# Gravitational potential modelling using a graphic processing unit with CUDA

Ante Miočić

Supervisor: dr.sc. Dejan Vinković

Split, September 2011

Bachelor Thesis in Physics

Department of Physics
Faculty of Natural Sciences and Mathematics
University of Split

**Abstract**

In this work we have implemented numerical calculation of gravitational potential on Graphical processing units (GPUs) using CUDA programing environment. GPUs execute algorithms based on massive parallelization that may result in significant computational speed ups. We tested precision of our algorithm using two examples of spherical density distributions with known analytical solutions. Speed ups achieved with GPUs compared to CPUs on these two examples was between 50 and 100 times. We also explored an extreme example of density distribution where mass exists in all cells within the computational domain. In this case a speed up of more than 100 was achieved. We conclude that GPUs are a desirable computational environment for numerical calculation of gravitational potential.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Background

## 1.1 Introduction

As technology advances so do our desires and needs to do something faster, efficient and more accurate. Many physical phenomena are nowadays simulated on a computer and for complex phenomena demanding large scale calculations we are forced to expand our techniques to satisfy these demands. Luckily we also have support form hardware manufacturers who build ever more powerful devices to serve our demands. NVIDIA has recently introduced CUDA environment that along with its powerful devices also brings new programming model to use all potential of NVIDIA graphical processing units (GPUs)

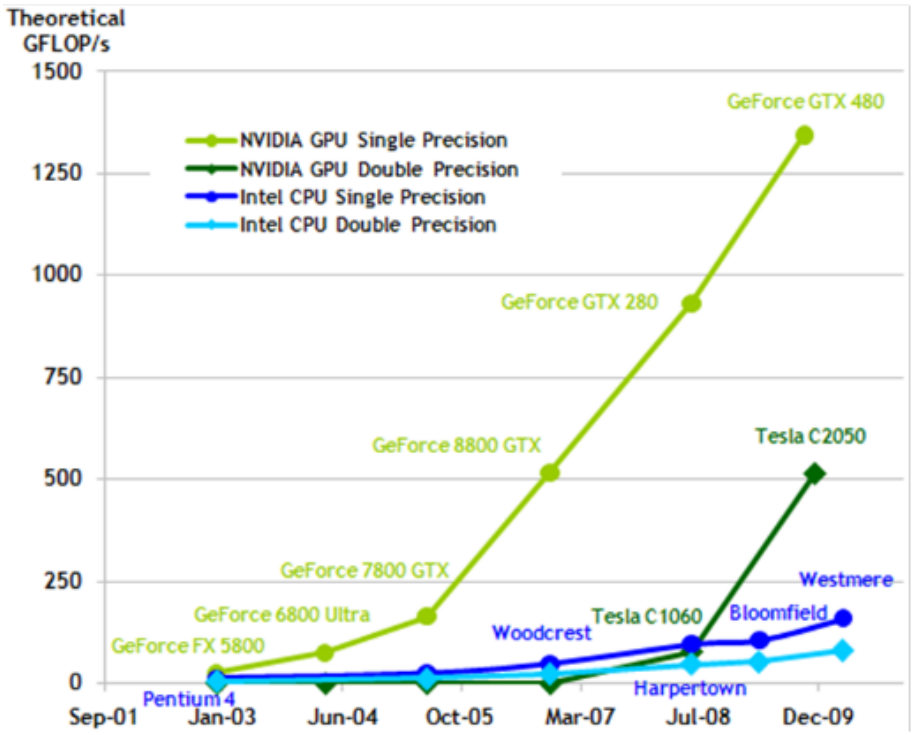


Figure 1.1: Floating point operations per second [1]

Figure 1.1 shows dramatic increase in performance of GPUs with time especially those performing single precision calculations. Since GPU architecture allows massive parallelism we can use environments like CUDA to solve our problems with much greater speed. To test speed up of CUDA we used

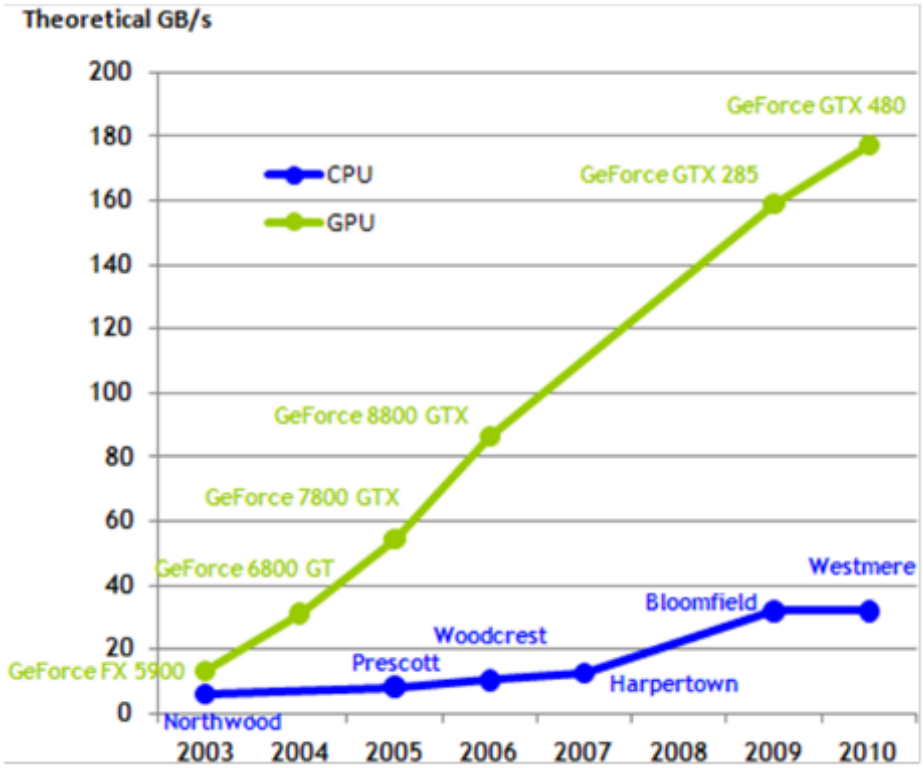


Figure 1.2: Memory bandwith for CPU and GPU [1]

gravitational potential in space as a model. By numerically calculating potential both on CPU and GPU and measuring run time for each architecture we determined speedup of performing calculations with CUDA. CUDA model can be applied to various problems in scientific numerics ranging from electrostatics and gravity all the way to black hole simulations (for more informations see [2]). For all those eager to learn more about CUDA syntax see [3].

## 1.2 CUDA-Compute unified device architecture

CUDA enables programmers to develop software that uses parallel possibilities of GPU. It unifies software environment which enables us to communicate directly with hardware and NVIDIA architecture which enables us to perform instructions simultaneously on a great number of threads.

GPU has potential to perform great number of simultaneous calculations mainly because of its design. GPU processors design dictates that majority of transistors should be rich with data processing elements as opposed to the CPU where large share of transistors consists of data caching and control flow elements (Figure 1.3). That enables software developers to run highly parallel computations on great number of data.[4]

NVIDIA's GPU processors which support CUDA model consist of Streaming Multiprocessors (abbreviated SMs). Number of multiprocessors varies form 1 to 130, average value being 30[4]. Once a grid is launched blocks of threads are arbitrarily assigned to multiprocessors. Blocks execute independently from one another and after executing one block SM automatically proceeds executing another block assigned to it. Further GPU employs Single Instruction Multiple Threads Architecture. SIMTs divide each block into a groups of 32 threads called warps[4]. Warps all execute identical instructions thereby speeding up the execution if threads in a warp are non divergent. If there is a need to synchronize threads in a block we can use __syncthreads() which will force all threads in a block to execute given commands and finalize reading and writing from shared memory.


Figure 1.3: CPU is rich with control flow (Control) and data caching (Cache) elements because these functions are important for running operating systems. Data processing elements (ALU) dominate GPU because they enable executing large number of instructions [1]

CUDA C is basically an extension of C and introduces kernel functions in C. Kernel functions can be viewed as a set of instructions executed in parallel on all threads in a grid. Kernel functions have some limitations. They:

- are not recursive;
- do not return value (i.e. type void);
- cannot call higher-order functions.

Kernel functions have prefixes:

- __ host __ function called on CPU and executed on CPU ;
- __ device __ function called on GPU and executed on GPU;
- __ global __ function called on CPU and executed on GPU.

We are mostly interested in global kernel function normally referred to simply as kernel. To completely define kernel beside providing with prefix __ global __ we also have to provide <<<... >>> execution configuration. General form of kernel function looks like

```
1 __global__ void kernel<<<dimGrid, dimBlock>>>();
```

Variables dimGrid and dimBlock are dim3 type variables and define number of blocks per grid and number of threads per block. Launching kernel creates an abstract 2D grid of 3D blocks. Maximum number of blocks dimGrid.x * dimGrid.y was for our specific GPU device 65535. Maximum number of threads dimBlock.x * dimBlock.y* dimBlock.z must not exceed 512. To determine coordinate of every thread in a block we use *struct* variable threadIdx. We can refer to threadIdx only inside kernel function.

Different memory spaces provide different access to threads, blocks and grid. Execution performance depends on what type of memory space is being used to write or read values

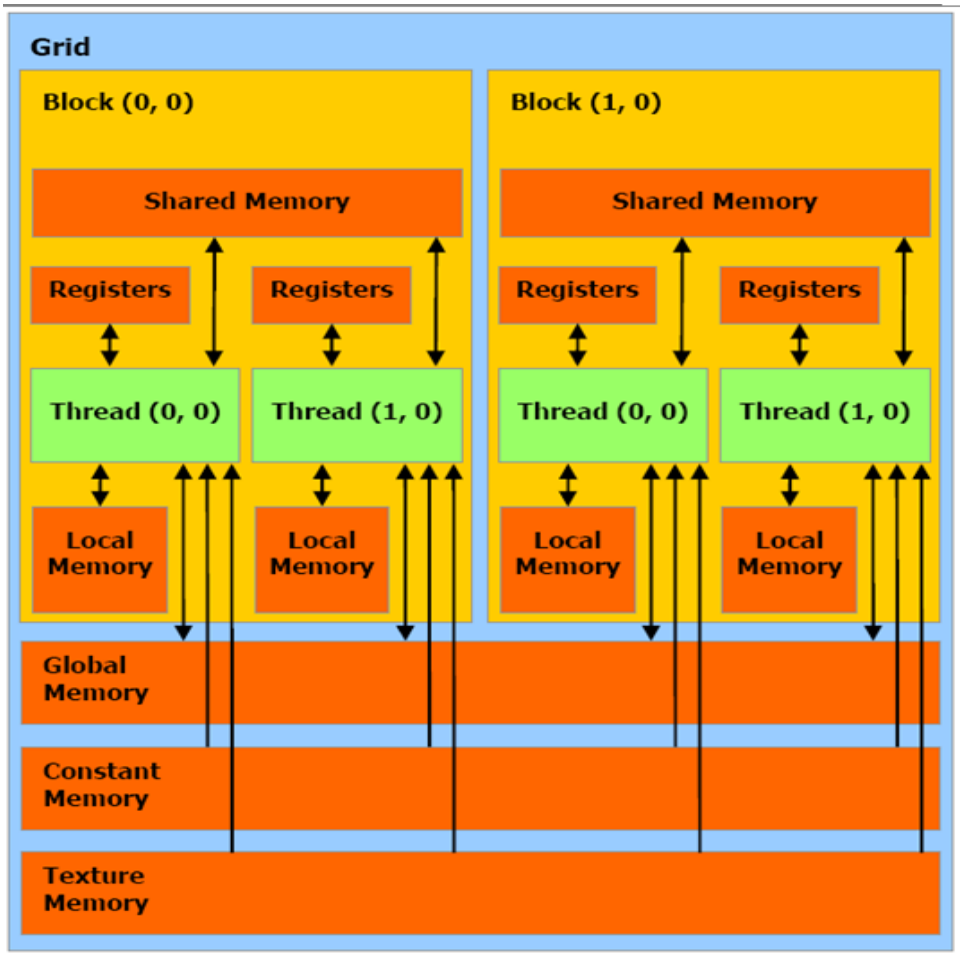


Figure 1.4: Memory hierarchy [8]— Arrows represent different memory access directions for threads execution on a GPU. For details of different type of memories see their description in Section 1.2

As illustrated by Figure 1.4 there are several types of memory spaces:

**Registers** Accessible to individual threads. Fastest memory space on GPU

**Shared memory** Accessible to all threads from a single block. If no diverging as fast as registers, supports reading and writing of data

**Constant memory** Accessible to all threads in grid but only supports reading. CPU only has the option of writing data

**Texture memory** Same features as constant but additionally provides data filtering

**Global memory** Accessible to all threads in a grid but up to 150 times slower than shared memory

**Local memory** Used to store data of individual threads but resides in global memory and hence much slower than registers

Programs written with CUDA C are compiled with nvcc compiler provided by NVIDIA. While compiling our code with nvcc CPU code is compiled independently from GPU code. Same applies to execution. Host CPU thread starts executing CPU code until it reaches kernel. Kernel creates grid and starts executing parallel instructions on GPU. CPU continues executing serial code until it reaches another kernel and creates new grid as shown in Figure 1.5. Therefore it is of great essence that before

kernel is launched other kernels have finished executing on GPU. If there are several GPU capable devices on system then CPU has to run several host threads to access each device

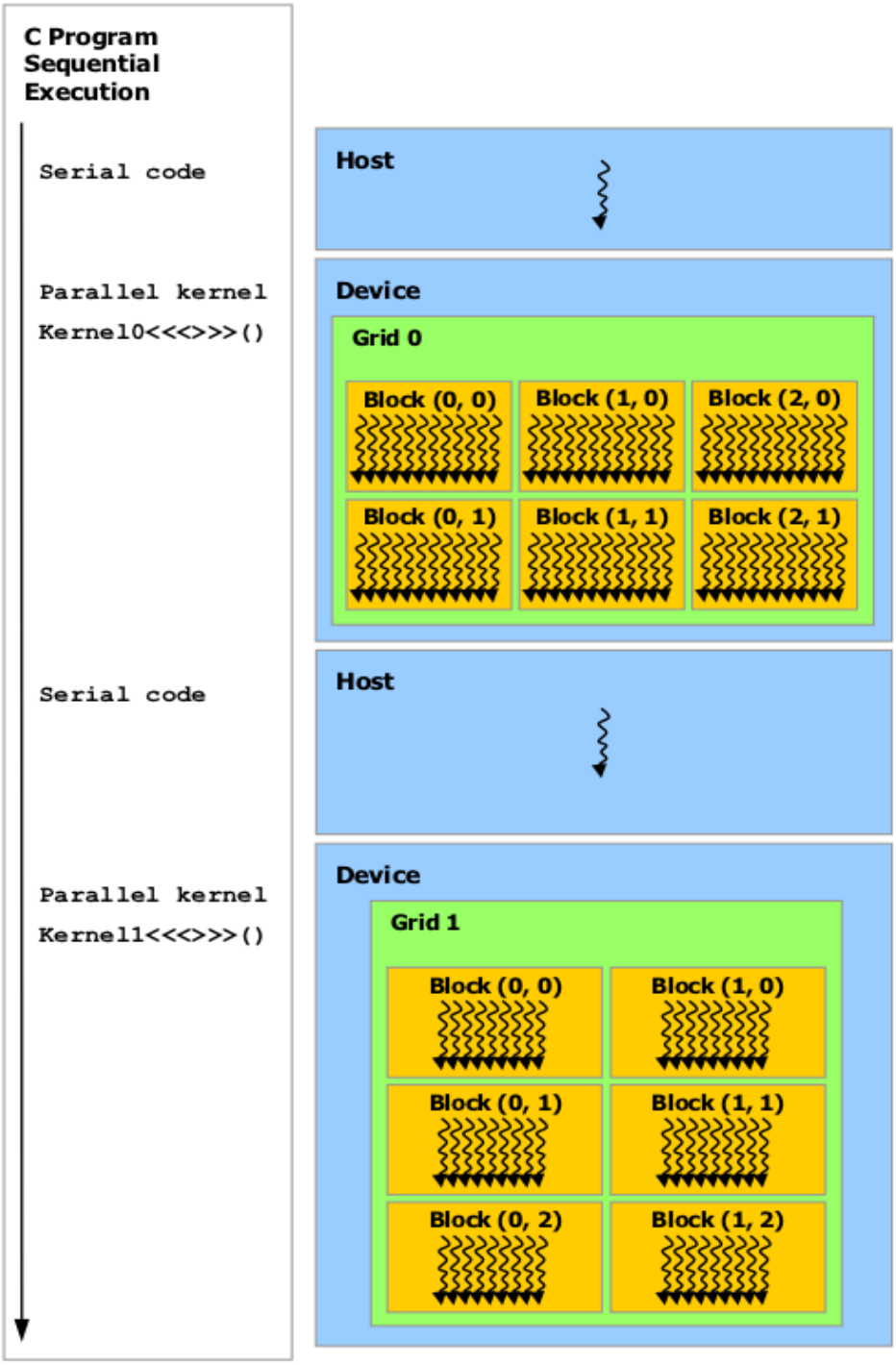


Figure 1.5: Code execution process [1]. Serial code is executed on CPU host thread (single black arrow) until it reaches kernel. After invoking kernel grid is created and kernel is executed in parallel fashion on GPU. However host thread simultaneously proceeds executing remaining CPU code until it reaches another kernel when a new grid is created

## 1.3 Gravitational potential of spherical bodies

In this work we are interested in the problem of gravitational potential. However before embarking on various large-scale problems, we decided to initially focus our work on gravitational potential of spherical bodies where we can obtain analytic results to debug and test precision of our algorithms. Let us observe gravitational potential of spherical bodies having radial function of density. Such bodies have density function $\rho = \rho(r)$.

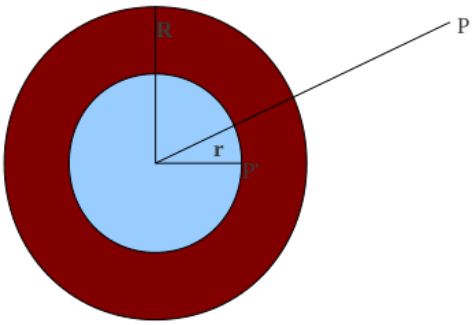


Figure 1.6: Sphere having radial density function

We are interested to determine what is the value of gravitational potential at points inside the sphere (like P') and at points outside the sphere (like P). Using Gauss theorem [5] for gravitational field

$$\oiint_S \vec{g}\, d\vec{s} = -4\pi GM$$

where S is closed surface, $\vec{g}$ gravitational field and M mass of the body enclosed by the surface S, we easily determine that gravitational field vector for any point P outside the sphere is given by

$$\vec{g} = -\frac{GM_{total}}{r^2}\hat{r}$$

$M_{total}$ being mass of the sphere having radius R and $\hat{r}$ unit vector pointing radially outward from the center of sphere. Gravitational potential is integral of gravitational field and is given by

$$V = -\frac{GM_{total}}{r},\ r > R \tag{1.1}$$

If we are interested in points P' inside the sphere we note that potential is sum 1.2 (Figure 1.7)

$V_1$ represents contribution of points ($r < x < R$), where we have used x to represent distance from the center of the sphere, to potential at point P'. $V_2$ represents contribution to the potential of sphere having radius r. Total potential at point P' is obviously

$$V = V_1 + V_2 \tag{1.2}$$

$V_2$ is easy to obtain because it's potential of the sphere having radius r calculated on it's surface. We have already solved that example using Gauss law and know that solution is $V_2 = -G\frac{M}{r}$, where M is total mass of the sphere having radius r. To calculate $V_1$ we note that we can represent our hollow sphere as an infinite number of spherical shells having radius x and thickness dx. Gravitational potential for a point inside single spherical shell having radius x is given by

$$dV_1 = -G\frac{dm}{x} = -4\pi G\rho x dx \tag{1.3}$$

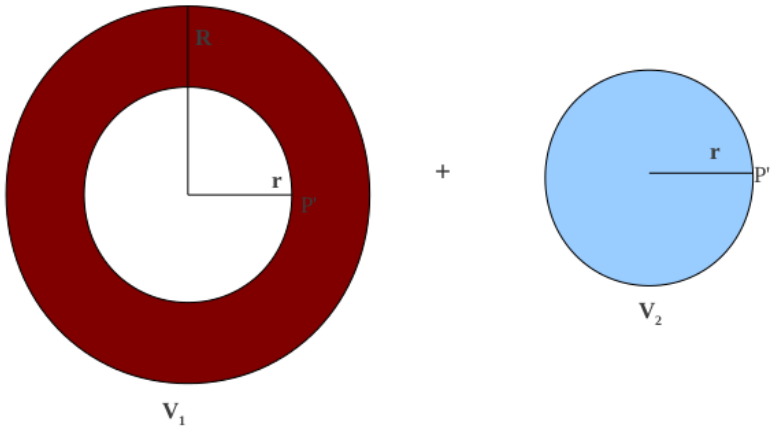


Figure 1.7: Contributions of thin shells to total potential

In order to calculate $V_1$ we just integrate (1.3) and take limits $x_1$=r and $x_2$=R.

If we are dealing with non spherical bodies or bodies having density function $\rho \neq \rho(r)$ easiest way of calculating potential is going straight to calculating numerical solution because normalized gravitational potential is defined as [6]

$$\phi(\vec{x}) = -\int_{\Omega} \frac{1}{|\vec{x} - \vec{r}|} \rho(\vec{r}) d^3 r \tag{1.4}$$

where $\Omega$ is integration domain. This integral can be very complex if density is not spherically symmetric or if integration domain $\Omega$ is not spherical and therefore we have to calculate it numerically.

# Chapter 2

# Methods

Our task was to calculate potential and determine execution time on CPU and GPU depending on solution domain. We were interested to determine actual speed up given by the following formula [7]

$$S = \frac{T_s}{T_p} \tag{2.1}$$

where $T_s$ is execution time of sequential algorithm and $T_p$ is execution time of parallel algorithm. To determine quality of our solutions we compared them to analytical solutions whenever possible. We observed three different examples involving spherically symmetric densities:

1. $\rho(r) = 1$ density of a sphere having radius R=10;
2. $\rho(r) = r^2$ density of a sphere having radius R=10;
3. $\rho(r) = \mathrm{e}^{-r^2}$ density throughout space.

Sphere was placed in center of box having linear dimensions equal to N. After finding analytical solutions for densities listed above we calculated numerical solution on CPU and GPU respectively, measuring execution time in process. Comparing analytical and numerical solutions we determined their quality and error in calculating numerical solution to analytical solution. After obtaining execution times for GPU and CPU we could determine speedup $S$.

## 2.1 Analytical solution

To determine analytical solution we refer to equations 1.1 and 1.2 Normalized potential function of gravitational potential is $\phi = \frac{V}{G}$.

As our first example we start with density function $\rho = 1$ . Employing our reasoning above we know that for a sphere of radius R=10 and given density function, analytical solution is of the form

$$\phi = \begin{pmatrix} \frac{2}{3}r^2\pi - 2\pi R^2 & r < R \\ -\frac{4\pi R^3}{3r} & r > R \end{pmatrix} \tag{2.2}$$

**Derivation**

Mass of sphere having radius $r$ and constant density $\rho = 1$

$$M = \int \rho d^3 r = 4\pi \int_0^r r'^2 \, dr' = \frac{4}{3}\pi r^3$$

Potential outside sphere:

$$\begin{aligned} M_{total} &= \frac{4}{3}\pi R^3 \\ \phi &= -\frac{M_{total}}{r} \\ \phi &= -\frac{4\pi R^3}{3r} \end{aligned}$$

Potential inside sphere:

$$\begin{aligned} \phi &= \phi_1(x=r) + \phi_2(r<x<R) \\ \phi_1 &= -\frac{M}{r} \end{aligned}$$

$$\begin{aligned} M &= \frac{4}{3}\pi r^3 \\ \phi_1 &= -\frac{4\pi r^2}{3} \end{aligned}$$

$$\phi_2 = -4\pi \int_r^R x\, dx = 2\pi r^2 - 2\pi R^2$$

$$\phi = \phi_1 + \phi_2 = \frac{2}{3} r^2 \pi - 2\pi R^2$$

As our second example we take density function to be of the form $\rho = r^2$ . Analytical solution for sphere of radius R will again be of the form

$$\phi = \begin{pmatrix} \frac{1}{5}\pi r^4 - R^4\pi & r < R \\ -\frac{4\pi R^5}{5r} & r > R \end{pmatrix} \tag{2.3}$$

**Derivation**

$$M = \int \rho d^3 r = 4\pi \int_0^r r'^4\, dr' = \frac{4}{5}\pi r^5$$

Outside the sphere:

$$\begin{aligned} M_{total} &= \frac{4}{5}\pi R^5 \\ \phi &= -\frac{M_{total}}{r} \\ \phi &= -\frac{4\pi R^5}{5r} \end{aligned}$$

Inside:

$$\begin{aligned} \phi &= \phi_1(x=r) + \phi_2(r<x<R) \\ \phi_1 &= -\frac{M}{r} \end{aligned}$$

$$\begin{aligned} M &= \frac{4}{5}\pi r^5 \\ \phi_1 &= -\frac{4\pi r^4}{5} \end{aligned}$$

$$\phi_2 = -4\pi \int_r^R x^3\, dx = r^4\pi - R^4\pi$$

$$\phi = \phi_1 + \phi_2 = \frac{1}{5}\pi r^4 - R^4\pi$$

Our third example is $\rho(r) = \mathrm{e}^{-r^2}$. This time we expand density domain for all points in rectangular box. Because our domain of integration is no longer of spherical shape we can not analytically solve integral 1.4 and have to proceed straight to numerical calculation.

## 2.2 Numerical solution and time measurement

To calculate numerical solutions I have written three independent programs:

- CPUpotential.cpp[1] CPU host code without parallelization;
- GPUpotential.cu[2] CUDA C code where potential is calculated on GPU;
- GPU+potential.cu[3] CUDA C code where both density and potential are calculated on GPU.

To numerically calculate integral 1.4 we first have to replace integral by discrete sum

$$\phi(i,j,k) = -\sum_{i',j',k'} \frac{\rho(i',j',k')}{\sqrt{(i-i')^2 + (j-j')^2 + (k-k')^2}} \Delta V \tag{2.4}$$

Since we are calculating normalized potential we can put $\Delta V = 1$ in 2.4. We can immediately deduce that to calculate potential for all points in space we have to include at least six for loops because we have 3 summation indices to calculate density at every point plus another 3 to calculate potential at every point. Our source code to calculate potential on CPU will therefore have the following outlook in C++ code

```
1  float Potential[N][N][N]; // potential
2  int I0 = (int)((float)N/2.0); // computational cube center
3  // loop over points in space to calculate Mass
4  for(i=-I0; i<=I0; i++) {
5       for(j=-J0; j<=J0; j++) {
6         for(k=-K0; k<=K0; k++) {
7
8      // mass at point (i,j,k)
9      MassCell = Density(i,j,k);
10     Mass += MassCell;
11
12     if (MassCell > 0.0) // continute only if this cell has some mass
13     // loop over points in space to calculate Potential
14         for(I=-I0; I<=I0; I++) {
15       Itemp = (I-i)*(I-i);
16       for(J=-J0; J<=J0; J++) {
17       Jtemp= (J-j)*(J-j);
18       for(K=-K0; K<=K0; K++) if (I!=i || J!=j || K!=k) {
19           Ktemp = (K-k)*(K-k);
```

[1] Check appendix A for complete code
[2] Check appendix B for complete code
[3] Check appendix B for complete code

```
          // add contribution to the potential at (I,J,K)
  Potential[I+I0][J+J0][K+K0] -= (MassCell/sqrt(Itemp+Jtemp+Ktemp));
} } } // end of loop over points in space to calculate Potential

} } } // end of loop over points in space to calculate Mass
```

We notice that all the values are written in 3D array Potential[][][]. First three loops are loops over space. We examine every point to find its contribution to potential at other points. We can exclude points where density is $\rho = 0$ i.e. empty space because they do not contribute to potential. If $\rho \neq 0$ we then proceed to add contribution of that specific mass to all other points in space.

Intuitively we can guess that we can greatly speed up the execution if we calculate in parallel contribution of mass to other points. To do this we employ kernel function that will be executed on GPU when called.

```
//kernel function
__global__ void racunaj(float *d_potential, int N, unsigned int Ntot,
 int l, int m, int n, float Mass)
{
   //thread global coordinate
   unsigned int tidx = blockIdx.x * blockDim.x + threadIdx.x;
   int I0=(int)((float)N/2.); // center of  sphere

   // potential at (i,j,k)
   float temp_potential = 0.0;

   if (tidx<Ntot){

     //calculate coordinates (i, j, k) of this thread
     int i = tidx%N;
     int j = ((tidx-i)/N)%N;
     int k = (((tidx-i)/N)-j)/N;

     // shift to the ceter
     i -= I0; j -= I0; k -= I0;

     if (i!=l || j!=m || k!=n) {
       // contribution to the potential at (i,j,k) from (l,m,n)
       temp_potential= d_potential[tidx]; // old value
       temp_potential -= Mass / sqrtf((i-l)*(i-l) + (j-m)*(j-m) +
       (k-n)*(k-n));
       d_potential[tidx]=temp_potential; // new value
     }
   }

} // end of: __global__ void racunaj
```

Doing this we have essentially transferred calculations of point contribution to device where they are being executed in parallel therefore effectively we have reduced our total of 6 for loops to 3. One of many new features is introduction of thread coordinate. Although we have a 3D problem for optimization reasons (memory access on GPU) we calculate it using 1D grid of blocks all having just one dimension by introducing global index tidx. Tidx has range $[0 : N^3 - 1]$ and can be used to index values of *d_potential*, an array to which we store values of potential calculated on device. *d_potential* holds

addresses to memory space allocated on global memory of GPU. After executing kernel for a single point we do the same for all points in space. First we set potential to zero for every point in space and copy that to GPU array *d_potential*. If domain of our density function is a sphere we calculate density of each point on CPU because of small amount of data to be processed. After calculating mass of each point we forward that information as an argument to our kernel function which then executes series of parallel instructions on GPU.

```
 for(i=-I0; i<=I0; i++) {
     for(j=-I0; j<=I0; j++) {
       for(k=-I0; k<=I0; k++) {

   // mass at point (i,j,k)
   MassCell = Density(i,j,k);

   if (MassCell > 0.0) // continute only if this cell has some mass
   // run krenel
   racunaj<<< nBlocks, nThreads >>>(d_potential, N, Ntot, i,j,
   k, MassCell);

   } } } // end of loop over points in space to calculate Mass
////////////////////////////////
  // copy device array d_potential to host array h_potential
  cudaMemcpy(h_potential, d_potential, size, cudaMemcpyDeviceToHost);
```

After looping through all space we copy data stored in device array to host array. If domain for our density function is all space then it would be optimal solution to calculate mass of all points on GPU and store them back to CPU array because for complex mass distributions calculating mass individually for each particle tends to be more time exhausting. Way to do this is shown in GPU+.cu. To measure time of execution one simply has to insert code in between following instructions

```
 static long myclock()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec * 1000000) + tv.tv_usec;
}
// initial final time difference function
double getRuntime(long* end, long* start)
{
    return (*end - *start);
}

int main(){
     .
     .
     .
  // capture initial time
  long start = myclock();

  //.......CODE TO BE EXECUTED.............

  //capture final time
```

```
23      long end = myclock();
24      // display execution time
25  std::cout << "" << std::setprecision(3)
26  << getRuntime(&end, &start)/1000000.0 << "" << std::endl;
27        .
28        .
29  return 0;
30  }
```

# Chapter 3

# Results

We observed three distinct problems involving three different density functions listed on page 8 in their respectful domains. To determine quality of our numerical calculations we first found analytical solution to the problem which was later compared to numerical. We determined speedup coefficient $S$ after performing series of calculations on CPU and GPU for different solution domain dimensions (i.e. different N) and determining respectful times of execution. All calculations where performed on CPU and GPU devices listed in Tables 3.1 and 3.2.

## 3.1 Problem 1

Density function of sphere having radius R=10 is $\rho = 1$. Analytical solution for this problem is given with function 2.2. Graph of analytical and numerical solution is displayed on Figure 3.1 together with associated error. Figures are plotted for N=201 number of dots along each linear length of our solution domain.

Execution time comparison for different values of N between CPU and GPU code is presented on Figure 3.2 in log-log format. Speedup for N=121 is $S = 52.6$

## 3.2 Problem 2

Density function of sphere having radius R=10 is $\rho = r^2$. Analytical solution for this problem is given with function 2.3. Graph of analytical and numerical solution is displayed on Figure 3.3 together with associated error. Figures are plotted for N=201 number of dots along each linear length of our solution domain.

Table 3.1: CPU device properties

| **Intel Xeon** | | | |
|---|---|---|---|
| Version | E5520 | Device ID | 0 |
| Vendor | Intel Corp. | Size | 1600 MHz |
| Capacity | 1600 MHz | Width | 64 bits |

Table 3.2: GPU device properties

| **GeForce GTX 285** | | | |
|---|---|---|---|
| Total global memory | 1023 Mb | Registers per block | 16384x32 bits |
| Shared Memory per block | 16 kb | Warp size | 32 |
| Total constant memory | 64 kb | Maxthreads per block | 512 |
| Revision number | 1.3 | Multiprocessors | 130 |

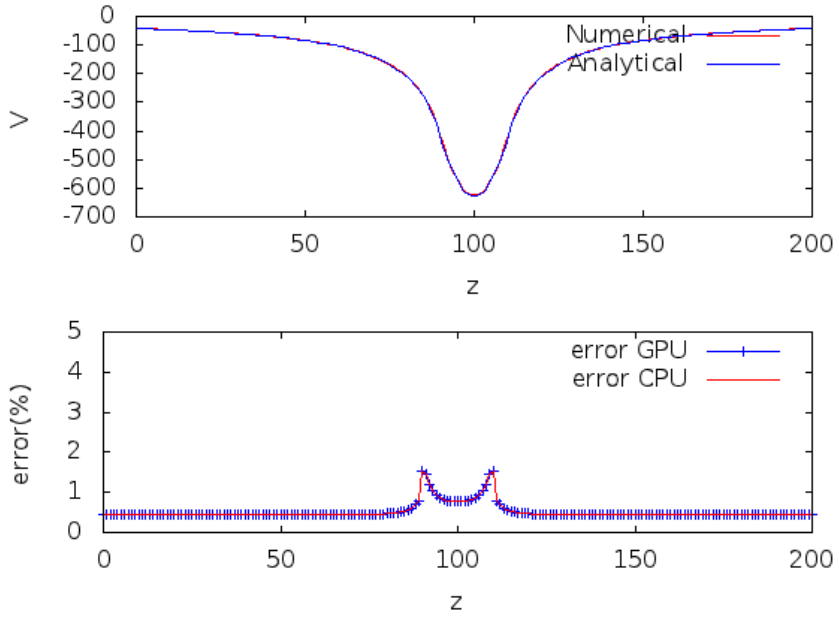


Figure 3.1: Analytical solution and deviation from analytical solution for $\rho = 1$. Analytical and numerical solution displayed on top part of the figure match each other

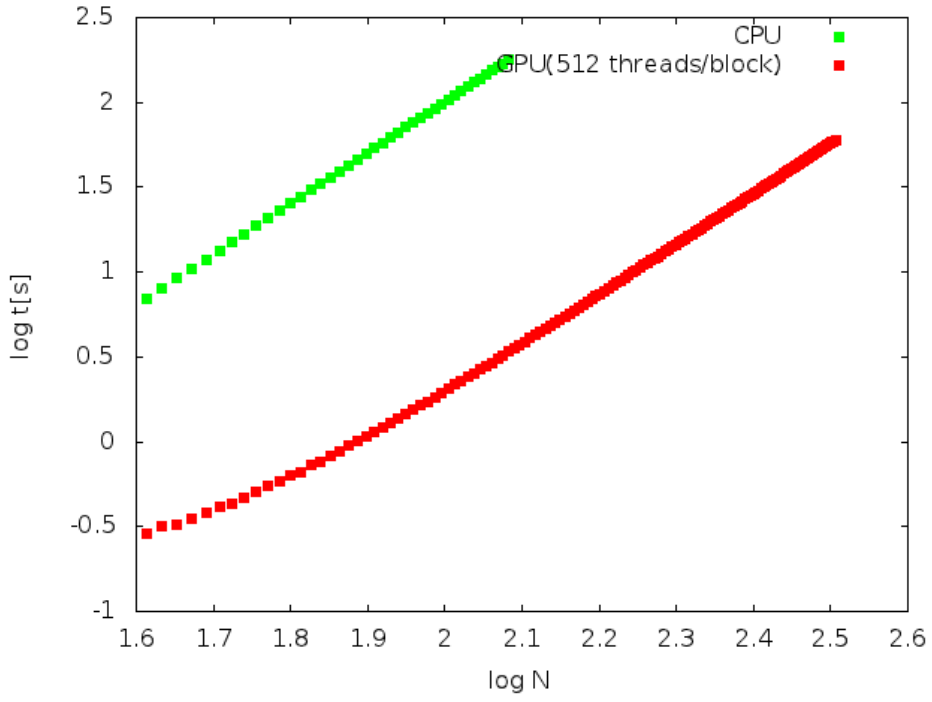


Figure 3.2: Execution time dependence on number of steps N for $\rho = 1$

Execution time comparison for different values of N between CPU and GPU code is presented on Figure 3.4 in log-log format. Speedup for N=121 is $S = 52.6$

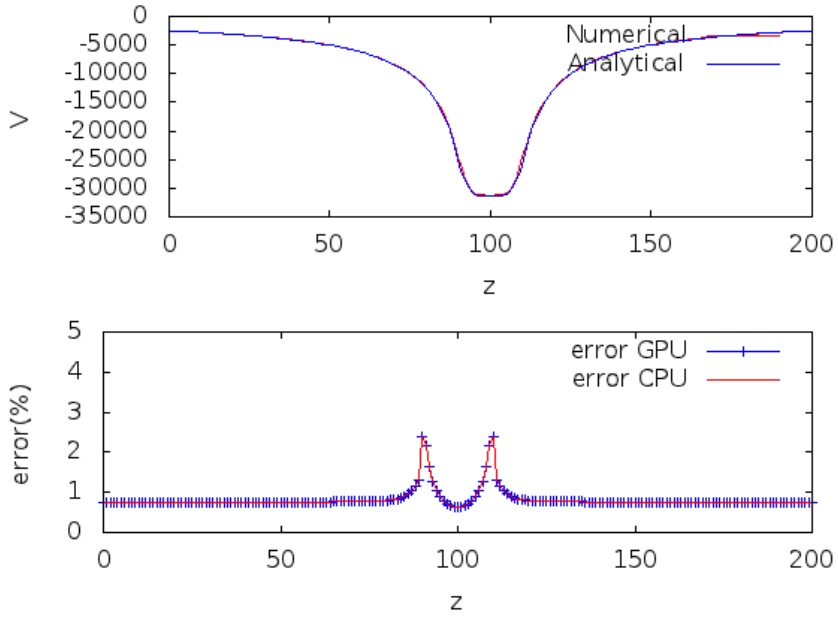


Figure 3.3: Analytical solution and deviation from analytical solution for $\rho = r^2$. Analytical and numerical solution displayed on top part of the figure match each other

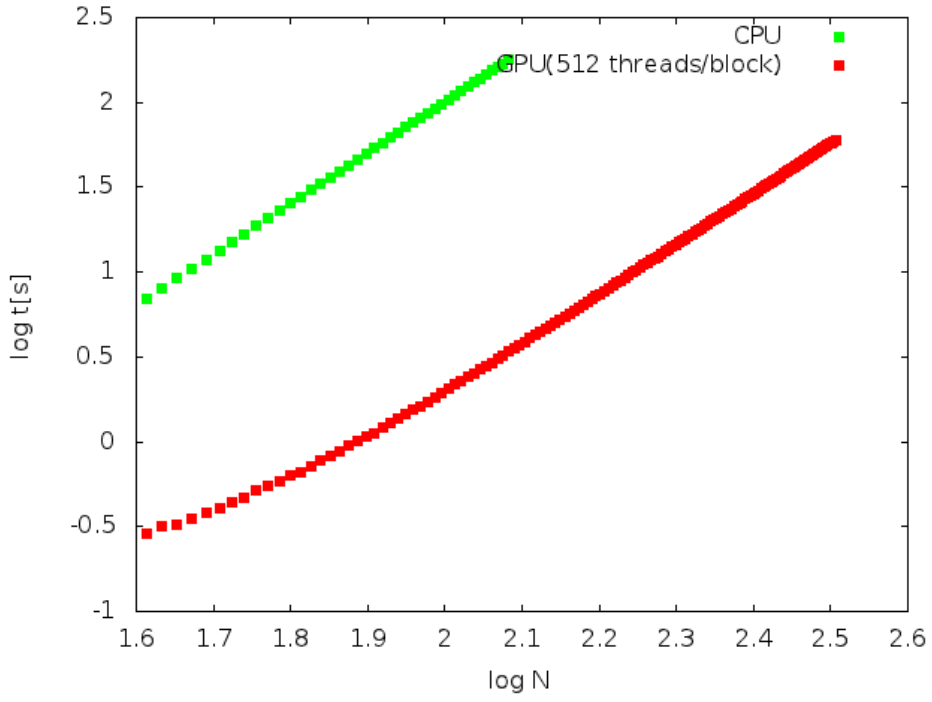


Figure 3.4: Execution time dependence on number of steps N for $\rho = r^2$

## 3.3 Problem 3

Density function is given by $\rho = \mathrm{e}^{-r^2}$ with domain extended to entire space. This example is important because here we have to calculate density for all points in the computational domain, while before

we had all density contained within a sphere of radius $R = 10$. Therefore here we try two different approaches: calculating density on CPU and calculating both density and potential on GPU to further optimize our code. Numerical solution is displayed on Figure 3.5 while execution time comparison for different values of N between CPU and GPU code is presented on Figure 3.6 in log-log format. Speedup for N=121 is $S = 88.0$ for GPU code and $S = 111.7$ for GPU+ code. Figures are plotted for N=201 number of dots along each linear length of our solution domain.

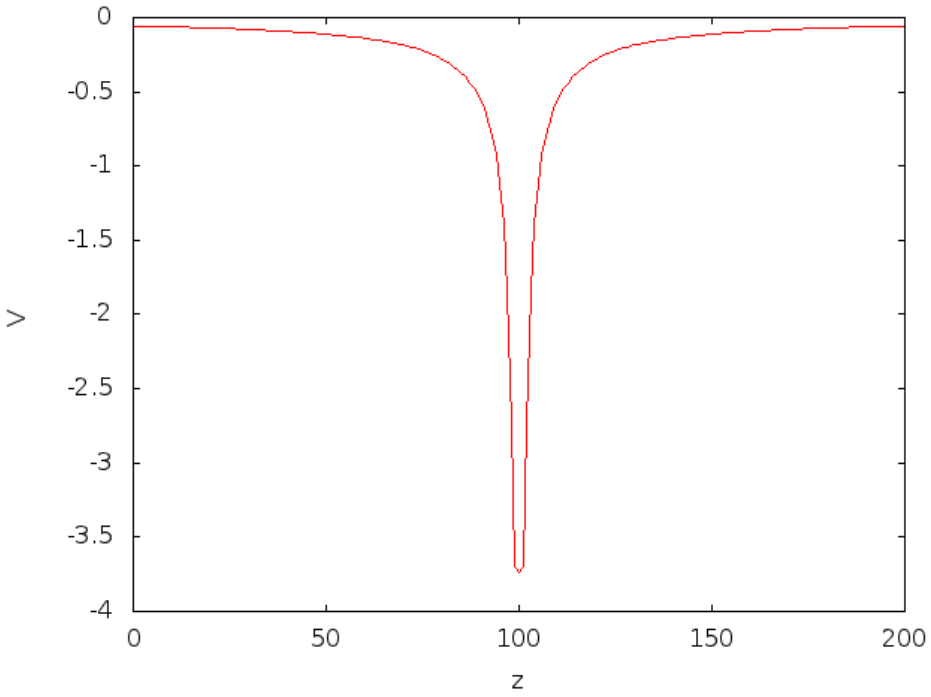


Figure 3.5: Numerical solution for $\rho = \mathrm{e}^{-r^2}$

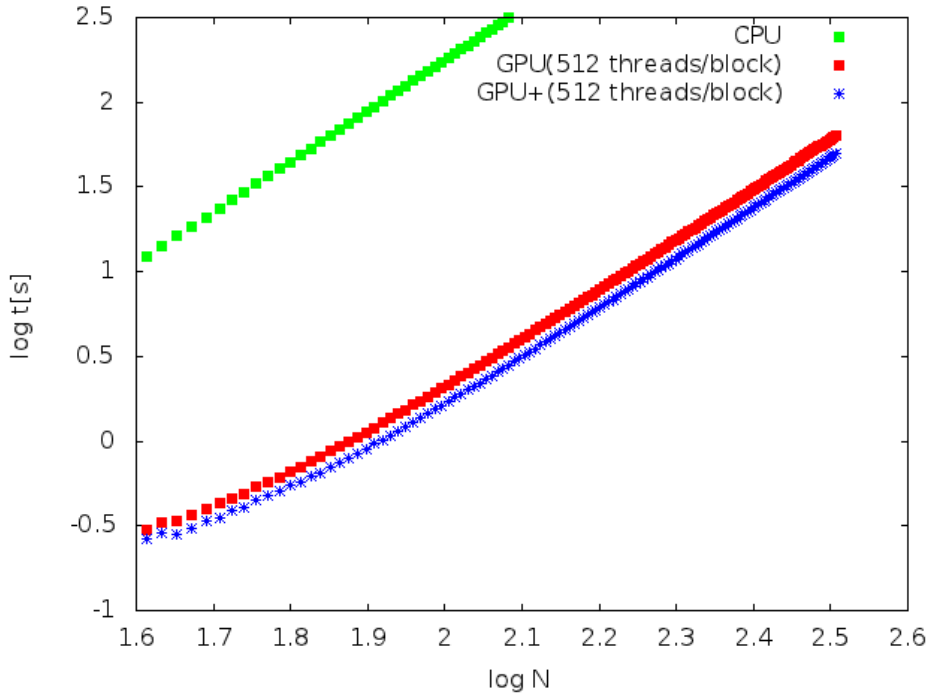


Figure 3.6: Execution time dependence on number of steps N for $\rho = \mathrm{e}^{-r^2}$. GPU represents execution time dependence obtained calculating density on CPU, while GPU+ represents execution time dependence obtained calculating density on GPU

# Chapter 4

# Discussion

First we can notice quality of our solutions in figures 3.1 and 3.3. Solutions are quite stable outside the sphere and some minor deviations start as we approach surface of sphere. Here potential has to satisfy continuity condition and also first derivative of potential has to be continuous. Deviations arise because our algorithm cannot accurately approximate spherical surface of density domain. Error decreases as radius of our density domain is expanded to larger values. As we approach to center of sphere error steadily reduces to values close to the ones for points outside sphere. Hence we can deduce that for non divergent density functions (like the ones in real world) solutions will be quite stable as long as we are far from borders of density domain. Since in real world our domain is limitless for real world application we need not bother with boundary errors. So for limitless domains our error pattern will look like the one inside sphere of density $\rho$. We can also conclude that GPU and CPU calculations equally well approximate solution to potential while working in single precision because of the same error pattern shown in figures.

Figures 3.2, 3.4 and 3.6 display execution time dependence. We can see that for figures 3.2 and 3.4 execution is greatly accelerated when our calculations are performed using CUDA model. Running time is decreased more than 50 times for N=121 steps and continues to decrease as we perform calculations for greater N's albeit in slower fashion. Because we have similar density functions and equal density domains our times of execution do not differ between two problems. But for problem 3.6 where we calculate potential in all points of solution domain execution time starts increasing rapidly for CPU because of extended domain. Execution of CUDA algorithm is accelerated 88 times for N=121 and if density is also calculated on GPU acceleration jumps to 111 times because more of a code is executed in parallel fashion. Code could have been accelerated at even greater rates if we employed mathematical functions with lesser precision but that would have contributed much more to overall error.

# Chapter 5

# Conclusions and recommendations

## 5.1 Conclusions

We have shown calculation of gravitational potential can be dramatically faster if implemented on GPUs. Performing calculations in CUDA environment speeds up calculations multiple times especially for calculations involving huge data sets that are processed with same set of instructions. To successfully utilise all possibilities of parallel programming user has to be familiar with CUDA thread hierarchy and memory hierarchy. More powerful CUDA capable devices execute algorithm with greater efficiency as does code that is highly optimized and adapt to CUDA environment using all its features. We have observed this effect in calculating gravitational potential in a region of space having some mass distribution $\rho$. First we have, if it was possible, determined analytical solution of problem. To test performance of CUDA model we wrote a separate code for CPU and one to be used in CUDA environment. After executing code and comparing analytical to numerical solutions it was shown that numerical solution approximates analytical solution with great accuracy. After measuring run time for both CPU and CUDA code it was shown that CUDA code speeds up calculations to over 100 times with speedup factor being greater for huge data sets with maximum parallelization of all operations to be executed on data. Therefore advantages of CUDA model are quite obvious for large problems where execution of code can last up to day or two. However even though more powerful hardware offers greater advantages to user, essential knowledge for successful solution to the problem is deep understanding of CUDA model and intuition to resolve problem on independent sub-problems that can itself be further resolved into smaller parts executed in parallel

## 5.2 Recommendations

To successfully solve given problems one should first become familiar with everything CUDA environment offers but also with its limitations. Although CUDA is very powerful environment it also requires basic understanding and logic so approach to problem should not be just from programming point of view but also from everyday experience in solving different problems that exhibit certain parallelism. We can also improve precision using self-adaptive grid close to the surface of density domain thereby more accurately approximate density surface. For spherical cases a spherical grid would be a better choice, but we use spherical case only as a test of a general situation.

# Appendix A

# CPU code

Listing A.1: CPU source code

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>
#include <iostream>
#include <iomanip>

float Density(int i,int j,int k) { // mass density
  return exp(-(i*i+j*j+k*k));
}

float Mass; // total mass (integral of Density)
// capture moment function
static long myclock()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec * 1000000) + tv.tv_usec;
}
// initial final time difference function
double getRuntime(long* end, long* start)
{
    return (*end - *start);
}


int main(int argc, char *argv[])
{
   int N=atoi(argv[1]); //input to exe file
   float Potential[N][N][N]; // potential
   FILE *output;
   int I, J, K; // counter over Potential
   int i, j, k; // counter over Density
   // computational cube center
   int I0 = (int)((float)N/2.0);
```

```
36    int J0 = I0, K0 = I0;
37    float MassCell; // mass of a cell
38    int Itemp, Jtemp, Ktemp;
39    long start = myclock();
40    // set potential to zero:
41    for(I=-I0; I<=I0; I++) {
42      for(J=-J0; J<=J0; J++) {
43        for(K=-K0; K<=K0; K++) Potential[I+I0][J+J0][K+K0]=0.0;
44      }
45    }
46
47    // set total mass to zero:
48    Mass = 0.0;
49
50    // loop over points in space to calculate Mass
51    for(i=-I0; i<=I0; i++) {
52      for(j=-J0; j<=J0; j++) {
53        for(k=-K0; k<=K0; k++) {
54
55    // mass at point (i,j,k)
56    MassCell = Density(i,j,k);
57    Mass += MassCell;
58
59    if (MassCell > 0.0) // continute only if this cell has some mass
60    // loop over points in space to calculate Potential
61    for(I=-I0; I<=I0; I++) {
62      Itemp = (I-i)*(I-i);
63      for(J=-J0; J<=J0; J++) {
64        Jtemp= (J-j)*(J-j);
65        for(K=-K0; K<=K0; K++) if (I!=i || J!=j || K!=k) {
66           Ktemp = (K-k)*(K-k);
67           // add contribution to the potential at (I,J,K)
68    Potential[I+I0][J+J0][K+K0] -= (MassCell/sqrt(Itemp+Jtemp+Ktemp));
69
70    } } } // end of loop over points in space to calculate Potential
71
72    } } } // end of loop over points in space to calculate Mass
73    long end = myclock();
74    //return time of execution
75    std::cout << " " << std::setprecision(3)
76    << getRuntime(&end, &start)/1000000.0 << " " << std::endl;
77
78    return 0;
79 }
```

# Appendix B

# GPU code

Listing B.1: GPU source code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda.h>
#include <sys/time.h>
#include <iostream>
#include <iomanip>

#define BLOCK 512

//density fuction
float Density(int i,int j,int k) { // mass density
    return exp(-(i*i+j*j+k*k));// return density at point in question
     /*
  if mass density is present only inside sphere R=10
    if (i*i+j*j+k*k<=10*10) return 1.0; // constant density
      else return 0.0;
  */
}

//kernel function
__global__ void racunaj(float *d_potential, int N, unsigned int Ntot,
int l, int m, int n, float Mass)
{
   //thread global coordinate
   unsigned int tidx = blockIdx.x * blockDim.x + threadIdx.x;
   int I0=(int)((float)N/2.); // center of  sphere

   // potential at (i,j,k)
   float temp_potential = 0.0;

   if (tidx<Ntot){

     //calculate coordinates (i, j, k) of this thread
     int i = tidx%N;
```

```
36      int j = ((tidx-i)/N)%N;
37      int k = (((tidx-i)/N)-j)/N;
38
39      // shift to the ceter
40      i -= I0; j -= I0; k -= I0;
41
42      if (i!=l || j!=m || k!=n) {
43        // contribution to the potential at (i,j,k) from (l,m,n)
44        temp_potential= d_potential[tidx]; // old value
45        temp_potential -= Mass / sqrtf((i-l)*(i-l) + (j-m)*(j-m) +
46        (k-n)*(k-n));
47        d_potential[tidx]=temp_potential; // new value
48      }
49    }
50
51 } // end of: __global__ void racunaj
52
53 // error check function
54 void checkCUDAError(const char *msg)
55 {
56     cudaError_t err = cudaGetLastError();
57     if( cudaSuccess != err)
58     {
59         fprintf(stderr, "Cuda error: %s: %s.\n", msg,
60                         cudaGetErrorString( err) );
61         exit(EXIT_FAILURE);
62     }
63 }
64 // capture moment function
65 static long myclock()
66 {
67     struct timeval tv;
68     gettimeofday(&tv, NULL);
69     return (tv.tv_sec * 1000000) + tv.tv_usec;
70 }
71 // initial final time difference function
72 double getRuntime(long* end, long* start)
73 {
74     return (*end - *start);
75 }
76
77 int main(int argc, char *argv[])
78 {
79     int N=atoi(argv[1]);
80     // exe file arguments, N number of stepsop
81     unsigned int Ntot=N*N*N;
82     //gravitational potential at (x,y,z), pointer to host memory
83     float *h_potential;
84     //gravitational potential at (x,y,z), pointer to device memory
85     float *d_potential;
```

```
86      int I0=(int)((float)N/2.); // (I0, J0, K0) center of  sphere
87      size_t size=N*N*N*sizeof(float);//size of memory to be allocated
88      int i,j,k;
89      float MassCell;
90      // capture initial time
91      long start = myclock();
92
93      //allocate memory to host
94      h_potential=(float *)malloc(size);
95      // allocate memory to device
96      cudaMalloc((void**) &d_potential, size);
97
98      // set potential values to zero
99      for (i=0; i<Ntot; i++) h_potential[i]=0.0;
100     // transfer zero values to device
101   cudaMemcpy(d_potential, h_potential, size, cudaMemcpyHostToDevice);
102
103
104     dim3 nThreads(BLOCK); // thread dimensions
105     dim3 nBlocks((int)(Ntot/BLOCK)+1);// Block dimension
106
107 ////////////////////////////////
108    // loop over points in space to calculate Mass
109    for(i=-I0; i<=I0; i++) {
110      for(j=-I0; j<=I0; j++) {
111        for(k=-I0; k<=I0; k++) {
112
113    // mass at point (i,j,k)
114    MassCell = Density(i,j,k);
115
116    if (MassCell > 0.0) // continute only if this cell has some mass
117    // run krenel
118    racunaj<<< nBlocks, nThreads >>>(d_potential, N, Ntot,
119    i,j,k, MassCell);
120
121    } } } // end of loop over points in space to calculate Mass
122 ////////////////////////////////
123     //checkCUDAError("kernel invocation");
124    // copy device array d_potential to host array h_potential
125   cudaMemcpy(h_potential, d_potential, size, cudaMemcpyDeviceToHost);
126     //checkCUDAError("memcpy");
127
128     //capture final time
129     long end = myclock();
130     // display execution time
131     std::cout << "" << std::setprecision(3)
132     << getRuntime(&end, &start)/1000000.0 << "" << std::endl;
133
134
135     FILE *output;
```

```
136     output=fopen("GPU_pot.txt", "w");
137
138     // write out potential in points (IO, J0, k)
139
140     i=I0;j=I0;
141     //for(i=0;i<N; i++)
142     //for(j=0;j<N; j++)
143     for(k=0;k<N; k++)
144        fprintf(output, "%d\t%e\n", k, h_potential[i+N*(j+k*N)]);
145
146     fclose(output);
147     //free memory space
148     cudaFree(d_potential); free(h_potential);
149
150     return 0;
151 }
```

Listing B.2: GPU+ source code

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <math.h>
 4 #include <cuda.h>
 5 #include <sys/time.h>
 6 #include <iostream>
 7 #include <iomanip>
 8
 9 #define BLOCK 512
10
11 //kernel density fuction
12 __global__ void Density(float *d_density, int N,
13 unsigned int Ntot) { // mass density
14
15    //thread global coordinate
16    unsigned int tidx = blockIdx.x * blockDim.x + threadIdx.x;
17    int I0=(int)((float)N/2.); // center of  sphere
18
19      if (tidx<Ntot){
20
21      //calculate coordinates (i, j, k) of this thread
22      int i = tidx%N;
23      int j = ((tidx-i)/N)%N;
24      int k = (((tidx-i)/N)-j)/N;
25
26      // shift to the ceter
27      i -= I0; j -= I0; k -= I0;
28      //write density at (i, j, k) to d_density
29      d_density[tidx]=expf(-(i*i+j*j+k*k));
30 }
31 }
32 //kernel potential function
```

```
__global__ void racunaj(float *d_potential, int N, unsigned int Ntot,
int l, int m, int n, float Mass)
{
   //thread global coordinate
   unsigned int tidx = blockIdx.x * blockDim.x + threadIdx.x;
   int I0=(int)((float)N/2.); // center of  sphere

   // potential at (i,j,k)
   float temp_potential = 0.0;

   if (tidx<Ntot){

     //calculate coordinates (i, j, k) of this thread
     int i = tidx%N;
     int j = ((tidx-i)/N)%N;
     int k = (((tidx-i)/N)-j)/N;

     // shift to the ceter
     i -= I0; j -= I0; k -= I0;

     if (i!=l || j!=m || k!=n) {
       // contribution to the potential at (i,j,k) from (l,m,n)
       temp_potential= d_potential[tidx]; // old value
       temp_potential -= Mass / sqrtf((i-l)*(i-l) +
       (j-m)*(j-m) + (k-n)*(k-n));
       d_potential[tidx]=temp_potential; // new value
     }
   }

} // end of: __global__ void racunaj

// error check function
void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.\n", msg,
                     cudaGetErrorString( err) );
        exit(EXIT_FAILURE);
    }
}
// capture moment function
static long myclock()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec * 1000000) + tv.tv_usec;
}
// initial final time difference function
```

```
83 double getRuntime(long* end, long* start)
84 {
85     return (*end - *start);
86 }
87
88 int main(int argc, char *argv[])
89 {
90     int N=atoi(argv[1]);
91     int thread_loop;
92     // exe file arguments, N number of steps
93     unsigned int Ntot=N*N*N;
94      //gravitational potential at (x,y,z), pointer to host memory
95     float *h_potential;
96     //gravitational potential at (x,y,z), pointer to device memory
97     float *d_potential;
98     //density at (x, y ,z), host and device pointers
99     float *h_density,*d_density;
100    int I0=(int)((float)N/2.); // (I0, J0, K0) center of  sphere
101    size_t size=N*N*N*sizeof(float);//size of memory to be allocated
102    int i,j,k;
103    float MassCell;
104    // capture initial time
105    long start = myclock();
106
107    //allocate memory to host
108    h_potential=(float *)malloc(size);
109    h_density=(float *)malloc(size);
110    // allocate memory to device
111    cudaMalloc((void**) &d_potential, size);
112    cudaMalloc((void**) &d_density, size);
113
114    // set potential values to zero
115    for (i=0; i<Ntot; i++) h_potential[i]=0.0;
116    // transfer zero values to device
117  cudaMemcpy(d_potential, h_potential, size, cudaMemcpyHostToDevice);
118
119
120    dim3 nThreads(BLOCK); // thread dimensions
121    dim3 nBlocks((int)(Ntot/BLOCK)+1);// Block dimension
122    //calculate density for each point in space
123    Density<<< nBlocks, nThreads >>>(d_density, N, Ntot);
124    //copy data back to CPU
125    cudaMemcpy(h_density, d_density, size, cudaMemcpyDeviceToHost);
126
127 ////////////////////////////
128   // loop over points in space to calculate Mass
129   for(i=-I0; i<=I0; i++) {
130     for(j=-I0; j<=I0; j++) {
131       for(k=-I0; k<=I0; k++) {
132
```

```
133    // mass at point (i,j,k)
134    MassCell = h_density[(i+I0)+N*((j+I0)+(k+I0)*N)];
135
136    if (MassCell > 0.0) // continute only if this cell has some mass
137     // run krenel
138    racunaj<<< nBlocks, nThreads >>>(d_potential, N, Ntot,
139    i,j,k, MassCell);
140
141    } } } // end of loop over points in space to calculate Mass
142 ////////////////////////////////
143     //checkCUDAError("kernel invocation");
144     // copy device array d_potential to host array h_potential
145   cudaMemcpy(h_potential, d_potential, size, cudaMemcpyDeviceToHost);
146     //checkCUDAError("memcpy");
147
148     //capture final time
149     long end = myclock();
150     // display execution time
151     std::cout << "" << std::setprecision(3)
152     << getRuntime(&end, &start)/1000000.0 << "" << std::endl;
153
154
155     FILE *output;
156     output=fopen("GPU_pot.txt", "w");
157
158     // write out potential in points (IO, J0, k)
159
160     i=I0;j=I0;
161     //for(i=0;i<N; i++)
162     //for(j=0;j<N; j++)
163     for(k=0;k<N; k++)
164        fprintf(output, "%d\t%e\n", k, h_potential[i+N*(j+k*N)]);
165
166     fclose(output);
167     //free memory space
168     cudaFree(d_potential); free(h_potential);
169     cudaFree(d_density); free(h_density);
170
171     return 0;
172 }
```

# Bibliography


[1] NVIDIA Corporation. *NVIDIA CUDA C programming guide*. NVIDIA, http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf, 2010.

[2] Wen mei W. Hwu, editor. *GPU computing gems, emerald edition*. Elsevier Inc., 2011.

[3] Jason Sanders and Edward Kandrot, editor. *CUDA by Example*. Addison-Wesley, 2010.

[4] M. Piskorec, "Programiranje na GPU procesorima uz pomoc CUDA arhitekture", FER, 2009.

[5] WIKIPEDIA, Gauss law, Semptember 2011. http://en.wikipedia.org/wiki/Gauss%27_law_for_gravity.

[6] WIKIPEDIA, Gravitational potential, September 2011. http://en.wikipedia.org/wiki/Gravitational_potential.

[7] WIKIPEDIA, Speedup, July 2011. http://en.wikipedia.org/wiki/Speedup.

[8] Electronic visualization laboratory http://www.evl.uic.edu/aej/525/pics/cuda_memory.jpg.