Mr Wags Documentation

Sofia Hedberg, Viktor Wase
PrecisIT, Uppsala, Sweden

2015

1. Introduction

Mr. Wags (*MultiResolution Wavelet based AlGorithm Simulation*) was PRECISIT's summer project of 2015. It is a massively parallel CUDA-based simulation of the lid driven cavity problem in 2D.

2. Theory

2.1. Lid Driven Cavity

The incompressible Navier Stokes equation describe the motion of dense fluids. If the velocity vector is denoted u and the pressure is denoted p the equation is as follows:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \frac{\alpha}{Re}\nabla^2 u = f$$

$$\nabla p = 0$$

where f is a forcing function, α is a scaling constant that depends on the units used and Re is Reynold's number.

2.2. Vorticity/Stream reformulation

The incompressible Navier Stokes equations are not the simplest of equations, but fortunately there is a reformulation that simplifies the system. This only works on a Cartesian Grid in exactly two dimensions.

Introduce vorticity w and stream Φ defined by

$$w = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}$$

and

$$\frac{\partial \Phi}{\partial y} = u_x$$

$$\frac{\partial \Phi}{\partial x} = -u_y$$

where u_x is the velocity in the x-direction and u_y in the y-direction.

The incompressible Navier Stokes takes the form

$$\frac{\partial w}{\partial t} = \frac{\nabla^2 w}{Re} - u_x \frac{\partial w}{\partial x} - u_y \frac{\partial w}{\partial y}$$

$$\nabla^2 \Phi = -w$$

By treating $-w$ as a forcing function in the second equation it turns into the simple Poisson equation. If this is solved then the values of all the variables of the right hand side of the first equation are known. It is then simple enough to numerically integrate w a time step.

The Navier Stokes system is thus functionally reduced to a series of Poisson equations.

2.3. Finite Difference Method

The Poisson equation $\nabla^2 u = f$ is discretized using the second order central difference method

$$\nabla^2 u \approx \frac{u((i+1)h, jh) + u((i-1)h, jh) + u(ih, (j-1)h) + u(ih, (j+1)h) - 4u(ih, jh)}{h^2}$$

where h is the step size in both the x-direction and the y-direction and i and j are the x and y indexes.

If h varies in any way this formula becomes much more complicated and would increase the computational time significantly. Adaptive grids usually forces the step size to vary; but by using multiple grids, each with a different h, this can be avoided.

These multiple grids are then connected using the Multigrid Method.

2.4. Boundary Conditions

The boundary grid points are treated as unknowns to keep h constant for each grid. However in order to enforce the boundary values a punishing parameter κ is introduced. We set $\kappa = 10^6$.

If one implements Dirichlet B.C.s where $u = g$ on the boundary then $u = g \rightarrow \kappa u = \kappa g$. Thus κg is added to the force vector and κ is added to the discetization of the Laplace operator.

2.5. Continuous Wavelet transform

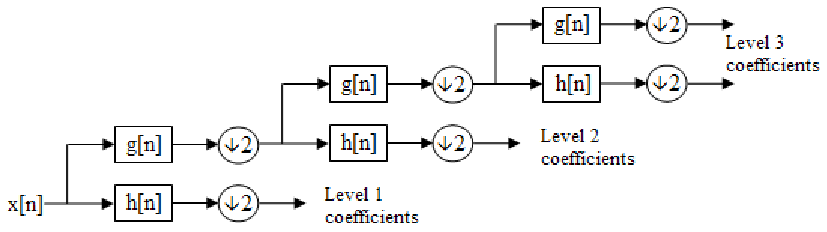
A transform method is required when decomposing a signal. One of the most frequently used is the Fourier transform. When applied it takes the signal from the time plane to the frequency plane. This method is suitable for stationary signals since it integrates over all times, from minus infinity to infinity. The transformation gives all frequencies in the time span, but does not take the time in consideration. This is a good method for a stationary signal that has the same frequency components at all times, but if the signal is non stationary with different frequencies at different times the Fourier transform will give you the same result as if the frequency components would appear at all times.

To solve this problem the Continues Wavelet transform was created. This is a transform method which takes both the time and the frequency in concern. In short terms the difference between Fourier transform and Wavelet transform is when the Fourier transform is integrated, it integrates over the whole time span and the output is the frequency span over all times. The Wavelet transform on the other hand takes a small wave, called a mother wavelet, which has similarity to the signal. The signal is integrated by taking small steps through the whole signal and the mother wavelet is compared to the signal in every step. The step length is then expanded and the mother wavelet is once more compared to the signal in every step. This procedure is repeated with increased step length and gives the result of all frequencies, from high to low, in the signal with a time relation, which gives the location in time.

Performance of Wavelet transformation is time consuming and a computer is required in nearly all cases. Transferring the Continues Wavelet transform to computer computation obliges a discretization, which causes lot of unnecessary data and is very time consuming. Therefore the Discrete Wavelet transform is wiser to use.

2.6. Discrete Wavelet Transform

The Discrete Wavelet transform is any Wavelet transform for which the wavelet is discretely sampled but continuous in time. Today it is often used in for example compression of pictures. The idea is to separate the high and low frequencies over and over again. This is done by passing the signal through filters as shown in the figure below.



The signal is first passed through a filter separating the high ($g[n]$) and low ($h[n]$) frequencies. The low frequencies are stored in Level 1 and the high frequencies are down sampled by two and again filtered separating high and low frequencies. In this way the signal is split in multiple levels and therefore easier to store and less space consuming. The signal is compressed. From the compression it is relatively simple to decompress the signal by doing the inverse of the compression, which will give the original signal.

For infinite or periodic signals the Discrete Wavelet transform work well. Although, when having a signal in a bounded domain, where signals are not infinite and periodic, the construction has some shortcomings. Furthermore, as well as in one dimension as in higher dimensions, signals are not sampled regularly and often have boundaries. In these cases we can use The Second Generation Wavelet transform.

2.7. Second generation Wavelet Transformation

The Second generation Wavelet transform has a number of advantages compared to the classical wavelet transform, like the Discrete Wavelet transform. It is by a factor of two quicker to compute and can be used to produce a multiresolution analysis that does not fit a uniform grid. Therefore it is a good method for compressing data combined with an adaptive grid.

The second generation wavelet transform have some similarities to the discrete wavelet transform and is seen as an extension of the later. The biggest difference is that the Second generation Wavelet transform is a method based on something called the lifting scheme. The lifting scheme has the properties to only save the points in a signal which are necessary for decompression. Instead of compressing the signal, like in the Discrete Wavelet transform where all points are stored for reconstruction of the signal in decompression, only some of the points in the signal are saved using the lifting scheme and therefore it needs much less data space. In calculations where big multitudes of data is used this is therefore a good method, or even a must to be able to handle these magnitudes of data.

For one dimension the theory of the second generation wavelet transform is pretty straight forward and can be explained in 6 steps:

1) Apply point throughout the signal with an even step length.

2) Split the points in even end odd points.

3) Extract every other point (even or odd points).

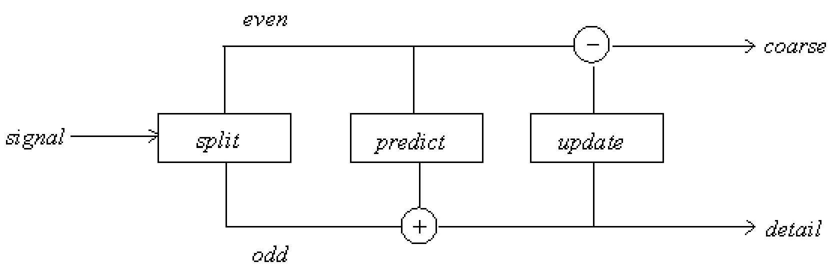
4) Predict the value of the point in between two of the remaining points (predict odd point when even points are saved and vice versa) by linear interpolation.

5) Compare calculated value with real value and see if it satisfies the tolerance. If yes, save the point.

6) Increase the step length (extract every other point again) and proceed from (4).

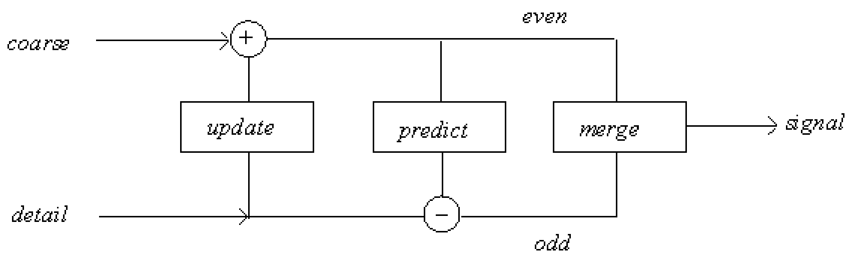
When just a few points remain in the signal, these are saved as anchor points to have something to start with in decompression. The steps of split, predict and save (update) can be visualized as in the figure below.

Digital signals are much correlated, and the correlation structure is local. Therefore the prediction and comparison to the original point is needed to catch points (within a known tolerance) where the signal is changing



at most. This is to maintain the parts of the signal where the biggest variations occur. The key is to extract the point in the signal that is below a threshold (satisfies the tolerance) and save the ones that makes major change in the signal so that the detail of the signal does not wane.

The decompression of the transform can easily be obtained just by doing the inverse of the compression. This means that all the steps calculated in the compression should be done in the opposite order, as showed in the figure below.



3. Structure

In its simplest terms *Mr. Wags* is a Finite Difference simulation of the incompressible Navier Stokes equation. It uses an adaptive multiresolution grid to ensure that the computational power is focused on the areas where it's most needed.

3.1. Wavelets

In Mr. Wags the Second Generation Wavelet transform is implemented. This since the input data is thought to contain a massive matrix and the data has to be scaled down. The compression is also used together with an adaptive grid, which means that the important points in the matrix is where the big changes occur, one of the properties that the Second generation Wavelet transform obtains.

Mr. Wags is in two dimensions and the theory of the theory of the Second generation Wavelet transform in one

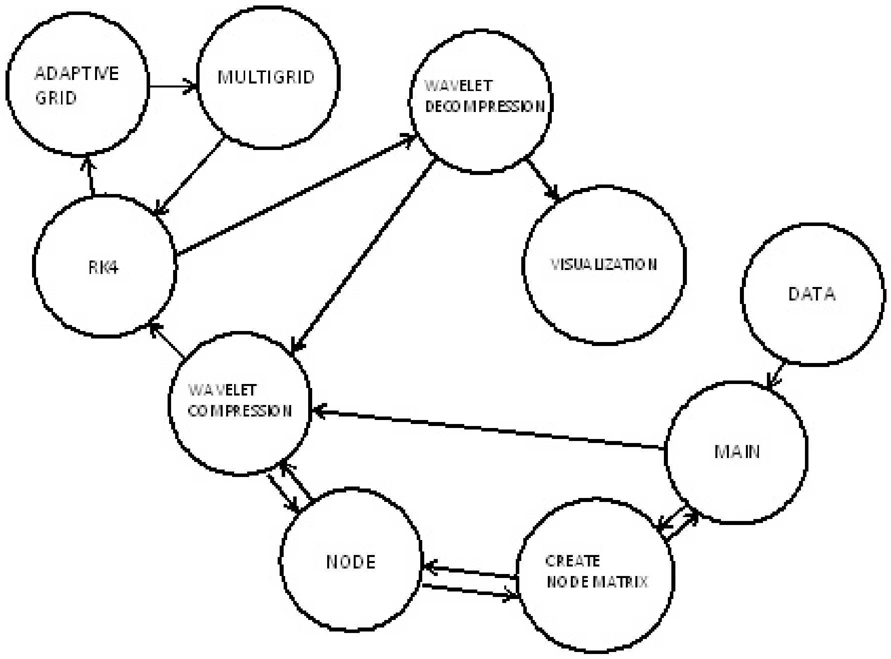
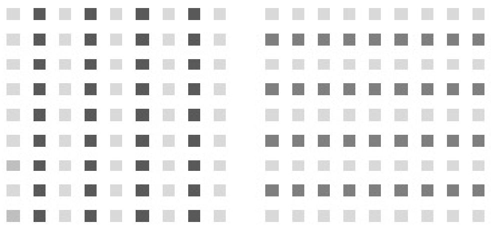


Figure 1. **The internal structure of Mr Wags.**

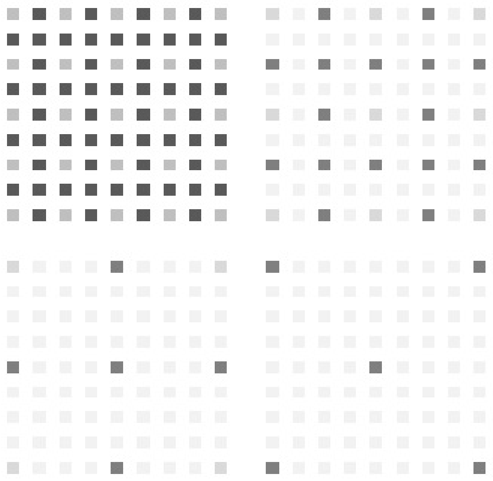
dimension has therefore been extended to another dimension. This has been done by executing the same steps as in one dimension but both vertically and horizontally.

To show the procedure in more detail a 9x9 matrix is taken as an example.



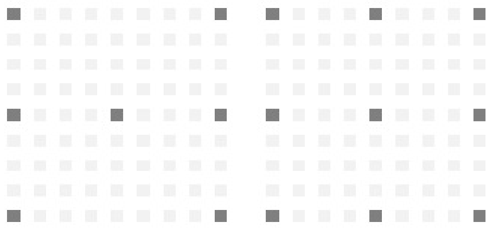
In the first step the computation is made horizontally, visualized in the figure to the left. A step length of two is chosen and points marked dark gray are extracted. With linear interpolation the value of the point in between the remaining points (grey points) is predicted and compared to the real value. The extracted points, which does not satisfy the tolerance level, are saved. The same procedure is than done vertically, shown in the figure to the right.

The process continues by expanding the step length by the power of two and extracts every other point for a second time, both horizontally and vertically. This process is repeated until the corner points are the only ones remaining and these, together with the middle point, are saved as anchor points. When the procedure is done, both horizontally and vertically, every step gives a pattern of extracted and remaining points as showed in the figure below.

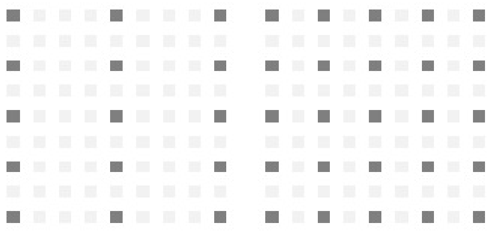


Seen in the figure to the upper left is the first step in the procedure with a step length of two, where the dark grey point represent the extracted points and the grey the remaining ones. To the upper right the pattern of the second step is visualized, where the step length is expanded by the power of two. In the last figure, to the lower right, the last step is presented. Only the corner points remain and are stored together with the middle point as anchor points. Even though this example is only for a 9x9 matrix, the procedure is completed the same way for larger matrices just by increasing the number of steps in the process.

The decompression in Mr. Wags is simply the inverse of the compression, where all the steps are done in the opposite order.



In the first step the procedure proceeds from the saved anchor points and the steps length used in the last iteration of the compression. By linear interpolation, calculated between the corner points first vertically and then horizontally, a new point is created in between as shown in the figure above. If the space for the new point already contains one of the saved points from the compression, that point is the one used. In the next step the step length is decreased by the power of two and the process is repeated, first vertically and then horizontally, as presented below.



These steps are continued until the whole matrix is reconstructed and then the decompression is complete.

The process for the compression and decompression in Mr. Wags is parallelized and executed completely on the GPU. For further work there are though a few lessons learned and some changes that have to be made.

- The information of every point in the matrix is stored in Nodes. These contain a lot of data and are therefore very big, which means that when the matrix gets bigger with more Nodes it gets unnecessarily space consuming. To make the calculations on the GPU more efficient there would be a good idea to create a new class where the Nodes contain just enough information for the Wavelet compression to work, and in that way be able to compute a bigger matrix on the GPU at once.
- The Wavelet compression in Mr. Wags is now built for sending the matrix as a whole to the GPU. This has to be changed when the matrix gets larger, alternatively by splitting the matrix in smaller chunks and sending them one by one to the GPU throughout a loop in the code on the CPU. This change has to be done in the decompression as well.
- Even though the theory of decompression directs that it is just the inverse of the compression, the code for decompression is not completely the inverse of the compression. This since the decompression is depending on serial computation (some tasks need to be executed before others) and therefore requires and loop directed from the CPU to make sure that the calculations are made in the right order. This problem does not arise in compression since computations in one point is independent of the calculations in the other points.

3.2. Node

In order to simplify the coding a class called `Node` was implemented. The idea is that all information in each point in each grid should should be contained in a Node object. That includes the value of the vorticity and stream, but also information regarding in which grid it is in, which its neighbouring nodes are and its position is in the grid.

Unfortunately each object is 80 bytes, which is rather big and thus restricts the size of the grids.

3.3. Adaptive Grid

The multigrid method requires a hierarchy of grids, where each grid is a subset of the previous with a smaller step size. Each of these grids are implemented as an object of the `AdaptiveGrid` class. These objects contain all information that the Multigrid algorithm needs in order to solve the Navier Stokes equation.

The `AdaptiveGrid` constructor is unfortunately still a host function. Moving it to the device (GPU) is non-trivial since it is highly serial and in some places even recursive.

The most important features of this class is perhaps the four `Node` arrays. The u-array is the current best guess for the solution of the Navier Stokes problem. The b-array is RHS of that problem. That means that the equation to be solved is $Du = b$, where D is the discrete Laplacian operator (with Dirichlet BC in it).

The d and w-arrays are used to store temporary information in the multigrid algorithm.

The origo variables gives the global index of the lower left corner of the grid.

It should be noted that the order of the nodes in the node-arrays are random. This has some severe impacts on the performance since each memory call has complexity O(N). This should probably be changed into a Hilbert-curve ordering, or they could be stored in a tree to reduce the complexity to O(log(N)).

```
class AdaptiveGrid{
public:
    Node *u, *b, *d, *w;

    std::vector<Node> vec;
    int  layerNr, numOfLayers, origo_x, origo_y, len;

    AdaptiveGrid* finerGrid;
    AdaptiveGrid* coarserGrid;
    datatype h;
}
```

3.4. Multigrid

The `multigridGpu` function is a recursive function that solves the Poisson equation on the grids defined by a collection of `AdaptiveGrid` objects that are sent as input.

How the multigrid methods works is an entire essay in itself, and I will not go into it here, but there is an abundance of good tutorials online.

The multigrid methods are highly serial and are perhaps not the obvious choice when computing on a GPU. However all of the components of the multigrid are easily parallelized (except perhaps for the smoother, but we will come to that) and when the data is big enough each of these components will get a decent speedup from GPU-acceleration.

The major components of multigrid are: interpolation, restriction and smoothing. The interpolation is linear, the restriction is copies the value of the `Node` object from the finer grid to the coarser grid.

The smoother is the weighted Jacobi with $w = 2/3$. This is a simple smoother with incredibly slow convergence and unfortunately the majority of the work of *Mr. Wags* will be computing the smoothing. The reason why we chose the method is that it is perfectly parallel and thus is very fast on a GPU. This is not a property that is common in smoothers and the more common ones like Gauss-Seidel and SIMPLE (which is rather difficult actually) would most likely perform worse on a GPU even though their convergence factor is higher.

However last year the *scheduled relaxation Jacobi method* was invented. It has all the nice properties of the regular Jacobi method (except its simplicity) and requires roughly 200 times fewer iterations than the regular Jacobi for problems of our size. We did not have time to implement it.

3.5. Runge-Kutta

We chose the 4:th order Runge-Kutta scheme as our time integration method simply because it is the simplest implicit time stepping method that still has a high order convergence. If shock waves are introduced this method might give rise to some unwanted behaviour since higher order methods tend to give undesired oscillations in those kinds of situations.

4. Methods

4.1. Wavelet compression

The Wavelet compression is using the Second generation Wavelet transform method in two dimensions. It takes the input Node matrix and sends it to the GPU. Calculations are done for all Nodes in parallel to determent which Nodes to save. For every Node a computation is made throughout a loop, where for each iteration the step length is expanded. Saved Nodes are altered by setting the Node values .isPicked equally to true and .layer to the number of iterations completed. Anchor points, in form of the corner Nodes and the middle Node, are automatically saved and the Node value .layers is set to the declared value LAYERS.

The transformed Node matrix is than sent back to the CPU, where the number of true values is stated in the variable countTrue. All true Nodes are stored in the array variable nodeArray in an order starting with Nodes in the roughest layer (LAYERS) and ending with the Nodes in the finest layer.

4.2. Wavelet decompression

Wavelet decompression executes the decompression of the Wavelet compression. The Nodes in the nodeArray are transferred to an empty Node matrix, with the size LEN_OF_MATRIX x LEN_OF_MATRIX. The matrix is sent to the GPU throughout a loop, initiated with the value LAYERS. New Nodes are created by linear interpolation, first vertically and then horizontally, filling the empty spaces in the matrix. When the loop has iterated from the roughest layer down to the finest the matrix is complete and sent back to the CPU.

4.3. Adaptive Grids

The adaptive grids are divided into two separate files: one .cpp file and one .cu file. The first one is just for the construction of new grids, while the other one contains the methods that are necessary in order to perform multigrid operations on the GPU.

5. Notable functions

5.1. Adaptive multigrid new

The `adaptive multigrid new` creates a number of grids based on the input data and then runs the multigrid method. The results are stored in the input argument `array`.

`array` is a node array of length `countTrue`. The stream and vort attributes of the nodes will be changed.

`origoArray` is an int array of length 2×`countTrue` that contains the global x and y indecies of the lowest leftmost point of each of the adaptive grids.

5.2. multigrid gpu

The adaptive multigrid is a recursive function that solves a system of equations using the Multigrid method on the GPU.

`k` is an internal counter and should always be $LAYERS - 1$ in all external calls.

`grid` is an array with all the adaptive grids.

`pre`, `sol` and `post` define the number of pre, post and solution iterations the smoother should make.

`countTrue` sets the length of the two arrays.

5.3. RK4

The `RK4` function applies one step of the 4:th order Runge-Kutta method using the multigrid method to calculate the relevant stream and vorticity values.

`dt` is the time step; it is defined in `parameters.h`

`y vector` is a node array of length `countTrue`. It contains all the points that the wavelet compression has deemed important. The new values are stored in this array.

`origoArray` is an int array of length 2×`countTrue` that contains the global x and y indecies of the lowest leftmost point of each of the adaptive grids.

`countTrue` sets the length of the two arrays.

6. Usage

To use Mr. Wags the parameters in the file *parameters.h* must be altered to represent the problem at hand. For example, if the size of the box is changed from unity Reynold's number must be altered correspondingly. Some other parameters (such as DELTA_T and TOLERANCE) restrict the magnitude error.

To compile the code a simple make command will suffice.

To run the code run the command: ./MrWags

7. Visualization

When creating a simulation there is sometimes a need to visualize it. The creators of CUDA, NVIDIA, recommend using CUDA together with the graphical open source API OpenGL. This since the API of OpenGL has been extended with libraries that combine OpenGL with CUDA. OpenGL is a cross language, multiplatform API and is an interface for graphical hardware. The language reminds you of C but is totally language independent.

The combination of OpenGL and CUDA creates a fast communication between the two directly on the GPU. In short terms the languages communicate with commands throughout shared memory in the frame buffer. This is done by OpenGL storing data in something called buffer objects which are than mapped into a CUDA memory space.

Even tough OpenGL can be run both from the CPU and the GPU the combination with CUDA gives an opportunity to visualize something directly on the GPU, cutting out the time consuming sending of data between the CPU and the GPU.

7.1. Lessons learned and further work

Even though Mr. Wags does not contain visualization right now, work has been spent on trying to visualize the simulation with OpenGL. An example code is appended to get a glimpse of how a simple OpenGL code can look like (not joined with CUDA). Therefore a lot can be done in this area to develop Mr. Wags and below follows some lessons learned that can be good to think about in further work.

- OpenGL is a language with a lot of libraries to understand. Lessons learned are that it takes fare more time to get the hang of it than expected.
- When using libraries in OpenGL, make sure that the ones you want to use is downloaded (for example FreeGLUT).
- Including libraries in the header must be done in a specific order. This since some of the libraries depends on others and therefore has to be included after those ones. Also make sure that they are written in the way that matches your operating system (for example include <GLUT/glut.h> on mac and include <GL/glut.h> on Linux).
- Using libraries you should be careful with defining datatype titles or macros with common names. This was

learned the hard way since there can be variables et simila with the same name in the OpenGL libraries with the same name, which causes errors.

- Make sure to initiate libraries that require initialization. For example glutInit(&argc, argv) which initializes GLUT (needed when for example setting window size, and start the visualization loop) and glewInit() which initializes GLEW (good to use tougher with the OpenGL/CUDA commands).
- Use required flag when compiling the code (for example GL for GLUT).

8. References

8.1. CUDA

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

8.2. Wavelets

http://www.ima.umn.edu/industrial/97_98/sweldens/fourth.html

http://www.ee.ucl.ac.uk/ iandreop/Angelopoulou_et_al_JVLSI_Comparisons_53_published.pdf

http://web.iitd.ac.in/ sumeet/WaveletTutorial.pdf

8.3. OpenGL

http://pebble.mines.edu/˜aelsherb/pdfs/journal _papers/2012/CUDA-OpenGL_Interoperability_to _Visualize_EM_Fields.pdf

http://on-demand.gputechconf.com/gtc/2012/presentations/S0267A-GTC2012-Mixing-Graphics-Compute.pdf

http://www.drdobbs.com/architecture-and-design/cuda-supercomputing-for-the-masses-part/ 222600097?pgno=2

8.4. Multigrid

Multigrid, *Academic Press*, U. Trottenberg.