Simulating and Benchmarking the Shallow-Water Fluid Dynamical Equations on Multiple Graphical Processing Units

D.P. Playne  K.A. Hawick  M.G.B. Johnson

Abstract

The shallow-water model equations provide a simple yet realistic benchmark problem in computational fluid dynamics (CFD) that can be implemented on a variety of computational platforms. Graphical Processing Units can be used to accelerate such problems either singly using a data parallel decompositional scheme or with multiple devices using a domain decompositional approach. We implement the SW equations on a range of modern GPUs with both parallel schemes and report on the typical performance. We compare integer optimised GPUs and very modern floating-point intensive GPU devices such as NVidia’s Kepler K20X, and also investigate different m-GPU communication methods for geometric decompositions. We give detailed performance results and a summary of the main parallelisation issues.

Keywords: shallow water system; weather simulation; climate simulation; parallel computing; single GPU; multiple GPU.

1 Introduction

Computational Fluid dynamical algorithms(Tritton 1988, Griebel, Dornseifer & Neunhoeffer 1998) remain the central core needed for both numerical weather prediction(Barry & Chorley 1989) and climate simulations(Hurrell, Meehl, Bader, Delworth, Kirtman & Wielicki 2009). These problems have been central to high performance computing since its inception with the ENIAC and other supercomputers in the 1950s(Dyson 2012). Modern weather forecast codes and climate analysis codes have become very sophisticated in modern times, often running ensembles of systems to attain appropriate statistical accuracy. Nevertheless the core of such codes is still a numerical integration of the atmosphere as a fluid and sometimes also a coupled ocean model system. In both cases the system is treated as a fluid with coupled cells of material each approximating a spatial region of air or ocean(C.A.J.Fletcher 1991a, C.A.J.Fletcher 1991b). A great many different numerical schemes have been employed in such codes and it is quite difficult to implement a complete such package as a benchmark(Bailey, Barszcz, Barton, Browning, Carter, Dagum, Fatoohi, Fineberg, Fredericks, Lasinski, Schreiber, Simon, Venkatakrishnan & Weeratunga 1994).

Figure 1: Visualisation of waves in a Shallow Water model simulation.

The shallow-water (SW) system of equations(Randall 2006) model a greatly simplified such problem and provide a more practicable benchmark algorithm and code that can be readily ported across different computational platforms, and most usefully for our work in this present paper, the SW equations also lend themselves to a range of different parallel decompositional schemes. As the name suggests, the SW system models a shallow fluid system that is somewhere between two and three dimensional. A few simplified fluid layers are used to approximate the bulk behaviour and simplify the calculations. Although in such simulations much of the details needed to study for example turbulence(W.D.McComb 1990) are lost, the SW model tends to work surprisingly well for thin-layer fluids(Hawick 2011) such as the planetary atmosphere or for certain shallow oceanic problems such as particular bays(Bailey 2010, Martinez, Campbell, Annable & Kiker 2008). Realistic short to medium range weather models make intensive use of initial condition values for the numerical integration(Hawick 1991) based on observational data form many sources such as satellites(Xiao, Zou, Pudeca, Shapiro & Velden 2002). Climate codes are more governed by statistical energy(Peixoto & H.Oort 1984) and system-wide coupling effects and exhibit less memory concerning specific initial condition details. Benchmarking and optimising the performance of the data assimilation scheme(Hawick, Bell, Dickinson, Surry & Wylie 1992) is a separable problem however and we do not tackle it in this present paper. Instead we focus on the computational integration(Playne & Hawick 2011) as typified by the needs of a model such as the SW system.

Although sophisticated spectral techniques(Barros...
& Kauranne 1991, Tett 1991) are increasingly being applied to weather and climate problems, the classic parallel strategy for computational fluid dynamics (Hawick, Bogucz, Degani, Fox & Robinson 1995) is that of a geometric data parallelism (Hawick & Playne 2011), whereby the mesh points of the equations are spread geometrically across a set of processors or processing cores. This strategy is ideal for a single Graphical Processing Unit (GPU) and modern GPUs (Leist, Playne & Hawick 2009) have an impressively large number of such cores. However not all such devices have the same degree of support for floating point calculations (Johnson, Playne & Hawick 2013), and some devices that have many cores have rather poorer FPU support of those cores. There is therefore an important tradeoff space of FPU support and number of cores and of course cost to be considered.

Additionally, schemes like the SW model which involves multiple stages of advection also offer a task decompositional option (Vu, Cats & Wolters 2008) whereby multiple GPUs are used to accelerate the same CPU host processor in tackling different parts of the numerical update. This is a particularly interesting strategy that can make use of the modern GPU capability of communicating data with one another directly without passing communications through the hosting CPU.

Modern multi-core processors and GPUs lend themselves well to computationally intensive calculations of this sort (Schmidt, Berzins, Thornock, Saad & Sutherland 2013). Many of the current Top 500 supercomputers in the world use GPUs to accelerate individual nodes. The regime of multiple GPU accelerators per node is however still not well explored. Computational Fluid Dynamics remains a highly significant problem for real supercomputer installations (Zaspel & Griebel 2012). CFD problems have also been formulated as lattice-gas and automaton model systems (Johnson, Playne & Hawick 2010, Lyes, Johnson & Hawick 2012). This approach can make use of integer-optimised devices but in this paper we focus on traditional CFD solutions using the solution of partial differential equations by numerical integration (Micikevicius 2009) and floating-point intensive processing elements.

The SW system itself has been studied before for many parallel platforms including a single GPU accelerator per CPU (Vinas, Lobeiras, Fraguela, Arenaz, Amor, Garcia, Castro & Doallo 2013). We however are able to consider the tradeoff space for very recent GPU models and also for multiple GPUs including the GPU-GPU direct data communication capability. We make use of NVIDIA’s Compute Unified Device Architecture (CUDA) software (NVI 2012) in this present work. CUDA software can be run on a large range of different GPU models and we report on a selection of these along with discussion of the implications for CFD simulation problems.

Our article is structured as follows: In Section 2 we summarise the Shallow Water model formulation that we investigate. We describe our parallelisation strategy for single and multiple GPUs in Section 4 and give some selected performance results for various devices in Section 5. We give a discussion of the results and offer some tentative conclusions in Section 6.

## 2 Shallow Water Model Formulation

The Shallow Water equations (SW) are a set of partial differential equations that can be derived from the more general Navier-Stokes equations. In the SW the horizontal length scale is considered to be much greater than the vertical length scale and a hydrostatic pressure along the direction of gravity. The vertical velocity can be removed from the Navier-Stokes equations by integrating vertically which allows a three-dimensional fluid problem to be turned into a two-dimensional height-field problem (Muller, Stamm, James & Thurey 2008).

The SW describe two conditions of the conservation of mass and the conservation of momentum. This is expressed as an advection-diffusion problem.

A shallow-water system can be represented by a fluid height- and velocity-field.

\[
\frac{\partial h}{\partial t} = -h \nabla \cdot \mathbf{v} - (\nabla h) \mathbf{v} \quad (1)
\]

\[
\frac{\partial \mathbf{v}}{\partial t} = g \nabla h - (\nabla \mathbf{v}) \mathbf{v} \quad (2)
\]

The fluid in this model is stored as a staggered grid where the height variable \( h \) represents the height of the fluid at the centre of each discrete cell. The horizontal velocity \( \mathbf{v} \) is represented by two variables \( \mathbf{v}_x \) and \( \mathbf{v}_y \) which are the velocities on the edges of the cell in the \( x \)- and \( y \)-dimensions respectively. This staggered grid is shown in Figure 2 where the dots represent the location of the height variable and the lines represent the horizontal velocities in the \( x \)- and \( y \)-dimensions. Staggered grids are commonly used for fluid simulations as they avoid some of the instabilities caused by a discrete co-located grid.

![Staggered grid used to store the shallow water system.](image)

Figure 2: Staggered grid used to store the shallow water system. The dots represent the location of the height field points and while the vertical and horizontal bars represent the velocity field points.

The method of simulating the SW used in this project is based on the description in (Muller et al. 2008). This method uses an explicit time-integration scheme to avoid the problem of solving a system of linear equations. Updating the fluid involves two separate stages - first the fields are advected using the velocity field and then the accelerations of the fields are computed. This process is shown in Algorithm 1.

A semi-Lagrangian method is used to compute the advection steps within the field. Essentially this involves performing a backwards trace of an imaginary particle at each lattice point. This must be performed separately for each of the lattice points representing the height and velocity fields. For each lattice point the velocity field at that point is calculated. This may
Algorithm 1 Update algorithm for the SW model.

for All steps do
    advect(h, v)
    advect(vx, v)
    advect(vy, v)
    update(h, v)
    update(vx, h)
    update(vy, h)
end for

require interpolation of velocity variables due to the staggered grid. This is illustrated in Figure 3.

Figure 3: Interpolation of the velocity field variables for the three different lattice points - height (left), vx (centre) and vy (right).

Once the appropriate velocity values have been interpolated to give the velocity field at the desired point in the grid, the velocity is used to trace an imaginary particle backwards in time to advect the field. Once the particle has been traced backwards in time (see Equation 3), the value of the field must be calculated at the exact point. This is performed using bi-linear interpolation which is bounded by the existing values of the field and helps to ensure stability.

\[ x_{t-\Delta t} = x - v\Delta t \]  

After the advection steps have been completed the fields are updated. The update of the height field depends the divergence of the velocity fields and the acceleration of the velocity fields is given by the gradient of the fluid height (above 0). This total fluid height depends on both the fluid height and the height of the ground to correctly handle non-constant ground planes.

The border conditions of the simulation are simple reflecting boundaries. These boundaries can be implemented by enforcing the condition that the velocity across the border is zero (fluid cannot flow through the wall) and mirroring the height of the fluid across the boundary. The velocity along the direction of the wall is unaffected by the boundary conditions and the fluid can flow along the wall with no friction.

This is a brief outline of the updating process of a simple shallow-water simulation based on the description in (Muller et al. 2008). This article considers how such a simulation can be implemented on an m-GPU (multi-GPU) system.

3 m-GPU Technology

Released late last year the Nvidia Tesla K20X is the most powerful single-GPU released to date. This Kepler architecture GPU contains 2688 CUDA cores, 6GB of GDDR ram and supports Dynamic Parallelism and Hyper-Q. With a peak double precision floating point performance of 1.31 TFLOP/s it is over twice as fast as the previous generation Tesla M2090 (666.1 TFLOP/s) despite only having an additional 10 watts TDP.

The GPU in this device is the Kepler GK110 which contains a number of NVIDIA’s new generation Streaming Multiprocessor (SMX) units shown in Figure 4. These SMX units contain a number of improvements over those in the GK104, most significant for scientific applications is the additional double precision processing units. For reference the architecture of the previous generation Streaming Multiprocessor (SM) units in a Fermi GPU are shown in Figure 5.
There are two competitive options for data transfer between devices in an m-GPU program. These are Direct Transfer and Direct Access which are peer-to-peer data transfer methods available with GPU Direct 2.0. These two methods require two GPUs of Fermi architecture or later with peer-to-peer memory access enabled and both rely on Unified Virtual Addressing (UVA). UVA maintains a single address space for all memory allocated on the host and on any GPU devices. This allows any memory access or copy to identify the memory location of a memory address whether it is on the host or on a GPU.

Direct Transfer allows the host to initiate a memory copy from a source the device memory of one GPU to a destination in another device. A Direct Transfer memory copy is faster than the old method of memory transfer between devices which had to copy the data into an area of host memory and then copy it into the second device. Direct Transfer eliminates this unnecessary additional memory transfer and the need for a host memory buffer. This method of device-device communication is shown in Figure 6.

![Figure 6: Direct Transfer method of communication where data is copied directly from the memory of one device to another.](image)

Direct Access eliminates the need for host initiated memory copies altogether by allowing threads from one GPU to access memory on another GPU. This Non-Uniform Memory Access (NUMA) is expected to be less efficient for transferring large contiguous blocks of data between two GPU devices but can have advantages for the transfer of border information as the memory copies can be removed entirely. This method of data access is shown in Figure 7.

![Figure 7: Direct Access method of communication where data stored in one device is read directly by a thread from another device.](image)

Unified Virtual Addressing (UVA) defines a single address space in which all memory addresses are defined for both the CPU and GPU memory. This allows the physical location of a memory address to be identified by the system. This feature is required for Direct Access as if there was not a single address space then the threads would not be able to identify which device’s memory a value should be read from. Direct Transfer does not require UVA but it does simplify the cudaMemcpy calls for a Direct Transfer copy as the programmer can simply specify two addresses and the system can work out where those memory addresses are stored. Previously the programmer had to use cudaMemcpyPeer, specify the source and destination addresses as well as the GPU devices they were stored on.

4 GPU Implementation Method

Domain decomposition is used to split the simulation of the Shallow Water model across multiple GPU devices. In the case described here where the system is split between two GPU devices, each device is allocated half of the system. The devices are responsible for storing and updating their half of the system. The system is split in the y-dimension because then the bordering cells in the lattice are stored in contiguous memory. It should be noted that splitting the simulation across a different number of devices may have other optimal decomposition methods, the optimal method to used will depend on the number of devices and is not discussed further here.

During the updating of the system, the data on the borders of each device’s domain must be exchanged in order for the simulation to be correct. This exchange of bordering information must be performed several times during the update (see Algorithm 1). The advection of the fields is performed using the current velocity field (Muller et al. 2008) and the new values computed for the borders of each domain must exchanged. This exchange must also be performed after the update of the height field (as it is required for the update of the velocities) and finally after the velocities have been updated. The bordering areas of the two lattice domains are shown in Figure 8.

![Figure 8: Bordering information that must be exchanged between the GPU devices.](image)

**Direct Transfer Implementation**

The first implementation of the m-GPU shallow water model uses the Direct Transfer method to exchange bordering information. Performing a direct memory copy from one device to another is simple in the more recent version of CUDA. All that is required is a simple cudaMemcpy call using the cudaMemcpyDefault flag. Because the memory addresses are defined with Unified Virtual Addressing (UVA), CUDA automatically works out where the source and destination memory areas are stored and copies the data between them. If these two areas are in the device memory of two different GPUs it will automatically use the Direct Transfer method.

Listing 1 shows a code snippet of the advection stage of the SW simulation. The three fields h, vx and vy are advected using the current velocity field by each GPU device. The devices are synchronised to ensure that the computation is completed before the bordering information is exchanged which is then performed by a series of call to cudaMemcpy.
Listing 1: Direct Transfer Implementation of the advection stage of the simulation. The three fields of the simulation are numbered to show which device they belong to - e.g. h1 is the height field on device 1, h2 is the height field on device 2.

```c
void advectH <<< (vx1, vy1, h1, ...);
advectVX <<< ... (vx2, vy2, h2, ...);
advectVY <<< ... (vx2, vy2, ...);
```

```c
// Synchronize threads
cudaSetDevice (0);
cudaThreadSynchronize ();
cudaSetDevice (1);
cudaThreadSynchronize ();
```

The disadvantage of the Direct Transfer method is that the GPU devices will be idle during the memory transfer. Unlike the copy through host, CUDA does not currently support concurrent computation and communication for peer-to-peer memory transfer.

Direct Access Implementation

The Direct Access method has no need for any calls to cudaMemcpy as any kernel that requires data stored in the other GPU device’s memory can simply read it. This does require each thread to have pointers to the fields on both GPUs. Using UVA, CUDA can address comparison only needs to be performed when neighbouring values are read as a thread will never be launched on one GPU to process a cell stored on the other device.

This is implemented in all of the kernels but the height advection kernel is shown as an example in Listing 2. This kernel is the same for both GPU devices and simple updates on half of the field as determined by y and y_max. If any address has a y index of less than Y2 the thread will read the value from the array on the first device, if it is greater than or equal to Y2 it will be read from the second device. This address comparison only needs to be performed when neighbouring values are read as a thread will never be launched on one GPU to process a cell stored on the other device.

```c
// Advection
cudaSetDevice (0);
advectH <<< ... (vx1, vy1, h1, ...);
advectVX <<< ... (vx2, vy2, h2, ...);
advectVY <<< ... (vx2, vy2, ...);
cudaSetDevice (1);
```

```c
// Exchange H
cudaMemcpy(&h1 [Y2xX], &h2 [Y2xX], X * sizeof (real), cudaMemcpyDefault);
cudaMemcpy(&h2 [Y2-1]x], &h1 [Y2-1]x], X * sizeof (real), cudaMemcpyDefault);
```

```c
// Exchange VX
cudaMemcpy(&vx1 [Y2xX], &vx2 [Y2xX], X * sizeof (real), cudaMemcpyDefault);
cudaMemcpy(&vx2 [Y2-1]x], &vx1 [Y2-1]x], X * sizeof (real), cudaMemcpyDefault);
```

```c
// Exchange VY
cudaMemcpy(&vy1 [Y2xX], &vy2 [Y2xX], X * sizeof (real), cudaMemcpyDefault);
cudaMemcpy(&vy2 [Y2-1]x], &vy1 [Y2-1]x], X * sizeof (real), cudaMemcpyDefault);
```

```c
// Synchronize threads
cudaSetDevice (0);
cudaThreadSynchronize ();
cudaSetDevice (1);
cudaThreadSynchronize ();
```

```c
// Exchange H
cudaMemcpy(&h1 [Y2xX], &h2 [Y2xX], X * sizeof (real), cudaMemcpyDefault);
cudaMemcpy(&h2 [Y2-1]x], &h1 [Y2-1]x], X * sizeof (real), cudaMemcpyDefault);
```

```c
// Exchange VX
cudaMemcpy(&vx1 [Y2xX], &vx2 [Y2xX], X * sizeof (real), cudaMemcpyDefault);
cudaMemcpy(&vx2 [Y2-1]x], &vx1 [Y2-1]x], X * sizeof (real), cudaMemcpyDefault);
```

```c
// Exchange VY
cudaMemcpy(&vy1 [Y2xX], &vy2 [Y2xX], X * sizeof (real), cudaMemcpyDefault);
cudaMemcpy(&vy2 [Y2-1]x], &vy1 [Y2-1]x], X * sizeof (real), cudaMemcpyDefault);
```

The performance of both these implementations are compared to each other and to a single-GPU implementation of the same simulation.

5 Performance Results

These two m-GPU implementations have been tested on several different m-GPU systems with different generations of GPU device. The GPUs tested are the Fermi architecture GeForce GTX590 and the Kepler architecture GTX690 and two GTX780s. In addition to these GeForce cards the simulations are tested on two Kepler architecture Teslacompute card K20Xs. These systems are running CUDA version 5.5 which supports NVIDIA GPU/Direct 2.0, the technology that provides the Direct Transfer and Direct Access methods.

The implementations have been tested for a range of system sizes from 64² to 8192² (GPU device memory allowing). These experiments compare both the computational performance of the different GPU devices as well as their support for device-device com-
Figure 9: Performance results of the m-GPU shallow water simulations for system sizes from $64^2$ to $8192^2$ on four m-GPU systems - GTX590, GTX690, GTX780 and K20X.

Figure 10: Performance results of the m-GPU shallow water simulations using Direct Access (DA) and Direct Transfer (DT) on 2x K20Xs compared with a single GPU simulation on a K20X. The results are shown in ln-ln scale.
munication. It is worth noting that the Tesla K20X and the GeForce GTX780 devices must communicate through the PCIe bus as the GPUs are in two separate cards while the GTX590 and GTX690 are both dual-GPU cards.

It can be seen from these results that (as is expected) the Tesla K20X compute cards easily provide the best performance. In fact the SW simulations running on a single K20X are only slightly slower than the same simulation running on two GTX780 devices.

One unusual result is the performance of the GTX590 devices which outperform the GTX690 and compete with the GTX Titan systems in both the single- and multi-GPU implementation. Given the nature of the floating-point based computation of the SW model it is not unexpected that the GTX590 may outperform the GTX690. The Fermi architecture GPUs have a significantly different architecture with more multiprocessors but only 32 cores per multiprocessor. In previous research it has been found that Fermi-architecture GPUs can outperform equivalence Kepler architecture GPUs for some floating-point processing problems (Johnson et al. 2013).

The performance results of the single-GPU and two m-GPU implementations of the SW model running on two K20X devices are shown in Figure 10. On this scale it is easier to see the difference between the implementations across the different system sizes. Similar relative performance is observed for all the m-GPU systems tested so only the results for the K20X are presented to make the graph easier to read.

Figure 10 show that the single-GPU implementation is the fastest for very small simulations ($64^2$ - $192^2$) as these small systems cannot use the full computational power of a single-GPU let alone two. As the system sizes increases the speedup from using two GPU devices approaches the optimal 2x.

The Direct Access implementation provides the best performance of the two m-GPU implementations and provides a noticeable speedup over the Direct Transfer method from 2x for system size of $64^2$ to 1.2x at $1024^2$. For systems larger this the performance of the two implementations becomes almost indistinguishable but the Direct Access method never imposes a performance penalty.

Table 1: Current prices (estimated from 2013 online information) of the evaluated graphics cards and their relative performance.

<table>
<thead>
<tr>
<th>Model</th>
<th>Price (NZD)</th>
<th>Speedup (vs 1x GTX590)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX590</td>
<td>$\approx$ 1,200</td>
<td>1x</td>
</tr>
<tr>
<td>GTX590 (2x GPU)</td>
<td>$\approx$ 1,200</td>
<td>1.92x</td>
</tr>
<tr>
<td>GTX690</td>
<td>$\approx$ 1,400</td>
<td>0.74x</td>
</tr>
<tr>
<td>GTX690 (2x GPU)</td>
<td>$\approx$ 1,400</td>
<td>1.40x</td>
</tr>
<tr>
<td>GTX Titan</td>
<td>$\approx$ 1,800</td>
<td>0.98x</td>
</tr>
<tr>
<td>GTX Titan x2</td>
<td>$\approx$ 3,600</td>
<td>1.86x</td>
</tr>
<tr>
<td>K20X</td>
<td>$\approx$ 4,500</td>
<td>1.71x</td>
</tr>
<tr>
<td>K20X x2</td>
<td>$\approx$ 9,000</td>
<td>3.38x</td>
</tr>
</tbody>
</table>

6 Discussion & Conclusions

We have described how a simulation of a shallow-water model can be implemented on a multi-GPU system using domain decomposition and two methods device-device communication. These implementations have been tested on several different GPU architectures to compare performance across recent devices.

The comparison of the different generation GPU devices has revealed the unexpected result that the older generation Fermi-architecture GTX590 can offer better performance than the newer Kepler-architecture equivalent card, the GTX690. Although the GTX690 contains many more cores than the GTX590, the configuration of the Kepler GK104 SMX units are not well suited to the memory and floating-point intensive kernels required by the shallow water model. This result was shown in both the single- and multi-GPU implementations. Given the relatively low price of the GTX 590 it shows that more expensive cards do not necessarily provide better CUDA performance.

Another surprising result of these experiments is the measured difference in performance between the GTX Titan and the K20X. Both of these cards should contain the same GK110 GPUs yet show a significant difference in performance. There is no clear reason for the performance difference as this simulation does not make use of the extra features enabled only in the K20X.

All the m-GPU capabilities devices tested show similar scaling behaviour with the Direct Access method providing a performance benefit for smaller
systems over the Direct Transfer method. The performance of these two implementations converges as the system size grows beyond 1024. At these larger system sizes the performance of both methods approaches the optimal 2x speedup over the single-GPU implementation.

Based on these findings we conclude that both methods are suitable for m-GPU implementations of this type of simulation using domain decomposition as they both offer similar performance for the large systems sizes m-GPU implementations would usually be used for. However, as the Direct Access method offers better performance for smaller systems and does not add significantly to the code complexity we find it to be the better method of device-device communication for this type of problem on both Fermi- and Kepler-architecture GPU devices.

Future work includes investigation of the performance difference between the GTX Titan and the K20X and an extension of this work to more GPU devices hosted in the same machine as well as GPU-accelerated compute clusters.

References


Muller, M., Stam, J., James, D. & Thurey, N. (2008), Real time physics class notes, Technical report, NVIDIA. Chapter 11 - Shallow Water Equations.


URL: http://onlinelibrary.wiley.com/doi


URL: http://journals.ametsoc.org/toc/mwre/130/3