Tenth International Conference on Computational Fluid Dynamics (ICCFD10), Barcelona, Spain, July 9-13, 2018

ICCFD10-057

GPUs Accelerated Three-Dimensional RANS Solver for Aerodynamic Simulations on Multiblock Grids

M. T. Nguyen*, P. Castonguay**, E. Laurendeau***
Corresponding author: minh-tuan.nguyen@polymtl.ca

* Postdoctoral fellow, Department of Mechanical Engineering, Polytechnique Montréal, Montreal, Quebec, Canada.
** Engineering Specialist, Advanced Aerodynamics Department, Bombardier Aerospace, Montreal, Quebec, Canada.
*** Professor, Department of Mechanical Engineering, Polytechnique Montréal, Montreal, Quebec, Canada.

Abstract: In this paper, graphical processing units (GPUs) are leveraged to accelerate Bombardier's Full Aircraft Navier-Stokes Solver (FANSC), a finite volume, cell-centered RANS flow solver for multiblock structured grids. The efficiency of different parallel smoothers on GPUs is studied, in the context of solving the RANS equations with a non-linear full approximation storage (FAS) multigrid scheme. Many variants of parallel red-black Gauss-Seidel and Jacobi solvers are investigated and their efficiency compared against sequential algorithms such as the lower-upper symmetric Gauss-Seidel solver on both CPUs and GPUs. Parametric studies on three-dimensional aircraft configurations are performed to identify the optimal smoothers and determine the optimal number of smoothing iterations on each multigrid level. The results show that the best runtime with the GPU code is obtained using a weaker smoother with more sweeps per multigrid level whereas the best runtime with the CPU code is obtained using a stronger smoother with fewer sweeps per multigrid level. Despite using a weaker smoother and therefore more iterations to converge to the final solutions, the GPU accelerated code is significantly faster than the CPU code.

Keywords: GPU, Multigrid, Implicit smoothers, RANS.

1 Introduction

In the past few years, Graphical Processing Units (GPUs) have increasingly played an important role in the world of high performance computing (HPC) due to their higher floating-point throughput, memory bandwidth and power efficiency compared to traditional CPU processors. In the field of Computational Fluid Dynamics (CFD) and more specifically external aerodynamics, many groups have successfully leveraged GPUs to accelerate simulations. The most impressive results have been obtained when accelerating high-order methods, which are well suited for GPUs due to their high ratios of floating point operations to memory operations. For example, Castonguay et al. [1] showed that using a high-order, compressible viscous flow solver for unstructured grids, very high computational throughput can be achieved. On a cluster where each node was equipped with 2 Intel Xeon x5670 and 1 Tesla C2070, their CPU+GPU code running on 16 nodes achieved almost a one order of magnitude speedup compared to the CPU-only code running on the same number of nodes. Other researchers have shown similar levels of performance when leveraging GPUs to accelerate high-order methods [2]. Although high order schemes have the potential to reduce the computational cost of simulations when

low error levels are required, they are generally less robust and require much effort to implement compared to low order schemes. Additionally, the use of low order schemes (usually 2nd order of accuracy) is more widely used in industrial applications due to their excellent robustness for a vast range of applications and their efficiency when engineering levels of accuracy are required. Fortunately, even low-order methods are well-suited for GPUs. The high number of mesh points required to accurately simulate flows over complex geometries leads to a large amount of parallelism that can be exploited on GPUs. Furthermore, since the performance of most low-order CFD codes is limited by the memory subsystem, they can benefit from the high memory bandwidth of GPUs. For these reasons, many researchers have successfully leveraged GPUs to simulate fluid flows using low order schemes and showed dramatically improved performance compared with conventional CPU implementations [3-6].

Unfortunately, several studies compared the performance of CPU and GPU codes using identical algorithms, which were often chosen to favor the GPU code. For example, many studies relied on the use of explicit schemes which are rarely used in low-order CFD codes used for production [7, 8]. In fact, many production codes rely exclusively on implicit methods where linear solvers are required and where sequential algorithms such as Gauss-Seidel and Incomplete LU (ILU) are used.
In this study, GPUs are leveraged using the parallel programming language CUDA to accelerate Bombardier Full Aircraft Navier-Stokes Solver (FANSC). The original version of FANSC uses many sequential algorithms such as the lower-upper symmetric Gauss-Seidel (LUSGS) solver used as a smoother in the non-linear multigrid scheme [9]. Those sequential algorithms are inefficient on massively parallel architectures such as GPUs. The main objective of this work is to study and identify parallel algorithms that are efficient on GPUs and have the potential to replace the sequential ones. More specifically, the efficiency of different smoothers in the context of solving the RANS equations with a non-linear multigrid scheme is studied. For example, many variants of red-black Gauss-Seidel and Jacobi solvers are developed and their efficiency studied on both CPUs and GPUs. Parametric studies for three-dimensional (3D) wing configurations are carried out to determine the optimal settings for GPU and CPU codes in terms of the type of implicit smoothers and the number of sweeps to use within each multigrid level (MG) sequence.

2 Bombardier's Full Aircraft Navier-Stokes Code (FANSC)

The flow solver used in this work is FANSC, which solves the RANS equations on multi-block structured grids [10] using a second-order, cell centered finite-volume method with a matrix dissipation scheme (MATD) [11]. The mean-flow governing equation can be expressed as

$$V\frac{\partial \boldsymbol{W}_c}{\partial t} = -\boldsymbol{R}(\boldsymbol{W}_c), \tag{1}$$

where $\boldsymbol{W}_c = [\rho, \rho u, \rho v, \rho w, pE]$ is the average state vector of conservative flow variables in the cell with volume V and $\boldsymbol{R}$ is the residual vector which includes contributions from the convective, viscous and dissipative fluxes. The subscript c denotes that the flow states are written using conservative variables. In this work, the steady-state equations are considered and pseudo-time stepping is not used, hence the temporal derivative in equation (1) is omitted. The flow states can be updated using inexact Newton iterations

$$\left[\frac{\partial \bar{\boldsymbol{R}}}{\partial \boldsymbol{W}_c}\right]^n \delta \boldsymbol{W}_c^{n+1} = -\boldsymbol{R}(\boldsymbol{W}_c^n), \tag{2}$$

where

$$\delta \boldsymbol{W}_c^{n+1} = \boldsymbol{W}_c^{n+1} - \boldsymbol{W}_c^n. \tag{3}$$

and $\left[\frac{\partial \bar{\boldsymbol{R}}}{\partial \boldsymbol{W}_c}\right]$ is an approximation of the true Jacobian matrix $\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{W}_c}\right]$. More specifically, a first order discretization of the dissipation fluxes and the thin-shear layer (TSL) approximation of the viscous

fluxes are used when constructing $\left[\frac{\partial \bar{\boldsymbol{R}}}{\partial \boldsymbol{W}_c}\right]$. To accelerate the convergence to the steady-state solution, a Full Approximation Storage (FAS) multigrid scheme [12] with W-cycles is employed in a recursive manner. On each grid level, the linear system in equation (2) is solved approximately using a fixed number of iterations of a smoother such as Jacobi or Gauss-Seidel on each block of the mesh. The description of different types of implicit smoothers that work efficiently on CPU and GPU architectures will be discussed in the next section 3.

The Spalart–Allmaras (SA) one-equation turbulence model [13] is employed. The discretization of SA equation analogously follows [13] in which the advection terms are discretized using a first order upwind scheme and the central finite difference method is used to discretize the diffusive terms. The non-linear turbulence transport equation is solved separately from the mean flow equations in a segregated approach. The SA equation is solved only on the finest grid. The turbulence effect of eddy viscosity is transferred to the coarse levels through a restriction process. To solve the linear system that arise when using Newton's method on the SA equation, an alternating direction implicit (ADI) scheme is used for the CPU code while a Jacobi solver is used for the GPU code.

3 Implicit smoothers

In this section, various solvers used as smoothers in the non-linear multigrid scheme are summarized. They consist of many sequential algorithms such as the lower-upper symmetric Gauss-Seidel (LUSGS) that work efficiently on CPU architectures and many variants of Gauss-Seidel and Jacobi solvers that are efficient on GPUs. The approximate Jacobian matrix $\left[\frac{\partial \bar{\boldsymbol{R}}}{\partial \boldsymbol{W}_c}\right]$ can be written as:

$$\left[\frac{\partial \bar{\boldsymbol{R}}}{\partial \boldsymbol{W}_c}\right] = [D_c] + [L_c] + [U_c], \tag{4}$$

where $[D_c]$ represents the diagonal terms of the matrix, $[L_c]$ the upper triangular terms and $[U_c]$ the lower-triangular terms. The diagonal, upper and lower block of approximate Jacobian matrix are given as follows:

$$[D_c]^n = \frac{1}{2}|A_c|_{i+\frac{1}{2}} + \frac{1}{2}|A_c|_{i-\frac{1}{2}} + {A_c}^v_{i+\frac{1}{2}} + {A_c}^v_{i-\frac{1}{2}}, \tag{5}$$

$$[U_c]^n = \frac{1}{2}A_{c_{i+\frac{1}{2}}} - \frac{1}{2}|A_c|_{i+\frac{1}{2}} - {A_c}^v_{i+\frac{1}{2}}, \tag{6}$$

$$[L_c]^n = -\frac{1}{2}A_{c_{i-\frac{1}{2}}} - \frac{1}{2}|A_c|_{i-\frac{1}{2}} - {A_c}^v_{i-\frac{1}{2}}. \tag{7}$$

where $|A_c|$, A_c and A_c^v are the absolute, convective and the viscous flux Jacobian, respectively.

The LUSGS smoother when applied to equation (2) involves executing the following two steps for a fixed number of sweeps

$$\begin{aligned} [D_c]^n \delta \boldsymbol{W}_c^{k+1/2} &= -\boldsymbol{R}(\boldsymbol{W}_c^n) - [L_c]^n \delta \boldsymbol{W}_c^{k+1/2} - [U_c]^n \delta \boldsymbol{W}_c^k, \\ [D_c]^n \delta \boldsymbol{W}_c^{k+1} &= -\boldsymbol{R}(\boldsymbol{W}_c^n) - [U_c]^n \delta \boldsymbol{W}_c^{k+1} - [L_c]^n \delta \boldsymbol{W}_c^{k+1/2}. \end{aligned} \tag{8}$$

The diagonal block matrixes are first inverted and their inverse is stored in memory. To reduce the storage overhead, the upper and lower block of the flux Jacobian matrices are computed on the fly. Unlike in the original presentation of Jameson and Yoon [14] , no approximation is made regarding the spectral radius of the approximate Jacobian matrix. Although there is parallelism available during the forward and backward sweeps for all cells where i+j+k is constant, the sweeps are accomplished following the natural order to improve cache reuse.

A variant of the LUSGS approach described above was proposed by Rossow [15] and will be denoted LUSGS-PRIM. Rossow shows that by writing the approximate Jacobian matrix using primitive variables $\boldsymbol{W}_p = [\rho, u, v, w, p]$ instead of the conservative variables $\boldsymbol{W}_c$, the calculation of off-diagonal terms in the Jacobian matrix is greatly simplified. The multi-symmetric sweep for LUSGS-PRIM are described in equation (9).

$$
\begin{aligned}
[D_p]^n \delta \boldsymbol{W}_p^{k+1/2} &= -\left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right] \boldsymbol{R}(\boldsymbol{W}_p^n) - [L_p]^n \delta \boldsymbol{W}_p^{k+1/2} - [U_p]^n \delta \boldsymbol{W}_p^k, \\
[D_p]^n \delta \boldsymbol{W}_p^{k+1} &= -\left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right] \boldsymbol{R}(\boldsymbol{W}_p^n) - [U_p]^n \delta \boldsymbol{W}_p^{k+1} - [L_p]^n \delta \boldsymbol{W}_p^{k+1/2}.
\end{aligned} \tag{9}
$$

where $\left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right]$ represents the transformation matrix from the conservative into primitive variables. The diagonal, lower and upper block of the flux Jacobian matrix in primitive variables are defined as

$$
\begin{aligned}
[D_p] &= \left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right] [D_c] \left[\frac{\partial \boldsymbol{W}_c}{\partial \boldsymbol{W}_p}\right], \\
[L_p] &= \left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right] [L_c] \left[\frac{\partial \boldsymbol{W}_c}{\partial \boldsymbol{W}_p}\right], \\
[U_p] &= \left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right] [U_c] \left[\frac{\partial \boldsymbol{W}_c}{\partial \boldsymbol{W}_p}\right].
\end{aligned} \tag{10}
$$

One significant disadvantages of the LUSGS algorithms given by equations (8) and (9) is that they are sequential. In other words, the change in the state variable of one cell can only be calculated once the change in the state variable of the previous cell is known. Sequential algorithms are not suited for GPU architectures and hence, alternative to these schemes were sought for. The simplest parallel smoother will be denoted by LUJACOBI-PRIM and involves using pointwise Jacobi iterations based on primitive variables. The change in the primitive variables is computed as

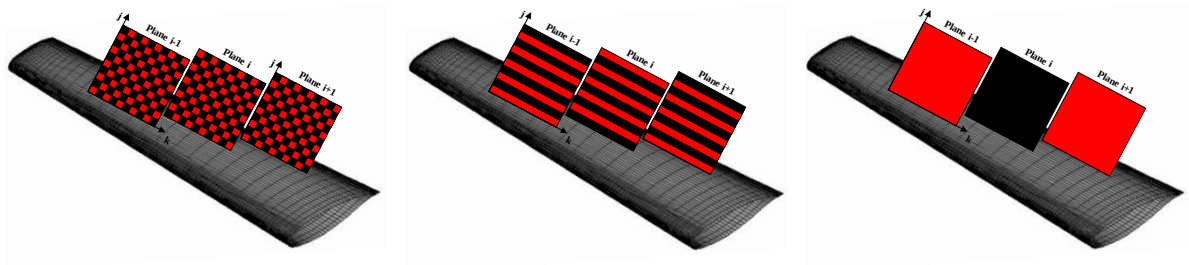
$$
[D_p]^n \delta \boldsymbol{W}_p^{k+1} = -\left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right] \boldsymbol{R}(\boldsymbol{W}^n) - [L_p]^n \delta \boldsymbol{W}_p^k - [U_p]^n \delta \boldsymbol{W}_p^k. \tag{11}
$$

With the LUJACOBI-PRIM scheme, each cell can be treated independently, which makes it very well suited for GPUs. Another parallel variant of the LUSGS-PRIM smoother can be written as

$$
\begin{aligned}
[D_p]^n \delta \boldsymbol{W}_p^{k+1/2} &= -\left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right] \boldsymbol{R}(\boldsymbol{W}^n) - [L_p]^n \delta \boldsymbol{W}_p^k + [U_p]^n \delta \boldsymbol{W}_p^{k-1/2}, \\
[D_p]^n \delta \boldsymbol{W}_p^{k+1} &= -\left[\frac{\partial \boldsymbol{W}_p}{\partial \boldsymbol{W}_c}\right] \boldsymbol{R}(\boldsymbol{W}^n) - [U_p]^n \delta \boldsymbol{W}_p^{k+1/2} + [L_p]^n \delta \boldsymbol{W}_p^k.
\end{aligned} \tag{12}
$$

Here, the contributions of lower and upper flux Jacobian are taken from the previous symmetric sweep. The smoother described by equation (12) will be referred to as Lagged-LUSGS-PRIM. With this smoother, dependencies within the forward and backward sweeps are removed, hence each step is data parallel.

Despite the fact that the LUJACOBI-PRIM and Lagged-LUSGS-PRIM smoothers are parallel, they are considered weak smoothers compared with LUSGS-PRIM. Another parallel variant of the LUSGS-PRIM smoother can be obtained by updating the cells following the red-black check board pattern shown in Figure 1-a, instead of following the natural numbering of the cells. This pattern allows the computation of $\partial \boldsymbol{W}_p$ in parallel for all cells of the same color using Jacobi iterations. This smoother will be referred as RB-PRIM.



(a) RB-PRIM (b) RB-LINE-K-PRIM (c) RB-PLANE-JK-PRIM

Figure 1: Various types of Red-Black schemes.

The native implementation of red-black Gauss Seidel (Figure 1-a), however, suffers from the poor

memory access pattern for GPU architectures. Indeed, 32 consecutive threads in a warp access the data with a stride of two for both load and store operations and which leads to a waste of the available bandwidth.

Several optimization techniques have been suggested to improve the access patterns of red-black Gauss-Seidel iterations including the reordered pattern combined with the reuse data via the shared memory or read only texture memory [16]. However, these techniques were not considered since they would have negatively impacted the performance of other functions in FANSC. In this study, the data access pattern is improved by coloring the grid lines and the grid planes following the red-black pattern, as shown in Figure 1b) and 1c). As for the traditional check-board red-black GS smoother, all cells of the same color are updated simultaneously. These smoothers, although potentially weaker than the checkboard red-black Gauss-Seidel, have more efficient memory access patterns. They will be referred to as the red-black line Gauss-Seidel smoother based on primitive variables (RB-LINE-PRIM) and the red-black plane Gauss-Seidel smoother based primitive variables (RB-PLANE-PRIM).

The efficiency of RB-LINE-PRIM and RB-PLANE-PRIM smoothers is highly dependent on the orientation of the lines and the planes. The RB-LINE-I-PRIM smoother corresponds to the case where the red and black lines are aligned with the i-direction, the RB-LINE-J-PRIM smoother when lines are aligned with the j-direction and the RB-LINE-K-PRIM smoother when lines are aligned with the k-direction. Similarly, the RB-PLANE-JK-PRIM smoother corresponds to the case where the red and black planes are the jk-planes, the RB-PLANE-IK-PRIM smoother where the planes are the ik-planes and the RB-PLANE-IJ-PRIM smoother where the planes are the ij-planes. Figure 1b) illustrates the cell coloring for the RB-LINE-K-PRIM smoother in which the red-black lines follow the k direction. Figure 1(c) shows the RB-PLANE-JK-PRIM in which the odd and even jk-planes (i = const) are colored in red and black. In FANSC, the data is stored following the natural numbering of the cells, hence only the RB-LINE-I-PRIM and RB-PLANE-IJ-PRIM have efficient memory access patterns. An efficient memory access pattern can be achieved with the other line and plane smoothers by swapping the i-j-k indices prior to running the case. The impact of the choice of i-j-k directions on the convergence of the RANS solver will be revisited in section 5.

Table 1 summarizes the smoothers that will be studied for the CPU and GPU codes. Although there are many more linear solvers that can be used to solve the linear system in equation (2), the ones presented here are simpler to implement and as shown in the next sections, can be effective smoother in a non-linear multigrid algorithm for the RANS equations.

Table 1. Implicit smoothers for CPU and GPU codes

CPU code	GPU code
1. LUSGS-PRIM	1. LUJACOBI-PRIM
2. LUSGS	2. Lagged LUSGS-PRIM
3. LUJACOBI-PRIM	3. RB-LINE-PRIM
4. Lagged LUSGS-PRIM	4. RB-PLANE-PRIM
	5. RB-PRIM

4 GPU Implementation

FANSC is parallelized on either multi-core CPUs or GPGPU. For the CPU code, the coarse-grain parallelism is implemented using the Message Passing Interface (MPI), and the implementation details are discussed in reference [17]. The CUDA programming language is used to implement the GPU version of FANSC. The simulations presented in this study were run on the Tesla K20 GPU that comprises a single chip GK110 with 13 streaming multi-processors (SMs) where each SM features 192 CUDA cores for a total of 2496 CUDA cores. The Tesla K20 GPU has a peak double precision floating point performance of 1.17 Tflops and a peak memory bandwidth of 208 Gb/s with ECC enabled. The CPU simulations were run on compute nodes each equipped with 2 Xeon E5-2670 v1 CPUs. The peak memory bandwidth per compute node for the CPUs is 102.4 GB/s. A summary of the hardware specifications for the CPUs and GPUs used in this study is presented in Table 2.

Table 2. Summary of the hardware specifications for the CPUs and GPUs used in this study

	Xeon E5-2670 CPU	Kepler K20 GPU
Number of CPUs/GPUs per computing node	2	2
Performance per CPU/GPU	2.6GHz, 8 cores, 16 threads	0.706 GHz, 13 SMs, 192 CUDA cores/SM
Theoretical peak performance per node	332.8 GFlop/s (DP)	2240 GFlop/s (DP)
Peak memory bandwidth per CPU/GPU	51.2 Gb/s	208 Gb/s (ECC off)
Peak memory bandwidth per node	102.4 Gb/s	416 Gb/s (ECC off)
Power per CPU/GPU	Thermal design power: 115 W	Board power: 225 W Idle power: 25 W
Released date	Q1'2012	Q4'2012

For the GPU code, two levels of parallelism are employed: fine-grained data parallelism for the computation within a grid block and coarse-grained parallelism for computation across multiple grid blocks. Fine-grained parallelism is implemented by associating one CUDA thread per cell, which implies that one thread typically needs to update five variables (the five states in vector $\boldsymbol{W}_c$ or $\boldsymbol{W}_p$). This approach creates more instruction-level parallelism and therefore more concurrent arithmetic operators and memory access in-flight per thread but it leads to a higher number of registers used per thread. For the coarse-grained parallelism, one asynchronous CUDA stream is used per block in the grid, which allows to overlap computations across multiple grid blocks, and therefore increase occupancy.

In FANSC, all data is stored using the Structures of Arrays (SoA) data structure to allow efficient coalesced memory accesses. Padding between the five variable fields ensures that the global memory fetches are fully aligned and coalesced, as shown in Figure 2.

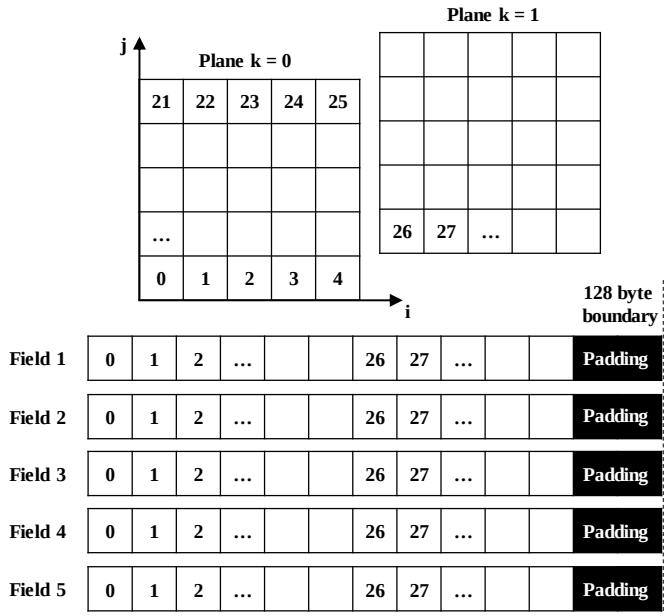


Figure 2: FANSC data storage layout in GPU global memory within one grid block.

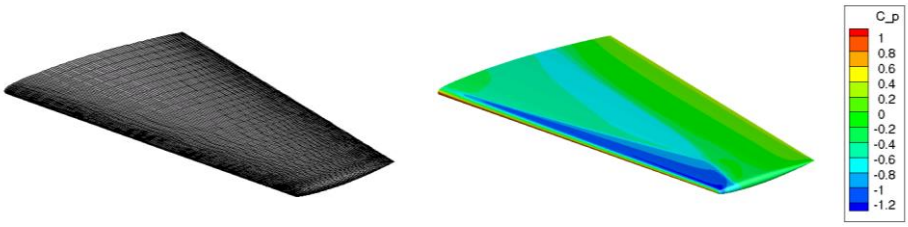
5 Results

5.1 Impact of grid block orientations on red-black line and plane smoothers

In this section, the efficiency of the RB-LINE-PRIM and RB-PLANE-PRIM smoothers is studied on the GPU. More specifically, the impact of coloring the lines or the planes along different i,j,k directions

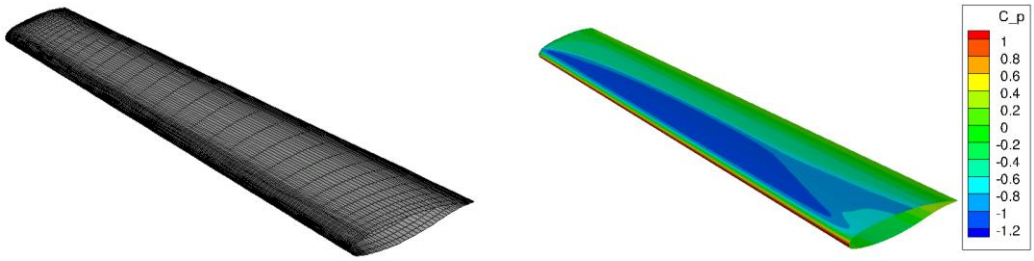
is investigated since it is known [18, 19] that the efficiency of those smoothers depends on the orientation of the colorings relative to the flow direction. Furthermore, since the memory access pattern is better for line colorings in the i-direction and plane colorings along i-j planes, the orientation of the i, j, and k directions are permuted to find the smoother with not only the best convergence rate but also with the lowest time per iteration. The most efficient smoothers from this section will be used as the representative smoothers for RB-LINE-PRIM and RB-PLANE-PRIM smoothers in section 5.2.

For both the RB-LINE-PRIM and RB-PLANE-PRIM smoothers, there are 3 possible colorings of the lines and planes respectively, leading to the following 6 smoothers: RB-LINE-I-PRIM, RB-LINE-J-PRIM, RB-LINE-K-PRIM, RB-PLANE-JK-PRIM, RB-PLANE-IK-PRIM and RB-PLANE-IJ-PRIM. The studies are carried out on two different test cases. The first case is the transonic flow over the Onera-M6 wing, at Mach 0.84, angle of attack 3.06 and Reynolds number 11.72×10^6. The second case is the transonic flow over the DPW-W1 wing, at Mach 0.76, angle of attack 0.61 and Reynolds number 5×10^6. A non-linear multigrid scheme with 3 levels is used to solve the RANS equations. The simulation is stopped when the L2-norm of the density residual has dropped by 6 orders of magnitude. The grid over the Onera-M6 geometry contains 1.3M cells on the finest level (level 0), 161K cells on the middle level (level 1) and 20K cells on the coarse level (level 2). The mesh was partitioned in 32 blocks for both the CPU and GPU runs to obtain a perfect load balancing. The surface mesh on the ONERA-M6 geometry is shown in Figure 3, along with the pressure coefficient contours of the converged solution. The grid over the DPW-W1 contains 2.3M cells on the finest level, 288K cells on the middle level and 36K cells on the coarsest level. The DPW-W1 mesh was portioned in 72 blocks to obtain a load balancing of 1.04 on 16 cores. The surface mesh on the DPW-W1 geometry is shown in Figure 4, along with the pressure coefficient contours of the converged solution. The number of iterations for the smoothers was set to 8, 8 and 10 on the finest level, the middle level and the coarsest level, respectively. These numbers are found to guarantee the convergence to steady solutions for all 6 implicit smoothers on both ONERA-M6 and DPW-W1 simulations.



Structured grid of OMERA-M6 C_p distribution

Figure 3: ONERA-M6 wing geometry, structure grid and pressure coefficient contours.



Structured grid of DPW-W1 C_p distribution

Figure 4: DPW-W1 wing geometry, structure grid and pressure coefficient contours.

Figure 5 shows the total time and total number of nonlinear MG iterations required to reach the steady solution along with time per MG cycle for the ONERA-M6 and DPW-W1 geometries. The unpermuted orientation of the i,j,k axes follows the convention in Figure 1: the spanwise direction is assigned to the *k* axis, the chordwise direction is assigned to the *i* axis, and the direction normal to the wing surface is set to the *j* axis. Figure 5-(a),(b) show that RB-LINE-K-PRIM and RB-PLANE-IK-PRIM require the lowest number of non-linear MG iterations to reduce the residual by 6 orders of magnitude for both ONERA-M6 and DPW-W1 cases but unfortunately, those smoothers do not have the lowest time per iteration, as shown in Figure 5-(e),(f). This is explained by the fact that the RB-LINE-K-PRIM and RB-

PLANE-IK-PRIM smoothers have inefficient uncoalesced memory access patterns due to the data layout used in FANSC (i-direction first, j-direction second, k-direction third). On the other hand, the RB-LINE-I and RB-PLANE-IJ smoothers have fully coalesced memory access patterns, which explain why they have the lowest time per iteration.

Fortunately, it is possible to permute the orientation of the i,j,k axes to improve the memory access patterns of the RB-LINE-K and RB-PLANE-IK-PRIM smoothers. By permuting the (i, j, k) orientations to (k, i, j), it is possible to obtain smoothers with the best convergence rate and good memory access patterns. Those line and plane smoothers are referred to as RB-LINE-I-SWAP-K2I and RB-PLANE-IJ-SWAP-J2K. Figure 5-(c)-(d) shows that those smoothers have the best total run time. For the remainder of this paper, they will simply be referred to as RB-LINE and RB-PLANE smoothers.

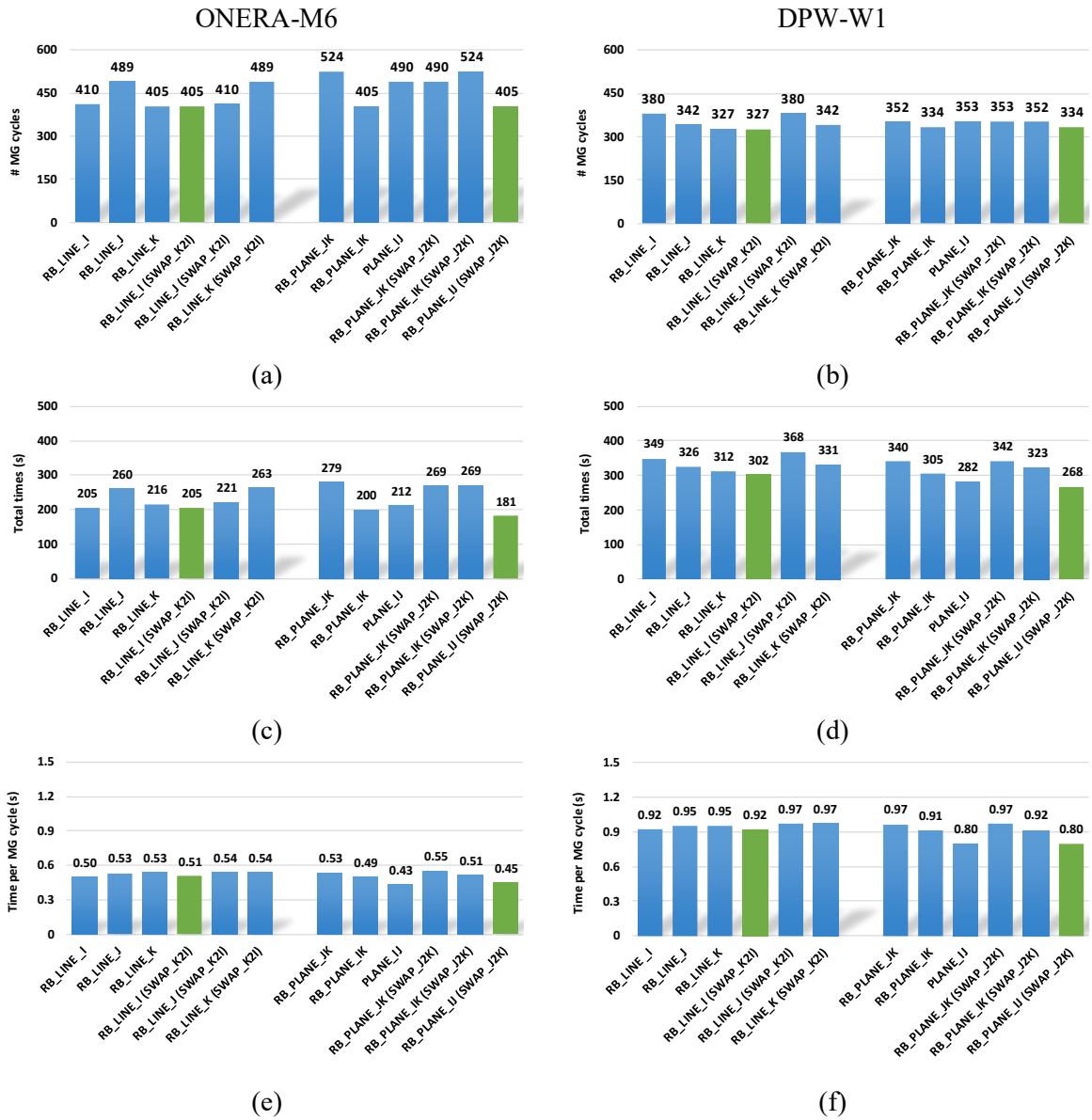


Figure 5: The effect of multiblock grid orientation on the GPU solver performance for ONERA-M6 and DPW-W1 wing geometries with 8, 8 and 10 sweeps for finest, coarser and coarsest grid, (a), (b): Number of MG cycle; (c), (d): Total simulation time; (e), (f): Computing time per MG cycle.

5.2 Parametric studies to identify the optimal implicit smoothers for the CPU and GPU codes

Parametric studies are now carried out on the ONERA-M6 and DPW-W1 test cases presented in Section 5.1. Various combinations of implicit smoothers and number of sweeps per MG level are studied to identify the combination that leads to the lowest simulation time with the CPU and GPU codes. The different type of implicit smoothers used for the CPU and GPU codes for the parameter study are shown in Table 1. The implicit smoothers used with the CPU code consist of the LUJACOBI-PRIM, Lagged LUSGS-PRIM, LUSGS-PRIM and LUSGS smoothers. For the GPU code, the LUJACOBI-PRIM, Lagged LUSGS-PRIM, RB-LINE-PRIM, RB-PLANE-PRIM and RB-PRIM smoothers are used. As mentioned in Section 5.1, the RB-LINE-PRIM and RB-PLANE-PRIM smoothers correspond to the RB-LINE-I-PRIM and RB-PLANE-IJ-PRIM on a swapped (i, j, k) to (k, i, j) grid.

Three levels of nonlinear MG are used to solve the RANS by reducing the L2-norm of the density residual by 6 orders of magnitude. The number of sweeps on each multigrid level is varied as shown in Figure 6. On the finest level (level 0) and middle level (level 1), seven different numbers of sweeps are considered, varying from 2 to 14 with interval of 2. On the coarsest level (level 2), seven different numbers of sweeps are considered, varying from 10 to 40 with interval of 5. The number of sweeps on the coarsest level is higher since the efficiency of multigrid schemes can be greatly improved by having an accurate coarsest grid solution. To limit the number of cases to run, the number of sweeps on the coarse levels (levels 1 and 2) is restricted to be higher than the number of sweeps on a finer level. The total number of combinations of sweeps per implicit smoother is thus 183. Since a total of 9 different smoothers are studied, a total of 1647 RANS simulations were performed.

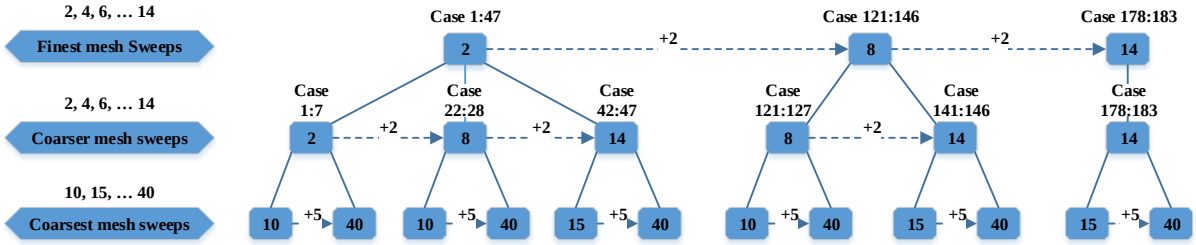


Figure 6: Number of sweeps per MG level settings for parameter study to obtain the optimal implicit smoother for CPU and GPU code.

For the CPU code, the solution is initialized by first performing 50 iterations on the coarsest level only followed by 50 iterations on the two coarsest levels (levels 1 and 2). It has been found that this strategy leads to lower overall run times with the CPU code. On the GPU code, no such initialization is performed since it was not found to improve the overall run times. This is explained by the fact that performance of the GPU code is lower on coarse levels. Figure 7 shows the speedup between 1 GPU and 2 CPUs for one iteration on the different MG levels when using the lagged LUSGS-PRIM solver with 8-8-10 sweeps. It is seen that on the coarser levels, the GPU loses some of its advantage over the CPUs, because of the limited parallelism available.

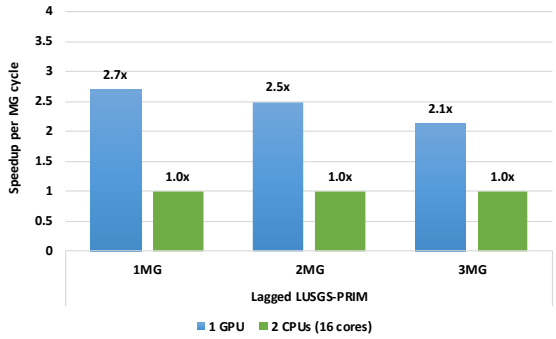
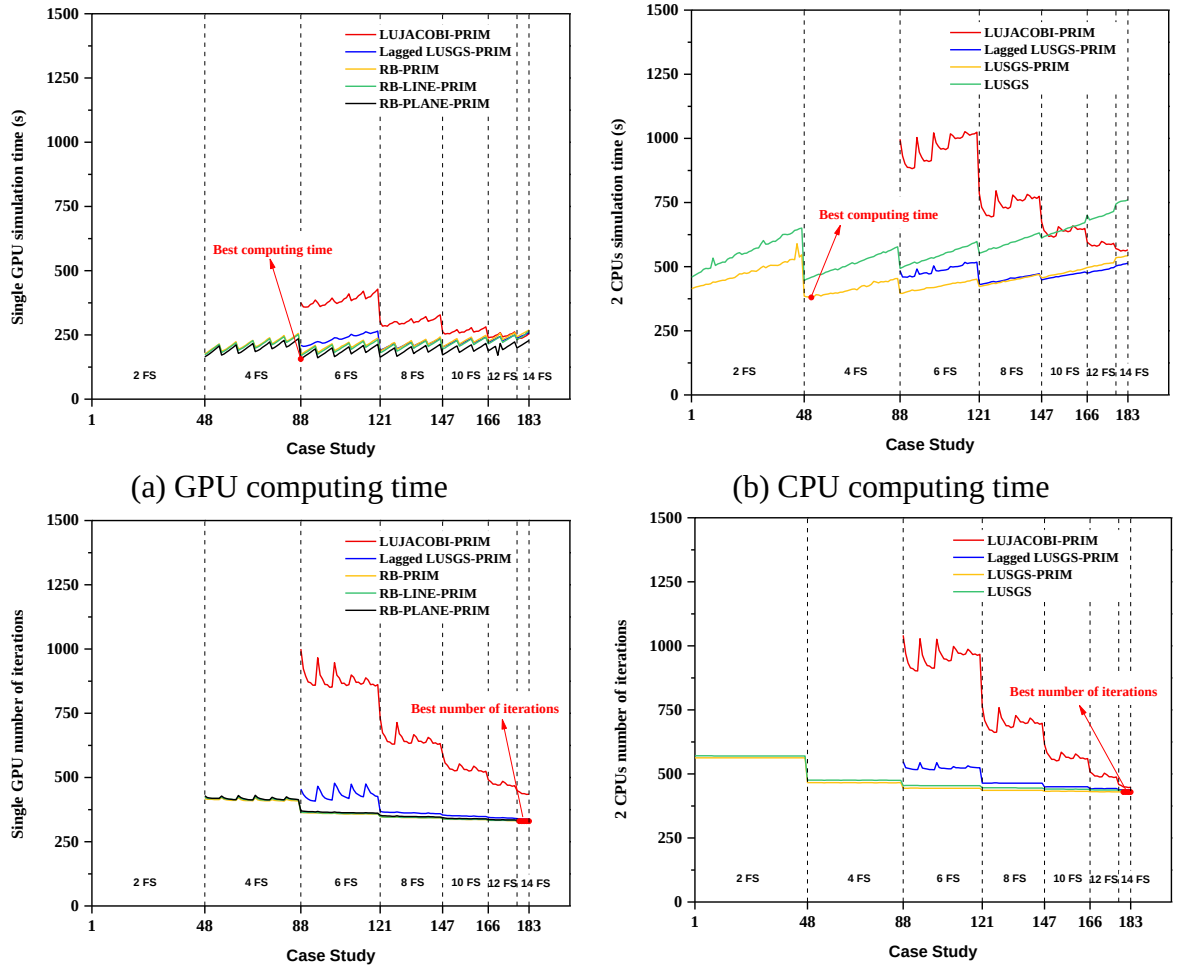


Figure 7: Speedup of 1 GPU versus 2 CPUs for one iteration on the different MG levels. The same solver configurations with 8, 8 and 10 sweeps for finest, coarser and coarsest grid were used on the DPW-W1 wing.

It should be mentioned that a similar parametric study was performed using only 1 and 2 MG levels,

but it was found that the run times were higher than when using 3 MG levels, for both the CPU and GPU code.

The GPU results are obtained using a single Kepler K20 while the CPU results are obtained using the 16 cores from two Xeon E5-2670 CPUs. The simulation times and the total number of iterations required to reduce the residual by six orders of magnitude using different combinations of the number of sweeps and implicit smoothers for the ONERA-M6 and DPW-W1 wings are shown in Figure 8 and Figure 9, respectively. Each point in those figures corresponds to a fully converged RANS simulation with FANSC. Some combinations of number of sweeps and implicit smoother did not converge, which can be expected when using a non-linear multigrid scheme with insufficient smoothing on some levels. For example, in Figure 8-(a),(b), using the LUJACOBI-PRIM and Lagged LUSGS-PRIM smoothers leads to divergence when using less than 6 sweeps on the finest level.

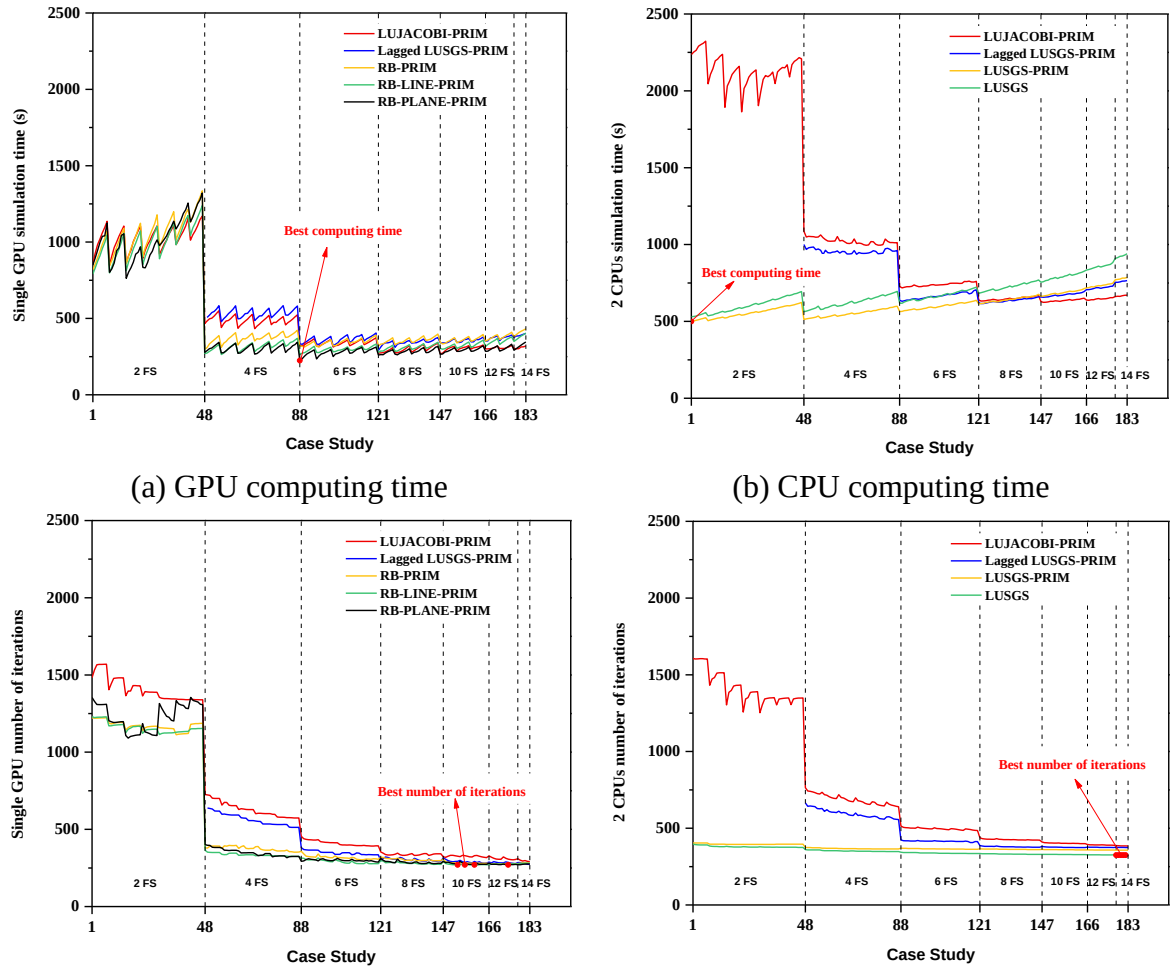


(a) GPU computing time (b) CPU computing time

(c) GPU number of MG cycles (d) CPU number of MG cycles

Figure 8: Simulation time and number of MG cycles for different combinations of implicit smoothers and sweeps for ONERA-M6 wing.

As expected, increasing the number of sweeps on each MG level reduces the number of iterations required to converge to the steady-state solution however, using more sweeps on each MG level also increases the time per iteration. For each type of implicit smoother, there exists an optimal number of sweeps on each level that leads to a good compromise between the number of iterations and time per iteration. In general, the weaker smoothers such as LUJACOBI-PRIM and Lagged LUSGS-PRIM provide good performance when using a large number of sweeps. This can be seen for both CPU and GPU codes. Interestingly, the performance of LUJACOBI-PRIM and Lagged LUSGS-PRIM with a high number of sweeps even surpasses the performance of the other stronger smoothers with a lower number of sweeps.



(a) GPU computing time (b) CPU computing time

(c) GPU number of MG cycles (d) CPU number of MG cycles

Figure 9: Simulation time for different combinations of implicit smoothers and sweeps (DPW-W1).

For the CPU code, when using the stronger smoothers such as LUSGS and LUSGS-PRIM, increasing the number of sweeps on the finest grid show mild or even no noticeable improvement on the convergence (Figure 8 (d) and Figure 9 (d)). It can be hypothesized that solving the linear system in Equation (2) to a tighter tolerance is not required to obtain good convergence with the inexact Newton method. This mild improvement in convergence rate does not compensate for the higher time per iteration due to the larger number of sweeps on each MG cycles.

Figure 8(b) and Figure 9(b) also demonstrate the advantage of using the primitive form of the flux Jacobian matrix instead of the conservative form. For the 183 study cases, the LUSGS-PRIM outperforms the LUSGS in term of computing time whereas the number of MG cycles are comparable, as expected. For the GPU code, the smoothers RB-LINE-PRIM and RB-PLANE-PRIM which are derivatives of the RB-PRIM smoother show comparable convergence to the RB-PRIM smoother even though they are considered as weaker smoothers. In addition, due to their fully coalesced memory access patterns, the RB-LINE-PRIM and RB-PLANE-PRIM smoothers outperform the RB-PRIM smoother in term of computing time. Recall that in section 5.1., the orientation of the i,j,k axes was chosen to obtain good convergence rates as well as good memory access patterns.

The combinations of smoother and number of sweeps that led to the lowest run times are shown in Table 3 and Table 4 for the CPU and GPU codes respectively. For the ONERA-M6 wing simulation, the best scheme on the GPU is the RB-PLANE-IJ smoother with 6-6-10 sweeps which requires 156.36s while the best scheme on the CPU is the LUSGS-PRIM smoother with 4-4-25 sweeps which requires 380.04s. Thus, for the ONERA-M6 case, the single GPU code is 2.43x faster than the two-CPU code at their optimal settings. For the DPW-W1 wing simulation, the best scheme on the GPU is the RB-PLANE-IJ smoother with 6-6-10 sweeps which requires 224.49s while the best scheme on the CPU is

the LUSGS-PRIM smoother with 2-2-10 sweeps which requires 499.6s. Therefore, the single GPU code is 2.23x faster than the two-CPU code at their optimal settings for the DPW-W1 wing.

The obtained speedup when using 1 GPU relative to 2-CPU is comparable to the ratio of maximum memory bandwidth between the Tesla K20 (208 Gb/s with ECC off) and two Xeon E5-2670 CPUs (102.4 Gb/s). Since the performance of FANSC is mostly limited by the memory bandwidth, these results are in-line with expectations.

Table 3. Best computing time and corresponding scheme settings for CPU and GPU codes (ONERA-M6)

	Implicit smoothers	Initial MG cycles	Ramping MG cycles	Number of sweeps on 3 MG levels	Time (s)
CPU	LUJACOBI-PRIM	0	449	14-14-25	560.55
	Lagged LUSGS-PRIM	100	364	8-8-10	430.35
	LUSGS	100	376	4-4-10	446.60
	LUSGS-PRIM	**100**	**366**	**4-4-25**	**380.04**
GPU	LUJACOBI-PRIM	0	449	14-14-15	233.56
	Lagged LUSGS-PRIM	0	369	8-8-10	190.40
	RB-PRIM	0	364	6-6-10	174.84
	RB-LINE-I-PRIM	0	365	6-6-10	168.93
	RB-PLANE-IJ-PRIM	**0**	**370**	**6-6-10**	**156.36**

Table 4. Best computing time and corresponding scheme settings for CPU and GPU codes (DPW-W1)

	Implicit smoothers	Initial MG cycles	Ramping MG cycles	Number of sweeps on 3 MG levels	Time (s)
CPU	LUJACOBI-PRIM	100	306	10-10-15	624.29
	Lagged LUSGS-PRIM	100	282	8-8-20	615.94
	LUSGS	100	296	2-2-10	530.84
	LUSGS-PRIM	**100**	**304**	**2-2-10**	**499.60**
GPU	LUJACOBI-PRIM	0	322	10-10-10	267.83
	Lagged LUSGS-PRIM	0	298	8-8-10	290.50
	RB-PRIM	0	380	4-4-10	297.55
	RB-LINE-I-PRIM	0	314	6-6-10	262.83
	RB-PLANE-IJ-PRIM	**0**	**293**	**6-6-10**	**224.49**

Conclusion

In this study, Bombardier's in-house CFD code FANSC was accelerated with GPGPUs. Non-linear multigrid smoothers that are efficient numerically while being well suited for massively parallel hardware architectures are identified. The results indicate that numerically weaker smoothers such as red-black line and plane smoothers can be very effective when an appropriate number of sweeps are

performed on each multigrid level. When both the CPU and GPU codes are used with their best respective solvers, speedups of 2.4x can be achieved on a single Tesla K20 GPU compared to two 8-core Xeon 2670 CPUs.

Acknowledgements

This work is supported by the National Science and Engineering Research Council (NSERC) of Canada. Computations were made on the supercomputer Guillimin from McGill University, managed by Calcul Québec and Compute Canada. The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), the ministère de l'Économie, de la science et de l'innovation du Québec (MESI) and the Fonds de recherche du Québec - Nature et technologies (FRQ-NT).

References

[1] Castonguay, P., et al. *On the development of a high-order, multi-GPU enabled, compressible viscous flow solver for mixed unstructured grids*. in *20th AIAA Computational Fluid Dynamics Conference*. 2011.

[2] Xu, C., et al. *Parallelizing a High-Order CFD Software for 3D, Multi-block, Structural Grids on the TianHe-1A Supercomputer*. 2013. Berlin, Heidelberg: Springer Berlin Heidelberg.

[3] DeVito, Z., et al. *Liszt: A domain specific language for building portable mesh-based PDE solvers*. in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2011.

[4] Kampolis, I.C., et al., *CFD-based analysis and two-level aerodynamic optimization on graphics processing units.* Computer Methods in Applied Mechanics and Engineering, 2010. **199**(9): p. 712-722.

[5] Jacobsen, D., J. Thibault, and I. Senocak. *An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters*. in *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. 2010.

[6] Mostafazadeh Davani, B., et al. *Unsteady Navier-Stokes Computations on GPU Architectures*. in *23rd AIAA Computational Fluid Dynamics Conference*. 2017.

[7] Lefebvre, M., et al., *Optimizing 2D and 3D structured Euler CFD solvers on Graphical Processing Units.* Computers & Fluids, 2012. **70**: p. 136-147.

[8] Brandvik, T. and G. Pullan, *An Accelerated 3D Navier–Stokes Solver for Flows in Turbomachines.* Journal of Turbomachinery, 2010. **133**(2): p. 021025-021025-9.

[9] Cagnone, J.S., et al., *Implicit multigrid schemes for challenging aerodynamic simulations on block-structured grids.* Computers & Fluids, 2011. **44**(1): p. 314-327.

[10] Laurendeau, E., Z. Zhu, and F. Mokhtarian. *Development of the FANSC Full Aircraft Navier-Stokes Code*. in *Proceedings of the 46th Annual Conference of the Canadian Aeronautics and Space Institute, Montreal*. 1999.

[11] Swanson, R.C. and E. Turkel, *Multistage Schemes With Multigrid for Euler and Navier-Stokes Equations*. 1997, NASA Langley Technical Report Server.

[12] Jameson, A., *Multigrid algorithms for compressible flow calculations*, in *Multigrid Methods II: Proceedings of the 2nd European Conference on Multigrid Methods held at Cologne, October 1–4, 1985*, W. Hackbusch and U. Trottenberg, Editors. 1986, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 166-201.

[13] Spalart, P. and S. Allmaras. *A one-equation turbulence model for aerodynamic flows*. in *30th aerospace sciences meeting and exhibit*. 1992.

[14] Jameson, A. and S. Yoon, *Lower-upper implicit schemes with multiple grids for the Euler equations.* AIAA Journal, 1987. **25**(7): p. 929-935.

[15] Rossow, C.C., *Convergence Acceleration for Solving the Compressible Navier-Stokes Equations.* AIAA Journal, 2006. **44**(2): p. 345-352.

[16] Cotronis, Y., E. Konstantinidis, and N.M. Missirlis, *A GPU Implementation for Solving the Convection Diffusion Equation Using the Local Modified SOR Method*, in *Numerical Computations with GPUs*, V. Kindratenko, Editor. 2014, Springer International Publishing:

Cham. p. 207-221.
[17] Sermeus, K., E. Laurendeau, and F. Parpia. *Parallelization and performance optimization of Bombardier multiblock structured Navier-Stokes solver on IBM eserver Cluster 1600*. in *45th AIAA Aerospace Sciences Meeting and Exhibit*. 2007.
[18] Elman, H.C. and M.P. Chernesky. *Ordering Effects on Relaxation Methods Applied to the Discrete Convection-Diffusion Equation*. 1994. New York, NY: Springer New York.
[19] Elman, H.C. and M.P. Chernesky, *Ordering Effects on Relaxation Methods Applied to the Discrete One-Dimensional Convection-Diffusion Equation.* SIAM Journal on Numerical Analysis, 1993. **30**(5): p. 1268-1290.