Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware

Tobias Brandvik[*] and Graham Pullan[†]
Whittle Laboratory, Department of Engineering,
University of Cambridge, Cambridge, CB3 0DY, UK

Abstract

The porting of two- and three-dimensional Euler solvers from a conventional CPU implementation to the novel target platform of the Graphics Processing Unit (GPU) is described. The motivation for such an effort is the impressive performance that GPUs offer: typically 10 times more floating point operations per second than a modern CPU, with over 100 processing cores and all at a very modest financial cost. Both codes were found to generate the same results on the GPU as the FORTRAN versions did on the CPU. The 2D solver ran up to 29 times quicker on the GPU than on the CPU; the 3D solver 16 times faster.

Nomenclature

c_v	Specific heat capacity at constant volume
e	Specific total energy $= c_v T + \frac{1}{2}V^2$
h_0	Specific stagnation enthalpy
p	Pressure
t	Time
u,v	Cartesian components of velocity
V	Velocity (magnitude)
T	Temperature
Y_p	Stagnation pressure loss coefficient $= \frac{p_{01}-p_0}{p_{01}-p_2}$

[*]tb302@cam.ac.uk
[†]gp10006@cam.ac.uk

ρ	Density

subscripts

0	Stagnation
1	Inlet
2	Exit

1 Introduction

Speed and accuracy are key factors in the evaluation of flow solver performance. In an industrial context, a *step change* in the metric of time per node per iteration allows the engineer either to improve accuracy by using more mesh points or higher fidelity modelling techniques, or to reduce run-times allowing a more rapid design-solve-results feedback cycle. To significantly impact the design process, an improvement in speed of at least one order of magnitude is required. This has been the objective of the current work.

The compute power of commodity CPUs continues to increase. During the 1980's and 1990's, the number of FLOPs (floating point operations per second) available on a PC grew as a result of exponential increases in CPU transistor count ("Moore's Law") and rises in clock speed. In recent years, clock speed has stabilised and improvements have come instead from parallel computing. Hardware can be either externally parallel (clusters of PCs networked together) or internally parallel (multi-core CPUs). However, there exists inside every modern PC a highly parallelised (approximately 100 core) processor with far greater peak FLOPs performance than the CPU: the graphics processing unit (GPU) (Figure 1). The higher peak performance is made possible by constraining the functionality of the GPU and employing highly parallelised hardware.

There has been little previous work on the acceleration of engineering CFD codes using graphics hardware. The implementation of a 2D incompressible Navier-Stokes solver on a GPU is described by Harris[9]. Here, the emphasis is on producing visually realistic flows. An extension of this work into three dimensions has been performed by Crane *et al.*[3]. Hagen *et al.*[8] present 2D and 3D Euler solvers with application to two types of spatially periodic flow: 2D shock-bubble interaction and 3D Rayleigh-Taylor instability. The solutions presented are physically sensible and speed-ups compared to an equivalent CPU code of 10-20 times are reported. The present authors have recently published results from a 2D Euler solver[1] with speed-ups of up to 40 times obtained.

This paper presents the development of two separate Euler solvers for GPUs, one 2D

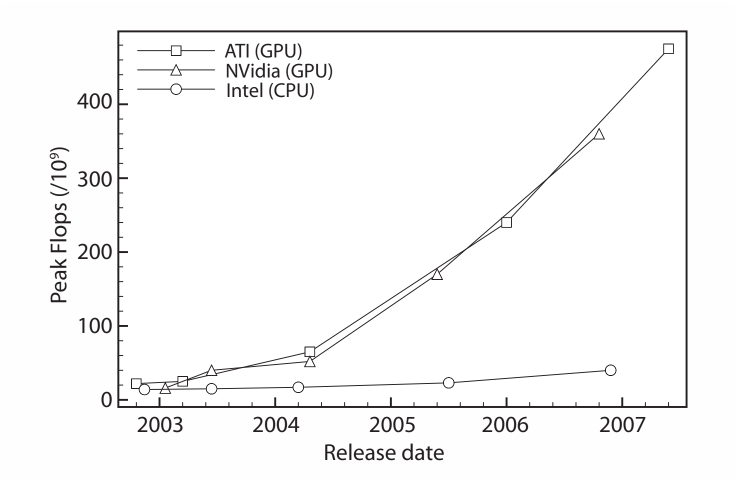


Figure 1: CPU (Intel) and GPU (NVIDIA and ATI) floating point performance

and the other 3D. The paper begins by describing current graphics hardware and how it can be programmed. The implementations of the solvers are discussed, particularly the changes to the code needed to maintain performance when going from two to three dimensions. For the 2D code, results for the flow through a transonic turbine cascade are presented; for the 3D code, secondary flow development in a low speed linear turbine cascade is used as a test case. The performance of the solvers, in terms of run-times compared to similar CPU codes, is also analysed.

2 Graphics acceleration hardware

GPUs are specialised co-processors. To adapt a CPU code to run on the GPU, some knowledge of the functionality of the GPU is required. The GPU is designed to transform a polygon model of a three-dimensional scene to an image rendered on the screen. This process is known as the graphics pipeline, Figure 2. The *vertex processor* receives a list of interconnected vertices (geometric primitives) from the CPU. At this stage, these vertices can be transformed and coloured. The *rasteriser* converts each geometric primitive into a group of *fragments* (pixels which are not yet displayed). Each fragment is then coloured by the *fragment processor* using information from the vertices or from *textures* (artwork) stored in GPU memory. Textures are similar to traditional 2D arrays on the CPU, but are stored in a special way so that the *texture cache* performs best for access patterns which have good 2D spatial locality.

To enable different transformations and shading effects, both the vertex and fragment processors are programmable and operate on 32-bit floating point numbers. The whole process is highly parallelised with many vertex and fragment processors on the GPU. Each geometric primitive can be sent to a different vertex processor and each fragment to a different fragment processor so that each final pixel is evaluated independently.

Up until the latest generation of GPUs[1] that support Microsoft's DirectX 10 specification, the graphics pipeline was implemented with different hardware for vertex and fragment shaders. However, in order to simplify the architecture and utilise the available transistors more efficiently, the two have now been combined into unified shaders capable of doing both vertex and fragment shading. Another new feature of the latest generation is that a shader can write to arbitrary locations in memory, not just the vertex or fragment it is currently operating on.

With these modifications, a modern GPU can, for general computing purposes, be viewed as a set of processing units, each with cached read/write access to main memory. In the computing literature, this is often referred to as a SIMD array, that is, each core must execute the same code on different instances of input data: Single Instruction Multiple Data. This is quite distinct from the multi-core CPUs that are now available. These are MIMD chips (Multiple Instruction Multiple Data) which means that each core behaves, in most respects, as an independent traditional single-core CPU.

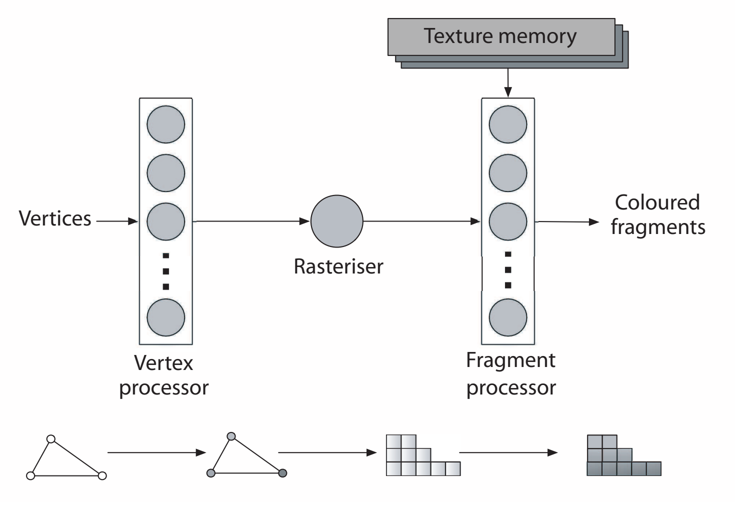


Figure 2: The graphics 'pipeline'

[1]This includes any GPU with NVIDIA's G80 chipset (released November 2006) and any GPU with AMD's R600 chipset (released May 2007), or later versions.

3 Programming methods

3.1 Introduction

Once the non-graphics community began to appreciate that GPUs were fast, programmable and could handle floating point numbers, general purpose GPU (the acronym 'GPGPU' is often used) applications began to be developed. In general, the fragment processor is targetted because this supports the gathering of data from texture memory. At this point we note that any CPU code which involves the repeated application of the same function to the elements of a large array of data can be re-coded for the GPU and may demonstrate performance benefits. CFD is, therefore, an ideal application for the GPU.

There are a number of options open to the CFD developer who wishes to convince the GPU that it is rendering the latest computer game when it is actually solving the Navier-Stokes equations. One possibility is to re-write the application so that calls are made to graphics libraries such as OpenGL or DirectX. However, this can result in a code that is not easily understood. The alternative is either to use the low-level interfaces from GPU hardware manufacturers which grant the programmer access to details of the GPU hardware without using graphics terminology, or to use a high-level language ([2, 16, 13]) where the GPU is hidden, as far as possible, from the developer. The latter option provides a welcome abstraction from 'textures', 'rendering', 'threads' and 'caches', at the expense of some loss of potential performance optimisation capability.

For simplicity, the high-level language BrookGPU[2] was chosen for the 2D solver. An attempt was also made to use it for the 3D solver, but poor performance led to the adoption of the lower-level interface CUDA, developed by GPU manufacturer NVIDIA.

3.2 BrookGPU

BrookGPU is the graphics hardware development of the Brook language for 'streaming' architectures. Streaming is an approach for producing highly parallel computers which are easy to scale to an arbitrary number of processors. As compared to conventional languages, the functionality of a streaming program is highly constrained and centres around the use of *streams* and *kernels*. A stream is a collection of data records similar to a traditional array. A kernel is a function that is implicitly called for each record in an input stream to produce a new record in an output stream. A kernel has access to read only memory (gather streams) but the result of a kernel operation can only depend on the current record in the input stream - no dependency on neighbouring records, for example, is permitted. Brook has previously been used to code a 2D Euler solver for the Merrimac streaming machine[7], but no results from the hardware are shown and the code is not aimed at GPUs. It is evident that the streaming paradigm is well suited to GPUs where: kernel=fragment processor, input stream=fragment, gather stream=texture memory.

3.3 CUDA

CUDA is a technology for GPU computing from NVIDIA. It exposes the hardware as a set of SIMD multiprocessors, each of which consists of a number of processors (currently 16). These processors have arbitrary read/write access to a global memory region called the *device memory*, but also share a memory region known as *shared memory*. Access times to the device memory are long, but it is large enough (up to 1.5 GB on current cards) to store entire data sets. The shared memory, on the other hand, is small (currently 16 KB) but has much shorter access times. It acts as an explicitly-managed cache, meaning that the developer is responsible for all data transfers to it. This is in contrast to a traditional cache on a CPU, where data fetches from main memory are automatically cached. Managing the data transfers between device memory and shared memory in an optimal way is crucial to getting good performance. The main requirement is that data transfers should be to or from contiguous blocks in device memory of at least 16 elements.

Traditional textures can also be stored in the device memory and accessed in a read-only fashion through the multiprocessor's *texture cache*. In addition, a limited amount of read-only memory is available for constants used by all the processors. This is accessed through the *constant cache*. Only the shared memory is used for the solver presented in this paper. A schematic overview of the hardware model is shown in Figure 3, which is taken from the CUDA documentation.

In a CUDA program, the developer sets up a large number of threads (often several thousand) that are grouped into thread blocks. A CUDA thread is the smallest unit of execution and has a set of registers and a program counter associated with it. This is similar to traditional CPU threads, but CUDA threads are much less expensive to create and swap between. Each thread block is executed on a single multiprocessor. It is possible to synchronize the threads within a block, allowing the threads to share data through the shared memory. Given that a thread block can consist of more threads than the number of processors in a multiprocessor, the hardware is responsible for scheduling the threads. This allows it to hide the latency of fetches from device memory by letting some threads perform computations while others wait for data to arrive.

For grid applications such as CFD, a natural way of structuring the code is to split the main grid into smaller grids which can fit into the shared memory. A block of threads is

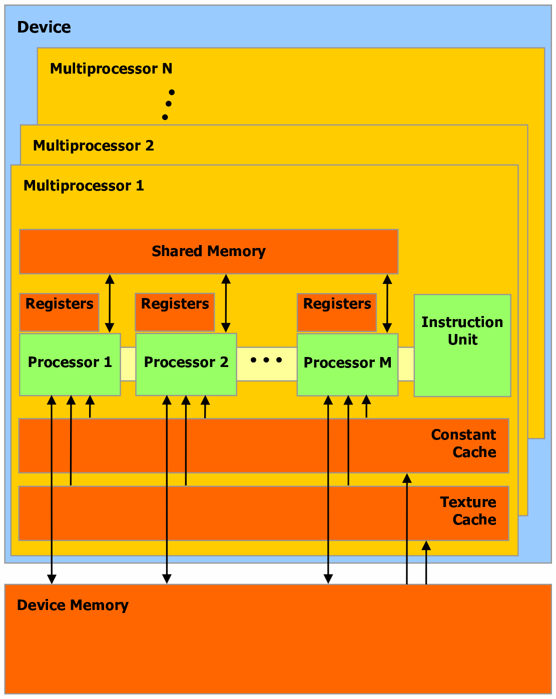


Figure 3: The CUDA hardware model

then started within each of the smaller grids to compute the updated variables.

4 2D Euler Solver

4.1 Introduction

The equations to be solved are the Euler equations for compressible inviscid flow in integral conservative form:

$$\int_{\mathsf{vol}} \frac{\partial}{\partial t} \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho e \end{bmatrix} d\mathsf{vol} = \oint_A \begin{bmatrix} \rho u \\ p + \rho u^2 \\ \rho u v \\ \rho u h_0 \end{bmatrix} dA_x + \oint_A \begin{bmatrix} \rho v \\ \rho u v \\ p + \rho v^2 \\ \rho v h_0 \end{bmatrix} dA_y \qquad (1)$$

where the integrals are over a volume vol with surface A, (A_x, A_y) is the inward facing surface area vector.

The problem is discretised using a finite volume approach with vertex storage in a structured grid of quadrilateral elements. The spatial derivatives are evaluated using a centred, second order, scheme and the temporal ones to first order using the Scree scheme (Denton[6]). Second order smoothing, spatially varying time-steps and negative feedback (to damp out large residuals) are employed but no multigrid method is implemented. The authors emphasise that this numerical approach is not novel and is based on the widely used methods of Denton[4, 5].

The implementation is split between the CPU and the GPU. All pre- and post-processing is done on the CPU, leaving only the computation itself to be performed on the GPU. For example, the CPU constructs the grid and evaluates the face areas and cell volumes. The initial guess of the flowfield is also done on the CPU. Each time-step of the computation then involves a series of kernels on the GPU which: evaluate the cell face fluxes; sum the fluxes into the cell; calculate the change in properties at each node; smooth the variables; and apply boundary conditions.

4.2 2D BrookGPU implementation

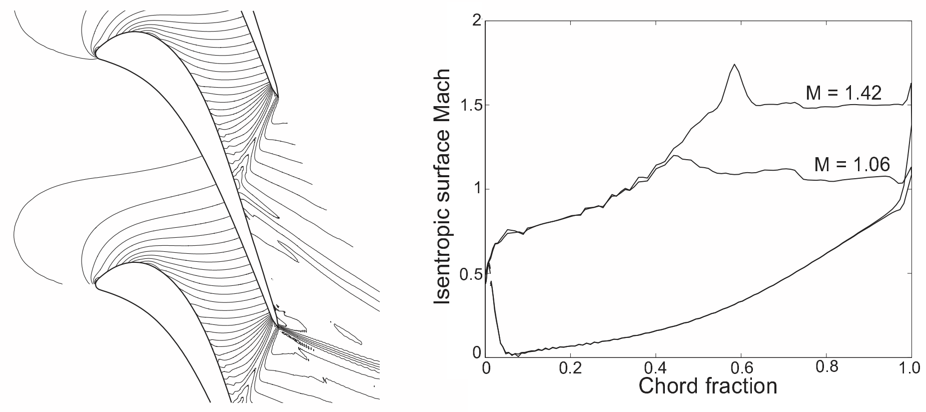
Each kernel operates on all the nodes, i.e. no distinction is made between boundary nodes and interior nodes. This can cause difficulties if an efficient code is to be obtained. For example, the change in a flow property at a node is formed by averaging the flux sums of the adjacent cells; four cells surround an interior node, but only two at a boundary node.

This problem is overcome using *dependent texturing.* The indices of the cells required to update a node are pre-computed on the CPU and loaded into GPU texture memory. For a given node, the kernel obtains the indices required and then looks up the relevant flux sums which are stored in a separate GPU texture. This avoids branching within the kernel.

4.3 Results

4.3.1 Example flowfield

Results from the VKI McDonald transonic turbine test case [14] are presented in Figure 4. The GPU results were found to be identical to those from a Fortran implementation so only the former are presented. The grid used employed 208 $\times$ 208 points. The isobars in Figure 4(a) show the shock structure emanating from the trailing edge at an exit Mach number of $M_2 = 1.42$. Figure 4(b) shows surface Mach number distributions at $M_2 = 1.06$ and $M_2 = 1.42$. The discontinuity on the suction-surface at 60% chord at the higher exit Mach number, caused by the trailing edge shock, is clear.



(a) Isobars at $M_2 = 1.42$ (b) Surface Mach number distributions

Figure 4: Results from the VKI 'McDonald'[14] transonic turbine test case

4.3.2 Performance

The GPU implementation obtained a 29x speed-up compared to a reference CPU implementation. This is shown in Figure 8, while the benchmarking hardware details are summarized in section 5.4.2. A 40x speed-up has been reported previously by the present authors[1] when comparing the same implementations, but running on a slower CPU.

The large speed-up can be explained by the memory fetches performed by the solver having good 2D spatial locality, thereby making efficient use of the GPU's texture cache. This leads to a large fraction of the GPU's memory bandwidth (which is an order of magnitude greater than the CPU's) being utilized.

5 3D Euler solver

5.1 Introduction

The 3D Euler solver uses the same algorithm as the 2D solver, but solves one extra momentum equation for the third spatial dimension. Again, only the computation is done on the GPU, with the CPU handling all pre- and post-processing.

5.2 3D BrookGPU implementation

Due to the greater number of memory fetches in the 3D algorithm, it was decided that using dependent texturing to deal with boundary nodes would lead to excessive bandwidth usage. Simple branching statements were used instead. Since the boundary branch is only entered for the small fraction of nodes lying on the surface, the performance hit is modest.

Although GPUs support 3D textures natively, this capability is not exposed by BrookGPU. It is therefore necessary to use a memory layout format known as flat 3D textures, as described by Harris[10]. Here, slices of the 3D volume are laid out next to each other in a large 2D texture. This is shown in figure 5. There is little overhead associated with flat 3D textures - an additional texture with the offset of each slice and some simple address arithmetic is all that is needed.

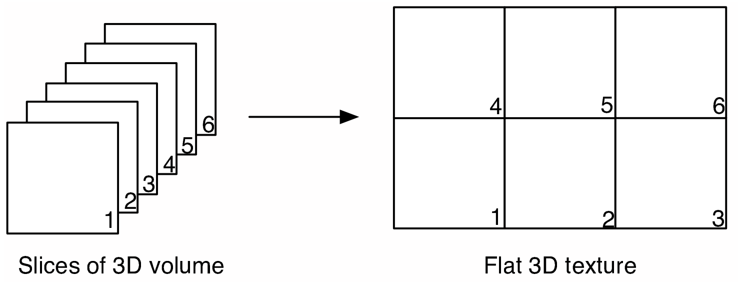


Figure 5: Flat 3D textures, after Harris[10]

5.3 3D CUDA implementation

The CUDA implementation of the 3D solver differs significantly from the BrookGPU implementation. The computations performed by the kernels remain the same, but rather than using textures, CUDA's shared memory region is used instead.

The main idea is to split the flow domain into into small 3D sub-blocks that are computed independently of each other. One thread is started for each element in a plane of the sub-block. At the start of the kernel, each thread loads in the necessary node values corresponding to its location. We then iterate upwards in the sub-block, each time fetching a new plane and computing the values for the current one. Given that the shared memory can hold 16 KB, a typical sub-block size is 16x10x5. Having 16 elements in the primary direction is necessary to get good memory bandwidth. For such sub-block sizes, the surface-to-volume ratio is low enough to get good data reuse. The overall approach is similar to that described by Williams[17] *et al.* for structured grid applications on the Cell microprocessor.

As an example, consider a generic kernel which operates on a 7-node stencil, i.e. the node being updated and the 6 nearest nodes in the i, j and k directions. For such an operation, it is necessary to keep three planes in shared memory at any given time as illustrated in Figure 6. The primary direction is along the i-axis, while the direction of iteration is along the j-axis. The planes are made as big in the k-direction as possible to maximize the reuse of data in shared memory. The result of the computation is written straight to the global memory. Given that the computation depends on the surrounding nodes, the threads at the boundaries of the plane only fetch data into shared memory and do not actually perform any computations.

5.4 Results

5.4.1 Example flowfield

The 3D code was evaluated on the low-speed linear turbine cascade geometry used by Harrison[11]. The simulations were run using a simple sheared H-mesh with $151 \times 49 \times 49$ points in the axial, pitchwise and spanwise directions respectively. The domain covered one half-span and extended one axial chord upstream and downstream of the blade row. This test case was chosen because the blade, being of a low aspect ratio and high deflection, exhibits significant secondary flows. Although secondary flows are caused by the rotation of vorticity present in the inlet endwall boundary layer, this process is inviscid and can be predicted by an Euler solver (provided that the incoming boundary layer is correctly

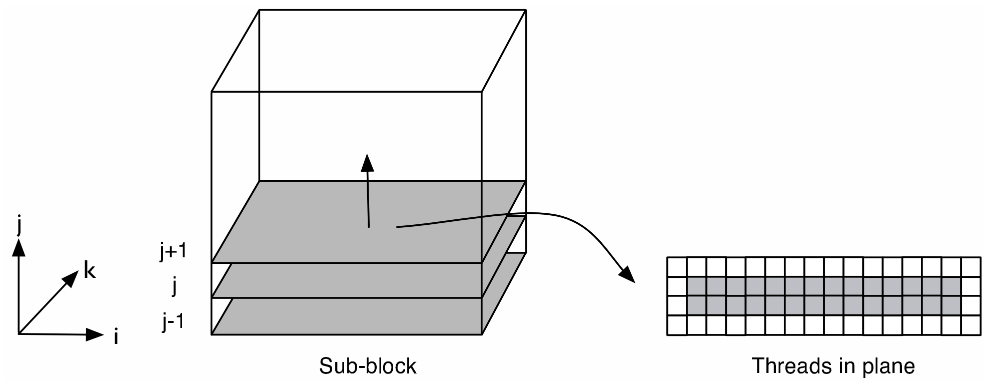


Figure 6: A generic kernel operating on the nearest neighbours. One thread is started for each element in the plane. The threads then iterate upwards together, each time fetching new data into shared memory. Only the interior threads shown highlighted perform computations.

prescribed). Figure 7 shows streamlines initiated in the inlet endwall boundary layer. Contours of stagnation pressure loss coefficient are also shown at inlet, 35%, 70% and 100% axial chord. In the manner described by Langston *et al.*[12] and Sieverding[15], the streamlines are seen to be driven toward the suction-surface by the cross-passage pressure gradient. Once there, they rise up the blade, then lift off and roll-up into the passage vortex. The contours shows that this vortex contains all the low stagnation pressure fluid present in the inlet boundary layer.

5.4.2 Performance

Figure 8 summarizes the speed-ups obtained for the various GPU implementations as compared to reference CPU implementations written in Fortran. The results are shown for a grid size of 300,000 nodes. The CPU solvers were run on a Intel Core 2 Duo 2.33 GHz processor (using a single core only). Both BrookGPU solvers used an ATI 1950XT, while the CUDA solver used an NVIDIA 8800GTX.

The performance of the 3D BrookGPU solver was found to be far from that of the the 2D version, the former obtaining only a 3x acceleration. It is thought that this is due to the flat 3D texture format reducing the spatial locality of the texture fetches, with out-of-plane lookups being far away from in-plane ones.

The CUDA solver showed a significant 16x speed-up over the CPU implementation. As for the 2D BrookGPU implementation, a high utilization of the GPU's peak memory

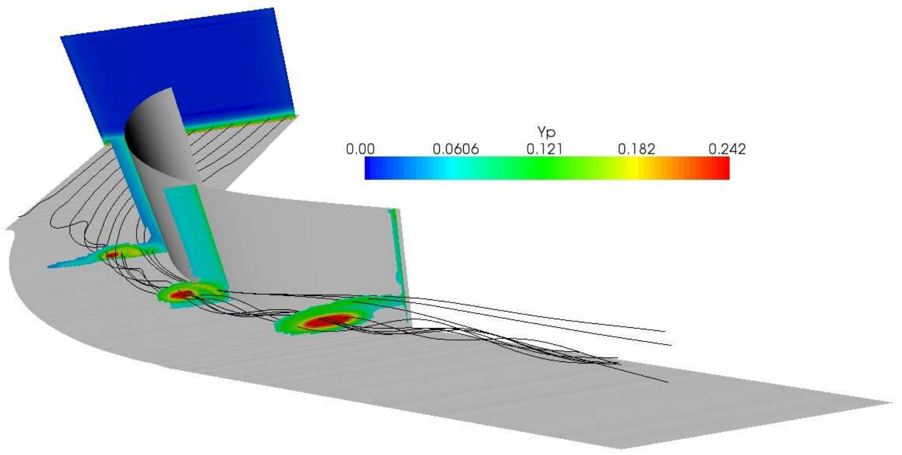


Figure 7: Results from the Harrison[11] low speed turbine cascade test case. Streamlines are started in the inlet boundary layer, contours are of loss coefficient, Y_p. The endwall and suction-surface are shown gray.

bandwidth is responsible. In the CUDA case however, we achieve this through to efficient shared memory management rather than cache-friendly texture lookups.

While not shown in this figure, the experience of the authors is that the GPU implementations scale linearly with both smaller and larger meshes. The CPU runs slightly faster on small meshes that fit entirely in the cache than it does on large meshes.

6 Conclusions

The speed-ups obtained in this work, 29× for the 2D solver and 16× for the 3D code, demonstrate that the GPU is well suited to CFD. The porting of an existing FORTRAN CFD code to run on a GPU requires significant effort and, to complicate matters, there are currently several programming interfaces available. It is very possible, however, that programmers will not be able to avoid modifying their codes to suit forthcoming hardware developments; many-core architectures, CPU or GPU, are likely to be the future of scientific computation.

Acknowledgments

The authors would like to thank NVIDIA for supplying the NVIDIA GPUs used in this work.

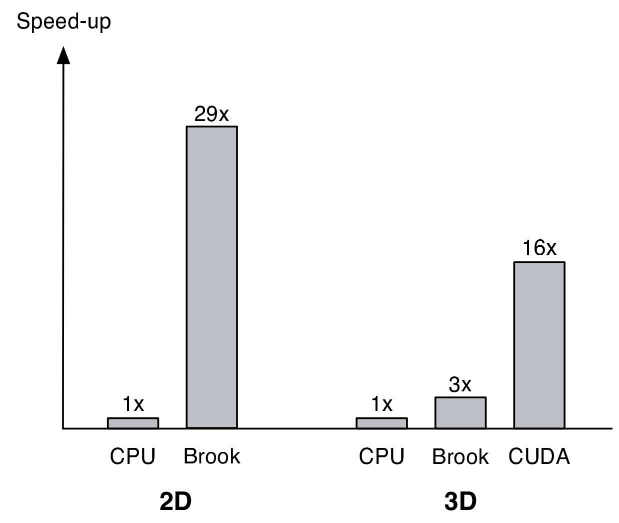


Figure 8: Speed-up comparison for a mesh size of 300,000 nodes

References

[1] Tobias Brandvik and Graham Pullan. Acceleration of a two-dimensional euler flow solver using commodity graphics hardware. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 221(12):1745–1748, 2007.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, F. Kayvon, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004*, pages 777–786, New York, NY, 2004.

[3] Kennan Crane, Ignacio Llamas, and Sarah Tariq. *GPU Gems 3*, chapter 30. Addison-Wesley Professional, 2007.

[4] J. D. Denton. An Improved Time Marching Method for Turbomachinery Flows. *ASME 82-GT-239*, 1982.

[5] J. D. Denton. The Calculation of Three Dimensional Viscous Flow Through Multistage Turbomachines. *ASME J. Turbomachinery*, 114(1), 1992.

[6] J. D. Denton. The Effects of Lean and Sweep on Transonic Fan Performance. *TASK Quart.*, pages 7–23, Jan 2002.

[7] M. Fatica, A. Jameson, and J. Alonso. Stream-FLO: an Euler Solver for Streaming Architectures. *AIAA 2004-1090*, 2004.

[8] Trond Runar Hagen, Knut-Andreas Lie, and Jostein R. Natvig. Solving the euler equations on graphics processing units. In *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 220–227. Springer, 2006.

[9] Mark J. Harris. *GPU Gems*, chapter 38. Addison-Wesley, 2004.

[10] Mark J. Harris, III William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 223, New York, NY, USA, 2005. ACM.

[11] S. Harrison. The Influence of Blade Lean on Turbine Losses. *ASME 90-GT-55*, 1990.

[12] L.S. Langston, M.L. Nice, and R.M. Hooper. Three-Dimensional Flow within a Turbine Blade Passage. *ASME J. Eng. for Power*, 99:21–28, 1977.

[13] Michael McCool and Stefanus DuToit. *Metaprogramming GPUs with Sh.* AK Peters, 2004.

[14] C. Sieverding. Base Pressure in Supersonic Flow. In *VKI LS: Transonic Flows in Turbomachinery*, 1976.

[15] C.H. Sieverding. Recent Progress in the Understanding of Basic Aspects of Secondary Flow in Turbine Blade Passages. *ASME J. Eng. for Gas Turbines and Power*, 107:248–257, 1985.

[16] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.

[17] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.