GPU Accelerated Linear System Solvers for OpenFOAM and Their Application to Sprays

A thesis submitted for the degree of Doctor of Philosophy

by

Joshua Dyson

Department of Mechanical, Aerospace and Civil Engineering, College of Engineering, Design and Physical Sciences

Brunel University London

Abstract

This thesis presents the development of GPU accelerated solvers for use in simulation of the primary atomization phenomenon. By using the open source continuum mechanics library, OpenFOAM, as a basis along with the NVidia CUDA API linear system solvers have been developed so that the multiphase solver runs in part on GPUs. This aims to reduce the enormous computational cost associated with modelling primary atomization. The modelling of such is vital to understanding the mechanisms that make combustion efficient. Firstly, the OpenFOAM code is benchmarked to assess both its suitability for atomization problems and to establish efficient operating parameters for comparison to GPU accelerations. This benchmarking then culminates in a comparison to an experimental test case, from the literature, dominated by surface tension, in 3D.

Finally, a comparison is made with a primary atomizing liquid sheet as published in the literature. A geometric multigrid method is employed to solve the pressure Poisson equations, the first use of a geometric multigrid method in 3D GPU accelerated VOF simulation. Detailed investigations are made into the compute efficiency of the GPU accelerated solver, comparing memory bandwidth usage to hardware maximums as well as GPU idling time. In addition, the components of the multigrid method are also investigated, including the effect of residual scaling. While the GPU based multigrid method shows some improvement over the equivalent CPU implementation, the costs associated with running on GPU cause this to not be significantly greater.

Contents

Acknowledgements

I must thank anyone that has contributed directly or indirectly to my study throughout my 7 years at Brunel University first during undergraduate and then post-graduate programs. Though there are some that deserve special thanks.

First my supervisor Dr Jun Xia for his guidance throughout this project, as well as organising numerous seminars that have contributed to my understanding of this subject. Dr Junji Shinjo for his help and advice during the initial stages of study, as well as others Dr Kadi Wan, Dr Lei Zhao, Dr Alan Gray and Dr George Fern for suggestions and guidance.

I must also thank the Thomas Gerald Gray scholarship for the funding that made this project possible as well as the trustees for their discussion during annual presentations.

I must thank my family and specifically my parents, Dr Paul Dyson and Dr Angela Dyson, for all the endless support and guidance they have given me.

Finally, I must thank my granny, Judy Dyson, for always being there when I needed it but who passed away just before this thesis was submitted. You will be greatly missed.

List of Figures

List of Tables

Nomenclature

Roman Symbols

A	Matrix of coefficients (linear system)
b	Source terms (linear system)
B	Byte (8 Bits)
C_γ	Interface Compression Coefficient
Ca	Capilary Number
Co	Courant Number
CoM	Centre of Mass [m]
d	Cell Size [m]
D	Jet Diameter [m]
$\overleftrightarrow{D}$	Rate of Strain Tensor [/s]
E	Parallel Efficiency
Eo	Eötvös number
g	Gravitational Acceleration [m/s^2]
h	Half Sheet Thickness [m]
I	Interpolation Matrix
l	Characteristic Length [m]
M	Preconditioning matrix
N	Number of Cores
n_f	Cell Surface Normal Vector
Oh	Ohnesorge number
p	Pressure [N/m^2]
P	Search direction vector (Linear System)
R	Residual Vector (Linear System)
Re	Reynolds Number
S_f	Cell Surface Area [m^2]

sf	Scaling Factor
t	Time [s]
T	Compute Time [s]
U	Velocity [m/s]
$\vec{U}$	Velocity Vector [m/s]
U_r	Relative Velocity [m/s]
We	Weber Number
x	Result Vector (linear system)
X_{liq}	Intact Liquid Core Length [m]

Greek Symbols

ρ	Density [kg/m^3]
σ	Surface Tension [n/m]
μ	Dynamic Viscosity [Pa.s]
γ	Volume Fraction
κ	Interface Curvature [m]
Δt	Time Step [s]
Δx	Grid Spacing [m]
π	3.14159…
Φ	Mass Flux [kg/s.m^2]

Acronyms

AMBER	Assisted Model Building with Energy Refinement
AMD	Advanced Micro Devices
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient

CLSVOF	Coupled Level Set and Volume Of Fluid
COO	Coordinate matrix format
CPU	Central Processing Unit
CSF	Continuum Surface Method
CSR	Compressed Sparse Row matrix
cuBLAS	CUDA Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
DICPCG	Diagonal Incomplete Cholesky Preconditioned Conjugate Gradient
DNS	Direct Numerical Simulation
DPCG	Diagonal Preconditioned Conjugate Gradient
DRAM	Dynamic Random-Access Memory
FLOPS	Floating Point Operations Per Second
GAMG	Generalised Geometric-Algebraic Multigrid
GenIDLEST	Generalized Incompressible Direct and Large Eddy Simulation of Turbulence
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
I/O	Input/Output
LDU	Lower Diagonal Upper
LES	Large Eddy Simulation
MPI	Message Passing Interface
NAMD	Nanoscale Molecular Dynamics
NTS	Number of Time Steps
OpenCL	Open Computing Language
OpenFOAM	Open Field Operation and Manipulation
PCIe	Peripheral Component Interconnect Express
PISO	Pressure Implicit with Splitting of Operators
RAM	Random Access Memory

RANS	Reynolds Averaged Navier-Stokes
SIMD	Single Instruction Multiple Data
SIMPLE	Semi-Implicit method for Pressure Linked Equations
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
VOF	Volume of Fluid

Chapter 1 Introduction

1.1 Motivation

Despite recent developments in new “cleaner” fuelled modes of transport, the burning of fossil fuels is still dominant in most forms of transport. Even though fossil fuel transport has existed for over a century the detailed mechanisms that make it possible and efficient are still not well understood. One of these critical mechanisms is fuel spray. Current developments in more efficient fuel spray are limited to resource intensive trial and error methods. To replace this process, it is desirable to simulate the phenomenon from first principles. However, this requires enormous computing resources. Therefore, work that has been done to date consists of high fidelity simulation of limited spray regions e.g. (Shinjo & Umemura, 2010). If simulation is to replace trial and error the computational cost must be reduced to the point where it becomes a viable alternative. One avenue to achieve this is alternate emerging computing platforms. These have the potential to reduce times to solution, power consumption and increase performance density. Or indeed a combination of these depending on the user’s requirements. Despite the potential of accelerated computing, it is still yet to be used in primary atomization modelling. Indeed, more generally accelerated computing is still in it’s infancy for much more general computational modelling approaches. Therefore, it is highly desireable to investigate the potentital of accelerated computing in making primary atomization mdeling a viable alternative to trail and error.

1.2 Accelerated Computing

Accelerated computing is a computing model that uses specialized processors to achieve some kind of performance benefit. In the quest for ever more powerful computing resources, accelerated computing has become the norm. Indeed, the two fastest supercomputers in the world employ some kind of accelerated computing (top500, 2015). As well as this the 32 most energy efficient supercomputers are configured in such a way (top500, 2015).

The common styles are that a standard Central processing unit (CPU) is used as a controller for an additional many core processor that is focused on throughput. These many core processors come in two distinct varieties. The first is the Co-Processor, popularised by Intel. It generally has fewer cores, this means it's the slowest performer but requires less specialist programming. Alternatively, graphics processing unit (GPUs) are generally considered the most mature technology. They offer the highest performance but require specialist programming knowledge. Their development has been driven by the computer games industry, often making them inexpensive. Manufacturers are generally AMD and NVidia, as will be explained later the manufacturer can dictate the software that must be used to program them.

The latest co-processor, Intel Xeon Phi, has around 70 cores with high speed access to on board memory as well as CPU speed access to system memory. GPUs have many cores, in the regime of four to five thousand with higher speed access to onboard memory however access to system memory is generally slow. The latest Xeon Phis, 2016, have been made as bootable devices, blurring the line between accelerators and CPUs.

1.3 Thesis Outline

This thesis is outlined as follows. Chapter 2 introduces the concepts behind primary atomization of liquid spray as well as computational methods used to simulate them, concluding with a review of previous investigations in the field. Chapter 3 introduces GPU computing and discusses its uses in several fields, primarily in computational fluid dynamics. Chapter 4 presents the computational methods employed and the governing equations involved. Chapter 5 presents initial test cases that show OpenFOAMs applicability to multiphase flows. Also shown are details of computational effort required. Chapter 6 investigates a test case that brings together the elements tested in the previous chapter to assess its use in primary atomization. Also discussed is accelerations to the OpenFOAM code using

GPUs. Chapter 7 details a much more complex test case of primary atomization. Chapter 8 explores an improved GPU accelerated method and its application to the case described in Chapter 7. Finally, some conclusions are made along with suggestions for further work.

Chapter 2 Review of Spray Atomization

2.1 Breakup Regimes of round jets

A liquid jet emanating from an orifice into stagnant air has been investigated since the 1800s. As a result of various investigations there is a common consensus of a group of regimes defined by three dimensionless numbers (Lefebvre, 1989). All are normalised by a characteristic length l, usually the nozzle diameter for round jets and the sheet thickness for sheets. The first, the Weber number (equation 1), is the ratio between the fluid's inertia and surface tension.

$$We = \frac{\rho U^2 l}{\sigma} \tag{1}$$

Second is the Reynolds number (equation 2), which is the ratio of inertial to viscous forces:

$$Re = \frac{\rho U l}{\mu} \tag{2}$$

Thirdly, the Ohnesorge number (equation 3), relates viscous forces to surface tension forces:

$$Oh = \frac{\mu}{\sqrt{\rho \sigma l}} \tag{3}$$

In each of these ρ is the fluid density, U the liquid velocity, σ the surface tension coefficient and μ the liquid viscosity. In addition, the Eötvös number (or Bond number) can also be used to describe the flow conditions, this is a relationship between gravitational forces and surface tension, shown in equation 4.

$$Eo = \frac{\rho g^2 l}{\sigma} \tag{4}$$

Where g is is the acceleration due to gravity, l is the characteristic length, ρ is the fluid density and σ is the surface tension coefficient. Finally, the Capillary number is the relationship between viscous and surface tension forces across an interface.

$$Ca = \frac{\mu U}{\sigma} \tag{5}$$

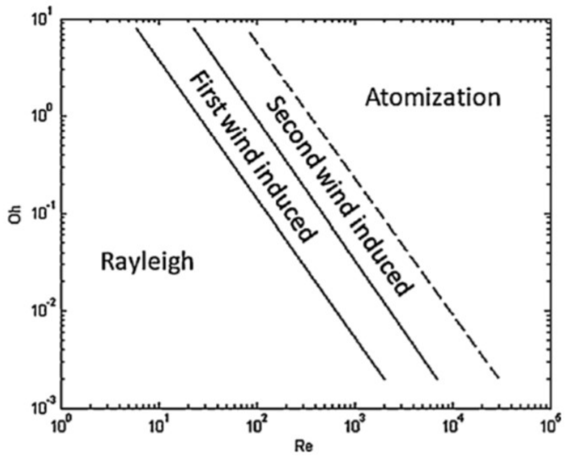


Figure 1*: Breakup regime characterization (Lefebvre, 1989)*

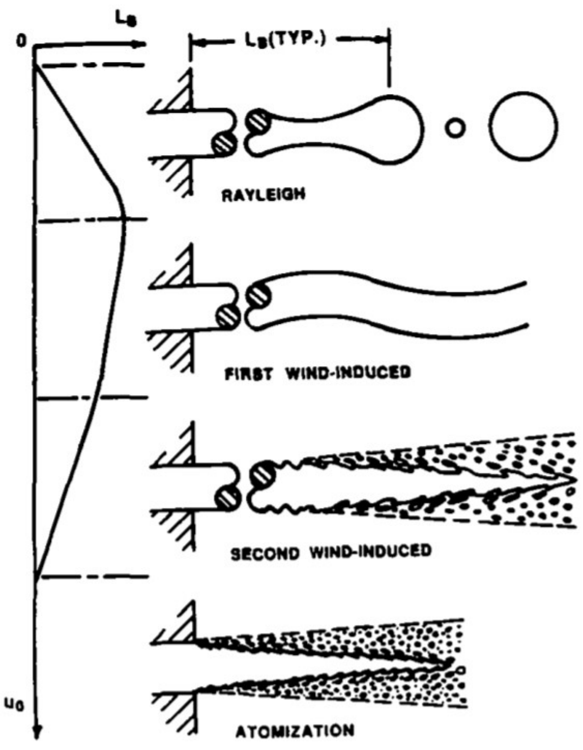


Figure 2: Flow patterns of breakup regimes in Figure 1 *(Faeth, 1991)*

The relationship of the Ohnesorge and Reynolds numbers, which are named in Figure *1*, gives a characterization of the breakup patterns that are visualised in Figure 2. In the Rayleigh regime surface tension forces are responsible for the jet breakup. Droplets in this regime are greater in diameter than the nozzle and the onset of breakup is many diameters after the nozzle exit, described by Rayleigh in (Rayleigh, 1879). In the 1st wind regime aerodynamic forces start to take over from surface tension as the driver of breakup. Droplets of about the nozzle diameter are formed and the point of breakup is reduced. In the 2nd wind induced regime greater aerodynamic forces cause ligaments and droplets to be broken off the liquid core even earlier, these liquid structures will then further breakup into droplets smaller than the nozzle. Finally, in the atomization regime the jet is fully turbulent and breakup begins at the exit of the nozzle. Eventually droplets many times smaller than the nozzle diameter will be produced. When applied to fuel sprays, which generally are injected into the combustion chamber at high velocity, the characteristic dimensionless numbers place the spray in the atomization regime.

The atomization of a spray is a complex multiscale problem and is commonly split into two distinct phases. The first is primary atomization, which takes place near to the nozzle exit. In this section the liquid core is generally intact with smaller ligaments or droplets emanating from the core, shown in blue in Figure 3. The next phase is secondary atomization where generally little or none of the liquid core is intact. The common forms of breakup in this region are the drops and ligaments that have detached from the liquid core breaking into yet smaller drops, shown in Figure 3 in green. Secondary atomization can be further divided into additional regimes. Using the classification shown in Figure 3 the dense regime will contain larger liquid structures that aren't of regular sizes. Following this in the intermediate regime the liquid structures have broken down into more regular spherical droplets. Finally, in the dilute regime these spherical droplets now sparsely populate the gas phase.

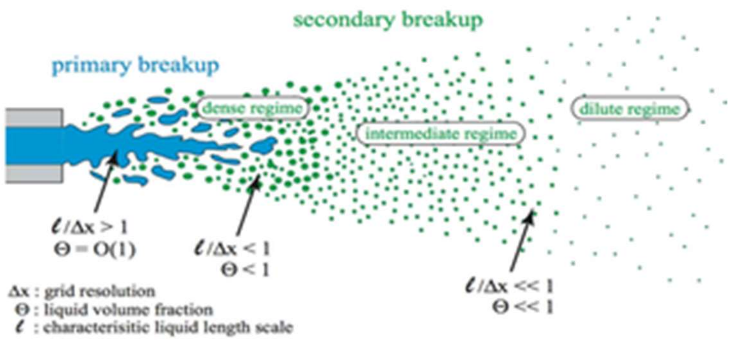


Figure 3: Illustration of breakup regimes (Sun, 2016)

2.2 Breakup of a liquid sheet

In a sheet configuration, the liquid phase is injected through a slit generally several times as wide as it is thick. This kind of injection configuration can be found in many applications but most commonly in aerospace engines. Early investigations into the instability of inviscid sheets was conducted by Squire (1953), Taylor (1959) and Hagerty & Shea (1955). Hagerty & Shea (1955) showed experimentally that the sinuous and varicose modes predicted by theoretical analysis could be found at the sheet interface as shown in Figure 4.

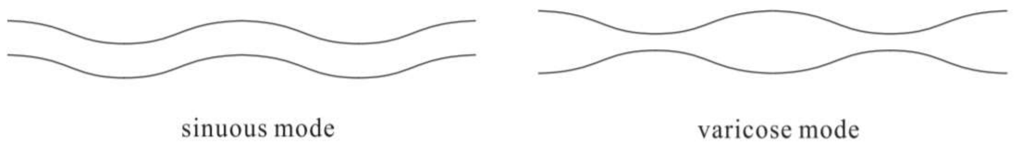


Figure 4: Illustration of Sinuous and Varicose modes (Wang, et al., 2015)

By furthering instability analysis to a viscous liquid sheet Li & Tankin, (1991) stated that viscosity does have a significant impact on the modes of instability showing that in contrast to inviscid sheets, where the only mode of instability is aerodynamic, viscous sheets have an additional viscosity enhanced instability.

Several investigations have described the atomization of a liquid sheet (Fraser, et al., 1962) & (Dombrowski & Johns, 1963). The review article by Sirignano & Mehring, (2000) describes three modes of breakup: rim, wave and perforated-sheet. The wave mode is shown in Figure 5. In the rim mode, the surface tension forces cause the free edge of the sheet to contract into a thick rim. This then breaks up in a mode corresponding to that of a liquid jet described previously. This mode is most prominent when the liquids viscosity and surface tension are both high. In the perforated sheet mode holes appear in the sheet that grow in size and coalesce producing ligaments of varying sizes that further break into drops.

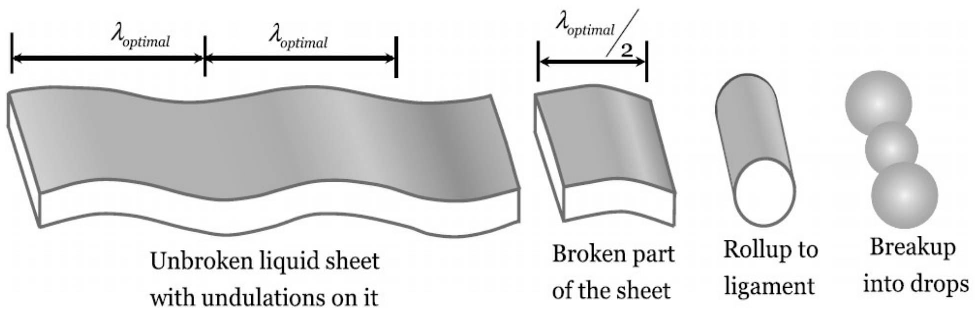


Figure 5: Popular version of the sheet atomization process (Deshpande, et al., 2015)

Finally, the wave mode is where the sheet is broken up by half or full wavelengths of the most unstable wave. The broken off sheet sections then roll into ligaments owing to surface tension before breaking up into droplets.

2.3 Computational Modelling of Atomization

The characteristics described previously partly explain why the primary atomization regime is still not well understood. While it is accepted that there are many phenomena that promote instability in the jet or sheet, the dominant phenomena are still to be identified. Using experimental techniques has been difficult owing to the need to produce images with a very high temporal and spatial resolution. Therefore, using numerical methods to study this area has received far more attention. As the spray breakup is often split into two regions, generally so is the modelling. In the primary atomization region there are complex topological changes in the fluid-gas interface therefore it is sensible to resolve these changes using the Eulerian coordinate system. However, in the secondary atomization regime the liquid structures are far more "simple" and can be generalised as spherical droplets. To this end the Lagrangian coordinate system is often used. Models are then created to simulate the secondary breakup of the spherical droplets e.g. (Apte, et al., 2003). More recently there has been development in the coupling of Eulerian and Lagrangian methods, with the aim of simulating the whole spray process at a reasonable cost (Herrmann, 2010).

Numerical simulations such as this require the ability to identify the interface between the liquid and gas in some way. These methods are often grouped into two distinct categories, each with their own strengths and weaknesses. These are explicit and implicit. In explicit methods, the computational mesh will move with the interface whereas in implicit methods some scalar field is used to describe the location of the interface on a fixed mesh. The most commonly used explicit method is the Front tracking method (Unverdi & Tryggvason, 1992). The flow field is described in the normal Eulerian mesh system, but an additional unstructured grid is used to describe the interface between multiple fluids. This additional grid moves through the stationary grid and thus requires regular grid reconstruction however as the interface is explicitly described surface tension is easy to describe. The most typically used implicit methods are the Volume Of Fluid (VOF) method (Hirt &

Nichols, 1981) and the level set method (Osher & Sethian, 1988). The VOF method uses a scalar value for a “volume fraction” this scalar is bounded by zero and one and represents the volume of one cell occupied by one fluid. In cells that are completely filled with one fluid or another, the volume fraction will be one (or zero). At the interface there will be cells containing both fluids and in these locations the volume fraction will be between one and zero this is visualised in Figure 6.

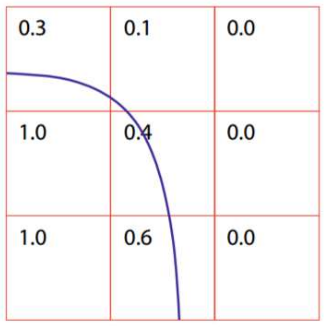


Figure 6: Excerpt of a domain showing an example of volume fractions in a mesh and the interface created with them (Elgeti & Sauerland, 2016)

The level set function is based on the transport of a function using the velocity field. The level set function can be described as the signed distance function to the interface, i.e. the interface is located at the zero-level set. Negative values are one fluid while positive are the other fluid.

The VOF methods strengths are its ability to easily manage merging and break-up of fluid structures, as well as maintaining a sharp interface and good mass conservation (Gopala & van Wachem, 2008). However, VOF suffers from an uncertainty in the interface curvature owing to the need to locally reconstruct the interface in each cell. This often leads to inaccuracies in the calculation of surface tension forces. Additionally, it also often suffers from unphysical numerical diffusion of the interface. Alternatively, the level set method has the reverse characteristics. More recently there has been a move to the coupled level set and volume of fluid solver (CLSVOF) (Sussman & Puckett, 2000), this combines the advantages of the VOF method with the level set method therefore outperforming both.

With a defined interface the next numerical obstacle is how to treat the singularities that occur at the interface. Two common methods can be found in multiphase CFD, the Continuum Surface Force (CSF) method (Brackbill, et al., 1992) and the Ghost fluid method (Fedkiw, et al., 1999). The original formulation of the CSF method made by Brackbill et al., (1992) was for the VOF method. Later this was extended to level set by Chang et al., (1996). In the CSF method, the interface is represented as a region with a thickness, as show in Figure 7. Therefore, the fluid properties (density and viscosity) are treated as a smooth function across the interface region. Surface tension is also transformed into a volume force applied across this region.

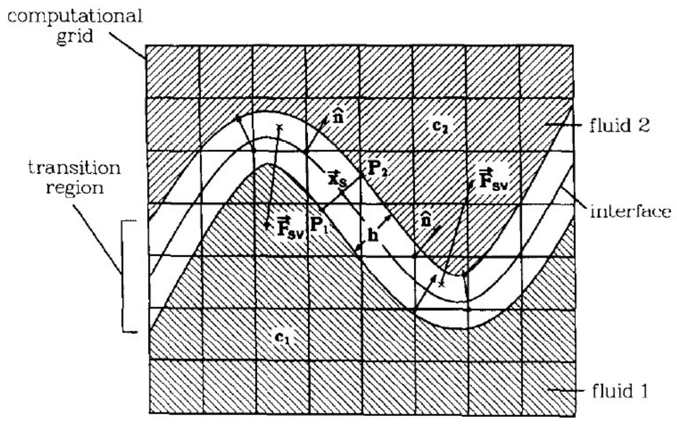


Figure 7: Visualisation of the CSF method in 2D (Brackbill, et al., 1992)

This treatment of the interface as a region with a thickness has a distinct drawback. Often a spurious velocity is introduced at the interface (Harvie, et al., 2006) due to a smooth pressure field being computed by the CSF method when in reality a pressure jump condition should exist at the interface. Despite this limitation the CSF method has found use in spray atomization simulations.

In order to combat the spurious currents common in the CSF method; Fedkiw et al., (1999) developed the ghost fluid method originally for use in the compressible

Euler equations. It was further developed by Liu et al., (2000) and Kang et al., (2000) to the point where it could be used with two phase incompressible flows. In the ghost fluid method, the surface tension force is applied by a jump condition in the pressure. With estimated jump conditions an algorithm will extend each fluid a few cells into the other, beyond the interface, at each time step. These are the so called "ghost regions" this is illustrated in Figure 8.

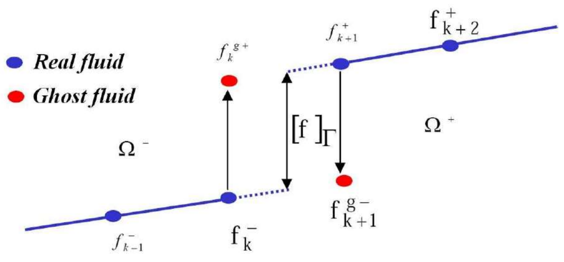


Figure 8: Illustration of the ghost fluid method (Pringuey, 2012)

In the band around the interface the governing equations (mass, momentum and energy) are solved for both fluids. The correct solution is then selected from the two available using the fluid descriptor from the interface capturing method. The original formulation as proposed by Fedkiw et al., (1999) was applied to the level set method, the sign of the function selecting the fluid in that location. The level set method is well suited to be a companion of the ghost fluid method as the jump conditions it requires are easily derived from the signed distance function. However, if used in combination with the VOF method a distance function will need to be constantly recalculated with the movement of the interface. The ghost fluid method has been used by many investigations into primary atomization in combination with level set (Desjardins, et al., 2008) and VOF (Menard, et al., 2007).

2.4 Turbulence Considerations

Primary atomization occurs from high speed liquid injection, so the corresponding length and time scales vary significantly. Therefore, resolving these scales numerically represents a significant challenge. Generally, in primary atomization two of the three common approaches are most used. These are direct numerical simulation (DNS) and large eddy simulation (LES) however Reynolds-Averaged Navier-Stokes (RANS) receives occasional attention (Gorokhovski & Herrmann, 2008).

DNS directly solves the Navier-Stokes equations with the aim of resolving all time and length scales. In single phase flows generally, the smallest scale to be resolved is the Kolmogorov length scale. While resolving this scale is also required in multiphase flows an additional scale that corresponds to the smallest liquid structure must also be resolved by the computational mesh. As the smallest liquid structure will tend to zero in the location of pinch off from the liquid core some modelling has to be introduced even in DNS. Gorokhovski & Herrmann (2008) stated that with these models the smallest droplets produced should be resolved by between two and five mesh cells. This leads to computational grids in the billions of cells to model the primary breakup regime. Computational resources have only recently become available to tackle simulations on this scale.

As DNS is so computationally expensive, significant effort has been placed into using LES as an alternative to achieve investigations that are affordable. Indeed, often cited examples of important work in the field e.g. (Bianchi, et al., 2007) are strictly LES modelled approaches. Though really these are often considered as "under-resolved DNS". Approaches such as these are based on the LES formulation for a single phase. The aim of single phase LES is to reduce the computational cost by ignoring the smallest length scales by filtering the Navier-Stokes equations. The effects of these small length scales are then modelled with sub-grid scale models such as that proposed by Smagorinsky (1963). However, the common interface tracking methods such as those described above can only track interfaces of the size of the grid. Therefore, in this configuration the two distinct phases are modelled with LES while the interface between them is described using under resolved DNS this requires mesh independence studies. However, investigations such as Chesnel et al., (2011) have developed sub-grid models for interface tracking and singularity treatment. This constitutes full LES modelling and their results showed only weak grid dependency.

RANS is the cheapest turbulence modelling method. It averages the flow over time and models fluctuations with additional transport equations. Its use in primary atomization is limited as it is not considered accurate enough to give useful information about the flow field (Jiang, et al., 2010).

2.5 Previous Primary Atomization Modelling Investigation

The most important modelling investigations found in the literature are those whose fluid parameters closely represent real atomization cases. The area that has received most attention is a round jet injected into quiescent air. An early investigation by De Villiers et al., (2004) used an LES and VOF approach, though with relatively coarse grids, on diesel spray breakup. The investigation found spray angle and drop size distributions, though these varied significantly with a finer grid. Bianchi et al., (2005) and Bianchi et al., (2007) used finer grids, again with an LES and VOF method on a similar diesel spray. By simulating part of the injector, they investigated its influence on the resulting spray parameters, showing there was a significant impact on the intact liquid core length. However, there is no grid convergence information to show how well the flow is resolved. By coupling CLSVOF with the ghost fluid method Menard et al., (2007) presented results for diesel like spray though Gorokhovski & Herrmann (2008) suggest that the mesh used was too coarse to be considered full DNS. Desjardins et al., (2008) developed a combined level set and ghost fluid method to reduce mass loss from the level set method. This was then used to investigate diesel like spray, though as with previous investigations no grid refinement is performed. Therefore, while it shows many different shapes and sizes of liquid structures it may still be under resolved. Sander & Weigand (2008) concentrated more on investigating the effects of the nozzle on liquid sheet breakup. Several simple nozzle types were evaluated as well as different velocity profiles that might be caused by turbulence in the nozzle. It was reported that the level of kinetic energy at the inlet had the most significant influence on the sheet breakup. In order to reduce to some extent, the number of grid cells required in theses complex flow fields Fuster et al., (2009) used an octree adaptive grid method, combined with VOF, reporting several results on atomization that agreed with experimental results.

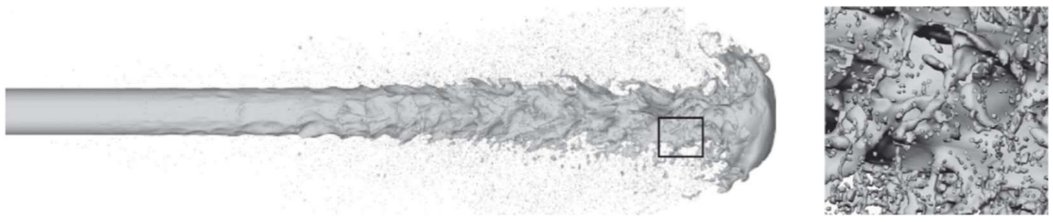


Figure 9: Liquid jet example taken from (Shinjo & Umemura, 2010)

The series by Shinjo and Umemura (Shinjo & Umemura, 2010), (Shinjo & Umemura, 2011) and (Shinjo & Umemura, 2011) is often cited as a benchmark simulation in this field. In this investigation the CLSVOF method for interface tracking, with the CSF method for evaluating surface tension forces, was used on a round liquid jet on a highly resolved grid, of the order of 0.35μm. The aim of the investigation was a "cause and effect" analysis of the effects of the jet tip on atomization. Therefore, the effects of the injector are ignored. Figure 9 shows the extensive ligament and droplet structures emanating from the tip but also a significant breakup in the jet behind it. It is suggested in this configuration of short injection diesel like spray the Tollmien-Schlichting instability is responsible for breakup. The computational cost of this investigation was huge, requiring 5760 CPU cores for over 2 weeks, illustrating the need to reduce the computational cost of these models. Desjardins et al., (2013) again used their level set and ghost fluid method to this time investigate the effects of changing surface tension coefficient. Showing that with an increased surface tension, as is often found in bio-fuels, much larger drops are generated after atomization. Salvador et al., (2016) presented an investigation into a diesel like flow configuration very similar to that used by Shinjo & Umemura (2010). However, they used an octree adaptive grid refinement system with the VOF and CSF methods. This reduced cell count down to just 12.8 million though it is stated there are still some mesh convergence issues. Ghiji et al., (2016) presented qualitative comparisons between VOF and LES simulations and experimental images. Despite the finest mesh used showing similarities to the experimental results significant sensitivity to cell size was found. However, the fluid properties are identical to those that would be found in real diesel applications. Grosshans et al., (2016) used the VOF and CSF methodologies with LES turbulence modelling to study varying physical and numerical parameters of a liquid jet atomizing in still air. While the Weber and Reynolds numbers place it clearly in the atomization regime and the jet is described as diesel like the density and viscosity ratios are significantly reduced for numerical stability. Despite this it is shown that varying density ratio has insignificant effect on the jet breakup. However, reducing the viscosity ratio resulted in smaller droplets. Finally, some injector effects were assessed, showing that in-nozzle turbulence and cavitation bubbles caused the liquid core to break up faster.

2.6 Computational Cost

From the litriture it is clear that multiphase modelling and in particular modelling primary atomization of fuel spray has a very high computational cost. This stems from not only needing to resolve the Kolmogorov length scales, as in single phase DNS, but also needing to resolve liquid breakup. This final criterion tends to zero therby requiring extrememly small cell sizes. This consequently leads to needing cell counts in the hundreds of milliions or even billions to solve even modest domain dimensions. As the next chapter will show GPUs have key computing metrics six or more times that of the CPUs currently available, often for less than double the power consumption and at a comparable price. In addition, their highly parallel nature makes them ideally suited to the very high cell counts commonly found in primary atomization simulation.

Chapter 3 GPU Computing

3.1 Overview

This chapter will give an introduction to GPU computing. Beginning with a history of how computing has evolved over the past decades. This provides a background to why the need for GPU computing has arisen. Next an overview of the architectural differences between GPUs and CPUs is given. Following this a general explanation of CUDA is given to provide context of how the coding model works. Finally, a discussion of previous work using GPUs for code acceleration is presented which culminates in assessing GPU usage in computational fluid dynamics.

3.2 History of Computing

Since personal computers began to become a normal occurrence in both the home and the work place, in the early 1980s, manufacturers have been in a quest for ever greater computational power. As the computational power grows, the requirements of software run on that hardware grows. To this end manufacturers, like Intel and AMD (Advanced Micro Devices), managed until recently to pack greater numbers of transistors running at higher clock frequencies into their CPUs.

As predicted by Moore's law (Moore, 1975) the number of components that could be placed onto a dense integrated circuit, at the same cost, doubled every two years shown in Figure 10. By combining this doubling of transistors on a circuit with increased transistor speed came the popular prediction of David House that CPU performance would double every 18 months.

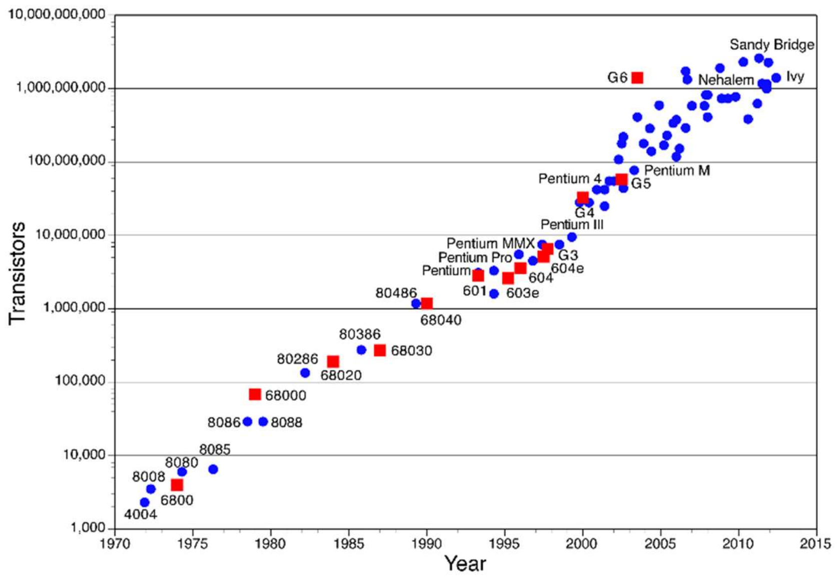


Figure 10: Illustration of Moore's Law (Assured Systems, 2016)

However, this relentless doubling of components could not be sustained. In the early 2000's the industry hit what was known as the "power wall". This came about by the larger number of components running at higher frequencies consuming exponentially more power. With power consumption comes heat generation, as the overall size of a CPU has not really changed, dissipating the heat becomes a significant challenge.

This need to dissipate heat lead to a new way of thinking. Instead of trying to run at ever higher speeds but still with the ability to fit more components onto a chip, a new solution had to be found. This solution was to reduce the clock frequency but divide the chip into multiple cores. This meant that each core would run at a lower speed than their predecessors, but the aggregate performance would be greater. Additionally, by running at lower clock frequencies less power would be consumed meaning less heat is generated, therefore there is a lower cooling requirement.

Thus in 2005 both Intel and AMD produced the first multicore CPUs. Intel, the Pentium D and AMD the Athlon 64 X2. Since these releases the number of cores packed onto a single CPU has become the new battle ground in microprocessor development. Currently high-end CPUs for server applications may have as many as

24 cores. Such is now the proliferation of multicore CPUs, they can be found in anything from mobile phones to games consoles to supercomputers.

This change in hardware configuration also caused software developers to change their way of programming. Previously only programs that used massive supercomputers had to be concerned with producing programs that ran in parallel. Most programs were simply written to run code in serial (one instruction after another). But with the advent of new multi core CPUs this was no longer efficient. Instead programmers had to find ways to introduce parallelism into their code. This gives rise to parallel computing.

There are a number of options for parallelising code but the one most commonly found in computational fluid dynamics is decomposition. In this method the problem is segmented into a number of smaller sections with each being solved on one processor or core. The boundaries between these sections will then communicate their values between cores. This inter processor communication is often what limits parallel computing. As such parallel computing codes are often benchmarked using parallel scaling. This can take the form of either strong or weak scaling performance. In strong scaling the problem size is kept the same and the number of cores is increased. The reduction in compute time obtained by increasing the number of cores used is then compared to ideal scaling. In ideal scaling the compute time would be reduced by the number of cores used. In other words, using four times as many cores would mean compute time becomes ¼ of what it would be on one core. Often in engineering instead of just comparing raw compute times the parallel efficiency is compared. This is defined by

$$E = \frac{1}{N}\frac{T_1}{T_p} \tag{6}$$

where E is the parallel efficiency, N is the number of cores, T_1 is the computational time for one core and T_p the time for said number of cores. Getting a parallel efficiency as close to 100% as possible is desirable. Values of above 100% are possible though uncommon, this is known as super linear scaling. This comes about generally by using more CPUs rather than cores, this increase in CPUs has a corresponding increase in cache size. So, for a fixed problem size more of the problem can reside in high speed cache which will reduce memory access times therefore reducing compute times.

Weak scaling is less commonly used as a comparison in parallel computing, here the problem size per core is kept constant. This means that with any increase in core count there is a corresponding increase in problem size. In this case the desirable outcome is that time to solution will remain constant across all core counts and problem sizes.

3.3 Emergence of General Purpose Graphics Processor Computing (GPGPU)

GPUs began emerging in the late 90s. First being used in gaming consoles that required hardware accelerated 3D images. These concepts began to move into computers giving the graphical user interface everyone has become accustomed to. Because of this requirement to generate images in fractions of a second GPUs were developed into very high throughput orientated processors. As an image is made up of many pixels GPUs developed into highly parallel compute units. All pixels are computed in parallel and returned to the display. Early GPUs weren't programmable so were restricted to just outputting images to displays. This began to change in the early 1990s with the use of graphics programming languages like OpenGL (Khronos Group, 2017), Direct3D (Microsoft, 2017) and Cg (Nvidia Corperation, 2017). It became possible to convert mathematical operations to a series of colour transformations. But this was very convoluted and so uptake was minimal.

In the early 2000s the advent of programming languages specifically aimed at harnessing the power of GPUs saw the first real scientific computing programs run on GPUs. While Sh/RapidMind (McCool & Du Toit, 2004) and BrookGPU (Buck, et al., 2004) meant GPU accelerated programs now didn't have to be written in two languages, using a completely new programming language still represented a barrier.

In 2006 NVidia released Compute unified device architecture (CUDA) (Nvidia Corperation, 2017) this caused the use of GPUs in general scientific and mathematical computing to explode. In contrast to previous GPU programming methods, CUDA was built as an extension to C/C++ and the scientific language Fortran. This difference allowed much easier integration into existing CPU codes or the development of specific GPU code without huge amounts of new knowledge. This step change in the access to the power of GPUs led to a flurry of advances in numerous areas such as; physics, maths and chemistry.

With all its benefits, CUDA still has one major drawback, it is proprietary technology owned by NVidia and so can only be used on NVidia GPUs. But with the scale of the uptake of GPGPU computing CUDA had created, several vendors came together in 2008 and developed the OpenCL standard (The Khronos Group, 2012). While OpenCL is cross platform and so works on any GPU, or any massively parallel device, it is generally accepted that it is not as stable or performs as well as CUDA. For this reason, the present work uses CUDA.

3.4 GPU hardware

A GPU can generally be found attached to a graphics card. This graphics card will interface with the motherboard of a computer by way of an input output slot. This I/O slot is generally PCIe. Also included on the graphics card will be an amount of DRAM (Dynamic Random-Access Memory), currently up to 24GB, as well as cooling and other components. As previously mentioned GPUs are generally manufactured by NVidia and AMD. Because the CUDA API which is only available on NVidia GPUs is the most mature and best performing option only NVidia GPUs will be described.

Looking at the GPU chip in detail will show why it is such an attractive proposition to high performance computing.

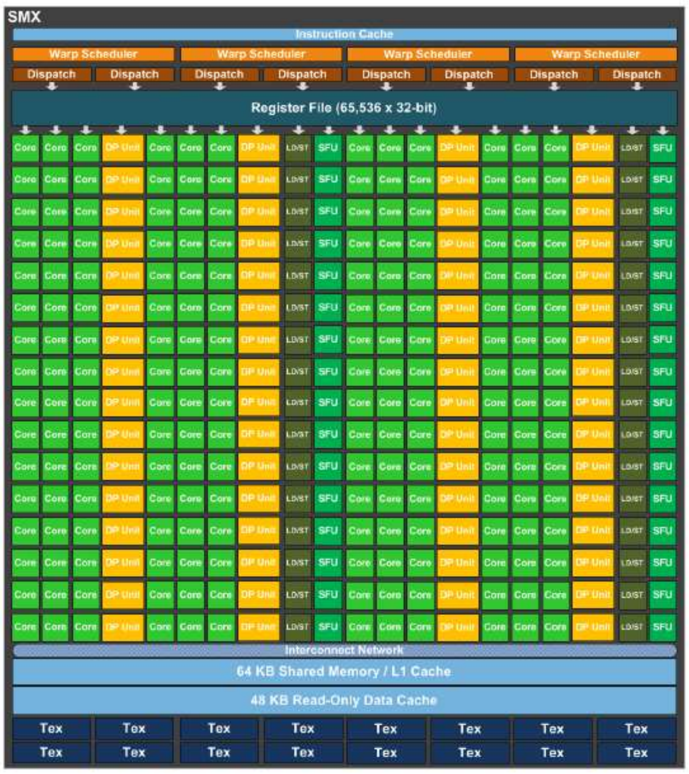
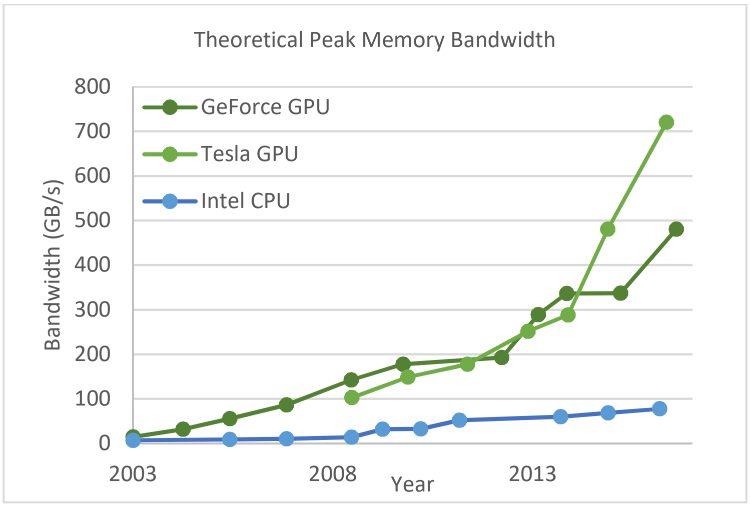

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 11: Block diagram of a streaming multiprocessor in an NVidia Kepler GPU (NVidia Coorperation, 2012)

Figure 11, above, details part of an NVidia Kepler GPU chip (specifically the NVidia titan black edition). What is featured is a streaming multiprocessor. Each chip has 15 Streaming Multiprocessors (SM) as well as six 64-bit memory controllers. These two main components are what sets a GPU apart from a CPU. Each SM controls 192 cores, each of which is capable of performing two single precision calculations per clock cycle. Coupled with each group of three cores is a double precision unit, giving 64 double precision units with two operations per cycle. This means in total theoretically this GPU is capable of 5121 GFLOPS in single precision or 1707 GFLOPS in double precision. The six memory controllers coupled with high clock frequency GDDR5 RAM also mean this GPU is capable of accessing its global memory at 336GB/s. Both figures are many times greater than the most powerful CPUs, never mind ones available at an equivalent price as illustrated in Figure 12.



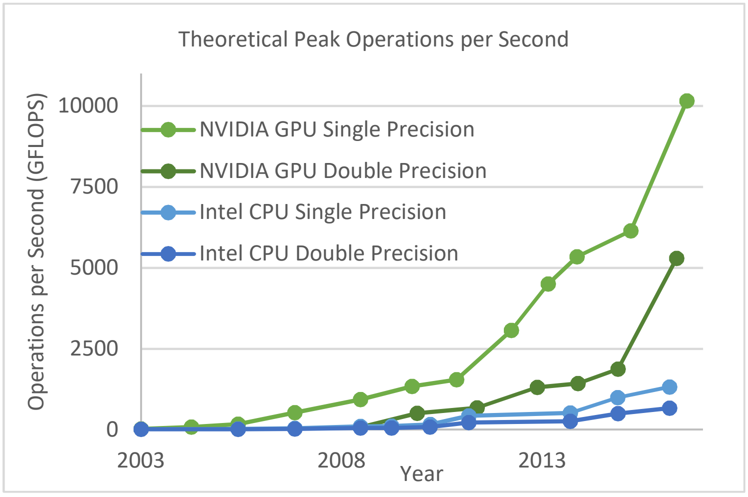


Figure 12: Comparison between CPU and GPU in terms of compute power and memory bandwidth (Nvidia Corperation, 2017)

The cores mentioned earlier all have the ability to execute a thread. Which is a collection of sequential code. But because the GPU is throughput orientated each core in a streaming multiprocessor must perform the same instruction. If the next instruction is different it must wait until all cores have completed the previous instruction. This is the so called the Single Instruction Multiple thread (SIMT) model. When cores need to execute different instructions from one and other so called ‘thread divergence’ is reached that can significantly harm performance.

Warps are a group of 32 threads that all access memory at the same time. In current NVidia GPUs the SM may run a number of warps at once. In order to hide long global memory access times some warps will be accessing memory while others are performing compute functions.

There are also a number of additional interesting features that make GPUs highly attractive massively parallel compute units. The first of these is the copy engine, this is the part of the GPU that deals with transferring data between the CPU and GPU. Having a copy engine that is separate from the compute engine means memory can be copied asynchronously to calculation. Some GPUs have one copy engine allowing calculation and copy either to the device or host while others have two copy engines allowing simultaneous copy from host to device, device to host and calculation. Of course, the data that's being used in the calculation must reside on the device to be used! As well as asynchronous copy and compute it is also possible to perform asynchronous computes, with the GPU scheduling them for best performance. Also, all code executed on the GPU is done so asynchronously to the CPU. Therefore, calculations can be performed on the CPU at the same time as the GPU.

3.5 CUDA Overview

Kernels are probably the most important part of CUDA. The CUDA programming guide defines them as "Functions … when called, are executed N times in parallel by N different CUDA threads." Therefore, kernels are a set of instructions executed in parallel, syntax is provided that indexes threads and blocks. A simple example is shown below:

```
__global__ void add(double* A, double* B, double* C)
{
        Int idx = threadIdx.x;
        C[idx] = A[idx] + B[idx];
}
```

Code 1: Example of a CUDA kernel

This simple kernel will add together the vectors A and B to give a result C. The *__global__* identifier signifies that the function is to be run on the GPU. The idx value is assigned as the thread index. The final line then performs the addition. With thread zero adding the first element of vector A to the first element of vector B with the answer being written as the first element of vector C. Thread one then adds the second elements and so on. It should be noted that these operations will not happen one after the other but in parallel and in any order.

Now that we have a kernel there is a specific way of executing this kernel. The following is an example using the kernel above.

```
Int main()
{
        Add<<<1, 32>>>(A, B, C);
}
```

Code 2: Example of a CUDA kernel launch

While this call is very similar to a C++ function call there is one key difference which is the numbers between the more-than and less-than signs. These numbers define the number of blocks and the number of threads in a block respectively. All threads in a block will be executed by the same SM, therefore to get best throughput the number of threads in a block should be optimised.

Before the execution of a kernel data must be transferred to the GPU. The results can then be transferred back to the CPU. This function is almost identical to the C++ memcpy function but with an additional parameter providing the direction the memory is copied, the options are host to device, device to host and device to device.

Blocks and threads were mentioned previously as well as that the number of threads in a block is user defined. But the question remains as to how to define these values. The first consideration is that there are limits defined by the GPU that is being used. To aid this explanation some details of the NVidia GPU used in this work are detailed below:

Memory – 6GB
CUDA Cores – 2880
Streaming multiprocessors – 15
Maximum threads per multiprocessor – 2048
Maximum threads per block – 1024
Maximum number of concurrent blocks per SM – 16
Maximum number of blocks per kernel - 2^{63}

As the maximum number of blocks is effectively infinite there is no need to restrict their number and enough should be used to cover all the data for operation. The ideal way to ensure maximum occupancy is to make sure that the maximum number of threads per streaming multiprocessor is reached. In this example the number of threads per block that will achieve full occupancy could be any of a number of options. In addition to this there are also limits on the number of registers and shared memory per SM. All of these things considered together is what will decide the best number of threads per block.

3.6 GPU Drawbacks

While GPUs present huge opportunities to improve scientific computing, they do have some limitations.

The most crucial aspect is that because GPUs are massively parallel compute units, running using the SIMT model, algorithms must be selected accordingly. Sometimes an efficient algorithm that is inherently sequential may perform poorly on a GPU but well on a CPU however a less efficient algorithm on CPUs when ported to a GPU may become much faster.

Because of the number of cores available to a GPU program special care must be taken to ensure that all of those threads are kept occupied. If this is not achieved, then the high latency memory access time cannot be hidden, and this will cause poor performance. In CPU programming it is common to only use as many threads as there are cores available. However, in the case of GPU programming it is generally better to use as many threads as possible given the data set being operated on. This should ensure that the GPU threads are filled and memory latency can be hidden.

Data transfer between CPU and GPU is a necessity but is costly. PCIe bandwidth runs at approximately 12GB/s, which is five times less than CPU memory bandwidth and about 30 times less than GPU bandwidth. Therefore, large memory transfers followed by short calculations should be avoided.

3.7 GPU Computing Review

As mentioned previously GPUs have been around for a few decades but their use in general purpose computing is still fairly new. One of the earliest real uses of General Purpose Graphics Processor (GPGPU) computing was by Thompson et al., (2002). They developed a framework that allowed them to use the early desktop GPUs for matrix multiplication. This resulted in a speed up of 3.2 times compared to CPUs of the day.

Molecular mechanics was one of the first fields to take advantage of the opportunities CUDA presented. Stone et al., (2007) showed that when applied to individual algorithms noticeable speed gains could be found. When applied to direct coulomb summation they could evaluate nearly 41 times as many atoms per second. They also investigated accelerating Molecular Dynamics Force Evaluation in the molecular dynamics code: NAMD (Nanoscale Molecular Dynamics). This was compared in a cluster configuration with multiple GPUs compared to multiple CPU cores. Results showed that a GPU performed about five times as fast as CPU cores. Anderson et al., (2008) claimed the first development of a general purpose molecular dynamics code run entirely on GPUs. Their tests were conducted on a cluster of dual core CPUs, approximately 18 months older than the GPU. Nevertheless, it was shown that the GPU implementation performed as fast as 36 processor cores. However, running in single precision (required by the GPU) did show greater errors appearing in the results. Friedrichs et al., (2009) reported that by developing a complete implementation of all-atom protein molecular dynamics and comparing it to Assisted Model Building with Energy Refinement (AMBER) run on a single core of a CPU an acceleration of 700 times could be seen. Interestingly originally this was developed in BrookGPU to run on an ATI GPU but this was found to perform poorly compared to the equivalent in CUDA. GPU computing has also been used to benefit financial modelling, resulting in a step change of how it's used. The massive speed ups of around 100 times have meant that options can be priced in real time as opposed to a couple of times per day. Fatone et al., (2012) showed the effects the GPU can have on performing black Scholes pricing as well as European options. Showing a speed up of 20 and 125 times respectively compared to serial CPU operation. Abbas-Turki et al., (2014) also reported on GPU acceleration of financial pricing this time using the Monte Carlo method. Comparisons were made pricing European options on a compute cluster. GPU results showed a speed improvement of about 40 times over equivalent numbers of quad core CPUs with linear scaling. Also compared is the power consumption of the system running in CPU and GPU mode. Because of the reduction in compute time an approximate 50 times power saving is made.

There has also been investigation into elementary mathematics problems. A common example of this is the work on sparse matrix-vector multiplication by Bell & Garland (2009) comparing their GPU implementation to single core CPUs showed up to 12 times improvement. This was from an average over a range of matrix densities and sizes. Additionally, analysis of memory bandwidth usage and computing performance in FLOPS was made.

3.8 GPU Computing in Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) has been a recent addition to the selection of GPU accelerated methods. The strategies used in accelerating CFD have taken two main avenues. The first avenue is to completely re-write a CFD code to run entirely on GPUs. While these often produce the most significant increases in speed they come at the cost of generally taking much longer to write as well as often being specialised to a particular problem. This is known as a full port. The second avenue is to accelerate sections of the CFD code known as a partial port. It is common that a CFD code used in a specific situation will have a small segment that accounts for the vast majority of the computational time expended to solve it. Therefore, a partial port offers the best compromise between biggest performance gains and the cost of code re-writing.

Because of the reduced time input partial ports have received the most attention. Shi et al., (2012) reported that when working with detailed chemical kinetics that often account for 90% of the simulation time, a speed up of up to 15 times could be found by solving the partial differential equations associated with the chemical kinetics. This speedup was found by comparing the wall clock time on a single core of a current CPU against the performance of the GPU on a number of mesh sizes. Khajeh-Saeed & Perot (2013) showed one of the largest uses of GPU computing in CFD. By accelerating the preconditioned conjugate gradient method, which took 90% of computational expenditure, GPUs performed 25 times faster than the equivalent number of CPU cores. Salvadore (2013) Accelerated a direct numerical simulation of turbulence by again finding that 90% of the computational time was spent evaluating the forcing terms. Speedup was then evaluated in two ways. The first was to compare a single core of a CPU to a GPU. The finding was that the GPU was 11 or 22 times faster depending on the age of the CPU. The second method was to find a similar simulation time by running multiple cores in parallel that would equal the speed of the GPU. This was 32 cores, power consumption was then also compared with the CPU system using 575W and the GPU 240W plus its CPU controller. It should be noted however that this comparison was made with an old (3 years) CPU architecture. Amritkar & Tafti (2016) reported on the partial port of Generalized incompressible direct and large eddy simulation of turbulence (GenIDLEST) where the costly linear solvers were ported to GPUs. Speed testing was conducted on two different cases in both single and double precision. Results showed that speedups of 8 and 4.5 times were possible on the two cases investigated in double precision. It should be noted that again this comparison was made on a CPU 3 years older than the GPU.

Griebel & Zaspel (2010) presented what is recognised to be the first GPU accelerated two-phase flow solver. They accelerated the conjugate gradient (CG) solver used to solve the Poisson equation as well as the re-initialization of the level set function. Using a rising bubble in water case, it was shown that on CG algorithms the GPU performed 16 times faster than a single core of a PC grade CPU. The level set re-initialization received eight times speed up. These also scaled linearly with increasing numbers of GPUs. Combined 8 GPUs performed 70 times faster than a single CPU core as shown in Figure 13.

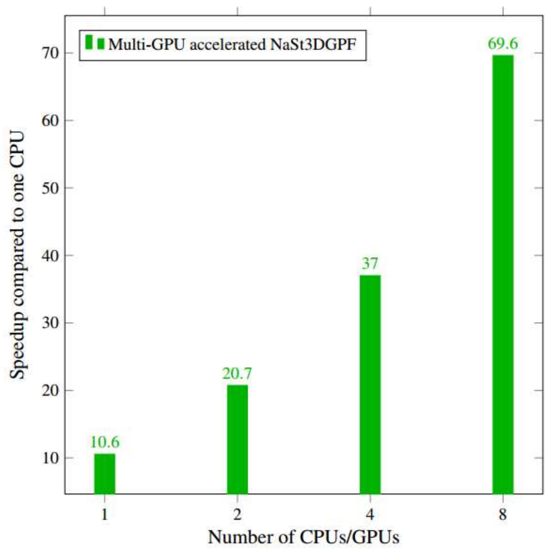


Figure 13: Illustration of the scaling shown by (Griebel & Zaspel, 2010)

Later this was updated so that all the code ran on multiple GPUs (Zaspel & Griebel, 2013). This resulted in an approximately 30% improvement. Instead of comparing to a single CPU core comparisons were made to dual hex-core CPUs that cost a similar amount to the GPU. With the GPU performing about 3.4 times faster makes the GPU equivalently cost efficient.

Elsen (2008) produced a complete port of the Navier-Stokes Stanford University solver and used it to investigate hypersonic flow. Comparing the total time to solution of a single CPU core with a consumer graphics card, on their highest mesh size, a speed up of 20 times was obtained. As this was an early GPU acceleration it used a hybrid of old shading programming and new direct coding inclusion (BrookGPU). There were also some investigations into the usage of the GPUs available performance in both memory bandwidth and FLOPS. Shinn & Vanka (2010) developed a GPU implementation of the Semi-Implicit Method for Pressure Linked Equations (SIMPLE) algorithm and compared it to the equivalent CPU version. With tests on various mesh sizes of a lid driven cavity flow the GPU was about 10 times faster than the single core CPU. Philips et al., (2010) showed an acceleration specifically aimed at use with GPU clusters. All of the flow solver is ported to the GPU, CPUs are used only for transfer between decomposed domain sections. When compared between 8 GPUs and 8 quad core CPUs the GPU implementation was about 5.8 times faster. By computing half a block of the domain at a time, the data transfer required could be overlapped with Message Passing Interface (MPI) communication. This resulted in the GPU becoming 9 times faster. Xu et al., (2014) presented an interesting solution to acceleration with GPUs. First by comparing single GPU performance to a pair of CPUs a speed up of 1.3 was found. The interesting point is that by using a collaborative GPU-CPU approach with load balancing algorithms it was possible to raise maximum problem size per node by 2.3 times as well as showing a 45% speed improvement over the GPU only approach. This was then used to study very large industrial problems showing a parallel efficiency of over 60% on 1024 nodes.

3.9 GPU usage in CFD for multiphase flow

As mentioned previously Griebel & Zaspel (2010) is considered to be the first usage of GPUs to accelerate multiphase CFD. Generally, the level-set method has received the most attention. Appleyard & Drikakis (2011) showed a 92 times increase in performance in solving the level set equation on GPUs over CPUs, though this is in single precision. Liang et al., (2014) developed an implementation of the seven-equation compressible two-phase model running effectively just on a GPU. A speed up over a single CPU core of 34 times was found, interestingly this was on the smallest number of cells tested. Multi GPUs were also tested with 8 GPUs showing a 68% parallel efficiency. Reddy & Banerjee, (2015) presented an acceleration of a volume of fluid (VOF) based solver. The pressure Poisson equation is solved on the GPU using the geometric multigrid method. Comparing serial CPU code running on a CPU 18 months older than the GPU a speedup of six times was found. The speedup was found on a 2D investigation of a liquid sheet breakup. Ikebata & Xiao (2016) accelerated their VOF code again by accelerating the Poisson equation solver on GPUs. Again, by using a multigrid method but this time as a preconditioner to a conjugate gradient method. The speed up results are a little difficult to identify but on a 6 million cell industrial problem an 18 times speed up of the equivalent Poisson solver on an unidentified CPU was found when using two GPUs.

3.10 Comparing GPU and CPU performance

Throughout the literature there is no consistent and fair way of comparing CPUs and GPUs. Comparisons seen range from comparing multi core CPUs with GPUs on performance per price or power consumption to comparing a single CPU core to multiple GPUs. Even in choosing hardware there can often be big differences, for example multiple sources in the literature have compared CPUs that are 3 years older than the GPU. As shown earlier CPUs (and GPUs) have developed according to Moore’s law so this 3-year gap would be equivalent to a 4 times performance difference.

Comparing GPU performance is easier though still there are discrepancies. Seemingly the most common way is to measure the efficient use of the GPUs available performance. Often operations are memory bandwidth limited so it makes sense to evaluate the performance in terms of memory bandwidth or as percentage of available memory bandwidth. Though again comparing different implementations is difficult due to hardware. When comparing implementations using identical hardware a fair comparison can be made. However, when the hardware is different this is more difficult. The main reason for this is that GPUs (especially those from NVidia) come in two categories. The first are consumer GPUs, which are aimed primarily at the computer games market. These are characterised by low cost and generally (but not always) have very poor double precision performance. The second are those specifically aimed at GPGPU computing, these are generally vastly more expensive than their consumer orientated stable mates. This cost does mean that double precision performance is good as well as adding additional features such as ECC memory protection and the ability to control maximum power consumption etc. Generally, memory bandwidth is the same across both categories.

3.11 Chapter Summary

This chapter has presented the fundamentals of GPGPU computing. Specifically, GPUs can be considered as compute units that are capable of performing calculations requiring very high data throughput. This ability comes from their numerous cores. Recently entering the scale of several thousand. Each core is able to perform simple instructions, but groups of cores controlled by a streaming multiprocessor must perform the same instruction. Also discussed was the CUDA programming language that makes the control of these threads possible. Previous investigations into this area were examined. This showed that GPU acceleration is capable of producing speed improvements orders of magnitude greater than CPUs. However, this headline figure is reserved for very specific applications. More generally single digit multiples of speed improvement can be expected. This chapter concluded with an analysis of how comparisons have been made between CPU and GPU accelerated codes demonstrating that there is no clear consensus as to how this should be performed.

Chapter 4 Computing methods

4.1 Overview

Chapter two gave an introduction to numerical methods used in atomization simulation, this chapter presents how they have been implemented in the chosen code: OpenFOAM. Therefore, it will detail governing equations, numerical schemes and solution procedures. Also, discussed are solutions to linear systems of equations which was shown in the previous chapter to be a suitable avenue to explore GPU acceleration. Finally, the hardware and software used throughout this thesis is presented.

4.2 OpenFOAM

First released in 2004, OpenFOAM is an open source toolbox for continuum mechanics. Though mainly known for its use in computational fluid dynamics. Originally written by Henry Weller at Imperial College in C++ to try to improve upon offerings in Fortran. It has proliferated into a multitude of problems found in academic CFD. Its proliferation has been assisted by its wide range of solvers which are applicable to everything from solving large scale fires (Wang, et al., 2011) to coastal engineering (Higuera, et al., 2013) and cavitation in diesel injectors (Salvador, et al., 2010). In addition, being written in C++ means that governing equations can be written in such a way that they closely resemble their mathematical formulation.

As mentioned its extensive library of solvers means it can be applied to many problems. In the current situation, solvers of most interest are the “interFoam” family of solvers. These have the following things in common:

- Eulerian finite volume method for discretization
- Pressure Implicit with Splitting of Operators (PISO) solver scheme
- Volume of fluid (VOF) method for interface capturing
- Surface tension described by the Continuum Surface Method (CSF)

- Optional turbulence modelling using Laminar, RANS or LES models.

4.3 Governing equations

As previously discussed the VOF method (Hirt & Nichols, 1981) uses a scalar to define the location of the two fluids. This scalar, known as the volume fraction, γ and is defined as follows:

$$\gamma \begin{cases} & 1\ is\ fluid\ 1 \\ 0<\gamma & <1\ is\ the\ interface \\ & 0\ is\ fluid\ 2 \end{cases}$$

The motion of a viscous fluid can generally be described by the Navier-Stokes equations (Anderson, 1995) & (Deng, et al., 2000). These are not new and have been in existence since the 1800s. For an incompressible, unsteady and laminar flow these can be written as a volume conservation equation

$$\nabla \cdot \vec{U} = 0 \tag{7}$$

Where $\vec{U}$ is the velocity vector. And a momentum conservation equation

$$\frac{\partial \rho \vec{U}}{\partial t} + \nabla \cdot \left(\rho \vec{U}\vec{U}\right) = -\nabla p + 2\nabla \cdot \left[\mu\, \overleftrightarrow{D}\right] + \rho \vec{g} + \sigma \kappa \nabla \gamma_1 \tag{8}$$

In which t is time, ρ is density, p is the pressure, μ is dynamic viscosity, $\overleftrightarrow{D}$ is the rate of strain tensor, $\vec{g}$ is the acceleration due to gravity and the final term on the RHS is the source term due to surface tension which applies only at the interface between the two fluids. In three dimensions this gives a set of four equations and four unknowns, fluid pressure and three components of velocity. Solving these equations analytically is difficult, indeed finding a solution is one of the millennium prize problems, carrying a $1,000,000 prize (Clay Mathematics Institute, 2017). Because of this they are normally solved numerically.

The rate of strain tensor in equation 8 is defined as

$$\overleftrightarrow{D} = \frac{1}{2}(\nabla\vec{U} + \left(\nabla\vec{U}\right)^T) \tag{9}$$

As the volume of fluid method is based on volume fraction function, combined momentum and continuity equations are solved. In addition, density and viscosity are described as fraction function weighted averages of the fluids in question.

$$\rho = \gamma_1\rho_1 + (1-\gamma_1)\rho_2 \tag{10}$$

$$\mu = \gamma_1\mu_1 + (1-\gamma_1)\mu_2 \tag{11}$$

In which the subscripts denote the fluid. In order to account for the source term of surface tension, $\sigma\kappa\nabla\gamma_1$, the continuum surface method first proposed by Brackbill et al., (1992) is used. This gives a force per unit volume. Of course this expression is only valid when the surface tension is constant, when there is variable surface tension additional shear stress needs to be accounted for. The curvature κ is defined as follows:

$$\kappa = -\nabla \cdot \left(\frac{\nabla\gamma}{|\nabla\gamma|}\right) \tag{12}$$

The transport equation for the volume fraction can be written as follows (for simplicity only a single phase is considered)

$$\frac{\partial\gamma}{\partial t} + \nabla \cdot \left(\gamma\vec{U}\right) = 0 \tag{13}$$

In order to maintain a sharp interface OpenFOAM uses an interface compression method that was originally developed by (Jasak & Weller, 1995). This interface compression method is a conservative form of equation 13 which ensures the boundedness of the volume fraction between zero and one.

$$\frac{\partial \gamma}{\partial t} + \nabla \cdot \left(\gamma \vec{U}\right) + \nabla \cdot [\vec{U}_r \gamma (1-\gamma)] = 0 \tag{14}$$

In which $\vec{U}_r$ is the relative velocity between the two phases. Because of the $\gamma(1-\gamma)$ expression in the interface compression term it only acts at the interface. It has been shown this method is accurate in relation to others but at the cost of requiring generally smaller courant numbers (Gopala & van Wachem, 2008). The relative velocity between the two phases is described by:

$$U_r = n_f \, min\left[C_\gamma \frac{|\Phi|}{|S_f|}, max\left(\frac{|\Phi|}{|S_f|}\right)\right] \tag{15}$$

In which n_f is the normal vector of the cell surface, Φ is the mass flux, S_f is the area of the cell surface and C_γ is a user chosen coefficient between zero and four. If zero is chosen, then there will be no interface compression. While a coefficient of one, as used throughout this work, provides a conservative compression. And finally, above one there is an enhanced compression.

Time step control is governed by the Courant number (Courant, et al., 1967) this gives the number of cells the flow will travel in one time step. It is determined by the following equation:

$$Co = \Delta t \sum_{i=1}^{n} \frac{U_i}{\Delta x_i} \tag{16}$$

Δt is the time step, Δx is the mesh size and U is the velocity. The subscript i denotes the direction, in three dimensions this would be x, y and z. In addition to the

Courant number Brackbill et al. (1992) state that to account for capillary waves the capillary time-step constraint should be adhered to.

$$\Delta t^{BKZ} \leq \frac{\Delta x}{2Ca} \tag{17}$$

Where the superscript BKZ denote the authors, Δx is the mesh size and Ca the capillary number described in chapter 2.

4.4 Solution procedure

The finite volume method is characterised by some large volume spit into a mesh of many smaller volumes. The governing equations are integrated over all the mesh elements in the domain. Using the divergence theorem, the equations containing divergence terms are converted to surface integrals, these terms are then evaluated as fluxes at the surface of each mesh volume (Versteeg & Malalasekera, 2007). As such in OpenFOAM notation all discretization schemes are prefixed with 'Gauss' denoting the Gaussian divergence theorem.

OpenFOAM provides numerous options for all categories of discretization. In an OpenFOAM setup they are split into the following sections:

- Time schemes
- Gradient schemes
- Divergence schemes
- Surface normal gradient schemes
- Laplacian schemes
- Interpolation schemes

Discretization for the time derivatives is achieved using the implicit second order Crank-Nicolson (Crank & Nicolson, 1996) scheme. For all gradient terms, the central differencing scheme is used. In all cases for interpolation from cell centres to faces the central differencing scheme is used. The surface normal gradient is calculated using a central differencing method, of the adjacent cell centres, without the available non-orthogonal correction as in all cases in this work non-orthogonality is zero. Laplacian terms again use a central differencing method to evaluate cell face values, the resulting surface normal gradient is then evaluated as previously stated.

Divergence discretization is different for different terms involved. The $\nabla \cdot \rho \vec{U}\vec{U}$ term uses the second order hybrid LinearUpwind scheme (Spalding, 1972). The $\nabla \cdot \left(\gamma \vec{U}\right)$ term uses the second order, total variation diminishing, limited vanLeer scheme (van Leer, 1974). Finally, the interface compression term $\nabla \cdot \vec{U}_r \gamma (1-\gamma)$ uses a central differencing scheme as its boundedness is assured by the MULES algoritham.

The pressure implicit with splitting of operators algorithm (PISO), first proposed by Issa et al., (1986) is used to obtain the solution to the momentum equation. First a discretized momentum predictor equation is solved using a “best-guess” pressure field, often the pressure from the previous time step, to compute an intermediate velocity field. This intermediate velocity field is then used to solve a pressure equation. Finally, the velocity field is corrected using the new values of pressure. The final two steps are performed iteratively until suitably accurate values are found. Issa et al., (1986) states that two pressure corrections are required to be second order accurate. This scheme is summarised in the flow chart in Figure 14.

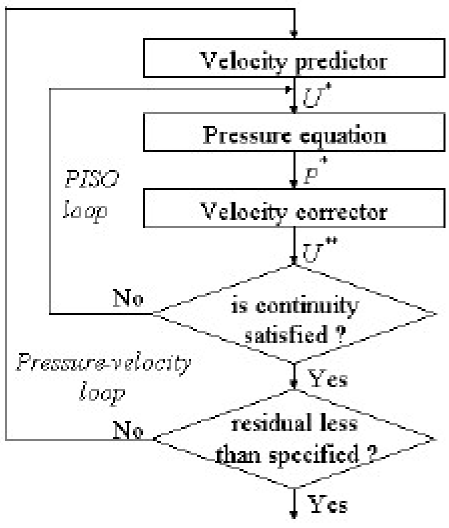


Figure 14: Outline of the PISO Algorithm (Giannopapa & Papadakis , 2007)

Included as part of the PISO algorithm is the multi-dimensional limiter for explicit solution (MULES) (OpenCFD Ltd, 2009). This is used to solve the volume fraction field and has been proven to be effective in producing boundedness in scalar fields, such as the volume fraction. It uses time step sub cycling to obtain the solution to the volume fraction field. This time step sub cycling takes place before the momentum predictor step in the PISO loop. It allows higher courant numbers to be used in the rest of the PISO loop as usually the solution of the phase volume fraction transport equation dictates the size of time step that can be used. At the start of each time step the Courant number is calculated and subsequently the new time step size. The time step sub cycling then uses equal divisions of the new time step size. So, with two sub cycles, as used in the present work, the sub cycle will advance time by exactly half of the time step size.

If the LES or RANS turbulence models are selected these are included in the pressure correction steps. Otherwise a laminar model is used, therefore if the cell size is small enough it can be considered DNS (Hemida, 2008).

Once discretized the momentum and pressure correction equations produce a linear system of equations for all the cells in the domain. This system can be described as the following:

$$Ax = b \tag{18}$$

Here A is a matrix of coefficients, x is unknown, and b is the source term of the governing equations. In OpenFOAM the matrix A is stored in the Lower Diagonal Upper (LDU) sparse matrix format. In this format the non-zero values of the diagonal, upper triangle and lower triangle are stored consecutively in dense vectors. There are then two additional dense vectors that define the location of those non-zero values in a co-ordinate format. The LDU format allows for easier implementation of some of the linear system solvers that are available to use in OpenFOAM.

The linear systems can be solved by one of 3 general methods. The first is to directly solve the system using something like Gaussian elimination, this would give an exact result (if there are no rounding errors). However, these methods are often too expensive even for small to medium sized matrices. Second is to use a stationary method such as the Gauss-Seidel or Jacobi methods. These work using a correction equation based upon the measurement of the error found in the previous iteration. Convergence is only guaranteed for some types of matrices and tends to be relatively slow. The final option is Krylov subspace methods. These methods start with an initial guess and therefore residual which iterate until the exact solution (without rounding errors) is found. Common options are the conjugate gradient or bi-conjugate gradient methods.

OpenFOAM uses all of these categories in some way. They are further divided into two groups. Those for symmetric matrices and those for asymmetric matrices. The most commonly used linear system solver for symmetric matrices in OpenFOAM is generally considered to be the preconditioned conjugate gradient method. For ease of explanation the preconditioner will be ignored to begin with. The conjugate gradient method was first proposed by Hestenes & Stiefel (1952) it was first developed as a direct solver but has found its popularity as an iterative solver.

Indeed, the exact solution of the linear system will be found in as many iterations as there are unknowns as long as there is no rounding. However, when there are as many unknowns as there are cells in the mesh, coupled with rounding errors produced by floating point number storage, this is almost never the case. Instead OpenFOAM uses a hard-coded limit of 1000 iterations. The algorithm below shows the conjugate gradient method. In addition to the matrix and vectors of the linear system, two additional vectors are required. These are the residual and search direction vectors. They are labelled R and P respectively.

```
$R_0 = b - Ax_0$
$P_0 = R_0$
$i = 0$
While I <= 1000
    $\alpha_i = \frac{R_i^T R_i}{P_i^T A P_i}$
    $x_{i+1} = x_i + \alpha_i P_i$
    $R_{i+1} = R_i - \alpha_i A P_i$
    if $R_{i+1}$ is small enough exit loop.
    $\beta_i = \frac{R_{i+1}^T R_{i+1}}{R_i^T R_i}$
    $P_{i+1} = R_{i+1} + \beta_i P_i$
    $i = i + 1$
end while loop
```

The answer to the linear system is then x_{i+1}at the point that R is small enough. The residual R is considered small when the sum of the magnitude of elements is less than a user specified value, this value is generally normalised by the magnitude of vector b. While this method is good, it can take many iterations to come to a converged result, particularly when the linear system is large. To combat this slow convergence a preconditioner is often used to improve convergence. The algorithm is very similar with minor changes. In addition to the storage requirements of the conjugate gradient method its preconditioned variant requires the storage of an additional vector, in this case Z, as well as the inverse of a preconditioning matrix, M. The algorithm is shown below.

$$R_0 = b - Ax_0$$

$$Z_0 = M^{-1}R_0$$

$$P_0 = Z_0$$

$$i = 0$$

While I <= 1000

$$\alpha_i = \frac{R_i^T R_i}{P_i^T A P_i}$$

$$x_{i+1} = x_i + \alpha_i P_i$$

$$R_{i+1} = R_i - \alpha_i A P_i$$

if R_{i+1} is small enough exit loop.

$$Z_{i+1} = M^{-1}R_{i+1}$$

$$\beta_i = \frac{Z_{i+1}^T R_{i+1}}{Z_i^T R_i}$$

$$P_{i+1} = Z_{i+1} + \beta_i P_i$$

$$i = i + 1$$

```
end while loop
```

The simplest preconditioner is the Jacobi or diagonal preconditioner. Here the diagonal of matrix A is used as the preconditioner. It is efficient for diagonally dominant matrices. In OpenFOAM it is common to use the Diagonal Incomplete Cholesky preconditioner.

The final preconditioner of interest is a bit different to those addressed previously. The Geometric-Algebraic Multi-Grid solver (GAMG) can be used either as a preconditioner or as a linear system solver in place of say the preconditioned conjugate gradient solver. GAMG is, as its name suggests, a multigrid method that gives efficient convergence. The advantage of multigrid methods is that the number of operations is proportional to the size of the problem, in contrast to others that increase quadratically or higher. The basic idea of a multigrid method is that classical solvers like the Jacobi method will stall at a certain convergence once high-frequency errors have been removed. So, at this point a coarser grid is constructed where the error is no longer smooth. The transfer of vectors to this coarse grid is known as restriction. This is repeated at coarser and coarser grids until some minimum is reached and a direct solver can be used to find a solution on the most course grid. The solution is then transferred up through finer and finer meshes until the original mesh is reached, with Jacobi smoothing iterations taking place at each level to remove any errors introduced by this interpolation operation.

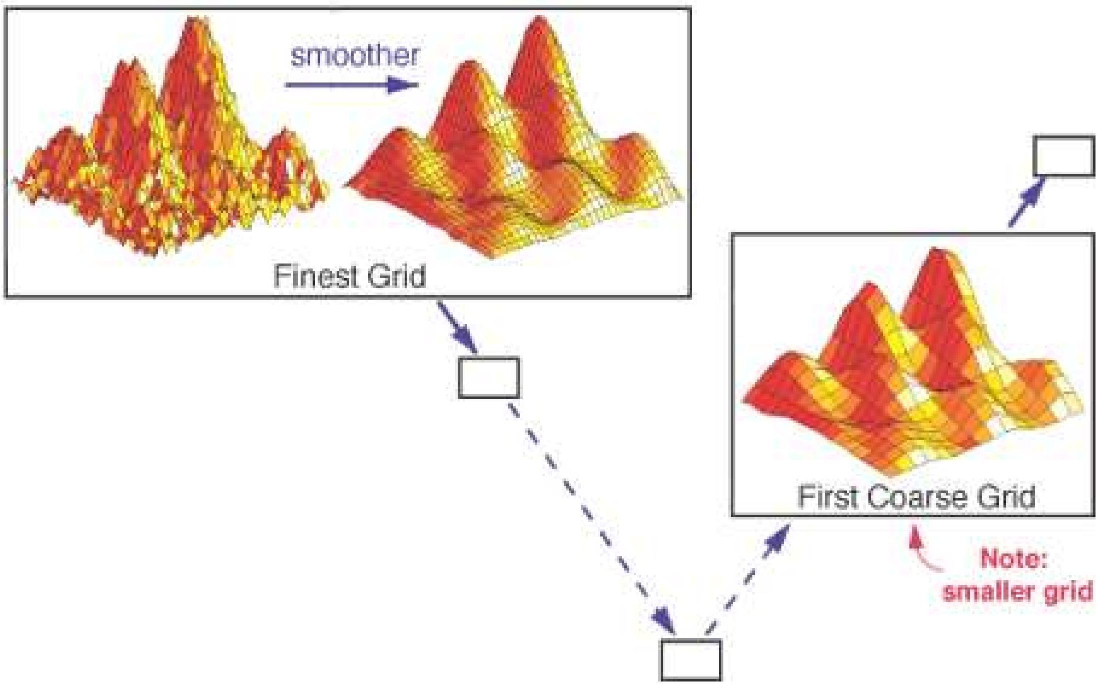


Figure 15: Illustration of the errors removed in a multigrid method (Lawrence Livermore National Laboratory, n.d.)

The multigrid method described above is a specific method that is known as the V cycle, so called due to the shape it makes when the order of grids is lain out as in Figure 15. There are alternatives which can be found in the literature, but this is as used in OpenFOAM. The final element of the multigrid is how the coarse grids are chosen. As the acronym GAMG suggests this can be either geometric or algebraic. The geometric method uses the geometry of the mesh to decide on closely related cells that can be combined into a single coarse grid cell. On the other hand, algebraic methods use only the coefficients of the matrix in the linear system to generate the course grid.

4.5 Computational Hardware

In the following chapters, all OpenFOAM computations have been performed using the same hardware and software stack. Hardware comes in the form of an HP Z820 workstation. Configured with twin Intel Xeon E5-2650 v2 CPUs which have eight cores running at 2.6GHz. 64GB of DDR3-1833 RAM and twin NVidia GeForce Titan Black edition graphics cards. As with all Intel CPUs since the early 2000s they are equipped with hyper threading. This is Intel's proprietary technology for allowing the operating system to address each real core as two logical cores. This allows multiple instructions on different data to be performed at once, commonly useful in general computer use. However, for CFD this is often not helpful as the additional MPI communication outweighs the benefits of possible overlap in computation (Keough, 2014). Key performance parameters of the two compute units are shown in the table below:

Computing Unit	Cost at launch ($)	Theoretical Peak DP performance (GFLOPS)	Memory (GB)	Max. Memory Bandwidth (GB/s)	Max. Power Consumption (W)
Intel Xeon E5-2650 V2 (8 Core)	1166	166.4	Up to 768	59.7	95
NVidia GeForce Titan Black Edition	999	1707	6	336	250

Table 1: Key performance criteria of the CPU and GPU used in this work

Software used consists of the Ubuntu 12.04 operating system. OpenFOAM version 2.3.1 and CUDA 6.0. All code compilation for OpenFOAM is conducted using the GCC open source compiler while CUDA code is compiled using NVCC (NVidia CUDA compiler) included as part of CUDA version 6. Message passing between cores in multi core mode is conducted using OpenMPI.

Of specific interest in the specifications listed in Table 1 is theoretical memory bandwidth. The vast majority of compute time in CFD codes is spent in memory bound linear system solvers. I.e. the performance bottleneck comes from memory bandwidth. With this in mind some analysis of where these specifications come from is useful. For a CPU using DDR RAM (as in this case) the memory bandwidth is evaluated by the product of the transfer rate and the bus data width. In the current case the DDR3 RAM has a transfer of 1.867×10^9 transfers per second with a bus width of 64 bits. This gives a peak memory bandwidth of 14.933 GB/s per module. In order to attain the value of 59.7 GB/s multiple modules are used, as is the case in the current setup, four 8 GB modules are attached to each CPU. This is known as quad channel memory. The GPU is similar, indeed its GDDR5 memory is built upon the same DDR3 technology as the CPU memory. However, it is capable of 7×10^9 transfers per second with a 384-bit bus width, this gives the quoted value of 336 GB/s. Achieving this speed is however generally impossible. Deakin & McIntosh-Smith (2015) showed that the similar architecture NVidia GeForce GTX 780 Ti showed a useable bandwidth of 83.9% of the quoted 336 GB/s maximum.

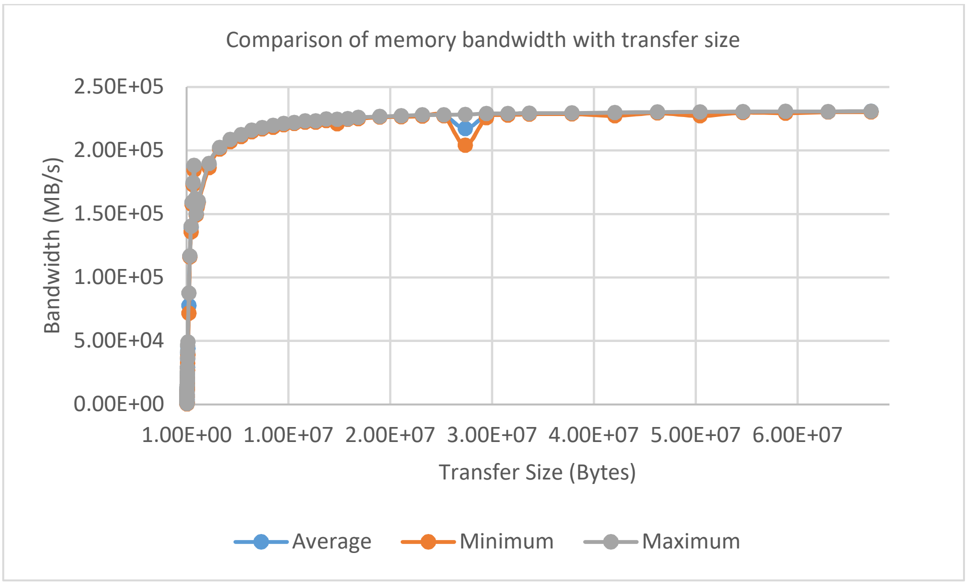


Figure 16: Comparison of GPU memory bandwidth with varying transfer size

To understand the available memory bandwidth of the NVidia GTX Titan Black in the system an investigation was conducted along the lines of that used by Delbosc (2015). Using the CudaMemcpy function an amount of data is copied from one location in the GPU memory to another. This amount of data is varied between 1,024 KB and 64 MB. This test is conducted 10 times with the results displayed in Figure 16. Plotted are minimum, maximum and average bandwidth usage of the 10 runs. Interestingly the maximum bandwidth achieved is only just shy of 231 GB/s which is just 68.75% of the theoretical peak. As suggested by Delbosc (2015) peak bandwidth is only observed above a transfer size of about 32 MB. Transfers below this size steadily reduce in measured bandwidth.

4.6 Chapter Summary

In this chapter an introduction to OpenFOAM has been given, the main point of which is that it is not a monolithic code and instead is a collection of libraries which are combined to makeup solvers for different applications. The solver chosen for this study is interFoam, an incompressible multiphase solver. It uses the volume of fluid interface capturing method with the continuum surface force method to treat singularities. It uses a unique interface compression term to ensure a sharp interface and solves the momentum equation with the PISO method. Additionally, the different linear system solvers were outlined, showing that OpenFOAM makes use of all categories of linear system solvers. Finally, the computational hardware used throughout this study was presented, a brief investigation into the GPU hardware showed useable memory bandwidth is significantly lower than the peak described by the manufacturer.

Chapter 5 Elementary Test Cases

5.1 Overview

In this chapter the OpenFOAM code presented in the previous chapter is assessed to confirm its suitability to be used in complex multiphase problems. Additionally, by investigating these problems the efficiency of the solver can be assessed. By using an efficient CPU based OpenFOAM setup an "apples to apples" comparison can be made with future GPU implementations. This should ensure that GPU developments are beneficial in real world applications.

5.2 Zalesak's Disk

The notched disk in a rotating flow field was first proposed by Zalesak (1979) since then it has been used by numerous researchers to validate the quality of interface capturing methods. While it has been used in numerous configurations only one will be considered here. This configuration is as such, a disk of radius 0.15m which has a notch removed that is 0.05m wide by 0.25m high. This notched disk is then placed into a domain that is 1m square. The disk is then rotated over 6.28s according to the velocity profile in equation 19.

$$u = \frac{\pi}{3.14}(0.5 - y) \text{ and } v = \frac{\pi}{3.14}(x - 0.5) \tag{19}$$

With an initial position of (0.5m, 0.75m), over the course of the problem the disk will complete one whole rotation. The shape of the disk at the end can then be compared with the original to quantify losses associated with the interface capturing method being tested.

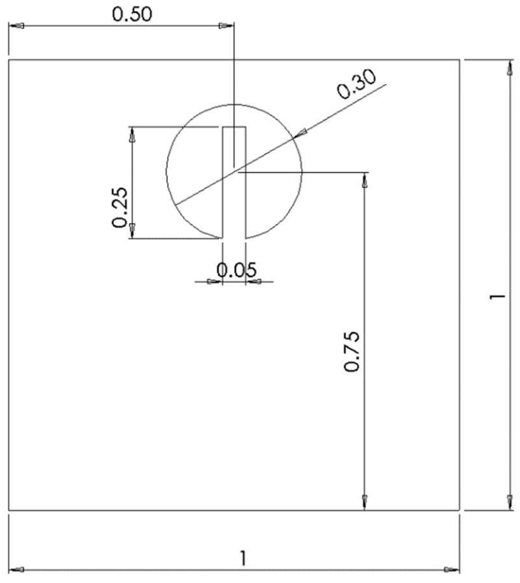


Figure 17: Initial condition of Zalesak's Disk

Three grid resolutions are compared, 100x100, 200x200 and 400x400. In each case the grids are Cartesian. A zero-normal gradient is applied to all boundries. As pressure and velocity calculation is switched off, at the boundry these effectively remain fixed values equal to the initial conditions. To properly compare the interfaces both the momentum and pressure correction equations are switched off. In this case, only the interface tracking equations are solved. The effect on velocity is that it remains constant throughout all time steps. In addition to this the result of the first computed time step is compared to the time step after 6.28s. The effect of this is that the interface is allowed to become a gradient across two or three cells as is common with VOF. This allows the interfaces to be compared on a like for like basis. The results shown below are a contour of volume fraction of 0.5. In all cases below the original interface (and reference solution) is shown by the black line while the final interface is in grey.

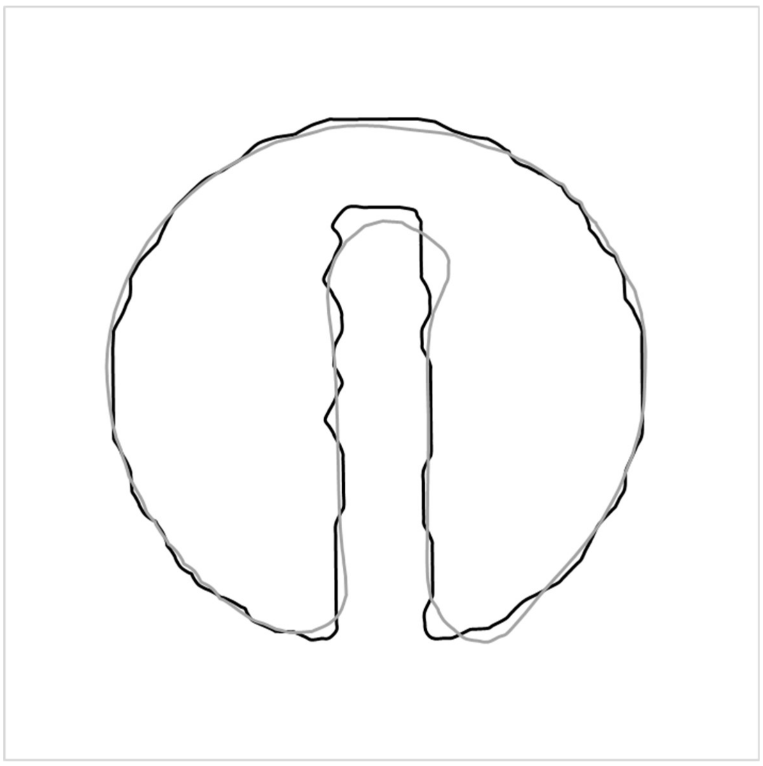


Figure 18: Interface at final time step (grey) compared to initial 100 grid (black)

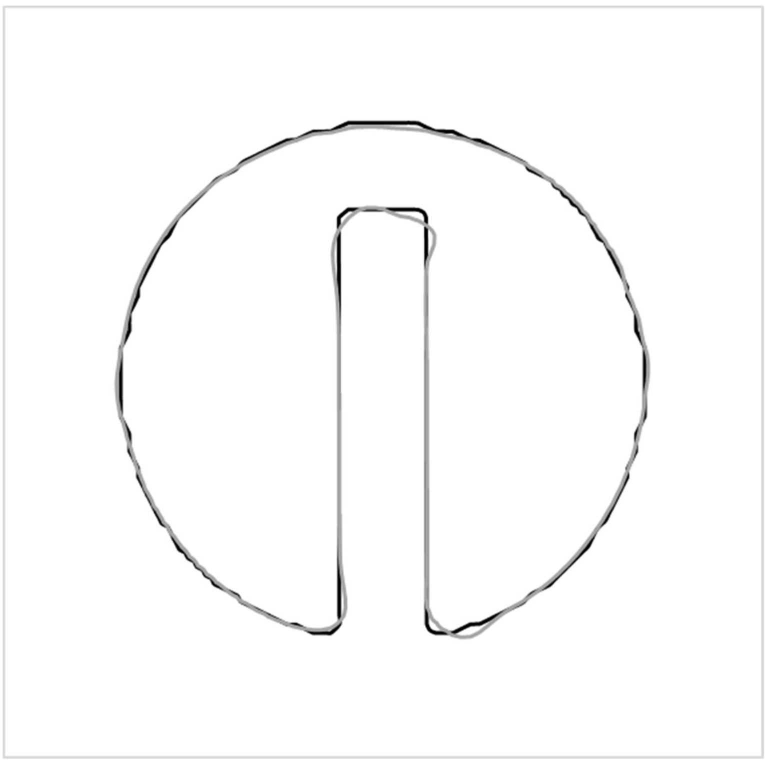


Figure 19: Interface at final time step (grey) compared to initial 200 grid (black)

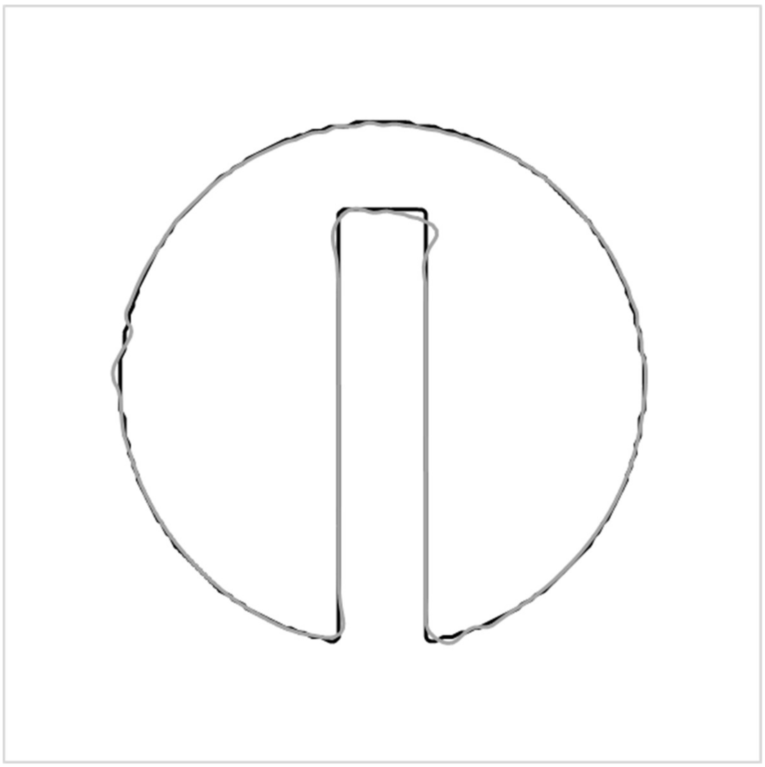


Figure 20: Interface at final time step (grey) compared to initial 400 grid (black)

Figures 18-20 clearly show, especially on the coarsest grid, that the sharp corners of the slot of the disk are difficult to maintain. Apart from the coarsest mesh the interface is quite well maintained. Indeed, it is likely not too far from results that could be achieved with the level set method which doesn't suffer from the numerical diffusion of the interface. To compare results quantitatively the error metric used by several other investigations is used e.g. (Deshpande, et al., 2012) and (Sussman & Puckett, 2000).

$$E = \frac{1}{L}\sum_{1}^{N} A|\gamma^{I} - \gamma^{F}| \tag{20}$$

Here L is the length of the interface. A is the area of a cell. The superscripts I and F are the initial and final quantities. The sum is across all cells in the domain.

100 x 100	200 x 200	400 x 400
0.003979	0.001692	0.000971

Table 2: Values of E (error) for each cell density

The results of this error metric are similar to (Deshpande, et al., 2012) though slightly greater, this is likely due to slight differences in numerical schemes or convergence criteria.

To assess the stability of OpenFOAM the 200 x 200 cell case was used with varying courant number. Five cases were run with the courant number halved in each one. Initially a comparison with the error metric described above can be made.

Courant number	1	0.5	0.25	0.125	0.0625
Error	0.003954	0.002240	0.001691	0.001653	0.001704

Table 3: Results of Courant number investigation

From this investigation it can be clearly seen that after dropping to a courant number of 0.25 there then becomes very little difference in the final interface. To show what kind of difference these errors translate to below is a visual representation of the interfaces.

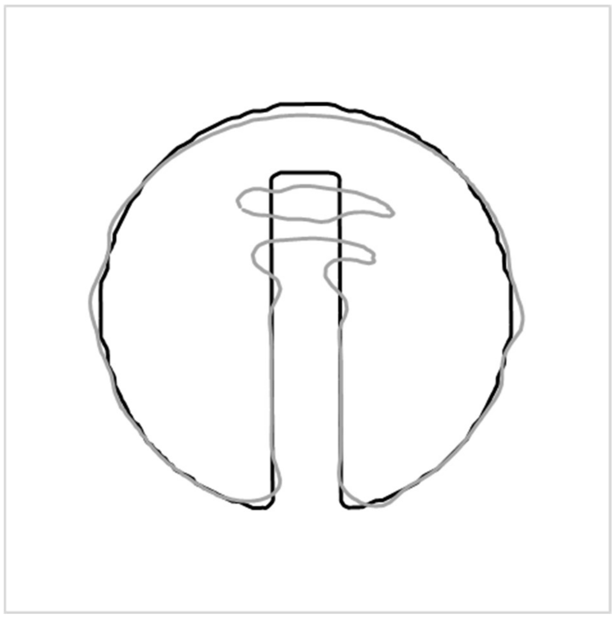


Figure 21: Interface comparison at Courant number of 1 (grey) compared to initial (black)

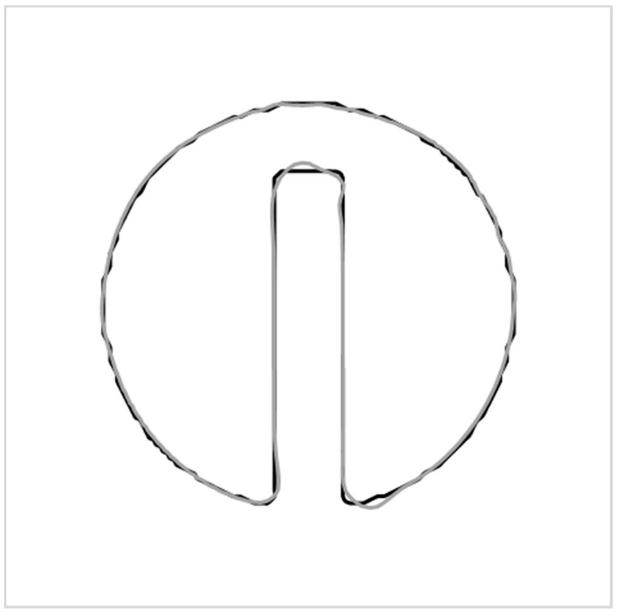


Figure 22: Interface comparison at Courant number of 0.0625 (grey) compared to initial (black)

In figures 21 & 22, as in previous comparisons the initial interface is shown in black while the final interface is shown in grey. These comparisons clearly show that with a courant number of 1, the ability to maintain the interface in this problem is poor. However as mentioned above there is very little difference at a courant number below 0.25.

5.3 Rayleigh-Taylor Instability

The Rayleigh-Taylor instability problem is one that has been used by many researchers in different configurations to investigate the performance of numerous codes (Tryggvason & Unverdi, 1990), (Bell & Marcus, 1992) & (López, et al., 2005). The setup consists of a heavy fluid above a lighter fluid. The interface is initially set as a cosine wave with an amplitude of 0.05. Under the effects of normal gravity, $9.81 m/s^2$, the two fluids will eventually swap places. Of interest though are the initial time steps where the heavier fluid accelerates and becomes a jet like structure.

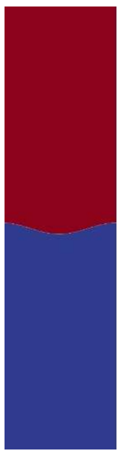


Figure 23: Initial conditions of Rayleigh-Taylor interface problem

In this investigation the two fluids used are air like and helium like shown in Figure 23 as red and blue respectively. The fluid properties are listed in Table 4, the subscripts indicate the fluid, one being air and two the helium.

$\rho_1\ (kg/m^3)$	$\mu_1\ (kg/ms)$	$\rho_2\ (kg/m^3)$	$\mu_2\ (kg/ms)$	σ (N/m)
1.225	3.13×10^{-3}	0.1694	3.13×10^{-3}	1×10^{-2}

Table 4: Rayleigh-Taylor instability fluid parameters

All boundries use a slip condition, where there is zero normal velocity and a zero-normal gradient for other variables. The computational domain is 1 unit by 4. The mesh resolution is expressed in terms of cells per unit length. To understand how OpenFOAM performs in a large range of mesh resolutions 32, 64, 128, 256 and 512 cells are used. As before the grid is Cartesian. As was done by (Herrmann, 2008) up to a time of 0.9s is simulated. This ensures that the flow structures will remain inside the boundaries in all mesh cases. Therefore, the only influence on the results should be the change in mesh resolution. In all cases a constant time stepping regime is used. In all the results that follow the reference solution of 512 cells is shown in black. The solution that the reference is being compared to is then shown in grey.

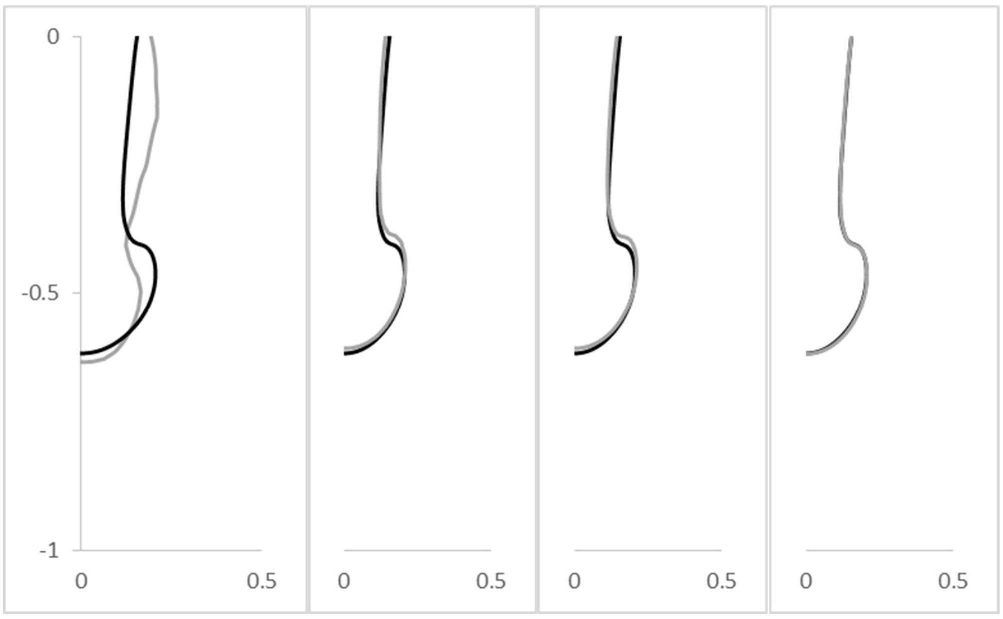


Figure 24: Interface at t=0.6s. From L to R, 32, 64, 128, 256 cells(in grey) per unit length, compared to reference solution (in black)

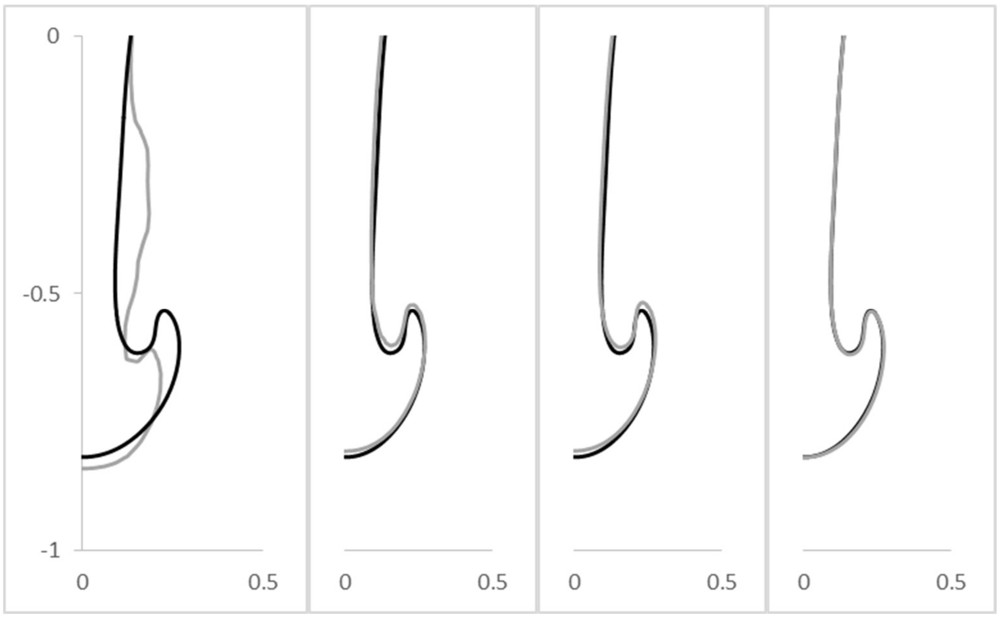


Figure 25: Interface at t=0.7s. From L to R, 32, 64, 128, 256 cells(in grey) per unit length, compared to reference solution (in black)

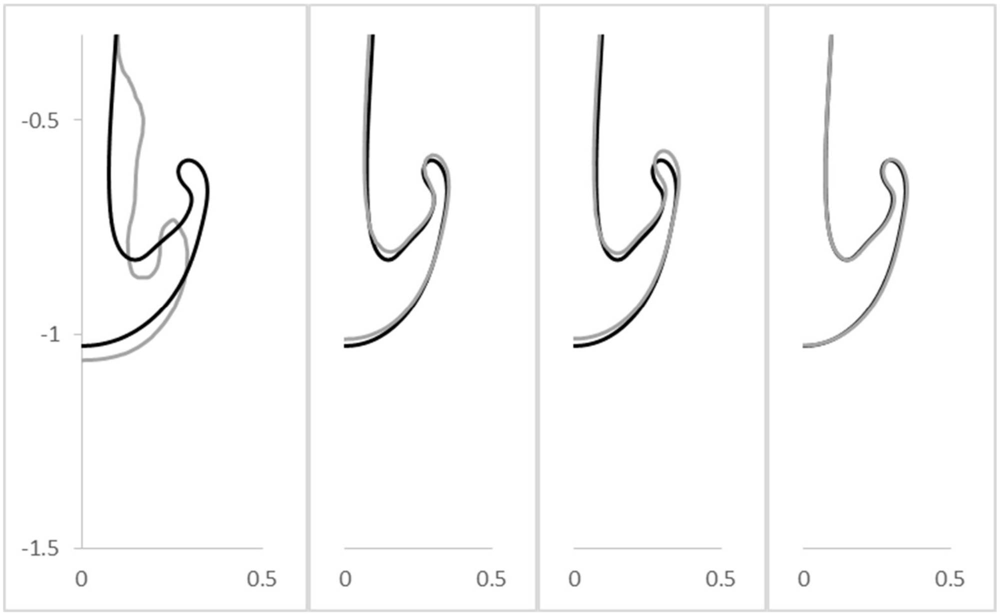


Figure 26: Interface at t=0.8s. From L to R, 32, 64, 128, 256 cells (in grey) per unit length, compared to reference solution (in black)

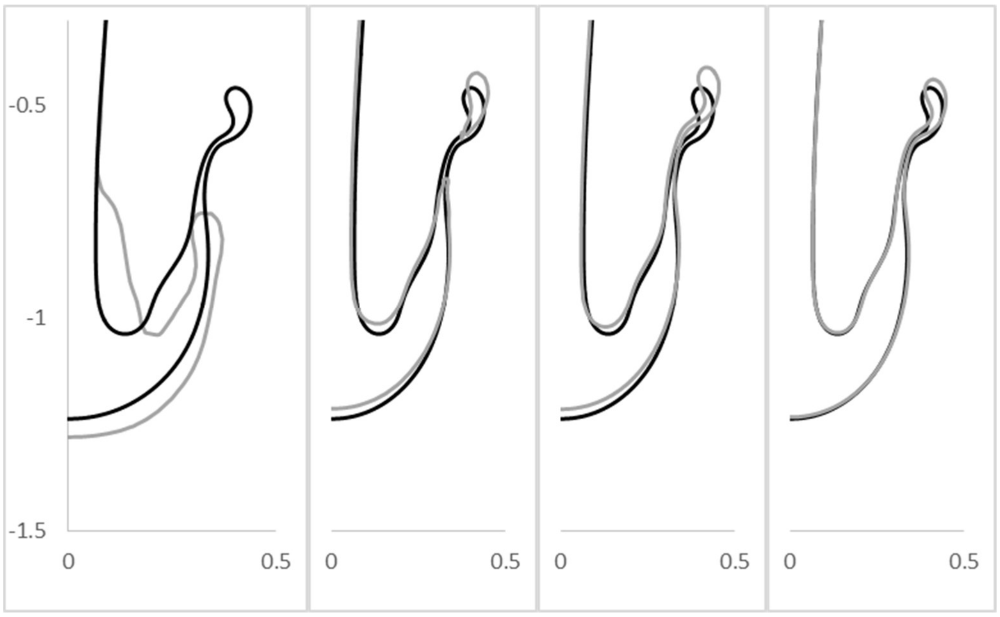


Figure 27: Interface at t=0.9s. From L to R, 32, 64, 128, 256 cells (in grey) per unit length, compared to reference solution (in black)

Individually comparing each grid to the reference gives some interesting conclusions. At all time steps apart from 0.9s there is no discernible difference between the mesh using 256 cells and the reference solution. At 0.9s 256 gives a slight over estimation of both the X and Y co-ordinates of the interface. Again, at all time steps apart from 0.9s both 64 and 128 cells show some slight differences to the reference solution, but these are fairly minor. At 0.9s 128 cells is still close to the reference solution but not as close as 256. The first significant difference to the reference solution comes at 0.9s in the 64-cell case. Here numerical breakup can be observed. While in all denser meshes there is a small thin liquid structure ending in a small droplet shape. However, with 64 cells the mesh is too course to capture this thin ligament. Finally, in all time steps the 32-cell representation represents a poor description compared to the reference.

5.4 Linear Solver Selection

The linear system solvers used can have a significant effect on the compute time of the case in question. As discussed earlier OpenFOAM has a number of different linear system solvers available. To assess the most efficient solver the 128-cell test case was used as a basis. The comparisons would be made among several of the options; conjugate gradient (CG), diagonal preconditioned conjugate gradient (DPCG), diagonal incomplete Cholesky preconditioned conjugate gradient (DICPCG), geometric-algebraic multigrid preconditioned conjugate gradient (GAMGPCG) and just geometric-algebraic multigrid (GAMG). The smooth solver is not compared here as it would likely take several times longer to reach convergence criteria than even the slowest solver presented here.

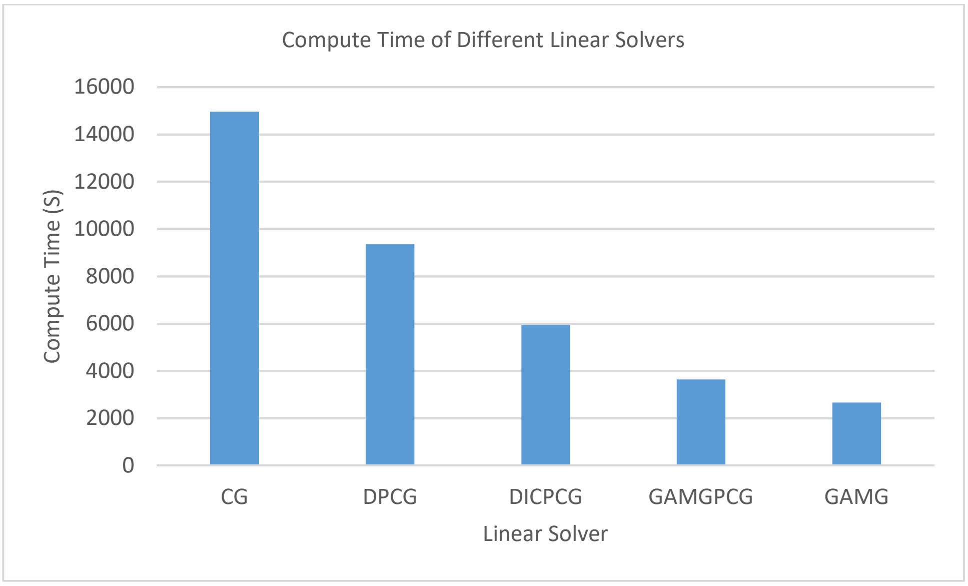


Figure 28: Comparison of compute times of different linear solvers

In each case the case setup was constant, the only differences were the linear system solver used for the pressure correction equations. The pressure correction equation was selected as it is the most expensive code section, indeed in these results the cost of solving the pressure equations was between 80% and 95% of the total computational time. The results come from the measured clock time and so includes some saving data to disk. All computations were done in serial using the test hardware with the GUI switched off.

The results show the huge impact the selection of the linear solver can have on the total computational time and so is a key consideration in generating CPU benchmarks. Indeed, using the GAMG solver reduces the computational cost by over 50% when compared to the commonly used DICPCG solver.

5.5 Rising Bubble

Hysing et al., (2009) proposed a quantitative benchmark to compare incompressible interfacial flow codes. The comparison is made on a 2D bubble rising a in a column of liquid. A diagram of the problem initial conditions is shown in Figure 29.

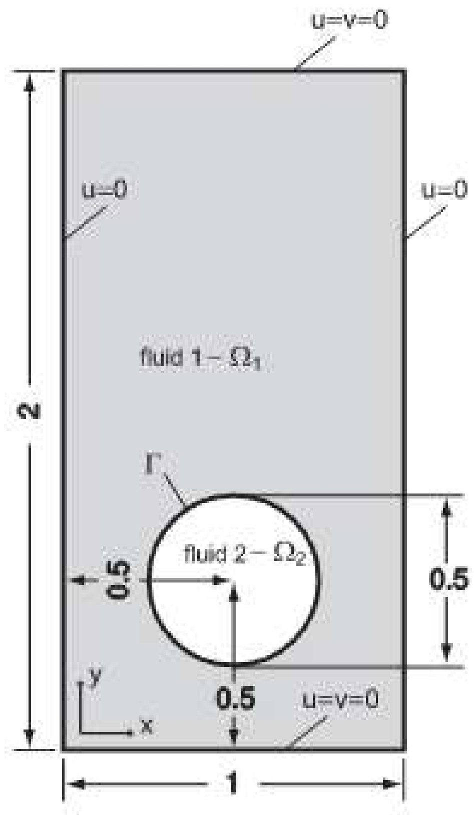


Figure 29: Configuration of the rising bubble case proposed by (Hysing, et al., 2009)

The initial configuration consists of a bubble with radius 0.25 with its centre located at [0.5, 0.5] in a 1 x 2 rectangular domain. The top and bottom boundaries are defined as no-slip while the vertical walls are slip boundaries. Cartesian meshes with different resolutions are used in the comparisons. Meshes are defined in terms of the number of cells along the upper or lower walls. In this investigation meshes of 40, 80, 160 and 320 cells will be used. In all cases the benchmark quantities are evaluated over 3s. However as has been mentioned previously the VOF method will produce a numerical smearing of the interface. Therefore, in all cases the initial bubble is simulated for 3s with gravity switched off. The result of these initial 3s runs are then used as the starting conditions for the comparison case, with gravity switched back on. This transition is shown in Figure 30. The properties of the fluids are show in Table 5 below. The subscripts denote the fluid, 1 being the ambient liquid and 2 being the gas bubble.

ρ_1 (kg/m^3)	ρ_2 (kg/m^3)	μ_1 (kg/m^3)	μ_2 (kg/m^3)	g (m/s^2)	σ (N/m)	Re (-)	Eo (-)
1000	100	10	1	0.98	24.5	35	10

Table 5: Parameters for bubble rise case

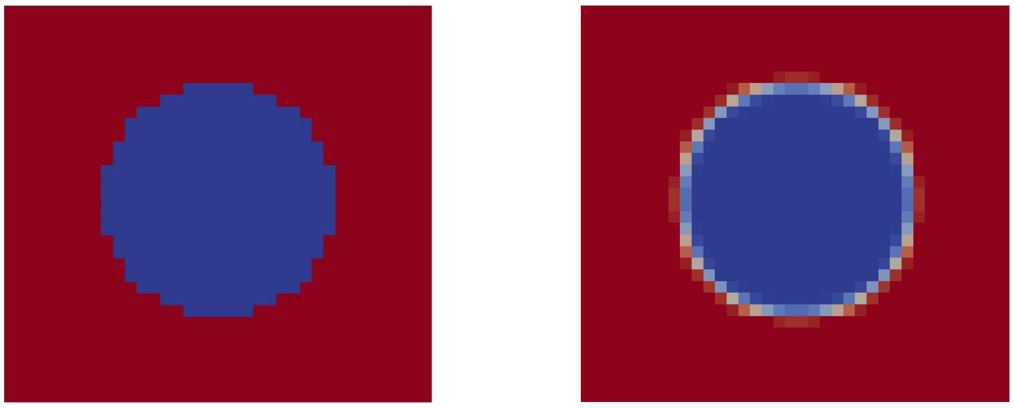


Figure 30: Interface smearing initialization on 40 cell mesh. (A) left, initial interface. (B) right, smeared interface. Red is liquid phase, Blue gas phase and transition is the interface

The intention of the investigation was to present a quantitative benchmark validation exercise for immiscible liquids undergoing topological changes. As there was not one accepted in contrast to other fields in CFD. To this end two exercises are presented. The first is as described above. The second, which is not considered here, has higher density and viscosity ratios as well as a lower surface tension coefficient. For each case several error metrics are used to establish the comparative performance of three codes, TP2D (Transport Phenomena in 2D), FreeLIFE (Free-Surface Library of Finite Element) and MooNMD (Mathematics and object-orientated Numerics in MagDeburg). In future comparisons in this study only the first two codes (TP2D and FreeLIFE) will be compared as they follow the Eulerian-Eulerian coordinate scheme, the same as interFoam, whereas MooNMD uses the Lagrangian-Eulerian scheme. Additionally, both codes, in contrast to interFoam, use the level set method for interface tracking. Finally, it should be noted that TP2D uses the same range of meshes as this study but FreeLIFE does not have a solution with 320 cells.

Initially the shape of the bubble at the final time step can be compared across the mesh densities with only results produced in this work using interFoam. For all cases the bubble outline identified by a volume fraction of γ=0.5.

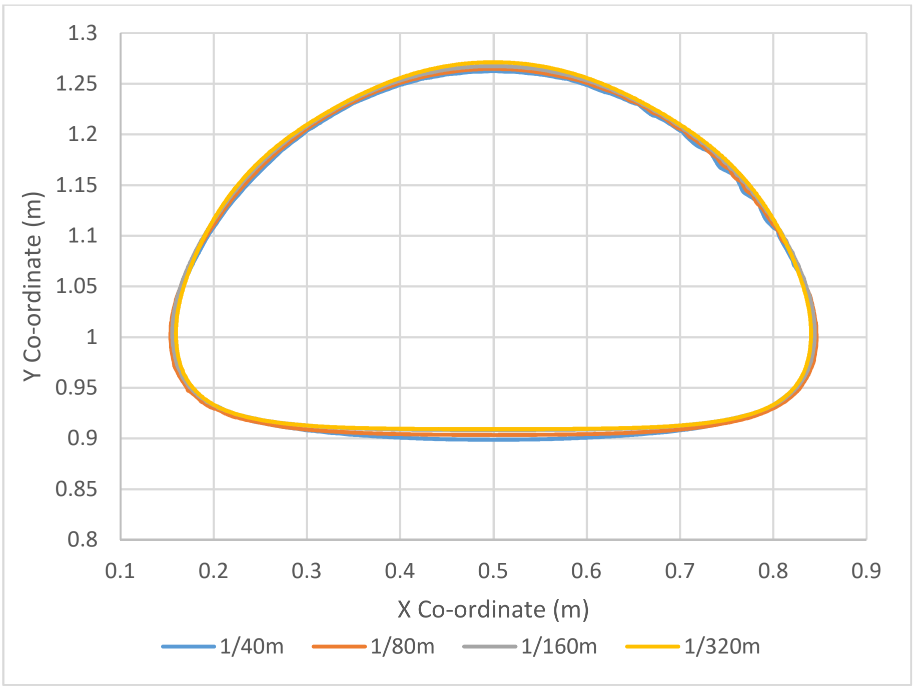


Figure 31: Bubble outline at t = 3s. Grids of 40 (Blue), 80 (Orange), 160 (Grey), 320 (Yellow)

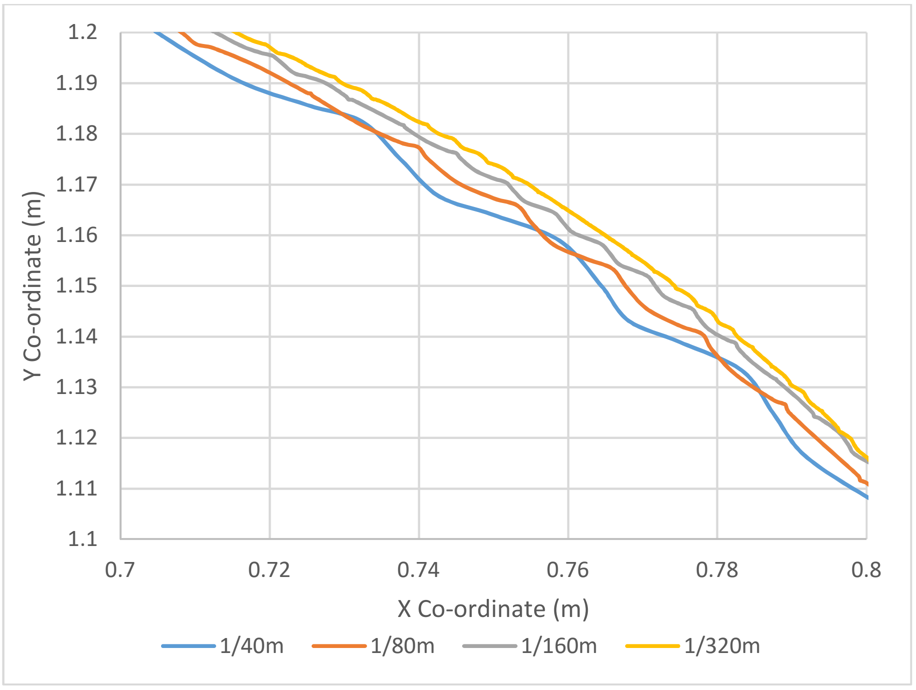


Figure 32: Close-up of bubble outline, Grids of 40 (Blue), 80 (Orange), 160 (Grey), 320 (Yellow)

From the view of the whole bubble at the final time step in Figure 31 it can be seen, apart from the coarsest mesh, the bubble outline shows little change across the meshes used. The close-up, Figure 32, shows this in more detail. Additionally, the close up shows the mesh convergence, indeed the difference between the two finest meshes is minimal.

The first comparison to published results that can be made is the degree of circularity. Here the perimeter of the bubble is compared to the perimeter of a perfect circle of an equivalent area. Therefore, a perfectly circular bubble will have a circularity of 1.

$$Degree\ of\ circularity = \frac{perimiter\ of\ area\ equivalent\ circle}{perimiter\ of\ bubble}$$

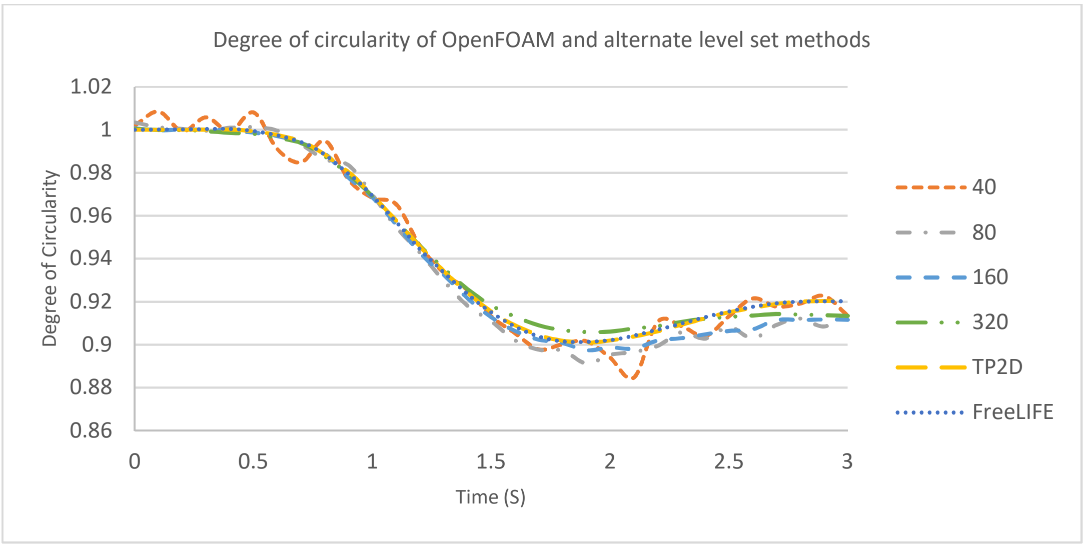


Figure 33: Comparison of circularity on various meshes in OpenFOAM and the level set methods used in (Hysing, et al., 2009)

In figure 33 it can be seen that the OpenFOAM results agree fairly well with those that are produced by the two level-set codes TP2D and FreeLIFE. Indeed from 0 to 2 seconds almost no difference can be seen on the higher cell density cases. The lowest cell density case however is poor. Next the bubble centre of mass is compared. This is defined by equation 21.

$$CoM = \frac{\int_{\Omega} \vec{x} dA}{\int_{\Omega} 1 dA} \tag{21}$$

In which $\vec{x}$ is the vector (x, y) position of the centre of a computational cell, Ω is the region occupied by the gas bubble and A is the bubble area.

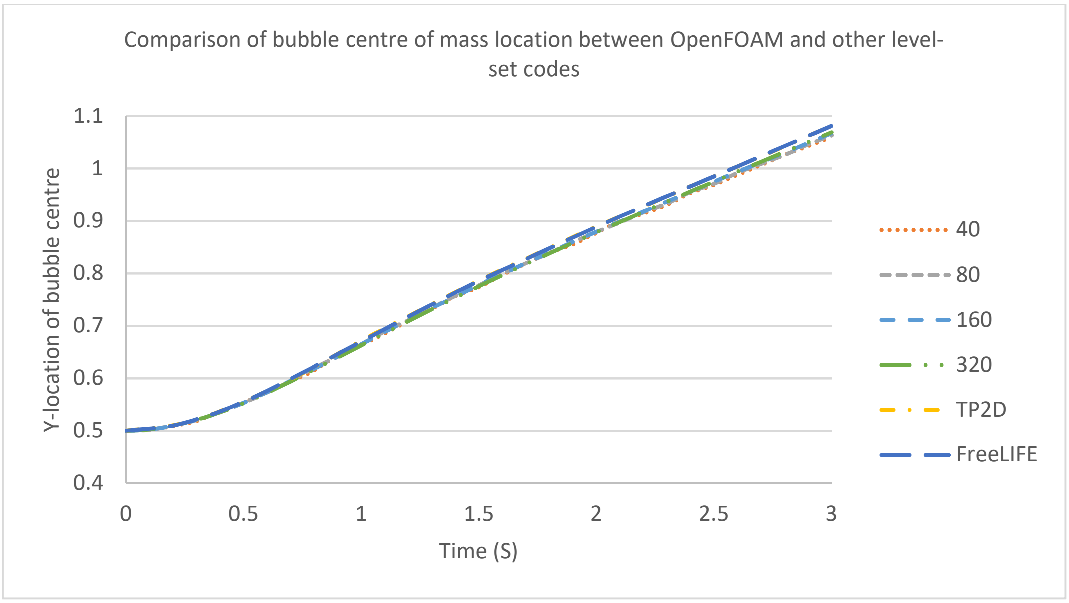


Figure 34: Comparison of bubble centre of mass

Finally, a comparison on the bubble rise velocity can be made. This is described by equation 22.

$$Rise\,Velocity = \frac{\int_{\Omega} \vec{U}\,dA}{\int_{\Omega} 1dA} \tag{22}$$

In which again Ω is the region of the gas bubble and $\vec{U}$ is the velocity vector.

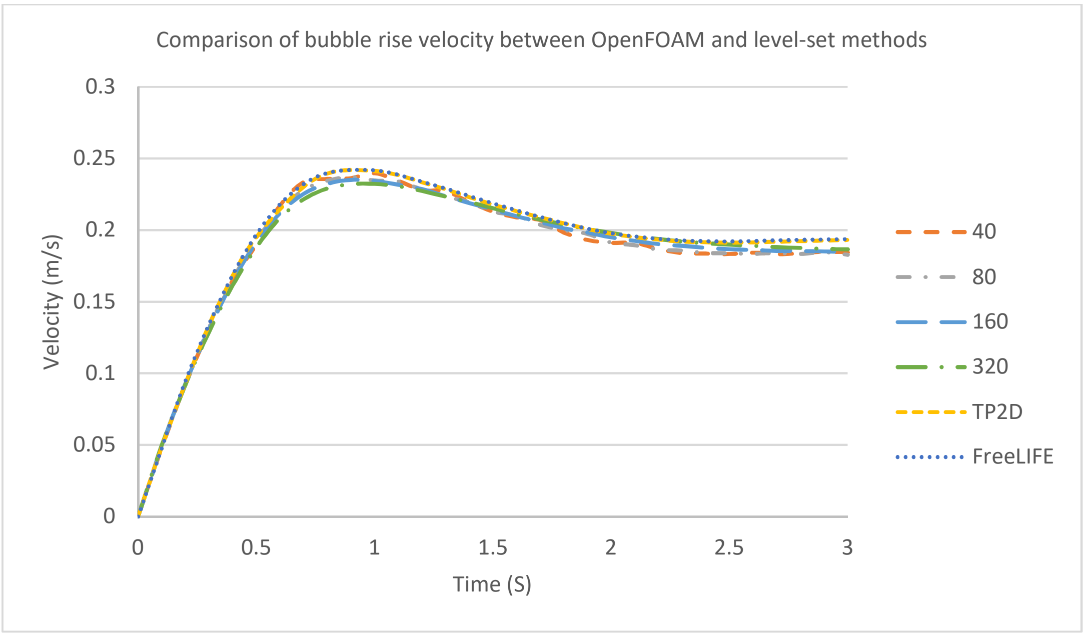


Figure 35: Comparison of bubble rise velocities

Again, the comparison with the two level-set codes presented in Hysing et al., (2009) is good although interFoam seems to show a slight under estimation of the bubble rise velocity which consequently gives an under estimation of the location of the bubble centre of mass. For completeness Hysing et al., (2009) provide CPU timings to give an idea of code efficiency. However, given the improvements in processor performance since this study, the efficiencies are of little value as a comparison with modern data. As an example, the most expensive cases took 35 hours and 30.25 hours for TP2D and FreeLIFE respectively. The most expensive case in this study took just 5.4 hours. While all were conducted on a single core of a multi core processor the processor age, and therefore its performance, varies significantly. In this regard some additional normalisation of processor performance would be needed to be added to compare code developments.

5.6 Performance Comparisons

The number of pressure corrector steps needed differs greatly from the (Issa, et al., 1986) suggested minimum of 2 steps required to reduce the discretization error if second order time stepping is used right up to the 15 suggested by Klostermann et al., (2013). Therefore, some further investigation is required. First a collection of runs were conducted on the 160 cell test case. In each case the bubble was initialised using 15 corrector steps. All cases were performed using a single core but with the Linux GUI switched off. The number of pressure correction steps was varied from 1 to 15, compute times are plotted below in Figure 36.

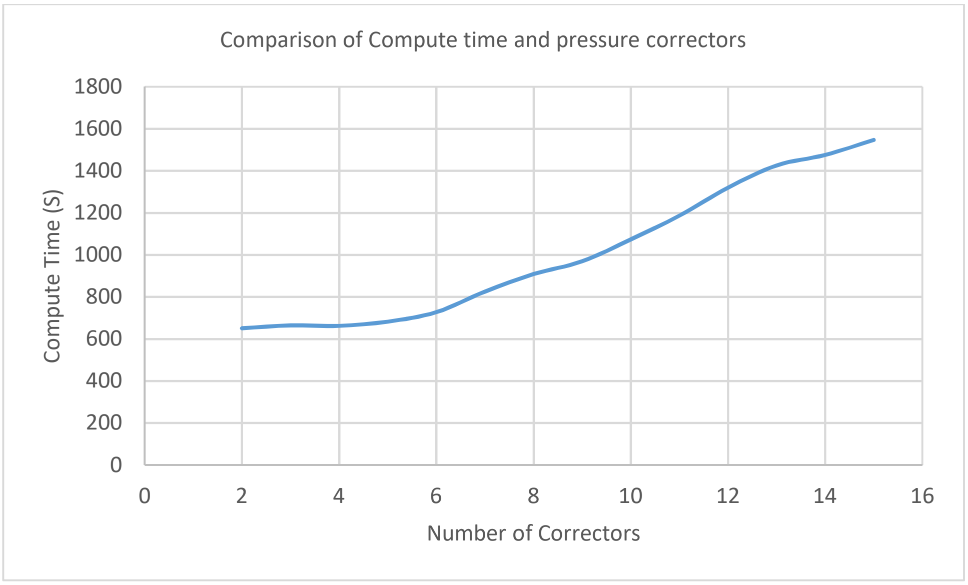


Figure 36: Graph comparing compute time with number of pressure correction steps

The first thing to note is that a single pressure corrector failed to produce a stable solution and so is not included in the results. The remaining results show that between two and four correctors give similar performance costs. After this however the cost rises sub linearly. With 15 correctors time is 2 1/3 times as great as 4 correctors. Results of the previous properties shown below were then compared in Figure 37 to Figure 39. In each comparison 2, 6, 10 and 15 pressure correctors are shown on a graph to improve clarity.

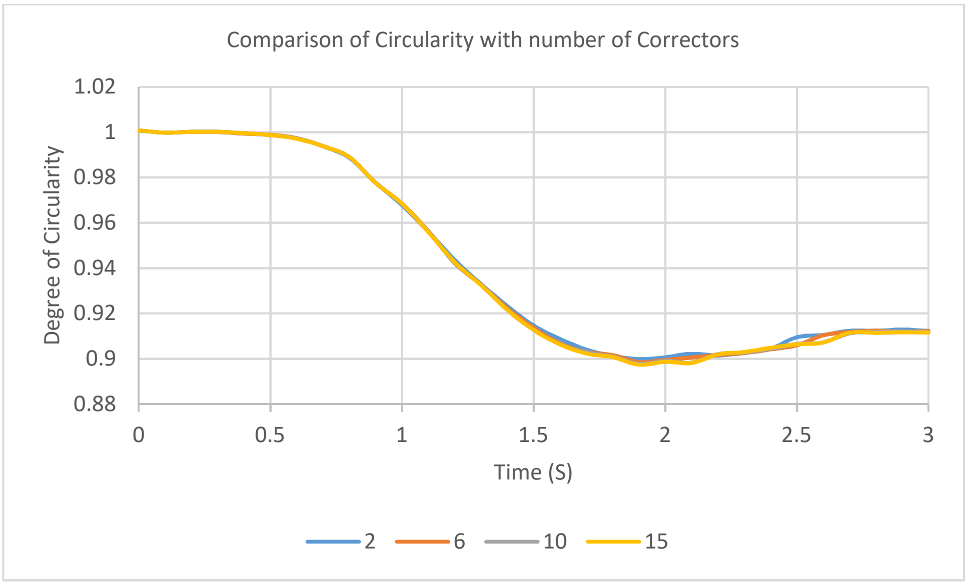


Figure 37: Circularity comparison with varying pressure correction steps

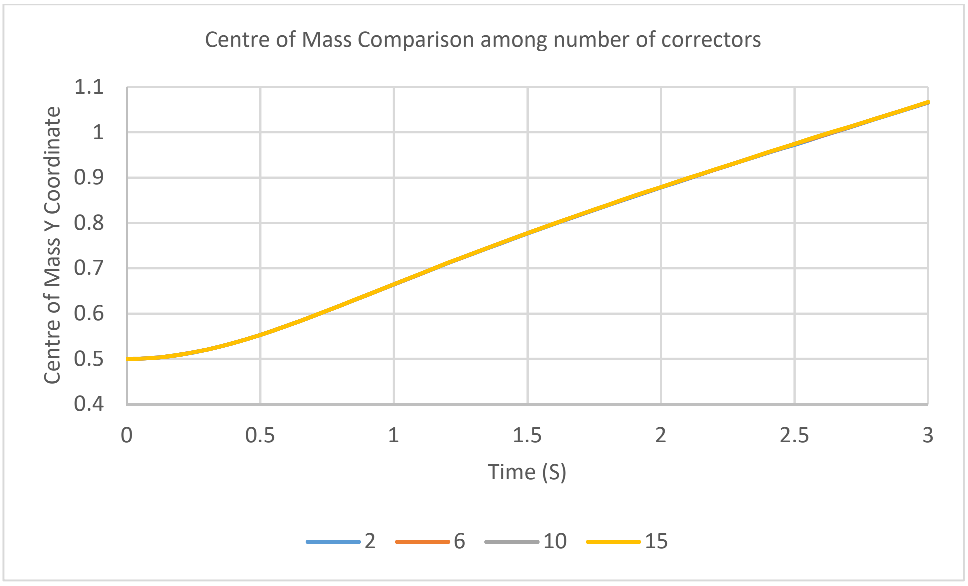


Figure 38: Centre of mass compared across varying pressure correction steps

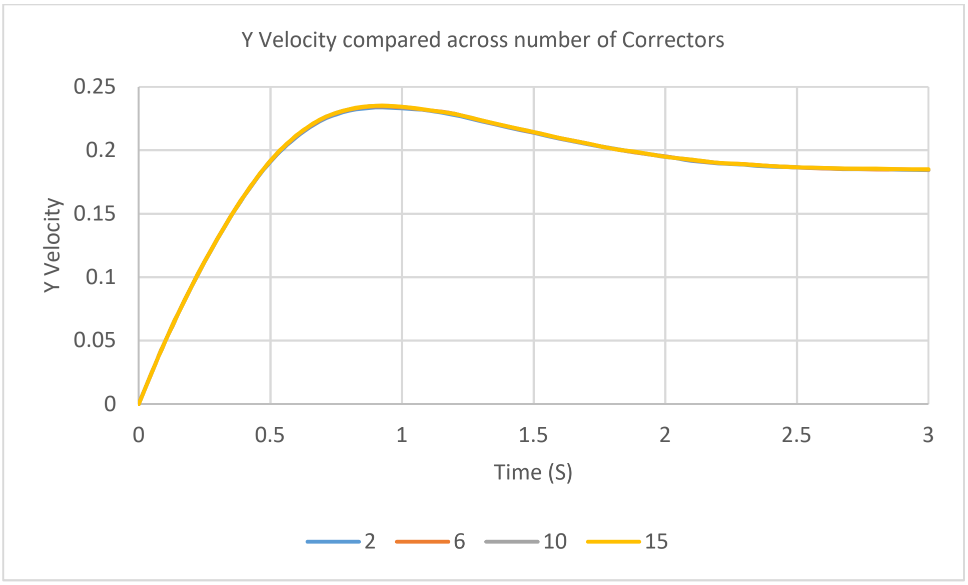


Figure 39: Comparison of Y velocity across number of pressure correction steps

These figures show that the quantitative results of this case do not very significantly with increasing number of pressure correctors. But what is the optimum efficient setup? If it is assumed that the largest number of correctors is the most accurate solution the error of each case can be evaluated in comparison to it.

$$100 - \frac{\|\sum_1^{NTS} R_1\| - \|\sum_1^{NTS} E_1\|}{\|\sum_1^{NTS} R_1\|} \times 100 \tag{23}$$

Here R is the actual values at each time step of the 15 pressure corrector steps case, E is the difference between the 15 pressure corrector steps case and the case in question and NTS is the number of time steps. Equation 21 effectively shows how different the result is compared to the optimum. Looking at results for Y-velocity, and 2 pressure correction steps. Across all time steps there is a loss in accuracy of 0.28% in the results but achieving that accuracy requires 138% more computing time.

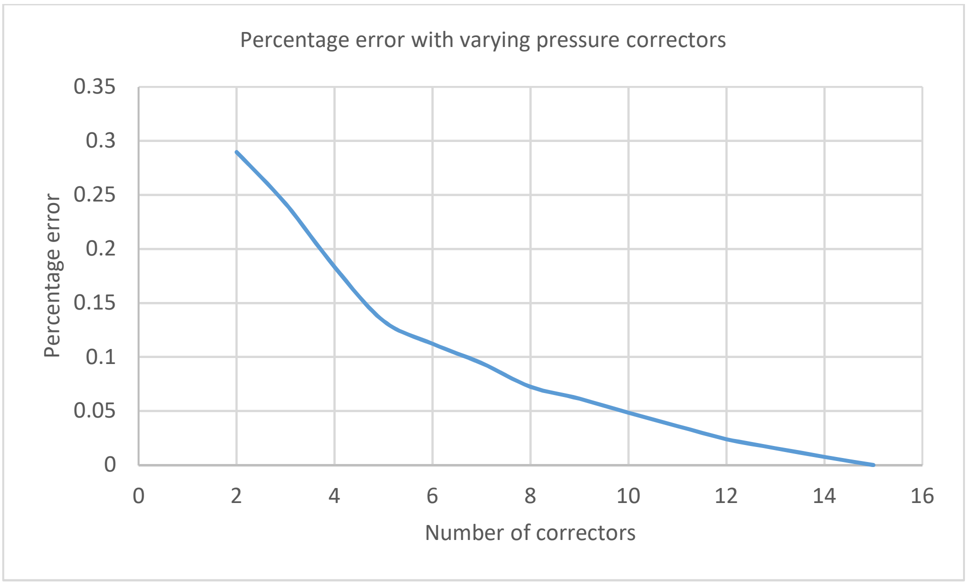


Figure 40: Percentage error across varying number of pressure correction steps

5.7 Parallel Scaling

Running OpenFOAM in parallel is common. The method used being domain decomposition. Optimizing this is very much dependent on both hardware available and the problem being solved. It is generally accepted that between 10,000 and 50,000 cells per core is most efficient. Less than 10,000 cells per core would result in high core to core communication while above 50,000 raw performance simply isn't good enough. In Keough (2014) indeed it is shown that the type of problem influences the number of cells needed to saturate a core.

To get a good idea of the systems parallel performance, again this case was used as a test. Each case was performed with the Linux GUI switched off using the 160-cell case. In addition to this hyper threading on the Intel CPU was switched off giving 16 logical cores on 16 real cores as opposed to 32 logical cores on 16 real. In all cases the scotch decomposition method was used to decompose the problem. Scotch is one of four options for decomposition, the others are simple, hierarchical and manual. Simple will just divide the domain into sections as defined by the user in terms of number in x y z directions. Hierarchical works in a similar fashion but instead equalises the number of cells per processor. This means it is very useful for domains that have meshes with refinement zones or if the number of cells in the three directions is not easily divisible by the number of processors available. Manual is where the user defines exactly which cells are allocated to which processor. The used scotch method aims to equalise the number of processor boundary cells between each processor, in theory this should equalise MPI communication. It requires no user input.

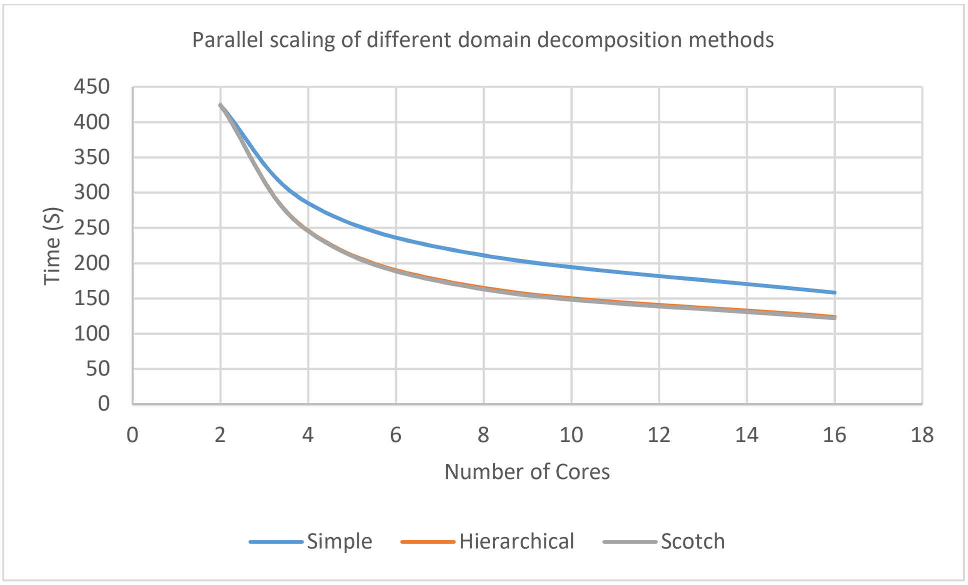


Figure 41: OpenFOAM parallel scaling with different domain decomposition methods on 160 cell case

Figure 41 shows a comparison between the different domain decomposition methods. It can be seen that at two processor cores there is no difference as would be expected because the methods can't make a difference. But after this clearly the simple method performs more poorly. However, there is very little difference between hierarchical and scotch so the advantage of scotch requiring no user input makes it a clear winner.

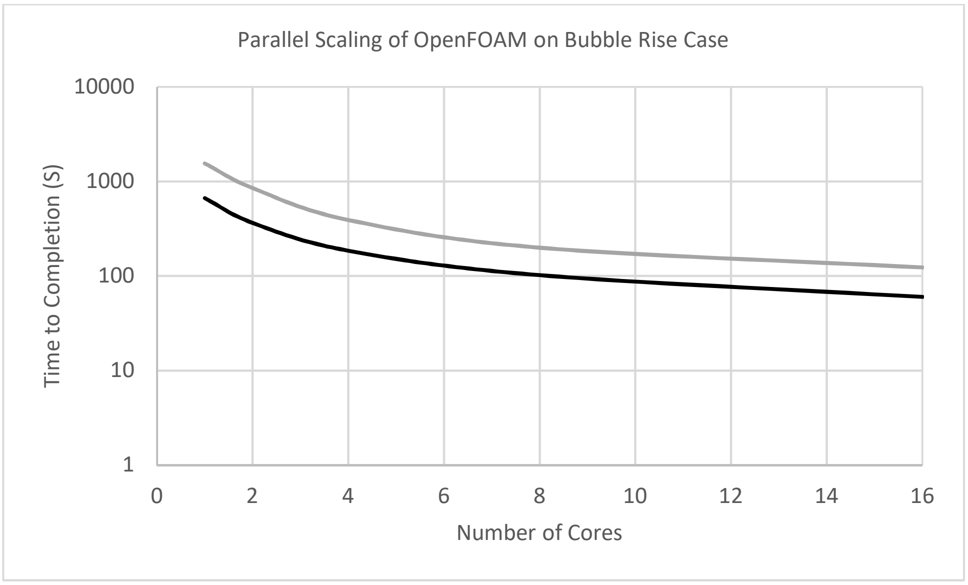


Figure 42: Parallel scaling of OpenFOAM

Strong scaling was compared with both 15 and 3 pressure corrector steps. In all cases as discussed previously, 3 pressure corrector steps was fastest. However, as the number of cores increases the gap reduces. In terms of parallel efficiency 15 correctors is always better, with the increased compute requirement being able to hide the intra core communication. With 15 correctors parallel efficiency is good up to 8 cores, staying above 90%, at 16 cores though it drops down to 78%. With 3 correctors efficiency is only above 90% with two cores. Dropping to 89% for 4 and 81% with 8. Finally, 16 cores have an efficiency of just 69%. This is explained by there only being 3200 cells per core so simply there is not enough computational requirement to hide the intra core communication. Additionally, the cost of decomposing and reconstructing the case (required cost to run in parallel) was found to be 60 seconds in every case no matter the number of cores or number of correctors. This cost is equal to the computational cost of 3 correctors running on 16 cores so is not insignificant in this instance.

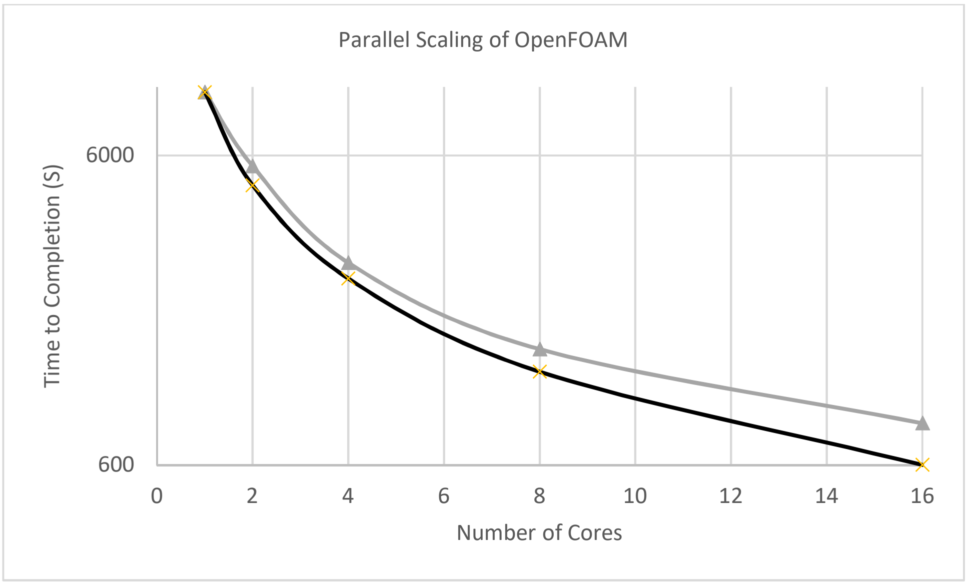


Figure 43: Parallel Scaling with 320 Cells, black is linear scaling, grey is scaling produced in this study

To further investigate the parallel performance of this case additional cases were used with higher cell counts. First, with the results shown in Figure 43 the 320-cell per unit case is shown. In black is linear scaling while in grey is scaling results obtained in this investigation. Even though the number of cells per core with 16 cores is now 12,800 the improvement in parallel efficiency is not great. Indeed, the rise is only about four percentage points from 69.3% to 73.4%.

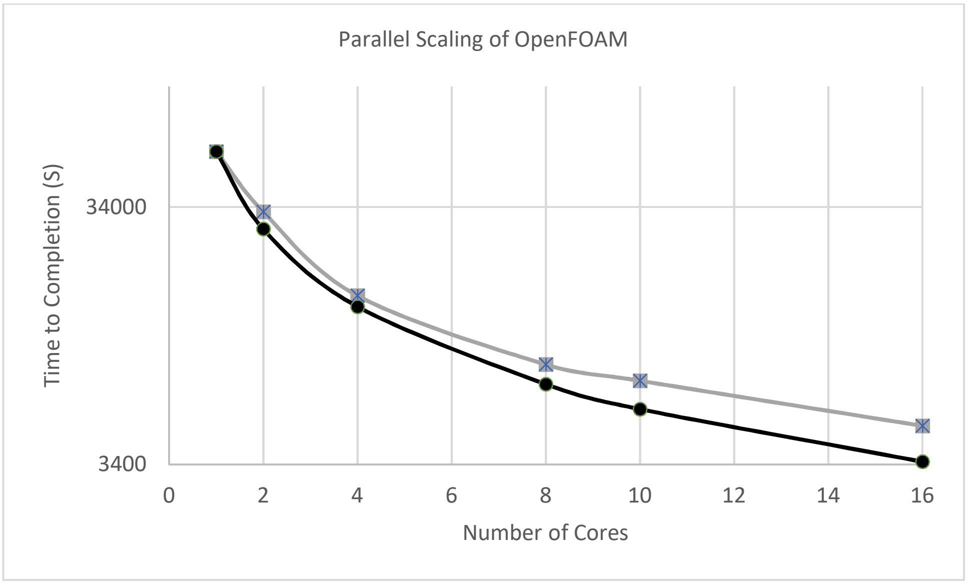


Figure 44: Parallel scaling of 480 cells per unit, black is linear scaling, grey is scaling produced in this study

Figure 44 has additional cells, now with 480 cells per unit length. This results in 28,800 cells per core. Again, black shows linear scaling while grey is scaling results in this study. Of note this time is that parallel efficiency now drops, albeit by less than one percent, from 73.4% to 72.6%. This would suggest that the optimum number of cells per core for this case lies between 13,000 and 28,000. A reasonable choice would be 20,000.

5.8 Chapter summary

This chapter has shown that OpenFOAM is a prime candidate for use in liquid atomization cases. While these cases are 2D it has demonstrated first that OpenFOAMs interface tracking method is accurate using a courant number of 0.25 it also can perform well when compared to interface tracking methods commonly found in liquid atomization studies. Setup parameters for future use have also been investigated to ensure future GPU comparisons are useful in real world applications. Finding that the three pressure correctors suggested by Deshpande (2012) are most efficient. In addition, the GAMG linear system solver is clearly the most efficient solver available. Finally, parallel scaling on the available computing platform has been assessed showing good strong scaling up to a point, with an estimation of 20,000 cells per core being most efficient.

Chapter 6 Surface Tension Dominated Test Case

6.1 Overview

Using the setup parameters found in the previous chapter interFoam will now be compared to an experimental liquid injection case, this will test its capabilities in three dimensions instead of just two. This will also give a good platform for development of an initial GPU acceleration which will be investigated in the latter portion of this chapter, this GPU acceleration will build upon the concepts discussed in chapter 4.

6.2 Experimental Description

To test the suitability of OpenFOAM in a primary spray atomization case as well as to provide a platform for GPU acceleration a relatively simple surface tension dominated case was selected. Longmire et al., (2001) presented an investigation into topological changes of liquid jet pinch off. Full two-dimensional velocity fields of a forced jet of glycerine and water solution pinching off in an ambient Dow Coming 200 series silicone fluid were measured using particle image velocimetry (PIV).

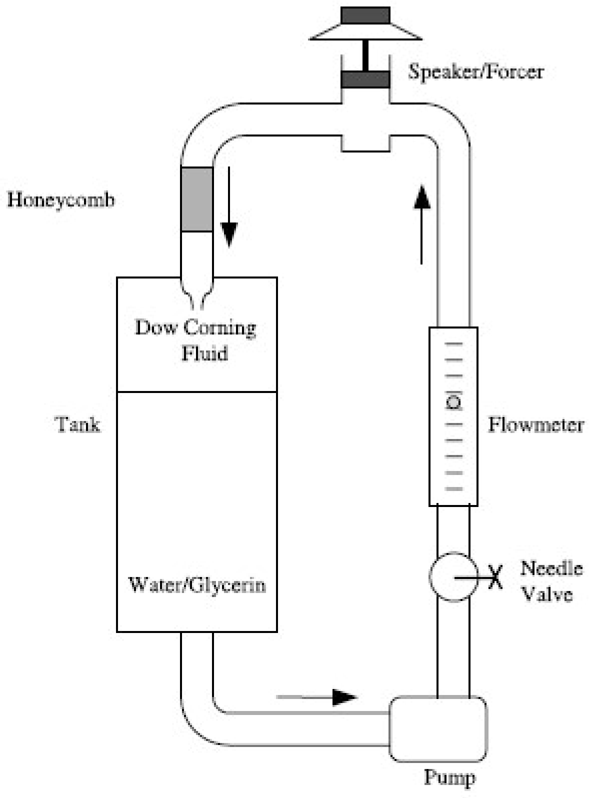


Figure 45: Diagram of experimental setup from (Longmire, et al., 2001)

Two series of experiments were performed, in the first series the Dow Coming ambient liquid viscosity is approximately 10 times that of the second series. The resulting pinch off dynamics will therefore differ. Additional parameters of the experiment are described in Table 6. These non-dimensional numbers use the inlet diameter, D, and velocity. Only the first experiment series is considered here, where the viscosity ratio is 0.17.

Reynolds number	Froude Number	Eötvös number	Density Ratio	Viscosity ratio	Strouhal number
34	0.2	6.1	1.19	0.17	4

Table 6: Dimensionless parameters used by (Longmire, et al., 2001)

Without forcing, on exit of the 10mm nozzle the liquid would contract and continue as a steady stream until it leaves the ambient Dow Coming fluid. However, if forcing is introduced, in this case at 10Hz, a sinusoidal velocity at the nozzle exit is created. This sinusoidal velocity will generate repeatable pinch-off conditions.

6.3 Computation Setup

In order to simulate the conditions described above a small section of the tank in Figure 45 just below the nozzle is used as the computational domain. Using the meshing program blockMesh that is included in OpenFOAM a regular orthogonal mesh is generated.

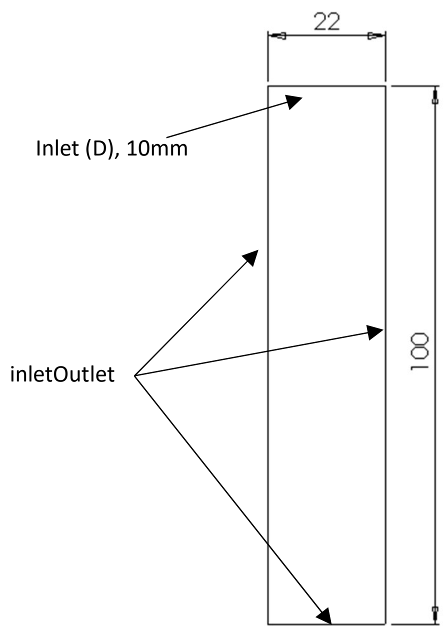


Figure 46: Illustration of computational setup

The domain is 22mm square in the jet cross section and 100mm long. Grid sizing is Δx/D = 0.05 where D is the jet diameter. Because of the Cartesian grid, the round nozzle is approximated by square cells. To produce the sinusoidal velocity profile a time dependent velocity condition was imposed at the inlet using the OpenFOAM user contributed library groovyBC. Apart from the top boundary described as the inlet the remaining boundaries are set using the OpenFOAM inletOutlet condition. In this boundary condition the condition will apply zero-normal gradient when the fluid is flowing out (flux is positive) while if there is backflow (flux is negative) a fixed user defined value will apply, in this case inflow of the ambient Dow Corning fluid. For pressure however the inletOutlet condition is replaced with a fixed value.

Quantitative comparisons come from contrasting the liquid jet shape between computation and experimental at instantaneous times. To quantify the time at which the image is taken the phase of the sinusoidal wave is used.

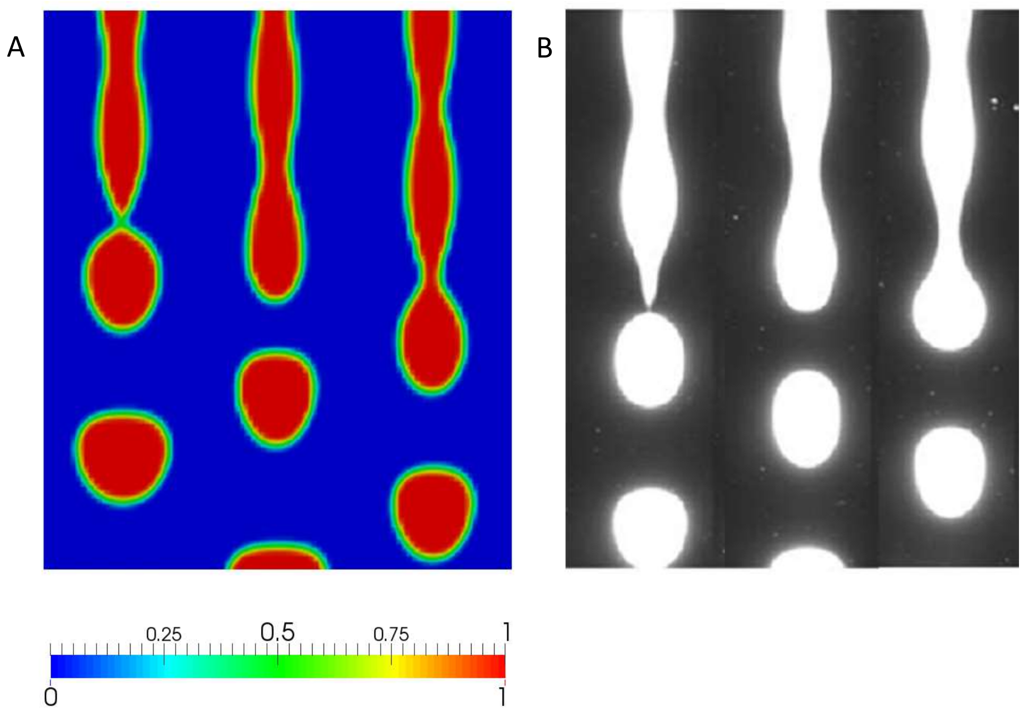


Figure 47: Comparison of computational results (A) to left, experimental results from (Longmire, Norman, & Gefroh, 2001) (B) to the right, showing the jet outline and droplet pinch off. In each series from L to R 0, 120 and 240 degrees of the sinusoidal profile.

Figure 47A, is coloured by the volume fraction of the injected water/glycerine mixture, being one (red), while the ambient Dow Coming is zero (blue). As is common when using the volume of fluid method there is an interface of partially filled cells which extends over three to four cells. The match between the liquid structures in both the computational and experimental results is acceptable. The computational results indeed show a good representation of the perturbations found in the liquid jet. A good example of well captured features is in the initial image at zero degrees. Here the experimental results show a very thin neck between the jet and what is about to become a droplet. In the computational results this is represented by a small area where the volume fraction doesn’t reach one. Additionally, the characteristic flattening of the droplets after they have pinched off the jet is well captured. However, it is noted that the numerical results show a slight under estimaton of the velocity profile.

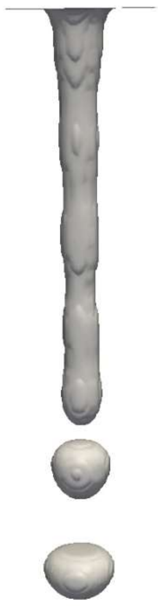


Figure 48: 3D representation of the computational results in this study using iso-surface of $\gamma = 0.5$

This experiment has previously been used by Pan & Suga (2003) to validate their investigation into the use of the level set method for multiphase problems involving interface breakup. For the quantitative results that follow, the results presented in Pan & Suga (2003) are included as a comparison. First the instantaneous axial velocity along the jet centreline is measured. Again, time description is in terms of degrees of the sinusoidal velocity inlet.

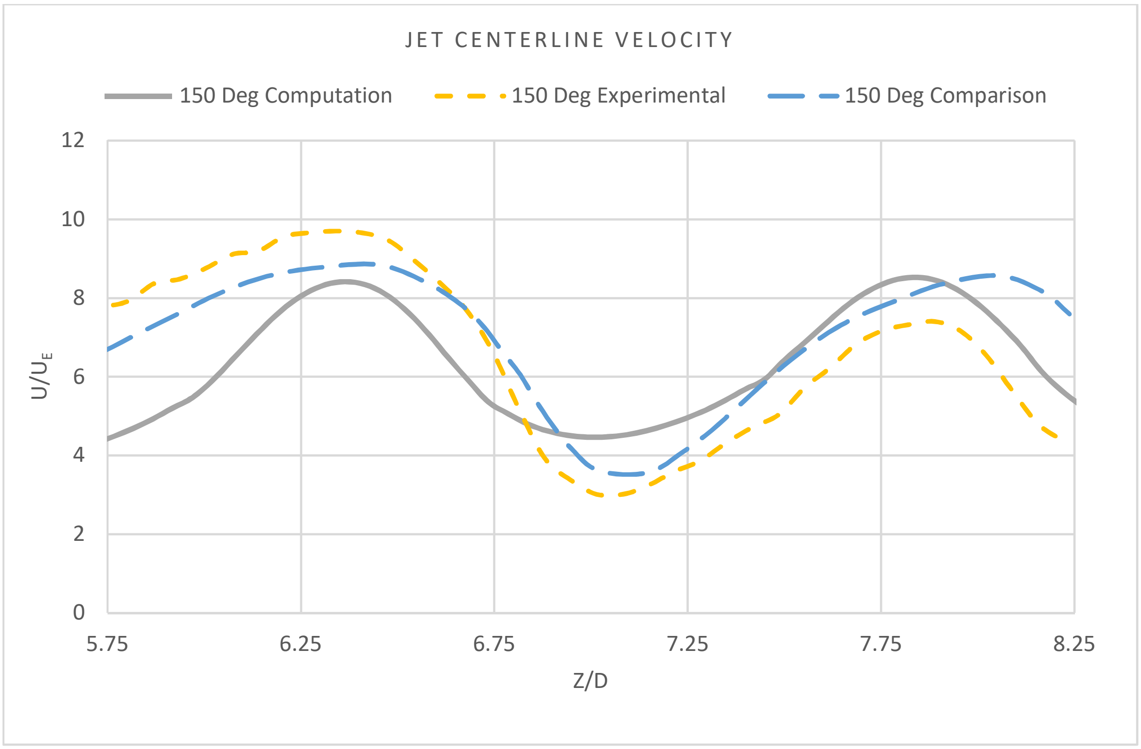


Figure 49: Jet centreline velocity at 150 degrees, including computational results from this study, experimental results from (Longmire, et al., 2001) and a comparison to (Pan & Suga, 2003)

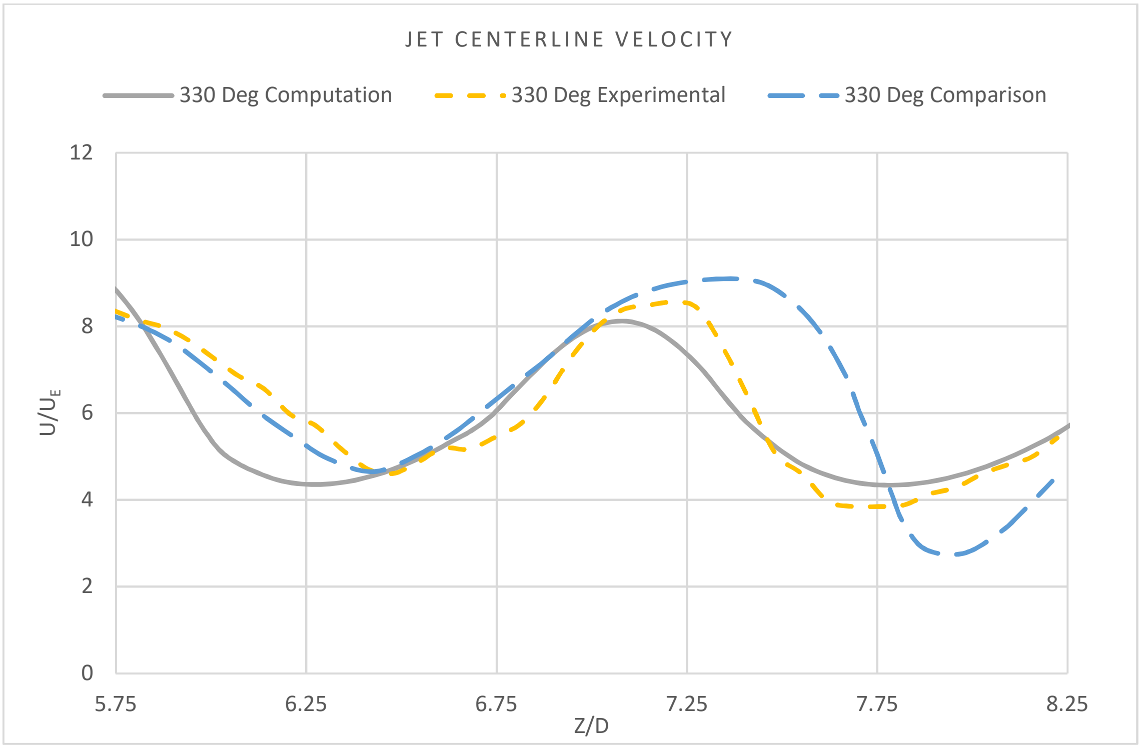


Figure 50: Jet centreline velocity at 330 degrees, including computational results from this study, experimental results from (Longmire, et al., 2001) and a comparison to (Pan & Suga, 2003)

Again, the agreement between the computational results in this study and the experimental results in Figure 49 & 50 is reasonable. It can be noted that the agreement is better downstream than upstream however. The magnitude of velocity is well captured in all cases. The experimental results and the present study captures this better than was managed by Pan & Suga (2003), specifically at the downstream minimum and maximum.

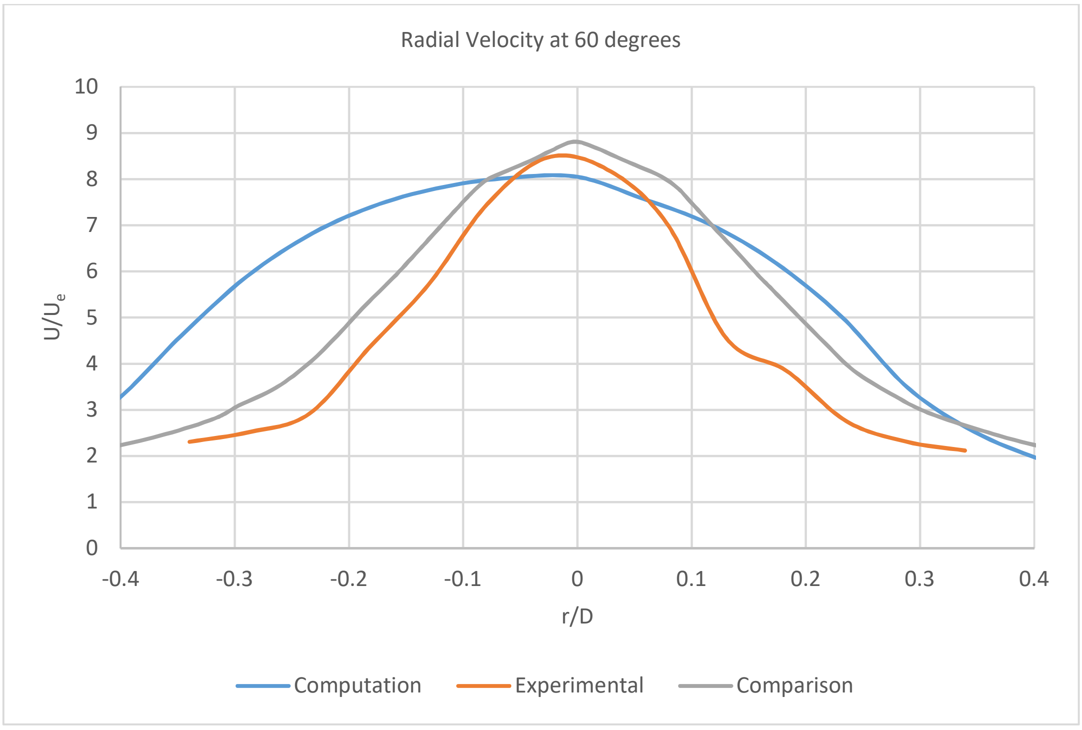


Figure 51: Radial velocity profile at z/D = 6.15 and 60 degrees, including computational results from this study, experimental results from (Longmire, et al., 2001) and a comparison to (Pan & Suga, 2003)

Further comparison can be made on the radial velocity profile. In Figure 51 the radial velocity profile at a distance of 6.15 diameters is shown, again experimental results and the computational comparison are shown. The results produced by Pan & Suga (2003) show a slight over prediction of the velocity found experimentally. Whereas in the present study, as was seen in the centreline velocity figures, a small under prediction of the axial velocity is made. In addition, the velocity gradient moving away from the jet centreline is lower. This is likely due to using an inlet condition that has a uniform velocity across the nozzle diameter.

6.4 Mesh independence

To assess the mesh independence of the solution 3 meshes were constructed. The time averaged flow rate of the exit of the domain was calculated for each mesh.

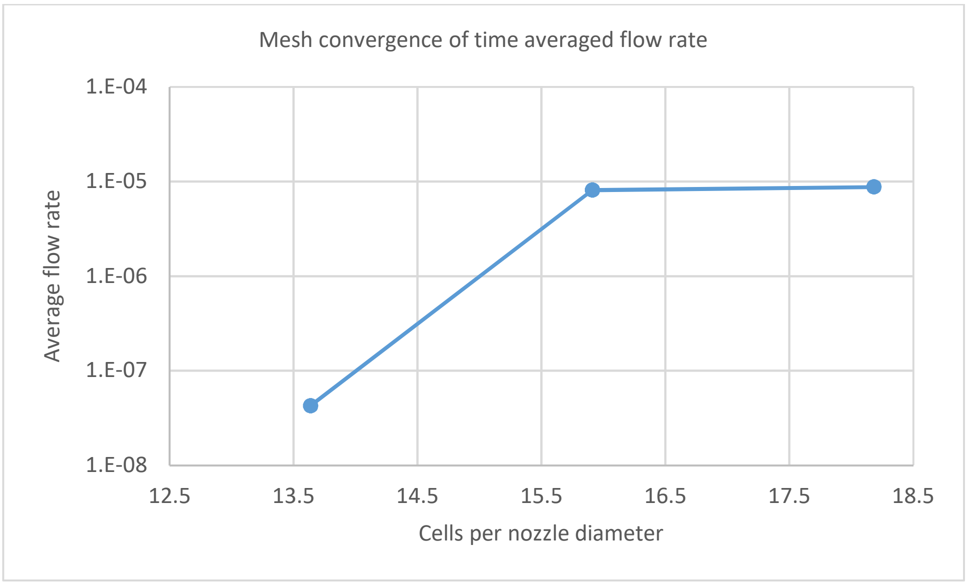


Figure 52: Mesh convergence of solution

The flow rate at the exit of the domain is used as the convergence metric as the axial velocity, which is the primary driver of flow rate, is also the key factor in the statistics presented for comparison. Note that the flow rate in this instance is small as the average injection velocity is 2.48×10^{-2}m/s.

6.5 Full port or partial port?

As discussed previously OpenFOAM is not a monolithic code and consists of many independent parts. Therefore, a full port would represent a significant challenge even for a large number of researchers with considerable time. In addition, a full port is only worthwhile if there are several sections of code making significant contributions to the overall compute time.

As seen in the literature commonly the most computationally expensive sections of CFD code are the linear system solvers. These are therefore the prime candidates for GPU acceleration. As shown previously, OpenFOAM has numerous linear system solvers available for a variety of applications. Tomczak (2013) showed that with their lid driven cavity case 87% of the compute time is taken up with solving the linear system produced by the pressure correction equation.

If it is assumed that memory bandwidth is the limiting factor in solving the system of linear equations (Bell & Garland, 2009) the GPU has a claimed advantage of 6 times over the CPU in this respect. In an idealised example taking from the above, if the linear system accounts for 87% of a 100 second compute time then a 6 times speed improvement would result in a compute time of 27.5s and a 3.6 times speed up.

6.6 Integration between OpenFOAM and GPU solver

As OpenFOAM is not a monolithic code but collection of libraries that have different functions, this can make integrating special functions easier. In the case of OpenFOAM linear system solvers these are also contained in libraries. Each linear system solver has its own library. Contained in the source code of this library are the methods used to solve the system as well as access to the controls the user inputs in dictionary files. Finally, in cases such as the preconditioned conjugate gradient method where there are sections of code used by multiple solvers, i.e. the preconditioners, interfaces to these libraries are included.

As mentioned previously the transport equations are discretised into a coefficient matrix. OpenFOAM uses a type of COO matrix format, LDU, to store the coefficients of this matrix. The LDU matrix stores the upper triangle, lower triangle and diagonals of the matrix in dense vectors. These dense vectors are addressed using the COO format. While the LDU format is useful for algorithms where operations on one specific part of the matrix are required, for example Gauss-Seidel. However, when using GPUs there are some elements of this format that are non-desirable. The first is in terms of data transfer, while the difference may seem small when large matrices are involved the difference is not insignificant. Secondly the algorithms that can make use of the LDU format are often not suited to the massively parallel nature of GPUs. As generally there are large elements of sequential code.

In the first instance of data transfer the COO format requires 1 floating point value and 2 integer values per non-zero matrix value. The CSR format however only requires 1 floating point and 1 integer per non-zero matrix value with an additional integer per matrix row. This will reduce the number of integer values by about 40% or 10 bytes per row. Additionally, as investigated by Bell & Garland (2009) the COO format often suffers from poor memory coalescence.

The way that a linear system solver is integrated is illustrated in Figure 53: a linear system solver is created that is loaded at run time. This solver is written in OpenFOAM 'format' it connects to the functions that OpenFOAM requires, such as final residual etc. Included in this is the conversion of the coefficient matrix from OpenFOAM LDU format to CSR format. A solve function in a separate library is called by this solver that solves the matrix on the GPU. This library is compiled using included OpenFOAM compile structures and the open source GCC compiler.

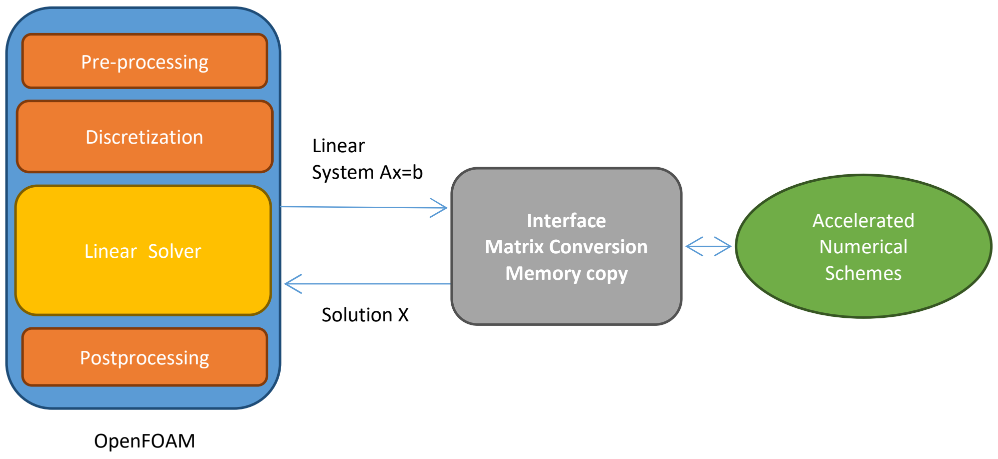


Figure 53: Outline of accelerated solver interface

The called solve function is written separately in CUDA and is compiled using the NVidia CUDA Compiler (NVCC). This additional library is passed pointers to CPU memory that contains the coefficient matrix and result vector. The shared library written in CUDA then performs all required functions of the linear system solver, including the algorithm and memory transfer to the GPU. The function that is called by the CPU solver then returns with a pointer to the result vector in CPU memory and information about solver performance; such as number of iterations and final residual.

6.7 Jacobi Solver for GPU

For solving the momentum equation OpenFOAM generally is configured to use a Gauss-Seidel smoother, one of a collection of smoothers that can be used as a solver or as a smoother for the GAMG solver.

The Gauss-Seidel method is in some ways undesirable as an algorithm for GPU computing as it includes some operations that are inherently sequential and not suited to massively parallel computing. An alternative and related method is the Jacobi method. The Jacobi method consists of the iteration of equation 24, which is written in matrix-vector format:

$$X^{(k+1)} = D^{-1}(B - (L + U)X^k) \tag{24}$$

Where D is the diagonal, L is the lower and U is the upper sections of a matrix A. while the superscripts indicate the iteration number. Each operation is then parallelised to run on the GPU. The operations used to iteratively solve the above equation are outlined below:

$R \times X = RX$	Multiply 1
$B - RX = BRX$	Subtract 1
$D \times BRX = X1$	Multiply 2
$X1 - X = X2$	Subtract 2

The general rule in parallelising these operations is that each matrix/vector row will be computed by one GPU thread.

6.8 GPU Acceleration

In order to test the capability of the GPU accelerations a number of cell sizes were selected. The comparison was made between a set-up run purely on CPUs using standard OpenFOAM solvers and an identical set-up using the GPU accelerated solvers. The results can be found in Table 7. Additionally, the speed up of the GPU momentum solver over its Gauss-Seidel and Jacobi counterparts running on CPU is shown. The speed up is calculated from the total measured wall clock times of each run.

Cell Size	Overall speed up	Momentum Speed up	Speed up over Jacobi on CPU
315K	1.75	1.09	2.59
1M	1.94	1.34	3.33
2.5M	1.53	1.47	3.59
5M	1.35	1.68	4.17

Table 7: Speed up of dripping case using GPU accelerated solvers over CPU solvers

Table 8 shows the percentage of time taken in each time step for the calculation and the various memory transfer operations. This explains the improved performance of the Jacobi acceleration (JSAccel), as the cell count of the problem increases. The gains in speed from using GPUs are found in calculations where the reduction in compute time, conducting the computation on a GPU, outweighs the disadvantage of a very slow transfer of data to the GPU. Therefore, as the amount of computations increase compared to data transfers, the GPUs advantage increases.

Operation	315K Cells	1M Cells	2.5M Cells	5M Cells
Calculation (%)	55.695	62.199	66.994	70.651
Host to Device (%)	39.002	32.350	27.430	23.672
Device to Host (%)	2.7914	2.3233	1.9352	1.5473
Device to Device (%)	2.5119	3.1277	3.6400	4.1296

Table 8: Memory transfer proportions in one time step of JSAccel

To ensure the GPU is being used efficiently the usage of its main performance metrics can be calculated. First a calculation of the computational throughput was made; it was found that raw compute power usage was orders of magnitude below maximum. Second the memory bandwidth of individual code sections was calculated to assess memory bandwidth usage. The results of the initial study can be found in Figure 54.

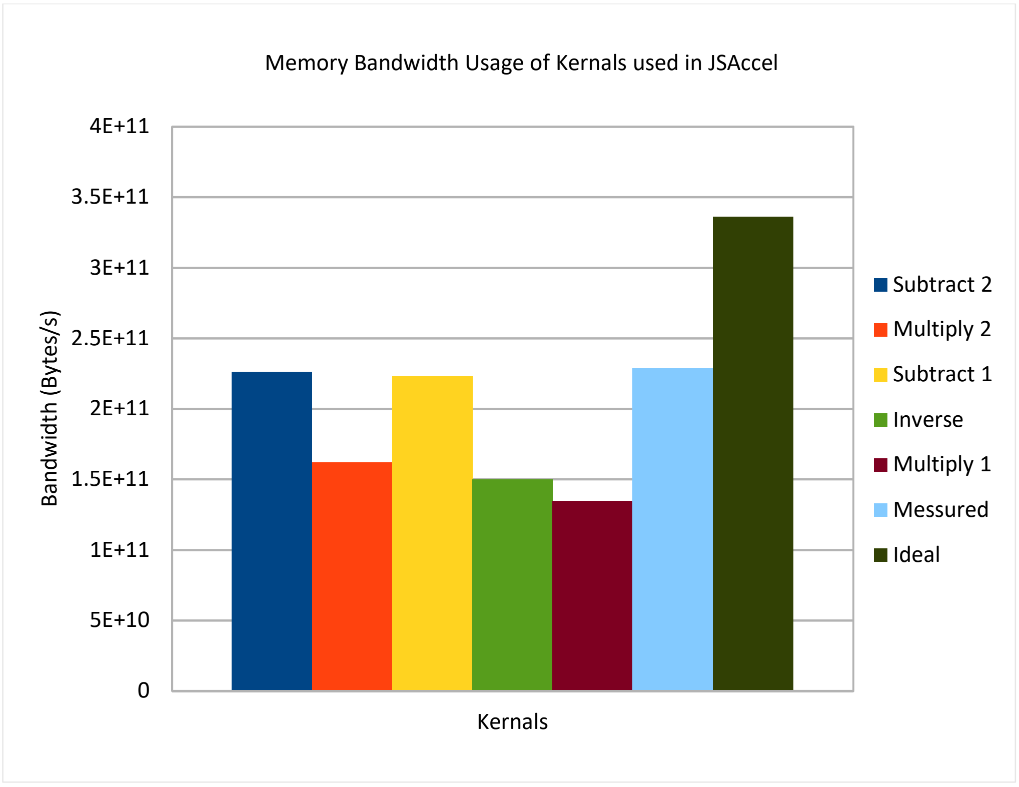


Figure 54: Initial memory bandwidth usage

It can be seen that some of the kernels used perform much more poorly than others, and of particular note is multiply1. This kernel accounts for approximately

50% of the computational cost of each iteration of the JSAccel solver, because this is where the matrix A is multiplied by the vector X. Therefore, savings in this area is highly desirable. The parallelisation method for this section of JSAccel is broadly similar to what would be used in a CPU code with one thread operating on each row of the matrix A. As mentioned previously GPUs access memory in chunks of 128 bytes. In this instance groups of threads are accessing memory that has gaps equal to the number of non-zero values per row. To remedy this and allow threads to access memory that is better coalesced, multiple threads per row can be used.

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;

if (idx >= num_rows)
   return;

double dot = 0;

int row_start = Ap[idx];
int row_end = Ap[idx+1];

for (int k = row_start; k<row_end; k++)
   dot += Ax[k] * x[Aj[k]];

y[idx] = dot;
```

Code 3: Original matrix vector multiplication

First in Code 3 an index for each thread is defined. Next an if statement is used as an escape for non-required threads. As threads are launched in groups of 256 it is likely there will be a small number that are not required, this statement means they do nothing and won't produce an error. Then the sum for each row is allocated. Following this the start and end of each row are accessed from global memory. A for loop is then used to iteratively sum the product of each matrix value and vector value, this section runs serially on each thread. Finally, the sum is written to global memory.

```
const int THREADS_PER_VECTOR = 4;
const int VECTORS_PER_BLOCK = BLOCK_SIZE/THREADS_PER_VECTOR;
const int THREADS_PER_BLOCK = VECTORS_PER_BLOCK * THREADS_PER_VECTOR;
const int thread_id   = THREADS_PER_BLOCK * blockIdx.x + threadIdx.x;    // global thread index

 if (thread_id >= (num_rows*THREADS_PER_VECTOR))  // Error
   return;

   __shared__ volatile double sdata[VECTORS_PER_BLOCK * THREADS_PER_VECTOR + THREADS_PER_VECTOR / 2];
   __shared__ volatile int ptrs[VECTORS_PER_BLOCK][2];

   const int thread_lane = threadIdx.x & (THREADS_PER_VECTOR - 1);         // thread index within the vector
   const int vector_id   = thread_id   /  THREADS_PER_VECTOR;              // global vector index
   const int vector_lane = threadIdx.x /  THREADS_PER_VECTOR;              // vector index within the block
   const int num_vectors = VECTORS_PER_BLOCK * gridDim.x;                  // total number of active vectors

   for(int row = vector_id; row < num_rows; row += num_vectors)
   {

      if(thread_lane < 2)
         ptrs[vector_lane][thread_lane] = Ap[row + thread_lane];
      const int row_start = ptrs[vector_lane][0];                 //same as: row_start = Ap[row];
      const int row_end   = ptrs[vector_lane][1];                 //same as: row_end   = Ap[row+1];

      double sum = 0;

      for(int jj = row_start + thread_lane; jj < row_end; jj += THREADS_PER_VECTOR)
        sum += Ax[jj] * x[Aj[jj]];

      sdata[threadIdx.x] = sum;

      if (THREADS_PER_VECTOR > 16) sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x + 16];
      if (THREADS_PER_VECTOR >  8) sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x +  8];
      if (THREADS_PER_VECTOR >  4) sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x +  4];
      if (THREADS_PER_VECTOR >  2) sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x +  2];
      if (THREADS_PER_VECTOR >  1) sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x +  1];

      if (thread_lane == 0){
         y[row] = sdata[threadIdx.x];
      }
   }
};
```

Code 4: CSR vector multiplication as proposed by (Bell & Garland, 2009)

In detail Code 4 can be explained as follows. The first 3 lines create variables that have information about vector sizes, the fourth has the thread index. Next an if statement is used as an escape for non-required threads as previously noted. The next two lines create on chip variables that can be accessed by all threads, these are used for the sum totals and row start and end values. The following two lines then give indexes for each thread in relation to their location in a group of threads operating on one row, which group it’s located in. The next two lines give indexes for the thread groups. The for loop is then used to compute the result for each matrix row. The next lines then access the start and end positions for the row being operated on, it uses two threads, one to access each value. Next a variable is allocated for the result of a thread. The final for statement then calculates the multiplication of the matrix values by the corresponding vector value this is added to the sum by each thread. This is done with the number of threads per row and if there are matrix values remaining the group operates again until all are computed. This gives each thread a part of the result which is transferred to shared memory. The transfer enables the next if statements to perform a reduction, each time using half as many threads as there are values to be summed. Finally, the first thread writes the final result to global memory.

Finding the number of threads to use per row can be achieved in the following way. The average number of non-zeros per row can be computed, in this case approximately 6.3. This number of threads would be ideal from the view point of memory coalescing, however due to the practicalities of GPU computing this is not the case. The reality of GPU computing is that after the multiplication has taken place, the values for each must be summed together, which is best performed with one addition per thread. Otherwise the idling threads will harm performance. With this considered, the only options are to use either 4 or 8 threads per row. Testing showed that the number of threads that are idling when using 8 threads per row harm performance more than does the worse memory coalescing using 4 threads per row. The results of this change and some other minor changes are shown in Figure 55. In the figure, the inverse kernel is used to calculate the inverse of the diagonal matrix. The memory bandwidth usage of each kernel is compared with each other as well as the maximum measured by an ideal operation (Measured). This measured value was found by the method detailed in Chapter 4. Finally, the maximum bandwidth claimed by the manufacturer (Ideal) is shown.

The results of this improvement in memory coalescing were soon evident with the time required to compute multiply2 being reduced by a third. Coupled with the other small improvements, an overall improvement of approximately 20% was achieved.

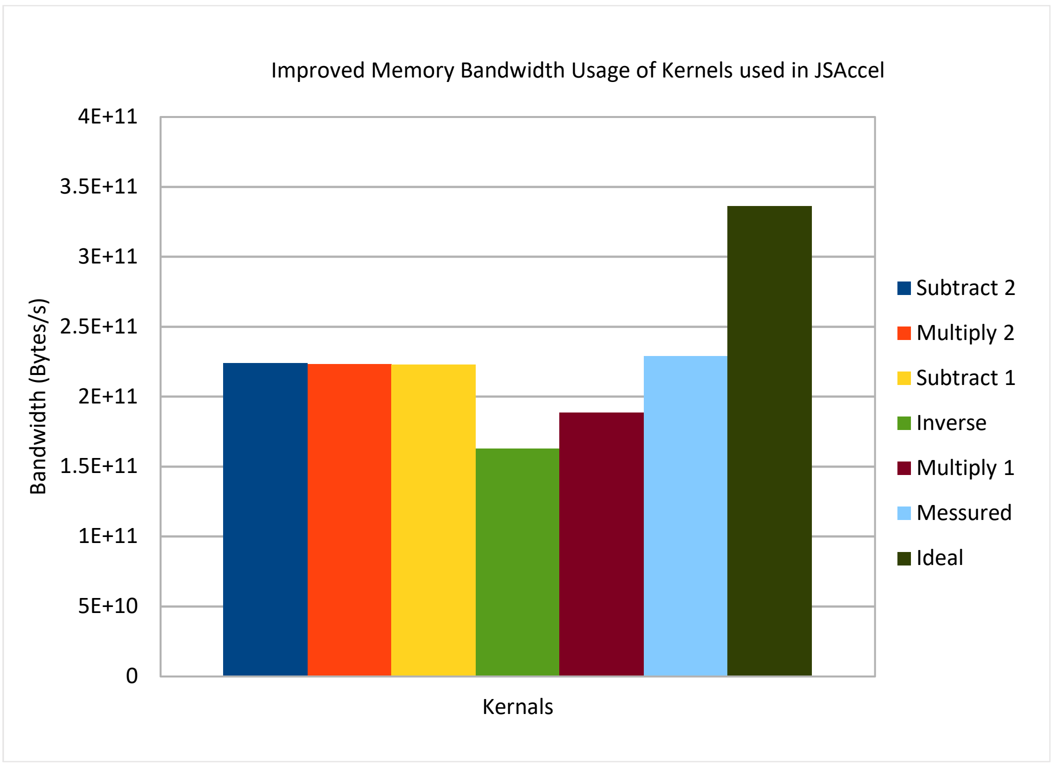


Figure 55: Final memory bandwidth usage

6.9 Compute time reliability

During testing of the GPU accelerated solvers it was noted that there was often significant variation in the compute time. In order to test this variation a collection of tests were performed on the reference hardware. 15 test runs were performed one after the other with no other interaction with the hardware. This should give a good indication of the variation seen. In the first instance the 315k cell test case referred to above was run for one second. This represents a compute time of around 7000 seconds.

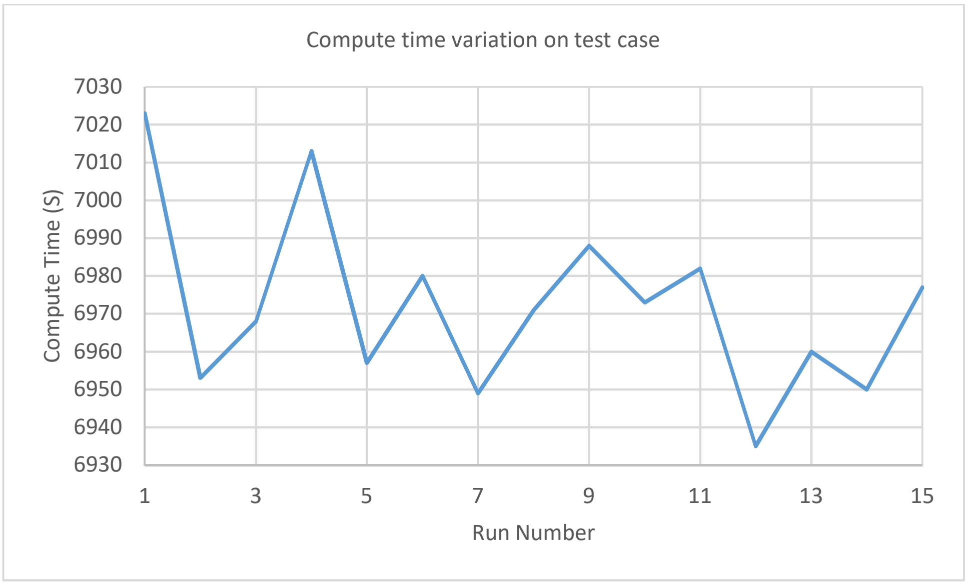


Figure 56: Compute time variation on test case

Figure 56 shows the variation on total compute time across the 15 tests cases, the order is the same as was conducted on the reference hardware. The variation seems not to come from change in real time. This could be expected to happen from additional residual memory usage by the operating system or with memory not being deallocated from previous runs etc. However, this is clearly not the case as there is no trend over real time. Indeed, the fastest run is the 12th of 15. While the variation is not hugely significant at approximately 1.2% it does remain unexplained.

An additional comparison was made on the much higher five million cell count though this was only run for two time steps to save real time. This produced a much greater variation in compute time.

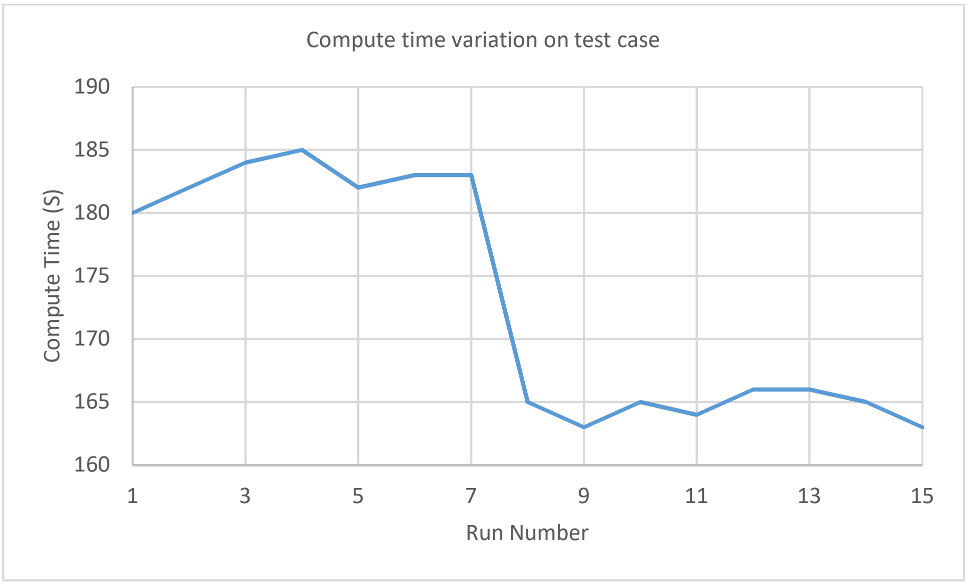


Figure 57: Compute time variation among higher cell count test session

As can be seen in figure 58 there is clearly more variation across real time, but the variation is far more significant, in the region of 12%. Looking into more detail at the breakdown of the compute profile it's possible to compare different sections of the code.

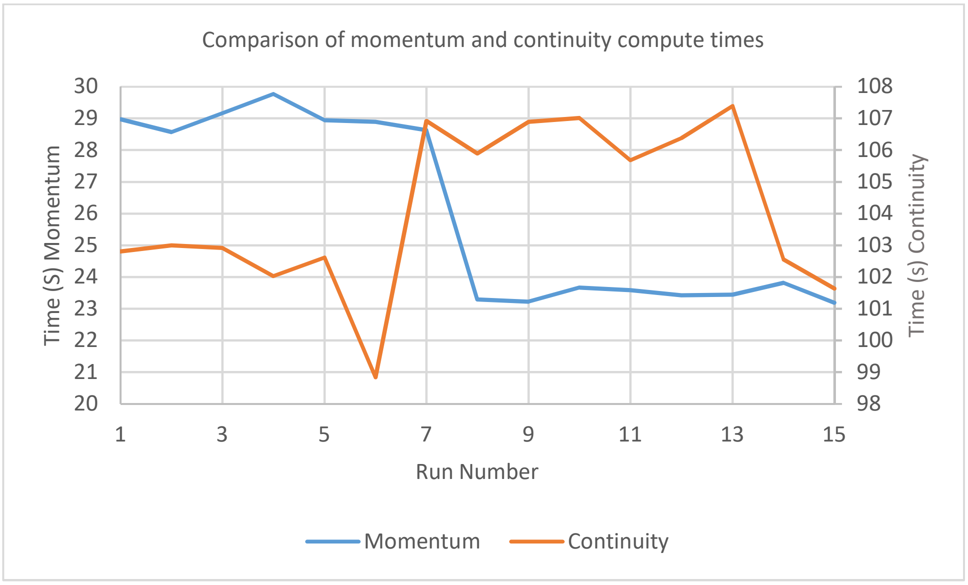


Figure 58: Compute times of continuity and momentum solvers

In figure 59 it can be seen that the plot of the momentum solver follows that of the whole solver compute time, however the continuity solver is much more varied.

While the variation of the momentum solver compute times is similar to the variation in the overall compute time the magnitude isn't accounted for. Though interestingly this demonstrates that the variation is unlikely to be coming from either specifically GPU or CPU compute processes. To investigate the possibility of the CPU computation causing the errors the Linux perf program was used to check the number of page-faults and cache misses. Both of these have the potential to increase the computational time of each run. The results showed that variation in these was minimal, page-faults and cache misses only varied by 0.02% across all the cases. This doesn't equate to the 12% difference in compute time. As this variation remains unexplained care should be taken to make comparisons over long compute time to reduce the variation in run time or if short compute time are required multiple computes should be performed and averaged.

6.10 Chapter Summary

This chapter has shown that interFoam performs well on a low Reynolds number liquid injection case. Indeed in the key metric of jet centreline velocity the computational results are close to experiment and in some areas are also better than alternative level set methods.

Following this, GPU accelerated linear solvers were developed for both the pressure and momentum equations. These showed a doubling of speed over the equivalent CPU linear solvers. Although a decent speedup there is room for improvement, the speedup of the pressure equation does not improve with increasing cell count. This type of improvement would be expected from a GPU as the additional data means the GPU cores can be kept occupied better. The high level of memory transfer in this implementation harms this performance and needs to be removed. Finally a compute time variation has been found that has a significant effect on short compute time problems and although smaller has an effect on longer compute time comparisons. Some explanations for this were investigated but a definitive conclusion was not found however recommendations for future investigations were suggested such that compute comparisons should be made long enough that the uncertainty is minimised.

Chapter 7 Sheet atomization case

7.1 Overview

This chapter will set out a much more complex and industrially significant problem in primary atomization. Bulding on the low Reynolds number case presented in the previous chapter. This test case will then be used to assess the suitability of GPU accelerated computing to computationally expensive industrial atomization problems.

7.2 Establishing a CPU benchmark

The case presented by Deshpande et al., (2015) is used as a basis. While the CFD modelling of the primary atomization of a liquid sheet is used as a comparison to other techniques, it does represent a near to real case. The primary atomization problem consists of a thin sheet that is injected into quiescent air at high velocity (200m/s) and therefore typically has a high Reynolds number. In this investigation the average liquid breakup length is compared across a number of different density ratios. This ratio is achieved by varying the ambient gas density the liquid is injected into. The cell density used puts the simulation in the DNS regime. The operating parameters are outlined in Table 9 below.

Liquid Density	666.7 kg/m^3
Liquid viscosity	$2.5 \times 10^{-4}\ Ns/m^2$
Gas Density	$39.22\ kg/m^3$
Gas Viscosity	$4.06 \times 10^{-5}\ Ns/m^2$
Coefficient of surface tension	$0.02\ N/m$
Sheet thickness	$200\ \mu m$
Injection speed	$200\ m/s$

Table 9: Operating conditions of sheet atomization

Dimensions are normalised by half the sheet thickness (h). The overall size of the domain used in the reference case is described as $90h \times 80h \times 270h$. With a maximum cell density of 12 cells per h resulting in a total cell count of 3.359×10^9. Because of this some economy of cell count is required in order to run this problem on the available hardware. However, this huge cell count re-enforces the need to investigate alternate computing platforms to reduce compute cost on these kinds of problems.

The reference case investigates multiple density ratios which are inversely proportional to the length of the sheet. The first economy can be made by only considering the highest density ratio. Indeed Deshpande et al., (2015) states “In order to show breakup length for the lowest density ratio … the domain was made sufficiently long in the stream wise direction”. Plots of intact liquid core length show that on average the highest density ratio sheet core-length is three times longer than the smallest. Therefore, the overall domain length is reduced by half to allow a reasonable section of no longer intact liquid core.

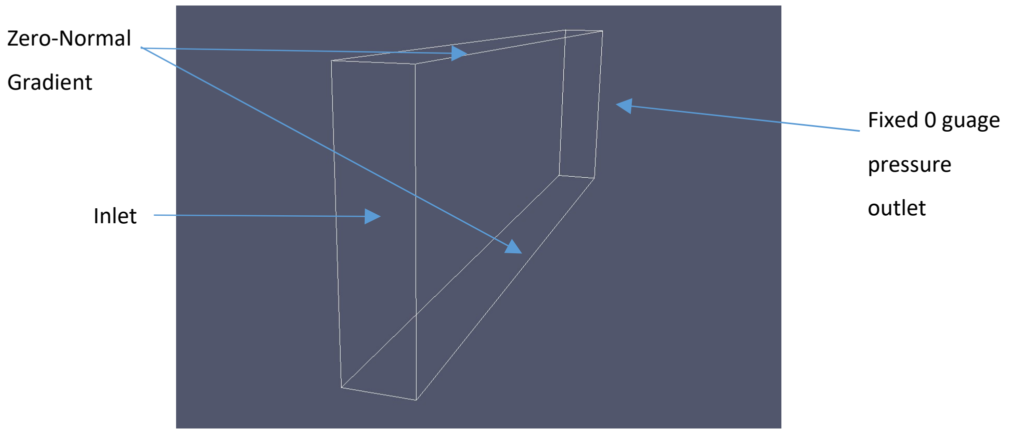


Figure 59: Outline of boundary conditions used

Figure 59 describes most of the boundry conditions used. The inlet is defined as a fixed velocity in a central slit equal to the sheet thickness ($2h$) of 200m/s. At the span-wise boundaries periodic conditions are applied. This allows for a significant reduction in the span wise dimension. Compared to the domain published, the one in this study is half the size in the stream wise direction and 1/9 of the size in the span wise direction. The remaining direction is also halved, after initial results showed that after breakup insignificant quantities of liquid could be found near the vertical boundaries (as show in Figure 60). This gives an overall domain of $135h \times 40h \times 10h$ where 2h is the sheet thickness.

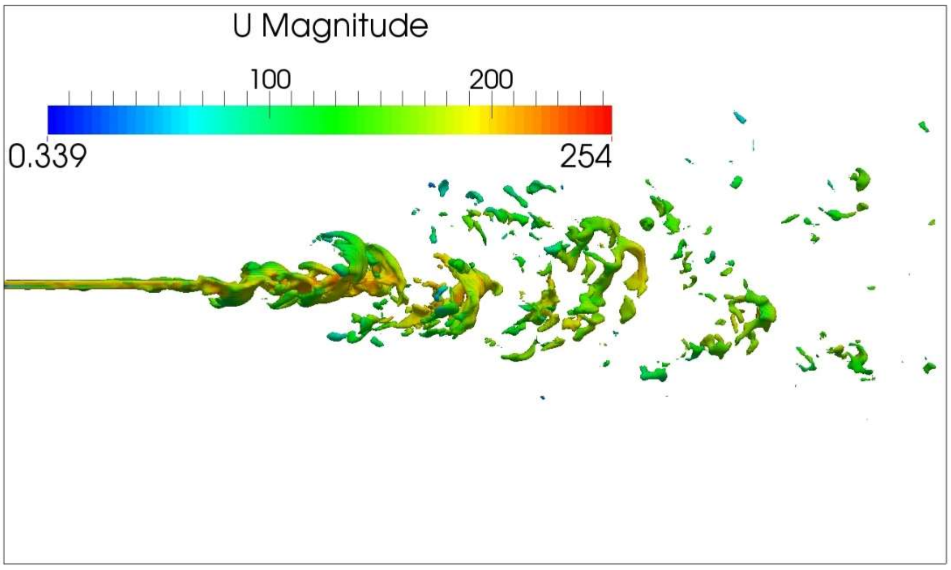


Figure 60: Sheet visualisation taken at 4×10^{-4}s. Iso contour of volume fraction = 0.5. Coloured by velocity in m/s

To confirm that this economy of cell count does not adversely affect the physical results of the simulation, the breakup length (X_{liq}) was compared. Deshpande et al., (2015) presented a plot of intact liquid core length over time for the various gas densities tested. They also presented a plot of how the intact liquid core length varies with mesh density. Using both of these an estimate of how the intact liquid core length varies over time on the coarsest mesh density can be plotted using the offset as shown in Figure 61.

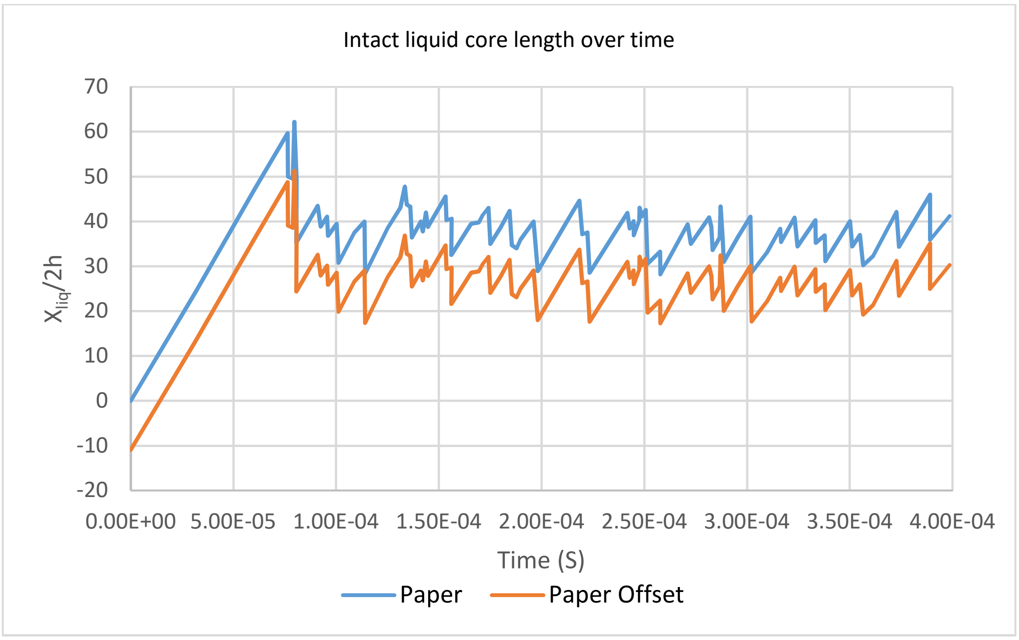


Figure 61: Graph of data presented by (Deshpande, et al., 2015) and what would be expected by using the the offset related to the reduced number of cells

This offset graph does not well represent the linear regime because a simple offset is used giving a negative value for intact liquid core length. However, the quasi-steady state period has an indication of the variation around the average intact liquid core length. The method for establishing the intact liquid core length is not well explained. So, for comparison a slice is cut through the centre of the domain and the point at which there is no longer a chain of volume fraction greater than 0.5 is taken as the intact core length.

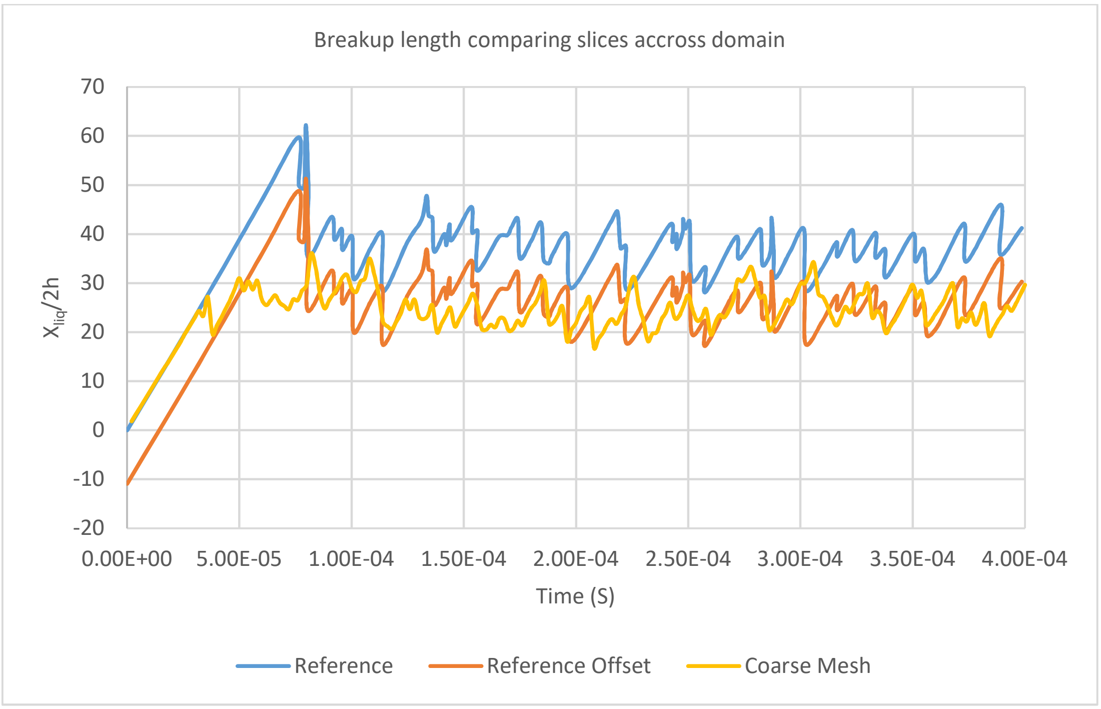


Figure 62: Comparison of data and data offset with results from this work

From these results it can be seen that in the quasi-steady state period the agreement between the results offset for cell density and those in this study is quite good. The average intact liquid core length in this region is very similar. However, the initial transient, linear growth regime results in a much shorter intact liquid core length. To investigate the effects of the different boundary conditions used, a comparison was made by doubling the domain length and also increasing the width, from 10h to 30h, and removing the cyclic boundary conditions. This gives a distance of 10h between the liquid sheet and the boundaries on the sides of the domain. As a result, the same ratio of sheet width to gap size between the sheet and boundary is obtained as is used in the reference case.

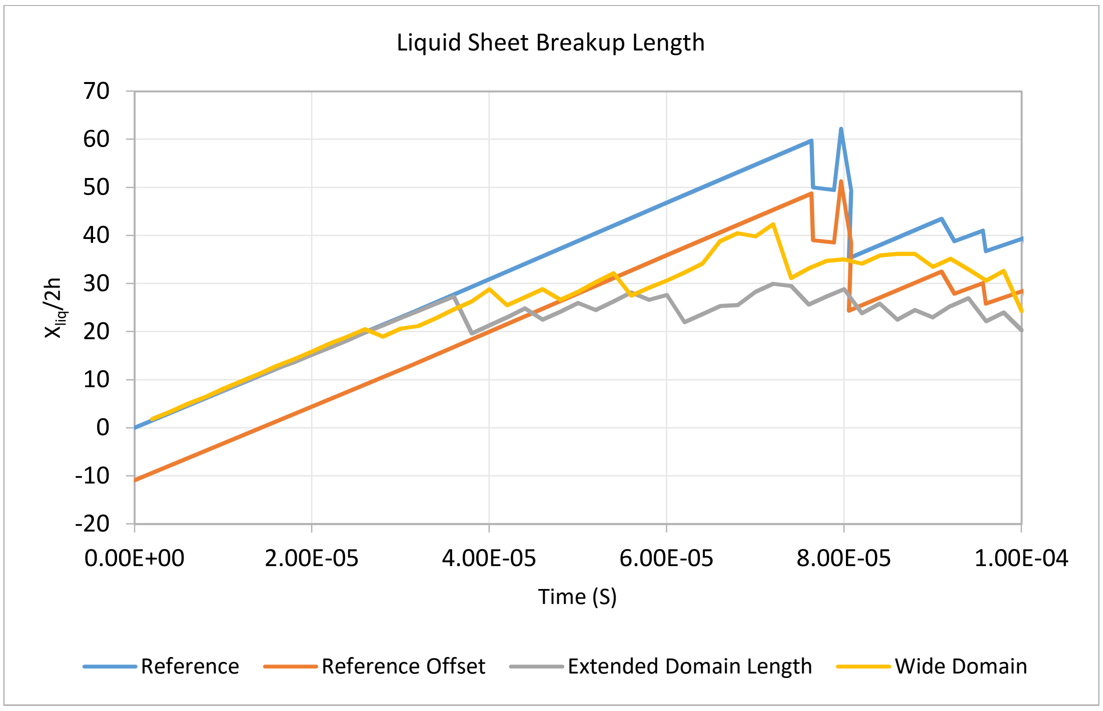


Figure 63: Comparison between alternative boundary conditions on intact liquid core length

This comparison shows that the downstream outlet boundary condition has an insignificant effect on the transient linear growth regime. However, increasing the domain width as well as using outlet boundaries with a gap between the jet and boundary noticeably increases the length of the jet at the end of the linear growth region.

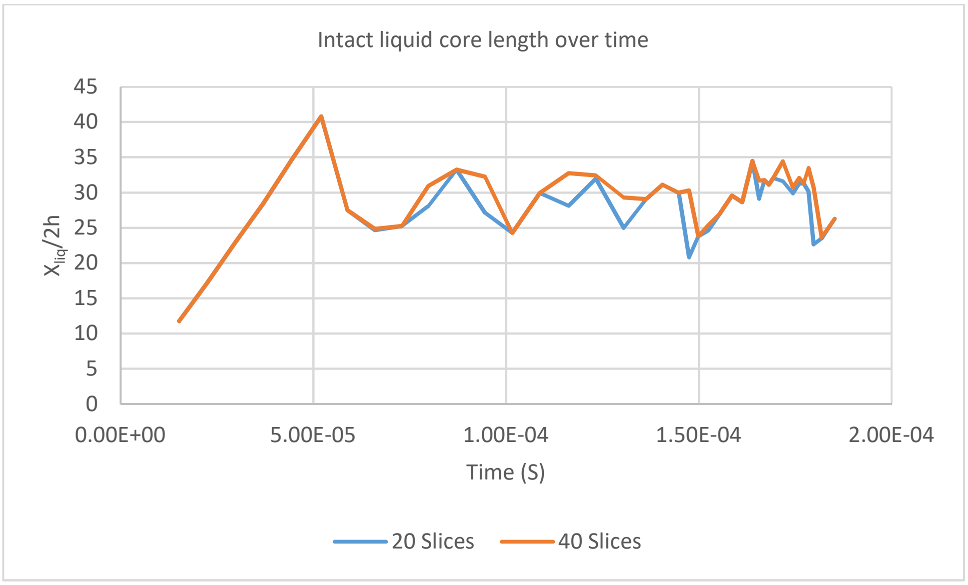


Figure 64: Breakup Length using multiple slices

When the cell density was increased the expected increase in intact liquid core length did not occur. This lead to the possibility that using a single slice to calculate the intact core length was not suitable. The possibility of there being span wise breakup could account for this lower than expected result. To investigate this further the number of slices used to assess intact core length was increased so multiple positions across the domain could be probed. The result from using 20 and 40 slices is shown in Figure 64 above. 20 slices gives one slice every four cells with 40 giving one every two cells in the span wise direction. The intact liquid core length is then evaluated on each slice, this gives the intact liquid core length at discrete points across the domain. The maximum intact core length can then be plotted for each timestep.

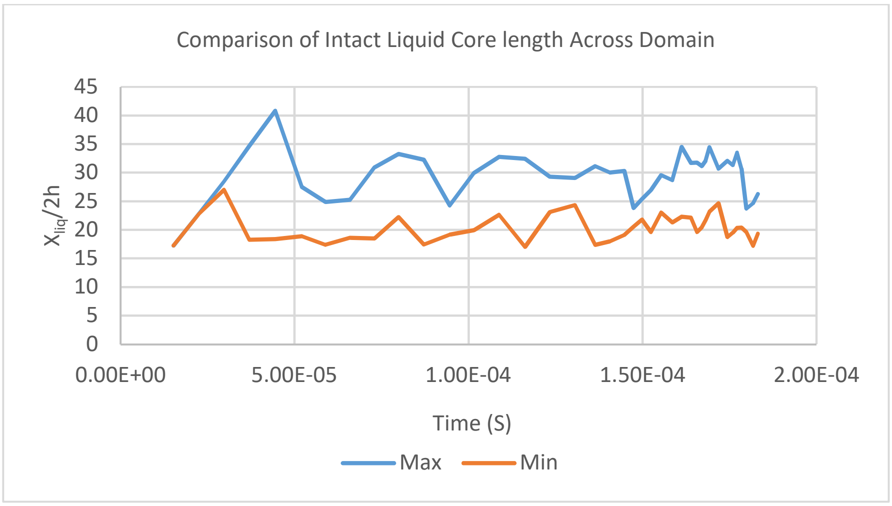


Figure 65: Variation of intact liquid core length across domain, using maximum and minimum intact core length at discrete points across the sheet width

Figure 65 shows the maximum and minimum intact liquid core length among 40 slices across the domain. It can be seen that for only a short period in time there is no variation across the domain but clearly throughout most of the simulation time there is significant variation across the width of the sheet. This indicates that there is significant span wise breakup in the sheet while there is still an intact core.

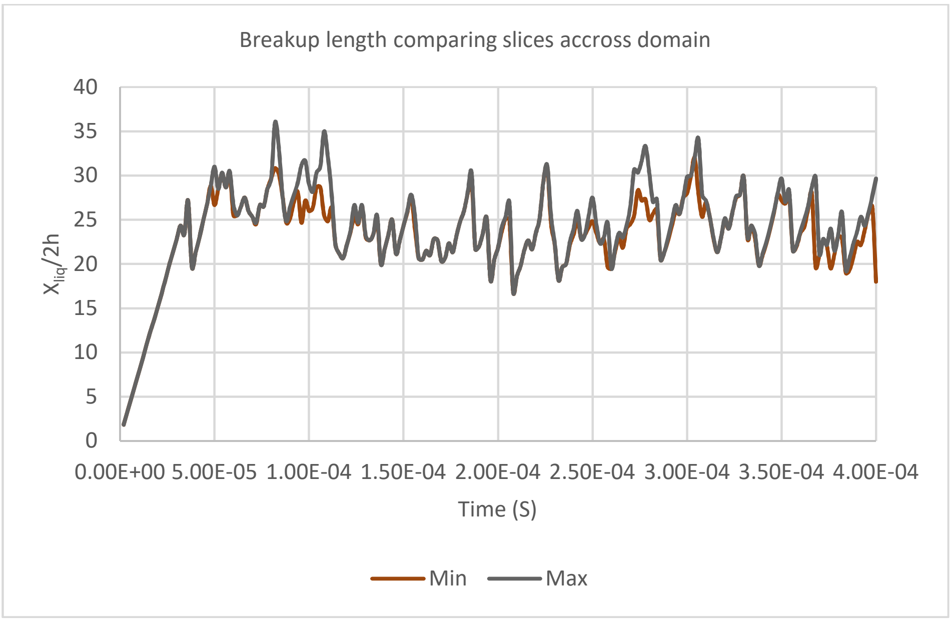


Figure 66: Variation in breakup length in spanwise direction on coarse mesh, using maximum and minimum intact core length at discrete points across the sheet width

Using the same method as previously described on the fine mesh, this time on the course mesh with a slice taken every 3 cells gives Figure 66. As opposed to the fine mesh where a significant difference is seen here there is very insignificant variation. Indeed, for large portions of simulated time the maximum and minimum values track each other almost exactly.

The difference between the dense and course mesh can be further visualised in the Figure 67 below. The first line drawn on the dense mesh shows the start of clear breaks in the liquid sheet, but not across the whole span. The second line and the line drawn on the coarse mesh show clear breaks across the whole span.

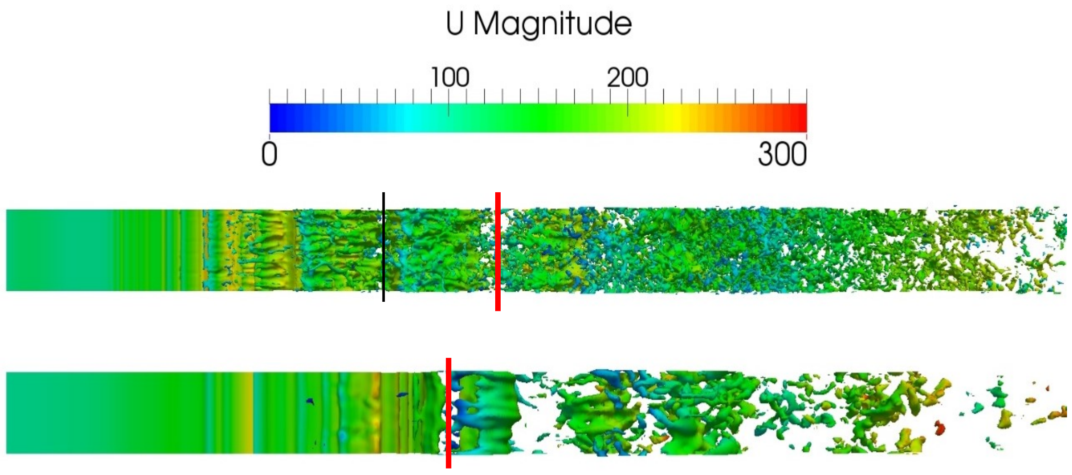


Figure 67: Iso contour of volume fraction = 0.5 on dense mesh (top) and coarse mesh (bottom), coloured by velocity in m/s

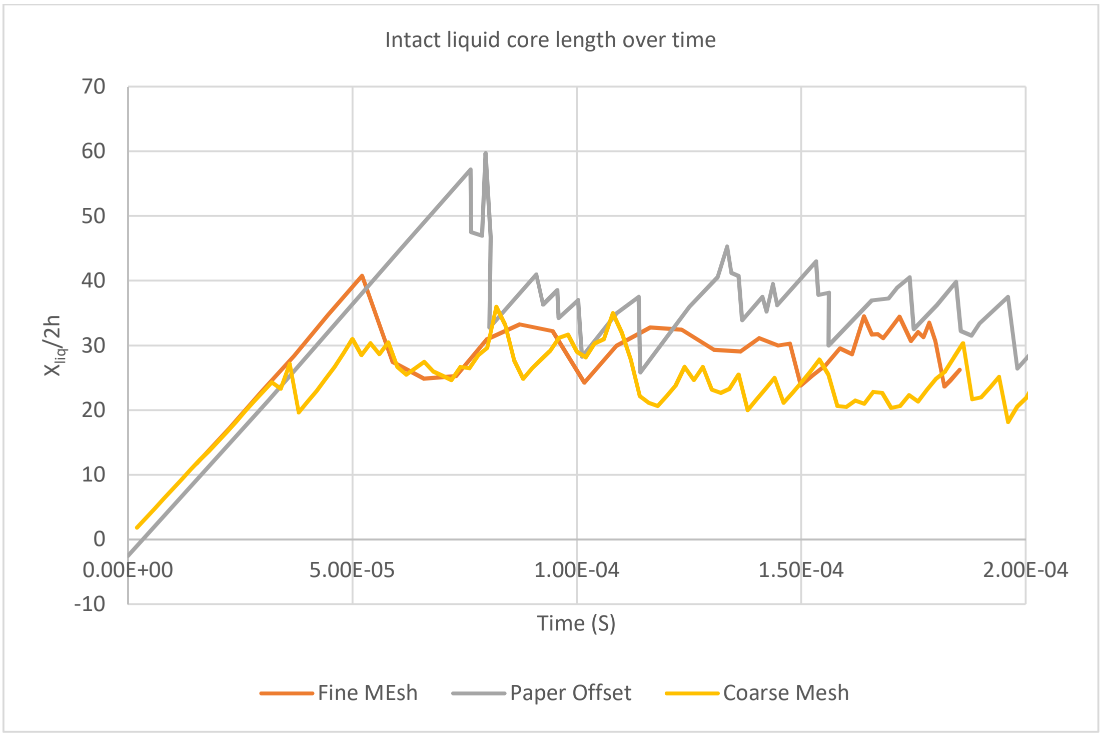


Figure 68: Comparison of intact liquid core length over time using published results and results from this study

The variation in intact liquid core lengths across the domain width is compared for both course and fine meshes in Figure 68. As expected the fine mesh produces a slightly lengthened intact liquid core length, although this lengthening is not as great as previously published. For the present application however, it is deemed acceptable.

Finally, a comparison was made in varying the span wise dimension and therefore the width of the injected liquid sheet. Figure 69 shows there is little significant difference among the among the 5 different domain widths tested. In all cases the domain width is described in terms of h defined at the beginning of this chapter.

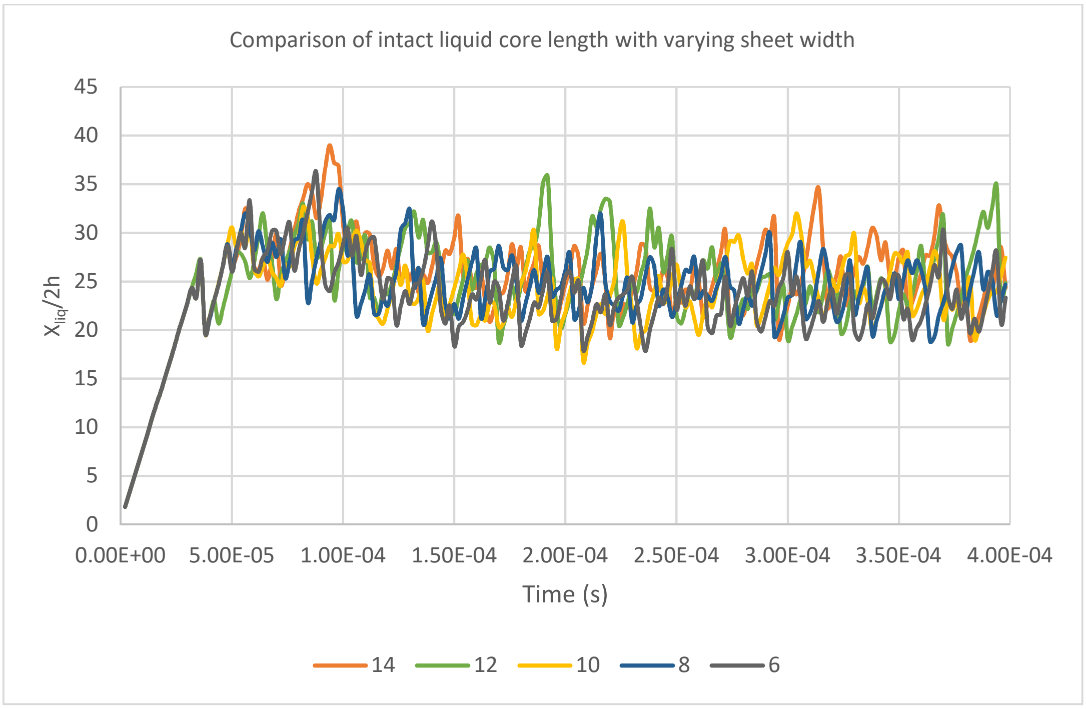


Figure 69: Intact liquid core length with varying domain width

7.3 Chapter summary

This chapter has shown that a complex sheet atomization case can be well represented on the available hardware using alternative boundary conditions and reduced domain size. While there are some differences, the average intact liquid-core length is well captured. However, a clear difference in the transient growth regime was found. Investigation into this issue showed that using periodic boundary conditions had the effect of reducing the peak intact liquid core length at the end of the linear growth regime. Additionally, it was shown that the coarsest mesh does not show the significant span-wise breakup that was found in the finest mesh. Finally, it was shown that changing the span-wise domain dimension showed no significant effect on the intact liquid core length.

Chapter 8 Multigrid Solver Running Entirely on GPU

8.1 Overview

This chapter will present a much improved GPU accelerated linear system solver, extending the solver presented in chapter 6 to a full multigrid method with the aim of reducing memory transfers between the CPU and GPU. First the solver will be explained with the key points detailed. Next performance tests will be made on the sheet atomization case presented in the previous chapter.

8.2 Solver Outline

As has been discussed previously the conjugate gradient solver is fairly good and efficient but it works best with either a multigrid preconditioner or just using a multigrid method can be even faster. Because of this and as very few previous investigations have reached the point of running a multigrid linear solver it seems a natural progression for this work. As touched on in the earlier chapters a multigrid solver consists of four main components. The first is a smoother, then there are ways of transferring results to course grids, transferring results from course to fine grids and a coarsest grid solver.

At least in the first instance the solver produced in this study uses a damped Jacobi smoother, injection for interpolation, agglomeration for restriction and conjugate gradient as the coarsest level solver. A V cycle is also used for grid cycling. Damped Jacobi is used as it performs well when ported to GPU. There is no clear consensus as to the most suitable smoother to use in GPU accelerated multigrid methods. Indeed Liu et al., (2015) showed that the damped Jacobi method performs better on GPUs than multiple other options. Others simply state that as Gauss-Seidel has faster convergence it is better to use without consideration for the alternatives e.g. (Reddy & Banerjee, 2015). As discussed in earlier chapters the conjugate gradient solver was originally developed as a direct solver and its convergence is guaranteed with as many iterations as there are unknowns. This makes it ideal for a situation where the number of unknowns could vary in addition this is also the option used by OpenFOAM. The prolongation and restriction operators were selected so as to keep the solver similar to OpenFOAMs.

8.3 Starting with a two-level multigrid solver

To begin with a two-level solver was developed. In this instance, the solver may also be referred to as a course grid corrector. Here two levels are used, the original fine grid and a generated course grid. The course grid generation method is similar to that outlined in (Versteeg & Malalasekera, 2007), i.e. a course grid cell consists of the sum of eight fine cells. This is generated as a sparse matrix in compressed sparse row format. This is then passed to the NVidia CUSP library (Nvidia Corporation, 2014). The CUSP library is designed for efficient GPU computation with sparse matrices. The CUSP library transposes the interpolation matrix to give a restriction matrix. The Galerkin product is then used to generate the course grid. The Galerkin product is shown below, R is the restriction matrix, I the interpolation and A the linear system matrix. The subscripts denote the level on which the matrix applies.

$$A_{k+1} = I_k^T A_k I_k \tag{25}$$

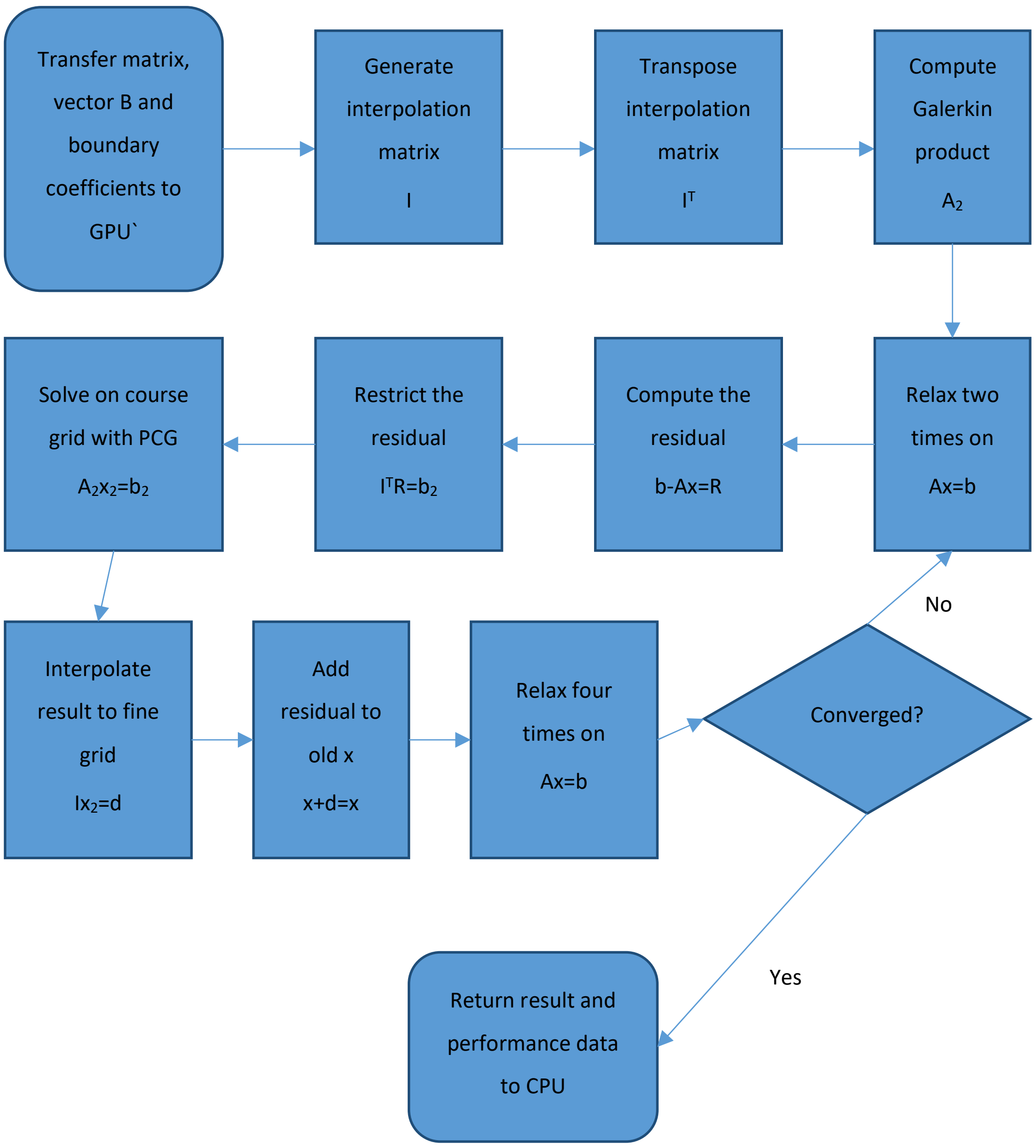
In order to produce the Galerkin product the CUSP library will first convert each matrix to the COO format. The COO format is more suited to the matrix multiplication process whereas as discussed previously the CSR format is better suited to matrix-vector products. The cost of this change is small and so worthwhile (Bell, et al., 2012). This consists of what is commonly known as the setup phase.

After the setup phase comes the solve phase, this section is iterative. First comes the pre-smoothing, a damped Jacobi method (damping of 2/3) operating on the fine grid. The error from this pre-smoothing is then restricted onto the course grid using the restriction matrix. A diagonally pre-conditioned conjugate gradient method is then used to solve the linear system that arises from the error equation, equation 26, shown below:

$$A_{k+1}X_{k+1} = e_{k+1} \tag{26}$$

The result of this iterative process, after interpolation, gives an estimation of the error on the fine grid. Therefore, the interpolated X vector is added to that generated in the pre-smoothing. Finally post smoothing, again damped Jacobi is used, is applied on the original linear system using the new X starting vector. The solve phase is applied iteratively until the convergence criteria is reached.

One additional point of note concerns the boundary condition correction for the two cyclic boundaries. OpenFOAM treats these boundaries as if they were the boundaries between processors in a multi process operation. Thus, coefficients are generated and the correction is applied part way through the iterative solver. A similar approach is used in the present solver. The interface coefficients generated by OpenFOAM are passed to the GPU and a correction is applied after every matrix vector multiplication. On the course grid the original coefficients are restricted in a similar way to the transfer of vectors from fine to course grids.



Transfer matrix, vector B and boundary coefficients to GPU`
Generate interpolation matrix I
Transpose interpolation matrix I^T
Compute Galerkin product A₂
Relax two times on Ax=b
Compute the residual b-Ax=R
Restrict the residual I^TR=b₂
Solve on course grid with PCG A₂x₂=b₂
Interpolate result to fine grid Ix₂=d
Add residual to old x x+d=x
Relax four times on Ax=b
Converged?
No
Yes
Return result and performance data to CPU

Figure 70: Flow chart of two level multigrid solver

8.4 Testing two level solver on sheet atomization case

The two level solver was applied to the sheet atomization case outlined in the previous chapter initially one time step was performed to understand memory bandwidth usage. The results of this analysis are shown in Figure 71. Bandwidth usage is around maximum in the vast majority of the routines used. Exceptions are the boundary correction routines, matrix vector multiplication and the restriction and interpolation operations. The boundary correction performs poorly as by its very nature there is non-coalesced access to the linear system matrix. This is because only the cells an X-dimension width apart are accessed. Matrix-vector multiplication also doesn't perform to the full capability of the GPU as the number of non-zero values in each matrix row varies. Therefore, in some cases a memory fetch operation will collect all data required while others will need multiple fetches.

Also of note is that the matrix-vector product represents around 45% of computational time for each time step. As this operation is only using about 33% of the GPUs theoretical bandwidth the acceleration over CPU will suffer.

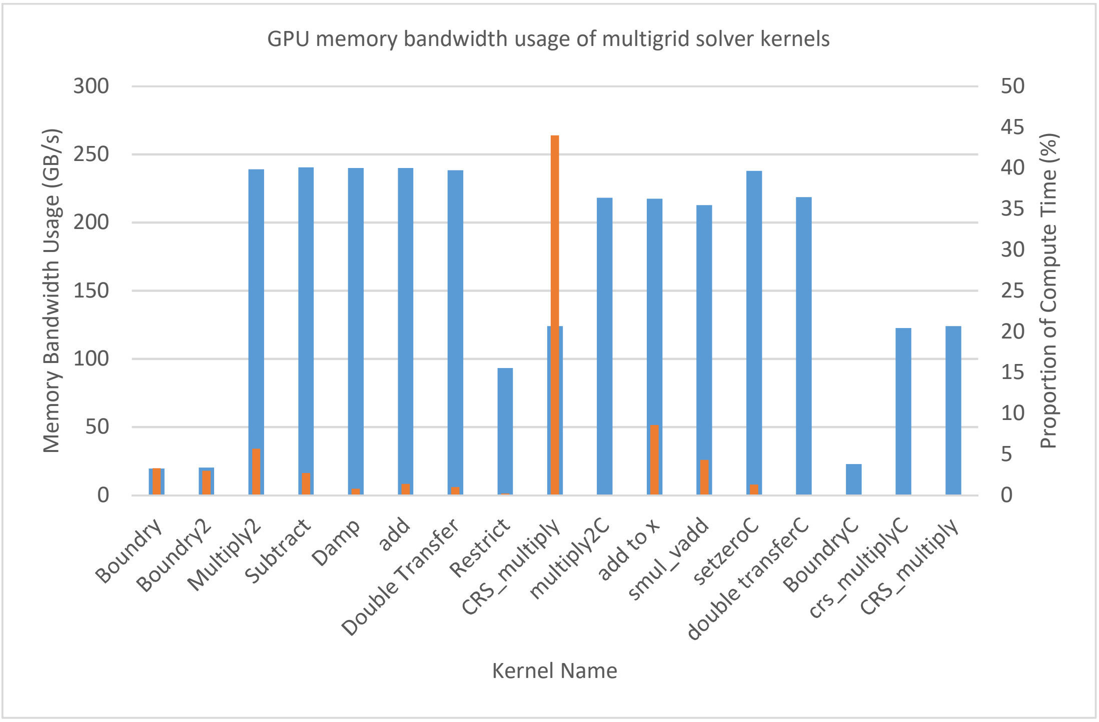


Figure 71: Compute analysis of AMG 2 level solver

In addition to investigating compute parameters it is also necessary to validate the results produced by the new solver. This was done in two ways. The first was to directly compare results of the linear solver with an OpenFOAM equivalent setup to behave in a comparable manner. Comparing the result of each cell after one-time step gave a maximum percentage difference of 0.005%. This shows that although not equal to machine accuracy the difference in result is well within a tolerable level. The difference can also come from minor differences in the computational algorithm used by each compute unit. For example, OpenFOAM uses Gauss-Seidel as its smoother and doesn't have a damped Jacobi option.

For the second method of validation the breakup length is computed and compared for each compute unit. This gives a comparison of how the codes compare over many time steps.

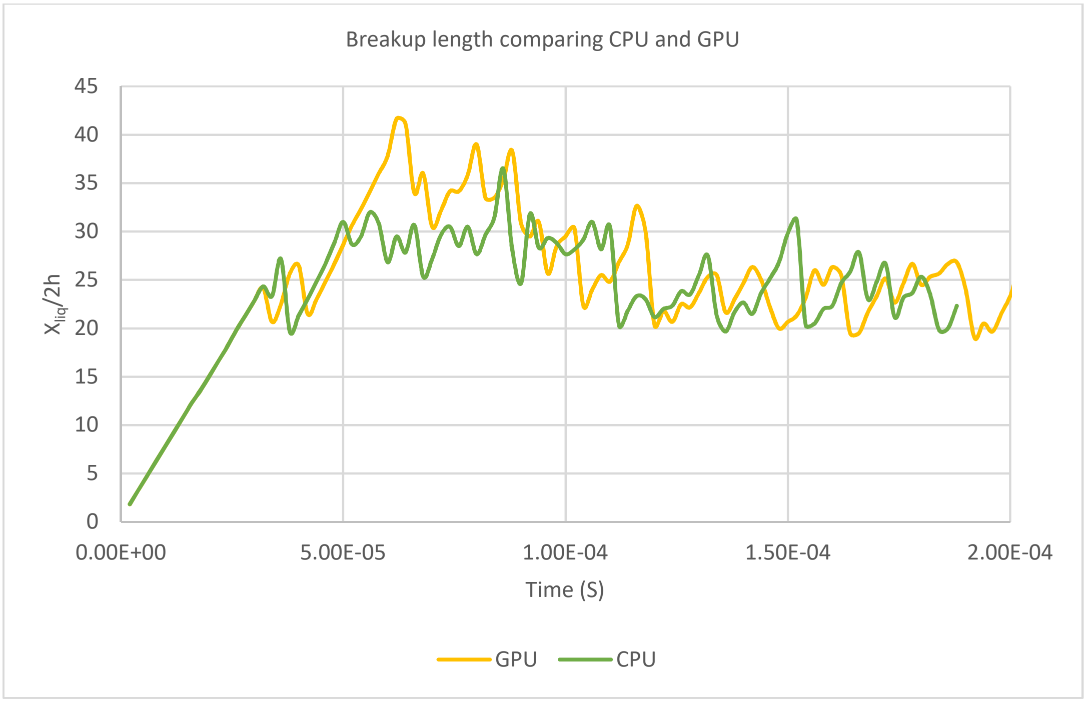


Figure 72: Breakup length comparison between CPU and GPU

The results of breakup length comparison showed a difference in the results obtained from the GPU as compared to the CPU. Upon investigation the GPU solver overall proved to be marginally less stable resulting in slightly increased courant number. This leads to requiring additional time steps, 1 in 97, again the differences in algorithm will account for this.

With this in mind, the reference case was re-run using a fixed time stepping method, 1e-8s being the applied Δt. The case was run for 40-time steps with the result in each cell directly compared. The average difference among all cells in the domain after 40-time steps was found to be 0.0011% with a maximum difference in one cell of 0.0024%. This demonstrates that the GPU code offers good accuracy. Indeed, with reducing the residual criteria for each time step the differences between GPU and CPU would decrease further.

Originally the multigrid solver was run using two pre and four post smoothing iterations. However the number of smoothing iterations has a direct impact on the number of iterations the multigrid solver takes to reach the convergence criteria. With this in mind a test was conducted to find the optimum number of smoothing iterations required for this problem. Each test was conducted using one time step and repeated 25 times to account for compute time variation. The averaged results are shown below in Figure 73.

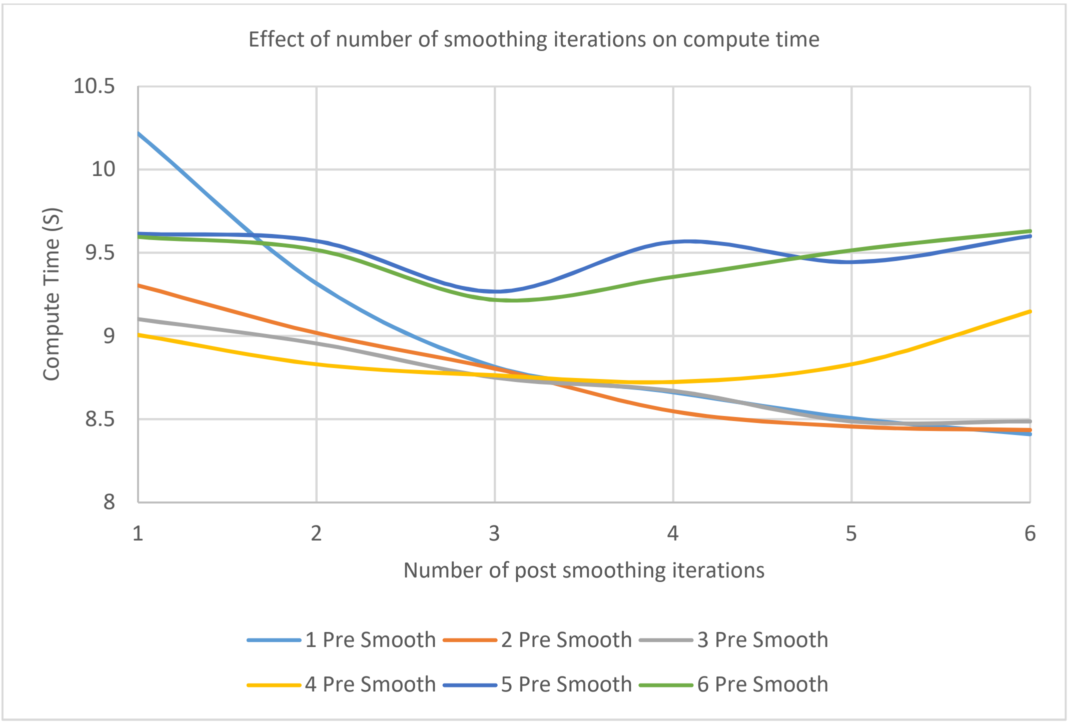


Figure 73: Compute time variation using different numbers of smoothing iterations

The variation in compute time comes from the reduction in overall number of iterations the multigrid solver requires. A smoothing iteration is relatively cheap compared to full multigrid iterations. This effect is clear from the results, the slowest configuration is that with one pre and post smoothing iteration. This configuration requires 39 multigrid iterations to reach convergence. However, if one pre and six post smoothing iterations are applied only 19 multigrid iterations are required for convergence and compute time is reduced by about 17.5%. The original usage of two pre and four post smoothing iterations is marginally improved by increasing post smoothing iterations to six.

Speedup is compared using the fixed time stepping method mentioned previously. This gives as like for like comparison as possible. The GPU coarse grid correction method gave an overall speed up of 1.93 over its CPU counterpart. This was found by comparing the total wall clock time of each run, which includes non-accelerated code. Additionally, splitting the total clock time down into the amount of time taken to compute pressure a speedup of 2.44 was found.

8.5 Comparison of speed on higher cell counts

To assess how the GPU performs in situations with higher cell counts three different cell densities were compared. In all cases 40-time steps were used to produce the comparison.

Cell Size	Number of Cells	Total speedup	Speedup of pressure corrections
h/3	1468800	1.926	2.444
h/4	3456000	2.539	3.386
h/6	11664000	3.520	4.831

Table 10: Comparison of speed up with varying cell counts

Table 10 shows the speedup obtained by using the two-level solver on higher cell counts. Cell size is expressed as in Deshpande et al., (2015) in non-dimensional form related to the sheet thickness. Ideally, the even more dense mesh using a cell size of $h/8$ would have been investigated as this is the largest case capable of being run on the reference hardware. However, the GPUs limited available memory was not large enough to accommodate this. The $h/6$ case represents a usage of about

3.8GB of the GPUs available 6GB of memory. This represents a consumption of 326 bytes per cell meaning the maximum possible cell count that can be run on a single GPU is around 18.5 million cells.

To understand the improvement in speed up over the CPU again memory bandwidth usage was calculated, this is show in Figure 74 below.

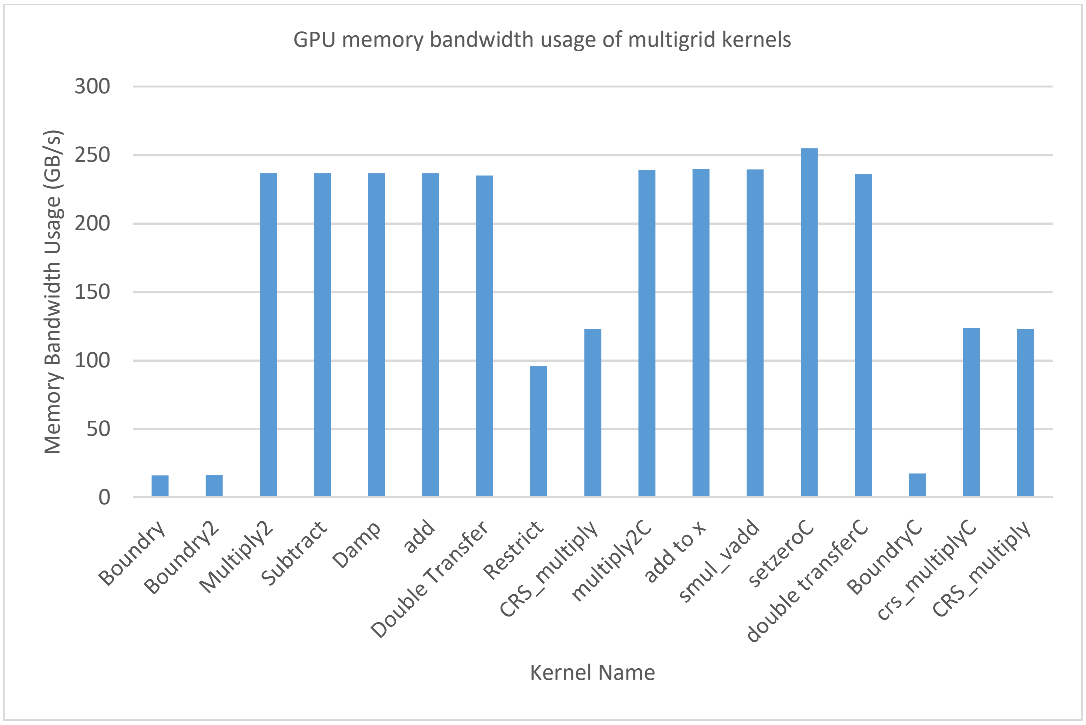
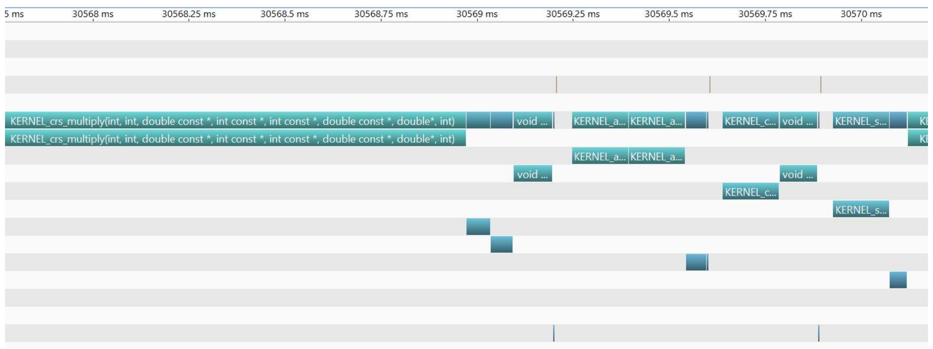


Figure 74: Memory bandwidth usage of h/6 cell size

The memory bandwidth usage of the high cell count case is quite similar to that in the standard case, though with some slight increases. Indeed, only the course level kernels show any real increase but this is only by a few GB/s.



5 ms
30568 ms
30568.25 ms
30568.5 ms
30568.75 ms
30569 ms
30569.25 ms
30569.5 ms
30569.75 ms
30570 ms
KERNEL_crs_multiply(int, int, double const *, int const *, int const *, double const *, double*, int)
KERNEL_crs_multiply(int, int, double const *, int const *, int const *, double const *, double*, int)
void ...
KERNEL_a...
KERNEL_c...
KERNEL_s...

Figure 75: Illustration of GPU timing

Figure 75 above is taken from the NVidia visual profiler. Shown are the kernels involved in one preconditioned conjugate gradient iteration. In each iteration, there are three noticeable gaps in compute time. These take place after vector operations such as sum and dot product. These operations use the CUBLAS library, the GPU accelerated version of BLAS (Basic Linear Algebra Subprograms). The result of these operations is a scalar returned to the CPU. This return operation is a synchronisation point and so the next GPU kernel cannot be launched until the return has completed. Additionally, the time cost of launching a kernel is not zero, and this must take place after the scalar is returned. This gap is small, of the order of 0.05ms, but occurs many times. This cost remains constant no matter how many cells are used in the problem under investigation. Therefore, in lower cell counts these gaps represent a significant portion of the compute time. But at higher cell counts where the number of iterations doesn’t really change these gaps diminish in percentage of overall compute time. In addition, the CUBLAS API is linked at its first usage rather than purely at compile time. Again, this has a cost of around 0.1s, though this only takes place at the start of computation so its effect when multiple time steps are involved becomes minimal.

Analysing compute and memory transfer timing shows further details of how these idling periods effect the overall compute time.

Cell Size	Compute	Host to device transfer	Device to host transfer	Device to device transfer
h/3	60.8308	1.36823	0.90981	0.41678
h/4	72.7214	1.43959	0.56024	0.48760
h/6	85.2864	1.20839	0.24313	0.47435

Table 11: Percentages of GPU compute time for one time step

Table 11 shows the percentage of GPU compute time for several classes of GPU operation. Memory transfer operations remain a fairly constant proportion of overall compute time. However, the compute proportion increases significantly. This reinforces the notion that the idling time remains constant with the longer compute times reducing the proportion idling time accounts for. In the coarsest mesh idling accounts for about 35.5% of the total compute time but with the finest mesh this decreases to just under 13%.

8.6 Multi-level solver

Following the testing of the course grid correction method the AMG solver was further developed to use multiple grid levels. Because of the mesh dimensions only the $h/4$ mesh case is considered. This allows the coarsening to take place on all mesh dimensions with the coarsest level still having an integer value of cells.

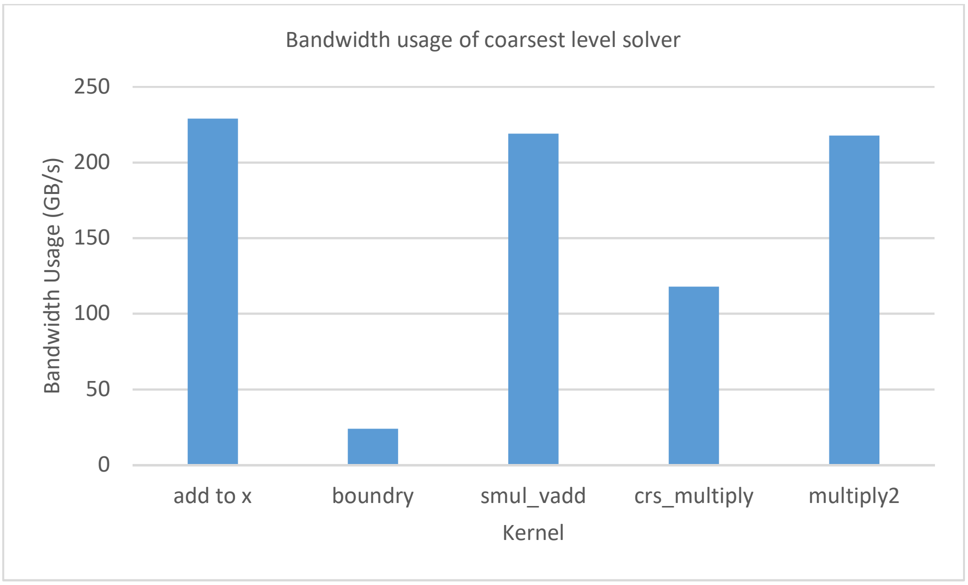


Figure 76: Memory bandwidth usage of coarsest level in two level solver

Figure 76 shows the memory bandwidth usage of the coarse level solver. The remaining levels are not assessed as they will perform much the same as previous examples. However, the coarsest level is of interest as it is now only computing the result of a 54,000 cell mesh. This is significant as keeping the GPU fully saturated with smaller and smaller numbers of threads can become a problem.

Initial comparisons on speedup were disappointing with no significant difference between the CPU and GPU results. As previously these measurements were taken from the overall CPU wall clock time, on a 40 time step compute run. The cause for this poor result seems to partially be down to additional iterations per time step. The CPU method only requires an additional one or two iterations per time step as compared to the course level corrector. On the other hand, the GPU required a factor of over 3 times as many iterations. This results in an overall time reduction of 10% for the GPU accelerated solver, whereas the CPU solver is 2.67 times faster.

To further investigate the effects of the different configurations of the OpenFOAM implementation and the GPU implementation a Jacobi smoother was written and implemented in OpenFOAM. An initial test showed that while the Jacobi smoother reduced computational efficiency to some extent it was by nowhere near as much as would be expected. Trottenberg et al., (2001) states that the smoothing factor should be 2.67 times greater. An arbitrary selection of time step showed that the Jacobi smoother takes an additional 6 multigrid iterations to converge, going from 21 to 27. This contrasts significantly with the GPU multigrid iterations of 73. Therefore, there is something significantly different. Extensive analysis of the GAMG OpenFOAM code shows that it uses a form of residual scaling. Though in contrast to (Liu & Zeng, 2010) and (Zhang, 1997) where a fixed scaling factor is used, OpenFOAM calculates one at each multigrid iteration. Using the notation described earlier in this chapter the scaling factor is calculated as below:

$$sf = \frac{|R| \cdot |d|}{|Ad| \cdot |d|}$$

This scaling factor is then effectively used as a damping factor in a Jacobi iteration to get a new vector, X. Progressing from coarse to fine grids this new value of X is added to the old to get a new starting point for relaxation. Indeed, switching off this scaling confirms it is the factor having a significant impact on the convergence speed. To this end when the scaling factor is switched off and CPU and GPU speeds are compared an overall speedup of 1.47 is found. When the residual scaling method is added to the GPU code an overall speed up of 1.26 is found compared to its equivalent CPU counterpart. Showing improvements over the original value obtained.

In further detail the solve time can be broken down into the following major parts; matrix conversion, memory copy time, solver setup time and solve time. These times are compared below in Table 12 on the final time step of a 40 step fixed time stepping method discussed previously.

Corrector	Matrix conversion	Memory copy	Setup phase	Solve phase	Number of iterations
1st	41.96	16.64	14.37	27.03	8
2nd	44.57	26.89	14.76	13.77	5
3rd	28.13	11.15	9.48	51.24	13

Table 12: Percentage of solve time in one time step of significant solver sections

As would be expected with a higher number of iterations the significance of the solve phase increases. However, despite this in all cases there is a significant amount of time spent in code sections that are not required in the CPU implementation. Specifically, the matrix conversion and memory copy sections, indeed in the corrector with the lowest iteration count this is over 70% of solver time.

The method of memory copying is to use page-locked memory, (Sanders & Kandrot, 2010) states that using page-locked memory can result in a halving of copy time. Therefore, to this point CUDA code has been written with this in mind but in order to perform this function other routines must be performed. These are allocating the page-locked memory and copying pageable memory to the page-locked location. So, the memory copy can be split further into these components.

Allocate page-locked memory	Copy pageable to page-locked	Copy page-locked to GPU global memory
48.02	39.82	12.16

Table 13: Percentages of time spent in memory copy phase

Table 13 shows that the time required for the memory copy setup operations actually represent most of the memory copy time, around 88%, the actual copy to GPU is only a small portion. So even if this small portion is doubled in size a significant saving could still be made. A test was conducted running the final time step 50 times to obtain an average of the saving made. These savings equated to a 56.8% saving in the memory copy phase.

8.7 Chapter Summary

A new GPU accelerated multigrid method has been presented. It has been developed using methods that represent a compromise between methods that are used by OpenFOAM as well those that are efficiently parallelised. Validation tests showed that the new solver was slightly less numerically stable. Despite this it produced results that were accurate to hundredths of a percent after 40 time steps. A course grid correction method was compared to its CPU counterpart showing an overall speed up of 3.5 times. Subsequently the cause of the speed improvement in higher cell counts was found to be the reduced importance of idling time after CUBLAS vector operations. Subsequent to this several aspects of a multi-level solver were investigated. These showed that the residual scaling used by OpenFOAMs GAMG method provides significant reductions in compute time.

Chapter 9 Conclusion and further work

9.1 Conclusion

Primary atomization of fuel spray is a complex problem that is most commonly investigated numerically. However, this numerical investigation is generally very computationally expensive.

In this work the open source code OpenFOAM has first been assessed on elementary numerical problems to understand how it performs in simulating key physical phenomena. Cases such as the Zalesak disk (Zalesak, 1979), Rayleigh-Taylor instability and rising bubble (Hysing, et al., 2009) have shown that OpenFOAM is a viable candidate for use in primary atomization investigations. In addition, the key conditions that effect its computational efficiency have been investigated.

In this work OpenFOAM was further validated against a low Reynolds number experimental test case (Longmire, et al., 2001). This showed reliable performance in flow fields dominated by surface tension. Indeed, despite being a general purpose CFD library it is comparable to other purpose built codes. In addition to this, using the NVidia developed API CUDA GPU, accelerated linear system solvers were further developed from previous research resulting in a doubling of speed over the CPU implementation.

Next a high Reynolds number sheet atomization case was replicated on the available hardware showing comparable results.

Finally, a GPU accelerated multigrid solver was developed to accelerate this high Reynolds number primary atomization case. The resulting solver represents the first usage of a GPU accelerated multigrid method in 3D VOF described primary atomization. Using a multigrid method similar to that used by OpenFOAM showed a speed improvement of 1.26 times. This is of note as the comparison is made to highly CPU optimised linear system solver.

9.2 Further work

The main aim of further work that could be undertaken in accelerating OpenFOAM for usage in primary atomization should be closer integration between the GPU and its CPU counterpart. A logical next step from the present work would be to port most of the PISO algorithm to GPU operation. This would include the momentum predictor and pressure correctors. This would involve porting the governing equation discretisation to the GPU in addition to the linear system solvers presented in this investigation. This would have the benefit of further reducing the expensive memory transfer operations involved in the present study. Indeed, with three pressure corrections and three velocity solutions there are six transfers to and from the GPU in each time step which could be reduced to one. Coupled with this a move away from OpenFOAMs LDU matrix format could be made, the conversion process of which has a significant cost in the present study. While each of these points would likely only represent a small speed improvement together they could amount to a significant speed increase.

Further investigation into multigrid methods presented in this study could also be conducted. The selection criteria for the methods used in this study were a compromise between similarity to those used in OpenFOAM and those that are efficiently parallelised. However, if the similarity to OpenFOAM is removed then there are numerous other examples of multigrid components found in the literature that could be investigated each with advantages and disadvantages. This could correspond to more efficient compute performance.

This investigation was mainly limited to using a single consumer grade GPU. Therefore, naturally interesting avenues of further work would be to further develop the source code produced here for usage on several GPUs. Some of the ground work has been laid with periodic boundary usage but efficient communication of processor boundary coefficients would still need to be found. This would allow problems with far greater cell counts to be investigated.

Finally, this work has not used turbulence modelling, instead working in the DNS regime to account for the turbulence found in primary atomization. However, this is only one method and there are numerous examples of using LES in this regard, found in the literature. As there is significant ongoing work to develop suitable LES methods to reduce the time to solve primary atomization problems it would be interesting to establish how a GPU may benefit this approach.

Publications

Some of the contents of this thesis have previously appeared in the following:

Dyson, J., Xia, J., Shinjo, J., Zhao, H., “GPU Accelerated Droplet Dynamics Simulation Using OpenFOAM”, ILASS Europe, 27th Annual Conference on Liquid Atomization and Spray Systems, Brighton, UK, September 2016.

References

Abbas-Turki, L., Vialle, S., Lapeyre, B. & Mercier, P., 2014. Pricing derivatives on graphics processing units using Monte Carlo simulation. *Concurrency and Computation-Practice & Experience,* 26(9), pp. 1679-1697.

Amritkar, A. & Tafti, D., 2016. Computational Fluid Dynamics Computations Using a Preconditioned Krylov Solver on Graphical Processing Units. *Journal of Fluids Engineering,* 138(1).

Anderson, J., 1995. *Computational Fluid Dynamics: The basics with applications.* Singapore: McGraw-Hill.

Anderson, J., Lorenz, C. & Travesset, A., 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics,* 227(10), pp. 5342-5359.

Appleyard, J. & Drikakis, D., 2011. *Higher-order CFD and interface tracking methods on highly-Parallel MPI and GPU systems.* Reading, Computers and Fluids, pp. 101-105.

Apte, S., Gorokhovski, M. & Moin, P., 2003. LES of atomizing spray with stochastic modeling of secondary breakup. *Interntional Journal of Multiphase Flow,* 29(9), pp. 1503-1522.

Assured Systems, 2016. *IOT growth means moore's law could soon be no more.* [Online]
Available at: http://www.assured-systems.com/news/article/moores-law--soon-to-be-no-more/
[Accessed 14 December 2016].

Bell, J. & Marcus, D., 1992. A Second-Order Projection Method for Variable Density Flows. *Journal of Computational Physics,* 101(2), pp. 334-348.

Bell, N., Dalton, S. & Olson, L. N., 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM JOURNAL ON SCIENTIFIC COMPUTING,* pp. C123-C152.

Bell, N. & Garland, M., 2009. *Implementing sparse matrix-vector multiplication on throughput-oriented processors.* Portland, ACM, p. 18.

Bianchi, G., Minelli, F., Scardovelli, R. & Zaleski, S., 2007. 3D large scale simulation of the high-speed liquid jet atomization. *SAE technical paper.*

Bianchi, G. et al., 2005. Improving the Knowledge of high-speed liquid jets atomization by using quasi-direct 3D simulation. *SAE technical paper.*

Brackbill, J., Kothe, D. & Zemach, C., 1992. A continuum method for modeling surface tension. *Journal of computational physics,* 100(2), pp. 335-354.

Buck, I. et al., 2004. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics,* 23(3), pp. 777-786.

Chang, Y., Hou, T., Merriman, B. & Osher, S., 1996. A Level Set Formulation of Eulerian Interface Capturing Methods for Incompressible Fluid Flows. *Journal of Computational Physics,* 124(2), pp. 449-464.

Chesnel, J., Reveillon, J., Ménard, T. & Demoulin, F., 2011. Large eddy simulation of liquid jet atomization. *Atomization and Sprays,* 21(9), pp. 711-736.

Clay Mathematics Institute, 2017. *Navier-Stokes Equation.* [Online]
Available at: http://www.claymath.org/millennium-problems/navier-stokes-equation
[Accessed 4 July 2017].

Courant, R., Friedrichs, K. & Lewy, H., 1967. On the partial difference equations of mathematical physics. *IBM Journal,* 11(2), pp. 215-234.

Crank, J. & Nicolson, P., 1996. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Advances in Computational Mathematics,* 6(3-4), pp. 207-226.

De Villiers, E., Gosman, A. & Weller, H., 2004. Large eddy simulation of primary diesel spray atomization. *SAE technical paper.*

Deakin, T. & McIntosh-Smith, S., 2015. *GPU-STREAM benchmarking the achievable memory bandwidth of graphics processing units.* Austin, IEEE/ACM SuperComputing.

Delbosc, N., 2015. *Real-time simulation of indoor air flow using the lattice boltzmann method on graphics processing unit,* Leeds: University of Leeds.

Deng, G., Piquet, J., Queutey, P. & Visonneau, M., 2000. Navier-Stokes equations for incompressible flows: finite-difference and finite-volume methods. In: *Handbook of Computational Fluid Mechanics.* London: Academic Press, pp. 25-99.

Deshpande, S., Anumolu, L. & Trujillo, M., 2012. Evaluating the performance of the two-phase flow solver interFoam. *Computational Science & Discovery,* 5(1).

Deshpande, S., Gurjar, S. & Trujillo, M., 2015. A computational study of an atomizing liquid sheet. *Physics of Fluids,* 27(8).

Desjardins, O., McCaslin, J., Owkes, M. & Brady, P., 2013. Direct numerical and large-eddy simulation of primary atomization in complex geometries. *Atomization and Sprays,* 23(11), pp. 1001-1048.

Desjardins, O., Moureau, V. & Pitsch, H., 2008. An accurate conservative level set/ghost fluid method for simulating turbulent atomization. *Journal of Computational Physics,* 227(18), pp. 8395-8416.

Dombrowski, N. & Johns, W., 1963. The aerodynamic instability and disintegration of viscous liquid sheets. *Chemical Engineering Science,* 18(3), pp. 203-214.

Elgeti, S. & Sauerland, H., 2016. Deforming fluid domains within the finite element method: five mesh-based tracking methods in comparison. *Archives of computational methods in engineering,* 23(2), pp. 323-361.

Elsen, E., 2008. Large Calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics,* 227(24), pp. 10148-10161.

Faeth, G., 1991. Structure and atomization properties of dense turbulent sprays. *International Symposium on Combustion,* 23(1), pp. 1345-1352.

Fatone, L. et al., 2012. Parallel option pricing on GPU: barrier options and realized variance options. *Journal of Supercomputing,* 62(3), pp. 1480-1501.

Fedkiw, R., Aslam, T., Merriman, B. & Osher, S., 1999. A Non-oscillatory Eulerian Approach to Interfaces in Multimatirial Flows (the Ghost Fluid Method). *Journal of Computational Physics,* 152(2), pp. 457-492.

Fraser, R., Eisenklam, P., Dombrowski, N. & Hasson, D., 1962. Drop formation from rapidly moving liquid sheets. *AIChE Journal,* 8(5), pp. 672-680.

Friedrichs, M., Eastman, P. & Vaidyanathan, V., 2009. Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry,* 30(6), pp. 864-872.

Fuster, D., Agbaglah, G., Josserand, C. & Popinet, S., 2009. Numerical simulation of droplets, bubbles and waves: state of the art. *Fluid Dynamics Research,* 41(6), pp. 1-24.

Fuster, D. et al., 2009. Simulation of primary atomization with an octree adaptive mesh refinement and VOF method. *International Journal of Multiphase Flow,* 35(6), pp. 530-565.

Ghiji, M. et al., 2016. Numerical and experimental investigation of early stage diesel sprays. *Fuel,* Volume 175, pp. 274-286.

Giannopapa, C. G. & Papadakis , G., 2007. *Indicative results and progress on the development of the unified single solution method for fluid-structure iteration problems.* San Antonio, American Society of Mechanical Engineers, pp. 87-91.

Gonnet, P., 2013. *Parallel scaling/efficieny plots.* [Online]
Available at: https://community.dur.ac.uk/pedro.gonnet/?p=141
[Accessed 25 January 2017].

Gopala, V. & van Wachem, B., 2008. Volume of fluid methods for immiscible-fluid and free-surface flows. *Chemical Engineering Journal,* 141(13), pp. 204-221.

Gorokhovski, M. & Herrmann, M., 2008. Modeling Primary Atomization. *Annual review of fluid mechanics,* Volume 40, pp. 343-366.

Griebel, M. & Zaspel, P., 2010. A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations. *Computer Science - Research and Development,* 25(1), pp. 65-73.

Grosshans, H. et al., 2016. Sensitivity of VOF simulations of the liquid jet breakup to physical and numerical parameters. *Computers and Fluids,* Volume 136, pp. 312-323.

Hagerty, W. & Shea, J., 1955. A study of the stability of plane fluid sheets. *Journal of applied mechanics,* 22(4), pp. 509-514.

Harvie, D., Davidson, M. & Rudman, M., 2006. An analysis of parasitic current generation in volume of fluid simulations. *Applied Mathematical Modelling,* 30(10), pp. 1056-1066.

Hemida, H., 2008. *OpenFOAM tutorial: Free surface tutorial using interFoam and rasInterFoam,* Goteborg: Chalmers University of Technology.

Herrmann, M., 2008. A balanced force refined level set grid method for two-phase flows on unstructured flow solver grids. *Journal of computational physics,* 227(4), pp. 2674-2706.

Herrmann, M., 2010. A parallel Eulerian interface tracking/Lagrangian point particle multi-scale coupling procedure. *Journal of Computational Physics,* 229(3), pp. 745-759.

Hestenes, M. & Stiefel, E., 1952. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards,* 49(6), pp. 409-436.

Higuera, P., Lara, J. L. & Losada, I. J., 2013. Realistic wave generation and active wave absorption for Navier-Stokes models Application to OpenFOAM. *Coastal Engineering,* Volume 71, pp. 102-118.

Hirt, C. & Nichols, B., 1981. Volume of fluid (VOF) method for the dynamics of free boundaries. *Journal of Computational Physics,* 39(1), pp. 201-225.

Hysing, S. et al., 2009. Quantitative benchmark computations of two-dimensional bubble dynamics. *International journal for numerical methods in fluids,* 60(11), pp. 1259-1288.

Ikebata, A. & Xiao, F., 2016. GPU-accelerated large-scale simulations of interfacial multiphase fluids for real-case applications. *Computers & Fluids,* Volume 141, pp. 235-249.

Issa, R., Gosman, A. & Watkins, A., 1986. The computation of compressible and incompressible recirculating flows by a non-iterative implicit scheme. *Journal of Computational Physics,* 62(1), pp. 66-82.

Jasak, H. & Weller, H., 1995. *Interface-tracking capabilities of the InterGamma differencing scheme,* London: Imperial College.

Jiang, X., Siamas, G., Jagus, K. & Karayiannis, T., 2010. Physical modelling and advanced simulations of gas-liquid two-phase jet flows in atomization and sprays. *Progress in Energy and Combusion Science,* 36(2), pp. 131-167.

Kang, M., Fedkiw, R. & Liu, X., 2000. A boundry condition capturing method for multiphase incompressible flow. *Journal of Scientific Computing,* 15(1), pp. 323-360.

Keough, S., 2014. *Optimising the Parallelisation of OpenFOAM Simulations,* Victoria: Maritime Division, Defence and Technology Organisation.

Khajeh-Saeed, A. & Perot, J., 2013. Direct numerical simulation of turbulence using GPU accelerated supercomputers. *Journal of Computational Physics,* Volume 235, pp. 241-257.

Khronos Group, 2017. *The OpenGL Registry.* [Online]
Available at: https://khronos.org/registry/OpenGL/index_gl.php
[Accessed 14 August 2017].

Klostermann, J., Schaake, K. & Schwarze, R., 2013. Numerical simulation of a single rising bubble by VOF with surface compression. *International journal for numerical methods in fluids,* 71(8), pp. 960-982.

Lawrence Livermore National Laboratory, n.d. *Scalable linear solvers.* [Online]
Available at: https://computation.llnl.gov/casc/sc2001_fliers/SLS/SLS01.html
[Accessed 27 December 2016].

Lefebvre, A., 1989. *Atomization and Sprays.* s.l.:Hemisphere Publishing Corporation.

Liang, S., Liu, W. & Yuan, L., 2014. Solving seven-equation model for compressible two-phase flow using multiple GPUs. *Computers and Fluids,* Volume 99, pp. 156-171.

Liu, H., Yang, B. & Chen, Z., 2015. Accelerating algebraic multigrid solvers on NVIDIA GPUs. *Computers & Mathematics with Applications,* 70(5), pp. 1162-1181.

Liu, Q. & Zeng, J., 2010. Convergence analysis of multigrid methods with residual scaling techniques. *Journal of computational and applied mathematics,* 234(10), pp. 2932-2942.

Liu, X., Fedkiw, R. & Kang, M., 2000. A boundry condition capturing method for poisson's equation on irregular domains. *Journal of computational physics,* 160(1), pp. 151-178.

Li, X. & Tankin, R., 1991. On the temporal instability of a two-dimensional viscous liquid sheet. *Journal of Fluid Mechanics,* Volume 226, pp. 425-443.

Longmire, E., Norman, T. & Gefroh, D., 2001. Dynamics of pinch-off in liquid/liquid jets with surface tension. *International Journal of Multiphase Flow,* 27(10), pp. 1735-1752.

López, J., Hernández, J., Gómez, P. & Faura, F., 2005. An improved PLIC-VOF method for tracking thin fluid structures in incompressible two-phase flows. *Journal of Computational Physics,* 208(1), pp. 51-74.

McCool, M. & Du Toit, S., 2004. *Metaprogramming GPUs with Sh.* s.l.:AK Peters.

Menard, T., Tanguy, S. & Berlemont, A., 2007. Coupling level set/VOF/ghost fluid methods: Validation and application to 3D simulation of the primary break-up of a liquid jet. *International journal of multiphase flow,* 33(5), pp. 510-5224.

Microsoft, 2017. *Direct3D Graphics.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/windows/desktop/dn903821(v=vs.85).aspx
[Accessed 3 June 2017].

Moore, G., 1975. *Progress in Digital Integrated Electronics.* s.l., IEEE, pp. 11-13.

Navarro-Martinez, S., 2014. Large eddy simulation of spray atomization with a probability density function method. *International Journal of Multiphase Flow,* Volume 63, pp. 11-22.

NVidia Coorperation, 2012. *NVidia's next generation CUDA compute architecture: Kepler GK110,* Santa Clara: NVidia Coorperation.

Nvidia Corperation, 2017. *Cg Toolkit.* [Online]
Available at: https://developer.nvidia.com/cg-toolkit
[Accessed 3 June 2017].

Nvidia Corperation, 2017. *CUDA C Programing Guide.* [Online]
Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4iwePFLKL
[Accessed 3 June 2017].

Nvidia Corporation, 2014. *CUSP Toolkit.* [Online]
Available at: https://developer.nvidia.com/cusp
[Accessed 3 June 2017].

Ohnesorge, W., 1936. Formation of Drops by Nozzles and the Breakup of Liquid Jets. *Journal of applied mathematics and mechanics,* Volume 16, pp. 355-358.

OpenCFD Ltd, 2009. *OpenFOAM User Guide,* Bracknell: OpenCFD Ltd.

Osher, S. & Sethian, J., 1988. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics,* 79(1), pp. 12-49.

Pan, Y. & Suga, K., 2003. Capturing the pinch-off of liquid jets by the level set method. *Journal of fluids engineering,* 125(5), pp. 922-927.

Phillips, E., Davis, R. & Owens, J., 2010. *Unsteady turbulent simulations on a cluster of graphics.* San Antonio, Texas, AIAA.

Pringuey, T., 2012. *Large Eddy Simulation of Primary Liquid-Sheet Breakup,* s.l.: Cambridge University.

Rayleigh, J., 1879. On the capillary phenomena of jets. *Proceedings of the Royal Society of London,* Volume 29, pp. 71-97.

Reddy, R. & Banerjee, R., 2015. GPU accelerated VOF based multiphase flow solver and its application to sprays. *Computers and Fluids,* Volume 117, pp. 287-303.

Salvadore, F., 2013. GPU accelerated flow solver for direct numerical simulation of turbulent flows. *Journal of computational physics,* Volume 235, pp. 129-142.

Salvador, F. J., Romero, J.-V., Roselló, M.-D. & Martínez-López, J., 2010. Validation of a code for modeling cavitation phenomena in Diesel injector nozzles. *Mathematical and Computer Modelling,* 52(7-8), pp. 1123-1132.

Salvador, F., Romero, J., Roselló, M. & Jaramillo, D., 2016. Numerical simulation of primary atomization in diesel spray at low injection pressure. *Journal of Computational and Applied Mathematics,* Volume 291, pp. 94-102.

Sanders, J. & Kandrot, E., 2010. Page-Locked Host Memory. In: *CUDA by example: an intorduction to general purpose GPU prgramming.* Boston: Pearson Education, Inc, pp. 186-192.

Sander, W. & Weigand, B., 2008. Direct numerical simulation and analysis of instability enhancing parameters in liquid sheets at moderate reynolds numbers. *Physics of fluids,* 20(5).

Shinjo, J. & Umemura, A., 2010. Simulation of liquid jet primary breakup: Dynamics of ligament and droplet formation. *International Journal of Multiphase Flow,* 36(7), pp. 513-532.

Shinjo, J. & Umemura, A., 2011. Detailed simulation of primary atomization mechanisms in Diesel jet sprays (isolated identification of liquid jet tip effects). *Proceedings of the combustion institute,* Volume 33, pp. 2089-2097.

Shinjo, J. & Umemura, A., 2011. Surface instability and primary atomization characteristics of straight liquid jet sprays. *International journal of multiphase flow,* 37(10), pp. 1294-1304.

Shinn, A. & Vanka, S., 2010. *Implementation of a semi-implicit pressure-based multigrid fluid flow algorithm on a graphics processing unit.* Lake Buena Vista, ASME, pp. 125-133.

Shi, Y., Green, W., Wong, H.-W. & Oluwole, O., 2012. Accelerating multi-dimensional combustion simulations using GPU and hybrid explicit/implicit ODE integration. *Combustion and Flame,* 159(7), pp. 2388-2397.

Sirignano, W. & Mehring, C., 2000. *Review of theory of distortion and disintegration of liquid streams.* Antalya, Turkey, Combustion Institute, pp. 609-655.

Smagorinsky, J., 1963. General circulation experiments with the primitives equations. *Monthly weather review,* Volume 63, pp. 99-161.

Spalding, D., 1972. A novel finite difference formulation for differential expressions involving both first and second derivatives. *International Journal for Numerical Methods in Engineering,* 4(4), pp. 551-559.

Squire, H., 1953. Investigation of the instability of a moving liquid film. *British Journal of Applied Physics,* Volume 4, pp. 167-169.

Stone, J. et al., 2007. Accelerating molecular modeling applications wich graphics processors. *Journal of Computational Chemistry,* 28(16), pp. 2618-2640.

Sun, Y., 2016. *Yifei Sun's Research.* [Online]
Available at: http://plaza.ufl.edu/yfsun/research.html
[Accessed 24 January 2017].

Sussman, M. & Puckett, E., 2000. A Coupled level set and volume-of-fluid method for ccomputing 3D and axisymmetric incompressible two-phase flows. *Journal of computational physics,* 162(2), pp. 301-337.

Taylor, G., 1959. The dynamics of thin sheets of fluid I. Water bells. *Proceedings of the Royal Society of London, Series A,* Volume 253, pp. 289-295.

The Khronos Group, 2012. *The OpenCL Specification.* [Online]
Available at: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf
[Accessed 27 July 2017].

Thompson, C. J., Hahn, S. & Oskin, M., 2002. *Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis.* Istanbul, IEEE COMPUTER SOCIETY, pp. 306-317.

Tomczak, T., 2013. Acceleration of iterative Navier-Stokes solvers on graphics processing units. *International Journal of computational fluid dynamics,* 27(4-5), pp. 201-209.

top500, 2015. *June 2015 Green 500 list.* [Online]
Available at: https://www.top500.org/green500/lists/2015/06/
[Accessed 16 February 2017].

top500, 2015. *June 2015 top 500 list.* [Online]
Available at: https://www.top500.org/lists/2015/06/
[Accessed 16 February 2017].

Trottenberg, U., Oosterlee, C. & Schüller, A., 2001. *Multigrid.* London: Academic Press.

Tryggvason, G. & Unverdi, S., 1990. Computations of three-dimensional Rayleigh-Taylor instability. *Physics of Fluids A,* Volume 2, pp. 656-659.

Unverdi, S. & Tryggvason, G., 1992. A front-tracking method for viscous, incompressible, multi-fluid flows. *Journal of computational physics,* 100(1), pp. 25-37.

van Leer, B., 1974. Towards the ultimate conservative difference scheme. II. Monotonicity and conservation combined in a scond-order scheme. *Journal of Computational Physics,* 14(4), pp. 361-370.

Versteeg, H. & Malalasekera, W., 2007. *An introduction to computational fluid dynamics: the finite volume method.* Harlow: Pearson Education.

Versteeg, H. & Malalasekera, W., 2007. Grid generation for the multigrid mehod. In: *An Introduction to Computational Fluid Dynamics: the finite volume method.* Harlow: Pearson Prentice Hall, p. 241.

Wang, C., Yang, L., Xie, L. & Chen, P., 2015. Weakly nonlinear instability of planar viscoelastic sheets. *Physics of Fluids,* 27(1).

Wang, Y., Chatterjee, P. & De Ris, J. L., 2011. Lerge eddy simulation of fire plumes. *Proceedings of the Combustion Institute,* 33(2), pp. 2473-2480.

Xu, C. et al., 2014. Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer. *Journal of Computational Physics,* Volume 278, pp. 275-297.

Zalesak, S., 1979. Fully multidimensional flux-corrected transport algorithm for fluids. *Journal of computational physics,* 31(3), pp. 335-362.

Zaspel, P. & Griebel, M., 2013. Solving incompressible two-phase flows on multi-GPU clusters. *Computers and Fluids,* Volume 80, pp. 356-364.

Zhang, J., 1997. Residual scaling techniques in multigrid, I: Equivalence proof. *Applied mathematics and computation,* 86(2-3), pp. 283-303.

Appendix: Multigrid Solver Code

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2014 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "cuda.h"
#include "cublas_v2.h"
#include "stdio.h"
#include "math.h"
#include <cusp/array1d.h>
#include <cusp/array2d.h>
#include <cusp/multiply.h>
#include <cusp/coo_matrix.h>
#include <cusp/csr_matrix.h>
#include <cusp/format_utils.h>
#include <cusp/elementwise.h>
#include <cusp/transpose.h>
#include <cusp/print.h>
#include <iostream>
#include <limits>

#include "amgaccel.H"

using namespace std ;

    const int streams = 1;
cudaStream_t stream[streams];
const int xcellwidth = 40; //cells in one row between periodic BCs
const int nboundcells = 86400; //cells in one periodic BC
const int BLOCK_SIZE = 256;
int debug = 0;


//
// CUDA API error checking function
//
static inline void
cudaCall(
        cudaError    err,
        const char * msg
```

```
            )
{
   if( cudaSuccess != err)
   {
      cerr << msg << " : "                    ;
      cerr << cudaGetErrorString( err ) ;
      cerr << "\n"                             ;
      exit(-1)                                ;
   } ;
} ;

//
// GPU kernel to multiply sparse matrix in CSR format by dense vector
//
static __global__ void
KERNEL_crs_multiply(
            int         offset,
            int         num_rows,
            const double * Ax,
            const int     * Aj,
            const int     * Ap,
            const double * x,
            double * y,
            int level
            )
{

const int THREADS_PER_VECTOR = 4;
const int VECTORS_PER_BLOCK = BLOCK_SIZE/THREADS_PER_VECTOR;
const int THREADS_PER_BLOCK = VECTORS_PER_BLOCK * THREADS_PER_VECTOR;
const int thread_id     = offset + THREADS_PER_BLOCK * blockIdx.x + threadIdx.x;

   if (thread_id >= ((num_rows*THREADS_PER_VECTOR)/level))
      return;

      __shared__ volatile double sdata[VECTORS_PER_BLOCK * THREADS_PER_VECTOR +
THREADS_PER_VECTOR / 2];
      __shared__ volatile int ptrs[VECTORS_PER_BLOCK][2];

      const int thread_lane = threadIdx.x & (THREADS_PER_VECTOR - 1);
      const int vector_id     = thread_id     /   THREADS_PER_VECTOR;
      const int vector_lane = threadIdx.x /   THREADS_PER_VECTOR;
      const int num_vectors = VECTORS_PER_BLOCK * gridDim.x;

      for(int row = vector_id; row < num_rows/level; row += num_vectors)
      {
            if(thread_lane < 2)
                  ptrs[vector_lane][thread_lane] = Ap[row*level + thread_lane];
            const int row_start = ptrs[vector_lane][0];
            const int row_end     = ptrs[vector_lane][1];

            double sum = 0;

            for(int jj = row_start + thread_lane; jj < row_end; jj += THREADS_PER_VECTOR){
                sum += Ax[jj] * x[Aj[jj]];
                }

            sdata[threadIdx.x] = sum;

            if (THREADS_PER_VECTOR > 16) sdata[offset + threadIdx.x] = sum = sum +
sdata[offset + threadIdx.x + 16];
            if (THREADS_PER_VECTOR >  8) sdata[offset + threadIdx.x] = sum = sum +
sdata[offset + threadIdx.x +  8];
```

```
        if (THREADS_PER_VECTOR >  4) sdata[offset + threadIdx.x] = sum = sum +
sdata[offset + threadIdx.x +  4];
        if (THREADS_PER_VECTOR >  2) sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x
+  2];
        if (THREADS_PER_VECTOR >  1) sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x
+  1];

        if (thread_lane == 0){
            y[row*level] = sdata[threadIdx.x];
        }
    }
};

//
// GPU kernel to multiply sparse matrix in CSR format by dense vector
//
// Where matrix is type int
//
static __global__ void
KERNEL_crs_multiplyint(
        int          num_rows,
        const int * Ax,
        const int     * Aj,
        const int     * Ap,
        const double * x,
        double * y
        )
{

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= num_rows)
        return;

    double dot = 0;

    int row_start = Ap[idx];
    int row_end = Ap[idx+1];

    for (int k = row_start; k<row_end; k++)
        dot += Ax[k] * x[Aj[k]];

    y[idx] = dot;

};

//
// GPU kernel to interpolate vector from oarse grid to fine grid
//
static __global__ void
KERNEL_crs_interpolate(
        int          num_rows,
        const int * Ax,
        const int     * Aj,
        const int     * Ap,
        const double * x,
        double * y
        )
{

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= num_rows)
```

```
        return;

    double dot = 0;

    int row_start = Ap[idx];
    int row_end = Ap[idx+1];

    if(idx == 0)
        row_start = 0;

    for (int k = row_start; k<row_end; k++)
        dot += Ax[k] * x[Aj[k]];


    y[idx] = dot;

};

//
// GPU kernel to restrict boundry interfaces
//
static __global__ void
KERNEL_boun_restrict(
                    int         num_rows,
                    int         Ydim,
                    int         Zdim,
                    const double * x,
                    double * y
                        )
{

    int id = blockIdx.x * blockDim.x + threadIdx.x;

    if (id >= num_rows)
    return;

    int j = id/(Ydim/2);
    int k = id-(j*(Ydim/2));
    int idx = j*(Ydim*2)+(k*2);

    y[id] = x[idx] + x[idx+1] + x[idx+Ydim] + x[idx+Ydim+1];

};

//
// GPU kernel to multiply sparse matrix in DIA format by dense vector
//
__global__ void
KERNEL_crs_multiply2(
        const int offset    ,
        const int n_rows ,
        const double * data ,
        const double * x ,
        double * R,
        int level )
{

    int idx = offset + blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= n_rows/level)
    return;

    R[idx*level] = data[idx*level] * x[idx*level];
```

```
}

//
// GPU kernel residual scaling
//
static  __global__  void
KERNEL_jacobiscale  (
                        int         offset,
                        int         n_rows,
                        double  *  B,
                        double  *  RX,
                        double  *  invDC,
                        double     damp,
                        double  *  X
                               )
{
    int  idx  =  offset  +  blockIdx.x  *  blockDim.x  +  threadIdx.x;

    if  (idx  >=  n_rows)    // Error
        return;

    X[idx]  =  damp  *  X[idx]  +  (B[idx]  -  damp  *  RX[idx])  *  invDC[idx];

}  ;

//
// GPU kernel to perform a jacobi soothing iteration
//
static  __global__  void
KERNEL_partjacobi  (
                        int         offset,
                        int         n_rows,
                        double  *  B,
                        double  *  RX,
                        double  *  invDC,
                        double     damp,
                        double  *  X,
                        double  *  X1
                               )
{
    int  idx  =  offset  +  blockIdx.x  *  blockDim.x  +  threadIdx.x;

    if  (idx  >=  n_rows)    // Error
        return;

    X1[idx]  =  ((B[  idx  ]  -  RX[  idx  ])  *  invDC[idx]  *  damp)  +  X[idx];

}  ;

//
// GPU kernel to apply cyclic boundry
//
static  __global__  void
KERNEL_boundry(
                const  int  n_rows,
                double  *  Q,
                double  *  P,
                double  *  convertx,
                int  cellwidth,
                int  boundcells)
{

        int  idx  =  blockIdx.x  *  blockDim.x  +  threadIdx.x;
        if (idx >= boundcells)
        return;

        Q[idx*cellwidth] -= P[(cellwidth-1)+idx*cellwidth] * convertx[idx];

}

//
// GPU kernel to apply cyclic boundry
//
static __global__ void
KERNEL_boundry2(
                const int n_rows,
                double * Q,
                double * P,
                double * convert2x,
                int cellwidth,
                int boundcells)
{

        int idx = blockIdx.x * blockDim.x + threadIdx.x;

        if (idx >= boundcells)
        return;

        Q[(cellwidth-1)+idx*cellwidth] -= P[idx*cellwidth] * convert2x[idx];

}

//
// GPU kernel for inverse of diagonal matrix
//
//    D^-1
//
static __global__ void
KERNEL_inverse (
                        int         offset,
                        int         n_rows,
                        double * valsD,
                        double * invD,
                        int level
                                )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= n_rows/level)    // Error
        return;

    invD[ idx*level ] = 1/valsD[ idx*level ];

} ;

//
// GPU kernel for generating interpolation matrix
//
static __global__ void
KERNEL_gen_interp (
                int X,
                int Y,
                int Z,
                int nnzI,
                int n_rowsI,
            int  n_colsI,
            int  *valsI,
            int  *c_idI,
            int  *r_idI
            )
{

      int  Xnew  =  X/2;
      int  Ynew  =  Y/2;
      int  Znew  =  Z/2;

   int  id  =  blockIdx.x  *  blockDim.x  +  threadIdx.x;

      int  idy  =  id/X;
      int  idz  =  id/(X*Y);
      int  idk  =  id/(X*Y*Z);
      int  posx=  (id-(idy*X))/2;
      int  posy=  (idy-(idz*Y))/2;
      int  posz=  (idz-(idk*Z))/2;

   if  (id  >=  n_rowsI)  // Error
      return;

      valsI[id]  =  1;
      r_idI[id]  =  id;
      c_idI[id]  =  posx+(posy*Xnew)+(posz*Xnew*Ynew);

}  ;

//
// GPU kernel to expand row indicies of CSR matrix to COO format
//
static  __global__  void
KERNEL_expand  (
                  int      n_rows,
                  int  *  r_idxR,
                  int  *  r_idxR2
                      )
{
   int  idx  =  blockIdx.x  *  blockDim.x  +  threadIdx.x;

   if  (idx  >=  n_rows)    // Error
      return;

      for  (int  k=r_idxR[idx];  k<r_idxR[idx+1];  k++){
            r_idxR2[k]  =  idx;
      }

}  ;

//
// GPU kernel for vector plus vector multiplied by scalar
//
static  __global__  void
KERNEL_smul_vadd  (
                  int          offset,
                  int          n_rows,
                  double  *  B,
                  double  *  RX,
                  double  *  alpha,
                  double  *  BRX,
                  int  level
                      )
```

```
{
   int  idx  =  offset  +  blockIdx.x  *  blockDim.x  +  threadIdx.x;

   if  (idx  >=  n_rows/level)     // Error
      return;

   BRX[  idx*level  ]  =  B[  idx*level  ]  +  alpha[0]  *  RX[  idx*level  ];

}  ;

//
//  GPU kernel for vector plus vector multiplied by scalar
//
static  __global__  void
KERNEL_add_to_x  (
                  int          offset,
                  int          n_rows,
                  double  *  alpha,
                  double  *  RX,
                  double  *  BRX,
                  int  level
                        )
{
   int  idx  =  offset  +  blockIdx.x  *  blockDim.x  +  threadIdx.x;

   if  (idx  >=  n_rows/level)     // Error
      return;

   BRX[  idx*level  ]  +=  alpha[0]  *  RX[  idx*level  ];

}  ;

//
//  GPU kernel for vector minus vector multiplied by scalar
//
static  __global__  void
KERNEL_add_to_xneg  (
                  int          offset,
                  int          n_rows,
                  double  *  alpha,
                  double  *  RX,
                  double  *  BRX,
                  int  level
                        )
{
   int  idx  =  offset  +  blockIdx.x  *  blockDim.x  +  threadIdx.x;

   if  (idx  >=  n_rows/level)     // Error
      return;

   BRX[  idx*level  ]  +=  -alpha[0]  *  RX[  idx*level  ];

}  ;

//
//  GPU kernel for vector addition
//
//    BRX = B + RX
//
static  __global__  void
KERNEL_add2  (
                  int          offset,
                  int          n_rows,
                  double  *  B,
                  double  *  RX,
                  double  *  BRX,
                  int  level
                        )
{
   int  idx  =  offset  +  blockIdx.x  *  blockDim.x  +  threadIdx.x;

   if  (idx  >=  n_rows/level)     // Error
      return;

   BRX[idx]  =  B[  idx  ]  +  RX[  idx  ];

} ;

//
//  GPU kernel for simple vector operation
//
//    BRX = B - RX
//
static  __global__  void
KERNEL_subtract  (
                  int          offset,
                  int          n_rows,
                  double  *  B,
                  double  *  RX,
                  double  *  BRX,
                  int  level
                        )
{
   int  idx  =  offset  +  blockIdx.x  *  blockDim.x  +  threadIdx.x;

   if  (idx  >=  n_rows/level)     // Error
      return;

   BRX[  idx*level  ]  =  B[  idx*level  ]  -  RX[  idx*level  ];

} ;

//
//  Set vector to zero
//
static  __global__  void
KERNEL_setzero  (
                  int          offset,
                  int          n_rows,
                  double  *  X
                        )
{
   int  idx  =  offset  +  blockIdx.x  *  blockDim.x  +  threadIdx.x;

   if  (idx  >=  n_rows)     // Error
      return;

   X[  idx  ]  =  0;

} ;

//
//  GPU kernel to copy type int from one location to another
//
static  __global__  void
KERNEL_inttransfer  (
                        int       size,
                        int  *  A,
                        int  *  B
                                )
{
    int  idx  =  blockIdx.x  *  blockDim.x  +  threadIdx.x;

    if  (idx  >=  size)    // Error
        return;

    B[  idx  ]  =  A  [idx];

}  ;

//
//  GPU kernel to correct CSR indexes
//
static  __global__  void
KERNEL_fix  (
                        int         nnzI,
                        int         n_rowsI,
                        int  *    r_idI,
                        int         nnzT,
                        int         n_rowsT,
                        int  *    r_idT
                                )
{
    int  idx  =  blockIdx.x  *  blockDim.x  +  threadIdx.x;

    if  (idx  >=  2)    // Error
        return;

    if  (idx  ==  0)
        r_idI[(n_rowsI)]=nnzI;

    if  (idx  ==  1)
        r_idT[(n_rowsT)]=nnzT;

}  ;

//
//  GPU kernel to divide one scalar by another
//
static  __global__  void
KERNEL_divide  (
                        double  *  r1,
                        double  *  r2,
                        double  *  Result
                                )
{

    Result[0]=  r1[0]/r2[0];

}  ;

//
//  GPU kernel to copy type double from one location to another
//
static  __global__  void
KERNEL_doubletransfer  (
                        int           size,
                        double  *  A,
                        double  *       B
                        )
{
   int  idx  =  blockIdx.x  *  blockDim.x  +  threadIdx.x;

   if  (idx  >=  size)     // Error
      return;

   B[  idx  ]  =  A  [idx];

}  ;

//
//  Wrapper for cublasDdot() function
//  Calculates the dot product
//
static  inline  double
dot_product  (
            cublasHandle_t  h,
            int  n_rows,
            const  double  *  v,
            const  double  *  w,
            int  level
                 )
{
   double  result;
      cublasDdot(h,  n_rows/level,  v,  level,  w,  level,  &result)           ;

   return  result  ;
}  ;

//
//  Wrapper for cublasDasum() function
//  Calculates the sum of the absolute values
//
static  inline  double
sum(
         cublasHandle_t  h,
         const  double  *  v,
         const  int  size,
         int  level
      )
{

   double  result;
      cublasDasum(h,  size/level,  v,  level,  &result)           ;

   return  result;
}  ;

//
//  Wrapper for residual scaling
//
static  void
jacobiscale(
                  int          n_rows  ,
                  double  *  B          ,
                  double  *  RX          ,
                  double  *  invDC        ,
                  double     damp       ,
                  double  *  X
             )
{
   int  size  =  n_rows;
   int streamsize = size / streams;
   int numBlocks = streamsize / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   for (int i = 0; i<streams; ++i){
         int offset = i * streamsize;

   KERNEL_jacobiscale<<<dimGrid,dimBlock,0,stream[i]>>>(offset, n_rows, B, RX, invDC, damp, X) ;

   }

if (debug >= 1)
   cudaCall (cudaGetLastError(), "KERNEL_jacobiscale FAILED") ;
} ;

//
// Wrapper for part of jacobi iteration
//
static void
partjacobi(
                  int        n_rows,
                  double  *  B,
                  double  *  RX,
                  double  *  invDC,
                  double     damp,
                  double  *  X,
                  double  *  X1
              )
{
   int size = n_rows;
   int streamsize = size / streams;
   int numBlocks = streamsize / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   for (int i = 0; i<streams; ++i){
         int offset = i * streamsize;

   KERNEL_partjacobi<<<dimGrid,dimBlock,0,stream[i]>>>(offset, n_rows, B, RX, invDC, damp, X,
X1) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after partjacobi");

   }

   cudaCall (cudaGetLastError(), "KERNEL_partjacobi FAILED") ;
} ;

//
// Wrapper for cyclic boundry application
//
static void
boundry(
            const int n_rows ,
            double * Q ,
            double * P ,
            double * convertx,
            int cellwidth,
            int boundcells)
{
   int size = boundcells;
   int numBlocks = size / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   KERNEL_boundry<<<numBlocks,BLOCK_SIZE>>>(n_rows, Q, P, convertx, cellwidth, boundcells) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after boundry");

   cudaCall (cudaGetLastError(), "KERNEL_boundry FAILED") ;
} ;

//
// Wrapper for cyclic boundry application 2
//
static void
boundry2(
            const int n_rows ,
            double * Q ,
            double * P ,
            double * convert2x,
            int cellwidth,
            int boundcells)
{
   int size = boundcells;
   int numBlocks = size / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   KERNEL_boundry2<<<dimGrid,dimBlock>>>(n_rows, Q, P, convert2x, cellwidth, boundcells) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after boundry2");

   cudaCall (cudaGetLastError(), "KERNEL_boundry2 FAILED") ;
} ;

//
// Wrapper to set vector equal to zero
//
static void
setzero(
            int        n_rows,
            double * X
             )
{
   int size = n_rows;
   int streamsize = size / streams;
   int numBlocks = streamsize / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   for (int i = 0; i<streams; ++i){
       int offset = i * streamsize;

   KERNEL_setzero<<<dimGrid,dimBlock,0,stream[i]>>>(offset, n_rows, X) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after setzero");

   }

   cudaCall (cudaGetLastError(), "KERNEL_setzero FAILED") ;
```

```
} ;

//
// Wrapper to generate interpolation matrix
//
static void
gen_interp(
            int X,
            int Y,
            int Z,
            int nnzI,
            int n_rowsI,
            int n_colsI,
            int *valsI,
            int *c_idI,
            int *r_idI
              )
{
   int size = n_rowsI;
   int numBlocks = size / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   KERNEL_gen_interp<<<dimGrid,dimBlock>>>(X, Y, Z, nnzI, n_rowsI, n_colsI, valsI, c_idI, r_idI) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after gen_interp");

   cudaCall (cudaGetLastError(), "KERNEL_gen_interp FAILED") ;
} ;

//
// Wrapper to correct CSR indexing
//
static void
fix(
               int      nnzI ,
               int      n_rowsI,
               int  *  r_idI ,
               int      nnzT ,
               int      n_rowsT,
               int  *  r_idT
              )
{
   int size = 2;
   int numBlocks = size / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   KERNEL_fix<<<dimGrid,dimBlock>>>(nnzI, n_rowsI, r_idI, nnzT, n_rowsT, r_idT) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after fix");

   cudaCall (cudaGetLastError(), "KERNEL_fix FAILED") ;
} ;

//
// Wrapper for inverse of diagonal matrix
//
static void
inversed(
            int          n_rows,
            double  *  valsD,
            double  *  invD,
            int  level
             )
{
   int  size  =  n_rows/level;
   int  streamsize  =  size  /  streams;
   int  numBlocks  =  streamsize  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks);
   dim3  dimBlock(BLOCK_SIZE);

   for  (int  i  =  0;  i<streams;  ++i){
         int  offset  =  i  *  streamsize;

   KERNEL_inverse<<<dimGrid,dimBlock,0,stream[i]>>>(offset,  n_rows,  valsD,  invD,  level)  ;

if  (debug  >=  1)
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  inversed");

   }

   cudaCall  (cudaGetLastError(),  "KERNEL_inverse  FAILED")  ;
}  ;

//
//  Wrapper for inverse of diagonal matrix
//
static  void
inversedc(
            int         n_rows,
            double  *  valsD,
            double  *  invD,
            int  level
             )
{
   int  size  =  n_rows/level;
   int  streamsize  =  size  /  streams;
   int  numBlocks  =  streamsize  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks);
   dim3  dimBlock(BLOCK_SIZE);

   for  (int  i  =  0;  i<streams;  ++i){
         int  offset  =  i  *  streamsize;

   KERNEL_inverse<<<dimGrid,dimBlock,0,stream[i]>>>(offset,  n_rows,  valsD,  invD,  level)  ;

if  (debug  >=  1)
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  inversed");

   }

   cudaCall  (cudaGetLastError(),  "KERNEL_inversec  FAILED")  ;
}  ;


//
//  Wrapper to interpolate coarse grid vector to fine grid
//
static  void
interpolate(
                  int         n_rows,
                  double  *  X,
                  double  *  X2,
                int  level
            )
{
   int  size  =  n_rows/level;
   int  streamsize  =  size  /  streams;
   int  numBlocks  =  streamsize  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks);
   dim3  dimBlock(BLOCK_SIZE);

   for  (int  i  =  0;  i<streams;  ++i){
        int  offset  =  i  *  streamsize;

   KERNEL_interpolate<<<dimGrid,dimBlock,0,stream[i]>>>(offset,  n_rows,  X,  X2,  level)  ;

if  (debug  >=  1)
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  interpolate");

   }

   cudaCall  (cudaGetLastError(),  "KERNEL_interpolate  FAILED")  ;
}  ;

//
//  Wrapper for transfer of array type int
//
static  void
inttransfer(
          int         dim,
          int  *      A,
          int  *      B
                 )
{
   int  size  =  dim  ;
   int  numBlocks  =  size  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks)  ;
   dim3  dimBlock(BLOCK_SIZE)  ;

   KERNEL_inttransfer  <<<dimGrid,  dimBlock>>>  (size,  A,  B)  ;

if  (debug  >=  1)
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  inttransfer");

   cudaCall  (cudaGetLastError(),  "KERNEL_inttransfer  FAILED")  ;
}  ;

//
//  Wrapper for transfer of array type double
//
static  void
doubletransfer(
          int        dim  ,
          double  *      A,
          double  *      B
                 )
{
   int  size  =  dim  ;
   int  numBlocks  =  size  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks)  ;
   dim3  dimBlock(BLOCK_SIZE)  ;

   KERNEL_doubletransfer  <<<dimGrid,  dimBlock>>>  (size,  A,  B)  ;

if  (debug  >=  1)
```

```
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after doubletransfer");

   cudaCall (cudaGetLastError(), "KERNEL_doubletransfer FAILED") ;
} ;

//
// Wrapper for vector opperation X = X1 - X2
//
static void
calcresidual(
            int        n_rows,
            double * X,
            double * X1,
            double * X2,
            int level
                    )
{
   int size = n_rows/level ;
   int streamsize = size / streams;
   int numBlocks = streamsize / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks) ;
   dim3 dimBlock(BLOCK_SIZE) ;

   for (int i = 0; i<streams; ++i){
         int offset = i * streamsize;

   KERNEL_subtract <<<dimGrid, dimBlock,0,stream[i]>>> (offset, n_rows, X, X1, X2, level) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after subtract");

   }

   cudaCall (cudaGetLastError(), "KERNEL_subtract FAILED") ;
} ;

//
// Wrapper for vector opperation BRX = B + alpha * RX
//
static void
gpuaddoffset(
            int        n_rows,
            double * B,
            double * RX,
            double * alpha,
            double * BRX,
            int level
             )
{
   int size = n_rows/level;
   int streamsize = size / streams;
   int numBlocks = streamsize / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   for (int i = 0; i<streams; ++i){
         int offset = i * streamsize;

   KERNEL_smul_vadd<<<dimGrid,dimBlock,0,stream[i]>>>(offset, n_rows, B, RX, alpha, BRX, level)
;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after smul_vadd");
   }

   cudaCall  (cudaGetLastError(),  "KERNEL_smul_vadd  FAILED")  ;
}  ;

//
//  Wrapper for vector opperation X = X + alpha * P
//
static  void
gpuaddtox(
            int          n_rows,
            double  *  alpha,
            double  *  RX,
            double  *  BRX,
            int  level
              )
{
   int  size  =  n_rows/level;
   int  streamsize  =  size  /  streams;
   int  numBlocks  =  streamsize  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks);
   dim3  dimBlock(BLOCK_SIZE);

   for  (int  i  =  0;  i<streams;  ++i){
         int  offset  =  i  *  streamsize;

   KERNEL_add_to_x<<<dimGrid,dimBlock,0,stream[i]>>>(offset,  n_rows,  alpha,  RX,  BRX,  level)  ;

if  (debug  >=  1)
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  gpuaddtox");

   }

   cudaCall  (cudaGetLastError(),  "KERNEL_add_to_x  FAILED")  ;
}  ;

//
//  Wrapper for vector opperation X = X - alpha * P
//
static  void
gpuaddtoxneg(
            int          n_rows,
            double  *  alpha,
            double  *  RX,
            double  *  BRX,
            int  level
              )
{
   int  size  =  n_rows/level;
   int  streamsize  =  size  /  streams;
   int  numBlocks  =  streamsize  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks);
   dim3  dimBlock(BLOCK_SIZE);

   for  (int  i  =  0;  i<streams;  ++i){
         int  offset  =  i  *  streamsize;

   KERNEL_add_to_xneg<<<dimGrid,dimBlock,0,stream[i]>>>(offset,  n_rows,  alpha,  RX,  BRX,  level)  ;

if  (debug  >=  1)
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  gpuaddtoxneg");
```

```
    }

    cudaCall  (cudaGetLastError(),  "KERNEL_add_to_xneg  FAILED")  ;
}  ;

//
//  Wrapper for vector opperation BRX = B + RX
//
static  void
gpuadd2(
                int           n_rows,
                double  *  B,
                double  *  RX,
                double  *  BRX,
                int  level
                  )
{
    int  size  =  n_rows/level;
    int  streamsize  =  size  /  streams;
    int  numBlocks  =  streamsize  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
    dim3  dimGrid(numBlocks);
    dim3  dimBlock(BLOCK_SIZE);

    for  (int  i  =  0;  i<streams;  ++i){
          int  offset  =  i  *  streamsize;

    KERNEL_add2<<<dimGrid,dimBlock,0,stream[i]>>>(offset,  n_rows,  B,  RX,  BRX,  level)  ;

if  (debug  >=  1)
    cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  gpuadd2");

    }

    cudaCall  (cudaGetLastError(),  "KERNEL_add2  FAILED")  ;
}  ;


//
//  Wrapper for multiplying sparse matrix by vector where matrix is type double
//
//          A * X = R
//
//  where A - matrix, X and R - vectors.
//
int
gpumultiply(
                const  int           n_rows,
                const  double  *  vals,
                const  int       *  c_idx,
                const  int       *  r_idx,
                const  double  *  X,
                double  *  R,
                int  level
                      )
{
    int  size  =  (n_rows/level)  *  4;
    int  streamsize  =  size  /  streams;// + (size % streams == 0 ? 0 : 1);
    int  numBlocks  =  streamsize  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
    dim3  dimGrid(numBlocks);
    dim3  dimBlock(BLOCK_SIZE);

    for  (int  i  =  0;  i<streams;  ++i){
          int  offset  =  i  *  streamsize;
```

```
   KERNEL_crs_multiply<<<dimGrid,dimBlock,0,stream[i]>>>  (
                                    offset,
                                    n_rows,
                                    vals,
                                    c_idx,
                                    r_idx,
                                    X,
                                    R,
                                    level
                                                           ) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after multiply");

   }

   cudaCall (cudaGetLastError(), "KERNEL_crs_multiply FAILED") ;

   return 0 ;
} ;

//
// Wrapper to restrict the boundry coefficents
//
int
gpuboundres(
            const int n_rows ,
            int Ydim,
            int  Zdim,
            const double * X,
            double * R
                 )
{
   int size = n_rows;
   int numBlocks = size / BLOCK_SIZE + (size % BLOCK_SIZE == 0 ? 0 : 1);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(BLOCK_SIZE);

   KERNEL_boun_restrict<<<dimGrid,dimBlock>>>  (
                                                           n_rows ,
                                                           Ydim     ,
                                                           Zdim    ,
                                                           X         ,
                                                           R
                                                         ) ;

if (debug >= 1)
   cudaCall (cudaThreadSynchronize(), "Thread Sync Failed after boundres");

   cudaCall (cudaGetLastError(), "KERNEL_boun_restrict FAILED") ;

   return 0 ;
} ;

//
//  Wrapper for multiplying sparse matrix by vector where matrix is type int
//
//        A * X = R
//
//  where A - matrix, X and R - vectors.
//
int
```

```
gpumultiplyint(
                const  int      n_rows,
                const  int  *  vals,
                const  int  *  c_idx,
                const  int  *  r_idx,
                const  double  *  X,
                double  *  R
                   )
{
   int  size  =  n_rows;
   int  numBlocks  =  size  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks);
   dim3  dimBlock(BLOCK_SIZE);

   KERNEL_crs_multiplyint<<<dimGrid,dimBlock>>>  (
                                                                  n_rows  ,
                                                                  vals      ,
                                                                  c_idx    ,
                                                                  r_idx    ,
                                                                  X            ,
                                                                  R
                                                               )  ;

if  (debug  >=  1)
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  multiplyint");

   cudaCall  (cudaGetLastError(),  "KERNEL_crs_multiplyint  FAILED")  ;

   return  0  ;
}  ;

//
//  Wrapper for multiplying sparse matrix by vector where matrix is type int
//
//         A * X = R
//
//  where A - matrix, X and R - vectors.
//
int
gpuinterpolate(
                const  int      n_rows,
                const  int  *  vals,
                const  int  *  c_idx,
                const  int  *  r_idx,
                const  double  *  X,
                double  *  R
                   )
{
   int  size  =  n_rows;
   int  numBlocks  =  size  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks);
   dim3  dimBlock(BLOCK_SIZE);

   KERNEL_crs_interpolate<<<dimGrid,dimBlock>>>  (
                                                                  n_rows  ,
                                                                  vals      ,
                                                                  c_idx    ,
                                                                  r_idx    ,
                                                                  X            ,
                                                                  R
                                                               )  ;

if  (debug  >=  1)
```

```
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  interpolate");

   cudaCall  (cudaGetLastError(),  "KERNEL_crs_interpolate  FAILED")  ;

   return  0  ;
}  ;

//
//  Wrapper for multiplying sparse matrix by vector where matrix is a diagonal
//
//         A * X = R
//
//  where A - matrix, X and R - vectors.
//
int
gpumultiply2(
               int          n_rows,
               const  double  *  vals,
               const  double  *  X,
               double  *  R,
               int  level
                      )
{
   int  size  =  n_rows/level;
   int  streamsize  =  size  /  streams;
   int  numBlocks  =  streamsize  /  BLOCK_SIZE  +  (size  %  BLOCK_SIZE  ==  0  ?  0  :  1);
   dim3  dimGrid(numBlocks);
   dim3  dimBlock(BLOCK_SIZE);

   for  (int  i  =  0;  i<streams;  ++i){
          int  offset  =  i  *  streamsize;

   KERNEL_crs_multiply2<<<dimGrid,dimBlock,0,stream[i]>>>  (
                                             offset  ,
                                                                      n_rows  ,
                                                                      vals     ,
                                                                      X           ,
                                                                      R           ,
                                             level
                                                                   )  ;

if  (debug  >=  1)
   cudaCall  (cudaThreadSynchronize(),  "Thread  Sync  Failed  after  multiply2");

   }

   cudaCall  (cudaGetLastError(),  "KERNEL_crs_multiply2  FAILED")  ;

   return  0  ;
}  ;

//
//  function that performs residual scaling
//
void  scale
       (
              cublasHandle_t  h,
              int  n_rows,
              double  *  pgpu_valsR,
              int  *  pgpu_c_idxR,
              int  *  pgpu_r_idxR,
              double  *  pgpu_X,
              double  *  RX,
            double  *  convert_gpu,
            double  *  convert2_gpu,
            int  cellwidth,
            int  boundcells,
            double  *  pgpu_B,
            double  *  invD,
            double  *  X1
        )
{

         gpumultiply(n_rows,  pgpu_valsR,  pgpu_c_idxR,  pgpu_r_idxR,  pgpu_X,  RX,  1);  //R * X = RX

         boundry(n_rows,  RX,  pgpu_X,  convert_gpu,  cellwidth,  boundcells);
         boundry2(n_rows,  RX,  pgpu_X,  convert2_gpu,  cellwidth,  boundcells);

        //RX = Acf

         double  ScalingFactor  =  dot_product(h,  n_rows,  pgpu_B,  pgpu_X,  1)/dot_product(h,  n_rows,
RX,  pgpu_X,  1);

        printf  ("scalingfactor:  %e  \n",ScalingFactor);

         jacobiscale(n_rows,  pgpu_B,  RX,  invD,  ScalingFactor,  pgpu_X);

};

//
//  function that performs a damped jacobi smoothing iteration
//
void  jacobiSmooth
        (
            int  n_rows,
            double  *  pgpu_valsR,
            int  *  pgpu_c_idxR,
            int  *  pgpu_r_idxR,
            double  *  pgpu_X,
            double  *  RX,
            double  *  convert_gpu,
            double  *  convert2_gpu,
            int  cellwidth,
            int  boundcells,
            double  *  pgpu_B,
            double  *  invD,
            double  damp,
            double  *  X1
        )
{
        printf  ("cell  width:  %d  \n",cellwidth);
        printf  ("boundcells:  %d  \n",boundcells);
         gpumultiply(n_rows,  pgpu_valsR,  pgpu_c_idxR,  pgpu_r_idxR,  pgpu_X,  RX,  1);  //R * X = RX

         boundry(n_rows,  RX,  pgpu_X,  convert_gpu,  cellwidth,  boundcells);
         boundry2(n_rows,  RX,  pgpu_X,  convert2_gpu,  cellwidth,  boundcells);

         partjacobi(n_rows,  pgpu_B,  RX,  invD,  damp,  pgpu_X,  X1);

         doubletransfer(n_rows,  X1,  pgpu_X);

};

//
//  Function that calculates the residual vector
//
```

```
void  calculateResidual
        (
                int  n_rows,
                double  *  pgpu_valsR,
                int  *  pgpu_c_idxR,
                int  *  pgpu_r_idxR,
                double  *  X1,
                double  *  RX1,
                double  *  convert_gpu,
                double  *  convert2_gpu,
                int  cellwidth,
                int  boundcells,
                double  *  pgpu_B,
                double  *  X2
        )
{
          gpumultiply(n_rows,  pgpu_valsR,  pgpu_c_idxR,  pgpu_r_idxR,  X1,  RX1,  1);  //R * X1 = RX1

          boundry(n_rows,  RX1,  X1,  convert_gpu,  cellwidth,  boundcells);
          boundry2(n_rows,  RX1,  X1,  convert2_gpu,  cellwidth,  boundcells);

          calcresidual(n_rows,  pgpu_B,  RX1,  X2,  1);  // B - RX1 = X2

};

//
//  Function to extract the diagonal of a CSR matrix
//
void  extractDiagonal
        (
                int  n_rows,
                int  nnzR,
                double  *  pgpu_valsR,
                int  *  pgpu_c_idxR,
                int  *  pgpu_r_idxR,
                double  *  invD
        )
{

        typedef  typename  cusp::array1d_view<  thrust::device_ptr<int>       >  DeviceIndexArrayView;
        typedef  typename  cusp::array1d_view<  thrust::device_ptr<double>  >  DeviceValueArrayView;

        typedef  cusp::csr_matrix_view<DeviceIndexArrayView,
                    DeviceIndexArrayView,
                    DeviceValueArrayView>  CSRDeviceView;

        thrust::device_ptr<int>       wrapped_device_r_idxR(pgpu_r_idxR);
        thrust::device_ptr<int>       wrapped_device_c_idxR(pgpu_c_idxR);
        thrust::device_ptr<double>  wrapped_device_valsR(pgpu_valsR);

        DeviceIndexArrayView  row_indices      (wrapped_device_r_idxR,  wrapped_device_r_idxR  +
(n_rows+1));
        DeviceIndexArrayView  column_indices(wrapped_device_c_idxR,  wrapped_device_c_idxR  +  nnzR);
        DeviceValueArrayView  values              (wrapped_device_valsR,  wrapped_device_valsR  +  nnzR);

        CSRDeviceView  A(n_rows,  n_rows,  nnzR,  row_indices,  column_indices,  values);

        cusp::array1d<double,  cusp::device_memory>  diagonal;

        cusp::extract_diagonal(A,  diagonal);

        double  *  pgpu_valsD  =  thrust::raw_pointer_cast(&diagonal[0]);
        inversed(n_rows,  pgpu_valsD,  invD,  1);
};

//
//  Function to perfom the Galkin product
//
void  Galkinproduct
        (
                int  n_rows,
                int  nnzR,
                double  *  pgpu_valsR,
                int  *  pgpu_c_idxR,
                int  *  pgpu_r_idxR,
                int  *  valsI,
                int  *  c_idI,
                int  *  r_idI,
                int  *  c_idT,
                int  *  r_idT,
                int  nnzI,
                int  n_rowsI,
                int  n_colsI,
                int  nnzT,
                int  n_rowsT,
                int  n_colsT,
                double  *  pgpu_valsC,
                int  *  pgpu_c_idC,
                int  *  pgpu_r_idC,
                int  *  nnzC,
                int  *  n_rowsC,
                double  *  invD4
        )
{

        typedef  typename  cusp::array1d_view<  thrust::device_ptr<int>      >  DeviceIndexArrayView;
        typedef  typename  cusp::array1d_view<  thrust::device_ptr<double>  >  DeviceValueArrayView;

        typedef  cusp::csr_matrix_view<DeviceIndexArrayView,
                    DeviceIndexArrayView,
                    DeviceValueArrayView>  CSRDeviceView;

        typedef  cusp::coo_matrix_view<DeviceIndexArrayView,
                    DeviceIndexArrayView,
                    DeviceIndexArrayView>  COODeviceView;

        typedef  cusp::csr_matrix_view<DeviceIndexArrayView,
                    DeviceIndexArrayView,
                    DeviceIndexArrayView>  CSRintDeviceView;

        thrust::device_ptr<int>      wrapped_device_r_idxR(pgpu_r_idxR);
        thrust::device_ptr<int>      wrapped_device_c_idxR(pgpu_c_idxR);
        thrust::device_ptr<double>  wrapped_device_valsR(pgpu_valsR);

        DeviceIndexArrayView  row_indices      (wrapped_device_r_idxR,  wrapped_device_r_idxR  +
(n_rows+1));
        DeviceIndexArrayView  column_indices(wrapped_device_c_idxR,  wrapped_device_c_idxR  +  nnzR);
        DeviceValueArrayView  values              (wrapped_device_valsR,  wrapped_device_valsR  +  nnzR);

        CSRDeviceView  A(n_rows,  n_rows,  nnzR,  row_indices,  column_indices,  values);

        thrust::device_ptr<int>      wrapped_device_r_idI(r_idI);
        thrust::device_ptr<int>      wrapped_device_c_idI(c_idI);
        thrust::device_ptr<int>      wrapped_device_valsI(valsI);
        thrust::device_ptr<int>      wrapped_device_r_idT(r_idT);
        thrust::device_ptr<int>      wrapped_device_c_idT(c_idT);

        DeviceIndexArrayView  rowT  (wrapped_device_r_idT,  wrapped_device_r_idT  +  nnzT);
        DeviceIndexArrayView  colT  (wrapped_device_c_idT,  wrapped_device_c_idT  +  nnzT);
        DeviceIndexArrayView  valI  (wrapped_device_valsI,  wrapped_device_valsI  +  nnzI);

        DeviceIndexArrayView  rowI  (wrapped_device_r_idI,  wrapped_device_r_idI  +  nnzI);
        DeviceIndexArrayView  colI  (wrapped_device_c_idI,  wrapped_device_c_idI  +  nnzI);

        COODeviceView  L(n_rowsI,  n_colsI,  nnzI,  rowI,  colI,  valI);

        CSRintDeviceView  K(n_rowsT,  n_colsT,  nnzT,  rowT,  colT,  valI);

        cusp::transpose(L,  K);

        cusp::coo_matrix<int,double,cusp::device_memory>C2;

        cusp::multiply(K,  A,  C2);

        cusp::coo_matrix<int,double,cusp::device_memory>C3;

        cusp::multiply(C2,  L,  C3);

        cusp::csr_matrix<int,double,cusp::device_memory>C;

        cusp::convert(C3,  C);

        *nnzC  =  C.num_entries;
        *n_rowsC  =  C.num_rows;

        int  nnz  =  C.num_entries;
                int  rows  =  C.num_rows;

        printf  ("nnzC:  %d\n",*nnzC);
        printf  ("n_rowsC:  %d\n",*n_rowsC);

        double  *  valsC  =  thrust::raw_pointer_cast(&C.values[0]);
        int  *  c_idC  =  thrust::raw_pointer_cast(&C.column_indices[0]);
        int  *  r_idC  =  thrust::raw_pointer_cast(&C.row_offsets[0]);

        doubletransfer(nnz,  valsC,  pgpu_valsC);
        inttransfer(nnz,  c_idC,  pgpu_c_idC);
        inttransfer(rows+1,  r_idC,  pgpu_r_idC);

        cusp::array1d<double,  cusp::device_memory>  diagonal;

        cusp::extract_diagonal(C,  diagonal);

        double  *  invD3  =  thrust::raw_pointer_cast(&diagonal[0]);

        inversedc(*n_rowsC,  invD3,  invD4,  1);

};


//
//  Function that performs the multigrid calculation
//
int  amgcompute
(
                int          n_rows,
                double            normFac,
                int               nnzR,
                const  double  *  valsR,
                const  int    *  c_idxR,
                const  int    *  r_idxR,
                int             nnzD,
                const  double  *  valsD,
                const  int  *  c_idxD,
                const  int  *  r_idxD,
                double    *  X,
                const  double  *  B,
                int        *  n_iter,
                double  *  epsilon,
                double        rTol,
                int        CP,
                double        Ires,
                double  *  convert,
                double        *  convert2
)
{

// Get starting time stamp
struct  timespec  start,  end;
clock_gettime(CLOCK_MONOTONIC_RAW,  &start);

        int  result  =  -1  ;

// Initialize CUBLAS
        cudaSetDevice(0);

        cublasStatus_t  s;
        cublasHandle_t  h;
        s  =  cublasCreate(&h);
        if  (s  !=  CUBLAS_STATUS_SUCCESS)  {
                printf  ("CUBLAS  initialization  failed\n");
                return  EXIT_FAILURE;
        }

// Define multigrid prameters
        int  maxlevels  =  3;

        int  tlevels;
        int  Tnnz=nnzR,  Tn_rows=n_rows;
        int  totalnnz=nnzR,  totalrows=(n_rows+1);
        int  ntotalrows=n_rows+1;
        int  Xdim[maxlevels],  Ydim[maxlevels],  Zdim[maxlevels];

        Xdim[0]  =  40;
        Ydim[0]  =  160;
        Zdim[0]  =  540;

        float  Xdimt,  Ydimt,  Zdimt;

        Xdimt  =  Xdim[0];
        Ydimt  =  Ydim[0];
        Zdimt  =  Zdim[0];

// Calculate the mesh dimensions at each level
        for(int  i  =  1;  i<maxlevels;  i++){
                Xdimt  =  Xdimt/2;
                Ydimt  =  Ydimt/2;
                Zdimt  =  Zdimt/2;
        if(floor(Xdimt)  ==  Xdimt  &&  floor(Ydimt)  ==  Ydimt  &&  floor(Zdimt)  ==  Zdimt){
                Tnnz  =  Tnnz/8;
                Tn_rows  =  Tn_rows/8;
```

```
            totalnnz  +=  Tnnz;
            totalrows  +=  (Tn_rows+1);
            ntotalrows  +=  (Tn_rows+1);
            printf  ("i  %d  \n",i);
    }
    }

clock_gettime(CLOCK_MONOTONIC_RAW,  &end);

double  delta_us  =  (end.tv_sec  -  start.tv_sec)  *  1000000  +  (end.tv_nsec  -  start.tv_nsec)  /  1000;

                printf  ("time  for  total  values:  %e  \n",delta_us);

// Create pointers to GPU Memory

    double*  pgpu_B          =  NULL;
    double*  pgpu_Bt          =  NULL;
    double*  pgpu_X          =  NULL;
    double*  pgpu_valsR    =  NULL;
    int*  pgpu_c_idxR      =  NULL;
    int*  pgpu_r_idxR      =  NULL;

// Create pointers to results on GPU memory

    double  *RX,  *BRX,  *X1,  *X2,  *invD,  *RX1,  residual;
    double  rho_1;          // \rho_{i-1}
    double  rho_2  =  0;    // \rho_{i-2}
    double  alpha  =  0;    // \alpha{i}
    double  beta  =  0;           // \beta_{i-1}
    int  levels  =  1;
    int  level  =  1;
    int  courselevel  =  n_rows/2;
    double  d1  =  4;
    double  d2  =  5;
    double  damp  =  d1/d2;
    double  damp2  =  1  -  damp;
    double  cgtoll;
    double  ScalingFactor  =  0.0;

    double  *  g_rho_1,  *  g_rho_2,  *  g_rho_3,  *  g_alpha,  *  g_beta;

clock_gettime(CLOCK_MONOTONIC_RAW,  &end);

delta_us  =  (end.tv_sec  -  start.tv_sec)  *  1000000  +  (end.tv_nsec  -  start.tv_nsec)  /  1000;

                printf  ("time  to  first  cuda  command:  %e  \n",delta_us);

// Create CUDA Streams

    cudaStream_t  stream[streams];

    const  double  almost_zero  =  numeric_limits<double>::min();

    for  (int  i  =  0;  i  <  streams;  ++i)
      cudaCall(cudaStreamCreate(&stream[i]),  "cudaStreamCreate  Failed");


clock_gettime(CLOCK_MONOTONIC_RAW,  &end);

delta_us  =  (end.tv_sec  -  start.tv_sec)  *  1000000  +  (end.tv_nsec  -  start.tv_nsec)  /  1000;

                printf  ("time  to  cuda  malloc:  %e  \n",delta_us);
```

```
	cudaCall(cudaMalloc((void**)(&g_rho_1), sizeof(double)), "cudaMalloc failed for g_rho_1");
	cudaCall(cudaMalloc((void**)(&g_rho_2), sizeof(double)), "cudaMalloc failed for g_rho_2");
	cudaCall(cudaMalloc((void**)(&g_rho_3), sizeof(double)), "cudaMalloc failed for g_rho_3");
	cudaCall(cudaMalloc((void**)(&g_alpha), sizeof(double)), "cudaMalloc failed for g_alpha");
	cudaCall(cudaMalloc((void**)(&g_beta), sizeof(double)), "cudaMalloc failed for g_beta");

// Create pointers to Host Pinned memory

	const double *p_B      = B;
	const double *p_X      = X;
	const double *p_valsR     = valsR;
	const int    *p_c_idxR   = c_idxR;
	const int    *p_r_idxR   = r_idxR;

// Define CPU values for debugging
/*
   double f1[n_rows];

   double f2[n_rows];

   double f3[n_rows];

   double f4[n_rows];

   double f5[n_rows];
*/

// Allocate memory on GPU for linear system A * X = B

	cudaCall(cudaMalloc((void**)(&pgpu_B), n_rows*sizeof(double)), "cudaMalloc failed for B");
	cudaCall(cudaMalloc((void**)(&pgpu_X), n_rows*sizeof(double)), "cudaMalloc failed for X");
	cudaCall(cudaMalloc((void**)(&pgpu_valsR), (totalnnz) * sizeof(double)), "cudaMalloc Failed
for valsR");
	cudaCall(cudaMalloc((void**)(&pgpu_c_idxR), (totalnnz) * sizeof(int)), "cudaMalloc Failed for
c_idxR");
	cudaCall(cudaMalloc((void**)(&pgpu_r_idxR), (totalrows) * sizeof(int)), "cudaMalloc Failed for
r_idxR");

// Allocate GPU memory for inverse of the diagonal

	cudaCall(cudaMalloc((void**)(&invD), totalrows*sizeof(double)), "cudaMalloc failed for invD");

clock_gettime(CLOCK_MONOTONIC_RAW, &end);

delta_us = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec - start.tv_nsec) / 1000;

			printf ("time for cuda malloc: %e \n",delta_us);

   double norm_b = normFac;
	printf ("norm B %e \n",norm_b);

clock_gettime(CLOCK_MONOTONIC_RAW, &end);

// Copy vectors X and B to GPU

	cudaCall(cudaMemcpyAsync(pgpu_B, B, n_rows * sizeof(double),  cudaMemcpyHostToDevice),
"CudaMemcpy Failed for B");
	cudaCall(cudaMemcpyAsync(pgpu_X, X, n_rows * sizeof(double),  cudaMemcpyHostToDevice),
"CudaMemcpy Failed for B");

// Copy vals for matrix R
```

```
        cudaCall(cudaMemcpyAsync (pgpu_valsR,  valsR,  nnzR * sizeof(double),
cudaMemcpyHostToDevice), "CudaMemcpyAsync failed for valsR") ;

// Copy c_idx for matrix R

        cudaCall(cudaMemcpyAsync (pgpu_c_idxR, c_idxR,  nnzR * sizeof(int),
cudaMemcpyHostToDevice), "CudaMemcpyAsync failed for c_idxR") ;

// Copy r_idx for matrix R

        cudaCall(cudaMemcpyAsync (pgpu_r_idxR, r_idxR,  (n_rows + 1) * sizeof(int),
cudaMemcpyHostToDevice), "CudaMemcpy failed for r_idxR") ;

// Create pointers for PCG vectors and boundry coefficents
        double *P, *Q, *R, *Z, *convert_gpu, *convert2_gpu;

        cudaCall(cudaMalloc((void**)(&X2), (totalrows)*sizeof(double)), "cudaMalloc failed for X2");

// Debug printouts
if (debug >= 1) {
        printf ("nnzR: %d\n",nnzR);
        printf ("nnzD: %d\n",nnzD);
        printf ("n_rows: %d\n",n_rows);
}

if (debug >= 2) {
    for (int i = 0; i<nnzR; ++i){
            printf ("vals [%d]: %e \n",i,valsR[i]);
            }

    for (int i = 0; i<(nnzR); ++i){
            printf ("vals [%d]: %e \n",i,valsR[i]);
            }

    for (int i = 0; i<(nnzR); ++i){
            printf ("c_idx [%d]: %d \n",i,c_idxR[i]);
            }

    for (int i = 0; i<(n_rows); ++i){
            printf ("r_idx [%d]: %d \n",i,r_idxR[i]);
            }

    for (int i = 0; i<(n_rows); ++i){
            printf ("X [%d]: %e \n",i,X[i]);
            }

    for (int i = 0; i<(n_rows); ++i){
            printf ("B [%d]: %.20e \n",i,B[i]);
            }

    for (int i = 0; i<(n_rows); ++i){
            printf ("valsD [%d]: %e \n",i,p_valsD[i]);
            }

    for (int i = 0; i<(nnzD); ++i){
            printf ("c_idxD [%d]: %d \n",i,c_idxD[i]);
            }

    for (int i = 0; i<(n_rows+1); ++i){
            printf ("r_idxD [%d]: %d \n",i,r_idxD[i]);
            }
}
```

```
// Multigrid Wrapper

        int  s_iter  =  1;
        int  s_iter2  =  2;
        *n_iter  =  1000;
        level  =  1;
        levels  =  1;

    double  normCL;

    if  (norm_b  <  almost_zero)
    {
        norm_b  =  1.0  ;
    }

// Setup Stage

// Generate interpolation matrix
// Define arrays of matrix size for all levels
        int  nnzI[maxlevels],  n_rowsI[maxlevels],  n_colsI[maxlevels];
        int  *valsI,  *c_idI,  *r_idI;

        nnzI[0]  =  0;

        for  (int  i=1;  i<(maxlevels);  i++){

        Xdim[i]  =  Xdim[i-1]/2;
        Ydim[i]  =  Ydim[i-1]/2;
        Zdim[i]  =  Zdim[i-1]/2;

        nnzI[i]  =  Xdim[i-1]  *  Ydim[i-1]  *  Zdim[i-1];

        n_rowsI[i]  =  Xdim[i-1]  *  Ydim[i-1]  *  Zdim[i-1];

        n_colsI[i]  =  Xdim[i]  *  Ydim[i]  *  Zdim[i];

        }

// Allocate GPU memory for interpolation matrix
        cudaCall(cudaMalloc  ((void**)(&valsI),  totalrows*sizeof(int)),  "cudaMalloc  failed  for  valsI")  ;
        cudaCall(cudaMalloc  ((void**)(&c_idI),  totalrows*sizeof(int)),  "cudaMalloc  failed  for  c_idI")  ;
        cudaCall(cudaMalloc  ((void**)(&r_idI),  totalrows*sizeof(int)),  "cudaMalloc  failed  for  r_idI")  ;

        int  nnzI_runT  =  0;
        int    n_rowsT_T=0;

// Generate the interpolation matrix for each level
        for  (int  i=1;  i<maxlevels;  i++){
            if  (i  !=  1){
            nnzI_runT  +=  nnzI[i-1];
            }
        gen_interp(Xdim[i-1],  Ydim[i-1],  Zdim[i-1],  nnzI[i],  n_rowsI[i],  n_colsI[i],  &valsI[nnzI_runT],
&c_idI[nnzI_runT],  &r_idI[nnzI_runT]);
        printf  ("nnzI_runT  %d  \n",nnzI_runT);
        }

// Create arrays for the restriction matrix
        int  nnzT[maxlevels],  n_rowsT[maxlevels],  n_colsT[maxlevels];
        int  *valsT,  *c_idT,  *r_idT;

        for  (int  i=1;  i<maxlevels;  i++){
        nnzT[i]  =  nnzI[i];
        n_rowsT[i]  =  n_colsI[i];

        n_colsT[i]  =  n_rowsI[i];
                n_rowsT_T  +=  n_rowsT[i];
        }

// Allocate GPU memory for restriction matrix
        cudaCall(cudaMalloc  ((void**)(&c_idT),  totalrows*sizeof(int)),  "cudaMalloc  failed  for  c_idI")  ;
        cudaCall(cudaMalloc  ((void**)(&r_idT),  totalrows*sizeof(int)),  "cudaMalloc  failed  for  r_idI")  ;

        int  rows[maxlevels],  nnz[maxlevels];
        rows[1]=n_rows;
        nnz[1]=nnzR;
        int  totalnnzold=0,  totalrowsold=0,  totalrows2old=0,  totalrows2=n_rows;
        totalnnz=nnzR,  totalrows=(n_rows+1);
        Tn_rows  =  n_rows,  Tnnz  =  nnzR;

// Extract the diagonal of the finest level
        extractDiagonal(n_rows,  nnzR,  pgpu_valsR,  pgpu_c_idxR,  pgpu_r_idxR,  invD);

// Perform the Galkin product at each level to generate course grids
        for(int  i  =  1;  i<(maxlevels);  i++){

        Galkinproduct(rows[i],  nnz[i],  &pgpu_valsR[totalnnzold],  &pgpu_c_idxR[totalnnzold],
&pgpu_r_idxR[totalrowsold],  &valsI[totalrows2old],  &c_idI[totalrows2old],  &r_idI[totalrows2old],
&c_idT[totalrows2old],  &r_idT[totalrows2old],  nnzI[i],  n_rowsI[i],  n_colsI[i],  nnzT[i],  n_rowsT[i],
n_colsT[i],  &pgpu_valsR[totalnnz],  &pgpu_c_idxR[totalnnz],  &pgpu_r_idxR[totalrows],  &nnz[i+1],
&rows[i+1],  &invD[totalrows2]);

                totalnnzold  =  totalnnz;
                totalrowsold  =  totalrows;
                totalrows2old  =  totalrows2;

                Tnnz  =  Tnnz/8;
                Tn_rows  =  Tn_rows/8;
                totalnnz  +=  Tnnz;
                totalrows  +=  (Tn_rows+1);
                totalrows2  +=  (Tn_rows);
        }

        int  boundCells_T=0;

        for  (int  i=1;  i<maxlevels;  i++){
                boundCells_T  +=  Ydim[i]  *  Zdim[i];
        }

// Allocate GPU memory for boundry coefficents
        cudaCall(cudaMalloc  ((void**)(&convert_gpu),  (nboundcells+boundCells_T)*sizeof(double)),
"cudaMalloc  failed  for  convert")  ;
        cudaCall(cudaMalloc  ((void**)(&convert2_gpu),  (nboundcells+boundCells_T)*sizeof(double)),
"cudaMalloc  failed  for  convert2")  ;

        cudaCall(cudaMemcpy  (convert_gpu,  convert,    nboundcells  *  sizeof(double),
cudaMemcpyHostToDevice),  "CudaMemcpy  failed  for  convert")  ;
        cudaCall(cudaMemcpy  (convert2_gpu,  convert2,    nboundcells  *  sizeof(double),
cudaMemcpyHostToDevice),  "CudaMemcpy  failed  for  convert2")  ;

// Restrict boundry coefficents for use at coarse levels
        int  boundCellsSum=0,  boundCellsSumOld=0;

                for  (int  i=1;  i<maxlevels;  i++){
                boundCellsSumOld  =  boundCellsSum;
                boundCellsSum  +=  (Ydim[i-1]  *  Zdim[i-1]);
        gpuboundres((Ydim[i])*(Zdim[i]), Ydim[i-1], Zdim[i-1], &convert_gpu[boundCellsSumOld],
&convert_gpu[boundCellsSum]);
        gpuboundres((Ydim[i])*(Zdim[i]), Ydim[i-1], Zdim[i-1], &convert2_gpu[boundCellsSumOld],
&convert2_gpu[boundCellsSum]);
        }

        nnzI_runT=0;
        int n_rowsT_runT=0;

                for (int i=1; i<maxlevels; i++){
                if (i != 1){
                nnzI_runT += nnzI[i-1];
                n_rowsT_runT += n_rowsT[i-1];
                }
        fix(nnzI[i], n_rowsI[i], &r_idI[nnzI_runT], nnzT[i], n_rowsT[i], &r_idT[n_rowsT_runT]);
        }

// Allocate GPU for intermediate vectors
        cudaCall(cudaMalloc((void**)(&R), n_rows*sizeof(double)), "cudaMalloc failed for R") ;
        cudaCall(cudaMalloc((void**)(&RX), n_rows*sizeof(double)), "cudaMalloc failed for RX");
        cudaCall(cudaMalloc((void**)(&BRX), n_rows*sizeof(double)), "cudaMalloc failed for BRX");
        cudaCall(cudaMalloc((void**)(&RX1), n_rows*sizeof(double)), "cudaMalloc failed for BR1");
        cudaCall(cudaMalloc((void**)(&X1), (totalrows)*sizeof(double)), "cudaMalloc failed for X1");
                printf ("X1 totalrows %d \n",totalrows);
        cudaCall(cudaMemcpyAsync(&X1[0], X, n_rows * sizeof(double),  cudaMemcpyHostToDevice),
"CudaMemcpy Failed for X");

        cudaCall(cudaMalloc ((void**)(&P), rows[maxlevels]*sizeof(double)), "cudaMalloc failed for P")
;
        cudaCall(cudaMalloc ((void**)(&Z), rows[maxlevels]*sizeof(double)), "cudaMalloc failed for Z")
;
        cudaCall(cudaMalloc ((void**)(&Q), rows[maxlevels]*sizeof(double)), "cudaMalloc failed for
Q") ;
        cudaCall(cudaMalloc ((void**)(&pgpu_Bt), n_rows*sizeof(double)), "cudaMalloc failed for Q") ;
        doubletransfer(n_rows, pgpu_B, pgpu_Bt);

        // show memory usage of GPU

                size_t free_byte ;

                size_t total_byte ;

                cudaMemGetInfo( &free_byte, &total_byte ) ;

                double free_db = (double)free_byte ;

                double total_db = (double)total_byte ;

                double used_db = total_db - free_db ;

                printf("GPU memory usage: used = %f, free = %f MB, total = %f MB\n",
used_db/1024.0/1024.0, free_db/1024.0/1024.0, total_db/1024.0/1024.0);

        double toll;

// Calculate the initial residual
        calculateResidual(n_rows, pgpu_valsR, pgpu_c_idxR, pgpu_r_idxR, X1, RX1, convert_gpu,
convert2_gpu, xcellwidth, nboundcells, pgpu_B, R);

// Calculate convergence criteria
        if (rTol == 0)
        toll = *epsilon;
        else
        toll  =  (sum(h,  R,  n_rows,  1)/norm_b)  *  rTol;

// Multigrid iterations
for  (int  niter  =  1;  niter  <=  *n_iter;  niter++)
{

        totalnnz=0,  totalrows=0;
        Tn_rows  =  n_rows,  Tnnz  =  nnzR;
        totalnnzold=0,  totalrowsold=0,  totalrows2old=0,  totalrows2=0;
        boundCellsSum=0,  boundCellsSumOld=0;

    for(int  i  =  1;  i<maxlevels;  i++)
    {

                boundCellsSumOld  =  boundCellsSum;
                boundCellsSum  +=  (Ydim[i-1]  *  Zdim[i-1]);
        if(i  !=  1){
                setzero(rows[i],  pgpu_X);
        }

// Jacobi Smoothing
        if(i  !=  1){
        for  (int  siter  =  1;  siter  <=  s_iter;  siter++)
        {
        jacobiSmooth(rows[i],  &pgpu_valsR[totalnnz],  &pgpu_c_idxR[totalnnz],  &pgpu_r_idxR[totalrows],
pgpu_X,  RX,  &convert_gpu[boundCellsSumOld],  &convert2_gpu[boundCellsSumOld],
xcellwidth/(pow(2,i-1)),  nboundcells/(pow(4,i-1)),  pgpu_B,  &invD[totalrows2],  damp,
&X1[totalrows2]);
        printf  ("pre  smooth  itteration:  %d  \n",siter);
        }
        }
        calculateResidual(rows[i],  &pgpu_valsR[totalnnz],  &pgpu_c_idxR[totalnnz],
&pgpu_r_idxR[totalrows],  &X1[totalrows2],  RX1,  &convert_gpu[boundCellsSumOld],
&convert2_gpu[boundCellsSumOld],  xcellwidth/(pow(2,i-1)),  nboundcells/(pow(4,i-1)),  pgpu_B,
&X2[totalrows2]);

                gpumultiplyint(rows[i+1],  &valsI[totalrows2],  &c_idT[totalrows2],  &r_idT[totalrows2],
&X2[totalrows2],  pgpu_B);

                totalnnz  +=  Tnnz;
                totalrows  +=  (Tn_rows+1);
                totalrows2  +=  (Tn_rows);
                Tnnz  =  Tnnz/8;
                Tn_rows  =  Tn_rows/8;

    }  // end of pre-smoothing

        setzero(rows[maxlevels],  &X1[totalrows2]);

// Calculate initial residual at coarsest level
            gpumultiply(rows[maxlevels],  &pgpu_valsR[totalnnz],  &pgpu_c_idxR[totalnnz],
&pgpu_r_idxR[totalrows],  &X1[totalrows2],  RX1,  1);

            boundry(rows[maxlevels],  RX1,  &X1[totalrows2],  &convert_gpu[boundCellsSum],
xcellwidth/(pow(2,maxlevels-1)),  nboundcells/(pow(4,maxlevels-1)));
            boundry2(rows[maxlevels],  RX1,  &X1[totalrows2],  &convert2_gpu[boundCellsSum],
xcellwidth/(pow(2,maxlevels-1)),  nboundcells/(pow(4,maxlevels-1)));

        setzero(rows[maxlevels],  &X2[totalrows2]);
                calcresidual(rows[maxlevels],  pgpu_B,  RX1,  &X2[totalrows2],  1);  // B - RX1 = X2

            normCL  =  sum(h,  pgpu_B,  rows[maxlevels],  1);
            residual  =  sum(h,  &X2[totalrows2],  rows[maxlevels],  1)  /  normCL;

        if(rTol  ==  0)
        cgtoll  =  *epsilon;
        else
        cgtoll  =  rTol  *  residual;

// Solve coasrest level using PCG
        for  (int  iter  =  1  ;  iter  <=  1000  ;  iter++)
        {

        cublasSetPointerMode(h,  CUBLAS_POINTER_MODE_DEVICE);

            gpumultiply2(rows[maxlevels],  &invD[totalrows2],  &X2[totalrows2],  Z,  1);  //1/D * X2 = Z

            cublasDdot(h,  rows[maxlevels],  &X2[totalrows2],  1,  Z,  1,  g_rho_1);

            if  (1  ==  iter)  {
                // p^1 = z^0;  Barlett: line 6
                doubletransfer(rows[maxlevels],  Z,  P);
            }  else  {
        divide(g_rho_1,  g_rho_2,  g_beta);
                gpuaddoffset(rows[maxlevels],  Z,  P,  g_beta,  P,  1)  ;                        // P = Z + beta * P
            }  ;
        setzero(rows[maxlevels],  Q);
            gpumultiply(rows[maxlevels],  &pgpu_valsR[totalnnz],  &pgpu_c_idxR[totalnnz],
&pgpu_r_idxR[totalrows],  P,  Q,  1);

            boundry(rows[maxlevels],  Q,  P,  &convert_gpu[boundCellsSum],  xcellwidth/(pow(2,maxlevels-
1)),  nboundcells/(pow(4,maxlevels-1)));
            boundry2(rows[maxlevels],  Q,  P,  &convert2_gpu[boundCellsSum],
xcellwidth/(pow(2,maxlevels-1)),  nboundcells/(pow(4,maxlevels-1)));

            cublasDdot(h,  rows[maxlevels],  P,  1,  Q,  1,  g_rho_3);

        divide(g_rho_1,  g_rho_3,  g_alpha);

            gpuaddtox(rows[maxlevels],    g_alpha,  P,  &X1[totalrows2],  1)  ;                  // X^i = X^{i-1} +
alpha_i * p^i
            gpuaddtoxneg(rows[maxlevels],  g_alpha,  Q,  &X2[totalrows2],  1)  ;

        doubletransfer(1,  g_rho_1,  g_rho_2);

        cublasSetPointerMode(h,  CUBLAS_POINTER_MODE_HOST);

            residual  =  sum(h,  &X2[totalrows2],  rows[maxlevels],  1)  /  normCL;

            if  (residual  <  (cgtoll))  // iteration succeeded
            {
        printf  ("CG  final  residual  %e  \n",residual);
        printf  ("CG  Itterations  %d  \n",iter);
        break  ;
            }  ;
        }  ;

// Post Smoothing iterations

    for  (int  i  =  (maxlevels-1);  i>=1;  i--)
    {
                boundCellsSumOld  =  boundCellsSum;
                boundCellsSum  -=  (Ydim[i-1]  *  Zdim[i-1]);

                Tnnz  =  Tnnz*8;
                Tn_rows = Tn_rows*8;
                totalnnz -= Tnnz;
                totalrows -= (Tn_rows+1);
                totalrows2 -= (Tn_rows);

        gpuinterpolate(rows[i], &valsI[totalrows2], &c_idI[totalrows2], &r_idI[totalrows2],
&X1[totalrows2+Tn_rows], pgpu_X); // Xh <- X2h

        doubletransfer(rows[i], &X2[totalrows2], pgpu_B);

// Perform residual scaling
        if(i != (maxlevels-1) || i == 1){
          gpumultiply(rows[i], &pgpu_valsR[totalnnz], &pgpu_c_idxR[totalnnz], &pgpu_r_idxR[totalrows],
pgpu_X, RX, 1); //R * X = RX

          boundry(rows[i], RX, pgpu_X, &convert_gpu[boundCellsSum], xcellwidth/(pow(2,i-1)),
nboundcells/(pow(4,i-1)));
          boundry2(rows[i], RX, pgpu_X, &convert2_gpu[boundCellsSum], xcellwidth/(pow(2,i-1)),
nboundcells/(pow(4,i-1)));

          ScalingFactor = dot_product(h, rows[i], pgpu_B, pgpu_X, 1)/dot_product(h, rows[i], RX,
pgpu_X, 1);

          jacobiscale(rows[i], pgpu_B, RX, &invD[totalrows2], ScalingFactor, pgpu_X);

        }

        gpuadd2(rows[i], &X1[totalrows2], pgpu_X, pgpu_X, 1); //Xh + X2H = X

        if (i == 1)
        doubletransfer(rows[i], pgpu_Bt, pgpu_B);

        for (int siter2 = 1; siter2 <= s_iter2; siter2++)
        {

        jacobiSmooth(rows[i], &pgpu_valsR[totalnnz], &pgpu_c_idxR[totalnnz], &pgpu_r_idxR[totalrows],
pgpu_X, RX, &convert_gpu[boundCellsSum], &convert2_gpu[boundCellsSum], xcellwidth/(pow(2,i-1)),
nboundcells/(pow(4,i-1)), pgpu_B, &invD[totalrows2], damp, &X1[totalrows2]);

        }
// End of Post-smoothing
    }

// Calculate final residual
        calculateResidual(n_rows, pgpu_valsR, pgpu_c_idxR, pgpu_r_idxR, X1, RX1, convert_gpu,
convert2_gpu, xcellwidth, nboundcells, pgpu_B, X2);

        level = 1;
        levels = 1;

        residual = sum(h, X2, n_rows, 1)/norm_b;
        printf ("residual: %e \n",residual);

// Break iteration loop if convergence criteria is met
                if (residual < toll)
                {
                        *epsilon = residual ;
                        *n_iter = niter;
                        result = 0;
                break;
                }
}
// End of Multigrid iterations
// Copy result back to CPU
        cudaCall(cudaMemcpy  ((void**)X,  pgpu_X,  n_rows  *  sizeof(double),
cudaMemcpyDeviceToHost),  "CudaMemcpy  of  X  back  to  host  failed");

// Destroy cuda streams

        for  (int  i  =  0;  i  <  streams;  ++i)
                cudaStreamDestroy(stream[i]);

// Free GPU memory for intermediate vectors and boundry coefficents
        cudaCall(cudaFree  (P),  "cudaFree  failed  for  P");
        cudaCall(cudaFree  (R),  "cudaFree  failed  for  R");
        cudaCall(cudaFree  (Q),  "cudaFree  failed  for  Q");
        cudaCall(cudaFree  (Z),  "cudaFree  failed  for  Z");
        cudaCall(cudaFree  (convert_gpu),  "cudaFree  failed  for  convert_gpu");
        cudaCall(cudaFree  (convert2_gpu),  "cudaFree  failed  for  convert2_gpu");

// Free GPU inputs

        cudaCall(cudaFree  (pgpu_B),  "cudaFree  failed  for  pgpu_B");
        cudaCall(cudaFree  (pgpu_Bt),  "cudaFree  failed  for  pgpu_B");
        cudaCall(cudaFree  (pgpu_X),  "cudaFree  failed  for  pgpu_X");
        cudaCall(cudaFree  (pgpu_valsR),  "cudaFree  failed  for  pgpu_valsR");
        cudaCall(cudaFree  (pgpu_c_idxR),  "cudaFree  failed  for  pgpu_c_idxR");
        cudaCall(cudaFree  (pgpu_r_idxR),  "cudaFree  failed  for  pgpu_c_idxR");

// Free interpolation and restriction matrixes

        cudaCall(cudaFree  (c_idI),  "cudaFree  failed  for  c_idI");
        cudaCall(cudaFree  (c_idT),  "cudaFree  failed  for  c_idT");
        cudaCall(cudaFree  (r_idI),  "cudaFree  failed  for  r_idI");
        cudaCall(cudaFree  (r_idT),  "cudaFree  failed  for  r_idT");
        cudaCall(cudaFree  (valsI),  "cudaFree  failed  for  valsI");

// Free GPU memory for intermediate steps

        cudaCall(cudaFree  (RX),  "cudaFree  failed  for  RX");
        cudaCall(cudaFree  (BRX),  "cudaFree  failed  for  BRX");
        cudaCall(cudaFree  (X1),  "cudaFree  failed  for  X1");
        cudaCall(cudaFree  (X2),  "cudaFree  failed  for  X2");
        cudaCall(cudaFree  (RX1),  "cudaFree  failed  for  RX1");
        cudaCall(cudaFree  (invD),  "cudaFree  failed  for  invD");

clock_gettime(CLOCK_MONOTONIC_RAW,  &end);

delta_us  =  (end.tv_sec  -  start.tv_sec)  *  1000000  +  (end.tv_nsec  -  start.tv_nsec)  /  1000;

                        printf  ("cuda  code:  %e  \n",delta_us);

        return  result;  // Return value for error checking in OpenFOAM code

}


// ************************************************************************* //
```