university of groningen

A GPU implementation of the 1D and 2D Conservation Element / Solution Element scheme

Frank Westers

January, 2020

Bachelor thesis

Computing Science

University of Groningen

Supervisor 1: Dr. M.H.F. Wilkinson

Supervisor 2: Prof. Dr. A.C. Telea

External Supervisor: Dr. M. Parsani

Abstract

The demand for faster and more efficient algorithms in fluid dynamics is high. The CE/SE scheme is a scheme for simulating flows, which is designed in such a way that many spatial elements can be calculated in parallel. This characteristic makes it very suitable for implementation using parallelism on GPUs. The aim of this thesis was to implement the 1D and 2D CE/SE Euler scheme on a GPU using CUDA. Next, the speed-up compared to the existing CPU algorithms was measured to determine whether it is worth converting more complicated versions of the scheme to GPU algorithms. In this study is was found that the algorithm is considerably faster on GPU than on CPU, respectively 23 times in double precision 1D and 1.5 times faster in 2D. Using single precision, GPU is 4.2 times faster in 2D. Still, by using multiple GPUs, the performance is most likely to increase even more. Therefore, the work presented in this thesis a starting point for further implementations of the CE/SE scheme on GPUs.

Contents

1. Introduction

1.1 Background

For ages, there has been a clear way of doing science: first, someone observed a particular phenomenon. Based on the observations, someone creates a model and tests if the predictions of the model are correct. Based on these tests, the model is adapted to give more accurate results. This process goes on until a model explains the observed phenomenon well enough. Very quickly, however, models became too complex to calculate by hand. Only very simple cases could be addressed using these models. For complex cases, a more simple model would be applied for analysis. The rise of computers changed this drastically. It became possible to do larger computations than ever before. New algorithms were designed for solving complex equations, and a whole new branch of physics came into existence: computational physics.

The above story has been particularly true for fluid physics. In 1822, the Navier-Stokes equation was derived, which describes the motion of viscous fluids. Only a few simple cases can be solved analytically. For more complex geometries, it has to be solved be solved numerically. The brache of physics concerned with numerically analyzing the physics of fluids is called computational fluid dynamics (CFD).

In 1995, a new method for solving the Navier-Stokes equation in 1 dimension was proposed by Sin-Chung Chang in Chang (1995). This new algorithm was called Conservation Element Solution Element (CE/SE). Chang et al. (1999) extended the algorithm to 2 dimensions a few years later and it has proven to be a useful method for simulating viscous fluid flows. The 1D version of the CE/SE algorithm has been implemented in C++ and Python in Shen et al. (2015) and Chena et al. (2011). A continuous search for faster and more efficient implementations is required, due to a high demand for solutions to more complex problems. To meet this demand, new algorithms are needed. The use of General-purpose GPU's (GPGPU) is a promising complement to

the existing CPU algorithms and provides new ways of implementing the CE/SE algorithm.
GPU computing has gained popularity in the past years. Instead of a few cores, which are optimized for sequential processing, GPU cards have thousands small cores optimized for parallel processing. In the CE/SE algorithm, space is divided into discrete pixels. The pixels describe the flow in the space they resemble. In each time step, the pixels are updated to resemble values of the next time step. As these calculations are only dependent of its neighbors in the previous time step, it provides excellent opportunities for parallelization. Since the number of pixels is high and the calculations are not very complex, it is likely that GPU computing will give a huge speed-up.
This project consists of implementing the one- (1D) and two-dimensional (2D) CE/SE algorithm on a GPU. The focus will be on increasing performance by optimizing memory usage and thread distribution. Secondly, a performance analysis will be executed to assess the quality of both GPU algorithms.

1.2 Objectives

The main objectives of this thesis are to

- Implement the 1D version of the CE/SE algorithm (described in section 2 in Chang et al. (1999)) on a GPU. The sequential C++ implementation described in Shen et al. (2015) is provided as a start.
- Implement the 2D version of the CE/SE algorithm (described in section 3, 4 & 5 in Chang et al. (1999)) on a GPU. A sequential C++ implementation and an OpenMP implementation are provided as a basis and are described in Shen et al. (2015).
- Assess the quality of the output of the GPU implementation by comparing the output with the existing implementations.
- Assess the performance of the GPU implementation by comparing the runtime with the existing implementations.

1.3 Structure of the thesis

The next chapter, chapter 2, will discuss the literature that has been written on this topic and which will be used in this thesis. To understand the implementation of the CE/SE scheme, basic knowledge about the scheme itself is required. Therefore, chapter 3 is devoted to giving a quick introduction into the scheme. Chapter 4 describes the actual implementation of the algorithm. This consists of the algorithm in pseudo-code and describes the possibilities for optimization as well. This implementation will be compared to its CPU counterpart in chapter 5. All of the chapters consists of two sections: one for the one-dimensional version and one for the two-dimensional version.
In the end, chapter 6 summarizes the conclusions of the research and provides recommendations for future work on this subject.

2. Literature survey

The CE/SE method is a relatively new technique for solving the Euler equation, and only a few people are working on this topic. This explains why many of the papers that will be cited in this thesis have been written by the same group of authors. Chang (1995) was the first to describe the Conservation Element Solution Element algorithm in 1995. Since then, it has been extended to more dimensions and improved by using different geometries, for example in Chang et al. (1998) and Chang et al. (1999). A C++ implementation in 1D and 2D was described in Shen et al. (2015). This implementation will be the main source for implementing the CE/SE scheme in CUDA.

A GPU implementation of the CE/SE method has been proposed earlier in Wei Ran et al. (2011). However, the code proposed in this algorithm is not in line with the original algorithm and has likely some possibilities for improvement. First of all, equation (4) in Wei Ran et al. (2011) is different from the standard CE/SE algorithm. In this equation, u_m is used to calculate a value f_m. According to the CE/SE algorithm, the spatial derivative of u_m should be used instead. Therefore the output will be different. Also, the output is measured at $t = 40$ according to figure 9. Most likely, this should be $t = 0.4$ seconds, because, after $40s$, the system should have reached equilibrium. All the symbols used above will be explained in the next chapter.

A way to improve performance is increasing the number of parallel threads. In Wei Ran et al. (2011), each thread calculates the density, momentum and energy for the next time step. Since the calculation of these three values is mainly independent, it is likely better to launch three times as many threads and to divide the calculation of density, momentum, and energy among the threads. Another minor improvement concerns the storage of the Jacobian $\mathbf{A}$. In the implementation, the matrix is stored in register memory, and matrix multiplication is used to calculate the output. However, since this matrix is only used once and it never changes, it is easier to hard-code the three lines, instead of using an external library to perform this multiplication.

The authors were not willing to provide the code they produced in the article. This makes it

impossible to check their results, and therefore, this thesis proposes an entirely new implementation of the CE/SE algorithm on GPU in 1 dimension.

Next, to the papers mentioned above, the CUDA programming guide by Nvidia CUDA ™ (2015) will be the primary source of information about CUDA. Also Lee et al. (2010) and Gregg and Hazelwood (2011) will be consulted for ensuring a proper comparison between the CPU and the GPU algorithm.

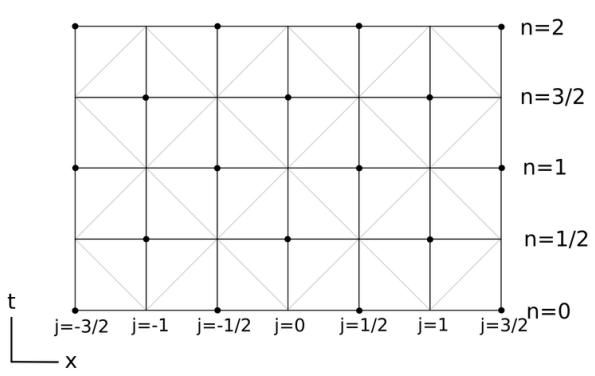
3. Introduction to the CE/SE scheme

This chapter is going to explain the basics of the CE/SE scheme and how to advance from a certain time step to the next one. This explanation is far from complete, and for a more extensive description of the scheme, the reader is recommended to read the technical memorandum by NASA, Chang et al. (1998).

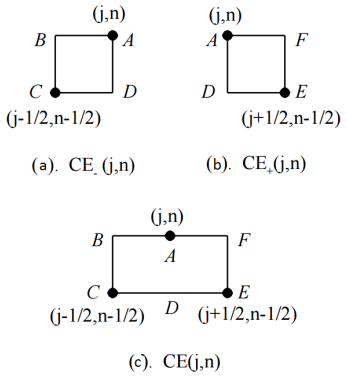
3.1 CE/SE in 1 dimension

In the year 1995, S.C. Chang published a paper which described a new method for numerically solving flow problems. The new method was called conservation element solution element (CE/SE). The development of the technique was guided by the belief that it should not make the assumption that the solution smooth and therefore the new technique can be used to calculate a numerical solution in places where this assumption is not valid, such as in shocks. More design principles and what drove the development of the scheme can be found in Chang (1995). Next to the ability of capturing shocks, the method is also easily extensible to multiple dimensions, as can be seen later in this chapter. Also, it is very suitable for parallel computation. These characteristics make the CE/SE scheme stand out from other well-known Euler solvers. In physics, the motion of a fluid can be derived from three conservation laws: the conservation of mass, momentum, and energy. Conservation of mass means that the mass inside a closed volume V can only change by a flow of mass through the borders of the volume. The same applies for momentum and energy. Mathematically, these conservation laws in 1 dimension are described by the Euler equation in integral form:

$$\oint_{S(V)} \langle f_m, u_m \rangle \cdot d\sigma \cdot \vec{n} = 0, \quad m = 1,2,3 \tag{3.1}$$



(a) CE/SE mesh



(b) The conservation elements[1]

Figure 3.1: Definition of the conservation elements and solution elements

The values u_1, u_2 and u_3 represent respectively the amount of mass, momentum and energy at each infinitesimal area $d\sigma$ on the surface of volume V. Similarly, f_1, f_2 and f_3 represent respectively the amount of mass, momentum and energy flowing out of the volume through $d\sigma$. $\vec{n}$ is the outward unit normal of $S(V)$. Therefore this equation represents all the three conservation laws. For the rest of this section, the subscript m will always be taken to run from 1 to 3.

Now, consider a two dimensional Euclidean space E_2, with space on one axis and time on the other. In this space, let let Ω denote all the mesh points (j,n), with $(2n) \in \mathbb{Z}$ and $(j+n+\frac{1}{2}) \in \mathbb{Z}$, as shown in figure 3.1a. j and n respresent the spatial and time coordinate respectively. Each mesh point is surrounded a so-called solution element (SE), whose borders are depicted by the dashed lines in figure 3.1a . Each point inside a solution element SE(j,n) belongs to exactly one mesh point (j,n). Now, for any point $(x,t) \in SE(j,n)$, u_m and f_m (with $m = 1,2,3$) can be approximated by the first-order Taylor expansion at position (x,y):

$$u_m^*(x,t;j,n) = (u_m)_j^n + (u_{mx})_j^n (x - x_j) + (u_{mt})_j^n (t - t^n) \tag{3.2}$$

$$f_m^*(x,t;j,n) = (f_m)_j^n + (f_{mx})_j^n (x - x_j) + (f_{mt})_j^n (t - t^n) \tag{3.3}$$

In this equation, $(u_m)_j^n$ and $(f_m)_j^n$ are the values of u_m and f_m at mesh position (j,n), respectively. Further, $(u_{mx})_j^n$ and $(f_{mx})_j^n$ are the spatial derivatives of $(u_m)_j^n$ and $(f_m)_j^n$ and $(u_{mt})_j^n$ and $(f_{mt})_j^n$ are the time derivatives.

Similarly to the solution element, we define rectangular conservation elements, whose corners are the mesh points in Ω and which are enclosed by the solid lines in figure 3.1a. The rectangle

that has element (j,n) as the upper right vertex is called $CE_-(j,n)$ and the one having (j,n) as the upper-left vertex is called $CE_+(j,n)$, as shown in figure 3.1b. Together they are called the conservation element, $CE(j,n)$. It should be noted that each $CE_\pm$ consists of 4 sides, of which 2 lie in $SE(j,n)$ and the other 2 in $SE(j\pm\frac{1}{2}, n-\frac{1}{2})$. Therefore, we know the flux through the bottom border and the sides of the CE and we can combine equation 3.2, 3.3 with 3.1 to derive the value for the mesh points at time $t=n$ using the values of the mesh points at time $t=n-\frac{1}{2}$. This results to the following equation, as is shown in Chang et al. (1998):

$$\left[(\mathbf{I}\mp\mathbf{A}^+)\vec{u}\pm(\mathbf{I}\mp(\mathbf{A}^+)^2)\frac{\Delta x}{4}(\vec{u}_x)\right]_j^n = \left[(\mathbf{I}\mp\mathbf{A}^+)\vec{u}\mp(\mathbf{I}\mp(\mathbf{A}^+)^2)\frac{\Delta x}{4}(\vec{u}_x)\right]_{j\pm\frac{1}{2}}^{n-\frac{1}{2}} \tag{3.4}$$

This equation introduces a new notation, which will also be used later on in the thesis, namely:

- Δx is the spatial distance between the mesh points (j,n) and $(j+1,n)$.
  Δt is the time difference between the mesh points (j,n) and $(j,n+1)$.
- Matrices are written as bold capital letters. $\mathbf{I}$ always represents the identity matrix and $\mathbf{A}^+=\frac{\Delta t}{\Delta x}\mathbf{A}$, where $\mathbf{A}$ is the 3x3 Jacobian matrix formed by $\frac{\partial f_m}{\partial u_k}$, with $k,m=1,2,3$.
- Vectors are denoted by an arrow on top a letter. $\vec{u}$ is the 3x1 column vector formed by u_1, u_2 and u_3. Similarly, $\vec{u_x}$ is 3x1 column vector formed by u_{mx}, $m=1,2,3$
- Lastly, if multiple elements inside brackets refer to the same mesh point, the indices n and j will be written on the outer brackets only, as can be seen in equation 3.4

Note that equation 3.4 represents two equations: one corresponding to the upper signs and the other to the lower signs. Now, we have two unknowns $\left(\vec{u}_j^n \text{ and } (\vec{u}_x)_j^n\right)$ and two equations (eq. 3.4), so all we have to do is solving this system of equations. Solving for $\vec{u}_j^n$ gives:

$$\vec{u}_j^n = \frac{1}{2}\left\{\left[\left(\mathbf{I}-\mathbf{A}^+\right)\left(\vec{u}-(\mathbf{I}+\mathbf{A}^+)\frac{\Delta x}{4}(\vec{u}_x)\right)\right]_{j+\frac{1}{2}}^{n-\frac{1}{2}} + \left[\left(\mathbf{I}+\mathbf{A}^+\right)\left(\vec{u}+(\mathbf{I}+\mathbf{A}^+)\frac{\Delta x}{4}(\vec{u}_x)\right)\right]_{j-\frac{1}{2}}^{n-\frac{1}{2}}\right\} \tag{3.5}$$

[1] Edited from figure 4 in Chang et al. (1998)

Equation 3.5 is the first part of the marching scheme in the CE/SE algorithm. If equation 3.4 is solved for $(\vec{u}_x)_j^n$, we obtain the second part in the marching scheme:

$$(\vec{u}_x)_j^n = \frac{1}{2}(\vec{S}_+ - \vec{S}_-)_j^n \quad \text{where} \quad (\vec{S}_\pm)_j^n = \left[(\mathbf{I} \mp \mathbf{A}^+)_j^n\right]^{-1}\left[(\mathbf{I} \mp \mathbf{A}^+)\,\vec{u} \mp (\mathbf{I} \mp (\mathbf{A}^+)^2)\,\vec{u}_x\right]_{j\pm\frac{1}{2}}^{n-\frac{1}{2}} \tag{3.6}$$

In Chang et al. (1999), it is shown that the matrix inverse in equation 3.6 exists, if the CFL-condition $\frac{(|v|+a)\Delta t}{\Delta x} \leq C_{max} = 1$ is satisfied at every mesh point. Here a is the speed of sound. Δt must be chosen such that this inequality is always satisfied.

The existing implementations, such as Shen et al. (2015), makes use of the so-called α-scheme, named after the bias variable α which will be introduced shortly. It can be shown that equation 3.6 is equal to:

$$\left(\vec{u}_{mx}\right)_j^n = \frac{1}{2}\left(\vec{u}_{x+} + \vec{u}_{x+}\right)_j^n \tag{3.7}$$

$$(u_{mx\pm})_j^n = \frac{2}{\Delta x}\left((u_m)_j^n - (u_m)_{j\pm\frac{1}{2}}^{n-\frac{1}{2}} - \Delta t(u_{mt})_{j\pm\frac{1}{2}}^{n-\frac{1}{2}}\right) \tag{3.8}$$

Now, simply taking the average of $(\vec{u}_{mx+})_j^n$ and $(\vec{u}_{mx-})_j^n$ as in equation 3.7 would give valid results for $(u_{mx})_j^n$ only if no discontinuities occur. At a discontinuity, $(j-\frac{1}{2}, n)$ and $(j+\frac{1}{2}, n)$ lie at the opposite sides of the boundary and hence, simply taking the average is not enough. The result must be smoothened by taking a biased average:

$$W_0(x_-, x_+, \alpha) = \frac{|x_+|^\alpha x_- + |x_-|^\alpha x_+}{|x_+|^\alpha + |x_-|^\alpha}, \text{ with } (|x_+|^\alpha + |x_-|^\alpha) > 0 \tag{3.9}$$

$$\vec{u_{mx}}_j^n = W_0\left((\vec{u}_{mx+})_j^n, (\vec{u}_{mx-})_j^n, \alpha\right) \tag{3.10}$$

The outcome of equation 3.9 will be biased towards the lowest value between x_- and x_+, for $\alpha > 0$. In a smooth area, $(\vec{u}_{x+})_j^n$ and $(\vec{u}_{x-})_j^n$ will lie close together, and the outcome of W_0 will be close to the regular central average. However, at a discontinuity, the difference between the input values of W_0 will be large. Since the input consists of the derivatives of u, a low value of $(\vec{u}_{x\pm})_j^n$ means a smooth area in u. Therefore, the average is biased towards the smooth region, that is, the lowest value. This greatly increases the accuracy if there are discontinuities in the solution. This thesis will use $\alpha = 2$ to be consistent with other literature on this topic.

To conclude, the CE/SE marching scheme consists of two equations, one for advancing to $(u_m)_j^n$ and another one for calculating $(u_{mx})_j^n$. Therefore, given $(u_m)_{j\pm\frac{1}{2}}^{n-\frac{1}{2}}$ and $(u_{mx})_{j\pm\frac{1}{2}}^{n-\frac{1}{2}}$, the next time level can be calculated as follows. Here, the equations have been rewritten to a form which is more suitable for implementation and which is used also in other literature.

$$\mathbf{A} = \frac{\partial f_m}{\partial u_k}, \quad m,k = 1,2,3 \tag{3.11}$$

$$(s_m)_j^n = \frac{\Delta x}{4}(u_{mx})_j^n + \frac{\Delta t}{\Delta x}(f_m)_j^n + \frac{(\Delta t)^2}{4\Delta x}(f_{mt})_j^n, \quad m = 1,2,3 \tag{3.12}$$

$$(u_{mx\pm})_j^n = \frac{2}{\Delta x}\left((u_m)_j^n - (u_m)_{j\pm\frac{1}{2}}^{n-\frac{1}{2}} - \Delta t(u_{mt})_{j\pm\frac{1}{2}}^{n-\frac{1}{2}}\right) \tag{3.13}$$

And the marching scheme is:

$$(u_m)_j^n = \frac{1}{2}\left[(u_m)_{j-1/2}^{n-1/2} + (u_m)_{j+1/2}^{n-1/2} + (s_m)_{j-1/2}^{n-1/2} + (s_m)_{j+1/2}^{n-1/2}\right] \tag{3.14}$$

$$(u_{mx})_j^n = W_0((u_{mx+})_j^n, (u_{mx-})_j^n, \alpha) \tag{3.15}$$

The above derivation makes use of a generalized version of the Euler equation. However, plugging in the original values for u_m and f_m (which are derived by the conservation laws), one obtains:

$$\vec{u} = \begin{bmatrix} \rho \\ \rho u \\ \rho E \end{bmatrix} \qquad \text{and} \qquad \vec{f} = \begin{bmatrix} u_2 \\ (\gamma-1)u_3 + \frac{(3-\gamma)(u_2)^2}{2u_1} \\ \gamma\frac{u_2 u_3}{u_1} - \frac{(\gamma-1)(u_2)^3}{2(u_1)^2} \end{bmatrix} \tag{3.16}$$

Here, ρ is the density, u is the x-velocity, E is the energy per unit mass and γ a constant, which depends on the fluid. Using equation 3.16, $\mathbf{A}$ can also be calculated, which completes the derivation of the 1D marching scheme. Now given an initial state at time $n = 0$, we can calculate the state at any time $n > 0$.

3.2 CE/SE in 2 dimensions

The marching scheme in two dimensions shows a lot of similarities with the scheme in one dimension, but the geometry and algebra is more complex. Therefore, this section has the same structure as section 3.1, but most of the mathematics will be left out. An extensive description of the 2D scheme can be found in Chang et al. (1998) or Chang et al. (1999).

Just as in the 1D case, we start by considering the Euler equation in integral form:

$$\oint_{S(V)} \vec{h}_m \cdot d\vec{s} = 0,\ m = 1,2,3,4 \tag{3.17}$$

where $\vec{h}_m = \langle f_m^x, f_m^y, u_m \rangle$. f_m^x and f_m^y denote the flux in the x- and y-direction respectively. For the rest of this chapter, the subscript $_m$ will run from 1 to 4.

Next, similar to the 1D version of the CE/SE scheme, let us define a uniform three dimensional space-time Euclidean space E_3. The first versions of the CE/SE scheme were based on triangular meshes, but this was later extended to quadrilateral meshes. The GPU code will be based on a quadrilateral mesh, because the CPU code provided as a starting point (Shen et al. (2015)) is also based on this. Figure 3.2a shows the layout of the mesh. The white circles represent the mesh points at time step $n \in \mathbb{Z}$ and the white triangles represent mesh points at time $n + \frac{1}{2}$. A mesh point at spatial coordinate (i, j) at time step n is denoted by $P_{i,j}^n$. The SE of $P_{i-1/2,j-1/2}^{n-1/2}$ is shown in figure 3.2b. The CE of a point $P_{i,j}^n$ consists of 4 cuboids, as shown in 3.2c. 3.2d shows a top view of the CE and names all the fluxes flowing in and out the of the CEs.

Just as in the 1D case, the values of $u_{i,j}^n$ and $f_{i,j}^n$ will be approximated by the first-order Taylor expansion around (i, j) and this approximation is used to evaluate equation 3.17. This results in

[1] This image is an copy-edit based on figure 2 in Shen et al. (2015).

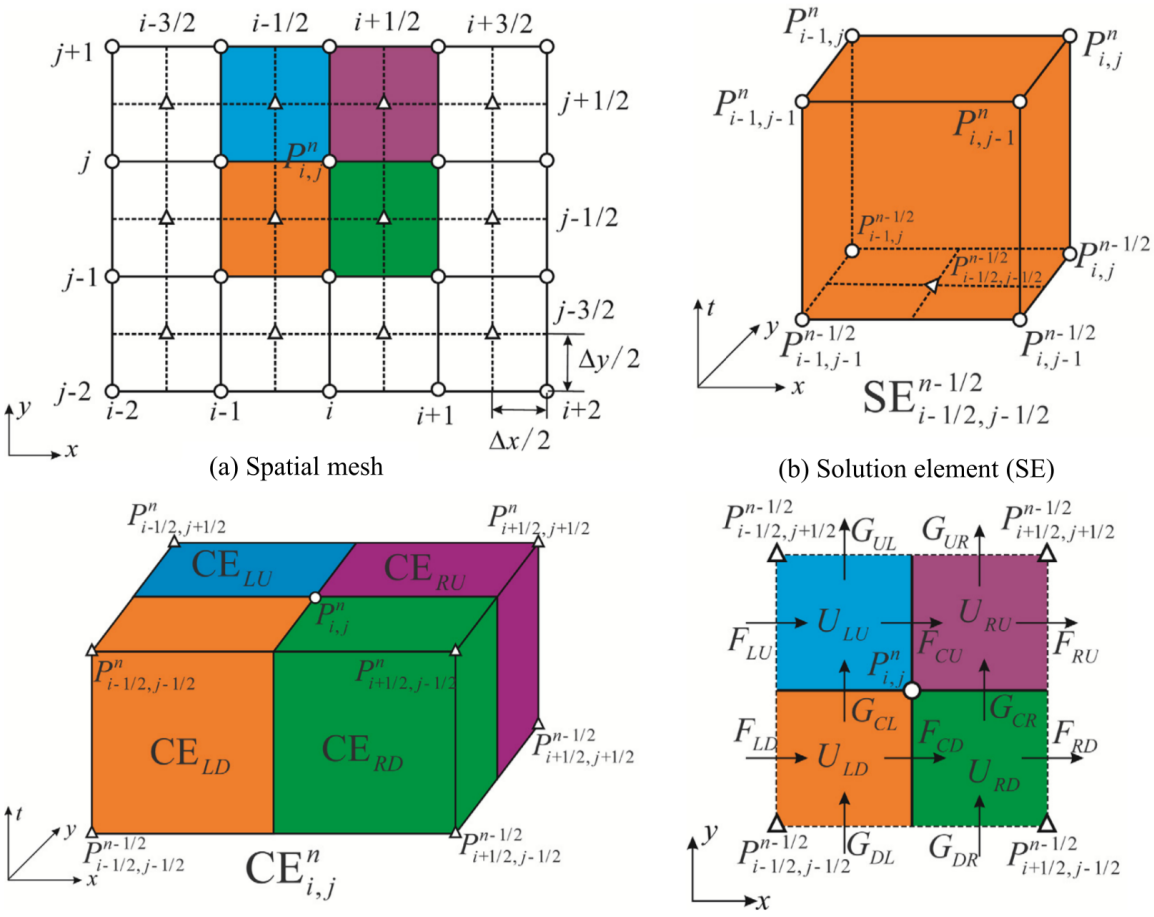


Figure 3.2: The definition of the CE and SE in 2D[1]

the following four equations, one for each CE:

$$(U_{LD})^n_{i,j} = \left[\vec{u} - \vec{u}_x\frac{\Delta x}{4} - \vec{u}_y\frac{\Delta y}{4}\right]^{n-1/2}_{i,j} = \left[U_{LD} + (F_{LD} - F_{CD})\frac{\Delta t}{\Delta x} + (G_{DL} - G_{CL})\frac{\Delta t}{\Delta y}\right]^{n-1/2}_{i,j} \tag{3.18}$$

$$(U_{RD})^n_{i,j} = \left[\vec{u} + \vec{u}_x\frac{\Delta x}{4} - \vec{u}_y\frac{\Delta y}{4}\right]^{n-1/2}_{i,j} = \left[U_{RD} + (F_{CD} - F_{RD})\frac{\Delta t}{\Delta x} + (G_{DR} - G_{CR})\frac{\Delta t}{\Delta y}\right]^{n-1/2}_{i,j} \tag{3.19}$$

$$(U_{RU})^n_{i,j} = \left[\vec{u} + \vec{u}_x\frac{\Delta x}{4} + \vec{u}_y\frac{\Delta y}{4}\right]^{n-1/2}_{i,j} = \left[U_{RU} + (F_{CU} - F_{RU})\frac{\Delta t}{\Delta x} + (G_{CR} - G_{UR})\frac{\Delta t}{\Delta y}\right]^{n-1/2}_{i,j} \tag{3.20}$$

$$(U_{LU})^n_{i,j} = \left[\vec{u} - \vec{u}_x\frac{\Delta x}{4} + \vec{u}_y\frac{\Delta y}{4}\right]^{n-1/2}_{i,j} = \left[U_{LU} + (F_{LU} - F_{CU})\frac{\Delta t}{\Delta x} + (G_{CL} - G_{UL})\frac{\Delta t}{\Delta y}\right]^{n-1/2}_{i,j} \tag{3.21}$$

This equation can be solved for the three unknowns $\vec{u}$, $\vec{u}_x$ and $\vec{u}_y$. This formula can be understood intuitively in the following way: The value of, for example, U_{LD} equals its value at the previous time step, plus the inflow minus outflow of the conserved property. As can be seen in

figure 3.2d, the inflow in the x-direction is $\frac{\Delta t}{\Delta x}F_{LD}$ and the outflow $\frac{\Delta t}{\Delta x}F_{CD}$. The same applies for the y-direction and the other CEs. In this way, eq. 3.18-3.21 can be easily verified. Adding these four equations together results in:

$$\vec{u}_{i,j}^{\,n} = \frac{1}{4}\Big[U_{LD} + U_{RD} + U_{RU} + U_{LU} + \frac{\Delta t}{\Delta x}\big(F_{LD} + F_{LU} - F_{RD} - F_{RU}\big) + \frac{\Delta t}{\Delta y}\big(G_{DL} + G_{DR} - G_{UL} - G_{UR}\big)\Big]_{i,j}^{n-1/2} \tag{3.22}$$

For the spatial derivatives, one can derive multiple solutions, because the number of equations in greater than the number of unknowns. For each dimension, two equations can be derived, one for each side of the CEs (above and below, or left and right). For $\vec{u}_x$, this results in:

$$\big(\vec{u}_x^{\,D}\big)_{i,j}^{n} = \frac{2}{\Delta x}\Big[U_{RD} - U_{LD} + \frac{2\Delta t}{(\Delta x)^2}\big(2F_{CD} - F_{LD} - F_{RD}\big) + \frac{2\Delta t}{\Delta y \Delta x}\big(G_{DR} - G_{CR} - G_{DL} + G_{CL}\big)\Big]_{i,j}^{n-1/2} \tag{3.23}$$

$$\big(\vec{u}_x^{\,U}\big)_{i,j}^{n} = \frac{2}{\Delta x}\Big[U_{RU} - U_{LU} + \frac{2\Delta t}{(\Delta x)^2}\big(2F_{CU} - F_{LU} - F_{RU}\big) + \frac{\Delta t}{\Delta y \Delta x}\big(G_{CR} - G_{UR} - G_{CL} + G_{UL}\big)\Big]_{i,j}^{n-1/2} \tag{3.24}$$

and solving for $\vec{u}_y$ leads to

$$\big(\vec{u}_y^{\,L}\big)_{i,j}^{n} = \frac{2}{\Delta y}\Big[U_{LU} - U_{LD} + \frac{\Delta 2t}{\Delta x \Delta y}\big(F_{LU} - F_{CU} - F_{LD} + F_{CD}\big) + \frac{2\Delta t}{(\Delta y)^2}\big(2G_{CL} - G_{DL} - G_{UL}\big)\Big]_{i,j}^{n-1/2} \tag{3.25}$$

$$\big(\vec{u}_y^{\,R}\big)_{i,j}^{n} = \frac{2}{\Delta y}\Big[U_{RU} - U_{RD} + \frac{\Delta 2t}{\Delta x \Delta y}\big(F_{CU} - F_{RU} - F_{CD} + F_{RD}\big) + \frac{2\Delta t}{(\Delta y)^2}\big(2G_{CR} - G_{DR} - G_{UR}\big)\Big]_{i,j}^{n-1/2} \tag{3.26}$$

In this derivation, it is assumed that the CFL condition is satisfied, which, in 2D is given by:

$$\frac{|u_x| + |a_x|\Delta t}{\Delta x} + \frac{|u_y| + |a_y|\Delta t}{\Delta y} \leq C_{max} = 1 \tag{3.27}$$

All that is left, is finding an equation for calculating the fluxes in the x- and y-direction. These equations are similar to the ones for calculating U: the flux are equal to the flux in the previous time step plus the increase flux, plus the increase flux in the orthogonal direction plus the flux

that was created over time. This leads to the following equation (only one equation for each direction is given, the others can easily be derived using figure 3.2d)

$$F_{LD} = \left(\vec{f}^{x} - \vec{f}_{y}^{x}\frac{\Delta y}{4} - \vec{f}_{t}^{x}\frac{\Delta t}{4}\right) \tag{3.28}$$

$$G_{DL} = \left(\vec{f}^{y} - \vec{f}_{x}^{y}\frac{\Delta y}{4} - \vec{f}_{t}^{y}\frac{\Delta t}{4}\right) \tag{3.29}$$

where f^x and f^y denote the total flux in the x- and y-direction respectively. f^x and f^y are calculated in the same way as in the 1D case, using the Jacobian $\mathbf{A}$. Only now we need two Jacboians: one for x-direction, $\mathbf{A}^x = \frac{\partial f_m^x}{\partial u_k}$, and one for the y-direction $\mathbf{A}^y = \frac{\partial f_m^y}{\partial u_k}$. Hence

$$\vec{f}_{x}^{x} = \mathbf{A}^x\vec{u}_x \qquad \vec{f}_{y}^{x} = \mathbf{A}^x\vec{u}_y \qquad \vec{f}_{t}^{x} = \mathbf{A}^x\left(\vec{f}_{x}^{x} - \vec{f}_{y}^{y}\right) \tag{3.30}$$

$$\vec{f}_{x}^{y} = \mathbf{A}^y\vec{u}_x \qquad \vec{f}_{y}^{y} = \mathbf{A}^y\vec{u}_y \qquad \vec{f}_{t}^{y} = \mathbf{A}^y\left(\vec{f}_{x}^{x} - \vec{f}_{y}^{y}\right) \tag{3.31}$$

Just as in the previous section, we can take the arithmetic average of the two solutions for the spatial derivatives, but this will give incorrect results at discontinuities. Therefore we define

$$(u_x)_{i,j}^n = W_0\left(\left[\left(u_{mx}^L\right)_{i,j}^n, \left(u_{mx}^U\right)_{i,j}^n, \alpha\right)\right. \qquad \text{and} \qquad (\vec{u}_y)_{i,j}^n = W_0\left(\left(u_{mx}^L\right)_{i,j}^n, \left(u_{mx}^R\right)_{i,j}^n, \alpha\right) \tag{3.32}$$

To conclude, it should be noted that the Euler equation in 2D dimension is slightly different from its 1D counterpart. The velocity is now split into an x-component and a y-component:

$$\vec{u} = \begin{bmatrix} \rho \\ \rho u_x \\ \rho u_y \\ \rho E \end{bmatrix} \quad \text{and} \quad \vec{f}^{x} = \begin{bmatrix} u_2 \\ (\gamma-1)u_4 + \frac{(3-\gamma)(u_2)^2}{2u_1} - \frac{(\gamma-1)(u_3)^2}{2u_1} \\ \frac{u_2 u_3}{u_1} \\ \gamma\frac{u_2 u_4}{u_1} - \frac{(\gamma-1)(u_2)\left[(u_2)^2+(u_3)^2\right]}{2(u_1)^2} \end{bmatrix} \quad \text{and} \quad \vec{f}^{y} = \begin{bmatrix} u_3 \\ \frac{u_2 u_3}{u_1} \\ (\gamma-1)u_4 + \frac{(3-\gamma)(u_3)^2}{2u_1} - \frac{(\gamma-1)(u_2)^2}{2u_1} \\ \gamma\frac{u_3 u_4}{u_1} - \frac{(\gamma-1)(u_3)\left[(u_2)^2+(u_3)^2\right]}{2(u_1)^2} \end{bmatrix} \tag{3.33}$$

Here, ρ is the density, u_x the x-velocity, u_y the y-velocity and E the energy per unit mass. This equation can be used to calculate $\mathbf{A}^x$ and $\mathbf{A}^y$. Now we have defined all the veriables in the marching scheme, so given an initial state, all following states can be calculated.

4. Implementation

This chapter will describe the actual implementation details of the CE/SE scheme on the GPU. For both the 1D case and the 2D case, first some theory will be recalled and it will be shown which parts of the algorithm can be parallelized. Next, the algorithm is presented in pseudo-code. An explanation of this algorithm and more details are presented in the final section. The 1D case was mainly implemented as an intermediate step towards implementing the 2D scheme. Therefore, the 2D scheme contains more optimizations than 1D scheme.

4.1 CE/SE scheme in 1 dimensions

4.1.1 Theory

Now that the mathematical foundation has been established in section 3.1, the 1 dimensional CE/SE GPU algorithm can be designed. The following items are provided as input to the algorithm:

- **Grid dimensions and size** The grid in 1 dimension is determined by a lower bound and an upper bound. The grid size is the number of pixels in which the grid is partitioned. Therefore, the grid size is one of the main factors in determining the precision of the algorithm.
- **Array $\mathbf{u_m}$ and $\mathbf{u_{mx}}$** This array contains the density, momentum and energy for each pixel in the grid at time $t = 0$. u_{mx} array contains the spatial derivatives of the properties at $t = 0$.
- **Courant number C** Determines the time step Δt. This value can never be higher than one, as mentioned in section 3.1.
- **End time $\mathbf{t_{end}}$** The algorithm will end when $t = t_{end}$.

The output will be the array u_m at $t = t_{end}$. This is result can be obtained by subsequently adding a value Δt to t and calculate ρ, ρv, E and u_x at $t+\Delta t$. Therefore the core of the algorithm consists of a loop which runs until $t = t_{end}$. Inside this loop, Δt is calculated first, by making use of the CFL condition:

$$\frac{(u+a)\Delta t}{\Delta x} \leq C_{max} = 1 \tag{4.1}$$

Next, $(u_m)_j^{n+\frac{1}{2}}$ is calculated for every spatial partition j. Finally, this value is used to calculate $(u_m)_j^{n+1}$, which describes the state of the system at $t + \Delta t$. Now t is updated, and the loop proceeds to the next iteration. It should be noted that the procedure for calculating $(u_m)_j^{n+\frac{1}{2}}$ is slightly different than for $(u_m)_j^{n+1}$. This has to do with the staggered mesh, as can be seen in figure 3.1a, which causes the boundary condition to be different.

Δt is bound by the CFL condition (equation 4.1), which means that it has to comply to $\Delta t \leq \frac{\Delta x}{a}$ at every position. The speed of sound a changes with every new time step, because the density and energy change. Therefore Δt must continuously be updated to comply with this condition. To achieve this, the maximum a among all spatial steps must be determined, because this speed of sound determines the maximum value of the time step size. This ensures $\Delta t \leq \frac{C\Delta x}{a}$.

4.1.2 GPU parallelization

In a sequential version of this algorithm, two loops are executed inside each other: the outer loop increases the time every iteration and the inner loop iterates over the spatial partitions. It is impossible to parallelize the outer loop because the calculation of each iteration depends on the previous one. The inner loop, however, can be parallelized, because the calculation of $(u_m)_j^n$ does not require $(u_m)_{i\neq j}^n$ to be calculated beforehand. Therefore, the value of $(u_m)^n$ for all spatial partitions can be determined at once.

Also, the calculation Δt can be parallelized. To determine the maximum speed of sound, the algorithm has to loop over all spatial partitions to determine the speed of sound at that position. This loop can also be parallelized. First, each spatial partition is assigned one thread to calculate the velocity at that point, and this value is placed in an array. Secondly, this array is distributed among blocks and threads to find the maximum velocity in the array.

In this thesis, an implementation on GPU's is used for the CE/SE scheme. The decision for using

a GPU instead of a CPU is based on the characteristics of both units. A CPU contains a few powerful, but expensive, cores, which are optimized for sequential processing. On the other hand, a GPU contains many less powerful cores, optimized parallel processing. In the CE/SE scheme, the number of spatial partitions can be high (many thousands of pixels), and the mathematical operations are relatively easy. Therefore, many less powerful cores working in parallel are preferred over a few powerful cores. A GPU implementation is likely to have lower execution time, especially for a larger number of partitions. Also, the choice has been made to use NVIDIA CUDA C for implementation instead of e.g. OpenCL. The main reason is that the research environment was more suitable for using CUDA because the NVIDIA GPU's were supplied for running the simulations. CUDA is optimized for NVIDIA GPU's, and therefore CUDA is likely to give better results. According to Fang et al. (2011), there is no difference between an optimized OpenCL application and an optimized CUDA application. Therefore the choice for CUDA does not impact the performance of the program in a negative way.

4.1.3 Algorithm design

Algorithm 1 gives the pseudo-code for the part of the code which is executed on the CPU. The kernel to find the maximum speed of sound among all spatial steps is shown in algorithm 2 and the kernel to advance to the next time step in algorithm 3. In all these algorithms, the end of a **forall .. do in parallel** block means, that a thread synchronization is performed.

4.1.4 Implementation and optimization

The pseudo-code above gives a general overview of how the CE/SE scheme can be implemented and which parts can be parallelized. For the actual implementation into CUDA, many more design decisions had to be made. These decisions will be discussed in this section. The actual CUDA implementation is attached to this thesis.

Finding Δt

The algorithm starts by finding the maximum velocity for the speed of sound among the spatial positions. As commented in algorithm 2, parallel reduction has been used to do this

Algorithm 1 CE/SE algorithm

Input:
The initial flow field $(u_m)^0$
The spatial derivative of $(u_m)^0$, called $(u_{mx})^0$
Courant number C
The end time t_{end}
The spatial step size Δx

Output:
The flow field $(u_m)^n$ at time $t = t_{end}$

1: **procedure** MAINCPU
2: Allocate space for $(u_m)^0$, $(u_m)^0$, $(u_m)^{n+1}$, $(u_m)^{n+1}$ on device
3: Allocate space to store Δt on device
4: Initialize the $(u_m)^0$ and $(u_m)^0$ on device
5: **while** $t < t_{end}$ **do**
6: Launch `findDT` kernel: find the lowest value of a and calculate Δt from it
7: t = t+Δt
8: Launch `CESE` kernel: advance $(u_m)^n$ and $(u_{mx})^n$ to $(u_m)^{n+1}$ and $(u_{mx})^{n+1}$
9: **end while**
10: Copy $(u_m)^n$ back to host
11: **end procedure**

efficiently. Parallel reduction is a tree-based technique, for finding the maximum value in an array. To understand it, consider an array of 256 elements. Now, 128 threads are launched. Each thread i compares the ith element and the $(i+128)$th element and writes the greatest of these into the ith element. Now the array to search in is reduced to 128 elements, and the same procedure is now executed by 64 threads. This continues until only one element is left, which is the maximum value in the array. If the number of elements gets lower than 64, still a full warp, which is 32 threads, will be launched, leaving some of the threads idle. Therefore there is no need in lowering the number of threads launched when the number of elements to search has become lower than 64 because 32 threads will be launched anyway. Also, because only one warp is running, a sequential execution of instructions in guaranteed. Hence, thread synchronization is not needed anymore. These two features make it more efficient to pull the final six compare-statements outside the loop and to code them manually. For example code and a more detailed explanation, see Harris (2012).

Another popular way to find the maximum in an array is to use the external library, called Thrust. Thrust is a toolkit from NVIDIA and tries to implement basic tasks like scanning and sorting ef-

Algorithm 2 Find correct value of Δt

Input:
The flow field (u_m), containing $Jx3$ elements
A grid consisting of B blocks, consisting of T threads each
Output:
The value of Δt based on the values in `data` and the CFL number C

1: **procedure** FINDDT
2: Distribute the elements in `data` among all threads, where `sub_data_j` is assigned to thread j
3: **forall** Block $b \in [0..B)$ **do in parallel**
4: **forall** thread $j \in$ block b **do in parallel**
5: maxInThread[j] = -1
6: **forall** element in `sub_data_j` **do in parallel**
7: u $\leftarrow \frac{(u_2)_j}{(u_1)_j}$
8: p $\leftarrow (\gamma - 1) \cdot \left((u_3)_j - \frac{1}{2} \cdot (u_1)_j \cdot u^2\right)$
9: a $\leftarrow \sqrt{|\gamma \frac{p}{(u_1)_j}|} + |u|$
10: **if** a > maxInThread[j] **then**
11: maxInThread[j] $\leftarrow$ a
12: **end if**
13: **end forall**
14: **end forall**
15: `maxInBlock[`b`]` $\leftarrow$ max value among `maxInThread[j]` for all threads in block B
16: ▷ This can be done efficiently using parallel reduction
17: Write `maxInBlock[`b`]` to global memory
18: **end forall**
19:
20: **if** I'm the block that finished last **then**
21: $b \leftarrow$ id of current block
22: Distribute `maxInBlock` among threads in b, `sub_data_j` is assigned to thread j
23: **forall** thread $j \in$ block b **do in parallel**
24: `maxInThread[j]` $\leftarrow$ max value in `sub_data_j`
25: **end forall**
26: maxVal $\leftarrow$ maximum value among `maxInThread[j]`
27: ▷ This can be done efficiently using parallel reduction
28: $\Delta t \leftarrow \frac{C \cdot \Delta x}{\text{maxVal}}$
29: **end if**
30: **end procedure**

Algorithm 3 Advance to next time step

Input:

The flow field $(u_m)^0$ and its spatial gradient $(u_{mx})^0$, both containing $Jx3$ elements
The time step Δt, spatial step Δx and constant α
A grid consisting of b blocks and $(J,3)$ threads in total

Output:

The flow field $(u_m)^1$ and its spatial gradient $(u_{mx})^1$

1: **procedure** CESE()
2: NextStep($(u_m)^0$,$(u_{mx})^0$,Δt,Δx,1)
3: NextStep($(u_m)^{\frac{1}{2}}$,$(u_{mx})^{\frac{1}{2}}$,Δt,Δx,0)
4: **end procedure**
5:
6: **procedure** NEXTSTEP($(u_m)^{n-\frac{1}{2}}$,$(u_{mx})^{n-\frac{1}{2}}$,Δt,`ishalf`)
7: **forall** ($j \in [0..J)$ and $m \in [1,2,3]$ **do in parallel**
8: $(u_{mt})_j^{n-\frac{1}{2}} \leftarrow -\mathbf{A}(u_m x)_j^{n-\frac{1}{2}}$
9: **end forall**
10: **forall** ($a \in [0..J)$ and $b \in [1,2,3]$ **do in parallel**
11: $(f_{mt})_j^{n-\frac{1}{2}} \leftarrow -\mathbf{A}(u_m t)_j^{n-\frac{1}{2}}$
12: Use eq. 3.16 to calculate $(f_m)_j^{n-\frac{1}{2}}$
13: $(s_m)_j^{n-\frac{1}{2}} \leftarrow (f_m)_j^{n-\frac{1}{2}} + \frac{\Delta t}{4}(f_{mt})_j^{n-\frac{1}{2}}$
14: **end forall**
15: **forall** ($a \in [0..J)$ and $b \in [1,2,3]$ **do in parallel**
16: **if** (`ishalf` $== 1 \vee a == 0) \wedge$ (`ishalf` $== 0 \vee a == J$) **then**
17: **if** `ishalf` $== 1$ **then**
18: $(u_m)_j^n \leftarrow \frac{1}{2}\left[(u_m)_j^{n-\frac{1}{2}} + (u_m)_{j+\frac{1}{2}}^{n-\frac{1}{2}} + \frac{\Delta x}{4}\left((u_{mx})_j^{n-\frac{1}{2}} - (u_{mx})_{j+\frac{1}{2}}^{n-\frac{1}{2}}\right) + \frac{\Delta t}{\Delta x}\left((s_m)_j^{n-\frac{1}{2}} + (s_m)_{j+\frac{1}{2}}^{n-\frac{1}{2}}\right)\right]$
19: **else**
20: $(u_m)_j^n \leftarrow \frac{1}{2}\left[(u_m)_j^{n-\frac{1}{2}} + (u_m)_{j-\frac{1}{2}}^{n-\frac{1}{2}} + \frac{\Delta x}{4}\left((u_{mx})_{j-\frac{1}{2}}^{n-\frac{1}{2}} - (u_{mx})_j^{n-\frac{1}{2}}\right) + \frac{\Delta t}{\Delta x}\left((s_m)_j^{n-\frac{1}{2}} + (s_m)_{j+\frac{1}{2}}^{n-\frac{1}{2}}\right)\right]$
21: **end if**
22: $(x_\pm) \leftarrow \pm\frac{2}{\Delta x}\left[(u_m)_{j\pm1/2}^{n-1/2} + \frac{\Delta t}{2}(u_{mt})_{j\pm1/2}^{n-1/2} - (u_m)_j^n\right]$
23: $(u_{mx})_j^n \leftarrow \frac{|(x_+)_j^n|^\alpha \cdot |(x_-)_j^n| + |(x_-)_j^n|^\alpha \cdot |(x_+)_j^n|}{|(x_+)_j^n|^\alpha + |(x_-)_j^n|^\alpha}$
24: **else**
25: $(u_m)_j^n \leftarrow (u_m)_j^{n-\frac{1}{2}}$
26: $(u_{mx})_j^n \leftarrow (u_{mx})_j^{n-\frac{1}{2}}$
27: **end if**
28: **end forall**
29: **end procedure**

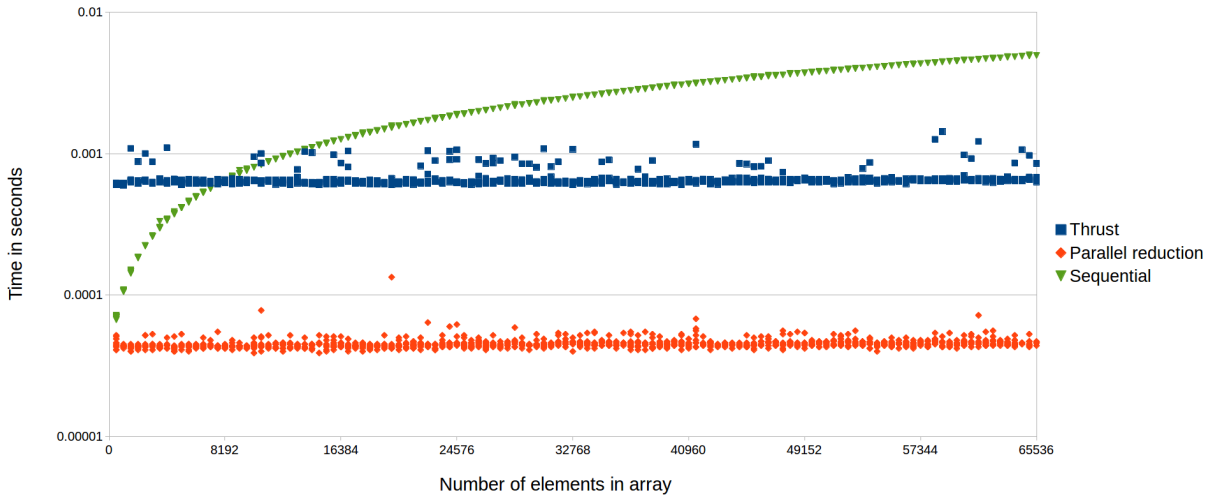


Figure 4.1: Comparison of various searching techniques

ficiently in parallel and to make it accessible and readable by using a high level of abstraction. More information can be found in Nvidia ™ (2016). The decision for using parallel reduction instead of Thrust has been made based on performance tests. Both methods have been implemented, and their runtimes have been compared. This is shown in figure 4.1. Note that the y-axis has a logarithmic scale. The green line is a sequential implementation. As can be seen, the sequential version is by far the slowest and increases linearly with the size of the array. Both Thrust and parallel reduction have a constant runtime, and parallel reduction is more than ten times faster than Thrust. The performance difference is likely due to the complexity of Thrust. Thrust is meant for making complex tasks more simple and readable. The cost of setting up the vectors and launching the library is too expensive for the simple task of finding the maximum value in an array. Therefore parallel reduction has been implemented for finding the maximum value in the array.

CE/SE kernel

Next, the CE/SE kernel is launched, in which most of the actual work is performed. The first decision to take is how many threads to launch: 1 per spatial position or 1 per property (density, momentum, energy) per spatial position. In Wei Ran et al. (2011) the former has been used. However, since the calculations of the properties are mostly independent, the cost of launching three threads may be less than letting one thread calculate three values. Therefore, the implementation proposed in this thesis uses the latter option.

Secondly, the memory layout must be optimized. The fastest option is to copy all the required data to shared memory at the beginning of the kernel, and only access shared memory. However, shared memory is limited to a block, and only threads within a block can access the shared memory of that block. This poses problems at the borders between blocks. Consider that the array (u_m), consisting of 100 elements, is distributed among two blocks and copied into shared memory. Now, the 49th element needs access to the 50th element, which is stored in the shared memory of the other block. The solution is to copy 51 elements to shared memory: [0..50] for block 1 and [50..99] for block 2. In this way, all elements can be calculated without the need of accessing global memory after the initial loading into shared memory. Only in every odd step, the very last element in the array cannot be updated but must be copied from the previous time step. This technique is called the overlapping scheme and is also described in Wei Ran et al. (2011).

At the end of the kernel, the shared memory is copied back to the global memory. However, this should happen to different arrays than those used to populate the shard memory. If the same array is used, a situation could occur in which some block has finished the kernel and copied the values back to the global memory, while the block next to it has not yet copied from global memory to its shared memory. In this case, the overlapping space points will already contain the values of the next time step. Therefore, two separate arrays must be used.
Next, u_{mt} and f_{mt} are also written to shared memory in separate arrays. In this way, the neighboring threads do not have to calculate u_{mt} and f_{mt} for their neighbors, but they can copy it from the shared memory. This is the reason why the threads are synced multiple times in the kernel.

It has been investigated whether constant or texture memory would be useful in storing the data. However, since all these two memory types are read-only, they are not suitable for use in this program. Texture memory is useful is data is written once and then accessed often. In the CE/SE algorithm, the data is accessed only before it is updated again. Therefore, shared memory is most suitable in this case. At various stages in the program, a thread synchronization is

needed to avoid read and write conflicts. In CUDA, this is implemented using a function called `__syncthread()`. The implementation of this function in PTX code is complicated and can cause unexpected side effects. One example of this is the following. When a kernel is called, often too many threads are launched, because the number of threads per block does not divide the number of required threads. In the first implementations of the algorithms for this project, the redundant threads were returned immediately, which worked fine. However, a deeper research into the `syncthread()` function, learns that a correct outcome of the program cannot be guaranteed when using this way of killing redundant threads. Because the killed threads will never reach the synchronization bar and therefore it is not guaranteed that the function will not wait for a killed thread. Therefore, all code except the `syncthread` function is surrounded by an if-else statement, making sure that all synchronization bars are passed by all threads, but also that the redundant threads do not execute any operations on the data. That is, the following code would not be guaranteed to work:

```
1  __global kernel(int sizeOfArray){
2      x = blockIdx.x*blockDim.x + threadIdx.x;
3      if(x >= sizeOfArray) return;
4
5      // Some code
6      __syncthreads();
7      // Other code;
8  }
```

It should be replaced by:

```
1  __global kernel(int sizeOfArray){
2      x = blockIdx.x*blockDim.x + threadIdx.x;
3      if(x < sizeOfArray){
4          // Some code
5      }
6      __syncthreads();
7      if(x < sizeOfArray){
8          // Other code;
9      }
10 }
```

This is also mentioned in the CUDA Programming Guide, section B.6 on synchronization functions.

4.2 CE/SE scheme in 2 dimensions

4.2.1 Theory and GPU parallelization

The structure of the 2D CE/SE algorithm is very similar to the 1D algorithm, with the only difference being the equations used. The input to the algorithm is the same, only now in 2D. That is, the grid size and dimensions are in the x- and y-direction and u_m and u_{mx} are given for each element (i, j) at $t = 0$. The format of the Courant number and the end time, t_{end} does not change. Similar to the 1D case, at a certain time step $t = t_0$, Δt will be determined, based on the CFL condition. Next, the values for ρ, ρu_x, ρu_y and E and their time derivatives at time $t = t_0 + \Delta t$ are calculated. In this calculation, the boundary conditions are imposed as well. This process is repeated as long as $t < t_{end}$. The CFL-condition in 2 dimensions is given by 3.27 and can be rewritten as:

$$\Delta t \leq \frac{C \cdot \mathrm{MIN}(\Delta x, \Delta y)}{|u| + |a|} \tag{4.2}$$

where $\mathrm{MIN}(x, y)$ returns the lowest value of x and y.

Because of these similarities between the 1D and the 2D CE/SE scheme, the parallelization strategy is also very similar. Because the calculation of the $(u_m)^n_j$ does not require $(u_m)^n_{i\neq j}$ to be known, the calculation $(u_m)^n_j$ for every position can be done in simultaneously. Also, the procedure of determining Δt does not change, except for the replacement of a few equations. These 2D equations do not depend on any value that is not know beforehand, so the determination of Δt can be parallelized in the same way as in the 1D case. All the arguments raised in section 4.1.2 for using CUDA C, can be used in the 2D case, except for one big difference: limited memory space. Since the 2D meshes are much larger than the 1D meshes, memory is the limiting factor in converting the CPU algorithm into the GPU algorithm. Accessing the global GPU memory is slow, but shared memory is too small to store all data. This trade-off will be discussed more extensively in section 4.2.3.

4.2.2 Algorithm design

The host code is almost the same as the 1D host code. The only difference is that all variables are allocated to store the 2D mesh and are extended to store the y-derivatives as well. Also the `findDT` kernel is very similar to the 1D version. The changes required to convert algorithm 2 to its 2D couterpart are given in in algorithm 4. The kernel to advance to the next time step is given in algorithm 5

Algorithm 4 Conversion of algorithm 2 to 2D

1: ▷ Replace line 5-7 by
2: $u_x \leftarrow \frac{(u_2)_j}{(u_1)_j}$
3: $u_y \leftarrow \frac{(u_3)_j}{(u_1)_j}$
4: $\mathrm{p} \leftarrow (\gamma - 1) \cdot \left((u_4)_j - \frac{1}{2} \cdot (u_1)_j \cdot \left((u_x)^2 + (u_y)^2 \right) + \right)$
5: $\mathrm{a} \leftarrow \sqrt{|\gamma \frac{p}{(u_1)_j}|} + \sqrt{(u_x)^2 + (u_y)^2}$
6: ▷ And line 28 by:
7: $\Delta t \leftarrow \frac{C \cdot \mathtt{MIN}(\Delta x, \Delta y)}{\mathrm{maxVal}}$

4.2.3 Implementation and optimization

The only big differences in implementation between the 1D version and the 2D version are related to the `CESE` kernel. The arrays that store the u_m, u_{mx} and u_{my} are much larger than in the 1D version, and this will dominate in the design of the algorithm. For example, to avoid unnecessary data transfers between the host and the device, these arrays will be initialized on the GPU. A small kernel will be launched, and each thread calculates a value of the u_m, u_{mx} and u_{my} at a spatial position. After this, the main loop begins, which runs until $t \geq t_{end}$. Inside this loop, first Δt is calculated, in the same way as in the 1D case. Immediately after this kernel has finished, the `CESE` kernel is launched. While the `CESE` kernel is running, Δt is copied back from the device to the host on a different stream, and the host variable t is updated. Using the Nvidia profiler, it can be shown that the copy of Δt from the device to the host is entirely in parallel with the `CESE` kernel, which provides a small speed-up.

Inside `CESE` kernel, the required data is copied from global memory into shared memory. Just as in the 1D case, the overlapping scheme is used to handle the boundaries between the blocks

Algorithm 5 Advance to next time step

Input:

The flow field $(u_m)^0$ and its spatial gradients, $(u_{mx})^0$ and $(u_{my})^0$, all containing $IxJx4$ elements

The time step Δt, spatial step Δx and constant α

A grid consisting of b blocks and $(I, J, 4)$ threads in total

Output:

The flow field $(u_m)^1$ and its spatial gradients, $(u_{mx})^1$ and $(u_{my})^1$

1: **procedure** CESE()
2: NextStep($(u_m)^0$,$(u_{mx})^0$,Δt,Δx,1)
3: NextStep($(u_m)^{\frac{1}{2}}$,$(u_{mx})^{\frac{1}{2}}$,Δt,Δx,0)
4: **end procedure**
5:
6: **procedure** NEXTSTEP($(u_m)^{n-\frac{1}{2}}$,$(u_{mx})^{n-\frac{1}{2}}$,Δt,`ishalf`)
7: **forall** $i \in [0..I)$ and $j \in [0..J)$ and $m \in [1,2,3,4]$ **do in parallel**
8: $i_0 \leftarrow i+1-$`isHalf`
9: $j_0 \leftarrow j+1-$`isHalf`
10: Use eq. 3.18-3.21 to calculate U_{LD}, U_{RD}, U_{RU} and U_{LU} at point $(i+1, j+1)$
11: Use eq. 3.28 and 3.30 to calculate F_{LD}, F_{RD}, F_{RU} and F_{LU} at point $(i+1, j+1)$
12: Use eq. 3.29 and 3.31 to calculate G_{DL}, G_{DR}, G_{UR} and G_{UL} at point $(i+1, j+1)$
13: $u_{old}[3] \leftarrow \{(u_m)^{n-\frac{1}{2}}_{i,j}, (u_m)^{n-\frac{1}{2}}_{i+1,j}, (u_m)^{n-\frac{1}{2}}_{i+1,j+1}, (u_m)^{n-\frac{1}{2}}_{i,j+1}\}$
14: **end forall**
15: **forall** $(a \in [0..J)$ and $b \in [1,2,3]$ **do in parallel**
16: Use eq. 3.22 to calculate $(u_m)^n_{i_0,j_0}$
17: $\text{UaL} \leftarrow \frac{2\cdot(u_m)^n_{i_0,j_0} - 2\cdot u_{old}[0] + \mathbf{A}^x\cdot(u_{mx})^{n-1/2}_{i,j} + \mathbf{A}^y\cdot(u_{my})^{n-1/2}_{i,j}}{\sqrt{(\Delta x)^2 + (\Delta y)^2}}$
18: $\text{UaR} \leftarrow \frac{2\cdot(u_m)^n_{i_0,j_0} - 2\cdot u_{old}[1] + \mathbf{A}^x\cdot(u_{mx})^{n-1/2}_{i+1,j+1} + \mathbf{A}^y\cdot(u_{my})^{n-1/2}_{i+1,j+1}}{\sqrt{(\Delta x)^2 + (\Delta y)^2}}$
19: $\text{UaC} \leftarrow W_0(\text{UaL}, \text{UaR})$
20: $\text{UbL} \leftarrow \frac{2\cdot(u_m)^n_{i_0,j_0} - 2\cdot u_{old}[2] + \mathbf{A}^x\cdot(u_{mx})^{n-1/2}_{i+1,j} + \mathbf{A}^y\cdot(u_{my})^{n-1/2}_{i+1,j}}{\sqrt{(\Delta x)^2 + (\Delta y)^2}}$
21: $\text{UbR} \leftarrow \frac{2\cdot(u_m)^n_{i_0,j_0} - 2\cdot u_{old}[3] + \mathbf{A}^x\cdot(u_{mx})^{n-1/2}_{i,j+1} + \mathbf{A}^y\cdot(u_{my})^{n-1/2}_{i,j+1}}{\sqrt{(\Delta x)^2 + (\Delta y)^2}}$
22: $\text{UbC} \leftarrow W_0(\text{UbL}, \text{UbR})$
23:
24: $(u_{mx})^n_{i_0,j_0} \leftarrow \frac{\sqrt{(\Delta x)^2 + (\Delta y)^2}\cdot(UaC-UbC)}{2\cdot\Delta x}$
25: $(u_{my})^n_{i_0,j_0} \leftarrow \frac{\sqrt{(\Delta x)^2 + (\Delta y)^2}\cdot(UaC+UbC)}{2\cdot\Delta y}$
26: **end forall**
27: **end procedure**

of shared memory. However, the 2D uses a more efficient version. In the overlapping scheme, each block fetches some extra data points on all sides, to be able to calculate the all if its data points. In the 1D case, one thread was assigned to each data point, including the extra data points. However, as can be seen in algorithm 3, every time step, the threads on one side of the block are idle. To avoid this, one thread is assigned to two overlapping data points on different sides of the block. The first iteration, it will calculate the data point on the left, and the second time the data point on the right, as can be seen in 5. This increases the number of data points that a block can run by $\frac{1}{2n}$% where n is the block size in the x- and y-direction.

There is one extra possible optimization, which has not been implemented in this project. Instead of using global and shared memory, it would be interesting to implement the algorithm, using surface memory. Surface memory uses CUDA arrays with a special flag, so it can be read and written using the surface API functions. It uses the same spatial locality as texture memory. The main drawback is that there is no read/write coherency within kernels. More information can be found in the CUDA programming guide, Nvidia ™ (2016). The spatial locality might improve the performance of the algorithm, since the new value of an element, depends solely on its neighboring elements. However, due to the limited cache size, it is unknown whether it would result in a performance gain. Implementing surface memory requires a complete rewrite of the program, but in future implementations, it would definitely be worth investigation more.

A last note about the implementation concerns single- and double-precision. Initially, both the CPU and the GPU programs used double precision, to increase the accuracy of the result. However, as will be shown in the next chapter, using double-precision has a huge impact on the speed of the application, compared to single precision. GPUs do not have native support for double precision, so double precision floating point operations are expensive (See also appendix A). Therefore, the use of single- or double-precision is a trade-off between speed and accuracy. Depending on the application, one might want to use either single or double precision.

6. Analysis and results

In this section, the runtimes of the 1D and 2D CE/SE algorithm on GPU will be discussed. To ensure correctness, it will start by validating the written code using reference solutions. Next, the degrees of freedom are determined, and the impact of all those variables will be measured. Finally, this is compared with the existing CPU algorithms.

5.1 1D analysis

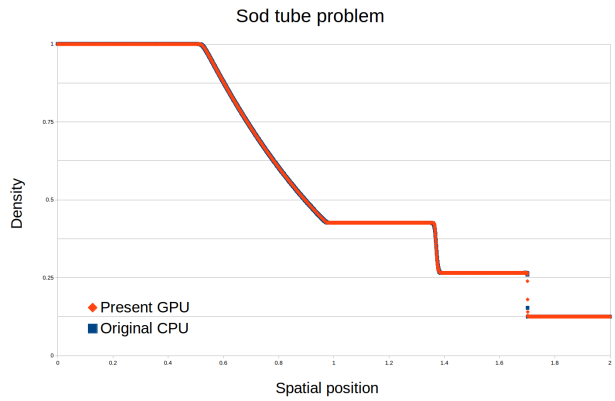
5.1.1 Validation

The first part of the analysis is validating the output of the program, by running various simulations for which the output is known. Four test cases have been selected from Shen et al. (2015), each having a different level of complexity. The initial conditions can be found in table 5.1. In this table, t_{end} is the time at which the simulation ends and CFL is the CFL number. The column 'gasses' gives the initial set-up for the simulations. For example, the Sod tube problem, is a tube of length 2, consisting of 2 gasses. On the left size of the tube (where $0 \leq x \leq 1$), the density is 1, and the pressure is 1. On the right side of the tube ($1 < x \leq 2$), the density is 0.125, and the pressure is 0.1. It can be seen as a tube with two gasses, separated by a thin film. At $t = 0$, the film is removed, and the simulation starts. The sides of the tube are non-reflective. For all cases $\gamma = 1.4$, the number of spatial steps is 4000 and $(u_{mx})_j^0 = 0, \forall\, j$.

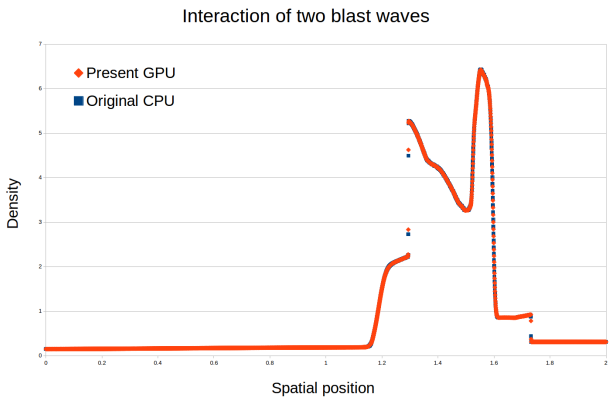
The first test case is the Sod tube problem, an idealized version of a 1-dimensional shock tube. The second test case is the interaction between blast waves. The reason to include this test case are the large discontinuities in the problem. Most schemes can calculate Sods tube problem, which has small discontinuities but fail in calculating the Blast wave interaction correctly.

Name	t_{end} (s)	CFL	Gasses
Sod tube problem	0.4	0.8	$(\rho,v,p)=\begin{cases}(1,0,1) & 0\le x\le 1\\(0.125,0,0.1) & 1<x\le 2\end{cases}$
Blast wave interaction problem	0.076	0.8	$(\rho,v,p)=\begin{cases}(1,0,1000) & 0\le x<0.2\\(1,0,0.01) & 0.2\le x<1.8\\(1,0,100) & 1.8\le x<2\end{cases}$
Shu-Osher problem	0.36	0.8	$(\rho,v,p)=\begin{cases}(3.8571,2.6293,10.333) & 0\le x\le 0.2\\(1+0.2\sin(25x),0,1) & 0.2<x\le 2\end{cases}$
Strong rarefaction problem	0.24	0.5	$(\rho,v,p)=\begin{cases}(1,-2,0.4) & 0\le x\le 1\\(1,2,0.4) & 1<x\le 2\end{cases}$

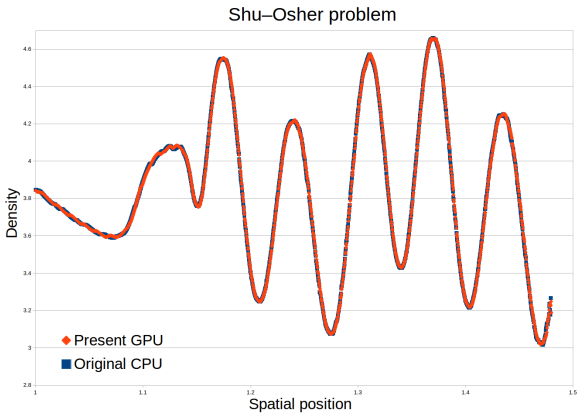
Table 5.1: The intial conditions for the four test cases



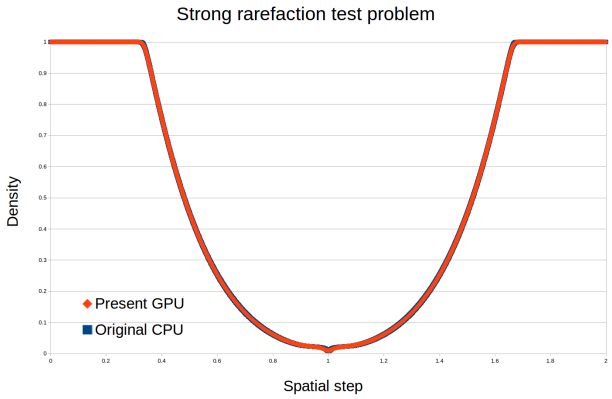
(a) The Sod tube problem



(b) Interaction of two blast waves



(c) Shu-Osher problem



(d) Strong rarefaction test problem

Figure 5.1: A comparison between the CPU and GPU implementation of the CE/SE scheme in 4 different cases

Also this problem has reflective walls, that is, a wave at the borders of the space the coordinate system cannot go out, but will be reversed. The third test case is the Shu-Osher problem, which of the interaction between a supersonic shock and an entropy sine wave. Last, the strong rarefaction problem consists of two waves moving in opposite direction, causing a near-vacuum to form in the center. The density and pressure should be positive at all times, however, at CFL of 0.8, the CE/SE scheme fails to satisfy this condition. Therefore, the CFL is lowered to 0.5 to run this simulation.

The output density of the four test cases for GPU implementation and the validated CPU program and are plotted in figure 5.1a to 5.1d appear. The red and the blue line are the output density of the GPU and the CPU respectively. As can be seen, the GPU implementation visually matches the CPU implementation for all test cases. This makes it likely that the GPU program has been implemented correctly. To strengthen this belief, the numerical outcome of all the test cases on GPU and CPU have been compared, and no differences have been found.

5.1.2 Runtime analysis

The quantitative analysis consists of a comparison in runtime between the sequential CPU and parallel GPU implementations. The influence of three variables on the runtime has been examined: the mesh size, CFL number, and the block size. The three properties will be individually discussed below. For the project, 1 CPU computer and three different Nvidia's GPU's were available. The specifications of the GPU's and the CPU's are given Appendix B. The GPU's are referred to by their official version names, namely Tesla K20, Tesla K40 and Tesla K80. To obtain valid results, we must assure that the comparison between the CPU and the GPU is fair. Ideally, one would compare a fully optimized parallel CPU algorithm to a fully optimized GPU algorithm. Unfortunately, a parallel CPU implementation is not available in 1D. Also, due to time limitations, it was not possible to implement this as a part of this project, so a serial algorithm has been used. This is important to keep in mind during the analysis of the results as a parallel C++ program would have a shorter runtime.

Now, we will examine the influence of each of the variables mentioned above.

Mesh size

The mesh size has the highest influence on the execution time of the algorithm. On the CPU, the runtime is proportional to the mesh size squared. Every time step, all the spatial positions are being updated. If the mesh size doubles, the number of time steps doubles to satisfy the CFL condition, and the number of spatial positions is doubled as well. The calculations of all the spatial positions for the new time step have been parallelized on GPU, so the execution time on the GPU is linearly dependent on the mesh size, according to the theory. Figure 5.2 shows the actual runtime on all the systems with respect to the number of spatial steps. This confirms the hypothesis that the execution time rises exponentially on the CPU and the GPU's more or less linear. Figure 5.3 shows the same graph, only with the CPU data points removed, to be able to better analyze the GPU runtimes. As can be seen, the graph can be divided into two parts, with its border at 78.000. After this number, the runtime rises much faster than before. Most likely, this is the point where the resources of the GPU are completely occupied, and it starts to execute blocks serially. It is also the point where the maximum speed-up can be reached, as shown in figure 5.4. The speed-up has been defined as the runtime on the CPU divided by the runtime on the GPU. The allocations and data transfers have been included for fairness. The maximum speed-up of the GPU over the CPU is around 20.

All measurements on the GPU's have been repeated ten times, and the averages have been taken to plot the figures 5.2-5.4. Error propagation has been omitted because the added accuracy is not required in this research. The step size was 500 mesh points, the CFL number was 0.4, and a block consisted of 256 threads. Sod's shock tube has been used as test case.

CFL number

As figure 5.5 shows, the runtime is inversely proportional to the CFL number. This is line with the theory, since $\Delta t \leq \frac{C\Delta x}{u}$. If C is being doubled, Δt becomes twice as large, so the number of time steps is being halved, which in turn doubles the runtime. The measurements have been performed ten times, and the average has been taken. Again, error propagation has been omitted. During all the simulations, the mesh size was 20.000 and the block size 256.

Block size

On a GPU, threads are being grouped into blocks, and this block size can vary. Since the optimal block size can differ for various sizes of the problem not only the block size is varied,

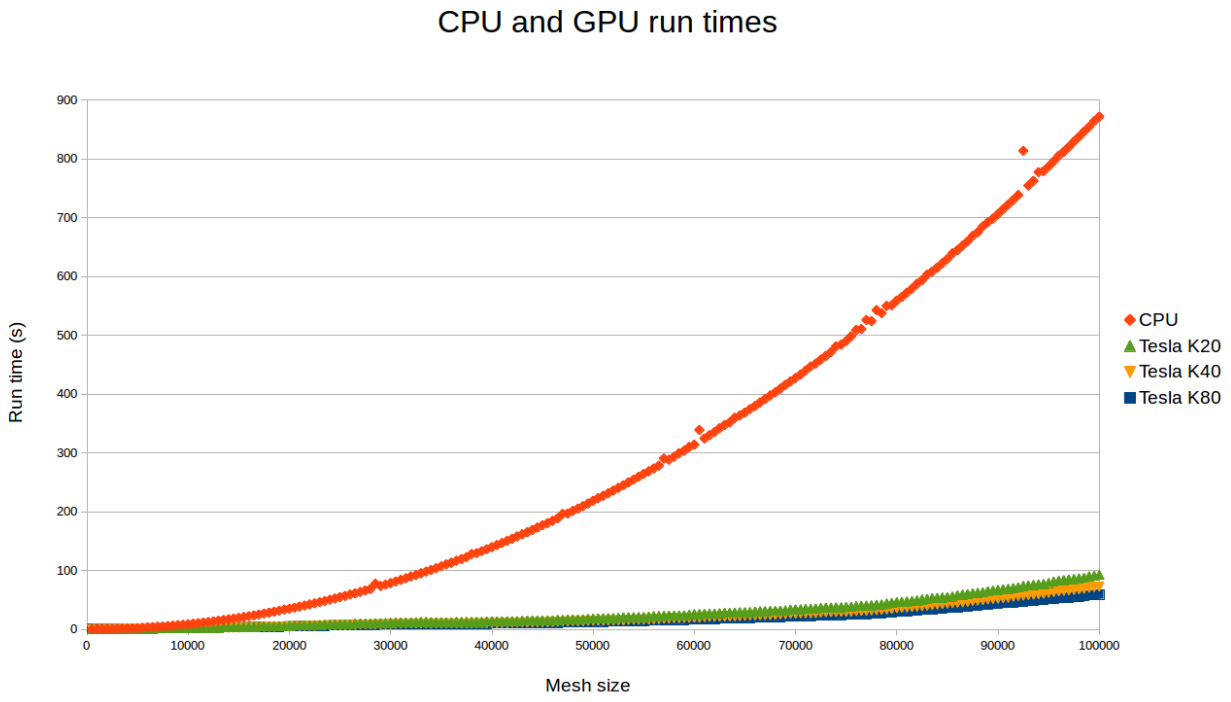


Figure 5.2: The runtimes of the CPU and GPU's for various mesh sizes

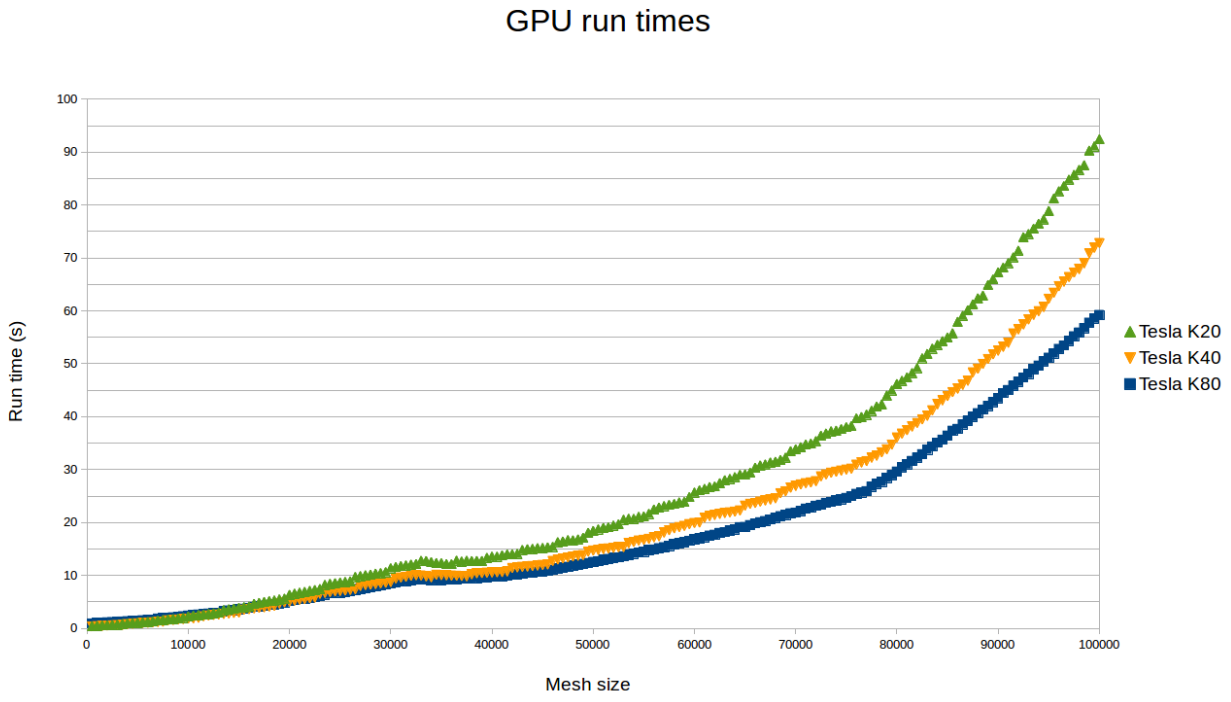


Figure 5.3: The runtimes of the GPU's for various mesh sizes

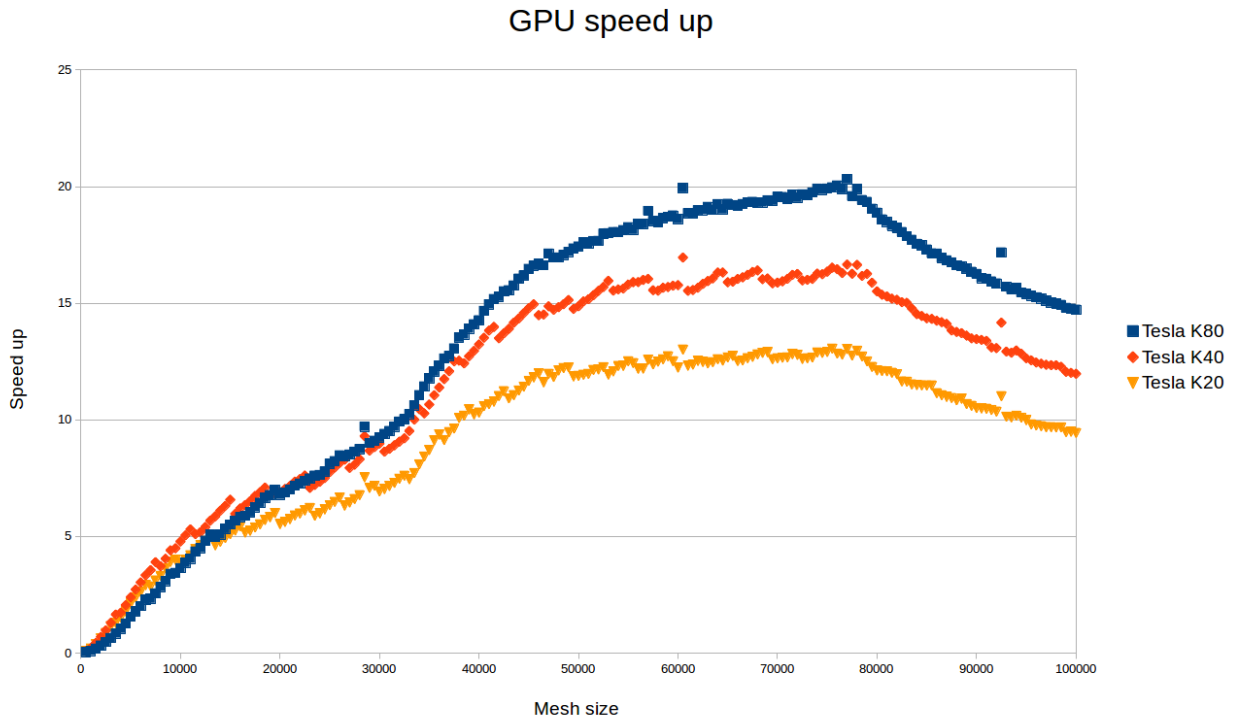


Figure 5.4: The speed-up of the GPU's for various mesh sizes

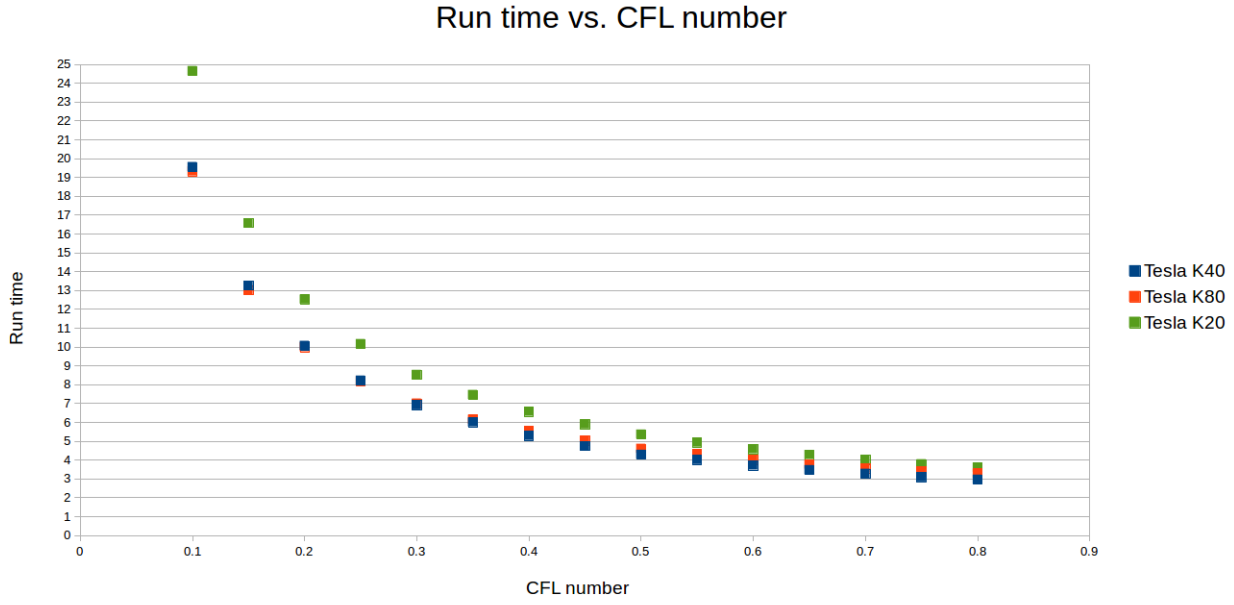


Figure 5.5: The runtimes of the GPU's for various CFL numbers

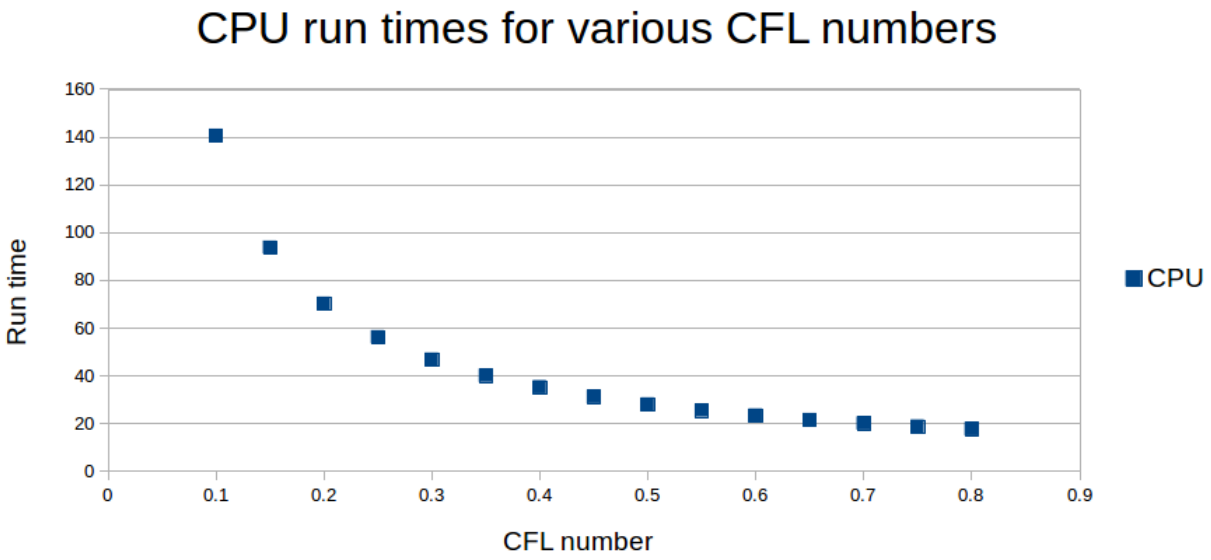


Figure 5.6: The runtimes of the CPU for various CFL numbers

but also the mesh size. The mesh sizes are increased by steps of 5000, and for each size, the block size is varied between 32, 64, 128, 256. An optimal block size is always a multiple of 32 because threads can only be launched in warps, which are groups of 32 threads. So if a block consists of 33 threads, still two warps (64 threads) are launched, and most threads will be idle. Also, on Tesla K20 and Tesla K40, the number of threads per block cannot exceed 256, because the shared memory to store the associated data is too small. Therefore the measured block sizes are between 32 and 256. As usual, the measurements have been repeated ten times, the average has been taken, and error analysis has not been performed. All the simulations have been run on Tesla K40. The results have been plotted in figure 5.7. It should be noted that in this plot, the lines between the data points are only to improve the readability of the graph. In no way, they represent a model or actual experimental outcomes.

It is interesting to see that for a block size of 256 threads, the runtime increases a lot from 75000 and higher, as was also observed in section 5.1.2. A block size of 128 or 64 threads shows a much more steady growth. This might be because the block size of 256 threads launches the fewest blocks, which causes the least overhead. At the same time, the shared memory is near the physical limit of the device, which makes it less flexible for optimizations at larger mesh sizes.

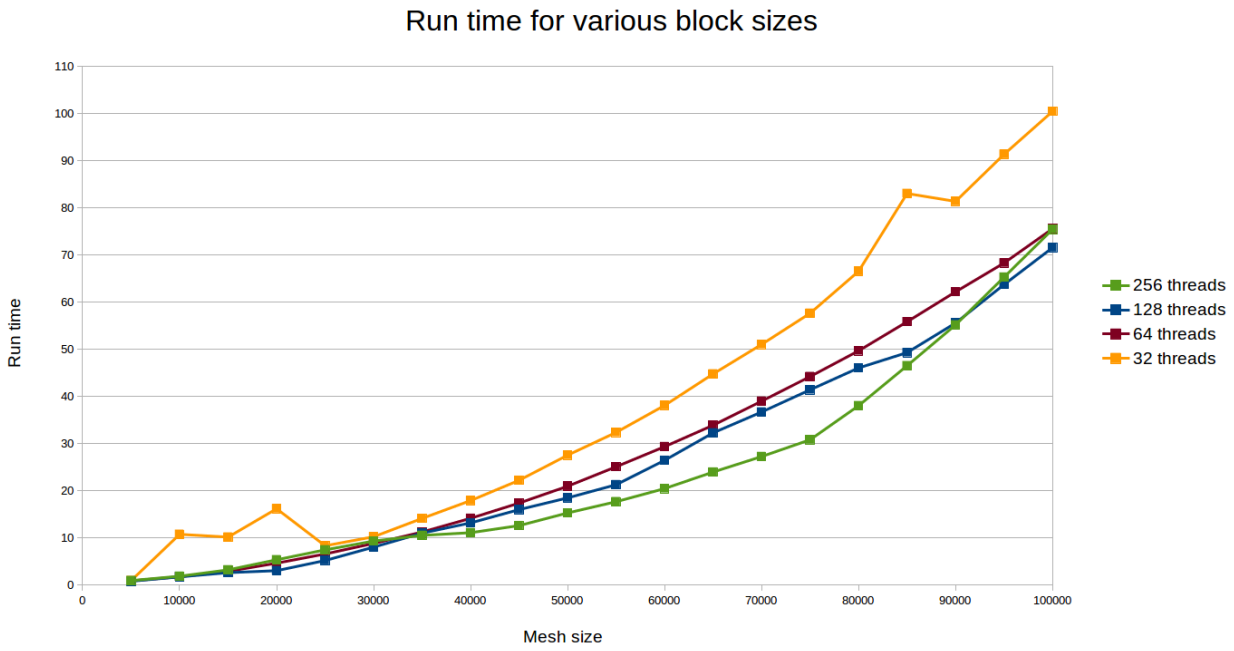


Figure 5.7: The runtimes of the GPU's for various block and mesh sizeson Tesla K40

5.1.3 Results

It can be concluded that the GPU implementation is faster than the CPU version, because it grows much less fast than the CPU version. Only at small meshes (less than 5.000 data points), CPU is faster, because the initialization of the data and the kernels takes more time on a GPU. As expected, the runtime is inversely proportional to the CFL number on all systems. Finally, in most cases on a Tesla K40 card, a block size of 256 threads is the best option, as this resulted in the shortest execution time. For very low and high numbers, the differences between 64, 128 and 256 threads were small. It can be concluded that the GPU implementation has been successful, as it provides a good speed-up over the existing CPU implementation. Under optimal conditions, the new version was approximately 20 times faster than the old version.

5.2 2D analysis

5.2.1 Validation

The following initial condition was used for validating the 2D GPU code:

$$(\rho, u_x, u_y, p) = \begin{cases} (1.50,0.0,0.0,1.5) & 0 \leq x \leq 0.5, 0 \leq y \leq 0.5 \\ (0.5323,1.206,0,0.30) & -0.5 \leq x < 0, 0 \leq y \leq 0.5 \\ (0.138,1.206,1.206,0.029) & -0.5 \leq x < 0, -0.5 \leq y < 0 \\ (0.5323,0.0,1.206,0.3) & 0 \leq x \leq 0.5, -0.5 \leq y < 0 \end{cases} \tag{5.1}$$

Visual validation is a good estimate to determine whether the code is correct, but since differences in colors can be hard to distinguish, a numerical approach will be taken. During the first few hundred iterations, the outcome in the GPU and CPU are completely identical. When the simulations runs longer, small differences between the outcome on the CPU and GPU appear. This starts at the 16th digit and this slowly expands to lower decimal places. In all the test cases that have been run, the CPU and GPU gave identical output for the first 10 digits. A lot of effort has been put in avoiding these differences, but no clear solution has been found. The phenomenon of different output on GPU and CPU has been mentioned in other literature, such as Whitehead and Fit-Florea (2016) and is discussed more extensively in appendix A. Due to these differences it is impossible to say whether the code is correct. All we know is that for every simulation, the output of the double-precision GPU and the double-precision CPU are identical up to the 10th digit. In the rest of the report, the code is assumed to be correct, or at least that the runtimes approximate the runtimes of a correct GPU implementation.

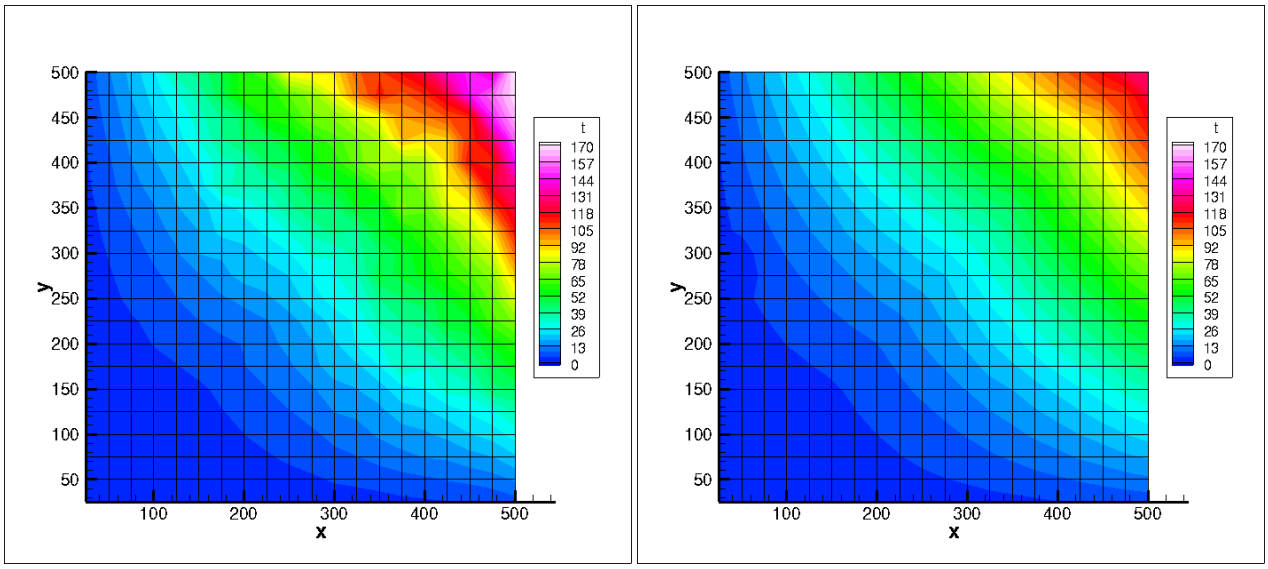
5.2.2 Runtime analysis

Just as in the 1D case, the runtime is dependent on three factors: the grid dimensions, the CFL number, and the block dimensions. Before discussing the influence of these factors individually, one important note must be made. The test cases in two dimensions are much bigger and require much more computing power. During an analysis of the GPU code, it was found that GPUs

are very slow in handling double precision floating points. Actually, the use of double precision one of the main bottlenecks. Therefore, for completeness, also the runtimes of a single precision version have been included. The single-precision for CPU has not been included since it was found that this version runs at almost exactly as fast as the double precision version. Therefore, including these extra simulations does not give relevant data for the comparison. The user has to decide whether the extra runtime of double precision is worth the extra precision.
All measurements have been repeated five times, and the average has been taken. Just as in the 1D case, error propagation has been omitted.

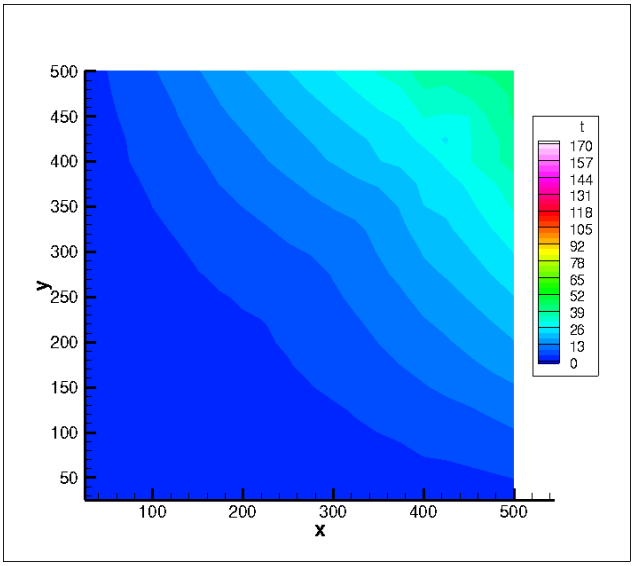
Grid dimensions

The grid dimensions are now split into two variables, `Nx` in the x-direction and `Ny` in the y-direction. The runtimes have been determined in the range from 25x25 to 500x500 data points, with a step size of 25 for each variable. Figure 5.8 shows the results of the measurements in a contour plot for the measurements on an OpenMP CPU, a double precision GPU, and a single precision GPU. For the OpenMP simulation, 16 threads were used and the `-O3` optimization flag was applied. The intersections of the lines on the plot correspond to the points at which measurements have been taken. The colors in between are generated by the program to display the data and do not correspond to a model or experimental data. To make the differences clearer, the values of `Ny` for `Nx` = 500 are plotted in figure 5.9a. Figure 5.9b shows the speed-up defined as the time the parallel CPU algorithm takes divided by the time the GPU algorithm takes. For fairness, all memory transfers and allocations have been included in the measurement of the runtime. Also, for both GPU and CPU relatively new and high-end devices have been used. (see B for the device specifications)
As can be seen in the figures mentioned above, the speed-up for double precision is around 1.4 if the number of data points is sufficiently large. For single-precision, the speed-up is growing from 3 to 4.3, but the larger the mesh, the higher the speed-up. It can be seen in figure 5.8 that meshes for which `Nx=Ny` are processed faster by the GPU than rectangular meshes. This may increase the speed-up even more. Therefore it can be concluded that the GPU is significantly faster than the CPU in single precision, while the performance gain in double precision is smaller.
A last interesting note is that the runtime on GPU is much more constant than on CPU. The



(a) The runtimes on the CPU

(b) The runtimes on the GPU in double precision

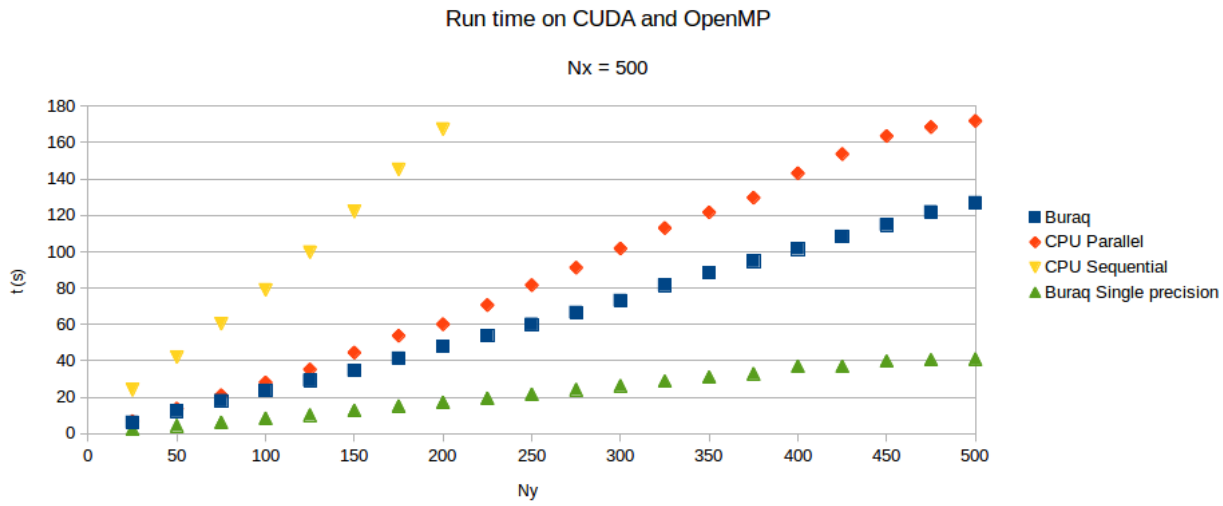


(c) The runtimes on the GPU in single precision

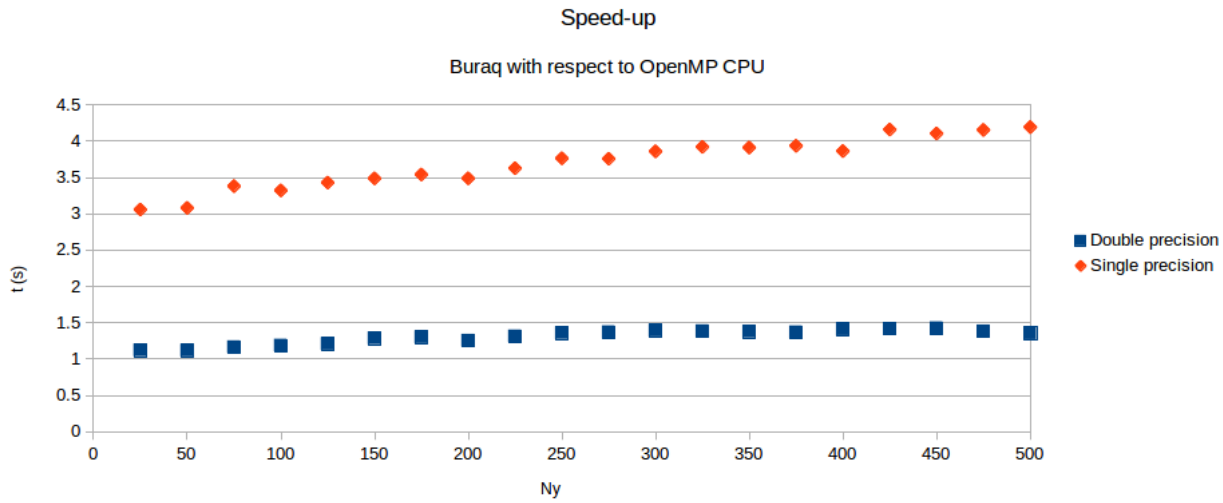
Figure 5.8: The runtimes for grid sizes as a contour plot. On the axis: `x=Nx` and `y=Ny`

average standard deviation in the results on CPU was 2.06 while on GPU it was 0.075. Also, the contour plot for the GPU is more symmetric along `Nx=Ny` than the CPU. Even though no extensive analysis on this topic has been performed, these two observations indicate that the runtime on the CPU is less predictable than on the GPU.

All measurements have been taken with a CFL number of 0.5 and $t_{end} = 2.0$



(a) The runtimes for Nx=500



(b) The speed-up for Nx=500

Figure 5.9: The runtimes of the GPU and the CPU compared for Nx = 500

CFL Number

Just as in the 1D CE/SE scheme, the runtime is expected to be inversely proportional to the CFL number. The results shown in figure 5.10 confirm this hypothesis. Because the execution time of the `CESE` kernel is slightly shorter on the GPU than on the CPU, the difference between the CPU and GPU is larger if the CFL number is small. In this experiment, all measurements have been taken with `Nx=Ny=250` and blocks consisting of 8x8 threads.

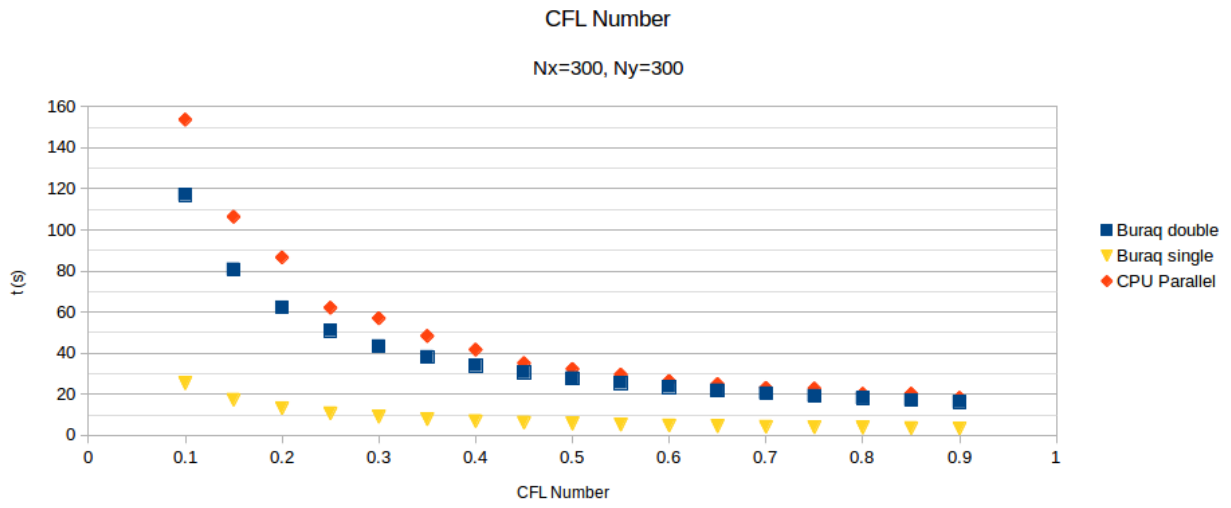


Figure 5.10: The runtime for various CFL numbers

Block dimensions

The number of threads per block is limited by the memory on GPU and the maximum number of threads per SM. Memory size turns out to be the limiting factor, and in order to maximize the number of threads per SM, four threads are assigned per data point (one per conserved property). On the Tesla K40 GPU (see appendix B for specifications), the shared memory can store the data for up to 64 mesh points, which is 256 threads per block.

An analysis of the executable in the Nvidia Profiler results in the plot given in figure 5.11. Also, the runtimes for several block dimensions are shown in 5.2. The latter also shows the standard deviation, to ensure that the values obtained have enough precision to draw conclusions based on them. As can be seen, for the test case (`Nx=Ny=250`, CFL=0.5, $t_{end} = 2.0$), the most optimal block distribution is a 8x6 grid. The most probable explanation for this is that the size of the blocks are large enough to make as many threads work in parallel, while not being too big in the y-direction. The data is stored in row-major order, so if a block is long in the y-direction, it will have more cache misses than when it has more elements in the x-direction. This might explain why the table 5.2 is not symmetric along the diagonal. In this specific test case, the grid is square-shaped. Taking a rectangular shape instead may influence the optimal block dimensions. More research can be done into this although the performance gain is small compared to the effort it takes to investigate it. Therefore, only the rectangular shape has been tested.

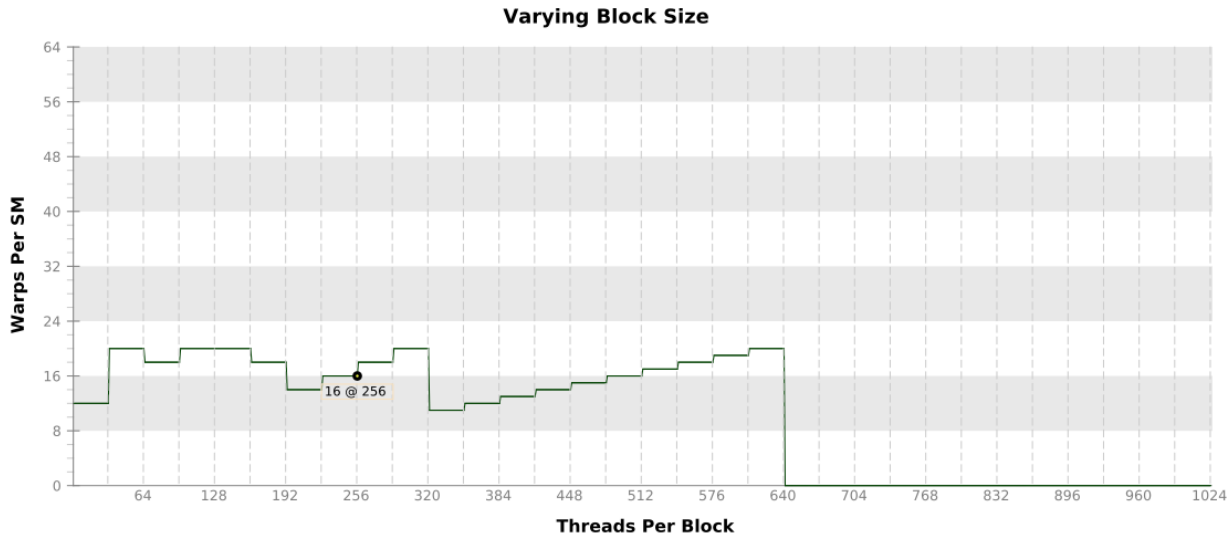


Figure 5.11: The theoretical number of warps per thread for different block sizes

y \ x	4	6	8
4	6.13 ± 0.002 s	6.74 ± 0.01 s	5.77 ± 0.02 s
6	6.77 ± 0.006 s	6.16 ± 0.008 s	5.49 ± 0.003 s
8	5.94 ± 0.03 s	5.68 ± 0.03 s	5.61 ± 0.03 s

Table 5.2: Runtimes for block dimensions on GPU, using double-precision

5.2.3 Results

Just as in the 1D case, there are three independent variables that influence the runtime besides a t_{end}. These are the grid dimensions, CFL number and the block dimensions. It was observed that the number of threads per block has a small influence on the runtime. In general, the larger the block size, the better the performance is. However, the influence is not big and may depend on the grid dimension as well.

The runtime is inversely proportional to the CFL number, as expected based on the theory. Because the number of time steps increases and the execution time of each step is a little shorter on the GPU than on the CPU, the GPU is especially faster than CPU if the CFL number is small.

Three main conclusions can be drawn from the measurements for various grid dimensions. First, it can be observed that if both the block and the grid dimension are square-shaped, then the runtimes on GPU are slightly shorter than other rectangular grids. Secondly, the results on

the GPU are more consistent that the results on GPU, that is, the standard deviation of the run-times on the GPU was much smaller than on the CPU. Finally, the GPU is faster than the CPU, for both double and single precision. When using double precision, a CFL number of 0.5 and $t_{end} = 2.0$, the maximum speed-up is 1.42, and 4.21 under the same conditions, but using single precision. This magnitude of speed-ups is in line with what can be expected for comparison with parallel CPU algorithms, according to Lee et al. (2010).

6. Summary and Recommendations

6.1 Summary and Conclusions

In the first chapter of this thesis, the four objectives of this thesis were formulated. The first two stated that the 1D and 2D CE/SE scheme had to be implemented on the GPU. The other two objectives were to check the validity of the GPU implementation and the speed-up compared to the existing implementations on CPU. To understand the GPU implementations, the 1D and 2D CE/SE scheme have been discussed in chapter 3. Both CE/SE schemes have been converted to an algorithm in pseudo-code and have been implemented on GPU. This process and the results have been presented in chapter 4.

The next objective was to validate the new implementations using the original implementations on CPU, which are described in Shen et al. (2015). The 1D code could be validated, and the results matched the results on the CPU. The 2D code was harder to validate, due to small differences between the GPU and CPU, but these are explained in appendix A.

Finally, the runtimes of the GPU and the CPU implementations have been compared. For 1D, where no parallel implementation on CPU is available, it can be seen that while the CPU runtime rises exponentially and the GPU runtime rises almost linearly. This leads to a great speed-up, and in some measurements, the GPU was more than 20 times faster than the CPU. For 2D, a parallel CPU implementation was also available. Still, for all test cases, the GPU was faster than the CPU. For double precision, the maximum measured speed-up was 1.42. Because it is known that GPU is slow at performing the double precision floating point operations, a single precision version has also been implemented. Then, the performance gain of the GPU over the CPU can be as high as a factor of 4. This is in line with what could be expected based on Lee et al. (2010). Therefore, it can be concluded that all the objectives for this thesis have been achieved and that the implementation on the GPU has been successful.

6.2 Discussion

Programming on GPUs and measuring performance should always be done with a lot of caution. Minor changes can cause a huge performance drop or even incorrect results. Therefore, most likely, the code written as part of this thesis will have possibilities for improvement. The speed-ups mentioned in this thesis are therefore not the maximum achievable values but should be read as indicators that the algorithm is suitable for implementation on GPU. Moreover, comparing the performance of a program on CPU and GPU is hard, because there is no clear benchmark to compare the CPU and GPU. In this thesis, some of latest models of GPU and CPU have been used and the specifications have been listed in appendix B. In this way, it is made clear how the results have been obtained, and the user of the code can decide which implementation to use.

Finally, a parallel version of the 1D code did not exist, and therefore the comparison between the GPU and the CPU is not fair. This has been mentioned clearly during the presentation of the runtimes in chapter 5.

6.3 Recommendations for Further Work

The research in this thesis was limited to structured grids. Further research could focus on extending the code to unstructured meshes, so the code can be used to analyze flows around objects. The main challenge would be to organize the code in such a way that threads can be assigned to the mesh points in an efficient and organized way.

Another way to extend this research would be implementing a 3D version or a version that uses the Navier-Stokes equation, instead of the Euler equation. This requires new models for storing memory and thread communication, to overcome the limitations of the GPU.

Last, the implementation written for this thesis can be improved by optimizing the code more. For example, creating a version which uses surface memory instead of shared and global memory could potentially be faster than the current version. Also, extending the implementation to use multiple GPUs could increase the performance of the algorithm drastically.

Appendix A: Floating point operations on GPU

During the validation of the GPU implementation, it was found that exactly the same operation can produce different results on a GPU and a CPU. This is a result of the different architecture and a different arrangement of commands in the created assembly code, as explained in Whitehead and Fit-Florea (2016). As an example, consider the function to calculate the weighted average of two numbers:

$$f(a,b) = \frac{a^2 b + b^2 a}{a^2 + b^2 + \epsilon} \tag{A.1}$$

where ϵ is a small integer added to avoid division by zero. The value of the numerator on the GPU can be different from value on the CPU given the same input values. This small different can be important, if the denominator is also small, which increases the error in further calculations.

This phenomenon actually happens in 2D CE/SE algorithm in the implementation of the function W_0. In every occurrence, the GPU was more precise than the CPU. A probable reason for this might be that the GPU has a hardware build-in fused-multiply-add (FMA) operator. To confirm this, the function has been replaced by a new version, using intrinsics to disable FMA on the device. This fixed the problem most of time, except for special cases where the rounding depended on more decimals than a double can store. Still, given the large number of operations and size of the meshes, this occurs many times. Since intrisics slow down the program and do not increase the accuracy, they have not been used in the final version of the program.

A second observation concerns the use of single float operations in double precision calculations. As a part of the troubleshooting process in validating the GPU implementation, NVPROF was used to count the number of single and double precision operations. It turned out that a program that only used double precision, executed many single precision operations. Checking the assembly code, it could be seen that these single precision operations were approximations

of the reciprocal (the `MUFU.RCP64H` instruction). The explanation is that CUDA does not have a hardware implementation for double precision division and therefore approximates this by using multiple single precision reciprocals. A second observation was that if instead of divison, multiplication of a reciprocal was used, CUDA only uses double precision operations. Therefore, if, for any reason, a programmer wants to be sure that CUDA uses double precision, then all division must be replaced by a multiplication with a reciprocal. That is:

```
1 A = B/C;
2 //should be replaced by
3 A = B*__drcp_rn(C);
```

Appendix B: Device specifications

B.1 The CPU system

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                56
On-line CPU(s) list:   0-55
Thread(s) per core:    2
Core(s) per socket:    14
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Stepping:              1
CPU MHz:               1200.000
BogoMIPS:              4801.59
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              35840K
NUMA node0 CPU(s):     0-13,28-41
NUMA node1 CPU(s):     14-27,42-55
```

B.2 The GPU system

B.2.1 Tesla K20

```
Major revision number:         3
Minor revision number:         5
Name:                          Tesla K20c
Total global memory:           5306777600
Total shared memory per block: 49152
Total registers per block:     65536
```

```
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    705500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     13
Kernel execution timeout:      No
```

B.2.2 Tesla K40

```
Major revision number:         3
Minor revision number:         5
Name:                          Tesla K40c
Total global memory:           11995054080
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    745000
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     15
Kernel execution timeout:      No
```

B.2.3 Tesla K80

```
Major revision number:         3
Minor revision number:         7
Name:                          Tesla K80
Total global memory:           11995578368
Total shared memory per block: 49152
```

```
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    823500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     13
Kernel execution timeout:      No
```

Bibliography

Chang, S.-C. (1995). The method of space-time conservation element and solution element - a new approach for solving the navier-stokes and euler equations. *Journal of Computational Physics*, 119:295–324.

Chang, S.-C. et al. (1998). Fundamentals of cese method. NASA Technical Memorandum 1998-208843. https://www.grc.nasa.gov/WWW/microbus/cese/PUBS/ltxjcp2d.pdf.

Chang, S.-C., X.Y., W., and Chow, C. (1999). The space-time conservation element and solution element method– a new high resolution and genuinely multidimensional paradigm for solving conservation laws. *Journal of Computational Physics*, 156:89–136.

Chena, Y.-Y. et al. (2011). Solvcon: A python-based cfd software framework for hybrid parallelization. *49th AIAA Aerospace Sciences Meeting*.

Fang, J., Varbanescu, A. L., and Sips, H. (2011). A comprehensive performance comparison of cuda and opencl. *ICPP, 2011*, page 216–225.

Gregg, C. and Hazelwood, K. (2011). Where is the data? why you cannot debate cpu vs. gpu performance without the answer. *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144.

Harris, M. (2012). Optimizing parallel reduction in cuda. http://vuduc.org/teaching/cse6230-hpcta-fa12/slides/cse6230-fa12–05b-reduction-notes.pdf. Accessed on October 23, 2016.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al. (2010). Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH Computer Architecture News*, 38(3):451–460.

Nvidia CUDA ™ (2015). *CUDA C Programming Guide 7.5*. Nvidia Corporation, pg-02829-001_v7.5 edition. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Nvidia ™ (2016). *THRUST QUICK START GUIDE*. Nvidia Corporation, du-06716-001_v8.0 edition. http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf.

Shen, H., Wen, C.-Y., and Zhang, D.-L. (2015). A characteristic space–time conservation element and solution element method for conservation laws. *Journal of Computational Physics*, 288:101–118.

Wei Ran, Cheng, W., Qin, F., and Luo, X. (2011). Gpu accelerated cese method for 1d shock tube problems. *Journal of Computational Physics*, 230:8797–8812.

Whitehead, N. and Fit-Florea, A. (2016). Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf. Accessed on October 24, 2016.