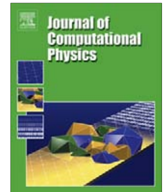
Journal of Computational Physics 227 (2008) 10148–10161

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp



# Large calculation of the flow over a hypersonic vehicle using a GPU

Erich Elsen *, Patrick LeGresley, Eric Darve

*Department of Mechanical Engineering, Stanford University, Durand Building 204, Stanford, CA 94305-4040, USA*

ARTICLE INFO

*Article history:*
Received 10 March 2008
Received in revised form 29 July 2008
Accepted 19 August 2008
Available online 9 September 2008

*Keywords:*
GPU
Brook
Stream processing
Compressible flow
Euler equations

ABSTRACT

Graphics processing units are capable of impressive computing performance up to 518 Gflops peak performance. Various groups have been using these processors for general purpose computing; most efforts have focussed on demonstrating relatively basic calculations, e.g. numerical linear algebra, or physical simulations for visualization purposes with limited accuracy. This paper describes the simulation of a hypersonic vehicle configuration with detailed geometry and accurate boundary conditions using the compressible Euler equations. To the authors' knowledge, this is the most sophisticated calculation of this kind in terms of complexity of the geometry, the physical model, the numerical methods employed, and the accuracy of the solution. The Navier–Stokes Stanford University Solver (NSSUS) was used for this purpose. NSSUS is a multi-block structured code with a provably stable and accurate numerical discretization which uses a vertex-based finite-difference method. A multi-grid scheme is used to accelerate the solution of the system. Based on a comparison of the Intel Core 2 Duo and NVIDIA 8800GTX, speed-ups of over 40× were demonstrated for simple test geometries and 20× for complex geometries.

© 2008 Elsevier Inc. All rights reserved.

## 1. Introduction

In the last 5 years a new generation of processors, often called streaming processors, has become available for compute intensive applications. This new generation includes processors on graphics cards (GPU), ClearSpeed, Cell (IBM–Toshiba–Sony), and Merrimac (Stanford University). Special purpose processors like the GRAPE family have been available for a longer time. All these processors excel at arithmetic operations and boast performance in the range of 200–350 Gflops. As has been realized for some time now, increase in performance of processors becomes increasingly difficult to realize by simply shrinking the size of transistors and increasing the clock frequency. Furthermore, techniques that increase single core performance such as out-of-order execution, pipelining and branch prediction use large amounts of die space and consume significant amounts of power. Still, Moore's law continues to apply: the number of transistors that can be placed on a chip is still increasing exponentially. Currently there are two different techniques to take advantage of all these transistors: CPUs put more traditional cores on a die while GPUs have numerous smaller but less capable cores coupled with a very fast memory system. The NVIDIA 8800GTX for example has 128 scalar processors at 1.35 GHz capable of processing concurrently thousands of threads with a bandwidth to memory of 80 Gbytes/s.

In this paper, we focus on the use of GPUs for scientific codes. Several languages or programming environments have been developed to utilize the computational power of GPUs: Sh (Michael McCool, University of Waterloo), Brook (Pat Hanrahan, Stanford University), CUDA (NVIDIA), CTM (AMD), and RapidMind. GPUs have already been used for many scientific applications but there is still a significant gap between the reality of large scale engineering codes (hundreds of thousands of lines, days or weeks of computer time for a single simulation) and what has been accomplished using GPUs. Demonstrating a

* Corresponding author.
*E-mail addresses:* eelsen@stanford.edu (E. Elsen), plegresley@nvidia.com (P. LeGresley), darve@stanford.edu (E. Darve).

0021-9991/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2008.08.023

complex engineering application remains largely to be seen. In this paper, we present a real engineering flow calculation running on a single GPU with "engineering" accuracy and numerics. It demonstrates the potential of these processors for high performance scientific computing.

## 2. Review of prior work on GPUs

The current state of the art in applying GPUs to computational fluid mechanics is either simulations for graphics purposes emphasizing speed and appearance over accuracy, or simulations generally dealing with 2D geometries and using simpler numerics not suited for complex engineering flows. We now review some previous effort in this direction. The most notable work of engineering significance is the work of Brandvik [1] who solved an Euler flow in 3D geometry.

Krüger and Westermann [2] implemented basic linear operators (vector–vector arithmetic, matrix–vector multiplication with full and sparse matrices) and measured a speed-up around 12–15 on ATI 9800 compared to Pentium 4 2.8 GHz. Applications to the conjugate gradient method and the Navier–Stokes equations in 2D are presented. Rumpf and Strzodka [3] applied the conjugate gradient method and Jacobi iterations to solve non-linear diffusion problems for image processing operations.

Bolz et al. [4] implemented sparse matrix solvers on GPU using the conjugate gradient method and a multi-grid acceleration. Their approach was tested on a 2D flow problem. A 2D unit square was chosen as test case. A speed-up by 2 was measured with a GeForce FX.

Goodnight et al. [5] implemented the multi-grid method on GPUs for three applications: simulation of heat transfer, modeling of fluid mechanics, and tone mapping of high dynamic range images. For the fluid mechanics application, the vorticity–stream function formulation was applied to solve for the vorticity field of a 2D airfoil. This was implemented on NVIDIA GeForceFX 5800 Ultra using Cg. A speed-up of 2.3 was measured compared to an AMD Athlon XP 1800.

In computer graphics where accuracy is not essential but speed is, flow simulations using the method of Stam [6] are very popular. It is a semi-Lagrangian method and allows large time-steps to be applied in solving the Navier–Stokes equations with excellent stability. Though the method is not accurate enough for engineering computation, it does capture the characteristics of fluid motion with nice visual appearance. Harris et al. [7] performed a rather comprehensive simulation of cloud visualization based on Stam's method [6]. Partial differential equations describe fluid motion, thermodynamic processes, buoyant forces, and water phase transitions. Liu et al. [8] performed various 3D flow calculations, e.g. flow over a city, using Stam's method [6]. Their goal is to have a real-time solver along with visualization running on the GPU. A Jacobi solver is used with a fixed number of iterations in order to obtain a satisfactory visual effect.

The Lattice–Boltzmann model (LBM) is attractive for GPU processors since it is simple to implement on sequential and parallel machines, requires a significant computational cost (therefore benefits from faster processors) and is capable of simulating flows around complex geometries. Li et al. [9,10] obtained a speed-up around 6 using Cg on an NVIDIA GeForce FX 5900 Ultra (vs. Pentium 4 2.53 GHz). See the work of Fan et al. [11] using a GPU cluster.

Scheidegger et al. [12] ported the simplified marker and cell (SMAC) method [13] for time-dependent incompressible flows. SMAC is a technique used primarily to model free surface flows. Scheidegger performed several 2D flow calculations and obtained speed-ups on NV 35 and NV 40 varying from 7 to 21. The error of the results was in the range $10^{-2}$–$10^{-3}$. See also the recent review by Owens et al. [14].

The work of Brandvik and Pullan [1] is the closest to our own. They implement a 2D and 3D compressible solver on the GPU in both BrookGPU and Nvidia's CUDA. They achieve speed-ups of 29 (2D) and 16 (3D), respectively, although the 3D BrookGPU version achieved a speed-up of only 3. A finite volume discretization with vertex storage and a structured grid of quadrilaterals was used. No multi-grid or multiple blocks were used.

## 3. Flow solver

The Navier–Stokes Stanford University Solver (NSSUS) solves the three-dimensional Unsteady Reynolds Averaged Navier–Stokes (URANS) equations on multi-block meshes using a vertex-centered solution with first to sixth order finite difference and artificial dissipation operators based on work by Mattson et al. [15], Svärd et al. [16], and Carpenter et al. [17] on Summation by Parts (SBP) operators. Boundary conditions are implemented using penalty terms based on the Simultaneous Approximation Term (SAT) approach [17]. Geometric multi-grid with support for irregular coarsening of meshes is also implemented. The prolongation and restriction operators are the standard full weighting except when irregular coarsening is required and for certain cells the prolongation becomes an injection and restriction is, naturally, the transpose of the prolongation operator. The operators are applied to each block independently.

The SBT and SAT approaches allow for provably stable handling of the boundary conditions (both physical boundaries and boundaries between blocks). The numerics of the code are investigated in the work of Nordstrom et al. [18].

In this work, we focus on a subset of the capabilities in NSSUS, namely the steady solution of the compressible Euler equations which come about if the viscous effects and heat transfer in the Navier–Stokes equations are neglected. Flows modeled using the Euler equations are routinely used as part of the analysis and design of transonic and supersonic aircraft, missiles, hypersonic vehicles, and launch vehicles. Current GPUs are well suited to solving the Euler equations since the use of double precision, needed for the fine mesh spacing required to properly resolve the boundary layer in RANS simulations, is not necessary.

The non-dimensional Euler equations in conservation form are

$$\frac{\partial \mathbf{W}}{\partial t}+\frac{\partial \mathbf{E}}{\partial x}+\frac{\partial \mathbf{F}}{\partial y}+\frac{\partial \mathbf{G}}{\partial z}=\mathbf{0}, \tag{1}$$

where **W** is the vector of conserved flow variables and **E**, **F**, and **G** are the Euler flux vectors defined as:

$$\begin{aligned}
\mathbf{W} &= [\rho, \rho u, \rho v, \rho w, \rho e],\\
\mathbf{E} &= [\rho u, \rho u^2+p, \rho uv, \rho uw, \rho uh],\\
\mathbf{F} &= [\rho v, \rho uv, \rho v^2+p, \rho vw, \rho vh],\\
\mathbf{G} &= [\rho w, \rho uw, \rho vw, \rho w^2+p, \rho wh].
\end{aligned}$$

In these equations, $\rho$ is the density, $u$, $v$, and $w$ are the cartesian velocity components, $p$ is the static pressure, and $h$ is the total enthalpy related to the total energy by $h=e+\frac{p}{\rho}$. For an ideal gas, the equation of state may be written as

$$p=(\gamma-1)\rho\left[e-\frac{1}{2}(u^2+v^2+w^2)\right]. \tag{2}$$

For the finite difference discretization a coordinate transformation from the physical coordinates $(x,y,z)$ to the computational coordinates $(\xi,\eta,\zeta)$ is performed to yield:

$$\frac{\partial \overline{\mathbf{W}}}{\partial t}+\frac{\partial \overline{\mathbf{E}}}{\partial \xi}+\frac{\partial \overline{\mathbf{F}}}{\partial \eta}+\frac{\partial \overline{\mathbf{G}}}{\partial \zeta}=\mathbf{0}, \tag{3}$$

where $\overline{\mathbf{W}}=\mathbf{W}/J$, $J$ is the coordinate transformation Jacobian, and

$$\overline{\mathbf{E}}=\frac{1}{J}(\xi_x\mathbf{E}+\xi_y\mathbf{F}+\xi_z\mathbf{G}),\quad \overline{\mathbf{F}}=\frac{1}{J}(\eta_x\mathbf{E}+\eta_y\mathbf{F}+\eta_z\mathbf{G}),\quad \overline{\mathbf{G}}=\frac{1}{J}(\zeta_x\mathbf{E}+\zeta_y\mathbf{F}+\zeta_z\mathbf{G}).$$

Discretizing the spatial operators results in a system of ordinary differential equations

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{W_{ijk}}{J_{ijk}}\right)+R_{ijk}=0, \tag{4}$$

at every node in the mesh. An explicit five-stage Runge–Kutta scheme using modified coefficients for a maximum stability region is used to advance the equations to a steady-state solution. The maximum stable timestep is computed based upon the restriction of the fastest wave speed in the convection equations, i.e. the spectral radius.

Computing the residual $R$ is the main computational cost; it includes the inviscid Euler fluxes, the artificial dissipation for stability, and the penalty terms for the boundary conditions. The penalty states, obtained either from physical boundary conditions or (for internal block boundaries) from the value of the flow solution in another block, are used to compute the penalty terms.

In the next sections, we describe our implementation of NSSUS for GPUs. This work was accomplished using BrookGPU which is a language that implements the streaming programming model on GPUs. We discuss the algorithms we created to implement NSSUS on the GPU and report numerical results and performance measurements.

## 4. Brook

Brook (also known as BrookGPU) was designed by Ian Buck et al. [19–21]. Brook is a source to source compiler which converts Brook code into C++ code and a high-level shader language like Cg or HLSL. This code then gets compiled into pixel shader assembly by an appropriate shader compiler like Microsoft's FXC or NVIDIA's CGC. The graphics driver finally maps the pixel shader assembly code into hardware instructions as appropriate to the architecture. It can run on top of either DirectX or OpenGL; we used the DirectX backend for all results in this paper.

The syntax of Brook is quite simple. It is based on C with some extensions. The data is represented as streams which are essentially arrays. These streams are operated on by kernels which have specific restrictions: each kernel is a short program to be executed concurrently on each record of the output stream(s). This implies that each instance of a kernel automatically has an output location associated with it. It is this location only to which output can be written. Scatter operations (writing to arbitrary memory locations) are not allowed. Gather operations (read with indirect addressing) are possible for input streams. Here is a trivial example:

```
kernel void add(float a[][], float b[][],out float result⟨⟩) {
  float2 my_index=indexof(result).xy;
  result=a[my_index]+b[my_index];
}
float a⟨100⟩; float b⟨100⟩; float c⟨100⟩;
add(a,b,c);
```

There will be one hundred of instances of the add kernel that are created, essentially implicitly executing a parallel for loop over all the elements of the output steam c. The indexof operator can be used to get the location a particular instance of the kernel will be writing to in the output stream(s).

## 5. Numerical accuracy considerations and performance comparisons between CPU and GPU

Producing identical results in a CPU and GPU implementation of an algorithm is, perhaps surprisingly, not a simple matter. Even if the exact same sequence of instructions are executed on each processor, it is quite possible for the results to be different. Current GPUs do not support the entire IEEE-754 standard. Some of the deviations are not, in the author's experience, generally a concern: not all rounding modes are supported; there is no support for denormalized numbers; and NaN and floating point exceptions are not handled identically. However, other differences are more significant and will affect most applications: division and square root are implemented in a non-standard-compliant fashion, and multiplication and addition can be combined by the compiler into a single instruction (FMAD) which has no counterpart on current CPUs.

```
X=A*B+C;  // FMAD
```

This instruction truncates the result of the intermediate multiplication leading to different behavior than if the operations were performed sequentially [22].

There are other differences between the architectures that can cause even a sequence of additions and multiplications (without FMADs) to yield different results. This is because the FPU registers are 80-bit on CPUs but only 32-bit on current generation GPUs. If the following sequence of operations was performed:

```
C=1E5+1E-5;   // C is in a register
D=10 * C;     // C is still in a register, so is D
E=D - 1E6;    // The result E is finally written to memory
```

On a GPU, E would be 0, while on a CPU it would contain the correct result of .0001. The result of the initial addition would be truncated to 1E5 to fit in the 32-bit registers of a GPU unlike the CPU where the 80-bit registers can represent the result of the addition.

Evaluation of transcendental functions are also likely to produce different results, especially for large values of the operand.

To further complicate matters, CPUs have an additional SIMD unit that is separate from the traditional FPU. This unit has its own 128-bit registers that are used to store either 4 single precision or 2 double precision numbers. This has implications both for speed-up and accuracy comparisons. Each number is now stored in its own 32-bit quarter of the register. The above operations would yield the same result on both platforms if the CPU was using the SIMD unit for the computation.

In addition, by utilizing the SIMD unit, the CPU performs these four operations simultaneously which leads to a significant increase in performance. Unfortunately, the SIMD unit can only be directly used by programming in assembly language or using "intrinsics" in a language such as C/C++. Intrinsics are essentially assembly language instructions, but allow the compiler to take care of instruction order optimization and register allocation. In most scientific applications writing at such a low level is impractical and rarely done; instead compilers that "auto-vectorize" code have been developed. They attempt to transform loops so that the above SIMD operations can be used.

## 6. Mapping the algorithms to the GPU

### 6.1. Classification of kernel types

In mapping the various algorithms to the GPU it is useful to classify kernels into four categories based on their memory access patterns. All of the kernels that make up the entire PDE solver can be classified into one of these categories. Portions of the computation that are often referred to as a unit, the artificial dissipation for example, are often composed of a sequence of many different kernels. For each kernel type we give a simple example of sequential C code, followed by how that code would be transformed into streaming BrookGPU code.

The categories are:

*Pointwise*. When all memory accesses, possibly from many different streams, are from the same location as the output location of the fragment. A simple example of this type of kernel would be calculating momentum at all vertices by multiplying the density and velocity at each vertex. Kernels of this type often have much greater computational density than the following three types of kernels:

```
for (int i=0; i < 100; ++i)
  c[i]=a[i]+b[i];
```

would be transformed into the above add kernel.

*Stencil.* Kernels of this type require data that is spatially local to the output location of the fragment. The data may or may not be local in memory depending on how the 3D data is mapped to 2D space. Difference approximations and multi-grid transfer operations lead to kernels of this type. These kernels often have a very low computational density, often performing only one arithmetic operation per memory load.

```
for (int x=left; x < right; ++x)
  for (int y=bottom; y < top; ++y)
    res[x]=(func[x+1][y]+func[x-1][y]+func[x][y+1] +
      func[x][y-1] - 4*func[x][y])/ delta;
```

would become

```
kernel void res(float delta, float func[][],out float res⟨⟩) {
  float2 my_index=indexof(res).xy;
  float2 up=my_index+float2(0, 1);
  float2 down=my_index - float2(0, 1);
  float2 right=my_index+float2(1, 0);
  float2 left=my_index - float2(1, 0);
  res=(func[up]+func[down]+func[right]+func[left] - 4*func[my_index])/delta;
}
```

*Unstructured gather.* While connectivity inside a block is structured, the blocks themselves are connected in an unstructured fashion. To access data from neighboring blocks, special data structures are created to be used by gather kernels which consolidate non-local information. Copying the sub-faces of a block into their own sub-face stream is a special case of this kind of kernel.

```
kernel void unstructGather(float2 pos[][], float data[], out float reshuffle⟨⟩) {
  float2 my_index=indexof(reshuffle);
  float2 gatherPos=pos[ my_index ];
  reshuffle=data[ gatherPos ];
}
```

The contents of the *pos* stream are indices that are used to access elements of the data stream.

*Reduction.* Reduction kernels are used to monitor the convergence of the solver. A reduction kernel outputs a single scalar by performing a commutative operation on all the elements of the input stream. Examples include the sum, product or maximum of all elements in a stream. Reduction operations are implemented in Brook using efficient tree data structures and an optimal number of passes [19].

### 6.2. Data layout

Because the entire iterative loop of the solver is performed on the GPU, we do not need to be constrained by the data layout the CPU version of NSSUS uses. A one-time translation to and from the GPU format can be done at the beginning and end of the complete solve with minimal overhead. This translation is on the order one second, whereas solves take minutes to tens of minutes.

To lay the 3D data out in the 2D texture memory, we used the standard "flat" 3D texture approach [23] where each 2D plane making up the 3D data is stored at a different location in the 2D stream. This leads to some additional indexing to figure out a fragment's 3D index from its location in the 2D stream (12 flops) and also additional work to convert back 3D indices (9 flops). Data for each block in the multi-block topology is stored in separate streams; the solver loops over the blocks and processes each one sequentially.

### 6.3. Summary of GPU code

A summary of the code execution is shown in Fig. 1. The existing preprocessing subroutines implemented on the CPU are unchanged. Additional GPU specific preprocessing code is run on the CPU to setup the communication patterns between blocks, and the treatment of the penalty states and penalty terms. The transfer of data from the host to the GPU includes the initial value of the solution, preprocessed quantities computed from the mesh coordinates, and weights and stencils used in the multi-grid scheme. Once the data is on the GPU the solver runs in a closed loop. The only data communicated back to the host are the $L_2$ norms of the residuals which are used for monitoring the convergence of the code, and the current solution if the output to a restart file is requested. The number of lines for the GPU implementation is approximately: 4500 lines

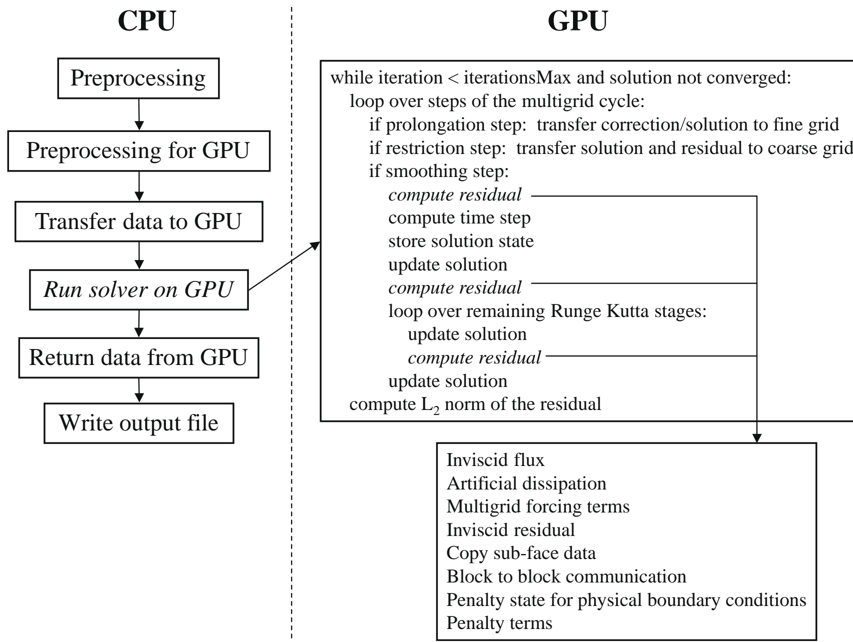



**Fig. 1.** Flowchart of NSSUS running on the GPU.

of Brook code, 8000 lines of supporting C++ code, and 1000 lines of new Fortran code. The original NSSUS code is in Fortran. It took the authors approximately 4 months to develop the necessary algorithms and make the changes to original code.

### 6.4. Algorithms

Constraints from the geometry of the mesh may require that in some blocks, especially at coarse multi-grid levels, the differencing in some directions is done at a lower order than otherwise desired if the number of points in that direction becomes too small. To accommodate this constraint imposed by realistic geometries and also to avoid writing 27 different kernels for each possible combination of order and direction (up to third order is currently implemented on the GPU), we apply all differencing stencils in each direction separately.

The numerics of the code are such that one-sided difference approximations are used near the boundaries of the domain. We designate a boundary point as a point where a special stencil is needed, and interior point as a point where the normal stencil is applied (see Fig. 2). To illustrate the unique stencils created by the SBP approach and to have a concrete example for the following discussion we describe the fourth order stencil:

$$\left.\frac{du}{dx}\right|_0 = -\frac{24}{17}u_0 + \frac{59}{34}u_1 + -\frac{4}{17}u_2 - \frac{3}{34}u_3,$$
$$\left.\frac{du}{dx}\right|_1 = -\frac{1}{2}u_0 + \frac{1}{2}u_2,$$
$$\left.\frac{du}{dx}\right|_2 = \frac{4}{43}u_0 - \frac{59}{86}u_1 + \frac{59}{86}u_3 - \frac{4}{43}u_4,$$
$$\left.\frac{du}{dx}\right|_3 = \frac{3}{98}u_0 - \frac{59}{98}u_2 + \frac{32}{49}u_4 - \frac{4}{49}u_5,$$
$$\left.\frac{du}{dx}\right|_i = \frac{1}{12}u_{i-2} - \frac{8}{12}u_{i-1} + \frac{8}{12}u_{i+1} - \frac{1}{12}u_{i+2}.$$

This distinction presents a problem for parallel data processors such as GPUs because boundary points perform a different calculation from interior points and furthermore different boundary points perform different calculations. This can lead to

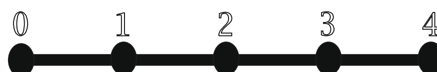



**Fig. 2.** Node 0 is on the boundary. In a fourth-order scheme, nodes 0–3 would be considered boundary points.

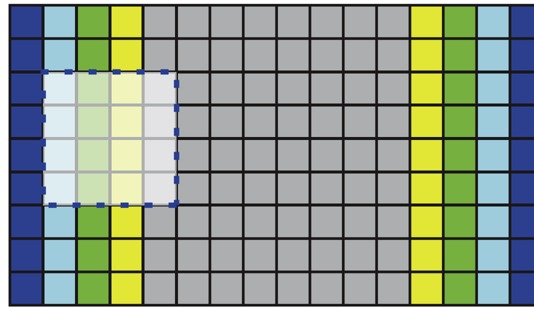


**Fig. 3.** This figure illustrates the stencil in the $x$-direction and the branching on the GPU. Each colored square represents a mesh node. The color corresponds to the stencil used for the node. Inner nodes (in grey) use the same stencil. For optimal efficiency, nodes inside a $4 \times 4$ square should branch coherently, i.e. use the same stencil (see square with a dashed line border). For this calculation, this is not the case near the boundary which leads to inefficiencies in the execution. The algorithm we propose reduces branching and leads to only one branch (instead of 3 here).

terrible branch coherency problems, see Fig. 3. However, regardless of the order of the discretization, the branching can be reduced to only checking if the fragment is a boundary point or not. While the calculation for each boundary point is different, it can be computed as a dot product between stencil coefficients and field values. For example, the zeroth element of the coefficient stream would consist of the values $(-\frac{24}{17}, \frac{59}{34}, -\frac{4}{17}, -\frac{3}{34})$ while another small position stream would hold the values $(0, 1, 2, 3)$ describing the locations in $u$ that correspond to the coefficients.

Thus by using two small 1D streams (that can be indexed using the boundary point's own location), only one branch to determine if the point is a boundary point or in the interior instead of three for each possible stencil is required. (*Note:* we count an `if...else...` statement as one branch.) The exact number depends on the branch granularity of the hardware which is theoretically $4 \times 4$ on the 8800. However, GPUbench [24] suggests that in practice $8 \times 8$ performs better than $4 \times 4$ and $16 \times 16$ even better than $8 \times 8$. For $16 \times 16$, the maximum possible number of branches is 15 but with this particular stencil there are only nine possibilities – one interior point plus four right boundary points and four left boundary points (which can be adjacent due to the flat 3D layout). Our technique reduces this maximum to two, independent of the stencil – one branch for interior points plus one branch for right and left boundary points. Higher order differencing would benefit even more from this technique.

Dealing with the boundary conditions and penalty terms in an efficient manner is significantly more difficult than either of the two previous cases. Fig. 4 shows how sub-faces and penalty terms are computed for each block. The unstructured connectivity between blocks leads to several sub-faces on each block. Each node on the blue block must be penalized against the corresponding node on the adjacent blocks. For example, the node on the blue block located at the intersection of all four green blocks must be penalized against the corner node in each of the four green blocks.

In Brook, it is not possible to stream over a subset of the entries in an array. Instead, we must go through all the $O(n^3)$ entries and use `if` statements to determine whether and what type of calculation need to be performed. This leads to a sig-

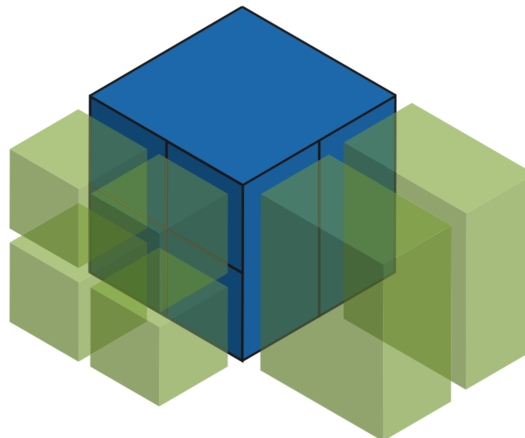


**Fig. 4.** The continuity of the solution across mesh blocks is enforced by computing penalty terms using the SAT approach [17]. The fact that the connectivity between blocks is unstructured creates special difficulty. On this figure, for each node on the faces of the blue block, we must identify the face of one of the green blocks from which the penalty terms are to be computed. In this case, the left face of the blue block intersects the faces of four distinct green blocks. This leads to the creation of four sub-faces on the blue block. For each sub-face, penalty terms need to be computed. Note that some nodes may belong to several sub-faces. (For the interpretation of color in this figure legend the reader is referred to the web version of this article.)

nificant loss of performance since effectively only $O(n^2)$ entries ("surface" entries) need to be operated on. This problem is made worse by the fact that certain nodes belong to multiple faces thereby requiring multiple passes. In order to solve these issues, we decided to copy the sub-face data into one smaller 2D stream (hereafter called sub-face stream); copy data from other blocks if necessary for the internal penalty states (called the neighbor stream). These streams are then used to calculate the penalty state for physical boundary conditions and the penalty terms. This step is computationally efficient since we primarily only process nodes which need to be operated on. This is a strictly $O(n^2)$ step. Finally, we apply the result back into the full 3D stream. This is shown in more details in Figs. 5 and 6.

The copying of the sub-face data into the sub-face stream is done by calculating and storing the location in the full 3D stream from which each fragment in the sub-face stream will gather. The copying of the data from other blocks into the neighbor stream is done by pre-computing and storing the block number and the location within that block from which each fragment in the sub-face stream gathers. This kernel requires multiple blocks as input and must branch to gather from the correct block. This is illustrated by the pseudo-code below, which can be implemented in Brook:

```
kernel void buildNeighborStream(float block1[][], float block2[][], ...,
                                float3 donor_list<>, out float penalty_data<>) {
  block=donor_list.x;
  gather_coord=donor_list.yz;
  if (block==1) penalty_data=block1[gather_coord];
  else if (block==2) penalty_data=block2[gather_coord];
  ...
}
```

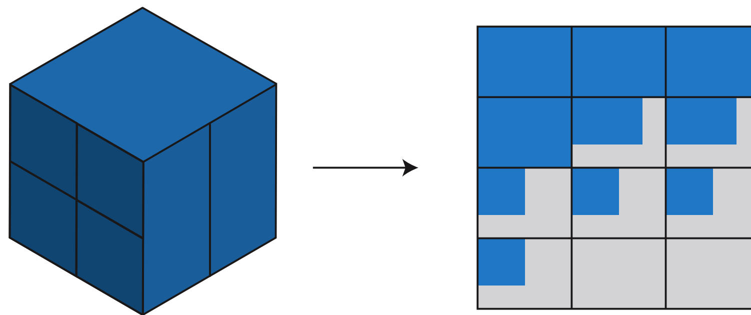


**Fig. 5.** To calculate the penalty terms efficiently for each sub-face, we first copy data from the 3D block into a smaller sub-face stream (shown on the right). In this figure, the block has 10 sub-faces. Assume that the largest sub-face can be stored in memory as a 2D rectangle of size $n_x \times n_y$. In the case shown, the sub-face stream is then composed of 12 $n_x \times n_y$ rectangles, two of which are unused. Some of the space is occupied by real data (in blue); the rest is unused (shown in grey). (For the interpretation of color in this figure legend the reader is referred to the web version of this article.)

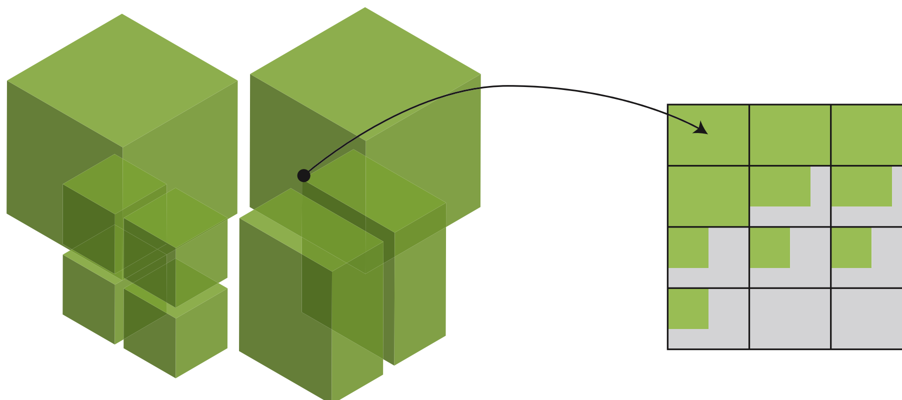


**Fig. 6.** This figure shows the mapping from neighboring blocks to the neighbor stream used to process the penalty terms for the blue block. There are four large blocks surrounding the blue block (top and bottom not shown). They lead to the first four green rectangles. The other rectangles are formed by the two blocks in the front right and the four smaller blocks in the front left. (For the interpretation of color in this figure legend the reader is referred to the web version of this article.)

An important point to make is that this method automatically handles the case of intersecting sub-faces (such as at edges and corners) where multiple boundary conditions and penalty terms need to be applied. In that respect, this approach leads to a significantly simpler code.

## 7. Results

The performance scaling of the code with block size is examined followed by an investigation of the performance of each of the three main kinds of kernels. We, then, examine the performance on meshes for complex geometries typical of realistic engineering problems. In all our tests, the CPU used was a single core of an Intel Core 2 Duo E6600 (2.4 GHz, 4 MB L2 cache) and the GPU used was an NVIDIA 8800GTX (128 scalar processor cores at 1.35 GHz).

For all the results given below, we observed a consistent accuracy compared to the original single precision code in the range of 5–6 significant digits, including the converged solution for the hypersonic vehicle. This is the accuracy to be expected since the GPU operates in single precision. We note that this good behavior is partly a result of considering the Euler equation. The Navier–Stokes equations for example often require a very fine mesh near the boundary to resolve the boundary layer. In that case, differences in mesh element sizes may result in loss of accuracy.

### 7.1. Performance scaling with block size

Fig. 7 shows the scaling of performance and speed-up with respect to the block size. These tests were run on single block cube geometries with freestream boundary conditions on all faces. As the data set becomes larger than the L2 cache, the CPU slows down by a factor of about 2. On the other hand, when the data set increases, the GPU becomes much more efficient, improving by about a factor of 100. The GPU does not reach its peak efficiency until it is working on streams with at least 32,000 elements.

The multi-grid cycle used in these and following tests was a 2 level V cycle. In principle, multi-grid should be used with more than 2 levels but for the compressible Euler equations, the presence of shocks limits the number of grids which can be efficiently used to two. Since our goal is to model a hypersonic vehicle in which shocks are present, we used 2 grids throughout this work even in cases where there is no shock. In 3D, 2 levels require computing on a grid approximately $8\times$ smaller than the original and we know from the single grid results that small grids will be slower than larger ones; consequently, we expect the multi-grid solver to be somewhat slower than the single grid. This is indeed the case. For 512 vertices, multi-grid is about twice as slow. For larger grids, the performance of multi-grid generally follows that of the single grid results but are slightly slower.

### 7.2. Performance of the three main kernel types

The three main different types of kernels have different performance characteristics which will be examined here. For pointwise kernels, we consider the inviscid flux kernel; stencil kernels will be represented by the residual calculation (differencing of the fluxes), and kernels with unstructured gathers by the boundary and penalty terms calculation. We do not examine reduction kernels since they have been studied elsewhere [19] and these kernels are less than 1% of the total runtime.

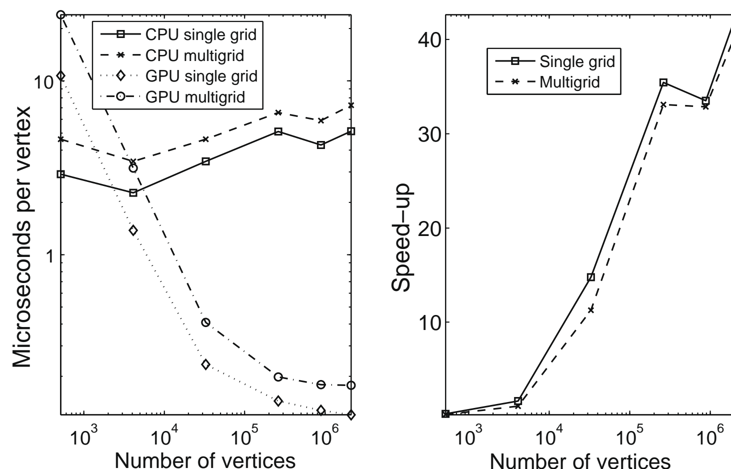



**Fig. 7.** Performance scaling with block size, first order.

Fig. 8 shows that the inviscid flux kernel scales similarly to the overall program (Fig. 7) although with a more marked increase at the largest size. This kernel has an approximately 1:1 ratio of flops to bytes loaded which suggests that it is still limited by the maximum memory bandwidth of the card. Indeed, the largest mesh achieved a bandwidth of 78 Gbytes/s which is nearly the theoretical peak of the card. The achievable memory bandwidth depends not only on the size of the data stream, but also its shape. The second largest mesh has an *x*-dimension that is divisible by 16, whereas the largest mesh has an *x*-dimension divisible by 128. This is the likely reason for the variations between these two stream sizes.

The second type of kernel, the stencil computation, also follows the same basic scaling pattern as the timings in Fig. 7 (Fig. 8). This particular kernel loads 5 bytes for every one (useful) flop it performs. This very poor ratio is due to loading

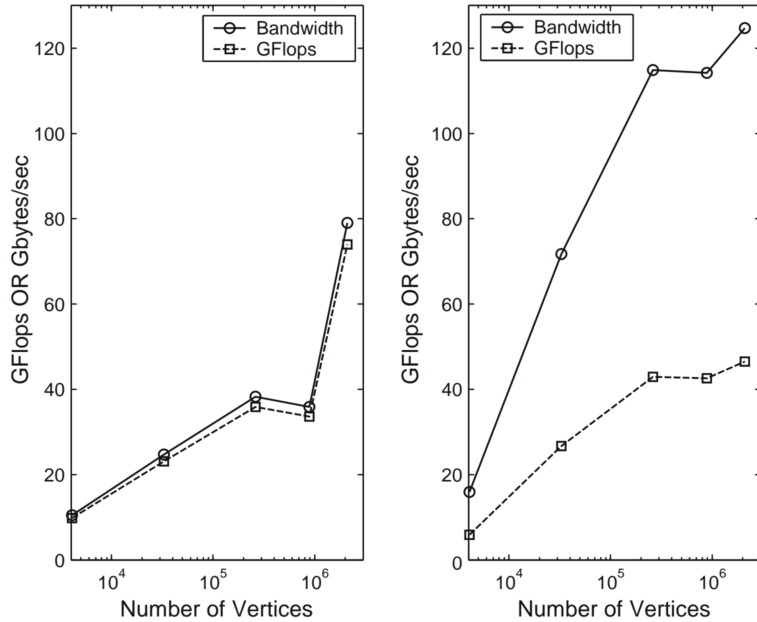



**Fig. 8.** (Left) Pointwise performance (inviscid flux calculation); (right) stencil performance (third-order residual calculation).

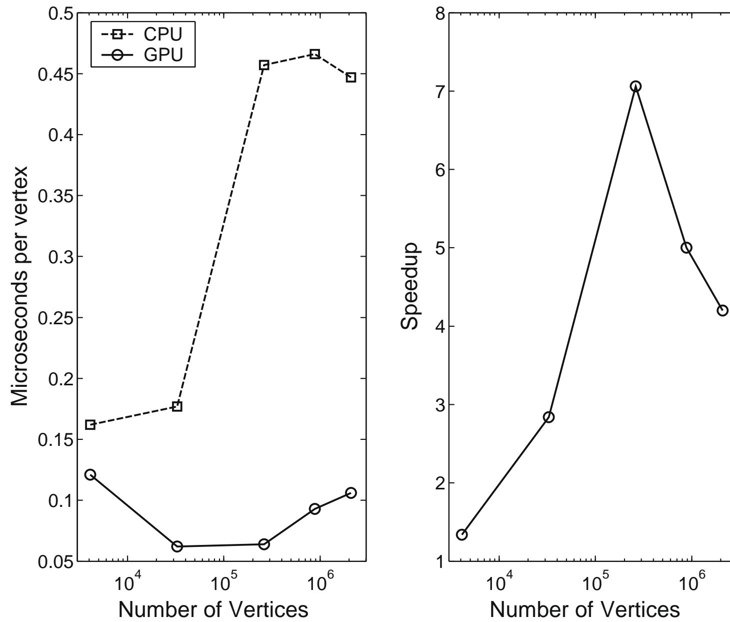



**Fig. 9.** Unstructured gather performance (boundary conditions and penalty terms calculation). The decrease in speed-up is due to an unavoidable $O(n^3)$ vs. $O(n^2)$ algorithmic difference in one of the kernels that make up the boundary calculations. See the discussion in the text.

their location slows the overall computation down. In practice, however, it is unlikely that the size of a given block will be larger than 2 million elements; so in most practical situations, we are in the region where the GPU speed-up is large.

### 7.3. Performance on real meshes

The NACA 0012 airfoil (from the National Advisory Committee for Aeronautics) is a symmetric, 12% thick airfoil, that is a standard test case geometry for computational fluid dynamics codes. Fig. 10 shows the mesh with three blocks used for this simulation (C-mesh topology) and Fig. 11 shows the Mach number around the airfoil.

The CPU code was compiled with the following options using the Intel Fortran compiler version 10: `-O2 -tpp7 -axWP -ipo`.

Table 1 shows the speed-ups for the NACA 0012 airfoil test case. As expected the speed-ups with multi-grid is lower than with a single grid because the computations on the coarser grids are not as efficient. However, over an order of magnitude reduction in computation time is still achieved.

For our final calculations, we used the hypersonic vehicle configuration from Marta and Alonso [25]. This is representative of a typical mesh used in the external aerodynamic analysis of aerospace vehicles. It is a 15-block mesh; two versions were used with approximately 720,000 and 1.5 million nodes. Because the blocks are processed sequentially on the GPU, an important consideration is not only the overall mesh size but the sizes of individual blocks. For the 1.5 million node mesh, the approximate average block size is 100,000 nodes, with a minimum of 10,000 and a maximum of 200,000 nodes. Fig. 12 shows the Mach number on the surface of the vehicle and the symmetry plane for a Mach 5 freestream.

In Table 2, we see the same general trend for speed-ups as the problem size and multi-grid cycle are varied. Beyond just the pure speed-up, it is also important to note the practical impact of the shortened computational time. For example, a converged solution for the 1.5 M node mesh using a 2-grid multi-grid cycle requires approximately 4 CPU hours, but only about 15 min on one GPU!

**Table 1**
Measured speed-ups for the NACA 0012 airfoil computation

| Order | Multi-grid cycle | Speed-up |
|---|---|---|
| First order | Single grid | 17.6 |
| Third order | Single grid | 15.1 |
| First order | 2 Grids | 15.6 |
| Third order | 2 Grids | 14.0 |

Note that higher orders and multiple grids generally lead to faster convergence, so that even though the speed-up is slightly reduced, the wall clock time to reach convergence would also be reduced (for both the CPU and GPU).

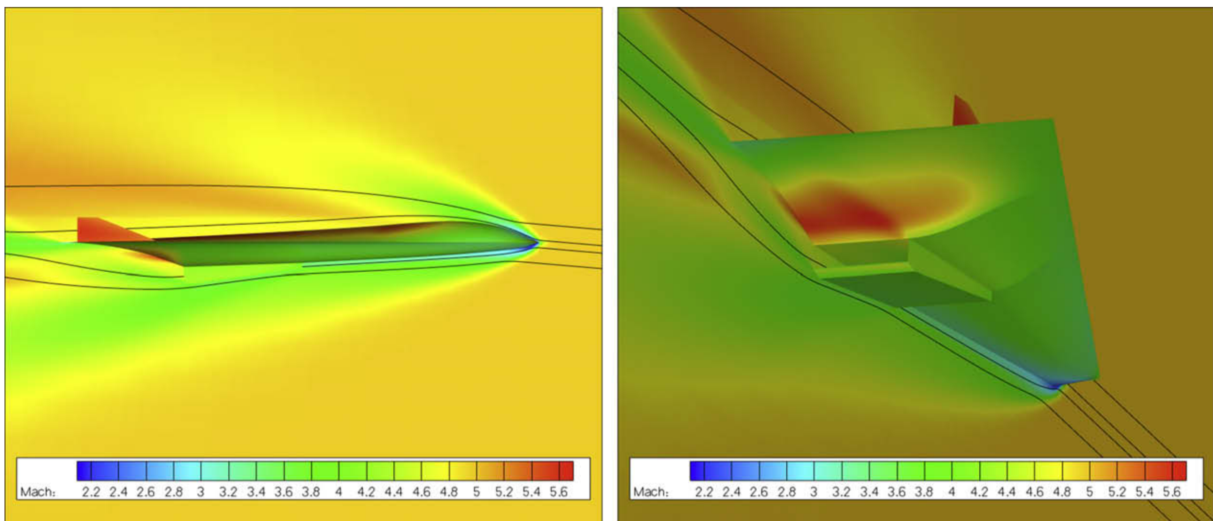



**Fig. 12.** Mach number – side and back views of the hypersonic vehicle.

## 8. Conclusion

We have implemented what we believe to be the most complex scientific simulation yet done on GPUs. Measured speed-ups range from 15$\times$ to over 40$\times$. To demonstrate the capabilities of the code we simulated a hypersonic vehicle in cruise at Mach 5 – something out of the reach of most previous fluid simulation works on GPUs. The three main types of kernels necessary for solving PDEs were presented and their performance characteristics analyzed. Suggestions to reduce branch incoherency due to stencils that vary at the boundaries were made. We have also created a novel technique to handle the complications created by the boundary conditions and the unstructured multi-block nature of the mesh.

Our analysis has identified further ways in which the performance can be improved. Performance on small blocks is

[15] K. Mattson, M. Svärd, M. Carpenter, J. Nordström, Accuracy requirements for transient aerodynamics, in: 16th AIAA Computational Fluid Dynamics Conference, Orlando, FL, 2003.
[16] M. Svärd, K. Mattsson, J. Nordström, Steady-state computations using summation-by-parts operators, J. Sci. Comput. 24 (1) (2005) 79–95.
[17] M.H. Carpenter, D. Gottlieb, S. Abarbanel, Time-stable boundary conditions for finite-difference schemes solving hyperbolic systems: methodology and application to high-order compact schemes, J. Comput. Phys. 111 (2) (1994) 220–236.
[18] J. Nordstrom, E. van der Weide, J. Gong, M. Svard, A hybrid method for the unsteady compressible Navier–Stokes equations, Annual CTR Research Briefs, Center for Turbulence Research, Stanford, 2007.
[19] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, ACM Trans. Graph. 23 (3) (2004) 777–786.
[20] I. Buck, High level languages for GPUs, in: SIGGRAPH '05: ACM SIGGRAPH 2005 Courses, ACM, New York, NY, USA, 2005, p. 109.
[21] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, A. Lefohn, GPGPU: general purpose computation on graphics hardware, in: SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes, 2004, p. 33.
[22] NVIDIA, CUDA Programming Guide 1.1, <http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf> November 2007.
[23] A. Lefohn, GPU data structures, in: GPGPU: General-Purpose Computation on Graphics Hardware Tutorial, International Conference for High Performance Computing, Networking, Storage and Analysis, November 2006.
[24] I. Buck, K. Fatahalian, P. Hanrahan, Gpubench: evaluating GPU performance for numerical and scientific applications, in: Poster Session at GP2 Workshop on General Purpose Computing on Graphics Processors, 2004. URL <http://gpubench.sourceforge.net>.
[25] A. Marta, J. Alonso, High-speed MHD flow control using adjoint-based sensitivities, AIAA paper 2006-8009, in: 14th AIAA/AHI International Space Planes and Hypersonic Systems and Technologies Conference, Canberra, Australia, November 2006.