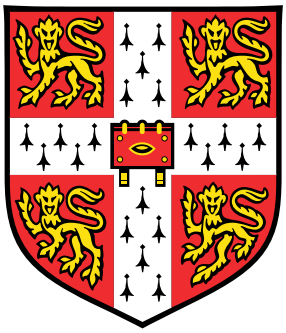
Computational Fluid Dynamics with Embedded Cut Cells on Graphics Hardware



Alo Roosing

Department of Physics

University of Cambridge

This dissertation is submitted for the degree of

Doctor of Philosophy

Supervisor

Dr Nikos Nikiforakis

St Edmund's College

September 2018

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text.

It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text.

It does not exceed the limit of sixty thousand words prescribed by the Degree Committee for Physics and Chemistry.

Alo Roosing
September 2018

Abstract

Computational Fluid Dynamics with Embedded Cut Cells on Graphics Hardware

Alo Roosing

The advent of general purpose computing on graphics cards has led to significant software speedup in many fields. Designing code for GPUs, however, requires careful consideration of the underlying hardware. This thesis explores the implementation of fluid dynamics simulations featuring embedded cut cells using the CUDA programming platform. We demonstrate efficient generation and handling of geometric surface data in rectilinear computational grids. This is added to a split Euler solver to define piecewise linear cut cells describing solid surfaces in fluid flows. To reduce the memory footprint of embedded boundaries, we present a system of compressed data structures. The software is extended to run on multiple graphics cards and shows good scaling.

Simulating embedded boundaries requires a description of object surfaces. We implement a fast and robust narrow band signed distance field generator for graphics cards based on the Characteristic/Scan Conversion algorithm for stereolithography files. The thesis presents an augmented approach to handle commonly occurring complex configurations and we show that the method is correct for all closed surfaces. We discuss efficient feature construction and work scheduling and demonstrate high-speed distance generation for complex geometries.

At the core of our simulation implementation is a split Euler solver for high-speed flow. We present a one-dimensional method that achieves coalesced memory access and uses shared memory caching to best harness the potential of GPU hardware. Multidimensional simulations use a framework of data transposes to align data with sweep dimensions to maintain optimal memory access. Analysis of the solver shows that compute resources are used efficiently.

The solver is extended to include cut cells describing solid boundaries in the domain. We present a compression and mapping method to reduce the memory footprint of the surface information. The cut cell solver is validated with different flow regimes and we simulate shock wave interaction with complex geometries to demonstrate the stability of the implementation.

We conclude with multi-card parallelisation and analyse existing literature on domain segmentation and GPU communication. We present a system of domain splitting and message passing with overlapping compute and communication streams. A comparison of naïve and GPU-aware Open MPI shows the benefits of using CUDA specific library calls. The complete software pipeline demonstrates good scaling for up to thirty-two cards on a GPU cluster.

Acknowledgements

I would like to express my sincere gratitude to Dr Nikos Nikiforakis for his guidance and support throughout my time at Cambridge. I am grateful to Dr Philip Blakely for his development and support work at the LSC and for always being helpful. My implementation of the cut cell algorithm would not be possible without the support of Nandan Gokhale who always found time to answer my questions. The SDF generator code benefitted from enlightening discussions with Oliver Strickson who also helped provide a proof of completeness for the underlying algorithm. I thank Jeffrey Salmond for his help with MPI and the Wilkes2 cluster. I would also like to thank Lukas Wutschitz for his insightful feedback on early drafts. I am grateful to have met the members of the LSC whose friendship has made my time at Cambridge immeasurably better.

Contents

Chapter 1

Introduction

Computational modelling of fluids and gases is of great interest in many academic and industrial fields. Numerical study of fluid dynamics has applications from quick and cheap analysis of complex phenomena to prototyping machines and experiments. Based on mathematical models and experimental measurements, computational fluid dynamics (CFD) has become an important part of scientific research. The emergence of powerful computer hardware and novel algorithmic techniques makes it an actively evolving field.

This thesis documents the research undertaken in partial fulfilment of a PhD degree in Physics at the Laboratory for Scientific Computing at the University of Cambridge. In this introductory chapter we offer a brief overview of the nature of CFD and list the novel contributions of the work. We discuss signed distance field generation, considerations of parallel programming with a focus on the CUDA platform, different meshing approaches and the use of multiple graphics cards.

CFD can be approached in a variety of ways depending on the investigated phenomenon and the mathematical formulations employed. We investigate shock-capturing methods with embedded boundaries using approximate Riemann solvers on graphics processing units (GPUs). The main focus of the research outlined in this thesis was to improve the memory footprint and performance of fluid simulations by speeding up mesh generation, using efficient data layouts and implementing efficient numerical work on graphics cards.

1.1 Motivation

Since its inception, fluid dynamics research has undergone major paradigm shifts. What started as purely theoretical work to explain observable phenomena evolved to include intricate experimental work applicable to the design and construction of new machines and vehicles. With the

introduction of computers the field now includes a third equally significant branch of computational fluid dynamics. All three aspects continue to undergo developent with improvements in formulations, accuracy and performance. The emergence of graphics processing units and interfaces for their utilisation in general purpose computing has introduced another paradigm shift in the field allowing for significant performance increases. This new platform has found use in academia as well as industry and a large amount of literature exists discussing a variety of strategies, bespoke formulations and best practices.

Our research was motivated by the lack of literature on complete software pipelines combining fast mesh generation, the split cut cell solver by Klein et al. [28] and multi-GPU simulations for high speed shock-capturing methods. These methods are applicable to a multitude of research projects from cityscape detonation analysis to supersonic flow around complex geometries and reactive processes in homogeneous explosives. There is always an interest in shorter runtimes from both automated grid generation as well as increased parallelisation without sacrificing numerical and physical accuracy. Literature exists on the individual components presented in this thesis but we discuss the interaction of the different parts and present a combined simulation pipeline implementation with increased robustness and speed for geometry generation, reduced memory footprint for boundary representation and an analysis of multi-card scaling. Due to the particularities of GPU hardware, it is not sufficient to simply translate existing CPU codes into CUDA. Indeed doing so can often lead to significantly slower software. We have thus developed most of the code from scratch, using libraries and existing optimal solutions as mentioned in the text.

We are motivated both by increased performance as well as how the different components best fit together and we discuss some of the implications of large-scale CFD simulations with embedded boundaries on massively parallel architectures with a focus on data structures, memory access patterns and sparse storage. The novel contributions of this thesis are:

- A fast and robust CUDA implementation of the CSC signed distance field algorithm by Mauch [36]
- A CUDA implementation of the split cut cell solver by Klein et al. [28] and the LPFS extension by Gokhale et al. [21]
- A discussion of the data structures and memory layout of sparse cut cell data on GPUs
- An investigation of the properties and design of large-scale multi-card simulations of 3D shock-capturing problems with cut cell boundaries

1.2 Fluid dynamics

The study of fluid dynamics has a long history before the invention of computers. The field includes equilibrium problems that feature little change over time and dynamic systems, with or without viscosity, which are all governed by the same physical rules that describe Newton's second law and the conservation of mass and energy. Their mathematical formulations, however, differ significantly.

Computational fluid dynamics seeks to present a system of materials within a spatial and temporal domain. This allows for cheap exploration of user-defined problems with flexible constraints and arbitrary materials. A collection of mathematical models describes the change in physical variables and determines the behaviour of the simulated material. Numerical methods are then used to advance the system in time and the resulting behaviour can be analysed.

A spatial domain is divided into discrete cells and the interactions of nearby sections during a series of time steps constitute a time-evolving system. The size and number of these cells determine the total amount of work done for each time step with smaller cells producing a more accurate solution.

Both equilibrium problems as well as the behaviour of discontinuous waves are described by systems of partial differential equations. These coupled equations express the influence of initial and boundary conditions on the evolution of systems of fluids or gases. With complex governing equations and large number of cells being used today, the total amount of time and computational resources taken up by a simulation have become bottlenecks for many scientific endeavours. The governing equations and the numerical methods we use to simulate high-speed fluids are described in Chapter 3.

1.3 Grid generation

The governing equations are solved on a grid of computational cells. This spatial discretisation greatly affects the algorithmic and programmatic approach to simulations and the numerical accuracy of the results. There are several ways to represent a domain discretely. The most intuitive is a regular Cartesian grid where for a domain of length L divided into M equal cells in some direction d, the side length of individual cells Δd is found by $\Delta d = \frac{L}{M}$.

Consider, however, the presence of complex geometries in the domain. When simulating flow around aerofoils, shock wave reflection from buildings or flow through the inside of an engine, we need to represent the computational cells in a way that differentiates between solid and fluid regions and handles the effects of geometry boundaries. Grid generation is a significant part of any CFD simulation featuring complex bodies. There are two main types of grids:

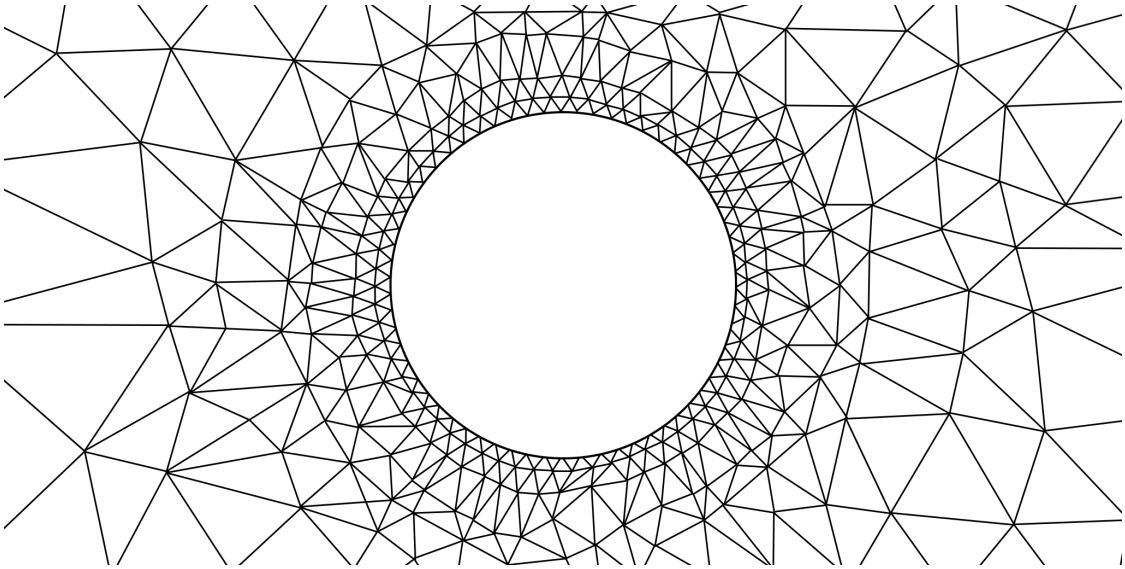


Fig. 1.1 Unstructured triangle mesh around a disc. The areas around surfaces often feature a higher number of smaller cells. The cells do not have a fixed number of neighbours and there can be great variation in their size.

Structured grids are logically Cartesian regular meshes. Every cell has a straightforward coordinate index and neighbours are easy to find. The shape of every cell in the computational domain is uniform but may be mapped to more complex physical spaces by a transform function.

Unstructured grids are a collection of cells without a regular arrangement. The shape of each cell can be arbitrary as can the number of neighbours. Common implementations use triangles or tetrahedra.

Unstructured

Unstructured grids can describe geometries very accurately as shown in figure 1.1. As the shape and location of the cells can vary, they can be generated to fit intricate surfaces faithfully. Such grids are popular in CFD simulations because of their accurate boundary representation and because they can be generated partly automatically. The generation of these grids for complex geometries, however, is not trivial and it is difficult to automate completely as user input is required to assess the suitability of the produced mesh. The interconnectivity of the mesh cells can also be very complex and difficult to generalise for different solvers and parallelisation. The meshes often feature high deformation and large differences in resolution. The cell areas and volumes are not uniform and neighbouring cell configuration can vary significantly.

Structured

Structured grids have the benefit of straightforward connectivity information and consistent indexing. They can describe embedded boundaries in several ways.

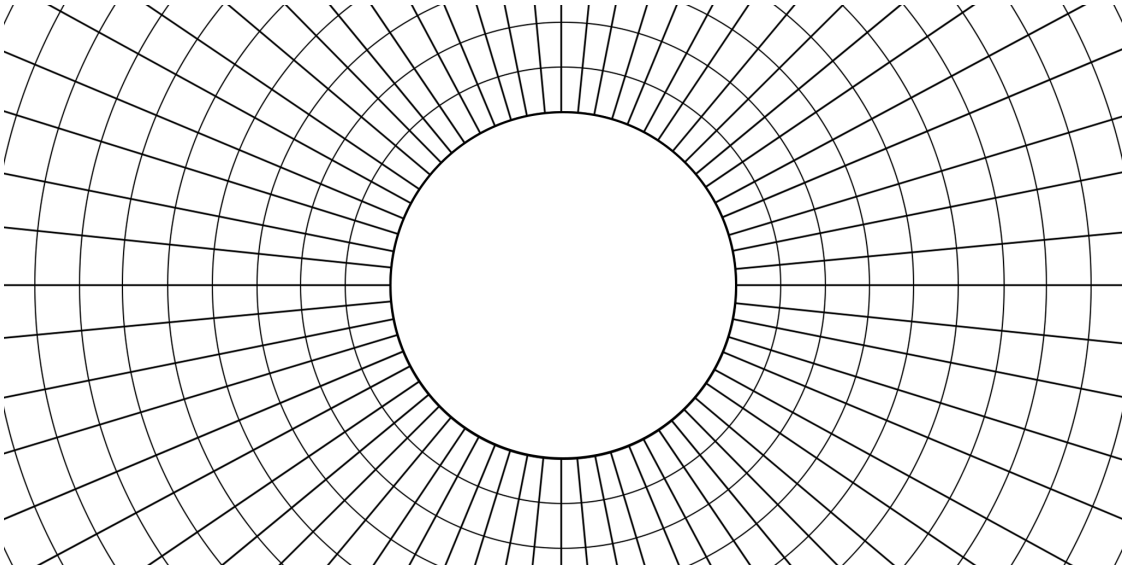


Fig. 1.2 Body fitted mesh around a disc. A regular Cartesian mesh has been mapped around a geometry. The cells retain their neighbours but can deform. While transforming a grid to wrap around simple convex shapes is straightforward, more complex geometries require non-trivial mapping and can lead to significant cell deformation.

Body-fitted meshes maintain a one-to-one mapping between a physical space and computational space. Each cell corresponds to a location relative to a boundary and also to a regular Cartesian grid. The mesh is wrapped around the geometry (figure 1.2) and a transformation function is used to switch between the coordinates of the two domains. The amount of deformation as a Cartesian grid is transformed to describe a complex geometry can lead to arbitrarily small cells and unequal work loads. Unique transforms between complex geometry and a regular mesh are difficult to find and may require long generation times.

Adaptive grids are rectilinear meshes which change the size of cells at specified locations. This may be due to the presence of boundaries or to track flow features. While the neighbour information of these grids is easily maintained, the change in local cell geometry adds to the work load and may require more communication between processes in a distributed system. The geometry representation will also always conform to the grid lines of the mesh.

Embedded grids describe the intersection of surfaces with a regular Cartesian grid. This includes methods like ghost fluid and cut cells. The computational domain is divided into equal cells which exist inside, outside or at the boundary of the geometry. The cells intersected by the boundary maintain additional information to adjust their fluxes, thereby simulating the effects of the object.

Ghost fluid approaches maintain a Cartesian grid and use a signed distance field around the surface of the geometry. These methods allow simulating both static as well as moving and

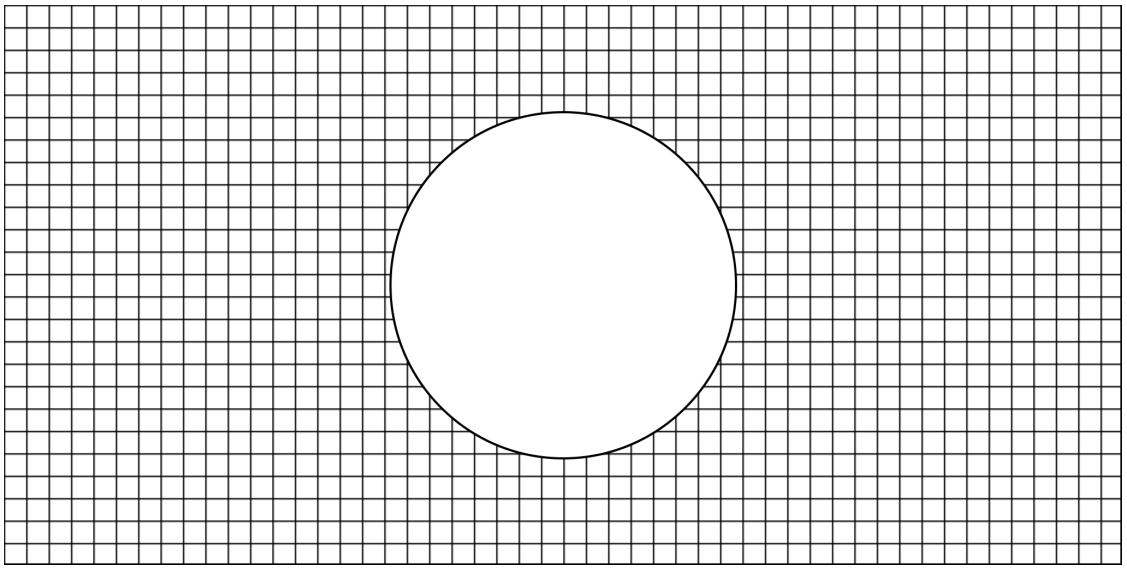


Fig. 1.3 Cut cell mesh around a disc. The underlying mesh is a Cartesian grid with equal rectangular cells. The cells intersected by the boundary hold additional geometry and flow information to simulate the effects of the object. Though clear cells have equal volumes, boundary cells can feature small volume ratios which can limit the global time step size.

deformable geometries while leaving the computational cells unchanged. While these find wide use, there are some drawbacks. The solution at the boundary is not necessarily conservative and the need to update a narrow band distance field propagated from the surface every time step can lead to poor parallelisation.

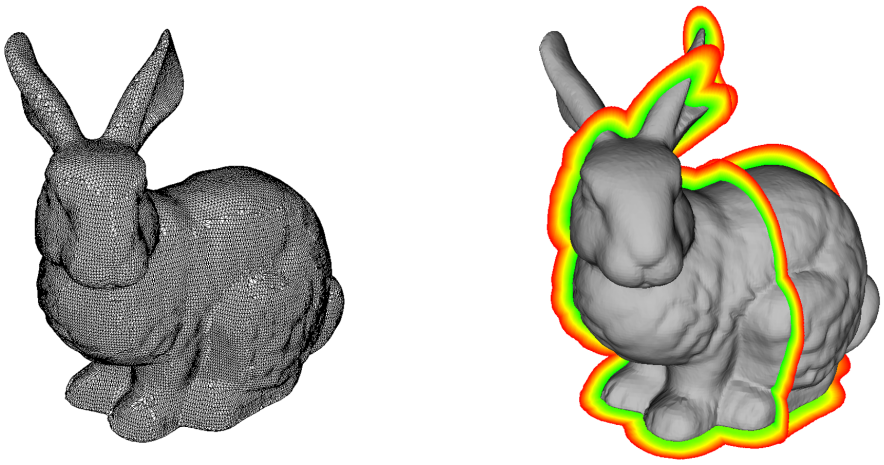
Cut cells describe piecewise linear approximations of boundaries intersecting Cartesian grids as shown in figure 1.3. The mesh generation is fully automated and does not need user validation. Though the geometry representation is a discretised approximation, high simulation resolutions lead to faithful representations of the input data. Individual cell geometry is augmented to store the additional information about its relation to the solid. Although the underlying data structure has equal cells, very small volume ratios may lead to limited time step size.

Structured regular meshes with embedded geometry representation are easily amenable to parallelisation. There is no need for transformation functions or user interaction at the generation phase. The computational cells are of identical size and shape, with the exception of extra information for boundary flux calculations. This additional data can be compressed to save memory space and the majority of the fluid calculations and variable updates are identical for all cells. Both ghost fluid and cut cell methods maintain local information about the surface and cells but as ghost fluid methods are not guaranteed to be conservative, we use cut cells. For static geometries, the mesh needs to be generated only at the start of a simulation and, as can be seen later, comparisons to experimental results demonstrate good accuracy. We use the split

cut cell method by Klein et al. [28] and the LPFS extension by Gokhale et al. [21] to calculate the behaviour at an object boundary. Detailed description of the geometry generation and flux calculations are given in Chapter 4.

1.3.1 Signed distance fields

The cut cell method requires a continuous signed distance field (SDF) to determine the intersection of a surface with the computational cells and produce its own representation of the boundary. While simple shapes can be described implicitly, complex surfaces require explicit representation. A popular format for describing geometry is stereolithography (STL) which finds use in fields from CFD to 3D printing. Surfaces are discretised into a set of triangular faces and have a well defined inside and outside (figure 1.4a). This information can be used to build an SDF around the input geometry. We therefore turn an explicit surface representation into an implicit one only to generate a final explicit structure for the fluid solver. There are several benefits to using an intermediate signed distance field generator. The SDF regulates the surface detail and guarantees that the surface information is correct and usable regardless of the computational cell size. Triangulated surfaces can intersect a cuboid cell in arbitrarily complex ways but the signed distance field algorithm transforms the input data into a set number of data points which can be used by the cut cell generator. Furthermore, implicit distance fields are useful in other methods and their inclusion in our software will provide longer term flexibility.



(a) An STL file represents a 3D surface as a collection of unordered triangles

(b) A narrow band signed distance field provides an implicit representation of a surface in a limited region around a geometry

Fig. 1.4 The CSC algorithm applied to the Stanford rabbit geometry [56] to produce a narrow band signed distance field for generating cut cells. (a) shows the collection of triangles which make up the input surface. (b) shows the surface plot (grey) of the signed distance field sampled at the cell vertices of a regular grid. We also show slices of the SDF values on the domain planes which show the distance values propagating in the surface normal direction from 0 at the surface (green) to the user specified limit (red).

Our generation of a signed distance field from STL geometry focuses on accuracy, performance and parallelisation and draws from research in computer graphics, computational geometry and parallel algorithms. For the cut cell approach, it is only necessary to know the distance to the surface within a small region around the geometry. We therefore focus on the generation of narrow band signed distance fields inside Cartesian grids based on the Characteristic/Scan Conversion (CSC) algorithm by Mauch [36] (figure 1.4b).

The algorithm constructs extrusions from surface features derived from the triangle elements of the STL format. The cells inside the extruded volumes are populated with the shortest distance to the appropriate feature and as the set of all extrusions fully covers the vicinity of the geometry, we end up with a complete signed distance field. The main benefits of the CSC algorithm are the limited extent of the extrusions, which reduces workload, and the parallelisability of the distance calculation. However, several surface configurations can arise in sufficiently complex geometries which require specific approaches to produce a correct SDF, making robust implementation non-trivial. In Chapter 2 we outline our implementation and adjustment of the CSC algorithm and introduce performance and robustness improvements to the original approach together with a discussion on the correctness of the underlying logic.

1.4 Parallelisation

Parallelisation of CFD simulations aims to solve subproblems in the computational domain simultaneously. Due to the arithmetic intensity and long runtimes, multi-threaded implementations are standard for most large-scale projects. Parallelisation often comprises aspects like workload distribution, data dependence, communication overhead, mesh connectivity and scaling.

An ideally parallelisable problem is a system of identical numerical problems which can be solved simultaneously with little to no communication between different components. From a programmatic point of view, fluid simulations describe information propagation in a discretised space for which inter-process communication is unavoidable but the regions of influence are often small. When working on a regular Cartesian grid, time marching hyperbolic problems are almost embarrassingly parallel. While there is data dependence between cells, it only extends to a limited neighbourhood and the majority of flux calculations and updates are independent. To divide a domain between several processes then only requires a small region of ghost cells around each patch and a way of communicating data after each time step.

Our SDF generator and fluid solver implementations are highly parallel with significant amount of work being done simultaneously at each phase of the pipeline. For the signed distance field generator we will tackle the question of which sections of the algorithm to parallelise and how to ensure atomic writes to global memory locations. For the fluid solver we need to ensure

correctness in an approach where parallel tasks depend on information from each other. We will also look at parallelisation across several processors and compute nodes with message passing and synchronisation points to ensure data consistency of subdivided simulations.

1.4.1 Graphics processing units

From multi-core CPU utilisation to distributed networks and supercomputers, conventional computing models have focused on a wide range of processor counts. Even with a structure amenable to distribution and parallelisation, CFD simulations on CPUs can have very long runtimes. The relatively recent emergence of general purpose massively parallel GPUs has lead to significant simulation speedup in many fields and the platform has become popular in scientific research due to the potential for considerable reductions in runtimes compared to conventional CPU implementations.

GPUs are designed to manipulate a high number of pixels with relatively simple arithmetic operations and matrix transforms. The resulting hardware is also suitable for numerically intensive problems in fields from linear algebra and CFD to data science and machine learning. As described in Chapter 3, our fluid simulation uses structures of discrete algebraic equations in a regular grid which is very close to the ideal GPU problem: independent identical calculations with multiple data. We use the CUDA programming platform by Nvidia [40] which offers an interface for general programming of selected GPUs.

1.4.2 CUDA programming framework

CUDA offers a C-like programming interface for general-purpose graphics processing unit (GPGPU) development. While traditional CPUs are most effective for serial or lightly parallel problems, GPUs are best suited for massively parallel execution where memory access is infrequent. The large number of cores and low-cost context switching offers the potential for high parallelism and reduced computation times. However, the nature of the hardware and software interface requires careful algorithm design. Approaches should seek to reduce memory access and use contiguous read and write patterns. GPUs are most effective for single instruction, multiple data (SIMD) problems, where identical independent calculations can be done using adjacent data.

1.4.3 Hardware

In CUDA terminology, the CPU system is called a *host* and the GPU a *device* (figure 1.5). Traditionally, device memory allocation and work launching is controlled from the host. The device

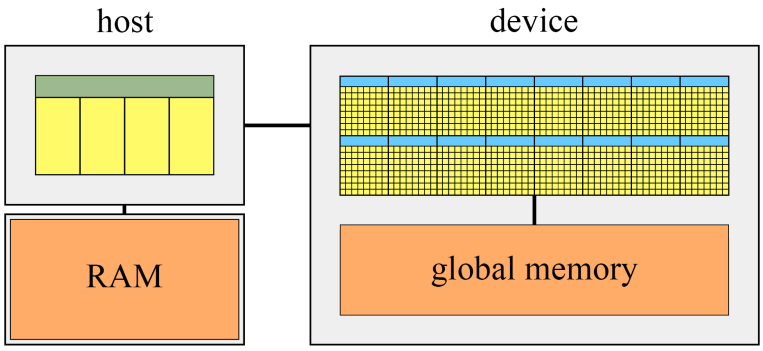


Fig. 1.5 The CPU is called a host and the GPU is called a device. The device has a high number of simpler cores and a bank of global memory. The host has a lower number of more sophisticated cores and access to the system memory. The host initiates computation and memory copies to the otherwise independent device.

has a large number of cores that are grouped into multiprocessors which has implications for the programming model. Memory exists in several levels on the device. The largest universally accessible bank is *global memory* where data can be copied to and from the host and read and written by every core. Lower down, each multiprocessor has access to *shared memory*. This smaller cache is addressable only by the cores on a multiprocessor, but it has faster access speeds than global memory. Shared memory can be used to explicitly cache data for better locality and improved runtime. The memory size is adjustable to distribute limited resources between user declared and automatic caches. Each core also has access to registers where intermediate results or locally declared variables can be stored. Registers are the fastest memory but are limited in size and overflow happens into slower memories, impacting performance.

Memory access is often the main bottleneck of CUDA applications. The bandwidth speeds between the host and the device are slow compared to the computation speed of the hardware. Ideally, transfer between the CPU and the GPU should be infrequent and in large data volumes. Global memory access is faster but still slow compared to the computation potential. With limited memory bandwidth, memory should be accessed in contiguous sections. When neighbouring cores request data in a coalesced manner, the number of memory transactions can be reduced. Shared memory patterns do not require coalesced access, but the data exists in banks and accessing different data from the same bank is serialised for older generation cards. While more recent hardware generations deal with the issue better, it is still an important aspect to consider in performance conscious applications.

There also exist other categories of memory on the device. Texture memory is often used for problems that display 2D data locality, constant memory can be used for globally accessible static

data, automatic L1 cache shares hardware with shared memory and automatic L2 cache is used for global memory access.

Memory allocation and deallocation on the device is done by calls to `cudaMalloc()` and `cudaFree()` from the host. The memory spaces of the host and device are disjoint and require explicit copying via `cudaMemcpy()`. Newer versions of CUDA have introduced unified addressing to present a single memory space to the programmer and hide explicit transfers between the host and device or multiple devices. Dynamic memory allocation for pointers in global memory is available on the device, but is often prohibitively slow. Memory allocation of variables and smaller arrays use registers or spill into slower memory. Shared memory can be allocated from the host during compile time or from the device during runtime. Transfer speeds between cards depend on the hardware configuration which is discussed in more detail in Chapter 5. In this thesis we use the Nvidia Tesla K20 [43] and Tesla P100 [44] GPU accelerators which are described in table 1.1.

	K20	P100
Cores	2496	3584
Peak DP performance	1.17 TFLOPS	4.7 TFLOPS
Global memory	5 GB	16 GB

Table 1.1 Hardware properties of Nvidia Tesla K20 and Tesla P100 GPUs.

1.4.4 Units of work

The programmer can launch *threads* – the combination of a single core with local registers for data – that are handled and scheduled by the GPU and execute algorithmically simultaneously but in practice partially sequentially in an unspecified order. The threads are grouped into *warps* of 32 which execute the same instruction simultaneously, making it advantageous to launch a multiple of 32 threads. In the case of conditional branching, both paths are taken sequentially which leads to loss of parallelism and some threads being idle while waiting for other branches to finish.

Warps are grouped into *blocks* of varying size. A single block occupies one multiprocessor, with one multiprocessor being capable of running many blocks. Each thread in a block can communicate through shared memory. While communication across blocks is theoretically possible, due to the arbitrary order of block execution, it cannot be relied on. An automated scheduler can switch out threads that are waiting for data from memory in order to execute threads that are ready to run. The scheduler can thus manage a massively parallel applications with limited hardware resources. The context switching is lightweight and memory fetches within a device and from the host can take place while the cores do numerical work. Best performance

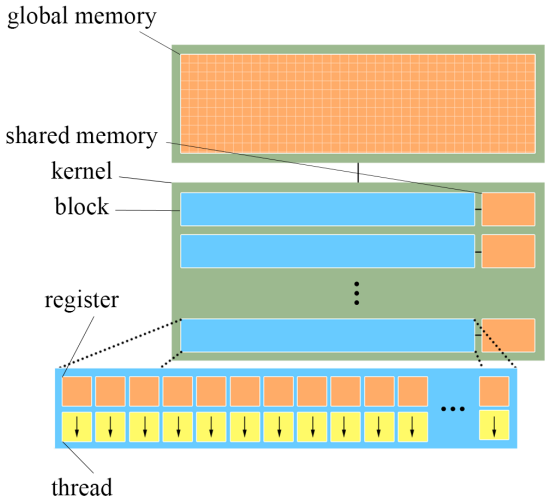


Fig. 1.6 Schematic of some CUDA concepts. A kernel is a unit of code that is executed by multiple threads grouped into blocks. Each thread has access to local registers and shared memory within a block. Global memory is accessible to all threads.

can then be achieved by oversubscribing the cores with independent calculations and letting the scheduler mask slower memory access.

Thread launch is done through *kernel* calls. These functions are initiated from the host to execute on the device. A call specifies the number of blocks and the number of threads per block: `myKernel<<<blockCount, threadCount>>>()`. A kernel function lists the instructions from the point of view of a single thread and the call parameters are either pointers to device memory or scalar values which are copied to the device at launch. In later device generations it is possible to launch kernels from the device using *dynamic parallelism* which can be useful for heterogeneous problems as discussed in Chapter 2. Kernel launches incur a small cost to allocate cores and copy parameters. The duration of a kernel depends on the number of threads, the amount of work each thread does and memory access patterns. Figure 1.6 shows a conceptual diagram of the different memory spaces and groupings from a programmer's point of view.

1.4.5 Multiple GPUs

The on-card memory varies with hardware but is generally less than that on a comparable cost multi-core CPU workstation. This places a limit on the resolution of a simulation that can be run on a single card. Multiple cards can offer more memory and additional parallelism if data is exchanged between GPUs efficiently. We make use of message passing to distribute and collect data during high-resolution simulations. A popular technology for inter-process communication

is the Message Passing Interface (MPI) standard [38]. The various implementations of the standard offer several useful operations from broadcasts to reductions. With the domain divided into patches between independent processors, simple CFD simulations can achieve very good scaling. Chapter 5 discusses the background literature on multi-card approaches and provides an analysis of the performance and scaling of our implementation.

1.5 Summary

The research and work documented in this thesis explores a parallelisable software pipeline for shock wave simulations featuring cut cell boundaries on graphics hardware using CUDA. We use cut cell mesh generation to achieve high parallelism due to the regular and uniform cell layout of the underlying fluid solver. Cut cells require a signed distance field to generate a piecewise linear boundary representation of the geometry. We use the Characteristic/Scan Conversion algorithm for triangulated surface input on GPUs. Due to the limited on-card memory of general purpose graphics cards, we investigate the utilisation and scaling of multi-card configurations.

The following chapters outline the nature of the numerical schemes and data structures and present an investigation of the performance and behaviour of the produced software. Chapter 2 discusses an augmented CSC algorithm for signed distance field generation on GPUs. Chapter 3 outlines the numerical models and schemes we use to solve the governing equations of fluid dynamics and describes the CUDA implementation. Chapter 4 discusses the cut cell approach to boundary representation, the numerical schemes proposed by Klein et al. and Gokhale et al. and the particulars of graphics card implementation. Chapter 5 discusses the use of multiple graphics cards and investigates the performance and scaling of high-resolution multi-card simulations.

Chapter 2

Signed Distance Field Generation

To represent objects within a simulation domain, we need to know where their boundaries lie. For fluid interaction with solids, we are interested in defining the intersection of the computational domain with surfaces. The signed distance function of a shape determines the closest distance from a point to that shape. This finds use in fields from computer graphics [11] to numerical modelling [20]. For complex geometries like cars and buildings, we use the union of the signed distance functions of geometric primitives that make up the shape to produce a signed distance field (SDF). These primitives are often triangles and the stereolithography (STL) file is one of the most popular formats in areas such as CFD [26] and 3D printing [65].

Narrow band SDFs are useful for quickly generating just the intersection between a computational mesh and an object where the distance information is not needed in the whole domain. For example, the signed distance field of a complex car body, as shown in figure 2.1, can be used to generate cut cell information. For complex shapes the SDF generation can take a long time and become a bottleneck when the subsequent numerical work is highly optimised and run on many-core architectures.

This chapter describes a fast and robust algorithm to generate level sets from triangulated surfaces on GPUs which can then be used to define cut cells and ultimately a mesh to use in CFD simulations. We provide an overview of the implementation and adjustment of the Characteristic/Scan Conversion (CSC) algorithm by Mauch [36]. We will outline improvements to the original approach and present an implementation on GPUs with a focus on how to manage information about many thousands of connected features. Using common and pathological test cases, we show the correctness of our implementation and the speeds achieved. This chapter is concerned with SDF generation in 3D. For two-dimensional simulations we follow the original 2D algorithm [36]. The work outlined in this chapter has been published as Roosing et al. [52]. Section 2.4 *Completeness of the CSC algorithm* features significant contributions by Dr Oliver Strickson.

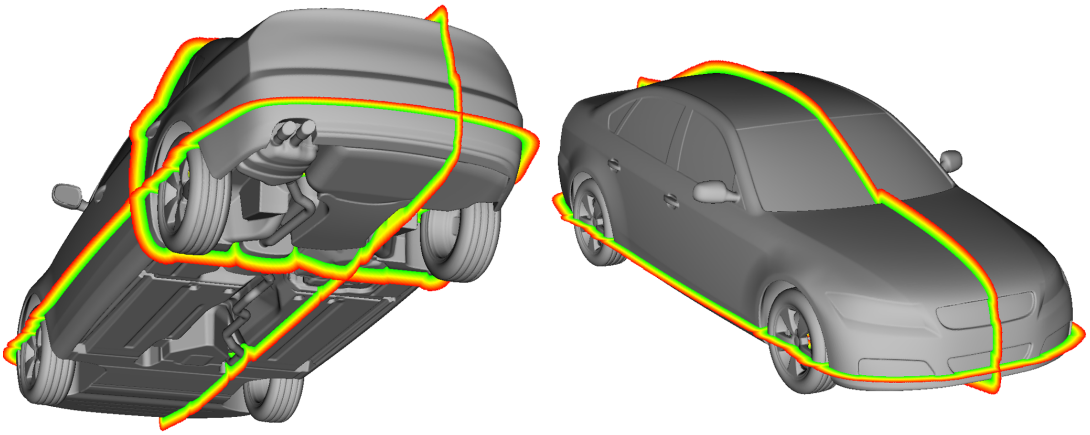


Fig. 2.1 The produced narrow band signed distance field and resulting surface plot of the DrivAer car model [59]. We show the surface plot (grey) inside a Cartesian grid and slices of the distance values increasing in the surface normal direction form 0 at the boundary (green) to a specified limit (red). Complex geometries can be processed quickly to generate embedded meshes in the initial phases of fluid simulations. Only creating a small shell around the underlying STL model is sufficient to describe a surface intersecting a Cartesian mesh. The high number of surface features are ordered and used to build extrusions which limit the space where distance calculations are made. Due to the low run times, high resolution computational domains can be used in conjunction with detailed models.

2.1 Background

Park et al. [48] have developed an algorithm for generating signed distances on the GPU for hierarchical grids. They sample mesh cells based on the complexity of the surface geometry and present good speedup compared to identical approaches on the CPU. Their use of angle-weighted pseudonormals at surface discontinuities is similar to the strategy we employ.

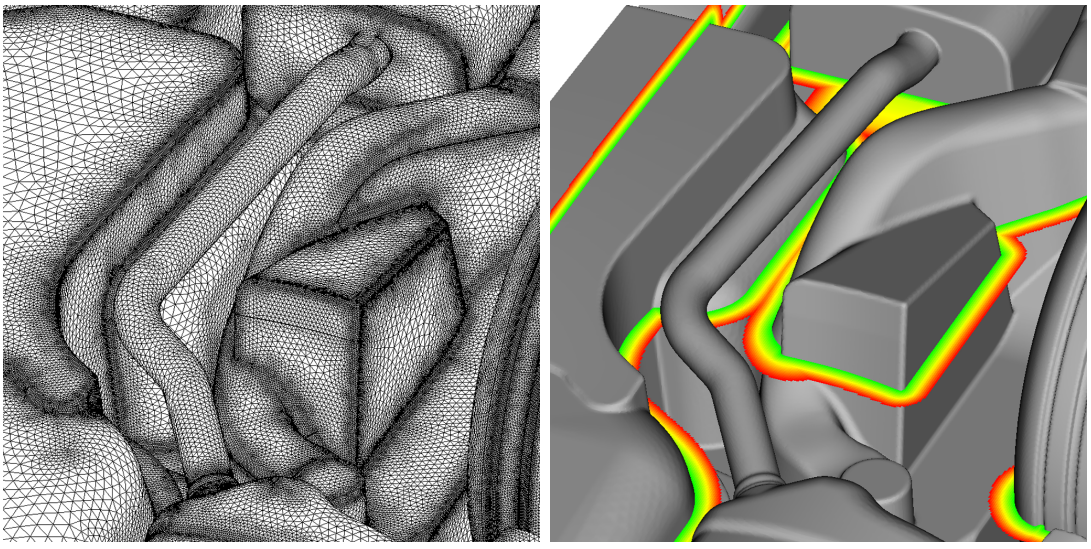
Sud et al. [57] describe a GPU signed distance field method based on Voronoi cells and slicing. Their speedup stems from the use of GPUs, culling far away features and clamping the rasterisation of the Voronoi cells. Though their approach is different from ours, the strategy of reducing calculations is similar to the current work. Their method doesn't store information about the connectivity of triangles but does use the CSC algorithm for suitable sub-problems and develops a new approach for problematic surface configurations. Our implementation is purely CSC based and fixes many of these geometric cases.

Sigg et al. [55] present a GPU implementation of the CSC algorithm for triangulated surfaces. Their work is focused on overcoming the need for vertex extrusions by combining edge and face extrusions. This is done in order to reduce the workload as well as avoid topological cases which

the CSC algorithm finds problematic. In this chapter we discuss a different methodology for the issues arising at vertices. An implementation of the CSC algorithm also exists by Mauch [35]. We use some of the insights of that code but have developed an independent strategy with updated feature generation, high parallelism and algorithmic improvements.

There is a lack of discussion in existing literature about how to best organise STL features for use with the CSC algorithm on GPUs. Specifically, it is not immediately clear how to efficiently produce extrusions from nearby surface triangles when no strict feature order is imposed in the input file. There are also gaps when it comes to discussing some complex cases which can arise in common geometries such as saddle vertices and ruff geometries discussed below. The original algorithm may encounter difficulties at these features when using high resolution grids. We will describe the efficient handling of STL features on GPUs, show robust extrusion building for unaddressed surface configurations and demonstrate fast narrow band SDF generation for a variety of complex test geometries. Based on personal communications, naïve CPU implementations can take several minutes to generate distance fields for test cases which our GPU implementation can handle in seconds.

2.2 Closest point distance transform



(a) Detail of DrivAer STL file (b) Produced surface plot and SDF slices

Fig. 2.2 Narrow band SDF results for the DrivAer geometry. (a) shows the high number of triangles that make up a detailed surface. (b) shows the surface plot (grey) and distance slices (gradient) from our implementation. The quickly generated field extends to a limited distance from the surface. As the SDF is free from gaps, the 0 crossing matches the STL input to within a fraction of Δx.

We outline the theory behind the CSC algorithm by Mauch [36]. The algorithm populates domain cells in the immediate vicinity of a geometry with the shortest distance to its surface. This is done by generating individual fields from triangulated surface features and combining them into a global signed distance field. Figure 2.2 shows the input and output of the algorithm. The initial data is a collection of triangles in 3D which describe a discontinuous surface (figure 2.2a). In a target Cartesian grid, the CSC algorithm populates the cells around the surface with the smallest distance to the object leading to an implicit description of the geometry (figure 2.2b).

The CSC algorithm can be used to generate the exact signed distance function of a surface within a regular grid. This function is defined at every point around the surface and grows in magnitude in the direction of the normal of the surface. For orientable surfaces, the positive and negative values divide a domain into an interior and an exterior, with the surface itself lying at 0. Let $\mathbf{x}$ be a point in the domain $\mathbb{R}^n$ and let $\partial\Omega$ be the surface. A signed distance function f is then defined as:

$$f(\mathbf{x}) = \min\{\mathrm{d}\{\mathbf{x}, \partial\Omega\}\},\ \forall \mathbf{x} \in \mathbb{R}^n, \tag{2.1}$$

where $\mathrm{d}\{\}$ gives the distance between a point and the surface. For smooth surfaces, $f(\mathbf{x})$ satisfies the Eikonal equation

$$|\nabla f| = 1. \tag{2.2}$$

In the case of discretised surfaces, however, there are discontinuities at the boundaries of the surface features. In this case, the signed distance field of the surface is the sum of the signed distance functions of all smooth regions of the surface.

The CSC algorithm uses the features of discrete surfaces to generate extrusions in their normal direction that are guaranteed to include at least the closest points to the original features. These extrusions are similar to Voronoi cells with the difference that they may include more than the closest points to a feature, they are artificially enlarged and may overlap. The sum of these extrusions will include all the closest points to the surface.

Let d_{ijk} be the minimum distance from the mesh cell c_{ijk} to the surface. When constructing extrusions E for all of the features of the surface, the CSC algorithm can be written as:

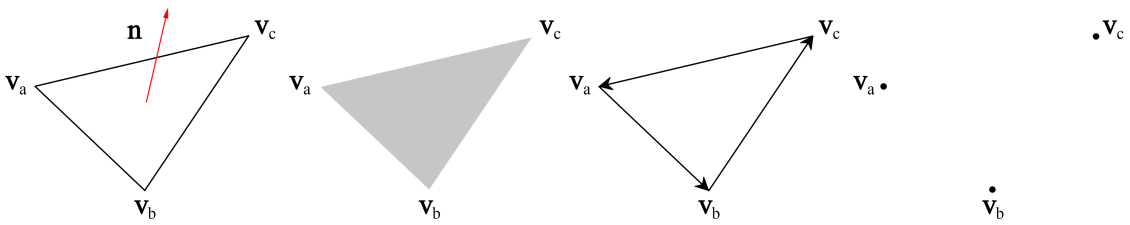
```
{ d_ijk = ∞ for all i, j, k }
for all e ∈ E do
    for all c_ijk ∈ e do
        d_new = distance to feature
        if |d_new| < |d_ijk| then
            d_ijk = d_new
        end if
    end for
end for
```

Calculating the minimum distances to the surface from the mesh cells within all of these areas produces a signed distance field. This operation is called a distance transform and results in an implicit description of the surface within the rectilinear mesh. As the computational work done is bounded by the number of surface features and the number of cells in the extrusions, the computational complexity of the algorithm is optimal: linear in both the feature count and the resolution of the mesh.

The CSC algorithm is limited to orientable closed surfaces or surfaces which can be considered closed in a limited domain. These geometries have a well-defined interior allowing for a signed distance field where the positive and negative distances are on either side of the surface, which lies at the 0 level set. The STL file format defines each triangle face using three counterclockwise vertices and a redundant surface normal:

```
facet normal ni nj nk
    outer loop
        vertex v1x v1y v1z
        vertex v2x v2y v2z
        vertex v3x v3y v3z
    endloop
endfacet
```

The distinct features of 3D surfaces are then the triangular faces, the triangle edges and the triangle vertices as shown in figure 2.3.



(a) STL feature (b) Face (c) Edges (d) Vertices

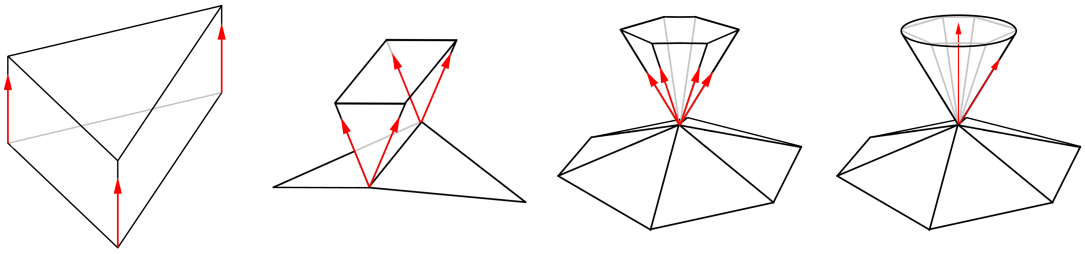
Fig. 2.3 STL surface features. The file elements are divided into three aspects and each feature is used to generate extrusions where the closest distance to the surface lies on the feature.

2.3 Extrusions

The extrusion polyhedra from the surface features encompass the area where the SDF is calculated. The CSC algorithm produces a narrow band SDF by extruding only a small user defined distance

away from the surface features which reduces distance calculation work. We list the different extrusion types, how they are generated and how our implementation diverges from the original description. We describe the categorisation of surface features and how this is used to reduce the amount of calculation that needs to be done, discussing previously unaddressed scenarios and proposed improvements.

The CSC algorithm describes the construction of extrusions containing at least the closest points to the discrete features. These extrusions are constructed based on the position, limits and normals of the underlying geometry. Extruding outward from a face produces a prism in the normal direction (figure 2.4a). An edge extrusion is a prism extruded from the line between two vertices in the directions of the two neighbouring faces (figure 2.4b) and a vertex extrusion is a pyramid defined by the normals of the adjacent faces that meet at the vertex (figure 2.4c).



(a) Face extrusion (b) Edge extrusion (c) Vertex extrusion (d) Vertex cone

Fig. 2.4 The extrusions from different surface features are generated in the face normal directions. The vertex extrusions can be simplified by assigning a cone which encompasses all of the face normals meeting at the vertex.

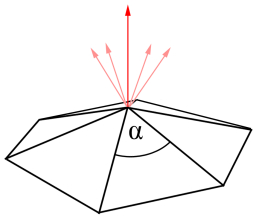


Fig. 2.5 The average pseudonormal, which is used as the axis of the cone, is constructed by weighting face normals by their respective angles α. The weighting deals with issues arising from many coplanar faces skewing the average normal.

While the two prism extrusions have a known number of faces, the vertex pyramid can be of arbitrary complexity which makes implementation and workload assessment difficult. We simplify the vertex extrusion by using a cone that encompasses all of the normals of the faces that meet at the vertex (figure 2.4d). The new vertex extrusion is constructed in the average direction of the normals weighted by the angle between the two edges of each triangle that

meet at the vertex in question. Taking the unweighted average can lead to incorrect extrusions. As described by Bærentzen and Aanæs [4], many coplanar faces sharing a vertex can shift the average disproportionally away from what would be the intuitive direction of the vertex. The result is an angle-weighted pseudonormal that correctly points in the average direction of the vertex (figure 2.5). This direction will be the axis of the cone and the adjacent normal most diverging from the average lies on the side of the cone. The obtained extrusion will then include all the points inside the original pyramid and has a simple description of points inside it. This fix fits well with the philosophy of the original algorithm where extrusions contain at least the closest points and only the global minimum magnitude value is recorded.

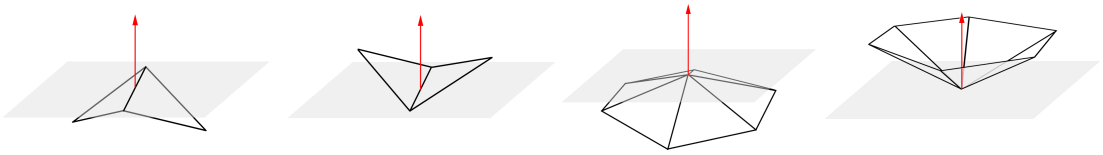
2.3.1 Surface curvature

The signed distance function describes both positive and negative values. The extrusions must then be constructed on both sides of the surface features. We adopt the convention that the interior of the surface is negative and the exterior is positive. The outward extrusions are then along the feature normal and the inward extrusions in the negative normal direction. For all the triangle faces, prism extrusions extending in both positive and negative directions will encompass the area closest to that face. Work can be reduced, however, for the edge and vertex cases based on the curvature of the local surface. Mauch introduces the concepts of convex and concave features.

Consider the plane described by the average of the surface normals at one vertex of an edge. That edge is convex if its neighbouring points lie below the plane, concave if they lie above and flat otherwise (figures 2.6a and 2.6b). The same is true for the coordinates of a vertex, which is convex when its neighbouring vertices all lie below the plane described by the vertex and the angle weighted average pseudonormal (figure 2.6c). A vertex is concave when its neighbours all lie above that plane (figure 2.6d), and flat when they all lie on the same plane. Convex features will only need positive extrusions, concave ones will only need negative ones and flat regions will need none as the surrounding extrusions from other features fill the area.

2.3.2 Saddle

In some cases, however, a vertex is not convex, concave or flat. A saddle point occurs when there are neighbouring vertices both above and below the plane described by the average pseudonormal of a vertex and the vertex itself as shown in figure 2.7a. Saddle points exist in common geometries and the original CSC algorithm does not deal with the gaps left between the other extrusions, leading to regions of undefined distances and an incorrect signed distance field. A fix for this problem, as suggested by Peikert and Sigg [49], is to use both a positive and a



(a) Convex edge (b) Concave edge (c) Convex vertex (d) Concave vertex

Fig. 2.6 The surface curvature is defined by the surface features and average normals. Convex edges and vertices have all of their neighbouring vertices below their average normal plane. Concave features have all of their neighbours above the plane.

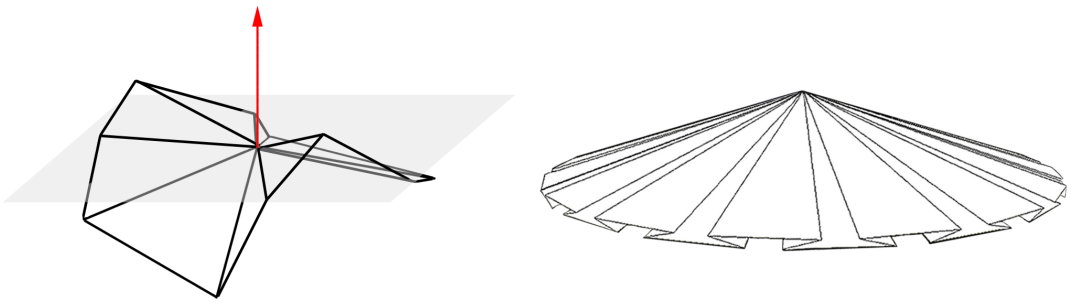
negative extrusion at these vertices. These special cases will then warrant double the workload of other curvatures but we observe that this strategy fully covers the volume around the vertex in all of our test cases and leads to a consistent signed distance field around complex discontinuities.

A question then arises about the shape of the extrusion from a saddle vertex. While smooth convex/concave regions create a convex gap that is limited by the normals of adjacent faces, in saddle point cases this volume can be complex and difficult to assign an extrusion to. By using a cone defined by the pseudonormal as the axis and the most diverging normal on its side, the relative order and configuration of other normals does not matter and we have a well formed vertex extrusion. For saddle shapes our approach is to first generate the cone for the positive side and then reflect it in the negative pseudonormal direction for the interior distance generation.

2.3.3 Ruff

Even fully convex/concave vertices can have normals which do not define a simple region to extrude into. Consider the case shown in figure 2.7b. The illustrated ruff-like shape is a valid orientable triangulated surface where faces with almost opposite pointing normals meet at a single vertex. As all the neighbouring points end up being on the same side of the pseudonormal plane, the vertex is classified as convex. However, the space enclosed by the sum of the face normals extends below the psudonormal plane at the vertex. Similarly, saddle points often, but not always, feature a collection of normals spanning more than the half-space.

A simple solution for these cases is to only consider normals pointing to the same side of the pseudonormal plane as the average normal itself. The volume bounded by these positive pointing normals will be strictly less than the half-space above the pseudonormal plane which is coverable with a cone. For our test cases this strategy fills the regions of ruff shapes and produces signed distance fields consistent with the input surfaces. It is possible, however, that a cone encompassing only positive pointing normals is not sufficient to cover the space between face and



(a) Saddle vertex (b) Ruff geometry

Fig. 2.7 High curvature geometry. Saddle points are vertices where there are neighbouring vertices on both sides of the pseudonormal plane. Ruff geometries can be convex, concave or saddle vertices which feature normals that point to more than the half-space around the pseudonormal plane. These lead to complex gaps between extrusions from other features or call for extrusions that cause sign ambiguity.

edge extrusions. In such cases a hemisphere can be constructed in the pseudonormal direction to cover the entire half-space above the vertex.

2.4 Completeness of the CSC algorithm

We show why the procedures outlined above cover the immediate vicinity of the surface without leaving any gaps and why sign conflicts can be resolved unambiguously. This is a novel contribution of this thesis as we demonstrate that by reasoning about the construction and limits of extrusions we can show that the original and augmented CSC algorithms are both correct.

Consider generating the SDF for all of $\mathbb{R}^3$, so that there are no holes. Space is divided by the surface into two regions, inside and outside, where moving from one region to the other along a continuous path necessitates crossing the surface. For any point outside, the closest point on the surface to this point could lie on either a face, an edge or a vertex.

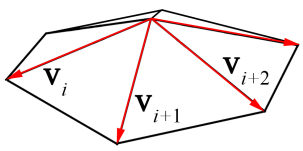


Fig. 2.8 Edge vectors at a vertex.

Consider the simplified case of a single vertex, with edge vectors extending to infinity. The vertex is at the origin, and the normalized edge vectors are labelled $\mathbf{v}_1, \ldots, \mathbf{v}_n$ (figure 2.8). We disallow

faces of zero area (adjacent edge vectors that are parallel or antiparallel). A face extrusion from the i^{th} face is then the set of points given by

$$\lambda \mathbf{v}_i + \mu \mathbf{v}_{i+1} + \nu(\mathbf{v}_i \times \mathbf{v}_{i+1}), \tag{2.3}$$

with $\lambda, \mu, \nu \geq 0$. The neighbouring edge extrusion is given by

$$\lambda \mathbf{v}_i + \mu(\mathbf{v}_{i-1} \times \mathbf{v}_i) + \nu(\mathbf{v}_i \times \mathbf{v}_{i+1}), \tag{2.4}$$

with $\lambda, \mu, \nu \geq 0$.

The vertex extrusion is the positive spanning set of the surrounding face normals, namely, the set of points given by

$$\sum_i \lambda_i \mathbf{n}_i, \tag{2.5}$$

where $\mathbf{n}_i = \mathbf{v}_i \times \mathbf{v}_{i+1} / |\mathbf{v}_i \times \mathbf{v}_{i+1}|$, $\mathbf{n}_N = \mathbf{v}_N \times \mathbf{v}_1 / |\mathbf{v}_N \times \mathbf{v}_1|$, and $\lambda_i \geq 0$ for each i.

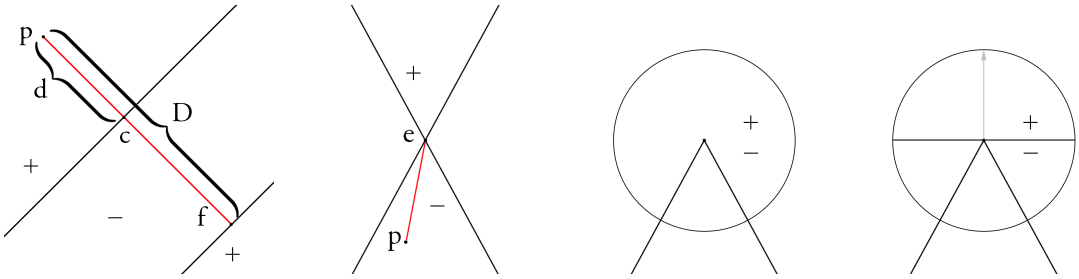
A positive spanning set of vectors in $\mathbb{R}^3$ is either:

1. an infinite, convex pyramid, whose edges are the convex hull of the vectors,
2. an infinite wedge, when two of the vectors are antiparallel,
3. a half-space of $\mathbb{R}^3$,
4. the entirety of $\mathbb{R}^3$.

A set of face normals can result in any one of these cases. The latter two cases can be obtained from ruff geometries.

For the SDF generation to be correct, these extrusions must fill the space to one side of the surface completely, since each extrusion is a superset of points where that edge, face or vertex is the closest point on the surface, and every point in space has at least one closest point to the surface, which must lie on *some* feature.

From the above, it is clear that the procedure will not lead to any gaps: the edge extrusions are defined by the sides of the face extrusions, without any space between them, and the vertex extrusions are defined by the span of normals of the faces meeting at a vertex, the convex hull of which will always include all of the normals. There are no other features of a triangulated surface and in the absence of gaps in a closed surface, every point in its vicinity must exist in an extrusion.



(a) Incorrect sign at face when an extrusion from face f extends to point **p** where the ambiguity is resolved by the smaller distance d from a nearby face which the incorrect extrusion must cross at point **c**

(b) Incorrect sign at point **p** from the edge e will be overridden by another extrusion where the edge e cannot be the closest feature as its two extrusions of different signs are disjoint

(c) Ambiguous sign at vertex caused by normals spanning all of $\mathbb{R}^3$ which will lead to the positive and negative extrusions to overlap

(d) Assigning a hemisphere in the pseudonormal direction in such cases is enough to generate two disjoint volumes where the closest points inside extrusions are dealt with correctly

Fig. 2.9 Sign ambiguity at features.

2.4.1 Conflicts between positive and negative extrusions

It is possible using the procedure described above for a given point of the domain (either inside or outside) to be both in a positive and a negative extrusion. Only one of these can be of the correct sign due to the orientability of the surface. Choosing the extruded distance field with minimum absolute value at the point in question is enough to resolve the majority of these conflicts. We describe the possible conflict cases below.

Incorrect distance information from face data

Consider a point strictly on a face (not an edge or a vertex). As the surface normals $\mathbf{n}_N$ point outside, there is some ϵ where $\epsilon\mathbf{n}_i$ is within the exterior of the surface (i.e. there is always free space immediately adjacent to a face in the normal direction). Thus, if a sign conflict arises from a face extrusion at a point **p**, then a face extrusion must have crossed the surface. Consider figure 2.9a: let the value of the incorrect extrusion from face f have absolute value D, and let the shortest distance from our point **p** to **c** where the extrusion crossed the surface be d, then $D > d + \epsilon$. This means that there is a closer point to **p** on the surface whose extrusion has the correct sign and the conflict does not cause any ambiguity.

Incorrect distance information from edge data

The situation is not as simple for edge vectors, since the face normals bounding an edge extrusion may point into the surface. However, note that if the edge extrusion (with incorrect sign) is given

by

$$\{\lambda \mathbf{v}_i + \mu(\mathbf{v}_{i-1} \times \mathbf{v}_i) + \nu(\mathbf{v}_i \times \mathbf{v}_{i+1}) \ : \ \lambda \geq 0, \ \mu, \nu > 0\} \tag{2.6}$$

then the edge extrusion of the correct sign is

$$\{\lambda \mathbf{v}_i - \mu(\mathbf{v}_{i-1} \times \mathbf{v}_i) - \nu(\mathbf{v}_i \times \mathbf{v}_{i+1}) \ : \ \lambda \geq 0, \ \mu, \nu > 0\}. \tag{2.7}$$

These sets are disjoint (figure 2.9b), meaning that if the incorrect edge extrusion conflicts with another extrusion of the opposite sign, the closest point on the surface cannot be on that edge at all, and there will exist an extrusion (from a face, vertex or another edge) with smaller absolute value. In other words, in cases where an edge extrusion would assign the wrong sign to a point, that edge cannot be the closest feature to that point, and in the absence of gaps, the point will also fall within some other extrusion with a smaller magnitude and the correct sign.

Incorrect distance information from vertex data

As with the edge data, it is possible for the face normals to point into the surface: that is, for $\epsilon \mathbf{n}_i$ to be in the interior of the surface for all $\epsilon > 0$.

The first three cases of vertex normal spans described above have the property that the corresponding extrusion of opposite sign is disjoint from the original one. Similar to edge extrusions, this means that in the case of conflicting information due to the propagation of an incorrect sign from the vertex, there is a closer point from another extrusion of the correct sign.

For the final case where the face normals span the domain, there is a genuine ambiguity between the positive and negative extrusions: they are both propagating information with the same absolute distance value, but with conflicting signs (figure 2.9c).

We solve the ambiguity by first computing an angle-weighted pseudonormal at the vertex. Bærentzen and Aanæs [4] show that this pseudonormal can be used as a discriminant for the surface at the vertex: if $\mathbf{p}$ is a point whose closest point on the surface is the vertex, then $\mathbf{N}_\alpha \cdot \mathbf{p} > 0$ when $\mathbf{p}$ is outside of the surface and $\mathbf{N}_\alpha \cdot \mathbf{p} < 0$ when $\mathbf{p}$ is inside the surface, where

$$\mathbf{N}_\alpha = \sum_i \alpha_i \mathbf{n}_i \tag{2.8}$$

is the pseudonormal, and α_i is the angle of the face with normal $\mathbf{n}_i$.

The extrusion is then performed only for the hemisphere oriented in the $\mathbf{N}_\alpha$ direction for the positive extrusion, and in the $-\mathbf{N}_\alpha$ direction for the negative extrusion (figure 2.9d). The positive

and negative extrusions are disjoint, apart from the plane normal to $\mathbf{N}_\alpha$. The closest point lying exactly on the plane can be excluded as if $\mathbf{p}$ is on the plane, then $\mathbf{N}_\alpha \cdot \mathbf{p} = 0$, and so by the discriminant property, $\mathbf{p}$ is on the surface and so is the vertex itself.

To show that this does not result in any gaps in the SDF, notice that a point outside of the positive hemisphere has $\mathbf{N}_\alpha \cdot \mathbf{p} < 0$, and so either $\mathbf{p}$ is closer to a point on the surface other than the vertex (and so must belong to another extrusion), or $\mathbf{p}$ is in the interior of the surface.

2.4.2 Cone extrusion of vertices

For cases where the normals at a vertex describe an infinite convex pyramid, we use a superset of the normals instead. The superset is formed by first constructing the angle-weighted pseudonormal $\mathbf{N}_\alpha$ as described above, finding the face normal $\mathbf{n}$ at the vertex which diverges most from it as $i_{\min} = \arg\min_i |\mathbf{n}_i \cdot \mathbf{N}_\alpha|$, where arg min gives the argument which minimises the result, and constructing a cone with this normal lying on its side.

The pseudonormal is within the original convex pyramid, since it is a positive combination of the normal vectors. Since $\mathbf{n}_{i_{\min}}$ minimized the right-hand side of this expression among the $\mathbf{n}_i$, the other $\mathbf{n}_i$ are contained within it, as is the original positive span, since

$$\mathbf{N}_\alpha \cdot \sum c_i \mathbf{n}_i = \sum c_i \mathbf{N}_\alpha \cdot \mathbf{n}_i > \sum c_i \mathbf{N}_\alpha \cdot \mathbf{n}_{i_{\min}} = \mathbf{N}_\alpha \cdot \mathbf{n}_{i_{\min}} \tag{2.9}$$

for any positive coefficients c_i with unit sum.

The above shows that the produced distance field will not have any gaps and that sign conflicts can be dealt with unambiguously. When limiting the algorithm to narrow bands, the same holds true as the extrusion distances are the same for all features.

2.5 Scan conversion

After generating the extrusions, we need to determine which domain cells lie inside them. This is a similar problem to scan conversion – a method in computer graphics that transforms mathematically described polygons into rasterised shapes. Mauch describes how we can determine which discretely spaced cells are inside continuous extrusions in 3D by reducing the scan conversion of a polyhedron to a series of 2D problems where slices of the extrusions are scan converted on planes of mesh cells.

However, the extrusions of the surface features are always either cones, hemispheres or convex prisms which can be rasterised in 3D. This is another novel contribution of our work. For the

prisms, we rasterise 3D regions of the mesh based on the half plane test. This strategy is used to find out if a point is within a convex polyhedron by determining if it is on the same side of all of the polyhedron faces. We find that this change fits better with the overall strategy of multithreaded computation and gives rise to other optimisation techniques described in the implementation section.
Let $\mathbf{c}_{xyz}$ be a cell in the domain D and let E be an extrusion with face normals N. Furthermore, let $\mathbf{p}_i$ be a point on the i^{th} face of an extrusion. The half plane test to determine if a point is within a convex polyhedron can be written as:

for all $\mathbf{c}_{xyz} \in D$ **do**
  for all $\mathbf{n}_i \in N$ **do**
    if $(\mathbf{n}_i \cdot \mathbf{c}_{xyz} - \mathbf{p}_i) < 0$ **then**
      return false
    end if
  end for
  return true
end for

In the implementation the scalar product is tested against some small value ϵ, to take into account numerical errors and the spacing of Cartesian grid cell centres.
A point is within a hemisphere if it is within a specified distance of the sphere centre and on the correct side of a plane. Points inside a cone satisfy the condition:

$$\frac{\mathbf{N}_\alpha \cdot (\mathbf{p} - \mathbf{v})}{|\mathbf{p} - \mathbf{v}|} > \mathbf{N}_\alpha \cdot \mathbf{n}_{\text{md}} \quad , \tag{2.10}$$

where $\mathbf{p}$ is the point, $\mathbf{v}$ is the vertex, $\mathbf{N}_\alpha$ is the unit cone axis and $\mathbf{n}_{\text{md}}$ is a unit vector on the side of the cone.

2.6 Data structures

The above algorithm was implemented in CUDA to quickly generate signed distance fields on GPUs. The implementation starts with reading in the structured STL file that lists the vertices of each triangle face in a counterclockwise direction and an outward pointing normal. This data is used to construct a single entry in a Face object, three entries in an Edge object and three entries in a Vertex object. These objects are collections of the spatial coordinates of the vertices and normals of the features. We list them as structures of arrays where all the x-coordinates are followed by all the y-coordinates and so on. We generate these objects on the CPU and copy them into the GPU global memory. While a Face object fully describes a triangle with a normal, Edge and Vertex objects need further processing to generate extrusions.

2.6.1 Edge data

An Edge object is created with two end-points of an edge and a normal of the triangle it was constructed from which is insufficient for an edge extrusion which needs two normals. Assuming a correct closed surface, there exists a single entry with identical end points but a different normal. We would like to find matching pairs of edge features in the fastest possible way without checking each pair of endpoints against all the others. As the order of triangles in an STL file can be arbitrary, we would like to order the entries in the Edge object such that the pairs are next to each other.

Sorting points in 3D has no one correct solution, more so for pairs of points. One approach is to generate Morton codes for all of the points. Working with 32 bit floating point values for all of the coordinates, we can generate 30 bit integer values called Morton codes for each 3D point. These values will retain their relative position when sorted. Specifically, the sorted Morton codes will produce a Z-curve ordering. For our purposes, the actual order does not matter, only that identical edge features are positioned consecutively.

An integer Morton code generated from the three floating point coordinate values of a point will designate the position of the point in a 1D array. We use a 30 bit integer value stored in a 32 bit `int` variable with the two highest bits set to 0. The 32 bit `float` coordinate values are first bit shifted to give 10 digits preceding the decimal point. We then expand the three values using bitwise operations as shown in figure 2.10 or more concisely in code:

```
x = (x | (x << 16)) & 0x030000FF;
x = (x | (x <<  8)) & 0x0300F00F;
x = (x | (x <<  4)) & 0x030C30C3;
x = (x | (x <<  2)) & 0x09249249;
```

The resulting three `int` values are used to build the Morton code by shifting the y and z values further and interleaving all three into a single variable as shown in figure 2.11.

The expansion and interleaving of three 10 bit values limits us to 1024^3 unique values. We specify the Morton domain to encompass a space that is defined by the smallest x, y and z Vertex values at one corner and the largest value Vertex at the opposite extreme.

An Edge entry with two end points can then be transformed into a unique 60 bit `long` value where two 30 bit `int` values are concatenated such that the larger value takes up the high bit positions. We launch a thread per Edge object and store the Morton codes in a `long` pointer on the GPU. Two edges with the same end points then have the same code and we can sort the codes to position identical edges next to each other.

We use the Thrust library [45] to sort a list of integer position indices based on the Morton code map using `thrust::sort_by_key`. We then reorder the Edge entries based on the indices using

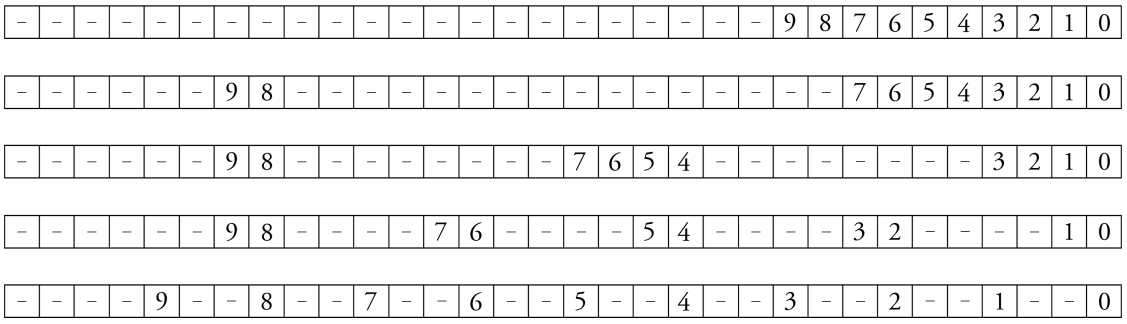


Fig. 2.10 Expanding a 10 bit variable in four steps. The figure illustrates the movement of the original bit position values in the variable (adapted from [50]).

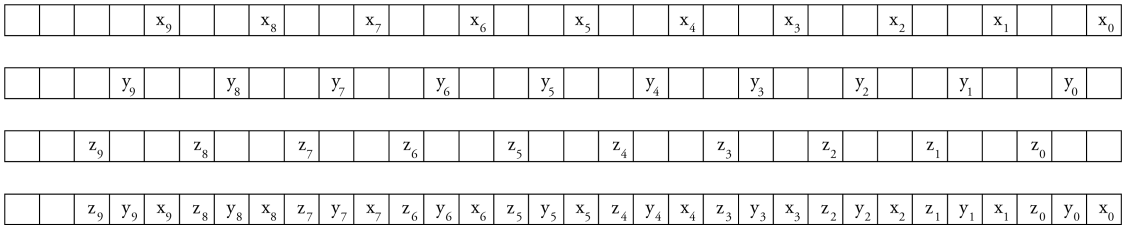


Fig. 2.11 Interleaving three bit-patterns into a single 32 bit `int` variable using OR operations (adapted from [50]).

`thrust::gather`. This produces groups of edge entries with identical Morton codes being consecutive. Finally, we launch a thread for each group and sequentially traverse the collection with identical codes, reordering them if necessary such that identical edges are consecutive. In practice, this still allows for high parallelism but in theory, there can be a noticeable difference in the amount of work each thread does when the underlying geometry has large variation in the size of triangles, leading to small non-identical edges having the same Morton code.

2.6.2 Vertex data

The entries of the Vertex object are similarly incomplete. Each entry has data about the position of the vertex and the normal of the face it was generated from. In addition we know the angle between the two edges connecting at the vertex on that face. We also retain data about the two other vertices on the original triangle. We again employ the Morton code strategy from the Edge object. We generate 30 bit `int` codes for each entry, sort a list of indices, reorder the Vertex objects and group identical entries together. This leads to an ordered list of entries where identical vertices are consecutive and each retains a unique normal and angle.

2.7 Extrusion generation

Once the data structures have been processed, we can generate the extrusions. There are three types of extrusions: prisms for the Face and Edge objects and cones or hemispheres for the Vertex objects. A prism is defined by six points and five sides. However, in order to tell if a point is within the area we are interested in, only four side normals and two points are needed (either two vertices on a face or the end points of a edge). A cone requires a point, an axis vector and the most diverging normal on its side. The hemisphere only requires a point and a clipping plane.

2.7.1 Face extrusion

The prism from a face is constructed by first extruding the three corner vertices by the user-specified distance in the normal direction. We then find side normals defined by the cross product of the counterclockwise ordering of the vertices when viewed from the inside of the prism. These three normals and points on the sides can be used to describe the planes of the prism sides. We use two of the original face vertices as the points on the planes. We then save the smallest and largest coordinate values of the original and extruded vertices. This produces a cubic axis aligned bounding volume (AABV) that contains the cell centres we wish to test for inclusion in the prism. The same is done for the negative extrusion of the original vertices in the flipped face normal direction. We end up with two Prism objects and their AABVs.

2.7.2 Edge extrusion

An edge extrusion is also a prism but extruded from a line. We start by determining if an edge is convex, concave or flat. Let the edge pair between vertices **a** and **b** be denoted by **ab** and **ba** with normals $\mathbf{n}_1$ and $\mathbf{n}_2$ respectively. We define a discriminant $d = (\mathbf{ab} \times \mathbf{n}_1) \cdot \mathbf{n}_2$. An edge is convex if $d > 0$, concave if $d < 0$ and flat if $d = 0$.

For convex edges we extrude the two end points of the edge by the specified distance in both the normal directions, thereby producing a prism. The four sides of the prism are described by the end points and the inward pointing normals constructed similarly to the face prisms. An AABV is also constructed to encompass the prism. For concave edges, the original normals are first flipped and the rest of the procedure is identical. For some geometries, it was discovered that a clipping plane at the base of the edge extrusion was needed to produce a smooth surface output. The plane is defined by the average normal and one of the end points of the edge.

2.7.3 Vertex extrusion

The regular vertex extrusion is a cone with a circle base. To construct it, we first scale the normals of the vertex entries by their angle. The collection of normals corresponding to a single vertex are then used to generate an average pseudonormal and find the largest angle between the average and the original normals.

All neighbouring vertices $\mathbf{v}_N$ are tested to see if they are above or below the plane defined by the original vertex $\mathbf{v}$ and the pseudonormal $\mathbf{N}_\alpha$. Consider the discriminant $d = \mathbf{v}\mathbf{v}_n \cdot \mathbf{N}_\alpha$. Vertex $\mathbf{v}$ is convex if $d > 0$, $\forall \mathbf{v}_N$, concave if $d < 0$, $\forall \mathbf{v}_N$, flat if $d = 0$, $\forall \mathbf{v}_N$ and a saddle point otherwise.

For convex, concave and saddle shapes we store the vertex coordinates, the pseudonormal and the most diverging positive pointing normal. Similarly to the case of prisms we define a bounding volume. The negative extrusion is constructed in the same way, but with a flipped average normal, where saddle points use a reflected positive extrusion. For ruff-like scenarios, we define an AABV of a hemisphere clipped at the pseudonormal plane.

When constructing the cone, the height is the user-defined maximum distance. For sharp corners, it may happen that the cone base is very large and if positioned diagonally in the domain, would require a large AABV which would extend far beyond the region closest to the vertex. To avoid testing unnecessarily many cells, we take the intersection of the AABV of the cone and the bounding volume of a sphere with the radius of the maximum distance centred at the vertex. This leads to a smaller AABV and fewer cells to calculate the SDF for.

2.8 Work scheduling

After all of the extrusions have been generated, we come to the problem of how to schedule the SDF generation on a GPU. For best performance, we would like to limit the number of calculations and memory transactions and do as much work as possible in parallel. The main variables in our software are the domain resolution, the desired maximum SDF distance and the number of surface features. Regardless of the extent of the computational domain, we only want to calculate the SDF for the sum of cells inside all of the extrusions, which often overlap. To limit which cells check for inclusion in which extrusions, the code works only on the cells inside the bounding volumes. Work is therefore only done on the cells most likely to be within any extrusion and we limit the tested cell and extrusion pairs. There are two obvious approaches to parallelism in this case.

The first is to check each cell in a bounding volume simultaneously. The start and end x-, y- and z-coordinates of the volume and the resolution of the domain are stored in the extrusion data. The number of cells the volume covers is then known and threads are launched according to the

size of the volume and the domain coordinates of each thread can be determined from the limits of the bounding volume. All threads check if they are within the bounding volume's extrusion in parallel. Using a `bool` flag, the code can keep track if a thread's coordinates are on the same side of all the extrusion faces. For threads that are inside, the distance to the feature can be calculated, and for threads with a smaller magnitude value than the previous one, we write the result to memory. This leads to warp divergence but as no action is taken for the other cases, there is no performance penalty. This implementation would launch a kernel per bounding volume where each thread works with the same data with the exception of their local coordinate data and the distance they calculate. For narrow band SDF generation of objects with uniform feature sizes, the bounding volumes are likely to be small and for high feature counts the kernel launch will dominate the runtime, leading to poor scaling.

The second approach is to parallelise over the surface features. A thread is launched per extrusion and it loops through each cell location within the bounding volume, determining whether to write a distance value to memory. For narrow bands and high feature counts, the serial traversal of bounding volume cells is relatively lightweight and fast. However, many of the extrusions overlap and the implementation must ensure that the smallest magnitude value is found. For parallel computation, the writing must be atomic, which will introduce some serialisation when multiple threads are working with the same domain coordinates. We use the `atomicCAS` method to try to write a `float` value into memory if the recorded value at the address has a larger magnitude. This attempt continues until the local value is successfully written to memory or a smaller magnitude value is written by another thread. The effect of the serialisation depends on the input geometry and the thread scheduling but the impact on the runtime is small compared to the overall amount of work.

Dynamic parallelism

Consider, however, geometries with few features in high resolution domains (e.g. when simulating flow over a box). When the number of cells inside extrusions is significantly higher than the feature count, looping over cells inside bounding volumes dominates the runtime. While the overall generation time is usually on the order of seconds, there is still scope for improved performance by using a hybrid of the two approaches outlined above. Dynamic parallelism allows for kernels to be launched from the device. Wang and Yalamanchili [64] provide an analysis of CUDA dynamic parallelism. They show that there is potential for speedup in several problems with inhomogeneous workload but that the greater overhead of launching kernels on the device can negate the benefits. Tang et al. [58] discuss a dynamic launch platform which seeks to launch device side kernels only when the potential computation time outweighs the launch overhead. They show good speed up for several benchmark problems. A hybrid approach would then launch a single kernel from the host, assigning a single thread for each bounding volume.

Launching kernels on the device has a greater overhead than host side launches but dynamic parallelism allows for more work to be done simultaneously. We therefore consider two alternatives: launching a thread per extrusion to loop through the cells or launching a thread per extrusion which will then itself dynamically launch a thread for each cell. The results section discusses the performance of both strategies.

2.9 Calculating the distance

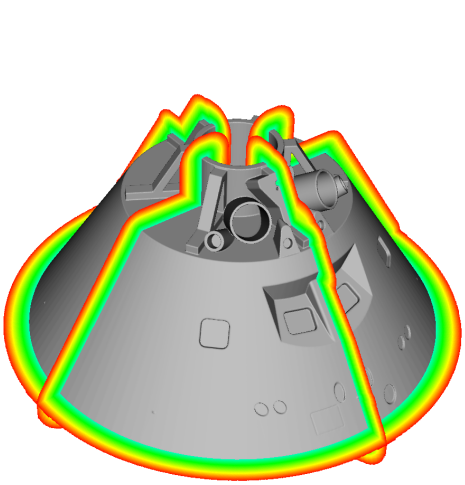
We allocate space in the GPU global memory for the 3D domain as a row major 1D `float` pointer. By storing the physical limits and width, height and depth information, we can find the x-, y- and z-coordinates of each cell from its offset in the pointer.

The SDF calculation kernel first checks if a cell centre is within the extrusion in question by performing a half plane test against the sides of the polyhedron for prisms, or a discriminant test for cones and hemispheres. To avoid machine epsilon errors and issues with testing discrete grid positions against continuum planes, we compare the results against small values from $10^{-4}\Delta$x to $10^{-3}\Delta$x where Δx is the length of a cell in one dimension. If the point is within the extrusion, we calculate the distance to the feature. If the absolute value is smaller than a user-defined maximum, and if the previous magnitude at that cell centre is larger, we write the result to global memory with the appropriate sign depending on the extrusion. For a face with normal $\mathbf{n}$ and a point $\mathbf{p}$ on its surface, the distance to point $\mathbf{c}$ can be found by $\mathbf{n} \cdot \mathbf{pc}$. For a vertex the value is the distance between two points in 3D. For edge extrusions it is the distance d from a point $\mathbf{p}$ to a 3D line $\mathbf{p}_1\mathbf{p}_2$:

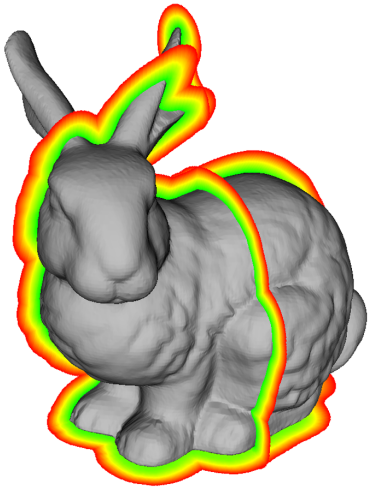
$$d = \frac{|(\mathbf{p} - \mathbf{p}_1) \times (\mathbf{p} - \mathbf{p}_2)|}{|\mathbf{p}_2 - \mathbf{p}_1|}. \tag{2.11}$$

2.10 Results

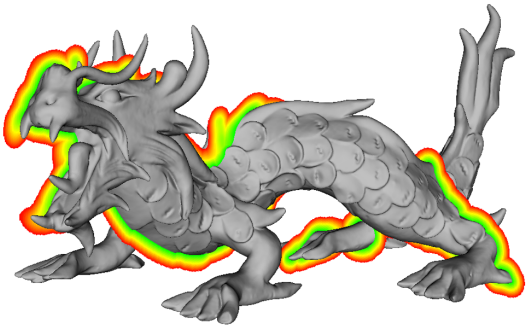
We present the results and timings for a number of test cases. The code was run on an Nvidia Tesla K20 card with common STL geometries. We show surface plots with pseudocolour and isosurfaces of the SDF and list the preprocessing and distance generation times. The produced code was validated against multiple common geometries which feature complex irregular surfaces as shown in figure 2.12. The geometries were picked to have a different number of triangles which range from $51,770$ for the Orion re-entry vehicle to $2,845,762$ for the DrivAer car body. We can therefore explore how the algorithm scales with increasing number of surface features. The intricate detail of all the models also helps show the robustness of the improved algorithm and demonstrate the high resolutions that our implementation can handle.



(a) Orion [18]



(b) Stanford Rabbit [56]



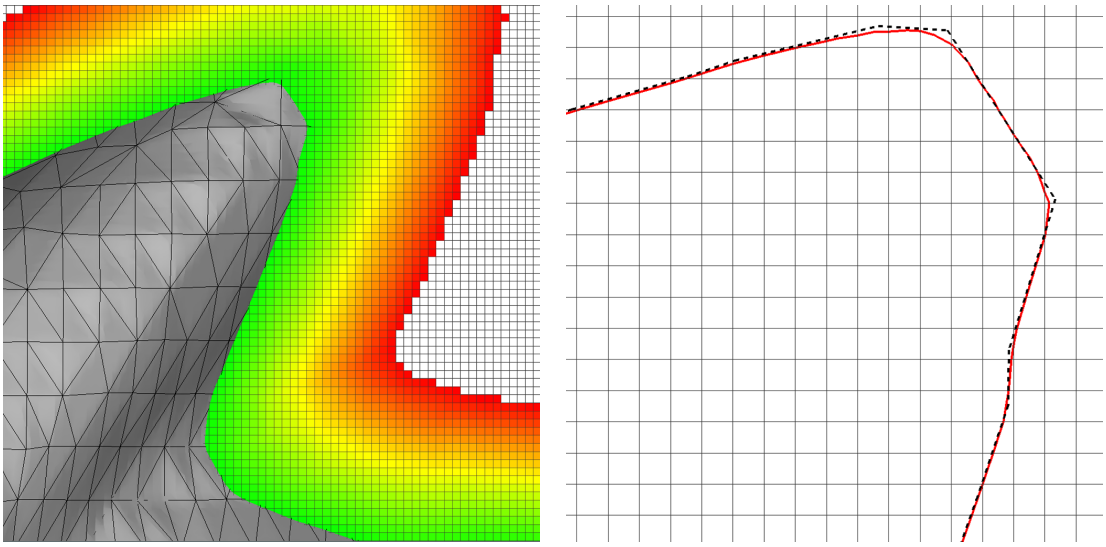
(c) XYZ RGB Dragon [56]

(d) Stanford Lucy [56]

Fig. 2.12 Surface plots (grey) and SDF slices (gradient) of test geometries. Narrow band signed distance fields were generated for complex shapes with varying feature counts on the GPU.

2.10.1 Accuracy

Figure 2.13 shows a zoomed-in region to illustrate the high resolution of the computational mesh, the SDF being set only in the immediate vicinity of the surface (2.13a) and how the produced surface matches the input mesh with an expected error of the order of the cell size (2.13b). (The visualisation software interpolates both the SDF values and the surface slices, making the image a close approximation, not an exact reproduction of the numerical result.)



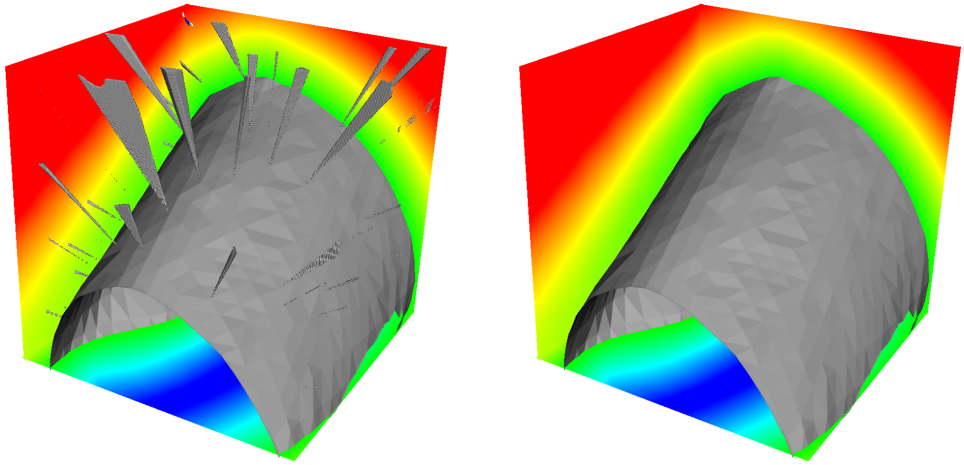
(a) Zero level set (grey) with STL edges (black)

(b) Zero level set (solid red) and STL slice (dashed black)

Fig. 2.13 Results of Stanford rabbit ear at $\Delta x = 0.125$, distance $= 2$. (a) shows how the produced zero level set (grey) matches the mesh lines of the STL triangles (black). With sufficient domain resolution the code reproduces the sharp discontinuities of the input geometry. (b) shows how the slices of the zero level set (solid red line) and the STL (dashed black line) match to within Δx. Note that the visualisation software interpolates values and that the numerical accuracy is often higher than the image.

Figure 2.14a illustrates the errors produced by only considering convex and concave vertices at the right ear of the Stanford rabbit geometry. When not addressing saddle points, gaps are left in the domain into which nearby extrusions may extend. As these values are never overwritten with the correct distance, artifacts may be produced. When a negative extrusion is not overwritten by a smaller magnitude positive extrusion on the outside of the surface, pyramid like protrusions are created in the level set. These errors may also appear as more distant spheres when the values near the surface are covered by neighbouring positive extrusions. Figure 2.14b shows the correctly produced SDF when generating extrusions on both sides of saddle vertices by building a cone around positive pointing normals and reflecting them to the negative pseudonormal direction.

While hemisphere generation at ruff geometries will produce the correct SDF, it emerged that it is sufficient to consider a cone extrusion restricted to positive pointing normals. Though the



(a) Errors when only considering convex and concave vertices

(b) Correct result when addressing saddle points

Fig. 2.14 Results of the Stanford rabbit ear at $\Delta x = 0.03$ show the issues with saddle points. When only assigning extrusions to convex and concave vertices, holes are left at saddle points. As no extrusion assigns a correct distance or sign, other extrusions can bleed into these regions. This can result in pyramid artifacts protruding from the surface where negative extrusions are never overwritten as seen in (a). At lower resolutions the errors can appear as artifacts away from the level set as the region closer to the surface gets the correct sign from nearby extrusions, but the ends of the interior extrusions are left uncorrected. (b) shows the correct SDF when assigning extrusions to saddle points.

correctness of this approach is not certain, in all of the test cases, a cone encompassing just the positive pointing normals produced no gaps. The volume of such an extrusion is less than that of a hemisphere and the workload is therefore smaller. Figure 2.15 shows the SDF for the ruff geometry of figure 2.7b. The hole left at the convex vertex is filled by generating a cone enclosing the positive pointing normals. Following several attempts, no surface could be found which would lead to an incorrect SDF, although it is possible that such a configuration can occur in common geometries. Figure 2.16 shows a pathological test case which features a normal pointing almost in the negative pseudonormal direction with the vertex being categorised as convex (2.16a). We show the hole left from other features (2.16b), the SDF in the cone around positive pointing normals (2.16c) and the correct distance field when applying the extrusion (2.16d).

2.10.2 Performance

Table 2.1 shows the feature counts and generation times of internal geometry data for various bodies. We list the minimum recorded durations of several runs per shape. This includes reading in a binary STL file, generating entries on the CPU, copying them to the GPU where vertices and edges are sorted and combined into unique features. The timing also includes the construction

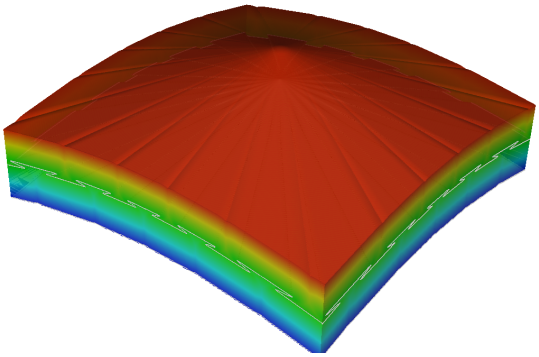


Fig. 2.15 A continuous SDF around a ruff geometry. A ruff vertex is classified as convex but the normals of the faces meeting at it span $\mathbb{R}^3$. By considering only the normals which point to the positive side of the pseudonormal plane, a strictly less than half-space can be filled with the distance to the vertex. The result is a continuous signed distance field around the surface.

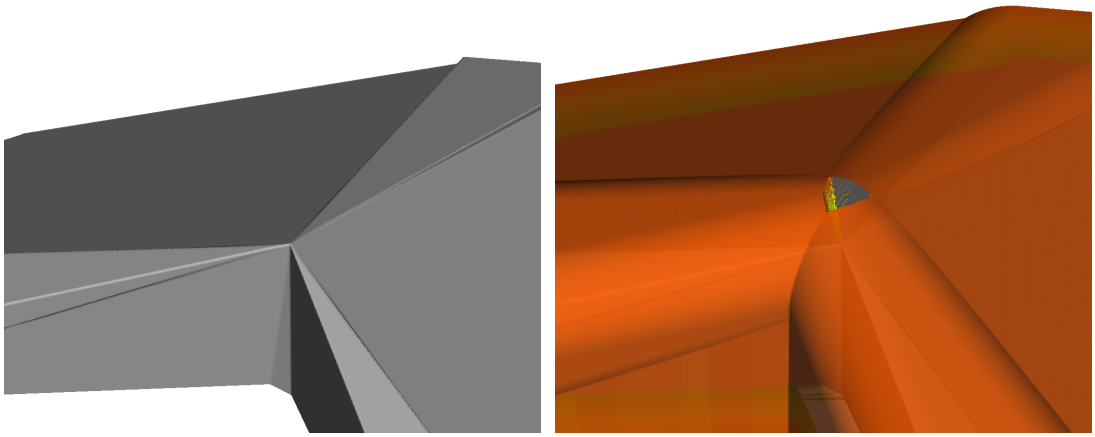
of the extrusion polyhedra on the GPU. We note a stable scaling which depends heavily on the feature count. The timing also depends on the uniformity of triangle sizes and the extent of the STL geometry which determine the uniqueness of Morton codes and how much serialisation occurs in feature construction.

Geometry	Faces	Time (s)
Orion	51,770	0.095
Stanford Rabbit	69,664	0.114
Stanford Dragon	100,000	0.154
XYZ RGB Dragon	721,788	0.951
Stanford Lucy	2,529,647	3.105
DrivAer	2,854,762	3.601

Table 2.1 Internal geometry generation times for different STL files on an Nvidia Tesla K20 card.

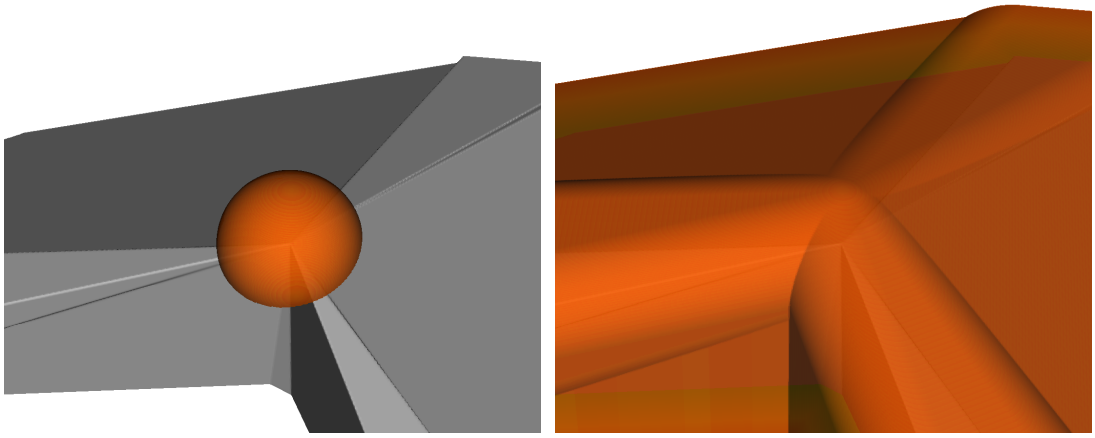
Table 2.2 shows the number of vertices listed in the input STL file, how many unique points they are combined into and what the proportion of saddle points is. Not all unaddressed saddle vertices lead to visible errors in the SDF as surrounding extrusions may combine into watertight surfaces and, depending on the resolution of the target mesh, the errors may not be noticeable. However, the resulting SDF will not be accurate for every resolution and the likelihood of disruptive errors increases with the number of saddle points. For the complex test surfaces, the number of saddle points was between 37.8% and 53.4% of the total number of vertices, making it necessary to have a robust strategy to deal with high curvature vertices.

Table 2.3 shows the time spent on generating the SDF for an Nvidia Tesla K20 card using dynamic parallelism. It lists the minimum recorded durations of multiple runs. This includes kernel launches and tests if cells are within extrusions and writing appropriate values to global memory.



(a) Pathological surface

(b) Hole left at vertex



(c) SDF in cone

(d) Correct SDF

Fig. 2.16 The pathological geometry case features a normal at a vertex which points in almost the opposite direction to the pseudonormal while the overall geometry is convex. Generating a cone of positive pointing normals fills the gap left between other extrusions.

Geometry	Total vertices	Unique	Saddle vertices	Proportion
Orion	155,310	25,876	9,795	37.8%
Stanford Rabbit	208,992	34,834	17,624	50.5%
Stanford Dragon	300,000	50,000	26,431	52.8%
XYZ RGB Dragon	2,165,364	360,894	192,882	53.4%
Stanford Lucy	7,589,232	1,264,847	620,974	49.1%
DrivAer	8,564,286	1,427,345	595,337	41.7%

Table 2.2 The STL file format lists each vertex multiple times, and our software combines them into unique points from which the appropriate extrusions are generated. For complex geometries, a large fraction of vertices are saddle points and need extrusions on both sides of the surface. The number of actual holes and errors in the SDF is different depending on the target domain resolution and the configuration of the surrounding surface.

Distance	Cell Size Δx	
	0.08	0.04
0.4	0.174	0.327
0.8	0.180	0.486

(a) Orion

Distance	Cell Size Δx	
	0.25	0.125
2	0.223	0.242
5	0.239	0.803

(b) Stanford Rabbit

Distance	Cell Size Δx	
	0.16	0.08
2	0.335	0.748
5	1.090	7.192

(c) Stanford Dragon

Distance	Cell Size Δx	
	0.53	0.26
5	2.214	2.266
10	2.258	5.277

(d) XYZ RGB Dragon

Distance	Cell Size Δx	
	4	2
20	7.688	7.700
40	7.721	7.741

(e) Stanford Lucy

Distance	Cell Size Δx	
	11.4e-3	5.7e-3
0.06	8.535	8.513
0.12	8.490	8.840

(f) DrivAer

Table 2.3 SDF generation times in seconds for test geometries on an Nvidia Tesla K20 with dynamic parallelism.

The results show low generation time for the simpler test cases but also poor scaling for higher feature counts. Table 2.4 shows the times when looping through bounding volume cells and not using dynamic parallelism. While the runtimes for simpler test cases are higher than for the parallel approach, as the feature count increases, the serial approach outperforms the alternative. This is due to both the higher launch cost of kernels on the device and a limited queue of active kernels and threads. The tipping point in performance is around 10^5 faces, past which the serial approach is consistently better. An optimal implementation would then find a balance between maintaining the maximum amount of active parallel calculation and making sure the hardware queue is not oversubscribed by doing serial traversal of bounding volumes that may otherwise wait too long for device side launch.

Both tables 2.3 and 2.4 show how the runtime depends on the cell size of the domain and the maximum distance of the SDF. These variables are the main measures of workload for single bounding volumes. As the number of cells in the volumes increases, more points need to be tested

Distance	Cell Size Δx	
	0.08	0.04
0.4	0.234	1.745
0.8	0.318	2.415

(a) Orion

Distance	Cell Size Δx	
	0.25	0.125
2	0.072	0.518
5	0.257	1.941

(b) Stanford Rabbit

Distance	Cell Size Δx	
	0.16	0.08
2	0.123	0.866
5	1.165	9.160

(c) Stanford Dragon

Distance	Cell Size Δx	
	0.53	0.26
5	0.143	0.702
10	0.591	3.972

(d) XYZ RGB Dragon

Distance	Cell Size Δx	
	4	2
20	0.071	0.316
40	1.395	1.548

(e) Stanford Lucy

Distance	Cell Size Δx	
	11.4e-3	5.7e-3
0.06	0.078	0.339
0.12	0.277	1.616

(f) DrivAer

Table 2.4 SDF generation times in seconds for test geometries on an Nvidia Tesla K20 without dynamic parallelism.

for inclusion in the extrusions which means more kernel launches or longer cell looping and potentially more conflicts in the atomic write to global memory. For the purposes of embedded mesh calculations, a distance of only a couple of cells is needed to produce an accurate surface description, and we have demonstrated low runtimes in these cases.

2.11 Limitations

While the current implementation introduces some improvements, there still remain limitations to the underlying algorithm. The CSC algorithm assumes a correct orientable surface, which means that there can be no flipped faces or gaps between faces. It is still possible to produce a correct SDF of a non-closed surface when clipping it to a smaller computational mesh where everything in the domain is on either one or the other side of the surface. The produced approach only creates a narrow band around the surface, leading to a secondary zero crossing between the negative limit of the SDF and the interior of the surface beyond the maximum distance. This can be fixed by sweeping along each of the coordinate axes and filling in unset values in the interior of the geometry. The geometry generation may be slow for large geometries with widely varying triangle sizes. In such domains, many smaller triangles can be assigned the same Morton code, leading to greater serialisation of the feature construction and higher generation times.

2.12 Conclusion

CFD simulations often feature relatively high resolutions and domains that extend far beyond the surface of embedded boundaries. There is a need for quickly generating the SDF of complex

geometries in limited regions of space, which still comprise a high number of small cells. The produced implementation allows for quick organisation and construction of internal geometry information, work scheduling and generating a signed distance field near the boundary of objects.

We have adjusted the original CSC algorithm to include angle weighted vertex normals and fixes for saddle points as described in literature. We have also presented a discussion on problems of the original algorithm at high curvature vertices and a fix for these cases. A discussion on the nature of the extrusions has shown that there are no areas left uncovered by their union and that sign conflicts do not lead to ambiguity. Though a hemisphere extrusion is the most certain way to ensure a correct SDF at high curvature vertices, in practice, a cone of positive pointing normals is often sufficient.

By using a set of common 3D geometry test cases, we have shown the robustness of the algorithm and demonstrated the performance of both the geometry preparation as well as the SDF generation for a range of feature counts and domain resolutions. Like the original implementation of the algorithm, the computational cost scales with the feature count of the triangulated surface and the number of cells within the bounding volumes.

We have presented a high performance generation of the necessary geometric data and the scheduling of work on GPUs. The resulting implementation constitutes the first element of the simulation pipeline: the geometry preprocessor. It offers a robust and fast way of generating 3D signed distance fields in high resolution Cartesian grids, from which we can generate cut cell data for arbitrarily complex geometries.

Chapter 3

Computational Fluid Dynamics

Computational fluid dynamics (CFD) is the field of modelling substance flow and interactions to analyse physical phenomena using numerical simulation. CFD is useful in many academic and industrial fields such as weather prediction, aircraft simulation and detonation studies. From mathematical models describing the information flow at an interface problem, complex systems can be built to explore various scenarios and regimes within fluid and gas dynamics. This section will describe the mathematical models and methods used to simulate fluids using a rectilinear Cartesian mesh. We will use a split approach where different dimensions are solved separately. This is well suited to GPU architecture with good memory access patterns. We use the MUSCL-Hancock reconstruction scheme which allows for shock waves and has second order accuracy. In this chapter we take a look at the existing literature on the use of graphics cards in numerical modelling, discuss our CUDA implementation and conclude with validation testing.

3.1 Background

Graphics cards have been utilised for general purpose computing since before dedicated GPGPUs and CUDA. Hagen et al. [23] present a 10-20 times speedup for CFD simulations using traditional graphics languages. The advent of specially designed hardware and simplified programming interfaces, however, led to a wider adoption of the massively parallel processors in several fields from molecular dynamics [2] to neural networks [62].

Cohen and Molemaker [16] present a fast double precision CUDA code for CFD using the Navier-Stokes equations. They discuss how, while the computational throughput of GPUs grows in accordance with Moore's law, bus speeds increase more slowly. Codes designed for graphics cards which minimise data transfer between the host and device would therefore outperform strategies

that only use GPUs for certain subproblems. Their dedicated GPU implementation demonstrates an eight times speedup compared to a traditional CPU code. Cohen and Molemaker identified a series of optimisations for any high performance GPU code. Among them are avoiding large data transfers between host and device, use of caching and organising data layout for optimal access. All of these suggestions were taken into consideration when designing our code.

Thibault and Senocak [60] present multi-card 3D Navier-Stokes simulations. They discuss how memory access patterns are vital to the good performance of any GPU implementation. The less sophisticated cores of a graphics card are not optimised for caching but arithmetic. It is the responsibility of the developer to manage data across the multiple memory levels. The design of a GPU implementation should then take into consideration memory coalescence, explicit caching in faster memory and fetch frequency. They also discuss how problems should be sufficiently large to properly mask data copying.

Brandvik and Pullan [10] present two- and three-dimensional Euler solvers on graphics hardware using CUDA. They report a 29 times speedup in 2D using the BrookGPU language and 16 times speedup in 3D using CUDA compared to a single core CPU implementation for high-speed flows around turbine blades. Similarly to the rest of the literature they emphasise data locality and present a 2D layout using both texture and shared memories. This further highlights the importance of reducing data fetches and using the relatively slow memory bandwidth optimally.

3.2 Governing equations

The standard governing equations for compressible inviscid flow are the Euler equations which can be written in the form:

$$\mathbf{U}_t + \mathbf{F}(\mathbf{U})_x + \mathbf{G}(\mathbf{U})_y + \mathbf{H}(\mathbf{U})_z = 0. \tag{3.1}$$

The solution vector $\mathbf{U}$ and flux terms $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{H}$ in the x-, y- and z-direction respectively can be written in vector form as:

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(E+p) \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vw \\ v(E+p) \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho w^2 + p \\ w(E+p) \end{bmatrix}, \tag{3.2}$$

where ρ is the density of the fluid, u is its velocity in the x-direction, v is the velocity in the y-direction, w is the velocity in the z-direction, E is energy per volume and p is pressure.

Energy per unit volume E can be expressed as:

$$E = \rho \left(\frac{1}{2} V^2 + e \right), \tag{3.3}$$

where

$$V^2 = (u^2 + v^2 + w^2). \tag{3.4}$$

The *specific internal energy* e is given by the ideal gas equation of state as:

$$e = \frac{p}{\rho \left(\gamma - 1 \right)}, \tag{3.5}$$

where γ is the *ratio of specific heats*. Throughout the thesis we use ideal gas with $\gamma = 1.4$.

These equations are suitable for our purposes for several reasons. Any perturbation only affects a limited subset of the domain at a later time, which means that we can divide the problem between several processors which are independent and need minimal communication, resulting in potentially good scaling. The hyperbolic equations allow for discontinuities and both shock and rarefaction waves.

3.3 Discretisation

We can solve the governing equations at certain locations and times by discretising them. Whereas the partial differential equations describe the behaviour of the system everywhere in continuous space, for numerical simulations we can break the representation of the mathematical model into discrete points, thereby replacing them with a system of algebraic equations.

3.3.1 Numerical discretisation

We divide the problem into discrete parts and map the mathematical model to a similarly subdivided space. This creates a piecewise representation such that as the cell sizes of the space grow smaller, the solution of the system approaches the solution to the continuous governing equations. We can then advance the system forward in time using a marching approach, thereby observing the evolution of the simulation or arriving at a steady state. There are two main approaches to the discretisation of equations for fluid simulations:

Finite-difference schemes describe the value of the continuous system at certain points in the domain. The approach is very popular in CFD literature and maps the differential form of the

equations to cell centres or cell vertices. Because it uses the differential form, it is not guaranteed to handle shock waves accurately.

Finite-volume schemes map the integral form directly into space by describing the cell average values of volumes, rather than points. As it is based on the integral form, the approach is conservative and suitable to handle discontinuities. The change of the values in a volume depends only on the fluxes into and out of the cell.

As we will focus on simulations featuring strong shock waves, we make use of finite volume schemes. The methods of equation discretisation are beyond the scope of this work. Detailed overviews can be found in Anderson [3]. Once we have the discretised form of the governing equations, we can use a selection of numerical techniques and solvers to advance the simulation in time.

3.3.2 Spatial discretisation

A three-dimensional simulation defines a space that extends in x-, y- and z-directions. In a Cartesian grid, the domain is divided into cells of size $\Delta x \times \Delta y \times \Delta z$. In dimension d, for a domain length L of M cells the cell size is given by:

$$\Delta d = \frac{L}{M}. \tag{3.6}$$

Figure 3.1 shows a 2D mesh.

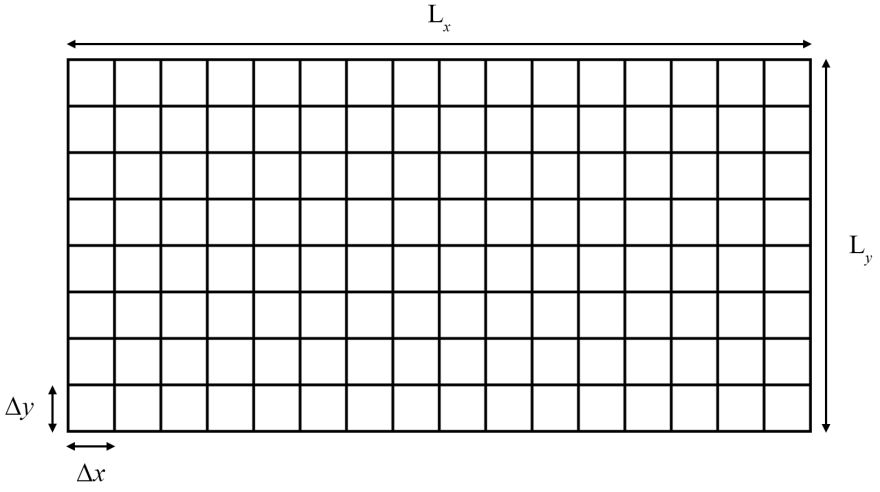


Fig. 3.1 An $L_x \times L_y$ 2D domain is divided into cells of size $\Delta x \times \Delta y$.

In the x-direction, a cell i spans the space $(x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}})$, where $x_{i-\frac{1}{2}}$ denotes the boundary of the cell x_i with x_{i-1}, and $x_{i+\frac{1}{2}}$ the boundary with x_{i+1}. The average value u of some variable at time t^n in cell i is given by:

$$u_i^n = \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(x, t^n)\, \mathrm{d}x. \tag{3.7}$$

Throughout the simulation, each cell will hold a local vector $\mathbf{U}$ that stores the physical values in the cell. At each time step, the data in this vector will be updated using flux vectors based on the interchange of information with neighbouring cells.

3.3.3 Temporal discretisation

Unlike spatial division, the time step is not uniform and needs to be chosen carefully to keep the system stable or else a shock wave may move more than a single cell in a time step. This inability to track discontinuities may lead to inaccurate results or unstable solutions. The formula to find a stable time step is given in Toro [61] as:

$$\Delta t = C_{\text{cfl}} \frac{\Delta x}{S_{\text{max}}}, \tag{3.8}$$

where C_{cfl} is the *Courant-Friedrichs-Lewy* coefficient that is given an empirical value appropriate to a problem. For many numerical schemes it has the range $0 \leq C_{\text{cfl}} \leq 1$. Unless otherwise specified, throughout the thesis we use $C_{\text{cfl}} = 0.9$. S_{max} is the speed of the fastest wave over all cells in the domain at the current time and is calculated as:

$$S_{\text{max}} = \max_i \left\{ |u| + a \right\}, \tag{3.9}$$

where u is velocity and a is the speed of sound in the medium considered, which for ideal gases can be calculated as:

$$a = \sqrt{\frac{\gamma p}{\rho}}. \tag{3.10}$$

3.4 Operator splitting

The above system can be solved in multidimensional domains by using operator splitting. To solve a 3D simulation we can split the overall update into individual sweeps where we solve in one dimension at a time. Let $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ be the solution sweeps in the x-, y- and z-directions respectively. The updated state $\mathbf{U}^{t+\Delta t}$ can be found by doing the dimensional sweeps one after the other:

$$\mathbf{U}^{t+\Delta t} = \mathcal{X}^{(\Delta t)}\mathcal{Y}^{(\Delta t)}\mathcal{Z}^{(\Delta t)}. \tag{3.11}$$

Note that each sweep uses the same Δt. This splitting is useful from a design point of view as well as allowing for optimisation of memory access which will be discussed below.

3.5 Update

The advance of cell (i, j, k) from a time t to $t + \Delta t$ takes the general form:

$$\mathbf{U}_i^{t+\Delta t} = \mathbf{U}_i^t + \frac{\Delta t}{\Delta x}\left[\mathbf{F}^t_{i-\frac{1}{2}} - \mathbf{F}^t_{i+\frac{1}{2}}\right] + \frac{\Delta t}{\Delta y}\left[\mathbf{G}^t_{j-\frac{1}{2}} - \mathbf{G}^t_{j+\frac{1}{2}}\right] + \frac{\Delta t}{\Delta z}\left[\mathbf{H}^t_{k-\frac{1}{2}} - \mathbf{H}^t_{k+\frac{1}{2}}\right], \tag{3.12}$$

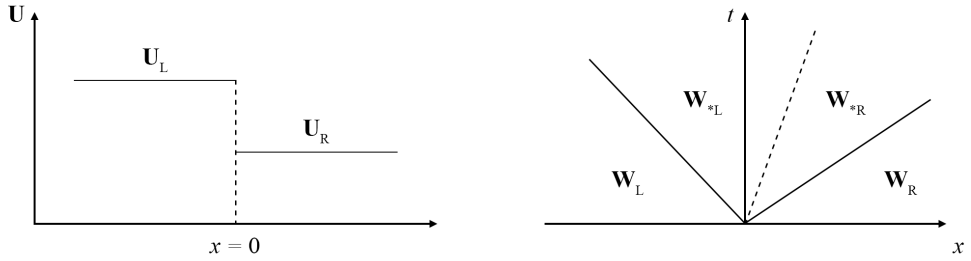
where $\mathbf{F}^t_{i\pm\frac{1}{2}}$, $\mathbf{G}^t_{j\pm\frac{1}{2}}$ and $\mathbf{H}^t_{k\pm\frac{1}{2}}$ are the fluxes through cell boundaries in the x-, y- and z-directions respectively which can be found by using a flux scheme.

3.6 Riemann problem for Euler equations

To find the intercell flux, we can solve the Riemann problem at the interface. Consider the one-dimensional Euler equations and a piecewise constant initial condition:

$$\left.\begin{aligned} \text{PDE:}\quad & \mathbf{U}_t + \mathbf{F}(\mathbf{U})_x = 0, \\ \text{IC:}\quad & \mathbf{U}(x,0) = \begin{cases} \mathbf{U}_\mathrm{L} & \text{if} \quad x \leq 0\,, \\ \mathbf{U}_\mathrm{R} & \text{if} \quad x > 0\,. \end{cases} \end{aligned}\right\} \tag{3.13}$$

The solution to the above initial value problem features three waves: a contact wave and two additional ones which may each be either a shock or a rarefaction wave (figure 3.2). The waves divide the solution space into four regions: $\mathbf{W}_\mathrm{L}$, $\mathbf{W}_{*\mathrm{L}}$, $\mathbf{W}_{*\mathrm{R}}$ and $\mathbf{W}_\mathrm{R}$, where we use the vector



(a) Initial conditions of the Riemann problem

(b) Structure of the solution to the Riemann problem in the x/t plane

Fig. 3.2 The Riemann problem features a left and right state on either side of a cell interface (a). The solution features three waves dividing the solution space into 4 regions $\mathbf{W}_L$, $\mathbf{W}_{*L}$, $\mathbf{W}_{*R}$ and $\mathbf{W}_R$ (b). We are interested in the variables in the star regions which determine the flux across the cell interface.

of primitive variables $\mathbf{W} = [\rho, u, p]^T$. The states $\mathbf{W}_L$ and $\mathbf{W}_R$, under the left and right waves are the initial conditions and the states $\mathbf{W}_{*L}$ and $\mathbf{W}_{*R}$ in the star region are unknown. While pressure and velocity are constant across the contact wave, density is different. By solving the Riemann problem, we are then looking to find the star variables p_*, u_*, ρ_{*L} and ρ_{*R} to determine the propagation of information at the interface and the substance flow across it. A more extensive overview of the Riemann problem can be found in Toro [61].

3.6.1 HLLC solver

While no exact closed form solution exists for the Riemann problem for Euler equations, iterative exact solvers can be used to find the solution numerically. These are often computationally complex and better performing *approximate solvers* have been devised which give satisfactory results.

We used the Harten-Lax-van Leer-Contact (HLLC) solver for the Riemann problem. This approximate solver is discussed in detail in Toro [61] and we provide a summary of the intercell flux calculation. Given the data $\mathbf{U}_L$ and $\mathbf{U}_R$ on either side of the cell interface, we start by estimating the pressure p_* from the primitive variable Riemann solver pressure p_{pvrs}:

$$\left.\begin{aligned} p_* = \max(0, p_{\text{pvrs}}), \quad p_{\text{pvrs}} = \tfrac{1}{2}(p_L + p_R) - \tfrac{1}{2}(p_R - p_L)\bar{\rho}\bar{a} \\ \bar{\rho} = \tfrac{1}{2}(\rho_L + \rho_R), \quad \bar{a} = \tfrac{1}{2}(a_L + a_R)\,. \end{aligned}\right\} \tag{3.14}$$

Next we calculate the estimated wave speeds according to Roe average eigenvalues which are given in Toro [61] as:

$$S_L = \tilde{u} - \tilde{a}, \quad S_R = \tilde{u} + \tilde{a}, \tag{3.15}$$

where $\tilde{u}$ is the particle speed and $\tilde{a}$ is the sound speed:

$$\tilde{u} = \frac{\sqrt{\rho_L}u_L + \sqrt{\rho_R}u_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}, \quad \tilde{a} = \left[(\gamma - 1)(\tilde{H} - \frac{1}{2}\tilde{u}^2)\right]^{1/2}. \tag{3.16}$$

$\tilde{H}$ is the approximated enthalpy given by:

$$\tilde{H} = \frac{\sqrt{\rho_L}H_L + \sqrt{\rho_R}H_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}, \tag{3.17}$$

and $H = (E + p)/\rho$. Based on the wave speed estimates, we find the intermediate speed S_*:

$$S_* = \frac{p_R - p_L + \rho_L u_L(S_L - u_L) - \rho_R u_R(S_R - u_R)}{\rho_L(S_L - u_L) - \rho_R(S_R - u_R)}. \tag{3.18}$$

Finally, we find the HLLC intercell flux based on the calculated wave speed estimates and the intermediate speed:

$$\mathbf{F}^{\text{hllc}}_{i+\frac{1}{2}} = \begin{cases} \mathbf{F}_L & \text{if} \quad 0 \leq S_L\,, \\ \mathbf{F}_{*L} & \text{if} \quad S_L \leq 0 \leq S_*\,, \\ \mathbf{F}_{*R} & \text{if} \quad S_* \leq 0 \leq S_R\,, \\ \mathbf{F}_R & \text{if} \quad S_R \leq 0\,, \end{cases} \tag{3.19}$$

where $\mathbf{F}_K = \mathbf{F}(\mathbf{U}_K)$ is the flux of either the left or right state. The star flux $\mathbf{F}_{*K}$ is defined as:

$$\mathbf{F}_{*K} = \mathbf{F}_K + S_K(\mathbf{U}_{*K} - \mathbf{U}_K). \tag{3.20}$$

The star state $\mathbf{U}_{*K}$ is given by:

$$\mathbf{U}_{*K} = \rho_K \left(\frac{S_K - u_K}{S_K - S_*}\right) \begin{bmatrix} 1 \\ S_* \\ v_K \\ w_* \\ \frac{E_K}{\rho_K}(S_K - u_K)\left[S_* + \frac{p_K}{\rho_K(S_K - u_K)}\right] \end{bmatrix}. \tag{3.21}$$

3.7 MUSCL-Hancock scheme

Before finding the fluxes across cell boundaries we reconstruct left and right states in each cell using the Monotonic Upstream–Centred Scheme for Conservation Laws (MUSCL) scheme [61]. First we construct piecewise linear values at each cell $\mathbf{U}_i$:

$$\mathbf{U}_i(x) = \mathbf{U}_i + \frac{(x - x_i)}{\Delta x}\Delta i, \quad x \in [0, \Delta x], \tag{3.22}$$

where Δi is a slope vector and the end points of the cell 0 and Δx correspond to the cell interfaces at $x_{i-\frac{1}{2}}$ and $x_{i+\frac{1}{2}}$. We employ a slope limiter that reduces the oscillations that may otherwise occur when applying a second-order method by extrapolating the values at the interface and thereby reducing the magnitude of big differences in neighbouring cells. The slope vector then becomes:

$$\overline{\Delta}_i = \xi_i \Delta_i, \tag{3.23}$$

where ξ_i is a slope limiter. After trying several limiters, we decided to use MINBEE (*mb*):

$$\xi_{mb}(r) = \begin{cases} 0, & r \leq 0, \\ r, & 0 \leq r \leq 1, \\ min\{1, \xi_R(r)\} & r \geq 1. \end{cases} \tag{3.24}$$

The boundary extrapolated values at the two interfaces of the cell in a single direction are:

$$\mathbf{U}_i^{\mathrm{L}}(x) = \mathbf{U}_i - \frac{1}{2}\overline{\Delta} i, \quad \mathbf{U}_i^{\mathrm{R}}(x) = \mathbf{U}_i + \frac{1}{2}\overline{\Delta} i. \tag{3.25}$$

Next we evolve these extrapolated values by half a time step:

$$\begin{aligned} \overline{\mathbf{U}}_i^{\mathrm{L}} &= \mathbf{U}_i^{\mathrm{L}} + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[\mathbf{F}(\mathbf{U}_i^{\mathrm{L}}) - \mathbf{F}(\mathbf{U}_i^{\mathrm{R}})\right] \\ \overline{\mathbf{U}}_i^{\mathrm{R}} &= \mathbf{U}_i^{\mathrm{R}} + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[\mathbf{F}(\mathbf{U}_i^{\mathrm{L}}) - \mathbf{F}(\mathbf{U}_i^{\mathrm{R}})\right] \end{aligned} \tag{3.26}$$

We can then calculate the solution to the Riemann problem at the interface where the two evolved boundary extrapolated values of neighbouring cells meet. The Riemann solver takes as input data $\mathbf{U}^{\mathrm{L}}$ and $\mathbf{U}^{\mathrm{R}}$ such that:

$$\mathbf{U}^{\mathrm{L}} \equiv \overline{\mathbf{U}}_i^{\mathrm{R}}; \quad \mathbf{U}^{\mathrm{R}} \equiv \overline{\mathbf{U}}_{i+1}^{\mathrm{L}}. \tag{3.27}$$

A cell interface then requires the information from the two cells on either side of it to do a reconstruction and solve the Riemann problem. Finding the flow through both interfaces of a cell in one dimension requires a stencil of five cells.

3.8 Boundary conditions

To solve the edge cells of the domain we surround the mesh with ghost cells (figure 3.3). Conventionally these are not solved themselves but provide data for the edge cells. Consider the edges of a domain using a five-cell stencil. To find the flows across the first and last interfaces we need two ghost cells on either end of the mesh. Multidimensional meshes are surrounded by layers of ghost cells in each direction.

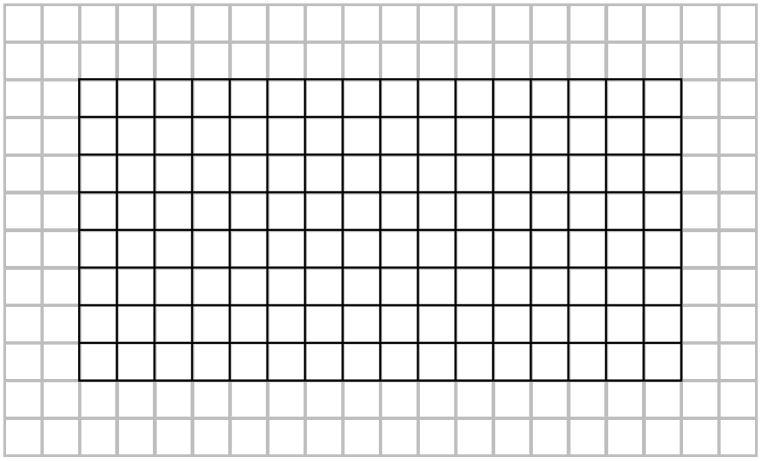


Fig. 3.3 To find the fluxes at domain limits, we surround the space with ghost cells. These hold data depending on specified boundary conditions. The width of the ghost cell layer depends on the stencil size.

We can impose boundary conditions at the edges of the domain by setting the data in the ghost cells in each direction every time step. We make use of two boundary conditions: transmissive and reflective. A transmissive boundary lets waves pass out of the domain. Let $\mathbf{U}$ be the state vector of a cell. For a domain of M cells we set the values in ghost cells $[-2,-1]$ and $[M, M+1]$ based on the states of the domain edge cells:

$$\left.\begin{aligned} \mathbf{U}_{-1} &= \mathbf{U}_0 \;; & \mathbf{U}_{-2} &= \mathbf{U}_1 \,, \\ \mathbf{U}_M &= \mathbf{U}_{M-1} \;; & \mathbf{U}_{M+1} &= \mathbf{U}_{M-2} \,. \end{aligned}\right\} \tag{3.28}$$

Transmissive boundary conditions are used to model open areas where no information passes into the domain. A reflective boundary condition stops information from passing out of the domain by maintaining ghost cells with the same flow variables as the edge cells but with a flipped sign for the perpendicular velocity component. This can be used to model solid walls and

the ground. Let $\tilde{\mathbf{U}}$ be the state vector $[\rho, \mathbf{v}^o, p]^{\mathbf{T}}$, where $\mathbf{v}^o$ are the velocity components excluding the perpendicular velocity u. A reflective boundary condition is then given by:

$$\left.\begin{array}{llll} \tilde{\mathbf{U}}_{-1} = \tilde{\mathbf{U}}_0 \ ; & \tilde{\mathbf{U}}_{-2} = \tilde{\mathbf{U}}_1 \ ; & u_{-1} = -u_0 \ ; & u_{-2} = -u_1 \,, \\ \tilde{\mathbf{U}}_M = \tilde{\mathbf{U}}_{M-1} \ ; & \tilde{\mathbf{U}}_{M+1} = \tilde{\mathbf{U}}_{M-2} \ ; & u_M = -u_{M-1} \ ; & u_{M+1} = -u_{M-2} \,. \end{array}\right\} \tag{3.29}$$

The above methods and numerical scheme were implemented for graphics cards using the CUDA programming platform. As discussed in Chapter 1, memory access patterns and work distribution are important to obtain good performance on GPUs. In the sections below, we provide an overview of our design and implementation of a three-dimensional split MUSCL-Hancock solver focusing on data layout and work scheduling.

3.9 Data structures

Given the importance of coalesced memory access when it comes to getting speedup from a GPGPU, we look at how to represent the simulation data in limited global memory and the strategies we use with a split solver in 2D and 3D domains.

The variables within each domain cell are stored in the GPUs global memory. Depending on the card generation, we have access to varying amounts of space to store the flow and flux variables. For the methods and numerical schemes outlined above, we need 64 bit `double` values for all physical and spatial information to deal with values of small magnitude that occur in cases of near-vacuum and small cut cells. For a 3D simulation, the main physical variables are the state vector $\mathbf{U}$ and the flux vector $\mathbf{F}$, each holding five variables:

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix}, \qquad \mathbf{F} = \begin{bmatrix} f_\rho \\ f_{\rho u} \\ f_{\rho v} \\ f_{\rho w} \\ f_E \end{bmatrix}. \tag{3.30}$$

3.9.1 One-dimensional simulations

For a split approach, we need to store at least these ten values for each computational cell. The addition of a tracer or reactive species concentration would add another variable to both vectors. To solve the Riemann problem at each cell interface and update flow variables, we need data

spanning the specified stencil surrounding the cell in question. For a single dimensional sweep this is cells ahead of and behind the interface on the considered axis.

When looking at the calculation of the Riemann solution and the intercell fluxes of the numerical scheme we see that we need to make use of all variables very close together in the algorithm. Keeping in mind the need for coalesced memory access, we follow the approach of Blakely [9]. The data for a single vector is laid out as a 1D pointer of size $N \times V$ where N is the number of cells in the domain and V is the number of variables the state vector holds. We store the data such that the same physical variables are located in a contiguous section of memory. For example, when storing primitive variables, elements 0 to $N-1$ are the density values of the N cells in the domain and elements N to $2 \times N - 1$ are the velocities etc.:

$$[\rho_0, \rho_1, \ldots, \rho_{N-1}, u_0, \ldots, u_{N-1}, p_0, \ldots, p_{N-1}].$$

This structure of arrays means that as neighbouring threads, representing individual cells, read in their variables, we minimise the number of memory accesses as the reading of contiguous addresses can be combined into fewer transactions. This reduction of traffic across the slowest memory leads to better performance and we can cache the data closer to the GPU cores for reuse. We denote the variable and flux pointers U and F respectively.

3.9.2 Multidimensional simulations

For 2D and 3D domains, we also use a 1D pointer and map the indices of the array to a coordinate system. We define a simulation domain as a multidimensional space with spatial limits and a resolution. The resolution of a simulation determines how many equally sized cells the domain has been divided into. Along each axis d of the domain we specify the start and end dimensions, denoting a length in physical units, and a count of cells N_d. A three-dimensional domain is then made up of $N_x \times N_y \times N_z$ cells and represents space $[\min_x, \max_x]$ in the x-direction, $[\min_y, \max_y]$ in the y-direction and $[\min_z, \max_z]$ in the z-direction. We call the total number of cells in the domain the length l of the domain. The underlying data array is then of size $N_x \times N_y \times N_z \times V$ or $l \times V$ (figure 3.4).

While logically multidimensional, we store the information corresponding to cells as a 1D pointer in memory such that the first variable of the first cell at $(0, 0, 0)$ maps to index 0 and the first variable of the last cell at $(N_x - 1, N_y - 1, N_z - 1)$ maps to index $l - 1$:

$$[\rho_0, \ldots, \rho_{N-1}, u_0, \ldots, u_{N-1}, v_0, \ldots, v_{N-1}, w_0, \ldots, w_{N-1}, p_0, \ldots, p_{N-1}].$$

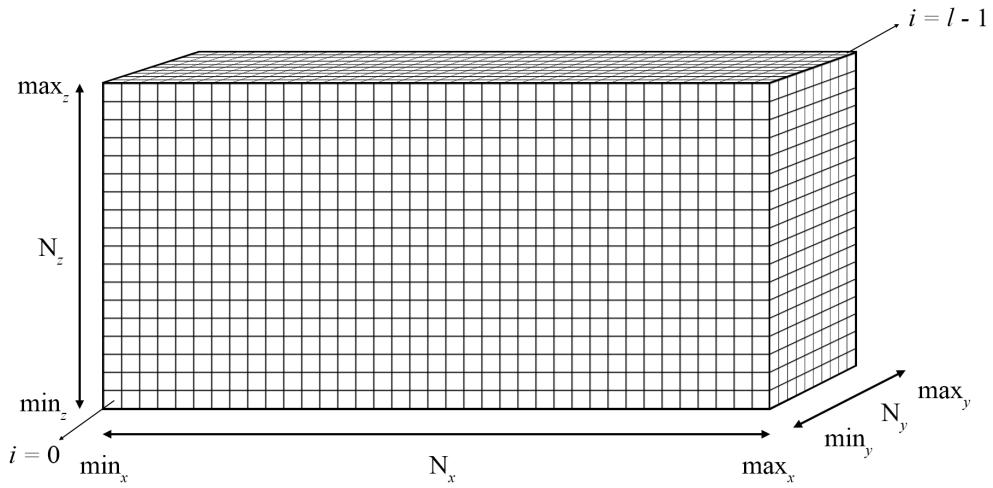


Fig. 3.4 A 3D domain spans the space $[\min_x, \max_x]$ in the x-direction, $[\min_y, \max_y]$ in the y-direction and $[\min_z, \max_z]$ in the z-direction. The mesh is divided into $N_x \times N_y \times N_z = l$ equal cells and is stored as a 1D pointer of size $l \times V$ elements where V is the number of physical variables. The cell at $(0, 0, 0)$ maps to index 0 and cell at $(N_x - 1, N_y - 1, N_z - 1)$ maps to index $l - 1$.

Any cell can then be referred to by its domain index (x, y, z) or its array index i. The conversion from the coordinates of the cell into its index i in the array is given by:

$$i = z \times N_x \times N_y + y \times N_x + x, \tag{3.31}$$

and the conversion from index i into the coordinate system is:

$$\begin{aligned} x &= i \bmod N_x, \\ y &= \lfloor i/N_x \rfloor \bmod N_y, \\ z &= \lfloor i/(N_x \times N_y) \rfloor, \end{aligned} \tag{3.32}$$

where mod is the remainder operator and $\lfloor \quad \rfloor$ is the floor function. Furthermore, the variable v of cell at index i is located at offset o of the memory pointer where

$$o = i + v \times l. \tag{3.33}$$

3.9.3 Transposition

While a 1D simulation maps a stencil across neighbouring cells to contiguous memory addresses, for multidimensional simulation, the concept of a cell's neighbours has different meaning depending on the direction we consider the domain in. When initially constructed, the data is laid out in layers such that the x-direction is the dominant (i.e. fastest changing) coordinate followed by y and z in order. For a split solver that switches between the three directions, only the first sweep of a time step would result in coalesced access. When looking at the domain along the y-axis, adjacent cells are separated by a stride of N_x in memory and for the z-sweep this stride is $N_x \times N_y$, leading to more memory transactions for a stencil and reduced performance.

We address this issue by transposing the data in global memory between the different sweeps. We rearrange the data so that the memory layout of the array always matches the cell neighbour configuration for a current sweep. This way the code can consider any Riemann problem passed to it as a 1D problem and the $2n$ neighbouring cells of cell at index i always span the range $[i-n, i+n]$ regardless of the sweep direction.

Consider the data array with its elements set to their indices:

$$[0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14,\ 15,\ 16,\ 17,\ 18,\ 19,\ 20,\ 21,\ 22,\ 23]\,.$$

In 2D the array maps to a matrix $\mathbf{A}$ from which we obtain the transpose $\mathbf{A}^{\mathrm{T}}$:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 12 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 & 22 & 23 \end{bmatrix}, \qquad \mathbf{A}^{\mathrm{T}} = \begin{bmatrix} 0 & 6 & 12 & 18 \\ 1 & 7 & 13 & 19 \\ 2 & 8 & 14 & 20 \\ 3 & 9 & 15 & 21 \\ 4 & 10 & 16 & 22 \\ 5 & 11 & 17 & 23 \end{bmatrix}.$$

The 1D pointer data is rearranged to

$$[0,\ 6,\ 12,\ 18,\ 1,\ 7,\ 13,\ 19,\ 2,\ 8,\ 14,\ 20,\ 3,\ 9,\ 15,\ 21,\ 4,\ 10,\ 16,\ 22,\ 5,\ 11,\ 17,\ 23]\,.$$

We thus achieve coalesced memory access for multidimensional simulations. The overall flow of the program will then start out with an original configuration, solve in the x-direction, transpose the data into y-dominant order and after a solve will transpose for the z-sweep. At the end of the time step we return the data to the original order.

2D transpose

The transposition of a 2D array in global memory is a solved problem with a good performance implementation presented by Ruetsch and Micikevicius [53]. This approach takes a source and target pointer in global memory and uses thread block level caching to achieve a matrix transpose with runtimes close to memory copy. We use an out-of-place transpose as 2D simulation domains are rarely square in proportion. The algorithm launches $\lceil \frac{N_x}{16} \rceil \times \lceil \frac{N_y}{16} \rceil$ thread blocks, where $\lceil \quad \rceil$is the ceiling function. The algorithm then allocates a $16 \times (16 + 1)$ block of shared memory, and each thread block transposes a small tile while achieving coalesced patterns in global memory and avoiding bank conflicts in shared memory. Bank conflicts occur when different threads try to access the same shared memory module. There are 16 modules and adding a $+1$ padding to the array avoids a 16 way conflict when a half-warp reads a tile column. For full details of the implementation, the reader is referred to Ruetsch and Micikevicius [53].

The matrix transpose is an involution operation meaning the transpose of the result gives the original configuration. For matrix $\mathbf{A}$ and a transpose T, it holds that:

$$\mathrm{T}\left(\mathrm{T}\left(\mathbf{A}\right)\right) = \mathbf{A}\,. \tag{3.34}$$

Using this information, we can call the same function after the y-sweep to reorder the domain data into its original form, ready for the next x-sweep. As only the state array U persists throughout a time step, we can use the equally sized flux array F to be the target pointer. The y-sweep then switches the names of the two vectors and we have a constant memory footprint with transposed data for different directions. Our data will then reside in U for the x-solve, be transposed into F for the y-solve and then be transposed into U for the next time step. The variables hold different data depending on the sweep:

	x-sweep	y-sweep	x-sweep
U	$\mathbf{U}$	$\mathbf{F}_y$	$\mathbf{U}$
F	$\mathbf{F}_x$	$\mathbf{U}$	$\mathbf{F}_x$

3D transpose

The solution for the transpose of a 3D array is not a trivial problem. Our aim is the same as for a two-dimensional simulation, where we would like the cells in a stencil to be located close together in memory to facilitate coalesced access and retain a dimension independent solver function that only varies the dominant velocity element. Jodra et al. [27] present a way of transposing 3D data efficiently on GPUs. They approach the problem similarly to the 2D matrix transpose above and show that there are six possible transposes of a three-dimensional block. The simplest of these

is the identity transpose or a simple copy that they use as the benchmark for their performance analysis.

Defining a transpose as a permutation of the order of axes (i, j, k), the copy transpose is (x, y, z) and the three involution transposes are (x, z, y), (z, y, x) and (y, x, z). These three can be seen as a rotation around one static axis. The two remaining non-trivial rotations are (y, z, x) and (z, x, y). As our aim is to transpose a 3D domain before each dimensional sweep, we require the series of transpositions to reposition back to the x-axis position after doing a z-solve in a looping manner. Starting with the order (x, y, z), we transpose to y-dominant order using the (y, x, z) function followed by (z, x, y) for the z-solve and finally (z, y, x) to reposition the domain to x-dominant order. The absolute order of the axes changes in the following manner: (x, y, z), (y, x, z), (z, y, x) and (x, y, z). The final transpose (z, y, x) back to the original order is the slowest but not by much as Jodra et al. [27] report an overall performance of 0.8 to 1.0 times the runtime of the identity transpose.

The first transposition (y, x, z) can be viewed as the transposition of N_z xy-planes. We launch $\lceil \frac{N_x}{16} \rceil \times \lceil \frac{N_y}{16} \rceil \times N_z$ blocks that solve 16×16 tiles as in the 2D approach above. The (z, x, y) transpose is more complex. Each tile reads in data in the xz-plane, transposes it and then writes into the appropriate position in the xy-plane. The (z, y, x) shuffle is similar to (y, x, z) where we transpose by planes. These, however, have contiguous columns which are separated by $N_x \times N_y$ strides. We therefore launch $\lceil \frac{N_x}{16} \rceil \times N_y \times \lceil \frac{N_z}{16} \rceil$ blocks that solve 16×16 tiles. To achieve coalesced global memory reads and writes, the y and z dimensions of the blocks are switched (while maintaining the correct domain mapping). For full details of the implementation, the reader is referred to Jodra et al. [27].

The out of place transposition in three dimensions also makes use of the arrays U and F. For the x-solve, data resides in U, which is then transposed into F for the y-solve and back into U for the z-solve. The final transposition will place the state vector data into F, and we simply change the pointers of U and F to restore the initial array configuration:

	x-sweep	y-sweep	z-sweep	x-sweep
U	$\mathbf{U}$	$\mathbf{F}_y$	$\mathbf{U}$	$\mathbf{F}_x \rightarrow \mathbf{U}$
F	$\mathbf{F}_x$	$\mathbf{U}$	$\mathbf{F}_z$	$\mathbf{U} \rightarrow \mathbf{F}_x$

3.9.4 Auxiliary data

To calculate a stable time step we need information about the maximum wave speed in the domain. This is the maximum of a 64 bit `double` value for each cell. As the flux data does not need to persist after a sweep update, we can use the memory space already allocated for the first N number of elements of F. Calculating the maximum value over all of the cell speeds can be done efficiently by using the Thrust library [45] and calling the `thrust::reduce` or `thrust::max_element` kernels at the start of each time step.

3.10 Allocation, solver and update

Based on the data structures outlined above, the data allocation, solver and the update scheme can be implemented to work with a one-dimensional problem solved in a framework of data transposes and coordinate transforms. This results in a simplified interface and a modular framework where introducing new solvers leads to good memory access patterns.

3.10.1 Initialisation

In the initialisation phase, memory is allocated on the GPU for all of the data structures needed throughout the simulation. This includes objects of the Mesh class which maintain device-side `double` pointers for the state and flux vectors. We use `cudaMalloc()` to allocate space in global memory and then call an initialisation kernel to set the initial conditions of the simulation on the GPU. While the main data pointer exists in GPU memory, information about the physical limits, resolution and other spatial data is stored on the host. The initialisation does not include any host-device copying as the host side code is mainly concerned with reading in user parameters, creating the appropriate data structure descriptions and calling kernels.

3.10.2 Thread and block data

A Riemann problem solution requires information on both sides of an interface. While stencil sizes vary with methods, all feature overlapping regions of dependence. For the second-order methods used in the current work, the stencils overlap by two cells with their neighbour on each side. The data dependence is only in reference to the state vector at the start of the time step such that all interface problems can be solved simultaneously with the same static initial data. Each thread finds the flux for a single cell interface and holds a local copy of the state vectors of the region – $\mathbf{U}^{\mathrm{L}}$, $\mathbf{U}^{\mathrm{C}}$, $\mathbf{U}^{\mathrm{R}}$ – so the undetermined order in which the threads execute does not affect the solution.

Having redundant data representation raises questions about where the data is stored and how much space it takes up. A good approach is found in Blakely [9] which uses shared memory caching to store data specific to a thread block. At the start of the flux calculations, the state vectors at the index of the central cells $\mathbf{U}^{\mathrm{C}}$ are copied into contiguous shared memory. Neighbouring threads in a block correspond to neighbouring cells in the domain and as warps can use a single memory transaction for aligned copies, this is an optimal approach to transport data closer to the GPU cores.

The left and right neighbour data is read from the adjacent addresses of the shared memory. Direct and safe communication between threads is only possible within a thread block. We need

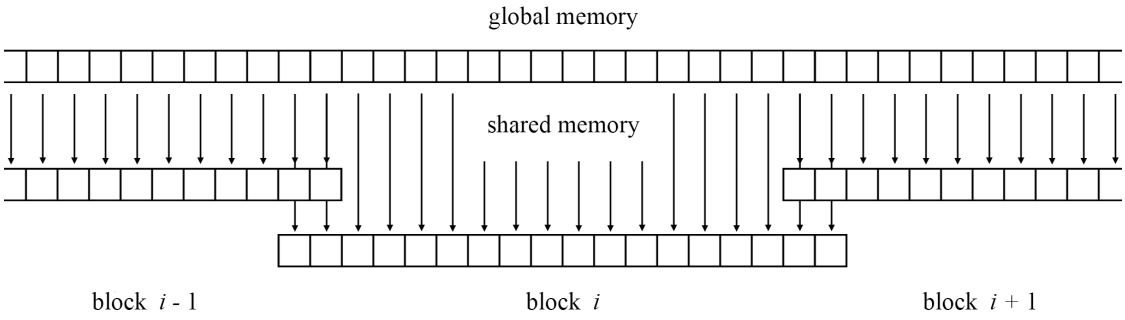


Fig. 3.5 Each thread copies from global memory to block specific shared memory to bring data closer to the cores. The blocks overlap with their neighbours to store all interface data locally. Transposing the data between sweeps ensures that threads access contiguous sections of memory.

to create ghost cells at each end of a thread block so that every cell has information about its neighbours. This approach means that the last two threads of block 0 will correspond to the same cells as the first two threads of block 1 and so on (figure 3.5). To find which cell a thread maps to we need to express its global index i in the data array in terms of its position in the block and grid:

$$i = D_\mathrm{b} I_\mathrm{b} - I_\mathrm{b} G_\mathrm{b} + I_\mathrm{t}, \tag{3.35}$$

where D_b is the block size `BlockDim.x`, I_b is the block index `blockIdx.x` in the grid, I_t is the thread index `threadIdx.x` within the block and G_b is the number of ghost cells per block. For MUSCL-Hancock this is 2.

The local copies made of the cached and global data are stored in limited size registers. It is therefore important to try to reduce the memory space needed to avoid register spills where new variables forces out old ones and each thread spends time writing and reading data from slower memory. For the MUSCL-Hancock scheme, we need data about the reconstructed states in cells to either side of the interface. The reconstruction itself requires information two cells to one side. Following Blakely [9], we assign a thread to each cell interface $i + \frac{1}{2}$ and maintain 3 local state vectors holding state data about cells $[i-1, i+1]$. After populating the state vectors with data from either global or shared memory, each thread will do reconstructions, slope limiting and half time step advancements based on the MUSCL-Hancock scheme. This will result in thread i holding data $\overline{\mathbf{U}}_i^L$, $\mathbf{U}_i$ and $\overline{\mathbf{U}}_i^R$. Each thread will then calculate the Riemann flux across its cell interface with data $\left(\overline{\mathbf{U}}_i^R, \overline{\mathbf{U}}_{i+1}^L\right)$. The HLLC solver requires local state vectors for the left and right states, a space to hold the solution and an additional vector to hold temporary data during the calculation. After the solve, each thread writes its solution into the flux vector F. Once all results have been found, a kernel is launched that applies the update to the state vector U in global memory.

3.11 Validation

The implementation was validated against several numerical and experimental results to demonstrate that the methods used are numerically robust and appropriate for the investigation of shock wave fluid dynamics. We also demonstrate the correctness of the dimensional splitting approach and the accuracy of the software using comparison with experimental and numerical results. The one-dimensional solver was tested against five shock tube problems. The two-dimensional code was validated against equivalent 2D membrane tests and shock bubble interaction. The three-dimensional code was validated against a bubble collapse experiment.

3.11.1 One-dimensional solver

The one-dimensional code was tested with the five shock tube tests given in Toro [61]. These include different initial values at either side of a diaphragm within a 1D domain. At the start of the simulation, the diaphragm is removed and the profiles of different variables at certain times can be compared against exact solutions. The initial configurations are shown in table 3.1. We use a resolution of 200 cells in a normalised domain in the range $[0, 1]$. These simulations are a good way of testing the robustness of the numerical methods and the correctness of our implementation as they feature several different wave configurations and have analytical exact solutions to compare against. Test 1 is a modified Sod shock tube problem which features a shock wave, a contact wave and a rarefaction wave to test the entropy conservation of the numerical method. Test 2 features two rarefaction waves and a contact wave with low pressure which demonstrates the robustness of the method in low-density scenarios. Tests 3 and 4 feature strong shocks that also show the robustness of the solver. Test 5 features a stationary contact wave to further demonstrate the correct functionality of the numerical methods. Figures 3.6 – 3.10 show the simulation results in red matched against the exact solution in black. We observe a good fit between our implementation and the exact analytical solution. The code captures various features and tracks sharp discontinuities as well as rarefaction waves.

Test	x_0	Time	ρ_L	u_L	p_L	ρ_R	u_R	p_R
1	0.3	0.20 s	1.0	0.75	1.0	0.125	0.0	0.1
2	0.5	0.15 s	1.0	−2.0	0.4	1.0	2.0	0.4
3	0.5	0.012 s	1.0	0.0	1000.0	1.0	0.0	0.01
4	0.4	0.035 s	5.99924	19.5975	460.894	5.99242	−6.19633	46.0950
5	0.8	0.012 s	1.0	−19.5975	1000.0	1.0	−19.5975	0.01

Table 3.1 Initial values on either side of a diaphragm x_0 in a 1D domain in the range [0,1] for the shock tube tests from Toro [61].

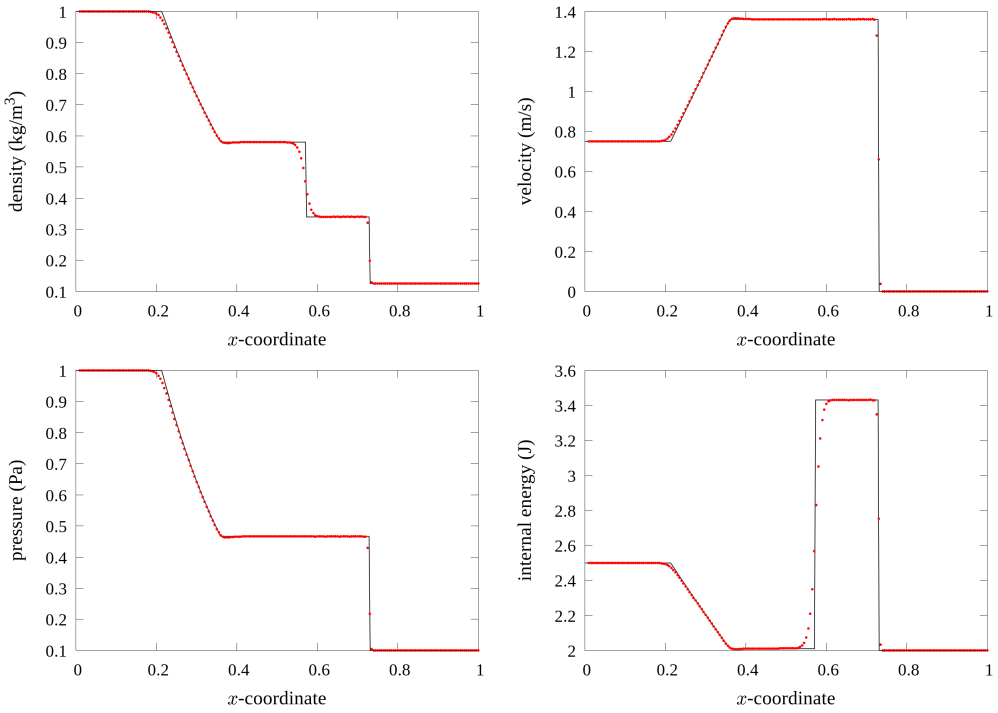


Fig. 3.6 Results for configuration 1 at time $t = 0.2\,\mathrm{s}$.

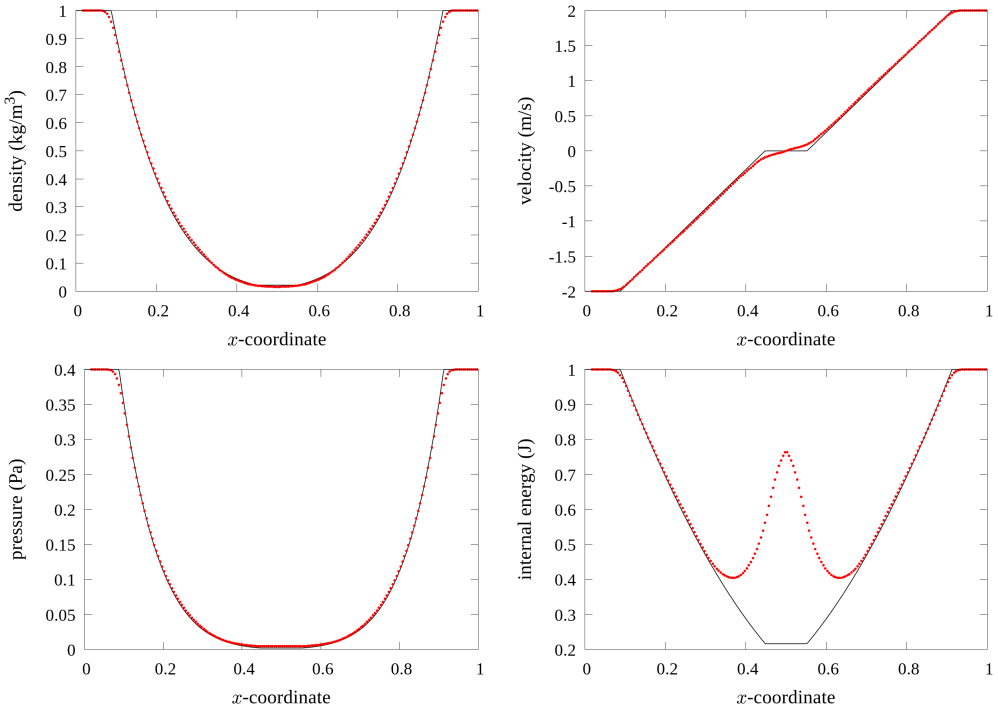


Fig. 3.7 Results for configuration 2 at time $t = 0.15\,\mathrm{s}$.

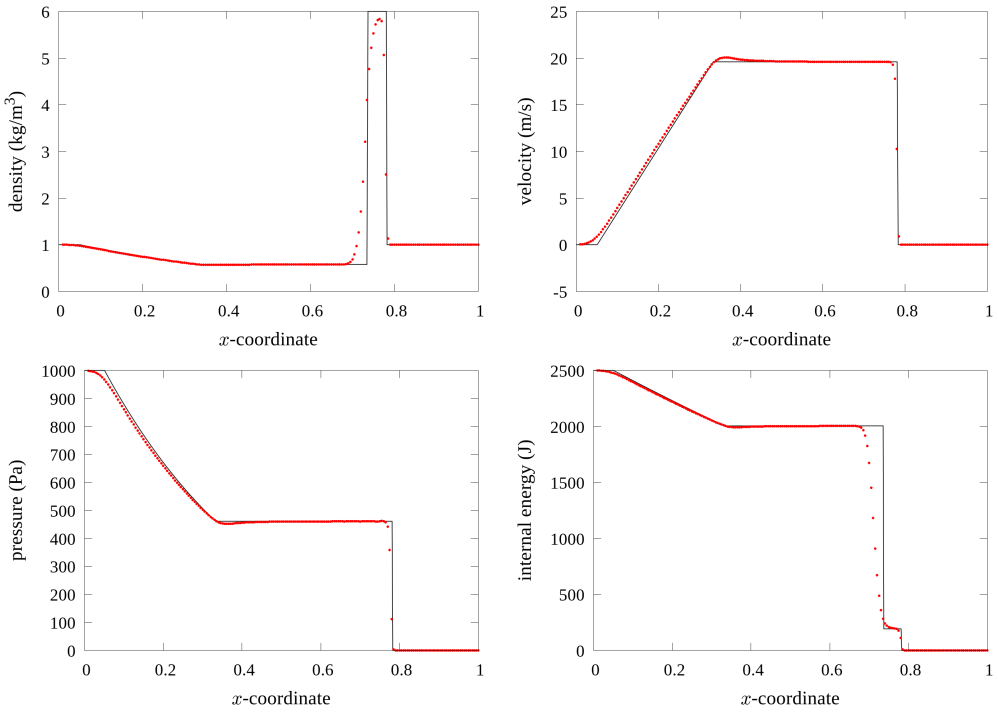


Fig. 3.8 Results for configuration 3 at time $t = 0.012\,\mathrm{s}$.

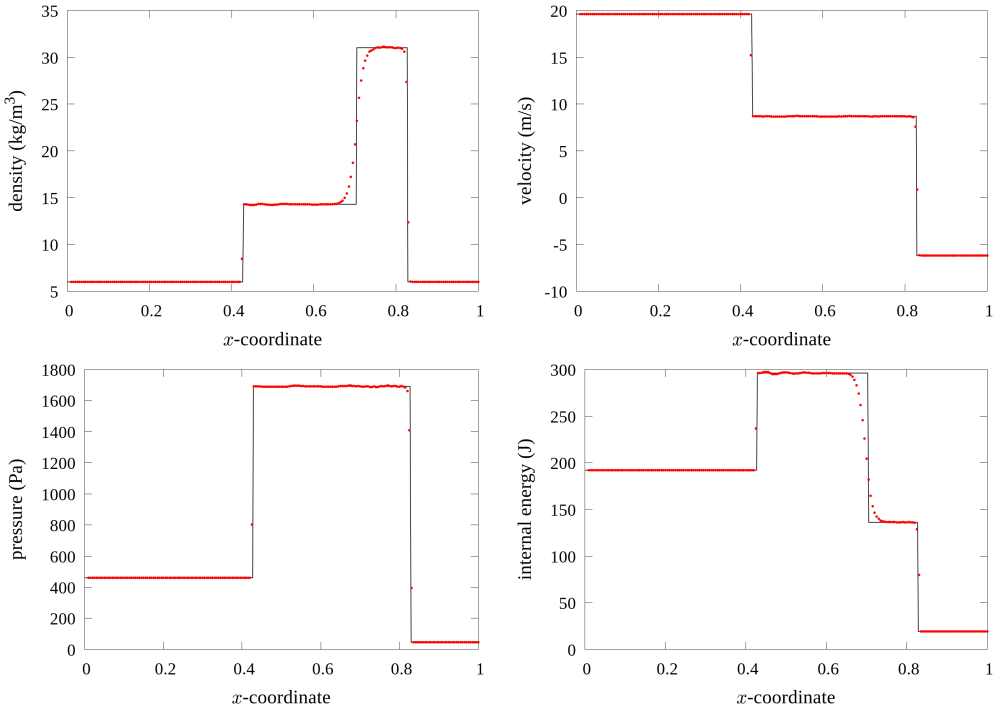


Fig. 3.9 Results for configuration 4 at time $t = 0.035\,\mathrm{s}$.

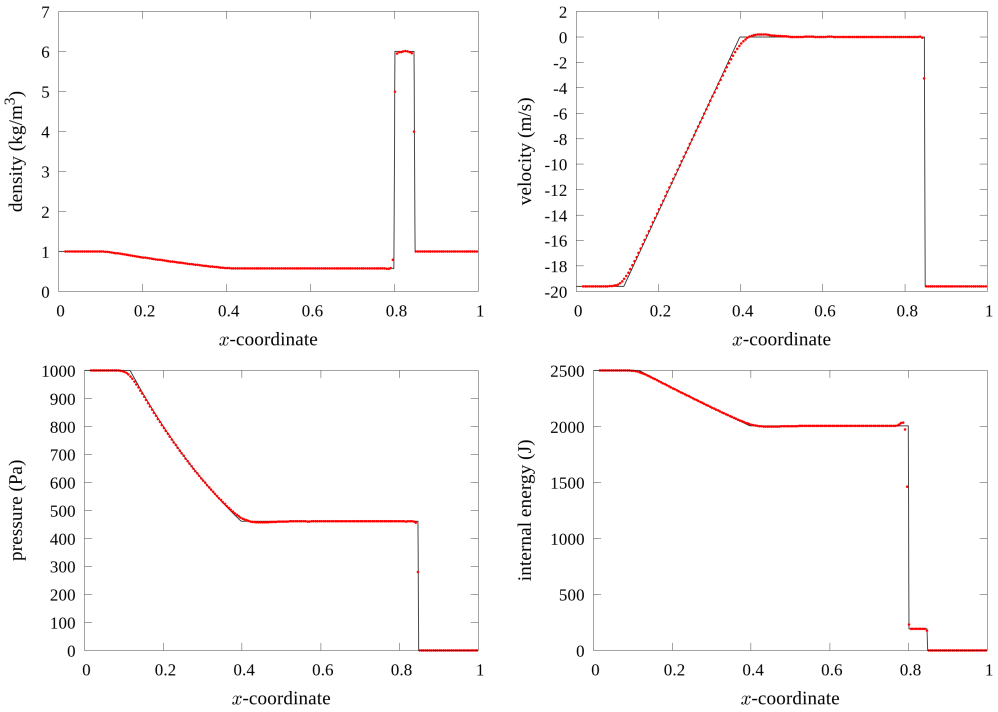


Fig. 3.10 Results for configuration 5 at time $t = 0.012\,\mathrm{s}$.

3.11.2 Two-dimensional solver

To test the correctness of the dimensional splitting, we use the two-dimensional equivalent of the shock tube problems by Kurganov and Tadmor [32]. A 2D domain is divided into four equal quadrants. The initial conditions in each part are listed in table 3.2. Figure 3.11 shows the ordering of the divisions. Two scenarios were simulated with resolution 400×400 and $C_{\mathrm{cfl}} = 0.475$ as given by Kurganov and Tadmor. At the start of the simulation, the membranes are removed and the system is allowed to evolve to a specified end time. By comparing against the numerical results of Kurganov and Tadmore we are able to verify the correctness of the data transpose framework and the operator splitting of the solver. Figure 3.12a and 3.12b show the density contours of the results taken from [32]. Figures 3.12c and 3.12d show the density contours of our implementation. We note the excellent match of the two approaches and our code's ability to faithfully capture complex features.

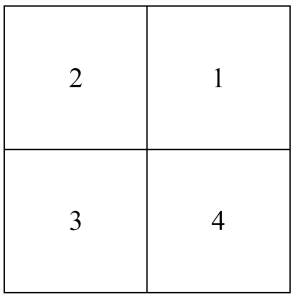


Fig. 3.11 Four quadrants of the 2D shock tube setup.

Q	ρ	u	v	p
1	1.1	0.0	0.0	1.1
2	0.5065	0.8939	0.0	0.35
3	1.1	0.8939	0.8939	1.1
4	0.5065	0.0	0.8939	0.35

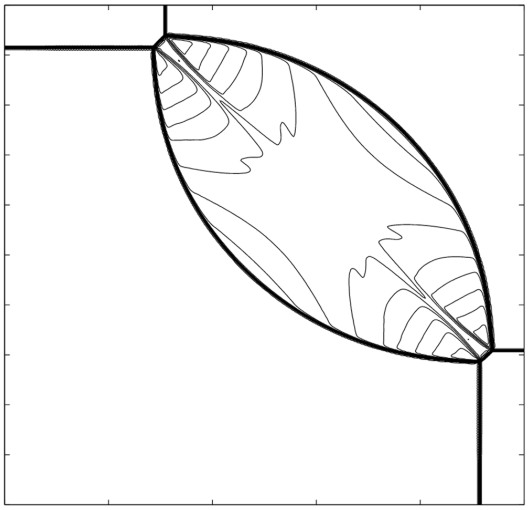
(a) Configuration 1

Q	ρ	u	v	p
1	1.0	0.75	-0.5	1.0
2	2.0	0.75	0.5	1.0
3	1.0	-0.75	0.5	1.0
4	3.0	-0.75	-0.5	1.0

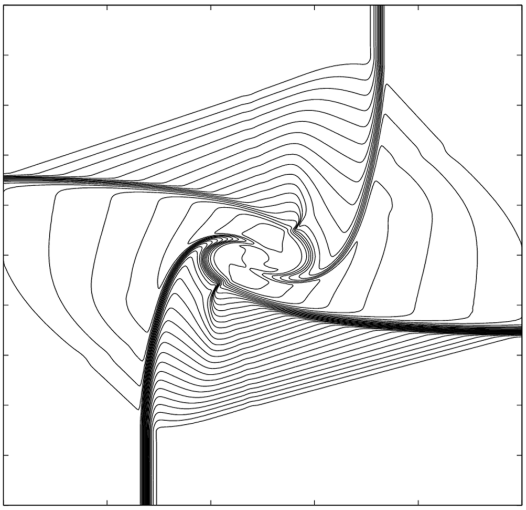
(b) Configuration 2

Table 3.2 Initial conditions in the four quadrants Q.

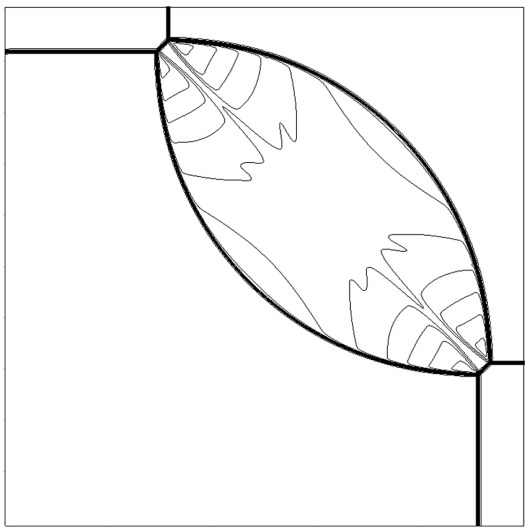
A bubble collapse simulation was also run in 2D. A Mach 1.22 shock wave hits a cylindrical bubble of ideal gas. As it passes, the collapse geometry provides a qualitative test for the software. We compare against the experimental results of a hydrogen bubble collapse by Haas and Sturtevant [22] and the numerical implementation by Quirk and Karni [51]. The experimental set up of Haas and Sturtevant used a mixture of helium and air for the bubble which Quirk and Karni calculate to have $\gamma = 1.648$. Our implementation only handles single material simulations with a constant ratio of specific heats across the domain. We used a 1600×1000 resolution mesh with $\gamma = 1.4$. The state ahead of the shock wave is $\rho = 1.225\,\mathrm{kg\,m\,s^{-3}}$, $u = v = 0\,\mathrm{m\,s^{-1}}$ and $p = 101\,325\,\mathrm{Pa}$. The initial state inside the 5 cm diameter bubble is $\rho = 0.182\,\mathrm{kg\,m\,s^{-3}}$, $u = v = 0\,\mathrm{m\,s^{-1}}$ and $p = 101\,325\,\mathrm{Pa}$. Cells at the boundary of the bubble have an initial condition that mixes the ambient and bubble variables based on the proportion of each in the volume which results in a smooth interface. The results are shown in figures 3.13 – 3.16. We note our implementation matches both the experimental Schlieren images as well as the numerical plots of Quirk and Karni. As the shock wave hits the bubble, we observe several complex flow features like the shock reflections from the initial surface as well as the far surface. Though we capture all of the features seen in the experiment and other simulations, their timings are different from the two sources as a lower γ affects the compressibility of the bubble. Nevertheless, the overall collapse of the bubble matches the two sources from literature with eventual separation and emerging vortices.



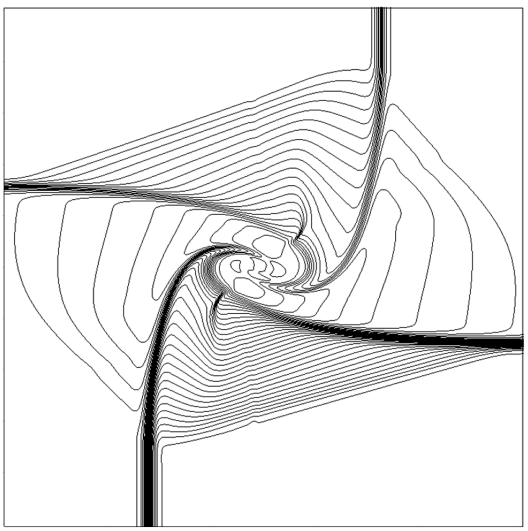
(a) Numerical density contours for configuration 1 at time $t = 0.25\,\mathrm{s}$ from [33]



(b) Numerical density contours for configuration 2 at time $t = 0.3\,\mathrm{s}$ from [33]

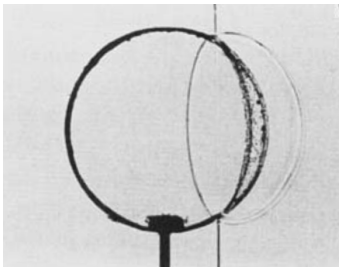


(c) Numerical density contours for configuration 1 at time $t = 0.25\,\mathrm{s}$ of our implementation

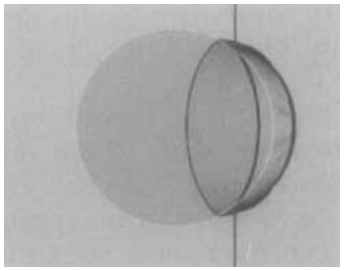


(d) Numerical density contours for configuration 2 at time $t = 0.3\,\mathrm{s}$ of our implementation

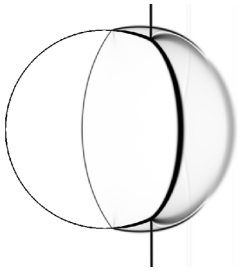
Fig. 3.12 Density contours for 2D shock tube problems.



(a) Experimental shadowgraph from [22]

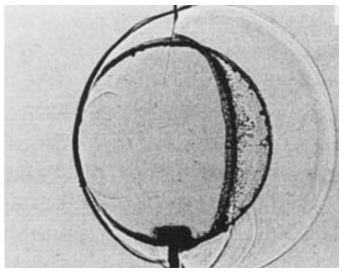


(b) Numerical Schlieren from [51]

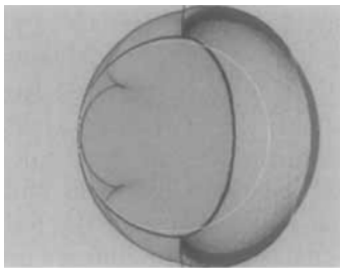


(c) Numerical Schlieren of our implementation

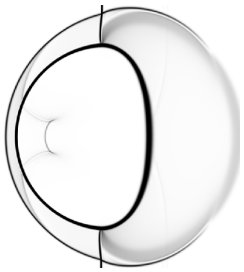
Fig. 3.13 Bubble collapse at $t = 32\,\mu s$.



(a) Experimental shadowgraph from [22]

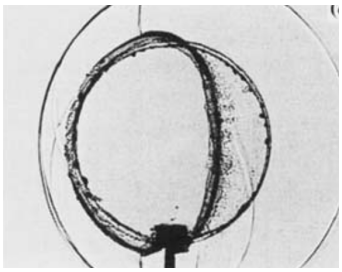


(b) Numerical Schlieren from [51]

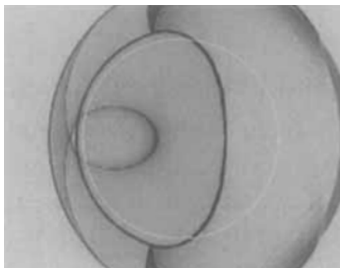


(c) Numerical Schlieren of our implementation

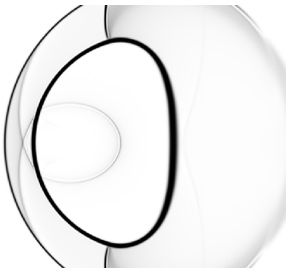
Fig. 3.14 Bubble collapse at $t = 62\,\mu s$.



(a) Experimental shadowgraph from [22]



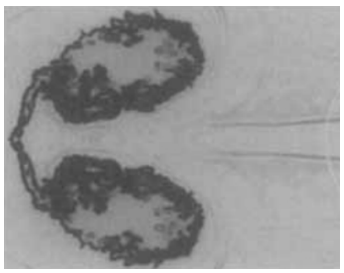
(b) Numerical Schlieren from [51]



(c) Numerical Schlieren of our implementation

Fig. 3.15 Bubble collapse at $t = 82\,\mu s$.

(a) Experimental shadowgraph from [22]



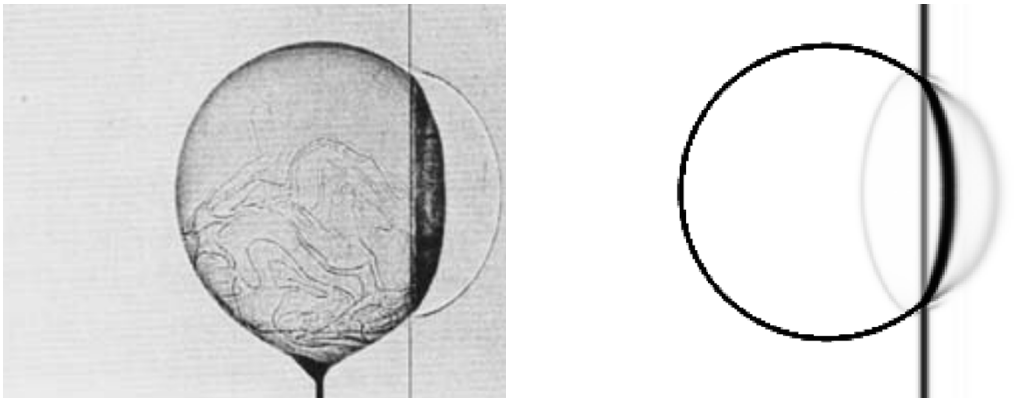
(b) Numerical Schlieren from [51]

(c) Numerical Schlieren of our implementation

Fig. 3.16 Bubble collapse at $t = 672\,\mu s$.

3.11.3 Three-dimensional solver

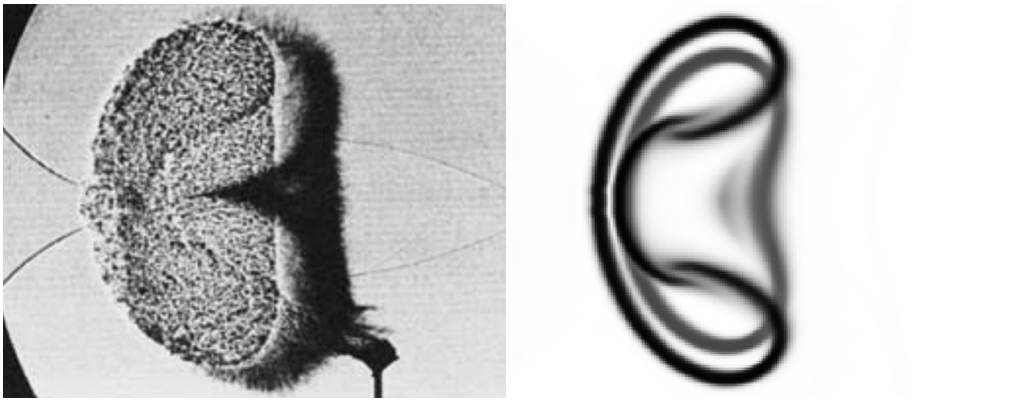
The 3D solver was validated against a spherical bubble collapse. A Mach 1.25 shock wave in an ambient gas with $\rho = 1.225\,\mathrm{kg\,m\,s^{-3}}$, $u = v = 0\,\mathrm{m\,s^{-1}}$ and $p = 101\,325\,\mathrm{Pa}$ hits a bubble of air and helium mixture with a diameter of 4.5 cm and $\rho = 0.182\,\mathrm{kg\,m\,s^{-3}}$, $u = v = 0\,\mathrm{m\,s^{-1}}$ and $p = 101\,325\,\mathrm{Pa}$. The simulation used a resolution of $480 \times 300 \times 300$. The results were validated against the shadowgraph images from Haas and Strurtevant [22]. Figures 3.17 – 3.20 show overlaid numerical Schlieren slices at different simulation times. The collapse of the bubble is captured well and exhibits the same geometry as those in literature as we capture the initial collapse and the consequent torus shape. The software is capable of handling three-dimensional shock wave dynamics and resolving detailed flow features.



(a) Experimental shadowgraph from [22]

(b) Overlaid numerical Schlieren slices of our implementation

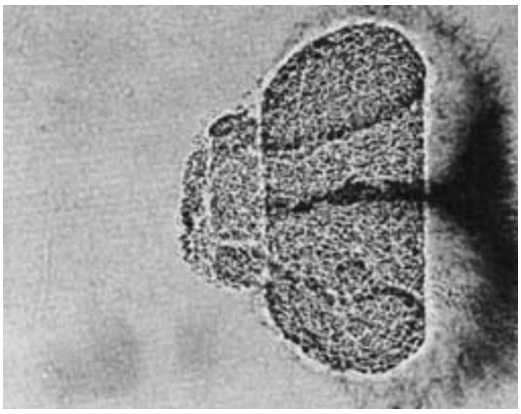
Fig. 3.17 3D Bubble collapse at $t = 20\,\mu\mathrm{s}$.



(a) Experimental shadowgraph from [22]

(b) Overlaid numerical Schlieren slices of our implementation

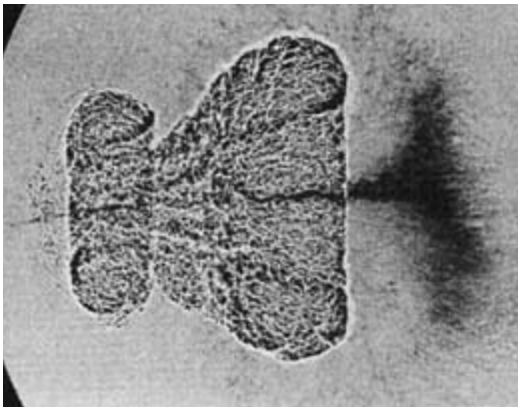
Fig. 3.18 3D Bubble collapse at $t = 223\,\mu\mathrm{s}$.



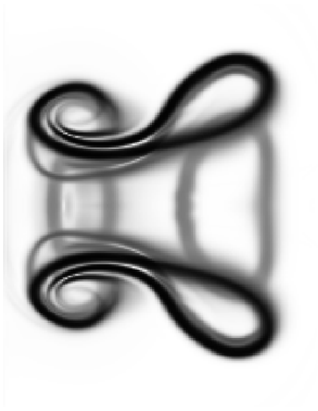
(a) Experimental shadowgraph from [22]

(b) Overlaid numerical Schlieren slices of our implementation

Fig. 3.19 3D Bubble collapse at $t = 350\,\mu s$.



(a) Experimental shadowgraph from [22]



(b) Overlaid numerical Schlieren slices of our implementation

Fig. 3.20 3D Bubble collapse at $t = 600\,\mu s$.

3.12 Performance

The data transpose between sweeps is designed to achieve coalesced memory access for the solver. As can be seen below, the data movement constitutes a considerable part of the runtime and we therefore explore the benefits of the transposition. A test environment was produced where the domain data is static and global memory access is coalesced in the x-sweep, has a stride of N_x in the y-sweep and a stride $N_x \times N_y$ in the z-sweep. 2D and a 3D bubble collapse simulations were run on a single Nvidia Tesla P100 card (table 1.1). The two-dimensional resolution is 9600×6000 and the three-dimensional resolution is $300 \times 300 \times 600$. The average time step and individual sweep durations of the strided and transposed approaches are listed in tables 3.3 and 3.4.

Data	Step (s)	Sweep (s)	Gain
Strided	0.311	x: 0.04 y: 0.27	1×
Transposed	0.097	x: 0.048 y: 0.048	3.21×

Table 3.3 The average times for the time step and sweeps of a 2D bubble collapse at a resolution 9600×6000 on a P100 card. Rearranging the data reduces the time step size by making the access pattern of each sweep equal. Transposing reduces the average duration of a time step 3.21 times.

Data	Step (s)	Sweep (s)	Gain
Strided	0.374	x: 0.04 y: 0.11 z: 0.22	1×
Transposed	0.181	x: 0.06 y: 0.06 z: 0.06	2.07×

Table 3.4 The average times for a 3D bubble collapse at a resolution $300 \times 300 \times 600$ on a P100 card. Transposing reduces the average duration of a time step 2.07 times.

From tables 3.3 and 3.4 we see the clear benefits of transposing data for optimal access patterns. With strided access the sweeps take different amounts of time as data is accessed at varying intervals. With a transposed approach we see a 3.21 times reduction in the average time step in 2D and 2.07 times faster steps in 3D. For the transpose cases, the listed sweep times include the rearrangement for the next sweep. The transposition is quick as we use highly optimised methods from literature. The sweep durations are unified as, unlike strided access, data access speeds are independent of the dimension in which the solver works. We can conclude that the data access patterns are an important consideration for modern GPU hardware and our solver software is designed to take advantage of the underlying memory hardware.

Kernel proportions

To measure the work proportions of different sections of the algorithm we analyse several simulation runs using the nvprof and Nvidia Visual Profiler (NVVP) tools [42]. These allow us to collect information about various runtime aspects. The simulations were run on an Nvidia Tesla K20 card.

For 2D profiling three ideal gas bubble collapse simulations were run to an end time $t = 675\,\mu s$ at increasing resolution using a Mach 1.22 shock wave. Table 3.6 shows the proportions of the total runtime taken by the most significant kernels. The majority of the time is spent in flux calculation kernels with the transposes making up another large portion. As the resolution increases, the time step kernel becomes less prominent and the flux and transpose kernel work both increase. Table 3.5 shows the runtimes and step counts of the three simulations without collecting profiling data. The times include everything from mesh memory allocation until the end of the simulation. When doubling the resolution the workload increases 4 times and the time step size is halved so to reach the same end time in a simulation requires 8 times more work. We see a 7.8 times runtime increase with the first doubling and 8.1 times with the second. Table 3.5 shows that the runtime scales with the resolution, the code has no significant constant time bottlenecks and the GPU resources are used efficiently.

Resolution	Steps	Time (s)
1600×1000	8902	124.9
3200×2000	18039	987.1
6400×4000	36635	7970.2

Table 3.5 2D bubble collapse on a K20 card.

Kernel	1600×1000	3200×2000	6400×2000
Flux	58.8%	58.4%	57.8%
Update	20.8%	21.0%	21.1%
Transpose	15.6%	16.0%	16.7%
Time step	3.5%	3.5%	3.5%
Boundary	0.4%	0.2%	0.1%

Table 3.6 Proportion of time spent on different kernels for 2D bubble collapse on a K20 card.

For 3D profiling similar bubble collapse simulations were run using a Mach 1.25 shock wave to a constant end time $t = 500\,\mu s$. Table 3.8 shows the proportions of the total runtime taken by the most significant kernels. We note the change in kernel order. The majority of the time is again spent in flux calculation kernels with the transposes becoming more significant as there is more data to shuffle and the 3D transpositions are more expensive. With the increase of resolution, the

time taken by these two sets of kernels also increases. Table 3.7 shows the simulation durations and the number of time steps when no profiling data is collected. When doubling resolution in 3D the workload increases 8 times for each time step. For twice as many time steps a total of 16 times more work is done to reach a constant end time. We observe a 15.3 times runtime increase with the first doubling and 15.2 times with the second. Table 3.7 illustrates that the implementation scales well with the increase of resolution.

Resolution	Steps	Time (s)
$120 \times 75 \times 75$	461	7.0
$240 \times 150 \times 150$	976	105.2
$480 \times 300 \times 300$	2035	1604.9

Table 3.7 3D bubble collapse on a K20 card.

Kernel	$120 \times 75 \times 75$	$240 \times 150 \times 150$	$480 \times 300 \times 300$
Flux	52.2%	53.2%	54.1%
Transpose	21.1%	22.9%	22.4%
Update	17.4%	17.7%	18.9%
Boundary	6.8%	3.9%	2.1%
Time step	1.9%	1.9%	2.0%

Table 3.8 Proportion of time spent on different kernels for 3D bubble collapse on a K20 card.

3.13 Conclusion

We have discussed the Euler equations for modelling fluids and the HLLC solver together with the MUSCL-Hancock method. We have implemented a CUDA fluid solver for GPUs by using a one-dimensional stencil approach and data transposes to obtain good memory access patterns. We have also described how we use shared memory and registers to cache data closer to the GPU cores. By mapping a thread to each cell interface we can make use of neighbouring information to reuse data close to the cores. The one-dimensional solver was validated against several shock-tube problems to demonstrate how the chosen methods deal with various wave configurations. Two-dimensional simulations were run to validate the transpose mechanism and show the software capturing complex flow features. A three-dimensional simulation demonstrates the functionality of the full solver code. By running resolution comparisons and profiling kernel work we have shown the portion of time taken up by different components. The performance of the software is good as we have achieved good memory access patterns and the majority of the work is spent on flux calculations which scale well with the resolution. In the next chapter we will describe the theory and implementation of solid geometries embedded in the computational domain.

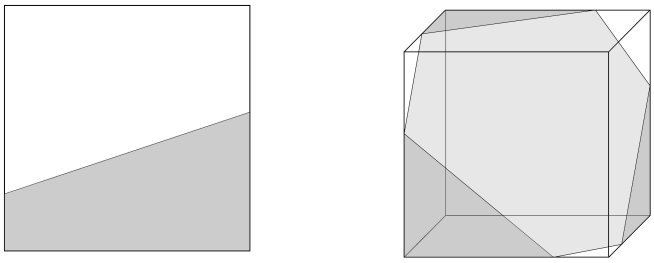
Chapter 4

Cut Cells

Cut cell methods describe embedded boundaries while retaining a Cartesian mesh structure. The approach creates a discretised representation of object surfaces while satisfying conservation at the boundary. Cut cell grids can be generated automatically and feature no distortion in cells away from the boundaries. Klein et al. [28] present a dimensionally split cut cell algorithm which is suited for parallelisation and can be implemented in a straightforward way to interface with our split solver. We make use of the Localised Proportional Flux Stabilisation (LPFS) extension by Gokhale et al. [21] which results in fewer oscillations at higher CFL numbers and leads to smoother solutions at stagnation points. This chapter will discuss the numerical methods of the cut cell solver, efficient data structures and a CUDA implementation to interface with the solver discussed previously. We validate the implementation using several test cases and conclude with a performance analysis.

There are various ways complex boundaries can intersect a cell but we limit the problem to lines or planes. For a 2D rectangular cell an intersection is only well defined for a straight line between two points. In 3D the intersection is a plane with three to six intersection points with the sides of a cuboid cell (figure 4.1).

A problem can arise where a cell is almost entirely inside a solid. When the fluid portion of a cut cell is small, it may limit the stable time step size of the whole simulation. This is known as the *small cell problem* and has been addressed in various ways. Clarke et al. [15] merge sufficiently small cells into neighbouring ones to avoid the instabilities. A similar approach is employed by Hartmann et al. [24] with adaptive mesh refinement in 3D. Another strategy is the flux distribution of Colella et al. [17] where the range of cut cell influence is increased and the non-conservative part of the calculated flux is distributed between surrounding cells. The h-box solution of Berger and Helzel [7] extends the influence of cut cells to regular cell dimensions to avoid the dependence on cell volume size in flux updates.



(a) 2D cell cut by a line (b) 3D cell cut by a plane

Fig. 4.1 Cut cell meshes model surfaces from intersections with a geometry. The zero level set data at cell edges and faces is used to construct a linear approximation of the boundary. In 2D a cell is cut by a line and in 3D we use a plane. The intersection divides a cell volume into clear (white) and solid (grey) regions.

4.1 Klein-Bates-Nikiforakis cut cell method

The above solutions exist within unsplit frameworks which are generally more difficult to parallelise well for a GPU. Klein et al. [28] demonstrate a dimensionally split cut cell algorithm which we will call the Klein-Bates-Nikiforakis (KBN) method. The method deals with the small cell problem by defining one-dimensional fluxes based on the cell volume without reducing the global time step. The method is fully conservative at boundaries, shows good stability and works with different flux approximation methods. Bennett et al. [6] have shown that KBN cut cells can be used to simulate moving boundaries without the need for the costly regeneration of body fitted grids.

4.1.1 Fluxes

The KBN method defines an intercell flux based on the size of a cut cell. Consider the one-dimensional case shown in figure 4.2.

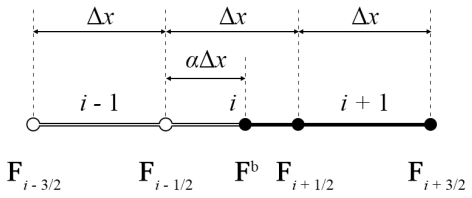


Fig. 4.2 A 1D cell i is cut by a boundary on the right. Cell $i-1$ is fully fluid and cell $i+1$ is fully solid. Cell i has a fluid fraction $\alpha\Delta x$ with a regular flux acting on its left interface and a boundary flux at its interface with the solid.

Cell i is cut by a boundary to the right and has a volume fraction $\alpha\Delta x$. If we extend the influence of cell i as if it were of regular size, the update would be:

$$\mathbf{U}_i^{t+\Delta t} = \mathbf{U}_i^t + \frac{\Delta t}{\Delta x}\left(\mathbf{F}_{i-\frac{1}{2}} - \mathbf{F}^{\mathrm{b}}\right), \tag{4.1}$$

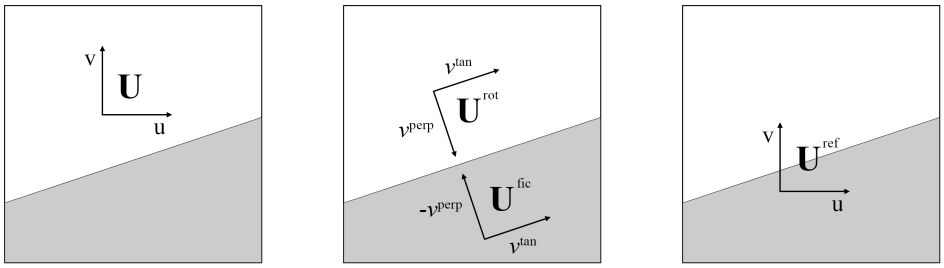
where $\mathbf{F}_{i-\frac{1}{2}}$ is the flux between cells and $\mathbf{F}^{\mathrm{b}}$ is the boundary flux. The conservative update for the cut cell would have to be:

$$\mathbf{U}_i^{t+\Delta t} = \mathbf{U}_i^t + \frac{\Delta t}{\alpha\Delta x}\left(\mathbf{F}^{\mathrm{KBN}}_{i-\frac{1}{2}} - \mathbf{F}^{\mathrm{b}}\right), \tag{4.2}$$

which gives the intercell flux between a cut cell and a regular neighbour:

$$\mathbf{F}^{\mathrm{KBN}}_{i-\frac{1}{2}} = \mathbf{F}^{\mathrm{b}} + \alpha\left(\mathbf{F}_{i-\frac{1}{2}} - \mathbf{F}^{\mathrm{b}}\right). \tag{4.3}$$

$\mathbf{F}^{\mathrm{b}}$ is a flux function of a reference state $\mathbf{F}\left(\mathbf{U}^{\mathrm{ref}}\right)$. The reference state is calculated as the solution to the Riemann problem in the wall-normal direction as shown in figure 4.3. In higher dimensions this is achieved by first rotating the physical state within the cell such that we find the normal and tangential velocity components of the flow (figure 4.3b). A fictitious state is then created by reflecting the normal component of the rotated state, thereby representing the reflective boundary condition at the interface. We find the reference state as the Riemann solution with the initial condition of the rotated and fictitious states and rotate it back to the original reference frame (figure 4.3b). The reference state is found at the start of a time step and is kept constant except for the pressure which is updated every sweep from the solution of the wall-normal Riemann problem using data from the last sweep.



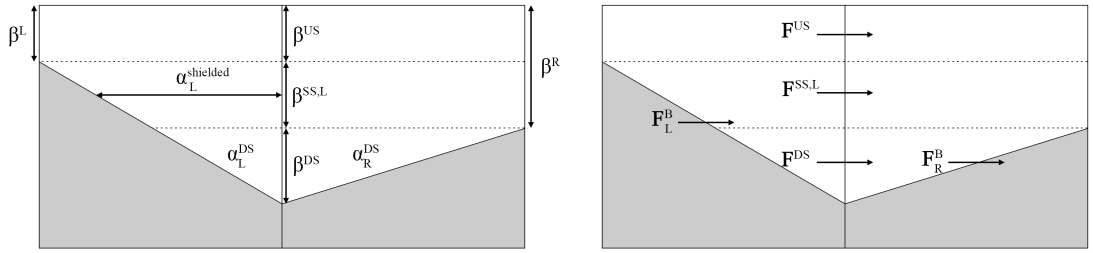
(a) State $\mathbf{U}$ in 2D

(b) Rotated and reflected states $\mathbf{U}^{\mathrm{rot}}$ and $\mathbf{U}^{\mathrm{fic}}$

(c) Reference state $\mathbf{U}^{\mathrm{ref}}$

Fig. 4.3 Calculating the reference state in 2D. The state $\mathbf{U}$ with velocities $\mathbf{u}$ and $\mathbf{v}$ in a cut cell (a) is rotated to match the wall normal to find the tangential component $v^{\tan}$ and perpendicular component v^{perp}(b). A reflected state $\mathbf{U}^{\mathrm{fic}}$ is found and the solution to the Riemann problem with the initial data of $\mathbf{U}^{\mathrm{rot}}$ and $\mathbf{U}^{\mathrm{fic}}$ gives the reference state $\mathbf{U}^{\mathrm{ref}}$ which is rotated back to the domain orientation (c).

In higher dimensions cut cells are divided into different regions based on the flow influences in a given direction. We define three regions in each cell that describe the influence these areas are subject to as shown in figure 4.4. The unshielded regions (US) behave like clear cells, areas singly shielded from the left or right (SS,L/R) are affected by a boundary on one side and doubly shielded areas (DS) see a boundary on both sides in the sweep direction. The fluxes across cell boundaries are combinations of singly shielded, doubly shielded and unshielded fluxes weighted by their proportion of the cell interface.



(a) Geometry variables

(b) Flux components

Fig. 4.4 Cut cells in 2D are divided into solid and clear regions. The clear regions are further divided based on shielding. (a) shows the normalised face fractions β and volume fractions α. (b) shows the flux components acting on each region.

4.1.2 Cut cell geometry

Cut cells define additional spatial variables describing the local fluid/solid ratio in each cell. The data is generated from signed distance field (SDF) values at cell vertices. The SDF generation at discrete points is described in Chapter 2. A zero crossing in the SDF on a cell face signals a boundary intersection and a cut cell. For each direction we define face fractions β to denote how much of a cell interface is uncovered by the solid. This is expressed in nondimensionalised form where the value 1 denotes a fully clear interface and 0 signals a cell face completely within the solid (figure 4.4a). In 2D, the left face fraction of cell (i, j) can be found from the signed distance field ϕ values at its vertices:

$$\beta_{i-\frac{1}{2},j} = \begin{cases} \left| \dfrac{\phi_{i-\frac{1}{2},j-\frac{1}{2}}}{\left|\phi_{i-\frac{1}{2},j-\frac{1}{2}}\right| + \left|\phi_{i-\frac{1}{2},j+\frac{1}{2}}\right|} \right| & \phi_{i-\frac{1}{2},j-\frac{1}{2}} > 0\,, \\ 1 - \left| \dfrac{\phi_{i-\frac{1}{2},j-\frac{1}{2}}}{\left|\phi_{i-\frac{1}{2},j-\frac{1}{2}}\right| + \left|\phi_{i-\frac{1}{2},j+\frac{1}{2}}\right|} \right| & \phi_{i-\frac{1}{2},j-\frac{1}{2}} < 0\,. \end{cases} \tag{4.4}$$

For 3D cells the face fraction is:

$$\beta = \frac{1}{2A} \sum_{i=1}^{n} \left(x_i y_{i+1} - x_{i+1} y_i \right), \tag{4.5}$$

with respect to the intersection points $(x_1 y_1), ..., (x_n y_n)$ and the face area A of regular cells. In 2D the above equation gives the volume fraction α of the cell. For a 3D case we can express the volume fraction as the volume of a convex polyhedron:

$$\alpha = \frac{1}{3V} \left| \sum_{i=1}^{N_\mathrm{F}} \left(\mathbf{x}_i \cdot \hat{\mathbf{n}}_i \right) A_i \right|, \tag{4.6}$$

where N_F is the face count, $\mathbf{x}_i$ is a point on a face, $\hat{\mathbf{n}}_i$ is a face normal, A_i is the area of a face and V is the volume of a regular cell. For the wall-normal Riemann problem the interface normal $\mathbf{n}$ at a cut cell can be found from the face fractions:

$$\mathbf{n} = \begin{bmatrix} \beta_{i-\frac{1}{2},j,k} - \beta_{i+\frac{1}{2},j,k} \\ \beta_{i,j-\frac{1}{2},k} - \beta_{i,j+\frac{1}{2},k} \\ \beta_{i,j,k-\frac{1}{2}} - \beta_{i,j,k+\frac{1}{2}} \end{bmatrix}. \tag{4.7}$$

The cell interfaces can then be divided into unshielded, singly shielded and doubly shielded portions. We also define the portions $\alpha_{\mathrm{L/R}}^{\mathrm{DS}}$ of cell volumes which are doubly shielded and the average distance $\alpha_{\mathrm{L/R}}^{\mathrm{shielded}}$ from the cell interface to the boundary for singly shielded regions (figure 4.4a). The implicit SDF definition is therefore transformed into an explicit representation which depends on the resolution of the domain.

4.2 Scheme update

After determining the appropriate geometry and fluxes, the scheme updates the mixed cells. Gokhale et al. [21] give a mixing flux for doubly shielded areas:

$$\mathbf{F}^{\mathrm{DS}} = \frac{1}{\left(\alpha_{\mathrm{L}}^{\mathrm{DS}} + \alpha_{\mathrm{R}}^{\mathrm{DS}}\right)} \left[\frac{\alpha_{\mathrm{L}}^{\mathrm{DS}} \alpha_{\mathrm{L}}^{\mathrm{DS}} \Delta x}{\beta^{\mathrm{DS}} \Delta t} \left(\mathbf{U}_{\mathrm{L}}^{n} - \mathbf{U}_{\mathrm{R}}^{n} \right) + \alpha_{\mathrm{L}}^{\mathrm{DS}} \mathbf{F}_{\mathrm{R}}^{\mathrm{B}} + \alpha_{\mathrm{R}}^{\mathrm{DS}} \mathbf{F}_{\mathrm{L}}^{\mathrm{B}} \right]. \tag{4.8}$$

The modified flux at a cell interface is defined as the area-weighted sum of the individual components:

$$\mathbf{F}^{\mathrm{modified}} = \frac{1}{\beta} \left[\beta^{\mathrm{US}} \mathbf{F}^{\mathrm{US}} + \beta^{\mathrm{SS,L}} \mathbf{F}^{\mathrm{SS,L}} + \beta^{\mathrm{SS,R}} \mathbf{F}^{\mathrm{SS,R}} + \beta^{\mathrm{DS}} \mathbf{F}^{\mathrm{DS}} \right], \tag{4.9}$$

where $\mathbf{F}^{\mathrm{SS}}$ is the appropriate $\mathbf{F}^{\mathrm{KBN}}$ flux. The x-sweep update of the state vector for a cut cell is:

$$\mathbf{U}_{i,j}^{n+\frac{1}{2}} = \mathbf{U}_{i,j}^{n} + \frac{\Delta t}{\alpha_{i,j}\Delta x}\left[\beta_{i-\frac{1}{2},j}\mathbf{F}_{i-\frac{1}{2},j}^{\mathrm{modified}} - \beta_{i+\frac{1}{2},j}\mathbf{F}_{i+\frac{1}{2},j}^{\mathrm{modified}} - \left(\beta_{i-\frac{1}{2},j} - \beta_{i+\frac{1}{2},j}\right)\mathbf{F}_{i,j}^{\mathrm{B}}\right]. \tag{4.10}$$

The equivalent updates must be done in the y-direction and in the case of a 3D simulation, for the z-sweep.

4.3 Localised Proportional Flux Stabilisation

We use the Localised Proportional Flux Stabilisation (LPFS) extension by Gokhale et al. [21] to modify the above flux calculations. Using information about the geometry and wave speed inside the cut cell, the LPFS method redefines the cut cell flux for smoother results at low-speed flows. The approach is to apply the explicit flux $\mathbf{F}_{i-\frac{1}{2}}$ for a proportional period Δt_{cc} of the time step where it is known to be stable and a mixing flux for the remainder, where:

$$\Delta t_{\mathrm{cc}} = C_{\mathrm{cfl}}\frac{\alpha\Delta x}{W_i}, \tag{4.11}$$

and W_i is the wave speed in the cut cell. This results in more of the flux being applied to the cut cell which Gokhale et al. report leads to improved results.

We define a modified KBN flux:

$$\mathbf{F}_{i-\frac{1}{2}}^{\mathrm{KBN,mod}} = \mathbf{F}^{\mathrm{b}} + \frac{\Delta t_{\mathrm{cc}}}{\Delta t}\left(\mathbf{F}_{i-\frac{1}{2}} - \mathbf{F}^{\mathrm{b}}\right). \tag{4.12}$$

The LPFS flux is then expressed as:

$$\mathbf{F}_{i-\frac{1}{2}}^{\mathrm{LPFS}} = \frac{\Delta t_{\mathrm{cc}}}{\Delta t}\mathbf{F}_{i-\frac{1}{2}} + \left(1 - \frac{\Delta t_{\mathrm{cc}}}{\Delta t}\right)\mathbf{F}_{i-\frac{1}{2}}^{\mathrm{KBN,mod}}. \tag{4.13}$$

The time step ratio is :

$$\frac{\Delta t_{\mathrm{cc}}}{\Delta t} = \epsilon\frac{\alpha W_{\max}}{W_i}, \tag{4.14}$$

where ϵ is a user defined parameter in the range $[0, 1]$ to account for errors arising from estimating the wave speeds in a cut cell. Both the KBN and the LPFS fluxes are first order at the boundary. The LPFS flux, however, produces smoother solutions near stagnation points and reduces the oscillations produced by the KBN flux at higher C_{cfl} numbers.

Gokhale et al. show that fully doubly shielded cells can cause unphysical states and that this can be remedied by state mixing. After every sweep we mix the states of the cells at fully doubly shielded regions, effectively reducing the resolution in these areas. We set the value of each cell in a small area around the doubly shielded pair to the volume average state of the region as $\mathbf{U} = (\sum_{i=1}^{n} \alpha_i \mathbf{U}_i) \,/\, (\sum_{i=1}^{n} \alpha_i)$, where n is the number of cells in the area. We use a 3×3 mixing area in 2D and a $3 \times 3 \times 3$ volume in 3D. The average state is based on the data after a dimensional sweep update and is carried out in each direction. As the regions may overlap, we impose a serial update of doubly shielded regions to avoid race conditions between threads but this introduces only a minimal impact on the runtime. This approach is not a rigorous fix for the methods' inability to handle small concavities but at high resolutions there are few fully doubly shielded cells and we observe good stability in simulations.

4.4 Implementation

In our pipeline, a simulation starts by allocating memory for the SDF and calculating a narrow band signed distance field around the surface of an input geometry. The SDF data is defined at cell vertices but the flow variable indices map to cell centres. A (N_x, N_y, N_z) domain will then also need a $(N_x + 1, N_y + 1, N_z + 1)$ grid to hold a single 32 bit SDF value per domain cell vertex. The signed distance field values at the eight corners of domain cell (x, y, z) are stored in SDF cells

$$\{(x + i, y + j, z + k) \mid (i, j, k) \in \{0, 1\}^3\}. \tag{4.15}$$

4.4.1 Cut cell variables

We would like to use most of a GPU's memory to store flow variables and fluxes thereby maximising the resolution of a simulation. For simulations without any cut cells, the vectors $\mathbf{U}$ and $\mathbf{F}$ constitute the main memory footprint. Consider, however, a simulation with cut cells. Naïvely we would define the vectors $\mathbf{A}$ of cut cell attributes, $\mathbf{D}$ of doubly shielded attributes, $\mathbf{U}^{\text{ref}}$ of reference states and $\mathbf{F}^{\text{B}}$ of boundary fluxes for each cell:

$$\mathbf{A} = \begin{bmatrix} \alpha \\ \beta_x \\ \beta_y \\ \beta_z \\ \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{bmatrix}, \qquad \mathbf{D} = \begin{bmatrix} \alpha_x^{\text{DS}} \\ \alpha_y^{\text{DS}} \\ \alpha_z^{\text{DS}} \\ \beta_x^{\text{DS}} \\ \beta_y^{\text{DS}} \\ \beta_z^{\text{DS}} \\ \beta_x^{\text{SS}} \\ \beta_y^{\text{SS}} \\ \beta_z^{\text{SS}} \end{bmatrix}, \qquad \mathbf{U}^{\text{ref}} = \begin{bmatrix} \rho^{\text{ref}} \\ u^{\text{ref}} \\ v^{\text{ref}} \\ w^{\text{ref}} \\ p^{\text{ref}} \end{bmatrix}, \qquad \mathbf{F}^{\text{B}} = \begin{bmatrix} f_\rho^{\text{B}} \\ f_{\rho u}^{\text{B}} \\ f_{\rho v}^{\text{B}} \\ f_{\rho w}^{\text{B}} \\ f_E^{\text{B}} \end{bmatrix}. \tag{4.16}$$

For a 3D simulation, this is twenty-six 64 bit variables per cell – considerably more than the 10 variables of the state and flux vectors. However, the cut cell variables need only be stored for mixed cells. In the mesh generation phase we can identify cut cells and store their variables in a compressed format. While only a small fraction of cells in a sufficiently high resolution simulation are mixed, they are positioned along the surfaces of arbitrarily shaped objects. We construct data structures that map sparse cells into contiguous memory and use two-way maps to address cut cells and their surrounding data.

Maps

To identify which cells in the domain are mixed, we create a map array that holds two 32 bit values for each cell. The first is the flag I which initially takes a value of either 1 or 0 denoting if a cell is mixed or not. The second value T is an `enum` that denotes if a cell is clear, mixed or solid. We store the values in each cell in the vector $\mathbf{M}$:

$$\mathbf{M} = [I,\ T]^{\mathrm{T}}. \tag{4.17}$$

We find the cut cell attributes of every cell based on the SDF values at cell vertices. We then set the flag value of cut cells to 1 and 0 everywhere else. Taking the prefix sum of the I flag array will give us an array **prefix** of size N. We use the Thrust library [45] method `thrust::inclusive_scan`. The last (and therefore maximum) value of this array is the number of mixed cells in the domain which we will call C. We can then allocate contiguous memory of length $C \times 26$ to hold the vectors $\mathbf{A}$, $\mathbf{D}$, $\mathbf{U}^{\mathrm{ref}}$ and $\mathbf{F}^{\mathrm{B}}$ for only cut cells, thereby storing the data in the minimum space of memory. By multiplying every I element with the corresponding **prefix** value, we can update the map to hold the index of mixed cells within the newly created sparse arrays, allowing us to explicitly address the cut variables of any domain cell (figure 4.5).

In theory, using maps we reduce the problem of storing an extra twenty-six 64 bit values for each cell down to only storing an extra 64 bits per cell and $C \times 26 \times 64$ bits in new arrays for the cut cells. However, we need to transpose the map every sweep to match the flow variable indices in memory. We therefore need to find space for the out of place transpose of the map data. As the flux and state vectors are locked into switching places with each other between time steps, we create a new array of equal size and type to $\mathbf{M}$ which we will call $\mathbf{M}^{\mathrm{TR}}$.

There exist other compression algorithms often used with sparse arrays which do not need a map value for each cell and refer to the non-zero element information using smaller lookup tables. These are often focused on sparse matrix-vector multiplication applications. An overview of GPU specific implementations can be found in Koza et al. [31]. We use our custom approach for two reasons. Firstly we would like to have two-way direct addressing of each cut cell from a thread coordinate. Secondly, as the solver adds fluxes to all cells at the same time, the map information can be additionally used to differentiate between clear and cut cells in the update kernel.

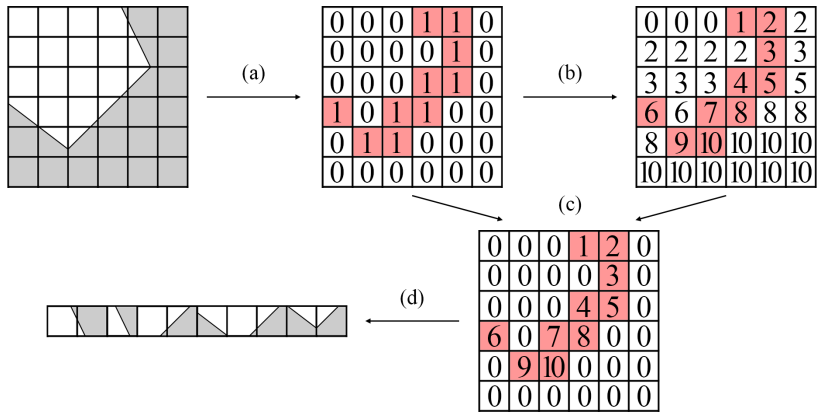


Fig. 4.5 Generation of compressed cut cell data. (a) We start with creating a map with cut cell indices set to 1 and all other cells set to 0. (b) Generating a prefix sum of the underlying 1D pointer allows us to enumerate the cut cells. (c) By multiplying the map and prefix sum we produce a new map holding the index of each cut cell with the rest of the cells set to 0. (d) From the prefix sum we know the total number of cut cells and we can allocate memory for each one while the produced map designates the position of the cell in the new compressed structure.

Further memory gains can be made when considering that only a fraction of the mixed cells are likely to be doubly shielded and vector **D** need only be stored for a subset of cut cells. We can introduce a map variable into vector **A** that marks doubly shielded cells and store **D** in a separate smaller array, the indices of which map to the indices of **A** and by extension, the computational domain. The number of doubly shielded cells is different in each direction and in our implementation we use three sparse arrays for **D** with separate map indices in **A**.

For reference states, boundary fluxes and cut cell flux modification, we launch a thread for each cut cell. We add three variables to the vector **A** to denote the x-, y- and z-index of each cut cell which will allow us to read the corresponding states in **U**. Similarly we maintain the coordinates of doubly shielded cells in each **D**. With the sparse layout of mixed cells in the domain, it is not clear if an attempt to access the spatial variables or boundary fluxes in a coalesced way from their array is possible or efficient.

In practice then, we take up 2×64 bits per cell for the cut cell map, $C \times 17 \times 64$ bits for the cut cell data and $C \times 6 \times 32$ bits for the coordinates of the cut cells and indices of doubly shielded cells. Additionally we store $C^{\mathrm{DS}} \times 9 \times 64$ bits for doubly shielded variables and $C^{\mathrm{DS}} \times 3 \times 32$ bits for the coordinates of doubly shielded indices. While these are more complicated data structures, they constitute a fraction of the memory taken up by the state and flux arrays. This approach stores only relevant information per cell with a transposable map and indices to explicitly address data.

For a three-dimensional simulation the new cut cell vector **A** and dimension specific doubly

shielded cell vectors $\mathbf{D}_\mathrm{d}$ are:

$$\begin{aligned} \mathbf{A} &= \left[\alpha,\, \beta_x,\, \beta_y,\, \beta_z,\, \hat{n}_x,\, \hat{n}_y,\, \hat{n}_z,\, i^{\mathrm{D}x},\, i^{\mathrm{D}y},\, i^{\mathrm{D}z},\, x,\, y,\, z\right]^{\mathrm{T}}, \\ 3 \times \mathbf{D}_\mathrm{d} &= \left[\alpha^{\mathrm{DS}},\, \beta^{\mathrm{DS}},\, \beta^{\mathrm{SS}},\, x,\, y,\, z\right]^{\mathrm{T}}. \end{aligned} \tag{4.18}$$

The doubly shielded and cut cell variable arrays are not transposed. Launching a thread per mixed cell does not obtain coalesced memory access of the flow or flux data but allows for larger simulations to be run on limited hardware. The resulting minimum memory footprint and quick modification kernels were two of the main aims of this project.

4.4.2 Initialisation

At the start of a simulation with N cells, we initialise data structures in the correct order to achieve the maximum possible resolution. The procedure is as follows:

1. Allocate and populate an SDF mesh.
2. Allocate and populate a dense mesh $\mathbf{A}^N$ for the cut cell variables of each cell.
3. Free the memory of the SDF mesh.
4. Allocate and populate a mesh $\mathbf{M}$ for cut cell map and its transpose space $\mathbf{M}^{\mathrm{TR}}$.
5. Generate a prefix sum of the map and update the flags in $\mathbf{M}$.
6. Allocate sparse mesh for vectors $\mathbf{A}$, $\mathbf{U}^{\mathrm{ref}}$ and $\mathbf{F}^{\mathrm{B}}$.
7. Copy attributes of only cut cells from $\mathbf{A}^N$ to $\mathbf{A}$.
8. Allocate mesh $\mathbf{M}^{\mathrm{DS}}$ for doubly shielded map.
9. For each dimension:
    - Populate $\mathbf{M}^{\mathrm{DS}}$ to mark doubly shielded cells.
    - Generate a prefix sum of the map and update the flags in $\mathbf{M}^{\mathrm{DS}}$.
    - Allocate and populate sparse mesh for vector $\mathbf{D}$.
10. Free the memories of $\mathbf{A}^N$, $\mathbf{M}^{\mathrm{DS}}$ and the prefix sums.
11. Allocate memory for $\mathbf{U}$ and $\mathbf{F}$, set initial conditions and start the simulation.

4.4.3 Simulation

Every times step the simulation solves the x-, y- and z-directions in order. For each sweep the simulation will:

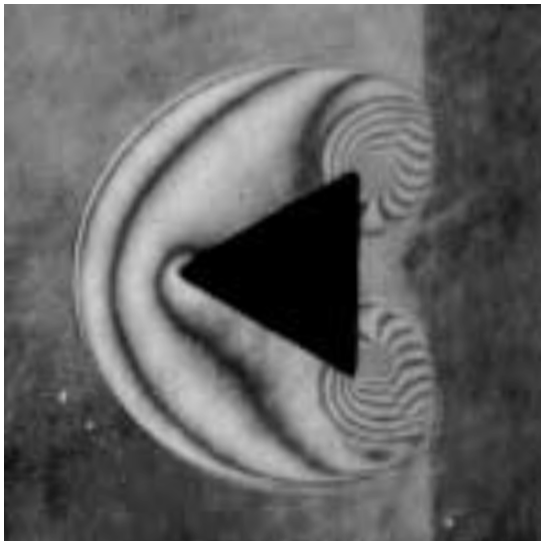
1. Calculate the intercell flux for all non-solid cells.
2. Find the reference state at cut cells.
3. Find the boundary flux at cut cells.
4. Modify the intercell flux of cut cells based on the boundary flux.
5. Update the states of all non-solid cells based on their intercell flux and geometry.

4.5 Validation

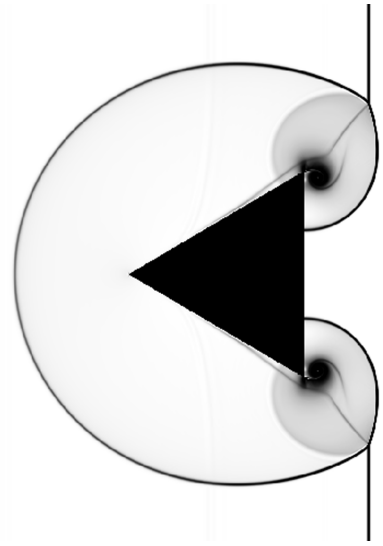
The cut cell implementation was validated against several experimental results with different geometries. This section describes the test configurations and compares our results to those in the literature. For two-dimensional simulations, we explore shock wave propagation over a wedge and a cylinder and compare against experimental and numerical results. To look at subsonic flow we simulate an aerofoil in a Mach 0.4 ideal gas and compare the resulting pressure coefficients at its surface against experimental readings. For three-dimensional validation we simulate shock waves encountering obstacles. All of the cut cell validation simulations used ambient data of $\rho = 1.225\,\mathrm{kg\,m\,s^{-3}}$ and $p = 101\,325\,\mathrm{Pa}$ with velocity 0 unless otherwise specified.

4.5.1 Shock reflection in Schardin's problem

Schardin's problem is a shock wave travelling over a finite wedge. The configuration leads to complex features from shock reflection to shock-vortex interaction and emerging acoustic waves. We validated our implementation against the simulation and experimental results of Chang and Chang [14]. We use initial conditions of a Mach 1.34 shock wave in a 2840×2000 domain.
Figure 4.6 shows our simulation results compared to the shadowgraph of Chang and Chang [14] at time 28 µs. Figure 4.7 shows the density contours at the same time. We note the shock reflection from the wedge, the diffraction of the shock past the tip and the vortices forming there. Figure 4.8 shows the results at time 108 µs. The shock has travelled around the wedge and the top and bottom stems have travelled along the far edge of the wedge to pass through each other to interact with the vortices at the tips. The resulting wave configuration is captured well and matches results in literature. Figure 4.9 shows the results at a tip at time 107 µs. We note that our code is able to capture intricate details such as the features that emerge as waves interact with the vortices also seen in the implementation by Chang and Chang [14]. We capture the small vortices above the main one, the acoustic waves propagating radially from it and the emerging transmitted shock.

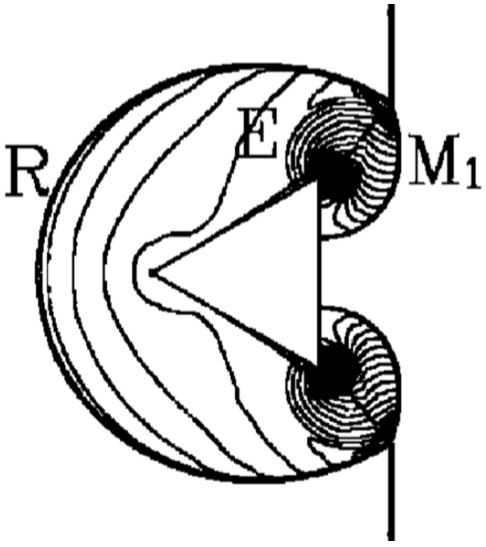


(a) Experimental shadowgraph from [14]

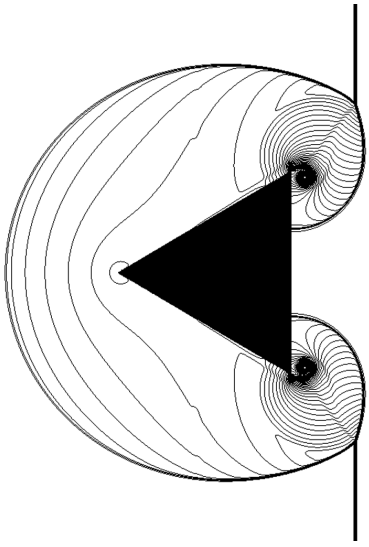


(b) Numerical Schlieren plot of our implementation

Fig. 4.6 Shadowgraph results of Schardin's problem at time 28 µs.

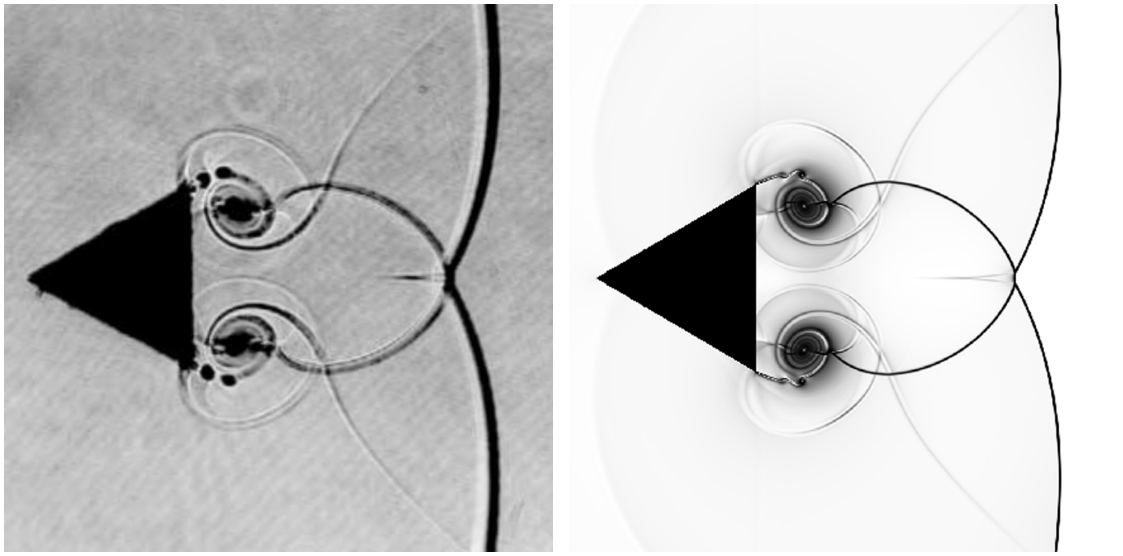


(a) Density contours from [14]



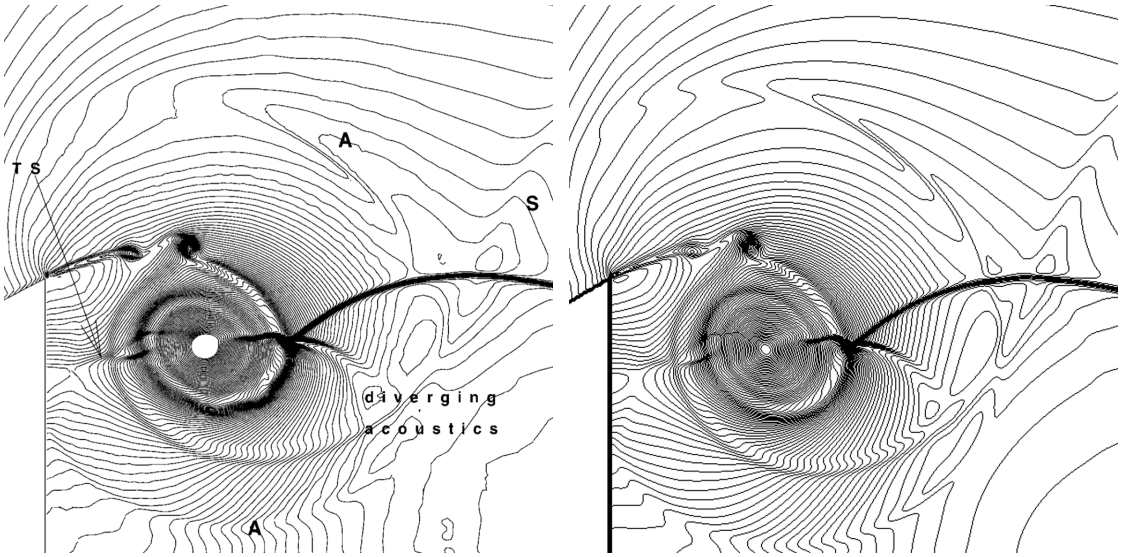
(b) Density contours from our implementation

Fig. 4.7 Density contours of Schardin's problem at time 28 µs.



(a) Experimental shadowgraph from [14] (b) Numerical Schlieren plot from our implementation

Fig. 4.8 Schardin's problem at time 108 µs.



(a) Density contours from [14] (b) Density contours from our implementation

Fig. 4.9 Schardin's problem at time 107 µs.

4.5.2 Shock wave passing around a cylinder

A 2D cylinder features a continuous change in surface normals which makes it a good test of robustness. We validated against the experimental results from Bryson and Gross [13]. The times of the original experimental images are not given and we match against images with comparable flow features. The simulation features a Mach 2.82 shock wave in a 3200×2400 domain. Figure 4.10 shows the experimental shadowgraph and the numerical Schlieren from our implementation. The results display a reflected shock (R.S), triple points (T.P.), Mach stems (M.S), contact discontinuities (C.D.) and vortices (V). The simulation matches the experimental results and we capture detailed features of a shock reflection around a smoothly changing geometry.

Figure 4.11 shows the results for a Mach 2.81 shock wave at a later time. A wake behind the cylinder has developed and the simulation displays a reflected shock (R.S), a triple point (T.P.), a Mach stems (M.S), a contact discontinuity (C.D.) and vortices (V). Our implementation matches the experimental results and captures the complex state of the wake.

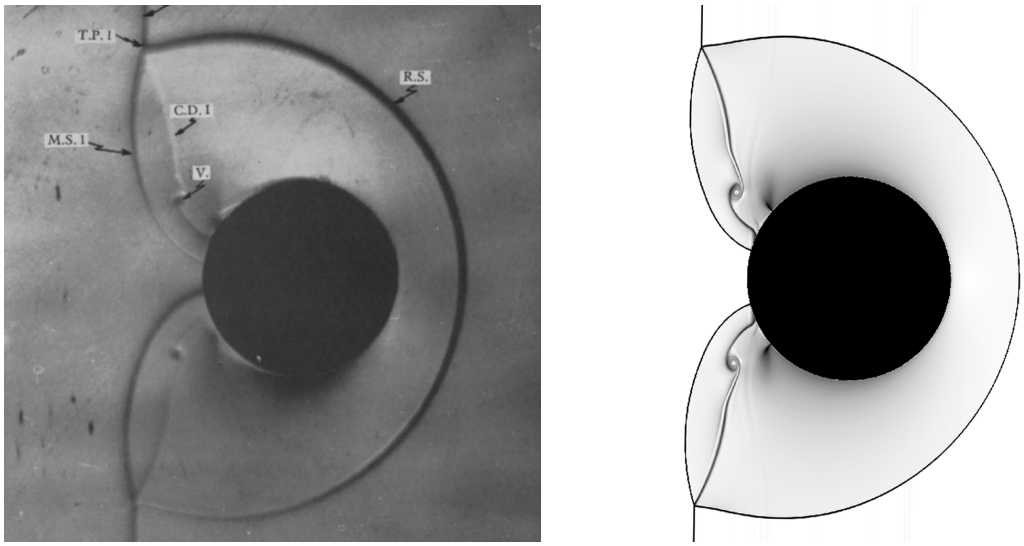
4.5.3 Flow around NACA 0012 aerofoil

To validate the 2D cut cell code quantitatively, a subsonic flow around NACA 0012 aerofoil was simulated. Once the simulation has run to steady state and all strong waves have dissipated, the pressure coefficient values along the surface of the aerofoil can be compared against experimental readings. The pressure coefficient C_p is a dimensionless value that expresses relative pressure in a system:

$$C_\mathrm{p} = \frac{p - p_\infty}{\frac{1}{2}\rho_\infty V_\infty^2}, \tag{4.19}$$

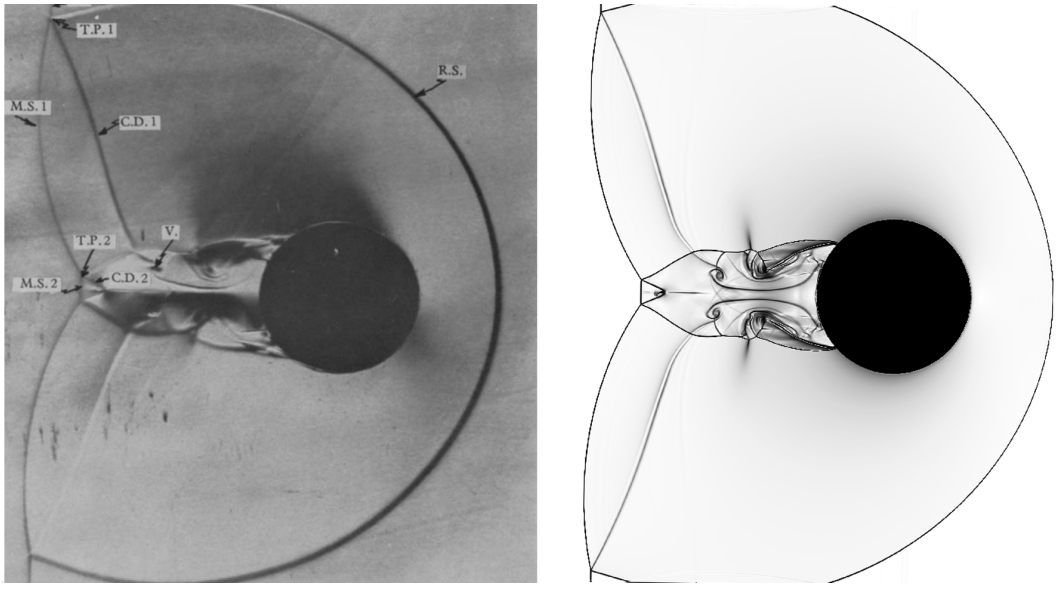
where p is the pressure reading at a location, p_∞ is the ambient pressure in the freestream, ρ_∞ is the ambient density in the freestream and V_∞ is the velocity in the freestream. A simulation was run with a Mach 0.4 flow, $p_\infty = 101\,325\,\mathrm{Pa}$, $\rho_\infty = 1.225\,\mathrm{kg\,m\,s^{-3}}$ and an aerofoil chord length spanning 6350 square cells. We compare against the results by Amick [1].

Figure 4.12 shows the results of the simulation compared with the experimental data. We note the smooth steady state of the system and the good match between the experimental and simulation pressure coefficients. The NACA test case demonstrates the code's ability to simulate subsonic flow around cut cell geometries with physical results comparable to experimental data.



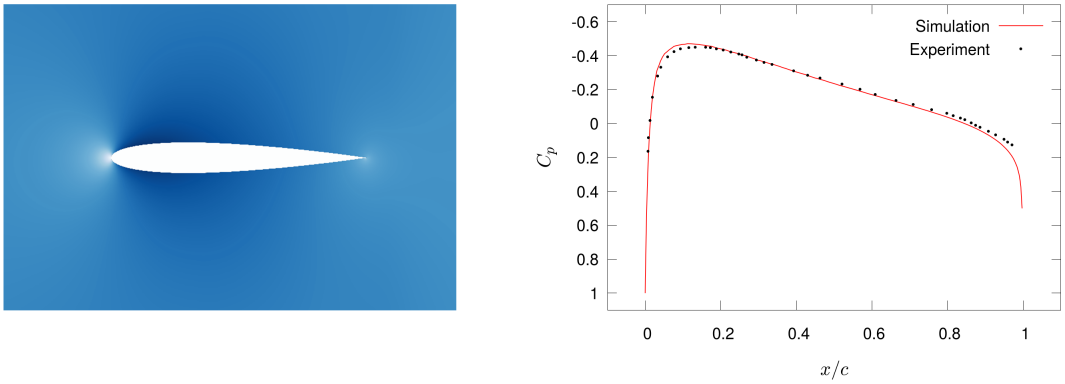
(a) Experimental shadowgraph from [13] (b) Numerical Schlieren plot from our implementation

Fig. 4.10 Mach 2.82 shock over a cylinder showing features such as the reflected shock (R.S), triple point (T.P.), Mach stem (M.S), contact discontinuity (C.D.) and vortices (V).



(a) Experimental shadowgraph from [13] (b) Numerical Schlieren plot from our implementation

Fig. 4.11 Mach 2.81 shock over a cylinder showing the reflected shock (R.S), triple point (T.P.), Mach shock (M.S), contact discontinuity (C.D.) and vortices (V).



(a) Pressure pseudocolour

(b) Pressure coefficient results

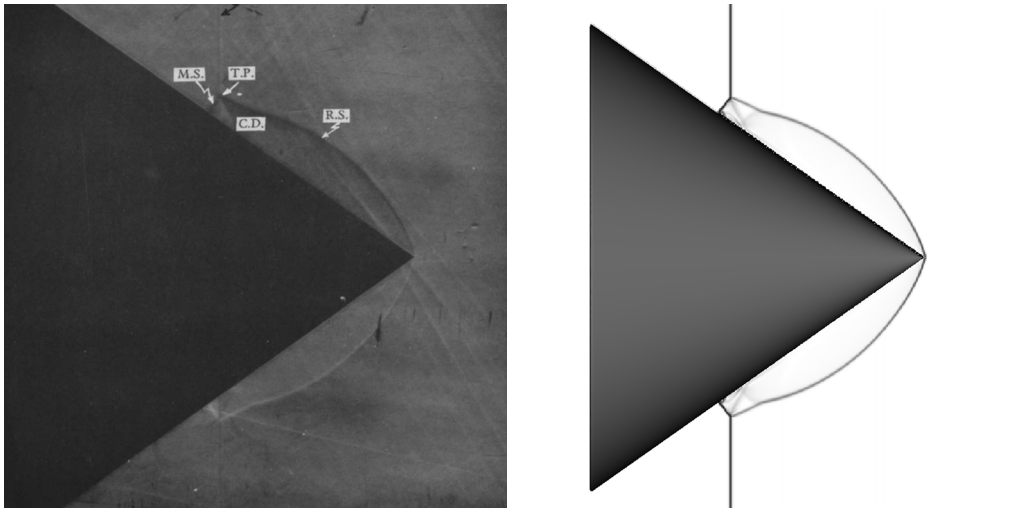
Fig. 4.12 Results of a Mach 0.4 ideal gas flow over a NACA 0012 aerofoil at $t = 0.015$ s with experimental [1] and simulation pressure coefficients C_p at x-coordinates along the chord length c of the aerofoil.

4.5.4 Shock reflection from a cone

For validation in 3D we tested against experimental results of shock wave interaction with a cone from Bryson and Gross [13]. A Mach 3.55 shock encounters a cone with semi-apex angle 35.1°. Figure 4.13 shows both the experimental shadowgraph and our implementation results. The code captures features like Mach stems (M.S.), triple points (T.P.), the reflected shock (R.S.) and contact discontinuities (C.D.). We note the agreement between the simulation and the experiment demonstrating that the code can accurately handle shock wave interaction with sharp features in three dimensions.

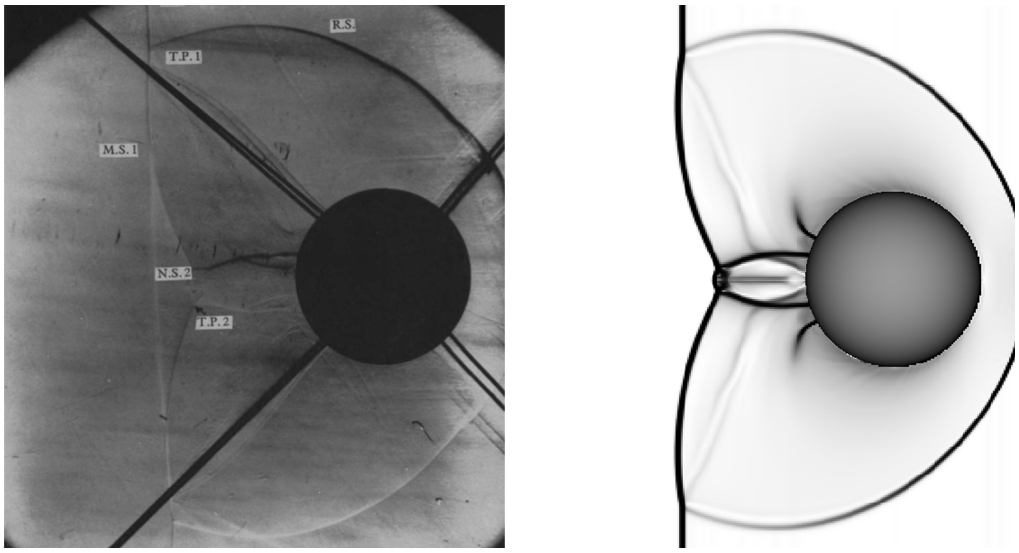
4.5.5 Shock reflection around a sphere

Analogously to the 2D case, we validated the 3D simulation using a sphere. Figure 4.14 shows the comparison between experimental results from Bryson and Gross [13] and our simulation of a Mach 2.89 shock wave travelling over a sphere. The test case demonstrates the ability to handle smoothly changing surfaces in three dimensions. We note capturing features like the triple point (T.P.), reflected shock (R.S.) and Mach stems (M.S.).



(a) Experimental shadowgraph from [13] (b) Numerical Schlieren plot from our implementation

Fig. 4.13 A Mach 3.55 shock over a cone with semi-apex angle 35.1° showing Mach stems (M.S.), triple points (T.P.), reflected shocks (R.S.) and contact discontinuities (C.D.).

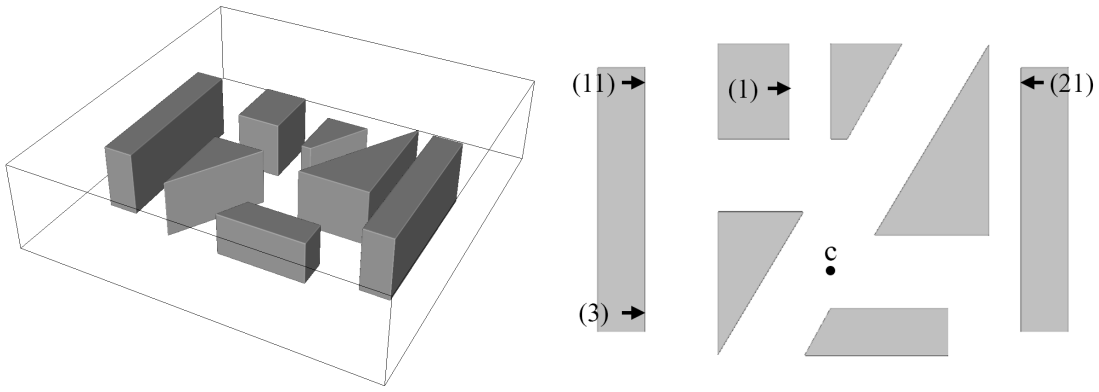


(a) Experimental shadowgraph from [13] (b) Numerical Schlieren plot from our implementation

Fig. 4.14 Mach 2.89 shock wave travelling over a sphere showing a triple point (T.P.), reflected shock (R.S.) and Mach stem (M.S.).

4.5.6 The Brittle test case

To validate the 3D implementation quantitatively, we simulate the detonation experiments by Brittle [12]. The study measures the overpressure from detonation in a complex street layout. The geometry features irregular buildings of different heights and orientations. The experiment modelled a 2000 kg truck bomb detonating in a 15 m street on a 1/50 scale. The buildings were concrete blocks and the explosion used the PE4 equivalent of 16 g of TNT. We modelled charge 1 at location $(0.9789, 0.3512, 0.04)$. Pressure gauges were placed at various locations to measure the effects of blast shielding and shock diffraction. The building layout together with the explosive and gauge locations are shown in figure 4.15.



(a) The 3D simulation domain

(b) Location of the blast charge (c) and gauges

Fig. 4.15 The Brittle test case features a street layout between irregular buildings (grey) of different heights.

Modelling detonation

To simulate the experiment, we model detonation as an energy source term. We assume that the energy release is proportional to the surface area of the sphere where the detonation occurs. Given the density ρ_d and mass m of an explosive, the volume of the charge is $V_0 = m\rho_\mathrm{d}$. We introduce energy from the ignition time t_0 until the end of detonation t_d. The end time can be found from the radius of the charge and the rate of burning u_CJ as $t_\mathrm{d} = r_\mathrm{d} u_\mathrm{CJ}$. Note that we include u_CJ when finding the fastest wave in the domain during detonation when calculating a stable time step. For the duration of the detonation, every time step we add energy to the cells inside the charge volume proportional to the energy density Q_d of the detonating material which is expressed as the product of density and mass specific energy $\hat{E}$ of an explosive. The amount of energy ΔE added every time step Δt is given by:

$$\Delta E = \Delta t \frac{Q_\mathrm{d}}{V_0} \frac{\mathrm{d}V}{\mathrm{d}t}, \tag{4.20}$$

where the expansion rate of the detonation sphere is given by

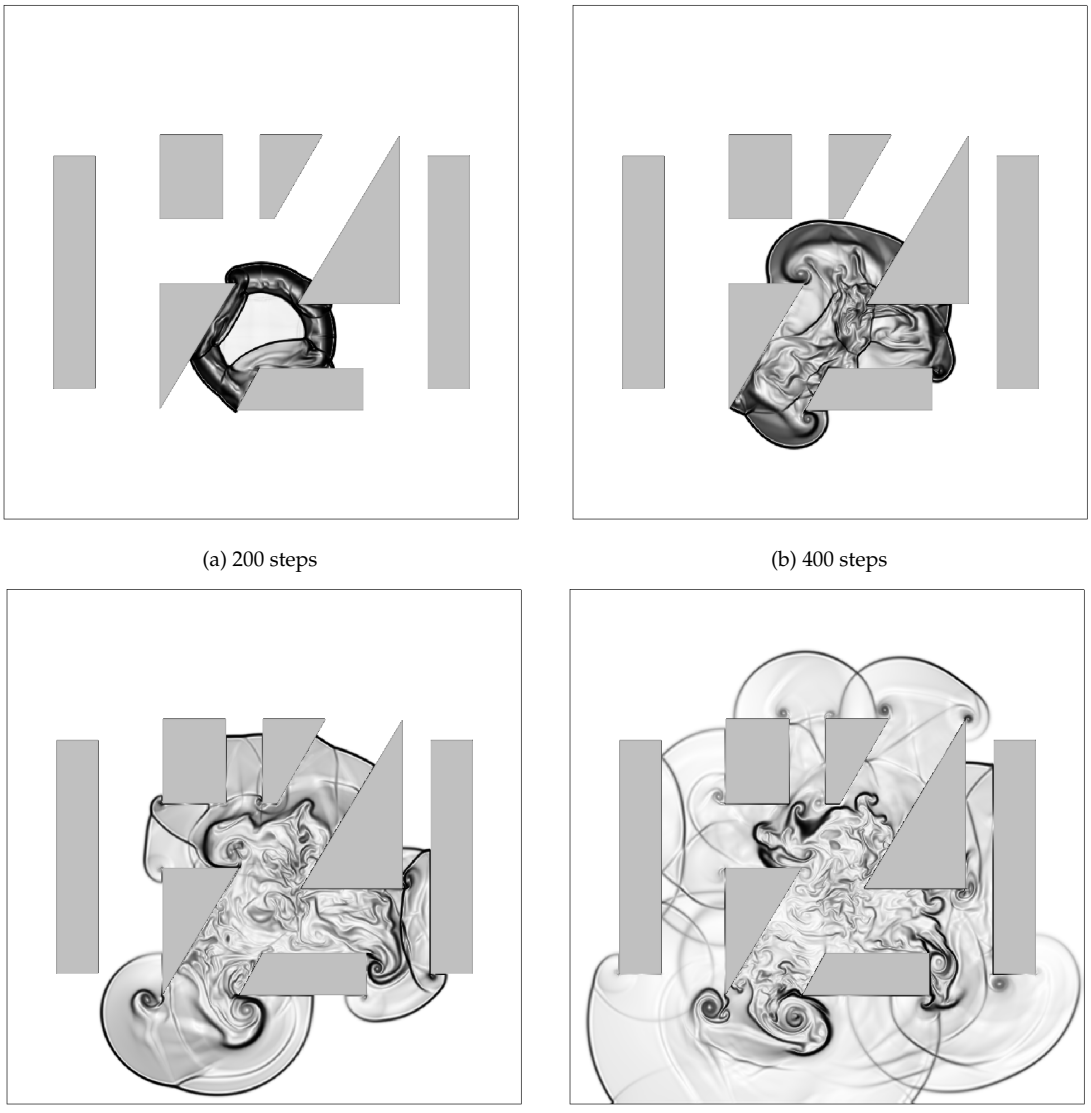
$$\frac{\mathrm{d}V}{\mathrm{d}t} = \begin{cases} 4\pi u_{\mathrm{CJ}}^3 t^2, & t < t_{\mathrm{d}}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.21}$$

The values of ρ_{d}, $\hat{E}$ and u_{CJ} depend on the explosive material. These are often found through experimental measurements and literature mentions several different values. We use the values given for TNT in Baker [5]: $\rho_{\mathrm{d}} = 1600\,\mathrm{kg\,m\,s^{-3}}$, $\hat{E} = 4520\,\mathrm{kJ\,kg^{-1}}$ and $u_{\mathrm{CJ}} = 6740\,\mathrm{m\,s^{-1}}$. The initial conditions set the density inside the charge volume to ρ_{d} and the ambient conditions to $\rho = 1.225\,\mathrm{kg\,m\,s^{-3}}$, $u = v = w = 0\,\mathrm{m\,s^{-1}}$ and $p = 101\,325\,\mathrm{Pa}$. The detonation occurs at $t = 0$ and the simulation was run to $t = 8\,\mathrm{ms}$. The domain extends $2.4\,\mathrm{m} \times 2.4\,\mathrm{m} \times 1.2\,\mathrm{m}$ with a resolution of $680 \times 680 \times 340$ on an Nvidia Tesla K20 card.

Results

Figure 4.16 shows the numerical Schlieren slice at $z = 0.04\,\mathrm{m}$ at various time steps. Figure 4.17 shows the simulation overpressure readings at gauges 1, 3, 11 and 21 compared against the experimental readings from Brittle [12]. We note that all four gauges have captured the overall trends in the pressure changes but with different timing and magnitude accuracy. We match gauge 1 best by capturing the timing and strength of the initial shock wave. The code captures subsequent drops and local pressure peaks. Though we match the trends in the pressure variations of gauges 3 and 11, the times of the initial peaks are earlier than in the experiment. There is also great variation in the accuracy of pressure peaks and the degree to which we capture later features.

Simulation readings from gauge 21 diverge most from experiment as the initial shock in the simulation arrives at the measuring point earlier and with greater strength. By the time the shock front reaches gauge 21 it will have encountered numerous cut cells along the faces of the buildings as well as sharp corners and other reflected waves. The mismatch here can be because of several reasons. Detonation events are violent and not matching experimental readings exactly can be due to using ideal gas equation of state for the explosive and modelling the explosion as a simple energy release, which is an approximation of the reactive processes that occur in ignition and detonation. We note that the same level of accuracy is shown in the literature such as the independent CPU implementation by Drazin [19]. Overall, we capture the features close to the source and while there is a loss of accuracy at distant and more shielded regions, the software can be used to analyse sophisticated scenarios with reasonable fidelity considering the approximations used.



(a) 200 steps

(b) 400 steps

(c) 800 steps

(d) 1200 steps

Fig. 4.16 Numerical Schlieren at $z = 0.04\,\mathrm{m}$ after various time steps.

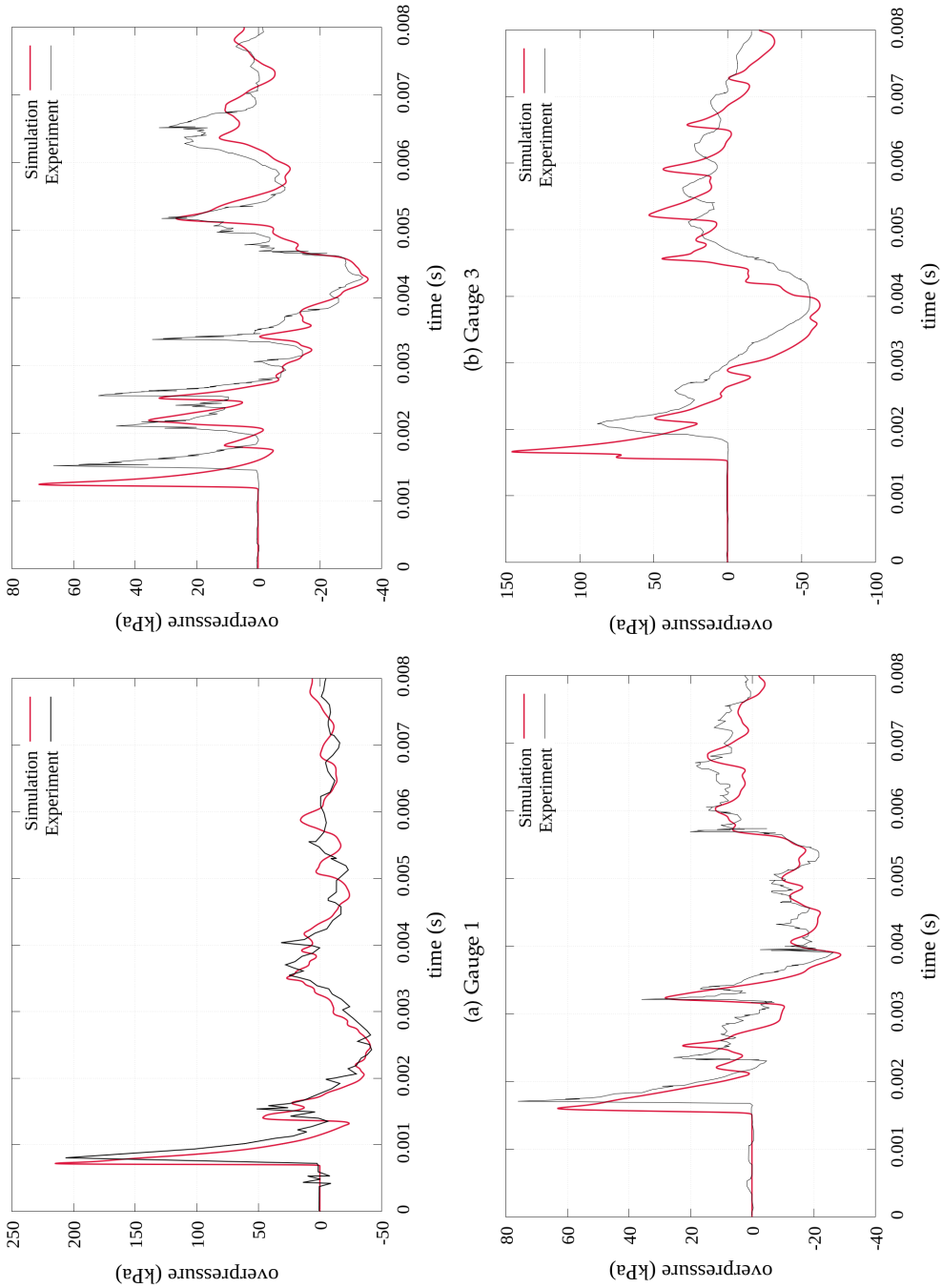


Fig. 4.17 Overpressures of gauges simulating explosive 1 using a resolution of $680 \times 680 \times 340$ and $\Delta x = 3.53 \times 10^{-3}$ m on an Nvidia Tesla K20 card.

4.5.7 Robustness of complex surfaces

To investigate the robustness of the cut cell code, subsonic flow and shock wave simulations were run in 3D using the DrivAer geometry. We test the stability of strong waves and regions of high density in domains with sharp corners and doubly shielded cells. Though not compared with any external results, these test cases demonstrate the numerical stability of the software and its ability to handle arbitrarily shaped objects in different flow regimes.

Flow around car body

The DrivAer car body was placed in a $100\,\mathrm{m\,s^{-1}}$ flow in the x-direction. The surface of the car is highly detailed and features thin regions of solid cells with sharp corners, many concavities and cells shielded from multiple sides. Figure 4.18 shows the density pseudocolour results of a simulation with resolution $600 \times 280 \times 160$ after 6096 time steps when the simulation has approached a stable state where any initial waves have dissipated and a wake is left behind the car. The simulation took 3603.5 seconds to run with 0.3% of that time spent on the fix at doubly shielded cells. A further 4.5 seconds was spent on geometry generation. A total of 135,787 cells were cut cells which constitutes 0.5% of the total cell count. There are a total of 15,130 doubly shielded cells or 11.1% of all mixed cells. We note the code is able to handle the intricate details of the car body and deal with complex cut cells.

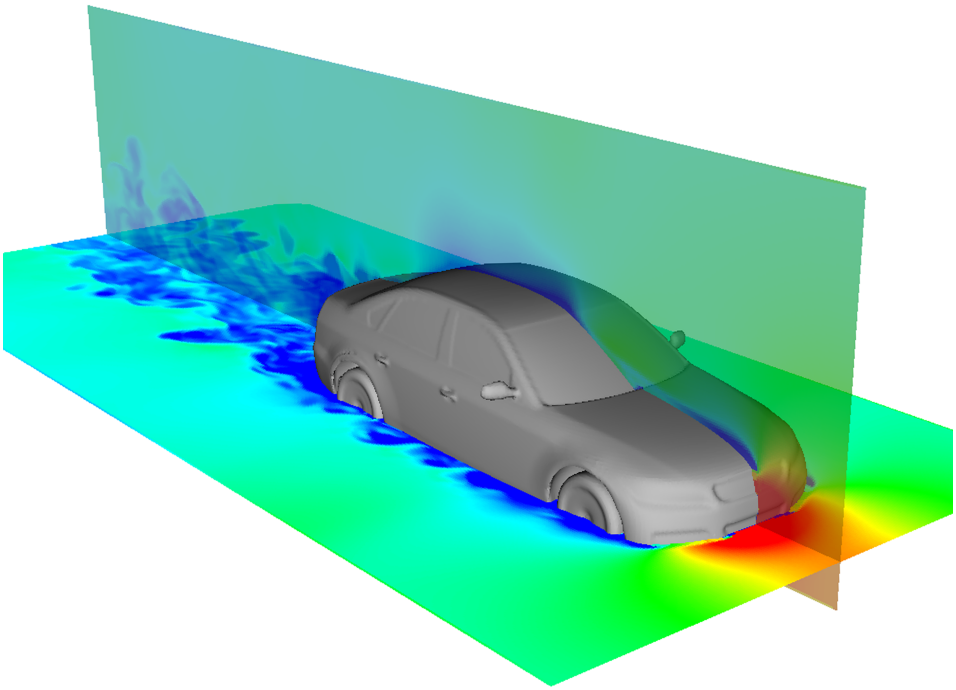


Fig. 4.18 Density pseudcolour results for a $100\,\mathrm{m\,s^{-1}}$ flow over the DrivAer geometry. The geometry features complex surface configurations with concavities, doubly shielded cells and smooth curvatures.

Shocks with concavities

To test stability with strong discontinuities, a Mach 2.0 shock wave was directed into the detailed region on the underside of the DrivAer geometry. There are several doubly shielded cells and reflections from smooth and sharp features. Figure 4.19 shows slices of density pseudocolour after 400 time steps at resolution $360 \times 288 \times 288$. There are $455,350$ cut cells making up 1.5% of the total. The $29,750$ doubly shielded cells make up 6.5% of all mixed cells. The simulation ran in 274.3 seconds of which 0.4% was spent on the redistribution at doubly shielded cells. An additional 3.7 seconds was spent on generating the geometry. The results demonstrate good numerical stability of simulating shock waves at arbitrarily complex surfaces with concavities.

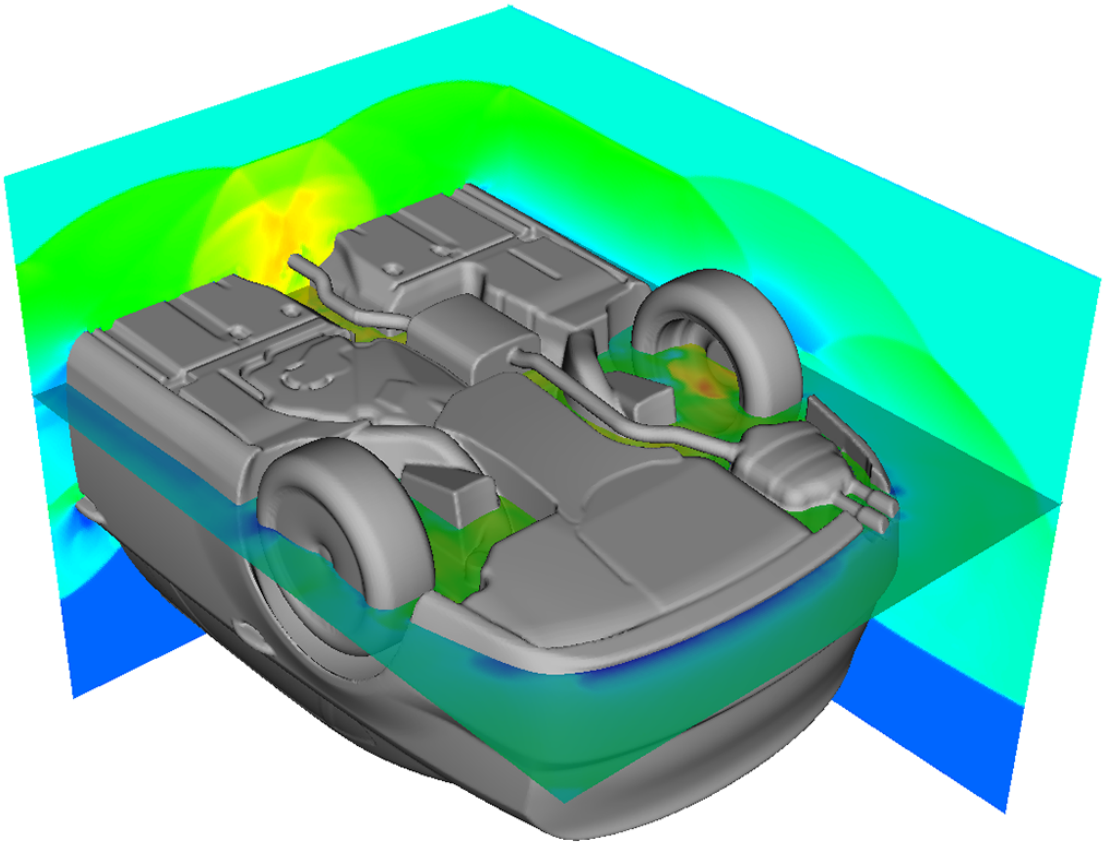


Fig. 4.19 Mach 2.0 shock reflected from the DrivAer geometry. The produced software can robustly handle high pressure at various surface features such as concavities and thin solid areas.

4.6 Performance

To demonstrate the performance impact of cut cell calculations, shock wave simulations were run at increasing resolutions. In 2D a Schardin's problem was simulated using a Mach 1.35 shock wave. Table 4.1 lists the times and number of steps it took to run the simulation to the same end time at three resolutions. We also list the number of cut cells for each run. The code scales better with increased cell counts but generally displays the expected behaviour showing that resources are made use of in an efficient way. The runtime increases 7.6 times at the first doubling of resolution and a further 7.9 times after the second which are close to the theoretical 8 times increase in workload as discussed in Chapter 3. Table 4.2 shows the proportions of time spent on different kernels at different resolutions. As the number of total cells increases, the portion of time spent on cut cell specific kernels decreases significantly. Even at the lowest resolution, the calculation of reference states, boundary fluxes and flux modifications takes up only 1.8% of the total runtime and this falls to 0.2% at four times the resolution. The increased time spent on transposes is caused by the need for a cell map as discussed above making it more expensive than the update kernel which now also includes cut cell specific calculations.

Resolution	Cut cells	Steps	Time (s)
1000×1000	924	1028	11.3
2000×2000	1854	2081	86.2
4000×4000	3704	4189	681.9

Table 4.1 2D Schardin's problem on a K20 card.

Kernel	1000×1000	2000×2000	4000×4000
Flux	58.5%	58.1%	57.3%
Transpose	18.8%	20.7%	21.7%
Update	16.2%	16.6%	16.9%
Time step	3.1%	3.2%	3.2%
Boundary	0.4%	0.2%	0.2%
Cut cell	1.8%	0.5%	0.2%

Table 4.2 Proportion of time spent on different kernels for Schardin's problem on a K20 card.

3D performance analysis was performed for the simulation of a shock wave interacting with a cone. Table 4.3 shows the number of steps and the time taken to simulate a Mach 2.0 shock wave travelling around a cone to a fixed final time. We also show the number of cut cells in the domain. The scaling is not as close to the theoretical 16 times increase in work as we see with the clear solver, ranging from 13.9 to 15.2 for cut cells with each resolution doubling. This is due to the larger effect of the map transpose in 3D. Table 4.4 lists the proportions of the total runtime taken by different kernels. We note the increased time spent on transposes which now includes both variable and map rearranging. The dominant kernel is still the flux calculation with cut cell modifications taking up only 0.5% to 2.4% of the runtime, decreasing with the increase of resolution.

Overall we note the low impact of the cut cell flux calculations when compared to the clear solver. Additional work is limited to mixed cells and, even in high resolution domains, this results in fast kernel execution. The memory overhead from storing cut cell data is low when compared to flow variables, even for complex surfaces. The largest increase in memory footprint and workload when compared to the clear solver is the map and its transpose space. Raising the resolution increases solver work significantly faster than flux modification work and the code shows good scaling with respect to the number of cells in 2D, although the efficiency decreases slightly in 3D. Overall, the simulation time is mainly dictated by flux calculations, updates and data transposes.

Resolution	Cut cells	Steps	Time (s)
$85 \times 85 \times 85$	2890	396	6.6
$170 \times 170 \times 170$	12124	815	91.9
$340 \times 340 \times 340$	48200	1688	1404.1

Table 4.3 3D shock over a cone on a K20 card.

Kernel	$85 \times 85 \times 85$	$170 \times 170 \times 170$	$340 \times 340 \times 340$
Flux	46.7%	49.8%	50.5%
Transpose	28.5%	28.7%	29.3%
Update	14.0%	15.1%	15.8%
Time step	1.7%	1.8%	1.9%
Boundary	5.7%	3.3%	1.8%
Cut cell	2.4%	0.9%	0.5%

Table 4.4 Proportion of time spent on different kernels for 3D shock over a cone on a K20 card.

4.7 Conclusion

We have presented a cut cell extension to the split Euler solver discussed in Chapter 3. We have described the Klein-Bates-Nikiforakis cut cell method and its extension by Gokhale et al. The sparse data needed for the method can be stored in contiguous memory with explicit addressing to allow for higher resolution simulations on limited hardware. Qualitative comparisons with experimental and numerical results demonstrate the code's ability to capture complex flow features in shock wave simulations featuring different boundary shapes. Quantative validation compared the pressure coefficient values along an aerofoil in a subsonic flow against experimental results. We also ran the Brittle detonation test case and compared the overpressure values at various locations and our implementation is capable of reproducing the trends in both shielded and unshielded areas. Complex shapes can be handled in different flow regimes and the software shows good numerical stability in the presence of sharp corners and concavities. The overhead from cut cells is limited and does not significantly impact the performance of the software. Although the need for a cut cell map increases the workload of the software, the overall runtime is mainly dictated by the flux calculations and variable updates. In the next chapter we will discuss multi-card parallelisation of our software necessary at higher resolutions.

Chapter 5

Multi-Card Parallelisation

Graphics cards have limited global memory and higher resolution simulations require multiple GPUs. The cost of memory transfers between cards can become a bottleneck for performance conscious applications. The communication time scales with the amount of data sent and the sending frequency but is also affected by the hardware connecting GPUs. Domains can be subdivided in different ways that either reduce the size of messages or ensure that sent data lies in contiguous sections in memory. In this chapter we will take a look at the existing literature on using multiple GPUs and present the design and performance of our approach. We will describe the Wilkes2 GPU cluster used for several scaling runs and analyse the results of our implementation.

5.1 Using multiple GPUs

Although the sparse storage of cut cell variables, states and fluxes offers memory reductions, a single GPU has relatively little global memory when considering the resolutions modern CFD simulations are run with. An Nvidia Tesla K20 [43] has approximately 5 GB of global memory and an Nvidia Tesla P100 [44] has around 16 GB. The theoretical maximum resolutions of our software for a simulation with no cut cells is $\sim 7900^2$ cells in 2D and $\sim 390^3$ cells for 3D for a K20 card and $\sim 14\,100^2$ in 2D and $\sim 580^3$ in 3D for a P100 card. In practice, a fraction of the on-card memory is reserved, needed for CUDA calls or used by libraries. The maximum resolutions are lowered when using cut cells or additional physical variables.

Larger simulations can be run by utilising many GPUs at the same time. By using the Message Passing Interface (MPI) communication framework [38] we can divide a large computational domain across multiple graphics cards and keep the flow across the domain consistent by

communicating ghost cell data. MPI is a communication standard used in parallel and distributed computing which allows several processes to exchange variable size data through send and receive instructions.

When utilising numerical methods and solving Riemann problems, each GPU behaves like a stand-alone processor in a single-card setup. However, some boundaries of the domains correspond to the limits of neighbouring GPUs. The ghost cells at these boundaries get their data from the domain cells of an adjacent card by message-passing. Additionally, the stable time step is found using the maximum wave speeds of all cards and is broadcast to all segments to keep the simulations consistent.

To obtain good scaling, every GPU should have the same amount of work to do while minimising the time it takes to communicate between cards. The best way to segment a domain is not immediately obvious. Though different solver branching may lead to slightly varying amounts of work in different cells, the number of cells in the domain determines most of the workload. Even when aiming to have the same number of cells on each GPU, it is not clear how the simulation space should be divided. A 3D domain can be equally segmented along any of its dimensions or a combination of them. There is a wealth of literature on multi-card utilisation for stencil operations and specifically fluid simulations.

5.2 Background

Thibault and Senocak [60] describe a multi-GPU implementation of a Navier-Stokes solver. They discuss how adding more cards to simple simulations can lead to poorer performance due to the high overhead of copying data between a GPU device and the host system, together with inter-host communication in distributed systems. They conclude that for best results, the GPUs should have sufficiently large subproblems to justify the communication between them. As we discuss below, the hardware used to connect GPUs to each other, to the host and across nodes not only affects transfer speeds but also dictates domain segmentation and the choice of communication framework.

Lefebre et al. [34] present a single precision Euler code with simple immersed boundaries that uses the MUSCL scheme to obtain second order accuracy. While they have different data layout than our code, many of their observations are relevant to graphics card development in general. Their 2D GPU solver obtains 60 times speedup compared to a single core CPU implementation. They also report a 10 times speedup in 3D and present a multi-GPU implementation which obtains good scaling with 2 cards. They show that the scaling is dependent on the problem size and demonstrate the benefits of page-locked memory allocation on the host. Memory that has been allocated using `malloc()` on the host is pageable and the system can swap away parts of

it as it optimises working with other data. A transfer between the host and a device requires the whole data to be pinned, however. When calling `cudaMemcpy()` a host-side copy is first initiated which stores the data in a page-locked buffer and only then copies it to the device. This extra copy can be avoided when allocating data in page-locked memory from the start using `cudaMallocHost()`. Using too much pinned memory can interfere with the optimisation methods of the system leading to poorer host performance. Lefebre et al. show a three times speedup between pinned and pageable memory copies. Although the initial storage is slower and may impact the host performance, the strategy can be useful when dealing with frequent smaller memory copies for inter-GPU communication.

Micikevicius [37] describes dividing a stencil-based computation domain between multiple GPUs. They discuss the advantages of ghost cells lying in contiguous sections. A division along a single dimension allows all MPI communication regions to be in consecutive memory addresses. They show further performance gains achieved from overlapping MPI communication with computation by dividing the solution of the domain into two phases – first solving the region corresponding to the ghost cells of the neighbouring card and then the rest of the domain while simultaneously communicating the ghost cell data. They show linear scaling for 2 and 4 GPUs and even super-linear speedup for some problem sizes. The implications of one-dimensional subdivision and coalesced buffer regions is taken into consideration in our multi-card implementation discussed below. We also employ the two-phase solver design to overlap MPI communication with CUDA computation.

The two-phase approach is popular in literature. Bernaschi et al. [8] present a multi-GPU implementation for the 3D Heisenberg spin glass model. They discuss separating boundary and core cell solvers, overlapping MPI communication with numerical work and test several MPI libraries to show good parallel efficiency for up to 8 cards. GPU-aware implementations show better performance but seem to interfere with CUDA streams used for overlapping. We will show similar comparisons between regular and GPU-aware library calls.

Xian and Takayuki [66] present an analysis of 1D, 2D and 3D domain decomposition when simulating flow around a sphere using a Lattice Boltzmann method on a multi-GPU system. For a 96 GPU setup they show a 3-4 times higher performance for the 3D partitioning compared to dividing in just one dimension. The best performing subdivision depends on the number of cards used, however, and they recommend a 1D approach for fewer than 10 cards, a 2D partitioning for up to 50 cards and a 3D domain decomposition when using more than 50 cards.They note the benefits of pinned host memory for MPI buffers and the downside of overusing page-locked memory. Their implementation also uses the two-phase communication masking technique, obtaining performance gains of 1.08 times for even a 96 GPU configuration when compared to a single-phase approach. Higher performance increases are obtained with lower card counts as each GPU has more work to do while MPI transfers are taking place.

Okamoto et al. [46] describe a multi-card implementation using MPI for simulating seismic wave propagation. They discuss how a 3D domain can be subdivided in several ways. A division along a single dimension reduces the problem size for each GPU while the communication size remains constant. Eventually there is a GPU-count for every static problem size where the MPI calls start to dominate the simulation time as the computation per card becomes very small. Okomoto et al. divide the domain in all directions as the communication time is proportional to the surface area of a subdomain which decreases with the size of the region. They keep their ghost cells in dedicated contiguous sections of memory independent of the layout of the interior cells.

Jacobsen et al. [25] discuss simulating incompressible flow using 128 cards on a GPU cluster showing 130 times speedup compared to two quad-core CPUs. They describe overlapping intra- and inter-node communication as well as employing the two-phase strategy of Micikevicius to accelerate CFD simulations. Their implementation divides the domain in a single dimension. While this again leads to larger data transfers it has the benefit of contiguous memory layout on the device. Jacobsen et al. discuss how 2D and 3D divisions have to use special memory layout which may affect the performance of the solver or rearrange ghost data to contiguous memory which adds overhead.

Schneible et al. [54] present a multi-card implementation designed to minimise communication time for stencil based applications. They report a 25% increase in the performance of a lattice quantum chromodynamics kernel by overlapping computation and communication. They discuss how the bandwidth between GPUs is not equal and minimising the surface area of a subdomain does not necessarily reduce communication time. Schneible et al. also describe the asymmetry between intra- and inter-node communication speeds and its implications for domain segmentation. They discuss how the optimal configuration depends on the hardware and the simulated problem. A brute force approach would have to benchmark $\mathcal{O}(N!)$ splitting scenarios where N is the number of GPUs. Schneible et al. outline a model which predicts the communication times of different domain segmentations based on target hardware parameters and demonstrate good agreement with benchmark tests. A similar model for our implementation is beyond the scope of the current work but the observation that reducing inter-node communication is often better than minimising subdomain surface area is taken into consideration for the implementation described below.

From the existing literature it is clear that without bespoke simulation parameter analysis the best approach for scalability is to segment a domain along a single direction. This will have the benefit of reducing inter-node communication while keeping the ghost-cells in contiguous sections of memory without adjustment to the data structures used in our solver. To mask the communication time between nodes, it is advantageous to structure our pipeline as a two-phase approach that overlaps ghost cell transfers with the domain cell solver. We move on to discuss the design of a multi-card extension to the cut-cell solver. We will describe the hardware used for several scaling runs and analyse the performance of different communication strategies.

5.3 Domain segmentation and communication

At the start of each time step our state data is ordered in the x-direction. In 2D y-direction ghost cells are contiguous, and in 3D z-direction ghost cells are in adjacent memory. We therefore split two-dimensional simulations along the y-axis and three-dimensional domains along the z-axis. The ghost cells at the lower boundary of process n then corresponds to domain cells of process $n-1$ and the upper boundary overlaps with process $n+1$ as shown in figure 5.1. For 3D simulations, if we solve these MPI ghost cells in the x- and y-sweeps, we only need to communicate them once at the start of the time step to keep the propagation of information consistent across the whole simulation.

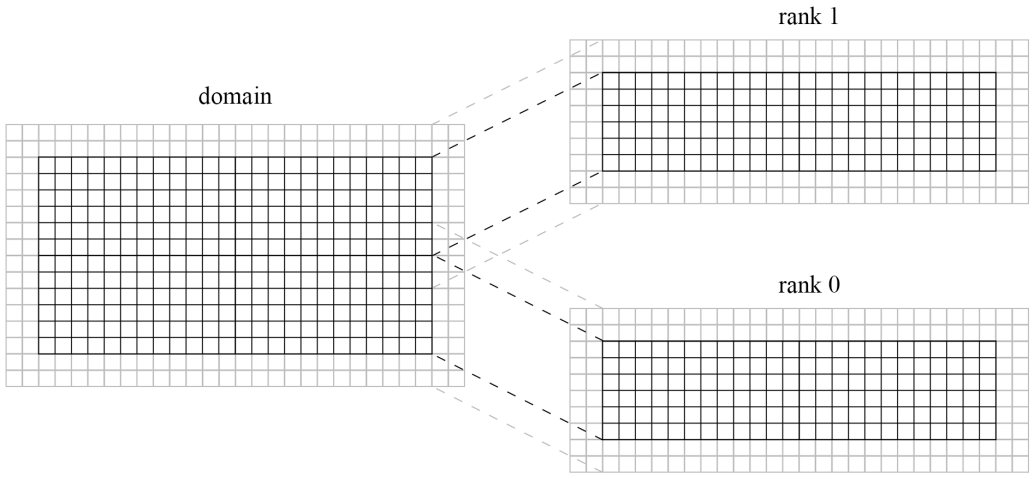


Fig. 5.1 Splitting a 2D domain between two cards. The core cells are divided exclusively but the ghost cells of the two processes overlap domain cells of their neighbours. External ghost cells are set using boundary conditions whereas the MPI ghost cells are populated by copying data from the domain cells of adjacent ranks.

The code is adjusted to include an MPI communication phase at the start of every time step. Each GPU sends local state data to the ghost cells of neighbouring cards while receiving data for its own boundary cells. There are several ways to order the calculations and communication when using multiple cards. The serial way is to first communicate all relevant ghost cell data to neighbours and then solve local subdomains according to the updated information. Following the work of Micikevicius [37] additional parallelism can be achieved by splitting every local domain into three parts: two communication bands and a core area as shown in figure 5.2. The width of the communication sections depend on the stencil size and we use layers of 4 cells at either end of a subdomain. The core section can be solved independently of neighbouring processes in the non-split directions while the communication operations populate ghost cells.

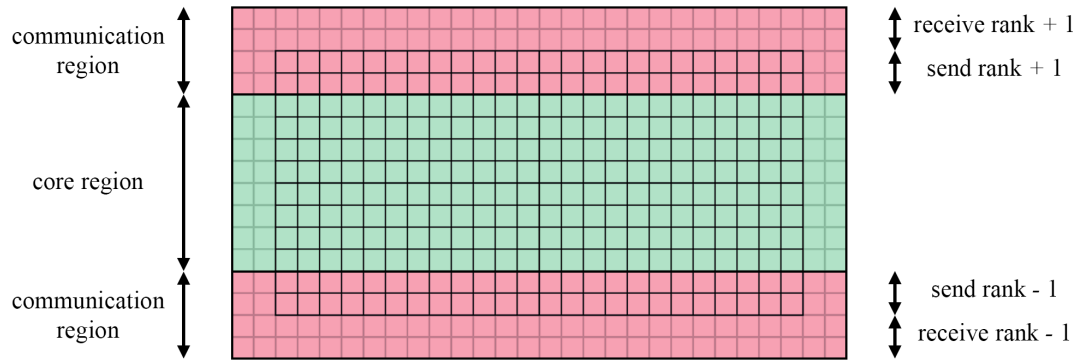


Fig. 5.2 A local subdomain is divided into three regions. The core cells can be solved independently from MPI communication of the ghost region data.

We split the solve into two phases: an MPI phase and a core solve. CUDA kernel calls are asynchronous with respect to the host and return almost immediately after being invoked. The kernels can run while work is done on the CPU and also in parallel with any work on the same GPU in another stream. Memory transfers can also be asynchronous when using non-overlapping addresses and pinned memory on the host. Every time step, we start by launching the kernels to solve the core section of a domain in a specific stream. In the naïve approach we then collect the data which is sent to neighbouring cards into a contiguous GPU buffer, copy the buffer into a CPU buffer using another stream and call MPI send-receive to send local data while receiving ghost cell data. For sufficiently big problems, the core solve takes longer than the MPI calls and we mask much of the communication time. Once the data has been swapped, we can solve the communication regions. As this does not depend on the core section, we can do this in a stream parallel to the original one (figure 5.3).

We use the Open MPI [47] library which supports GPU applications. It has GPU-aware functionality and can distinguish between host and device memory spaces from the pointers passed to method calls. The MPI routines are then able to communicate between GPUs without using the host memory. This avoids several copies and offers a more streamlined programming interface. To further improve communication performance we use non-blocking MPI calls and explicit synchronisation. We make use of this CUDA aware behaviour and compare its performance with the naïve approach. Open MPI also features GPUDirect technology [41]. Initial versions of GPUDirect allowed inter-node messages to avoid an additional copy in the host memory. The current version features many optimisations for inter- and intra-node GPU communication. We assess the suitability of explicit GPUDirect implementation in the performance section below.

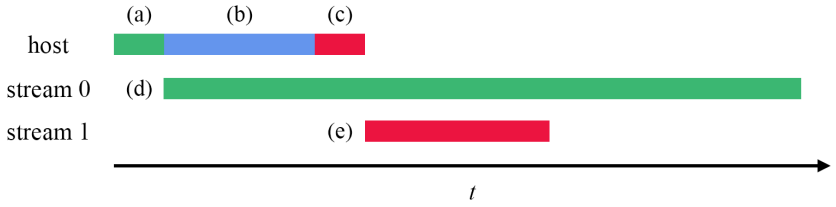
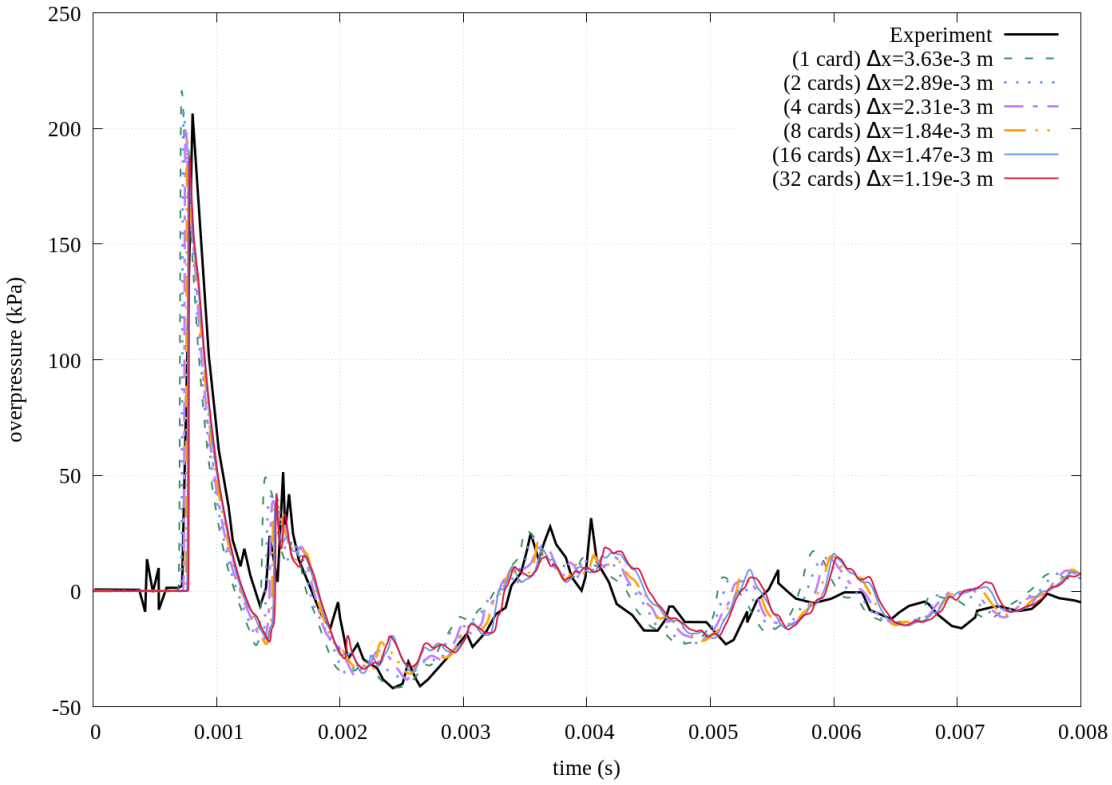


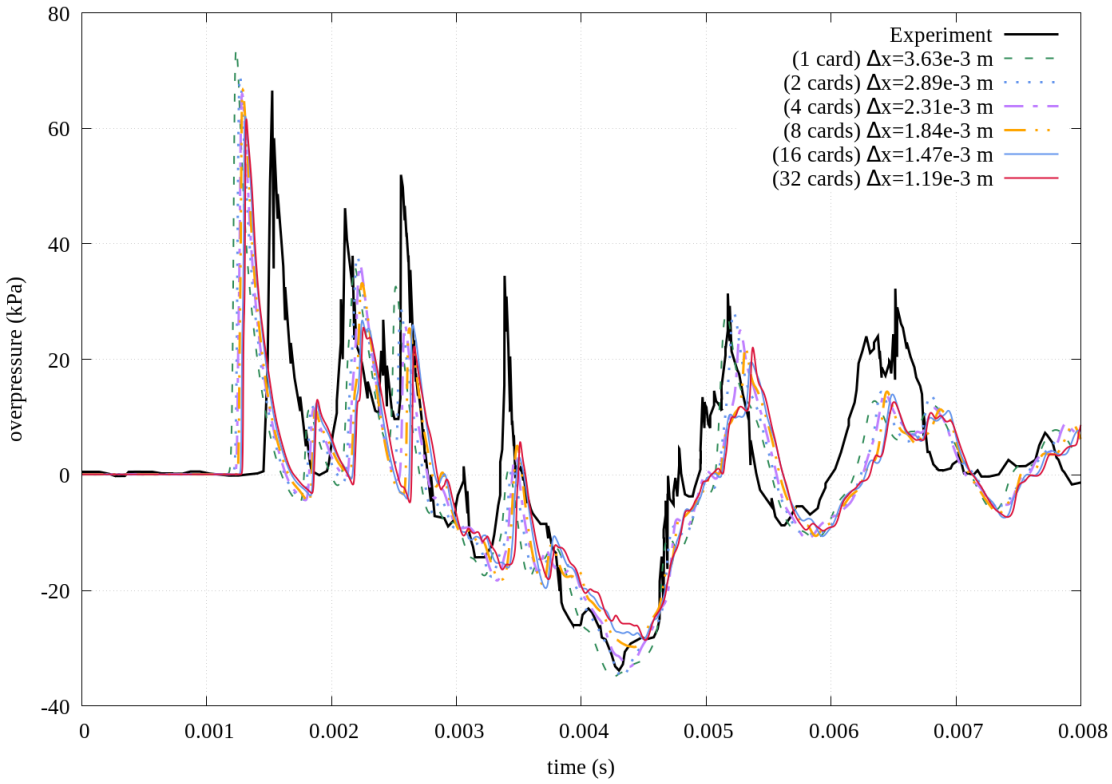
Fig. 5.3 Overlapping communication and solver. (a) launch core region solver. (b) MPI communication to swap ghost region data. (c) launch ghost region solver. The two kernels (d) and (e) can execute in parallel to each other and host operations.

5.3.1 Validation

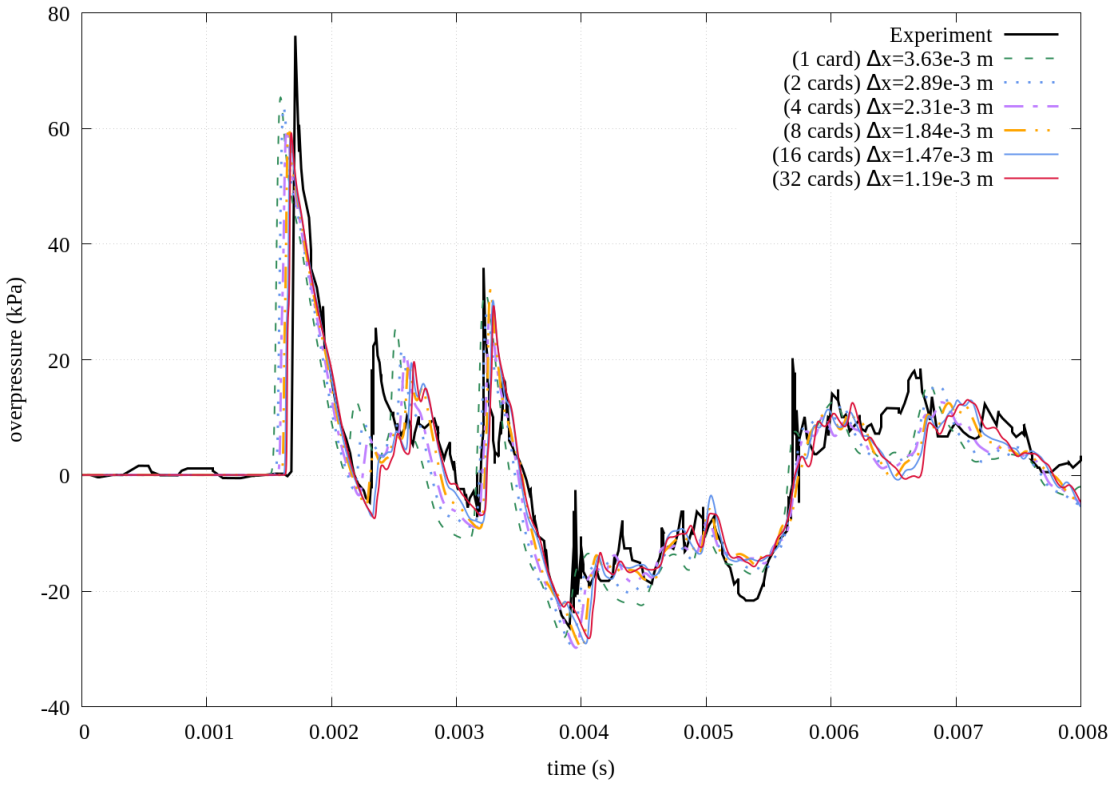
To validate our domain segmentation implementation, we simulated the Brittle test case discussed in Chapter 4. The domain is split along the z-axis between a different number of cards. Six simulations were run with various resolutions filling different Nvidia Tesla P100 card configurations completely. Figure 5.4 shows the results compared against a single card run. We note that the splitting leads to correct results and that the increase in resolution, which is possible with multiple cards, improves the accuracy of the simulation. When using 32 cards, the initial pressure peak arrives closer to the experimental time. The results show the ability of our implementation to utilise multiple GPUs by splitting the simulation domain between several processors and communicating ghost cell data. We can utilise multiple graphics cards to run high resolution cut cell simulations to better describe the pressure tendencies of complex detonation simulations in 3D. In the following section we will look at the performance and scaling of overlapping communication with computation using the Wilkes2 GPU cluster.



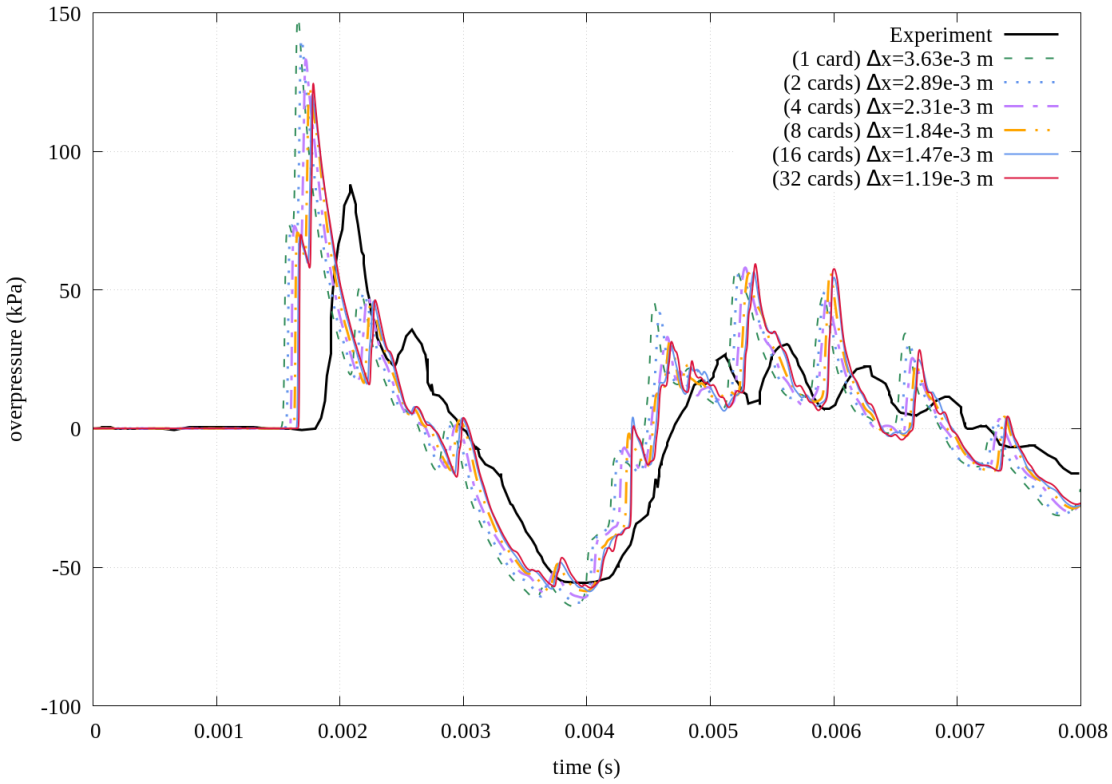
(a) Gauge 1



(b) Gauge 3



(c) Gauge 11



(d) Gauge 21

Fig. 5.4 Overpressures of gauges simulating explosive 1 using several P100 cards with different resolutions showing the effects of cell size on matching experimental data.

5.4 The Wilkes2 cluster

The Wilkes2 cluster [63] is part of the CSD3 high-performance computing platform at the University of Cambridge. It has 360 Nvidia Tesla P100 GPUs on 90 Dell EMC server nodes connected with Mellanox EDR Infiniband. Figure 5.5 shows the layout of the nodes on the cluster. Each node has 4 cards mounted on PCIe bus, 96 GB of memory and a 12-core Broadwell processors. The P100 cards are more powerful than the K20 GPUs. Their peak double precision performance is 4.7 TFLOPS compared to 1.17 TFLOPS on the K20. The P100 has 3584 cores and 16 GB of global memory while the K20 has 2496 cores and 5 GB of memory. The four cards on a node are capable of direct communication through their shared PCIe bus. Data transfer between the cards is possible without passing through the CPU memory buffers leading to potentially better scaling of multi-card implementations. When using more than four GPUs at a time, several nodes must be used and data is sent to other nodes using slower Infiniband connection which provides a one-way bandwidth of 12.5 GB/s.

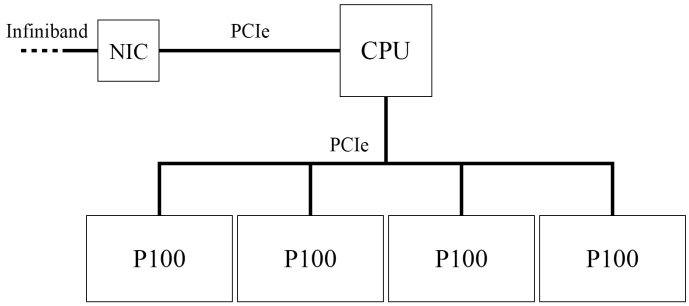
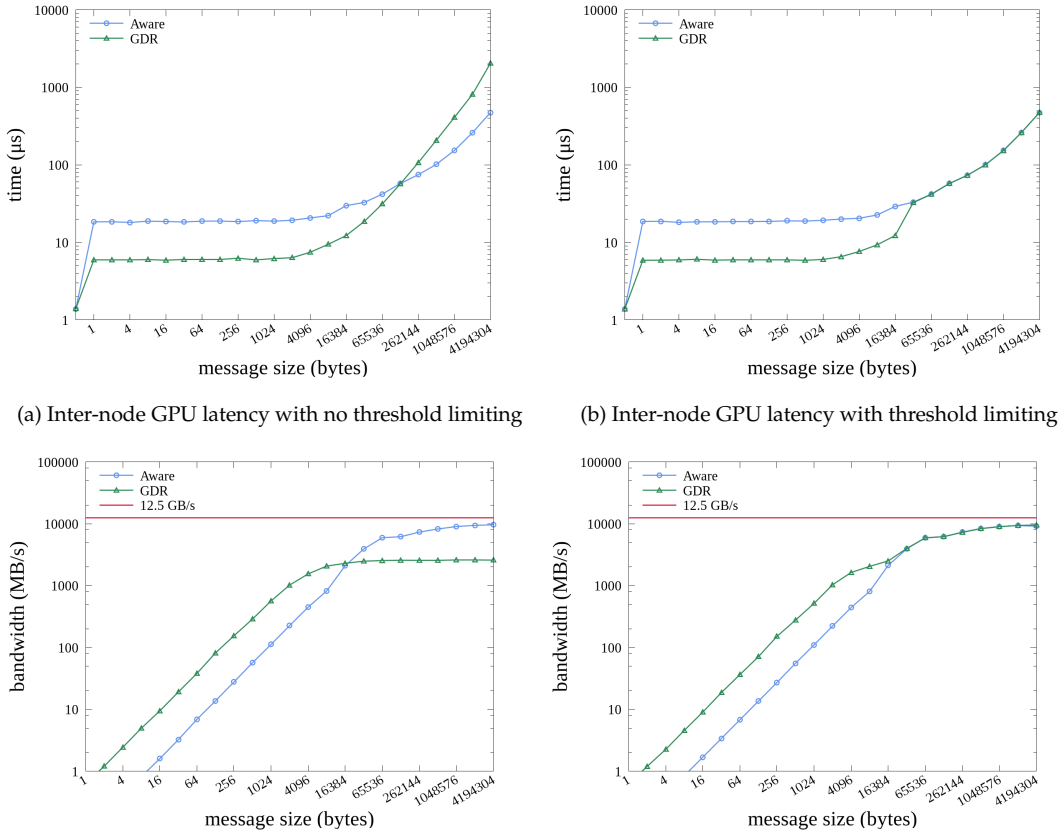


Fig. 5.5 Each Wilkes2 cluster node has four Tesla P100 cards connected with a common bus. The cards are controlled by a CPU connected via PCIe. The nodes are connected through Infiniband using a network interface controller (NIC).

5.4.1 OSU micro-benchmarks

To assess the performance potential of the Wilkes2 cluster and Open MPI we ran OSU micro-benchmarks [39]. These benchmarks from Ohio State University can be used to measure the performance of MPI implementations. We test the latency and bandwidth between cards on different nodes and compare the results of CUDA aware MPI calls with and without GPUDirect. The latency test uses multiple blocking send and receive calls between two processors and records the average one-way delivery time for various message sizes. The bandwidth test calculates the maximum data rate by using multiple non-blocking send and receive calls between two processes. For inter-node communication, CUDA aware uses host staging of messages while GPUDirect bypasses the CPU to directly communicate with the target GPU on another node. Figure 5.6 shows the latency and bandwidth results for various message sizes. While GPUDirect offers better performance at lower message sizes, host staging outperforms it at larger ones due to hardware limitations. Open MPI defines a flag `--mca btl_openib_cuda_rdma_limit`

which allows us to set a message size threshold past which GPUDirect calls are staged on the host. The default value is 30 kB which is around where CPU staging starts to outperform GPUDirect. Using the flag, GPUDirect implementation can offer the best possible performance at all message sizes. The hardware limitation and Open MPI threshold value mean that there is no performance gain from GPUDirect at messages larger than 30 kB. The ghost cell information for simulations that require multiple GPUs to run is often orders of magnitude larger than this limit. For example, the smallest message size we use to communicate a single variable in the ghost region of our lowest resolution scaling run discussed below is $429 \times 429 \times 2 \times 64$ bits or $\sim$3 MB. It is therefore unlikely that our implementation can benefit from GPUDirect when using multiple nodes on Wilkes2.



(a) Inter-node GPU latency with no threshold limiting

(b) Inter-node GPU latency with threshold limiting

(c) Inter-node GPU bandwidth with no threshold limiting

(d) Inter-node GPU bandwidth with threshold limiting

Fig. 5.6 OSU micro-benchmark test results on P100 cards on Wilkes2. CUDA aware MPI calls use host staging for messages while GPUDirect (GDR) bypasses the CPU. GDR performance is better for lower message sizes but deteriorates for larger messages due to hardware limitations. Open MPI defines a threshold value for message sizes at which GDR commands are staged through the host. The default value is 30 kB which can be seen in graphs (b) and (d). The Infiniband EDR interface has a peak bandwidth of 12.5 GB/s which the benchmark runs approach at higher message sizes.

5.5 Multi-card performance profiling

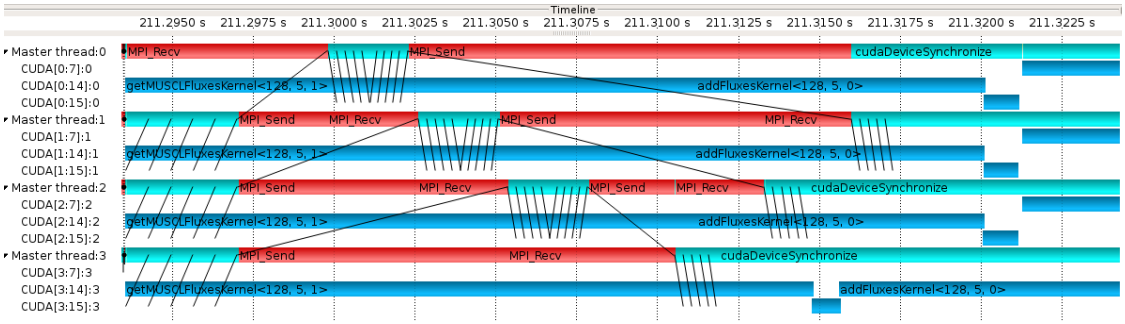
The performance of the multi-card software was analysed on the Wilkes2 cluster. We present the behaviour of the two different MPI approaches, the performance of static size problems across multiple cards and the scaling of the clear and cut cell code.

5.5.1 Communication analysis

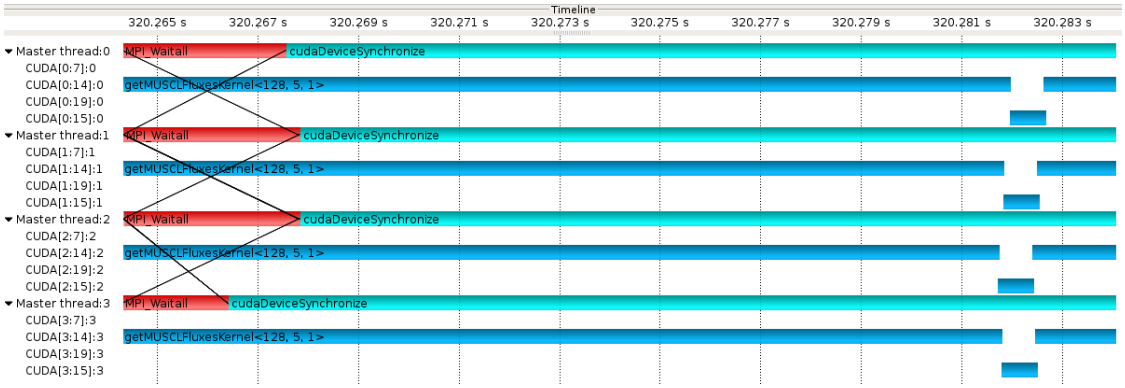
The software was profiled using the Score-P measurement infrastructure [30] and the resulting information was visualised using Vampir [29]. Figure 5.7 shows the Vampir timelines of a 3D bubble collapse on four P100 cards using naïve and CUDA aware communication. The figure focuses on the start of a time step and shows four main threads mapped to cards on a single node and the concurrent CUDA streams on each card. We note the overlap of communication (shown in red) with the core cell solves (in blue). Black lines in figure 5.7a show the collection of the five physical variables in ghost cells, their communication between cards and eventual copy to the target GPU. Although the communication is fully masked by the solver, it takes up almost the entire x-solve time. Figure 5.7b demonstrates fewer data copies for the CUDA aware approach where the communication takes only a fraction of the core solve duration.

Figure 5.8 shows the accumulated exclusive time per function group for the whole duration of the simulation and across all processes. The Kernels portion is the solver code which dominates the simulation time. The CUDA section is memory copies to and from devices and MPI denotes communication and synchronisation. The naïve approach in figure 5.8a shows more than twice the amount of time spent on MPI than the CUDA aware timings in figure 5.8b. The naïve method also spends more time on memory copies. These results for the whole simulation are consistent with the timings for a single time step shown in figure 5.7. Although the communication and transfers are almost completely masked using streams as shown in figure 5.7, the difference between the two approaches is clearly shown.

The amount of data communicated between cards is the same irrespective of the local subdomain depth in the z-direction. This means that as we increase the number of GPUs, the communication will eventually take longer than the solve which seeks to mask it. This will happen earlier for the naïve approach which shows a longer MPI communication time. We therefore expect the CUDA aware communication to perform better at higher card counts for static problem sizes.

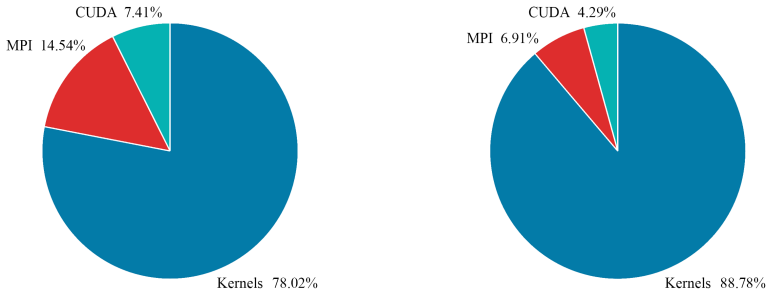


(a) The start of a time step for naïve communication. The red MPI processes are overlapped with the blue x-solve of the core cells but take almost as long as the solve.



(b) The start of a time step for CUDA aware communication. The red MPI processes are overlapped with the blue x-solve and take a fraction of the time as CUDA aware message passing is faster than the naïve approach shown above.

Fig. 5.7 Vampir visualisation for processes at the start of a time step for naïve and CUDA aware communication with four P100 cards. There are four main threads mapped to cards and overlapping CUDA streams on each card. The black lines show data movement between the hosts and devices and across nodes.



(a) Naïve (b) CUDA aware

Fig. 5.8 Accumulated exclusive time per function group. The charts show the proportion of different function types across the whole simulation on four P100 cards for both naïve and CUDA aware communication. The Kernels portion is the solver code which dominates both implementations. The CUDA section is for memory copies to and from cards and the MPI portion is communication and synchronisation. The naïve approach shows higher memory copy times and more than twice the duration for communication.

5.5.2 Solver performance

The performance of the code was analysed using two simulation scenarios: a bubble collapse and the Brittle test case. The bubble collapse simulation is a Mach 1.25 shock wave travelling in the z-direction hitting a bubble of diameter 0.05 m in a 0.1 m $\times$ 0.1 m $\times$ 0.2 m domain. The ambient conditions are $\rho = 1.225\,\mathrm{kg\,m\,s^{-3}}$, $u = v = w = 0\,\mathrm{m\,s^{-1}}$ and $p = 101\,325\,\mathrm{Pa}$. The density inside the bubble is $\rho = 0.182\,\mathrm{kg\,m\,s^{-3}}$. The simulation was run to $t = 750\,\mu\mathrm{s}$. The Brittle test case is the one described previously in chapter 4 without recording the pressure gauges. It was run to $t = 8\,\mathrm{ms}$.

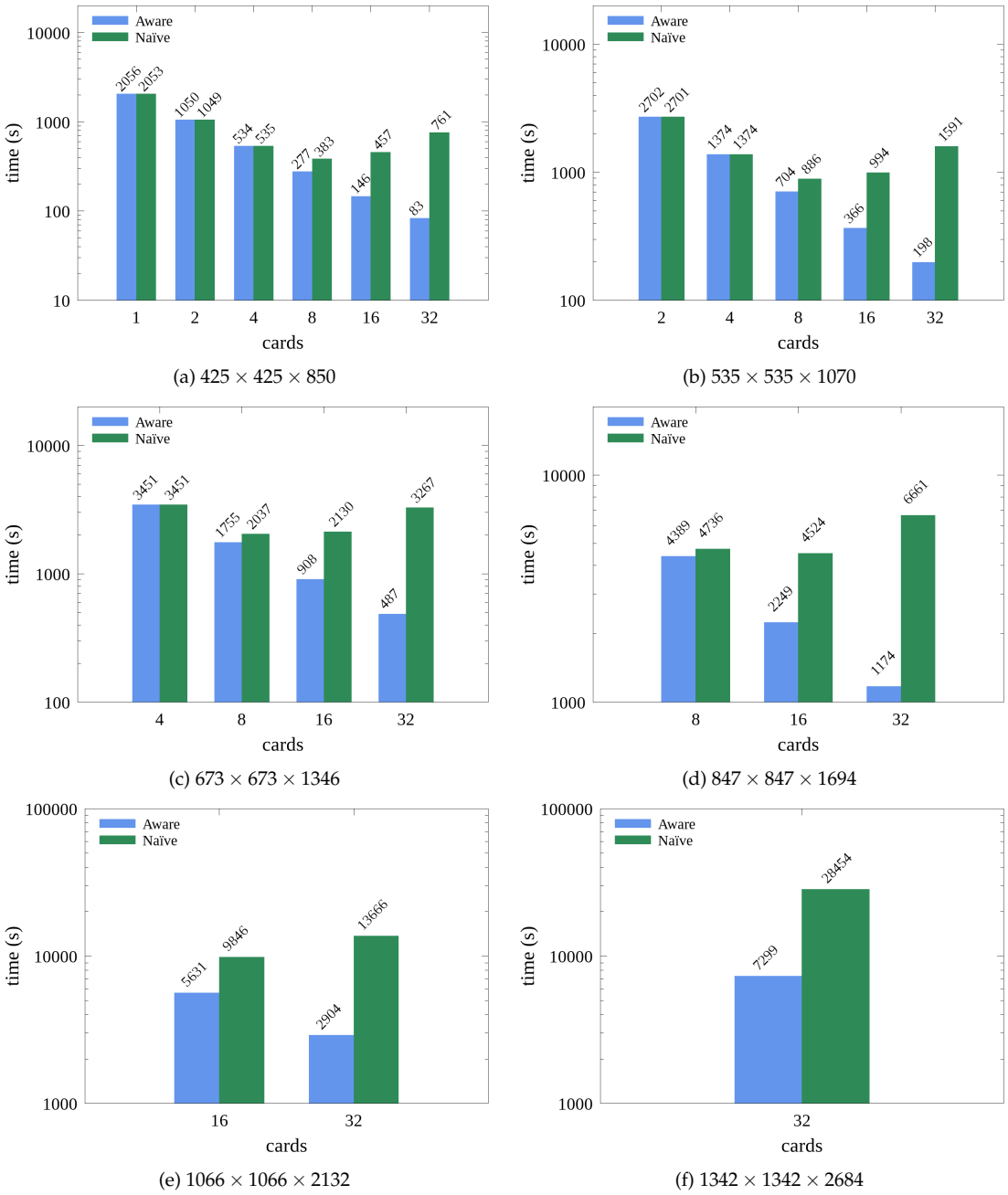
Each simulation was run with six different resolutions as shown in table 5.1. These fill different number of P100 cards to $\sim$90 % capacity.

No	Cards	Collapse	Brittle
1	1	$425 \times 425 \times 850$	$640 \times 640 \times 320$
2	2	$535 \times 535 \times 1070$	$804 \times 804 \times 402$
3	4	$673 \times 673 \times 1346$	$1014 \times 1014 \times 507$
4	8	$847 \times 847 \times 1694$	$1270 \times 1270 \times 635$
5	16	$1066 \times 1066 \times 2132$	$1580 \times 1580 \times 790$
6	32	$1342 \times 1342 \times 2684$	$1940 \times 1940 \times 970$

Table 5.1 Performance run resolutions fill a different number of P100 cards to $\sim$90 %.

All possible configurations were run using naïve and CUDA aware Open MPI. The naïve approach copies ghost cells via the host using pinned memory copies and CUDA aware uses device pointers passed to asynchronous MPI calls. The lowest resolution simulations can be run using all card counts, the second resolution can be run on two or more cards and so on, which, with six resolutions, two scenarios and two MPI strategies, gives a total of 84 runs. The performance results are illustrated in figures 5.9 and 5.10.

The two approaches result in very different timings. The naïve implementation of manually copying from the source device to the host, sending an MPI message and copying to the target device offers good performance on a single node. However, as we communicate between nodes (4+ cards in figures 5.9 and 5.10), we see a drop in performance and eventually an increase in the runtime compared to lower card counts. As seen in the previous section, the memory copies between the host and device are time consuming and we can observe the communication eventually taking up most of the simulation time. The CUDA aware implementation passes device pointers to host side MPI calls and offers significantly better performance as seen in the consistent reduction in runtimes when increasing the number of cards even for the lowest resolution problems in figures 5.9a and 5.10a. We know from the communication analysis that the CUDA aware implementation avoids several memory copies between the host and devices and that the MPI messaging take less time. The results show that the GPU-aware Open MPI implementation offers significantly better performance compared to simplistic message passing.



(a) 425 × 425 × 850

(b) 535 × 535 × 1070

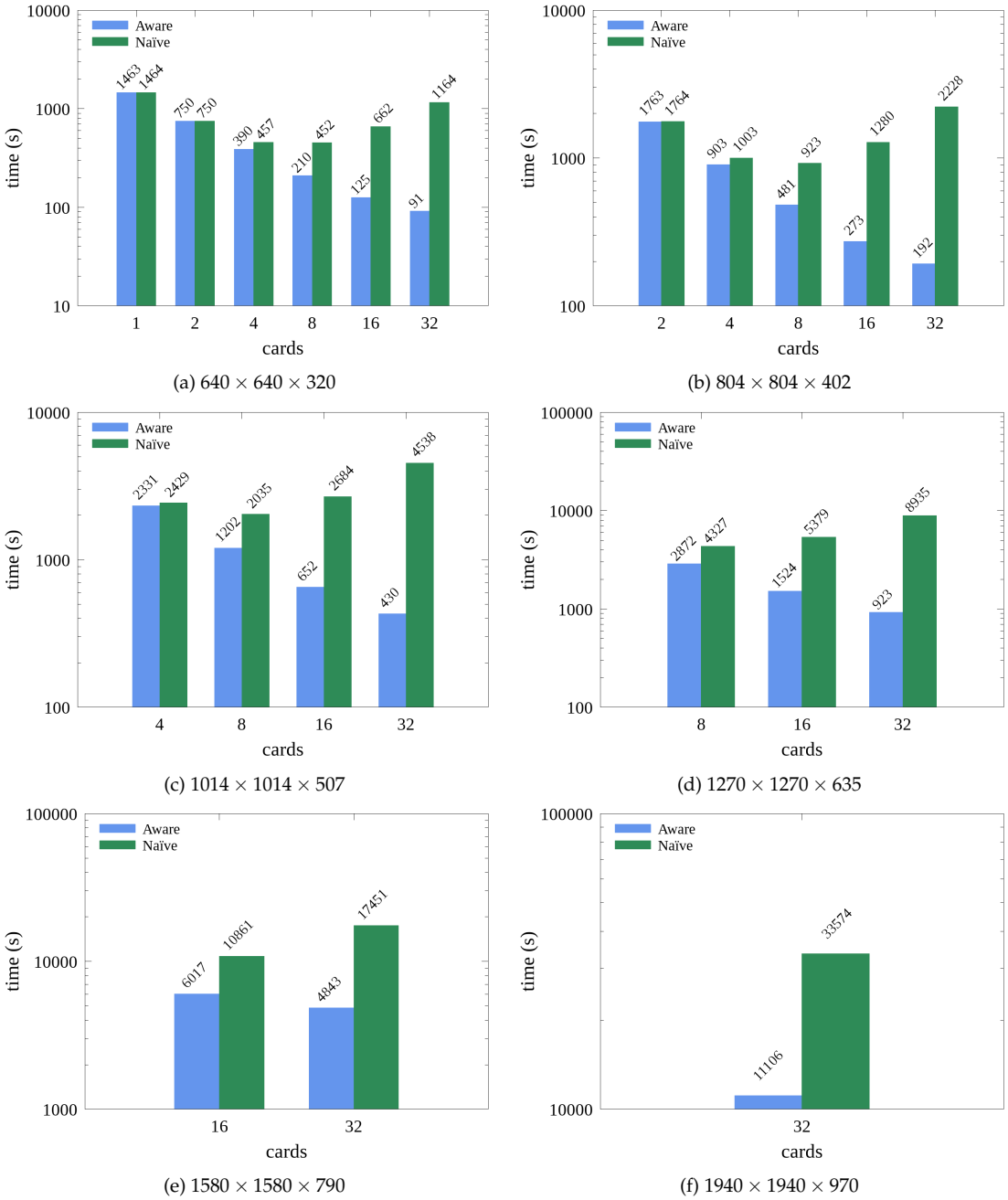
(c) 673 × 673 × 1346

(d) 847 × 847 × 1694

(e) 1066 × 1066 × 2132

(f) 1342 × 1342 × 2684

Fig. 5.9 Runtimes for static resolution 3D bubble collapse simulations on different number of cards. The CUDA aware MPI calls achieve good performance for multiple cards. The naïve approach runs better on a single node than on multiple when memory copies and communication start to dominate the runtime.



(a) $640 \times 640 \times 320$

(b) $804 \times 804 \times 402$

(c) $1014 \times 1014 \times 507$

(d) $1270 \times 1270 \times 635$

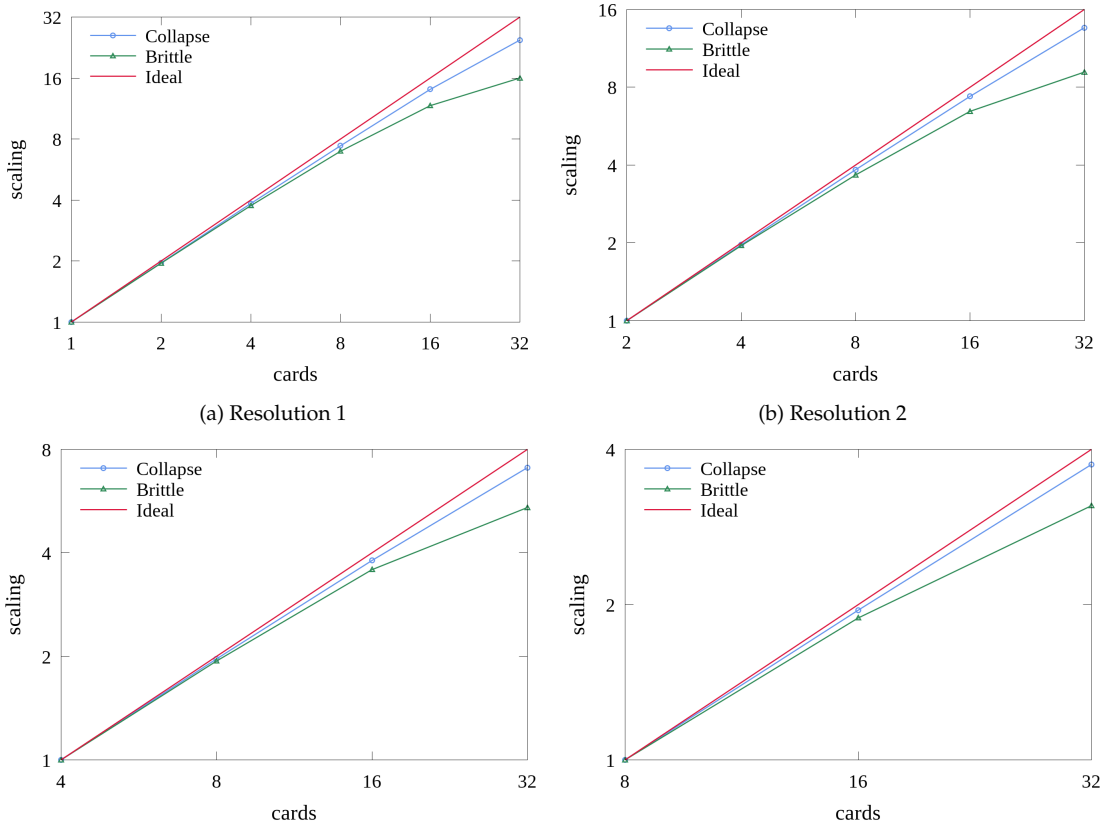
(e) $1580 \times 1580 \times 790$

(f) $1940 \times 1940 \times 970$

Fig. 5.10 Runtimes for static resolution Brittle test case simulations on different number of cards. CUDA aware MPI outperforms the naïve approach which loses efficiency due to copies via host memory.

5.5.3 Scaling

Figure 5.11 shows the strong scaling of the software at the different resolutions using CUDA aware MPI calls. We note the excellent scaling of the clear solver for the lowest resolution as shown in figure 5.11a. Even at 32 cards spanning 8 nodes, the software obtains good speedup compared to a single-card run. The cut cell simulation also scales well but worse than the clear one. The scaling starts to diverge more from ideal at the introduction of more nodes. This is likely because of the increased message size of the cut cell simulation which features a larger cross section along the z-axis, making the stream overlap less effective. For each process, the communication of a static size problem remains constant but smaller subproblems reduce the runtime of the solver. The core section solve may therefore be finished before the MPI communication and ghost region solve at higher card counts.



(a) Resolution 1

(b) Resolution 2

(c) Resolution 3

(d) Resolution 4

Fig. 5.11 Fixed resolution bubble collapse and Brittle code scaling for different number of cards. The clear solver scales excellently for all resolutions and card counts, demonstrating the efficiency in masking communication time with domain splitting and overlapped solvers. The cut cell simulations scale worse due to larger message sizes, but are comparable to the clear solver for up to four nodes.

5.6 Conclusion

The fluid solver software has been extended to a well scaling multi-card software pipeline for clear and cut cell simulations based on schemes chosen following a review of existing literature. Dividing a domain along a single axis allows us to communicate contiguous ghost cell regions. By solving these ghost cell regions in the x- and y-direction, we have to populate them only once per time step. Splitting local domains into ghost and core regions allows us to overlap kernel execution with communication to mask data transfer. We have validated the multi-card implementation using the Brittle blast study using up to 32 cards.

We have profiled the Wilkes2 cluster using OSU micro-benchmarks. We showed that GPUDirect offers significant inter-node performance gain at small message sizes but that this does not benefit software with larger data transfers. We have explored the behaviour of naïve and CUDA aware communication methods. Analysis of the simulation timeline and exclusive duration of different function groups has shown that CUDA aware MPI calls involve fewer copies, take considerably less time and are better masked by the solver stream.

To explore the performance and scaling of our code, we ran several simulations using varying resolutions and card counts. The results show that the code scales well when using CUDA aware MPI calls. Both the clear and cut cell solvers gain good speedup from using multiple cards. Passing messages through host memory demonstrates the expected performance drop as MPI calls start to dominate the overall runtime of the code. CUDA aware implementation features scaling close to linear with the clear test case showing better performance due to a smaller message size.

Chapter 6

Summary and Further Work

Modern graphics hardware can be utilised in general purpose computing to achieve impressive performance. The high number of cores and fast context switching on GPUs provide a significant performance potential for computational fluid dynamics codes. However, programming on GPUs requires careful software design. Development must take into consideration the nature of the underlying hardware and accompanying software interfaces. Most resources are devoted to computation at the expense of caching and data fetching. Memory transactions are considerably slower than arithmetic work and, in order to obtain the best possible performance, data access patterns should be designed to reduce the number of transactions by addressing contiguous memory. These patterns can be used in stencil operations, like shock wave fluid dynamics, and geometry applications, like calculating signed distances in a Cartesian grid.

Signed distance generation

We set out to describe the use of embedded geometries in computational fluid dynamics using graphics hardware. In order to simulate complex boundaries using cut cells we need a description of the surface intersection with the computational mesh. We start by generating a signed distance field based on an input geometry file. A robust GPU implementation of the CSC algorithm by Mauch was developed. We discussed the construction of geometric information from unordered STL data. We presented improvements to deal with complex surface features that can arise in common geometries and showed the completeness of the underlying algorithm. We also looked at work scheduling on graphics cards to quickly generate a narrow band distance field, comparing conventional CUDA kernels with dynamic parallelism and determined that the overhead from device side launches is significant for more complex geometries. The resulting signed distance field generator is used to build a cut cell mesh for our fluid solver.

Computational fluid dynamics

We have presented a CUDA code which solves the Euler equations to simulate shock waves and high-speed flow. We use a framework of array transposes to reposition data for best access patterns in a multidimensional domain. The split solver and dimensional sweeps were validated against several test cases from literature and an analysis of its performance demonstrated the benefits of transposing data and the showed good use of computational resources.

The solver was modified to include cut cells to simulate rigid geometries in the domain. We implemented the method by Klein et al. and the extension by Gokhale et al. The method requires the storage of addition cut cell variables. We presented a discussion about how best to store the data in compact structures to allow high resolution simulations with explicit cut cell addressing. The cut cell solver was validated against several test cases with different flow regimes to demonstrate the correctness and stability of the implementation. An analysis of the combined software showed that most of the work is done by solver kernels and the impact of the cut cells is minimal with the largest memory and runtime effects stemming from a map data structure.

Multi-card parallelisation

The produced software was run on multiple graphics cards. Based on a review of relevant literature, we extended the solver code to split a domain in a single direction and communicate contiguous ghost cells using overlapping streams. We validated the multi-card implementation using the Brittle test case. We explored the performance and scaling of the clear and cut cell solver using the Wilkes2 GPU cluster. An analysis of different MPI strategies showed that CUDA aware Open MPI calls offered good performance for large message sizes and our code scaled well using up to 32 P100 cards spanning eight compute nodes.

The final software pipeline includes fast signed distance generation for complex geometries and a split solver. The threads read and write data using coalesced access patterns and shared memory caching. The cut cell data is stored in a compressed format with explicit addressing allowing us to run high resolution simulations which are not significantly limited by the size of embedded boundaries. The maximum resolution can be increased by using multiple graphics cards and due to contiguous ghost regions and stream overlapping we observe good scaling when using CUDA aware Open MPI.

Limitations and further work

There are several limitations to the produced software. The SDF generator requires orientable closed input surfaces with no gaps or flipped faces. The fluid solver is not optimised for local

register use and could benefit from an in-depth analysis of memory pressure. The solver is designed around a split approach and the data transposes constitute a significant portion of the compute time. The KBN cut cell method is first order at the boundary and the handling of doubly shielded cells is not a rigorous fix for the instability of such regions. Though the use of compressed data structures for cut cell data greatly reduces the memory footprint of boundaries, the necessary map and its transposition further increase the proportion of time spent on memory access. The multi-card parallelisation does not address explicit peer-to-peer access on a single node which could lead to better performance of smaller simulation configurations. The above limitations are all good candidates for future work. In addition it would be useful to look at the behaviour of higher order surface methods and unsplit solvers together with adaptive mesh refinement. Further analysis of different hardware layouts and alternative multi-card communication strategies would offer a better overview of the bottlenecks and limitations of stencil operation software on graphics hardware.

Despite the limitations, the produced software constitutes a robust and well parallelised fluid solver with embedded boundaries on graphics hardware. We have demonstrated how literature and software can be utilised to extend existing algorithms to massively parallel architectures and how to harness the performance potential of such hardware by carefully considering its unique properties.

References

[1] Amick, J. L. (1950). Comparison of the experimental pressure distribution on an NACA 0012 profile at high speeds with that calculated by the relaxation method.

[2] Anderson, J. A., Lorenz, C. D., and Travesset, A. (2008). General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359.

[3] Anderson Jr., J. (1995). *Computational Fluid Dynamics: The Basics with Application*. McGraw-Hill International Editions.

[4] Baerentzen, J. A. and Aanaes, H. (2005). Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253.

[5] Baker, W., Cox, P., Westine, P., Kulesz, J., and Strehlow, R. (1983). *Explosion Hazards and Evaluation*. Elsevier Scientific Publishing Co., Amsterdam, The Netherlands.

[6] Bennett, W., Nikiforakis, N., and Klein, R. (2018). A moving boundary flux stabilization method for Cartesian cut-cell grids using directional operator splitting. *Journal of Computational Physics*, 368:333–358.

[7] Berger, M. and Helzel, C. (2012). A simplified h-box method for embedded boundary grids. *SIAM Journal on Scientific Computing*, 34(2):A861–A888.

[8] Bernaschi, M., Bisson, M., Fatica, M., and Phillips, E. (2012). An introduction to multi-GPU programming for physicists. *The European Physical Journal Special Topics*, 210(1):17–31.

[9] Blakely, P. (2010). Implementing Finite Volume algorithms on GPUs, 2nd UK GPU Computing Conference, Cambridge, UK.

[10] Brandvik, T. and Pullan, G. (2008). Acceleration of a 3D Euler solver using commodity graphics hardware. In *46th AIAA aerospace sciences meeting and exhibit*, page 607.

[11] Bridson, R., Marino, S., and Fedkiw, R. (2003). Simulation of clothing with folds and wrinkles. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 28–36. Eurographics Association.

[12] Brittle, M. (2004). Blast propagation in a geometrically complex urban environment. Master's thesis, Royal Military College of Science.

[13] Bryson, A. and Gross, R. (1961). Diffraction of strong shocks by cones, cylinders, and spheres. *Journal of Fluid Mechanics*, 10(1):1–16.

[14] Chang, S.-M. and Chang, K.-S. (2000). On the shock–vortex interaction in Schardin's problem. *Shock Waves*, 10(5):333–343.

[15] Clarke, D. K., Hassan, H., and Salas, M. (1986). Euler calculations for multielement airfoils using Cartesian grids. *AIAA journal*, 24(3):353–358.

[16] Cohen, J. and Molemaker, M. J. (2009). A fast double precision CFD code using CUDA. *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pages 414–429.

[17] Colella, P., Graves, D. T., Keen, B. J., and Modiano, D. (2006). A Cartesian grid embedded boundary method for hyperbolic conservation laws. *Journal of Computational Physics*, 211(1):347–366.

[18] de León, M. Orion Capsule, NASA 3D resources. *https://nasa3d.arc.nasa.gov/detail/orion-capsule*. Retrieved 09.2018.

[19] Drazin, W. (2018). *Blast Propagation and Damage in Urban Topographies*. PhD thesis, University of Cambridge.

[20] Fedkiw, R. P., Aslam, T., Merriman, B., and Osher, S. (1999). A non-oscillatory Eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *Journal of computational physics*, 152(2):457–492.

[21] Gokhale, N., Nikiforakis, N., and Klein, R. (2018). A dimensionally split Cartesian cut cell method for hyperbolic conservation laws. *Journal of Computational Physics*, 364:186–208.

[22] Haas, J.-F. and Sturtevant, B. (1987). Interaction of weak shock waves with cylindrical and spherical gas inhomogeneities. *Journal of Fluid Mechanics*, 181:41–76.

[23] Hagen, T. R., Lie, K.-A., and Natvig, J. R. (2006). Solving the Euler equations on graphics processing units. In *International Conference on Computational Science*, pages 220–227. Springer.

[24] Hartmann, D., Meinke, M., and Schröder, W. (2011). A strictly conservative Cartesian cut-cell method for compressible viscous flows on adaptive grids. *Computer Methods in Applied Mechanics and Engineering*, 200(9-12):1038–1052.

[25] Jacobsen, D., Thibault, J., and Senocak, I. (2010). An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, page 522.

[26] Janßen, C., Koliha, N., and Rung, T. (2015). A fast and rigorously parallel surface voxelization technique for GPU-accelerated CFD simulations. *Communications in Computational Physics*, 17(5):1246–1270.

[27] Jodra, J. L., Gurrutxaga, I., and Muguerza, J. (2015). Efficient 3D transpositions in graphics processing units. *International Journal of Parallel Programming*, 43(5):876–891.

[28] Klein, R., Bates, K., and Nikiforakis, N. (2009). Well-balanced compressible cut-cell simulation of atmospheric flow. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 367(1907):4559–4575.

[29] Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M. S., and Nagel, W. E. (2008). The Vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer.

[30] Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al. (2012). Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing*, pages 79–91. Springer.

[31] Koza, Z., Matyka, M., Szkoda, S., and Mirosław, Ł. (2014). Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM Journal on Scientific Computing*, 36(2):C219–C239.

[32] Kurganov, A. and Tadmor, E. (2002). Solution of two-dimensional Riemann problems for gas dynamics without Riemann problem solvers. *Numerical Methods for Partial Differential Equations*, 18(5):584–608.

[33] Lax, P. D. and Liu, X.-D. (1998). Solution of two-dimensional Riemann problems of gas dynamics by positive schemes. *SIAM Journal on Scientific Computing*, 19(2):319–340.

[34] Lefebvre, M., Guillen, P., Le Gouez, J.-M., and Basdevant, C. (2012). Optimizing 2D and 3D structured Euler CFD solvers on graphical processing units. *Computers & Fluids*, 70:136–147.

[35] Mauch, S. stdlib. *https://bitbucket.org/seanmauch/stlib/src/*. Retrieved 06.2016.

[36] Mauch, S. (2000). A fast algorithm for computing the closest point and distance transform. *http://www.acm.caltech.edu/seanm/software/cpt/cpt.pdf, Retrieved 20.02.2017*.

[37] Micikevicius, P. (2009). 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 79–84. ACM.

[38] MPI Forum. The Message Passing Interface. *https://www.mpi-forum.org/*. Retrieved 09.2018.

[39] Network-Based Computing Laboratory, Ohio State University. OSU Micro-Benchmarks. *http://mvapich.cse.ohio-state.edu/benchmarks/*. Retrieved 09.2018.

[40] Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53.

[41] Nvidia. Developing a Linux kernel module using RDMA for GPUDirect. *https://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf*. Retrieved 09.2018.

[42] Nvidia. Nvidia profiler. *https://docs.nvidia.com/cuda/profiler-users-guide/index.html*. Retrieved 09.2018.

[43] Nvidia. Tesla GPU Accelerators. *http://www.nvidia.co.uk/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf*. Retrieved 09.2018.

[44] Nvidia. Tesla P100 GPU Accelerator. *https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf*. Retrieved 09.2018.

[45] Nvidia. Thrust library. *http://docs.nvidia.com/cuda/thrust/*. Retrieved 09.2018.

[46] Okamoto, T., Takenaka, H., Nakamura, T., and Aoki, T. (2010). Accelerating large-scale simulation of seismic wave propagation by multi-GPUs and three-dimensional domain decomposition. *Earth, planets and space*, 62(12):939–942.

[47] Open MPI project. Open MPI. *https://www.open-mpi.org/*. Retrieved 09.2018.

[48] Park, T., Lee, S.-H., Kim, J.-H., and Kim, C.-H. (2010). CUDA-based signed distance field calculation for adaptive grids. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1202–1206. IEEE.

[49] Peikert, R. and Sigg, C. (2005). Optimized bounding polyhedra for GPU-based distance transform. In *Proceedings of Dagstuhl Seminar on Scientific Visualization*.

[50] Pharr, M., Jakob, W., and Humphreys, G. (2016). *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann.

[51] Quirk, J. J. and Karni, S. (1996). On the dynamics of a shock–bubble interaction. *Journal of Fluid Mechanics*, 318:129–163.

[52] Roosing, A., Strickson, O., and Nikiforakis, N. (2019). Fast Distance Fields for Fluid Dynamics Mesh Generation on Graphics Hardware. *Commun. Comput. Phys.*, (26):654–680.

[53] Ruetsch, G. and Micikevicius, P. (2009). Optimizing matrix transpose in CUDA. *Nvidia CUDA SDK Application Note*, 18.

[54] Schneible, J., Riha, L., Malik, M., El-Ghazawi, T., and Alexandru, A. (2015). Communication efficient work distributions in stencil operation based applications. *Concurrency and Computation: Practice and Experience*, 27(13):3262–3280.

[55] Sigg, C., Peikert, R., and Gross, M. (2003). Signed Distance Transform Using Graphics Hardware. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03. IEEE Computer Society.

[56] Stanford Computer Graphics Laboratory. The Stanford 3D scanning repository. *http://graphics.stanford.edu/data/3Dscanrep/*. Retrieved 09.2018.

[57] Sud, A., Otaduy, M. A., and Manocha, D. (2004). DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum*, 23(3):557–566.

[58] Tang, X., Pattnaik, A., Jiang, H., Kayiran, O., Jog, A., Pai, S., Ibrahim, M., Kandemir, M. T., and Das, C. R. (2017). Controlled Kernel Launch for dynamic parallelism in GPUs. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 649–660. IEEE.

[59] Technical University of Munich. DrivAer model. *https://www.aer.mw.tum.de/en/research-groups/automotive/drivaer/*. Retrieved 09.2018.

[60] Thibault, J. C. and Senocak, I. (2009). CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In *Proceedings of the 47th AIAA aerospace sciences meeting*, pages 2009–758.

[61] Toro, E. (2009). *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Springer-Verlag.

[62] Uetz, R. and Behnke, S. (2009). Large-scale object recognition with CUDA-accelerated hierarchical neural networks. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 1, pages 536–541. IEEE.

[63] University of Cambridge Research Computing Service. Wilkes2. *https://www.csd3.cam.ac.uk/*. Retrieved 09.2018.

[64] Wang, J. and Yalamanchili, S. (2014). Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 51–60. IEEE.

[65] Wong, K. V. and Hernandez, A. (2012). A review of additive manufacturing. *ISRN Mechanical Engineering*, 2012.

[66] Xian, W. and Takayuki, A. (2011). Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing*, 37(9):521–535.