Universität Zürich

Bachelor Thesis in Physics

Fast cosmological N-body simulations on graphics processing units

Author:
Simon Schwegler

Supervisor:
Dr. Joachim Stadel

December 7, 2011

Contents

Abstract

This thesis adresses the development, implementation and theoretical background of two different N-body simulation techniques for astrophysical dynamics: The treecode algorithm and the self-consistent field code (SCF). An attempt has been made to implement both methods on CUDA (Compute Unified Device Architecture), a special GPGPU (General Purpose Computation on Graphics Processing Unit) architecture which allows the programmer to directly and transparently implement algorithms on NVIDIA graphics cards. I present a complete implementation of the self-consistent field code on the CUDA architecture as well as a partial implementation of the treecode on the CPU.

Chapter 1

Introduction

Cosmological N-body simulations are among the most interesting and demanding topics in astrophysics: In order to understand the structure formation of galaxies and galaxy clusters (including their dark-matter halo), the dynamics of a large number of gravitationally interacting particles has to be understood at once. This large collection of particles is called a N-body system. The analytical tools that are available to analyze such systems are limited and the use of computer simulations becomes inevitable. Usually, the goal of such algorithms is not to simulate each and every particle of the system but to perform a *Monte-Carlo* simulation where just a subset of the true number of particles is simulated in order to achieve a desired accuracy. For example, a useful simulation of the Milky Way will not require the simulation of all the 10^{11} stars in the system but only a reasonable subset of them to resolve relevant features like its spiral arms.
Such a N-body simulation consists of two major steps: A force calculation and an integration step. In the force step, the force acting on each particle is calculated up to a certain required precision. Then, in the integration step, these forces are used to calculate the movement of the particle. The present thesis mainly deals with the implementation and optimization of the force calculation, whereas for the integration step the well-known standard Leapfrog integrator has been used.
The most naive way to calculate the forces acting on the individual particles in a N-body system is the *direct summation*: We simply loop over all particles and calculate the interaction with each other particle by using Netwons law of gravitation. Clearly this method scales as $\mathcal{O}(N^2)$ which results in an inacceptably long computation time for interesting systems (i.e. systems with a large number of particles). In the past decades, a number of approximative force calculation techniques have been developed in order to reduce the complexity. Two of these methods will be presented in this thesis, the *treecode* algorithm which scales with $\mathcal{O}(N \log N)$ and the *self-consistent field code* method (SCF), scaling with $\mathcal{O}(N)$. The treecode method is rather general and can be applied to a large collection of interesting problems. The SCF code implemented in the context of this thesis is much more adapted to a specific problem set and quickly looses its accuracy if the initial conditions are not chosen carefully. But note that modifications are available which enable this algorithm to perform with good accuracy on a larger set of problems (see [3], [4], [5]) while still scaling

with $\mathcal{O}(N)$.
Traditionally, the N-body simulations are carried out on special-purpose hardware like for example the CRAY-2 system to achieve maximal computational perfomance (i.e. an optimal rate of simulated particles per time unit). In the past few years, GPGPU (General-purpose computing on graphics processing units) architectures like NVIDIA's CUDA became competitive to specialised systems and are now heavily used for gravitational N-body simulations [9]. The implementation of such algorithms on the GPU tends to be more complicated than on the CPU, hence a whole chapter is dedicated to the CUDA implementation of the SCF code.

Chapter 2

Treecodes

2.1 Overview

In a treecode, the particles are hierarchically grouped in a tree-structure (for example a binary tree or an octree). The generation of such a tree in the context of N-body simulations is called the *treebuild*. Both, octress as well as binary trees are actually used in treecode-algorithms ([6], [7]). In what follows, we assume the structure to be a binary-tree, because its illustration is simpler.
If we now want to compute the acceleration acting on a particular particle at $\vec{p}_a$, we can use the tree structure to distinguish between two regimes of interactions, a set of particles S_{near} which are sufficiently near $\vec{p}_a$ and a set of particles S_{far} which lie sufficiently far away from $\vec{p}_a$. The interactions between $\vec{p}_a$ and S_{near} are evaluated by direct summation, whereas for the interaction between $\vec{p}_a$ and S_{far} we approximate the interaction by a multipole expansion, since this is a good approximation for the interaction between particles with a large distance between each other. In order to determine which of the particles are elements of S_{far} and which of them elements of S_{near}, we need to traverse the tree-structure. In the N-body setting, this traversal is called the *treewalk*.

2.2 Treebuild

In the treebuild, we recursively dissect the simulation domain including the particles into cuboids (in 3D) or rectangles (in 2D). In what follows, we assume the simulation to be 2-dimensional.
The pseudocode for the initialization of the treebuild and the recursive algorithm to generate the complete tree are given by algorithm 1 and algorithm 2. Note

Algorithm 1 Treebuild_Init

```
rootcell.x ← 0.5
rootcell.y ← 0.5
rootcell.length ← 1.0
rootcell.width ← 1.0
tree ← rootcell
Treebuild(root, x)
```

Algorithm 2 Treebuild

```
Input: root: the root cell of the current recursion level
Input: axis: the axis along which to partition within the current recursion level
Output: Builds the whole tree when called as in Algorithm 1

  Cell leftchild
  Cell rightchild
  Particlelist leftparticles
  Particlelist rightparticles
  if axis=x then
    $(leftparticles, rightparticles)$ ← partition(axis, root.x, root.particles)
    leftchild.x ← root.x - $\frac{root.width}{4}$
    leftchild.y ← root.y
    leftchild.width ← $\frac{root.width}{2}$
    leftchild.height ← root.height
    leftchild.particles ← leftparticles
    rightchild.x ← root.x + $\frac{root.width}{4}$
    rightchild.y ← root.y
    rightchild.width ← $\frac{root.width}{2}$
    rightchild.height ← root.height
    rightchild.particles ← rightparticles
  end if
  if axis=y then
    Similar to the above code for axis=x, just the roles of x and y are inverted
  end if
  if axis=x then
    axis ← y
  else
    axis ← x
  end if
  if leftcell.particles > BUCKETSIZE then
    Treebuild(leftchild, axis)
  end if
  if rightcell.particles > BUCKETSIZE then
    Treebuild(rightchild, axis)
  end if
  return
```

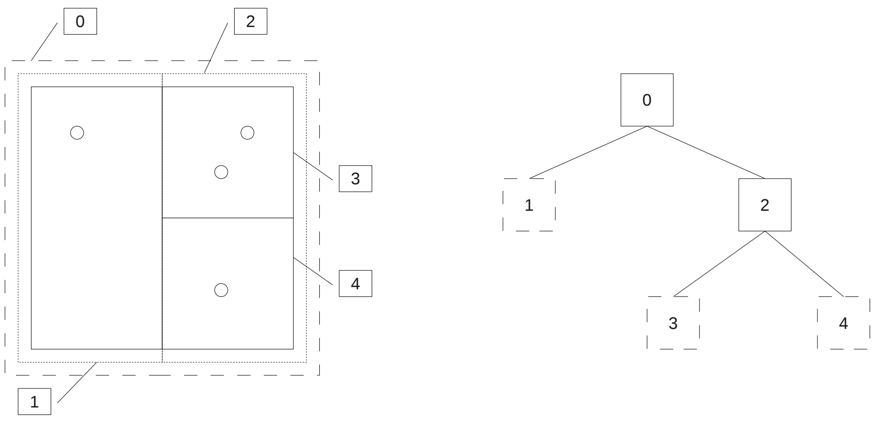

Figure 2.1: A binary tree resulting from 4 particles and a bucketsize of 2: On the left, we have a sketch of the spatial distribution of the particles and the cells (The circles symbolize the particles). On the right, the hierarchical tree structure is depicted. The numbers show the correspondence between the cells in space and their position in the tree hierarchy. Dashed cells to the right indicate bucket cells.

that the recursion is aborted as soon as a cell contains `BUCKETSIZE` or less particles.
Figure 2.2 shows a simple binary tree resulting from a distribution of 4 particles and a bucketsize of 2.
The core of algorithm 2 is a sub-algorithm which splits the particles within one cell along the dissection axis (in 2D) or the dissection plane (3D). This function performs a partitioning of the particles without using any intermediate data structures, i.e. it acts directly on the array which stores the particles. Since we are dealing with a potentially large number of particles, it is necessary to do this task as fast as possible. One good choice to do this is the algorithm `partition` which is a subprocedure of the quicksort algorithm and performs with $\mathcal{O}(N)$ and low overhead [16].
Figure 2.1 shows a binary tree generated by my partial implementation of the treecode for 200 particles and a bucket size of 5.

2.3 Treewalk

After having generated the tree, we can evaluate the forces acting on each particle by traversing the tree. In this process, we descend the tree and collect interactions until we reach a bucket cell where we let the collected interactions act onto the particles which are contained in this bucket.
In this way, the cost to compute the acceleration acting on one particle is reduced to $\mathcal{O}(\log N)$ [7] and the whole force calculation is now of $\mathcal{O}(N \log N)$.
There are different ways to actually perform the traversal of the tree, one of them is suggested by [6]:
We recursively walk down the tree and each recursive call of algorithm 3 corresponds to the visit of one cell in the tree. To keep track of the already determined

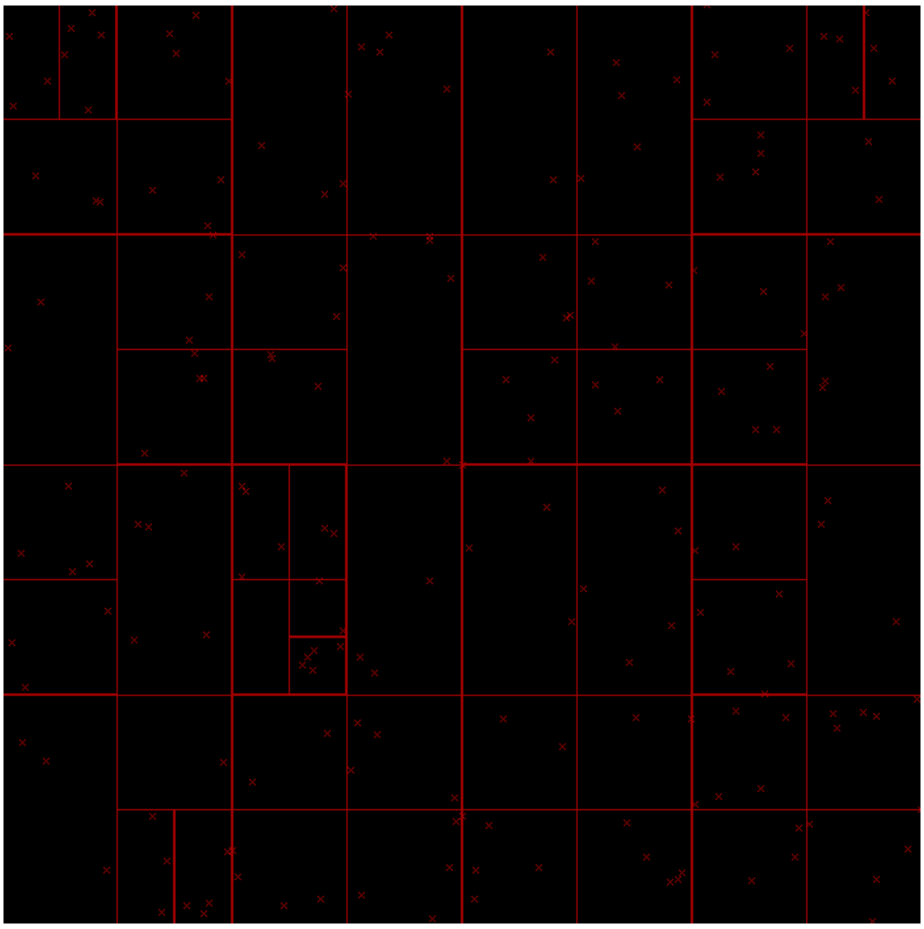

Figure 2.2: A binary tree generated by the treebuild algorithm (Algorithm 2) with 200 particles and a bucket size of 5

interactions from previous calls of the recursion, three lists are passed through the subsequent recursive calls of algorithm 3:

1. a *checklist* which stores the cells for which it could not have been determined whether they interact directly or via multipole expansion with the particles of the current cell (i.e. the cell we are currently visiting in the tree).
2. a *particlelist* which contains the cells that are directly interacting with the particles of the current cell.
3. a *momentlist* which contains the cells that are interacting via multipole expansion with the particles of the current cell.

In each recursion call, we check for each cell in the checklist if we can shuffle it to the particle- or momentlist until we reach a leaf of the recursion (i.e. a bucket cell). In this way, we perform an preorder traversal of the tree.
Note that if a cell C interacts via multipole with a certain other cell c, also the whole subtree rooted at C will interact with c by multipoles since all the cells of the subtree are subsets of C. A similar statement holds for the direct interaction. This means that cells that are put onto the particlelist or the momentlist during the treewalk can be savely removed from the checklist since the same interaction also holds for the children which are called in later recursive calls.
Also note that (as already stated) the lists are passed recursively: If the recursion returns to a particular cell after having processed the subtree rooted at this cell, the lists within this cell look the same as before processing the subtree.
The recursion halts when a bucket cell is reached. At this point, the algorithm loops over all particles in the bucket cell and lets the accumulated interactions act on the particles within the bucket cell. The complete algorithm for walking the tree is then given by algorithm 3 [6]:
Additionally to the above mentioned lists, we introduced a temporary list of cells, the *staylist.* The undecided cells are moved onto this list such that each cell c is eventually deleted from the checklist and the while loop is guaranteed to terminate. After the while loop, we shuffle the undecided cells back from the staylist onto the checklist.
In this algorithm, 12 different cases can occur which lead to a total of 4 different outcomes as illustrated in figure 2.3. The decision for a certain outcome is made depending on the *opening radius* of the cell c which is defined by

$$r_{open} = \frac{2}{\sqrt{3}} r_{max} \theta \tag{2.1}$$

where θ is a control parameter and r_{max} is the distance from the center-of-mass of the cell to the most distant corner of the cell [6]. This quantity controls how strongly the algorithm tends to multipole expand the interactions: If $r_{open} = 0$, c will multipole-interact with each other cell, if $r_{open} = \infty$, c will directly interact with all other cells.
One can satisfy himself that the outcome "undecided" is resolved to either "direct" or "multipole" by walking deeper down the tree and the outcome "open" is resolved to "direct" or "multipole" by further processing the while loop in algorithm 3 . Consequently, this algorithm is guaranteed to terminate, since at the latest when the recursion reaches a bucket, the checklist will be emptied completely.

Algorithm 3 Treewalk

Input: Cell: The cell which is currently visited in the tree
Input: checklist: see above
Input: particlelist, momentlist: See above
Output: assigns all interactions to the particles in the buckets of the subtree rooted at Cell
Output: to compute *all* interactions, the algorithm must be called as Treewalk(root, checklist=$\emptyset$, particlelist=$\emptyset$, momentlist=$\emptyset$)

```
while c ∈ checklist do
  if Cell ∈ OpenBall(c) ∨ (isBucket(Cell) ∧ Cell ∩ OpenBall(c) ≠ ∅) then
    checklist ← checklist - c
    if isBucket(c) then
      case 1: direct interaction
      particlelist ← particlelist + c
    else
      case 2: opening
      checklist ← checklist + children(c)
    end if
  else if Cell ∩ OpenBall(c) = ∅ then
    case 3: multipole interaction
    checklist ← checklist - c
    momentlist ← momentlist + c
  else
    case 4: undecided
    checklist ← checklist - c
    staylist ← staylist + c
  end if
end while

for all c ∈ staylist do
  staylist ← staylist - c
  checklist ← checklist +c
end for

if isBucket(Cell) then
  for all c ∈ Cell do
    calculateGravity(c, particlelist, momentlist)
  end for
else
  leftchecklist ← checklist + Cell.rightchild
  rightchecklist ← checklist + Cell.leftchild
  Treewalk(Cell.leftchild, leftchecklist, particlelist, momentlist)
  Treewalk(Cell.rightchild, rightchecklist, particlelist, momentlist)
end if
```

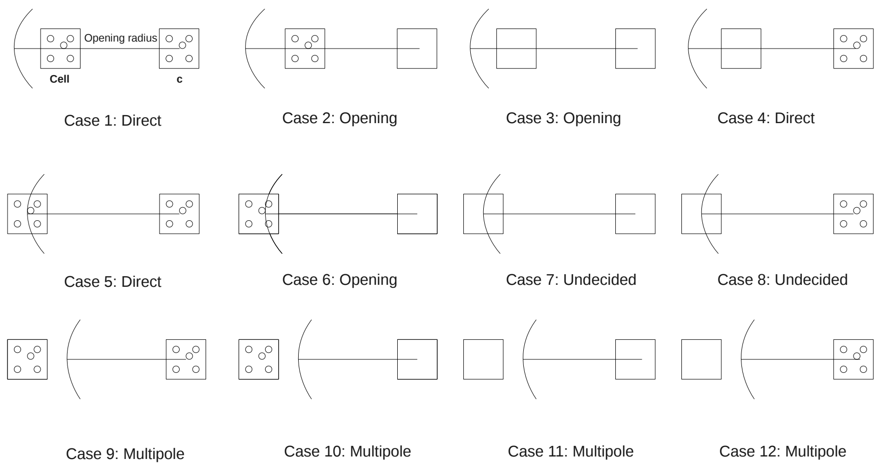

Figure 2.3: Different outcomes for one iteration of the while loop in algorithm 3. The left cell corresponds to *Cell* and the right one to c from algorithm 3. The line and the arc denote the opening radius of c. A bucket is symbolized by small circles within the square.

2.4 Multipole expansion

In its simplest form, the potential acting on a particle at $\vec{p}_a$ caused by a set of other particles at $\vec{p}_n, n = 0, ..., N-1$ with masses $m_n, n = 0, ..., N-1$ at sufficiently large distance can be approximated by a first order multipole expansion:

$$\Phi(\vec{r}) = \frac{\sum_{i=0}^{N-1} m_i}{||\vec{r}||} \tag{2.2}$$

This is a good point to start testing the program when writing the code.
Although higher order multipole expansions in spherical coordinates are more elegant (and simple), they are computationally inefficient for higher orders of the expansion [6]. Therefore, the expansion should be carried out in Cartesian coordinates.
More sophisticated expansion techniques for calculating higher order Cartesian terms around arbitrary expansion points can for example be found in [6]

2.5 Parallel implementation

Although the recursive algorithms described above will work on traditional CPU's, the algorithm should nevertheless be transformed into a non-recursive form because recursive calls can significantly decrease performance due to function-calling overhead [6]. For a CUDA implementation, the recursive algorithm would even be out of question since CUDA kernels [1] do not support recursive kernel calls.

[1]functions written in CUDA are called *kernels*

When calling functions in a recursive fashion, an internal *callstack* managed by the operating system, is used for each call which contains the variables having been passed as function arguments during *all* the previous recursive calls. But this stack can just as well be generated and managed manually.
The manual *callstack* is a list containing an arbitrary number of *callstack entries*. These entries are simply 4-tuples of the lists we would pass to the Treewalk function when using the recursive algorithm 3: $(cell, checklist, particlelist, momentlist)$. In addition, the callstack has two procedures `pop()` and `push()` which allow to add a new entry to the stack (`push()`) and to store and then delete the last entry which has been pushed to the stack (`pop()`).
With these modifications, we can rewrite the algorithm in a non-recursive fashion as shown in algorithm 4.

Algorithm 4 Non-recursive Treewalk

```
Input: rootcell: the root from algorithm 1
Input: callstack: the callstack used to emulate the recursive calls
Output: perform the complete treewalk without recursions

  callstack.push(rootcell, $\emptyset$, $\emptyset$, $\emptyset$)
  while s $\in$ callstack do
    stackentry $\leftarrow$ callstack.pop()
    Cell $\leftarrow$ stackentry.cell
    checklist $\leftarrow$ stackentry.checklist
    particlelist $\leftarrow$ stackentry.particlelist
    momentlist $\leftarrow$ stackentry.momentlist
    while c $\in$ checklist do
      identical to the corresponding code in algorithm 3
    end while

    for all c $\in$ staylist do
      identical to the corresponding code in algorithm 3
    end for

    if isBucket(Cell) then
      identical to the corresponding code in algorithm 3
    else
      leftchild $\leftarrow$ Cell.leftchild
      rightchild $\leftarrow$ Cell.rightchild
      leftchecklist $\leftarrow$ checklist + Cell.rightchild
      rightchecklist $\leftarrow$ checklist + Cell.leftchild
      callstack.push(rightchild, rightchecklist, particlelist, momentlist)
      callstack.push(leftchild, leftchecklist, particlelist, momentlist)
    end if
  end while
```

Chapter 3

Self-consistent field codes

3.1 Solution of Poisson equation by biorthogonal basis expansions

The self-consistent field code goes a different way than the treecode: Instead of thinking in terms of interactions between individual particles, we start with the general Poisson equation for gravity:

$$\Delta\Phi(\vec{r}) = 4\pi G\rho(\vec{r}) \tag{3.1}$$

where Φ is the potential of the gravitational force field and ρ is a (possibly continuous) mass-density distribution (which will later be specialised to the discrete distribution of N individual particles).
We try the following series ansatz [1] for the solution of equation (3.1):

$$\rho(\vec{r}) = \sum_{nlm} A_{nlm}\rho_{nlm}(\vec{r}) \tag{3.2}$$

$$\Phi(\vec{r}) = \sum_{nlm} A_{nlm}\Phi_{nlm}(\vec{r}) \tag{3.3}$$

where the indices n, l, m denote the "quantum numbers" since they have a similar meaning as the actual quantum numbers in the study of quantum mechanical systems (i.e. the index n is the "radial quantum number", indexing the moments of the radial part of the solution, whereas l and m label the azimuthal and polar angle respectively). the A_{nlm} are the coefficients of the expansion and ρ_{nlm}, Φ_{nlm} are the expansion functions to be determined.
In order to proceed, we have to assume that we are dealing with an analytical mass density distribution and that our N-body simulation is written to either solve exactly this model or a closely matching one. We call the analytical model the *underlying* model and denote its density and potential by $\rho_{underlying}$ and $\Phi_{underlying}$ respectively. We now impose the conditions which the expansions (3.2) and (3.3) have to satisfy (we use dimensionless units $G = 1$):

1. The pairs with same index triples (Φ_{nlm}, ρ_{nlm}) in the two expansions must solve the Poisson equation individually:

$$\Delta\Phi_{nlm} = 4\pi\rho_{nlm}(\vec{r}) \qquad \forall n, l, m \tag{3.4}$$

By linearity of the Poisson equation, this condition ensures, that the expansions yield a solution of (3.1)

2. The first terms of the expansions ($n = 0, l = 0, m = 0$) must match the underlying model:
$$\rho_{000} \propto \rho_{underlying} \tag{3.5}$$
$$\Phi_{000} \propto \Phi_{underlying} \tag{3.6}$$
If our simulated particles closely match the underlying model, the resulting expansion will rapidly converge and we are allowed to truncate at low order.

3. The pairs of same index triples in the two expansions must be biorthogonal:
$$\int \rho_{nlm}(\vec{r})[\Phi_{n'l'm'}(\vec{r})]^* \,\mathrm{d}\vec{r} \propto \delta_{nlm}^{n'l'm'} \tag{3.7}$$
where $\delta_{nlm}^{n'l'm'} = \delta_n^{n'}\delta_l^{l'}\delta_m^{m'}$. This condition is sufficient to determine the expansion coefficients A_{nlm}, since
$$\begin{aligned}\int \rho(\vec{r})[\Phi_{n'l'm'}(\vec{r})]^* \,\mathrm{d}\vec{r} &= \int \sum_{nlm} A_{nlm}\rho_{nlm}(\vec{r})[\Phi_{n'l'm'}(\vec{r})]^* \,\mathrm{d}\vec{r} \\ &= \sum_{nlm} A_{nlm} \int \rho_{nlm}(\vec{r})[\Phi_{n'l'm'}(\vec{r})]^* \,\mathrm{d}\vec{r} \\ &\equiv \sum_{nlm} A_{nlm} I_{nlm}^{n'l'm'} \delta_{nlm}^{n'l'm'} \\ &= A_{n'l'm'} I_{n'l'm'}^{n'l'm'}\end{aligned} \tag{3.8}$$
If we relabel (n', l', m') to (n, l, m) we obtain the expression for the expansion coefficients
$$A_{nlm} = \frac{1}{I_{nlm}^{nlm}} \int \rho(\vec{r})[\Phi_{nlm}(\vec{r})]^* \,\mathrm{d}\vec{r} \tag{3.9}$$
where
$$I_{nlm}^{nlm} = \int \rho_{nlm}(\vec{r})[\Phi_{nlm}(\vec{r})]^* \,\mathrm{d}\vec{r} \tag{3.10}$$
Note that $\rho(\vec{r})$ in (3.9) is given by the *known* discrete particle distribution of the N particles in our simulation:
$$\rho(r, \theta, \phi) = \sum_{k=0}^{N-1} \frac{m_k}{r_k^2} \delta(r - r_k)\delta(\phi - \phi_k)\delta(\theta - \theta_k) \tag{3.11}$$
Also note that we would not have been able to obtain (3.9) without the demand for the biorthogonality of the pairs (Φ_{nlm}, ρ_{nlm}).

3.2 Constructing the Hernquist basis

The program written for this thesis uses the Hernquist distribution as the underlying model:

$$\rho_{underlying}(\vec{r}) = \rho_{000}(\vec{r}) = \frac{1}{2\pi}\frac{1}{r}\frac{1}{(1+r)^3} \tag{3.12}$$

and by virtue of the Poisson equation:

$$\Phi_{underlying}(\vec{r}) = \Phi_{000}(\vec{r}) = -\frac{1}{1+r} \tag{3.13}$$

[1] use the following ansatz for the higher moments of the expansion:

$$\rho_{nlm}(\vec{r}) = \frac{K_{nl}}{2\pi}\frac{r^l}{r(1+r)^{2l+3}}W_{nl}(r)\sqrt{4\pi}Y_{lm}(\theta,\phi) \tag{3.14}$$

$$\Phi_{nlm}(\vec{r}) = -\frac{r^l}{(1+r)^{2l+1}}W_{nl}(r)\sqrt{4\pi}Y_{lm}(\theta,\phi) \tag{3.15}$$

Instead of using the original variable $r \in [0,\infty)$ we switch to the "renormalized variable" that maps the interval $[0,\infty)$ onto $[-1,1)$ to circumvent evaluations of improper integrals:

$$\xi(r) = \frac{r-1}{r+1} \tag{3.16}$$

$$r(\xi) = \frac{1+\xi}{1-\xi} \tag{3.17}$$

$$\mathrm{d}r = \frac{2}{(1-\xi)^2}\mathrm{d}\xi \tag{3.18}$$

$$1+r(\xi) = \frac{2}{1-\xi} \tag{3.19}$$

It can be shown [1] that $W_{nl}(r)$ satisfies the Gegenbauer differential equation (written in Sturm-Liouville Form):

$$\frac{\mathrm{d}}{\mathrm{d}\xi}\left[(1-\xi^2)^{2l+2}\frac{\mathrm{d}W_{nl}}{\mathrm{d}\xi}\right] = n(n+4l+3)\underbrace{(1-\xi^2)^{2l+1}}_{w(\xi)}W_{nl} \tag{3.20}$$

As known from Sturm-Liouville theory, the weighting function can be read off from (3.20) as $w(\xi) = (1-\xi^2)^{2l+1}$ and the solutions of (3.20) $W_{nl}(\xi) \equiv C_n^{(2l+\frac{3}{2})}(\xi)$ (the Gegenbauer polynomials) are thus orthogonal in the following sense [19]:

$$\int_{-1}^{1} C_n^{(2l+1)}\,(1-\xi^2)^{2l+1}\,C_{n'}^{(2l+1)}\,\mathrm{d}\xi \propto \delta_{nn'} \tag{3.21}$$

All the three conditions for a good basis from the last section, (3.4), (3.5) and (3.7), are in fact satisfied by the Hernquist ansatz (3.14) and (3.15). We show as an example that with the help of (3.21) we can explicitly confirm the biorthogonality condition:

$$\begin{aligned}
\int \rho(\vec{r})[\Phi_{n'l'm'}(\vec{r})]^* \, \mathrm{d}\vec{r} = & -\frac{K_{nl}}{2\pi} 4\pi \\
& \times \underbrace{\int_{-\pi}^{\pi} \int_{0}^{2\pi} Y_{lm}(\theta,\phi) Y_{l'm'}(\theta,\phi) \sin(\theta) \, \mathrm{d}\theta \mathrm{d}\phi}_{=\delta_{ll'}\delta_{mm'}} \\
& \times \int_{0}^{\infty} \frac{r^l}{r(1+r)^{2l+3}} \frac{r^l}{(1+r)^{2l'+1}} C_n^{(2l+\frac{3}{2})} C_{n'}^{(2l'+\frac{3}{2})} r^2 \, \mathrm{d}r \\
= & -\frac{K_{nl}}{2\pi} 4\pi \delta_{ll'}\delta_{mm'} \times \\
& \int_{-1}^{1} \frac{(1+\xi)^{l-1}}{(1-\xi)^{l-1}} \frac{(1-\xi)^{2l+3}}{2^{2l+3}} \frac{(1+\xi)^l}{(1-\xi)^l} \frac{(1-\xi)^{2l+1}}{2^{2l+1}} \frac{(1+\xi)^2}{(1-\xi)^2} C_n^{(2l+\frac{3}{2})} C_{n'}^{(2l+\frac{3}{2})} \, \mathrm{d}\xi \\
= & -\frac{K_{nl}}{2\pi} 4\pi \delta_{ll'}\delta_{mm'} \int_{-1}^{1} (1-\xi)^{2l+1}(1+\xi)^{2l+1} \frac{1}{2^{4l+4}} C_n^{(2l+\frac{3}{2})} C_{n'}^{(2l+\frac{3}{2})} \, \mathrm{d}\xi \\
= & -\frac{K_{nl}}{2\pi} 4\pi \delta_{ll'}\delta_{mm'} \frac{1}{2^{4l+4}} \underbrace{\int_{-1}^{1} (1-\xi^2)^{2l+1} C_n^{(2l+\frac{3}{2})} C_{n'}^{(2l+\frac{3}{2})} \, \mathrm{d}\xi}_{\propto \delta_{nn'}} \\
\propto \quad & \delta_{nlm}^{n'l'm'}
\end{aligned} \tag{3.22}$$

3.3 Evaluation of the accelerations

We are still left with the problem of determining the coefficients (3.9)

$$A_{nlm} = \frac{1}{I_{nlm}^{nlm}} \int \rho(\vec{r})[\Phi_{nlm}(\vec{r})]^* \, \mathrm{d}\vec{r} \tag{3.9}$$

and eventually of computing the acceleration (a_r, a_θ, a_ϕ) at a given point (r, θ, ϕ) Since the functions $\rho(\vec{r})$ and $\Phi(\vec{r})$ must be real, $\rho_{nlm}(\vec{r})$ (3.14), $\Phi_{nlm}(\vec{r})$ (3.15) and A_{nlm} (3.9) can also be expected to be real. Consequently, we should try to elliminate all the imaginary parts in the spherical harmonics $Y_{lm}(\theta, \phi)$, since these are the only complex valued functions that enter the equations. The results are expansions containing only real quantities [1]:

$$\rho(r,\theta,\phi) = \sum_{l=0}^{\infty} \sum_{m=0}^{l} P_{lm}(\cos\theta) \left[A_{lm}(r)\cos(m\phi) + B_{lm}(r)\sin(m\phi)\right] \tag{3.23}$$

and

$$\Phi(r,\theta,\phi) = \sum_{l=0}^{\infty} \sum_{m=0}^{l} P_{lm}(\cos\theta) \left[C_{lm}(r)\cos(m\phi) + D_{lm}(r)\sin(m\phi)\right] \tag{3.24}$$

with $A_{lm}, B_{lm}, C_{lm}, D_{lm} \in \mathbb{R}$
For cleaner notation, we introduce the real-valued version of (3.15), $\tilde{\Phi}_{nl}$:

$$\tilde{\Phi}_{nl} \equiv \frac{\Phi_{nlm}}{Y_{lm}(\theta,\phi)} \tag{3.25}$$

When computing $\vec{a} = -\nabla\Phi = -\left[\frac{\partial}{\partial r} + \frac{1}{r}\frac{\partial}{\partial\theta} + \frac{1}{r\sin\theta}\frac{\partial}{\partial\phi}\right]\Phi$ we obtain the accelerations of the gravitational field:

$$a_r(r,\theta,\phi) = -\sum_{l=0}^{\infty}\sum_{m=0}^{l} P_{lm}(\cos\theta)\left[E_{lm}(r)\cos(m\phi) + F_{lm}(r)\sin(m\phi)\right] \tag{3.26}$$

$$a_\theta(r,\theta,\phi) = -\frac{1}{r}\sum_{l=0}^{\infty}\sum_{m=0}^{l} \frac{\mathrm{d}P_{lm}(\cos\theta)}{\mathrm{d}\theta}\left[C_{lm}(r)\cos(m\phi) + D_{lm}(r)\sin(m\phi)\right] \tag{3.27}$$

$$a_\phi(r,\theta,\phi) = -\frac{1}{r}\sum_{l=0}^{\infty}\sum_{m=0}^{l} m\frac{P_{lm}(\cos\theta)}{\sin\theta}\left[D_{lm}(r)\cos(m\phi) - C_{lm}(r)\sin(m\phi)\right] \tag{3.28}$$

where (k is the index over the particles)

$$\begin{aligned}\begin{pmatrix} C_{lm}(r) \\ D_{lm}(r) \\ E_{lm}(r) \\ F_{lm}(r)\end{pmatrix} &= N_{lm}\sum_{n=0}^{\infty}\frac{1}{I_{nlm}^{nlm}}\begin{pmatrix}\tilde{\Phi}_{nl}(r) \\ \tilde{\Phi}_{nl}(r) \\ \frac{\mathrm{d}}{\mathrm{d}r}\tilde{\Phi}_{nl}(r) \\ \frac{\mathrm{d}}{\mathrm{d}r}\tilde{\Phi}_{nl}(r)\end{pmatrix}\sum_{k=0}^{N-1} m_k\tilde{\Phi}_{nl}(r_k)P_{lm}(\cos\theta_k)\begin{pmatrix}\cos m\phi_k \\ \sin m\phi_k \\ \cos m\phi_k \\ \sin m\phi_k\end{pmatrix} \\ &\equiv N_{lm}\sum_{n=0}^{\infty}\frac{1}{I_{nlm}^{nlm}}\begin{pmatrix}\tilde{\Phi}_{nl}(r) \\ \tilde{\Phi}_{nl}(r) \\ \frac{\mathrm{d}}{\mathrm{d}r}\tilde{\Phi}_{nl}(r) \\ \frac{\mathrm{d}}{\mathrm{d}r}\tilde{\Phi}_{nl}(r)\end{pmatrix}\begin{pmatrix}\mathbf{X}_{nlm} \\ \mathbf{X}_{nlm}\end{pmatrix}\end{aligned} \tag{3.29}$$

In practice the sums over n and l are truncated at low order (for example $l_{max} = 3$ and $n_{max} = 3$) which means that the only costly (i.e. large) sums are:

$$\mathbf{X}_{nlm} = \sum_{k=0}^{N-1} m_k\tilde{\Phi}_{nl}(r_k)P_{lm}(\cos\theta_k)\begin{pmatrix}\cos m\phi_k \\ \sin m\phi_k\end{pmatrix} \tag{3.30}$$

These sums take $\mathcal{O}(N)$ to calculate. They must only be tabulated once per integration step and we call the resulting table the **X**-table. With these tabulated values, a single evaluation of (3.26) - (3.28) is of $\mathcal{O}(1)$ and consequently the evaluation of the accelerations on all N particles is of $\mathcal{O}(N)$.

3.4 Unrolling recursion relations

A closer look at the formulas (3.26) - (3.28) and the program code will reveal that there are a lot of evaluations of the same generic form

$$F = F_l^m(x) \tag{3.31}$$

These functions are namely $P_l^m(cos\theta)$ and $\frac{\mathrm{d}P_l^m(\cos\theta)}{\mathrm{d}\theta}$ from (3.26) - (3.28) and several power- and factorial functions in the code. The only stable way to evaluate these functions is by recursion formulae [18]. Normally, the functions (3.31) are computed for l, m up to a certain order. Assuming that we implemented a function $F(x, l, m)$ to compute the values from (3.31), the naive way to perform the computations would be algorithm 5. Note that, unless the values are tabulated inside $F(x, l, m)$, each call will start the recursion relation again, starting from $l = 0$, $m = 0$ although these values have already been calculated in previous iterations of the l and m loops in algorithm 5! To avoid these redundant evaluations, we rather use an “unrolled” recursion relation, i.e. we perfom the recursions directly within the l and m loops of algorithm 5. For example, for the associated Legendre polynomials P_l^m we have a stable explicit expression for P_m^m and P_{m+1}^m and a two-point recursion formula of the form [18]:

Algorithm 5 Naive usage of a recursively defined formula

```
for $l = 0$ to $l_{max}$ do
  for $m = 0$ to $l$ do
    $F_l^m = F(x, l, m)$
    use $F_l^m$ for computations here
  end for
end for
```

$$P_{l+2}^m = P_{l\to l+2}^m \left[P_l^m, P_{l+1}^m\right] \tag{3.32}$$

which allows us to unroll the recursions as shown in algorithm 6. [1]

Algorithm 6 Unrolled recursive formula for the associated Legendre polynomials

```
for $m = 0$ to $l_{max}$ do
  $P_m^m$ (by explicit formula)
  $P_{m+1}^m$ (by explicit formula)
  $P_l^m \leftarrow P_m^m$
  $P_{l+1}^m \leftarrow P_{m+1}^m$
  for $l = m$ to $l_{max}$ do
    use $P_l^m$ for computations at this point
    temp $\leftarrow P_{l+1}^m$
    $P_{l+1}^m \leftarrow P_{l\to l+2}^m \left[P_l^m, P_{l+1}^m\right]$
    $P_l^m \leftarrow$ temp
  end for
end for
```

[1]Note that also the order the l- and m-loops has been swapped

Chapter 4

CUDA implementation of the self-consistent field code

As already indirectly stated in chapter 3 the SCF algorithm is composed of two sub-algorithms which both scale as $\mathcal{O}(N)$:

1. The evaluation of the **X**-table (3.30)
2. The evaluation of the accelerations (3.26) - (3.28) for all the N particles.

The serial CPU code that I have written is a lengthy but straightforward application of the knowledge from chapter 3.
On the other hand, to understand the GPU code it is necessary to learn some new concepts which will be explained in the following.
CUDA is a general purpose parallel computing architecture developed by the graphics card vendor NVIDIA to give programmers the ability to perform general numerical computations on NVIDIA graphics cards in a transparent and scalable fashion. The scalability is achieved by an API that works with an abstraction of the actual GPU- [1] and memory layout of the graphics card. These abstraction layers will be explained in the coming two sections.

4.1 CUDA GPU layout

A standard CPU contains around 2 to 8 cores that can perform computations in parallel. On the other hand, for example the NVIDIA GeForce GTX 590 'Fermi' GPU contains 16 parallel multiprocessors with 32 cores each, meaning that we have to distribute the work over 512 physically parallel processors.
The CUDA API offers an abstraction of the physical GPU, multiprocessors and cores called *grids*, *blocks* and *threads*. The correspondence is as follows:

1. whole GPU (containing all multiprocessors) $\cong$ grid
2. multiprocessor (containing multiple cores) $\cong$ block
3. core $\cong$ thread

[1]Whenever we refer to the GPU, we only mean the *multiprocessors* on the graphics card, without the memory

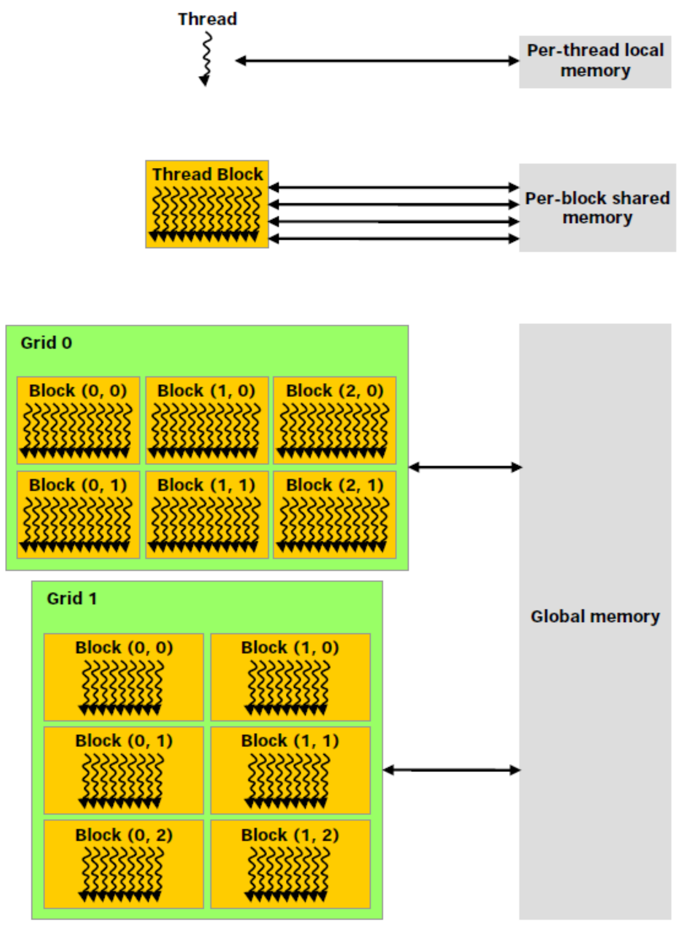

Figure 4.1: Illustration of the CUDA memory layout, from [14]

CUDA takes care of the correct mapping of the concepts to the right onto the ones to the left and the programmer should think exclusively in terms of grids, blocks and threads.

4.2 CUDA memory Layout

Each of the three abstraction concepts from the previous section has its own memory space. The correspondence is as follows (cf. figure 4.1):

1. grid $\leftrightarrow$ global memory
2. block $\leftrightarrow$ shared memory
3. thread $\leftrightarrow$ register memory (a.k.a. local memory)

The memory spaces are hierarchical, just as the GPU abstractions from the previous section, i.e. the shared memory within one block can only be accessed

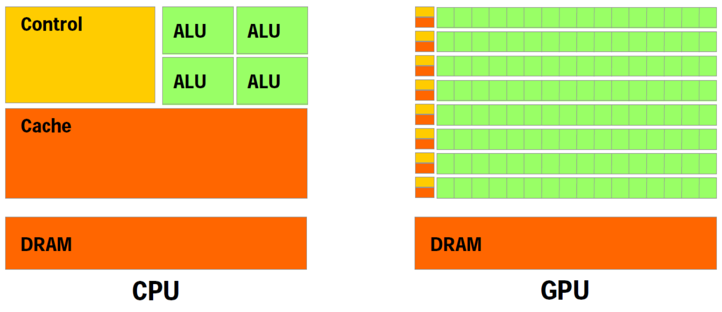

Figure 4.2: Amount of transistors and memory devoted to the different components of the CPU and the GPU , from [14]

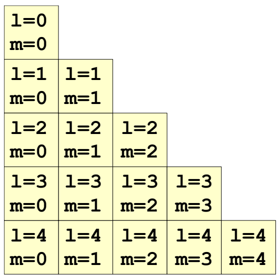

Figure 4.3: Different terms of the **X**-table illustrated as a triangle

by threads within this block and the register memory belonging to one thread can only be accessed by this thread.
Since memory accesses to the global memory of the GPU are very slow, it is crucial to write the program such that a minimum of global memory accesses occur, allowing the data to reside in the fast register or shared memory as long as possible.
On a CPU on the other hand, each core has plenty of cache memory (which corresponds to shared memory on the GPU), which mitigates the problem of global memory accesses (cf. figure 4.2). [2]
This means that the most important difference between GPU and CPU programming is the dealing with the scarcity of fast shared and register memory.

4.3 The X-table kernel

We consider equation (3.30) for a fixed n-index and write these values as $\mathbf{X}_{lm}$. For a fast evaluation, unrolling the recursions for P_{lm} and $\frac{\mathrm{d}}{\mathrm{d}\theta}P_{lm}(\cos\theta)$ is mandatory. This implies that each $\mathbf{X}_{lm}$ with l and m depends on all $\mathbf{X}_{l'm'}$ with lower indices, as explained in section 3.4. To get a clear depiction of this fact, we imagine the different values as arranged in a triangle (c.f. figure 4.3), where each of the squares with higher index depends on all the squares with

[2]ALU: Arithmetic logic units, the transistors which actually perform computations. DRAM: Dynamic random-access memory, the global memory of the processor.

lower index (in both l and m direction). Below, we will omit the indices and use a simple triangle whenever we want to emphasize that the values are not independent but computed by an unrolled recursion relation.

4.3.1 Step 1: Computing the terms

The first step is to compute the values in (3.30) for each index 5-tuple (n, l, m, x, k) where $x = 0$ stands for the cosine component and $x = 1$ for the sine component in (3.30) respectively. Figure 4.4 shows the kernel layout that was used to compute the terms. Each block is two dimensional with the n-index along the vertical and the k-index along the horizontal. This means that each thread computes one $\mathbf{X}_{lm}^{x=0}$ and one $\mathbf{X}_{lm}^{x=1}$ table for one particular (n, k) index. The computed triangles are held in shared memory to allow fast accesses when they are needed in subsection 4.3.2
Since the number of triangles that can be computed within one block has an upper bound given by the amount of shared memory available per block, this layout achieves maximal parallelism (i.e. a maximal number of threads per block). To see why, imagine for example that all the triangles for different n would be computed within *one* thread. One may think that we are then allowed to extend the number of threads in the block horizontally by a factor of $(n_{max} + 1)$ since we have reduced the number of threads along the vertical by the same amount. But since *all* the triangles have to reside in shared memory, such a layout would simply multiply the amount of shared memory occupied by one thread by a factor of $(n_{max} + 1)$ compared to the layout from Figure 4.4, which prohibits the extension of the block along the horizontal. Consequently, parallelism would be reduced by a factor $(n_{max} + 1)$.

4.3.2 Step 2: Compression

We now have to sum up the terms locally on each block. To distinguish this local (per block) summation from the global summation described in the next subsection, we refer to the algorithm described here as the *compression* of the terms.
The data in shared memory is arranged linearly and such that terms with consecutive k-indices are also consecutive in shared memory. More precisely, the shared memory is arranged according to the mapping:

$$i_{shared}(n, x, l, m) = D_x \left[n \cdot N_{lm} + x \cdot \frac{N_{lm}}{2} + \frac{l(l+1)}{2} + m \right] + i_{thread} \qquad (4.1)$$

where i_{shared} is the memory location in the shared memory, $N_{lm} = \frac{(l_{max}+1)(l_{max}+2)}{2}$ is the number of entries within one triangle, i_{thread} the index along the horizontal of the thread that is writing to the memory and D_x the horizontal block dimension. This means that the memory is divided into "chunks" of entries with same (n, x, l, m) and increasing k index which can be summed up individually. Figure 4.5 shows an example of parallel summation of one such "chunk" with 8 threads. The rectangles filled with random numbers represent fields within the shared memory and the numbers within the circles denote the index of the thread (along the horizontal of the grid) which is summing up two values respectively.

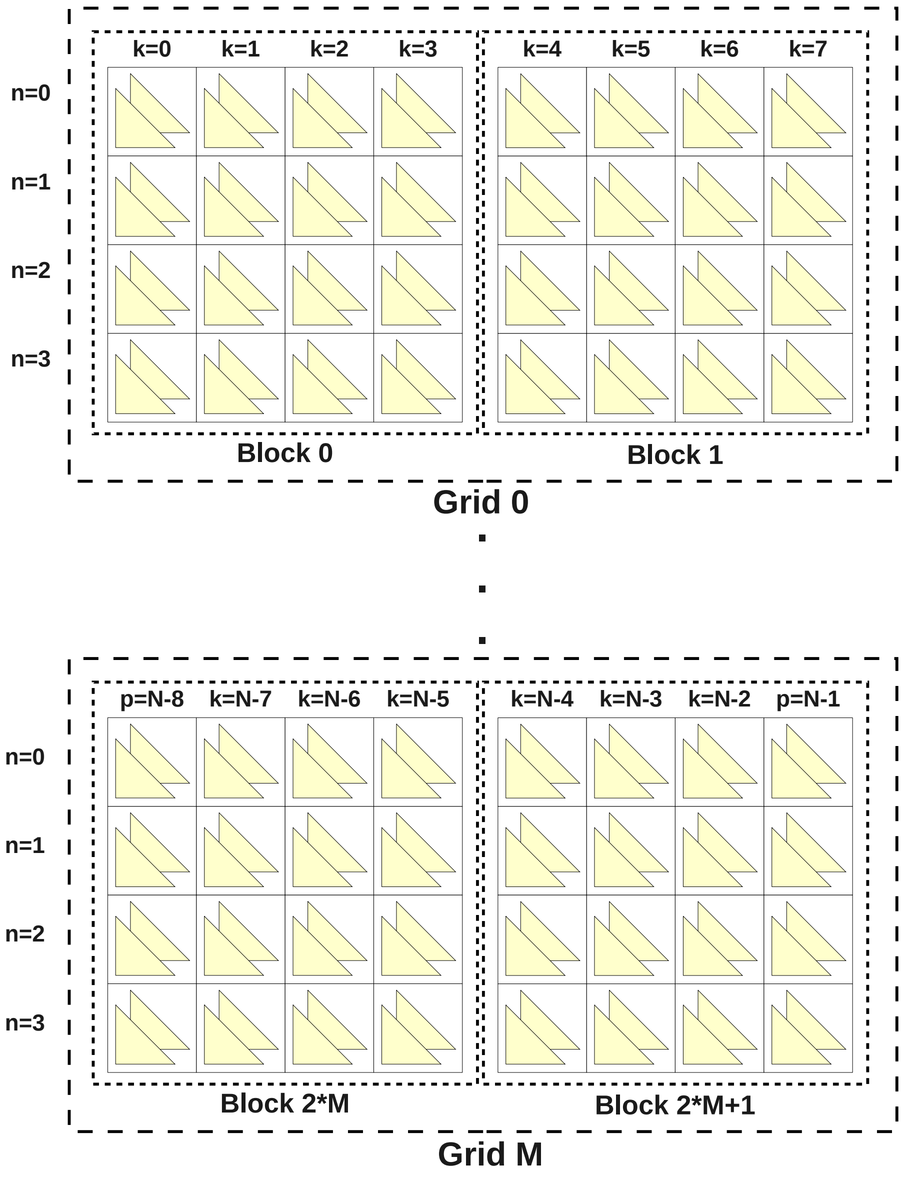

Figure 4.4: Kernel layout to compute the individual terms in (3.30) with 4 particles per block, 2 blocks per grid and $n_{max} = 3$. For visualization purposes the dimensionality of the blocks and grids is reduced compared to the actual kernel.

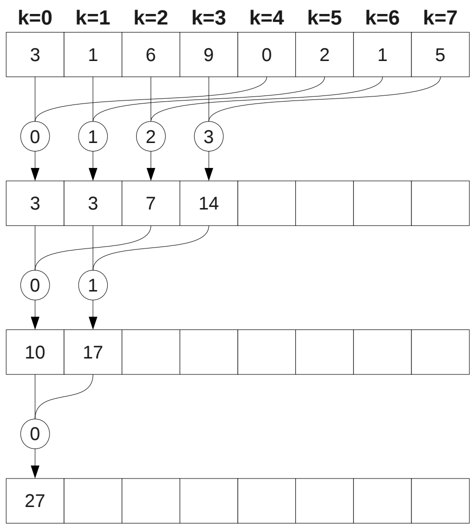

Figure 4.5: Parallel *compression* (summation) within one "chunk" of shared memory fields with identical indices (n, x, l, m). In this example, one block processes 8 particles.

After having performed all the compressions, the result for one block are $2 \cdot (n_{max} + 1)$ triangles (two for each n-index) which are then written to the global memory.

4.3.3 Step 3: Reduction

As we have seen, each block produces one "column" of triangles which is written to global memory. Since each block processes only 32 threads (due to the small amount of shared memory), there will be a lot of data written to the global memory which has to be further reduced in an efficient way. Fortunately, CUDA already comes with an algorithm called `reduce` to compute sums of large arrays. We can directly employ this algorithm to sum up the many columns of triangles to one single column containing the final $\mathbf{X}_{nlm}$ values from (3.30)

4.4 The acceleration kernel

The acceleration kernel computes the equations (3.26)- (3.28). The values of the $\mathbf{X}$-table are clearly constant during the whole runtime of this kernel. CUDA offers yet another memory space dedicated to the storage of such constant values, the *constant memory*, which allows very fast accesses.
Since we do not need to store any data in the shared memory but only need register memory to carry out the unrolled recursion relations, the dimensionality of one block is much less severely restricted than in the $\mathbf{X}$-table kernel. We could use a similar layout as in the $\mathbf{X}$-table kernel, but instead we also can simply

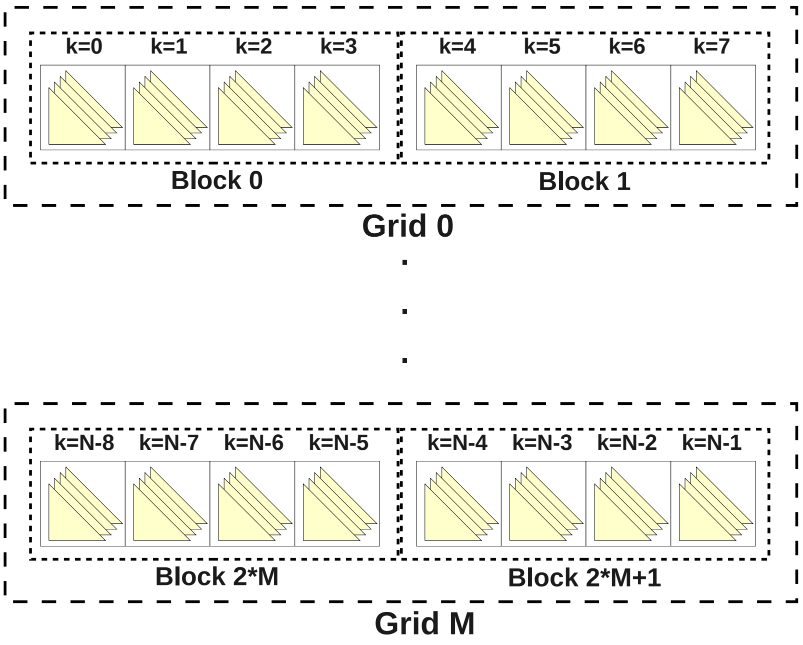

Figure 4.6: Layout of the acceleration kernel with 4 particles per block and 2 blocks per grid. For visualization purposes the dimensionality of the blocks and grids is reduced compared to the actual kernel.

perform the sums over n, l and m for fixed k within one single thread and extend the block dimensionality horizontally. The layout of the acceleration kernel is shown in Figure 4.6. The stacked triangles symbolize the sum (loop) over n in (3.29): Within each iteration of n, one sine and one cosine triangle is computed.

Chapter 5

Results

In this chapter, I present several results obtained from the SCF simulation outlined in the previous two chapters: A performance comparison between the GPU and the CPU implementation is presented, as well as the result from two simple simulation situations (a non-perturbed and a perturbed Hernquist galaxy).

5.1 Reconstruction of density and potential

Equations (3.23) and (3.24) can be used to reconstruct the mass density and potential from a given input mass density. The input density is generated using rejection method and obeys the Hernquist mass profile (3.12). The results can be used as a test of the code for bugs or gross inaccuracies. Figures 5.1 - 5.4 show the results of such a reconstruction procedure for $(n_{max} = 0, l_{max} = 0)$ and $(n_{max} = 5, l_{max} = 5)$ respectively.
The radial variable r in the input distribution has been softened by doing the replacement $r \to \sqrt{r^2 + \epsilon^2}$ with $\epsilon < 1$ to prevent the distribution from exploding at the origin. This softening accounts for the deviation between the two curves in figure 5.1 and especially 5.2.
As it should be apparent from the figures, the code succeeds to well approximate the input already at low order.

5.2 Non-perturbed Hernquist galaxy

Another test for the code is the computation of the dynamics of the non-perturbed Hernquist profile. The result expected for this type of initial density distribution would be a collapsing sphere that contracts to a point, then again expands and eventually settles down into a stable state. This sequence is reproduced by the simulation as can be seen from figures 5.5 to 5.8. The simulation shown in these figures has been performed with 2^{20} particles which were then downsampled to 2^{13} particles for plotting.

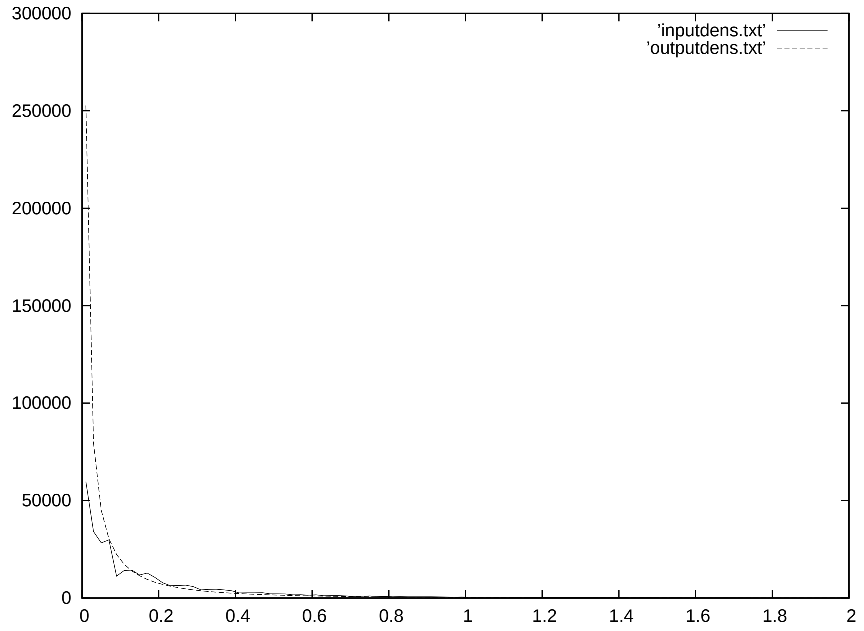

Figure 5.1: Reconstruction of the input mass density with $n_{max} = 0$ and $l_{max} = 0$. The dashed line indicates the output. The horizontal axis shows the radius r, the vertical axis the density $\rho(r)$.

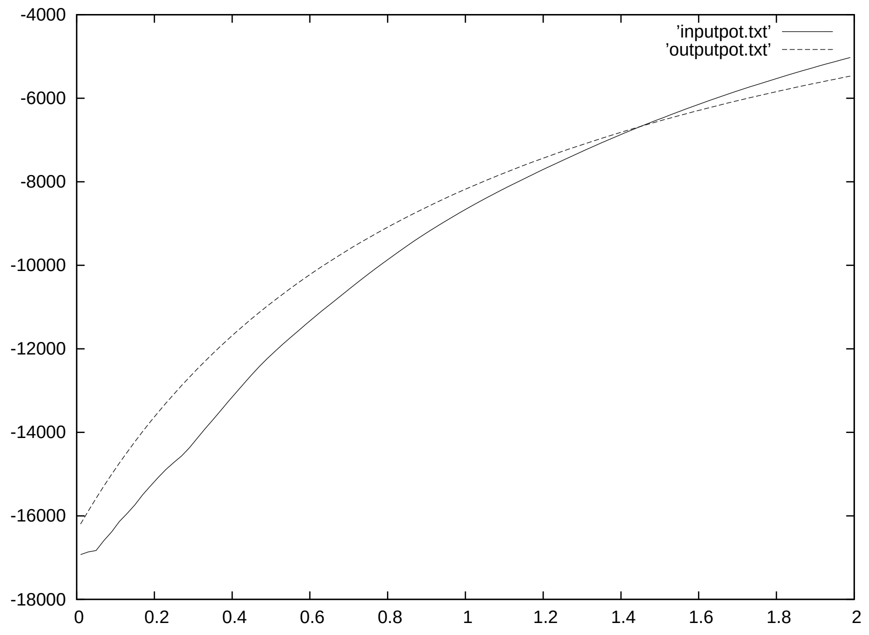

Figure 5.2: Reconstruction of the input potential with $n_{max} = 0$ and $l_{max} = 0$. The dashed line indicates the output. The horizontal axis shows the radius r, the vertical axis the potential $\Phi(r)$.

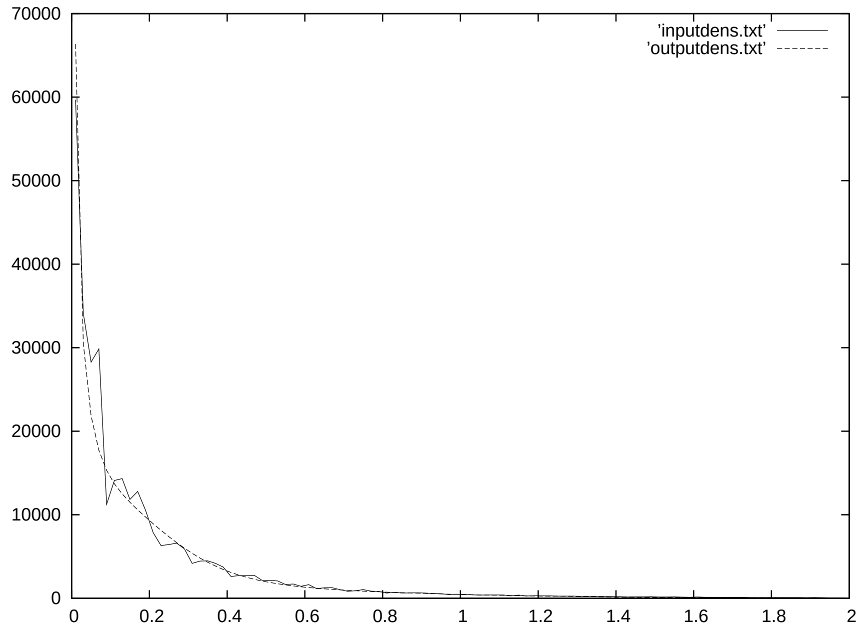

Figure 5.3: Reconstruction of the input mass density with $n_{max} = 5$ and $l_{max} = 5$. The dashed line indicates the output. The horizontal axis shows the radius r, the vertical axis the density $\rho(r)$.

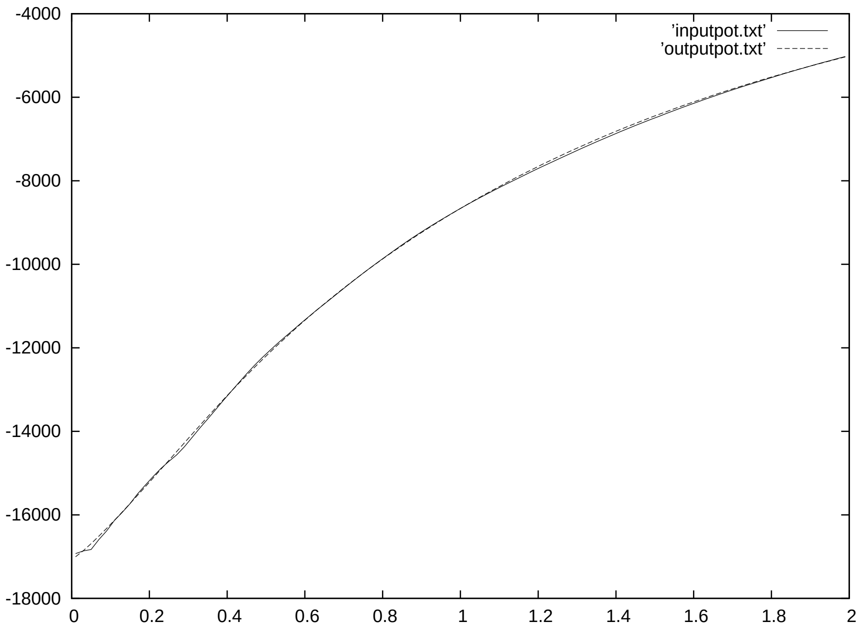

Figure 5.4: Reconstruction of the input potential with $n_{max} = 5$ and $l_{max} = 5$. The dashed line indicates the output. The horizontal axis shows the radius r, the vertical axis the potential $\Phi(r)$.

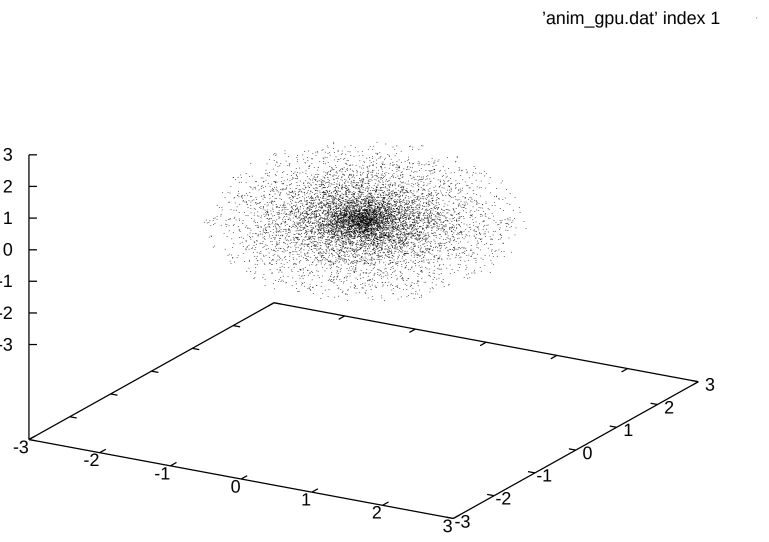

Figure 5.5: Start of the simulation at $t = 0$

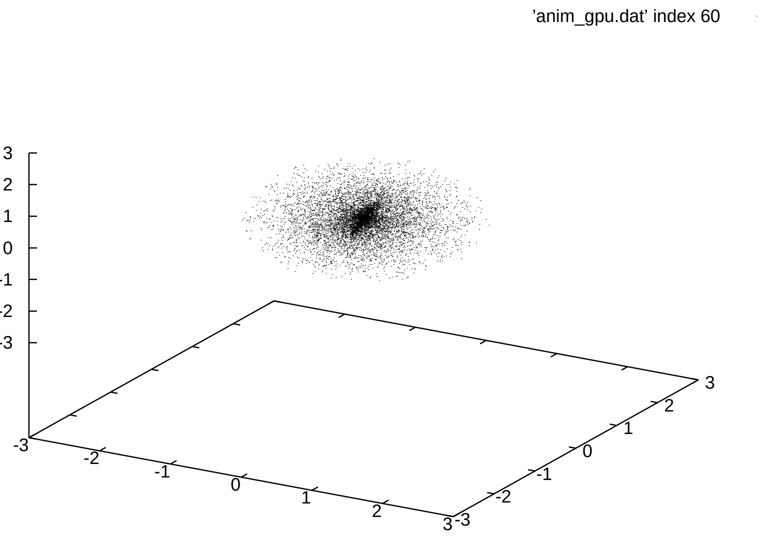

Figure 5.6: Collapse of the galaxy at $t = 0.002$

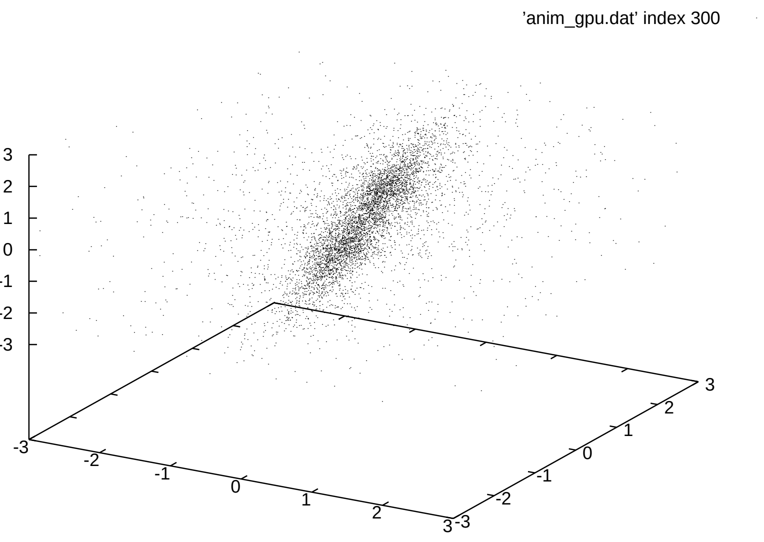

Figure 5.7: Expansion of the galaxy after the collapse at $t = 0.03$

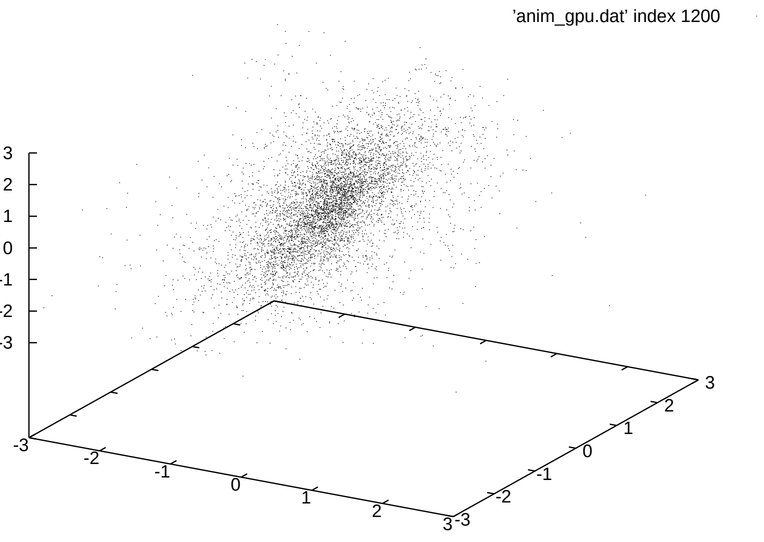

Figure 5.8: A stable state reached at $t = 0.12$

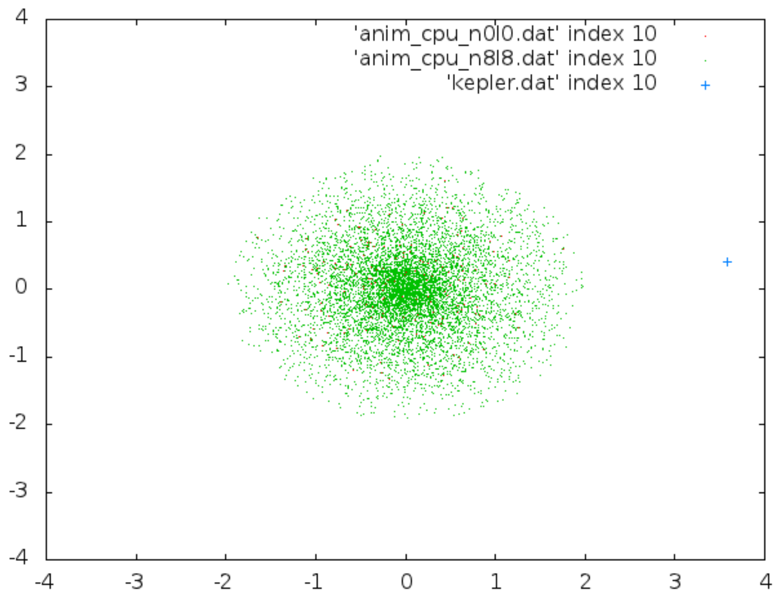

Figure 5.9: x-y plot of a superposition of perturbed Hernquist galaxies at $t = 0.001$. The red dots are the simulation with $(l_{max} = 0, n_{max} = 0)$ and the green dots are the simulation with $(l_{max} = 8, n_{max} = 8)$. The blue cross is the Kepler particle.

5.3 Perturbed Hernquist galaxy

As a stress-test for the algorithm, we can introduce a perturbation in the form of a massive particle orbiting on a Keplerian orbit around the galaxy. The orbit of the massive particle is computed semi-analytically using Keplers equations [15]:

$$r = a(1 - e\cos\eta) \tag{5.1}$$

$$\phi(r) = \arccos\left[\frac{a(1-e^2)}{re} - \frac{1}{e}\right] \tag{5.2}$$

with the parametrization

$$t(\eta) = \sqrt{\frac{a^3}{GM}}(\eta - e\sin\eta) \tag{5.3}$$

where a is the semi major axis and e the eccentricity of the orbit. Note that equation (5.3) has to be solved for η numerically (for example using the Netwon-Raphson method). The interaction between the "Kepler particle" and the galaxy is computed by direct summation.
The results for $(l_{max} = 0, n_{max} = 0)$ and $(l_{max} = 8, n_{max} = 8)$ with 8192 particles each have been superimposed in figures 5.9 to 5.11 to show the differences between a perturbed evolution with low order and high order expansion.
Figure 5.11 impressively shows the gross error in the dynamics caused by a truncation at low order.

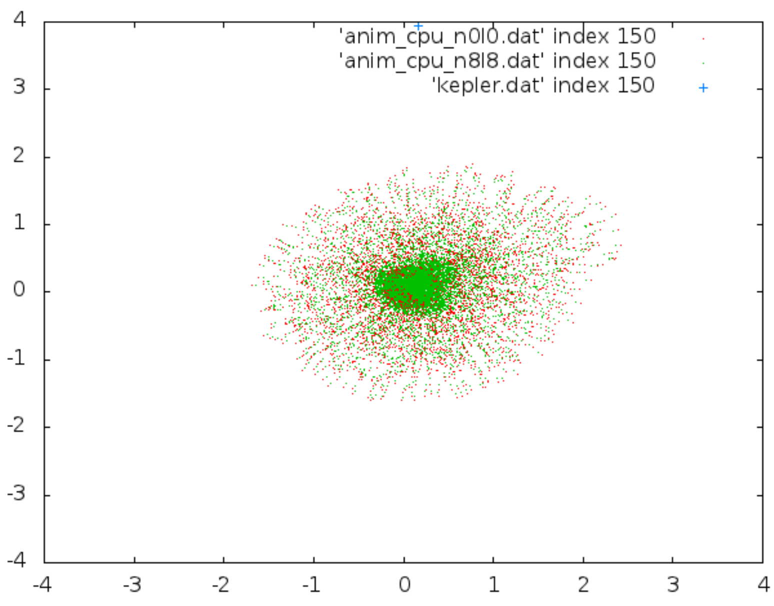

Figure 5.10: x-y plot of a superposition of perturbed Hernquist galaxies at $t = 0.015$. The red dots are the simulation with $(l_{max} = 0, n_{max} = 0)$ and the green dots are the simulation with $(l_{max} = 8, n_{max} = 8)$. The blue cross is the Kepler particle.

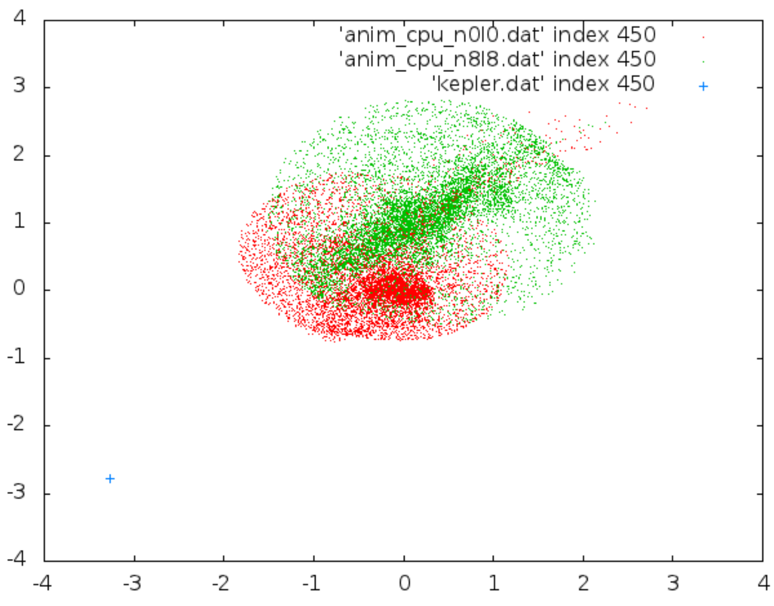

Figure 5.11: x-y plot of a superposition of perturbed Hernquist galaxies at $t = 0.045$. The red dots are the simulation with $(l_{max} = 0, n_{max} = 0)$ and the green dots are the simulation with $(l_{max} = 8, n_{max} = 8)$. The blue cross is the Kepler particle.

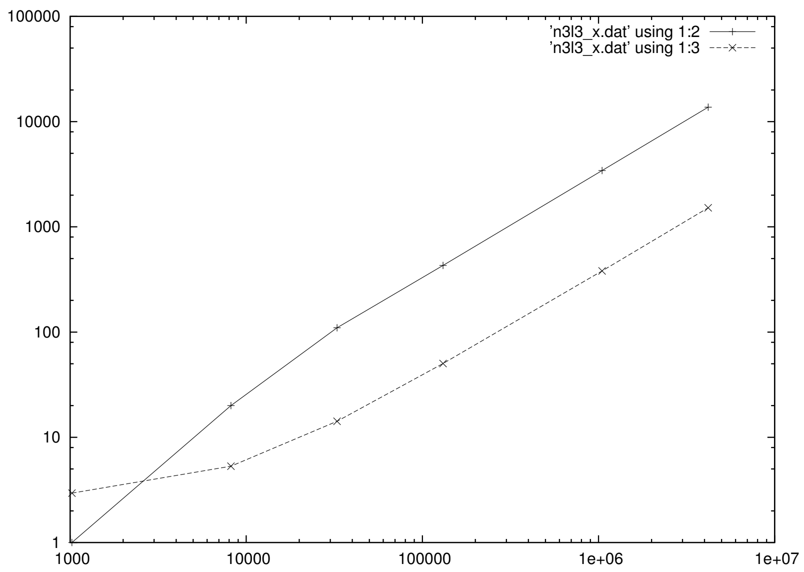

Figure 5.12: Loglog plot of the performance for the **X**-tabulation with ($n_{max} = 3, l_{max} = 3$). The horizontal axis shows the number of particles, the vertical axis shows the execution time in milliseconds. The dashed line is the result on the GPU, the continuous line the result on the CPU.

5.4 Performance

Several performance tests have been done to compare the timing on the CPU and the GPU, as can be seen in figures 5.12 to 5.15.

The timings have been performed with a Intel Xeon 2.40 GHz processor (using only one of the six available cores) and a GeForce GTX 590 with 512 CUDA cores and a clock rate of 1.23 GHz.
The computation time scales linearly with the number of particles, as already asserted.
For ($n_{max} = 3, l_{max} = 3$) and $N = 2^{20}$ particles, the **X**-tabulation code on the GPU is roughly 8.5 times faster than the CPU code. The computation of the acceleration shows a speedup by a factor of roughly 49. For ($n_{max} = 3, l_{max} = 4$), the results are similar.
The rather poor performance increase of the **X**-tabulation strongly indicates a suboptimal programming of the corresponding CUDA kernel which motivates further investigation to detect the performance sinkhole(s).

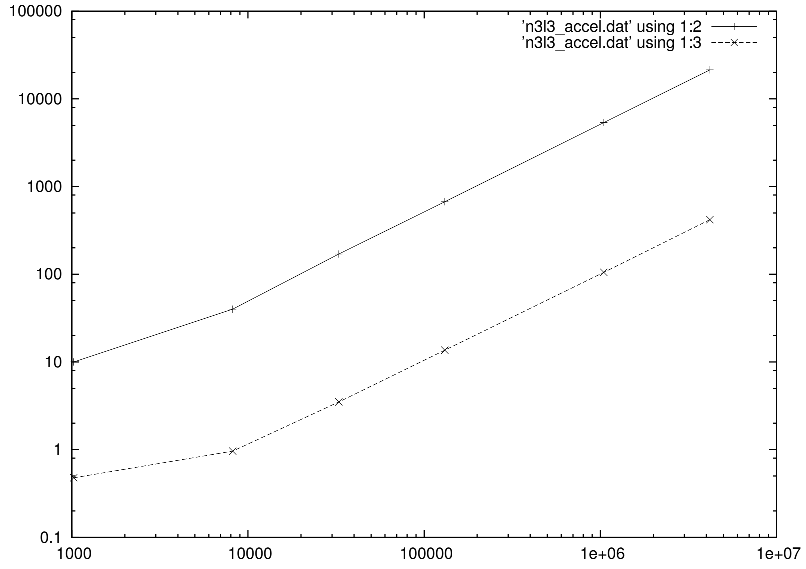

Figure 5.13: Loglog plot of the performance for the acceleration computation with $(n_{max} = 3, l_{max} = 3)$. The horizontal axis shows the number of particles, the vertical axis shows the execution time in milliseconds. The dashed line is the result on the GPU, the continuous line the result on the CPU.

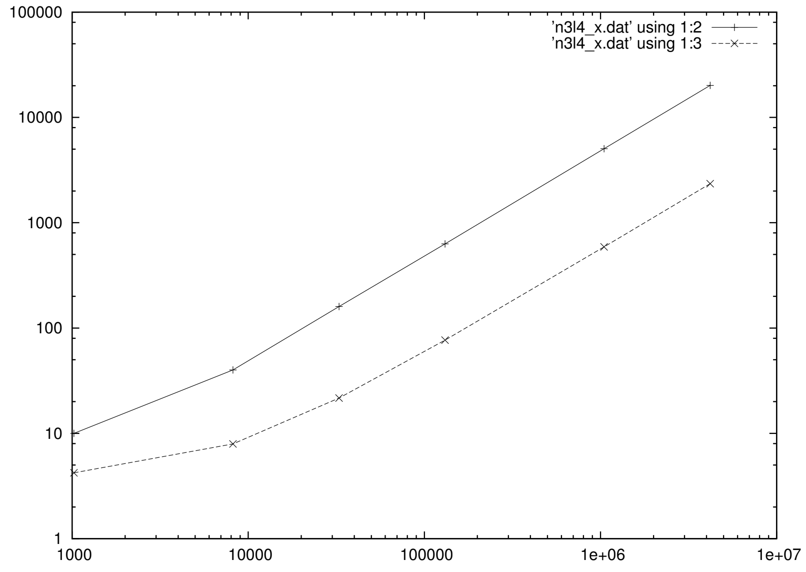

Figure 5.14: Loglog plot of the performance for the **X**-tabulation with $(n_{max} = 3, l_{max} = 4)$. The horizontal axis shows the number of particles, the vertical axis shows the execution time in milliseconds. The dashed line is the result on the GPU, the continuous line the result on the CPU.

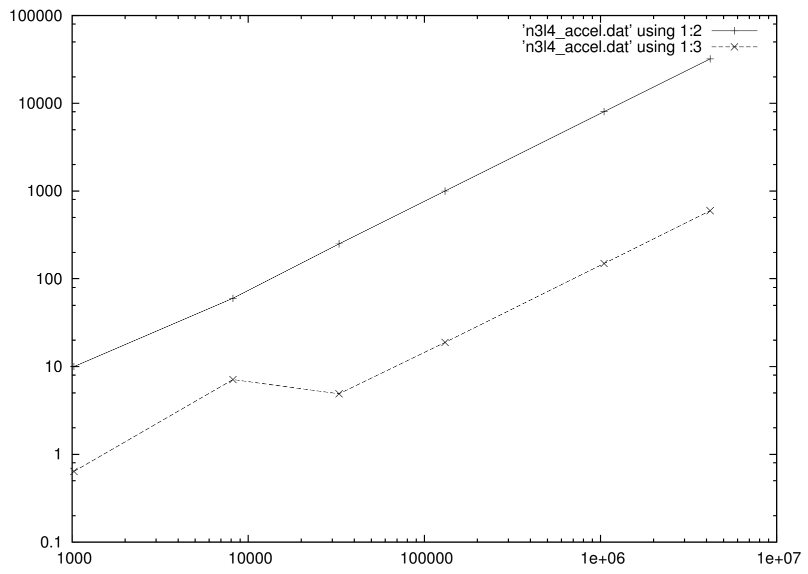

Figure 5.15: Loglog plot of the performance for the acceleration computation with ($n_{max} = 3, l_{max} = 4$). The horizontal axis shows the number of particles, the vertical axis shows the execution time in milliseconds. The dashed line is the result on the GPU, the continuous line the result on the CPU.

Chapter 6

Conclusions

In this thesis, I presented a partial implementation of the treecode with a fully working treebuild and a executable treewalk which can walk the tree structure but does not yet use multipole expansions (i.e. in the current form, the treewalk performs a direction summation by setting $r_{open} = \infty$). I abandoned the project of porting the treecode onto the GPU since it turned out that the complexity of such a task is beyond the scope of a bachelor thesis [8].
Instead, I turned my attentation to the GPU implementation of the SCF code which I successfully implemented on CUDA.
The speedup of this novel code is about a factor of 8.5 for the **X**-table and a factor of 49 for the acceleration computation. The poor speedup of the **X**-table computation motivates an analysis of the corresponding CUDA kernel to find the performance sinkholes.
Note that thanks to several optimizations (unrolling recursion relations, tabulating intermediate values et cetera), already the CPU code is able to compute over 10^6 particles within a sensible period of time.
A disadvantage of the SCF code is its demand for large amounts of shared memory which enforces the truncation of the expansion already at relatively low order $((n_{max} = 3, l_{max} = 4))$. It would be desirable to write an extension for my code which circumvents this problem.
The SCF code has proven to be very fast and easily parallelizable. Further work could be invested in implementing more general galaxy types in the GPU code with methods presented for example in [4], [5].

Acknowledgements

I would like to thank my supervisor Dr. Joachim Stadel for his constant and highly motivated support during the development of the program, his help with the numerous problems that arose during my work and his ability to keep my motivation at a high level.
Furthermore I would like to thank Dr. Prasenjit Saha from whose knowledge I could profit in the form of a number of valuable pieces of advice.
Finally, I thank all my fellow students and my colleagues in the office for a good time during the development of my thesis. Thank you all!

Sourcecode

The sourcecode to this thesis is available from my webpage
`http://www.physik.uzh.ch/~simons/bacode`
There is no documentation for the program but upon request I can give advice about compiling and using the code.
If there is interest, I will consider modifying and cleaning the code for easier understanding and usage.

Bibliography

[1] Lars Hernquist and Jeremiah P. Ostriker, A self-consistent field code for galactic dynamics, The Astrophysical Journal , 1992.

[2] Lars Hernquist and Stein Sigurdsson, A parallel self-consistent field code, Astrophysical Journal v.446, p.717, 1995.

[3] Prasenjit Saha, Designer basis functions for potentials in galactic dynamics, Mon. Not. R. Astron. Soc., 1993.

[4] Martin D. Weinberg, High-Accuracy minimum relaxation N-Body simulations using orthogonal series force computation, Astrophysical Journal, 1996

[5] Martin D. Weinberg, An adaptive algorithm for N-Body field expansions, The Astronomical Journal, 1999

[6] Joachim Gerhard Stadel, Cosmological N-body Simulations and their Analysis, Dissertation, 2001

[7] John Dubinski, A parallel tree code, New Astronomy, 1996

[8] Gaburov E. and Bédorf J. and Zwart S.P.,Gravitational tree-code on graphics processing units: implementation in CUDA, Elsevier, 2010

[9] Bédorf J. and Gaburov, E. and Zwart S.P., A sparse octree gravitational N-body code that runs entirely on the GPU processor, Arxiv preprint arXiv:1106.1900, 2011

[10] Dehnen W., A hierarchical O(n) force calculation algorithm, Journal of Computational Physics, Elsevier, 2002

[11] Hamada T. and Nitadori K. and Benkrid K. and Ohno Y. and Morimoto G. and Masada T. and Shibata Y. and Oguri K. and Taiji, M., A novel multiple-walk parallel algorithm for the Barnes–Hut treecode on GPUs–towards cost effective, high performance N-body simulation, Computer Science-Research and Development, Springer, 2009

[12] Mark Harris, Optimizing parallel reduction in CUDA, Educational slides

[13] NVIDIA C., C programming best practices guide, Cuda SDK, 2009

[14] Nvidia C., Compute Unified Device Architecture Programming Guide, NVIDIA: Santa Clara CA, 2007

[15] Prasenjit Saha, Introduction to Astrophysics, Lecture Script, 2010

[16] Renato Pajarola, Algorithmen und Datenstrukturen, Lecture Slides, 2010

[17] Sanders J. and Kandrot E., CUDA by example: an introduction to general-purpose GPU programming, Addison-Wesley Professional, 2010

[18] Press W.H. and Teukolsky S.A. and Vetterling W.T. and Flannery B.P., Numerical recipes 3rd edition: The art of scientific computing, Cambridge University Press, 2007

[19] Arfken G.B. and Weber H.J. and Weber H., Mathematical methods for physicists, Academic press New York, 1995