Proceedings of ASME Turbo Expo 2011
GT2011
June 6-10, 2011, Vancouver, British Columbia, Canada

GT2011-46224

IMPLEMENTATION OF AN EDGE-BASED NAVIER-STOKES SOLVER FOR UNSTRUCTURED GRIDS IN GRAPHICS PROCESSING UNITS

Fernando Gisbert, Roque Corral* and Guillermo Pastor
Technology and Methods Department
Industria de Turbopropulsores S.A.
28830 Madrid, Spain
e-mail: {Fernando.Gisbert,Roque.Corral,Guillermo.Pastor}@itp.es

ABSTRACT

The implementation of an edge-based three-dimensional RANS equations solver for unstructured grids that runs on both central processing units (CPUs) and graphics processing units (GPUs) is presented. This CPU/GPU duality is kept without double-writing the code, reducing programming and maintenance costs. The GPU implementation is based on the standard OpenCL language. The code has been parallelized using MPI. Some turbomachinery benchmark cases are presented. For all cases, an order of magnitude reduction in computational time is achieved when the code is executed on GPUs instead of CPUs.

NOMENCLATURE

CFL	Courant-Friedrich-Lewis number
C_p	Specific heat at constant pressure
C_{pt}	Total pressure coefficient: $(p_{01}-p_{02})/(p_{01}-p_{s2})$
CTA	Centro de Tecnologías Aeronáuticas
C_v	Specific heat at constant volume
E	Total energy
k	Conductivity
LPT	Low Pressure Turbine
$\mathbf{n}$	Edge normal: $[n_x\ n_y\ n_z]$
N_{procs}	Number of MPI processes
$\#ed_i$	Number of edges that share the node i
p	Pressure: $\rho(\gamma-1)\left(E-\frac{1}{2}\mathbf{v}\cdot\mathbf{v}\right)$
P_J	Block-Jacobi preconditioning matrix
$\mathbf{q}$	Heat flux: $-k\nabla T$
RANS	Reynolds Averaged Navier-Stokes
T	Temperature
$\mathbf{U}$	Conservative variables: $[\rho\ \rho u\ \rho v\ \rho w\ \rho E]$
$\mathbf{v}$	Velocity: $[u\ v\ w]$
v_n	Edge normal velocity: $\mathbf{v}\cdot\mathbf{n}$
$\mathbf{x}$	Vertex coordinates: $[x\ y\ z]$
α	Swirl angle
γ	Gas constant: C_p/C_v
μ	Laminar viscosity
ρ	Density
Σ	Control volume boundary surface
σ	Edge area
τ_{ij}	Viscous stress tensor: $\mu(\partial_i v_j+\partial_j v_i)-\frac{2}{3}\mu(\nabla\cdot\mathbf{v})\delta_{ij}$
ϑ	Control volume

INTRODUCTION

The use of GPUs for general purpose computing is becoming increasingly popular because of the outstanding computing performance of GPUs compared with modern CPUs [1], and the popularization of programming languages that ease the GPU programming.

GPUs contain a large number of processor elements, of the order of hundreds, which are connected to the GPU memory through a bus that provides a high bandwidth when transferring data between the memory and the processor. Thus, while the

*Also associate professor at the Department of Engine Propulsion and Fluid Dynamics of the School of Aeronautics, UPM, Madrid

	CPU	GPU
Number of processors	6	448
Bandwidth (Gb/s)	32	144
Cache Size	12 Mb	768 Kb

Table 1: Comparison of representative properties of a modern CPU (Intel Xeon X5680) and a modern GPU (NVIDIA C2050).

amount of data residing in the processor is small, the access to additional data is very fast. In the CPUs, however, the number of processors is not as large as in the GPUs, and the data transfer between memory and processors occurs at a much lower rate. These limitations are partially overcome by the existence of the processor's cache memory that allows a re-use of the already accessed data. Modern GPUs have a very reduced cache memory compared with that of CPUs. A comparison of the characteristics of a modern CPU and a GPU is presented in table 1. These differences in the processor architectures have a direct translation in the programming languages that are suitable for each device. GPU programming has traditionally required the knowledge of specific programming languages with a particular nomenclature that is hard to translate to more general purpose applications. This situation changed substantially with the release of CUDA [1] by NVIDIA in 2006. CUDA is an extension to the C programming language that makes the execution of a general purpose program in an NVIDIA GPU much easier. As a result, the effort to write and execute a code on CPUs or GPUs is now comparable. Since the computing performance of GPUs is roughly one order of magnitude better than that of CPUs, the number of applications developed for GPUs has grown substantially.

Two years after CUDA had been released, another programming language named OpenCL [2] appeared as a joint effort of the industry to provide a standard programming language for heterogeneous multi-processor platforms. Programs written in OpenCL are not only able to be executed on GPUs, but on any multi-processor platform, such as multi-core CPUs, IBM Cell processors, etc. This paradigm offers a clear advantage with respect to CUDA, since different multi-core platforms are supported with exactly the same code in a transparent way. However, its detractors state that an optimal software implementation is actually very hardware-dependent, therefore the use of heterogeneous platforms requires heterogeneous data access and execution patterns that make the code optimized for a chosen platform highly inefficient for the others. Besides, it is also acknowledged that writing the code in CUDA is more direct than writing it in OpenCL, minimizing the time spent in setting up the GPU simulation and allowing the programmer to focus just on the numerical aspects of the code. However, there are wrappers, such as PyOpenCL [3], that make the OpenCL setting-up easier and allow some degree of abstraction of the basic OpenCL platform layer.

There are a number of works describing the GPU implementation of CFD solvers either for structured or unstructured grids. For all cases, a substantial reduction of the computing time was reported when compared to their CPU counterparts. Elsen et al. [4] solved the Euler equations on structured grids, using Brook GPU [5] as a programming language and achieving speed-ups of up to 15 with respect to a single CPU execution. Tölke [6] solved the Lattice-Boltzmann equations on structured grids using CUDA as programming language, obtaining roughly an order of magnitude reduction in the computing time. Jacobsen et al. [7] solved the 3D incompressible Navier-Stokes equations in structured grids on multiple GPUs using CUDA and MPI, achieving also an order of magnitude speed-up with respect to the CPU computational times. Jespersen [8] used CUDA to accelerate parts of a RANS equation solver for structured grids. Brandvik and Pullan [9] implemented a GPU parallel version of a RANS solver for structured grids and reported a speed-up of up to 20 with respect to its CPU counterpart. They used a Python algorithm to translate the solver source code into a number of multi-processors platforms, CUDA among them. Recently Castonguay et al. [10] used multiple GPUs to solve the RANS equations on unstructured grids using high-order discretizations. This type of discretizations require a large amount of computation per node and are well suited to be solved on GPUs obtaining speed-ups of about 70 when compared with the equivalent CPU execution. Corrigan et al. [11, 12] designed an automatic code porting algorithm capable of generating valid CUDA code from Fortran legacy code. The resulting CUDA code could then be executed on GPUs, allowing speed-ups of roughly an order of magnitude compared with a single CPU process.

In this work we present the OpenCL re-implementation of an existing CFD solver [13] in order to allow its execution on one or more multi-core platforms, GPUs included. The existing RANS solver, known as Mu^2s^2T, uses hybrid unstructured grids to discretize the spatial domain and an edge-based data structure to compute the fluxes. A second-order MUSCL scheme conforms the spatial discretization [14], which is marched in time with an explicit five-stage Runge-Kutta [15]. Block-Jacobi preconditioning [16] and multigrid [17] are used to accelerate steady state convergence. Turbulence effects are modeled with the $k-\omega$ model [18]. The solver is parallelized using MPI. The entire code is written in Fortran. The new code, that reproduces all the features of the existing one, has been re-written using a mixture of C++ and OpenCL. This approach has three main advantages:

1. The C++ language provides enough flexibility to naturally integrate the source code written in OpenCL as part of the C++ class that manages the simulation. That, combined with the use of compiler directives, allows the generation of a

unified source code to be executed either on a CPU or on a GPU, avoiding the need to have separate solvers for the CPU and the GPU architectures. This has in turn several positive aspects: the CFD results are ensured to be exactly the same when running on the CPU or on the GPU, and the debugging of the solver can be performed on the CPU side, where there are applications that make this process much easier than on the GPU.

2. The use of compiler directives, which is a standard programming practice, avoids the need to have a script to translate the code from one programming language to another. These scripts often assume a number of ad-hoc coding conventions that make the programming practices unnecessarily rigid. Moreover, the extra programming cost is small and the portions of the source code that are exclusively used in the CPU or the GPU simulations are clearly delimited and naturally removed of the compilation process when the GPU or the CPU versions of the solver are generated.
3. Programming with OpenCL allows the use of several multiprocessor platforms, not only NVIDIA ones, obtaining comparable performances in the cases studied.

The rest of paper is organized as follows: first, the RANS equations and their discretization over an unstructured grid are presented. Next, we discuss the OpenCL implementation of the parts of the solver that require the largest computation effort. Then it is explained how the unified CPU/GPU source code is generated. The performance of the resulting code is analyzed by executing it on several GPU models, obtaining acceptable speedups in all cases when compared to the CPU execution time. Finally, we will present comparisons between the solver results and some experimental measurements that show the degree of accuracy of the code.

RANS SOLVER DESCRIPTION

The RANS equations may be expressed in compact form as:

$$\frac{d}{dt}\int_{\vartheta}\mathbf{U}d\vartheta+\int_{\Sigma}\mathbf{F}\cdot\mathbf{n}d\sigma=0 \tag{1}$$

where $\mathbf{F}=\mathbf{F}_c-\mathbf{F}_v$ represents the sum of convective and viscous fluxes. The expression for the fluxes is:

$$\mathbf{F}_c\cdot\mathbf{n}=\begin{bmatrix}\rho v_n\\ \rho u v_n+p\cdot n_x\\ \rho v v_n+p\cdot n_y\\ \rho w v_n+p\cdot n_z\\ (\rho E+p)v_n\end{bmatrix},\qquad \mathbf{F}_v\cdot\mathbf{n}=\begin{bmatrix}0\\ \tau_{xn}\\ \tau_{yn}\\ \tau_{zn}\\ \mathbf{v}^T\cdot\tau\cdot\mathbf{n}-\mathbf{q}\cdot\mathbf{n}\end{bmatrix} \tag{2}$$

being $\tau_{kn}=\tau_{kx}\cdot n_x+\tau_{ky}\cdot n_y+\tau_{kz}\cdot n_z$, where $k=x,y,z$. The discrete equations are obtained from Eq. (1) using a finite volume formulation, where the control volumes are constructed making use of the dual mesh (see Fig. 1), that is, the mesh that results from connecting the centroids of the cells that surround a node [19]. Thus, the conservative variables are stored in the cell vertex, and the fluxes over the control volume faces are obtained summing up contributions of all control volume faces. The preconditioned semi-discrete version of Eq. (1) for each node i of the mesh may be written as:

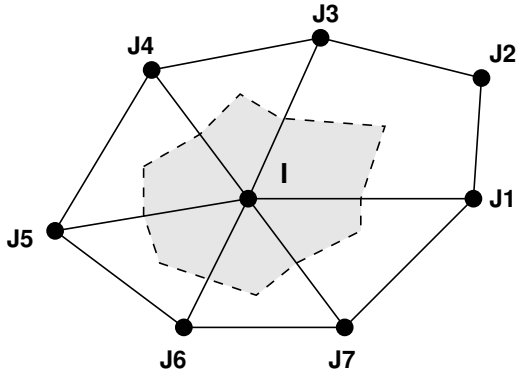


Figure 1: Hybrid cell grid and associated dual mesh.

$$P_{Ji}^{-1}\frac{d\left(\mathbf{U}_i\cdot\vartheta_i\right)}{dt}+\sum_{j=1}^{\#ed_i}\frac{1}{2}\left(\mathbf{F}_i+\mathbf{F}_j\right)\cdot\mathbf{n}_{ij}\sigma_{ij}=0 \tag{3}$$

The contributions of the domain boundary faces are omitted for simplicity. Eq. (2) shows that the gradient of the primitive variables is needed in order to evaluate the viscous fluxes. Besides, the MUSCL scheme also needs the gradient of the flow variables to build the convective fluxes [14]. Therefore, the gradient of the variables,

$$\nabla\mathbf{U}_i=\frac{1}{\vartheta_i}\sum_{j=1}^{\#ed_i}\frac{1}{2}\left(\mathbf{U}_i+\mathbf{U}_j\right)\mathbf{n}_{ij}\sigma_{ij} \tag{4}$$

must be evaluated before the computation of the fluxes is performed. Finally, the formula for the computation of the block-Jacobi preconditioning matrix of Eq. 3 is:

$$P_{Ji}^{-1}=\frac{1}{\vartheta_i}\frac{1}{CFL}\sum_{j=1}^{\#ed_i}\left(\frac{1}{2}\left|A_{ij}\right|+B_{ij}\frac{1}{\left|\mathbf{x}_j-\mathbf{x}_i\right|}\right)\sigma_{ij} \tag{5}$$

where $\left|A_{ij}\right|$ is the absolute value of the inviscid fluxes Jacobian matrix and B_{ij} contains the viscous terms contribution to the preconditioning matrix.

Algorithm 1 Generic edge loop for an edge-based solver. `ReadPointData` represents the point data needed to perform the inner loop computations, `F(data1,data2)` the inner loop operations and `WritePointResult` the writing of the resulting data. `Nedges` is the number of grid edges and `edgeNode` is the edge-node connectivity.

```
void edgeComputations(...)
{
  for(edge=0;edge<Nedges;edge++)
  {
    point1 = edgeNode(1,edge);
    point2 = edgeNode(2,edge);
    data1  = ReadPointData(point1);
    data2  = ReadPointData(point2);
    term   = F(data1,data2);
    WritePointResult(point1,term);
    WritePointResult(point2,term);
  }
}
```

IMPLEMENTATION

All the routines needed to perform a Runge-Kutta iteration are programmed as kernels in order to be executed on a GPU: the block-Jacobi preconditioning matrix computation, the gradient and fluxes evaluation, the boundary conditions and the conservative variables updating. This ensures that there is no need to exchange data between the CPU and the GPU during the execution process, which is an expensive operation that may degrade the code performance when running on a GPU. The only information that has to be communicated from the GPU to the CPU is the data of the domain frontiers when several GPUs are used in parallel, since by the time the code parallelism was implemented there was no way for two GPUs to exchange data without having to rely on the CPU that controls them. NVIDIA has recently released GPUDirect [20], which is a technology that allows direct communication between GPUs, but it is not yet supported on all GPUs and platforms and it has not been used in this work. Anyway when the number of domains is small, as it is the case in the present study, the communication time is much smaller than the computing time and the code parallel performance is not seriously compromised.

Next, we will discuss the implementation details of the most time consuming kernels among those listed above. When the code is executed on a CPU and the time per Runge-Kutta iteration is measured, the computation of the block-Jacobi preconditioning takes 10%, the computation of the gradient 10% and the computation of the fluxes 60%. Together, they represent 80% of the total execution time. Therefore a correct implementation of these operations is crucial in order to obtain a computationally efficient solver in whatever platform.

Algorithm 2 OpenCL kernel version of Algorithm (1).

```
__kernel void edgeComputations(...)
{
  edge   = get_global_id(0);
  point1 = edgeNode(1,edge);
  point2 = edgeNode(2,edge);
  data1  = ReadPointData(point1);
  data2  = ReadPointData(point2);
  term   = F(data1,data2);
  WritePointResult(point1,term);
  WritePointResult(point2,term);
}
```

The CPU coding of a generic edge loop representative of Eqs. (3), (4) or (5) is written in Alg. 1. There is just one single process which is in charge of performing all the loop computations. The option of running multiple processes using OpenMP is not considered, since the parallelization is done explicitly using domain decomposition and running separate processes for each sub-domain, that are communicated using MPI.

Before presenting the multi-processor version of Alg. 1 a brief clarification of the OpenCL nomenclature will help following the discussion below. In OpenCL the functions that are executed on the multi-processor device are called kernels. A call to an OpenCL kernel distributes the execution of the kernel source code in a number of threads called work items. The typical number of work items for a GPU is of the order of thousands. The total number of work items of a given kernel is called the kernel global size. Each work item runs independently of the others, but multi-processors usually take advantage of those situations where a group of threads execute the same instruction at the same time.

Since the edge is the minimal entity of an edge-based solver, it seems a natural choice to assign each work item the computations of a single edge. The equivalent kernel for Alg. 1 is outlined in Alg. 2. Each work item computes an edge whose index is given by the `get_global_id(0)` function that returns, for each item, which is its rank within the total number of threads. When this kernel is executed on the GPU the result is almost certainly wrong. This is due to the fact that two different threads can access the same memory position at the same time. If it is a read access one of the threads must wait for the other to finish, the kernel execution time is increased since the data transfer rate is smaller but the overall result is correct. But if it is a write access the written data are corrupted, and the final result is randomly wrong. For these reasons, the memory contention, i.e., the simultaneous access to the same memory position, must be avoided. That requires reordering the edge loop to prevent a node from appearing twice in the same thread group. Thus the edges are grouped, and the size of these groups depends on GPU fea-

Algorithm 3 Sequence of OpenCL kernel calls when the edge loop is split in several edge groups. `groupEdges` points, for each group, to the first edge index of the group.

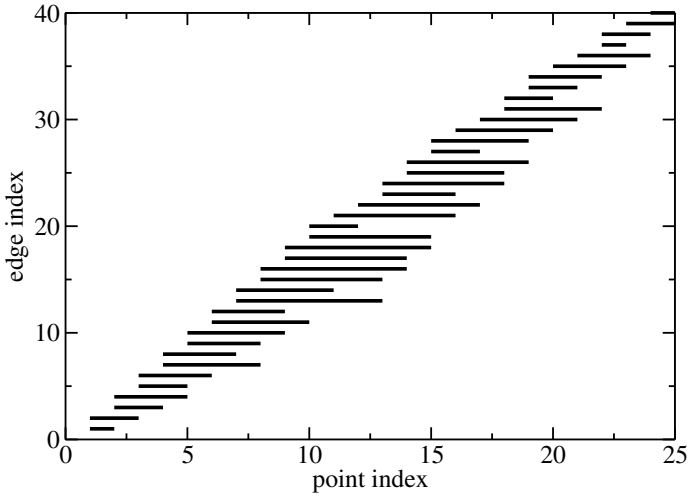
```
...
for(group=0;group<nGroups;group++)
{
  edgeComputations.globalSize =
  groupEdges[group+1] - groupEdges[group];
  edgeComputations(...,group,groupEdges);
}
...
```

Algorithm 4 Modified OpenCL kernel version of Algorithm (2) used when the edge loop has been split into a number of edge groups. Only the lines that are modified are shown. `groupEdges` points, for each group, to the first edge index of the group.

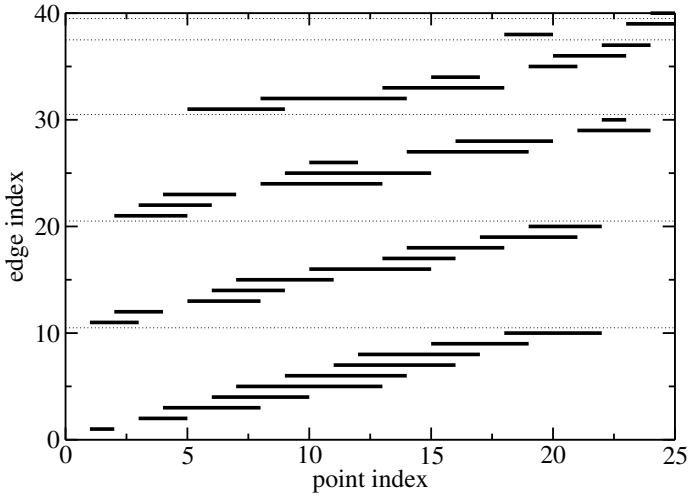
```
__kernel void edgeComputations(...,
                               group,
                               groupEdges)
{
  edge = get_global_id(0)+groupEdges[group];
  ...
}
```

tures such as the number of processors and the maximum number of simultaneous threads per processor. An example of an edge ordering for a simple case with 25 nodes and 40 edges with groups of up to 10 edges is depicted in Fig. 2b. As a result of the grouping process, an additional array, called `groupEdges`, has been created. This array yields, for each group, which is its first edge. Thus, the number of edges of a given group is `groupEdges[group+1]-groupEdges[group]`, and the index of the first edge of the group is `groupEdges[ngroup]`. After the grouping has been performed, we execute as many calls to the OpenCL kernel as edge groups have been found, as specified in Alg. 3. For each kernel, the total number of work items is the number of edges of the group. The OpenCL kernel of Alg. 2 is also slightly modified to make the edges within the group point to the correct global index. The resulting kernel is presented in Alg. 4, where only the lines that change with respect to Alg. 2 are written. Executing Alg. 3 in the GPU produces correct results.

The next question that arises after ensuring that the results are correct is if the kernel implementation is optimal to be executed in whatever computing platform. The limiting factor for all cases is the data transfer rate between the memory and the processors. The faster the transfer of data, the faster the code will perform, since the speed at which the processors can process the data is actually much higher than the speed at which the data enters the processing units. When the grid is structured, the access pattern to the grid data is regular and the compiler knows in advance where to find them. That allows a fast data transfer between memory and processor. For unstructured grids, however, the memory location of the edge nodes is not known a priori by the compiler since the access to the data is controlled by an array of pointers, in our case the `edgeNode` array. This is referred to as indirect addressing in the literature. The access to the memory in these situations is much less efficient and hence some improvements must be introduced to avoid an excessive performance degradation. The strategy may be different depending on whether the processor has cache memory or not.



(a) Reverse Cuthill-McKee ordering to minimize cache-misses.



(b) Reverse Cuthill-McKee followed by an ordering by groups to avoid simultaneous memory access within a group.

Figure 2: Edge and point numbering with different strategies.

In cached-processors like standard modern CPUs, when the executing process requires data from the processor memory, it places these data in the cache. It takes not only the required data but also a block of contiguous data. As these data are in the cache, they can be re-used at no cost. If data outside of this block are required, then they must be taken from the memory again, consuming much more time. This is called a cache miss. Hence, if we renumber the grid nodes and reorder the edges making nearby edges point to nearby points, the number of cache misses is minimized when performing the edge loop of Alg. 1. The reordering algorithm applied in this work is the reverse Cuthill-McKee (RCM) [21]. An example of the resulting edge-node ordering when this technique is applied to the same case with 25 nodes and 40 edges presented above is depicted in Fig. 2a.

In processors without or with small-sized cache memory, such as modern GPUs, this reordering technique is of little help. As stated in the introduction, GPUs base his superior performance in the higher values of the data transfer rate between the memory and the processor elements. But the conditions for achieving such rate are usually very stringent, and certainly hard to meet if indirect memory addressing is used. For instance, NVIDIA GPUs' memory bandwidth is greatly degraded if a memory access pattern such as `vec[stride*point+variable]` is used with `stride>1`, and the access pattern `vec[point+variable*numberOfPoints]` is preferred instead. However, when the data is accessed indirectly, the OpenCL vector types, such as `float4`, `float8`, etc. provide higher bandwidth, hence all the indirect accesses are performed using these vector types whenever possible. Nevertheless, it continues to be crucial to minimize as much as possible the amount of data transferred from the global GPU memory to the local on-chip memory, performing as many operations as possible with variables that physically reside on the local memory.

Therefore, the parameter that influences the performance the most is the relation between the number of floating point operations (FLOP) and the number of indirect reads or writes. Roughly speaking, the larger the number of FLOPS per indirect addressing, the better the performance and thus the greater the benefit expected when porting the code execution from a CPU to a GPU. That is why the CFD codes that employ high order discontinuous Galerkin discretizations [10], which require performing many FLOPS per grid node, have reported the largest speed-ups when comparing GPU and CPU execution times. But in codes where the amount of computation per cell is not as high, like the one we are presenting here, the speed-up can be seriously compromised if we do not pay attention to this issue. An excellent review of the techniques employed to minimize the number of indirect addressings in edge-based solvers can be found in [22]. In order to better understand the importance of this optimization we present here two limit cases: the gradient loop and the flux loop.

Algorithm 5 OpenCL kernel for computing the fluxes looping over each node's neighbors.

```
__kernel void nodeComputations(...)
{
  node = get_global_id(0);
  data1 = ReadPointData(node);
  totalTerm = 0;
  for(neigh=0;neigh<neighbors;neigh++)
  {
    point2 = neighbor(neigh);
    data2  = ReadPointData(point2);
    term   = F(data1,data2);
    totalTerm = totalTerm + term;
  }
  WritePointResult(node,totalTerm)
}
```

Gradient evaluation

When the gradient evaluation of Eq. (4) is programmed according to the code presented in Alg. 2, the number of indirect addressings per edge is 6, two for reading the variables, two for reading the gradient and two for updating it. However, the number of operations performed inside the loop is very small, hence the performance of the loop is completely controlled by the memory access. One simple way of reducing the number of indirect memory accesses is presented in Alg. 5. In this case, the loop is performed over grid nodes, and for each node, an inner loop over all the edges that surround it is executed. The number of indirect addressings per edge in this case has been reduced to one, for reading the variables of the neighbor node. Since the total number of edges is now doubled (each edge is processed twice, once per conforming node), the total number of indirect addressings has been divided by three. When the loop is executed as an OpenCL kernel, the number of work items is the number of grid nodes. For each node we compute the contributions of the edges that share it, storing just the final result and not the intermediate ones as it was done in Alg. 2.

To measure the performance of Alg. 2 and Alg. 5, both kernels have been implemented using OpenCL and executed in a Tesla C1060 GPU. The original edge loop of Alg. 1 has also been run in an Intel Core2 P8600 CPU @ 2.4GHz. Thus, when the Alg. 2 kernel is executed in the GPU, a speed-up of 4.5 is obtained with respect to the edge loop executed in the CPU. If the modified Alg. 5 kernel is executed in the same GPU, the speed-up is 14. The increase in speed-up is 3.01, which agrees well with the reduction in the number of indirect addressings.

Convective and viscous fluxes evaluation

Although the evaluation procedure of the convective and viscous fluxes is conceptually analogous to that of the gradient, the situation is different because the number of FLOPS per indirect addressing is much higher. Even though it has been shown that the number of indirect addressings can be reduced by a factor of three using the modified loop, it must be noted that each edge is processed twice, hence the number of FLOPS of the modified loop is twice as large as that of the original edge loop. This fact may counter-balance the positive effect of reducing the number of indirect addressings. Thus, in the case of the fluxes evaluation, if the Alg. 2 kernel is executed in the same GPU as before, a speed-up of 19 is obtained with respect to the execution time of the Alg. 1 loop in the CPU. However, if the modified Alg. 5 kernel is used, the speed-up is reduced to 15.

If the kernel total execution time is split in the time spent accessing and writing the data (t_{mem}) and the time spent doing operations (t_{op}), the total execution time for those kernels written following Alg. 2 is

$$t_E = t_{mem} + t_{op}$$

while the execution time for those others that have been written like Alg. 5 is:

$$t_N = \frac{t_{mem}}{3} + 2t_{op}$$

These relations allow us to quantify both t_{mem} and t_{op}. It also shows that the Alg. 2 kernels will perform better than the Alg. 5 ones as long as $t_{mem} \leq 1.5t_{op}$.

COMBINED C++/OPENCL PROGRAMMING

There are a number of questions that must be addressed if the CPU and OpenCL versions of a solver have to be written in parallel. We will highlight the main obstacles found, which can be summarized in two main subjects: using of compiler directives to avoid double writing the OpenCL kernels as C functions and enabling the integration of OpenCL variables in C++.

Compiler directives

The use of compiler directives avoids the need to have two completely separate versions of the same code, one for CPUs and another for GPUs. Since the single thread loop (Alg. 1) and the equivalent OpenCL kernel (Alg. 2) share the same core operations, they could be combined with minimal changes to obtain a unique code by using compiler directives (Alg 6). The source code is then automatically changed by the compiler at compilation time. Thus, if the `__CPU_MODE__` compiler directive is enabled when compiling the solver source code, Alg. 1 is recovered. Otherwise, the OpenCL kernel of Alg. 2 is generated.

Algorithm 6 Combined CPU/OpenCL kernel version of Algorithm (1).

```
__kernel void edgeComputations(...)
{
#ifdef __CPU_MODE__
  for(edge=0;edge<Nedges;edge++)
  {
#else
  edge   = get_global_id(0);
#endif
  point1 = edgeNode(1,edge);
  point2 = edgeNode(2,edge);
  data1  = ReadPointData(point1);
  data2  = ReadPointData(point2);
  term   = F(data1,data2);
  WritePointResult(point1,term);
  WritePointResult(point2,term);
#ifdef __CPU_MODE__
  }
#endif
}
```

The way the C++ code invokes the kernel also changes depending on the execution mode. When the `__CPU_MODE__` compiler directive is used, the kernel is called as a conventional routine. However, when the code is run in OpenCL mode, there is an OpenCL specific command to execute the kernel [2, subsec. 5.8]. This simple but flexible solution avoids the need to use code translators to automatically generate the source code for one platform or another.

Use of OpenCL variables

Variables in OpenCL can be either global if they physically reside in the GPU memory or local if they are allocated in each GPU processors' local memory. In either case, they are generated as a memory buffer that is reserved by the time the GPU code is executed. If the same variable with the same name must be used when the code is executed in a CPU, there must be some means of by-passing the way OpenCL creates the variables and applying the normal C allocation instead. This is accomplished by creating a C++ class that serves as a wrapper for these OpenCL variables. That class internally uses the same compiler directive presented above to switch between OpenCL and CPU modes so that the end user is transparent to that duality.

For the CPU mode to be able to completely re-use the OpenCL kernel codings there is still one last step: OpenCL defines vector types that pack a number of scalar data types. For instance, an element of the `float8` vector type contains a group

of eight floats. As it has been mentioned before, these OpenCL variables are extensively used in our code, since it turns out that the data transfer between the GPU global memory and the GPU processor's memory is produced at a faster rate if these vector types are used in the indirect addressings. Thus, a set of C++ classes with the same semantics and operations as these of the vector type has been created; one class for each of the 2, 4, 8 and 16 vector types. Using these classes the use of OpenCL variables in the core operations of the CPU loops is transparent. Moreover, the performance of the CPU code is not degraded at all and the results of the CPU and GPU simulations are exactly the same.

CODE PERFORMANCE

The resulting code, compiled with the 4.5 version of the GCC compiler [23] runs 5% slower than its Fortran predecessor, compiled with the 10.3 version of the PGI compiler [24]. Different compilers have been used since these are the ones that provided the best speed results for each case. The performance of the new solver has been measured with a typical turbomachinery design case consisting of a single row with one million mesh points. The case has been run in a number of platforms to assess the versatility of the code on different processor architectures:

1. An Intel Xeon E5620 @ 2.4GHz with four CPU cores. Even though this chip is not the fastest chip in the market, it allows the simultaneous execution of eight processes with negligible loss of performance.
2. A cluster of Intel Xeon X5472@3GHz with an Infiniband network. Each processor of the cluster allows the simultaneous execution of four processes without substantial loss of performance, and the Infiniband network allows a fast communication between them.
3. A machine with four NVIDIA Quadro Fx3800 GPUs. The primary use of these GPUs is graphics processing, not computing. Besides, they have modest computing capabilities compared with the more recent NVIDIA or ATI GPUs.
4. A machine with two NVIDIA Tesla C1060 GPUs. These are GPUs exclusively dedicated to computing, being far more powerful than the Quadro GPUs.
5. A machine with an ATI Radeon HD 5970 GPU. As the Quadro GPU, this one is also used for graphics processing. However, it has superior computing capabilities, since it is a newer product. It must be noted that the potential of this GPU could not be fully exploited due to hardware restrictions when using OpenCL in this platform. Thus, just half of the theoretical computational power is used.

The CPU model used to host the GPUs is not relevant, since all the computing is done inside the GPU. When the GPUs are used in parallel, an equal number of independent CPU processes is created using MPI. This approach allows each GPU to be managed by a single CPU process, making the CPU/GPU communications easier. The domain decomposition for the parallel runs has been generated using the ParMeTiS partitioning library [25].

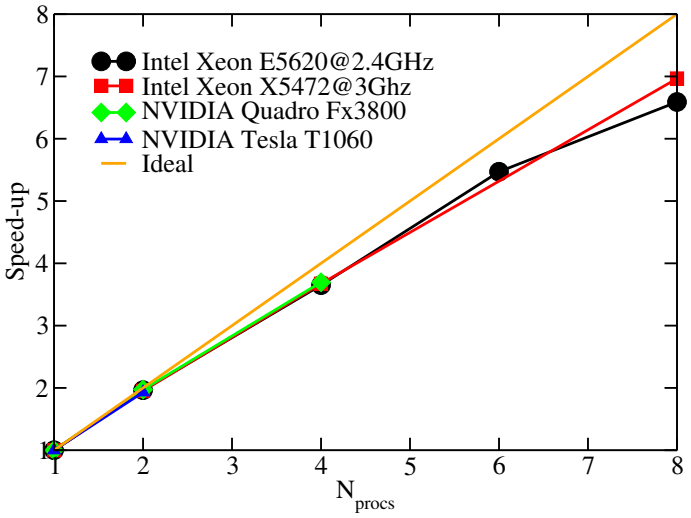


Figure 3: Speed-up curve of the CFD solver for a 1 million points mesh.

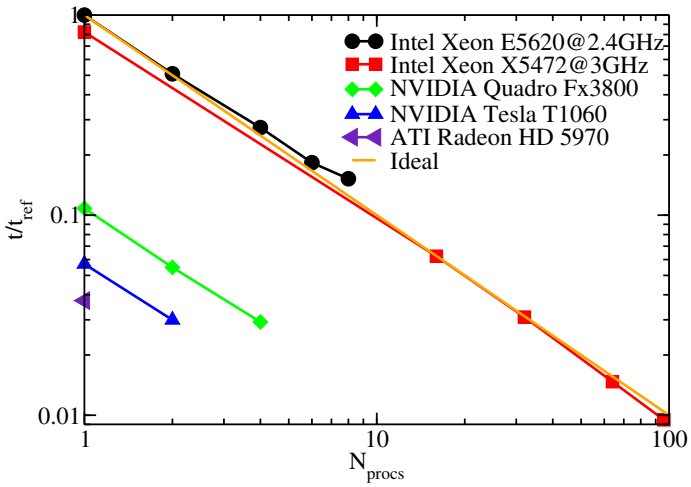


Figure 4: Execution times of the CFD solver for a 1 million points mesh, non-dimensionalized with the execution time of the single process case run in the Intel Xeon E5620.

The speed-up curve of this case is plotted in Fig. 3. It shows good scalability for all the cases considered. The comparison of the execution times in each platform is depicted in Fig. 4 where two interesting results may be seen. On one side, the CPU version of the CFD solver shows good parallel scalability up to 96 processes with a case that has just one million points. On the other side, the single processor GPU simulations are faster than their CPU counterparts by a factor of 9 for the Quadro GPUs, 17

for the Tesla GPUs and 26 for the ATI GPU. However, the Intel Xeon E5620 can run up to eight parallel processes with minimal loss in performance, hence it would be more appropriate to compare the $N_{procs} = 8$ simulation with the single GPU ones. In that case, the speed-up values are 1.4 for the Quadro GPU, 2.7 for the Tesla GPU and 4 for the ATI GPU. Analogously, the Intel Xeon X5472 shows an speed-up of nearly four when running on its four cores, hence the GPU/CPU execution time comparison yields a 2.3 speed-up for the Quadro GPU, 4.3 for the Tesla GPU and 6.6 for the ATI GPU.

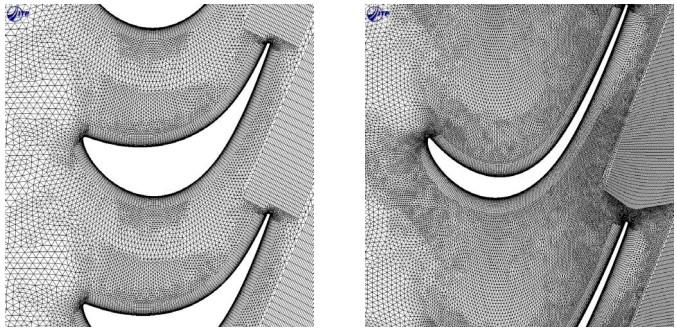
If more powerful CPUs, such as the Intel Xeon X5677 @ 3.46 GHz, were considered, an additional 1.4 speed-up factor would have been achieved in the CPU simulations, making the CPU and GPU execution times even more similar. In that hypothetical case, the speed-ups would be 2 for the Tesla GPU and 3 for the ATI GPU. Thus, even though the simulations run remarkably faster in the GPUs, using top-end CPUs is still attractive in terms of code performance. By keeping the ability to run in CPUs, all these machines can continue to be used. Besides, the use of OpenCL as programming language instead of CUDA allows the execution of Mu^2s^2T in the ATI GPU, which is not an NVIDIA GPU and therefore cannot execute codes programmed using CUDA. Moreover, there is no need to modify the source code to obtain an excellent speed-up value when compared to the NVIDIA GPUs, therefore the statement that the source code optimization for the NVIDIA GPU is very different from that of the ATI GPU does not hold, at least for this case.

RESULTS

Some additional cases have been run in order to validate the results obtained with the CFD code. Thus, the flow field around two LPT vane geometries, depicted in Figs. 5 and 6, has been solved with the present CFD solver running on four NVIDIA Quadro GPUs. The comparison between the experimental measurements obtained at the CTA high-speed wind tunnel [26] and the CFD solution is depicted in Figs. 7 and 8. Both the total pressure coefficient C_{pt} and the swirl angle α at a measurement plane located at $x/C_{ax} = 1.6$ downstream the vane trailing edge have been compared.

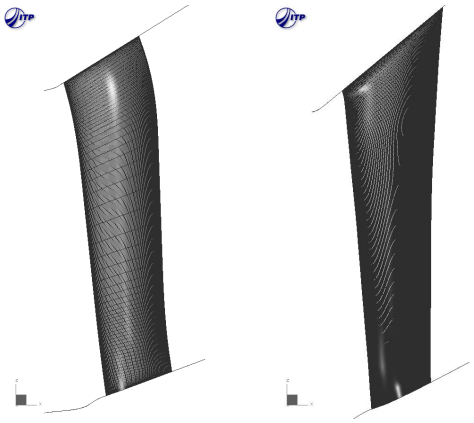
Fig. 7 shows the comparison between the measurements and the predicted flow field for the vane depicted in Figs. 5a and 6a, that has been meshed with a semi-unstructured mesh with 1 million points and 95 radial planes. This case has been run in four Quadro Fx3800 GPUs, and the steady-state solution has been obtained in 10 minutes. The predicted values are in close agreement with the experimental measurements for both quantities. The largest differences are presented in the secondary flow zone, where the solver over-predicts the total pressure, even though the position of the peaks of losses and angles is well reproduced.

Fig. 8 shows the same comparison, but for the vane geometry of Figs. 5b and 6b, that has been meshed with a finer semi-



(a) (b)

Figure 5: Blade to blade view of the geometry and mesh of two LPT vanes.



(a) (b)

Figure 6: Meridional view of the geometry and mesh of two LPT vanes.

unstructured grid with 3 million points and 107 radial planes. The steady-state solution for this case has been obtained after having run 35 minutes in four Quadro Fx3800 GPUs. The differences between the solver results and the experimental measurements are also small, being more noticeable in the tip secondary flow zone of the total pressure and in the swirl angle distribution.

Even though the results reported for both configurations are acceptable, it must be noted that dealing with such low Reynolds number flows would require advanced and sophisticated transition models capable of predicting the position and shape of the suction side separation bubble. This model, which has been reported to improve the simulation capabilities of the CPU version of Mu^2s^2T [27], has not been implemented yet in this OpenCL version of the code, hence the ability to predict the variation of the losses and angle at very low Reynolds number is still limited. However, the main purpose of these simulations is to show that

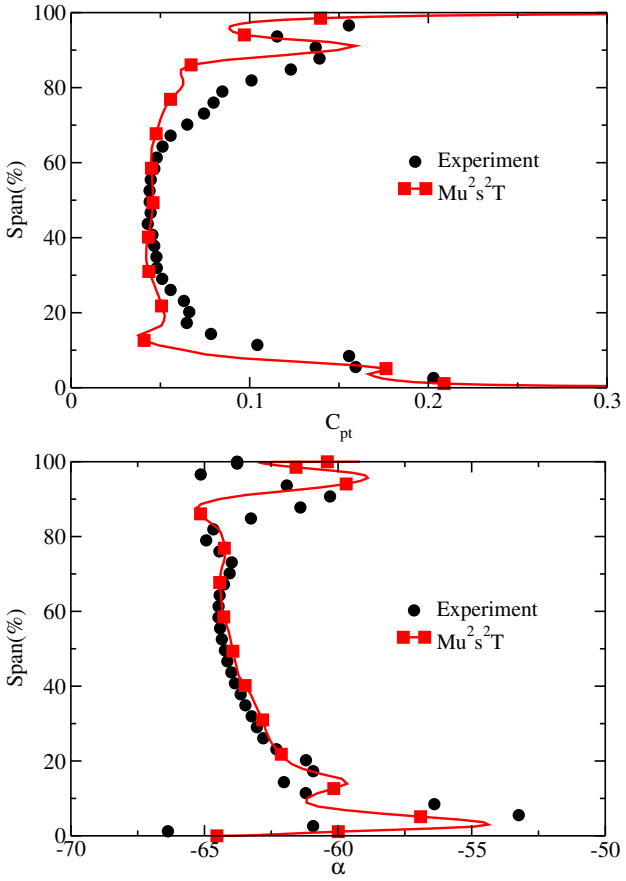


Figure 7: Comparison between the measured and predicted radial distributions of C_{pt} (top) and swirl angle (bottom) at $x/C_{ax} = 1.6$ for the LPT vane of Figs. 5a and 6a.

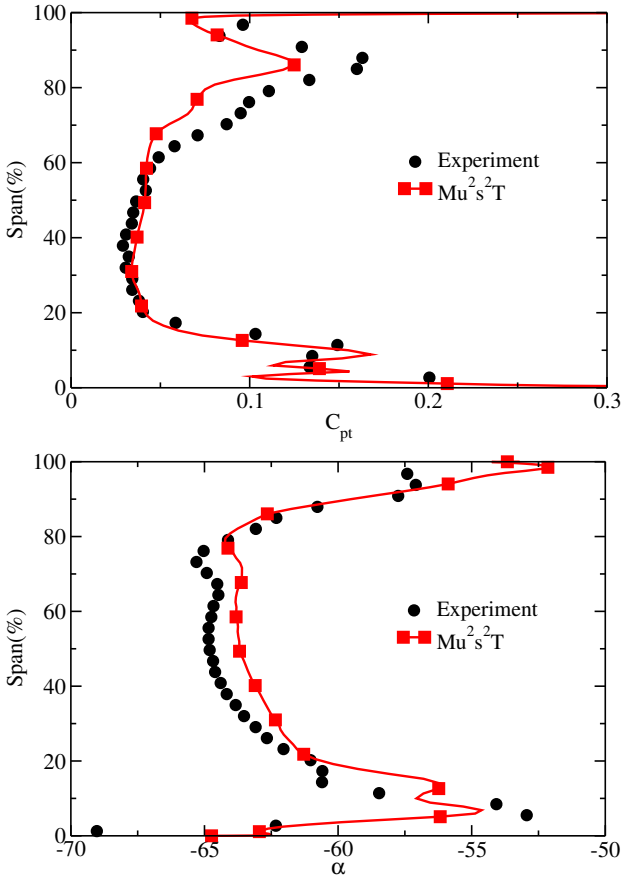


Figure 8: Comparison between the measured and predicted radial distributions of C_{pt} (top) and swirl angle (bottom) at $x/C_{ax} = 1.6$ for the LPT vane of Figs. 5b and 6b.

realistic configurations, with realistic grids and turbulence models, may be effectively implemented in a GPU environment using OpenCL.

CONCLUSIONS

The implementation of an edge-based Navier-Stokes solver for unstructured grids has been presented. The solver is primarily written in C++ and it is intended to be executed either on CPUs or GPUs, hence a strategy for achieving this goal is presented. The main advantage of this approach is that it avoids double writing significant parts of the source code. The OpenCL programming language has been used to write those portions of the code that must be executed in the GPU. This programming approach allows the resulting code to be executed on a wide range of architectures with minimal implementation penalty and ensures the uniqueness of the solution for all platforms.

Details of the implementation of the most time consuming computations have been presented. Since the code is unstructured, the access to the data needed to evaluate gradients, fluxes, etc. is not direct and presents limitations which are more severe when running on GPUs. On one side, simultaneous access to the data must be avoided in order to obtain correct results. On the other, care must be taken to ensure a high performance of the code in GPUs. The key parameter for achieving high performance in such platforms is to maximize the number of FLOPS per indirect memory access.

The resulting code has been executed in several CPUs and GPUs, obtaining a good scalability when multiple processes are run in parallel. The solver runs up to 26 times faster on a modern GPU when compared to a single process run on a CPU. If the multi-core capability of modern CPUs is exploited, modern GPUs continue to be up to 4 times faster than multi-core CPUs, hence saving computation time and allowing the possibility to study more complex flow phenomena that were out of the scope in the standard design loop. Future work will focus on allowing a wider range of simulations, such as unsteady rotor-stator sim-

ulations, and improving or including more fluid modellisation capabilities to deal with typical turbomachinery flow regimes.

ACKNOWLEDGEMENTS

The authors would like to thank ITP for its permission to publish this work. They would also like to thank Bull for letting them test the CFD code in their Tesla cluster in Grenoble and finally they would like to express their gratitude to Carlos Vasco and Eduardo García from the Technology and Methods Department at ITP for setting-up the hardware used for the presented tests.

REFERENCES

[1] NVIDIA, "CUDA Programming Guide," http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf, October 2010.

[2] Khronos Group, "OpenCL 1.1 Specification," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, September 2010.

[3] Klöckner, A., "PyOpenCL: OpenCL parallel computation API from Python," http://mathema.tician.de/software/pyopencl.

[4] Elsen, E., LeGresley, P., and Darve, E., "Large calculation of the flow over a hypersonic vehicle using a GPU," *Journal of Computational Physics*, Vol. 227, 2008, pp. 10148–10161.

[5] "BrookGPU," http://graphics.stanford.edu/projects/brookgpu/.

[6] Tölke, J., "Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA," *Computing and Visualisation in Science*, Vol. 13, No. 1, 2010, pp. 29–39.

[7] Jacobsen, D. A., Thibault, J. C., and Senocak, I., "And MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," *Proceedings of the 48th AIAA Aerospace Sciences Meeting*, No. AIAA 2010-522, Orlando, Florida, 4-7 January 2010.

[8] Jespersen, D. C., "Acceleration of a CFD Code with a GPU," Tech. Rep. NAS-09-003, NASA, Ames Research Center, Moffett Field, CA 94035, U.S.A., 2009.

[9] Brandvik, T. and Pullan, G., "An Accelerated 3D Navier-Stokes Solver for Flows in Turbomachines," *Proceedings of ASME Turbo Expo*, No. GT2009-60052, Orlando, Florida, June 8-12 2009.

[10] Castonguay, P., Williams, D., Vincent, P., and Jameson, A., "A Fast, Scalable Unstructured Compressible Viscous Flow Solver," *NVIDIA GPU Technology Conference*, 2010.

[11] Corrigan, A., Camelli, F., Löhner, R., and Wallin, J., "Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware," *Proceeding of the 19th AIAA Computational Fluid Dynamics*, No. AIAA 2009-4001, San Antonio, Texas, 22-25 June 2009.

[12] Corrigan, A., Camelli, F., Löhner, R., and Mut, F., "Semi-Automatic Porting of a Large-Scale Fortran CFD Code to GPUs," To appear in the International Journal for Numerical Methods in Fluids.

[13] Burgos, M., Contreras, J., and Corral, R., "Efficient Edge-Based Rotor/Stator Interaction Method," *AIA*, Vol. 49, No. 1, January 2011, pp. 19–31.

[14] Luo, H., Baum, J., and Löhner, R., "Edge-Based Finite Element Scheme for the Euler Equations," *AIAA Journal*, Vol. 32, 1994, pp. 1183–1190.

[15] Martinelli, L., *Calculations of viscous flow with a multigrid method*, Ph.D. thesis, Princeton University, 1987.

[16] Pierce, N., *Preconditioned Multigrid Methods for Compressible Flow Calculations on Stretched Meshes*, Ph.D. thesis, University of Oxford, 1997.

[17] Mavriplis, D., "Directional Agglomeration Multigrid Techniques for High Reynolds Number Viscous Flows," *AIAA Journal*, Vol. 37, No. 10, October 1999, pp. 1222–1230.

[18] Wilcox, D. C., *Turbulence Modeling for CFD*, DCW Industries, Inc., 2006.

[19] Gómez, R., *Una Estructura de Datos basada en Aristas para la Resolución de las Ecuaciones de Navier-Stokes*, Ph.D. thesis, Escuela Técnica Superior de Ingenieros Aeronáuticos, Universidad Politécnica de Madrid, Noviembre 2000.

[20] "NVIDIA GPUDirect," http://developer.nvidia.com/object/gpudirect.html, June 2010.

[21] Cuthill, E. and McKee, J., "Reducing the bandwidth of sparse symmetric matrices," *Proceedings of the ACM National Conference*, 1969, pp. 157–192.

[22] Löhner, R., *Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, 2nd Edition*, Wiley, 2008.

[23] "GCC , the GNU Compiler Collection," http://gcc.gnu.org/.

[24] The Portland Group, "PGI Compiler," http://www.pgroup.com/.

[25] Karypis, G., Schloegel, K., and Kumar, V., "ParMeTiS, Parallel Graph Partitioning and Sparse Matrix Ordering Library, version 3.1," August 15 2003.

[26] Vázquez, R., Iturregui, J., Arsuaga, M., and Armañanzas, L., "A New Transonic Test Turbine Facility," *XVI International Symposium on Air Breathing Engines (ISABE)*, 2003.

[27] Corral, R. and Gisbert, F., "Prediction of a Separation-induced Transition using a Correlation-based Transition Model," *Proceedings of ASME Turbo Expo*, No. GT2010-23239, Glasgow, Scotland, June 2010.