Acceleration of CFD using CUDA

Student: 4M5ZS

Cavendish Laboratory, Department of Physics, J J Thomson Avenue, Cambridge. CB3 0HE

Abstract

This work presents a comparison of the results obtained when the high-resolution MUSCL-Hancock scheme is implemented for the 2D Euler equations on CPU and GPU, respectively. With an Nvidia A30 GPU, it was found that the GPU sped up the computation time of the test-problems considered by factors of several hundred, while obtaining the same results as on CPU. One dimensional thread-blocks were used for the updates and we find that further work is needed to establish whether this is preferable to two-dimensional blocks.

1. Introduction

In this text we present a comparison of the solution to the 2D Euler equations as numerically computed on CPU versus GPU. Section 2 gives a short outline of the hardware differences between the two, with reference to how this impacts optimal memory access. Carrying on to the physics, section 3 presents the 2D Euler equations for an ideal gas with an outline of the proceedure by which these may be solved using the finite volume method (FVM).

In section 4, a brief review of existing work, in which the 2D Euler equations are solved numerically using GPUs, is provided.

Section 5 gives an outline of the MUSCL-Hancock scheme, the high-resolution numerical solver applied here for the Euler equations. We will also discuss the specifics of its computational implementation for CPU and GPU, respectively.

Section 6 outlines the algorithmic accumulation of the material in the preceeding sections, namely the computational set-up used here to solve the Euler equations on CPU and GPU, respectively. Results obtained here for two test cases from the literature are highlighted in section 7. This section also provides quantitative comparisons of the results obtained on CPU and GPU.

Finally, section 8 gives a detailed comparison of the performance and results obtained when running the initial value problems on CPU and GPU.

Finally, section 5 discusses our results in relation to those obtained in the literature. Limitations and potential extensions to this work are also presented.

2. CPUs and GPUs: Hardware Characteristics and Memory Access

2.1. CPUs

Modern CPUs are typically organised into several cores whose principal physical components are its registers; a control unit (CU); arithmetic units for logic, integer, floating point (and sometimes vector) arithmetic operations; and cache. In the absence of specific instructions to execute on multiple cores, a *program* is executed by a single core given machine code which initially resides in the computer's main memory (RAM). In severely condensed form, the roles of the separate components of the CPU can be summarised as follows:

1. The registers hold information on which the core directly operates. A register is composed of a given number of memory cells, each capable of storing one bit. Collectively, the bits composing a register are known as a *word*. The number of bits in a word corresponds directly to the amount of information that a core can process in each *operation*.
2. CU is responsible for executing incoming machine code instructions and passing or fetching data to and from the registers for further manipulation.
3. Cache, part of which may reside outside the core, stores data and instructions for fast and convenient access by the core. A cache segment is made up of several addressed words.

We shall focus here on the memory architecture of the modern CPU, as this is where it differs most significantly from GPUs. The cache is dedicated to holding data to which the CPU requires fast access, and is partitioned into a hierarchy of L1, L2, and L3 cache. Any

information passed to or retrieved from the computer's main memory (RAM) must pass through the cache hierarchy before reaching a core [1]. The L1 cache is specific to each core and is situated physically within it. Its smaller size, compared to the L2 and L3 caches, makes it easier to address and search through, which further contributes to its speed. The L2 and L3 caches provide further segments for fast memory provision to the core, with the L3 cache typically larger and shared between cores. Thus, access speeds increase chronologically over the L1, L2, and L3 caches.

When a core receives an instruction to fetch data from RAM, the CU instructs a sub-unit known as the *cache-controller* to search for the relevant data in cache. The cache-controller associates a cache address with blocks of RAM using a system of *tags* derived from the former [2]. Searching first in L1 cache, the cache-controller moves to L2 cache and finally L3 cache if there is no hit. If the relevant data are not found in cache it must be fetched from RAM. This is significantly more time-consuming than cache access for a number of reasons: the cache-RAM bandwidth is typically significantly lower than the cache-core bandwidth, the RAM is physically situated further from the CPU, and the data must pass through the cache hierarchy [2].

In practice, when the CPU needs to read several contiguous words from RAM, it batches these instructions and reads from RAM into cache blocks. The width of the blocks that are read simultaneously depends on the given architecture, but once in cache, the CPU has fast access to all data within a given block. A similar process happens for writes to RAM, during which words in a block of contiguous memory in cache can be written simultaneously. Consequently, it is beneficial to structure memory accesses so as to read and write in a contiguous manner from and to RAM.

2.2. GPUs

We discuss here the architecture and memory layout of modern Nvidia GPUs similar to the one (Nvidia A30) on which simulations were done in this work. Nvidia GPUs are structured around the software abstraction of *threads*. Each thread represents a computational unit that can read from and write to the GPUs global memory (similar to RAM on a CPU) and execute functions on the data stored on its registers. Threads are organised into *blocks*, which in turn are organised in a *grid*. Threads in a grid are instantiated when a *kernel* is launched on the GPU. A kernel is a set of arguments and instructions accepted and executed by all threads in the grid. The only information unique to a thread are a set of indices indicating its position within the block and grid, respectively. Any branching behavior (on instructions within the kernel) unique to a cell is ultimately predicated on these indices. Threads in block may communicate by means of a segment known as *shared memory*, whereas threads in different blocks may not directly communicate within the lifetime of a kernel. Shared memory may be allocated dynamically before the launch of a kernel or in static manner directly from the kernel [3]. The instructions in a kernel, the size of the grid, and the number of threads in a block, up to some limit, are controlled by the programmer through Nvidia's GPU API known as CUDA.

At the physical level, modern Nvidia GPUs are organised into streaming multiprocessors (SMs) composed of several GPU cores. Similar to the cores of a CPU, each GPU core has dedicated arithmetic units on which it may execute one operation per clock cycle. However, GPU cores do not have dedicated cache or control unitS. The cache is shared at the level of the SM along with a control unit that fetches instructions for the different cores. When a kernel is launched, instructions are loaded simultaneously to all SMs (or a number sufficient for the threads in a grid). The CU of the SM instructs exactly 32 cores to carry out an instruction for a thread each, simultaneously. 32 threads for which operations are made synchronously are known collectively as a *warp*. The global, cache, and shared memory access architecture is optimised to synchronously fetch or load contiguous memory segments for the threads in a warp [3]. Once a set of instructions has been executed for a warp, the state of the associated threads is saved on SMs *register file* and the cores switch to carry out the same instructions for a warp of 32 different threads [3]. This process, by which cores execute the same instructions for different threads, is known as the single-instruction-multiple-thread paradigm (SIMT). Synchronise statements may be included in the kernels to ensure that specific threads fall into a warp executing memory accesses concurrently. In a manner similar to the memory access pattern for a CPU, we note that it is beneficial to avoid branching on warps during memory accesses, in order that the maximal bandwidth may be achieved. However, GPUs are able to achieve drastic improvements over CPUs by concurrently operating on the fetched data using multiple cores.

3. The 2D Euler Equations and Finite Volume Method

Let ρ, u, v, E, p denote the density, x-velocity, y-velocity, total energy, and pressure, respectively. The

compressible Euler equations in two dimensions may be expressed as:

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho v_x\\ \rho v_y\\ E\end{pmatrix}+\frac{\partial}{\partial x}\cdot\begin{pmatrix}\rho u\\ \rho v_x^2+p\\ \rho v_x v_y\\ v_x(E+p)\end{pmatrix}+\frac{\partial}{\partial y}\cdot\begin{pmatrix}\rho v_y\\ \rho v_x v_y\\ \rho v_y^2+p\\ v_y(E+p)\end{pmatrix}=0,$$

or, more compactly

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

The conservative variables for which the system is expressed are the density (ρ), x-momentum (ρv_x), y-momentum (ρv_y), and total energy (E). For an ideal gas, the total energy and pressure are related by an equation of state given

$$p=(\gamma-1)\Big(E-\frac{(\rho v_x)^2+(\rho v_y)^2}{2\rho}\Big), \tag{2}$$

where γ is the *adiabatic index*, or heat capacity ratio. We take here the conventional choice of $\gamma = 1.4$, which is relatively accurate for primarily diatomic gases. The speed of sound at a point may be found by the relation

$$c_s=\sqrt{\gamma\frac{p}{\rho}}. \tag{3}$$

3.1. FVM

Consider now a uniform Cartesian mesh for which a cell (i, j) is centred at (x_i, y_i), and for which the distance between cell centres is given by Δx and Δy in x- and y-directions, respectively. Denote by t^n a point in time, Δt seconds after which is time t^{n+1}. A finite volume representation of equation (**??**) may be found by first writing

$$\mathbf{u}_{i,j}^n=\frac{1}{\Delta x\Delta y}\int_{y_i-\Delta y}^{y_i+\Delta y}\int_{x_i-\Delta x}^{x_i+\Delta x}\mathbf{u}(x,y,t^n)dx\,dy, \tag{4}$$

$$\mathbf{f}_{i\pm1/2}^n(\mathbf{u})=\frac{1}{\Delta t}\int_{t^n}^{t^{n+1}}\mathbf{f}(\mathbf{u}(x\pm\Delta x,y,t))dt, \tag{5}$$

$$\mathbf{g}_{j\pm1/2}^n(\mathbf{u})=\frac{1}{\Delta t}\int_{t^n}^{t^{n+1}}\mathbf{g}(\mathbf{u}(x,y\pm\Delta y,t))dt. \tag{6}$$

In other words, $\mathbf{u}_{i,j}^n$, is the volume integral of $\mathbf{u}(x, y, t^n)$ over cell (i, j), averaged by the cell's volume. Similarly, the values $\mathbf{f}_{i\pm1/2}^n(\mathbf{u})$ and $\mathbf{g}_{j\pm1/2}^n(\mathbf{u})$ represent the total flux over a cell's face in the time interval $[t^n, t^{n+1}]$, averaged by the time difference Δt.

Integrating equation 1 and using Leibniz's integral theorem, we may then write

$$\mathbf{u}_{i,j}^{n+1}=\mathbf{u}_{i,j}^n-\frac{\Delta t}{\Delta x}\Big(\mathbf{f}_{i+1/2}^n(\mathbf{u})-\mathbf{f}_{i-1/2}^n(\mathbf{u})\Big)-\frac{\Delta t}{\Delta y}\Big(\mathbf{g}_{j+1/2}(\mathbf{u})-\mathbf{g}_{j-1/2}(\mathbf{u})\Big) \tag{7}$$

This is an exact formula for the space-integrated average $\mathbf{u}_{i,j}^{n+1}$. For the sake of symmetry, the finite volume method additionally makes the assumption that this average value is obtained in the relevant variables at the cell's volumetric (areal) centre. The difficulty, then, lies in approximating the time-integrated average fluxes $\mathbf{f}_{i\pm1/2}(\mathbf{u}), \mathbf{g}_{j\pm1/2}(\mathbf{u})$ at cell interfaces using the existing solution $\mathbf{u}_{i,j}^n$.

3.2. Flux Approximation

There are various high resolution schemes for calculating the face-fluxes in the finite volume method. These may be broadly divided as follows:

1. The first set of methods use the available solution $\mathbf{u}^n$ along with some interpolation method Φ to obtain an approximate value at the face from which the flux is approximated as $\mathbf{f}_f^n(\mathbf{u})\approx\mathbf{f}(\Phi(\mathbf{u}^n))$,
2. A second set of methods directly interpolate cell centred fluxes, $\mathbf{f}_f^n(\mathbf{u})\approx\Gamma(\mathbf{f}(\mathbf{u}^n))$,
3. The third paradigm uses a combination of the two.

In this paper we use the MUSCL-Hancock scheme, which falls into the first category. We detail this method in section 5.

3.3. Dimensional Splitting

Once approximate face fluxes $\mathbf{f}_{i\pm1/2}$ and $\mathbf{g}_{j\pm1/2}$ have been found, one may calculate the updated solution $\mathbf{u}^{n+1}$ using equation 7. In a dimensionally unsplit approach, the fluxes $\mathbf{f}_{i\pm1/2}$ and $\hat{\mathbf{g}}_{j\pm1/2}$ are approximated using the same available solution $\mathbf{u}_{i,j}^n$. The problem with such an approach is that, provided both velocity components are non-zero, mass will be transported diagonally on the grid. Thus, for cell (i, j), the fluxes $\mathbf{f}_{i\pm1/2}$ and $\mathbf{g}_{j\pm1/2}$ would have to be approximated by taking into account the solution at cell $(i \pm 1, y \pm 1)$. This constrains the maximal stable time-step and is algorithmically complicated due to the misalignment with the grid. A simpler method is the *dimensionally split* approach. In the split approach, approximate fluxes are calculated first in one direction, using only neighbouring cell values in that direction. Those fluxes are then used to calculate an intermediate solution $\bar{\mathbf{u}}$. Afterwards, fluxes are calculated in the orthogonal direction, again using only neighbouring cell values of $\bar{\mathbf{u}}$ in that direction. Finally, these fluxes

are used to calculate a full update $\mathbf{u}^{n+1}$. If x-values are updated first, this approach may be expressed as:

$$\bar{\mathbf{u}}_{i,j} = \mathbf{u}^n_{i,j} - \frac{\Delta t}{\Delta x}(\mathbf{f}_{i+1/2} - \mathbf{f}_{i-1/2}) \tag{8}$$

$$\mathbf{u}^{n+1}_{i,j} = \bar{\mathbf{u}}_{i,j} - (\mathbf{g}_{j+1/2} - \mathbf{g}_{j-1/2}) \tag{9}$$

Similarly, an update could have been made first in the y-direction. An update may also be obtained by averaging the results from successively updating in the x- and y-directions in various configurations (Strang splitting [4]). In this work use the approach of updating first in the x- direction and then the y-direction.

4. Euler Equations on GPU - a Short Review

To the author's best knowledge, the 2D Euler were first solved numerically using a high resolution scheme on GPUs in 2006 by Hagen, Lie, and Natvig [5]. Since the release of CUDA, in 2007, the ease of implementing general-purpose codes for scientific computations on GPUs has improved significantly.

Several authors have implemented 2D solutions to the Euler equations, and extensions thereof, using CUDA. The author of this text is aware of flux calculation schemes such as MUSCL-Hancock, Lax-Friedrichs, and linear interpolation having been implemented for the Euler equations [6] [7] [8].

For ease of indexing on a mesh, CUDA allows the programmer to structure threads in blocks of up to 3 dimensions. In all of the papers reviewed, the authors decide to organise threads into blocks of 2 dimensions for 2D simulations. Values from a 2D slice of the mesh, with dimensions equivalent to the thread-block, are fetched from global memory such that each thread holds the state of one cell on its allocated registers.

The speed-up, relative to CPU computation, reported by different authors varies widely depending on the test case and grid dimensions. In general however, the authors report increasing relative speed-up as the mesh size increases [7] [8].

5. The MUSCL-Hancock Method

Attributed to Hancock [4], it was decided to use this method to solve the 2D Euler equation-solvers here since an exact Riemann solver may be included in its implementation. This feature enables the method to distinguish all possible wave solutions to the Riemann problem formed when cell values either side of a face are interpolated to it. Furthermore, the method is second order accurate, monotone, and total variation diminishing (TVD) [4]. The latter two features are important qualities. The monotone feature means that the resulting solution will not display oscillations beyond the ranges of its initial values. The TVD property implies that the sum of differences between neighbouring cells is decreasing as the simulation progresses in time. This is expected behavior for most physical systems in which we expect neighbouring points to reach some form of equilibrium.

5.1. Slope Limiting

The MUSCL-Hancock scheme also falls into the category of *slope-limited schemes*. Slope limited schemes aim at constructing states at the back and front faces of a cell, in a given direction, using the difference between its values and those of neighbouring cells. For a linear slope reconstruction, when limiting on a variable q in the x-direction, the parameters

$$\Delta_{i-1/2,j} = \mathbf{q}^n_{i,j} - \mathbf{q}^n_{i-1,j}, \quad \Delta_{i+1/2,j} = \mathbf{q}^n_{i+1,j} - \mathbf{q}^n_{i,j},$$
$$\Rightarrow \Delta_{i,j} = \frac{1}{2}\Big((1+\omega)\Delta_{i-1/2,j} + (1-\omega)\Delta_{i+1/2,j}\Big), \tag{10}$$

are defined, where $\omega \in [-1, 1]$. The value $\Delta_{i,j}$ can be interpreted as a measure of the slope in q in the positive x-direction. Given knowledge of the direction of waves in the simulation, the parameter ω may be used to better approximate the slope given this additional information. In this work, the agnostic choice of $\omega = 0$ has been applied.

After estimation of the slope, a limiting parameter ξ is calculated as a modifying factor on the slope estimate $\Delta_{i,j}$. This purpose of this modifying factor is to revert the slope reconstruction to first order if a discontinuity is likely in the underlying field. In all cases, the inputs to such a calculation is the ratio of differences $r = \Delta_{i-1/2,j}/\Delta_{i+1/2,j}$ and the ratio $\xi_R(r) = 2/(1+r)$. This work has settled on the Superbee limiter [4], defined by

$$\xi(r) = \begin{cases} 0 & \text{if } r \leq 0 \\ 2r & \text{if } 0 < r \leq 0.5 \\ 1 & \text{if } 0.5 < r \leq 1 \\ \min(r, \xi_R(r), 2) & \text{if } r > 1. \end{cases} \tag{11}$$

Slope-limited values at the left (back) and right (front) faces of a cell are then computed as

$$q^{L,n}_{i,j} = q^n_{i,j} - \frac{1}{2}\xi(r)\Delta_i, \quad q^{R,n}_{i,j} = q^n_{i,j} + \frac{1}{2}\xi(r)\Delta_i. \tag{12}$$

If $\Delta_{i-1/2,j}$ and $\Delta_{i+1/2,j}$ have different sign, it indicates that a local extremum in $q(x, y, t)$ may be located in cell

(i, j). Thus, the slope at this cell is forced to zero by the limiter ξ. Very large or small positive values of r suggest a discontinuity, in which case the slope is again perturbed toward zero. For values of r closer to one, the slope reconstruction is left unperturbed.

For a linear reconstruction as above, where only differences with immediate neighbours are used, the reconstructed boundary states $q_{i,j}^{L,n}$ and $q_{i,j}^{R,n}$ are second order accurate in space. However, higher order polynomials could equally be constructed using neighbouring cell values further removed, to achieve estimates of the boundary states with higher accuracy. Equivalent slope limiting is conducted for all variables $q \in \mathbf{u}$.

Having obtained the slope-limited interface states, $\mathbf{u}_{i,j}^{L,n}$ and $\mathbf{u}_{i,j}^{R,n}$, these are advanced half a time-step according to equation (7), applied to the interface states:

$$\mathbf{u}_{i,j}^{L,n+1/2} = \mathbf{u}_{i,j}^{L,n} - \frac{1}{2}\frac{\Delta t}{\Delta x}\Big(\mathbf{f}(\mathbf{u}_{i,j}^{R,n}) - \mathbf{f}(\mathbf{u}_{i,j}^{L,n})\Big) \tag{13}$$

$$\mathbf{u}_{i,j}^{R,n+1/2} = \mathbf{u}_{i,j}^{R,n} - \frac{1}{2}\frac{\Delta t}{\Delta x}\Big(\mathbf{f}(\mathbf{u}_{i,j}^{R,n}) - \mathbf{f}(\mathbf{u}_{i,j}^{L,n})\Big). \tag{14}$$

This step ensures the overall slope limited method if TVD and second order accurate in time [4]. At each cell interface, we now have a Riemann problem for which $\mathbf{u}_{i,j}^{R,n+1/2}$ and $\mathbf{u}_{i+1,j}^{L,n+1/2}$ constitute the left and right states, respectively.

5.2. Riemann Problem

Only at this stage does the flow of operations in the MUSCL-Hancock scheme diverge from those of other slope-limited methods. In MUSCL-Hancock, one attempts to solve the Riemann problem, as mentioned above, directly. (A classifying term for methods that directly solve a Riemann problem, exactly or approximately, at the faces of each cell in a mesh are known as *Godunov-type methods* [4].) While there exist methods that achieve relatively accurate approximate solutions, we describe here the exact solution, as described in [4], for the full Riemann problem resulting from the 2D Euler equations.

At an interface, the left- and right limited states $\mathbf{u}_{i,j}^{R,n+1/2}$ and $\mathbf{u}_{i+1,j}^{L,n+1/2}$ are first converted to primitive variables ρ_L, v_L, v_L^o, p_L and ρ_R, v_R, v_R^o, p_R. In the case of an x-update, we have $v = v_x, v_o = v_y$. For a Riemann problem with two states as above, a *star region* will form, separating a shock and a rarefaction, two shocks, or two rarefactions. All waves start at the interface (corresponding to the initial discontinuity) and propagate out in one direction or the other.

The star region is characterised by star states, v^* and p^*, in velocity and pressure, and partitioned by a *contact discontinuity*, across which only the density jumps. We are interested in the solution to the Riemann problem only at the initial discontinuity (i.e. the interface). To determine the solution at this point, we determine its position relative to the contact discontinuity. Only the star state and the initial state on the corresponding side are necessary to determine the solution at the interface. The following steps provide the remaining details:

1. No analytic formula may be formulated for the pressure, p^* in the star region. It must be found iteratively as the root of the function:

   $$f_R(p^*, \rho_R, v_R, p_R) + f_L(p^*, \rho_L, v_L, p_L) + (v_R - v_L) = 0,$$

   where

   $$f_K = \begin{cases} (p^* - p_K)\sqrt{\frac{2}{\rho_K((\gamma+1)p^* + (\gamma-1)p_K)}} & \text{if } p^* > p_K \\ \frac{2c_{s,K}}{\gamma-1}\Big[\Big(\frac{p^*}{p_K}\Big)^{(\gamma-1)/(2\gamma)} - 1\Big] & \text{if } p^* < p_K. \end{cases}$$

   Given an initial guess for p^*, a Newton-Raphson method is used for this purpose.
2. The velocity v^* in the star region is then found as

   $$v^* = \frac{1}{2}(v_R + v_L + f_R(p^*, \rho_R, v_R, p_R) + f_L(p^*, \rho_L, v_L, p_L)).$$

   This velocity corresponds to the velocity of the contact discontinuity.
3. A coefficient α is set to indicate the side of the interface, relative to the contact discontinuity:

   $$\alpha = \begin{cases} -1 & \text{if } v^* > 0 \\ 1 & \text{if } v^* < 0. \end{cases}$$

   Thus, $\alpha = -1$ if interface lies left of the contact discontinuity and vice-versa.
4. Correspondingly, let

   $$(rho_K, v_K, v_K^o, p_K) = \begin{cases} (\rho_L, v_L, v_L^o, p_L) & \text{if } \alpha = -1 \\ (\rho_R, v_R, v_R^o, p_R) & \text{if } \alpha = 1. \end{cases}$$

5. We are interested now in the wave, whether a shock or rarefaction, laying on the same side of the contact discontinuity as the interface. If $p^* > p_K$, this wave is a shock, and we proceed to step 6. Else, this wave is a rarefaction and we proceed to step 7.
6. The shock velocity, S_K, is defined by

   $$S_K = v_K + \alpha c_{s,K}\sqrt{\frac{(\gamma+1)p^* + (\gamma-1)p_K)}{2\gamma p_K}}.$$

   If $\alpha S_K > 0$, interface lies in star region (between shock and contact discontinuity). Density ρ_K^* in

star region, on same side of contact discontinuity as interface, is found as

$$\rho_K^* = \rho_K\Big(\frac{(\gamma+1)p^* + (\gamma-1)p_K}{(\gamma-1)p^* + (\gamma+1)p_K}\Big) \qquad (15)$$

Thus, **interface takes state** $(\rho_K^*, v^*, v_K^o, p^*)$.

If $\alpha S_K < 0$, the shock front is between the interface and contact discontinuity, such that the **interface takes on values of initial state** $(\rho_K^{'} v_K, v_K^o, p_K)$.

7. If $p^* < p_K$, wave on same side of contact discontinuity as interface is a rarefaction. Inside star region, but on same side of the contact discontinuity as the interface, the density and speed of sound are given by

$$\rho_K^* = \rho_K\Big(\frac{p^*}{p_K}\Big)^{1/\gamma}, \quad c_{s,K}^* = c_{s,K}\Big(\frac{p^*}{p_K}\Big)^{(\gamma-1)/(2\gamma)},$$

respectively. The rarefaction fan will have a head and a tail, whose velocities are defined by

$$S_{HK} = v_K + \alpha c_{s,K}, \quad S_{TK} = v^* + \alpha c_{s,K}^*.$$

If $\alpha S_{TK} > 0$, interface lies between contact discontinuity and rarefaction fan. **Interface then takes on values for corresponding side of star region** $(\rho_K^*, v^*, v_K^o, p^*)$.

If $\alpha S_{TK} < 0 < \alpha S_{TK}$, interface lies inside rarefaction fan. In this case, **the solution at the interface take on the values inside the rarefaction fan**, $(\rho_K^{FAN}, v_K^{FAN}, v_K^o, p_K^{FAN})$, where

$$\rho_K^{FAN} = \frac{\rho_K}{\gamma+1}\Big(2 - \alpha\frac{(\gamma-1)v_K}{c_{s,K}}\Big)^{2/(\gamma-1)}$$
$$v_K^{FAN} = \frac{-2\alpha}{\gamma+1}\Big(c_{s,K} - \alpha\frac{(\gamma-1)v_K}{2}\Big)$$
$$p_K^{FAN} = \frac{p_K}{\gamma+1}\Big(c_{s,K} - \alpha\frac{(\gamma-1)v_K}{2}\Big)^{2\gamma/(\gamma-1)}.$$

Otherwise, $\alpha S_{HK} < 0$, in which case the interface lies on the same side of the rarefaction fan *and* the contact discontinuity. **Interface then takes on values of initial state** $(\rho_K^{'} v_K, v_K^o, p_K)$.

We note, in particular, that the velocities, v_L^o, v_R^o, *parallel to the interface*, do not play a role in determining the type of waves emanating from the interface. Consequently, the solution, v^o, at the interface, **corresponds to the value in the initial state**, $\mathbf{u}_{i,j}^{R,n+1/2}$ **or** $\mathbf{u}_{i+1,j}^{L,n+1/2}$, on the same side of the contact discontinuity as the interface.

Denoting the solutions to the Rimeann solutions "*Rie*", we are left with primitive variable solutions $(\rho^{L,Rie}, v_x^{L,Rie}, v_y^{L,Rie}, p^{L,Rie})$ and $(\rho^{R,Rie}, v_x^{R,Rie}, v_y^{R,Rie}, p^{R,Rie})$ at the left and right interfaces of each cell, respectively.

5.3. Updating the Solution

We may update the solution in a cell using the Riemann solutions at the interfaces in a given direction. First, the primitive variable interface solutions, as above, are converted to conservative variables, giving $\mathbf{u}_{i,j}^{L,Rie,n}$ and $\mathbf{u}_{i,j}^{R,Rie,n}$. Due to the computation of a single solution at each cell interface, we note that

$$\mathbf{u}_{i,j}^{R,Rie,n} = \mathbf{u}_{i+1,j}^{R,Rie,n}, \quad \mathbf{u}_{i,j}^{R,Rie,n} = \mathbf{u}_{i,j+1}^{R,Rie,n}$$

for an update in the x- and y- directions, respectively. If we update first in the x-direction, as in this work, the intermediate state $\bar{\mathbf{u}}_{i,j}$ for each cell (i, j) may be obtained from equation 8 as

$$\bar{\mathbf{u}}_{i,j}^n = \mathbf{u}_{i,j}^n - \frac{\Delta t}{\Delta x}(\mathbf{f}(\mathbf{u}_{i,j}^{R,Rie,n}) - \mathbf{f}(\mathbf{u}_{i,j}^{L,Rie,n}))$$

For a y-update, limited slope reconstructions are computed in that direction from the intermediate solution $\bar{\mathbf{u}}_{i,j}^n$ obtained from an x-update. Based on these values, Riemann solutions are found for the top, $\bar{\mathbf{u}}_{i,j}^{R,Rie,n}$, and bottom, $\bar{\mathbf{u}}_{i,j}^{L,Rie,n}$, faces of a cell. Solutions at time-step $n+1$ may finally be found as

$$\mathbf{u}_{i,j}^{n+1} = \bar{\mathbf{u}}_{i,j}^n - \frac{\Delta t}{\Delta x}(\mathbf{f}(\bar{\mathbf{u}}_{i,j}^{R,Rie,n}) - \mathbf{f}(\bar{\mathbf{u}}_{i,j}^{L,Rie,n})).$$

5.4. The Time Step

In a dimensionally split approach, the Riemann problem at the interface of two cells produces waves emanating out perpendicularly from the interface. While waves may physically travel in any direction, these waves constitute the numerical basis on which a solution is updated in a given direction. For too large a time-step, these "numerical waves" will have time to travel across the length of a cell to a different interface, potentially invalidating the Riemann solution found there, for the time interval $t \in [t^n, t^{n+1}]$. The time-step Δ must be sufficiently small during each iteration of a dimensionally split numerical method to prevent this from happening in either direction of update.

The speed of information travel is the sum of the material velocity and the speed at which pressure disturbances propagate (i.e. the speed of sound c_s) at that

point. Consequently, in 2D systems as those considered here, the maximal wave speed, λ_{max}, over the discrete domain is given

$$\lambda_{max} = \max_{i,j}\Big(|\mathbf{v}_{i,j} + c_{s,(i,j)}\Big). \tag{16}$$

Importantly, the velocities and sound speeds implied by any boundary conditions at the edges of the domain must be included in this computation [4].

To prevent waves emanating from one interface from reaching a parallel interface in front or behind it within the interval Δt, we must have $\lambda_{max}\Delta t \leq \min(\Delta x, \Delta y)$ on a uniform Cartesian grid. Consequently, at the start of each iteration of the MUSCL-Hancock method, we must calculate

$$\Delta t = C\frac{\min(\Delta x, \Delta y)}{\lambda_{max}}, \tag{17}$$

where $C \in (0, 1]$ is the so-called Courant-Friedrichs-Lévy (CFL) number. It is shown in [4] that the a dimensionally split implementation of MUSCL-Hancock scheme is stable for all $C \in (0, 1]$. Consequently, we have used $C = 1$ in this work in order to achieve a maximal time-step Δt for each update.

6. Computational Set Up and its Rationale

The MUSCL-Hancock method with exact Riemann solver and dimensional splitting was implemented in C++ for computation on CPU and GPU, respectively. In both cases, double precision arithmetic was used and the star state for the pressure, p^*, was solved using Newton-Raphson with a tolerance of 10^{-14}. No Riemann problem was considered if the absolute difference in all primitive variables at a reconstructed state were $< 10^{-14}$.

6.1. CPU Code

The computer used had an Intel® Xeon® Silver 4314 CPU and Nvidia A30 GPU. The CPU and GPU codes may be found in the zip-file attached to this report. Instructions for the GPU were written in CUDA and all code was compiled at optimization level 0 using Nvidia's nvcc compiler for consistency.

For the CPU-code, the variable values on the discretised domain were stored using `ArrayXXd` containers included in the Eigen library, automatically manages memory for the arrays. A significant advantage of Eigen over C-style arrays is that it uses expression templates and lazy evaluation when evaluating element-wise operations [9]. The expression templates drastically reduce the overhead of element-wise operations by combining expressions and using vectorization where possible. The lazy evaluation feature enables the compiler to defer evaluation of expressions to the point at which they are needed, thus batching operations, and reducing the number of reads to and from RAM. Furthermore, indexing is simple in Eigen, such that operating on a given row or column of an array may be implemented in an easily readable, yet efficient manner.

Solutions between updates in any direction were stored on the `Eigen::ArrayXXd` members of a `Grid` object. The `Grid` class had methods to set initial values for a given problem, convert between primitive and conservative variables, and to compute the flux on the given variables. A `Solver` class with a function `step` was responsible for carrying out one time-step of the MUSCL-Hancock method using instances of `Grid`. Furthermore, `step` could instantiate objects of type `Slope_Limiter` and `Riemann` (a sublass of `Grid`) to compute reconstructed slopes and Riemann-problem solutions, respectively.

6.2. GPU Code

The CUDA code was implemented using C-style arrays to hold the variable values on the grid. An object of type `CUDA_Grid` managed references to the GPU arays on the host computer. Like the CPU-code, separate variables were stored in separate arrays to allow for coalesced reads of data from RAM to a warps of threads within the kernels. In contrast to the CPU-code, intermediate fields such as the flux, reconstructed slopes, and Riemann problem solutions were not stored as contiguous arrays on the global memory of GPU. Rather, after having read the cell values from memory, a master kernel `register_step` carried out an update in a given direction before writing values back to GPU memory. The name was derived from the effort used to minimise the amount of information held by a kernel at any given time to avoid *spills* to the slower, global memory [3]. Initialisation of solutions were done directly on the GPU, such that a solution was only copied back to the host (CPU) once a simulation was complete. In all kernels, the number of threads called was equivalent to the number of cells in the underlying mesh, such that each thread could alter the values of exactly one cell.

6.3. Block Structure

As mentioned in section 4, the authors of previous works used two-dimensional thread-blocks operating on values from two-dimensional slices of the underlying grid. In this work, the decision was made to read and operate only on a one-dimensional slice from within each block. The reason for this decision requires a brief discussion of the information necessary for a thread to update the values in a cell in a dimensionally split scheme.

6.4. Avoiding Overlaps

If the threads in a block cannot collectively cover an entire row (column) of cells in a uniform mesh during the calculation of fluxes in the x-direction (y-direction), successive blocks must necessarily read some of the same values from the mesh in order that the entire domain be updated. The number of overlapping reads, o, for two blocks may be expressed $o = \text{stencil} - 1$.

Assume we are interested in approximating the fluxes in the x-direction for the cells whose values have been read to a thread-block of width n_x. For a mesh of width N, the number of blocks, X_B, necessary to update all cells in a row is given

$$X_B = \Big(1 + \Big\lceil \frac{N - n_x}{n_x - o} \Big\rceil\Big)\nu, \tag{18}$$

where ν denotes the number of variables in the problem. In a row, it follows the the number of repeated reads is given $(X_B - \nu)o$. Denoting the mesh height by M, the number of duplicate reads, D_x, necessary for an x-update totals then to

$$D_x = M(X_B - \nu)o = M\nu o\Big\lceil \frac{N - n_x}{n_x - o} \Big\rceil. \tag{19}$$

A similar argument for the y-update, during which blocks have height n_y, shows that

$$D_y = N\nu o\Big\lceil \frac{M - n_y}{n_y - 2o} \Big\rceil. \tag{20}$$

The number of duplicate reads is decreasing in the width (height) of the blocks n_x (n_y). In fact, for the case in which $n_x = N$ and $n_y = M$, there will be no overlapping reads. Furthermore, refining the mesh by a factor β, will cause the the number of duplicate reads to grow at a rate of β^2.

In this work, we have followed the approach of letting n_x and n_y be as large as possible. In practice, current Nvidia computers allow a maximal block size of 1024 threads. Consequently, some overlap is necessary in meshes whose dimensions exceed this number. Additionally, limits on the size of shared memory and the number of registers per thread-block may put further restrictions on the possible number of threads in a block for a given numerical scheme.

In this study, meshes of dimension $[200\times200]$, $[400\times400]$, $[600\times600]$ and $[197\times500]$ were used, allowing for a single thread-block to update one entire row or column during x- and y-updates, respectively. In the absence of overlaps, it was also possible to update the solution-arrays in-place, avoiding the added complexity of reading from one array of solutions at a time n and writing the solutions at time $n+1$ into a different one.

6.5. Transposing the Domain

As was mentioned in subsection 2, it is beneficial to read and write to global memory in a coalesced manner. If arrays are stored in row-major order, this is done automatically during an x-update. However, during a y-update, cell values from the same column must be read to and written by adjacent threads of a block, significantly lowering the memory bandwidth. It was thus decided to transpose the underlying variable matrices after each x- and y-update to allow threads-blocks in the succeeding update to read and write in a coalesced manner. The transpose operation requires $2\nu MN$ accesses to memory. However, the standard coalesced approach for transpose on GPU was implemented, by which blocks of dimension (32×32) allow warps to read to and write from contiguous sections of memory [10].

7. Results

The first test-case implemented here is that of case 3 from [11]. On a domain of $[0,1]\times[0,1]$, the initial values were set as in 1. Simulations were conducted with a final time of $t = 0.3$ on grids of sizes $[200\times200]$, $[400\times400]$ and $[600\times600]$ on both platforms. Plots are shown in 1 for the pressures and normalized CPU-GPU differences thereof after completed simulations on the $[400\times400]$ grid. The plots have been overlaid with density contours. Table 2 demonstrates the average times for completion of one time-step and the times for the entire simulation.

	Left $x < 0.5$					Right $x \geq 0.5$			
	ρ	v_x	v_y	p		ρ	v_x	v_y	p
Top	0.5323	1.206	0	0.3	$y \geq 0.5$	1.5	0	0	1.5
Bottom	0.138	1.206	1.206	0.029	$y < 0.5$	0.5323	0	1.206	0.3

Table 1: Initial Conditions for Quadrant Problem from [11]. All values in SI units.

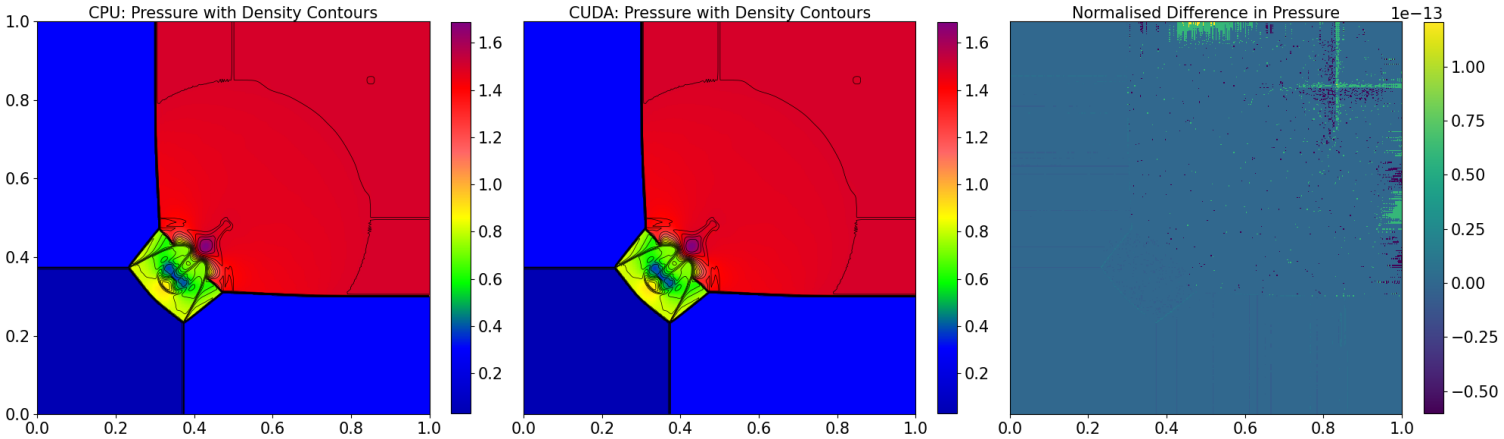

Figure 1: Solution of Quadrant Problem at $t = 0.3$ on CPU and GPU. Normalized difference, $(p_{CPU} - p_{GPU})/(max(p_{CPU}) - min(p_{CPU}))$ on right.

Grid Dim	200×200			400×400			600×600		
	CPU	CUDA	ϕ	CPU	CUDA	ϕ	CPU	CUDA	ϕ
Total	43800111	412634	106.1	378079167	812632	465.3	1322618686	1511350	875.1
Av. Iter.	288679	459.921	627.7	1157920	1473.68	785.7	2597570	2307.74	1125.6

Table 2: Simulation times for CPU versus GPU along with speed-up ratio, ϕ, for quadrant problem. All times given in milliseconds. "Total" indicates total time for simulation with initialisation of solver objects, setting initial values, and copying back to CPU. "Av. Iter" indicates average time, over simulation to $t = 0.3$, for a time-step update on the given platform. Times computed from CPU-side using `chrono`.

The second test-case, from [12], was that of a helium bubble suspended in air at atmospheric pressure being impacted by a shock from the left on a domain of $[0, 0.0025] \times [-0.0445, 0.0445]$. The helium bubble was given an initial radius of 0.025, with its centre at $(0, 0.035)$. The right-moving shock, at a Mach of 1.22, was given an initial location of $x = 0.005$. Initial conditions are shown in 3. (Note that initial values left of the shock were computed analytically using the values in the air to the right and the standard Rankine-Hugionot conditions). The simulation was run on CPU and GPU and sampled at times $t \in [0.000037, 0.000185, 0.000482]$, the pressure results of which, overlaid by density contours, are presented in figure 2 for the GPU along with the normalized CPU-GPU difference at $t = 0.000482$. Note that in [12], the solutions are presented at non-dimensionalised times $\hat{t}$ calculated as the actual time t divided by the time taken to traverse the radius of the helium bubble if traveling at 1.22 Mach. The actual times $0.000037, 0.000185$ and 0.000482 correspond to non-dimensionalised times $\hat{t} = 0.6, 3.0, 7.8$ at which solutions are sampled in [12].

	ρ	v_x	v_y	p
Air Left	1.7755	110.6272	0	159059.99
Air Right	1.29	0	0	101325
Helium Bubble	0.214	0	0	101325

Table 3: Initial Conditions for Bubble Problem from [12]

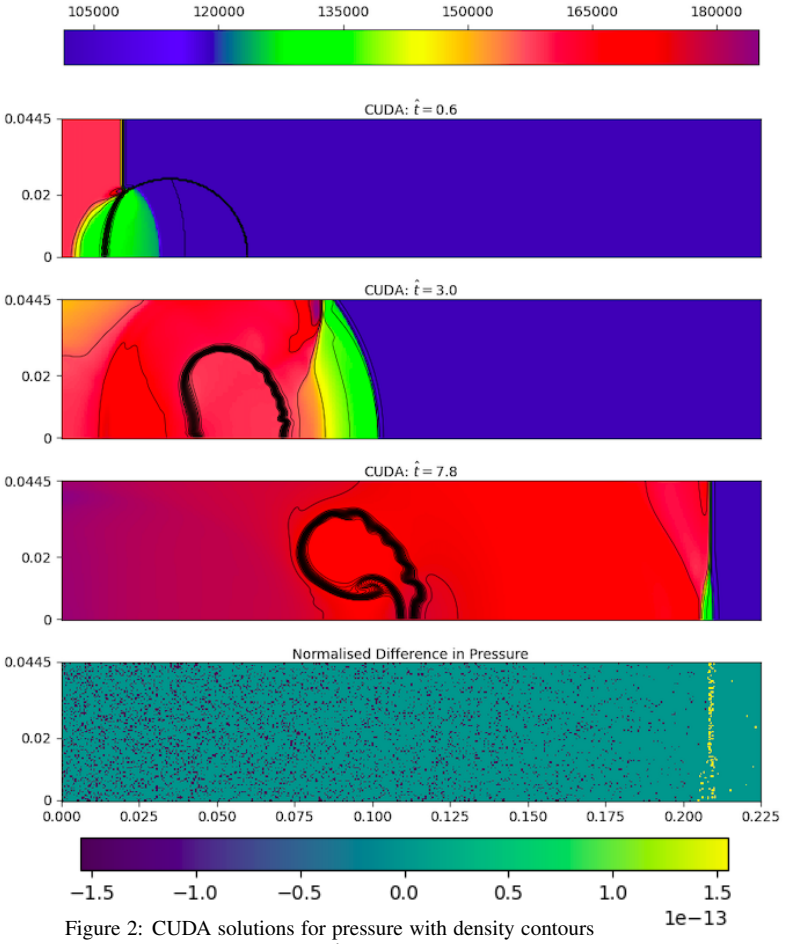

Figure 2: CUDA solutions for pressure with density contours at non-dimensionalised times $\hat{t}$. Bottom shows normalised CPU-GPU difference at $\hat{t} = 7.8$

8. Discussion

As seen from figures 1 and 2, the results for the simulations carried out on CPU and GPU are indistinguishable to the order of 10^{-13} at completion of the test cases. Given speed-ups by factors of several hundred, at least for the quadrant problem, this similarity attests to the power of Nvidia GPUs to produce computational results consistent with CPUs within a much smaller time-span.

A comparison of our results for the quadrant problem with those found in [11] reveal a close correspondence. In particular, our results are nearly indistinguishable from those obtained there using a high order slope-limited reconstruction method, similar to MUSCL-Hancock.

The density contours obtained for the bubble problem in [12] also reveal close similarity with the solutions found there using high-resolution implicit Godunov-type solvers.

8.1. GPU Performance

We note that the the total relative speed-up of the simulations increases non-linearly with increasing grid-size. This is expected given the overhead necessary to copy data back from the GPU to the host, the necessary communication between the two when calculating the time step before each iteration, and the time taken for the CPU to call the different CUDA kernels. For a smaller mesh size, the relative amount of time needed for such tasks is larger compared to bigger mesh sizes, for which the majority of time is spent updating solutions. By consequence, we see lesser changes, as the grid size increases, to the relative speed up for the average iteration time on the two platforms.

The MUSCL-Hancock scheme with exact Riemann solver is a special scheme due to the fact that the number of Riemann problems solved in an iteration depends on the number of cell interfaces at which the absolute difference $|\mathbf{u}^{R,n}_{i,j} - \mathbf{u}^{L,n}_{i+1,j}|$ is sufficiently small. As was mentioned in section 6, a Riemann problem is not computed if the absolute differences in the primitive variables (ρ, v, p) left and right of an interface are all below 10^{-14}. Furthermore, since the Newton-Raphson root-finding algorithm may take many steps to subceed a tolerance of 10^{-14}, the number of floating point operations in a given update is closely dependent on the number of interfaces at which a Riemann problem must be computed. Nvidia GPUs are optimised for SIMT operations on threads in a warp. Divergence in the flow of instructions when some threads in a warp need to compute a Riemann problem and others don't will inevitably lead to thread latency. By contrast, the CPU is able to march through the interfaces one-by-one, computing a Riemann problem only where necessary.

As a consequence of the above, we expect the MUSCL-Hancock scheme to display lesser relative speed up on GPU compared to other schemes in which the number of FLOPS necessary to update the values of a cell are fixed.

8.2. Further Work

A natural extension to this work would be to make the small, albeit necessary, adjustments to the CUDA code in order that Grids larger than 1024 in any given dimension may be accommodated. An optimal improvement under the present paradigm, where one-dimensional rather than two-dimensional blocks are used, would be to allow maximally sized blocks of 1024 threads to update 1024 cells in each iteration, looping back to the start of the next row (column) when the end of a row (column) is reached. Such an adjustment would require some minimal changes to the size of shared memory and the indexing within the `register_step` kernel. A consequent issue in such an approach would be that it is unclear whether the total shared memory and register file of a single multiprocessor would be able to accommodate the memory requirements for 1024 threads, if global memory spills are to be avoided, when using an exact Riemann solver.

Furthermore, it would be beneficial to incorporate a rigorous comparison of the performance of the one-dimensional blocks used here with the two-dimensional blocks used by other authors. This would necessarily imply profiling the time taken for a transpose and comparing it to the speed-up (if any) from using one-dimensional blocks as here.

References

[1] A. Sembrandt, E. Hagersten, D. Black-Schaffer, Data placement across the cache hierarchy: Minimizing data movement with reuse-aware placement, IEEE, 2016.

[2] D. Patterson, Computer Organization and Design RISC-V Edition: The Hardware Software Interface, Morgan Kaufmann, 2020.

[3] Nvidia, Cuda c++ programming guide, Tech. rep., Nvidia (2024).

[4] E. Toro, Riemann Solvers and Numerical Methods for Fluid Dynamics, Springer-Verlag, 2009.

[5] T. Hagen, K. Lie, J. Natvig, Computational Science - ICCS 2006, International Conference on Computational Science, 2006, Ch. Solving the Euler Equations on Graphics Processing Units.

[6] A. Brodtkorp, M. Sætra, Simulating the euler equations on multiple gpus using python, Frontiers in Physics (2022).

[7] F. Wei, J. Liang, L. Jun, F. Ding, X. Zheng, Gpu acceleration of a 2d compressible euler solver on cuda-based block-structured cartesian meshes, Journal of the Brazilian Society of Mechanical Sciences and Engineering (2020).
[8] C. Bard, J. Dorelli, A simple gpu-accelerated two-dimensional muscl-hancock solver for ideal magnetohydrodynamics, Journal of Computational Physics (2013).
[9] Accessed at: https://eigen.tuxfamily.org/dox/TopicInsideEigenExample.html.
[10] M. Harris, An efficient matrix transpose in cuda c/c++, Nvidia Technical Blog (2013).
[11] R. Liska, B. Wendroff, Comparison of several difference schemes on 1d and 2d test problems for the euler equations, Journal of Scientific Computing (2003).
[12] A. Bagabir, D. Drikakis, Mach number effects on shock-bubble interaction, Shock Waves (2001).