the improved GPU Implicit performance compared to the explicit CPU solver as reflected by the increase of the speedup for the GPU implicit solver in Fig. 12(b) compared to Fig. 12(a). Higher CFL values indeed reduce the total number of flow iterations required for the flow convergence, which entails fewer ILU builds. At the same time the linear solver requires more linear iterations to converge, since the linear system is more ill-conditioned, but this does not outweigh the reduction in time achieved by avoiding more ILU builds.

As a result, the increase in the CFL number improved also the acceleration of the CPU implicit solver compared to the reference CPU explicit solver. In the next test case, we focus consequently more on the effect of the CFL change for the GPU acceleration of the implicit solver compared to the CPU implicit solver.

5.2. LS89

The LS89 test case is described in the experimental work of Arts [42]. In this section, a benchmark is presented of the implicit solver on the CPU and on the GPU for different parameters: First, the CFL number is varying within a stable envelop, then the linear solver stopping criteria. Based on the observations in Section 3 about the linear solver benefiting less from GPU than the assembly, the global GPU speedup of the implicit solver is expected to decrease as a result of the growing linear solver portion. In Fig. 13 the profiling results on the CPU of the implicit solver are depicted for a set of 6 stable CFL numbers (25–150) along with the GPU speedup. The figure confirms the negative effect of the CFL increase on the GPU performance even with the *on-demand* factorization.



(a) $CFL_{Imp.} = 50$. (b) $CFL_{Imp.} = 100$.

Fig. 12. Speedup of all solvers with reference to the explicit solver on E5-2640 for two different CFL numbers for the implicit solver and the same CFL number for the explicit solvers ($CFL_{Exp.} = 2.5$).

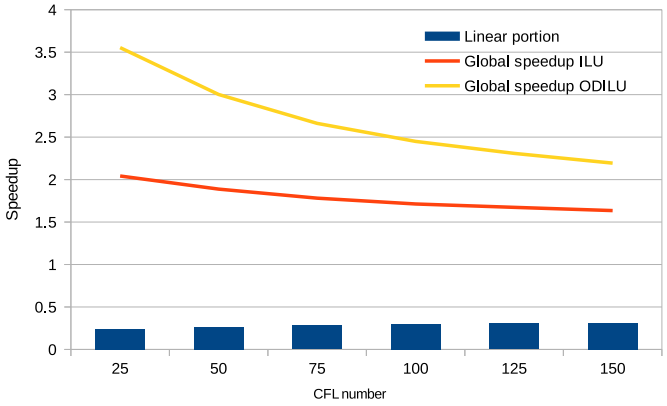


Fig. 13. Plot of the effect of the CFL increase on the portion of the linear solver on CPU and the global speedup both with full ILU update and *on-demand* update (OD-ILU).

The linear solver stopping criteria are also important in the benchmark as they influence directly the solving phase of the linear systems. The GPU speedup decreases for more severe stopping criteria. Such a severe stopping condition does not improve the flow convergence and makes every single flow iteration slower. Therefore, an adequate relative stopping criterion of 10^{-3} has been chosen for all the benchmark shown in this work.

6. Discussion

Explicit solvers on a single core CPU are the slowest alternative in all studied test cases since stencil-based operations have a great potential of parallelization that the serial implementation is not using. The GPU parallelization enables the explicit solver to still compete with implicit solver despite their low convergence rates. An interesting point is that explicit solver are so efficient in memory usage that very heavy meshes of millions of cells still fit in the GPU global memory.

When the mesh fits the GPU memory, the GPU implicit solver is the fastest in the two cases since it combines the accelerated stencil operation for the residual and the Jacobian calculation while not suffering very much from the preconditioning due to the *on-demand* factorization. Optimizing the preconditioner update within a large sequence of linear systems such as in a steady CFD simulation is not a new topic. Hartmann et al. [49] assessed the impact of using a periodically updated preconditioner on the number of linear solver iterations on the CPU. In [50] Tebbens et al. introduced a similar performance assessment for a proposed method of approximate preconditioner update on the CPU. Anzt et al. [51] use and iterative ILU on the GPU for which the preconditioner of one system is the first guess for the next system. He shows that updating a previous preconditioner could be fast and effective.

The memory usage of implicit solver limits, however, the mesh size to about 1 million cells for Tesla K40 and 0.25 million for GeForce 780 with the actual solver memory footprint. This number can fluctuate based on memory usage optimization but unless no matrix storage is done it should remain in the same order of magnitude. Therefore for heavy meshes, the implicit GPU is not available and then the explicit GPU and implicit CPU have comparable performance. For some applications (e.g. the adjoint method) the flow should be resolved to machine accuracy (Residual drop of 10^{-16}). In that case, the ratio of convergence between the explicit and the implicit schemes is much larger. At the same time for other applications (e.g. chemical kinetics in reactive-flow simulations) meshes are very large for actual CPUs capacities to use an implicit solver, consequently the explicit solver is the only alternative available.

The order of the space integration has an impact on the amount of computation for every flow iteration. This might change the ratio between space and the time integration for the implicit solver. Nevertheless, as the same space integration is to be found in both implicit and explicit solver, the order of the discretization scheme has no impact on the choice of the best combination between time integration method and hardware. The finding of this work apply mostly to Finite Volume discretized flows in turbomachinery. The convergence ratio, which is very important in the comparison depends also on acceleration techniques of both methods.

7. Conclusion

Different combination of time integration methods and computational hardware have been presented. A classification has been introduced based on the suitability of CFD operations to the GPU hardware. The comparison highlighted the high possible acceleration of explicit solvers as based on embarrassingly parallel functions. Difficulties in accelerating implicit solvers have been discussed including inherently sequential incomplete factorization used as a preconditioner for ill-conditioned linear systems in the implicit time integration. We observed that the GPU is able to extend the range of usability of explicit solvers to averagely good converging solvers.

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] Z. Johan, T.J. Hughes, A globally convergent matrix-free algorithm for implicit time-marching schemes arising in finite element analysis in fluids, Comput. Methods Appl. Mech. Engrg. 87 (2) (1991) 281–304.
[2] Tobias Brandvik, Graham Pullan, An accelerated 3D Navier–Stokes solver for flows in turbomachines, J. Turbomach. 133 (2) (2011).
[3] B. Brock, A. Belt, J.J. Billings, M. Guidry, Explicit integration with GPU acceleration for large kinetic networks, J. Comput. Phys. 302 (2015) 591–602.
[4] K.I. Karantasis, E.D. Polychronopoulos, J.A. Ekaterinaris, High order accurate simulation of compressible flows on GPU clusters over software distributed shared memory, Comput. & Fluids 93 (2014) 18–29.
[5] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, J. Comput. Phys. 227 (24) (2008) 10148–10161.
[6] M. Lefebvre, P. Guillen, J.-M. Le Gouez, C. Basdevant, Optimizing 2D and 3D structured Euler CFD solvers on graphical processing units, Comput. & Fluids 70 (2012) 136–147.
[7] B. Van Leer, W.-T. Lee, P.L. Roe, K.G. Powell, C.-H. Tai, Design of optimally smoothing multistage schemes for the Euler equations, Commun. Appl. Numer. Methods 8 (10) (1992) 761–769.
[8] J. MichÃAlek, M. Monaldi, T. Arts, Aerodynamic performance of a very high lift low pressure turbine airfoil (t106c) at low Reynolds and high Mach number with effect of free stream turbulence intensity, J. Turbomach. 134 (6) (2012) 061009.
[9] E. Chow, H. Anzt, J. Dongarra, Asynchronous iterative algorithm for computing incomplete factorizations on GPUs, in: International Conference on High Performance Computing, Springer, 2015, pp. 1–16.
[10] H. Anzt, E. Chow, J. Dongarra, Iterative sparse triangular solves for preconditioning, in: European Conference on Parallel Processing, Springer, 2015, pp. 650–661.
[11] H. Anzt, E. Chow, T. Huckle, J. Dongarra, Batched generation of incomplete sparse approximate inverses on GPUs, in: Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, IEEE Press, 2016, pp. 49–56.
[12] L. Luo, J.R. Edwards, H. Luo, F. Mueller, A fine-grained block ILU scheme on regular structures for GPGPUs, Comput. & Fluids 119 (2015) 149–161.
[13] L. Fu, Z. Gao, K. Xu, F. Xu, A multi-block viscous flow solver based on GPU parallel methodology, Comput. & Fluids 95 (2014) 19–39.
[14] M.h. Aissa, L. Mueller, T. Verstraete, C. Vuik, Acceleration of turbomachinery steady simulations on GPU, in: Euro-Par 2016: Parallel Processing Workshops, Springer, 2017, in press.
[15] K.E. Niemeyer, C.-J. Sung, Accelerating moderately stiff chemical kinetics in reactive-flow simulations using GPUs, J. Comput. Phys. 256 (2014) 854–871.
[16] A. Hartstein, V. Srinivasan, T. Puzak, P. Emma, On the nature of cache miss behavior: Is it $\sqrt{2}$, J. Instr.-Level Parallelism 10 (2008) 1–22.
[17] S. Hong, H. Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in: ACM SIGARCH Computer Architecture News, Vol. 37, ACM, 2009, pp. 152–163.
[18] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, W.-m.W. Hwu, An adaptive performance modeling tool for GPU architectures, in: ACM Sigplan Notices, Vol. 45, ACM, 2010, pp. 105–114.
[19] K. Li, W. Yang, K. Li, Performance analysis and optimization for SpMV on GPU using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. 26 (1) (2015) 196–205.
[20] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al., Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, ACM SIGARCH Comput. Archit. News 38 (3) (2010) 451–460.
[21] V. Volkov, Understanding latency hiding on GPUs.
[22] D.B. Kirk, W.H. Wen-mei, Programming Massively Parallel Processors: A Hands-on Approach, second ed., Newnes, 2013.
[23] J. Cheng, M. Grossman, T. McKercher, Professional Cuda C Programming, John Wiley & Sons, 2014.
[24] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.
[25] J. Blazek, Computational Fluid Dynamics:Principles and Applications, Elsevier, 2001.
[26] P.L. Roe, Approximate Riemann solvers, parameter vectors, and difference schemes, J. Comput. Phys. 43 (2) (1981) 357–372.
[27] B.V. Leer, Towards the ultimate conservative difference scheme V. A second order sequel to Godunov's method, J. Comput. Phys. 32 (1979) 101–136.
[28] V. Venkatakrishnan, Preconditioned conjugate gradient methods for the compressible Navier–Stokes equations, AIAA J. 29 (1991) 1092–1110.
[29] S.R. Allmaras, F.T. Johnson, P.R. Spalart, Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model, ICCFD7-1902.
[30] O. Kardani, A. Lyamin, K. Krabbenhøft, Application of a GPU-accelerated hybrid preconditioned conjugate gradient approach for large 3D problems in computational geomechanics, Comput. Math. Appl. 69 (10) (2015) 1114–1131.
[31] A. Harten, P.D. Lax, B. Van Leer, On upstream differencing and Godunov-type schemes for hyperbolic conservation laws, in: Upwind and High-Resolution Schemes, Springer, 1997, pp. 53–79.

[32] N. Bell, M. Garland, Implementing sparse matrix–vector multiplication on throughput-oriented processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 18.
[33] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, 2003.
[34] R. Li, Y. Saad, GPU-accelerated preconditioned iterative linear solvers, J. Supercomput. 63 (2) (2013) 443–466.
[35] V. Minden, B. Smith, M.G. Knepley, Preliminary implementation of PETSc using GPUs, in: GPU Solutions to Multi-Scale Problems in Science and Engineering, Springer, 2013, pp. 131–140.
[36] M. Naumov, P. Castonguay, J. Cohen, Parallel graph coloring with applications to the incomplete-LU factorization on the GPU, Tech. Rep., Nvidia Technical Report NVR-2015-001, Nvidia Corp., Santa Clara, CA, 2015.
[37] M.S.D. Lukarski, Parallel Sparse Linear Algebra for Multi-Core and Many-Core Platforms (Ph.D. thesis), Georgia Institute of Technology, 2012.
[38] M. Naumov, Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU, Tech. Rep. NVR-2011 1, NVIDIA Corp., Westford, MA, USA, 2011.
[39] E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM J. Sci. Comput. 37 (2) (2015) C169–C193.
[40] A. Frommer, D.B. Szyld, On asynchronous iterations, J. Comput. Appl. Math. 123 (1) (2000) 201–216.
[41] C. Cecka, A.J. Lew, E. Darve, Assembly of finite element methods on graphics processors, Internat. J. Numer. Methods Engrg. 85 (5) (2011) 640–669.
[42] T. Arts, M. Lambert de Rouvroit, A.W. Rutherford, Aero-Thermal Investigation of a Highly Loaded Transonic Linear Turbine Guide Vane Cascade, TN 174, von Karman Institute for Fluid Dynamics, 1990.
[43] M.H. Aissa, T. Verstraete, C. Vuik, Aerodynamic optimization of supersonic compressor cascade using differential evolution on GPU, in: International Conference of Numerical Analysis and Applied Mathematics 2015, Vol. 1738, (ICNAAM 2015), AIP Publishing, 2016, 480077.
[44] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, K. Rupp, B.F. Smith, S. Zampini, H. Zhang, PETSc Web page, (2015). URL http://www.mcs.anl.gov/petsc.
[45] S. Xu, D. Radford, M. Meyer, J.-D. Müller, Stabilisation of discrete steady adjoint solvers, J. Comput. Phys. 299 (2015) 175–195.
[46] N. Bell, J. Hoberock, Thrust: A productivity-oriented library for CUDA, GPU Comput. Gems Jade Ed. 2 (2011) 359–371.
[47] PARALUTION Labs, PARALUTION v.1.1.0, http://www.paralution.com/ (2016).
[48] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, I. Yamazaki, J. Dongarra, Optimizing krylov subspace solvers on graphics processing units, in: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, IEEE, 2014, pp. 941–949.
[49] S. Hartmann, J. Duintjer Tebbens, K.J. Quint, A. Meister, Iterative solvers within sequences of large linear systems in non-linear structural mechanics, ZAMM-J. Appl. Math. Mech./Z. Angew. Math. Mech. 89 (9) (2009) 711–728.
[50] J.D. Tebbens, M. Tuma, Efficient preconditioning of sequences of nonsymmetric linear systems, SIAM J. Sci. Comput. 29 (5) (2007) 1918–1941.
[51] H. Anzt, E. Chow, J. Saak, J. Dongarra, Updating incomplete factorization preconditioners for model order reduction, Numer. Algorithms 73 (3) (2016) 611–630.