49th AIAA Aerospace Sciences Meeting & Exhibit, 4-7 January 2011, Orlando FL

AIAA-2011-0944

Acceleration of an unstructured hybrid mesh RANS solver by porting to GPU architectures

WN Dawes* & PC Dhanasekaran
CFD Laboratory, Department of Engineering, University of Cambridge, Cambridge CB2 1PZ, UK.
and
WP Kellar,
Cambridge Flow Solutions Ltd, Compass House, Vision Park, Cambridge, CB4 9AD, UK

The modern CFD process consists of mesh generation, flow solving and post-processing integrated into an automated workflow. During the last several years we have developed and published research aimed at producing a meshing and geometry editing system, implemented in an end-to-end parallel, scalable manner and capable of automatic handling of large scale, real world applications. The particular focus of this paper is the associated unstructured mesh RANS flow solver and the porting of it to GPU architectures. After briefly describing the solver itself, the special issues associated with porting codes using unstructured data structures are discussed – followed by some application examples.

I. Introduction

Our research has been guided in recent years by considering the overall efficiency of the CFD process, consisting of integrated mesh generation, flow solving and post-processing, and the need to link this in an automated workflow to enhance industrial productivity. For several years we have focused on the mesh generation as described in Dawes *et al* [2006-2009] and Janke *et al* [2008] as this was clearly the limiting factor for geometries of medium to high complexity. It seems natural now to return to the flow solver itself which is again the bottleneck. Over the last decade flow solver technology has made only modest advances algorithmically but there has been a revolution in the hardware available – cheap, general purpose, commodity graphics cards: GPUs.

The GPU was originally developed to specialize in compute-intensive, highly parallel operations typical of graphics rendering. Recently the development of high level programming languages like CUDA C for nVIDIA cards (see *http://www.nvidia.com/object/cuda_home_new.html*) has allowed the GPU to evolve into a very flexible and powerful general purpose processor. Current GPUs offer 32 & 64 bit floating point precision and the potential of very substantial Gflop rate improvements over conventional CPU-based systems. The CFD community has been quick to recognize the potential benefits – see for example Brandvik *et al* [2003, 2008], Corrigan et al [2010], Patnaik *et al* [2010] & Jacobsen *et al* [2010] and code speed ups of at least an order of magnitude have been regularly reported. What is clear, however, is that it is much easier to take advantage of the architecture of a GPU with a structured mesh data type.

The purpose of this paper is to describe recent work porting a conventional unstructured mesh RANS flow solver to GPU architectures. The organization of this paper is as follows: Section II describes the basic RANS solver; Section III describes briefly the basic architecture of a GPU and the associated code porting issues; Section IV shows some practical applications; and Section V concludes and indicates some future work themes.

* wnd@eng.cam.ac.uk

II. The basic unstructured mesh RANS flow solver

Our RANS solver ("NEWT") was written in the 1990's and applied extensively to turbomachinery, as described for example in Dawes [1992, 1993], Longley et al [2000], Angel et al [1998] and Jackson [1996]. Later, extensions were made to other application areas; see, for example, Kellar *et al* [1999], Dawes *et al* [2001], Smith at al [2003]. The solver was very simple and consisted of the Reynolds Averaged Navier Stokes equations discretised using a second order finite volume approach on an all-tetrahedral mesh with both a simple mixing length and also a low Reynolds number k-ε turbulence model. Time marching was via a standard 4-step Runge-Kutta method with explicit residual averaging; blended second-fourth order artificial smoothing was applied to stabilize the solution.

For the present research the code needed to be brought up to date somewhat – including conversion from Fortran to C. The most significant change was to extend the data structures to permit the use of hybrid meshes consisting of hexahedral, prismatic, pyramidal and tetrahedral finite volumes. This was straightforwardly achieved using the face-based data structure shown in Figure 1. Each face is either quadrilateral or triangular and the main computational work in the solver is forming flux sums by looping over faces and then adding or subtracting the face flux to the two cells neighboring the face. A separate loop over boundary faces applies appropriate boundary conditions.

The current version of the code uses single step Adams-Bashforth time integration (to minimise parallel communication cost compared to multi-step Runge-Kutta) but does not yet contain multi-grid convergence acceleration.

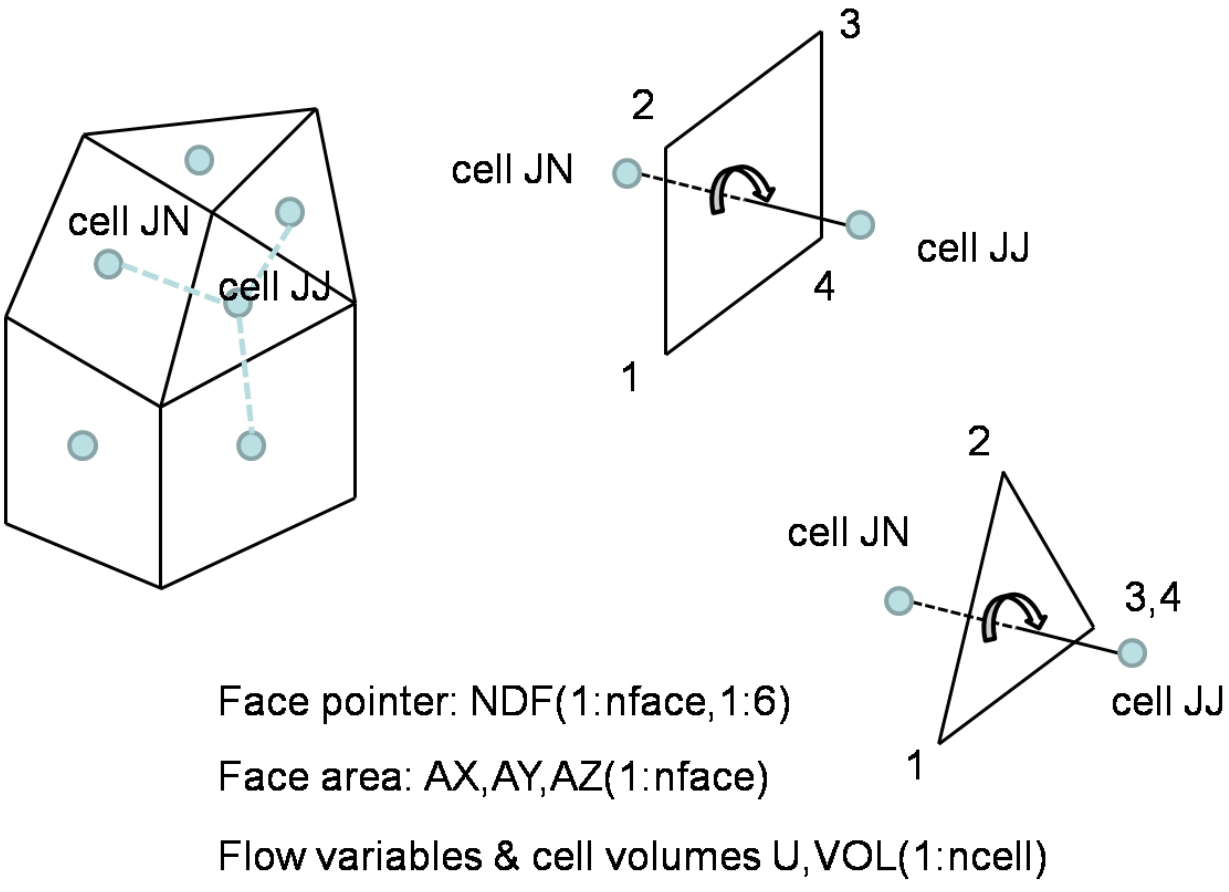


Figure 1: Basic face-based data structures used in the flow solver

III. Porting to GPU architectures

The basic architecture of a GPU is sketched in Figure 2. The GPU is a *device* which acts as a co-processor to the host CPU, has its own memory, *device memory*, and can run many *threads* in parallel. Data-parallel portions of an application are executed in the device as *kernels*; a kernel is executed as a *grid* of *thread blocks*. Within each thread block, threads can use fast *shared memory* and can synchronize their operation; threads in different blocks cannot communicate. Each thread runs the same program (Single Instruction Multiple Thread) and uses its ID to compute addresses and make control decisions.

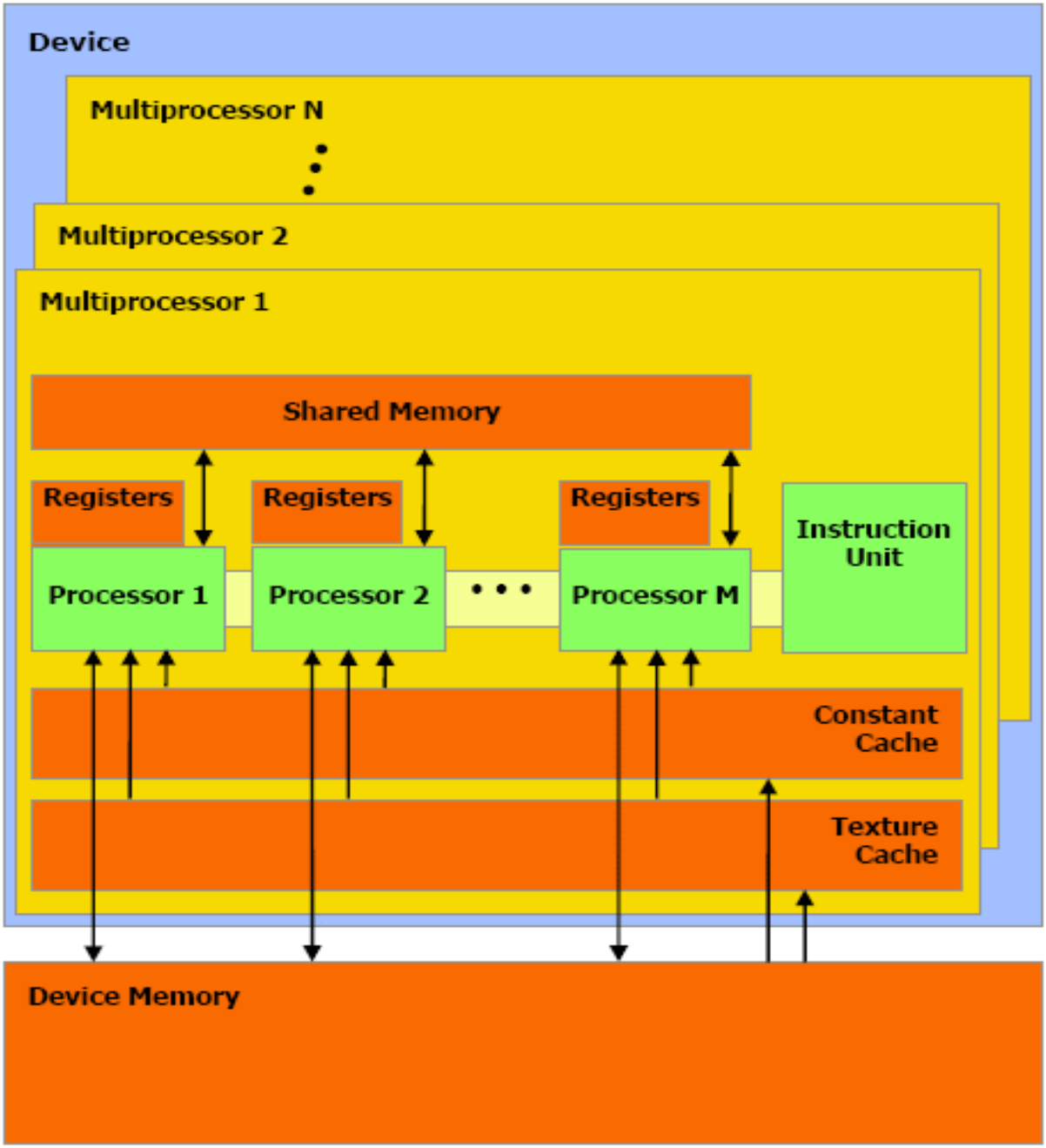


Figure 2: Basic architecture of a GPU (reference: *www.nvidia.com*)

The host can read and write to device memory which in turn communicates with each multiprocessor. Device memory is off-chip, large, and kernel inputs and outputs reside here. Shared Memory is on-chip, as fast as the Registers, but small and shared amongst the cores. Table 1 shows appropriate data for a typical GPU.

Number of devices	**2**
Global memory on each device	**1 GB**
Number of multiprocessors	**30**
Number of cores per multiprocessor	**8**
Total number of cores	**240**
Amount of constant memory	**64 kb**
Amount of shared memory	**16 kb**
Maximum number of threads per block	**512**
Warp size	**32**
Maximum size of each block	**512x512x64**
Maximum size of each grid	**65535x65535x1**
Clock rate	**1.24 GHz**

Table 1: The basic properties of a typical GPU, the GeForce GTX 295

The CUDA programming language (see *http://www.nvidia.com/object/cuda_home_new.html*) is based on standard C with minimal extensions to allow the program to be executed in a number of parallel threads. These extensions cover:

- computation partitioning
  - function declarations: `__host__, __global__, __device__`
  - mapping threads to the device: `function<<<n,m>>>(arg1,arg2,…);`
- data partitioning
  - data declarations: : `__host__, __global__, __device__`
- data management
  - copying to/from host: `cudaMemcpy(a_d,a,size,cudaMemcpyDeviceToHost);`
- concurrency management
  - for example: `__synchtreads();`

The essence of porting the flow solver to the GPU is to replace the face loops with corresponding threads. There are several key issues to be faced associated with the unstructured data storage. The most basic requirement is the re-ordering/re-numbering of the basic face-based looping structure to make memory access more efficient.

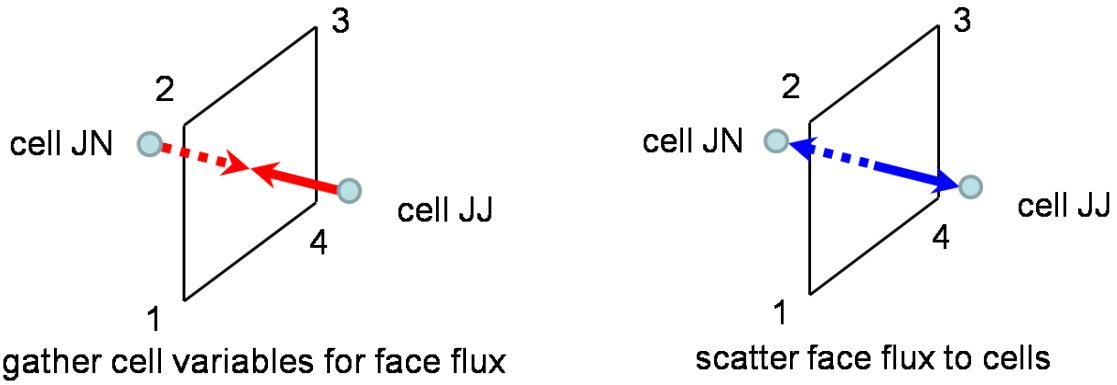


Figure 3: The basic gather/scatter operations associated with cell face flux sums

Figure 3 illustrates another issue: the algorithm proceeds first by gathering cell-based data to the face to compute the face flux; then this flux is scattered back to the flux sums accumulated within neighbouring cells – and hence to

updated flow variables. The face loop needs to be *coloured* (like for a vector processor) so that there is no memory contention when face flux sums are executed (and scattered to neighbour cells) as set of thread blocks.

To illustrate the sort of changes which need to be made Figure 4 shows two corresponding code fragments. The main difference is that in the GPU implementation all the variables associated with a face are *coalesced* into contiguous temporary store, *sxtr[]* , to improve memory access and the face fluxes are stored as part of the colouring process and updated via an additional kernel. However, this coalescence is performed on the host CPU and is not yet as efficiently implemented as it will be in future. Great care also must be taken in selecting appropriate block sizes and we are not yet exploiting fully the fast device shared memory. As Figure 4 implies, it is common to split individual C loops into several CUDA kernels to get higher performance.

```
// HYBRID_NEWT - calculation of fluxes over faces
// in C code
...
...
...
for (int j =0; j < mesh.nface; j++) {
   //get face neighbours
      jj = mesh.ndf[0][j];
      jn = mesh.ndf[1][j];
    ...
    ...
   //calculate diffusive fluxes
      dfro = facav * (variables.ro  [jj] - variables.ro  [jn]);
      dfre = facav * (variables.roe [jj] - variables.roe [jn]);
      dfrx = facav * (variables.rovx[jj] - variables.rovx[jn]);
      dfrt = facav * (variables.rovt[jj] - variables.rovt[jn]);
      dfrr = facav * (variables.rovr[jj] - variables.rovr[jn]);

   //modify convective fluxes
      fdiffo = flro - dfro;
      fdiffe = flre - dfre;
      fdiffx = flrx - dfrx;
      fdifft = flrt - dfrt;
      fdiffr = flrr - dfrr;

   //update flux sum in face neighbour "jj"
      sro[jj] = sro[jj] + fdiffo;
      sre[jj] = sre[jj] + fdiffe;
      srx[jj] = srx[jj] + fdiffx;
      srt[jj] = srt[jj] + fdifft;
      srr[jj] = srr[jj] + fdiffr;

   //update flux sum in face neighbour "jn"
      sro[jn] = sro[jn] - fdiffo;
      sre[jn] = sre[jn] - fdiffe;
      srx[jn] = srx[jn] - fdiffx;
      srt[jn] = srt[jn] - fdifft;
      srr[jn] = srr[jn] - fdiffr;
}
```

```
// GPU Kernel for face loop
// all the flow variables are stored in a temporary linear array "sxtr"
// with 18 variables for each cell
// all the flux terms  are stored in "fdout" - each face has got 5 terms

__global__ void solver_floop (real *fdout, real *bdout,
                              real *sol,
                              real *bxtr, real *fxtr, real *sxtr,
                              Bcs  *bcs, int *surf, Solver_t *solver,
                              int *ndf, int *ndb,
                              Msize mesh) {
  ...
  ...
  ...
  //get thread id
     idx = (blockIdx.x*blockDim.x)+threadIdx.x;
     if(idx > mesh.nface-1) return;

  //get face neighbours
     jj = ndf[idx*2  ];
     jn = ndf[idx*2+1];
   ...
   ...
  //calculate diffusive fluxes
     dfro = facav * (sxtr[jj*18+10]   - sxtr[jn*18+10]);
     dfre = facav * (sxtr[jj*18+11]   - sxtr[jn*18+11]);
     dfrx = facav * (sxtr[jj*18+15]   - sxtr[jn*18+15]);
     dfrt = facav * (sxtr[jj*18+16]   - sxtr[jn*18+16]);
     dfrr = facav * (sxtr[jj*18+17]   - sxtr[jn*18+17]);

  //update flux sum and store them
     fdout[idx*5  ] = flro - dfro;
     fdout[idx*5+1] = flre - dfre;
     fdout[idx*5+2] = flrx - dfrx;
     fdout[idx*5+3] = flrt - dfrt;
     fdout[idx*5+4] = flrr - dfrr;
}
```

```
// GPU kernel for cell loop
// update cells from fluxes
// stored in srvals - 5 terms for each cell - sro, sre, srx, srt, srr

__global__ void solver_update_srvalsf (real *srvals, real *fdout,
                                       int *flist,int *nfs,
                                       int *nfe, Msize mesh){
...
...
...
//get thread id
  idx = blockIdx.x*blockDim.x+threadIdx.x;

//update cell variables from flux sum
  for (int j = nfs[idx]; j < nfe[idx]; j++) {
    k1 = abs(flist[j])-1;
    sign = (flist[j] > 0)? 1:-1;
    for (int k =0; k < 5; k++)
      srvals[idx*5+k] = srvals[idx*5+k] + sign*fdout[k1*5+k];
  }
}
```

Figure 4: Code fragments in C (left) and in CUDA C (right).

The next section will show application to realistic test cases and report timings.

IV. Practical application

As practical demonstration we show sample simulations and timings for two standard turbine blade test cases.

The *first* blade selected, named ACE-RD (see Haller [1979]), is typical of an aero-engine HP turbine and is characterized by very high flow turning and streamline curvature. The blade was tested in cascade and with the basic parameters shown in the Table.

Inlet flow angle	**56 deg**
Exit flow angle	**-65 deg**
Exit Mach number	**0.90**
Reynolds number	**2.e+05**

Table: Basic parameters of the ACE/RD blade

An overview of the mesh is shown in Figure 5 and the predicted flow field in Figure 6. The hybrid mesh contained ~117k cells and was generated by the *BoXeR*™ software (see *www.cambridgeflowsolutions.com*).

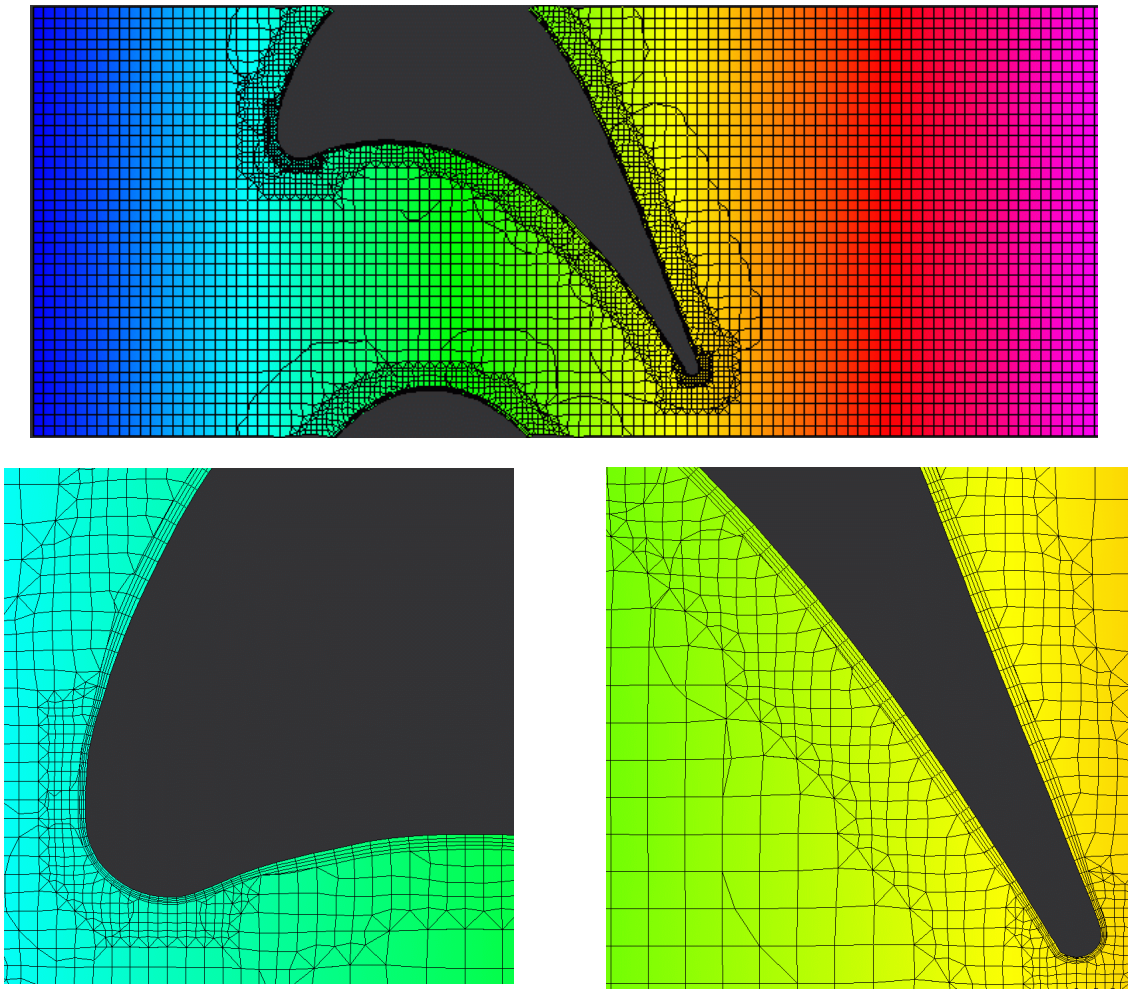
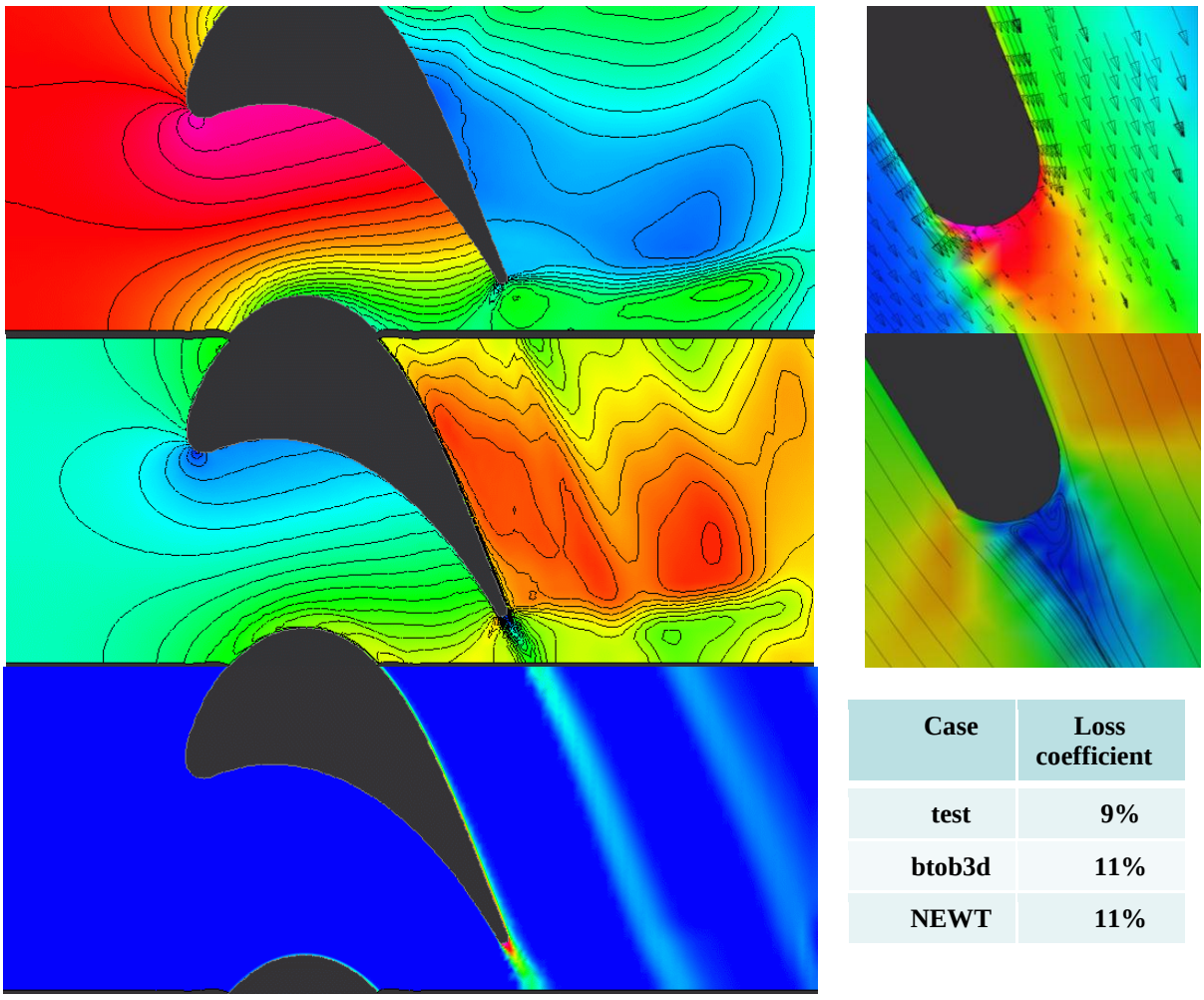


Figure 5: Hybrid mesh used for the ACE-RD cascade



Case	Loss coefficient
test	9%
btob3d	11%
NEWT	11%

Figure 6: Overview of the predicted flowfield for the ACE-RD cascade; on the left static pressure, Mach number and entropy; on the right some TE detail and loss coefficient comparisons.

As shown in the Table the current GPU version of the solver runs around a factor of nine (9.04) faster on the GeForce GTX 295 GPU compared to the host CPU (a single core of an Intel Core 2 Quad Q8300, 2.5GHz); we expect that this will increase by at least a factor of two in the near future with better data management.

Original Fortran code	**687 seconds**
C code on Cpu	**540 seconds**
C code on Gpu	**76 seconds**
Speed up Cpu/Gpu	**9.04**

Table: Computer run times for the ACE-RD case

The *second* blade selected, named LA (see Hodson *et al* [1986]), is typical of an aero-engine LP turbine and is also characterized by very high flow turning and streamline curvature. The blade was tested in cascade with the aim of measuring the 3D secondary flow structures associated with the turning of the endwall boundary layer through the blade row.

The basic parameters are shown in the following Table.

Inlet flow angle	**38.8 deg**
Exit flow angle	**-54 deg**
Exit Mach number	**0.70**
Reynolds number	**2.e+05**

Table: Basic parameters of the LA blade

The hybrid mesh contained ~550k cells (for a half-span) and was similar in character and quality to that for the earlier ACE-RD blade and so is not illustrated to save space; again, it was generated by the *BoXeR*™ software (see *www.cambridgeflowsolutions.com*). Some plots of the predicted flowfield are shown in Figure 7. Included is an experimentally measured blade exit total pressure traverse (from Hodson *et al* [1986). The strongly three-dimensional secondary flow structures are clearly in evidence and are generally well resolved in the predictions.

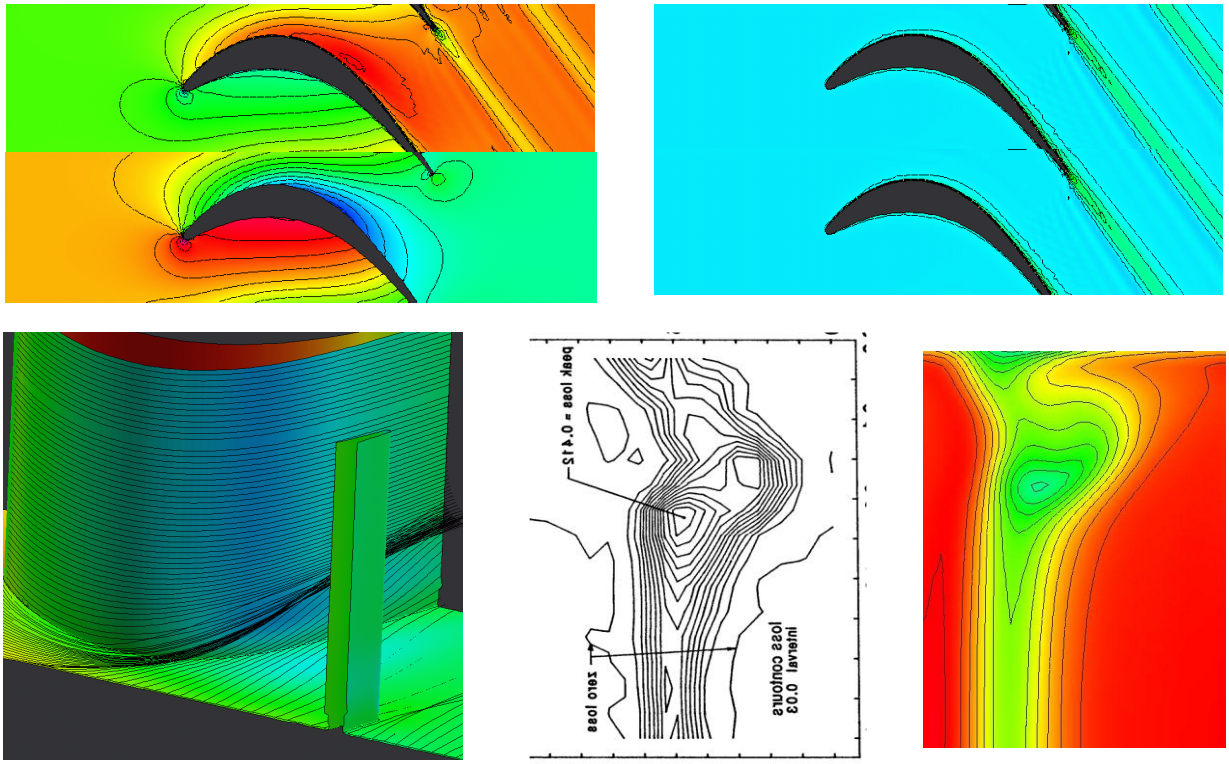


Figure 7: Overview of the predicted flowfield for the LA cascade; top left mid-span Mach number & static pressure; top right mid-span entropy; bottom row (from left to right) predicted secondary flow structures, measured & predicted total pressure traverse at x/c=142%.

As shown in the Table the current GPU version of the solver runs around a factor of ten (10.6) faster on the GeForce GTX 295 GPU compared to the host CPU. This is a little faster than in the previous case as it is fully 3D so the ratio of interior faces to boundary faces is rather higher changing the relative work a little.

Original Fortran code	**3846 seconds**
C code on Cpu	**3055 seconds**
C code on Gpu	**364 seconds**
Speed up Cpu/Gpu	**10.6**

Table: Computer run times for the LA case

V. Concluding Remarks

This paper has described the extension of our NEWT RANS flow solver to run on unstructured, hybrid meshes made up of hexahedra, pyramids, prisms and tetrahedrons. Next, the porting of the code to run on nVIDIA GPU devices using CUDA C was discussed. Application to two sample turbomachinery test cases showed a very useful speedup, around a factor of ten – and more is expected in the near future with better data management.

Future work will focus on better data coalescence on the host CPU and better use of fast device shared memory on the GPU. Using MPI to link multiple GPUs will lead to further speedup but, more importantly, will allow larger memory and hence larger, more realistic problems to be tackled.

In addition, we are currently adding agglomeration multi-grid to the basic solver – this is expected to accelerate convergence by a further factor of 3-5 holding out the prospect of fully 3D blade-blade simulations in of order a minute on a single GPU!

VI. References

Angel & Hill (1998) DERA Report AS/PTD/CR980263/1.0

BoXeR™: *www.cambridgeflowsolutions.com*

Brandvik T & Pullan G "Acceleration of a 2D Euler flow solver using commodity graphics hardware" J.Proc.IMechE, Part C, Vol.221, pp.1745-1748, 2003

Brandvik T & Pullan G "Acceleration of a 3D Euler flow solver using commodity graphics hardware"" 46th AIAA Aerospace Sciences Meeting & Exhibit, Reno 2008

Corrigan A, Camelli F, Lohner R & Mut F "Porting of as edge-based CFD solver to GPU's" AIAA-2010-523, Orlando FL, 2010

Dawes WN "Towards a fully integrated parallel geometry kernel, mesh generator, flow solver & post-processor", 44th AIAA Aerospace Sciences Meeting & Exhibit, 9-12 January 2006, Reno, NV, AIAA-2006-45023

Dawes WN, Harvey SA, Fellows S, Favaretto CF & Vellivelli A "Viscous Layer Meshes from Level Sets on Cartesian Meshes", 45th AIAA Aerospace Sciences Meeting & Exhibit, 8-11 January 2007, Reno, NV, AIAA-2007-0555

Dawes WN, Harvey SA, Fellows S, Eccles N, Jaeggi D & Kellar WP "A practical demonstration of scalable, parallel mesh generation"47th AIAA Aerospace Sciences Meeting & Exhibit, 5-8 January 2009, Orlando, FL, AIAA-2009-0981

Dawes WN, "The simulation of three-dimensional viscous flow in turbomachinery geometries using a solution-adaptive unstructured mesh methodology", Transactions of the ASME J. of Turbomachinery, Vol 114, No 3, pp 528, 1992.

Dawes WN "The extension of a solution-adaptive 3D Navier-Stokes solver towards geometries of arbitrary complexity", (92-GT-363) Transactions of the ASME J. of Turbomachinery, Vol 115, No 2, pp 283-, 1993.

Dawes WN, Dhanasekaran PC, Demargne AAJ, Kellar WP & Savill AM "Reducing Bottlenecks in the CAD-to-Mesh-to-Solution Cycle Time to allow CFD to Participate in Design". ASME Gas Turbine Conference, Munich, 2000-GT-0517 Transactions of the ASME, J. of Turbo, Vol 123, pp 552-557, 2001

Haller BJ "Transonic turbine cascade" PhD Thesis, Cambridge University, 1979

Hodson HP and Dominy RG "Three dimensional flow in a low pressure turbine cascade at its design condition" ASME Paper 86-GT-106, 1986

Jackson (1996) DRA Report AS/PTD/CR96023/1

Jacobsen DA, Thibault JC and Senocak I "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters" AIAA-2010-522, Orlando FL, 2010

Janke E, Hodson HP, Faccini, Popovic I, Lehmann K, Georgakis C, Pons L, Lutum E, Wallin F, Dawes WN & Favaretto F "Selected aerothermal CFD analyses of high pressure turbine topics within the AITEB-2 project", ECCOMAS-2008, 5th European Congress on Computational Methods in Applied Science and Engineering, Venice-Italy June 30-July 5, 2008

Kellar WP, Savill AM & Dawes WN "Integrated CAD/CFD Visualisation Of A Generic Formula 1 Car Front Wheel Flowfield". Lecture Notes in Computer Science Volume1593, 1999

Longley, J.P. & Demargne, A.A.J. "The Aerodynamic interaction of stator shroud leakage and mainstream flows in compressors", ASME Paper GT-xx-00, 2000

nVIDIA/CUDA: http://www.nvidia.com/object/cuda_home_new.html

Patnaik G & Obenschain KS "Using GPUs on HPC applications to satisfy low-power computational requirements", AIAA-2010-524, Orlando FL, 2010

Smith AN, Babinsky H, Dhansekaran PC, Savill AM & Dawes WN "Computational investigation of groove controlled shock wave boundary layer interaction" AIAA Paper 2003-0446, Reno, January 2003

-oOo-