Acceleration of a 2D Euler Solver using graphics hardware

T. Brandvik[1], G. Pullan[2]

CUED/A-TURBO/TR.132 2007

1 - Research assistant, CUED.
2 - Research fellow, CUED.

Report based on fourth year undergraduate project 2006/7 by T. Brandvik.

Summary

This report details work undertaken to implement a 2D Euler solver for Graphics Processing Units (GPUs). The starting point for this effort is a reference implementation written in Fortran that runs on the Central Processing Unit (CPU).

The motivation for using the GPU is that it offers significantly more computational power than the CPU. Several methods for programming the GPU are presented, and the abstraction of the GPU as a stream computer is explained. The approach chosen for this work, the BrookGPU C extension and the accompanying source-to-source compiler, is given particular focus.

The increase in computational power of the GPU comes at the expense of less flexible control logic, a restriction that requires several changes to eliminate the use of conditional branching to detect boundaries. The solver presented in this work uses a procedure called dependent texturing to overcome this problem. This method allows the indices used in the solver's finite areas scheme to be pre-computed on the CPU.

The GPU solver achieves a 40x speed-up compared to the CPU implementation for large grids. An analysis of the GPU assembly code indicates that the speed of the GPU solver is limited by the memory system, and that only 30% of the computational power is used.

Contents

List of Figures

1 Introduction

The stagnation in the clock-speed of central processing units (CPUs) has led to significant interest in parallel architectures that offer increasing computational power by using many separate processing units[8]. Modern commodity graphics hardware contains such an architecture in the form of the graphics processing unit (GPU).

Given the computational requirements of simulating fluid flows, the GPU is an interesting target platform for CFD codes. This report details the work carried out to port an existing Fortran implementation of a 2D Euler solver to the GPU. The structure of the report is outlined below.

Section 2 gives the background for the project. The motivation for using the GPU and the approach taken to implement the Euler solver is explained, with particular focus on the abstraction of the GPU as a stream computer. A short overview of previous work on running CFD algorithms on GPUs is also given.

Section 3 discusses the implementation of the GPU solver, explaining the changes to the reference implementation made necessary by the restrictions of GPU architecture.

Section 4 presents the test cases used for validation. The speed and memory performance of the solver is also given. The performance of the solver is investigated in detail by performing an analysis of the underlying assembly code.

Finally, some interesting directions for future work are described in section 5.

2 Background

2.1 Stream computing

The current state of semiconductor technology is one where computation is cheap but bandwidth expensive. This is the basic motivation for the interest in stream computing, a programming model that aims to reduce bandwidth requirements by exploiting the data parallelism inherent in many computing problems.

The stream programming model is based on two elements, *streams* and *kernels*. A stream is a collection of records, much like a normal array in traditional programming languages such as C or Fortran. A kernel is a functional element that is implicitly called for each record in an input stream to produce a new record that is put in an output stream. In addition to input and output streams, a kernel also has access to so-called gather streams - read-only memory that can be freely indexed into. Depending on the constraints of the target hardware, it is possible to have multiple input and output streams.

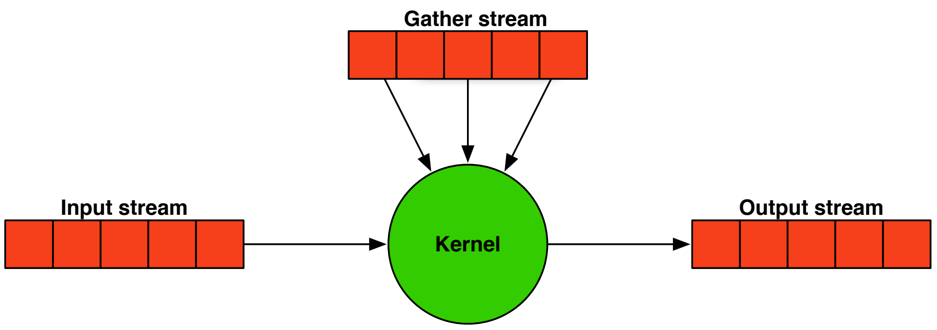

Figure 1: The stream computing model.

The result of a kernel operation can only depend on the current record in the input stream, no dependency on neighbouring records is allowed. This has two benefits:

1. The overall task of computing an output stream from an input stream can be split into many smaller tasks that can each be executed independently on different processing units. In theory, stream architectures can therefore scale to an arbitrary number of processors.

2. The lack of inter-dependancy allows for memory latency hiding. Whenever a memory fetch from a gather stream is required to compute the output from a stream record, the computation can be put into a queue and the processing unit can begin operating on another record while it waits for the fetch result.

It is important to note that benefits of memory latency hiding are only achieved if there are enough computational operations in between memory fetches to keep the processing units busy. The ratio between computational instructions and memory fetches is discussed by Dally et al.[4], who give it the name *arithmetic intensity*. Programs with high arithmetic intensity tend to be bound by the computational speed of the system, while those with a low arithmetic intensity tend to be bound by the memory access speed.

Several general-purpose stream architectures have been proposed, each with its own chip design and programming language. Recent efforts include the MIT Raw architecture and the StreaMIT language[16], the Stanford Imagine system and StreamC language[16], as well as the the the Merrimac streaming supercomputer and the Brook language[4].

The Merrimac design is particularly relevant to the work presented in this report. This architecture consists of a series of nodes, each containing several custom stream processors, interconnected with off-the-shelf components. Although no actual hardware implementation exists yet, a cycle-accurate simulator has been developed and targeted by several applications, including a two-dimensional multi-grid solver for the Euler equations[10] and parts of the Gromacs algorithm for protein folding prediction[7]. The Brook language used to implement these applications is essentially ANSI C with a few keyword extensions relevant to stream processors.

Recognising the potential of modern graphics processing units (GPUs) as an alternative target platform to custom-built supercomputers for stream applications, the Stanford graphics group has also developed a subset of Brook called BrookGPU[2]. BrookGPU contains many of the same features as Brook itself, but is limited in some regards by the restrictions of the underlying graphics hardware. Nevertheless, promising results have been achieved by the same group using this combination for image segmentation and linear algebra.

2.2 GPUs as Stream Processors

2.2.1 Motivation for using GPUs

Modern GPUs are designed to accelerate the drawing of three-dimensional graphics. The majority of applications that make use of them are computer games, but CAD applications also make up a significant market. Traditionally, the set of operations supported by the GPU have been fixed by the hardware and limited to simple transformations and lighting instructions. However, the increasing demand for realism in computer games has led to GPUs offering more programmability to support arbitrary graphics operations. Together with the accelerating increase in the processing power of GPUs compared CPUs, this has lead to much interest in using the GPU for general computing purposes. This field is known under the name GPGPU, an acronym for *General Purpose Computing on the GPU*.

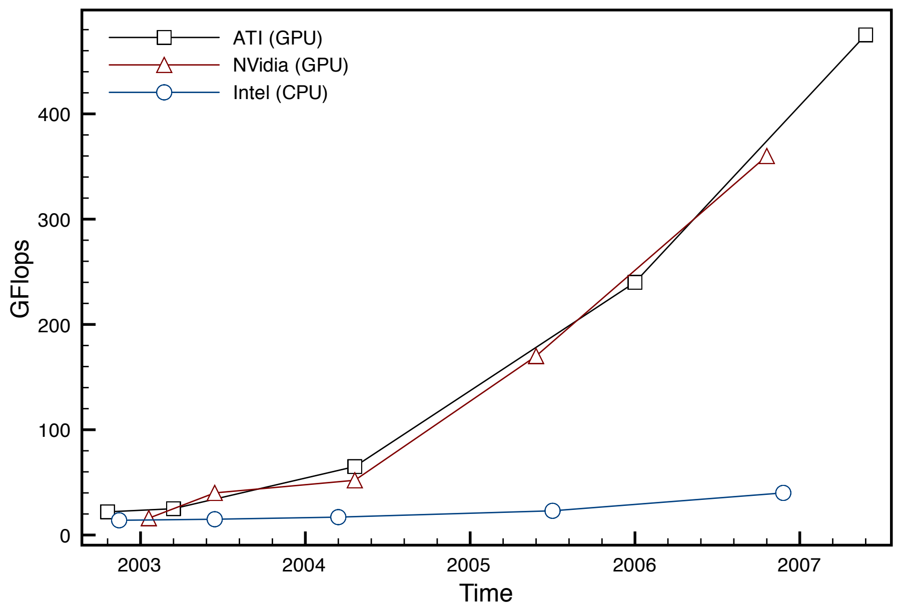

Figure 2: Floating point operations per second for the CPU and GPU.

Figure 3 shows that a recent GPU is significantly more powerful than its CPU contemporary, and that the computing power of GPUs are increasing at a greater rate than that of CPUs. There are two main reasons for this:

1. The GPU is a special-purpose processor. As such, it is able to devote more of its transistors to computation than a CPU which must be capable of running any type of program. Given the similar transistor counts for both chips, GPUs therefore offer more computing power per chip.

2. The GPU employs a parallel architecture so each generation can improve on the speed of previous ones by adding more cores, subject to the limits of space, heat and cost. CPUs, on the other hand, have traditionally used a serial design with a single core, relying instead on greater clock speeds and shrinking transistors to drive more powerful processors. While this approach has been reliable in the past, it is now showing signs of stagnation as the limit of current manufacturing technology is being reached. Recent CPUs therefore tend to feature two or more cores, but GPUs still enjoy a significant advantage in this area for the time being. Some convergence is expected in this area, as indicated by AMD's acquisition of ATI and Intel's announced plans for a many-core CPU/GPU hybrid chip.

Taking advantage of any multi-core architecture requires programs to be written for parallel execution. For computational fluid dynamics, this has traditionally meant splitting the flow domain into several parts that are solved independently on each processor node in a cluster, with the flow properties at boundaries being communicated between the nodes after each time-step. This is also the process adopted for GPUs, but the GPU introduces several additional constraints that make the stream programming paradigm particularly useful. A consideration of the graphics pipeline will show why this is.

2.2.2 The Graphics Pipeline

The graphics pipeline refers to the sequence of operations applied to transform a polygon model of a three-dimensional scene to an image rendered on

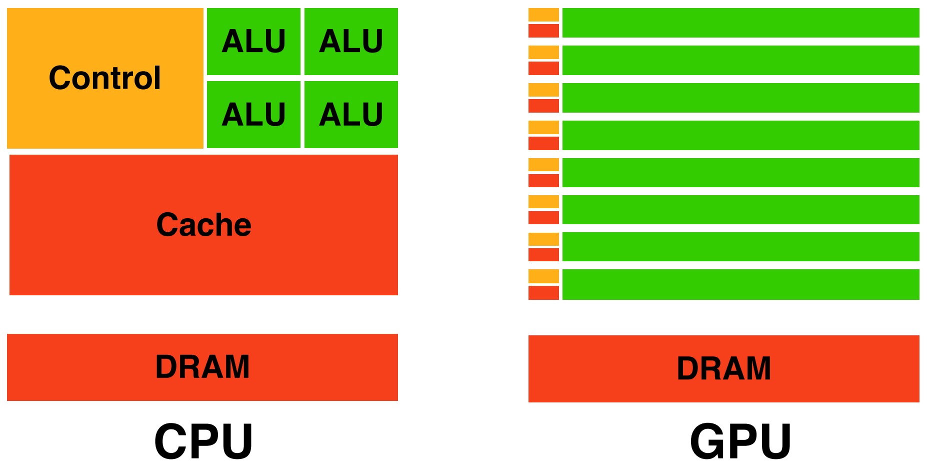

Figure 3: The use of transistors in CPUs and GPUs. An ALU (Arithmetic Logic Unit) is the basic computing unit of processors. Adapted from NVidia.

the screen.

The first stage in the pipeline is the vertex processor which receives a list of interconnected vertices from the CPU. It then applies transformation operations such as scaling and rotation, as well as lighting to determine the colour of each vertex. These are then rasterised to give individual fragments whose final colour is computed by the fragment processor. Rasterisation refers to the process of taking a vector image and turning it into one approximated by fragments (a fragment is the term used to describe a pixel before it is actually rendered on screen). The colour of the fragment is determined by interpolating between the colours of the vertices and possibly fetching values from a texture in memory. In computer games, the texture normally contains artwork that is applied to the surface of objects, but it can hold any 32-bit floating point values.

Finally, the fragments visible from the camera position are selected and displayed on the screen. Alternatively, the fragments can be written back to texture memory for another pass through the pipeline. The overall process is show in figure 4. Note that there are many vertex processors and fragment processors, with the latter normally outnumbering the former.

Both the vertex processor and the fragment processor are programmable

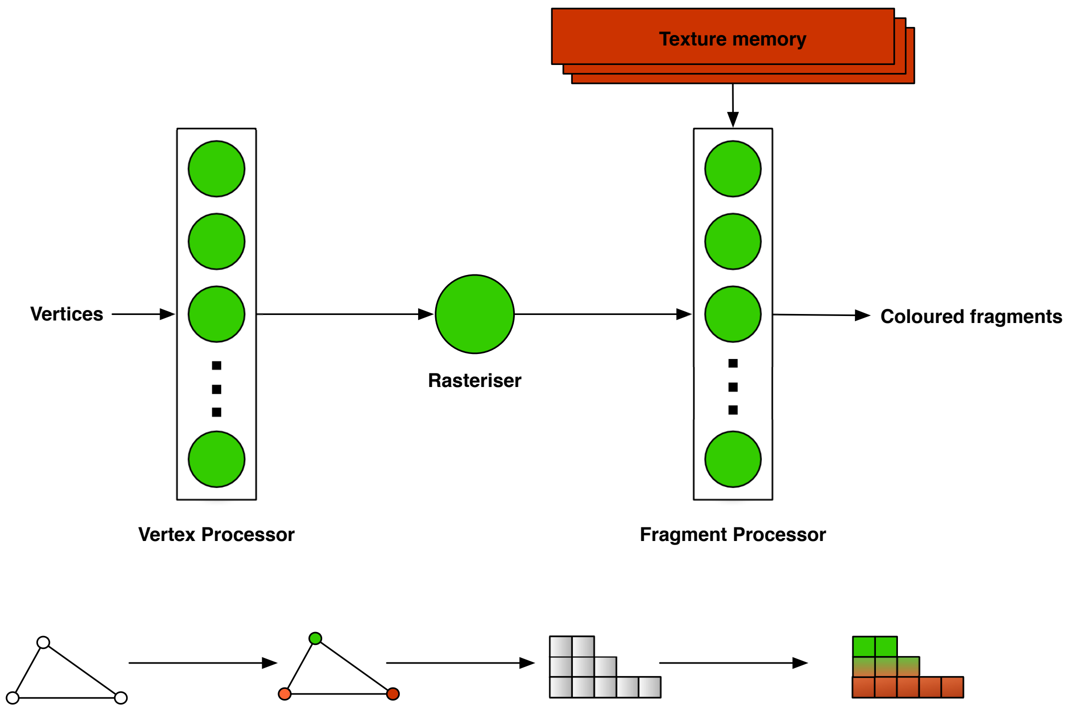

Figure 4: The graphics pipeline.

by the developer. Both also have access to temporary registers for storing the intermediate values of a computation and looking up constants, but only the fragment processor has the ability to look up values in texture memory. In other words, the fragment processor supports *gather* operations (b = a[i]), but does not support *scatter* operations (a[i] = b) since only the output location corresponding to the current input location can be written to. The greater number of fragment processors, combined with their memory access capability, has led most GPU programs to do their computing on the fragment processor instead of the vertex processor.

Most computations are performed as four-component vector operations - this follows naturally from colours being specified as three components and an opacity value. These are carried out with 32-bit floating point precision, but small deviations from the IEEE 754 specification followed by CPUs exist for all boards[15].

The limited control logic of the fragment processor means that support for branching instructions is limited, and such operations are likely to be expensive. This follows from all fragment processors having to apply the same instruction at the same time. If a branching condition is evaluated differently between any processors, both sides of the branch must be evaluated by all processors. Hence, the performance penalty incurred depends on the branch divergence. Loops that can't be unrolled at compile-time also suffer from similar problems.

From a programming perspective, it is the lack of support for scatter operations and the less flexible control capabilities that set the GPU apart from a traditional cluster setup. While the stream programming model makes little mention of branching restrictions, it does encapsulate the no-scatter constraint. In addition, it also provides the benefit of hiding from the developer the splitting of the input stream to the different processors. The graphics pipeline is therefore typically abstracted as a stream computer using the procedure below:

- A single rectangle that covers the whole screen is is issued to the vertex processor. This gets rasterised into thousand to millions of fragments that are fed to the fragment processor.

- A fragment shader program is applied to each fragment independently by the fragment processor. During this process, gather operations are allowed through texture fetches. The mapping at this stage is simple - the fragment processor is equivalent to the stream processor while texture memory is used to store input and gather streams.
- Output streams are simply the output from the fragment processors. These can be displayed on the screen or written to texture memory.

2.2.3 Changes in the latest generation

While the above model of the graphics pipeline is still largely accurate, the latest generation of graphics cards from NVidia and ATI that have come out while this project has been running have introduced some changes. In particular, the distinction between vertex processors and fragment processors has been removed in favour of unified shaders that can act as either type. This is both a simplification of the architecture and a response to different games requiring different amounts of vertex transformations and fragment shading, making a unified pool of computing power more efficient.

In addition, both companies now have low level programming interfaces that offer easier access to the hardware for developers wishing to do general computations on it. The importance of this development will become clear once the challenges of programming the GPU have been discussed.

2.3 Programming the GPU

As with the CPU, there are several approaches to programming the GPU. These differ mainly in how portable they are across different GPUs and operating systems, and the level of hardware abstraction provided. In general, low-level approaches tend to lead to better performance at the expense of portability and development time. However, this trade-off is not linear and good high-level abstractions can both ease the development burden and still offer good performance. The main options available to the GPU programmer are discussed below.

2.3.1 Direct use of Graphics APIs

There are two major programming APIs that are used for 3D graphics, Direct3D and OpenGL. Direct3D is part of Microsoft's proprietary DirectX API collection and is widely used by computer games targeting the Windows and Xbox platforms. OpenGL is a standard specification that has efficient implementations on a variety of platforms, including Windows, Mac OS X, Linux, Unix and the Playstation 3. Like Direct3D, it is widely used by computer games, but is also popular for CAD applications due to its cross-platform availability.

In the context of general purpose programming on the GPU, these APIs enable the developer to allocate memories for textures and control the invocation of shader programs. The shader programs themselves are normally written in a high-level shading language such as NVidia's Cg[19] or Microsoft's HLSL[21]. These interfaces and languages represent the traditional route to programming the GPU, but there are several obstacles associated with this approach. First, the program must be expressed in terms of graphics pipeline operations which generally have little to do with what the program is actually doing. Consequently, the developer must be familiar with the peculiarities of the GPU to develop a working program with good performance. Second, the final program is dependent on GPU drivers and shader compilers that change more frequently and tend to be more unstable than their CPU counterparts. This poses problems for heterogeneous GPU cluster configurations or consumer applications that have to run on a variety of cards and platforms, but is less of an issue if the hardware and software environments can be controlled.

2.3.2 Low-level GPGPU APIs

In response to the growing interest for GPU computing, NVidia and ATI have both released low-level APIs for their latest generation of graphics cards that expose the hardware in a non-graphics manner. ATI's effort is called CTM (Close to Metal). It is a software interface that exposes the GPU in fashion similar to the stream computing model discussed earlier. NVidia's CUDA (Compute Unified Device Architecture) is a C-like programming language with some extensions. It treats the GPU as a collection of independent processors that can be programmed in a fashion somewhat similar to multi-core CPUs.

These technologies bypass the graphics abstraction layer of DirectX and OpenGL, treating the GPU instead as a new type of high-performance processor with its own instruction set. This approach will be more familiar to most developers of scientific applications. One potential problem is that CUDA only works with NVidia GPUs and CTM only works with ATI GPUs. Given that the two companies tend to leap-frog eachother in terms processing power, this might be an issue if one always wants to run on the fastest hardware available.

2.3.3 High-level libraries and languages

A third approach to programming the GPU is to use libraries and languages that provide a higher level of abstraction than the solutions mentioned earlier. Given the complexities of programming with the graphics APIs or CUDA/CTM, it is perhaps not surprising that several such languages exist:

- Accelerator [24] is a library from Microsoft for its C# programming language. It allows the developer access to the GPU through operations on data-parallel arrays.

- Sh[20] is a similar effort developed at the University of Waterloo for the C++ programming language. Although the open-source variant is no longer actively supported, it forms the core of the commercial RapidMind software platform that targets the GPU, multi-core CPUs and the Cell processor.

- BrookGPU[2] from Stanford University provides a few simple extensions to ANSI C that encapsulates the GPU as a stream computer using streams and kernels. It can be compiled into programs that run on most recent NVidia and ATI cards on both the DirectX and OpenGL platforms. An experimental runtime for CTM is also available. Brook is actively developed and there exists a small developer community around the GPGPU.org website that provides help with most issues.

For this project, BrookGPU was chosen because it:

- abstracts away the underlying hardware in a stream programming model, yet is low-level enough to still deliver the speed increase of the GPU;
- has a proven track record of performance and stability as demonstrated by its use in the Stanford Folding@Home GPU client[23];
- works with NVidia and ATI graphics cards on both Windows and Linux;
- is open-source so the inner workings of can be examined.

2.4 Previous work on CFD codes for GPUs

Much work has been done on using the GPU to accelerate the simulation of visually appealing flows for computer games. An example of this is the work of Harris et al.[14] on cloud evolution for flight simulators.

For physically realistic flows, the body of previous work is much smaller. A description of an implementation of the incompressible Euler equations can be found in [22]. For compressible flows, Hagen et al. have implemented a 2D and 3D Euler solver for the GPU[13]. They achieve speed-ups between 10-20x for shock-bubble interactions and Rayleigh-Taylor instabilities. The main difference between their work and the implementation presented in this report is the addition of solid boundaries.

3 Implementation

This section describes the original Fortran implementation of the 2D Euler solver and the strategy used to rewrite it in BrookGPU. A comparison of the results from the two solvers is given for a variety of test cases. Their speed and memory requirements are also analysed, and the implications of the speed-up seen with the GPU solver are discussed.

3.1 Solving the Euler equations

The 2D Euler equations for compressible flow can be expressed in conservation form as below:

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{f}}{\partial x} + \frac{\partial \mathbf{g}}{\partial y} = 0 \tag{1}$$

where $\mathbf{w}$ is the conserved quantities and $\mathbf{f}$ and $\mathbf{g}$ the flux vectors:

$$\mathbf{w} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{pmatrix}, \ \mathbf{f} = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{pmatrix}, \ \mathbf{g} = \begin{pmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vH \end{pmatrix}.$$

As is convention, u and v are the horizontal and vertical velocity components, ρ is the density, p the pressure, H the stagnation enthalpy and E the energy.

When solving these equations on a computer, the flow domain is discretised and the derivatives approximated using finite areas. There are many ways of doing this; the Fortran solver in the present work uses a cell-centred Lax-Friedrichs scheme with vertex storage that is stabilised through smoothing. Denton's Scree scheme [6] is used for the time integration as it operates down to very low Mach numbers and requires low levels of artificial viscosity. Simulations of both duct flows and blade cascades can be done, the latter through the use of periodic boundary conditions. An H-grid is used to represent the geometry in all cases. The cell and vertex numbering scheme is showin in figure 5.

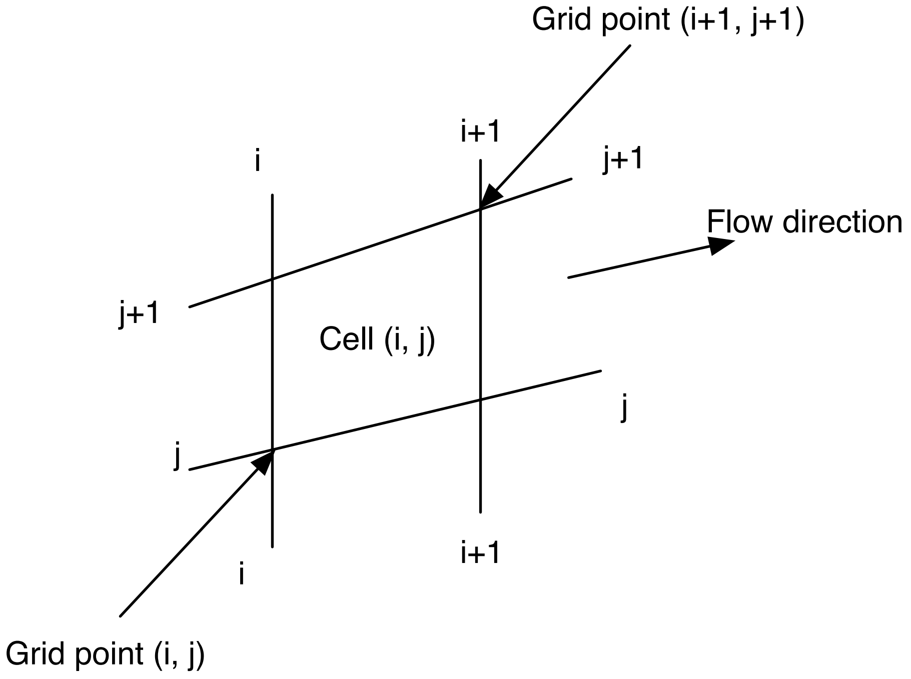

Figure 5: Cell and vertex numbering.

3.2 The Brook solver

Before discussing the details of the Brook solver, it is useful to have an overview of how a typical Brook program is compiled to run on a GPU.

A Brook program in defined in terms of kernels and streams. Brook uses a source-to-source compiler called BRCC to compile the kernels into Cg pixel shaders wrapped by C++ code. The Cg shaders are then further compiled into GPU assembly using NVidia's Cg compiler. The compiled C++ wrappers and GPU assembly run on the Brook Runtime that manages the texture memory and kernel invocation using either DirectX or OpenGL. A GPU emulation back-end that runs on the CPU is also available for testing purposes. The whole process is shown in figure 6.

As an example, consider the kernel *calc_changes* from the Euler solver:

```
kernel void calc_changes(float deltat, float area<>, float4 iflux[][],
                         float4 jflux[][], out float4 delta<>) {
  float2 up = {0, 1};
  float2 right = {1, 0};
  delta = (iflux[indexof delta.xy] - jflux[indexof delta.xy + up] -
        iflux[indexof delta.xy + right] + jflux[indexof delta.xy])*deltat/area;
}
```

The kernel calculates the changes to the flow variables in each cell based on the fluxes into the cell. Some noteworthy points are mentioned below:

- The main body of the kernel is called implicitly for each element in the output stream.
- The area stream is a normal input stream, indicated by the <> braces. This means that only the area corresponding to the current cell is available at any time.
- Due to the cell-corner storage of the flow variables, information is needed from the neighbouring cells when computing the change of the current cell since these store the the fluxes across the top and right faces. The flux streams are therefore defined as gather streams, indicated by the [][] braces. These can be freely indexed into like normal arrays as shown in the calculation in the main body. The indexof operator gives the index of the current stream element, and is used for relative indexing.
- The float4 type declarations refer to floating point vectors of length four. Addition and multiplication operations can be performed on all the values of a float4 at once.

A simplified version of the GPU assembly and the C++ wrapper is shown below:

GPU assembly:

```
static const char* __calc_changes_fp40 = {
"PARAM c[6] = { program.local[0..4],\n"
" { 0, 1 } };\n"
"TEMP R0;\n"
"TEMP R1;\n"
"TEMP R2;\n"
"TEMP R3;\n"
"TEMP R4;\n"
"TEMP RC;\n"
"TEMP HC;\n"
"ADDR  R0.zw, fragment.texcoord[1].xyxy, c[1];\n"
"ADDR  R0.xy, fragment.texcoord[1], c[2].zwzw;\n"
"TEX   R1, R0.zwzw, texture[1], RECT;\n"
"ADDR  R3.xy, R0.zwzw, c[5].yxzw;\n"
"ADDR  R0.zw, R0.xyxy, c[5].xyxy;\n"
"TEX   R2, R0.zwzw, texture[2], RECT;\n"
"ADDR  R1, R1, -R2;\n"
"TEX   R2, R3, texture[1], RECT;\n"
"ADDR  R1, R1, -R2;\n"
"TEX   R2, R0, texture[2], RECT;\n"
"ADDR  R1, R1, R2;\n"
"TEX   R4.x, fragment.texcoord[0], texture[0], RECT;\n"
"MULR  R1, R1, c[0].x;\n"
"RCPR  R0.w, R4.x;\n"
"MULR  result.color, R1, R0.w;\n"
"END \n"
}
```

C++ wrapper:

```
void  calc_changes (const float  deltat,
                  ::brook::stream area,
                  ::brook::stream iflux,
                  ::brook::stream jflux,
                  ::brook::stream delta) {
  static ::brook::kernel  __k(__calc_changes_fp40);
  __k->PushConstant(deltat);
  __k->PushStream(area);
  __k->PushGatherStream(iflux);
  __k->PushGatherStream(jflux);
  __k->PushOutput(delta);
  __k->Map();
}
```

Being able to inspect the Cg assembly is a valuable tool for understanding what is going on behind the scenes of a Brook kernel. Some noteworthy points are mentioned below:

- The shader uses several registers to store intermediate results (six in this case).
- There are five texture fetches in total, two for each of the flux directions and one for the area. Even though the area is just a single float, it is as expensive to fetch as an entire float4.
- Three separate textures are used, one for each flux direction and one for the area.
- The fragment shader cannot to direct division, instead the reciprocal is taken and then used in a multiplication.

The C++ function *calc_changes* can be used in a C++ program that might include other functions that run on the CPU. Data transfer between C++ arrays and Brook streams is handled by read/write functions that can copy data back and forth between main memory and GPU textures.

The full Brook solver includes similar kernels for calculating the fluxes, distributing the cell changes to the corner nodes and smoothing the flow variables. These are roughly equivalent to their Fortran function counterparts. Streams are used throughout instead of arrays.

The conversion from Fortran to Brook is mostly straightforward once the stream programming model is understood. This is in large part because the locality of the Euler equations means that they lend themselves well to parallel execution. Other problems that depend on global information or do not fit neatly into the stream model would require larger algorithmic changes.

One issue that did arise in the conversion was that of boundary conditions. The stream programming model requires that the same operation is performed on every element in the input stream. However, the lack of symmetry at the boundaries requires the derivatives to be discretised differently than in the interior flow. In the Fortran solver, this is solved by conditional branching based on the index of the current cell. Unfortunately, branching is inefficient and in some cases unsupported on GPUs, so another approach called dependent texturing was taken instead.

Dependent texturing refers to the process of using one texture to store the address of a value to be fetched from another texture. This allows the indices to be used in the interior and at the boundaries to be pre-computed before the kernel is invoked, removing the need for branching within the kernel itself. As an example, consider the Fortran function *sum_changes* that calculates the new flow properties at a node based on the flux balances of the surrounding cells:

```
      subroutine sum_changes (delta, prop)

c    Change for interior nodes
      do i=2,ni-1
      do j=2,nj-1
      add = 0.25*(delta(i,j) + delta(i-1,j) + delta(i-1,j-1) +
    &        delta(i,j-1))
      prop(i,j) = prop(i,j) + add
      end do
      end do

c    Change for lower boundary
      do i=2,ni-1
      add = 0.5*(delta(i-1,1) + delta(i,1))
      prop(I,1) = prop(I,1) + add

c    Slightly different sums for upper boundary, inlet/outlet and corners
     .
     .
     .

      return
      end
```

A direct Brook port of this function would include several if-else blocks to test for what sum should be used. Instead, the indices to use in the sum are pre-computed on the CPU:

```
#define INDEX(ni,j,i) ((ni)*(j) + (i))
void set_sum_index(int ni, int nj, Index4 *sum_index) {
  int i, j;
  Index4 index;

  for (i = 1; i < ni-1; i++) {
    // Change for interior nodes
    for (j = 1; j < nj-1; j++) {
      index.i1 = float2(i, j);
       index.i2 = float2(i-1, j);
       index.i3 = float2(i-1, j-1);
       index.i4 = float2(i, j-1);
       sum_index[INDEX(ni, j, i)] = index;
     }
   }
   for (i = 1; i < ni-1; i++) {
     // Change for lower boundary
     index.i1 = float2(i-1,0);
     index.i2 = float2(i-1,0);
     index.i3 = float2(i,0);
     index.i4 = float2(i,0);
     sum_index[INDEX(ni, 0, i)] = index;
   }
   // Slightly different sums of upper boundary, inlet/outlet and corners
     .
     .
     .
 }
```

The *sum_index* array resulting from this is then given to the kernel as a stream alongside the other arguments:

```
kernel void sum_changes(Index4 index<>, float4 primary_old<>,
                     float4 delta[][], out float4 primary<>) {
  primary = primary_old + 0.25f*(delta[index.i1] + delta[index.i2] +
  delta[index.i3] + delta[index.i4]);
}
```

Note that this approach increases the memory required per grid point - a full memory analysis of the solver will be given later. Besides getting rid of branching in the kernel, dependent texturing also simplifies the kernel itself. Its correctness is therefore easier to validate, and most of the debugging effort can be on the CPU side of things where better tools are available.

A similar approach is used in the kernel that calculates the fluxes through each cell face. For the interior flow, linear interpolation between the nodes at either end of the face is used to calculate the flow variables at the center of the cell face. To calculate the mass flux through the j-faces, this approach leads to the following kernel code:

```
mass_flux_j = 0.5f*((primary[indexof flux_j.xy].y +
               primary[indexof flux_j.xy + right].y)*dlj.x +
               (primary[indexof flux_j.xy].z +
               primary[indexof flux_j.xy + right].z)*dlj.y);
```

However, at the solid boundaries the mass flux should be set to zero instead of calculated like above. To avoid conditional brancing, a flux multiplier array is pre-computed on the CPU and given to the kernel just like the index array discussed earlier. For interior nodes, the multiplier is simply a half, but for the solid boundary nodes it is zero. This results in the following modified kernel code:

```
mass_flux_j = mul*((primary[indexof flux_j.xy].y +
              primary[indexof flux_j.xy + right].y)*dlj.x +
              (primary[indexof flux_j.xy].z +
              primary[indexof flux_j.xy + right].z)*dlj.y);
```

As can be seen, the constant multiplier 0.5f from before has been replaced by the variable multiplier mul.

Figure 7 shows a block diagram of the solver, indicating which parts are done on the CPU and which parts are done on the GPU. The middle block shows the arrays and streams (dli and dlj are the projected lengths in the i- and j-directions respectively).

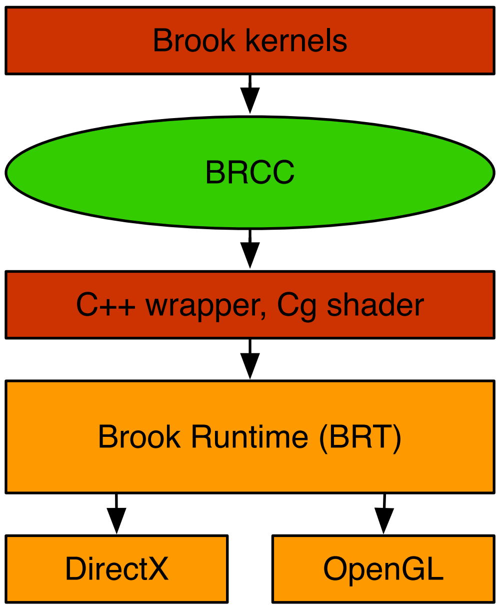

Figure 6: The BRCC source-to-source compiler and Brook Runtime. Adapted from the Brook website.

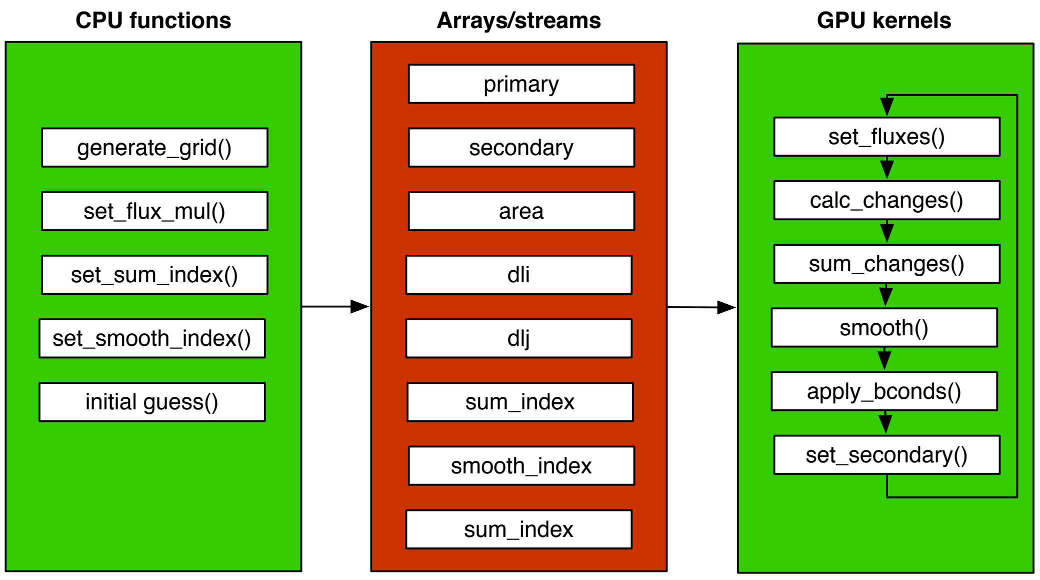

Figure 7: A flow diagram for the GPU solver, showing pre-computation on the CPU and the GPU kernels.

4 Discussion and Results

4.1 Test cases

Several test cases were used to validate the implementation of the GPU solver against the CPU reference implementation. These include sub- and supersonic duct flows, as well as a blade cascade. In all cases, the flow predicted by the GPU was identical to that predicted by the CPU. A selection of test cases is shown below.

4.1.1 A simple sub-sonic nozzle

Figures 8 and 9 show the mach contours predicted by the CPU and GPU solvers for sub-sonic flow through a simple nozzle. The grid uses uniform spacing in the horisontal and vertical directions. The solutions are almost symmetrical about the bump, indicating the ability of the Scree scheme to eliminate most of the artificial viscosity.

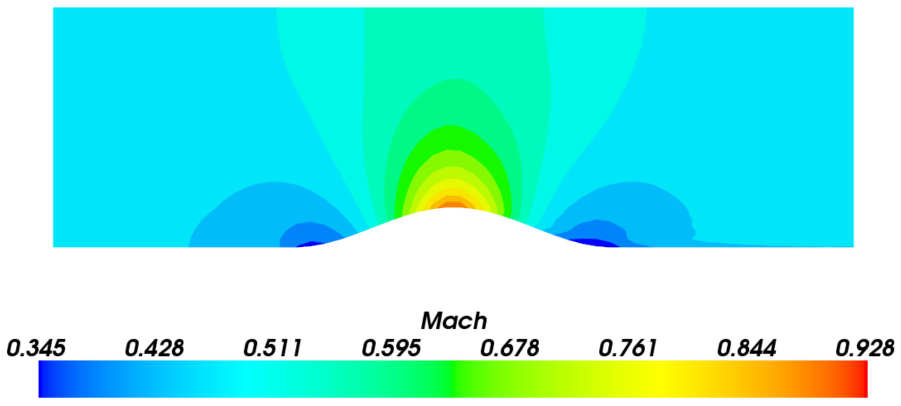

Figure 8: Mach contours for subsonic nozzle predicted by CPU solver.

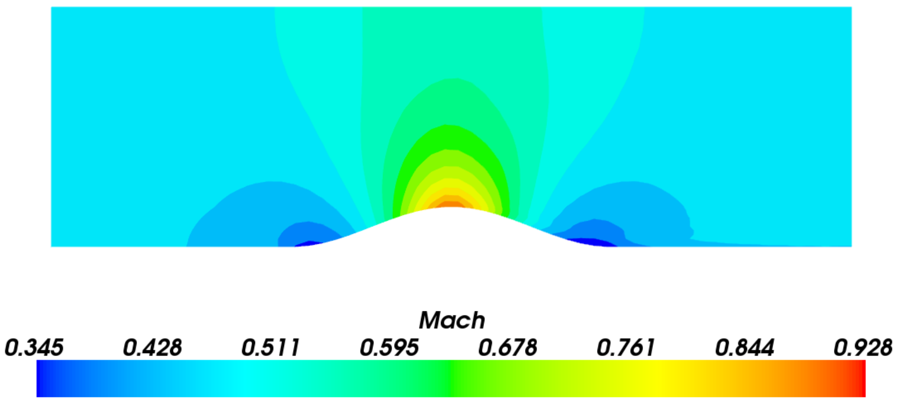

Figure 9: Mach contours for subsonic nozzle predicted by GPU solver.

4.1.2 Turbine blade

The solver is also capable of simulating cascade flows using periodic boundary conditions. Figure 10 shows the mach contours for the turbine blade shape used in[5].

4.1.3 Supersonic wedge

Figure 11 shows the mach contours for supersonic flow over a wedge with an inlet mach number of 1.6. For this flow, there exists an exact analytical solution with complete shock cancellation. The solver comes close to this solution, but the finite grid size smears the shock a bit and results in some reflection.

4.2 Performance

4.2.1 Speed

Figure 4.2.1 shows the time taken for 5000 time-steps for the CPU and the GPU. Two different GPUs were tested, the first a relatively old NVidia 6800

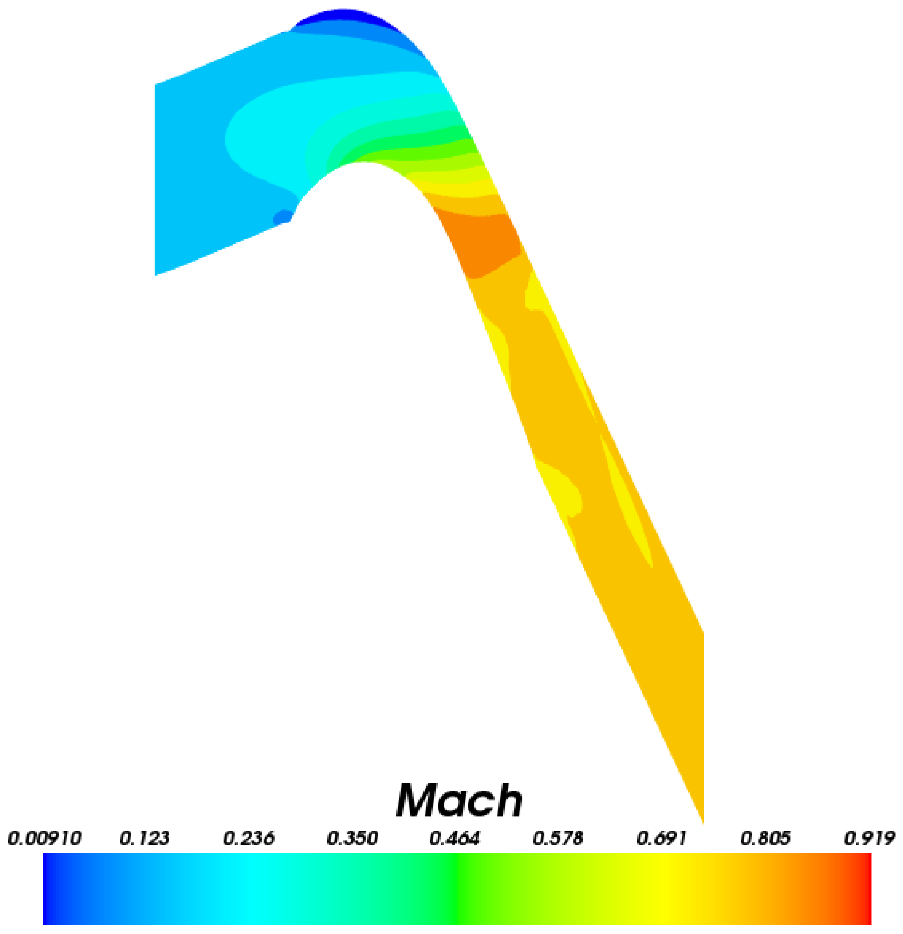

Figure 10: Mach contours predicted by GPU for a blade cascade. The grid used is 60x60, with smaller point separation near the leading edge.

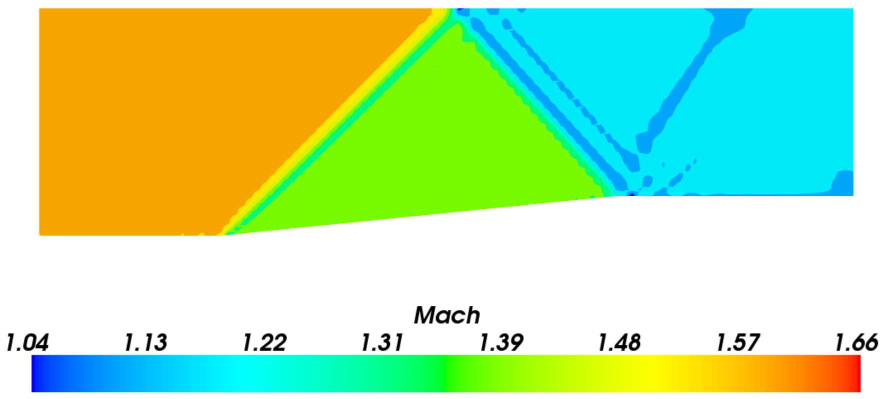

Figure 11: Mach contours predicted by GPU for a supersonic wedge. The grid is 100x100 and uses uniform spacing in both directions.

and the other a recent ATI 1950XT. Compared to the Intel Core 2 Duo, the 6800 offers a 6x speed-up for large grids, while the 1950XT shows a 40x speed-up. Larger speed-ups are seen when compared to the older AMD Opteron.

While the time taken by the GPU solver increases linearly with grid size, the CPU solver slows down for large grids. The reason for this has not been investigated fully, but a likely explanation is that the CPU makes less efficient use of its cache for large grids. The fact that the Duo (2MB cache) is slower than the Opteron (1MB cache) on small grids, but faster on large grids, supports this theory.

A more detailed performance analysis of the GPU solver is possible by inspecting the shader assembly generated by Cg for each kernel. An example of such assembly code has already been given in section 3. Table 1 shows the number of instructions and texture fetches for each kernel, as well as the fraction of the overall run time used by the kernel. The data is for a ATI 1950XT running on a 1000x1000 grid. Note that the most time-consuming kernels (with the exception of *apply_bconds*, a special case because it contains

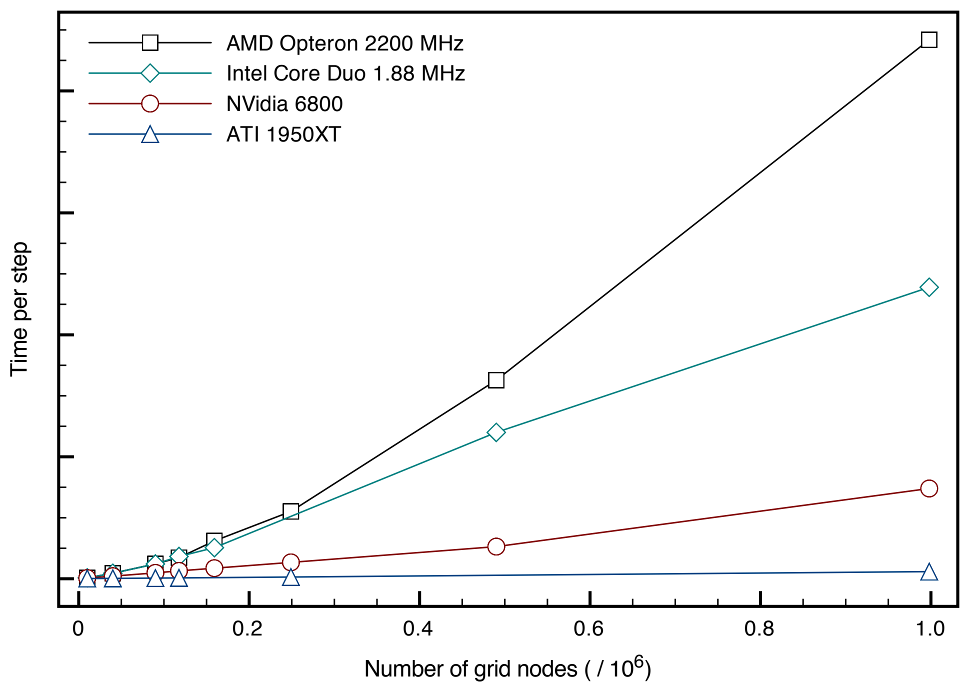

Figure 12: Run time vs. Grid nodes for different CPUs and GPUs.

some conditional branching) are those with the greatest number of texture fetches, indicating that the solver is memory-bound.

Kernel	Run time	Instructions	Fetches
apply_bconds	29%	65	6
sum_changes	27%	39	13
smooth	25%	34	11
calc_fluxes	8%	35	5
calc_changes	7%	26	5
set_secondary	4%	17	1

Table 1: The expected number of cycles used by each kernel.

Using data from the GPU benchmarking suite GPUBench[17] from Stanford, it is possible to estimate that fetching a float4 from texture memory takes approximately 20 cycles, a number that decreases with the number of consecutive fetches (e.g. one fetch takes 20 cycles, but five consecutive fetches take 80 cycles). Clearly, most of the kernels do not make enough computations to hide the cost of these memory fetches. For example, 5000 time steps on the 1000x1000 grid takes 117 seconds. The total number of instructions per time step is 216, so the overall instruction rate is:

$$\frac{216 \cdot 5000 \cdot 1000 \cdot 1000}{117} = 9.2 \cdot \ 10^9 Instructions/s$$

The maximum instruction issue rate for the 1950XT is $30 \cdot 10^9$ Instructions/s, which means that only 30% of the GPU's computational power is being used. The bandwidth used by the solver can be calculated in a similar way by considering the number of texture fetches, giving 28 GByte/s. The actual number is likely to be somewhat lower since the driver will replace some of these texture fetches with cache fetches. The maximum bandwidth of the 1950XT given by GPUBench is 27.3 GByte/s, so the solver is close to maximum bandwidth consumption. This is as expected since the kernels are memory-bound.

Making use of more of the GPU's computational power can be done by increasing the arithmetic intensity of the solver. Including a viscous model would require more computations and go some way towards this goal.

Another approach would be to change the implementation to use an explicit instead of implicit caching scheme, reducing the number of expensive memory fetches needed. This is currently not possible to do using Brook, but NVidia's latest G80 GPU and the CUDA framework provides some of the functionality needed. Given the difficulties of increasing the bandwidth compared to the processing power, it seems likely that hierarchal memory systems with several layers of caching will become more common. Indeed, some of the people behind Brook are involved with another research project called Sequoia[9] that is meant to facilitate programs for such architectures.

4.2.2 Memory

The use of index and multiplier streams described in section 3 results in a higher memory requirement for the GPU solver than the CPU solver. Table 2 shows the size of the arrays and streams used by both implementations and the resulting total memory use. The almost three-fold increase in the memory requirement of the GPU solver reduces the maximum size of the simulations that can be run by the same factor.

Although the values shown are for a 2D solver, the situation for a structured 3D solver would be similar, but with higher overall memory requirements for both the GPU and CPU due to the extra momentum equation and geometry variable. Using a pessimistic estimate of the GPU solver requiring 300 bytes per grid node in 3D, the total number of nodes for a typical 512 Mb graphics card would be 1.7 million. This is still more than sufficient to run a typical 100x100x100 cascade simulation.

It should also be noted that for an unstructured solver, the disparity between the GPU and CPU memory requirements would be smaller. This is because the CPU would be unable to infer boundary locations from a conditional test of the grid index, requiring instead explicit storage of node connectivity.

Array/stream	CPU (bytes)	GPU (bytes)
primary	16	16
secondary	16	16
area	4	4
dli	8	8
dlj	8	8
smooth index	0	52
sum index	0	32
flux multiplier	0	4
Total bytes	52	140

Table 2: Arrays and streams used by the GPU and CPU solver implementations

5 Future directions

5.1 Solver improvements

The GPU solver presented in this report serves as a proof-of-concept that GPUs are capable of running fluid simulations significantly faster than CPUs. However, the solver itself is quite simplistic and its usefulness is limited by being 2D and inviscid. Obvious extensions are therefore the additions of the ability to handle 3D flows and some form of turbulence modeling

Another interesting extension would be to change the solver to work on unstructured grids. Although structured grids work well for simple turbomachinery configurations, there are many other application domains that would require unstructured grids to represent their geometries. Two common data structures used in meshing algorithms for unstructured grids, kd-trees and octrees, have been implemented on the GPU[11][18]. It might therefore be possible to offload some of the normally time-consuming unstructured grid generation process to the GPU.

5.2 GPU Clusters

The memory analysis in section 4.2.2 shows that it is feasible to run simulations on grids with a few million node points on a single graphics card. Larger simulations would require a cluster of GPUs, introducing another memory bottleneck to the system. Buttari et al.[3] discuss this problem in the context of running the SUMMA[12] matrix multiplication algorithm on a cluster of Playstation3 consoles containing Cell processors. Compared to the on-board memory bus found in the Cell or the GPU, the gigabit ethernet connections used in such clusters are relatively low-speed. Following Buttari's procedure, the number of computational operations that must be performed per 32-bit float transferred over the network in order to keep the processors of the ATI 1950XT busy is:

$$\frac{Computational\ power}{Bandwidth} = \frac{250\,GFlops/s}{0.125\,GByte/s} = 8000\,Flops/32\,bits$$

Due to the surface-to-volume effect (the number of transferred values scales as N^2, but the number of computations scales as N^3), such a high ratio is possible if each node has enough memory to hold a sufficiently large data set. For a 512 MByte graphics card operating on a 100x100x100 grid, this results in 800 operations required per float. As shown in section 4.2.1, the GPU solver performs approximately 200 operations per float, but this would increase if expanded to 3D viscous flows . Hence, it seems likely that a cluster of GPUs running a 3D solver would achieve around 50% of the ideal scaling factor of unity.

6 Conclusions

The work presented in this report has shown the feasibility of solving the 2D Euler equations on the GPU. Simulations of cascade flows with periodic boundary conditions have been performed successfully.

The GPU solver running on the ATI 1950XT GPU shows a 40x speed-up over the Fortran reference implementation running on an Intel Core 2 Duo CPU. By analysing the GPU assembly code, it was shown that the GPU solver is memory-bound and only uses around 30% of the computational power of the GPU - indicating the potential for even larger speed-ups for a viscous solver with higher arithmetic intensity.

References

[1] D. Beazley and P. Lomdahl. Feeding a large scale physics application to python, 1997.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[3] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca. A rough guide to scientific computing on the playstation 3. Technical report, Innovative Computing Laboratory, University of Tennessee Knoxville, 2007.

[4] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J-H A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC'03*, Phoenix, Arizona, November 2003.

[5] J. D. Denton. An improved time marching method for turbomachinery flow calculation. *The American Society of Mechanical Engineers*, 1982.

[6] J. D. Denton. The effects of lean and sweep on transonic fan performance. *TASK Quarterly*, pages 7–23, 2002.

[7] Mattan Erez, Jung Ho Ahn, Ankit Garg, William J. Dally, and Eric Darve. Analysis and Performance Results of a Molecular Modeling Application on Merrimac. In *SC'04*, Pittsburgh, Pennsylvaniva, November 2004.

[8] Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[9] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex

Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[10] Massimiliano Fatica, Antony Jameson, and Juan J. Alonso. StreamFLO: an euler solver for streaming architectures. In *IAA paper 2004-1090, 42nd Aerospace Sciences Meeting and Exhibit Conference*, Reno, California, January 2004.

[11] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[12] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[13] 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.

[14] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[15] K. Hillesand and A. Das Lastra. Gpu floating-point paranoia. In *Proceedings of GP2*, 2004.

[16] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[17] Stanford University Graphics Lab. *GPUBench. http://graphics.stanford.edu/projects/gpubench/.*

[18] Aaron Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006.

[19] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM Press.

[20] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[21] Microsoft. *HLSL Reference. http://msdn2.microsoft.com/en-us/library/bb205181.aspx.*

[22] NVidia. *GPU Gems 2.* NVidia, 2004.

[23] Stanford. *Folding@Home on ATI GPU's: a major step forward. http://folding.stanford.edu/FAQ-ATI.html.*

[24] 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 Press.