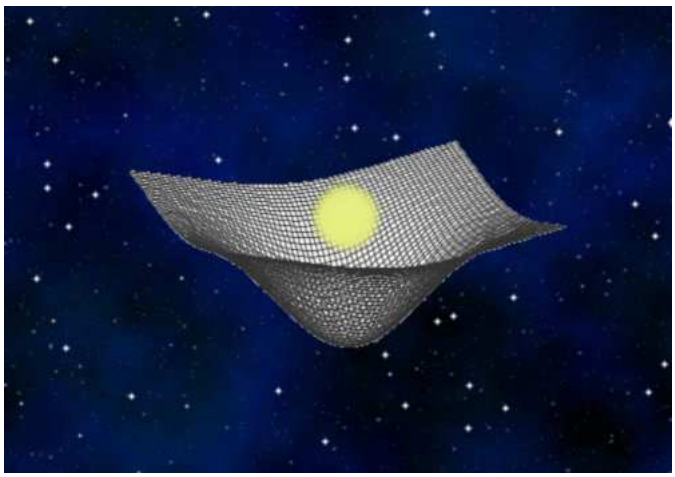
Simulating the gravitational field of a non-rotating neutron star on GPUs



Student: Gkratsia Tantilian
Supervisors: Prof. K. Kokkotas, Prof. N. Stergioulas, Dr. B. Zink

Postgraduate program: Computational Physics
Department: Physics
Institute:
Aristotle University of Thessaloniki
and
Eberhard Karls Universität Tübingen

Thessaloniki
July 2011

Abstract

The main subject of this project is how can we simulate the gravitational field of a neutron star on GPUs. First we need to solve the Einstein's equations for a non-rotating neutron star. We adopt the ADM 3+1 formalism and we use the spatial conformal flatness approximation - CFC, where we have to solve only elliptic equations. To calculate the internal parameters of the star, we use the equations of equilibrium. For faster results, we choose parallel programming on GPUs, in CUDA programming language, while we use a multigrid numerical technique. After some optimizations we achieved important speed ups in CUDA, compared to an initial implementation.

Aknowledgements

I would like to express my appreciation to Prof. Kostas Kokkotas and Prof. Nikolaos Stergioulas for giving me the opportunity to study and understand all the subjects of this project and for supporting me during these studies.

Special thanks to Dr. Burkhard Zink who guided me in a unique way through the knowledge, made everything simpler to understand and inspired me to move on.

How can I forget the kindest person who made me feel welcome during my studies in Tübingen and supported when I was in need, miss Heike Fricke.

Finally, I would like to thank my family, my friends and all the professors who supported me during all my way from my school years until today.

Gkratsia Tantilian,
July 2011

Contents

1 Introduction

In Einstein's theory of general relativity, gravity is not an ordinary force, but rather a property of spacetime geometry. This is currently the most accepted theory of gravity.

Using General Relativity, we expect that high density stellar objects, such as neutron stars, will produce gravitational waves while rotating, due to instabilities. In the case of a binary system, the waves will be even stronger and it will be easier detect them.

The main purpose of this thesis, is to perform a simulation that would show us the behavior of these gravitational waves. But that is the final step. First of all we have to simulate the gravitational field of a non-rotating neutron star. Of course, there are already many simulations for that. What is interesting in our simulation is the numerical techniques and the way it is working. It is important for us to make a simulation as fast as possible, so we can generalize it for the case of rotating neutron stars.

The programming language we choose to work with is CUDA. Since it is possible to use parallel computing to solve our equations, we prefer to do it on GPUs, using CUDA.

Let's see step by step the theoretical approximations, the numerical methods and the way the simulation is implemented.

2 Theory

As mentioned before, the theory that we use is Einstein's General Relativity. General relativity (GR) is a theory of gravitation that was developed by Albert Einstein between 1907 and 1915. According to general relativity, the observed gravitational attraction between masses results from their warping of space and time.That turns the theory of gravity into a theory of spacetime geometry.

By the beginning of the 20th century, Newton's law of universal gravitation had been accepted for more than two hundred years as a valid description of the gravitational force between masses. In Newton's model, gravity is the result of an attractive force between massive objects. Although even Newton was bothered by the unknown nature of that force, the basic framework was extremely successful at describing motion.

Experiments and observations show that Einstein's description of gravitation accounts for several effects that are unexplained by Newton's law, such as minute anomalies in the orbits of Mercury and other planets. General relativity also predicts novel effects of gravity, such as gravitational waves, gravitational lensing and an effect of gravity on time known as gravitational time dilation. Many of these predictions have been confirmed by experiment, while others are the subject of ongoing research.

The Einstein's equations that describe the geometry of spacetime is:

$$R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R = \frac{8\pi G}{c^4} T_{\mu\nu} \tag{1}$$

The solution of the above equations, is the metric tensor $g_{\mu\nu}$. For a flat spacetime, in Cartesian coordinates we get:

$$g_{\mu\nu} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2}$$

So, our purpose is to solve the Einstein's equations for the spacetime

of a neutron star.

2.1 CFC - Approximation

The approximation that we use, is based on the description of the paper: *Dimmelmeier et al. (2002).*

We adopt the ADM (Arnowitt-Deser-Misner) 3+1 formalism to split the 4D spacetime into 3D space + time coordinate. In this case, the line element takes the following form:

$$ds^2 = -N^2 dt^2 + \gamma_{ij}(dx^i + \beta^i dt)(dx^j + \beta^j dt), \tag{3}$$

where N is the lapse function, which describes the rate of advance of time along a timelike unit vector normal to a spacelike hyper-surface, β^i is the spacelike shift three-vector, which describes the motion of coordinates within a hyper-surface and γ_{ij} is the spatial three-metric.

Within the spatial conformal flatness condition (CFC), we approximate the general metric $g_{\mu\nu}$ by replacing the spatial three-metric γ_{ij} with the conformally flat three-metric:

$$\gamma_{ij} = \phi^4 \hat{\gamma}_{ij}, \tag{4}$$

where $\hat{\gamma}_{ij}$ is the flat metric. In Cartesian coordinates:

$$\hat{\gamma}_{ij} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5}$$

That means, we approximate the three-metric with the flat three-metric multiplied by a function ϕ^4 (the conformal factor). With the CFC approximation and the ADM 3+1 formalism, the Einstein's equations take the form of the following five coupled elliptic (Poisson-like)

equations:

$$\hat{\nabla}^2\phi = -2\pi\phi^5\left(\rho h W^2 - P + \frac{K_{ij}K^{ij}}{16\pi}\right),$$

$$\hat{\nabla}^2(N\phi) = -2\pi N\phi^5\left(\rho h(3W^2-2) + 5P + \frac{7K_{ij}K^{ij}}{16\pi}\right), \qquad (6)$$

$$\hat{\nabla}^2\beta^i = 16\pi N\phi^4 S^i + 2\hat{K}^{ij}\hat{\nabla}\left(\frac{N}{\phi^6}\right) - \frac{1}{3}\hat{\nabla}^i\hat{\nabla}_k\beta^k,$$

where ϕ: conformal factor, ρ: rest mass density, $h = 1+\varepsilon P/\rho$: specific relativistic enthalpy (ε: internal energy density), P: pressure, $W = Nu^t$: Lorentz factor, $K_{ij} = \mathcal{L}_{\mathbf{n}}\gamma_{ij}$: extrinsic curvature, $K = K_i{}^i$: the trace of the extrinsic curvature, S^i: the momenta.

2.2 TOV

As we can see above, to solve our equations we need some initial data about the neutron star, such as its density and pressure. We can obtain this information from a TOV solver.

The TOV (Tolman-Ophenheimer-Volkoff) equations for a spherical star, are three coupled differential equations that relate the mass function, the density and the pressure of the star:

$$\frac{dm}{dr} = 4\pi r^2\rho(r),$$

$$\frac{dP}{dr} = -\frac{\rho m}{r^2}\left(1+\frac{P}{\rho}\right)\left(1+\frac{4\pi P r^3}{m}\right)\left(1-\frac{2m}{r}\right)^{-1}, \qquad (7)$$

$$\frac{d\Phi}{dr} = -\frac{1}{\rho}\frac{dP}{dr}\left(1+\frac{P}{\rho}\right)^{-1},$$

where $m(r)$: mass, $P(r)$: pressure, $\rho(r)$: rest mass density, $\Phi(r) =$

$\ln N(r)$. The line element is in the form of:

$$ds^2 = -e^{2\Phi}dt^2 + e^{2\lambda}dr^2 + r^2 d\Omega^2 \tag{8}$$

where $e^{2\lambda} = \left(1 - \frac{2m}{r}\right)^{-1}$.

Using the equation of state: $P = K\rho^{\Gamma}$ (Γ: polytropic index, K: a constant), we solve the TOV equations using a Runge-Kutta method of 4th order. For a neutron star we choose: $\Gamma = 2$ and $K = 100$.

From these equations we obtain the pressure, the density, the mass and the radius of the star (from the center to the point where there is no pressure). Also, from these data we have the boundary conditions of our problem.

2.3 The model

Now that we have the theoretical basis and we know the physical problem we have to face, we need to set up the model that we will simulate.

As we mentioned before, we want to simulate the gravitational field of a non-rotating neutron star. So, having in mind the equations we have to solve and the fact that our star is not rotating, we can make a proper gauge choice, where the spacelike shift three vector is zero: $\beta^i = 0$. Also, since our system is static, there is no motion, we can set the extrinsic curvature $K_{ij} = 0$.

With these conditions, our equations take the following form:

$$\begin{aligned} \hat{\nabla}^2\phi &= -2\pi\phi^5 \left(\rho h W^2 - P\right), \\ \hat{\nabla}^2(N\phi) &= -2\pi N\phi^5 \left(\rho h(3W^2 - 2) + 5P\right), \end{aligned} \tag{9}$$

which are two non-linear elliptic (Poisson-like) differential equations. The pressure and the density functions, are given from the TOV equations and the equation of state.

The unknown quantities are ϕ (conformal factor) and N (lapse function). What we need to do is to solve iteratively the first equation

for ϕ and the second for $\psi = N\phi$. Then, we obtain:

$$N = \frac{\psi}{\phi}. \tag{10}$$

Knowing these functions we obtain $\tilde{g}_{\mu\nu}$:

$$\tilde{g}_{\mu\nu} = \begin{pmatrix} -N^2 & 0 & 0 & 0 \\ 0 & & & \\ 0 & & \gamma_{ij} & \\ 0 & & & \end{pmatrix}, \tag{11}$$

where

$$\gamma_{ij} = \phi^4 \hat{\gamma}_{ij} = \phi^4 \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{12}$$

In conclusion, we only need to find the functions $N(x, y, z)$ and $\phi(x, y, z)$. In the next chapters we will discuss about the numerical methods and computational techniques we will use, to achieve that.

Notice that the CFC condition is exact for spherical spacetimes. Nevertheless, we mention it here in preparation for future extensions to non-spherical spacetimes. Also, from the solution of the TOV equations, we can also obtain $N(r)$ and $\phi(r)$ - obviously, we are not using this result, but only $P(r)$ in order to initiate our elliptic solver. A complete simulation would also evolve the hydrodynamics equations, in order to achieve a time-evolution of the full system.

3 Numerical methods

Continuous problems must sometimes be replaced by a discrete problem whose solution is known to approximate that of the continuous problem; this process is called discretization. For example, the solution of a differential equation is a function. This function must be represented by a finite amount of data, for instance by its value at a finite number of points in its domain, even though this domain is a continuum.

There are two main categories of methods of solving a problem numerically: Direct methods and Iterative methods. Direct methods compute the solution to a problem in a finite number of computations. These methods would give the precise answer if they were performed in infinite precision arithmetic. Examples include Gaussian elimination, the QR factorization method for solving systems of linear equations, and the simplex method of linear programming.

In contrast to direct methods, iterative methods are not expected to terminate in a number of steps. Starting from an initial guess, iterative methods form successive approximations that converge to the exact solution only in the limit of an infinite number of iterations. A threshold is specified in order to decide when a sufficiently accurate solution has been found. Even using infinite precision arithmetic these methods would not reach the solution within a finite number of steps, in general. Examples include Newton's method, the bisection method and the Gauss-Seidel method. In computational matrix algebra, iterative methods are generally needed for large problems.

In our simulation we use an iterative method, called Gauss-Seidel method. Next we will describe this method and see how we can implement it numerically.

3.1 The Gauss - Seidel method

In this section, we will describe how we can solve an elliptic differential equation (Poisson-like) using the Gauss-Seidel method.

Suppose we have the Poisson equation: $\nabla^2\phi = f$, which can also

be written as:

$$\frac{\partial^2\phi}{\partial x} + \frac{\partial^2\phi}{\partial y} + \frac{\partial^2\phi}{\partial z} = f. \tag{13}$$

Let u be the numerical solution of the equation. Then, we can replace the partial differential operators with the following schemes:

$$\begin{aligned}
\left.\frac{\partial^2\phi}{\partial x}\right|_{i,j,k} &= \frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{h_x^2}, \\
\left.\frac{\partial^2\phi}{\partial y}\right|_{i,j,k} &= \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{h_y^2}, \\
\left.\frac{\partial^2\phi}{\partial z}\right|_{i,j,k} &= \frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{h_z^2},
\end{aligned} \tag{14}$$

where h_i is the grid size of the i-direction. By representing our solution in a 3D grid, (i,j,k) are the coordinates of the solution in our grid.

If we replace the operators with the respective numerical form and set $h_x = h_y = h_z = h$, our equation takes the following form:

$$\begin{aligned}
\tfrac{u_{i-1,j,k} - 2\cdot u_{i,j,k} + u_{i+1,j,k}}{h^2} + \tfrac{u_{i,j-1,k} - 2\cdot u_{i,j,k} + u_{i,j+1,k}}{h^2} + \\
\tfrac{u_{i,j,k-1} - 2\cdot u_{i,j,k} + u_{i,j,k+1}}{h^2} = f.
\end{aligned} \tag{15}$$

Finally, we solve this equation with respect to $u_{i,j,k}$ and get the relation:

$$\begin{aligned}
u_{i,j,k} = A_i(u_{i+1,j,k} + u_{i-1,j,k}) + A_j(u_{i,j+1,k} + u_{i,j-1,k}) \\
+ A_k(u_{i,j,k+1} + u_{i,j,k-1}) - A_f f
\end{aligned} \tag{16}$$

where

$$
\begin{aligned}
A_i &= \frac{h_y^2 h_z^2}{2D}, \\
A_j &= \frac{h_x^2 h_z^2}{2D}, \\
A_k &= \frac{h_x^2 h_y^2}{2D}, \\
A_f &= \frac{h_x^2 h_y^2 h_z^2}{2D}, \\
D &= h_x^2 h_y^2 + h_x^2 h_z^2 + h_y^2 h_z^2.
\end{aligned}
\tag{17}
$$

Using the above equation we can calculate the value of the solution at the point (i,j,k) if we know the the surrounding values. In the figure 1 we represent the scheme in two dimensional problem.

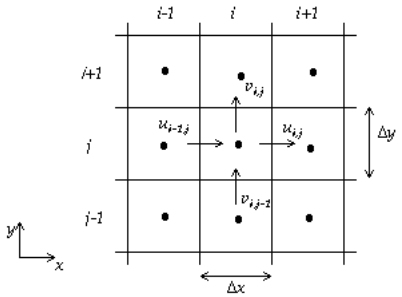


Figure 1:
The Gauss-Seidel method for 2-D grid

Gauss-Seidel is an iterative method. These methods are also called relaxation methods since we have an initial guess of the solution and through the iterations we let it to relax to the real solution, reducing the error. To solve the Poisson equation with the Gauss-Seidel method, we have to set the boundary conditions, make an initial guess of the solution and relax (start the iterations) until we reach the desired accuracy.

3.2 Red - Black

As one can notice from the representation of the method, to find the solution in a point of the grid, we need the values of the upper, lower, left and right points. But what if at the same time we are renewing the value of one of these points? In few cases that could cause some problems. Especially if we have parallel computations, it is possible to try to read and write at the same time, the value of a point. For that, the relaxation can be split into two phases.

Imagine the grid as a chess board. Having the Gauss-Seidel method in mind, we can see that the white squares are independent. Their value depends only on the surrounding black squares.

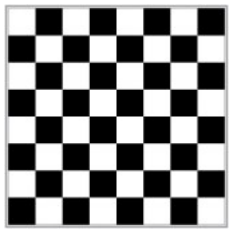


Figure 2:
The chess board as an analog of Red-Black relaxation

To avoid conflicts we relax first for the points at the *black* squares, and then for the points at the *white* squares. This technique is called *Red-Black Relaxation* and it is well represented in the figure 3.

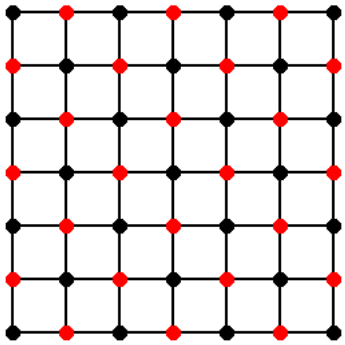


Figure 3:
A representation of the Red-Black relaxation

3.3 Multigrid

Although the method we described in the previous chapters, is effective and converges to the desired solution, it is quite slow, especially when we have to make calculations for a high resolution grid. Imagine a 100x100x100 grid. That means we have to calculate the solution for 1 000 000 points for each full-grid iteration. Before implementing it and use it as our solver, we can make some improvements of the algorithm.

A very good technique that will speed up our iterative method, is what we will present on this chapter and it is called *multigrid*. In *multigrid* we reduce the calculations by changing the grid resolution. Let's see the whole idea, step by step.

Let $Au = f$ be our numerical equation in matrix form. Where A is the left hand side operator of the differential equation, u is our numerical solution and f is the *source* function.

Now we can define the residual

$$r = f - Au, \tag{18}$$

obviously if u is the exact solution, then the residual is zero ($r = 0$). Otherwise, the residual is related to the accuracy of the solution. To be more specific, the norm $\| r \|_\infty$ is what defines our accuracy. Numerically, if we have a matrix with the values of the residual for each grid point, then $\| r \|_\infty = \max(r_i)$.

Now, lets assume that v is the exact solution of the problem. We can relate the exact solution with the numerical solution with the relation:

$$v = u + e, \tag{19}$$

where e is the error of the solution. This means that we have the equation $Av = f$. Now we can get:

$$\left.\begin{array}{l} Av = f \\ v = u + e \end{array}\right\} \Rightarrow A(u + e) = f \Rightarrow \\ Au + Ae = f \Rightarrow Ae = f - Au. \tag{20}$$

But, we know that $f - Au = r$ so we obtain the equation:

$$Ae = r, \tag{21}$$

which means that we can use the residual of our equation and consider it as the source function of the equation for the error, and solve for the error. After finding the error we can go back and correct our numerical solution. But what is the reason to do that, since the number of iterations will be the same?

If we examine carefully our iterative method, we can realize that it is a method where each value of the solution is affected by the neighborhood. So, the solver does not see further, to converge to the exact solution faster. This means that if there are high frequency errors, they will vanish fast enough, but for low frequency errors, it needs too many iterations and it converges very slowly.

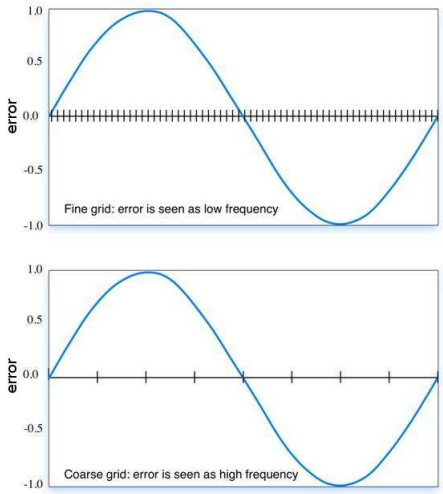


Figure 4:
Low frequency errors appear as high frequency errors in a low resolution grid

A good way to face this problem is to change the grid resolution. That's because a low frequency error in a high resolution grid, appears as high frequency error in a lower resolution grid, as we can see in the figure 4.

So, what we do in the *multigrid* method is to set the initial problem, with the necessary boundary conditions and source function and perform a few relaxations steps. Afterwards we take the residual and turn it into the source function of the error, in a grid of half resolution.

The boundary conditions for the error are set to zero, since that is the desired solution. After a couple of iterations we do the same for the error, so we solve for the error of the error. After reaching the highest level with the lowest resolution, we go back and correct the errors and in the end we correct the solution.

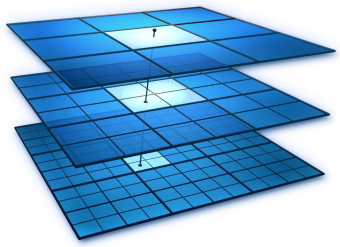


Figure 5:
The levels of the multigrid method

The operator that turns a grid (for example the residual) into a half-resolution grid, is called Restriction and it is an interpolation operation. The restriction transports the residual of the fine grid to the coarser grid. The typical process (for a 2D grid) averages the neighboring values according to a stencil, that can be represented as:

$$I_h^H = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}. \tag{22}$$

This is the so called *full weighting restriction.*

On the other hand, the operator that turns a grid (for example the error) into a double-resolution grid, is called Prolongation and it is also an interpolating operation. The prolongation injects the error of the coarser grid to the finer grid, so we can correct the solution. The typical process (for a 2D grid) averages the neighboring values according to a stencil, as

$$I_H^h = \frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{23}$$

The *multigrid* method starts from the finest level where we solve for the solution, and goes to a coarser level where we solve for the error. Afterwards it goes back to the finest level by correcting the errors and in the end the solution. This full cycle is called V-cycle.

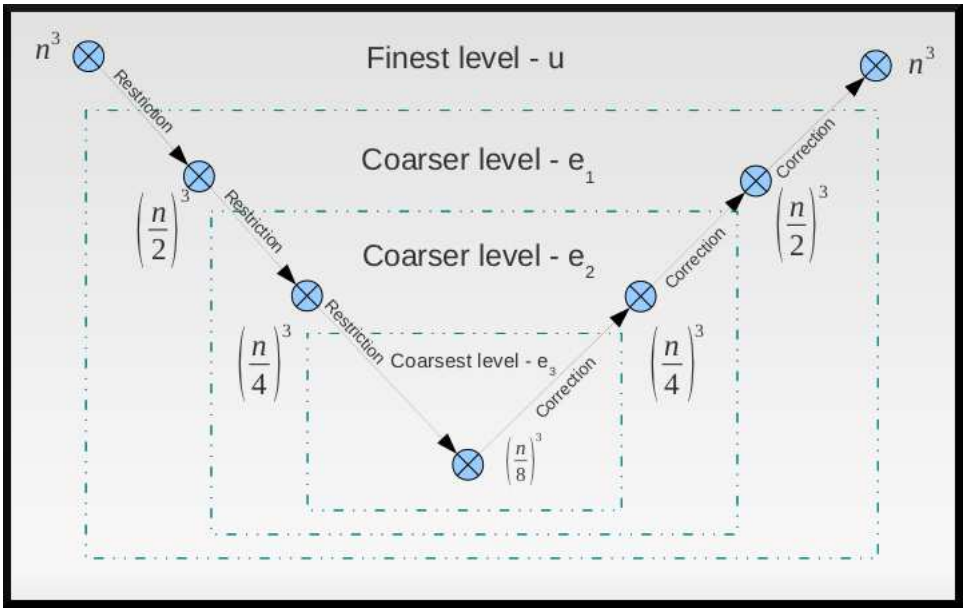


Figure 6:
A representation of the full V-cycle

To see the speed up of this method, we solve the Poisson equation with and without multigrid, using the Gauss-Seidel method. In the graphs below we can see the residual for each case as a function of the number of iterations.

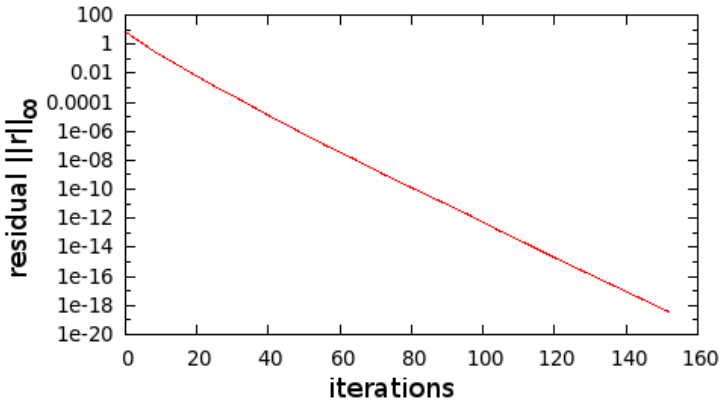


Figure 7:
Gauss-Seidel without multigrid

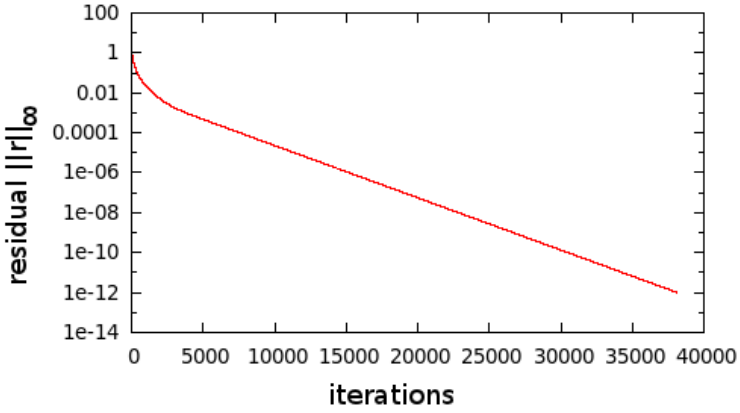


Figure 8: *Gauss-Seidel with multigrid*

As we can see, without the multigrid we need around 35000 iterations for a relative accuracy of 10^{-12}, while for the same accuracy with multigrid, we need around 90 iterations.

4 GPU computing

Now that we have seen the physical problem we have to solve, the numerical method and the algorithms that we will use, we need to choose a fast and effective programming language. But, before choosing the language, it would be a good idea to consider the hardware we will use.

As our models become more complicated we need faster computational performance. On the other hand, the more the technology evolves and we get faster computers, the more complicated problems we can deal with.

According to Moore's law the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every one and a half to two years. So, we expect to see an improvement in CPU performance. But, what is happening is that CPU computational performance is growing at a smaller rate.

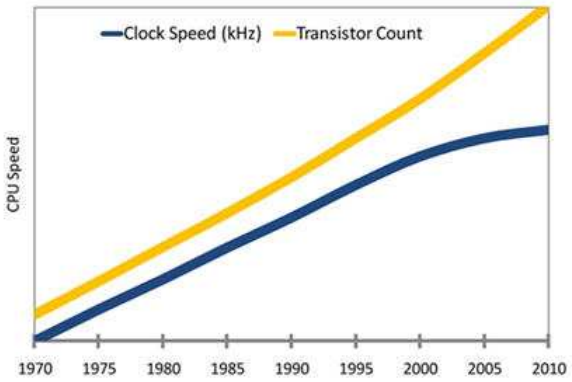


Figure 9:
Moore's law

There are many reasons why we see that the CPU clock speed grows slower. One of them is the design limits. To make CPUs faster we need more transistors, so we need **smaller transistors** so we can keep the CPU design in a small size. But, we are approaching a limit of our capabilities of designing small circuits.

Another way to speed up a CPU clock is to supply it with more power. Nowadays CPUs are supplied with over 200 A current. The

problem that could appear is **overheating** and even the melting of the circuits, if there is no sufficient cooling.

These problems can be faced and are not as important compared to the limitations of the atomic scale. There are physical restrictions that will not allow us to make smaller circuits, because below a specific size **quantum effects** appear, such as quantum tunnelling.

Another problem that appears using a single CPU is that even if the CPU was fast enough, the bus speed of the CPU slot and the memory slot adds a limitation to the performance.

The solution to these problems is to turn to parallel computing. By making parallel computations on many CPUs, we reduce the needs of the performance for each CPU, but we need many computational units. Either many computers in a cluster or many CPU cores in a computer. Nowadays the CPU industry realized the problem and turned in producing multi-core CPUs.

With parallel computing we increase the computation performance. But still, we need to set up computer clusters to get enough performance for our needs. Do we have any other choice? Is it possible to set up a single computer to get the performance we need? The answer is that we can do that, if we turn from CPUs to GPUs.

4.1 GPUs

The computer game industry is one of the biggest and most demanding industries in the world. The games become more realistic with higher image quality. Which means more vertexes have to be rendered and at the same time there must be simulations of the light diffusion in the rendering stage. All these calculations to give a good performance and desired frame-rate in the games, need high performance parallel computations.

These demands of the computer industry lead graphic card companies to design the needed hardware. NVIDIA GPUs are highly parallel, multi-threaded, many-core processors with very high processing performance and memory bandwidth.

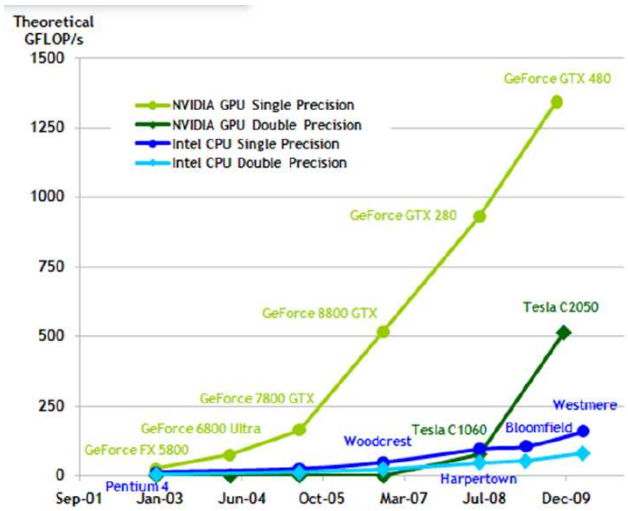


Figure 10:
GPU and CPU peak performance

In the figure 10, we see the CPU and GPU peak performance in Gigaflops (flops: floating point operations per second), comparing single and double precision devices. In the figure 11 we see the CPU and GPU memory bandwidth in Gigabytes per second.

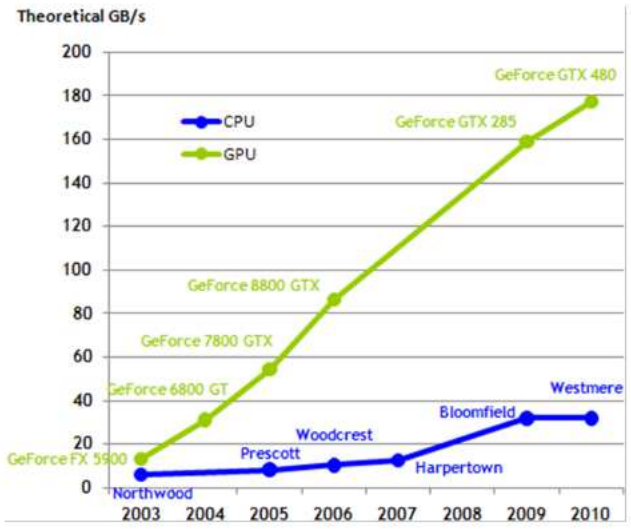


Figure 11:
GPU and CPU memory speed

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. This can be schematically represented in the figure 12.

Figure 12:
CPU and GPU architecture

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accel-

erated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

One of the biggest advantages of GPU Computing, is that we can have this computational power in a small device, even in a personal computer.

Now that we met GPUs and saw why it is preferable to set our simulation calculations on GPUs instead of CPUs, lets see how we can manage that.

4.2 CUDA

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture - with a new parallel programming model and instruction set architecture - that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

At its core are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are simply exposed to the programmer as a minimal set of language extensions.

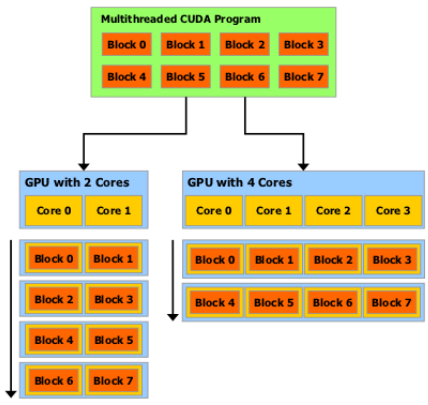


Figure 13:
Thread scalability for different GPUs

A multi-threaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores. That makes the program auto-scalable.

As mentioned before, programming in CUDA-C is not difficult, for a C-programmer, since it is in the same environment with some extensions and extra keywords.

The most important thing in CUDA programming is to understand the thread hierarchy and the way we order them on GPUs. The threads in CUDA are grouped in blocks of threads. The blocks are organized on a grid. For optimization reasons we should define the dimensions of the grid and of the blocks. In the figure 14 we can see a good illustration of that.

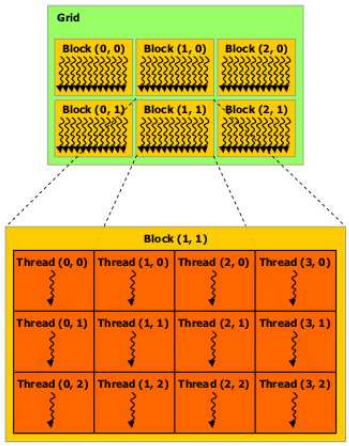


Figure 14:
The threads are organized into blocks which are aligned in a grid

CUDA C extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. A kernel is defined using the **__global__** declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<< ... >>> execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the

built-in **threadIdx** variable.

Lets see an example, to see how it works. In the following example we will make a program in CUDA for vector addition. Lets assume that we have the N-dimensional vectors A and B and we want to obtain the vector C=A+B. In this case we have to add all the elements of each vector one-by-one. So first we create a *kernel* that does that:

```
__global__ void VAdd(float *A, float *B, float *C){

  int i = threadIDx.x;
  C[i] = A[i] + B[i];

}
```

Then, we should call this *kernel* from the main function, with the necessary parameters to define the grid and block dimensions. Since in the above function we defined i as the x-coordinate of threads, if we want to get valid results we have to call it in a one-dimensional block with x-dimension equal to the vector size. And since we have only one block, our grid is also 1 by 1. So, the main function will be as below:

```
void main(){

  // Vector dimension
  int N=100;
  // Define vectors A, B and C
  float *A,*B,*C;
  // Initializing vectors A and B
  for(int i=0;i<N;i++){
  A[i]=1;
  B[i]=2;
  }
  // Calling the kernel
  VAdd<<< 1, N >>>(A,B,C);

}
```

Now, if we run this program and export the values of vector C, we will see that all it's components are equal to 3.

This was a very simple application on CUDA programming just to see how it works. The parameters that define the grid and block dimensions are variables of the type **dim3**. Each component of these variables is not defined, it is set to 1. For example:

```
dim3 blockD;
blockD.x=100;
```

If we leave it like that, it assumes that:

```
blockD.y=1;
blockD.z=1;
```

The block dimensions can be from one to three (1D to 3D block), when the grid dimensions can be from one to two (1D or 2D grid). As we can see in the representation of the next picture, each block has its

coordinates - indexes on the grid, and each thread has its coordinates in the block. That makes a unique index for each thread which is defined by the block and thread indexes. For a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy).

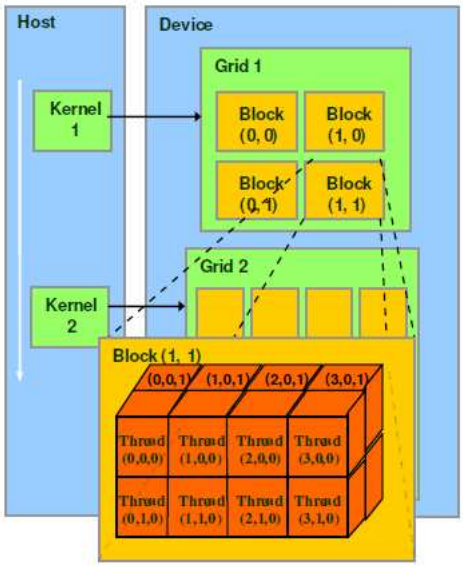


Figure 15:
Thread indexes in the 3D blocks in a 2D grid

Now, lets discuss some built-in variable that are useful for our simulation. These variables are predefined, built-in variables and we cannot define variables with these names:

gridDim: This variable is of type dim3 and contains the dimensions of the grid.

blockDim: This variable is of type dim3 and contains the dimensions of the block.

blockIdx: This variable is of type uint3 and contains the block index within the grid.

threadIdx: This variable is of type uint3 and contains the thread index within the block.

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points **in the kernel** by calling the **__syncthreads()** intrinsic function; __syncthreads() acts as a barrier at which all threads in the block must wait before any is allowed to proceed. __syncthreads() is a function that synchronizes the threads in a specific kernel when they share data in a shared memory.

Sometimes it is possible to start executing a kernel while another one is not finished yet, since the threads are executed asynchronously. This can cause some data conflicts and lead to false results. In this case we use the built-in function **__cudaThreadSynchronize()** after a kernel and before calling another kernel.

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application. The memory structure can be illustrated as in the following figure.

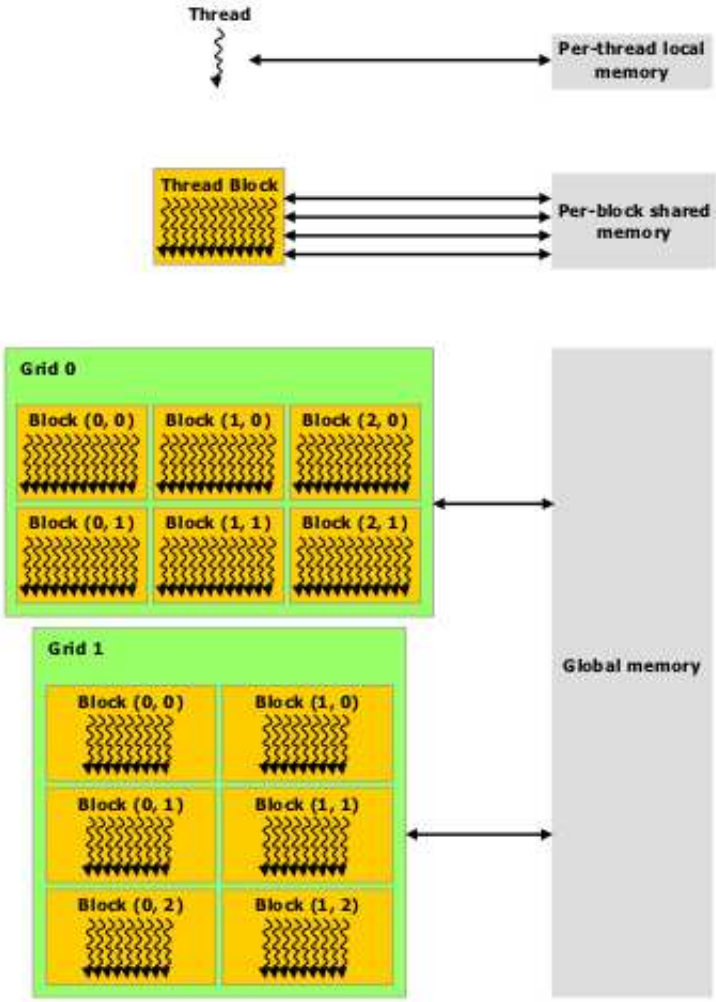


Figure 16:
The memory structure on GPUs

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.

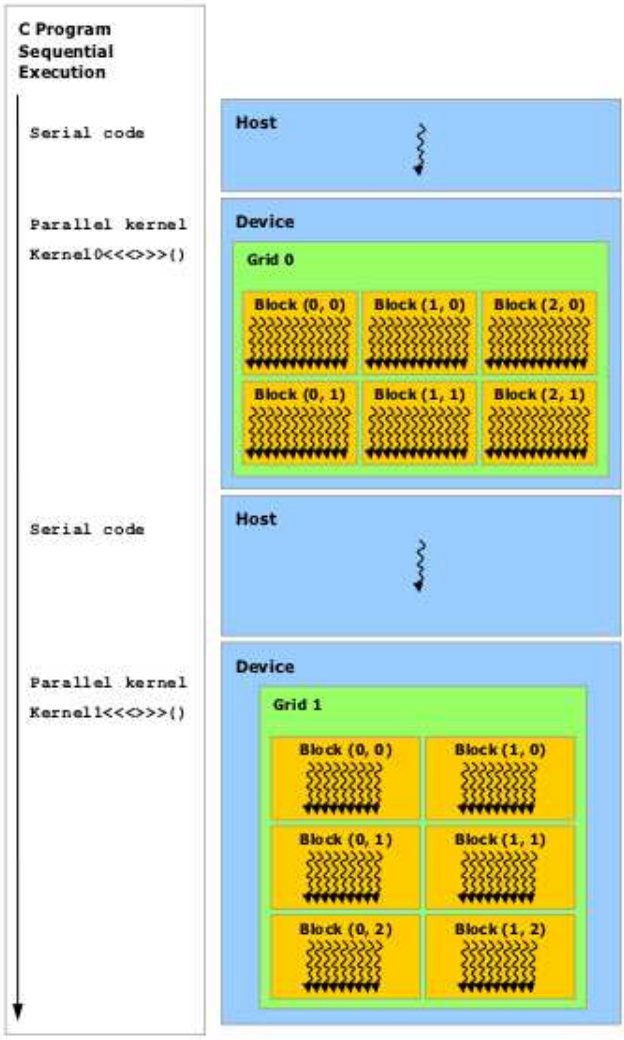


Figure 17:
Heterogeneous programming on the Host and Device

If we want to exchange data from host to device or the other way around, we need to allocate memory and copy the respective pointers. Memory allocation on the device can be done using the command:

```
cudaMalloc((void)&d_A, Nsizeof(type));
```

just as the usual **malloc** in C. Copying from device to host or host to device can be done done with the command:

```
                // Copying from Host to Device
cudaMemcpy(d_A,A,Nsizeof(type),cudaMemcpyHostToDevice);
                // Copying from Device to Host
cudaMemcpy(d_A,A,Nsizeof(type),cudaMemcpyDeviceToHost);
```

In conclusion we can say that in CUDA programming we have to set up the problem on the host, set the needed parameters and call the kernels to make our calculations on the device. The data flow can be described from the picture below.

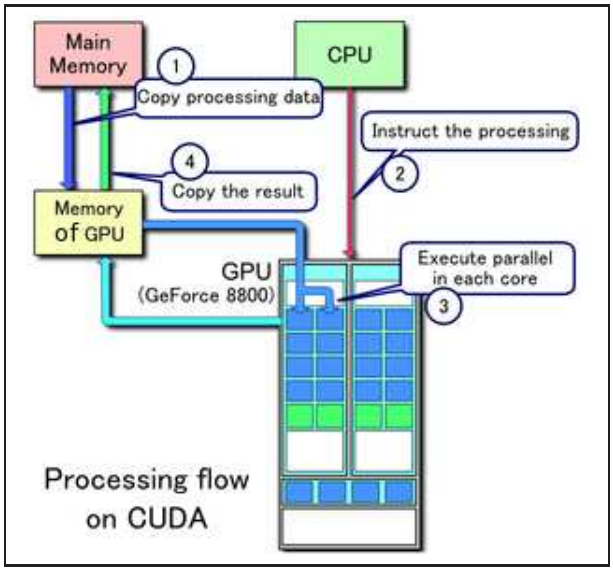


Figure 18:
Processing flow on CUDA

According to this illustration, first we initialize our data to the host memory (RAM), then we transfer it to the device memory (memory of GPU) and we execute parallel in each core. After obtaining the final results , we copy back the solution to the host memory, for output.

Now that we understood the physical problem, we saw the numerical methods and techniques that we will use to solve the problem, and we choose the fast and effective way of implementation on GPUs, it is time to combine all this knowledge for setting up our simulation.

In the following chapters we will see how we implemented our simulation, the description of the code and some optimization of the code for better performance. Finally, we will present and discuss the final results.

5 The simulation

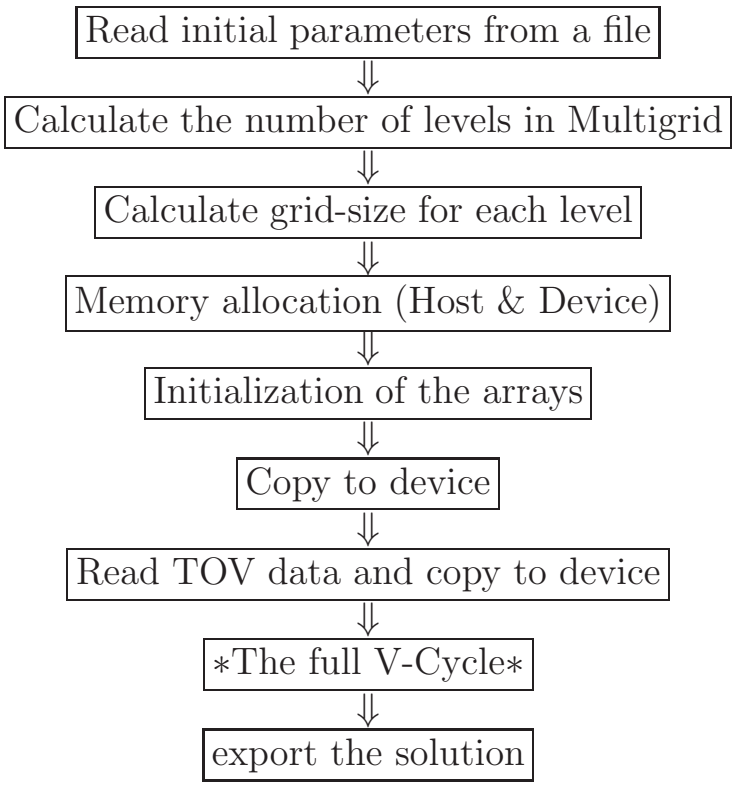
Now that we have all the tools we need, we can set up our numerical solver and get to the results. As mentioned before, our program will be written in CUDA-C. Also we saw that CUDA-C is an extension of C/C++. So the way we implement our solver is to define some functions first and then use them as we need.

Although we implemented a TOV solver, what we will do, is to read the initial data from another solver (RNS - Rapidly Rotating Neutron Star - by Nikolaos Stergioulas, Stergioulas & Friedman - 1995).

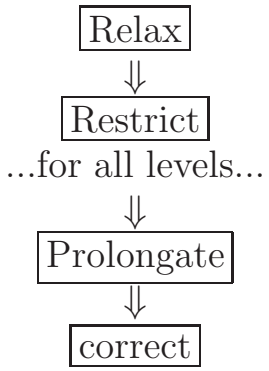
The main functions that we will use are:

- A function to read the data from the output files of the RNS solver
- A function to calculate the grid and lock dimensions according to the problem size
- A function to calculate the multigrid levels according to the grids dimensions
- A function to allocate memory for all the functions
- A function to copy data from Host to Devise and another for doing the opposite
- A function to calculate the residual r from the solution u
- A function to free all the allocated memory
- A function to initialize the functions
- A function for Restriction and another for Prolongation

After setting these functions, we create the *main* function where the algorithm can be described from the following scheme:



What is happening in the full V-cycle is:



In our first version of the program (without optimization) the initialization of the source (for facing the non-linearity) was being done on the host side. So each time we had to copy data to the host and then back to the device.

In the following section we will see the code description, to see how was the simulation implemented.

5.1 Code description

In this section we describe some essential parts of the code, so it becomes clear how we implemented the model. First of all, lets see the main function:

```
// Reading the parameters from file (prameters.txt) and
displaying them in the screen
...
pFile = fopen("parameters.txt","r");
skip(2,pFile,str); // skips 2 lines wile reading a file
fscanf(pFile, "%s\t%d",str,&Nx);// gets the value of Nx
...(scan all the other parameters)...
```

In the first part of main function we read the file `parameters.txt` and get the values for the grid dimensions etc. Next we calculate the number of steps we have to do in multigrid, using the function `lcalc`:

```
// Calculate the number of steps in multigrid
real el,elx,ely,elz;
elx = lcalc(Nx);
ely = lcalc(Ny);
elz = lcalc(Nz);
```

where we have:

```
real lcalc(int N)
{
real l;
l = log10(N-1)/log10(2);
return l;
}
```

After that we use `malloc` to allocate memory for the host side pointers:

```
// calculating the dimensions for each level
int *nx, *ny, *nz;
nx = (int*)malloc(el*sizeof(int));
ny = (int*)malloc(el*sizeof(int));
nz = (int*)malloc(el*sizeof(int));
nx[0] = Nx;
ny[0] = Ny;
nz[0] = Nz;
for (int i=1;i<el;i++)
{
nx[i] = ((nx[i-1]-1)/2)+1;
ny[i] = ((ny[i-1]-1)/2)+1;
nz[i] = ((nz[i-1]-1)/2)+1;
}
...
```

When we finish with memory allocation on the host and on the device, using `cudaMalloc`, we initialize the source on the host and copy it to the device, using the function `cudaMemCpy`:

```
cudaMalloc( (void**) &d_u[i],
nx[i]*ny[i]*nz[i]*sizeof(real));
...
cudaMemcpy(d_u[i],u[i],
nx[i]*ny[i]*nz[i]*sizeof(real),cudaMemcpyHostToDevice);
...
```

After allocating and initializing all the functions we will use and before starting the solution procedure, we read the data from the TOV solver:

```
...
fscanf(fP, "%lf\t%lf\t%lf\t%lf",&x,&y,&z,
      &P[i+Nx*(j+Ny*k)]);
fscanf(frho, "%lf\t%lf\t%lf\t%lf",&x,&y,&z,
      &rho[i+Nx*(j+Ny*k)]);
fscanf(fN, "%lf\t%lf\t%lf\t%lf",&x,&y,&z,
      &N[i+Nx*(j+Ny*k)]);
...
```

After copying these data to the device, we are ready to start solving the equations. In the next part of the solver, there are three main loops. The outer loop is for facing the non-linearity. So inside the loop we have an initial guess for ϕ and we use it to define the right hand side of the equations. Then, we solve the equations and we have a better approximation for ϕ. So, we use the new solution to define the source of the elliptic equation, and then repeat this until we obtain the desired accuracy.

The next loop is the loop for V-cycles. In our case we choose one V-cycle per loop. In each V-cycle we have a relaxation loop for each level of *Multigrid* and the Restriction and Prolongation operators. Also, for each level, since the grid dimensions are not the same, we use the `dim_calc` function to calculate each time the grid and block dimensions for each *kernel*.

```
// outer loop
for(int o_loop = 0; o_loop<outer_loop; o_loop++){
dim_calc(Dg,Db,nx[0],ny[0],nz[0],devProp);
// Initializing with new solution
src_initd<<< Dg, Db >>>(K, gamma, Nx, Ny, Nz,d_rho,d_P,
               d_u[0], d_s[0], d_ns[0],d_N);
```

Now the V-cycle starts. During the ascending branch we *restrict*, *relax* and calculate the function r_i for all the points of the grid.

```
// loop over V-cycles
for(int cycle=0;cycle<vcycles;cycle++){
// ascending
dim_calc(Dg,Db,nx[di],ny[di],nz[di],devProp);
// Restriction
for(int bz=0;bz<nz[di];bz++) {
IH<<< Dg, Db >>>(d_s[di], d_r[di-1],
  nx[di], ny[di], nz[di], nx[di-1],ny[di-1],nz[di-1],bz);
cudaThreadSynchronize(); // *** wait for kernel to complete ***
IH<<<Dg,Db>>>(d_ns[di], d_nr[di-1],
  nx[di], ny[di], nz[di], nx[di-1],ny[di-1],nz[di-1],bz);
cudaThreadSynchronize(); // *** wait for kernel to complete ***
}
```

```
// relax for u
relax(...);
// calculate the residual
r_calc(...);
// relax for Nu
relax(...);
// calculate the residual
r_calc(...);
```

```
// descending
dim_calc(Dg,Db,nx[di],ny[di],nz[di],devProp);
// Prolongation
for(int bz=0;bz<nz[di];bz++) {
Ih<<< Dg, Db >>>(d_s[di], d_r[di-1],
  nx[di], ny[di], nz[di], nx[di-1],ny[di-1],nz[di-1],bz);
cudaThreadSynchronize(); // *** wait for kernel to complete ***
Ih<<<Dg,Db>>>(d_ns[di], d_nr[di-1],
  nx[di], ny[di], nz[di], nx[di-1],ny[di-1],nz[di-1],bz);
cudaThreadSynchronize(); // *** wait for kernel to complete ***
}
```

After completing the full V-cycle and the outer loop, we copy back the results to the host and export them to external files, using the function `export_data`. And finally, before ending the program, we free all the allocated memory using:

```
free(x);
```

for the main memory, and

```
cudaFree(x);
```

for the device memory.

Now, lets see the functions we used above. First, lets see the function that calculates the grid and block dimensions:

```
// Calculating block and grid dimensions (Kernel parameters)
void dim_calc(dim3 &Dg, dim3 &Db, int Nx, int Ny, int Nz){
int tx, ty, tz; // block dimensions
tx = 8;
ty = 8;
tz = 4;
Db.x = tx;
Db.y = ty;
Db.z = tz;
if((Nx-2)%tx==0)
{
Dg.x = (Nx-2)/tx;
}
else
{
Dg.x = (Nx-2)/tx+1;
}
if((Ny-2)%ty==0)
{
Dg.y = (Ny-2)/ty;
}
else
{
Dg.y = (Ny-2)/ty+1;
}
}
```

In the optimized version, tx, ty and tz, are defined by the device parameters. Nextstarts, we will discuss the initialization function.

```
__global__ void src_initd(real K, real gamma, int Nx, int Ny,
    int Nz, real *rho, real *P, real *u, real *s,
     real *ns, real *N){
...  defining constants epsilon and W ...
// leave the first column
int i = threadIdx.x+blockDim.x*blockIdx.x+1;
// Leave the first line
int j = threadIdx.y+blockDim.y*blockIdx.y+1;
int k = threadIdx.z+blockDim.z*bz+1;
// i = even or odd depending on j+init (red - black)
int i = 2*ii+(j+init)%2;
if(i>(Nx-2)||j>(Ny-2)||k>(Nz-2)){return;}
s[i+Nx*(j+Ny*k)] = - 2.0*pi*pow(u[i+Nx*(j+Ny*k)],5)*
(rho[i+Nx*(j+Ny*k)]*(1.0+epsilon)*pow(W,2)+P[i+Nx*(j+Ny*k)]*
pow(W,2)-P[i+Nx*(j+Ny*k)]);
ns[i+Nx*(j+Ny*k)] = 2.0*pi*N[i+Nx*(j+Ny*k)]*
pow(u[i+Nx*(j+Ny*k)],5)*((rho[i+Nx*(j+Ny*k)]*(1+epsilon)+
P[i+Nx*(j+Ny*k)])*(3.0*pow(W,2)-2)-5.0*P[i+Nx*(j+Ny*k)]);
}
}
```

Now, lets discuss how we implement the restriction and prolongation functions on GPUs.

```
// I_h^H (restriction)
__global__ void IH(real *un, real *uo, int nxn,
    int nyn,int nzn, int nx, int ny, int nz, int bz)
{
int i = threadIdx.x+blockDim.x*blockIdx.x;
int j = threadIdx.y+blockDim.y*blockIdx.y;
int k = threadIdx.z+blockDim.z*bz;
// check if i,j or k are out of desired space
if(i>(nxn-1)||j>(nyn-1)||k>(nzn-1))
{
return;
}
if(i==0||i==(nxn-1)||j==0||j==(nyn-1)||k==0||k==(nzn-1))
{
un[i+nxn*(j+nyn*k)] = uo[2*i+nx*(2*j+ny*2*k)];
}
else
{
un[i+nxn*(j+nyn*k)] = ((uo[(2*i-1)+nx*((2*j-1)+ny*(2*k-1))]+
uo[(2*i+1)+nx*((2*j-1)+ny*(2*k-1))] + uo[(2*i+1)+nx*((2*j+1)+ny*(2*k-1))] +
uo[(2*i-1)+nx*((2*j+1)+ny*(2*k-1))] + uo[(2*i-1)+nx*((2*j-1)+ny*(2*k+1))] +
uo[(2*i+1)+nx*((2*j-1)+ny*(2*k+1))] + uo[(2*i+1)+nx*((2*j+1)+ny*(2*k+1))] +
uo[(2*i-1)+nx*((2*j+1)+ny*(2*k+1))]) + 2.0*(uo[(2*i)+nx*((2*j-1)+ny*(2*k-1))] +
uo[(2*i)+nx*((2*j+1)+ny*(2*k-1))] + uo[(2*i-1)+nx*((2*j)+ny*(2*k-1))] +
uo[(2*i+1)+nx*((2*j)+ny*(2*k-1))] + uo[(2*i-1)+nx*((2*j-1)+ny*(2*k))] +
uo[(2*i-1)+nx*((2*j+1)+ny*(2*k))] + uo[(2*i+1)+nx*((2*j+1)+ny*(2*k))] +
uo[(2*i+1)+nx*((2*j-1)+ny*(2*k))] + uo[(2*i)+nx*((2*j-1)+ny*(2*k+1))] +
uo[(2*i)+nx*((2*j+1)+ny*(2*k+1))] +uo[(2*i-1)+nx*((2*j)+ny*(2*k+1))] +
uo[(2*i+1)+nx*((2*j)+ny*(2*k+1))]) + 4.0*(uo[(2*i)+nx*((2*j)+ny*(2*k-1))] +
uo[(2*i-1)+nx*((2*j)+ny*(2*k))] + uo[(2*i+1)+nx*((2*j)+ny*(2*k))] +
uo[(2*i)+nx*((2*j-1)+ny*(2*k))] + uo[(2*i)+nx*((2*j+1)+ny*(2*k))] +
uo[(2*i)+nx*((2*j)+ny*(2*k+1))]) + 8.0*uo[(2*i)+nx*((2*j)+ny*(2*k))])/64.0; }
}
```

```
// I_H^h (prolongation)
__global__ void Ih(real *un, real *uo, int nxn,
    int nyn, int nzn, int nx, int ny, int nz, int bz)
{
int i = threadIdx.x+blockDim.x*blockIdx.x+1;
int j = threadIdx.y+blockDim.y*blockIdx.y+1;
int k = threadIdx.z+blockDim.z*bz+1;
// check if i,j or k are out of range
if(i>(nxn-2)||j>(nyn-2)||k>(nzn-2))
{
return;
}
int ic = i/2;
int jc = j/2;
int kc = k/2;
if(i%2==0&&j%2==0&&k%2==0)
{
un[i+nxn*(j+nyn*k)] = uo[ic+nx*(jc+ny*kc)] + un[i+nxn*(j+nyn*k)];
}
if(i%2==0&&j%2==0&&k%2!=0)
{
un[i+nxn*(j+nyn*k)] = (uo[ic+nx*(jc+ny*kc)]+uo[ic+nx*(jc+ny*(kc+1))])/2.0
    + un[i+nxn*(j+nyn*k)];
}
if(i%2==0&&j%2!=0&&k%2==0)
{
un[i+nxn*(j+nyn*k)] = (uo[ic+nx*(jc+ny*kc)]+uo[ic+nx*(jc+1+ny*kc)])/2.0
    + un[i+nxn*(j+nyn*k)];
}
if(i%2!=0&&j%2==0&&k%2==0)
{
un[i+nxn*(j+nyn*k)] = (uo[ic+nx*(jc+ny*kc)]+uo[ic+1+nx*(jc+ny*kc)])/2.0
   + un[i+nxn*(j+nyn*k)];
}
if(i%2!=0&&j%2==0&&k%2!=0)
{
un[i+nxn*(j+nyn*k)] = (uo[ic+nx*(jc+ny*kc)]+uo[ic+1+nx*(jc+ny*kc)]+
   uo[ic+nx*(jc+ny*(kc+1))]+uo[ic+1+nx*(jc+ny*(kc+1))])/4.0 +
    un[i+nxn*(j+nyn*k)];
}
```

```
if(i%2!=0&&j%2!=0&&k%2==0)
{
un[i+nxn*(j+nyn*k)] = (uo[ic+nx*(jc+ny*kc)]+uo[ic+1+nx*(jc+ny*kc)]+
   uo[ic+nx*(jc+1+ny*kc)]+uo[ic+1+nx*(jc+1+ny*kc)])/4.0 +
    un[i+nxn*(j+nyn*k)];
}
if(i%2!=0&&j%2!=0&&k%2!=0)
{
un[i+nxn*(j+nyn*k)] = (uo[ic+nx*(jc+ny*kc)]+uo[ic+1+nx*(jc+ny*kc)]+
   uo[ic+nx*(jc+1+ny*kc)]+uo[ic+nx*(jc+ny*(kc+1))]+
   uo[ic+1+nx*(jc+1+ny*kc)]+uo[ic+1+nx*(jc+ny*(kc+1))]+
   uo[ic+nx*(jc+1+ny*(kc+1))]+uo[ic+1+nx*(jc+1+ny*(kc+1))])/8.0
    + un[i+nxn*(j+nyn*k)];
}
if(i%2==0&&j%2!=0&&k%2!=0)
{
un[i+nxn*(j+nyn*k)] = (uo[ic+nx*(jc+ny*kc)]+uo[ic+nx*(jc+1+ny*kc)]+
   uo[ic+nx*(jc+ny*(kc+1))]+uo[ic+nx*(jc+1+ny*(kc+1))])/4.0 +
    un[i+nxn*(j+nyn*k)];
}
}
```

The last function that we will discuss, is the most important. This is the Gauss-Seidel solver, including the *Red-Black relaxation*.

```
// Relaxation function
void relax(real nx, real ny, real nz, real xmin, real xmax, real ymin,
    real ymax, real zmin, real zmax, real *u, real *s,
     real *r,dim3 Dg, dim3 Db, int in)
{
for(int inter=0;inter<in;inter++)
{
// dx, dy, dz
real dx = (xmax - xmin)/((real)nx-1);
real dy = (ymax - ymin)/((real)ny-1);
real dz = (zmax - zmin)/((real)nz-1);
// dx^2, dy^2, dz^2
real dx2 = dx*dx;
real dy2 = dy*dy;
real dz2 = dz*dz;
// D....
real D = dx2*dy2 + dx2*dz2 + dy2*dz2;
real Di = (dy2*dz2)/((real)2.0*D);
real Dj = (dx2*dz2)/((real)2.0*D);
real Dk = (dx2*dy2)/((real)2.0*D);
real Ds = (dx2*dy2*dz2)/((real)2.0*D);
for (int bz=0; bz<nz-1; bz++)
{
EDESolve<<< Dg, Db >>>(u,nx,ny,nz,Di,Dj,Dk,Ds,s,bz,0);// Red
cudaThreadSynchronize();// *** wait for kernel to complete ***
EDESolve<<< Dg, Db >>>(u,nx,ny,nz,Di,Dj,Dk,Ds,s,bz,1); // black
cudaThreadSynchronize();// *** wait for kernel to complete ***
}
}
}
```

This function is a host side function which calls the device solver with some parameters. For example, the last input of the solver, defines if it is the red or the black relaxation. Next is the devise function (*kernel*) which solves the equation using Gauss-Seidel method.

```
// Kernel definition - Gauss-Seidel solver function
__global__ void EDESolve(real *u, int Nx, int Ny, int Nz,
    real Di, real Dj, real Dk, real Ds, real *s, int bz, int init)
{
// i,j,k....
int i = threadIdx.x+blockDim.x*blockIdx.x+1; // leave the first column
int j = threadIdx.y+blockDim.y*blockIdx.y+1; // Leave the first line
int k = threadIdx.z+blockDim.z*bz+1;
if(i>(Nx-2)||j>(Ny-2)||k>(Nz-2)){return;}
if((i+j+k)%2==init) // implementing Red - Black
{
u[i+Nx*(j+Ny*k)] = (Di*(u[(i+1)+Nx*(j+Ny*k)]+
   u[(i-1)+Nx*(j+Ny*k)]) + Dj*(u[i+Nx*((j+1)+Ny*k)]+
   u[i+Nx*((j-1)+Ny*k)]) + Dk*(u[i+Nx*(j+Ny*(k+1))]+
   u[i+Nx*(j+Ny*(k-1))])) - Ds*s[i+Nx*(j+Ny*k)];
}
}
```

All these *kernels* have a parameter `bz` which defines the z-coordination in the domain grid.

6 Optimization

After finishing our first version of the solver, we tried to improve its performance. To optimize the solver we need to discuss some technical issues about the data flow in CUDA and which configurations are the most optimized.

6.1 Basic optimization

There are some basic rules about the optimization in CUDA.

First of all, we need to optimize our algorithms. This means we need to maximize the independent parallelism and minimize latencies among calculations.

Also, as we mentioned in the respective section, GPUs are made for computations and not for caching. This means, that sometimes it is better to recalculate something, than to cache it.

The most common procedure that slows down a CUDA program, is the data exchange between host and device. We have to be careful because, no matter how fast our GPUs are, if we exchange data with the host, then the port-bus speed is what counts. In some cases there is no choice and we have to do it, but generally we try to avoid it.

Another optimization issue is the memory control. On GPUs there are thee kind of memories. The one is per thread, the local memory, the other is per block the shared memory and there is the global memory which is the main memory of GPUs. One has to be careful about how to manage the threads so they will be more optimized. Generally, shared memory is a lot faster so we try to have in each block data that are related, so we don't need to access the global memory.

Maximizing multiprocessor usage is another optimization technique. To do this we need to know what is the maximum allowed block size on the device. This is possible with the command:

```
cudaGetDeviceProperties(&devProp,0);
```

Where `devProp` is a variable of type: `cudaDeviceProp`. This variable is a structure with a lot of information about our device. What we need are the maximum threads per each dimension of a block, which we can obtain in this way:

```
int mtbx = devProp.maxThreadsDim[0];
int mtby = devProp.maxThreadsDim[1];
int mtbz = devProp.maxThreadsDim[2];
```

6.2 Changes in our code

With some little changes in our code we achieved great speed up. First of all, we replaced the initialization function on the part where we face the non-linearity, with another one which is on GPU. So now we don't need to copy data between the host and the device, at that point. Of course, there is still a place, where we need to do that, and

it is at the part where we calculate the residual $\|r\|_\infty$. To deal with this part we need to make a sorting function in CUDA and this is not easy to be implemented.

Next, we got our device properties using:

```
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp,0);
```

and used

```
int mtbx = devProp.maxThreadsDim[0];
int mtby = devProp.maxThreadsDim[1];
int mtbz = devProp.maxThreadsDim[2];
```

to define our block dimensions. We used these values in the `dim_calc` function where we calculate the block and grid dimensions:

```
void dim_calc(dim3 &Dg, dim3 &Db,
int Nx, int Ny, int Nz, cudaDeviceProp devProp)
{
// get number of multiprocessors
int mpc=devProp.multiProcessorCount;
// get maximum threads per block dimensions
int mtbx = devProp.maxThreadsDim[0];
int mtby = devProp.maxThreadsDim[1];
int mtbz = devProp.maxThreadsDim[2];
// get maximum grid dimensions
int mgx = devProp.maxGridSize[0];
int mgy = devProp.maxGridSize[1];
int tx, ty, tz; // block dimensions
tx = mtbx;
ty = mtby;
tz = mtbz;
Db.x = tx;
Db.y = ty;
Db.z = tz;
if((Nx-2)%tx==0)
{
Dg.x = (Nx-2)tx;
}
else
{
Dg.x = (Nx-2)tx+1;
}
if((Ny-2)%ty==0)
{
Dg.y = (Ny-2)ty;
}
else
{
Dg.y = (Ny-2)ty+1;
}
}
```

By making these changes we achieved a speed up around 5.85x.

7 Final results

Using the theory described in the first chapter, we solved the elliptic Poisson-like differential equations (9) with respect to the conformal factor ϕ and the lapse function N. In the following figures we can see the solutions in the equatorial plane (z=0).

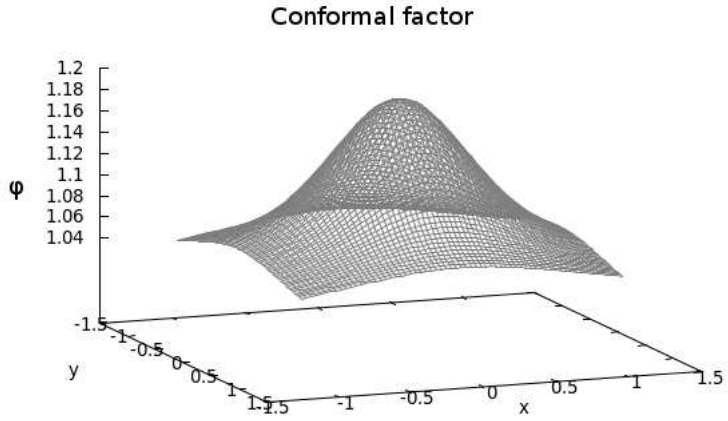


Figure 19:
Numerical solution for ϕ

In the solution above (Figure 19) we see the conformal factor which is around 1.2 in the center of the star, and the further we go from the star, the more it tends smoothly to become one, which is the case of flat spacetime.

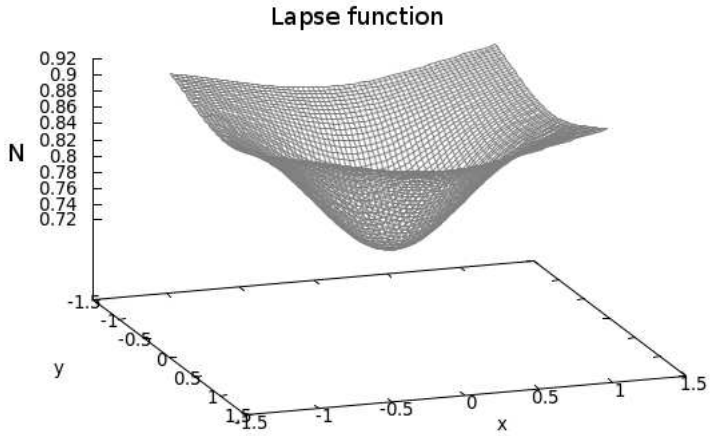


Figure 20:
The numerical solution for the lapse function N

In this solution (Figure 20) we see the lapse function which is around 0.72 in the center of the star and the further we go from it, the more it tends smoothly to become one, which is the case of flat spacetime. These solutions are with a relative accuracy of 10^{-5} and they are as expected. Now, using this solution of the conformal factor, we calculate the source function of the first differential equation (the right hand side). In the figure 21 we see a graphical representation of this source.

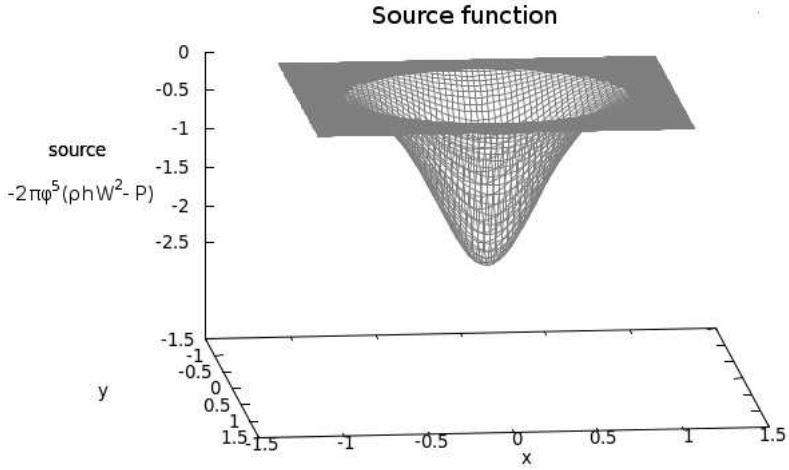


Figure 21:
The source function, using the solution ϕ

Our first attempt of solving a simple 3D Poisson equation numerically was in C without multigrid. For 3000 iterations it took 374.96 seconds, while we need 35 000 iterations to find the solution with accuracy 10^{-12}. When in the same problem we applied the *Multigrid* technique, the speed up was impressive. For 120 iterations (it needed around 90 iterations to reach the solution with accuracy 10^{-12}) it took 8.91 seconds (42 times faster).

Afterwards we solved the same simple 3D Poisson equation, but this time on GPUs. Without *Multigrid* and for 3000 iterations it took 57.48 seconds (6.52 times faster than CPU). When we applied the *Multigrid* technique on GPUs we got the solution with 120 iterations and accuracy $10^{-}12$ in 1.18 seconds. This was a speed up around 7.55 times, but still without any optimization.

	CPU	*GPU*
no_MG	374.96s	57.48s
MG	8.91s	1.186s

Table 1:
Comparing numerical solution durations

Now, going back to our specific equations, and for a grid size of 65^3, our solver on GPUs using *Multigrid* gave us the results in table 1. In the figure 22, we see how is the evolution of the residual $\|r\|_\infty$ during the 30 V-cycles of *Multigrid*.

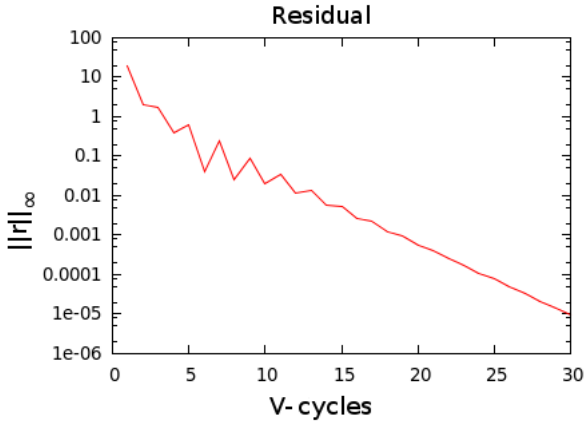


Figure 22:
The residual $\|r\|_\infty$

As we can see, in the beginning there are some instabilities. This is because of the non-linearity of the equations. Since our initial source was not the proper one, because of the rough guess for ϕ, the solution was not correct either. After a few V-cycles, when we obtain a good approximation of ϕ, the residual reduces monotonously.

These solutions were from a computer with GPU with 480 CUDA cores in double precision. After the optimization, the test continued on a different machine (GTX 460 with 336 CUDA cores, single precision). According to NVIDIA, single precision GPUs are $\sim 4.2\times$ faster than GPUs in double precision. The same solver, on the new machine gave us the same results in 2.75 seconds which is 3.26 times faster than double precision. It is less than 4.2 times because there are less CUDA cores.

After the optimizations we described in the previous section, our solver gave us the same results in 0.47 seconds which is 5.85 times faster.

So we can say that since the solver is $7.55\times$ faster on a GPU than on a CPU, without optimization, and after the optimization we get a $5.85\times$ speed up, in the end we have the solver $44.16\times$ faster on GPUs than on CPU. And probably there can be more optimizations, related to memory management.

There is one more step that could be done for a better optimization, and this is making a device-side norm calculation function for the residual. After that (or even without that), we would be ready to proceed to the simulation of a rotating neutron star.

References

1. *Relativistic simulations of rotational core collapse I. Methods, initial models, and code tests H. Dimmelmeier, J.A. Font, and E. Müller, A&A 388, 917 - 935 (2002)*

2. *Black Holes, White Dwarfs and Neutron stars: THE PHYSICS OF COMPACT OBJECTS - Stuart L. Shapiro, Saul A. Teukolsky, Wiley-Interscience, 1983*

3. *A Multigrid Tutorial, Second Edition - William L. Briggs, Van Emden Henson, Steve F. McCormic, SIAM - 2000*

4. *Wikipedia (http://www.wikipedia.org/)*

5. *NVIDIA CUDA C Programming Guide v3.1 - NVIDIA, www.nvidia.com*