Massively Parallel Solution of the Shallow Water Equations on GPU Clusters

William R. Saunders*
Department of Mathematical Sciences, University of Bath,

September 2014

* e-mail: wrs20@bath.ac.uk, wrs@wrsaunders.co.uk

Contents

Abstract

The aim of the project was to implement an efficient solver for the Shallow Water Equations by investigating both explicit and implicit time stepping schemes. The main result consists of a multiple Graphics Processing Unit (GPU) implementation complete with two explicit solvers and one semi-implicit solver, all of which scale well to multiple GPUs; more specifically, decent strong scaling and near optimal weak scaling. Testing indicated that device memory bandwidth is effectively used by all major CUDA kernels with a peak useful bandwidth of 107.39 GB/s, corresponding to 60.7 % of device maximum memory bandwidth on a NVIDIA Tesla M2090. Efficient halo communication between GPUs is achieved using the Generic Communication Library (GCL) with a recorded peak global transfer rate of 2.50 GB/s.

Acknowledgements

I would like to thank my supervisors Robert Scheichl and Eike Müller alongside Thomas Melvin and Nigel Wood from the UK Met Office for their support throughout the project.

Furthermore, I would like to acknowledge that the work presented here made use of the Emerald High Performance Computing facility made available by the Centre for Innovation. The Centre is formed by the universities of Oxford, Southampton, Bristol, and University College London in partnership with the STFC Rutherford-Appleton Laboratory

1 Introduction

1.1 Problem background

Modern Numerical Weather Prediction (NWP) requires the fast solution of the equations of fluid dynamics which describe weather systems in the atmosphere. Hence it is desirable to investigate numerical schemes which are capable of producing suitably accurate results within time frames small enough for the result to be useful under operational constraints. The Shallow Water Equations (SWE) provide a model system to assess the capabilities of both a numerical time stepping scheme and its implementation as a tool for the computation of the dynamics in a weather system.

The SWE describe the flow of an incompressible fluid as a free surface above a ground plain, for example a ripple on the surface of a pond. This report applies the SWE to model the flow of a pressure field coupled with a velocity field in a simple square domain, with the main investigation focused upon the performance of three time stepping schemes implemented upon Graphics Processing Units (GPUs). Two of these schemes are explicit with the third scheme being semi-implicit. Further analysis compared the higher order explicit scheme with the semi-implicit scheme in both speed and accuracy.

Recent work within the NERC "GungHo" [1] project has demonstrated that the elliptic solve of the pressure correction step, a component of the Semi-Implicit method, will scale to several thousand CPU cores [2] and to 16384 GPUs [3]. However due to considerable differences in environment between CPUs and GPUs, production of a GPU implementation that scales to many GPUs that is efficient requires particular consideration applied to the methods used. A suitable method is required to efficiently handle the movement of data between hosts and devices, whilst effectively leveraging the GPU many core architecture and large device memory bandwidth. Theoretically in a chip to chip comparison a GPU implementation should vastly out perform a CPU implementation

with a much higher performance per watt, but how effectively an implementation scales to multiple GPUs is of particular interest.

1.2 Graphics Processing Units (GPUs)

1.2.1 Introduction to GPUs

Originally, dedicated GPUs were designed as accelerators solely for graphics processing allowing developers to offload frequent but intensive tasks onto a GPU for a performance improvement. As the requirements for graphics development changed, GPU architectures evolved to contain huge (500-3000) numbers of generic cores per device, such that the operation performed by these cores could change on a per application basis. From a graphics processing perspective this is referred to as a Unified Shading Architecture. Terminology used to describe these cores varies between manufacturers. As this study is heavily focused towards hardware devices from NVIDIA [4] they will be referred to as CUDA cores [5]. CUDA cores are combined into groups of 32 cores, along with an amount of L1 memory cache to form a Streaming Multiprocessor (SM).

Alongside the developments in hardware, complementary developments in application program interfaces (APIs) specifically CUDA and OpenCL [6] were produced. By programming CUDA cores for generic massively parallel operations via these APIs, GPUs have become accelerator cards for many tasks other than graphics processing. Modern GPUs have evolved to cater for this secondary use with features not typically required in graphics such as double precision compute capability and error correcting memory.

The implementation discussed in this report was developed to run upon NVIDIA Tesla M2090 [7] GPUs, each of these devices contains 512 CUDA cores. In addition to the large number of cores, a GPU operates using a vast ($\mathcal{O}(1000)$) number of threads of execution. To achieve maximum performance more threads than cores should be used. The benefit of starting multiple threads per core is that the core has the capability to quickly and efficiently switch between threads. By switching between threads, the thread scheduler aims to avoid cores waiting idle whilst data is fetched from memory. This is referred to as latency hiding. Due to the ratio between floating point operations and memory operations, all three time stepping schemes result in a computation which is limited by the data bandwidth between the memory and the processor performing the work (memory bandwidth). This is the case for both a CPU implementation, where memory bandwidths are of $\mathcal{O}(10)$ GB/s, and a GPU implementation with device memory bandwidth $\mathcal{O}(100)$ GB/s. Tesla M2090 GPUs as provided by the Emerald HPC [9] facility offer a peak memory bandwidth of 177 GB/s.

Problems limited by computation rate, rather than memory bandwidth, could also potentially benefit from a GPU accelerator. Large number of cores and threads allow GPUs to offer peak performances of $\mathcal{O}(1000)$ GFLOPS [1] whilst modern CPUs targeted at the scientific computing market only offer peak performance $\mathcal{O}(100)$ GFLOPS. The older generation Tesla M2090 claims a 666 GFLOPS peak double precision computation rate, however more recent GPUS such as the NVIDIA Tesla K40 offer a 1.43 TFLOPS peak double precision computation rate. A comparison between the peak computation rate and peak memory bandwidth demonstrates that the M2090 is capable of performing approximately thirty double precision calculations in the time taken for a single eight byte memory operation.

[1] Floating Point Operation per Second

1.2.2 CUDA [5] execution model

The following example CUDA kernel performs element wise addition between two vectors stored in device memory and writes the output to a vector also in device memory. This is a good example for a memory bound problem. Furthermore this is an example of a Single Instruction Multiple Data (SIMD) problem where the same operation, in this case addition, is independently applied element wise to the vectors.

First, `__global__` declares that the function may be called from the host and that no value shall be returned as indicated by `void`. After the definition of the function name, `abpc`, the input variables are listed. Here the kernel is to be launched with three input pointers, each pointing to the start of a double precision array located in device memory. To distinguish device pointers from host pointers a common convention is to start device pointers with `d_` to avoid confusion. Line 5 is the simplest possible example of a stencil, as it only involves the single central element, more complicated stencils may involve neighbouring elements such as the three point average in (2.12).

```
1 __global__ void abpc(double *d_a, double *d_b, double *d_c){
2
3 int ix=blockIdx.x*blockDim.x+threadIdx.x;
4
5 d_a[ix]=d_b[ix]+d_c[ix];
6 }
```

In terms of individual GPU threads the code is written to assign a thread to compute the element wise sum for a set index. Hence for arrays each containing 1024 elements the kernel is launched with a total of 1024 threads. Thread zero will read the first element from arrays `d_b` and `d_c` compute the sum and write the result to the first element of array `d_a`. Thread one will perform the same operation for the second elements and so on.

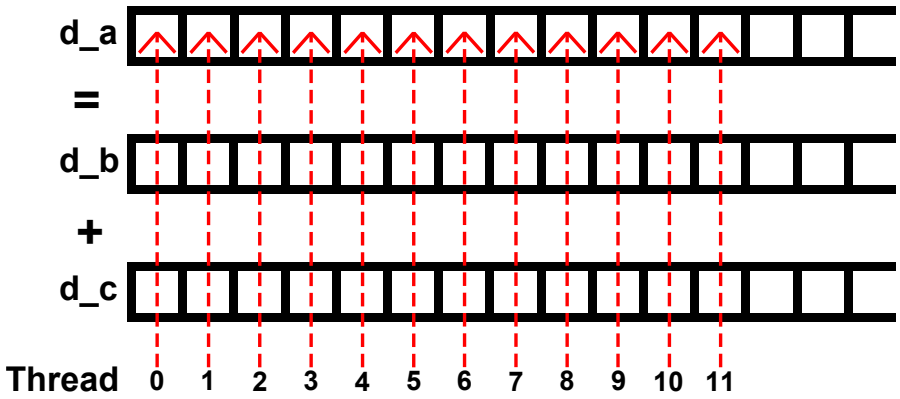

Figure 1: 12 GPU threads performing element wise addition.

Furthermore, the large pool of threads to be executed is subdivided into groups of thread blocks. Upon execution a thread block is launched by the GPU on a SM in chunks of 32 threads referred to as a warp. Line 8 in the code below launches the CUDA kernel with 1024 threads broken down into 4 thread blocks each containing 256 threads. Using this kernel launch configuration a unique index is computed for each thread by line 3 in the above code.

```
7 //To launch the kernel on a vector length 1024 with 4 blocks of 256 threads.
8 abpc<<<4,256>>>(d_a,d_b,d_c);
```

A setup procedure for this kernel would first require GPU device memory to be allocated for all three storage arrays, subsequently initial values for arrays `d_b` and `d_c` are copied from host memory into the allocated space. After the kernel has completed the computation, the values can either be copied back to host memory or retained in GPU device memory for later use. It should be noted that kernel launches do not block the host code until completion, after kernel launch the host continues execution of CPU code in parallel with GPU processing. This allows for operations to both occur on the host and between the host and device during kernel computation. Exploiting this feature can increase the performance of an implementation by hiding operations such as device to host memory operations behind computation.

1.3 Aims, objectives and report structure

The main aim of this report is to analyse and assess the efficiency of three multi-GPU solvers for the SWE, two explicit solvers and one semi-implicit solver. Memory bandwidth and computation rate are the primary tools used to assess implementation efficiency at the per GPU level, with scaling tests the primary indicator of multi-GPU efficiency.

Section two firstly focuses upon the spatial discretisation of the SWE on the square domain, secondly defining each of the three time discretisation methods in detail. Finally section two describes a Conjugate Gradient solver applicable to all three methods along with a MATLAB implementation to test the principle methods. Section three investigates observed rates of error convergence in space and time for the three schemes, followed by investigation into the effect of relative exit tolerances on the overall output error. Section four discusses specific elements of the CUDA implementation such as data layout and communication between GPUs. GPU memory usage and GPU computation rate are investigated in section five for all majors components. Penultimately, section six discusses weak and strong scaling. Finally section seven conducts a runtime comparison between the three schemes.

2 Model problem and method

We formulate the unforced non-linear shallow water equations without dissipation as a coupled hyperbolic system of equations involving conservation of mass and a momentum equation,

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{v}\phi) = 0 \tag{2.1}$$

$$\frac{\partial \mathbf{v}}{\partial t} - q\phi\mathbf{v}^{\perp} + \nabla\left(\phi + \frac{1}{2}|\mathbf{v}|^2\right) = 0, \tag{2.2}$$

where

$$q := \frac{\xi + f}{\phi}, \tag{2.3}$$

$$\mathbf{v}^{\perp} := (-v, u)^T, \tag{2.4}$$

$$\xi = \nabla^{\perp}\mathbf{v} := \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}, \text{ for } \mathbf{v} = (u, v)^T. \tag{2.5}$$

Here $\phi > 0$ represents the height of the pressure field above some ground plain. It is assumed that the maximum amplitude of the waves in this pressure field is small in comparison to the horizontal length scale. $\mathbf{v}$ denotes the velocity vector field. All fields are dependent on coordinate $\mathbf{x} = (x, y)$ and time t. The second term in equation (2.2) represents the advection of a potential vorticity constructed with a two dimensional curl $\xi = \nabla^{\perp}\mathbf{v}$ and a Coriolis force f. This is advected by a flux $\phi\mathbf{v}^{\perp}$ in the perpendicular direction. The existence of the seemingly cancelling ϕ in the second term in (2.2) is deliberately constructed to aid conservation of important physical quantities in the discretised system, finally the third term represents an energy gradient.

This system is solved in a two dimensional (unit) square domain with periodic boundary conditions on all four edges. Suitable initial conditions consist of a smooth non-zero pressure profile defined on the whole domain with small maximum amplitude, coupled with a smooth velocity vector field.

2.1 Discretisation

2.1.1 Grid structure

The two dimensional square domain is discretised using a uniform Arakawa C-grid [12], a type of structured (spatially regular) grid with quantities stored in specific locations as in figure 3. This is suitable as the domain is square and there are no complicated geometries within the domain which could require a more advanced grid structure. Furthermore using a structured grid removes any requirement to explicitly store the grid. The location of elements is determined entirely through Cartesian indexing. From a computation perspective, in comparison to an unstructured grid, this reduces the memory requirements of all time stepping schemes and reduces the complexity of the computation. For example, computing the pressure fluxes through all the vertical cell faces can be performed with a single stencil.

In this report "horizontal" refers to the x coordinate direction, with "vertical" referring to the y coordinate direction. Pressure values are stored in the centre of the grid cells with horizontal velocity values located on the centre of vertical cell edges and vertical velocity values stored on the centres of the horizontal cell vertices. The values of the potential vorticity q are stored at the vertices of the grid cells.

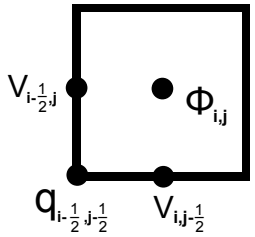

Figure 2: Location of variables in discretisation grid.

The Cartesian structured grid combined with a row major method provides a intrinsic map between the location of elements on the grid and corresponding location in memory. Whereby for each quantity stored on the grid, elements which are horizontally adjacent on the grid are adjacent in memory. This property is crucial for efficient device memory access by CUDA kernels. With data aligned in memory and threads contiguous in the x-direction, the GPU is able to coalesces

the memory access for these adjacent elements. Hence improving the memory access performance for the CUDA kernel.

2.1.2 Spatial discretisation

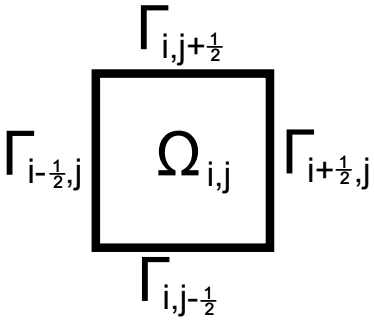

Figure 3: Cell and cell edge naming convention.

We apply the C-grid discretisation by subdividing the domain into n^2 identical square cells $\Omega_{i,j}$, $i,j = 1,\ldots,n$. The edges of each cell have length $h := \frac{1}{n}$ and are denoted $\Gamma_{i-\frac{1}{2},j}$, indexed with respect to the centre of the cell. In general the following discretisations use capitals solely for approximate quantities and lower case for continuous fields. Defining,

$$\Phi_{i,j} = \frac{1}{h^2}\int_{\Omega_{i,j}} \phi \; d\mathbf{x}, \tag{2.6}$$

$$V_{i-\frac{1}{2},j} = \frac{1}{h}\int_{\Gamma_{i-\frac{1}{2},j}} \mathbf{v}\cdot\nu \; ds, \tag{2.7}$$

$$V_{i,j-\frac{1}{2}} = \frac{1}{h}\int_{\Gamma_{i,j-\frac{1}{2}}} \mathbf{v}\cdot\nu \; ds. \tag{2.8}$$

Integrating 2.1 over cell $\Omega_{i,j}$ with side length h, with application of the divergence theorem:

$$h^2\frac{\partial \Phi_{i,j}}{\partial t} + \int_{\Omega_{i,j}} \nabla\cdot(\mathbf{v}\phi)\; d\mathbf{x} = 0, \tag{2.9}$$

$$h^2\frac{\partial \Phi_{i,j}}{\partial t} + \int_{\partial\Omega_{i,j}} (\mathbf{v}\phi)\cdot\nu \; ds = 0. \tag{2.10}$$

Similarly integrating over the cell edge $\Gamma_{i-\frac{1}{2},j}$, with $q := \frac{\nabla^\perp\cdot\mathbf{v}+f}{\phi}$, outward pointing vector ν with respect to the centre of cell $\Omega_{i,j}$.

$$h\frac{\partial (M_v V)_{i-\frac{1}{2},j}}{\partial t} + \int_{\Gamma_{i-\frac{1}{2},j}} q(\phi\mathbf{v}^\perp)\cdot\nu\, ds + \int_{\Gamma_{i-\frac{1}{2},j}} (\nu\cdot\nabla)(\phi + \frac{1}{2}|\mathbf{v}|^2)\, ds = 0, \tag{2.11}$$

M_v represents a tridiagonal velocity mass matrix, a result of approximating V by a linear

function on each cell and is defined in each coordinate direction by:

$$(M_v V)_{i-\frac{1}{2},j} := \frac{1}{6}\left(V_{i-\frac{3}{2},j} + 4V_{i-\frac{1}{2},j} + V_{i+\frac{1}{2},j}\right), \tag{2.12}$$

$$(M_v V)_{i,j-\frac{1}{2}} := \frac{1}{6}\left(V_{i,j-\frac{3}{2}} + 4V_{i,j-\frac{1}{2}} + V_{i,j+\frac{1}{2}}\right). \tag{2.13}$$

This essentially smears out the velocity field via a weighted average centred on each edge. The flux through edge $\Gamma_{i-\frac{1}{2},j}$ in (2.11) is constructed by using the velocity stored in the centre of the edge length h combined with a Φ value constructed by taking an equal weighted average of the pressure values stored either side:

$$\Phi_{i-\frac{1}{2},j} := \frac{1}{2}(\Phi_{i-1,j} + \Phi_{i,j}) \tag{2.14}$$

$$\int_{\Gamma_{i-\frac{1}{2},j}} \phi\mathbf{v}\ ds \approx h\Phi_{i-\frac{1}{2},j}V_{i-\frac{1}{2},j}. \tag{2.15}$$

Alternatives such as an upwind method, where pressure values are approximated based on the direction of flow, could be used to produce the flux on the cell edges, but such methods are beyond the scope of this study. In (2.11) we approximate the Potential Vorticity by $Q \approx q$, stored at the vertices of the grids cells by (2.16) below. As with the pressure field, a central difference between adjacent vertices evaluates Q at the centre of the cell edges.

$$Q_{i-\frac{1}{2},j-\frac{1}{2}} \approx \frac{\nabla^\perp \cdot \mathbf{v} + f}{\phi}|_{\mathbf{x}_{i-\frac{1}{2},j-\frac{1}{2}}} \tag{2.16}$$

$$Q_{i-\frac{1}{2},j} := \frac{1}{2}(Q_{i-\frac{1}{2},j-\frac{1}{2}} + Q_{i-\frac{1}{2},j+\frac{1}{2}}) \tag{2.17}$$

Here the two dimensional curl is approximated at the vertices using central differencing and ϕ is replaced by an average of the Φ values in the centres of the four cells adjacent to the vertex. f represents the Coriolis force, a constant scalar valued function on the domain.

$$\nabla^\perp \cdot \mathbf{v}(\mathbf{x}_{i-\frac{1}{2},j-\frac{1}{2}}) \approx \frac{V_{i,j-\frac{1}{2}} - V_{i-1,j-\frac{1}{2}}}{h} - \frac{V_{i-\frac{1}{2},j} - V_{i-\frac{1}{2},j-1}}{h} \tag{2.18}$$

$$\phi(\mathbf{x}_{i-\frac{1}{2},j-\frac{1}{2}}) \approx \Phi_{i-\frac{1}{2},j-\frac{1}{2}} := \frac{1}{4}\left(\Phi_{i,j} + \Phi_{i-1,j} + \Phi_{i,j-1} + \Phi_{i-1,j-1}\right) \tag{2.19}$$

Similarly the perpendicular flux at the centre of the vertical and horizontal edges is constructed as follows:

$$(\phi\mathbf{v}^\perp)\cdot\nu(\mathbf{x}_{i-\frac{1}{2},j}) \approx F_{i-\frac{1}{2},j} := -\frac{1}{4}\left(\Phi_{i-1,j-\frac{1}{2}}V_{i-1,j-\frac{1}{2}} + \Phi_{i-1,j+\frac{1}{2}}V_{i-1,j+\frac{1}{2}} + \Phi_{i,j-\frac{1}{2}}V_{i,j-\frac{1}{2}} + \Phi_{i,j+\frac{1}{2}}V_{i,j+\frac{1}{2}}\right) \tag{2.20}$$

$$(\phi\mathbf{v}^\perp)\cdot\nu(\mathbf{x}_{i,j-\frac{1}{2}}) \approx F_{i,j-\frac{1}{2}} := \frac{1}{4}\left(\Phi_{i-\frac{1}{2},j}V_{i-\frac{1}{2},j} + \Phi_{i+\frac{1}{2},j}V_{i+\frac{1}{2},j} + \Phi_{i-\frac{1}{2},j-1}V_{i-\frac{1}{2},j-1} + \Phi_{i+\frac{1}{2},j-1}V_{i+\frac{1}{2},j-1}\right) \tag{2.21}$$

The final term required for the spatial discretisation is the energy $\phi + \frac{1}{2}|\mathbf{v}|^2$, which is averaged on a cell-by-cell basis and stored at the centre of the cells. With central differencing used to approximated the energy gradient at the cell edges one obtains:

$$E_{i,j} = \Phi_{i,j} + \frac{1}{2}\left[\left(\frac{V_{i-\frac{1}{2},j} + V_{i+\frac{1}{2},j}}{2}\right)^2 + \left(\frac{V_{i,j-\frac{1}{2}} + V_{i,j+\frac{1}{2}}}{2}\right)^2\right]. \tag{2.22}$$

Using these quantities, (2.1) and (2.2) are discretised at the centre and southern and western edges of cell (i,j) by,

$$\frac{\partial \Phi_{i,j}}{\partial t} + \frac{1}{h}\left(-\Phi_{i-\frac{1}{2},j}V_{i-\frac{1}{2},j} + \Phi_{i+\frac{1}{2},j}V_{i+\frac{1}{2},j} - \Phi_{i,j-\frac{1}{2}}V_{i,j-\frac{1}{2}} + \Phi_{i,j+\frac{1}{2}}V_{i,j+\frac{1}{2}}\right) = 0 \tag{2.23}$$

$$\frac{\partial}{\partial t}\left(M_v V\right)_{i-\frac{1}{2},j} + Q_{i-\frac{1}{2},j}F_{i-\frac{1}{2},j} + \frac{E_{i,j} - E_{i-1,j}}{h} = 0 \tag{2.24}$$

$$\frac{\partial}{\partial t}\left(M_v V\right)_{i,j-\frac{1}{2}} + Q_{i,j-\frac{1}{2}}F_{i,j-\frac{1}{2}} + \frac{E_{i,j} - E_{i,j-1}}{h} = 0 \tag{2.25}$$

These are the equations solved via time stepping in the implementation.

2.1.3 Explicit temporal discretisation

This report covers three methods for integrating forward in time, two are explicit with the third a semi implicit scheme. The principle idea for the explicit schemes is to formulate (2.1) and (2.2) in standard form as,

$$\frac{\partial \boldsymbol{\Xi}}{\partial t} = \mathbf{f}(\boldsymbol{\Xi}, t),\ \boldsymbol{\Xi} := (\phi, u, v)^T. \tag{2.26}$$

Time stepping schemes may then evaluate the right hand side to determine the approximate partial derivative in time. The first scheme will take a central difference across two time steps as a discretisation for the partial time derivative, resulting in a Leapfrog method. Where the function $\mathbf{f}$ is evaluated at the intermediate stage:

$$\frac{\boldsymbol{\Xi}^{n+1} - \boldsymbol{\Xi}^{n-1}}{2\Delta t} \approx \mathbf{f}(\boldsymbol{\Xi}^n),\ \boldsymbol{\Xi}^n := (\phi(t^n), u(t^n), v(t^n))^T. \tag{2.27}$$

After a spatial discretisation as in section 2.1.2 we can define $\mathbf{F}$ explicitly in component form at each point on the grid.

$$\frac{\partial \Phi_{i,j}}{\partial t} \approx [\mathbf{F}_\Phi(\Phi, \mathbf{v})]_{i,j} := \frac{-1}{h}(-\Phi_{i-\frac{1}{2},j}V_{i-\frac{1}{2},j} + \Phi_{i+\frac{1}{2},j}V_{i+\frac{1}{2},j} - \Phi_{i,j-\frac{1}{2}}V_{i,j-\frac{1}{2}} + \Phi_{i,j+\frac{1}{2}}V_{i,j+\frac{1}{2}}) \tag{2.28}$$

$$\frac{\partial}{\partial t}(M_v V)_{i-\frac{1}{2},j} \approx [\mathbf{F}_\mathbf{V}(\Phi, \mathbf{v})]_{i-\frac{1}{2},j} := -Q_{i-\frac{1}{2},j}F_{i-\frac{1}{2},j} - \frac{E_{i,j} - E_{i-1,j}}{h} \tag{2.29}$$

$$\frac{\partial}{\partial t}(M_v V)_{i,j-\frac{1}{2}} \approx [\mathbf{F}_\mathbf{V}(\Phi, \mathbf{v})]_{i,j-\frac{1}{2}} := -Q_{i,j-\frac{1}{2}}F_{i,j-\frac{1}{2}} - \frac{E_{i,j} - E_{i,j-1}}{h}. \tag{2.30}$$

This leads to the explicit update operation for the pressure field Φ,

$$\Phi_{i,j}^{n+1} = \Phi_{i,j}^{n-1} + 2\Delta t[\mathbf{F}_\Phi(\Phi^n, \mathbf{v}^n)]_{i,j} \tag{2.31}$$

and the following linear systems involving the velocity mass matrix:

$$M_v V_{i-\frac{1}{2},j}^{n+1} = M_v V_{i-\frac{1}{2},j}^{n-1} + 2\Delta t[\mathbf{F}_\mathbf{V}(\Phi^n, \mathbf{v}^n)]_{i-\frac{1}{2},j}, \tag{2.32}$$

$$M_v V_{i,j-\frac{1}{2}}^{n+1} = M_v V_{i,j-\frac{1}{2}}^{n-1} + 2\Delta t[\mathbf{F}_\mathbf{V}(\Phi^n, \mathbf{v}^n)]_{i,j-\frac{1}{2}}. \tag{2.33}$$

Solving the two linear systems constitutes the last step of each Leapfrog iteration, i.e. solve for V' such that:

$$(M_v V')_{i-\frac{1}{2},j} = [\mathbf{F}_\mathbf{V}(\Phi^n, \mathbf{v}^n)]_{i-\frac{1}{2},j}, \tag{2.34}$$

$$(M_v V')_{i,j-\frac{1}{2}} = [\mathbf{F}_\mathbf{V}(\Phi^n, \mathbf{v}^n)]_{i,j-\frac{1}{2}} \tag{2.35}$$

and update the velocity field with the computed increments:

$$V_{i-\frac{1}{2},j}^{n+1} = V_{i-\frac{1}{2},j}^{n-1} + 2\Delta t\ V'_{i-\frac{1}{2},j}, \tag{2.36}$$

$$V_{i,j-\frac{1}{2}}^{n+1} = V_{i,j-\frac{1}{2}}^{n-1} + 2\Delta t\ V'_{i,j-\frac{1}{2}}. \tag{2.37}$$

The second method is an explicit three stage Runge Kutta method (RK3), described by the Butcher Tableau in Table 2. A multi-stage method constructs the next time step iteration by evaluating $\mathbf{f}$ successively at each stage. By involving one or more stages these methods aim to improve the accuracy and stability of the method.

0			
c_2	a_{21}		
c_3	a_{31}	a_{32}	
	b_1	b_2	b_3

Table 1: Butcher Tableau indexing.

0			
$\frac{1}{2}$	1		
1	$\frac{1}{4}$	$\frac{1}{4}$	
	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{4}{6}$

Table 2: Butcher Tableau for RK3 method.

The Butcher Tableau provides a concise arrangement to store the coefficients for a Runge Kutta scheme. The following algorithm represents an outline structure for the computation of one time step via the RK3 scheme described by table 2.

Data: $\mathbf{y}^n$, $\mathbf{y}^n := (\Phi(t^n), \mathbf{V}(t^n))$
Result: $\mathbf{y}^{n+1}$
Set $\mathbf{y} := \mathbf{y}^n$
for $l = 1 : 3$ **do**
    $\mathbf{k}_l = \Delta t \mathbf{F}(\mathbf{y})$
    $\tilde{\mathbf{k}}$:=0
    for $m = 1 : l$ **do**
        $\tilde{\mathbf{k}} = \tilde{\mathbf{k}} + \tilde{a}_{lm}\mathbf{k}_l$ given: $\tilde{a}_{lm} = \begin{cases} a_{l+1,m} & \text{if } l < 3 \\ b_m & \text{if } l = 3 \end{cases}$
    end
    $\mathbf{y} = \mathbf{y}^n + \tilde{\mathbf{k}}$
end
Set $\mathbf{y}^{n+1} := \mathbf{y}$

Algorithm 1: One step of RK3.

The outlined algorithm combined with the explicit definition of $\mathbf{F}$ gives a complete method to step forwards in time at each point in the discretisation grid. More specifically a method to compute the state vector at the next time step (Φ^{n+1}, V^{n+1}) from the state vector of the system at the current time step.

Data: (Φ^n, V^n)
Result: (Φ^{n+1}, V^{n+1})
Set $\Phi_{i,j} = \Phi^n_{i,j}$, $V_{i-\frac{1}{2},j} = V^n_{i-\frac{1}{2},j}$, $V_{i,j-\frac{1}{2}} = V^n_{i,j-\frac{1}{2}}$
for $l = 1:3$ **do**
$(k^{\phi}_l)_{i,j} = \Delta t[\mathbf{F}_{\Phi}(\Phi, \mathbf{v})]_{i,j}$
$(k^{\mathbf{v}}_l)_{i-\frac{1}{2},j} = \Delta t[\mathbf{F}_{\mathbf{V}}(\Phi, \mathbf{v})]_{i-\frac{1}{2},j}$
$(k^{\mathbf{v}}_l)_{i,j-\frac{1}{2}} = \Delta t[\mathbf{F}_{\mathbf{V}}(\Phi, \mathbf{v})]_{i,j-\frac{1}{2}}$
Set $(K^{\phi})_{i,j} = 0$, $(K^{\mathbf{v}})_{i-\frac{1}{2},j} = 0$, $(K^{\mathbf{v}})_{i,j-\frac{1}{2}} = 0$
for $m = 1:l$ **do**
$(K^{\phi})_{i,j} = (K^{\phi})_{i,j} + \tilde{a}_{l,m}(k^{\phi}_m)_{i,j}$
$(K^{\mathbf{v}})_{i-\frac{1}{2},j} = (K^{\mathbf{v}})_{i-\frac{1}{2},j} + \tilde{a}_{l,m}(k^{\mathbf{v}}_m)_{i-\frac{1}{2},j}$
$(K^{\mathbf{v}})_{i,j-\frac{1}{2}} = (K^{\mathbf{v}})_{i,j-\frac{1}{2}} + \tilde{a}_{l,m}(k^{\mathbf{v}}_m)_{i,j-\frac{1}{2}}$
end
Update $\Phi_{i,j} = \Phi^n_{i,j} + (K^{\phi})_{i,j}$
Solve two systems for the velocities:
$(M_v V')_{i-\frac{1}{2},j} = (K^{\mathbf{v}})_{i-\frac{1}{2},j}$
$(M_v V')_{i,j-\frac{1}{2}} = (K^{\mathbf{v}})_{i,j-\frac{1}{2}}$
For velocity update:
$V_{i-\frac{1}{2},j} = V^n_{i-\frac{1}{2},j} + V'_{i-\frac{1}{2},j}$
$V_{i,j-\frac{1}{2}} = V^n_{i,j-\frac{1}{2}} + V'_{i,j-\frac{1}{2}}$
end
Update $\Phi^{n+1}_{i,j} = \Phi_{i,j}$, $V^{n+1}_{i-\frac{1}{2},j} = V_{i-\frac{1}{2},j}$, $V^{n+1}_{i,j-\frac{1}{2}} = V_{i,j-\frac{1}{2}}$.

Algorithm 2: RK3 method.

As expected, comparison between the Leapfrog and RK3 schemes immediately reveals the increased cost of a multiple stage method. The RK3 scheme requires three times as many evaluations of the function $\mathbf{F}$ and three times the number of linear systems are to be solved per time step iteration. In addition to the increased computation to combine the multiple stages into the next time step iteration.

However, the RK3 method offers a smaller temporal discretisation error in comparison to Leapfrog. Hence fewer iterations would be required to compute a solution at a given time with a given accuracy. The increased cost per RK3 iteration may be offset by the decreased number of required iterations, however this investigation is outside the scope of this report.

2.1.4 Semi-implicit temporal discretisation

We use an implicit method where several quantities are time averaged on the RHS as follows:

$$\Phi_{i,j}^{n+1} = \Phi_{i,j}^{n} - \frac{1}{h}\left(-\overline{\Phi_{i-\frac{1}{2},j}V_{i-\frac{1}{2},j}}^{t} + \overline{\Phi_{i+\frac{1}{2},j}V_{i+\frac{1}{2},j}}^{t} - \overline{\Phi_{i,j-\frac{1}{2}}V_{i,j-\frac{1}{2}}}^{t} + \overline{\Phi_{i,j+\frac{1}{2}}V_{i,j+\frac{1}{2}}}^{t}\right) \tag{2.38}$$

$$M_v V_{i-\frac{1}{2},j}^{n+1} = M_v V_{i-\frac{1}{2},j}^{n} - \left(\overline{Q_{i-\frac{1}{2},j}F_{i-\frac{1}{2},j}}^{t} + \frac{\overline{E_{i,j}}^{t} - \overline{E_{i-1,j}}^{t}}{h}\right). \tag{2.39}$$

$$M_v V_{i,j-\frac{1}{2}}^{n+1} = M_v V_{i,j-\frac{1}{2}}^{n} - \left(\overline{Q_{i,j-\frac{1}{2}}F_{i,j-\frac{1}{2}}}^{t} + \frac{\overline{E_{i,j}}^{t} - \overline{E_{i,j-1}}^{t}}{h}\right). \tag{2.40}$$

With time averaging defined as,

$$\overline{X}^{t} := \int_{t_n}^{t_{n+1}} X(t)dt \approx (\alpha X^{n+1} + (1-\alpha)X^{n})\Delta t, \tag{2.41}$$

with constant $0 \leq \alpha \leq 1$, we shall take $\alpha = 0.5$, for which we expect the numerical error to be second order with respect to time step size. In a linear case this would result in the Crank Nickolson method. However in the nonlinear case, given a current state (Φ^n, V^n) an Inexact Newton method is used to iteratively compute the next state (Φ^{n+1}, V^{n+1}) as follows:

Data: (Φ^n, V^n)
Result: (Φ^{n+1}, V^{n+1})
Set $(\tilde{\Phi}, \tilde{V}) = (\Phi^n, V^n)$
for $l = 1, \ldots, l_{max}$ **do**

1. Compute an approximation to the time averaged quantities: $\overline{\Phi V}^t$, $\overline{QF}^t$ and $\overline{E}^t$ using (Φ^n, V^n), $(\tilde{\Phi}, \tilde{V})$.

2. Evaluate new residuals:

$$(R_\Phi)_{i,j} = \tilde{\Phi}_{i,j} - \Phi^n_{i,j} + \frac{1}{h}\left(-\overline{\Phi_{i-\frac{1}{2},j} V_{i-\frac{1}{2},j}}^t + \overline{\Phi_{i+\frac{1}{2},j} V_{i+\frac{1}{2},j}}^t - \overline{\Phi_{i,j-\frac{1}{2}} V_{i,j-\frac{1}{2}}}^t + \overline{\Phi_{i,j+\frac{1}{2}} V_{i,j+\frac{1}{2}}}^t\right) \tag{2.42}$$

$$(R_V)_{i-\frac{1}{2},j} = (M_v \tilde{V})_{i-\frac{1}{2},j} - (M_v V^n)_{i-\frac{1}{2},j} + \overline{Q_{i-\frac{1}{2},j} F_{i-\frac{1}{2},j}}^t + \frac{\overline{E_{i,j}}^t - \overline{E_{i-1,j}}^t}{h} \tag{2.43}$$

$$(R_V)_{i,j-\frac{1}{2}} = (M_v \tilde{V})_{i,j-\frac{1}{2}} - (M_v V^n)_{i,j-\frac{1}{2}} + \overline{Q_{i,j-\frac{1}{2}} F_{i,j-\frac{1}{2}}}^t + \frac{\overline{E_{i,j}}^t - \overline{E_{i,j-1}}^t}{h} \tag{2.44}$$

3. Compute the Newton increment (Φ', V') from the following inexact Jacobian system,

$$-(R_\Phi)_{i,j} = \Phi'_{i,j} + \frac{\alpha\Delta t}{h}\left(-\Phi^*_{i-\frac{1}{2},j} V'_{i-\frac{1}{2},j} + \Phi^*_{i+\frac{1}{2},j} V'_{i+\frac{1}{2},j} - \Phi^*_{i,j-\frac{1}{2}} V'_{i,j-\frac{1}{2}} + \Phi^*_{i,j+\frac{1}{2}} V'_{i,j+\frac{1}{2}}\right), \tag{2.45}$$

$$-(R_V)_{i-\frac{1}{2},j} = (M_v^* V')_{i-\frac{1}{2},j} + \alpha\Delta t \frac{\Phi'_{i,j} - \Phi'_{i-1,j}}{h}, \tag{2.46}$$

$$-(R_V)_{i,j-\frac{1}{2}} = (M_v^* V')_{i,j-\frac{1}{2}} + \alpha\Delta t \frac{\Phi'_{i,j} - \Phi'_{i,j-1}}{h}, \tag{2.47}$$

   As we will see below this involves solving a shifted Laplace equation for Φ' and then back-substitution to find V'.

4. Update $(\tilde{\Phi}, \tilde{V})$, $\tilde{\Phi} = \tilde{\Phi} + \Phi'$ and $\tilde{V} = \tilde{V} + V'$.

end

Algorithm 3: Nonlinear solver

Computation of (Φ', V') utilises an approximation to the pressure field Φ^*, this could be the previous pressure field Φ^n or a constant field of height 1. Similarly M_v^* represents an approximation to the mass matrix, this report will take $M_v^* := \mathbb{I}$.

Step 3 is achieved by first substituting (2.46) and (2.47) into (2.45) to produce,

$$\begin{aligned}\Phi'_{i,j} + \frac{(\alpha\Delta t)^2}{h^2}[&\Phi^*_{i-\frac{1}{2},j}(\Phi'_{i,j} - \Phi'_{i-1,j}) + \Phi^*_{i+\frac{1}{2},j}(\Phi'_{i,j} - \Phi'_{i+1,j}) \\ &+\Phi^*_{i,j-\frac{1}{2}}(\Phi'_{i,j} - \Phi'_{i,j-1}) + \Phi^*_{i,j+\frac{1}{2}}(\Phi'_{i,j} - \Phi'_{i,j+1})] = -\tilde{R}^l_{i,j}.\end{aligned} \tag{2.48}$$

The LHS of (2.48), which only depends on the pressure correction Φ', shall be represented by

$(H\Phi')_{i,j}$. In the case of $\Phi^* = 1$, this is simply the shifted Laplace operator. The RHS of (2.48) is constructed by,

$$\begin{aligned}\tilde{R}_{i,j} := (R_\Phi)_{i,j} + \frac{\alpha\Delta t}{h}[&\Phi^*_{i-\frac{1}{2},j}(R_v)_{i-\frac{1}{2},j} - \Phi^*_{i+\frac{1}{2},j}(R_v)_{i+\frac{1}{2},j} \\ &\Phi^*_{i,j-\frac{1}{2}}(R_v)_{i,j-\frac{1}{2}} - \Phi^*_{i,j+\frac{1}{2}}(R_v)_{i,j+\frac{1}{2}}],\end{aligned} \tag{2.49}$$

resulting in the following system to be solved in each Inexact Newton iteration:

$$(H\Phi')_{ij} = -\tilde{R}_{i,j}. \tag{2.50}$$

Given Φ', the values of V' are found via back substitution:

$$V'_{i-\frac{1}{2},j} = -\frac{\alpha\Delta t}{h}(\Phi'_{i,j} - \Phi'_{i-1,j}) + (R_v)_{i-\frac{1}{2},j} \tag{2.51}$$

$$V'_{i,j-\frac{1}{2}} = -\frac{\alpha\Delta t}{h}(\Phi'_{i,j} - \Phi'_{i,j-1}) + (R_v)_{i,j-\frac{1}{2}}. \tag{2.52}$$

2.2 Conjugate Gradient (CG) application in the method.

All three methods require a linear system in the form $\mathbf{Ax} = \mathbf{b}$ for unknown $\mathbf{x}$ to be solved at least once per time step iteration. For the two explicit schemes this is a result of a velocity mass matrix solve required for each coordinate direction, hence one iteration of Leapfrog requires two system solves. Similarly one iteration of the RK3 scheme requires two solves per stage resulting in six system solves per time step iteration.

The mass matrix in both velocity directions consists of a sparse tridiagonal matrix with an additional non zero element offset from the diagonal by n places every n rows due to the periodic boundaries. Similarly the shifted Laplace operator described by (2.48) could be represented by a sparse matrix with non zero entries mainly existing on the subdiagonal, diagonal and superdiagonal with additional entries on the n^{th} superdiagonal and subdiagonal. Additional entries will exist due to the periodic boundaries.

Both velocity mass matrices have a condition number $\kappa \sim \mathcal{O}(1)$ that is independent of the grid spacing. However the structure of the shifted Laplace operator represents a matrix with a larger condition number than that of the mass matrices. Furthermore the shifted Laplace operator matrix is dependant on grid and time step size, with larger time steps and smaller grid spacing resulting in larger condition numbers. As all three linear systems are represented by positive definite matrices, a Conjugate Gradient (CG) solver will be suitable as represented by the algorithm below. Since all the diagonal elements have a very similar magnitude, we do not precondition the system with a Jacobi iteration.

Data: $\mathbf{A}$,$\mathbf{b}$
Result: $\mathbf{x}$
Compute
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$,
$\mathbf{p}_0 := \mathbf{r}_0$
for $j = 1 : max\ iterations$ **do**
  $\alpha_j := \frac{\mathbf{r}_j^T \mathbf{r}_j}{\mathbf{r}_j^T \mathbf{A}\mathbf{p}_j}$
  $\mathbf{x}_{j+1} := \mathbf{x}_j + \alpha_j \mathbf{p}_j$
  $\mathbf{r}_{j+1} := \mathbf{r}_j - \alpha_j \mathbf{A}\mathbf{p}_j$
  if $\frac{||\mathbf{r}_{j+1}||_2}{||\mathbf{r}_0||_2} \leq Exit\ tolerance$ **then**
    $\mathbf{x} := \mathbf{x}_{j+1}$
  end
  $\beta_j := \frac{\mathbf{r}_{j+1}^T \mathbf{r}_{j+1}}{\mathbf{r}_j^T \mathbf{r}_j}$
  $\mathbf{p}_{j+1} := \mathbf{r}_{j+1} + \beta_j \mathbf{p}_j$
end

Algorithm 4: CG method as defined in reference [10].

As the matrix vector multiplications in all three methods represent stencil operations on scalar fields the implemented CG method is constructed as matrix free, by using these stencil operations. Firstly this reduces the memory requirements of the CG implementations through the removal of the requirement to store the matrix. Secondly this reduces the number of global memory operations required to perform the matrix vector multiplication on the GPU. This occurs as coefficients required to perform the stencil operation to the input state are stored locally in cache and do not have to be read from device memory per element.

2.3 MATLAB implementation.

For the above methods a prototype implementation in MATLAB was written to assess stability and accuracy. This MATLAB implementation was written with ease of porting to CUDA/C++ as a higher priority than optimizing for MATLAB.

For example, the application of a stencil based operation on a matrix. The following code demonstrates two possible methods to apply a velocity mass matrix to a MATLAB matrix object containing velocities. The first code utilises a circular shift of the entire matrix in a given direction with element wise addition, the second code utilises a double for loop to iterate over the entire matrix containing the input velocities. The first code would be much more efficient for a MATLAB implementation, however the second code is much closer to the final desired CUDA code. Hence CUDA kernels can be implemented more quickly based on the less efficient MATLAB code, with the advantage that the stencil has already undergone testing in MATLAB.

```
First code.
out=circshift(in,[1,0])/6 +(4/6)*in + circshift(in,[-1,0])/6;

Second code.
for j=1:n
    for i=2:n-1
        out(i,j)=(in(i-1,j)/6) +(4/6)*in(i,j) + (in(i+1,j)/6);
    end
```

```
end
```

3 Method convergence analysis

3.1 Spatial and time convergence

The aim of this section is to determine how the L2 norm error of each of the three methods relates to the chosen grid spacing and time step size. Formulated below as,

$$||\Phi_{h,\Delta t} - \hat{\Phi}||_{L2} \leq C_1 \Delta t^{\alpha} + C_2 h^{\beta}, \tag{3.1}$$

where $\Phi_{h,\Delta t}$ represents the pressure field at a fixed, non zero point in time calculated using one of the three schemes, with grid spacing h and time step size Δt. The error induced by one or more CG solves per iteration is minimised by setting a sufficiently small CG exit tolerance. This tolerance is chosen small enough to ensure that the dominant errors in the compared pressure profile are almost entirely spatial and time errors.

The initial condition used to determine the errors is constructed with a pressure and velocity field designed to be stationary, this pressure field is denoted $\hat{\Phi}$. For $\hat{\Phi}$, the pressure and velocity fields are defined radially around a centre (x_0, y_0), with Ψ a velocity in the tangential direction $(x - x_0, y - y_0)^{\perp}$.

$$r := \sqrt{(x - x_0)^2 + (y - y_0)^2}, \tag{3.2}$$

$$\text{Constants } \beta = 6,\ \omega = \frac{3}{20},\ \sigma = \frac{1}{5},\ f = 0.3, \tag{3.3}$$

$$\hat{\Phi}(r) := \begin{cases} 1 - \frac{1}{20}\exp\left(-\left(\frac{r}{\omega}\right)^{\beta}\right)\left(1 + \cos(\pi\frac{r^2}{\sigma^2})\right) & \text{if } r < \sigma \\ 1 & \text{if } r \geq \sigma \end{cases}, \tag{3.4}$$

$$\Psi(r) := \frac{rf}{2}\left(\sqrt{1 + \frac{4}{\sigma f^2}\frac{\partial \hat{\Phi}}{\partial r}} - 1\right) \tag{3.5}$$

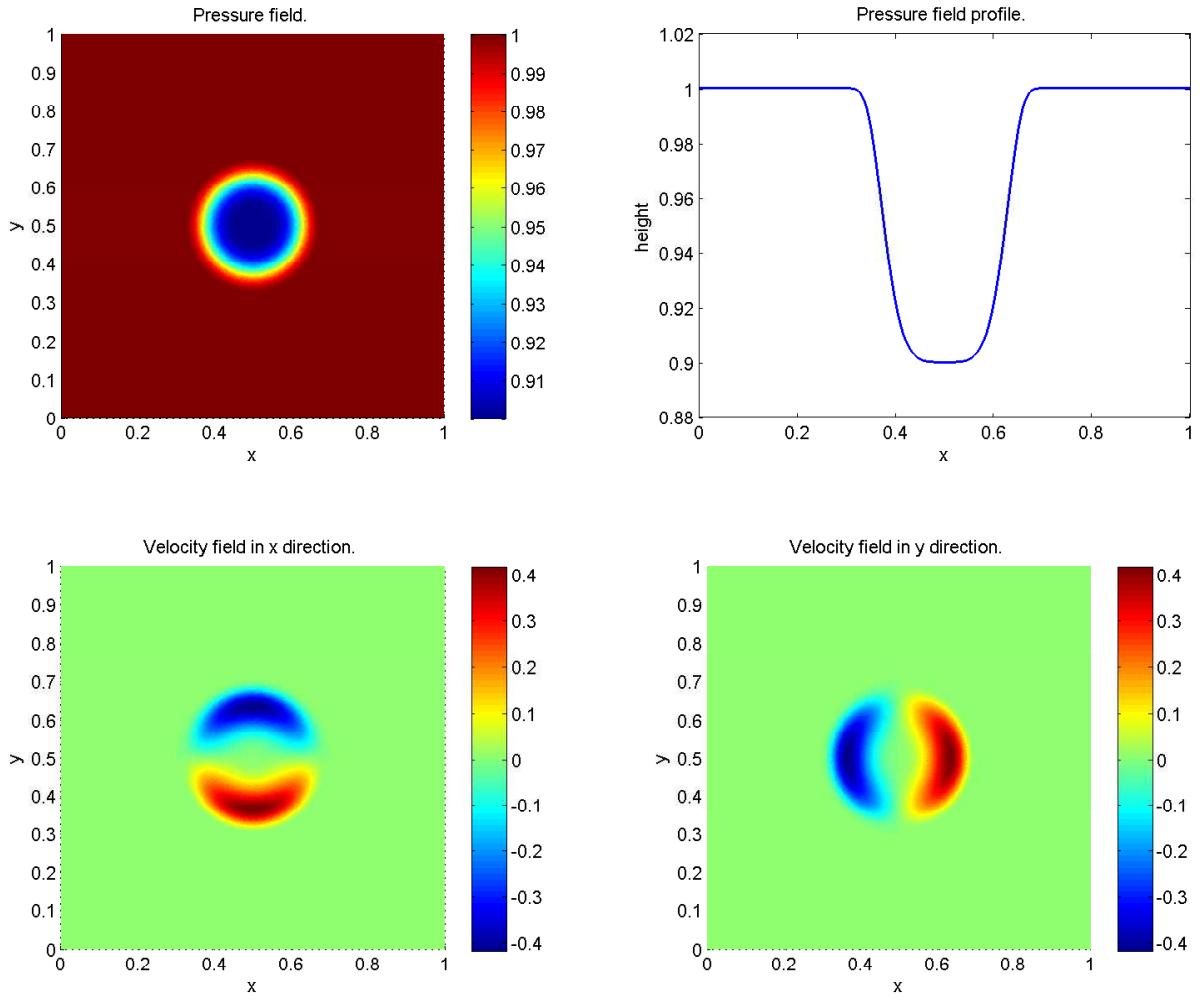

Figure 4: Pressure profile and velocity field for stationary vortex initial condition.

From this reference pressure field exact spatial discretisation errors can be evaluated after integration forward in time to a physical time of $t = 0.03$ using a small enough time step that temporal error may be ignored. Fitting a straight line to the log of the error produced by varying the grid spacing and integrating to a fixed point in time produces an estimate for C_2 and β, under the assumption that $C_1 \Delta t^{\alpha}$ is small.

$$\log(C_2) + \beta \log(h) \approx \log(||\Phi_{h,\Delta t} - \hat{\Phi}||_{L2}) \tag{3.6}$$

To determine the time discretisation errors in each scheme a reference profile $\tilde{\Phi}_{h,\delta t}$ is computed at 0.03s using a set grid spacing h and a small time step δt for each scheme using the same stationary vortex initial condition. For the computation of this reference profile, δt is again chosen to ensure that the error in the reference profile is predominately spatial. By then varying only the time step

size and computing the L2 norm of the difference between the reference profile and a test profile computed with a larger time step leaves the time discretisation error.

$$\log(C_1) + \alpha \log(\Delta t) \approx \log(||\Phi_{h,\Delta t} - \tilde{\Phi}_{h,\delta t}||_{L2}) \tag{3.7}$$

3.1.1 Spatial error tests

To determine spatial error convergence we apply the first method described in section 3.1 to each of the three time stepping schemes. By using the same small time step for each scheme and the same set of varying grid sizes we produce the following directly comparable plots. Errors are calculated after computing a solution at $t = 0.03$, requiring 768 iterations for each grid.

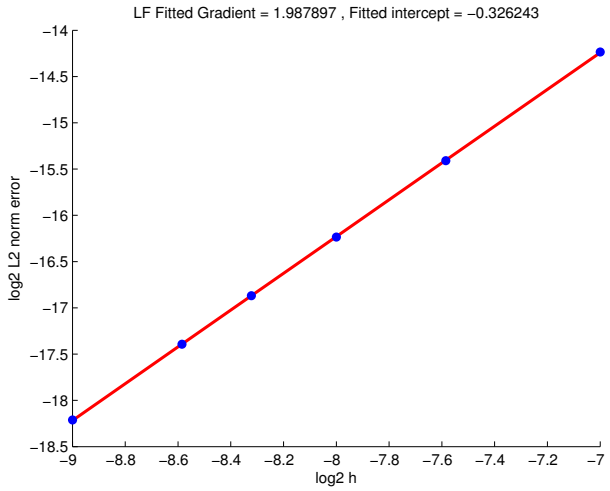

Figure 5: Leapfrog approximate log base two spatial error with fitted straight line. $h \in \{\frac{1}{512}, \frac{1}{384}, \frac{1}{320}, \frac{1}{256}, \frac{1}{192}, \frac{1}{128}\}$, $\Delta t := \frac{1}{25600}$.

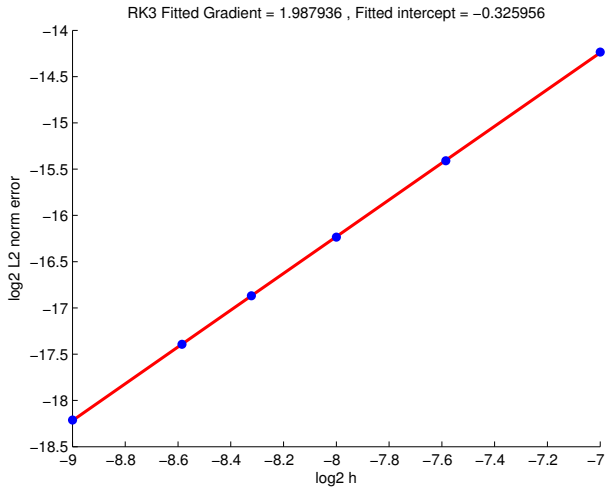

Figure 6: RK3 approximate log base two spatial error with fitted straight line. $h \in \{\frac{1}{512}, \frac{1}{384}, \frac{1}{320}, \frac{1}{256}, \frac{1}{192}, \frac{1}{128}\}$, $\Delta t := \frac{1}{25600}$.

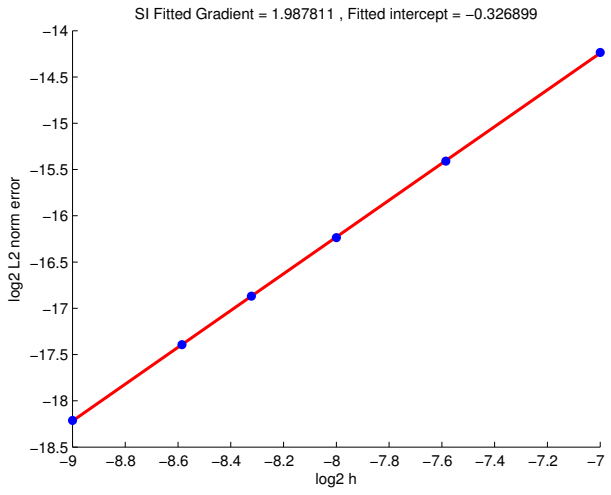

Figure 7: Semi-Implicit approximate log base two spatial error with fitted straight line. $h \in \{\frac{1}{512}, \frac{1}{384}, \frac{1}{320}, \frac{1}{256}, \frac{1}{192}, \frac{1}{128}\}$, $\Delta t := \frac{1}{25600}$.

Figures 5, 6 and 7 demonstrate that the three time stepping schemes have the same second order spatial error convergence. As indicated by the fitted straight lines with gradients that are approximately 2. The extent of the similarities between these plots indicates that all three time stepping implementations exhibit the same spatial error.

3.1.2 Time discretisation error tests.

The following figure is the result of applying the second method described in section 3.1 to each time stepping scheme. A solution is calculated at $t = 0.03$ using a set grid size of $h = \frac{1}{512}$, from this solution an L2 norm error can be computed using the stationary vortex initial profile. For later comparison between the schemes a horizontal, black, dashed line is drawn to represent an error of 2^{-25} and the intercept points are indicated.

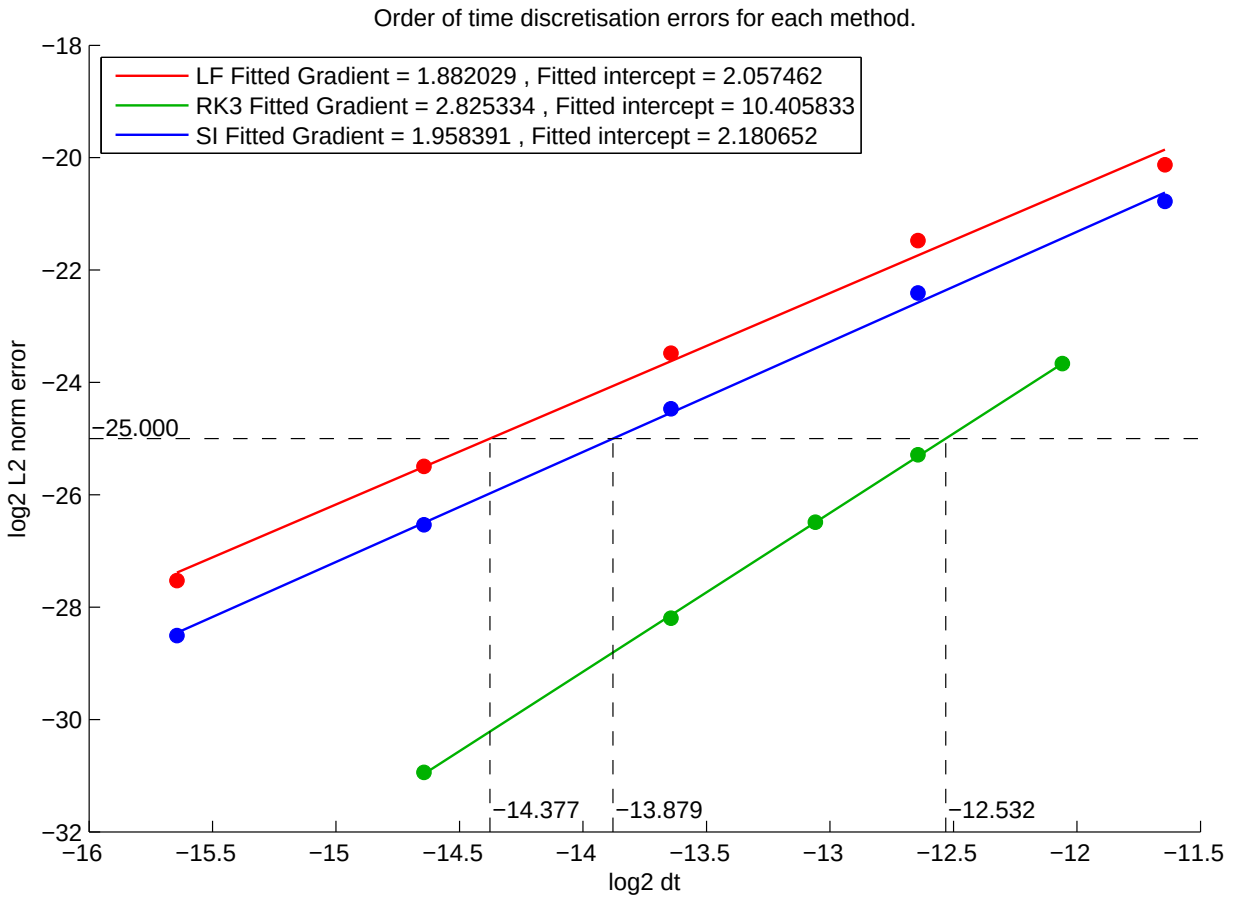

Figure 8: Log base 2 time discretisation errors for all three methods with set grid size $h = \frac{1}{512}$. With Leapfrog, Semi-Implicit using $\Delta t \in \{\frac{1}{51200}, \frac{1}{25600}, \frac{1}{12800}, \frac{1}{6400}, \frac{1}{3200}\}$ and RK3 using $\Delta t \in \{\frac{1}{25600}, \frac{1}{12800}, \frac{3}{25600}, \frac{1}{6400}, \frac{3}{12800}\}$.

These results indicate that Leapfrog and the Semi-Implicit scheme are second order in time. With the RK3 scheme demonstrating third order convergence in time. By setting a target L2 error of 2^{-25} at $t = 0.03$ and by establishing the intercept points for each scheme we determine the required time step size for each scheme.

Time stepping scheme	$\log_2$ intercept from plot	Time step size	Time step size normalised against Leapfrog
Leapfrog	-14.377	$4.70*10^{-5}$	1
RK3	-12.532	$1.69*10^{-4}$	3.59
Semi-Implicit	-13.879	$6.64*10^{-5}$	1.41

Table 3: Estimated required time step size for each scheme to obtain a L2 error of $2^{-25} \approx 3 * 10^{-8}$ at $t = 0.03$ with a grid size $h = \frac{1}{512}$, using the stationary vortex initial condition.

3.2 Inexact Newton and Conjugate Gradient error analysis

For a each of the three schemes it is anticipated that an output will contain an error dependant on not only the mesh and time step size but also an error attributed to the accuracy of the Inexact Newton and of the CG solve performed one or more times per iteration. In the case of the previous convergence analysis the exit tolerance of the CG solver was set sufficiently small to force the dominate errors to be the discretisation errors. However for practical purposes a *relative exit tolerance*[2] should be chosen small enough as to provide sufficient accuracy at each CG solve and large enough as to avoid excessive and expensive CG iterations. This section discusses the determination of a suitable relative exit tolerance.

For all three time stepping schemes, the grid and time step sizes were fixed at sufficiently fine values. Then only the exit tolerances were varied for the solvers. For the two explicit methods this involves varying the relative exit tolerance used as an exit condition for the CG solver for the two velocity mass matrix systems. In the Semi-Implicit scheme there are two relative exit tolerances, the first as an exit condition for the CG solver applied to the shifted Laplace solve; the second relative tolerance is an exit condition on the Inexact Newton (IN) method employed as a non linear solver for the Semi-Implicit scheme. For convergence it is expected that the exit tolerance on the CG solver has to be at least as small as the tolerance used for the IN solver exit condition. From this the exit tolerance for the IN iterations was varied with the exit tolerance for the CG solve set ten times smaller.

The following plots are the result of running each scheme to a set point in time whilst varying the exit tolerances as described. As with the previous error analysis the initial condition for the two explicit schemes is chosen to be a stationary vortex, this allows exact L2 norm errors to be computed between this initial condition and the output pressure profile at the set time of $t = 0.4$. This value was chosen to ensure that a satisfactory number of CG and IN solves have occurred.

For the Semi-Implicit scheme a relatively accurate pressure profile was computed at a non zero time using the RK3 scheme with small mesh and time step sizes. The initial pressure profile for this test case was chosen to be that of stationary vortex except unlike in the previous explicit tests, the velocity field consists of a relatively large and positive horizontal velocity with a relatively small and negative vertical velocity. This provides a test case where both pressure and velocity fields vary over time in a non-rotational manner as a ripple rather than a vortex.

[2]relative exit tolerance: the magnitude of the residual error at each iteration r_k is divided by the magnitude of the residual error at the first iteration r_0.

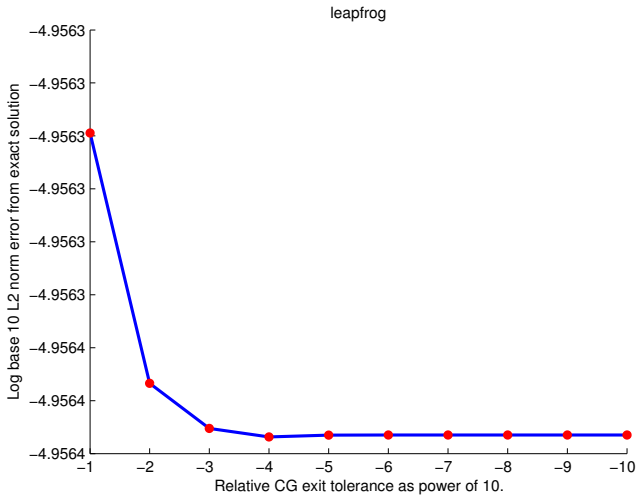

Figure 9: Leapfrog L2-norm error against relative CG exit tolerance, $h = \frac{1}{512}$, $\Delta t = \frac{1}{25600}$ with 10240 iterations.

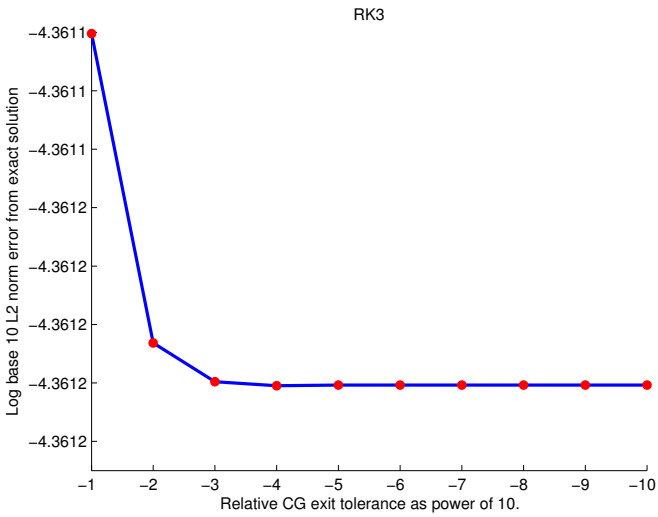

Figure 10: RK3 L2-norm error against relative CG exit tolerance, $h = \frac{1}{256}$, $\Delta t = \frac{1}{2560}$ with 1000 iterations.

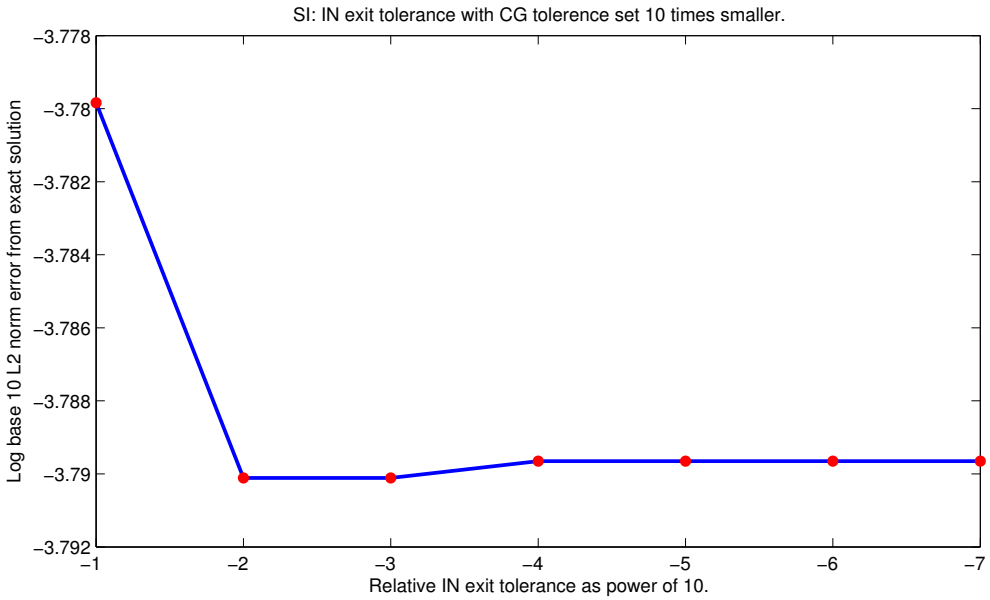

Figure 11: SI L2-norm error against relative IN exit tolerance, $h = \frac{1}{512}$, $\Delta t = \frac{1}{1024}$, 10 iterations with CG relative exit tolerance set 10 times smaller than relative IN exit tolerance.

Figures 9 and 10 indicate that a relative exit tolerance of 10^{-1} is sufficiently small for the CG solver applied to the velocity mass matrix solves in both the Leapfrog and RK3 schemes. As the plots are essentially flat, computing further iterations of CG will not be worthwhile. Similarly figure 11 indicates that a relative exit tolerance of 10^{-1} is sufficient for the Inexact Newton solver

combined with a CG solver with an exit condition based upon a relative tolerance of 10^{-2}. When applying these relative exit tolerances we expect output errors to be dominated by spatial and time discretisation errors rather than CG solve errors.

Furthermore an implementation aimed at a production environment should consider a CG solve exit condition based upon a set number of iterations as opposed to a exit tolerance. Current Met Office Semi-Implicit implementations use a relative CG exit tolerance of 10^{-4} combined with 4 Inexact Newton iterations.

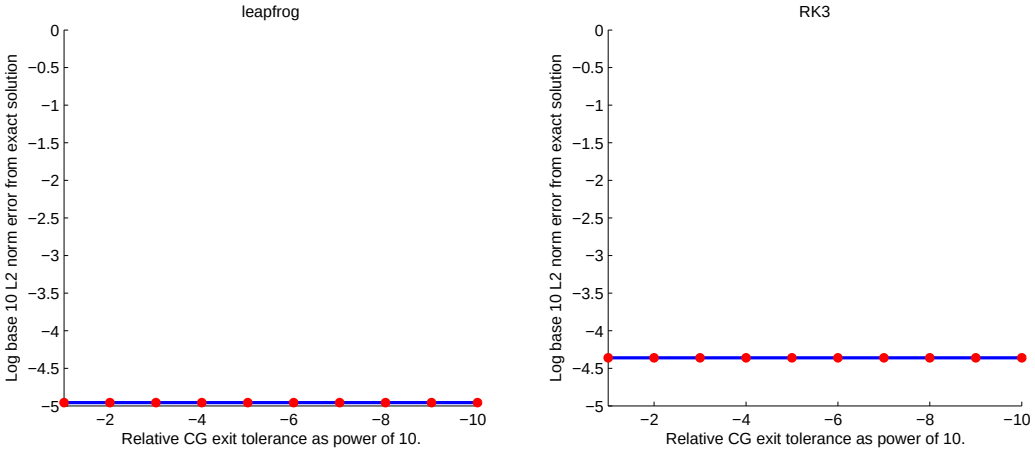

Figure 12: Plots demonstrating the scales of figure 9 and 10.

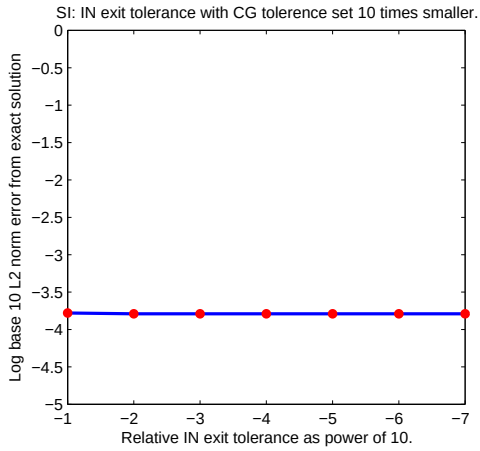

Figure 13: Plot demonstrating the scale of figure 11.

4 Implementation in CUDA

4.1 Introduction to CUDA and Multi GPU

Physically the GPUs exist as pieces of hardware within a node, they are built with on-board memory referred to as device memory and communicate with other components in the node via the

Peripheral Component Interconnect express (PCIe) bus. This bus is relatively slow with bandwidth of order $\mathcal{O}(10)$ GB/s and for this reason all significant computation will be performed on the GPU using device memory.

As the domain for the model problem is a two dimensional square, distributing the domain over available GPUs is handled by subdividing into smaller square sub-domains. Computation and movement of data on a sub-domain is then designated to one of the available GPUs.

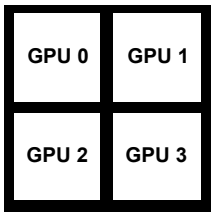

Figure 14: Domain distributed across four GPUs

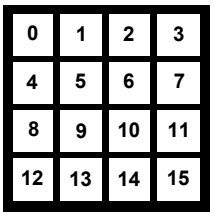

Figure 15: Domain distributed across sixteen GPUs

As with a CPU computation where data is not shared between processes a multi GPU computation requires communication between GPUs. From a topological standpoint the implementation is constructed similar to that of a CPU implementation by launching one processes for each GPU on a node. Data is subsequently communicated between processes using a Message Passing Interface (MPI), to improve the efficiency of communication a CUDA aware MPI implementation was chosen.

By using GPUDirect [11], the CUDA aware MPI can communicate data between GPUs whilst avoiding intermediate steps involving main system memory. This is the case for communication between GPUs within the same node and for GPUs residing in separate nodes. Improving the efficiency of communication should allow for better scaling to larger numbers of GPUs.

A halo exchange occurs when a GPU operating on a portion of the domain requires data from the adjacent portion stored upon a different GPU. All matrices are allocated with an additional element on all sides to give space to store the transferred data hence the term "halo". Halo exchanges between GPUs form the majority of the communications for all three of the implementations.

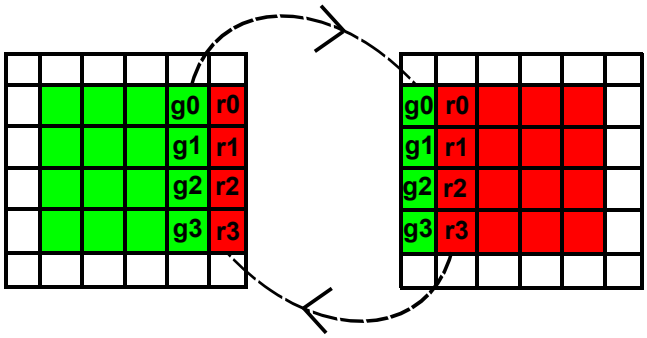

Figure 16: The movement of data in a horizontal halo exchange.

Given the time cost and frequency ($\mathcal{O}(10)$ per time step iteration) of halo exchanges there is significant motivation to ensure that this bidirectional operation is efficient. Employing a third

party library named Generic Communication Library (GCL) [13] facilitates efficient transfers by rearranging the halo data prior to performing the transfer via GPUDirect. GCL also enables an implementation to transfer halos in parallel to other operations, providing a straightforward method to hide communications behind computation work resulting in a higher efficiency. In the following example line 1 packs the data and subsequently initiates the halo exchange of a vector named `q_field_l` on the following line. Line 6 blocks the process until halo exchange completion after which the halo is unpacked.

```
1 he->pack(q_field_l);
2 he->start_exchange();
3
4 // Intermediate commands here.
5
6 he->wait();
7 he->unpack(q_field_l);
```

Alternatively, GCL can be instructed to perform halo exchanges for multiple fields at once. The final major third party library employed in the implementation is the NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library [14]. This library is analogous to traditional CPU BLAS implementations, yet designed to perform all operations on CUDA devices. From an implementation writing perspective calls to the cuBLAS library are extremely similar to their BLAS counterparts.

4.2 CUDA implementation, structure and design

4.2.1 Outline implementation structure

The steps below represents a general structure outline for main code:

1. Setup CUDA, MPI & GCL.
2. Read input variables to determine problem size, method, tolerances etc.
3. Allocate memory for and setup initial condition.
4. Run chosen method.
5. Post processing of results if required.
6. Clean up allocated memory.

Important variables are carried around in three separate structures acting as containers for general variables, tolerances and other miscellaneous options. Similarly a structure containing three double pointers provides a container to store a state of the system consisting of two velocity fields and a pressure field. Each of the methods is passed a state structure pointing to the memory containing an initial condition, on completion the same memory is used as an output for the final pressure and velocities. Adopting the same approach to the CG solvers provides functions that return arrays for $\mathbf{x}$, given an input RHS $\mathbf{b}$ for $A\mathbf{x} = \mathbf{b}$.

An important feature of all three methods is that all the data remains on the GPU throughout the computation. No data is copied over the PCIe bus between iterations or within iterations. If

the state vectors are required in host memory for some intermediate computation, then the state vectors will need to be copied from the GPU to host and back again between iteration. Due to the limited bandwidth of the PCIe bus this would significantly impact the performance of the implementation.

Additional memory is allocated within each method, as all of the three methods have different memory requirements. Furthermore it is particularly important to avoid memory allocation and deallocation within timed portions of code, as these operations are slow. The aforementioned CG solver function avoids this wasted time with additional input pointers referring to space that may be used for temporary storage.

4.2.2 Data layout and global reductions

As the domain is two dimensional, when implementing the spatial discretisation it is convenient to work with two coordinate directions. However, in memory one dimensional arrays store the values in a given field using a row major method, with a resultant structure consisting of blocks storing interior values separated by stored halo values as illustrated by figure 17. This is the structure that the GCL library is expecting to use, for user written code a simple macro provides a map between a two dimensional index and a one dimensional index. In the macro, `NX` and `NY` are the total side length of the sub-domain including halos, with `ix` and `iy` the input indices.

```
1  #define LINIDX_2D(NX,NY,ix,iy)\
2  ( (NX)*((iy)+1) \
3  + ((ix)+1)\
4  )
```

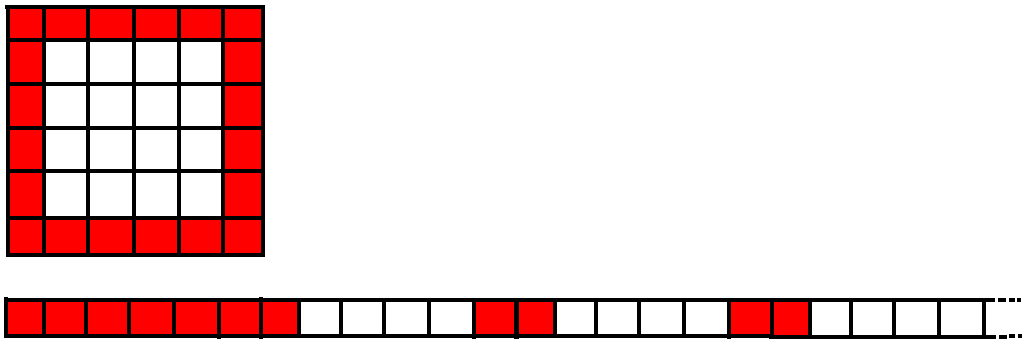

Figure 17: Data layout in memory with halo regions shaded red.

As discussed previously, consecutive CUDA threads should refer to horizontally adjacent values such that memory access can be coalesced. Consecutive threads which are vertically adjacent would require memory access n_{loc} elements apart, where n_{loc} denotes the local side length.

An additional complication arising from this data structure is how to efficiently perform global reduction operations. Given an array of values the cuBLAS library provides efficient tools to compute an absolute sum, or the dot product between two vectors. However direct use of these tools would incorporate values stored in halo regions in the global reduction, leading to an incorrect result. A method was sought that efficiently computes the desired result whilst correctly handling the values stored in halo regions.

Two methods were considered. A potential, but excluded method could use an additional array to mask halo values by storing ones in the places of interior values and zeros in the indexes containing halo values. However storing this mask would require both additional memory and additional memory operations. As computing the dot product of a vector with itself would require squaring all the values in an array using a CUDA kernel, then computing the dot product between the squared values and the suggested mask array.

The method favoured by this implementation is utilise the CUDA kernel given below to set all halo values in a given input array to zero. Following the removal of the halos, calls to the cuBLAS library provide efficient methods to compute absolute sums or dot products. In the case of a dot product between two vectors, such as in each iteration of CG, only one input vector needs to have halos removed. Although the kernel does not efficiently use available memory bandwidth the resultant method performs better than a mask based approach.

```
__global__ void strip_halo(double *d_a, vars_store vars){

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

if (ix < vars.NX-1 ){
d_a[ix]=0.0;
d_a[1+ix+(vars.NY-1)*vars.NX]=0.0;
d_a[(ix+1)*vars.NX-1]=0.0;
d_a[(ix+1)*vars.NX]=0.0;
}}
```

5 Analysis of CUDA implementation

5.1 Time breakdown per iteration

The aim of this section is to determine the proportion of time spent performing CG solves per iteration for each method. For this analysis each time stepping scheme was run on a single GPU, to avoid halo exchanges and to time duration of each component using the NVIDIA profiler. The test domain used had side length $n = 2048$ with an initial pressure profile $\hat{\Phi}(r)$ consisting of a central depression and velocity fields V set to 0.2, as given below.

$$r := \sqrt{(x - x_0)^2 + (y - y_0)^2}, \tag{5.1}$$

$$\text{Constants } \beta = 6,\ \omega = 0.15,\ \sigma = 0.2,\ f = 0.3, \tag{5.2}$$

$$\hat{\Phi}(r) := \begin{cases} 1 - \frac{1}{20}\exp\left(-\left(\frac{r}{\omega}\right)^{\beta}\right)\left(1 + \cos(\pi\frac{r^2}{\sigma^2})\right) & \text{if } r < \sigma \\ 1 & \text{if } r \geq \sigma \end{cases}, \tag{5.3}$$

$$V_{i-\frac{1}{2},j} = 0.2,\ V_{i,j-\frac{1}{2}} = 0.2,\ i, j = 1, \ldots, n. \tag{5.4}$$

The time step size was set to $\delta t = \frac{1}{20480}$ for all three schemes (from a CFL number of 0.1). From the results of section 3.2 a relative CG exit tolerance of 10^{-1} was chosen for the both Leapfrog and RK3 schemes. However for the Semi-Implicit method a CG exit tolerance of 10^{-2} combined with a Inexact Newton exit tolerance of 10^{-1} was applied. Using these tolerances, each mass matrix

solve in the explicit schemes required three CG iterations per solve. With the Semi-Implicit scheme requiring one iteration of Inexact Newton, which involved two CG iterations for the Shifted Laplace solve. For the Semi-Implicit method, only computation within the Inexact Newton loop was timed. Hence these results only investigate the proportion of time in each Newton iteration that is the elliptic solver (the CG solver).

Time stepping scheme	Total CG solve time for $(M_v V)_{i-\frac{1}{2},j}$ (ms)	Total CG solve time for $(M_v V)_{i,j-\frac{1}{2}}$ (ms)	Time spent not in CG (ms)
Leapfrog	14.95 (3 CG iterations)	15.08 (3 CG iterations)	9.00
RK3	44.85 (9 CG iterations)	45.24 (9 CG iterations)	54.88

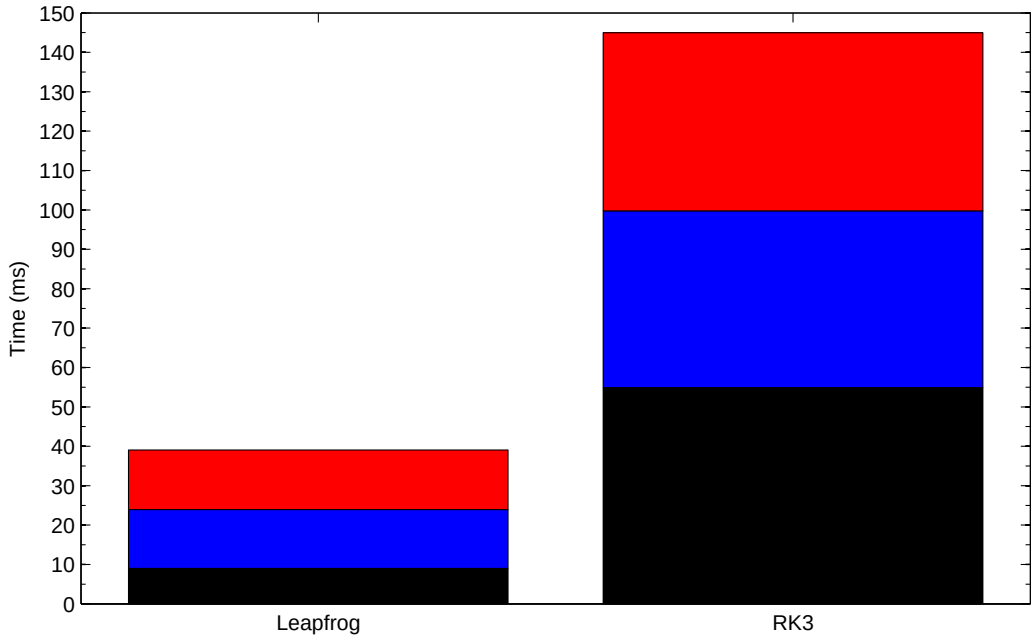

Figure 18: Time breakdown of one iteration of Leapfrog and RK3. Blue represents the time taken by the horizontal, $(MV)_{i-\frac{1}{2},j}$ CG solves. Red represents the vertical, $(MV)_{i,j-\frac{1}{2}}$ CG solves. Black represents the time taken to perform the rest of the iteration, such as evaluations of $\mathbf{F}$ and vector addition.

Total CG solve time for Shifted Laplace (ms)	Time taken to compute remaining proportion of IN iteration (ms)
10.72 (2 CG iterations)	19.76

Table 4: Breakdown of one Inexact Newton iteration for side length $n = 2048$, $\delta t = \frac{1}{20480}$, IN exit tolerance 10^{-1}, CG exit tolerance 10^{-2}.

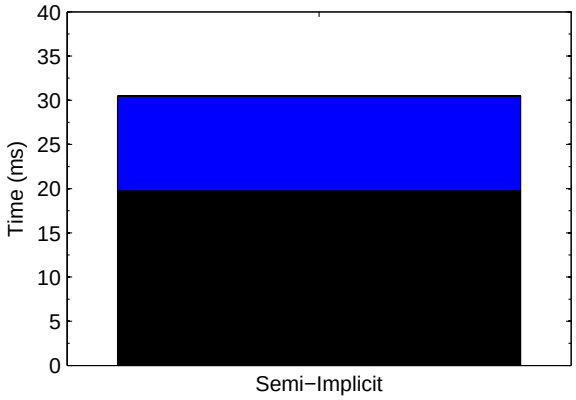

Figure 19: Graphical breakdown of table 4 with CG time in blue.

These results demonstrate that the CG solves form a significant proportion of the time spent computing each iteration.

5.2 Memory bandwidth and computation rate for the CG solvers

Firstly the reference CG algorithm (algorithm 4) is taken as a theoretical best case scenario for the CG implementation. This represents the minimum number of operations required to perform one iteration of CG per element in the domain. It is then assumed that the GPU caches all values read from device memory to obtain a lower bound for device memory bandwidth which is referred to as useful bandwidth.

Operation		Double precision operations	Memory read operations	Memory write operations
$A\mathbf{p_j}$:	Shifted Laplace	9	1	1
	$(M_vV)_{i-\frac{1}{2},j}$	5	1	1
	$(M_vV)_{i,j-\frac{1}{2}}$	5	1	1
$\mathbf{r}_j^T\mathbf{r}_j$ & $\mathbf{r}_j^T(A\mathbf{p}_j)$		2	2	0
$\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j\mathbf{p}_j$ & $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j A\mathbf{p}_j$		2	2	1
$\mathbf{p}_{j+1} = \mathbf{r}_{j+1} + \beta * \mathbf{p}_j$		2	2	1
Totals for Shifted Laplace		19	11	4
Totals for $(M_vV)_{i-\frac{1}{2},j}$		15	11	4
Totals for $(M_vV)_{i,j-\frac{1}{2}}$		15	11	4

Table 5: Theoretical minimum number of operations per element for all three CG methods, assuming perfect memory caching.

As previously discussed the Tesla M2090 claims to be able to perform 30 double precision operations per memory operation. The final three lines in the above table show that there are less than double the number of computation operations than memory references, hence demonstrating the

extent to which any sensible CG implementation is bound by memory bandwidth over computation rate.

The next step is to time each CG method applied to a problem of known size for a known number of iterations to determine useful memory bandwidth and computation rates. The following results are calculated from the recorded wall time taken for 200 CG iterations on a problem consisting of 5120^2 elements. As expected the achieved computation rates are well below peak performance of 666 GFLOPs.

Method	Time taken (s)	Computation rate (GFLOPs)	Useful memory bandwidth (GB/s)
Shifted Laplace	5.61	17.76 (2.7 %)	104.45 (59.0 %)
$(M_v V)_{i-\frac{1}{2},j}$	5.53	14.22 (2.1 %)	105.96 (59.9 %)
$(M_v V)_{i,j-\frac{1}{2}}$	5.56	14.14 (2.1 %)	105.38 (59.5 %)

Table 6: Estimated performance metrics using 200 CG iterations on a domain consisting of 5120^2 elements on a NVIDIA Tesla M2090. Given percentages are with respect to a claimed peak compute performance of 666 GFLOPs and 177 GB/s memory bandwidth.

For more detailed memory bandwidth usage, each CUDA kernel could be manually timed and the above method applied via manually counting memory references. Alternatively, the NVIDIA Visual Profiler [17] analyses a given input executable to return desired metrics for each kernel run within that executable. For example, as tabulated below, the amount of memory bandwidth used by individual kernels.

Operation	kernel name	Memory bandwidth (GB/s)	Average runtime (ms)
$A\mathbf{p_j}$	`helm_op`	143.22 (80.9 %)	0.821
	`mmu_kernel`	142.90 (80.7 %)	0.762
	`mmv_kernel`	142.99 (80.8 %)	0.804
$\mathbf{r}_j^T\mathbf{r}_j$ &	`strip_halo`	52.59 (29.7%)	0.007
$\mathbf{r}_j^T(A\mathbf{p}_j)$	`dot_kernel(cuBLAS)`	148.57 (83.9%)	0.442
$\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j\mathbf{p}_j$ & $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j A\mathbf{p}_j$	`axpy_kernel_val(cuBLAS)`	153.26 (86.6%)	0.806
$\mathbf{p}_{j+1} = \mathbf{r}_{j+1} + \beta * \mathbf{p}_j$	`axpy`	148.73 (84.0%)	0.877

Table 7: Device memory bandwidths for CUDA kernels forming the CG solvers as reported by the NVIDIA Visual Profiler. Run on a single Tesla M2090 with side length $n = 2048$. Percentages given with respect to a claimed peak of 177 GB/s

5.3 Efficient evaluation of the function F

For the two explicit methods a significant part of the computation required to produce a new iteration is the evaluation of $\mathbf{F}$, defined in the spatial discretisation section as:

$$[\mathbf{F}_\Phi(\Phi,\mathbf{v})]_{i,j} := \frac{-1}{h}(-\Phi_{i-\frac{1}{2},j}V_{i-\frac{1}{2},j} + \Phi_{i+\frac{1}{2},j}V_{i+\frac{1}{2},j} - \Phi_{i,j-\frac{1}{2}}V_{i,j-\frac{1}{2}} + \Phi_{i,j+\frac{1}{2}}V_{i,j+\frac{1}{2}}), \tag{5.5}$$

$$[\mathbf{F}_\mathbf{V}(\Phi,\mathbf{v})]_{i-\frac{1}{2},j} := -Q_{i-\frac{1}{2},j}F_{i-\frac{1}{2},j} - \frac{E_{i,j}-E_{i-1,j}}{h}, \tag{5.6}$$

$$[\mathbf{F}_\mathbf{V}(\Phi,\mathbf{v})]_{i,j-\frac{1}{2}} := -Q_{i,j-\frac{1}{2}}F_{i,j-\frac{1}{2}} - \frac{E_{i,j}-E_{i,j-1}}{h}. \tag{5.7}$$

More specifically each Leapfrog iteration requires one evaluation of $\mathbf{f}$ with each RK3 iteration requiring three evaluations, one at each stage. Hence it is crucial that the CUDA implementation of $\mathbf{f}$ be as efficient as possible, as with previous work implementing a Forward Euler method [16]. The principle problem in the evaluation of $\mathbf{F}$ is efficiently computing and performing the halo exchange for the Q field, which is a prerequisite for the computation of $\mathbf{F}_\mathbf{V}$. The implemented solution breaks the computation into two distinct parts; the first computes $\mathbf{F}_\Phi$ via the stencil operation described by (5.5) and computes Q at the corners of the grid cells using (2.18) and (2.19). Following a halo exchange of Q, $\mathbf{F}_\mathbf{V}$ is computed at the edges of the grid cells using the two stencil operations provided by (5.6) and (5.7).

Data: Φ,$\mathbf{v}$
Result: $\mathbf{F}(\Phi,\mathbf{v})$

1. Compute $\mathbf{F}_\Phi(\Phi,\mathbf{v})$ and $Q_{i-\frac{1}{2},j-\frac{1}{2}}$ for $i,j = 1,\ldots,n$.
2. Perform halo exchange of Q field.
3. Compute $\mathbf{F}_\mathbf{V}(\Phi,\mathbf{v})$.

Algorithm 5: Algorithm to compute $\mathbf{F}(\Phi,\mathbf{v})$

This algorithm automatically divides the computation into two separate CUDA kernels; the first named `q_n_fun` performs step one by applying the stencil to compute $\mathbf{F}_\Phi$ and the stencil to compute new Q field. Similarly the second kernel named `u_v_fun` performs step three via stencils (5.6) and (5.7). By grouping two or more stencil operations into a single kernel and explicitly caching variables required by both stencils we ensure that these values are read from device memory only once. The combining of multiple stencil operations in this manner is referred to as kernel fusion. Table 8 below presents the useful read and write bandwidths for the two kernels, with table 9 presenting the estimated computation rate and recorded memory bandwidths. As with previous analysis, all double precision operations are estimated to take the same time to perform.

Kernel name	Memory read operations	Memory write operations	Time taken (ms)	Useful memory bandwidth (GB/s)
`q_n_fun`	3	2	2.272	68.77 (38.9 %)
`u_v_fun`	4	2	3.632	51.63 (29.2 %)

Table 8: Estimated useful bandwidth assuming perfect caching using a domain consisting of 2048^2 elements on a NVIDIA Tesla M2090 with timings provided by the NVIDIA profiler. Given percentages are with respect to a claimed peak memory bandwidth of 177 GB/s.

Kernel name	Average runtime (ms)	Double precision operations per element	Estimated computation rate (GFLOPs)	Percentage of peak computation rate (%)
`q_n_fun`	2.272	25	46.15	6.9
`u_v_fun`	3.632	58	66.98	10.1

Kernel name	Memory bandwidth (GB/s)	Percentage of peak (%)
`q_n_fun`	114.65	64.8
`u_v_fun`	87.20	49.3

Table 9: Metrics provided by the NVIDIA profiler for a domain of side length $n = 2048$ ran on a NVIDIA Tesla M2090.

Both CUDA kernels have approximately half the useful bandwidth of the CG implementations, however their computation rate is much higher. It should be noted that these two kernels apply a more complicated stencil structure.

5.4 Performance metrics for Inexact Newton kernels

As in the previous two sections, useful memory read bandwidth and useful memory write bandwidth was established for each constituent CUDA kernel in the Inexact Newton method. The computation rate was approximated by counting the performed number of double precision operations for each kernel. The following table matches the operations required in Algorithm 3 with the corresponding CUDA kernel and its estimated computation rate.

Operation performed	Kernel name	Double precision operations	Time taken (ms)	Computation rate (GFLOPS)
Compute $Q_{i-\frac{1}{2},j-\frac{1}{2}}$ values	`q_func`	10	1.755	23.90 (3.6 %)
Compute $F_{i-\frac{1}{2},j}$ & $F_{i,j-\frac{1}{2}}$ values	`F_u_v_func`	28	2.706	43.40 (6.5 %)
Compute $E_{i,j}$ values	`e_fun`	7	1.374	21.37 (3.2 %)
Compute time average $\overline{X}$ values	`ta_ker`	5	0.880	23.83 (3.6 %)
Evaluate $(R_\Phi)_{i,j}$	`phi_residual`	35	3.019	48.63 (7.3 %)
Evaluate $(R_V)_{i-\frac{1}{2},j}$	`U_residual`	14	1.875	31.32 (4.7 %)
Evaluate $(R_V)_{i,j-\frac{1}{2}}$	`V_residual`	14	1.916	30.65 (4.6 %)
Compute $V'_{i-\frac{1}{2},j}$ & $V'_{i,j-\frac{1}{2}}$	`vel_inc`	8	2.487	13.49 (2.0 %)
Compute $\mathbf{a} = \mathbf{a} + \mathbf{b}$	`daapb`	8	0.873	38.44 (5.8 %)

Table 10: Semi-Implicit CUDA kernels matched with performed operation.

Kernel name	Memory read operations	Memory write operations	Time taken (ms)	Useful memory bandwidth (GB/s)	Percentage of peak (%)
`q_func`	3	1	1.755	71.23	40.2
`F_u_v_func`	4	2	2.706	69.29	39.1
`e_fun`	3	1	1.374	90.97	51.4
`ta_ker`	2	1	0.880	106.53	60.2
`phi_residual`	6	1	3.019	72.46	40.9
`U_residual`	4	1	1.875	83.34	47.1
`V_residual`	4	1	1.916	81.55	46.1
`daapb`	2	1	0.873	107.39	60.7
`vel_inc`	3	2	2.487	62.83	35.5

Table 11: Useful memory bandwidths for the Semi-Implicit implementation.

The Semi-Implicit implementation in this report performs the time averaging operation for the energies E and the perpendicular fluxes QF in two parts. By computing an intermediate value then time averaging with the corresponding value at the previous time step, using two separate kernels. An improvement would be to perform the time averaging operation at the same time the new intermediate value is computed. This would save one memory read and one memory write per element.

6 Multi-GPU scaling performance

This section investigates how each of the three time stepping schemes behave as the domain is split across a varying number of GPUs. Two different types of scaling are investigated, namely strong scaling and weak scaling. In strong scaling the domain is fixed at a predetermined size, independent of the number of GPUs applied. Hence the sub-domain size per GPU decreases as the number of applied GPUs increases. In contrast for weak scaling the sub-domain size remains constant as the number of GPUs increases resulting in an increase in the overall domain size.

The initial condition used consists of a pressure field with a central depression combined with a velocity field set to 0.2, as defined in equations (5.1) to (5.4). 100 time step iterations were computed for each scheme with a fixed time step size using a CFL number of 0.1 for all tests. Rather than use relative exit tolerances as exit conditions, an explicit number of iterations was computed in all three CG methods and for the Inexact Newton solver. The velocity mass matrix CG solve performs three iterations with the Shifted Laplace CG solve also set to three iterations. Similarly the Semi-implicit method performs two Inexact Newton iterations per time step. All tests were performed on the Emerald supercomputer. In all the tests, efficiency is computed against the time taken for one GPU to complete the 100 iterations without any halo exchanges.

6.1 Leapfrog

Number of GPUs	Sub-domain size	Time (s)	Scaling efficiency (%)
1	5120^2	25.71	100.0
4	2560^2	7.02	91.6
9	1707^2	3.83	74.6
16	1280^2	2.58	62.3
25	1024^2	2.05	50.2
36	853^2	1.71	41.8
64	640^2	1.41	28.6

Table 12: Leapfrog strong scaling.

Number of GPUs	Global domain size	Time (s)	Scaling efficiency (%)
1	5120^2	25.71	100.0
4	10240^2	26.60	96.7
9	15360^2	27.29	94.2
16	20480^2	29.20	88.1
25	25600^2	28.01	91.8
36	30720^2	28.38	90.6
64	40960^2	30.55	84.2

Table 13: Leapfrog weak scaling.

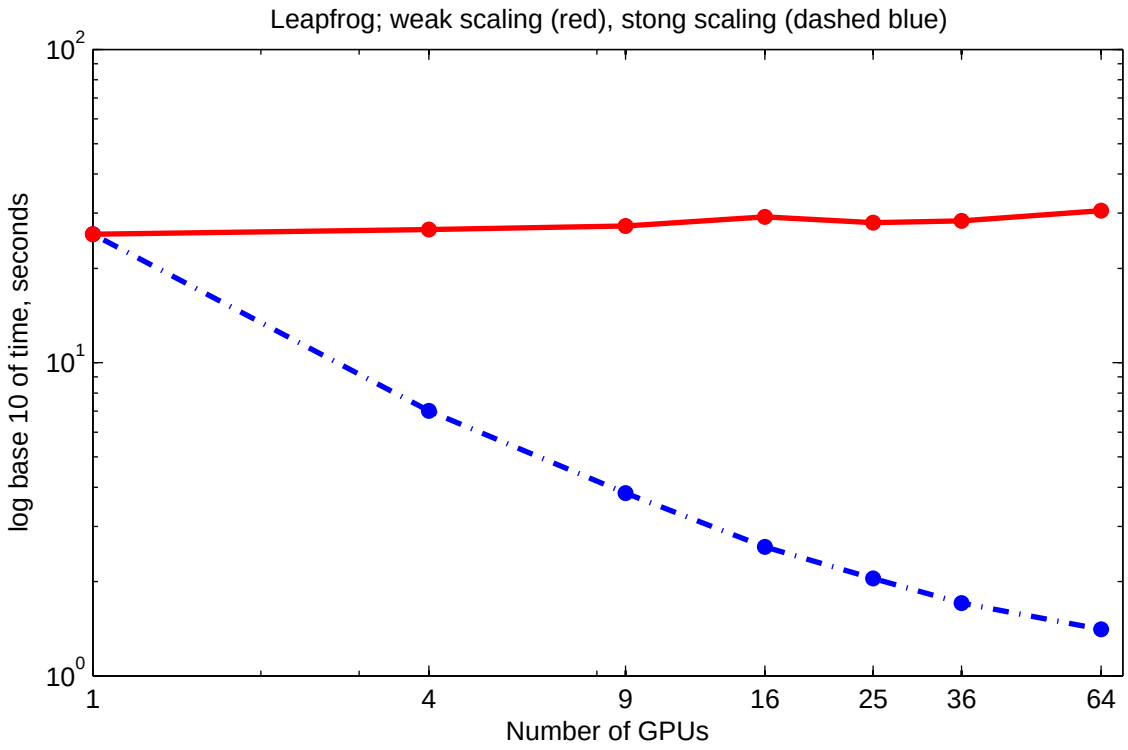

Figure 20: Log-Log plot of weak scaling and strong scaling for the Leapfrog scheme.

6.2 RK3

Number of GPUs	Sub-domain size	Time (s)	Scaling efficiency (%)
1	5120^2	91.52	100.0
4	2560^2	27.13	84.3
9	1707^2	17.69	57.5
16	1280^2	13.38	42.7
25	1024^2	11.42	32.4
36	853^2	10.33	24.6
64	640^2	4.44	32.2

Table 14: RK3 strong scaling.

Number of GPUs	Global domain size	Time (s)	Scaling efficiency (%)
1	5120^2	91.52	100.0
4	10240^2	99.53	91.9
9	15360^2	106.21	86.2
16	20480^2	104.14	87.9
25	25600^2	105.30	86.9
36	30720^2	105.92	86.4
64	40960^2	107.61	85.0

Table 15: RK3 weak scaling.

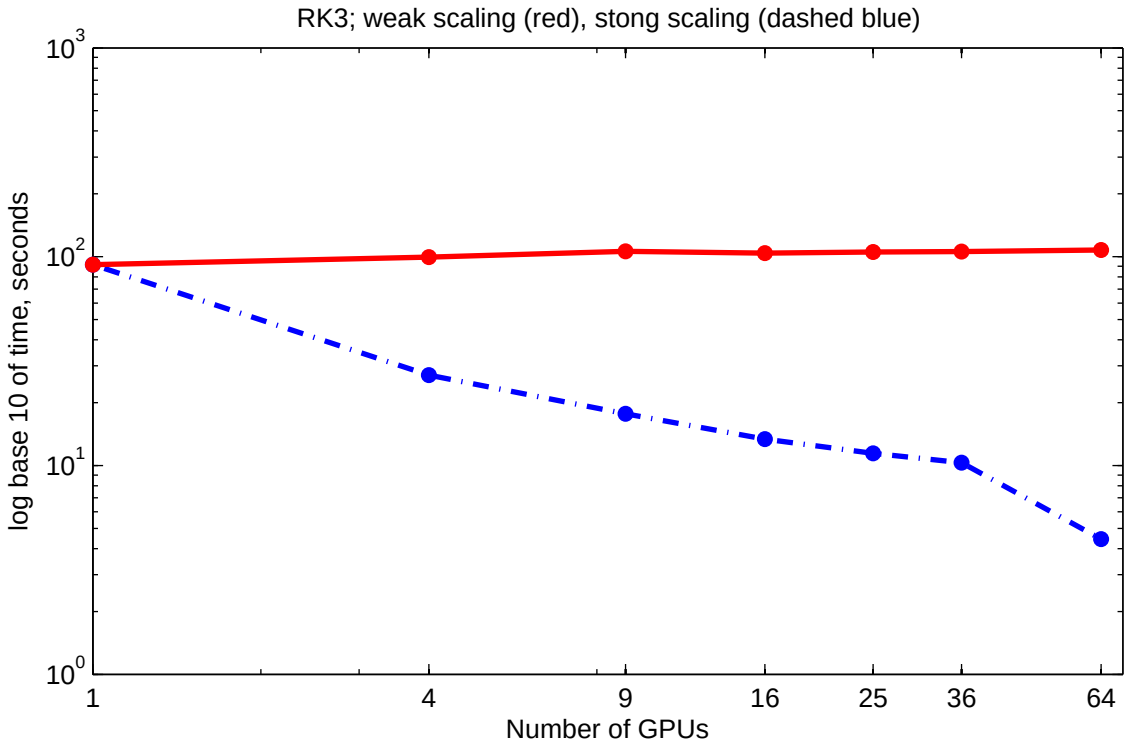

Figure 21: Log-Log plot of weak scaling and strong scaling for the RK3 scheme.

6.3 Semi-Implicit

Number of GPUs	Sub-domain size	Time (s)	Scaling efficiency (%)
1	5120^2	52.34	100.0
4	2560^2	14.62	89.5
9	1706^2	7.25	80.2
16	1280^2	4.55	72.0
25	1024^2	3.43	61.0
36	853^2	2.73	53.2
64	640^2	2.04	40.1

Table 16: Semi-Implicit strong scaling.

Number of GPUs	Global domain size	Time (s)	Scaling efficiency (%)
1	5120^2	52.34	100.0
4	10240^2	55.86	93.7
9	15360^2	55.39	94.5
16	20480^2	54.94	95.3
25	25600^2	55.93	93.6
36	30720^2	54.73	95.7
64	40960^2	57.37	93.0

Table 17: Semi-Implicit weak scaling.

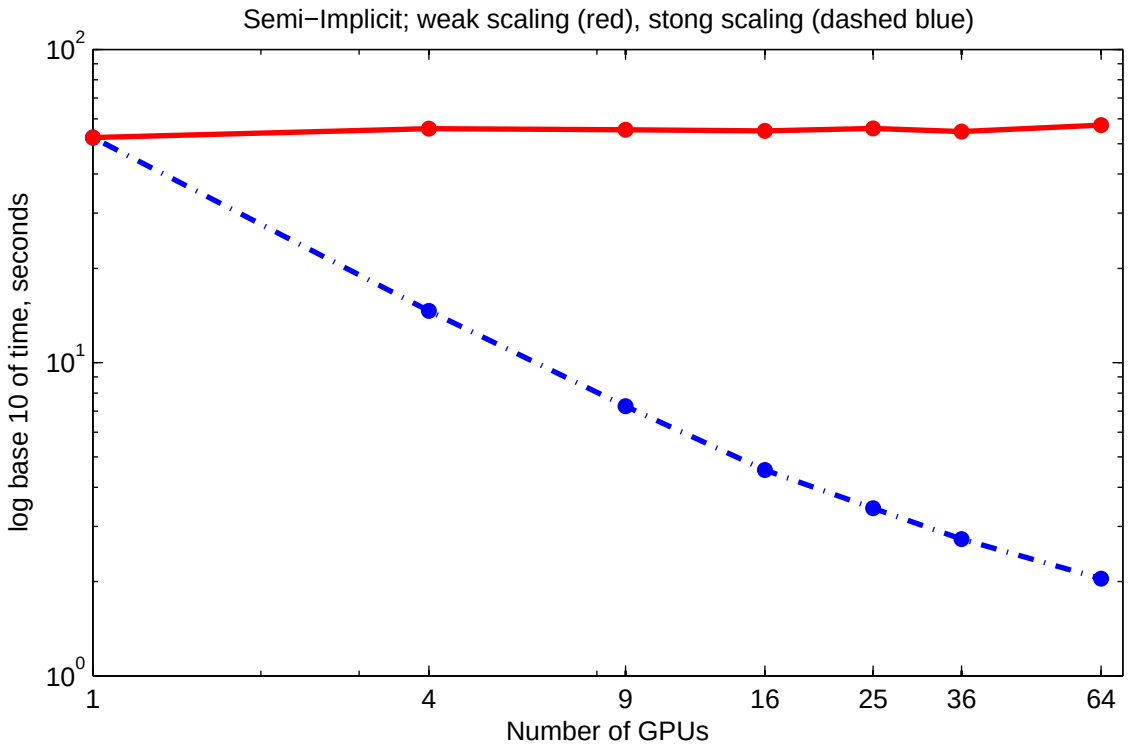

Figure 22: Log-Log plot of weak scaling and strong scaling for the Semi-Implicit scheme.

All three time stepping schemes demonstrate decent strong scaling to 36 GPUs with the Semi-Implicit scheme scaling reasonably well to 64 GPUs. When a domain of side length $n = 5120$ is distributed across 64 GPUs the resultant sub-domain size of 640^2 elements per GPU becomes too small to be efficiently worked upon. All three schemes perform near optimally as the problem size increases at the same rate as the number of GPUs. In this scenario the sub-domain sizes remain large enough to be efficiently worked upon by the GPU.

To determine whether halo exchange time was a root cause of the loss of scaling efficiency 300 halo exchanges were timed in a strong scaling scenario and a weak scaling scenario.

Number of GPUs	Sub-domain edge length	Time for 300 halo exchanges (s)	Estimated bandwidth (GB/s) (Global)
4	2560	0.055	0.84
9	1706	0.051	0.90
16	1280	0.056	0.82
25	1024	0.048	0.96
36	853	0.053	0.86

Table 18: Estimated halo exchange bandwidth scaling in a strong scaling scenario.

Number of GPUs	Sub-domain edge length	Time for 300 halo exchanges (s)	Estimated bandwidth (GB/s) (Global)
4	5120	0.079	1.16
9	5120	0.096	1.42
16	5120	0.123	1.49
25	5120	0.136	1.69
36	5120	0.110	2.50

Table 19: Estimated halo exchange bandwidth scaling in a weak scaling scenario.

These halo exchange times and corresponding data bandwidths strongly suggest that the halo exchanges are not a cause of reduced scaling efficiency. Furthermore, this indicates that the GCL library is performing these halo exchanges efficiently.

7 Time comparison between schemes

To compare the runtime of the three schemes we created a test problem using the initial condition defined in (5.1) to (5.4) on a domain with side length $n = 5120$. Solver tolerances were set using the results of section 3.2 at 10^{-1} for the mass matrix CG solvers and for the Inexact Newton exit tolerance, with the corresponding Shifted Laplace CG solver using 10^{-2} as an exit tolerance. Each scheme was then run to compute a solution at $t = 0.01$ using varying stable time step sizes defined by setting a CFL number, with the results in the table below. The listed iterations are the maximum recorded number of iterations over all computed time steps.

Scheme	CFL number	Number of time steps	Time taken (s)	Number of CG iterations
Leapfrog	0.1	512	33.83	3
	0.2	256	16.91	3
RK3	0.1	512	133.44	3
	0.2	256	66.72	3
	0.3	171	44.51	3
	0.4	128	31.01	3

Scheme	CFL number	Number of time steps	Time taken (s)	Number of Inexact Newton iterations	Number of CG iterations per IN iteration
Semi-Implicit	0.1	512	48.51	3	2
	0.2	256	24.25	3	2
	0.3	171	16.14	3	2
	0.4	128	12.31	4	3
	0.6	86	8.68	5	4
	0.8	64	6.64	5	4
	1.6	32	3.80	7	6

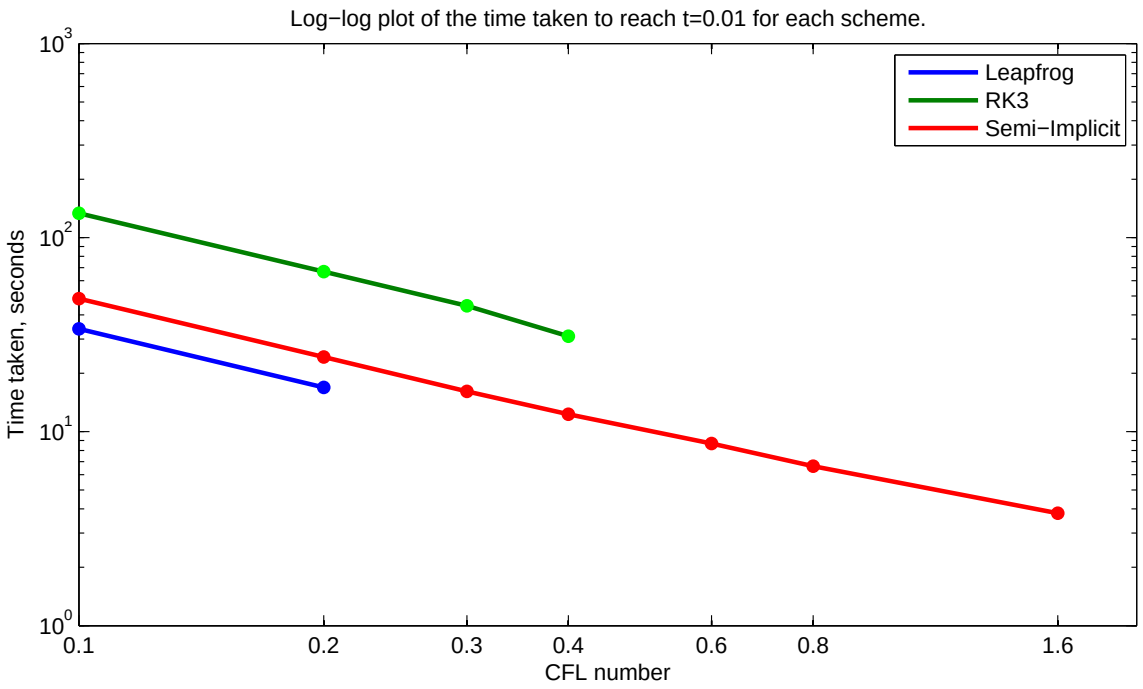

Figure 23: Time taken by each time stepping scheme to compute a solution at $t = 0.01$ on a domain side length $n = 5120$ using 4 GPUS, plotted on a log-log scale.

As expected the more expensive RK3 method has a longer runtime than that of the cheaper Leapfrog and Semi-Implicit schemes. However as discussed earlier, the RK3 scheme should deliver comparable accuracy to the Leapfrog and Semi-Implicit schemes using a larger time step. As demonstrated, the Semi-Implicit scheme remains stable for much larger time steps than both explicit methods with much shorter run times, but the computed solution may well be inaccurate. The investigation into how each method performs whilst meeting a desired accuracy is not covered in this report but could form part of future work.

8 Conclusions

The Semi-Implicit method as implemented on GPUs in this study scales extremely well in a weak scaling scenario, whilst also offering second order error convergence spatially and temporally. The strong scaling demonstrated is reasonable, but clearly the implementation will not scale well to hundreds of GPUs in a strong scaling scenario. In comparison to Leapfrog we demonstrated that the Semi-Implicit method exhibits the same order of accuracy whilst allowing larger time steps to be chosen, resulting in shorter overall computation times. Similarly, for a given time step size we demonstrate that the RK3 method is much more computationally expensive than the Semi-Implicit method. However, unlike the Semi-Implicit method, RK3 offers third order temporal error convergence.

Future work could investigate further into how the CG solver output error affects the error of the overall solution. An alternative implementation could apply a CG exit condition based upon a set number of iterations, as opposed to a relative tolerance. Furthermore, alternative elliptic

solvers such as Multigrid may yield higher performance than CG as a solver for the Shifted Laplace equation.

Section 7 of this report compares time stepping schemes through comparison of the time taken to compute a solution for a given time step size. A more sophisticated comparison should include not only the time taken, but also the achieved accuracy of the output solution.

References

[1] Globally Uniform Next Generation Highly Optimized `https://puma.nerc.ac.uk/trac/GungHo`

[2] Eike Müller, Robert Scheichl
Massively parallel solvers for elliptic PDEs in Numerical Weather- and Climate Prediction
arXiv:1307.2036

[3] Eike Hermann Müller, Robert Scheichl, Benson Muite, and Eero Vainikko
Petascale elliptic solvers for anisotropic PDEs on GPU clusters
arXiv:1402.3545

[4] NVIDIA Home page
`www.nvidia.co.uk`

[5] Compute Unified Device Architecture
NVIDIA CUDA Home Page
`http://www.nvidia.com/object/cuda_home_new.html`

[6] Khronos Group OpenCL
`https://www.khronos.org/opencl/`

[7] NVIDIA Tesla M2090 specifications
`http://www.nvidia.com/docs/IO/43395/tesla-m2090-board-specifications.pdf`

[8] NVIDIAs Next Generation CUDA Compute Architecture: Fermi
Whitepaper
`http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`

[9] Emerald
e-Infrastructure South GPU supercomputer
`http://www.einfrastructuresouth.ac.uk/cfi/emerald`

[10] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, Second addition with corrections, January 2000, page 178.

[11] NVIDIA GPUDirect
`https://developer.nvidia.com/gpudirect`

[12] Arakawa, A.
Lamb, V.R.
"Computational design of the basic dynamical processes of the UCLA general circulation model", *Volume 17 of Methods in computational physics*, page 181

[13] Generic Communication Library (GCL)
Mauro Bianco, Ugo Varetto
Swiss National Supercomputing Centre (CSCS).
Mauro Bianco,
An Interface for Halo Exchange Pattern

[14] NVIDIA cuBLAS
`https://developer.nvidia.com/cublas`

[15] Eike Müller, Xu Guo, Robert Scheichl, Sinan Shi
Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs
arXiv:1302.7193

[16] W R Saunders
Explicit GPU solver for the unforced non-linear shallow water equations without dissipation

[17] NVIDIA Visual Profiler
`https://developer.nvidia.com/nvidia-visual-profiler`