Shallow Water Equations Implemented on GPUs

M. Abouali and Jose E. Castillo

March 16, 2014

Publication Number: CSRCR2014-03

Computational Science & Engineering Faculty and Students Research Articles

Database Powered by the Computational Science Research Center Computing Group & Visualization Lab

COMPUTATIONAL SCIENCE & ENGINEERING

SAN DIEGO STATE UNIVERSITY

Computational Science Research Center
College of Sciences
5500 Campanile Drive
San Diego, CA 92182-1245
(619) 594-3430

Shallow Water Equations Implemented on GPUs

M. Abouali *[1] and Jose E. Castillo[2]

[1]Ph.D. Candidate at San Diego State University
[2]Director, Computational Science Research Center, San Diego State University

March 16, 2014

1 Introduction

Shallow Water Equations (SWE) are obtained by integrating the Navier-Stokes' equation in the vertical direction. The use of "shallow" in the name can be misleading, as these equations can be used in simulating both deep and shallow regions; in fact, "shallow" relates only to the scale. In general, the vertical scale of the motion in atmospheric and oceanic flows is much smaller than the horizontal scale of the motion; therefore, they have a much smaller length scale in the vertical than in the horizontal. Hence, the "shallow" in the name: the vertical scale of the motion is much smaller than the horizontal scale of the motion. It should be noted that, as the flow depth increases, a smaller time scale must be used to keep the model stable.

1.1 Continuity Equation

The first SWE is the continuity equation, which is written as follows:

$$\frac{\partial h}{\partial t} + \frac{\partial hu}{\partial x} + \frac{\partial hv}{\partial t} = S_i - S_o, \tag{1}$$

where u is the eastward velocity component, or the velocity component along the x-axis; v is the northward velocity component, or the velocity component along the y-axis; h is the water depth; S_i is the source term; and S_o is the sink term.

Currently, we do not support a wetting and drying scheme; however, in order to facilitate the addition of such a scheme, the continuity equation should be written in terms of the water table height. In this case, the bathymatry, B, and the water table height, w, must both be measured from the same reference level. We employed the Geoid reference level, which is typically chosen, see Figure 1 for a more detailed explanation of these variables. Here, $h = w - B$, with the continuity equation rewritten as follows:

$$\frac{\partial w}{\partial t} + \frac{\partial hu}{\partial x} + \frac{\partial hv}{\partial t} = S_i - S_o + \frac{\partial B}{\partial t}, \tag{2}$$

where, if the morphological changes in the bottom floor are ignored, the continuity equation can then be simplified to:

$$\frac{\partial w}{\partial t} + \frac{\partial hu}{\partial x} + \frac{\partial hv}{\partial t} = S_i - S_o. \tag{3}$$

1.2 Momentum Equations

The u-momentum equation in its conservative form can be written as follows:

$$\frac{\partial (hu)}{\partial t} + \frac{\partial hu^2 + 0.5gh^2}{\partial x} + \frac{\partial huv}{\partial y} = -gh\frac{\partial B}{\partial x} - \frac{gu\sqrt{u^2+v^2}}{C_z^2} + \tau_u^w, \tag{4}$$

*maboualiedu@gmail.com

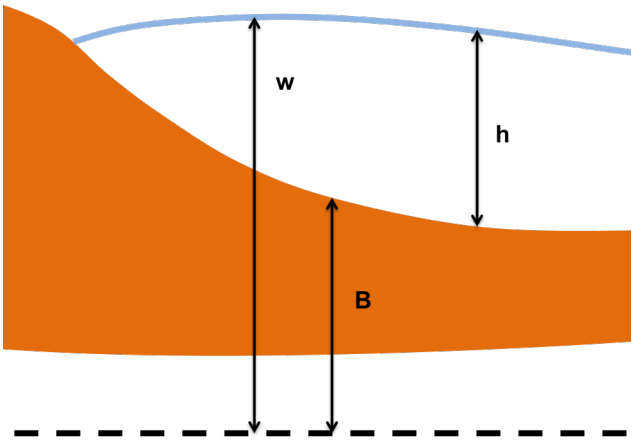


Figure 1: Shallow Water Equation (SWE) variables in vertical profile.

where $g = 9.81\ [ms^{-2}]$ is the Earth's gravitational acceleration constant, C_z is the Chezy constant, and τ_u^w is the forcing resulting from the wind stress. Likewise, the v-momentum equation can be written as follows:

$$\frac{\partial(hv)}{\partial t} + \frac{\partial huv}{\partial x} + \frac{\partial hv^2 + 0.5gh^2}{\partial y} = -gh\frac{\partial B}{\partial y} - \frac{gv\sqrt{u^2+v^2}}{C_z^2} + \tau_v^w. \tag{5}$$

Note that the momentum equation is written in its conservative form. All variables are defined in the physical domain; however, the calculation is done in the computational domain. The standard approach is to transform all velocity components in the computational domain, and rewrite the SWE equation using only the divergence and gradient defined on the computational domain. However, this would produce numerical errors and compromise the conservation in conventional curvilinear grids Ferziger & Millovan (2001). Thus, to reduce this error, all the variables are kept in the physical domain and the operators are transformed internally.

1.3 De-singularizing

Note that both momentum equations compute the changes in hu and hv; however, to obtain u and v, we must first divide them by h, i.e., the water depth. This can result in h becoming a very small number, and, due to finite precision available on computers, this division can be erroneous. To avoid such situations, the water depth must first be compared to a user-defined d_{crit}. If $h > d_{crit}$,then:

$$(u, v) = (hu, hv)/h, \tag{6}$$

otherwise, i.e. $h <= d_{crit}$; the velocity components are calculated as follows:

$$(u, v) = (hu, hv) \cdot \left(\frac{\sqrt{2}h}{\sqrt{h^4 + max(h^4, K)}} \right). \tag{7}$$

K is also a user defined value. The choice of K has a significant impact on both the performance and accuracy. If it is too large, it will dampen the velocity and, and if it is too small, a smaller time step will be required to keep the scheme stable. Kurganov and Petrova Kurganov *et al.* (2001); Kurganov & Petrova (2007) chose K, as follows:

$$K = max(\Delta x^4, \Delta y^4), \tag{8}$$

however, Brodtkorb Brodtkorb *et al.* (2012) proposed the following method to choose K:

$$K = K_0 max(1, min(\Delta x, \Delta y)), \tag{9}$$

where $K_0 = 10^{-2}$ for single precision calculations, and it is set to an even smaller value for double-precision calculations.

1.4 Bottom Roughness

The roughness of the surface can be provided in the model by defining the Chezy coefficient, C_z. The code reads in a 2D array of numbers for C_z; therefore, the user have the possibility to vary the roughness depending on the location. Sometimes, it is easier to use Manning coefficients. In case where Manning coefficients is available, the user can provide the Manning coefficient and the Chezy coefficient is calculated as follows:

$$C_z = \frac{h^{1/6}}{n}. \tag{10}$$

It is also possible to calculate Chezy coefficient using the White Colebrook equation, which is written as follows:

$$C_z = 18 \log_1 0 \left(\frac{12h}{K_s} \right), \tag{11}$$

where K_s is the Nikuradse roughness length. All these options are available to the user. The user also have this option to apply a constant bottom roughness trough out the field. To keep the performance of the code and avoid if-clauses in the code, which will compromise the performance, this must decided pre-compilation.

1.5 Numerical Scheme and Solution Strategy

Castillo-Grone's Mimetic operators are used to discretized the equations spatially and the RK3 scheme is used to discretize the system in time. There are two approaches to advance the equations. The first approach is to advance each equation one after each other. This would help to artificially decouple these three equations but it results to interleaving the variables in time. However, for better quality and accuracy, it was decided to solve all the three equations together and simultaneously.

commonly it is believed that using a C-Grid approach is a better choice for such equations Ezer & Mellor (2004). In C-Grid approach, also known as staggered grids, u component is stored at the middle of the edges that are perpendicular to x-axis, v is stored at the middle of the edges that are perpendicular to y-axis, and w is stored at the cell centers. This approach, i.e. C-Grid, is indeed providing many benefits, such as eliminating the need for interpolation, as long as the equations are written in non-conservative forms and the Coriolis terms are ignored. Once a curvilniear grid is used, one usually replaces (u, v) by $(\tilde{U}, \tilde{V})$ to be able to use the C-Grid. $\tilde{U}$ and $\tilde{V}$ are the transformed velocity in computational domain, whereas u and v are velocity component in the physical domain. As explained by Ferziger Ferziger & Millovan (2001), this approach compromises the conservative properties of the equation. Ferziger provides a very nice explanation on how C-Grids loose their benefits and attractiveness once the general curvilinear grids are used. He also explains why it is indeed even better to not transform the variables and use an A-Grid approach once general curvilinear is used. For the same reason, and the author past experience also shows that using the A-Grid in general curvilinear grids has many advantages over C-Gird, the least of all is that once A-Grid is used in general curvilinear grids, there are less interpolation needed, completely the opposite of the case once the Cartesian grids are in use. Therefore, it was decided to use an A-Grid approach where variables are defined at the cell centers.

The general strategy for solving the above mentioned equations can be summarized as follows:

1. Interpolate variable from the cell centers to the middle of the edges in order to calculate fluxes.
2. calculate the rhs for the two momentum equations and the continuity equations
3. advance the the variables
4. apply the boundary conditions

Note that the RK3 is used for the time integration, which has three steps. Therefore, in order to advance time from t_n to t_{n+1} one has to repeat the above steps three times. Since the grid is not changing the interpolant is precompute; therefore, this step is only a sparse matrix computation and is relatively very fast. It is made even faster once it is implemented on GPUs. All interpolants are generated using the $p_d = 3$, i.e. cubic interpolation.

2 GPU Implementations

Central Processing Units (CPUs) have come a long way since their inception: they have become extraordinarily stronger and more efficient with the passage of time. One factor that determines CPU strength is its clock speed. The higher the clock speed, the more computations can be performed by a CPU in a given amount of time. However, since 2005, there has been no improvement in the clock speed of CPUs. Due to current technological limitations, it is virtually impossible to increase the speed of CPUs. Therefore, to make stronger CPUs, companies are now fitting more and more CPUs on a single chip, known as multicore CPUs. This enables the user to harness multiple CPUs simultaneously, in order to carry out more computations at one time, termed "parallel computation" or "parallel processing".

There are two general approaches for parallel processing: in the first, all computing cores reside in a single machine, and each core has access to all the data; in the second, the computing cores are on separate machines or nodes, and must communicate with each other via network connections.

Graphics Processing Units (GPUs) were originally designed to offload portion of the computations from the CPU that are required for displaying graphics. Therefore, the CPU has more time to complete other tasks. Due to their different design it is possible to cluster many computing cores on one GPU. As a result, users were interested to perform all sort of computation on the GPUs and instead of merely graphics-related. The term General Purpose GPU (GPGPU) was coined; and the GPGPU has since become the center of the high performance computing (HPC) and many supercomputers around the word are equipped with these devices.

For example, an Intel i7 extreme edition processor costs almost $1100 and it provides 6 cores; while an NVIDIA GTX Titan GPU costs approximately the same and provides 2688 cores. Although it should be mentioned that cores on CPUs are much stronger than the cores on GPUs; however, the sheer number of cores on GPUs make their performance significantly better.

There is yet another important difference between the cores on CPUs and those on GPUs: CPU cores are designed based on Multiple Instruction Multiple Data (MIMD) architecture, while GPU cores feature the Single Instruction Multiple Threads (SIMT) approach. This means that CPU cores are able to act independently, i.e., while one core is evaluating a sine function another can simultaneously evaluate a cosine function; in the case of GPUs, all cores must evaluate the same function, though the data on each core can vary. For example: If half the cores on a given GPU are needed to evaluate a sine function, and the other half needed to evaluate a cosine function, one task must be completed before the other can begin, resulting in half of the cores remaining idle at any one time. This has a significant impact on GPU performance, and substantially increases the amount of time needed to perform a task. Despite this fact, GPU performance is still faster than that of a CPU. While coding the SWE solver for this thesis all of these items were taken into consideration, and the code written in such a way that most of the if-clauses were either removed, or were decided during pre-compilation.

An additional difference between CPUs and GPUs is the implementation of the cache memory. Older versions of the GPU did not contain cache memory; therefore, it was suggested that read-only fields be mapped as textures, which were then cached, and later provided a higher throughput of data. as a result, many fluid solver codes mapped the data field at time t_n as a texture field; however, with the advent of the newer Fermi and Kepler NVIDIA GPUs, which are equipped with cache memory, users are no longer required to do this.

Many studies have already established that a good written code on a GPU can easily outperform a highly optimized code on a CPU Abouali *et al.* (2013); therefore, the timing provided in this paper is only provided to illustrate that the GPU code developed here is also able to confirm past findings, and uses GPU resources in an optimum manner.

2.1 Preprocessing and The Input/Output file

All the inputs and required parameters are read from a single NetCDF file. The output is also stored in the same NetCDF file. As a result, NetCDF is required during compiling and linking.

3 Results

3.1 Timing CPU versus GPU

One of the problems associated with GPUs is the time needed to transfer or upload data from the host memory into the GPU's memory. The data must reside in the GPU's memory before it can perform any calculation, and the transfer between the host and device can be a time-consuming task. In order to optimize this transfer and reduce

it as much as possible, the user must make sure there are enough computations being done on the data being sent to the GPU, and refrain from downloading anything back if it is not needed.

The SWE code developed for this thesis requires transfer of data on only two occasions: during the initialization phase of the code, after everything has been read from the disk; and when the user has requested that the data be written to the disk.

The first part of the program to be tested is the interpolation. Since the grid being utilized is structured, we decided to fix the location of the participating points on the source grid to interpolate onto the destination grid. So, instead of storing the entire sparse matrix and the information on the rows and columns of the nonzero entries, we decided to store only the nonzero entries and compute the location based on the known cell index, i.e., $cell_i$ and $cell_j$. To do this, we first developed a special CUDA kernel to perform this matrix multiplication. To test how well this kernel worked, a grid with 200×200 cells was selected. Both the CPU and GPU version were timed; the timing on the CPU includes only the computation, i.e., the interpolation, while the timing on the GPU includes the time needed to transfer data to the GPU, perform the computation, and download the results. Note that it is not necessary to transfer data with every iteration. Therefore, we decided to perform multiple interpolations before downloading the results. Two test cases were designed. In the first case, 20 interpolations were performed before downloading the results; and in the second case, 40 interpolations were performed. The CPU version of the code required 78 seconds for the first case and 150 seconds for the seconds case; the time needed for the GPU and its speed is shown in Tables 1 and 2.

Table 1: Timing the interpolation kernel on GPU (20 interpolation before data transform).

GPU Timing										
Block Size	4×4	5×5	4×8	6×6	7×7	8×8	9×9	10×10	11×11	1×32
Configurable Shared Memory										
GTX 480	13.25	10.92	10.07	11.18	11.77	11.61	11	13.32	11.17	-
C2050	16.16	13.21	12.10	13.26	14.24	13.94	12.69	16.59	13.07	-
Predefined Shared Memory										
GTX 480	13.25	10.95	10.04	11.15	11.81	11.4	10.68	13.19	11.62	9.37
C2050	16.18	13.24	12.04	13.35	14.27	13.98	12.77	16.43	12.75	11.1
Speed up										
Block Size	4×4	5×5	4×8	6×6	7×7	8×8	9×9	10×10	11×11	1×32
Configurable Shared Memory										
GTX 480	5.9	7.1	7.7	7.0	6.6	6.7	7.1	5.9	7.0	-
C2050	4.8	5.9	6.4	5.9	5.5	5.6	6.1	4.7	6.0	-
Predefined Shared Memory										
GTX 480	5.9	7.1	7.8	7.0	6.6	6.8	7.3	5.9	6.7	8.3
C2050	4.8	5.9	6.5	5.8	5.5	5.6	6.1	4.7	6.1	7.0

Table 1 and Table 2 show that the speed up improved when there was more computing to be done. Moreover, using the predefined shared memory also improved the speed up. This suggests that speed up is a function of the requested configuration. It is possible to achieve higher speed up once the user decides to store the results less frequently.

To test the entire code, i.e. the SWE solver, a rotated domain with 205×205 nodes was generated. Both the CPU and GPU codes simulated one hour and the time needed to store the results was ignored. Table 3 shows the timing between the CPU and GPU.

All of the timing results mentioned above are the average of 10 cases. The code was run for 12 times. The maximum and minimum time was dropped out and the remaining 10 samples were averaged. As previously mentioned, many studies have already shown that the GPUs compute faster than CPUs. That's a known fact and it was reconfirmed here. Therefore, instead of focusing on how much faster they are, it is very common to just report how much data the GPU can process in a given time. From now on, instead of reporting the CPU time and the GPU time, we will only report how much faster than real time the GPU was able to process. The timing will also include the time needed to store the results on the storage device. However, to reduce the effect of time needed to store the results on disk, an asynchronous approach is used. To test these effects the same code simulated two hours of simulation with various frequencies of writing the outputs and the timings are shown in 4

Table 2: Timing the interpolation kernel on GPU (40 interpolation before data transform).

GPU Timing										
Block Size	4×4	5×5	4×8	6×6	7×7	8×8	9×9	10×10	11×11	1×32
Configurable Shared Memory										
GTX 480	22.8	18.23	16.57	18.89	20.35	20.13	18.98	23.81	19.3	-
C2050	28.54	22.8	20.69	23.02	25.6	24.98	21.76	29.95	24.1	-
Predefined Shared Memory										
GTX 480	22.78	18.29	16.51	18.72	20.24	19.57	17.51	23.12	20.46	15.15
C2050	28.51	22.85	20.55	23.15	25.46	24.79	21.59	29.6	24.03	18.71
Speed up										
Block Size	4×4	5×5	4×8	6×6	7×7	8×8	9×9	10×10	11×11	1×32
Configurable Shared Memory										
GTX 480	6.6	8.2	9.1	7.9	7.4	7.5	7.9	6.3	7.8	-
C2050	5.3	6.6	7.2	6.5	5.9	6.0	6.9	5.0	6.2	-
Predefined Shared Memory										
GTX 480	6.6	8.2	9.1	8.0	7.4	7.7	8.6	6.5	7.3	9.9
C2050	5.3	6.6	7.3	6.5	5.9	6.1	6.9	5.1	6.2	8.0

Table 3: Timing SWE solver.

Platform	Time [ms]	Speed Up
Intel i3	155843	-
GTX 480	14104	-

Table 4: Timing SWE Solver on GTX 480.

# Snapshots	Write Interval [m]	Total Time [ms]
120	1	28725
12	10	28256
8	15	28248
2	60	28229
1	120	28220

3.2 Validation

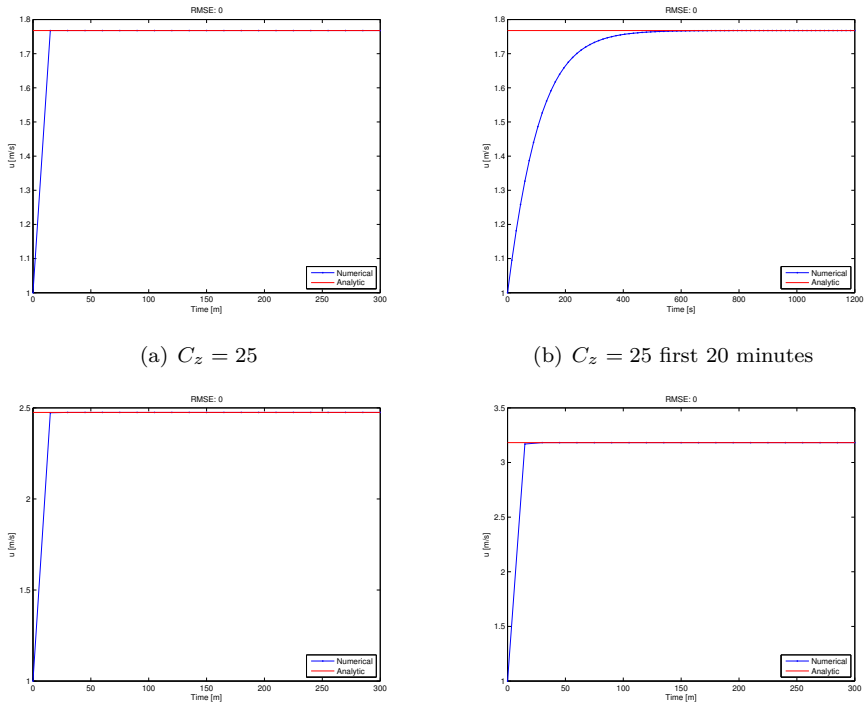
The first test in SWE is to check if the scheme is well-balanced. Fortunately there is an easy test for that. A channel with 3000m width and a length of 15000m was descritized using $dx = dy = 75m$. This will result into a grid with 45×205 nodes. The bottom bathymetry was set to have a slope of 0.001 along the x-axis, i.e. the length of the channel. Velocity was initialized to be zero, i.e. fluid is at rest at $t = 0$. The water table was set to be at the same height through out the entire domain. The periodic boundary condition was used everywhere. The code simulated 10 hours. Since there are no forcing, the water table is at the same elevation everywhere, and the fluid started at rest, there should not have been any flow generated. The model output was checked at the end of simulating 10 hours. As expected there was no flow generated anywhere and the water table appeared undisturbed. This shows that the different terms in the model balance each other.

To better assess the accuracy of the code and validate its results, another test was performed. The same channel as described above was used. However, in this second test, the water table was given the same slope as the bottom bathymetry, resulting to a constant depth of 5 meters through out the channel. The boundary condition is set to be periodic for u component of the velocity and the v component of the velocity is assumed to be zero at the boundaries. v was initalized to zero; however, u was initialized to be 1 ms^{-2}. The water table at the boundary is set in such a way that the water depth would be 5 meters at the boundaries. Various numerical, theoretical, and lab experiments have all confirmed that under such conditions the flow should reach a constant velocity determined

by:

$$u = C_z\sqrt{SD}, \tag{12}$$

where S is the bed-slope, D is the water depth, and C_z is the Chezy roughness parameter. Different C_z was tested. In all cases the model simulated 5 hours. Amazingly, it was seen that in all cases the code quickly reached the desired velocity without any oscillations, Figure 2. The error was zero. Figure 2(b) shows the first 20 minutes of the simulation with temporal resolution of 15 seconds, i.e. every 15 seconds one output is stored to the file. It can be seen that u component of velocity starts to increase to reach the level that it should be without any oscillations. In many other numerical models, during the same test many oscillations are seen and typically the velocity did not reach the exact desired value Wijbenga (1985a,b); Borthwick & Barber (1992). Moreover, in these studies it was seen that with the passage of time the deviation became slightly larger. The contour line for water table is shown in Figure 3.



(a) $C_z = 25$ (b) $C_z = 25$ first 20 minutes

(c) $C_z = 35$ (d) $C_z = 45$

Figure 2: Changes of error by increasing operator order

3.3 Rotated Domain

A domain with 205×205 nodes was generated. This grid was orthogonal; however, it was rotated by 45 degrees around the center. So that the velocity components does not align with the grid cells, Figure 4. Note that u is

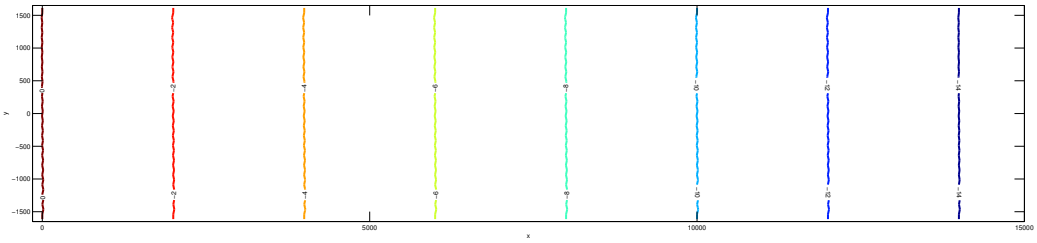


Figure 3: Contour lines for the water table in Uniform flow test.

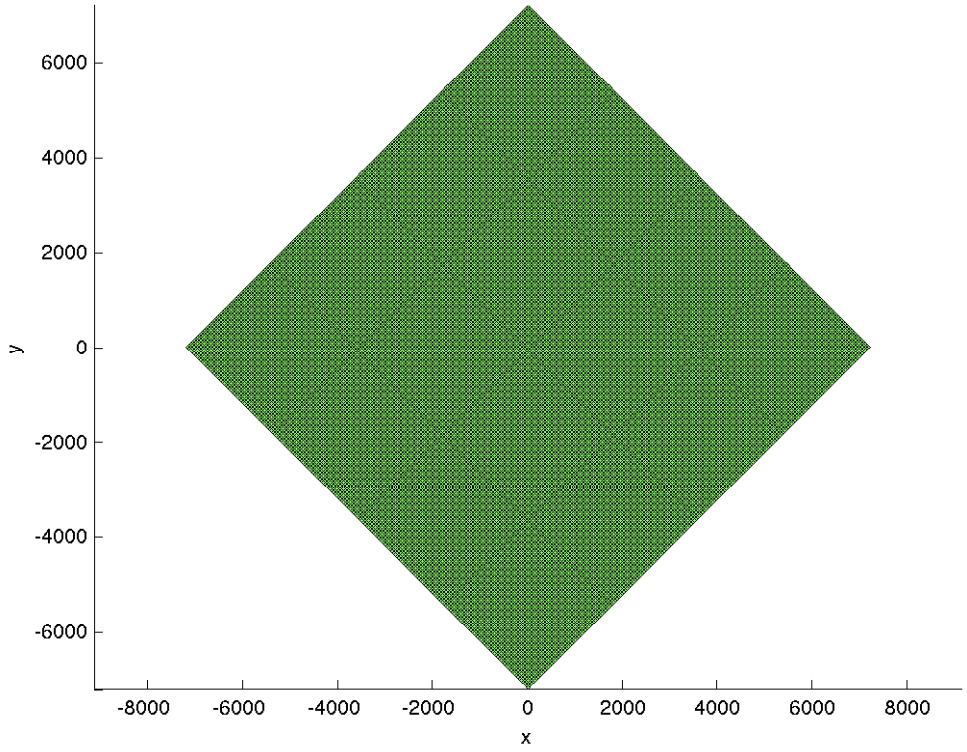


Figure 4: Orthognal Rotated Domain.

along the x-axis and v is along the y-axis; therefore, they do not match the grid orientation and the curvilinear computations are necessary.

both components of the velocity were initialized to zero and the water table was initialized with the following function:

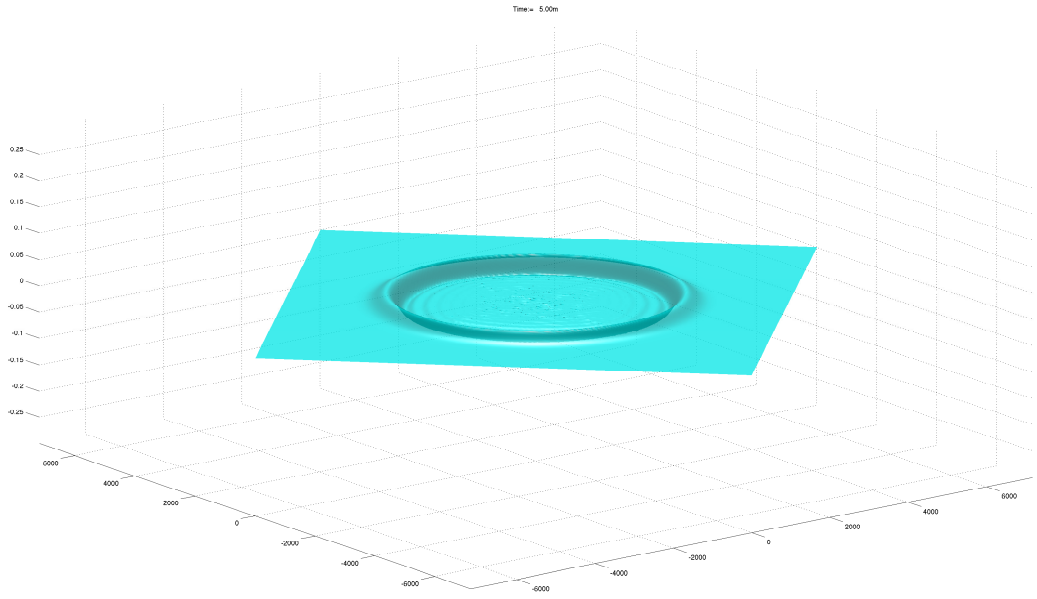
$$w = \frac{1}{\sqrt{1 + e^{80r}}}, \tag{13}$$

where r is the normalized distance from the center of the domain. All boundary conditions are periodic. The water table after 5 minute of simulation is shown in Figure 5.

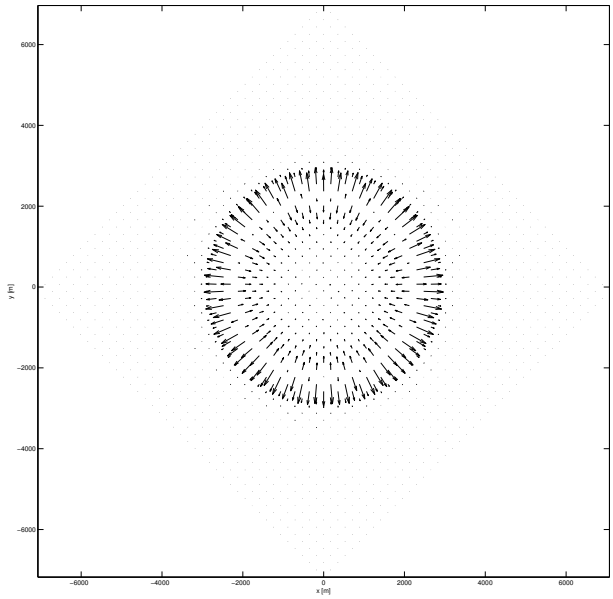
3.4 Long Channel With Island

One of the goals when using a curvilinear grid is to match the boundaries of the domain properly. Yet, in cases where the domain includes an island in the middle of the ocean, one is required to produce multiple domains, and then link them together. Using multiple domains is beyond the scope of this research; however, the ability to include a few small islands in the simulation, a land/sea mask was added to the scheme. Fluxes were automatically set to zero for all cells defined as "dry" cells. Note that there was no wet or dry scheme implemented; therefore, all cells should stay either dry or wet for the entire simulation. Also note that, having a dry cell in the middle of the domain is a source of discontinuity in the solution. CGM difference operators, ike all other difference operators, are to some extent sensitive to the discontinuity in the solution. Because of this, extra care must be taken so that the model does not become unstable.

To test such situation, a channel, 20000 m long and 3000 m wide, was discretized using $dx = dy = 50\ m$. A periodic boundary condition was applied for both u and v component of the velocity. The bathy metry was sloped, $s = 0.001$, along the channel length, i.e. x-axis, and w was set to provide constant pressure head at the beginning and the end of the channel. A cubic island with length of 600 $m \times 600\ m$ was defined in the middle of the channel. No-Slip boundary condition is set along the edges of the island and no flux condition is set for w. No computation



(a) Water Table

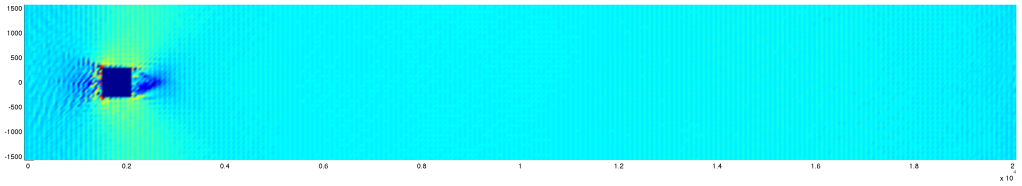


(b) Velocity Vectors

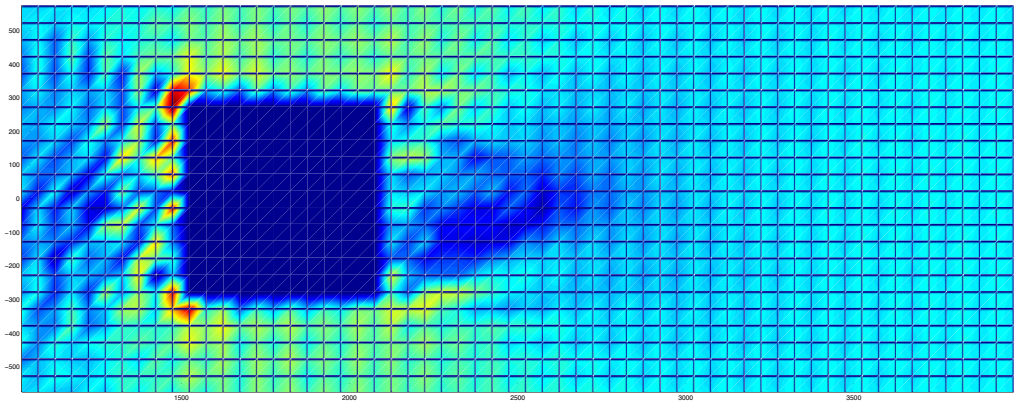
Figure 5: Changes of error by increasing operator order

is performed over dry cells though. On NVIDIA GTX 480 it took 216 seconds to simulate 24 hours of simulaiton. Figure 6 shows the results after 24 hours of simulation with $dt = 1$ $[s]$.

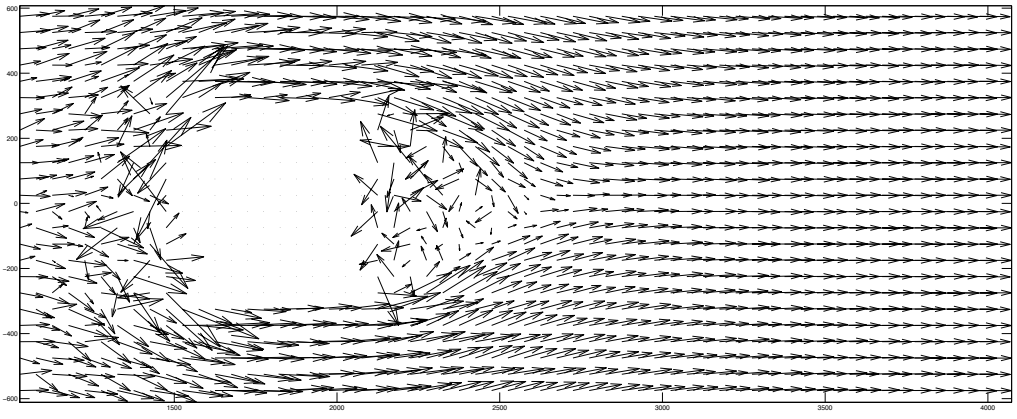
Figure 6 does not show any creation of Karman Vortices even after 24 hours of simulation. This could be due to too much dissipation or grid resolution. Here the reason for not seeing these vortices is the low resolution. To show this, a finer grid with $dx = dy = 10$ $[m]$ was selected.Also C_z was set to 20; therefore, we are introducing even more dissipation. To keep the model stable dt was set to 0.1 $[s]$. The 2 hours of simulation took about 15 minutes on



(a) Water Table



(b) Water Table zoomed around the island



(c) Velocity Vectors

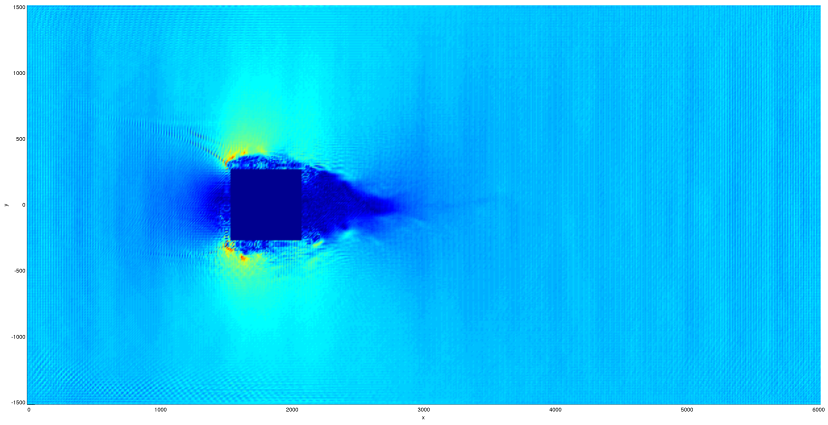
Figure 6: Simulating a channel with a dry zone in the middle, $C_z = 35$, $t = 86400\ [s] = 24\ [hr]$, $dx = dy = 50\ [m]$.

NVIDIA GTX 480. Note that despite all calculation being double precision, this device is best for single precision calculation and is not even optimized for double precision computation, yet it was possible to achieve 8 times faster than real time. Every 30 seconds a snapshot was written to the disk. By playing these snapshots, i.e. creating a movie, the karman vortices could clearly be seen. A snapshot at the end of the 2 hours is shown in Figure 7.

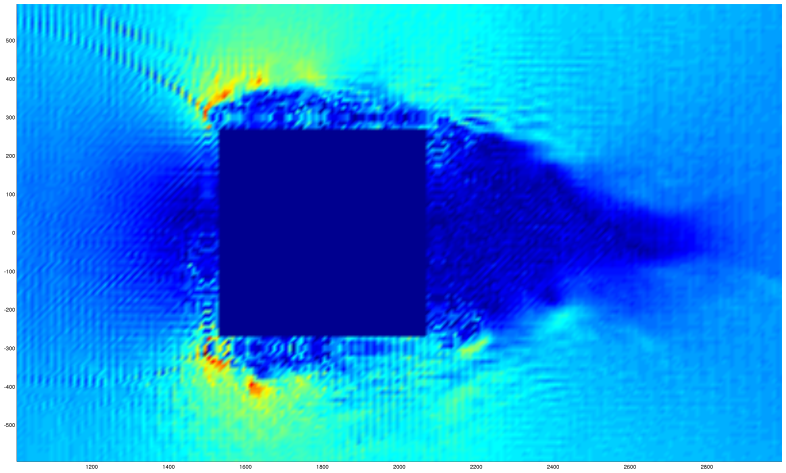
3.5 Monterey Bay

Another test case is the Monterey bay. The bathymetry of the Monterey bay is shown in Figure 8. Due to the presence of very deep underwater valleys in the bathymetry, the simulation of this region is rather challenging. The fast changing bathymetry results into very steep slopes; making it difficult in the momentum equation to balance the bottom slope term properly. One solution is to use a much higher resolution in those region. We tried a 500 m grid resolution and still it was not fine enough to handle those slopes properly; as a result, the imbalance in the momentum equation generated big waves which causes the model to collapse the moment those very high waves reached the shallower region. This suggest that the current model is too sensitive to bottom bathymetry and special care is needed for those regions.

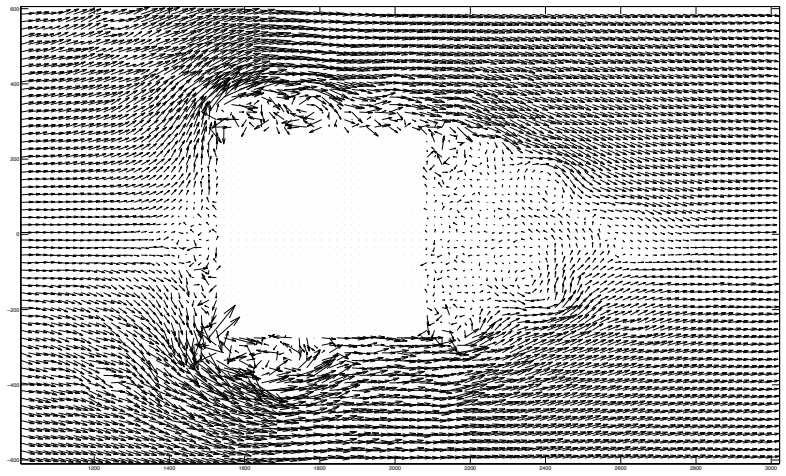
Although not realistic, the bottom bathymetry was set to a constant depth and a grid with 210×205 nodes was generated. The fluid was initialized to be at rest at time zero. At the north boundary a sinusoidal function with an



(a) Water Table



(b) Water Table zoomed around the island



(c) Velocity Vectors

Figure 7: Simulating a channel with a dry zone in the middle, $C_z = 20$, $t = 7200\ [s] = 2\ [hr]$, $dx = dy = 10\ [m]$.

amplitude of 1 m was set as boundary condition for the water table; and at the south boundary another sinusoidal function was set, which had the same frequency but half the amplitude. The function at the northern boundary and the southern boundary had 180 degree phase shift. At the west boundary the Neumann's boundary condition was assigned and at the east boundary (the shore line) the no flux boundary condition was assigned. Figure 9 shows the water table at two different times. Figure 10 shows the changes of water table at three different locations during

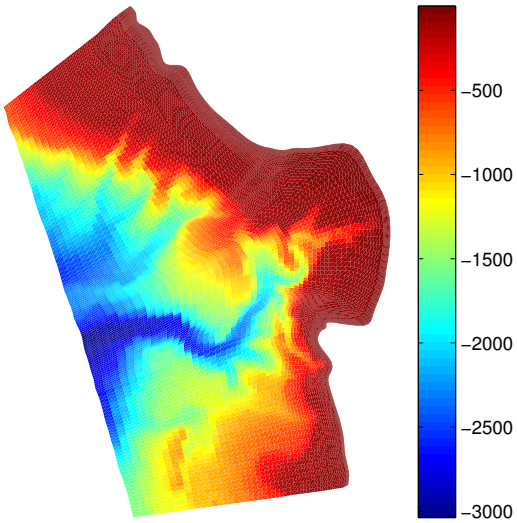
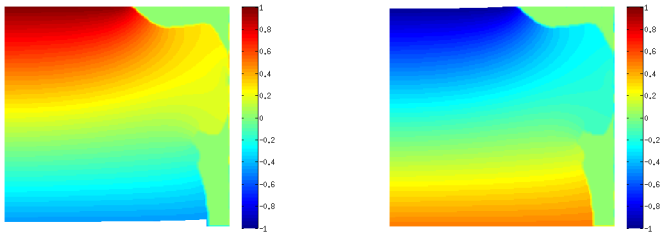


Figure 8: Monterey Bay Bathymetry.

the entire simulation.



(a) Water Table at $Time = 6\ hr$. (b) Water Table at $Time = 18\ hr$

Figure 9: Water table in the Monterey bay with flat bathymetry.

References

CUDA C Best Practices Guide.

CUDA C Programming Guide.

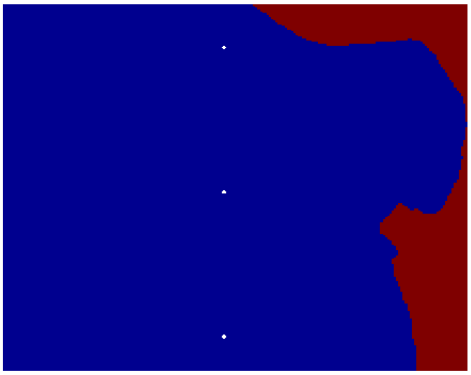
Fermi Tuning Guide.

Kepler Compatibility Guid.

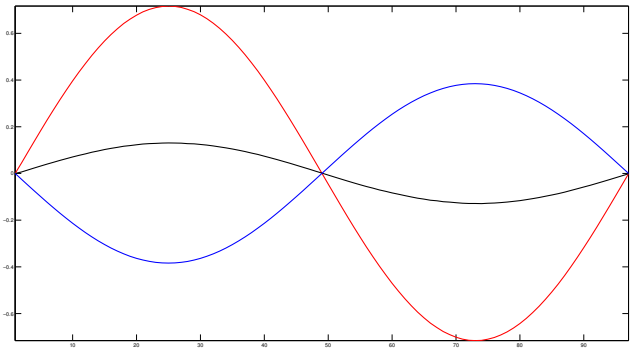
Kepler Tuning Guide.

Abouali, Mohammad, Timmermans, Joris, Castillo, Jose E., & Su, Bob Z. 2013. A high performance GPU implementation of surface energy balance system (SEBS) based on CUDA-C. *environmental modeling and software*, **41**, 134–138.

Borthwick, A.G., & Barber, R.W. 1992. River and Reservoir flow modelling using the transformed shallow water equations. *International Journal of Numerical Methods in Fluids*, **14**, 1193–1217.



(a) Location of the three points.



(b) Changes in water table at fix points during simulation time.

Figure 10: Changes in Water table at three points during 24 hours simulation. (Red) a point closer to norther boundary, (blue) a point closer to souther boundary, and (black) a point almost in the middle of the domain.

Brodtkorb, André R., Sætra, Martin L., & Altinakar, Mustafa. 2012. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, **55**(0), 1 – 12.

Ezer, Tal, & Mellor, G.L. 2004. A generalized coordinate ocean model and a comparison of the bottom boundary layer dynamics in terrain-following and in z-level grids. *Ocean Modelling*, **6**, 379–403.

Ferziger, Joel H., & Millovan, Peric. 2001. *Computational Methods for fluid dynamics*. 3 edn. Vol. 1. Springer.

Kurganov, A., & Petrova, G. 2007. A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. *Communications in Mathematical Sciences*, **5**, 133–160.

Kurganov, A., Noelle, S., & Petrova, G. 2001. Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations. *SIAM Journal of Scientific Computing*, **23**, 707–740.

Wijbenga, J.H.A. 1985a. Determination of flow patterns in rivers with curvilinear coordinates. *In: Proc. 21st IAHR Congr.*

Wijbenga, J.H.A. 1985b. Steady depth-averaged flow calculations on curvilinear grids. *In: Proc. 2nd Int. Conf. on the Hydraulics of floods and floods control.*