GPU-based acceleration of a water wave model

Michal Biskup

IMM-MSc. 2010-21

Abstract

A two-dimensional water wave model based on potential flow is investigated with the intention of simulating marine water on the large scale. The transformed linearized Laplace problem, which arises in the model is solved by the Multigrid method on the GPU using CUDA. Full Multigrid cycle is extended by an additional sequence of V-cycles to achieve desired accuracy. The algorithm is terminated after the iterative method error becomes insignificant compared to the discretization error of the finite difference method. Multigrid is completely parallelized to fit the GPU architecture. Every step of the algorithm is designed in such a way, that no communication between GPU threads is necessary. Jacobi method is used as the smoothing step in the Multigrid. A discretization scheme using an additional layer of grid points lying outside the fluid domain, described in [A.P. Engsig-Karup, H.B. Bingham, O. Lindberg, An efficient flexible-order model for 3D nonlinear water waves], is used. All GPU computations are performed using double-precision. For devices, which do not support double -precision operations, single-precision mode is also available. Experiments performed on NVIDIA Tesla C1060 show linear growth of execution time with the problem size. The method is robust enough for available memory to be the limiting factor, rather than required execution time. Problem sizes with over 67 millions of unknowns were tested. Experiments show monotonic convergence and rate of convergence is independent of the problem size, for all considered cases.

Table of Contents

Illustration Index

Introduction

The main goal of this project is to create a efficient tool capable of simulating marine water phenomena on a very large scale. Navier-Stokes equations can be used successfully to model local phenomena like viscous effects close to solid boundaries, however computational power limitations become prohibitive, when we need to deal with large scale problems. A potential flow approach [1] is used instead and focus is placed on finding the fastest way of solving the problem using available hardware.

In the recent years parallelism in computing began to play an increasingly important role. Modern CPUs do not follow Moore's Law anymore due to heating problems and miniaturization issues. To overcome this problem hardware manufacturers focus on increasing number of cores in the processor, rather than trying to make a single core faster. Multiprocessor hardware is not the domain of few large organizations anymore, which own huge processor clusters. Even desktop computers today contain at least a dual-core processor and processors having more cores are getting cheaper and more popular. With the release of Tesla GPU series by NVIDIA, high performance computing became even more available. For example the Tesla C1060, which will be used for this project, contains 240 thread processors and at the time of writing even more powerful configurations are available. What is more NVIDIA released CUDA technology, which makes it much easier to communicate with the GPU. Unfortunately existing algorithms usually are not capable of running on more than one core and the increase in hardware power does not result in any speed up. One of the motivations for this project is investigation of various algorithms and a choice and implementation of the one which is most suitable for modern hardware.

Previous work

There is a lot of research focused on exploiting the power of modern GPU processors. The study made by IBM [2] shows ways to optimize sparse matrix vector multiplications – a common building block used in most if not all PDE solvers. The paper describes challenges faced during implementation of a parallel algorithm on the GPU. The efficiency of the algorithm depends on various hardware dependent implementation details, like the choice of data storage and data access schemes. A significant speedup is shown, when threads are mapped in such a way, that coalesced memory reads are

possible and when cache is used.

In [3] the heat transfer problem is modeled by the Poisson equation and accelerated on GPU. This is particularly interesting for this project, as both heat transfer and potential flow for water waves are very similar in mathematical formulation as well as numerical solution. OpenGL and cg was used to communicate with the GPU. Only 32-bit precision was used for the project, as that was the hardware limit at the time. This is going to be extended to full double precision in this project. [3] Presents speedups of about 15X compared to the CPU implementation, however the problem size was very limited by the used hardware (2d problem of size 1024 x 1024). Results were also limited by the fact that 32-bit precision hardware was available only as a CPU emulation and a Radeon 9700 with only 24-bit precision was used to produce actual results. Modern hardware today seems to be much more ready to face challenges in this project both in terms of pure hardware power and software support (CUDA).

A more recent paper [4] uses Biconjugate Gradient Method (BiCG) with Multigrid as a preconditioner to solve the Poisson problem. Jacobi method is used as the smoother. The approach of the project was to integrate GPU acceleration into existing libraries, rather than produce a most efficient solution. Results show a very poor speedup (2x to 3x) of the used methods. The main reason for this is that BiCG cannot be easily parallelized. The results in the paper are one of the reasons, why BiCG and other Krylov subspace methods will be avoided in this project.

Another approach [5] uses yet different algorithms to solve elliptic problems. Conjugate Gradient method is used with the Multigrid as a preconditioner. The limitation of this method is that it can only work for symmetric and positive-definite matrices. The paper shows that using the standard compressed row storage format (CRS) does not work efficiently on the GPU. An ICRS – interleaved CRS – is proposed to solve the problems with non-coalesced GPU memory reads. This format however still introduces overhead and the need of additional data structures for for storing column numbers and displacement vectors. The paper shows results, where a problem with 862151 equations is solved in 1.2 seconds on Tesla C1060. Solving larger problems was not attempted. The setup of the AMG (Algebraic Multigrid) was not accelerated on the GPU due to problems with parallelizing the algorithm.

Finally [6] presents timing and speedup results of the parallelized version of the very popular GMRES algorithm. The results show a 20x speedup for a problem size of about 1.1 million of unknowns. The exact problem was not defined it the paper however, so it is difficult to compare the results to other work. An expensive part of the algorithm – incomplete LU decomposition – is replaced by block ILU. The sparse matrix is divided into blocks, which are processed in parallel. It is worth to notice, that the absolute time on Tesla T10P GPU for the largest problem size considered (1.1 million) was 22.1 seconds, which is quite a lot.

GMRES, while very successful in the single thread environment, introduces a lot of unresolved problems in the parallel environment. A detailed study of the scalability of GMRES in parallel environment in [7] shows, that much care must be taken while choosing a method for orthogonalization process for the Krylov subspace. Modified Gram-Schmidt process exhibits large communication overhead. The paper suggest using Householder reflections instead. The algorithm was tested on machines consisting of huge processor arrays, where communication is easy, because all processors are well connected (in a 2d grid – Intel Paragon and 3d torus – Cray T3E). Communication with other processors is required at every single step of the orthogonalization process.

[8] Is an another attempt of creating a parallel GMRES, which try to reduce communication costs. The results show that global communication in the inner products degrades performance, so that increasing the processor count above ~200 results in slowdown rather than a speedup of the algorithm.

In [9] a number of QR decomposition schemes is discussed and a block version of Householder orthogonalization is implemented on the GPU. The speedups approach 4.9x, but only for very large problem sizes and again a lot of communications is involved, which prevents exploiting full potential of the GPU.

To date no completely parallel version of the GMRES was proposed. In every case the orthogonalization step requires huge communication overhead, despite several attempts to optimize the

algorithm. Therefore in this project another approach is proposed. Because communication between threads on the GPU is so inefficient, algorithms such as GMRES should be avoided. In fact all Krylov subspace methods exhibit the same communication problems. That this the main reason why in this project the Multigrid method will be used to completely solve the problem within desired accuracy, rather than being just a preconditioner for other methods. It is also worth to notice that if a standard matrix format is used, for example the CRS – compressed row storage format, then it is impossible to fully benefit from specific non-zero value arrangement in a given problem. In this project a custom matrix format will be used, so that a well known shape of the sparse matrix could be exploited with maximum efficiency. Any communication between GPU threads will be avoided, as it degrades performance much more than on a CPU.

Theory

Navier-Stokes equations are known to produce good predictions of the behavior of a Newtonian fluid, which closely match observations. The equations however are usually impossible to solve analytically and numerical solutions are limited by available computational power to very small domain sizes. According to [10] when we want to simulate the behavior of sea water, we can take several assumptions which will greatly simplify the problem analytically, while still remaining accurate enough in the cases we want to study. We can ignore the fluid viscous effects and only express the conservation of mass, energy and momentum. The equations then simplify to Euler equations. Furthermore we can assume that the fluid is irrotational, which simplifies the problem even more resulting in potential flow described by the Laplace equation.

$$\frac{\partial^2 \Phi}{\partial x^2}+\frac{\partial^2 \Phi}{\partial y^2}=\nabla^2 \Phi=0$$ Laplace equation is 2D

Φ in the equation above is the scalar velocity potential function. When modeling the behavior of a fluid we want to learn the velocity of fluid particles at any given point in the domain. This results in a vector field, where a vector at any point represents the velocity. The same information can be represented by a scalar function Φ, which requires only one value at every point in the domain. The direction of the slope then represents the direction of the velocity vector and the slope steepness represents the magnitude of the velocity.

The free surface evolution is governed by:

$$\partial_t \zeta=-\nabla \zeta \cdot \nabla \tilde{\phi}+\tilde{w}(1+\nabla \zeta \cdot \nabla \zeta)$$

$$\partial_t \tilde{\phi}=-g\zeta-\frac{1}{2}(\nabla \tilde{\phi} \cdot \nabla \tilde{\phi}-\tilde{w}^2(1+\nabla \zeta \cdot \nabla \zeta))$$

To find $\tilde{w}$ and integrate these equations forward in time we need to solve the Laplace problem with a known $\tilde{\phi}$ at the free surface and boundary conditions defined below:

$$\phi=\tilde{\phi}, \quad y=\zeta$$
$$\nabla^2 \phi+\partial_{yy}\phi=0, \quad -h<y<\zeta$$
$$\partial_y \phi+\nabla h \cdot \nabla \phi=0, \quad z=-h$$

Using potential flow to describe fluid behavior works well for small amplitude non-breaking waves.

The model becomes inaccurate, when waves reach breaking point. Using potential flow approach it is also impossible to model turbulent flow around obstacles. Fortunately most waves at the sea satisfy the assumptions, that is the amplitude of these waves is typically much smaller than the still water depth. Shallow water regions are an exception to that, as well as areas with extreme weather conditions. Potential flow model can be applied to a very large domain and it can be combined with another model, which takes into account viscous forces at places where these forces start to play an important role (close to solid boundaries). This project only takes into account the potential flow model.

One particular problem in fluid dynamics, is that even given a rectangular container, the free surface at the top can have an arbitrary shape, which does not map well to the rectangular grid and makes the boundary condition time dependent,because the surface is in constant motion. In order to solve that problem we map the (x, y) domain into the (x, σ) called the sigma domain as described in [1]. In sigma space the domain remains rectangular regardless of the free surface shape and can be easily mapped to a rectangular grid. The following transformation is used to the problem to time-invariant σ -space:

$$\sigma \equiv \frac{y+h(x)}{\zeta(x,t)+h(x)} \equiv \frac{y+h(x)}{d(x,t)}$$

The Laplace problem then becomes:

$$\Phi = \tilde{\phi}, \quad \sigma = 1$$
$$\nabla^2 \Phi + \nabla^2 \sigma (\partial_\sigma \Phi) + 2 \nabla \sigma \cdot \nabla (\partial_\sigma \Phi) + (\nabla \sigma \cdot \nabla \sigma + (\partial_y \sigma)^2) \partial_{\sigma\sigma} \Phi = 0, \quad 0 \le \sigma \le 1$$
$$(\partial_y \sigma + \nabla h \cdot \nabla \sigma)(\partial_\sigma \Phi) + \nabla h \cdot \nabla \Phi = 0, \quad \sigma = 0$$

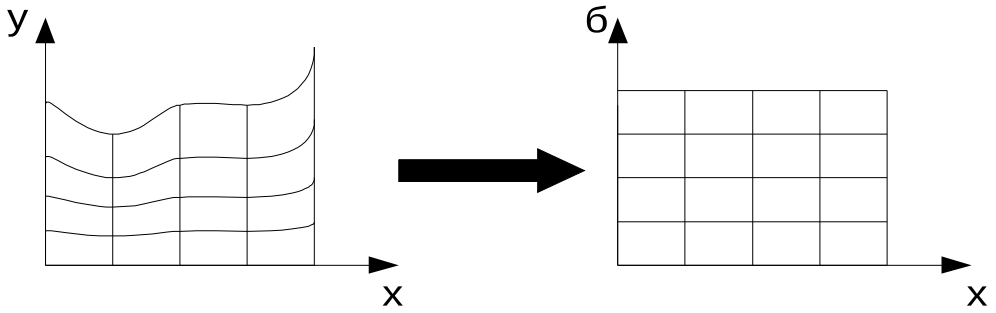


Figure 1: Physical domain to sigma domain transformation

The grid has a constant rectangular shape, so we do not need to have to maintain a grid, in which points sometimes appear inside the fluid domain and sometimes are outside. At any time we can go back to the physical domain by reversing the transformation.

If we assume that water waves have an amplitude much smaller than still water depth $(\zeta \ll h)$, then nonlinear terms in the problem described above can be omitted. The whole problem then becomes the transformed linearized Laplace problem:

$$\Phi_{xx} + \frac{1}{h^2}\Phi_{\sigma\sigma} = 0$$

Solving the transformed linearized Laplace problem is the most demanding part of the algorithm in terms of CPU resources and this project will aim at removing this computational bottleneck by using the power of GPU.

Finite difference method

In a general case a direct solution the the partial differential equation, which arises in the problem, is impossible. However there are several methods, which can approximate the exact solution. In this project the well known finite difference method is used to approximate derivatives in the equation. Given the definition of the derivative:

$$f'(a)=\lim_{h\to 0}\frac{f(a+h)-f(a)}{h}$$

Definition of the derivative

We can create a good approximation of the derivative, by fixing "h" to be equal to some small value. Of course the smaller "h" gets, the better the approximation will be. This is the basis of the finite difference method. The whole continuous function domain is then divided into a discrete grid of points. The more points are in the grid, the smaller distances between these points get and the better approximation is produced.

Using the definition above we can calculate approximations to first-order derivatives of a two dimensional function u:

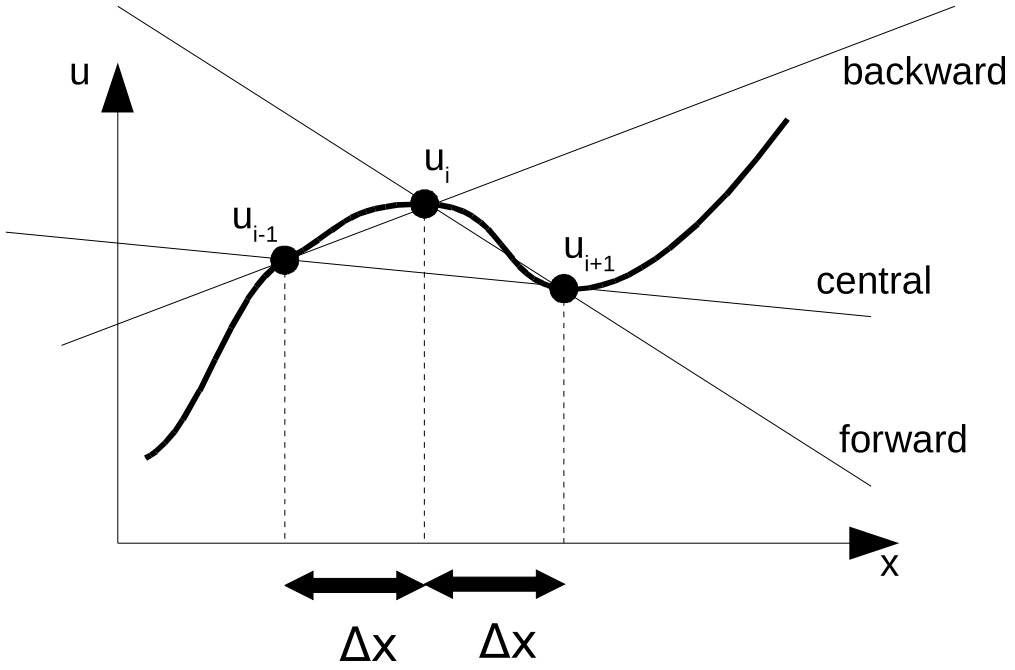


Figure 2: Approximation of first-order derivatives

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1}-u_i}{\Delta x}$$

Forward difference

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_i-u_{i-1}}{\Delta x}$$

Backward difference

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{1}{2}\left(\frac{u_{i+1}-u_i}{\Delta x}+\frac{u_i-u_{i-1}}{\Delta x}\right)=\frac{u_{i+1}-u_{i-1}}{2\Delta x}$$

Central difference

The central difference is simply the mean value of the forward and backward differences. The truncation error of the central derivative is $O(\Delta x)^2$ as opposed to $O(\Delta x)$ for forward and backward differences.

Second-order derivatives can be derived from the first-order derivatives:

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i = \left[\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)\right]_i \approx \frac{\frac{u_{i+1}-u_i}{\Delta x} - \frac{u_i - u_{i-1}}{\Delta x}}{\Delta x} = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2}$$

Second-order derivative approximation

We can also approximate mixed derivatives in a similar way:

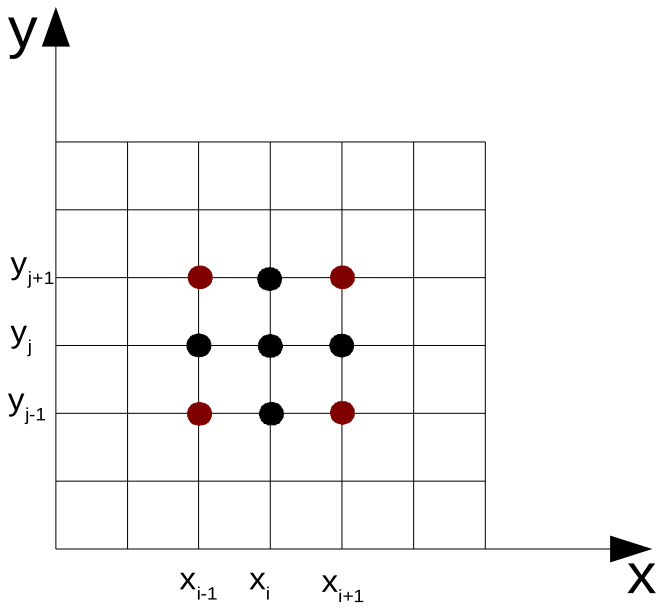


Figure 3: Grid points used for derivative calculations

$$\frac{\partial^2 u}{\partial x \partial y} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial y}\right) = \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial x}\right)$$

$$\left(\frac{\partial^2 u}{\partial x \partial y}\right)_{i,j} \approx \frac{\left(\frac{\partial u}{\partial y}\right)_{i+1,j} - \left(\frac{\partial u}{\partial y}\right)_{i-1,j}}{2\Delta x}$$

$$\left(\frac{\partial u}{\partial y}\right)_{i+1,j} \approx \frac{u_{i+1,j+1} - u_{i+1,j-1}}{2\Delta y}$$

$$\left(\frac{\partial u}{\partial y}\right)_{i-1,j} \approx \frac{u_{i-1,j+1} - u_{i-1,j-1}}{2\Delta y}$$

$$\left(\frac{\partial^2 u}{\partial x \partial y}\right)_{i,j} \approx \frac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{4\Delta x \Delta y}$$

Mixed derivatives approximation

In the continuous domain we want to find a function, which will correctly describe the velocity potential over the whole domain. After moving to the discrete grid of points the whole problem transforms into a linear system of equations with amount of unknowns equal to the number of points in the grid. This is done by writing one equation for every grid point and replacing the derivatives by their finite difference approximations as defined earlier in this document. If we look at the definition of the finite difference, we can see that for every point in the grid we must have neighboring points defined. At domain boundaries we run into trouble, because there is no point defined outside of the domain. As suggested in [1] an additional layer of ghost points is introduced outside the domain. The additional layer is added at the bottom of the domain and at the domain walls to solve this problem.

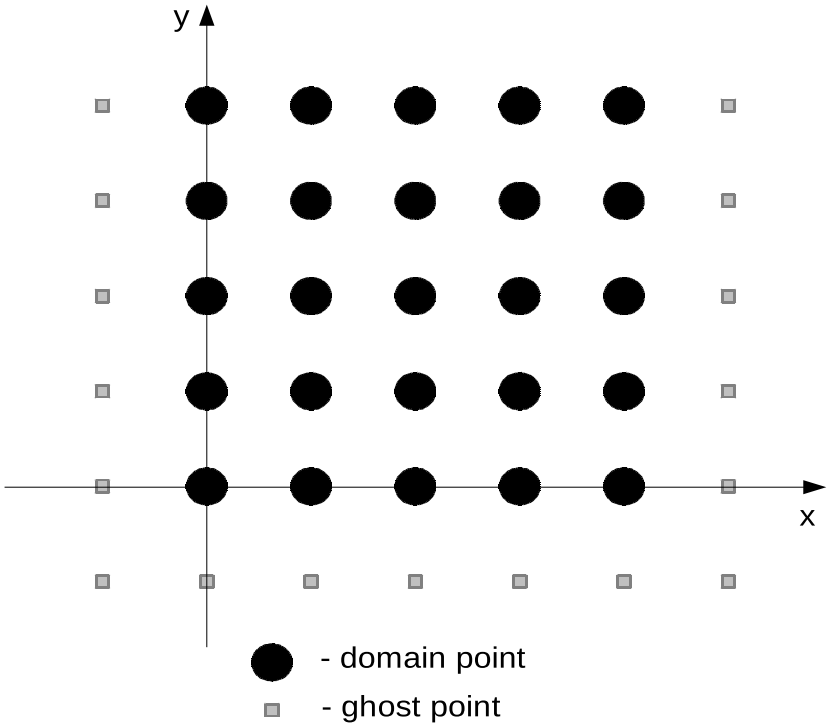


Figure 4: Domain and ghost points arrangement

The linear system

The finite difference method produces a system of linear equations, which needs to be solved. In general current hardware limitations prevent finding a solution to very large linear systems – ones which have millions or tens of millions unknowns. Fortunately the linear system, which arises in the finite difference method is usually very sparse, that is most entries in matrix A are equal to zero. This fact can be used to greatly optimize the algorithm and obtain the solution for problem sizes otherwise beyond our reach.

For the linear system $Ax = b$ the matrix coefficients can be efficiently calculated using the algorithm presented in [11]. Let us analyze how the matrix A would look like for a linearized Laplace problem in 2D on a 3 x 3 grid.

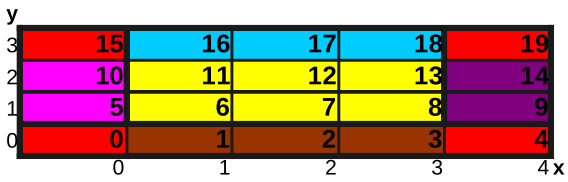


Figure 5: 3 x 3 grid with ghost layers and colored boundary conditions

On the figure above we can see a 3 x 3 domain of points with ghost layers at the walls and at the bottom. Hence the total grid dimensions are 5 x 4. The domain itself is marked with yellow, while the surface is marked with blue. We can also notice the violet points used to impose the boundary condition at the walls, as well as brown points used for the bottom boundary conditions. For this setup the matrix A and the vector b would look like this:

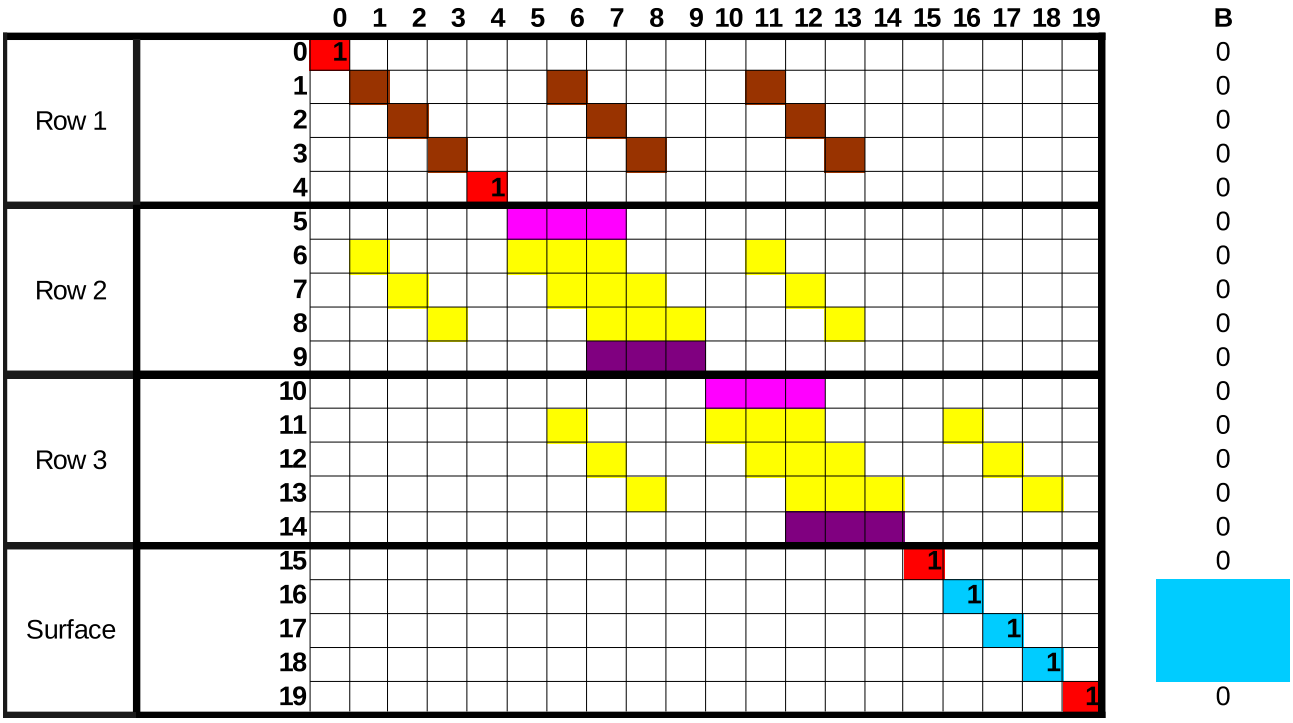


Figure 6: Sparsity of Matrix A and vector b in the Laplace problem

Only colored entries have non zero values. All other entries always are equal zero. We can notice that in this setup the matrix is very sparse. Most non zero entries lie either on the diagonal or at well defined places close to the diagonal. We can also notice that the last row contains only ones on the diagonal. That is to reflect the fact, that we already know from the problem definition the values at the free surface. It is also worth to notice the values marked in red, which correspond to grid corners. These values are set to 1 just to avoid making a singular matrix. We are not interested in solution at these points anyway, since they lie outside of the domain.

For simplicity and to reduce memory consumption we assume that the points are distributed uniformly in the grid. This has an important implication. Because the distance between neighboring pointa is the same everywhere, the finite difference coefficients also remain the same for every point in the grid. Hence not only we do not need to store the matrix in a dense format, we do not even have to use any

generic sparse matrix format. Instead we just store few values describing the coefficients – that is 5 values for points in the middle of the domain (yellow) and 3 points for wall and bottom points. Hence the memory usage for the matrix A is negligible and remains constant regardless of the problems size. The memory usage could be reduced even further, but the code is written with the possibility of using a non-uniform grid in mind. Using a non-uniform grid will either need more memory or more computation to calculate coefficients at a given point, however the benefit is that we can allocate more points where velocity potential function is suspected to be more variable – that is typically near the surface and used less points where the function remains almost constant.

Solution

The linear system $Ax = b$ can be solved directly using one of the well known methods, such as Gaussian Elimination. This approach works very well for small problem sizes. However the aim of this project is to work with a problem size of the order of several millions of unknowns and Gaussian Elimination with its computational complexity of n^3 quickly becomes impractical. It is possible to solve the problem in 2D using Gaussian Elimination, however in 3D the matrix A becomes too dense for the method to be effective.

Iterative methods

Iterative methods do not solve a linear system directly. Instead they try to approximate the solution by executing the algorithm several times. Each iteration gives a better approximation and the whole algorithm is terminated when the computed solution is accurate enough.

Jacobi method

There are several well known stationary iterative methods. Most popular methods include the Jacobi method, the Gauss-Seidel method or the successive over-relaxation method. These methods are described and analyzed in detail in [12]. In this project we will focus on the Jacobi method, since it is very straightforward and can be completely parallelized without any modifications. The method begins by splitting the matrix A into two components A = D + R, where D contains only elements lying on the diagonal and R contains all other elements from A. After that we can obtain the next iteration given results from the previous one using the following formula:

$$x^{(k+1)} = D^{-1}(B - Rx^{(k)})$$

Element wise the formula looks in the following way:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j} a_{ij} x_j^{(k)}\right)$$

Because in our setup we always use 3-point stencils in one dimension or 5-point stencil in two dimensions, the sum in the above equation always consists of either 3 or 5 elements. Note that the sparse matrix presented above has only 3 or 5 non-zero entries in each row (ignoring the surface and corner points). Thus the following optimizations are used in the implementation of the Jacobi method:

- the sum is replaced by 3 or 5 explicit subtractions depending on the unknown we want to solve
- the division is replaced by a multiplication of the inverse as divisions are expensive to calculate
- corner points represent the equation x * 1= 0 and lie outside of the domain, so they are ignored (Jacobi method is not performed at these points)
- matrix rows corresponding to the surface points closely resemble the identity matrix. For these points the equation takes the form x * 1 = B. Instead of solving this trivial equation we can inject the surface values directly into our solution vector X. Thanks to that the B vector becomes obsolete and does not have to be used at all

This yields an a very cheap method in terms of number of required operations. The memory usage is twice the size of the solution, because we need to remember the previous solution in order to calculate the new one.

Part of the CUDA Kernel implementing the Jacobi step. (A case for points in the middle of the domain).

```
if ( x >= 1 && x < LengthX - 1)
{
  newSolution[idx] = B[idx];
  newSolution[idx] -= solution[idx - LengthX] * MiddleCoefficients[middleIdx + 0];
  newSolution[idx] -= solution[idx - 1] * MiddleCoefficients[middleIdx + 1];
  newSolution[idx] -= solution[idx + 1] * MiddleCoefficients[middleIdx + 3];
  newSolution[idx] -= solution[idx + LengthX] * MiddleCoefficients[middleIdx + 4];

  // Calculatte (B-MX) / D
```

```
  newSolution[idx] *= MiddleCoefficients[middleIdx + 2];
}
```

In order to start the method we need an initial guess of the solution, which is typically X = 0.

Multigrid

Iterative methods like the Jacobi method described above, work very well at the beginning, but the rate of convergence deteriorates when the number of iterations becomes large. This means that we are left with an error, which is very expensive to eliminate if we want to continue using the Jacobi method. If we perform Fourier analysis of the error, we will notice that high-frequency components are quickly removed, however low-frequency components remain even for a very large number of iterations. In order to solve this problem the Multigrid method is used [12].

The Multigrid method begins with executing few steps of the Jacobi method or some other iterative method. Usually it is enough to execute just one such iteration, which in the literature is called either the relaxation or the smoothing step. After that the whole problem is redefined on a grid with fewer domain points. On the coarser grid the low frequency error components become high frequency components and can be reduced faster by the application of the underlying iterative method. Not only do we get a better convergence rate on the coarser grid, but also each smoothing step takes less time to perform, because there are fewer domain points. Having a better approximation on the coarser grid we now need to use it to correct the solution on the finer grid.

In general we do not need to restrict ourselves to use just two grids (a coarser and a finer one). We can use an arbitrary number of grids as will be shown later.

Grid size and shape setup

In this project a uniform rectangular grid is used. All grids in the Multigrid algorithm are defined in the following way:

A grid of level n has dimensions: $(2^n+1)\times(2^n+1)$

Including ghost layers a grid of level n has dimensions: $(2^n+3)\times(2^n+2)$

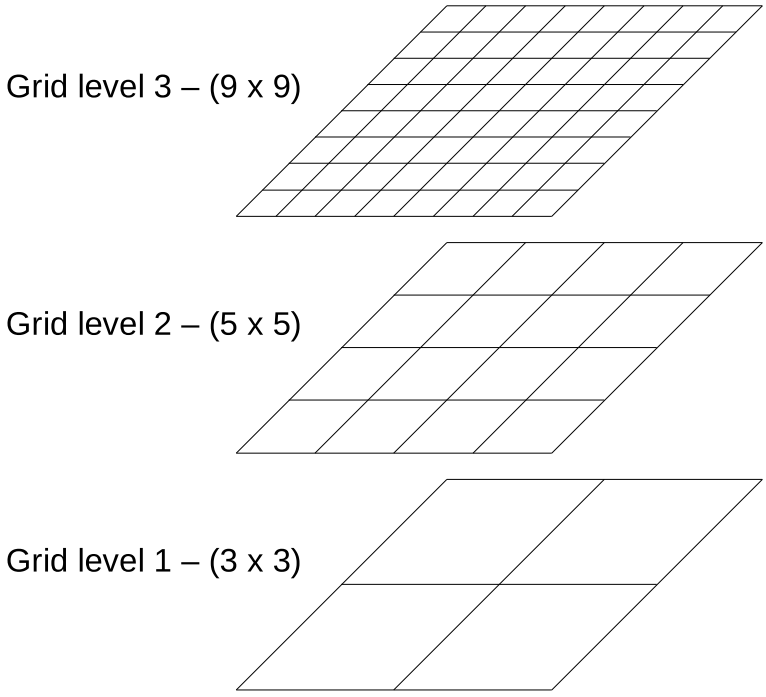


Figure 7: Grid levels

In general a grid can have arbitrary size, but having sizes defined as above makes it easier to define restriction and prolongation operations, which are used to transfer data between grid levels. The smallest grid level is 1. Grid level 0 would contain only 1 point and it would make little sense, as one would have to impose all boundary conditions and the kinematic condition on that single point.

The highest grid level is limited by memory size on the hardware. The table below shows memory consumption details:

	Space needed to store at least the solution	Accumulated space needed for all grids up to maximum level with 4 grids per level	
Level	**MB**	**MB**	
1	0.0002	0.0006	
2	0.0003	0.0019	
3	0.0008	0.0052	
4	0.0026	0.0157	
5	0.01	0.0520	
6	0.03	0.1870	
7	0.13	0.7067	
8	0.51	3	
9	2	11	
10	8	43	
11	32	171	
12	128	684	
13	512	**2733**	← **Actual limit in this project**
14	**2049**	10928	←**Tesla C1060 theoretical limit**
15	8193	43701	

Table 1: Memory consumption

The table above assumes double precision, each value using 8 bytes of memory. Any algorithm executed on Tesla – no matter how memory efficient – will need to have enough room to store at least the solution in the memory. Having that in mind we can see, that Tesla C1060 with 4GB of memory on board can handle at most a grid of level 14.

In this project we actually need to store 4 vectors per grid level: two solution vectors required by the Jacobi method, the B vector, and the residual vector. Further more we need to allocate space not only for the finest grid, but also for all lower grid levels. The total memory consumption is shown in the right column of the table. We can observe, that Tesla C1060 can handle the Multigrid method with the finest grid level up to 13. In order to solve the problem at grid level 13, one needs to find a solution to a linear system with $(2^{13}+3)*(2^{13}+2)=67{,}149{,}830$ unknowns.

Prolongation

In order for the Multigrid to work we need to define an algorithm to move data from a coarser grid to a finer grid. This is the prolongation step.

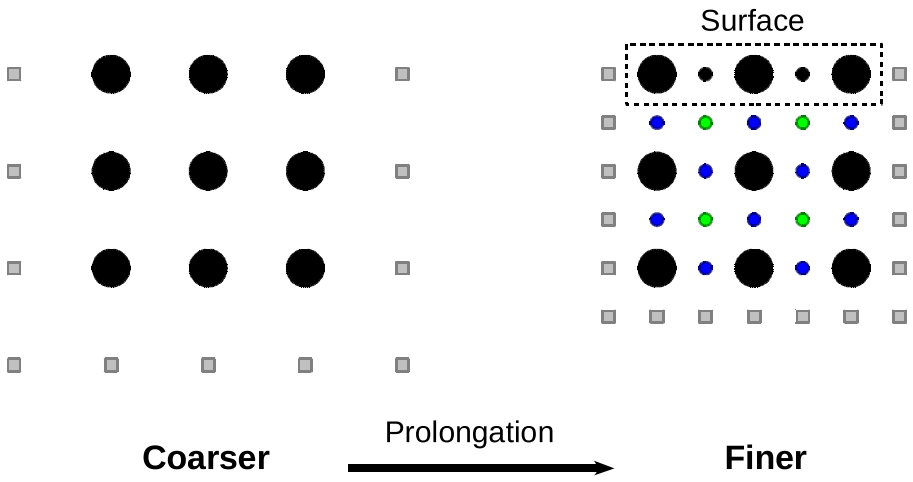


Figure 8: Prolongation from a 3 x 3 grid to a 5 x 5 grid

The procedure:

1. Direct injection - On overlapping points directly copy the values from the coarser grid to the finer grid (do not perform this on the surface, because we have an exact solution there)
2. Linear interpolation - On points lying between two overlapping points (blue) – calculate the mean value from the two neighboring points
3. Linear interpolation - On mid-points (green) calculate the mean value from the 4 black neighboring points

The steps above handle all points within the domain. However the ghost points still remain undefined. Experiment has shown that not updating the ghost point layer (i.e. leaving the values from the previous step) causes the whole Multigrid algorithm to diverge to infinity for large problem sizes. Therefore we need to obtain a good initial guess for points lying at the ghost layer. In this project this is done through extrapolation of domain points onto the ghost layer:

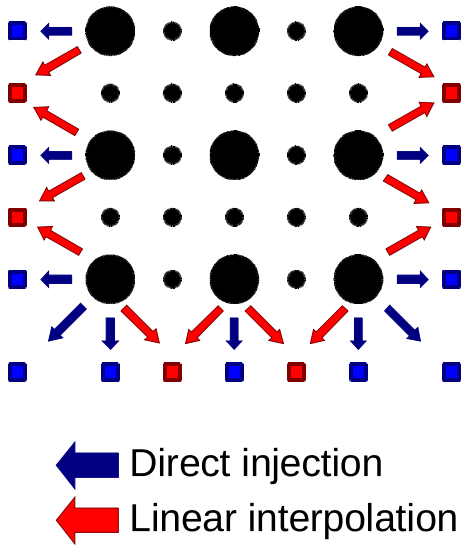


Figure 9: Ghost layer extrapolation

Notice that all ghost points on the finer grid take their values only from points on the coarser grid (bigger points), which causes the whole prolongation process to use the coarse grid as read-only memory and the fine grid as the write-only memory, which completely avoids any memory bank collisions and allows all reads and writes to be automatically coalesced – an important property as far as GPU performance is concerned.

Restriction

Restriction allows us to move from a finer to a coarser grid. This can be done in various ways - the simplest being direct injection. However a more accurate approach is to perform full weighting [12]. Every point on the coarse grid except the surface takes its value from 9 neighboring points on the fine grid. The weights at each neighboring point are the following:

$$\begin{matrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{matrix}$$

Figure 10: Weights in restriction

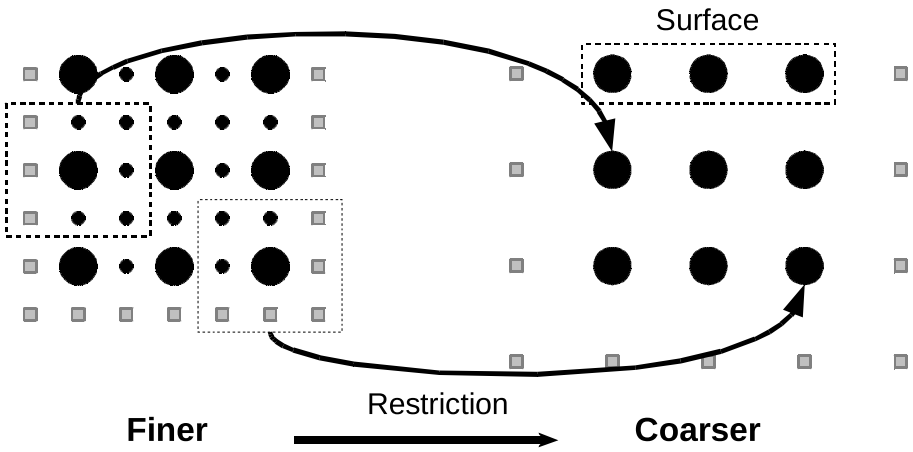


Figure 11: Restriction from a 5 x 5 grid to a 3 x 3 grid (full weighting for two example points is shown as arrows)

Notice that the surface is never updated by neither the prolongation nor restriction operations. We already know the exact solution at the surface points from the initial condition, so we do not want to overwrite these values.

V-cycle

There are various schemes describing different ways of visiting grid levels [12]. In our case the standard v-cycle is used. The algorithm is defined as follows:

1. Perform a number of smoothing steps on the finest grid
2. Calculate the residual $r = b - Ax$
3. Restrict residual to a lower grid level (store the result in the B vector) – this is equivalent to defining the error equation $Ae = r$
4. Set initial guess of the error to zero
5. If we are on the lowest level then directly solve the linear system. Other wise perform steps 1-4 for the coarser grid
6. Prolong the solution calculated on the coarser grid and use it to correct the finer grid.
7. Perform a number of smoothing steps on the finest grid

For simplicity 20 Jacobi steps are performed on the coarsest grid instead of directly solving the system. Experiment has shown that the Jacobi method is extremely fast on small grid sizes and 20 iterations produces enough accuracy, so for smaller grids this is equivalent to a direct solver.

Source code for the v-cycle:

```
        public void GPUVCycle(int lowestLevel, int highestLevel, int
numberOfPreSmoothingSteps, int numberOfPostSmoothingSteps)
       {
           for (int level = highestLevel; level > lowestLevel; level--)
           {

               for (int j = 0; j < numberOfPreSmoothingSteps; j++)
               {
                   GPUJacobiRelax(level);
               }

               GPUCalculateResidual(level);
               GPURestrictResidualToB(level);
               GPUSetInitialGuessToZero(level - 1);
           }

           // solve the problem at lowest level (just use many Jacobi steps)
           // Can be replaced by a direct solver
           for (int i = 0; i < 20; i++)
           {
               GPUJacobiRelax(lowestLevel);
           }

           for (int level = lowestLevel+1 ; level <= highestLevel; level++)
           {
               GPUProlongSolution(level - 1);
               for (int j = 0; j < numberOfPostSmoothingSteps; j++)
               {
                   GPUJacobiRelax(level);
               }
           }
       }
```

Full Multigrid Cycle

The rate of convergence of Multigrid strongly depends on a good initial guess. Therefore instead of using X=0 as the initial guess and performing a number of v-cycles starting on the highest grid level, we can instead begin at the lowest level – calculate a good approximation of the solution on the coarse grid and use it as a quite good initial guess on the finer grid. This results in a zig-zag pattern as we perform a v-cycle for ever higher grid level.

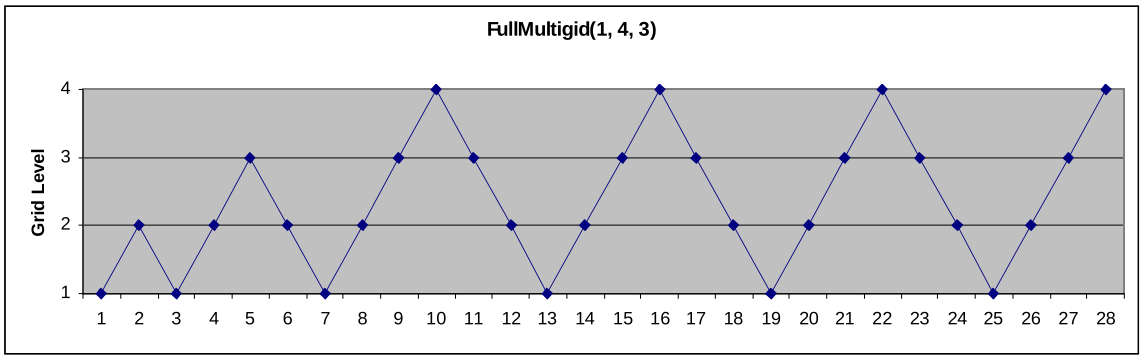


Figure 12: An example showing how grid levels are visited. Here we perform a total of 3 v-cycles on the finest grid level (2 more than the standard FMV)

The implemented algorithm extends the standard Full Multigrid as described in [12] by an arbitrary number of additional v-cycles. In the example above the algorithm begins at grid level 1 and uses level 4 as the finest level. A total of 3 v-cycles is performed on the finest grid level. Standard Full Multigrid Cycle gives a good enough approximation to be used as a preconditioner for other methods, for example one of the Krylov subspace methods. If we want to solve the problem with Multigrid however, we need to perform an additional number of v-cycles to achieve desired accuracy. This scheme can be thought of as a Multigrid solver with a Full Multigrid preconditioner.

Source code for the Full Multigrid cycle:

```
        public void GPUFullMultigidCycle(int lowestLevel, int highestLevel, int
numberOfVcycles, int numberOfPreSmoothingSteps, int numberOfPostSmoothingSteps)
        {
            for (int i = lowestLevel; i < highestLevel; i++)
            {
                GPUDefineProblem(i);
                GPUVCycle(lowestLevel, i, numberOfPreSmoothingSteps,
                        numberOfPostSmoothingSteps);
                GPUProlongSolution(i);
            }
            GPUDefineProblem(highestLevel);
            for (int i = 0; i < numberOfVcycles; i++)
            {
                GPUVCycle(lowestLevel, highestLevel, numberOfPreSmoothingSteps,
                        numberOfPostSmoothingSteps);
            }
        }
```

Implementation

Two versions

Two versions of the software were written. One version is designed to be run on CPU only, while the other is going to use help of the GPU. The rationale for writing two versions doing the same thing is the following: Debugging GPU code is either impossible or very troublesome. If the code is running on the GPU, the CPU and a debugger has no direct access to it. CUDA supports an emulation mode, in which the CPU emulates the GPU behavior. However from my previous experience debugging emulated code proves to be incredibly troublesome. In particular it turns out, that in some cases the emulated code is not executed in the same way as the GPU code. Further more emulated code execution speed is extremely slow. CUDA was designed quite recently and is not yet well integrated with existing IDEs (Integrated Development Environments) – in particular with MS Visual Studio which was mostly used for the development. On the other hand CPU-only programs do not share these problems at all. The multi-grid method, while conceptually straightforward, becomes much more complex when it comes to implementation details and data bookkeeping. A simple typo error can linger in the code and produce completely incorrect results. Therefore it is imperative to be able to have a look at the inner workings of the algorithm at any time. CUDA code does not support that yet.

The purpose of the CPU version is to design the algorithm, which works and to prove that it produces correct results. Most debugging and testing has been done using the CPU version first. Another goal is to make the code as easy to understand as possible. It is also important that all data structures should flexible enough to enable very easy experimenting with the code. The execution speed is not a priority. Instead easiness of error checking and debugging is the priority, hence the CPU version uses only one core and its data structures are quite slow (due checking for exceptions, etc.) compared to a raw array implementation.

The priority for GPU version on the other hand is speed. Once it is proven, that the algorithm is conceptually correct, it can be ported to CUDA to increase performance.

Technology choice

An important design decision is the choice of technology. A language of choice is C#. There are several reasons for this choice. It is a strongly typed language. It features automatic garbage collection.

Memory access is managed. It is completely object oriented. It compiles to machine code using just-in-time compilation. Furthermore it has a very strong IDE support, including: refactoring, extracting code into methods, auto-completion and many more. All these arguments put together eliminate any competition completely. A possible issue with C# and .NET is that it is a proprietary technology owned by Microsoft. The Tesla machine used for testing the solution is running Linux on the other hand. In Linux environment MONO framework is used instead of .NET and MonoDevelop is used as the IDE. In order to access CUDA from C# - the CUDA.NET library is used. Another benefit of the decisions made above is, that the source code has the 'out-of-the-box' property, which is also one of the goals for this project. This means that it can be distributed to anyone and very little needs to be to to compile and execute the code. One only needs to install CUDA SDK to be able to run the code. There is no need to specify additional include paths, external library locations, custom build rules and set up thousands of dependencies as in the case of C++ for example.

To sum up:

- Linux
  - Novell MONO framework
  - CUDA.NET
  - IDE: MonoDevelop
- Windows
  - Microsoft .NET Framework
  - CUDA.NET
  - IDE: Microsoft Visual Studio

Data structures

Notation

A set of domain points is typically arranged in a rectangular grid, hence a natural way to store function values at these points is to put them into a matrix. We can then access these points by specifying the i-

th row and j-th column of the matrix. However the matrix we are dealing with is not just an abstract mathematical construct. It represents the actual alignment of points in the physical domain. All points in a 2d space are accessed by specifying a pair of Cartesian coordinates (x, y). This is where we run into notation issues, which is confusing and could be the source of potential implementation errors. The matrix notation (i, j) is incompatible with the Cartesian notation (x, y). For example a matrix entry (0, 0) represents the top left corner of the matrix, while the same point (0, 0) in the physical domain represents a bottom left corner in a typical arrangement.

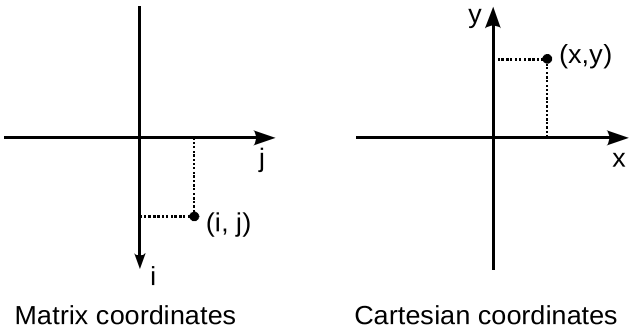


Figure 1 - comparison of coordinate systems

Differences: order of axes, direction of vertical axis

To avoid confusion and to take into account, that every matrix entry represents a point in a 2d space, all matrix elements will be accessed by Cartesian coordinates too.

Furthermore sometimes it is more convenient to treat the set of domain points as a 2-dimensional grid and sometimes it is more useful to treat it as a 1-dimensional array. For example, when we want to solve a system of linear equations $Ax = b$, the vector x is considered to have only one dimension, but at the same time each element in x also corresponds to a two-dimensional point in our domain. In addition to that, the method includes putting a layer of ghost points at the bottom and at the walls of the domain. These points lie beyond the domain and are sometimes included in the indexing and sometimes not. For example, if we want to read our final solution, it is more convenient to use an indexing, which excludes the ghost points.

Data indexing comparison:

	6	7	8	
	3	4	5	
	0	1	2	

Figure 13: Indexing a 3 x 3 domain in 1d

y

	(0, 2)	(1, 2)	(2, 2)	
	(0, 1)	(1, 1)	(2, 1)	
	(0, 0)	(1, 0)	(2, 0)	

x

Figure 14: Indexing a 3 x 3 domain in 2d

15	16	17	18	*19*
10	11	12	13	*14*
5	6	7	8	*9*
0	*1*	*2*	*3*	*4*

Figure 15: Indexing a 3 x 3 physical domain including the ghost layer, effectively making it a 5 x 4 data structure

y

(0, 3)	*(1, 3)*	*(2, 3)*	*(3, 3)*	*(4, 3)*
(0, 2)	*(1, 2)*	*(2, 2)*	*(3, 2)*	*(4, 2)*
(0, 1)	*(1, 1)*	*(2, 1)*	*(3, 1)*	*(4, 1)*
(0, 0)	*(1, 0)*	*(2, 0)*	*(3, 0)*	*(4, 0)*

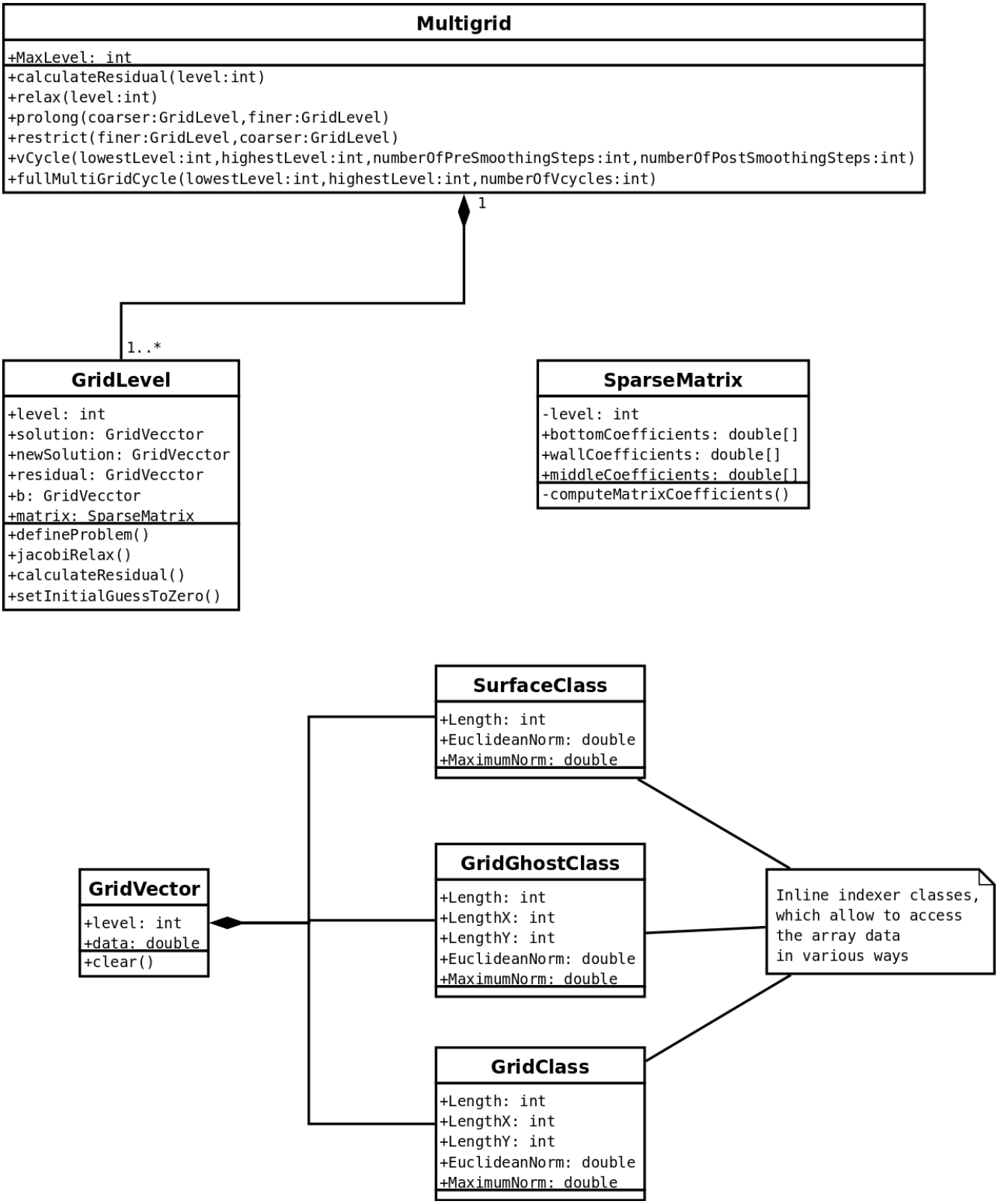
x

Figure 16: Indexing domain with ghost layers in 2d

	0	1	2	

Figure 17: Indexing surface points of a 3 x 3 physical domain

Class diagram (CPU version)



GridVector class

The class which actually holds data is the GridVector class. The class behaves as a hybrid between a vector and a matrix. Data is stored in one array of doubles. However this data can be accessed in different ways, depending on what is more convenient. In our example setup we have a two-dimensional domain. The most convenient way to access this data is to treat it as a two dimensional array. On the other hand after the linear system of equations is defined, we need to operate on the one-dimensional vector X.

The GridVector class has 3 helper classes defined inside: SurfaceClass, GridGhostClass and GridClass. These classes hide the bookkeeping details from the developer, so that cryptic calculations are avoided. Additional helper classes are neccessery to implement indexed properties, as that feature is not available in C# until C# 4.0 is released. This approach proved to be quite inefficient during testing the CPU version of the algorithm. However thanks to this approach instead of accessing an i-th point in the domain like this:

```
data[LengthX * ((i / LengthX) + 1) + 1 + (i % LengthX)];
```

One can access the vector like this:

```
GridVector vec;
vec.Grid[i];
vec.Grid[x, y];
```

To access the domain together with the ghost layers:

```
vec.GridGhost[x, y];
vec.GridGhost[i];
```

One can also easily access the surface points, which is especially useful for setting the initial condition of the problem:

```
vec.Surface[i];
```

All subclasses listed above also have a set of useful properties.

```
vec.Grid.LengthX; // amount of points in horizontal direction
vec.Grid.LengthY; // amount of points in vertical direction
```

```
vec.Grid.Length; // total amount of points in the domain
vec.Grid.EuclideanNorm; // euclidean norm of the vector
vec.Grid.MaximumNorm; // maximum norm of the vector
```

One can also use these properties for the GridGhost or for the Surface classes.

GridLevel class

GridLevel class represents a single grid level in the Multigrid algorithm. It contains everything necessary to define and solve the problem at a given grid level, that is a sparse matrix A and a set of 4 GridVectors:

- Solution – that is the X in our AX=B linear system
- newSolution – a second temporary solution vector, which is required by the Jacobi method
- B – the right-hand side of the linear system
- Residual – used for residual calculation

This class contains two important methods:

`jacobiRelax()` - performs a single step of the Jacobi method at the current level

`calculateResidual()` - calculates the residual at the current level

MultiGrid class

This class is responsible for the main logic of the Multigrid algorithm. It contains a set of grid levels and methods to operate on them:

relax(int level) – invokes a relaxation or smoothing step on a given level

prolong(int level) – prolongs the solution at a given level to the next one – the prolonged solution is automatically added to the solution, which existed on the higher level previously, hence it automatically includes the correction step

restrictResidualToB(int level) – performs restriction of the residual at a given level and stores the resutls in the B vector at the coarser level. This method is used to set up the error equation at the lower level

vCycle(int lowestLevel, int highestLevel, int numberOfPreSmoothingSteps, int

numberOfPostSmoothingSteps) – uses methods listed above and implements the actual v-cycle

fullMultigidCycle(int lowestLevel, int highestLevel, int numberOfVcycles) – performs the full multigrid method. It also allows to specify an arbitrary number of v-cycles to be executed afterwards

GPU Implementation

Conceptually the GPU implementation executes exactly the same algorithm as the CPU version. However because of several CUDA limitations, there is a number of significant differences in implementation.

Data structures

CUDA does not support advanced high-level constructs like classes. It is based on the C language and supports kernels which consist of simple functions. All data structures are simple arrays of doubles (or floats).

Overcoming function argument limitations

A kernel function in CUDA only supports a small (and constant) number of arguments. It is impossible to declare all necessary variables for each grid level in Multigrid. To overcome this problem data structures in the GPU version are merged. That is - there is only one X, A, B and R to represent the linear system AX=B (and R=B-AX) regardless of the number of grid levels. The size of each array is defined by the formula:

$\sum_{i=1}^{n} \left(2^i+3\right)*\left(2^i+2\right)$ where n is the finest grid level.

For example a grid of level 1 consists of 20 points indexed from 0 to 19. An element with index 20 will therefore be the first element of grid level 2. And so on.

Index values at which a given grid level starts have been precomputed and stored in a separate kernel function, which is later in-lined by the compiler during compilation.

Overcoming pointer operation limitations

In general we need two variables for the Jacobi step (named solution and newSolution in the source

code). After each Jacobi step the results are stored in the newSolution variable. After that the variables need to be swapped.

```
Temp = solution
solution = newSolution
newSolution = Temp
```

After the variables have been swapped the algorithm can continue. The swap operation is impossible, because all function arguments are passed by value to CUDA kernels.

Overcoming function size limitations

Whenever we use a function inside another function in CUDA, that function is in-lined by the compiler. It is therefore impossible to create very complex (long code) algorithms, which are executed by just one kernel. The desire was to create one kernel for the Full Multigrid method, which could be treated as a black box and internally perform all necessary steps. That proved to be impossible and the Full Multigrid as well as the v-cycle are governed by CPU code. This has only significance for very small problem sizes. For problem sizes we are interested in, the amount of CPU calls to GPU workload ratio is very small and this problem becomes insignificant.

Overcoming fixed startup parameters for a CUDA kernel

When the host issues a call to begin execution of a kernel, all details regarding the number of threads (block shape, number of blocks) must be defined in advance and remain static during kernel execution. This is the second reason, why the v-cycle and Full Multigrid logic is not implemented as one kernel. In the most extreme case once we launch 67 million threads to process the finest grid – all those threads must execute and waste resources and time on the coarsest grid, where only 20 threads are needed.

Testing

A number of unit tests for both CPU and GPU version has been implemented to verify correctness of every step of the algorithm and to automate testing. CPU unit tests use the standard unit testing functionality provided by Visual Studio, while GPU unit tests are implemented as standalone functions for easier portability.

Data structure tests:

- SurfaceTest()
- GridGhostTest()
- GridTest()

These tests verify, that high-level data structures work correctly (read and assignment operations place correct values in correct places in the underlying low-level array).

Convergence tests:

- JacobiRelaxTest()
- VcycleTest()
- FullMultigridCycleTest()

These tests verify 4 parameters after every iteration of the algorithm: error maximum norm, error euclidean norm, residual maximum norm and residual euclidean norm. All these parameters must decrease at any point in time or else the test will fail. Furthermore after all iterations done, the 4 parameters are checked if they reached desired low enough values. That is necessary to detect a case, where a change in the source code did not cause the algorithm to diverge, but slowed its convergence rate.

Grid Transition tests:

- prolongTestUniformFunction() - checks prolongation of a constant function
- prolongTestIncreasingValues() - checks prolongation of a 2d linear function
- restricTest() - tests a restriction of a 2d linear function
- restrictProlong() - performs restriction and then prolongation, which should be an identity operation, the grid is initialized according to the function f(x,y) = sin(x) + cos(y)
- prolongRestrict() - same as above, but executed in the opposite way

GPU grid transition tests perform the work twice: on the CPU and on the GPU. The results are then compared. GPU convergence tests execute only on the GPU and use the analytical solution for comparison. This is to reduce the test execution time.

Results

Setting up a Test Case

After computing solution to the Laplace problem, we face one serious issue. There is no way to confirm whether the computed solution is correct or not, because we have no idea how the solution is supposed to look like. In order to test, if the solver works correctly, we investigate a special case, where we know the exact value of the scalar velocity potential function at any point in the domain, including the surface. The function is given by the formula:

$$\phi(x,y) = -H\,l\,\frac{\cosh(k(y+h))}{\sinh(kh)} * \sin(\omega t) * \cos(k\,x)$$

$H - wave\,height$
$l - phase\,speed$
$k - wave\,number$
$h - still\,water\,depth$
$\omega - angular\,frequency$
$t - time$

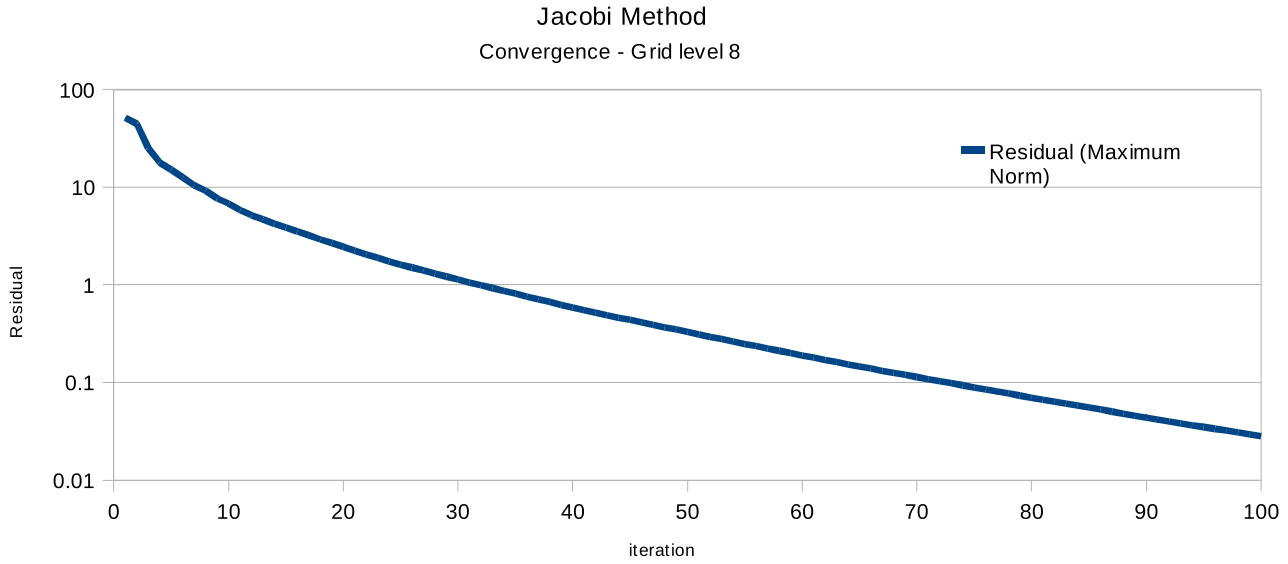
Knowing the analytical solution it is possible to measure the total error, that is the difference between the analytical solution and computed solution. The total error consists of three components: The truncation error of the finite difference method, the computation error coming from the fact that machine computations have limited precision and the approximation error coming from the iterative method used.

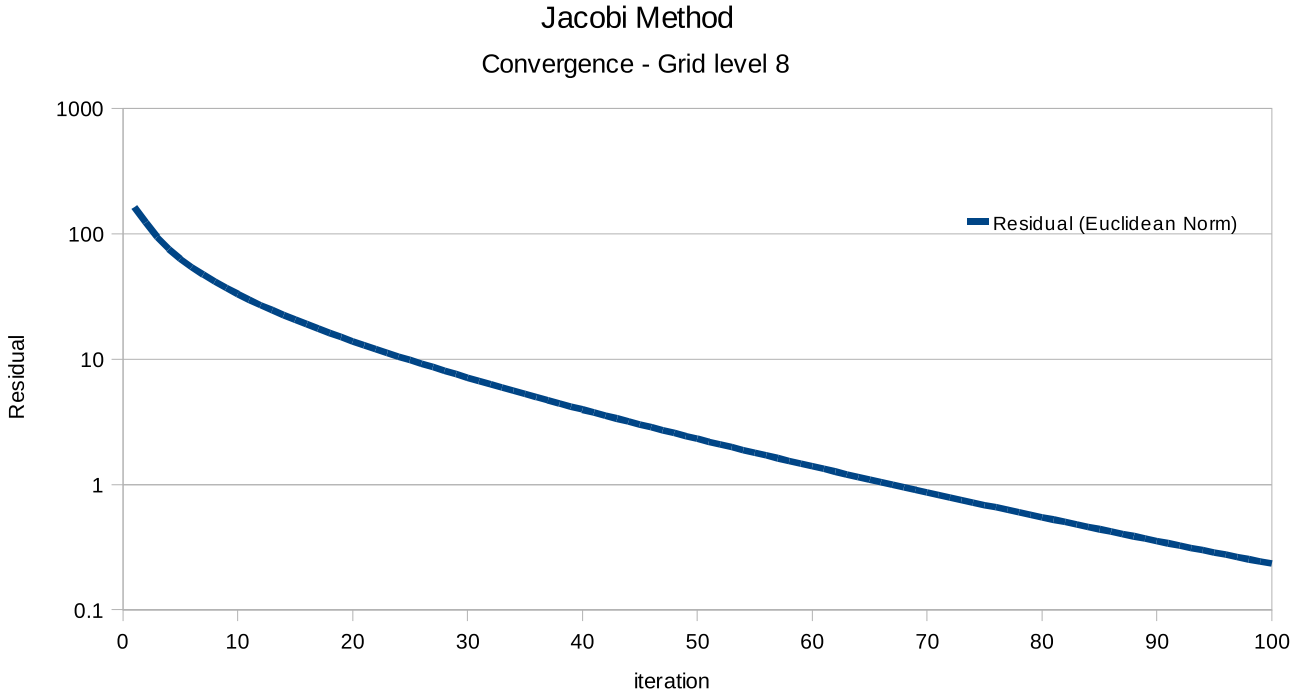
Verifying convergence

Several parameters are measured to prove that the implemented algorithm converges to the correct solution. In general a monotonically decreasing function which is bounded from below is guaranteed to converge. If the function would increase at any point, additional mathematical proof is needed to make sure that the function would not diverge. The euclidean and maximum norms of the residual are measured. Both norms are by definition non-negative, so the only remaining thing is to check if the norms decrease with iterations.

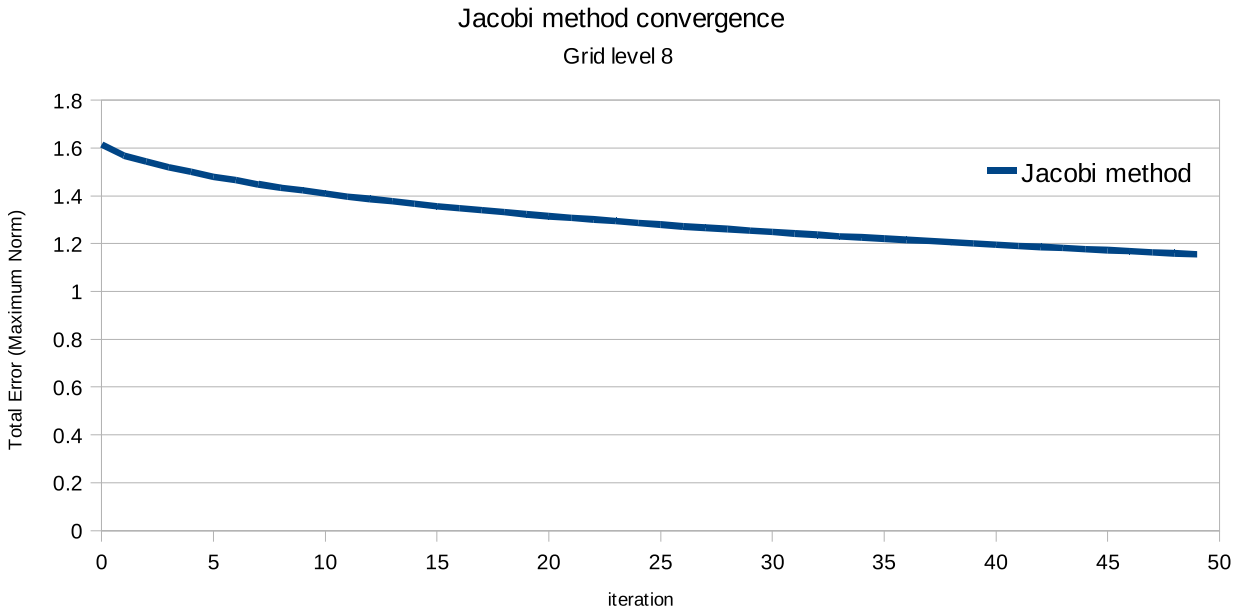
Jacobi method

Below are just two examples showing the behavior of the residual as the Jacobi method is performed. We can see that both euclidean and maximum norms are strictly decreasing. The graphs below show the results just for grid level 8, however automated unit tests described in the previous chapter ensure convergence for all supported problem sizes.





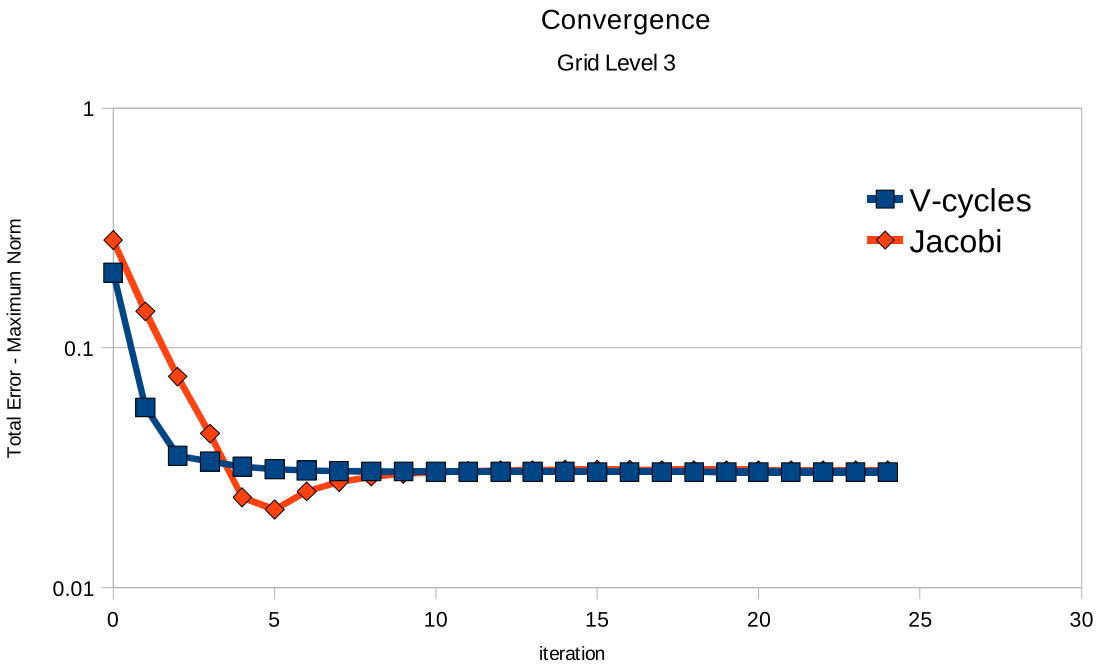
Of course the fact that the residual converges to 0 only proves the solution in our linear system converges. It does not prove that it converges to the correct value. The results presented below show that the total error, that this the difference between the analytical solution and computed solution is also decreasing.



An interesting behavior of the error maximum norm becomes visible for small problem sizes. In the graph below we can see, that at certain point the error increases slightly just before reaching the best convergence allowed by the discretization error. This can be explained in the following way. The solution can be seen as a point in n-dimensional space, where n is the number of unknowns. There are 3 kinds of solution vectors which are interesting in this case.

- analytical solution
- exact solution – which includes the discretization error of the finite difference method
- computed solution – which also includes approximation error of the iterative method

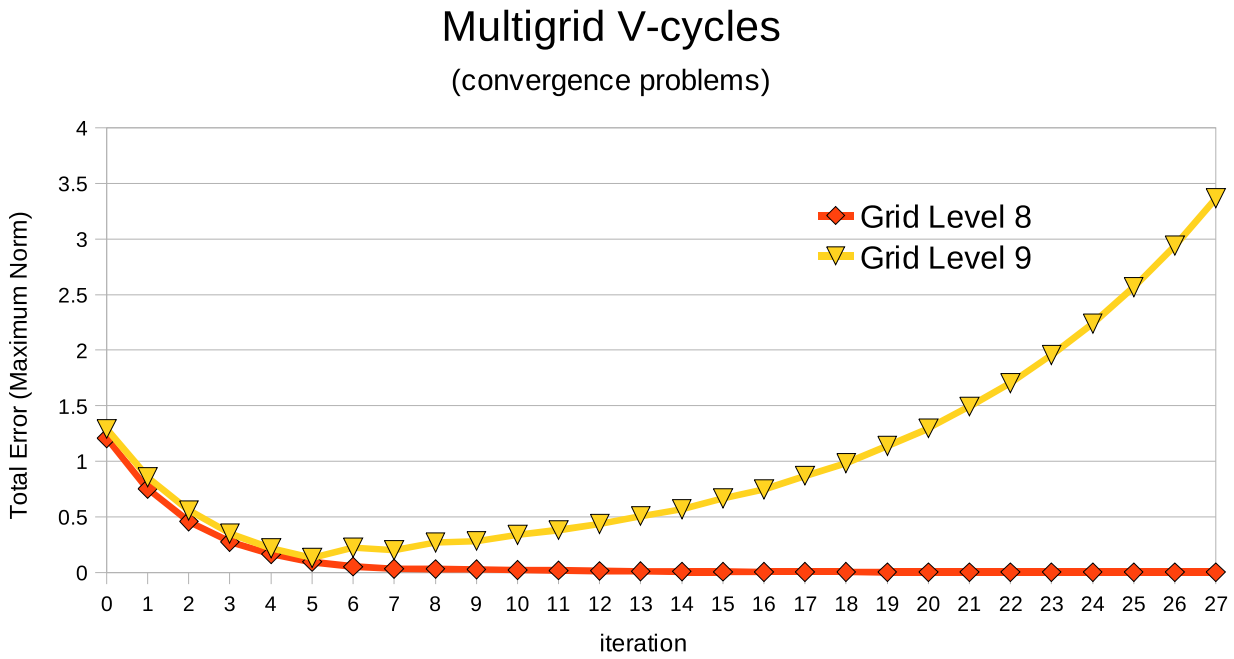
When an iterative method is used, consecutive iterations move the computed solution closer to the exact solution, but not the analytical solution, in the n-dimensional space. It may happen that the computed solution passes close to the analytical solution and then moves away from it on its way to the exact solution. That is the case in the example below. Of course in general case we have no idea what the analytical solution is, so we cannot stop the iterative method earlier. The graph below also shows the error in the Multigrid method for comparison.



Multigrid

Early divergent case

An interesting result comes from a very early version of the algorithm.



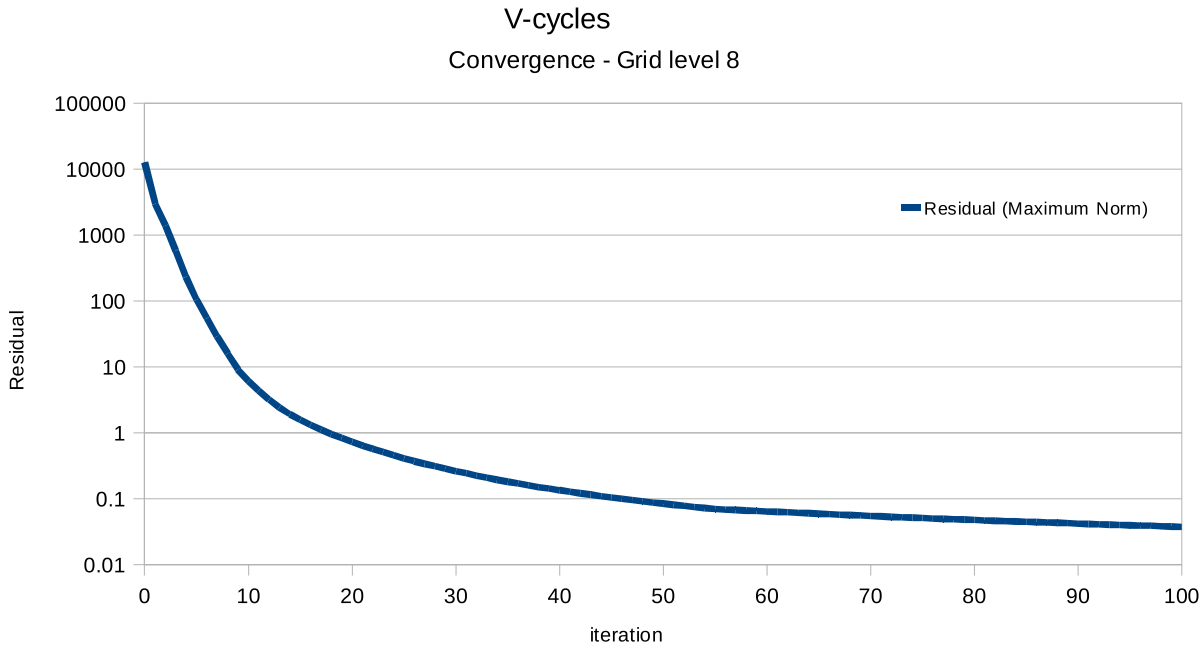
We can see here that for small problem sizes (up to grid level 8) the solution converges, however for

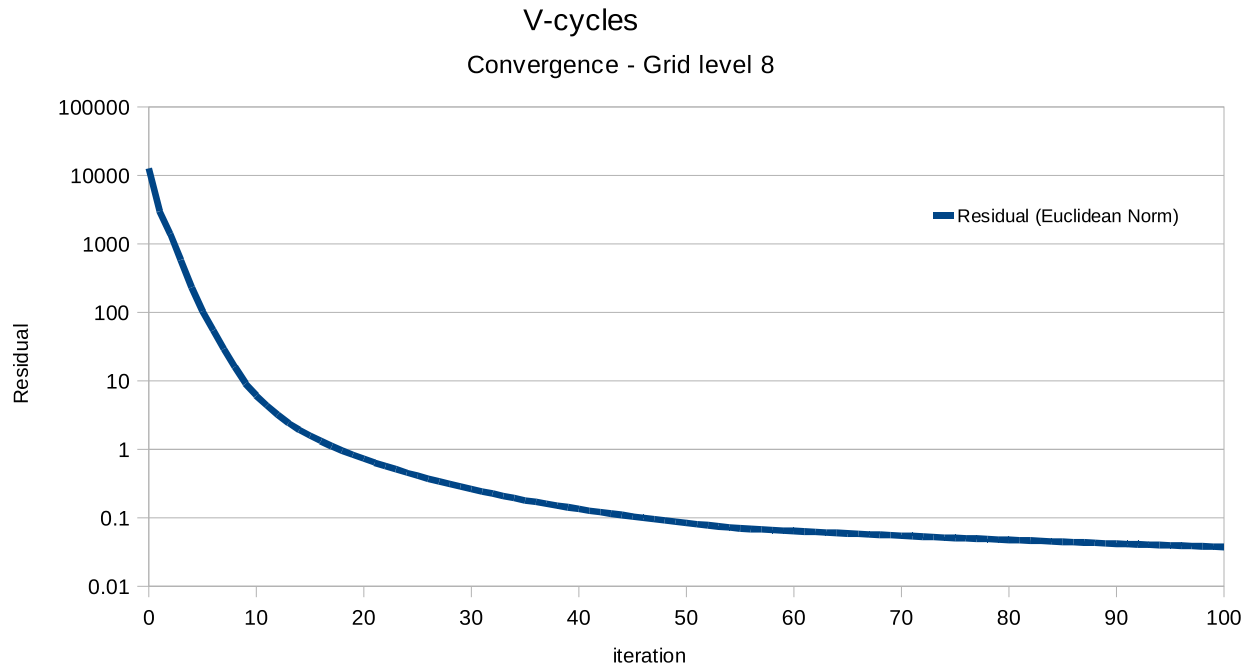
larger problems (starting with level 9) the solution converges only for few initial steps and then diverges to infinity. A this stage the extrapolation of the ghost layer in the prolongation step was not implemented yet and that is the reason for divergence.

Another important observation is the fact, that in [12] the easiest implementation of the grid transition steps is to use direct injection. More advanced methods like linear interpolation for prolongation and full weighting for restriction are said to be more accurate. Results from this project show, that it is not only recommended but actually mandatory to use these methods. If only direct injection is used, then the solution also diverges, just like on the graph above. Only prolongation with both linear interpolation of the domain points and extrapolation of ghost points together with full weighting in restriction produced a numerically stable algorithm, which converged for all tested problem sizes.

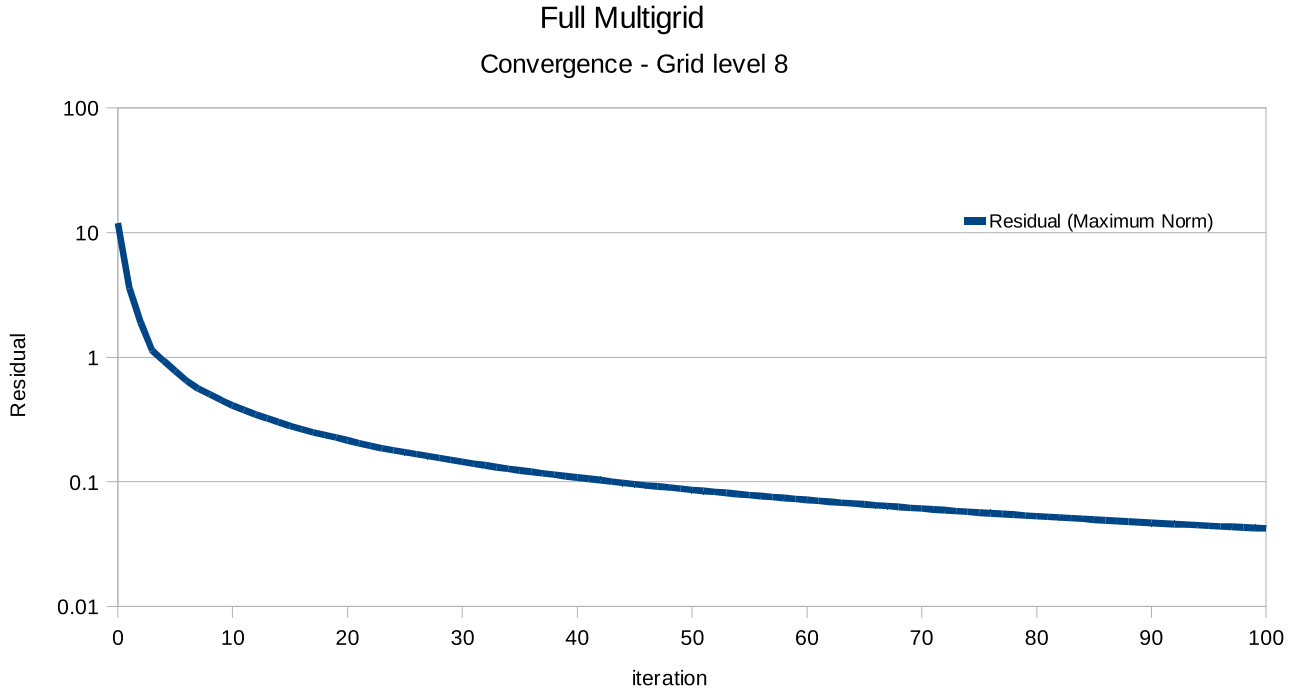
Results from the final version

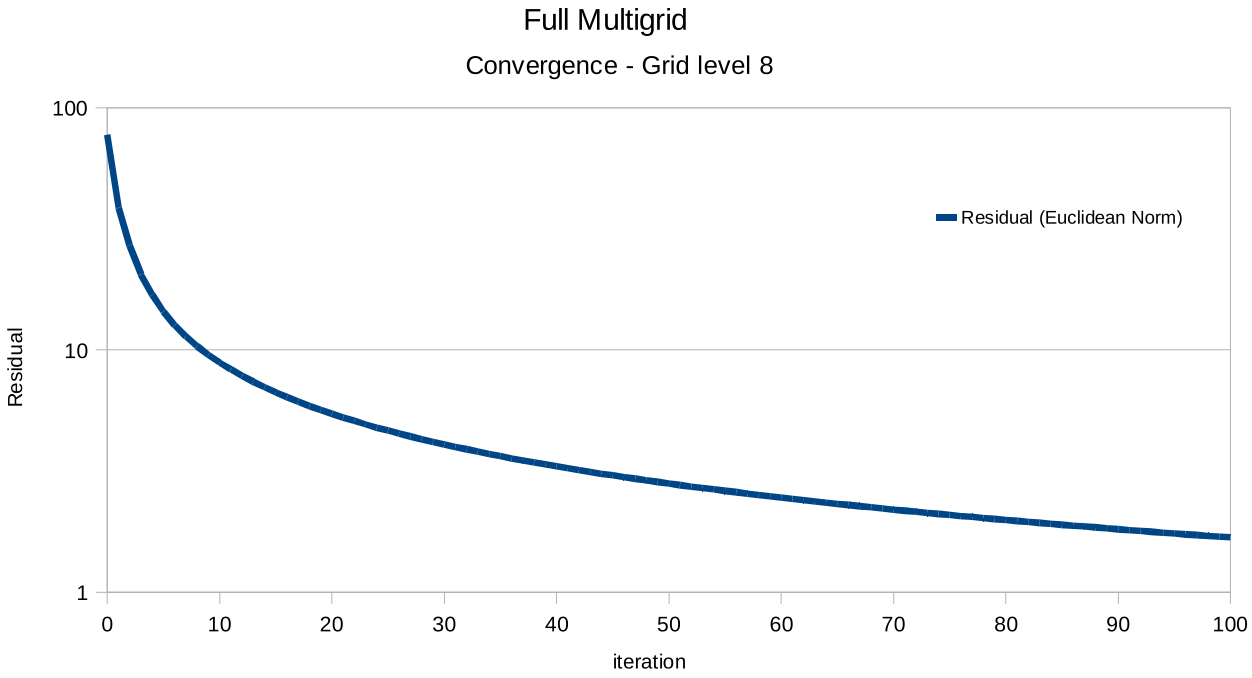
Tests used to verify the Jacobi method are also used to check the Multigrid algorithm. Presented below are graphs showing the behavior of the residual, when the Multigrid method is applied. We can notice that initially the residual values are very large, but quickly become much smaller than the values produced by just Jacobi iterations.



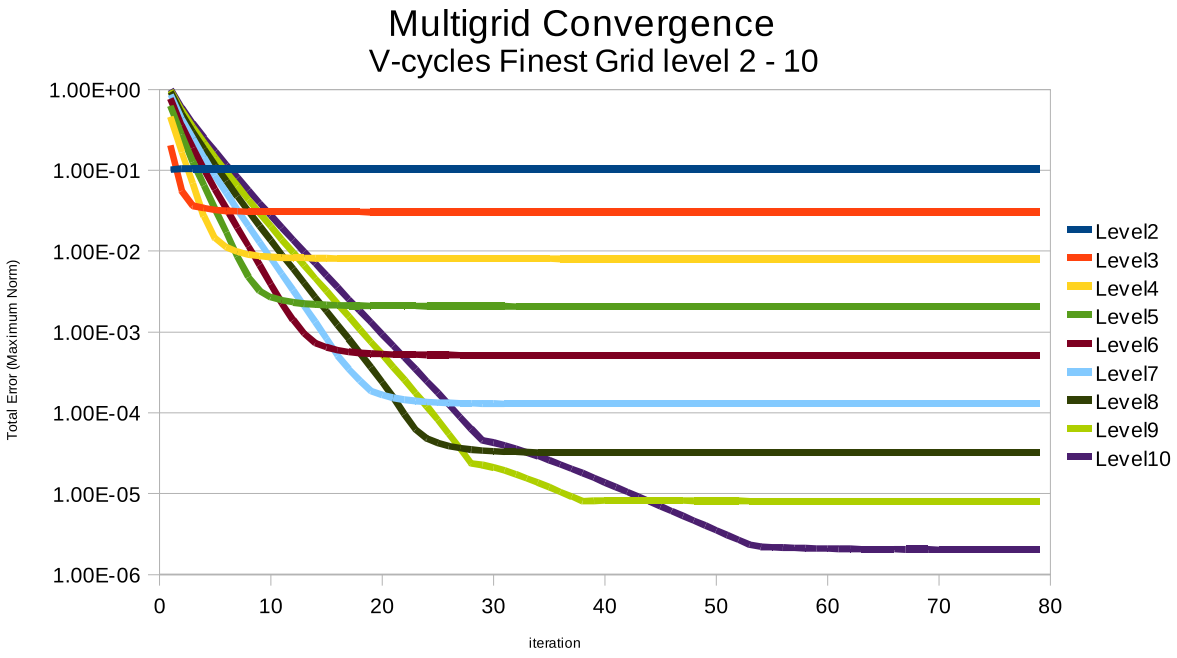


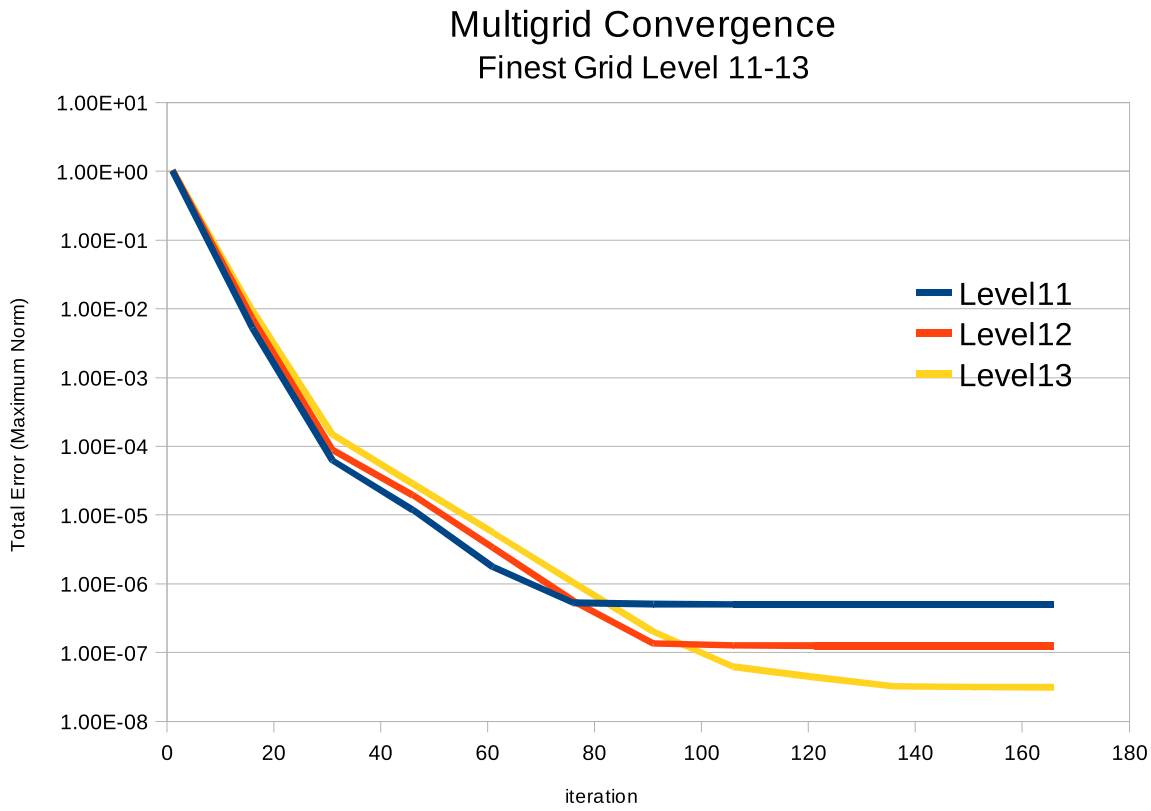
Full Multigrid cycle was also tested. From the results below we can see, that the residual is already very small after just 1 iteration, much smaller compared to performing a v-cycle with an initial guess X=0.





The total error of the Multigrid method has been investigated and the results are presented below for all supported grid levels.





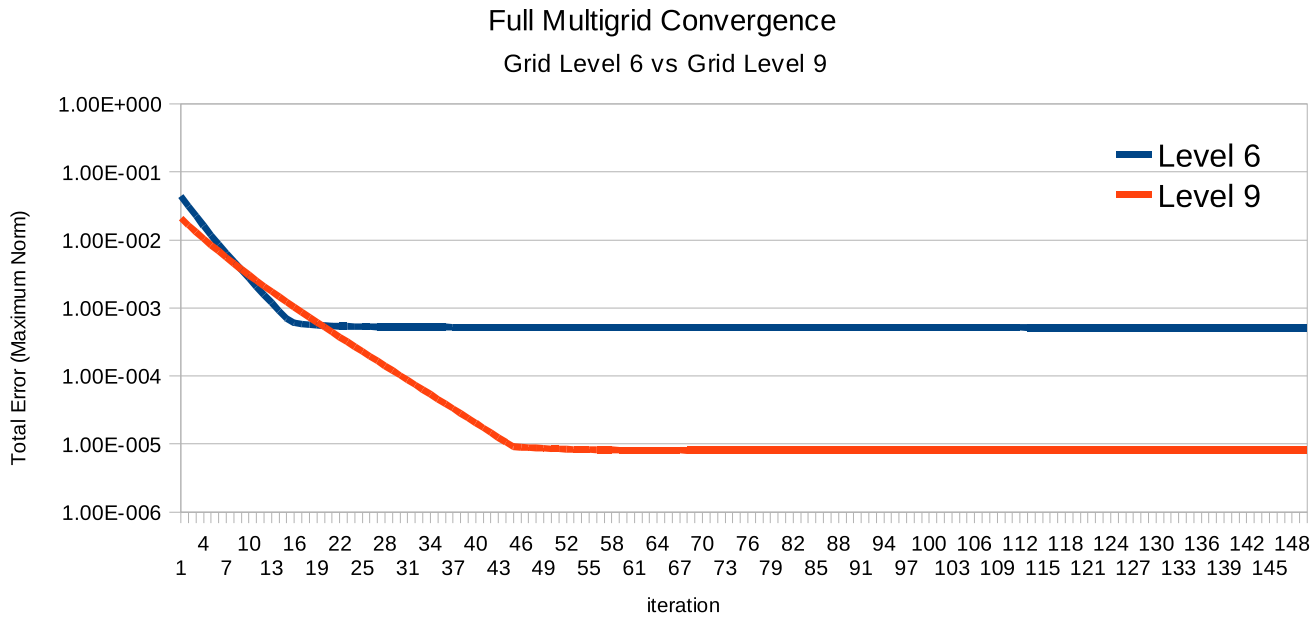
The first thing interesting to notice is the fact that the rate of convergence seems to be independent of the problem size. There is a very small dependence for smaller grid levels – the algorithm converges just a little bit slower for large problem sizes, but this difference becomes negligibly small for grid levels 11-13.

Another interesting fact is, that for every grid level there is a point at which the convergence suddenly stops. This is the point where the Multigrid approximation comes so close to the exact solution, that the approximation error can be considered zero (or at least below precision offered by double precision computations). The only error that remains is the discretization error from the finite difference method, which cannot be removed by the Multigrid.

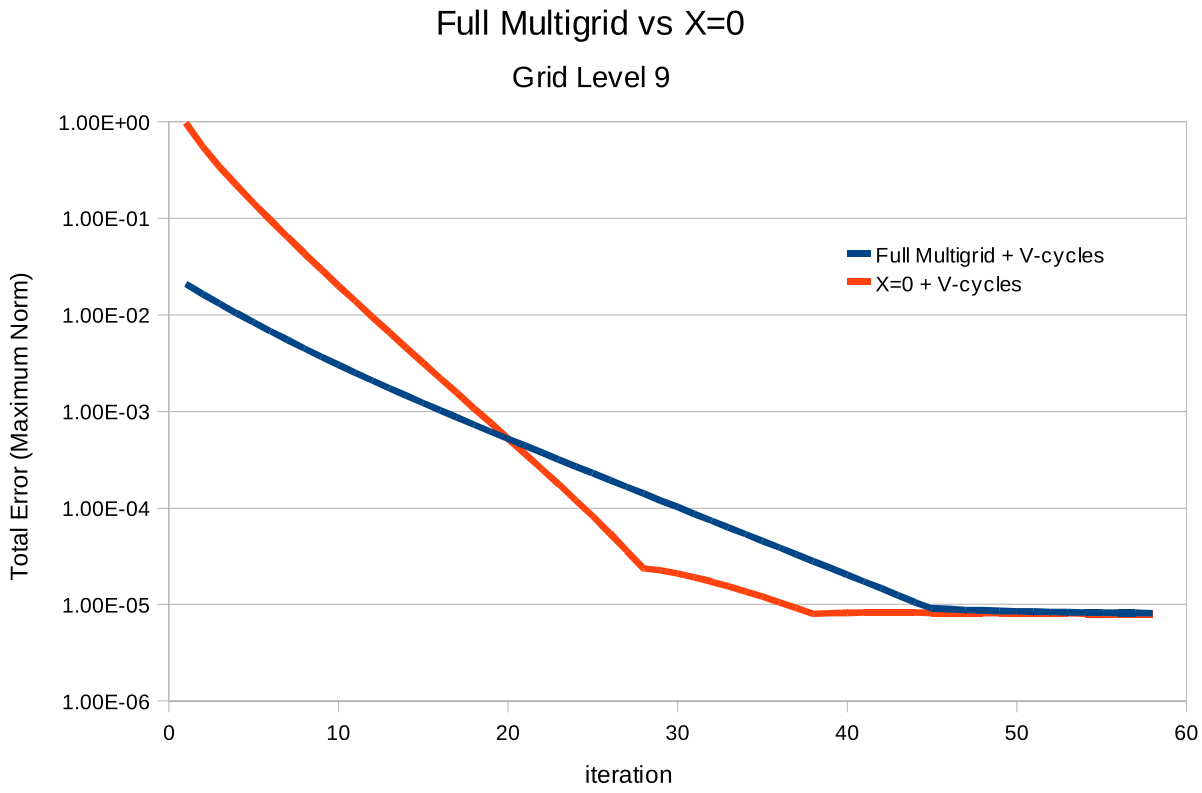
The rate of converge shows exponential decay of the error regardless of problem size and iteration count (straight line on a graph with logarithmic scale). The decay stops being exponential only after we approach the discretization error. An interesting behavior is visible at error values between 10^{-4} and 10^{-5} for grid levels, which can achieve such accuracy. The rate of convergence suddenly drops,

nevertheless still remains exponential.

For comparison a similar graph for the Full Multigrid method (grid levels 6 and 9) is shown below, which has similar properties, except a sudden drop in convergence rate.



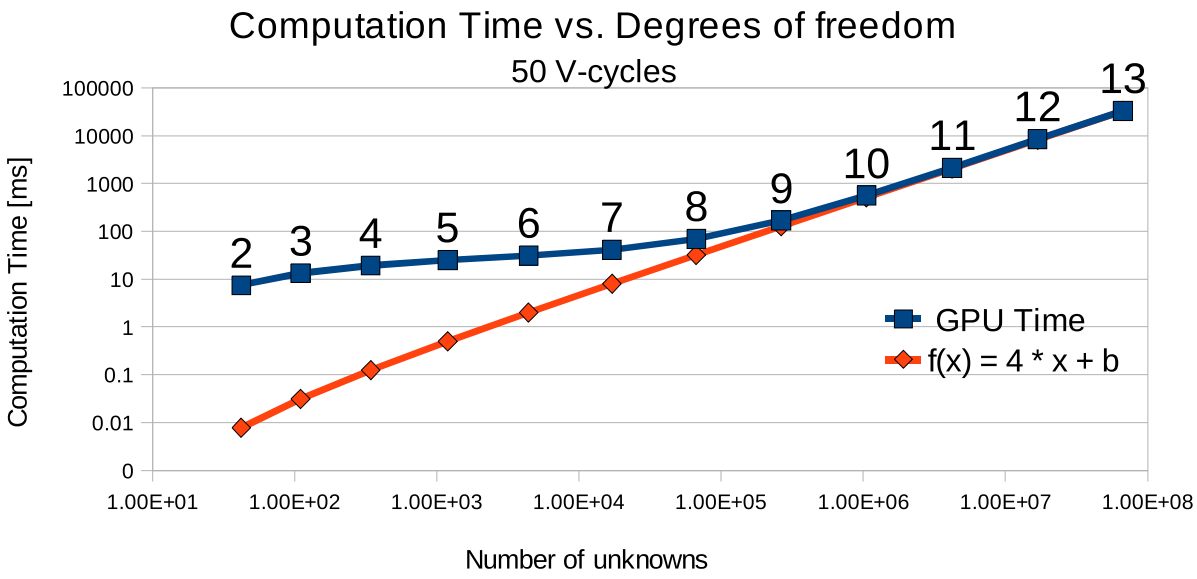
In this project two different schemes are used to solve the problem with Multigrid. We can either assume that our initial guess is X=0 and then perform a number of v-cycles or we can use Full Multigrid to obtain a much better initial guess before we continue with v-cycles to increase the accuracy even further. The graph below shows a comparison of both approaches at grid level 9.



The graph above shows a very interesting result. We could assume that performing Full Multigrid is beneficial, because right at the beginning we get an approximation almost 2 orders of magnitude better. That is why this scheme is often used as a preconditioner for other methods. One can obtain quite a good initial guess with very little computations. However when we notice the rate of convergence and iteration at which the convergence deteriorates, we come to the conclusion that it is not worth to use Full Multigrid at all in our case. Although it gives a much better initial guess, the algorithm converges more slowly when it is used. We can reach best possible approximation earlier, if only v-cycles are used.

Benchmarks

So far we have only considered rate of convergence as a function of iteration count. The graph below shows how much time is needed to perform 50 v-cycles on Tesla C1060 and what is the scalability of the algorithm on the GPU.



We can see that the total time is directly proportional to the number of unknowns for large problem size . That is: increasing the grid level by 1, we increase the number of unknowns by about 4, and computation time also increases by 4. This means, that the Multigrid implementation scales linearly with the problem size, which is the main reason what it is superior to direct methods like Gaussian Elimination for large problem sizes. For smaller problem sizes the amount of time to compute the solution is so small, that other factors become significant, for example API calls or communication time with the hardware. That is the reason why we fail to observe linear scaling for smaller problem sizes on the graph.

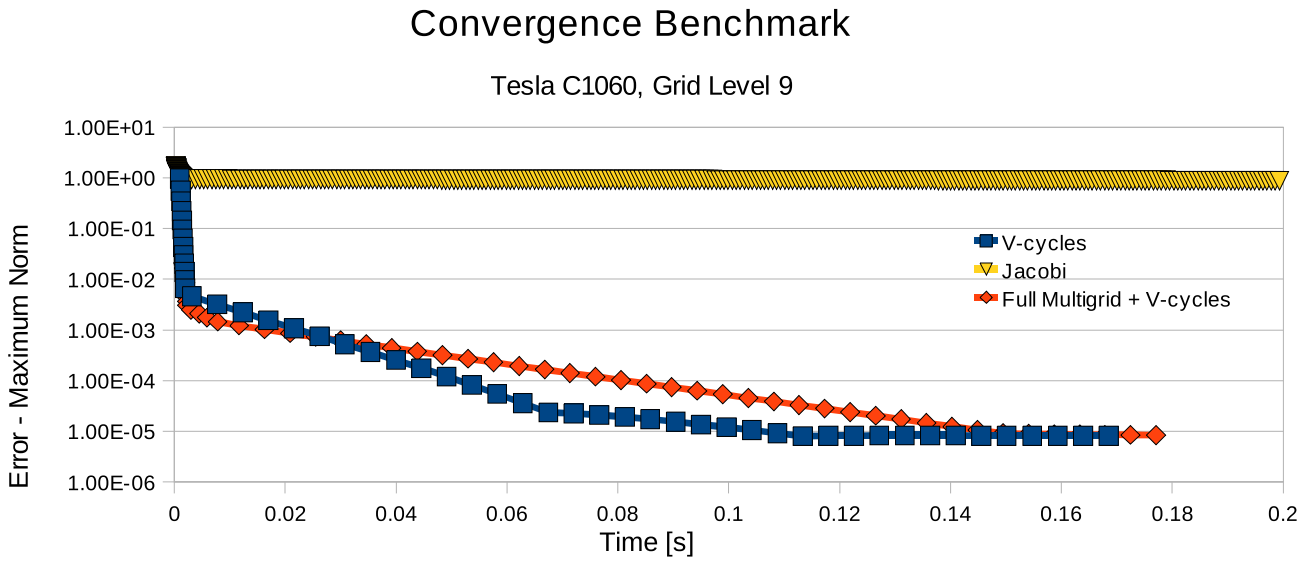
The complete timing results are summarized in the table below.

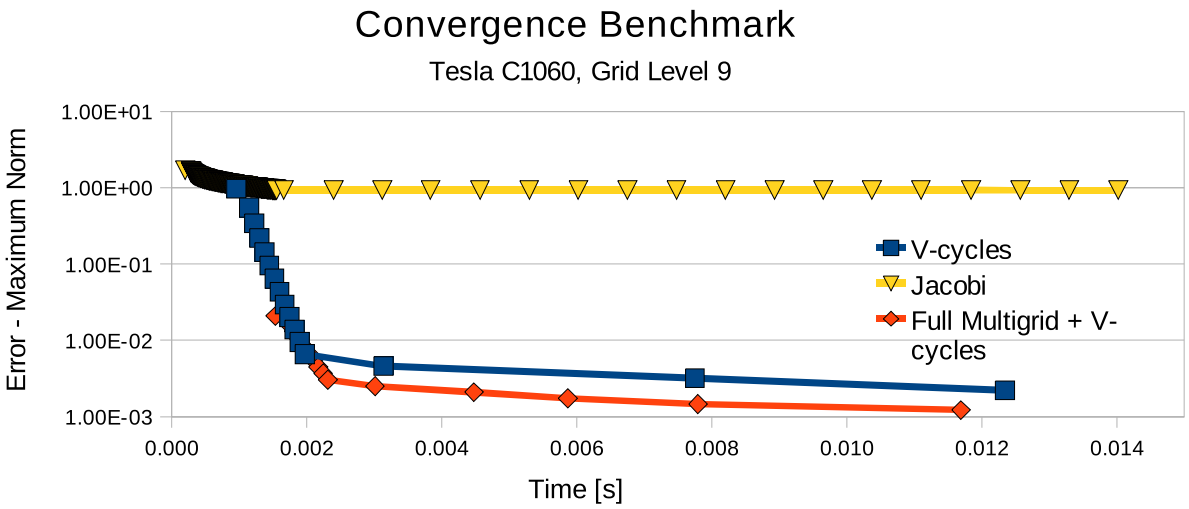
- Send time – includes time needed to initialize CUDA, all data structures and to send initial conditions to the GPU
- GPU time – actual computation time on the GPU
- Receive time – time needed to transfer the solution back to host

50 V-cycles on Tesla C1060

Level	Nb of unknowns	GPU Time [ms]	Send Time [ms]	Receive Time [ms]
2	42	7.43	93.62	21.07
3	110	13.3	50.04	19.97
4	342	19.26	49.45	19.16
5	1 190	25.07	49.82	18.65
6	4 422	31.05	49.81	18.57
7	17 030	40.99	50.09	20.41
8	66 822	69.09	51.45	29.06
9	264 710	168.1	55.19	63.04
10	1 053 702	557.55	68.7	188.32
11	4 204 550	2121.43	155.78	642.98
12	16 797 702	8521.41	326.58	2302.96
13	67 149 830	32882.86	1152.97	7879.42

In general if one method converges faster than the other in terms of iteration count, it doesn't have to be like that when we actually measure computation time and not iterations. An algorithm requiring smaller number of iterations to reach the same accuracy may need much more time to perform each iteration. The graphs below show timing of algorithms used in this project executed on Tesla C1060.





Even though performing Jacobi step is much cheaper in therms of computation time (Tesla C1060 performs about 1300 Jacobi iterations compared to only 50 v-cycles in the same time), the results show that it is absolutely not enough to compete with the Multigrid method. Timing results also confirm the earlier corollary, that it is not worth to use the Full Multigrid method. Although it produces a better result early, it actually converges more slowly later on. Another interesting fact is the property of CUDA API used. It is able to perform few early iterations much faster, than the remaining iterations.

CPU results

Level	CPU Time [ms]
2	224.03
3	347.04
4	792.37
5	2434.5
6	8763.16
7	33615.14
8	131639.54
9	521852.14
10	2079863.07

Table 2: 50 V-cycles (CPU version 1 core)

The table above shows the timing of th CPU version. As mentioned earlier the CPU version was not designed to compete with the GPU version, but rather as a helper in developing and debugging the algorithm. Several checks and additional tests in the code made the development process much easier, but the price for that was reduced efficiency. The results above are therefore shown just for reference

and should not be used for a CPU vs GPU comparison.

Conclusion

An efficient solution to the linearized transformed Laplace problem was implemented using the GPU. The Laplace problem is the computational bottleneck for the potential flow model described in [1]. The algorithm can handle very large problem sizes efficiently, so that the amount of available memory becomes the limiting factor, rather than computation time. The largest problem size consisted of a grid with over 67 mln points. Care was taken when choosing appropriate algorithms to solve the problem, as well as when optimizing the algorithm on the GPU. As a result a solution with linear scalability was created. The algorithm consists only of parts, which are completely parallelizable. It was shown, that using the Multigrid algorithm as a standalone solver and not just as a preconditioner is not only possible, but can reach accuracies limited only by the discretization error of the finite difference method. Using a custom sparse matrix format and a custom sparse matrix-vector multiplication operation greatly reduced both the memory footprint and amount of GPU instructions. Furthermore it was shown that it is not worth to use the Full Multigrid method if Multigrid is used as a standalone method to solve the problem. Finally the results show that using just direct injection in restriction and prolongation steps will not only produce less accurate results, but actually cause the whole algorithm to diverge. Therefore it is not only recommended, but mandatory to use more accurate schemes (linear interpolation and extrapolation of ghost points in the prolongation step and full weighting in restriction step).

Future work

There are several ways the existing solution could be extended. Gauss-Seidel method is known to converge faster than the Jacobi method. It is worth to check if the red-black Gauss-Seidel will produce better results in the current setup. Furthermore the algorithm could be extended into 3D. It is also worth to consider the use of non-uniform grids and to extend the algorithm to work on a completely non-linear wave model. The existing implementation could also be used to solve other problems beyond simulation of water waves, such as the heat transfer.

Bibliography

1: A.P. Engsig-Karup, H.B. Bingham, O. Lindberg, An efficient flexible-order model for 3D nonlinear water waves, 2008
2: Muthu Manikandan Baskaran, Rajesh Bordawekar, Optimizing Sparse Matrix-Vector Multiplication on GPUs, 2003
3: Nolan Goodnight, Gregory Lewin, David Luebke, Kevin Skadron, A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware, 2003
4: Dominik Goddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Hilmar Wobker, Christian Becker, Stefan Turek, Using GPUs to Improve Multigrid Solver Performance on a Cluster, 2008
5: Gundolf Haase , Manfred Liebmann, Craig C. Douglas , Gernot Plank, A Parallel Algebraic Multigrid Solver on Graphics Processing Units,
6: Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan, Solving Sparse Linear Systems on NVIDIA Tesla GPUs, 2009
7: Maria Sosonkina, Donald C. S. Allison, Layne T. Watson, Scalability analysis of parallel GMRES implementations, 2001
8: E. de Sturler, H. A. van der Vorst, Communication cost reduction for Krylov Methods on Parallel Computers, 1994
9: Andrew Kerr, Dan Campbe, Mark Richards, QR Decomposition on GPUs, 2009
10: Ib A. Svendsen, Ivar G. Jonsson, Hydrodynamics of coastal regions, 1976
11: Bengt Fornberg, Calculation of weights in finite difference formulas, 1998
12: William L. Briggs, A Multigrid Tutorial, 1987